Abstract

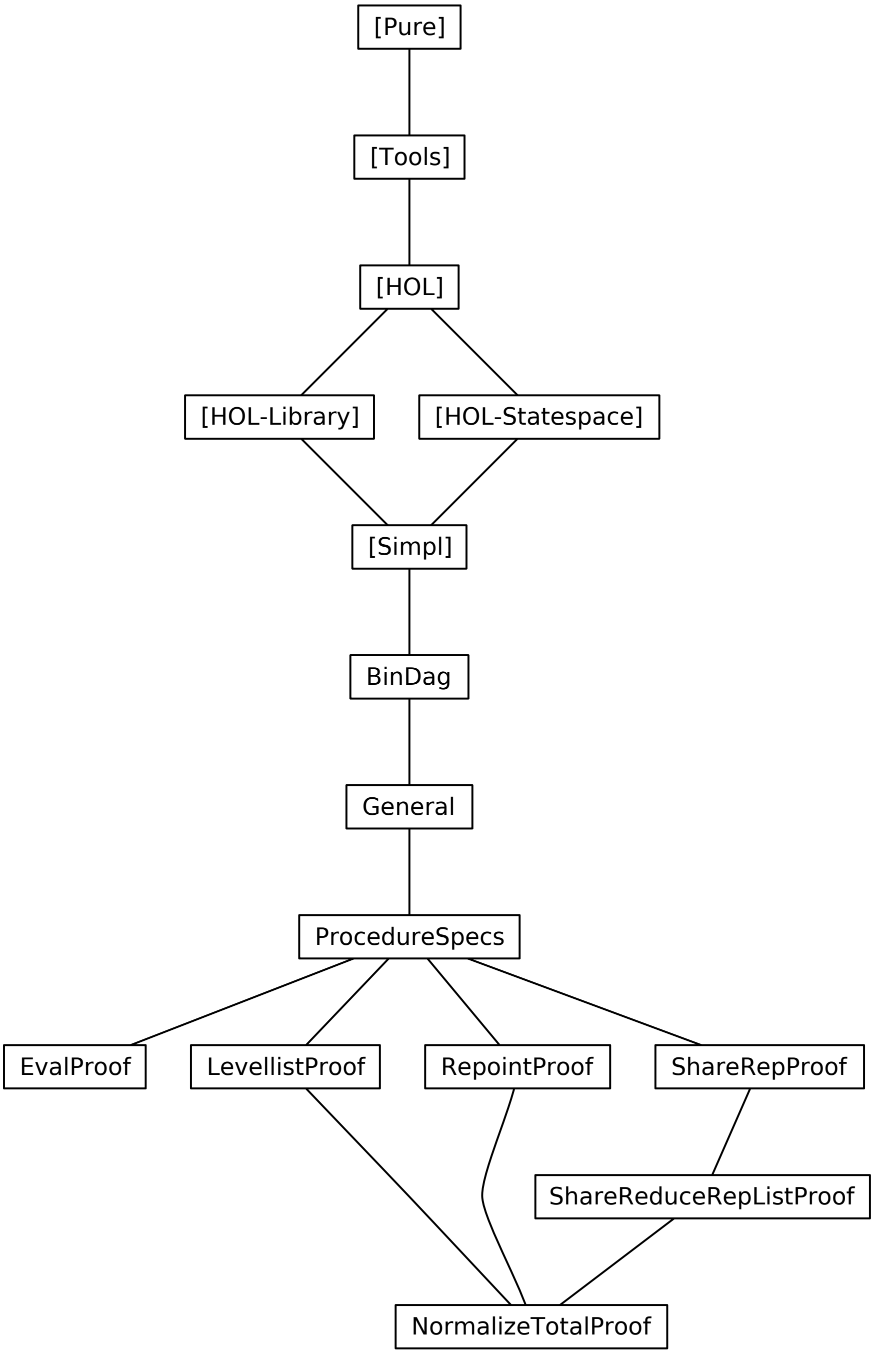
We present the verification of the normalisation of a binary decision diagram (BDD). The normalisation follows the original algorithm presented by Bryant in 1986 and transforms an ordered BDD in a reduced, ordered and shared BDD. The verification is based on Hoare logics.

BDD-Normalisation

Veronika Ortner and Norbert Schirmer

March 17, 2025

Contents


[Pure]
[Tools]
[HOL]
[HOL-Library]
[HOL-Statespace]
[Simpl]
BinDag
General
ProcedureSpecs
EvalProof
LevellistProof
RepointProof
ShareRepProof
ShareReduceRepListProof
NormalizeTotalProof

1 Introduction

In [1] we describe the partial correctness proofs for BDD normalisation. We extend this work to total correctness in these theories.

2 BDD Abstractions

theory *BinDag*
imports *Simpl.Simpl-Heap*
begin

datatype *dag* = *Tip* | *Node dag ref dag*

lemma [*simp*]: *Node lt a rt* ≠ *lt*
  by (*induct lt*) *auto*

lemma [*simp*]: *lt* ≠ *Node lt a rt*
  by (*induct lt*) *auto*

lemma [*simp*]: *Node lt a rt* ≠ *rt*
  by (*induct rt*) *auto*

lemma [*simp*]: *rt* ≠ *Node lt a rt*
  by (*induct rt*) *auto*

primrec *set-of*:: *dag* ⇒ *ref set* **where**
  set-of-Tip: *set-of Tip* = {}
  | *set-of-Node*: *set-of* (*Node lt a rt*) = {*a*} ∪ *set-of lt* ∪ *set-of rt*

primrec *DAG*:: *dag* ⇒ *bool* **where**
  DAG Tip = *True*
  | *DAG* (*Node l a r*) = (*a* ∉ *set-of l* ∧ *a* ∉ *set-of r* ∧ *DAG l* ∧ *DAG r*)

primrec *subdag*:: *dag* ⇒ *dag* ⇒ *bool* **where**
  subdag Tip t = *False*
  | *subdag* (*Node l a r*) *t* = (*t*=*l* ∨ *t*=*r* ∨ *subdag l t* ∨ *subdag r t*)

lemma *subdag-size*: *subdag t s* ⟹ *size s* < *size t*
  by (*induct t*) *auto*

lemma *subdag-neq*: *subdag t s* ⟹ *t*≠*s*
by (*induct t*) (*auto dest*: *subdag-size*)

lemma *subdag-trans* [*trans*]: *subdag t s* ⟹ *subdag s r* ⟹ *subdag t r*
by (*induct t*) *auto*

lemma *subdag-NodeD*:

```
  subdag t (Node lt a rt) ⟹ subdag t lt ∧ subdag t rt
  by (induct t) auto

lemma subdag-not-sym: ⋀t. ⟦subdag s t; subdag t s⟧ ⟹ P
  by (induct s) (auto dest: subdag-NodeD)

instantiation dag :: order
begin

definition
  less-dag-def: s < (t::dag) ⟷ subdag t s

definition
  le-dag-def: s ≤ (t::dag) ⟷ s=t ∨ s < t

lemma le-dag-refl: (x::dag) ≤ x
  by (simp add: le-dag-def)

lemma le-dag-trans:
  fixes x::dag and y and z
  assumes x-y: x ≤ y and y-z: y ≤ z
  shows x ≤ z
proof (cases x=y)
  case True with y-z show ?thesis by simp
next
  case False
  note x-neq-y = this
  with x-y have x-less-y: x < y by (simp add: le-dag-def)
  show ?thesis
  proof (cases y=z)
    case True
    with x-y show ?thesis by simp
  next
    case False
    with y-z have y < z by (simp add: le-dag-def)
    with x-less-y have x < z
      by (auto simp add: less-dag-def intro: subdag-trans)
    thus ?thesis
      by (simp add: le-dag-def)
  qed
qed

lemma le-dag-antisym:
  fixes x::dag and y
  assumes x-y: x ≤ y and y-x: y ≤ x
  shows x = y
proof (cases x=y)
  case True thus ?thesis .
next
```

case *False*
with *x-y y-x* **show** *?thesis*
by (*auto simp add*: *less-dag-def le-dag-def intro*: *subdag-not-sym*)
qed

lemma *dag-less-le*:
fixes *x*::*dag* **and** *y*
shows $(x < y) = (x \leq y \wedge x \neq y)$
by (*auto simp add*: *less-dag-def le-dag-def dest*: *subdag-neq*)

instance
by *standard* (*auto simp add*: *dag-less-le le-dag-refl intro*: *le-dag-trans dest*: *le-dag-antisym*)

end

lemma *less-dag-Tip* [*simp*]: $\neg$ ($x <$ *Tip*)
by (*simp add*: *less-dag-def*)

lemma *less-dag-Node*: $x <$ (*Node l a r*) =
$(x \leq l \vee x \leq r)$
by (*auto simp add*: *order-le-less less-dag-def*)

lemma *less-dag-Node'*: $x <$ (*Node l a r*) =
$(x{=}l \vee x{=}r \vee x < l \vee x < r)$
by (*simp add*: *less-dag-def*)

lemma *less-Node-dag*: (*Node l a r*) $< x \Longrightarrow l < x \wedge r < x$
by (*auto simp add*: *less-dag-def dest*: *subdag-NodeD*)

lemma *less-dag-set-of*: $x < y \Longrightarrow$ *set-of x* $\subseteq$ *set-of y*
by (*unfold less-dag-def*, *induct y*, *auto*)

lemma *le-dag-set-of*: $x \leq y \Longrightarrow$ *set-of x* $\subseteq$ *set-of y*
apply (*unfold le-dag-def*)
apply (*erule disjE*)
apply *simp*
apply (*erule less-dag-set-of*)
done

lemma *DAG-less*: *DAG y* $\Longrightarrow x < y \Longrightarrow$ *DAG x*
by (*induct y*) (*auto simp add*: *less-dag-Node'*)

lemma *less-DAG-set-of*:
assumes *x-less-y*: $x < y$
assumes *DAG-y*: *DAG y*
shows *set-of x* $\subset$ *set-of y*
proof (*cases y*)
case *Tip* **with** *x-less-y* **show** *?thesis* **by** *simp*
next

```
  case (Node l a r)
  with DAG-y obtain a: a ∉ set-of l a ∉ set-of r
    by simp
  from Node obtain l-less-y: l < y and r-less-y: r < y
    by (simp add: less-dag-Node)
  from Node a obtain
    l-subset-y: set-of l ⊂ set-of y and
    r-subset-y: set-of r ⊂ set-of y
    by auto
  from Node x-less-y have x=l ∨ x=r ∨ x < l ∨ x < r
    by (simp add: less-dag-Node')
  thus ?thesis
  proof (elim disjE)
    assume x=l
    with l-subset-y show ?thesis by simp
  next
    assume x=r
    with r-subset-y show ?thesis by simp
  next
    assume x < l
    hence set-of x ⊆ set-of l
      by (rule less-dag-set-of)
    also note l-subset-y
    finally show ?thesis .
  next
    assume x < r
    hence set-of x ⊆ set-of r
      by (rule less-dag-set-of)
    also note r-subset-y
    finally show ?thesis .
  qed
qed


lemma in-set-of-decomp:
  p ∈ set-of t = (∃ l r. t=(Node l p r) ∨ subdag t (Node l p r))
  (is ?A = ?B)
proof
  assume ?A thus ?B
    by (induct t) auto
next
  assume ?B thus ?A
    by (induct t) auto
qed

primrec Dag:: ref ⇒ (ref ⇒ ref) ⇒ (ref ⇒ ref) ⇒ dag ⇒ bool
where
Dag p l r Tip = (p = Null) |
Dag p l r (Node lt a rt) = (p = a ∧ p ≠ Null ∧
```

$Dag\ (l\ p)\ l\ r\ lt \wedge Dag\ (r\ p)\ l\ r\ \ rt)$

lemma *Dag-Null* [*simp*]: $Dag\ Null\ l\ r\ \ t = (t = Tip)$
by (*cases t*) *simp-all*

lemma *Dag-Ref* [*simp*]:
$p \neq Null \Longrightarrow Dag\ p\ l\ r\ \ t = (\exists\, lt\ rt.\ t{=}Node\ lt\ p\ rt \wedge$
$Dag\ (l\ p)\ l\ r\ lt \wedge Dag\ (r\ p)\ l\ r\ rt)$
by (*cases t*) *auto*

lemma *Null-notin-Dag* [*simp*, *intro*]: $\bigwedge p\ l\ r.\ Dag\ p\ l\ r\ t \Longrightarrow Null \notin set\text{-}of\ t$
by (*induct t*) *auto*

theorem *notin-Dag-update-l* [*simp*]:
$\bigwedge\ p.\ q \notin set\text{-}of\ t \Longrightarrow Dag\ p\ (l(q := y))\ r\ \ t = Dag\ p\ l\ r\ t$
by (*induct t*) *auto*

theorem *notin-Dag-update-r* [*simp*]:
$\bigwedge\ p.\ q \notin set\text{-}of\ t \Longrightarrow Dag\ p\ l\ (r(q := y))\ t = Dag\ p\ l\ r\ t$
by (*induct t*) *auto*

lemma *Dag-upd-same-l-lemma*: $\bigwedge p.\ p \neq Null \Longrightarrow \neg\ Dag\ p\ (l(p{:=}p))\ r\ t$
apply (*induct t*)
apply *simp*
apply (*simp* (*no-asm-simp*) *del*: *fun-upd-apply*)
apply (*simp* (*no-asm-simp*) *only*: *fun-upd-apply refl if-True*)
apply *blast*
done

lemma *Dag-upd-same-l* [*simp*]: $Dag\ p\ (l(p{:=}p))\ r\ t = (p{=}Null \wedge t{=}Tip)$
apply (*cases p=Null*)
apply *simp*
apply (*fast dest*: *Dag-upd-same-l-lemma*)
done

Dag-upd-same-l prevents $p \neq Null \Longrightarrow Dag\ p\ (l(p := p))\ r\ t = X$ from looping, because of *Dag-Ref* and *fun-upd-apply*.

lemma *Dag-upd-same-r-lemma*: $\bigwedge p.\ p \neq Null \Longrightarrow \neg\ Dag\ p\ l\ (r(p{:=}p))\ t$
apply (*induct t*)
apply *simp*
apply (*simp* (*no-asm-simp*) *del*: *fun-upd-apply*)
apply (*simp* (*no-asm-simp*) *only*: *fun-upd-apply refl if-True*)
apply *blast*
done

lemma *Dag-upd-same-r* [*simp*]: $Dag\ p\ l\ (r(p{:=}p))\ t = (p{=}Null \wedge t{=}Tip)$
apply (*cases p=Null*)
apply *simp*

```
  apply (fast dest: Dag-upd-same-r-lemma)
  done

lemma  Dag-update-l-new [simp]: ⟦set-of t ⊆ set alloc⟧
     ⟹ Dag p (l(new (set alloc) := x)) r t = Dag p l r t
  by (rule notin-Dag-update-l) fastforce

lemma  Dag-update-r-new [simp]: ⟦set-of t ⊆ set alloc⟧
     ⟹ Dag p l (r(new (set alloc) := x)) t = Dag p l r t
  by (rule notin-Dag-update-r) fastforce

lemma Dag-update-lI [intro]:
    ⟦Dag p l r t; q ∉ set-of t⟧ ⟹ Dag p (l(q := y)) r t
  by simp

lemma Dag-update-rI [intro]:
    ⟦Dag p l r t; q ∉ set-of t⟧ ⟹ Dag p l (r(q := y)) t
  by simp

lemma Dag-unique: ⋀ p t2. Dag p l r t1 ⟹ Dag p l r t2 ⟹ t1=t2
  by (induct t1) auto

lemma Dag-unique1: Dag p l r t ⟹ ∃!t. Dag p l r t
  by (blast intro: Dag-unique)

lemma Dag-subdag: ⋀ p. Dag p l r t ⟹ subdag t s ⟹ ∃ q. Dag q l r s
  by (induct t) auto

lemma Dag-root-not-in-subdag-l [simp,intro]:
  assumes Dag (l p) l r t
  shows p ∉ set-of t
proof −
  {
    fix lt rt
    assume t: t = Node lt p rt
    from assms have Dag (l p) l r lt
      by (clarsimp simp only: t Dag.simps)
    with assms have t=lt
      by (rule Dag-unique)
    with t have False
      by simp
  }
  moreover
  {
    fix lt rt
    assume subdag: subdag t (Node lt p rt)
    with assms obtain q where Dag q l r (Node lt p rt)
      by (rule Dag-subdag [elim-format]) iprover
    hence Dag (l p) l r lt
```

```
      by auto
    with assms have t=lt
      by (rule Dag-unique)
    moreover
    have subdag t lt
    proof -
      note subdag
      also have subdag (Node lt p rt) lt by simp
      finally show ?thesis .
    qed
    ultimately have False
      by (simp add: subdag-neq)
  }
  ultimately show ?thesis
    by (auto simp add: in-set-of-decomp)
qed

lemma Dag-root-not-in-subdag-r [simp, intro]:
  assumes Dag (r p) l r t
  shows p ∉ set-of t
proof -
  {
    fix lt rt
    assume t: t = Node lt p rt
    from assms have Dag (r p) l r rt
      by (clarsimp simp only: t Dag.simps)
    with assms have t=rt
      by (rule Dag-unique)
    with t have False
      by simp
  }
  moreover
  {
    fix lt rt
    assume subdag: subdag t (Node lt p rt)
    with assms obtain q where Dag q l r (Node lt p rt)
      by (rule Dag-subdag [elim-format]) iprover
    hence Dag (r p) l r rt
      by auto
    with assms have t=rt
      by (rule Dag-unique)
    moreover
    have subdag t rt
    proof -
      note subdag
      also have subdag (Node lt p rt) rt by simp
      finally show ?thesis .
    qed
    ultimately have False
```

by (*simp add*: *subdag-neq*)
}
ultimately show *?thesis*
by (*auto simp add*: *in-set-of-decomp*)
qed

lemma *Dag-is-DAG*: $\bigwedge p\ l\ r.\ Dag\ p\ l\ r\ t \Longrightarrow DAG\ t$
by (*induct t*) *auto*

lemma *heaps-eq-Dag-eq*:
$\bigwedge p.\ \forall x \in set\text{-}of\ t.\ l\ x = l'\ x \wedge r\ x = r'\ x$
$\Longrightarrow Dag\ p\ l\ r\ t = Dag\ p\ l'\ r'\ t$
by (*induct t*) *auto*

lemma *heaps-eq-DagI1*:
$[\![Dag\ p\ l\ r\ t;\ \forall x{\in}set\text{-}of\ t.\ l\ x = l'\ x \wedge r\ x = r'\ x]\!]$
$\Longrightarrow Dag\ p\ l'\ r'\ t$
by (*rule heaps-eq-Dag-eq* [*THEN iffD1*])

lemma *heaps-eq-DagI2*:
$[\![Dag\ p\ l'\ r'\ t;\ \forall x{\in}set\text{-}of\ t.\ l\ x = l'\ x \wedge r\ x = r'\ x]\!]$
$\Longrightarrow Dag\ p\ l\ r\ t$
by (*rule heaps-eq-Dag-eq* [*THEN iffD2*]) *auto*

lemma *Dag-unique-all-impl-simp* [*simp*]:
$Dag\ p\ l\ r\ t \Longrightarrow (\forall t.\ Dag\ p\ l\ r\ t \longrightarrow P\ t) = P\ t$
by (*auto dest*: *Dag-unique*)

lemma *Dag-unique-ex-conj-simp* [*simp*]:
$Dag\ p\ l\ r\ t \Longrightarrow (\exists t.\ Dag\ p\ l\ r\ t \wedge P\ t) = P\ t$
by (*auto dest*: *Dag-unique*)

lemma *Dags-eq-hp-eq*:
$\bigwedge p\ p'.\ [\![Dag\ p\ l\ r\ t;\ Dag\ p'\ l'\ r'\ t]\!] \Longrightarrow$
$p'{=}p \wedge (\forall no \in set\text{-}of\ t.\ l'\ no = l\ no \wedge r'\ no = r\ no)$
by (*induct t*) *auto*

definition *isDag* :: $ref \Rightarrow (ref \Rightarrow ref) \Rightarrow (ref \Rightarrow ref) \Rightarrow bool$
where $isDag\ p\ l\ r = (\exists t.\ Dag\ p\ l\ r\ t)$

definition *dag* :: $ref \Rightarrow (ref \Rightarrow ref) \Rightarrow (ref \Rightarrow ref) \Rightarrow dag$
where $dag\ p\ l\ r = (THE\ t.\ Dag\ p\ l\ r\ t)$

lemma *Dag-conv-isDag-dag*: $Dag\ p\ l\ r\ t = (isDag\ p\ l\ r \wedge t{=}dag\ p\ l\ r)$
apply (*simp add*: *isDag-def dag-def*)
apply (*rule iffI*)
apply (*rule conjI*)
apply *blast*

apply (*subst the1-equality*)
apply (*erule Dag-unique1*)
apply *assumption*
apply (*rule refl*)
apply *clarify*
apply (*rule theI*)
apply *assumption*
apply (*erule* (*1*) *Dag-unique*)
done

lemma *Dag-dag*: *Dag p l r t* $\Longrightarrow$ *dag p l r* = *t*
by (*simp add*: *Dag-conv-isDag-dag*)

end

3 General Lemmas on BDD Abstractions

theory *General* **imports** *BinDag* **begin**

definition *subdag-eq*:: *dag* ⇒ *dag* ⇒ *bool* **where**
subdag-eq t_1 t_2 = (t_1 = t_2 ∨ *subdag* t_1 t_2)

primrec *root* :: *dag* ⇒ *ref*
where
root Tip = *Null* |
root (*Node l a r*) = *a*

fun *isLeaf* :: *dag* ⇒ *bool* **where**
isLeaf Tip = *False*
| *isLeaf* (*Node Tip v Tip*) = *True*
| *isLeaf* (*Node* (*Node l* v_1 *r*) v_2 *Tip*) = *False*
| *isLeaf* (*Node Tip* v_1 (*Node l* v_2 *r*)) = *False*

datatype *bdt* = *Zero* | *One* | *Bdt-Node bdt nat bdt*

fun *bdt-fn* :: *dag* ⇒ (*ref* ⇒ *nat*) ⇒ *bdt option* **where**
bdt-fn Tip = (λ*bdtvar* . *None*)
| *bdt-fn* (*Node Tip vref Tip*) =
(λ*bdtvar* .
(*if* (*bdtvar vref* = *0*)
then Some Zero
else (*if* (*bdtvar vref* = *1*)
then Some One
else None)))
| *bdt-fn* (*Node Tip vref* (*Node l vref1 r*)) = (λ*bdtvar* . *None*)
| *bdt-fn* (*Node* (*Node l vref1 r*) *vref Tip*) = (λ*bdtvar* . *None*)
| *bdt-fn* (*Node* (*Node l1 vref1 r1*) *vref* (*Node l2 vref2 r2*)) =
(λ*bdtvar* .

```
  (if (bdtvar vref = 0 ∨ bdtvar vref = 1)
   then None
   else
    (case (bdt-fn (Node l1 vref1 r1) bdtvar) of
     None ⇒ None
     |(Some b1) ⇒
       (case (bdt-fn (Node l2 vref2 r2) bdtvar) of
        None ⇒ None
       |(Some b2) ⇒ Some (Bdt-Node b1 (bdtvar vref) b2)))))
```

abbreviation *bdt == bdt-fn*

```
primrec eval :: bdt ⇒ bool list ⇒ bool
where
eval Zero env = False |
eval One env  = True |
eval (Bdt-Node l v r) env  = (if (env ! v) then eval r env else eval l env)
```

```
fun ordered-bdt:: bdt ⇒ bool where
ordered-bdt Zero = True
| ordered-bdt One = True
| ordered-bdt (Bdt-Node (Bdt-Node l1 v1 r1) v (Bdt-Node l2 v2 r2)) =
    ((v1 < v) ∧ (v2 < v) ∧
     (ordered-bdt (Bdt-Node l1 v1 r1)) ∧ (ordered-bdt (Bdt-Node l2 v2 r2)))
| ordered-bdt (Bdt-Node (Bdt-Node l1 v1 r1) v r) =
    ((v1 < v) ∧ (ordered-bdt (Bdt-Node l1 v1 r1)))
| ordered-bdt (Bdt-Node l v (Bdt-Node l2 v2 r2)) =
    ((v2 < v) ∧ (ordered-bdt (Bdt-Node l2 v2 r2)))
| ordered-bdt (Bdt-Node l v r) = True
```

```
fun ordered:: dag ⇒ (ref⇒nat) ⇒ bool where
ordered Tip = (λ var. True)
| ordered (Node (Node l_1 v_1 r_1) v (Node l_2 v_2 r_2)) =
    (λ var. (var v_1 < var v ∧ var v_2 < var v) ∧
        (ordered (Node l_1 v_1 r_1) var) ∧ (ordered (Node l_2 v_2 r_2) var))
| ordered (Node Tip v Tip) = (λ var. (True))
| ordered (Node Tip v r) =
     (λ var. (var (root r) < var v) ∧ (ordered r var))
| ordered (Node l v Tip) =
     (λ var. (var (root l) < var v) ∧ (ordered l var))
```

```
primrec max-var :: bdt ⇒ nat
where
max-var Zero = 0 |
```

max-var One = *1* |
max-var (*Bdt-Node l v r*) = *max v* (*max* (*max-var l*) (*max-var r*))

lemma *eval-zero*: ⟦*bdt* (*Node l v r*) *var* = *Some x*;
var (*root* (*Node l v r*)) = (*0*::*nat*)⟧ ⟹ *x* = *Zero*
apply (*cases l*)
apply (*cases r*)
apply *simp*
apply *simp*
apply (*cases r*)
apply *simp*
apply *simp*
done

lemma *bdt-Some-One-iff* [*simp*]:
(*bdt t var* = *Some One*) = (∃ *p*. *t* = *Node Tip p Tip* ∧ *var p* = *1*)
apply (*induct t rule*: *bdt-fn.induct*)
apply (*auto split*: *option.splits*)
done

lemma *bdt-Some-Zero-iff* [*simp*]:
(*bdt t var* = *Some Zero*) = (∃ *p*. *t* = *Node Tip p Tip* ∧ *var p* = *0*)
apply (*induct t rule*: *bdt-fn.induct*)
apply (*auto split*: *option.splits*)
done

lemma *bdt-Some-Node-iff* [*simp*]:
(*bdt t var* = *Some* (*Bdt-Node bdt1 v bdt2*)) =
(∃ *p l r*. *t* = *Node l p r* ∧ *bdt l var* = *Some bdt1* ∧ *bdt r var* = *Some bdt2* ∧
1 < *v* ∧ *var p* = *v*)
apply (*induct t rule*: *bdt-fn.induct*)
prefer *5*
apply (*fastforce split*: *if-splits option.splits*)
apply *auto*
done

lemma *balanced-bdt*:
⋀ *p bdt1*. ⟦ *Dag p low high t*; *bdt t var* = *Some bdt1*; *no* ∈ *set-of t*⟧
⟹ (*low no* = *Null*) = (*high no* = *Null*)
proof (*induct t*)
case *Tip*
then show *?case* **by** *simp*
next
case (*Node lt a rt*)
note *NN*= *this*
have *bdt1*: *bdt* (*Node lt a rt*) *var* = *Some bdt1* **by** *fact*
have *no-in-t*: *no* ∈ *set-of* (*Node lt a rt*) **by** *fact*
have *p-tree*: *Dag p low high* (*Node lt a rt*) **by** *fact*

```
from Node.prems obtain
  lt: Dag (low p) low high lt and
  rt: Dag (high p) low high rt
  by auto
show ?case
proof (cases lt)
  case (Node llt l rlt)
  note Nlt=this
  show ?thesis
  proof (cases rt)
    case (Node lrt r rrt)
    note Nrt=this
    from Nlt Nrt bdt1 obtain lbdt rbdt where
      lbdt-def: bdt lt var = Some lbdt and
      rbdt-def: bdt rt var = Some rbdt and
      bdt1-def: bdt1 = Bdt-Node lbdt (var a) rbdt
      by (auto split: if-split-asm option.splits)
    from no-in-t show ?thesis
    proof (simp, elim disjE)
      assume  no = a
      with p-tree Nlt Nrt show ?thesis
        by auto
    next
      assume no ∈ set-of lt
      with Node.hyps lbdt-def lt show ?thesis
        by simp
    next
      assume no ∈ set-of rt
      with Node.hyps rbdt-def rt show ?thesis
        by simp
    qed
  next
    case Tip
    with Nlt bdt1 show ?thesis
      by simp
  qed
next
  case Tip
  note ltTip=this
  show ?thesis
  proof (cases rt)
    case Tip
    with ltTip bdt1 no-in-t p-tree show ?thesis
      by auto
  next
    case (Node rlt r rrt)
    with bdt1 ltTip show ?thesis
      by simp
  qed
```

qed
qed

primrec *dag-map* :: (*ref* ⇒ *ref*) ⇒ *dag* ⇒ *dag*
where
dag-map f Tip = *Tip* |
dag-map f (*Node l a r*) = (*Node* (*dag-map f l*) (*f a*) (*dag-map f r*))

definition *wf-marking* :: *dag* ⇒ (*ref* ⇒ *bool*) ⇒ (*ref* ⇒ *bool*) ⇒ *bool* ⇒ *bool*
where
wf-marking t mark-old mark-new marked =
(*case t of Tip* ⇒ *mark-new* = *mark-old*
| (*Node lt p rt*) ⇒
  (∀ *n*. *n* ∉ *set-of t* ⟶ *mark-new n* = *mark-old n*) ∧
  (∀ *n*. *n* ∈ *set-of t* ⟶ *mark-new n* = *marked*))

definition *dag-in-levellist*:: *dag* ⇒ (*ref list list*) ⇒ (*ref* ⇒ *nat*) ⇒ *bool*
where *dag-in-levellist t levellist var* = (*t* ≠ *Tip* ⟶
    (∀ *st*. *subdag-eq t st* ⟶ *root st* ∈ *set* (*levellist* ! (*var* (*root st*)))))

lemma *set-of-nn*: ⟦ *Dag p low high t*; *n* ∈ *set-of t*⟧ ⟹ *n* ≠ *Null*
apply (*induct t*)
apply *simp*
apply *auto*
done

lemma *subnodes-ordered* [*rule-format*]:
∀ *p*. *n* ∈ *set-of t* ⟶ *Dag p low high t* ⟶ *ordered t var* ⟶ *var n* <= *var p*
apply (*induct t*)
apply *simp*
apply (*intro allI*)
apply (*erule-tac x*=*low p* **in** *allE*)
apply (*erule-tac x*=*high p* **in** *allE*)
apply *clarsimp*
apply (*case-tac t1*)
apply (*case-tac t2*)
apply *simp*
apply *fastforce*
apply (*case-tac t2*)
apply *fastforce*
apply *fastforce*
done

lemma *ordered-set-of*:
⋀ *x*. ⟦*ordered t var*; *x* ∈ *set-of t*⟧ ⟹ *var x* <= *var* (*root t*)
apply (*induct t*)
apply *simp*

apply *simp*
apply (*elim disjE*)
apply *simp*
apply (*case-tac t1*)
apply *simp*
apply (*case-tac t2*)
apply *fastforce*
apply *fastforce*
apply (*case-tac t2*)
apply *simp*
apply (*case-tac t1*)
apply *fastforce*
apply *fastforce*
done

lemma *dag-setofD*: ⋀ *p low high n*. ⟦ *Dag p low high t*; *n* ∈ *set-of t* ⟧ ⟹
(∃ *nt*. *Dag n low high nt*) ∧ (∀ *nt*. *Dag n low high nt* ⟶ *set-of nt* ⊆ *set-of t*)
apply (*induct t*)
apply *simp*
apply *auto*
apply *fastforce*
apply (*fastforce dest*: *Dag-unique*)
apply (*fastforce dest*: *Dag-unique*)
apply *blast*
apply *blast*
done

lemma *dag-setof-exD*: ⟦*Dag p low high t*; *n* ∈ *set-of t*⟧ ⟹ ∃ *nt*. *Dag n low high nt*
apply (*simp add*: *dag-setofD*)
done

lemma *dag-setof-subsetD*: ⟦*Dag p low high t*; *n* ∈ *set-of t*; *Dag n low high nt*⟧ ⟹ *set-of nt* ⊆ *set-of t*
apply (*simp add*: *dag-setofD*)
done

lemma *subdag-parentdag-low*: *not* <= *lt* ⟹ *not* <= (*Node lt p rt*) **for** *not*
apply (*cases not* = *lt*)
apply (*cases lt*)
apply *simp*
apply (*cases rt*)
apply *simp*
apply (*simp add*: *le-dag-def less-dag-def*)
apply (*simp add*: *le-dag-def less-dag-def*)
apply (*simp add*: *le-dag-def less-dag-def*)
apply (*simp add*: *le-dag-def less-dag-def*)
done

```
lemma subdag-parentdag-high: not <= rt ⟹ not <= (Node lt p rt) for not
apply (cases not = rt)
apply (cases lt)
apply simp
apply (cases rt)
apply simp
apply (simp add: le-dag-def less-dag-def)
apply (simp add: le-dag-def less-dag-def)
apply (simp add: le-dag-def less-dag-def)
apply (simp add: le-dag-def less-dag-def)
done

lemma set-of-subdag: ⋀ p not no.
⟦Dag p low high t; Dag no low high not; no ∈ set-of t⟧ ⟹ not <= t
proof (induct t)
  case Tip
  then show ?case by simp
next
  case (Node lt po rt)
  note rtNode=this
  from Node.prems have ppo: p=po
    by simp
  show ?case
  proof (cases no = p)
    case True
    with ppo Node.prems have not = (Node lt po rt)
      by (simp add: Dag-unique del: Dag-Ref)
    with Node.prems ppo show ?thesis by (simp add: subdag-eq-def)
  next
    assume  no ≠ p
    with Node.prems have no-in-ltorrt: no ∈ set-of lt ∨ no ∈ set-of rt
      by simp
    show ?thesis
    proof (cases no ∈ set-of lt)
      case True
      from Node.prems ppo have Dag (low po) low high lt
        by simp
      with Node.prems ppo True have not <= lt
        apply −
        apply (rule Node.hyps)
        apply assumption+
        done
      with Node.prems no-in-ltorrt show ?thesis
        apply −
        apply (rule subdag-parentdag-low)
        apply simp
        done
    next
      assume no ∉ set-of lt
```

```
      with no-in-ltorrt have no-in-rt: no ∈ set-of rt
        by simp
      from Node.prems ppo have Dag (high po) low high rt
        by simp
      with Node.prems ppo no-in-rt have not <= rt
        apply −
        apply (rule Node.hyps)
        apply assumption+
        done
      with Node.prems no-in-rt show ?thesis
        apply −
        apply (rule subdag-parentdag-high)
        apply simp
        done
    qed
  qed
qed

lemma children-ordered: ⟦ordered (Node lt p rt) var⟧ ⟹
ordered lt var ∧ ordered rt var
proof (cases lt)
  case Tip
  note ltTip=this
  assume orderedNode: ordered (Node lt p rt) var
  show ?thesis
  proof (cases rt)
    case Tip
    note rtTip = this
    with ltTip show ?thesis
      by simp
  next
    case (Node lrt r rrt)
    with orderedNode ltTip show ?thesis
      by simp
  qed
next
  case (Node llt l rlt)
  note ltNode=this
  assume orderedNode: ordered (Node lt p rt) var
  show ?thesis
  proof (cases rt)
    case Tip
    note rtTip = this
    with orderedNode ltNode show ?thesis by simp
  next
    case (Node lrt r rrt)
    note rtNode = this
    with orderedNode ltNode show ?thesis by simp
  qed
```

qed

lemma *ordered-subdag*: ⟦*ordered t var*; *not* <= *t*⟧ ⟹ *ordered not var* **for** *not*
proof (*induct t*)
  case *Tip*
  then show *?thesis* **by** (*simp add*: *less-dag-def le-dag-def*)
next
  case (*Node lt p rt*)
  show *?case*
  proof (*cases not* = *Node lt p rt*)
    case *True*
    with *Node.prems* **show** *?thesis* **by** *simp*
  next
    assume *notnt*: *not* ≠ *Node lt p rt*
    with *Node.prems* **have** *notstltorrt*: *not* <= *lt* ∨ *not* <= *rt*
      apply −
      apply (*simp add*: *less-dag-def le-dag-def*)
      apply *fastforce*
      done
    from *Node.prems* **have** *ord-lt*: *ordered lt var*
      apply −
      apply (*drule children-ordered*)
      apply *simp*
      done
    from *Node.prems* **have** *ord-rt*: *ordered rt var*
      apply −
      apply (*drule children-ordered*)
      apply *simp*
      done
    show *?thesis*
    proof (*cases not* <= *lt*)
      case *True*
      with *ord-lt* **show** *?thesis*
        apply −
        apply (*rule Node.hyps*)
        apply *assumption+*
        done
    next
      assume ¬ *not* ≤ *lt*
      with *notstltorrt* **have** *notinrt*: *not* <= *rt*
        by *simp*
      from *Node.hyps* **have** *hyprt*: ⟦*ordered rt var*; *not* ≤ *rt*⟧ ⟹ *ordered not var*
by *simp*
      from *notinrt ord-rt* **show** *?thesis*
        apply −
        apply (*rule hyprt*)
        apply *assumption+*
        done
    qed

```
  qed
qed

lemma subdag-ordered:
⋀ not no p. ⟦ordered t var; Dag p low high t; Dag no low high not;
            no ∈ set-of t⟧ ⟹ ordered not var
proof (induct t)
  case Tip
  from Tip.prems show ?case by simp
next
  case (Node lt po rt)
  note nN=this
  show ?case
  proof (cases lt)
    case Tip
    note ltTip=this
    show ?thesis
    proof (cases rt)
      case Tip
      from Node.prems have ppo: p=po
        by simp
      from Tip ltTip Node.prems have no=p
        by simp
      with ppo Node.prems have not=(Node lt po rt)
        by (simp del: Dag-Ref add: Dag-unique)
      with Node.prems show ?thesis by simp
    next
      case (Node lrnot rn rrnot)
      from Node.prems ltTip Node have ord-rt: ordered rt var
        by simp
      from Node.prems have ppo: p=po
        by simp
      from Node.prems have ponN: po ≠ Null
        by auto
      with ppo ponN ltTip Node.prems have *: Dag (high po) low high rt
        by auto
      show ?thesis
      proof (cases no=po)
        case True
        with ppo Node.prems have not = Node lt po rt
          by (simp del: Dag-Ref add: Dag-unique)
        with Node.prems show ?thesis
          by simp
      next
        case False
        with Node.prems ltTip have no ∈ set-of rt
          by simp
        with ord-rt * ‹Dag no low high not› show ?thesis
```

by (*rule Node.hyps*)
qed
qed
next
case (*Node llt l rlt*)
note *ltNode=this*
show *?thesis*
proof (*cases rt*)
case *Tip*
from *Node.prems Tip ltNode* **have** *ord-lt*: *ordered lt var*
by *simp*
from *Node.prems* **have** *ppo*: *p=po*
by *simp*
from *Node.prems* **have** *ponN*: *po* ≠ *Null*
by *auto*
with *ppo ponN Tip Node.prems ltNode* **have** ∗: *Dag* (*low po*) *low high lt*
by *auto*
show *?thesis*
proof (*cases no=po*)
case *True*
with *ppo Node.prems* **have** *not* = (*Node lt po rt*)
by (*simp del*: *Dag-Ref add*: *Dag-unique*)
with *Node.prems* **show** *?thesis* **by** *simp*
next
case *False*
with *Node.prems Tip* **have** *no* ∈ *set-of lt*
by *simp*
with *ord-lt* ∗ ‹*Dag no low high not*› **show** *?thesis*
by (*rule Node.hyps*)
qed
next
case (*Node lrt r rrt*)
from *Node.prems* **have** *ppo*: *p=po*
by *simp*
from *Node.prems Node ltNode* **have** *ord-lt*: *ordered lt var*
by *simp*
from *Node.prems Node ltNode* **have** *ord-rt*: *ordered rt var*
by *simp*
from *Node.prems* **have** *ponN*: *po* ≠ *Null*
by *auto*
with *ppo ponN Node Node.prems ltNode* **have** *lt-Dag*: *Dag* (*low po*) *low high lt*
by *auto*
with *ppo ponN Node Node.prems ltNode* **have** *rt-Dag*: *Dag* (*high po*) *low high rt*
by *auto*
show *?thesis*
proof (*cases no* = *po*)
case *True*

with *ppo Node.prems* **have** *not* = (*Node lt po rt*)
by (*simp del*: *Dag-Ref add*: *Dag-unique*)
with *Node.prems* **show** *?thesis* **by** *simp*
next
assume $no \neq po$
with *Node.prems* **have** *no-in-lt-or-rt*: $no \in$ *set-of lt* $\vee$ $no \in$ *set-of rt*
by *simp*
show *?thesis*
proof (*cases no* $\in$ *set-of lt*)
case *True*
with *ord-lt lt-Dag Node.prems* **show** *?thesis*
apply −
apply (*rule Node.hyps*)
apply *assumption+*
done
next
assume $no \notin$ *set-of lt*
with *no-in-lt-or-rt* **have** *no-in-rt*: $no \in$ *set-of rt*
by *simp*
from *Node.hyps* **have** *hyp2*:
$\bigwedge$*p no not*. ⟦*ordered rt var*; *Dag p low high rt*; *Dag no low high not*; *no*
$\in$ *set-of rt*⟧
$\Longrightarrow$ *ordered not var*
apply −
apply *assumption*
done
from *no-in-rt ord-rt rt-Dag Node.prems* **show** *?thesis*
apply −
apply (*rule hyp2*)
apply *assumption+*
done
qed
qed
qed
qed
qed

lemma *elem-set-of*: $\bigwedge$ *x st*. ⟦$x \in$ *set-of st*; *set-of st* $\subseteq$ *set-of t*⟧ $\Longrightarrow$ $x \in$ *set-of t*
by *blast*

definition *wf-ll* :: *dag* ⇒ *ref list list* ⇒ (*ref* ⇒ *nat*) ⇒ *bool*
where
wf-ll t levellist var =
((∀ *p*. $p \in$ *set-of t* ⟶ $p \in$ *set* (*levellist* ! *var p*)) ∧
(∀ $k <$ *length levellist*. ∀ $p \in$ *set* (*levellist* ! *k*). $p \in$ *set-of t* ∧ *var p* = *k*))

definition *cong-eval* :: *bdt* ⇒ *bdt* ⇒ *bool* (**infix** ‹∼› *60*)
where *cong-eval* bdt_1 bdt_2 = (*eval* bdt_1 = *eval* bdt_2)

lemma *cong-eval-sym*: *l* ∼ *r* = *r* ∼ *l*
by (*auto simp add*: *cong-eval-def*)

lemma *cong-eval-trans*: ⟦ *l* ∼ *r*; *r* ∼ *t*⟧ ⟹ *l* ∼ *t*
by (*simp add*: *cong-eval-def*)

lemma *cong-eval-child-high*: *l* ∼ *r* ⟹ *r* ∼ (*Bdt-Node l v r*)
apply (*simp add*: *cong-eval-def*)
apply (*rule ext*)
apply *auto*
done

lemma *cong-eval-child-low*: *l* ∼ *r* ⟹ *l* ∼ (*Bdt-Node l v r*)
apply (*simp add*: *cong-eval-def*)
apply (*rule ext*)
apply *auto*
done

definition *null-comp* :: (*ref* ⇒ *ref*) ⇒ (*ref* ⇒ *ref*) ⇒ (*ref* ⇒ *ref*) (**infix** ‹∝› *60*)
where *null-comp a b* = (λ *p*. (*if* (*b p*) = *Null then Null else* ((*a* ∘ *b*) *p*)))

lemma *null-comp-not-Null* [*simp*]: *h q* ≠ *Null* ⟹ (*g* ∝ *h*) *q* = *g* (*h q*)
by (*simp add*: *null-comp-def*)

lemma *id-trans*: (*a* ∝ *id*) (*b* (*c p*)) = (*a* ∝ *b*) (*c p*)
by (*simp add*: *null-comp-def*)

definition *Nodes* :: *nat* ⇒ *ref list list* ⇒ *ref set*
where *Nodes i levellist* = (⋃*k*∈{*k*. *k* < *i*} . *set* (*levellist* ! *k*))

inductive-set *Dags* :: *ref set* ⇒ (*ref* ⇒ *ref*) ⇒ (*ref* ⇒ *ref*) ⇒ *dag set*
for *nodes low high*
where
DagsI: ⟦ *set-of t* ⊆ *nodes*; *Dag p low high t*; *t* ≠ *Tip*⟧
⟹ *t* ∈ *Dags nodes low high*

lemma *empty-Dags* [*simp*]: *Dags* {} *low high* = {}
apply *rule*
apply *rule*
apply (*erule Dags.cases*)
apply (*case-tac t*)
apply *simp*

apply *simp*
apply *rule*
done

definition *isLeaf-pt* :: *ref* ⇒ (*ref* ⇒ *ref*) ⇒ (*ref* ⇒ *ref*) ⇒ *bool*
where *isLeaf-pt p low high* = (*low p* = *Null* ∧ *high p* = *Null*)

definition *repNodes-eq* :: *ref* ⇒ *ref* ⇒ (*ref* ⇒ *ref*) ⇒ (*ref* ⇒ *ref*) ⇒ (*ref* ⇒ *ref*) ⇒ *bool*
where
repNodes-eq p q low high rep ==
(*rep* ∝ *high*) *p* = (*rep* ∝ *high*) *q* ∧ (*rep* ∝ *low*) *p* = (*rep* ∝ *low*) *q*

definition *isomorphic-dags-eq* :: *dag* ⇒ *dag* ⇒ (*ref* ⇒ *nat*) ⇒ *bool*
where
isomorphic-dags-eq st_1 st_2 *var* =
(∀ bdt_1 bdt_2. (*bdt* st_1 *var* = *Some* bdt_1 ∧ *bdt* st_2 *var* = *Some* bdt_2 ∧ (bdt_1 = bdt_2))
⟶ st_1 = st_2)

lemma *isomorphic-dags-eq-sym*: *isomorphic-dags-eq* st_1 st_2 *var* = *isomorphic-dags-eq* st_2 st_1 *var*
by (*auto simp add*: *isomorphic-dags-eq-def*)

definition *shared* :: *dag* ⇒ (*ref* ⇒ *nat*) ⇒ *bool*
where *shared t var* = (∀ st_1 st_2. (st_1 <= *t* ∧ st_2 <= *t*) ⟶ *isomorphic-dags-eq* st_1 st_2 *var*)

fun *reduced* :: *dag* ⇒ *bool* **where**
reduced Tip = *True*
| *reduced* (*Node Tip v Tip*) = *True*
| *reduced* (*Node l v r*) = (*l* ≠ *r* ∧ *reduced l* ∧ *reduced r*)

primrec *reduced-bdt* :: *bdt* ⇒ *bool*
where
reduced-bdt Zero = *True*
| *reduced-bdt One* = *True*
| *reduced-bdt* (*Bdt-Node lbdt v rbdt*) =
(*if lbdt* = *rbdt then False*
else (*reduced-bdt lbdt* ∧ *reduced-bdt rbdt*))

lemma *replicate-elem*: *i* < *n* ==> (*replicate n x* !*i*) = *x*

apply (*induct n*)
apply *simp*
apply (*cases i*)
apply *auto*
done

lemma *no-in-one-ll*:
⟦*wf-ll pret levellista var*; *i*<*length levellista*; *j* < *length levellista*;
no ∈ *set* (*levellista* ! *i*); *i*≠*j*⟧
⟹ *no* ∉ *set* (*levellista* ! *j*)
apply (*unfold wf-ll-def*)
apply (*erule conjE*)
apply (*rotate-tac 5*)
apply (*frule-tac x* = *i* **and** *?R*= *no* ∈ *set-of pret* ∧ *var no* = *i* **in** *allE*)
apply (*erule impE*)
apply *simp*
apply (*rotate-tac 6*)
apply (*erule-tac x*=*no* **in** *ballE*)
apply *assumption*
apply *simp*
apply (*cases no* ∉ *set* (*levellista* ! *j*))
apply *assumption*
apply (*erule-tac x*=*j* **in** *allE*)
apply (*erule impE*)
apply *assumption*
apply (*rotate-tac 7*)
apply (*erule-tac x*=*no* **in** *ballE*)
prefer *2*
apply *assumption*
apply (*elim conjE*)
apply (*thin-tac* ∀ *q*. *q* ∈ *set-of pret* ⟶ *q* ∈ *set* (*levellista* ! *var q*))
apply *fastforce*
done

lemma *nodes-in-wf-ll*:
⟦*wf-ll pret levellista var*; *i* < *length levellista*; *no* ∈ *set* (*levellista* ! *i*)⟧
⟹ *var no* = *i* ∧ *no* ∈ *set-of pret*
apply (*simp add*: *wf-ll-def*)
done

lemma *subelem-set-of-low*:
⋀ *p*. ⟦ *x* ∈ *set-of t*; *x* ≠ *Null*; *low x* ≠ *Null*; *Dag p low high t* ⟧
⟹ (*low x*) ∈ *set-of t*
proof (*induct t*)
case *Tip*
then show *?case* **by** *simp*
next
case (*Node lt po rt*)
note *tNode*=*this*

then have *ppo*: $p=po$ **by** *simp*
show *?case*
proof (*cases* $x=p$)
  case *True*
  with *Node.prems* **have** *lxrootlt*: $low\ x = root\ lt$
  proof (*cases lt*)
    case *Tip*
    with *True Node.prems* **show** *?thesis*
      by *auto*
  next
    case (*Node llt l rlt*)
    with *Node.prems True* **show** *?thesis*
      by *auto*
  qed
  with *True Node.prems* **have** $low\ x \in set\text{-}of\ (Node\ lt\ p\ rt)$
  proof (*cases lt*)
    case *Tip*
    with *lxrootlt Node.prems* **show** *?thesis*
      by *simp*
  next
    case (*Node llt l rlt*)
    with *lxrootlt Node.prems* **show** *?thesis*
      by *simp*
  qed
  with *ppo* **show** *?thesis*
    by *simp*
next
  assume *xnp*: $x \neq p$
  with *Node.prems* **have** $x \in set\text{-}of\ lt \vee x \in set\text{-}of\ rt$
    by *simp*
  show *?thesis*
  proof (*cases* $x \in set\text{-}of\ lt$)
    case *True*
    note *xinlt=this*
    from *Node.prems* **have** *Dag* (*low p*) *low high lt*
      by *fastforce*
    with *Node.prems True* **have** $low\ x \in set\text{-}of\ lt$
      apply −
      apply (*rule Node.hyps*)
      apply *assumption+*
      done
    with *Node.prems* **show** *?thesis*
      by *auto*
  next
    assume *xnotinlt*: $x \notin set\text{-}of\ lt$
    with *xnp Node.prems* **have** *xinrt*: $x \in set\text{-}of\ rt$
      by *simp*
    from *Node.prems* **have** *Dag* (*high p*) *low high rt*
      by *fastforce*

```
      with Node.prems xinrt have low x ∈ set-of rt
        apply −
        apply (rule Node.hyps)
        apply assumption+
        done
      with Node.prems show ?thesis
        by auto
    qed
  qed
qed

lemma subelem-set-of-high:
⋀ p. ⟦ x ∈ set-of t; x ≠ Null; high x ≠ Null; Dag p low high t ⟧
 ⟹ (high x) ∈ set-of t
proof (induct t)
  case Tip
  then show ?case by simp
next
  case (Node lt po rt)
  note tNode=this
  then have ppo: p=po by simp
  show ?case
  proof (cases x=p)
    case True
    with Node.prems have lxrootlt: high x = root rt
    proof (cases rt)
      case Tip
      with True Node.prems show ?thesis
        by auto
    next
      case (Node lrt l rrt)
      with Node.prems True show ?thesis
        by auto
    qed
    with True Node.prems have high x ∈ set-of (Node lt p rt)
    proof (cases rt)
      case Tip
      with lxrootlt Node.prems show ?thesis
        by simp
    next
      case (Node lrt l rrt)
      with lxrootlt Node.prems show ?thesis
        by simp
    qed
    with ppo show ?thesis
      by simp
  next
    assume xnp: x ≠ p
    with Node.prems have x ∈ set-of lt ∨ x ∈ set-of rt
```

```
      by simp
    show ?thesis
    proof (cases x ∈ set-of lt)
      case True
      note xinlt=this
      from Node.prems have Dag (low p) low high lt
        by fastforce
      with Node.prems True have high x ∈ set-of lt
        apply −
        apply (rule Node.hyps)
        apply assumption+
        done
      with Node.prems show ?thesis
        by auto
    next
      assume xnotinlt: x ∉ set-of lt
      with xnp Node.prems have xinrt: x ∈ set-of rt
        by simp
      from Node.prems have Dag (high p) low high rt
        by fastforce
      with Node.prems xinrt have high x ∈ set-of rt
        apply −
        apply (rule Node.hyps)
        apply assumption+
        done
      with Node.prems show ?thesis
        by auto
    qed
  qed
qed

lemma set-split: {k. k<(Suc n)} = {k. k<n} ∪ {n}
apply auto
done


lemma Nodes-levellist-subset-t:
⟦wf-ll t levellist var; i<= length levellist⟧ ⟹ Nodes i levellist ⊆ set-of t
proof (induct i)
  case 0
  show ?case by (simp add: Nodes-def)
next
  case (Suc n)
  from Suc.prems Suc.hyps have Nodesn-in-t: Nodes n levellist ⊆ set-of t
    by simp
  from Suc.prems have ∀ x ∈ set (levellist ! n). x ∈ set-of t
    apply −
    apply (rule ballI)
    apply (simp add: wf-ll-def)
```

apply (*erule conjE*)
apply (*thin-tac* ∀ *q*. *q* ∈ *set-of t* ⟶ *q* ∈ *set* (*levellist* ! *var q*))
apply (*erule-tac x*=*n* **in** *allE*)
apply (*erule impE*)
apply *simp*
apply *fastforce*
done
with *Suc.prems* **have** *set* (*levellist* ! *n*) ⊆ *set-of t*
apply *blast*
done
with *Suc.prems Nodesn-in-t* **show** *?case*
apply (*simp add*: *Nodes-def*)
apply (*simp add*: *set-split*)
done
qed

lemma *bdt-child*:
⟦ *bdt* (*Node* (*Node llt l rlt*) *p* (*Node lrt r rrt*)) *var* = *Some bdt1*⟧
⟹ ∃ *lbdt rbdt*. *bdt* (*Node llt l rlt*) *var* = *Some lbdt* ∧
bdt (*Node lrt r rrt*) *var* = *Some rbdt*
by (*simp split*: *option.splits*)

lemma *subbdt-ex-dag-def*:
⋀ *bdt1 p*. ⟦*Dag p low high t*; *bdt t var* = *Some bdt1*; *Dag no low high not*;
no ∈ *set-of t*⟧ ⟹ ∃ *bdt2*. *bdt not var* = *Some bdt2* **for** *not*
proof (*induct t*)
case *Tip*
then show *?case* **by** *simp*
next
case (*Node lt po rt*)
note *pNode*=*this*
with *Node.prems* **have** *p-po*: *p*=*po* **by** *simp*
show *?case*
proof (*cases no* = *po*)
case *True*
note *no-eq-po*=*this*
from *p-po Node.prems no-eq-po* **have** *not* = (*Node lt po rt*) **by** (*simp add*: *Dag-unique del*: *Dag-Ref*)
with *Node.prems* **have** *bdt not var* = *Some bdt1* **by** (*simp add*: *le-dag-def*)
then show *?thesis* **by** *simp*
next
assume *no* ≠ *po*
with *Node.prems* **have** *no-in-lt-or-rt*: *no* ∈ *set-of lt* ∨ *no* ∈ *set-of rt* **by** *simp*
show *?thesis*
proof (*cases no* ∈ *set-of lt*)
case *True*
note *no-in-lt*=*this*
from *Node.prems p-po* **have** *lt-dag*: *Dag* (*low po*) *low high lt* **by** *simp*

```
      from Node.prems have lbdt-def: ∃ lbdt. bdt lt var = Some lbdt
      proof (cases lt)
        case Tip
        with Node.prems no-in-lt show ?thesis by (simp add: le-dag-def)
      next
        case (Node llt l rlt)
        note lNode=this
        show ?thesis
        proof (cases rt)
          case Tip
          note rNode=this
          with lNode Node.prems show ?thesis by simp
        next
          case (Node lrt r rrt)
          note rNode=this
          with lNode Node.prems show ?thesis by (simp split: option.splits)
        qed
      qed
      then obtain lbdt where bdt lt var = Some lbdt..
      with Node.prems lt-dag no-in-lt show ?thesis
        apply −
        apply (rule Node.hyps)
        apply assumption+
        done
    next
      assume no ∉ set-of lt
      with no-in-lt-or-rt have no-in-rt: no ∈ set-of rt by simp
      from Node.prems p-po have rt-dag: Dag (high po) low high rt by simp
      from Node.hyps have hyp2: ⋀ rbdt. ⟦Dag (high po) low high rt; bdt rt var =
Some rbdt; Dag no low high not; no ∈ set-of rt⟧ ⟹ ∃ bdt2. bdt not var = Some
bdt2
        by simp
      from Node.prems have lbdt-def: ∃ rbdt. bdt rt var = Some rbdt
      proof (cases rt)
        case Tip
        with Node.prems no-in-rt show ?thesis by (simp add: le-dag-def)
      next
        case (Node lrt l rrt)
        note rNode=this
        show ?thesis
        proof (cases lt)
          case Tip
          note lTip=this
          with rNode Node.prems show ?thesis by simp
        next
          case (Node llt r rlt)
          note lNode=this
          with rNode Node.prems show ?thesis by (simp split: option.splits)
        qed
```

```
      qed
      then obtain rbdt where bdt rt var = Some rbdt..
      with Node.prems rt-dag no-in-rt show ?thesis
        apply −
        apply (rule hyp2)
        apply assumption+
        done
    qed
  qed
qed

lemma subbdt-ex:
⋀ bdt1. ⟦ (Node lst stp rst) <= t; bdt t var = Some bdt1⟧
 ⟹ ∃ bdt2. bdt (Node lst stp rst) var = Some bdt2
proof (induct t)
  case Tip
  then show ?case by (simp add: le-dag-def)
next
  case (Node lt p rt)
  note pNode=this
  show ?case
  proof (cases Node lst stp rst = Node lt p rt)
    case True
    with Node.prems show ?thesis by simp
  next
    assume Node lst stp rst ≠ Node lt p rt
     with Node.prems have Node lst stp rst < Node lt p rt apply (simp add:
le-dag-def) apply auto done
    then have in-ltrt: Node lst stp rst <= lt ∨ Node lst stp rst <= rt
      by (simp add: less-dag-Node)
    show ?thesis
    proof (cases Node lst stp rst <= lt)
      case True
      note in-lt=this
      from Node.prems have lbdt-def: ∃ lbdt. bdt lt var = Some lbdt
      proof (cases lt)
        case Tip
        with Node.prems in-lt show ?thesis by (simp add: le-dag-def)
      next
        case (Node llt l rlt)
        note lNode=this
        show ?thesis
        proof (cases rt)
          case Tip
          note rNode=this
          with lNode Node.prems show ?thesis by simp
        next
          case (Node lrt r rrt)
          note rNode=this
```

```
      with lNode Node.prems show ?thesis by (simp split: option.splits)
    qed
  qed
  then obtain lbdt where bdt lt var = Some lbdt..
  with Node.prems in-lt show ?thesis
    apply -
    apply (rule Node.hyps)
    apply assumption+
    done
next
  assume ¬ Node lst stp rst ≤ lt
  with in-ltrt have in-rt: Node lst stp rst <= rt by simp
   from Node.hyps have hyp2: ⋀ rbdt. ⟦Node lst stp rst <= rt; bdt rt var =
Some rbdt⟧ ⟹ ∃ bdt2. bdt (Node lst stp rst) var = Some bdt2
    by simp
  from Node.prems have rbdt-def: ∃ rbdt. bdt rt var = Some rbdt
  proof (cases rt)
    case Tip
    with Node.prems in-rt show ?thesis by (simp add: le-dag-def)
  next
    case (Node lrt l rrt)
    note rNode=this
    show ?thesis
    proof (cases lt)
      case Tip
      note lNode=this
      with rNode Node.prems show ?thesis by simp
    next
      case (Node lrt r rrt)
      note lNode=this
      with rNode Node.prems show ?thesis by (simp split: option.splits)
    qed
  qed
  then obtain rbdt where bdt rt var = Some rbdt..
  with Node.prems in-rt show ?thesis
    apply -
    apply (rule hyp2)
    apply assumption+
    done
qed
qed
qed


lemma var-ordered-children:
⋀ p. ⟦ Dag p low high t; ordered t var; no ∈ set-of t;
    low no ≠ Null; high no ≠ Null⟧
  ⟹ var (low no) < var no ∧ var (high no) < var no
proof (induct t)
```

```
  case Tip
  then show ?case by simp
next
  case (Node lt po rt)
  then have ppo: p=po by simp
  show ?case
  proof (cases no = po)
    case True
    note no-po=this
    from Node.prems have var (low po) < var po ∧ var (high po) < var po
    proof (cases lt)
      case Tip
      note ltTip=this
      with Node.prems no-po ppo show ?thesis by simp
    next
      case (Node llt l rlt)
      note lNode=this
      show ?thesis
      proof (cases rt)
        case Tip
        note rTip=this
        with Node.prems no-po ppo show ?thesis by simp
      next
        case (Node lrt r rrt)
        note rNode=this
        with Node.prems ppo no-po lNode  show ?thesis by (simp del: Dag-Ref)
      qed
    qed
    with no-po show ?thesis by simp
  next
    assume  no ≠ po
    with Node.prems have no-in-ltrt: no ∈ set-of lt ∨ no ∈ set-of rt
      by simp
    show ?thesis
    proof (cases no ∈ set-of lt)
      case True
      note no-in-lt=this
      from Node.prems ppo have lt-dag: Dag (low po) low high lt
        by simp
      from Node.prems have ord-lt: ordered lt var
        apply −
        apply (drule children-ordered)
        apply simp
        done
      from no-in-lt lt-dag ord-lt Node.prems show ?thesis
        apply −
        apply (rule Node.hyps)
        apply assumption+
        done
```

next
assume *no ∉ set-of lt*
with *no-in-ltrt* **have** *no-in-rt*: *no ∈ set-of rt* **by** *simp*
from *Node.prems ppo* **have** *rt-dag*: *Dag (high po) low high rt* **by** *simp*
from *Node.hyps* **have** *hyp2*: ⟦*Dag (high po) low high rt*; *ordered rt var*; *no ∈ set-of rt*; *low no ≠ Null*; *high no ≠ Null*⟧
⟹ *var (low no) < var no ∧ var (high no) < var no*
by *simp*
from *Node.prems* **have** *ord-rt*: *ordered rt var*
apply −
apply (*drule children-ordered*)
apply *simp*
done
from *rt-dag ord-rt no-in-rt Node.prems* **show** *?thesis*
apply −
apply (*rule hyp2*)
apply *assumption+*
done
qed
qed
qed

lemma *nort-null-comp*:
assumes *pret-dag*: *Dag p low high pret* **and**
prebdt-pret: *bdt pret var = Some prebdt* **and**
nort-dag: *Dag (repc no) (repb ∝ low) (repb ∝ high) nort* **and**
ord-pret: *ordered pret var* **and**
wf-llb: *wf-ll pret levellistb var* **and**
nbsll: *nb < length levellistb* **and**
repbc-nc: ∀ *nt. nt ∉ set (levellistb ! nb) ⟶ repb nt = repc nt* **and**
xsnb-in-pret: ∀ *x ∈ set-of nort. var x < nb ∧ x ∈ set-of pret*
shows ∀ *x ∈ set-of nort. ((repc ∝ low) x = (repb ∝ low) x ∧*
(repc ∝ high) x = (repb ∝ high) x)
proof (*rule ballI*)
fix *x*
assume *x-in-nort*: *x ∈ set-of nort*
with *nort-dag* **have** *xnN*: *x ≠ Null*
apply −
apply (*rule set-of-nn* [*rule-format*])
apply *auto*
done
from *x-in-nort xsnb-in-pret* **have** *xsnb*: *var x <nb*
by *simp*
from *x-in-nort xsnb-in-pret* **have** *x-in-pret*: *x ∈ set-of pret*
by *blast*
show *(repc ∝ low) x = (repb ∝ low) x ∧ (repc ∝ high) x = (repb ∝ high) x*
proof (*cases (low x) ≠ Null*)
case *True*
with *pret-dag prebdt-pret x-in-pret* **have** *highnN*: *(high x) ≠ Null*

apply −
apply (*drule balanced-bdt*)
apply *assumption+*
apply *simp*
done
from *x-in-pret ord-pret highnN True* **have** *children-var-smaller*: *var* (*low x*) < *var x* ∧ *var* (*high x*) < *var x*
apply −
apply (*rule var-ordered-children*)
apply (*rule pret-dag*)
apply (*rule ord-pret*)
apply (*rule x-in-pret*)
apply (*rule True*)
apply (*rule highnN*)
done
with *xsnb* **have** *lowxsnb*: *var* (*low x*) < *nb*
by *arith*
from *children-var-smaller xsnb* **have** *highxsnb*: *var* (*high x*) < *nb*
by *arith*
from *x-in-pret xnN True pret-dag* **have** *lowxinpret*: (*low x*) ∈ *set-of pret*
apply −
apply (*drule subelem-set-of-low*)
apply *assumption*
apply (*thin-tac x* ≠ *Null*)
apply *assumption+*
done
with *wf-llb* **have** *low x* ∈ *set* (*levellistb* ! (*var* (*low x*)))
by (*simp add*: *wf-ll-def*)
with *wf-llb nbsll lowxsnb* **have** *low x* ∉ *set* (*levellistb* ! *nb*)
apply −
apply (*rule-tac ?i*=(*var* (*low x*)) **and** *?j*=*nb* **in** *no-in-one-ll*)
apply *auto*
done
with *repbc-nc* **have** *repclow*: *repc* (*low x*) = *repb* (*low x*)
by *auto*
from *x-in-pret xnN highnN pret-dag* **have** *highxinpret*: (*high x*) ∈ *set-of pret*
apply −
apply (*drule subelem-set-of-high*)
apply *assumption*
apply (*thin-tac x* ≠ *Null*)
apply *assumption+*
done
with *wf-llb* **have** *high x* ∈ *set* (*levellistb* ! (*var* (*high x*)))
by (*simp add*: *wf-ll-def*)
with *wf-llb nbsll highxsnb* **have** *high x* ∉ *set* (*levellistb* ! *nb*)
apply −
apply (*rule-tac ?i*=(*var* (*high x*)) **and** *?j*=*nb* **in** *no-in-one-ll*)
apply *auto*
done

with *repbc-nc* **have** *repchigh*: *repc* (*high x*) = *repb* (*high x*)
by *auto*
with *repclow* **show** *?thesis*
by (*simp add*: *null-comp-def*)
next
assume ¬ *low x* ≠ *Null*
then have *lowxNull*: *low x* = *Null* **by** *simp*
with *pret-dag x-in-pret prebdt-pret* **have** *highxNull*: *high x* =*Null*
apply −
apply (*drule balanced-bdt*)
apply *simp*
apply *simp*
apply *simp*
done
from *lowxNull* **have** *repclowNull*: (*repc* ∝ *low*) *x* = *Null*
by (*simp add*: *null-comp-def*)
from *lowxNull* **have** *repblowNull*: (*repb* ∝ *low*) *x* = *Null*
by (*simp add*: *null-comp-def*)
with *repclowNull* **have** *lowxrepbc*: (*repc* ∝ *low*) *x* = (*repb* ∝ *low*) *x*
by *simp*
from *highxNull* **have** *repchighNull*: (*repc* ∝ *high*) *x* = *Null*
by (*simp add*: *null-comp-def*)
from *highxNull* **have** (*repb* ∝ *high*) *x* = *Null*
by (*simp add*: *null-comp-def*)
with *repchighNull* **have** *highxrepbc*: (*repc* ∝ *high*) *x* = (*repb* ∝ *high*) *x*
by *simp*
with *lowxrepbc* **show** *?thesis*
by *simp*
qed
qed

lemma *wf-ll-Nodes-pret*:
⟦*wf-ll pret levellista var*; *nb* < *length levellista*; *x* ∈ *Nodes nb levellista*⟧
⟹ *x* ∈ *set-of pret* ∧ *var x* < *nb*
apply (*simp add*: *wf-ll-def Nodes-def*)
apply (*erule conjE*)
apply (*thin-tac* ∀ *q*. *q* ∈ *set-of pret* ⟶ *q* ∈ *set* (*levellista* ! *var q*))
apply (*erule exE*)
apply (*elim conjE*)
apply (*erule-tac x*=*xa* **in** *allE*)
apply (*erule impE*)
apply *arith*
apply (*erule-tac x*=*x* **in** *ballE*)
apply *auto*
done

lemma *bdt-Some-var1-One*:
⋀ *x*. ⟦ *bdt t var* = *Some x*; *var* (*root t*) = *1*⟧
⟹ *x* = *One* ∧ *t* = (*Node Tip* (*root t*) *Tip*)

```
proof (induct t)
  case Tip
  then show  ?case by simp
next
  case (Node lt p rt)
  note tNode = this
  show ?case
  proof (cases lt)
    case Tip
    note ltTip=this
    show ?thesis
    proof (cases rt)
      case Tip
      note rtTip = this
      with ltTip Node.prems show ?thesis by auto
    next
      case (Node lrt r rrt)
      note rtNode=this
      with Node.prems ltTip show ?thesis by auto
    qed
  next
    case (Node llt l rlt)
    note ltNode=this
    show ?thesis
    proof (cases rt)
      case Tip
      with ltNode Node.prems show ?thesis by auto
    next
      case (Node lrt r rrt)
      note rtNode=this
      with ltNode Node.prems show ?thesis by auto
    qed
  qed
qed

lemma bdt-Some-var0-Zero:
⋀ x. ⟦ bdt t var = Some x; var (root t) = 0⟧
⟹ x = Zero ∧ t = (Node Tip (root t) Tip)
proof (induct t)
  case Tip
  then show  ?case by simp
next
  case (Node lt p rt)
  note tNode = this
  show ?case
  proof (cases lt)
    case Tip
    note ltTip=this
    show ?thesis
```

```
    proof (cases rt)
      case Tip
      note rtTip = this
      with ltTip Node.prems show ?thesis by auto
    next
      case (Node lrt r rrt)
      note rtNode=this
      with Node.prems ltTip show ?thesis by auto
    qed
  next
    case (Node llt l rlt)
    note ltNode=this
    show ?thesis
    proof (cases rt)
      case Tip
      with ltNode Node.prems show ?thesis by auto
    next
      case (Node lrt r rrt)
      note rtNode=this
      with ltNode Node.prems show ?thesis by auto
    qed
  qed
qed

lemma reduced-children-parent:
⟦ reduced l; l= (Node llt lp rlt); reduced r; r=(Node lrt rp rrt);
  lp ≠ rp ⟧
 ⟹ reduced (Node l p r)
  by simp


lemma Nodes-subset: Nodes i levellista ⊆ Nodes (Suc i) levellista
  apply (simp add: Nodes-def)
  apply (simp add: set-split)
  done

lemma Nodes-levellist:
⟦ wf-ll pret levellista var; nb < length levellista; p ∈ Nodes nb levellista⟧
 ⟹ p ∉ set (levellista ! nb)
  apply (simp add: Nodes-def)
  apply (erule exE)
  apply (rule-tac i=x and j=nb in no-in-one-ll)
  apply auto
  done

lemma Nodes-var-pret:
 ⟦wf-ll pret levellista var; nb < length levellista; p ∈ Nodes nb levellista⟧
  ⟹ var p < nb ∧ p ∈ set-of pret
  apply (simp add: Nodes-def wf-ll-def)
```

apply (*erule conjE*)
apply (*thin-tac* ∀ *q*. *q* ∈ *set-of pret* ⟶ *q* ∈ *set* (*levellista* ! *var q*))
apply (*erule exE*)
apply (*erule-tac x*=*x* **in** *allE*)
apply (*erule impE*)
apply *arith*
apply (*erule-tac x*=*p* **in** *ballE*)
apply *arith*
apply *simp*
done

lemma *Dags-root-in-Nodes*:
assumes *t-in-DagsSucnb*: *t* ∈ *Dags* (*Nodes* (*Suc nb*) *levellista*) *low high*
shows ∃ *p* . *Dag p low high t* ∧ *p* ∈ *Nodes* (*Suc nb*) *levellista*
proof −
from *t-in-DagsSucnb* **obtain** *p* **where** *t-dag*: *Dag p low high t* **and** *t-subset-Nodes*: *set-of t* ⊆ *Nodes* (*Suc nb*) *levellista* **and** *t-nTip*: *t*≠ *Tip*
by (*fastforce elim*: *Dags.cases*)
from *t-dag t-nTip* **have** *p*≠*Null* **by** (*cases t*) *auto*
with *t-subset-Nodes t-dag* **have** *p* ∈ *Nodes* (*Suc nb*) *levellista*
by (*cases t*) *auto*
with *t-dag* **show** *?thesis*
by *auto*
qed

lemma *subdag-dag*:
⋀ *p*. ⟦*Dag p low high t*; *st* <= *t*⟧ ⟹ ∃ *stp*. *Dag stp low high st*
proof (*induct t*)
case *Tip*
then show *?case*
by (*simp add*: *less-dag-def le-dag-def*)
next
case (*Node lt po rt*)
note *t-Node*=*this*
with *Node.prems* **have** *p-po*: *p*=*po*
by *simp*
show *?case*
proof (*cases st* = *Node lt po rt*)
case *True*
note *st-t*=*this*
with *Node.prems* **show** *?thesis*
by *auto*
next
assume *st-nt*: *st* ≠ *Node lt po rt*
with *Node.prems p-po* **have** *st-subdag-lt-rt*: *st*<=*lt* ∨ *st* <=*rt*
by (*auto simp add*:*le-dag-def less-dag-def*)

from *Node.prems p-po* **obtain** *lp rp* **where** *lt-dag*: *Dag lp low high lt* **and** *rt-dag*: *Dag rp low high rt*
by *auto*
show *?thesis*
proof (*cases st<=lt*)
case *True*
note *st-lt=this*
with *lt-dag* **show** *?thesis*
apply−
apply (*rule Node.hyps*)
apply *auto*
done
next
assume ¬ *st* ≤ *lt*
with *st-subdag-lt-rt* **have** *st-rt*: *st <= rt*
by *simp*
from *Node.hyps* **have** *rhyp*: ⟦*Dag rp low high rt*; *st* ≤ *rt*⟧ ⟹ ∃ *stp*. *Dag stp low high st*
by *simp*
from *st-rt rt-dag* **show** *?thesis*
apply −
apply (*rule rhyp*)
apply *auto*
done
qed
qed
qed

lemma *Dags-subdags*:
assumes *t-in-Dags*: *t* ∈ *Dags nodes low high* **and**
st-t: *st <= t* **and**
st-nTip: *st* ≠ *Tip*
shows *st* ∈ *Dags nodes low high*
proof −
from *t-in-Dags* **obtain** *p* **where** *t-dag*: *Dag p low high t* **and** *t-subset-Nodes*: *set-of t* ⊆ *nodes* **and** *t-nTip*: *t*≠ *Tip*
by (*fastforce elim*: *Dags.cases*)
from *st-t* **have** *set-of st* ⊆ *set-of t*
by (*simp add*: *le-dag-set-of*)
with *t-subset-Nodes* **have** *st-subset-fnctNodes*: *set-of st* ⊆ *nodes*
by *blast*
from *st-t t-dag* **obtain** *stp* **where** *Dag stp low high st*
apply −
apply (*drule subdag-dag*)
apply *auto*
done
with *st-subset-fnctNodes st-nTip* **show** *?thesis*
apply −
apply (*rule DagsI*)

```
    apply auto
    done
qed


lemma Dags-Nodes-cases:
assumes P-sym: ⋀ t1 t2. P t1 t2 var = P t2 t1 var and
  dags-in-lower-levels:
  ⋀ t1 t2. ⟦t1 ∈ Dags (fnct `(Nodes n levellista)) low high;
           t2 ∈ Dags (fnct `(Nodes n levellista)) low high⟧
          ⟹ P t1 t2 var and
  dags-in-mixed-levels:
  ⋀ t1 t2. ⟦t1 ∈ Dags (fnct `(Nodes n levellista)) low high;
           t2 ∈ Dags (fnct `(Nodes (Suc n) levellista)) low high;
           t2 ∉ Dags (fnct `(Nodes n levellista)) low high⟧
          ⟹ P t1 t2 var and
  dags-in-high-level:
   ⋀ t1 t2. ⟦t1 ∈ Dags (fnct `(Nodes (Suc n) levellista)) low high;
            t1 ∉ Dags (fnct `(Nodes n levellista)) low high;
            t2 ∈ Dags (fnct `(Nodes (Suc n) levellista)) low high;
            t2 ∉ Dags (fnct `(Nodes n levellista)) low high⟧
           ⟹ P t1 t2 var
shows ∀ t1 t2. t1 ∈ Dags (fnct `(Nodes (Suc n) levellista)) low high ∧
               t2 ∈ Dags (fnct `(Nodes (Suc n) levellista)) low high
               ⟶ P t1 t2 var
proof (intro allI impI , elim conjE)
  fix t1 t2
  assume t1-in-higher-levels: t1 ∈ Dags (fnct ` Nodes (Suc n) levellista) low high
  assume t2-in-higher-levels: t2 ∈ Dags (fnct ` Nodes (Suc n) levellista) low high
  show P t1 t2 var
  proof (cases t1 ∈ Dags (fnct ` Nodes n levellista) low high)
    case True
    note t1-in-ll = this
    show ?thesis
    proof (cases t2 ∈ Dags (fnct ` Nodes n levellista) low high)
      case True
      note t2-in-ll=this
      with t1-in-ll dags-in-lower-levels show ?thesis
        by simp
    next
      assume t2-notin-ll: t2 ∉ Dags (fnct ` Nodes n levellista) low high
      with t1-in-ll t2-in-higher-levels dags-in-mixed-levels show ?thesis
        by simp
    qed
  next
    assume t1-notin-ll: t1 ∉ Dags (fnct ` Nodes n levellista) low high
    show ?thesis
    proof (cases t2 ∈ Dags (fnct ` Nodes n levellista) low high)
      case True
```

note *t2-in-ll=this*
with *dags-in-mixed-levels t1-in-higher-levels t1-notin-ll P-sym* **show** *?thesis*
by *auto*
next
assume *t2-notin-ll*: *t2* ∉ *Dags* (*fnct* ‘ *Nodes n levellista*) *low high*
with *t1-notin-ll t1-in-higher-levels t2-in-higher-levels dags-in-high-level* **show** *?thesis*
by *simp*
qed
qed
qed

lemma *Null-notin-Nodes*: ⟦*Dag p low high t*; *nb* <= *length levellista*; *wf-ll t levellista var*⟧ ⟹ *Null* ∉ *Nodes nb levellista*
apply (*simp add*: *Nodes-def wf-ll-def del*: *Dag-Ref*)
apply (*rule allI*)
apply (*rule impI*)
apply (*elim conjE*)
apply (*thin-tac* ∀ *q*. *P q* **for** *P*)
apply (*erule-tac x*=*x* **in** *allE*)
apply (*erule impE*)
apply *simp*
apply (*erule-tac x*=*Null* **in** *ballE*)
apply (*erule conjE*)
apply (*drule set-of-nn* [*rule-format*])
apply *auto*
done

lemma *Nodes-in-pret*: ⟦*wf-ll t levellista var*; *nb* <= *length levellista*⟧ ⟹ *Nodes nb levellista* ⊆ *set-of t*
apply –
apply *rule*
apply (*simp add*: *wf-ll-def Nodes-def*)
apply (*erule exE*)
apply (*elim conjE*)
apply (*thin-tac* ∀ *q*. *q* ∈ *set-of t* ⟶ *q* ∈ *set* (*levellista* ! *var q*))
apply (*erule-tac x*=*xa* **in** *allE*)
apply (*erule impE*)
apply *simp*
apply (*erule-tac x*=*x* **in** *ballE*)
apply *auto*
done

lemma *restrict-root-Node*:
⟦*t* ∈ *Dags* (*repc* ‘*Nodes* (*Suc nb*) *levellista*) (*repc* ∝ *low*) (*repc* ∝ *high*); *t* ∉ *Dags* (*repc* ‘*Nodes nb levellista*) (*repc* ∝ *low*) (*repc* ∝ *high*);

ordered t var; ∀ *no* ∈ *Nodes* (*Suc nb*) *levellista*. *var* (*repc no*) <= *var no* ∧ *repc* (*repc no*) = *repc no*; *wf-ll pret levellista var*; *nb* < *length levellista*;*repc* ‘*Nodes* (*Suc nb*) *levellista* ⊆ *Nodes* (*Suc nb*) *levellista*⟧
⟹ ∃ *q*. *Dag* (*repc q*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t* ∧ *q* ∈ *set* (*levellista* ! *nb*)
proof (*elim Dags.cases*)
  fix *p* **and** *ta* :: *dag*
  assume *t-notin-DagsNodesnb*: *t* ∉ *Dags* (*repc* ‘ *Nodes nb levellista*) (*repc* ∝ *low*) (*repc* ∝ *high*)
  assume *t-ta*: *t* = *ta*
  assume *ta-in-repc-NodesSucnb*: *set-of ta* ⊆ *repc* ‘ *Nodes* (*Suc nb*) *levellista*
  assume *ta-dag*: *Dag p* (*repc* ∝ *low*) (*repc* ∝ *high*) *ta*
  assume *ta-nTip*: *ta* ≠ *Tip*
  assume *ord-t*: *ordered t var*
  assume *varrep-prop*: ∀ *no* ∈ *Nodes* (*Suc nb*) *levellista*. *var* (*repc no*) <= *var no* ∧ *repc* (*repc no*) = *repc no*
  assume *wf-lla*: *wf-ll pret levellista var*
  assume *nbslla*: *nb* < *length levellista*
  assume *repcNodes-in-Nodes*: *repc* ‘*Nodes* (*Suc nb*) *levellista* ⊆ *Nodes* (*Suc nb*) *levellista*
  from *ta-nTip ta-dag* **have** *p-nNull*: *p*≠ *Null*
    by *auto*
  with *ta-nTip ta-dag* **obtain** *lt rt* **where** *ta-Node*: *ta* = *Node lt p rt*
    by *auto*
  with *ta-nTip ta-dag* **have** *p-in-ta*: *p* ∈ *set-of ta*
    by *auto*
  with *ta-in-repc-NodesSucnb* **have** *p-in-repcNodes-Sucnb*: *p* ∈ *repc* ‘*Nodes* (*Suc nb*) *levellista*
    by *auto*
  show *?thesis*
    proof (*cases p* ∈ *repc* ‘(*set* (*levellista* ! *nb*)))
      case *True*
      then obtain *q* **where**
        p-repca: *p*=*repc q* **and**
        a-in-llanb: *q* ∈ *set* (*levellista* ! *nb*)
        by *auto*
      with *ta-dag t-ta* **show** *?thesis*
        apply −
        apply (*rule-tac x*=*q* **in** *exI*)
        apply *simp*
        done
    next
      assume *p-notin-repc-llanb*: *p* ∉ *repc* ‘ *set* (*levellista* ! *nb*)
        with *p-in-repcNodes-Sucnb* **have** *p-in-repc-Nodesnb*: *p* ∈ *repc* ‘*Nodes nb levellista*
        apply −
        apply (*erule imageE*)
        apply *rule*
        apply (*simp add*: *Nodes-def*)
        apply (*simp add*: *Nodes-def*)

apply (*erule exE conjE*)
apply (*case-tac xa=nb*)
apply *simp*
apply (*rule-tac x=xa* **in** *exI*)
apply *auto*
done
have *t* ∈ *Dags* (*repc* '*Nodes nb levellista*) (*repc* ∝ *low*) (*repc* ∝ *high*)
proof −
have *set-of t* ⊆ *repc* '*Nodes nb levellista*
proof (*rule*)
fix *x* :: *ref*
assume *x-in-t*: *x* ∈ *set-of t*
with *ord-t* **have** *var x* <= *var* (*root t*)
apply −
apply (*rule ordered-set-of*)
apply *auto*
done
with *t-ta ta-Node* **have** *varx-varp*: *var x* <= *var p*
by *auto*
from *p-in-repc-Nodesnb* **obtain** *k* **where** *ksnb*: *k* < *nb* **and** *p-in-repc-llak*: *p* ∈ *repc* '(*set* (*levellista* ! *k*))
by (*auto simp add*: *Nodes-def ImageE*)
then obtain *q* **where** *p-repcq*: *p=repc q* **and** *q-in-llak*: *q* ∈ *set* (*levellista* ! *k*)
by *auto*
from *q-in-llak wf-lla nbslla ksnb* **have** *varqk*: *var q* = *k*
by (*simp add*: *wf-ll-def*)
have *Nodesnb-in-NodesSucnb*: *Nodes nb levellista* ⊆ *Nodes* (*Suc nb*) *levellista*
by (*rule Nodes-subset*)
from *q-in-llak ksnb* **have** *q* ∈ *Nodes nb levellista*
by (*auto simp add*: *Nodes-def*)
with *varrep-prop Nodesnb-in-NodesSucnb* **have** *var* (*repc q*) <= *var q*
by *auto*
with *varqk ksnb p-repcq* **have** *var p* < *nb*
by *auto*
with *varx-varp* **have** *varx-snb*: *var x* < *nb*
by *auto*
from *x-in-t t-ta ta-in-repc-NodesSucnb* **obtain** *a* **where**
x-repca: *x*= *repc a* **and**
a-in-NodesSucnb: *a* ∈ *Nodes* (*Suc nb*) *levellista*
by *auto*
with *varrep-prop* **have** *rx-x*: *repc x* = *x*
by *auto*
have *x* ∈ *set-of pret*
proof −
from *wf-lla nbslla* **have** *Nodes* (*Suc nb*) *levellista* ⊆ *set-of pret*
apply −
apply (*rule Nodes-in-pret*)

apply *auto*
done
with *x-in-t t-ta ta-in-repc-NodesSucnb repcNodes-in-Nodes* **show** *?thesis*
by *auto*
qed
with *wf-lla* **have** $x \in$ *set* (*levellista* ! (*var x*))
by (*auto simp add*: *wf-ll-def*)
with *varx-snb* **have** $x \in$ *Nodes nb levellista*
by (*auto simp add*: *Nodes-def*)
with *rx-x* **show** $x \in$ *repc* '*Nodes nb levellista*
apply −
apply *rule*
apply (*subgoal-tac x=repc x*)
apply *auto*
done
qed
with *ta-nTip ta-dag t-ta* **show** *?thesis*
apply −
apply (*rule DagsI*)
apply *auto*
done
qed
with *t-notin-DagsNodesnb* **show** *?thesis*
by *auto*
qed
qed

lemma *same-bdt-var*: ⟦*bdt* (*Node lt1 p1 rt1*) *var* = *Some bdt1*; *bdt* (*Node lt2 p2 rt2*) *var* = *Some bdt1*⟧
⟹ *var p1* = *var p2*
proof (*induct bdt1*)
case *Zero*
then obtain *var-p1*: *var p1* = *0* **and** *var-p2*: *var p2* = *0*
by *simp*
then show *?case*
by *simp*
next
case *One*
then obtain *var-p1*: *var p1* = *1* **and** *var-p2*: *var p2* = *1*
by *simp*
then show *?case*
by *simp*
next
case (*Bdt-Node lbdt v rbdt*)
then obtain *var-p1*: *var p1* = *v* **and** *var-p2*: *var p2* = *v*

```
    by simp
  then show ?case by simp
qed

lemma bdt-Some-Leaf-var-le-1:
⟦Dag p low high t; bdt t var = Some x; isLeaf-pt p low high⟧
  ⟹ var p <= 1
proof (induct t)
  case Tip
  thus ?case by simp
next
  case (Node lt p rt)
  note tNode=this
  from Node.prems tNode show ?case
    apply (simp add: isLeaf-pt-def)
    apply (case-tac var p = 0)
    apply simp
    apply (case-tac var p = Suc 0)
    apply simp
    apply simp
    done
qed

lemma subnode-dag-cons:
⋀ p. ⟦Dag p low high t; no ∈ set-of t⟧ ⟹ ∃ not. Dag no low high not
proof (induct t)
  case Tip
  thus ?case by simp
next
  case (Node lt q rt)
  with Node.prems have q-p: p = q
    by simp
  from Node.prems have lt-dag: Dag (low p) low high lt
    by auto
  from Node.prems have rt-dag: Dag (high p) low high rt
    by auto
  show ?case
  proof (cases no ∈ set-of lt)
    case True
    with Node.hyps lt-dag show ?thesis
      by simp
  next
    assume no-notin-lt: no ∉ set-of lt
    show ?thesis
    proof (cases no=p)
      case True
      with Node.prems q-p show ?thesis
        by auto
    next
```

assume *no-neq-p*: $no \neq p$
with *Node.prems no-notin-lt* **have** *no-in-rt*: $no \in$ *set-of rt*
by *simp*
with *rt-dag Node.hyps* **show** *?thesis*
by *auto*
qed
qed
qed

lemma *nodes-in-taken-in-takeSucn*: $no \in set\ (take\ n\ nodeslist) \Longrightarrow no \in set\ (take\ (Suc\ n)\ nodeslist)$
proof −
assume *no-in-taken*: $no \in set\ (take\ n\ nodeslist)$
have $set\ (take\ n\ nodeslist) \subseteq set\ (take\ (Suc\ n)\ nodeslist)$
apply −
apply (*rule set-take-subset-set-take*)
apply *simp*
done
with *no-in-taken* **show** *?thesis*
by *blast*
qed

lemma *ind-in-higher-take*: $\bigwedge n\ k.\ [\![n < k;\ \ n < length\ xs]\!] \Longrightarrow xs\ !\ n \in set\ (take\ k\ xs)$
apply (*induct xs*)
apply *simp*
apply *simp*
apply (*case-tac n*)
apply *simp*
apply (*case-tac k*)
apply *simp*
apply *simp*
apply *simp*
apply (*case-tac k*)
apply *simp*
apply *simp*
done

lemma *take-length-set*: $\bigwedge n.\ n{=}length\ xs \Longrightarrow set\ (take\ n\ xs) = set\ xs$
apply (*induct xs*)
apply (*auto simp add*: *take-Cons split*: *nat.splits*)
done

lemma *repNodes-eq-ext-rep*: $[\![low\ no \neq nodeslist!\ n;\ high\ no \neq nodeslist\ !\ n;$
$low\ sn \neq nodeslist\ !\ n;\ high\ sn \neq nodeslist\ !\ n]\!]$
$\Longrightarrow repNodes\text{-}eq\ sn\ no\ low\ high\ repa = repNodes\text{-}eq\ sn\ no\ low\ high\ (repa(nodeslist$
$!\ n := repa\ (low\ (nodeslist\ !\ n))))$
by (*simp add*: *repNodes-eq-def null-comp-def*)

lemma *filter-not-empty*: $[\![x \in set\ xs;\ P\ x]\!] \Longrightarrow filter\ P\ xs \neq []$
by (*induct xs*) *auto*

lemma $x \in set\ (filter\ P\ xs) \Longrightarrow P\ x$
by *auto*

lemma *hd-filter-in-list*: $filter\ P\ xs \neq [] \Longrightarrow hd\ (filter\ P\ xs) \in set\ xs$
by (*induct xs*) *auto*

lemma *hd-filter-in-filter*: $filter\ P\ xs \neq [] \Longrightarrow hd\ (filter\ P\ xs) \in set\ (filter\ P\ xs)$
by (*induct xs*) *auto*

lemma *hd-filter-prop*:
assumes *non-empty*: $filter\ P\ xs \neq []$
shows $P\ (hd\ (filter\ P\ xs))$
proof −
from *non-empty* **have** $hd\ (filter\ P\ xs) \in set\ (filter\ P\ xs)$
by (*rule hd-filter-in-filter*)
thus *?thesis*
by *auto*
qed

lemma *index-elem*: $x \in set\ xs \Longrightarrow \exists\, i{<}length\ xs.\ x = xs\ !\ i$
apply (*induct xs*)
apply *simp*
apply (*case-tac x=a*)
apply *auto*
done

lemma *filter-hd-P-rep-indep*:
$[\![\forall\, x.\ P\ x\ x;\ \forall\, a\ b.\ P\ x\ a \longrightarrow P\ a\ b \longrightarrow P\ x\ b;\ filter\ (P\ x)\ xs \neq []]\!] \Longrightarrow$
$hd\ (filter\ (P\ (hd\ (filter\ (P\ x)\ xs)))\ xs) = hd\ (filter\ (P\ x)\ xs)$
apply (*induct xs*)
apply *simp*
apply (*case-tac P x a*)
using [[*simp-depth-limit=2*]]

apply (*simp*)
apply *clarsimp*
apply (*fastforce dest*: *hd-filter-prop*)
done

lemma *take-Suc-not-last*:
$\bigwedge n.$ $[\![x \in set\ (take\ (Suc\ n)\ xs);\ x{\neq}xs!n;\ n < length\ xs]\!] \Longrightarrow x \in set\ (take\ n\ xs)$
apply (*induct xs*)
apply *simp*
apply (*case-tac n*)
apply *simp*
using [[*simp-depth-limit=2*]]
apply *fastforce*
done

lemma *P-eq-list-filter*: $\forall x \in set\ xs.\ P\ x = Q\ x \Longrightarrow filter\ P\ xs = filter\ Q\ xs$
apply (*induct xs*)
apply *auto*
done

lemma *hd-filter-take-more*: $\bigwedge n\ m.[\![filter\ P\ (take\ n\ xs) \neq [];\ n \leq m]\!] \Longrightarrow$
$hd\ (filter\ P\ (take\ n\ xs)) = hd\ (filter\ P\ (take\ m\ xs))$
apply (*induct xs*)
apply *simp*
apply (*case-tac n*)
apply *simp*
apply (*case-tac m*)
apply *simp*
apply *clarsimp*
done

end

4 Definitions of Procedures

theory *ProcedureSpecs*
imports *General Simpl.Vcg*
begin

record *globals* =
var-' :: *ref* ⇒ *nat*
low-' :: *ref* ⇒ *ref*
high-' :: *ref* ⇒ *ref*
rep-' :: *ref* ⇒ *ref*
mark-' :: *ref* ⇒ *bool*
next-' ::*ref* ⇒ *ref*

```
record 'g bdd-state = 'g state +
  varval-' :: bool list
  p-' :: ref
  R-' :: bool
  levellist-' :: ref list
  nodeslist-' :: ref
  node-':: ref
  m-' :: bool
  n-' :: nat
```

```
procedures
  Eval (p, varval | R) =
    IF (´p→´var = 0) THEN ´R :==False
     ELSE IF (´p→´var = 1) THEN ´R :==True
       ELSE IF (´varval ! (´p→´var)) THEN CALL Eval (´p→´high, ´varval, ´R)
         ELSE CALL Eval (´p→´low, ´varval, ´R)
         FI
       FI
     FI
```

```
procedures
  Levellist (p, m, levellist | levellist) =
    IF (´p ≠ Null)
     THEN
       IF (´p → ´mark ≠ ´m)
       THEN
         ´levellist :== CALL Levellist ( ´p → ´low, ´m, ´levellist );;
         ´levellist :== CALL Levellist ( ´p → ´high, ´m, ´levellist );;
         ´p → ´next :== ´levellist ! (´p→´var);;
         ´levellist ! (´p→´var) :== ´p;;
         ´p → ´mark :==´m
       FI
     FI
```

```
procedures
  ShareRep (nodeslist, p) =
  IF (isLeaf-pt ´p ´low ´high)
   THEN ´p → ´rep :== ´nodeslist
   ELSE
        WHILE (´nodeslist ≠ Null) DO
```

```
        IF (repNodes-eq ´nodeslist ´p ´low ´high ´rep)
        THEN ´p→´rep :== ´nodeslist;; ´nodeslist :== Null
        ELSE ´nodeslist :== ´nodeslist→´next
        FI
      OD
  FI
```

procedures

```
 ShareReduceRepList (nodeslist | ) =
 ´node :== ´nodeslist;;
  WHILE (´node ≠ Null) DO
    IF (¬ isLeaf-pt ´node ´low ´high ∧
         ´node → ´low → ´rep = ´node → ´high → ´rep )
    THEN ´node → ´rep :== ´node → ´low → ´rep
    ELSE CALL ShareRep (´nodeslist , ´node )
    FI;;
    ´node :==´node → ´next
  OD
```

procedures

```
 Repoint (p|p) =
 IF ( ´p ≠ Null )
  THEN
    ´p :== ´p → ´rep;;
    IF ( ´p ≠ Null )
    THEN ´p → ´low :== CALL Repoint (´p → ´low);;
         ´p → ´high :== CALL Repoint (´p → ´high)
    FI
  FI
```

procedures

```
 Normalize (p|p) =
 ´levellist :==replicate (´p→´var +1) Null;;
  ´levellist :== CALL Levellist (´p, (¬ ´p→´mark) , ´levellist);;
 (´n :==0;;
  WHILE (´n < length ´levellist) DO
    CALL ShareReduceRepList(´levellist ! ´n);;
    ´n :==´n + 1
  OD);;
  ´p :== CALL Repoint (´p)
```

end

5 Proof of Procedure Eval

theory *EvalProof* **imports** *ProcedureSpecs* **begin**

lemma (**in** *Eval-impl*) *Eval-modifies*:
  shows $\forall \sigma.\ \Gamma\vdash\{\sigma\}$ *PROC Eval* (´*p*, ´*varval*, ´*R*)
  {*t*. *t may-not-modify-globals* σ}
  apply (*hoare-rule HoarePartial.ProcRec1*)
  apply (*vcg spec=modifies*)
  done

lemma (**in** *Eval-impl*) *Eval-spec*:
  shows ∀ σ *t bdt1*. Γ⊢
  {|σ. *Dag* ´*p* ´*low* ´*high t* ∧ *bdt t* ´*var* = *Some bdt1*|}
  ´*R* :== *PROC Eval*(´*p*, ´*varval*)
  {|´*R* = *eval bdt1* $^\sigma$*varval* |}
apply (*hoare-rule HoarePartial.ProcRec1*)
apply *vcg*
apply *clarsimp*
apply *safe*
apply (*case-tac bdt1*)
apply *simp*
apply *fastforce*
apply *fastforce*
apply *simp*
apply (*case-tac bdt1*)
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply (*case-tac bdt1*)
apply *fastforce*
apply *fastforce*
apply *fastforce*
apply (*case-tac bdt1*)
apply *fastforce*
apply *fastforce*
apply *fastforce*
done

end

6 Proof of Procedure Levellist

theory *LevellistProof* **imports** *ProcedureSpecs Simpl.HeapList* **begin**

hide-const (**open**) *DistinctTreeProver.set-of tree.Node tree.Tip*

lemma (**in** *Levellist-impl*) *Levellist-modifies*:
  shows ∀ σ. Γ⊢{σ} ´*levellist* :== *PROC Levellist* (´*p*, ´*m*, ´*levellist*)
      {*t*. *t may-only-modify-globals σ in* [*mark*,*next*]}
  apply (*hoare-rule HoarePartial.ProcRec1*)
  apply (*vcg spec=modifies*)
  done

lemma *all-stop-cong*: (∀ *x*. *P x*) = (∀ *x*. *P x*)
  by *simp*

lemma *Dag-RefD*:
  ⟦*Dag p l r t*; *p*≠*Null*⟧ ⟹
    ∃ *lt rt*. *t*=*Node lt p rt* ∧ *Dag* (*l p*) *l r lt* ∧ *Dag* (*r p*) *l r rt*
  by *simp*

lemma *Dag-unique-ex-conjI*:
  ⟦*Dag p l r t*; *P t*⟧ ⟹ (∃ *t*. *Dag p l r t* ∧ *P t*)
  by *simp*

lemma *dag-Null* [*simp*]: *dag Null l r* = *Tip*
  by (*simp add*: *dag-def*)

definition *first*:: *ref list* ⇒ *ref* **where**
first ps = (*case ps of* [] ⇒ *Null* | (*p*#*rs*) ⇒ *p*)

lemma *first-simps* [*simp*]:
first [] = *Null*
first (*r*#*rs*) = *r*
by (*simp-all add*: *first-def*)

definition *Levellist*:: *ref list* ⇒ (*ref* ⇒ *ref*) ⇒ (*ref list list*) ⇒ *bool* **where**
Levellist hds next ll ⟷ (*map first ll* = *hds*) ∧
      (∀ *i* < *length hds*. *List* (*hds* ! *i*) *next* (*ll*!*i*))

lemma *Levellist-unique*:
  assumes *ll*: *Levellist hds next ll*
  assumes *ll'*: *Levellist hds next ll'*
  shows *ll*=*ll'*
proof −
  from *ll* **have** *length ll* = *length hds*
    by (*clarsimp simp add*: *Levellist-def*)
  moreover
  from *ll'* **have** *length ll'* = *length hds*
    by (*clarsimp simp add*: *Levellist-def*)

ultimately have *leq*: *length ll* = *length ll′* **by** *simp*
show *?thesis*
proof (*rule nth-equalityI* [*OF leq*, *rule-format*])
fix *i*
assume *i* < *length ll*
with *ll ll′*
show *ll*!*i* = *ll′*!*i*
apply (*clarsimp simp add*: *Levellist-def*)
apply (*erule-tac x*=*i* **in** *allE*)
apply (*erule-tac x*=*i* **in** *allE*)
apply *simp*
by (*erule List-unique*)
qed
qed

lemma *Levellist-unique-ex-conj-simp* [*simp*]:
Levellist hds next ll ⟹ (∃ *ll*. *Levellist hds next ll* ∧ *P ll*) = *P ll*
by (*auto dest*: *Levellist-unique*)

lemma *in-set-concat-idx*:
x ∈ *set* (*concat xss*) ⟹ ∃ *i* < *length xss*. *x* ∈ *set* (*xss*!*i*)
apply (*induct xss*)
apply *simp*
apply *clarsimp*
apply (*erule disjE*)
apply (*rule-tac x*=*0* **in** *exI*)
apply *simp*
apply *auto*
done

definition *wf-levellist* :: *dag* ⇒ *ref list list* ⇒ *ref list list* ⇒
(*ref* ⇒ *nat*) ⇒ *bool* **where**
wf-levellist t levellist-old levellist-new var =
(*case t of Tip* ⇒ *levellist-old* = *levellist-new*
| (*Node lt p rt*) ⇒
(∀ *q*. *q* ∈ *set-of t* ⟶ *q* ∈ *set* (*levellist-new* ! (*var q*))) ∧
(∀ *i* ≤ *var p*. (∃ *prx*. (*levellist-new* ! *i*) = *prx*@(*levellist-old* ! *i*)
∧ (∀ *pt* ∈ *set prx*. *pt* ∈ *set-of t* ∧ *var pt* = *i*))) ∧
(∀ *i*. (*var p*) < *i* ⟶ (*levellist-new* ! *i*) = (*levellist-old* ! *i*)) ∧
(*length levellist-new* = *length levellist-old*))

lemma *wf-levellist-subset*:
assumes *wf-ll*: *wf-levellist t ll ll′ var*
shows *set* (*concat ll′*) ⊆ *set* (*concat ll*) ∪ *set-of t*
proof (*cases t*)
case *Tip* **with** *wf-ll* **show** *?thesis* **by** (*simp add*: *wf-levellist-def*)
next
case (*Node lt p rt*)

```
show ?thesis
proof -
  {
    fix n
    assume n ∈ set (concat ll')
    from in-set-concat-idx [OF this]
    obtain i where i-bound: i < length ll' and n-in: n ∈ set (ll' ! i)
      by blast
    have n ∈ set (concat ll) ∪ set-of t
    proof (cases i ≤ var p)
      case True
      with wf-ll obtain prx where
        ll'-ll: ll' ! i = prx @ ll ! i and
        prx: ∀ pt ∈ set prx. pt ∈ set-of t  and
        leq: length ll' = length ll
        apply (clarsimp simp add: wf-levellist-def Node)
        apply (erule-tac x=i in allE)
        apply clarsimp
        done
      show ?thesis
      proof (cases n ∈ set prx)
        case True
        with prx have n ∈ set-of t
          by simp
        thus ?thesis by simp
      next
        case False
        with n-in ll'-ll
        have n ∈ set (ll ! i)
          by simp
        with i-bound leq
        have n ∈ set (concat ll)
          by auto
        thus ?thesis by simp
      qed
    next
      case False
      with wf-ll obtain ll'!i = ll!i length ll' = length ll
        by (auto simp add: wf-levellist-def Node)
      with n-in i-bound
      have n ∈ set (concat ll)
        by auto
      thus ?thesis by simp
    qed
  }
  thus ?thesis by auto
qed
qed
```

```
lemma Levellist-ext-to-all: ((∃ ll. Levellist hds next ll ∧ P ll) ⟶ Q)
      =
      (∀ ll. Levellist hds next ll ∧ P ll ⟶ Q)
apply blast
done

lemma Levellist-length: Levellist hds p ll ⟹ length ll = length hds
  by (auto simp add: Levellist-def)


lemma map-update:
  ⋀i. i < length xss ⟹ map f (xss[i := xs]) = (map f xss) [i := f xs]
apply (induct xss)
apply  simp
apply (case-tac i)
apply  simp
apply simp
done


lemma (in Levellist-impl) Levellist-spec-total':
shows ∀ ll σ t. Γ,Θ⊢t
       {|σ. Dag ´p ´low ´high t ∧ (´p ≠ Null ⟶ (´p→´var) < length ´levellist) ∧
            ordered t ´var ∧ Levellist ´levellist ´next ll ∧
            (∀ n ∈ set-of t.
             (if ´mark n = ´m
              then n ∈ set (ll ! ´var n) ∧
                   (∀ nt p. Dag n ´low ´high nt ∧ p ∈ set-of nt
                    ⟶ ´mark p = ´m)
              else n ∉ set (concat ll)))|}
         ´levellist :== PROC Levellist (´p, ´m, ´levellist)
       {|∃ ll'. Levellist ´levellist ´next ll' ∧ wf-levellist t ll ll' σvar ∧
        wf-marking t σmark ´mark σm ∧
        (∀ p. p ∉ set-of t ⟶ σnext p = ´next p)
        |}
apply (hoare-rule HoareTotal.ProcRec1
          [where r=measure (λ(s,p). size (dag sp slow shigh))])
apply vcg
apply (rule conjI)
apply  clarify
apply  (rule conjI)
apply   clarify
apply   (clarsimp simp del: BinDag.set-of.simps split del: if-split)
defer
apply   (rule impI)
```

```
apply  (clarsimp simp del: BinDag.set-of.simps split del: if-split)
defer
apply  (clarsimp simp add: wf-levellist-def wf-marking-def)
apply (simp only: Levellist-ext-to-all )
proof −
  fix ll var low high mark next nexta p levellist m lt rt
  assume pnN: p ≠ Null
  assume mark-p: mark p = (¬ m)
  assume lt: Dag (low p) low high lt
  assume rt: Dag (high p) low high rt
  from pnN lt rt have Dag-p: Dag p low high (Node lt p rt) by simp
  from Dag-p rt
  have size-rt-dec: size (dag (high p) low high) < size (dag p low high)
    by (simp only: Dag-dag) simp
  from Dag-p lt
  have size-lt-dec: size (dag (low p) low high) < size (dag p low high)
    by (simp only: Dag-dag) simp
  assume ll: Levellist levellist next ll

  assume marked-child-ll:
    ∀ n ∈ set-of (Node lt p rt).
        if mark n = m
        then n ∈ set (ll ! var n) ∧
             (∀ nt p. Dag n low high nt ∧ p ∈ set-of nt ⟶ mark p = m)
        else n ∉ set (concat ll)
  with mark-p have p-notin-ll: p ∉ set (concat ll)
    by auto
  assume varsll′: var p < length levellist
  with ll have varsll: var p < length ll
    by (simp add: Levellist-length)
  assume orderedt: ordered (Node lt p rt) var
  show (low p ≠ Null ⟶ var (low p) < length levellist) ∧
          ordered lt var ∧
          (∀ n ∈ set-of lt.
              if mark n = m
              then n ∈ set (ll ! var n) ∧
                   (∀ nt p. Dag n low high nt ∧ p ∈ set-of nt ⟶ mark p = m)
              else n ∉ set (concat ll)) ∧
          size (dag (low p) low high) < size (dag p low high) ∧
          (∀ marka nexta levellist lla.
              Levellist levellist nexta lla ∧
              wf-levellist lt ll lla var ∧ wf-marking lt mark marka m ∧
              (∀ p. p ∉ set-of lt ⟶ next p = nexta p)⟶
              (high p ≠ Null ⟶ var (high p) < length levellist) ∧
              ordered rt var ∧
              (∃ lla. Levellist levellist nexta lla ∧
                     (∀ n ∈ set-of rt.
                        if marka n = m
                        then n ∈ set (lla ! var n) ∧
```

(∀ *nt p*. *Dag n low high nt* ∧ *p* ∈ *set-of nt* ⟶
marka p = *m*)
else n ∉ *set* (*concat lla*)) ∧
size (*dag* (*high p*) *low high*) < *size* (*dag p low high*) ∧
(∀ *markb nextb levellist llb*.
Levellist levellist nextb llb ∧
wf-levellist rt lla llb var ∧
wf-marking rt marka markb m ∧
(∀ *p*. *p* ∉ *set-of rt* ⟶ *nexta p* = *nextb p*) ⟶
(∃ *ll'*. *Levellist* (*levellist*[*var p* := *p*])
(*nextb*(*p* := *levellist* ! *var p*)) *ll'* ∧
wf-levellist (*Node lt p rt*) *ll ll' var* ∧
wf-marking (*Node lt p rt*) *mark* (*markb*(*p* := *m*)) *m* ∧
(∀ *pa*. *pa* ∉ *set-of* (*Node lt p rt*) ⟶
next pa =
(*if pa* = *p then levellist* ! *var p*
else nextb pa))))))

proof (*cases lt*)
case *Tip*
note *lt-Tip* = *this*
show *?thesis*
proof (*cases rt*)
case *Tip*
show *?thesis*
using *size-rt-dec Tip lt-Tip Tip lt rt*
apply *clarsimp*
subgoal premises *prems* **for** *marka nexta levellista lla markb nextb levellistb llb*
proof −
have *lla*: *Levellist levellista nexta lla* **by** *fact*
have *llb*: *Levellist levellistb nextb llb* **by** *fact*
have *wfll-lt*: *wf-levellist Tip ll lla var*
wf-marking Tip mark marka m **by** *fact*+

then have *ll-lla*: *ll* = *lla*
by (*simp add*: *wf-levellist-def*)

moreover
with *wfll-lt lt-Tip lt* **have** *marka* = *mark*
by (*simp add*: *wf-marking-def*)
moreover
have *wfll-rt*:*wf-levellist Tip lla llb var*
wf-marking Tip marka markb m **by** *fact*+
then have *lla-llb*: *lla* = *llb*
by (*simp add*: *wf-levellist-def*)
moreover
with *wfll-rt Tip rt* **have** *markb* = *marka*
by (*simp add*: *wf-marking-def*)
moreover

```
      from varsll llb ll-lla lla-llb
      obtain var p < length levellistb var p < length llb
        by (simp add: Levellist-length)
      with llb pnN
      have llc: Levellist (levellistb[var p := p]) (nextb(p := levellistb ! var p))
                  (llb[var p := p # llb ! var p])
        apply (clarsimp simp add: Levellist-def map-update)
        apply (erule-tac x=i in allE)
        apply clarsimp
        apply (subgoal-tac p ∉ set (llb ! i) )
        prefer 2
        using  p-notin-ll ll-lla lla-llb
        apply  simp
        apply (case-tac i=var p)
        apply  simp
        apply simp
        done
      ultimately
      show ?thesis
        using lt-Tip Tip varsll
        apply (clarsimp simp add: wf-levellist-def wf-marking-def)
      proof −
        fix i
        assume varsllb: var p < length llb
        assume i ≤ var p
        show ∃ prx. llb[var p := p#llb!var p]!i = prx @ llb!i ∧
                  (∀ pt∈set prx. pt = p ∧ var pt = i)
        proof (cases i = var p)
          case True
          with pnN lt rt varsllb lt-Tip Tip show ?thesis
            apply −
            apply (rule-tac x=[p] in exI)
            apply (simp add: subdag-eq-def)
            done
        next
          assume i ≠ var p
          with varsllb show ?thesis
            apply −
            apply (rule-tac x=[] in exI)
            apply (simp add: subdag-eq-def)
            done
        qed
      qed
    qed
  done
next
  case (Node dag1 a dag2)
  have rt-node: rt = Node dag1 a dag2 by fact
  with rt have high-p: high p = a
```

```
    by simp
  have s: ⋀nexta. (∀ p. next p = nexta p) = (next = nexta)
    by auto
  show ?thesis
    using size-rt-dec size-lt-dec rt-node lt-Tip Tip lt rt
    apply (clarsimp simp del: set-of-Node split del: if-split simp add: s)
    subgoal premises prems for marka levellista lla
    proof −
      have lla: Levellist levellista next lla by fact
      have wfll-lt:wf-levellist Tip ll lla var
                 wf-marking Tip mark marka m by fact+
      from this have ll-lla: ll = lla
        by (simp add: wf-levellist-def)
      moreover
      from wfll-lt lt-Tip lt have marklrec: marka = mark
        by (simp add: wf-marking-def)
      from orderedt varsll lla ll-lla rt-node lt-Tip high-p
      have var-highp-bound: var (high p) < length levellista
        by (auto simp add: Levellist-length)
      from orderedt high-p rt-node lt-Tip
      have ordered-rt: ordered (Node dag1 (high p) dag2) var
        by simp
      from high-p marklrec marked-child-ll lt rt lt-Tip rt-node ll-lla
      have mark-rt: (∀ n∈set-of (Node dag1 (high p) dag2).
            if marka n = m
            then n ∈ set (lla ! var n) ∧
                 (∀ nt p. Dag n low high nt ∧ p ∈ set-of nt ⟶ marka p = m)
            else n ∉ set (concat lla))
        apply (simp only: BinDag.set-of.simps)
        apply clarify
        apply (drule-tac x=n in bspec)
        apply  blast
        apply assumption
        done
      show ?thesis
        apply (rule conjI)
        apply  (rule var-highp-bound)
        apply (rule conjI)
        apply  (rule ordered-rt)
        apply (rule conjI)
        apply  (rule mark-rt)
        apply clarify
        apply clarsimp
        subgoal premises prems for markb nextb levellistb llb
        proof −
          have llb: Levellist levellistb nextb llb by fact
          have wfll-rt: wf-levellist (Node dag1 (high p) dag2) lla llb var by fact
        have wfmarking-rt: wf-marking (Node dag1 (high p) dag2) marka markb
m by fact
```

from *wfll-rt varsll llb ll-lla*
obtain *var-p-bounds*: *var p* < *length levellistb var p* < *length llb*
  by (*simp add*: *Levellist-length wf-levellist-def*)
with *p-notin-ll ll-lla wfll-rt*
have *p-notin-llb*: ∀ *i* < *length llb*. *p* ∉ *set* (*llb* ! *i*)
  apply −
  apply (*intro allI impI*)
  apply (*clarsimp simp add*: *wf-levellist-def*)
  apply (*case-tac i* ≤ *var* (*high p*))
  apply (*drule-tac x*=*i* **in** *spec*)
  using *orderedt rt-node lt-Tip high-p*
  apply *clarsimp*
  apply (*drule-tac x*=*i* **in** *spec*)
  apply (*drule-tac x*=*i* **in** *spec*)
  apply *clarsimp*
  done
with *llb pnN var-p-bounds*
have *llc*: *Levellist* (*levellistb*[*var p* := *p*])
    (*nextb*(*p* := *levellistb* ! *var p*))
    (*llb*[*var p* := *p* # *llb* ! *var p*])
  apply (*clarsimp simp add*: *Levellist-def map-update*)
  apply (*erule-tac x*=*i* **in** *allE*)
  apply (*erule-tac x*=*i* **in** *allE*)
  apply *clarsimp*
  apply (*case-tac i*=*var p*)
  apply *simp*
  apply *simp*
  done
then show *?thesis*
  apply *simp*
  using *wfll-rt wfmarking-rt*
    lt-Tip rt-node varsll orderedt lt rt pnN ll-lla marklrec
  apply (*clarsimp simp add*: *wf-levellist-def wf-marking-def*)
  apply (*intro conjI*)
  apply (*rule allI*)
  apply (*rule conjI*)
  apply (*erule-tac x*=*q* **in** *allE*)
  apply (*case-tac var p* = *var q*)
  apply *fastforce*
  apply *fastforce*
  apply (*case-tac var p* = *var q*)
  apply *hypsubst-thin*
  apply *fastforce*
  apply *fastforce*
  apply (*rule allI*)
  apply (*rotate-tac 4*)
  apply (*erule-tac x*=*i* **in** *allE*)
  apply (*case-tac i*=*var p*)
  apply *simp*

apply (*case-tac var* (*high p*) < *i*)
apply *simp*
apply *simp*
apply (*erule exE*)
apply (*rule-tac x=prx* **in** *exI*)
apply (*intro conjI*)
apply *simp*
apply *clarify*
apply (*rotate-tac 15*)
apply (*erule-tac x=pt* **in** *ballE*)
apply *fastforce*
apply *fastforce*
done
qed
done
qed
done
qed
next
case (*Node llt l rlt*)
have *lt-Node*: *lt* = *Node llt l rlt* **by** *fact*
from *orderedt lt varsll′ lt-Node*
obtain *ordered-lt*:
ordered lt var (*low p* ≠ *Null* ⟶ *var* (*low p*) < *length levellist*)
by (*cases rt*) *auto*
from *lt lt-Node marked-child-ll*
have *mark-lt*: ∀ *n*∈*set-of lt*.
if mark n = *m*
then n ∈ *set* (*ll* ! *var n*) ∧
(∀ *nt p*. *Dag n low high nt* ∧ *p* ∈ *set-of nt* ⟶ *mark p* = *m*)
else n ∉ *set* (*concat ll*)
apply (*simp only*: *BinDag.set-of.simps*)
apply *clarify*
apply (*drule-tac x=n* **in** *bspec*)
apply *blast*
apply *assumption*
done
show *?thesis*
apply (*intro conjI ordered-lt mark-lt size-lt-dec*)
apply (*clarify*)
apply (*simp add*: *size-rt-dec split del*: *if-split*)
apply (*simp only*: *Levellist-ext-to-all*)
subgoal premises *prems* **for** *marka nexta levellista lla*
proof −
have *lla*: *Levellist levellista nexta lla* **by** *fact*
have *wfll-lt*: *wf-levellist lt ll lla var* **by** *fact*
have *wfmarking-lt*:*wf-marking lt mark marka m* **by** *fact*
from *wfll-lt lt-Node*
have *lla-eq-ll*: *length lla* = *length ll*

by (*simp add*: *wf-levellist-def*)
with *ll lla* **have** *lla-eq-ll'*: *length levellista* = *length levellist*
by (*simp add*: *Levellist-length*)
with *orderedt rt lt-Node lt varsll'*
obtain *ordered-rt*:
ordered rt var (*high p* ≠ *Null* ⟶ *var* (*high p*) < *length levellista*)
by (*cases rt*) *auto*
from *wfll-lt lt-Node*
have *nodes-in-lla*: ∀ *q*. *q* ∈ *set-of lt* ⟶ *q* ∈ *set* (*lla* ! (*q*→*var*))
by (*simp add*: *wf-levellist-def*)
from *wfll-lt lt-Node lt*
have *lla-st*: (∀ *i* ≤ (*low p*)→*var*.
(∃ *prx*. (*lla* ! *i*) = *prx*@(*ll* ! *i*) ∧
(∀ *pt* ∈ *set prx*. *pt* ∈ *set-of lt* ∧ *pt*→*var* = *i*)))
by (*simp add*: *wf-levellist-def*)
from *wfll-lt lt-Node lt*
have *lla-nc*: ∀ *i*. ((*low p*)→*var*) < *i* ⟶ (*lla* ! *i*) = (*ll* ! *i*)
by (*simp add*: *wf-levellist-def*)
from *wfmarking-lt lt-Node lt*
have *mot-nc*: ∀ *n*. *n* ∉ *set-of lt* ⟶ *mark n* = *marka n*
by (*simp add*: *wf-marking-def*)
from *wfmarking-lt lt-Node lt*
have *mit-marked*: ∀ *n*. *n* ∈ *set-of lt* ⟶ *marka n* = *m*
by (*simp add*: *wf-marking-def*)
from *marked-child-ll nodes-in-lla mot-nc mit-marked lla-st*
have *mark-rt*: ∀ *n*∈*set-of rt*.
if marka n = *m*
then n ∈ *set* (*lla* ! *var n*) ∧
(∀ *nt p*. *Dag n low high nt* ∧ *p* ∈ *set-of nt* ⟶ *marka p* = *m*)
else n ∉ *set* (*concat lla*)
apply −
apply (*rule ballI*)
apply (*drule-tac x*=*n* **in** *bspec*)
apply (*simp*)
proof −
fix *n*

assume *nodes-in-lla*: ∀ *q*. *q* ∈ *set-of lt* ⟶ *q* ∈ *set* (*lla* ! *var q*)
assume *mot-nc*: ∀ *n*. *n* ∉ *set-of lt* ⟶ *mark n* = *marka n*
assume *mit-marked*: ∀ *n*. *n* ∈ *set-of lt* ⟶ *marka n* = *m*
assume *marked-child-ll*: *if mark n* = *m*
then n ∈ *set* (*ll* ! *var n*) ∧
(∀ *nt p*. *Dag n low high nt* ∧ *p* ∈ *set-of nt* ⟶ *mark p* = *m*)
else n ∉ *set* (*concat ll*)

assume *lla-st*: ∀ *i*≤*var* (*low p*).
∃ *prx*. *lla* ! *i* = *prx* @ *ll* ! *i* ∧
(∀ *pt*∈*set prx*. *pt* ∈ *set-of lt* ∧ *var pt* = *i*)

assume *n-in-rt*: $n \in$ *set-of rt*
show *n-in-lla-marked*: *if marka* $n = m$
  then $n \in$ *set* (*lla* ! *var n*) $\wedge$
    ($\forall$ *nt p. Dag n low high nt* $\wedge$ $p \in$ *set-of nt* $\longrightarrow$ *marka* $p = m$)
  else $n \notin$ *set* (*concat lla*)
proof (*cases* $n \in$ *set-of lt*)
 case *True*
 from *True nodes-in-lla* **have** *n-in-ll*: $n \in$ *set* (*lla* ! *var n*)
  by *simp*
 moreover
 from *True wfmarking-lt*
 have *marka* $n = m$
  apply (*cases lt*)
  apply (*auto simp add*: *wf-marking-def*)
  done
 moreover
 {
  fix *nt p*
  assume *Dag n low high nt*
  with *lt True* **have** *subset-nt-lt*: *set-of nt* $\subseteq$ *set-of lt*
   by (*rule dag-setof-subsetD*)
  moreover assume $p \in$ *set-of nt*
  ultimately have $p \in$ *set-of lt*
   by *blast*
  with *mit-marked* **have** *marka* $p = m$
   by *simp*
 }
 ultimately show *?thesis*
  using *n-in-rt*
  apply *clarsimp*
  done
next
 assume *n-notin-lt*: $n \notin$ *set-of lt*
 show *?thesis*
 proof (*cases marka* $n = m$)
  case *True*
  from *n-notin-lt mot-nc* **have** *marka-eq-mark*: *mark* $n =$ *marka n*
   by *simp*
  from *marka-eq-mark True* **have** *n-marked*: *mark* $n = m$
   by *simp*
  from *rt n-in-rt* **have** *nnN*: $n \neq$ *Null*
   apply −
   apply (*rule set-of-nn* [*rule-format*])
   apply *fastforce*
   apply *assumption*
   done
  from *marked-child-ll n-in-rt marka-eq-mark nnN n-marked*
  have *n-in-ll*: $n \in$ *set* (*ll* ! *var n*)
   by *fastforce*

```
      from marked-child-ll n-in-rt marka-eq-mark nnN n-marked lt rt
      have nt-mark: ∀ nt p. Dag n low high nt ∧ p ∈ set-of nt ⟶ mark p =
m
        by simp
      from nodes-in-lla n-in-ll lla-st
      have n-in-lla: n ∈ set (lla ! var n)
      proof (cases var (low p) < (var n))
        case True
        with lla-nc have (lla ! var n) = (ll ! var n)
          by fastforce
        with n-in-ll show ?thesis
          by fastforce
      next
        assume varnslp: ¬ var (low p) < var n
        with lla-st
        have ll-in-lla: ∃ prx. lla ! (var n) = prx @ ll ! (var n)
          apply −
          apply (erule-tac x=var n in allE)
          apply fastforce
          done
        with n-in-ll show ?thesis
          by fastforce
      qed
      {
        fix nt pt
        assume nt-Dag: Dag n low high nt
        assume pt-in-nt: pt ∈ set-of nt
        have marka pt = m
        proof (cases pt ∈ set-of lt)
          case True
          with mit-marked show ?thesis
            by fastforce
        next
          assume pt-notin-lt: pt ∉ set-of lt
          with mot-nc have mark pt = marka pt
            by fastforce
          with nt-mark nt-Dag pt-in-nt show ?thesis
            by fastforce
        qed
      }
      then have nt-marka:
        ∀ nt pt. Dag n low high nt ∧ pt ∈ set-of nt ⟶ marka pt = m
        by fastforce
      with n-in-lla nt-marka True show ?thesis
        by fastforce
    next
      case False
      note n-not-marka = this
      with wfmarking-lt n-notin-lt
```

have *mark n* $\neq$ *m*
  by (*simp add*: *wf-marking-def lt-Node*)
with *marked-child-ll*
have *n-notin-ll*: *n* $\notin$ *set* (*concat ll*)
  by *simp*
show *?thesis*
proof (*cases n* $\in$ *set* (*concat lla*))
  case *False* **with** *n-not-marka* **show** *?thesis* **by** *simp*
next
  case *True*
  with *wf-levellist-subset* [*OF wfll-lt*] *n-notin-ll*
  have *n* $\in$ *set-of lt*
    by *blast*
  with *n-notin-lt* **have** *False* **by** *simp*
  thus *?thesis* **..**
qed
qed
qed
qed
show *?thesis*
  apply (*intro conjI ordered-rt mark-rt*)
  apply *clarify*
  subgoal premises *prems* **for** *markb nextb levellistb llb*
  proof −
    have *llb*: *Levellist levellistb nextb llb* **by** *fact*
    have *wfll-rt*: *wf-levellist rt lla llb var* **by** *fact*
    have *wfmarking-rt*: *wf-marking rt marka markb m* **by** *fact*
    show *?thesis*
    proof (*cases rt*)
      case *Tip*
      from *wfll-rt Tip* **have** *lla-llb*: *lla* = *llb*
        by (*simp add*: *wf-levellist-def*)
      moreover
      from *wfmarking-rt Tip rt* **have** *markb* = *marka*
        by (*simp add*: *wf-marking-def*)
      moreover
      from *wfll-lt varsll llb lla-llb*
      obtain *var-p-bounds*: *var p* < *length levellistb var p* < *length llb*
        by (*simp add*: *Levellist-length wf-levellist-def lt-Node Tip*)
      with *p-notin-ll lla-llb wfll-lt*
      have *p-notin-llb*: $\forall$ *i* < *length llb*. *p* $\notin$ *set* (*llb* ! *i*)
        apply −
        apply (*intro allI impI*)
        apply (*clarsimp simp add*: *wf-levellist-def lt-Node*)
        apply (*case-tac i* $\leq$ *var l*)
        apply (*drule-tac x*=*i* **in** *spec*)
        using *orderedt Tip lt-Node*
        apply *clarsimp*
        apply (*drule-tac x*=*i* **in** *spec*)

```
    apply (drule-tac x=i in spec)
    apply clarsimp
    done
  with llb pnN var-p-bounds
  have llc: Levellist (levellistb[var p := p])
              (nextb(p := levellistb ! var p))
              (llb[var p := p # llb ! var p])
    apply (clarsimp simp add: Levellist-def map-update)
    apply (erule-tac x=i in allE)
    apply (erule-tac x=i in allE)
    apply clarsimp
    apply (case-tac i=var p)
    apply  simp
    apply simp
    done
  ultimately show ?thesis
    using Tip lt-Node varsll orderedt lt rt pnN wfll-lt wfmarking-lt
    apply (clarsimp simp add: wf-levellist-def wf-marking-def)
    apply (intro conjI)
    apply  (rule allI)
    apply  (rule conjI)
    apply   (erule-tac x=q in allE)
    apply   (case-tac var p = var q)
    apply    fastforce
    apply   fastforce
    apply  (case-tac var p = var q)
    apply   hypsubst-thin
    apply   fastforce
    apply  fastforce
    apply (rule allI)
    apply (rotate-tac 4)
    apply (erule-tac x=i in allE)
    apply (case-tac i=var p)
    apply  simp
    apply (case-tac var (low p) < i)
    apply  simp
    apply simp
    apply (erule exE)
    apply (rule-tac x=prx in exI)
    apply (intro conjI)
    apply  simp
    apply clarify
    apply (rotate-tac 15)
    apply (erule-tac x=pt in ballE)
    apply  fastforce
    apply fastforce
    done
next
  case (Node lrt r rrt)
```

have *rt-Node*: $rt = Node\ lrt\ r\ rrt$ **by** *fact*
from *wfll-rt rt-Node*
have *llb-eq-lla*: *length llb* = *length lla*
  by (*simp add*: *wf-levellist-def*)
with *llb lla*
have *llb-eq-lla'*: *length levellistb* = *length levellista*
  by (*simp add*: *Levellist-length*)
from *wfll-rt rt-Node*
have *nodes-in-llb*: $\forall q.\ q \in set\text{-}of\ rt \longrightarrow q \in set\ (llb\ !\ (q \rightarrow var))$
  by (*simp add*: *wf-levellist-def*)
from *wfll-rt rt-Node rt*
have *llb-st*: $(\forall\ i \le (high\ p) \rightarrow var.$
  $(\exists\ prx.\ (llb\ !\ i) = prx @ (lla\ !\ i) \wedge$
  $(\forall pt \in set\ prx.\ pt \in set\text{-}of\ rt \wedge pt \rightarrow var = i)))$
  by (*simp add*: *wf-levellist-def*)
from *wfll-rt rt-Node rt*
have *llb-nc*:
  $\forall i.\ ((high\ p) \rightarrow var) < i \longrightarrow (llb\ !\ i) = (lla\ !\ i)$
  by (*simp add*: *wf-levellist-def*)
from *wfmarking-rt rt-Node rt*
have *mort-nc*: $\forall n.\ n \notin set\text{-}of\ rt \longrightarrow marka\ n = markb\ n$
  by (*simp add*: *wf-marking-def*)
from *wfmarking-rt rt-Node rt*
have *mirt-marked*: $\forall n.\ n \in set\text{-}of\ rt \longrightarrow markb\ n = m$
  by (*simp add*: *wf-marking-def*)
with *p-notin-ll wfll-rt wfll-lt*
have *p-notin-llb*: $\forall i < length\ llb.\ p \notin set\ (llb\ !\ i)$
  apply −
  apply (*intro allI impI*)
  apply (*clarsimp simp add*: *wf-levellist-def lt-Node rt-Node*)
  apply (*case-tac* $i \le var\ r$)
  apply (*drule-tac* $x{=}i$ **in** *spec*)
  using *orderedt rt-Node lt-Node*
  apply *clarsimp*
  apply (*erule disjE*)
  apply *clarsimp*
  apply (*case-tac* $i \le var\ l$)
  apply (*drule-tac* $x{=}i$ **in** *spec*)
  apply *clarsimp*
  apply *clarsimp*
  apply (*subgoal-tac* $llb\ !\ i = lla\ !\ i$)
  prefer *2*
  apply *clarsimp*
  apply (*case-tac* $i \le var\ l$)
  apply (*drule-tac* $x{=}i$ **in** *spec*, *erule impE*, *assumption*)
  apply *clarsimp*
  using *orderedt rt-Node lt-Node*
  apply *clarsimp*
  apply *clarsimp*

```
  done
from wfll-lt wfll-rt varsll lla llb
obtain var-p-bounds: var p < length levellistb var p < length llb
  by (simp add: Levellist-length wf-levellist-def lt-Node rt-Node)
with p-notin-llb llb pnN var-p-bounds
have llc: Levellist (levellistb[var p := p])
              (nextb(p := levellistb ! var p))
              (llb[var p := p # llb ! var p])
  apply (clarsimp simp add: Levellist-def map-update)
  apply (erule-tac x=i in allE)
  apply (erule-tac x=i in allE)
  apply clarsimp
  apply (case-tac i=var p)
  apply  simp
  apply simp
  done
then show ?thesis
proof (clarsimp)
  show wf-levellist (Node lt p rt) ll (llb[var p := p#llb ! var p]) var ∧
      wf-marking (Node lt p rt) mark (markb(p := m)) m
  proof −
    have nodes-in-upllb: ∀ q. q ∈ set-of (Node lt p rt)
      ⟶ q ∈ set (llb[var p :=p # llb ! var p] ! (var q))
      apply −
      apply (rule allI)
      apply (rule impI)
    proof −
      fix q
      assume q-in-t: q ∈ set-of (Node lt p rt)
      show q-in-upllb:
        q ∈ set (llb[var p :=p # llb ! var p] ! (var q))
      proof (cases q ∈ set-of rt)
        case True
        with nodes-in-llb have q-in-llb: q ∈ set (llb ! (var q))
          by fastforce
        from orderedt rt-Node lt-Node lt rt
        have ordered-rt: ordered rt var
          by fastforce
        from True rt ordered-rt rt-Node lt lt-Node have var q ≤ var r
          apply −
          apply (drule subnodes-ordered)
          apply fastforce
          apply fastforce
          apply fastforce
          done
        with orderedt rt lt rt-Node lt-Node have var q < var p
          by fastforce
        then have
          llb[var p :=p#llb ! var p] ! var q =
```

```
      llb ! var q
      by fastforce
    with q-in-llb show ?thesis
      by fastforce
next
  assume q-notin-rt: q ∉ set-of rt
  show q ∈ set (llb[var p :=p # llb ! var p] ! var q)
  proof (cases q ∈ set-of lt)
    case True
    assume q-in-lt: q ∈ set-of lt
    with nodes-in-lla have q-in-lla: q ∈ set (lla ! (var q))
      by fastforce
    from orderedt rt-Node lt-Node lt rt
    have ordered-lt: ordered lt var
      by fastforce
    from q-in-lt lt ordered-lt rt-Node rt lt-Node
    have var q ≤ var l
      apply −
      apply (drule subnodes-ordered)
      apply fastforce
      apply fastforce
      apply fastforce
      done
    with orderedt rt lt rt-Node lt-Node have qsp: var q < var p
      by fastforce
    then show ?thesis
    proof (cases var q ≤ var (high p))
      case True
      with llb-st
      have ∃ prx. (llb ! (var q)) = prx@(lla ! (var q))
        by fastforce
      with nodes-in-lla q-in-lla
      have q-in-llb: q ∈ set (llb ! (var q))
        by fastforce
      from qsp
      have llb[var p :=p#llb ! var p]!var q =
              llb ! (var q)
        by fastforce
      with q-in-llb show ?thesis
        by fastforce
    next
      assume ¬ var q ≤ var (high p)
      with llb-nc have llb ! (var q) = lla ! (var q)
        by fastforce
      with q-in-lla have q-in-llb: q ∈ set (llb ! (var q))
        by fastforce
      from qsp have
        llb[var p :=p # llb ! var p] ! var q = llb ! (var q)
        by fastforce
```

```
        with q-in-llb show ?thesis
          by fastforce
      qed
    next
      assume q-notin-lt: q ∉ set-of lt
      with q-notin-rt rt lt rt-Node lt-Node q-in-t have qp: q = p
        by fastforce
      with varsll lla-eq-ll llb-eq-lla have var p < length llb
        by fastforce
      with qp show ?thesis
        by simp
    qed
  qed
qed
have prx-ll-st: ∀ i ≤ var p.
 (∃ prx. llb[var p :=p#llb!var p]!i = prx@(ll!i) ∧
      (∀ pt ∈ set prx. pt ∈ set-of (Node lt p rt) ∧ var pt = i))
  apply −
  apply (rule allI)
  apply (rule impI)
proof −
  fix i
  assume isep: i ≤ var p
  show ∃ prx. llb[var p :=p#llb!var p]!i = prx@ll!i ∧
    (∀ pt∈set prx. pt ∈ set-of (Node lt p rt) ∧ var pt = i)
  proof (cases i = var p)
    case True
    with orderedt lt lt-Node rt rt-Node
    have lpsp: var (low p) < var p
      by fastforce
    with orderedt lt lt-Node rt rt-Node
    have hpsp: var (high p) < var p
      by fastforce
    with lpsp lla-nc
    have llall: lla ! var p = ll ! var p
      by fastforce
    with hpsp llb-nc have llb ! var p = ll ! var p
      by fastforce
    with llb-eq-lla lla-eq-ll isep True varsll lt rt show ?thesis
      apply −
      apply (rule-tac x=[p] in exI)
      apply (rule conjI)
      apply simp
      apply (rule ballI)
      apply fastforce
      done
  next
    assume inp: i ≠ var p
    show ?thesis
```

```
proof (cases var (low p) < i)
  case True
  with lla-nc have llall: lla ! i = ll ! i
    by fastforce
  assume vpsi: var (low p) < i
  show ?thesis
  proof (cases var (high p) < i)
    case True
    with llall llb-nc have llb ! i = ll ! i
      by fastforce
    with inp True vpsi varsll lt rt show ?thesis
      apply -
      apply (rule-tac x=[] in exI)
      apply (rule conjI)
      apply simp
      apply (rule ballI)
      apply fastforce
      done
  next
    assume isehp: ¬ var (high p) < i
    with vpsi lla-nc have lla-ll: lla ! i = ll ! i
      by fastforce
    with isehp llb-st
    have prx-lla: ∃prx. llb ! i = prx @ lla ! i ∧
      (∀pt∈set prx. pt ∈ set-of rt ∧ var pt = i)
      apply -
      apply (erule-tac x=i in allE)
      apply simp
      done
    with lla-ll inp rt show ?thesis
      apply -
      apply (erule exE)
      apply (rule-tac x=prx in exI)
      apply simp
      done
  qed
next
  assume iselp: ¬ var (low p) < i
  show ?thesis
  proof (cases var (high p) < i)
    case True
    with llb-nc have llb-ll: llb ! i = lla ! i
      by fastforce
    with iselp lla-st
    have prx-ll: ∃prx. lla ! i = prx @ ll ! i ∧
      (∀pt∈set prx. pt ∈ set-of lt ∧ var pt = i)
      apply -
      apply (erule-tac x=i in allE)
      apply simp
```

done
with *llb-ll inp lt* **show** *?thesis*
apply −
apply (*erule exE*)
apply (*rule-tac x=prx* **in** *exI*)
apply *simp*
done
next
assume *isehp*: ¬ *var* (*high p*) < *i*
from *iselp lla-st*
have *prxl*: ∃ *prx*. *lla* ! *i* = *prx* @ *ll* ! *i* ∧
(∀ *pt*∈*set prx*. *pt* ∈ *set-of lt* ∧ *var pt* = *i*)
by *fastforce*
from *isehp llb-st*
have *prxh*: ∃ *prx*. *llb* ! *i* = *prx* @ *lla* ! *i* ∧
(∀ *pt*∈*set prx*. *pt* ∈ *set-of rt* ∧ *var pt* = *i*)
by *fastforce*
with *prxl inp lt pnN rt* **show** *?thesis*
apply −
apply (*elim exE*)
apply (*rule-tac x=prxa* @ *prx* **in** *exI*)
apply *simp*
apply (*elim conjE*)
apply *fastforce*
done
qed
qed
qed
qed
have *big-Nodes-nc*: ∀ *i*. (*p*−>*var*) < *i*
⟶ (*llb*[*var p* :=*p* # *llb* ! *var p*]) ! *i* = *ll* ! *i*
apply −
apply (*rule allI*)
apply (*rule impI*)
proof −
fix *i*
assume *psi*: *var p* < *i*
with *orderedt lt rt lt-Node rt-Node* **have** *lpsi*: *var* (*low p*) < *i*
by *fastforce*
with *lla-nc* **have** *lla-ll*: *lla* ! *i* = *ll* ! *i*
by *fastforce*
from *psi orderedt lt rt lt-Node rt-Node* **have** *hpsi*: *var* (*high p*) < *i*
by *fastforce*
with *llb-nc* **have** *llb-lla*: *llb* ! *i* = *lla* ! *i*
by *fastforce*
from *psi*
have *upllb-llb*: *llb*[*var p* :=*p*#*llb*!*var p*]!*i* = *llb*!*i*
by *fastforce*
from *upllb-llb llb-lla lla-ll*

```
    show llb[var p :=p # llb ! var p] ! i = ll ! i
      by fastforce
  qed
  from lla-eq-ll llb-eq-lla
  have length-eq: length (llb[var p :=p # llb ! var p]) = length ll
    by fastforce
  from length-eq big-Nodes-nc prx-ll-st nodes-in-upllb
  have wf-ll-upllb:
    wf-levellist (Node lt p rt) ll (llb[var p :=p # llb ! var p]) var
    by (simp add: wf-levellist-def)
  have mark-nc:
    ∀ n. n ∉ set-of (Node lt p rt) ⟶ (markb(p:=m)) n = mark n
    apply –
    apply (rule allI)
    apply (rule impI)
  proof –
    fix n
    assume nnit: n ∉ set-of (Node lt p rt)
    with lt rt have nnilt: n ∉ set-of lt
      by fastforce
    from nnit lt rt have nnirt: n ∉ set-of rt
      by fastforce
    with nnilt mot-nc mort-nc have mb-eq-m: markb n = mark n
      by fastforce
    from nnit have n≠p
      by fastforce
    then have upmarkb-markb: (markb(p :=m)) n = markb n
      by fastforce
    with mb-eq-m show (markb(p :=m)) n = mark n
      by fastforce
  qed
  have mark-c: ∀ n. n ∈ set-of (Node lt p rt) ⟶ (markb(p :=m)) n
= m
    apply –
    apply (intro allI)
    apply (rule impI)
  proof –
    fix n
    assume nint: n ∈ set-of (Node lt p rt)
    show (markb(p :=m)) n = m
    proof (cases n=p)
      case True
      then show ?thesis
        by fastforce
    next
      assume nnp: n ≠ p
      show ?thesis
      proof (cases n ∈ set-of rt)
        case True
```

```
                with mirt-marked have markb n = m
                  by fastforce
                with nnp show ?thesis
                  by fastforce
              next
                assume nninrt:  n ∉ set-of rt
                with nint nnp have ninlt: n ∈ set-of lt
                  by fastforce
                with mit-marked have marka-m: marka n = m
                  by fastforce
                from mort-nc nninrt have marka n = markb n
                  by fastforce
                with marka-m have markb n = m
                  by fastforce
                with nnp show ?thesis
                  by fastforce
              qed
            qed
          qed
          from mark-c mark-nc
          have wf-mark: wf-marking (Node lt p rt) mark (markb(p :=m)) m
            by (simp add: wf-marking-def)
          with wf-ll-upllb show ?thesis
            by fastforce
        qed
      qed
    qed
  qed
  done
 qed
 done
qed
next
 fix var low high p lt rt and levellist and
   ll::ref list list and mark::ref ⇒ bool and next
 assume pnN: p ≠ Null
 assume ll: Levellist levellist next ll
 assume vpsll: var p < length levellist
 assume orderedt: ordered (Node lt p rt) var
 assume marked-child-ll: ∀ n∈set-of (Node lt p rt).
         if mark n = mark p
         then n ∈ set (ll ! var n) ∧
              (∀ nt pa. Dag n low high nt ∧ pa ∈ set-of nt ⟶ mark pa = mark p)
         else n ∉ set (concat ll)
 assume lt: Dag (low p) low high lt
 assume rt: Dag (high p) low high rt
 show wf-levellist (Node lt p rt) ll ll var ∧
      wf-marking (Node lt p rt) mark mark (mark p)
 proof -
```

from *marked-child-ll pnN lt rt* **have** *marked-st*:
$(\forall\, pa.\ pa \in$ *set-of* $($*Node lt p rt*$) \longrightarrow$ *mark pa* $=$ *mark p*$)$
apply –
apply (*drule-tac x=p* **in** *bspec*)
apply *simp*
apply (*clarsimp*)
apply (*erule-tac x=*(*Node lt p rt*) **in** *allE*)
apply *simp*
done
have *nodest-in-ll*:
$\forall\, q.\ q \in$ *set-of* (*Node lt p rt*) $\longrightarrow q \in$ *set* (*ll* ! *var q*)
proof –
from *marked-child-ll pnN* **have** *pinll*: $p \in$ *set* (*ll* ! *var p*)
apply –
apply (*drule-tac x=p* **in** *bspec*)
apply *simp*
apply *fastforce*
done
from *marked-st marked-child-ll lt rt* **show** *?thesis*
apply –
apply (*rule allI*)
apply (*erule-tac x=q* **in** *allE*)
apply (*rule impI*)
apply (*erule impE*)
apply *assumption*
apply (*drule-tac x=q* **in** *bspec*)
apply *simp*
apply *fastforce*
done
qed
have *levellist-nc*: $\forall\ i \le$ *var p*. ($\exists$ *prx*. *ll* ! *i* = *prx*@(*ll* ! *i*) $\wedge$
($\forall$ *pt* $\in$ *set prx*. *pt* $\in$ *set-of* (*Node lt p rt*) $\wedge$ *var pt* = *i*))
apply –
apply (*rule allI*)
apply (*rule impI*)
apply (*rule-tac x=*[] **in** *exI*)
apply *fastforce*
done
have *ll-nc*: $\forall\ i.$ (*var p*) $< i \longrightarrow$ *ll* ! *i* = *ll* ! *i*
by *fastforce*
have *length-ll*: *length ll* = *length ll*
by *fastforce*
with *ll-nc levellist-nc nodest-in-ll*
have *wf*: *wf-levellist* (*Node lt p rt*) *ll ll var*
by (*simp add*: *wf-levellist-def*)
have *m-nc*: $\forall\ n.\ n \notin$ *set-of* (*Node lt p rt*) $\longrightarrow$ *mark n* = *mark n*
by *fastforce*
from *marked-st* **have** $\forall\ n.\ n \in$ *set-of* (*Node lt p rt*) $\longrightarrow$ *mark n* = *mark p*
by *fastforce*

with *m-nc* **have** *wf-marking* (*Node lt p rt*) *mark mark* (*mark p*)
by (*simp add*: *wf-marking-def*)
with *wf* **show** *?thesis*
by *fastforce*
qed
qed

lemma *allD*: ∀ *ll*. *P ll* ⟹ *P ll*
by *blast*

lemma *replicate-spec*: ⟦∀ *i* < *n*. *xs* ! *i* = *x*; *n*=*length xs*⟧
⟹ *replicate* (*length xs*) *x* = *xs*
apply *hypsubst-thin*
apply (*induct xs*)
apply *simp*
apply *force*
done

lemma (**in** *Levellist-impl*) *Levellist-spec-total*:
shows ∀ σ *t*. Γ,Θ⊢$_t$
⦃σ. *Dag ´p ´low ´high t* ∧ (∀ *i* < *length ´levellist*. *´levellist* ! *i* = *Null*) ∧
length ´levellist = *´p* → *´var* + *1* ∧
ordered t ´var ∧ (∀ *n* ∈ *set-of t*. *´mark n* = (¬ *´m*))⦄
´levellist :== *PROC Levellist* (*´p*, *´m*, *´levellist*)
⦃∃ *ll*. *Levellist ´levellist ´next ll* ∧ *wf-ll t ll* $^\sigma$*var* ∧
length ´levellist = $^\sigma$*p* → $^\sigma$*var* + *1* ∧
wf-marking t $^\sigma$*mark ´mark* $^\sigma$*m* ∧
(∀ *p*. *p* ∉ *set-of t* ⟶ $^\sigma$*next p* = *´next p*)⦄
apply (*hoare-rule HoareTotal.conseq*)
apply (*rule-tac ll*=*replicate* ($^\sigma$*p*→$^\sigma$*var* + *1*) [] **in** *allD* [*OF Levellist-spec-total'*])
apply (*intro allI impI*)
apply (*rule-tac x*=σ **in** *exI*)
apply (*rule-tac x*=*t* **in** *exI*)
apply (*rule conjI*)
apply (*clarsimp split*:*if-split-asm simp del*: *concat-replicate-trivial*)
apply (*frule replicate-spec* [*symmetric*])
apply (*simp*)
apply (*clarsimp simp add*: *Levellist-def*)
apply (*case-tac i*)
apply *simp*
apply *simp*
apply (*simp add*: *Collect-conv-if split*:*if-split-asm*)
apply *vcg-step*
apply (*elim exE conjE*)
apply (*rule-tac x*=*ll'* **in** *exI*)
apply *simp*
apply (*thin-tac* ∀ *p*. *p* ∉ *set-of t* ⟶ *next p* = *nexta p*)
apply (*simp add*: *wf-levellist-def wf-ll-def*)
apply (*case-tac t* = *Tip*)

apply *simp*
apply (*rule conjI*)
apply *clarsimp*
apply (*case-tac k*)
apply *simp*
apply *simp*
apply (*subgoal-tac length ll′=Suc* (*var Null*))
apply (*simp add*: *Levellist-length*)
apply *fastforce*
apply (*split dag.splits*)
apply *simp*
apply (*elim conjE*)
apply (*intro conjI*)
apply (*rule allI*)
apply (*erule-tac x=pa* **in** *allE*)
apply *clarify*
prefer *2*
apply (*simp add*: *Levellist-length*)
apply (*rule allI*)
apply (*rule impI*)
apply (*rule ballI*)
apply (*rotate-tac 11*)
apply (*erule-tac x=k* **in** *allE*)
apply (*rename-tac dag1 ref dag2 k pa*)
apply (*subgoal-tac k* <= *var ref*)
prefer *2*
apply (*subgoal-tac ref* = *p*)
apply *simp*
apply *clarify*
apply (*erule-tac ?P* = *Dag p low high* (*Node dag1 ref dag2*) **in** *rev-mp*)
apply (*simp* (*no-asm*))
apply (*rotate-tac 14*)
apply (*erule-tac x=k* **in** *allE*)
apply *clarify*
apply (*erule-tac x=k* **in** *allE*)
apply *clarify*
apply (*case-tac k*)
apply *simp*
apply *simp*
done

end

7 Proof of Procedure ShareRep

theory *ShareRepProof* **imports** *ProcedureSpecs Simpl.HeapList* **begin**

lemma (**in** *ShareRep-impl*) *ShareRep-modifies*:
shows $\forall\sigma.\ \Gamma\vdash\{\sigma\}$ *PROC ShareRep* (*´nodeslist*, *´p*)

{*t. t may-only-modify-globals σ in* [*rep*]}
apply (*hoare-rule HoarePartial.ProcRec1*)
apply (*vcg spec=modifies*)
done

lemma *hd-filter-cons*:
⋀ *i*. ⟦ *P* (*xs* ! *i*) *p*; *i* < *length xs*; ∀ *no* ∈ *set* (*take i xs*). ¬ *P no p*; ∀ *a b*. *P a b* = *P b a*⟧
⟹ *xs* ! *i* = *hd* (*filter* (*P p*) *xs*)
apply (*induct xs*)
apply *simp*
apply (*case-tac P a p*)
apply *simp*
apply (*case-tac i*)
apply *simp*
apply *simp*
apply (*case-tac i*)
apply *simp*
apply *auto*
done

lemma (**in** *ShareRep-impl*) *ShareRep-spec-total*:
shows
∀ σ *ns*. Γ,Θ$\vdash_t$
⦃σ. *List ´nodeslist ´next ns* ∧
(∀ *no* ∈ *set ns*. *no* ≠ *Null* ∧
((*no*→*´low* = *Null*) = (*no*→*´high* = *Null*)) ∧
(*isLeaf-pt ´p ´low ´high* ⟶ *isLeaf-pt no ´low ´high*) ∧
no→*´var* = *´p*→*´var*) ∧
´p ∈ *set ns*⦄
PROC ShareRep (*´nodeslist*, *´p*)
⦃ ($^\sigma$*p* → *´rep* = *hd* (*filter* (λ *sn*. *repNodes-eq sn* $^\sigma$*p* $^\sigma$*low* $^\sigma$*high* $^\sigma$*rep*) *ns*)) ∧
(∀ *pt*. *pt* ≠ $^\sigma$*p* ⟶ *pt*→$^\sigma$*rep* = *pt*→*´rep*) ∧
($^\sigma$*p*→*´rep*→$^\sigma$*var* = $^\sigma$*p* → $^\sigma$*var*)⦄
apply (*hoare-rule HoareTotal.ProcNoRec1*)
apply (*hoare-rule anno*=
IF (*isLeaf-pt ´p ´low ´high*)
THEN ´p → *´rep* :== *´nodeslist*
ELSE
WHILE (*´nodeslist* ≠ *Null*)
INV ⦃∃ *prx sfx*. *List ´nodeslist ´next sfx* ∧ *ns*=*prx*@*sfx* ∧
¬ *isLeaf-pt ´p ´low ´high* ∧
(∀ *no* ∈ *set ns*. *no* ≠ *Null* ∧
((*no*→$^\sigma$*low* = *Null*) = (*no*→$^\sigma$*high* = *Null*)) ∧
(*isLeaf-pt* $^\sigma$*p* $^\sigma$*low* $^\sigma$*high* ⟶ *isLeaf-pt no* $^\sigma$*low* $^\sigma$*high*) ∧
no→$^\sigma$*var* = $^\sigma$*p*→$^\sigma$*var*) ∧
$^\sigma$*p* ∈ *set ns* ∧
((∃ *pt* ∈ *set prx*. *repNodes-eq pt* $^\sigma$*p* $^\sigma$*low* $^\sigma$*high* $^\sigma$*rep*)

$\longrightarrow$ ´*rep* $^{\sigma}p$ = *hd* (*filter* (λ *sn*. *repNodes-eq sn* $^{\sigma}p$ $^{\sigma}low$ $^{\sigma}high$ $^{\sigma}rep$) *prx*) $\wedge$
($\forall$ *pt*. *pt* $\neq$ $^{\sigma}p$ $\longrightarrow$ *pt*→$^{\sigma}$*rep* = *pt*→´*rep*)) $\wedge$
(($\forall$ *pt* $\in$ *set prx*. $\neg$ *repNodes-eq pt* $^{\sigma}p$ $^{\sigma}low$ $^{\sigma}high$ $^{\sigma}rep$) $\longrightarrow$ $^{\sigma}rep$ = ´*rep*) $\wedge$
(´*nodeslist* $\neq$ *Null* $\longrightarrow$
($\forall$ *pt* $\in$ *set prx*. $\neg$ *repNodes-eq pt* $^{\sigma}p$ $^{\sigma}low$ $^{\sigma}high$ $^{\sigma}rep$)) $\wedge$
(´*p* = $^{\sigma}p$ $\wedge$ ´*high* = $^{\sigma}high$ $\wedge$ ´*low* = $^{\sigma}low$)|}
VAR MEASURE (*length* (*list* ´*nodeslist* ´*next*))
DO
IF (*repNodes-eq* ´*nodeslist* ´*p* ´*low* ´*high* ´*rep*)
THEN ´*p*→´*rep* :== ´*nodeslist*;; ´*nodeslist* :== *Null*
ELSE ´*nodeslist* :== ´*nodeslist*→´*next*
FI
OD
FI **in** *HoareTotal.annotateI*)
apply *vcg*
using [[*simp-depth-limit* = *2*]]
apply (*rule conjI*)
apply *clarify*
apply (*simp* (*no-asm-use*))
prefer *2*
apply *clarify*
apply (*rule-tac x*=[] **in** *exI*)
apply (*rule-tac x*=*ns* **in** *exI*)
apply (*simp* (*no-asm-use*))
prefer *2*
apply *clarify*
apply (*rule conjI*)
apply *clarify*
apply (*rule conjI*)
apply (*clarsimp simp add*: *List-list*)
apply (*simp* (*no-asm-use*))
apply (*rule conjI*)
apply *assumption*
prefer *2*
apply *clarify*
apply (*simp* (*no-asm-use*))
apply (*rule conjI*)
apply (*clarsimp simp add*: *List-list*)
apply (*simp only*: *List-not-Null simp-thms triv-forall-equality*)
apply *clarify*
apply (*simp only*: *triv-forall-equality*)
apply (*rename-tac sfx*)
apply (*rule-tac x*=*prx*@[*nodeslist*] **in** *exI*)
apply (*rule-tac x*=*sfx* **in** *exI*)
apply (*rule conjI*)
apply *assumption*
apply (*rule conjI*)
apply *simp*
prefer *4*

```
apply   (elim exE conjE)
apply   (simp (no-asm-use))
apply   hypsubst
using  [[simp-depth-limit = 100]]
proof −

  fix ns var low high rep next p nodeslist
  assume ns: List nodeslist next ns
  assume no-prop:  ∀ no∈set ns.
          no ≠ Null ∧
          (low no = Null) = (high no = Null) ∧
          (isLeaf-pt p low high ⟶ isLeaf-pt no low high) ∧ var no = var p
  assume p-in-ns: p ∈ set ns
  assume p-Leaf: isLeaf-pt p low high
  show nodeslist = hd [sn←ns . repNodes-eq sn p low high rep] ∧
        var nodeslist = var p
  proof −
    from p-in-ns no-prop have p-not-Null: p≠Null
      using [[simp-depth-limit=2]]
      by auto
    from p-in-ns have ns ≠ []
      by (cases ns) auto
    with ns obtain ns' where ns': ns = nodeslist#ns'
      by (cases nodeslist=Null) auto
    with no-prop p-Leaf obtain
      isLeaf-pt nodeslist low high and
      var-eq: var nodeslist = var p and
      nodeslist≠Null
      using [[simp-depth-limit=2]]
      by auto
    with p-not-Null p-Leaf have repNodes-eq nodeslist p low high rep
      by (simp add: repNodes-eq-def isLeaf-pt-def null-comp-def)
    with ns' var-eq
    show ?thesis
      by simp
  qed
next

  fix var::ref⇒nat and low high rep repa p prx sfx next
  assume sfx: List Null next sfx
  assume p-in-ns: p ∈ set (prx @ sfx)
  assume no-props: ∀ no∈set (prx @ sfx).
          no ≠ Null ∧
          (low no = Null) = (high no = Null) ∧
          (isLeaf-pt p low high ⟶ isLeaf-pt no low high) ∧ var no = var p
  assume match-prx: (∃ pt∈set prx. repNodes-eq pt p low high rep) ⟶
                    repa p = hd [sn←prx . repNodes-eq sn p low high rep] ∧
                   (∀ pt. pt ≠ p ⟶ rep pt = repa pt)
  show repa p = hd [sn←prx @ sfx . repNodes-eq sn p low high rep] ∧
```

(∀ *pt*. *pt* ≠ *p* ⟶ *rep pt* = *repa pt*) ∧ *var* (*repa p*) = *var p*
proof −
from *sfx*
have *sfx-Nil*: *sfx*=[]
by *simp*
with *p-in-ns* **have** *ex-match*: (∃ *pt*∈*set prx*. *repNodes-eq pt p low high rep*)
apply −
apply (*rule-tac x*=*p* **in** *bexI*)
apply (*simp add*: *repNodes-eq-def*)
apply *simp*
done
hence *not-empty*: [*sn*←*prx* . *repNodes-eq sn p low high rep*] ≠ []
apply −
apply (*erule bexE*)
apply (*rule filter-not-empty*)
apply *auto*
done
from *ex-match match-prx* **obtain**
found: *repa p* = *hd* [*sn*←*prx* . *repNodes-eq sn p low high rep*] **and**
unmodif: ∀ *pt*. *pt* ≠ *p* ⟶ *rep pt* = *repa pt*
by *blast*
from *hd-filter-in-list* [*OF not-empty*] *found*
have *repa p* ∈ *set prx*
by *simp*
with *no-props*
have *var* (*repa p*) = *var p*
using [[*simp-depth-limit*=*2*]]
by *simp*
with *found unmodif sfx-Nil*
show *?thesis*
by *simp*
qed
next

fix *var low high p repa next nodeslist prx sfx*
assume *nodeslist-not-Null*: *nodeslist* ≠ *Null*
assume *p-no-Leaf*: ¬ *isLeaf-pt p low high*
assume *no-props*: ∀ *no*∈*set prx* ∪ *set* (*nodeslist* # *sfx*).
no ≠ *Null* ∧ (*low no* = *Null*) = (*high no* = *Null*) ∧ *var no* = *var p*
assume *p-in-ns*: *p* ∈ *set prx* ∨ *p* ∈ *set* (*nodeslist* # *sfx*)
assume *match-prx*: (∃ *pt*∈*set prx*. *repNodes-eq pt p low high repa*) ⟶
repa p = *hd* [*sn*←*prx* . *repNodes-eq sn p low high repa*]
assume *nomatch-prx*: ∀ *pt*∈*set prx*. ¬ *repNodes-eq pt p low high repa*
assume *nomatch-nodeslist*: ¬ *repNodes-eq nodeslist p low high repa*
assume *sfx*: *List* (*next nodeslist*) *next sfx*
show (∀ *no*∈*set prx* ∪ *set* (*nodeslist* # *sfx*).
no ≠ *Null* ∧ (*low no* = *Null*) = (*high no* = *Null*) ∧ *var no* = *var p*) ∧
((∃ *pt*∈*set* (*prx* @ [*nodeslist*]). *repNodes-eq pt p low high repa*) ⟶
repa p = *hd* [*sn*←*prx* @ [*nodeslist*] . *repNodes-eq sn p low high repa*]) ∧

$(next\ nodeslist \neq Null \longrightarrow$
$(\forall pt \in set\ (prx\ @\ [nodeslist]).\ \neg\ repNodes\text{-}eq\ pt\ p\ low\ high\ repa))$

proof –
from *nomatch-prx nomatch-nodeslist*
have $((\exists pt \in set\ (prx\ @\ [nodeslist]).\ repNodes\text{-}eq\ pt\ p\ low\ high\ repa) \longrightarrow$
$repa\ p = hd\ [sn \leftarrow prx\ @\ [nodeslist]\ .\ repNodes\text{-}eq\ sn\ p\ low\ high\ repa])$
by *auto*
moreover
from *nomatch-prx nomatch-nodeslist*
have $(next\ nodeslist \neq Null \longrightarrow$
$(\forall pt \in set\ (prx\ @\ [nodeslist]).\ \neg\ repNodes\text{-}eq\ pt\ p\ low\ high\ repa))$
by *auto*
ultimately show *?thesis*
using *no-props*
by (*intro conjI*)
qed
next

fix *var low high p repa next nodeslist prx sfx*
assume *nodeslist-not-Null*: $nodeslist \neq Null$
assume *sfx*: *List nodeslist next sfx*
assume *p-not-Leaf*: $\neg\ isLeaf\text{-}pt\ p\ low\ high$
assume *no-props*: $\forall no \in set\ prx \cup set\ sfx.$
$no \neq Null\ \wedge$
$(low\ no = Null) = (high\ no = Null)\ \wedge$
$(isLeaf\text{-}pt\ p\ low\ high \longrightarrow isLeaf\text{-}pt\ no\ low\ high) \wedge var\ no = var\ p$
assume *p-in-ns*: $p \in set\ prx \vee p \in set\ sfx$
assume *match-prx*: $(\exists pt \in set\ prx.\ repNodes\text{-}eq\ pt\ p\ low\ high\ repa) \longrightarrow$
$repa\ p = hd\ [sn \leftarrow prx\ .\ repNodes\text{-}eq\ sn\ p\ low\ high\ repa]$
assume *nomatch-prx*: $\forall pt \in set\ prx.\ \neg\ repNodes\text{-}eq\ pt\ p\ low\ high\ repa$
assume *match*: *repNodes-eq nodeslist p low high repa*
show $(\forall no \in set\ prx \cup set\ sfx.$
$no \neq Null\ \wedge$
$(low\ no = Null) = (high\ no = Null)\ \wedge$
$(isLeaf\text{-}pt\ p\ low\ high \longrightarrow isLeaf\text{-}pt\ no\ low\ high) \wedge var\ no = var\ p) \wedge$
$(p \in set\ prx \vee p \in set\ sfx) \wedge$
$((\exists pt \in set\ prx \cup set\ sfx.\ repNodes\text{-}eq\ pt\ p\ low\ high\ repa) \longrightarrow$
$nodeslist =$
$hd\ ([sn \leftarrow prx\ .\ repNodes\text{-}eq\ sn\ p\ low\ high\ repa]\ @$
$[sn \leftarrow sfx\ .\ repNodes\text{-}eq\ sn\ p\ low\ high\ repa])) \wedge$
$((\forall pt \in set\ prx \cup set\ sfx.\ \neg\ repNodes\text{-}eq\ pt\ p\ low\ high\ repa) \longrightarrow$
$repa = repa(p := nodeslist))$
proof –
from *nodeslist-not-Null sfx*
obtain *sfx'* **where** *sfx'*: $sfx = nodeslist \# sfx'$
by (*cases nodeslist=Null*) *auto*
from *nomatch-prx match sfx'*
have *hd*: $hd\ ([sn \leftarrow prx\ .\ repNodes\text{-}eq\ sn\ p\ low\ high\ repa]\ @$
$[sn \leftarrow sfx\ .\ repNodes\text{-}eq\ sn\ p\ low\ high\ repa]) = nodeslist$

by *simp*
from *match sfx'*
have *triv*: ((∀ *pt*∈*set prx* ∪ *set sfx*. ¬ *repNodes-eq pt p low high repa*) ⟶
repa = *repa*(*p* := *nodeslist*))
by *simp*
show *?thesis*
apply (*rule conjI*)
apply (*rule no-props*)
apply (*intro conjI*)
apply (*rule p-in-ns*)
apply (*simp add*: *hd*)
apply (*rule triv*)
done
qed
qed

end

8 Proof of Procedure ShareReduceRepList

theory *ShareReduceRepListProof* **imports** *ShareRepProof* **begin**

lemma (**in** *ShareReduceRepList-impl*) *ShareReduceRepList-modifies*:
shows ∀ σ. Γ⊢{σ} *PROC ShareReduceRepList* (´*nodeslist*)
{*t*. *t may-only-modify-globals* σ *in* [*rep*]}
apply (*hoare-rule HoarePartial.ProcRec1*)
apply (*vcg spec=modifies*)
done

lemma *hd-filter-app*: ⟦*filter P xs* ≠ []; *zs*=*xs*@*ys*⟧ ⟹
hd (*filter P zs*) = *hd* (*filter P xs*)
by (*induct xs arbitrary*: *n m*) *auto*

lemma (**in** *ShareReduceRepList-impl*) *ShareReduceRepList-spec-total*:
defines *var-eq* ≡ (λ*ns var*. (∀ *no1* ∈ *set ns*. ∀ *no2* ∈ *set ns*. *no1*→*var* = *no2*→*var*))
shows
∀ σ *ns*. Γ$\vdash_t$
⦃σ. *List* ´*nodeslist* ´*next ns* ∧
(∀ *no* ∈ *set ns*.
no ≠ *Null* ∧ ((*no*→´*low* = *Null*) = (*no*→´*high* = *Null*)) ∧
no→´*low* ∉ *set ns* ∧ *no*→´*high* ∉ *set ns* ∧
(*isLeaf-pt no* ´*low* ´*high* = (*no*→´*var* ≤ *1*)) ∧
(*no*→´*low* ≠ *Null* ⟶ (*no*→´*low*)→´*rep* ≠ *Null*) ∧
((´*rep* ∝ ´*low*) *no* ∉ *set ns*)) ∧
var-eq ns ´*var*⦄
PROC ShareReduceRepList (´*nodeslist*)
⦃(∀ *no*. *no* ∉ *set ns* ⟶ *no*→$^\sigma$*rep* = *no*→´*rep*) ∧
(∀ *no* ∈ *set ns*. *no*→´*rep* ≠ *Null* ∧

(*if* ((´*rep* ∝ ^σ*low*) *no* = (´*rep* ∝ ^σ*high*) *no* ∧ *no*→ ^σ*low* ≠ *Null*)
then (*no*→´*rep* = (´*rep* ∝ ^σ*low*) *no*)
else ((*no*→´*rep*) ∈ *set ns* ∧ *no*→´*rep*→´*rep* = *no*→´*rep* ∧
(∀ *no1* ∈ *set ns*.
((´*rep* ∝ ^σ*high*) *no1* = (´*rep* ∝ ^σ*high*) *no* ∧
(´*rep* ∝ ^σ*low*) *no1* = (´*rep* ∝ ^σ*low*) *no*) = (*no*→´*rep* = *no1*→´*rep*)))))⦄
apply (*hoare-rule HoareTotal.ProcNoRec1*)
apply (*hoare-rule anno*=
´*node* :== ´*nodeslist*;;
WHILE (´*node* ≠ *Null*)
INV ⦃∃ *prx sfx*. *List* ´*node* ´*next sfx* ∧
List ´*nodeslist* ´*next ns* ∧ *ns*=*prx*@*sfx* ∧
(∀ *no* ∈ *set ns*.
no ≠ *Null* ∧ ((*no*→^σ*low* = *Null*) = (*no*→^σ*high* = *Null*)) ∧
no→^σ*low* ∉ *set ns* ∧ *no*→^σ*high* ∉ *set ns* ∧
(*isLeaf-pt no* ^σ*low* ^σ*high* = (*no*→^σ*var* ≤ *1*)) ∧
(*no*→^σ*low* ≠ *Null* ⟶ (*no*→^σ*low*)→^σ*rep* ≠ *Null*) ∧
((^σ*rep* ∝ ^σ*low*) *no* ∉ *set ns*)) ∧
var-eq ns ´*var* ∧
(∀ *no*. *no* ∉ *set prx* ⟶ *no*→^σ*rep* = *no* →´*rep*) ∧
(∀ *no* ∈ *set prx*. *no*→´*rep* ≠ *Null* ∧
(*if* ((´*rep* ∝ ^σ*low*) *no* = (´*rep* ∝ ^σ*high*) *no* ∧ *no*→^σ*low* ≠ *Null*)
then (*no*→´*rep* = (´*rep* ∝ ^σ*low*) *no*)
else ((*no*→´*rep*)=*hd* (*filter* (λ*sn*. *repNodes-eq sn no* ^σ*low* ^σ*high* ´*rep*)
prx) ∧
((*no*→´*rep*)→´*rep*) = *no*→´*rep* ∧
(∀ *no1* ∈ *set prx*.
((´*rep* ∝ ^σ*high*) *no1* = (´*rep* ∝ ^σ*high*) *no* ∧
(´*rep* ∝ ^σ*low*) *no1* = (´*rep* ∝ ^σ*low*) *no*) =
(*no*→´*rep* = *no1*→´*rep*))))) ∧
´*nodeslist*= ^σ*nodeslist* ∧ ´*high*=^σ*high* ∧ ´*low*=^σ*low* ∧ ´*var*=^σ*var*⦄
VAR MEASURE (*length* (*list* ´*node* ´*next*))
DO
IF (¬ *isLeaf-pt* ´*node* ´*low* ´*high* ∧
´*node* → ´*low* → ´*rep* = ´*node* → ´*high* → ´*rep*)
THEN ´*node* → ´*rep* :== ´*node* → ´*low* → ´*rep*
ELSE CALL ShareRep (´*nodeslist* , ´*node*)
FI;;
´*node* :==´*node* → ´*next*
OD **in** *HoareTotal.annotateI*)
apply (*vcg spec*=*spec-total*)
apply (*rule-tac x*=[] **in** *exI*)
apply (*rule-tac x*=*ns* **in** *exI*)
using [[*simp-depth-limit* = *2*]]
apply (*simp* (*no-asm-use*))
prefer *2*
using [[*simp-depth-limit* = *4*]]
apply (*clarsimp*)
prefer *2*

apply (*rule conjI*)
apply *clarify*
apply (*rule conjI*)
apply (*clarsimp simp add*: *List-list*)
apply (*simp only*: *List-not-Null simp-thms triv-forall-equality*)
apply *clarify*
apply (*simp only*: *triv-forall-equality*)
apply (*rename-tac sfx*)
apply (*rule-tac x*=*prx*@[*node*] **in** *exI*)
apply (*rule-tac x*=*sfx* **in** *exI*)
apply (*rule conjI*)
apply *assumption*
apply (*rule conjI*)
apply (*simp* (*no-asm*))
apply (*rule conjI*)
apply (*assumption*)
prefer *2*
apply *clarify*
apply (*simp only*: *List-not-Null simp-thms triv-forall-equality*)
apply *clarify*
apply (*simp only*: *triv-forall-equality*)
apply (*rename-tac sfx*)
apply (*rule-tac x*=*prx*@*node*#*sfx* **in** *exI*)
apply (*rule conjI*)
apply *assumption*
apply (*rule conjI*)
apply (*rule ballI*)
apply (*frule-tac x*=*no* **in** *bspec*, *assumption*)
apply (*drule-tac x*=*node* **in** *bspec*)
apply (*simp* (*no-asm-use*))
apply (*elim conjE*)
apply (*rule conjI*)
apply *assumption*
apply (*rule conjI*)
apply *assumption*
apply (*unfold var-eq-def*)
apply (*drule-tac x*=*node* **in** *bspec*, *simp*)
apply (*drule-tac x*=*no* **in** *bspec*,*assumption*)
apply (*simp add*: *isLeaf-pt-def*)
apply (*rule conjI*)
apply (*simp* (*no-asm*))
apply (*clarify*)
apply (*rule conjI*)
apply (*subgoal-tac List node next* (*node*#*sfx*))
apply (*simp only*: *List-list*)
apply (*simp* (*no-asm*))
apply (*simp* (*no-asm-simp*))
apply (*rule-tac x*=*prx*@[*node*] **in** *exI*)
apply (*rule-tac x*=*sfx* **in** *exI*)

apply (*rule conjI*)
apply *assumption*
apply (*rule conjI*)
apply (*simp* (*no-asm*))
apply (*rule conjI*)
apply (*assumption*)
using [[*simp-depth-limit* = *100*]]
proof −
  fix *var low high rep nodeslist ns repa next no*
  assume *ns*: *List nodeslist next ns*
  assume *no-in-ns*: *no* ∈ *set ns*
  assume *while-inv*: ∀ *no*∈*set ns*.
      repa no ≠ *Null* ∧
      (*if* (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no* ∧ *high no* ≠ *Null*
       then repa no = (*repa* ∝ *low*) *no*
       else repa no = *hd* [*sn*←*ns* . *repNodes-eq sn no low high repa*] ∧
         repa (*repa no*) = *repa no* ∧
         (∀ *no1*∈*set ns*.
           ((*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no* ∧
            (*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*) =
           (*repa no* = *repa no1*)))
  assume *pre*: ∀ *no*∈*set ns*.
      no ≠ *Null* ∧
      (*low no* = *Null*) = (*high no* = *Null*) ∧
      low no ∉ *set ns* ∧
      high no ∉ *set ns* ∧
      isLeaf-pt no low high = (*var no* ≤ *Suc 0*) ∧
      (*low no* ≠ *Null* ⟶ *rep* (*low no*) ≠ *Null*) ∧ (*rep* ∝ *low*) *no* ∉ *set ns*
  assume *same-var*: ∀ *no1*∈*set ns*. ∀ *no2*∈*set ns*. *var no1* = *var no2*
  assume *share-case*: (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no* ⟶ *high no* = *Null*
  assume *unmodif*: ∀ *no*. *no* ∉ *set ns* ⟶ *rep no* = *repa no*
  show *hd* [*sn*←*ns* . *repNodes-eq sn no low high repa*] ∈ *set ns* ∧
    repa (*hd* [*sn*←*ns* . *repNodes-eq sn no low high repa*]) =
    hd [*sn*←*ns* . *repNodes-eq sn no low high repa*]
  proof −
    from *no-in-ns pre* **obtain**
      no-nNull: *no* ≠ *Null* **and**
      no-balanced: (*low no* = *Null*) = (*high no* = *Null*) **and**
      isLeaf-var: *isLeaf-pt no low high* = (*var no* ≤ *Suc 0*)
      by *blast*
    have *repNodes-eq-same-node*: *repNodes-eq no no low high repa*
      by (*simp add*: *repNodes-eq-def*)
    from *no-in-ns* **have** *ns-nempty*: *ns* ≠ []
      by *auto*
    from *no-in-ns repNodes-eq-same-node*
    have *repNodes-not-empty*: [*sn*←*ns* . *repNodes-eq sn no low high repa*] ≠ []
      by (*rule filter-not-empty*)
    then have *hd-term-in-ns*: *hd* [*sn*←*ns* . *repNodes-eq sn no low high repa*] ∈ *set*
ns

```
      by (rule hd-filter-in-list)
    with while-inv obtain
      repa-hd-nNull: repa  (hd [sn←ns . repNodes-eq sn no low high repa]) ≠ Null
      by auto
    let ?hd = hd [sn←ns . repNodes-eq sn no low high repa]
    from hd-term-in-ns  pre obtain
      hd-nNull:  ?hd ≠ Null and
      hd-balanced:
        (low (hd [sn←ns . repNodes-eq sn no low high repa]) = Null) =
         (high (hd [sn←ns . repNodes-eq sn no low high repa]) = Null) and
      hd-isLeaf-var:
      isLeaf-pt (hd [sn←ns . repNodes-eq sn no low high repa]) low high =
      (var (hd [sn←ns . repNodes-eq sn no low high repa]) ≤ Suc 0)
      by blast
    have repa (hd [sn←ns . repNodes-eq sn no low high repa]) =
      hd [sn←ns . repNodes-eq sn no low high repa]
    proof (cases high no = Null)
      case True
      with no-balanced have low no = Null
        by simp
      with True have no-Leaf: isLeaf-pt no low high
        by (simp add: isLeaf-pt-def)
      with isLeaf-var have varno: var no <= 1
        by simp
      from same-var [rule-format, OF no-in-ns hd-term-in-ns] varno
      have var (hd [sn←ns . repNodes-eq sn no low high repa]) ≤ 1
        by simp
      with hd-isLeaf-var have
        isLeaf-pt (hd [sn←ns . repNodes-eq sn no low high repa]) low high
        by simp
      with while-inv hd-term-in-ns repNodes-not-empty show ?thesis
        apply (simp add: isLeaf-pt-def)
        apply (erule-tac x=
          hd [sn←ns . repNodes-eq sn no low high repa] in ballE)
        prefer 2
        apply simp
        apply (simp (no-asm-use) add: repNodes-eq-def)
        apply (rule filter-hd-P-rep-indep)
        apply  (simp (no-asm-simp))
        apply  (simp (no-asm-simp))
        apply assumption
        done
    next
      assume hno-nNull:  high no ≠ Null
      with share-case have repchildren-neq: (repa ∝ low) no ≠ (repa ∝ high) no
        by simp
      from repNodes-not-empty have
          repNodes-eq  (hd [sn←ns . repNodes-eq sn no low high repa]) no low high
repa
```

by (*rule hd-filter-prop*)
then
have (*repa* ∝ *low*) (*hd* [*sn*←*ns* . *repNodes-eq sn no low high repa*]) =
(*repa* ∝ *low*) *no* ∧
(*repa* ∝ *high*) (*hd* [*sn*←*ns* . *repNodes-eq sn no low high repa*]) =
(*repa* ∝ *high*) *no*
by (*simp add*: *repNodes-eq-def*)
with *repchildren-neq* **have**
(*repa* ∝ *low*) (*hd* [*sn*←*ns* . *repNodes-eq sn no low high repa*])
≠ (*repa* ∝ *high*) (*hd* [*sn*←*ns* . *repNodes-eq sn no low high repa*])
by *simp*
with *while-inv hd-term-in-ns repNodes-not-empty* **show** *?thesis*
apply (*simp add*: *isLeaf-pt-def*)
apply (*erule-tac x*=
hd [*sn*←*ns* . *repNodes-eq sn no low high repa*] **in** *ballE*)
prefer *2*
apply *simp*
apply (*simp* (*no-asm-use*) *add*: *repNodes-eq-def*)
apply (*rule filter-hd-P-rep-indep*)
apply *simp*
apply *fastforce*
apply *fastforce*
done
qed
with *hd-term-in-ns*
show *?thesis*
by *simp*
qed
next

fix *var low high rep nodeslist repa next node prx sfx*
assume *ns*: *List nodeslist next* (*prx* @ *node* # *sfx*)
assume *sfx*: *List* (*next node*) *next sfx*
assume *node-not-Null*: *node* ≠ *Null*
assume *nodes-balanced-ordered*: ∀ *no*∈*set* (*prx* @ *node* # *sfx*).
no ≠ *Null* ∧
(*low no* = *Null*) = (*high no* = *Null*) ∧
low no ∉ *set* (*prx* @ *node* # *sfx*) ∧
high no ∉ *set* (*prx* @ *node* # *sfx*) ∧
isLeaf-pt no low high = (*var no* ≤ (*1*::*nat*)) ∧
(*low no* ≠ *Null* ⟶ *rep* (*low no*) ≠ *Null*) ∧
(*rep* ∝ *low*) *no* ∉ *set* (*prx* @ *node* # *sfx*)
assume *all-nodes-same-var*: ∀ *no1*∈*set* (*prx* @ *node* # *sfx*).
∀ *no2*∈*set* (*prx* @ *node* # *sfx*). *var no1* = *var no2*
assume *rep-repa-nc*: ∀ *no*. *no* ∉ *set prx* ⟶ *rep no* = *repa no*
assume *while-inv*: ∀ *no*∈*set prx*.
repa no ≠ *Null* ∧
(*if* (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no* ∧ *low no* ≠ *Null*
then repa no = (*repa* ∝ *low*) *no*

$else\ repa\ no = hd\ [sn \leftarrow prx\ .\ repNodes\text{-}eq\ sn\ no\ low\ high\ repa] \wedge$
$repa\ (repa\ no) = repa\ no \wedge$
$(\forall\ no1 \in set\ prx.$
$((repa \propto high)\ no1 = (repa \propto high)\ no \wedge$
$(repa \propto low)\ no1 = (repa \propto low)\ no) =$
$(repa\ no = repa\ no1)))$

assume *not-Leaf*: $\neg\ isLeaf\text{-}pt\ node\ low\ high$
assume *repchildren-eq-nln*: $repa\ (low\ node) = repa\ (high\ node)$
show $(\forall\ no.\ no \notin set\ (prx\ @\ [node]) \longrightarrow$
$rep\ no = (repa(node := repa\ (high\ node)))\ no) \wedge$
$(\forall\ no \in set\ (prx\ @\ [node]).$
$(repa(node := repa\ (high\ node)))\ no \neq Null \wedge$
$(if\ (repa(node := repa\ (high\ node)) \propto low)\ no =$
$(repa(node := repa\ (high\ node)) \propto high)\ no \wedge$
$low\ no \neq Null$
$then\ (repa(node := repa\ (high\ node)))\ no =$
$(repa(node := repa\ (high\ node)) \propto low)\ no$
$else\ (repa(node := repa\ (high\ node)))\ no =$
$hd\ [sn \leftarrow prx\ @\ [node]\ .$
$repNodes\text{-}eq\ sn\ no\ low\ high$
$(repa(node := repa\ (high\ node)))] \wedge$
$(repa(node := repa\ (high\ node)))$
$((repa(node := repa\ (high\ node)))\ no) =$
$(repa(node := repa\ (high\ node)))\ no \wedge$
$(\forall\ no1 \in set\ (prx\ @\ [node]).$
$((repa(node := repa\ (high\ node)) \propto high)\ no1 =$
$(repa(node := repa\ (high\ node)) \propto high)\ no \wedge$
$(repa(node := repa\ (high\ node)) \propto low)\ no1 =$
$(repa(node := repa\ (high\ node)) \propto low)\ no) =$
$((repa(node := repa\ (high\ node)))\ no =$
$(repa(node := repa\ (high\ node)))\ no1))))$
(**is** *?NodesUnmodif* $\wedge$ *?NodesModif*)
proof −
— This proof was originally conducted without the substitution *repa* (*low node*) = *repa* (*high node*) preformed. So don't be confused if we show everythin for *repa* (*low node*).
from *rep-repa-nc*
have *nodes-unmodif*: *?NodesUnmodif*
by *auto*
hence *rep-Sucna-nc*:
$(\forall\ no.\ no \notin set\ (prx\ @\ [node])$
$\longrightarrow rep\ no = (repa(node := repa\ (low\ (node\ ))))\ no)$
by *auto*
have *nodes-modif*: *?NodesModif* (**is** $\forall\ no \in set\ (prx\ @\ [node]).\ ?P\ no \wedge ?Q\ no$)
proof (*rule ballI*)
fix *no*
assume *no-in-take-Sucna*: $no \in set\ (prx\ @\ [node])$
show $?P\ no \wedge ?Q\ no$
proof (*cases* $no = node$)

case *False*
note *no-noteq-nln=this*
with *no-in-take-Sucna*
have *no-in-take-n*: *no* ∈ *set prx*
  by *auto*
with *no-in-take-n while-inv* **obtain**
  repa-no-nNull: *repa no* ≠ *Null* **and**
  repa-cases: (*if* (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no* ∧ *low no* ≠ *Null*
  then repa no = (*repa* ∝ *low*) *no*
  else repa no = *hd* [*sn*←*prx* . *repNodes-eq sn no low high repa*]
  ∧ *repa* (*repa no*) = *repa no* ∧
  (∀ *no1*∈*set prx*. ((*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no*
  ∧ (*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*)
  = (*repa no* = *repa no1*)))
  using [[*simp-depth-limit* = *2*]]
  by *auto*
from *no-in-take-n*
have *no-in-nodeslist*: *no* ∈ *set* (*prx* @ *node* # *sfx*)
  by *auto*
from *repa-no-nNull no-noteq-nln* **have** *ext-repa-nNull*: *?P no*
  by *auto*
from *no-in-nodeslist nodes-balanced-ordered* **obtain**
  nln-nNull: *node* ≠ *Null* **and**
  nln-balanced-children: (*low node* = *Null*) = (*high node* = *Null*) **and**
  lnln-notin-nodeslist: *low node* ∉ *set* (*prx* @ *node* # *sfx*) **and**
  hnln-notin-nodeslist: *high node* ∉ *set* (*prx* @ *node* # *sfx*) **and**
  isLeaf-var-nln: *isLeaf-pt node low high* = (*var node* ≤ *1*) **and**
  node-nNull-rap-nNull-nln: (*low node* ≠ *Null*
  ⟶ *rep* (*low node*) ≠ *Null*) **and**
  nln-varrep-le-var: (*rep* ∝ *low*) *node* ∉ *set* (*prx* @ *node* # *sfx*)
  apply (*erule-tac x*=*node* **in** *ballE*)
  apply *auto*
  done
from *no-in-nodeslist nodes-balanced-ordered no-in-take-Sucna*
obtain
  no-nNull: *no* ≠ *Null* **and**
  balanced-children: (*low no* = *Null*) = (*high no* = *Null*) **and**
  lno-notin-nodeslist: *low no* ∉ *set* (*prx* @ *node* # *sfx*) **and**
  hno-notin-nodeslist: *high no* ∉ *set* (*prx* @ *node* # *sfx*) **and**
  isLeaf-var-no: *isLeaf-pt no low high* = (*var no* ≤ *1*) **and**
  node-nNull-rep-nNull: (*low no* ≠ *Null* ⟶ *rep* (*low no*) ≠ *Null*) **and**
  varrep-le-var: (*rep* ∝ *low*) *no* ∉ *set* (*prx* @ *node* # *sfx*)
  apply −
  apply (*erule-tac x*=*no* **in** *ballE*)
  apply *auto*
  done
from *lno-notin-nodeslist*
have *ext-rep-null-comp-low*:
  (*repa* (*node* := *repa* (*low node*)) ∝ *low*) *no* = (*repa* ∝ *low*) *no*

by (*auto simp add*: *null-comp-def*)
from *hno-notin-nodeslist*
have *ext-rep-null-comp-high*:
(*repa* (*node* := *repa* (*low node*)) ∝ *high*) *no* = (*repa* ∝ *high*) *no*
by (*auto simp add*: *null-comp-def*)
have *share-reduce-if*: *?Q no*
proof (*cases* (*repa* (*node* := *repa* (*low node*)) ∝ *low*) *no* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* ∧ *low no* ≠ *Null*)
case *True*
then obtain
red-case: (*repa* (*node* := *repa* (*low node*)) ∝ *low*) *no* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* **and**
lno-nNull: *low no* ≠ *Null*
by *simp*
from *lno-nNull balanced-children* **have** *hno-nNull*: *high no* ≠ *Null*
by *simp*
from *True ext-rep-null-comp-low ext-rep-null-comp-high*
have *repchildren-eq-no*: (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no*
by *simp*
with *repa-cases lno-nNull* **have** *repa no* = (*repa* ∝ *low*) *no*
by *auto*
with *ext-rep-null-comp-low no-noteq-nln*
have (*repa*(*node* := *repa* (*low node*))) *no* =
(*repa* (*node* := *repa* (*low node*)) ∝ *low*) *no*
by *simp*
with *True repchildren-eq-nln* **show** *?thesis*
by *auto*
next
assume *share-case-ext*:
¬ ((*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* ∧ *low no* ≠ *Null*)
from *not-Leaf isLeaf-var-nln*
have *1* < *var node*
by *simp*
with *all-nodes-same-var*
have *all-nodes-nl-Suc0-l-var*: ∀ *x* ∈ *set* (*prx* @ *node* # *sfx*). *1* < *var x*
using [[*simp-depth-limit=1*]]
by *auto*
with *nodes-balanced-ordered*
have *all-nodes-nl-noLeaf*:
∀ *x* ∈ *set* (*prx* @ *node* # *sfx*). ¬ *isLeaf-pt x low high*
apply −
apply *rule*
apply (*drule-tac x=x* **in** *bspec*,*assumption*)
apply (*drule-tac x=x* **in** *bspec*,*assumption*)
apply *auto*
done
from *nodes-balanced-ordered*
have *all-nodes-nl-balanced*:

$\forall x \in set\ (prx\ @\ node\ \#\ sfx).\ (low\ x = Null) = (high\ x = Null)$
apply −
apply *rule*
apply (*drule-tac x=x* **in** *bspec*,*assumption*)
apply *auto*
done
from *all-nodes-nl-Suc0-l-var no-in-nodeslist*
have *Suc0-l-var-no*: $1 < var\ no$
by *auto*
with *isLeaf-var-no* **have** *no-nLeaf*: $\neg$ *isLeaf-pt no low high*
by *simp*
with *balanced-children* **have** *lno-nNull*: $low\ no \neq Null$
by (*simp add*: *isLeaf-pt-def*)
with *balanced-children* **have** *hno-nNull*: $high\ no \neq Null$
by *simp*
with *share-case-ext ext-rep-null-comp-low ext-rep-null-comp-high lno-nNull*

have *repchildren-neq-no*: $(repa \propto low)\ no \neq (repa \propto high)\ no$
by (*simp add*: *null-comp-def*)
with *repa-cases*
have *share-case-inv*:
$repa\ no = hd\ [sn{\leftarrow}prx\ .\ repNodes\text{-}eq\ sn\ no\ low\ high\ repa] \wedge$
$repa\ (repa\ no) = repa\ no \wedge$
$(\forall\ no1{\in}set\ prx.\ ((repa \propto high)\ no1 = (repa \propto high)\ no \wedge$
$(repa \propto low)\ no1 = (repa \propto low)\ no) = (repa\ no = repa\ no1))$
by *auto*
then have *repa-no*: $repa\ no = hd\ [sn{\leftarrow}prx\ .\ repNodes\text{-}eq\ sn\ no\ low\ high$
$repa]$
by *simp*
from *Suc0-l-var-no* **have** $\forall x \in set\ (prx\ @\ node\ \#\ sfx).\ 1 < var\ no$
by *auto*
from *no-in-take-n* **have** $[sn{\leftarrow}prx\ .\ repNodes\text{-}eq\ sn\ no\ low\ high\ repa] \neq []$
apply −
apply (*rule filter-not-empty*)
apply (*auto simp add*: *repNodes-eq-def*)
done
then have *repNodes-eq*
$(hd\ [sn{\leftarrow}prx.\ repNodes\text{-}eq\ sn\ no\ low\ high\ repa])\ no\ low\ high\ repa$
by (*rule hd-filter-prop*)
with *repa-no*
have *rep-children-eq-no-repa-no*:
$(repa \propto low)\ (repa\ no) = (repa \propto low)\ no \wedge$
$(repa \propto high)\ (repa\ no) = (repa \propto high)\ no$
by (*simp add*: *repNodes-eq-def*)
from *lno-notin-nodeslist rep-repa-nc*
have *rep-repa-nc-low-no*: $rep\ (low\ no) = repa\ (low\ no)$
apply −
apply (*erule-tac x=low no* **in** *allE*)
apply *auto*

done
have ∀ *x* ∈ *set* (*prx* @ [*node*]).
repNodes-eq x no low high (*repa*(*node* := *repa* (*low node*))) =
repNodes-eq x no low high repa
proof (*rule ballI*, *unfold repNodes-eq-def*)
fix *x*
assume *x-in-take-Sucn*: *x* ∈ *set* (*prx* @ [*node*])
hence *x-in-nodeslist*: *x* ∈ *set* (*prx* @ *node* # *sfx*)
by *auto*
with *all-nodes-nl-noLeaf nodes-balanced-ordered*
have *children-nNull-x*: *low x* ≠ *Null* ∧ *high x* ≠ *Null*
apply −
apply (*drule-tac x*=*x* **in** *bspec*,*assumption*)
apply (*drule-tac x*=*x* **in** *bspec*,*assumption*)
apply (*auto simp add*: *isLeaf-pt-def*)
done
from *x-in-nodeslist nodes-balanced-ordered*
have *low x* ∉ *set* (*prx* @ *node* # *sfx*) ∧ *high x* ∉ *set* (*prx* @ *node* # *sfx*)
apply −
apply (*drule-tac x*=*x* **in** *bspec*,*assumption*)
apply *auto*
done
with *lno-notin-nodeslist hno-notin-nodeslist*
children-nNull-x lno-nNull hno-nNull
show ((*repa*(*node* := *repa* (*low node*)) ∝ *high*) *x* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* ∧
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *x* =
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no*) =
((*repa* ∝ *high*) *x* = (*repa* ∝ *high*) *no* ∧
(*repa* ∝ *low*) *x* = (*repa* ∝ *low*) *no*)
by (*simp add*: *null-comp-def*)
qed
then have *filter-extrep-rep*:
[*sn*←(*prx* @ [*node*]). *repNodes-eq sn no low high*
(*repa*(*node* := *repa* (*low node*)))] =
[*sn*←(*prx* @ [*node*]) . *repNodes-eq sn no low high repa*]
by (*rule P-eq-list-filter*)
from *no-in-take-n*
have *filter-n-notempty*: [*sn*←*prx*. *repNodes-eq sn no low high repa*] ≠ []
apply (*rule filter-not-empty*)
apply (*simp add*: *repNodes-eq-def*)
done
then have *hd* [*sn*←*prx*. *repNodes-eq sn no low high repa*] =
hd [*sn*←*prx*@[*node*]. *repNodes-eq sn no low high repa*]
by *auto*
with *no-noteq-nln filter-extrep-rep repa-no*
have *ext-repa-no*: (*repa*(*node*:= *repa* (*low node*))) *no* =
hd [*sn*←*prx*@[*node*] . *repNodes-eq sn no low high*
(*repa*(*node* := *repa* (*low node*)))]

by *simp*
have (*repa*(*node* := *repa* (*low node*))) (*repa no*) = *repa no*
proof (*cases repa no* = *node*)
case *True*
note *rno-nln*=*this*
from *rep-repa-nc-low-no rep-children-eq-no-repa-no lno-nNull node-nNull-rep-nNull*
have *low-rep-no-nNull*: *low* (*repa no*) ≠ *Null*
apply (*simp add*: *null-comp-def*)
apply *auto*
done
with *nodes-balanced-ordered rno-nln*
have *high-rap-no-nNull*: *high* (*repa no*) ≠ *Null*
apply −
apply (*drule-tac x*=*repa no* **in** *bspec*)
apply *auto*
done
with *low-rep-no-nNull rno-nln rep-children-eq-no-repa-no*
have *repa* (*low node*) = (*repa* ∝ *low*) *no* ∧
repa (*high node*) = (*repa* ∝ *high*) *no*
by (*simp add*: *null-comp-def*)
with *repchildren-eq-nln* **have** (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no*
by *simp*
with *repchildren-neq-no* **show** *?thesis*
by *simp*
next
assume *rno-not-nln*: *repa no* ≠ *node*
from *share-case-inv* **have** *repa* (*repa no*) = *repa no*
by *auto*
with *rno-not-nln* **show** *?thesis*
by *simp*
qed
with *no-noteq-nln* **have** *ext-repa-ext-repa*:
(*repa*(*node* := *repa* (*low node*)))
((*repa*(*node* := *repa* (*low node*))) *no*)
= (*repa*(*node* := *repa* (*low node*))) *no*
by *simp*
have (∀ *no1*∈*set* (*prx*@[*node*]).
((*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no1* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* ∧
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no1* =
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no*) =
((*repa*(*node* := *repa* (*low node*))) *no* =
(*repa*(*node* := *repa* (*low node*))) *no1*))
proof (*rule ballI*)
fix *no1*
assume *no1-in-take-Sucn*: *no1* ∈ *set* (*prx*@[*node*])
hence *no1-in-nodeslist*: *no1* ∈ *set* (*prx* @ *node* # *sfx*)
by *auto*

show ((*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no1* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* ∧
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no1* =
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no*) =
((*repa*(*node* := *repa* (*low node*))) *no* =
(*repa*(*node* := *repa* (*low node*))) *no1*)
proof (*cases no1* = *node*)
case *True*
show *?thesis*
proof (*rule*, *elim conjE*)
assume *ext-repa-no-no1*:
(*repa*(*node* := *repa* (*low node*))) *no* =
(*repa*(*node* := *repa* (*low node*))) *no1*
with *True no-noteq-nln*
have *repa-no-repa-low-nln*: *repa no* = *repa* (*low node*)
by *simp*
from *filter-n-notempty*
have *repa-no-in-take-n*:
hd [*sn*←*prx*. *repNodes-eq sn no low high repa*]
∈ *set prx*
apply −
apply (*rule hd-filter-in-list*)
apply *auto*
done
with *repa-no*
have *repa-no-in-nodeslist*: *repa no* ∈ *set* (*prx* @ *node* # *sfx*)
by *auto*
from *lnln-notin-nodeslist rep-repa-nc*
have *rep-repa-low-nln*: *rep* (*low node*) = *repa* (*low node*)
by *auto*
from *all-nodes-nl-noLeaf nln-balanced-children*
have *low node* ≠ *Null*
by (*auto simp add*: *isLeaf-pt-def*)
with *rep-repa-low-nln lnln-notin-nodeslist lno-nNull*
nln-varrep-le-var
have *repa* (*low node*) ∉ *set* (*prx* @ *node* # *sfx*)
by (*simp add*: *null-comp-def*)
with *repa-no-repa-low-nln repa-no-in-nodeslist*
show (*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no1* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* ∧
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no1* =
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no*
by *simp*
next
assume *no-no1-high*:
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no1* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no*
assume *no-no1-low*:
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no1* =

(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no*
from *True repchildren-eq-nln*
have *repachildren-eq-no1*: *repa* (*low no1*) = *repa* (*high no1*)
by *simp*
from *not-Leaf True nln-balanced-children*
have *children-nNull-no1*: (*low no1*) ≠ *Null* ∧ *high no1* ≠ *Null*
by (*simp add*: *isLeaf-pt-def*)
with *repachildren-eq-no1*
have *repchildren-eq-no1*: (*repa* ∝ *low*) *no1* = (*repa* ∝ *high*) *no1*
by (*simp add*: *null-comp-def*)
from *no-no1-low children-nNull-no1 lno-nNull*
lnln-notin-nodeslist lno-notin-nodeslist True
have *rep-low-eq-no-no1*: (*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*
by (*simp add*: *null-comp-def*)
from *no-no1-high children-nNull-no1 hno-nNull*
hnln-notin-nodeslist hno-notin-nodeslist True
have *rep-high-eq-no-no1*: (*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no*
by (*simp add*: *null-comp-def*)
with *rep-low-eq-no-no1 repchildren-eq-no1*
have (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no*
by *simp*
with *repchildren-neq-no*
show (*repa*(*node* := *repa* (*low node*))) *no* =
(*repa*(*node* := *repa* (*low node*))) *no1*
by *simp*
qed
next
assume *no1-neq-nln*: *no1* ≠ *node*
from *no1-in-nodeslist nodes-balanced-ordered*
have *children-notin-nl-no1*:
low no1 ∉ *set* (*prx* @ *node* # *sfx*) ∧ *high no1* ∉ *set* (*prx* @ *node* # *sfx*)
apply −
apply (*drule-tac x=no1* **in** *bspec*,*assumption*)
by *auto*
from *no1-neq-nln no1-in-take-Sucn*
have *no1-in-take-n*: *no1* ∈ *set prx*
by *auto*
from *no1-in-nodeslist all-nodes-nl-noLeaf all-nodes-nl-balanced*
have *children-nNull-no1*: (*low no1*) ≠ *Null* ∧ *high no1* ≠ *Null*
by (*auto simp add*: *isLeaf-pt-def*)
show *?thesis*
proof (*rule*, *elim conjE*)
assume *ext-repa-high-no1-no*:
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no1*
= (*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no*
assume *ext-repa-low-no1-no*:
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no1*
= (*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no*
from *children-nNull-no1 hno-nNull ext-repa-high-no1-no*

children-notin-nl-no1
hno-notin-nodeslist
have *repa-high-no1-no*: $(repa \propto high)\ no1 = (repa \propto high)\ no$
by (*simp add*: *null-comp-def*)
from *children-nNull-no1 lno-nNull ext-repa-low-no1-no*
children-notin-nl-no1 lno-notin-nodeslist
have *repa-low-no1-no*: $(repa \propto low)\ no1 = (repa \propto low)\ no$
by (*simp add*: *null-comp-def*)
from *repchildren-neq-no repa-high-no1-no repa-low-no1-no*
have $(repa \propto low)\ no1 \neq (repa \propto high)\ no1$
by *simp*
from *no1-in-take-n share-case-inv repa-high-no1-no repa-low-no1-no*
have *repa no* = *repa no1*
by *auto*
with *no-noteq-nln no1-neq-nln*
show $(repa(node := repa\ (low\ node)))\ no =$
$(repa(node := repa\ (low\ node)))\ no1$
by *simp*
next
assume $(repa(node := repa\ (low\ node)))\ no =$
$(repa(node := repa\ (low\ node)))\ no1$
with *no-noteq-nln no1-neq-nln*
have *repa no* = *repa no1*
by *simp*
with *share-case-inv no1-in-take-n*
have $((repa \propto high)\ no1 = (repa \propto high)\ no \wedge$
$(repa \propto low)\ no1 = (repa \propto low)\ no)$
by *auto*
with *children-notin-nl-no1 children-nNull-no1 lno-notin-nodeslist*
hno-notin-nodeslist lno-nNull hno-nNull
show $(repa(node := repa\ (low\ node)) \propto high)\ no1 =$
$(repa(node := repa\ (low\ node)) \propto high)\ no \wedge$
$(repa(node := repa\ (low\ node)) \propto low)\ no1 =$
$(repa(node := repa\ (low\ node)) \propto low)\ no$
by (*auto simp add*: *null-comp-def*)
qed
qed
qed
from *ext-repa-ext-repa ext-repa-no share-case-ext repchildren-eq-nln this*
show *?thesis*
using [[*simp-depth-limit=4*]]
by *auto*
qed
with *ext-repa-nNull* **show** *?thesis*
by *auto*
next
assume *no-nln*: *no* = *node*
hence *no-in-nodeslist*: $no \in set\ (prx\ @\ node\ \#\ sfx)$
by *simp*

from *no-nln not-Leaf no-in-nodeslist*
nodes-balanced-ordered [*rule-format*, *OF this*] **obtain**
low-no-nNull: *low no* ≠ *Null* **and**
high-no-nNull: *high no* ≠ *Null* **and**
rep-low-no-nNull: *rep* (*low no*) ≠ *Null* **and**
lno-notin-nl: *low no* ∉ *set* (*prx* @ *node* # *sfx*) **and**
hno-notin-nl: *high no* ∉ *set* (*prx* @ *node* # *sfx*) **and**
children-nNull-no: (*low no* ≠ *Null*) ∧ (*high no* ≠ *Null*)
apply (*unfold isLeaf-pt-def*)
apply *blast*
done
then have *low no* ∉ *set prx*
by *auto*
with *rep-repa-nc no-nln rep-low-no-nNull*
have (*repa*(*node* := *repa* (*low node*))) *no* ≠ *Null*
by *simp*
moreover
have (*if* (*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* ∧ *low no* ≠ *Null*
then (*repa*(*node* := *repa* (*low node*))) *no* =
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no*
else (*repa*(*node* := *repa* (*low node*))) *no* =
hd [*sn*←*prx*@[*node*]. *repNodes-eq sn no low high*
(*repa*(*node* := *repa* (*low node*)))] ∧
(*repa*(*node* := *repa* (*low node*)))
((*repa*(*node* := *repa* (*low node*))) *no*) =
(*repa*(*node* := *repa* (*low node*))) *no* ∧
(∀ *no1*∈*set* (*prx*@[*node*]).
((*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no1* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* ∧
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no1* =
(*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no*) =
((*repa*(*node* := *repa* (*low node*))) *no* =
(*repa*(*node* := *repa* (*low node*))) *no1*)))
proof (*cases* (*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no* =
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no* ∧ *low no* ≠ *Null*)
case *True*
note *red-case*=*this*
with *children-nNull-no lno-notin-nl hno-notin-nl*
have (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no*
by (*auto simp add*: *null-comp-def*)
from *children-nNull-no lno-notin-nl*
have *ext-repa-eq-repa-low*: (*repa*(*node* := *repa* (*low node*)) ∝ *low*) *no*
= (*repa* ∝ *low*) *no*
by (*auto simp add*: *null-comp-def*)
from *children-nNull-no hno-notin-nl*
have *ext-repa-eq-repa-high*:
(*repa*(*node* := *repa* (*low node*)) ∝ *high*) *no*
= (*repa* ∝ *high*) *no*

```
          by (auto simp add: null-comp-def)
        from no-nln children-nNull-no
        have repa (low node) = (repa ∝ low) no
          by (simp add: null-comp-def)
        with red-case ext-repa-eq-repa-high ext-repa-eq-repa-low no-nln
        show ?thesis
          using [[simp-depth-limit=2]]
          by (auto simp del: null-comp-not-Null)
      next
        assume share-case: ¬ ((repa(node := repa (low node)) ∝ low) no
          = (repa(node := repa (low node)) ∝ high) no ∧ low no ≠ Null)
        with low-no-nNull have (repa(node := repa (low node)) ∝ low) no
          ≠ (repa(node := repa (low node)) ∝ high) no
          by simp
        with children-nNull-no lno-notin-nl hno-notin-nl
        have (repa ∝ low) no ≠ (repa ∝ high) no
          by (auto simp add: null-comp-def)
        with children-nNull-no have repa (low no) ≠ repa (high no)
          by (simp add: null-comp-def)
        with repchildren-eq-nln no-nln show ?thesis
          by simp
      qed
      ultimately show ?thesis
        using repchildren-eq-nln
        apply −
        apply (simp only:)
        apply (simp (no-asm))
        done
    qed
  qed
  from nodes-unmodif nodes-modif
  show ?thesis by iprover
 qed
next
 fix var low high rep nodeslist repa next node prx sfx repb
 assume ns: List nodeslist next (prx @ node # sfx)
 assume sfx: List (next node) next sfx
 assume nodes-balanced-ordered: ∀ no∈set (prx @ node # sfx).
     no ≠ Null ∧
     (low no = Null) = (high no = Null) ∧
     low no ∉ set (prx @ node # sfx) ∧
     high no ∉ set (prx @ node # sfx) ∧
     isLeaf-pt no low high = (var no ≤ (1::nat)) ∧
     (low no ≠ Null ⟶ rep (low no) ≠ Null) ∧
     (rep ∝ low) no ∉ set (prx @ node # sfx)
 assume all-nodes-same-var: ∀ no1∈set (prx @ node # sfx).
     ∀ no2∈set (prx @ node # sfx). var no1 = var no2
 assume rep-repa-nc: ∀ no. no ∉ set prx ⟶ rep no = repa no
 assume while-inv: ∀ no∈set prx.
```

repa no ≠ Null ∧
(if (repa ∝ low) no = (repa ∝ high) no ∧ low no ≠ Null
then repa no = (repa ∝ low) no
else repa no = hd [sn←prx . repNodes-eq sn no low high repa] ∧
repa (repa no) = repa no ∧
(∀ no1∈set prx.
((repa ∝ high) no1 = (repa ∝ high) no ∧
(repa ∝ low) no1 = (repa ∝ low) no) =
(repa no = repa no1)))
assume *share-cond*:
¬ (¬ isLeaf-pt node low high ∧ repa (low node) = repa (high node))
assume *repb-node*:
repb node = hd [sn←prx @ node # sfx . repNodes-eq sn node low high repa]
assume *repa-repb-nc*: ∀ pt. pt ≠ node ⟶ repa pt = repb pt
assume *var-repb-node*: var (repb node) = var node
show (∀ no. no ∉ set (prx @ [node]) ⟶ rep no = repb no) ∧
(∀ no∈set (prx @ [node]).
repb no ≠ Null ∧
(if (repb ∝ low) no = (repb ∝ high) no ∧ low no ≠ Null
then repb no = (repb ∝ low) no
else repb no =
hd [sn←prx @ [node] . repNodes-eq sn no low high repb] ∧
repb (repb no) = repb no ∧
(∀ no1∈set (prx @ [node]).
((repb ∝ high) no1 = (repb ∝ high) no ∧
(repb ∝ low) no1 = (repb ∝ low) no) =
(repb no = repb no1))))
proof −
have *rep-repb-nc*: (∀ no. no ∉ set (prx @ [node]) ⟶ rep no = repb no)
proof (*intro allI impI*)
fix *no*
assume *no-notin-take-Sucn*: no ∉ set (prx @ [node])
with *rep-repa-nc*
have *rep-repa-nc-Sucn*: rep no = repa no
by *auto*
from *no-notin-take-Sucn* **have** no ≠ node
by *auto*
with *repa-repb-nc* **have** repa no = repb no
by *auto*
with *rep-repa-nc-Sucn* **show** rep no = repb no
by *simp*
qed
moreover
have *repb-no-share-def*:
(∀ no∈set (prx @ [node]).
¬ ((repb ∝ low) no = (repb ∝ high) no ∧ low no ≠ Null) ⟶
repb no = hd [sn←(prx @ [node]) . repNodes-eq sn no low high repb])
proof (*intro ballI impI*)
fix *no*

```
assume no-in-take-Sucn: no ∈ set (prx @ [node])
assume share-prop: ¬ ((repb ∝ low) no = (repb ∝ high) no ∧ low no ≠ Null)
from share-prop have share-or:
  (repb ∝ low) no ≠ (repb ∝ high) no ∨ low no = Null
  using [[simp-depth-limit=2]]
  by simp
from no-in-take-Sucn have no-in-nl: no ∈ set (prx @ node # sfx)
  by auto
from nodes-balanced-ordered [rule-format, OF this] obtain
  no-nNull: no ≠ Null and
  balanced-no: (low no = Null) = (high no = Null) and
  lno-notin-nl: low no ∉ set (prx @ node # sfx) and
  hno-notin-nl: high no ∉ set (prx @ node # sfx) and
  isLeaf-var-no: isLeaf-pt no low high = (var no ≤ 1)
  by auto
have nodes-notin-nl-neq-nln: ∀ p. p ∉ set (prx @ node # sfx) ⟶ p ≠ node
  by auto
show repb no = hd [sn←(prx @ [node]). repNodes-eq sn no low high repb]
proof (cases no = node)
  case False
  note no-notin-nl=this
  with no-in-take-Sucn have no-in-take-n: no ∈ set prx
    by auto
  from False repa-repb-nc have repb-repa-no: repb no = repa no
    by auto
  with while-inv [rule-format, OF no-in-take-n] no-in-take-n obtain
    repa-no-nNull: repa no ≠ Null and
    while-share-red-exp:
    (if (repa ∝ low) no = (repa ∝ high) no ∧ low no ≠ Null
        then repa no = (repa ∝ low) no
        else repa no = hd [sn←prx . repNodes-eq sn no low high repa] ∧
        repa (repa no) = repa no ∧
        (∀ no1∈set prx. ((repa ∝ high) no1 = (repa ∝ high) no ∧
        (repa ∝ low) no1 = (repa ∝ low) no) = (repa no = repa no1)))
    using [[simp-depth-limit = 2]]
    by auto
  from no-in-take-n
  have filter-take-n-notempty: [sn←prx.
    repNodes-eq sn no low high repa] ≠ []
    apply −
    apply (rule filter-not-empty)
    apply (auto simp add: repNodes-eq-def)
    done
  then have hd-term-n-Sucn:
    hd [sn←prx. repNodes-eq sn no low high repa]
        = hd [sn←prx@[node] . repNodes-eq sn no low high repa]
    by auto
  thus ?thesis
  proof (cases low no = Null)
```

```
  case True
  note lno-Null=this
  with balanced-no have hno-Null: high no = Null
    by simp
  from lno-Null hno-Null have isLeaf-no: isLeaf-pt no low high
    by (simp add: isLeaf-pt-def)
  from True while-share-red-exp
  have while-low-Null:
    repa no = hd [sn←prx. repNodes-eq sn no low high repa] ∧
     repa (repa no) = repa no ∧
     (∀ no1∈set prx. ((repa ∝ high) no1 = (repa ∝ high) no
     ∧ (repa ∝ low) no1 = (repa ∝ low) no) = (repa no = repa no1))
    by auto
  have all-nodes-in-nl-Leafs:
    ∀ x ∈ set (prx @ node # sfx). isLeaf-pt x low high
  proof (intro ballI)
    fix x
    assume x-in-nodeslist: x ∈ set (prx @ node # sfx)
    from isLeaf-no isLeaf-var-no have var no ≤ 1
      by simp
    with all-nodes-same-var [rule-format, OF x-in-nodeslist no-in-nl]
    have var x ≤ 1
      by simp
    with nodes-balanced-ordered [rule-format, OF x-in-nodeslist]
    show isLeaf-pt x low high
      by (auto simp add: isLeaf-pt-def)
  qed
  have ∀ x ∈ set (prx@[node]). repNodes-eq x no low high repb
        = repNodes-eq x no low high repa
  proof (rule ballI)
    fix x
    assume x-in-take-Sucn: x ∈ set (prx@[node])
    hence x-in-nodeslist: x ∈ set (prx @ node # sfx)
      by auto
    with all-nodes-in-nl-Leafs have isLeaf-pt x low high
      by auto
    with isLeaf-no repa-repb-nc show repNodes-eq x no low high repb
          = repNodes-eq x no low high repa
      by (simp add: repNodes-eq-def null-comp-def isLeaf-pt-def)
  qed
  then have  [sn←(prx@[node]). repNodes-eq sn no low high repa]
        = [sn←(prx@[node]) . repNodes-eq sn no low high repb]
    apply −
    apply (rule P-eq-list-filter)
    apply simp
    done
  with hd-term-n-Sucn while-low-Null repb-repa-no show ?thesis
    by auto
next
```

assume *lno-nNull*: *low no* ≠ *Null*
with *balanced-no* **have** *hno-nNull*: *high no* ≠ *Null*
  by *simp*
with *lno-nNull* **have** *no-nLeaf*: ¬ *isLeaf-pt no low high*
  by (*simp add*: *isLeaf-pt-def*)
with *isLeaf-var-no* **have** *Sucn-s-varno*: *1* < *var no*
  by *auto*
with *no-in-nl all-nodes-same-var*
have *all-nodes-nl-var*: ∀ *x* ∈ *set* (*prx* @ *node* # *sfx*). *1* < *var x*
  apply −
  apply (*rule ballI*)
  apply (*drule-tac x*=*no* **in** *bspec*,*assumption*)
  apply (*drule-tac x*=*x* **in** *bspec*,*assumption*)
  apply *auto*
  done
with *nodes-balanced-ordered*
have *all-nodes-nl-nLeaf*:
  ∀ *x* ∈ *set* (*prx* @ *node* # *sfx*). ¬ *isLeaf-pt x low high*
  apply −
  apply (*rule ballI*)
  apply (*drule-tac x*=*x* **in** *bspec*,*assumption*)
  apply (*drule-tac x*=*x* **in** *bspec*,*assumption*)
  apply *auto*
  done
from *lno-nNull share-or*
have *repbchildren-eq-no*: (*repb* ∝ *low*) *no* ≠ (*repb* ∝ *high*) *no*
  by *simp*
with *lno-nNull hno-nNull lno-notin-nl hno-notin-nl repa-repb-nc*
  nodes-notin-nl-neq-nln
have *repachildren-eq-no*: (*repa* ∝ *low*) *no* ≠ (*repa* ∝ *high*) *no*
  using [[*simp-depth-limit*=*2*]]
  by (*simp add*: *null-comp-def*)
with *while-share-red-exp*
have *repa-no-def*:
  repa no = *hd* [*sn*←*prx* . *repNodes-eq sn no low high repa*]
  by *auto*
with *no-notin-nl repa-repb-nc*
have *repb no* = *hd* [*sn*←*prx*. *repNodes-eq sn no low high repa*]
  by *simp*
with *hd-term-n-Sucn*
have *repb-no-hd-term-repa*: *repb no* =
    hd [*sn*←*prx*@[*node*] . *repNodes-eq sn no low high repa*]
  by *simp*
have ∀ *x* ∈ *set* (*prx*@[*node*]).
  repNodes-eq x no low high repa = *repNodes-eq x no low high repb*
proof (*intro ballI*)
  fix *x*
  assume *x-in-take-Sucn*: *x* ∈ *set* (*prx*@[*node*])
  hence *x-in-nodeslist*: *x* ∈ *set* (*prx* @ *node* # *sfx*)

```
        by auto
      with all-nodes-nl-nLeaf have x-nLeaf: ¬ isLeaf-pt x low high
        by auto
      from nodes-balanced-ordered [rule-format, OF x-in-nodeslist] obtain
        balanced-x: (low x = Null) = (high x = Null) and
        lx-notin-nl: low x ∉ set (prx @ node # sfx) and
        hx-notin-nl: high x ∉ set (prx @ node # sfx)
        by auto
      with nodes-notin-nl-neq-nln lno-notin-nl hno-notin-nl lno-nNull
        hno-nNull repa-repb-nc
      show  repNodes-eq x no low high repa = repNodes-eq x no low high repb
        by (simp add: repNodes-eq-def null-comp-def)
    qed
    then have  [sn←(prx@[node]). repNodes-eq sn no low high repa] =
      [sn←(prx@[node]). repNodes-eq sn no low high repb]
      apply −
      apply (rule P-eq-list-filter)
      apply auto
      done
    with repb-no-hd-term-repa show ?thesis
      by simp
  qed
next
  assume no-nln: no = node
  with repb-node have repb-no-def: repb no =
    hd [sn←(prx @ node # sfx). repNodes-eq sn node low high repa]
    by simp
  show ?thesis
  proof (cases isLeaf-pt no low high)
    case True
    note isLeaf-no=this
    have ∀ x ∈ set (prx @ node # sfx). repNodes-eq x no low high repb
      = repNodes-eq x no low high repa
    proof (rule ballI)
      fix x
      assume x-in-nodeslist: x ∈ set (prx @ node # sfx)
      have all-nodes-in-nl-Leafs:
        ∀ x ∈ set (prx @ node # sfx). isLeaf-pt x low high
      proof (intro ballI)
        fix x
        assume x-in-nodeslist:  x ∈ set (prx @ node # sfx)
        from isLeaf-no isLeaf-var-no have var no ≤ 1
          by simp
        with all-nodes-same-var [rule-format, OF x-in-nodeslist no-in-nl]
        have var x ≤ 1
          by simp
        with nodes-balanced-ordered [rule-format, OF x-in-nodeslist]
        show isLeaf-pt x low high
          by (auto simp add: isLeaf-pt-def)
```

qed
with *x-in-nodeslist* **have** *isLeaf-pt x low high*
by *auto*
with *isLeaf-no repa-repb-nc*
show *repNodes-eq x no low high repb* = *repNodes-eq x no low high repa*
by (*simp add*: *repNodes-eq-def null-comp-def isLeaf-pt-def*)
qed
with *repb-no-def no-nln* **have** *repb-no-whole-nl*: *repb no* =
hd [*sn*← (*prx* @ *node* # *sfx*). *repNodes-eq sn node low high repb*]
apply −
apply (*subgoal-tac*
[*sn*← (*prx*@*node*#*sfx*). *repNodes-eq sn node low high repa*]
= [*sn*←(*prx* @ *node* # *sfx*) . *repNodes-eq sn node low high repb*])
apply *simp*
apply (*rule P-eq-list-filter*)
apply *auto*
done
from *no-in-take-Sucn no-nln*
have [*sn*← (*prx*@[*node*]). *repNodes-eq sn node low high repb*] ≠ []
apply −
apply (*rule filter-not-empty*)
apply (*auto simp add*: *repNodes-eq-def*)
done
then
have *hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn node low high repb*] =
hd [*sn*←(*prx* @ *node* # *sfx*). *repNodes-eq sn node low high repb*]
apply −
apply (*rule hd-filter-app* [*symmetric*])
apply *auto*
done
with *repb-no-whole-nl no-nln* **show** *?thesis*
by *simp*
next
assume *no-nLeaf*: ¬ *isLeaf-pt no low high*
with *share-or balanced-no* **have** (*repb* ∝ *low*) *no* ≠ (*repb* ∝ *high*) *no*
using [[*simp-depth-limit=2*]]
by (*simp add*: *isLeaf-pt-def*)
from *no-nLeaf share-cond no-nln* **have** *repa* (*low no*) ≠ *repa* (*high no*)
by *auto*
with *no-nLeaf balanced-no* **have** (*repa* ∝ *low*) *no* ≠ (*repa* ∝ *high*) *no*
by (*simp add*: *null-comp-def isLeaf-pt-def*)
have ∀ *x* ∈ *set* (*prx*@*node*#*sfx*). *repNodes-eq x no low high repb*
= *repNodes-eq x no low high repa*
proof (*rule ballI*)
fix *x*
assume *x-in-nodeslist*: *x* ∈ *set* (*prx*@*node*#*sfx*)
have *all-nodes-in-nl-Leafs*:
∀ *x* ∈ *set* (*prx*@*node*#*sfx*). ¬ *isLeaf-pt x low high*
proof (*intro ballI*)

```
    fix x
    assume x-in-nodeslist: x ∈ set (prx@node#sfx)
    from no-nLeaf isLeaf-var-no have 1 < var no
      by simp
    with all-nodes-same-var [rule-format, OF x-in-nodeslist no-in-nl]
    have 1 < var x
      by auto
    with nodes-balanced-ordered [rule-format, OF x-in-nodeslist]
    show ¬ isLeaf-pt x low high
      apply (unfold isLeaf-pt-def)
      apply fastforce
      done
  qed
  with x-in-nodeslist have x-nLeaf: ¬ isLeaf-pt x low high
    by auto
  from nodes-balanced-ordered [rule-format, OF x-in-nodeslist]
  have (low x = Null) = (high x = Null)
        ∧ low x ∉ set (prx@node#sfx) ∧ high x ∉ set (prx@node#sfx)
    by auto
  with x-nLeaf balanced-no no-nLeaf repa-repb-nc
    nodes-notin-nl-neq-nln lno-notin-nl hno-notin-nl
  show repNodes-eq x no low high repb = repNodes-eq x no low high repa
    using [[simp-depth-limit=2]]
    by (simp add: repNodes-eq-def null-comp-def isLeaf-pt-def)
qed
with repb-no-def no-nln
have repb-no-whole-nl:
  repb no = hd [sn←(prx@node#sfx). repNodes-eq sn node low high repb]
  apply −
  apply (subgoal-tac
        [sn←(prx@node#sfx). repNodes-eq sn node low high repa]
        = [sn←(prx@node#sfx). repNodes-eq sn node low high repb])
  apply simp
  apply (rule P-eq-list-filter)
  apply auto
  done
from no-in-take-Sucn no-nln
have [sn←(prx@[node]) . repNodes-eq sn node low high repb]  ≠ []
  apply −
  apply (rule filter-not-empty)
  apply (auto simp add: repNodes-eq-def)
  done
then have
  hd [sn← (prx@[node]) . repNodes-eq sn node low high repb] =
  hd [sn←(prx@node#sfx) . repNodes-eq sn node low high repb]
  apply −
  apply (rule hd-filter-app [symmetric])
  apply auto
  done
```

with *repb-no-whole-nl no-nln* **show** *?thesis*
by *simp*
qed
qed
qed
have *repb-no-red-def*: (∀ *no*∈*set* (*prx*@[*node*]).(*repb* ∝ *low*) *no* = (*repb* ∝ *high*) *no*
∧ *low no* ≠ *Null* ⟶ *repb no* = (*repb* ∝ *low*) *no*)
proof (*intro ballI impI*)
fix *no*
assume *no-in-take-Sucn*: *no* ∈ *set* (*prx*@[*node*])
assume *red-cond-no*: (*repb* ∝ *low*) *no* = (*repb* ∝ *high*) *no* ∧ *low no* ≠ *Null*
from *no-in-take-Sucn* **have** *no-in-nl*: *no* ∈ *set* (*prx*@*node*#*sfx*)
by *auto*
from *nodes-balanced-ordered* [*rule-format*, *OF this*]**obtain**
no-nNull: *no* ≠ *Null* **and**
balanced-no: (*low no* = *Null*) = (*high no* = *Null*) **and**
lno-notin-nl: *low no* ∉ *set* (*prx*@*node*#*sfx*) **and**
hno-notin-nl: *high no* ∉ *set* (*prx*@*node*#*sfx*) **and**
isLeaf-var-no: *isLeaf-pt no low high* = (*var no* ≤ *1*)
by *auto*
have *nodes-notin-nl-neq-nln*: ∀ *p*. *p* ∉ *set* (*prx*@*node*#*sfx*) ⟶ *p* ≠ *node*
by *auto*
show *repb no* = (*repb* ∝ *low*) *no*
proof (*cases no* = *node*)
case *False*
note *no-notin-nl*=*this*
with *no-in-take-Sucn* **have** *no-in-take-n*: *no* ∈ *set prx*
by *auto*
from *False repa-repb-nc* **have** *repb-repa-no*: *repb no* = *repa no*
by *auto*
with *while-inv* [*rule-format*, *OF no-in-take-n*] **obtain**
repa-no-nNull: *repa no* ≠ *Null* **and**
while-share-red-exp:
(*if* (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no* ∧ *low no* ≠ *Null*
then repa no = (*repa* ∝ *low*) *no*
else repa no = *hd* [*sn*←*prx*. *repNodes-eq sn no low high repa*] ∧
repa (*repa no*) = *repa no* ∧
(∀ *no1*∈*set prx*. ((*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no* ∧
(*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*) = (*repa no* = *repa no1*)))
using [[*simp-depth-limit*=*2*]]
by *auto*
from *red-cond-no nodes-notin-nl-neq-nln lno-notin-nl*
hno-notin-nl while-share-red-exp balanced-no repa-repb-nc
have *red-repa-no*: *repa no* = (*repa* ∝ *low*) *no*
by (*auto simp add*: *null-comp-def*)
from *red-cond-no nodes-notin-nl-neq-nln lno-notin-nl repa-repb-nc*
have (*repb* ∝ *low*) *no* = (*repa* ∝ *low*) *no*
by (*auto simp add*: *null-comp-def*)

with *red-repa-no no-notin-nl balanced-no repa-repb-nc*
have *repb no* = (*repb* ∝ *low*) *no*
by *auto*
with *red-cond-no* **show** *?thesis*
by *auto*
next
assume *no* = *node*
with *share-cond*
have *share-cond-pre*:
isLeaf-pt no low high ∨ *repa* (*low no*) ≠ *repa* (*high no*)
by *simp*
show *?thesis*
proof (*cases isLeaf-pt no low high*)
case *True*
with *red-cond-no* **show** *?thesis*
by (*simp add*: *isLeaf-pt-def*)
next
assume *no-nLeaf*: ¬ *isLeaf-pt no low high*
with *share-cond-pre*
have *repa* (*low no*) ≠ *repa* (*high no*)
by *simp*
with *no-nLeaf lno-notin-nl hno-notin-nl nodes-notin-nl-neq-nln*
balanced-no repa-repb-nc
have *repb* (*low no*) ≠ *repb* (*high no*)
using [[*simp-depth-limit=2*]]
by (*auto simp add*: *isLeaf-pt-def*)
with *no-nLeaf balanced-no* **have** (*repb* ∝ *low*) *no* ≠ (*repb* ∝ *high*) *no*
by (*simp add*: *null-comp-def isLeaf-pt-def*)
with *red-cond-no* **show** *?thesis*
by *simp*
qed
qed
qed
have *while-while*: (∀ *no*∈*set* (*prx*@[*node*]).
repb no ≠ *Null* ∧
(*if* (*repb* ∝ *low*) *no* = (*repb* ∝ *high*) *no* ∧ *low no* ≠ *Null*
then repb no = (*repb* ∝ *low*) *no*
else repb no = *hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*] ∧
repb (*repb no*) = *repb no* ∧
(∀ *no1*∈*set* ((*prx*@[*node*])). ((*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*
∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*) = (*repb no* = *repb no1*))))
(**is** ∀ *no*∈*set* (*prx*@[*node*]). *?P no* ∧ *?Q no*)
proof (*intro ballI*)
fix *no*
assume *no-in-take-Sucn*: *no* ∈ *set* (*prx*@[*node*])
hence *no-in-nl*: *no* ∈ *set* (*prx*@*node*#*sfx*)
by *auto*
from *nodes-balanced-ordered* [*rule-format*, *OF this*] **obtain**
no-nNull: *no* ≠ *Null* **and**

balanced-no: (*low no* = *Null*) = (*high no* = *Null*) **and**
lno-notin-nl: *low no* ∉ *set* (*prx*@*node*#*sfx*) **and**
hno-notin-nl: *high no* ∉ *set* (*prx*@*node*#*sfx*) **and**
isLeaf-var-no: *isLeaf-pt no low high* = (*var no* ≤ *1*)
by *auto*
from *no-in-take-Sucn*
have *filter-take-Sucn-not-empty*:
[*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*] ≠ []
apply −
apply (*rule filter-not-empty*)
apply (*auto simp add*: *repNodes-eq-def*)
done
then have *hd-filter-Sucn-in-Sucn*:
hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*] ∈
set (*prx*@[*node*])
by (*rule hd-filter-in-list*)
have *nodes-notin-nl-neq-nln*: ∀ *p*. *p* ∉ *set* (*prx*@*node*#*sfx*) ⟶ *p* ≠ *node*
by *auto*
show *?P no* ∧ *?Q no*
proof (*cases no* = *node*)
case *False*
note *no-notin-nl*=*this*
with *no-in-take-Sucn*
have *no-in-take-n*: *no* ∈ *set prx*
by *auto*
from *False repa-repb-nc* **have** *repb-repa-no*: *repb no* = *repa no*
by *auto*
with *while-inv* [*rule-format*, *OF no-in-take-n*] **obtain**
repa-no-nNull: *repa no* ≠ *Null* **and**
while-share-red-exp:
(*if* (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no* ∧ *low no* ≠ *Null*
then repa no = (*repa* ∝ *low*) *no*
else repa no = *hd* [*sn*←*prx*. *repNodes-eq sn no low high repa*] ∧
repa (*repa no*) = *repa no* ∧
(∀ *no1*∈*set prx*. ((*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no* ∧
(*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*) = (*repa no* = *repa no1*)))
using [[*simp-depth-limit*=*2*]]
by *auto*
from *repb-repa-no repa-no-nNull* **have** *repb-no-nNull*: *?P no*
by *simp*
have *?Q no*
proof (*cases* (*repb* ∝ *low*) *no* = (*repb* ∝ *high*) *no* ∧ *low no* ≠ *Null*)
case *True*
with *no-in-take-Sucn repb-no-red-def* **show** *?thesis*
by *auto*
next
assume *share-case-repb*:
¬ ((*repb* ∝ *low*) *no* = (*repb* ∝ *high*) *no* ∧ *low no* ≠ *Null*)
with *repb-no-share-def no-in-take-Sucn*

have *repb-no-def*: *repb no* = *hd* [*sn*← (*prx*@[*node*]).
  repNodes-eq sn no low high repb]
  by *auto*
with *share-case-repb*
have (*repb* ∝ *low*) *no* ≠ (*repb* ∝ *high*) *no* ∨ *low no* = *Null*
  using [[*simp-depth-limit=2*]]
  by *simp*
thus *?thesis*
proof (*cases low no* = *Null*)
  case *True*
  note *lno-Null=this*
  with *balanced-no* **have** *hno-Null*: *high no* = *Null*
    by *simp*
  from *lno-Null hno-Null* **have** *isLeaf-no*: *isLeaf-pt no low high*
    by (*simp add*: *isLeaf-pt-def*)
  from *True while-share-red-exp*
  have *while-low-Null*:
    repa no = *hd* [*sn*←*prx*. *repNodes-eq sn no low high repa*] ∧
      repa (*repa no*) = *repa no* ∧
      (∀ *no1*∈*set prx*. ((*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no*
    ∧ (*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*) = (*repa no* = *repa no1*))
    by *auto*
  from *no-in-take-n*
  have [*sn*←*prx*. *repNodes-eq sn no low high repa*] ≠ []
    apply −
    apply (*rule filter-not-empty*)
    apply (*auto simp add*: *repNodes-eq-def*)
    done
    then have *hd-term-n-Sucn*: *hd* [*sn*←*prx*. *repNodes-eq sn no low high*
repa] =
      hd [*sn*←(*prx*@[*node*]) . *repNodes-eq sn no low high repa*]
    apply −
    apply (*rule hd-filter-app* [*symmetric*])
    apply *auto*
    done
  have *all-nodes-in-nl-Leafs*:
    ∀ *x* ∈ *set* (*prx*@*node*#*sfx*). *isLeaf-pt x low high*
  proof (*intro ballI*)
    fix *x*
    assume *x-in-nodeslist*: *x* ∈ *set* (*prx*@*node*#*sfx*)
    from *isLeaf-no isLeaf-var-no* **have** *var no* ≤ *1*
      by *simp*
    with *all-nodes-same-var* [*rule-format*, *OF x-in-nodeslist no-in-nl*]
    have *var x* ≤ *1*
      by *simp*
    with *nodes-balanced-ordered* [*rule-format*, *OF x-in-nodeslist*]
    show *isLeaf-pt x low high*
      by (*auto simp add*: *isLeaf-pt-def*)
  qed

from *no-in-take-Sucn* **have**
filter-Sucn-no-notempty:
[*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*] ≠ []
apply −
apply (*rule filter-not-empty*)
apply (*auto simp add*: *repNodes-eq-def*)
done
then have *hd-term-in-take-Sucn*:
hd [*sn*←(*prx*@[*node*]) . *repNodes-eq sn no low high repb*]
∈ *set* (*prx*@[*node*])
by (*rule hd-filter-in-list*)
then have *hd-term-in-nl*:
hd [*sn*←(*prx*@[*node*]) . *repNodes-eq sn no low high repb*]
∈ *set* (*prx*@*node*#*sfx*)
by *auto*
with *all-nodes-in-nl-Leafs*
have *hd-term-Leaf*: *isLeaf-pt* (*hd* [*sn*← (*prx*@[*node*]).
repNodes-eq sn no low high repb]) *low high*
by *auto*
from *while-low-Null* **have** *repa* (*repa no*) = *repa no*
by *auto*
with *no-notin-nl repa-repb-nc*
have *repa-repb-no-repb*: *repa* (*repb no*) = *repb no*
by *auto*
have *repb-repb-no*: *repb* (*repb no*) = *repb no*
proof (*cases repb no* = *node*)
case *False*
with *repa-repb-nc repa-repb-no-repb* **show** *?thesis*
by *auto*
next
assume *repb-no-nln*: *repb no* = *node*
with *hd-term-Leaf isLeaf-no all-nodes-in-nl-Leafs*
have *nested-hd-repa-repb*:
hd [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]) . *repNodes-eq sn no low high repb*])
low high repa] =
hd [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repb]
by (*simp add*: *isLeaf-pt-def repNodes-eq-def null-comp-def*)
from *hd-term-in-take-Sucn*
have [*sn*←(*prx*@[*node*]). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repb] ≠ []
apply −
apply (*rule filter-not-empty*)
apply (*auto simp add*: *repNodes-eq-def*)
done
then have *hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn*

(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repb] =
hd [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repb]
apply −
apply (*rule hd-filter-app* [*symmetric*])
apply *auto*
done
then have *hd-term-nodeslist-Sucn*:
hd [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repb] =
hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repb]
by *simp*
from *no-in-take-Sucn filter-Sucn-no-notempty*
have *filter-filter*: *hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repb] =
hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
apply −
apply (*rule filter-hd-P-rep-indep*)
apply (*auto simp add*: *repNodes-eq-def*)
done
from *repb-no-def repb-no-nln repb-node*
have *repb* (*repb no*) = *hd* [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repa]
by *simp*
with *nested-hd-repa-repb*
have *repb* (*repb no*) = *hd* [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repb]
by *simp*
with *hd-term-nodeslist-Sucn*
have *repb* (*repb no*) = *hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn*
(*hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*])
low high repb]
by *simp*
with *filter-filter*
have *repb* (*repb no*) = *hd* [*sn*←(*prx*@[*node*]).
repNodes-eq sn no low high repb]
by *simp*
with *repb-no-def* **show** *?thesis*
by *simp*
qed
have *two-nodes-repb*: (∀ *no1*∈*set* (*prx*@[*node*]).

((*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*
∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*) = (*repb no* = *repb no1*))
proof (*intro ballI*)
fix *no1*
assume *no1-in-take-Sucn*: *no1* ∈ *set* (*prx*@[*node*])
then have *no1* ∈ *set* (*prx*@*node*#*sfx*) **by** *auto*
with *all-nodes-in-nl-Leafs*
have *isLeaf-no1*: *isLeaf-pt no1 low high*
by *auto*
with *isLeaf-no*
have *repbchildren-eq-no-no1*: (*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*
∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*
by (*simp add*: *null-comp-def isLeaf-pt-def*)
from *isLeaf-no1 isLeaf-no*
have *repachildren-eq-no-no1*: (*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no*
∧ (*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*
by (*simp add*: *null-comp-def isLeaf-pt-def*)
from *while-low-Null*
have *while-low-same-rep*: (∀ *no1*∈*set prx*.
((*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no*
∧ (*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*) = (*repa no* = *repa no1*))
by *auto*
show ((*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no* ∧
(*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*) = (*repb no* = *repb no1*)
proof (*cases no1* = *node*)
case *False*
with *no1-in-take-Sucn* **have** *no1* ∈ *set prx*
by *auto*
with *while-low-same-rep repachildren-eq-no-no1*
have *repa no* = *repa no1*
by *auto*
with *repa-repb-nc no-notin-nl False repbchildren-eq-no-no1*
show *?thesis*
by *auto*
next
assume *no1-nln*: *no1* = *node*
hence *no1-in-take-Sucn*: *no1* ∈ *set* (*prx*@[*node*])
by *auto*
hence *no1-in-nl*: *no1* ∈ *set* (*prx*@*node*#*sfx*)
by *auto*
from *nodes-balanced-ordered* [*rule-format*, *OF this*] **have**
balanced-no1: (*low no1* = *Null*) = (*high no1* = *Null*)
by *auto*
with *no1-in-take-Sucn repb-no-share-def isLeaf-no1*
have *repb-no1*: *repb no1* = *hd* [*sn*←(*prx*@[*node*]).
repNodes-eq sn no1 low high repb]
by (*auto simp add*: *isLeaf-pt-def*)
from *balanced-no1 isLeaf-no1 isLeaf-no balanced-no*
have *repbchildren-eq-no1-no*: (*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*

∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*
by (*simp add*: *null-comp-def isLeaf-pt-def*)
have ∀ *x* ∈ *set* (*prx*@[*node*]). *repNodes-eq x no low high repb*
= *repNodes-eq x no1 low high repb*
proof (*intro ballI*)
fix *x*
assume *x-in-take-Sucn*: *x* ∈ *set* (*prx*@[*node*])
with *repbchildren-eq-no1-no* **show** *repNodes-eq x no low high repb*
= *repNodes-eq x no1 low high repb*
by (*simp add*: *repNodes-eq-def*)
qed
then have [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
= [*sn*←(*prx*@[*node*]). *repNodes-eq sn no1 low high repb*]
by (*rule P-eq-list-filter*)
with *repb-no-def repb-no1* **have** *repb-no-no1*: *repb no* = *repb no1*
by *simp*
with *repbchildren-eq-no1-no* **show** *?thesis*
by *simp*
qed
qed
with *repb-repb-no repb-no-share-def no-in-take-Sucn share-case-repb*
show *?thesis*
using [[*simp-depth-limit=4*]]
by *auto*
next
assume *lno-nNull*: *low no* ≠ *Null*
with *share-case-repb*
have *repbchildren-neq-no*: (*repb* ∝ *low*) *no* ≠ (*repb* ∝ *high*) *no*
by *auto*
from *balanced-no lno-nNull*
have *hno-nNull*: *high no* ≠ *Null*
by *simp*
with *repbchildren-neq-no lno-nNull repa-repb-nc*
lno-notin-nl hno-notin-nl nodes-notin-nl-neq-nln
have *repachildren-neq-no*: (*repa* ∝ *low*) *no* ≠ (*repa* ∝ *high*) *no*
using [[*simp-depth-limit=2*]]
by (*auto simp add*: *null-comp-def*)
with *while-share-red-exp*
have *repa-while-inv*: *repa* (*repa no*) = *repa no*
∧ (∀ *no1*∈*set prx*. ((*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no*
∧ (*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*) = (*repa no* = *repa no1*))
by *auto*
from *lno-nNull hno-nNull*
have *no-nLeaf*: ¬ *isLeaf-pt no low high*
by (*simp add*: *isLeaf-pt-def*)
have *all-nodes-in-nl-nLeafs*:
∀ *x* ∈ *set* (*prx*@*node*#*sfx*). ¬ *isLeaf-pt x low high*
proof (*intro ballI*)
fix *x*

assume *x-in-nodeslist*: $x \in set$ (*prx*@*node*#*sfx*)
from *no-nLeaf isLeaf-var-no* **have** $1 < var\ no$
by *simp*
with *all-nodes-same-var* [*rule-format*, *OF x-in-nodeslist no-in-nl*]
have $1 < var\ x$
by *simp*
with *nodes-balanced-ordered* [*rule-format*, *OF x-in-nodeslist*]
show ¬ *isLeaf-pt x low high*
using [[*simp-depth-limit* = *2*]]
by (*auto simp add*: *isLeaf-pt-def*)
qed
have *repb-repb-no*: *repb* (*repb no*) = *repb no*
proof −
from *repa-while-inv no-notin-nl repa-repb-nc*
have *repa* (*repb no*) = *repb no*
by *simp*
from *hd-filter-Sucn-in-Sucn repb-no-def*
have *repb-no-in-take-Sucn*: *repb no* ∈ *set* (*prx*@[*node*])
by *simp*
hence *repb-no-in-nl*: *repb no* ∈ *set* (*prx*@*node*#*sfx*)
by *auto*
from *all-nodes-in-nl-nLeafs repb-no-in-nl*
have *repb-no-nLeaf*: ¬ *isLeaf-pt* (*repb no*) *low high*
by *auto*
from *nodes-balanced-ordered* [*rule-format*, *OF repb-no-in-nl*]
have (*low* (*repb no*) = *Null*) = (*high* (*repb no*) = *Null*)
∧ *low* (*repb no*) ∉ *set* (*prx*@*node*#*sfx*) ∧
high (*repb no*) ∉ *set* (*prx*@*node*#*sfx*)
by *auto*
from *filter-take-Sucn-not-empty*
have *repNodes-eq* (*hd* [*sn*←(*prx*@[*node*]).
repNodes-eq sn no low high repb]) *no low high repb*
by (*rule hd-filter-prop*)
with *repb-no-def* **have** *repNodes-eq* (*repb no*) *no low high repb*
by *simp*
then have (*repb* ∝ *low*) (*repb no*) = (*repb* ∝ *low*) *no*
∧ (*repb* ∝ *high*) (*repb no*) = (*repb* ∝ *high*) *no*
by (*simp add*: *repNodes-eq-def*)
with *repbchildren-neq-no* **have** (*repb* ∝ *low*) (*repb no*)
≠ (*repb* ∝ *high*) (*repb no*)
by *simp*
with *repb-no-in-take-Sucn repb-no-share-def*
have *repb-repb-no-double-hd*:
repb (*repb no*) = *hd* [*sn*←(*prx*@[*node*]).
repNodes-eq sn (*repb no*) *low high repb*]
by *auto*
from *filter-take-Sucn-not-empty*
have *hd* [*sn*←(*prx*@[*node*]).
repNodes-eq sn (*repb no*) *low high repb*] = *repb no*

apply (*simp only*: *repb-no-def*)
apply (*rule filter-hd-P-rep-indep*)
apply (*auto simp add*: *repNodes-eq-def*)
done
with *repb-repb-no-double-hd* **show** *?thesis*
by *simp*
qed
have ($\forall$ *no1* $\in$ *set* (*prx*@[*node*]).
(($repb \propto high$) *no1* = ($repb \propto high$) *no* $\wedge$
($repb \propto low$) *no1* = ($repb \propto low$) *no*) = (*repb no* = *repb no1*))
proof (*intro ballI*)
fix *no1*
assume *no1-in-take-Sucn*: *no1* $\in$ *set* (*prx*@[*node*])
hence *no1-in-nl*: *no1* $\in$ *set* (*prx*@*node*#*sfx*)
by *auto*
from *all-nodes-in-nl-nLeafs no1-in-nl*
have *no1-nLeaf*: $\neg$ *isLeaf-pt no1 low high*
by *auto*
from *nodes-balanced-ordered* [*rule-format*, *OF no1-in-nl*]
have *no1-props*: (*low no1* = *Null*) = (*high no1* = *Null*)
$\wedge$ *low no1* $\notin$ *set* (*prx*@*node*#*sfx*) $\wedge$ *high no1* $\notin$ *set* (*prx*@*node*#*sfx*)
by *auto*
show (($repb \propto high$) *no1* = ($repb \propto high$) *no*
$\wedge$ ($repb \propto low$) *no1* = ($repb \propto low$) *no*) = (*repb no* = *repb no1*)
proof (*cases no1* = *node*)
case *False*
note *no1-neq-nln*=*this*
with *no1-in-take-Sucn*
have *no1-in-take-n*: *no1* $\in$ *set prx*
by *auto*
with *repa-while-inv* **have** (($repa \propto high$) *no1* = ($repa \propto high$) *no*
$\wedge$ ($repa \propto low$) *no1* = ($repa \propto low$) *no*) = (*repa no* = *repa no1*)
by *fastforce*
with *no1-props no1-nLeaf no-nLeaf balanced-no lno-notin-nl*
hno-notin-nl nodes-notin-nl-neq-nln no-notin-nl
no1-neq-nln repa-repb-nc
show *?thesis*
using [[*simp-depth-limit*=*1*]]
by (*auto simp add*: *null-comp-def isLeaf-pt-def*)
next
assume *no1-nln*: *no1* = *node*
show *?thesis*
proof
assume *repbchildren-eq-no1-no*:
($repb \propto high$) *no1* = ($repb \propto high$) *no*
$\wedge$ ($repb \propto low$) *no1* = ($repb \propto low$) *no*
with *repbchildren-neq-no*
have ($repb \propto high$) *no1* $\neq$ ($repb \propto low$) *no1*
by *auto*

with *repb-no-share-def no1-in-take-Sucn*
have *repb-no1-def*: *repb no1* = *hd* [*sn*←(*prx*@[*node*]).
repNodes-eq sn no1 low high repb]
by *auto*
have *filter-no1-eq-filter-no*: [*sn*←(*prx*@[*node*]).
repNodes-eq sn no1 low high repb] =
[*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
proof −
have ∀ *x* ∈ *set* (*prx*@[*node*]).
repNodes-eq x no1 low high repb =
repNodes-eq x no low high repb
proof (*intro ballI*)
fix *x*
assume *x-in-take-Sucn*: *x* ∈ *set* (*prx*@[*node*])
with *repbchildren-eq-no1-no*
show *repNodes-eq x no1 low high repb* =
repNodes-eq x no low high repb
by (*simp add*: *repNodes-eq-def*)
qed
then show *?thesis*
by (*rule P-eq-list-filter*)
qed
with *repb-no1-def repb-no-def* **show** *repb no* = *repb no1*
by *simp*
next
assume *repb-no-no1-eq*: *repb no* = *repb no1*
from *no1-nln repb-node repb-no-def* **have** *repb-no1-def*:
repb no1 =
hd [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn node low high repa*]
by *auto*
with *no1-nln repb-no-def repb-no-no1-eq*
have *repb-Sucn-repa-nl-hd*: *hd* [*sn*←(*prx*@[*node*]).
repNodes-eq sn no low high repb] =
hd [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no1 low high repa*]
by *simp*
from *filter-take-Sucn-not-empty*
have *hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
= *hd* [*sn*←(*prx*@*node*#*sfx*) . *repNodes-eq sn no low high repb*]
apply −
apply (*rule hd-filter-app* [*symmetric*])
apply *auto*
done
then have *hd-Sucn-hd-whole-list*:
hd [*sn*←(*prx*@[*node*]) .
repNodes-eq sn no low high repb] =
hd [*sn*← (*prx*@*node*#*sfx*). *repNodes-eq sn no low high repb*]
by *simp*
have *hd-nl-repb-repa*:
[*sn*← (*prx*@*node*#*sfx*). *repNodes-eq sn no low high repb*]

= [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no low high repa*]
proof −
have ∀ *x* ∈ *set* (*prx*@*node*#*sfx*).
repNodes-eq x no low high repb =
repNodes-eq x no low high repa
proof (*intro ballI*)
fix *x*
assume *x-in-nl*: *x* ∈ *set* (*prx*@*node*#*sfx*)
from *all-nodes-in-nl-nLeafs x-in-nl*
have *x-nLeaf*: ¬ *isLeaf-pt x low high*
by *auto*
from *nodes-balanced-ordered* [*rule-format*, *OF x-in-nl*]
have *x-props*: (*low x* = *Null*) = (*high x* = *Null*) ∧
low x ∉ *set* (*prx*@*node*#*sfx*) ∧ *high x* ∉ *set* (*prx*@*node*#*sfx*)
by *auto*
with *x-nLeaf lno-nNull hno-nNull lno-notin-nl hno-notin-nl*
nodes-notin-nl-neq-nln repa-repb-nc
show *repNodes-eq x no low high repb* =
repNodes-eq x no low high repa
using [[*simp-depth-limit=1*]]
by (*simp add*: *repNodes-eq-def isLeaf-pt-def null-comp-def*)
qed
then show *?thesis*
by (*rule P-eq-list-filter*)
qed
with *repb-Sucn-repa-nl-hd hd-Sucn-hd-whole-list*
have *filter-nl-no-no1*:
hd [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no low high repa*]
= *hd* [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no1 low high repa*]
by *simp*
from *no-in-nl* **have** *filter-no-not-empty*:
[*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no low high repa*] ≠ []
apply −
apply (*rule filter-not-empty*)
apply (*auto simp add*: *repNodes-eq-def*)
done
from *no1-in-nl* **have** *filter-no1-not-empty*:
[*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no1 low high repa*] ≠ []
apply −
apply (*rule filter-not-empty*)
apply (*auto simp add*: *repNodes-eq-def*)
done
from *repb-no-def hd-Sucn-hd-whole-list hd-nl-repb-repa*
have *repb no* =
hd [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no low high repa*]
by *simp*
with *hd-filter-prop* [*OF filter-no-not-empty*]
have *repNodes-no-repa*: *repNodes-eq* (*repb no*) *no low high repa*
by *auto*

from *repb-no1-def no1-nln*
have
repb no1 = *hd* [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no1*
low high repa]
by *simp*
with *hd-filter-prop* [*OF filter-no1-not-empty*]
have *repNodes-eq* (*repb no1*) *no1 low high repa*
by *auto*
with *filter-nl-no-no1 repNodes-no-repa repb-no-no1-eq*
have (*repa* ∝ *high*) *no1* =
(*repa* ∝ *high*) *no* ∧ (*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*
by (*simp add*: *repNodes-eq-def*)
with *hno-nNull no1-props no1-nLeaf lno-nNull lno-notin-nl*
hno-notin-nl nodes-notin-nl-neq-nln repa-repb-nc
show (*repb* ∝ *high*) *no1* =
(*repb* ∝ *high*) *no* ∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*
using [[*simp-depth-limit=1*]]
by (*auto simp add*: *isLeaf-pt-def null-comp-def*)
qed
qed
qed
with *repb-repb-no repb-no-share-def share-case-repb no-in-take-Sucn*
show *?thesis*
using [[*simp-depth-limit=1*]]
by *auto*
qed
qed
with *repb-no-nNull* **show** *?thesis*
by *simp*
next
assume *no-nln*: *no* = *node*
with *repb-node* **have** *repb-no-def*:
repb no = *hd* [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no low high repa*]
by *simp*
from *no-nln* **have** *no* ∈ *set* (*prx*@*node*#*sfx*)
by *auto*
then have *filter-nl-repa-not-empty*:
[*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no low high repa*] ≠ []
apply −
apply (*rule filter-not-empty*)
apply (*auto simp add*: *repNodes-eq-def*)
done
then have *hd-filter-nl-in-nl*:
hd [*sn*←(*prx*@*node*#*sfx*). *repNodes-eq sn no low high repa*] ∈ *set* (*prx*@*node*#*sfx*)
by (*rule hd-filter-in-list*)
with *repb-no-def*
have *repb-no-in-nodeslist*: *repb no* ∈ *set* (*prx*@*node*#*sfx*)
by *simp*
from *nodes-balanced-ordered* [*rule-format*,*OF this*]

have *repb-no-nNull*: *repb no* ≠ *Null*
by *auto*
from *share-cond no-nln* **have** *share-cond-or*:
isLeaf-pt no low high ∨ *repa* (*low no*) ≠ *repa* (*high no*)
by *auto*
have *share-reduce-if*: (*if* (*repb* ∝ *low*) *no* = (*repb* ∝ *high*) *no* ∧ *low no* ≠ *Null*
then repb no = (*repb* ∝ *low*) *no*
else repb no = *hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*] ∧
repb (*repb no*) = *repb no*
∧ (∀ *no1*∈*set* (*prx*@[*node*]). ((*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*
∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*) = (*repb no* = *repb no1*)))
proof (*cases isLeaf-pt no low high*)
case *True*
note *isLeaf-no*=*this*
then have *lno-Null*: *low no* = *Null* **by** (*simp add*: *isLeaf-pt-def*)
from *isLeaf-no no-in-take-Sucn repb-no-share-def*
have *repb-no-repb-def*: *repb no*
= *hd* [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
by (*auto simp add*: *isLeaf-pt-def*)
from *isLeaf-no nodes-balanced-ordered* [*rule-format*, *OF no-in-nl*]
have *var-no*: *var no* ≤ *1*
by *auto*
have *all-nodes-nl-var-l-1*: ∀ *x* ∈ *set* (*prx*@*node*#*sfx*). *var x* ≤ *1*
proof (*intro ballI*)
fix *x*
assume *x-in-nl*: *x* ∈ *set* (*prx*@*node*#*sfx*)
from *all-nodes-same-var* [*rule-format*, *OF x-in-nl no-in-nl*] *var-no*
show *var x* ≤ *1*
by *auto*
qed
have *all-nodes-nl-Leafs*: ∀ *x* ∈ *set* (*prx*@*node*#*sfx*). *isLeaf-pt x low high*
proof (*intro ballI*)
fix *x*
assume *x-in-nl*: *x* ∈ *set* (*prx*@*node*#*sfx*)
with *all-nodes-nl-var-l-1* **have** *var x* ≤ *1*
by *auto*
with *nodes-balanced-ordered* [*rule-format*, *OF x-in-nl*]
show *isLeaf-pt x low high*
by *auto*
qed
have *repb-repb-no*: *repb* (*repb no*) = *repb no*
proof −
from *repb-no-share-def no-in-take-Sucn lno-Null*
have *repb-no-def*: *repb no* =
hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
by *auto*
with *hd-filter-Sucn-in-Sucn*

have *repb-no-in-take-Sucn*: *repb no* ∈ *set* (*prx*@[*node*])
  by *simp*
hence *repb-no-in-nl*: *repb no* ∈ *set* (*prx*@[*node*])
  by *auto*
with *all-nodes-nl-Leafs*
have *repb-no-Leaf*: *isLeaf-pt* (*repb no*) *low high*
  by *auto*
with *repb-no-in-take-Sucn repb-no-share-def*
have *repb-repb-no-def*: *repb* (*repb no*) =
  hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn* (*repb no*) *low high repb*]
  by (*auto simp add*: *isLeaf-pt-def*)
from *filter-take-Sucn-not-empty*
show *?thesis*
  apply (*simp only*: *repb-repb-no-def*)
  apply (*simp only*: *repb-no-def*)
  apply (*rule filter-hd-P-rep-indep*)
  apply (*auto simp add*: *repNodes-eq-def*)
  done
qed
have *two-nodes-repb*: (∀ *no1*∈*set* (*prx*@[*node*]).
  ((*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no* ∧
  (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*) = (*repb no* = *repb no1*))
proof (*intro ballI*)
  fix *no1*
  assume *no1-in-take-Sucn*: *no1* ∈ *set* (*prx*@[*node*])
  from *no1-in-take-Sucn*
  have *no1* ∈ *set* (*prx*@*node*#*sfx*)
    by *auto*
  with *all-nodes-nl-Leafs*
  have *isLeaf-no1*: *isLeaf-pt no1 low high*
    by *auto*
  with *repb-no-share-def no1-in-take-Sucn*
  have *repb-no1-def*: *repb no1* =
    hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn no1 low high repb*]
    by (*auto simp add*: *isLeaf-pt-def*)
  show ((*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*
    ∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*) = (*repb no* = *repb no1*)
  proof
    assume *repbchildren-eq-no1-no*: (*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*
      ∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*
    have [*sn*←(*prx*@[*node*]). *repNodes-eq sn no1 low high repb*]
      = [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
    proof −
      have ∀ *x* ∈ *set* (*prx*@[*node*]).
        repNodes-eq x no1 low high repb = *repNodes-eq x no low high repb*
      proof (*intro ballI*)
        fix *x*
        assume *x-in-take-Sucn*: *x* ∈ *set* (*prx*@[*node*])
        with *repbchildren-eq-no1-no*

show *repNodes-eq x no1 low high repb* = *repNodes-eq x no low high repb*
by (*simp add*: *repNodes-eq-def*)
qed
then show *?thesis*
by (*rule P-eq-list-filter*)
qed
with *repb-no1-def repb-no-repb-def*
show *repb no* = *repb no1*
by *simp*
next
assume *repb-no-no1*: *repb no* = *repb no1*
with *isLeaf-no isLeaf-no1*
show (*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*
∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*
by (*simp add*: *null-comp-def isLeaf-pt-def*)
qed
qed
with *repb-repb-no lno-Null no-in-take-Sucn repb-no-share-def* **show** *?thesis*
by *auto*
next
assume *no-nLeaf*: ¬ *isLeaf-pt no low high*
with *balanced-no* **obtain**
lno-nNull: *low no* ≠ *Null* **and**
hno-nNull: *high no* ≠ *Null*
by (*simp add*: *isLeaf-pt-def*)
from *no-nLeaf nodes-balanced-ordered* [*rule-format*, *OF no-in-nl*]
have *var-no*: *1* < *var no*
by *auto*
have *all-nodes-nl-var-l-1*: ∀ *x* ∈ *set* (*prx@node#sfx*). *1* < *var x*
proof (*intro ballI*)
fix *x*
assume *x-in-nl*: *x* ∈ *set* (*prx@node#sfx*)
with *all-nodes-same-var* [*rule-format*, *OF x-in-nl no-in-nl*] *var-no*
show *1* < *var x*
by *simp*
qed
have *all-nodes-nl-nLeafs*: ∀ *x* ∈ *set* (*prx@node#sfx*). ¬ *isLeaf-pt x low high*
proof (*intro ballI*)
fix *x*
assume *x-in-nl*: *x* ∈ *set* (*prx@node#sfx*)
with *all-nodes-nl-var-l-1* **have** *1* < *var x*
by *auto*
with *nodes-balanced-ordered* [*rule-format*, *OF x-in-nl*] **show** ¬ *isLeaf-pt x low high*
by *auto*
qed
from *no-nLeaf share-cond-or*

have *repachildren-neq-no*: *repa* (*low no*) ≠ *repa* (*high no*)
  by *auto*
with *lno-nNull hno-nNull*
have (*repa* ∝ *low*) *no* ≠ (*repa* ∝ *high*) *no*
  by (*simp add*: *null-comp-def*)
with *repa-repb-nc lno-notin-nl hno-notin-nl*
  nodes-notin-nl-neq-nln lno-nNull hno-nNull
have *repbchildren-neq-no*: (*repb* ∝ *low*) *no* ≠ (*repb* ∝ *high*) *no*
  using [[*simp-depth-limit=1*]]
  by (*auto simp add*: *null-comp-def*)
have *repb-repb-no*: *repb* (*repb no*) = *repb no*
proof −
  from *repb-no-share-def no-in-take-Sucn repbchildren-neq-no*
  have *repb-no-def*: *repb no* =
    hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
    by *auto*
  from *filter-take-Sucn-not-empty*
  have *repNodes-eq* (*repb no*) *no low high repb*
    apply (*simp only*: *repb-no-def*)
    apply (*rule hd-filter-prop*)
    apply *simp*
    done
  with *repbchildren-neq-no*
   have *repbchildren-neq-repb-no*: (*repb* ∝ *low*) (*repb no*) ≠ (*repb* ∝ *high*)
(*repb no*)
    by (*simp add*: *repNodes-eq-def*)
  from *filter-take-Sucn-not-empty*
  have *repb no* ∈ *set* (*prx*@[*node*])
    apply (*simp only*: *repb-no-def*)
    apply (*rule hd-filter-in-list*)
    apply *simp*
    done
  with *repbchildren-neq-repb-no repb-no-share-def*
  have *repb-repb-no-def*: *repb* (*repb no*) =
    hd [*sn*←(*prx*@[*node*]) . *repNodes-eq sn* (*repb no*) *low high repb*]
    by *auto*
  from *filter-take-Sucn-not-empty* **show** *?thesis*
    apply (*simp only*: *repb-repb-no-def*)
    apply (*simp only*: *repb-no-def*)
    apply (*rule filter-hd-P-rep-indep*)
    apply (*auto simp add*: *repNodes-eq-def*)
    done
qed
have *two-nodes-repb*: (∀ *no1*∈*set* (*prx*@[*node*]).
  ((*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no* ∧
  (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*) = (*repb no* = *repb no1*))
  (**is** (∀ *no1*∈*set* (*prx*@[*node*]). *?P no1*))
proof (*intro ballI*)
  fix *no1*

assume *no1-in-take-Sucn*: *no1* ∈ *set* (*prx*@[*node*])
hence *no1-in-nodeslist*: *no1* ∈ *set* (*prx*@*node*#*sfx*)
  by *auto*
with *all-nodes-nl-nLeafs*
have *no1-nLeaf*: ¬ *isLeaf-pt no1 low high*
  by *auto*
show *?P no1*
proof
 assume *repbchildren-eq-no1-no*: (*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*
    ∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*
  with *repbchildren-neq-no* **have** (*repb* ∝ *high*) *no1* ≠ (*repb* ∝ *low*) *no1*
    by *auto*
  with *no1-in-take-Sucn repb-no-share-def* **have** *repb-no1-def*: *repb no1*
=
    hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn no1 low high repb*]
    by *auto*
  from *repb-no-share-def no-in-take-Sucn repbchildren-neq-no*
  have *repb-no-def*: *repb no* =
    hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
    by *auto*
  have [*sn*←(*prx*@[*node*]). *repNodes-eq sn no1 low high repb*] =
    [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
  proof −
    have ∀ *x* ∈ *set* (*prx*@[*node*]).
      repNodes-eq x no1 low high repb = *repNodes-eq x no low high repb*
    proof (*intro ballI*)
      fix *x*
      assume *x-in-take-Sucn*: *x* ∈ *set* (*prx*@[*node*])
      with *repbchildren-eq-no1-no*
      show *repNodes-eq x no1 low high repb* = *repNodes-eq x no low high repb*
        by (*simp add*: *repNodes-eq-def*)
    qed
    then show *?thesis*
      by (*rule P-eq-list-filter*)
  qed
  with *repb-no-def repb-no1-def* **show** *repb no* = *repb no1*
    by *simp*
next
  assume *repb-no-no1*: *repb no* = *repb no1*
  from *repb-no-share-def no-in-take-Sucn repbchildren-neq-no*
  have *repb-no-def*: *repb no* =
    hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn no low high repb*]
    by *auto*
  from *filter-take-Sucn-not-empty*
  have *repb no* ∈ *set* (*prx*@[*node*])
    apply (*simp only*: *repb-no-def*)
    apply (*rule hd-filter-in-list*)
    apply *simp*

done
then have *repb-no-in-nl*: *repb no* ∈ *set* (*prx*@*node*#*sfx*)
by *auto*
from *filter-take-Sucn-not-empty*
have *repNodes-repb-no*: *repNodes-eq* (*repb no*) *no low high repb*
apply (*simp only*: *repb-no-def*)
apply (*rule hd-filter-prop*)
apply *simp*
done
show (*repb* ∝ *high*) *no1* = (*repb* ∝ *high*) *no*
∧ (*repb* ∝ *low*) *no1* = (*repb* ∝ *low*) *no*
proof (*cases* (*repb* ∝ *low*) *no1* = (*repb* ∝ *high*) *no1*)
case *True*
note *red-cond*=*this*
from *no1-in-nodeslist all-nodes-nl-nLeafs*
have *no1-nLeaf*: ¬ *isLeaf-pt no1 low high*
by *auto*
from *nodes-balanced-ordered* [*rule-format*, *OF no1-in-nodeslist*]
have *no1-props*: (*low no1* ∉ *set* (*prx*@*node*#*sfx*))
∧ (*high no1* ∉ *set* (*prx*@*node*#*sfx*)) ∧(*low no1* = *Null*) = (*high*
no1 = *Null*)
∧ ((*rep* ∝ *low*) *no1* ∉ *set* (*prx*@*node*#*sfx*))
by *auto*
with *red-cond no1-nLeaf no1-in-take-Sucn repb-no-red-def*
have *repb-no1-def*: *repb no1* = (*repb* ∝ *low*) *no1*
by (*auto simp add*: *isLeaf-pt-def*)
with *no1-nLeaf no1-props* **have** *repb no1* = *repb* (*low no1*)
by (*simp add*: *null-comp-def isLeaf-pt-def*)
from *no1-props no1-nLeaf* **have** *rep* (*low no1*) ∉ *set* (*prx*@*node*#*sfx*)
by (*auto simp add*: *isLeaf-pt-def null-comp-def*)
with *rep-repb-nc no1-props*
have *repb* (*low no1*) ∉ *set* (*prx*@*node*#*sfx*)
by *auto*
with *repb-no1-def repb-no-no1 no1-props no1-nLeaf*
have *repb no* ∉ *set* (*prx*@*node*#*sfx*)
by (*simp add*: *isLeaf-pt-def null-comp-def*)
with *repb-no-in-nl* **show** *?thesis*
by *simp*
next
assume (*repb* ∝ *low*) *no1* ≠ (*repb* ∝ *high*) *no1*
with *repb-no-share-def no1-in-take-Sucn*
have *repb-no1-def*: *repb no1* =
hd [*sn*←(*prx*@[*node*]). *repNodes-eq sn no1 low high repb*]
by *auto*
from *no1-in-take-Sucn*
have [*sn*←(*prx*@[*node*]). *repNodes-eq sn no1 low high repb*] ≠ []
apply −
apply (*rule filter-not-empty*)
apply (*auto simp add*: *repNodes-eq-def*)

```
          done
        then
        have repNodes-repb-no1: repNodes-eq (repb no1) no1 low high repb
          apply (simp only: repb-no1-def )
          apply (rule hd-filter-prop)
          apply simp
          done
        with repNodes-repb-no repb-no-no1
        have repNodes-eq no1 no low high repb
          by (simp add: repNodes-eq-def)
        then show ?thesis
          by (simp add: repNodes-eq-def)
      qed
    qed
  qed
  with repb-repb-no repb-no-share-def no-in-take-Sucn repbchildren-neq-no
  show ?thesis
    using [[simp-depth-limit=2]]
    by fastforce
qed
with repb-no-nNull show ?thesis
  by simp
qed
qed
with rep-repb-nc show ?thesis
  by (intro conjI)
qed
qed

end
```

9 Proof of Procedure Repoint

theory *RepointProof* **imports** *ProcedureSpecs* **begin**

hide-const (**open**) *DistinctTreeProver.set-of tree.Node tree.Tip*

lemma (**in** *Repoint-impl*) *Repoint-modifies*:
  shows $\forall \sigma.\ \Gamma \vdash \{\sigma\}$ *´p :== PROC Repoint (´p)*
    $\{t.$ *t may-only-modify-globals* σ *in* [*low*,*high*]$\}$
  apply (*hoare-rule HoarePartial.ProcRec1*)
  apply (*vcg spec=modifies*)
  done

lemma *low-high-exchange-dag*:
assumes *pt-same*: $\forall pt.\ pt \notin$ *set-of lt* $\longrightarrow$ *low pt = lowa pt* $\wedge$ *high pt = higha pt*
assumes *pt-changed*: $\forall pt \in$ *set-of lt. lowa pt* = (*rep* $\propto$ *low*) *pt* $\wedge$
    higha pt = (*rep* $\propto$ *high*) *pt*
assumes *rep-pt*: $\forall pt \in$ *set-of rt. rep pt = pt*

```
shows ⋀q. Dag q (rep ∝ low) (rep ∝ high) rt ⟹
          Dag q (rep ∝ lowa) (rep ∝ higha) rt
using rep-pt
proof (induct rt)
  case Tip thus ?case by simp
next
  case (Node lrt q' rrt)
  have Dag q (rep ∝ low) (rep ∝ high) (Node lrt q' rrt) by fact
  then obtain
    q': q = q' and
    q-notNull: q ≠ Null and
    lrt: Dag ((rep ∝ low) q) (rep ∝ low) (rep ∝ high) lrt and
    rrt: Dag ((rep ∝ high) q) (rep ∝ low) (rep ∝ high) rrt
    by auto
  have rlowa-rlow: ((rep ∝ lowa) q) = ((rep ∝ low) q)
  proof (cases q ∈ set-of lt)
    case True
    note q-in-lt=this
    with pt-changed have lowa-q: lowa q = (rep ∝ low) q
      by simp
    thus (rep ∝ lowa) q = (rep ∝ low) q
    proof (cases low q = Null)
      case True
      with lowa-q have lowa q = Null
        by (simp add: null-comp-def)
      with True show ?thesis
        by (simp add: null-comp-def)
    next
      assume lq-nNull: low q ≠ Null
      show ?thesis
      proof (cases (rep ∝ low) q = Null)
        case True
        with lowa-q have lowa q = Null
          by simp
        with True show ?thesis
          by (simp add: null-comp-def)
      next
        assume rlq-nNull: (rep ∝ low) q ≠ Null
        with lrt lowa-q have lowa q ∈ set-of lrt
          by auto
        with Node.prems Node have lowa q ∈ set-of (Node lrt q' rrt)
          by simp
        with Node.prems have rep (lowa q) = lowa q
          by auto
        with lowa-q rlq-nNull show ?thesis
          by (simp add: null-comp-def)
      qed
    qed
  next
```

```
  assume q-notin-lt: q ∉ set-of lt
  with pt-same have low q = lowa q
    by auto
  thus ?thesis
    by (simp add: null-comp-def)
qed
have rhigha-rhigh: ((rep ∝ higha) q) = ((rep ∝ high) q)
proof (cases q ∈ set-of lt)
  case True
  note q-in-lt=this
  with pt-changed have higha-q: higha q = (rep ∝ high) q
    by simp
  thus ?thesis
  proof (cases high q = Null)
    case True
    with higha-q have higha q = Null
      by (simp add: null-comp-def)
    with True show ?thesis
      by (simp add: null-comp-def)
  next
    assume hq-nNull: high q ≠ Null
    show ?thesis
    proof (cases (rep ∝ high) q = Null)
      case True
      with higha-q have higha q = Null
        by simp
      with True show ?thesis
        by (simp add: null-comp-def)
    next
      assume rhq-nNull: (rep ∝ high) q ≠ Null
      with rrt higha-q have higha q ∈ set-of rrt
        by auto
      with Node.prems Node have higha q ∈ set-of (Node lrt q' rrt)
        by simp
      with Node.prems have rep (higha q) = higha q
        by auto
      with higha-q rhq-nNull show ?thesis
        by (simp add: null-comp-def)
    qed
  qed
next
  assume q-notin-lt: q ∉ set-of lt
  with pt-same have high q = higha q
    by auto
  thus ?thesis
    by (simp add: null-comp-def)
qed
with rrt have rhigha-mixed-dag:
  Dag ((rep ∝ higha) q) (rep ∝ low) (rep ∝ high) rrt
```

by *simp*
from *lrt rlowa-rlow* **have** *rlowa-mixed-dag*:
Dag ((*rep* ∝ *lowa*) *q*) (*rep* ∝ *low*) (*rep* ∝ *high*) *lrt*
by *simp*
from *Node.prems* **have** *lrt-rep-eq*: ∀ *pt*∈*set-of lrt*. *rep pt* = *pt*
by *simp*
from *Node.prems* **have** *rrt-rep-eq*: ∀ *pt*∈*set-of rrt*. *rep pt* = *pt*
by *simp*
from *rlowa-mixed-dag lrt-rep-eq* **have** *lowa-lrt*:
Dag ((*rep* ∝ *lowa*) *q*) (*rep* ∝ *lowa*) (*rep* ∝ *higha*) *lrt*
apply −
apply (*rule Node.hyps*)
apply *auto*
done
from *rhigha-mixed-dag rrt-rep-eq* **have** *higha-rrt*:
Dag ((*rep* ∝ *higha*) *q*) (*rep* ∝ *lowa*) (*rep* ∝ *higha*) *rrt*
apply −
apply (*rule Node.hyps*)
apply *auto*
done
with *lowa-lrt q′ q-notNull*
show *Dag q* (*rep* ∝ *lowa*) (*rep* ∝ *higha*) (*Node lrt q′ rrt*)
by *simp*
qed

lemma (**in** *Repoint-impl*) *Repoint-spec′*:
shows
∀ σ. Γ⊢ {σ}
´*p* :== *PROC Repoint* (´*p*)
⦃∀ *rept*. ((*Dag* (($^\sigma$*rep* ∝ *id*) $^\sigma$*p*) ($^\sigma$*rep* ∝ $^\sigma$*low*) ($^\sigma$*rep* ∝ $^\sigma$*high*) *rept*)
∧ (∀ *no* ∈ *set-of rept*. $^\sigma$*rep no* = *no*))
⟶ *Dag* ´*p* ´*low* ´*high rept* ∧
(∀ *pt*. *pt* ∉ *set-of rept* ⟶ $^\sigma$*low pt* = ´*low pt* ∧ $^\sigma$*high pt* = ´*high pt*)⦄
apply (*hoare-rule HoarePartial.ProcRec1*)
apply *vcg*
apply (*rule conjI*)
apply *clarify*
prefer *2*
apply (*intro impI allI*)
apply (*simp add*: *null-comp-def*)
apply (*rule conjI*)
prefer *2*
apply (*clarsimp*)
apply *clarify*
proof −

fix *low high p rep lowa higha pa lowb highb pb rept*
assume *p-nNull*: $p \neq Null$
assume *rp-nNull*: $rep\ p \neq Null$
assume *rec-spec-lrept*:
$\forall rept.\ Dag\ ((rep \propto id)\ (low\ (rep\ p)))\ (rep \propto low)\ (rep \propto high)\ rept$
$\wedge\ (\forall no \in set\text{-}of\ rept.\ rep\ no = no)$
$\longrightarrow Dag\ pa\ lowa\ higha\ rept\ \wedge$
$(\forall pt.\ pt \notin set\text{-}of\ rept \longrightarrow low\ pt = lowa\ pt \wedge high\ pt = higha\ pt)$
assume *rec-spec-rrept*:
$\forall rept.\ Dag\ ((rep \propto id)\ (higha\ (rep\ p)))\ (rep \propto lowa(rep\ p := pa))\ (rep \propto higha)$
$rept$
$\wedge\ (\forall no \in set\text{-}of\ rept.\ rep\ no = no)$
$\longrightarrow Dag\ pb\ lowb\ highb\ rept\ \wedge$
$(\forall pt.\ pt \notin set\text{-}of\ rept \longrightarrow (lowa(rep\ p := pa))\ pt = lowb\ pt \wedge higha\ pt =$
$highb\ pt)$
assume *rept-dag*: $Dag\ ((rep \propto id)\ p)\ (rep \propto low)\ (rep \propto high)\ rept$
assume *rno-rept*: $\forall no \in set\text{-}of\ rept.\ rep\ no = no$
show $Dag\ (rep\ p)\ lowb\ (highb(rep\ p := pb))\ rept\ \wedge$
$(\forall pt.\ pt \notin set\text{-}of\ rept \longrightarrow low\ pt = lowb\ pt \wedge high\ pt = (highb(rep\ p := pb))$
$pt)$
proof −
from *rp-nNull rept-dag p-nNull* **obtain** *lrept rrept* **where**
rept-def: $rept = Node\ lrept\ (rep\ p)\ rrept$
by *auto*
with *rept-dag p-nNull* **have** *lrept-dag*:
$Dag\ ((rep \propto low)\ (rep\ p))\ (rep \propto low)\ (rep \propto high)\ lrept$
by *simp*
from *rept-def rept-dag p-nNull* **have** *rrept-dag*:
$Dag\ ((rep \propto high)\ (rep\ p))\ (rep \propto low)\ (rep \propto high)\ rrept$
by *simp*
from *rno-rept rept-def* **have** *rno-lrept*: $\forall\ no \in set\text{-}of\ lrept.\ rep\ no = no$
by *auto*
from *rno-rept rept-def* **have** *rno-rrept*: $\forall\ no \in set\text{-}of\ rrept.\ rep\ no = no$
by *auto*
have *repoint-post-low*:
$Dag\ pa\ lowa\ higha\ lrept\ \wedge$
$(\forall pt.\ pt \notin set\text{-}of\ lrept \longrightarrow low\ pt = lowa\ pt \wedge high\ pt = higha\ pt)$
proof −
from *lrept-dag* **have** $Dag\ ((rep \propto id)\ (low\ (rep\ p)))\ (rep \propto low)\ (rep \propto high)$
$lrept$
by (*simp add*: *id-trans*)
with *rec-spec-lrept rno-lrept* **show** *?thesis*
apply −
apply (*erule-tac x=lrept* **in** *allE*)
apply (*erule impE*)
apply *simp*
apply *assumption*
done
qed

hence *low-lowa-nc*: (∀ *pt*. *pt* ∉ *set-of lrept* ⟶ *low pt* = *lowa pt* ∧ *high pt* = *higha pt*)
by *simp*
from *lrept-dag repoint-post-low* **obtain**
pa-def: *pa* = (*rep* ∝ *low*) (*rep p*) **and**
lowa-higha-def: (∀ *no* ∈ *set-of lrept*. *lowa no* = (*rep* ∝ *low*) *no* ∧ *higha no* = (*rep* ∝ *high*) *no*)
apply −
apply (*drule Dags-eq-hp-eq*)
apply *auto*
done
from *rept-dag* **have** *rept-DAG*: *DAG rept*
by (*rule Dag-is-DAG*)
with *rept-def* **have** *rp-notin-lrept*: *rep p* ∉ *set-of lrept*
by *simp*
from *rept-DAG rept-def* **have** *rp-notin-rrept*: *rep p* ∉ *set-of rrept*
by *simp*
have *Dag* ((*rep* ∝ *id*) (*higha* (*rep p*))) (*rep* ∝ *lowa*(*rep p* := *pa*)) (*rep* ∝ *higha*) *rrept*
proof −
from *low-lowa-nc rp-notin-lrept* **have** (*rep* ∝ *high*) (*rep p*) = (*rep* ∝ *higha*) (*rep p*)
by (*auto simp add*: *null-comp-def*)
with *rrept-dag* **have** *higha-mixed-rrept*:
Dag ((*rep* ∝ *id*) (*higha* (*rep p*))) (*rep* ∝ *low*) (*rep* ∝ *high*) *rrept*
by (*simp add*: *id-trans*)
thm *low-high-exchange-dag*
with *low-lowa-nc lowa-higha-def rno-rrept* **have** *lowa-higha-rrept*:
Dag ((*rep* ∝ *id*) (*higha* (*rep p*))) (*rep* ∝ *lowa*) (*rep* ∝ *higha*) *rrept*
apply −
apply (*rule low-high-exchange-dag*)
apply *auto*
done
have *Dag* ((*rep* ∝ *id*) (*higha* (*rep p*))) (*rep* ∝ *lowa*) (*rep* ∝ *higha*) *rrept* =
Dag ((*rep* ∝ *id*) (*higha* (*rep p*))) (*rep* ∝ *lowa*(*rep p* := *pa*)) (*rep* ∝ *higha*) *rrept*
proof −
have ∀ *no* ∈ *set-of rrept*. (*rep* ∝ *lowa*) *no* = (*rep* ∝ *lowa*(*rep p* := *pa*)) *no* ∧
(*rep* ∝ *higha*) *no* = (*rep* ∝ *higha*) *no*
proof
fix *no*
assume *no-in-rrept*: *no* ∈ *set-of rrept*
with *rp-notin-rrept* **have** *no* ≠ *rep p*
by *blast*
thus (*rep* ∝ *lowa*) *no* = (*rep* ∝ *lowa*(*rep p* := *pa*)) *no* ∧
(*rep* ∝ *higha*) *no* = (*rep* ∝ *higha*) *no*
by (*simp add*: *null-comp-def*)
qed

thus *?thesis*
by (*rule heaps-eq-Dag-eq*)
qed
with *lowa-higha-rrept* **show** *?thesis*
by *simp*
qed
with *rec-spec-rrept rno-rrept* **have** *repoint-rrept*: *Dag pb lowb highb rrept* ∧
(∀ *pt*. *pt* ∉ *set-of rrept* ⟶
(*lowa*(*rep p* := *pa*)) *pt* = *lowb pt* ∧ *higha pt* = *highb pt*)
apply −
apply (*erule-tac x*=*rrept* **in** *allE*)
apply (*erule impE*)
apply *simp*
apply *assumption*
done
then have *ab-nc*: (∀ *pt*. *pt* ∉ *set-of rrept* ⟶
(*lowa*(*rep p* := *pa*)) *pt* = *lowb pt* ∧ *higha pt* = *highb pt*)
by *simp*
from *repoint-rrept rrept-dag* **obtain**
pb-def: *pb* = ((*rep* ∝ *high*) (*rep p*)) **and**
lowb-highb-def: (∀ *no* ∈ *set-of rrept*. *lowb no* = (*rep* ∝ *low*) *no* ∧ *highb no* = (*rep* ∝ *high*) *no*)
apply −
apply (*drule Dags-eq-hp-eq*)
apply *auto*
done
have *rept-end-dag*: *Dag* (*rep p*) *lowb* (*highb*(*rep p* := *pb*)) *rept*
proof −
have ∀ *no* ∈ *set-of rept*. *lowb no* = (*rep* ∝ *low*) *no* ∧ (*highb*(*rep p* := *pb*)) *no* = (*rep* ∝ *high*) *no*
proof
fix *no*
assume *no-in-rept*: *no* ∈ *set-of rept*
show *lowb no* = (*rep* ∝ *low*) *no* ∧ (*highb*(*rep p* := *pb*)) *no* = (*rep* ∝ *high*) *no*
proof (*cases no* ∈ *set-of rrept*)
case *True*
with *lowb-highb-def pb-def* **show** *?thesis*
by *simp*
next
assume *no-notin-rrept*: *no* ∉ *set-of rrept*
show *?thesis*
proof (*cases no* ∈ *set-of lrept*)
case *True*
with *no-notin-rrept rp-notin-lrept ab-nc*
have *ab-nc-no*: *lowa no* = *lowb no* ∧ *higha no* = *highb no*
apply −
apply (*erule-tac x*=*no* **in** *allE*)
apply (*erule impE*)

apply *simp*
apply (*subgoal-tac no* $\neq$ *rep p*)
apply *simp*
apply *blast*
done
from *lowa-higha-def True* **have**
lowa no = (*rep* $\propto$ *low*) *no* $\wedge$ *higha no* = (*rep* $\propto$ *high*) *no*
by *auto*
with *ab-nc-no* **have** *lowb no* = (*rep* $\propto$ *low*) *no* $\wedge$ *highb no* =(*rep* $\propto$ *high*) *no*
by *simp*
with *rp-notin-lrept True* **show** *?thesis*
apply (*subgoal-tac no* $\neq$ *rep p*)
apply *simp*
apply *blast*
done
next
assume *no-notin-lrept*: *no* $\notin$ *set-of lrept*
with *no-in-rept rept-def no-notin-rrept* **have** *no-rp*: *no* = *rep p*
by *simp*
with *rp-notin-lrept low-lowa-nc* **have** *a-nc*:
low no = *lowa no* $\wedge$ *high no* = *higha no*
by *auto*
from *rp-notin-rrept no-rp ab-nc* **have**
(*lowa*(*rep p* := *pa*)) *no* = *lowb no* $\wedge$ *higha no* = *highb no*
by *auto*
with *a-nc pa-def no-rp* **have** (*rep* $\propto$ *low*) *no* = *lowb no* $\wedge$ *high no* = *highb no*
by *auto*
with *pb-def no-rp* **show** *?thesis*
by *simp*
qed
qed
qed
with *rept-dag* **have** *Dag* (*rep p*) *lowb* (*highb*(*rep p* := *pb*)) *rept* =
Dag (*rep p*) (*rep* $\propto$ *low*) (*rep* $\propto$ *high*) *rept*
apply −
thm *heaps-eq-Dag-eq*
apply (*rule heaps-eq-Dag-eq*)
apply *auto*
done
with *rept-dag p-nNull* **show** *?thesis*
by *simp*
qed
have ($\forall$ *pt*. *pt* $\notin$ *set-of rept* $\longrightarrow$ *low pt* = *lowb pt* $\wedge$ *high pt* = (*highb*(*rep p* := *pb*)) *pt*)
proof (*intro allI impI*)
fix *pt*
assume *pt-notin-rept*: *pt* $\notin$ *set-of rept*

with *rept-def* **obtain**
pt-notin-lrept: $pt \notin$ *set-of lrept* **and**
pt-notin-rrept: $pt \notin$ *set-of rrept* **and**
pt-neq-rp: $pt \neq rep\ p$
by *simp*
with *low-lowa-nc ab-nc* **show** $low\ pt = lowb\ pt \wedge high\ pt = (highb(rep\ p := pb))\ pt$
by *auto*
qed
with *rept-end-dag* **show** *?thesis*
by *simp*
qed
qed

lemma (**in** *Repoint-impl*) *Repoint-spec*:
shows
$\forall \sigma\ rept.\ \Gamma\vdash \{\!|\sigma.\ Dag\ ((\acute{}rep \propto id)\ \acute{}p)\ (\acute{}rep \propto \acute{}low)\ (\acute{}rep \propto \acute{}high)\ rept$
$\wedge (\forall\ no \in set\text{-}of\ rept.\ \acute{}rep\ no = no)\ |\!\}$
$\acute{}p :== PROC\ Repoint\ (\acute{}p)$
$\{\!|Dag\ \acute{}p\ \acute{}low\ \acute{}high\ rept \wedge$
$(\forall pt.\ pt \notin set\text{-}of\ rept \longrightarrow {}^{\sigma}low\ pt = \acute{}low\ pt \wedge {}^{\sigma}high\ pt = \acute{}high\ pt)|\!\}$
apply (*hoare-rule HoarePartial.ProcRec1*)
apply *vcg*
apply (*rule conjI*)
prefer *2*
apply (*clarsimp simp add*: *null-comp-def*)
apply *clarify*
apply (*rule conjI*)
prefer *2*
apply *clarsimp*
apply *clarify*
proof −
fix *rept low high rep p*
assume *rept-dag*: $Dag\ ((rep \propto id)\ p)\ (rep \propto low)\ (rep \propto high)\ rept$
assume *rno-rept*: $\forall no{\in}set\text{-}of\ rept.\ rep\ no = no$
assume *p-nNull*: $p \neq Null$
assume *rp-nNull*: $rep\ p \neq Null$
show $\exists lrept.$
$Dag\ ((rep \propto id)\ (low\ (rep\ p)))\ (rep \propto low)\ (rep \propto high)\ lrept \wedge$
$(\forall no{\in}set\text{-}of\ lrept.\ rep\ no = no) \wedge$
$(\forall lowa\ higha\ pa.$
$Dag\ pa\ lowa\ higha\ lrept \wedge$
$(\forall pt.\ pt \notin set\text{-}of\ lrept \longrightarrow$
$low\ pt = lowa\ pt \wedge high\ pt = higha\ pt) \longrightarrow$
$(\exists rrept.$
$Dag\ ((rep \propto id)\ (higha\ (rep\ p)))\ (rep \propto lowa(rep\ p := pa))$
$(rep \propto higha)\ rrept \wedge$
$(\forall no{\in}set\text{-}of\ rrept.\ rep\ no = no) \wedge$
$(\forall lowb\ highb\ pb.$

```
                    Dag pb lowb highb rrept ∧
                    (∀ pt. pt ∉ set-of rrept ⟶
                          (lowa(rep p := pa)) pt = lowb pt ∧
                          higha pt = highb pt) ⟶
                    Dag (rep p) lowb (highb(rep p := pb)) rept ∧
                    (∀ pt. pt ∉ set-of rept ⟶
                          low pt = lowb pt ∧
                          high pt = (highb(rep p := pb)) pt))))
proof −
  from rp-nNull rept-dag p-nNull obtain lrept rrept where
    rept-def: rept = Node lrept (rep p) rrept
    by auto
  with rept-dag p-nNull have lrept-dag:
    Dag ((rep ∝ low) (rep p)) (rep ∝ low) (rep ∝ high) lrept
    by simp
  from rept-def rept-dag p-nNull have rrept-dag:
    Dag ((rep ∝ high) (rep p)) (rep ∝ low) (rep ∝ high) rrept
    by simp
  from rno-rept rept-def have rno-lrept: ∀ no ∈ set-of lrept. rep no = no
    by auto
  from rno-rept rept-def have rno-rrept: ∀ no ∈ set-of rrept. rep no = no
    by auto
  show ?thesis
    apply (rule-tac x=lrept in exI)
    apply (rule conjI)
    apply (simp add: id-trans lrept-dag)
    apply (rule conjI)
    apply (rule rno-lrept)
    apply clarify
    subgoal premises prems for lowa higha pa
    proof −
      have lrepta: Dag pa lowa higha lrept by fact
      have low-lowa-nc:
        ∀ pt. pt ∉ set-of lrept ⟶ low pt = lowa pt ∧ high pt = higha pt by fact
      from lrept-dag lrepta obtain
        pa-def: pa = (rep ∝ low) (rep p) and
        lowa-higha-def: ∀ no ∈ set-of lrept.
        lowa no = (rep ∝ low) no ∧ higha no = (rep ∝ high) no
        apply −
        apply (drule Dags-eq-hp-eq)
        apply auto
        done
      from rept-dag have rept-DAG: DAG rept
        by (rule Dag-is-DAG)
      with rept-def have rp-notin-lrept: rep p ∉ set-of lrept
        by simp
      from rept-DAG rept-def have rp-notin-rrept: rep p ∉ set-of rrept
        by simp
      have rrepta: Dag ((rep ∝ id) (higha (rep p)))
```

(*rep* ∝ *lowa*(*rep p* := *pa*)) (*rep* ∝ *higha*) *rrept*
proof −
  from *low-lowa-nc rp-notin-lrept*
  have (*rep* ∝ *high*) (*rep p*) = (*rep* ∝ *higha*) (*rep p*)
    by (*auto simp add*: *null-comp-def*)
  with *rrept-dag* **have** *higha-mixed-rrept*:
    Dag ((*rep* ∝ *id*) (*higha* (*rep p*))) (*rep* ∝ *low*) (*rep* ∝ *high*) *rrept*
    by (*simp add*: *id-trans*)
  thm *low-high-exchange-dag*
  with *low-lowa-nc lowa-higha-def rno-rrept*
  have *lowa-higha-rrept*:
      Dag ((*rep* ∝ *id*) (*higha* (*rep p*))) (*rep* ∝ *lowa*) (*rep* ∝ *higha*) *rrept*
    apply −
    apply (*rule low-high-exchange-dag*)
    apply *auto*
    done
  have *Dag* ((*rep* ∝ *id*) (*higha* (*rep p*))) (*rep* ∝ *lowa*) (*rep* ∝ *higha*) *rrept* =
      Dag ((*rep* ∝ *id*) (*higha* (*rep p*)))
          (*rep* ∝ *lowa*(*rep p* := *pa*)) (*rep* ∝ *higha*) *rrept*
  proof −
    have ∀ *no* ∈ *set-of rrept*.
          (*rep* ∝ *lowa*) *no* = (*rep* ∝ *lowa*(*rep p* := *pa*)) *no* ∧
          (*rep* ∝ *higha*) *no* = (*rep* ∝ *higha*) *no*
    proof
      fix *no*
      assume *no-in-rrept*: *no* ∈ *set-of rrept*
      with *rp-notin-rrept* **have** *no* ≠ *rep p*
        by *blast*
      thus (*rep* ∝ *lowa*) *no* = (*rep* ∝ *lowa*(*rep p* := *pa*)) *no* ∧
        (*rep* ∝ *higha*) *no* = (*rep* ∝ *higha*) *no*
        by (*simp add*: *null-comp-def*)
    qed
    thus *?thesis*
      by (*rule heaps-eq-Dag-eq*)
  qed
  with *lowa-higha-rrept* **show** *?thesis*
    by *simp*
qed
show *?thesis*
  apply (*rule-tac x*=*rrept* **in** *exI*)
  apply (*rule conjI*)
  apply (*rule rrepta*)
  apply (*rule conjI*)
  apply (*rule rno-rrept*)
  apply *clarify*
  subgoal premises *prems* **for** *lowb highb pb*
  proof −
    have *rreptb*: *Dag pb lowb highb rrept* **by** *fact*
    have *ab-nc*: ∀ *pt*. *pt* ∉ *set-of rrept* ⟶

```
                  (lowa(rep p := pa)) pt = lowb pt ∧ higha pt = highb pt by
fact
      from rreptb rrept-dag obtain
        pb-def: pb = ((rep ∝ high) (rep p)) and
        lowb-highb-def: ∀ no ∈ set-of rrept.
                 lowb no = (rep ∝ low) no ∧ highb no = (rep ∝ high) no
        apply −
        apply (drule Dags-eq-hp-eq)
        apply auto
        done
      have rept-end-dag: Dag (rep p) lowb (highb(rep p := pb)) rept
      proof −
        have ∀ no ∈ set-of rept.
             lowb no = (rep ∝ low) no ∧ (highb(rep p := pb)) no = (rep ∝
high) no
        proof
          fix no
          assume no-in-rept: no ∈ set-of rept
          show lowb no = (rep ∝ low) no ∧
               (highb(rep p := pb)) no = (rep ∝ high) no
          proof (cases no ∈ set-of rrept)
            case True
            with lowb-highb-def pb-def show ?thesis
              by simp
          next
            assume no-notin-rrept: no ∉ set-of rrept
            show ?thesis
            proof (cases no ∈ set-of lrept)
              case True
              with no-notin-rrept rp-notin-lrept ab-nc
              have ab-nc-no: lowa no = lowb no ∧ higha no = highb no
                apply −
                apply (erule-tac x=no in allE)
                apply (erule impE)
                apply simp
                apply (subgoal-tac no ≠ rep p)
                apply simp
                apply blast
                done
              from lowa-higha-def True have
                lowa no = (rep ∝ low) no ∧ higha no = (rep ∝ high) no
                by auto
              with ab-nc-no
              have lowb no = (rep ∝ low) no ∧ highb no =(rep ∝ high) no
                by simp
              with rp-notin-lrept True show ?thesis
                apply (subgoal-tac no ≠ rep p)
                apply simp
                apply blast
```

done
next
assume *no-notin-lrept*: $no \notin$ *set-of lrept*
with *no-in-rept rept-def no-notin-rrept* **have** *no-rp*: $no = rep\ p$
by *simp*
with *rp-notin-lrept low-lowa-nc*
have *a-nc*: $low\ no = lowa\ no \wedge high\ no = higha\ no$
by *auto*
from *rp-notin-rrept no-rp ab-nc*
have $(lowa(rep\ p := pa))\ no = lowb\ no \wedge higha\ no = highb\ no$
by *auto*
with *a-nc pa-def no-rp*
have $(rep \propto low)\ no = lowb\ no \wedge high\ no = highb\ no$
by *auto*
with *pb-def no-rp* **show** *?thesis*
by *simp*
qed
qed
qed
with *rept-dag*
have $Dag\ (rep\ p)\ lowb\ (highb(rep\ p := pb))\ rept =$
$Dag\ (rep\ p)\ (rep \propto low)\ (rep \propto high)\ rept$
apply $-$
apply (*rule heaps-eq-Dag-eq*)
apply *auto*
done
with *rept-dag p-nNull* **show** *?thesis*
by *simp*
qed
have $(\forall pt.\ pt \notin$ *set-of rept* $\longrightarrow low\ pt = lowb\ pt \wedge$
$high\ pt = (highb(rep\ p := pb))\ pt)$
proof (*intro allI impI*)
fix *pt*
assume *pt-notin-rept*: $pt \notin$ *set-of rept*
with *rept-def* **obtain**
pt-notin-lrept: $pt \notin$ *set-of lrept* **and**
pt-notin-rrept: $pt \notin$ *set-of rrept* **and**
pt-neq-rp: $pt \neq rep\ p$
by *simp*
with *low-lowa-nc ab-nc*
show $low\ pt = lowb\ pt \wedge high\ pt = (highb(rep\ p := pb))\ pt$
by *auto*
qed
with *rept-end-dag* **show** *?thesis*
by *simp*
qed
done
qed
done

```
  qed
qed

lemma (in Repoint-impl) Repoint-spec-total:
shows
  ∀σ rept. Γ⊢t {|σ. Dag ((´rep ∝ id) ´p) (´rep ∝ ´low) (´rep ∝ ´high) rept
  ∧ (∀ no ∈ set-of rept. ´rep no = no) |}
  ´p :== PROC Repoint (´p)
  {|Dag ´p ´low ´high rept ∧
  (∀pt. pt ∉ set-of rept ⟶ σlow pt = ´low pt ∧ σhigh pt = ´high pt)|}

apply (hoare-rule HoareTotal.ProcRec1
          [where r=measure (λ(s,p). size
                      (dag ((srep ∝ id) sp) (srep ∝ slow) (srep ∝ shigh)))])
apply vcg
apply (rule conjI)
prefer 2
apply  (clarsimp simp add: null-comp-def)
apply clarify
apply (rule conjI)
prefer 2
apply  clarsimp
apply clarify
proof −
  fix rept low high rep p
  assume rept-dag: Dag ((rep ∝ id) p) (rep ∝ low) (rep ∝ high) rept
  assume rno-rept: ∀no∈set-of rept. rep no = no
  assume p-nNull: p ≠ Null
  assume rp-nNull:  rep p ≠ Null
  show ∃lrept.
            Dag ((rep ∝ id) (low (rep p))) (rep ∝ low) (rep ∝ high) lrept ∧
            (∀no∈set-of lrept. rep no = no) ∧
            size (dag ((rep ∝ id) (low (rep p))) (rep ∝ low) (rep ∝ high))
            < size (dag ((rep ∝ id) p) (rep ∝ low) (rep ∝ high)) ∧
            (∀lowa higha pa.
                Dag pa lowa higha lrept ∧
                (∀pt. pt ∉ set-of lrept ⟶
                      low pt = lowa pt ∧ high pt = higha pt) ⟶
                (∃rrept.
                    Dag ((rep ∝ id) (higha (rep p))) (rep ∝ lowa(rep p := pa))
                     (rep ∝ higha) rrept ∧
                    (∀no∈set-of rrept. rep no = no) ∧
                    size (dag ((rep ∝ id) (higha (rep p)))
                           (rep ∝ lowa(rep p := pa)) (rep ∝ higha))
                    < size (dag ((rep ∝ id) p) (rep ∝ low) (rep ∝ high)) ∧
                    (∀lowb highb pb.
                        Dag pb lowb highb rrept ∧
                        (∀pt. pt ∉ set-of rrept ⟶
                              (lowa(rep p := pa)) pt = lowb pt ∧
```

```
             higha pt = highb pt) ⟶
          Dag (rep p) lowb (highb(rep p := pb)) rept ∧
          (∀ pt. pt ∉ set-of rept ⟶
             low pt = lowb pt ∧
             high pt = (highb(rep p := pb)) pt))))
proof −
  from rp-nNull rept-dag p-nNull obtain lrept rrept where
    rept-def: rept = Node lrept (rep p) rrept
    by auto
  with rept-dag p-nNull have lrept-dag:
    Dag ((rep ∝ low) (rep p)) (rep ∝ low) (rep ∝ high) lrept
    by simp
  from rept-def rept-dag p-nNull have rrept-dag:
    Dag ((rep ∝ high) (rep p)) (rep ∝ low) (rep ∝ high) rrept
    by simp
  from rno-rept rept-def have rno-lrept: ∀ no ∈ set-of lrept. rep no = no
    by auto
  from rno-rept rept-def have rno-rrept: ∀ no ∈ set-of rrept. rep no = no
    by auto
  show ?thesis
    apply (rule-tac x=lrept in exI)
    apply (rule conjI)
    apply (simp add: id-trans lrept-dag)
    apply (rule conjI)
    apply (rule rno-lrept)
    apply (rule conjI)
    using rept-dag rept-def
    apply (simp only: Dag-dag)
    apply (clarsimp simp add: id-trans Dag-dag)
    apply clarify
    subgoal premises prems for lowa higha pa
    proof −
      have lrepta: Dag pa lowa higha lrept by fact
      have low-lowa-nc:
        ∀ pt. pt ∉ set-of lrept ⟶ low pt = lowa pt ∧ high pt = higha pt by fact
      from lrept-dag lrepta obtain
        pa-def: pa = (rep ∝ low) (rep p) and
        lowa-higha-def: ∀ no ∈ set-of lrept.
        lowa no = (rep ∝ low) no ∧ higha no = (rep ∝ high) no
        apply −
        apply (drule Dags-eq-hp-eq)
        apply auto
        done
      from rept-dag have rept-DAG: DAG rept
        by (rule Dag-is-DAG)
      with rept-def have rp-notin-lrept: rep p ∉ set-of lrept
        by simp
      from rept-DAG rept-def have rp-notin-rrept: rep p ∉ set-of rrept
        by simp
```

```
have rrepta: Dag ((rep ∝ id) (higha (rep p)))
              (rep ∝ lowa(rep p := pa)) (rep ∝ higha) rrept
proof −
  from low-lowa-nc rp-notin-lrept
  have (rep ∝ high) (rep p) = (rep ∝ higha) (rep p)
    by (auto simp add: null-comp-def)
  with rrept-dag have higha-mixed-rrept:
    Dag ((rep ∝ id) (higha (rep p))) (rep ∝ low) (rep ∝ high) rrept
    by (simp add: id-trans)
  thm low-high-exchange-dag
  with low-lowa-nc lowa-higha-def rno-rrept
  have lowa-higha-rrept:
      Dag ((rep ∝ id) (higha (rep p))) (rep ∝ lowa) (rep ∝ higha) rrept
    apply −
    apply (rule low-high-exchange-dag)
    apply auto
    done
 have Dag ((rep ∝ id) (higha (rep p))) (rep ∝ lowa) (rep ∝ higha) rrept =
       Dag ((rep ∝ id) (higha (rep p)))
              (rep ∝ lowa(rep p := pa)) (rep ∝ higha) rrept
  proof −
    have ∀ no ∈ set-of rrept.
             (rep ∝ lowa) no = (rep ∝ lowa(rep p := pa)) no ∧
             (rep ∝ higha) no = (rep ∝ higha) no
    proof
      fix no
      assume no-in-rrept: no ∈ set-of rrept
      with rp-notin-rrept have no ≠ rep p
        by blast
      thus (rep ∝ lowa) no = (rep ∝ lowa(rep p := pa)) no ∧
        (rep ∝ higha) no = (rep ∝ higha) no
        by (simp add: null-comp-def)
    qed
    thus ?thesis
      by (rule heaps-eq-Dag-eq)
  qed
  with lowa-higha-rrept show ?thesis
    by simp
qed
show ?thesis
  apply (rule-tac x=rrept in exI)
  apply (rule conjI)
  apply  (rule rrepta)
  apply (rule conjI)
  apply  (rule rno-rrept)
  apply (rule conjI)
  using rept-dag rept-def rrepta
  apply  (simp only: Dag-dag)
  apply  (clarsimp simp add: id-trans Dag-dag)
```

```
          apply clarify
          subgoal premises prems for lowb highb pb
          proof -
            have rreptb: Dag pb lowb highb rrept by fact
            have ab-nc: ∀ pt. pt ∉ set-of rrept ⟶
                            (lowa(rep p := pa)) pt = lowb pt ∧ higha pt = highb pt by
fact
            from rreptb rrept-dag obtain
              pb-def: pb = ((rep ∝ high) (rep p)) and
              lowb-highb-def: ∀ no ∈ set-of rrept.
                              lowb no = (rep ∝ low) no ∧ highb no = (rep ∝ high) no
              apply -
              apply (drule Dags-eq-hp-eq)
              apply auto
              done
            have rept-end-dag: Dag (rep p) lowb (highb(rep p := pb)) rept
            proof -
              have ∀ no ∈ set-of rept.
                     lowb no = (rep ∝ low) no ∧ (highb(rep p := pb)) no = (rep ∝
high) no
              proof
                fix no
                assume no-in-rept: no ∈ set-of rept
                show lowb no = (rep ∝ low) no ∧
                       (highb(rep p := pb)) no = (rep ∝ high) no
                proof (cases no ∈ set-of rrept)
                  case True
                  with lowb-highb-def pb-def show ?thesis
                    by simp
                next
                  assume no-notin-rrept: no ∉ set-of rrept
                  show ?thesis
                  proof (cases no ∈ set-of lrept)
                    case True
                    with no-notin-rrept rp-notin-lrept ab-nc
                    have ab-nc-no: lowa no = lowb no ∧ higha no = highb no
                      apply -
                      apply (erule-tac x=no in allE)
                      apply (erule impE)
                      apply simp
                      apply (subgoal-tac no ≠ rep p)
                      apply simp
                      apply blast
                      done
                    from lowa-higha-def True have
                      lowa no = (rep ∝ low) no ∧ higha no = (rep ∝ high) no
                      by auto
                    with ab-nc-no
                    have lowb no = (rep ∝ low) no ∧ highb no =(rep ∝ high) no
```

```
          by simp
        with rp-notin-lrept True show ?thesis
          apply (subgoal-tac no ≠ rep p)
          apply simp
          apply blast
          done
      next
        assume no-notin-lrept: no ∉ set-of lrept
        with no-in-rept rept-def no-notin-rrept have no-rp: no = rep p
          by simp
        with rp-notin-lrept low-lowa-nc
        have a-nc: low no = lowa no ∧ high no = higha no
          by auto
        from rp-notin-rrept no-rp ab-nc
        have (lowa(rep p := pa)) no = lowb no ∧ higha no = highb no
          by auto
        with a-nc pa-def no-rp
        have (rep ∝ low) no = lowb no ∧ high no = highb no
          by auto
        with pb-def no-rp show ?thesis
          by simp
      qed
    qed
  qed
  with rept-dag
  have Dag (rep p) lowb (highb(rep p := pb)) rept =
        Dag (rep p) (rep ∝ low) (rep ∝ high) rept
    apply −
    apply (rule heaps-eq-Dag-eq)
    apply auto
    done
  with rept-dag p-nNull show ?thesis
    by simp
qed
have (∀ pt. pt ∉ set-of rept ⟶ low pt = lowb pt ∧
          high pt = (highb(rep p := pb)) pt)
proof (intro allI impI)
  fix pt
  assume pt-notin-rept: pt ∉ set-of rept
  with rept-def obtain
    pt-notin-lrept: pt ∉ set-of lrept and
    pt-notin-rrept: pt ∉ set-of rrept and
    pt-neq-rp: pt ≠ rep p
    by simp
  with low-lowa-nc ab-nc
  show low pt = lowb pt ∧ high pt = (highb(rep p := pb)) pt
    by auto
qed
with rept-end-dag show ?thesis
```

```
          by simp
        qed
      done
    qed
  done
 qed
qed

end
```

10 Proof of Procedure Normalize

```
theory NormalizeTotalProof imports LevellistProof ShareReduceRepListProof
            RepointProof begin

hide-const (open) DistinctTreeProver.set-of tree.Node tree.Tip

lemma (in Normalize-impl) Normalize-modifies:
  shows
   ∀σ. Γ⊢{σ} ´p :== PROC Normalize (´p)
     {t. t may-only-modify-globals σ in [rep,mark,low,high,next]}
  apply (hoare-rule HoarePartial.ProcRec1)
  apply (vcg spec=modifies)
  done

lemma (in Normalize-impl) Normalize-spec:
  shows ∀σ pret prebdt. Γ⊢t
  {|σ. Dag ´p ´low ´high pret ∧ ordered pret ´var ∧
   ´p ≠ Null ∧ (∀n. n ∈ set-of pret ⟶ ´mark n = ´mark ´p) ∧
   bdt pret ´var = Some prebdt|}
  ´p :== PROC Normalize(´p)
  {|(∀pt. pt ∉ set-of pret
    ⟶ σrep pt = ´rep pt ∧ σlow pt = ´low pt ∧ σhigh pt = ´high pt ∧
       σmark pt = ´mark pt ∧ σnext pt = ´next pt) ∧
  (∃postt. Dag ´p ´low ´high postt ∧ reduced postt ∧
  shared postt σvar ∧ ordered postt σvar ∧
  set-of postt ⊆ set-of pret ∧
  (∃postbdt. bdt postt σvar = Some postbdt ∧ prebdt ∼ postbdt)) ∧
  (∀ no. no ∈ set-of pret ⟶ ´mark no = (¬ σmark σp)) |}
  apply (hoare-rule HoareTotal.ProcNoRec1)
  apply (hoare-rule anno=
    ´levellist :==replicate (´p→´var + 1) Null;;
    ´levellist :== CALL Levellist (´p, (¬ ´p→´mark) , ´levellist);;
    (ANNO (τ,ll). {|τ. Levellist ´levellist ´next ll ∧
                Dag σp σlow σhigh pret ∧ ordered pret σvar ∧ σp ≠ Null ∧
              (bdt pret σvar = Some prebdt) ∧
              wf-ll pret ll σvar ∧
              length ´levellist = ((σp → σvar) + 1) ∧
              wf-marking pret σmark ´mark (¬ σmark σp) ∧
```

```
          (∀ pt. pt ∉ set-of pret ⟶ σnext pt = ´next pt) ∧
          ´low = σlow ∧ ´high = σhigh  ∧  ´p = σp ∧ ´rep = σrep ∧
          ´var = σvar⦄
  ´n :==0;;
  WHILE (´n < length ´levellist)
  INV ⦃Levellist ´levellist ´next ll ∧
     Dag σp σlow σhigh pret ∧ ordered pret σvar ∧ σp ≠ Null ∧
     (bdt pret σvar  = Some prebdt) ∧ wf-ll pret ll σvar  ∧
     length τlevellist = ((σp → σvar) + 1) ∧
     wf-marking pret σmark τmark (¬ σmark σp) ∧
     τlow = σlow ∧ τhigh = σhigh  ∧  τp = σp ∧ τrep = σrep  ∧ τvar = σvar ∧
     ´n ≤ length  τlevellist ∧
     (∀ pt i. (pt ∉ set-of pret ∨ (´n <= i ∧ pt ∈ set (ll ! i) ∧
             i <length τlevellist )
             ⟶ σrep pt = ´rep pt)) ∧
     ´rep ‘ Nodes ´n ll ⊆ Nodes ´n ll ∧
     (∀ no ∈ Nodes ´n ll.
       no→´rep→σvar <= no→σvar ∧
       (∃ not nort. Dag (´rep no) (´rep ∝ σlow ) (´rep ∝ σhigh ) nort ∧
         Dag no σlow σhigh not ∧ reduced nort ∧
         ordered nort σvar ∧ set-of nort ⊆ ´rep ‘ Nodes ´n ll ∧
         (∀  no ∈ set-of nort. ´rep no = no) ∧
         (∃ nobdt norbdt. bdt not σvar = Some nobdt ∧
           bdt nort σvar = Some norbdt ∧ nobdt ∼ norbdt))) ∧
     (∀ t1 t2.
        t1∈Dags (´rep ‘(Nodes ´n ll))(´rep ∝ σlow )(´rep ∝ σhigh)∧
        t2∈Dags (´rep ‘(Nodes ´n ll))(´rep ∝ σlow )(´rep ∝ σhigh)
        ⟶
        isomorphic-dags-eq t1 t2  σvar) ∧
     ´levellist = τlevellist ∧ ´next = τnext ∧ ´mark = τmark ∧ ´low = σlow ∧
     ´high = σhigh  ∧ ´p = σp ∧  ´var = σvar ⦄
  VAR MEASURE (length ´levellist − ´n)
  DO
  CALL ShareReduceRepList(´levellist ! ´n);;
  ´n :==´n + 1
  OD
  ⦃(∃ postnormt. Dag (´rep σp) (´rep ∝ σlow ) (´rep ∝ σhigh ) postnormt ∧
    reduced postnormt ∧ shared postnormt σvar ∧
    ordered postnormt σvar ∧ set-of postnormt ⊆ set-of pret ∧
     (∃ postnormbdt. bdt postnormt σvar = Some postnormbdt ∧ prebdt ∼ post-
normbdt) ∧
    (∀  no ∈ set-of postnormt. (´rep no = no))) ∧
    ordered pret σvar  ∧ σp ≠ Null ∧
    (∀  pt. pt ∉ set-of pret ⟶ σrep pt = ´rep pt) ∧
    ´levellist = τlevellist ∧ ´next = τnext ∧ ´mark = τmark ∧ ´low = σlow ∧ ´high
= σhigh ∧
    ´p = σp ∧ (∀  no. no ∈ set-of pret ⟶ ´mark no = (¬ σmark σp)) ⦄)
  ;;
  ´p :== CALL Repoint (´p)
```

in *HoareTotal.annotateI*)
apply (*vcg spec=spec-total*)
prefer *2*

apply (*simp add*: *Nodes-def null-comp-def*)

apply (*rule-tac x=pret* **in** *exI*)
apply *clarify*
apply (*rule conjI*)
apply *clarsimp*
apply (*case-tac i*)
apply *simp*
apply *simp*
apply (*rule conjI*)
apply *simp*
apply (*rule conjI*)
apply *simp*
apply (*rule conjI*)
apply *simp*
apply *clarify*
apply (*simp* (*no-asm-use*) *only*: *simp-thms*)
apply (*rule-tac x=ll* **in** *exI*)
apply (*rule conjI*)
apply *assumption*
apply *clarify*
apply (*simp only*: *simp-thms triv-forall-equality True-implies-equals*)
apply (*rule-tac x=postnormt* **in** *exI*)
apply (*rule conjI*)
apply *simp*
apply (*rule conjI*)
apply *simp*
apply *clarify*
apply (*simp* (*no-asm-simp*))
prefer *2*

apply *clarify*
apply (*simp only*: *simp-thms triv-forall-equality True-implies-equals*)
apply (*rule-tac x=ll!n* **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *Levellist-def*)
prefer *3*

apply (*clarify*)
apply (*simp* (*no-asm-use*) *only*: *simp-thms triv-forall-equality True-implies-equals*)

proof −
— End of while (invariant + false condition) to end of inner SPEC
fix *var p rep mark vara lowa higha pa levellista repa marka nexta varb ll nb pret prebdt* **and** *low* :: *ref* ⇒ *ref* **and**

high :: *ref* ⇒ *ref* **and** *repb* :: *ref* ⇒ *ref*
assume *ll*: *Levellist levellista nexta ll*
assume *wf-lla*: *wf-ll pret ll var*
assume *length-lla*: *length levellista* = *var p* + *1*
assume *ord-pret*: *ordered pret var*
assume *pnN*: *p* ≠ *Null*
assume *rep-repb-nc*:
∀ *pt i*. *pt* ∉ *set-of pret* ∨ *nb* ≤ *i* ∧ *pt* ∈ *set* (*ll* ! *i*) ∧
i < *length levellista*
⟶ *rep pt* = *repb pt*

assume *wf-marking-prop*: *wf-marking pret mark marka* (¬ *mark p*)
assume *pret-dag*: *Dag p low high pret*
assume *prebdt*: *bdt pret var* = *Some prebdt*
assume *not-nbslla*: ¬ *nb* < *length levellista*
assume *nb-le-lla*: *nb* ≤ *length levellista*

assume *normalize-prop*: ∀ *no*∈*Nodes nb ll*.
var (*repb no*) ≤ *var no* ∧
(∃ *not nort*. *Dag* (*repb no*) (*repb* ∝ *low*) (*repb* ∝ *high*) *nort* ∧
Dag no low high not ∧ *reduced nort* ∧ *ordered nort var* ∧
set-of nort ⊆ *repb* ‘ *Nodes nb ll* ∧
(∀ *no*∈*set-of nort*. *repb no* = *no*) ∧
(∃ *nobdt norbdt*. *bdt not var* = *Some nobdt* ∧
bdt nort var = *Some norbdt* ∧ *nobdt* ∼ *norbdt*))
assume *repbNodes-in-Nodes*: *repb* ‘ *Nodes nb ll* ⊆ *Nodes nb ll*
assume *shared-mult-dags*:
∀ *t1 t2*. *t1* ∈ *Dags* (*repb* ‘ *Nodes nb ll*) (*repb* ∝ *low*) (*repb* ∝ *high*) ∧
t2 ∈ *Dags* (*repb* ‘ *Nodes nb ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
⟶ *isomorphic-dags-eq t1 t2 var*
show (∃ *postnormt*. *Dag* (*repb p*) (*repb* ∝ *low*) (*repb* ∝ *high*) *postnormt* ∧
reduced postnormt ∧ *shared postnormt var* ∧
ordered postnormt var ∧ *set-of postnormt* ⊆ *set-of pret* ∧
(∃ *postnormbdt*.
bdt postnormt var = *Some postnormbdt* ∧ *prebdt* ∼ *postnormbdt*) ∧
(∀ *no* ∈ *set-of postnormt*. *repb no* = *no*)) ∧
ordered pret var ∧ *p* ≠ *Null* ∧
(∀ *pt*. *pt* ∉ *set-of pret* ⟶ *rep pt* = *repb pt*) ∧
(∀ *no*. *no* ∈ *set-of pret* ⟶ *marka no* = (¬ *mark p*))

proof −
from *ll* **have** *length-ll-eq*: *length levellista* = *length ll*
by (*simp add*: *Levellist-length*)
from *rep-repb-nc* **have** *rep-nc-post*: ∀ *pt*. *pt* ∉ *set-of pret* ⟶ *rep pt* = *repb pt*
by *auto*
from *pnN pret-dag* **obtain** *lt rt* **where** *pret-def*: *pret* = *Node lt p rt*
by *auto*
from *wf-marking-prop pret-def*
have *marking-inverted*: (∀ *no*. *no* ∈ *set-of pret* ⟶ *marka no* = (¬ *mark p*))

by (*simp add*: *wf-marking-def*)
from *not-nbslla nb-le-lla* **have** *nb-length-lla*: *nb* = *length levellista*
by *simp*
with *length-lla* **have** *varp-s-nb*: *var p* < *nb*
by *simp*
from *pret-def* **have** *p-in-pret*: *p* ∈ *set-of pret*
by *simp*
with *wf-lla* **have** *p* ∈ *set* (*ll* ! (*var p*))
by (*simp add*: *wf-ll-def*)
with *varp-s-nb* **have** *p-in-Nodes*: *p* ∈ *Nodes nb ll*
by (*auto simp add*: *Nodes-def*)
with *normalize-prop* **obtain** *not nort* **where**
varrepno-varno: *var* (*repb p*) ≤ *var p* **and**
nort-dag: *Dag* (*repb p*) (*repb* ∝ *low*) (*repb* ∝ *high*) *nort* **and**
not-dag: *Dag p low high not* **and**
red-nort: *reduced nort* **and**
ord-nort: *ordered nort var* **and**
nort-in-repNodes: *set-of nort* ⊆ *repb* ‘ *Nodes nb ll* **and**
nort-repb: (∀ *no*∈*set-of nort*. *repb no* = *no*) **and**
bdt-prop: ∃ *nobdt norbdt*. *bdt not var* = *Some nobdt* ∧ *bdt nort var* = *Some norbdt* ∧
nobdt ∼ *norbdt*
by *auto*

from *wf-lla nb-length-lla* **have** *Nodes-in-pret*: *Nodes nb ll* ⊆ *set-of pret*
apply −
apply (*rule Nodes-in-pret*)
apply (*auto simp add*: *length-ll-eq*)
done
from *pret-dag wf-lla nb-length-lla* **have** *Null* ∉ *Nodes nb ll*
apply −
apply (*rule Null-notin-Nodes*)
apply (*auto simp add*: *length-ll-eq*)
done
with *p-in-Nodes repbNodes-in-Nodes* **have** *rp-nNull*: *repb p* ≠ *Null*
by *auto*
with *nort-dag* **have** *nort-nTip*: *nort*≠ *Tip*
by *auto*
have ∃ *postnormt*. *Dag* (*repb p*) (*repb* ∝ *low*) (*repb* ∝ *high*) *postnormt* ∧
reduced postnormt ∧ *shared postnormt var* ∧
ordered postnormt var ∧ *set-of postnormt* ⊆ *set-of pret* ∧
(∃ *postnormbdt*.
bdt postnormt var = *Some postnormbdt* ∧ *prebdt* ∼ *postnormbdt*) ∧
(∀ *no* ∈ *set-of postnormt*. *repb no* = *no*)
proof (*rule-tac x*=*nort* **in** *exI*)
from *nort-in-repNodes repbNodes-in-Nodes Nodes-in-pret*
have *nort-in-pret*: *set-of nort* ⊆ *set-of pret*
by *blast*
from *not-dag pret-dag* **have** *not-pret*: *not* = *pret*

```
  by (simp add: Dag-unique)
with bdt-prop prebdt have pret-bdt-prop:
  ∃ postnormbdt.
  bdt nort var = Some postnormbdt ∧ prebdt ∼ postnormbdt
  by auto
from shared-mult-dags have shared nort var
proof (auto simp add: shared-def isomorphic-dags-eq-def)
  fix st1 st2 bdt1
  assume shared-imp:
    ∀ t1 t2. t1∈Dags (repb ‘ Nodes nb ll) (repb ∝ low) (repb ∝ high) ∧
    t2 ∈ Dags (repb ‘ Nodes nb ll) (repb ∝ low) (repb ∝ high)
    ⟶
    (∃ bdt1. bdt t1 var = Some bdt1 ∧ bdt t2 var = Some bdt1) ⟶ t1 = t2
  assume st1-nort: st1 ≤ nort
  assume st2-nort: st2 ≤ nort
  assume bdt-st1: bdt st1 var = Some bdt1
  assume bdt-st2: bdt st2 var = Some bdt1
  from nort-in-repNodes nort-dag nort-nTip
  have nort-in-DagsrNodes:
    nort ∈ Dags (repb ‘(Nodes nb ll)) (repb ∝ low) (repb ∝ high)
    apply −
    apply (rule DagsI)
    apply auto
    done
  show st1 = st2
  proof (cases st1)
    case Tip
    note st1-Tip=this
    with bdt-st1 bdt-st2 show ?thesis
      by auto
  next
    case (Node lst1 st1p rst1)
    note st1-Node=this
    then have st1-nTip: st1 ≠ Tip
      by simp
    show ?thesis
    proof (cases st2)
      case Tip
      with bdt-st1 bdt-st2 show ?thesis
        by auto
    next
      case (Node lst2 st2p rst2)
      note st2-Node=this
      then have st2-nTip: st2 ≠ Tip
        by simp
      from nort-in-DagsrNodes st1-nort ord-nort wf-lla st1-nTip
      have st1-in-Dags:
        st1 ∈ Dags (repb ‘ Nodes nb ll) (repb ∝ low) (repb ∝ high)
        apply −
```

apply (*rule Dags-subdags*)
apply *auto*
done
from *nort-in-DagsrNodes st2-nort ord-nort wf-lla st2-nTip*
have *st2-in-Dags*:
st2 $\in$ *Dags* (*repb* ‘ *Nodes nb ll*) (*repb* $\propto$ *low*) (*repb* $\propto$ *high*)
apply −
apply (*rule Dags-subdags*)
apply *auto*
done
from *st1-in-Dags st2-in-Dags bdt-st1 bdt-st2 shared-imp* **show** *st1=st2*
by *simp*
qed
qed
qed
with *nort-dag red-nort ord-nort nort-in-pret pret-bdt-prop nort-repb*
show *Dag* (*repb p*) (*repb* $\propto$ *low*) (*repb* $\propto$ *high*) *nort* $\wedge$
reduced nort $\wedge$ *shared nort var* $\wedge$ *ordered nort var* $\wedge$
set-of nort $\subseteq$ *set-of pret* $\wedge$
($\exists$ *postnormbdt*.
bdt nort var = *Some postnormbdt* $\wedge$ *prebdt* $\sim$ *postnormbdt*) $\wedge$
($\forall$ *no* $\in$ *set-of nort*. *repb no* = *no*)
apply −
apply (*intro conjI*)
apply *assumption+*
done
qed
with *wf-lla length-lla ord-pret pnN rep-nc-post marking-inverted*
show *?thesis*
by *simp*
qed
next
— From postcondition inner SPEC to final postcondition
fix *var low high p rep levellist marka next*
nexta lowb highb pb levellista ll repa pret prebdt
and *mark*::*ref*$\Rightarrow$*bool* **and** *postnormt postnormbdt*
assume *ll*: *Levellist levellista nexta ll*
assume *repoint-spec*:
Dag pb lowb highb postnormt
$\forall$ *pt*. *pt* $\notin$ *set-of postnormt* $\longrightarrow$ *low pt* = *lowb pt* $\wedge$ *high pt* = *highb pt*
assume *pret-dag*: *Dag p low high pret*
assume *ord-pret*: *ordered pret var*
assume *pnN*: *p* $\neq$ *Null*
assume *onemark-pret*:
$\forall$ *n*. *n* $\in$ *set-of pret* $\longrightarrow$ *mark n* = *mark p* (**is** $\forall$ *n*. *?in-pret n* $\longrightarrow$ *?eq-mark-p n*)
assume *pret-bdt*: *bdt pret var* = *Some prebdt*

assume *wf-ll*: *wf-ll pret ll var* **and**
length-ll:*length levellist* =*var p* + *1* **and**

wf-marking-ll: *wf-marking pret mark marka* (¬ *mark p*)
assume
postnormt-dag: *Dag* (*repa p*) (*repa* ∝ *low*) (*repa* ∝ *high*) *postnormt* **and**
reduced-postnormt: *reduced postnormt* **and**
shared-postnormt: *shared postnormt var* **and**
ordered-postnormt: *ordered postnormt var* **and**
subset-pret: *set-of postnormt* ⊆ *set-of pret***and**
sim-bdt: *bdt postnormt var* = *Some postnormbdt prebdt* ∼ *postnormbdt* **and**
postdag-repa: ∀ *no* ∈ *set-of postnormt*. *repa no* = *no* **and**
rep-eq: ∀ *pt*. *pt* ∉ *set-of pret* ⟶ *rep pt* = *repa pt* **and**
pret-marked: ∀ *no*. *no* ∈ *set-of pret* ⟶ *marka no* = (¬ *mark p*)
assume *unmodif-next*: ∀ *p*. *p* ∉ *set-of pret* ⟶ *next p* = *nexta p*
show (∀ *pt*. *pt* ∉ *set-of pret*
⟶ *low pt* = *lowb pt* ∧ *high pt* = *highb pt* ∧
mark pt = *marka pt*)

proof −
from *ll* **have** *length-ll-eq*: *length levellista* = *length ll*
by (*simp add*: *Levellist-length*)
from *repoint-spec pnN subset-pret*
have *repoint-nc*: (∀ *pt*. *pt* ∉ *set-of pret*
⟶ *low pt* = *lowb pt* ∧ *high pt* = *highb pt*) ∧ *Dag pb lowb highb postnormt*
by *auto*
then have *lowhigh-b-eq*: ∀ *pt*. *pt* ∉ *set-of pret*
⟶ *low pt* = *lowb pt* ∧ *high pt* = *highb pt*
by *fastforce*
from *wf-marking-ll pret-dag pnN*
have *mark-b-eq*: ∀ *pt*. *pt* ∉ *set-of pret* ⟶ *mark pt* = *marka pt*
apply −
apply (*simp add*: *wf-marking-def*)
apply (*split dag.splits*)
apply *simp*
apply (*rule allI*)
apply (*rule impI*)
apply (*elim conjE*)
apply (*erule-tac x*=*pt* **in** *allE*)
apply *fastforce*
done
with *lowhigh-b-eq rep-eq unmodif-next*
have *pret-nc*: ∀ *pt*. *pt* ∉ *set-of pret*
⟶ *rep pt* = *repa pt* ∧ *low pt* = *lowb pt* ∧ *high pt* = *highb pt* ∧
mark pt = *marka pt* ∧ *next pt* = *nexta pt*
by *blast*

from *pret-nc*
show *?thesis*
by *fastforce*
qed
next

— invariant to invariant
fix *var low high p rep mark pret prebdt levellist ll next marka n repc*
  and *repb* :: *ref* ⇒ *ref*
assume *ll*: *Levellist levellist next ll*
assume *pret-dag*: *Dag p low high pret*
assume *ord-pret*: *ordered pret var*
assume *pnN*: *p* ≠ *Null*
assume *prebdt-pret*: *bdt pret var* = *Some prebdt*
assume *wf-ll*: *wf-ll pret ll var*
assume *lll*: *length levellist* = *var p* + *1*
assume *n-Suc-var-p*: *n* < *var p* + *1*
assume *wf-marking-m-ma*: *wf-marking pret mark marka* (¬ *mark p*)

assume *rep-nc*: ∀ *pt i*.
    pt ∉ *set-of pret* ∨ *n* ≤ *i* ∧ *pt* ∈ *set* (*ll* ! *i*) ∧ *i* < *var p* + *1* ⟶
    rep pt = *repb pt*
assume *repbNodes-in-Nodes*: *repb* ‘ *Nodes n ll* ⊆ *Nodes n ll*
assume
  normalize-prop: ∀ *no*∈*Nodes n ll*.
  var (*repb no*) ≤ *var no* ∧
  (∃ *not nort*. *Dag* (*repb no*) (*repb* ∝ *low*) (*repb* ∝ *high*) *nort* ∧
  Dag no low high not ∧ *reduced nort* ∧ *ordered nort var* ∧
  set-of nort ⊆ *repb* ‘ *Nodes n ll* ∧
  (∀ *no*∈*set-of nort*. *repb no* = *no*) ∧
  (∃ *nobdt*. *bdt not var* = *Some nobdt* ∧
  (∃ *norbdt*. *bdt nort var* = *Some norbdt* ∧
  nobdt ∼ *norbdt*)))
assume
  isomorphic-dags-eq:
  ∀ *t1 t2*. *t1* ∈ *Dags* (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)∧
  t2 ∈ *Dags* (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
  ⟶ *isomorphic-dags-eq t1 t2 var*
show (∀ *no*∈*set* (*ll* ! *n*).
      no ≠ *Null* ∧
      (*low no* = *Null*) = (*high no* = *Null*) ∧
      low no ∉ *set* (*ll* ! *n*) ∧
      high no ∉ *set* (*ll* ! *n*) ∧
      isLeaf-pt no low high = (*var no* ≤ *1*) ∧
      (*low no* ≠ *Null* ⟶ *repb* (*low no*) ≠ *Null*) ∧ (*repb* ∝ *low*) *no* ∉ *set* (*ll*
! *n*)) ∧
    (∀ *no1*∈*set* (*ll* ! *n*). ∀ *no2*∈*set* (*ll* ! *n*). *var no1* = *var no2*) ∧
    (∀ *repa*. (∀ *no*. *no* ∉ *set* (*ll* ! *n*) ⟶ *repb no* = *repa no*) ∧
        (∀ *no*∈*set* (*ll* ! *n*).
          repa no ≠ *Null* ∧
          (*if* (*repa* ∝ *low*) *no* = (*repa* ∝ *high*) *no* ∧ *low no* ≠ *Null*
           then repa no = (*repa* ∝ *low*) *no*
           else repa no ∈ *set* (*ll* ! *n*) ∧
             repa (*repa no*) = *repa no* ∧

(∀ *no1*∈*set* (*ll* ! *n*).
((*repa* ∝ *high*) *no1* = (*repa* ∝ *high*) *no* ∧
(*repa* ∝ *low*) *no1* = (*repa* ∝ *low*) *no*) =
(*repa no* = *repa no1*)))) ⟶
var p + *1* − (*n* + *1*) < *var p* + *1* − *n* ∧
n + *1* ≤ *var p* + *1* ∧
(∀ *pt i*. *pt* ∉ *set-of pret* ∨ (*n* + *1* ≤ *i* ∧ *pt* ∈ *set* (*ll* ! *i*) ∧ *i* < *var p*
+ *1*) ⟶
rep pt = *repa pt*) ∧
repa ' *Nodes* (*n* + *1*) *ll* ⊆ *Nodes* (*n* + *1*) *ll* ∧
(∀ *no*∈*Nodes* (*n* + *1*) *ll*.
var (*repa no*) ≤ *var no* ∧
(∃ *not nort*.
Dag (*repa no*) (*repa* ∝ *low*) (*repa* ∝ *high*) *nort* ∧
Dag no low high not ∧
reduced nort ∧
ordered nort var ∧
set-of nort ⊆ *repa* ' *Nodes* (*n* + *1*) *ll* ∧
(∀ *no*∈*set-of nort*. *repa no* = *no*) ∧
(∃ *nobdt*.
bdt not var = *Some nobdt* ∧
(∃ *norbdt*. *bdt nort var* = *Some norbdt* ∧ *nobdt* ∼ *norbdt*))))
∧
(∀ *t1 t2*.
t1 ∈ *Dags* (*repa* ' *Nodes* (*n* + *1*) *ll*) (*repa* ∝ *low*) (*repa* ∝ *high*) ∧
t2 ∈ *Dags* (*repa* ' *Nodes* (*n* + *1*) *ll*) (*repa* ∝ *low*) (*repa* ∝ *high*)
⟶
isomorphic-dags-eq t1 t2 var))

proof −
from *ll* **have** *length-ll-eq*: *length levellist* = *length ll*
by (*simp add*: *Levellist-length*)
from *n-Suc-var-p lll* **have** *nsll*: *n* < *length levellist* **by** *simp*
hence *nseqll*: *n* ≤ *length levellist* **by** *simp*
have *srrl-precond*: (∀ *no* ∈ *set* (*ll* ! *n*).
no ≠ *Null* ∧
(*low no* = *Null*) = (*high no* = *Null*) ∧
low no ∉ *set* (*ll* ! *n*) ∧
high no ∉ *set* (*ll* ! *n*) ∧
isLeaf-pt no low high = (*var no* ≤ *1*) ∧
(*low no* ≠ *Null* ⟶ *repb* (*low no*) ≠ *Null*) ∧
(*repb* ∝ *low*) *no* ∉ *set* (*ll* ! *n*))
proof (*intro ballI*)
fix *no*
assume *no-in-lln*: *no* ∈ *set* (*ll* ! *n*)
with *wf-ll nsll* **have** *no-in-pret-var*: *no* ∈ *set-of pret* ∧ *var no* = *n*
by (*simp add*: *wf-ll-def length-ll-eq*)
with *pret-dag* **have** *no-nNull*: *no* ≠ *Null*
apply −
apply (*rule set-of-nn*)

```
  apply auto
  done
from pret-dag prebdt-pret no-in-pret-var
have balanced-no: (low no = Null) = (high no = Null)
  apply −
  apply (erule conjE)
  apply (rule-tac p=p and low=low in balanced-bdt)
  apply auto
  done
have low-no-notin-lln: low no ∉ set (ll ! n)
proof (cases low no = Null)
  case True
  note lno-Null=this
  with balanced-no have hno-Null: high no = Null
    by simp
  show ?thesis
  proof (cases low no ∈ set (ll ! n))
    case True
    with wf-ll nsll have low no ∈ set-of pret ∧ var (low no) = n
      by (auto simp add: wf-ll-def length-ll-eq)
    with pret-dag have low no ≠ Null
      apply −
      apply (rule set-of-nn)
      apply auto
      done
    with lno-Null show ?thesis
      by simp
  next
    assume lno-notin-lln: low no ∉ set (ll ! n)
    then show ?thesis
      by simp
  qed
next
  assume lno-nNull: low no ≠ Null
  with balanced-no have hno-nNull: high no ≠ Null
    by simp
  with lno-nNull pret-dag ord-pret no-in-pret-var
 have var-children-smaller: var (low no) < var no ∧ var (high no) < var no
    apply −
    apply (rule var-ordered-children)
    apply auto
    done
  show ?thesis
  proof (cases low no ∈ set (ll ! n))
    case True
    with wf-ll nsll have low no ∈ set-of pret ∧ var (low no) = n
      by (simp add: wf-ll-def length-ll-eq)
    with var-children-smaller no-in-pret-var show ?thesis
      by simp
```

```
  next
    assume low no ∉ set (ll ! n)
    thus ?thesis
      by simp
  qed
qed
have high-no-notin-lln: high no ∉ set (ll ! n)
proof (cases high no = Null)
  case True
  note hno-Null=this
  with balanced-no have lno-Null: low no = Null
    by simp
  show ?thesis
  proof (cases high no ∈ set (ll ! n))
    case True
    with wf-ll nsll have high no ∈ set-of pret ∧ var (high no) = n
      by (auto simp add: wf-ll-def length-ll-eq)
    with pret-dag have high no ≠ Null
      apply −
      apply (rule set-of-nn)
      apply auto
      done
    with hno-Null show ?thesis
      by simp
  next
    assume hno-notin-lln: high no ∉ set (ll ! n)
    then show ?thesis
      by simp
  qed
next
  assume hno-nNull: high no ≠ Null
  with balanced-no have lno-nNull: low no ≠ Null
    by simp
  with hno-nNull pret-dag ord-pret no-in-pret-var
 have var-children-smaller: var (low no) < var no ∧ var (high no) < var no
    apply −
    apply (rule var-ordered-children)
    apply auto
    done
  show ?thesis
  proof (cases high no ∈ set (ll ! n))
    case True
    with wf-ll nsll have high no ∈ set-of pret ∧ var (high no) = n
      by (simp add: wf-ll-def length-ll-eq)
    with var-children-smaller no-in-pret-var show ?thesis
      by simp
  next
    assume high no ∉ set (ll ! n)
    thus ?thesis
```

```
      by simp
  qed
qed
from no-in-pret-var pret-dag no-nNull obtain not where
  no-dag-ex: Dag no low high not
  apply -
  apply (rotate-tac 2)
  apply (drule subnode-dag-cons)
  apply (auto simp del: Dag-Ref)
  done
with pret-dag prebdt-pret no-in-pret-var obtain nobdt
  where nobdt-ex:
  bdt not var = Some nobdt
  apply -
  apply (drule subbdt-ex-dag-def)
  apply auto
  done
have isLeaf-var: isLeaf-pt no low high = (var no ≤ 1)
proof
  assume no-isLeaf: isLeaf-pt no low high
  from nobdt-ex no-dag-ex no-isLeaf show var no ≤ 1
    apply -
    apply (rule bdt-Some-Leaf-var-le-1)
    apply auto
    done
next
  assume varno-le-1: var no ≤ 1
  show isLeaf-pt no low high
  proof (cases var no = 0)
    case True
    with nobdt-ex no-nNull no-dag-ex have not = Node Tip no Tip
      apply -
      apply (drule bdt-Some-var0-Zero)
      apply auto
      done
    with no-dag-ex show isLeaf-pt no low high
      by (simp add: isLeaf-pt-def)
  next
    assume var no ≠ 0
    with varno-le-1 have var no = 1
      by simp
    with nobdt-ex no-nNull no-dag-ex have not = Node Tip no Tip
      apply -
      apply (drule bdt-Some-var1-One)
      apply auto
      done
    with no-dag-ex show isLeaf-pt no low high
      by (simp add: isLeaf-pt-def)
  qed
```

qed
have *repb-low-nNull*: (*low no* ≠ *Null* ⟶ *repb* (*low no*) ≠ *Null*)
proof
assume *lno-nNull*: *low no* ≠ *Null*
with *no-nNull no-in-pret-var pret-dag* **have** *lno-in-pret*: *low no* ∈ *set-of pret*
apply −
apply (*rule-tac low*=*low* **in** *subelem-set-of-low*)
apply *auto*
done
from *lno-nNull balanced-no* **have** *hno-nNull*: *high no* ≠ *Null*
by *simp*
with *lno-nNull pret-dag ord-pret no-in-pret-var*
have *var-children-smaller*: *var* (*low no*) < *var no* ∧ *var* (*high no*) < *var no*
apply −
apply (*rule var-ordered-children*)
apply *auto*
done
with *no-in-pret-var* **have** *var-lno-l-n*: *var* (*low no*) <*n*
by *simp*
with *wf-ll lno-in-pret nsll* **have** *low no* ∈ *set* (*ll* ! (*var* (*low no*)))
by (*simp add*: *wf-ll-def length-ll-eq*)
with *lno-in-pret var-lno-l-n* **have** *low no* ∈ *Nodes n ll*
apply (*simp add*: *Nodes-def*)
apply (*rule-tac x*=*var* (*low no*) **in** *exI*)
apply *simp*
done
hence *repb* (*low no*) ∈ *repb* ‘ *Nodes n ll*
by *simp*
with *repbNodes-in-Nodes* **have** *repb-lno-in-Nodes*:
repb (*low no*) ∈ *Nodes n ll*
by *blast*
from *pret-dag wf-ll nsll* **have** *Null* ∉ *Nodes n ll*
apply −
apply (*rule Null-notin-Nodes*)
apply (*auto simp add*: *length-ll-eq*)
done
with *repb-lno-in-Nodes* **show** *repb* (*low no*) ≠ *Null*
by *auto*
qed
have *Null-notin-lln*: *Null* ∉ *set* (*ll* ! *n*)
proof (*cases Null* ∈ *set* (*ll* ! *n*))
case *True*
with *wf-ll nsll* **have** *Null* ∈ *set-of pret* ∧ *var* (*Null*) = *n*
by (*simp add*: *wf-ll-def length-ll-eq*)
with *pret-dag* **have** *Null* ≠ *Null*
apply −
apply (*rule set-of-nn*)
apply *auto*
done

```
  thus ?thesis
    by auto
next
  assume Null ∉ set (ll ! n)
  thus ?thesis
    by simp
qed
have (repb ∝ low) no ∉ set (ll ! n)
proof (cases low no = Null)
  case True
  with Null-notin-lln show ?thesis
    by (simp add: null-comp-def)
next
  assume lno-nNull: low no ≠ Null
  with no-nNull no-in-pret-var pret-dag have lno-in-pret: low no ∈ set-of pret
    apply –
    apply (rule-tac low=low in subelem-set-of-low)
    apply auto
    done
  from lno-nNull have propto-eq-comp: (repb ∝ low) no = repb (low no)
    by (simp add: null-comp-def)
  from lno-nNull balanced-no have hno-nNull: high no ≠ Null
    by simp
  with lno-nNull pret-dag ord-pret no-in-pret-var
  have var-children-smaller: var (low no) < var no ∧ var (high no) < var no
    apply –
    apply (rule var-ordered-children)
    apply auto
    done
  with no-in-pret-var have var-lno-l-n: var (low no) <n
    by simp
  with wf-ll lno-in-pret nsll have low no ∈ set (ll ! (var (low no)))
    by (simp add: wf-ll-def length-ll-eq)
  with lno-in-pret var-lno-l-n have lno-in-Nodes-n: low no ∈ Nodes n ll
    apply (simp add: Nodes-def)
    apply (rule-tac x=var (low no) in exI)
    apply simp
    done
  hence repb (low no) ∈ repb ‘ Nodes n ll
    by simp
  with  repbNodes-in-Nodes have repb-lno-in-Nodes:
    repb (low no) ∈ Nodes n ll
    by blast
   with lno-in-Nodes-n normalize-prop have var (repb (low no)) ≤ var (low
no)
    by auto
  with var-lno-l-n have var-rep-lno-l-n:  var (repb (low no)) < n
    by simp
  with repb-lno-in-Nodes have ∃ k < n. repb (low no) ∈ set (ll ! k)
```

by (*auto simp add*: *Nodes-def*)
with *wf-ll propto-eq-comp nsll* **show** $(repb \propto low)\ no \notin set\ (ll\ !\ n)$
apply −
apply (*erule exE*)
apply (*rule-tac i=k* **and** *j=n* **in** *no-in-one-ll*)
apply (*auto simp add*: *length-ll-eq*)
done
qed
with *no-nNull balanced-no low-no-notin-lln high-no-notin-lln isLeaf-var repb-low-nNull*
show $no \neq Null \wedge$
$(low\ no = Null) = (high\ no = Null) \wedge$
$low\ no \notin set\ (ll\ !\ n) \wedge high\ no \notin set\ (ll\ !\ n) \wedge$
$isLeaf\text{-}pt\ no\ low\ high = (var\ no \leq 1) \wedge$
$(low\ no \neq Null \longrightarrow repb\ (low\ no) \neq Null) \wedge$
$(repb \propto low)\ no \notin set\ (ll\ !\ n)$
by *auto*
qed
have *all-nodes-same-var*: $\forall no1 \in set\ (ll\ !\ n).\ \forall no2 \in set\ (ll\ !\ n).\ var\ no1 =$
$var\ no2$
proof (*intro ballI impI*)
fix *no1 no2*
assume $no1 \in set\ (ll\ !\ n)$
with *wf-ll nsll* **have** *var-lln-i*: $var\ no1 = n$
by (*simp add*: *wf-ll-def length-ll-eq*)
assume $no2 \in set\ (ll\ !\ n)$
with *wf-ll nsll* **have** $var\ no2 = n$
by (*simp add*: *wf-ll-def length-ll-eq*)
with *var-lln-i* **show** $var\ no1 = var\ no2$
by *simp*
qed
have $(\forall repa.\ (\forall no.\ no \notin set\ (ll\ !\ n) \longrightarrow repb\ no = repa\ no) \wedge$
$(\forall no \in set\ (ll\ !\ n).$
$repa\ no \neq Null \wedge$
$(if\ (repa \propto low)\ no = (repa \propto high)\ no \wedge low\ no \neq Null$
$then\ repa\ no = (repa \propto low)\ no$
$else\ repa\ no \in set\ (ll\ !\ n) \wedge$
$repa\ (repa\ no) = repa\ no \wedge$
$(\forall no1 \in set\ (ll\ !\ n).$
$((repa \propto high)\ no1 = (repa \propto high)\ no \wedge$
$(repa \propto low)\ no1 = (repa \propto low)\ no) =$
$(repa\ no = repa\ no1)))) \longrightarrow$
$var\ p + 1 - (n + 1) < var\ p + 1 - n \wedge$
$n + 1 \leq var\ p + 1 \wedge$
$(\forall pt\ i.\ pt \notin set\text{-}of\ pret \vee (n + 1 \leq i \wedge pt \in set\ (ll\ !\ i) \wedge i < var\ p$
$+ 1) \longrightarrow$
$rep\ pt = repa\ pt) \wedge$
$repa\ `\ Nodes\ (n + 1)\ ll \subseteq Nodes\ (n + 1)\ ll \wedge$
$(\forall no \in Nodes\ (n + 1)\ ll.$
$var\ (repa\ no) \leq var\ no \wedge$

(∃ *not nort*.
Dag (*repa no*) (*repa* ∝ *low*) (*repa* ∝ *high*) *nort* ∧
Dag no low high not ∧
reduced nort ∧
ordered nort var ∧
set-of nort ⊆ *repa* ' *Nodes* (*n* + *1*) *ll* ∧
(∀ *no*∈*set-of nort*. *repa no* = *no*) ∧
(∃ *nobdt*.
bdt not var = *Some nobdt* ∧
(∃ *norbdt*. *bdt nort var* = *Some norbdt* ∧ *nobdt* ∼ *norbdt*))))
∧
(∀ *t1 t2*.
t1 ∈ *Dags* (*repa* ' *Nodes* (*n* + *1*) *ll*) (*repa* ∝ *low*) (*repa* ∝ *high*) ∧
t2 ∈ *Dags* (*repa* ' *Nodes* (*n* + *1*) *ll*) (*repa* ∝ *low*) (*repa* ∝ *high*)
⟶
isomorphic-dags-eq t1 t2 var))
(**is** (∀ *repc*. *?srrl-post repc* ⟶ *?norm-inv repc*))
proof (*intro allI impI*, *elim conjE*)
fix *repc*
assume *repbc-nc*: ∀ *no*. *no* ∉ *set* (*ll* ! *n*) ⟶ *repb no* = *repc no*
assume *rep-prop*: ∀ *no*∈*set* (*ll* ! *n*).
repc no ≠ *Null* ∧
(*if* (*repc* ∝ *low*) *no* = (*repc* ∝ *high*) *no* ∧ *low no* ≠ *Null*
then repc no = (*repc* ∝ *low*) *no*
else repc no ∈ *set* (*ll* ! *n*) ∧
repc (*repc no*) = *repc no* ∧
(∀ *no1*∈*set* (*ll* ! *n*).
((*repc* ∝ *high*) *no1* = (*repc* ∝ *high*) *no* ∧
(*repc* ∝ *low*) *no1* = (*repc* ∝ *low*) *no*) =
(*repc no* = *repc no1*)))
show *?norm-inv repc*
proof −
from *n-Suc-var-p* **have** *termi*: *var p* + *1* − (*n* + *1*) < *var p* + *1* − *n*
by *arith*
from *wf-ll repbc-nc nsll*
have *Nodes-n-rep-nc*: ∀ *p*. *p* ∈ *Nodes n ll* ⟶ *repb p* = *repc p*
apply −
apply (*rule allI*)
apply (*rule impI*)
apply (*simp add*: *Nodes-def*)
apply (*erule exE*)
apply (*erule-tac x*=*p* **in** *allE*)
apply (*drule-tac i*=*x* **and** *j*=*n* **in** *no-in-one-ll*)
apply (*auto simp add*: *length-ll-eq*)
done
from *repbNodes-in-Nodes*
have *Nodes-n-rep-in-Nodesn*:
∀ *p*. *p* ∈ *Nodes n ll* ⟶ *repb p* ∈ *Nodes n ll*
by *auto*

from *wf-ll nsll* **have** *Nodes n ll* ⊆ *set-of pret*
  apply –
  apply (*rule Nodes-in-pret*)
  apply (*auto simp add*: *length-ll-eq*)
  done
with *Nodes-n-rep-in-Nodesn*
have *Nodes-n-rep-in-pret*: ∀ *p*. *p* ∈ *Nodes n ll* ⟶ *repb p* ∈ *set-of pret*
  apply –
  apply (*intro allI impI*)
  apply *blast*
  done
have *Nodes-repbc-Dags-eq*: ∀ *p t*. *p* ∈ *Nodes n ll*
  ⟶ *Dag* (*repb p*) (*repb* ∝ *low*) (*repb* ∝ *high*) *t* =
  Dag (*repc p*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t*
proof (*intro allI impI*)
  fix *p t*
  assume *p-in-Nodes*: *p* ∈ *Nodes n ll*
  then have *repp-nc*: *repb p* = *repc p*
    by (*rule Nodes-n-rep-nc* [*rule-format*])
  from *p-in-Nodes normalize-prop* **obtain** *nort* **where**
    nort-repb-dag: *Dag* (*repb p*) (*repb* ∝ *low*) (*repb* ∝ *high*) *nort* **and**
    nort-in-repbNodes: *set-of nort* ⊆ *repb* ‘ *Nodes n ll*
    apply –
    apply (*erule-tac x*=*p* **in** *ballE*)
    prefer *2*
    apply *auto*
    done
  from *nort-in-repbNodes repbNodes-in-Nodes*
  have *nort-in-Nodesn*: *set-of nort* ⊆ *Nodes n ll*
    by *blast*
  from *pret-dag wf-ll nsll* **have** *Null* ∉ *Nodes n ll*
    apply –
    apply (*rule Null-notin-Nodes*)
    apply (*auto simp add*: *length-ll-eq*)
    done
  with *p-in-Nodes repbNodes-in-Nodes* **have** *repp-nNull*: *repb p* ≠ *Null*
    by *auto*
  from *nort-repb-dag repp-nc*
  have *nort-repbc-dag*: *Dag* (*repc p*) (*repb* ∝ *low*) (*repb* ∝ *high*) *nort*
    by *simp*
  from *nort-in-Nodesn* **have** ∀ *x* ∈ *set-of nort*. *x* ∈ *Nodes n ll*
    apply –
    apply (*rule ballI*)
    apply *blast*
    done
  with *wf-ll nsll* **have** ∀ *x* ∈ *set-of nort*. *x* ∈ *set-of pret* ∧ *var x* < *n*
    apply –
    apply (*rule ballI*)
    apply (*rule wf-ll-Nodes-pret*)

apply (*auto simp add*: *length-ll-eq*)
done
with *pret-dag prebdt-pret nort-repbc-dag ord-pret wf-ll nsll repbc-nc*
have
∀ *x* ∈ *set-of nort*. (*repc* ∝ *low*) *x* = (*repb* ∝ *low*) *x* ∧
(*repc* ∝ *high*) *x* = (*repb* ∝ *high*) *x*
apply −
apply (*rule nort-null-comp*)
apply (*auto simp add*: *length-ll-eq*)
done
with *nort-repbc-dag repp-nc*
have *Dag* (*repc p*) (*repb* ∝ *low*) (*repb* ∝ *high*) *nort* =
Dag (*repc p*) (*repc* ∝ *low*) (*repc* ∝ *high*) *nort*
apply −
apply (*rule heaps-eq-Dag-eq*)
apply (*rule ballI*)
apply (*erule-tac x*=*x* **in** *ballE*)
apply (*elim conjE*)
apply (*rule conjI*)
apply *auto*
done
with *nort-repbc-dag repp-nc* **show**
Dag (*repb p*) (*repb* ∝ *low*) (*repb* ∝ *high*) *t* =
Dag (*repc p*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t*
apply *auto*
apply (*rotate-tac 2*)
apply (*frule-tac Dag-unique*)
apply (*rotate-tac 1*)
apply *simp*
apply *simp*
apply (*frule Dag-unique*)
apply (*rotate-tac 3*)
apply *simp*
apply *simp*
done
qed
from *rep-prop* **have** *repbc-changes*: ∀ *no*∈*set* (*ll* ! *n*).
repc no ≠ *Null* ∧
(*if* (*repc* ∝ *low*) *no* = (*repc* ∝ *high*) *no* ∧ *low no* ≠ *Null*
then repc no = (*repc* ∝ *low*) *no*
else repc no ∈ *set* (*ll* ! *n*) ∧ *repc* (*repc no*) = *repc no* ∧
(∀ *no1*∈*set* (*ll* ! *n*). ((*repc* ∝ *high*) *no1* = (*repc* ∝ *high*) *no* ∧
(*repc* ∝ *low*) *no1* = (*repc* ∝ *low*) *no*) = (*repc no* = *repc no1*)))
by *blast*
from *nsll lll* **have** *n-var-prop*: *n* + *1* <= *var p* + *1*
by *simp*
from *rep-nc* **have** *Sucn-repb-nc*: (∀ *pt*. *pt* ∉ *set-of pret* ∨
(∃ *i*. *n* + *1* ≤ *i* ∧ *pt* ∈ *set* (*ll* ! *i*) ∧ *i* < *var p* + *1*)
⟶ *rep pt* = *repb pt*)

```
  apply −
  apply (intro allI impI)
  apply (erule-tac x=pt in allE)
  apply auto
  apply (rule-tac x=i in exI)
  apply auto
  done
have repc-nc:
  (∀ pt. pt ∉ set-of pret ∨
  (∃ i. n + 1 ≤ i ∧ pt ∈ set (ll ! i) ∧ i < var p + 1)
  ⟶ rep pt = repc pt)
proof (intro allI impI)
  fix pt
  assume pt-notin-lower-ll: pt ∉ set-of pret ∨
    (∃ i. n + 1 ≤ i ∧ pt ∈ set (ll ! i) ∧ i < var p + 1)
  show rep pt = repc pt
  proof (cases pt ∉ set-of pret)
    case True
    with wf-ll nsll have pt ∉ set (ll ! n)
      apply (simp add: wf-ll-def length-ll-eq)
      apply (case-tac pt ∈ set (ll ! n))
      apply (subgoal-tac pt ∈ set-of pret)
      apply (auto)
      done
    with repbc-nc have repb pt = repc pt
      by auto
    with Sucn-repb-nc True show ?thesis
      by auto
  next
    assume pt-in-pret: ¬ pt ∉ set-of pret
    with pt-notin-lower-ll have pt-in-higher-ll:
      ∃ i. n + 1 ≤ i ∧ pt ∈ set (ll ! i) ∧ i < var p + 1
      by simp
    with nsll wf-ll lll have pt-notin-lln: pt ∉ set (ll ! n)
      apply −
      apply (erule exE)
      apply (rule-tac i=i and j=n in no-in-one-ll)
      apply (auto simp add: length-ll-eq)
      done
    with repbc-nc have repb pt = repc pt
      by auto
    with Sucn-repb-nc pt-in-higher-ll show ?thesis
      by auto
  qed
qed
from wf-ll nsll
have Nodesn-notin-lln: ∀ no ∈ Nodes n ll. no ∉ set (ll ! n)
  apply (simp add: Nodes-def)
  apply clarify
```

```
  apply (drule no-in-one-ll)
  apply (auto simp add: length-ll-eq)
  done
with repbc-nc
have Nodesn-repnc: ∀ no ∈ Nodes n ll. repb no = repc no
  apply −
  apply (rule ballI)
  apply (erule-tac x=no in allE)
  apply simp
  done
then have repbNodes-repcNodes:
  repb `(Nodes n ll) = repc `(Nodes n ll)
  apply −
  apply rule
  apply blast
  apply rule
  apply (erule imageE)
  apply (erule-tac x=xa in ballE)
  prefer 2
  apply simp
  apply rule
  apply auto
  done
have repcNodes-in-Nodes:
  repc ` Nodes (n + 1) ll ⊆ Nodes (n + 1) ll
proof
  fix x
  assume x-in-repcNodesSucn:  x ∈ repc ` Nodes (n + 1) ll
  show x ∈ Nodes (n + 1) ll
  proof (cases x ∈ repc `Nodes n ll)
    case True
    with repbNodes-repcNodes repbNodes-in-Nodes have x ∈ Nodes n ll
      by auto
    with Nodes-subset show ?thesis
      by auto
  next
    assume  x ∉ repc `Nodes n ll
    with x-in-repcNodesSucn have x-in-repclln: x ∈ repc `set (ll ! n)
      apply (auto simp add: Nodes-def)
      apply (case-tac k<n)
      apply auto
      apply (case-tac k = n)
      apply simp
      apply arith
      done
    from x-in-repclln show ?thesis
    proof (elim imageE)
      fix y
      assume x-repcy: x = repc y
```

assume *y-in-repclln*: y ∈ set (ll ! n)
from *rep-prop y-in-repclln* **obtain**
  repcy-nNull: repc y ≠ Null **and**
  red-prop: (repc ∝ low) y = (repc ∝ high) y ∧
  low y ≠ Null ⟶ repc y = (repc ∝ high) y **and**
  share-prop: ((repc ∝ low) y = (repc ∝ high) y ⟶ low y = Null)
  ⟶ repc y ∈ set (ll ! n) ∧ repc (repc y) = repc y ∧
  (∀ no1∈set (ll ! n).
  ((repc ∝ high) no1 = (repc ∝ high) y ∧
  (repc ∝ low) no1 = (repc ∝ low) y) = (repc y = repc no1))
  using [[simp-depth-limit = 4]]
  by *auto*
from *wf-ll nsll y-in-repclln* **obtain**
  y-in-pret: y ∈ set-of pret **and**
  vary-n: var y = n
  by (*auto simp add*: *wf-ll-def length-ll-eq*)
from *y-in-pret pret-dag* **have** *y-nNull*: y ≠ Null
  apply −
  apply (*rule set-of-nn*)
  apply *auto*
  done
show x ∈ Nodes (n + 1) ll
proof (*cases* low y = Null)
  case *True*
  from *pret-dag prebdt-pret True y-in-pret*
  have *highy-Null*: high y = Null
    apply −
    apply (*drule balanced-bdt*)
    apply *auto*
    done
  with *share-prop True* **obtain**
    repcy-in-llb: repc y ∈ set (ll ! n) **and**
    rry-ry: repc (repc y) = repc y **and**
    y-other-node-prop: ∀ no1∈set (ll ! n).
    ((repc ∝ high) no1 = (repc ∝ high) y ∧
    (repc ∝ low) no1 = (repc ∝ low) y) = (repc y = repc no1)
    by *simp*
  from *repcy-in-llb x-repcy* **show** *?thesis*
    by (*auto simp add*: *Nodes-def*)
next
  assume *lowy-nNull*: low y ≠ Null
  with *pret-dag prebdt-pret y-in-pret*
  have *highy-nNull*: high y ≠ Null
    apply −
    apply (*drule balanced-bdt*)
    apply *auto*
    done
  show *?thesis*
  proof (*cases* (repc ∝ low) y = (repc ∝ high) y)

```
        case True
        with red-prop lowy-nNull have repc y = (repc ∝ high) y
          by auto
        with highy-nNull have red-repc-y: repc y = repc (high y)
          by (simp add: null-comp-def)
        from pret-dag ord-pret y-in-pret lowy-nNull  highy-nNull

        have var (low y) < var y ∧ var (high y) < var y
          apply −
          apply (rule var-ordered-children)
          apply auto
          done
        with  vary-n have varhighy: var (high y) < n
          by auto
        from y-in-pret y-nNull highy-nNull pret-dag
        have high y ∈ set-of pret
          apply −
          apply (drule subelem-set-of-high)
          apply auto
          done
        with wf-ll varhighy have high y ∈ Nodes n ll
          by (auto simp add: wf-ll-def Nodes-def)
        with red-repc-y have repc y ∈ repc ‘Nodes n ll
          by simp
        with x-repcy have x ∈ repc ‘Nodes n ll
          by simp
        with repbNodes-repcNodes repbNodes-in-Nodes
        have x ∈ Nodes n ll
          by auto
        with Nodes-subset show ?thesis
          by auto
      next
        assume (repc ∝ low) y ≠ (repc ∝ high) y
        with share-prop obtain
          repcy-in-llbn: repc y ∈ set (ll ! n) and
          rry-ry: repc (repc y) = repc y and
          y-other-node-share: ∀ no1∈set (ll ! n).
          ((repc ∝ high) no1 = (repc ∝ high) y ∧
          (repc ∝ low) no1 = (repc ∝ low) y) = (repc y = repc no1)
          by auto
        with repcy-in-llbn  x-repcy have x ∈ set (ll ! n)
          by auto
        then show ?thesis
          by (auto simp add: Nodes-def)
      qed
    qed
  qed
 qed
qed
```

have (∀ *no*∈*Nodes* (*n* + *1*) *ll*.
  var (*repc no*) ≤ *var no* ∧
  (∃ *not nort*. *Dag* (*repc no*) (*repc* ∝ *low*) (*repc* ∝ *high*) *nort* ∧
  Dag no low high not ∧
  reduced nort ∧ *ordered nort var* ∧
  set-of nort ⊆ *repc* ' *Nodes* (*n* + *1*) *ll* ∧
  (∀ *no*∈*set-of nort*. *repc no* = *no*) ∧
  (∃ *nobdt*. *bdt not var* = *Some nobdt* ∧
  (∃ *norbdt*. *bdt nort var* = *Some norbdt* ∧ *nobdt* ∼ *norbdt*))))
  (**is** ∀ *no*∈*Nodes* (*n* + *1*) *ll*. *?Q i no*)
proof (*intro ballI*)
  fix *no*
  assume *no-in-Nodes*: *no* ∈ *Nodes* (*n* + *1*) *ll*
  from *wf-ll no-in-Nodes nsll* **have** *no-in-pret*: *no* ∈ *set-of pret*
    apply (*simp add*: *wf-ll-def Nodes-def length-ll-eq*)
    apply (*erule conjE*)
    apply (*thin-tac* ∀ *q*. *q* ∈ *set-of pret* ⟶ *q* ∈ *set* (*ll* ! *var q*))
    apply (*erule exE*)
    apply (*erule-tac x*=*x* **in** *allE*)
    apply (*erule impE*)
    apply *arith*
    apply (*erule-tac x*=*no* **in** *ballE*)
    apply *auto*
    done
  from *pret-dag no-in-pret* **have** *nonNull*: *no*≠ *Null*
    apply −
    apply (*rule set-of-nn* [*rule-format*])
    apply *auto*
    done
  show *?Q i no*
  proof (*cases no* ∈ *Nodes n ll*)
    case *True*
    note *no-in-Nodesn*=*this*
    with *wf-ll nsll no-in-Nodes*
    have *no-notin-llbn*: *no* ∉ *set* (*ll* ! *n*)
      apply −
      apply (*simp add*: *Nodes-def length-ll-eq*)
      apply (*elim exE*)
      apply (*drule-tac ?i*=*xa* **and** *?j*=*n* **in** *no-in-one-ll*)
      apply *arith*
      apply *simp*
      apply *auto*
      done
    with *repbc-nc* **have** *repb-no-eq-repc-no*: *repb no* = *repc no*
      by *simp*
    from *repbc-nc no-in-Nodes no-notin-llbn normalize-prop True*
    have *varrep-eq-var*: *var* (*repc no*) ≤ *var no*
      apply −
      apply (*erule-tac x*=*no* **in** *ballE*)

prefer *2*
apply *simp*
apply (*erule-tac x=no* **in** *allE*)
apply *simp*
done
moreover
from *True normalize-prop no-in-Nodes* **obtain** *not nort* **where**
nort-dag: *Dag* (*repb no*) (*repb* $\propto$ *low*) (*repb* $\propto$ *high*) *nort* **and**
ord-nort: *ordered nort var* **and**
subset-nort-not: *set-of nort* $\subseteq$ *repb* ‘(*Nodes n ll*) **and**
not-dag: *Dag no low high not* **and**
red-nort: *reduced nort* **and**
nort-repb: (∀ *no*∈*set-of nort*. *repb no* = *no*) **and**
bdt-prop: ∃ *nobdt norbdt*. *bdt not var* = *Some nobdt* ∧
bdt nort var = *Some norbdt* ∧ *nobdt* ∼ *norbdt*
by *blast*
moreover
from *Nodesn-notin-lln repbc-nc nort-repb subset-nort-not repbN-odes-in-Nodes*
have *nort-repc*:
(∀ *no*∈*set-of nort*. *repc no* = *no*)
apply *auto*
apply (*subgoal-tac no* ∈ *Nodes n ll*)
apply *fastforce*
apply *blast*
done
moreover
from *nort-dag* **have** *nortnodesnN*: (∀ *no*. *no* ∈ *set-of nort* ⟶ *no* ≠ *Null*)
apply −
apply (*rule allI*)
apply (*rule impI*)
apply (*rule set-of-nn*)
apply *auto*
done
moreover
have *Dag* (*repc no*) (*repc* $\propto$ *low*) (*repc* $\propto$ *high*) *nort*
proof −
from *no-notin-llbn repbc-nc* **have** *repbc-no*: *repc no* = *repb no*
by *fastforce*
with *nort-dag*
have *nortrepbc-dag*: *Dag* (*repc no*) (*repb* $\propto$ *low*) (*repb* $\propto$ *high*) *nort*
by *simp*
from *wf-ll nseqll* **have** *Nodes n ll* $\subseteq$ *set-of pret*
apply −
apply (*rule Nodes-levellist-subset-t*)
apply *assumption+*
apply (*simp add*: *length-ll-eq*)
done

with *repbNodes-in-Nodes subset-nort-not*
have *subset-nort-pret*: *set-of nort* ⊆ *set-of pret*
by *simp*
have *vxsn-in-pret*: ∀ *x* ∈ *set-of nort*. *var x* < *n* ∧ *x* ∈ *set-of pret*
proof (*rule ballI*)
fix *x*
assume *x-in-nort*: *x* ∈ *set-of nort*
from *x-in-nort subset-nort-not repbNodes-in-Nodes*
have *x* ∈ *Nodes n ll*
by *blast*
with *wf-ll nsll* **have** *xsn*: *var x* < *n*
apply (*simp add*: *wf-ll-def Nodes-def length-ll-eq*)
apply (*erule conjE*)
apply (*thin-tac* ∀ *q*. *q* ∈ *set-of pret* ⟶ *q* ∈ *set* (*ll* ! *var q*))
apply (*erule exE*)
apply (*erule-tac x=xa* **in** *allE*)
apply (*erule impE*)
apply *arith*
apply (*erule-tac x=x* **in** *ballE*)
apply *auto*
done
from *x-in-nort subset-nort-pret* **have** *x-in-pret*: *x* ∈ *set-of pret*
by *blast*
with *xsn* **show** *var x* < *n* ∧ *x* ∈ *set-of pret* **by** *simp*
qed
with *pret-dag prebdt-pret nortrepbc-dag ord-pret wf-ll nsll*
repbc-nc
have ∀ *x* ∈ *set-of nort*. ((*repc* ∝ *low*) *x* = (*repb* ∝ *low*) *x* ∧
(*repc* ∝ *high*) *x* = (*repb* ∝ *high*) *x*)
apply −
apply (*rule nort-null-comp*)
apply (*auto simp add*: *length-ll-eq*)
done
with *nort-dag*
have *Dag* (*repc no*) (*repc* ∝ *low*) (*repc* ∝ *high*) *nort* =
Dag (*repc no*) (*repb* ∝ *low*) (*repb* ∝ *high*) *nort*
apply −
apply (*rule heaps-eq-Dag-eq*)
apply *simp*
done
with *nortrepbc-dag* **show** *?thesis*
by *simp*
qed
moreover
have *set-of nort* ⊆ *repc* ‘(*Nodes* (*n* + *1*) *ll*)
proof −
have *Nodesn-in-NodesSucn*: *Nodes n ll* ⊆ *Nodes* (*n* + *1*) *ll*
by (*simp add*: *Nodes-def set-split*)
then have *repbNodesn-in-repbNodesSucn*:

repb ‘(*Nodes n ll*) ⊆ *repb* ‘(*Nodes* (*n* + *1*) *ll*)
by *blast*
from *wf-ll nsll*
have *Nodes-n-notin-lln*: ∀ *no* ∈ *Nodes n ll*. *no* ∉ *set* (*ll* ! *n*)
apply (*simp add*: *Nodes-def length-ll-eq*)
apply *clarify*
apply (*drule no-in-one-ll*)
apply *auto*
done
with *repbc-nc* **have** ∀ *no* ∈ *Nodes n ll*. *repb no* = *repc no*
apply −
apply (*rule ballI*)
apply (*erule-tac x*=*no* **in** *allE*)
apply *simp*
done
then have *repbNodes-repcNodes*:
repb ‘(*Nodes n ll*) = *repc* ‘(*Nodes n ll*)
apply −
apply *rule*
apply *blast*
apply *rule*
apply (*erule imageE*)
apply (*erule-tac x*=*xa* **in** *ballE*)
prefer *2*
apply *simp*
apply *rule*
apply *auto*
done
from *Nodesn-in-NodesSucn*
have *repc* ‘(*Nodes n ll*) ⊆ *repc* ‘(*Nodes* (*n* + *1*) *ll*)
by *blast*
with *repbNodes-repcNodes subset-nort-not repbNodesn-in-repbNodesSucn*

show *?thesis* **by** *simp*
qed
ultimately show *?thesis*
by *blast*
next
assume *no* ∉ *Nodes n ll*
with *no-in-Nodes* **have** *no-in-llbn*: *no* ∈ *set* (*ll* ! *n*)
apply (*simp add*: *Nodes-def*)
apply (*erule exE*)
apply (*erule-tac x*=*x* **in** *allE*)
apply (*case-tac x*<*n*)
apply *simp*
apply *simp*
apply (*elim conjE*)
apply (*case-tac x*=*n*)
apply *simp*

apply *arith*
done
with *wf-ll nsll* **have** *varno*: *var no* = *n*
by (*simp add*: *wf-ll-def length-ll-eq*)
from *repbc-changes no-in-llbn*
have *repbcno-changes*: *repc no* ≠ *Null* ∧
((*repc* ∝ *low*) *no* = (*repc* ∝ *high*) *no* ∧ *low no* ≠ *Null*
⟶ *repc no* = (*repc* ∝ *high*) *no*) ∧
(((*repc* ∝ *low*) *no* = (*repc* ∝ *high*) *no* ⟶ *low no* = *Null*)
⟶ *repc no* ∈ *set* (*ll* ! *n*) ∧ *repc* (*repc no*) = *repc no* ∧
(∀ *no1*∈*set* (*ll* ! *n*). ((*repc* ∝ *high*) *no1* = (*repc* ∝ *high*) *no* ∧
(*repc* ∝ *low*) *no1* = (*repc* ∝ *low*) *no*) = (*repc no* = *repc no1*)))
(**is** *?rnonN* ∧ *?repreduce* ∧ *?repshare*)
using [[*simp-depth-limit=4*]]
by (*simp split*: *if-split*)
then obtain
rnonN: *?rnonN* **and**
repreduce: *?repreduce* **and**
repshare: *?repshare*
by *blast*
have *repcn-normalize*: *var* (*repc no*) ≤ *var no* ∧
(∃ *not nort*. *Dag* (*repc no*) (*repc* ∝ *low*) (*repc* ∝ *high*) *nort* ∧
Dag no low high not ∧ *reduced nort* ∧ *ordered nort var* ∧
set-of nort ⊆ *repc* ‘ *Nodes* (*n* + *1*) *ll* ∧
(∀ *no*∈*set-of nort*. *repc no* = *no*) ∧
(∃ *nobdt*. *bdt not var* = *Some nobdt* ∧
(∃ *norbdt*. *bdt nort var* = *Some norbdt* ∧ *nobdt* ∼ *norbdt*)))
(**is** *?varrep* ∧ *?repcn-prop*
is *?varrep* ∧
(∃ *not nort*. *?nort-dag nort* ∧ *?not-dag not* ∧ *?red nort* ∧
?ord nort ∧ *?nort-in-Nodes nort* ∧ *?repcno-no-n nort* ∧ *?bdt-equ not*
nort))
proof (*cases high no* = *Null*)
case *True*
note *highnoNull=this*
with *pret-dag prebdt-pret no-in-pret*
have *lownoNull*: *low no* = *Null*
apply −
apply (*drule balanced-bdt*)
apply *assumption+*
apply *simp*
done
with *repshare* **have** *repcnoinlln*:*repc no* ∈ *set* (*ll* ! *n*)
by *simp*
with *wf-ll nsll* **have** *varrno-n*: *var* (*repc no*) = *n*
by (*simp add*: *wf-ll-def length-ll-eq*)
with *varno* **have** *varrep*: *?varrep*
by *simp*
from *wf-ll nsll no-in-llbn varrno-n*

have *varrno-varno*: *var* (*repc no*) = *var no*
  by (*simp add*: *wf-ll-def length-ll-eq*)
from *wf-ll nsll repcnoinlln*
have *rno-in-pret*: *repc no* ∈ *set-of pret*
  by (*simp add*: *wf-ll-def length-ll-eq*)
from *repcnoinlln repshare lownoNull*
have *reprep-eq-rep*: *repc* (*repc no*) = *repc no*
  by *simp*
with *repcnoinlln repshare lownoNull*
have *repchildreneq*:
  ((*repc* ∝ *high*) (*repc no*) = (*repc* ∝ *high*) *no* ∧
  (*repc* ∝ *low*) (*repc no*) = (*repc* ∝ *low*) *no*)
  by *simp*
have *repcn-prop*: *?repcn-prop*
  apply −
  apply (*rule-tac x*=(*Node Tip no Tip*) **in** *exI*)
  apply (*rule-tac x*=(*Node Tip* (*repc no*) *Tip*) **in** *exI*)
  apply (*intro conjI*)
  apply *simp*
  prefer *3*
  apply *simp*
  prefer *3*
  apply *simp*
proof −
  from *pret-dag pnN rno-in-pret* **have** *rnonN*: *repc no* ≠ *Null*
    apply (*case-tac repc no* = *Null*)
    apply *auto*
    done
  from *highnoNull repchildreneq*
  have *rhighNull*: (*repc* ∝ *high*) (*repc no*) = *Null*
    by (*simp add*: *null-comp-def*)
  from *lownoNull repchildreneq*
  have *rlowNull*: (*repc* ∝ *low*) (*repc no*) = *Null*
    by (*simp add*: *null-comp-def*)
  with *rhighNull rnonN*
  show *repc no* ≠ *Null* ∧ (*repc* ∝ *low*) (*repc no*) = *Null* ∧
    (*repc* ∝ *high*) (*repc no*) = *Null*
    by *simp*
next
  from *nonNull lownoNull highnoNull*
  show *?not-dag* (*Node Tip no Tip*)
    by *simp*
next
  from *no-in-Nodes*
  show *set-of* (*Node Tip* (*repc no*) *Tip*) ⊆ *repc* ‘ *Nodes* (*n* + *1*) *ll*
    by *simp*
next
  show ∀ *no*∈*set-of* (*Node Tip* (*repc no*) *Tip*). *repc no* = *no*
  proof

```
      fix pt
      assume pt-in-repcLeaf: pt ∈ set-of (Node Tip (repc no) Tip)
      with reprep-eq-rep show repc pt = pt
        by simp
    qed
  next
    show ?bdt-equ (Node Tip no Tip) (Node Tip (repc no) Tip)
    proof (cases var no = 0)
      case True
      note vno-Null=this
     then have nobdt: bdt (Node Tip no Tip) var = Some Zero by simp
      from varrep  vno-Null have varrno: var (repc no) = 0 by simp
      then have norbdt: bdt (Node Tip (repc no) Tip) var = Some Zero
by simp
      from nobdt norbdt vno-Null varrno show ?thesis
        by (simp add: cong-eval-def)
    next
      assume vno-not-Null: var no ≠ 0
      show ?thesis
      proof (cases var no = 1)
        case True
        note vno-One=this
      then have nobdt: bdt (Node Tip no Tip) var = Some One by simp
        from varrno-varno vno-One
        have bdt (Node Tip (repc no) Tip) var = Some One by simp
        with nobdt show ?thesis by (auto simp add: cong-eval-def)
      next
        assume vno-nOne: var no ≠ 1
        with vno-not-Null have onesvno: 1 < var no by simp
        from nonNull lownoNull highnoNull
        have no-dag: Dag no low high (Node Tip no Tip)
          by simp
         with pret-dag no-in-pret have not-in-pret: (Node Tip no Tip) ≤
pret
          by (metis set-of-subdag)
        with prebdt-pret have ∃ bdt2. bdt (Node Tip no Tip) var = Some
bdt2
          by (metis subbdt-ex)
        with onesvno show ?thesis
          by simp
      qed
    qed
  qed
  with varrep reprep-eq-rep show ?thesis by simp
next
  assume hno-nNull: high no ≠ Null
  with pret-dag prebdt-pret no-in-pret  have lno-nNull: low no ≠ Null
    by (metis balanced-bdt)
```

from *no-in-pret nonNull hno-nNull pret-dag*
have *hno-in-pret*: *high no* ∈ *set-of pret*
  by (*metis subelem-set-of-high*)
with *wf-ll*
have *hno-in-ll*: *high no* ∈ *set* (*ll* ! (*var* (*high no*)))
  by (*simp add*: *wf-ll-def*)
from *pret-dag ord-pret no-in-pret lno-nNull hno-nNull*

have *varhnos-varno*: *var* (*high no*) < *var no*
  by (*metis var-ordered-children*)
with *varno* **have** *varhnos-n*: *var* (*high no*) < *n* **by** *simp*
with *hno-in-ll* **have** *hno-in-Nodesn*: *high no* ∈ *Nodes n ll*
  apply (*simp add*: *Nodes-def*)
  apply (*rule-tac x=var* (*high no*) **in** *exI*)
  apply *simp*
  done
from *wf-ll nsll hno-in-ll   varhnos-n*
have *high no* ∉ *set* (*ll* ! *n*)
  apply −
  apply (*rule no-in-one-ll*)
  apply (*auto simp add*: *length-ll-eq*)
  done
with *repbc-nc* **have** *repb-repc-high*: *repb* (*high no*) = *repc* (*high no*) **by**
simp
with *normalize-prop hno-in-Nodesn varhnos-varno varno*
have *high-normalize*: *var* (*repc* (*high no*)) ≤ *var* (*high no*) ∧
  (∃ *not nort*. *Dag* (*repc* (*high no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *nort* ∧
  Dag (*high no*) *low high not* ∧ *reduced nort* ∧
  ordered nort var ∧ *set-of nort* ⊆ *repb* '(*Nodes n ll*) ∧
  (∀ *no*∈*set-of nort*. *repb no* = *no*) ∧
  (∃ *nobdt norbdt*. *bdt not var* = *Some nobdt* ∧ *bdt nort var* =
  Some norbdt ∧ *nobdt* ∼ *norbdt*))
  (**is** *?varrep-high* ∧
    (∃ *not nort*. *?repbchigh-dag nort* ∧ *?high-dag not* ∧
    ?redhigh nort ∧ *?ordhigh nort* ∧ *?rephigh-in-Nodes nort* ∧
    ?repbno-no nort ∧ *?highdd-prop not nort*)
    is *?varrep-high* ∧ *?not-nort-prop*)
  apply *simp*
  apply (*erule-tac x=high no* **in** *ballE*)
  apply (*simp del*: *Dag-Ref*)
  apply *simp*
  done
then have *varrep-high*: *?varrep-high* **by** *simp*
from *varhnos-n varrep-high* **have** *varrephno-s-n*:
  var (*repc* (*high no*)) < *n*
  by *simp*
from *Nodes-subset*
have *Nodes n ll* ⊆ *Nodes* (*Suc n*) *ll*

by *auto*
with *hno-in-Nodesn repcNodes-in-Nodes*
have *repc* (*high no*) ∈ *Nodes* (*Suc n*) *ll*
apply *simp*
apply *blast*
done
with *wf-ll nsll* **have** *repc* (*high no*) ∈ *set-of pret*
apply (*simp add*: *wf-ll-def Nodes-def length-ll-eq*)
apply (*elim conjE exE*)
apply (*thin-tac* ∀ *q*. *q* ∈ *set-of pret* ⟶ *q* ∈ *set* (*ll* ! *var q*))
apply (*erule-tac x*=*x* **in** *allE*)
apply (*erule impE*)
apply *simp*
apply (*erule-tac x*=*repc* (*high no*) **in** *ballE*)
apply *auto*
done
with *wf-ll varrephno-s-n*
have ∃ *k*<*n*. *repc* (*high no*) ∈ *set* (*ll* ! *k*)
by (*auto simp add*: *wf-ll-def*)
with *wf-ll nsll* **have** *repc* (*high no*) ∉ *set* (*ll* ! *n*)
apply −
apply (*erule exE*)
apply (*rule-tac i*=*k* **and** *j*=*n* **in** *no-in-one-ll*)
apply (*auto simp add*: *length-ll-eq*)
done
with *repbc-nc*
have *repbchigh-idem*: *repb* (*repc* (*high no*)) = *repc* (*repc* (*high no*))
by *auto*
from *high-normalize*
have *not-nort-prop-high*: *?not-nort-prop* **by** (*simp del*: *Dag-Ref*)
from *not-nort-prop-high* **obtain** *hnot* **where** *high-dag*: *?high-dag hnot*
by *auto*
from *wf-ll nsll*
have ∀ *no* ∈ *Nodes n ll*. *no* ∉ *set* (*ll* ! *n*)
apply (*simp add*: *Nodes-def length-ll-eq*)
apply *clarify*
apply (*drule no-in-one-ll*)
apply *auto*
done
with *repbc-nc* **have** ∀ *no* ∈ *Nodes n ll*. *repb no* = *repc no*
apply −
apply (*rule ballI*)
apply (*erule-tac x*=*no* **in** *allE*)
apply *simp*
done
then
have *repbNodes-repcNodes*:
repb ‘(*Nodes n ll*) = *repc* ‘(*Nodes n ll*)
apply −

apply *rule*
apply *blast*
apply *rule*
apply (*erule imageE*)
apply (*erule-tac x=xa* **in** *ballE*)
prefer *2*
apply *simp*
apply *rule*
apply *auto*
done
then have *repcNodes-repbNodes*:
repc ‘(*Nodes n ll*) = *repb* ‘(*Nodes n ll*)
by *simp*
from *pret-dag nsll wf-ll*
have *null-notin-Nodesn*: *Null* ∉ *Nodes n ll*
apply −
apply (*rule Null-notin-Nodes*)
apply (*auto simp add*: *length-ll-eq*)
done
from *hno-in-Nodesn* **have** *repc* (*high no*) ∈ *repc* ‘(*Nodes n ll*)
by *blast*
with *repbNodes-in-Nodes repcNodes-repbNodes*
have *repc* (*high no*) ∈ *Nodes n ll*
apply *simp*
apply *blast*
done
with *null-notin-Nodesn* **have** *rhn-nNull*: *repc* (*high no*) ≠ *Null*
by *auto*

from *no-in-pret nonNull lno-nNull pret-dag*
have *lno-in-pret*: *low no* ∈ *set-of pret*
by (*rule subelem-set-of-low*)
with *wf-ll*
have *lno-in-ll*: *low no* ∈ *set* (*ll* ! (*var* (*low no*)))
by (*simp add*: *wf-ll-def*)
from *pret-dag ord-pret no-in-pret lno-nNull hno-nNull*
have *varlnos-varno*: *var* (*low no*) < *var no*
apply −
apply (*drule var-ordered-children*)
apply *assumption+*
apply *auto*
done
with *varno* **have** *varlnos-n*: *var* (*low no*) < *n* **by** *simp*
with *lno-in-ll* **have** *lno-in-Nodesn*: *low no* ∈ *Nodes n ll*
apply (*simp add*: *Nodes-def*)
apply (*rule-tac x=var* (*low no*) **in** *exI*)
apply *simp*

```
  done
from varlnos-n wf-ll nsll lno-in-ll
have low no ∉ set (ll ! n)
  apply −
  apply (rule no-in-one-ll)
  apply (auto simp add: length-ll-eq)
  done
 with repbc-nc have repb-repc-low: repb (low no) = repc (low no) by
simp
with normalize-prop lno-in-Nodesn varlnos-varno varno
have low-normalize: var (repc (low no)) ≤ var (low no) ∧
  (∃ not nort. Dag (repc (low no)) (repb ∝ low) (repb ∝ high) nort ∧
  Dag (low no) low high not ∧ reduced nort ∧ ordered nort var ∧
  set-of nort ⊆ repb ‘(Nodes n ll) ∧
  (∀ no∈set-of nort. repb no = no) ∧
    (∃ nobdt norbdt. bdt not var = Some nobdt ∧ bdt nort var = Some
norbdt ∧
  nobdt ∼ norbdt))
  (is ?varrep-low ∧
    (∃ not nort. ?repbclow-dag nort ∧ ?low-dag not ∧ ?redhigh nort ∧
    ?ordhigh nort ∧ ?replow-in-Nodes nort  ∧ ?low-repno-no nort
    ∧ ?lowdd-prop not nort)
    is ?varrep-low ∧ ?not-nort-prop-low)
  apply simp
  apply (erule-tac x=low no in ballE)
  apply (simp del: Dag-Ref)
  apply simp
  done
then have varrep-low: ?varrep-low by simp
from low-normalize have not-nort-prop-low: ?not-nort-prop-low
  by (simp del: Dag-Ref)
from lno-in-Nodesn have repc (low no) ∈ repc ‘(Nodes n ll)
  by blast
with repbNodes-in-Nodes repcNodes-repbNodes
have repc (low no) ∈ Nodes n ll
  apply simp
  apply blast
  done
with null-notin-Nodesn have rln-nNull: repc (low no) ≠ Null
  by auto


show ?thesis
proof (cases repc (low no) = repc (high no))
  case True
  note red-case=this
  with repreduce lno-nNull hno-nNull
  have rno-eq-hrno: repc no = repc (high no)
    by (simp add: null-comp-def)
```

from *varhnos-varno rno-eq-hrno varrep-high* **have** *varrep*: *?varrep* **by** *simp*
from *not-nort-prop-high not-nort-prop-low* **have** *repcn-prop*: *?repcn-prop*
apply −
apply (*elim exE*)
apply (*rename-tac rnot lnot rnort lnort*)
apply (*rule-tac x*=(*Node lnot no rnot*) **in** *exI*)
apply (*rule-tac x*=*rnort* **in** *exI*)
apply (*elim conjE*)
apply (*intro conjI*)
prefer *7*
apply (*elim exE*)
apply (*rename-tac rnot lnot rnort lnort rnobdt lnobdt rnorbdt lnorbdt*)
apply (*elim conjE*)
apply (*case-tac Suc 0* < *var no*)
apply (*rule-tac x*=(*Bdt-Node lnobdt* (*var no*) *rnobdt*) **in** *exI*)
apply (*rule conjI*)
prefer *2*
apply (*rule-tac x*=*rnorbdt* **in** *exI*)
apply (*rule conjI*)
proof −
fix *rnot lnot rnort lnort*
assume *highnort-dag*:
Dag (*repc* (*high no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rnort*
assume *ord-nort*: *ordered rnort var*
assume *rnort-in-repNodes*: *set-of rnort* ⊆ *repb* ‘ *Nodes n ll*
from *rnort-in-repNodes repbNodes-in-Nodes*
have *nort-in-Nodes*: *set-of rnort* ⊆ *Nodes n ll*
by *blast*
from *varhnos-n varrep-high*
have *vrhnos-n*: *var* (*repc* (*high no*)) < *n* **by** *simp*
from *rhn-nNull highnort-dag*
have ∃ *lno rno*. *rnort* = *Node lno* (*repc* (*high no*)) *rno* **by** *fastforce*
with *highnort-dag rhn-nNull* **have** *root rnort* = *repc* (*high no*) **by** *auto*
with *ord-nort* **have** ∀ *x* ∈ *set-of rnort*. *var x* <= *var* (*repc* (*high no*))
apply −
apply (*rule ballI*)
apply (*drule ordered-set-of*)
apply *auto*
done
with *vrhnos-n* **have** *vxsn*: ∀ *x* ∈ *set-of rnort*. *var x* < *n*
by *fastforce*
from *nort-in-Nodes* **have** ∀ *x* ∈ *set-of rnort*. *x* ∈ *Nodes n ll*
by *auto*
with *wf-ll nsll*
have *x-in-pret*: ∀ *x* ∈ *set-of rnort*. *x* ∈ *set-of pret*
apply −

```
    apply (rule ballI)
    apply (drule wf-ll-Nodes-pret)
    apply (auto simp add: length-ll-eq)
    done
  from vxsn x-in-pret
  have vxsn-in-nort: ∀ x ∈ set-of rnort. var x <n ∧ x ∈ set-of pret
    by auto
  with pret-dag prebdt-pret highnort-dag ord-pret wf-ll nsll
    repbc-nc
  have ∀ x ∈ set-of rnort. (repc ∝ low) x = (repb ∝ low) x ∧
    (repc ∝ high) x = (repb ∝ high) x
    apply –
    apply (rule nort-null-comp)
    apply (auto simp add: length-ll-eq)
    done
  with rno-eq-hrno
  have Dag (repc no) (repc ∝ low) (repc ∝ high) rnort =
    Dag (repc no) (repb ∝ low) (repb ∝ high) rnort
    apply –
    apply (rule heaps-eq-Dag-eq)
    apply simp
    done
  with highnort-dag rno-eq-hrno
  show Dag (repc no) (repc ∝ low) (repc ∝ high) rnort by simp
next
  fix rnot lnot rnort lnort
  assume lnot-dag: Dag (low no) low high lnot
  assume rnot-dag: Dag (high no) low high rnot
  with lnot-dag  nonNull
  show Dag no low high (Node lnot no rnot) by simp
next
  fix rnot lnot rnort lnort
  assume  reduced rnort
  then show reduced rnort by simp
next
  fix rnort
  assume ordered rnort var
  then show ordered rnort var by simp
next
  fix rnort
  assume rnort-in-Nodes:  set-of rnort ⊆ repb ‘ Nodes n ll
  have Nodes n ll ⊆ Nodes (n + 1) ll
    by (simp add: Nodes-def set-split)
  then have repc ‘ Nodes n ll ⊆ repc ‘ Nodes (n + 1) ll
    by blast
  with rnort-in-Nodes repbNodes-repcNodes
  show  set-of rnort ⊆ repc ‘ Nodes (n + 1) ll
    by (simp add: Nodes-def)
next
```

fix *rnort rnorbdt*
assume *bdt rnort var = Some rnorbdt*
then show *bdt rnort var = Some rnorbdt* **by** *simp*
next
fix *rnot lnot rnort lnort rnobdt lnobdt rnorbdt lnorbdt*
assume *rcongeval*: *rnobdt ∼ rnorbdt*
assume *lnort-dag*:
Dag (*repc* (*low no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *lnort*
assume *rnort-dag*:
Dag (*repc* (*high no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rnort*
assume *lnorbdt-def*: *bdt lnort var = Some lnorbdt*
assume *rnorbdt-def*: *bdt rnort var = Some rnorbdt*
assume *lcongeval*:*lnobdt ∼ lnorbdt*
from *red-case lnort-dag rnort-dag*
have *lnort-rnort*: *lnort = rnort*
by (*simp add*: *Dag-unique del*: *Dag-Ref*)
with *lnorbdt-def lcongeval rnorbdt-def*
have *lnobdt-rnorbdt*: *lnobdt ∼ rnorbdt* **by** *simp*
with *rcongeval* **have** *lnobdt ∼ rnobdt*
apply −
apply (*rule cong-eval-trans*)
apply (*auto simp add*: *cong-eval-sym*)
done
then have *Bdt-Node lnobdt* (*var no*) *rnobdt ∼ rnobdt*
apply −
apply (*simp add*: *cong-eval-sym* [*rule-format*])
apply (*rule cong-eval-child-high*)
apply *assumption*
done
with *rcongeval* **show** *Bdt-Node lnobdt* (*var no*) *rnobdt ∼ rnorbdt*
apply −
apply (*rotate-tac 1*)
apply (*rule cong-eval-trans*)
apply *auto*
done
next
fix *lnot rnot lnobdt rnobdt*
assume *lnot-dag*: *Dag* (*low no*) *low high lnot*
assume *rnot-dag*: *Dag* (*high no*) *low high rnot*
assume *lnobdt-def*: *bdt lnot var = Some lnobdt*
assume *rnobdt-def*: *bdt rnot var = Some rnobdt*
assume *onesvarno*: *Suc 0 < var no*
with *rnobdt-def lnot-dag rnot-dag lnobdt-def*
show *bdt* (*Node lnot no rnot*) *var* =
Some (*Bdt-Node lnobdt* (*var no*) *rnobdt*)
by *simp*
next
fix *rnot lnot rnort lnort rnobdt lnobdt rnorbdt lnorbdt*
assume *lnobdt-def*: *bdt lnot var = Some lnobdt*

assume *rnobdt-def*: *bdt rnot var = Some rnobdt*
assume *rnorbdt-def*: *bdt rnort var = Some rnorbdt*
assume *cong-rno-rnor*: *rnobdt ∼ rnorbdt*
assume *lnot-dag*: *Dag (low no) low high lnot*
assume *rnot-dag*: *Dag (high no) low high rnot*
assume ¬ *Suc 0 < var no*
then have *varnoseq1*: *var no = 0 ∨ var no = 1* **by** *auto*
show ∃ *nobdt. bdt (Node lnot no rnot) var = Some nobdt ∧*
 (∃ *norbdt. bdt rnort var = Some norbdt ∧ nobdt ∼ norbdt*)
proof (*cases var no = 0*)
 case *True*
 note *vnoNull=this*
 with *pret-dag ord-pret no-in-pret lno-nNull hno-nNull*
 show *?thesis*
  apply −
  apply (*drule var-ordered-children*)
  apply *auto*
  done
next
 assume *var no ≠ 0*
 with *varnoseq1* **have** *vnoOne*: *var no = 1* **by** *simp*
 from *pret-dag ord-pret no-in-pret lno-nNull hno-nNull*
  vnoOne
 have *vlvrNull*: *var (low no) = 0 ∧ var (high no) = 0*
  apply −
  apply (*drule var-ordered-children*)
  apply *auto*
  done
 then have *vlNull*: *var (low no) = 0* **by** *simp*
 from *vlvrNull* **have** *vrNull*: *var (high no) = 0* **by** *simp*
 from *lnobdt-def lnot-dag vlNull lno-nNull*
 have *lnobdt-Zero*: *lnobdt = Zero*
  apply −
  apply (*drule bdt-Some-var0-Zero*)
  apply *auto*
  done
 from *rnobdt-def rnot-dag vrNull hno-nNull*
 have *rnobdt-Zero*: *rnobdt = Zero*
  apply −
  apply (*drule bdt-Some-var0-Zero*)
  apply *auto*
  done
 from *lnobdt-Zero lnobdt-def* **have** *bdt lnot var = Some Zero* **by**
simp
 with *lnot-dag vlNull*
 have *lnot-Node*: *lnot = (Node Tip (low no) Tip)*
  by *auto*
 from *rnobdt-Zero rnobdt-def rnot-dag vrNull*
 have *rnot-Node*: *rnot = (Node Tip (high no) Tip)*

by *auto*
from *pret-dag no-in-pret* **obtain** *not* **where**
not-ex: *Dag no low high not*
apply −
apply (*drule dag-setof-exD*)
apply *auto*
done
with *pret-dag no-in-pret* **have** *not-ex-in-pret*: *not* <= *pret*
apply −
apply (*rule set-of-subdag*)
apply *auto*
done
from *not-ex lnot-dag rnot-dag nonNull*
have *not-def*: *not* = (*Node lnot no rnot*)
by (*simp add*: *Dag-unique del*: *Dag-Ref*)
with *not-ex-in-pret prebdt-pret*
have *nobdt-ex*: ∃ *nobdt*. *bdt* (*Node lnot no rnot*) *var* = *Some nobdt*
apply −
apply (*rule subbdt-ex*)
apply *auto*
done
then obtain *nobdt* **where**
nobdt-def: *bdt* (*Node lnot no rnot*) *var* = *Some nobdt* **by** *auto*
from *not-def* **have** *root not* = *no* **by** *simp*
with *nobdt-def vnoOne not-def* **have** *not* = (*Node Tip no Tip*)
apply −
apply (*drule bdt-Some-var1-One*)
apply *auto*
done
with *not-def* **have** *rnot* = *Tip* **by** *simp*
with *rnot-Node* **show** *?thesis* **by** *simp*
qed
next
fix *rnot lnot rnort lnort*
assume *rnort-in-repb-Nodesn*: *set-of rnort* ⊆ *repb* ‘ *Nodes n ll*
assume *rnort-repb-no*: ∀ *no*∈*set-of rnort*. *repb no* = *no*
from *repbNodes-in-Nodes rnort-in-repb-Nodesn*
have *rnort-in-Nodesn*: *set-of rnort* ⊆ *Nodes n ll*
by *blast*
show ∀ *no*∈*set-of rnort*. *repc no* = *no*
proof
fix *pt*
assume *pt-in-rnort*: *pt* ∈ *set-of rnort*
with *rnort-in-Nodesn* **have** *pt* ∈ *Nodes n ll*
by *blast*
with *Nodesn-notin-lln* **have** *pt* ∉ *set* (*ll* ! *n*)
by *auto*
with *repbc-nc* **have** *repb pt* = *repc pt*
by *auto*

with *rnort-repb-no pt-in-rnort* **show** *repc pt = pt*
by *auto*
qed
qed
with *varrep* **show** *?thesis* **by** *simp*
next
assume *share-case-cond*: *repc* (*low no*) ≠ *repc* (*high no*)
with *lno-nNull hno-nNull*
have *share-case-cond-propto*: (*repc* ∝ *low*) *no* ≠ (*repc* ∝ *high*) *no*
by (*simp add*: *null-comp-def*)
with *repshare* **obtain**
rno-in-llbn: *repc no* ∈ *set* (*ll* ! *n*) **and**
rrno-eq-rno: *repc* (*repc no*) = *repc no* **and**
twonodes-in-llbn-prop: (∀ *no1*∈*set* (*ll* ! *n*).
((*repc* ∝ *high*) *no1* = (*repc* ∝ *high*) *no* ∧
(*repc* ∝ *low*) *no1* = (*repc* ∝ *low*) *no*) = (*repc no* = *repc no1*))
by *auto*
from *wf-ll rno-in-llbn nsll*
have *varrepno-n*: *var* (*repc no*) = *n*
by (*simp add*: *wf-ll-def length-ll-eq*)
with *varno* **have** *varrep*: *?varrep*
by *simp*
from *not-nort-prop-high not-nort-prop-low* **have** *repcn-prop*: *?repcn-prop*
apply−
apply (*elim exE*)
apply (*rename-tac rnot lnot rnort lnort*)
apply (*rule-tac x=Node lnot no rnot* **in** *exI*)
apply (*rule-tac x=Node lnort* (*repc no*) *rnort* **in** *exI*)
apply (*elim conjE*)
apply (*intro conjI*)
prefer *7*
apply (*elim exE*)
apply (*rename-tac rnot lnot rnort lnort rnobdt lnobdt rnorbdt lnorbdt*)
apply (*elim conjE*)
apply (*case-tac Suc 0 < var no*)
apply (*rule-tac x=*(*Bdt-Node lnobdt* (*var no*) *rnobdt*) **in** *exI*)
apply (*rule conjI*)
prefer *2*
apply (*rule-tac x=*(*Bdt-Node lnorbdt* (*var* (*repc no*)) *rnorbdt*) **in**
exI)
apply (*rule conjI*)
proof −
fix *rnot lnot rnort lnort*
assume *rnort-dag*:
Dag (*repc* (*high no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rnort*
assume *lnort-dag*:
Dag (*repc* (*low no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *lnort*
assume *rnort-in-repNodes*: *set-of rnort* ⊆ *repb* ‘ *Nodes n ll*
assume *lnort-in-repNodes*: *set-of lnort* ⊆ *repb* ‘ *Nodes n ll*

from *rnort-in-repNodes repbNodes-in-Nodes*
have *rnort-in-Nodes*: *set-of rnort* ⊆ *Nodes n ll*
  by *simp*
from *lnort-in-repNodes repbNodes-in-Nodes*
have *lnort-in-Nodes*: *set-of lnort* ⊆ *Nodes n ll*
  by *simp*
from *rnort-in-Nodes*
have *rnortx-in-Nodes*: ∀ *x* ∈ *set-of rnort*. *x* ∈ *Nodes n ll*
  by *blast*
with *wf-ll nsll*
have ∀ *x* ∈ *set-of rnort*. *x* ∈ *set-of pret* ∧ *var x* < *n*
  apply −
  apply (*rule ballI*)
  apply (*rule wf-ll-Nodes-pret*)
  apply (*auto simp add*: *length-ll-eq*)
  done
with *pret-dag prebdt-pret rnort-dag ord-pret wf-ll nsll*
  repbc-nc
have ∀ *x* ∈ *set-of rnort*. (*repc* ∝ *low*) *x* = (*repb* ∝ *low*) *x* ∧
  (*repc* ∝ *high*) *x* = (*repb* ∝ *high*) *x*
  apply −
  apply (*rule nort-null-comp*)
  apply (*auto simp add*: *length-ll-eq*)
  done
then have *Dag* (*repc* (*high no*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *rnort* =
  Dag (*repc* (*high no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rnort*
  apply −
  apply (*rule heaps-eq-Dag-eq*)
  apply *assumption*
  done
with *rnort-dag*
have *rnort-dag-repc*:
  Dag (*repc* (*high no*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *rnort*
  by *simp*
from *lnort-in-Nodes*
have *lnortx-in-Nodes*: ∀ *x* ∈ *set-of lnort*. *x* ∈ *Nodes n ll*
  by *blast*
with *wf-ll nsll*
have ∀ *x* ∈ *set-of lnort*. *x* ∈ *set-of pret* ∧ *var x* < *n*
  apply −
  apply (*rule ballI*)
  apply (*rule wf-ll-Nodes-pret*)
  apply (*auto simp add*: *length-ll-eq*)
  done
with *pret-dag prebdt-pret lnort-dag ord-pret wf-ll nsll*
  repbc-nc
have ∀ *x* ∈ *set-of lnort*. (*repc* ∝ *low*) *x* = (*repb* ∝ *low*) *x* ∧
  (*repc* ∝ *high*) *x* = (*repb* ∝ *high*) *x*
  apply −

apply (*rule nort-null-comp*)
apply (*auto simp add*: *length-ll-eq*)
done
then have
Dag (*repc* (*low no*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *lnort* =
Dag (*repc* (*low no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *lnort*
apply −
apply (*rule heaps-eq-Dag-eq*)
apply *assumption*
done
with *lnort-dag*
have *lnort-dag-repc*:
Dag (*repc* (*low no*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *lnort*
by *simp*
from *lno-nNull hno-nNull*
have *propto-comp*: (*repc* ∝ *low*) *no* = *repc* (*low no*) ∧
(*repc* ∝ *high*) *no* = *repc* (*high no*)
by (*simp add*: *null-comp-def*)
from *rno-in-llbn twonodes-in-llbn-prop rrno-eq-rno*
have (*repc* ∝ *high*) (*repc no*) = (*repc* ∝ *high*) *no* ∧
(*repc* ∝ *low*) (*repc no*) = (*repc* ∝ *low*) *no*
by *simp*
with *propto-comp lnort-dag-repc rnort-dag-repc lno-nNull hno-nNull*

rnonN
show *Dag*(*repc no*)(*repc* ∝ *low*)(*repc* ∝ *high*)(*Node lnort* (*repc no*)
rnort)
by *auto*
next
fix *rnot lnot rnort lnort*
assume *rnot-dag*: *Dag* (*high no*) *low high rnot*
assume *lnot-dag*: *Dag* (*low no*) *low high lnot*
with *rnot-dag nonNull*
show *Dag no low high* (*Node lnot no rnot*)
by *simp*
next
fix *rnort lnort*
assume *rnort-dag*:
Dag (*repc* (*high no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rnort*
assume *lnort-dag*:
Dag (*repc* (*low no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *lnort*
assume *red-rnort*: *reduced rnort*
assume *red-lnort*: *reduced lnort*
from *rhn-nNull rnort-dag* **obtain** *lrnort rrnort* **where**
rnort-Node: *rnort* = (*Node lrnort* (*repc* (*high no*)) *rrnort*)
by *auto*
from *rln-nNull lnort-dag* **obtain** *llnort rlnort* **where**
lnort-Node: *lnort* = (*Node llnort* (*repc* (*low no*)) *rlnort*)
by *auto*

from *twonodes-in-llbn-prop rrno-eq-rno rno-in-llbn hno-nNull lno-nNull*
have ((*repc* ∝ *high*) (*repc no*)) = *repc* (*high no*) ∧
((*repc* ∝ *low*) (*repc no*)) = *repc* (*low no*)
apply −
apply (*erule-tac x*=*repc no* **in** *ballE*)
apply (*auto simp add*: *null-comp-def*)
done
with *share-case-cond*
have ((*repc* ∝ *high*) (*repc no*)) ≠ ((*repc* ∝ *low*) (*repc no*))
by *auto*
with *red-lnort red-rnort rnort-Node lnort-Node share-case-cond*
show *reduced* (*Node lnort* (*repc no*) *rnort*)
apply −
apply (*rule-tac lp*=*repc* (*low no*) **and** *rp*=*repc* (*high no*) **and** *llt*=*llnort* **and** *rlt* = *rlnort* **and** *lrt*=*lrnort* **and** *rrt*=*rrnort* **in** *reduced-children-parent*)
apply *auto*
done
next
fix *lnort rnort*
assume *lnort-dag*:
Dag (*repc* (*low no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *lnort*
assume *ord-lnort*: *ordered lnort var*
assume *rnort-dag*:
Dag (*repc* (*high no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rnort*
assume *ord-rnort*: *ordered rnort var*
assume *lnort-in-repNodes*: *set-of lnort* ⊆ *repb* ‘*Nodes n ll*
assume *rnort-in-repNodes*: *set-of rnort* ⊆ *repb* ‘*Nodes n ll*
from *lnort-in-repNodes repbNodes-in-Nodes*
have *lnort-in-Nodes*: *set-of lnort* ⊆ *Nodes n ll*
by *simp*
from *rnort-in-repNodes repbNodes-in-Nodes*
have *rnort-in-Nodes*: *set-of rnort* ⊆ *Nodes n ll*
by *simp*

from *rhn-nNull rnort-dag* **obtain** *lrnort rrnort* **where**
rnort-Node: *rnort* = (*Node lrnort* (*repc* (*high no*)) *rrnort*)
by *auto*
from *rln-nNull lnort-dag* **obtain** *llnort rlnort* **where**
lnort-Node: *lnort* = (*Node llnort* (*repc* (*low no*)) *rlnort*)
by *auto*
from *lnort-dag rln-nNull lnort-in-Nodes*
have *repc* (*low no*) ∈ *set-of lnort*
by *auto*
with *lnort-in-Nodes*
have *repc* (*low no*) ∈ *Nodes n ll*
by *blast*
with *wf-ll nsll*

have *vrlno-sn*: *var* (*repc* (*low no*)) < *n*
apply −
apply (*drule wf-ll-Nodes-pret*)
apply (*auto simp add*: *length-ll-eq*)
done
from *rnort-dag rhn-nNull rnort-in-Nodes*
have *repc* (*high no*) ∈ *set-of rnort*
by *auto*
with *rnort-in-Nodes*
have *repc* (*high no*) ∈ *Nodes n ll*
by *blast*
with *wf-ll nsll* **have** *vrhno-sn*: *var* (*repc* (*high no*)) < *n*
apply −
apply (*drule wf-ll-Nodes-pret*)
apply (*auto simp add*: *length-ll-eq*)
done
with *varrepno-n vrlno-sn lnort-dag ord-lnort rnort-dag rnort-Node lnort-Node ord-rnort*
show *ordered* (*Node lnort* (*repc no*) *rnort*) *var*
by *auto*
next
fix *lnort rnort*
assume *lnort-in-Nodes*: *set-of lnort* ⊆ *repb* '*Nodes n ll*
assume *rnort-in-Nodes*: *set-of rnort* ⊆ *repb* '*Nodes n ll*
from *lnort-in-Nodes repbNodes-repcNodes*
have *lnort-in-repcNodes*: *set-of lnort* ⊆ *repc* '*Nodes n ll*
by *simp*
from *rnort-in-Nodes repbNodes-repcNodes*
have *rnort-in-repcNodes*: *set-of rnort* ⊆ *repc* '*Nodes n ll*
by *simp*
have *nNodessubset*: *Nodes n ll* ⊆ *Nodes* (*n+1*) *ll*
by (*simp add*: *Nodes-subset*)
then have *repc-Nodes-subset*:
repc '*Nodes n ll* ⊆ *repc* '*Nodes* (*n+1*) *ll*
by *blast*
from *no-in-Nodes* **have** *repc no* ∈ *repc* '*Nodes* (*n+1*) *ll*
by *blast*
with *repc-Nodes-subset lnort-in-repcNodes rnort-in-repcNodes*
show *set-of* (*Node lnort* (*repc no*) *rnort*) ⊆
repc '*Nodes* (*n* + *1*) *ll*
apply *simp*
apply *blast*
done
next
fix *rnot lnot rnort lnort rnobdt lnobdt rnorbdt lnorbdt*
assume *lnobdt-def*: *bdt lnot var* = *Some lnobdt*
assume *rnobdt-def*: *bdt rnot var* = *Some rnobdt*
assume *rnorbdt-def*: *bdt rnort var* = *Some rnorbdt*
assume *cong-rno-rnor*: *rnobdt* ∼ *rnorbdt*

assume *lnot-dag*: *Dag* (*low no*) *low high lnot*
assume *rnot-dag*: *Dag* (*high no*) *low high rnot*
assume ¬ *Suc 0* < *var no*
then have *varnoseq1*: *var no* = *0* ∨ *var no* = *1* **by** *auto*
show ∃ *nobdt*. *bdt* (*Node lnot no rnot*) *var* = *Some nobdt* ∧
(∃ *norbdt*. *bdt* (*Node lnort* (*repc no*) *rnort*) *var* = *Some norbdt* ∧
nobdt ∼ *norbdt*)
proof (*cases var no* = *0*)
case *True*
note *vnoNull*=*this*
with *pret-dag ord-pret no-in-pret lno-nNull hno-nNull*
show *?thesis*
apply −
apply (*drule var-ordered-children*)
apply *auto*
done
next
assume *var no* ≠ *0*
with *varnoseq1* **have** *vnoOne*: *var no* = *1* **by** *simp*
from *pret-dag ord-pret no-in-pret lno-nNull hno-nNull*
vnoOne
have *vlvrNull*: *var* (*low no*) = *0* ∧ *var* (*high no*) = *0*
apply −
apply (*drule var-ordered-children*)
apply *auto*
done
then have *vlNull*: *var* (*low no*) = *0* **by** *simp*
from *vlvrNull* **have** *vrNull*: *var* (*high no*) = *0* **by** *simp*
from *lnobdt-def lnot-dag vlNull lno-nNull*
have *lnobdt-Zero*: *lnobdt* = *Zero*
apply −
apply (*drule bdt-Some-var0-Zero*)
apply *auto*
done
from *rnobdt-def rnot-dag vrNull hno-nNull*
have *rnobdt-Zero*: *rnobdt* = *Zero*
apply −
apply (*drule bdt-Some-var0-Zero*)
apply *auto*
done
from *lnobdt-Zero lnobdt-def*
have *bdt lnot var* = *Some Zero* **by** *simp*
with *lnot-dag vlNull*
have *lnot-Node*: *lnot* = (*Node Tip* (*low no*) *Tip*)
by *auto*
from *rnobdt-Zero rnobdt-def rnot-dag vrNull*
have *rnot-Node*: *rnot* = (*Node Tip* (*high no*) *Tip*)
by *auto*
from *pret-dag no-in-pret* **obtain** *not*

where *not-ex*: *Dag no low high not*
apply −
apply (*drule dag-setof-exD*)
apply *auto*
done
with *pret-dag no-in-pret* **have** *not-ex-in-pret*: *not <= pret*
apply −
apply (*rule set-of-subdag*)
apply *auto*
done
from *not-ex lnot-dag rnot-dag nonNull*
have *not-def*: *not* = (*Node lnot no rnot*)
by (*simp add*: *Dag-unique del*: *Dag-Ref*)
with *not-ex-in-pret prebdt-pret*
have *nobdt-ex*: ∃ *nobdt*. *bdt* (*Node lnot no rnot*) *var* = *Some nobdt*
apply −
apply (*rule subbdt-ex*)
apply *auto*
done
then obtain *nobdt* **where**
nobdt-def: *bdt* (*Node lnot no rnot*) *var* = *Some nobdt* **by** *auto*
from *not-def* **have** *root not* = *no* **by** *simp*
with *nobdt-def vnoOne not-def*
have *not* = (*Node Tip no Tip*)
apply −
apply (*drule bdt-Some-var1-One*)
apply *auto*
done
with *not-def* **have** *rnot* = *Tip* **by** *simp*
with *rnot-Node* **show** *?thesis* **by** *simp*
qed
next
fix *lnot rnot lnobdt rnobdt*
assume *lnot-dag*: *Dag* (*low no*) *low high lnot*
assume *rnot-dag*: *Dag* (*high no*) *low high rnot*
assume *lnobdt-def*: *bdt lnot var* = *Some lnobdt*
assume *rnobdt-def*: *bdt rnot var* = *Some rnobdt*
assume *onesvarno*: *Suc 0* < *var no*
with *rnobdt-def lnot-dag rnot-dag lnobdt-def*
show *bdt* (*Node lnot no rnot*) *var* =
Some (*Bdt-Node lnobdt* (*var no*) *rnobdt*) **by** *simp*
next
fix *rnot lnot rnort lnort rnobdt lnobdt rnorbdt lnorbdt*
assume *rnort-dag*:
Dag (*repc* (*high no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rnort*
assume *lnort-dag*:
Dag (*repc* (*low no*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *lnort*
assume *rnorbdt-def*: *bdt rnort var* = *Some rnorbdt*
assume *lnorbdt-def*: *bdt lnort var* = *Some lnorbdt*

assume *varno-bOne*: *Suc 0 < var no*
with *varno* **have** *Suc 0 < n* **by** *simp*
with *varrepno-n* **have** *Suc 0 < var (repc no)* **by** *simp*
with *rnorbdt-def lnorbdt-def*
show *bdt (Node lnort (repc no) rnort) var =*
Some (Bdt-Node lnorbdt (var (repc no)) rnorbdt)
by *simp*
next
fix *rnobdt lnobdt rnorbdt lnorbdt*
assume *lcong-eval*: *lnobdt ∼ lnorbdt*
assume *rcong-eval*: *rnobdt ∼ rnorbdt*
from *varno varrepno-n* **have** *var (repc no) = var no* **by** *simp*
with *lcong-eval rcong-eval*
show *Bdt-Node lnobdt (var no) rnobdt ∼*
Bdt-Node lnorbdt (var (repc no)) rnorbdt
apply (*unfold cong-eval-def*)
apply (*rule ext*)
by *simp*
next
fix *rnot lnot rnort lnort*
assume *lnort-repb*: ∀ *no*∈*set-of lnort. repb no = no*
assume *rnort-repb*: ∀ *no*∈*set-of rnort. repb no = no*
assume *rnort-in-repb-Nodesn*: *set-of rnort ⊆ repb ‘ Nodes n ll*
assume *lnort-in-repb-Nodesn*: *set-of lnort ⊆ repb ‘ Nodes n ll*
from *repbNodes-in-Nodes rnort-in-repb-Nodesn*
have *rnort-in-Nodesn*: *set-of rnort ⊆ Nodes n ll*
by *blast*
from *repbNodes-in-Nodes lnort-in-repb-Nodesn*
have *lnort-in-Nodesn*: *set-of lnort ⊆ Nodes n ll*
by *blast*
have *rnort-repc*: ∀ *no*∈*set-of rnort. repc no = no*
proof
fix *pt*
assume *pt-in-rnort*: *pt ∈ set-of rnort*
with *rnort-in-Nodesn* **have** *pt ∈ Nodes n ll*
by *blast*
with *Nodesn-notin-lln* **have** *pt ∉ set (ll ! n)*
by *auto*
with *repbc-nc* **have** *repb pt = repc pt*
by *auto*
with *rnort-repb pt-in-rnort* **show** *repc pt = pt*
by *auto*
qed
have *lnort-repc*: ∀ *no*∈*set-of lnort. repc no = no*
proof
fix *pt*
assume *pt-in-lnort*: *pt ∈ set-of lnort*
with *lnort-in-Nodesn* **have** *pt ∈ Nodes n ll*
by *blast*

```
          with Nodesn-notin-lln have pt ∉ set (ll ! n)
            by auto
          with repbc-nc have repb pt = repc pt
            by auto
          with lnort-repb pt-in-lnort show repc pt = pt
            by auto
        qed
        show ∀ no∈set-of (Node lnort (repc no) rnort). repc no = no
        proof
          fix pt
          assume pt-in-rept: pt ∈ set-of (Node lnort (repc no) rnort)
          show repc pt = pt
          proof (cases pt ∈ set-of lnort)
            case True
            with lnort-repc show ?thesis
              by auto
          next
            assume pt-notin-lnort: pt ∉ set-of lnort
            show ?thesis
            proof (cases pt ∈ set-of rnort)
              case True
              with rnort-repc show ?thesis
                by auto
            next
              assume pt-notin-rnort: pt ∉ set-of rnort
              with pt-notin-lnort pt-in-rept have pt = repc no
                by simp
              with rrno-eq-rno show repc pt = pt
                by simp
            qed
          qed
        qed
      qed

      with varrep rrno-eq-rno show ?thesis by simp
    qed
  qed
  with rnonN show ?thesis by simp
 qed
qed
note while-while-prop=this
from wf-ll nsll
have ∀ no ∈ Nodes n ll. no ∉ set (ll ! n)
  apply (simp add: Nodes-def length-ll-eq)
  apply clarify
  apply (drule no-in-one-ll)
  apply auto
  done
with repbc-nc have ∀ no ∈ Nodes n ll. repb no = repc no
```

apply −
apply (*rule ballI*)
apply (*erule-tac x=no* **in** *allE*)
apply *simp*
done
then have *repbNodes-repcNodes*:
repb '(*Nodes n ll*) = *repc* '(*Nodes n ll*)
apply −
apply *rule*
apply *blast*
apply *rule*
apply (*erule imageE*)
apply (*erule-tac x=xa* **in** *ballE*)
prefer *2*
apply *simp*
apply *rule*
apply *auto*
done
then have *repcNodes-repbNodes*:
repc '(*Nodes n ll*) = *repb* '(*Nodes n ll*)
by *simp*
have *repbc-dags-eq*:
Dags (*repc* ' *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*) =
Dags (*repb* ' *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
apply −
apply *rule*
apply *rule*
apply (*erule Dags.cases*)
apply (*rule DagsI*)
prefer *4*
apply *rule*
apply (*erule Dags.cases*)
apply (*rule DagsI*)
proof −
fix *x p t*
assume *t-in-repcNodes*: *set-of t* ⊆ *repc* ' *Nodes n ll*
assume *x-t*: *x=t*
with *t-in-repcNodes repcNodes-repbNodes*
show *set-of x* ⊆ *repb* ' *Nodes n ll*
by *simp*
next
fix *x p t*
assume *t-in-repcNodes*: *set-of t* ⊆ *repc* ' *Nodes n ll*
assume *t-dag*: *Dag p* (*repc* ∝ *low*) (*repc* ∝ *high*) *t*
assume *t-nTip*: *t* ≠ *Tip*
assume *x-t*: *x=t*
from *t-nTip t-dag* **have** *p* ≠ *Null*
apply −
apply (*case-tac p=Null*)

```
    apply auto
    done
  with t-nTip t-dag obtain lt rt where t-Node: t=Node lt p rt
    by auto
  from t-in-repcNodes t-dag t-nTip t-Node obtain q where
    rq-p: repc q = p and q-in-Nodes: q ∈ Nodes n ll
    apply simp
    apply (elim conjE)
    apply (erule imageE)
    apply auto
    done
  from q-in-Nodes have repb q = repc q
    by (rule Nodes-n-rep-nc [rule-format])
  with rq-p have repbq-p: repb q = p by simp
  from q-in-Nodes
  have Dag (repb q) (repb ∝ low) (repb ∝ high) t =
    Dag (repc q) (repc ∝ low) (repc ∝ high) t
    by (rule Nodes-repbc-Dags-eq [rule-format])
  with t-dag rq-p have Dag (repb q) (repb ∝ low) (repb ∝ high) t by simp
  with repbq-p x-t show Dag p (repb ∝ low) (repb ∝ high) x
    by simp
next
  fix x p t
  assume t-in-repcNodes: set-of t ⊆ repb ‘ Nodes n ll
  assume x-t: x=t
  with t-in-repcNodes repbNodes-repcNodes
  show set-of x ⊆ repc ‘ Nodes n ll
    by simp
next
  fix x p t
  assume t-in-repcNodes: set-of t ⊆ repb ‘ Nodes n ll
  assume t-dag: Dag p (repb ∝ low) (repb ∝ high) t
  assume t-nTip:  t ≠ Tip
  assume x-t: x=t
  from t-nTip t-dag have p ≠ Null
    apply −
    apply (case-tac p=Null)
    apply auto
    done
  with t-nTip t-dag obtain lt rt where t-Node: t=Node lt p rt
    by auto
  from t-in-repcNodes t-dag t-nTip t-Node obtain q where
    rq-p: repb q = p and q-in-Nodes: q ∈ Nodes n ll
    apply simp
    apply (elim conjE)
    apply (erule imageE)
    apply auto
    done
  from q-in-Nodes have repb q = repc q
```

by (*rule Nodes-n-rep-nc* [*rule-format*])
with *rq-p* **have** *repbq-p*: *repc q* = *p* **by** *simp*
from *q-in-Nodes*
have *Dag* (*repb q*) (*repb* ∝ *low*) (*repb* ∝ *high*) *t* =
Dag (*repc q*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t*
by (*rule Nodes-repbc-Dags-eq* [*rule-format*])
with *t-dag rq-p* **have** *Dag* (*repc q*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t* **by** *simp*
with *repbq-p x-t* **show** *Dag p* (*repc* ∝ *low*) (*repc* ∝ *high*) *x*
by *simp*
next
fix *x p* **and** *t* :: *dag*
assume *x-t*: *x* = *t*
assume *t-nTip*: *t* ≠ *Tip*
with *x-t* **show** *x*≠ *Tip* **by** *simp*
next
fix *x p* **and** *t* :: *dag*
assume *x-t*: *x* = *t*
assume *t-nTip*: *t* ≠ *Tip*
with *x-t* **show** *x*≠ *Tip* **by** *simp*
qed
from *pret-dag wf-ll nsll*
have *null-notin-Nodes-Suc-n*: *Null* ∉ *Nodes* (*Suc n*) *ll*
by − (*rule Null-notin-Nodes*,*auto simp add*: *length-ll-eq*)
{ fix *t1 t2*
assume *t1-in-DagsNodesn*:
t1 ∈ *Dags* (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *t2-notin-DagsNodesn*:
t2 ∉ *Dags* (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *t2-in-DagsNodesSucn*:
t2 ∈ *Dags* (*repc* ‘ *Nodes* (*Suc n*) *ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *isomorphic-dags-eq-asm*:
∀ *t1 t2*. *t1* ∈ *Dags* (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
∧ *t2* ∈ *Dags* (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
⟶ *isomorphic-dags-eq t1 t2 var*
assume *repbc-Dags*:
Dags (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*) =
Dags (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
from *t1-in-DagsNodesn repbc-Dags*
have *t1-repb-subnode*:
t1 ∈ *Dags* (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
by *simp*
from *t2-in-DagsNodesSucn*
have *t2-in-DagsNodesSucn*:
t2 ∈ *Dags* (*repc* ‘ *Nodes* (*Suc n*) *ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
by *simp*
from *repbNodes-in-Nodes repbNodes-repcNodes*
have *repcNodesn-in-Nodesn*: *repc* ‘*Nodes n ll* ⊆ *Nodes n ll*
by *simp*
from *t1-in-DagsNodesn* **obtain** *q* **where**

Dag-q-Nodes-n:
Dag (*repc q*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t1* ∧ *q* ∈ *Nodes n ll*
proof (*elim Dags.cases*)
fix *p t*
assume *t1-t*: *t1* = *t*
assume *t-in-repcNodesn*: *set-of t* ⊆ *repc* ‘ *Nodes n ll*
assume *t-dag*: *Dag p* (*repc* ∝ *low*) (*repc* ∝ *high*) *t*
assume *t-nTip*: *t* ≠ *Tip*
assume *obtain-prop*: ⋀*q*. *Dag* (*repc q*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t1* ∧
q ∈ *Nodes n ll* ⟹ *?thesis*
from *t-nTip t-dag* **have** *p* ≠ *Null*
apply −
apply (*case-tac p=Null*)
apply *auto*
done
with *t-nTip t-dag* **obtain** *lt rt* **where** *t-Node*: *t=Node lt p rt*
by *auto*
from *t-in-repcNodesn t-dag t-nTip t-Node* **obtain** *k* **where**
rk-p: *repc k* = *p* **and** *k-in-Nodes*: *k* ∈ *Nodes n ll*
apply *simp*
apply (*elim conjE*)
apply (*erule imageE*)
apply *auto*
done
with *t1-t t-dag obtain-prop rk-p k-in-Nodes* **show** *?thesis*
by *auto*
qed
with *wf-ll nsll* **have** *varq-sn*: (*var q* < *n*)
apply (*simp add*: *Nodes-def*)
apply (*elim conjE*)
apply (*erule exE*)
apply (*simp add*: *wf-ll-def length-ll-eq*)
apply (*elim conjE*)
apply (*thin-tac* ∀ *q*. *q* ∈ *set-of pret* ⟶ *q* ∈ *set* (*ll* ! *var q*))
apply (*erule-tac x=x* **in** *allE*)
apply *auto*
done
from *Dag-q-Nodes-n* **have** *q-in-Nodesn*: *q* ∈ *Nodes n ll*
by *simp*
then have ∃ *k<n*. *q* ∈ *set* (*ll* ! *k*)
by (*simp add*: *Nodes-def*)
with *wf-ll nsll* **have** *q* ∉ *set* (*ll* ! *n*)
apply −
apply (*erule exE*)
apply (*rule-tac i=k* **and** *j=n* **in** *no-in-one-ll*)
apply (*auto simp add*: *length-ll-eq*)
done
with *repbc-nc* **have** *repbc-q*: *repc q* = *repb q*
apply −

apply (*erule-tac x=q* **in** *allE*)
apply *auto*
done
from *normalize-prop q-in-Nodesn* **have** *var* (*repb q*) <= *var q*
apply −
apply (*erule-tac x=q* **in** *ballE*)
apply *auto*
done
with *repbc-q* **have** *var-repc-q*: *var* (*repc q*) <= *var q*
by *simp*
with *varq-sn* **have** *var-repc-q-n*: *var* (*repc q*) < *n*
by *simp*

from *Nodes-subset Dag-q-Nodes-n while-while-prop*
have *ord-t1-var-rep*: *ordered t1 var* ∧ *var* (*repc q*) <= *var q*
apply (*elim conjE*)
apply (*erule-tac x=q* **in** *ballE*)
apply *auto*
done
then have *ord-t1*: *ordered t1 var* **by** *simp*
from *ord-t1-var-rep* **have** *varrep-q*: *var* (*repc q*) <= *var q* **by** *simp*
from *t2-in-DagsNodesSucn* **have** *ord-t2*: *ordered t2 var*
proof (*elim Dags.cases*)
fix *p t*
assume *t-in-repcNodes*: *set-of t* ⊆ *repc* ‘ *Nodes* (*Suc n*) *ll*
assume *t-nTip*: *t* ≠ *Tip*
assume *t2t*: *t2* = *t*
assume *t-dag*: *Dag p* (*repc* ∝ *low*) (*repc* ∝ *high*) *t*
from *t-in-repcNodes* **have** *x-in-repcNodesSucn*:
∀ *x*. *x* ∈ *set-of t* ⟶ *x* ∈ *repc* ‘ *Nodes* (*Suc n*) *ll*
apply −
apply *auto*
done
from *t-nTip t-dag* **have** *p* ≠ *Null*
apply −
apply (*case-tac p=Null*)
apply *auto*
done
with *t-nTip t-dag* **obtain** *lt rt* **where** *t-Node*: *t=Node lt p rt*
by *auto*
then have *p* ∈ *set-of t*
by *auto*
with *x-in-repcNodesSucn* **have** *p* ∈ *repc* ‘ *Nodes* (*Suc n*) *ll*
by *simp*
then obtain *a* **where** *repca-p*: *p=repc a* **and**
a-in-NodesSucn: *a* ∈ *Nodes* (*Suc n*) *ll*
by *auto*
with *repca-p while-while-prop t-dag t2t* **show** *?thesis*
apply −

apply (*erule-tac x=a* **in** *ballE*)
apply (*elim conjE exE*)
apply (*subgoal-tac nort = t*)
prefer *2*
apply (*simp add*: *Dag-unique*)
apply *auto*
done
qed
from *while-while-prop* **have** *while-prop-part*:
∀ *no* ∈ *Nodes* (*Suc n*) *ll*.
var (*repc no*) <= *var no*
by *auto*
from *while-while-prop* **have** *rep-rep-nort*:
∀ *no*∈*Nodes* (*n* + *1*) *ll*. (∃ *nort*. *Dag* (*repc no*) (*repc* ∝ *low*) (*repc* ∝ *high*) *nort* ∧
(∀ *no*∈*set-of nort*. *repc no* = *no*))
by *auto*
from *repcNodes-in-Nodes null-notin-Nodes-Suc-n*
have ∀ *no* ∈ *Nodes* (*n*+*1*) *ll*. *repc no* ≠ *Null*
by *auto*
with *rep-rep-nort* **have** ∀ *no* ∈ *Nodes* (*n*+*1*) *ll*. *repc* (*repc no*) = (*repc no*)
apply −
apply (*rule ballI*)
apply (*erule-tac x=no* **in** *ballE*)
prefer *2*
apply *simp*
apply (*erule-tac x=no* **in** *ballE*)
apply (*erule exE*)
apply (*subgoal-tac repc no* ∈ *set-of nort*)
apply (*elim conjE*)
apply (*erule-tac x=repc no* **in** *ballE*)
apply *simp*
apply *simp*
apply (*simp*)
apply (*elim conjE*)
apply (*thin-tac* ∀ *no*∈*set-of nort*. *repc no* = *no*)
apply *auto*
done
with *t2-in-DagsNodesSucn t2-notin-DagsNodesn ord-t2 while-prop-part*
wf-ll nsll repcNodes-in-Nodes **obtain** *a* **where**
t2-repc-dag: *Dag* (*repc a*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t2* **and**
a-in-lln: *a* ∈ *set* (*ll* ! *n*)
apply −
apply (*drule restrict-root-Node*)
apply (*auto simp add*: *length-ll-eq*)
done
with *wf-ll nsll* **have** *a-in-pret*: *a* ∈ *set-of pret*
by (*simp add*: *wf-ll-def length-ll-eq*)

from *wf-ll nsll a-in-lln* **have** *vara-n*: *var a* = *n*
  by (*simp add*: *wf-ll-def length-ll-eq*)
from *a-in-lln rep-prop* **obtain**
  repp-nNull: *repc a* ≠ *Null* **and**
  repp-reduce: (*repc* ∝ *low*) *a* = (*repc* ∝ *high*) *a* ∧ *low a* ≠ *Null*
  ⟶ *repc a* = (*repc* ∝ *high*) *a* **and**
  repp-share: ((*repc* ∝ *low*) *a* = (*repc* ∝ *high*) *a* ⟶ *low a* = *Null*)
  ⟶ *repc a* ∈ *set* (*ll* ! *n*) ∧
  repc (*repc a*) = *repc a* ∧
  (∀ *no1*∈*set* (*ll* ! *n*). ((*repc* ∝ *high*) *no1* = (*repc* ∝ *high*) *a* ∧
  (*repc* ∝ *low*) *no1* = (*repc* ∝ *low*) *a*) = (*repc a* = *repc no1*))
  using [[*simp-depth-limit=4*]]
  by *auto*
from *t2-repc-dag a-in-lln repp-nNull* **obtain** *lt2 rt2* **where**
  t2-Node: *t2* = (*Node lt2* (*repc a*) *rt2*)
  by *auto*
have *isomorphic-dags-eq t1 t2 var*
proof (*cases* (*repc* ∝ *low*) *a* = (*repc* ∝ *high*) *a* ∧ *low a* ≠ *Null*)
  case *True*
  note *red=this*
  then have *red-case*: (*repc* ∝ *low*) *a* = (*repc* ∝ *high*) *a*
    by *simp*
  from *red* **have** *low-nNull*: *low a* ≠ *Null*
    by *simp*
  with *pret-dag prebdt-pret a-in-pret* **have** *highp-nNull*: *high a* ≠ *Null*
    apply −
    apply (*drule balanced-bdt*)
    apply *auto*
    done
  from *pret-dag ord-pret a-in-pret low-nNull highp-nNull*
  have *var-children-smaller*: *var* (*low a*) < *var a* ∧ *var* (*high a*) < *var a*
    apply −
    apply (*rule var-ordered-children*)
    apply *auto*
    done
  from *pret-dag a-in-pret* **have** *a-nNull*: *a* ≠ *Null*
    apply −
    apply (*rule set-of-nn* [*rule-format*])
    apply *auto*
    done
  with *a-in-pret highp-nNull pret-dag* **have** *high a* ∈ *set-of pret*
    apply −
    apply (*drule subelem-set-of-high*)
    apply *auto*
    done
  with *wf-ll* **have** *high a* ∈ *set* (*ll* ! (*var* (*high a*)))
    by (*simp add*: *wf-ll-def*)
  with *a-in-lln t2-repc-dag var-children-smaller vara-n*
  have ∃ *k*<*n*. (*high a*) ∈ *set* (*ll* ! *k*)

by *auto*
then have *higha-in-Nodesn*: (*high a*) ∈ *Nodes n ll*
by (*simp add*: *Nodes-def*)
then have *rhigha-in-rNodesn*: *repc* (*high a*) ∈ *repc* ‘ *Nodes n ll*
by *simp*
from *higha-in-Nodesn normalize-prop* **obtain** *rt* **where**
rt-dag: *Dag* (*repb* (*high a*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rt* **and**
rt-in-repbNort: *set-of rt* ⊆ *repb* ‘*Nodes n ll*
apply –
apply (*erule-tac x*=*high a* **in** *ballE*)
apply *auto*
done
from *rt-in-repbNort repbNodes-repcNodes*
have *rt-in-repcNodesn*: *set-of rt* ⊆ *repc* ‘*Nodes n ll*
by *blast*
from *rt-dag higha-in-Nodesn*
have *repcrt-dag*: *Dag* (*repc* (*high a*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *rt*
apply –
apply (*drule Nodes-repbc-Dags-eq* [*rule-format*])
apply *auto*
done
have *rt-nTip*: *rt* ≠ *Tip*
proof –
have *repc* (*high a*) ≠ *Null*
proof –
note *rhigha-in-rNodesn*
also have *repc* ‘*Nodes n ll* ⊆ *repc* ‘*Nodes* (*Suc n*) *ll*
using *Nodes-subset*
by *blast*
also have . . . ⊆ *Nodes* (*Suc n*) *ll*
using *repcNodes-in-Nodes*
by *simp*
finally show *?thesis*
using *null-notin-Nodes-Suc-n*
by *auto*
qed
with *repcrt-dag* **show** *?thesis* **by** *auto*
qed
from *a-nNull a-in-pret low-nNull pret-dag* **have** *low a* ∈ *set-of pret*
apply –
apply (*drule subelem-set-of-low*)
apply *auto*
done
with *wf-ll* **have** *low a* ∈ *set* (*ll* ! (*var* (*low a*)))
by (*simp add*: *wf-ll-def*)
with *a-in-lln t2-repc-dag var-children-smaller vara-n*
have ∃ *k*<*n*. (*low a*) ∈ *set* (*ll* ! *k*)
by *auto*
then have (*low a*) ∈ *Nodes n ll*

by (*simp add*: *Nodes-def*)
then have *rlow-in-rNodesn*: *repc* (*low a*) ∈ *repc* ‘ *Nodes n ll*
by *simp*
show *?thesis*
proof −
from *repp-reduce low-nNull highp-nNull red-case*
have *repc-p-def*: *repc a* = *repc* (*high a*)
by (*simp add*: *null-comp-def*)
with *rt-in-repcNodesn repcrt-dag rhigha-in-rNodesn a-in-lln t2-repc-dag repc-p-def rt-nTip*
have *t2-in-Dags-Nodesn*:
t2 ∈ *Dags* (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
apply −
apply (*rule DagsI*)
apply *simp*
apply (*subgoal-tac t2=rt*)
apply (*auto simp add*: *Dag-unique*)
done
from *t1-in-DagsNodesn t2-in-Dags-Nodesn repbc-dags-eq isomorphic-dags-eq-asm*
show *shared-t1-t2*: *isomorphic-dags-eq t1 t2 var*
apply −
apply (*erule-tac x=t1* **in** *allE*)
apply (*erule-tac x=t2* **in** *allE*)
apply *simp*
done
qed
next
assume *share*: ¬ ((*repc* ∝ *low*) *a* = (*repc* ∝ *high*) *a* ∧ *low a* ≠ *Null*)
then
have *share*: (*repc* ∝ *low*) *a* ≠ (*repc* ∝ *high*) *a* ∨ *low a* = *Null*
using [[*simp-depth-limit=1*]]
by *simp*
with *repp-share* **obtain**
ra-in-llbn: *repc a* ∈ *set* (*ll* ! *n*) **and**
rra-ra: *repc* (*repc a*) = *repc a* **and**
two-nodes-share: (∀ *no1*∈*set* (*ll* ! *n*).
((*repc* ∝ *high*) *no1* = (*repc* ∝ *high*) *a* ∧
(*repc* ∝ *low*) *no1* = (*repc* ∝ *low*) *a*) = (*repc a* = *repc no1*))
using [[*simp-depth-limit=3*]]
by *simp*
from *wf-ll ra-in-llbn nsll* **have** *var-repc-a-n*: *var* (*repc a*) = *n*
by (*auto simp add*: *wf-ll-def length-ll-eq*)
show *?thesis*
proof (*auto simp add*: *isomorphic-dags-eq-def*)
fix *bdt1*
assume *bdt-t1*: *bdt t1 var* = *Some bdt1*
assume *bdt-t2*: *bdt t2 var* = *Some bdt1*
show *t1* = *t2*

proof (*cases t1*)
case *Tip*
with *bdt-t1* **show** *?thesis*
by *simp*
next
case (*Node lt1 p1 rt1*)
note *t1-Node=this*
with *Dag-q-Nodes-n* **have** *p1*=(*repc q*)
by *simp*
with *t2-Node bdt-t1 bdt-t2 t1-Node* **have** *var* (*repc q*) = *var* (*repc a*)
apply −
apply (*rule same-bdt-var*)
apply *auto*
done
with *var-repc-q-n var-repc-a-n* **show** *?thesis*
by *simp*
qed
qed
qed }
note *mixed-Dags-case* = *this*
from *repbc-dags-eq isomorphic-dags-eq*
have *dags-shared*:
∀ *t1 t2*. *t1* ∈ *Dags* (*repc* ‘ *Nodes* (*Suc n*) *ll*)(*repc* ∝ *low*)(*repc* ∝ *high*)∧
t2 ∈ *Dags* (*repc* ‘ *Nodes* (*Suc n*) *ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
⟶ *isomorphic-dags-eq t1 t2 var*
apply −
apply (*rule Dags-Nodes-cases*)
apply (*rule isomorphic-dags-eq-sym*)
proof −
fix *t1 t2*
assume *t1-in-Dagsn*:
t1 ∈ *Dags* (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *t2-in-Dagsn*:
t2 ∈ *Dags* (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *isomorphic-dags-eq-asm*:
∀ *t1 t2*. *t1* ∈ *Dags* (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*) ∧
t2 ∈ *Dags* (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
⟶ *isomorphic-dags-eq t1 t2 var*
assume *repb-repc-Dags*:
Dags (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*) =
Dags (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
with *t1-in-Dagsn t2-in-Dagsn isomorphic-dags-eq-asm*
show *isomorphic-dags-eq t1 t2 var* **by** *simp*
next
fix *t1 t2*
assume *t1-in-DagsNodesn*:
t1 ∈ *Dags* (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *t2-notin-DagsNodesn*:
t2 ∉ *Dags* (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)

assume *t2-in-DagsNodesSucn*:
t2 ∈ *Dags* (*repc* ‘ *Nodes* (*Suc n*) *ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *isomorphic-dags-eq-asm*:
∀ *t1 t2*. *t1* ∈ *Dags* (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*) ∧
t2 ∈ *Dags* (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
⟶ *isomorphic-dags-eq t1 t2 var*
assume *repbc-Dags*:
Dags (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*) =
Dags (*repb* ‘ *Nodes n ll*) (*repb* ∝ *low*) (*repb* ∝ *high*)
from *t1-in-DagsNodesn t2-notin-DagsNodesn t2-in-DagsNodesSucn*
isomorphic-dags-eq-asm repbc-Dags
show *isomorphic-dags-eq t1 t2 var*
apply −
apply (*rule mixed-Dags-case*)
apply *auto*
done
next
fix *t1 t2*
assume *t1-in-DagsNodesSucn*:
t1 ∈ *Dags* (*repc* ‘ *Nodes* (*Suc n*) *ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *t1-notin-DagsNodesn*:
t1 ∉ *Dags* (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *t2-in-DagsNodesSucn*:
t2 ∈ *Dags* (*repc* ‘ *Nodes* (*Suc n*) *ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
assume *t2-notin-DagsNodesn*:
t2 ∉ *Dags* (*repc* ‘ *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)

from *t1-in-DagsNodesSucn* **have** *ord-t1*: *ordered t1 var*
proof (*elim Dags.cases*)
fix *p t*
assume *t-in-repcNodes*: *set-of t* ⊆ *repc* ‘ *Nodes* (*Suc n*) *ll*
assume *t-nTip*: *t* ≠ *Tip*
assume *t2t*: *t1* = *t*
assume *t-dag*: *Dag p* (*repc* ∝ *low*) (*repc* ∝ *high*) *t*
from *t-in-repcNodes*
have *x-in-repcNodesSucn*:
∀ *x*. *x* ∈ *set-of t* ⟶ *x* ∈ *repc* ‘ *Nodes* (*Suc n*) *ll*
apply −
apply *auto*
done
from *t-nTip t-dag* **have** *p* ≠ *Null*
apply −
apply (*case-tac p=Null*)
apply *auto*
done
with *t-nTip t-dag* **obtain** *lt rt* **where** *t-Node*: *t=Node lt p rt*
by *auto*

then have *p* ∈ *set-of t*
by *auto*
with *x-in-repcNodesSucn* **have** *p* ∈ *repc* ‘ *Nodes* (*Suc n*) *ll*
by *simp*
then obtain *a* **where**
repca-p: *p*=*repc a* **and** *a-in-NodesSucn*: *a* ∈ *Nodes* (*Suc n*) *ll*
by *auto*
with *repca-p while-while-prop t-dag t2t* **show** *?thesis*
apply −
apply (*erule-tac x*=*a* **in** *ballE*)
apply (*elim conjE exE*)
apply (*subgoal-tac nort* = *t*)
prefer *2*
apply (*simp add*: *Dag-unique*)
apply *auto*
done
qed
from *while-while-prop*
have *while-prop-part*: ∀ *no* ∈ *Nodes* (*Suc n*) *ll*.
var (*repc no*) <= *var no*
by *auto*
from *while-while-prop* **have** *rep-rep-nort*:
∀ *no*∈*Nodes* (*n* + *1*) *ll*.
(∃ *nort*. *Dag* (*repc no*) (*repc* ∝ *low*) (*repc* ∝ *high*) *nort* ∧
(∀ *no*∈*set-of nort*. *repc no* = *no*))
by *auto*
from *repcNodes-in-Nodes null-notin-Nodes-Suc-n*
have ∀ *no* ∈ *Nodes* (*n*+*1*) *ll*. *repc no* ≠ *Null*
by *auto*
with *rep-rep-nort*
have *rep-rep-no*: ∀ *no* ∈ *Nodes* (*n*+*1*) *ll*. *repc* (*repc no*) = (*repc no*)
apply −
apply (*rule ballI*)
apply (*erule-tac x*=*no* **in** *ballE*)
prefer *2*
apply *simp*
apply (*erule-tac x*=*no* **in** *ballE*)
apply (*erule exE*)
apply (*subgoal-tac repc no* ∈ *set-of nort*)
apply (*elim conjE*)
apply (*erule-tac x*=*repc no* **in** *ballE*)
apply *simp*
apply *simp*
apply (*simp*)
apply (*elim conjE*)
apply (*thin-tac* ∀ *no*∈*set-of nort*. *repc no* = *no*)
apply *auto*
done
with *t1-in-DagsNodesSucn t1-notin-DagsNodesn ord-t1 while-prop-part*

wf-ll
  nsll repcNodes-in-Nodes **obtain** *a1* **where**
  t1-repc-dag: *Dag* (*repc a1*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t1* **and**
  a1-in-lln: *a1* ∈ *set* (*ll* ! *n*)
  apply −
  apply (*drule restrict-root-Node*)
  apply (*auto simp add*: *length-ll-eq*)
  done
with *wf-ll nsll* **have** *a1-in-pret*: *a1* ∈ *set-of pret*
  by (*simp add*: *wf-ll-def length-ll-eq*)
from *wf-ll nsll a1-in-lln* **have** *vara1-n*: *var a1* = *n*
  by (*simp add*: *wf-ll-def length-ll-eq*)
from *a1-in-lln rep-prop* **obtain**
  repa1-nNull: *repc a1* ≠ *Null* **and**
  repa1-reduce: (*repc* ∝ *low*) *a1* = (*repc* ∝ *high*) *a1* ∧ *low a1* ≠ *Null*
  ⟶ *repc a1* = (*repc* ∝ *high*) *a1* **and**
  repa1-share: ((*repc* ∝ *low*) *a1* = (*repc* ∝ *high*) *a1* ⟶ *low a1* = *Null*)
  ⟶ *repc a1* ∈ *set* (*ll* ! *n*) ∧ *repc* (*repc a1*) = *repc a1* ∧
  (∀ *no1*∈*set* (*ll* ! *n*). ((*repc* ∝ *high*) *no1* = (*repc* ∝ *high*) *a1* ∧
  (*repc* ∝ *low*) *no1* = (*repc* ∝ *low*) *a1*) = (*repc a1* = *repc no1*))
  using [[*simp-depth-limit*=*4*]]
  by *auto*
from *t1-repc-dag a1-in-lln repa1-nNull* **obtain** *lt1 rt1* **where**
  t1-Node: *t1* = (*Node lt1* (*repc a1*) *rt1*)
  by *auto*

from *t2-in-DagsNodesSucn* **have** *ord-t2*: *ordered t2 var*
proof (*elim Dags.cases*)
  fix *p t*
  assume *t-in-repcNodes*: *set-of t* ⊆ *repc* ‘ *Nodes* (*Suc n*) *ll*
  assume *t-nTip*: *t* ≠ *Tip*
  assume *t2t*: *t2* = *t*
  assume *t-dag*: *Dag p* (*repc* ∝ *low*) (*repc* ∝ *high*) *t*
  from *t-in-repcNodes*
  have *x-in-repcNodesSucn*:
    ∀ *x*. *x* ∈ *set-of t* ⟶ *x* ∈ *repc* ‘ *Nodes* (*Suc n*) *ll*
    apply −
    apply *auto*
    done
  from *t-nTip t-dag* **have** *p* ≠ *Null*
    apply −
    apply (*case-tac p*=*Null*)
    apply *auto*
    done
  with *t-nTip t-dag* **obtain** *lt rt* **where** *t-Node*: *t*=*Node lt p rt*
    by *auto*

then have *p* ∈ *set-of t*
by *auto*
with *x-in-repcNodesSucn* **have** *p* ∈ *repc* ‘ *Nodes* (*Suc n*) *ll*
by *simp*
then obtain *a* **where**
repca-p: *p*=*repc a* **and** *a-in-NodesSucn*: *a* ∈ *Nodes* (*Suc n*) *ll*
by *auto*
with *repca-p while-while-prop t-dag t2t* **show** *?thesis*
apply −
apply (*erule-tac x*=*a* **in** *ballE*)
apply (*elim conjE exE*)
apply (*subgoal-tac nort* = *t*)
prefer *2*
apply (*simp add*: *Dag-unique*)
apply *auto*
done
qed
from *rep-rep-no t2-in-DagsNodesSucn t2-notin-DagsNodesn ord-t2 while-prop-part wf-ll*
nsll repcNodes-in-Nodes **obtain** *a2* **where**
t2-repc-dag: *Dag* (*repc a2*) (*repc* ∝ *low*) (*repc* ∝ *high*) *t2* **and**
a2-in-lln: *a2* ∈ *set* (*ll* ! *n*)
apply −
apply (*drule restrict-root-Node*)
apply (*auto simp add*: *length-ll-eq*)
done
with *wf-ll nsll* **have** *a2-in-pret*: *a2* ∈ *set-of pret*
by (*simp add*: *wf-ll-def length-ll-eq*)
from *wf-ll nsll a2-in-lln* **have** *vara2-n*: *var a2* = *n*
by (*simp add*: *wf-ll-def length-ll-eq*)
from *a2-in-lln rep-prop* **obtain**
repa2-nNull: *repc a2* ≠ *Null* **and**
repa2-reduce: (*repc* ∝ *low*) *a2* = (*repc* ∝ *high*) *a2* ∧ *low a2* ≠ *Null*
⟶ *repc a2* = (*repc* ∝ *high*) *a2* **and**
repa2-share: ((*repc* ∝ *low*) *a2* = (*repc* ∝ *high*) *a2* ⟶ *low a2* = *Null*)
⟶ *repc a2* ∈ *set* (*ll* ! *n*) ∧ *repc* (*repc a2*) = *repc a2* ∧
(∀ *no1*∈*set* (*ll* ! *n*). ((*repc* ∝ *high*) *no1* = (*repc* ∝ *high*) *a2* ∧
(*repc* ∝ *low*) *no1* = (*repc* ∝ *low*) *a2*) = (*repc a2* = *repc no1*))
using [[*simp-depth-limit* = *4*]]
by *auto*
from *t2-repc-dag a2-in-lln repa2-nNull* **obtain** *lt2 rt2* **where**
t2-Node: *t2* = (*Node lt2* (*repc a2*) *rt2*)
by *auto*
show *isomorphic-dags-eq t1 t2 var*
proof (*cases* (*repc* ∝ *low*) *a1* = (*repc* ∝ *high*) *a1* ∧ *low a1* ≠ *Null*)
case *True*
note *t1-red-cond*=*this*
with *t1-red-cond* **have** *t1-red-case*: (*repc* ∝ *low*) *a1* = (*repc* ∝ *high*) *a1*
by *simp*

from *t1-red-cond* **have** *lowa1-nNull*: *low a1* ≠ *Null*
by *simp*
with *pret-dag prebdt-pret a1-in-pret* **have** *higha1-nNull*: *high a1* ≠ *Null*
apply −
apply (*drule balanced-bdt*)
apply *auto*
done
from *pret-dag ord-pret a1-in-pret lowa1-nNull higha1-nNull*
have *var-children-smaller-a1*: *var* (*low a1*) < *var a1* ∧ *var* (*high a1*) < *var a1*
apply −
apply (*rule var-ordered-children*)
apply *auto*
done
from *pret-dag a1-in-pret* **have** *a1-nNull*: *a1* ≠ *Null*
apply −
apply (*rule set-of-nn* [*rule-format*])
apply *auto*
done
with *a1-in-pret higha1-nNull pret-dag* **have** *high a1* ∈ *set-of pret*
apply −
apply (*drule subelem-set-of-high*)
apply *auto*
done
with *wf-ll* **have** *high a1* ∈ *set* (*ll* ! (*var* (*high a1*)))
by (*simp add*: *wf-ll-def*)
with *a1-in-lln t1-repc-dag var-children-smaller-a1 vara1-n*
have ∃ *k*<*n*. (*high a1*) ∈ *set* (*ll* ! *k*)
by *auto*
then have *higha1-in-Nodesn*: (*high a1*) ∈ *Nodes n ll*
by (*simp add*: *Nodes-def*)
then have *rhigha1-in-rNodesn*: *repc* (*high a1*) ∈ *repc* ' *Nodes n ll*
by *simp*
from *higha1-in-Nodesn normalize-prop* **obtain** *rt1* **where**
rt1-dag: *Dag* (*repb* (*high a1*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rt1* **and**
rt1-in-repbNort: *set-of rt1* ⊆ *repb* '*Nodes n ll*
apply −
apply (*erule-tac x*=*high a1* **in** *ballE*)
apply *auto*
done
from *rt1-in-repbNort repbNodes-repcNodes*
have *rt1-in-repcNodesn*: *set-of rt1* ⊆ *repc* '*Nodes n ll*
by *blast*
from *rt1-dag higha1-in-Nodesn*
have *repcrt1-dag*: *Dag* (*repc* (*high a1*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *rt1*
apply −
apply (*drule Nodes-repbc-Dags-eq* [*rule-format*])
apply *auto*
done

have *rt1-nTip*: *rt1* ≠ *Tip*
proof −
  have *repc* (*high a1*) ≠ *Null*
  proof −
    note *rhigha1-in-rNodesn*
    also have *repc ‘Nodes n ll* ⊆ *repc ‘Nodes* (*Suc n*) *ll*
      using *Nodes-subset*
      by *blast*
    also have . . . ⊆ *Nodes* (*Suc n*) *ll*
      using *repcNodes-in-Nodes*
      by *simp*
    finally show *?thesis*
      using *null-notin-Nodes-Suc-n*
      by *auto*
  qed
  with *repcrt1-dag* **show** *?thesis* **by** *auto*
qed
from *repa1-reduce lowa1-nNull higha1-nNull t1-red-case*
have *repc-a1-def*: *repc a1* = *repc* (*high a1*)
  by (*simp add*: *null-comp-def*)
with *rt1-in-repcNodesn repcrt1-dag rhigha1-in-rNodesn a1-in-lln*
  t1-repc-dag repc-a1-def rt1-nTip
have *t1-in-Dags-Nodesn*:
  t1 ∈ *Dags* (*repc ‘ Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
  apply −
  apply (*rule DagsI*)
  apply *simp*
  apply (*subgoal-tac t1*=*rt1*)
  apply (*auto simp add*: *Dag-unique*)
  done
show *?thesis*
proof (*cases* (*repc* ∝ *low*) *a2* = (*repc* ∝ *high*) *a2* ∧ *low a2* ≠ *Null*)
  case *True*
  note *t2-red-cond*=*this*
 with *t2-red-cond* **have** *t2-red-case*: (*repc* ∝ *low*) *a2* = (*repc* ∝ *high*) *a2*
    by *simp*
  from *t2-red-cond* **have** *lowa2-nNull*: *low a2* ≠ *Null*
    by *simp*
 with *pret-dag prebdt-pret a2-in-pret* **have** *higha2-nNull*: *high a2* ≠ *Null*

    apply −
    apply (*drule balanced-bdt*)
    apply *auto*
    done
  from *pret-dag ord-pret a2-in-pret lowa2-nNull higha2-nNull*
  have *var-children-smaller-a2*:
  var (*low a2*) < *var a2* ∧ *var* (*high a2*) < *var a2*
    apply −
    apply (*rule var-ordered-children*)

```
  apply auto
  done
from pret-dag a2-in-pret have a2-nNull: a2 ≠ Null
  apply −
  apply (rule set-of-nn [rule-format])
  apply auto
  done
with a2-in-pret higha2-nNull pret-dag have high a2 ∈ set-of pret
  apply −
  apply (drule subelem-set-of-high)
  apply auto
  done
with wf-ll have high a2 ∈ set (ll ! (var (high a2)))
  by (simp add: wf-ll-def)
with a2-in-lln t2-repc-dag var-children-smaller-a2 vara2-n
have ∃ k<n. (high a2) ∈ set (ll ! k)
  by auto
then have higha2-in-Nodesn: (high a2) ∈ Nodes n ll
  by (simp add: Nodes-def)
then have rhigha2-in-rNodesn: repc (high a2) ∈ repc ‘ Nodes n ll
  by simp
from higha2-in-Nodesn normalize-prop obtain rt2 where
  rt2-dag: Dag (repb (high a2)) (repb ∝ low) (repb ∝ high) rt2 and
  rt2-in-repbNort: set-of rt2 ⊆ repb ‘Nodes n ll
  apply −
  apply (erule-tac x=high a2 in ballE)
  apply auto
  done
from rt2-in-repbNort repbNodes-repcNodes
have rt2-in-repcNodesn: set-of rt2 ⊆ repc ‘Nodes n ll
  by blast
from rt2-dag higha2-in-Nodesn
have repcrt2-dag: Dag (repc (high a2)) (repc ∝ low) (repc ∝ high) rt2
  apply −
  apply (drule Nodes-repbc-Dags-eq [rule-format])
  apply auto
  done
have rt2-nTip: rt2 ≠ Tip
proof −
  have repc (high a2) ≠ Null
  proof −
    note rhigha2-in-rNodesn
    also have repc ‘Nodes n ll ⊆ repc ‘Nodes (Suc n) ll
      using Nodes-subset
      by blast
    also have . . . ⊆ Nodes (Suc n) ll
      using repcNodes-in-Nodes
      by simp
    finally show ?thesis
```

```
            using null-notin-Nodes-Suc-n
            by auto
          qed
          with repcrt2-dag show ?thesis by auto
        qed
        from repa2-reduce lowa2-nNull higha2-nNull t2-red-case
        have repc-a2-def: repc a2 = repc (high a2)
          by (simp add: null-comp-def)
        with rt2-in-repcNodesn repcrt2-dag rhigha2-in-rNodesn a2-in-lln
          t2-repc-dag repc-a2-def rt2-nTip
        have t2-in-Dags-Nodesn:
          t2 ∈ Dags (repc ` Nodes n ll) (repc ∝ low) (repc ∝ high)
          apply −
          apply (rule DagsI)
          apply simp
          apply (subgoal-tac t2=rt2)
          apply (auto simp add: Dag-unique)
          done
             from isomorphic-dags-eq t1-in-Dags-Nodesn t2-in-Dags-Nodesn
repbc-dags-eq
        show ?thesis
          by auto
      next
        assume t2-share-cond:
          ¬ ((repc ∝ low) a2 = (repc ∝ high) a2 ∧ low a2 ≠ Null)
        from t1-in-Dags-Nodesn t2-notin-DagsNodesn t2-in-DagsNodesSucn
          isomorphic-dags-eq repbc-dags-eq
        show ?thesis
          apply −
          apply (rule mixed-Dags-case)
          apply auto
          done
      qed
    next
      assume t1-share-cond:
        ¬((repc ∝ low) a1 = (repc ∝ high) a1 ∧ low a1 ≠ Null)
      with repa1-share obtain
        repca1-in-llbn: repc a1 ∈ set (ll ! n) and
        reprepa1: repc (repc a1) = repc a1 and
        twonodes-llbn-a1:
        (∀ no1∈set (ll ! n). ((repc ∝ high) no1 = (repc ∝ high) a1 ∧
        (repc ∝ low) no1 = (repc ∝ low) a1) = (repc a1 = repc no1))
        using [[simp-depth-limit=2]]
        by auto
      show ?thesis
      proof (cases (repc ∝ low) a2 = (repc ∝ high) a2 ∧ low a2 ≠ Null)
        case True
        note t2-red-cond=this
       with t2-red-cond have t2-red-case: (repc ∝ low) a2 = (repc ∝ high) a2
```

by *simp*
from *t2-red-cond* **have** *lowa2-nNull*: *low a2* ≠ *Null*
by *simp*
with *pret-dag prebdt-pret a2-in-pret* **have** *higha2-nNull*: *high a2* ≠ *Null*

apply −
apply (*drule balanced-bdt*)
apply *auto*
done
from *pret-dag ord-pret a2-in-pret lowa2-nNull higha2-nNull*
have *var-children-smaller-a2*:
var (*low a2*) < *var a2* ∧ *var* (*high a2*) < *var a2*
apply −
apply (*rule var-ordered-children*)
apply *auto*
done
from *pret-dag a2-in-pret* **have** *a2-nNull*: *a2* ≠ *Null*
apply −
apply (*rule set-of-nn* [*rule-format*])
apply *auto*
done
with *a2-in-pret higha2-nNull pret-dag* **have** *high a2* ∈ *set-of pret*
apply −
apply (*drule subelem-set-of-high*)
apply *auto*
done
with *wf-ll*
have *high a2* ∈ *set* (*ll* ! (*var* (*high a2*)))
by (*simp add*: *wf-ll-def*)
with *a2-in-lln t2-repc-dag var-children-smaller-a2 vara2-n*
have ∃ *k*<*n*. (*high a2*) ∈ *set* (*ll* ! *k*)
by *auto*
then have *higha2-in-Nodesn*: (*high a2*) ∈ *Nodes n ll*
by (*simp add*: *Nodes-def*)
then have *rhigha2-in-rNodesn*: *repc* (*high a2*) ∈ *repc* ' *Nodes n ll*
by *simp*
from *higha2-in-Nodesn normalize-prop* **obtain** *rt2* **where**
rt2-dag: *Dag* (*repb* (*high a2*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rt2* **and**
rt2-in-repbNort: *set-of rt2* ⊆ *repb* '*Nodes n ll*
apply −
apply (*erule-tac x*=*high a2* **in** *ballE*)
apply *auto*
done
from *rt2-in-repbNort repbNodes-repcNodes*
have *rt2-in-repcNodesn*: *set-of rt2* ⊆ *repc* '*Nodes n ll*
by *blast*
from *rt2-dag higha2-in-Nodesn*
have *repcrt2-dag*: *Dag* (*repc* (*high a2*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *rt2*
apply −

apply (*drule Nodes-repbc-Dags-eq* [*rule-format*])
apply *auto*
done
have *rt2-nTip*: *rt2* ≠ *Tip*
proof −
have *repc* (*high a2*) ≠ *Null*
proof −
note *rhigha2-in-rNodesn*
also have *repc* '*Nodes n ll* ⊆ *repc* '*Nodes* (*Suc n*) *ll*
using *Nodes-subset*
by *blast*
also have ... ⊆ *Nodes* (*Suc n*) *ll*
using *repcNodes-in-Nodes*
by *simp*
finally show *?thesis*
using *null-notin-Nodes-Suc-n*
by *auto*
qed
with *repcrt2-dag* **show** *?thesis* **by** *auto*
qed
from *repa2-reduce lowa2-nNull higha2-nNull t2-red-case*
have *repc-a2-def*: *repc a2* = *repc* (*high a2*)
by (*simp add*: *null-comp-def*)
with *rt2-in-repcNodesn repcrt2-dag rhigha2-in-rNodesn a2-in-lln*
t2-repc-dag repc-a2-def rt2-nTip
have *t2-in-Dags-Nodesn*:
t2 ∈ *Dags* (*repc* ' *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
apply −
apply (*rule DagsI*)
apply *simp*
apply (*subgoal-tac t2*=*rt2*)
apply (*auto simp add*: *Dag-unique*)
done
from *t2-in-Dags-Nodesn t1-notin-DagsNodesn t1-in-DagsNodesSucn*
isomorphic-dags-eq repbc-dags-eq
have *isomorphic-dags-eq t2 t1 var*
apply −
apply (*rule mixed-Dags-case*)
apply *auto*
done
then show *?thesis*
by (*simp add*: *isomorphic-dags-eq-sym*)
next
assume *t2-shared-cond*:
¬ ((*repc* ∝ *low*) *a2* = (*repc* ∝ *high*) *a2* ∧ *low a2* ≠ *Null*)
with *repa2-share* **obtain**
repca2-in-llbn: *repc a2* ∈ *set* (*ll* ! *n*) **and**
reprepa2: *repc* (*repc a2*) = *repc a2* **and**
twonodes-llbn-a2: (∀ *no1*∈*set* (*ll* ! *n*).

((*repc* ∝ *high*) *no1* = (*repc* ∝ *high*) *a2* ∧
(*repc* ∝ *low*) *no1* = (*repc* ∝ *low*) *a2*) = (*repc a2* = *repc no1*))
using [[*simp-depth-limit=2*]]
by *auto*
from *twonodes-llbn-a2 a1-in-lln*
have *share-a1-a2*:
((*repc* ∝ *high*) *a1* = (*repc* ∝ *high*) *a2* ∧
(*repc* ∝ *low*) *a1* = (*repc* ∝ *low*) *a2*) = (*repc a2* = *repc a1*)
by *auto*
from *twonodes-llbn-a1 repca1-in-llbn reprepa1*
have *children-repc-eq-a1*: (*repc* ∝ *high*) (*repc a1*) = (*repc* ∝ *high*) *a1*
∧
(*repc* ∝ *low*) (*repc a1*) = (*repc* ∝ *low*) *a1*
by *auto*
from *twonodes-llbn-a2 repca2-in-llbn reprepa2*
have *children-repc-eq-a2*: (*repc* ∝ *high*) (*repc a2*) = (*repc* ∝ *high*) *a2*
∧
(*repc* ∝ *low*) (*repc a2*) = (*repc* ∝ *low*) *a2*
by *auto*
from *t1-Node t2-Node* **show** *?thesis*
proof (*clarsimp simp add*: *isomorphic-dags-eq-def*)
fix *bdt1*
assume *t1-bdt*: *bdt* (*Node lt1* (*repc a1*) *rt1*) *var* = *Some bdt1*
assume *t2-bdt*: *bdt* (*Node lt2* (*repc a2*) *rt2*) *var* = *Some bdt1*
show *lt1* = *lt2* ∧ *repc a1* = *repc a2* ∧ *rt1* = *rt2*
proof (*cases bdt1*)
case *Zero*
with *t1-bdt t1-Node* **obtain**
lt1-Tip: *lt1* = *Tip* **and**
rt1-Tip: *rt1* = *Tip*
by *simp*
from *Zero t2-bdt t2-Node* **obtain**
lt2-Tip: *lt2* = *Tip* **and**
rt2-Tip: *rt2* = *Tip*
by *simp*
from *t1-repc-dag t1-Node lt1-Tip* **have** (*repc* ∝ *low*) (*repc a1*) =
Null
by *simp*
with *children-repc-eq-a1*
have *repc-low-a1-Null*: (*repc* ∝ *low*) *a1* = *Null*
by *simp*
from *t1-repc-dag t1-Node rt1-Tip*
have (*repc* ∝ *high*) (*repc a1*) = *Null*
by *simp*
with *children-repc-eq-a1*
have *repc-high-a1-Null*: (*repc* ∝ *high*) *a1* = *Null*
by *simp*
from *t2-repc-dag t2-Node lt2-Tip* **have** (*repc* ∝ *low*) (*repc a2*) =
Null

by *simp*
with *children-repc-eq-a2*
have *repc-low-a2-Null*: (*repc* ∝ *low*) *a2* = *Null*
by *simp*
from *t2-repc-dag t2-Node rt2-Tip*
have (*repc* ∝ *high*) (*repc a2*) = *Null*
by *simp*
with *children-repc-eq-a2*
have *repc-high-a2-Null*: (*repc* ∝ *high*) *a2* = *Null*
by *simp*
with *share-a1-a2* *repc-low-a1-Null repc-high-a1-Null*
repc-low-a2-Null repc-high-a2-Null
have *repc a2* = *repc a1*
by *auto*
with *lt1-Tip rt1-Tip lt2-Tip rt2-Tip* **show** *?thesis*
by *auto*
next
case *One*
with *t1-bdt t1-Node* **obtain**
lt1-Tip: *lt1* = *Tip* **and**
rt1-Tip: *rt1* = *Tip*
by *simp*
from *One t2-bdt t2-Node* **obtain**
lt2-Tip: *lt2* = *Tip* **and**
rt2-Tip: *rt2* = *Tip*
by *simp*
from *t1-repc-dag t1-Node lt1-Tip* **have** (*repc* ∝ *low*) (*repc a1*) =
Null
by *simp*
with *children-repc-eq-a1*
have *repc-low-a1-Null*: (*repc* ∝ *low*) *a1* = *Null*
by *simp*
from *t1-repc-dag t1-Node rt1-Tip* **have** (*repc* ∝ *high*) (*repc a1*) =
Null
by *simp*
with *children-repc-eq-a1*
have *repc-high-a1-Null*: (*repc* ∝ *high*) *a1* = *Null*
by *simp*
from *t2-repc-dag t2-Node lt2-Tip* **have** (*repc* ∝ *low*) (*repc a2*) =
Null
by *simp*
with *children-repc-eq-a2*
have *repc-low-a2-Null*: (*repc* ∝ *low*) *a2* = *Null*
by *simp*
from *t2-repc-dag t2-Node rt2-Tip* **have** (*repc* ∝ *high*) (*repc a2*) =
Null
by *simp*
with *children-repc-eq-a2*
have *repc-high-a2-Null*: (*repc* ∝ *high*) *a2* = *Null*

by *simp*
with *share-a1-a2 repc-low-a1-Null repc-high-a1-Null repc-low-a2-Null repc-high-a2-Null*
have *repc a2* = *repc a1*
by *auto*
with *lt1-Tip rt1-Tip lt2-Tip rt2-Tip* **show** *?thesis*
by *auto*
next
case (*Bdt-Node lbdt v rbdt*)
note *bdt-Node-case*=*this*
with *t1-bdt t1-Node* **obtain**
lbdt-def-lt1: *bdt lt1 var* = *Some lbdt* **and**
rbdt-def-rt1: *bdt rt1 var* = *Some rbdt*
by *auto*
from *t2-bdt bdt-Node-case t2-Node* **obtain**
lbdt-def-lt2: *bdt lt2 var* = *Some lbdt* **and**
rbdt-def-rt2: *bdt rt2 var* = *Some rbdt*
by *auto*
from *lbdt-def-lt1 t1-Node t1-repc-dag children-repc-eq-a1*
have (*repc* ∝ *low*) *a1* ≠ *Null*
by *auto*
then have *low-a1-nNull*: (*low a1*) ≠ *Null*
by (*auto simp*: *null-comp-def*)
from *rbdt-def-rt1 t1-Node t1-repc-dag children-repc-eq-a1*
have (*repc* ∝ *high*) *a1* ≠ *Null*
by *auto*
then have *high-a1-nNull*: (*high a1*) ≠ *Null*
by (*auto simp*: *null-comp-def*)
from *lbdt-def-lt2 t2-Node t2-repc-dag children-repc-eq-a2*
have (*repc* ∝ *low*) *a2* ≠ *Null*
by *auto*
then have *low-a2-nNull*: (*low a2*) ≠ *Null*
by (*auto simp*: *null-comp-def*)
from *rbdt-def-rt2 t2-Node t2-repc-dag children-repc-eq-a2*
have (*repc* ∝ *high*) *a2* ≠ *Null*
by *auto*
then have *high-a2-nNull*: (*high a2*) ≠ *Null*
by (*auto simp*: *null-comp-def*)

from *pret-dag ord-pret a1-in-pret low-a1-nNull high-a1-nNull*
have *var-children-smaller-a1*:
var (*low a1*) < *var a1* ∧ *var* (*high a1*) < *var a1*
apply −
apply (*rule var-ordered-children*)
apply *auto*
done
from *pret-dag a1-in-pret* **have** *a1-nNull*: *a1* ≠ *Null*
apply −

apply (*rule set-of-nn* [*rule-format*])
apply *auto*
done

with *a1-in-pret high-a1-nNull pret-dag* **have** *high a1* ∈ *set-of pret*
apply −
apply (*drule subelem-set-of-high*)
apply *auto*
done
with *wf-ll*
have *high a1* ∈ *set* (*ll* ! (*var* (*high a1*)))
by (*simp add*: *wf-ll-def*)
with *a1-in-lln t1-repc-dag var-children-smaller-a1 vara1-n*
have ∃ *k*<*n*. (*high a1*) ∈ *set* (*ll* ! *k*)
by *auto*
then have *higha1-in-Nodesn*: (*high a1*) ∈ *Nodes n ll*
by (*simp add*: *Nodes-def*)
then have *rhigha1-in-rNodesn*:
repc (*high a1*) ∈ *repc* ' *Nodes n ll*
by *simp*
from *higha1-in-Nodesn normalize-prop* **obtain** *rt1h* **where**
rt1-dag: *Dag* (*repb* (*high a1*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rt1h* **and**
rt1-in-repbNort: *set-of rt1h* ⊆ *repb* '*Nodes n ll*
apply −
apply (*erule-tac x*=*high a1* **in** *ballE*)
apply *auto*
done
from *rt1-in-repbNort repbNodes-repcNodes*
have *rt1-in-repcNodesn*: *set-of rt1h* ⊆ *repc* '*Nodes n ll*
by *blast*
from *rt1-dag higha1-in-Nodesn*
have *repcrt1-dag*:
Dag (*repc* (*high a1*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *rt1h*
apply −
apply (*drule Nodes-repbc-Dags-eq* [*rule-format*])
apply *auto*
done
from *t1-Node t1-repc-dag high-a1-nNull children-repc-eq-a1*
have *Dag* (*repc* (*high a1*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *rt1*
by (*auto simp add*: *null-comp-def*)
with *repcrt1-dag* **have** *rt1h-rt1*: *rt1h* = *rt1* **by** (*simp add*: *Dag-unique*)
have *rt1-nTip*: *rt1* ≠ *Tip*
proof −
have *repc* (*high a1*) ≠ *Null*
proof −
note *rhigha1-in-rNodesn*
also have
repc '*Nodes n ll* ⊆ *repc* '*Nodes* (*Suc n*) *ll*

using *Nodes-subset*
by *blast*
also have ... ⊆ *Nodes* (*Suc n*) *ll*
using *repcNodes-in-Nodes*
by *simp*
finally show *?thesis*
using *null-notin-Nodes-Suc-n*
by *auto*
qed
with *repcrt1-dag rt1h-rt1* **show** *?thesis* **by** *auto*
qed
with *rt1-in-repcNodesn repcrt1-dag rhigha1-in-rNodesn a1-in-lln t1-repc-dag rt1h-rt1*
have *rt1-in-Dags-Nodesn*:
rt1 ∈ *Dags* (*repc* ' *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
apply −
apply (*rule DagsI*)
apply *auto*
done

from *a1-nNull a1-in-pret low-a1-nNull pret-dag*
have *low a1* ∈ *set-of pret*
apply −
apply (*drule subelem-set-of-low*)
apply *auto*
done
with *wf-ll* **have**
low a1 ∈ *set* (*ll* ! (*var* (*low a1*))) **by** (*simp add*: *wf-ll-def*)
with *a1-in-lln t1-repc-dag var-children-smaller-a1 vara1-n*
have ∃ *k*<*n*. (*low a1*) ∈ *set* (*ll* ! *k*)
by *auto*
then have *lowa1-in-Nodesn*: (*low a1*) ∈ *Nodes n ll*
by (*simp add*: *Nodes-def*)
then have *rlowa1-in-rNodesn*: *repc* (*low a1*) ∈ *repc* ' *Nodes n ll*
by *simp*
from *lowa1-in-Nodesn normalize-prop* **obtain** *lt1h* **where**
lt1-dag: *Dag* (*repb* (*low a1*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *lt1h* **and**
lt1-in-repbNort: *set-of lt1h* ⊆ *repb* '*Nodes n ll*
apply −
apply (*erule-tac x*=*low a1* **in** *ballE*)
apply *auto*
done
from *lt1-in-repbNort repbNodes-repcNodes*
have *lt1-in-repcNodesn*: *set-of lt1h* ⊆ *repc* '*Nodes n ll*
by *blast*
from *lt1-dag lowa1-in-Nodesn*
have *repclt1-dag*: *Dag* (*repc* (*low a1*)) (*repc* ∝ *low*) (*repc* ∝ *high*)

lt1h

apply −
apply (*drule Nodes-repbc-Dags-eq* [*rule-format*])
apply *auto*
done
from *t1-Node t1-repc-dag low-a1-nNull children-repc-eq-a1*
have *Dag* (*repc* (*low a1*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *lt1*
by (*auto simp add*: *null-comp-def*)
with *repclt1-dag* **have** *lt1h-lt1*: *lt1h* = *lt1* **by** (*simp add*: *Dag-unique*)
have *lt1-nTip*: *lt1* ≠ *Tip*
proof −
have *repc* (*low a1*) ≠ *Null*
proof −
note *rlowa1-in-rNodesn*
also have
repc '*Nodes n ll* ⊆ *repc* '*Nodes* (*Suc n*) *ll*
using *Nodes-subset*
by *blast*
also have ... ⊆ *Nodes* (*Suc n*) *ll*
using *repcNodes-in-Nodes*
by *simp*
finally show *?thesis*
using *null-notin-Nodes-Suc-n*
by *auto*
qed
with *repclt1-dag lt1h-lt1* **show** *?thesis* **by** *auto*
qed
with *lt1-in-repcNodesn repclt1-dag rlowa1-in-rNodesn a1-in-lln*
t1-repc-dag lt1h-lt1
have *lt1-in-Dags-Nodesn*:
lt1 ∈ *Dags* (*repc* ' *Nodes n ll*) (*repc* ∝ *low*) (*repc* ∝ *high*)
apply −
apply (*rule DagsI*)
apply *auto*
done

from *pret-dag ord-pret a2-in-pret low-a2-nNull high-a2-nNull*
have *var-children-smaller-a2*:
var (*low a2*) < *var a2* ∧ *var* (*high a2*) < *var a2*
apply −
apply (*rule var-ordered-children*)
apply *auto*
done
from *pret-dag a2-in-pret* **have** *a2-nNull*: *a2* ≠ *Null*
apply −
apply (*rule set-of-nn* [*rule-format*])
apply *auto*

done

with *a2-in-pret high-a2-nNull pret-dag* **have** *high a2* ∈ *set-of pret*
apply −
apply (*drule subelem-set-of-high*)
apply *auto*
done
with *wf-ll* **have** *high a2* ∈ *set* (*ll* ! (*var* (*high a2*)))
by (*simp add*: *wf-ll-def*)
with *a2-in-lln t2-repc-dag var-children-smaller-a2 vara2-n*
have ∃ *k*<*n*. (*high a2*) ∈ *set* (*ll* ! *k*)
by *auto*
then have *higha2-in-Nodesn*: (*high a2*) ∈ *Nodes n ll*
by (*simp add*: *Nodes-def*)
then have *rhigha2-in-rNodesn*:
repc (*high a2*) ∈ *repc* ‘ *Nodes n ll*
by *simp*
from *higha2-in-Nodesn normalize-prop* **obtain** *rt2h* **where**
rt2-dag: *Dag* (*repb* (*high a2*)) (*repb* ∝ *low*) (*repb* ∝ *high*) *rt2h* **and**
rt2-in-repbNort: *set-of rt2h* ⊆ *repb* ‘*Nodes n ll*
apply −
apply (*erule-tac x*=*high a2* **in** *ballE*)
apply *auto*
done
from *rt2-in-repbNort repbNodes-repcNodes*
have *rt2-in-repcNodesn*: *set-of rt2h* ⊆ *repc* ‘*Nodes n ll*
by *blast*
from *rt2-dag higha2-in-Nodesn*
have *repcrt2-dag*:
Dag (*repc* (*high a2*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *rt2h*
apply −
apply (*drule Nodes-repbc-Dags-eq* [*rule-format*])
apply *auto*
done
from *t2-Node t2-repc-dag high-a2-nNull children-repc-eq-a2*
have *Dag* (*repc* (*high a2*)) (*repc* ∝ *low*) (*repc* ∝ *high*) *rt2*
by (*auto simp add*: *null-comp-def*)
with *repcrt2-dag* **have** *rt2h-rt2*: *rt2h* = *rt2* **by** (*simp add*: *Dag-unique*)
have *rt2-nTip*: *rt2* ≠ *Tip*
proof −
have *repc* (*high a2*) ≠ *Null*
proof −
note *rhigha2-in-rNodesn*
also have
repc ‘*Nodes n ll* ⊆ *repc* ‘*Nodes* (*Suc n*) *ll*
using *Nodes-subset*
by *blast*
also have . . . ⊆ *Nodes* (*Suc n*) *ll*

```
      using repcNodes-in-Nodes
      by simp
    finally show ?thesis
      using null-notin-Nodes-Suc-n
      by auto
  qed
  with repcrt2-dag rt2h-rt2 show ?thesis by auto
qed
with rt2-in-repcNodesn repcrt2-dag rhigha2-in-rNodesn a2-in-lln
  t2-repc-dag  rt2h-rt2
have rt2-in-Dags-Nodesn:
  rt2 ∈ Dags (repc ‘ Nodes n ll) (repc ∝ low) (repc ∝ high)
  apply −
  apply (rule DagsI)
  apply auto
  done



from a2-nNull a2-in-pret low-a2-nNull pret-dag
have low a2 ∈ set-of pret
  apply −
  apply (drule subelem-set-of-low)
  apply auto
  done
with wf-ll have low a2 ∈ set (ll ! (var (low a2)))
  by (simp add: wf-ll-def)
with a2-in-lln t2-repc-dag var-children-smaller-a2 vara2-n
have ∃ k<n. (low a2) ∈ set (ll ! k)
  by auto
then have lowa2-in-Nodesn: (low a2) ∈ Nodes n ll
  by (simp add: Nodes-def)
then have rlowa2-in-rNodesn: repc (low a2) ∈ repc ‘ Nodes n ll
  by simp
from lowa2-in-Nodesn normalize-prop obtain lt2h where
 lt2-dag:  Dag (repb (low a2)) (repb ∝ low) (repb ∝ high) lt2h and
  lt2-in-repbNort: set-of lt2h ⊆ repb ‘Nodes n ll
  apply −
  apply (erule-tac x=low a2 in ballE)
  apply auto
  done
from lt2-in-repbNort repbNodes-repcNodes
have lt2-in-repcNodesn: set-of lt2h ⊆ repc ‘Nodes n ll
  by blast
from lt2-dag lowa2-in-Nodesn
 have repclt2-dag: Dag (repc (low a2)) (repc ∝ low) (repc ∝ high)
lt2h
  apply −
  apply (drule Nodes-repbc-Dags-eq [rule-format])
```

```
      apply auto
      done
    from t2-Node t2-repc-dag low-a2-nNull children-repc-eq-a2
    have Dag (repc (low a2)) (repc ∝ low) (repc ∝ high) lt2
      by (auto simp add: null-comp-def)
  with repclt2-dag have lt2h-lt2: lt2h = lt2 by (simp add: Dag-unique)
    have lt2-nTip: lt2 ≠ Tip
    proof –
      have repc (low a2) ≠ Null
      proof –
        note rlowa2-in-rNodesn
        also have
          repc ‘Nodes n ll ⊆ repc ‘Nodes (Suc n) ll
          using Nodes-subset
          by blast
        also have ... ⊆ Nodes (Suc n) ll
          using repcNodes-in-Nodes
          by simp
        finally show ?thesis
          using null-notin-Nodes-Suc-n
          by auto
      qed
      with repclt2-dag lt2h-lt2 show ?thesis by auto
    qed
    with lt2-in-repcNodesn repclt2-dag rlowa2-in-rNodesn a2-in-lln
      t2-repc-dag lt2h-lt2
    have lt2-in-Dags-Nodesn:
      lt2 ∈ Dags (repc ‘ Nodes n ll) (repc ∝ low) (repc ∝ high)
      apply –
      apply (rule DagsI)
      apply auto
      done


       from isomorphic-dags-eq lt1-in-Dags-Nodesn lt2-in-Dags-Nodesn
repbc-dags-eq
    have shared-lt1-lt2: isomorphic-dags-eq lt1 lt2 var
      by auto
      from isomorphic-dags-eq rt1-in-Dags-Nodesn rt2-in-Dags-Nodesn
repbc-dags-eq
    have shared-rt1-rt2: isomorphic-dags-eq rt1 rt2 var
      by auto

    from shared-lt1-lt2 lbdt-def-lt1 lbdt-def-lt2 have lt1-lt2: lt1 = lt2
      by (auto simp add: isomorphic-dags-eq-def)
    then have root-lt1-lt2: root lt1 = root lt2
      by auto
     from lt1-nTip t1-repc-dag t1-Node have (repc ∝ low) (repc a1) ≠
Null
```

by *auto*
with *lt1-nTip t1-repc-dag t1-Node* **obtain** *llt1 lt1p rlt1* **where**
lt1-Node: *lt1* = *Node llt1 lt1p rlt1*
by *auto*
with *t1-repc-dag t1-Node children-repc-eq-a1 lt1-nTip*
have *root-lt1*: *root lt1* = (*repc* ∝ *low*) *a1*
by *auto*
from *lt2-nTip t2-repc-dag t2-Node* **have** (*repc* ∝ *low*) (*repc a2*) ≠
Null
by *auto*
with *lt2-nTip t2-repc-dag t2-Node* **obtain** *llt2 lt2p rlt2* **where**
lt2-Node: *lt2* = *Node llt2 lt2p rlt2*
by *auto*
with *t2-repc-dag t2-Node children-repc-eq-a2 lt2-nTip*
have *root-lt2*: *root lt2* = (*repc* ∝ *low*) *a2*
by *auto*
from *root-lt1-lt2 root-lt2 root-lt1*
have *low-a1-a2*: (*repc* ∝ *low*) *a1* = (*repc* ∝ *low*) *a2*
by *auto*
from *shared-rt1-rt2 rbdt-def-rt1 rbdt-def-rt2* **have** *rt1-rt2*: *rt1* = *rt2*
by (*auto simp add*: *isomorphic-dags-eq-def*)
then have *root-rt1-rt2*: *root rt1* = *root rt2*
by *auto*
from *rt1-nTip t1-repc-dag t1-Node* **have** (*repc* ∝ *high*) (*repc a1*) ≠
Null
by *auto*
with *rt1-nTip t1-repc-dag t1-Node* **obtain** *lrt1 rt1p rrt1* **where**
rt1-Node: *rt1* = *Node lrt1 rt1p rrt1*
by *auto*
with *t1-repc-dag t1-Node children-repc-eq-a1 rt1-nTip*
have *root-rt1*: *root rt1* = (*repc* ∝ *high*) *a1*
by *auto*
from *rt2-nTip t2-repc-dag t2-Node*
have (*repc* ∝ *high*) (*repc a2*) ≠ *Null*
by *auto*
with *rt2-nTip t2-repc-dag t2-Node* **obtain** *lrt2 rt2p rrt2* **where**
rt2-Node: *rt2* = *Node lrt2 rt2p rrt2*
by *auto*
with *t2-repc-dag t2-Node children-repc-eq-a2 rt2-nTip*
have *root-rt2*: *root rt2* = (*repc* ∝ *high*) *a2*
by *auto*
from *root-rt1-rt2 root-rt2 root-rt1*
have *high-a1-a2*: (*repc* ∝ *high*) *a1* = (*repc* ∝ *high*) *a2*
by *auto*
from *low-a1-a2 high-a1-a2 share-a1-a2*
have *repc a1* = *repc a2*
by *auto*
with *lt1-lt2 rt1-rt2* **show** *?thesis*
by *auto*

```
          qed
         qed
        qed
       qed
      qed
     from termi dags-shared while-while-prop repcNodes-in-Nodes repc-nc n-var-prop
       wf-marking-m-ma
      show ?thesis
       by auto
     qed
    qed
    with srrl-precond all-nodes-same-var
    show ?thesis
     apply −
     apply (intro conjI)
     apply assumption+
     done
  qed
qed

end
```

References

[1] V. Ortner and N. Schirmer. Verification of BDD normalization. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 2005*, volume 3603 of *LNCS*, pages 261–277. Springer, 2005.