van Emde Boas Trees

Thomas Ammer and Peter Lammich

March 17, 2025

Abstract

The *van Emde Boas tree* or *van Emde Boas priority queue* [1, 2] is a data structure supporting membership test, insertion, predecessor and successor search, minimum and maximum determination and deletion in $\mathcal{O}(\log\log|\mathcal{U}|)$ time, where $\mathcal{U} = \{0, ..., 2^n - 1\}$ is the overall range to be considered. The presented formalization follows Chapter 20 of the popular *Introduction to Algorithms (3rd ed.)* [3] by Cormen, Leiserson, Rivest and Stein (CLRS), extending the list of formally verified CLRS algorithms [4]. Our current formalization is based on the first author's bachelor's thesis.

First, we prove correct a *functional* implementation, w.r.t. an abstract data type for sets. Apart from functional correctness, we show a resource bound, and runtime bounds w.r.t. manually defined timing functions [5] for the operations.

Next, we refine the operations to Imperative HOL [6, 7] with time [8], and show correctness and complexity. This yields a practically more efficient implementation, and eliminates the manually defined timing functions from the trusted base of the proof.

Contents

theory *VEBT-Definitions* **imports**
Main
HOL−Library.Extended-Nat
HOL−Library.Code-Target-Numeral
HOL−Library.Code-Target-Nat

begin

1 Preliminaries and Preparations

1.1 Data Type Definition

datatype *VEBT* = *is-Node*: *Node* (*info*:(*nat*∗*nat*) *option*)(*deg*: *nat*)(*treeList*: *VEBT list*) (*summary*:*VEBT*)
|
is-Leaf: *Leaf* *bool* *bool*

hide-const (**open**) *info deg treeList summary*

locale *VEBT-internal* **begin**

1.2 Functions for obtaining high and low bits of an input number.

definition *high* :: *nat* ⇒ *nat* ⇒ *nat* **where**
high x n = (*x div* (*2*^*n*))

definition *low* :: *nat* ⇒ *nat* ⇒ *nat* **where**
low x n = (*x mod* (*2*^*n*))

1.3 Some auxiliary lemmata

lemma *inthall*[*termination-simp*]: (⋀ *x*. *x* ∈ *set xs* ⟹ *P x*) ⟹ *n* < *length xs* ⟹ *P* (*xs* ! *n*)
⟨*proof*⟩

lemma *intind*: *i* < *n* ⟹ *P x* ⟹ *P* (*replicate n x* ! *i*)
⟨*proof*⟩

lemma *concat-inth*:(*xs* @[*x*]@*ys*)! (*length xs*) = *x*
⟨*proof*⟩

lemma *pos-n-replace*: *n*<*length xs* ⟹ *length xs* = *length* (*take n xs* @ [*y*] @*drop* (*Suc n*) *xs*)
⟨*proof*⟩

lemma *inthrepl*: *i* < *n* ⟹ (*replicate n x*) ! *i* = *x* ⟨*proof*⟩

lemma *nth-repl*: *m*<*length xs* ⟹ *n* <*length xs* ⟹ *m*≠ *n* ⟹(*take n xs* @ [*x*] @ *drop* (*n*+*1*) *xs*) !
m = *xs* ! *m*
⟨*proof*⟩

lemma [*termination-simp*]:**assumes** *high x deg* < *length treeList*
shows*size* (*treeList* ! *high x deg*) < *Suc* (*size-list size treeList* + *size s*)
⟨*proof*⟩

1.4 Auxiliary functions for defining valid Van Emde Boas Trees

This function checks whether an element occurs in a Leaf

fun *naive-member* :: *VEBT* ⇒ *nat* ⇒ *bool* **where**
naive-member (*Leaf a b*) *x* = (*if x* = *0 then a else if x* = *1 then b else False*)|
naive-member (*Node - 0 - -*) *-* = *False*|
naive-member (*Node - deg treeList s*) *x* = (*let pos* = *high x* (*deg div 2*) *in*
(*if pos* < *length treeList then naive-member* (*treeList* ! *pos*) (*low x* (*deg div 2*)) *else False*))

Test for elements stored by using the provide min-max-fields

fun *membermima* :: *VEBT* ⇒ *nat* ⇒ *bool* **where**
membermima (*Leaf - -*) *-* = *False*|
membermima (*Node None 0 - -*)*-* =*False*|
membermima (*Node* (*Some* (*mi*,*ma*)) *0 - -*) *x* = (*x* = *mi* ∨ *x* = *ma*)|
membermima (*Node* (*Some* (*mi*, *ma*)) *deg treeList -*) *x* = (*x* = *mi* ∨ *x* = *ma* ∨ (
let pos = *high x* (*deg div 2*) *in* (*if pos* < *length treeList*
then membermima (*treeList* ! *pos*) (*low x* (*deg div 2*)) *else False*)))|
membermima (*Node None* (*deg*) *treeList -*) *x* = (*let pos* = *high x* (*deg div 2*) *in*
(*if pos* < *length treeList then membermima* (*treeList* ! *pos*) (*low x* (*deg div 2*)) *else False*))

lemma *length-mul-elem*:(∀ *x* ∈ *set xs*. *length x* = *n*) ⟹ *length* (*concat xs*) = (*length xs*) ∗ *n*
⟨*proof*⟩

We combine both auxiliary functions: The following test returns true if and only if an element occurs in the tree with respect to our interpretation no matter where it is stored.

definition *both-member-options* :: *VEBT* ⇒ *nat* ⇒ *bool* **where**
both-member-options t x = (*naive-member t x* ∨ *membermima t x*)

end
context begin
interpretation *VEBT-internal* ⟨*proof*⟩

definition *set-vebt* :: *VEBT* ⇒ *nat set* **where**
set-vebt t = {*x*. *both-member-options t x*}
end

1.5 Inductive Definition of semantically valid Vam Emde Boas Trees

Invariant for verification proofs

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

inductive *invar-vebt*::*VEBT* ⇒ *nat* ⇒ *bool* **where**

invar-vebt (Leaf a b) (Suc 0) |
(∀ t ∈ set treeList. invar-vebt t n) ⟹ invar-vebt summary m ⟹ length treeList = 2^m
⟹ m = n ⟹ deg = n + m ⟹ (∄ i. both-member-options summary i)
⟹(∀ t ∈ set treeList. ∄ x. both-member-options t x)
⟹ invar-vebt (Node None deg treeList summary) deg|
(∀ t ∈ set treeList. invar-vebt t n) ⟹ invar-vebt summary m
⟹ length treeList = 2^m ⟹ m = Suc n ⟹ deg = n + m ⟹ (∄ i. both-member-options summary i)
⟹ (∀ t ∈ set treeList. ∄ x. both-member-options t x)
⟹ invar-vebt (Node None deg treeList summary) deg|
(∀ t ∈ set treeList. invar-vebt t n) ⟹ invar-vebt summary m ⟹ length treeList = 2^m ⟹ m = n
⟹deg = n + m⟹ (∀ i < 2^m. (∃ x. both-member-options (treeList ! i) x) ⟷ (both-member-options summary i)) ⟹
(mi = ma ⟶ (∀ t ∈ set treeList. ∄ x. both-member-options t x)) ⟹
mi ≤ ma ⟹ ma < 2^deg ⟹
(mi ≠ ma ⟶
(∀ i < 2^m.
(high ma n = i ⟶ both-member-options (treeList ! i) (low ma n)) ∧
(∀ x. (high x n = i ∧ both-member-options (treeList ! i) (low x n)
) ⟶ mi < x ∧ x ≤ ma)))
⟹ invar-vebt (Node (Some (mi, ma)) deg treeList summary) deg|
(∀ t ∈ set treeList. invar-vebt t n) ⟹invar-vebt summary m ⟹ length treeList = 2^m
⟹ m = Suc n ⟹deg = n + m⟹(∀ i < 2^m. (∃ x. both-member-options (treeList ! i) x) ⟷ (both-member-options summary i)) ⟹
(mi = ma ⟶ (∀ t ∈ set treeList. ∄ x. both-member-options t x)) ⟹
mi ≤ ma ⟹ ma < 2^deg ⟹
(mi ≠ ma ⟶
(∀ i < 2^m.
(high ma n = i ⟶ both-member-options (treeList ! i) (low ma n)) ∧
(∀ x. (high x n = i ∧ both-member-options (treeList ! i) (low x n)
) ⟶ mi < x ∧ x ≤ ma)))
⟹ invar-vebt (Node (Some (mi, ma)) deg treeList summary) deg

end

context *VEBT-internal* **begin**

definition *in-children n treeList x ≡ both-member-options (treeList ! high x n) (low x n)*

functional validness definition

fun *valid′ :: VEBT ⇒ nat ⇒ bool* **where**
valid′ (Leaf - -) d ⟷ d=1
| valid′ (Node mima deg treeList summary) deg′ ⟷
(
deg=deg′ ∧ (
let n = deg div 2; m = deg − n in
(∀ t ∈ set treeList. valid′ t n)
∧ valid′ summary m

$\wedge$ *length treeList* = *2^m*
$\wedge$ (
case mima of
None $\Rightarrow$ ($\nexists$ *i. both-member-options summary i*) $\wedge$ ($\forall$ *t* $\in$ *set treeList.* $\nexists$ *x. both-member-options t x*)
| *Some* (*mi*,*ma*) $\Rightarrow$
mi $\leq$ *ma* $\wedge$ *ma*<*2^deg*
$\wedge$ ($\forall$ *i* < *2^m.* ($\exists$ *x. both-member-options* (*treeList* ! *i*) *x*) $\longleftrightarrow$ (*both-member-options summary i*))
$\wedge$ (*if mi*=*ma then* ($\forall$ *t* $\in$ *set treeList.* $\nexists$ *x. both-member-options t x*)
else
in-children n treeList ma
$\wedge$ ($\forall$ *x* < *2^deg. in-children n treeList x* $\longrightarrow$ *mi*<*x* $\wedge$ *x*$\leq$*ma*)
)
)
)
)

equivalence proofs

lemma *high-bound-aux*: *ma* < *2^(n+m)* $\Longrightarrow$ *high ma n* < *2^m*
⟨*proof*⟩

lemma *valid-eq1*:
assumes *invar-vebt t d*
shows *valid′ t d*
⟨*proof*⟩

lemma *even-odd-cases*:
fixes *x* :: *nat*
obtains *n* **where** *x*=*n*+*n* | *n* **where** *x* = *n* + *Suc n*
⟨*proof*⟩

lemma *valid-eq2*: *valid′ t d* $\Longrightarrow$ *invar-vebt t d*
⟨*proof*⟩

lemma *valid-eq*: *valid′ t d* $\longleftrightarrow$ *invar-vebt t d*
⟨*proof*⟩

lemma [*termination-simp*]: **assumes** *odd* (*v*::*nat*) **shows** *v div 2* < *v*
⟨*proof*⟩

lemma [*termination-simp*]:**assumes** *n* > *1* **and** *odd n* **shows** *Suc* (*n div 2*) < *n*
⟨*proof*⟩

end

1.6 Function for generating an empty tree of arbitrary degree respectively order

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

fun *vebt-buildup* :: *nat* ⇒ *VEBT* **where**
  vebt-buildup 0 = *Leaf False False*|
  vebt-buildup (*Suc 0*) = *Leaf False False*|
  vebt-buildup n = (*if even n then* (*let half* = *n div 2 in*
    Node None n (*replicate* (*2^half*) (*vebt-buildup half*)) (*vebt-buildup half*))
   else (*let half* = *n div 2 in*
    Node None n (*replicate* (*2^*(*Suc half*)) (*vebt-buildup half*)) (*vebt-buildup* (*Suc half*))))

end

context *VEBT-internal* **begin**

lemma *buildup-nothing-in-leaf*: ¬ *naive-member* (*vebt-buildup n*) *x*
⟨*proof*⟩

lemma *buildup-nothing-in-min-max*:¬ *membermima* (*vebt-buildup n*) *x*
⟨*proof*⟩

The empty tree generated by $vebt_buildup$ is indeed a valid tree.

lemma *buildup-gives-valid*: *n>0* ⟹ *invar-vebt* (*vebt-buildup n*) *n*
⟨*proof*⟩

lemma *mi-ma-2-deg*: **assumes** *invar-vebt* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *n* **shows** *mi≤ ma* ∧ *ma* < *2^deg*
⟨*proof*⟩

lemma *deg-not-0*: *invar-vebt t n* ⟹ *n* > *0*
⟨*proof*⟩

lemma *set-n-deg-not-0*:**assumes** ∀ *t*∈*set treeList*. *invar-vebt t n***and** *length treeList* = *2^m* **shows** *n* ≥ *1*
⟨*proof*⟩

lemma *both-member-options-ding*: **assumes***invar-vebt* (*Node info deg treeList summary*) *n* **and** *x<2^deg***and** *both-member-options* (*treeList* ! (*high x* (*deg div 2*))) (*low x* (*deg div 2*))**shows** *both-member-options* (*Node info deg treeList summary*) *x*
⟨*proof*⟩

lemma *exp-split-high-low*: **assumes** *x* < *2^*(*n*+*m*) **and** *n* > *0* **and** *m*> *0*
shows *high x n* < *2^m* **and** *low x n* < *2^n*
⟨*proof*⟩

lemma *low-inv*: **assumes** *x*< *2^n* **shows** *low* (*y*∗*2^n* + *x*) *n* = *x* ⟨*proof*⟩

lemma *high-inv*: **assumes** $x < 2\hat{}n$ **shows** *high* $(y*2\hat{}n + x)$ $n = y$ ⟨*proof*⟩

lemma *both-member-options-from-chilf-to-complete-tree*:
assumes *high x* (*deg div 2*) < *length treeList* **and** *deg* ≥*1* **and** *both-member-options* (*treeList* ! (*high x* (*deg div 2*))) (*low x* (*deg div 2*))
shows *both-member-options* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x*
⟨*proof*⟩

lemma *both-member-options-from-complete-tree-to-child*:
assumes *deg* ≥*1* **and** *both-member-options* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x*
shows *both-member-options* (*treeList* ! (*high x* (*deg div 2*))) (*low x* (*deg div 2*)) ∨ *x* = *mi* ∨ *x* = *ma*
⟨*proof*⟩

lemma *pow-sum*: (*divide*::*nat* ⇒ *nat* ⇒ *nat*) ((*2*::*nat*) ^((*a*::*nat*)+(*b*::*nat*))) (*2*^*a*) = *2*^*b*
⟨*proof*⟩

fun *elim-dead*::*VEBT* ⇒ *enat* ⇒ *VEBT* **where**
elim-dead (*Leaf a b*) - = *Leaf a b* |
elim-dead (*Node info deg treeList summary*) ∞ =
(*Node info deg* (*map* (λ *t*. *elim-dead t* (*enat* (*2*^(*deg div 2*)))) *treeList*)
(*elim-dead summary* ∞))|
elim-dead (*Node info deg treeList summary*) (*enat l*) =
(*Node info deg* (*take* (*l div* (*2*^(*deg div 2*))) (*map* (λ *t*. *elim-dead t* (*enat* (*2*^(*deg div 2*))))*treeList*))
(*elim-dead summary* ((*enat* (*l div* (*2*^(*deg div 2*)))))))

lemma *elimnum*: *invar-vebt* (*Node info deg treeList summary*) *n* ⟹
elim-dead (*Node info deg treeList summary*) (*enat* ((*2*::*nat*)^*n*)) = (*Node info deg treeList summary*)
⟨*proof*⟩

lemma *elimcomplete*: *invar-vebt* (*Node info deg treeList summary*) *n* ⟹
elim-dead (*Node info deg treeList summary*) ∞ = (*Node info deg treeList summary*)
⟨*proof*⟩

end
end

theory *VEBT-Member* **imports** *VEBT-Definitions*
begin

2 Member Function

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

fun *vebt-member* :: *VEBT* ⇒ *nat* ⇒ *bool* **where**
vebt-member (*Leaf a b*) *x* = (*if x* = *0 then a else if x* = *1 then b else False*)|
vebt-member (*Node None - - -*) *x* = *False*|

vebt-member (*Node* - *0* - -) *x* = *False*|
vebt-member (*Node* - (*Suc 0*) - -) *x* = *False*|
vebt-member (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (
if x = *mi then True else*
if x = *ma then True else*
if x < *mi then False else*
if x > *ma then False else* (
let h = *high x* (*deg div 2*);
l = *low x* (*deg div 2*) *in*(
if h < *length treeList*
then vebt-member (*treeList* ! *h*) *l*
else False)))

end

context *VEBT-internal* **begin**

lemma *member-inv*:
assumes *vebt-member* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x*
shows *deg* ≥ *2* ∧
(*x* = *mi* ∨ *x* = *ma* ∨ (*x* < *ma* ∧ *x* > *mi* ∧ *high x* (*deg div 2*) < *length treeList* ∧
vebt-member (*treeList* ! (*high x* (*deg div 2*))) (*low x* (*deg div 2*))))
⟨*proof*⟩

definition *bit-concat*::*nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
bit-concat h l d = *h*∗*2*^*d* + *l*

lemma *bit-split-inv*: *bit-concat* (*high x d*) (*low x d*) *d* = *x*
⟨*proof*⟩

definition *set-vebt'*::*VEBT* ⇒ *nat set* **where**
set-vebt' t = {*x*. *vebt-member t x*}

lemma *Leaf-0-not*: **assumes** *invar-vebt* (*Leaf a b*) *0* **shows** *False*
⟨*proof*⟩

lemma *valid-0-not*: *invar-vebt t 0* ⟹ *False*
⟨*proof*⟩

theorem *valid-tree-deg-neq-0*: (¬ *invar-vebt t 0*)
⟨*proof*⟩

lemma *deg-1-Leafy*: *invar-vebt t n* ⟹ *n* = *1* ⟹ ∃ *a b*. *t* = *Leaf a b*
⟨*proof*⟩

lemma *deg-1-Leaf*: *invar-vebt t 1* ⟹ ∃ *a b*. *t* = *Leaf a b*
⟨*proof*⟩

corollary *deg1Leaf*: *invar-vebt t 1* $\longleftrightarrow$ ($\exists$ *a b. t* = *Leaf a b*)
⟨*proof*⟩

lemma *deg-SUcn-Node*: **assumes** *invar-vebt tree* (*Suc* (*Suc n*)) **shows**
$\exists$ *info treeList s. tree* = *Node info* (*Suc* (*Suc n*)) *treeList s*
⟨*proof*⟩

lemma *invar-vebt* (*Node info deg treeList summary*) *deg* $\Longrightarrow$ *deg* > *1*
⟨*proof*⟩

lemma *deg-deg-n*: **assumes** *invar-vebt* (*Node info deg treeList summary*) *n* **shows** *deg* = *n*
⟨*proof*⟩

lemma *member-valid-both-member-options*:
invar-vebt tree n $\Longrightarrow$ *vebt-member tree x* $\Longrightarrow$ (*naive-member tree x* $\vee$ *membermima tree x*)
⟨*proof*⟩

lemma *member-bound*: *vebt-member tree x* $\Longrightarrow$ *invar-vebt tree n* $\Longrightarrow$ *x* < *2^n*
⟨*proof*⟩

theorem *inrange*: **assumes** *invar-vebt t n* **shows** *set-vebt' t* $\subseteq$ {*0..2^n−1*}
⟨*proof*⟩

theorem *buildup-gives-empty*: *set-vebt'* (*vebt-buildup n*) = {}
⟨*proof*⟩

fun *minNull*::*VEBT* ⇒ *bool* **where**
minNull (*Leaf False False*) = *True*|
minNull (*Leaf - -*) = *False*|
minNull (*Node None - - -*) = *True*|
minNull (*Node* (*Some -*) *- - -*) = *False*

lemma *min-Null-member*: *minNull t* $\Longrightarrow$ ¬ *vebt-member t x*
⟨*proof*⟩

lemma *not-min-Null-member*: ¬ *minNull t* $\Longrightarrow$ $\exists$ *y. both-member-options t y*
⟨*proof*⟩

lemma *valid-member-both-member-options*: *invar-vebt t n* $\Longrightarrow$ *both-member-options t x* $\Longrightarrow$ *vebt-member t x*
⟨*proof*⟩

corollary *both-member-options-equiv-member*: **assumes** *invar-vebt t n*
shows *both-member-options t x* $\longleftrightarrow$ *vebt-member t x*
⟨*proof*⟩

lemma *member-correct*: *invar-vebt t n* $\Longrightarrow$ *vebt-member t x* $\longleftrightarrow$ *x* $\in$ *set-vebt t*
⟨*proof*⟩

corollary *set-vebt-set-vebt′-valid*: **assumes** *invar-vebt t n* **shows** *set-vebt t =set-vebt′ t*
⟨*proof*⟩

lemma *set-vebt-finite*: *invar-vebt t n* ⟹ *finite* (*set-vebt′ t*)
⟨*proof*⟩

lemma *mi-eq-ma-no-ch*:**assumes** *invar-vebt* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *deg* **and** *mi* = *ma*
shows (∀ *t* ∈ *set treeList*. ∄ *x*. *both-member-options t x*) ∧ (∄ *x*. *both-member-options summary x*)
⟨*proof*⟩

end
end

theory *VEBT-Insert* **imports** *VEBT-Member*
begin

3 Insert Function

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

fun *vebt-insert* :: *VEBT* ⇒ *nat* ⇒*VEBT* **where**
vebt-insert (*Leaf a b*) *x* = (*if x=0 then Leaf True b else if x = 1 then Leaf a True else Leaf a b*)|
vebt-insert (*Node info 0 ts s*) *x* = (*Node info 0 ts s*)|
vebt-insert (*Node info* (*Suc 0*) *ts s*) *x* = (*Node info* (*Suc 0*) *ts s*)|
vebt-insert (*Node None* (*Suc deg*) *treeList summary*) *x* =
(*Node* (*Some* (*x*,*x*)) (*Suc deg*) *treeList summary*)|
vebt-insert (*Node* (*Some* (*mi*,*ma*)) *deg treeList summary*) *x* = (
let xn = (*if x* < *mi then mi else x*);
minn = (*if x* < *mi then x else mi*);
l = *low xn* (*deg div 2*);
h = *high xn* (*deg div 2*) *in* (
if h < *length treeList* ∧ ¬ (*x* = *mi* ∨ *x* = *ma*) *then*
Node (*Some* (*minn*, *max xn ma*)) *deg* (*treeList*[*h*:= *vebt-insert* (*treeList* ! *h*) *l*])
(*if minNull* (*treeList* ! *h*) *then vebt-insert summary h else summary*)
else (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*)))

end

context *VEBT-internal* **begin**

lemma *insert-simp-norm*:
assumes *high x* (*deg div 2*) < *length treeList* **and** (*mi*::*nat*)< *x* **and** *deg*≥ *2* **and** *x* ≠ *ma*
shows *vebt-insert* (*Node* (*Some* (*mi*,*ma*)) *deg treeList summary*) *x* =
Node (*Some* (*mi*, *max x ma*)) *deg* (*treeList* [(*high x* (*deg div 2*)):= *vebt-insert* (*treeList* ! (*high x* (*deg div 2*))) (*low x* (*deg div 2*))])
(*if minNull* (*treeList* ! (*high x* (*deg div 2*))) *then vebt-insert summary* (*high x* (*deg*

div 2)) *else summary*)
⟨*proof*⟩

lemma *insert-simp-excp*:
assumes *high mi* (*deg div 2*) < *length treeList* **and** (*x*::*nat*) < *mi* **and** *deg*≥ *2* **and** *x* ≠ *ma*
shows *vebt-insert* (*Node* (*Some* (*mi*,*ma*)) *deg treeList summary*) *x* =
Node (*Some* (*x*, *max mi ma*)) *deg* (*treeList*[(*high mi* (*deg div 2*)) := *vebt-insert* (*treeList* ! (*high mi* (*deg div 2*))) (*low mi* (*deg div 2*))])
(*if minNull* (*treeList* ! (*high mi* (*deg div 2*))) *then vebt-insert summary* (*high mi* (*deg div 2*)) *else summary*)
⟨*proof*⟩

lemma *insert-simp-mima*: **assumes** *x* = *mi* ∨ *x* = *ma* **and** *deg* ≥ *2*
shows *vebt-insert* (*Node* (*Some* (*mi*,*ma*)) *deg treeList summary*) *x* = (*Node* (*Some* (*mi*,*ma*)) *deg treeList summary*)
⟨*proof*⟩

lemma *valid-insert-both-member-options-add*: *invar-vebt t n* ⟹ *x*< *2*^*n* ⟹ *both-member-options* (*vebt-insert t x*) *x*
⟨*proof*⟩

lemma *valid-insert-both-member-options-pres*: *invar-vebt t n* ⟹ *x*<*2*^*n* ⟹ *y* < *2*^*n* ⟹ *both-member-options t x*
⟹ *both-member-options* (*vebt-insert t y*) *x*
⟨*proof*⟩

lemma *post-member-pre-member*:*invar-vebt t n* ⟹ *x*< *2*^*n* ⟹ *y* <*2*^*n* ⟹ *vebt-member* (*vebt-insert t x*) *y* ⟹ *vebt-member t y* ∨ *x* = *y*
⟨*proof*⟩

end
end

theory *VEBT-InsertCorrectness* **imports** *VEBT-Member VEBT-Insert*
begin

context *VEBT-internal* **begin**

4 Correctness of the Insert Operation

4.1 Validness Preservation

theorem *valid-pres-insert*: *invar-vebt t n* ⟹ *x*< *2*^*n* ⟹ *invar-vebt* (*vebt-insert t x*) *n*
⟨*proof*⟩

4.2 Correctness with Respect to Set Interpretation

theorem *insert-corr*:
 assumes *invar-vebt t n* **and** $x < 2\hat{}n$
 shows *set-vebt′ t* ∪ {*x*} = *set-vebt′* (*vebt-insert t x*)
⟨*proof*⟩

corollary *insert-correct*: **assumes** *invar-vebt t n* **and** $x < 2\hat{}n$ **shows**
 set-vebt t ∪ {*x*} = *set-vebt* (*vebt-insert t x*)
 ⟨*proof*⟩

fun *insert′*::*VEBT* ⇒ *nat* ⇒ *VEBT* **where**
 insert′ (*Leaf a b*) *x* = *vebt-insert* (*Leaf a b*) *x*|
 insert′ (*Node info deg treeList summary*) *x* =
 (*if* $x \geq 2\hat{}deg$ *then* (*Node info deg treeList summary*)
 else vebt-insert (*Node info deg treeList summary*) *x*)

theorem *insert′-pres-valid*: **assumes** *invar-vebt t n* **shows** *invar-vebt* (*insert′ t x*) *n*
 ⟨*proof*⟩

theorem *insert′-correct*: **assumes** *invar-vebt t n*
 shows *set-vebt* (*insert′ t x*) = (*set-vebt t* ∪ {*x*})∩{*0..2^n−1*}
⟨*proof*⟩

end
end

theory *VEBT-MinMax* **imports** *VEBT-Member*
begin

5 The Minimum and Maximum Operation

fun *vebt-mint* :: *VEBT* ⇒ *nat option* **where**
 vebt-mint (*Leaf a b*) = (*if a then Some 0 else if b then Some 1 else None*)|
 vebt-mint (*Node None - - -*) = *None*|
 vebt-mint (*Node* (*Some* (*mi*,*ma*)) - - -) = *Some mi*

fun *vebt-maxt* :: *VEBT* ⇒ *nat option* **where**
 vebt-maxt (*Leaf a b*) = (*if b then Some 1 else if a then Some 0 else None*)|
 vebt-maxt (*Node None - - -*) = *None*|
 vebt-maxt (*Node* (*Some* (*mi*,*ma*)) - - -) = *Some ma*

context *VEBT-internal* **begin**

fun *option-shift*::(′*a*⇒′*a*⇒′*a*) ⇒′*a option* ⇒′*a option*⇒ ′*a option* **where**
option-shift - None - = *None*|
option-shift - - None = *None*|

option-shift f (*Some a*) (*Some b*) = *Some* (*f a b*)

definition *power*::*nat option* ⇒ *nat option* ⇒ *nat option* (**infixl**‹$\hat{}_o$› *81*) **where**
power= *option-shift* (^)

definition *add*::*nat option* ⇒ *nat option* ⇒ *nat option* (**infixl**‹$+_o$› *79*) **where**
add= *option-shift* (+)

definition *mul*::*nat option* ⇒ *nat option* ⇒ *nat option* (**infixl**‹$*_o$› *80*) **where**
mul = *option-shift* (∗)

fun *option-comp-shift*::(′*a* ⇒ ′*a* ⇒ *bool*) ⇒ ′*a option* ⇒ ′*a option* ⇒ *bool* **where**
option-comp-shift - None - = *False*|
option-comp-shift - - None = *False*|
option-comp-shift f (*Some x*) (*Some y*) = *f x y*

fun *less*::*nat option* ⇒ *nat option* ⇒ *bool* (**infixl**‹$<_o$› *80*) **where**
less x y= *option-comp-shift* (<) *x y*

fun *lesseq*::*nat option* ⇒ *nat option* ⇒ *bool* (**infixl**‹$\leq_o$› *80*) **where**
lesseq x y = *option-comp-shift* (≤) *x y*

fun *greater*::*nat option* ⇒ *nat option* ⇒ *bool* (**infixl**‹$>_o$› *80*) **where**
greater x y = *option-comp-shift* (>) *x y*

lemma *add-shift*:$x+y = z \longleftrightarrow$ *Some x* $+_o$ *Some y* = *Some z*
⟨*proof*⟩

lemma *mul-shift*:$x*y = z \longleftrightarrow$ *Some x* $*_o$ *Some y* = *Some z* ⟨*proof*⟩

lemma *power-shift*:$x\hat{}y = z \longleftrightarrow$ *Some x* $\hat{}_o$ *Some y* = *Some z* ⟨*proof*⟩

lemma *less-shift*: $x < y \longleftrightarrow$ *Some x* $<_o$ *Some y* ⟨*proof*⟩

lemma *lesseq-shift*: $x \leq y \longleftrightarrow$ *Some x* $\leq_o$ *Some y* ⟨*proof*⟩

lemma *greater-shift*: $x > y \longleftrightarrow$ *Some x* $>_o$ *Some y* ⟨*proof*⟩

definition *max-in-set* :: *nat set* ⇒ *nat* ⇒ *bool* **where**
max-in-set xs x ⟷ ($x \in xs \wedge (\forall\ y \in xs.\ y \leq x)$)

lemma *maxt-member*: *invar-vebt t n* ⟹ *vebt-maxt t* = *Some maxi* ⟹ *vebt-member t maxi*
⟨*proof*⟩

lemma *maxt-corr-help*: *invar-vebt t n* ⟹ *vebt-maxt t* = *Some maxi* ⟹ *vebt-member t x* ⟹ *maxi* ≥ *x*
⟨*proof*⟩

lemma *maxt-corr-help-empty*: $invar\text{-}vebt\ t\ n \Longrightarrow vebt\text{-}maxt\ t = None \Longrightarrow set\text{-}vebt'\ t = \{\}$
⟨*proof*⟩

theorem *maxt-corr*:**assumes** $invar\text{-}vebt\ t\ n$ **and** $vebt\text{-}maxt\ t = Some\ x$ **shows** $max\text{-}in\text{-}set\ (set\text{-}vebt'\ t)\ x$
⟨*proof*⟩

theorem *maxt-sound*:**assumes** $invar\text{-}vebt\ t\ n$ **and** $max\text{-}in\text{-}set\ (set\text{-}vebt'\ t)\ x$ **shows** $vebt\text{-}maxt\ t = Some\ x$
⟨*proof*⟩

definition *min-in-set* :: *nat set* ⇒ *nat* ⇒ *bool* **where**
$min\text{-}in\text{-}set\ xs\ x \longleftrightarrow (x \in xs \land (\forall\ y \in xs.\ y \geq x))$

lemma *mint-member*: $invar\text{-}vebt\ t\ n \Longrightarrow vebt\text{-}mint\ t = Some\ maxi \Longrightarrow vebt\text{-}member\ t\ maxi$
⟨*proof*⟩

lemma *mint-corr-help*: $invar\text{-}vebt\ t\ n \Longrightarrow vebt\text{-}mint\ t = Some\ mini \Longrightarrow vebt\text{-}member\ t\ x \Longrightarrow mini \leq x$
⟨*proof*⟩

lemma *mint-corr-help-empty*: $invar\text{-}vebt\ t\ n \Longrightarrow vebt\text{-}mint\ t = None \Longrightarrow set\text{-}vebt'\ t = \{\}$
⟨*proof*⟩

theorem *mint-corr*:**assumes** $invar\text{-}vebt\ t\ n$ **and** $vebt\text{-}mint\ t = Some\ x$ **shows** $min\text{-}in\text{-}set\ (set\text{-}vebt'\ t)\ x$
⟨*proof*⟩

theorem *mint-sound*:**assumes** $invar\text{-}vebt\ t\ n$ **and** $min\text{-}in\text{-}set\ (set\text{-}vebt'\ t)\ x$ **shows** $vebt\text{-}mint\ t = Some\ x$
⟨*proof*⟩

lemma *summaxma*:**assumes** $invar\text{-}vebt\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ deg$ **and** $mi \neq ma$
shows $the\ (vebt\text{-}maxt\ summary) = high\ ma\ (deg\ div\ 2)$
⟨*proof*⟩

lemma *maxbmo*: $vebt\text{-}maxt\ t = Some\ x \Longrightarrow both\text{-}member\text{-}options\ t\ x$
⟨*proof*⟩

lemma *misiz*:$invar\text{-}vebt\ t\ n \Longrightarrow Some\ m = vebt\text{-}mint\ t \Longrightarrow m < 2\hat{}n$
⟨*proof*⟩

lemma *mintlistlength*: **assumes** $invar\text{-}vebt\ (Node\ (Some\ (mi,\ ma))\ deg\ treeList\ summary)\ n$
$mi \neq ma$ **shows** $ma > mi \land (\exists\ m.\ Some\ m = vebt\text{-}mint\ summary \land m < 2\hat{}(n - n\ div\ 2))$
⟨*proof*⟩

lemma *power-minus-is-div*:
$b \leq a \Longrightarrow (2 :: nat)\ \hat{}\ (a - b) = 2\ \hat{}\ a\ div\ 2\ \hat{}\ b$
⟨*proof*⟩

lemma *nested-mint*:**assumes** *invar-vebt* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *n* *n* = *Suc* (*Suc va*)
¬ *ma* < *mi* *ma* ≠ *mi* **shows**
high (*the* (*vebt-mint summary*) ∗ (2 ∗ 2 ^ (*va div* 2)) + *the* (*vebt-mint* (*treeList* ! *the* (*vebt-mint summary*)))) (*Suc* (*va div* 2))
< *length treeList*
⟨*proof*⟩

lemma *minminNull*: *vebt-mint t* = *None* ⟹ *minNull t*
⟨*proof*⟩

lemma *minNullmin*: *minNull t* ⟹ *vebt-mint t* = *None*
⟨*proof*⟩

end
end

theory *VEBT-Succ* **imports** *VEBT-Insert VEBT-MinMax*
begin

6 The Successor Operation

definition *is-succ-in-set* :: *nat set* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
is-succ-in-set xs x y = (*y* ∈ *xs* ∧ *y* > *x* ∧ (∀ *z* ∈ *xs*. (*z* > *x* ⟶ *z* ≥ *y*)))

context *VEBT-internal* **begin**

corollary *succ-member*: *is-succ-in-set* (*set-vebt′ t*) *x y* = (*vebt-member t y* ∧ *y* > *x* ∧ (∀ *z*. *vebt-member t z* ∧ *z* > *x* ⟶ *z* ≥ *y*))
⟨*proof*⟩

6.1 Auxiliary Lemmas on Sets and Successorship

lemma *finite* (*A*:: *nat set*) ⟹ *A* ≠ {}⟹ *Min A* ∈ *A*
⟨*proof*⟩

lemma *obtain-set-succ*: **assumes** (*x*::*nat*) < *z* **and** *max-in-set A z* **and** *finite B* **and** *A*=*B* **shows** ∃ *y*. *is-succ-in-set A x y*
⟨*proof*⟩

lemma *succ-none-empty*: **assumes** (∄ *x*. *is-succ-in-set* (*xs*) *a x*) **and** *finite xs***shows** ¬ (∃ *x* ∈ *xs*. *ord-class.greater x a*)
⟨*proof*⟩

end

6.2 The actual Function

context begin
  interpretation *VEBT-internal* ⟨*proof*⟩

```
fun vebt-succ :: VEBT ⇒ nat ⇒ nat option where
  vebt-succ (Leaf - b) 0 = (if b then Some 1 else None)|
  vebt-succ (Leaf - -) (Suc n) = None|
  vebt-succ (Node None - - -) - = None|
  vebt-succ (Node - 0 - -) - = None|
  vebt-succ (Node - (Suc 0) - -) - = None|
  vebt-succ (Node (Some (mi, ma)) deg treeList summary) x = (
        if x < mi then (Some mi)
        else (let l = low x (deg div 2); h = high x (deg div 2) in
             if h < length treeList then
                let maxlow = vebt-maxt (treeList ! h) in (
                   if maxlow ≠ None ∧ (Some l <_o maxlow) then
                       Some (2^(deg div 2)) *_o Some h +_o vebt-succ (treeList ! h) l
                   else let sc = vebt-succ summary h in
                        if sc = None then None
                        else Some (2^(deg div 2)) *_o sc +_o vebt-mint (treeList ! the sc) )
              else None))
```

end

6.3 Lemmas for Term Decomposition

context *VEBT-internal* **begin**

lemma *succ-min*: **assumes** $deg \geq 2$ **and** $(x::nat) < mi$ **shows**
  vebt-succ (Node (Some (mi, ma)) deg treeList summary) x = Some mi
  ⟨*proof*⟩

lemma *succ-greatereq-min*: **assumes** $deg \geq 2$ **and** $(x::nat) \geq mi$ **shows**

```
  vebt-succ (Node (Some (mi, ma)) deg treeList summary) x = (let l = low x (deg div 2); h = high x
(deg div 2) in
                    if h < length treeList then

                            let maxlow = vebt-maxt (treeList ! h) in
                            (if maxlow ≠ None ∧ (Some l <_o maxlow) then
                                                Some (2^(deg div 2)) *_o Some h +_o vebt-succ (treeList ! h) l
                             else let sc = vebt-succ summary h in
                             if sc = None then None
                             else Some (2^(deg div 2)) *_o sc +_o vebt-mint (treeList ! the sc) )

                     else None)
```
  ⟨*proof*⟩

lemma *succ-list-to-short*: **assumes** $deg \geq 2$ **and** $x \geq mi$ **and** *high x* $(deg\ div\ 2) \geq$ *length treeList*
shows
vebt-succ (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) x = *None*
⟨*proof*⟩

lemma *succ-less-length-list*: **assumes** $deg \geq 2$ **and** $x \geq mi$ **and** *high x* $(deg\ div\ 2) <$ *length treeList*
shows
vebt-succ (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) x =
(*let l* = *low x* (*deg div 2*); *h* = *high x* (*deg div 2*) ; *maxlow* = *vebt-maxt* (*treeList* ! *h*) *in*
(*if maxlow* ≠ *None* ∧ (*Some l* $<_o$ *maxlow*) *then*
Some (2^(*deg div 2*)) $*_o$ *Some h* $+_o$ *vebt-succ* (*treeList* ! *h*) *l*
else let sc = *vebt-succ summary h in*
if sc = *None then None*
else Some (2^(*deg div 2*)) $*_o$ *sc* $+_o$ *vebt-mint* (*treeList* !*the sc*)))
⟨*proof*⟩

6.4 Correctness Proof

theorem *succ-corr*: *invar-vebt t n* ⟹ *vebt-succ t x* = *Some sx* == *is-succ-in-set* (*set-vebt′ t*) *x sx*
⟨*proof*⟩

corollary *succ-empty*: **assumes** *invar-vebt t n*
shows (*vebt-succ t x* = *None*) = ({*y*. *vebt-member t y* ∧ *y* > *x*} = {})
⟨*proof*⟩

theorem *succ-correct*: *invar-vebt t n* ⟹ *vebt-succ t x* = *Some sx* ⟷ *is-succ-in-set* (*set-vebt t*) *x sx*
⟨*proof*⟩

lemma *is-succ-in-set S x y* ⟷ *min-in-set* {*s* . *s* ∈ *S* ∧ *s* > *x*} *y*
⟨*proof*⟩

lemma *helpyd*:*invar-vebt t n* ⟹ *vebt-succ t x* = *Some y* ⟹ *y* < 2^*n*
⟨*proof*⟩

lemma *geqmaxNone*:
assumes *invar-vebt* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *n* $x \geq ma$
shows *vebt-succ* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = *None*
⟨*proof*⟩

end
end

theory *VEBT-Pred* **imports** *VEBT-MinMax VEBT-Insert*
begin

7 The Predecessor Operation

definition *is-pred-in-set* :: *nat set* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**

is-pred-in-set xs x y = $(y \in xs \wedge y < x \wedge (\forall\ z \in xs.\ (z < x \longrightarrow z \leq y)))$

context *VEBT-internal* **begin**

7.1 Lemmas on Sets and Predecessorship

corollary *pred-member*: *is-pred-in-set* (*set-vebt′ t*) *x y* = (*vebt-member t y* ∧ *y* < *x* ∧ (∀ *z*. *vebt-member t z* ∧ *z* < *x* ⟶ *z* ≤ *y*))
⟨*proof*⟩

lemma *finite* (*A*:: *nat set*) ⟹ *A* ≠ {}⟹ *Max A* ∈ *A*
⟨*proof*⟩

lemma *obtain-set-pred*: **assumes** (*x*::*nat*) > *z* **and** *min-in-set A z* **and** *finite A* **shows** ∃ *y*. *is-pred-in-set A x y*
⟨*proof*⟩

lemma *pred-none-empty*: **assumes** (∄ *x*. *is-pred-in-set* (*xs*) *a x*) **and** *finite xs***shows** ¬ (∃ *x* ∈ *xs*. *ord-class.less x a*)
⟨*proof*⟩

end

7.2 The actual Function for Predecessor Search

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

fun *vebt-pred* :: *VEBT* ⇒ *nat* ⇒ *nat option* **where**
vebt-pred (*Leaf - -*) *0* = *None*|
vebt-pred (*Leaf a -*) (*Suc 0*) = (*if a then Some 0 else None*)|
vebt-pred (*Leaf a b*) *-* = (*if b then Some 1 else if a then Some 0 else None*)|
vebt-pred (*Node None - - -*) *-* = *None*|
vebt-pred (*Node - 0 - -*) *-* = *None*|
vebt-pred (*Node -* (*Suc 0*) *- -*) *-* = *None*|
vebt-pred (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (
    if x > ma then Some ma
    else (*let l* = *low x* (*deg div 2*); *h* = *high x* (*deg div 2*) *in*
        if h < length treeList then
            let minlow = *vebt-mint* (*treeList* ! *h*) *in* (
                if minlow ≠ None ∧ (*Some l* $>_o$ *minlow*) *then*
                    Some (*2^*(*deg div 2*)) $*_o$ *Some h* $+_o$ *vebt-pred* (*treeList* ! *h*) *l*
                else let pr = *vebt-pred summary h in*
                        if pr = *None then* (
                            if x > mi then Some mi
                            else None)
                        else Some (*2^*(*deg div 2*)) $*_o$ *pr* $+_o$ *vebt-maxt* (*treeList* ! *the pr*))
        else None))

end

context *VEBT-internal* **begin**

7.3 Auxiliary Lemmas

lemma *pred-max*:
  assumes $deg \geq 2$ **and** $(x::nat) > ma$
  shows *vebt-pred (Node (Some (mi, ma)) deg treeList summary) x = Some ma*
  ⟨*proof*⟩

lemma *pred-lesseq-max*:
  assumes $deg \geq 2$ **and** $(x::nat) \leq ma$
  shows *vebt-pred (Node (Some (mi, ma)) deg treeList summary) x = (let l = low x (deg div 2); h = high x (deg div 2) in*
if h < length treeList then

let minlow = vebt-mint (treeList ! h) in
(if minlow ≠ None ∧ (Some l $>_o$ *minlow) then*
Some (2^(deg div 2)) $*_o$ *Some h* $+_o$ *vebt-pred (treeList ! h) l*
else let pr = vebt-pred summary h in
if pr = None then (if x > mi then Some mi else None)
else Some (2^(deg div 2)) $*_o$ *pr* $+_o$ *vebt-maxt (treeList ! the pr))*

else None)
  ⟨*proof*⟩

lemma *pred-list-to-short*:
  assumes $deg \geq 2$ **and** *ord-class.less-eq x ma* **and** *high x (deg div 2) ≥ length treeList*
  shows *vebt-pred (Node (Some (mi, ma)) deg treeList summary) x = None*
  ⟨*proof*⟩

lemma *pred-less-length-list*:
  assumes $deg \geq 2$ **and** *ord-class.less-eq x ma* **and** *high x (deg div 2) < length treeList*
  shows
  vebt-pred (Node (Some (mi, ma)) deg treeList summary) x = (let l = low x (deg div 2); h = high x (deg div 2); minlow = vebt-mint (treeList ! h) in
(if minlow ≠ None ∧ (Some l $>_o$ *minlow) then*
Some (2^(deg div 2)) $*_o$ *Some h* $+_o$ *vebt-pred (treeList ! h) l*
else let pr = vebt-pred summary h in
if pr = None then (if x > mi then Some mi else None)
else Some (2^(deg div 2)) $*_o$ *pr* $+_o$ *vebt-maxt (treeList ! the pr)))*
  ⟨*proof*⟩

7.4 Correctness Proof

theorem *pred-corr*: *invar-vebt t n* ⟹ *vebt-pred t x = Some px == is-pred-in-set (set-vebt′ t) x px*
⟨*proof*⟩

corollary *pred-empty*: **assumes** *invar-vebt t n*

shows (*vebt-pred t x = None*) = ({*y*. *vebt-member t y* ∧ *y* < *x*} = {})
⟨*proof*⟩

theorem *pred-correct*: *invar-vebt t n* ⟹ *vebt-pred t x = Some sx* ⟷ *is-pred-in-set* (*set-vebt t*) *x sx*
⟨*proof*⟩

lemma *helpypredd*:*invar-vebt t n* ⟹ *vebt-pred t x = Some y* ⟹ *y* < *2^n*
⟨*proof*⟩

lemma *invar-vebt t n* ⟹ *vebt-pred t x = Some y* ⟹ *y* < *x*
⟨*proof*⟩

end
end

theory *VEBT-Delete* **imports** *VEBT-Pred VEBT-Succ*
begin

8 Deletion

8.1 Function Definition

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

```
fun vebt-delete :: VEBT ⇒ nat ⇒ VEBT where
  vebt-delete (Leaf a b) 0 = Leaf False b|
  vebt-delete (Leaf a b) (Suc 0) = Leaf a False|
  vebt-delete (Leaf a b) (Suc (Suc n)) = Leaf a b|
  vebt-delete (Node None deg treeList summary) - = (Node None deg treeList summary)|
  vebt-delete (Node (Some (mi, ma)) 0 trLst smry) x = (Node (Some (mi, ma)) 0 trLst smry) |
  vebt-delete (Node (Some (mi, ma)) (Suc 0) tr sm) x = (Node (Some (mi, ma)) (Suc 0) tr sm) |
  vebt-delete (Node (Some (mi, ma)) deg treeList summary) x =(
          if (x < mi ∨ x > ma) then (Node (Some (mi, ma)) deg treeList summary)
          else if (x = mi ∧ x = ma) then (Node None deg treeList summary)
          else let xn = (if x = mi
                      then the (vebt-mint summary) ∗ 2^(deg div 2)
                           + the (vebt-mint (treeList ! the (vebt-mint summary)))
                       else x);
                  minn = (if x = mi then xn else mi);
                  l = low xn (deg div 2);
                  h = high xn (deg div 2) in
                  if h < length treeList
                  then(
                      let newnode = vebt-delete (treeList ! h) l;
                          newlist = treeList[h:= newnode]in
                          if minNull newnode
                          then( let sn = vebt-delete summary h in(
                               Node (Some (minn, if xn  = ma then
```

(*let maxs* = *vebt-maxt sn in* (
if maxs = *None*
then minn
else $2^\frown(deg\ div\ 2) * the\ maxs$
+ *the* (*vebt-maxt* (*newlist* ! *the maxs*))))
else ma)) *deg newlist sn*))
else (*Node* (*Some* (*minn*, (*if xn* = *ma*
then $h * 2^\frown(deg\ div\ 2)$ + *the*(*vebt-maxt* (*newlist* ! *h*))
else ma))) *deg newlist summary*))
else (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*))

end

8.2 Auxiliary Lemmas

context *VEBT-internal* **begin**

context begin

lemma *delt-out-of-range*:
assumes $x < mi \lor x > ma$ **and** $deg \geq 2$
shows
vebt-delete (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* =(*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*)
⟨*proof*⟩

lemma *del-single-cont*:
assumes $x = mi \land x = ma$ **and** $deg \geq 2$
shows *vebt-delete* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (*Node None deg treeList summary*)
⟨*proof*⟩

lemma *del-in-range*:
assumes $x \geq mi \land x \leq ma$ **and** $mi \neq ma$ **and** $deg \geq 2$
shows
vebt-delete (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (*let xn* = (*if x* = *mi*
then the (*vebt-mint summary*) $* 2^\frown(deg\ div\ 2)$
+ *the* (*vebt-mint* (*treeList* ! *the* (*vebt-mint summary*)))
else x);
minn = (*if x* = *mi then xn else mi*);
l = *low xn* (*deg div 2*);
h = *high xn* (*deg div 2*) *in*
if h < *length treeList*
then(
let newnode = *vebt-delete* (*treeList* ! *h*) *l*;
newlist = *treeList*[*h*:= *newnode*] *in*
if minNull newnode
then(
let sn = *vebt-delete summary h in*

```
(Node (Some (minn, if xn  = ma then (let maxs = vebt-maxt sn in
                                      (if maxs = None
                                         then minn
                                         else 2^(deg div 2) * the maxs
                                             + the (vebt-maxt (newlist ! the maxs))
                                      )
                                  )
                              else ma))
            deg newlist sn)
        )else
          (Node (Some (minn, (if xn = ma then
                             h * 2^(deg div 2) + the( vebt-maxt (newlist ! h))
                                    else ma)))
            deg newlist summary )
      )else
        (Node (Some (mi, ma)) deg treeList summary))
⟨proof⟩
```

lemma *del-x-not-mia*:
assumes $x > mi \wedge x \leq ma$ **and** $mi \neq ma$ **and** $deg \geq 2$ **and** *high x (deg div 2) = h* **and** *low x (deg div 2) = l***and** *high x (deg div 2) < length treeList*
shows

```
 vebt-delete (Node (Some (mi, ma)) deg treeList summary) x = (
              let  newnode = vebt-delete (treeList ! h) l;
                       newlist = treeList[h:= newnode] in
                      if minNull newnode
                      then(
                         let sn = vebt-delete summary h in
                        (Node (Some (mi, if x  = ma then (let maxs = vebt-maxt sn in
                                                          (if maxs = None
                                                             then mi
                                                             else 2^(deg div 2) * the maxs
                                                                 + the (vebt-maxt (newlist ! the maxs))
                                                          )
                                                      )
                                                  else ma))
                                deg newlist sn)
                      )else
                        (Node (Some (mi, (if x = ma then
                                         h * 2^(deg div 2) + the( vebt-maxt (newlist ! h))
                                                else ma)))
                          deg newlist summary )
)
  ⟨proof⟩
```

lemma *del-x-not-mi*:
assumes $x > mi \wedge x \leq ma$ **and** $mi \neq ma$ **and** $deg \geq 2$ **and** *high x (deg div 2) = h* **and** *low x (deg div 2) = l***and** *newnode = vebt-delete (treeList ! h) l*
and *newlist = treeList[h:= newnode]* **and** *high x (deg div 2) < length treeList*

shows
vebt-delete (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (
if minNull newnode
then(
let sn = *vebt-delete summary h in*
(*Node* (*Some* (*mi*, *if x* = *ma then* (*let maxs* = *vebt-maxt sn in*
(*if maxs* = *None*
then mi
else 2^(*deg div* 2) ∗ *the maxs*
+ *the* (*vebt-maxt* (*newlist* ! *the maxs*))
)
)
else ma))
deg newlist sn)
)*else*
(*Node* (*Some* (*mi*, (*if x* = *ma then*
h ∗ 2^(*deg div* 2) + *the*(*vebt-maxt* (*newlist* ! *h*))
else ma)))
deg newlist summary)
) ⟨*proof*⟩

lemma *del-x-not-mi-new-node-nil*:
assumes $x > mi \wedge x \leq ma$ **and** $mi \neq ma$ **and** $deg \geq 2$ **and** *high x* (*deg div* 2) = *h* **and**
low x (*deg div* 2) = *l***and** *newnode* = *vebt-delete* (*treeList* ! *h*) *l* **and** *minNull newnode* **and**
sn = *vebt-delete summary h* **and** *newlist* =*treeList*[*h*:= *newnode*] **and** *high x* (*deg div* 2) < *length treeList*
shows
vebt-delete (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (*Node* (*Some* (*mi*, *if x* = *ma then* (*let maxs* = *vebt-maxt sn in*
(*if maxs* = *None*
then mi
else 2^(*deg div* 2) ∗ *the maxs*
+ *the* (*vebt-maxt* (*newlist* ! *the maxs*))
)
)
else ma)) *deg newlist sn*)
⟨*proof*⟩

lemma *del-x-not-mi-newnode-not-nil*:
assumes $x > mi \wedge x \leq ma$ **and** $mi \neq ma$ **and** $deg \geq 2$ **and** *high x* (*deg div* 2) = *h* **and**
low x (*deg div* 2) = *l***and** *newnode* = *vebt-delete* (*treeList* ! *h*) *l* **and** ¬ *minNull newnode* **and**
newlist = *treeList*[*h*:= *newnode*] **and***high x* (*deg div* 2) < *length treeList*
shows
vebt-delete (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* =
(*Node* (*Some* (*mi*, (*if x* = *ma then*
h ∗ 2^(*deg div* 2) + *the*(*vebt-maxt* (*newlist* ! *h*))
else ma)))
deg newlist summary)
⟨*proof*⟩

```
lemma del-x-mia: assumes x = mi ∧ x < ma and mi ≠ ma  and deg ≥ 2
  shows vebt-delete (Node (Some (mi, ma)) deg treeList summary) x =(
             let xn = the (vebt-mint summary) ∗ 2^(deg div 2)
                                + the (vebt-mint (treeList ! the (vebt-mint summary)));
                                      minn = xn;
                 l = low xn (deg div 2);
                 h = high xn (deg div 2) in
                  if h < length treeList
                    then(
                       let newnode = vebt-delete (treeList ! h) l;
                          newlist = treeList[h:= newnode]in
                         if minNull newnode
                         then(
                            let sn = vebt-delete summary h in
                           (Node (Some (minn, if xn  = ma then (let maxs = vebt-maxt sn in
                                                              (if maxs = None
                                                                then minn
                                                                else 2^(deg div 2) ∗ the maxs
                                                                   + the (vebt-maxt (newlist ! the maxs))
                                                              )
                                                           )
                                                        else ma))
                                deg newlist sn)
                         )else
                           (Node (Some (minn, (if xn = ma then
                                             h ∗ 2^(deg div 2) + the( vebt-maxt (newlist ! h))
                                                    else ma)))
                            deg newlist summary )
                    )else
                     (Node (Some (mi, ma)) deg treeList summary)
          )
  ⟨proof⟩
```

```
lemma del-x-mi:
  assumes x = mi ∧ x < ma and mi ≠ ma  and deg ≥ 2  and high xn (deg div 2) = h and
    xn =  the (vebt-mint summary) ∗ 2^(deg div 2) + the (vebt-mint (treeList ! the (vebt-mint sum-
mary)))
    low xn (deg div 2) = land  high xn (deg div 2) < length treeList
  shows
  vebt-delete (Node (Some (mi, ma)) deg treeList summary) x =(
                       let newnode = vebt-delete (treeList ! h) l;
                          newlist = treeList[h:= newnode]in
                         if minNull newnode
                         then(
                            let sn = vebt-delete summary h in
                           (Node (Some (xn, if xn  = ma then (let maxs = vebt-maxt sn in
                                                              (if maxs = None
                                                                then xn
```

else $2\widehat{\ }(deg\ div\ 2) * the\ maxs$
$+ the\ (vebt\text{-}maxt\ (newlist\ !\ the\ maxs))$
)
)
else ma))
deg newlist sn)
)*else*
(*Node* (*Some* (*xn*, (*if xn* = *ma then*
$h * 2\widehat{\ }(deg\ div\ 2) + the(\ vebt\text{-}maxt\ (newlist\ !\ h))$
else ma)))
deg newlist summary))

⟨*proof*⟩

lemma *del-x-mi-lets-in*:
assumes $x = mi \wedge x < ma$ **and** $mi \neq ma$ **and** $deg \geq 2$ **and** *high xn* (*deg div 2*) = *h* **and**
xn = *the* (*vebt-mint summary*) $* 2\widehat{\ }(deg\ div\ 2)$ + *the* (*vebt-mint* (*treeList* ! *the* (*vebt-mint summary*)))
low xn (*deg div 2*) = *l***and** *high xn* (*deg div 2*) < *length treeList* **and**
newnode = *vebt-delete* (*treeList* ! *h*) *l* **and** *newlist* = *treeList*[*h*:= *newnode*]
shows *vebt-delete* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* =(*if minNull newnode*
then(
let sn = *vebt-delete summary h in*
(*Node* (*Some* (*xn*, *if xn* = *ma then* (*let maxs* = *vebt-maxt sn in*
(*if maxs* = *None*
then xn
else $2\widehat{\ }(deg\ div\ 2) * the\ maxs$
$+ the\ (vebt\text{-}maxt\ (newlist\ !\ the\ maxs))$
)
)
else ma))
deg newlist sn)
)*else*
(*Node* (*Some* (*xn*, (*if xn* = *ma then*
$h * 2\widehat{\ }(deg\ div\ 2) + the(\ vebt\text{-}maxt\ (newlist\ !\ h))$
else ma)))
deg newlist summary))
⟨*proof*⟩

lemma *del-x-mi-lets-in-minNull*:
assumes $x = mi \wedge x < ma$ **and** $mi \neq ma$ **and** $deg \geq 2$ **and** *high xn* (*deg div 2*) = *h* **and**
xn = *the* (*vebt-mint summary*) $* 2\widehat{\ }(deg\ div\ 2)$ + *the* (*vebt-mint* (*treeList* ! *the* (*vebt-mint summary*)))
low xn (*deg div 2*) = *l***and** *high xn* (*deg div 2*) < *length treeList* **and**
newnode = *vebt-delete* (*treeList* ! *h*) *l* **and** *newlist* =*treeList*[*h*:= *newnode*] **and**
minNull newnode **and** *sn* = *vebt-delete summary h*
shows
vebt-delete (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* =
(*Node* (*Some* (*xn*, *if xn* = *ma then* (*let maxs* = *vebt-maxt sn in*

(*if* *maxs* = *None*
then *xn*
else 2^(*deg* *div* 2) ∗ *the* *maxs*
+ *the* (*vebt-maxt* (*newlist* ! *the* *maxs*))
)
)
else *ma*)) *deg* *newlist* *sn*)
⟨*proof*⟩

lemma *del-x-mi-lets-in-not-minNull*:
assumes *x* = *mi* ∧ *x* < *ma* **and** *mi* ≠ *ma* **and** *deg* ≥ 2 **and** *high* *xn* (*deg* *div* 2) = *h* **and**
xn = *the* (*vebt-mint* *summary*) ∗ 2^(*deg* *div* 2) + *the* (*vebt-mint* (*treeList* ! *the* (*vebt-mint* *summary*)))
low *xn* (*deg* *div* 2) = *l***and** *high* *xn* (*deg* *div* 2) < *length* *treeList* **and**
newnode = *vebt-delete* (*treeList* ! *h*) *l* **and** *newlist* = *treeList*[*h*:= *newnode*] **and**
¬*minNull* *newnode*
shows
vebt-delete (*Node* (*Some* (*mi*, *ma*)) *deg* *treeList* *summary*) *x* =
(*Node* (*Some* (*xn*, (*if* *xn* = *ma* *then*
h ∗ 2^(*deg* *div* 2) + *the*(*vebt-maxt* (*newlist* ! *h*))
else *ma*)))
deg *newlist* *summary*)
⟨*proof*⟩

theorem *dele-bmo-cont-corr*:*invar-vebt* *t* *n* ⟹ (*both-member-options* (*vebt-delete* *t* *x*) *y* ⟷ *x* ≠ *y* ∧ *both-member-options* *t* *y*)
⟨*proof*⟩

end

corollary *invar-vebt* *t* *n* ⟹ *both-member-options* *t* *x* ⟹ *x* ≠ *y* ⟹ *both-member-options* (*vebt-delete* *t* *y*) *x*
⟨*proof*⟩

corollary *invar-vebt* *t* *n* ⟹ *both-member-options* *t* *x* ⟹ ¬ *both-member-options* (*vebt-delete* *t* *x*) *x*
⟨*proof*⟩

corollary *invar-vebt* *t* *n* ⟹ *both-member-options* (*vebt-delete* *t* *y*) *x* ⟹ *both-member-options* *t* *x* ∧ *x* ≠ *y*
⟨*proof*⟩

end
end

theory *VEBT-DeleteCorrectness* **imports** *VEBT-Delete*
begin

context *VEBT-internal* **begin**

8.3 Validness Preservation

theorem *delete-pres-valid*: $invar\text{-}vebt\ t\ n \Longrightarrow invar\text{-}vebt\ (vebt\text{-}delete\ t\ x)\ n$
⟨*proof*⟩

corollary *dele-member-cont-corr*: $invar\text{-}vebt\ t\ n \Longrightarrow (vebt\text{-}member\ (vebt\text{-}delete\ t\ x)\ y \longleftrightarrow x \neq y \wedge vebt\text{-}member\ t\ y)$
⟨*proof*⟩

8.4 Correctness with Respect to Set Interpretation

theorem *delete-correct'*: **assumes** $invar\text{-}vebt\ t\ n$
shows $set\text{-}vebt'\ (vebt\text{-}delete\ t\ x) = set\text{-}vebt'\ t - \{x\}$
⟨*proof*⟩

corollary *delete-correct*: **assumes** $invar\text{-}vebt\ t\ n$
shows $set\text{-}vebt'\ (vebt\text{-}delete\ t\ x) = set\text{-}vebt\ t - \{x\}$
⟨*proof*⟩

end
end

theory *VEBT-Uniqueness* **imports** *VEBT-InsertCorrectness VEBT-Succ VEBT-Pred VEBT-DeleteCorrectness*
begin

context *VEBT-internal* **begin**

9 Uniqueness Property of valid Trees

Two valid van Emde Boas trees having equal degree number and representing the same integer set are equal.

theorem *uniquetree*: $invar\text{-}vebt\ t\ n \Longrightarrow invar\text{-}vebt\ s\ n \Longrightarrow set\text{-}vebt'\ t = set\text{-}vebt'\ s \Longrightarrow s = t$
⟨*proof*⟩

corollary $invar\text{-}vebt\ t\ n \Longrightarrow set\text{-}vebt'\ t = \{\} \Longrightarrow t = vebt\text{-}buildup\ n$
⟨*proof*⟩

corollary *unique-tree*: $invar\text{-}vebt\ t\ n \Longrightarrow invar\text{-}vebt\ s\ n \Longrightarrow set\text{-}vebt\ t = set\text{-}vebt\ s \Longrightarrow s = t$
⟨*proof*⟩

corollary $invar\text{-}vebt\ t\ n \Longrightarrow set\text{-}vebt\ t = \{\} \Longrightarrow t = vebt\text{-}buildup\ n$
⟨*proof*⟩

All valid trees can be generated by $vebt - insertion$ chains on an empty tree with same degree parameter:

inductive *perInsTrans*:: $VEBT \Rightarrow VEBT \Rightarrow bool$ **where**
$perInsTrans\ t\ t|$
$(t = vebt\text{-}insert\ s\ x) \Longrightarrow perInsTrans\ t\ u \Longrightarrow perInsTrans\ s\ u$

lemma *perIT-concat*: *perInsTrans s t* $\Longrightarrow$ *perInsTrans t u* $\Longrightarrow$ *perInsTrans s u*
⟨*proof*⟩

lemma assumes *invar-vebt t n* **shows**
perInsTrans (*vebt-buildup n*) *t*
⟨*proof*⟩

end
end

theory *VEBT-Height* **imports** *VEBT-Definitions Complex-Main*
begin

context *VEBT-internal* **begin**

10 Heights of van Emde Boas Trees

fun *height*:: *VEBT* $\Rightarrow$ *nat* **where**
height (*Leaf a b*) = *0*|
height (*Node - deg treeList summary*) = (*1*+ *Max* (*height* ‘ (*insert summary* (*set treeList*))))

abbreviation *lb x* $\equiv$ *log 2 x*

lemma *setceilmax*: *invar-vebt s m* $\Longrightarrow \forall$ *t* $\in$ *set listy*. *invar-vebt t n*
$\Longrightarrow$*m* = *Suc n* $\Longrightarrow$($\forall$ *t* $\in$ *set listy*. *height t* = $\lceil$*lb n* $\rceil$) $\Longrightarrow$ *height s* = $\lceil$*lb m*$\rceil$
$\Longrightarrow$ *Max* (*height* ‘ (*insert s* (*set listy*))) = $\lceil$*lb m*$\rceil$
⟨*proof*⟩

lemma *log-ceil-idem*:
assumes(*x*::*real*) $\geq$ *1*
shows $\lceil$*lb x* $\rceil$ = $\lceil$*lb* $\lceil$*x*$\rceil\rceil$
⟨*proof*⟩

lemma *heigt-uplog-rel*:*invar-vebt t n* $\Longrightarrow$ (*height t*) = $\lceil$*lb n*$\rceil$
⟨*proof*⟩

lemma *two-powr-height-bound-deg*:
assumes *invar-vebt t n*
shows *2*^(*height t*) $\leq$ *2*∗(*n*::*nat*)
⟨*proof*⟩

Main Theorem

theorem *height-double-log-univ-size*:
assumes *u* = *2*^*deg* **and** *invar-vebt t deg*
shows *height t* $\leq$ *1* + *lb* (*lb u*)
⟨*proof*⟩

lemma *height-compose-list*: *t*$\in$ *set treeList* $\Longrightarrow$

Max (*height* ‘ (*insert summary* (*set treeList*))) ≥ *height t*
⟨*proof*⟩

lemma *height-compose-child*: *t*∈ *set treeList* ⟹
height (*Node info deg treeList summary*) ≥ *1+* *height t* ⟨*proof*⟩

lemma *height-compose-summary*: *height* (*Node info deg treeList summary*) ≥ *1+* *height summary*
⟨*proof*⟩

lemma *height-i-max*: *i* < *length x13* ⟹
height (*x13* ! *i*) ≤ *max foo* (*Max* (*height* ‘ *set x13*))
⟨*proof*⟩

lemma *max-ins-scaled*: *n*∗ *height x14* ≤ *m* + *n*∗ *Max* (*insert* (*height x14*) (*height* ‘ *set x13*))
⟨*proof*⟩

lemma *max-idx-list*:
assumes *i* < *length x13*
shows *n* ∗ *height* (*x13* !*i*) ≤ *Suc* (*Suc* (*n* ∗ *max* (*height x14*) (*Max* (*height* ‘ *set x13*))))
⟨*proof*⟩

end
end

theory *VEBT-Bounds* **imports** *VEBT-Height VEBT-Member VEBT-Insert VEBT-Succ VEBT-Pred*
begin

11 Upper Bounds for canonical Functions: Relationships between Run Time and Tree Heights

11.1 Membership test

context begin

interpretation *VEBT-internal* ⟨*proof*⟩

fun T_{member}:: *VEBT* ⇒ *nat* ⇒ *nat* **where**
T_{member} (*Leaf a b*) *x* = *2* + (*if x* = *0 then 1 else 1* +(*if x*=*1 then 1 else 1*))|
T_{member} (*Node None - - -*) *x* = *2*|
T_{member} (*Node - 0 - -*) *x* = *2*|
T_{member} (*Node -* (*Suc 0*) *- -*) *x* = *2*|
T_{member} (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = *2* + (
if x = *mi then 1 else 1*+ (
if x = *ma then 1 else 1*+(
if x < *mi then 1 else 1*+ (
if x > *ma then 1 else 9* +
(*let*
h = *high x* (*deg div 2*);
l = *low x* (*deg div 2*) *in*

(*if* $h <$ *length treeList*
  then $1 + T_{member}$ (*treeList* ! *h*) *l*
  else 1))))))

fun T_{member}':: *VEBT* ⇒ *nat* ⇒ *nat* **where**
 T_{member}' (*Leaf a b*) $x = 1$|
 T_{member}' (*Node None - - -*) $x = 1$|
 T_{member}' (*Node - 0 - -*) $x = 1$|
 T_{member}' (*Node -* (*Suc 0*) *- -*) $x = 1$|
 T_{member}' (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) $x = 1+($
if $x = mi$ *then* 0 *else* (
if $x = ma$ *then* 0 *else* (
if $x < mi$ *then* 0 *else* (
if $x > ma$ *then* 0 *else if* ($x>mi \wedge x < ma$) *then*
(*let*
  h = *high x* (*deg div 2*);
  l = *low x* (*deg div 2*) *in*
  (*if* $h <$ *length treeList*
  then T_{member}' (*treeList* ! *h*) *l*
  else 0))
else 0))))

lemma *height-node*: *invar-vebt* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *n*
  ⟹ *height* (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) >= 1
⟨*proof*⟩

theorem *member-bound-height*: *invar-vebt t n* ⟹ T_{member} *t x* ≤ (1+*height t*)∗15
⟨*proof*⟩

theorem *member-bound-height′*: *invar-vebt t n* ⟹ T_{member}' *t x* ≤ (1+*height t*)
⟨*proof*⟩

theorem *member-bound-size-univ*: *invar-vebt t n* ⟹ $u = 2\hat{}n$ ⟹ T_{member} *t x* ≤ 30 + 15 ∗ *lb* (*lb u*)
⟨*proof*⟩

11.2 Minimum, Maximum, Emptiness Test

fun T_{mint}:: *VEBT* ⇒ *nat* **where**
 T_{mint} (*Leaf a b*) = (1+ (*if a then* 0 *else* 1 + (*if b then* 1 *else* 1)))|
 T_{mint} (*Node None - - -*) = 1|
 T_{mint} (*Node* (*Some* (*mi*,*ma*)) *- - -*) = 1

lemma *mint-bound*: T_{mint} *t* ≤ 3 ⟨*proof*⟩

fun T_{maxt}:: *VEBT* ⇒ *nat* **where**

T_{maxt} (*Leaf a b*) = (*1*+ (*if b then 1 else 1* +(*if a then 1 else 1*)))|
T_{maxt} (*Node None - - -*) = *1*|
T_{maxt} (*Node* (*Some* (*mi*,*ma*)) *- - -*) = *1*

lemma *maxt-bound*: T_{maxt} *t* ≤ *3* ⟨*proof*⟩

fun $T_{minNull}$::*VEBT* ⇒ *nat* **where**
$T_{minNull}$ (*Leaf False False*) = *1*|
$T_{minNull}$ (*Leaf - -*) = *1*|
$T_{minNull}$ (*Node None - - -*) = *1*|
$T_{minNull}$ (*Node* (*Some -*) *- - -*) = *1*

lemma *minNull-bound*: $T_{minNull}$ *t* ≤ *1*
⟨*proof*⟩

11.3 Insertion

fun T_{insert}::*VEBT* ⇒ *nat* ⇒*nat* **where**
T_{insert} (*Leaf a b*) *x* = *1*+ (*if x*=*0 then 1 else 1* + (*if x*=*1 then 1 else 1*))|
T_{insert} (*Node info 0 ts s*) *x* = *1*|
T_{insert} (*Node info* (*Suc 0*) *ts s*) *x* = *1*|
T_{insert} (*Node None* (*Suc deg*) *treeList summary*) *x* = *2*|
T_{insert} (*Node* (*Some* (*mi*,*ma*)) *deg treeList summary*) *x* = *19*+
(*let xn* = (*if x* < *mi then mi else x*); *minn* = (*if x*< *mi then x else mi*);
l= *low xn* (*deg div 2*); *h* = *high xn* (*deg div 2*)
in
(*if h* < *length treeList* ∧ ¬ (*x* = *mi* ∨ *x* = *ma*)*then*
T_{insert} (*treeList* ! *h*) *l* + $T_{minNull}$ (*treeList* ! *h*)+
(*if minNull* (*treeList* ! *h*) *then* T_{insert} *summary h else 1*)
else 1))

fun $T_{insert}{}'$::*VEBT* ⇒ *nat* ⇒*nat* **where**
$T_{insert}{}'$ (*Leaf a b*) *x* = *1*|
$T_{insert}{}'$ (*Node info 0 ts s*) *x* = *1*|
$T_{insert}{}'$ (*Node info* (*Suc 0*) *ts s*) *x* = *1*|
$T_{insert}{}'$ (*Node None* (*Suc deg*) *treeList summary*) *x* = *1*|
$T_{insert}{}'$ (*Node* (*Some* (*mi*,*ma*)) *deg treeList summary*) *x* =
(*let xn* = (*if x* < *mi then mi else x*); *minn* = (*if x*< *mi then x else mi*);
l= *low xn* (*deg div 2*); *h* = *high xn* (*deg div 2*)
in (*if h* < *length treeList* ∧ ¬ (*x* = *mi* ∨ *x* = *ma*)*then*
$T_{insert}{}'$ (*treeList* ! *h*) *l* +
(*if minNull* (*treeList* ! *h*) *then* $T_{insert}{}'$ *summary h else 1*) *else 1*))

lemma *insersimp*:**assumes** *invar-vebt t n* **and** ∄ *x*. *both-member-options t x* **shows** T_{insert} *t y* ≤ *3*
⟨*proof*⟩

lemma *insertsimp*: *invar-vebt t n* $\Longrightarrow$ *minNull t* $\Longrightarrow$ T_{insert} *t l* $\leq$ *3*
⟨*proof*⟩

lemma *insersimp'*:**assumes** *invar-vebt t n* **and** $\nexists$ *x. both-member-options t x* **shows** T_{insert}' *t y* $\leq$ *1*
⟨*proof*⟩

lemma *insertsimp'*: *invar-vebt t n* $\Longrightarrow$ *minNull t* $\Longrightarrow$ T_{insert}' *t l* $\leq$ *1*
⟨*proof*⟩

theorem *insert-bound-height*: *invar-vebt t n* $\Longrightarrow$ T_{insert} *t x* $\leq$ (*1*+*height t*)∗*23*
⟨*proof*⟩

theorem *insert-bound-size-univ*: *invar-vebt t n* $\Longrightarrow$ *u* = *2*^*n* $\Longrightarrow$ T_{insert} *t x* $\leq$ *46* + *23* ∗ *lb* (*lb u*)
⟨*proof*⟩

theorem *insert'-bound-height*: *invar-vebt t n* $\Longrightarrow$ T_{insert}' *t x* $\leq$ (*1*+*height t*)
⟨*proof*⟩

11.4 Successor Function

fun T_{succ}::*VEBT* ⇒ *nat* ⇒ *nat* **where**
  T_{succ} (*Leaf - b*) *0* = *1*+ (*if b then 1 else 1*)|
  T_{succ} (*Leaf - -*) (*Suc n*) = *1*|
  T_{succ} (*Node None - - -*) *-* = *1*|
  T_{succ} (*Node - 0 - -*) *-* = *1*|
  T_{succ} (*Node -* (*Suc 0*) *- -*) *-* = *1*|
  T_{succ} (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = *1*+ (
    if x < *mi then 1*
    else (*let l* = *low x* (*deg div 2*); *h* = *high x* (*deg div 2*) *in 10* +
      (*if h* < *length treeList then 1*+ T_{maxt} (*treeList* ! *h*) + (
        let maxlow = *vebt-maxt* (*treeList* ! *h*) *in 3* +
        (*if maxlow* ≠ *None* ∧ (*Some l* $<_o$ *maxlow*) *then*
          4 + T_{succ} (*treeList* ! *h*) *l*
        else let sc = *vebt-succ summary h in 1*+ T_{succ} *summary h* + *1* + (
        if sc = *None then 1*
        else (*4* + T_{mint} (*treeList* ! *the sc*)))))

      else 1)))

fun T_{succ}'::*VEBT* ⇒ *nat* ⇒ *nat* **where**
  T_{succ}' (*Leaf - b*) *0* = *1*|
  T_{succ}' (*Leaf - -*) (*Suc n*) = *1*|
  T_{succ}' (*Node None - - -*) *-* = *1*|
  T_{succ}' (*Node - 0 - -*) *-* = *1*|
  T_{succ}' (*Node -* (*Suc 0*) *- -*) *-* = *1*|
  T_{succ}' (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* =(
    if x < *mi then 1*

else (*let* l = *low* x (*deg div 2*); h = *high* x (*deg div 2*) *in*
(*if* $h <$ *length treeList then* (
let maxlow = *vebt-maxt* (*treeList* ! h) *in*
(*if maxlow* $\neq$ *None* $\wedge$ (*Some* $l <_o$ *maxlow*) *then*
1+ $T_{succ}{}'$ (*treeList* ! h) l
else let sc = *vebt-succ summary* h *in* $T_{succ}{}'$ *summary* h + (
if sc = *None then 1*
else 1)))
else 1)))

theorem *succ-bound-height*: *invar-vebt* t $n \Longrightarrow T_{succ}$ t $x \leq$ (*1*+*height* t)$*$*27*
⟨*proof*⟩

theorem *succ-bound-size-univ*: *invar-vebt* t $n \Longrightarrow u =$ *2*^$n \Longrightarrow$ T_{succ} t $x \leq$ *54* + *27* $*$ *lb* (*lb* u)
⟨*proof*⟩

theorem *succ′-bound-height*: *invar-vebt* t $n \Longrightarrow T_{succ}{}'$ t $x \leq$ (*1*+*height* t)
⟨*proof*⟩

theorem *succ-bound-size-univ′*: *invar-vebt* t $n \Longrightarrow u =$ *2*^$n \Longrightarrow$ $T_{succ}{}'$ t $x \leq$ *2* + *lb* (*lb* u)
⟨*proof*⟩

11.5 Predecessor Function

fun T_{pred}:: *VEBT* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
T_{pred} (*Leaf* - -) *0* = *1*|
T_{pred} (*Leaf* a -) (*Suc 0*) = *1*+ (*if* a *then 1 else 1*)|
T_{pred} (*Leaf* a b) - = *1*+ (*if* b *then 1 else 1*+ (*if* a *then 1 else 1*))|

T_{pred} (*Node None* - - -) - = *1*|
T_{pred} (*Node* - *0* - -) - = *1*|
T_{pred} (*Node* - (*Suc 0*) - -) - = *1*|
T_{pred} (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) x = *1*+ (
if $x >$ *ma then 1*
else (*let* l = *low* x (*deg div 2*); h = *high* x (*deg div 2*) *in 10* + *1*+
(*if* $h <$ *length treeList then*

let minlow = *vebt-mint* (*treeList* ! h) *in 2* + T_{mint}(*treeList* ! h) + *3* +
(*if minlow* $\neq$ *None* $\wedge$ (*Some* $l >_o$ *minlow*) *then*
4 + T_{pred} (*treeList* ! h) l
else let pr = *vebt-pred summary* h *in* *1* + T_{pred} *summary* h+ *1* + (
if pr = *None then 1* + (*if* $x >$ *mi then 1 else 1*)
else 4 + T_{maxt} (*treeList* ! *the pr*)))
else 1)))

theorem *pred-bound-height*: *invar-vebt* t $n \Longrightarrow T_{pred}$ t $x \leq$ (*1*+*height* t)$*$*29*
⟨*proof*⟩

theorem *pred-bound-size-univ*: *invar-vebt t n* $\Longrightarrow$ *u* = *2^n* $\Longrightarrow$ T_{pred} *t x* $\leq$ *58* + *29* $*$ *lb* (*lb u*)
⟨*proof*⟩

fun T_{pred}'::*VEBT* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
T_{pred}' (*Leaf* - -) *0* = *1*|
T_{pred}' (*Leaf a* -) (*Suc 0*) = *1*|
T_{pred}' (*Leaf a b*) - = *1*|

T_{pred}' (*Node None* - - -) - = *1*|
T_{pred}' (*Node* - *0* - -) - = *1*|
T_{pred}' (*Node* - (*Suc 0*) - -) - = *1*|
T_{pred}' (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (
if x > *ma then 1*
else (*let l* = *low x* (*deg div 2*); *h* = *high x* (*deg div 2*) *in*
(*if h* < *length treeList then*

let minlow = *vebt-mint* (*treeList* ! *h*) *in*
(*if minlow* $\neq$ *None* $\wedge$ (*Some l* $>_o$ *minlow*) *then*
1+ T_{pred}' (*treeList* ! *h*) *l*
else let pr = *vebt-pred summary h in* T_{pred}' *summary h*+ (
if pr = *None then 1*
else 1))
else 1)))

theorem *pred-bound-height'*: *invar-vebt t n*$\Longrightarrow$ T_{pred}' *t x* $\leq$ (*1* + *height t*)
⟨*proof*⟩

theorem *pred-bound-size-univ'*: *invar-vebt t n* $\Longrightarrow$ *u* = *2^n* $\Longrightarrow$ T_{pred}' *t x* $\leq$ *2* + *lb* (*lb u*)
⟨*proof*⟩

end
end

theory *VEBT-DeleteBounds* **imports** *VEBT-Bounds VEBT-Delete VEBT-DeleteCorrectness*
begin

11.6 Running Time Bounds for Deletion

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

fun T_{delete}::*VEBT* $\Rightarrow$ *nat* $\Rightarrow$ *nat* **where**
T_{delete} (*Leaf a b*) *0* = *1*|
T_{delete} (*Leaf a b*) (*Suc 0*) = *1*|
T_{delete} (*Leaf a b*) (*Suc* (*Suc n*)) = *1*|
T_{delete} (*Node None deg treeList summary*) - = *1*|
T_{delete} (*Node* (*Some* (*mi*, *ma*)) *0 treeList summary*) *x* = *1*|
T_{delete} (*Node* (*Some* (*mi*, *ma*)) (*Suc 0*) *treeList summary*) *x* =*1* |
T_{delete} (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = *3* + (

$if\ (x < mi \vee x > ma)\ then\ 1$
$else\ 3 + (if\ (x = mi \wedge x = ma)\ then\ 3$
$else\ 13 + (\ if\ x = mi\ then\ T_{mint}\ summary + T_{mint}\ (treeList\ !\ the\ (vebt\text{-}mint\ summary)) + 7\ else\ 1\ ) +$
$(if\ x = mi\ then\ 1\ else\ 1)\ +$
$(\ \ let\ xn = (if\ x = mi$
$then\ the\ (vebt\text{-}mint\ summary) * 2\hat{}(deg\ div\ 2) + the\ (vebt\text{-}mint\ (treeList\ !\ the\ (vebt\text{-}mint\ summary)))$
$else\ x);$
$minn = (if\ x = mi\ then\ xn\ else\ mi);$
$l = low\ xn\ (deg\ div\ 2);$
$h = high\ xn\ (deg\ div\ 2)\ in$
$if\ h < length\ treeList$
$then(\ 4 + T_{delete}\ (treeList\ !\ h)\ l\ +($
$let\ newnode = vebt\text{-}delete\ (treeList\ !\ h)\ l;$
$newlist = treeList[h{:=}\ newnode] in\ 1 + T_{minNull}\ newnode + ($
$if\ minNull\ newnode$
$then(\ \ 1 +\ T_{delete}\ summary\ h + ($
$let\ sn = vebt\text{-}delete\ summary\ h\ in$
$2 + (if\ xn\ = ma\ then\ 1 + T_{maxt}\ sn + (let\ maxs = vebt\text{-}maxt\ sn\ in$
$1 + (if\ maxs = None$
$then\ 1$
$else\ 8 +\ T_{maxt}\ (newlist\ !\ the\ maxs)$
$)\ )$
$else\ 1)$
$))else$
$2 + (if\ xn = ma\ then\ 6 +\ T_{maxt}\ (newlist\ !\ h)\ \ else\ \ 1)$
$)))else\ \ 1\ \ )))$

end

context *VEBT-internal* **begin**

lemma *tdeletemimi*: $deg \geq 2 \Longrightarrow T_{delete}\ (Node\ (Some\ (mi,\ mi))\ deg\ treeList\ summary)\ x \leq 9$
⟨*proof*⟩

lemma *minNull-delete-time-bound*: $invar\text{-}vebt\ t\ n \Longrightarrow minNull\ (vebt\text{-}delete\ t\ x) \Longrightarrow\ T_{delete}\ t\ x \leq 9$
⟨*proof*⟩

lemma *delete-bound-height*: $invar\text{-}vebt\ t\ n \Longrightarrow\ T_{delete}\ t\ x \leq (1+\ height\ t)*70$
⟨*proof*⟩

theorem *delete-bound-size-univ*: $invar\text{-}vebt\ t\ n \Longrightarrow u =\ 2\hat{}n \Longrightarrow\ \ T_{delete}\ \ t\ x \leq 140 + 70 * lb\ (lb\ u)$
⟨*proof*⟩

fun $T_{delete}'{::}\ VEBT \Rightarrow nat \Rightarrow nat$ **where**
$T_{delete}'\ (Leaf\ a\ b)\ 0 = 1|$
$T_{delete}'\ (Leaf\ a\ b)\ (Suc\ 0) = 1|$

$T_{delete}{}'$ (*Leaf a b*) (*Suc* (*Suc n*)) = *1*|
$T_{delete}{}'$ (*Node None deg treeList summary*) - = *1*|
$T_{delete}{}'$ (*Node* (*Some* (*mi*, *ma*)) *0 treeList summary*) *x* = *1*|
$T_{delete}{}'$ (*Node* (*Some* (*mi*, *ma*)) (*Suc 0*) *treeList summary*) *x* =*1* |
$T_{delete}{}'$ (*Node* (*Some* (*mi*, *ma*)) *deg treeList summary*) *x* = (
if (*x* < *mi* ∨ *x* > *ma*) *then 1*
else if (*x* = *mi* ∧ *x* = *ma*) *then 1*
else (*let xn* = (*if x* = *mi*
then the (*vebt-mint summary*) ∗ *2*^(*deg div 2*) + *the* (*vebt-mint* (*treeList* ! *the* (*vebt-mint summary*)))
else x);
minn = (*if x* = *mi then xn else mi*);
l = *low xn* (*deg div 2*);
h = *high xn* (*deg div 2*) *in*
if h < *length treeList*
then($T_{delete}{}'$ (*treeList* ! *h*) *l* +(
let newnode = *vebt-delete* (*treeList* ! *h*) *l*;
newlist = *treeList*[*h*:= *newnode*]*in*
if minNull newnode
then $T_{delete}{}'$ *summary h*
else 1
))*else 1*))

lemma *tdeletemimi′*:*deg* ≥ *2* ⟹ $T_{delete}{}'$ (*Node* (*Some* (*mi*, *mi*)) *deg treeList summary*) *x* ≤ *1*
⟨*proof*⟩

lemma *minNull-delete-time-bound′*: *invar-vebt t n* ⟹ *minNull* (*vebt-delete t x*) ⟹ $T_{delete}{}'$ *t x* ≤ *1*
⟨*proof*⟩

lemma *delete-bound-height′*: *invar-vebt t n* ⟹ $T_{delete}{}'$ *t x* ≤ *1*+ *height t*
⟨*proof*⟩

theorem *delete-bound-size-univ′*: *invar-vebt t n* ⟹ *u* = *2*^*n* ⟹ $T_{delete}{}'$ *t x* ≤ *2* + *lb* (*lb u*)
⟨*proof*⟩

end
end

theory *VEBT-Space* **imports** *VEBT-Definitions Complex-Main*
begin

12 Space Complexity and *buildup* Time Consumption

12.1 Space Comlexity of valid van Emde Boas Trees

Space Complexity is linear in relation to universe sizes

context *VEBT-internal* **begin**

fun *space*:: *VEBT* ⇒ *nat* **where**
space (*Leaf a b*) = *3*|
space (*Node info deg treeList summary*) = 5 + *space summary* + *length treeList* + *foldr* (λ *a b. a*+*b*) (*map space treeList*) *0*

fun *space′*:: *VEBT* ⇒ *nat* **where**
space′ (*Leaf a b*) = *4*|
space′ (*Node info deg treeList summary*) = *6* + *space′ summary* + *foldr* (λ *a b. a*+*b*) (*map space′ treeList*) *0*

Count in reals

fun *cnt*:: *VEBT* ⇒ *real* **where**
cnt (*Leaf a b*) = *1*|
cnt (*Node info deg treeList summary*) = *1* + *cnt summary* + *foldr* (λ *a b. a*+*b*) (*map cnt treeList*) *0*

12.2 Auxiliary Lemmas for List Summation

lemma *list-every-elemnt-bound-sum-bound*:∀ *x* ∈ *set xs. f x* ≤ *bound* ⟹ *foldr* (λ *a b. a*+*b*) (*map f xs*) *i* ≤ *length xs* ∗ *bound* + *i*
⟨*proof*⟩

lemma *list-every-elemnt-bound-sum-bound-real*:∀ *x* ∈ *set* (*xs*::′*a list*). (*f*::′*a*⇒*real*) *x* ≤ (*bound*::*real*) ⟹ *foldr* (λ *a b. a*+*b*) (*map f xs*) *i* ≤ *real*(*length xs*) ∗ *bound* + *i*
⟨*proof*⟩

lemma *foldr-one*: *d* ≤ *foldr* (+) *ys* (*d*::*nat*)
⟨*proof*⟩

lemma *foldr-zero*: ∀ *i* < *length xs. xs* ! *i* > *0* ⟹
foldr (λ *a b. a*+*b*) *xs* (*d*::*nat*) − *d* ≥ *length xs*
⟨*proof*⟩

lemma *foldr-mono*: *length xs* = *length ys* ⟹∀ *i* < *length xs. xs* ! *i* < *ys* ! *i* ⟹ *c* ≤ *d* ⟹
foldr (λ *a b. a*+*b*) *xs c* + *length ys* ≤ *foldr* (λ *a b. a*+*b*) *ys* (*d*::*nat*)
⟨*proof*⟩

lemma *two-realpow-ge-two* :(*n*::*real*)≥ *1* ⟹ (*2*::*real*)ˆ*n* ≥ *2*
⟨*proof*⟩

lemma *foldr0*: *foldr* (+) *xs* (*c*+*d*) = *foldr* (+) *xs* (*d*::*real*) + *c*
⟨*proof*⟩

lemma *f-g-map-foldr-bound*: (∀ *x* ∈ *set xs. f x* ≤ *c* ∗ *g x*)
⟹ *foldr* (λ *a b. a*+*b*) (*map f xs*) *d* ≤ *c* ∗ *foldr* (λ *a b. a*+*b*) (*map g xs*) (*0*::*real*) + *d*
⟨*proof*⟩

lemma *real-nat-list*: *real* (*foldr* (+) (*map f xs*) (*c*::*nat*))
= *foldr* (+) (*map* (λ *x. real*(*f x*))*xs*) *c*

⟨*proof*⟩

12.3 Actual Space Reasoning

lemma *space-space'*: $space'\ t > space\ t$
⟨*proof*⟩

lemma *cnt-bound*:
defines $c \equiv 1.5$
shows $invar\text{-}vebt\ t\ n \Longrightarrow cnt\ t \le 2*((2\hat{\ }n - c)::real)$
⟨*proof*⟩

theorem *cnt-bound'*: $invar\text{-}vebt\ t\ n \Longrightarrow cnt\ t \le 2 * (2\ \hat{}\ n - 1)$
⟨*proof*⟩

lemma *space-cnt*: $space'\ t \le 6*cnt\ t$
⟨*proof*⟩

lemma *space-2-pow-bound*: **assumes** $invar\text{-}vebt\ t\ n$ **shows** $real\ (space'\ t) \le 12 * (2\hat{\ }n - 1)$
⟨*proof*⟩

lemma *space'-bound*:
assumes $invar\text{-}vebt\ t\ n\ u = 2\hat{\ }n$
shows $space'\ t \le 12 * u$
⟨*proof*⟩

Main Theorem

theorem *space-bound*:
assumes $invar\text{-}vebt\ t\ n\ u = 2\hat{\ }n$
shows $space\ t \le 12 * u$
⟨*proof*⟩

12.4 Complexity of Generation Time

Space complexity is closely related to tree generation time complexity

Time approximation for replicate function. $T_{replicate}\ n\ t\ x$ denotes runnig time of the n-times replication of x into a list. t models runtime for generation of a single x.

fun $T_{buildup}::nat \Rightarrow nat$ **where**
$T_{buildup}\ 0 = 3|$
$T_{buildup}\ (Suc\ 0) = 3|$
$T_{buildup}\ n = (if\ even\ n\ then\ 1 + (let\ half = n\ div\ 2\ in$
$9 + T_{buildup}\ half + (2\hat{\ }half) * (T_{buildup}\ half + 1))$
$else\ (let\ half = n\ div\ 2\ in$
$11 + T_{buildup}\ (Suc\ half) + (2\hat{\ }(Suc\ half)) * (T_{buildup}\ half + 1)))$

fun $T_{build}::nat \Rightarrow nat$ **where**
$T_{build}\ 0 = 4|$
$T_{build}\ (Suc\ 0) = 4|$

T_{build} n = (*if even n then 1* + (*let half* = *n div 2 in*
10 + T_{build} *half* + (2^*half*) ∗ (T_{build} *half*))
else (*let half* = *n div 2 in*
12 + T_{build} (*Suc half*) + (2^(*Suc half*))∗ (T_{build} *half*)))

lemma *buildup-build-time*: $T_{buildup}$ n < T_{build} n
⟨*proof*⟩

lemma *listsum-bound*: (⋀ *x*. *x* ∈ *set xs* ⟹ *f x* ≥ (*0*::*real*)) ⟹
foldr (+) (*map f xs*) *y* ≥ *y*
⟨*proof*⟩

lemma *cnt-non-neg*: *cnt t* ≥ *0*
⟨*proof*⟩

lemma *foldr-same*: (⋀ *x y*. *x* ∈ *set* (*xs*::*real list*) ⟹ *y* ∈ *set xs* ⟹ *x* = *y*) ⟹
(⋀ *x* . (*x*::*real*) ∈ *set xs* ⟹ *x* = (*y*::*real*)) ⟹
foldr (λ (*a*::*real*) (*b*::*real*). *a*+*b*) *xs 0* = *real* (*length xs*) ∗ *y*
⟨*proof*⟩

lemma *foldr-same-int*: (⋀ *x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *x* = *y*) ⟹
(⋀ *x* . *x* ∈ *set xs* ⟹ *x* = *y*) ⟹
foldr (+) *xs 0* = (*length xs*) ∗ *y*
⟨*proof*⟩

lemma *t-build-cnt*: T_{build} *n* ≤ *cnt* (*vebt-buildup n*) ∗ *13*
⟨*proof*⟩

lemma *t-buildup-cnt*: $T_{buildup}$ *n* ≤ *cnt* (*vebt-buildup n*) ∗ *13*
⟨*proof*⟩

lemma *count-buildup*: *cnt* (*vebt-buildup n*) ≤ *2* ∗ *2*^*n*
⟨*proof*⟩

lemma *count-buildup′*: *cnt* (*vebt-buildup n*) ≤ *2* ∗ (*2*::*nat*)^*n*
⟨*proof*⟩

theorem *vebt-buildup-bound*: *u* = *2*^*n* ⟹ $T_{buildup}$ *n* ≤ *26* ∗ *u*
⟨*proof*⟩

Count in natural numbers

fun *cnt′*:: *VEBT* ⇒ *nat* **where**
cnt′ (*Leaf a b*) = *1*|
cnt′ (*Node info deg treeList summary*) = *1* + *cnt′ summary* + *foldr* (λ *a b*. *a*+*b*) (*map cnt′ treeList*) *0*

lemma *cnt-cnt-eq*:*cnt t* = *cnt′ t*
⟨*proof*⟩

end
end

13 Functional Interface

theory *VEBT-Intf-Functional*
imports *Main*
VEBT-Definitions VEBT-Space
VEBT-Uniqueness
VEBT-Member
VEBT-Insert VEBT-InsertCorrectness
VEBT-MinMax
VEBT-Pred VEBT-Succ
VEBT-Bounds
VEBT-Delete VEBT-DeleteCorrectness VEBT-DeleteBounds

begin

13.1 Code Generation Setup

13.1.1 Code Equations

Code generator seems to not support patterns and nat code target

```
context begin
  interpretation VEBT-internal ⟨proof⟩

lemma vebt-member-code[code]:
  vebt-member (Leaf a b) x = (if x = 0 then a else if x=1 then b else False)
  vebt-member (Node None t r e) x = False
  vebt-member (Node (Some (mi, ma)) deg treeList summary) x =
  (if deg = 0 ∨ deg = Suc 0 then False else (
    if x = mi then True else
    if x = ma then True else
    if x < mi then False else
   if x > ma then False else
   (let
        h = high x (deg div 2);
       l = low x (deg div 2) in
     (if h < length treeList
         then vebt-member (treeList ! h) l
         else False))))
     ⟨proof⟩

lemma vebt-insert-code[code]:
  vebt-insert (Leaf a b) x = (if x=0 then Leaf True b else if x=1 then Leaf a True else Leaf a b)
  vebt-insert (Node info deg treeList summary) x = (
    if deg ≤ 1 then
      (Node info deg treeList summary)
```

```
  else ( case info of
    None ⇒  (Node (Some (x,x)) deg  treeList summary)
  | Some mima ⇒ ( case mima of (mi, ma) ⇒ (
      let
        xn = (if x < mi then mi else x);
        minn = (if x< mi then x else mi);
        l= low xn (deg div 2); h = high xn (deg div 2)
      in (
        if h < length treeList ∧ ¬ (x = mi ∨ x = ma) then
          Node (Some (minn, max xn ma))
               deg
               (treeList[h:= vebt-insert (treeList ! h) l])
               (if minNull (treeList ! h) then  vebt-insert summary h else summary)
        else Node (Some (mi, ma)) deg treeList summary)
  ))))
  ⟨proof⟩
```

lemma *vebt-succ-code*[*code*]:

```
vebt-succ (Leaf a b) x = (if b∧ x = 0 then Some 1 else None)
vebt-succ (Node info  deg treeList summary) x = (if deg ≤ 1 then None else
(case info of None ⇒ None |
(Some mima) ⇒ (case mima of (mi, ma) ⇒ (
               if x < mi then (Some mi)
               else (let l = low x (deg div 2); h = high x (deg div 2) in(
                    if h < length treeList then
                           let maxlow = vebt-maxt (treeList ! h) in
                           (if maxlow ≠ None ∧ (Some l <_o  maxlow) then
                                             Some (2^(deg div 2)) *_o Some h +_o vebt-succ (treeList ! h) l
                            else let sc = vebt-succ summary h in
                            if sc = None then None
                            else Some (2^(deg div 2)) *_o sc +_o vebt-mint (treeList ! the sc) )

                    else None))))))
⟨proof⟩
```

lemma *vebt-pred-code*[*code*]:

```
vebt-pred (Leaf a b) x = (if x = 0 then None else if x = 1 then
                                 (if a then Some 0 else None) else
                          (if b then Some 1 else if a then Some 0 else None)) and
vebt-pred (Node info deg treeList summary) x =(if deg ≤ 1 then None else (
case info of None ⇒ None |
(Some mima) ⇒ (case mima of (mi, ma) ⇒ (
                     if x > ma then Some ma
                     else (let l = low x (deg div 2); h = high x (deg div 2) in
                     if h < length treeList then
                          let minlow = vebt-mint (treeList ! h) in
                          (if minlow ≠ None ∧ (Some l >_o  minlow) then
                                            Some (2^(deg div 2)) *_o Some h +_o vebt-pred (treeList ! h) l
                           else let pr = vebt-pred summary h in
```

if pr = None then (if x > mi then Some mi else None)
else Some (2^(deg div 2)) $*_o$ *pr* $+_o$ *vebt-maxt (treeList ! the pr))*
else None)))))
⟨*proof*⟩

lemma *vebt-delete-code*[*code*]:
vebt-delete (Leaf a b) x = (if x = 0 then Leaf False b else if x = 1 then Leaf a False else Leaf a b)
vebt-delete (Node info deg treeList summary) x = (
case info of
None ⇒ (Node info deg treeList summary)
| Some mima ⇒ (
if deg ≤ 1 then (Node info deg treeList summary)
else (case mima of (mi, ma) ⇒ (
if (x < mi ∨ x > ma) then (Node (Some (mi, ma)) deg treeList summary)
else if (x = mi ∧ x = ma) then (Node None deg treeList summary)
else let
*xn = (if x = mi then the (vebt-mint summary) * 2^(deg div 2)*
+ the (vebt-mint (treeList ! the (vebt-mint summary)))
else x);
minn = (if x = mi then xn else mi);
l = low xn (deg div 2);
h = high xn (deg div 2)
in
if h < length treeList then let
newnode = vebt-delete (treeList ! h) l;
newlist = treeList[h:= newnode]
in
if minNull newnode then let
sn = vebt-delete summary h;
maxn =
if xn = ma then let
maxs = vebt-maxt sn
in
if maxs = None then minn
*else 2^(deg div 2) * the maxs + the (vebt-maxt (newlist ! the maxs))*
else ma
in (Node (Some (minn, maxn)) deg newlist sn)
else let
*maxn = (if xn = ma then h * 2^(deg div 2) + the(vebt-maxt (newlist ! h))*
else ma)
in (Node (Some (minn, maxn)) deg newlist summary)
else (Node (Some (mi, ma)) deg treeList summary)
))))
⟨*proof*⟩
end

lemmas [*code*] =
VEBT-internal.high-def VEBT-internal.low-def VEBT-internal.minNull.simps

VEBT-internal.less.simps VEBT-internal.mul-def VEBT-internal.add-def
VEBT-internal.option-comp-shift.simps VEBT-internal.option-shift.simps

export-code
vebt-buildup
vebt-insert
vebt-member
vebt-maxt
vebt-mint
vebt-pred
vebt-succ
vebt-delete
checking *SML*

13.2 Correctness Lemmas

named-theorems *vebt-simps* ‹*Simplifier rules for VEBT functional interface*›

locale *vebt-inst* =
fixes *n* :: *nat*
begin

interpretation *VEBT-internal* ⟨*proof*⟩

13.2.1 Space Bound

theorem *vebt-space-linear-bound*:
fixes *t*
defines $u \equiv 2\hat{\ } n$
shows *invar-vebt t n* ⟹ *space t* ≤ *12*∗*u*
⟨*proof*⟩

13.2.2 Buildup

lemma *invar-vebt-buildup*[*vebt-simps*]: *invar-vebt* (*vebt-buildup n*) *n* ⟷ *n*>*0*
⟨*proof*⟩

lemma *set-vebt-buildup*[*vebt-simps*]: *set-vebt* (*vebt-buildup i*) = {}
⟨*proof*⟩

lemma *time-vebt-buildup*: $u = 2\hat{\ } n \Longrightarrow T_{buildup}\ n \leq 26 * u$
⟨*proof*⟩

13.2.3 Equality

lemma *set-vebt-equal*[*vebt-simps*]: *invar-vebt* t_1 *n* ⟹ *invar-vebt* t_2 *n* ⟹ t_1=t_2 ⟷ *set-vebt* t_1 = *set-vebt* t_2
⟨*proof*⟩

13.2.4 Member

lemma *set-vebt-member*[*vebt-simps*]: *invar-vebt t n* $\Longrightarrow$ *vebt-member t x* $\longleftrightarrow$ *x*$\in$*set-vebt t*
⟨*proof*⟩

theorem *time-vebt-member*: *invar-vebt t n* $\Longrightarrow$ *u* = *2^n* $\Longrightarrow$ T_{member} *t x* $\leq$ *30* + *15* $*$ *lb* (*lb u*)
⟨*proof*⟩

13.2.5 Insert

theorem *invar-vebt-insert*[*vebt-simps*]: *invar-vebt t n* $\Longrightarrow$ *x*< *2^n* $\Longrightarrow$ *invar-vebt* (*vebt-insert t x*) *n*
⟨*proof*⟩

theorem *set-vebt-insert*[*vebt-simps*]: *invar-vebt t n* $\Longrightarrow$ *x* < *2^n* $\Longrightarrow$ *set-vebt* (*vebt-insert t x*) = *set-vebt t* $\cup$ {*x*}
⟨*proof*⟩

theorem *time-vebt-insert*: *invar-vebt t n* $\Longrightarrow$ *u* = *2^n* $\Longrightarrow$ T_{insert} *t x* $\leq$ *46* + *23* $*$ *lb* (*lb u*)
⟨*proof*⟩

13.2.6 Maximum

theorem *set-vebt-maxt*: *invar-vebt t n* $\Longrightarrow$ *vebt-maxt t* = *Some x* $\longleftrightarrow$ *max-in-set* (*set-vebt t*) *x*
⟨*proof*⟩

theorem *set-vebt-maxt'*: *invar-vebt t n* $\Longrightarrow$ *vebt-maxt t* = *Some x* $\longleftrightarrow$ (*x*$\in$*set-vebt t* $\wedge$ ($\forall$ *y*$\in$*set-vebt t*. *x*$\geq$*y*))
⟨*proof*⟩

lemma *set-vebt-maxt''*[*vebt-simps*]:
invar-vebt t n $\Longrightarrow$ *vebt-maxt t* = (*if set-vebt t* = {} *then None else Some* (*Max* (*set-vebt t*)))
⟨*proof*⟩

lemma *time-vebt-maxt*: T_{maxt} *t* $\leq$ *3*
⟨*proof*⟩

13.2.7 Minimum

theorem *set-vebt-mint*[*vebt-simps*]: *invar-vebt t n* $\Longrightarrow$ *vebt-mint t* = *Some x* $\longleftrightarrow$ *min-in-set* (*set-vebt t*) *x*
⟨*proof*⟩

theorem *set-vebt-mint'*: *invar-vebt t n* $\Longrightarrow$ *vebt-mint t* = *Some x* $\longleftrightarrow$ (*x*$\in$*set-vebt t* $\wedge$ ($\forall$ *y*$\in$*set-vebt t*. *x*$\leq$*y*))
⟨*proof*⟩

lemma *set-vebt-mint''*[*vebt-simps*]:
invar-vebt t n $\Longrightarrow$ *vebt-mint t* = (*if set-vebt t* = {} *then None else Some* (*Min* (*set-vebt t*)))
⟨*proof*⟩

lemma *time-vebt-mint*: T_{mint} *t* $\leq$ *3*
⟨*proof*⟩

13.3 Emptiness determination

A tree is empty if and only if its minimum is None

lemma *vebt-minNull-mint*: *minNull t* $\longleftrightarrow$ *vebt-mint t* = *None*
⟨*proof*⟩

lemma *set-vebt-minNull*: *invar-vebt t n* $\Longrightarrow$ *minNull t* $\longleftrightarrow$ *set-vebt t* = {}
⟨*proof*⟩

lemma *time-vebt-minNull*: $T_{minNull}$ *t* $\leq$ *1*
⟨*proof*⟩

13.3.1 Successor

theorem *set-vebt-succ*: *invar-vebt t n* $\Longrightarrow$ *vebt-succ t x* = *Some sx* $\longleftrightarrow$ *is-succ-in-set* (*set-vebt t*) *x sx*
⟨*proof*⟩

lemma *set-vebt-succ'*[*vebt-simps*]: *invar-vebt t n* $\Longrightarrow$ *vebt-succ t x* = (*if* ∃ *y* ∈ *set-vebt t*. *y* > *x then Some* (*LEAST y* ∈ *set-vebt t*. *y* > *x*) *else None*)
⟨*proof*⟩

theorem *time-vebt-succ*:
fixes *t* **defines** *u* ≡ *2^n*
shows *invar-vebt t n* $\Longrightarrow$ T_{succ} *t x* $\leq$ *54* + *27* ∗ *lb* (*lb u*)
⟨*proof*⟩

13.3.2 Predecessor

theorem *set-vebt-pred*: *invar-vebt t n* $\Longrightarrow$ *vebt-pred t x* = *Some px* $\longleftrightarrow$ *is-pred-in-set* (*set-vebt t*) *x px*
⟨*proof*⟩

theorem *set-vebt-pred'*[*vebt-simps*]: *invar-vebt t n* $\Longrightarrow$
vebt-pred t x = (*if* ∃ *y* ∈ *set-vebt t*. *y* < *x then Some* (*GREATEST y*. *y* ∈ *set-vebt t* ∧ *y* < *x*) *else None*)
⟨*proof*⟩

theorem *time-vebt-pred*: **fixes** *t* **defines** *u* ≡ *2^n*
shows *invar-vebt t n* $\Longrightarrow$ T_{pred} *t x* $\leq$ *58* + *29* ∗ *lb* (*lb u*)
⟨*proof*⟩

13.3.3 Delete

theorem *invar-vebt-delete*[*vebt-simps*]: *invar-vebt t n* $\Longrightarrow$ *invar-vebt* (*vebt-delete t x*) *n*
⟨*proof*⟩

theorem *set-vebt-delete*[*vebt-simps*]: *invar-vebt t n* $\Longrightarrow$ *set-vebt* (*vebt-delete t x*) = *set-vebt t* − {*x*}
⟨*proof*⟩

theorem *time-vebt-delete*: **fixes** *t* **defines** $u \equiv 2\hat{\ }n$
shows *invar-vebt t n* $\Longrightarrow T_{delete}\ \ t\ x \leq 140 + 70 * lb\ (lb\ u)$
⟨*proof*⟩

end

13.4 Interface Usage Example

experiment
begin

definition *test n xs ys* ≡ *let*
t = *vebt-buildup n*;
t = *foldl vebt-insert t* (*0#xs*);

f = (λ*x*. *if vebt-member t x then x else the* (*vebt-pred t x*))
in
map f ys

context fixes *n* :: *nat* **begin**
interpretation *vebt-inst n* ⟨*proof*⟩

lemmas [*simp*] = *vebt-simps*

lemma [*simp*]:
assumes *invar-vebt t n* ∀ *x*∈*set xs*. *x*<*2^n*
shows *invar-vebt* (*foldl vebt-insert t xs*) *n*
⟨*proof*⟩

lemma [*simp*]:
assumes *invar-vebt t n* ∀ *x*∈*set xs*. *x*<*2^n*
shows *set-vebt* (*foldl vebt-insert t xs*) = *set-vebt t* ∪ *set xs*
⟨*proof*⟩

lemma ⟦∀ *x*∈*set xs*. *x*<*2^n*; *n*>*0*⟧ $\Longrightarrow$ *test n xs ys* = *map* (λ*y*. (*GREATEST y'*. *y'*∈*insert 0* (*set xs*) ∧ *y'*≤*y*)) *ys*
⟨*proof*⟩

end

end

end

theory *VEBT-List-Assn*
imports
Separation-Logic-Imperative-HOL/Sep-Main
HOL−Library.Rewrite

begin

13.5 Lists

fun *list-assn* :: ('*a* ⇒ '*c* ⇒ *assn*) ⇒ '*a list* ⇒ '*c list* ⇒ *assn* **where**
list-assn P [] [] = *emp*
| *list-assn P* (*a*#*as*) (*c*#*cs*) = *P a c* ∗ *list-assn P as cs*
| *list-assn* - - - = *false*

lemma *list-assn-aux-simps*[*simp*]:
list-assn P [] *l'* = (↑(*l'*=[]))
list-assn P l [] = (↑(*l*=[]))
⟨*proof*⟩

lemma *list-assn-aux-append*[*simp*]:
length l1=*length l1'* ⟹
list-assn P (*l1*@*l2*) (*l1'*@*l2'*)
= *list-assn P l1 l1'* ∗ *list-assn P l2 l2'*
⟨*proof*⟩

lemma *list-assn-aux-ineq-len*: *length l* ≠ *length li* ⟹ *list-assn A l li* = *false*
⟨*proof*⟩

lemma *list-assn-aux-append2*[*simp*]:
assumes *length l2*=*length l2'*
shows *list-assn P* (*l1*@*l2*) (*l1'*@*l2'*)
= *list-assn P l1 l1'* ∗ *list-assn P l2 l2'*
⟨*proof*⟩

lemma *list-assn-simps*[*simp*]:
(*list-assn P*) [] [] = *emp*
(*list-assn P*) (*a*#*as*) (*c*#*cs*) = *P a c* ∗ (*list-assn P*) *as cs*
(*list-assn P*) (*a*#*as*) [] = *false*
(*list-assn P*) [] (*c*#*cs*) = *false*
⟨*proof*⟩

lemma *list-assn-mono*:
⟦⋀*x x'*. *P x x'*$\Longrightarrow_A$*P' x x'*⟧ ⟹ *list-assn P l l'* $\Longrightarrow_A$ *list-assn P' l l'*
⟨*proof*⟩

lemma *list-assn-cong*[*fundef-cong*]:
  assumes $xs=xs'$ $xsi=xsi'$
  assumes $\bigwedge x\ xi.\ x \in set\ xs' \Longrightarrow xi \in set\ xsi' \Longrightarrow A\ x\ xi = A'\ x\ xi$
  shows $list\text{-}assn\ A\ xs\ xsi = list\text{-}assn\ A'\ xs'\ xsi'$
  ⟨*proof*⟩

term *prod-list*

definition $listI\text{-}assn\ I\ A\ xs\ xsi \equiv$
    $\uparrow(length\ xsi=length\ xs \wedge I \subseteq \{0..<length\ xs\})$
  $*\ Finite\text{-}Set.fold\ (\lambda i\ a.\ a * A\ (xs!i)\ (xsi!i))\ 1\ I$

lemmas *comp-fun-commute-fold-insert* =
  comp-fun-commute-on.fold-insert[**where** $S=UNIV$, *folded comp-fun-commute-def′*, *simplified*]

lemma *aux*: $Finite\text{-}Set.fold\ (\lambda i\ aa.\ aa * P\ ((a\ \#\ as)\ !\ i)\ ((c\ \#\ cs)\ !\ i))\ emp\ \{0..<Suc\ (length\ as)\}$
  $= P\ a\ c * Finite\text{-}Set.fold\ (\lambda i\ aa.\ aa * P\ (as\ !\ i)\ (cs\ !\ i))\ emp\ \{0..<length\ as\}$
⟨*proof*⟩

lemma *list-assn-conv-idx*: $list\text{-}assn\ A\ xs\ xsi = listI\text{-}assn\ \{0..<length\ xs\}\ A\ xs\ xsi$
  ⟨*proof*⟩

lemma *listI-assn-conv*: $n=length\ xs \Longrightarrow listI\text{-}assn\ \{0..<n\}\ A\ xs\ xsi = list\text{-}assn\ A\ xs\ xsi$
  ⟨*proof*⟩

lemma *listI-assn-conv′*: $n=length\ xs \Longrightarrow listI\text{-}assn\ \{0..<n\}\ A\ xs\ xsi * F = list\text{-}assn\ A\ xs\ xsi * F$
  ⟨*proof*⟩

lemma *listI-assn-finite*[*simp*]: $\neg finite\ I \Longrightarrow listI\text{-}assn\ I\ A\ xs\ xsi = false$
  ⟨*proof*⟩

find-theorems *Finite-Set.fold name*: *cong*

lemma *mult-fun-commute*: $comp\text{-}fun\text{-}commute\ (\lambda i\ (a::assn).\ a * f\ i)$
  ⟨*proof*⟩

lemma *listI-assn-weak-cong*:
  assumes *I*: $I=I'$ $A=A'$ $length\ xs=length\ xs'$ $length\ xsi=length\ xsi'$
  assumes *A*: $\bigwedge i.\ [\![i \in I;\ i<length\ xs;\ length\ xs=length\ xsi\ ]\!]$
    $\Longrightarrow xs!i = xs'!i \wedge xsi!i = xsi'!i$
  shows $listI\text{-}assn\ I\ A\ xs\ xsi = listI\text{-}assn\ I'\ A'\ xs'\ xsi'$
  ⟨*proof*⟩

lemma *listI-assn-cong*:

assumes *I*: *I*=*I'* *length xs*=*length xs'* *length xsi*=*length xsi'*
assumes *A*: ⋀*i*. ⟦*i*∈*I*; *i*<*length xs*; *length xs*=*length xsi* ⟧
⟹ *xs*!*i* = *xs'*!*i* ∧ *xsi*!*i* = *xsi'*!*i*
∧ *A* (*xs*!*i*) (*xsi*!*i*) = *A'* (*xs'*!*i*) (*xsi'*!*i*)
shows *listI-assn I A xs xsi* = *listI-assn I' A' xs' xsi'*
⟨*proof*⟩

lemma *listI-assn-insert*: *i*∉*I* ⟹ *i*<*length xs* ⟹
listI-assn (*insert i I*) *A xs xsi* = *A* (*xs*!*i*) (*xsi*!*i*) ∗ *listI-assn I A xs xsi*
⟨*proof*⟩

lemma *listI-assn-extract*:
assumes *i*∈*I* *i*<*length xs*
shows *listI-assn I A xs xsi* = *A* (*xs*!*i*) (*xsi*!*i*) ∗ *listI-assn* (*I*−{*i*}) *A xs xsi*
⟨*proof*⟩

lemma *listI-assn-reinsert*:
assumes *P* $\Longrightarrow_A$ *A* (*xs*!*i*) (*xsi*!*i*) ∗ *listI-assn* (*I*−{*i*}) *A xs xsi* ∗ *F*
assumes *i*<*length xs* *i*∈*I*
assumes *listI-assn I A xs xsi* ∗ *F* $\Longrightarrow_A$ *Q*
shows *P* $\Longrightarrow_A$ *Q*
⟨*proof*⟩

lemma *listI-assn-reinsert-upd*:
fixes *xs xsi* :: - *list*
assumes *P* $\Longrightarrow_A$ *A x xi* ∗ *listI-assn* (*I*−{*i*}) *A xs xsi* ∗ *F*
assumes *i*<*length xs* *i*∈*I*
assumes *listI-assn I A* (*xs*[*i*:=*x*]) (*xsi*[*i*:=*xi*]) ∗ *F* $\Longrightarrow_A$ *Q*
shows *P* $\Longrightarrow_A$ *Q*
⟨*proof*⟩

lemma *listI-assn-reinsert'*:
assumes *P* $\Longrightarrow_A$ *A* (*xs*!*i*) (*xsi*!*i*) ∗ *listI-assn* (*I*−{*i*}) *A xs xsi* ∗ *F*
assumes *i*<*length xs* *i*∈*I*
assumes <*listI-assn I A xs xsi* ∗ *F*>*c*<*Q*>
shows <*P*>*c*<*Q*>
⟨*proof*⟩

lemma *listI-assn-reinsert-upd'*:
fixes *xs xsi* :: - *list*
assumes *P* $\Longrightarrow_A$ *A x xi* ∗ *listI-assn* (*I*−{*i*}) *A xs xsi* ∗ *F*
assumes *i*<*length xs* *i*∈*I*
assumes <*listI-assn I A* (*xs*[*i*:=*x*]) (*xsi*[*i*:=*xi*]) ∗ *F*> *c* <*Q*>
shows <*P*> *c* <*Q*>
⟨*proof*⟩

```
lemma subst-not-in:
  assumes i∉I   i<length xs
  shows listI-assn I A (xs[i:=x1]) (xsi[i := x2]) = listI-assn I A xs xsi
  ⟨proof⟩

lemma listI-assn-subst:
  assumes i∉I  i<length xs
  shows listI-assn (insert i I) A (xs[i:=x1]) (xsi[i := x2]) = A x1 x2 ∗ listI-assn I A xs xsi
  ⟨proof⟩

lemma extract-pre-list-assn-lengthD: h ⊨ list-assn A xs xsi ⟹ length xsi = length xs
  ⟨proof⟩

method unwrap-idx for i ::nat =
  (rewrite in <⌑>-<-> list-assn-conv-idx),
  (rewrite in <⌑>-<-> listI-assn-extract[where i=i]),
  (simp split: if-splits; fail),
  (simp split: if-splits; fail)

method wrap-idx uses R =
  (rule R),
  frame-inference,
  (simp split: if-splits; fail),
  (simp split: if-splits; fail),
  (subst listI-assn-conv, (simp; fail))

method extract-pre-pure uses dest =
  (rule hoare-triple-preI | drule asm-rl[of -⊨-]),
  (determ ‹elim mod-starE dest[elim-format]›)?,
  ((determ ‹thin-tac - ⊨ -›)+)?,
  (simp (no-asm) only: triv-forall-equality)?

lemma rule-at-index:
  assumes
    1:P ⟹A list-assn A xs xsi ∗ F and
    2[simp]:i < length xs and
    3:<A (xs ! i) (xsi ! i)  ∗
    listI-assn ({0..<length xs}− {i}) A xs xsi ∗ F> c <Q'> and
    4: ⋀ r. Q' r ⟹A A (xs ! i) (xsi ! i)  ∗
   listI-assn ({0..<length xs}− {i}) A xs xsi∗ F' r
  shows
      <P>c <λ r. list-assn A xs xsi ∗ F' r>
  ⟨proof⟩


end

theory VEBT-BuildupMemImp
```

imports
VEBT-List-Assn
VEBT-Space
Deriving.Derive
VEBT-Member VEBT-Insert
HOL−Library.Countable
Time-Reasoning/Time-Reasoning VEBT-DeleteBounds
begin

14 Imperative van Emde Boas Trees

datatype *VEBTi* = *Nodei* (*nat*∗*nat*) *option nat VEBTi array VEBTi* | *Leafi bool bool*

derive *countable VEBTi*
instance *VEBTi* :: *heap* ⟨*proof*⟩

14.1 Assertions on van Emde Boas Trees

fun *vebt-assn-raw* :: *VEBT* ⇒ *VEBTi* ⇒ *assn* **where**
vebt-assn-raw (*Leaf a b*) (*Leafi ai bi*) = ↑(*ai*=*a* ∧ *bi*=*b*)
| *vebt-assn-raw* (*Node mmo deg tree-list summary*) (*Nodei mmoi degi tree-array summaryi*) = (
↑(*mmoi*=*mmo* ∧ *degi*=*deg*)
∗ *vebt-assn-raw summary summaryi*
∗ ($\exists_A$ *tree-is. tree-array* $\mapsto_a$ *tree-is* ∗ *list-assn vebt-assn-raw tree-list tree-is*)
)
| *vebt-assn-raw* - - = *false*

lemmas [*simp del*] = *vebt-assn-raw.simps*

context *VEBT-internal* **begin**

lemmas [*simp*] = *vebt-assn-raw.simps*

lemma *TBOUND-VEBT-case*[*TBOUND*]: **assumes** ⋀ *a b*. *ti* = *Leafi a b* ⟹ *TBOUND* (*f a b*) (*bnd a b*)
⋀ *info deg treeArray summary* . *ti* = *Nodei info deg treeArray summary* ⟹
TBOUND (*f′ info deg treeArray summary*) (*bnd′ info deg treeArray summary*)

shows *TBOUND* (*case ti of Leafi a b* ⇒ *f a b* |
Nodei info deg treeArray summary ⇒ *f′ info deg treeArray summary*)
(*case ti of Leafi a b* ⇒ *bnd a b* |
Nodei info deg treeArray summary ⇒ *bnd′ info deg treeArray summary*)
⟨*proof*⟩

Some Lemmas

lemma *length-corresp*:($\exists_A$ *tree-is. tree-array* $\mapsto_a$ *tree-is*) = *true* ⟹ *return* (*length tree-is*) = *Array-Time.len tree-array*

⟨*proof*⟩

lemma *heaphelp*:**assumes** $h \models$
$xa \mapsto_a$ *tree-is* ∗ *list-assn vebt-assn-raw treeList tree-is* ∗
vebt-assn-raw summary xb ∗↑(*None* = *None* ∧ *n* = *n*)∗
↑ (*xc* = *Nodei None n xa xb*)
shows $h \models$ *vebt-assn-raw* (*Node None n treeList summary*) *xc*
⟨*proof*⟩

lemma *assnle*: *list-assn vebt-assn-raw treeList tree-is* ∗ (*x13* $\mapsto_a$ *tree-is* ∗ *vebt-assn-raw summary x14*) $\Longrightarrow_A$
vebt-assn-raw summary x14 ∗ *x13* $\mapsto_a$ *tree-is* ∗ *list-assn vebt-assn-raw treeList tree-is*
⟨*proof*⟩

lemma *ext*: *y* < *length treeList* ⟹*x13* $\mapsto_a$ *tree-is* ∗ (*vebt-assn-raw summary x14* ∗
(*vebt-assn-raw* (*treeList* ! *y*) (*tree-is* ! *y*) ∗ *listI-assn* ({*0*..<*length treeList*} − {*y*}) *vebt-assn-raw treeList tree-is*))
$\Longrightarrow_A$ (*x13* $\mapsto_a$ *tree-is* ∗ *vebt-assn-raw summary x14* ∗
(*listI-assn* ({*0*..<*length treeList*} − {*y*}) *vebt-assn-raw treeList tree-is*))∗*vebt-assn-raw* (*treeList* ! *y*) (*tree-is* ! *y*)
⟨*proof*⟩

lemma *txe*:*y* < *length treeList* ⟹ *vebt-assn-raw* (*treeList* ! *y*) (*tree-is* ! *y*) ∗ *x13* $\mapsto_a$ *tree-is* ∗ *vebt-assn-raw summary x14* ∗
listI-assn ({*0*..<*length treeList*} − {*y*}) *vebt-assn-raw treeList tree-is* $\Longrightarrow_A$
vebt-assn-raw summary x14 ∗ *x13* $\mapsto_a$ *tree-is* ∗ *list-assn vebt-assn-raw treeList tree-is*
⟨*proof*⟩

lemma *recomp*: *i* < *length treeList* ⟹ *vebt-assn-raw* (*treeList* ! *i*) (*tree-is* ! *i*) ∗
listI-assn ({*0*..<*length treeList*} − {*i*}) *vebt-assn-raw treeList tree-is* ∗
x13 $\mapsto_a$ *tree-is* ∗
vebt-assn-raw summary x14 $\Longrightarrow_A$
vebt-assn-raw summary x14 ∗ *x13* $\mapsto_a$ *tree-is* ∗ *list-assn vebt-assn-raw treeList tree-is*
⟨*proof*⟩

lemma *repack*: *i* < *length treeList* ⟹
vebt-assn-raw (*treeList* ! *i*) (*tree-is* ! *i*) ∗
Rest ∗
(*x13* $\mapsto_a$ *tree-is* ∗ *vebt-assn-raw summary x14* ∗
listI-assn ({*0*..<*length treeList*} − {*i*}) *vebt-assn-raw treeList tree-is*)
$\Longrightarrow_A$ *Rest*∗ *vebt-assn-raw summary x14* ∗ *x13* $\mapsto_a$ *tree-is* ∗ *list-assn vebt-assn-raw treeList tree-is*
⟨*proof*⟩

lemma *big-assn-simp*: *h* < *length treeList* ⟹
vebt-assn-raw (*vebt-delete*(*treeList* ! *h*) *l*) *x* ∗
↑ (*xaa* = *vebt-mint* (*vebt-delete*(*treeList* ! *h*) *l*)) ∗
(*x13* $\mapsto_a$ (*tree-is* [*h* := *x*]) ∗
vebt-assn-raw summary x14 ∗

listI-assn ({0..<length treeList} − {h}) vebt-assn-raw treeList tree-is) $\Longrightarrow_A$
x13 $\mapsto_a$ tree-is[h:=x] ∗ vebt-assn-raw summary x14 ∗ ↑ (xaa = vebt-mint (vebt-delete(treeList ! h) l)) ∗
list-assn vebt-assn-raw (treeList[h:= (vebt-delete(treeList ! h) l)]) (tree-is[h:= x])
⟨proof⟩

lemma *tcd*: i < length treeList ⟹ length treeList = length treeList′ ⟹
vebt-assn-raw y x ∗ x13 $\mapsto_a$ tree-is[i:= x] ∗ vebt-assn-raw summary x14 ∗ listI-assn ({0..<length treeList} − {i}) vebt-assn-raw (treeList[i :=y]) (tree-is[i := x])
$\Longrightarrow_A$ x13 $\mapsto_a$ tree-is[i:= x] ∗ vebt-assn-raw summary x14 ∗ list-assn vebt-assn-raw (treeList[i :=y]) (tree-is[i := x])
⟨proof⟩

lemma *big-assn-simp′*: h < length treeList ==> xaa = vebt-delete (treeList ! h)l ⟹
vebt-assn-raw xaa x ∗ ↑ (xb = vebt-mint xaa) ∗
(x13 $\mapsto_a$ tree-is[h := x] ∗ vebt-assn-raw summary x14 ∗
listI-assn ({0..<length treeList} − {h}) vebt-assn-raw treeList tree-is) $\Longrightarrow_A$
(x13 $\mapsto_a$ tree-is[h:= x] ∗ vebt-assn-raw summary x14 ∗ ↑ (xb = vebt-mint xaa) ∗
list-assn vebt-assn-raw (treeList[h:= xaa]) (tree-is[h:= x]))
⟨proof⟩

lemma *refines-case-VEBTi[refines-rule]*: **assumes** ti = ti′ ⋀ a b. refines (f1 a b) (f1′ a b)
⋀ info deg treeArray summary . refines (f2 info deg treeArray summary) (f2′ info deg treeArray summary)
shows refines (case ti of Leafi a b ⇒ f1 a b |
Nodei info deg treeArray summary ⇒ f2 info deg treeArray summary)
(case ti′ of Leafi a b⇒ f1′ a b |
Nodei info deg treeArray summary ⇒ f2′ info deg treeArray summary)
⟨proof⟩

14.2 High and low Bitsequences Definition

definition *highi::nat ⇒ nat ⇒ nat Heap* **where**
highi x n == return (x div (2^n))

definition *lowi::nat ⇒ nat ⇒ nat Heap* **where**
lowi x n == return (x mod (2^n))

lemma *highi-h*: <emp> highi x n <λ r. ↑(r = high x n)>
⟨proof⟩

lemma *highi-hT*: <emp> highi x n <λ r. ↑(r = high x n)>T[1]
⟨proof⟩

lemma *lowi-h*: <emp> lowi x n <λ r. ↑(r = low x n)>
⟨proof⟩

lemma *lowi-hT*: <emp> lowi x n <λ r. ↑(r = low x n)>T[1]

⟨*proof*⟩

15 Imperative Implementation of *vebt − buildup*

```
fun replicatei::nat ⇒'a Heap  ⇒ ('a list) Heap where
  replicatei  0 x = return []|
  replicatei (Suc n) x = do{ y <− x;
                         ys <− replicatei n x;
                         return (y#ys)  }
```

lemma *time-replicate*: ⟦⋀*h*. *time x h* ≤ *c* ⟧ ⟹ *time* (*replicatei n x*) *h* ≤ (*1*+(*1*+*c*)∗*n*)
⟨*proof*⟩

lemma *TBOUND-replicate*: ⟦*TBOUND x c*⟧ ⟹ *TBOUND* (*replicatei n x*) (*1*+(*1*+*c*)∗*n*)
⟨*proof*⟩

lemma *refines-replicate*[*refines-rule*]:
refines f f' ⟹ *refines* (*replicatei n f*) (*replicatei n f'*)
⟨*proof*⟩

```
fun vebt-buildupi'::nat ⇒ VEBTi Heap where
  vebt-buildupi' 0 = return (Leafi False False)|
  vebt-buildupi' (Suc 0) = return (Leafi False False)|
  vebt-buildupi' n =  (if even n then (let half = n div 2 in do{
                          treeList <− replicatei  (2^half) (vebt-buildupi' half);
                          assert' (length treeList = 2^half);
                          trees <− Array-Time.of-list  treeList;
                          summary <− (vebt-buildupi' half);
                          return (Nodei None n  trees summary)})
             else (let half = n div 2 in  do{
                          treeList <−    replicatei (2^(Suc half)) (vebt-buildupi' half);
                          assert' (length treeList = 2^Suc half);
                          trees <− Array-Time.of-list  treeList;
                          summary <− (vebt-buildupi' (Suc half));
                          return (Nodei None n trees summary)} ))
```

end

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

```
fun vebt-buildupi::nat ⇒ VEBTi Heap where
  vebt-buildupi 0 = return (Leafi False False)|
  vebt-buildupi (Suc 0) = return (Leafi False False)|
  vebt-buildupi n =  (if even n then (let half = n div 2 in do{
                          treeList <− replicatei  (2^half) (vebt-buildupi half);
                          trees <− Array-Time.of-list  treeList;
                          summary <− (vebt-buildupi half);
                          return (Nodei None n  trees summary)})
```

else (*let half* = *n div 2 in* *do*{
treeList <− *replicatei* (*2*^(*Suc half*)) (*vebt-buildupi half*);
trees <− *Array-Time.of-list treeList*;
summary <− (*vebt-buildupi* (*Suc half*));
return (*Nodei None n trees summary*)}))

end

context *VEBT-internal* **begin**

lemma *vebt-buildupi-refines*: *refines* (*vebt-buildupi n*) (*vebt-buildupi′ n*)
⟨*proof*⟩

fun *T-vebt-buildupi* **where**
T-vebt-buildupi 0 = *Suc 0*
| *T-vebt-buildupi* (*Suc 0*) = *Suc 0*
| *T-vebt-buildupi* (*Suc* (*Suc n*)) = (
if even n then
Suc (*Suc* (*Suc* (*T-vebt-buildupi* (*Suc* (*n div 2*)) +
(*4* ∗ *2* ^(*n div 2*) + *2* ∗ (*T-vebt-buildupi* (*Suc* (*n div 2*)) ∗ *2* ^(*n div 2*))))))
else
Suc (*Suc* (*Suc* (*T-vebt-buildupi* (*Suc* (*Suc* (*n div 2*))) +
(*8* ∗ *2* ^(*n div 2*) + *4* ∗ (*T-vebt-buildupi* (*Suc* (*n div 2*)) ∗ *2* ^(*n div 2*)))))))

lemma *TBOUND-vebt-buildupi*:
defines *foo* ≡ *T-vebt-buildupi*
shows *TBOUND* (*vebt-buildupi′ n*) (*foo n*)
⟨*proof*⟩

lemma *T-vebt-buildupi*: *time* (*vebt-buildupi′ n*) *h* ≤ *T-vebt-buildupi n*
⟨*proof*⟩

lemma *repli-cons-repl*: <*Q*> *x* <λ *r*. *Q*∗ *A y r* > ⟹ <*Q*> *replicatei n x* <λ *r*. *Q*∗*list-assn A* (*replicate n y*) *r* >
⟨*proof*⟩

corollary *repli-emp*: <*emp*> *x* <λ *r*. *A y r* > ⟹ <*emp*> *replicatei n x* <λ *r*. *list-assn A* (*replicate n y*) *r* >
⟨*proof*⟩

lemma *builupi′corr*: <*emp*> *vebt-buildupi′ n* <λ *r*. *vebt-assn-raw* (*vebt-buildup n*) *r*>
⟨*proof*⟩

lemma *htt-vebt-buildupi′*: < *emp*> (*vebt-buildupi′ n*) <λ *r*. *vebt-assn-raw* (*vebt-buildup n*) *r*> *T* [*T-vebt-buildupi n*]
⟨*proof*⟩

lemma *builupicorr*: <*emp*> *vebt-buildupi n* <λ *r*. *vebt-assn-raw* (*vebt-buildup n*) *r*>

⟨proof⟩

lemma *htt-vebt-buildupi*: <emp> (vebt-buildupi n) <λ r. vebt-assn-raw (vebt-buildup n) r> T [T-vebt-buildupi n]
⟨proof⟩

Closed bound for $T - vebt - buildupi$

Amortization

lemma *T-vebt-buildupi-gq-0*: *T-vebt-buildupi n > 0*
⟨proof⟩

fun *T-vebt-buildupi′*::*nat* ⇒ *int* **where**
T-vebt-buildupi′ 0 = 1
| T-vebt-buildupi′ (Suc 0) = 1
| T-vebt-buildupi′ (Suc (Suc n)) = (
if even n then
3+(T-vebt-buildupi′ (Suc (n div 2)) +
(4 ∗ 2 ^ (n div 2) + 2 ∗ (T-vebt-buildupi′ (Suc (n div 2)) ∗ 2 ^ (n div 2))))
else
3+ (T-vebt-buildupi′ (Suc (Suc (n div 2))) +
(8 ∗ 2 ^ (n div 2) + 4 ∗ (T-vebt-buildupi′ (Suc (n div 2)) ∗ 2 ^ (n div 2)))))

lemma *Tbuildupi-buildupi′*: *T-vebt-buildupi n = T-vebt-buildupi′ n*
⟨proof⟩

fun *Tb*::*nat* ⇒ *int* **where**
Tb 0 = 3
| Tb (Suc 0) =3
| Tb (Suc (Suc n)) = (
if even n then
5+ Tb (Suc (n div 2)) + (Tb (Suc (n div 2))) ∗ 2 ^ (Suc (n div 2))
else
5 + Tb (Suc (Suc (n div 2))) + (Tb (Suc (n div 2))) ∗ 2 ^(Suc (Suc (n div 2))))

lemma *Tb-T-vebt-buildupi′*: *T-vebt-buildupi′ n ≤ Tb n − 2*
⟨proof⟩

fun *Tb′*::*nat* ⇒ *nat* **where**
Tb′ 0 = 3
| Tb′ (Suc 0) =3
| Tb′ (Suc (Suc n)) = (
if even n then
5+ Tb′ (Suc (n div 2)) + (Tb′ (Suc (n div 2))) ∗ 2 ^ (Suc (n div 2))
else
5 + Tb′ (Suc (Suc (n div 2))) + (Tb′ (Suc (n div 2))) ∗ 2 ^(Suc (Suc (n div 2))))

lemma *Tb-Tb′*: *Tb t = Tb′ t*
⟨proof⟩

lemma *Tb-T-vebt-buildupi*: *T-vebt-buildupi n* ≤ *Tb n* − *2*
⟨*proof*⟩

lemma *Tb-T-vebt-buildupi''*: *T-vebt-buildupi n* ≤ *Tb' n* − *2*
⟨*proof*⟩

lemma *Tb'-cnt*: *Tb' n* ≤ *5* ∗ *cnt'* (*vebt-buildup n*)
⟨*proof*⟩

lemma *T-vebt-buildupi-cnt'*: *T-vebt-buildupi n* ≤ *5* ∗ *cnt* (*vebt-buildup n*)
⟨*proof*⟩

lemma *T-vebt-buildupi-univ*:
assumes *u* =*2*^*n*
shows *T-vebt-buildupi n* ≤*10* ∗ *u*
⟨*proof*⟩

lemma *htt-vebt-buildupi'-univ*:
assumes *u* = *2*^*n*
shows
< *emp*> (*vebt-buildupi' n*) <λ *r*. *vebt-assn-raw* (*vebt-buildup n*) *r*> *T* [*10* ∗ *u*]
⟨*proof*⟩

We obtain the main theorem for *buildupi*

lemma *htt-vebt-buildupi-univ*:
assumes *u* = *2*^*n*
shows
< *emp*> (*vebt-buildupi n*) <λ *r*. *vebt-assn-raw* (*vebt-buildup n*) *r*> *T* [*10* ∗ *u*]
⟨*proof*⟩

lemma *vebt-buildupi-rule*: <↑ (*n* > *0*)> *vebt-buildupi n* <λ *r*. *vebt-assn-raw* (*vebt-buildup n*) *r* > *T*[*10* ∗ *2*^*n*]
⟨*proof*⟩

lemma *TBOUND-buildupi*: **assumes** *n*>*0* **shows** *TBOUND* (*vebt-buildupi n*) (*10* ∗ *2* ^ *n*)
⟨*proof*⟩

16 Minimum and Maximum Determination

end

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

fun *vebt-minti*::*VEBTi* ⇒ *nat option Heap* **where**
vebt-minti (*Leafi a b*) = (*if a then return* (*Some 0*) *else if b then return* (*Some 1*) *else return None*)|
vebt-minti (*Nodei None - - -*) = *return None*|

vebt-minti (*Nodei* (*Some* (*mi*,*ma*)) - - -) = *return* (*Some mi*)

fun *vebt-maxti*::*VEBTi* ⇒ *nat option Heap* **where**
vebt-maxti (*Leafi a b*) = (*if b then return* (*Some 1*) *else if a then return* (*Some 0*) *else return None*)|
vebt-maxti (*Nodei None* - - -) = *return None*|
vebt-maxti (*Nodei* (*Some* (*mi*,*ma*)) - - -) = *return* (*Some ma*)

end

context *VEBT-internal* **begin**

lemma *vebt-minti-h*:<*vebt-assn-raw t ti*> *vebt-minti ti* <λ*r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-mint t*)>

⟨*proof*⟩

lemma *vebt-maxti-h*:<*vebt-assn-raw t ti*> *vebt-maxti ti* <λ*r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-maxt t*)>
⟨*proof*⟩

lemma *TBOUND-vebt-maxti*[*TBOUND*]: *TBOUND* (*vebt-maxti t*) *1*
⟨*proof*⟩

lemma *TBOUND-vebt-minti*[*TBOUND*]: *TBOUND* (*vebt-minti t*) *1*
⟨*proof*⟩

lemma *vebt-minti-hT*:<*vebt-assn-raw t ti*> *vebt-minti ti* <λ*r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-mint t*)>*T*[*1*]
⟨*proof*⟩

lemma *vebt-maxti-hT*:<*vebt-assn-raw t ti*> *vebt-maxti ti* <λ*r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-maxt t*)>*T*[*1*]
⟨*proof*⟩

lemma *vebt-maxtilist*:*i* < *length ts* ⟹
<*list-assn vebt-assn-raw ts tsi*> *vebt-maxti* (*tsi* ! *i*)
< λ *r*. ↑(*r* = *vebt-maxt* (*ts* ! *i*)) ∗*list-assn vebt-assn-raw ts tsi*>
⟨*proof*⟩

lemma *vebt-mintilist*:*i* < *length ts* ⟹
<*list-assn vebt-assn-raw ts tsi*> *vebt-minti* (*tsi* ! *i*)
< λ *r*. ↑(*r* = *vebt-mint* (*ts* ! *i*)) ∗*list-assn vebt-assn-raw ts tsi*>
⟨*proof*⟩

17 Membership Test on imperative van Emde Boas Trees

end

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

partial-function (*heap-time*) *vebt-memberi*:: *VEBTi* ⇒ *nat* ⇒ *bool Heap* **where**

```
vebt-memberi t x =
(case t of
    (Leafi a b ) ⇒ return (if x = 0 then a else if x=1 then b else False) |
    (Nodei info deg treeList summary ) ⇒ (
        case info of None ⇒ return False |
        (Some (mi, ma)) ⇒ ( if deg ≤ 1 then return False else (
                            if x = mi then return True else
                            if x = ma then return True else
                            if x < mi then return False else
                            if x > ma then return False else
                            (do {
                                h ← highi x (deg div 2);
                                l ← lowi x (deg div 2);
                                len ← Array-Time.len treeList;
                                if h < len then do {
                                th ← Array-Time.nth treeList h;
                                vebt-memberi th l
                                } else return False
                            })))))
```

end

context *VEBT-internal* **begin**

partial-function (*heap-time*) *vebt-memberi'*:: *VEBT* ⇒ *VEBTi* ⇒ *nat* ⇒ *bool Heap* **where**

```
vebt-memberi' t ti x =
(case ti of
    (Leafi a b ) ⇒ return (if x = 0 then a else if x=1 then b else False) |
    (Nodei info deg treeArray summary ) ⇒ ( do {assert' (is-Node t);
        case info of None ⇒ return False |
        (Some (mi, ma)) ⇒ ( if deg ≤ 1 then return False else (
                            if x = mi then return True else
                            if x = ma then return True else
                            if x < mi then return False else
                            if x > ma then return False else
                            (do {
                                let (info',deg',treeList,summary') =
                                 (case t of (Node info' deg' treeList summary') ⇒
                                  (info', deg', treeList, summary'));
                             assert'(info= info' ∧ deg = deg');
                                h ← highi x (deg div 2);
                                l ← lowi x (deg div 2);
                                assert'(l = low x (deg div 2) ∧ h = high x (deg div 2));
                                len ← Array-Time.len treeArray;
                                assert'(len = length treeList);
                                if h < len then do {
                                        assert'(h = high x (deg div 2) ∧ h < length treeList);
```

th ← *Array-Time.nth treeArray h*;
vebt-memberi′ (*treeList* ! *h*) *th l* }
else return False
}))}}))

lemma *highsimp*: *return* (*high x n*) = *highi x n*
⟨*proof*⟩

lemma *lowsimp*: *return* (*low x n*) = *lowi x n*
⟨*proof*⟩

lemma *TBOUND-highi*[*TBOUND*]: *TBOUND* (*highi x n*) *1*
⟨*proof*⟩

lemma *TBOUND-lowi*[*TBOUND*]: *TBOUND* (*lowi x n*) *1*
⟨*proof*⟩

Correctness of $vebt - memberi$

lemma *vebt-memberi′-rf-abstr*: <*vebt-assn-raw t ti*> *vebt-memberi′ t ti x* <λ*r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-member t x*)>
⟨*proof*⟩

lemma *TBOUND-vebt-memberi*:
defines *foo-def*: ⋀ *t x*. *foo t x* ≡ *4* ∗ (*1*+*height t*)
shows *TBOUND* (*vebt-memberi′ t ti x*) (*foo t x*)
⟨*proof*⟩

lemma *vebt-memberi-refines*: *refines* (*vebt-memberi ti x*) (*vebt-memberi′ t ti x*)
⟨*proof*⟩

lemma *htt-vebt-memberi*:
<*vebt-assn-raw t ti*>*vebt-memberi ti x* <λ *r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-member t x*)>*T*[*5* + *5* ∗ *height t*]
⟨*proof*⟩

lemma *htt-vebt-memberi-invar-vebt*: **assumes** *invar-vebt t n* **shows**
<*vebt-assn-raw t ti*> *vebt-memberi ti x* <λ *r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-member t x*)>*T*[*5* + *5* ∗ (*nat* ⌈*lb n* ⌉)]
⟨*proof*⟩

17.1 $minNulli$: empty tree?

fun *minNulli*::*VEBTi* ⇒ *bool Heap* **where**
minNulli (*Leafi False False*) = *return True*|
minNulli (*Leafi - -*) = *return False*|
minNulli (*Nodei None - - -*) = *return True*|
minNulli (*Nodei* (*Some -*) *- - -*) = *return False*

lemma *minNulli-rule*[*sep-heap-rules*]: <*vebt-assn-raw t ti*> *minNulli ti* <λ*r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *minNull t*)>

⟨proof⟩

lemma *TBOUND-minNulli*[*TBOUND*]: *TBOUND* (*minNulli t*) *1*
⟨proof⟩

lemma *minNrulli-ruleT*:
<vebt-assn-raw t ti> minNulli ti <λr. vebt-assn-raw t ti ∗ ↑(r = minNull t)>T[1]
⟨proof⟩

18 Imperative *vebt − insert* to van Emde Boas Tree

end

context begin
interpretation *VEBT-internal* ⟨proof⟩

partial-function (*heap-time*) *vebt-inserti*::*VEBTi* ⇒ *nat* ⇒*VEBTi Heap* **where**
vebt-inserti t x = (case t of
(Leafi a b) ⇒ (if x=0 then return (Leafi True b) else if x=1
then return (Leafi a True) else return (Leafi a b)) |
(Nodei info deg treeArray summary) ⇒ (case info of None ⇒
if deg ≤ 1 then
return (Nodei info deg treeArray summary)
else
return (Nodei (Some (x,x)) deg treeArray
summary)|
(Some minma) ⇒
(if deg ≤ 1
then return (Nodei info deg treeArray summary)
else (do{
mi <− return (fst minma);
ma <− return (snd minma);
xn <− (if x < mi then return mi else return x);
minn <− (if x < mi then return x else return mi);
l<− lowi xn (deg div 2);
h <− highi xn (deg div 2);
len ← Array-Time.len treeArray;
if h < len ∧ ¬ (x = mi ∨ x = ma) then do {
node <− Array-Time.nth treeArray h;
empt <− minNulli node;
newnode <− vebt-inserti node l;
newarray <− Array-Time.upd h newnode treeArray;
newsummary<−(if empt then
vebt-inserti summary h
else return summary);
man <− (if xn > ma then return xn else return ma);
return (Nodei (Some (minn, man)) deg newarray
newsummary)}

```
                                              else return (Nodei (Some (mi,ma)) deg treeArray
summary)
                                       }))))
```

end

context *VEBT-internal* **begin**

partial-function (*heap-time*) *vebt-inserti′*:: *VEBT* ⇒ *VEBTi* ⇒ *nat* ⇒ *VEBTi Heap* **where**

```
 vebt-inserti′ t ti x = (case ti of
          (Leafi a b) ⇒ (if x=0 then return (Leafi True b) else if x=1
                    then return (Leafi a True) else return (Leafi a b)) |
          (Nodei info deg treeArray summary) ⇒ ( case info of None ⇒
                                      if deg ≤ 1 then
                                        return (Nodei info deg treeArray summary)
                                      else
                                           return (Nodei (Some (x,x)) deg treeArray
summary)|
                                 (Some minma) ⇒
                                    ( if deg ≤ 1
                                 then return (Nodei info deg treeArray summary)
                                  else (
                          do{
                            assert′ (is-Node t);
                            let (info′,deg′,treeList,summary′) =
                            (case t of (Node info′ deg′ treeList summary′) ⇒
                            (info′, deg′, treeList, summary′));
                            assert′(info= info′ ∧ deg = deg′);
                                  let (mi′, ma′) = (the info′);
                                  mi <− return (fst minma);
                                  ma <− return (snd minma);
                                  xn <− (if x < mi then return mi else return x);
                                  let xn′ = (if x < mi′ then mi′ else x);
                                  minn <− (if x < mi then return x else return mi);
                                  let minn′ = (if x < mi′ then x else mi′);
                                  l<− lowi xn (deg div 2);
                                  assert′ (l = low xn′ (deg′ div 2));
                                  h <− highi xn (deg div 2);
                                  len ← Array-Time.len treeArray;
                                  if h < len ∧ ¬ (x = mi ∨ x = ma) then do {
                                    assert′ (h = high xn′ (deg′ div 2));
                                    assert′( h < length treeList);
                                    node <− Array-Time.nth treeArray h;
                                    empt <− minNulli node;
                                    assert′ (empt = minNull (treeList ! h));
                                    newnode <− vebt-inserti′ (treeList ! h) node l;
                                    newarray <− Array-Time.upd h newnode treeArray;
                                    newsummary<−(if empt then
                                          vebt-inserti′ summary′ summary h
```

else return summary);
man <− (*if xn* > *ma then return xn else return ma*);
return (*Nodei* (*Some* (*minn*, *man*)) *deg newarray newsummary*)}
else return (*Nodei* (*Some* (*mi*,*ma*)) *deg treeArray summary*)
}))))

lemmas *listI-assn-wrap-insert* = *listI-assn-reinsert-upd*′[
where *x*=*VEBT-Insert.vebt-insert* - - **and** *A*=*vebt-assn-raw*]

lemma *vebt-inserti*′*-rf-abstr*: <*vebt-assn-raw t ti*> *vebt-inserti*′ *t ti x* <λ*r*. *vebt-assn-raw* (*vebt-insert t x*) *r* >
⟨*proof*⟩

lemma *TBOUND-minNull*: *minNull t* ⟹ *TBOUND* (*vebt-inserti*′ *t ti x*) *1*
⟨*proof*⟩

lemma *TBOUND-vebt-inserti*:
defines *foo-def*: ⋀ *t x*. *foo t x* ≡ *if minNull t then 1 else 13* ∗ (*1*+*height t*)
shows *TBOUND* (*vebt-inserti*′ *t ti x*) (*foo t x*)
⟨*proof*⟩

lemma *vebt-inserti-refines*: *refines* (*vebt-inserti ti x*) (*vebt-inserti*′ *t ti x*)
⟨*proof*⟩

lemma *htt-vebt-inserti*:
<*vebt-assn-raw t ti*> *vebt-inserti ti x* <λ *r*. *vebt-assn-raw* (*vebt-insert t x*) *r*>*T*[*13* + *13* ∗ *height t*]
⟨*proof*⟩

lemma *htt-vebt-inserti-invar-vebt*: **assumes** *invar-vebt t n* **shows**
<*vebt-assn-raw t ti*> *vebt-inserti ti x* <λ *r*. *vebt-assn-raw* (*vebt-insert t x*) *r*>*T*[*13* + *13* ∗ (*nat* ⌈*lb n* ⌉)]
⟨*proof*⟩

end
end

theory *VEBT-SuccPredImperative*
imports *VEBT-BuildupMemImp VEBT-Succ VEBT-Pred*
begin

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

19 Imperative Successor

partial-function (*heap-time*) *vebt-succi*::*VEBTi* ⇒ *nat* ⇒ (*nat option*) *Heap* **where**

```
vebt-succi t x = (case t of (Leafi a b) ⇒(if x = 0 then (if b then return (Some 1) else return None)
                        else return None)|
              (Nodei info deg treeArray summary) ⇒ (
               case info of None ⇒ return None |
              (Some mima) ⇒ ( if deg ≤ 1 then return None else
                      (if x < fst mima then return (Some (fst mima)) else
                       if x ≥ snd mima then return None else
                        do {
                          l <− lowi x (deg div 2);
                          h <− highi x (deg div 2);
                          aktnode <− Array-Time.nth treeArray h;
                            maxlow <− vebt-maxti aktnode;
                            if (maxlow ≠ None ∧ (Some l <o  maxlow))
                            then do {
                                    succy <− vebt-succi aktnode l;
                                     return ( Some (2^(deg div 2)) *o Some h +o succy)
                                    }
                            else do {
                                    succsum <−  vebt-succi summary h;
                                    if succsum = None then
                                       return None
                                     else
                                       do{
                                      nextnode <− Array-Time.nth treeArray (the succsum);
                                          minnext <− vebt-minti nextnode;
                                      return (Some (2^(deg div 2)) *o succsum +o minnext)
                                         }
                                    }

                           })
)))
```

end

context *VEBT-internal* **begin**

partial-function (*heap-time*) *vebt-succi'*::*VEBT* ⇒ *VEBTi* ⇒ *nat* ⇒ (*nat option*) *Heap* **where**

```
vebt-succi' t ti x = (case ti of (Leafi a b) ⇒(if x = 0 then (if b then return (Some 1) else return
None)
                        else return None)|
              (Nodei info deg treeArray summary) ⇒ do { assert'( is-Node t);
              let (info',deg',treeList,summary') =
             (case t of Node info' deg' treeList summary' ⇒ (info',deg',treeList,summary'));
               assert'(info'=info ∧ deg'=deg ∧ is-Node t);
              case info of None ⇒ return None |
             (Some mima) ⇒ (if deg ≤ 1 then return None else
                      (if x < fst mima then return (Some (fst mima)) else
                       if x ≥ snd mima then return None else
                        do {
```

```
l <− lowi x (deg div 2);
h <− highi x (deg div 2);

assert'(l = low x (deg div 2));
assert'(h = high x (deg div 2));
assert'(h < length treeList);

aktnode <− Array-Time.nth treeArray h;
let aktnode' = treeList!h;

    maxlow <− vebt-maxti aktnode;
    assert' (maxlow = vebt-maxt aktnode');
    if (maxlow ≠ None ∧ (Some l <o  maxlow))
    then do {
            succy <− vebt-succi' aktnode' aktnode l;
             return ( Some (2^(deg div 2)) *o Some h +o succy)
            }
    else do {
            succsum <−  vebt-succi' summary' summary h;
       assert'(succsum = None ⟷ vebt-succ summary' h = None);
            if succsum = None then do{
                return None}
              else
                do{
               nextnode <− Array-Time.nth treeArray (the succsum);
                    minnext <− vebt-minti nextnode;
                return (Some (2^(deg div 2)) *o succsum +o minnext)
                  }
            }

 })
)})
```

theorem *vebt-succi'-rf-abstr*:*invar-vebt t n* ⟹ <*vebt-assn-raw t ti*> *vebt-succi' t ti x* <λ*r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-succ t x*)>
⟨*proof*⟩

lemma *TBOUND-vebt-succi*:
defines *foo-def*: ⋀ *t x*. *foo t x* ≡ *7* ∗ (*1*+*height t*)
shows *TBOUND* (*vebt-succi' t ti x*) (*foo t x*)
⟨*proof*⟩

lemma *vebt-succi-refines*: *refines* (*vebt-succi ti x*) (*vebt-succi' t ti x*)
⟨*proof*⟩

lemma *htt-vebt-succi*: **assumes** *invar-vebt t n*
shows <*vebt-assn-raw t ti*> *vebt-succi ti x* <λ *r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-succ t x*) >*T*[*7* + *7*∗(*nat* ⌈*lb n*⌉)]
⟨*proof*⟩

end

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

partial-function (*heap-time*) *vebt-predi*::*VEBTi* ⇒ *nat* ⇒ (*nat option*) *Heap* **where**

```
vebt-predi t x = (case t of (Leafi a b) ⇒(if x ≥ 2then (if b then return (Some 1) else if a then return
(Some 0) else return None)
                        else if x = 1 then (if a then return (Some 0) else return None) else
return None)|
          (Nodei info deg treeArray summary) ⇒ (
        case info of None ⇒ return None |
         (Some mima) ⇒ ( if deg ≤ 1 then return None else
                (if x > snd mima then return (Some (snd mima)) else
                  do {
                    l <− lowi x (deg div 2);
                    h <− highi x (deg div 2);
                      aktnode <− Array-Time.nth treeArray h;
                      minlow <− vebt-minti aktnode;
                      if (minlow ≠ None ∧ (Some l >o  minlow))
                      then do {
                              predy <− vebt-predi aktnode l;
                               return ( Some (2^(deg div 2)) *o Some h +o predy)
                              }
                      else do {
                              predsum <−  vebt-predi summary h;
                              if predsum = None then
                                  if x > fst mima then
                                      return (Some (fst mima))
                                      else
                                        return None
                                else
                                  do{
                                 nextnode <− Array-Time.nth treeArray (the predsum);
                                      maxnext <− vebt-maxti nextnode;
                                 return (Some (2^(deg div 2)) *o predsum +o maxnext)
                                    }
                              }
                      }))))
```

end
context *VEBT-internal* **begin**

20 Imperative Predecessor

partial-function (*heap-time*) *vebt-predi'*::*VEBT* ⇒ *VEBTi* ⇒ *nat* ⇒ (*nat option*) *Heap* **where**

```
vebt-predi' t ti x = (case ti of (Leafi a b) ⇒(if x ≥ 2then (if b then return (Some 1) else if a then
```

```
return (Some 0) else return None)
                                   else if x = 1 then (if a then return (Some 0) else return None) else
return None)|
            (Nodei info deg treeArray summary) ⇒ (  do { assert'( is-Node t);
            let (info',deg',treeList,summary') =
           (case t of Node info' deg' treeList summary' ⇒ (info',deg',treeList,summary'));
            assert'(info'=info ∧ deg'=deg ∧ is-Node t);
          case info of None ⇒ return None |
           (Some mima) ⇒ ( if deg ≤ 1 then return None else
                   (if x > snd mima then return (Some (snd mima)) else
                     do {
                        l <− lowi x (deg div 2);
                        h <− highi x (deg div 2);

                        assert'(l = low x (deg div 2));
                        assert'(h = high x (deg div 2));
                        assert'(h < length treeList);

                          aktnode <− Array-Time.nth treeArray h;
                           let aktnode' = treeList!h;
                          minlow <− vebt-minti aktnode;
                          assert' (minlow = vebt-mint aktnode');

                          if (minlow ≠ None ∧ (Some l >o  minlow))
                          then do {
                               predy <− vebt-predi' aktnode' aktnode l;
                                return ( Some (2^(deg div 2)) *o Some h +o predy)
                               }
                          else do {
                               predsum <−  vebt-predi' summary' summary h;
                            assert'(predsum = None ⟷ vebt-pred summary' h = None);
                               if predsum = None then
                                 if x > fst mima then
                                   return (Some (fst mima))
                                   else
                                    return None
                                else
                                 do{
                                 nextnode <− Array-Time.nth treeArray (the predsum);
                                   maxnext <− vebt-maxti nextnode;
                                 return (Some (2^(deg div 2)) *o predsum +o maxnext)
                                   }
                               }
                      }))}))
```

theorem *vebt-pred'-rf-abstr*:*invar-vebt t n* ⟹ <*vebt-assn-raw t ti*> *vebt-predi' t ti x* <λ*r*. *vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-pred t x*)>
⟨*proof*⟩

lemma *TBOUND-vebt-predi*:
defines *foo-def*: ⋀ *t x. foo t x* ≡ *7* ∗ (*1*+*height t*)
shows *TBOUND* (*vebt-predi′ t ti x*) (*foo t x*)
⟨*proof*⟩

lemma *vebt-predi-refines*: *refines* (*vebt-predi ti x*) (*vebt-predi′ t ti x*)
⟨*proof*⟩

lemma *htt-vebt-predi*: **assumes** *invar-vebt t n*
shows <*vebt-assn-raw t ti*> *vebt-predi ti x* <λ *r. vebt-assn-raw t ti* ∗ ↑(*r* = *vebt-pred t x*) >*T*[*7* + *7*∗(*nat* ⌈*lb n*⌉)]
⟨*proof*⟩

end
end

theory *VEBT-DelImperative* **imports** *VEBT-DeleteCorrectness VEBT-SuccPredImperative*
begin

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

21 Imperative Delete

partial-function (*heap-time*) *vebt-deletei*::*VEBTi* ⇒ *nat* ⇒ *VEBTi Heap* **where**

```
vebt-deletei t x = (case t of (Leafi a b) ⇒ (if x = 0 then return (Leafi False b) else
                                   if x = 1 then return (Leafi a False) else
                                    return (Leafi a b)) |
                         (Nodei info deg treeArray summary) ⇒ (
                              if deg ≤ 1 then return  (Nodei info deg treeArray summary) else
                                  case info of None ⇒ return  (Nodei info deg treeArray summary)|
                                         (Some mima) ⇒ ( if x < fst mima ∨ x > snd mima then return
(Nodei info deg treeArray summary)
                                                    else  if fst mima = x ∧ snd mima = x then return  (Nodei
None deg treeArray summary)
                                                  else do{ xminew <− (if x = fst mima then do {
                                                                    firstcluster <− vebt-minti summary;
                                                                   firsttree <− Array-Time.nth  treeArray (the
firstcluster);
                                                                    mintft <− vebt-minti firsttree;
                                                                 let xn = (2^(deg div 2) ∗ (the firstcluster) +
                                                                               (the mintft) );
                                                                    return (xn, xn)
                                                                           }
                                                                   else return (x, fst mima));
                                                        let xnew = fst xminew;
                                                        let minew = snd xminew;
                                                        h <− highi xnew (deg div 2);
                                                        l <− lowi xnew (deg div 2);
```

```
                                  aktnode <− Array-Time.nth treeArray h;
                                  aktnode'<−vebt-deletei aktnode l;
                                  treeArray' <− Array-Time.upd h aktnode' treeArray;
                                  miny <− vebt-minti aktnode';
                                 (if (miny = None) then
                                  do{
                                      summary' <−vebt-deletei summary h;
                                      ma <−  (if xnew = snd mima then
                                      do{
                                          summax <− vebt-maxti summary';
                                          if summax = None then
                                                return minew
                                          else do{
                                               maxtree <− Array-Time.nth treeArray' (the
summax);

                                                  mofmtree<− vebt-maxti maxtree;
                                                  return (the summax ∗ 2^(deg div 2) +
                                                                 the mofmtree )
                                                  }
                                         }
                                          else return (snd mima));
                                       return (Nodei (Some (minew, ma)) deg treeArray'
summary')

                                  } else  if xnew = snd mima then
                                    do{
                                        nextree <− Array-Time.nth treeArray' h;
                                        maxnext<− vebt-maxti nextree;
                                        let ma = h ∗ 2^(deg div 2) +
                                                       (the maxnext);
                                      return (Nodei (Some ( minew, ma)) deg treeArray'
summary)

                                       }
                                      else return (Nodei (Some (minew, snd mima)) deg
treeArray' summary) )

                                  })))
```

end

context *VEBT-internal* **begin**

Some general lemmas

lemma *midextr*: $(P * Q * Q' * R \Longrightarrow_A X) \Longrightarrow (P * R * Q * Q' \Longrightarrow_A X)$
⟨*proof*⟩

lemma *groupy*: $A * B * (C * D) \Longrightarrow_A X \Longrightarrow A * B * C * D \Longrightarrow_A X$
⟨*proof*⟩

lemma *swappa*: $B * A * C \Longrightarrow_A X \Longrightarrow A * B * C \Longrightarrow_A X$
⟨*proof*⟩

lemma *mulcomm*: $(i{::}nat) * (2 * 2\ \hat{}\ (va\ div\ 2)) = (2 * 2\ \hat{}\ (va\ div\ 2)) * i$
⟨*proof*⟩

Modified function with ghost variable

partial-function (*heap-time*) *vebt-deletei′*::*VEBT* ⇒ *VEBTi* ⇒ *nat* ⇒ *VEBTi Heap* **where**

```
vebt-deletei′ t ti x = (case ti of (Leafi a b) ⇒ (if x = 0 then return (Leafi False b) else
                                 if x = 1 then return (Leafi a False) else
                                  return (Leafi a b)) |
                      (Nodei info deg treeArray summary) ⇒ (
                           do { assert′( is-Node t);
                     let (info′,deg′,treeList,summary′) =
                   (case t of Node info′ deg′ treeList summary′
                              ⇒ (info′,deg′,treeList,summary′));
                     assert′(info′=info ∧ deg′=deg ∧ is-Node t);
                          if deg ≤ 1 then return (Nodei info deg treeArray summary) else
                              case info of None ⇒ return (Nodei info deg treeArray summary)|
                                   (Some mima) ⇒ (
                                        if x < fst mima ∨ x > snd mima then return (Nodei info deg
treeArray summary)
                                          else if fst mima = x ∧ snd mima = x then return (Nodei
None deg treeArray summary)
                                        else do{ xminew <− (if x = fst mima then do {
                                                      firstcluster <− vebt-minti summary;
                                                     firsttree <− Array-Time.nth treeArray (the
firstcluster);
                                                      mintft <− vebt-minti firsttree;
                                                    let xn = (2^(deg div 2) * (the firstcluster) +
                                                                (the mintft) );
                                                      return (xn, xn)
                                                          }
                                                     else return (x, fst mima));
                                             let xnew = fst xminew;
                                             let xn′ =
                                              (if x = fst (the info′)
                                               then the (vebt-mint summary′) * 2^(deg div 2)
                                            + the (vebt-mint (treeList ! the (vebt-mint summary′)))
                                               else x);
                                             assert′ (xnew = xn′);
                                             let minew = snd xminew;
                                            assert′ (minew = (if x = fst (the info′) then xn′ else fst
(the info′)));
                                             h <− highi xnew (deg div 2);
                                             assert′ (h = high xnew (deg div 2));
                                             assert′( h < length treeList);
                                             l <− lowi xnew (deg div 2);
                                             assert′(l = low xnew (deg div 2));
                                             aktnode <− Array-Time.nth treeArray h;
                                             aktnode′<−vebt-deletei′ (treeList ! h) aktnode l;
```

```
treeArray' <− Array-Time.upd h aktnode' treeArray;
let funnode = vebt-delete (treeList ! h) l;
let treeList' = treeList[h:= funnode];
  miny <− vebt-minti aktnode';
  assert' (miny = vebt-mint funnode);
 (if (miny = None) then
  do{
      summaryi' <−vebt-deletei' summary' summary h;
      ma <−  (if xnew = snd mima then
      do{
          summax <− vebt-maxti summaryi';
                 assert' (summax = vebt-maxt (vebt-delete summary' h));

          if summax = None then
                return minew
          else do{
               maxtree <− Array-Time.nth treeArray' (the summax);

                   mofmtree<− vebt-maxti maxtree;
                   return (the summax ∗ 2^(deg div 2) +
                                the mofmtree )
                   }
        }
          else return (snd mima));
     return (Nodei (Some (minew, ma)) deg treeArray' summaryi')

  } else  if xnew = snd mima then
    do{
        nextree <− Array-Time.nth treeArray' h;
        maxnext<− vebt-maxti nextree;
        assert' (maxnext = vebt-maxt (treeList' ! h));
        let ma = h ∗ 2^(deg div 2) +
                      (the maxnext);
      return (Nodei (Some ( minew, ma)) deg treeArray' summary)

      }
      else return (Nodei (Some (minew, snd mima)) deg treeArray' summary) )

  })}))
```

theorem *deleti'-rf-abstr*: *invar-vebt t n* $\Longrightarrow$ *<vebt-assn-raw t ti> vebt-deletei' t ti x< vebt-assn-raw (vebt-delete t x)>*
⟨*proof*⟩

lemma *TBOUND-vebt-deletei*:
  defines *foo-def*: $\bigwedge$ *t x. foo t x* ≡ *if minNull (vebt-delete t x) then 1 else 20* ∗ *(1+height t)*
  shows *TBOUND (vebt-deletei' t ti x) (foo t x)*
⟨*proof*⟩

lemma *vebt-deletei-refines*: *refines* (*vebt-deletei ti x*) (*vebt-deletei′ t ti x*)
⟨*proof*⟩

lemma *htt-vebt-deletei*: **assumes** *invar-vebt t n*
shows <*vebt-assn-raw t ti*> *vebt-deletei ti x* <λ *r*. *vebt-assn-raw* (*vebt-delete t x*) *r* >*T*[*20* + *20*∗(*nat* ⌈*lb n*⌉)]
⟨*proof*⟩

end
end

22 Imperative Interface

theory *VEBT-Intf-Imperative*
imports
VEBT-Definitions
VEBT-Uniqueness
VEBT-Member
VEBT-Insert VEBT-InsertCorrectness
VEBT-MinMax
VEBT-Pred VEBT-Succ
VEBT-Delete VEBT-DeleteCorrectness
VEBT-Bounds
VEBT-DeleteBounds
VEBT-Space
VEBT-Intf-Functional
VEBT-List-Assn
VEBT-BuildupMemImp
VEBT-SuccPredImperative
VEBT-DelImperative
begin

22.1 Code Export

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

lemmas [*code*] = *replicatei.simps vebt-memberi.simps highi-def lowi-def vebt-inserti.simps*
minNulli.simps vebt-succi.simps vebt-predi.simps vebt-deletei.simps

greater.simps

end

export-code
vebt-buildupi
vebt-memberi

vebt-inserti
vebt-maxti vebt-minti
vebt-predi vebt-succi
vebt-deletei

checking *SML-imp*

22.2 Interface

definition *vebt-assn::nat ⇒ nat set ⇒ VEBTi ⇒ assn* **where**
vebt-assn n s ti ≡ ∃$_A$ *t. vebt-assn-raw t ti* ∗ ↑(*s* = *set-vebt t* ∧ *invar-vebt t n*)

22.2.1 Buildup

context begin
interpretation *VEBT-internal* ⟨*proof*⟩

interpretation *vebt-inst* **for** *n* ⟨*proof*⟩

lemma *vebt-buildupi-rule-basic*[*sep-heap-rules*]: *n* > *0* ⟹ <*emp*> *vebt-buildupi n* <λ *r. vebt-assn n* {} *r* >
⟨*proof*⟩

lemma *vebt-buildupi-rule*: <↑ (*n* > *0*)> *vebt-buildupi n* <λ *r. vebt-assn n* {} *r* > *T*[*10* ∗ *2*^*n*]
⟨*proof*⟩

22.2.2 Member

lemma *vebt-memberi-rule*: <*vebt-assn n s ti*> *vebt-memberi ti x* <λ *r. vebt-assn n s ti* ∗ ↑(*r* = (*x* ∈ *s*))>*T*[*5* + *5* ∗ (*nat* ⌈*lb n* ⌉)]
⟨*proof*⟩

22.2.3 Insert

lemma *vebt-inserti-rule*: *x* < *2*^*n* ⟹ <*vebt-assn n s ti*> *vebt-inserti ti x* <λ *r. vebt-assn n* (*s* ∪ {*x*}) *r* >*T*[*13* + *13* ∗ (*nat* ⌈*lb n* ⌉)]
⟨*proof*⟩

22.2.4 Maximum

lemma *vebt-maxti-rule*: <*vebt-assn n s ti*> *vebt-maxti ti* <λ *r. vebt-assn n s ti* ∗ ↑(*r* = *Some y* ⟷ *max-in-set s y*)>*T*[*1*]
⟨*proof*⟩

22.2.5 Minimum

lemma *vebt-minti-rule*: <*vebt-assn n s ti*> *vebt-minti ti* <λ *r. vebt-assn n s ti* ∗ ↑(*r* = *Some y* ⟷ *min-in-set s y*)>*T*[*1*]
⟨*proof*⟩

22.2.6 Successor

lemma *vebt-succi-rule*: <*vebt-assn n s ti*> *vebt-succi ti x* <λ *r*. *vebt-assn n s ti* ∗ ↑(*r* = *Some y* ⟷ *is-succ-in-set s x y*)>*T*[*7* + *7* ∗ (*nat* ⌈*lb n* ⌉)]
⟨*proof*⟩

22.2.7 Predecessor

lemma *vebt-predi-rule*: <*vebt-assn n s ti*> *vebt-predi ti x* <λ *r*. *vebt-assn n s ti* ∗ ↑(*r* = *Some y* ⟷ *is-pred-in-set s x y*)>*T*[*7* + *7* ∗ (*nat* ⌈*lb n* ⌉)]
⟨*proof*⟩

22.2.8 Delete

lemma *vebt-deletei-rule*: <*vebt-assn n s ti* > *vebt-deletei ti x* <λ *r*. *vebt-assn n* (*s* − {*x*}) *r* >*T*[*20* + *20* ∗ (*nat* ⌈*lb n* ⌉)]
⟨*proof*⟩

22.3 Setup of VCG

lemmas *vebt-heap-rules*[*THEN htt-htD*,*sep-heap-rules*] =
vebt-buildupi-rule
vebt-memberi-rule
vebt-inserti-rule
vebt-maxti-rule
vebt-minti-rule
vebt-succi-rule
vebt-predi-rule
vebt-deletei-rule

end
end

23 Interface Usage Example

theory *VEBT-Example*
imports *VEBT-Intf-Imperative VEBT-Example-Setup*
begin

23.1 Test Program

definition *test n xs ys* ≡ *do* {
t ← *vebt-buildupi n*;
t ← *mfold* (λ*x s*. *vebt-inserti s x*) (*0*#*xs*) *t*;

let f = (λ*x*. if_m *vebt-memberi t x then return x else the* $\$_m$ (*vebt-predi t x*));

mmap f ys
}

23.2 Correctness without Time

The non-time part of our datastructure is fully integrated into sep-auto

lemma *fold-list-rl*[*sep-heap-rules*]: ∀ *x*∈*set xs*. *x*<*2^n* ⟹ *hoare-triple*
(*vebt-assn n s t*)
(*mfold* (λ*x s*. *vebt-inserti s x*) *xs t*)
(λ*t*′. *vebt-assn n* (*s* ∪ *set xs*) *t*′)
⟨*proof*⟩

lemma *test-hoare*: ⟦∀ *x*∈*set xs*. *x*<*2^n*; *n*>*0*⟧ ⟹
<*emp*> (*test n xs ys*) <λ*r*. ↑(*r* = *map* (λ*y*. (*GREATEST y*′. *y*′∈*insert 0* (*set xs*) ∧ *y*′≤*y*)) *ys*)
$>_t$
⟨*proof*⟩

23.3 Time Bound Reasoning

We use some ad-hoc reasoning to also show the time-bound of our test program. A generalization of such methods, or the integration of this entry into existing reasoning frameworks with time is left to future work.

lemma *insert-time-pure*[*cond-TBOUND*]:*a* < *2^n* ⟹
§*vebt-assn n S ti*§ *TBOUND* (*vebt-inserti ti a*) (*13* + *13* ∗ *nat* ⌈*log 2* (*real n*)⌉)
⟨*proof*⟩

lemma *member-time-pure*[*cond-TBOUND*]:§*vebt-assn n S ti*§ *TBOUND* (*vebt-memberi ti a*) (*5* + *5* ∗ *nat* ⌈*log 2* (*real n*)⌉)
⟨*proof*⟩

lemma *pred-time-pure*[*cond-TBOUND*]:§*vebt-assn n S ti*§ *TBOUND* (*vebt-predi ti a*) (*7* + *7* ∗ *nat* ⌈*log 2* (*real n*)⌉)
⟨*proof*⟩

lemma *TBOUND-mfold*[*cond-TBOUND*]:
(⋀ *x*. *x* ∈ *set xs* ⟹ *x* < *2^n*) ⟹
§ *vebt-assn n S ti* § *TBOUND* (*mfold* (λ*x s*. *vebt-inserti s x*) *xs ti*) (*length xs* ∗ (*13* + *13* ∗ *nat* ⌈*log 2 n* ⌉) + *1*)
⟨*proof*⟩

lemma *TBOUND-mmap*[*cond-TBOUND*]:
defines *b-def*: *b ys n* ≡ *1* + *length ys* ∗ (*5* + *5* ∗ *nat* ⌈*log 2* (*real n*)⌉ + *9* + *7* ∗ *nat* ⌈*log 2* (*real n*)⌉)
shows § *vebt-assn n S ti* § *TBOUND*
(*mmap* (λ*x*. if_m *vebt-memberi ti x then return x*
else vebt-predi ti x ⪢= (λ*x*. *return* (*the x*))) *ys*) (*b ys n*)
⟨*proof*⟩

lemma *TBOUND-test*[*cond-TBOUND*]: ⟦∀ *x*∈*set xs*. *x*<*2^n*; *n*>*0* ⟧ ⟹
§ ↑ (*n*> *0*) § *TBOUND* (*test n xs ys*) (*10* ∗ *2^n* + (
(*length* (*0*#*xs*) ∗ (*13* + *13* ∗ *nat* ⌈*log 2 n* ⌉) + *1*) +

(1 + length ys * (5 + 5 * nat ⌈log 2 (real n)⌉ + 9 + 7 * nat ⌈log 2 (real n)⌉))))
⟨proof⟩

lemma test-hoare-with-time: ⟦∀ x∈set xs. x<2^n; n>0⟧ ⟹
<emp> (test n xs ys) <λr. ↑(r = map (λy. (GREATEST y'. y'∈insert 0 (set xs) ∧ y'≤y)) ys) * true >
T[10 * 2 ^ n +
(length (0 # xs) * (13 + 13 * nat ⌈log 2 (real n)⌉) + 1 +
(1 + length ys * (5 + 5 * nat ⌈log 2 (real n)⌉ + 9 + 7 * nat ⌈log 2 (real n)⌉)))]
⟨proof⟩

end

24 Conclusion

We have formalized van Emde Boas trees in Isabelle, proving correct a functional and an imperative version, together with space and run-time bounds. This work amends a list [4] of formally verified CLRS algorithms [3].

Closing we sketch some enhancements of van Emde Boas trees in Isabelle. An examination of the data structure points out that there is probably a *join* operation with the semantics *set-vebt* (*vebt-join s t*) = *set-vebt s* ∪ *set-vebt t*. We make the restriction of joining only valid trees with equal degree numbers. Obviously, the join of two leaves is trivial. If one tree is empty or singleton, a join is implemented by immediately returning the other tree or performing an insertion before. Otherwise, summary and subtrees are to be joined recursively and afterwards we have to determine minimum and maximum. Certainly, this last step can be complicated, because argument trees may also coincide on minima or maxima.

One may also consider the treatment of associated satellite data. Those are to be stored in ordinary subtrees, whereas the definition of summary trees does not have to be changed. We can transfer this to Isabelle by introducing another data type representing van Emde Boas trees. The adapted *naive-member* and *membermima* still refer to integer keys, but we add an auxiliary function *assoc* such that *assoc t x y* holds iff the key x is associated with the value y. A *both-member-options* is also defined and can be used for specifying a suitable validness invariant. We may show a conjecture like *both-member-options t x* ⟷ ∃ *y*. *assoc t x y*. Besides, valid trees enforce keys to be associated with at most one value. All canonical functions f are shifted to the new type and return a key-value pair (x, y) or the modified tree. Proofs for being x the desired successor etc. are obtained by reuse and adaptation of prior proofs. In addition, modified canonical functions f' may only return the associated values y. We show the proposition $\exists x.\ f\ t\ i = (x, y) \longleftrightarrow f'\ t\ i = y$. All writing operations require a reasoning regarding the proper (non-)modification of associations. The modified functions f' are to be exposed to a user later on.

Moreover, we did not consider lazy implementation. Currently, *vebt-buildup n* generates a full van Emde Boas tree of degree n. A *lazy implementation* would construct a subtree only if needed. From this just a constant amount of additional effort per recursive step will arise. Thereby, proven running time bounds of $\mathcal{O}(\log \log u)$ will be preserved. Beside this, a lazy implementation can also be obtained by exporting verified Isabelle code to Haskell, which heavily applies the lazy evaluation technique.

Obviously, a lazy implementation would drastically reduce memory usage. Each insertion allocates $\mathcal{O}(\log \log u)$ space and hence an implementation that does not store empty subtrees gives us memory consumption in $\mathcal{O}(n \cdot \log \log u)$ where n is the number of elements currently stored. Furthermore, one may replace ordinary arrays by *dynamic perfect hashing* [9] allowing treatment of elements in (amortized) constant time and linear space. Unfortunately, a linear memory consumption in $\mathcal{O}(n)$ is achieved at cost of some worst case runtime bounds [10]. By this, $\mathcal{O}(\log \log u)$ is turned to an amortized bound for *vebt-insert* and *vebt-delete*, since the complexity of those functions is indeed affected by the amortization. An implementation of this van Emde Boas tree variant requires verified dynamic perfect hashing and amortization in Isabelle to build on.

We used Imperative HOL due to its support of arrays and type reflexive references that are necessary for setting up a recursive tree data structure. For generating verified code, however, there also exist other frameworks, e.g. Isabelle LLVM [11] [12]. It supports refinement-based verification of correctness and worst-case time complexities. Additionally, verified programs can be exported to LLVM code, which itself is compiled to executable machine code. Strikingly, code of the introsort algorithm generated by this formalization stayed competitive with the GNU C++ library [12].

References

[1] P. van Emde Boas. “Preserving order in a forest in less than logarithmic time”. In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. 1975, pp. 75–84. DOI: 10.1109/SFCS.1975.26.

[2] P. E. Boas, R. Kaas, and E. Zijlstra. “Design and implementation of an efficient priority queue”. In: *Mathematical Systems Theory* 10.1 (1976), pp. 99–127. DOI: 10.1007/bf01683268.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[4] T. Nipkow, M. Eberl, and M. P. L. Haslbeck. “Verified Textbook Algorithms. A Biased Survey”. In: *ATVA 2020, Automated Technology for Verification and Analysis*. Ed. by D. V. Hung and O. Sokolsky. Vol. 12302. LNCS. Invited paper. Springer, 2020, pp. 25–53.

[5] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer, Bohua Zhan. *Functional Algorithms, Verified!* https://functional-algorithms-verified.org/. 2021.

[6] P. Lammich and R. Meis. “A Separation Logic Framework for Imperative HOL”. In: *Archive of Formal Proofs* (Nov. 2012). https://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html, Formal proof development. ISSN: 2150-914x.

[7] P. Lammich. “Refinement to Imperative HOL”. In: *Journal of Automated Reasoning* 62 (Apr. 2019). DOI: 10.1007/s10817-017-9437-1.

[8] B. Zhan and M. P. L. Haslbeck. *Verifying Asymptotic Time Complexity of Imperative Programs in Isabelle*. 2018. arXiv: 1802.01336 [cs.LO].

[9] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. “Dynamic Perfect Hashing: Upper and Lower Bounds”. In: vol. 0. Jan. 1988, pp. 524–531. DOI: 10.1109/SFCS.1988.21968.

[10] M. Straka. “Functional Data Structures and Algorithms”. https://ufal.mff.cuni.cz/~straka/theses/doctoral-functional_data_structures_and_algorithms.pdf. PhD thesis. Charles University in Prague, Faculty of Mathematics and Physics, 2013.

[11] P. Lammich. “Generating Veried LLVM from Isabelle/HOL”. English. In: *ITP 2019: Interactive Theorem Proving*. Interactive Theorem Proving, ITP 2019 ; Conference date: 08-09-2019 Through 13-09-2019. June 2019.

[12] M. P. L. Haslbeck and P. Lammich. “For a Few Dollars More”. In: *Programming Languages and Systems*. Ed. by N. Yoshida. Cham: Springer International Publishing, 2021, pp. 292–319. ISBN: 978-3-030-72019-3.