A Verified Imperative Implementation of B-Trees

Niels Mündler

Abstract

In this work, we use the interactive theorem prover Isabelle/HOL to verify an imperative implementation of the classical B-tree data structure [1]. The implementation supports set membership, insertion, deletion, iteration and range queries with efficient binary search for intra-node navigation. This is accomplished by first specifying the structure abstractly in the functional modeling language HOL and proving functional correctness. Using manual refinement, we derive an imperative implementation in Imperative/HOL. We show the validity of this refinement using the separation logic utilities from the Isabelle Refinement Framework [2]. The code can be exported to the programming languages SML, Scala and OCaml. This entry contains two developments:

- *B-Trees* This formalisation is discussed in greater detail in the corresponding Bachelor's Thesis[3].
- B^+*-Trees* This formalisation also supports range queries and is discussed in a paper published at ICTAC 2022.

Contents

1 Definition of the B-Tree **2**
1.1 Datatype definition . . . 2
1.2 Inorder and Set . . . 3
1.3 Height and Balancedness . . . 3
1.4 Order . . . 4
1.5 Auxiliary Lemmas . . . 4

2 Maximum and minimum height **7**
2.1 Definition of node/size . . . 7
2.2 Maximum number of nodes for a given height . . . 8
2.3 Maximum height for a given number of nodes . . . 8

3 Set interpretation **10**
3.1 Auxiliary functions . . . 10
3.2 The split function locale . . . 11
3.3 Membership . . . 11

3.4 Insertion 11
3.5 Deletion 13
3.6 Proofs of functional correctness 15
3.7 Set specification by inorder 24

4 Abstract split functions **24**
4.1 Linear split 24
4.2 Binary split 26

5 Definition of the B-Plus-Tree **27**
5.1 Datatype definition 27
5.2 Inorder and Set 28
5.3 Height and Balancedness 29
5.4 Order 29
5.5 Auxiliary Lemmas 30
5.6 Auxiliary functions 36
5.7 The split function locale 36

6 Abstract split functions **37**
6.1 Linear split 37

7 Set interpretation **38**
7.1 Membership 40
7.2 Insertion 41
7.3 Proofs of functional correctness 43
7.4 Deletion 49
7.5 Set specification by inorder 57

theory *BTree*
imports *Main HOL−Data-Structures.Sorted-Less HOL−Data-Structures.Cmp*
begin

hide-const (**open**) *Sorted-Less.sorted*
abbreviation *sorted-less* ≡ *Sorted-Less.sorted*

1 Definition of the B-Tree

1.1 Datatype definition

B-trees can be considered to have all data stored interleaved as child nodes and separating elements (also keys or indices). We define them to either be a Node that holds a list of pairs of children and indices or be a completely empty Leaf.

datatype *'a btree = Leaf | Node ('a btree ∗ 'a) list 'a btree*

type-synonym *'a btree-list* = (*'a btree* ∗ *'a*) *list*
type-synonym *'a btree-pair* = (*'a btree* ∗ *'a*)

abbreviation *subtrees* **where** *subtrees xs* ≡ (*map fst xs*)
abbreviation *separators* **where** *separators xs* ≡ (*map snd xs*)

1.2 Inorder and Set

The set of B-tree elements is defined automatically.

thm *btree.set*
value *set-btree* (*Node* [(*Leaf*, (*0*::*nat*)), (*Node* [(*Leaf*, *1*), (*Leaf*, *10*)] *Leaf*, *12*), (*Leaf*, *30*), (*Leaf*, *100*)] *Leaf*)

The inorder view is defined with the help of the concat function.

fun *inorder* :: *'a btree* ⇒ *'a list* **where**
inorder Leaf = [] |
inorder (*Node ts t*) = *concat* (*map* (λ (*sub*, *sep*). *inorder sub* @ [*sep*]) *ts*) @ *inorder t*

abbreviation *inorder-pair* ≡ λ(*sub*,*sep*). *inorder sub* @ [*sep*]
abbreviation *inorder-list ts* ≡ *concat* (*map inorder-pair ts*)

thm *inorder.simps*

value *inorder* (*Node* [(*Leaf*, (*0*::*nat*)), (*Node* [(*Leaf*, *1*), (*Leaf*, *10*)] *Leaf*, *12*), (*Leaf*, *30*), (*Leaf*, *100*)] *Leaf*)

1.3 Height and Balancedness

class *height* =
fixes *height* :: *'a* ⇒ *nat*

instantiation *btree* :: (*type*) *height*
begin

fun *height-btree* :: *'a btree* ⇒ *nat* **where**
height Leaf = *0* |
height (*Node ts t*) = *Suc* (*Max* (*height* ‘ (*set* (*subtrees ts*@[*t*]))))

instance ⟨*proof*⟩

end

Balancedness is defined is close accordance to the definition by Ernst

fun *bal*:: *'a btree* ⇒ *bool* **where**
bal Leaf = *True* |
bal (*Node ts t*) = (

(∀ sub ∈ set (subtrees ts). height sub = height t) ∧
(∀ sub ∈ set (subtrees ts). bal sub) ∧ bal t
)

value *height (Node [(Leaf, (0::nat)), (Node [(Leaf, 1), (Leaf, 10)] Leaf, 12), (Leaf, 30), (Leaf, 100)] Leaf)*

1.4 Order

The order of a B-tree is defined just as in the original paper by Bayer.

fun *order*:: *nat ⇒ 'a btree ⇒ bool* **where**
order k Leaf = True |
order k (Node ts t) = (
(*length ts* ≥ *k*) ∧
(*length ts* ≤ *2∗k*) ∧
(∀ *sub* ∈ *set (subtrees ts). order k sub*) ∧ *order k t*
)

The special condition for the root is called *root_order*

fun *root-order*:: *nat ⇒ 'a btree ⇒ bool* **where**
root-order k Leaf = True |
root-order k (Node ts t) = (
(*length ts* > *0*) ∧
(*length ts* ≤ *2∗k*) ∧
(∀ *s* ∈ *set (subtrees ts). order k s*) ∧ *order k t*
)

1.5 Auxiliary Lemmas

lemma *separators-split*:
set (separators (l@(a,b)#r)) = set (separators l) ∪ set (separators r) ∪ {b}
⟨*proof*⟩

lemma *subtrees-split*:
set (subtrees (l@(a,b)#r)) = set (subtrees l) ∪ set (subtrees r) ∪ {a}
⟨*proof*⟩

lemma *finite-set-ins-swap*:
assumes *finite A*
shows *max a (Max (Set.insert b A)) = max b (Max (Set.insert a A))*
⟨*proof*⟩

lemma *finite-set-in-idem*:
assumes *finite A*
shows *max a (Max (Set.insert a A)) = Max (Set.insert a A)*

⟨*proof*⟩

lemma *height-Leaf*: *height t* = *0* ⟷ *t* = *Leaf*
⟨*proof*⟩

lemma *height-btree-order*:
height (*Node* (*ls*@[*a*]) *t*) = *height* (*Node* (*a*#*ls*) *t*)
⟨*proof*⟩

lemma *height-btree-sub*:
height (*Node* ((*sub*,*x*)#*ls*) *t*) = *max* (*height* (*Node ls t*)) (*Suc* (*height sub*))
⟨*proof*⟩

lemma *height-btree-last*:
height (*Node* ((*sub*,*x*)#*ts*) *t*) = *max* (*height* (*Node ts sub*)) (*Suc* (*height t*))
⟨*proof*⟩

lemma *set-btree-inorder*: *set* (*inorder t*) = *set-btree t*
⟨*proof*⟩

lemma *child-subset*: *p* ∈ *set t* ⟹ *set-btree* (*fst p*) ⊆ *set-btree* (*Node t n*)
⟨*proof*⟩

lemma *some-child-sub*:
assumes (*sub*,*sep*) ∈ *set t*
shows *sub* ∈ *set* (*subtrees t*)
and *sep* ∈ *set* (*separators t*)
⟨*proof*⟩

lemma *bal-all-subtrees-equal*: *bal* (*Node ts t*) ⟹ (∀ *s1* ∈ *set* (*subtrees ts*). ∀ *s2* ∈ *set* (*subtrees ts*). *height s1* = *height s2*)
⟨*proof*⟩

lemma *fold-max-set*: ∀ *x* ∈ *set t*. *x* = *f* ⟹ *fold max t f* = *f*
⟨*proof*⟩

lemma *height-bal-tree*: *bal* (*Node ts t*) ⟹ *height* (*Node ts t*) = *Suc* (*height t*)
⟨*proof*⟩

lemma *bal-split-last*:
assumes *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)

shows *bal* (*Node* (*ls*@*rs*) *t*)
and *height* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) = *height* (*Node* (*ls*@*rs*) *t*)
⟨*proof*⟩

lemma *bal-split-right*:
assumes *bal* (*Node* (*ls*@*rs*) *t*)
shows *bal* (*Node* *rs* *t*)
and *height* (*Node* *rs* *t*) = *height* (*Node* (*ls*@*rs*) *t*)
⟨*proof*⟩

lemma *bal-split-left*:
assumes *bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*)
shows *bal* (*Node* *ls* *a*)
and *height* (*Node* *ls* *a*) = *height* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*)
⟨*proof*⟩

lemma *bal-substitute*: ⟦*bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*); *height* *t* = *height* *c*; *bal* *c*⟧ ⟹ *bal* (*Node* (*ls*@(*c*,*b*)#*rs*) *t*)
⟨*proof*⟩

lemma *bal-substitute-subtree*: ⟦*bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*); *height* *a* = *height* *c*; *bal* *c*⟧ ⟹ *bal* (*Node* (*ls*@(*c*,*b*)#*rs*) *t*)
⟨*proof*⟩

lemma *bal-substitute-separator*: *bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*) ⟹ *bal* (*Node* (*ls*@(*a*,*c*)#*rs*) *t*)
⟨*proof*⟩

lemma *order-impl-root-order*: ⟦*k* > *0*; *order* *k* *t*⟧ ⟹ *root-order* *k* *t*
⟨*proof*⟩

lemma *sorted-inorder-list-separators*: *sorted-less* (*inorder-list* *ts*) ⟹ *sorted-less* (*separators* *ts*)
⟨*proof*⟩

corollary *sorted-inorder-separators*: *sorted-less* (*inorder* (*Node* *ts* *t*)) ⟹ *sorted-less* (*separators* *ts*)
⟨*proof*⟩

lemma *sorted-inorder-list-subtrees*:
sorted-less (*inorder-list* *ts*) ⟹ ∀ *sub* ∈ *set* (*subtrees* *ts*). *sorted-less* (*inorder* *sub*)

⟨*proof*⟩

corollary *sorted-inorder-subtrees*: *sorted-less* (*inorder* (*Node ts t*)) $\Longrightarrow$ ∀ *sub* ∈ *set* (*subtrees ts*). *sorted-less* (*inorder sub*)
⟨*proof*⟩

lemma *sorted-inorder-list-induct-subtree*:
sorted-less (*inorder-list* (*ls*@(*sub*,*sep*)#*rs*)) $\Longrightarrow$ *sorted-less* (*inorder sub*)
⟨*proof*⟩

corollary *sorted-inorder-induct-subtree*:
sorted-less (*inorder* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)) $\Longrightarrow$ *sorted-less* (*inorder sub*)
⟨*proof*⟩

lemma *sorted-inorder-induct-last*: *sorted-less* (*inorder* (*Node ts t*)) $\Longrightarrow$ *sorted-less* (*inorder t*)
⟨*proof*⟩

end
theory *BTree-Height*
imports *BTree*
begin

2 Maximum and minimum height

Textbooks usually provide some proofs relating the maxmimum and minimum height of the BTree for a given number of nodes. We therefore introduce this counting and show the respective proofs.

2.1 Definition of node/size

thm *BTree.btree.size*

value *size* (*Node* [(*Leaf*, (*0*::*nat*)), (*Node* [(*Leaf*, *1*), (*Leaf*, *10*)] *Leaf*, *12*), (*Leaf*, *30*), (*Leaf*, *100*)] *Leaf*)

The default size function does not suit our needs as it regards the length of the list in each node. We would like to count the number of nodes in the tree only, not regarding the number of keys.

fun *nodes*::*'a btree* ⇒ *nat* **where**
nodes Leaf = *0* |
nodes (*Node ts t*) = *1* + (∑ *t*←*subtrees ts*. *nodes t*) + (*nodes t*)

value *nodes* (*Node* [(*Leaf*, (*0*::*nat*)), (*Node* [(*Leaf*, *1*), (*Leaf*, *10*)] *Leaf*, *12*), (*Leaf*, *30*), (*Leaf*, *100*)] *Leaf*)

2.2 Maximum number of nodes for a given height

lemma *sum-list-replicate*: $sum\text{-}list\ (replicate\ n\ c) = n*c$
⟨*proof*⟩

abbreviation $bound\ k\ h \equiv ((k+1)\hat{\ }h - 1)$

lemma *nodes-height-upper-bound*:
$[\![order\ k\ t;\ bal\ t]\!] \Longrightarrow nodes\ t * (2*k) \leq bound\ (2*k)\ (height\ t)$
⟨*proof*⟩

To verify our lower bound is sharp, we compare it to the height of artificially constructed full trees.

fun $full\text{-}node::nat \Rightarrow 'a \Rightarrow nat \Rightarrow 'a\ btree$ **where**
$full\text{-}node\ k\ c\ 0 = Leaf|$
$full\text{-}node\ k\ c\ (Suc\ n) = (Node\ (replicate\ (2*k)\ ((full\text{-}node\ k\ c\ n),c))\ (full\text{-}node\ k\ c\ n))$

value $let\ k = (2::nat)\ in\ map\ (\lambda x.\ nodes\ x * 2*k)\ (map\ (full\text{-}node\ k\ (1::nat))\ [0,1,2,3,4])$
value $let\ k = (2::nat)\ in\ map\ (\lambda x.\ ((2*k+(1::nat))\hat{\ }(x)-1))\ [0,1,2,3,4]$

lemma *compow-comp-id*: $c > 0 \Longrightarrow f \circ f = f \Longrightarrow (f\ \hat{}\hat{}\ c) = f$
⟨*proof*⟩

lemma *compow-id-point*: $f\ x = x \Longrightarrow (f\ \hat{}\hat{}\ c)\ x = x$
⟨*proof*⟩

lemma *height-full-node*: $height\ (full\text{-}node\ k\ a\ h) = h$
⟨*proof*⟩

lemma *bal-full-node*: $bal\ (full\text{-}node\ k\ a\ h)$
⟨*proof*⟩

lemma *order-full-node*: $order\ k\ (full\text{-}node\ k\ a\ h)$
⟨*proof*⟩

lemma *full-btrees-sharp*: $nodes\ (full\text{-}node\ k\ a\ h) * (2*k) = bound\ (2*k)\ h$
⟨*proof*⟩

lemma *upper-bound-sharp-node*:
$t = full\text{-}node\ k\ a\ h \Longrightarrow height\ t = h \wedge order\ k\ t \wedge bal\ t \wedge bound\ (2*k)\ h = nodes\ t * (2*k)$
⟨*proof*⟩

2.3 Maximum height for a given number of nodes

lemma *nodes-height-lower-bound*:
$[\![order\ k\ t;\ bal\ t]\!] \Longrightarrow bound\ k\ (height\ t) \leq nodes\ t * k$

⟨*proof*⟩

To verify our upper bound is sharp, we compare it to the height of artificially constructed minimally filled (=slim) trees.

fun *slim-node*::*nat* ⇒ *'a* ⇒ *nat* ⇒ *'a btree* **where**
slim-node k c 0 = Leaf|
slim-node k c (*Suc n*) = (*Node* (*replicate k* ((*slim-node k c n*),*c*)) (*slim-node k c n*))

value *let k* = (*2*::*nat*) *in map* (*λx. nodes x* ∗ *k*) (*map* (*slim-node k* (*1*::*nat*)) [*0*,*1*,*2*,*3*,*4*])
value *let k* = (*2*::*nat*) *in map* (*λx.* ((*k*+*1*::*nat*)^(*x*)−*1*)) [*0*,*1*,*2*,*3*,*4*]

lemma *height-slim-node*: *height* (*slim-node k a h*) = *h*
⟨*proof*⟩

lemma *bal-slim-node*: *bal* (*slim-node k a h*)
⟨*proof*⟩

lemma *order-slim-node*: *order k* (*slim-node k a h*)
⟨*proof*⟩

lemma *slim-nodes-sharp*: *nodes* (*slim-node k a h*) ∗ *k* = *bound k h*
⟨*proof*⟩

lemma *lower-bound-sharp-node*:
t = *slim-node k a h* ⟹ *height t* = *h* ∧ *order k t* ∧ *bal t* ∧ *bound k h* = *nodes t* ∗ *k*
⟨*proof*⟩

Since BTrees have special roots, we need to show the overall nodes seperately

lemma *nodes-root-height-lower-bound*:
assumes *root-order k t*
and *bal t*
shows *2*∗((*k*+*1*)^(*height t* − *1*) − *1*) + (*of-bool* (*t* ≠ *Leaf*))∗*k* ≤ *nodes t* ∗ *k*
⟨*proof*⟩

lemma *nodes-root-height-upper-bound*:
assumes *root-order k t*
and *bal t*
shows *nodes t* ∗ (*2*∗*k*) ≤ (*2*∗*k*+*1*)^(*height t*) − *1*
⟨*proof*⟩

lemma *root-order-imp-divmuleq*: *root-order k t* ⟹ (*nodes t* ∗ *k*) *div k* = *nodes t*
⟨*proof*⟩

lemma *nodes-root-height-lower-bound-simp*:
assumes *root-order k t*

and *bal t*
and $k > 0$
shows $(2*((k+1)^\frown(height\ t - 1) - 1))\ div\ k + (of\text{-}bool\ (t \neq Leaf)) \leq nodes\ t$
⟨*proof*⟩

lemma *nodes-root-height-upper-bound-simp*:
assumes *root-order k t*
and *bal t*
shows $nodes\ t \leq ((2*k+1)^\frown(height\ t) - 1)\ div\ (2*k)$
⟨*proof*⟩

definition *full-tree* = *full-node*

fun *slim-tree* **where**
slim-tree k c 0 = *Leaf* |
slim-tree k c (*Suc h*) = *Node* [(*slim-node k c h*, *c*)] (*slim-node k c h*)

lemma *lower-bound-sharp*:
$k > 0 \Longrightarrow t = slim\text{-}tree\ k\ a\ h \Longrightarrow height\ t = h \wedge root\text{-}order\ k\ t \wedge bal\ t \wedge nodes\ t * k = 2*((k+1)^\frown(height\ t - 1) - 1) + (of\text{-}bool\ (t \neq Leaf))*k$
⟨*proof*⟩

lemma *upper-bound-sharp*:
$k > 0 \Longrightarrow t = full\text{-}tree\ k\ a\ h \Longrightarrow height\ t = h \wedge root\text{-}order\ k\ t \wedge bal\ t \wedge ((2*k+1)^\frown(height\ t) - 1) = nodes\ t * (2*k)$
⟨*proof*⟩

end
theory *BTree-Set*
imports *BTree*
HOL−Data-Structures.Set-Specs
begin

3 Set interpretation

3.1 Auxiliary functions

fun *split-half*:: $('a\ btree \times 'a)\ list \Rightarrow (('a\ btree \times 'a)\ list \times ('a\ btree \times 'a)\ list)$ **where**
split-half xs = (*take* (*length xs div 2*) *xs*, *drop* (*length xs div 2*) *xs*)

lemma *drop-not-empty*: $xs \neq [] \Longrightarrow drop\ (length\ xs\ div\ 2)\ xs \neq []$
⟨*proof*⟩

lemma *split-half-not-empty*: $length\ xs \geq 1 \Longrightarrow \exists\, ls\ sub\ sep\ rs.\ split\text{-}half\ xs = (ls,(sub,sep)\#rs)$
⟨*proof*⟩

3.2 The split function locale

Here, we abstract away the inner workings of the split function for B-tree operations.

locale *split* =
 fixes *split* :: ('*a btree*×'*a*::*linorder*) *list* ⇒ '*a* ⇒ (('*a btree*×'*a*) *list* × ('*a btree*×'*a*) *list*)
 assumes *split-req*:
  ⟦*split xs p* = (*ls*,*rs*)⟧ ⟹ *xs* = *ls* @ *rs*
  ⟦*split xs p* = (*ls*@[(*sub*,*sep*)],*rs*); *sorted-less* (*separators xs*)⟧ ⟹ *sep* < *p*
  ⟦*split xs p* = (*ls*,(*sub*,*sep*)#*rs*); *sorted-less* (*separators xs*)⟧ ⟹ *p* ≤ *sep*
begin

lemmas *split-conc* = *split-req*(*1*)
lemmas *split-sorted* = *split-req*(*2*,*3*)

lemma [*termination-simp*]:(*ls*, (*sub*, *sep*) # *rs*) = *split ts y* ⟹
  size sub < *Suc* (*size-list* (λ*x*. *Suc* (*size* (*fst x*))) *ts* + *size l*)
 ⟨*proof*⟩

fun *invar-inorder* **where** *invar-inorder k t* = (*bal t* ∧ *root-order k t*)

definition *empty-btree* = *Leaf*

3.3 Membership

fun *isin*:: '*a btree* ⇒ '*a* ⇒ *bool* **where**
 isin (*Leaf*) *y* = *False* |
 isin (*Node ts t*) *y* = (
  case split ts y of (-,(*sub*,*sep*)#*rs*) ⇒ (
   if y = *sep then*
    True
   else
    isin sub y
)
 | (-,[]) ⇒ *isin t y*
)

3.4 Insertion

The insert function requires an auxiliary data structure and auxiliary invariant functions.

datatype '*b* up_i = T_i '*b btree* | Up_i '*b btree* '*b* '*b btree*

fun *order-*up_i **where**
 order-up_i *k* (T_i *sub*) = *order k sub* |

$order\text{-}up_i\ k\ (Up_i\ l\ a\ r) = (order\ k\ l \wedge order\ k\ r)$

fun $root\text{-}order\text{-}up_i$ **where**
$root\text{-}order\text{-}up_i\ k\ (T_i\ sub) = root\text{-}order\ k\ sub\ |$
$root\text{-}order\text{-}up_i\ k\ (Up_i\ l\ a\ r) = (order\ k\ l \wedge order\ k\ r)$

fun $height\text{-}up_i$ **where**
$height\text{-}up_i\ (T_i\ t) = height\ t\ |$
$height\text{-}up_i\ (Up_i\ l\ a\ r) = max\ (height\ l)\ (height\ r)$

fun $bal\text{-}up_i$ **where**
$bal\text{-}up_i\ (T_i\ t) = bal\ t\ |$
$bal\text{-}up_i\ (Up_i\ l\ a\ r) = (height\ l = height\ r \wedge bal\ l \wedge bal\ r)$

fun $inorder\text{-}up_i$ **where**
$inorder\text{-}up_i\ (T_i\ t) = inorder\ t\ |$
$inorder\text{-}up_i\ (Up_i\ l\ a\ r) = inorder\ l\ @\ [a]\ @\ inorder\ r$

The following function merges two nodes and returns separately split nodes if an overflow occurs

fun $node_i$:: $nat \Rightarrow ('a\ btree \times 'a)\ list \Rightarrow 'a\ btree \Rightarrow 'a\ up_i$ **where**
$node_i\ k\ ts\ t = ($
$if\ length\ ts \leq 2{*}k\ then\ T_i\ (Node\ ts\ t)$
$else\ ($
$\quad case\ split\text{-}half\ ts\ of\ (ls,\ (sub,sep)\#rs) \Rightarrow$
$\quad\quad Up_i\ (Node\ ls\ sub)\ sep\ (Node\ rs\ t)$
$\quad)$
$)$

lemma *nodei-ti-simp*: $node_i\ k\ ts\ t = T_i\ x \Longrightarrow x = Node\ ts\ t$
$\langle proof \rangle$

fun ins:: $nat \Rightarrow 'a \Rightarrow 'a\ btree \Rightarrow 'a\ up_i$ **where**
$ins\ k\ x\ Leaf = (Up_i\ Leaf\ x\ Leaf)\ |$
$ins\ k\ x\ (Node\ ts\ t) = ($
$case\ split\ ts\ x\ of$
$\quad (ls,(sub,sep)\#rs) \Rightarrow$
$\quad\quad (if\ sep = x\ then$
$\quad\quad\quad T_i\ (Node\ ts\ t)$
$\quad\quad else$
$\quad\quad\quad (case\ ins\ k\ x\ sub\ of$
$\quad\quad\quad\quad Up_i\ l\ a\ r \Rightarrow$
$\quad\quad\quad\quad\quad node_i\ k\ (ls\ @\ (l,a)\#(r,sep)\#rs)\ t\ |$
$\quad\quad\quad\quad T_i\ a \Rightarrow$
$\quad\quad\quad\quad\quad T_i\ (Node\ (ls\ @\ (a,sep)\ \#\ rs)\ t)))\ |$
$\quad (ls,\ []) \Rightarrow$
$\quad\quad (case\ ins\ k\ x\ t\ of$

```
      Upᵢ l a r ⇒
        nodeᵢ k (ls@[(l,a)]) r |
      Tᵢ a ⇒
        Tᵢ (Node ls a)
  )
)
```

```
fun treeᵢ::'a upᵢ ⇒ 'a btree where
  treeᵢ (Tᵢ sub) = sub |
  treeᵢ (Upᵢ l a r) = (Node [(l,a)] r)

fun insert::nat ⇒ 'a ⇒ 'a btree ⇒ 'a btree where
  insert k x t = treeᵢ (ins k x t)
```

3.5 Deletion

The following deletion method is inspired by Bayer (70) and Fielding (80). Rather than stealing only a single node from the neighbour, the neighbour is fully merged with the potentially underflowing node. If the resulting node is still larger than allowed, the merged node is split again, using the rules known from insertion splits. If the resulting node has admissable size, it is simply kept in the tree.

```
fun rebalance-middle-tree where
  rebalance-middle-tree k ls Leaf sep rs Leaf = (
  Node (ls@(Leaf,sep)#rs) Leaf
) |
  rebalance-middle-tree k ls (Node mts mt) sep rs (Node tts tt) = (
  if length mts ≥ k ∧ length tts ≥ k then
    Node (ls@(Node mts mt,sep)#rs) (Node tts tt)
  else (
    case rs of [] ⇒ (
      case nodeᵢ k (mts@(mt,sep)#tts) tt of
       Tᵢ u ⇒
        Node ls u |
       Upᵢ l a r ⇒
        Node (ls@[(l,a)]) r) |
    (Node rts rt,rsep)#rs ⇒ (
      case nodeᵢ k (mts@(mt,sep)#rts) rt of
      Tᵢ u ⇒
        Node (ls@(u,rsep)#rs) (Node tts tt) |
      Upᵢ l a r ⇒
        Node (ls@(l,a)#(r,rsep)#rs) (Node tts tt))
))
```

Deletion

All trees are merged with the right neighbour on underflow. Obviously for the last tree this would not work since it has no right neighbour. Therefore this tree, as the only exception, is merged with the left neighbour. However since we it does not make a difference, we treat the situation as if the second to last tree underflowed.

```
fun rebalance-last-tree where
  rebalance-last-tree k ts t = (
case last ts of (sub,sep) ⇒
   rebalance-middle-tree k (butlast ts) sub sep [] t
)
```

Rather than deleting the minimal key from the right subtree, we remove the maximal key of the left subtree. This is due to the fact that the last tree can easily be accessed and the left neighbour is way easier to access than the right neighbour, it resides in the same pair as the separating element to be removed.

```
fun split-max where
  split-max k (Node ts t) = (case t of Leaf ⇒ (
  let (sub,sep) = last ts in
    (Node (butlast ts) sub, sep)
)|
- ⇒
case split-max k t of (sub, sep) ⇒
  (rebalance-last-tree k ts sub, sep)
)
```

```
fun del where
  del k x Leaf = Leaf |
  del k x (Node ts t) = (
  case split ts x of
    (ls,[]) ⇒
     rebalance-last-tree k ls (del k x t)
  | (ls,(sub,sep)#rs) ⇒ (
      if sep ≠ x then
        rebalance-middle-tree k ls (del k x sub) sep rs t
      else if sub = Leaf then
        Node (ls@rs) t
      else let (sub-s, max-s) = split-max k sub in
        rebalance-middle-tree k ls sub-s max-s rs t
  )
)
```

```
fun reduce-root where
  reduce-root Leaf = Leaf |
  reduce-root (Node ts t) = (case ts of
   [] ⇒ t |
   - ⇒ (Node ts t)
)
```

fun *delete* **where** *delete k x t* = *reduce-root* (*del k x t*)

An invariant for intermediate states at deletion. In particular we allow for an underflow to 0 subtrees.

fun *almost-order* **where**
  almost-order k Leaf = *True* |
  almost-order k (*Node ts t*) = (
  (*length ts* ≤ *2∗k*) ∧
  (∀ *s* ∈ *set* (*subtrees ts*). *order k s*) ∧
   order k t
)

A recursive property of the "spine" we want to walk along for splitting off the maximum of the left subtree.

fun *nonempty-lasttreebal* **where**
  nonempty-lasttreebal Leaf = *True* |
  nonempty-lasttreebal (*Node ts t*) = (
    (∃ *ls tsub tsep*. *ts* = (*ls*@[(*tsub*,*tsep*)]) ∧ *height tsub* = *height t*) ∧
     nonempty-lasttreebal t
)

3.6 Proofs of functional correctness

lemma *split-set*:
  assumes *split ts z* = (*ls*,(*a*,*b*)#*rs*)
  shows (*a*,*b*) ∈ *set ts*
    and (*x*,*y*) ∈ *set ls* ⟹ (*x*,*y*) ∈ *set ts*
    and (*x*,*y*) ∈ *set rs* ⟹ (*x*,*y*) ∈ *set ts*
    and *set ls* ∪ *set rs* ∪ {(*a*,*b*)} = *set ts*
    and ∃ *x* ∈ *set ts*. *b* ∈ *Basic-BNFs.snds x*
  ⟨*proof*⟩

lemma *split-length*:
  split ts x = (*ls*, *rs*) ⟹ *length ls* + *length rs* = *length ts*
  ⟨*proof*⟩

Isin proof

thm *isin-simps*

lemma *sorted-ConsD*: *sorted-less* (*y* # *xs*) ⟹ *x* ≤ *y* ⟹ *x* ∉ *set xs*
  ⟨*proof*⟩

lemma *sorted-snocD*: *sorted-less* (*xs* @ [*y*]) ⟹ *y* ≤ *x* ⟹ *x* ∉ *set xs*
  ⟨*proof*⟩

lemmas *isin-simps2* = *sorted-lems sorted-ConsD sorted-snocD*

lemma *isin-sorted*: *sorted-less* (*xs*@*a*#*ys*) $\Longrightarrow$
  ($x \in$ *set* (*xs*@*a*#*ys*)) = (*if* $x < a$ *then* $x \in$ *set xs else* $x \in$ *set* (*a*#*ys*))
  ⟨*proof*⟩

lemma *isin-sorted-split*:
  assumes *sorted-less* (*inorder* (*Node ts t*))
    and *split ts x* = (*ls*, *rs*)
  shows $x \in$ *set* (*inorder* (*Node ts t*)) = ($x \in$ *set* (*inorder-list rs* @ *inorder t*))
⟨*proof*⟩

lemma *isin-sorted-split-right*:
  assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
    and *sorted-less* (*inorder* (*Node ts t*))
    and *sep* $\neq$ *x*
  shows $x \in$ *set* (*inorder-list* ((*sub*,*sep*)#*rs*) @ *inorder t*) = ($x \in$ *set* (*inorder sub*))
⟨*proof*⟩

theorem *isin-set-inorder*: *sorted-less* (*inorder t*) $\Longrightarrow$ *isin t x* = ($x \in$ *set* (*inorder t*))
⟨*proof*⟩

lemma *node*$_i$*-cases*: *length xs* $\leq$ *k* $\vee$ ($\exists$ *ls sub sep rs*. *split-half xs* = (*ls*,(*sub*,*sep*)#*rs*))
⟨*proof*⟩

lemma *root-order-tree*$_i$: *root-order-up*$_i$ (*Suc k*) *t* = *root-order* (*Suc k*) (*tree*$_i$ *t*)
  ⟨*proof*⟩

lemma *node*$_i$*-root-order*:
  assumes *length ts* > *0*
    and *length ts* $\leq$ *4*∗*k*+*1*
    and $\forall x \in$ *set* (*subtrees ts*). *order k x*
    and *order k t*
  shows *root-order-up*$_i$ *k* (*node*$_i$ *k ts t*)
⟨*proof*⟩

lemma *node*$_i$*-order-helper*:
  assumes *length ts* $\geq$ *k*
    and *length ts* $\leq$ *4*∗*k*+*1*
    and $\forall x \in$ *set* (*subtrees ts*). *order k x*

and *order k t*
shows *case* ($node_i$ *k ts t*) *of* T_i *t* ⇒ *order k t* | - ⇒ *True*
⟨*proof*⟩

lemma $node_i$*-order*:
assumes *length ts* ≥ *k*
and *length ts* ≤ *4*∗*k*+*1*
and ∀ *x* ∈ *set* (*subtrees ts*). *order k x*
and *order k t*
shows *order-up*$_i$ *k* ($node_i$ *k ts t*)

⟨*proof*⟩

lemma *ins-order*:
order k t ⟹ *order-up*$_i$ *k* (*ins k x t*)
⟨*proof*⟩

lemma *ins-root-order*:
assumes *root-order k t*
shows *root-order-up*$_i$ *k* (*ins k x t*)
⟨*proof*⟩

lemma *height-list-split*: *height-up*$_i$ (Up_i (*Node ls a*) *b* (*Node rs t*)) = *height* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*)
⟨*proof*⟩

lemma $node_i$*-height*: *height-up*$_i$ ($node_i$ *k ts t*) = *height* (*Node ts t*)
⟨*proof*⟩

lemma *bal-up*$_i$*-tree*: *bal-up*$_i$ *t* = *bal* ($tree_i$ *t*)
⟨*proof*⟩

lemma *bal-list-split*: *bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*) ⟹ *bal-up*$_i$ (Up_i (*Node ls a*) *b* (*Node rs t*))
⟨*proof*⟩

lemma $node_i$*-bal*:
assumes *bal* (*Node ts t*)
shows *bal-up*$_i$ ($node_i$ *k ts t*)
⟨*proof*⟩

lemma *height-up*$_i$*-merge*: *height-up*$_i$ (*Up*$_i$ *l a r*) = *height t* ⟹ *height* (*Node* (*ls*@(*t*,*x*)#*rs*) *tt*) = *height* (*Node* (*ls*@(*l*,*a*)#(*r*,*x*)#*rs*) *tt*)
⟨*proof*⟩

lemma *ins-height*: *height-up*$_i$ (*ins k x t*) = *height t*
⟨*proof*⟩

lemma *ins-bal*: *bal t* ⟹ *bal-up*$_i$ (*ins k x t*)
⟨*proof*⟩

lemma *node*$_i$*-inorder*: *inorder-up*$_i$ (*node*$_i$ *k ts t*) = *inorder* (*Node ts t*)
⟨*proof*⟩

corollary *node*$_i$*-inorder-simps*:
node$_i$ *k ts t* = *T*$_i$ *t*′ ⟹ *inorder t*′ = *inorder* (*Node ts t*)
node$_i$ *k ts t* = *Up*$_i$ *l a r* ⟹ *inorder l* @ *a* # *inorder r* = *inorder* (*Node ts t*)
⟨*proof*⟩

lemma *ins-sorted-inorder*: *sorted-less* (*inorder t*) ⟹ (*inorder-up*$_i$ (*ins k* (*x*::(′*a*::*linorder*)) *t*)) = *ins-list x* (*inorder t*)
⟨*proof*⟩

lemma *ins-list-split*:
assumes *split ts x* = (*ls*, *rs*)
and *sorted-less* (*inorder* (*Node ts t*))
shows *ins-list x* (*inorder* (*Node ts t*)) = *inorder-list ls* @ *ins-list x* (*inorder-list rs* @ *inorder t*)
⟨*proof*⟩

lemma *ins-list-split-right-general*:
assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
and *sorted-less* (*inorder-list ts*)
and *sep* ≠ *x*
shows *ins-list x* (*inorder-list* ((*sub*,*sep*)#*rs*) @ *zs*) = *ins-list x* (*inorder sub*) @ *sep* # *inorder-list rs* @ *zs*
⟨*proof*⟩

corollary *ins-list-split-right*:
assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)

and *sorted-less* (*inorder* (*Node ts t*))
and *sep* ≠ *x*
shows *ins-list x* (*inorder-list* ((*sub*,*sep*)#*rs*) @ *inorder t*) = *ins-list x* (*inorder sub*) @ *sep* # *inorder-list rs* @ *inorder t*
⟨*proof*⟩

lemma *ins-list-idem-eq-isin*: *sorted-less xs* ⟹ *x* ∈ *set xs* ⟷ (*ins-list x xs* = *xs*)
⟨*proof*⟩

lemma *ins-list-contains-idem*: ⟦*sorted-less xs*; *x* ∈ *set xs*⟧ ⟹ (*ins-list x xs* = *xs*)
⟨*proof*⟩

declare $node_i$*.simps* [*simp del*]
declare $node_i$*-inorder* [*simp add*]

lemma *ins-inorder*: *sorted-less* (*inorder t*) ⟹ (*inorder-up*$_i$ (*ins k x t*)) = *ins-list x* (*inorder t*)
⟨*proof*⟩

declare $node_i$*.simps* [*simp add*]
declare $node_i$*-inorder* [*simp del*]

thm *ins.induct*
thm *btree.induct*

lemma $tree_i$*-bal*: *bal-up*$_i$ *u* ⟹ *bal* ($tree_i$ *u*)
⟨*proof*⟩

lemma $tree_i$*-order*: ⟦*k* > *0*; *root-order-up*$_i$ *k u*⟧ ⟹ *root-order k* ($tree_i$ *u*)
⟨*proof*⟩

lemma $tree_i$*-inorder*: *inorder-up*$_i$ *u* = *inorder* ($tree_i$ *u*)
⟨*proof*⟩

lemma *insert-bal*: *bal t* ⟹ *bal* (*insert k x t*)
⟨*proof*⟩

lemma *insert-order*: ⟦*k* > *0*; *root-order k t*⟧ ⟹ *root-order k* (*insert k x t*)
⟨*proof*⟩

lemma *insert-inorder*: *sorted-less* (*inorder t*) ⟹ *inorder* (*insert k x t*) = *ins-list x* (*inorder t*)

⟨*proof*⟩

Deletion proofs

thm *list.simps*

lemma *rebalance-middle-tree-height*:
assumes *height t* = *height sub*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
shows *height* (*rebalance-middle-tree k ls sub sep rs t*) = *height* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
⟨*proof*⟩

lemma *rebalance-last-tree-height*:
assumes *height t* = *height sub*
and *ts* = *list*@[(*sub*,*sep*)]
shows *height* (*rebalance-last-tree k ts t*) = *height* (*Node ts t*)
⟨*proof*⟩

lemma *split-max-height*:
assumes *split-max k t* = (*sub*,*sep*)
and *nonempty-lasttreebal t*
and *t* ≠ *Leaf*
shows *height sub* = *height t*
⟨*proof*⟩

lemma *order-bal-nonempty-lasttreebal*: ⟦*k* > *0*; *root-order k t*; *bal t*⟧ ⟹ *nonempty-lasttreebal t*
⟨*proof*⟩

lemma *bal-sub-height*: *bal* (*Node* (*ls*@*a*#*rs*) *t*) ⟹ (*case rs of* [] ⇒ *True* | (*sub*,*sep*)#- ⇒ *height sub* = *height t*)
⟨*proof*⟩

lemma *del-height*: ⟦*k* > *0*; *root-order k t*; *bal t*⟧ ⟹ *height* (*del k x t*) = *height t*
⟨*proof*⟩

lemma *rebalance-middle-tree-inorder*:
assumes *height t* = *height sub*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
shows *inorder* (*rebalance-middle-tree k ls sub sep rs t*) = *inorder* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
⟨*proof*⟩

lemma *rebalance-last-tree-inorder*:
assumes *height t* = *height sub*
and *ts* = *list*@[(*sub*,*sep*)]
shows *inorder* (*rebalance-last-tree k ts t*) = *inorder* (*Node ts t*)
⟨*proof*⟩

lemma *butlast-inorder-app-id*: *xs* = *xs'* @ [(*sub*,*sep*)] ⟹ *inorder-list xs'* @ *inorder sub* @ [*sep*] = *inorder-list xs*
⟨*proof*⟩

lemma *split-max-inorder*:
assumes *nonempty-lasttreebal t*
and *t* ≠ *Leaf*
shows *inorder-pair* (*split-max k t*) = *inorder t*
⟨*proof*⟩

lemma *height-bal-subtrees-merge*: ⟦*height* (*Node as a*) = *height* (*Node bs b*); *bal* (*Node as a*); *bal* (*Node bs b*)⟧
⟹ ∀ *x* ∈ *set* (*subtrees as*) ∪ {*a*}. *height x* = *height b*
⟨*proof*⟩

lemma *bal-list-merge*:
assumes $bal\text{-}up_i$ (Up_i (*Node as a*) *x* (*Node bs b*))
shows *bal* (*Node* (*as*@(*a*,*x*)#*bs*) *b*)
⟨*proof*⟩

lemma $node_i$-*bal*-up_i:
assumes $bal\text{-}up_i$ ($node_i$ *k ts t*)
shows *bal* (*Node ts t*)
⟨*proof*⟩

lemma $node_i$-*bal-simp*: $bal\text{-}up_i$ ($node_i$ *k ts t*) = *bal* (*Node ts t*)
⟨*proof*⟩

lemma *rebalance-middle-tree-bal*: *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) ⟹ *bal* (*rebalance-middle-tree k ls sub sep rs t*)
⟨*proof*⟩

lemma *rebalance-last-tree-bal*: ⟦*bal* (*Node ts t*); *ts* ≠ []⟧ ⟹ *bal* (*rebalance-last-tree k ts t*)
⟨*proof*⟩

lemma *split-max-bal*:
assumes *bal t*
and *t* ≠ *Leaf*
and *nonempty-lasttreebal t*

shows *bal* (*fst* (*split-max k t*))
⟨*proof*⟩

lemma *del-bal*:
assumes *k* > *0*
and *root-order k t*
and *bal t*
shows *bal* (*del k x t*)
⟨*proof*⟩

lemma *rebalance-middle-tree-order*:
assumes *almost-order k sub*
and ∀ *s* ∈ *set* (*subtrees* (*ls*@*rs*)). *order k s order k t*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
and *length* (*ls*@(*sub*,*sep*)#*rs*) ≤ *2*∗*k*
and *height sub* = *height t*
shows *almost-order k* (*rebalance-middle-tree k ls sub sep rs t*)
⟨*proof*⟩

lemma *rebalance-middle-tree-last-order*:
assumes *almost-order k t*
and ∀ *s* ∈ *set* (*subtrees* (*ls*@(*sub*,*sep*)#*rs*)). *order k s*
and *rs* = []
and *length* (*ls*@(*sub*,*sep*)#*rs*) ≤ *2*∗*k*
and *height sub* = *height t*
shows *almost-order k* (*rebalance-middle-tree k ls sub sep rs t*)
⟨*proof*⟩

lemma *rebalance-last-tree-order*:
assumes *ts* = *ls*@[(*sub*,*sep*)]
and ∀ *s* ∈ *set* (*subtrees* (*ts*)). *order k s almost-order k t*
and *length ts* ≤ *2*∗*k*
and *height sub* = *height t*
shows *almost-order k* (*rebalance-last-tree k ts t*)
⟨*proof*⟩

lemma *split-max-order*:
assumes *order k t*
and *t* ≠ *Leaf*
and *nonempty-lasttreebal t*
shows *almost-order k* (*fst* (*split-max k t*))
⟨*proof*⟩

lemma *del-order*:
assumes *k* > *0*
and *root-order k t*

and *bal t*
shows *almost-order k* (*del k x t*)
⟨*proof*⟩

thm *del-list-sorted*

lemma *del-list-split*:
assumes *split ts x* = (*ls*, *rs*)
and *sorted-less* (*inorder* (*Node ts t*))
shows *del-list x* (*inorder* (*Node ts t*)) = *inorder-list ls* @ *del-list x* (*inorder-list rs* @ *inorder t*)
⟨*proof*⟩

lemma *del-list-split-right*:
assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
and *sorted-less* (*inorder* (*Node ts t*))
and *sep* ≠ *x*
shows *del-list x* (*inorder-list* ((*sub*,*sep*)#*rs*) @ *inorder t*) = *del-list x* (*inorder sub*) @ *sep* # *inorder-list rs* @ *inorder t*
⟨*proof*⟩

thm *del-list-idem*

lemma *del-inorder*:
assumes *k* > *0*
and *root-order k t*
and *bal t*
and *sorted-less* (*inorder t*)
shows *inorder* (*del k x t*) = *del-list x* (*inorder t*)
⟨*proof*⟩

lemma *reduce-root-order*: ⟦*k* > *0*; *almost-order k t*⟧ ⟹ *root-order k* (*reduce-root t*)
⟨*proof*⟩

lemma *reduce-root-bal*: *bal* (*reduce-root t*) = *bal t*
⟨*proof*⟩

lemma *reduce-root-inorder*: *inorder* (*reduce-root t*) = *inorder t*
⟨*proof*⟩

lemma *delete-order*: ⟦*k* > *0*; *bal t*; *root-order k t*⟧ ⟹ *root-order k* (*delete k x t*)
⟨*proof*⟩

lemma *delete-bal*: ⟦$k > 0$; *bal t*; *root-order k t*⟧ ⟹ *bal* (*delete k x t*)
⟨*proof*⟩

lemma *delete-inorder*: ⟦$k > 0$; *bal t*; *root-order k t*; *sorted-less* (*inorder t*)⟧ ⟹ *inorder* (*delete k x t*) = *del-list x* (*inorder t*)
⟨*proof*⟩

3.7 Set specification by inorder

interpretation *S-ordered*: *Set-by-Ordered* **where**
empty = *empty-btree* **and**
insert = *insert* (*Suc k*) **and**
delete = *delete* (*Suc k*) **and**
isin = *isin* **and**
inorder = *inorder* **and**
inv = *invar-inorder* (*Suc k*)
⟨*proof*⟩

declare $node_i$.*simps*[*simp del*]

end

end
theory *BTree-Split*
imports *BTree-Set*
begin

4 Abstract split functions

4.1 Linear split

Finally we show that the split axioms are feasible by providing an example split function

fun *linear-split-help*:: (-×'*a*::*linorder*) *list* ⇒ - ⇒ (-×-) *list* ⇒ ((-×-) *list* × (-×-) *list*) **where**
linear-split-help [] *x prev* = (*prev*, []) |
linear-split-help ((*sub*, *sep*)#*xs*) *x prev* = (*if sep* < *x then linear-split-help xs x* (*prev* @ [(*sub*, *sep*)]) *else* (*prev*, (*sub*,*sep*)#*xs*))

fun *linear-split*:: (-×'*a*::*linorder*) *list* ⇒ - ⇒ ((-×-) *list* × (-×-) *list*) **where**
linear-split xs x = *linear-split-help xs x* []

Linear split is similar to well known functions, therefore a quick proof can be done.

lemma *linear-split-alt*: *linear-split xs x* = (*takeWhile* (λ(-,*s*). *s*<*x*) *xs*, *dropWhile* (λ(-,*s*). *s*<*x*) *xs*)
⟨*proof*⟩

global-interpretation *btree-linear-search*: *split linear-split*

defines *btree-ls-isin* = *btree-linear-search.isin*
and *btree-ls-ins* = *btree-linear-search.ins*
and *btree-ls-insert* = *btree-linear-search.insert*
and *btree-ls-del* = *btree-linear-search.del*
and *btree-ls-delete* = *btree-linear-search.delete*
⟨*proof*⟩

Some examples follow to show that the implementation works and the above lemmas make sense. The examples are visualized in the thesis.

abbreviation $btree_i$ ≡ *btree-ls-insert*
abbreviation $btree_d$ ≡ *btree-ls-delete*

value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
root-order k x
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
bal x
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
sorted-less (*inorder x*)
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
x
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
$btree_i$ *k 9 x*
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
$btree_i$ *k 1* ($btree_i$ *k 9 x*)
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
$btree_d$ *k 10* ($btree_i$ *k 1* ($btree_i$ *k 9 x*))
value *let k*=*2*::*nat*; *x*::*nat btree* = (*Node* [(*Node* [(*Leaf*, *3*),(*Leaf*, *5*),(*Leaf*, *6*)] *Leaf*, *10*)] (*Node* [(*Leaf*, *14*), (*Leaf*, *20*)] *Leaf*)) *in*
$btree_d$ *k 3* ($btree_d$ *k 10* ($btree_i$ *k 1* ($btree_i$ *k 9 x*)))

For completeness, we also proved an explicit proof of the locale requirements.

lemma *some-child-sm*: *linear-split-help t y xs* = (*ls*,(*sub*,*sep*)#*rs*) ⟹ *y* ≤ *sep*
⟨*proof*⟩

lemma *linear-split-append*: *linear-split-help xs p ys* = (*ls*,*rs*) $\Longrightarrow$ *ls*@*rs* = *ys*@*xs*
⟨*proof*⟩

lemma *linear-split-sm*: ⟦*linear-split-help xs p ys* = (*ls*,*rs*); *sorted-less* (*separators* (*ys*@*xs*)); ∀ *sep* ∈ *set* (*separators ys*). *p* > *sep*⟧ $\Longrightarrow$ ∀ *sep* ∈ *set* (*separators ls*). *p* > *sep*
⟨*proof*⟩

value *linear-split* [((*Leaf*::*nat btree*), *2*)] (*1*::*nat*)

lemma *linear-split-gr*:
⟦*linear-split-help xs p ys* = (*ls*,*rs*); *sorted-less* (*separators* (*ys*@*xs*)); ∀ (*sub*,*sep*) ∈ *set ys*. *p* > *sep*⟧ $\Longrightarrow$
(*case rs of* [] ⇒ *True* | (-,*sep*)#- ⇒ *p* ≤ *sep*)
⟨*proof*⟩

lemma *linear-split-req*:
assumes *linear-split xs p* = (*ls*,(*sub*,*sep*)#*rs*)
and *sorted-less* (*separators xs*)
shows *p* ≤ *sep*
⟨*proof*⟩

lemma *linear-split-req2*:
assumes *linear-split xs p* = (*ls*@[(*sub*,*sep*)],*rs*)
and *sorted-less* (*separators xs*)
shows *sep* < *p*
⟨*proof*⟩

interpretation *split linear-split*
⟨*proof*⟩

4.2 Binary split

It is possible to define a binary split predicate. However, even proving that it terminates is uncomfortable.

```
function (sequential) binary-split-help:: (-×'a::linorder) list ⇒ (-×'a) list ⇒ (-×'a)
list ⇒ 'a ⇒ ((-×-) list × (-×-) list) where
  binary-split-help ls [] rs x = (ls,rs) |
  binary-split-help ls as rs x = (let (mls, mrs) = split-half as in (
  case mrs of (sub,sep)#mrrs ⇒ (
    if x < sep then binary-split-help ls mls (mrs@rs) x
    else if x > sep then binary-split-help (ls@mls@[(sub,sep)]) mrrs rs x
    else (ls@mls, mrs@rs)
    )
  )
)
```

⟨*proof*⟩
termination
⟨*proof*⟩

fun *binary-split* **where**
binary-split as x = *binary-split-help* [] *as* [] *x*

We can show that it will return sublists that concatenate to the original list again but will not show that it fulfils sortedness properties.

lemma *binary-split-help as bs cs x* = (*ls*,*rs*) ⟹ (*as*@*bs*@*cs*) = (*ls*@*rs*)
⟨*proof*⟩

lemma ⟦*sorted-less* (*separators* (*as*@*bs*@*cs*)); *binary-split-help as bs cs x* = (*ls*,*rs*); ∀ *y* ∈ *set* (*separators as*). *y* < *x*⟧
⟹ ∀ *y* ∈ *set* (*separators ls*). *y* < *x*
⟨*proof*⟩

end
theory *BPlusTree*
imports *Main HOL−Data-Structures.Sorted-Less HOL−Data-Structures.Cmp HOL−Library.Multiset*
begin

hide-const (**open**) *Sorted-Less.sorted*
abbreviation *sorted-less* ≡ *Sorted-Less.sorted*

5 Definition of the B-Plus-Tree

5.1 Datatype definition

B-Plus-Trees are basically B-Trees, that don't have empty Leafs but Leafs that contain the relevant data.

datatype *'a bplustree* = *Leaf* (*vals*: *'a list*) | *Node* (*keyvals*: (*'a bplustree* ∗ *'a*) *list*) (*lasttree*: *'a bplustree*)

type-synonym *'a bplustree-list* = (*'a bplustree* ∗ *'a*) *list*
type-synonym *'a bplustree-pair* = (*'a bplustree* ∗ *'a*)

abbreviation *subtrees* **where** *subtrees xs* ≡ (*map fst xs*)
abbreviation *separators* **where** *separators xs* ≡ (*map snd xs*)

5.2 Inorder and Set

The set of B-Plus-tree needs to be manually defined, regarding only the leaves. This overrides the default instantiation.

fun *set-nodes* :: *'a bplustree* ⇒ *'a set* **where**
set-nodes (*Leaf ks*) = {} |
set-nodes (*Node ts t*) = ⋃(*set* (*map set-nodes* (*subtrees ts*))) ∪ (*set* (*separators ts*)) ∪ *set-nodes t*

fun *set-leaves* :: *'a bplustree* ⇒ *'a set* **where**
set-leaves (*Leaf ks*) = *set ks* |
set-leaves (*Node ts t*) = ⋃(*set* (*map set-leaves* (*subtrees ts*))) ∪ *set-leaves t*

The inorder is a view of only internal seperators

fun *inorder* :: *'a bplustree* ⇒ *'a list* **where**
inorder (*Leaf ks*) = [] |
inorder (*Node ts t*) = *concat* (*map* (λ (*sub*, *sep*). *inorder sub* @ [*sep*]) *ts*) @ *inorder t*

abbreviation *inorder-list ts* ≡ *concat* (*map* (λ (*sub*, *sep*). *inorder sub* @ [*sep*]) *ts*)

The leaves view considers only its leafs.

fun *leaves* :: *'a bplustree* ⇒ *'a list* **where**
leaves (*Leaf ks*) = *ks* |
leaves (*Node ts t*) = *concat* (*map leaves* (*subtrees ts*)) @ *leaves t*

abbreviation *leaves-list ts* ≡ *concat* (*map leaves* (*subtrees ts*))

fun *leaf-nodes* **where**
leaf-nodes (*Leaf xs*) = [*Leaf xs*] |
leaf-nodes (*Node ts t*) = *concat* (*map leaf-nodes* (*subtrees ts*)) @ *leaf-nodes t*

abbreviation *leaf-nodes-list ts* ≡ *concat* (*map leaf-nodes* (*subtrees ts*))

And the elems view contains all elements of the tree

fun *elems* :: *'a bplustree* ⇒ *'a list* **where**
elems (*Leaf ks*) = *ks* |
elems (*Node ts t*) = *concat* (*map* (λ (*sub*, *sep*). *elems sub* @ [*sep*]) *ts*) @ *elems t*

abbreviation *elems-list ts* ≡ *concat* (*map* (λ (*sub*, *sep*). *elems sub* @ [*sep*]) *ts*)

thm *leaves.simps*
thm *inorder.simps*
thm *elems.simps*

value *leaves* (*Node* [(*Leaf* [], (*0::nat*)), (*Node* [(*Leaf* [], *1*), (*Leaf* [], *10*)] (*Leaf* []), *12*), ((*Leaf* []), *30*), ((*Leaf* []), *100*)] (*Leaf* []))

5.3 Height and Balancedness

```
class height =
  fixes height :: 'a ⇒ nat

instantiation bplustree :: (type) height
begin

fun height-bplustree :: 'a bplustree ⇒ nat where
  height (Leaf ks) = 0 |
  height (Node ts t) = Suc (Max (height ‘ (set (subtrees ts@[t]))))

instance ⟨proof⟩

end
```

Balancedness is defined is close accordance to the definition by Ernst

```
fun bal:: 'a bplustree ⇒ bool where
  bal (Leaf ks) = True |
  bal (Node ts t) = (
    (∀ sub ∈ set (subtrees ts). height sub = height t) ∧
    (∀ sub ∈ set (subtrees ts). bal sub) ∧ bal t
  )


value height (Node [(Leaf [], (0::nat)), (Node [(Leaf [], 1), (Leaf [], 10)] (Leaf []),
12), ((Leaf []), 30), ((Leaf []), 100)] (Leaf []))
value bal (Node [(Leaf [], (0::nat)), (Node [(Leaf [], 1), (Leaf [], 10)] (Leaf []),
12), ((Leaf []), 30), ((Leaf []), 100)] (Leaf []))
```

5.4 Order

The order of a B-tree is defined just as in the original paper by Bayer.

```
fun order:: nat ⇒ 'a bplustree ⇒ bool where
  order k (Leaf ks) = ((length ks ≥ k)  ∧ (length ks ≤ 2*k)) |
  order k (Node ts t) = (
  (length ts ≥ k)  ∧
  (length ts ≤ 2*k) ∧
  (∀ sub ∈ set (subtrees ts). order k sub) ∧ order k t
)
```

The special condition for the root is called *root_order*

```
fun root-order:: nat ⇒ 'a bplustree ⇒ bool where
  root-order k (Leaf ks) = (length ks ≤ 2*k) |
  root-order k (Node ts t) = (
  (length ts > 0) ∧
  (length ts ≤ 2*k) ∧
  (∀ s ∈ set (subtrees ts). order k s) ∧ order k t
)
```

5.5 Auxiliary Lemmas

lemma *separators-split*:
set (*separators* (*l*@(*a*,*b*)#*r*)) = *set* (*separators* *l*) ∪ *set* (*separators* *r*) ∪ {*b*}
⟨*proof*⟩

lemma *subtrees-split*:
set (*subtrees* (*l*@(*a*,*b*)#*r*)) = *set* (*subtrees* *l*) ∪ *set* (*subtrees* *r*) ∪ {*a*}
⟨*proof*⟩

lemma *finite-set-ins-swap*:
assumes *finite A*
shows *max a* (*Max* (*Set.insert b A*)) = *max b* (*Max* (*Set.insert a A*))
⟨*proof*⟩

lemma *finite-set-in-idem*:
assumes *finite A*
shows *max a* (*Max* (*Set.insert a A*)) = *Max* (*Set.insert a A*)
⟨*proof*⟩

lemma *height-Leaf*: *height t* = *0* ⟷ (∃ *ks*. *t* = (*Leaf ks*))
⟨*proof*⟩

lemma *height-bplustree-order*:
height (*Node* (*ls*@[*a*]) *t*) = *height* (*Node* (*a*#*ls*) *t*)
⟨*proof*⟩

lemma *height-bplustree-sub*:
height (*Node* ((*sub*,*x*)#*ls*) *t*) = *max* (*height* (*Node ls t*)) (*Suc* (*height sub*))
⟨*proof*⟩

lemma *height-bplustree-last*:
height (*Node* ((*sub*,*x*)#*ts*) *t*) = *max* (*height* (*Node ts sub*)) (*Suc* (*height t*))
⟨*proof*⟩

lemma *set-leaves-leaves*: *set* (*leaves t*) = *set-leaves t*
⟨*proof*⟩

lemma *set-nodes-nodes*: *set* (*inorder t*) = *set-nodes t*
⟨*proof*⟩

lemma *child-subset-leaves*: *p* ∈ *set t* ⟹ *set-leaves* (*fst p*) ⊆ *set-leaves* (*Node t n*)
⟨*proof*⟩

lemma *child-subset*: *p* ∈ *set t* ⟹ *set-nodes* (*fst p*) ⊆ *set-nodes* (*Node t n*)

⟨*proof*⟩

lemma *some-child-sub*:
assumes (*sub*,*sep*) ∈ *set t*
shows *sub* ∈ *set* (*subtrees t*)
and *sep* ∈ *set* (*separators t*)
⟨*proof*⟩

lemma *bal-all-subtrees-equal*: *bal* (*Node ts t*) ⟹ (∀ *s1* ∈ *set* (*subtrees ts*). ∀ *s2* ∈ *set* (*subtrees ts*). *height s1* = *height s2*)
⟨*proof*⟩

lemma *fold-max-set*: ∀ *x* ∈ *set t*. *x* = *f* ⟹ *fold max t f* = *f*
⟨*proof*⟩

lemma *height-bal-tree*: *bal* (*Node ts t*) ⟹ *height* (*Node ts t*) = *Suc* (*height t*)
⟨*proof*⟩

lemma *bal-split-last*:
assumes *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
shows *bal* (*Node* (*ls*@*rs*) *t*)
and *height* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) = *height* (*Node* (*ls*@*rs*) *t*)
⟨*proof*⟩

lemma *bal-split-right*:
assumes *bal* (*Node* (*ls*@*rs*) *t*)
shows *bal* (*Node rs t*)
and *height* (*Node rs t*) = *height* (*Node* (*ls*@*rs*) *t*)
⟨*proof*⟩

lemma *bal-split-left*:
assumes *bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*)
shows *bal* (*Node ls a*)
and *height* (*Node ls a*) = *height* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*)
⟨*proof*⟩

lemma *bal-substitute*: ⟦*bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*); *height t* = *height c*; *bal c*⟧ ⟹ *bal* (*Node* (*ls*@(*c*,*b*)#*rs*) *t*)
⟨*proof*⟩

lemma *bal-substitute-subtree*: ⟦*bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*); *height a* = *height c*; *bal*

$c]\!] \Longrightarrow$ *bal* (*Node* (*ls*@(*c*,*b*)#*rs*) *t*)
⟨*proof*⟩

lemma *bal-substitute-separator*: *bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*) ⟹ *bal* (*Node* (*ls*@(*a*,*c*)#*rs*) *t*)
⟨*proof*⟩

lemma *order-impl-root-order*: ⟦*k* > *0*; *order k t*⟧ ⟹ *root-order k t*
⟨*proof*⟩

lemma *sorted-inorder-list-separators*: *sorted-less* (*inorder-list ts*) ⟹ *sorted-less* (*separators ts*)
⟨*proof*⟩

corollary *sorted-inorder-separators*: *sorted-less* (*inorder* (*Node ts t*)) ⟹ *sorted-less* (*separators ts*)
⟨*proof*⟩

lemma *sorted-inorder-list-subtrees*:
sorted-less (*inorder-list ts*) ⟹ ∀ *sub* ∈ *set* (*subtrees ts*). *sorted-less* (*inorder sub*)
⟨*proof*⟩

corollary *sorted-inorder-subtrees*: *sorted-less* (*inorder* (*Node ts t*)) ⟹ ∀ *sub* ∈ *set* (*subtrees ts*). *sorted-less* (*inorder sub*)
⟨*proof*⟩

lemma *sorted-inorder-list-induct-subtree*:
sorted-less (*inorder-list* (*ls*@(*sub*,*sep*)#*rs*)) ⟹ *sorted-less* (*inorder sub*)
⟨*proof*⟩

corollary *sorted-inorder-induct-subtree*:
sorted-less (*inorder* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)) ⟹ *sorted-less* (*inorder sub*)
⟨*proof*⟩

lemma *sorted-inorder-induct-last*: *sorted-less* (*inorder* (*Node ts t*)) ⟹ *sorted-less* (*inorder t*)
⟨*proof*⟩

lemma *sorted-leaves-list-subtrees*:
sorted-less (*leaves-list ts*) ⟹ ∀ *sub* ∈ *set* (*subtrees ts*). *sorted-less* (*leaves sub*)
⟨*proof*⟩

corollary *sorted-leaves-subtrees*: *sorted-less* (*leaves* (*Node ts t*)) $\Longrightarrow$ $\forall$ *sub* $\in$ *set* (*subtrees ts*). *sorted-less* (*leaves sub*)
⟨*proof*⟩

lemma *sorted-leaves-list-induct-subtree*:
sorted-less (*leaves-list* (*ls*@(*sub*,*sep*)#*rs*)) $\Longrightarrow$ *sorted-less* (*leaves sub*)
⟨*proof*⟩

corollary *sorted-leaves-induct-subtree*:
sorted-less (*leaves* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)) $\Longrightarrow$ *sorted-less* (*leaves sub*)
⟨*proof*⟩

lemma *sorted-leaves-induct-last*: *sorted-less* (*leaves* (*Node ts t*)) $\Longrightarrow$ *sorted-less* (*leaves t*)
⟨*proof*⟩

Additional lemmas on the sortedness of the whole tree, which is correct alignment of navigation structure and leave data

fun *inbetween* **where**
inbetween f l Nil t u = *f l t u* |
inbetween f l ((*sub*,*sep*)#*xs*) *t u* = (*f l sub sep* $\wedge$ *inbetween f sep xs t u*)

thm *fold-cong*

lemma *cong-inbetween*[*fundef-cong*]:
⟦*a* = *b*; *xs* = *ys*; $\bigwedge l'$ u' *sub sep*. (*sub*,*sep*) $\in$ *set ys* $\Longrightarrow$ *f* l' *sub* u' = *g* l' *sub* u'; $\bigwedge l'$ u'. *f* l' *a* u' = *g* l' *b* u'⟧
$\Longrightarrow$ *inbetween f l xs a u* = *inbetween g l ys b u*
⟨*proof*⟩

fun *aligned* :: *'a* ::*linorder* $\Rightarrow$ - **where**
aligned l (*Leaf ks*) *u* = ($l < u \wedge (\forall x \in$ *set ks*. $l < x \wedge x \leq u$)) |
aligned l (*Node ts t*) *u* = (*inbetween aligned l ts t u*)

lemma *sorted-less-merge*: *sorted-less* (*as*@[*a*]) $\Longrightarrow$ *sorted-less* (*a*#*bs*) $\Longrightarrow$ *sorted-less* (*as*@*a*#*bs*)
⟨*proof*⟩

thm *aligned.simps*

lemma *leaves-cases*: *x* $\in$ *set* (*leaves* (*Node ts t*)) $\Longrightarrow$ ($\exists$ (*sub*,*sep*) $\in$ *set ts*. *x* $\in$ *set* (*leaves sub*)) $\vee$ *x* $\in$ *set* (*leaves t*)
⟨*proof*⟩

lemma *align-sub*: *aligned l* (*Node ts t*) *u* $\Longrightarrow$ (*sub*,*sep*) $\in$ *set ts* $\Longrightarrow$ $\exists\, l'$ $\in$ *set* (*separators ts*) $\cup$ {*l*}. *aligned* l' *sub sep*
⟨*proof*⟩

lemma *align-last*: *aligned l* (*Node* (*ts*@[(*sub*,*sep*)]) *t*) *u* $\Longrightarrow$ *aligned sep t u*
⟨*proof*⟩

lemma *align-last'*: *aligned l* (*Node ts t*) *u* $\Longrightarrow$ $\exists$ *l'* $\in$ *set* (*separators ts*) $\cup$ {*l*}. *aligned l' t u*
⟨*proof*⟩

lemma *aligned-sorted-inorder*: *aligned l t u* $\Longrightarrow$ *sorted-less* (*l*#(*inorder t*)@[*u*])
⟨*proof*⟩

lemma *separators-in-inorder-list*: *set* (*separators ts*) $\subseteq$ *set* (*inorder-list ts*)
⟨*proof*⟩

lemma *separators-in-inorder*: *set* (*separators ts*) $\subseteq$ *set* (*inorder* (*Node ts t*))
⟨*proof*⟩

lemma *aligned-sorted-separators*: *aligned l* (*Node ts t*) *u* $\Longrightarrow$ *sorted-less* (*l*#(*separators ts*)@[*u*])
⟨*proof*⟩

lemma *aligned-leaves-inbetween*: *aligned l t u* $\Longrightarrow$ $\forall$ *x* $\in$ *set* (*leaves t*). $l < x \wedge x \leq u$
⟨*proof*⟩

lemma *aligned-leaves-list-inbetween*: *aligned l* (*Node ts t*) *u* $\Longrightarrow$ $\forall$ *x* $\in$ *set* (*leaves-list ts*). $l < x \wedge x \leq u$
⟨*proof*⟩

lemma *aligned-split-left*: *aligned l* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) *u* $\Longrightarrow$ *aligned l* (*Node ls sub*) *sep*
⟨*proof*⟩

lemma *aligned-split-right*: *aligned l* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) *u* $\Longrightarrow$ *aligned sep* (*Node rs t*) *u*
⟨*proof*⟩

lemma *aligned-subst*: *aligned l* (*Node* (*ls*@(*sub'*, *subl*)#(*sub*,*subsep*)#*rs*) *t*) *u* $\Longrightarrow$
aligned subl subsub subsep $\Longrightarrow$
aligned l (*Node* (*ls*@(*sub'*,*subl*)#(*subsub*,*subsep*)#*rs*) *t*) *u*
⟨*proof*⟩

lemma *aligned-subst-emptyls*: *aligned l* (*Node* ((*sub*,*subsep*)#*rs*) *t*) *u* $\Longrightarrow$ *aligned l subsub subsep* $\Longrightarrow$
aligned l (*Node* ((*subsub*,*subsep*)#*rs*) *t*) *u*
⟨*proof*⟩

lemma *aligned-subst-last*: *aligned l* (*Node* (*ts'*@[(*sub'*, *sep'*)]) *t*) *u* $\Longrightarrow$ *aligned sep'*

t' u $\Longrightarrow$
  aligned l (*Node* (*ts'*@[(*sub'*, *sep'*)]) *t'*) *u*
  ⟨*proof*⟩

fun *Laligned* :: *'a* ::*linorder bplustree* ⇒ - **where**
Laligned (*Leaf ks*) *u* = (∀ *x* ∈ *set ks*. *x* ≤ *u*) |
Laligned (*Node ts t*) *u* = (*case ts of* [] ⇒ (*Laligned t u*) |
 (*sub*,*sep*)#*ts'* ⇒ ((*Laligned sub sep*) ∧ *inbetween aligned sep ts' t u*))

lemma *Laligned-nonempty-Node*: *Laligned* (*Node* ((*sub*,*sep*)#*ts'*) *t*) *u* =
  ((*Laligned sub sep*) ∧ *inbetween aligned sep ts' t u*)
  ⟨*proof*⟩

lemma *aligned-imp-Laligned*: *aligned l t u* $\Longrightarrow$ *Laligned t u*
  ⟨*proof*⟩

lemma *Laligned-split-left*: *Laligned* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) *u* $\Longrightarrow$ *Laligned* (*Node ls sub*) *sep*
  ⟨*proof*⟩

lemma *Laligned-split-right*: *Laligned* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) *u* $\Longrightarrow$ *aligned sep* (*Node rs t*) *u*
  ⟨*proof*⟩

lemma *Lalign-sub*: *Laligned* (*Node* ((*a*,*b*)#*ts*) *t*) *u* $\Longrightarrow$ (*sub*,*sep*) ∈ *set ts* $\Longrightarrow$ ∃ *l'* ∈ *set* (*separators ts*) ∪ {*b*}. *aligned l' sub sep*
  ⟨*proof*⟩

lemma *Lalign-last*: *Laligned* (*Node* (*ts*@[(*sub*,*sep*)]) *t*) *u* $\Longrightarrow$ *aligned sep t u*
  ⟨*proof*⟩

lemma *Lalign-last'*: *Laligned* (*Node* ((*a*,*b*)#*ts*) *t*) *u* $\Longrightarrow$ ∃ *l'* ∈ *set* (*separators ts*) ∪ {*b*}. *aligned l' t u*
  ⟨*proof*⟩

lemma *Lalign-Llast*: *Laligned* (*Node ts t*) *u* $\Longrightarrow$ *Laligned t u*
  ⟨*proof*⟩

lemma *Laligned-sorted-inorder*: *Laligned t u* $\Longrightarrow$ *sorted-less* ((*inorder t*)@[*u*])
⟨*proof*⟩

lemma *Laligned-sorted-separators*: *Laligned* (*Node ts t*) *u* $\Longrightarrow$ *sorted-less* ((*separators ts*)@[*u*])
  ⟨*proof*⟩

lemma *Laligned-leaves-inbetween*: *Laligned t u* $\Longrightarrow$ ∀ *x* ∈ *set* (*leaves t*). *x* ≤ *u*
⟨*proof*⟩

lemma *Laligned-leaves-list-inbetween*: *Laligned* (*Node ts t*) *u* $\Longrightarrow$ $\forall$ *x* $\in$ *set* (*leaves-list ts*). *x* $\leq$ *u*
⟨*proof*⟩

lemma *Laligned-subst-last*: *Laligned* (*Node* (*ts'*@[(*sub'*, *sep'*)]) *t*) *u* $\Longrightarrow$ *aligned sep' t' u* $\Longrightarrow$
Laligned (*Node* (*ts'*@[(*sub'*, *sep'*)]) *t'*) *u*
⟨*proof*⟩

lemma *Laligned-subst*: *Laligned* (*Node* (*ls*@(*sub'*, *subl*)#(*sub*,*subsep*)#*rs*) *t*) *u* $\Longrightarrow$
aligned subl subsub subsep $\Longrightarrow$
Laligned (*Node* (*ls*@(*sub'*,*subl*)#(*subsub*,*subsep*)#*rs*) *t*) *u*
⟨*proof*⟩

lemma *concat-leaf-nodes-leaves*: (*concat* (*map leaves* (*leaf-nodes t*))) = *leaves t*
⟨*proof*⟩

lemma *leaf-nodes-not-empty*: *leaf-nodes t* $\neq$ []
⟨*proof*⟩

end
theory *BPlusTree-Split*
imports *BPlusTree*
begin

5.6 Auxiliary functions

fun *split-half*:: - *list* $\Rightarrow$ - *list* $\times$ - *list* **where**
split-half xs = (*take* ((*length xs* + *1*) *div 2*) *xs*, *drop* ((*length xs* + *1*) *div 2*) *xs*)

lemma *split-half-conc*: *split-half xs* = (*ls*, *rs*) = (*xs* = *ls*@*rs* $\wedge$ *length ls* = (*length xs* + *1*) *div 2*)
⟨*proof*⟩

lemma *drop-not-empty*: *xs* $\neq$ [] $\Longrightarrow$ *drop* (*length xs div 2*) *xs* $\neq$ []
⟨*proof*⟩

lemma *take-not-empty*: *xs* $\neq$ [] $\Longrightarrow$ *take* ((*length xs* + *1*) *div 2*) *xs* $\neq$ []
⟨*proof*⟩

lemma *split-half-not-empty*: *length xs* $\geq$ *1* $\Longrightarrow$ $\exists$ *ls a rs*. *split-half xs* = (*ls*@[*a*],*rs*)
⟨*proof*⟩

5.7 The split function locale

Here, we abstract away the inner workings of the split function for B-tree operations.

lemma *leaves-conc*: *leaves* (*Node* (*ls*@*rs*) *t*) = *leaves-list ls* @ *leaves-list rs* @ *leaves*

t
⟨*proof*⟩

locale *split-tree* =
fixes *split* :: (*'a bplustree*×*'a*::{*linorder*,*order-top*}) *list* ⇒ *'a* ⇒ ((*'a bplustree*×*'a*) *list* × (*'a bplustree*×*'a*) *list*)
assumes *split-req*:
⟦*split xs p* = (*ls*,*rs*)⟧ ⟹ *xs* = *ls* @ *rs*
⟦*split xs p* = (*ls*@[(*sub*,*sep*)],*rs*); *sorted-less* (*separators xs*)⟧ ⟹ *sep* < *p*
⟦*split xs p* = (*ls*,(*sub*,*sep*)#*rs*); *sorted-less* (*separators xs*)⟧ ⟹ *p* ≤ *sep*
begin

lemmas *split-conc* = *split-req*(*1*)
lemmas *split-sorted* = *split-req*(*2*,*3*)

lemma [*termination-simp*]:(*ls*, (*sub*, *sep*) # *rs*) = *split ts y* ⟹
size sub < *Suc* (*size-list* (λ*x*. *Suc* (*size* (*fst x*))) *ts* + *size l*)
⟨*proof*⟩

lemma *leaves-split*: *split ts x* = (*ls*,*rs*) ⟹ *leaves* (*Node ts t*) = *leaves-list ls* @ *leaves-list rs* @ *leaves t*
⟨*proof*⟩

end

locale *split-list* =
fixes *split-list* :: (*'a*::{*linorder*,*order-top*}) *list* ⇒ *'a* ⇒ *'a list* × *'a list*
assumes *split-list-req*:
⟦*split-list ks p* = (*kls*,*krs*)⟧ ⟹ *ks* = *kls* @ *krs*
⟦*split-list ks p* = (*kls*@[*sep*],*krs*); *sorted-less ks*⟧ ⟹ *sep* < *p*
⟦*split-list ks p* = (*kls*,(*sep*)#*krs*); *sorted-less ks*⟧ ⟹ *p* ≤ *sep*

locale *split-full* = *split-tree*: *split-tree split* + *split-list split-list*
for *split*::
(*'a bplustree* × *'a*::{*linorder*,*order-top*}) *list* ⇒ *'a*
⇒ (*'a bplustree* × *'a*) *list* × (*'a bplustree* × *'a*) *list*
and *split-list*::
'a::{*linorder*,*order-top*} *list* ⇒ *'a*
⇒ *'a list* × *'a list*

6 Abstract split functions

6.1 Linear split

Finally we show that the split axioms are feasible by providing an example split function

Linear split is similar to well known functions, therefore a quick proof can be done.

fun *linear-split* **where** *linear-split xs x* = (*takeWhile* (λ(-,*s*). *s*<*x*) *xs*, *dropWhile* (λ(-,*s*). *s*<*x*) *xs*)
fun *linear-split-list* **where** *linear-split-list xs x* = (*takeWhile* (λ*s*. *s*<*x*) *xs*, *dropWhile* (λ*s*. *s*<*x*) *xs*)

end
theory *BPlusTree-Set*
  imports
    BPlusTree-Split
    HOL−Data-Structures.Set-Specs
begin

7 Set interpretation

lemma *insert-list-length*[*simp*]:
  assumes *sorted-less ks*
    and *set* (*insert-list k ks*) = *set ks* ∪ {*k*}
    and *sorted-less ks* ⟹ *sorted-less* (*insert-list k ks*)
  shows *length* (*insert-list k ks*) = *length ks* + (*if k* ∈ *set ks then 0 else 1*)
⟨*proof*⟩

lemma *delete-list-length*[*simp*]:
  assumes *sorted-less ks*
    and *set* (*delete-list k ks*) = *set ks* − {*k*}
    and *sorted-less ks* ⟹ *sorted-less* (*delete-list k ks*)
  shows *length* (*delete-list k ks*) = *length ks* − (*if k* ∈ *set ks then 1 else 0*)
⟨*proof*⟩

lemma *ins-list-length*[*simp*]:
  assumes *sorted-less ks*
  shows *length* (*ins-list k ks*) = *length ks* + (*if k* ∈ *set ks then 0 else 1*)
  ⟨*proof*⟩

lemma *del-list-length*[*simp*]:
  assumes *sorted-less ks*
  shows *length* (*del-list k ks*) = *length ks* − (*if k* ∈ *set ks then 1 else 0*)
  ⟨*proof*⟩

locale *split-set* = *split-tree*: *split-tree split*
  for *split*::
    (*'a bplustree* × *'a*::{*linorder*,*order-top*}) *list* ⇒ *'a*
      ⇒ (*'a bplustree* × *'a*) *list* × (*'a bplustree* × *'a*) *list* +
  fixes *isin-list* :: *'a* ⇒ (*'a*::{*linorder*,*order-top*}) *list* ⇒ *bool*
  and *insert-list* :: *'a* ⇒ (*'a*::{*linorder*,*order-top*}) *list* ⇒ *'a list*
  and *delete-list* :: *'a* ⇒ (*'a*::{*linorder*,*order-top*}) *list* ⇒ *'a list*
  assumes *insert-list-req*:

    sorted-less ks ⟹ *isin-list x ks* = (*x* ∈ *set ks*)
    sorted-less ks ⟹ *insert-list x ks* = *ins-list x ks*
    sorted-less ks ⟹ *delete-list x ks* = *del-list x ks*
begin

lemmas *split-req* = *split-tree.split-req*
lemmas *split-conc* = *split-tree.split-req*(*1*)
lemmas *split-sorted* = *split-tree.split-req*(*2*,*3*)

lemma *insert-list-length*[*simp*]:
  assumes *sorted-less ks*
  shows *length* (*insert-list k ks*) = *length ks* + (*if k* ∈ *set ks then 0 else 1*)
  ⟨*proof*⟩

lemma *set-insert-list*[*simp*]:
  sorted-less ks ⟹ *set* (*insert-list k ks*) = *set ks* ∪ {*k*}
  ⟨*proof*⟩

lemma *sorted-insert-list*[*simp*]:
  sorted-less ks ⟹ *sorted-less* (*insert-list k ks*)
  ⟨*proof*⟩

lemma *delete-list-length*[*simp*]:
  assumes *sorted-less ks*
  shows *length* (*delete-list k ks*) = *length ks* − (*if k* ∈ *set ks then 1 else 0*)
  ⟨*proof*⟩

lemma *set-delete-list*[*simp*]:
  sorted-less ks ⟹ *set* (*delete-list k ks*) = *set ks* − {*k*}
  ⟨*proof*⟩

lemma *sorted-delete-list*[*simp*]:
  sorted-less ks ⟹ *sorted-less* (*delete-list k ks*)
  ⟨*proof*⟩

definition *empty-bplustree* = (*Leaf* [])

7.1 Membership

fun *isin*:: *'a bplustree* ⇒ *'a* ⇒ *bool* **where**
  isin (*Leaf ks*) *x* = (*isin-list x ks*) |
  isin (*Node ts t*) *x* = (
    case split ts x of (-,(*sub*,*sep*)#*rs*) ⇒ (
      isin sub x
)
  | (-,[]) ⇒ *isin t x*
)

Isin proof

thm *isin-simps*

lemma *sorted-ConsD*: *sorted-less* (*y* # *xs*) ⟹ *x* ≤ *y* ⟹ *x* ∉ *set xs*
  ⟨*proof*⟩

lemma *sorted-snocD*: *sorted-less* (*xs* @ [*y*]) ⟹ *y* ≤ *x* ⟹ *x* ∉ *set xs*
  ⟨*proof*⟩

lemmas *isin-simps2* = *sorted-lems sorted-ConsD sorted-snocD*

lemma *isin-sorted*: *sorted-less* (*xs*@*a*#*ys*) ⟹
  (*x* ∈ *set* (*xs*@*a*#*ys*)) = (*if x* < *a then x* ∈ *set xs else x* ∈ *set* (*a*#*ys*))
  ⟨*proof*⟩

lemma *isin-sorted-split*:
  assumes *Laligned* (*Node ts t*) *u*
    and *sorted-less* (*leaves* (*Node ts t*))
    and *split ts x* = (*ls*, *rs*)
  shows *x* ∈ *set* (*leaves* (*Node ts t*)) = (*x* ∈ *set* (*leaves-list rs* @ *leaves t*))
⟨*proof*⟩

lemma *isin-sorted-split-right*:
  assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
    and *sorted-less* (*leaves* (*Node ts t*))
    and *Laligned* (*Node ts t*) *u*
  shows *x* ∈ *set* (*leaves-list* ((*sub*,*sep*)#*rs*) @ *leaves t*) = (*x* ∈ *set* (*leaves sub*))
⟨*proof*⟩

theorem *isin-set-inorder*:
  assumes *sorted-less* (*leaves t*)
    and *aligned l t u*
  shows *isin t x* = (*x* ∈ *set* (*leaves t*))

⟨*proof*⟩

theorem *isin-set-Linorder*:
  assumes *sorted-less* (*leaves t*)
    and *Laligned t u*
  shows *isin t x* = (*x* ∈ *set* (*leaves t*))
  ⟨*proof*⟩

corollary *isin-set-Linorder-top*:
  assumes *sorted-less* (*leaves t*)
    and *Laligned t top*
  shows *isin t x* = (*x* ∈ *set* (*leaves t*))
  ⟨*proof*⟩

7.2 Insertion

The insert function requires an auxiliary data structure and auxiliary invariant functions.

datatype *'b* up_i = T_i *'b bplustree* | Up_i *'b bplustree 'b 'b bplustree*

fun *order-*up_i **where**
  order-up_i *k* (T_i *sub*) = *order k sub* |
  order-up_i *k* (Up_i *l a r*) = (*order k l* ∧ *order k r*)

fun *root-order-*up_i **where**
  root-order-up_i *k* (T_i *sub*) = *root-order k sub* |
  root-order-up_i *k* (Up_i *l a r*) = (*order k l* ∧ *order k r*)

fun *height-*up_i **where**
  height-up_i (T_i *t*) = *height t* |
  height-up_i (Up_i *l a r*) = *max* (*height l*) (*height r*)

fun *bal-*up_i **where**
  bal-up_i (T_i *t*) = *bal t* |
  bal-up_i (Up_i *l a r*) = (*height l* = *height r* ∧ *bal l* ∧ *bal r*)

fun *inorder-*up_i **where**
  inorder-up_i (T_i *t*) = *inorder t* |
  inorder-up_i (Up_i *l a r*) = *inorder l* @ [*a*] @ *inorder r*

fun *leaves-*up_i **where**
  leaves-up_i (T_i *t*) = *leaves t* |
  leaves-up_i (Up_i *l a r*) = *leaves l* @ *leaves r*

fun *aligned-*up_i **where**
  aligned-up_i *l* (T_i *t*) *u* = *aligned l t u* |
  aligned-up_i *l* (Up_i *lt a rt*) *u* = (*aligned l lt a* ∧ *aligned a rt u*)

fun *Laligned-up*$_i$ **where**
Laligned-up$_i$ (T_i *t*) *u* = *Laligned t u* |
Laligned-up$_i$ (*Up*$_i$ *lt a rt*) *u* = (*Laligned lt a* ∧ *aligned a rt u*)

The following function merges two nodes and returns separately split nodes if an overflow occurs

```
fun node_i:: nat ⇒ ('a bplustree × 'a) list ⇒ 'a bplustree ⇒ 'a up_i where
  node_i k ts t = (
  if length ts ≤ 2*k then T_i (Node ts t)
  else (
    case split-half ts of (ls, rs) ⇒
      case last ls of (sub,sep) ⇒
        Up_i (Node (butlast ls) sub) sep (Node rs t)
    )
  )

fun Lnode_i:: nat ⇒ 'a list ⇒ 'a up_i where
  Lnode_i k ts = (
  if length ts ≤ 2*k then T_i (Leaf ts)
  else (
    case split-half ts of (ls, rs) ⇒
      Up_i (Leaf ls) (last ls) (Leaf rs)
    )
  )

fun ins:: nat ⇒ 'a ⇒ 'a bplustree ⇒ 'a up_i where
  ins k x (Leaf ks) = Lnode_i k (insert-list x ks) |
  ins k x (Node ts t) = (
  case split ts x of
    (ls,(sub,sep)#rs) ⇒
        (case ins k x sub of
          Up_i l a r ⇒
             node_i k (ls@(l,a)#(r,sep)#rs) t |
          T_i a ⇒
            T_i (Node (ls@(a,sep)#rs) t)) |
    (ls, []) ⇒
      (case ins k x t of
         Up_i l a r ⇒
           node_i k (ls@[(l,a)]) r |
         T_i a ⇒
           T_i (Node ls a)
  )
)

fun tree_i::'a up_i ⇒ 'a bplustree where
  tree_i (T_i sub) = sub |
```

tree$_i$ (*Up*$_i$ *l a r*) = (*Node* [(*l*,*a*)] *r*)

fun *insert*::*nat* ⇒ *'a* ⇒ *'a bplustree* ⇒ *'a bplustree* **where**
insert k x t = *tree*$_i$ (*ins k x t*)

7.3 Proofs of functional correctness

lemma *nodei-ti-simp*: *node*$_i$ *k ts t* = *T*$_i$ *x* ⟹ *x* = *Node ts t*
⟨*proof*⟩

lemma *Lnodei-ti-simp*: *Lnode*$_i$ *k ts* = *T*$_i$ *x* ⟹ *x* = *Leaf ts*
⟨*proof*⟩

lemma *split-set*:
assumes *split ts z* = (*ls*,(*a*,*b*)#*rs*)
shows (*a*,*b*) ∈ *set ts*
and (*x*,*y*) ∈ *set ls* ⟹ (*x*,*y*) ∈ *set ts*
and (*x*,*y*) ∈ *set rs* ⟹ (*x*,*y*) ∈ *set ts*
and *set ls* ∪ *set rs* ∪ {(*a*,*b*)} = *set ts*
and ∃ *x* ∈ *set ts*. *b* ∈ *Basic-BNFs.snds x*
⟨*proof*⟩

lemma *split-length*:
split ts x = (*ls*, *rs*) ⟹ *length ls* + *length rs* = *length ts*
⟨*proof*⟩

lemma *node*$_i$*-cases*: *length xs* ≤ *k* ∨ (∃ *ls sub sep rs*. *split-half xs* = (*ls*@[(*sub*,*sep*)],*rs*))
⟨*proof*⟩

lemma *Lnode*$_i$*-cases*: *length xs* ≤ *k* ∨ (∃ *ls sep rs*. *split-half xs* = (*ls*@[*sep*],*rs*))
⟨*proof*⟩

lemma *root-order-tree*$_i$: *root-order-up*$_i$ (*Suc k*) *t* = *root-order* (*Suc k*) (*tree*$_i$ *t*)
⟨*proof*⟩

lemma *length-take-left*: *length* (*take* ((*length ts* + *1*) *div 2*) *ts*) = (*length ts* + *1*) *div 2*
⟨*proof*⟩

lemma *node*$_i$*-root-order*:
assumes *length ts* > *0*
and *length ts* ≤ *4*∗*k*+*1*
and ∀ *x* ∈ *set* (*subtrees ts*). *order k x*
and *order k t*
shows *root-order-up*$_i$ *k* (*node*$_i$ *k ts t*)

⟨proof⟩

lemma *node$_i$-order-helper*:
assumes *length ts* $\geq$ *k*
and *length ts* $\leq$ *4*$*$*k*+*1*
and $\forall x \in$ *set* (*subtrees ts*). *order k x*
and *order k t*
shows *case* (*node$_i$ k ts t*) *of* T_i *t* $\Rightarrow$ *order k t* | - $\Rightarrow$ *True*
⟨proof⟩

lemma *node$_i$-order*:
assumes *length ts* $\geq$ *k*
and *length ts* $\leq$ *4*$*$*k*+*1*
and $\forall x \in$ *set* (*subtrees ts*). *order k x*
and *order k t*
shows *order-up$_i$ k* (*node$_i$ k ts t*)
⟨proof⟩

lemma *Lnode$_i$-root-order*:
assumes *length ts* > *0*
and *length ts* $\leq$ *4*$*$*k*
shows *root-order-up$_i$ k* (*Lnode$_i$ k ts*)
⟨proof⟩

lemma *Lnode$_i$-order-helper*:
assumes *length ts* $\geq$ *k*
and *length ts* $\leq$ *4*$*$*k*+*1*
shows *case* (*Lnode$_i$ k ts*) *of* T_i *t* $\Rightarrow$ *order k t* | - $\Rightarrow$ *True*
⟨proof⟩

lemma *Lnode$_i$-order*:
assumes *length ts* $\geq$ *k*
and *length ts* $\leq$ *4*$*$*k*
shows *order-up$_i$ k* (*Lnode$_i$ k ts*)
⟨proof⟩

lemma *ins-order*:
k > *0* $\Longrightarrow$ *sorted-less* (*leaves t*) $\Longrightarrow$ *order k t* $\Longrightarrow$ *order-up$_i$ k* (*ins k x t*)
⟨proof⟩

lemma *ins-root-order*:
assumes *k* > *0 sorted-less* (*leaves t*) *root-order k t*
shows *root-order-up$_i$ k* (*ins k x t*)

⟨*proof*⟩

lemma *height-list-split*: $height\text{-}up_i$ (Up_i (*Node ls a*) *b* (*Node rs t*)) = *height* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*)
⟨*proof*⟩

lemma $node_i$*-height*: $height\text{-}up_i$ ($node_i$ *k ts t*) = *height* (*Node ts t*)
⟨*proof*⟩

lemma $Lnode_i$*-height*: $height\text{-}up_i$ ($Lnode_i$ *k xs*) = *height* (*Leaf xs*)
⟨*proof*⟩

lemma $bal\text{-}up_i$*-tree*: $bal\text{-}up_i$ *t* = *bal* ($tree_i$ *t*)
⟨*proof*⟩

lemma *bal-list-split*: *bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*) $\Longrightarrow$ $bal\text{-}up_i$ (Up_i (*Node ls a*) *b* (*Node rs t*))
⟨*proof*⟩

lemma $node_i$*-bal*:
assumes *bal* (*Node ts t*)
shows $bal\text{-}up_i$ ($node_i$ *k ts t*)
⟨*proof*⟩

lemma $node_i$*-aligned*:
assumes *aligned l* (*Node ts t*) *u*
shows $aligned\text{-}up_i$ *l* ($node_i$ *k ts t*) *u*
⟨*proof*⟩

lemma $node_i$*-Laligned*:
assumes *Laligned* (*Node ts t*) *u*
shows $Laligned\text{-}up_i$ ($node_i$ *k ts t*) *u*
⟨*proof*⟩

lemma *length-right-side*: *length xs* > *1* $\Longrightarrow$ *length* (*drop* ((*length xs* + *1*) *div 2*) *xs*) > *0*
⟨*proof*⟩

lemma $Lnode_i$*-aligned*:
assumes *aligned l* (*Leaf ks*) *u*
and *sorted-less ks*
and *k* > *0*
shows $aligned\text{-}up_i$ *l* ($Lnode_i$ *k ks*) *u*
⟨*proof*⟩

lemma *height-up*$_i$*-merge*: *height-up*$_i$ (*Up*$_i$ *l a r*) = *height t* ⟹ *height* (*Node* (*ls*@(*t*,*x*)#*rs*) *tt*) = *height* (*Node* (*ls*@(*l*,*a*)#(*r*,*x*)#*rs*) *tt*)
⟨*proof*⟩

lemma *ins-height*: *height-up*$_i$ (*ins k x t*) = *height t*
⟨*proof*⟩

lemma *ins-bal*: *bal t* ⟹ *bal-up*$_i$ (*ins k x t*)
⟨*proof*⟩

lemma *node*$_i$*-leaves*: *leaves-up*$_i$ (*node*$_i$ *k ts t*) = *leaves* (*Node ts t*)
⟨*proof*⟩

corollary *node*$_i$*-leaves-simps*:
node$_i$ *k ts t* = *T*$_i$ *t'* ⟹ *leaves t'* = *leaves* (*Node ts t*)
node$_i$ *k ts t* = *Up*$_i$ *l a r* ⟹ *leaves l* @ *leaves r* = *leaves* (*Node ts t*)
⟨*proof*⟩

lemma *Lnode*$_i$*-leaves*: *leaves-up*$_i$ (*Lnode*$_i$ *k xs*) = *leaves* (*Leaf xs*)
⟨*proof*⟩

corollary *Lnode*$_i$*-leaves-simps*:
Lnode$_i$ *k xs* = *T*$_i$ *t* ⟹ *leaves t* = *leaves* (*Leaf xs*)
Lnode$_i$ *k xs* = *Up*$_i$ *l a r* ⟹ *leaves l* @ *leaves r* = *leaves* (*Leaf xs*)
⟨*proof*⟩

lemma *ins-list-split*:
assumes *Laligned* (*Node ts t*) *u*
and *sorted-less* (*leaves* (*Node ts t*))
and *split ts x* = (*ls*, *rs*)
shows *ins-list x* (*leaves* (*Node ts t*)) = *leaves-list ls* @ *ins-list x* (*leaves-list rs* @ *leaves t*)
⟨*proof*⟩

lemma *ins-list-split-right*:
assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
and *sorted-less* (*leaves* (*Node ts t*))
and *Laligned* (*Node ts t*) *u*
shows *ins-list x* (*leaves-list* ((*sub*,*sep*)#*rs*) @ *leaves t*) = *ins-list x* (*leaves sub*) @ *leaves-list rs* @ *leaves t*

⟨*proof*⟩

lemma *ins-list-idem-eq-isin*: *sorted-less xs* $\Longrightarrow$ $x \in$ *set xs* $\longleftrightarrow$ (*ins-list x xs* = *xs*)
⟨*proof*⟩

lemma *ins-list-contains-idem*: ⟦*sorted-less xs*; $x \in$ *set xs*⟧ $\Longrightarrow$ (*ins-list x xs* = *xs*)
⟨*proof*⟩

lemma *aligned-insert-list*: *sorted-less ks* $\Longrightarrow$ $l < x$ $\Longrightarrow$ $x \leq u$ $\Longrightarrow$ *aligned l* (*Leaf ks*) *u* $\Longrightarrow$ *aligned l* (*Leaf* (*insert-list x ks*)) *u*
⟨*proof*⟩

lemma *align-subst-two*: *aligned l* (*Node* (*ts*@[(*sub*,*sep*)]) *t*) *u* $\Longrightarrow$ *aligned sep lt a* $\Longrightarrow$ *aligned a rt u* $\Longrightarrow$ *aligned l* (*Node* (*ts*@[(*sub*,*sep*),(*lt*,*a*)]) *rt*) *u*
⟨*proof*⟩

lemma *align-subst-three*: *aligned l* (*Node* (*ls*@(*subl*,*sepl*)#(*subr*,*sepr*)#*rs*) *t*) *u* $\Longrightarrow$ *aligned sepl lt a* $\Longrightarrow$ *aligned a rt sepr* $\Longrightarrow$ *aligned l* (*Node* (*ls*@(*subl*,*sepl*)#(*lt*,*a*)#(*rt*,*sepr*)#*rs*) *t*) *u*
⟨*proof*⟩

declare $node_i$*.simps* [*simp del*]
declare $node_i$*-leaves* [*simp add*]

lemma *ins-inorder*:
assumes $k > 0$
and *aligned l t u*
and *sorted-less* (*leaves t*)
and *root-order k t*
and $l < x$ $x \leq u$
shows ($leaves\text{-}up_i$ (*ins k x t*)) = *ins-list x* (*leaves t*) $\wedge$ $aligned\text{-}up_i$ *l* (*ins k x t*) *u*
⟨*proof*⟩

declare $node_i$*.simps* [*simp add*]
declare $node_i$*-leaves* [*simp del*]

lemma *Laligned-insert-list*: *sorted-less ks* $\Longrightarrow$ $x \leq u$ $\Longrightarrow$ *Laligned* (*Leaf ks*) *u* $\Longrightarrow$ *Laligned* (*Leaf* (*insert-list x ks*)) *u*
⟨*proof*⟩

lemma *Lalign-subst-two*: *Laligned* (*Node* (*ts*@[(*sub*,*sep*)]) *t*) *u* $\Longrightarrow$ *aligned sep lt a* $\Longrightarrow$ *aligned a rt u* $\Longrightarrow$ *Laligned* (*Node* (*ts*@[(*sub*,*sep*),(*lt*,*a*)]) *rt*) *u*
⟨*proof*⟩

lemma *Lalign-subst-three*: *Laligned* (*Node* (*ls*@(*subl*,*sepl*)#(*subr*,*sepr*)#*rs*) *t*) *u* $\Longrightarrow$ *aligned sepl lt a* $\Longrightarrow$ *aligned a rt sepr* $\Longrightarrow$ *Laligned* (*Node* (*ls*@(*subl*,*sepl*)#(*lt*,*a*)#(*rt*,*sepr*)#*rs*)

t) *u*
  ⟨*proof*⟩

lemma *Lnode$_i$-Laligned*:
  assumes *Laligned* (*Leaf ks*) *u*
    and *sorted-less ks*
    and $k > 0$
  shows *Laligned-up$_i$* (*Lnode$_i$ k ks*) *u*
  ⟨*proof*⟩

declare *node$_i$.simps* [*simp del*]
declare *node$_i$-leaves* [*simp add*]

lemma *ins-Linorder*:
  assumes $k > 0$
    and *Laligned t u*
    and *sorted-less* (*leaves t*)
    and *root-order k t*
    and $x \leq u$
  shows (*leaves-up$_i$* (*ins k x t*)) = *ins-list x* (*leaves t*) ∧ *Laligned-up$_i$* (*ins k x t*) *u*
  ⟨*proof*⟩

declare *node$_i$.simps* [*simp add*]
declare *node$_i$-leaves* [*simp del*]

thm *ins.induct*
thm *bplustree.induct*

lemma *tree$_i$-bal*: *bal-up$_i$ u* ⟹ *bal* (*tree$_i$ u*)
  ⟨*proof*⟩

lemma *tree$_i$-order*: ⟦$k > 0$; *root-order-up$_i$ k u*⟧ ⟹ *root-order k* (*tree$_i$ u*)
  ⟨*proof*⟩

lemma *tree$_i$-inorder*: *inorder-up$_i$ u* = *inorder* (*tree$_i$ u*)
  ⟨*proof*⟩

lemma *tree$_i$-leaves*: *leaves-up$_i$ u* = *leaves* (*tree$_i$ u*)
  ⟨*proof*⟩

lemma *tree$_i$-aligned*: *aligned-up$_i$ l a u* ⟹ *aligned l* (*tree$_i$ a*) *u*
  ⟨*proof*⟩

lemma *tree$_i$-Laligned*: *Laligned-up$_i$ a u* ⟹ *Laligned* (*tree$_i$ a*) *u*
  ⟨*proof*⟩

lemma *insert-bal*: *bal t* ⟹ *bal* (*insert k x t*)
⟨*proof*⟩

lemma *insert-order*: ⟦*k* > *0*; *sorted-less* (*leaves t*); *root-order k t*⟧ ⟹ *root-order k* (*insert k x t*)
⟨*proof*⟩

lemma *insert-inorder*:
assumes *k* > *0 root-order k t sorted-less* (*leaves t*) *aligned l t u l* < *x x* ≤ *u*
shows *leaves* (*insert k x t*) = *ins-list x* (*leaves t*)
and *aligned l* (*insert k x t*) *u*
⟨*proof*⟩

lemma *insert-Linorder*:
assumes *k* > *0 root-order k t sorted-less* (*leaves t*) *Laligned t u x* ≤ *u*
shows *leaves* (*insert k x t*) = *ins-list x* (*leaves t*)
and *Laligned* (*insert k x t*) *u*
⟨*proof*⟩

corollary *insert-Linorder-top*:
assumes *k* > *0 root-order k t sorted-less* (*leaves t*) *Laligned t top*
shows *leaves* (*insert k x t*) = *ins-list x* (*leaves t*)
and *Laligned* (*insert k x t*) *top*
⟨*proof*⟩

7.4 Deletion

The following deletion method is inspired by Bauer (70) and Fielding (??). Rather than stealing only a single node from the neighbour, the neighbour is fully merged with the potentially underflowing node. If the resulting node is still larger than allowed, the merged node is split again, using the rules known from insertion splits. If the resulting node has admissable size, it is simply kept in the tree.

fun *rebalance-middle-tree* **where**
rebalance-middle-tree k ls (*Leaf ms*) *sep rs* (*Leaf ts*) = (
if length ms ≥ *k* ∧ *length ts* ≥ *k then*
Node (*ls*@(*Leaf ms*,*sep*)#*rs*) (*Leaf ts*)
else (
case rs of [] ⇒ (
case $Lnode_i$ *k* (*ms*@*ts*) *of*
T_i *u* ⇒
Node ls u |
Up_i *l a r* ⇒
Node (*ls*@[(*l*,*a*)]) *r*) |
(*Leaf rrs*,*rsep*)#*rs* ⇒ (
case $Lnode_i$ *k* (*ms*@*rrs*) *of*
T_i *u* ⇒

```
        Node (ls@(u,rsep)#rs) (Leaf ts) |
      Upᵢ l a r ⇒
        Node (ls@(l,a)#(r,rsep)#rs) (Leaf ts))
)) |
  rebalance-middle-tree k ls (Node mts mt) sep rs (Node tts tt) = (
  if length mts ≥ k ∧ length tts ≥ k then
    Node (ls@(Node mts mt,sep)#rs) (Node tts tt)
  else (
    case rs of [] ⇒ (
      case nodeᵢ k (mts@(mt,sep)#tts) tt of
       Tᵢ u ⇒
        Node ls u |
       Upᵢ l a r ⇒
        Node (ls@[(l,a)]) r) |
    (Node rts rt,rsep)#rs ⇒ (
      case nodeᵢ k (mts@(mt,sep)#rts) rt of
      Tᵢ u ⇒
        Node (ls@(u,rsep)#rs) (Node tts tt) |
      Upᵢ l a r ⇒
        Node (ls@(l,a)#(r,rsep)#rs) (Node tts tt))
))
```

All trees are merged with the right neighbour on underflow. Obviously for the last tree this would not work since it has no right neighbour. Therefore this tree, as the only exception, is merged with the left neighbour. However since we it does not make a difference, we treat the situation as if the second to last tree underflowed.

```
fun rebalance-last-tree where
  rebalance-last-tree k ts t = (
case last ts of (sub,sep) ⇒
   rebalance-middle-tree k (butlast ts) sub sep [] t
)
```

Rather than deleting the minimal key from the right subtree, we remove the maximal key of the left subtree. This is due to the fact that the last tree can easily be accessed and the left neighbour is way easier to access than the right neighbour, it resides in the same pair as the separating element to be removed.

```
fun del where
  del k x (Leaf xs) = (Leaf (delete-list x xs)) |
  del k x (Node ts t) = (
  case split ts x of
    (ls,[]) ⇒
     rebalance-last-tree k ls (del k x t)
  | (ls,(sub,sep)#rs) ⇒ (
     rebalance-middle-tree k ls (del k x sub) sep rs t
  )
)
```

fun *reduce-root* **where**
reduce-root (*Leaf xs*) = (*Leaf xs*) |
reduce-root (*Node ts t*) = (*case ts of*
[] ⇒ *t* |
- ⇒ (*Node ts t*)
)

fun *delete* **where** *delete k x t* = *reduce-root* (*del k x t*)

An invariant for intermediate states at deletion. In particular we allow for an underflow to 0 subtrees.

fun *almost-order* **where**
almost-order k (*Leaf xs*) = (*length xs* ≤ *2*∗*k*) |
almost-order k (*Node ts t*) = (
(*length ts* ≤ *2*∗*k*) ∧
(∀ *s* ∈ *set* (*subtrees ts*). *order k s*) ∧
order k t
)

Deletion proofs

thm *list.simps*

lemma *rebalance-middle-tree-height*:
assumes *height t* = *height sub*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
shows *height* (*rebalance-middle-tree k ls sub sep rs t*) = *height* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
⟨*proof*⟩

lemma *rebalance-last-tree-height*:
assumes *height t* = *height sub*
and *ts* = *list*@[(*sub*,*sep*)]
shows *height* (*rebalance-last-tree k ts t*) = *height* (*Node ts t*)
⟨*proof*⟩

lemma *bal-sub-height*: *bal* (*Node* (*ls*@*a*#*rs*) *t*) ⟹ (*case rs of* [] ⇒ *True* | (*sub*,*sep*)#- ⇒ *height sub* = *height t*)
⟨*proof*⟩

lemma *del-height*: ⟦*k* > *0*; *root-order k t*; *bal t*⟧ ⟹ *height* (*del k x t*) = *height t*
⟨*proof*⟩

lemma *rebalance-middle-tree-inorder*:
assumes *height t* = *height sub*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
shows *leaves* (*rebalance-middle-tree k ls sub sep rs t*) = *leaves* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
⟨*proof*⟩

lemma *rebalance-last-tree-inorder*:
assumes *height t* = *height sub*
and *ts* = *list*@[(*sub*,*sep*)]
shows *leaves* (*rebalance-last-tree k ts t*) = *leaves* (*Node ts t*)
⟨*proof*⟩

lemma *butlast-inorder-app-id*: *xs* = *xs′* @ [(*sub*,*sep*)] ⟹ *inorder-list xs′* @ *inorder sub* @ [*sep*] = *inorder-list xs*
⟨*proof*⟩

lemma *height-bal-subtrees-merge*: ⟦*height* (*Node as a*) = *height* (*Node bs b*); *bal* (*Node as a*); *bal* (*Node bs b*)⟧
⟹ ∀ *x* ∈ *set* (*subtrees as*) ∪ {*a*}. *height x* = *height b*
⟨*proof*⟩

lemma *bal-list-merge*:
assumes $bal\text{-}up_i$ (Up_i (*Node as a*) *x* (*Node bs b*))
shows *bal* (*Node* (*as*@(*a*,*x*)#*bs*) *b*)
⟨*proof*⟩

lemma $node_i$-bal-up_i:
assumes $bal\text{-}up_i$ ($node_i$ *k ts t*)
shows *bal* (*Node ts t*)
⟨*proof*⟩

lemma $node_i$-*bal-simp*: $bal\text{-}up_i$ ($node_i$ *k ts t*) = *bal* (*Node ts t*)
⟨*proof*⟩

lemma *rebalance-middle-tree-bal*:
assumes *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
shows *bal* (*rebalance-middle-tree k ls sub sep rs t*)
⟨*proof*⟩

lemma *rebalance-last-tree-bal*: ⟦*bal* (*Node ts t*); *ts* ≠ []⟧ ⟹ *bal* (*rebalance-last-tree k ts t*)
⟨*proof*⟩

lemma *Leaf-merge-aligned*: *aligned l* (*Leaf ms*) *m* ⟹ *aligned m* (*Leaf rs*) *r* ⟹

aligned l (*Leaf* (*ms*@*rs*)) *r*
⟨*proof*⟩

lemma *Node-merge-aligned*:
inbetween aligned l mts mt sep ⟹
inbetween aligned sep tts tt u ⟹
inbetween aligned l (*mts* @ (*mt*, *sep*) # *tts*) *tt u*
⟨*proof*⟩

lemma *aligned-subst-last-merge*: *aligned l* (*Node* (*ts'*@[(*sub'*, *sep'*),(*sub*,*sep*)]) *t*) *u*
⟹ *aligned sep' t' u* ⟹
aligned l (*Node* (*ts'*@[(*sub'*, *sep'*)]) *t'*) *u*
⟨*proof*⟩

lemma *aligned-subst-last-merge-two*: *aligned l* (*Node* (*ts*@[(*sub'*,*sep'*),(*sub*,*sep*)]) *t*) *u* ⟹ *aligned sep' lt a* ⟹ *aligned a rt u* ⟹ *aligned l* (*Node* (*ts*@[(*sub'*,*sep'*),(*lt*,*a*)]) *rt*) *u*
⟨*proof*⟩

lemma *aligned-subst-merge*: *aligned l* (*Node* (*ls*@(*lsub*, *lsep*)#(*sub*,*sep*)#(*rsub*,*rsep*)#*rs*) *t*) *u* ⟹ *aligned lsep sub' rsep* ⟹
aligned l (*Node* (*ls*@(*lsub*, *lsep*)#(*sub'*, *rsep*)#*rs*) *t*) *u*
⟨*proof*⟩

lemma *aligned-subst-merge-two*: *aligned l* (*Node* (*ls*@(*lsub*, *lsep*)#(*sub*,*sep*)#(*rsub*,*rsep*)#*rs*) *t*) *u* ⟹ *aligned lsep sub' a* ⟹
aligned a rsub' rsep ⟹ *aligned l* (*Node* (*ls*@(*lsub*, *lsep*)#(*sub'*,*a*)#(*rsub'*, *rsep*)#*rs*) *t*) *u*
⟨*proof*⟩

lemma *rebalance-middle-tree-aligned*:
assumes *aligned l* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) *u*
and *height t* = *height sub*
and *sorted-less* (*leaves* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*))
and *k* > *0*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
shows *aligned l* (*rebalance-middle-tree k ls sub sep rs t*) *u*
⟨*proof*⟩

lemma *Node-merge-Laligned*:
Laligned (*Node mts mt*) *sep* ⟹
inbetween aligned sep tts tt u ⟹
Laligned (*Node* (*mts* @ (*mt*, *sep*) # *tts*) *tt*) *u*
⟨*proof*⟩

lemma *Laligned-subst-last-merge*: *Laligned* (*Node* (*ts'*@[(*sub'*, *sep'*),(*sub*,*sep*)]) *t*) *u* ⟹ *aligned sep' t' u* ⟹
Laligned (*Node* (*ts'*@[(*sub'*, *sep'*)]) *t'*) *u*
⟨*proof*⟩

lemma *Laligned-subst-last-merge-two*: *Laligned* (*Node* (*ts*@[(*sub'*,*sep'*),(*sub*,*sep*)]) *t*) *u* ⟹ *aligned sep' lt a* ⟹ *aligned a rt u* ⟹ *Laligned* (*Node* (*ts*@[(*sub'*,*sep'*),(*lt*,*a*)]) *rt*) *u*
⟨*proof*⟩

lemma *Laligned-subst-merge*: *Laligned* (*Node* (*ls*@(*lsub*, *lsep*)#(*sub*,*sep*)#(*rsub*,*rsep*)#*rs*) *t*) *u* ⟹ *aligned lsep sub' rsep* ⟹
Laligned (*Node* (*ls*@(*lsub*, *lsep*)#(*sub'*, *rsep*)#*rs*) *t*) *u*
⟨*proof*⟩

lemma *Laligned-subst-merge-two*: *Laligned* (*Node* (*ls*@(*lsub*, *lsep*)#(*sub*,*sep*)#(*rsub*,*rsep*)#*rs*) *t*) *u* ⟹ *aligned lsep sub' a* ⟹
aligned a rsub' rsep ⟹ *Laligned* (*Node* (*ls*@(*lsub*, *lsep*)#(*sub'*,*a*)#(*rsub'*, *rsep*)#*rs*) *t*) *u*
⟨*proof*⟩

lemma *xs-front*: *xs* @ [(*a*,*b*)] = (*x*,*y*)#*xs'* ⟹ *xs* @ [(*a*,*b*),(*c*,*d*)] = (*z*,*zz*)#*xs''* ⟹ (*x*,*y*) = (*z*,*zz*)
⟨*proof*⟩

lemma *rebalance-middle-tree-Laligned*:
assumes *Laligned* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) *u*
and *height t* = *height sub*
and *sorted-less* (*leaves* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*))
and *k* > *0*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
shows *Laligned* (*rebalance-middle-tree k ls sub sep rs t*) *u*
⟨*proof*⟩

lemma *rebalance-last-tree-aligned*:
assumes *aligned l* (*Node* (*ls*@[(*sub*,*sep*)]) *t*) *u*
and *height t* = *height sub*
and *sorted-less* (*leaves* (*Node* (*ls*@[(*sub*,*sep*)]) *t*))
and *k* > *0*
shows *aligned l* (*rebalance-last-tree k* (*ls*@[(*sub*,*sep*)]) *t*) *u*
⟨*proof*⟩

lemma *rebalance-last-tree-Laligned*:
assumes *Laligned* (*Node* (*ls*@[(*sub*,*sep*)]) *t*) *u*
and *height t* = *height sub*
and *sorted-less* (*leaves* (*Node* (*ls*@[(*sub*,*sep*)]) *t*))
and *k* > *0*
shows *Laligned* (*rebalance-last-tree k* (*ls*@[(*sub*,*sep*)]) *t*) *u*
⟨*proof*⟩

lemma *del-bal*:
assumes *k* > *0*
and *root-order k t*

and *bal t*
shows *bal* (*del k x t*)
⟨*proof*⟩

lemma *rebalance-middle-tree-order*:
assumes *almost-order k sub*
and ∀ *s* ∈ *set* (*subtrees* (*ls*@*rs*)). *order k s order k t*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
and *length* (*ls*@(*sub*,*sep*)#*rs*) ≤ *2*∗*k*
and *height sub* = *height t*
shows *almost-order k* (*rebalance-middle-tree k ls sub sep rs t*)
⟨*proof*⟩

lemma *rebalance-middle-tree-last-order*:
assumes *almost-order k t*
and ∀ *s* ∈ *set* (*subtrees* (*ls*@(*sub*,*sep*)#*rs*)). *order k s*
and *rs* = []
and *length* (*ls*@(*sub*,*sep*)#*rs*) ≤ *2*∗*k*
and *height sub* = *height t*
shows *almost-order k* (*rebalance-middle-tree k ls sub sep rs t*)
⟨*proof*⟩

lemma *rebalance-last-tree-order*:
assumes *ts* = *ls*@[(*sub*,*sep*)]
and ∀ *s* ∈ *set* (*subtrees* (*ts*)). *order k s almost-order k t*
and *length ts* ≤ *2*∗*k*
and *height sub* = *height t*
shows *almost-order k* (*rebalance-last-tree k ts t*)
⟨*proof*⟩

lemma *del-order*:
assumes *k* > *0*
and *root-order k t*
and *bal t*
and *sorted* (*leaves t*)
shows *almost-order k* (*del k x t*)
⟨*proof*⟩

thm *del-list-sorted*

lemma *del-list-split*:
assumes *Laligned* (*Node ts t*) *u*
and *sorted-less* (*leaves* (*Node ts t*))

and *split ts x* = (*ls*, *rs*)
shows *del-list x* (*leaves* (*Node ts t*)) = *leaves-list ls* @ *del-list x* (*leaves-list rs* @ *leaves t*)
⟨*proof*⟩

corollary *del-list-split-aligned*:
assumes *aligned l* (*Node ts t*) *u*
and *sorted-less* (*leaves* (*Node ts t*))
and *split ts x* = (*ls*, *rs*)
shows *del-list x* (*leaves* (*Node ts t*)) = *leaves-list ls* @ *del-list x* (*leaves-list rs* @ *leaves t*)
⟨*proof*⟩

lemma *del-list-split-right*:
assumes *Laligned* (*Node ts t*) *u*
and *sorted-less* (*leaves* (*Node ts t*))
and *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
shows *del-list x* (*leaves-list* ((*sub*,*sep*)#*rs*) @ *leaves t*) = *del-list x* (*leaves sub*) @ *leaves-list rs* @ *leaves t*
⟨*proof*⟩

corollary *del-list-split-right-aligned*:
assumes *aligned l* (*Node ts t*) *u*
and *sorted-less* (*leaves* (*Node ts t*))
and *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
shows *del-list x* (*leaves-list* ((*sub*,*sep*)#*rs*) @ *leaves t*) = *del-list x* (*leaves sub*) @ *leaves-list rs* @ *leaves t*
⟨*proof*⟩

thm *del-list-idem*

lemma *del-inorder*:
assumes $k > 0$
and *root-order k t*
and *bal t*
and *sorted-less* (*leaves t*)
and *aligned l t u*
and $l < x$ $x \leq u$
shows *leaves* (*del k x t*) = *del-list x* (*leaves t*) ∧ *aligned l* (*del k x t*) *u*
⟨*proof*⟩

lemma *del-Linorder*:
assumes $k > 0$
and *root-order k t*
and *bal t*
and *sorted-less* (*leaves t*)
and *Laligned t u*

and $x \leq u$
shows *leaves* (*del k x t*) = *del-list x* (*leaves t*) ∧ *Laligned* (*del k x t*) *u*
⟨*proof*⟩

lemma *reduce-root-order*: ⟦*k* > *0*; *almost-order k t*⟧ ⟹ *root-order k* (*reduce-root t*)
⟨*proof*⟩

lemma *reduce-root-bal*: *bal* (*reduce-root t*) = *bal t*
⟨*proof*⟩

lemma *reduce-root-inorder*: *leaves* (*reduce-root t*) = *leaves t*
⟨*proof*⟩

lemma *reduce-root-Laligned*: *Laligned* (*reduce-root t*) *u* = *Laligned t u*
⟨*proof*⟩

lemma *delete-order*: ⟦*k* > *0*; *bal t*; *root-order k t*; *sorted-less* (*leaves t*)⟧ ⟹
root-order k (*delete k x t*)
⟨*proof*⟩

lemma *delete-bal*: ⟦*k* > *0*; *bal t*; *root-order k t*⟧ ⟹ *bal* (*delete k x t*)
⟨*proof*⟩

lemma *delete-Linorder*:
assumes *k* > *0 root-order k t sorted-less* (*leaves t*) *Laligned t u bal t x* ≤ *u*
shows *leaves* (*delete k x t*) = *del-list x* (*leaves t*)
and *Laligned* (*delete k x t*) *u*
⟨*proof*⟩

corollary *delete-Linorder-top*:
assumes *k* > *0 root-order k t sorted-less* (*leaves t*) *Laligned t top bal t*
shows *leaves* (*delete k x t*) = *del-list x* (*leaves t*)
and *Laligned* (*delete k x t*) *top*
⟨*proof*⟩

7.5 Set specification by inorder

fun *invar-leaves* **where** *invar-leaves k t* = (
bal t ∧
root-order k t ∧
Laligned t top
)

interpretation *S-ordered*: *Set-by-Ordered* **where**
empty = *empty-bplustree* **and**
insert = *insert* (*Suc k*) **and**

delete = *delete* (*Suc k*) **and**
isin = *isin* **and**
inorder = *leaves* **and**
inv = *invar-leaves* (*Suc k*)
⟨*proof*⟩

declare $node_i$.*simps*[*simp del*]

end

lemma *sorted-ConsD*: *sorted-less* (*y* # *xs*) $\Longrightarrow x \leq y \Longrightarrow x \notin$ *set xs*
⟨*proof*⟩

lemma *sorted-snocD*: *sorted-less* (*xs* @ [*y*]) $\Longrightarrow y \leq x \Longrightarrow x \notin$ *set xs*
⟨*proof*⟩

lemmas *isin-simps2* = *sorted-lems sorted-ConsD sorted-snocD*

lemma *isin-sorted*: *sorted-less* (*xs*@*a*#*ys*) $\Longrightarrow$
($x \in$ *set* (*xs*@*a*#*ys*)) = (*if* $x < a$ *then* $x \in$ *set xs else* $x \in$ *set* (*a*#*ys*))
⟨*proof*⟩

context *split-list*
begin

fun *isin-list* :: *'a* ⇒ *'a list* ⇒ *bool* **where**
isin-list x ks = (*case split-list ks x of*
(*ls*,*Nil*) ⇒ *False* |
(*ls*,*sep*#*rs*) ⇒ *sep* = *x*
)

fun *insert-list* **where**
insert-list x ks = (*case split-list ks x of*
(*ls*,*Nil*) ⇒ *ls*@[*x*] |
(*ls*,*sep*#*rs*) ⇒ *if sep* = *x then ks else ls*@*x*#*sep*#*rs*
)

fun *delete-list* **where**
delete-list x ks = (*case split-list ks x of*
(*ls*,*Nil*) ⇒ *ks* |
(*ls*,*sep*#*rs*) ⇒ *if sep* = *x then ls*@*rs else ks*
)

lemmas *split-list-conc* = *split-list-req*(*1*)

lemmas *split-list-sorted* = *split-list-req*(*2*,*3*)

lemma *isin-sorted-split-list*:
assumes *sorted-less xs*
  and *split-list xs x* = (*ls*, *rs*)
 shows (*x* ∈ *set xs*) = (*x* ∈ *set rs*)
⟨*proof*⟩

lemma *isin-sorted-split-list-right*:
 assumes *split-list ts x* = (*ls*, *sep#rs*)
  and *sorted-less ts*
 shows *x* ∈ *set* (*sep#rs*) = (*x* = *sep*)
⟨*proof*⟩

theorem *isin-list-set*:
 assumes *sorted-less xs*
 shows *isin-list x xs* = (*x* ∈ *set xs*)
 ⟨*proof*⟩

lemma *insert-sorted-split-list*:
assumes *sorted-less xs*
  and *split-list xs x* = (*ls*, *rs*)
 shows *ins-list x xs* = *ls* @ *ins-list x rs*
⟨*proof*⟩

lemma *insert-sorted-split-list-right*:
 assumes *split-list ts x* = (*ls*, *sep#rs*)
  and *sorted-less ts*
  and *x* ≠ *sep*
 shows *ins-list x* (*sep#rs*) = (*x#sep#rs*)
⟨*proof*⟩

theorem *insert-list-set*:
 assumes *sorted-less xs*
 shows *insert-list x xs* = *ins-list x xs*
 ⟨*proof*⟩

lemma *delete-sorted-split-list*:
assumes *sorted-less xs*
  and *split-list xs x* = (*ls*, *rs*)
 shows *del-list x xs* = *ls* @ *del-list x rs*
⟨*proof*⟩

lemma *delete-sorted-split-list-right*:

assumes *split-list ts x* = (*ls*, *sep*#*rs*)
and *sorted-less ts*
and $x \neq sep$
shows *del-list x* (*sep*#*rs*) = *sep*#*rs*
⟨*proof*⟩

theorem *delete-list-set*:
assumes *sorted-less xs*
shows *delete-list x xs* = *del-list x xs*
⟨*proof*⟩

end

context *split-full*
begin

sublocale *split-set split isin-list insert-list delete-list*
⟨*proof*⟩

end

end
theory *BPlusTree-Range*
imports *BPlusTree*
HOL−Data-Structures.Set-Specs
HOL−Library.Sublist
BPlusTree-Split
begin

Lrange describes all elements in a set that are greater or equal to l, a lower bounded range (with no upper bound)

definition *Lrange* **where**
Lrange l X = $\{x \in X.\ x \geq l\}$

definition *lrange-filter l* = *filter* ($\lambda x.\ x \geq l$)

lemma *lrange-filter-iff-Lrange*: *set* (*lrange-filter l xs*) = *Lrange l* (*set xs*)
⟨*proof*⟩

fun *lrange-list* **where**
lrange-list l (*x*#*xs*) = (*if* $x \geq l$ *then* (*x*#*xs*) *else lrange-list l xs*) |
lrange-list l [] = []

lemma *sorted-leq-lrange*: *sorted-wrt* ($\leq$) *xs* $\Longrightarrow$ *lrange-list* (*l*::'*a*::*linorder*) *xs* = *lrange-filter l xs*
⟨*proof*⟩

lemma *sorted-less-lrange*: *sorted-less xs* ⟹ *lrange-list* (*l*::′*a*::*linorder*) *xs* = *lrange-filter l xs*
⟨*proof*⟩

lemma *lrange-list-sorted*: *sorted-less* (*xs*@*x*#*ys*) ⟹
lrange-list l (*xs*@*x*#*ys*) =
(*if l* < *x then* (*lrange-list l xs*)@*x*#*ys else lrange-list l* (*x*#*ys*))
⟨*proof*⟩

lemma *lrange-filter-sorted*: *sorted-less* (*xs*@*x*#*ys*) ⟹
lrange-filter l (*xs*@*x*#*ys*) =
(*if l* < *x then* (*lrange-filter l xs*)@*x*#*ys else lrange-filter l* (*x*#*ys*))
⟨*proof*⟩

lemma *lrange-suffix*: *suffix* (*lrange-list l xs*) *xs*
⟨*proof*⟩

locale *split-range* = *split-tree split*
for *split*::
(′*a bplustree* × ′*a*::{*linorder*,*order-top*}) *list* ⇒ ′*a*
⇒ (′*a bplustree* × ′*a*) *list* × (′*a bplustree* × ′*a*) *list* +
fixes *lrange-list* :: ′*a* ⇒ (′*a*::{*linorder*,*order-top*}) *list* ⇒ ′*a list*
assumes *lrange-list-req*:

sorted-less ks ⟹ *lrange-list l ks* = *lrange-filter l ks*
begin

fun *lrange*:: ′*a bplustree* ⇒ ′*a* ⇒ ′*a list* **where**
lrange (*Leaf ks*) *x* = (*lrange-list x ks*) |
lrange (*Node ts t*) *x* = (
case split ts x of (-,(*sub*,*sep*)#*rs*) ⇒ (
lrange sub x @ *leaves-list rs* @ *leaves t*
)
| (-,[]) ⇒ *lrange t x*
)

lrange proof

lemma *lrange-sorted-split*:
assumes *Laligned* (*Node ts t*) *u*
and *sorted-less* (*leaves* (*Node ts t*))
and *split ts x* = (*ls*, *rs*)
shows *lrange-filter x* (*leaves* (*Node ts t*)) = *lrange-filter x* (*leaves-list rs* @ *leaves t*)
⟨*proof*⟩

lemma *lrange-sorted-split-right*:

assumes *split ts x* = (*ls*, (*sub*,*sep*)#*rs*)
and *sorted-less* (*leaves* (*Node ts t*))
and *Laligned* (*Node ts t*) *u*
shows *lrange-filter x* (*leaves-list* ((*sub*,*sep*)#*rs*) @ *leaves t*) = *lrange-filter x* (*leaves sub*)@*leaves-list rs*@*leaves t*
⟨*proof*⟩

theorem *lrange-set*:
assumes *sorted-less* (*leaves t*)
and *aligned l t u*
shows *lrange t x* = *lrange-filter x* (*leaves t*)
⟨*proof*⟩

Now the alternative explanation that first obtains the correct leaf node and in a second step obtains the correct element from the leaf node.

```
fun leaf-nodes-lrange:: 'a bplustree ⇒ 'a ⇒ 'a bplustree list where
  leaf-nodes-lrange (Leaf ks) x = [Leaf ks] |
  leaf-nodes-lrange (Node ts t) x = (
      case split ts x of (-,(sub,sep)#rs) ⇒ (
             leaf-nodes-lrange sub x @ leaf-nodes-list rs @ leaf-nodes t
      )
   | (-,[]) ⇒ leaf-nodes-lrange t x
  )
```

lrange proof

lemma *concat-leaf-nodes-leaves-list*: (*concat* (*map leaves* (*leaf-nodes-list ts*))) = *leaves-list ts*
⟨*proof*⟩

theorem *leaf-nodes-lrange-set*:
assumes *sorted-less* (*leaves t*)
and *aligned l t u*
shows *suffix* (*lrange-filter x* (*leaves t*)) (*concat* (*map leaves* (*leaf-nodes-lrange t x*)))
⟨*proof*⟩

lemma *leaf-nodes-lrange-not-empty*: ∃ *ks list*. *leaf-nodes-lrange t x* = (*Leaf ks*)#*list* ∧ (*Leaf ks*) ∈ *set* (*leaf-nodes t*)
⟨*proof*⟩

Note that, conveniently, this argument is purely syntactic, we do not need to show that this has anything to do with linear orders

lemma *leaf-nodes-lrange-pre-lrange*: *leaf-nodes-lrange t x* = (*Leaf ks*)#*list* ⟹ *lrange-list x ks* @ (*concat* (*map leaves list*)) = *lrange t x*
⟨*proof*⟩

We finally obtain a function that is way easier to reason about in the imperative setting

fun *concat-leaf-nodes-lrange* **where**
concat-leaf-nodes-lrange t x = (case leaf-nodes-lrange t x of (Leaf ks)#list ⇒ lrange-list x ks @ (concat (map leaves list)))

lemma *concat-leaf-nodes-lrange-lrange*: *concat-leaf-nodes-lrange t x = lrange t x*
⟨*proof*⟩

end

context *split-list*
begin

definition *lrange-split* **where**
lrange-split l xs = (case split-list xs l of (ls,rs) ⇒ rs)

lemma *lrange-filter-split*:
assumes *sorted-less xs*
and *split-list xs l = (ls,rs)*
shows *lrange-list l xs = rs*
find-theorems *split-list*
⟨*proof*⟩

lemma *lrange-split-req*:
assumes *sorted-less xs*
shows *lrange-split l xs = lrange-filter l xs*
⟨*proof*⟩

end

context *split-full*
begin

sublocale *split-range split lrange-split*
⟨*proof*⟩

end

end
theory *BPlusTree-SplitCE*
imports
BPlusTree-Set
BPlusTree-Range
begin

global-interpretation *bplustree-linear-search-list*: *split-list linear-split-list*
defines *bplustree-ls-isin-list = bplustree-linear-search-list.isin-list*
and *bplustree-ls-insert-list = bplustree-linear-search-list.insert-list*
and *bplustree-ls-delete-list = bplustree-linear-search-list.delete-list*
and *bplustree-ls-lrange-list = bplustree-linear-search-list.lrange-split*

⟨*proof*⟩

declare *bplustree-linear-search-list.isin-list.simps*[*code*]
declare *bplustree-linear-search-list.insert-list.simps*[*code*]
declare *bplustree-linear-search-list.delete-list.simps*[*code*]

global-interpretation *bplustree-linear-search*:
split-full linear-split linear-split-list

defines *bplustree-ls-isin* = *bplustree-linear-search.isin*
and *bplustree-ls-ins* = *bplustree-linear-search.ins*
and *bplustree-ls-insert* = *bplustree-linear-search.insert*
and *bplustree-ls-del* = *bplustree-linear-search.del*
and *bplustree-ls-delete* = *bplustree-linear-search.delete*
and *bplustree-ls-lrange* = *bplustree-linear-search.lrange*
⟨*proof*⟩

lemma [*code*]: *bplustree-ls-isin* (*Leaf ks*) *x* = *bplustree-ls-isin-list x ks*
⟨*proof*⟩
declare *bplustree-linear-search.isin.simps*(*2*)[*code*]

lemma [*code*]: *bplustree-ls-ins k x* (*Leaf ks*) =
bplustree-linear-search.$Lnode_i$ *k* (*bplustree-ls-insert-list x ks*)
⟨*proof*⟩
declare *bplustree-linear-search.ins.simps*(*2*)[*code*]

lemma [*code*]: *bplustree-ls-del k x* (*Leaf ks*) =
Leaf (*bplustree-ls-delete-list x ks*)
⟨*proof*⟩
declare *bplustree-linear-search.del.simps*(*2*)[*code*]

find-theorems *bplustree-ls-isin*

Some examples follow to show that the implementation works and the above lemmas make sense. The examples are visualized in the thesis.

abbreviation $bplustree_q$ ≡ *bplustree-ls-isin*
abbreviation $bplustree_i$ ≡ *bplustree-ls-insert*
abbreviation $bplustree_d$ ≡ *bplustree-ls-delete*

definition *uint8-max* ≡ *2*^*8*−*1*::*nat*
declare *uint8-max-def*[*simp*]

typedef *uint8* = {*n*::*nat*. *n* ≤ *uint8-max*}
⟨*proof*⟩

setup-lifting *type-definition-uint8*

instantiation *uint8* :: *linorder*
begin

lift-definition *less-eq-uint8* :: *uint8* ⇒ *uint8* ⇒ *bool*
is (*less-eq*::*nat* ⇒ *nat* ⇒ *bool*) ⟨*proof*⟩

lift-definition *less-uint8* :: *uint8* ⇒ *uint8* ⇒ *bool*
is (*less*::*nat* ⇒ *nat* ⇒ *bool*) ⟨*proof*⟩

instance
⟨*proof*⟩
end

instantiation *uint8* :: *order-top*
begin

lift-definition *top-uint8* :: *uint8* **is** *uint8-max*::*nat*
⟨*proof*⟩

instance
⟨*proof*⟩
end

instantiation *uint8* :: *numeral*
begin

lift-definition *one-uint8* :: *uint8* **is** *1*::*nat*
⟨*proof*⟩

lift-definition *plus-uint8* :: *uint8* ⇒ *uint8* ⇒ *uint8*
is λ*a b*. *min* (*a* + *b*) *uint8-max*
⟨*proof*⟩

instance ⟨*proof*⟩
end

instantiation *uint8* :: *equal*
begin

lift-definition *equal-uint8* :: *uint8* ⇒ *uint8* ⇒ *bool*
is (=) ⟨*proof*⟩

instance ⟨*proof*⟩
end

value *uint8-max*

value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [21,22,23]))) in*
root-order k x
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [21,22,23]))) in*
bal x
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [50,55,56]))) in*
sorted-less (leaves x)
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [50,55,56]))) in*
Laligned x top
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [50,55,56]))) in*
x
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [50,55,56]))) in*
$bplustree_q$ *x 4*
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [50,55,56]))) in*
$bplustree_q$ *x 20*
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [50,55,56]))) in*
$bplustree_i$ *k 9 x*
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [50,55,56]))) in*
$bplustree_i$ *k 1 (*$bplustree_i$ *k 9 x)*
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [50,55,56]))) in*
$bplustree_d$ *k 10 (*$bplustree_i$ *k 1 (*$bplustree_i$ *k 9 x))*
value *let k=2::nat; x::uint8 bplustree = (Node [(Node [(Leaf [1,2], 2),(Leaf [3,4], 4),(Leaf [5,6,7], 8)] (Leaf [9,10]), 10)] (Node [(Leaf [11,12,13,14], 14), (Leaf [15,17], 20)] (Leaf [50,55,56]))) in*
$bplustree_d$ *k 3 (*$bplustree_d$ *k 10 (*$bplustree_i$ *k 1 (*$bplustree_i$ *k 9 x)))*

end

References

[1] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972. doi:10.1007/BF00288683. URL https://doi.org/10.1007/BF00288683.

[2] Peter Lammich. The imperative refinement framework. *Archive of Formal Proofs*, August 2016. ISSN 2150-914x. https://isa-afp.org/entries/Refine_Imperative_HOL.html, Formal proof development.

[3] Niels Mündler. A verified imperative implementation of b-trees. Bachelor's thesis, Technische Universität München, München, 2021. URL https://mediatum.ub.tum.de/1596550.