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

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 ..

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}
  by simp

lemma subtrees-split:
  set (subtrees (l@(a,b)#r)) = set (subtrees l) ∪ set (subtrees r) ∪ {a}
  by simp
```

```
lemma finite-set-ins-swap:
  assumes finite A
  shows max a (Max (Set.insert b A)) = max b (Max (Set.insert a A))
  using Max-insert assms max.commute max.left-commute by fastforce

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

using *Max-insert assms max.commute max.left-commute* **by** *fastforce*

lemma *height-Leaf*: *height t = 0 ⟷ t = Leaf*
by (*induction t*) (*auto*)

lemma *height-btree-order*:
height (*Node* (*ls*@[*a*]) *t*) = *height* (*Node* (*a*#*ls*) *t*)
by *simp*

lemma *height-btree-sub*:
height (*Node* ((*sub*,*x*)#*ls*) *t*) = *max* (*height* (*Node ls t*)) (*Suc* (*height sub*))
by *simp*

lemma *height-btree-last*:
height (*Node* ((*sub*,*x*)#*ts*) *t*) = *max* (*height* (*Node ts sub*)) (*Suc* (*height t*))
by (*induction ts*) *auto*

lemma *set-btree-inorder*: *set* (*inorder t*) = *set-btree t*
apply(*induction t*)
apply(*auto*)
done

lemma *child-subset*: *p* ∈ *set t* ⟹ *set-btree* (*fst p*) ⊆ *set-btree* (*Node t n*)
apply(*induction p arbitrary*: *t n*)
apply(*auto*)
done

lemma *some-child-sub*:
assumes (*sub*,*sep*) ∈ *set t*
shows *sub* ∈ *set* (*subtrees t*)
and *sep* ∈ *set* (*separators t*)
using *assms* **by** *force*+

lemma *bal-all-subtrees-equal*: *bal* (*Node ts t*) ⟹ (∀ *s1* ∈ *set* (*subtrees ts*). ∀ *s2* ∈ *set* (*subtrees ts*). *height s1* = *height s2*)
by (*metis BTree.bal.simps*(*2*))

lemma *fold-max-set*: ∀ *x* ∈ *set t*. *x* = *f* ⟹ *fold max t f* = *f*
apply(*induction t*)
apply(*auto simp add*: *max-def-raw*)
done

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

by (*induction ts*) *auto*

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*)
using *assms* **by** *auto*

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*)
using *assms* **by** (*auto simp add*: *image-constant-conv*)

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*)
using *assms* **by** (*auto simp add*: *image-constant-conv*)

lemma *bal-substitute*: ⟦*bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*); *height t* = *height c*; *bal c*⟧ ⟹ *bal* (*Node* (*ls*@(*c*,*b*)#*rs*) *t*)
unfolding *bal.simps*
by *auto*

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

lemma *bal-substitute-separator*: *bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*) ⟹ *bal* (*Node* (*ls*@(*a*,*c*)#*rs*) *t*)
unfolding *bal.simps*
by *auto*

lemma *order-impl-root-order*: ⟦*k* > *0*; *order k t*⟧ ⟹ *root-order k t*
apply(*cases t*)
apply(*auto*)
done

lemma *sorted-inorder-list-separators*: *sorted-less* (*inorder-list ts*) ⟹ *sorted-less* (*separators ts*)
apply(*induction ts*)
apply (*auto simp add*: *sorted-lems*)
done

corollary *sorted-inorder-separators*: *sorted-less* (*inorder* (*Node ts t*)) ⟹ *sorted-less* (*separators ts*)
using *sorted-inorder-list-separators sorted-wrt-append*
by *auto*

lemma *sorted-inorder-list-subtrees*:
sorted-less (*inorder-list ts*) ⟹ ∀ *sub* ∈ *set* (*subtrees ts*). *sorted-less* (*inorder sub*)
apply(*induction ts*)
apply (*auto simp add*: *sorted-lems*)+
done

corollary *sorted-inorder-subtrees*: *sorted-less* (*inorder* (*Node ts t*)) ⟹ ∀ *sub* ∈ *set* (*subtrees ts*). *sorted-less* (*inorder sub*)
using *sorted-inorder-list-subtrees sorted-wrt-append* **by** *auto*

lemma *sorted-inorder-list-induct-subtree*:
sorted-less (*inorder-list* (*ls*@(*sub*,*sep*)#*rs*)) ⟹ *sorted-less* (*inorder sub*)
by (*simp add*: *sorted-wrt-append*)

corollary *sorted-inorder-induct-subtree*:
sorted-less (*inorder* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)) ⟹ *sorted-less* (*inorder sub*)
by (*simp add*: *sorted-wrt-append*)

lemma *sorted-inorder-induct-last*: *sorted-less* (*inorder* (*Node ts t*)) ⟹ *sorted-less* (*inorder t*)
by (*simp add*: *sorted-wrt-append*)

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-list* (*replicate n c*) = *n*∗*c*
apply(*induction n*)
apply(*auto simp add*: *ring-class.ring-distribs*(*2*))
done

abbreviation *bound k h* ≡ ((*k*+*1*)^*h* − *1*)

lemma *nodes-height-upper-bound*:
⟦*order k t*; *bal t*⟧ ⟹ *nodes t* ∗ (*2*∗*k*) ≤ *bound* (*2*∗*k*) (*height t*)
proof(*induction t rule*: *nodes.induct*)
case (*2 ts t*)
let *?sub-height* = ((*2* ∗ *k* + *1*) ^ *height t* − *1*)
have *sum-list* (*map nodes* (*subtrees ts*)) ∗ (*2*∗*k*) =
sum-list (*map* (λ*t*. *nodes t* ∗ (*2* ∗ *k*)) (*subtrees ts*))
using *sum-list-mult-const* **by** *metis*
also have ... ≤ *sum-list* (*map* (λ*x*. *?sub-height*) (*subtrees ts*))
using *2*
using *sum-list-mono*[*of subtrees ts* λ*t*. *nodes t* ∗ (*2* ∗ *k*) λ*x*. *bound* (*2* ∗ *k*) (*height t*)]
by (*metis bal.simps*(*2*) *order.simps*(*2*))
also have ... = *sum-list* (*replicate* (*length ts*) *?sub-height*)
using *map-replicate-const*[*of ?sub-height subtrees ts*] *length-map*
by *simp*
also have ... = (*length ts*)∗(*?sub-height*)
using *sum-list-replicate* **by** *simp*
also have ... ≤ (*2*∗*k*)∗(*?sub-height*)
using *2.prems*(*1*)
by *simp*
finally have *sum-list* (*map nodes* (*subtrees ts*))∗(*2*∗*k*) ≤ *?sub-height*∗(*2*∗*k*)
by *simp*
moreover have (*nodes t*)∗(*2*∗*k*) ≤ *?sub-height*

```
    using 2 by simp
  ultimately have (nodes (Node ts t))*(2*k) ≤
        2*k
       + ?sub-height * (2*k)
       + ?sub-height
    unfolding nodes.simps add-mult-distrib
    by linarith
  also have ... =  2*k + (2*k)*((2 * k + 1) ^ height t) - 2*k + (2 * k + 1)
^ height t - 1
    by (simp add: diff-mult-distrib2 mult.assoc mult.commute)
  also have ... = (2*k)*((2 * k + 1) ^ height t) + (2 * k + 1) ^ height t - 1
    by simp
  also have ... = (2*k+1)^(Suc(height t)) - 1
    by simp
  finally show ?case
    by (metis 2.prems(2) height-bal-tree)
qed simp
```

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

```
fun full-node::nat ⇒ 'a ⇒ nat ⇒ 'a btree where
  full-node k c 0 = Leaf|
  full-node k c (Suc n) = (Node (replicate (2*k) ((full-node k c n),c)) (full-node k
c n))

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

lemma compow-comp-id: c > 0 ⟹ f ∘ f = f ⟹ (f ^^ c) = f
  apply(induction c)
   apply auto
  by fastforce


lemma compow-id-point: f x = x ⟹ (f ^^ c) x = x
  apply(induction c)
   apply auto
  done

lemma height-full-node: height (full-node k a h) = h
  apply(induction k a h rule: full-node.induct)
   apply (auto simp add: set-replicate-conv-if)
  done

lemma bal-full-node: bal (full-node k a h)
  apply(induction k a h rule: full-node.induct)
   apply auto
  done
```

lemma *order-full-node*: *order k* (*full-node k a h*)
apply(*induction k a h rule*: *full-node.induct*)
apply *auto*
done

lemma *full-btrees-sharp*: *nodes* (*full-node k a h*) ∗ (*2*∗*k*) = *bound* (*2*∗*k*) *h*
apply(*induction k a h rule*: *full-node.induct*)
apply (*auto simp add*: *height-full-node algebra-simps sum-list-replicate*)
done

lemma *upper-bound-sharp-node*:
t = *full-node k a h* ⟹ *height t* = *h* ∧ *order k t* ∧ *bal t* ∧ *bound* (*2*∗*k*) *h* = *nodes t* ∗ (*2*∗*k*)
by (*simp add*: *bal-full-node height-full-node order-full-node full-btrees-sharp*)

2.3 Maximum height for a given number of nodes

lemma *nodes-height-lower-bound*:
⟦*order k t*; *bal t*⟧ ⟹ *bound k* (*height t*) ≤ *nodes t* ∗ *k*
proof(*induction t rule*: *nodes.induct*)
case (*2 ts t*)
let *?sub-height* = ((*k* + *1*) ^ *height t* − *1*)
have *k*∗(*?sub-height*) ≤ (*length ts*)∗(*?sub-height*)
using *2.prems*(*1*)
by *simp*
also have ... = *sum-list* (*replicate* (*length ts*) *?sub-height*)
using *sum-list-replicate* **by** *simp*
also have ... = *sum-list* (*map* (λ*x*.*?sub-height*) (*subtrees ts*))
using *map-replicate-const*[*of ?sub-height subtrees ts*] *length-map*
by *simp*
also have ... ≤ *sum-list* (*map* (λ*t*. *nodes t* ∗ *k*) (*subtrees ts*))
using *2*
using *sum-list-mono*[*of subtrees ts* λ*x*. *bound k* (*height t*) λ*t*. *nodes t* ∗ *k*]
by (*metis bal.simps*(*2*) *order.simps*(*2*))
also have ... = *sum-list* (*map nodes* (*subtrees ts*)) ∗ *k*
using *sum-list-mult-const*[*of nodes k subtrees ts*] **by** *auto*
finally have *sum-list* (*map nodes* (*subtrees ts*))∗*k* ≥ *?sub-height*∗*k*
by *simp*
moreover have (*nodes t*)∗*k* ≥ *?sub-height*
using *2* **by** *simp*
ultimately have (*nodes* (*Node ts t*))∗*k* ≥
k
+ *?sub-height* ∗ *k*
+ *?sub-height*
unfolding *nodes.simps add-mult-distrib*
by *linarith*
also have
k + *?sub-height* ∗ *k* + *?sub-height* =

```
    k + k*((k + 1) ^ height t) - k + (k + 1) ^ height t - 1
   by (simp add: diff-mult-distrib2 mult.assoc mult.commute)
 also have ... = k*((k + 1) ^ height t) + (k + 1) ^ height t - 1
   by simp
 also have ... = (k+1)^(Suc(height t)) - 1
   by simp
 finally show ?case
   by (metis 2.prems(2) height-bal-tree)
qed simp
```

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
  apply(induction k a h rule: full-node.induct)
   apply (auto simp add: set-replicate-conv-if)
  done

lemma bal-slim-node: bal (slim-node k a h)
  apply(induction k a h rule: full-node.induct)
   apply auto
  done

lemma order-slim-node: order k (slim-node k a h)
  apply(induction k a h rule: full-node.induct)
   apply auto
  done

lemma slim-nodes-sharp: nodes (slim-node k a h) * k = bound k h
  apply(induction k a h rule: slim-node.induct)
    apply (auto simp add: height-slim-node algebra-simps sum-list-replicate com-
pow-id-point)
  done

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
  by (simp add: bal-slim-node height-slim-node order-slim-node slim-nodes-sharp)
```

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 $2*((k+1)\hat{}(height\ t\ -\ 1)\ -\ 1)\ +\ (of\text{-}bool\ (t \neq Leaf))*k\ \leq\ nodes\ t\ *\ k$
proof (*cases t*)
case (*Node ts t*)
let *?sub-height* = $((k\ +\ 1)\ \hat{}\ height\ t\ -\ 1)$
from *Node* **have** *?sub-height* $\leq$ *length ts* $*$ *?sub-height*
using *assms*
by (*simp add*: *Suc-leI*)
also have ... = *sum-list* (*replicate* (*length ts*) *?sub-height*)
using *sum-list-replicate*
by *simp*
also have ... = *sum-list* (*map* ($\lambda x.$ *?sub-height*) (*subtrees ts*))
using *map-replicate-const*[*of ?sub-height subtrees ts*] *length-map*
by *simp*
also have ... $\leq$ *sum-list* (*map* ($\lambda t.$ *nodes t* $*$ *k*) (*subtrees ts*))
using *Node*
sum-list-mono[*of subtrees ts* $\lambda x.$ $(k+1)\hat{}(height\ t)\ -\ 1$ $\lambda x.$ *nodes x* $*$ *k*]
nodes-height-lower-bound assms
by *fastforce*
also have ... = *sum-list* (*map nodes* (*subtrees ts*)) $*$ *k*
using *sum-list-mult-const*[*of nodes k subtrees ts*] **by** *simp*
finally have *sum-list* (*map nodes* (*subtrees ts*))$*k \geq$ *?sub-height*
by *simp*

moreover have (*nodes t*)$*k \geq$ *?sub-height*
using *Node assms nodes-height-lower-bound*
by *auto*
ultimately have (*nodes* (*Node ts t*))$*k \geq$
?sub-height
$+$ *?sub-height* $+$ *k*
unfolding *nodes.simps add-mult-distrib*
by *linarith*
then show *?thesis*
using *Node assms*(*2*) *height-bal-tree* **by** *fastforce*
qed *simp*

lemma *nodes-root-height-upper-bound*:
assumes *root-order k t*
and *bal t*
shows *nodes t* $*$ $(2*k) \leq (2*k+1)\hat{}(height\ t)\ -\ 1$
proof(*cases t*)
case (*Node ts t*)
let *?sub-height* = $((2\ *\ k\ +\ 1)\ \hat{}\ height\ t\ -\ 1)$
have *sum-list* (*map nodes* (*subtrees ts*)) $*$ $(2*k)$ =
sum-list (*map* ($\lambda t.$ *nodes t* $*$ $(2\ *\ k)$) (*subtrees ts*))
using *sum-list-mult-const* **by** *metis*
also have ... $\leq$ *sum-list* (*map* ($\lambda x.$*?sub-height*) (*subtrees ts*))

```
    using Node
      sum-list-mono[of subtrees ts λx. nodes x * (2*k)  λx. (2*k+1)^(height t) −
1]
      nodes-height-upper-bound assms
    by fastforce
  also have ... = sum-list (replicate (length ts) ?sub-height)
    using map-replicate-const[of ?sub-height subtrees ts] length-map
    by simp
  also have ... = (length ts)*(?sub-height)
    using sum-list-replicate by simp
  also have ... ≤ (2*k)*?sub-height
    using assms Node
    by simp
  finally have sum-list (map nodes (subtrees ts))*(2*k) ≤ ?sub-height*(2*k)
    by simp
  moreover have (nodes t)*(2*k) ≤ ?sub-height
    using Node assms nodes-height-upper-bound
    by auto
  ultimately have (nodes (Node ts t))*(2*k) ≤
        2*k
       + ?sub-height * (2*k)
       + ?sub-height
    unfolding nodes.simps add-mult-distrib
    by linarith
  also have ... =  2*k + (2*k)*((2 * k + 1) ^ height t) − 2*k + (2 * k + 1)
^ height t − 1
    by (simp add: diff-mult-distrib2 mult.assoc mult.commute)
  also have ... = (2*k)*((2 * k + 1) ^ height t) + (2 * k + 1) ^ height t − 1
    by simp
  also have ... = (2*k+1)^(Suc(height t)) − 1
    by simp
  finally show ?thesis
    by (metis Node assms(2) height-bal-tree)
qed simp

lemma root-order-imp-divmuleq: root-order k t ⟹ (nodes t * k) div k = nodes t
  using root-order.elims(2) by fastforce

lemma nodes-root-height-lower-bound-simp:
  assumes root-order k t
    and bal t
    and k > 0
  shows (2*((k+1)^(height t − 1) − 1)) div k + (of-bool (t ≠ Leaf)) ≤ nodes t
proof (cases t)
  case Node
  have (2*((k+1)^(height t − 1) − 1)) div k + (of-bool (t ≠ Leaf)) =
(2*((k+1)^(height t − 1) − 1) + (of-bool (t ≠ Leaf))*k) div k
    using Node assms
    using div-plus-div-distrib-dvd-left[of k k (2 * Suc k ^ (height t − Suc 0) − Suc
```

```
(Suc 0))]
    by (auto simp add: algebra-simps simp del: height-btree.simps)
  also have ... ≤ (nodes t * k) div k
    using nodes-root-height-lower-bound[OF assms(1,2)] div-le-mono
    by blast
  also have ... = nodes t
    using root-order-imp-divmuleq[OF assms(1)]
    by simp
  finally show ?thesis .
qed simp

lemma nodes-root-height-upper-bound-simp:
  assumes root-order k t
    and bal t
  shows nodes t ≤ ((2*k+1)^(height t) - 1) div (2*k)
proof -
  have nodes t = (nodes t * (2*k)) div (2*k)
    using root-order-imp-divmuleq[OF assms(1)]
    by simp
  also have ... ≤ ((2*k+1)^(height t) - 1) div (2*k)
    using div-le-mono nodes-root-height-upper-bound[OF assms] by blast
  finally show ?thesis .
qed

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 ⟹ t = slim-tree k a h ⟹ height t = h ∧ root-order k t ∧ bal t ∧ nodes
t * k = 2*((k+1)^(height t - 1) - 1) + (of-bool (t ≠ Leaf))*k
  apply (cases h)
  using slim-nodes-sharp[of k a]
  apply (auto simp add: algebra-simps bal-slim-node height-slim-node order-slim-node)
  done

lemma upper-bound-sharp:
  k > 0 ⟹ t = full-tree k a h ⟹ height t = h ∧ root-order k t ∧ bal t ∧
((2*k+1)^(height t) - 1) = nodes t * (2*k)
  unfolding full-tree-def
  using order-impl-root-order[of k t]
  by (simp add: bal-full-node height-full-node order-full-node full-btrees-sharp)


end
theory BTree-Set
  imports BTree
```

HOL−Data-Structures.Set-Specs
begin

3 Set interpretation

3.1 Auxiliary functions

fun *split-half*:: ('*a btree*×'*a*) *list* ⇒ (('*a btree*×'*a*) *list* × ('*a btree*×'*a*) *list*) **where**
split-half xs = (*take* (*length xs div 2*) *xs*, *drop* (*length xs div 2*) *xs*)

lemma *drop-not-empty*: *xs* ≠ [] ⟹ *drop* (*length xs div 2*) *xs* ≠ []
apply(*induction xs*)
apply(*auto split*!: *list.splits*)
done

lemma *split-half-not-empty*: *length xs* ≥ *1* ⟹ ∃ *ls sub sep rs*. *split-half xs* = (*ls*,(*sub*,*sep*)#*rs*)
using *drop-not-empty*
by (*metis* (*no-types*, *opaque-lifting*) *drop0 drop-eq-Nil eq-snd-iff hd-Cons-tl le-trans not-one-le-zero split-half.simps*)

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*)
using *split-conc*[*of ts y ls* (*sub*,*sep*)#*rs*] **by** *auto*

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-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
```

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

```
fun node_i:: nat ⇒ ('a btree × 'a) list ⇒ 'a btree ⇒ '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, (sub,sep)#rs) ⇒
      Up_i (Node ls sub) sep (Node rs t)
```

```
    )
  )

lemma nodei-ti-simp: node_i k ts t = T_i x ⟹ x = Node ts t
  apply (cases length ts ≤ 2*k)
   apply (auto split!: list.splits)
  done


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

fun insert::nat ⇒ 'a ⇒ 'a btree ⇒ 'a btree where
  insert k x t = tree_i (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 node_i k (mts@(mt,sep)#tts) tt of
       T_i u ⇒
        Node ls u |
       Up_i l a r ⇒
        Node (ls@[(l,a)]) r) |
    (Node rts rt,rsep)#rs ⇒ (
      case node_i k (mts@(mt,sep)#rts) rt of
      T_i u ⇒
        Node (ls@(u,rsep)#rs) (Node tts tt) |
      Up_i 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*
using *split-conc assms* **by** *fastforce+*

lemma *split-length*:
split ts x = (*ls*, *rs*) ⟹ *length ls* + *length rs* = *length ts*
by (*auto dest*: *split-conc*)

Isin proof

thm *isin-simps*

lemma *sorted-ConsD*: *sorted-less* (*y* # *xs*) ⟹ *x* ≤ *y* ⟹ *x* ∉ *set xs*
by (*auto simp*: *sorted-Cons-iff*)

lemma *sorted-snocD*: *sorted-less* (*xs* @ [*y*]) ⟹ *y* ≤ *x* ⟹ *x* ∉ *set xs*
by (*auto simp*: *sorted-snoc-iff*)

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*))
by (*auto simp*: *isin-simps2*)

lemma *isin-sorted-split*:
assumes *sorted-less* (*inorder* (*Node ts t*))
and *split ts x* = (*ls*, *rs*)
shows *x* ∈ *set* (*inorder* (*Node ts t*)) = (*x* ∈ *set* (*inorder-list rs* @ *inorder t*))
proof (*cases ls*)
case *Nil*
then have *ts* = *rs*
using *assms* **by** (*auto dest*!: *split-conc*)
then show *?thesis* **by** *simp*
next
case *Cons*
then obtain *ls*′ *sub sep* **where** *ls-tail-split*: *ls* = *ls*′ @ [(*sub*,*sep*)]
by (*metis list.simps*(*3*) *rev-exhaust surj-pair*)
then have *sep* < *x*
using *split-req*(*2*)[*of ts x ls*′ *sub sep rs*]

```
    using sorted-inorder-separators[OF assms(1)]
    using assms
    by simp
  then show ?thesis
    using assms(1) split-conc[OF assms(2)] ls-tail-split
    using isin-sorted[of inorder-list ls' @ inorder sub sep inorder-list rs @ inorder
t x]
    by auto
qed

lemma isin-sorted-split-right:
  assumes split ts x = (ls, (sub,sep)#rs)
    and sorted-less (inorder (Node ts t))
    and sep ≠ x
  shows x ∈ set (inorder-list ((sub,sep)#rs) @ inorder t) = (x ∈ set (inorder sub))
proof −
  from assms have x < sep
  proof −
    from assms have sorted-less (separators ts)
      by (simp add: sorted-inorder-separators)
    then show ?thesis
      using split-req(3)
      using assms
      by fastforce
  qed
  moreover have sorted-less (inorder-list ((sub,sep)#rs) @ inorder t)
    using assms sorted-wrt-append split-conc
    by fastforce
  ultimately show ?thesis
    using isin-sorted[of inorder sub sep inorder-list rs @ inorder t x]
    by simp
qed


theorem isin-set-inorder: sorted-less (inorder t) ⟹ isin t x = (x ∈ set (inorder
t))
proof(induction t x rule: isin.induct)
  case (2 ts t x)
  then obtain ls rs where list-split: split ts x = (ls, rs)
    by (meson surj-pair)
  then have list-conc: ts = ls @ rs
    using split-conc by auto
  show ?case
  proof (cases rs)
    case Nil
    then have isin (Node ts t) x = isin t x
      by (simp add: list-split)
    also have ... = (x ∈ set (inorder t))
      using 2.IH(1) list-split Nil
```

```
    using 2.prems sorted-inorder-induct-last by auto
  also have ... = (x ∈ set (inorder (Node ts t)))
    using isin-sorted-split[of ts t x ls rs]
    using 2.prems list-split list-conc Nil
    by simp
  finally show ?thesis .
 next
  case (Cons a list)
  then obtain sub sep where a-split: a = (sub,sep)
    by (cases a)
  then show ?thesis
  proof (cases x = sep)
    case True
    then show ?thesis
      using list-conc Cons a-split list-split
      by auto
  next
    case False
    then have isin (Node ts t) x = isin sub x
      using list-split Cons a-split False
      by auto
    also have ... = (x ∈ set (inorder sub))
      using 2.IH(2)
     using 2.prems False a-split list-conc list-split local.Cons sorted-inorder-induct-subtree
by fastforce
    also have ... = (x ∈ set (inorder (Node ts t)))
      using isin-sorted-split[OF 2.prems list-split]
      using isin-sorted-split-right 2.prems list-split Cons a-split False
      by simp
    finally show ?thesis .
  qed
 qed
qed auto
```

```
lemma node_i-cases: length xs ≤ k ∨ (∃ ls sub sep rs. split-half xs = (ls,(sub,sep)#rs))
proof −
 have ¬ length xs ≤ k ⟹ length xs ≥ 1
  by linarith
 then show ?thesis
  using split-half-not-empty
  by blast
qed
```

```
lemma root-order-tree_i: root-order-up_i (Suc k) t = root-order (Suc k) (tree_i t)
```

```
  apply (cases t)
   apply auto
  done

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 (cases length ts ≤ 2*k)
  case True
  then show ?thesis
    using assms
    by (simp add: node_i.simps)
next
  case False
  then obtain ls sub sep rs where split-half-ts:
    take (length ts div 2) ts = ls
    drop (length ts div 2) ts = (sub,sep)#rs
    using split-half-not-empty[of ts]
    by auto
  then have length-rs: length rs = length ts − (length ts div 2) − 1
    using length-drop
    by (metis One-nat-def add-diff-cancel-right' list.size(4))
  also have ... ≤ 4*k − ((4*k + 1) div 2)
    using assms(2) by simp
  also have ... = 2*k
    by auto
  finally have length rs ≤ 2*k
    by simp
  moreover have length rs ≥ k
    using False length-rs by simp
  moreover have set ((sub,sep)#rs) ⊆ set ts
    by (metis split-half-ts(2) set-drop-subset)
  ultimately have o-r: order k sub order k (Node rs t)
    using split-half-ts assms by auto
  moreover have length ls ≥ k
    using length-take assms split-half-ts False
    by auto
  moreover have  length ls ≤ 2*k
    using assms(2) split-half-ts
    by auto
  ultimately have o-l: order k (Node ls sub)
    using set-take-subset assms split-half-ts
    by fastforce
  from o-r o-l show ?thesis
    by (simp add: node_i.simps False split-half-ts)
qed
```

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 (*cases length ts* $\leq$ *2∗k*)
  case *True*
  then show *?thesis*
    using *assms*
    by (*simp add*: *node$_i$.simps*)
next
  case *False*
  then obtain *sub sep rs* **where**
    drop (*length ts div 2*) *ts* = (*sub*,*sep*)#*rs*
    using *split-half-not-empty*[*of ts*]
    by *auto*
  then show *?thesis*
    using *assms* **by** (*simp add*: *node$_i$.simps*)
qed

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*)

  apply(*cases node$_i$ k ts t*)
  using *node$_i$-root-order node$_i$-order-helper assms* **apply** *fastforce*
  apply (*metis node$_i$-root-order assms*(*2*,*3*,*4*) *le0 length-greater-0-conv*
    list.size(*3*) *node$_i$.simps order-up$_i$.simps*(*2*) *root-order-up$_i$.simps*(*2*) *up$_i$.distinct*(*1*))
  done

lemma *ins-order*:
  order k t $\Longrightarrow$ *order-up$_i$ k* (*ins k x t*)
proof(*induction k x t rule*: *ins.induct*)
  case (*2 k x ts t*)
  then obtain *ls rs* **where** *split-res*: *split ts x* = (*ls*, *rs*)
    by (*meson surj-pair*)
  then have *split-app*: *ls*@*rs* = *ts*
    using *split-conc*
    by *simp*

  show *?case*
  proof (*cases rs*)

```
    case Nil
    then have order-up_i k (ins k x t)
      using 2 split-res
      by simp
    then show ?thesis
      using Nil 2 split-app split-res Nil node_i-order
      by (auto split!: up_i.splits simp del: node_i.simps)
  next
    case (Cons a list)
    then obtain sub sep where a-prod: a  = (sub, sep)
      by (cases a)
    then show ?thesis
    proof (cases x = sep)
      case True
      then show ?thesis
        using 2 a-prod Cons split-res
        by simp
    next
      case False
      then have order-up_i k (ins k x sub)
        using 2.IH(2) 2.prems a-prod local.Cons split-app split-res by auto
      then show ?thesis
        using 2 split-app Cons length-append node_i-order a-prod split-res
       by (auto split!: up_i.splits simp del: node_i.simps simp add: order-impl-root-order)
    qed
  qed
qed simp



lemma ins-root-order:
  assumes root-order k t
  shows root-order-up_i k (ins k x t)
proof(cases t)
  case (Node ts t)
  then obtain ls rs where split-res: split ts x = (ls, rs)
    by (meson surj-pair)
  then have split-app: ls@rs = ts
    using split-conc
    by fastforce

  show ?thesis
  proof (cases rs)
    case Nil
    then have order-up_i k (ins k x t) using Node assms split-res
      by (simp add: ins-order)
    then show ?thesis
      using Nil Node split-app split-res assms node_i-root-order
     by (auto split!: up_i.splits simp del: node_i.simps simp add: order-impl-root-order)
```

```
  next
    case (Cons a list)
    then obtain sub sep where a-prod: a  = (sub, sep)
      by (cases a)
    then show ?thesis
    proof (cases x = sep)
      case True
      then show ?thesis using assms Node a-prod Cons split-res
        by simp
    next
      case False
      then have order-up_i k (ins k x sub)
        using Node a-prod assms ins-order local.Cons split-app by auto
      then show ?thesis
          using assms split-app Cons length-append Node node_i-root-order a-prod
split-res
      by (auto split!: up_i.splits simp del: node_i.simps simp add: order-impl-root-order)
    qed
  qed
qed simp



lemma height-list-split: height-up_i (Up_i (Node ls a) b (Node rs t)) = height (Node
(ls@(a,b)#rs) t)
  by (induction ls) (auto simp add: max.commute)

lemma node_i-height: height-up_i (node_i k ts t) = height (Node ts t)
proof(cases length ts ≤ 2∗k)
  case False
  then obtain ls sub sep rs where
    split-half-ts: split-half ts = (ls, (sub, sep) # rs)
    by (meson node_i-cases)
  then have node_i k ts t = Up_i (Node ls (sub)) sep (Node rs t)
    using False by simp
  then show ?thesis
    using split-half-ts
   by (metis append-take-drop-id fst-conv height-list-split snd-conv split-half.elims)
qed simp



lemma bal-up_i-tree: bal-up_i t = bal (tree_i t)
  apply(cases t)
   apply auto
  done

lemma bal-list-split: bal (Node (ls@(a,b)#rs) t) ⟹ bal-up_i (Up_i (Node ls a) b
(Node rs t))
```

by (*auto simp add*: *image-constant-conv*)

lemma *node$_i$-bal*:
assumes *bal* (*Node ts t*)
shows *bal-up$_i$* (*node$_i$ k ts t*)
using *assms*
proof(*cases length ts* $\leq$ *2*$*$*k*)
case *False*
then obtain *ls sub sep rs* **where**
split-half-ts: *split-half ts* = (*ls*, (*sub*, *sep*) # *rs*)
by (*meson node$_i$-cases*)
then have *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
using *assms append-take-drop-id*[**where** *n*=*length ts div 2* **and** *xs*=*ts*]
by *auto*
then show *?thesis*
using *split-half-ts assms False*
by (*auto simp del*: *bal.simps bal-up$_i$.simps dest*!: *bal-list-split*[*of ls sub sep rs t*])
qed *simp*

lemma *height-up$_i$-merge*: *height-up$_i$* (*Up$_i$ l a r*) = *height t* $\Longrightarrow$ *height* (*Node* (*ls*@(*t*,*x*)#*rs*) *tt*) = *height* (*Node* (*ls*@(*l*,*a*)#(*r*,*x*)#*rs*) *tt*)
by *simp*

lemma *ins-height*: *height-up$_i$* (*ins k x t*) = *height t*
proof(*induction k x t rule*: *ins.induct*)
case (*2 k x ts t*)
then obtain *ls rs* **where** *split-list*: *split ts x* = (*ls*,*rs*)
by (*meson surj-pair*)
then have *split-append*: *ls*@*rs* = *ts*
using *split-conc*
by *auto*
then show *?case*
proof (*cases rs*)
case *Nil*
then have *height-sub*: *height-up$_i$* (*ins k x t*) = *height t*
using *2* **by** (*simp add*: *split-list*)
then show *?thesis*
proof (*cases ins k x t*)
case (*T$_i$ a*)
then have *height* (*Node ts t*) = *height* (*Node ts a*)
using *height-sub*
by *simp*
then show *?thesis*
using *T$_i$ Nil split-list split-append*
by *simp*
next
case (*Up$_i$ l a r*)
then have *height* (*Node ls t*) = *height* (*Node* (*ls*@[(*l*,*a*)]) *r*)
using *height-btree-order height-sub* **by** (*induction ls*) *auto*

```
      then show ?thesis using 2 Nil split-list Up_i split-append
        by (simp del: node_i.simps add: node_i-height)
    qed
  next
    case (Cons a list)
    then obtain sub sep where a-split: a = (sub,sep)
      by (cases a)
    then show ?thesis
    proof (cases x = sep)
      case True
      then show ?thesis
        using Cons a-split 2 split-list
        by (simp del: height-btree.simps)
    next
      case False
      then have height-sub: height-up_i (ins k x sub) = height sub
        by (metis 2.IH(2) a-split Cons split-list)
      then show ?thesis
      proof (cases ins k x sub)
        case (T_i a)
        then have height a = height sub
          using height-sub by auto
      then have height (Node (ls@(sub,sep)#rs) t) = height (Node (ls@(a,sep)#rs)
t)
          by auto
        then show ?thesis
          using T_i height-sub False Cons 2 split-list a-split split-append
          by (auto simp add: image-Un max.commute finite-set-ins-swap)
      next
        case (Up_i l a r)
      then have height (Node (ls@(sub,sep)#list) t) = height (Node (ls@(l,a)#(r,sep)#list)
t)
          using height-up_i-merge height-sub
          by fastforce
        then show ?thesis
          using Up_i False Cons 2 split-list a-split split-append
       by (auto simp del: node_i.simps simp add: node_i-height image-Un max.commute
finite-set-ins-swap)
      qed
    qed
  qed
qed simp



lemma ins-bal: bal t ⟹ bal-up_i (ins k x t)
proof(induction k x t rule: ins.induct)
  case (2 k x ts t)
  then obtain ls rs where split-res: split ts x = (ls, rs)
```

```
    by (meson surj-pair)
  then have split-app: ls@rs = ts
    using split-conc
    by fastforce

  show ?case
  proof (cases rs)
    case Nil
    then show ?thesis
    proof (cases ins k x t)
      case (T_i a)
      then have bal (Node ls a) unfolding bal.simps
          by (metis 2.IH(1) 2.prems append-Nil2 bal.simps(2) bal-up_i.simps(1)
height-up_i.simps(1) ins-height local.Nil split-app split-res)
      then show ?thesis
        using Nil T_i 2 split-res
        by simp
    next
      case (Up_i l a r)
      then have
        (∀ x∈set (subtrees (ls@[(l,a)])). bal x)
        (∀ x∈set (subtrees ls). height r = height x)
        using 2 Up_i Nil split-res split-app
        by simp-all (metis height-up_i.simps(2) ins-height max-def)
      then show ?thesis unfolding ins.simps
        using Up_i Nil 2 split-res
        by (simp del: node_i.simps add: node_i-bal)
    qed
  next
    case (Cons a list)
    then obtain sub sep where a-prod: a  = (sub, sep) by (cases a)
    then show ?thesis
    proof (cases x = sep)
      case True
      then show ?thesis
        using a-prod 2 split-res Cons by simp
    next
      case False
      then have bal-up_i (ins k x sub) using 2 split-res
        using a-prod local.Cons split-app by auto
      show ?thesis
      proof (cases ins k x sub)
        case (T_i x1)
        then have  height x1 = height t
       by (metis 2.prems a-prod add-diff-cancel-left' bal-split-left(1) bal-split-left(2)
height-bal-tree height-up_i.simps(1) ins-height local.Cons plus-1-eq-Suc split-app)
        then show ?thesis
          using split-app Cons T_i 2 split-res a-prod
          by auto
```

```
      next
        case (Up_i l a r)

        then have
          ∀x ∈ set (subtrees (ls@(l,a)#(r,sep)#list)). bal x
          using Up_i split-app Cons 2 ‹bal-up_i (ins k x sub)› by auto
          moreover have ∀x ∈ set (subtrees (ls@(l,a)#(r,sep)#list)). height x =
height t
         using False Up_i split-app Cons 2 ‹bal-up_i (ins k x sub)› ins-height split-res
a-prod
          apply auto
          by (metis height-up_i.simps(2) sup.idem sup-nat-def)
        ultimately show ?thesis using Up_i Cons 2 split-res a-prod
          by (simp del: node_i.simps add: node_i-bal)
      qed
    qed
  qed
qed simp




lemma node_i-inorder: inorder-up_i (node_i k ts t) = inorder (Node ts t)
  apply(cases length ts ≤ 2*k)
   apply (auto split!: list.splits)

  supply R = sym[OF append-take-drop-id, of map - ts (length ts div 2)]
  thm R
  apply(subst R)
  apply (simp del: append-take-drop-id add: take-map drop-map)
  done

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)
   apply (metis inorder-up_i.simps(1) node_i-inorder)
  by (metis append-Cons inorder-up_i.simps(2) node_i-inorder self-append-conv2)


lemma ins-sorted-inorder: sorted-less (inorder t) ⟹ (inorder-up_i (ins k (x::('a::linorder))
t)) = ins-list x (inorder t)
  apply(induction k x t rule: ins.induct)
   using split-axioms apply (auto split!: prod.splits list.splits up_i.splits simp del:
node_i.simps
      simp add:  node_i-inorder node_i-inorder-simps)

  oops
```

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 (*cases ls*)
case *Nil*
then show *?thesis*
using *assms* **by** (*auto dest*!: *split-conc*)
next
case *Cons*
then obtain *ls′ sub sep* **where** *ls-tail-split*: *ls* = *ls′* @ [(*sub*,*sep*)]
by (*metis list.distinct*(*1*) *rev-exhaust surj-pair*)
moreover have *sep* < *x*
using *split-req*(*2*)[*of ts x ls′ sub sep rs*]
using *sorted-inorder-separators*
using *assms*(*1*) *assms*(*2*) *ls-tail-split*
by *auto*
moreover have *sorted-less* (*inorder-list ls*)
using *assms sorted-wrt-append split-conc* **by** *fastforce*
ultimately show *?thesis* **using** *assms*(*2*) *split-conc*[*OF assms*(*1*)]
using *ins-list-sorted*[*of inorder-list ls′* @ *inorder sub sep*]
by *auto*
qed

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 −
from *assms* **have** *x* < *sep*
proof −
from *assms* **have** *sorted-less* (*separators ts*)
by (*simp add*: *sorted-inorder-list-separators*)
then show *?thesis*
using *split-req*(*3*)
using *assms*
by *fastforce*
qed
moreover have *sorted-less* (*inorder-pair* (*sub*,*sep*))
by (*metis* (*no-types*, *lifting*) *assms*(*1*) *assms*(*2*) *concat.simps*(*2*) *concat-append list.simps*(*9*) *map-append sorted-wrt-append split-conc*)
ultimately show *?thesis*
using *ins-list-sorted*[*of inorder sub sep*]
by *auto*

qed

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*
using *assms sorted-wrt-append split.ins-list-split-right-general split-axioms* **by** *fastforce*

lemma *ins-list-idem-eq-isin*: *sorted-less xs* ⟹ *x* ∈ *set xs* ⟷ (*ins-list x xs* = *xs*)
apply(*induction xs*)
apply *auto*
done

lemma *ins-list-contains-idem*: ⟦*sorted-less xs*; *x* ∈ *set xs*⟧ ⟹ (*ins-list x xs* = *xs*)
using *ins-list-idem-eq-isin* **by** *auto*

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(*induction k x t rule*: *ins.induct*)
case (*1 k x*)
then show *?case* **by** *auto*
next
case (*2 k x ts t*)
then obtain *ls rs* **where** *list-split*: *split ts x* = (*ls*,*rs*)
by (*cases split ts x*)
then have *list-conc*: *ts* = *ls*@*rs*
using *split.split-conc split-axioms* **by** *blast*
then show *?case*
proof (*cases rs*)
case *Nil*
then show *?thesis*
proof (*cases ins k x t*)
case (*T*$_i$ *a*)
then have *IH*:*inorder a* = *ins-list x* (*inorder t*)
using *2.IH*(*1*) *2.prems list-split local.Nil sorted-inorder-induct-last*
by *auto*

have *inorder-up*$_i$ (*ins k x* (*Node ts t*)) = *inorder-list ls* @ *inorder a*
using *list-split T*$_i$ *Nil* **by** (*auto simp add*: *list-conc*)

```
      also have ... = inorder-list ls @ (ins-list x (inorder t))
        by (simp add: IH)
      also have ... = ins-list x (inorder (Node ts t))
        using ins-list-split
        using 2.prems list-split Nil by auto
      finally show ?thesis .
    next
      case (Up_i l a r)
      then have IH:inorder-up_i (Up_i l a r) = ins-list x (inorder t)
       using 2.IH(1) 2.prems list-split local.Nil sorted-inorder-induct-last by auto

      have inorder-up_i (ins k x (Node ts t)) = inorder-list ls @ inorder-up_i (Up_i l
a r)
        using list-split Up_i Nil by (auto simp add: list-conc)
      also have ... = inorder-list ls @ ins-list x (inorder t)
        using IH by simp
      also have ... = ins-list x (inorder (Node ts t))
        using ins-list-split
        using 2.prems list-split local.Nil by auto
      finally show ?thesis .
    qed
  next
    case (Cons h list)
    then obtain sub sep where h-split: h = (sub,sep)
      by (cases h)

    then have sorted-inorder-sub: sorted-less (inorder sub)
      using 2.prems list-conc local.Cons sorted-inorder-induct-subtree
      by fastforce
    then show ?thesis
    proof(cases x = sep)
      case True
      then have x ∈ set (inorder (Node ts t))
        using list-conc h-split Cons by simp
      then have ins-list x (inorder (Node ts t)) = inorder (Node ts t)
        using 2.prems ins-list-contains-idem by blast
      also have ... = inorder-up_i (ins k x (Node ts t))
        using list-split h-split Cons True by auto
      finally show ?thesis by simp
    next
      case False
      then show ?thesis
      proof (cases ins k x sub)
        case (T_i a)
        then have IH:inorder a = ins-list x (inorder sub)
          using 2.IH(2) 2.prems list-split Cons sorted-inorder-sub h-split False
          by auto
        have inorder-up_i (ins k x (Node ts t)) = inorder-list ls @ inorder a @ sep
# inorder-list list @ inorder t
```

```
        using h-split False list-split T_i Cons by simp
      also have ... = inorder-list ls @ ins-list x (inorder sub) @ sep # inorder-list
list @ inorder t
        using IH by simp
       also have ... = ins-list x (inorder (Node ts t))
        using ins-list-split ins-list-split-right
        using list-split 2.prems Cons h-split False by auto
       finally show ?thesis .
     next
       case (Up_i l a r)
       then have IH:inorder-up_i (Up_i l a r) = ins-list x (inorder sub)
        using 2.IH(2) False h-split list-split local.Cons sorted-inorder-sub
        by auto
       have inorder-up_i (ins k x (Node ts t)) = inorder-list ls @ inorder l @ a #
inorder r  @ sep # inorder-list list @ inorder t
        using h-split False list-split Up_i Cons by simp
      also have ... = inorder-list ls @ ins-list x (inorder sub) @ sep # inorder-list
list @ inorder t
        using IH by simp
       also have ... = ins-list x (inorder (Node ts t))
        using ins-list-split ins-list-split-right
        using list-split 2.prems Cons h-split False by auto
       finally show ?thesis .
     qed
   qed
  qed
qed

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)
  apply(cases u)
   apply(auto)
  done

lemma tree_i-order: ⟦k > 0; root-order-up_i k u⟧ ⟹ root-order k (tree_i u)
  apply(cases u)
   apply(auto simp add: order-impl-root-order)
  done

lemma tree_i-inorder: inorder-up_i u = inorder (tree_i u)
  apply (cases u)
```

apply *auto*
done

lemma *insert-bal*: *bal t* $\Longrightarrow$ *bal* (*insert k x t*)
using *ins-bal*
by (*simp add*: *tree$_i$-bal*)

lemma *insert-order*: $[\![k > 0; \textit{root-order } k\ t]\!] \Longrightarrow$ *root-order k* (*insert k x t*)
using *ins-root-order*
by (*simp add*: *tree$_i$-order*)

lemma *insert-inorder*: *sorted-less* (*inorder t*) $\Longrightarrow$ *inorder* (*insert k x t*) = *ins-list x* (*inorder t*)
using *ins-inorder*
by (*simp add*: *tree$_i$-inorder*)

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 (*cases height t*)
case *0*
then have *t* = *Leaf sub* = *Leaf* **using** *height-Leaf assms* **by** *auto*
then show *?thesis* **by** *simp*
next
case (*Suc nat*)
then obtain *tts tt* **where** *t-node*: *t* = *Node tts tt*
using *height-Leaf* **by** (*cases t*) *simp*
then obtain *mts mt* **where** *sub-node*: *sub* = *Node mts mt*
using *assms* **by** (*cases sub*) *simp*
then show *?thesis*
proof (*cases length mts* ≥ *k* ∧ *length tts* ≥ *k*)
case *False*
then show *?thesis*
proof (*cases rs*)
case *Nil*
then have *height-up$_i$* (*node$_i$ k* (*mts*@(*mt*,*sep*)#*tts*) *tt*) = *height* (*Node* (*mts*@(*mt*,*sep*)#*tts*) *tt*)
using *node$_i$-height* **by** *blast*
also have ... = *max* (*height t*) (*height sub*)
by (*metis assms*(*1*) *height-up$_i$.simps*(*2*) *height-list-split sub-node t-node*)
finally have *height-max*: *height-up$_i$* (*node$_i$ k* (*mts* @ (*mt*, *sep*) # *tts*) *tt*) =

max (*height t*) (*height sub*) **by** *simp*
then show *?thesis*
proof (*cases node*$_i$ *k* (*mts*@(*mt*,*sep*)#*tts*) *tt*)
case (*T*$_i$ *u*)
then have *height u* = *max* (*height t*) (*height sub*) **using** *height-max* **by** *simp*
then have *height* (*Node ls u*) = *height* (*Node* (*ls*@[(*sub*,*sep*)]) *t*)
by (*induction ls*) (*auto simp add*: *max.commute*)
then show *?thesis* **using** *Nil False T*$_i$
by (*simp add*: *sub-node t-node*)
next
case (*Up*$_i$ *l a r*)
then have *height* (*Node* (*ls*@[(*sub*,*sep*)]) *t*) = *height* (*Node* (*ls*@[(*l*,*a*)]) *r*)
using *assms*(*1*) *height-max* **by** (*induction ls*) *auto*
then show *?thesis*
using *Up*$_i$ *Nil sub-node t-node* **by** *auto*
qed
next
case (*Cons a list*)
then obtain *rsub rsep* **where** *a-split*: *a* = (*rsub*, *rsep*)
by (*cases a*)
then obtain *rts rt* **where** *r-node*: *rsub* = *Node rts rt*
using *assms*(*2*) *Cons height-Leaf Suc* **by** (*cases rsub*) *simp-all*

then have *height-up*$_i$ (*node*$_i$ *k* (*mts*@(*mt*,*sep*)#*rts*) *rt*) = *height* (*Node* (*mts*@(*mt*,*sep*)#*rts*) *rt*)
using *node*$_i$*-height* **by** *blast*
also have ... = *max* (*height rsub*) (*height sub*)
by (*metis r-node height-up*$_i$*.simps*(*2*) *height-list-split max.commute sub-node*)
finally have *height-max*: *height-up*$_i$ (*node*$_i$ *k* (*mts* @ (*mt*, *sep*) # *rts*) *rt*) = *max* (*height rsub*) (*height sub*) **by** *simp*
then show *?thesis*
proof (*cases node*$_i$ *k* (*mts*@(*mt*,*sep*)#*rts*) *rt*)
case (*T*$_i$ *u*)
then have *height u* = *max* (*height rsub*) (*height sub*)
using *height-max* **by** *simp*
then show *?thesis*
using *T*$_i$ *False Cons r-node a-split sub-node t-node* **by** *auto*
next
case (*Up*$_i$ *l a r*)
then have *height-max*: *max* (*height l*) (*height r*) = *max* (*height rsub*) (*height sub*)
using *height-max* **by** *auto*
then show *?thesis*
using *Cons a-split r-node Up*$_i$ *sub-node t-node* **by** *auto*
qed
qed
qed (*simp add*: *sub-node t-node*)
qed

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)*
using *rebalance-middle-tree-height assms* **by** *auto*

lemma *split-max-height*:
assumes *split-max k t = (sub,sep)*
and *nonempty-lasttreebal t*
and *t ≠ Leaf*
shows *height sub = height t*
using *assms*
proof(*induction t arbitrary*: *k sub sep*)
case *Node1*: (*Node tts tt*)
then obtain *ls tsub tsep* **where** *tts-split*: *tts = ls@[(tsub,tsep)]* **by** *auto*
then show *?case*
proof (*cases tt*)
case *Leaf*
then have *height (Node (ls@[(tsub,tsep)]) tt) = max (height (Node ls tsub))*
(Suc (height tt))
using *height-btree-last height-btree-order* **by** *metis*
moreover have *split-max k (Node tts tt) = (Node ls tsub, tsep)*
using *Leaf Node1 tts-split* **by** *auto*
ultimately show *?thesis*
using *Leaf Node1 height-Leaf max-def* **by** *auto*
next
case *Node2*: (*Node l a*)
then obtain *subsub subsep* **where** *sub-split*: *split-max k tt = (subsub,subsep)*
by (*cases split-max k tt*)
then have *height subsub = height tt* **using** *Node1 Node2* **by** *auto*
moreover have *split-max k (Node tts tt) = (rebalance-last-tree k tts subsub,*
subsep)
using *Node1 Node2 tts-split sub-split* **by** *auto*
ultimately show *?thesis* **using** *rebalance-last-tree-height Node1 Node2* **by** *auto*
qed
qed *auto*

lemma *order-bal-nonempty-lasttreebal*: ⟦*k > 0*; *root-order k t*; *bal t*⟧ ⟹ *nonempty-lasttreebal t*
proof(*induction k t rule*: *order.induct*)
case (*2 k ts t*)
then have *length ts > 0* **by** *auto*
then obtain *ls tsub tsep* **where** *ts-split*: *ts = (ls@[(tsub,tsep)])*
by (*metis eq-fst-iff length-greater-0-conv snoc-eq-iff-butlast*)
moreover have *height tsub = height t*
using *2.prems*(*3*) *ts-split* **by** *auto*
moreover have *nonempty-lasttreebal t* **using** *2 order-impl-root-order* **by** *auto*

ultimately show *?case* **by** *simp*
qed *simp*

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

lemma *del-height*: ⟦*k* > *0*; *root-order k t*; *bal t*⟧ ⟹ *height* (*del k x t*) = *height t*
proof(*induction k x t rule*: *del.induct*)
case (*2 k x ts t*)
then obtain *ls list* **where** *list-split*: *split ts x* = (*ls*, *list*) **by** (*cases split ts x*)
then show *?case*
proof(*cases list*)
case *Nil*
then have *height* (*del k x t*) = *height t*
using *2 list-split order-bal-nonempty-lasttreebal*
by (*simp add*: *order-impl-root-order*)
moreover obtain *lls sub sep* **where** *ls* = *lls*@[(*sub*,*sep*)]
using *split-conc 2 list-split Nil*
by (*metis append-Nil2 nonempty-lasttreebal.simps*(*2*) *order-bal-nonempty-lasttreebal*)
moreover have *Node ls t* = *Node ts t* **using** *split-conc Nil list-split* **by** *auto*
ultimately show *?thesis*
using *rebalance-last-tree-height 2 list-split Nil split-conc*
by (*auto simp add*: *max.assoc sup-nat-def max-def*)
next
case (*Cons a rs*)
then have *rs-height*: *case rs of* [] ⇒ *True* | (*rsub*,*rsep*)#- ⇒ *height rsub* = *height t*
using *2.prems*(*3*) *bal-sub-height list-split split-conc* **by** *blast*
from *Cons* **obtain** *sub sep* **where** *a-split*: *a* = (*sub*,*sep*) **by** (*cases a*)
consider (*sep-n-x*) *sep* ≠ *x* |
(*sep-x-Leaf*) *sep* = *x* ∧ *sub* = *Leaf* |
(*sep-x-Node*) *sep* = *x* ∧ (∃ *ts t*. *sub* = *Node ts t*)
using *btree.exhaust* **by** *blast*
then show *?thesis*
proof *cases*
case *sep-n-x*
have *height-t-sub*: *height t* = *height sub*
using *2.prems*(*3*) *a-split list-split local.Cons split.split-set*(*1*) *split-axioms* **by** *fastforce*
have *height-t-del*: *height* (*del k x sub*) = *height t*
by (*metis 2.IH*(*2*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *a-split bal.simps*(*2*) *list-split local.Cons order-impl-root-order root-order.simps*(*2*) *sep-n-x some-child-sub*(*1*) *split-set*(*1*))
then have *height* (*rebalance-middle-tree k ls* (*del k x sub*) *sep rs t*) = *height* (*Node* (*ls*@((*del k x sub*),*sep*)#*rs*) *t*)
using *rs-height rebalance-middle-tree-height* **by** *simp*
also have ... = *height* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
using *height-t-sub 2.prems height-t-del*

```
      by auto
    also have ... = height (Node ts t)
      using 2 a-split sep-n-x list-split Cons split-set(1) split-conc
      by auto
    finally show ?thesis
      using sep-n-x Cons a-split list-split 2
      by simp
  next
    case sep-x-Leaf
    then have height (Node ts t) = height (Node (ls@rs) t)
      using bal-split-last(2) 2.prems(3) a-split list-split Cons split-conc
      by metis
    then show ?thesis
      using a-split list-split Cons sep-x-Leaf 2 by auto
  next
    case sep-x-Node
    then obtain sts st where sub-node: sub = Node sts st by blast
    obtain sub-s max-s where sub-split: split-max k sub = (sub-s, max-s)
      by (cases split-max k sub)
    then have height sub-s = height t
    by (metis 2.prems(1) 2.prems(2) 2.prems(3) a-split bal.simps(2) btree.distinct(1)
list-split Cons order-bal-nonempty-lasttreebal order-impl-root-order root-order.simps(2)
some-child-sub(1) split-set(1) split-max-height sub-node)
   then have height (rebalance-middle-tree k ls sub-s max-s rs t) = height (Node
(ls@(sub-s,sep)#rs) t)
      using rs-height rebalance-middle-tree-height by simp
    also have ... = height (Node ts t)
      using 2 a-split sep-x-Node list-split Cons split-set(1) ‹height sub-s = height
t›
      by (auto simp add: split-conc[of ts])
      finally show ?thesis using sep-x-Node Cons a-split list-split 2 sub-node
sub-split
      by auto
  qed
 qed
qed simp
```

```
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)
 apply(cases sub; cases t)
 using assms
   apply (auto
    split!: btree.splits up_i.splits list.splits
```

```
    simp del: node_i.simps
    simp add: node_i-inorder-simps
    )
  done

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)
  using rebalance-middle-tree-inorder assms by auto

lemma butlast-inorder-app-id: xs = xs' @ [(sub,sep)] ⟹ inorder-list xs' @ inorder
sub @ [sep] = inorder-list xs
  by simp

lemma split-max-inorder:
  assumes nonempty-lasttreebal t
    and t ≠ Leaf
  shows inorder-pair (split-max k t) = inorder t
  using assms
proof (induction k t rule: split-max.induct)
  case (1 k ts t)
  then show ?case
  proof (cases t)
    case Leaf
    then have ts = butlast ts @ [last ts]
      using 1.prems(1) by auto
    moreover obtain sub sep where last ts = (sub,sep)
      by fastforce
    ultimately show ?thesis
      using Leaf
      apply (auto split!: prod.splits btree.splits)
      by (simp add: butlast-inorder-app-id)
  next
    case (Node tts tt)
    then have IH: inorder-pair (split-max k t) = inorder t
      using 1.IH 1.prems(1) by auto
    obtain sub sep where split-sub-sep: split-max k t = (sub,sep)
      by fastforce
    then have height-sub: height sub = height t
       by (metis 1.prems(1) Node btree.distinct(1) nonempty-lasttreebal.simps(2)
split-max-height)
    have inorder-pair (split-max k (Node ts t)) = inorder (rebalance-last-tree k ts
sub) @ [sep]
      using Node 1 split-sub-sep by auto
    also have ... = inorder-list ts @ inorder sub @ [sep]
      using rebalance-last-tree-inorder height-sub 1.prems
      by (auto simp del: rebalance-last-tree.simps)
    also have ... = inorder (Node ts t)
```

using *IH split-sub-sep* **by** *simp*
finally show *?thesis* **.**
qed
qed *simp*

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*
by (*metis Suc-inject Un-iff bal.simps*(*2*) *height-bal-tree singletonD*)

lemma *bal-list-merge*:
assumes *bal-up*$_i$ (*Up*$_i$ (*Node as a*) *x* (*Node bs b*))
shows *bal* (*Node* (*as*@(*a*,*x*)#*bs*) *b*)
proof −
have ∀ *x*∈*set* (*subtrees* (*as* @ (*a*, *x*) # *bs*)). *bal x*
using *subtrees-split assms* **by** *auto*
moreover have *bal b*
using *assms* **by** *auto*
moreover have ∀ *x*∈*set* (*subtrees as*) ∪ {*a*} ∪ *set* (*subtrees bs*). *height x* = *height b*
using *assms height-bal-subtrees-merge*
unfolding *bal-up*$_i$*.simps*
by *blast*
ultimately show *?thesis*
by *auto*
qed

lemma *node*$_i$*-bal-up*$_i$:
assumes *bal-up*$_i$ (*node*$_i$ *k ts t*)
shows *bal* (*Node ts t*)
using *assms*
proof(*cases length ts* ≤ *2*∗*k*)
case *False*
then obtain *ls sub sep rs* **where** *split-list*: *split-half ts* = (*ls*, (*sub*,*sep*)#*rs*)
using *node*$_i$*-cases* **by** *blast*
then have *node*$_i$ *k ts t* = *Up*$_i$ (*Node ls sub*) *sep* (*Node rs t*)
using *False* **by** *auto*
moreover have *ts* = *ls*@(*sub*,*sep*)#*rs*
by (*metis append-take-drop-id fst-conv local.split-list snd-conv split-half.elims*)
ultimately show *?thesis*
using *bal-list-merge*[*of ls sub sep rs t*] *assms*
by (*simp del*: *bal.simps bal-up*$_i$*.simps*)
qed *simp*

lemma *node*$_i$*-bal-simp*: *bal-up*$_i$ (*node*$_i$ *k ts t*) = *bal* (*Node ts t*)
using *node*$_i$*-bal node*$_i$*-bal-up*$_i$ **by** *blast*

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

k ls sub sep rs t)
proof (*cases t*)
case *t-node*: (*Node tts tt*)
assume *assms*: *bal* (*Node* (*ls* @ (*sub*, *sep*) # *rs*) *t*)
then obtain *mts mt* **where** *sub-node*: *sub* = *Node mts mt*
by (*cases sub*) (*auto simp add*: *t-node*)
have *sub-heights*: *height sub* = *height t bal sub bal t*
using *assms* **by** *auto*
show *?thesis*
proof (*cases length mts* ≥ *k* ∧ *length tts* ≥ *k*)
case *True*
then show *?thesis*
using *t-node sub-node assms*
by (*auto simp del*: *bal.simps*)
next
case *False*
then show *?thesis*
proof (*cases rs*)
case *Nil*
have *height-up*$_i$ (*node*$_i$ *k* (*mts*@(*mt*,*sep*)#*tts*) *tt*) = *height* (*Node* (*mts*@(*mt*,*sep*)#*tts*) *tt*)
using *node*$_i$*-height* **by** *blast*
also have ... = *Suc* (*height tt*)
by (*metis height-bal-tree height-up*$_i$*.simps*(*2*) *height-list-split max.idem sub-heights*(*1*) *sub-heights*(*3*) *sub-node t-node*)
also have ... = *height t*
using *height-bal-tree sub-heights*(*3*) *t-node* **by** *fastforce*
finally have *height-up*$_i$ (*node*$_i$ *k* (*mts*@(*mt*,*sep*)#*tts*) *tt*) = *height t* **by** *simp*
moreover have *bal-up*$_i$ (*node*$_i$ *k* (*mts*@(*mt*,*sep*)#*tts*) *tt*)
by (*metis bal-list-merge bal-up*$_i$*.simps*(*2*) *node*$_i$*-bal sub-heights*(*1*) *sub-heights*(*2*) *sub-heights*(*3*) *sub-node t-node*)
ultimately show *?thesis*
apply (*cases node*$_i$ *k* (*mts*@(*mt*,*sep*)#*tts*) *tt*)
using *assms Nil sub-node t-node* **by** *auto*
next
case (*Cons r rs*)
then obtain *rsub rsep* **where** *r-split*: *r* = (*rsub*,*rsep*) **by** (*cases r*)
then have *rsub-height*: *height rsub* = *height t bal rsub*
using *assms Cons* **by** *auto*
then obtain *rts rt* **where** *r-node*: *rsub* = (*Node rts rt*)
apply(*cases rsub*) **using** *t-node* **by** *simp*
have *height-up*$_i$ (*node*$_i$ *k* (*mts*@(*mt*,*sep*)#*rts*) *rt*) = *height* (*Node* (*mts*@(*mt*,*sep*)#*rts*) *rt*)
using *node*$_i$*-height* **by** *blast*
also have ... = *Suc* (*height rt*)
by (*metis Un-iff* ‹*height rsub* = *height t*› *assms bal.simps*(*2*) *bal-split-last*(*1*) *height-bal-tree height-up*$_i$*.simps*(*2*) *height-list-split list.set-intros*(*1*) *Cons max.idem r-node r-split set-append some-child-sub*(*1*) *sub-heights*(*1*) *sub-node*)
also have ... = *height rsub*

```
      using height-bal-tree r-node rsub-height(2) by fastforce
    finally have 1: height-up_i (node_i k (mts@(mt,sep)#rts) rt) = height rsub .
    moreover have 2: bal-up_i (node_i k (mts@(mt,sep)#rts) rt)
      by (metis bal-list-merge bal-up_i.simps(2) node_i-bal r-node rsub-height(1)
rsub-height(2) sub-heights(1) sub-heights(2) sub-node)
    ultimately show ?thesis
    proof (cases node_i k (mts@(mt,sep)#rts) rt)
      case (T_i u)
      then have bal (Node (ls@(u,rsep)#rs) t)
        using 1 2 Cons assms t-node subtrees-split sub-heights r-split rsub-height
        unfolding bal.simps by (auto simp del: height-btree.simps)
      then show ?thesis
        using Cons assms t-node sub-node r-split r-node False T_i
        by (auto simp del: node_i.simps bal.simps)
    next
      case (Up_i l a r)
      then have bal (Node (ls@(l,a)#(r,rsep)#rs) t)
        using 1 2 Cons assms t-node subtrees-split sub-heights r-split rsub-height
        unfolding bal.simps by (auto simp del: height-btree.simps)
      then show ?thesis
        using Cons assms t-node sub-node r-split r-node False Up_i
        by (auto simp del: node_i.simps bal.simps)
    qed
  qed
qed
qed (simp add: height-Leaf)


lemma rebalance-last-tree-bal: ⟦bal (Node ts t); ts ≠ []⟧ ⟹ bal (rebalance-last-tree
k ts t)
  using rebalance-middle-tree-bal append-butlast-last-id[of ts]
  apply(cases last ts)
  apply(auto simp del: bal.simps rebalance-middle-tree.simps)
  done


lemma split-max-bal:
  assumes bal t
    and t ≠ Leaf
    and nonempty-lasttreebal t
  shows bal (fst (split-max k t))
  using assms
proof(induction k t rule: split-max.induct)
  case (1 k ts t)
  then show ?case
  proof (cases t)
    case Leaf
    then obtain sub sep where last-split: last ts = (sub,sep)
      using 1 by auto
```

```
    then have height sub = height t using 1 by auto
    then have bal (Node (butlast ts) sub) using 1 last-split by auto
    then show ?thesis using 1 Leaf last-split by auto
  next
    case (Node tts tt)
   then obtain sub sep where t-split: split-max k t = (sub,sep) by (cases split-max
k t)
    then have height sub = height t using 1 Node
      by (metis btree.distinct(1) nonempty-lasttreebal.simps(2) split-max-height)
    moreover have bal sub
      using 1.IH 1.prems(1) 1.prems(3) Node t-split by fastforce
    ultimately have bal (Node ts sub)
      using 1 t-split Node by auto
    then show ?thesis
      using rebalance-last-tree-bal t-split Node 1
      by (auto simp del: bal.simps rebalance-middle-tree.simps)
  qed
qed simp

lemma del-bal:
  assumes k > 0
    and root-order k t
    and bal t
  shows bal (del k x t)
  using assms
proof(induction k x t rule: del.induct)
  case (2 k x ts t)
  then obtain ls rs where list-split: split ts x = (ls,rs)
    by (cases split ts x)
  then show ?case
  proof (cases rs)
    case Nil
    then have bal (del k x t) using 2 list-split
      by (simp add: order-impl-root-order)
    moreover have height (del k x t) = height t
      using 2 del-height by (simp add: order-impl-root-order)
    moreover have ts ≠ [] using 2 by auto
    ultimately have bal (rebalance-last-tree k ts (del k x t))
      using 2 Nil order-bal-nonempty-lasttreebal rebalance-last-tree-bal
      by simp
    then have bal (rebalance-last-tree k ls (del k x t))
      using list-split split-conc Nil by fastforce
    then show ?thesis
      using 2 list-split Nil
      by auto
  next
    case (Cons r rs)
    then obtain sub sep where r-split: r = (sub,sep) by (cases r)
    then have sub-height: height sub = height t bal sub
```

using *2 Cons list-split split-set(1)* **by** *fastforce+*
consider (*sep-n-x*) *sep* ≠ *x* |
(*sep-x-Leaf*) *sep* = *x* ∧ *sub* = *Leaf* |
(*sep-x-Node*) *sep* = *x* ∧ (∃ *ts t*. *sub* = *Node ts t*)
using *btree.exhaust* **by** *blast*
then show *?thesis*
proof *cases*
case *sep-n-x*
then have *bal* (*del k x sub*) *height* (*del k x sub*) = *height sub* **using** *sub-height*
apply (*metis 2.IH*(*2*) *2.prems*(*1*) *2.prems*(*2*) *list-split local.Cons order-impl-root-order r-split root-order.simps*(*2*) *some-child-sub*(*1*) *split-set*(*1*))
by (*metis 2.prems*(*1*) *2.prems*(*2*) *list-split Cons order-impl-root-order r-split root-order.simps*(*2*) *some-child-sub*(*1*) *del-height split-set*(*1*) *sub-height*(*2*))
moreover have *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
using *2.prems*(*3*) *list-split Cons r-split split-conc* **by** *blast*
ultimately have *bal* (*Node* (*ls*@(*del k x sub*,*sep*)#*rs*) *t*)
using *bal-substitute-subtree*[*of ls sub sep rs t del k x sub*] **by** *metis*
then have *bal* (*rebalance-middle-tree k ls* (*del k x sub*) *sep rs t*)
using *rebalance-middle-tree-bal*[*of ls del k x sub sep rs t k*] **by** *metis*
then show *?thesis*
using *2 list-split Cons r-split sep-n-x* **by** *auto*
next
case *sep-x-Leaf*
moreover have *bal* (*Node* (*ls*@*rs*) *t*)
using *bal-split-last*(*1*) *list-split split-conc r-split*
by (*metis 2.prems*(*3*) *Cons*)
ultimately show *?thesis*
using *2 list-split Cons r-split* **by** *auto*
next
case *sep-x-Node*
then obtain *sts st* **where** *sub-node*: *sub* = *Node sts st* **by** *auto*
then obtain *sub-s max-s* **where** *sub-split*: *split-max k sub* = (*sub-s*, *max-s*)
by (*cases split-max k sub*)
then have *height sub-s* = *height sub*
using *split-max-height*
by (*metis 2.prems*(*1*) *2.prems*(*2*) *btree.distinct*(*1*) *list-split Cons order-bal-nonempty-lasttreebal order-impl-root-order r-split root-order.simps*(*2*) *some-child-sub*(*1*) *split-set*(*1*) *sub-height*(*2*) *sub-node*)
moreover have *bal sub-s*
using *split-max-bal*
by (*metis 2.prems*(*1*) *2.prems*(*2*) *btree.distinct*(*1*) *fst-conv list-split local.Cons order-bal-nonempty-lasttreebal order-impl-root-order r-split root-order.simps*(*2*) *some-child-sub*(*1*) *split-set*(*1*) *sub-height*(*2*) *sub-node sub-split*)
moreover have *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
using *2.prems*(*3*) *list-split Cons r-split split-conc* **by** *blast*
ultimately have *bal* (*Node* (*ls*@(*sub-s*,*sep*)#*rs*) *t*)
using *bal-substitute-subtree*[*of ls sub sep rs t sub-s*] **by** *metis*
then have *bal* (*Node* (*ls*@(*sub-s*,*max-s*)#*rs*) *t*)
using *bal-substitute-separator* **by** *metis*

```
    then have bal (rebalance-middle-tree k ls sub-s max-s rs t)
      using rebalance-middle-tree-bal[of ls sub-s max-s rs t k] by metis
    then show ?thesis
      using 2 list-split Cons r-split sep-x-Node sub-node sub-split by auto
  qed
 qed
qed simp


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(cases t)
 case Leaf
 then have sub = Leaf using height-Leaf assms by auto
 then show ?thesis using Leaf assms by auto
next
 case t-node: (Node tts tt)
 then obtain mts mt where sub-node: sub = Node mts mt
  using assms by (cases sub) (auto)
 then show ?thesis
 proof(cases length mts ≥ k ∧ length tts ≥ k)
  case True
  then have order k sub using assms
   by (simp add: sub-node)
  then show ?thesis
   using True t-node sub-node assms by auto
 next
  case False
  then show ?thesis
  proof (cases rs)
   case Nil
   have order-up_i k (node_i k (mts@(mt,sep)#tts) tt)
    using node_i-order[of k mts@(mt,sep)#tts tt] assms(1,3) t-node sub-node
    by (auto simp del: order-up_i.simps node_i.simps)
   then show ?thesis
    apply(cases node_i k (mts@(mt,sep)#tts) tt)
    using assms t-node sub-node False Nil apply (auto simp del: node_i.simps)
    done
  next
   case (Cons r rs)
   then obtain rsub rsep where r-split: r = (rsub,rsep) by (cases r)
   then have rsub-height: height rsub = height t
    using assms Cons by auto
   then obtain rts rt where r-node: rsub = (Node rts rt)
```

```
      apply(cases rsub) using t-node by simp
    have order-up_i k (node_i k (mts@(mt,sep)#rts) rt)
      using node_i-order[of k mts@(mt,sep)#rts rt] assms(1,2) t-node sub-node
r-node r-split Cons
      by (auto simp del: order-up_i.simps node_i.simps)
    then show ?thesis
      apply(cases node_i k (mts@(mt,sep)#rts) rt)
      using assms t-node sub-node False Cons r-split r-node apply (auto simp
del: node_i.simps)
      done
  qed
 qed
qed


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 (cases t)
 case Leaf
 then have sub = Leaf using height-Leaf assms by auto
 then show ?thesis using Leaf assms by auto
next
 case t-node: (Node tts tt)
 then obtain mts mt where sub-node: sub = Node mts mt
  using assms by (cases sub) (auto)
 then show ?thesis
 proof(cases length mts ≥ k ∧ length tts ≥ k)
  case True
  then have order k sub using assms
   by (simp add: sub-node)
  then show ?thesis
   using True t-node sub-node assms by auto
 next
  case False
  have order-up_i k (node_i k (mts@(mt,sep)#tts) tt)
   using node_i-order[of k mts@(mt,sep)#tts tt] assms t-node sub-node
   by (auto simp del: order-up_i.simps node_i.simps)
  then show ?thesis
   apply(cases node_i k (mts@(mt,sep)#tts) tt)
   using assms t-node sub-node False Nil apply (auto simp del: node_i.simps)
   done
 qed
qed
```

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*)
using *rebalance-middle-tree-last-order assms* **by** *auto*

lemma *split-max-order*:
assumes *order k t*
and *t* ≠ *Leaf*
and *nonempty-lasttreebal t*
shows *almost-order k* (*fst* (*split-max k t*))
using *assms*
proof(*induction k t rule*: *split-max.induct*)
case (*1 k ts t*)
then obtain *ls sub sep* **where** *ts-not-empty*: *ts* = *ls*@[(*sub*,*sep*)]
by *auto*
then show *?case*
proof (*cases t*)
case *Leaf*
then show *?thesis* **using** *ts-not-empty 1* **by** *auto*
next
case (*Node*)
then obtain *s-sub s-max* **where** *sub-split*: *split-max k t* = (*s-sub*, *s-max*)
by (*cases split-max k t*)
moreover have *height sub* = *height s-sub*
by (*metis 1.prems*(*3*) *Node Pair-inject append1-eq-conv btree.distinct*(*1*)
nonempty-lasttreebal.simps(*2*) *split-max-height sub-split ts-not-empty*)
ultimately have *almost-order k* (*rebalance-last-tree k ts s-sub*)
using *rebalance-last-tree-order*[*of ts ls sub sep k s-sub*]
1 ts-not-empty Node sub-split
by *force*
then show *?thesis*
using *Node 1 sub-split* **by** *auto*
qed
qed *simp*

lemma *del-order*:
assumes *k* > *0*
and *root-order k t*
and *bal t*
shows *almost-order k* (*del k x t*)
using *assms*
proof (*induction k x t rule*: *del.induct*)
case (*2 k x ts t*)
then obtain *ls list* **where** *list-split*: *split ts x* = (*ls*, *list*) **by** (*cases split ts x*)
then show *?case*

proof (*cases list*)
case *Nil*
then have *almost-order k* (*del k x t*) **using** *2 list-split*
by (*simp add*: *order-impl-root-order*)
moreover obtain *lls lsub lsep* **where** *ls-split*: *ls* = *lls*@[(*lsub*,*lsep*)]
using *2 Nil list-split*
by (*metis append-Nil2 nonempty-lasttreebal.simps*(*2*) *order-bal-nonempty-lasttreebal split-conc*)
moreover have *height t* = *height* (*del k x t*) **using** *del-height 2*
by (*simp add*: *order-impl-root-order*)
moreover have *length ls* = *length ts*
using *Nil list-split*
by (*auto dest*: *split-length*)
ultimately have *almost-order k* (*rebalance-last-tree k ls* (*del k x t*))
using *rebalance-last-tree-order*[*of ls lls lsub lsep k del k x t*]
by (*metis 2.prems*(*2*) *2.prems*(*3*) *Un-iff append-Nil2 bal.simps*(*2*) *list-split Nil root-order.simps*(*2*) *singletonI split-conc subtrees-split*)
then show *?thesis*
using *2 list-split Nil* **by** *auto*
next
case (*Cons r rs*)

from *Cons* **obtain** *sub sep* **where** *r-split*: *r* = (*sub*,*sep*) **by** (*cases r*)

have *inductive-help*:
case rs of [] ⇒ *True* | (*rsub*,*rsep*)#- ⇒ *height rsub* = *height t*
∀ *s*∈*set* (*subtrees* (*ls* @ *rs*)). *order k s*
Suc (*length* (*ls* @ *rs*)) ≤ *2* ∗ *k*
order k t
using *Cons r-split 2.prems list-split split-set*
by (*auto dest*: *split-conc split*!: *list.splits*)

consider (*sep-n-x*) *sep* ≠ *x* |
(*sep-x-Leaf*) *sep* = *x* ∧ *sub* = *Leaf* |
(*sep-x-Node*) *sep* = *x* ∧ (∃ *ts t*. *sub* = *Node ts t*)
using *btree.exhaust* **by** *blast*
then show *?thesis*
proof *cases*
case *sep-n-x*
then have *almost-order k* (*del k x sub*) **using** *2 list-split Cons r-split order-impl-root-order*
by (*metis bal.simps*(*2*) *root-order.simps*(*2*) *some-child-sub*(*1*) *split-set*(*1*))
moreover have *height* (*del k x sub*) = *height t*
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *bal.simps*(*2*) *list-split Cons order-impl-root-order r-split root-order.simps*(*2*) *some-child-sub*(*1*) *del-height split-set*(*1*))
ultimately have *almost-order k* (*rebalance-middle-tree k ls* (*del k x sub*) *sep rs t*)
using *rebalance-middle-tree-order*[*of k del k x sub ls rs t sep*]
using *inductive-help*

```
      using Cons r-split sep-n-x list-split by auto
    then show ?thesis using 2 Cons r-split sep-n-x list-split by auto
  next
    case sep-x-Leaf
    then have almost-order k (Node (ls@rs) t) using inductive-help by auto
    then show ?thesis using 2 Cons r-split sep-x-Leaf list-split by auto
  next
    case sep-x-Node
    then obtain sts st where sub-node: sub = Node sts st by auto
    then obtain sub-s max-s where sub-split: split-max k sub = (sub-s, max-s)
      by (cases split-max k sub)
    then have height sub-s = height t using split-max-height
      by (metis 2.prems(1) 2.prems(2) 2.prems(3) bal.simps(2) btree.distinct(1)
list-split Cons order-bal-nonempty-lasttreebal order-impl-root-order r-split root-order.simps(2)
some-child-sub(1) split-set(1) sub-node)
    moreover have almost-order k sub-s using split-max-order
      by (metis 2.prems(1) 2.prems(2) 2.prems(3) bal.simps(2) btree.distinct(1)
fst-conv list-split local.Cons order-bal-nonempty-lasttreebal order-impl-root-order r-split
root-order.simps(2) some-child-sub(1) split-set(1)  sub-node sub-split)
    ultimately have almost-order k (rebalance-middle-tree k ls sub-s max-s rs t)
      using rebalance-middle-tree-order[of k sub-s ls rs t max-s] inductive-help
      by auto
    then show ?thesis
      using 2 Cons r-split list-split sep-x-Node sub-split by auto
  qed
 qed
qed simp



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 (cases ls)
 case Nil
 then show ?thesis
   using assms by (auto dest!: split-conc)
next
 case Cons
 then obtain ls' sub sep where ls-tail-split: ls = ls' @ [(sub,sep)]
   by (metis list.distinct(1) rev-exhaust surj-pair)
 moreover have sep < x
   using split-req(2)[of ts x ls' sub sep rs]
   using assms(1) assms(2) ls-tail-split sorted-inorder-separators
   by blast
```

```
  moreover have sorted-less (inorder-list ls)
    using assms sorted-wrt-append split-conc by fastforce
  ultimately show ?thesis using assms(2) split-conc[OF assms(1)]
    using del-list-sorted[of inorder-list ls' @ inorder sub sep]
    by auto
qed



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 −
  from assms have x < sep
  proof −
    from assms have sorted-less (separators ts)
      using sorted-inorder-separators by blast
    then show ?thesis
      using split-req(3)
      using assms
      by fastforce
  qed
  moreover have sorted-less (inorder sub @ sep # inorder-list rs @ inorder t)
    using assms sorted-wrt-append[where xs=inorder-list ls]
    by (auto dest!: split-conc)
  ultimately show ?thesis
    using del-list-sorted[of inorder sub sep]
    by auto
qed

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)
  using assms
proof (induction k x t rule: del.induct)
  case (2 k x ts t)
  then obtain ls rs where list-split: split ts x = (ls, rs)
    by (meson surj-pair)
  then have list-conc: ts = ls @ rs
    using split.split-conc split-axioms by blast
  show ?case
```

proof (*cases rs*)
case *Nil*
then have *IH*: *inorder* (*del k x t*) = *del-list x* (*inorder t*)
by (*metis 2.IH*(*1*) *2.prems bal.simps*(*2*) *list-split order-impl-root-order root-order.simps*(*2*) *sorted-inorder-induct-last*)
have *inorder* (*del k x* (*Node ts t*)) = *inorder* (*rebalance-last-tree k ts* (*del k x t*))
using *list-split Nil list-conc* **by** *auto*
also have ... = *inorder-list ts* @ *inorder* (*del k x t*)
proof −
obtain *ts' sub sep* **where** *ts-split*: *ts* = *ts'* @ [(*sub*, *sep*)]
by (*meson 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *nonempty-lasttreebal.simps*(*2*) *order-bal-nonempty-lasttreebal*)
then have *height sub* = *height t*
using *2.prems*(*3*) **by** *auto*
moreover have *height t* = *height* (*del k x t*)
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *bal.simps*(*2*) *del-height order-impl-root-order root-order.simps*(*2*))
ultimately show *?thesis*
using *rebalance-last-tree-inorder*
using *ts-split* **by** *auto*
qed
also have ... = *inorder-list ts* @ *del-list x* (*inorder t*)
using *IH* **by** *blast*
also have ... = *del-list x* (*inorder* (*Node ts t*))
using *2.prems*(*4*) *list-conc list-split Nil del-list-split*
by *auto*
finally show *?thesis* **.**
next
case (*Cons h rs*)
then obtain *sub sep* **where** *h-split*: *h* = (*sub*,*sep*)
by (*cases h*)
then have *node-sorted-split*:
sorted-less (*inorder* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*))
root-order k (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
bal (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
using *2.prems h-split list-conc Cons* **by** *blast+*
consider (*sep-n-x*) *sep* ≠ *x* | (*sep-x-Leaf*) *sep* = *x* ∧ *sub* = *Leaf* | (*sep-x-Node*) *sep* = *x* ∧ (∃ *ts t*. *sub* = *Node ts t*)
using *btree.exhaust* **by** *blast*
then show *?thesis*
proof *cases*
case *sep-n-x*
then have *IH*: *inorder* (*del k x sub*) = *del-list x* (*inorder sub*)
by (*metis 2.IH*(*2*) *2.prems*(*1*) *2.prems*(*2*) *bal.simps*(*2*) *bal-split-left*(*1*) *h-split list-split local.Cons node-sorted-split*(*1*) *node-sorted-split*(*3*) *order-impl-root-order root-order.simps*(*2*) *some-child-sub*(*1*) *sorted-inorder-induct-subtree split-set*(*1*))
from *sep-n-x* **have** *inorder* (*del k x* (*Node ts t*)) = *inorder* (*rebalance-middle-tree k ls* (*del k x sub*) *sep rs t*)

using *list-split Cons h-split* **by** *auto*
also have ... = *inorder* (*Node* (*ls*@(*del k x sub*, *sep*)#*rs*) *t*)
proof −
have *height t* = *height* (*del k x sub*)
using *del-height*
using *order-impl-root-order 2.prems*
by (*auto simp add*: *order-impl-root-order Cons list-conc h-split*)
moreover have *case rs of* [] ⇒ *True* | (*rsub*, *rsep*) # *list* ⇒ *height rsub* = *height t*
using *2.prems*(*3*) *bal-sub-height list-conc Cons* **by** *blast*
ultimately show *?thesis*
using *rebalance-middle-tree-inorder*
by *simp*
qed
also have ... = *inorder-list ls* @ *del-list x* (*inorder sub*) @ *sep* # *inorder-list rs* @ *inorder t*
using *IH* **by** *simp*
also have ... = *del-list x* (*inorder* (*Node ts t*))
using *del-list-split*[*of ts x ls* (*sub*,*sep*)#*rs t*]
using *del-list-split-right*[*of ts x ls sub sep rs t*]
using *list-split list-conc h-split Cons 2.prems*(*4*) *sep-n-x*
by *auto*
finally show *?thesis* **.**
next
case *sep-x-Leaf*
then have *del-list x* (*inorder* (*Node ts t*)) = *inorder* (*Node* (*ls*@*rs*) *t*)
using *list-conc h-split Cons*
using *del-list-split*[*OF list-split 2.prems*(*4*)]
by *simp*
also have ... = *inorder* (*del k x* (*Node ts t*))
using *list-split sep-x-Leaf list-conc h-split Cons*
by *auto*
finally show *?thesis* **by** *simp*
next
case *sep-x-Node*
obtain *ssub ssep* **where** *split-split*: *split-max k sub* = (*ssub*, *ssep*)
by *fastforce*
from *sep-x-Node* **have** *x* = *sep*
by *simp*
then have *del-list x* (*inorder* (*Node ts t*)) = *inorder-list ls* @ *inorder sub* @ *inorder-list rs* @ *inorder t*
using *list-split list-conc h-split Cons 2.prems*(*4*)
using *del-list-split*[*OF list-split 2.prems*(*4*)]
using *del-list-sorted1*[*of inorder sub sep inorder-list rs* @ *inorder t x*]
sorted-wrt-append
by *auto*
also have ... = *inorder-list ls* @ *inorder-pair* (*split-max k sub*) @ *inorder-list rs* @ *inorder t*
using *sym*[*OF split-max-inorder*[*of sub k*]]

```
      using order-bal-nonempty-lasttreebal[of k sub] 2.prems
        list-conc h-split Cons sep-x-Node
      by (auto simp del: split-max.simps simp add: order-impl-root-order)
    also have ... = inorder-list ls @ inorder ssub @ ssep # inorder-list rs @
inorder t
      using split-split by auto
    also have ... = inorder (rebalance-middle-tree k ls ssub ssep rs t)
    proof −
      have height t = height ssub
        using split-max-height
       by (metis 2.prems(1,2,3) bal.simps(2) btree.distinct(1) h-split list-split lo-
cal.Cons order-bal-nonempty-lasttreebal order-impl-root-order root-order.simps(2)
sep-x-Node some-child-sub(1) split-set(1) split-split)
      moreover have case rs of [] ⇒ True | (rsub, rsep) # list ⇒ height rsub =
height t
        using 2.prems(3) bal-sub-height list-conc local.Cons
        by blast
      ultimately show ?thesis
        using rebalance-middle-tree-inorder
        by auto
    qed
    also have ... = inorder (del k x (Node ts t))
      using list-split sep-x-Node list-conc h-split Cons split-split
      by auto
    finally show ?thesis by simp
  qed
 qed
qed auto

lemma reduce-root-order: ⟦k > 0; almost-order k t⟧ ⟹ root-order k (reduce-root
t)
 apply(cases t)
  apply(auto split!: list.splits simp add: order-impl-root-order)
 done

lemma reduce-root-bal: bal (reduce-root t) = bal t
 apply(cases t)
  apply(auto split!: list.splits)
 done


lemma reduce-root-inorder: inorder (reduce-root t) = inorder t
 apply (cases t)
  apply (auto split!: list.splits)
 done


lemma delete-order: ⟦k > 0; bal t; root-order k t⟧ ⟹ root-order k (delete k x t)
 using del-order
```

by (*simp add*: *reduce-root-order*)

lemma *delete-bal*: ⟦*k* > *0*; *bal t*; *root-order k t*⟧ ⟹ *bal* (*delete k x t*)
using *del-bal*
by (*simp add*: *reduce-root-bal*)

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*)
using *del-inorder*
by (*simp add*: *reduce-root-inorder*)

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 (*standard*, *goal-cases*)
case (*2 s x*)
then show *?case*
by (*simp add*: *isin-set-inorder*)
next
case (*3 s x*)
then show *?case* **using** *insert-inorder*
by *simp*
next
case (*4 s x*)
then show *?case* **using** *delete-inorder*
by *auto*
next
case (*6 s x*)
then show *?case* **using** *insert-order insert-bal*
by *auto*
next
case (*7 s x*)
then show *?case* **using** *delete-order delete-bal*
by *auto*
qed (*simp add*: *empty-btree-def*)+

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 −

have *linear-split-help xs x prev* = (*prev* @ *takeWhile* (λ(-, *s*). *s* < *x*) *xs*, *dropWhile* (λ(-, *s*). *s* < *x*) *xs*)
for *prev*
apply (*induction xs arbitrary*: *prev*)
apply *auto*
done
thus *?thesis* **by** *auto*
qed

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*
apply *unfold-locales*
unfolding *linear-split-alt*
apply (*auto split*: *list.splits*)
subgoal
by (*metis* (*no-types*, *lifting*) *case-prodD in-set-conv-decomp takeWhile-eq-all-conv takeWhile-idem*)

subgoal
by (*metis case-prod-conv hd-dropWhile le-less-linear list.sel*(*1*) *list.simps*(*3*))
done

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

abbreviation $btree_i \equiv$ *btree-ls-insert*
abbreviation $btree_d \equiv$ *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*) $\Longrightarrow$ $y \leq sep$
apply(*induction t y xs rule*: *linear-split-help.induct*)
apply(*simp-all*)
by (*metis Pair-inject le-less-linear list.inject*)

lemma *linear-split-append*: *linear-split-help xs p ys* = (*ls*,*rs*) $\Longrightarrow$ *ls*@*rs* = *ys*@*xs*
apply(*induction xs p ys rule*: *linear-split-help.induct*)
apply(*simp-all*)
by (*metis Pair-inject*)

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
  apply(induction xs p ys rule: linear-split-help.induct)
   apply(simp-all)
  by (metis prod.inject)+

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⟧ ⟹
(case rs of [] ⇒ True | (-,sep)#- ⇒ p ≤ sep)
  apply(cases rs)
  by (auto simp add: some-child-sm)


lemma linear-split-req:
  assumes  linear-split xs p = (ls,(sub,sep)#rs)
    and sorted-less (separators xs)
  shows  p ≤ sep
  using assms linear-split-gr by fastforce

lemma linear-split-req2:
  assumes  linear-split xs p = (ls@[(sub,sep)],rs)
    and sorted-less (separators xs)
  shows  sep < p
  using linear-split-sm[of xs p [] ls@[(sub,sep)] rs]
  using assms(1) assms(2)
 by (metis Nil-is-map-conv Un-iff append-self-conv2 empty-iff empty-set linear-split.elims
prod-set-simps(2) separators-split snd-eqD snds.intros)


interpretation split linear-split
  by (simp add: linear-split-req linear-split-req2 linear-split-append split-def)
```

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)
    )
  )
```

)
  by *pat-completeness auto*
termination
  apply(*relation measure* (λ(*ls*,*xs*,*rs*,*x*). *length xs*))
   apply (*auto*)
  by (*metis append-take-drop-id length-Cons length-append lessI trans-less-add2*)

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*)
  apply(*induction as bs cs x arbitrary*: *ls rs rule*: *binary-split-help.induct*)
   apply (*auto simp add*: *drop-not-empty split*!: *list.splits*)
  subgoal for *ls a b va rs x lsa rsa aa ba x22*
    apply(*cases cmp x ba*)
      apply *auto*
      apply (*metis Cons-eq-appendI append-eq-appendI append-take-drop-id*)
     apply (*metis append-take-drop-id*)
    by (*metis Cons-eq-appendI append-eq-appendI append-take-drop-id*)
  done

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*
  oops

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 ..

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*}
by *simp*

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

lemma *finite-set-ins-swap*:
assumes *finite A*
shows *max a* (*Max* (*Set.insert b A*)) = *max b* (*Max* (*Set.insert a A*))
using *Max-insert assms max.commute max.left-commute* **by** *fastforce*

lemma *finite-set-in-idem*:
assumes *finite A*
shows *max a* (*Max* (*Set.insert a A*)) = *Max* (*Set.insert a A*)
using *Max-insert assms max.commute max.left-commute* **by** *fastforce*

lemma *height-Leaf*: *height t* = *0* ⟷ (∃ *ks*. *t* = (*Leaf ks*))
by (*induction t*) (*auto*)

lemma *height-bplustree-order*:
height (*Node* (*ls*@[*a*]) *t*) = *height* (*Node* (*a*#*ls*) *t*)
by *simp*

lemma *height-bplustree-sub*:
height (*Node* ((*sub*,*x*)#*ls*) *t*) = *max* (*height* (*Node ls t*)) (*Suc* (*height sub*))
by *simp*

lemma *height-bplustree-last*:
height (*Node* ((*sub*,*x*)#*ts*) *t*) = *max* (*height* (*Node ts sub*)) (*Suc* (*height t*))
by (*induction ts*) *auto*

lemma *set-leaves-leaves*: *set* (*leaves t*) = *set-leaves t*

apply(*induction t*)
apply(*auto*)
done

lemma *set-nodes-nodes*: *set* (*inorder t*) = *set-nodes t*
apply(*induction t*)
apply(*auto simp add*: *rev-image-eqI*)
done

lemma *child-subset-leaves*: *p* ∈ *set t* ⟹ *set-leaves* (*fst p*) ⊆ *set-leaves* (*Node t n*)
apply(*induction p arbitrary*: *t n*)
apply(*auto*)
done

lemma *child-subset*: *p* ∈ *set t* ⟹ *set-nodes* (*fst p*) ⊆ *set-nodes* (*Node t n*)
apply(*induction p arbitrary*: *t n*)
apply(*auto*)
done

lemma *some-child-sub*:
assumes (*sub*,*sep*) ∈ *set t*
shows *sub* ∈ *set* (*subtrees t*)
and *sep* ∈ *set* (*separators t*)
using *assms* **by** *force*+

lemma *bal-all-subtrees-equal*: *bal* (*Node ts t*) ⟹ (∀ *s1* ∈ *set* (*subtrees ts*). ∀ *s2* ∈ *set* (*subtrees ts*). *height s1* = *height s2*)
by (*metis BPlusTree.bal.simps*(*2*))

lemma *fold-max-set*: ∀ *x* ∈ *set t*. *x* = *f* ⟹ *fold max t f* = *f*
apply(*induction t*)
apply(*auto simp add*: *max-def-raw*)
done

lemma *height-bal-tree*: *bal* (*Node ts t*) ⟹ *height* (*Node ts t*) = *Suc* (*height t*)
by (*induction ts*) *auto*

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*)
using *assms* **by** *auto*

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*)
using *assms* **by** (*auto simp add*: *image-constant-conv*)

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*)
using *assms* **by** (*auto simp add*: *image-constant-conv*)

lemma *bal-substitute*: ⟦*bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*); *height t* = *height c*; *bal c*⟧ ⟹ *bal* (*Node* (*ls*@(*c*,*b*)#*rs*) *t*)
unfolding *bal.simps*
by *auto*

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

lemma *bal-substitute-separator*: *bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*) ⟹ *bal* (*Node* (*ls*@(*a*,*c*)#*rs*) *t*)
unfolding *bal.simps*
by *auto*

lemma *order-impl-root-order*: ⟦*k* > *0*; *order k t*⟧ ⟹ *root-order k t*
apply(*cases t*)
apply(*auto*)
done

lemma *sorted-inorder-list-separators*: *sorted-less* (*inorder-list ts*) ⟹ *sorted-less* (*separators ts*)
apply(*induction ts*)
apply (*auto simp add*: *sorted-lems*)
done

corollary *sorted-inorder-separators*: *sorted-less* (*inorder* (*Node ts t*)) ⟹ *sorted-less* (*separators ts*)
using *sorted-inorder-list-separators sorted-wrt-append*

by *auto*

lemma *sorted-inorder-list-subtrees*:
sorted-less (*inorder-list ts*) $\Longrightarrow \forall$ *sub* $\in$ *set* (*subtrees ts*). *sorted-less* (*inorder sub*)
apply(*induction ts*)
apply (*auto simp add*: *sorted-lems*)+
done

corollary *sorted-inorder-subtrees*: *sorted-less* (*inorder* (*Node ts t*)) $\Longrightarrow \forall$ *sub* $\in$ *set* (*subtrees ts*). *sorted-less* (*inorder sub*)
using *sorted-inorder-list-subtrees sorted-wrt-append* **by** *auto*

lemma *sorted-inorder-list-induct-subtree*:
sorted-less (*inorder-list* (*ls*@(*sub*,*sep*)#*rs*)) $\Longrightarrow$ *sorted-less* (*inorder sub*)
by (*simp add*: *sorted-wrt-append*)

corollary *sorted-inorder-induct-subtree*:
sorted-less (*inorder* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)) $\Longrightarrow$ *sorted-less* (*inorder sub*)
by (*simp add*: *sorted-wrt-append*)

lemma *sorted-inorder-induct-last*: *sorted-less* (*inorder* (*Node ts t*)) $\Longrightarrow$ *sorted-less* (*inorder t*)
by (*simp add*: *sorted-wrt-append*)

lemma *sorted-leaves-list-subtrees*:
sorted-less (*leaves-list ts*) $\Longrightarrow \forall$ *sub* $\in$ *set* (*subtrees ts*). *sorted-less* (*leaves sub*)
apply(*induction ts*)
apply (*auto simp add*: *sorted-wrt-append*)+
done

corollary *sorted-leaves-subtrees*: *sorted-less* (*leaves* (*Node ts t*)) $\Longrightarrow \forall$ *sub* $\in$ *set* (*subtrees ts*). *sorted-less* (*leaves sub*)
using *sorted-leaves-list-subtrees sorted-wrt-append* **by** *auto*

lemma *sorted-leaves-list-induct-subtree*:
sorted-less (*leaves-list* (*ls*@(*sub*,*sep*)#*rs*)) $\Longrightarrow$ *sorted-less* (*leaves sub*)
by (*simp add*: *sorted-wrt-append*)

corollary *sorted-leaves-induct-subtree*:
sorted-less (*leaves* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)) $\Longrightarrow$ *sorted-less* (*leaves sub*)
by (*simp add*: *sorted-wrt-append*)

lemma *sorted-leaves-induct-last*: *sorted-less* (*leaves* (*Node ts t*)) $\Longrightarrow$ *sorted-less* (*leaves t*)
by (*simp add*: *sorted-wrt-append*)

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 ∧ inbetween f sep xs t u)

thm *fold-cong*

lemma *cong-inbetween*[*fundef-cong*]:
⟦*a = b*; *xs = ys*; ⋀*l' u' sub sep*. *(sub,sep) ∈ set ys ⟹ f l' sub u' = g l' sub u'*; ⋀*l' u'*. *f l' a u' = g l' b u'*⟧
⟹ *inbetween f l xs a u = inbetween g l ys b u*
apply(*induction ys arbitrary*: *l a b u xs*)
apply *auto*
apply *fastforce+*
done

fun *aligned* :: *'a* ::*linorder* ⇒ - **where**
aligned l (Leaf ks) u = (l < u ∧ (∀ x ∈ set ks. l < x ∧ x ≤ u)) |
aligned l (Node ts t) u = (inbetween aligned l ts t u)

lemma *sorted-less-merge*: *sorted-less (as@[a]) ⟹ sorted-less (a#bs) ⟹ sorted-less (as@a#bs)*
using *sorted-mid-iff* **by** *blast*

thm *aligned.simps*

lemma *leaves-cases*: *x ∈ set (leaves (Node ts t)) ⟹ (∃ (sub,sep) ∈ set ts. x ∈ set (leaves sub)) ∨ x ∈ set (leaves t)*
apply (*induction ts*)
apply *auto*
done

lemma *align-sub*: *aligned l (Node ts t) u ⟹ (sub,sep) ∈ set ts ⟹ ∃ l' ∈ set (separators ts) ∪ {l}. aligned l' sub sep*
apply(*induction ts arbitrary*: *l*)
apply *auto*
done

lemma *align-last*: *aligned l (Node (ts@[(sub,sep)]) t) u ⟹ aligned sep t u*
apply(*induction ts arbitrary*: *l*)
apply *auto*
done

lemma *align-last'*: *aligned l (Node ts t) u ⟹ ∃ l' ∈ set (separators ts) ∪ {l}. aligned l' t u*
apply(*induction ts arbitrary*: *l*)
apply *auto*

```
  done

lemma aligned-sorted-inorder: aligned l t u ⟹ sorted-less (l#(inorder t)@[u])
proof(induction l t u rule: aligned.induct)
  case (2 l ts t u)
  then show ?case
  proof(cases ts)
    case Nil
    then show ?thesis
      using 2 by auto
  next
    case Cons
    then obtain ts' sub sep where ts-split: ts = ts'@[(sub,sep)]
      by (metis list.distinct(1) rev-exhaust surj-pair)
    moreover from 2 have sorted-less (l#(inorder-list ts))
    proof (induction ts arbitrary: l)
      case (Cons a ts')
      then show ?case
      proof (cases a)
        case (Pair sub sep)
        then have aligned l sub sep inbetween aligned sep ts' t u
          using Cons.prems by simp+
        then have aligned sep (Node ts' t) u
          by simp
        then have sorted-less (sep#inorder-list ts')
          using Cons
          by (metis insert-iff list.set(2))
        moreover have sorted-less (l#inorder sub@[sep])
          using Cons
          by (metis Pair ‹aligned l sub sep› list.set-intros(1))
        ultimately show ?thesis
          using Pair sorted-less-merge[of l#inorder sub sep inorder-list ts']
          by simp
      qed
    qed simp
    moreover have sorted-less (sep#inorder t@[u])
    proof −
      from 2 have aligned sep t u
        using align-last ts-split by blast
      then show ?thesis
        using 2.IH by blast
    qed
    ultimately show ?thesis
      using sorted-less-merge[of l#inorder-list ts'@inorder sub sep inorder t@[u]]
      by simp
  qed
qed simp

lemma separators-in-inorder-list: set (separators ts) ⊆ set (inorder-list ts)
```

apply (*induction ts*)
apply *auto*
done

lemma *separators-in-inorder*: *set* (*separators ts*) ⊆ *set* (*inorder* (*Node ts t*))
by *fastforce*

lemma *aligned-sorted-separators*: *aligned l* (*Node ts t*) *u* ⟹ *sorted-less* (*l*#(*separators ts*)@[*u*])
by (*smt* (*verit*, *ccfv-threshold*) *aligned-sorted-inorder separators-in-inorder sorted-inorder-separators sorted-lems*(*2*) *sorted-wrt.simps*(*2*) *sorted-wrt-append subset-eq*)

lemma *aligned-leaves-inbetween*: *aligned l t u* ⟹ ∀ *x* ∈ *set* (*leaves t*). *l* < *x* ∧ *x* ≤ *u*
proof (*induction l t u rule*: *aligned.induct*)
case (*1 l ks u*)
then show *?case* **by** *auto*
next
case (*2 l ts t u*)
have ∗: *sorted-less* (*l*#*inorder* (*Node ts t*)@[*u*])
using *2.prems aligned-sorted-inorder* **by** *blast*
show *?case*
proof
fix *x* **assume** *x* ∈ *set* (*leaves* (*Node ts t*))
then consider (*sub*) ∃(*sub*,*sep*) ∈ *set ts*. *x* ∈ *set* (*leaves sub*) | (*last*) *x* ∈ *set* (*leaves t*)
by *fastforce*
then show *l* < *x* ∧ *x* ≤ *u*
proof (*cases*)
case *sub*
then obtain *sub sep* **where** (*sub*,*sep*) ∈ *set ts x* ∈ *set* (*leaves sub*) **by** *auto*
then obtain *l*′ **where** *aligned l*′ *sub sep l*′ ∈ *set* (*separators ts*) ∪ {*l*}
using *2.prems*(*1*) ‹(*sub*, *sep*) ∈ *set ts*› *align-sub* **by** *blast*
then have ∀ *x* ∈ *set* (*leaves sub*). *l*′ < *x* ∧ *x* ≤ *sep*
using *2.IH*(*1*) ‹(*sub*,*sep*) ∈ *set ts*› **by** *auto*
moreover from ∗ **have** *l* ≤ *l*′
by (*metis Un-insert-right* ‹*l*′ ∈ *set* (*separators ts*) ∪ {*l*}› *append-Cons boolean-algebra-cancel.sup0 dual-order.eq-iff insert-iff less-imp-le separators-in-inorder sorted-snoc sorted-wrt.simps*(*2*) *subset-eq*)
moreover from ∗ **have** *sep* ≤ *u*
by (*metis* ‹(*sub*, *sep*) ∈ *set ts*› *less-imp-le list.set-intros*(*1*) *separators-in-inorder some-child-sub*(*2*) *sorted-mid-iff2 sorted-wrt-append subset-eq*)
ultimately show *?thesis*
by (*meson* ‹*x* ∈ *set* (*leaves sub*)› *order.strict-trans1 order.trans*)
next
case *last*
then obtain *l*′ **where** *aligned l*′ *t u l*′ ∈ *set* (*separators ts*) ∪ {*l*}
using *align-last*′ *2.prems* **by** *blast*
then have ∀ *x* ∈ *set* (*leaves t*). *l*′ < *x* ∧ *x* ≤ *u*

using *2.IH(2)* **by** *auto*
moreover from $*$ **have** $l \leq l'$
by (*metis Un-insert-right ‹l' ∈ set (separators ts) ∪ {l}› append-Cons boolean-algebra-cancel.sup0 dual-order.eq-iff insert-iff less-imp-le separators-in-inorder sorted-snoc sorted-wrt.simps(2) subset-eq*)
ultimately show *?thesis*
by (*meson ‹x ∈ set (leaves t)› order.strict-trans1 order.trans*)
qed
qed
qed

lemma *aligned-leaves-list-inbetween*: *aligned l (Node ts t) u ⟹ ∀ x ∈ set (leaves-list ts). l < x ∧ x ≤ u*
by (*metis Un-iff aligned-leaves-inbetween leaves.simps(2) set-append*)

lemma *aligned-split-left*: *aligned l (Node (ls@(sub,sep)#rs) t) u ⟹ aligned l (Node ls sub) sep*
apply(*induction ls arbitrary*: *l*)
apply *auto*
done

lemma *aligned-split-right*: *aligned l (Node (ls@(sub,sep)#rs) t) u ⟹ aligned sep (Node rs t) u*
apply(*induction ls arbitrary*: *l*)
apply *auto*
done

lemma *aligned-subst*: *aligned l (Node (ls@(sub', subl)#(sub,subsep)#rs) t) u ⟹ aligned subl subsub subsep ⟹*
aligned l (Node (ls@(sub',subl)#(subsub,subsep)#rs) t) u
apply (*induction ls arbitrary*: *l*)
apply *auto*
done

lemma *aligned-subst-emptyls*: *aligned l (Node ((sub,subsep)#rs) t) u ⟹ aligned l subsub subsep ⟹*
aligned l (Node ((subsub,subsep)#rs) t) u
by *auto*

lemma *aligned-subst-last*: *aligned l (Node (ts'@[(sub', sep')]) t) u ⟹ aligned sep' t' u ⟹*
aligned l (Node (ts'@[(sub', sep')]) t') u
apply (*induction ts' arbitrary*: *l*)
apply *auto*
done

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*)
by *simp*

lemma *aligned-imp-Laligned*: *aligned l t u* ⟹ *Laligned t u*
apply (*induction l t u rule*: *aligned.induct*)
apply *simp*
subgoal for *l ts t u*
apply(*cases ts*)
apply *auto*
apply *blast*
done
done

lemma *Laligned-split-left*: *Laligned* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) *u* ⟹ *Laligned* (*Node ls sub*) *sep*
apply(*cases ls*)
apply (*auto dest*!: *aligned-imp-Laligned*)
apply (*meson aligned.simps*(*2*) *aligned-split-left*)
done

lemma *Laligned-split-right*: *Laligned* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*) *u* ⟹ *aligned sep* (*Node rs t*) *u*
apply(*cases ls*)
apply (*auto split*!: *list.splits dest*!: *aligned-imp-Laligned*)
apply (*meson aligned.simps*(*2*) *aligned-split-right*)
done

lemma *Lalign-sub*: *Laligned* (*Node* ((*a*,*b*)#*ts*) *t*) *u* ⟹ (*sub*,*sep*) ∈ *set ts* ⟹ ∃ *l'* ∈ *set* (*separators ts*) ∪ {*b*}. *aligned l' sub sep*
apply(*induction ts arbitrary*: *a b*)
apply (*auto dest*!: *aligned-imp-Laligned*)
done

lemma *Lalign-last*: *Laligned* (*Node* (*ts*@[(*sub*,*sep*)]) *t*) *u* ⟹ *aligned sep t u*
by (*cases ts*) (*auto simp add*: *align-last*)

lemma *Lalign-last'*: *Laligned* (*Node* ((*a*,*b*)#*ts*) *t*) *u* ⟹ ∃ *l'* ∈ *set* (*separators ts*) ∪ {*b*}. *aligned l' t u*
apply(*induction ts arbitrary*: *a b*)
apply (*auto dest*!: *aligned-imp-Laligned*)
done

lemma *Lalign-Llast*: *Laligned* (*Node ts t*) *u* ⟹ *Laligned t u*
apply(*cases ts*)
apply *auto*

using *aligned-imp-Laligned Lalign-last′ Laligned-nonempty-Node*
by *metis*

lemma *Laligned-sorted-inorder*: *Laligned t u* $\Longrightarrow$ *sorted-less* ((*inorder t*)@[*u*])
proof(*induction t u rule*: *Laligned.induct*)
case (*1 ks u*)
then show *?case* **by** *auto*
next
case (*2 ts t u*)
then show *?case*
apply (*cases ts*)
apply *auto*
by (*metis aligned.simps*(*2*) *aligned-sorted-inorder append-assoc inorder.simps*(*2*) *sorted-less-merge*)
qed

lemma *Laligned-sorted-separators*: *Laligned* (*Node ts t*) *u* $\Longrightarrow$ *sorted-less* ((*separators ts*)@[*u*])
by (*smt* (*verit*, *del-insts*) *Laligned-sorted-inorder separators-in-inorder sorted-inorder-separators sorted-wrt-append subset-eq*)

lemma *Laligned-leaves-inbetween*: *Laligned t u* $\Longrightarrow \forall x \in$ *set* (*leaves t*). $x \leq u$
proof (*induction t u rule*: *Laligned.induct*)
case (*1 ks u*)
then show *?case* **by** *auto*
next
case (*2 ts t u*)
have $*$: *sorted-less* (*inorder* (*Node ts t*)@[*u*])
using *2.prems Laligned-sorted-inorder* **by** *blast*
show *?case*
proof (*cases ts*)
case *Nil*
show *?thesis*
proof
fix *x* **assume** $x \in$ *set* (*leaves* (*Node ts t*))
then have $x \in$ *set* (*leaves t*)
using *Nil* **by** *auto*
moreover have *Laligned t u*
using *2.prems Nil* **by** *auto*
ultimately show $x \leq u$
using *2.IH*(*1*) *Nil*
by *simp*
qed
next
case (*Cons h ts′*)
then obtain *a b* **where** *h-split*: $h = (a,b)$
by (*cases h*)

show *?thesis*
proof
fix x **assume** $x \in$ *set* (*leaves* (*Node ts t*))
then consider (*first*) $x \in$ *set* (*leaves a*) | (*sub*) $\exists$(*sub*,*sep*) $\in$ *set ts'*. $x \in$ *set* (*leaves sub*) | (*last*) $x \in$ *set* (*leaves t*)
using *Cons h-split* **by** *fastforce*
then show $x \leq u$
proof (*cases*)
case *first*
moreover have *Laligned a b*
using *2.prems Cons h-split* **by** *auto*
moreover have $b \leq u$
by (*metis* ∗ *h-split less-imp-le list.set-intros*(*1*) *local.Cons separators-in-inorder some-child-sub*(*2*) *sorted-wrt-append subsetD*)
ultimately show *?thesis*
using *2.IH*(*2*)[*OF Cons sym*[*OF h-split*]]
by *auto*
next
case *sub*
then obtain *sub sep* **where** (*sub*,*sep*) $\in$ *set ts'* $x \in$ *set* (*leaves sub*) **by** *auto*

then obtain l' **where** *aligned l' sub sep* $l' \in$ *set* (*separators ts'*) $\cup$ {b}
using *2.prems Lalign-sub h-split local.Cons* **by** *blast*
then have $\forall x \in$ *set* (*leaves sub*). $l' < x \wedge x \leq$ *sep*
by (*meson aligned-leaves-inbetween*)
moreover from ∗ **have** *sep* $\leq u$
by (*metis 2.prems Laligned-sorted-separators* ‹(*sub*, *sep*) $\in$ *set ts'*› *insert-iff less-imp-le list.set*(*2*) *local.Cons some-child-sub*(*2*) *sorted-wrt-append*)
ultimately show *?thesis*
by (*meson* ‹$x \in$ *set* (*leaves sub*)› *order.strict-trans1 order.trans*)
next
case *last*
then obtain l' **where** *aligned l' t u* $l' \in$ *set* (*separators ts'*) $\cup$ {b}
using *2.prems Lalign-last' h-split local.Cons* **by** *blast*
then have $\forall x \in$ *set* (*leaves t*). $l' < x \wedge x \leq u$
by (*meson aligned-leaves-inbetween*)
then show *?thesis*
by (*meson* ‹$x \in$ *set* (*leaves t*)› *order.strict-trans1 order.trans*)
qed
qed
qed
qed

lemma *Laligned-leaves-list-inbetween*: *Laligned* (*Node ts t*) $u \Longrightarrow \forall x \in$ *set* (*leaves-list ts*). $x \leq u$
by (*metis Un-iff Laligned-leaves-inbetween leaves.simps*(*2*) *set-append*)

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

Laligned (*Node* (*ts'*@[(*sub'*, *sep'*)]) *t'*) *u*
apply (*cases ts'*)
apply (*auto*)
by (*meson aligned.simps*(*2*) *aligned-subst-last*)

lemma *Laligned-subst*: *Laligned* (*Node* (*ls*@(*sub'*, *subl*)#(*sub*,*subsep*)#*rs*) *t*) *u* ⟹
aligned subl subsub subsep ⟹
Laligned (*Node* (*ls*@(*sub'*,*subl*)#(*subsub*,*subsep*)#*rs*) *t*) *u*
apply (*induction ls*)
apply *auto*
apply (*meson aligned.simps*(*2*) *aligned-subst*)
done

lemma *concat-leaf-nodes-leaves*: (*concat* (*map leaves* (*leaf-nodes t*))) = *leaves t*
apply(*induction t rule*: *leaf-nodes.induct*)
subgoal by *auto*
subgoal for *ts t*
apply(*induction ts*)
apply *simp*
apply *auto*
done
done

lemma *leaf-nodes-not-empty*: *leaf-nodes t* ≠ []
by (*induction t*) *auto*

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

5.6 Auxiliary functions

fun *split-half*:: - *list* ⇒ - *list* × - *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* ∧ *length ls* = (*length xs* + *1*) *div 2*)
by *force*

lemma *drop-not-empty*: *xs* ≠ [] ⟹ *drop* (*length xs div 2*) *xs* ≠ []
apply(*induction xs*)
apply(*auto split*!: *list.splits*)
done

lemma *take-not-empty*: *xs* ≠ [] ⟹ *take* ((*length xs* + *1*) *div 2*) *xs* ≠ []
apply(*induction xs*)
apply(*auto split*!: *list.splits*)
done

lemma *split-half-not-empty*: *length xs* $\geq$ *1* $\Longrightarrow$ $\exists$ *ls a rs. split-half xs* = (*ls*@[*a*],*rs*)
using *take-not-empty*
by (*metis* (*no-types*, *opaque-lifting*) *Ex-list-of-length One-nat-def le-trans length-Cons list.size*(*4*) *nat-1-add-1 not-one-le-zero rev-exhaust split-half .simps take0 take-all-iff*)

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*
apply(*induction ls*)
apply *auto*
done

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*)⟧ $\Longrightarrow$ *xs* = *ls* @ *rs*
⟦*split xs p* = (*ls*@[(*sub*,*sep*)],*rs*); *sorted-less* (*separators xs*)⟧ $\Longrightarrow$ *sep* < *p*
⟦*split xs p* = (*ls*,(*sub*,*sep*)#*rs*); *sorted-less* (*separators xs*)⟧ $\Longrightarrow$ *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* $\Longrightarrow$
size sub < *Suc* (*size-list* (λ*x*. *Suc* (*size* (*fst x*))) *ts* + *size l*)
using *split-conc*[*of ts y ls* (*sub*,*sep*)#*rs*] **by** *auto*

lemma *leaves-split*: *split ts x* = (*ls*,*rs*) $\Longrightarrow$ *leaves* (*Node ts t*) = *leaves-list ls* @ *leaves-list rs* @ *leaves t*
using *leaves-conc split-conc* **by** *blast*

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*)⟧ $\Longrightarrow$ *ks* = *kls* @ *krs*
⟦*split-list ks p* = (*kls*@[*sep*],*krs*); *sorted-less ks*⟧ $\Longrightarrow$ *sep* < *p*
⟦*split-list ks p* = (*kls*,(*sep*)#*krs*); *sorted-less ks*⟧ $\Longrightarrow$ *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 −
have *distinct* (*insert-list k ks*)
using *assms*(*1*) *assms*(*3*) *strict-sorted-iff* **by** *blast*
then have *length* (*insert-list k ks*) = *card* (*set* (*insert-list k ks*))
by (*simp add*: *distinct-card*)
also have ... = *card* (*set ks* ∪ {*k*})
using *assms*(*2*) **by** *presburger*
also have ... = *card* (*set ks*) + (*if k* ∈ *set ks then 0 else 1*)
by (*cases k* ∈ *set ks*) (*auto simp add*: *insert-absorb*)

also have ... = *length ks* + (*if k* ∈ *set ks then 0 else 1*)
using *assms*(*1*) *distinct-card strict-sorted-iff* **by** *auto*
finally show *?thesis*.
qed

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 −
have *distinct* (*delete-list k ks*)
using *assms*(*1*) *assms*(*3*) *strict-sorted-iff* **by** *blast*
then have *length* (*delete-list k ks*) = *card* (*set* (*delete-list k ks*))
by (*simp add*: *distinct-card*)
also have ... = *card* (*set ks* − {*k*})
using *assms*(*2*) **by** *presburger*
also have ... = *card* (*set ks*) − (*if k* ∈ *set ks then 1 else 0*)
by (*cases k* ∈ *set ks*) (*auto*)
also have ... = *length ks* − (*if k* ∈ *set ks then 1 else 0*)
by (*metis assms*(*1*) *distinct-card strict-sorted-iff*)
finally show *?thesis*.
qed

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*)
using *insert-list-length*[*of ks ins-list k*]
by (*simp add*: *assms set-ins-list sorted-ins-list*)

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*)
using *delete-list-length*[*of ks ins-list k*]
by (*simp add*: *assms set-del-list sorted-del-list*)

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 $\Longrightarrow$ *insert-list x ks* = *ins-list x ks*
sorted-less ks $\Longrightarrow$ *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* $\in$ *set ks then 0 else 1*)
using *insert-list-req*
by (*simp add*: *assms*)

lemma *set-insert-list*[*simp*]:
sorted-less ks $\Longrightarrow$ *set* (*insert-list k ks*) = *set ks* $\cup$ {*k*}
by (*simp add*: *insert-list-req set-ins-list*)

lemma *sorted-insert-list*[*simp*]:
sorted-less ks $\Longrightarrow$ *sorted-less* (*insert-list k ks*)
by (*simp add*: *insert-list-req sorted-ins-list*)

lemma *delete-list-length*[*simp*]:
assumes *sorted-less ks*
shows *length* (*delete-list k ks*) = *length ks* − (*if k* $\in$ *set ks then 1 else 0*)
using *insert-list-req*
by (*simp add*: *assms*)

lemma *set-delete-list*[*simp*]:
sorted-less ks $\Longrightarrow$ *set* (*delete-list k ks*) = *set ks* − {*k*}
by (*simp add*: *insert-list-req set-del-list*)

lemma *sorted-delete-list*[*simp*]:
sorted-less ks $\Longrightarrow$ *sorted-less* (*delete-list k ks*)
by (*simp add*: *insert-list-req sorted-del-list*)

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

7.1 Membership

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

)

Isin proof

thm *isin-simps*

lemma *sorted-ConsD*: *sorted-less* (*y* # *xs*) ⟹ *x* ≤ *y* ⟹ *x* ∉ *set xs*
by (*auto simp*: *sorted-Cons-iff*)

lemma *sorted-snocD*: *sorted-less* (*xs* @ [*y*]) ⟹ *y* ≤ *x* ⟹ *x* ∉ *set xs*
by (*auto simp*: *sorted-snoc-iff*)

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*))
by (*auto simp*: *isin-simps2*)

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 (*cases ls*)
case *Nil*
then have *ts* = *rs*
using *assms* **by** (*auto dest*!: *split-conc*)
then show *?thesis* **by** *simp*
next
case *Cons*
then obtain *ls′ sub sep* **where** *ls-tail-split*: *ls* = *ls′* @ [(*sub*,*sep*)]
by (*metis list.simps*(*3*) *rev-exhaust surj-pair*)
then have *x-sm-sep*: *sep* < *x*
using *split-req*(*2*)[*of ts x ls′ sub sep rs*]
using *Laligned-sorted-separators*[*OF assms*(*1*)]
using *assms sorted-cons sorted-snoc*
by *blast*
moreover have *leaves-split*: *leaves* (*Node ts t*) = *leaves-list ls* @ *leaves-list rs* @ *leaves t*
using *assms*(*3*) *split-tree.leaves-split* **by** *blast*
then show *?thesis*
proof (*cases leaves-list ls*)
case *Nil*
then show *?thesis*
using *leaves-split* **by** *auto*

```
  next
    case Cons
    then obtain leavesls' l' where leaves-tail-split: leaves-list ls = leavesls' @ [l']
      by (metis list.simps(3) rev-exhaust)
    then have l' ≤ sep
    proof −
      have l' ∈ set (leaves-list ls)
        using leaves-tail-split by force
      then have l' ∈ set (leaves (Node ls' sub))
        using ls-tail-split
        by auto
      moreover have Laligned (Node ls' sub) sep
        using assms split-conc[OF assms(3)] Cons ls-tail-split
        using Laligned-split-left[of ls' sub sep rs t u]
        by simp
      ultimately show ?thesis
        using Laligned-leaves-inbetween[of Node ls' sub sep]
        by blast
    qed
    then show ?thesis
      using assms(2) ls-tail-split leaves-tail-split leaves-split x-sm-sep
      using isin-sorted[of leavesls' l' leaves-list rs @ leaves t x]
      by auto
  qed
qed

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 −
  from assms have x ≤ sep
  proof −
    from assms have sorted-less (separators ts)
    by (meson Laligned-sorted-inorder sorted-cons sorted-inorder-separators sorted-snoc)
    then show ?thesis
      using split-req(3)
      using assms
      by fastforce
  qed
  moreover have leaves-split: leaves (Node ts t) = leaves-list ls @ leaves sub @
leaves-list rs @ leaves t
    using split-conc[OF assms(1)] by auto
  ultimately show ?thesis
  proof (cases leaves-list rs @ leaves t)
    case Nil
    then show ?thesis
      using leaves-split by auto
```

```
  next
    case (Cons r' rs')
    then have sep < r'
        by (metis Laligned-split-right aligned-leaves-inbetween assms(1) assms(3)
leaves.simps(2) list.set-intros(1) split-set.split-conc split-set-axioms)
    then have  x < r'
      using ‹x ≤ sep› by auto
    moreover have sorted-less (leaves-list ((sub,sep)#rs) @ leaves t)
      using assms sorted-wrt-append split-conc
      by fastforce
    ultimately show ?thesis
      using isin-sorted[of leaves sub r' rs' x] Cons
      by auto
  qed
qed


theorem isin-set-inorder:
  assumes sorted-less (leaves t)
    and aligned l t u
  shows isin t x = (x ∈ set (leaves t))
  using assms
proof(induction t x arbitrary: l u rule: isin.induct)
  case (2 ts t x)
  then obtain ls rs where list-split: split ts x = (ls, rs)
    by (meson surj-pair)
  then have list-conc: ts = ls @ rs
    using split-conc by auto
  show ?case
  proof (cases rs)
    case Nil
    then have isin (Node ts t) x = isin t x
      by (simp add: list-split)
    also have ... = (x ∈ set (leaves t))
      using 2.IH(1)[of ls rs] list-split Nil
      using 2.prems sorted-leaves-induct-last align-last'
      by metis
    also have ... = (x ∈ set (leaves (Node ts t)))
      using isin-sorted-split
      using 2.prems list-split list-conc Nil
       by (metis aligned-imp-Laligned leaves.simps(2) leaves-conc same-append-eq
self-append-conv)
    finally show ?thesis .
  next
    case (Cons a list)
    then obtain sub sep where a-split: a = (sub,sep)
      by (cases a)
      then have isin (Node ts t) x = isin sub x
        using list-split Cons a-split
```

```
      by auto
    also have ... = (x ∈ set (leaves sub))
      using 2.IH(2)[of ls rs (sub,sep) list sub sep]
    using 2.prems a-split list-conc list-split local.Cons sorted-leaves-induct-subtree
          align-sub
      by (metis in-set-conv-decomp)
    also have ... = (x ∈ set (leaves (Node ts t)))
      using isin-sorted-split
      using isin-sorted-split-right 2.prems list-split Cons a-split
      using aligned-imp-Laligned by blast
    finally show ?thesis .
  qed
qed (auto simp add: insert-list-req)


theorem isin-set-Linorder:
  assumes sorted-less (leaves t)
    and Laligned t u
  shows isin t x = (x ∈ set (leaves t))
  using assms
proof(induction t x arbitrary: u rule: isin.induct)
  case (2 ts t x)
  then obtain ls rs where list-split: split ts x = (ls, rs)
    by (meson surj-pair)
  then have list-conc: ts = ls @ rs
    using split-conc by auto
  show ?case
  proof (cases rs)
    case Nil
    then have isin (Node ts t) x = isin t x
      by (simp add: list-split)
    also have ... = (x ∈ set (leaves t))
       by (metis 2.IH(1) 2.prems(1) 2.prems(2) Lalign-Llast list-split local.Nil
sorted-leaves-induct-last)
    also have ... = (x ∈ set (leaves (Node ts t)))
      using isin-sorted-split
      using 2.prems list-split list-conc Nil
      by simp
    finally show ?thesis .
  next
    case (Cons a list)
    then obtain sub sep where a-split: a = (sub,sep)
      by (cases a)
      then have isin (Node ts t) x = isin sub x
        using list-split Cons a-split
        by auto
      also have ... = (x ∈ set (leaves sub))
        using 2.IH(2)[of ls rs (sub,sep) list sub sep]
      using 2.prems a-split list-conc list-split local.Cons sorted-leaves-induct-subtree
```

align-sub
by (*metis Lalign-Llast Laligned-split-left*)
also have ... = (*x* ∈ *set* (*leaves* (*Node ts t*)))
using *isin-sorted-split*
using *isin-sorted-split-right 2.prems list-split Cons a-split*
by *simp*
finally show *?thesis* **.**
qed
qed (*auto simp add*: *insert-list-req*)

corollary *isin-set-Linorder-top*:
assumes *sorted-less* (*leaves t*)
and *Laligned t top*
shows *isin t x* = (*x* ∈ *set* (*leaves t*))
using *assms isin-set-Linorder*
by *simp*

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 $\Rightarrow$ 'a *bplustree* **where**
  $tree_i$ (T_i *sub*) = *sub* |
  $tree_i$ (Up_i *l a r*) = (*Node* [(*l*,*a*)] *r*)

fun *insert*::*nat* $\Rightarrow$ 'a $\Rightarrow$ 'a *bplustree* $\Rightarrow$ 'a *bplustree* **where**
  insert k x t = $tree_i$ (*ins k x t*)

7.3 Proofs of functional correctness

lemma $node_i$*-ti-simp*: $node_i$ *k ts t* = T_i *x* $\Longrightarrow$ *x* = *Node ts t*
  apply (*cases length ts* $\leq$ *2*$*$*k*)
  apply (*auto split*!: *list.splits prod.splits*)
  done

lemma $Lnode_i$*-ti-simp*: $Lnode_i$ *k ts* = T_i *x* $\Longrightarrow$ *x* = *Leaf ts*
  apply (*cases length ts* $\leq$ *2*$*$*k*)
  apply (*auto split*!: *list.splits*)
  done

lemma *split-set*:
  assumes *split ts z* = (*ls*,(*a*,*b*)#*rs*)
  shows (*a*,*b*) $\in$ *set ts*
    and (*x*,*y*) $\in$ *set ls* $\Longrightarrow$ (*x*,*y*) $\in$ *set ts*
    and (*x*,*y*) $\in$ *set rs* $\Longrightarrow$ (*x*,*y*) $\in$ *set ts*
    and *set ls* $\cup$ *set rs* $\cup$ {(*a*,*b*)} = *set ts*
    and $\exists$ *x* $\in$ *set ts*. *b* $\in$ *Basic-BNFs.snds x*
  using *split-conc assms* **by** *fastforce*+

lemma *split-length*:
  split ts x = (*ls*, *rs*) $\Longrightarrow$ *length ls* + *length rs* = *length ts*
  by (*auto dest*: *split-conc*)

lemma $node_i$*-cases*: *length xs* $\leq$ *k* $\vee$ ($\exists$ *ls sub sep rs*. *split-half xs* = (*ls*@[(*sub*,*sep*)],*rs*))
proof −
  have ¬ *length xs* $\leq$ *k* $\Longrightarrow$ *length xs* $\geq$ *1*
    by *linarith*
  then show *?thesis*
    using *split-half-not-empty*
    by *fastforce*
qed

lemma $Lnode_i$*-cases*: *length xs* $\leq$ *k* $\vee$ ($\exists$ *ls sep rs*. *split-half xs* = (*ls*@[*sep*],*rs*))
proof −
  have ¬ *length xs* $\leq$ *k* $\Longrightarrow$ *length xs* $\geq$ *1*
    by *linarith*

then show *?thesis*
using *split-half-not-empty*
by *fastforce*
qed

lemma *root-order-tree$_i$*: *root-order-up$_i$* (*Suc k*) *t* = *root-order* (*Suc k*) (*tree$_i$ t*)
apply (*cases t*)
apply *auto*
done

lemma *length-take-left*: *length* (*take* ((*length ts* + *1*) *div 2*) *ts*) = (*length ts* + *1*) *div 2*
apply (*cases ts*)
apply *auto*
done

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 (*cases length ts* ≤ *2*∗*k*)
case *True*
then show *?thesis*
using *assms*
by (*simp add*: *node$_i$.simps*)
next
case *False*
then obtain *ls sub sep rs* **where** *split-half-ts*:
take ((*length ts* + *1*) *div 2*) *ts* = *ls*@[(*sub*,*sep*)]
using *split-half-not-empty*[*of ts*]
by *auto*
then have *length-ls*: *length ls* = (*length ts* + *1*) *div 2* − *1*
by (*metis One-nat-def add-diff-cancel-right′ add-self-div-2 bits-1-div-2 length-append length-take-left list.size*(*3*) *list.size*(*4*) *odd-one odd-succ-div-two*)
also have ... ≤ (*4*∗*k* + *1*) *div 2*
using *assms*(*2*) **by** *simp*
also have ... = *2*∗*k*
by *auto*
finally have *length ls* ≤ *2*∗*k*
by *simp*
moreover have *length ls* ≥ *k*
using *False length-ls* **by** *simp*
moreover have *set* (*ls*@[(*sub*,*sep*)]) ⊆ *set ts*
by (*metis split-half-ts*(*1*) *set-take-subset*)
ultimately have *o-r*: *order k* (*Node ls sub*)
using *split-half-ts assms* **by** *auto*
have

butlast (*take* ((*length ts* + *1*) *div 2*) *ts*) = *ls*
last (*take* ((*length ts* + *1*) *div 2*) *ts*) = (*sub*,*sep*)
using *split-half-ts* **by** *auto*
then show *?thesis*
using *o-r assms set-drop-subset*[*of* - *ts*]
by (*auto simp add*: *False split-half-ts split*!: *prod.splits*)
qed

lemma *node*$_i$*-order-helper*:
assumes *length ts* ≥ *k*
and *length ts* ≤ *4*∗*k*+*1*
and ∀ *x* ∈ *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 (*cases length ts* ≤ *2*∗*k*)
case *True*
then show *?thesis*
using *assms*
by (*simp add*: *node*$_i$*.simps*)
next
case *False*
then obtain *sub sep ls* **where**
take ((*length ts* + *1*) *div 2*) *ts* = *ls*@[(*sub*,*sep*)]
using *split-half-not-empty*[*of ts*]
by *fastforce*
then show *?thesis*
using *assms* **by** *simp*
qed

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*)
apply(*cases node*$_i$ *k ts t*)
using *node*$_i$*-root-order node*$_i$*-order-helper assms* **apply** *fastforce*
by (*metis* (*full-types*) *assms le-0-eq nat-le-linear node*$_i$*.elims node*$_i$*-root-order order-up*$_i$*.simps*(*2*) *root-order-up*$_i$*.simps*(*2*) *up*$_i$*.simps*(*4*) *verit-comp-simplify1*(*3*))

lemma *Lnode*$_i$*-root-order*:
assumes *length ts* > *0*
and *length ts* ≤ *4*∗*k*
shows *root-order-up*$_i$ *k* (*Lnode*$_i$ *k ts*)
proof (*cases length ts* ≤ *2*∗*k*)
case *True*
then show *?thesis*

using *assms*
by *simp*
next
case *False*
then obtain *ls sep rs* **where** *split-half-ts*:
take ((*length ts* + *1*) *div 2*) *ts* = *ls*@[*sep*]
drop ((*length ts* + *1*) *div 2*) *ts* = *rs*
using *split-half-not-empty*[*of ts*]
by *auto*
then have *length-ls*: *length ls* = ((*length ts* + *1*) *div 2*) − *1*
by (*metis One-nat-def add-diff-cancel-right' add-self-div-2 bits-1-div-2 length-append length-take-left list.size*(*3*) *list.size*(*4*) *odd-one odd-succ-div-two*)
also have ... < (*4*∗*k* + *1*) *div 2*
using *assms*(*2*)
by (*smt* (*z3*) *Groups.add-ac*(*2*) *One-nat-def split-half-ts add.right-neutral diff-is-0-eq' div-le-mono le-add-diff-inverse le-neq-implies-less length-append length-take-left less-add-Suc1 less-imp-diff-less list.size*(*4*) *nat-le-linear not-less-eq plus-nat.simps*(*2*))
also have ... = *2*∗*k*
by *auto*
finally have *length ls* < *2*∗*k*
by *simp*
moreover have *length ls* ≥ *k*
using *False length-ls* **by** *simp*
ultimately have *o-l*: *order k* (*Leaf* (*ls*@[*sep*]))
using *set-take-subset assms split-half-ts*
by *fastforce*
then show *?thesis*
using *assms split-half-ts False*
by *auto*
qed

lemma $Lnode_i$*-order-helper*:
assumes *length ts* ≥ *k*
and *length ts* ≤ *4*∗*k*+*1*
shows *case* (*Lnode*$_i$ *k ts*) *of* T_i *t* ⇒ *order k t* | - ⇒ *True*
proof (*cases length ts* ≤ *2*∗*k*)
case *True*
then show *?thesis*
using *assms*
by (*simp add*: $node_i$*.simps*)
next
case *False*
then obtain *sep ls* **where**
take ((*length ts* + *1*) *div 2*) *ts* = *ls*@[*sep*]
using *split-half-not-empty*[*of ts*]
by *fastforce*
then show *?thesis*
using *assms* **by** *simp*
qed

lemma *Lnode$_i$-order*:
assumes *length ts* $\geq$ *k*
and *length ts* $\leq$ *4∗k*
shows *order-up$_i$ k* (*Lnode$_i$ k ts*)
apply(*cases Lnode$_i$ k ts*)
apply (*metis Lnode$_i$-order-helper One-nat-def add.right-neutral add-Suc-right assms*(*1*) *assms*(*2*) *le-imp-less-Suc less-le order-up$_i$.simps*(*1*) *up$_i$.simps*(*5*))
by (*metis Lnode$_i$.elims Lnode$_i$-root-order assms*(*1*) *assms*(*2*) *diff-is-0-eq′ le-0-eq le-add-diff-inverse mult-2 order-up$_i$.simps*(*2*) *root-order-up$_i$.simps*(*2*) *up$_i$.simps*(*3*) *verit-comp-simplify1*(*3*))

lemma *ins-order*:
k > *0* $\Longrightarrow$ *sorted-less* (*leaves t*) $\Longrightarrow$ *order k t* $\Longrightarrow$ *order-up$_i$ k* (*ins k x t*)
proof(*induction k x t rule*: *ins.induct*)
case (*1 k x ts*)
then show *?case*
by *auto*
next
case (*2 k x ts t*)
then obtain *ls rs* **where** *split-res*: *split ts x* = (*ls*, *rs*)
by (*meson surj-pair*)
then have *split-app*: *ts* = *ls*@*rs*
using *split-conc*
by *simp*

show *?case*
proof (*cases rs*)
case *Nil*
then have *order-up$_i$ k* (*ins k x t*)
using *2 split-res sorted-leaves-induct-last*
by *auto*
then show *?thesis*
using *Nil 2 split-app split-res Nil node$_i$-order*
by (*auto split*!: *up$_i$.splits simp del*: *node$_i$.simps*)
next
case (*Cons a list*)
then obtain *sub sep* **where** *a-prod*: *a* = (*sub*, *sep*)
by (*cases a*)
then have *sorted-less* (*leaves sub*)
using *2*(*4*) *Cons sorted-leaves-induct-subtree split-app*
by *blast*
then have *order-up$_i$ k* (*ins k x sub*)
using *2.IH*(*2*) *2.prems a-prod local.Cons split-app split-res*
by *auto*
then show *?thesis*
using *2 split-app Cons length-append node$_i$-order*[*of k ls*@-#-#*list*] *a-prod*

split-res
by (*auto split*!: up_i.*splits simp del*: $node_i$.*simps simp add*: *order-impl-root-order*)
qed
qed

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(*cases t*)
case (*Leaf ks*)
then show *?thesis*
using *assms* **by** (*auto simp add*: *Lnode*$_i$*-order min-absorb2*)
next
case (*Node ts t*)
then obtain *ls rs* **where** *split-res*: *split ts x* = (*ls*, *rs*)
by (*meson surj-pair*)
then have *split-app*: *ls*@*rs* = *ts*
using *split-conc*
by *fastforce*

show *?thesis*
proof (*cases rs*)
case *Nil*
then have *order-up*$_i$ *k* (*ins k x t*)
using *Node assms split-res sorted-leaves-induct-last*
using *ins-order*[*of k t*]
by *auto*
then show *?thesis*
using *Nil Node split-app split-res assms node*$_i$*-root-order*
by (*auto split*!: up_i.*splits simp del*: $node_i$.*simps simp add*: *order-impl-root-order*)
next
case (*Cons a list*)
then obtain *sub sep* **where** *a-prod*: *a* = (*sub*, *sep*)
by (*cases a*)
then have *sorted-less* (*leaves sub*)
using *Node assms*(*2*) *local.Cons sorted-leaves-induct-subtree split-app*
by *blast*
then have *order-up*$_i$ *k* (*ins k x sub*)
using *Node a-prod assms ins-order local.Cons split-app*
by *auto*
then show *?thesis*
using *assms split-app Cons length-append Node node*$_i$*-root-order a-prod split-res*
by (*auto split*!: up_i.*splits simp del*: $node_i$.*simps simp add*: *order-impl-root-order*)
qed
qed

lemma *height-list-split*: *height-up$_i$* (*Up$_i$* (*Node ls a*) *b* (*Node rs t*)) = *height* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*)
by (*induction ls*) (*auto simp add*: *max.commute*)

lemma *node$_i$-height*: *height-up$_i$* (*node$_i$ k ts t*) = *height* (*Node ts t*)
proof(*cases length ts* ≤ *2*∗*k*)
case *False*
then obtain *ls sub sep rs* **where**
split-half-ts: *split-half ts* = (*ls*@[(*sub*,*sep*)], *rs*)
by (*meson node$_i$-cases*)
then have *node$_i$ k ts t* = *Up$_i$* (*Node ls* (*sub*)) *sep* (*Node rs t*)
using *False* **by** *simp*
then have *height-up$_i$* (*node$_i$ k ts t*) = *height* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
by (*metis height-list-split*)
also have ... = *height* (*Node ts t*)
by (*metis* (*no-types*, *lifting*) *Pair-inject append-Cons append-eq-append-conv2 append-take-drop-id self-append-conv split-half.simps split-half-ts*)
finally show *?thesis*.
qed *simp*

lemma *Lnode$_i$-height*: *height-up$_i$* (*Lnode$_i$ k xs*) = *height* (*Leaf xs*)
by (*auto*)

lemma *bal-up$_i$-tree*: *bal-up$_i$ t* = *bal* (*tree$_i$ t*)
apply(*cases t*)
apply *auto*
done

lemma *bal-list-split*: *bal* (*Node* (*ls*@(*a*,*b*)#*rs*) *t*) ⟹ *bal-up$_i$* (*Up$_i$* (*Node ls a*) *b* (*Node rs t*))
by (*auto simp add*: *image-constant-conv*)

lemma *node$_i$-bal*:
assumes *bal* (*Node ts t*)
shows *bal-up$_i$* (*node$_i$ k ts t*)
using *assms*
proof(*cases length ts* ≤ *2*∗*k*)
case *False*
then obtain *ls sub sep rs* **where**
split-half-ts: *split-half ts* = (*ls*@[(*sub*,*sep*)], *rs*)
by (*meson node$_i$-cases*)
then have *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
using *assms append-take-drop-id*[**where** *n*=(*length ts* + *1*) *div 2* **and** *xs*=*ts*]
by *auto*
then show *?thesis*
using *split-half-ts assms False*
by (*auto simp del*: *bal.simps bal-up$_i$.simps dest*!: *bal-list-split*[*of ls sub sep rs t*])

qed *simp*

lemma *node$_i$-aligned*:
assumes *aligned l* (*Node ts t*) *u*
shows *aligned-up$_i$ l* (*node$_i$ k ts t*) *u*
using *assms*
proof (*cases length ts* ≤ *2*∗*k*)
case *False*
then obtain *ls sub sep rs* **where**
split-half-ts: *split-half ts* = (*ls*@[(*sub*,*sep*)], *rs*)
by (*meson node$_i$-cases*)
then have *aligned l* (*Node ls sub*) *sep*
by (*metis aligned-split-left append.assoc append-Cons append-take-drop-id assms prod.sel*(*1*) *split-half.simps*)
moreover have *aligned sep* (*Node rs t*) *u*
by (*smt* (*z3*) *Pair-inject aligned-split-right append.assoc append-Cons append-Nil2 append-take-drop-id assms same-append-eq split-half.simps split-half-ts*)
ultimately show *?thesis*
using *split-half-ts False* **by** *auto*
qed *simp*

lemma *node$_i$-Laligned*:
assumes *Laligned* (*Node ts t*) *u*
shows *Laligned-up$_i$* (*node$_i$ k ts t*) *u*
using *assms*
proof (*cases length ts* ≤ *2*∗*k*)
case *False*
then obtain *ls sub sep rs* **where**
split-half-ts: *split-half ts* = (*ls*@[(*sub*,*sep*)], *rs*)
by (*meson node$_i$-cases*)
then have *Laligned* (*Node ls sub*) *sep*
by (*metis Laligned-split-left append.assoc append-Cons assms split-half-conc*)
moreover have *aligned sep* (*Node rs t*) *u*
by (*metis Laligned-split-right append.assoc append-Cons append-Nil2 assms same-append-eq split-half-conc split-half-ts*)
ultimately show *?thesis*
using *split-half-ts False* **by** *auto*
qed *simp*

lemma *length-right-side*: *length xs* > *1* ⟹ *length* (*drop* ((*length xs* + *1*) *div 2*) *xs*) > *0*
by *auto*

lemma *Lnode$_i$-aligned*:
assumes *aligned l* (*Leaf ks*) *u*
and *sorted-less ks*
and *k* > *0*
shows *aligned-up$_i$ l* (*Lnode$_i$ k ks*) *u*

```
  using assms
proof (cases length ks ≤ 2∗k)
  case False
  then obtain ls sep rs where split-half-ts:
    take ((length ks + 1) div 2) ks = ls@[sep]
    drop ((length ks + 1) div 2) ks = rs
    using split-half-not-empty[of ks]
    by auto
  moreover have sorted-less (ls@[sep])
    by (metis append-take-drop-id assms(2) sorted-wrt-append split-half-ts(1))
  ultimately have aligned l (Leaf (ls@[sep])) sep
    using split-half-conc[of ks ls@[sep] rs] assms sorted-snoc-iff[of ls sep]
    by auto
  moreover have aligned sep (Leaf rs) u
  proof −
    have length rs > 0
      using False assms(3) split-half-ts(2) by fastforce
    then obtain sep′ rs′ where rs = sep′ # rs′
      by (cases rs) auto
    moreover have sep < sep′
    by (metis append-take-drop-id assms(2) calculation in-set-conv-decomp sorted-mid-iff
sorted-snoc-iff split-half-ts(1) split-half-ts(2))
    moreover have sorted-less rs
      by (metis append-take-drop-id assms(2) sorted-wrt-append split-half-ts(2))
    ultimately show ?thesis
      using split-half-ts split-half-conc[of ks ls@[sep] rs] assms
      by auto
  qed
  ultimately show ?thesis
    using split-half-ts False by auto
qed simp

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)
  by simp

lemma ins-height: height-up_i (ins k x t) = height t
proof(induction k x t rule: ins.induct)
  case (2 k x ts t)
  then obtain ls rs where split-list: split ts x = (ls,rs)
    by (meson surj-pair)
  then have split-append: ts = ls@rs
    using split-conc
    by auto
  then show ?case
  proof (cases rs)
    case Nil
    then have height-sub: height-up_i (ins k x t) = height t
      using 2 by (simp add: split-list)
```

```
    then show ?thesis
    proof (cases ins k x t)
      case (T_i a)
      then have height (Node ts t) = height (Node ts a)
        using height-sub
        by simp
      then show ?thesis
        using T_i Nil split-list split-append
        by simp
    next
      case (Up_i l a r)
      then have height (Node ls t) = height (Node (ls@[(l,a)]) r)
        using height-bplustree-order height-sub by (induction ls) auto
      then show ?thesis using 2 Nil split-list Up_i split-append
        by (simp del: node_i.simps add: node_i-height)
    qed
  next
    case (Cons a list)
    then obtain sub sep where a-split: a = (sub,sep)
      by (cases a) auto
    then have height-sub: height-up_i (ins k x sub) = height sub
      by (metis 2.IH(2) a-split Cons split-list)
    then show ?thesis
    proof (cases ins k x sub)
      case (T_i a)
      then have height a = height sub
        using height-sub by auto
     then have height (Node (ls@(sub,sep)#rs) t) = height (Node (ls@(a,sep)#rs)
t)
        by auto
      then show ?thesis
        using T_i height-sub Cons 2 split-list a-split split-append
        by (auto simp add: image-Un max.commute finite-set-ins-swap)
    next
      case (Up_i l a r)
    then have height (Node (ls@(sub,sep)#list) t) = height (Node (ls@(l,a)#(r,sep)#list)
t)
        using height-up_i-merge height-sub
        by fastforce
      then show ?thesis
        using Up_i Cons 2 split-list a-split split-append
      by (auto simp add: node_i.simps simp add: node_i-height image-Un max.commute
finite-set-ins-swap)
    qed
  qed
qed simp
```

lemma *ins-bal*: *bal t* $\Longrightarrow$ *bal-up*$_i$ (*ins k x t*)
proof(*induction k x t rule*: *ins.induct*)
case (*2 k x ts t*)
then obtain *ls rs* **where** *split-res*: *split ts x* = (*ls*, *rs*)
by (*meson surj-pair*)
then have *split-app*: *ts* = *ls*@*rs*
using *split-conc*
by *fastforce*

show *?case*
proof (*cases rs*)
case *Nil*
then show *?thesis*
proof (*cases ins k x t*)
case (*T*$_i$ *a*)
then have *bal* (*Node ls a*) **unfolding** *bal.simps*
by (*metis 2.IH*(*1*) *2.prems append-Nil2 bal.simps*(*2*) *bal-up*$_i$*.simps*(*1*)
height-up$_i$*.simps*(*1*) *ins-height local.Nil split-app split-res*)
then show *?thesis*
using *Nil T*$_i$ *2 split-res*
by *simp*
next
case (*Up*$_i$ *l a r*)
then have
(∀ *x*∈*set* (*subtrees* (*ls*@[(*l*,*a*)])). *bal x*)
(∀ *x*∈*set* (*subtrees ls*). *height r* = *height x*)
using *2 Up*$_i$ *Nil split-res split-app*
by *simp-all* (*metis height-up*$_i$*.simps*(*2*) *ins-height max-def*)
then show *?thesis* **unfolding** *ins.simps*
using *Up*$_i$ *Nil 2 split-res*
by (*simp del*: *node*$_i$*.simps add*: *node*$_i$*-bal*)
qed
next
case (*Cons a list*)
then obtain *sub sep* **where** *a-prod*: *a* = (*sub*, *sep*) **by** (*cases a*)
then have *bal-up*$_i$ (*ins k x sub*) **using** *2 split-res*
using *a-prod local.Cons split-app* **by** *auto*
show *?thesis*
proof (*cases ins k x sub*)
case (*T*$_i$ *x1*)
then have *height x1* = *height t*
by (*metis 2.prems a-prod add-diff-cancel-left' bal-split-left*(*1*) *bal-split-left*(*2*)
height-bal-tree height-up$_i$*.simps*(*1*) *ins-height local.Cons plus-1-eq-Suc split-app*)
then show *?thesis*
using *split-app Cons T*$_i$ *2 split-res a-prod*
by *auto*
next
case (*Up*$_i$ *l a r*)

```
    then have
      ∀ x ∈ set (subtrees (ls@(l,a)#(r,sep)#list)). bal x
      using Up_i split-app Cons 2 ‹bal-up_i (ins k x sub)› by auto
    moreover have ∀ x ∈ set (subtrees (ls@(l,a)#(r,sep)#list)). height x = height
t
      using Up_i split-app Cons 2 ‹bal-up_i (ins k x sub)› ins-height split-res a-prod
      apply auto
      by (metis height-up_i.simps(2) sup.idem sup-nat-def)
    ultimately show ?thesis using Up_i Cons 2 split-res a-prod
      by (simp del: node_i.simps add: node_i-bal)
  qed
 qed
qed simp
```

```
lemma node_i-leaves: leaves-up_i (node_i k ts t) = leaves (Node ts t)
proof (cases length ts ≤ 2*k)
 case False
 then obtain ls sub sep rs where
  split-half-ts: split-half ts = (ls@[(sub,sep)], rs)
  by (meson node_i-cases)
 then have leaves-up_i (node_i k ts t) = leaves-list ls @ leaves sub @ leaves-list rs
@ leaves t
  using False by auto
 also have ... = leaves (Node ts t)
  using split-half-ts split-half-conc[of ts ls@[(sub,sep)] rs] by auto
 finally show ?thesis.
qed simp

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)
  apply (metis leaves-up_i.simps(1) node_i-leaves)
 by (metis leaves-up_i.simps(2) node_i-leaves)

lemma Lnode_i-leaves: leaves-up_i (Lnode_i k xs) = leaves (Leaf xs)
proof (cases length xs ≤ 2*k)
 case False
 then obtain ls sub sep rs where
  split-half-ts: split-half xs = (ls@[sep], rs)
  by (meson Lnode_i-cases)
 then have leaves-up_i (Lnode_i k xs) = ls @ sep # rs
  using False by auto
 also have ... = leaves (Leaf xs)
  using split-half-ts split-half-conc[of xs ls@[sep] rs] by auto
 finally show ?thesis.
qed simp
```

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)
apply (*metis leaves-up$_i$.simps(1) Lnode$_i$-leaves*)
by (*metis leaves-up$_i$.simps(2) Lnode$_i$-leaves*)

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 (*cases ls*)
case *Nil*
then show *?thesis*
using *assms* **by** (*auto dest*!: *split-conc*)
next
case *Cons*
then obtain *ls′ sub sep* **where** *ls-tail-split*: *ls = ls′ @ [(sub,sep)]*
by (*metis list.distinct(1) rev-exhaust surj-pair*)
have *sorted-inorder*: *sorted-less (inorder (Node ts t))*
using *Laligned-sorted-inorder assms(1) sorted-cons sorted-snoc* **by** *blast*
moreover have *x-sm-sep*: *sep < x*
using *split-req(2)[of ts x ls′ sub sep rs]*
using *sorted-inorder-separators[of ts t] sorted-inorder*
using *assms ls-tail-split*
by *auto*
moreover have *leaves-split*: *leaves (Node ts t) = leaves-list ls @ leaves-list rs @ leaves t*
using *assms(3) split-tree.leaves-split* **by** *blast*
then show *?thesis*
proof (*cases leaves-list ls*)
case *Nil*
then show *?thesis*
by (*metis append-self-conv2 leaves-split*)
next
case *Cons*
then obtain *leavesls′ l′* **where** *leaves-tail-split*: *leaves-list ls = leavesls′ @ [l′]*
by (*metis list.simps(3) rev-exhaust*)
then have *l′ ≤ sep*
proof −
have *l′ ∈ set (leaves-list ls)*
using *leaves-tail-split* **by** *force*
then have *l′ ∈ set (leaves (Node ls′ sub))*

```
      using ls-tail-split
      by auto
    moreover have Laligned (Node ls' sub) sep
      using assms split-conc[OF assms(3)] Cons ls-tail-split
      using Laligned-split-left[of ls' sub sep rs t u]
      by simp
    ultimately show ?thesis
      using Laligned-leaves-inbetween[of Node ls' sub sep]
      by blast
  qed
  moreover have sorted-less (leaves-list ls)
    using assms(2) leaves-split sorted-wrt-append by auto
  ultimately show ?thesis
    using assms(2) ls-tail-split leaves-tail-split leaves-split x-sm-sep
    using ins-list-sorted[of leavesls' l' x leaves-list rs@leaves t]
    by auto
 qed
qed

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 −
 from assms have x-sm-sep: x ≤ sep
 proof −
  from assms have sorted-less (separators ts)
    using Laligned-sorted-separators sorted-cons sorted-snoc by blast
  then show ?thesis
    using split-req(3)
    using assms
    by blast
 qed
 then show ?thesis
 proof (cases leaves-list rs @ leaves t)
  case Nil
  moreover have leaves-list ((sub,sep)#rs) @ leaves t = leaves sub @ leaves-list
rs @ leaves t
    by simp
  ultimately show ?thesis
    by (metis self-append-conv)
 next
  case (Cons r' rs')
  then have sep < r'
     by (metis aligned-leaves-inbetween Laligned-split-right assms(1) assms(3)
leaves.simps(2) list.set-intros(1) split-set.split-conc split-set-axioms)
  then have  x < r'
```

```
    using ‹x ≤ sep› by auto
  moreover have sorted-less (leaves sub @ [r'])
  proof −
    have sorted-less (leaves-list ls @ leaves sub @ leaves-list rs @ leaves t)
      using assms(1) assms(2) split-tree.leaves-split split-set-axioms by fastforce
    then show ?thesis
      using assms
      by (metis Cons sorted-mid-iff sorted-wrt-append)
  qed
  ultimately show ?thesis
    using ins-list-sorted[of leaves sub r' x rs'] Cons
    by auto
 qed
qed



lemma ins-list-idem-eq-isin: sorted-less xs ⟹ x ∈ set xs ⟷ (ins-list x xs = xs)
 apply(induction xs)
  apply auto
 done

lemma ins-list-contains-idem: ⟦sorted-less xs; x ∈ set xs⟧ ⟹ (ins-list x xs = xs)
 using ins-list-idem-eq-isin by auto

lemma aligned-insert-list: sorted-less ks ⟹ l < x ⟹ x ≤ u ⟹ aligned l (Leaf
ks) u ⟹ aligned l (Leaf (insert-list x ks)) u
 using insert-list-req
 by (simp add: set-ins-list)

lemma align-subst-two: aligned l (Node (ts@[(sub,sep)]) t) u ⟹ aligned sep lt a
⟹ aligned a rt u ⟹ aligned l (Node (ts@[(sub,sep),(lt,a)]) rt) u
 apply(induction ts arbitrary: l)
 apply auto
 done

lemma align-subst-three: aligned l (Node (ls@(subl,sepl)#(subr,sepr)#rs) t) u ⟹
aligned sepl lt a ⟹ aligned a rt sepr ⟹ aligned l (Node (ls@(subl,sepl)#(lt,a)#(rt,sepr)#rs)
t) u
 apply(induction ls arbitrary: l)
 apply auto
 done


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 ≤ u
  shows (leaves-up_i (ins k x t)) = ins-list x (leaves t) ∧ aligned-up_i l (ins k x t) u
  using assms
proof(induction k x t arbitrary: l u rule: ins.induct)
  case (1 k x ks)
  then show ?case
  proof (safe, goal-cases)
    case -: 1
    then show ?case
      using 1 insert-list-req by auto
  next
    case 2
    from 1 have aligned l (Leaf (insert-list x ks)) u
      by (metis aligned-insert-list leaves.simps(1))
    moreover have sorted-less (insert-list x ks)
      using 1.prems(3) split-set.insert-list-req split-set-axioms
      by auto
    ultimately show ?case
      using Lnode_i-aligned[of l insert-list x ks u k] 1
      by (auto simp del: Lnode_i.simps split-half.simps)
  qed
next
  case (2 k x ts t)
  then obtain ls rs where list-split: split ts x = (ls,rs)
    by (cases split ts x)
  then have list-conc: ts = ls@rs
    using split-set.split-conc split-set-axioms by blast
  then show ?case
  proof (cases rs)
    case Nil
    then obtain ts' sub' sep' where ts = ts'@[(sub',sep')]
      apply(cases ts)
      using 2 list-conc Nil apply(simp)
      by (metis isin.cases list.distinct(1) rev-exhaust)
    have IH: leaves-up_i (ins k x t) = ins-list x (leaves t) ∧ aligned-up_i sep' (ins k
x t) u
      proof −

        note 2.IH(1)[OF sym[OF list-split] Nil 2.prems(1), of sep' u]
        have sorted-less (leaves t)
          using 2.prems(3) sorted-leaves-induct-last by blast
        moreover have sep' < x
          using split-req[of ts x] list-split
           by (metis 2.prems(2) ‹ts = ts' @ [(sub', sep')]› aligned-sorted-separators
local.Nil self-append-conv sorted-cons sorted-snoc)
        moreover have aligned sep' t u
```

```
        using 2.prems(2) ‹ts = ts' @ [(sub', sep')]› align-last by blast
      ultimately show ?thesis
        using  2.IH(1)[OF sym[OF list-split] Nil 2.prems(1), of sep' u]
        using 2.prems list-split local.Nil sorted-leaves-induct-last
        using order-impl-root-order
        by auto
    qed
  show ?thesis
  proof (cases ins k x t)
    case (T_i a)
    have IH: leaves a = ins-list x (leaves t) ∧ aligned sep' a u
      using IH T_i by force
    show ?thesis
    proof(safe, goal-cases)
      case 1
      have leaves-up_i (ins k x (Node ts t)) = leaves-list ls @ leaves a
        using list-split T_i Nil by (auto simp add: list-conc)
      also have ... = leaves-list ls @ (ins-list x (leaves t))
        by (simp add: IH)
      also have ... = ins-list x (leaves (Node ts t))
        using ins-list-split
        using 2.prems list-split Nil
     by (metis aligned-imp-Laligned append-self-conv2 concat.simps(1) list.simps(8))
      finally show ?case .
    next
      case 2
      have aligned-up_i l (ins k x (Node ts t)) u = aligned l (Node ts a) u
        using Nil T_i list-split list-conc by simp
      moreover have aligned l (Node ts a) u
        using 2.prems(2)
        by (metis IH ‹ts = ts' @ [(sub', sep')]› aligned-subst-last)
      ultimately show ?case
        by auto
    qed
  next
    case (Up_i lt a rt)
     then have IH:leaves-up_i (Up_i lt a rt) = ins-list x (leaves t) ∧ aligned-up_i
sep' (Up_i lt a rt) u
      using IH by auto

    show ?thesis
    proof (safe, goal-cases)
      case 1
      have leaves-up_i (ins k x (Node ts t)) = leaves-list ls @ leaves-up_i (Up_i lt a
rt)
        using list-split Up_i Nil by (auto simp add: list-conc)
      also have ... = leaves-list ls @ ins-list x (leaves t)
        using IH by simp
      also have ... = ins-list x (leaves (Node ts t))
```

```
        using ins-list-split
        using 2.prems list-split local.Nil
      by (metis aligned-imp-Laligned append-self-conv2 concat.simps(1) list.simps(8))
       finally show ?case.
     next
      case 2
      have aligned-up_i l (ins k x (Node ts t)) u = aligned-up_i l (node_i k (ts @ [(lt,
a)]) rt) u
        using Nil Up_i list-split list-conc node_i-aligned by simp
       moreover have aligned l (Node (ts@[(lt,a)]) rt) u
        using 2.prems(2) IH ‹ts = ts' @ [(sub', sep')]› align-subst-two by fastforce
       ultimately show ?case
        using node_i-aligned
        by auto
     qed
   qed
 next
   case (Cons h list)
   then obtain sub sep where h-split: h = (sub,sep)
     by (cases h)

   then have sorted-inorder-sub: sorted-less (leaves sub)
     using 2.prems list-conc Cons sorted-leaves-induct-subtree
     by fastforce
   moreover have order-sub: order k sub
     using 2.prems list-conc Cons h-split
     by auto
   then show ?thesis

   proof (cases ls)
     case Nil
     then have aligned-sub: aligned l sub sep
       using 2.prems(2) list-conc h-split Cons
       by auto
     then have IH: leaves-up_i (ins k x sub) = ins-list x (leaves sub) ∧ aligned-up_i
l (ins k x sub) sep
     proof −
       have x ≤ sep
           using 2.prems(2) aligned-sorted-separators h-split list-split local.Cons
sorted-cons sorted-snoc split-set.split-req(3) split-set-axioms
         by blast
       then show ?thesis
       using 2.IH(2)[OF sym[OF list-split] Cons sym[OF h-split], of l sep]
         using 2.prems list-split local.Nil aligned-sub sorted-inorder-sub order-sub
         using order-impl-root-order
         by auto
     qed
     then show ?thesis
     proof (cases ins k x sub)
```

case (T_i a)
have *IH*:*leaves a* = *ins-list x* (*leaves sub*) ∧ *aligned l a sep*
using T_i *IH* **by** (*auto*)
show *?thesis*
proof (*safe*, *goal-cases*)
case *1*
have *leaves-up*$_i$ (*ins k x* (*Node ts t*)) = *leaves-list ls* @ *leaves a* @ *leaves-list list* @ *leaves t*
using *h-split list-split* T_i *Cons* **by** *simp*
also have ... = *leaves-list ls* @ *ins-list x* (*leaves sub*) @ *leaves-list list* @ *leaves t*
using *IH* **by** *simp*
also have ... = *ins-list x* (*leaves* (*Node ts t*))
using *ins-list-split ins-list-split-right*
using *list-split 2.prems Cons h-split*
by (*metis aligned-imp-Laligned*)
finally show *?case*.
next
case *2*
have *aligned-up*$_i$ *l* (*ins k x* (*Node ts t*)) *u* = *aligned l* (*Node* ((*a*,*sep*)#*list*) *t*) *u*
using *Nil Cons list-conc list-split h-split* T_i **by** *simp*
moreover have *aligned l* (*Node* ((*a*,*sep*)#*list*) *t*) *u*
using *aligned-sub 2.prems*(*2*) *IH h-split list-conc Cons Nil*
by *auto*
ultimately show *?case*
by *auto*
qed
next
case (*Up*$_i$ *lt a rt*)
then have *IH*:*leaves-up*$_i$ (*Up*$_i$ *lt a rt*) = *ins-list x* (*leaves sub*) ∧ *aligned-up*$_i$ *l* (*Up*$_i$ *lt a rt*) *sep*
using *IH h-split list-split Cons sorted-inorder-sub*
by *auto*
show *?thesis*
proof (*safe*, *goal-cases*)
case *1*
have *leaves-up*$_i$ (*ins k x* (*Node ts t*)) = *leaves-list ls* @ *leaves lt* @ *leaves rt* @ *leaves-list list* @ *leaves t*
using *h-split list-split Up*$_i$ *Cons* **by** *simp*
also have ... = *leaves-list ls* @ *ins-list x* (*leaves sub*) @ *leaves-list list* @ *leaves t*
using *IH* **by** *simp*
also have ... = *ins-list x* (*leaves* (*Node ts t*))
using *ins-list-split ins-list-split-right*
using *list-split 2.prems Cons h-split*
by (*metis aligned-imp-Laligned*)
finally show *?case*.
next

```
        case 2
          have aligned-up_i l (ins k x (Node ts t)) u = aligned-up_i l (node_i k
((lt,a)#(rt,sep)#list) t) u
           using Nil Cons list-conc list-split h-split Up_i by simp
          moreover have aligned l (Node ((lt,a)#(rt,sep)#list) t) u
           using aligned-sub 2.prems(2) IH h-split list-conc Cons Nil
           by auto
          ultimately show ?case
           using node_i-aligned by auto
        qed
      qed
    next
      case ls-split': Cons
      then obtain ls' sub' sep' where ls-split: ls = ls'@[(sub',sep')]
        by (metis list.discI old.prod.exhaust snoc-eq-iff-butlast)
      then have aligned-sub: aligned sep' sub sep
        using 2.prems(2) list-conc h-split Cons
        using align-last aligned-split-left by blast
      then have IH: leaves-up_i (ins k x sub) = ins-list x (leaves sub) ∧ aligned-up_i
sep' (ins k x sub) sep
      proof −
        have x ≤ sep
           using 2.prems(2) aligned-sorted-separators h-split list-split local.Cons
sorted-cons sorted-snoc split-set.split-req(3) split-set-axioms
          by blast
        moreover have sep' < x
           using 2.prems(2) aligned-sorted-separators list-split ls-split sorted-cons
sorted-snoc split-set.split-req(2) split-set-axioms
          by blast
        ultimately show ?thesis
        using 2.IH(2)[OF sym[OF list-split] Cons sym[OF h-split], of sep' sep]
          using 2.prems list-split ls-split aligned-sub sorted-inorder-sub order-sub
          using order-impl-root-order
          by auto
      qed
      then show ?thesis
      proof (cases ins k x sub)
        case (T_i a)
        have IH:leaves a = ins-list x (leaves sub) ∧ aligned sep' a sep
          using T_i IH by (auto)
        show ?thesis
        proof (safe, goal-cases)
          case 1
         have leaves-up_i (ins k x (Node ts t)) = leaves-list ls @ leaves a @ leaves-list
list @ leaves t
           using h-split list-split T_i Cons by simp
          also have ... = leaves-list ls @ ins-list x (leaves sub) @ leaves-list list @
leaves t
           using IH by simp
```

also have ... = *ins-list x* (*leaves* (*Node ts t*))
using *ins-list-split ins-list-split-right*
using *list-split 2.prems Cons h-split*
by (*metis aligned-imp-Laligned*)
finally show *?case*.
next
case *2*
have *aligned-up$_i$ l* (*ins k x* (*Node ts t*)) *u* = *aligned l* (*Node* (*ls'*@(*sub'*,*sep'*)#(*a*,*sep*)#*list*) *t*) *u*
using *Nil Cons list-conc list-split h-split T$_i$ ls-split* **by** *simp*
moreover have *aligned l* (*Node* (*ls'*@(*sub'*,*sep'*)#(*a*,*sep*)#*list*) *t*) *u*
using *aligned-sub 2.prems*(*2*) *IH h-split list-conc Cons Nil ls-split*
using *aligned-subst* **by** *fastforce*
ultimately show *?case*
by *auto*
qed
next
case (*Up$_i$ lt a rt*)
then have *IH*:*leaves-up$_i$* (*Up$_i$ lt a rt*) = *ins-list x* (*leaves sub*) ∧ *aligned-up$_i$ sep'* (*Up$_i$ lt a rt*) *sep*
using *IH h-split list-split Cons sorted-inorder-sub*
by *auto*
show *?thesis*
proof (*safe*, *goal-cases*)
case *1*
have *leaves-up$_i$* (*ins k x* (*Node ts t*)) = *leaves-list ls* @ *leaves lt* @ *leaves rt* @ *leaves-list list* @ *leaves t*
using *h-split list-split Up$_i$ Cons* **by** *simp*
also have ... = *leaves-list ls* @ *ins-list x* (*leaves sub*) @ *leaves-list list* @ *leaves t*
using *IH* **by** *simp*
also have ... = *ins-list x* (*leaves* (*Node ts t*))
using *ins-list-split ins-list-split-right*
using *list-split 2.prems Cons h-split*
by (*metis aligned-imp-Laligned*)
finally show *?case*.
next
case *2*
have *aligned-up$_i$ l* (*ins k x* (*Node ts t*)) *u* = *aligned-up$_i$ l* (*node$_i$ k* (*ls'*@(*sub'*,*sep'*)#(*lt*,*a*)#(*rt*,*sep*)#*list*) *t*) *u*
using *Nil Cons list-conc list-split h-split Up$_i$ ls-split* **by** *simp*
moreover have *aligned l* (*Node* (*ls'*@(*sub'*,*sep'*)#(*lt*,*a*)#(*rt*,*sep*)#*list*) *t*) *u*
using *aligned-sub 2.prems*(*2*) *IH h-split list-conc Cons Nil ls-split align-subst-three*
by *auto*
ultimately show *?case*
using *node$_i$-aligned* **by** *auto*
qed

```
    qed
  qed
 qed
qed

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

lemma Laligned-insert-list: sorted-less ks ⟹ x ≤ u ⟹ Laligned (Leaf ks) u ⟹
Laligned (Leaf (insert-list x ks)) u
 using insert-list-req
 by (simp add: set-ins-list)

lemma Lalign-subst-two: Laligned (Node (ts@[(sub,sep)]) t) u ⟹ aligned sep lt a
⟹ aligned a rt u ⟹ Laligned (Node (ts@[(sub,sep),(lt,a)]) rt) u
 apply(induction ts)
 apply (auto)
 by (meson align-subst-two aligned.simps(2))

lemma Lalign-subst-three: Laligned (Node (ls@(subl,sepl)#(subr,sepr)#rs) t) u
⟹ aligned sepl lt a ⟹ aligned a rt sepr ⟹ Laligned (Node (ls@(subl,sepl)#(lt,a)#(rt,sepr)#rs)
t) u
 apply(induction ls)
 apply auto
 by (meson align-subst-three aligned.simps(2))

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
 using assms
proof (cases length ks ≤ 2*k)
 case False
 then obtain ls sep rs where split-half-ts:
   take ((length ks + 1) div 2) ks = ls@[sep]
   drop ((length ks + 1) div 2) ks = rs
   using split-half-not-empty[of ks]
   by auto
 moreover have sorted-less (ls@[sep])
   by (metis append-take-drop-id assms(2) sorted-wrt-append split-half-ts(1))
 ultimately have Laligned (Leaf (ls@[sep])) sep
   using split-half-conc[of ks ls@[sep] rs] assms sorted-snoc-iff[of ls sep]
   by auto
 moreover have aligned sep (Leaf rs) u
 proof −
   have length rs > 0
     using False assms(3) split-half-ts(2) by fastforce
   then obtain sep' rs' where rs = sep' # rs'
```

```
    by (cases rs) auto
  moreover have sep < sep'
  by (metis append-take-drop-id assms(2) calculation in-set-conv-decomp sorted-mid-iff
sorted-snoc-iff split-half-ts(1) split-half-ts(2))
  moreover have sorted-less rs
    by (metis append-take-drop-id assms(2) sorted-wrt-append split-half-ts(2))
  ultimately show ?thesis
    using split-half-ts split-half-conc[of ks ls@[sep] rs] assms
    by auto
 qed
 ultimately show ?thesis
  using split-half-ts False by auto
qed simp

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 ≤ u
 shows (leaves-up_i (ins k x t)) = ins-list x (leaves t) ∧ Laligned-up_i (ins k x t) u
 using assms
proof(induction k x t arbitrary: u rule: ins.induct)
 case (1 k x ks)
 then show ?case
 proof (safe, goal-cases)
  case -: 1
  then show ?case
    using 1 insert-list-req by auto
 next
  case 2
  from 1 have Laligned (Leaf (insert-list x ks)) u
    by (metis Laligned-insert-list leaves.simps(1))
  moreover have sorted-less (insert-list x ks)
    using 1.prems(3) split-set.insert-list-req split-set-axioms
    by auto
  ultimately show ?case
    using Lnode_i-Laligned[of insert-list x ks u k] 1
    by (auto simp del: Lnode_i.simps split-half.simps)
 qed
next
 case (2 k x ts t)
 then obtain ls rs where list-split: split ts x = (ls,rs)
  by (cases split ts x)
 then have list-conc: ts = ls@rs
  using split-set.split-conc split-set-axioms by blast
```

```
  then show ?case
  proof (cases rs)
    case Nil
    then obtain ts' sub' sep' where ts = ts'@[(sub',sep')]
      apply(cases ts)
      using 2 list-conc Nil apply(simp)
      by (metis isin.cases list.distinct(1) rev-exhaust)
    have IH: leaves-up_i (ins k x t) = ins-list x (leaves t) ∧ aligned-up_i sep' (ins k
x t) u

      proof −

        note ins-inorder[of k]
        have sorted-less (leaves t)
          using 2.prems(3) sorted-leaves-induct-last by blast
        moreover have sep' < x
          using split-req[of ts x] list-split
          by (metis 2.prems(2) Laligned-sorted-separators ‹ts = ts' @ [(sub', sep')]›
local.Nil self-append-conv sorted-snoc)
        moreover have aligned sep' t u
          using 2.prems(2) Lalign-last ‹ts = ts' @ [(sub', sep')]› by blast
        ultimately show ?thesis
       by (meson 2.prems(1) 2.prems(4) 2.prems(5) ins-inorder order-impl-root-order
root-order.simps(2))
      qed
    show ?thesis
    proof (cases ins k x t)
      case (T_i a)
      have IH: leaves a = ins-list x (leaves t) ∧ aligned sep' a u
        using IH T_i by force
      show ?thesis
      proof(safe, goal-cases)
        case 1
        have leaves-up_i (ins k x (Node ts t)) = leaves-list ls @ leaves a
          using list-split T_i Nil by (auto simp add: list-conc)
        also have ... = leaves-list ls @ (ins-list x (leaves t))
          by (simp add: IH)
        also have ... = ins-list x (leaves (Node ts t))
          using ins-list-split
          using 2.prems list-split Nil
          by auto
        finally show ?case .
      next
        case 2
        have Laligned-up_i (ins k x (Node ts t)) u = Laligned (Node ts a) u
          using Nil T_i list-split list-conc by simp
        moreover have Laligned (Node ts a) u
          using 2.prems(2)
          by (metis IH ‹ts = ts' @ [(sub', sep')]› Laligned-subst-last)
```

```
      ultimately show ?case
        by auto
    qed
  next
    case (Up_i lt a rt)
     then have IH:leaves-up_i (Up_i lt a rt) = ins-list x (leaves t) ∧ aligned-up_i
sep' (Up_i lt a rt) u
      using IH by auto

    show ?thesis
    proof (safe, goal-cases)
      case 1
      have leaves-up_i (ins k x (Node ts t)) = leaves-list ls @ leaves-up_i (Up_i lt a
rt)
        using list-split Up_i Nil by (auto simp add: list-conc)
      also have ... = leaves-list ls @ ins-list x (leaves t)
        using IH by simp
      also have ... = ins-list x (leaves (Node ts t))
        using ins-list-split
        using 2.prems list-split local.Nil by auto
      finally show ?case.
    next
      case 2
      have Laligned-up_i (ins k x (Node ts t)) u = Laligned-up_i (node_i k (ts @ [(lt,
a)]) rt) u
        using Nil Up_i list-split list-conc node_i-aligned by simp
      moreover have Laligned (Node (ts@[(lt,a)]) rt) u
      using 2.prems(2) IH ‹ts = ts' @ [(sub', sep')]› Lalign-subst-two by fastforce
      ultimately show ?case
        using node_i-Laligned
        by auto
    qed
  qed
next
  case (Cons h list)
  then obtain sub sep where h-split: h = (sub,sep)
    by (cases h)

  then have sorted-inorder-sub: sorted-less (leaves sub)
    using 2.prems list-conc Cons sorted-leaves-induct-subtree
    by fastforce
  moreover have order-sub: order k sub
    using 2.prems list-conc Cons h-split
    by auto
  then show ?thesis

  proof (cases ls)
    case Nil
    then have aligned-sub: Laligned sub sep
```

using *2.prems(2) list-conc h-split Cons*
by *auto*
then have *IH*: *leaves-up$_i$ (ins k x sub) = ins-list x (leaves sub) ∧ Laligned-up$_i$ (ins k x sub) sep*
proof −
have $x \leq sep$
using *2.prems(2) Laligned-sorted-separators h-split list-split local.Cons sorted-snoc split-set.split-req(3) split-set-axioms*
by *blast*
then show *?thesis*
using *2.IH(2)[OF sym[OF list-split] Cons sym[OF h-split], of sep]*
using *2.prems list-split local.Nil aligned-sub sorted-inorder-sub order-sub*
using *order-impl-root-order*
by *auto*
qed
then show *?thesis*
proof (*cases ins k x sub*)
case (T_i *a*)
have *IH*:*leaves a = ins-list x (leaves sub) ∧ Laligned a sep*
using T_i *IH* **by** (*auto*)
show *?thesis*
proof (*safe*, *goal-cases*)
case *1*
have *leaves-up$_i$ (ins k x (Node ts t)) = leaves-list ls @ leaves a @ leaves-list list @ leaves t*
using *h-split list-split* T_i *Cons* **by** *simp*
also have ... = *leaves-list ls @ ins-list x (leaves sub) @ leaves-list list @ leaves t*
using *IH* **by** *simp*
also have ... = *ins-list x (leaves (Node ts t))*
using *ins-list-split ins-list-split-right*
using *list-split 2.prems Cons h-split* **by** *auto*
finally show *?case*.
next
case *2*
have *Laligned-up$_i$ (ins k x (Node ts t)) u = Laligned (Node ((a,sep)#list) t) u*
using *Nil Cons list-conc list-split h-split* T_i **by** *simp*
moreover have *Laligned (Node ((a,sep)#list) t) u*
using *aligned-sub 2.prems(2) IH h-split list-conc Cons Nil*
by *auto*
ultimately show *?case*
by *auto*
qed
next
case (*Up$_i$ lt a rt*)
then have *IH*:*leaves-up$_i$ (Up$_i$ lt a rt) = ins-list x (leaves sub) ∧ Laligned-up$_i$ (Up$_i$ lt a rt) sep*
using *IH h-split list-split Cons sorted-inorder-sub*

by *auto*
show *?thesis*
proof (*safe*, *goal-cases*)
case *1*
have *leaves-up*$_i$ (*ins k x* (*Node ts t*)) = *leaves-list ls* @ *leaves lt* @ *leaves rt* @ *leaves-list list* @ *leaves t*
using *h-split list-split Up*$_i$ *Cons* **by** *simp*
also have ... = *leaves-list ls* @ *ins-list x* (*leaves sub*) @ *leaves-list list* @ *leaves t*
using *IH* **by** *simp*
also have ... = *ins-list x* (*leaves* (*Node ts t*))
using *ins-list-split ins-list-split-right*
using *list-split 2.prems Cons h-split* **by** *auto*
finally show *?case*.
next
case *2*
have *Laligned-up*$_i$ (*ins k x* (*Node ts t*)) *u* = *Laligned-up*$_i$ (*node*$_i$ *k* ((*lt*,*a*)#(*rt*,*sep*)#*list*) *t*) *u*
using *Nil Cons list-conc list-split h-split Up*$_i$ **by** *simp*
moreover have *Laligned* (*Node* ((*lt*,*a*)#(*rt*,*sep*)#*list*) *t*) *u*
using *aligned-sub 2.prems*(*2*) *IH h-split list-conc Cons Nil*
by *auto*
ultimately show *?case*
using *node*$_i$*-Laligned* **by** *auto*
qed
qed
next
case *ls-split'*: *Cons*
then obtain *ls' sub' sep'* **where** *ls-split*: *ls* = *ls'*@[(*sub'*,*sep'*)]
by (*metis list.discI old.prod.exhaust snoc-eq-iff-butlast*)
then have *aligned-sub*: *aligned sep' sub sep*
using *2.prems*(*2*) *list-conc h-split Cons*
using *Lalign-last Laligned-split-left*
by *blast*
then have *IH*: *leaves-up*$_i$ (*ins k x sub*) = *ins-list x* (*leaves sub*) ∧ *aligned-up*$_i$ *sep'* (*ins k x sub*) *sep*
proof −
have *x* ≤ *sep*
using *2.prems*(*2*) *Laligned-sorted-separators h-split list-split local.Cons sorted-snoc split-set.split-req*(*3*) *split-set-axioms*
by *blast*
moreover have *sep'* < *x*
using *2.prems*(*2*) *Laligned-sorted-separators list-split ls-split sorted-cons sorted-snoc split-set.split-req*(*2*) *split-set-axioms*
by *blast*
ultimately show *?thesis*
using *2.prems*(*1*) *aligned-sub ins-inorder order-sub sorted-inorder-sub*
using *order-impl-root-order*
by *blast*

qed
then show *?thesis*
proof (*cases ins k x sub*)
case (T_i *a*)
have *IH*:*leaves a* = *ins-list x* (*leaves sub*) ∧ *aligned sep′ a sep*
using T_i *IH* **by** (*auto*)
show *?thesis*
proof (*safe*, *goal-cases*)
case *1*
have *leaves-up*$_i$ (*ins k x* (*Node ts t*)) = *leaves-list ls* @ *leaves a* @ *leaves-list list* @ *leaves t*
using *h-split list-split* T_i *Cons* **by** *simp*
also have ... = *leaves-list ls* @ *ins-list x* (*leaves sub*) @ *leaves-list list* @ *leaves t*
using *IH* **by** *simp*
also have ... = *ins-list x* (*leaves* (*Node ts t*))
using *ins-list-split ins-list-split-right*
using *list-split 2.prems Cons h-split* **by** *auto*
finally show *?case*.
next
case *2*
have *Laligned-up*$_i$ (*ins k x* (*Node ts t*)) *u* = *Laligned* (*Node* (*ls′*@(*sub′*,*sep′*)#(*a*,*sep*)#*list*) *t*) *u*
using *Nil Cons list-conc list-split h-split* T_i *ls-split* **by** *simp*
moreover have *Laligned* (*Node* (*ls′*@(*sub′*,*sep′*)#(*a*,*sep*)#*list*) *t*) *u*
using *aligned-sub 2.prems*(*2*) *IH h-split list-conc Cons Nil ls-split*
using *Laligned-subst* **by** *fastforce*
ultimately show *?case*
by *auto*
qed
next
case (*Up*$_i$ *lt a rt*)
then have *IH*:*leaves-up*$_i$ (*Up*$_i$ *lt a rt*) = *ins-list x* (*leaves sub*) ∧ *aligned-up*$_i$ *sep′* (*Up*$_i$ *lt a rt*) *sep*
using *IH h-split list-split Cons sorted-inorder-sub*
by *auto*
show *?thesis*
proof (*safe*, *goal-cases*)
case *1*
have *leaves-up*$_i$ (*ins k x* (*Node ts t*)) = *leaves-list ls* @ *leaves lt* @ *leaves rt* @ *leaves-list list* @ *leaves t*
using *h-split list-split Up*$_i$ *Cons* **by** *simp*
also have ... = *leaves-list ls* @ *ins-list x* (*leaves sub*) @ *leaves-list list* @ *leaves t*
using *IH* **by** *simp*
also have ... = *ins-list x* (*leaves* (*Node ts t*))
using *ins-list-split ins-list-split-right*
using *list-split 2.prems Cons h-split* **by** *auto*
finally show *?case*.

```
      next
        case 2
           have Laligned-up_i (ins k x (Node ts t)) u = Laligned-up_i (node_i k
(ls'@(sub',sep')#(lt,a)#(rt,sep)#list) t) u
          using Nil Cons list-conc list-split h-split Up_i ls-split by simp
         moreover have Laligned (Node (ls'@(sub',sep')#(lt,a)#(rt,sep)#list) t)
u
             using aligned-sub 2.prems(2) IH h-split list-conc Cons Nil ls-split
Lalign-subst-three
          by auto
         ultimately show ?case
          using node_i-Laligned by auto
       qed
     qed
   qed
  qed
qed

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)
  apply(cases u)
   apply(auto)
  done

lemma tree_i-order: ⟦k > 0; root-order-up_i k u⟧ ⟹ root-order k (tree_i u)
  apply(cases u)
   apply(auto simp add: order-impl-root-order)
  done

lemma tree_i-inorder: inorder-up_i u = inorder (tree_i u)
  apply (cases u)
   apply auto
  done

lemma tree_i-leaves: leaves-up_i u = leaves (tree_i u)
  apply (cases u)
   apply auto
  done

lemma tree_i-aligned: aligned-up_i l a u ⟹ aligned l (tree_i a) u
  apply (cases a)
```

apply *auto*
done

lemma *tree$_i$-Laligned*: *Laligned-up$_i$ a u* $\Longrightarrow$ *Laligned* (*tree$_i$ a*) *u*
apply (*cases a*)
apply *auto*
done

lemma *insert-bal*: *bal t* $\Longrightarrow$ *bal* (*insert k x t*)
using *ins-bal*
by (*simp add*: *tree$_i$-bal*)

lemma *insert-order*: $[\![k > 0$; *sorted-less* (*leaves t*); *root-order k t*$]\!] \Longrightarrow$ *root-order k* (*insert k x t*)
using *ins-root-order*
by (*simp add*: *tree$_i$-order*)

lemma *insert-inorder*:
assumes *k > 0 root-order k t sorted-less* (*leaves t*) *aligned l t u l* $<$ *x x* $\leq$ *u*
shows *leaves* (*insert k x t*) = *ins-list x* (*leaves t*)
and *aligned l* (*insert k x t*) *u*
using *ins-inorder assms*
by (*simp-all add*: *tree$_i$-leaves tree$_i$-aligned*)

lemma *insert-Linorder*:
assumes *k > 0 root-order k t sorted-less* (*leaves t*) *Laligned t u x* $\leq$ *u*
shows *leaves* (*insert k x t*) = *ins-list x* (*leaves t*)
and *Laligned* (*insert k x t*) *u*
using *ins-Linorder insert-inorder assms*
by (*simp-all add*: *tree$_i$-leaves tree$_i$-Laligned*)

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*
using *insert-Linorder top-greatest assms* **by** *simp-all*

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) |
      Up_i 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 node_i k (mts@(mt,sep)#tts) tt of
       T_i u ⇒
        Node ls u |
       Up_i l a r ⇒
        Node (ls@[(l,a)]) r) |
    (Node rts rt,rsep)#rs ⇒ (
      case node_i k (mts@(mt,sep)#rts) rt of
      T_i u ⇒
        Node (ls@(u,rsep)#rs) (Node tts tt) |
      Up_i 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 (cases height t)
  case 0
  then obtain ts subs where t = Leaf ts sub = Leaf subs using height-Leaf assms
    by metis
  moreover have rs = (rsub,rsep) # list ⟹ rsub = Node rts rt ⟹ False
    for rsub rsep list rts rt
```

```
  proof (goal-cases)
    case 1
    then have height rsub = height t
      using assms(2) by auto
    then have height rsub = 0
      using 0 by simp
    then show ?case
      using 1(2) height-Leaf by blast
  qed
  ultimately show ?thesis
    by (auto split!: list.splits bplustree.splits)
next
  case (Suc nat)
  then obtain tts tt where t-node: t = Node tts tt
    using height-Leaf by (cases t) simp
  then obtain mts mt where sub-node: sub = Node mts mt
    using assms by (cases sub) simp
  then show ?thesis
  proof (cases length mts ≥ k ∧ length tts ≥ k)
    case False
    then show ?thesis
    proof (cases rs)
      case Nil
        then have height-up_i (node_i k (mts@(mt,sep)#tts) tt) = height (Node
(mts@(mt,sep)#tts) tt)
        using node_i-height by blast
      also have ... = max (height t) (height sub)
        by (metis assms(1) height-up_i.simps(2) height-list-split sub-node t-node)
      finally have height-max: height-up_i (node_i k (mts @ (mt, sep) # tts) tt) =
max (height t) (height sub) by simp
      then show ?thesis
      proof (cases node_i k (mts@(mt,sep)#tts) tt)
        case (T_i u)
          then have height u = max (height t) (height sub) using height-max by
simp
        then have height (Node ls u) = height (Node (ls@[(sub,sep)]) t)
          by (induction ls) (auto simp add: max.commute)
        then show ?thesis using Nil False T_i
          by (simp add: sub-node t-node)
      next
        case (Up_i l a r)
        then have height (Node (ls@[(sub,sep)]) t) =  height (Node (ls@[(l,a)]) r)
          using assms(1) height-max by (induction ls) auto
        then show ?thesis
          using Up_i Nil sub-node t-node by auto
      qed
    next
      case (Cons a list)
      then obtain rsub rsep where a-split: a = (rsub, rsep)
```

by (*cases a*)
then obtain *rts rt* **where** *r-node*: *rsub* = *Node rts rt*
using *assms*(*2*) *Cons height-Leaf Suc* **by** (*cases rsub*) *simp-all*

then have *height-up*$_i$ (*node*$_i$ *k* (*mts*@(*mt*,*sep*)#*rts*) *rt*) = *height* (*Node* (*mts*@(*mt*,*sep*)#*rts*) *rt*)
using *node*$_i$*-height* **by** *blast*
also have ... = *max* (*height rsub*) (*height sub*)
by (*metis r-node height-up*$_i$*.simps*(*2*) *height-list-split max.commute sub-node*)
finally have *height-max*: *height-up*$_i$ (*node*$_i$ *k* (*mts* @ (*mt*, *sep*) # *rts*) *rt*) = *max* (*height rsub*) (*height sub*) **by** *simp*
then show *?thesis*
proof (*cases node*$_i$ *k* (*mts*@(*mt*,*sep*)#*rts*) *rt*)
case (*T*$_i$ *u*)
then have *height u* = *max* (*height rsub*) (*height sub*)
using *height-max* **by** *simp*
then show *?thesis*
using *T*$_i$ *False Cons r-node a-split sub-node t-node* **by** *auto*
next
case (*Up*$_i$ *l a r*)
then have *height-max*: *max* (*height l*) (*height r*) = *max* (*height rsub*) (*height sub*)
using *height-max* **by** *auto*
then show *?thesis*
using *Cons a-split r-node Up*$_i$ *sub-node t-node* **by** *auto*
qed
qed
qed (*simp add*: *sub-node t-node*)
qed

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*)
using *rebalance-middle-tree-height assms* **by** *auto*

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

lemma *del-height*: ⟦*k* > *0*; *root-order k t*; *bal t*⟧ ⟹ *height* (*del k x t*) = *height t*
proof(*induction k x t rule*: *del.induct*)
case (*2 k x ts t*)
then obtain *ls list* **where** *list-split*: *split ts x* = (*ls*, *list*) **by** (*cases split ts x*)
then show *?case*
proof(*cases list*)
case *Nil*

```
    then have height (del k x t) = height t
      using 2 list-split
      by (simp add: order-impl-root-order)
    moreover obtain lls sub sep where ls = lls@[(sub,sep)]
      using split-conc 2 list-split Nil
     by (metis append-Nil2 less-nat-zero-code list.size(3) old.prod.exhaust rev-exhaust
root-order.simps(2))
    moreover have Node ls t = Node ts t using split-conc Nil list-split by auto
    ultimately show ?thesis
      using rebalance-last-tree-height 2 list-split Nil split-conc
      by (auto simp add: max.assoc sup-nat-def max-def)
  next
    case (Cons a rs)
     then have rs-height: case rs of [] ⇒ True | (rsub,rsep)#- ⇒ height rsub =
height t
      using 2.prems(3) bal-sub-height list-split split-conc by blast
    from Cons obtain sub sep where a-split: a = (sub,sep) by (cases a)

      have height-t-sub: height t = height sub
     using 2.prems(3) a-split list-split local.Cons split-set.split-set(1) split-set-axioms
by fastforce
      have height-t-del: height (del k x sub) = height t
        by (metis 2.IH(2) 2.prems(1) 2.prems(2) 2.prems(3) a-split bal.simps(2)
list-split local.Cons order-impl-root-order root-order.simps(2) some-child-sub(1) split-set(1))
      then have height (rebalance-middle-tree k ls (del k x sub) sep rs t) = height
(Node (ls@((del k x sub),sep)#rs) t)
        using rs-height rebalance-middle-tree-height by simp
      also have ... = height (Node (ls@(sub,sep)#rs) t)
        using height-t-sub 2.prems height-t-del
        by auto
      also have ... = height (Node ts t)
        using 2 a-split list-split Cons split-set(1) split-conc
        by auto
      finally show ?thesis
        using  Cons a-split list-split 2
        by simp
  qed
qed simp




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)
  apply(cases sub; cases t)
  using assms
```

apply (*auto*
split!: *bplustree.splits* up_i*.splits list.splits*
simp del: $node_i$*.simps* $Lnode_i$*.simps*
simp add: $node_i$*-leaves-simps* $Lnode_i$*-leaves-simps*
)
done

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*)
using *rebalance-middle-tree-inorder assms* **by** *auto*

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

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*
by (*metis Suc-inject Un-iff bal.simps*(*2*) *height-bal-tree singletonD*)

lemma *bal-list-merge*:
assumes *bal-up*$_i$ (*Up*$_i$ (*Node as a*) *x* (*Node bs b*))
shows *bal* (*Node* (*as*@(*a*,*x*)#*bs*) *b*)
proof −
have ∀ *x*∈*set* (*subtrees* (*as* @ (*a*, *x*) # *bs*)). *bal x*
using *subtrees-split assms* **by** *auto*
moreover have *bal b*
using *assms* **by** *auto*
moreover have ∀ *x*∈*set* (*subtrees as*) ∪ {*a*} ∪ *set* (*subtrees bs*). *height x* = *height b*
using *assms height-bal-subtrees-merge*
unfolding *bal-up*$_i$*.simps*
by *blast*
ultimately show *?thesis*
by *auto*
qed

lemma $node_i$*-bal-up*$_i$:
assumes *bal-up*$_i$ ($node_i$ *k ts t*)
shows *bal* (*Node ts t*)
using *assms*
proof(*cases length ts* ≤ *2*∗*k*)
case *False*
then obtain *ls sub sep rs* **where** *split-list*: *split-half ts* = (*ls*@[(*sub*,*sep*)], *rs*)
using $node_i$*-cases* **by** *blast*
then have $node_i$ *k ts t* = *Up*$_i$ (*Node ls sub*) *sep* (*Node rs t*)

```
    using False by auto
  moreover have ts = ls@(sub,sep)#rs
   by (metis append-Cons append-Nil2 append-eq-append-conv2 local.split-list same-append-eq
split-half-conc)
  ultimately show ?thesis
    using bal-list-merge[of ls sub sep rs t] assms
    by (simp del: bal.simps bal-up_i.simps)
qed simp

lemma node_i-bal-simp: bal-up_i (node_i k ts t) = bal (Node ts t)
  using node_i-bal node_i-bal-up_i by blast

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 (cases t)
  case t-node: (Leaf txs)
  then obtain mxs where sub-node: sub = Leaf mxs
    using assms by (cases sub) (auto simp add: t-node)
  then have sub-heights: height sub = height t bal sub bal t
    apply (metis Suc-inject assms bal-split-left(1) bal-split-left(2) height-bal-tree)
    apply (meson assms bal.simps(2) bal-split-left(1))
    using assms bal.simps(2) by blast
  show ?thesis
  proof (cases length mxs ≥ k ∧ length txs ≥ k)
    case True
    then show ?thesis
      using t-node sub-node assms
      by (auto simp del: bal.simps)
  next
    case False
    then show ?thesis
    proof (cases rs)
      case Nil
      have height-up_i (Lnode_i k (mxs@txs)) = height (Leaf (mxs@txs))
        using Lnode_i-height by blast
      also have ... = 0
        by simp
      also have ... = height t
        using height-bal-tree sub-heights(3) t-node by fastforce
      finally have height-up_i (Lnode_i k (mxs@txs)) = height t .
      moreover have bal-up_i (Lnode_i k (mxs@txs))
      by (simp add: bal-up_i.elims(3) height-Leaf height-up_i.simps(2) max-nat.neutr-eq-iff)
      ultimately show ?thesis
        apply (cases Lnode_i k (mxs@txs))
        using assms Nil sub-node t-node by auto
    next
      case (Cons r rs)
      then obtain rsub rsep where r-split: r = (rsub,rsep) by (cases r)
```

```
      then have rsub-height: height rsub = height t bal rsub
        using assms Cons by auto
      then obtain rxs where r-node: rsub = Leaf rxs
        apply(cases rsub) using assms t-node by auto
      have height-up_i (Lnode_i k (mxs@rxs)) = height (Leaf (mxs@rxs))
        using Lnode_i-height by blast
      also have ... = 0
        by auto
      also have ... = height rsub
        using height-bal-tree r-node rsub-height(2) by fastforce
      finally have 1: height-up_i (Lnode_i k (mxs@rxs)) = height rsub .
      moreover have 2: bal-up_i (Lnode_i k (mxs@rxs))
        by simp
      ultimately show ?thesis
      proof (cases Lnode_i k (mxs@rxs))
        case (T_i u)
        then have bal (Node (ls@(u,rsep)#rs) t)
          using 1 2 Cons assms t-node subtrees-split sub-heights r-split rsub-height
          unfolding bal.simps by (auto simp del: height-bplustree.simps)
        then show ?thesis
          using Cons assms t-node sub-node r-split r-node False T_i
          by (auto simp del: node_i.simps bal.simps)
      next
        case (Up_i l a r)
        then have bal (Node (ls@(l,a)#(r,rsep)#rs) t)
          using 1 2 Cons assms t-node subtrees-split sub-heights r-split rsub-height
          unfolding bal.simps by (auto simp del: height-bplustree.simps)
        then show ?thesis
          using Cons assms t-node sub-node r-split r-node False Up_i
          by (auto simp del: node_i.simps bal.simps)
      qed
    qed
  qed
next
  case t-node: (Node tts tt)
  then obtain mts mt where sub-node: sub = Node mts mt
    using assms by (cases sub) (auto simp add: t-node)
  have sub-heights: height sub = height t bal sub bal t
    using assms by auto
  show ?thesis
  proof (cases length mts ≥ k ∧ length tts ≥ k)
    case True
    then show ?thesis
      using t-node sub-node assms
      by (auto simp del: bal.simps)
  next
    case False
    then show ?thesis
    proof (cases rs)
```

case *Nil*
have *height-up$_i$ (node$_i$ k (mts@(mt,sep)#tts) tt) = height (Node (mts@(mt,sep)#tts) tt)*
using *node$_i$-height* **by** *blast*
also have *... = Suc (height tt)*
by *(metis height-bal-tree height-up$_i$.simps(2) height-list-split max.idem sub-heights(1) sub-heights(3) sub-node t-node)*
also have *... = height t*
using *height-bal-tree sub-heights(3) t-node* **by** *fastforce*
finally have *height-up$_i$ (node$_i$ k (mts@(mt,sep)#tts) tt) = height t* **by** *simp*
moreover have *bal-up$_i$ (node$_i$ k (mts@(mt,sep)#tts) tt)*
by *(metis bal-list-merge bal-up$_i$.simps(2) node$_i$-bal sub-heights(1) sub-heights(2) sub-heights(3) sub-node t-node)*
ultimately show *?thesis*
apply *(cases node$_i$ k (mts@(mt,sep)#tts) tt)*
using *assms Nil sub-node t-node* **by** *auto*
next
case *(Cons r rs)*
then obtain *rsub rsep* **where** *r-split: r = (rsub,rsep)* **by** *(cases r)*
then have *rsub-height: height rsub = height t bal rsub*
using *assms Cons* **by** *auto*
then obtain *rts rt* **where** *r-node: rsub = (Node rts rt)*
apply(*cases rsub*) **using** *t-node* **by** *simp*
have *height-up$_i$ (node$_i$ k (mts@(mt,sep)#rts) rt) = height (Node (mts@(mt,sep)#rts) rt)*
using *node$_i$-height* **by** *blast*
also have *... = Suc (height rt)*
by *(metis Un-iff ‹height rsub = height t› assms bal.simps(2) bal-split-last(1) height-bal-tree height-up$_i$.simps(2) height-list-split list.set-intros(1) Cons max.idem r-node r-split set-append some-child-sub(1) sub-heights(1) sub-node)*
also have *... = height rsub*
using *height-bal-tree r-node rsub-height(2)* **by** *fastforce*
finally have *1: height-up$_i$ (node$_i$ k (mts@(mt,sep)#rts) rt) = height rsub* **.**
moreover have *2: bal-up$_i$ (node$_i$ k (mts@(mt,sep)#rts) rt)*
by *(metis bal-list-merge bal-up$_i$.simps(2) node$_i$-bal r-node rsub-height(1) rsub-height(2) sub-heights(1) sub-heights(2) sub-node)*
ultimately show *?thesis*
proof *(cases node$_i$ k (mts@(mt,sep)#rts) rt)*
case *(T_i u)*
then have *bal (Node (ls@(u,rsep)#rs) t)*
using *1 2 Cons assms t-node subtrees-split sub-heights r-split rsub-height*
unfolding *bal.simps* **by** *(auto simp del: height-bplustree.simps)*
then show *?thesis*
using *Cons assms t-node sub-node r-split r-node False T_i*
by *(auto simp del: node$_i$.simps bal.simps)*
next
case *(Up$_i$ l a r)*
then have *bal (Node (ls@(l,a)#(r,rsep)#rs) t)*
using *1 2 Cons assms t-node subtrees-split sub-heights r-split rsub-height*

```
      unfolding bal.simps by (auto simp del: height-bplustree.simps)
    then show ?thesis
      using Cons assms t-node sub-node r-split r-node False Up_i
      by (auto simp del: node_i.simps bal.simps)
  qed
 qed
 qed
qed


lemma rebalance-last-tree-bal: ⟦bal (Node ts t); ts ≠ []⟧ ⟹ bal (rebalance-last-tree
k ts t)
 using rebalance-middle-tree-bal append-butlast-last-id[of ts]
 apply(cases last ts)
 apply(auto simp del: bal.simps rebalance-middle-tree.simps)
 done

lemma Leaf-merge-aligned: aligned l (Leaf ms) m ⟹ aligned m (Leaf rs) r ⟹
aligned l (Leaf (ms@rs)) r
 by auto

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
 apply(induction mts arbitrary: l)
 apply auto
 done

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
 apply (induction ts' arbitrary: l)
 apply auto
 done

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
 apply(induction ts arbitrary: l)
 apply auto
 done

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
 apply (induction ls arbitrary: l)
 apply auto
 done
```

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*
apply(*induction ls arbitrary*: *l*)
apply *auto*
done

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 (*cases t*)
case *t-node*: (*Leaf txs*)
then obtain *mxs* **where** *sub-node*: *sub* = *Leaf mxs*
using *assms* **by** (*cases sub*) (*auto simp add*: *t-node*)
show *?thesis*
proof (*cases length mxs* ≥ *k* ∧ *length txs* ≥ *k*)
case *True*
then show *?thesis*
using *t-node sub-node assms*
by (*auto simp del*: *bal.simps*)
next
case *False*
then show *?thesis*
proof (*cases rs*)
case *rs-nil*: *Nil*
then have *sorted-leaves*: *sorted-less* (*mxs*@*txs*)
using *assms*(*3*) *rs-nil t-node sub-node sorted-wrt-append*
by *auto*
then show *?thesis*
proof (*cases ls*)
case *ls-nil*: *Nil*
then have *aligned l* (*Leaf* (*mxs*@*txs*)) *u*
using *t-node sub-node assms rs-nil False*
using *assms*
by *auto*
then have *aligned-up*$_i$ *l* (*Lnode*$_i$ *k* (*mxs*@*txs*)) *u*
using *Lnode*$_i$*-aligned sorted-leaves assms* **by** *blast*
then show *?thesis*
using *False t-node sub-node rs-nil ls-nil*
by (*auto simp del*: *Lnode*$_i$*.simps split*!: *up*$_i$*.split*)
next
case *Cons*
then obtain *ls' lsub lsep* **where** *ls-Cons*: *ls* = *ls'*@[(*lsub*,*lsep*)]

by (*metis list.discI old.prod.exhaust snoc-eq-iff-butlast*)
then have *aligned lsep* (*Leaf* (*mxs*@*txs*)) *u*
using *Leaf-merge-aligned*
using *align-last aligned-split-left assms*(*1*) *t-node rs-nil sub-node*
by *blast*
moreover have *sorted-less* (*mxs*@*txs*)
using *assms*(*3*) *rs-nil t-node sub-node*
by (*auto simp add*: *sorted-wrt-append*)
ultimately have *aligned-up*$_i$ *lsep* (*Lnode*$_i$ *k* (*mxs*@*txs*)) *u*
using *Lnode*$_i$*-aligned assms*(*4*) **by** *blast*
then show *?thesis*
using *False t-node sub-node rs-nil ls-Cons assms*
using *aligned-subst-last-merge*[*of l ls*′ *lsub lsep sub sep t u*]
using *aligned-subst-last-merge-two*[*of l ls*′ *lsub lsep sub sep t u*]
by (*auto simp del*: *Lnode*$_i$*.simps split*!: *up*$_i$*.split*)
qed
next
case *rs-Cons*: (*Cons r rs*)
then obtain *rsub rsep* **where** *r-split*[*simp*]: *r* = (*rsub*,*rsep*) **by** (*cases r*)
then have *height rsub* = *0*
using ‹⋀*thesis*. (⋀*mxs*. *sub* = *Leaf mxs* ⟹ *thesis*) ⟹ *thesis*› *assms*(*2*) *assms*(*5*) *rs-Cons*
by *fastforce*
then obtain *rxs* **where** *rs-Leaf*[*simp*]: *rsub* = *Leaf rxs*
by (*cases rsub*) *auto*
then have *sorted-leaves*: *sorted-less* (*mxs*@*rxs*)
using *assms*(*3*) *rs-Cons sub-node sorted-wrt-append r-split*
by (*auto simp add*: *sorted-wrt-append*)
then show *?thesis*
proof (*cases ls*)
case *ls-nil*: *Nil*
then have *aligned l* (*Leaf* (*mxs*@*rxs*)) *rsep*
using *sub-node assms rs-Cons False*
by *auto*
then have *aligned-up*$_i$ *l* (*Lnode*$_i$ *k* (*mxs*@*rxs*)) *rsep*
using *Lnode*$_i$*-aligned sorted-leaves assms* **by** *blast*
then show *?thesis*
using *False t-node sub-node rs-Cons ls-nil assms*
by (*auto simp del*: *Lnode*$_i$*.simps split*!: *up*$_i$*.split*)
next
case *Cons*
then obtain *ls*′ *lsub lsep* **where** *ls-Cons*: *ls* = *ls*′@[(*lsub*,*lsep*)]
by (*metis list.discI old.prod.exhaust snoc-eq-iff-butlast*)
then have *aligned lsep* (*Leaf* (*mxs*@*rxs*)) *rsep*
using *Leaf-merge-aligned*
using *align-last aligned-split-left assms*(*1*) *t-node rs-Cons sub-node*
by (*metis aligned.elims*(*2*) *aligned-split-right bplustree.distinct*(*1*) *bplustree.inject*(*2*) *inbetween.simps*(*2*) *r-split rs-Leaf*)
then have *aligned-up*$_i$ *lsep* (*Lnode*$_i$ *k* (*mxs*@*rxs*)) *rsep*

```
        using Lnode_i-aligned assms(4) sorted-leaves by blast
      then show ?thesis
        using False t-node sub-node rs-Cons ls-Cons assms
        using aligned-subst-merge[of l ls' lsub lsep sub sep rsub rsep rs]
        using aligned-subst-merge-two[of l ls' lsub lsep sub sep rsub rsep rs t u]
        by (auto simp del: Lnode_i.simps split!: up_i.split)
    qed
   qed
  qed
next
  case t-node: (Node tts tt)
  then obtain mts mt where sub-node: sub = Node mts mt
    using assms by (cases sub) (auto simp add: t-node)
  show ?thesis
  proof (cases length tts ≥ k ∧ length mts ≥ k)
    case True
    then show ?thesis
      using t-node sub-node assms
      by (auto simp del: bal.simps)
  next
    case False
    then show ?thesis
    proof (cases rs)
      case rs-nil: Nil
      then have sorted-leaves: sorted-less (leaves-list mts @ leaves mt @ leaves-list
tts @ leaves tt)
        using assms(3) rs-nil t-node sub-node
        by (auto simp add: sorted-wrt-append)
      then show ?thesis
      proof (cases ls)
        case ls-nil: Nil
        then have aligned l (Node (mts@(mt,sep)#tts) tt) u
          using t-node sub-node assms rs-nil False
          by (auto simp add: Node-merge-aligned)
        then  have aligned-up_i l (node_i k (mts@(mt,sep)#tts) tt) u
          using node_i-aligned sorted-leaves assms by blast
        then show ?thesis
          using False t-node sub-node rs-nil ls-nil
          by (auto simp del: node_i.simps split!: up_i.split)
      next
        case Cons
        then obtain ls' lsub lsep where ls-Cons: ls = ls'@[(lsub,lsep)]
          by (metis list.discI old.prod.exhaust snoc-eq-iff-butlast)
        then have aligned lsep (Node (mts@(mt,sep)#tts) tt) u
          using t-node sub-node assms rs-nil False ls-Cons
         by (metis Node-merge-aligned align-last aligned.simps(2) aligned-split-left)
        then have aligned-up_i lsep (node_i k (mts@(mt,sep)#tts) tt) u
          using node_i-aligned assms(4) sorted-leaves by blast
        then show ?thesis
```

using *False t-node sub-node rs-nil ls-Cons assms*
using *aligned-subst-last-merge*[*of l ls′ lsub lsep sub sep t u*]
using *aligned-subst-last-merge-two*[*of l ls′ lsub lsep sub sep t u*]
by (*auto simp del*: $node_i$.*simps split*!: up_i.*split*)
qed
next
case *rs-Cons*: (*Cons r rs*)
then obtain *rsub rsep* **where** *r-split*[*simp*]: *r* = (*rsub*,*rsep*)
by (*cases r*)
then have *height rsub* ≠ *0*
using *assms rs-Cons t-node* **by** *auto*
then obtain *rts rt* **where** *rs-Node*: *rsub* = *Node rts rt*
by (*cases rsub*) *auto*
have *sorted-less* (*leaves sub* @ *leaves rsub*)
using *assms*(*3*) *rs-Cons r-split*
by (*simp add*: *sorted-wrt-append*)
then have *sorted-leaves*: *sorted-less* (*leaves-list mts* @ *leaves mt* @ *leaves-list rts* @ *leaves rt*)
by (*simp add*: *rs-Node sub-node*)
then show *?thesis*
proof (*cases ls*)
case *ls-nil*: *Nil*
then have *aligned l* (*Node* (*mts*@(*mt*,*sep*)#*rts*) *rt*) *rsep*
using *sub-node assms rs-Cons False rs-Node*
by (*metis Node-merge-aligned aligned.simps*(*2*) *append-self-conv2 inbetween.simps*(*2*) *r-split*)
then have *aligned-up*$_i$ *l* (*node*$_i$ *k* (*mts*@(*mt*,*sep*)#*rts*) *rt*) *rsep*
using *node*$_i$*-aligned sorted-leaves assms* **by** *blast*
then show *?thesis*
using *False t-node sub-node rs-Cons ls-nil assms rs-Node*
by (*auto simp del*: $node_i$.*simps split*!: up_i.*split*)
next
case *Cons*
then obtain *ls′ lsub lsep* **where** *ls-Cons*: *ls* = *ls′*@[(*lsub*,*lsep*)]
by (*metis list.discI old.prod.exhaust snoc-eq-iff-butlast*)
then have *aligned lsep* (*Node* (*mts*@(*mt*,*sep*)#*rts*) *rt*) *rsep*
using *Node-merge-aligned*
using *align-last aligned-split-left assms*(*1*) *t-node rs-Cons sub-node*
by (*metis aligned.simps*(*2*) *aligned-split-right inbetween.simps*(*2*) *r-split rs-Node*)
then have *aligned-up*$_i$ *lsep* (*node*$_i$ *k* (*mts*@(*mt*,*sep*)#*rts*) *rt*) *rsep*
using *sorted-leaves node*$_i$*-aligned assms*(*4*) **by** *blast*
then show *?thesis*
using *False t-node sub-node rs-Cons ls-Cons assms rs-Node*
using *aligned-subst-merge*[*of l ls′ lsub lsep sub sep rsub rsep rs*]
using *aligned-subst-merge-two*[*of l ls′ lsub lsep sub sep rsub rsep rs t u*]
by (*auto simp del*: $node_i$.*simps split*!: up_i.*split*)
qed
qed

qed
qed

lemma *Node-merge-Laligned*:
Laligned (*Node mts mt*) *sep* ⟹
inbetween aligned sep tts tt u ⟹
Laligned (*Node* (*mts* @ (*mt*, *sep*) # *tts*) *tt*) *u*
apply(*induction mts*)
apply *auto*
using *Node-merge-aligned* **by** *blast*

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*
apply (*induction ts'*)
apply *auto*
by (*metis* (*no-types*, *opaque-lifting*) *Node-merge-aligned aligned.simps*(*2*) *aligned-split-left*
inbetween.simps(*1*))

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*
apply(*induction ts*)
apply *auto*
by (*meson aligned.simps*(*2*) *aligned-subst-last-merge-two*)

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*
apply (*induction ls*)
apply *auto*
by (*meson aligned.simps*(*2*) *aligned-subst-merge*)

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*
apply(*induction ls*)
apply *auto*
by (*meson aligned.simps*(*2*) *aligned-subst-merge-two*)

lemma *xs-front*: *xs* @ [(*a*,*b*)] = (*x*,*y*)#*xs'* ⟹ *xs* @ [(*a*,*b*),(*c*,*d*)] = (*z*,*zz*)#*xs''* ⟹
(*x*,*y*) = (*z*,*zz*)
apply(*induction xs*)
apply *auto*
done

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 (cases t)
  case t-node: (Leaf txs)
  then obtain mxs where sub-node: sub = Leaf mxs
    using assms by (cases sub) (auto simp add: t-node)
  show ?thesis
  proof (cases length mxs ≥ k ∧ length txs ≥ k)
    case True
    then show ?thesis
      using t-node sub-node assms
      by auto
  next
    case False
    then show ?thesis
    proof (cases rs)
      case rs-nil: Nil
      then have sorted-leaves: sorted-less (mxs@txs)
        using assms(3) rs-nil t-node sub-node sorted-wrt-append
        by auto
      then show ?thesis
      proof (cases ls)
        case ls-nil: Nil
        then have Laligned (Leaf (mxs@txs)) u
          using t-node sub-node assms rs-nil False
          using assms
          by auto
        then  have Laligned-up_i (Lnode_i k (mxs@txs)) u
          using Lnode_i-Laligned sorted-leaves assms by blast
        then show ?thesis
          using False t-node sub-node rs-nil ls-nil
          by (auto simp del: Lnode_i.simps split!: up_i.split)
      next
        case Cons
        then obtain ls' lsub lsep where ls-Cons: ls = ls'@[(lsub,lsep)]
          by (metis list.discI old.prod.exhaust snoc-eq-iff-butlast)
        then have aligned lsep (Leaf (mxs@txs)) u
            using Leaf-merge-aligned Lalign-last Laligned-split-left assms(1) rs-nil
sub-node t-node
          by blast
        moreover have sorted-less (mxs@txs)
          using assms(3) rs-nil t-node sub-node
          by (auto simp add: sorted-wrt-append)
        ultimately have aligned-up_i lsep (Lnode_i k (mxs@txs)) u
          using Lnode_i-aligned assms(4) by blast
        then show ?thesis
```

```
        using False t-node sub-node rs-nil ls-Cons assms
        using Laligned-subst-last-merge[of ls' lsub lsep sub sep t u]
        using Laligned-subst-last-merge-two[of ls' lsub lsep sub sep t u]
        by (auto simp del: Lnode_i.simps split!: up_i.split)
      qed
    next
      case rs-Cons: (Cons r rs)
      then obtain rsub rsep where r-split[simp]: r = (rsub,rsep) by (cases r)
      then have height rsub = 0
        using ‹⋀thesis. (⋀mxs. sub = Leaf mxs ⟹ thesis) ⟹ thesis› assms(2)
assms(5) rs-Cons
        by fastforce
      then obtain rxs where rs-Leaf[simp]: rsub = Leaf rxs
        by (cases rsub) auto
      then have sorted-leaves: sorted-less (mxs@rxs)
        using assms(3) rs-Cons sub-node sorted-wrt-append r-split
        by (auto simp add: sorted-wrt-append)
      then show ?thesis
      proof (cases ls)
        case ls-nil: Nil
        then have Laligned (Leaf (mxs@rxs)) rsep
          using sub-node assms rs-Cons False
          by auto
        then  have Laligned-up_i (Lnode_i k (mxs@rxs)) rsep
          using Lnode_i-Laligned sorted-leaves assms by blast
        then show ?thesis
          using False t-node sub-node rs-Cons ls-nil assms
          by (auto simp del: Lnode_i.simps split!: up_i.split)
      next
        case Cons
        then obtain ls' lsub lsep where ls-Cons: ls = ls'@[(lsub,lsep)]
          by (metis list.discI old.prod.exhaust snoc-eq-iff-butlast)
        then have aligned lsep (Leaf (mxs@rxs)) rsep
          using Leaf-merge-aligned
          using assms(1) t-node rs-Cons sub-node
         by (metis Lalign-last Laligned-split-left Laligned-split-right aligned.elims(2)
bplustree.distinct(1) bplustree.inject(2) inbetween.simps(2) r-split rs-Leaf)
        then have aligned-up_i lsep (Lnode_i k (mxs@rxs)) rsep
          using Lnode_i-aligned assms(4) sorted-leaves by blast
        then show ?thesis
          using False t-node sub-node rs-Cons ls-Cons assms
          using Laligned-subst-merge[of ls' lsub lsep sub sep rsub rsep rs]
          using Laligned-subst-merge-two[of ls' lsub lsep sub sep rsub rsep rs t u]
          by (auto simp del: Lnode_i.simps split!: up_i.split)
      qed
    qed
  qed
next
  case t-node: (Node tts tt)
```

```
  then obtain mts mt where sub-node: sub = Node mts mt
    using assms by (cases sub) (auto simp add: t-node)
  show ?thesis
  proof (cases length tts ≥ k ∧ length mts ≥ k)
    case True
    then show ?thesis
      using t-node sub-node assms
      by (auto simp del: bal.simps)
  next
    case False
    then show ?thesis
    proof (cases rs)
      case rs-nil: Nil
      then have sorted-leaves: sorted-less (leaves-list mts @ leaves mt @ leaves-list
tts @ leaves tt)
        using assms(3) rs-nil t-node sub-node
        by (auto simp add: sorted-wrt-append)
      then show ?thesis
      proof (cases ls)
        case ls-nil: Nil
        then have Laligned (Node (mts@(mt,sep)#tts) tt) u
          using t-node sub-node assms rs-nil False
       by (metis Lalign-last Laligned-nonempty-Node Node-merge-Laligned aligned.simps(2)
append-self-conv2)
        then  have Laligned-up_i (node_i k (mts@(mt,sep)#tts) tt) u
          using node_i-Laligned sorted-leaves assms by blast
        then show ?thesis
          using False t-node sub-node rs-nil ls-nil
          by (auto simp del: node_i.simps split!: up_i.split)
      next
        case Cons
        then obtain ls' lsub lsep where ls-Cons: ls = ls'@[(lsub,lsep)]
          by (metis list.discI old.prod.exhaust snoc-eq-iff-butlast)
        then have aligned lsep (Node (mts@(mt,sep)#tts) tt) u
          using t-node sub-node assms rs-nil False ls-Cons
       by (metis Lalign-last Laligned-split-left Node-merge-aligned aligned.simps(2))
        then have aligned-up_i lsep (node_i k (mts@(mt,sep)#tts) tt) u
          using node_i-aligned assms(4) sorted-leaves by blast
        then show ?thesis
          using False t-node sub-node rs-nil ls-Cons assms
          using Laligned-subst-last-merge[of ls' lsub lsep sub sep t u]
          using Laligned-subst-last-merge-two[of ls' lsub lsep sub sep t u]
            by (auto simp del: node_i.simps bal.simps height-bplustree.simps split!:
up_i.split list.splits)
      qed
    next
      case rs-Cons: (Cons r rs)
      then obtain rsub rsep where r-split[simp]: r = (rsub,rsep)
        by (cases r)
```

```
      then have height rsub ≠ 0
        using assms rs-Cons t-node by auto
      then obtain rts rt where rs-Node: rsub = Node rts rt
        by (cases rsub) auto
      have sorted-less (leaves sub @ leaves rsub)
        using assms(3) rs-Cons r-split
        by (simp add: sorted-wrt-append)
      then have sorted-leaves: sorted-less (leaves-list mts @ leaves mt @ leaves-list
rts @ leaves rt)
        by (simp add: rs-Node sub-node)
      then show ?thesis
      proof (cases ls)
        case ls-nil: Nil
        then have Laligned (Node (mts@(mt,sep)#rts) rt) rsep
          using sub-node assms rs-Cons False rs-Node
          by (metis Laligned-nonempty-Node Node-merge-Laligned aligned.simps(2)
append-self-conv2 inbetween.simps(2) r-split)
        then  have Laligned-up_i (node_i k (mts@(mt,sep)#rts) rt) rsep
          using node_i-Laligned by blast
        then show ?thesis
          using False t-node sub-node rs-Cons ls-nil assms rs-Node
          by (auto simp del: node_i.simps split!: up_i.split)
      next
        case Cons
        then obtain ls' lsub lsep where ls-Cons: ls = ls'@[(lsub,lsep)]
          by (metis list.discI old.prod.exhaust snoc-eq-iff-butlast)
        then have aligned lsep (Node (mts@(mt,sep)#rts) rt) rsep
          using Node-merge-aligned
          using assms(1) t-node rs-Cons sub-node
         by (metis Lalign-last Laligned-split-left Laligned-split-right aligned.simps(2)
inbetween.simps(2) r-split rs-Node)
        then have aligned-up_i lsep (node_i k (mts@(mt,sep)#rts) rt) rsep
          using sorted-leaves node_i-aligned assms(4) by blast
        then show ?thesis
          using False t-node sub-node rs-Cons ls-Cons assms rs-Node
          using Laligned-subst-merge[of ls' lsub lsep sub sep rsub rsep rs]
          using Laligned-subst-merge-two[of ls' lsub lsep sub sep rsub rsep rs t u]
          by (auto simp del: node_i.simps split!: up_i.split)
      qed
    qed
  qed
qed

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
```

using *rebalance-middle-tree-aligned*[*of l ls sub sep* [] *t u k*] *assms*
by *auto*

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*
using *rebalance-middle-tree-Laligned*[*of ls sub sep* [] *t u k*] *assms*
by *auto*

lemma *del-bal*:
assumes *k* > *0*
and *root-order k t*
and *bal t*
shows *bal* (*del k x t*)
using *assms*
proof(*induction k x t rule*: *del.induct*)
case (*2 k x ts t*)
then obtain *ls rs* **where** *list-split*: *split ts x* = (*ls*,*rs*)
by (*cases split ts x*)
then show *?case*
proof (*cases rs*)
case *Nil*
then have *bal* (*del k x t*) **using** *2 list-split*
by (*simp add*: *order-impl-root-order*)
moreover have *height* (*del k x t*) = *height t*
using *2 del-height* **by** (*simp add*: *order-impl-root-order*)
moreover have *ts* ≠ [] **using** *2* **by** *auto*
ultimately have *bal* (*rebalance-last-tree k ts* (*del k x t*))
using *2 Nil rebalance-last-tree-bal*
by *simp*
then have *bal* (*rebalance-last-tree k ls* (*del k x t*))
using *list-split split-conc Nil* **by** *fastforce*
then show *?thesis*
using *2 list-split Nil*
by *auto*
next
case (*Cons r rs*)
then obtain *sub sep* **where** *r-split*: *r* = (*sub*,*sep*) **by** (*cases r*)
then have *sub-height*: *height sub* = *height t bal sub*
using *2 Cons list-split split-set*(*1*) **by** *fastforce*+
then have *bal* (*del k x sub*) *height* (*del k x sub*) = *height sub* **using** *sub-height*
apply (*metis 2.IH*(*2*) *2.prems*(*1*) *2.prems*(*2*) *list-split local.Cons order-impl-root-order r-split root-order.simps*(*2*) *some-child-sub*(*1*) *split-set*(*1*))
by (*metis 2.prems*(*1*) *2.prems*(*2*) *list-split Cons order-impl-root-order r-split root-order.simps*(*2*) *some-child-sub*(*1*) *del-height split-set*(*1*) *sub-height*(*2*))
moreover have *bal* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)

```
    using 2.prems(3) list-split Cons r-split split-conc by blast
  ultimately have bal (Node (ls@(del k x sub,sep)#rs) t)
    using bal-substitute-subtree[of ls sub sep rs t del k x sub] by metis
  then have bal (rebalance-middle-tree k ls (del k x sub) sep rs t)
    using rebalance-middle-tree-bal[of ls del k x sub sep rs t k] by metis
  then show ?thesis
    using 2 list-split Cons r-split by auto
 qed
qed simp


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(cases t)
 case (Leaf txs)
 then obtain subxs where sub = Leaf subxs
  using height-Leaf assms by metis
 then show ?thesis
  using assms Leaf
  by (auto split!: list.splits bplustree.splits)
next
 case t-node: (Node tts tt)
 then obtain mts mt where sub-node: sub = Node mts mt
  using assms by (cases sub) (auto)
 then show ?thesis
 proof(cases length mts ≥ k ∧ length tts ≥ k)
  case True
  then have order k sub using assms
   by (simp add: sub-node)
  then show ?thesis
   using True t-node sub-node assms by auto
 next
  case False
  then show ?thesis
  proof (cases rs)
   case Nil
   have order-up_i k (node_i k (mts@(mt,sep)#tts) tt)
    using node_i-order[of k mts@(mt,sep)#tts tt] assms(1,3) t-node sub-node
    by (auto simp del: order-up_i.simps node_i.simps)
   then show ?thesis
    apply(cases node_i k (mts@(mt,sep)#tts) tt)
    using assms t-node sub-node False Nil apply (auto simp del: node_i.simps)
    done
  next
```

```
    case (Cons r rs)
    then obtain rsub rsep where r-split: r = (rsub,rsep) by (cases r)
    then have rsub-height: height rsub = height t
      using assms Cons by auto
    then obtain rts rt where r-node: rsub = (Node rts rt)
      apply(cases rsub) using t-node by simp
    have order-up_i k (node_i k (mts@(mt,sep)#rts) rt)
      using node_i-order[of k mts@(mt,sep)#rts rt] assms(1,2) t-node sub-node
r-node r-split Cons
      by (auto simp del: order-up_i.simps node_i.simps)
    then show ?thesis
      apply(cases node_i k (mts@(mt,sep)#rts) rt)
       using assms t-node sub-node False Cons r-split r-node apply (auto simp
del: node_i.simps)
      done
  qed
 qed
qed


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 (cases t)
 case (Leaf txs)
 then obtain subxs where sub = Leaf subxs
   using height-Leaf assms by metis
 then show ?thesis
   using assms Leaf
   by (auto split!: list.splits bplustree.splits)
next
 case t-node: (Node tts tt)
 then obtain mts mt where sub-node: sub = Node mts mt
   using assms by (cases sub) (auto)
 then show ?thesis
 proof(cases length mts ≥ k ∧ length tts ≥ k)
   case True
   then have order k sub using assms
     by (simp add: sub-node)
   then show ?thesis
     using True t-node sub-node assms by auto
 next
   case False
   have order-up_i k (node_i k (mts@(mt,sep)#tts) tt)
     using node_i-order[of k mts@(mt,sep)#tts tt] assms t-node sub-node
```

```
    by (auto simp del: order-up_i.simps node_i.simps)
  then show ?thesis
    apply(cases node_i k (mts@(mt,sep)#tts) tt)
    using assms t-node sub-node False Nil apply (auto simp del: node_i.simps)
    done
 qed
qed

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)
 using rebalance-middle-tree-last-order assms by auto


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)
 using assms
proof (induction k x t rule: del.induct)
 case (1 k x xs)
 then show ?case
  by auto
next
 case (2 k x ts t)
 then obtain ls list where list-split: split ts x = (ls, list) by (cases split ts x)
 then show ?case
 proof (cases list)
  case Nil
  then have almost-order k (del k x t) using 2 list-split
   by (simp add: order-impl-root-order sorted-wrt-append)
  moreover obtain lls lsub lsep where ls-split: ls = lls@[(lsub,lsep)]
   using 2 Nil list-split
  by (metis append-Nil length-0-conv less-nat-zero-code old.prod.exhaust rev-exhaust
root-order.simps(2) split-conc)
  moreover have height t = height (del k x t) using del-height 2
   by (simp add: order-impl-root-order)
  moreover have length ls = length ts
   using Nil list-split
   by (auto dest: split-length)
  ultimately have almost-order k (rebalance-last-tree k ls (del k x t))
   using rebalance-last-tree-order[of ls lls lsub lsep k del k x t]
    by (metis 2.prems(2) 2.prems(3) Un-iff append-Nil2 bal.simps(2) list-split
Nil root-order.simps(2) singletonI split-conc subtrees-split)
```

```
    then show ?thesis
      using 2 list-split Nil by auto
  next
    case (Cons r rs)

    from Cons obtain sub sep where r-split: r = (sub,sep) by (cases r)

    have inductive-help:
      case rs of [] ⇒ True | (rsub,rsep)#- ⇒ height rsub = height t
      ∀ s∈set (subtrees (ls @ rs)). order k s
      Suc (length (ls @ rs)) ≤ 2 * k
      order k t
      using Cons r-split 2.prems list-split split-set
      by (auto dest: split-conc split!: list.splits)
     then have almost-order k (del k x sub) using 2 list-split Cons r-split or-
der-impl-root-order
    by (metis bal.simps(2) root-order.simps(2) some-child-sub(1) sorted-leaves-induct-subtree
split-conc split-set(1))
    moreover have height (del k x sub) = height t
      by (metis 2.prems(1) 2.prems(2) 2.prems(3) bal.simps(2) list-split Cons or-
der-impl-root-order r-split root-order.simps(2) some-child-sub(1) del-height split-set(1))
   ultimately have almost-order k (rebalance-middle-tree k ls (del k x sub) sep rs
t)
      using rebalance-middle-tree-order[of k del k x sub ls rs t sep]
      using inductive-help
      using Cons r-split list-split by auto
    then show ?thesis using 2 Cons r-split list-split by auto
  qed
qed




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 (cases ls)
  case Nil
  then show ?thesis
    using assms by (auto dest!: split-conc)
next
  case Cons
  then obtain ls' sub sep where ls-tail-split: ls = ls' @ [(sub,sep)]
    by (metis list.distinct(1) rev-exhaust surj-pair)
```

```
  have sorted-inorder: sorted-less (inorder (Node ts t))
    using Laligned-sorted-inorder assms(1) sorted-cons sorted-snoc by blast
  moreover have sep < x
    using split-req(2)[of ts x ls' sub sep rs]
    using assms ls-tail-split sorted-inorder sorted-inorder-separators
    by blast
  moreover have leaves-split: leaves (Node ts t) = leaves-list ls @ leaves-list rs @
leaves t
    using assms(3) split-tree.leaves-split by blast
  then show ?thesis
  proof (cases leaves-list ls)
    case Nil
    then show ?thesis
      by (metis append-self-conv2 leaves-split)
  next
    case Cons
    then obtain leavesls' l' where leaves-tail-split: leaves-list ls = leavesls' @ [l']
      by (metis list.simps(3) rev-exhaust)
    then have l' ≤ sep
    proof −
      have l' ∈ set (leaves-list ls)
        using leaves-tail-split by force
      then have l' ∈ set (leaves (Node ls' sub))
        using ls-tail-split
        by auto
      moreover have Laligned (Node ls' sub) sep
        using assms split-conc[OF assms(3)] Cons ls-tail-split
        using Laligned-split-left
        by simp
      ultimately show ?thesis
        using Laligned-leaves-inbetween[of Node ls' sub sep]
        by blast
    qed
  moreover have sorted-less (leaves (Node ts t))
    using assms sorted-wrt-append split-conc by fastforce
 ultimately show ?thesis using assms(2) split-conc[OF assms(3)] leaves-tail-split
    using del-list-sorted[of leavesls' l' leaves-list rs @ leaves t x] ‹sep < x›
    by auto
  qed
qed

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)
  using aligned-imp-Laligned assms(1) assms(2) assms(3) del-list-split by blast
```

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 −
have *sorted-inorder*: *sorted-less* (*inorder* (*Node ts t*))
using *Laligned-sorted-inorder assms*(*1*) *sorted-cons sorted-snoc* **by** *blast*
from *assms* **have** *x* ≤ *sep*
proof −
from *assms* **have** *sorted-less* (*separators ts*)
using *sorted-inorder-separators sorted-inorder* **by** *blast*
then show *?thesis*
using *split-req*(*3*)
using *assms*
by *fastforce*
qed
then show *?thesis*
proof (*cases leaves-list rs* @ *leaves t*)
case *Nil*
moreover have *leaves-list* ((*sub*,*sep*)#*rs*) @ *leaves t* = *leaves sub* @ *leaves-list rs* @ *leaves t*
by *simp*
ultimately show *?thesis*
by (*metis self-append-conv*)
next
case (*Cons r′ rs′*)
then have *sep* < *r′*
by (*metis aligned-leaves-inbetween Laligned-split-right assms*(*1*) *assms*(*3*) *leaves.simps*(*2*) *list.set-intros*(*1*) *split-set.split-conc split-set-axioms*)
then have *x* < *r′*
using ‹*x* ≤ *sep*› **by** *auto*
moreover have *sorted-less* (*leaves sub* @ *leaves-list rs* @ *leaves t*)
proof −
have *sorted-less* (*leaves-list ls* @ *leaves sub* @ *leaves-list rs* @ *leaves t*)
using *assms*
by (*auto dest*!: *split-conc*)
then show *?thesis*
using *assms*
by (*metis Cons sorted-wrt-append*)
qed
ultimately show *?thesis*
using *del-list-sorted*[*of leaves sub r′ rs′*] *Cons*
by *auto*
qed
qed

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*
using *aligned-imp-Laligned assms*(*1*) *assms*(*2*) *assms*(*3*) *split-set.del-list-split-right split-set-axioms* **by** *blast*

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* ≤ *u*
shows *leaves* (*del k x t*) = *del-list x* (*leaves t*) ∧ *aligned l* (*del k x t*) *u*
using *assms*
proof (*induction k x t arbitrary*: *l u rule*: *del.induct*)
case (*1 k x xs*)
then have *leaves* (*del k x* (*Leaf xs*)) = *del-list x* (*leaves* (*Leaf xs*))
by (*simp add*: *insert-list-req*)
moreover have *aligned l* (*del k x* (*Leaf xs*)) *u*
proof −
have *l* < *u*
using *1.prems*(*6*) *1.prems*(*7*) **by** *auto*
moreover have ∀ *x* ∈ *set xs* − {*x*}. *l* < *x* ∧ *x* ≤ *u*
using *1.prems*(*5*) **by** *auto*
ultimately show *?thesis*
using *set-del-list insert-list-req*
by (*metis 1*(*4*) *aligned.simps*(*1*) *del.simps*(*1*) *leaves.simps*(*1*))
qed
ultimately show *?case*
by *simp*
next
case (*2 k x ts t l u*)
then obtain *ls rs* **where** *list-split*: *split ts x* = (*ls*, *rs*)
by (*meson surj-pair*)
then have *list-conc*: *ts* = *ls* @ *rs*
using *split-set.split-conc split-set-axioms* **by** *blast*
show *?case*
proof (*cases rs*)
case *Nil*
then obtain *ls′ lsub lsep* **where** *ls-split*: *ls* = *ls′* @ [(*lsub*,*lsep*)]
by (*metis 2.prems*(*2*) *append-Nil2 list.size*(*3*) *list-conc old.prod.exhaust root-order.simps*(*2*) *snoc-eq-iff-butlast zero-less-iff-neq-zero*)

then have *IH*: *leaves* (*del k x t*) = *del-list x* (*leaves t*) ∧ *aligned lsep* (*del k x t*) *u*
using *2.IH*(*1*)[*OF list-split*[*symmetric*] *Nil*, *of lsep u*]
by (*metis* (*no-types*, *lifting*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *2.prems*(*4*) *2.prems*(*5*) *2.prems*(*7*) ‹*ls* = *ls'* @ [(*lsub*, *lsep*)]› *align-last aligned-sorted-separators bal.simps*(*2*) *list-conc list-split local.Nil order-impl-root-order root-order.simps*(*2*) *self-append-conv sorted-cons sorted-leaves-induct-last sorted-snoc split-set.split-req*(*2*) *split-set-axioms*)
have *leaves* (*del k x* (*Node ts t*)) = *leaves* (*rebalance-last-tree k ts* (*del k x t*))
using *list-split Nil list-conc* **by** *auto*
also have ... = *leaves-list ts* @ *leaves* (*del k x t*)
proof −
obtain *ts' sub sep* **where** *ts-split*: *ts* = *ts'* @ [(*sub*, *sep*)]
using ‹*ls* = *ls'* @ [(*lsub*, *lsep*)]› *list-conc local.Nil* **by** *blast*
then have *height sub* = *height t*
using *2.prems*(*3*) **by** *auto*
moreover have *height t* = *height* (*del k x t*)
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *bal.simps*(*2*) *del-height order-impl-root-order root-order.simps*(*2*))
ultimately show *?thesis*
using *rebalance-last-tree-inorder*
using *ts-split* **by** *auto*
qed
also have ... = *leaves-list ts* @ *del-list x* (*leaves t*)
using *IH* **by** *blast*
also have ... = *del-list x* (*leaves* (*Node ts t*))
by (*metis 2.prems*(*4*) *2.prems*(*5*) *aligned-imp-Laligned append-self-conv2 concat.simps*(*1*) *list.simps*(*8*) *list-conc list-split local.Nil self-append-conv split-set.del-list-split split-set-axioms*)
finally have *0*: *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*)) **.**
moreover have *aligned l* (*del k x* (*Node ts t*)) *u*
proof −
have *aligned l* (*Node ls* (*del k x t*)) *u*
using *IH list-conc Nil 2.prems ls-split*
using *aligned-subst-last*
by (*metis self-append-conv*)
moreover have *sorted-less* (*leaves* (*Node ls* (*del k x t*)))
using *2.prems*(*4*) ‹*leaves-list ts* @ *del-list x* (*leaves t*) = *del-list x* (*leaves* (*Node ts t*))› ‹*leaves-list ts* @ *leaves* (*del k x t*) = *leaves-list ts* @ *del-list x* (*leaves t*)› *list-conc local.Nil sorted-del-list*
by *auto*
ultimately have *aligned l* (*rebalance-last-tree k ls* (*del k x t*)) *u*
using *rebalance-last-tree-aligned*
by (*metis* (*no-types*, *lifting*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *UnCI bal.simps*(*2*) *del-height list.set-intros*(*1*) *list-conc ls-split order-impl-root-order root-order.simps*(*2*) *set-append some-child-sub*(*1*))
then show *?thesis* **using** *list-split ls-split 2.prems Nil*
by *simp*
qed

ultimately show *?thesis*
by *simp*
next
case (*Cons h rs*)
then obtain *sub sep* **where** *h-split*: *h* = (*sub*,*sep*)
by (*cases h*)
then have *node-sorted-split*:
sorted-less (*leaves* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*))
root-order k (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
bal (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
using *2.prems h-split list-conc Cons* **by** *blast+*
{
assume *IH*: *leaves* (*del k x sub*) = *del-list x* (*leaves sub*)
have *leaves* (*del k x* (*Node ts t*)) = *leaves* (*rebalance-middle-tree k ls* (*del k x sub*) *sep rs t*)
using *Cons list-split h-split 2.prems*
by *auto*
also have ... = *leaves* (*Node* (*ls*@(*del k x sub*, *sep*)#*rs*) *t*)
using *rebalance-middle-tree-inorder*[*of t del k x sub rs*]
by (*smt* (*verit*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *bal.simps*(*2*) *bal-sub-height del-height h-split list-split local.Cons node-sorted-split*(*3*) *order-impl-root-order rebalance-middle-tree-inorder root-order.simps*(*2*) *some-child-sub*(*1*) *split-set*(*1*))
also have ... = *leaves-list ls* @ *leaves* (*del k x sub*) @ *leaves-list rs* @ *leaves t*
by *auto*
also have ... = *leaves-list ls* @ *del-list x* (*leaves sub* @ *leaves-list rs* @ *leaves t*)
using *del-list-split-right-aligned*[*of l ts t u x ls sub sep rs*]
using *list-split Cons 2.prems*(*4*,*5*) *h-split IH list-conc*
by *auto*
also have ... = *del-list x* (*leaves-list ls* @ *leaves sub* @ *leaves-list rs* @ *leaves t*)
using *del-list-split-aligned*[*of l ts t u x ls* (*sub*,*sep*)#*rs*]
using *list-split Cons 2.prems*(*4*,*5*) *h-split IH list-conc*
by *auto*
finally have *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))
using *list-conc Cons h-split*
by *auto*
}
then show *?thesis*
proof (*cases ls*)
case *Nil*
then have *IH*: *leaves* (*del k x sub*) = *del-list x* (*leaves sub*) ∧ *aligned l* (*del k x sub*) *sep*
using *2.IH*(*2*)[*OF list-split*[*symmetric*] *Cons h-split*[*symmetric*], *of l sep*]
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*5*) *2.prems*(*6*) *aligned.simps*(*2*) *aligned-sorted-separators append-self-conv2 bal.simps*(*2*) *h-split inbetween.simps*(*2*) *list.set-intros*(*1*) *list-conc list-split local.Cons local.Nil node-sorted-split*(*1*) *node-sorted-split*(*3*) *order-impl-root-order root-order.simps*(*2*) *some-child-sub*(*1*) *sorted-cons sorted-leaves-induct-subtree sorted-snoc split-set.split-req*(*3*) *split-set-axioms*)

then have *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))
using ‹*leaves* (*del k x sub*) = *del-list x* (*leaves sub*) ⟹ *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))› **by** *blast*
then have *sorted-less* (*leaves* (*del k x* (*Node ts t*)))
using *2.prems*(*4*) *sorted-del-list* **by** *auto*
then have *sorted-leaves*: *sorted-less* (*leaves* (*Node* (*ls*@(*del k x sub*, *sep*)#*rs*) *t*))
using *list-split Cons h-split*
using *rebalance-middle-tree-inorder*[*of t del k x sub rs k ls sep*]
using *2.prems*(*4*) *2.prems*(*5*) *IH* ‹*leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))› *del-list-split-aligned del-list-split-right-aligned*
by *auto*
from *IH* **have** *aligned l* (*del k x* (*Node ts t*)) *u*
proof −
have *aligned l* (*Node* (*ls*@(*del k x sub*, *sep*)#*rs*) *t*) *u*
using *2.prems*(*5*) *IH h-split list-conc local.Cons local.Nil* **by** *auto*
then have *aligned l* (*rebalance-middle-tree k ls* (*del k x sub*) *sep rs t*) *u*
using *rebalance-middle-tree-aligned sorted-leaves*
by (*smt* (*verit*, *best*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *append-self-conv2 bal.simps*(*2*) *bal-sub-height del-height h-split list.set-intros*(*1*) *list-conc local.Cons local.Nil order-impl-root-order root-order.simps*(*2*) *some-child-sub*(*1*))
then show *?thesis*
using *list-split Cons h-split*
by *auto*
qed
then show *?thesis*
using ‹*leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))› **by** *blast*
next
case -: (*Cons a list*)
then obtain *ls′ lsub lsep* **where** *l-split*: *ls* = *ls′*@[(*lsub*,*lsep*)]
by (*metis list.discI old.prod.exhaust snoc-eq-iff-butlast*)
then have *aligned lsep sub sep*
using *2.prems*(*5*) *align-last aligned-split-left h-split list-conc local.Cons*
by *blast*
then have *IH*: *leaves* (*del k x sub*) = *del-list x* (*leaves sub*) ∧ *aligned lsep* (*del k x sub*) *sep*
using *2.IH*(*2*)[*OF list-split*[*symmetric*] *Cons h-split*[*symmetric*], *of lsep sep*]
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*5*) *aligned-sorted-separators bal.simps*(*2*) *bal-split-left*(*1*) *h-split l-split list-split local.Cons node-sorted-split*(*1*) *node-sorted-split*(*3*) *order-impl-root-order root-order.simps*(*2*) *some-child-sub*(*1*) *sorted-cons sorted-leaves-induct-subtree sorted-snoc split-set.split-req*(*2*) *split-set.split-req*(*3*) *split-set-axioms split-set*(*1*))
then have *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))
using ‹*leaves* (*del k x sub*) = *del-list x* (*leaves sub*) ⟹ *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))› **by** *blast*
then have *sorted-less* (*leaves* (*del k x* (*Node ts t*)))
using *2.prems*(*4*) *sorted-del-list* **by** *auto*
then have *sorted-leaves*: *sorted-less* (*leaves* (*Node* (*ls*@(*del k x sub*, *sep*)#*rs*) *t*))

```
        using list-split Cons h-split
        using rebalance-middle-tree-inorder[of t del k x sub rs k ls sep]
        using 2.prems(4) 2.prems(5) IH ‹leaves (del k x (Node ts t)) = del-list x
(leaves (Node ts t))› del-list-split-aligned del-list-split-right-aligned
        by auto
      from IH have aligned l (del k x (Node ts t)) u
      proof −
        have aligned l (Node (ls@(del k x sub, sep)#rs) t) u
          using 2.prems(5) IH h-split list-conc local.Cons l-split
          using aligned-subst by fastforce
        then have aligned l (rebalance-middle-tree k ls (del k x sub) sep rs t) u
          using rebalance-middle-tree-aligned sorted-leaves
            by (smt (verit, best) 2.prems(1) 2.prems(2) 2.prems(3) bal.simps(2)
bal-sub-height del-height h-split list-split local.Cons node-sorted-split(3) order-impl-root-order
root-order.simps(2) some-child-sub(1) split-set(1))
        then show ?thesis
          using list-split Cons h-split
          by auto
      qed
      then show ?thesis
        using ‹leaves (del k x (Node ts t)) = del-list x (leaves (Node ts t))› by blast
    qed
  qed
qed

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 ≤ u
  shows leaves (del k x t) = del-list x (leaves t) ∧ Laligned (del k x t) u
  using assms
proof (induction k x t arbitrary: u rule: del.induct)
  case (1 k x xs)
  then have leaves (del k x (Leaf xs)) = del-list x (leaves (Leaf xs))
    by (simp add: insert-list-req)
  moreover have Laligned (del k x (Leaf xs)) u
  proof −
    have ∀ x ∈ set xs − {x}. x ≤ u
      using 1.prems(5) by auto
    then show ?thesis
      using set-del-list insert-list-req
      by (metis 1(4) Laligned.simps(1) del.simps(1) leaves.simps(1))
  qed
  ultimately show ?case
    by simp
next
```

case (*2 k x ts t u*)
then obtain *ls rs* **where** *list-split*: *split ts x* = (*ls*, *rs*)
by (*meson surj-pair*)
then have *list-conc*: *ts* = *ls* @ *rs*
using *split-set.split-conc split-set-axioms* **by** *blast*
show *?case*
proof (*cases rs*)
case *Nil*
then obtain *ls′ lsub lsep* **where** *ls-split*: *ls* = *ls′* @ [(*lsub*,*lsep*)]
by (*metis 2.prems*(*2*) *append-Nil2 list.size*(*3*) *list-conc old.prod.exhaust root-order.simps*(*2*) *snoc-eq-iff-butlast zero-less-iff-neq-zero*)
then have *IH*: *leaves* (*del k x t*) = *del-list x* (*leaves t*) ∧ *aligned lsep* (*del k x t*) *u*
by (*metis* (*no-types*, *lifting*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *2.prems*(*4*) *2.prems*(*5*) *2.prems*(*6*) *Lalign-last Laligned-sorted-separators bal.simps*(*2*) *del-inorder list-conc list-split local.Nil order-impl-root-order root-order.simps*(*2*) *self-append-conv sorted-leaves-induct-last sorted-snoc split-set.split-req*(*2*) *split-set-axioms*)
have *leaves* (*del k x* (*Node ts t*)) = *leaves* (*rebalance-last-tree k ts* (*del k x t*))
using *list-split Nil list-conc* **by** *auto*
also have ... = *leaves-list ts* @ *leaves* (*del k x t*)
proof −
obtain *ts′ sub sep* **where** *ts-split*: *ts* = *ts′* @ [(*sub*, *sep*)]
using ‹*ls* = *ls′* @ [(*lsub*, *lsep*)]› *list-conc local.Nil* **by** *blast*
then have *height sub* = *height t*
using *2.prems*(*3*) **by** *auto*
moreover have *height t* = *height* (*del k x t*)
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *bal.simps*(*2*) *del-height order-impl-root-order root-order.simps*(*2*))
ultimately show *?thesis*
using *rebalance-last-tree-inorder*
using *ts-split* **by** *auto*
qed
also have ... = *leaves-list ts* @ *del-list x* (*leaves t*)
using *IH* **by** *blast*
also have ... = *del-list x* (*leaves* (*Node ts t*))
by (*metis 2.prems*(*4*) *2.prems*(*5*) *append-self-conv2 concat.simps*(*1*) *list.simps*(*8*) *list-conc list-split local.Nil self-append-conv split-set.del-list-split split-set-axioms*)
finally have *0*: *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*)) **.**
moreover have *Laligned* (*del k x* (*Node ts t*)) *u*
proof −
have *Laligned* (*Node ls* (*del k x t*)) *u*
using *IH list-conc Nil 2.prems ls-split*
by (*metis Laligned-subst-last self-append-conv*)
moreover have *sorted-less* (*leaves* (*Node ls* (*del k x t*)))
using *2.prems*(*4*) ‹*leaves-list ts* @ *del-list x* (*leaves t*) = *del-list x* (*leaves* (*Node ts t*))› ‹*leaves-list ts* @ *leaves* (*del k x t*) = *leaves-list ts* @ *del-list x* (*leaves t*)› *list-conc local.Nil sorted-del-list*
by *auto*
ultimately have *Laligned* (*rebalance-last-tree k ls* (*del k x t*)) *u*

using *rebalance-last-tree-Laligned*
by (*metis* (*no-types*, *lifting*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *UnCI bal.simps*(*2*) *del-height list.set-intros*(*1*) *list-conc ls-split order-impl-root-order root-order.simps*(*2*) *set-append some-child-sub*(*1*))
then show *?thesis* **using** *list-split ls-split 2.prems Nil*
by *simp*
qed
ultimately show *?thesis*
by *simp*
next
case (*Cons h rs*)
then obtain *sub sep* **where** *h-split*: *h* = (*sub*,*sep*)
by (*cases h*)
then have *node-sorted-split*:
sorted-less (*leaves* (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*))
root-order k (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
bal (*Node* (*ls*@(*sub*,*sep*)#*rs*) *t*)
using *2.prems h-split list-conc Cons* **by** *blast*+
{
assume *IH*: *leaves* (*del k x sub*) = *del-list x* (*leaves sub*)
have *leaves* (*del k x* (*Node ts t*)) = *leaves* (*rebalance-middle-tree k ls* (*del k x sub*) *sep rs t*)
using *Cons list-split h-split 2.prems*
by *auto*
also have ... = *leaves* (*Node* (*ls*@(*del k x sub*, *sep*)#*rs*) *t*)
using *rebalance-middle-tree-inorder*[*of t del k x sub rs*]
by (*smt* (*verit*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *bal.simps*(*2*) *bal-sub-height del-height h-split list-split local.Cons node-sorted-split*(*3*) *order-impl-root-order rebalance-middle-tree-inorder root-order.simps*(*2*) *some-child-sub*(*1*) *split-set*(*1*))
also have ... = *leaves-list ls* @ *leaves* (*del k x sub*) @ *leaves-list rs* @ *leaves t*
by *auto*
also have ... = *leaves-list ls* @ *del-list x* (*leaves sub* @ *leaves-list rs* @ *leaves t*)
using *del-list-split-right*[*of ts t u x ls sub sep rs*]
using *list-split Cons 2.prems*(*4*,*5*) *h-split IH list-conc*
by *auto*
also have ... = *del-list x* (*leaves-list ls* @ *leaves sub* @ *leaves-list rs* @ *leaves t*)
using *del-list-split*[*of ts t u x ls* (*sub*,*sep*)#*rs*]
using *list-split Cons 2.prems*(*4*,*5*) *h-split IH list-conc*
by *auto*
finally have *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))
using *list-conc Cons h-split*
by *auto*
}
then show *?thesis*
proof (*cases ls*)
case *Nil*
then have *IH*: *leaves* (*del k x sub*) = *del-list x* (*leaves sub*) ∧ *Laligned* (*del k*

x sub) *sep*
by (*smt* (*verit*, *ccfv-threshold*) *2.IH*(*2*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*5*) *Laligned-nonempty-Node Laligned-sorted-separators append-self-conv2 bal.simps*(*2*) *h-split list.set-intros*(*1*) *list-conc list-split local.Cons node-sorted-split*(*1*) *node-sorted-split*(*3*) *order-impl-root-order root-order.simps*(*2*) *some-child-sub*(*1*) *sorted-leaves-induct-subtree sorted-wrt-append split-set.split-req*(*3*) *split-set-axioms*)
then have *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))
using ‹*leaves* (*del k x sub*) = *del-list x* (*leaves sub*) ⟹ *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))› **by** *blast*
then have *sorted-less* (*leaves* (*del k x* (*Node ts t*)))
using *2.prems*(*4*) *sorted-del-list* **by** *auto*
then have *sorted-leaves*: *sorted-less* (*leaves* (*Node* (*ls*@(*del k x sub*, *sep*)#*rs*) *t*))
using *list-split Cons h-split*
using *rebalance-middle-tree-inorder*[*of t del k x sub rs k ls sep*]
using *2.prems*(*4*) *2.prems*(*5*) *IH* ‹*leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))› *del-list-split del-list-split-right*
by *auto*
from *IH* **have** *Laligned* (*del k x* (*Node ts t*)) *u*
proof −
have *Laligned* (*Node* (*ls*@(*del k x sub*, *sep*)#*rs*) *t*) *u*
using *2.prems*(*5*) *IH h-split list-conc local.Cons local.Nil* **by** *auto*
then have *Laligned* (*rebalance-middle-tree k ls* (*del k x sub*) *sep rs t*) *u*
using *rebalance-middle-tree-Laligned sorted-leaves*
by (*smt* (*verit*, *best*) *2.prems*(*1*) *2.prems*(*2*) *2.prems*(*3*) *append-self-conv2 bal.simps*(*2*) *bal-sub-height del-height h-split list.set-intros*(*1*) *list-conc local.Cons local.Nil order-impl-root-order root-order.simps*(*2*) *some-child-sub*(*1*))
then show *?thesis*
using *list-split Cons h-split*
by *auto*
qed
then show *?thesis*
using ‹*leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))› **by** *blast*
next
case -: (*Cons a list*)
then obtain *ls′ lsub lsep* **where** *l-split*: *ls* = *ls′*@[(*lsub*,*lsep*)]
by (*metis list.discI old.prod.exhaust snoc-eq-iff-butlast*)
then have *aligned lsep sub sep*
using *2.prems*(*5*) *Lalign-last Laligned-split-left h-split list-conc local.Cons* **by** *blast*
then have *IH*: *leaves* (*del k x sub*) = *del-list x* (*leaves sub*) ∧ *aligned lsep* (*del k x sub*) *sep*
by (*metis 2.prems*(*1*) *2.prems*(*2*) *2.prems*(*5*) *Laligned-sorted-separators bal.simps*(*2*) *bal-split-left*(*1*) *del-inorder h-split l-split list-split local.Cons node-sorted-split*(*1*) *node-sorted-split*(*3*) *order-impl-root-order root-order.simps*(*2*) *some-child-sub*(*1*) *sorted-leaves-induct-subtree sorted-snoc split-set.split-req*(*2*) *split-set.split-req*(*3*) *split-set-axioms split-set*(*1*))
then have *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))
using ‹*leaves* (*del k x sub*) = *del-list x* (*leaves sub*) ⟹ *leaves* (*del k x* (*Node ts t*)) = *del-list x* (*leaves* (*Node ts t*))› **by** *blast*

```
      then have sorted-less (leaves (del k x (Node ts t)))
        using 2.prems(4) sorted-del-list by auto
      then have sorted-leaves: sorted-less (leaves (Node (ls@(del k x sub, sep)#rs)
t))
        using list-split Cons h-split
        using rebalance-middle-tree-inorder[of t del k x sub rs k ls sep]
        using 2.prems(4) 2.prems(5) IH ‹leaves (del k x (Node ts t)) = del-list x
(leaves (Node ts t))› del-list-split del-list-split-right
        by auto
      from IH have Laligned (del k x (Node ts t)) u
      proof −
        have Laligned (Node (ls@(del k x sub, sep)#rs) t) u
          using 2.prems(5) IH h-split list-conc local.Cons l-split
          using Laligned-subst by fastforce
        then have Laligned (rebalance-middle-tree k ls (del k x sub) sep rs t) u
          using rebalance-middle-tree-Laligned sorted-leaves
            by (smt (verit, best) 2.prems(1) 2.prems(2) 2.prems(3) bal.simps(2)
bal-sub-height del-height h-split list-split local.Cons node-sorted-split(3) order-impl-root-order
root-order.simps(2) some-child-sub(1) split-set(1))
        then show ?thesis
          using list-split Cons h-split
          by auto
      qed
      then show ?thesis
        using ‹leaves (del k x (Node ts t)) = del-list x (leaves (Node ts t))› by blast
    qed
  qed
qed

lemma reduce-root-order: ⟦k > 0; almost-order k t⟧ ⟹ root-order k (reduce-root
t)
  apply(cases t)
   apply(auto split!: list.splits simp add: order-impl-root-order)
  done

lemma reduce-root-bal: bal (reduce-root t) = bal t
  apply(cases t)
   apply(auto split!: list.splits)
  done


lemma reduce-root-inorder: leaves (reduce-root t) = leaves t
  apply (cases t)
   apply (auto split!: list.splits)
  done

lemma reduce-root-Laligned: Laligned (reduce-root t) u = Laligned t u
  apply(cases t)
  apply (auto split!: list.splits)
```

done

lemma *delete-order*: $[\![k > 0;\ bal\ t;\ root\text{-}order\ k\ t;\ sorted\text{-}less\ (leaves\ t)]\!] \Longrightarrow$
root-order k (*delete k x t*)
using *del-order*
by (*simp add*: *reduce-root-order*)

lemma *delete-bal*: $[\![k > 0;\ bal\ t;\ root\text{-}order\ k\ t]\!] \Longrightarrow bal\ (delete\ k\ x\ t)$
using *del-bal*
by (*simp add*: *reduce-root-bal*)

lemma *delete-Linorder*:
assumes $k > 0$ *root-order k t sorted-less* (*leaves t*) *Laligned t u bal t* $x \leq u$
shows *leaves* (*delete k x t*) = *del-list x* (*leaves t*)
and *Laligned* (*delete k x t*) *u*
using *reduce-root-Laligned*[*of del k x t u*] *reduce-root-inorder*[*of del k x t*]
using *del-Linorder*[*of k t u x*]
using *assms*
by *simp-all*

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*
using *assms delete-Linorder top-greatest*
by *simp-all*

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 (*standard*, *goal-cases*)
case (*2 s x*)
then show *?case*
using *isin-set-Linorder-top*
by *simp*
next

```
  case (3 s x)
  then show ?case
    using insert-Linorder-top
    by simp
next
  case (4 s x)
  then show ?case using delete-Linorder-top
    by auto
next
  case (6 s x)
  then show ?case using insert-order insert-bal insert-Linorder-top
    by auto
next
  case (7 s x)
  then show ?case using delete-order delete-bal delete-Linorder-top
    by auto
qed (simp add: empty-bplustree-def)+



declare node_i.simps[simp del]

end


lemma sorted-ConsD: sorted-less (y # xs) ⟹ x ≤ y ⟹ x ∉ set xs
  by (auto simp: sorted-Cons-iff)

lemma sorted-snocD: sorted-less (xs @ [y]) ⟹ y ≤ x ⟹ x ∉ set xs
  by (auto simp: sorted-snoc-iff)


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))
  by (auto simp: isin-simps2)

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 (cases ls)
  case Nil
  then have xs = rs
    using assms by (auto dest!: split-list-conc)
  then show ?thesis by simp
next
  case Cons
  then obtain ls' sep where ls-tail-split: ls = ls' @ [sep]
    by (metis list.simps(3) rev-exhaust)
  then have x-sm-sep: sep < x
    using split-list-req(2)[of xs x ls' sep rs]
    using assms sorted-cons sorted-snoc
    by blast
  moreover have xs = ls@rs
    using assms split-list-conc by simp
  ultimately show ?thesis
    using isin-sorted[of ls' sep rs]
    using assms ls-tail-split
    by auto
qed

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 (cases rs)
  case Nil
  then show ?thesis
```

by *simp*
next
case (*Cons sep′ rs*)
from *assms* **have** $x < sep'$
by (*metis le-less less-trans list.set-intros*(*1*) *local.Cons sorted-Cons-iff sorted-wrt-append split-list-conc split-list-sorted*(*2*))
moreover have $ts = ls@sep\#sep'\#rs$
using *split-list-conc*[*OF assms*(*1*)] *Cons* **by** *auto*
moreover have *sorted-less* (*sep*#*sep′*#*rs*)
using *Cons assms calculation*(*2*) *sorted-wrt-append* **by** *blast*
ultimately show *?thesis*
using *isin-sorted*[*of* [*sep*] *sep′ rs x*] *Cons*
by *simp*
qed

theorem *isin-list-set*:
assumes *sorted-less xs*
shows *isin-list x xs* = (*x* ∈ *set xs*)
using *assms*
using *isin-sorted-split-list*[*of xs x*]
using *isin-sorted-split-list-right*[*of xs x*]
by (*auto split*!: *list.splits*)

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 (*cases ls*)
case *Nil*
then have *xs* = *rs*
using *assms* **by** (*auto dest*!: *split-list-conc*)
then show *?thesis*
using *Nil* **by** *simp*
next
case *Cons*
then obtain *ls′ sep* **where** *ls-tail-split*: *ls* = *ls′* @ [*sep*]
by (*metis list.simps*(*3*) *rev-exhaust*)
then have *x-sm-sep*: *sep* < *x*
using *split-list-req*(*2*)[*of xs x ls′ sep rs*]
using *assms sorted-cons sorted-snoc*
by *blast*
moreover have *xs* = *ls*@*rs*
using *assms split-list-conc* **by** *simp*
ultimately show *?thesis*
using *ins-list-sorted*[*of ls′ sep x rs*]
using *assms ls-tail-split sorted-wrt-append*[*of* (<) *ls rs*]
by *auto*
qed

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 −
have *x* < *sep*
by (*meson assms*(*1*) *assms*(*2*) *assms*(*3*) *le-neq-trans split-list-sorted*(*2*))
then show *?thesis*
using *ins-list-sorted*[*of* [] *sep*]
using *assms*
by *auto*
qed

theorem *insert-list-set*:
assumes *sorted-less xs*
shows *insert-list x xs* = *ins-list x xs*
using *assms split-list-conc*
using *insert-sorted-split-list*[*of xs x*]
using *insert-sorted-split-list-right*[*of xs x*]
by (*auto split*!: *list.splits prod.splits*)

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 (*cases ls*)
case *Nil*
then have *xs* = *rs*
using *assms* **by** (*auto dest*!: *split-list-conc*)
then show *?thesis*
using *Nil* **by** *simp*
next
case *Cons*
then obtain *ls*′ *sep* **where** *ls-tail-split*: *ls* = *ls*′ @ [*sep*]
by (*metis list.simps*(*3*) *rev-exhaust*)
then have *x-sm-sep*: *sep* < *x*
using *split-list-req*(*2*)[*of xs x ls*′ *sep rs*]
using *assms sorted-cons sorted-snoc*
by *blast*
moreover have *xs* = *ls*@*rs*
using *assms split-list-conc* **by** *simp*
ultimately show *?thesis*
using *del-list-sorted*[*of ls*′ *sep rs*]
using *assms ls-tail-split sorted-wrt-append*[*of* (<) *ls rs*]
by *auto*
qed

lemma *delete-sorted-split-list-right*:
  assumes *split-list ts x* = (*ls*, *sep*#*rs*)
    and *sorted-less ts*
    and *x* ≠ *sep*
  shows *del-list x* (*sep*#*rs*) = *sep*#*rs*
proof −
  have *sorted-less* (*sep*#*rs*)
   by (*metis assms*(*1*) *assms*(*2*) *sorted-wrt-append split-list.split-list-conc split-list-axioms*)
  moreover have *x* < *sep*
    by (*meson assms*(*1*) *assms*(*2*) *assms*(*3*) *le-neq-trans split-list-sorted*(*2*))
  ultimately show *?thesis*
    using *del-list-sorted*[*of* [] *sep rs x*]
    by *simp*
qed

theorem *delete-list-set*:
  assumes *sorted-less xs*
  shows *delete-list x xs* = *del-list x xs*
  using *assms split-list-conc*[*of xs x*]
  using *delete-sorted-split-list*[*of xs x*]
  using *delete-sorted-split-list-right*[*of xs x*]
  by (*auto split*!: *list.splits prod.splits*)

end

context *split-full*
begin

sublocale *split-set split isin-list insert-list delete-list*
  using *isin-list-set insert-list-set delete-list-set*
  by *unfold-locales auto*

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* ∈ *X*. *x* ≥ *l*}

definition *lrange-filter l = filter* (λ*x*. *x* ≥ *l*)

lemma *lrange-filter-iff-Lrange*: *set* (*lrange-filter l xs*) = *Lrange l* (*set xs*)
by (*auto simp add*: *lrange-filter-def Lrange-def*)

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

lemma *sorted-leq-lrange*: *sorted-wrt* (≤) *xs* ⟹ *lrange-list* (*l*::'*a*::*linorder*) *xs* = *lrange-filter l xs*
apply(*induction xs*)
apply(*auto simp add*: *lrange-filter-def*)
by (*metis dual-order.trans filter-True*)

lemma *sorted-less-lrange*: *sorted-less xs* ⟹ *lrange-list* (*l*::'*a*::*linorder*) *xs* = *lrange-filter l xs*
by (*simp add*: *sorted-leq-lrange strict-sorted-iff*)

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*))
by (*induction xs arbitrary*: *x*) *auto*

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*))
by (*metis lrange-list-sorted sorted-less-lrange sorted-wrt-append*)

lemma *lrange-suffix*: *suffix* (*lrange-list l xs*) *xs*
apply(*induction xs*)
apply (*auto dest*: *suffix-ConsI*)
done

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 (cases ls)
  case Nil
  then have ts = rs
    using assms by (auto dest!: split-conc)
  then show ?thesis by simp
next
  case Cons
  then obtain ls' sub sep where ls-tail-split: ls = ls' @ [(sub,sep)]
    by (metis list.simps(3) rev-exhaust surj-pair)
  then have x-sm-sep: sep < x
    using split-req(2)[of ts x ls' sub sep rs]
    using Laligned-sorted-separators[OF assms(1)]
    using assms sorted-cons sorted-snoc
    by blast
  moreover have leaves-split: leaves (Node ts t) = leaves-list ls @ leaves-list rs @
leaves t
    using assms(3) leaves-split by blast
  then show ?thesis
  proof (cases leaves-list ls)
    case Nil
    then show ?thesis
      using leaves-split
      by (metis self-append-conv2)
  next
    case Cons
    then obtain leavesls' l' where leaves-tail-split: leaves-list ls = leavesls' @ [l']
      by (metis list.simps(3) rev-exhaust)
    then have l' ≤ sep
    proof −
      have l' ∈ set (leaves-list ls)
        using leaves-tail-split by force
      then have l' ∈ set (leaves (Node ls' sub))
        using ls-tail-split
        by auto
      moreover have Laligned (Node ls' sub) sep
```

```
      using assms split-conc[OF assms(3)] Cons ls-tail-split
      using Laligned-split-left[of ls' sub sep rs t u]
      by simp
    ultimately show ?thesis
      using Laligned-leaves-inbetween[of Node ls' sub sep]
      by blast
  qed
  then have l' < x
    using le-less-trans x-sm-sep by blast
  then show ?thesis
    using assms(2) ls-tail-split leaves-tail-split leaves-split x-sm-sep
    using lrange-filter-sorted[of leavesls' l' leaves-list rs @ leaves t x]
    by (auto simp add: lrange-filter-def)
 qed
qed


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 −
 from assms have x ≤ sep
 proof −
   from assms have sorted-less (separators ts)
   by (meson Laligned-sorted-inorder sorted-cons sorted-inorder-separators sorted-snoc)
   then show ?thesis
     using split-req(3)
     using assms
     by fastforce
 qed
 moreover have leaves-split: leaves (Node ts t) = leaves-list ls @ leaves sub @
leaves-list rs @ leaves t
   using split-conc[OF assms(1)] by auto
 ultimately show ?thesis
 proof (cases leaves-list rs @ leaves t)
   case Nil
   then show ?thesis
   by (metis assms(1) leaves-split same-append-eq self-append-conv split-tree.leaves-split
split-tree-axioms)
 next
   case (Cons r' rs')
   then have sep < r'
      by (metis (mono-tags, lifting) Laligned-split-right aligned-leaves-inbetween
append.right-neutral append-assoc assms(1) assms(3) concat.simps(1) leaves-conc
list.set-intros(1) list.simps(8) split-tree.split-conc split-tree-axioms)
   then have  x < r'
```

```
    using ‹x ≤ sep› by auto
  moreover have sorted-less (leaves-list ((sub,sep)#rs) @ leaves t)
    using assms sorted-wrt-append split-conc
    by fastforce
  ultimately show ?thesis
    using lrange-filter-sorted[of leaves sub r' rs' x] Cons
    by auto
 qed
qed


theorem lrange-set:
 assumes sorted-less (leaves t)
  and aligned l t u
 shows lrange t x = lrange-filter x (leaves t)
 using assms
proof(induction t x arbitrary: l u rule: lrange.induct)
 case (1 ks x)
 then show ?case
  using lrange-list-req
  by auto
next
 case (2 ts t x)
 then obtain ls rs where list-split: split ts x = (ls, rs)
  by (meson surj-pair)
 then have list-conc: ts = ls @ rs
  using split-conc by auto
 show ?case
 proof (cases rs)
  case Nil
  then have lrange (Node ts t) x = lrange t x
   by (simp add: list-split)
  also have ... = lrange-filter x (leaves t)
   by (metis 2.IH(1) 2.prems(1) 2.prems(2) align-last' list-split local.Nil sorted-leaves-induct-last)
  also have ... = lrange-filter x (leaves (Node ts t))
   by (metis 2.prems(1) 2.prems(2) aligned-imp-Laligned leaves.simps(2) list-conc
list-split local.Nil lrange-sorted-split same-append-eq self-append-conv split-tree.leaves-split
split-tree-axioms)
  finally show ?thesis .
 next
  case (Cons a list)
  then obtain sub sep where a-split: a = (sub,sep)
   by (cases a)
   then have lrange (Node ts t) x = lrange sub x @ leaves-list list @ leaves t
    using list-split Cons a-split
    by auto
   also have ... = lrange-filter x (leaves sub) @ leaves-list list @ leaves t
    using 2.IH(2)[of ls rs (sub,sep) list sub sep]
   using 2.prems a-split list-conc list-split local.Cons sorted-leaves-induct-subtree
```

```
          align-sub
      by (metis in-set-conv-decomp)
    also have ... = lrange-filter x (leaves (Node ts t))
        by (metis 2.prems(1) 2.prems(2) a-split aligned-imp-Laligned list-split
local.Cons lrange-sorted-split lrange-sorted-split-right)
    finally show ?thesis .
  qed
qed
```

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
  apply(induction ts)
  subgoal by auto
  subgoal using concat-leaf-nodes-leaves by auto
  done

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)))
  using assms
proof(induction t x arbitrary: l u rule: lrange.induct)
  case (1 ks x)
  then show ?case
    apply simp
    by (metis lrange-suffix sorted-less-lrange)
next
  case (2 ts t x)
  then obtain ls rs where list-split: split ts x = (ls, rs)
    by (meson surj-pair)
  then have list-conc: ts = ls @ rs
    using split-conc by auto
  show ?case
  proof (cases rs)
    case Nil
    then have ∗: leaf-nodes-lrange (Node ts t) x = leaf-nodes-lrange t x
```

by (*simp add*: *list-split*)
moreover have *suffix* (*lrange-filter x* (*leaves t*)) (*concat* (*map leaves* (*leaf-nodes-lrange t x*)))
by (*metis 2.IH*(*1*) *2.prems*(*1*) *2.prems*(*2*) *align-last'* *list-split local.Nil sorted-leaves-induct-last*)
then have *suffix* (*lrange-filter x* (*leaves* (*Node ts t*))) (*concat* (*map leaves* (*leaf-nodes-lrange t x*)))
by (*metis 2.prems*(*1*) *2.prems*(*2*) *aligned-imp-Laligned leaves.simps*(*2*) *list-conc list-split local.Nil lrange-sorted-split same-append-eq self-append-conv split-tree.leaves-split split-tree-axioms*)
ultimately show *?thesis* **by** *simp*
next
case (*Cons a list*)
then obtain *sub sep* **where** *a-split*: *a* = (*sub*,*sep*)
by (*cases a*)
then have *leaf-nodes-lrange* (*Node ts t*) *x* = *leaf-nodes-lrange sub x* @ *leaf-nodes-list list* @ *leaf-nodes t*
using *list-split Cons a-split*
by *auto*
moreover have ∗: *suffix* (*lrange-filter x* (*leaves sub*)) (*concat* (*map leaves* (*leaf-nodes-lrange sub x*)))
by (*metis 2.IH*(*2*) *2.prems*(*1*) *2.prems*(*2*) *a-split align-sub in-set-conv-decomp list-conc list-split local.Cons sorted-leaves-induct-subtree*)
then have *suffix* (*lrange-filter x* (*leaves* (*Node ts t*))) (*concat* (*map leaves* (*leaf-nodes-lrange sub x* @ *leaf-nodes-list list* @ *leaf-nodes t*)))
proof (*goal-cases*)
case *1*
have *lrange-filter x* (*leaves* (*Node ts t*)) = *lrange-filter x* (*leaves sub* @ *leaves-list list* @ *leaves t*)
by (*metis* (*no-types*, *lifting*) *2.prems*(*1*) *2.prems*(*2*) *a-split aligned-imp-Laligned append.assoc concat-map-maps fst-conv list.simps*(*9*) *list-split local.Cons lrange-sorted-split maps-simps*(*1*))
also have ... = *lrange-filter x* (*leaves sub*) @ *leaves-list list* @ *leaves t*
by (*metis 2.prems*(*1*) *2.prems*(*2*) *a-split aligned-imp-Laligned calculation list-split local.Cons lrange-sorted-split-right split-range.lrange-sorted-split split-range-axioms*)
moreover have (*concat* (*map leaves* (*leaf-nodes-lrange sub x* @ *leaf-nodes-list list* @ *leaf-nodes t*))) = (*concat* (*map leaves* (*leaf-nodes-lrange sub x*)) @ *leaves-list list* @ *leaves t*)
using *concat-leaf-nodes-leaves-list*[*of list*] *concat-leaf-nodes-leaves*[*of t*]
by *simp*
ultimately show *?case*
using ∗
by *simp*
qed
ultimately show *?thesis* **by** *simp*
qed
qed

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

```
  apply(induction t x rule: leaf-nodes-lrange.induct)
  apply (auto split!: prod.splits list.splits)
  by (metis Cons-eq-appendI fst-conv in-set-conv-decomp split-conc)
```

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(induction t x arbitrary: ks list rule: leaf-nodes-lrange.induct)
  case (1 ks x)
  then show ?case by simp
next
  case (2 ts t x ks list)
  then show ?case
  proof(cases split ts x)
    case split: (Pair ls rs)
    then show ?thesis
    proof (cases rs)
      case Nil
      then show ?thesis
        using 2.IH(1) 2.prems split by auto
    next
      case (Cons subsep rss)
      then show ?thesis
      proof(cases subsep)
        case sub-sep: (Pair sub sep)
        thm 2.IH(2) 2.prems
        have ∃list'. leaf-nodes-lrange sub x = (Leaf ks)#list'
          using 2.prems split Cons sub-sep leaf-nodes-lrange-not-empty[of sub x]
            apply simp
          by fastforce
        then obtain list' where ∗: leaf-nodes-lrange sub x = (Leaf ks)#list'
          by blast
       moreover have list = list'@concat (map (leaf-nodes ∘ fst) rss) @ leaf-nodes
t
          using ∗
          using 2.prems split Cons sub-sep
          by simp
        ultimately show ?thesis
       using split 2.IH(2)[OF split[symmetric] Cons sub-sep[symmetric] ∗,symmetric]
           Cons sub-sep concat-leaf-nodes-leaves-list[of rss] concat-leaf-nodes-leaves[of
t]
          by simp
      qed
    qed
  qed
qed
```

We finally obtain a function that is way easier to reason about in the im-

perative 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 −
obtain *ks list* **where** ∗: *leaf-nodes-lrange t x* = (*Leaf ks*)#*list*
using *leaf-nodes-lrange-not-empty* **by** *blast*
then have *concat-leaf-nodes-lrange t x* = *lrange-list x ks* @ (*concat* (*map leaves list*))
by *simp*
also have . . . = *lrange t x*
using *leaf-nodes-lrange-pre-lrange*[*OF* ∗]
by *simp*
finally show *?thesis* **.**
qed

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(*cases rs*)
case *rs-Nil*: *Nil*
then show *?thesis*
proof(*cases ls*)
case *Nil*
then show *?thesis*
using *assms split-list-req*(*1*)[*of xs l ls rs*] *rs-Nil*
by *simp*
next
case *Cons*
then obtain *lss sep* **where** *snoc*: *ls* = *lss*@[*sep*]
by (*metis append-butlast-last-id list.simps*(*3*))
then have *sep* < *l*
using *assms*(*1*) *assms*(*2*) *split-list-req*(*2*) **by** *blast*
then show *?thesis*
using *lrange-list-sorted*[*of lss sep rs l*]
snoc split-list-req(*1*)[*OF assms*(*2*)]
assms rs-Nil

```
    by simp
  qed
next
  case ls-Cons: (Cons sep rss)
  then have *: l ≤ sep
    using assms(1) assms(2) split-list-req(3) by auto
  then show ?thesis
  proof(cases ls)
    case Nil
    then show ?thesis
    using lrange-list-sorted[of ls sep rss l]
          split-list-req(1)[OF assms(2)] assms
          ls-Cons *
    by simp
  next
    case Cons
    then obtain lss sep2 where snoc: ls = lss@[sep2]
      by (metis append-butlast-last-id list.simps(3))
    then have sep2 < l
      using assms(1) assms(2) split-list-req(2) by blast
    moreover have sorted-less (lss@[sep2])
    using assms(1) assms(2) ls-Cons snoc sorted-mid-iff sorted-snoc split-list-req(1)
by blast
    ultimately show ?thesis
      using lrange-list-sorted[of ls sep rss l]
            lrange-list-sorted[of lss sep2 [] l]
            split-list-req(1)[OF assms(2)] assms
            ls-Cons * snoc
      by simp
  qed
qed

lemma lrange-split-req:
  assumes sorted-less xs
  shows lrange-split l xs = lrange-filter l xs
  unfolding lrange-split-def
  using lrange-filter-split[of xs l] assms
  using sorted-less-lrange
  by (simp split!: prod.splits)

end

context split-full
begin

sublocale split-range split lrange-split
  using lrange-split-req
  by unfold-locales auto
```

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*
apply *unfold-locales*
unfolding *linear-split.simps*
apply (*auto split*: *list.splits*)
subgoal
by (*metis* (*no-types*, *lifting*) *case-prodD in-set-conv-decomp takeWhile-eq-all-conv takeWhile-idem*)
subgoal
by (*metis case-prod-conv hd-dropWhile le-less-linear list.sel*(*1*) *list.simps*(*3*))
done

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*
apply *unfold-locales*
unfolding *linear-split.simps*
subgoal by (*auto split*: *list.splits*)
subgoal
apply (*auto split*: *list.splits*)
by (*metis* (*no-types*, *lifting*) *case-prodD in-set-conv-decomp takeWhile-eq-all-conv takeWhile-idem*)
subgoal by (*metis case-prod-conv hd-dropWhile le-less-linear list.sel*(*1*) *list.simps*(*3*))
done

lemma [*code*]: *bplustree-ls-isin* (*Leaf ks*) *x* = *bplustree-ls-isin-list x ks*
by (*simp add*: *bplustree-ls-isin-list-def*)
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*)
by (*simp add*: *bplustree-ls-insert-list-def*)
declare *bplustree-linear-search.ins.simps*(*2*)[*code*]

lemma [*code*]: *bplustree-ls-del k x* (*Leaf ks*) =
Leaf (*bplustree-ls-delete-list x ks*)
by (*simp add*: *bplustree-ls-delete-list-def*)
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*}
by *auto*

setup-lifting *type-definition-uint8*

instantiation *uint8* :: *linorder*
begin

lift-definition *less-eq-uint8* :: *uint8* ⇒ *uint8* ⇒ *bool*
is (*less-eq*::*nat* ⇒ *nat* ⇒ *bool*) **.**

lift-definition *less-uint8* :: *uint8* ⇒ *uint8* ⇒ *bool*
is (*less*::*nat* ⇒ *nat* ⇒ *bool*) **.**

instance
by *standard* (*transfer*; *auto*)+
end

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

lift-definition *top-uint8* :: *uint8* **is** *uint8-max*::*nat*
by *simp*

instance
by *standard* (*transfer*; *simp*)
end

instantiation *uint8* :: *numeral*
begin

lift-definition *one-uint8* :: *uint8* **is** *1*::*nat*
by *auto*

lift-definition *plus-uint8* :: *uint8* ⇒ *uint8* ⇒ *uint8*
is λ*a b*. *min* (*a* + *b*) *uint8-max*
by *simp*

instance by *standard* (*transfer*; *auto*)
end

instantiation *uint8* :: *equal*
begin

lift-definition *equal-uint8* :: *uint8* ⇒ *uint8* ⇒ *bool*
is (=) **.**

instance by *standard* (*transfer*; *auto*)
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.