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 and deletion 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. We examine the runtime of all operations indirectly by reproducing results of the logarithmic relationship between height and the number of nodes. The results are discussed in greater detail in the related Bachelor's Thesis [3].

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*) $\Longrightarrow$ ($\forall$ *s1* $\in$ *set* (*subtrees ts*). $\forall$ *s2* $\in$ *set* (*subtrees ts*). *height s1* = *height s2*)
by (*metis BTree.bal.simps*(*2*))

lemma *fold-max-set*: $\forall$ *x* $\in$ *set t*. *x* = *f* $\Longrightarrow$ *fold max t f* = *f*
apply(*induction t*)
apply(*auto simp add*: *max-def-raw*)
done

lemma *height-bal-tree*: *bal* (*Node ts t*) $\Longrightarrow$ *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*⟧ $\Longrightarrow$ *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*⟧ $\Longrightarrow$ *bal* (*Node* (*ls*@(*c*,*b*)#*rs*) *t*)
using *bal-substitute*
by *auto*

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

lemma *order-impl-root-order*: $[\![k > 0;\ order\ k\ t]\!] \Longrightarrow root\text{-}order\ k\ t$
apply(*cases t*)
apply(*auto*)
done

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

corollary *sorted-inorder-separators*: *sorted-less* (*inorder* (*Node ts t*)) $\Longrightarrow$ *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*)

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 compow-id-point*)
done

lemma *lower-bound-sharp-node*:
$t = slim\text{-}node\ k\ a\ h \Longrightarrow height\ t = h \wedge order\ k\ t \wedge bal\ t \wedge bound\ k\ h = nodes\ t * k$
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* (λx. *?sub-height*) (*subtrees ts*))
using *map-replicate-const*[*of ?sub-height subtrees ts*] *length-map*
by *simp*
also have ... $\leq$ *sum-list* (*map* (λt. *nodes t* $*$ *k*) (*subtrees ts*))
using *Node*
sum-list-mono[*of subtrees ts* λx. $(k+1)\hat{}(height\ t) - 1$ λx. *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) ≤ (2*k+1)^(height t) − 1*
proof(*cases t*)
case (*Node ts t*)
let *?sub-height = ((2 * k + 1) ^ height t − 1)*
have *sum-list (map nodes (subtrees ts)) * (2*k) =*
*sum-list (map (λt. nodes t * (2 * k)) (subtrees ts))*
using *sum-list-mult-const* **by** *metis*
also have *... ≤ sum-list (map (λx.?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* $\leq$ *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 ≤ 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 ⟹ 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* $\Longrightarrow$ *bal* (*tree*$_i$ *u*)
apply(*cases u*)
apply(*auto*)

done

lemma *tree$_i$-order*: $[\![k > 0;$ *root-order-up$_i$ k u*$]\!] \Longrightarrow$ *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*: $[\![k > 0;$ *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 or-
```

der-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 local.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*) ⟹ *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*⟧ ⟹ ∀ *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 Array-SBlit
  imports Separation-Logic-Imperative-HOL.Array-Blit
begin



code-printing code-module array-blit ⇀ (OCaml)
  ‹
```

```
  let array-blit src si dst di len = (
    if src=dst then
      raise (Invalid-argument array-blit: Same arrays)
    else
      Array.blit src (Z.to-int si) dst (Z.to-int di) (Z.to-int len)
  )
›
```

code-printing constant *blit′* ⇀
(*OCaml*) (*fun* () −> */array′-blit - - - - -*)

export-code *blit* **checking** *OCaml*

5 Same array Blit

The standard framework already provides a function to copy array elements.

term *blit*
thm *blit-rule*
thm *blit.simps*

definition *sblit a s d l* ≡ *blit a s a d l*

When copying values within arrays, blit only works for moving elements to the left.

lemma *sblit-rule*[*sep-heap-rules*]:
  assumes *LEN*:
    si+*len* ≤ *length lsrc*
    and *DST-SM*: *di* ≤ *si*
  shows
    < *src* ↦$_a$ *lsrc* >
    sblit src si di len
    <λ-. *src* ↦$_a$ (*take di lsrc* @ *take len* (*drop si lsrc*) @ *drop* (*di*+*len*) *lsrc*)
    >
  unfolding *sblit-def*
  using *LEN DST-SM*
proof (*induction len arbitrary*: *lsrc si di*)
  case *0* **thus** *?case* **by** *sep-auto*
next
  case (*Suc len*)
  note [*sep-heap-rules*] = *Suc.IH*

  have [*simp*]: ⋀*x*. *lsrc* ! *si* # *take len* (*drop* (*Suc si*) *lsrc*) @ *x*
      = *take* (*Suc len*) (*drop si lsrc*) @ *x*
    apply *simp*
    by (*metis Suc.prems*(*1*) *add-Suc-right Cons-nth-drop-Suc*
      less-Suc-eq-le add.commute not-less-eq take-Suc-Cons
      Nat.trans-le-add2)

```
  from Suc.prems show ?case
    by (sep-auto simp: take-update-last drop-upd-irrelevant)
qed
```

5.1 A reverse blit

The function rblit may be used to copy elements a defined offset to the right

```
primrec rblit :: - array ⇒ nat ⇒ - array ⇒ nat ⇒ nat ⇒ unit Heap where
  rblit - - - - 0 = return ()
| rblit src si dst di (Suc l) = do {
      x ← Array.nth src (si+l);
      Array.upd (di+l) x dst;
      rblit src si dst di l
    }
```

For separated arrays it is equivalent to normal blit. The proof follows similarly to the corresponding proof for blit.

```
lemma rblit-rule[sep-heap-rules]:
  assumes LEN: si+len ≤ length lsrc di+len ≤ length ldst
  shows
    < src ↦a lsrc
      ∗ dst ↦a ldst >
    rblit src si dst di len
    <λ-. src ↦a lsrc
      ∗ dst ↦a (take di ldst @ take len (drop si lsrc) @ drop (di+len) ldst)
    >
  using LEN
proof (induction len arbitrary: ldst)
  case 0 thus ?case by sep-auto
next
  case (Suc len)
  note [sep-heap-rules] = Suc.IH

  have [simp]: drop (di + len) (ldst[di + len := lsrc ! (si + len)])
      = lsrc ! (si + len) # drop (Suc (di + len)) ldst
   by (metis Cons-nth-drop-Suc Suc.prems(2) Suc-le-eq add-Suc-right drop-upd-irrelevant
length-list-update lessI nth-list-update-eq)

  have take len (drop si lsrc) @ [lsrc ! (si + len)] = take (Suc len) (drop si lsrc)
  proof −
    have len < length (drop si lsrc)
      using Suc.prems(1) by force
    then show take len (drop si lsrc) @ [lsrc ! (si + len)] = take (Suc len) (drop
si lsrc)
      by (metis (no-types) Suc.prems(1) add-leD1 nth-drop take-Suc-conv-app-nth)
  qed
  then have [simp]: ⋀x. take len (drop si lsrc) @
```

```
    lsrc ! (si + len) # x = take (Suc len) (drop si lsrc) @ x
    by simp
  from Suc.prems show ?case
    by (sep-auto simp: take-update-last drop-upd-irrelevant)
qed

definition srblit a s d l ≡ rblit a s a d l
```

However, within arrays we can now copy to the right.

```
lemma srblit-rule[sep-heap-rules]:
  assumes LEN:
    di+len ≤ length lsrc
    and DST-GR: di ≥ si
  shows
    < src ↦a lsrc >
    srblit src si di len
    <λ-. src ↦a (take di lsrc @ take len (drop si lsrc) @ drop (di+len) lsrc)
    >
  unfolding srblit-def
  using LEN DST-GR
proof (induction len arbitrary: lsrc si di)
  case 0 thus ?case by sep-auto
next
  case (Suc len)
  note [sep-heap-rules] = Suc.IH

  have[simp]: take len (drop si (lsrc[di + len := lsrc ! (si + len)]))
        = take len (drop si lsrc)
   by (metis Suc.prems(2) ab-semigroup-add-class.add.commute add-le-cancel-right
take-drop take-update-cancel)
  have [simp]: drop (di + len) (lsrc[di + len := lsrc ! (si + len)])
        = lsrc ! (si+len) # drop (Suc di + len) lsrc
     by (metis Suc.prems(1) add-Suc-right add-Suc-shift add-less-cancel-left ap-
pend-take-drop-id le-imp-less-Suc le-refl plus-1-eq-Suc same-append-eq take-update-cancel
upd-conv-take-nth-drop)
  have take len (drop si lsrc) @
    [lsrc ! (si + len)] = take (Suc len) (drop si lsrc)
  proof −
    have len < length lsrc − si
      using Suc.prems(1) Suc.prems(2) by linarith
    then show ?thesis
      by (metis (no-types) Suc.prems(1) Suc.prems(2) add-leD1 le-add-diff-inverse
length-drop nth-drop take-Suc-conv-app-nth)
  qed
  then have [simp]: ⋀x. take len (drop si lsrc) @
    lsrc ! (si + len) # x = take (Suc len) (drop si lsrc) @ x
    by simp
  from Suc.prems show ?case
    by (sep-auto simp: take-update-last drop-upd-irrelevant)
```

qed

5.2 Modeling target language blit

For convenience, a function that is oblivious to the direction of the shift is defined.

definition *safe-sblit a s d l* ≡
  if s > d then
    sblit a s d l
  else
    srblit a s d l

We obtain a heap rule similar to the one of blit, but for copying within one array.

lemma *safe-sblit-rule*[*sep-heap-rules*]:
  assumes *LEN*:
    len > 0 ⟶ *di+len* ≤ *length lsrc* ∧ *si+len* ≤ *length lsrc*
  shows
    < *src* ↦$_a$ *lsrc* >
    safe-sblit src si di len
    <λ-. *src* ↦$_a$ (*take di lsrc* @ *take len* (*drop si lsrc*) @ *drop* (*di+len*) *lsrc*)
    >
  unfolding *safe-sblit-def*
  using *LEN*
  apply(*cases len*)
   apply(*sep-auto simp add*: *sblit-def srblit-def*)[]
  apply *sep-auto*
  done

thm *blit-rule*
thm *safe-sblit-rule*

5.3 Code Generator Setup

Note that the requirement for correctness is even weaker here than in SML/OCaml. In particular, if the length of the slice to copy is equal to 0, we will never throw an exception. We therefore manually handle this case, where nothing happens at all.

code-printing code-module *array-sblit* ⇀ (*SML*)
  ‹
  fun array-sblit src si di len = (
    if len > 0 then
      ArraySlice.copy {
        di = *IntInf.toInt di*,
        src = *ArraySlice.slice* (*src*,*IntInf.toInt si*,*SOME* (*IntInf.toInt len*)),

```
        dst = src}
      else ()
    )
›

code-printing code-module array-sblit ⇀ (OCaml)
  ‹
   let array-sblit src si di len = (
      if len > Z.zero then
        (Array.blit src (Z.to-int si) src (Z.to-int di) (Z.to-int len))
      else ()
    )
›

definition safe-sblit' where
  [code del]: safe-sblit' src si di len
      = safe-sblit src (nat-of-integer si) (nat-of-integer di)
          (nat-of-integer len)

lemma [code]:
  safe-sblit src si di len
      = safe-sblit' src (integer-of-nat si) (integer-of-nat di)
          (integer-of-nat len) by (simp add: safe-sblit'-def)


code-printing constant safe-sblit' ⇀
  (SML) (fn/ ()/ => /array'-sblit - - - -)
  and (Scala) { ('-: Unit)/=>/
      def safescopy(src: Array['-], srci: Int, dsti: Int, len: Int) = {
       if (len > 0)
          System.arraycopy(src, srci, src, dsti, len)
        else
          ()
      }
      safescopy(-.array,-.toInt,-.toInt,-.toInt)
    }

code-printing constant safe-sblit' ⇀
  (OCaml) (fun () -> /array'-sblit - - - -)


export-code safe-sblit checking SML Scala OCaml
```

5.4 Derived operations

```
definition array-shr where
  array-shr a i k ≡ do {
  l ← Array.len a;
  safe-sblit a i (i+k) (l−(i+k))
```

}

find-theorems *Array.len*

lemma *array-shr-rule*[*sep-heap-rules*]:
  < *src* $\mapsto_a$ *lsrc* >
    array-shr src i k
    <λ-. *src* $\mapsto_a$ (*take* (*i*+*k*) *lsrc* @ *take* (*length lsrc* − (*i*+*k*)) (*drop i lsrc*))
    >
  unfolding *array-shr-def*
  by *sep-auto*

lemma *array-shr-rule-alt*:
  < *src* $\mapsto_a$ *lsrc* >
    array-shr src i k
    <λ-. *src* $\mapsto_a$ (*take* (*length lsrc*) (*take* (*i*+*k*) *lsrc* @ (*drop i lsrc*)))
    >
  by (*sep-auto simp add*: *min-def*)

definition *array-shl* **where**
  array-shl a i k ≡ *do* {
  l ← *Array.len a*;
  safe-sblit a i (*i*−*k*) (*l*−*i*)
}

lemma *array-shl-rule*[*sep-heap-rules*]:

    < *src* $\mapsto_a$ *lsrc* >
    array-shl src i k
    <λ-. *src* $\mapsto_a$ (*take* (*i*−*k*) *lsrc* @ (*drop i lsrc*) @ *drop* (*i* − *k* + (*length lsrc* − *i*))
lsrc)
    >
  unfolding *array-shl-def*
  by *sep-auto*

lemma *array-shl-rule-alt*:

    ⟦*i* ≤ *length lsrc*; *k* ≤ *i*⟧ ⟹
    < *src* $\mapsto_a$ *lsrc* >
    array-shl src i k
    <λ-. *src* $\mapsto_a$ (*take* (*i*−*k*) *lsrc* @ (*drop i lsrc*) @ *drop* (*length lsrc* − *k*) *lsrc*)
    >
  by *sep-auto*

end
theory *Partially-Filled-Array*

imports
Refine-Imperative-HOL.IICF-Array-List
Array-SBlit
begin

6 Partially Filled Arrays

An array that is only partially filled. The number of actual elements contained is kept in a second element. This represents a weakened version of the array_list from IICF.

type-synonym *'a pfarray = 'a array-list*

6.1 Operations on Partly Filled Arrays

definition *is-pfa* **where**
is-pfa c l ≡ λ(a,n). ∃$_A$ *l'*. *a* ↦$_a$ *l'* ∗ ↑(*c* = *length l'* ∧ *n* ≤ *c* ∧ *l* = (*take n l'*))

lemma *is-pfa-prec*[*safe-constraint-rules*]: *precise* (*is-pfa c*)
unfolding *is-pfa-def*[*abs-def*]
apply(*rule preciseI*)
apply(*simp split*: *prod.splits*)
using *preciseD snga-prec* **by** *fastforce*

definition *pfa-init* **where**
pfa-init cap v n ≡ *do* {
a ← *Array.new cap v*;
return (*a*,*n*)
}

lemma *pfa-init-rule*[*sep-heap-rules*]: *n* ≤ *N* ⟹ < *emp* > *pfa-init N x n* <*is-pfa N* (*replicate n x*)>
by (*sep-auto simp*: *pfa-init-def is-pfa-def*)

definition *pfa-empty* **where**
pfa-empty cap ≡ *pfa-init cap default 0*

lemma *pfa-empty-rule*[*sep-heap-rules*]: < *emp* > *pfa-empty N* <*is-pfa N* []>
by (*sep-auto simp*: *pfa-empty-def is-pfa-def*)

definition *pfa-length* ≡ *arl-length*

lemma *pfa-length-rule*[*sep-heap-rules*]:
<*is-pfa c l a*>
pfa-length a

<λr. is-pfa c l a ∗ ↑(r=length l)>
by (*sep-auto simp*: *pfa-length-def arl-length-def is-pfa-def*)

definition *pfa-capacity* ≡ λ(a,n). *Array.len a*

lemma *pfa-capacity-rule*[*sep-heap-rules*]:
<*is-pfa c l a*>
pfa-capacity a
<λr. *is-pfa c l a* ∗ ↑(c=r)>
by (*sep-auto simp*: *pfa-capacity-def arl-length-def is-pfa-def*)

definition *pfa-is-empty* ≡ *arl-is-empty*

lemma *pfa-is-empty-rule*[*sep-heap-rules*]:
<*is-pfa c l a*>
pfa-is-empty a
<λr. *is-pfa c l a* ∗ ↑(r⟷(l=[]))>
by (*sep-auto simp*: *pfa-is-empty-def arl-is-empty-def is-pfa-def*)

definition *pfa-append* ≡ λ(a,n) *x*. *do* {
Array.upd n x a;
return (a,n+1)
}

lemma *pfa-append-rule*[*sep-heap-rules*]:
n < *c* ⟹
< *is-pfa c l* (a,n) >
pfa-append (a,n) *x*
<λ(a′,n′). *is-pfa c* (*l*@[*x*]) (a′,n′) ∗ ↑(a′ = a ∧ n′ = n+1) >
by (*sep-auto*
simp: *pfa-append-def arl-append-def is-pfa-def take-update-last neq-Nil-conv*
split: *prod.splits nat.split*)

definition *pfa-last* ≡ *arl-last*

lemma *pfa-last-rule*[*sep-heap-rules*]:
l≠[] ⟹
<*is-pfa c l a*>
pfa-last a
<λr. *is-pfa c l a* ∗ ↑(r=*last l*)>

by (*sep-auto simp*: *pfa-last-def arl-last-def is-pfa-def last-take-nth-conv*)

definition *pfa-butlast* :: *'a::heap pfarray* ⇒ *'a pfarray Heap* **where**
pfa-butlast ≡ λ(*a*,*n*).
return (*a*,*n*−*1*)

lemma *pfa-butlast-rule*[*sep-heap-rules*]:
<*is-pfa c l* (*a*,*n*)>
pfa-butlast (*a*,*n*)
<λ(*a'*,*n'*). *is-pfa c* (*butlast l*) (*a'*,*n'*) ∗ ↑(*a'* = *a*)>
by (*sep-auto*
split: *prod.splits*
simp: *pfa-butlast-def is-pfa-def butlast-take*)

definition *pfa-get* ≡ *arl-get*

lemma *pfa-get-rule*[*sep-heap-rules*]:
i < *length l* ⟹
< *is-pfa c l a*>
pfa-get a i
<λ*r*. *is-pfa c l a* ∗ ↑((*l*!*i*) = *r*)>
by (*sep-auto simp*: *is-pfa-def pfa-get-def arl-get-def split*: *prod.split*)

definition *pfa-set* ≡ *arl-set*

lemma *pfa-set-rule*[*sep-heap-rules*]:
i<*length l* ⟹
<*is-pfa c l a*>
pfa-set a i x
<λ*a'*. *is-pfa c* (*l*[*i*:=*x*]) *a'* ∗ ↑(*a'* = *a*)>
by (*sep-auto simp*: *pfa-set-def arl-set-def is-pfa-def split*: *prod.split*)

definition *pfa-shrink* :: *nat* ⇒ *'a::heap pfarray* ⇒ *'a pfarray Heap* **where**
pfa-shrink k ≡ λ(*a*,*n*).
return (*a*,*k*)

lemma *pfa-shrink-rule*[*sep-heap-rules*]:
k ≤ *length xs* ⟹

```
    < is-pfa c xs (a,n) >
      pfa-shrink k (a,n)
    <λ(a',n'). is-pfa c (take k xs) (a',n') * ↑(n' = k ∧ a'=a) >
  by (sep-auto
      simp: pfa-shrink-def is-pfa-def min.absorb1
      split: prod.splits nat.split)


definition pfa-shrink-cap :: nat ⇒ 'a::heap pfarray ⇒ 'a pfarray Heap where
  pfa-shrink-cap k ≡ λ(a,n). do {
  a' ← array-shrink a k;
  return (a',min k n)
}


lemma pfa-shrink-cap-rule-preserve[sep-heap-rules]:
   ⟦n ≤ k; k ≤ c⟧ ⟹
    < is-pfa c l (a,n) >
      pfa-shrink-cap k (a,n)
    <λa'. is-pfa k l a' >t
  by (sep-auto
      simp: pfa-shrink-cap-def is-pfa-def min.absorb1 min.absorb2
      split: prod.splits nat.split)



lemma pfa-shrink-cap-rule:
   ⟦k ≤ c⟧ ⟹
    < is-pfa c l a >
      pfa-shrink-cap k a
    <λa'. is-pfa k (take k l) a' >t
  by (sep-auto
      simp: pfa-shrink-cap-def is-pfa-def min.absorb1 min.absorb2
      split: prod.splits nat.split dest: mod-starD)


definition array-ensure a s x ≡ do {
    l←Array.len a;
    if l≥s then
      return a
    else do {
      a'←Array.new s x;
      blit a 0 a' 0 l;
      return a'
    }
  }

lemma array-ensure-rule[sep-heap-rules]:
  shows
```

$< a \mapsto_a la >$
$array\text{-}ensure\ a\ s\ x$
$<\lambda a'.\ a' \mapsto_a (la\ @\ replicate\ (s - length\ la)\ x)>_t$
unfolding *array-ensure-def*
by *sep-auto*

definition *pfa-ensure* :: $'a::\{heap,default\}\ pfarray \Rightarrow nat \Rightarrow 'a\ pfarray\ Heap$ **where**
pfa-ensure $\equiv \lambda(a,n)\ k.\ do\ \{$
$a' \leftarrow array\text{-}ensure\ a\ k\ default;$
return (a',n)
$\}$

lemma *pfa-ensure-rule*[*sep-heap-rules*]:
$< is\text{-}pfa\ c\ l\ (a,n) >$
pfa-ensure $(a,n)\ k$
$<\lambda(a',n').\ is\text{-}pfa\ (max\ c\ k)\ l\ (a',n') * \uparrow(n' = n \wedge c \geq n)>_t$
by (*sep-auto*
simp: *pfa-ensure-def is-pfa-def*)

definition *pfa-copy* $\equiv$ *arl-copy*

lemma *pfa-copy-rule*[*sep-heap-rules*]:
$< is\text{-}pfa\ c\ l\ a >$
pfa-copy a
$<\lambda r.\ is\text{-}pfa\ c\ l\ a * is\text{-}pfa\ c\ l\ r>_t$
by (*sep-auto simp*: *pfa-copy-def arl-copy-def is-pfa-def*)

definition *pfa-blit* :: $'a::heap\ pfarray \Rightarrow nat \Rightarrow 'a::heap\ pfarray \Rightarrow nat \Rightarrow nat \Rightarrow$ *unit Heap* **where**
pfa-blit $\equiv \lambda(src,sn)\ si\ (dst,dn)\ di\ l.\ blit\ src\ si\ dst\ di\ l$

lemma *min-nat*: $min\ a\ (a+b) = (a::nat)$
by *auto*

lemma *pfa-blit-rule*[*sep-heap-rules*]:
assumes *LEN*: $si+len \leq sn\ di \leq dn\ di+len \leq dc$
shows
$< is\text{-}pfa\ sc\ src\ (srci,sn)$
$* is\text{-}pfa\ dc\ dst\ (dsti,dn) >$
pfa-blit $(srci,sn)\ si\ (dsti,dn)\ di\ len$
$<\lambda\text{-}.\ is\text{-}pfa\ sc\ src\ (srci,sn)$
$* is\text{-}pfa\ dc\ (take\ di\ dst\ @\ take\ len\ (drop\ si\ src)\ @\ drop\ (di+len)\ dst)\ (dsti,max\ (di+len)\ dn)$

```
    >
  using LEN apply(sep-auto simp add: min-nat is-pfa-def pfa-blit-def min.commute
min.absorb1 heap: blit-rule)
   apply (simp add: min.absorb1 take-drop)
  apply (simp add: drop-take max-def)
  done

definition pfa-drop :: ('a::heap) pfarray ⇒ nat ⇒ 'a pfarray ⇒ 'a pfarray Heap
where
  pfa-drop ≡ λ(src,sn) si (dst,dn). do {
   blit src si dst 0 (sn−si);
   return (dst,(sn−si))
}



lemma pfa-drop-rule[sep-heap-rules]:
  assumes LEN: k ≤ sn (sn−k) ≤ dc
  shows
    < is-pfa sc src (srci,sn)
      ∗ is-pfa dc dst (dsti,dn) >
    pfa-drop (srci,sn) k (dsti,dn)
    <λ(dsti',dn'). is-pfa sc src (srci,sn)
      ∗ is-pfa dc (drop k src) (dsti',dn')
      ∗ ↑(dsti' = dsti)
    >
   using LEN apply (sep-auto simp add: drop-take is-pfa-def pfa-drop-def dest!:
mod-starD heap: pfa-blit-rule)
  done


definition pfa-append-grow ≡ λ(a,n) x. do {
  l ← pfa-capacity (a,n);
  a' ← if l = n
  then array-grow a (l+1) x
  else Array.upd n x a;
  return (a',n+1)
}



lemma pfa-append-grow-full-rule[sep-heap-rules]: n = c ⟹
     <is-pfa c l (a,n)>
  array-grow a (c+1) x
     <λa'. is-pfa (c+1) (l@[x]) (a',n+1)>t
  apply(sep-auto simp add: is-pfa-def
      heap del: array-grow-rule)
  apply(vcg heap del: array-grow-rule heap add: array-grow-rule[of l (Suc (length
l)) a x])
```

```
  apply simp
 apply(rule ent-ex-postI[where ?x=l@[x]])
 apply sep-auto
 done


lemma pfa-append-grow-less-rule: n < c ⟹
 <is-pfa c l (a,n)>
    Array.upd n x a
<λa'. is-pfa c (l@[x]) (a',n+1)>t
 apply(sep-auto simp add: is-pfa-def take-update-last)
 done

lemma pfa-append-grow-rule[sep-heap-rules]:
 <is-pfa c l (a,n)>
 pfa-append-grow (a,n) x
 <λ(a',n'). is-pfa (if c = n then c+1 else c) (l@[x]) (a',n') * ↑(n'=n+1 ∧ c ≥
n)>t
 apply(subst pfa-append-grow-def)
 apply(rule hoare-triple-preI)
 apply (sep-auto
     heap add: pfa-append-grow-full-rule pfa-append-grow-less-rule)
  apply(sep-auto simp add: is-pfa-def)
 apply(sep-auto simp add: is-pfa-def)
 done


definition pfa-append-grow' ≡ λ(a,n) x. do {
 a' ← pfa-ensure (a,n) (n+1);
 a'' ← pfa-append a' x;
 return a''
}

lemma pfa-append-grow'-rule[sep-heap-rules]:
 <is-pfa c l (a,n)>
 pfa-append-grow' (a,n) x
 <λ(a',n'). is-pfa (max (n+1) c) (l@[x]) (a',n') * ↑(n'=n+1 ∧ c ≥ n)>t
 unfolding pfa-append-grow'-def
 by (sep-auto simp add: max-def)


definition pfa-insert ≡ λ(a,n) i x. do {
 a' ← array-shr a i 1;
 a'' ← Array.upd i x a;
 return (a'',n+1)
}


lemma list-update-last: length ls = Suc i ⟹ ls[i:=x] = (take i ls)@[x]
```

by (*metis append-eq-conv-conj length-Suc-rev-conv list-update-length*)

lemma *pfa-insert-rule*[*sep-heap-rules*]:
⟦$i \leq n$; $n < c$⟧ ⟹
<*is-pfa c l* (*a*,*n*)>
pfa-insert (*a*,*n*) *i x*
<λ(*a*′,*n*′). *is-pfa c* (*take i l*@*x*#*drop i l*) (*a*′,*n*′) ∗ ↑($n' = n+1 \wedge a=a'$)>
unfolding *pfa-insert-def is-pfa-def*
by (*sep-auto simp add*: *list-update-append1 list-update-last*
Suc-diff-le drop-take min-def)

definition *pfa-insert-grow* :: *'a*::{*heap*,*default*} *pfarray* ⇒ *nat* ⇒ *'a* ⇒ *'a pfarray Heap*
where *pfa-insert-grow* ≡ λ(*a*,*n*) *i x*. *do* {
a′ ← *pfa-ensure* (*a*,*n*) (*n*+*1*);
a′′ ← *pfa-insert a*′ *i x*;
return a′′
}

lemma *pfa-insert-grow-rule*[*sep-heap-rules*]:
$i \leq n$ ⟹
<*is-pfa c l* (*a*,*n*)>
pfa-insert-grow (*a*,*n*) *i x*
<λ(*a*′,*n*′). *is-pfa* (*max c* (*n*+*1*)) (*take i l*@*x*#*drop i l*) (*a*′,*n*′) ∗ ↑($n'=n+1 \wedge c \geq n$)>$_t$
unfolding *pfa-insert-grow-def*
by (*sep-auto heap add*: *pfa-insert-rule*[*of i n max c* (*Suc n*)])

definition *pfa-extend* **where**
pfa-extend ≡ λ (*a*,*n*) (*b*,*m*). *do*{
blit b 0 a n m;
return (*a*,*n*+*m*)
}

lemma *pfa-extend-rule*[*sep-heap-rules*]:
$n+m \leq c$ ⟹
<*is-pfa c l1* (*a*,*n*) ∗ *is-pfa d l2* (*b*,*m*)>
pfa-extend (*a*,*n*) (*b*,*m*)
<λ(*a*′,*n*′). *is-pfa c* (*l1*@*l2*) (*a*′,*n*′) ∗ ↑($a' = a \wedge n'=n+m$) ∗ *is-pfa d l2* (*b*,*m*)>$_t$
unfolding *pfa-extend-def*
by (*sep-auto simp add*: *is-pfa-def min.absorb1 min.absorb2 heap add*: *blit-rule*)

definition *pfa-extend-grow* **where**
pfa-extend-grow ≡ λ (*a*,*n*) (*b*,*m*). *do*{
a′ ← *array-ensure a* (*n*+*m*) *default*;
blit b 0 a′ *n m*;
return (*a*′,*n*+*m*)

}

lemma *pfa-extend-grow-rule*[*sep-heap-rules*]:
<*is-pfa c l1* (*a*,*n*) ∗ *is-pfa d l2* (*b*,*m*)>
pfa-extend-grow (*a*,*n*) (*b*,*m*)
<λ(*a*′,*n*′). *is-pfa* (*max c* (*n*+*m*)) (*l1*@*l2*) (*a*′,*n*′) ∗ ↑(*n*′=*n*+*m* ∧ *c* ≥ *n*) ∗ *is-pfa d l2* (*b*,*m*)>$_t$
unfolding *pfa-extend-grow-def*
by (*sep-auto simp add*: *is-pfa-def min.absorb1 min.absorb2 heap add*: *blit-rule*)

definition *pfa-append-extend-grow* **where**
pfa-append-extend-grow ≡ λ (*a*,*n*) *x* (*b*,*m*). *do*{
a′ ← *array-ensure a* (*n*+*m*+*1*) *default*;
a′′ ← *Array.upd n x a*′;
blit b 0 a′′ (*n*+*1*) *m*;
return (*a*′′,*n*+*m*+*1*)
}

lemma *pfa-append-extend-grow-rule*[*sep-heap-rules*]:
<*is-pfa c l1* (*a*,*n*) ∗ *is-pfa d l2* (*b*,*m*)>
pfa-append-extend-grow (*a*,*n*) *x* (*b*,*m*)
<λ(*a*′,*n*′). *is-pfa* (*max c* (*n*+*m*+*1*)) (*l1*@*x*#*l2*) (*a*′,*n*′) ∗ ↑(*n*′=*n*+*m*+*1* ∧ *c* ≥ *n*) ∗ *is-pfa d l2* (*b*,*m*)>$_t$
unfolding *pfa-append-extend-grow-def*
by (*sep-auto simp add*: *list-update-last is-pfa-def min.absorb1 min.absorb2 heap add*: *blit-rule*)

definition *pfa-delete* ≡ λ(*a*,*n*) *i*. *do* {
array-shl a (*i*+*1*) *1*;
return (*a*,*n*−*1*)
}

lemma *pfa-delete-rule*[*sep-heap-rules*]:
i < *n* ⟹
<*is-pfa c l* (*a*,*n*)>
pfa-delete (*a*,*n*) *i*
<λ(*a*′,*n*′). *is-pfa c* (*take i l*@*drop* (*i*+*1*) *l*) (*a*′,*n*′) ∗ ↑(*n*′ = *n*−*1* ∧ *a*=*a*′)>
unfolding *pfa-delete-def is-pfa-def*
apply (*sep-auto simp add*: *drop-take min-def*)
by (*metis Suc-diff-Suc diff-zero dual-order.strict-trans2 le-less-Suc-eq zero-le*)

end
theory *Basic-Assn*
imports
Refine-Imperative-HOL.Sepref-HOL-Bindings
Refine-Imperative-HOL.Sepref-Basic

begin

7 Auxilary imperative assumptions

The following auxiliary assertion types and lemmas were provided by Peter Lammich

7.1 List-Assn

lemma *list-assn-cong*[*fundef-cong*]:
⟦ *xs*=*xs'*; *ys*=*ys'*; ⋀*x y*. ⟦ *x*∈*set xs*; *y*∈*set ys* ⟧ ⟹ *A x y* = *A' x y* ⟧ ⟹ *list-assn A xs ys* = *list-assn A' xs' ys'*
by (*induction xs ys arbitrary*: *xs' ys' rule*: *list-assn.induct*) *auto*

lemma *list-assn-app-one*: *list-assn P* (*l1*@[*x*]) (*l1'*@[*y*]) = *list-assn P l1 l1'* ∗ *P x y*
by *simp*

lemma *list-assn-len*: *h* ⊨ *list-assn A xs ys* ⟹ *length xs* = *length ys*
using *list-assn-aux-ineq-len* **by** *fastforce*

lemma *list-assn-append-Cons-left*: *list-assn A* (*xs*@*x*#*ys*) *zs* = (∃$_A$ *zs1 z zs2*. *list-assn A xs zs1* ∗ *A x z* ∗ *list-assn A ys zs2* ∗ ↑(*zs1*@*z*#*zs2* = *zs*))
by (*sep-auto simp add*: *list-assn-aux-cons-conv list-assn-aux-append-conv1 intro*!: *ent-iffI*)

lemma *list-assn-aux-append-Cons*:
shows *length xs* = *length zsl* ⟹ *list-assn A* (*xs*@*x*#*ys*) (*zsl*@*z*#*zsr*) = (*list-assn A xs zsl* ∗ *A x z* ∗ *list-assn A ys zsr*)
by (*sep-auto simp add*: *mult.assoc*)

7.2 Prod-Assn

lemma *prod-assn-cong*[*fundef-cong*]:
⟦ *p*=*p'*; *pi*=*pi'*; *A* (*fst p*) (*fst pi*) = *A'* (*fst p*) (*fst pi*); *B* (*snd p*) (*snd pi*) = *B'* (*snd p*) (*snd pi*) ⟧
⟹ (*A*×$_a$*B*) *p pi* = (*A'*×$_a$*B'*) *p' pi'*
apply (*cases p*; *cases pi*)
by *auto*

end
theory *BTree-Imp*

imports
BTree
Partially-Filled-Array
Basic-Assn
begin

8 Imperative B-tree Definition

The heap data type definition. Anything stored on the heap always contains data, leafs are represented as None.

datatype *'a btnode* =
Btnode (*'a btnode ref option*∗*'a*) *pfarray 'a btnode ref option*

Selector Functions

primrec *kvs* :: *'a*::*heap btnode* ⇒ (*'a btnode ref option*∗*'a*) *pfarray* **where**
[*sep-dflt-simps*]: *kvs* (*Btnode ts* -) = *ts*

primrec *last* :: *'a*::*heap btnode* ⇒ *'a btnode ref option* **where**
[*sep-dflt-simps*]: *last* (*Btnode* - *t*) = *t*

term *arrays-update*

Encoding to natural numbers, as required by Imperative/HOL

fun
btnode-encode :: *'a*::*heap btnode* ⇒ *nat*
where
btnode-encode (*Btnode ts t*) = *to-nat* (*ts*, *t*)

instance *btnode* :: (*heap*) *heap*
apply (*rule heap-class.intro*)
apply (*rule countable-classI* [*of btnode-encode*])
apply (*metis btnode-encode.elims from-nat-to-nat fst-conv snd-conv*)
..

The refinement relationship to abstract B-trees.

fun *btree-assn* :: *nat* ⇒ *'a*::*heap btree* ⇒ *'a btnode ref option* ⇒ *assn* **where**
btree-assn k Leaf None = *emp* |
btree-assn k (*Node ts t*) (*Some a*) =
(∃*A tsi ti tsi'*.
a ↦*r Btnode tsi ti*
∗ *btree-assn k t ti*
∗ *is-pfa* (*2*∗*k*) *tsi' tsi*
∗ *list-assn* ((*btree-assn k*) ×*a id-assn*) *ts tsi'*
) |
btree-assn - - - = *false*

With the current definition of deletion, we would also need to directly reason on nodes and not only on references to them.

fun *btnode-assn* :: *nat* ⇒ *'a::heap btree* ⇒ *'a btnode* ⇒ *assn* **where**
btnode-assn k (*Node ts t*) (*Btnode tsi ti*) =
(∃$_A$ *tsi'*.
btree-assn k t ti
∗ *is-pfa* (*2∗k*) *tsi' tsi*
∗ *list-assn* ((*btree-assn k*) ×$_a$ *id-assn*) *ts tsi'*
) |
btnode-assn - - - = false

abbreviation *blist-assn k* ≡ *list-assn* ((*btree-assn k*) ×$_a$ *id-assn*)

end
theory *BTree-ImpSet*
imports
BTree-Imp
BTree-Set
begin

9 Imperative Set operations

9.1 Auxiliary operations

definition *split-relation xs* ≡
λ(*as*,*bs*) *i*. *i* ≤ *length xs* ∧ *as* = *take i xs* ∧ *bs* = *drop i xs*

lemma *split-relation-alt*:
split-relation as (*ls*,*rs*) *i* = (*as* = *ls*@*rs* ∧ *i* = *length ls*)
by (*auto simp add*: *split-relation-def*)

lemma *split-relation-length*: *split-relation xs* (*ls*,*rs*) (*length xs*) = (*ls* = *xs* ∧ *rs* = [])
by (*simp add*: *split-relation-def*)

lemma *list-assn-prod-map*: *list-assn* (*A* ×$_a$ *B*) *xs ys* = *list-assn B* (*map snd xs*) (*map snd ys*) ∗ *list-assn A* (*map fst xs*) (*map fst ys*)
apply(*induct* (*A* ×$_a$ *B*) *xs ys rule*: *list-assn.induct*)
apply(*auto simp add*: *ab-semigroup-mult-class.mult.left-commute ent-star-mono star-aci*(*2*) *star-assoc*)
done

lemma *id-assn-list*: *h* ⊨ *list-assn id-assn* (*xs*::*'a list*) *ys* ⟹ *xs* = *ys*
apply(*induction id-assn*::(*'a* ⇒ *'a* ⇒ *assn*) *xs ys rule*: *list-assn.induct*)
apply(*auto simp add*: *less-Suc-eq-0-disj pure-def*)
done

lemma *snd-map-help*:
  $x \leq$ *length tsi* $\Longrightarrow$
      $(\forall j{<}x.$ *snd* (*tsi* ! *j*) = ((*map snd tsi*)!*j*))
  $x <$ *length tsi* $\Longrightarrow$ *snd* (*tsi*!*x*) = ((*map snd tsi*)!*x*)
  by *auto*

lemma *split-ismeq*: $((a{::}nat) \leq b \wedge X) = ((a < b \wedge X) \vee (a = b \wedge X))$
  by *auto*

lemma *split-relation-map*: *split-relation as* (*ls*,*rs*) *i* $\Longrightarrow$ *split-relation* (*map f as*) (*map f ls*, *map f rs*) *i*
  apply(*induction as arbitrary*: *ls rs i*)
   apply(*auto simp add*: *split-relation-def take-map drop-Cons'*)
  apply(*metis list.simps*(*9*) *take-map*)
  done

lemma *split-relation-access*: $[\![$*split-relation as* (*ls*,*rs*) *i*; *rs* = *r*#*rrs*$]\!] \Longrightarrow$ *as*!*i* = *r*
  by (*simp add*: *split-relation-alt*)

lemma *index-to-elem-all*: $(\forall j{<}$*length xs*. *P* (*xs*!*j*)) = $(\forall x \in$ *set xs*. *P x*)
  by (*simp add*: *all-set-conv-nth*)

lemma *index-to-elem*: *n* < *length xs* $\Longrightarrow$ $(\forall j{<}n.$ *P* (*xs*!*j*)) = $(\forall x \in$ *set* (*take n xs*). *P x*)
  by (*simp add*: *all-set-conv-nth*)

definition *split-half* :: ('*a*::*heap* × '*b*::{*heap*}) *pfarray* ⇒ *nat Heap*
  where
    split-half a ≡ *do* {
  l ← *pfa-length a*;
  return (*l div 2*)
}

lemma *split-half-rule*[*sep-heap-rules*]: <
    is-pfa c tsi a
  ∗ *list-assn R ts tsi*>
    split-half a
  <λ*i*.
      is-pfa c tsi a
    ∗ *list-assn R ts tsi*
    ∗ ↑(*i* = *length ts div 2* ∧ *split-relation ts* (*BTree-Set.split-half ts*) *i*)>
  unfolding *split-half-def split-relation-def*
  apply(*rule hoare-triple-preI*)
  apply(*sep-auto dest*!: *list-assn-len mod-starD*)

done

9.2 The imperative split locale

This locale extends the abstract split locale, assuming that we are provided with an imperative program that refines the abstract split function.

locale *imp-split* = *abs-split*: *BTree-Set.split split*
for *split*::
('*a btree* × '*a*::{*heap*,*default*,*linorder*}) *list* ⇒ '*a*
⇒ ('*a btree* × '*a*) *list* × ('*a btree* × '*a*) *list* +
fixes *imp-split*:: ('*a btnode ref option* × '*a*::{*heap*,*default*,*linorder*}) *pfarray* ⇒ '*a* ⇒ *nat Heap*
assumes *imp-split-rule* [*sep-heap-rules*]:*sorted-less* (*separators ts*) ⟹
<*is-pfa c tsi* (*a*,*n*)
∗ *blist-assn k ts tsi*>
imp-split (*a*,*n*) *p*
<λ*i*.
is-pfa c tsi (*a*,*n*)
∗ *blist-assn k ts tsi*
∗ ↑(*split-relation ts* (*split ts p*) *i*)>$_t$
begin

9.3 Membership

partial-function (*heap*) *isin* :: '*a btnode ref option* ⇒ '*a* ⇒ *bool Heap*
where
isin p x =
(*case p of*
None ⇒ *return False* |
(*Some a*) ⇒ *do* {
node ← !*a*;
i ← *imp-split* (*kvs node*) *x*;
tsl ← *pfa-length* (*kvs node*);
if i < *tsl then do* {
s ← *pfa-get* (*kvs node*) *i*;
let (*sub*,*sep*) = *s in*
if x = *sep then*
return True
else
isin sub x
} *else*
isin (*last node*) *x*
}
)

9.4 Insertion

datatype '*b btupi* =
T_i '*b btnode ref option* |

```
  Up_i 'b btnode ref option 'b 'b btnode ref option

fun btupi-assn where
  btupi-assn k (abs-split.T_i l) (T_i li) =
   btree-assn k l li |
  btupi-assn k (abs-split.Up_i l a r) (Up_i li ai ri) =
   btree-assn k l li ∗ id-assn a ai ∗ btree-assn k r ri |
  btupi-assn - - - = false



definition node_i :: nat ⇒ ('a btnode ref option × 'a) pfarray ⇒ 'a btnode ref
option ⇒ 'a btupi Heap where
  node_i k a ti ≡ do {
    n ← pfa-length a;
    if n ≤ 2∗k then do {
      a' ← pfa-shrink-cap (2∗k) a;
      l ← ref (Btnode a' ti);
      return (T_i (Some l))
    }
    else do {
      b ← (pfa-empty (2∗k) :: ('a btnode ref option × 'a) pfarray Heap);
      i ← split-half a;
      m ← pfa-get a i;
      b' ← pfa-drop a (i+1) b;
      a' ← pfa-shrink i a;
      a'' ← pfa-shrink-cap (2∗k) a';
      let (sub,sep) = m in do {
        l ← ref (Btnode a'' sub);
        r ← ref (Btnode b' ti);
        return (Up_i (Some l) sep (Some r))
      }
    }
}


partial-function (heap) ins :: nat ⇒ 'a ⇒ 'a btnode ref option ⇒ 'a btupi Heap
  where
    ins k x apo = (case apo of
  None ⇒
    return (Up_i None x None) |
  (Some ap) ⇒ do {
    a ← !ap;
    i ← imp-split (kvs a) x;
    tsl ← pfa-length (kvs a);
    if i < tsl then do {
      s ← pfa-get (kvs a) i;
      let (sub,sep) = s in
      if sep = x then
```

```
      return (T_i apo)
    else do {
      r ← ins k x sub;
      case r of
        (T_i lp) ⇒ do {
          pfa-set (kvs a) i (lp,sep);
          return (T_i apo)
        } |
        (Up_i lp x' rp) ⇒ do {
          pfa-set (kvs a) i (rp,sep);
          if tsl < 2*k then do {
              kvs' ← pfa-insert (kvs a) i (lp,x');
              ap := (Btnode kvs' (last a));
              return (T_i apo)
          } else do {
            kvs' ← pfa-insert-grow (kvs a) i (lp,x');
            node_i k kvs' (last a)
          }
        }
      }
    else do {
      r ← ins k x (last a);
      case r of
        (T_i lp) ⇒ do {
          ap := (Btnode (kvs a) lp);
          return (T_i apo)
        } |
        (Up_i lp x' rp) ⇒
          if tsl < 2*k then do {
            kvs' ← pfa-append (kvs a) (lp,x');
            ap := (Btnode kvs' rp);
            return (T_i apo)
          } else do {
            kvs' ← pfa-append-grow' (kvs a) (lp,x');
            node_i k kvs' rp
        }
    }
  }
)
```

definition *insert* :: *nat* ⇒ ('*a*::{*heap*,*default*,*linorder*}) ⇒ '*a btnode ref option* ⇒ '*a btnode ref option Heap* **where**

```
  insert ≡ λk x ti. do {
  ti' ← ins k x ti;
  case ti' of
```

```
    T_i sub ⇒ return sub |
    Up_i l a r ⇒ do {
      kvs ← pfa-init (2∗k) (l,a) 1;
      t' ← ref (Btnode kvs r);
      return (Some t')
    }
}
```

9.5 Deletion

```
definition rebalance-middle-tree:: nat ⇒ (('a::{default,heap,linorder}) btnode ref
option × 'a) pfarray ⇒ nat ⇒ 'a btnode ref option ⇒ 'a btnode Heap
  where
    rebalance-middle-tree ≡ λ k tsi i r-ti. (
  case r-ti of
  None ⇒ do {
      (r-sub,sep) ← pfa-get tsi i;
      case r-sub of None ⇒  return (Btnode tsi r-ti)
  } |
  Some p-t ⇒ do {
      (r-sub,sep) ← pfa-get tsi i;
      case r-sub of (Some p-sub) ⇒  do {
      ti ← !p-t;
      sub ← !p-sub;
      l-sub ← pfa-length (kvs sub);
      l-tts ← pfa-length (kvs ti);
      if l-sub ≥ k ∧ l-tts ≥ k then do {
        return (Btnode tsi r-ti)
      } else do {
        l-tsi ← pfa-length tsi;
        if i+1 = l-tsi then do {
          mts' ← pfa-append-extend-grow (kvs sub) (last sub,sep) (kvs ti);
          res-node_i ← node_i k mts' (last ti);
          case res-node_i of
            T_i u ⇒ do {
              tsi' ← pfa-shrink i tsi;
              return (Btnode tsi' u)
            } |
            Up_i l a r ⇒ do {
              tsi' ← pfa-set tsi i (l,a);
              return (Btnode tsi' r)
            }
        } else do {
          (r-rsub,rsep) ← pfa-get tsi (i+1);
          case r-rsub of Some p-rsub ⇒ do {
            rsub ← !p-rsub;
            mts' ← pfa-append-extend-grow (kvs sub) (last sub,sep) (kvs rsub);
            res-node_i ← node_i k mts' (last rsub);
            case res-node_i of
```

```
            T_i u ⇒ do {
             tsi' ← pfa-set tsi i (u,rsep);
             tsi'' ← pfa-delete tsi' (i+1);
             return (Btnode tsi'' r-ti)
            } |
            Up_i l a r ⇒ do {
             tsi' ← pfa-set tsi i (l,a);
             tsi'' ← pfa-set tsi' (i+1) (r,rsep);
             return (Btnode tsi'' r-ti)
            }
           }
          }
         }
        }
  }
})
```

definition *rebalance-last-tree*:: *nat* ⇒ (('*a*::{*default*,*heap*,*linorder*}) *btnode ref option* × '*a*) *pfarray* ⇒ '*a btnode ref option* ⇒ '*a btnode Heap*
  where
   rebalance-last-tree ≡ λ*k tsi ti*. *do* {
  l-tsi ← *pfa-length tsi*;
  rebalance-middle-tree k tsi (*l-tsi*−*1*) *ti*
}

9.6 Refinement of the abstract B-tree operations

definition *empty* ::('*a*::{*default*,*heap*,*linorder*}) *btnode ref option Heap*
  where *empty* = *return None*

lemma *P-imp-Q-implies-P*: *P* ⟹ (*Q* ⟶ *P*)
  by *simp*

lemma *sorted-less* (*inorder t*) ⟹
  <*btree-assn k t ti*>
    isin ti x
  <λ*r*. *btree-assn k t ti* ∗ ↑(*abs-split.isin t x* = *r*)>$_t$
proof(*induction t x arbitrary*: *ti rule*: *abs-split.isin.induct*)
  case (*1 x*)
  then show *?case*
   apply(*subst isin.simps*)
   apply (*cases ti*)
    apply (*auto simp add*: *return-cons-rule*)
   done
next
  case (*2 ts t x*)

```
  then obtain ls rs where list-split[simp]: split ts x = (ls,rs)
    by (cases split ts x)
  then show ?case
  proof (cases rs)

    case [simp]: Nil
    show ?thesis
      apply(subst isin.simps)
      apply(sep-auto)
      using 2.prems sorted-inorder-separators apply blast
     apply(auto simp add: split-relation-def dest!: sym[of []] mod-starD list-assn-len)[]
      apply(rule hoare-triple-preI)
     apply(auto simp add: split-relation-def dest!: sym[of []] mod-starD list-assn-len)[]
     using 2(3) apply(sep-auto heap: 2.IH(1)[of ls []] simp add: sorted-wrt-append)
      done
  next
    case [simp]: (Cons h rrs)
    obtain sub sep where h-split[simp]: h = (sub,sep)
      by (cases h)
    show ?thesis
    proof (cases sep = x)

      case [simp]: True
      then show ?thesis
        apply(simp split: list.splits prod.splits)
        apply(subst isin.simps)
        using 2.prems sorted-inorder-separators apply(sep-auto)
         apply(rule hoare-triple-preI)
            apply(auto simp add: split-relation-alt list-assn-append-Cons-left dest!:
mod-starD list-assn-len)[]
        apply(rule hoare-triple-preI)
       apply(auto simp add: split-relation-def dest!: sym[of []] mod-starD list-assn-len)[]
        done
    next
      case [simp]: False
      show ?thesis
        apply(simp split: list.splits prod.splits)
        apply safe
        using False apply simp
        apply(subst isin.simps)
        using 2.prems sorted-inorder-separators
        apply(sep-auto)

            apply(auto simp add: split-relation-alt list-assn-append-Cons-left dest!:
mod-starD list-assn-len)[]

            apply(auto simp add: split-relation-alt list-assn-append-Cons-left dest!:
mod-starD list-assn-len)[]
```

```
        apply(rule norm-pre-ex-rule)+
        apply(rule hoare-triple-preI)
       subgoal for p tsi n ti xsi suba sepa zs1 z zs2 -
        apply(subgoal-tac z = (suba, sepa), simp)
        using 2(3) apply(sep-auto
            heap:2.IH(2)[of ls rs h rrs sub sep]
            simp add: sorted-wrt-append)
        using list-split Cons h-split apply simp-all

         apply(rule P-imp-Q-implies-P)
         apply(rule ent-ex-postI[where x=(tsi,n)])
         apply(rule ent-ex-postI[where x=ti])
         apply(rule ent-ex-postI[where x=(zs1 @ (suba, sepa) # zs2)])
         apply(rule ent-ex-postI[where x=zs1])
         apply(rule ent-ex-postI[where x=z])
         apply(rule ent-ex-postI[where x=zs2])
         apply sep-auto

      apply (metis (no-types, lifting) list-assn-aux-ineq-len list-assn-len nth-append-length
star-false-left star-false-right)
        done

       apply(rule hoare-triple-preI)
       apply(auto simp add: split-relation-def dest!: mod-starD list-assn-len)[]
       done
   qed
  qed
qed

declare abs-split.node_i.simps [simp add]

lemma node_i-rule: assumes c-cap: 2*k ≤ c c ≤ 4*k+1
  shows <is-pfa c tsi (a,n) * list-assn ((btree-assn k) ×_a id-assn) ts tsi * btree-assn
k t ti>
  node_i k (a,n) ti
  <λr. btupi-assn k (abs-split.node_i k ts t) r >_t
proof (cases length ts ≤ 2 * k)
  case [simp]: True
  then show ?thesis
    apply(subst node_i-def)
    apply(rule hoare-triple-preI)
    apply(sep-auto dest!: mod-starD list-assn-len)
      apply(sep-auto simp add: is-pfa-def)[]
    using c-cap apply(sep-auto simp add: is-pfa-def)[]
     apply(sep-auto  dest!: mod-starD list-assn-len)[]
    using True apply(sep-auto dest!: mod-starD list-assn-len)
    done
next
```

note *max.absorb1* [*simp del*] *max.absorb2* [*simp del*] *max.absorb3* [*simp del*] *max.absorb4* [*simp del*]
note *min.absorb1* [*simp del*] *min.absorb2* [*simp del*] *min.absorb3* [*simp del*] *min.absorb4* [*simp del*]
case [*simp*]: *False*
then obtain *ls sub sep rs* **where**
split-half-eq: *BTree-Set.split-half ts* = (*ls*,(*sub*,*sep*)#*rs*)
using *abs-split.node$_i$-cases* **by** *blast*
then show *?thesis*
apply(*subst node$_i$-def*)
apply(*rule hoare-triple-preI*)
apply(*sep-auto dest*!: *mod-starD list-assn-len*)
apply(*sep-auto simp add*: *split-relation-alt split-relation-length is-pfa-def dest*!: *mod-starD list-assn-len*)
using *False* **apply**(*sep-auto simp add*: *split-relation-alt*)
using *False* **apply**(*sep-auto simp add*: *is-pfa-def*)[]
apply(*sep-auto*)[]
apply(*sep-auto simp add*: *is-pfa-def split-relation-alt*)[]
using *c-cap* **apply**(*sep-auto simp add*: *is-pfa-def*)[]
apply(*sep-auto*)[]
using *c-cap* **apply**(*sep-auto simp add*: *is-pfa-def*)[]
using *c-cap* **apply**(*simp*)
apply(*vcg*)
apply(*simp*)
apply(*rule impI*)
subgoal for *- - - - rsa subi ba rn lsi al ar -*
thm *ent-ex-postI*
thm *ent-ex-postI*[**where** *x*=*take* (*length tsi div 2*) *tsi*]

apply(*rule ent-ex-postI*[**where** *x*=(*rsa*,*rn*)])
apply(*rule ent-ex-postI*[**where** *x*=*ti*])
apply(*rule ent-ex-postI*[**where** *x*=(*drop* (*Suc* (*length tsi div 2*)) *tsi*)])
apply(*rule ent-ex-postI*[**where** *x*=*lsi*])
apply(*rule ent-ex-postI*[**where** *x*=*subi*])
apply(*rule ent-ex-postI*[**where** *x*=*take* (*length tsi div 2*) *tsi*])

apply(*simp add*: *split-relation-alt*)
apply(*subgoal-tac tsi* =
take (*length tsi div 2*) *tsi* @ (*subi*, *ba*) # *drop* (*Suc* (*length tsi div 2*)) *tsi*)
apply(*rule back-subst*[**where** *a*=*blist-assn k ts* (*take* (*length tsi div 2*) *tsi* @ (*subi*, *ba*) # (*drop* (*Suc* (*length tsi div 2*)) *tsi*)) **and** *b*=*blist-assn k ts tsi*])
apply(*rule back-subst*[**where** *a*=*blist-assn k* (*take* (*length tsi div 2*) *ts* @ (*sub*, *sep*) # *rs*) **and** *b*=*blist-assn k ts*])
apply(*subst list-assn-aux-append-Cons*)
apply *sep-auto*
apply *sep-auto*
apply *simp*
apply *simp*
apply(*rule back-subst*[**where** *a*=*tsi* ! (*length tsi div 2*) **and** *b*=(*subi*, *ba*)])

apply(*rule id-take-nth-drop*)
apply *simp*
apply *simp*
done
done
qed
declare *abs-split.node$_i$.simps* [*simp del*]

lemma *node$_i$-no-split*: *length ts* $\leq$ *2*k* $\Longrightarrow$ *abs-split.node$_i$ k ts t* = *abs-split.T$_i$* (*Node ts t*)
by (*simp add*: *abs-split.node$_i$.simps*)

lemma *node$_i$-rule-app*: $[\![$*2*k* $\leq$ *c*; *c* $\leq$ *4*k+1*$]\!] \Longrightarrow$
<*is-pfa c* (*tsi'* @ [(*li*, *ai*)]) (*aa*, *al*) $*$
blist-assn k ls tsi' $*$
btree-assn k l li $*$
id-assn a ai $*$
btree-assn k r ri> *node$_i$ k* (*aa*, *al*) *ri*
<*btupi-assn k* (*abs-split.node$_i$ k* (*ls* @ [(*l*, *a*)]) *r*)>$_t$
proof −
note *node$_i$-rule*[*of k c* (*tsi'* @ [(*li*, *ai*)]) *aa al* (*ls* @ [(*l*, *a*)]) *r ri*]
moreover assume *2*k* $\leq$ *c c* $\leq$ *4*k+1*
ultimately show *?thesis*
by (*simp add*: *mult.left-assoc*)
qed

lemma *node$_i$-rule-ins2*: $[\![$*2*k* $\leq$ *c*; *c* $\leq$ *4*k+1*; *length ls* = *length lsi*$]\!] \Longrightarrow$
<*is-pfa c* (*lsi* @ (*li*, *ai*) # (*ri*,*a'i*) # *rsi*) (*aa*, *al*) $*$
blist-assn k ls lsi $*$
btree-assn k l li $*$
id-assn a ai $*$
btree-assn k r ri $*$
id-assn a' a'i $*$
blist-assn k rs rsi $*$
btree-assn k t ti> *node$_i$ k* (*aa*, *al*)
ti <*btupi-assn k* (*abs-split.node$_i$ k* (*ls* @ (*l*, *a*) # (*r*,*a'*) # *rs*) *t*)>$_t$
proof −
assume [*simp*]: *2*k* $\leq$ *c c* $\leq$ *4*k+1 length ls* = *length lsi*
moreover note *node$_i$-rule*[*of k c* (*lsi* @ (*li*, *ai*) # (*ri*,*a'i*) # *rsi*) *aa al* (*ls* @ (*l*, *a*) # (*r*,*a'*) # *rs*) *t ti*]
ultimately show *?thesis*
by (*simp add*: *mult.left-assoc list-assn-aux-append-Cons*)
qed

lemma *ins-rule*:
sorted-less (*inorder t*) $\Longrightarrow$ <*btree-assn k t ti*>
ins k x ti

```
  <λr. btupi-assn k (abs-split.ins k x t) r>t
proof (induction k x t arbitrary: ti rule: abs-split.ins.induct)
  case (1 k x)
  then show ?case
    apply(subst ins.simps)
    apply (sep-auto simp add: pure-app-eq)
    done
next
  case (2 k x ts t)
  obtain ls rrs where list-split: split ts x = (ls,rrs)
    by (cases split ts x)
  have [simp]: sorted-less (separators ts)
    using 2.prems sorted-inorder-separators by simp
  have [simp]: sorted-less (inorder t)
    using 2.prems sorted-inorder-induct-last by simp
  show ?case
  proof (cases rrs)
    case Nil
    then show ?thesis
    proof (cases abs-split.ins k x t)
      case (Ti a)
      then show ?thesis
        apply(subst ins.simps)
        apply(sep-auto)
        subgoal for p tsil tsin tti
          using Nil list-split
          by (simp add: list-assn-aux-ineq-len split-relation-alt)
        subgoal for p tsil tsin tti tsi' i tsin' - sub sep
          apply(rule hoare-triple-preI)
          using Nil list-split
          by (simp add: list-assn-aux-ineq-len split-relation-alt)
        subgoal for p tsil tsin tti tsi'
          thm 2.IH(1)[of ls rrs tti]
       using Nil list-split Ti apply(sep-auto split!: list.splits simp add: split-relation-alt
            heap add: 2.IH(1)[of ls rrs tti])
          subgoal for ai
            apply(cases ai)
             apply sep-auto
            apply sep-auto
            done
          done
        done
    next
      case (Upi l a r)
      then show ?thesis
        apply(subst ins.simps)
        apply(sep-auto)
        subgoal for p tsil tsin tti
          using Nil list-split
```

```
      by (simp add: list-assn-aux-ineq-len split-relation-alt)
    subgoal for p tsil tsin tti tsi' i tsin' - sub sep
      using Nil list-split
      by (simp add: list-assn-aux-ineq-len split-relation-alt)
    subgoal for p tsil tsin tti tsi' i tsin'
      thm 2.IH(1)[of ls rrs tti]
      using Nil list-split Up_i apply(sep-auto split!: list.splits
          simp add: split-relation-alt
          heap add: 2.IH(1)[of ls rrs tti])
      subgoal for ai
       apply(cases ai)
        apply sep-auto
       apply(rule hoare-triple-preI)
       apply(sep-auto)
         apply(auto dest!: mod-starD simp add: is-pfa-def)[]
        apply (sep-auto)
       subgoal for li ai ri
         apply(subgoal-tac length (ls @ [(l,a)]) ≤ 2*k)
          apply(simp add: node_i-no-split)
          apply(rule ent-ex-postI[where x=(tsil,Suc tsin)])
          apply(rule ent-ex-postI[where x=ri])
          apply(rule ent-ex-postI[where x=tsi' @ [(li, ai)]])
          apply(sep-auto)
         apply (sep-auto dest!: mod-starD list-assn-len simp add: is-pfa-def)
         done

         apply(sep-auto heap add: node_i-rule-app)
         done
       done
     done
  qed
 next
  case (Cons a rs)
  obtain sub sep where a-split: a = (sub,sep)
    by (cases a)
  then have [simp]: sorted-less (inorder sub)
   using 2.prems abs-split.split-axioms list-split Cons sorted-inorder-induct-subtree
split-def
    by fastforce
  then show ?thesis
  proof(cases x = sep)
    case True
    show ?thesis
      apply(subst ins.simps)
      apply(sep-auto)
      subgoal for p tsil tsin tti tsi j subi
        using Cons list-split a-split True
        by sep-auto
      subgoal for p tsil tsin tti tsi j - - subi sepi
```

```
    apply(rule hoare-triple-preI)
    using Cons list-split a-split True
    apply(subgoal-tac sepi = sep)
     apply (sep-auto simp add: split-relation-alt)
    apply(sep-auto simp add: list-assn-prod-map dest!: mod-starD id-assn-list)
    by (metis length-map snd-conv snd-map-help(2) split-relation-access)
  subgoal for p tsil tsin tti tsi j
    apply(rule hoare-triple-preI)
    using Cons list-split
    by (sep-auto simp add: split-relation-alt dest!: mod-starD list-assn-len)
  done
next
  case False
  then show ?thesis
  proof (cases abs-split.ins k x sub)
    case (T_i a')
    then show ?thesis
      apply(auto simp add: Cons list-split a-split False)
      using False apply simp
      apply(subst ins.simps)
      apply vcg
       apply auto
      apply(rule norm-pre-ex-rule)+

      apply vcg
       apply sep-auto
      using list-split Cons
      apply(simp add: split-relation-alt list-assn-append-Cons-left)
      apply (rule impI)
      apply(rule norm-pre-ex-rule)+
      apply(rule hoare-triple-preI)
      apply sep-auto

      subgoal for p tsil tsin ti zs1 subi sepi zs2 - - suba
        apply(subgoal-tac sepi = x)
        using list-split Cons a-split
         apply(auto dest!: mod-starD )[]
        apply(auto dest!: mod-starD list-assn-len)[]
        done

      subgoal for p tsil tsin ti zs1 subi sepi zs2 - - n z suba sepa
        apply (cases a, simp)
        apply(subgoal-tac subi = suba, simp)
        using list-split a-split T_i False
         apply (vcg heap: 2)
           apply(auto split!: btupi.splits)

         apply vcg
          apply simp
```

```
          apply vcg
          apply simp
         subgoal for a'i q r
          apply(rule impI)
          apply(simp add: list-assn-append-Cons-left)
          apply(rule ent-ex-postI[where x=(tsil,tsin)])
          apply(rule ent-ex-postI[where x=ti])
          apply(rule ent-ex-postI[where x=(zs1 @ (a'i, sepi) # zs2)])
          apply(rule ent-ex-postI[where x=zs1])
          apply(rule ent-ex-postI[where x=(a'i,sep)])
          apply(rule ent-ex-postI[where x=zs2])
          apply sep-auto
           apply (simp add: pure-app-eq)
          apply(sep-auto dest!: mod-starD list-assn-len)[]
          done
      apply (metis list-assn-aux-ineq-len Pair-inject list-assn-len nth-append-length
star-false-left star-false-right)
        done
       subgoal for p tsil tsin ti zs1 subi sepi zs2 - - suba
        apply(auto dest!: mod-starD list-assn-len)[]
        done
       done
    next
     case (Up_i l w r)
     then show ?thesis
      apply(auto simp add: Cons list-split a-split False)
      using False apply simp
      apply(subst ins.simps)
      apply vcg
       apply auto
      apply(rule norm-pre-ex-rule)+

      apply vcg
       apply sep-auto
      using list-split Cons
      apply(simp add: split-relation-alt list-assn-append-Cons-left)
      apply (rule impI)
      apply(rule norm-pre-ex-rule)+
      apply(rule hoare-triple-preI)
      apply sep-auto

      subgoal for p tsil tsin ti zs1 subi sepi zs2 - - suba
       apply(subgoal-tac sepi = x)
       using list-split Cons a-split
        apply(auto dest!: mod-starD )[]
       apply(auto dest!: mod-starD list-assn-len)[]
       done

      subgoal for p tsil tsin ti zs1 subi sepi zs2 - - n z suba sepa
```

apply(*subgoal-tac subi = suba*, *simp*)
thm *2(2)*[*of ls rrs a rs sub sep*]
using *list-split a-split Cons* Up_i *False*
apply (*sep-auto heap*: *2(2)*)
apply(*auto split*!: *btupi.splits*)

apply *vcg*
apply *simp*
subgoal for *li wi ri u*
apply (*cases u*,*simp*)
apply (*sep-auto dest*!: *mod-starD list-assn-len heap*: *pfa-insert-grow-rule*)
apply (*simp add*: *is-pfa-def*)[]
apply (*metis le-less-linear length-append length-take less-not-refl min.absorb2 trans-less-add1*)
apply(*simp add*: *is-pfa-def*)
apply (*metis add-Suc-right length-Cons length-append length-take min.absorb2*)
apply(*sep-auto split*: *prod.splits dest*!: *mod-starD list-assn-len*)[]

apply(*subgoal-tac length* (*ls* @ [(*l*,*w*)]) ≤ *2*∗*k*)
apply(*simp add*: $node_i$*-no-split*)
apply(*rule ent-ex-postI*[**where** *x*=(*tsil*,*Suc tsin*)])
apply(*rule ent-ex-postI*[**where** *x*=*ti*])
apply(*rule ent-ex-postI*[**where** *x*=(*zs1* @ (*li*, *wi*) # (*ri*, *sep*) # *zs2*)])
apply(*sep-auto dest*!: *mod-starD list-assn-len*)
apply (*sep-auto dest*!: *mod-starD list-assn-len simp add*: *is-pfa-def*)
done
apply *vcg*
apply *simp*
subgoal for *x21 x22 x23 u*
apply (*cases u*,*simp*)
thm *pfa-insert-grow-rule*[**where** *?l*=((*zs1* @ (*suba*, *sepi*) # *zs2*)[*length ls* := (*x23*, *sepa*)])]
apply (*sep-auto dest*!: *mod-starD list-assn-len heap*: *pfa-insert-grow-rule*)
apply (*simp add*: *is-pfa-def*)[]
apply (*metis le-less-linear length-append length-take less-not-refl min.absorb2 trans-less-add1*)
apply(*auto split*: *prod.splits dest*!: *mod-starD list-assn-len*)[]

apply (*vcg heap*: $node_i$*-rule-ins2*)
apply *simp*
apply *simp*
apply *simp*
apply *sep-auto*
done
apply(*auto dest*!: *mod-starD list-assn-len*)[]
done
subgoal for *p tsil tsin ti zs1 subi sepi zs2 - - suba*
apply(*auto dest*!: *mod-starD list-assn-len*)[]

```
        done
      done
    qed
  qed
 qed
qed
```

The imperative insert refines the abstract insert.

```
lemma insert-rule:
 assumes k > 0 sorted-less (inorder t)
 shows <btree-assn k t ti>
 insert k x ti
 <λr. btree-assn k (abs-split.insert k x t) r>t
 unfolding insert-def
 apply(cases abs-split.ins k x t)
  apply(sep-auto split!: btupi.splits heap: ins-rule[OF assms(2)])
 using assms
 apply(vcg heap: ins-rule[OF assms(2)])
 apply(simp split!: btupi.splits)
 apply(vcg)
  apply auto[]
 apply vcg
 apply auto[]
 subgoal for l a r li ai ri tsa tsn ti
   apply(rule ent-ex-postI[where x=(tsa,tsn)])
   apply(rule ent-ex-postI[where x=ri])
   apply(rule ent-ex-postI[where x=[(li, ai)]])
   apply sep-auto
   done
 done
```

The "pure" resulting rule follows automatically.

```
lemma insert-rule′:
 shows <btree-assn (Suc k) t ti ∗ ↑(abs-split.invar-inorder (Suc k) t ∧ sorted-less
(inorder t))>
 insert (Suc k) x ti
 <λri.∃A r. btree-assn (Suc k) r ri ∗ ↑(abs-split.invar-inorder (Suc k) r ∧ sorted-less
(inorder r) ∧ inorder r = (ins-list x (inorder t)))>t
 using abs-split.insert-bal abs-split.insert-order abs-split.insert-inorder
 by (sep-auto heap: insert-rule simp add: sorted-ins-list)

lemma list-update-length2 [simp]:
 (xs @ x # y # ys)[Suc (length xs) := z] = (xs @ x # z # ys)
 by (induct xs, auto)


lemma node_i-rule-ins: ⟦2∗k ≤ c; c ≤ 4∗k+1; length ls = length lsi⟧ ⟹
<is-pfa c (lsi @ (li, ai) # rsi) (aa, al) ∗
  blist-assn k ls lsi ∗
```

btree-assn k l li ∗
id-assn a ai ∗
blist-assn k rs rsi ∗
btree-assn k t ti>
node$_i$ *k* (*aa*, *al*) *ti*
<*btupi-assn k* (*abs-split.node*$_i$ *k* (*ls* @ (*l*, *a*) # *rs*) *t*)>$_t$
proof −
assume [*simp*]: *2*∗*k* ≤ *c c* ≤ *4*∗*k*+*1 length ls* = *length lsi*
moreover note *node*$_i$*-rule*[*of k c* (*lsi* @ (*li*, *ai*) # *rsi*) *aa al* (*ls* @ (*l*, *a*) # *rs*) *t ti*]
ultimately show *?thesis*
by (*simp add*: *mult.left-assoc list-assn-aux-append-Cons*)
qed

lemma *btupi-assn-T*: *h* ⊨ *btupi-assn k* (*abs-split.node*$_i$ *k ts t*) (*T*$_i$ *x*) ⟹ *abs-split.node*$_i$ *k ts t* = *abs-split.T*$_i$ (*Node ts t*)
apply(*auto simp add*: *abs-split.node*$_i$*.simps dest*!: *mod-starD split*!: *list.splits*)
done

lemma *btupi-assn-Up*: *h* ⊨ *btupi-assn k* (*abs-split.node*$_i$ *k ts t*) (*Up*$_i$ *l a r*) ⟹
abs-split.node$_i$ *k ts t* = (
case BTree-Set.split-half ts of (*ls*, (*sub*,*sep*)#*rs*) ⇒
abs-split.Up$_i$ (*Node ls sub*) *sep* (*Node rs t*))
apply(*auto simp add*: *abs-split.node*$_i$*.simps dest*!: *mod-starD split*!: *list.splits*)
done

lemma *second-last-access*:(*xs*@*a*#*b*#*ys*) ! *Suc*(*length xs*) = *b*
by (*simp add*: *nth-via-drop*)

lemma *pfa-assn-len*:*h* ⊨ *is-pfa k ls* (*a*,*n*) ⟹ *n* ≤ *k* ∧ *length ls* = *n*
by (*auto simp add*: *is-pfa-def*)

lemma *rebalance-middle-tree-rule*:
assumes *height t* = *height sub*
and *case rs of* (*rsub*,*rsep*) # *list* ⇒ *height rsub* = *height t* | [] ⇒ *True*
and *i* = *length ls*
shows <*is-pfa* (*2*∗*k*) *tsi* (*a*,*n*) ∗ *blist-assn k* (*ls*@(*sub*,*sep*)#*rs*) *tsi* ∗ *btree-assn k t ti*>
rebalance-middle-tree k (*a*,*n*) *i ti*
<λ*r*. *btnode-assn k* (*abs-split.rebalance-middle-tree k ls sub sep rs t*) *r* >$_t$
apply(*simp add*: *list-assn-append-Cons-left*)
apply(*rule norm-pre-ex-rule*)+
proof(*goal-cases*)
case (*1 lsi z rsi*)
then show *?case*
proof(*cases z*)
case *z-split*: (*Pair subi sepi*)
then show *?thesis*

```
  proof(cases sub)
    case sub-leaf[simp]: Leaf
    then have t-leaf[simp]: t = Leaf using assms
      by (cases t) auto
    show ?thesis
      apply (subst rebalance-middle-tree-def)
      apply (rule hoare-triple-preI)
      apply (vcg)
     using assms apply (auto dest!: mod-starD list-assn-len split!: option.splits)
      apply (vcg)
      apply (auto dest!: mod-starD list-assn-len split!: option.splits)
      apply (rule ent-ex-postI[where x=tsi])
      apply sep-auto
      done
  next
    case sub-node[simp]: (Node mts mt)
    then obtain tts tt where t-node[simp]: t = Node tts tt using assms
      by (cases t) auto
    then show ?thesis
    proof(cases length mts ≥ k ∧ length tts ≥ k)
      case True
      then show ?thesis
        apply(subst rebalance-middle-tree-def)
        apply(rule hoare-triple-preI)
        apply(sep-auto dest!: mod-starD)
        using assms apply (auto  dest!: list-assn-len)[]

        using assms apply(sep-auto  split!: prod.splits)
         using assms apply (auto simp del: height-btree.simps dest!: mod-starD
list-assn-len)[]
        using z-split apply(auto)[]
         subgoal for - - - - - - - - - tp tsia' tsin' - -  - - - - - - - - - tsia tsin tti ttsi
sepi subp
          apply(auto dest!: mod-starD list-assn-len simp: prod-assn-def)[]
          apply(vcg)
           apply(auto)[]
           apply(rule ent-ex-postI[where x=lsi@(Some subp, sepi)#rsi])
           apply(rule ent-ex-postI[where x=(tsia, tsin)])
           apply(rule ent-ex-postI[where x=tti])
           apply(rule ent-ex-postI[where x=ttsi])
           apply(sep-auto)[]
          apply(rule hoare-triple-preI)
          using True apply(auto dest!: mod-starD list-assn-len)
          done
        done
    next
      case False
      then show ?thesis
      proof(cases rs)
```

case *Nil*
then show *?thesis*
apply(*subst rebalance-middle-tree-def*)
apply(*rule hoare-triple-preI*)
apply(*sep-auto dest*!: *mod-starD*)
using *assms* **apply** (*auto dest*!: *list-assn-len*)[]

apply(*sep-auto split*!: *prod.splits*)
using *assms* **apply** (*auto simp del*: *height-btree.simps dest*!: *mod-starD list-assn-len*)[]
using *z-split* **apply**(*auto*)[]
subgoal for - - - - - - - - *tp tsia' tsin'* - - - - - - - - - *tsia tsin tti ttsi*
apply(*auto dest*!: *mod-starD list-assn-len simp*: *prod-assn-def*)[]
apply(*vcg*)
using *False* **apply**(*auto dest*!: *mod-starD list-assn-len*)
done
apply(*sep-auto dest*!: *mod-starD*)
using *assms* **apply** (*auto dest*!: *list-assn-len*)[]
using *assms* **apply** (*auto dest*!: *list-assn-len*)[]
apply(*sep-auto*)
using *assms* **apply** (*auto dest*!: *list-assn-len mod-starD*)[]
using *assms* **apply** (*auto dest*!: *list-assn-len mod-starD*)[]

subgoal for - - - - - - *tp tsia tsin tti ttsi* - - - - - - - - *tsia' tsin' tti' tsi' subi sepi subp*
apply(*subgoal-tac z* = (*subi*, *sepi*))
prefer *2*
apply (*metis assms*(*3*) *list-assn-len nth-append-length*)
apply *simp*
apply(*vcg*)
subgoal

apply(*rule entailsI*)

using *False* **apply** (*auto dest*!: *list-assn-len mod-starD*)[]
done
apply (*auto del*: *impCE*)
apply(*thin-tac* - ⊨ -)+
apply(*rule hoare-triple-preI*)

apply(*sep-auto heap add*: *node$_i$-rule-ins dest*!: *mod-starD del*: *impCE*)
apply (*auto dest*!: *pfa-assn-len*)[]
apply (*auto dest*!: *pfa-assn-len list-assn-len*)[]
subgoal
apply(*thin-tac* - ⊨ -)+
apply(*rule hoare-triple-preI*)
apply(*sep-auto split*!: *btupi.splits del*: *impCE*)
apply(*auto dest*!: *btupi-assn-T mod-starD del*: *impCE*)[]
apply(*rule ent-ex-postI*[**where** *x*=*lsi*])

```
            apply sep-auto
           apply (sep-auto del: impCE)
              apply(auto dest!: btupi-assn-Up mod-starD split!: list.splits del:
impCE)[]
           subgoal for li ai ri
            apply(rule ent-ex-postI[where x=lsi @ [(li, ai)]])
            apply sep-auto
            done
           done
          apply (sep-auto del: impCE)
          using assms apply(auto dest!: pfa-assn-len list-assn-len mod-starD)[]
          using assms apply(auto dest!: pfa-assn-len list-assn-len mod-starD)[]
          done
         done
       next
        case (Cons rss rrs)
        then show ?thesis
         apply(subst rebalance-middle-tree-def)
         apply(rule hoare-triple-preI)
         apply(sep-auto dest!: mod-starD)
         using assms apply (auto  dest!: list-assn-len)[]

          apply(sep-auto  split!: prod.splits)
          using assms apply (auto simp del: height-btree.simps dest!: mod-starD
list-assn-len)[]
          apply(auto)[]
         subgoal for - - - - - - - - tp tsia' tsin' - -  - - - - - - - - tsia tsin tti ttsi
          apply(auto dest!: mod-starD list-assn-len simp: prod-assn-def)[]
          apply(vcg)
          using False apply(auto dest!: mod-starD list-assn-len)
          done
         apply(sep-auto dest!: mod-starD del: impCE)
         using assms apply (auto dest!: list-assn-len)[]
         apply(sep-auto del: impCE)
         using assms apply (auto dest!: list-assn-len mod-starD)[]

         subgoal for list-heap1 list-heap2 - - - - - - tp ttsia' ttsin' tti' ttsi' - - - -
- - - - ttsia ttsin tti ttsi subi sepi subp
          apply(subgoal-tac z = (subi, sepi))
           prefer 2
           apply (metis assms(3) list-assn-len nth-append-length)
          apply simp
          apply(vcg)
          subgoal

           apply(rule entailsI)

           using False apply (auto dest!: list-assn-len mod-starD)[]
           done
```

apply *simp*
subgoal for *subtsi subti subts ti subi subtsl ttsl*

supply R = $node_i$*-rule-ins*[**where** k=k **and** c=(*max* (*2* $*$ k) (*Suc* (- + *ttsin*))) **and** *lsi*=*subts*]
thm R
apply(*cases subtsi*)
apply(*sep-auto heap add*: R *pfa-append-extend-grow-rule dest*!: *mod-starD del*: *impCE*)

using *assms* **apply** (*auto dest*!: *list-assn-len pfa-assn-len*)[]
using *assms* **apply** (*auto dest*!: *list-assn-len pfa-assn-len*)[]
using *assms* **apply** (*auto dest*!: *list-assn-len pfa-assn-len*)[]
apply(*sep-auto split*!: *btupi.splits del*: *impCE*)
using *assms* **apply** (*auto dest*!: *list-assn-len pfa-assn-len*)[]
apply(*thin-tac* - $\models$ -)+
apply(*rule hoare-triple-preI*)
apply (*cases rsi*)
apply(*auto dest*!: *list-assn-len mod-starD*)[]

subgoal for *subtsa subtsn mtsa mtsn mtt mtsi* - - - - - - - - *rsubsep* - *rrsi rssi*

apply (*cases rsubsep*)
apply(*subgoal-tac rsubsep* = *rrsi*)
prefer *2*
using *assms* **apply**(*auto dest*!: *list-assn-len mod-starD del*: *impCE simp add*: *second-last-access*)[]
apply (*simp add*: *prod-assn-def*)
apply(*cases rss*)
apply *simp*
subgoal for *rsubi rsepi rsub rsep*
apply(*subgoal-tac height rsub* $\neq$ *0*)
prefer *2*
using *assms* **apply**(*auto*)[]
apply(*cases rsubi*; *cases rsub*)
apply *simp*+

apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
subgoal for *rsubi rsubts rsubt rsubtsi′ rsubti rsubtsi subnode*
apply(*cases kvs subnode*)
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))

apply (*vcg* (*ss*))
subgoal for - *rsubtsn subtsmergedi*
apply (*cases subtsmergedi*)
apply *simp*
apply (*vcg* (*ss*))
subgoal for *subtsmergeda* -
supply R = $node_i$*-rule-ins*[**where**
k=*k* **and**
c=*max* (*2*∗*k*) (*Suc* (*subtsn* + *rsubtsn*)) **and**
ls=*mts* **and**
al=*Suc* (*subtsn*+*rsubtsn*) **and**
aa=*subtsmergeda* **and**
ti=*rsubti* **and**
rsi=*rsubtsi* **and**
li=*subti* **and** *a*=*sep* **and** *ai*=*sep*
]
thm *R*
apply(*rule P-imp-Q-implies-P*)
apply(*auto del*: *impCE dest*!: *mod-starD list-assn-len*)[]
apply(*rule hoare-triple-preI*)
apply(*subgoal-tac subtsn* ≤ *2*∗*k* ∧ *rsubtsn* ≤ *2*∗*k*)
prefer *2*
apply (*auto simp add*: *is-pfa-def*)[]
apply (*sep-auto heap add*: *R del*: *impCE*)
apply(*sep-auto split*!: *btupi.splits del*: *impCE*)
using *assms* **apply**(*auto dest*!: *mod-starD list-assn-len*)[]
apply(*sep-auto del*: *impCE*)
using *assms* **apply**(*auto dest*!: *mod-starD list-assn-len pfa-assn-len del*: *impCE*)[]
apply(*thin-tac* - ⊨ -)+
apply(*rule hoare-triple-preI*)
apply (*drule btupi-assn-T mod-starD* | *erule conjE exE*)+
apply *vcg*
apply *simp*
subgoal for *rsubtsi ai tsian*
apply(*cases tsian*)
apply *simp*
apply(*rule P-imp-Q-implies-P*)
apply(*rule ent-ex-postI*[**where** *x*=*lsi* @ (*ai*, *rsep*) # *rssi*])
apply(*rule ent-ex-postI*[**where** *x*=(*ttsia*, *ttsin*)])
apply(*rule ent-ex-postI*[**where** *x*=*tti*])
apply(*rule ent-ex-postI*[**where** *x*=*ttsi*])
using *assms* **apply** (*sep-auto dest*!: *list-assn-len*)
done
subgoal for - - *rsubp rsubtsa* - *li ai ri*
apply(*sep-auto del*: *impCE*)
using *assms* **apply**(*auto dest*!: *list-assn-len*)[]
apply(*sep-auto del*: *impCE*)

```
                    using assms apply(auto dest!: list-assn-len)[]
                    apply(thin-tac - ⊨ -)+
                    apply(rule hoare-triple-preI)
                   apply (drule btupi-assn-Up mod-starD | erule conjE exE)+
                    apply vcg

                    apply(simp split!: list.split)
                       apply(rule ent-ex-postI[where x=(lsi @ (li, ai) # (ri,
rsepi) # rssi)])
                     apply(rule ent-ex-postI[where x=(ttsia, ttsin)])
                     apply(rule ent-ex-postI[where x=tti])
                     apply(rule ent-ex-postI[where x=ttsi])
                    using assms apply (sep-auto dest!: list-assn-len)
                       apply(rule ent-ex-postI[where x=(lsi @ (li, ai) # (ri,
rsepi) # rssi)])
                    apply(rule ent-ex-postI[where x=(ttsia, ttsin)])
                    apply(rule ent-ex-postI[where x=tti])
                    apply(rule ent-ex-postI[where x=ttsi])
                    using assms apply (sep-auto dest!: list-assn-len)
                    done
                  done
                done
               done
              done
             done
            done
           done
          done
         done
        done
      qed
    qed
  qed
 qed
qed

lemma rebalance-last-tree-rule:
  assumes height t = height sub
    and ts = list@[(sub,sep)]
  shows <is-pfa (2*k) tsi tsia * blist-assn k ts tsi * btree-assn k t ti>
  rebalance-last-tree k tsia ti
  <λr. btnode-assn k (abs-split.rebalance-last-tree k ts  t) r >t
  apply(subst rebalance-last-tree-def)
  apply(rule hoare-triple-preI)
  using assms apply(auto dest!: mod-starD)
  apply(subgoal-tac length tsi − Suc 0 = length list)
   prefer 2
   apply(auto dest!: list-assn-len)[]
  using assms apply(sep-auto)
  supply R = rebalance-middle-tree-rule[where
      ls=list and
```

```
    rs=[] and
    i=length tsi - 1, simplified]
  apply(cases tsia)
  using R by blast

partial-function (heap) split-max ::nat ⇒ ('a::{default,heap,linorder}) btnode ref
option ⇒ ('a btnode ref option × 'a) Heap
  where
    split-max k r-t = (case r-t of Some p-t ⇒ do {
   t ← !p-t;
   (case (last t) of None ⇒ do {
      (sub,sep) ← pfa-last (kvs t);
      tsi' ← pfa-butlast (kvs t);
      p-t := Btnode tsi' sub;
      return (Some p-t, sep)
  } |
    Some x ⇒ do {
      (sub,sep) ← split-max k (Some x);
      p-t' ← rebalance-last-tree k (kvs t) sub;
      p-t := p-t';
      return (Some p-t, sep)
  })
})



declare abs-split.split-max.simps [simp del] abs-split.rebalance-last-tree.simps [simp
del] height-btree.simps [simp del]

lemma split-max-rule:
  assumes abs-split.nonempty-lasttreebal t
    and t ≠ Leaf
  shows <btree-assn k t ti>
  split-max k ti
  <((btree-assn k) ×a id-assn) (abs-split.split-max k t)>t
  using assms
proof(induction k t arbitrary: ti rule: abs-split.split-max.induct)
  case (2 Leaf)
  then show ?case by auto
next
  case (1 k ts tt)
  then show ?case
  proof(cases tt)
    case Leaf
    then show ?thesis
      apply(subst split-max.simps)
      apply (vcg)
      using assms apply auto[]
      apply (vcg (ss))
```

apply *simp*
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply(*rule hoare-triple-preI*)
apply (*vcg* (*ss*))
using *1* **apply**(*auto dest*!: *mod-starD*)[]
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
subgoal for *tp tsi tti tsi' tnode subsep sub sep*
apply(*cases tsi*)
apply(*rule hoare-triple-preI*)
apply (*vcg*)
apply(*auto simp add*: *prod-assn-def abs-split.split-max.simps split*!: *prod.splits*)
subgoal for *tsia tsin - - tsin' lastsep lastsub*
apply(*rule ent-ex-postI*[**where** *x*=(*tsia*, *tsin'*)])
apply(*rule ent-ex-postI*[**where** *x*=*sub*])
apply(*rule ent-ex-postI*[**where** *x*=(*butlast tsi'*)])
using *1* **apply** (*auto dest*!: *mod-starD simp add*: *list-assn-append-Cons-left*)
apply *sep-auto*
done
done
apply(*sep-auto*)
done
next
case (*Node tts ttt*)
have *IH-help*: *abs-split.nonempty-lasttreebal tt* $\Longrightarrow$
tt $\neq$ *Leaf* $\Longrightarrow$
<*btree-assn k* (*Node tts ttt*) (*Some ttp*)> *split-max k* (*Some ttp*) <(*btree-assn k* $\times_a$ *id-assn*) (*abs-split.split-max k tt*)>$_t$
for *ttp*
using *1.IH Node* **by** *blast*
obtain *butlasttts l-sub l-sep* **where** *ts-split*:*tts* = *butlasttts*@[(*l-sub*, *l-sep*)]
using *1 Node* **by** *auto*
from *Node* **show** *?thesis*
apply(*subst split-max.simps*)
apply (*vcg*)
using *1* **apply** *auto*[]
apply (*vcg* (*ss*))

apply *simp*
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
apply (*vcg* (*ss*))
using *1* **apply**(*auto dest*!: *mod-starD*)[]
apply (*vcg* (*ss*))
subgoal for *tp tsi tti tsi' tnode ttp*
using *1.prems* **apply** (*vcg heap add*: *IH-help*)
apply *simp*
apply *simp*
apply(*subst prod-assn-def*)
apply(*cases abs-split.split-max k tt*)
apply (*auto simp del*: *abs-split.split-max.simps abs-split.rebalance-last-tree.simps height-btree.simps*)[]
subgoal for *ttsubi ttmaxi ttsub ttmax butlasttsi' lasttssubi butlastts lasttssub lasttssepi lasttssep*
apply(*rule hoare-triple-preI*)
supply *R* = *rebalance-last-tree-rule*[**where** *k*=*k* **and** *tsia*=*tsi* **and** *ti*=*ttsubi* **and** *t*=*ttsub* **and** *tsi*=*tsi'* **and** *ts*= (*butlasttsi'* @ [(*lasttssubi*, *lasttssepi*)])
and *list*=*butlasttsi'* **and** *sub*=*lasttssubi* **and** *sep*=*lasttssepi*]
thm *R*
using *ts-split*

apply (*sep-auto heap add*: *R*
simp del: *abs-split.split-max.simps abs-split.rebalance-last-tree.simps height-btree.simps*
dest!: *mod-starD*)
apply (*metis abs-split.nonempty-lasttreebal.simps*(*2*) *abs-split.split-max-height btree.distinct*(*1*))
apply *simp*
apply(*rule hoare-triple-preI*)
apply (*simp add*: *prod-assn-def*)
apply *vcg*
apply(*subst abs-split.split-max.simps*)
using *1.prems* **apply**(*auto dest*!: *mod-starD split*!: *prod.splits btree.splits*)
subgoal for - - - - - - - - - - *tp'*
apply(*cases abs-split.rebalance-last-tree k* (*butlasttsi'* @ [(*lasttssubi*, *lasttssepi*)]) *ttsub*; *cases tp'*)
apply *auto*
apply(*rule ent-ex-preI*)
subgoal for - - *tsia' tsin' tt'* - *tsi'*
apply(*rule ent-ex-postI*[**where** *x*=(*tsia'*, *tsin'*)])
apply(*rule ent-ex-postI*[**where** *x*=*tt'*])
apply(*rule ent-ex-postI*[**where** *x*=*tsi'*])

```
            apply sep-auto
            done
          done
        done
      done
    done
  qed
qed

partial-function (heap) del ::nat ⇒ 'a ⇒ ('a::{default,heap,linorder}) btnode ref
option ⇒ 'a btnode ref option Heap
  where
    del k x ti = (case ti of None ⇒ return None |
   Some p ⇒ do {
   node ← !p;
   i ← imp-split (kvs node) x;
   tsl ← pfa-length (kvs node);
   if i < tsl then do {
     (sub,sep) ← pfa-get (kvs node) i;
     if sep ≠ x then do {
       sub' ← del k x sub;
       kvs' ← pfa-set (kvs node) i (sub',sep);
       node' ← rebalance-middle-tree k kvs' i (last node);
       p := node';
       return (Some p)
      }
     else if sub = None then do{
       kvs' ← pfa-delete (kvs node) i;
       p := (Btnode kvs' (last node));
       return (Some p)
     }
     else do {
        sm ← split-max k sub;
        kvs' ← pfa-set (kvs node) i sm;
        node' ← rebalance-middle-tree k kvs' i (last node);
        p := node';
        return (Some p)
     }
   } else do {
       t' ← del k x (last node);
       node' ← rebalance-last-tree k (kvs node) t';
       p := node';
       return (Some p)
    }
})

lemma rebalance-middle-tree-update-rule:
  assumes height tt = height sub
    and case rs of (rsub,rsep) # list ⇒ height rsub = height tt | [] ⇒ True
```

```
    and i = length ls
  shows <is-pfa (2 * k) (zs1 @ (x', sep) # zs2) a * btree-assn k sub x' *
     blist-assn k ls zs1 *
     id-assn sep sep *
     blist-assn k rs zs2 *
     btree-assn k tt ti>
  rebalance-middle-tree k a i ti
   <btnode-assn k (abs-split.rebalance-middle-tree k ls sub sep rs tt)>t
proof (cases a)
  case [simp]: (Pair a n)
  note R=rebalance-middle-tree-rule[of tt sub rs i ls k zs1@(x', sep)#zs2 a n sep
ti]
  show ?thesis
    apply(rule hoare-triple-preI)
   using R assms apply (sep-auto dest!: mod-starD list-assn-len simp add: prod-assn-def)
    using assn-times-assoc star-aci(3) by auto
qed

lemma del-rule:
  assumes bal t and sorted (inorder t) and root-order k t and k > 0
  shows <btree-assn k t ti>
  del k x ti
  <btree-assn k (abs-split.del k x t)>t
  using assms
proof (induction k x t arbitrary: ti rule: abs-split.del.induct)
  case (1 k x)
  then show ?case
    apply(subst del.simps)
    apply sep-auto
    done
next
  case (2 k x ts tt ti)
  obtain ls rs where split-ts[simp]: split ts x = (ls, rs)
    by (cases split ts x)
  obtain tss lastts-sub lastts-sep where last-ts: ts = tss@[(lastts-sub, lastts-sep)]
    using 2.prems  apply auto
    by (metis abs-split.isin.cases neq-Nil-rev-conv)
  show ?case
  proof(cases rs)
    case Nil
    then show ?thesis
      apply(subst del.simps)
      apply sep-auto
      using 2.prems(2) sorted-inorder-separators apply blast
      apply(rule hoare-triple-preI)
      apply (sep-auto)
     using Nil apply (auto simp add: split-relation-alt dest!: mod-starD list-assn-len)[]
     using Nil apply (auto simp add: split-relation-alt dest!: mod-starD list-assn-len)[]
     using Nil apply (auto simp add: split-relation-alt dest!: mod-starD list-assn-len)[]
```

```
      apply (sep-auto heap add: 2.IH(1))
      using 2.prems apply (auto dest!: mod-starD)[]
      using 2.prems apply (auto dest!: mod-starD simp add: sorted-wrt-append)[]
      using 2.prems order-impl-root-order apply (auto dest!: mod-starD)[]
      using 2.prems apply (auto)[]
      subgoal for tp tsia tsin tti tsi i - - tti'
       apply(rule hoare-triple-preI)
         supply R = rebalance-last-tree-rule[where t=(abs-split.del k x tt) and
ti=tti' and ts=ts and sub=lastts-sub
           and list=tss and sep=lastts-sep]
       thm R
       using last-ts apply(sep-auto heap add: R)
        using 2.prems abs-split.del-height[of k tt x] order-impl-root-order[of k tt]
apply (auto dest!: mod-starD)[]
        apply simp
       apply(rule hoare-triple-preI)
       apply (sep-auto)
       apply(cases abs-split.rebalance-last-tree k ts (abs-split.del k x tt))
        apply(auto simp add: split-relation-alt dest!: mod-starD list-assn-len)
       subgoal for tnode
        apply (cases tnode; sep-auto)
        done
       done
     done
  next
    case [simp]: (Cons rrs rss)
    then obtain sub sep where [simp]: rrs = (sub, sep)
     by (cases rrs)
    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 show ?thesis
      apply(subst del.simps)
      apply sep-auto
      using 2.prems(2) sorted-inorder-separators apply blast
      apply(vcg (ss))
      apply(vcg (ss))
      apply(vcg (ss))
       apply(vcg (ss))
       apply(vcg (ss))
      apply(vcg (ss))
      apply(vcg (ss))
      apply(vcg (ss))
      apply(vcg (ss))
       apply(vcg (ss))
```

apply(*vcg* (*ss*))
apply(*vcg* (*ss*))
apply *simp*
apply(*vcg* (*ss*))
apply(*vcg* (*ss*))
apply(*vcg* (*ss*))
subgoal for *tp tsi ti' tsi' tnode i tsi'l subsep subi sepi*

apply(*auto simp add*: *split-relation-alt list-assn-append-Cons-left*;
rule norm-pre-ex-rule; *rule norm-pre-ex-rule*; *rule norm-pre-ex-rule*;
rule hoare-triple-preI;
auto dest!: *mod-starD*)[]
apply (*auto simp add*: *split-relation-alt dest*!: *list-assn-len*)[]
subgoal for *lsi subi rsi*
apply(*subgoal-tac subi* = *None*)
prefer *2*
apply(*auto dest*!: *list-assn-len*)[]
supply *R* = *2.IH*(*2*)[*of ls rs rrs rss sub sep*]
thm *R*
using *split-ts* **apply**(*sep-auto heap add*: *R*)
using *2.prems* **apply** *auto*[]
apply (*metis 2.prems*(*2*) *sorted-inorder-induct-subtree*)
using *2.prems* **apply** *auto*[]
apply (*meson 2.prems*(*4*) *order-impl-root-order*)
using *2.prems*(*4*) **apply** *fastforce*
apply(*vcg* (*ss*))
apply(*vcg* (*ss*))
apply(*vcg* (*ss*))
apply (*auto simp add*: *split-relation-alt dest*!: *list-assn-len*)[]
apply(*vcg* (*ss*))
apply(*vcg* (*ss*); *simp*)
apply(*cases tsi*; *simp*)
subgoal for *subi' - tsia' tsin'*
supply *R* = *rebalance-middle-tree-update-rule*
thm *R*

apply(*auto dest*!: *list-assn-len*)[]
apply(*rule hoare-triple-preI*)
apply (*sep-auto heap add*: *R dest*!: *mod-starD*)
using *2.prems abs-split.del-height*[*of k sub x*] *order-impl-root-order*[*of k sub*] **apply** (*auto*)[]
using *2.prems* **apply** (*auto split*!: *list.splits*)[]
apply *auto*[]
apply *sep-auto*
subgoal for - - - - - - - - - - - - - - - *tnode''*
apply (*cases* (*abs-split.rebalance-middle-tree k ls* (*abs-split.del k x sub*) *sepi rss tt*); *cases tnode''*)
apply *sep-auto*
apply *sep-auto*

```
            done
          done
        done
       apply (auto simp add: split-relation-alt dest!: mod-starD list-assn-len)[]

      subgoal for subnode lsi subi rsi
        apply(subgoal-tac subi = Some subnode)
         prefer 2
         apply(auto dest!: list-assn-len)[]
        supply R = 2.IH(2)[of ls rs rrs rss sub sep]
        thm R
        using split-ts apply(sep-auto heap add: R)
        using 2.prems apply auto[]
          apply (metis 2.prems(2) sorted-inorder-induct-subtree)
        using 2.prems apply auto[]
         apply (meson 2.prems(4) order-impl-root-order)
        using 2.prems(4) apply fastforce
        apply(vcg (ss))
        apply(vcg (ss))
         apply(vcg (ss))
         apply (auto simp add: split-relation-alt dest!: list-assn-len)[]
        apply(vcg (ss))
        apply(vcg (ss); simp)
        apply(cases tsi; simp)
        subgoal for x' xab a n
          supply R = rebalance-middle-tree-update-rule
          thm R

          apply(auto dest!: list-assn-len)[]
          apply(rule hoare-triple-preI)
          apply (sep-auto heap add: R dest!: mod-starD)
           using 2.prems abs-split.del-height[of k sub x] order-impl-root-order[of
k sub] apply (auto)[]
          using 2.prems apply (auto split!: list.splits)[]
           apply auto[]
          apply sep-auto
          subgoal for - - - - - - - - - - - - - - tnode'
           apply (cases (abs-split.rebalance-middle-tree k ls (abs-split.del k x sub)
sepi rss tt); cases tnode')
            apply sep-auto
           apply sep-auto
           done
          done
        done
      done
     apply(rule hoare-triple-preI)
       using Cons  apply (auto simp add: split-relation-alt dest!: mod-starD
list-assn-len)[]
     done
```

```
    next
      case sep-x-Leaf
      then show ?thesis
        apply(subst del.simps)
        apply sep-auto
        using 2.prems(2) sorted-inorder-separators apply blast
        apply(vcg (ss))
        apply(vcg (ss))
        apply(vcg (ss))
         apply(vcg (ss))
         apply(vcg (ss))
        apply(vcg (ss))
        apply(vcg (ss))
        apply(vcg (ss))
        apply(vcg (ss))
         apply(vcg (ss))
          apply(vcg (ss))
          apply(vcg (ss))
          apply simp
         apply(vcg (ss))
         apply(vcg (ss))
         apply(vcg (ss))
        subgoal for tp tsi ti' tsi' tnode i tsi'l subsep subi sepi

          apply(auto simp add: split-relation-alt list-assn-append-Cons-left;
              rule norm-pre-ex-rule; rule norm-pre-ex-rule; rule norm-pre-ex-rule;
              rule hoare-triple-preI;
              auto dest!: mod-starD)[]

          subgoal for lsi subi rsi
            apply(cases tsi)
            apply (sep-auto)
             apply(auto simp add: is-pfa-def dest!: list-assn-len)[]
               apply (metis add-Suc-right le-imp-less-Suc length-append length-take
less-add-Suc1 less-trans-Suc list.size(4) min.cobounded2 not-less-eq)
            apply vcg
            apply auto
            subgoal for tsin tsia
              apply(rule ent-ex-postI[where x=(tsia, tsin-1)])
              apply(rule ent-ex-postI[where x=ti'])
              apply(rule ent-ex-postI[where x=lsi@rsi])
              apply (sep-auto dest!: list-assn-len)
              done
            done
            apply (auto simp add: split-relation-alt dest!: list-assn-len)[]
           apply (auto simp add: split-relation-alt dest!: list-assn-len)[]
          apply (auto simp add: split-relation-alt dest!: list-assn-len)[]
          done
        apply(rule hoare-triple-preI)
```

```
        using Cons  apply (auto simp add: split-relation-alt dest!: mod-starD
list-assn-len)[]
      done
  next
    case sep-x-Node
    then show ?thesis
      apply(subst del.simps)
      apply sep-auto
      using 2.prems(2) sorted-inorder-separators apply blast
      apply(vcg (ss))
      apply(vcg (ss))
      apply(vcg (ss))
       apply(vcg (ss))
       apply(vcg (ss))
      apply(vcg (ss))
      apply(vcg (ss))
      apply(vcg (ss))
      apply(vcg (ss))
       apply(vcg (ss))
        apply(vcg (ss))
        apply(vcg (ss))
        apply simp
       apply(vcg (ss))
       apply(vcg (ss))
       apply(vcg (ss))
      subgoal for subts subt tp tsi ti tsi' tnode i tsi'l subsep subi sepi

        apply(auto simp add: split-relation-alt list-assn-append-Cons-left;
            rule norm-pre-ex-rule; rule norm-pre-ex-rule; rule norm-pre-ex-rule;
            rule hoare-triple-preI;
            auto dest!: mod-starD)[]
          apply (auto simp add: split-relation-alt dest!: list-assn-len)[]
         apply (auto simp add: split-relation-alt dest!: list-assn-len)[]

        subgoal for subnode lsi subi rsi
          apply(subgoal-tac subi = Some subnode)
           apply (simp del: btree-assn.simps)
           supply R = split-max-rule[of Node subts subt k Some subnode]
          thm R
           apply(sep-auto heap add: R simp del: btree-assn.simps)
             using 2.prems apply(auto dest!: list-assn-len mod-starD simp del:
bal.simps order.simps)[]
          subgoal
          proof(goal-cases)
            case 1
            then have order k (Node subts subt)
              by blast
            moreover have k > 0
              by (simp add: 2.prems(4))
```

ultimately obtain *sub-ls lsub lsep* **where** *sub-ts-split*: *subts* = *sub-ls*@[(*lsub*,*lsep*)]
by (*metis abs-split.isin.cases le-0-eq list.size*(*3*) *order.simps*(*2*) *rev-exhaust zero-less-iff-neq-zero*)
from *1* **have** *bal* (*Node subts subt*)
by *auto*
then have *height lsub* = *height subt*
by (*simp add*: *sub-ts-split*)
then show *?thesis* **using** *sub-ts-split* **by** *blast*
qed
using *2.prems abs-split.order-bal-nonempty-lasttreebal*[*of k subt*] *order-impl-root-order*[*of k subt*]
apply(*auto*)[]
apply (*auto simp add*: *split-relation-alt dest*!: *list-assn-len*)[]
apply *vcg*
apply *auto*[]
apply(*cases abs-split.split-max k* (*Node subts subt*); *simp*)
subgoal for *split-res - split-sub split-sep*
apply(*cases split-res*; *simp*)
subgoal for *split-subi split-sepi*
supply *R* = *rebalance-middle-tree-update-rule*[
of tt split-sub rss length lsi ls k lsi split-subi split-sep rsi tsi ti
]
thm *R*

apply(*auto simp add*: *prod-assn-def dest*!: *list-assn-len*)
apply (*sep-auto*)
apply(*rule hoare-triple-preI*)
apply(*auto dest*!: *mod-starD*)[]
apply (*sep-auto heap add*: *R*)
using *2.prems abs-split.split-max-height*[*of k sub*] *order-impl-root-order*[*of k sub*]
abs-split.order-bal-nonempty-lasttreebal[*of k sub*] **apply** (*auto*)[]
using *2.prems abs-split.split-max-bal*[*of sub k*] *order-impl-root-order*[*of k sub*]
apply (*auto split*!: *list.splits*)[]
apply *auto*[]
apply(*rule hoare-triple-preI*)
apply(*auto dest*!: *mod-starD*)[]
subgoal for *subtsi''a subtsi''n ti subtsi'' tnode'*
apply(*cases* (*abs-split.rebalance-middle-tree k ls split-sub split-sep rss tt*); *cases tnode'*)
apply *auto*
apply *sep-auto*
done
done
done
apply (*auto simp add*: *split-relation-alt dest*!: *list-assn-len*)[]
done

```
        apply (auto simp add: split-relation-alt dest!: list-assn-len)[]
        done
      apply(rule hoare-triple-preI)
       using Cons  apply (auto simp add: split-relation-alt dest!: mod-starD
list-assn-len)[]
      done
   qed
  qed
qed

definition reduce-root ::('a::{default,heap,linorder}) btnode ref option ⇒ 'a btnode
ref option Heap
  where
    reduce-root ti = (case ti of
  None ⇒ return None |
  Some p-t ⇒ do {
    node ← !p-t;
    tsl ← pfa-length (kvs node);
    case tsl of 0 ⇒ return (last node) |
    - ⇒ return ti
})

lemma reduce-root-rule:
  <btree-assn k t ti> reduce-root ti <btree-assn k (abs-split.reduce-root t)>t
  apply(subst reduce-root-def)
  apply(cases t; cases ti)
     apply (sep-auto split!: nat.splits list.splits)+
  done

definition delete ::nat ⇒ 'a ⇒ ('a::{default,heap,linorder}) btnode ref option ⇒
'a btnode ref option Heap
  where
    delete k x ti = do {
  ti' ← del k x ti;
  reduce-root ti'
}

lemma delete-rule:
  assumes bal t and root-order k t and k > 0 and sorted (inorder t)
  shows <btree-assn k t ti> delete k x ti <btree-assn k (abs-split.delete k x t)>t
  apply(subst delete-def)
  using assms apply (sep-auto heap add: del-rule reduce-root-rule)
  done

lemma empty-rule:
  shows <emp>
  empty
  <λr. btree-assn k (abs-split.empty-btree) r>
  apply(subst empty-def)
```

apply(*sep-auto simp add*: *abs-split.empty-btree-def*)
done

end

end
theory *Imperative-Loops*
imports
Refine-Imperative-HOL.Sepref-HOL-Bindings
Refine-Imperative-HOL.Pf-Mono-Prover
Refine-Imperative-HOL.Pf-Add

begin

10 Imperative Loops

An auxiliary while rule provided by Peter Lammich.

lemma *heap-WHILET-rule*:
assumes
wf R
$P \Longrightarrow_A I\ s$
$\bigwedge s.\ <I\ s * true>\ bi\ s\ <\lambda r.\ I\ s * \uparrow(r \longleftrightarrow b\ s)>_t$
$\bigwedge s.\ b\ s \Longrightarrow <I\ s * true>\ f\ s\ <\lambda s'.\ I\ s' * \uparrow((s',\ s) \in R)>_t$
$\bigwedge s.\ \neg\ b\ s \Longrightarrow I\ s \Longrightarrow_A Q\ s$
shows $<P * true>$ *heap-WHILET bi f s* $<Q>_t$
proof −
have $<I\ s * true>$ *heap-WHILET bi f s* $<\lambda s'.\ I\ s' * \uparrow(\neg\ b\ s')>_t$
using *assms*(*1*)
proof (*induction arbitrary*:)
case (*less s*)
show *?case*
proof (*cases b s*)
case *True*
then show *?thesis*
by (*subst heap-WHILET-unfold*) (*sep-auto heap*: *assms*(*3*,*4*) *less*)
next
case *False*
then show *?thesis*
by (*subst heap-WHILET-unfold*) (*sep-auto heap*: *assms*(*3*))
qed
qed
then show *?thesis*
apply (*rule cons-rule*[*rotated 2*])
apply (*intro ent-star-mono assms*(*2*) *ent-refl*)
apply *clarsimp*
apply (*intro ent-star-mono assms*(*5*) *ent-refl*)
.

qed

lemma *heap-WHILET-rule'*:
assumes
wf R
$P \Longrightarrow_A I\ s\ si * F$
$\bigwedge si\ s.\ <I\ s\ si * F>\ bi\ si\ <\lambda r.\ I\ s\ si * F * \uparrow(r \longleftrightarrow b\ s)>_t$
$\bigwedge si\ s.\ b\ s \Longrightarrow <I\ s\ si * F>\ f\ si\ <\lambda si'.\ \exists_A s'.\ I\ s'\ si' * F * \uparrow((s',\ s) \in R)>_t$
$\bigwedge si\ s.\ \neg\ b\ s \Longrightarrow I\ s\ si * F \Longrightarrow_A Q\ s\ si$
shows $<P>$ *heap-WHILET bi f si* $<\lambda si.\ \exists_A s.\ Q\ s\ si>_t$
proof −
have $<I\ s\ si * F>$ *heap-WHILET bi f si* $<\lambda si'.\ \exists_A s'.\ I\ s'\ si' * F * \uparrow(\neg\ b\ s')>_t$
using *assms(1)*
proof (*induction arbitrary*: *si*)
case (*less s*)
show *?case*
proof (*cases b s*)
case *True*
then show *?thesis*
apply (*subst heap-WHILET-unfold*)
apply (*sep-auto heap*: *assms(3,4) less*)
done
next
case *False*
then show *?thesis*
by (*subst heap-WHILET-unfold*) (*sep-auto heap*: *assms(3)*)
qed
qed
then show *?thesis*
apply (*rule cons-rule*[*rotated 2*])
apply (*intro ent-star-mono assms(2) ent-refl*)
apply *clarsimp*
apply (*sep-auto*)
apply (*erule ent-frame-fwd*[*OF assms(5)*])
apply *frame-inference*
by *sep-auto*

qed

I derived my own version, simply because it was a better fit to my use case.

corollary *heap-WHILET-rule''*:
assumes
wf R
$P \Longrightarrow_A I\ s$
$\bigwedge s.\ <I\ s * true>\ bi\ s\ <\lambda r.\ I\ s * \uparrow(r \longleftrightarrow b\ s)>_t$
$\bigwedge s.\ b\ s \Longrightarrow <I\ s * true>\ f\ s\ <\lambda s'.\ I\ s' * \uparrow((s',\ s) \in R)>_t$
$\bigwedge s.\ \neg\ b\ s \Longrightarrow I\ s \Longrightarrow_A Q\ s$
shows $<P>$ *heap-WHILET bi f s* $<Q>_t$

supply R = *heap-WHILET-rule'*[*of R P* $\lambda s\ si.\ \uparrow(s = si) * I\ s\ s$ - *true bi b f* $\lambda s\ si.\uparrow(s = si) * Q\ s * true$]
thm R
using *assms ent-true-drop* **apply**(*sep-auto heap*: R *assms*)
done

end
theory *BTree-ImpSplit*
imports
BTree-ImpSet
BTree-Split
Imperative-Loops
begin

11 Imperative split operations

So far, we have only given a functional specification of a possible split. We will now provide imperative split functions that refine the functional specification. However, rather than tracing the execution of the abstract specification, the imperative versions are implemented using while-loops.

11.1 Linear split

The linear split is the most simple split function for binary trees. It serves a good example on how to use while-loops in Imperative/HOL and how to prove Hoare-Triples about its application using loop invariants.

definition *lin-split* :: ('*a*::*heap* × '*b*::{*heap*,*linorder*}) *pfarray* ⇒ '*b* ⇒ *nat Heap*
where
lin-split ≡ λ (*a*,*n*) *p*. *do* {

i ← *heap-WHILET*
(λ*i*. *if i*<*n then do* {
(-,*s*) ← *Array.nth a i*;
return (*s*<*p*)
} *else return False*)
(λ*i*. *return* (*i*+*1*))
0;

return i
}

lemma *lin-split-rule*:
< *is-pfa c xs* (*a*,*n*)>
lin-split (*a*,*n*) *p*
<λ*i*. *is-pfa c xs* (*a*,*n*) ∗ ↑(*i*≤*n* ∧ (∀ *j*<*i*. *snd* (*xs*!*j*) < *p*) ∧ (*i*<*n* ⟶ *snd* (*xs*!*i*)≥*p*))>$_t$

unfolding *lin-split-def*

supply R = *heap-WHILET-rule''*[**where**
R = *measure* ($\lambda i.\ n - i$)
and I = $\lambda i.$ *is-pfa c xs* (a,n) $*$ $\uparrow(i \leq n \wedge (\forall j<i.\ snd\ (xs!j) < p))$
and b = $\lambda i.\ i<n \wedge snd\ (xs!i) < p$
and Q=$\lambda i.$ *is-pfa c xs* (a,n) $*$ $\uparrow(i \leq n \wedge (\forall j<i.\ snd\ (xs!j) < p) \wedge (i<n \longrightarrow snd\ (xs!i) \geq p))$
]
thm R

apply (*sep-auto decon*: R *simp*: *less-Suc-eq is-pfa-def*) []
apply (*metis nth-take snd-eqD*)
apply (*metis nth-take snd-eqD*)
apply (*sep-auto simp*: *is-pfa-def less-Suc-eq*)+
apply (*metis nth-take*)
apply(*sep-auto simp*: *is-pfa-def*)
apply (*metis le-simps*(*3*) *less-Suc-eq less-le-trans nth-take*)
apply(*sep-auto simp*: *is-pfa-def*)+
done

11.2 Binary split

To obtain an efficient B-Tree implementation, we prefer a binary split function. To explore the searching procedure and the resulting proof, we first implement the split on singleton arrays.

definition *bin'-split* :: *'b*::{*heap*,*linorder*} *array-list* $\Rightarrow$ *'b* $\Rightarrow$ *nat Heap*
where
bin'-split $\equiv$ λ(a,n) p. *do* {
(*low'*,*high'*) $\leftarrow$ *heap-WHILET*
(λ(*low*,*high*). *return* (*low* < *high*))
(λ(*low*,*high*). *let mid* = ((*low* + *high*) *div 2*) *in*
do {
$s \leftarrow$ *Array.nth a mid*;
if $p < s$ *then*
return (*low*, *mid*)
else if $p > s$ *then*
return (*mid*+*1*, *high*)
else return (*mid*,*mid*)
})
(*0*::*nat*,n);
return low'
}

thm *sorted-wrt-nth-less*

lemma *bin'-split-rule*:

sorted-less xs ⟹
< *is-pfa c xs* (*a*,*n*)>
bin′-split (*a*,*n*) *p*
<λ*l*. *is-pfa c xs* (*a*,*n*) ∗ ↑(*l* ≤ *n* ∧ (∀*j*<*l*. *xs*!*j* < *p*) ∧ (*l*<*n* ⟶ *xs*!*l*≥*p*)) >$_t$
unfolding *bin′-split-def*

supply *R* = *heap-WHILET-rule′′*[**where**
R = *measure* (λ(*l*,*h*). *h*−*l*)
and *I* = λ(*l*,*h*). *is-pfa c xs* (*a*,*n*) ∗ ↑(*l*≤*h* ∧ *h* ≤ *n* ∧ (∀*j*<*l*. *xs*!*j* < *p*) ∧ (*h*<*n* ⟶ *xs*!*h*≥*p*))
and *b* = λ(*l*,*h*). *l*<*h*
and *Q*=λ(*l*,*h*). *is-pfa c xs* (*a*,*n*) ∗ ↑(*l* ≤ *n* ∧ (∀*j*<*l*. *xs*!*j* < *p*) ∧ (*l*<*n* ⟶ *xs*!*l*≥*p*))
]
thm *R*

apply (*sep-auto decon*: *R simp*: *less-Suc-eq is-pfa-def*) []
subgoal for *l′ aa l′a aaa ba j*
proof −
assume *0*: *n* ≤ *length l′a*
assume *a*: *l′a* ! ((*aa* + *n*) *div 2*) < *p*
moreover assume *aa* < *n*
ultimately have *b*: ((*aa*+*n*)*div 2*) < *n*
by *linarith*
then have (*take n l′a*) ! ((*aa* + *n*) *div 2*) < *p*
using *a* **by** *auto*
moreover assume *sorted-less* (*take n l′a*)
ultimately have ⋀*j*. *j* < (*aa*+*n*)*div 2* ⟹ (*take n l′a*) ! *j* < (*take n l′a*) ! ((*aa* + *n*) *div 2*)
using
sorted-wrt-nth-less[**where** *?P*=(<) **and** *xs*=(*take n l′a*) **and** *?j*=((*aa* + *n*) *div 2*)]
a b 0 **by** *auto*
moreover fix *j* **assume** *j* < (*aa*+*n*) *div 2*
ultimately show *l′a* ! *j* < *p* **using** *0 b*
using ‹*take n l′a* ! ((*aa* + *n*) *div 2*) < *p*› *dual-order.strict-trans* **by** *auto*
qed
subgoal for *l′ aa b l′a aaa ba j*
proof −
assume *t0*: *n* ≤ *length l′a*
assume *t1*: *aa* < *b*
assume *a*: *l′a* ! ((*aa* + *b*) *div 2*) < *p*
moreover assume *b* ≤ *n*
ultimately have *b*: ((*aa*+*b*)*div 2*) < *n* **using** *t1*
by *linarith*
then have (*take n l′a*) ! ((*aa* + *b*) *div 2*) < *p*
using *a* **by** *auto*
moreover assume *sorted-less* (*take n l′a*)
ultimately have ⋀*j*. *j* < (*aa*+*b*)*div 2* ⟹ (*take n l′a*) ! *j* < (*take n l′a*) ! ((*aa*

```
+ b) div 2)
      using
        sorted-wrt-nth-less[where ?P=(<) and xs=(take n l'a) and ?j=((aa + b)
div 2)]
        a b t0 by auto
    moreover fix j assume j < (aa+b) div 2
    ultimately show l'a ! j < p using t0 b
      using ‹take n l'a ! ((aa + b) div 2) < p› dual-order.strict-trans by auto
  qed
     apply sep-auto
      apply (metis le-less nth-take)
     apply (metis le-less nth-take)
    apply sep-auto
  subgoal for l' aa l'a aaa ba j
  proof −
    assume t0: aa < n
    assume t1:  n ≤ length l'a
    assume t4: sorted-less (take n l'a)
    assume t5: j < (aa + n) div 2
    have (aa+n) div 2 < n using t0 by linarith
    then have (take n l'a) ! j < (take n l'a) ! ((aa + n) div 2)
      using t0 sorted-wrt-nth-less[where xs=take n l'a and ?j=((aa + n) div 2)]
        t1 t4 t5 by auto
    then show ?thesis
      using ‹(aa + n) div 2 < n› t5 by auto
  qed
  subgoal for l' aa b l'a aaa ba j
  proof −
    assume t0: aa < b
    assume t1:  n ≤ length l'a
    assume t3: b ≤ n
    assume t4: sorted-less (take n l'a)
    assume t5: j < (aa + b) div 2
    have (aa+b) div 2 < n using t3 t0 by linarith
    then have (take n l'a) ! j < (take n l'a) ! ((aa + b) div 2)
      using t0 sorted-wrt-nth-less[where xs=take n l'a and ?j=((aa + b) div 2)]
        t1 t4 t5 by auto
    then show ?thesis
      using ‹(aa + b) div 2 < n› t5 by auto
  qed
    apply (metis nth-take order-mono-setup.refl)
   apply sep-auto
  apply (sep-auto simp add: is-pfa-def)
  done
```

Then, using the same loop invariant, a binary split for B-tree-like arrays is derived in a straightforward manner.

```
definition bin-split :: ('a::heap × 'b::{heap,linorder}) pfarray ⇒ 'b ⇒ nat Heap
  where
```

```
   bin-split ≡ λ(a,n) p. do {
  (low′,high′) ← heap-WHILET
    (λ(low,high). return (low < high))
    (λ(low,high). let mid = ((low  + high) div 2) in
     do {
      (-,s) ← Array.nth a mid;
      if p < s then
         return (low, mid)
      else if p > s then
         return (mid+1, high)
      else return (mid,mid)
     })
    (0::nat,n);
  return low′
}
```

thm *nth-take*

lemma *nth-take-eq*: *take n ls = take n ls′ ⟹ i < n ⟹ ls!i = ls′!i*
  by (*metis nth-take*)

lemma *map-snd-sorted-less*: ⟦*sorted-less* (*map snd xs*); *i < j*; *j < length xs*⟧
      ⟹ *snd* (*xs* ! *i*) < *snd* (*xs* ! *j*)
 by (*metis* (*mono-tags*, *opaque-lifting*) *length-map less-trans nth-map sorted-wrt-iff-nth-less*)

lemma *map-snd-sorted-lesseq*: ⟦*sorted-less* (*map snd xs*); *i* ≤ *j*; *j < length xs*⟧
      ⟹ *snd* (*xs* ! *i*) ≤ *snd* (*xs* ! *j*)
  by (*metis eq-iff less-imp-le map-snd-sorted-less order.not-eq-order-implies-strict*)

lemma *bin-split-rule*:
sorted-less (*map snd xs*) ⟹
< *is-pfa c xs* (*a*,*n*)>
 bin-split (*a*,*n*) *p*
 <λ*l*. *is-pfa c xs* (*a*,*n*) ∗ ↑(*l* ≤ *n* ∧ (∀*j*<*l*. *snd*(*xs*!*j*) < *p*) ∧ (*l*<*n* ⟶ *snd*(*xs*!*l*)≥*p*))
>$_t$

  unfolding *bin-split-def*

  supply *R* = *heap-WHILET-rule′′*[**where**
      R = *measure* (λ(*l*,*h*). *h*−*l*)
      and *I* = λ(*l*,*h*). *is-pfa c xs* (*a*,*n*) ∗ ↑(*l*≤*h* ∧ *h* ≤ *n* ∧ (∀*j*<*l*. *snd* (*xs*!*j*) < *p*)
∧ (*h*<*n* ⟶ *snd* (*xs*!*h*)≥*p*))
      and *b* = λ(*l*,*h*). *l*<*h*
      and *Q*=λ(*l*,*h*). *is-pfa c xs* (*a*,*n*) ∗ ↑(*l* ≤ *n* ∧ (∀*j*<*l*. *snd* (*xs*!*j*) < *p*) ∧ (*l*<*n*
⟶ *snd* (*xs*!*l*)≥*p*))
      ]
  thm *R*

apply (*sep-auto decon*: *R simp*: *less-Suc-eq is-pfa-def*) []

apply(*auto dest*!: *sndI nth-take-eq*[*of n - - (- + -) div 2*])[]
apply(*auto dest*!: *sndI nth-take-eq*[*of n - - (- + -) div 2*])[]
apply (*sep-auto dest*!: *sndI*)
subgoal for *ls i ls′ - - j*
using *map-snd-sorted-lesseq*[*of take n ls′ j* (*i* + *n*) *div 2*]
less-mult-imp-div-less **apply**(*auto*)[]
done
subgoal for *ls i j ls′ - - j′*
using *map-snd-sorted-lesseq*[*of take n ls′ j′* (*i* + *j*) *div 2*]
less-mult-imp-div-less **apply**(*auto*)[]
done
apply *sep-auto*
subgoal for *ls i ls′ - - j*
using *map-snd-sorted-less*[*of take n ls′ j* (*i* + *n*) *div 2*]
less-mult-imp-div-less
apply(*auto*)[]
done
subgoal for *ls i j ls′ - - j′*
using *map-snd-sorted-less*[*of take n ls′ j′* (*i* + *j*) *div 2*]
less-mult-imp-div-less
apply(*auto*)[]
done
apply (*metis le-less nth-take-eq*)
apply *sep-auto*
apply (*sep-auto simp add*: *is-pfa-def*)
done

11.3 Refinement of an abstract split

We provide a certain abstract split function that is particularly easy to analyse. The idea of this function is due to Peter Lammich.

definition *abs-split xs x* = (*takeWhile* (λ(-,*s*). *s*<*x*) *xs*, *dropWhile* (λ(-,*s*). *s*<*x*) *xs*)

interpretation *btree-abs-search*: *split abs-split*
unfolding *abs-split-def sym*[*OF linear-split-alt*]
by *unfold-locales*

Any function that yields the heap rule we have obtained for bin_split and lin_split also refines this abstract split.

locale *imp-split-smeq* =
fixes *split-fun* :: (′*a*::{*heap*,*default*,*linorder*} *btnode ref option* × ′*a*) *array* × *nat* ⇒ ′*a* ⇒ *nat Heap*
assumes *split-rule*: *sorted-less* (*separators xs*) ⟹
<*is-pfa c xs* (*a*, *n*)>
split-fun (*a*, *n*) (*p*::′*a*)

$<\lambda r.\ is\text{-}pfa\ c\ xs\ (a,\ n)\ *$
$\uparrow (r \leq n \wedge$
$(\forall j{<}r.\ snd\ (xs\ !\ j) < p) \wedge$
$(r < n \longrightarrow p \leq snd\ (xs\ !\ r)))>_t$
begin

lemma *abs-split-full*: $\forall (\text{-},s) \in set\ xs.\ s < p \Longrightarrow abs\text{-}split\ xs\ p = (xs,[])$
by (*simp add*: *abs-split-def*)

lemma *abs-split-split*:
assumes $n < length\ xs$
and $(\forall (\text{-},s) \in set\ (take\ n\ xs).\ s < p)$
and $(case\ (xs!n)\ of\ (\text{-},s) \Rightarrow \neg(s < p))$
shows $abs\text{-}split\ xs\ p = (take\ n\ xs,\ drop\ n\ xs)$
using *assms* **apply** (*auto simp add*: *abs-split-def*)
apply (*metis* (*mono-tags*, *lifting*) *id-take-nth-drop old.prod.case takeWhile-eq-all-conv takeWhile-tail*)
by (*metis* (*no-types*, *lifting*) *Cons-nth-drop-Suc case-prod-conv dropWhile.simps*(*2*) *dropWhile-append2 id-take-nth-drop*)

lemma *split-rule-abs-split*:
shows
$sorted\text{-}less\ (separators\ ts) \Longrightarrow <$
$is\text{-}pfa\ c\ tsi\ (a,n)$
$*\ list\text{-}assn\ (A \times_a id\text{-}assn)\ ts\ tsi>$
$split\text{-}fun\ (a,n)\ p$
$<\lambda i.$
$is\text{-}pfa\ c\ tsi\ (a,n)$
$*\ list\text{-}assn\ (A \times_a id\text{-}assn)\ ts\ tsi$
$*\ \uparrow(split\text{-}relation\ ts\ (abs\text{-}split\ ts\ p)\ i)>_t$
apply(*rule hoare-triple-preI*)
apply (*sep-auto heap*: *split-rule dest*!: *mod-starD id-assn-list*
simp add: *list-assn-prod-map split-ismeq*)
apply(*auto simp add*: *is-pfa-def*)
proof −

fix $h\ l'$ **assume** *heap-init*:
$h \models a \mapsto_a l'$
$map\ snd\ ts = (map\ snd\ (take\ n\ l'))$
$n \leq length\ l'$

show *full-thm*: $\forall j{<}n.\ snd\ (l'\ !\ j) < p \Longrightarrow$
$split\text{-}relation\ ts\ (abs\text{-}split\ ts\ p)\ n$
proof −
assume *sm-list*: $\forall j{<}n.\ snd\ (l'\ !\ j) < p$

then have $\forall j < length\ (map\ snd\ (take\ n\ l'))$. $((map\ snd\ (take\ n\ l'))!j) < p$
by *simp*
then have $\forall j{<}length\ (map\ snd\ ts)$. $((map\ snd\ ts)!j) < p$
using *heap-init* **by** *simp*
then have $\forall (\text{-},s) \in set\ ts.\ s < p$
by (*metis case-prod-unfold in-set-conv-nth length-map nth-map*)
then have *abs-split ts p* = (*ts*, [])
using *abs-split-full*[*of ts p*] **by** *simp*
then show *split-relation ts* (*abs-split ts p*) *n*
using *split-relation-length*
by (*metis heap-init*(*2*) *heap-init*(*3*) *length-map length-take min.absorb2*)

qed
then show $\forall j{<}n.\ snd\ (l'\ !\ j) < p \Longrightarrow$
$p \leq snd\ (take\ n\ l'\ !\ n) \Longrightarrow$
split-relation ts (*abs-split ts p*) *n*
by *simp*

show *part-thm*: $\bigwedge x.\ x < n \Longrightarrow$
$\forall j{<}x.\ snd\ (l'\ !\ j) < p \Longrightarrow$
$p \leq snd\ (l'\ !\ x) \Longrightarrow$ *split-relation ts* (*abs-split ts p*) *x*
proof −
fix *x* **assume** *x-sm-len*: $x < n$
moreover assume *sm-list*: $\forall j{<}x.\ snd\ (l'\ !\ j) < p$
ultimately have $\forall j{<}x.\ ((map\ snd\ l')\ !\ j) < p$
using *heap-init*
by *auto*
then have $\forall j{<}x.\ ((map\ snd\ ts)!j) < p$
using *heap-init x-sm-len*
by *auto*
moreover have *x-sm-len-ts*: $x < n$
using *heap-init x-sm-len* **by** *auto*
ultimately have $\forall (\text{-},x) \in set\ (take\ x\ ts).\ x < p$
by (*auto simp add*: *in-set-conv-nth min.absorb2*)+
moreover assume $p \leq snd\ (l'\ !\ x)$
then have *case l'*!*x of* $(\text{-},s) \Rightarrow \neg(s < p)$
by (*simp add*: *case-prod-unfold*)
then have *case ts*!*x of* $(\text{-},s) \Rightarrow \neg(s < p)$
using *heap-init x-sm-len x-sm-len-ts*
by (*metis* (*mono-tags*, *lifting*) *case-prod-unfold length-map length-take min.absorb2 nth-take snd-map-help*(*2*))
ultimately have *abs-split ts p* = (*take x ts*, *drop x ts*)
using *x-sm-len-ts abs-split-split*[*of x ts p*] *heap-init*
by (*metis length-map length-take min.absorb2*)
then show *split-relation ts* (*abs-split ts p*) *x*
using *x-sm-len-ts*
by (*metis append-take-drop-id heap-init*(*2*) *heap-init*(*3*) *length-map length-take less-imp-le-nat min.absorb2 split-relation-alt*)
qed

qed

sublocale *imp-split abs-split split-fun*
apply(*unfold-locales*)
apply(*sep-auto heap*: *split-rule-abs-split*)
done

end

11.4 Obtaining executable code

In order to obtain fully defined functions, we need to plug our split function implementations into the locales we introduced previously.

interpretation *btree-imp-linear-split*: *imp-split-smeq lin-split*
apply *unfold-locales*
apply(*sep-auto heap*: *lin-split-rule*)
done

Obtaining actual code turns out to be slightly more difficult due to the use of locales. However, we successfully obtain the B-tree insertion and membership query with binary search splitting.

global-interpretation *btree-imp-binary-split*: *imp-split-smeq bin-split*
defines *btree-isin* = *btree-imp-binary-split.isin*
and *btree-ins* = *btree-imp-binary-split.ins*
and *btree-insert* = *btree-imp-binary-split.insert*
and *btree-del* = *btree-imp-binary-split.del*
and *btree-split-max* = *btree-imp-binary-split.split-max*
and *btree-delete* = *btree-imp-binary-split.delete*
and *btree-empty* = *btree-imp-binary-split.empty*
apply *unfold-locales*
apply(*sep-auto heap*: *bin-split-rule*)
done

declare *btree-imp-binary-split.ins.simps*[*code*]
declare *btree-imp-binary-split.isin.simps*[*code*]
declare *btree-imp-binary-split.del.simps*[*code*] *btree-imp-binary-split.split-max.simps*[*code*]

export-code *btree-empty btree-isin btree-insert btree-delete* **checking** *OCaml SML Scala*
export-code *btree-empty btree-isin btree-insert btree-delete* **in** *OCaml* **module-name** *BTree*
export-code *btree-empty btree-isin btree-insert btree-delete* **in** *SML* **module-name** *BTree*
export-code *btree-empty btree-isin btree-insert btree-delete* **in** *Scala* **module-name** *BTree*

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.