

Functional Data Structures

Tobias Nipkow

February 17, 2016

Abstract

A collection of verified functional data structures. The emphasis is on conciseness of algorithms and succinctness of proofs, more in the style of a textbook than a library of efficient algorithms.

Contents

1	Binary Tree	3
2	Three-Way Comparison	7
3	Lists Sorted wrt $<$	7
4	List Insertion and Deletion	9
5	Implementing Ordered Sets	12
6	Tree Implementation of Sets	14
7	Association List Update and Deletion	15
8	Implementing Ordered Maps	19
9	Unbalanced Tree as Map	21
10	Function <i>isin</i> for Tree2	22
11	AVL Tree Implementation of Sets	23
12	Function <i>lookup</i> for Tree2	33
13	AVL Tree Implementation of Maps	34
14	Red-Black Trees	35
15	Red-Black Tree Implementation of Sets	36

16 Red-Black Tree Implementation of Maps	40
17 2-3 Trees	42
18 2-3 Tree Implementation of Sets	43
19 2-3 Tree Implementation of Maps	52
20 2-3-4 Trees	55
21 2-3-4 Tree Implementation of Sets	56
22 2-3-4 Tree Implementation of Maps	69
23 1-2 Brother Tree Implementation of Sets	73
24 1-2 Brother Tree Implementation of Maps	84
25 AA Tree Implementation of Sets	89
26 AA Implementation of Maps	92
27 Splay Tree Implementation of Sets	93
28 Splay Tree Implementation of Maps	97
29 Bibliographic Notes	101

1 Binary Tree

```
theory Tree
imports Main
begin

datatype 'a tree =
  is_Leaf: Leaf ( $\langle \rangle$ ) |
  Node (left: 'a tree) (val: 'a) (right: 'a tree) (( $1 \langle \_ / \_ / \_ \rangle$ ))
  where
    left Leaf = Leaf
    | right Leaf = Leaf
datatype_compat tree
```

Can be seen as counting the number of leaves rather than nodes:

```
definition size1 :: 'a tree  $\Rightarrow$  nat where
  size1 t = size t + 1
```

```
lemma size1_simps[simp]:
  size1  $\langle \rangle$  = 1
  size1  $\langle l, x, r \rangle$  = size1 l + size1 r
by (simp_all add: size1_def)
```

```
lemma size_0_iff_Leaf: size t = 0  $\longleftrightarrow$  t = Leaf
by(cases t) auto
```

```
lemma neq_Leaf_iff: (t  $\neq \langle \rangle$ ) = ( $\exists l a r. t = \langle l, a, r \rangle$ )
by (cases t) auto
```

```
lemma finite_set_tree[simp]: finite(set_tree t)
by(induction t) auto
```

```
lemma size_map_tree[simp]: size (map_tree f t) = size t
by (induction t) auto
```

```
lemma size1_map_tree[simp]: size1 (map_tree f t) = size1 t
by (simp add: size1_def)
```

1.1 The Height

```
class height = fixes height :: 'a  $\Rightarrow$  nat
```

```
instantiation tree :: (type)height
begin
```

```

fun height_tree :: 'a tree => nat where
  height Leaf = 0 |
  height (Node t1 a t2) = max (height t1) (height t2) + 1

instance ..

end

lemma height_map_tree[simp]: height (map_tree f t) = height t
by (induction t) auto

lemma size1_height: size t + 1 ≤ 2 ^ height (t::'a tree)
proof(induction t)
  case (Node l a r)
  show ?case
  proof (cases height l ≤ height r)
    case True
    have size(Node l a r) + 1 = (size l + 1) + (size r + 1) by simp
    also have size l + 1 ≤ 2 ^ height l by(rule Node.IH(1))
    also have size r + 1 ≤ 2 ^ height r by(rule Node.IH(2))
    also have (2::nat) ^ height l ≤ 2 ^ height r using True by simp
    finally show ?thesis using True by (auto simp: max_def mult_2)
  next
  case False
  have size(Node l a r) + 1 = (size l + 1) + (size r + 1) by simp
  also have size l + 1 ≤ 2 ^ height l by(rule Node.IH(1))
  also have size r + 1 ≤ 2 ^ height r by(rule Node.IH(2))
  also have (2::nat) ^ height r ≤ 2 ^ height l using False by simp
  finally show ?thesis using False by (auto simp: max_def mult_2)
  qed
qed simp

```

1.2 The set of subtrees

```

fun subtrees :: 'a tree ⇒ 'a tree set where
  subtrees ⟨⟩ = {⟨⟩} |
  subtrees (⟨l, a, r⟩) = insert ⟨l, a, r⟩ (subtrees l ∪ subtrees r)

lemma set_treeE: a ∈ set_tree t ⇒ ∃ l r. ⟨l, a, r⟩ ∈ subtrees t
by (induction t)(auto)

lemma Node_notin_subtrees_if[simp]: a ∉ set_tree t ⇒ Node l a r ∉ sub-
trees t

```

by (*induction t*) *auto*

lemma *in_set_tree_if*: $\langle l, a, r \rangle \in \text{subtrees } t \implies a \in \text{set_tree } t$
by (*metis Node_notin_subtrees_if*)

1.3 List of entries

fun *preorder* :: 'a tree $\Rightarrow$ 'a list **where**
preorder $\langle \rangle = []$ |
preorder $\langle l, x, r \rangle = x \# \text{preorder } l @ \text{preorder } r$

fun *inorder* :: 'a tree $\Rightarrow$ 'a list **where**
inorder $\langle \rangle = []$ |
inorder $\langle l, x, r \rangle = \text{inorder } l @ [x] @ \text{inorder } r$

lemma *set_inorder[simp]*: $\text{set } (\text{inorder } t) = \text{set_tree } t$
by (*induction t*) *auto*

lemma *set_preorder[simp]*: $\text{set } (\text{preorder } t) = \text{set_tree } t$
by (*induction t*) *auto*

lemma *length_preorder[simp]*: $\text{length } (\text{preorder } t) = \text{size } t$
by (*induction t*) *auto*

lemma *length_inorder[simp]*: $\text{length } (\text{inorder } t) = \text{size } t$
by (*induction t*) *auto*

lemma *preorder_map*: $\text{preorder } (\text{map_tree } f \ t) = \text{map } f \ (\text{preorder } t)$
by (*induction t*) *auto*

lemma *inorder_map*: $\text{inorder } (\text{map_tree } f \ t) = \text{map } f \ (\text{inorder } t)$
by (*induction t*) *auto*

1.4 Binary Search Tree predicate

fun (**in** *linorder*) *bst* :: 'a tree $\Rightarrow$ bool **where**
bst $\langle \rangle \longleftrightarrow \text{True}$ |
bst $\langle l, a, r \rangle \longleftrightarrow \text{bst } l \wedge \text{bst } r \wedge (\forall x \in \text{set_tree } l. x < a) \wedge (\forall x \in \text{set_tree } r. a < x)$

In case there are duplicates:

fun (**in** *linorder*) *bst_eq* :: 'a tree $\Rightarrow$ bool **where**
bst_eq $\langle \rangle \longleftrightarrow \text{True}$ |
bst_eq $\langle l, a, r \rangle \longleftrightarrow$
bst_eq $l \wedge \text{bst_eq } r \wedge (\forall x \in \text{set_tree } l. x \leq a) \wedge (\forall x \in \text{set_tree } r. a \leq x)$

lemma (in *linorder*) *bst_eq_if_bst*: *bst t* $\implies$ *bst_eq t*
by (*induction t*) (*auto*)

lemma (in *linorder*) *bst_eq_imp_sorted*: *bst_eq t* $\implies$ *sorted (inorder t)*
apply (*induction t*)
apply (*simp*)
by (*fastforce simp: sorted_append sorted_Cons intro: less_imp_le less_trans*)

lemma (in *linorder*) *distinct_preorder_if_bst*: *bst t* $\implies$ *distinct (preorder t)*
apply (*induction t*)
apply *simp*
apply (*fastforce elim: order.asym*)
done

lemma (in *linorder*) *distinct_inorder_if_bst*: *bst t* $\implies$ *distinct (inorder t)*
apply (*induction t*)
apply *simp*
apply (*fastforce elim: order.asym*)
done

1.5 The heap predicate

fun *heap* :: '*a*::*linorder* tree $\Rightarrow$ bool **where**
heap Leaf = *True* |
heap (Node l m r) =
 (*heap l* $\wedge$ *heap r* $\wedge$ ($\forall x \in \text{set_tree } l \cup \text{set_tree } r. m \leq x$))

1.6 Function *mirror*

fun *mirror* :: '*a* tree $\Rightarrow$ '*a* tree **where**
mirror $\langle \rangle$ = *Leaf* |
mirror $\langle l, x, r \rangle$ = $\langle \text{mirror } r, x, \text{mirror } l \rangle$

lemma *mirror_Leaf[simp]*: *mirror t* = $\langle \rangle \longleftrightarrow t = \langle \rangle$
by (*induction t*) *simp_all*

lemma *size_mirror[simp]*: *size (mirror t)* = *size t*
by (*induction t*) *simp_all*

lemma *size1_mirror[simp]*: *size1 (mirror t)* = *size1 t*
by (*simp add: size1_def*)

lemma *height_mirror[simp]*: *height (mirror t)* = *height t*

by (*induction t*) *simp_all*

lemma *inorder_mirror*: *inorder*(*mirror t*) = *rev*(*inorder t*)

by (*induction t*) *simp_all*

lemma *map_mirror*: *map_tree f* (*mirror t*) = *mirror* (*map_tree f t*)

by (*induction t*) *simp_all*

lemma *mirror_mirror*[*simp*]: *mirror*(*mirror t*) = *t*

by (*induction t*) *simp_all*

end

2 Three-Way Comparison

theory *Cmp*

imports *Main*

begin

datatype *cmp* = *LT* | *EQ* | *GT*

class *cmp* = *linorder* +

fixes *cmp* :: 'a $\Rightarrow$ 'a $\Rightarrow$ *cmp*

assumes *LT*[*simp*]: *cmp x y* = *LT* $\longleftrightarrow$ $x < y$

assumes *EQ*[*simp*]: *cmp x y* = *EQ* $\longleftrightarrow$ $x = y$

assumes *GT*[*simp*]: *cmp x y* = *GT* $\longleftrightarrow$ $x > y$

lemma *case_cmp_if*[*simp*]: (*case c of EQ $\Rightarrow$ e | LT $\Rightarrow$ l | GT $\Rightarrow$ g*) =
(*if c = LT then l else if c = GT then g else e*)

by(*simp split: cmp.split*)

end

3 Lists Sorted wrt <

theory *Sorted_Less*

imports *Less_False*

begin

hide_const *sorted*

Is a list sorted without duplicates, i.e., wrt <? Could go into theory List under a name like *sorted_less*.

```

fun sorted :: 'a::linorder list  $\Rightarrow$  bool where
  sorted [] = True |
  sorted [x] = True |
  sorted (x#y#zs) = (x < y  $\wedge$  sorted(y#zs))

```

```

lemma sorted_Cons_iff:
  sorted(x # xs) = (sorted xs  $\wedge$  ( $\forall y \in \text{set } xs. x < y$ ))
by(induction xs rule: sorted.induct) auto

```

```

lemma sorted_snoc_iff:
  sorted(xs @ [x]) = (sorted xs  $\wedge$  ( $\forall y \in \text{set } xs. y < x$ ))
by(induction xs rule: sorted.induct) auto

```

```

lemma sorted_cons: sorted (x#xs)  $\Longrightarrow$  sorted xs
by(simp add: sorted_Cons_iff)

```

```

lemma sorted_cons': ASSUMPTION (sorted (x#xs))  $\Longrightarrow$  sorted xs
by(rule ASSUMPTION_D [THEN sorted_cons])

```

```

lemma sorted_snoc: sorted (xs @ [y])  $\Longrightarrow$  sorted xs
by(simp add: sorted_snoc_iff)

```

```

lemma sorted_snoc': ASSUMPTION (sorted (xs @ [y]))  $\Longrightarrow$  sorted xs
by(rule ASSUMPTION_D [THEN sorted_snoc])

```

```

lemma sorted_mid_iff:
  sorted(xs @ y # ys) = (sorted(xs @ [y])  $\wedge$  sorted(y # ys))
by(induction xs rule: sorted.induct) auto

```

```

lemma sorted_mid_iff2:
  sorted(x # xs @ y # ys) =
    (sorted(x # xs)  $\wedge$  x < y  $\wedge$  sorted(xs @ [y])  $\wedge$  sorted(y # ys))
by(induction xs rule: sorted.induct) auto

```

```

lemma sorted_mid_iff': NO_MATCH [] ys  $\Longrightarrow$ 
  sorted(xs @ y # ys) = (sorted(xs @ [y])  $\wedge$  sorted(y # ys))
by(rule sorted_mid_iff)

```

```

lemmas sorted_lems = sorted_mid_iff' sorted_mid_iff2 sorted_cons' sorted_snoc'

```

Splay trees need two additional *sorted* lemmas:

```

lemma sorted_snoc_le:
  ASSUMPTION(sorted(xs @ [x]))  $\Longrightarrow$  x  $\leq$  y  $\Longrightarrow$  sorted (xs @ [y])
by (auto simp add: Sorted_Less.sorted_snoc_iff ASSUMPTION_def)

```

```

lemma sorted_Cons_le:
  ASSUMPTION(sorted( $x \# xs$ )  $\implies y \leq x \implies \text{sorted } (y \# xs)$ )
by (auto simp add: Sorted_Less.sorted_Cons_iff ASSUMPTION_def)

end

```

4 List Insertion and Deletion

```

theory List_Ins_Del
imports Sorted_Less
begin

```

4.1 Elements in a list

```

fun elems :: 'a list  $\Rightarrow$  'a set where
  elems [] = {} |
  elems (x#xs) = Set.insert x (elems xs)

lemma elems_app: elems (xs @ ys) = (elems xs  $\cup$  elems ys)
by (induction xs) auto

lemma elems_eq_set: elems xs = set xs
by (induction xs) auto

lemma sorted_Cons_iff:
  sorted( $x \# xs$ ) = (sorted xs  $\wedge (\forall y \in \text{elems } xs. x < y)$ )
by(simp add: elems_eq_set Sorted_Less.sorted_Cons_iff)

lemma sorted_snoc_iff:
  sorted(xs @ [x]) = (sorted xs  $\wedge (\forall y \in \text{elems } xs. y < x)$ )
by(simp add: elems_eq_set Sorted_Less.sorted_snoc_iff)

```

The above two rules introduce quantifiers. It turns out that in practice this is not a problem because of the simplicity of the "isin" functions that implement *elems*. Nevertheless it is possible to avoid the quantifiers with the help of some rewrite rules:

```

lemma sorted_ConsD: sorted (y # xs)  $\implies x \leq y \implies x \notin \text{elems } xs$ 
by (auto simp: sorted_Cons_iff)

lemma sorted_snocD: sorted (xs @ [y])  $\implies y \leq x \implies x \notin \text{elems } xs$ 
by (auto simp: sorted_snoc_iff)

lemmas elems_simps = sorted_lems elems_app

```

lemmas *elems_simps1* = *elems_simps sorted_Cons_iff sorted_snoc_iff*
lemmas *elems_simps2* = *elems_simps sorted_ConsD sorted_snocD*

4.2 Inserting into an ordered list without duplicates:

fun *ins_list* :: 'a::linorder $\Rightarrow$ 'a list $\Rightarrow$ 'a list **where**
ins_list x [] = [x] |
ins_list x (a#xs) =
 (if x < a then x#a#xs else if x=a then a#xs else a # *ins_list* x xs)

lemma *set.ins_list*: *elems (ins_list x xs) = insert x (elems xs)*
by(*induction xs*) *auto*

lemma *distinct_if_sorted*: *sorted xs $\implies$ distinct xs*
apply(*induction xs rule: sorted.induct*)
apply *auto*
by (*metis in_set_conv_decomp_first less_imp_not_less sorted_mid_iff2*)

lemma *sorted_ins_list*: *sorted xs $\implies$ sorted (ins_list x xs)*
by(*induction xs rule: sorted.induct*) *auto*

lemma *ins_list_sorted*: *sorted (xs @ [a]) $\implies$*
ins_list x (xs @ a # ys) =
(if x < a then ins_list x xs @ (a#ys) else xs @ ins_list x (a#ys))
by(*induction xs*) (*auto simp: sorted_simps*)

In principle, *sorted (?xs @ [a]) $\implies$ ins_list ?x (?xs @ ?a # ?ys) = (if ?x < ?a then ins_list ?x ?xs @ ?a # ?ys else ?xs @ ins_list ?x (?a # ?ys))* suffices, but the following two corollaries speed up proofs.

corollary *ins_list_sorted1*: *sorted (xs @ [a]) $\implies$ a $\leq$ x $\implies$*
ins_list x (xs @ a # ys) = xs @ ins_list x (a#ys)
by(*auto simp add: ins_list_sorted*)

corollary *ins_list_sorted2*: *sorted (xs @ [a]) $\implies$ x < a $\implies$*
ins_list x (xs @ a # ys) = ins_list x xs @ (a#ys)
by(*auto simp: ins_list_sorted*)

lemmas *ins_list_simps* = *sorted_simps ins_list_sorted1 ins_list_sorted2*

Splay trees need two additional *ins_list* lemmas:

lemma *ins_list_Cons*: *sorted (x # xs) $\implies$ ins_list x xs = x # xs*
by (*induction xs*) *auto*

lemma *ins_list_snoc*: *sorted (xs @ [x]) $\implies$ ins_list x xs = xs @ [x]*
by(*induction xs*) (*auto simp add: sorted_mid_iff2*)

4.3 Delete one occurrence of an element from a list:

```
fun del_list :: 'a  $\Rightarrow$  'a list  $\Rightarrow$  'a list where
  del_list x [] = [] |
  del_list x (a#xs) = (if x=a then xs else a # del_list x xs)
```

```
lemma del_list_idem:  $x \notin \text{elems } xs \implies \text{del\_list } x \ xs = xs$ 
by (induct xs) simp_all
```

```
lemma elems_del_list_eq:
  distinct xs  $\implies \text{elems } (\text{del\_list } x \ xs) = \text{elems } xs - \{x\}$ 
apply (induct xs)
apply simp
apply (simp add: elems_eq_set)
apply blast
done
```

```
lemma sorted_del_list: sorted xs  $\implies \text{sorted } (\text{del\_list } x \ xs)$ 
apply (induction xs rule: sorted.induct)
apply auto
by (meson order.strict_trans sorted_Cons_iff)
```

```
lemma del_list_sorted: sorted (xs @ a # ys)  $\implies$ 
  del_list x (xs @ a # ys) = (if x < a then del_list x xs @ a # ys else xs @
  del_list x (a # ys))
by (induction xs)
  (fastforce simp: sorted_lems sorted_Cons_iff elems_eq_set intro!: del_list_idem)+
```

In principle, $\text{sorted } (?xs @ ?a \# ?ys) \implies \text{del_list } ?x (?xs @ ?a \# ?ys) = (\text{if } ?x < ?a \text{ then } \text{del_list } ?x ?xs @ ?a \# ?ys \text{ else } ?xs @ \text{del_list } ?x (?a \# ?ys))$ suffices, but the following corollaries speed up proofs.

```
corollary del_list_sorted1: sorted (xs @ a # ys)  $\implies a \leq x \implies$ 
  del_list x (xs @ a # ys) = xs @ del_list x (a # ys)
by (auto simp: del_list_sorted)
```

```
corollary del_list_sorted2: sorted (xs @ a # ys)  $\implies x < a \implies$ 
  del_list x (xs @ a # ys) = del_list x xs @ a # ys
by (auto simp: del_list_sorted)
```

```
corollary del_list_sorted3:
  sorted (xs @ a # ys @ b # zs)  $\implies x < b \implies$ 
  del_list x (xs @ a # ys @ b # zs) = del_list x (xs @ a # ys) @ b # zs
by (auto simp: del_list_sorted sorted_lems)
```

corollary *del_list_sorted4*:

$sorted\ (xs\ @\ a\ \# \ ys\ @\ b\ \# \ zs\ @\ c\ \# \ us) \implies x < c \implies$
 $del_list\ x\ (xs\ @\ a\ \# \ ys\ @\ b\ \# \ zs\ @\ c\ \# \ us) = del_list\ x\ (xs\ @\ a\ \# \ ys\ @\ b\ \# \ zs)\ @\ c\ \# \ us$
by (*auto simp: del_list_sorted sorted_lems*)

corollary *del_list_sorted5*:

$sorted\ (xs\ @\ a\ \# \ ys\ @\ b\ \# \ zs\ @\ c\ \# \ us\ @\ d\ \# \ vs) \implies x < d \implies$
 $del_list\ x\ (xs\ @\ a\ \# \ ys\ @\ b\ \# \ zs\ @\ c\ \# \ us\ @\ d\ \# \ vs) =$
 $del_list\ x\ (xs\ @\ a\ \# \ ys\ @\ b\ \# \ zs\ @\ c\ \# \ us)\ @\ d\ \# \ vs$
by (*auto simp: del_list_sorted sorted_lems*)

lemmas *del_list_simps = sorted_lems*

del_list_sorted1
del_list_sorted2
del_list_sorted3
del_list_sorted4
del_list_sorted5

Splay trees need two additional *del_list* lemmas:

lemma *del_list_notin_Cons*: $sorted\ (x\ \# \ xs) \implies del_list\ x\ xs = xs$
by(*induction xs*)(*auto simp: sorted_Cons_iff*)

lemma *del_list_sorted_app*:

$sorted(xs\ @\ [x]) \implies del_list\ x\ (xs\ @\ ys) = xs\ @\ del_list\ x\ ys$
by (*induction xs*) (*auto simp: sorted_mid_iff2*)

end

5 Implementing Ordered Sets

theory *Set_by_Ordered*

imports *List_Ins_Del*

begin

locale *Set* =

fixes *empty* :: 's

fixes *insert* :: 'a $\Rightarrow$'s $\Rightarrow$'s

fixes *delete* :: 'a $\Rightarrow$'s $\Rightarrow$'s

fixes *isin* :: 's $\Rightarrow$ 'a $\Rightarrow$ bool

fixes *set* :: 's $\Rightarrow$ 'a set

fixes *invar* :: 's $\Rightarrow$ bool

assumes *set_empty*: $set\ empty = \{\}$

assumes *set_isin*: $invar\ s \implies isin\ s\ x = (x \in set\ s)$

```

assumes set_insert:  $\text{invar } s \implies \text{set}(\text{insert } x \ s) = \text{Set.insert } x \ (\text{set } s)$ 
assumes set_delete:  $\text{invar } s \implies \text{set}(\text{delete } x \ s) = \text{set } s - \{x\}$ 
assumes invar_empty:  $\text{invar empty}$ 
assumes invar_insert:  $\text{invar } s \implies \text{invar}(\text{insert } x \ s)$ 
assumes invar_delete:  $\text{invar } s \implies \text{invar}(\text{delete } x \ s)$ 

locale Set_by_Ordered =
fixes empty :: 't
fixes insert :: 'a::linorder  $\Rightarrow$  't  $\Rightarrow$  't
fixes delete :: 'a  $\Rightarrow$  't  $\Rightarrow$  't
fixes isin :: 't  $\Rightarrow$  'a  $\Rightarrow$  bool
fixes inorder :: 't  $\Rightarrow$  'a list
fixes inv :: 't  $\Rightarrow$  bool
assumes empty:  $\text{inorder empty} = []$ 
assumes isin:  $\text{inv } t \wedge \text{sorted}(\text{inorder } t) \implies$ 
   $\text{isin } t \ x = (x \in \text{elems } (\text{inorder } t))$ 
assumes insert:  $\text{inv } t \wedge \text{sorted}(\text{inorder } t) \implies$ 
   $\text{inorder}(\text{insert } x \ t) = \text{ins\_list } x \ (\text{inorder } t)$ 
assumes delete:  $\text{inv } t \wedge \text{sorted}(\text{inorder } t) \implies$ 
   $\text{inorder}(\text{delete } x \ t) = \text{del\_list } x \ (\text{inorder } t)$ 
assumes inv_empty:  $\text{inv empty}$ 
assumes inv_insert:  $\text{inv } t \wedge \text{sorted}(\text{inorder } t) \implies \text{inv}(\text{insert } x \ t)$ 
assumes inv_delete:  $\text{inv } t \wedge \text{sorted}(\text{inorder } t) \implies \text{inv}(\text{delete } x \ t)$ 
begin

sublocale Set
  empty insert delete isin elems o inorder  $\lambda t. \text{inv } t \wedge \text{sorted}(\text{inorder } t)$ 
proof(standard, goal_cases)
  case 1 show ?case by (auto simp: empty)
next
  case 2 thus ?case by(simp add: isin)
next
  case 3 thus ?case by(simp add: insert set_ins_list)
next
  case (4 s x) thus ?case
    using delete[OF 4, of x] by (auto simp: distinct_if_sorted elems_del_list_eq)
next
  case 5 thus ?case by(simp add: empty inv_empty)
next
  case 6 thus ?case by(simp add: insert inv_insert sorted_ins_list)
next
  case 7 thus ?case by (auto simp: delete inv_delete sorted_del_list)
qed

```

end

end

6 Tree Implementation of Sets

theory *Tree_Set*

imports

~~/src/HOL/Library/Tree

Cmp

Set_by_Ordered

begin

fun *isin* :: 'a::cmp tree $\Rightarrow$ 'a $\Rightarrow$ bool **where**

isin Leaf *x* = False |

isin (Node *l a r*) *x* =

(case *cmp* *x a* of

LT $\Rightarrow$ *isin* *l x* |

EQ $\Rightarrow$ True |

GT $\Rightarrow$ *isin* *r x*)

hide_const (open) *insert*

fun *insert* :: 'a::cmp $\Rightarrow$ 'a tree $\Rightarrow$ 'a tree **where**

insert *x* Leaf = Node Leaf *x* Leaf |

insert *x* (Node *l a r*) =

(case *cmp* *x a* of

LT $\Rightarrow$ Node (*insert* *x l*) *a r* |

EQ $\Rightarrow$ Node *l a r* |

GT $\Rightarrow$ Node *l a* (*insert* *x r*))

fun *del_min* :: 'a tree $\Rightarrow$ 'a * 'a tree **where**

del_min (Node *l a r*) =

(if *l* = Leaf then (*a,r*) else let (*x,l'*) = *del_min* *l* in (*x*, Node *l' a r*))

fun *delete* :: 'a::cmp $\Rightarrow$ 'a tree $\Rightarrow$ 'a tree **where**

delete *x* Leaf = Leaf |

delete *x* (Node *l a r*) =

(case *cmp* *x a* of

LT $\Rightarrow$ Node (*delete* *x l*) *a r* |

GT $\Rightarrow$ Node *l a* (*delete* *x r*) |

EQ $\Rightarrow$ if *r* = Leaf then *l* else let (*a',r'*) = *del_min* *r* in Node *l a' r'*)

6.1 Functional Correctness Proofs

lemma *sorted*(*inorder* *t*) $\implies$ *isin* *t* *x* = (*x* $\in$ *elems* (*inorder* *t*))
by (*induction* *t*) (*auto simp*: *elems_simps1*)

lemma *isin_set*: *sorted*(*inorder* *t*) $\implies$ *isin* *t* *x* = (*x* $\in$ *elems* (*inorder* *t*))
by (*induction* *t*) (*auto simp*: *elems_simps2*)

lemma *inorder_insert*:
sorted(*inorder* *t*) $\implies$ *inorder*(*insert* *x* *t*) = *ins_list* *x* (*inorder* *t*)
by(*induction* *t*) (*auto simp*: *ins_list_simps*)

lemma *del_minD*:
del_min *t* = (*x*, *t'*) $\implies$ *t* $\neq$ *Leaf* $\implies$ *x* $\#$ *inorder* *t'* = *inorder* *t*
by(*induction* *t* *arbitrary*: *t'* *rule*: *del_min.induct*)
(*auto simp*: *sorted_lems* *split*: *prod.splits* *if_splits*)

lemma *inorder_delete*:
sorted(*inorder* *t*) $\implies$ *inorder*(*delete* *x* *t*) = *del_list* *x* (*inorder* *t*)
by(*induction* *t*) (*auto simp*: *del_list_simps* *del_minD* *split*: *prod.splits*)

interpretation *Set.by_Ordered*
where *empty* = *Leaf* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* =
delete
and *inorder* = *inorder* **and** *inv* = $\lambda_.$ *True*
proof (*standard*, *goal_cases*)
 case 1 **show** ?*case* **by** *simp*
next
 case 2 **thus** ?*case* **by**(*simp add*: *isin_set*)
next
 case 3 **thus** ?*case* **by**(*simp add*: *inorder_insert*)
next
 case 4 **thus** ?*case* **by**(*simp add*: *inorder_delete*)
qed (*rule* *TrueI*)
end

7 Association List Update and Deletion

theory *AList_Upd_Del*
imports *Sorted_Less*
begin

abbreviation *sorted1* $ps \equiv sorted(map\ fst\ ps)$

Define own *map_of* function to avoid pulling in an unknown amount of lemmas implicitly (via the simpset).

hide_const (open) *map_of*

fun *map_of* :: ('a*'b)list $\Rightarrow$ 'a $\Rightarrow$ 'b option **where**
map_of [] = (λx . None) |
map_of ((a,b)#ps) = (λx . if $x=a$ then Some b else *map_of* ps x)

Updating an association list:

fun *upd_list* :: 'a::linorder $\Rightarrow$ 'b $\Rightarrow$ ('a*'b) list $\Rightarrow$ ('a*'b) list **where**
upd_list x y [] = [(x,y)] |
upd_list x y ((a,b)#ps) =
 (if $x < a$ then (x,y)#(a,b)#ps else
 if $x = a$ then (x,y)#ps else (a,b) # *upd_list* x y ps)

fun *del_list* :: 'a::linorder $\Rightarrow$ ('a*'b)list $\Rightarrow$ ('a*'b)list **where**
del_list x [] = [] |
del_list x ((a,b)#ps) = (if $x = a$ then ps else (a,b) # *del_list* x ps)

7.1 Lemmas for *map_of*

lemma *map_of_ins_list*: *map_of* (*upd_list* x y ps) = (*map_of* ps)(x := Some y)
by(*induction* ps) *auto*

lemma *map_of_append*: *map_of* (ps @ qs) x =
 (case *map_of* ps x of None $\Rightarrow$ *map_of* qs x | Some y $\Rightarrow$ Some y)
by(*induction* ps)(*auto*)

lemma *map_of_None*: *sorted* (x # *map fst* ps) $\implies$ *map_of* ps x = None
by (*induction* ps) (*auto simp: sorted_lems sorted_Cons_iff*)

lemma *map_of_None2*: *sorted* (*map fst* ps @ [x]) $\implies$ *map_of* ps x = None
by (*induction* ps) (*auto simp: sorted_lems*)

lemma *map_of_del_list*: *sorted1* ps $\implies$
map_of(*del_list* x ps) = (*map_of* ps)(x := None)
by(*induction* ps) (*auto simp: map_of_None sorted_lems fun_eq_iff*)

lemma *map_of_sorted_Cons*: *sorted* (a # *map fst* ps) $\implies$ $x < a \implies$
map_of ps x = None

by (*meson less_trans map_of_None sorted_Cons_iff*)

lemma *map_of_sorted_snoc*: *sorted (map fst ps @ [a]) $\implies a \leq x \implies$*
map_of ps x = None

by (*meson le_less_trans map_of_None2 not_less sorted_snoc_iff*)

lemmas *map_of_sorteds = map_of_sorted_Cons map_of_sorted_snoc*

lemmas *map_of_simps = sorted_lems map_of_append map_of_sorteds*

7.2 Lemmas for *upd_list*

lemma *sorted_upd_list*: *sorted1 ps $\implies$ sorted1 (upd_list x y ps)*

apply(*induction ps*)

apply *simp*

apply(*case_tac ps*)

apply *auto*

done

lemma *upd_list_sorted*: *sorted1 (ps @ [(a,b)]) $\implies$*

upd_list x y (ps @ (a,b) # qs) =
(if x < a then upd_list x y ps @ (a,b) # qs
else ps @ upd_list x y ((a,b) # qs))

by(*induction ps*) (*auto simp: sorted_lems*)

In principle, *sorted1 (?ps @ [(?a, ?b)]) $\implies$ upd_list ?x ?y (?ps @ (?a, ?b) # ?qs) = (if ?x < ?a then upd_list ?x ?y ?ps @ (?a, ?b) # ?qs else ?ps @ upd_list ?x ?y ((?a, ?b) # ?qs))* suffices, but the following two corollaries speed up proofs.

corollary *upd_list_sorted1*: $\llbracket \text{sorted (map fst ps @ [a]); } x < a \rrbracket \implies$

upd_list x y (ps @ (a,b) # qs) = upd_list x y ps @ (a,b) # qs

by (*auto simp: upd_list_sorted*)

corollary *upd_list_sorted2*: $\llbracket \text{sorted (map fst ps @ [a]); } a \leq x \rrbracket \implies$

upd_list x y (ps @ (a,b) # qs) = ps @ upd_list x y ((a,b) # qs)

by (*auto simp: upd_list_sorted*)

lemmas *upd_list_simps = sorted_lems upd_list_sorted1 upd_list_sorted2*

Splay trees need two additional *upd_list* lemmas:

lemma *upd_list_Cons*:

sorted1 ((x,y) # xs) $\implies$ upd_list x y xs = (x,y) # xs

by (*induction xs*) *auto*

lemma *upd_list_snoc*:

$sorted1\ (xs\ @\ [(x,y)]) \implies upd_list\ x\ y\ xs = xs\ @\ [(x,y)]$
by(*induction xs*) (*auto simp add: sorted_mid_iff2*)

7.3 Lemmas for *del_list*

lemma *sorted_del_list*: $sorted1\ ps \implies sorted1\ (del_list\ x\ ps)$
apply(*induction ps*)
apply *simp*
apply(*case_tac ps*)
apply *auto*
by (*meson order.strict_trans sorted_Cons_iff*)

lemma *del_list_idem*: $x \notin set(map\ fst\ xs) \implies del_list\ x\ xs = xs$
by (*induct xs*) *auto*

lemma *del_list_sorted*: $sorted1\ (ps\ @\ (a,b) \# qs) \implies$
 $del_list\ x\ (ps\ @\ (a,b) \# qs) =$
 $(if\ x < a\ then\ del_list\ x\ ps\ @\ (a,b) \# qs$
 $else\ ps\ @\ del_list\ x\ ((a,b) \# qs))$
by(*induction ps*)
(fastforce simp: sorted_lems sorted_Cons_iff intro!: del_list_idem)+

In principle, $sorted1\ (?ps\ @\ (?a, ?b) \# ?qs) \implies del_list\ ?x\ (?ps\ @\ (?a, ?b) \# ?qs) = (if\ ?x < ?a\ then\ del_list\ ?x\ ?ps\ @\ (?a, ?b) \# ?qs\ else\ ?ps\ @\ del_list\ ?x\ ((?a, ?b) \# ?qs))$ suffices, but the following corollaries speed up proofs.

corollary *del_list_sorted1*: $sorted1\ (xs\ @\ (a,b) \# ys) \implies a \leq x \implies$
 $del_list\ x\ (xs\ @\ (a,b) \# ys) = xs\ @\ del_list\ x\ ((a,b) \# ys)$
by (*auto simp: del_list_sorted*)

lemma *del_list_sorted2*: $sorted1\ (xs\ @\ (a,b) \# ys) \implies x < a \implies$
 $del_list\ x\ (xs\ @\ (a,b) \# ys) = del_list\ x\ xs\ @\ (a,b) \# ys$
by (*auto simp: del_list_sorted*)

lemma *del_list_sorted3*:
 $sorted1\ (xs\ @\ (a,a') \# ys\ @\ (b,b') \# zs) \implies x < b \implies$
 $del_list\ x\ (xs\ @\ (a,a') \# ys\ @\ (b,b') \# zs) = del_list\ x\ (xs\ @\ (a,a') \# ys)$
 $@\ (b,b') \# zs$
by (*auto simp: del_list_sorted sorted_lems*)

lemma *del_list_sorted4*:
 $sorted1\ (xs\ @\ (a,a') \# ys\ @\ (b,b') \# zs\ @\ (c,c') \# us) \implies x < c \implies$
 $del_list\ x\ (xs\ @\ (a,a') \# ys\ @\ (b,b') \# zs\ @\ (c,c') \# us) = del_list\ x\ (xs$
 $@\ (a,a') \# ys\ @\ (b,b') \# zs)\ @\ (c,c') \# us$

by (*auto simp: del_list_sorted sorted_lems*)

lemma *del_list_sorted5*:

sorted1 (*xs @ (a,a') # ys @ (b,b') # zs @ (c,c') # us @ (d,d') # vs*)
 $\implies x < d \implies$
del_list *x* (*xs @ (a,a') # ys @ (b,b') # zs @ (c,c') # us @ (d,d') # vs*)
 $=$
del_list *x* (*xs @ (a,a') # ys @ (b,b') # zs @ (c,c') # us*) @ (*d,d'*) # *vs*
by (*auto simp: del_list_sorted sorted_lems*)

lemmas *del_list_simps = sorted_lems*

del_list_sorted1

del_list_sorted2

del_list_sorted3

del_list_sorted4

del_list_sorted5

Splay trees need two additional *del_list* lemmas:

lemma *del_list_notin_Cons*: *sorted* (*x # map fst xs*) $\implies$ *del_list* *x xs* = *xs*
by(*induction xs*)(*auto simp: sorted_Cons_iff*)

lemma *del_list_sorted_app*:

sorted(*map fst xs @ [x]*) $\implies$ *del_list* *x (xs @ ys)* = *xs @ del_list x ys*
by (*induction xs*) (*auto simp: sorted_mid_iff2*)

end

8 Implementing Ordered Maps

theory *Map_by_Ordered*

imports *AList_Upd_Del*

begin

locale *Map* =

fixes *empty* :: 'm

fixes *update* :: 'a $\Rightarrow$ 'b $\Rightarrow$ 'm $\Rightarrow$ 'm

fixes *delete* :: 'a $\Rightarrow$ 'm $\Rightarrow$ 'm

fixes *lookup* :: 'm $\Rightarrow$ 'a $\Rightarrow$ 'b option

fixes *invar* :: 'm $\Rightarrow$ bool

assumes *map_empty*: *lookup empty* = (λ _. *None*)

and *map_update*: *invar m* $\implies$ *lookup*(*update a b m*) = (*lookup m*)(*a* := *Some b*)

and *map_delete*: *invar m* $\implies$ *lookup*(*delete a m*) = (*lookup m*)(*a* := *None*)

and *invar_empty*: *invar empty*

```

and invar_update: invar m  $\implies$  invar(update a b m)
and invar_delete: invar m  $\implies$  invar(delete a m)

locale Map_by_Ordered =
fixes empty :: 't
fixes update :: 'a::linorder  $\Rightarrow$  'b  $\Rightarrow$  't  $\Rightarrow$  't
fixes delete :: 'a  $\Rightarrow$  't  $\Rightarrow$  't
fixes lookup :: 't  $\Rightarrow$  'a  $\Rightarrow$  'b option
fixes inorder :: 't  $\Rightarrow$  ('a * 'b) list
fixes inv :: 't  $\Rightarrow$  bool
assumes empty: inorder empty = []
and lookup: inv t  $\wedge$  sorted1 (inorder t)  $\implies$ 
  lookup t a = map_of (inorder t) a
and update: inv t  $\wedge$  sorted1 (inorder t)  $\implies$ 
  inorder(update a b t) = upd_list a b (inorder t)
and delete: inv t  $\wedge$  sorted1 (inorder t)  $\implies$ 
  inorder(delete a t) = del_list a (inorder t)
and inv_empty: inv empty
and inv_update: inv t  $\wedge$  sorted1 (inorder t)  $\implies$  inv(update a b t)
and inv_delete: inv t  $\wedge$  sorted1 (inorder t)  $\implies$  inv(delete a t)
begin

sublocale Map
  empty update delete lookup  $\lambda t. inv\ t \wedge sorted1\ (inorder\ t)$ 
proof(standard, goal_cases)
  case 1 show ?case by (auto simp: lookup empty inv_empty)
next
  case 2 thus ?case
    by(simp add: fun_eq_iff update inv_update map_of_ins_list lookup sorted_upd_list)
next
  case 3 thus ?case
    by(simp add: fun_eq_iff delete inv_delete map_of_del_list lookup sorted_del_list)
next
  case 4 thus ?case by(simp add: empty inv_empty)
next
  case 5 thus ?case by(simp add: update inv_update sorted_upd_list)
next
  case 6 thus ?case by (auto simp: delete inv_delete sorted_del_list)
qed

end

end

```

9 Unbalanced Tree as Map

theory *Tree_Map*

imports

Tree_Set

Map_by_Ordered

begin

fun *lookup* :: ('a::cmp*'b) tree $\Rightarrow$ 'a $\Rightarrow$ 'b option **where**

lookup Leaf x = None |

lookup (Node l (a,b) r) x =

(case cmp x a of LT $\Rightarrow$ *lookup* l x | GT $\Rightarrow$ *lookup* r x | EQ $\Rightarrow$ Some b)

fun *update* :: 'a::cmp $\Rightarrow$ 'b $\Rightarrow$ ('a*'b) tree $\Rightarrow$ ('a*'b) tree **where**

update x y Leaf = Node Leaf (x,y) Leaf |

update x y (Node l (a,b) r) = (case cmp x a of

LT $\Rightarrow$ Node (*update* x y l) (a,b) r |

EQ $\Rightarrow$ Node l (x,y) r |

GT $\Rightarrow$ Node l (a,b) (*update* x y r))

fun *delete* :: 'a::cmp $\Rightarrow$ ('a*'b) tree $\Rightarrow$ ('a*'b) tree **where**

delete x Leaf = Leaf |

delete x (Node l (a,b) r) = (case cmp x a of

LT $\Rightarrow$ Node (*delete* x l) (a,b) r |

GT $\Rightarrow$ Node l (a,b) (*delete* x r) |

EQ $\Rightarrow$ if r = Leaf then l else let (ab',r') = del_min r in Node l ab' r')

9.1 Functional Correctness Proofs

lemma *lookup_map_of*:

sorted1(*inorder* t) $\Longrightarrow$ *lookup* t x = *map_of* (*inorder* t) x

by (*induction* t) (*auto simp: map_of_simps split: option.split*)

lemma *inorder_update*:

sorted1(*inorder* t) $\Longrightarrow$ *inorder*(*update* a b t) = *upd_list* a b (*inorder* t)

by(*induction* t) (*auto simp: upd_list_simps*)

lemma *inorder_delete*:

sorted1(*inorder* t) $\Longrightarrow$ *inorder*(*delete* x t) = *del_list* x (*inorder* t)

by(*induction* t) (*auto simp: del_list_simps del_minD split: prod.splits*)

interpretation *Map_by_Ordered*

where *empty* = Leaf **and** *lookup* = *lookup* **and** *update* = *update* **and** *delete* = *delete*

```

and inorder = inorder and inv =  $\lambda\_.$  True
proof (standard, goal_cases)
  case 1 show ?case by simp
next
  case 2 thus ?case by(simp add: lookup_map_of)
next
  case 3 thus ?case by(simp add: inorder_update)
next
  case 4 thus ?case by(simp add: inorder_delete)
qed auto

end
theory Tree2
imports Main
begin

datatype ('a, 'b) tree =
  Leaf ( $\langle \rangle$ ) |
  Node 'b ('a, 'b)tree 'a ('a, 'b) tree ((1<-/, -/, -/, -/))

fun inorder :: ('a, 'b)tree  $\Rightarrow$  'a list where
inorder Leaf = [] |
inorder (Node _ l a r) = inorder l @ a # inorder r

fun height :: ('a, 'b) tree  $\Rightarrow$  nat where
height Leaf = 0 |
height (Node _ l a r) = max (height l) (height r) + 1

end

```

10 Function *isin* for *Tree2*

```

theory Isin2
imports
  Tree2
  Cmp
  Set_by_Ordered
begin

fun isin :: ('a::cmp, 'b) tree  $\Rightarrow$  'a  $\Rightarrow$  bool where
isin Leaf x = False |
isin (Node _ l a r) x =
  (case cmp x a of

```

$LT \Rightarrow isin\ l\ x \mid$
 $EQ \Rightarrow True \mid$
 $GT \Rightarrow isin\ r\ x)$

lemma $sorted(inorder\ t) \Rightarrow isin\ t\ x = (x \in elems(inorder\ t))$
by ($induction\ t$) ($auto\ simp: elems_sims1$)

lemma $isin_set: sorted(inorder\ t) \Rightarrow isin\ t\ x = (x \in elems(inorder\ t))$
by ($induction\ t$) ($auto\ simp: elems_sims2$)

end

11 AVL Tree Implementation of Sets

theory *AVL_Set*
imports *Cmp Isin2*
begin

type_synonym $'a\ avl_tree = ('a, nat)\ tree$

Invariant:

fun $avl :: 'a\ avl_tree \Rightarrow bool$ **where**
 $avl\ Leaf = True \mid$
 $avl\ (Node\ h\ l\ a\ r) =$
 $((height\ l = height\ r \vee height\ l = height\ r + 1 \vee height\ r = height\ l + 1)$
 $\wedge$
 $h = max\ (height\ l)\ (height\ r) + 1 \wedge avl\ l \wedge avl\ r)$

fun $ht :: 'a\ avl_tree \Rightarrow nat$ **where**
 $ht\ Leaf = 0 \mid$
 $ht\ (Node\ h\ l\ a\ r) = h$

definition $node :: 'a\ avl_tree \Rightarrow 'a \Rightarrow 'a\ avl_tree \Rightarrow 'a\ avl_tree$ **where**
 $node\ l\ a\ r = Node\ (max\ (ht\ l)\ (ht\ r) + 1)\ l\ a\ r$

definition $balL :: 'a\ avl_tree \Rightarrow 'a \Rightarrow 'a\ avl_tree \Rightarrow 'a\ avl_tree$ **where**
 $balL\ l\ a\ r =$
 $(if\ ht\ l = ht\ r + 2\ then$
 $\quad case\ l\ of$
 $\quad Node\ _\ bl\ b\ br \Rightarrow$
 $\quad if\ ht\ bl < ht\ br\ then$
 $\quad \quad case\ br\ of$
 $\quad \quad Node\ _\ cl\ c\ cr \Rightarrow node\ (node\ bl\ b\ cl)\ c\ (node\ cr\ a\ r)$
 $\quad \quad else\ node\ bl\ b\ (node\ br\ a\ r)$

else node l a r)

definition *balR* :: 'a avl_tree $\Rightarrow$ 'a $\Rightarrow$ 'a avl_tree $\Rightarrow$ 'a avl_tree **where**
balR l a r =
 (if ht r = ht l + 2 then
 case r of
 Node _ bl b br $\Rightarrow$
 if ht bl > ht br then
 case bl of
 Node _ cl c cr $\Rightarrow$ node (node l a cl) c (node cr b br)
 else node (node l a bl) b br
 else node l a r)

fun *insert* :: 'a::cmp $\Rightarrow$ 'a avl_tree $\Rightarrow$ 'a avl_tree **where**
insert x Leaf = Node 1 Leaf x Leaf |
insert x (Node h l a r) = (case cmp x a of
 EQ $\Rightarrow$ Node h l a r |
 LT $\Rightarrow$ balL (insert x l) a r |
 GT $\Rightarrow$ balR l a (insert x r))

fun *del_max* :: 'a avl_tree $\Rightarrow$ 'a avl_tree * 'a **where**
del_max (Node _ l a r) =
 (if r = Leaf then (l,a) else let (r',a') = del_max r in (balL l a r', a'))

lemmas *del_max_induct* = *del_max.induct*[case_names Node Leaf]

fun *del_root* :: 'a avl_tree $\Rightarrow$ 'a avl_tree **where**
del_root (Node h Leaf a r) = r |
del_root (Node h l a Leaf) = l |
del_root (Node h l a r) = (let (l', a') = del_max l in balR l' a' r)

lemmas *del_root_cases* = *del_root.cases*[case_names Leaf_t Node_Leaf Node_Node]

fun *delete* :: 'a::cmp $\Rightarrow$ 'a avl_tree $\Rightarrow$ 'a avl_tree **where**
delete _ Leaf = Leaf |
delete x (Node h l a r) =
 (case cmp x a of
 EQ $\Rightarrow$ del_root (Node h l a r) |
 LT $\Rightarrow$ balR (delete x l) a r |
 GT $\Rightarrow$ balL l a (delete x r))

11.1 Functional Correctness Proofs

Very different from the AFP/AVL proofs

11.1.1 Proofs for insert

lemma *inorder_balL*:

$\text{inorder } (\text{balL } l \ a \ r) = \text{inorder } l \ @ \ a \ \# \ \text{inorder } r$
by (*auto simp: node_def balL_def split: tree.splits*)

lemma *inorder_balR*:

$\text{inorder } (\text{balR } l \ a \ r) = \text{inorder } l \ @ \ a \ \# \ \text{inorder } r$
by (*auto simp: node_def balR_def split: tree.splits*)

theorem *inorder_insert*:

$\text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{insert } x \ t) = \text{ins_list } x \ (\text{inorder } t)$
by (*induct t*)
(*auto simp: ins_list_simps inorder_balL inorder_balR*)

11.1.2 Proofs for delete

lemma *inorder_del_maxD*:

$\llbracket \text{del_max } t = (t', a); t \neq \text{Leaf} \rrbracket \implies$
 $\text{inorder } t' \ @ \ [a] = \text{inorder } t$
by(*induction t arbitrary: t' rule: del_max.induct*)
(*auto simp: inorder_balL split: if_splits prod.splits tree.split*)

lemma *inorder_del_root*:

$\text{inorder } (\text{del_root } (\text{Node } h \ l \ a \ r)) = \text{inorder } l \ @ \ \text{inorder } r$
by(*induction Node h l a r arbitrary: l a r h rule: del_root.induct*)
(*auto simp: inorder_balL inorder_balR inorder_del_maxD split: if_splits prod.splits*)

theorem *inorder_delete*:

$\text{sorted}(\text{inorder } t) \implies \text{inorder } (\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$
by(*induction t*)
(*auto simp: del_list_simps inorder_balL inorder_balR*
inorder_del_root inorder_del_maxD split: prod.splits)

11.1.3 Overall functional correctness

interpretation *Set_by_Ordered*

where *empty* = *Leaf* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* =
delete

and *inorder* = *inorder* **and** *inv* = $\lambda _.$ *True*

proof (*standard, goal_cases*)

case 1 **show** ?*case* **by** *simp*

next

case 2 **thus** ?*case* **by** (*simp add: isin_set*)

next

```

    case 3 thus ?case by(simp add: inorder_insert)
next
    case 4 thus ?case by(simp add: inorder_delete)
qed (rule TrueI)+

```

11.2 AVL invariants

Essentially the AFP/AVL proofs

11.2.1 Insertion maintains AVL balance

```

declare Let_def [simp]

```

```

lemma [simp]: avl t  $\implies$  ht t = height t
by (induct t) simp_all

```

```

lemma height_balL:
   $\llbracket \text{height } l = \text{height } r + 2; \text{avl } l; \text{avl } r \rrbracket \implies$ 
   $\text{height } (\text{balL } l \ a \ r) = \text{height } r + 2 \vee$ 
   $\text{height } (\text{balL } l \ a \ r) = \text{height } r + 3$ 
by (cases l) (auto simp: node_def balL_def split: tree.split)

```

```

lemma height_balR:
   $\llbracket \text{height } r = \text{height } l + 2; \text{avl } l; \text{avl } r \rrbracket \implies$ 
   $\text{height } (\text{balR } l \ a \ r) = \text{height } l + 2 \vee$ 
   $\text{height } (\text{balR } l \ a \ r) = \text{height } l + 3$ 
by (cases r) (auto simp add: node_def balR_def split: tree.split)

```

```

lemma [simp]: height (node l a r) = max (height l) (height r) + 1
by (simp add: node_def)

```

```

lemma avl_node:
   $\llbracket \text{avl } l; \text{avl } r; \text{height } l = \text{height } r \vee \text{height } l = \text{height } r + 1 \vee \text{height } r = \text{height } l + 1 \rrbracket$ 
   $\implies \text{avl}(\text{node } l \ a \ r)$ 
by (auto simp add: max_def node_def)

```

```

lemma height_balL2:
   $\llbracket \text{avl } l; \text{avl } r; \text{height } l \neq \text{height } r + 2 \rrbracket \implies$ 
   $\text{height } (\text{balL } l \ a \ r) = (1 + \max (\text{height } l) (\text{height } r))$ 
by (cases l, cases r) (simp_all add: balL_def)

```

```

lemma height_balR2:
   $\llbracket \text{avl } l; \text{avl } r; \text{height } r \neq \text{height } l + 2 \rrbracket \implies$ 

```

$height\ (balR\ l\ a\ r) = (1 + \max\ (height\ l)\ (height\ r))$
by (*cases l, cases r*) (*simp_all add: balR_def*)

lemma *avl_balL*:

assumes *avl l avl r and height l = height r $\vee$ height l = height r + 1*
 $\vee$ *height r = height l + 1 $\vee$ height l = height r + 2*
shows *avl(balL l a r)*
proof(*cases l*)
 case *Leaf*
 with *assms* **show** *?thesis* **by** (*simp add: node_def balL_def*)
next
 case (*Node ln ll lr lh*)
 with *assms* **show** *?thesis*
 proof(*cases height l = height r + 2*)
 case *True*
 from *True Node assms* **show** *?thesis*
 by (*auto simp: balL_def intro!: avl_node split: tree.split*) *arith+*
 next
 case *False*
 with *assms* **show** *?thesis* **by** (*simp add: avl_node balL_def*)
 qed
qed

lemma *avl_balR*:

assumes *avl l and avl r and height l = height r $\vee$ height l = height r + 1*
 $\vee$ *height r = height l + 1 $\vee$ height r = height l + 2*
shows *avl(balR l a r)*
proof(*cases r*)
 case *Leaf*
 with *assms* **show** *?thesis* **by** (*simp add: node_def balR_def*)
next
 case (*Node rn rl rr rh*)
 with *assms* **show** *?thesis*
 proof(*cases height r = height l + 2*)
 case *True*
 from *True Node assms* **show** *?thesis*
 by (*auto simp: balR_def intro!: avl_node split: tree.split*) *arith+*
 next
 case *False*
 with *assms* **show** *?thesis* **by** (*simp add: balR_def avl_node*)
 qed
qed

Insertion maintains the AVL property:

```

theorem avl_insert_aux:
  assumes avl t
  shows avl(insert x t)
    (height (insert x t) = height t  $\vee$  height (insert x t) = height t + 1)
using assms
proof (induction t)
  case (Node h l a r)
  case 1
  with Node show ?case
  proof(cases x = a)
    case True
    with Node 1 show ?thesis by simp
  next
  case False
  with Node 1 show ?thesis
  proof(cases x < a)
    case True
    with Node 1 show ?thesis by (auto simp add:avl_balL)
  next
  case False
  with Node 1  $\langle x \neq a \rangle$  show ?thesis by (auto simp add:avl_balR)
  qed
qed
case 2
from 2 Node show ?case
proof(cases x = a)
  case True
  with Node 1 show ?thesis by simp
next
  case False
  with Node 1 show ?thesis
  proof(cases x < a)
    case True
    with Node 2 show ?thesis
    proof(cases height (insert x l) = height r + 2)
      case False with Node 2  $\langle x < a \rangle$  show ?thesis by (auto simp:
height_balL2)
    next
    case True
    hence (height (balL (insert x l) a r) = height r + 2)  $\vee$ 
      (height (balL (insert x l) a r) = height r + 3) (is ?A  $\vee$  ?B)
    using Node 2 by (intro height_balL) simp_all
  
```

```

    thus ?thesis
  proof
    assume ?A
    with 2  $\langle x < a \rangle$  show ?thesis by (auto)
  next
    assume ?B
    with True 1 Node(2)  $\langle x < a \rangle$  show ?thesis by (simp) arith
  qed
qed
next
case False
with Node 2 show ?thesis
proof(cases height (insert x r) = height l + 2)
  case False
  with Node 2  $\langle \neg x < a \rangle$  show ?thesis by (auto simp: height_balR2)
next
case True
hence (height (balR l a (insert x r)) = height l + 2)  $\vee$ 
      (height (balR l a (insert x r)) = height l + 3) (is ?A  $\vee$  ?B)
  using Node 2 by (intro height_balR) simp_all
thus ?thesis
proof
  assume ?A
  with 2  $\langle \neg x < a \rangle$  show ?thesis by (auto)
next
  assume ?B
  with True 1 Node(4)  $\langle \neg x < a \rangle$  show ?thesis by (simp) arith
qed
qed
qed
qed simp_all

```

11.2.2 Deletion maintains AVL balance

```

lemma avl_del_max:
  assumes avl x and  $x \neq \text{Leaf}$ 
  shows avl (fst (del_max x))  $\wedge$  height x = height (fst (del_max x))  $\vee$ 
        height x = height (fst (del_max x)) + 1
  using assms
proof (induct x rule: del_max_induct)
  case (Node h l a r)
  case 1
  thus ?case using Node

```

```

    by (auto simp: height_balL height_balL2 avl_balL
        linorder_class.max.absorb1 linorder_class.max.absorb2
        split:prod.split)
next
  case (Node h l a r)
  case 2
  let ?r' = fst (del_max r)
  from ⟨avl x⟩ Node 2 have avl l and avl r by simp_all
  thus ?case using Node 2 height_balL[of l ?r' a] height_balL2[of l ?r' a]
    apply (auto split:prod.splits simp del:avl.simps) by arith+
qed auto

lemma avl_del_root:
  assumes avl t and t ≠ Leaf
  shows avl (del_root t)
using assms
proof (cases t rule:del_root_cases)
  case (Node_Node h lh ll ln lr n rh rl rn rr)
  let ?l = Node lh ll ln lr
  let ?r = Node rh rl rn rr
  let ?l' = fst (del_max ?l)
  from ⟨avl t⟩ and Node_Node have avl ?r by simp
  from ⟨avl t⟩ and Node_Node have avl ?l by simp
  hence avl(?l') height ?l = height(?l') ∨
    height ?l = height(?l') + 1 by (rule avl_del_max,simp)+
  with ⟨avl t⟩ Node_Node have height ?l' = height ?r ∨ height ?l' = height
    ?r + 1
    ∨ height ?r = height ?l' + 1 ∨ height ?r = height ?l' + 2 by
fastforce
  with ⟨avl ?l'⟩ ⟨avl ?r⟩ have avl (balR ?l' (snd (del_max ?l)) ?r)
    by (rule avl_balR)
  with Node_Node show ?thesis by (auto split:prod.splits)
qed simp_all

lemma height_del_root:
  assumes avl t and t ≠ Leaf
  shows height t = height (del_root t) ∨ height t = height (del_root t) + 1
using assms
proof (cases t rule: del_root_cases)
  case (Node_Node h lh ll ln lr n rh rl rn rr)
  let ?l = Node lh ll ln lr
  let ?r = Node rh rl rn rr
  let ?l' = fst (del_max ?l)
  let ?t' = balR ?l' (snd (del_max ?l)) ?r

```

```

from ⟨avl t⟩ and Node_Node have avl ?r by simp
from ⟨avl t⟩ and Node_Node have avl ?l by simp
hence avl(?l') by (rule avl_del_max,simp)
have l'_height: height ?l = height ?l' ∨ height ?l = height ?l' + 1 using
⟨avl ?l⟩ by (intro avl_del_max) auto
have t_height: height t = 1 + max (height ?l) (height ?r) using ⟨avl t⟩
Node_Node by simp
have height t = height ?t' ∨ height t = height ?t' + 1 using ⟨avl t⟩
Node_Node
proof(cases height ?r = height ?l' + 2)
  case False
    show ?thesis using l'_height t_height False by (subst height_balR2[OF
⟨avl ?l'⟩ ⟨avl ?r⟩ False])+ arith
  next
    case True
      show ?thesis
      proof(cases rule: disjE[OF height_balR[OF True ⟨avl ?l'⟩ ⟨avl ?r⟩, of snd
(del_max ?l)]])
        case 1
          thus ?thesis using l'_height t_height True by arith
        next
          case 2
            thus ?thesis using l'_height t_height True by arith
      qed
    qed
  thus ?thesis using Node_Node by (auto split:prod.splits)
qed simp_all

```

Deletion maintains the AVL property:

```

theorem avl_delete_aux:
  assumes avl t
  shows avl(delete x t) and height t = (height (delete x t)) ∨ height t =
height (delete x t) + 1
using assms
proof (induct t)
  case (Node h l n r)
  case 1
    with Node show ?case
    proof(cases x = n)
      case True
        with Node 1 show ?thesis by (auto simp:avl_del_root)
      next
        case False
          with Node 1 show ?thesis
    qed
  qed

```

```

proof(cases  $x < n$ )
  case True
    with Node 1 show ?thesis by (auto simp add:avl_balR)
next
  case False
    with Node 1  $\langle x \neq n \rangle$  show ?thesis by (auto simp add:avl_balL)
qed
qed
case 2
with Node show ?case
proof(cases  $x = n$ )
  case True
    with 1 have height (Node  $h\ l\ n\ r$ ) = height (del_root (Node  $h\ l\ n\ r$ ))
       $\vee$  height (Node  $h\ l\ n\ r$ ) = height (del_root (Node  $h\ l\ n\ r$ )) + 1
      by (subst height_del_root,simp_all)
    with True show ?thesis by simp
next
  case False
    with Node 1 show ?thesis
    proof(cases  $x < n$ )
      case True
        show ?thesis
        proof(cases height  $r = \text{height } (\text{delete } x\ l) + 2$ )
          case False with Node 1  $\langle x < n \rangle$  show ?thesis by(auto simp: balR_def)
          next
            case True
              hence (height (balR (delete  $x\ l$ )  $n\ r$ ) = height (delete  $x\ l$ ) + 2)  $\vee$ 
                height (balR (delete  $x\ l$ )  $n\ r$ ) = height (delete  $x\ l$ ) + 3 (is ?A  $\vee$ 
?B)
              using Node 2 by (intro height_balR) auto
            thus ?thesis
          proof
            assume ?A
            with  $\langle x < n \rangle$  Node 2 show ?thesis by(auto simp: balR_def)
          next
            assume ?B
            with  $\langle x < n \rangle$  Node 2 show ?thesis by(auto simp: balR_def)
          qed
        qed
      next
        case False
          show ?thesis
          proof(cases height  $l = \text{height } (\text{delete } x\ r) + 2$ )
            case False with Node 1  $\langle \neg x < n \rangle\ \langle x \neq n \rangle$  show ?thesis by(auto

```

```

simp: balL_def)
  next
  case True
  hence (height (balL l n (delete x r)) = height (delete x r) + 2) ∨
        height (balL l n (delete x r)) = height (delete x r) + 3 (is ?A ∨
?B)
    using Node 2 by (intro height_balL) auto
  thus ?thesis
  proof
    assume ?A
    with ⟨¬x < n⟩ ⟨x ≠ n⟩ Node 2 show ?thesis by (auto simp: balL_def)
  next
    assume ?B
    with ⟨¬x < n⟩ ⟨x ≠ n⟩ Node 2 show ?thesis by (auto simp: balL_def)
  qed
qed
qed
qed
qed simp_all
end

```

12 Function *lookup* for Tree2

```

theory Lookup2
imports
  Tree2
  Cmp
  Map.by_Ordered
begin

fun lookup :: ('a::cmp * 'b, 'c) tree ⇒ 'a ⇒ 'b option where
lookup Leaf x = None |
lookup (Node _ l (a,b) r) x =
  (case cmp x a of LT ⇒ lookup l x | GT ⇒ lookup r x | EQ ⇒ Some b)

lemma lookup_map_of:
  sorted1(inorder t) ⇒ lookup t x = map_of (inorder t) x
by(induction t) (auto simp: map_of_simps split: option.split)

end

```

13 AVL Tree Implementation of Maps

theory *AVL_Map*

imports

AVL_Set

Lookup2

begin

fun *update* :: '*a*::*cmp* $\Rightarrow$ '*b* $\Rightarrow$ ('*a**'*b*) *avl_tree* $\Rightarrow$ ('*a**'*b*) *avl_tree* **where**
update *x y Leaf* = *Node* 1 *Leaf* (*x,y*) *Leaf* |
update *x y* (*Node* *h l* (*a,b*) *r*) = (case *cmp* *x a* of
EQ $\Rightarrow$ *Node* *h l* (*x,y*) *r* |
LT $\Rightarrow$ *balL* (*update* *x y l*) (*a,b*) *r* |
GT $\Rightarrow$ *balR* *l* (*a,b*) (*update* *x y r*))

fun *delete* :: '*a*::*cmp* $\Rightarrow$ ('*a**'*b*) *avl_tree* $\Rightarrow$ ('*a**'*b*) *avl_tree* **where**
delete _ *Leaf* = *Leaf* |
delete *x* (*Node* *h l* (*a,b*) *r*) = (case *cmp* *x a* of
EQ $\Rightarrow$ *del_root* (*Node* *h l* (*a,b*) *r*) |
LT $\Rightarrow$ *balR* (*delete* *x l*) (*a,b*) *r* |
GT $\Rightarrow$ *balL* *l* (*a,b*) (*delete* *x r*))

13.1 Functional Correctness Proofs

theorem *inorder_update*:

sorted1(*inorder* *t*) $\Longrightarrow$ *inorder*(*update* *x y t*) = *upd_list* *x y* (*inorder* *t*)

by (*induct* *t*) (*auto simp*: *upd_list_simps* *inorder_balL* *inorder_balR*)

theorem *inorder_delete*:

sorted1(*inorder* *t*) $\Longrightarrow$ *inorder* (*delete* *x t*) = *del_list* *x* (*inorder* *t*)

by(*induction* *t*)

(*auto simp*: *del_list_simps* *inorder_balL* *inorder_balR*
inorder_del_root *inorder_del_maxD* *split*: *prod.splits*)

interpretation *Map_by_Ordered*

where *empty* = *Leaf* **and** *lookup* = *lookup* **and** *update* = *update* **and** *delete*
= *delete*

and *inorder* = *inorder* **and** *inv* = $\lambda_.$ *True*

proof (*standard*, *goal_cases*)

case 1 **show** ?*case* **by** *simp*

next

case 2 **thus** ?*case* **by**(*simp add*: *lookup_map_of*)

next

```

    case 3 thus ?case by(simp add: inorder_update)
next
    case 4 thus ?case by(simp add: inorder_delete)
qed auto

end

```

14 Red-Black Trees

```

theory RBT
imports Tree2
begin

```

```

datatype color = Red | Black

```

```

type_synonym 'a rbt = ('a,color)tree

```

```

abbreviation R where R l a r ≡ Node Red l a r

```

```

abbreviation B where B l a r ≡ Node Black l a r

```

```

fun bal :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt where

```

```

bal (R t1 a1 t2) a2 (R t3 a3 t4) = R (B t1 a1 t2) a2 (B t3 a3 t4) |
bal (R (R t1 a1 t2) a2 t3) a3 t4 = R (B t1 a1 t2) a2 (B t3 a3 t4) |
bal (R t1 a1 (R t2 a2 t3)) a3 t4 = R (B t1 a1 t2) a2 (B t3 a3 t4) |
bal t1 a1 (R t2 a2 (R t3 a3 t4)) = R (B t1 a1 t2) a2 (B t3 a3 t4) |
bal t1 a1 (R (R t2 a2 t3) a3 t4) = R (B t1 a1 t2) a2 (B t3 a3 t4) |
bal t1 a t2 = B t1 a t2

```

```

fun paint :: color ⇒ 'a rbt ⇒ 'a rbt where

```

```

paint c Leaf = Leaf |
paint c (Node _ l a r) = Node c l a r

```

```

fun balL :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt where

```

```

balL (R t1 x t2) y t3 = R (B t1 x t2) y t3 |
balL bl x (B t1 y t2) = bal bl x (R t1 y t2) |
balL bl x (R (B t1 y t2) z t3) = R (B bl x t1) y (bal t2 z (paint Red t3)) |
balL t1 x t2 = R t1 x t2

```

```

fun balR :: 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt where

```

```

balR t1 x (R t2 y t3) = R t1 x (B t2 y t3) |
balR (B t1 x t2) y t3 = bal (R t1 x t2) y t3 |
balR (R t1 x (B t2 y t3)) z t4 = R (bal (paint Red t1) x t2) y (B t3 z t4) |

```

$balR\ t1\ x\ t2 = R\ t1\ x\ t2$

```

fun combine :: 'a rbt  $\Rightarrow$  'a rbt  $\Rightarrow$  'a rbt where
  combine Leaf t = t |
  combine t Leaf = t |
  combine (R t1 a t2) (R t3 c t4) =
    (case combine t2 t3 of
      R u2 b u3  $\Rightarrow$  (R (R t1 a u2) b (R u3 c t4)) |
      t23  $\Rightarrow$  R t1 a (R t23 c t4)) |
  combine (B t1 a t2) (B t3 c t4) =
    (case combine t2 t3 of
      R t2' b t3'  $\Rightarrow$  R (B t1 a t2') b (B t3' c t4) |
      t23  $\Rightarrow$  balL t1 a (B t23 c t4)) |
  combine t1 (R t2 a t3) = R (combine t1 t2) a t3 |
  combine (R t1 a t2) t3 = R t1 a (combine t2 t3)

end

```

15 Red-Black Tree Implementation of Sets

```

theory RBT_Set
imports
  RBT
  Cmp
  Isin2
begin

```

```

fun ins :: 'a::cmp  $\Rightarrow$  'a rbt  $\Rightarrow$  'a rbt where
  ins x Leaf = R Leaf x Leaf |
  ins x (B l a r) =
    (case cmp x a of
      LT  $\Rightarrow$  bal (ins x l) a r |
      GT  $\Rightarrow$  bal l a (ins x r) |
      EQ  $\Rightarrow$  B l a r) |
  ins x (R l a r) =
    (case cmp x a of
      LT  $\Rightarrow$  R (ins x l) a r |
      GT  $\Rightarrow$  R l a (ins x r) |
      EQ  $\Rightarrow$  R l a r)

```

```

definition insert :: 'a::cmp  $\Rightarrow$  'a rbt  $\Rightarrow$  'a rbt where
  insert x t = paint Black (ins x t)

```

```

fun del :: 'a::cmp ⇒ 'a rbt ⇒ 'a rbt
and delL :: 'a::cmp ⇒ 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
and delR :: 'a::cmp ⇒ 'a rbt ⇒ 'a ⇒ 'a rbt ⇒ 'a rbt
where
  del x Leaf = Leaf |
  del x (Node _ l a r) =
    (case cmp x a of
      LT ⇒ delL x l a r |
      GT ⇒ delR x l a r |
      EQ ⇒ combine l r) |
  delL x (B t1 a t2) b t3 = balL (del x (B t1 a t2)) b t3 |
  delL x l a r = R (del x l) a r |
  delR x t1 a (B t2 b t3) = balR t1 a (del x (B t2 b t3)) |
  delR x l a r = R l a (del x r)

```

definition delete :: 'a::cmp ⇒ 'a rbt ⇒ 'a rbt **where**
 delete x t = paint Black (del x t)

15.1 Functional Correctness Proofs

lemma inorder_paint: inorder(paint c t) = inorder t
by(induction t) (auto)

lemma inorder_bal:
 inorder(bal l a r) = inorder l @ a # inorder r
by(induction l a r rule: bal.induct) (auto)

lemma inorder_ins:
 sorted(inorder t) ⇒ inorder(ins x t) = ins_list x (inorder t)
by(induction x t rule: ins.induct) (auto simp: ins_list_simps inorder_bal)

lemma inorder_insert:
 sorted(inorder t) ⇒ inorder(insert x t) = ins_list x (inorder t)
by (simp add: insert_def inorder_ins inorder_paint)

lemma inorder_balL:
 inorder(balL l a r) = inorder l @ a # inorder r
by(induction l a r rule: balL.induct)(auto simp: inorder_bal inorder_paint)

lemma inorder_balR:
 inorder(balR l a r) = inorder l @ a # inorder r
by(induction l a r rule: balR.induct) (auto simp: inorder_bal inorder_paint)

lemma inorder_combine:

```

inorder (combine l r) = inorder l @ inorder r
by (induction l r rule: combine.induct)
(auto simp: inorder_balL inorder_balR split: tree.split color.split)

```

lemma *inorder_del*:

```

sorted (inorder t)  $\implies$  inorder (del x t) = del_list x (inorder t)
sorted (inorder l)  $\implies$  inorder (delL x l a r) =
  del_list x (inorder l) @ a # inorder r
sorted (inorder r)  $\implies$  inorder (delR x l a r) =
  inorder l @ a # del_list x (inorder r)
by (induction x t and x l a r and x l a r rule: del_delL_delR.induct)
(auto simp: del_list_simps inorder_combine inorder_balL inorder_balR)

```

lemma *inorder_delete*:

```

sorted (inorder t)  $\implies$  inorder (delete x t) = del_list x (inorder t)
by (auto simp: delete_def inorder_del inorder_paint)

```

interpretation *Set.by_Ordered*

where *empty* = *Leaf* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*

and *inorder* = *inorder* **and** *inv* = $\lambda_.$ *True*

proof (*standard*, *goal_cases*)

case 1 **show** ?*case* **by** *simp*

next

case 2 **thus** ?*case* **by** (*simp add: isin_set*)

next

case 3 **thus** ?*case* **by** (*simp add: inorder_insert*)

next

case 4 **thus** ?*case* **by** (*simp add: inorder_delete*)

qed *auto*

15.2 Structural invariants

fun *color* :: 'a rbt $\Rightarrow$ *color* **where**

color *Leaf* = *Black* |

color (*Node* *c* _ _) = *c*

fun *bheight* :: 'a rbt $\Rightarrow$ *nat* **where**

bheight *Leaf* = 0 |

bheight (*Node* *c* *l* *x* *r*) = (if *c* = *Black* then *Suc*(*bheight* *l*) else *bheight* *l*)

fun *inv1* :: 'a rbt $\Rightarrow$ *bool* **where**

inv1 *Leaf* = *True* |

inv1 (Node *c l a r*) =
 (*inv1 l* ∧ *inv1 r* ∧ (*c* = *Black* ∨ *color l* = *Black* ∧ *color r* = *Black*))

fun *inv1_root* :: 'a rbt ⇒ bool — Weaker version **where**
inv1_root Leaf = *True* |
inv1_root (Node *c l a r*) = (*inv1 l* ∧ *inv1 r*)

fun *inv2* :: 'a rbt ⇒ bool **where**
inv2 Leaf = *True* |
inv2 (Node *c l x r*) = (*inv2 l* ∧ *inv2 r* ∧ *bheight l* = *bheight r*)

lemma *inv1_rootI[simp]*: *inv1 t* ⇒ *inv1_root t*
by (*cases t*) *simp*+

definition *rbt* :: 'a rbt ⇒ bool **where**
rbt t = (*inv1 t* ∧ *inv2 t* ∧ *color t* = *Black*)

lemma *color_paint_Black*: *color (paint Black t)* = *Black*
by (*cases t*) *auto*

theorem *rbt_Leaf*: *rbt Leaf*
by (*simp add: rbt_def*)

lemma *paint_inv1_root*: *inv1_root t* ⇒ *inv1_root (paint c t)*
by (*cases t*) *auto*

lemma *inv1_paint_Black*: *inv1_root t* ⇒ *inv1 (paint Black t)*
by (*cases t*) *auto*

lemma *inv2_paint*: *inv2 t* ⇒ *inv2 (paint c t)*
by (*cases t*) *auto*

lemma *inv1_bal*: $\llbracket \text{inv1_root } l; \text{inv1_root } r \rrbracket \Rightarrow \text{inv1 } (\text{bal } l \text{ a } r)$
by (*induct l a r rule: bal.induct*) *auto*

lemma *bheight_bal*:
bheight l = *bheight r* ⇒ *bheight (bal l a r)* = *Suc (bheight l)*
by (*induct l a r rule: bal.induct*) *auto*

lemma *inv2_bal*:
 $\llbracket \text{inv2 } l; \text{inv2 } r; \text{bheight } l = \text{bheight } r \rrbracket \Rightarrow \text{inv2 } (\text{bal } l \text{ a } r)$
by (*induct l a r rule: bal.induct*) *auto*

15.2.1 Insertion

```

lemma inv1_ins: assumes inv1 t
  shows color t = Black  $\implies$  inv1 (ins x t) inv1_root (ins x t)
using assms
by (induct x t rule: ins.induct) (auto simp: inv1_bal)

lemma inv2_ins: assumes inv2 t
  shows inv2 (ins x t) bheight (ins x t) = bheight t
using assms
by (induct x t rule: ins.induct) (auto simp: inv2_bal bheight_bal)

theorem rbt_ins: rbt t  $\implies$  rbt (insert x t)
by (simp add: inv1_ins inv2_ins color_paint_Black inv1_paint_Black inv2_paint
  rbt_def insert_def)

end

```

16 Red-Black Tree Implementation of Maps

```

theory RBTree_Map
imports
  RBTree_Set
  Lookup2
begin

fun upd :: 'a::cmp  $\Rightarrow$  'b  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  ('a*'b) rbt where
  upd x y Leaf = R Leaf (x,y) Leaf |
  upd x y (B l (a,b) r) = (case cmp x a of
    LT  $\Rightarrow$  bal (upd x y l) (a,b) r |
    GT  $\Rightarrow$  bal l (a,b) (upd x y r) |
    EQ  $\Rightarrow$  B l (x,y) r) |
  upd x y (R l (a,b) r) = (case cmp x a of
    LT  $\Rightarrow$  R (upd x y l) (a,b) r |
    GT  $\Rightarrow$  R l (a,b) (upd x y r) |
    EQ  $\Rightarrow$  R l (x,y) r)

definition update :: 'a::cmp  $\Rightarrow$  'b  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  ('a*'b) rbt where
  update x y t = paint Black (upd x y t)

fun del :: 'a::cmp  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  ('a*'b) rbt
and delL :: 'a::cmp  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  'a*'b  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  ('a*'b) rbt
and delR :: 'a::cmp  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  'a*'b  $\Rightarrow$  ('a*'b) rbt  $\Rightarrow$  ('a*'b) rbt

```

where

$del\ x\ Leaf = Leaf \mid$
 $del\ x\ (Node\ c\ t1\ (a,b)\ t2) = (case\ cmp\ x\ a\ of$
 $\quad LT \Rightarrow delL\ x\ t1\ (a,b)\ t2 \mid$
 $\quad GT \Rightarrow delR\ x\ t1\ (a,b)\ t2 \mid$
 $\quad EQ \Rightarrow combine\ t1\ t2) \mid$
 $delL\ x\ (B\ t1\ a\ t2)\ b\ t3 = balL\ (del\ x\ (B\ t1\ a\ t2))\ b\ t3 \mid$
 $delL\ x\ t1\ a\ t2 = R\ (del\ x\ t1)\ a\ t2 \mid$
 $delR\ x\ t1\ a\ (B\ t2\ b\ t3) = balR\ t1\ a\ (del\ x\ (B\ t2\ b\ t3)) \mid$
 $delR\ x\ t1\ a\ t2 = R\ t1\ a\ (del\ x\ t2)$

definition $delete :: 'a::cmp \Rightarrow ('a*b)\ rbt \Rightarrow ('a*b)\ rbt$ **where**
 $delete\ x\ t = paint\ Black\ (del\ x\ t)$

16.1 Functional Correctness Proofs

lemma *inorder_upd*:

$sorted1(inorder\ t) \Longrightarrow inorder(upd\ x\ y\ t) = upd_list\ x\ y\ (inorder\ t)$

by(*induction* $x\ y\ t$ *rule*: *upd.induct*)

(*auto simp*: *upd_list_simps* *inorder_bal*)

lemma *inorder_update*:

$sorted1(inorder\ t) \Longrightarrow inorder(update\ x\ y\ t) = upd_list\ x\ y\ (inorder\ t)$

by(*simp add*: *update_def* *inorder_upd* *inorder_paint*)

lemma *inorder_del*:

$sorted1(inorder\ t1) \Longrightarrow inorder(del\ x\ t1) = del_list\ x\ (inorder\ t1)$ **and**

$sorted1(inorder\ t1) \Longrightarrow inorder(delL\ x\ t1\ a\ t2) =$

$del_list\ x\ (inorder\ t1) @ a \# inorder\ t2$ **and**

$sorted1(inorder\ t2) \Longrightarrow inorder(delR\ x\ t1\ a\ t2) =$

$inorder\ t1 @ a \# del_list\ x\ (inorder\ t2)$

by(*induction* $x\ t1$ **and** $x\ t1\ a\ t2$ **and** $x\ t1\ a\ t2$ *rule*: *del_delL_delR.induct*)

(*auto simp*: *del_list_simps* *inorder_combine* *inorder_balL* *inorder_balR*)

lemma *inorder_delete*:

$sorted1(inorder\ t) \Longrightarrow inorder(delete\ x\ t) = del_list\ x\ (inorder\ t)$

by(*simp add*: *delete_def* *inorder_del* *inorder_paint*)

interpretation *Map_by_Ordered*

where *empty* = *Leaf* **and** *lookup* = *lookup* **and** *update* = *update* **and** *delete* = *delete*

and *inorder* = *inorder* **and** *inv* = $\lambda_.$ *True*

proof (*standard*, *goal_cases*)

case 1 **show** ?*case* **by** *simp*

```

next
  case 2 thus ?case by(simp add: lookup_map_of)
next
  case 3 thus ?case by(simp add: inorder_update)
next
  case 4 thus ?case by(simp add: inorder_delete)
qed auto

end

```

17 2-3 Trees

```

theory Tree23
imports Main
begin

```

```

class height =
fixes height :: 'a  $\Rightarrow$  nat

```

```

datatype 'a tree23 =
  Leaf ( $\langle \rangle$ ) |
  Node2 'a tree23 'a 'a tree23 ( $\langle -, -, - \rangle$ ) |
  Node3 'a tree23 'a 'a tree23 'a 'a tree23 ( $\langle -, -, -, - \rangle$ )

```

```

fun inorder :: 'a tree23  $\Rightarrow$  'a list where
inorder Leaf = [] |
inorder(Node2 l a r) = inorder l @ a # inorder r |
inorder(Node3 l a m b r) = inorder l @ a # inorder m @ b # inorder r

```

```

instantiation tree23 :: (type)height
begin

```

```

fun height_tree23 :: 'a tree23  $\Rightarrow$  nat where
height Leaf = 0 |
height (Node2 l _ r) = Suc(max (height l) (height r)) |
height (Node3 l _ m _ r) = Suc(max (height l) (max (height m) (height r)))

```

```

instance ..

```

```

end

```

Balanced:

```

fun bal :: 'a tree23  $\Rightarrow$  bool where

```

```

    bal Leaf = True |
    bal (Node2 l _ r) = (bal l & bal r & height l = height r) |
    bal (Node3 l _ m _ r) =
      (bal l & bal m & bal r & height l = height m & height m = height r)

end

```

18 2-3 Tree Implementation of Sets

```

theory Tree23_Set
imports
  Tree23
  Cmp
  Set_by_Ordered
begin

fun isin :: 'a::cmp tree23  $\Rightarrow$  'a  $\Rightarrow$  bool where
  isin Leaf x = False |
  isin (Node2 l a r) x =
    (case cmp x a of
      LT  $\Rightarrow$  isin l x |
      EQ  $\Rightarrow$  True |
      GT  $\Rightarrow$  isin r x) |
  isin (Node3 l a m b r) x =
    (case cmp x a of
      LT  $\Rightarrow$  isin l x |
      EQ  $\Rightarrow$  True |
      GT  $\Rightarrow$ 
        (case cmp x b of
          LT  $\Rightarrow$  isin m x |
          EQ  $\Rightarrow$  True |
          GT  $\Rightarrow$  isin r x))

datatype 'a upi = Ti 'a tree23 | Upi 'a tree23 'a 'a tree23

fun treei :: 'a upi  $\Rightarrow$  'a tree23 where
  treei (Ti t) = t |
  treei (Upi l a r) = Node2 l a r

fun ins :: 'a::cmp  $\Rightarrow$  'a tree23  $\Rightarrow$  'a upi where
  ins x Leaf = Upi Leaf x Leaf |
  ins x (Node2 l a r) =
    (case cmp x a of

```

```

LT ⇒
  (case ins x l of
    Ti l' => Ti (Node2 l' a r) |
    Upi l1 b l2 => Ti (Node3 l1 b l2 a r)) |
EQ ⇒ Ti (Node2 l x r) |
GT ⇒
  (case ins x r of
    Ti r' => Ti (Node2 l a r') |
    Upi r1 b r2 => Ti (Node3 l a r1 b r2))) |
ins x (Node3 l a m b r) =
  (case cmp x a of
    LT ⇒
      (case ins x l of
        Ti l' => Ti (Node3 l' a m b r) |
        Upi l1 c l2 => Upi (Node2 l1 c l2) a (Node2 m b r)) |
      EQ ⇒ Ti (Node3 l a m b r) |
      GT ⇒
        (case cmp x b of
          GT ⇒
            (case ins x r of
              Ti r' => Ti (Node3 l a m b r') |
              Upi r1 c r2 => Upi (Node2 l a m) b (Node2 r1 c r2)) |
            EQ ⇒ Ti (Node3 l a m b r) |
          LT ⇒
            (case ins x m of
              Ti m' => Ti (Node3 l a m' b r) |
              Upi m1 c m2 => Upi (Node2 l a m1) c (Node2 m2 b r))))

```

hide_const insert

definition insert :: 'a::cmp ⇒ 'a tree23 ⇒ 'a tree23 **where**
 insert x t = tree_i(ins x t)

datatype 'a up_d = T_d 'a tree23 | Up_d 'a tree23

fun tree_d :: 'a up_d ⇒ 'a tree23 **where**
 tree_d (T_d t) = t |
 tree_d (Up_d t) = t

fun node21 :: 'a up_d ⇒ 'a ⇒ 'a tree23 ⇒ 'a up_d **where**
 node21 (T_d t1) a t2 = T_d(Node2 t1 a t2) |
 node21 (Up_d t1) a (Node2 t2 b t3) = Up_d(Node3 t1 a t2 b t3) |

$node21 \ (Up_d \ t1) \ a \ (Node3 \ t2 \ b \ t3 \ c \ t4) = T_d(Node2 \ (Node2 \ t1 \ a \ t2) \ b \ (Node2 \ t3 \ c \ t4))$

fun $node22 :: 'a \ tree23 \Rightarrow 'a \Rightarrow 'a \ up_d \Rightarrow 'a \ up_d$ **where**
 $node22 \ t1 \ a \ (T_d \ t2) = T_d(Node2 \ t1 \ a \ t2) \mid$
 $node22 \ (Node2 \ t1 \ b \ t2) \ a \ (Up_d \ t3) = Up_d(Node3 \ t1 \ b \ t2 \ a \ t3) \mid$
 $node22 \ (Node3 \ t1 \ b \ t2 \ c \ t3) \ a \ (Up_d \ t4) = T_d(Node2 \ (Node2 \ t1 \ b \ t2) \ c \ (Node2 \ t3 \ a \ t4))$

fun $node31 :: 'a \ up_d \Rightarrow 'a \Rightarrow 'a \ tree23 \Rightarrow 'a \Rightarrow 'a \ tree23 \Rightarrow 'a \ up_d$ **where**
 $node31 \ (T_d \ t1) \ a \ t2 \ b \ t3 = T_d(Node3 \ t1 \ a \ t2 \ b \ t3) \mid$
 $node31 \ (Up_d \ t1) \ a \ (Node2 \ t2 \ b \ t3) \ c \ t4 = T_d(Node2 \ (Node3 \ t1 \ a \ t2 \ b \ t3) \ c \ t4) \mid$
 $node31 \ (Up_d \ t1) \ a \ (Node3 \ t2 \ b \ t3 \ c \ t4) \ d \ t5 = T_d(Node3 \ (Node2 \ t1 \ a \ t2) \ b \ (Node2 \ t3 \ c \ t4) \ d \ t5)$

fun $node32 :: 'a \ tree23 \Rightarrow 'a \Rightarrow 'a \ up_d \Rightarrow 'a \Rightarrow 'a \ tree23 \Rightarrow 'a \ up_d$ **where**
 $node32 \ t1 \ a \ (T_d \ t2) \ b \ t3 = T_d(Node3 \ t1 \ a \ t2 \ b \ t3) \mid$
 $node32 \ t1 \ a \ (Up_d \ t2) \ b \ (Node2 \ t3 \ c \ t4) = T_d(Node2 \ t1 \ a \ (Node3 \ t2 \ b \ t3 \ c \ t4)) \mid$
 $node32 \ t1 \ a \ (Up_d \ t2) \ b \ (Node3 \ t3 \ c \ t4 \ d \ t5) = T_d(Node3 \ t1 \ a \ (Node2 \ t2 \ b \ t3) \ c \ (Node2 \ t4 \ d \ t5))$

fun $node33 :: 'a \ tree23 \Rightarrow 'a \Rightarrow 'a \ tree23 \Rightarrow 'a \Rightarrow 'a \ up_d \Rightarrow 'a \ up_d$ **where**
 $node33 \ l \ a \ m \ b \ (T_d \ r) = T_d(Node3 \ l \ a \ m \ b \ r) \mid$
 $node33 \ t1 \ a \ (Node2 \ t2 \ b \ t3) \ c \ (Up_d \ t4) = T_d(Node2 \ t1 \ a \ (Node3 \ t2 \ b \ t3 \ c \ t4)) \mid$
 $node33 \ t1 \ a \ (Node3 \ t2 \ b \ t3 \ c \ t4) \ d \ (Up_d \ t5) = T_d(Node3 \ t1 \ a \ (Node2 \ t2 \ b \ t3) \ c \ (Node2 \ t4 \ d \ t5))$

fun $del_min :: 'a \ tree23 \Rightarrow 'a \ * \ 'a \ up_d$ **where**
 $del_min \ (Node2 \ Leaf \ a \ Leaf) = (a, \ Up_d \ Leaf) \mid$
 $del_min \ (Node3 \ Leaf \ a \ Leaf \ b \ Leaf) = (a, \ T_d(Node2 \ Leaf \ b \ Leaf)) \mid$
 $del_min \ (Node2 \ l \ a \ r) = (let \ (x,l') = del_min \ l \ in \ (x, \ node21 \ l' \ a \ r)) \mid$
 $del_min \ (Node3 \ l \ a \ m \ b \ r) = (let \ (x,l') = del_min \ l \ in \ (x, \ node31 \ l' \ a \ m \ b \ r))$

fun $del :: 'a::cmp \Rightarrow 'a \ tree23 \Rightarrow 'a \ up_d$ **where**
 $del \ x \ Leaf = T_d \ Leaf \mid$
 $del \ x \ (Node2 \ Leaf \ a \ Leaf) =$
 $\quad (if \ x = a \ then \ Up_d \ Leaf \ else \ T_d(Node2 \ Leaf \ a \ Leaf)) \mid$
 $del \ x \ (Node3 \ Leaf \ a \ Leaf \ b \ Leaf) =$
 $\quad T_d(if \ x = a \ then \ Node2 \ Leaf \ b \ Leaf \ else$
 $\quad \quad if \ x = b \ then \ Node2 \ Leaf \ a \ Leaf$

```

      else Node3 Leaf a Leaf b Leaf) |
del x (Node2 l a r) =
  (case cmp x a of
    LT ⇒ node21 (del x l) a r |
    GT ⇒ node22 l a (del x r) |
    EQ ⇒ let (a',t) = del_min r in node22 l a' t) |
del x (Node3 l a m b r) =
  (case cmp x a of
    LT ⇒ node31 (del x l) a m b r |
    EQ ⇒ let (a',m') = del_min m in node32 l a' m' b r |
    GT ⇒
      (case cmp x b of
        LT ⇒ node32 l a (del x m) b r |
        EQ ⇒ let (b',r') = del_min r in node33 l a m b' r' |
        GT ⇒ node33 l a m b (del x r)))

```

definition `delete :: 'a::cmp ⇒ 'a tree23 ⇒ 'a tree23` **where**
`delete x t = tree_d(del x t)`

18.1 Functional Correctness

18.1.1 Proofs for isin

lemma `sorted(inorder t) ⇒ isin t x = (x ∈ elems (inorder t))`
by (`induction t`) (`auto simp: elems_simps1 ball_Un`)

lemma `isin_set: sorted(inorder t) ⇒ isin t x = (x ∈ elems (inorder t))`
by (`induction t`) (`auto simp: elems_simps2`)

18.1.2 Proofs for insert

lemma `inorder_ins:`
`sorted(inorder t) ⇒ inorder(tree_i(ins x t)) = ins_list x (inorder t)`
by(`induction t`) (`auto simp: ins_list_simps split: up_i.splits`)

lemma `inorder_insert:`
`sorted(inorder t) ⇒ inorder(insert a t) = ins_list a (inorder t)`
by(`simp add: insert_def inorder_ins`)

18.1.3 Proofs for delete

lemma `inorder_node21: height r > 0 ⇒`
`inorder (tree_d (node21 l' a r)) = inorder (tree_d l') @ a # inorder r`
by(`induct l' a r rule: node21.induct`) `auto`

lemma *inorder_node22*: $\text{height } l > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node22 } l \ a \ r')) = \text{inorder } l @ a \# \text{inorder } (\text{tree}_d \ r')$
by(*induct* $l \ a \ r'$ *rule*: *node22.induct*) *auto*

lemma *inorder_node31*: $\text{height } m > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node31 } l' \ a \ m \ b \ r)) = \text{inorder } (\text{tree}_d \ l') @ a \# \text{inorder } m$
 $@ b \# \text{inorder } r$
by(*induct* $l' \ a \ m \ b \ r$ *rule*: *node31.induct*) *auto*

lemma *inorder_node32*: $\text{height } r > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node32 } l \ a \ m' \ b \ r)) = \text{inorder } l @ a \# \text{inorder } (\text{tree}_d \ m')$
 $@ b \# \text{inorder } r$
by(*induct* $l \ a \ m' \ b \ r$ *rule*: *node32.induct*) *auto*

lemma *inorder_node33*: $\text{height } m > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node33 } l \ a \ m \ b \ r')) = \text{inorder } l @ a \# \text{inorder } m @ b \#$
 $\text{inorder } (\text{tree}_d \ r')$
by(*induct* $l \ a \ m \ b \ r'$ *rule*: *node33.induct*) *auto*

lemmas *inorder_nodes* = *inorder_node21* *inorder_node22*
inorder_node31 *inorder_node32* *inorder_node33*

lemma *del_minD*:
 $\text{del_min } t = (x, t') \implies \text{bal } t \implies \text{height } t > 0 \implies$
 $x \# \text{inorder}(\text{tree}_d \ t') = \text{inorder } t$
by(*induction* t *arbitrary*: t' *rule*: *del_min.induct*)
(auto simp: inorder_nodes split: prod.splits)

lemma *inorder_del*: $\llbracket \text{bal } t ; \text{sorted}(\text{inorder } t) \rrbracket \implies$
 $\text{inorder}(\text{tree}_d (\text{del } x \ t)) = \text{del_list } x (\text{inorder } t)$
by(*induction* t *rule*: *del.induct*)
(auto simp: del_list_simps inorder_nodes del_minD split: prod.splits)

lemma *inorder_delete*: $\llbracket \text{bal } t ; \text{sorted}(\text{inorder } t) \rrbracket \implies$
 $\text{inorder}(\text{delete } x \ t) = \text{del_list } x (\text{inorder } t)$
by(*simp* *add*: *delete_def* *inorder_del*)

18.2 Balancedness

18.2.1 Proofs for insert

First a standard proof that *ins* preserves *bal*.

instantiation $up_i :: (\text{type})\text{height}$
begin

```

fun height_up_i :: 'a up_i  $\Rightarrow$  nat where
  height (T_i t) = height t |
  height (Up_i l a r) = height l

```

```

instance ..

```

```

end

```

```

lemma bal_ins: bal t  $\Longrightarrow$  bal (tree_i(ins a t))  $\wedge$  height(ins a t) = height t
by (induct t) (auto split: up_i.split)

```

Now an alternative proof (by Brian Huffman) that runs faster because two properties (balance and height) are combined in one predicate.

```

inductive full :: nat  $\Rightarrow$  'a tree23  $\Rightarrow$  bool where
  full 0 Leaf |
   $\llbracket$ full n l; full n r $\rrbracket \Longrightarrow$  full (Suc n) (Node2 l p r) |
   $\llbracket$ full n l; full n m; full n r $\rrbracket \Longrightarrow$  full (Suc n) (Node3 l p m q r)

```

```

inductive_cases full_elims:

```

```

  full n Leaf
  full n (Node2 l p r)
  full n (Node3 l p m q r)

```

```

inductive_cases full_0_elim: full 0 t
inductive_cases full_Suc_elim: full (Suc n) t

```

```

lemma full_0_iff [simp]: full 0 t  $\longleftrightarrow$  t = Leaf
by (auto elim: full_0_elim intro: full.intros)

```

```

lemma full_Leaf_iff [simp]: full n Leaf  $\longleftrightarrow$  n = 0
by (auto elim: full_elims intro: full.intros)

```

```

lemma full_Suc_Node2_iff [simp]:
  full (Suc n) (Node2 l p r)  $\longleftrightarrow$  full n l  $\wedge$  full n r
by (auto elim: full_elims intro: full.intros)

```

```

lemma full_Suc_Node3_iff [simp]:
  full (Suc n) (Node3 l p m q r)  $\longleftrightarrow$  full n l  $\wedge$  full n m  $\wedge$  full n r
by (auto elim: full_elims intro: full.intros)

```

```

lemma full_imp_height: full n t  $\Longrightarrow$  height t = n
by (induct set: full, simp_all)

```

lemma *full_imp_bal*: $\text{full } n \ t \implies \text{bal } t$
by (*induct set*: *full*, *auto dest*: *full_imp_height*)

lemma *bal_imp_full*: $\text{bal } t \implies \text{full } (\text{height } t) \ t$
by (*induct t*, *simp_all*)

lemma *bal_iff_full*: $\text{bal } t \iff (\exists n. \text{full } n \ t)$
by (*auto elim!*: *bal_imp_full full_imp_bal*)

The *insert* function either preserves the height of the tree, or increases it by one. The constructor returned by the *insert* function determines which: A return value of the form $T_i \ t$ indicates that the height will be the same. A value of the form $Up_i \ l \ p \ r$ indicates an increase in height.

fun *full_i* :: $\text{nat} \Rightarrow 'a \ \text{up}_i \Rightarrow \text{bool}$ **where**
full_i $n \ (T_i \ t) \iff \text{full } n \ t \mid$
full_i $n \ (Up_i \ l \ p \ r) \iff \text{full } n \ l \wedge \text{full } n \ r$

lemma *full_i_ins*: $\text{full } n \ t \implies \text{full}_i \ n \ (\text{ins } a \ t)$
by (*induct rule*: *full.induct*) (*auto split*: *up_i.split*)

The *insert* operation preserves balance.

lemma *bal_insert*: $\text{bal } t \implies \text{bal } (\text{insert } a \ t)$
unfolding *bal_iff_full insert_def*
apply (*erule exE*)
apply (*drule full_i_ins* [*of* _ _ *a*])
apply (*cases ins a t*)
apply (*auto intro*: *full.intros*)
done

18.3 Proofs for delete

instantiation *up_d* :: $(\text{type})\text{height}$
begin

fun *height_up_d* :: $'a \ \text{up}_d \Rightarrow \text{nat}$ **where**
height $(T_d \ t) = \text{height } t \mid$
height $(Up_d \ t) = \text{height } t + 1$

instance ..

end

lemma *bal_tree_d_node21*:
 $\llbracket \text{bal } r; \text{bal } (\text{tree}_d \ l'); \text{height } r = \text{height } l' \rrbracket \implies \text{bal } (\text{tree}_d \ (\text{node21 } l' \ a \ r))$

by(*induct l' a r rule: node21.induct*) *auto*

lemma *bal_tree_d_node22*:

$\llbracket \text{bal}(\text{tree}_d\ r'); \text{bal}\ l; \text{height}\ r' = \text{height}\ l \rrbracket \implies \text{bal}(\text{tree}_d(\text{node22}\ l\ a\ r'))$
by(*induct l a r' rule: node22.induct*) *auto*

lemma *bal_tree_d_node31*:

$\llbracket \text{bal}(\text{tree}_d\ l'); \text{bal}\ m; \text{bal}\ r; \text{height}\ l' = \text{height}\ r; \text{height}\ m = \text{height}\ r \rrbracket$
 $\implies \text{bal}(\text{tree}_d(\text{node31}\ l'\ a\ m\ b\ r))$
by(*induct l' a m b r rule: node31.induct*) *auto*

lemma *bal_tree_d_node32*:

$\llbracket \text{bal}\ l; \text{bal}(\text{tree}_d\ m'); \text{bal}\ r; \text{height}\ l = \text{height}\ r; \text{height}\ m' = \text{height}\ r \rrbracket$
 $\implies \text{bal}(\text{tree}_d(\text{node32}\ l\ a\ m'\ b\ r))$
by(*induct l a m' b r rule: node32.induct*) *auto*

lemma *bal_tree_d_node33*:

$\llbracket \text{bal}\ l; \text{bal}\ m; \text{bal}(\text{tree}_d\ r'); \text{height}\ l = \text{height}\ r'; \text{height}\ m = \text{height}\ r' \rrbracket$
 $\implies \text{bal}(\text{tree}_d(\text{node33}\ l\ a\ m\ b\ r'))$
by(*induct l a m b r' rule: node33.induct*) *auto*

lemmas *bals = bal_tree_d_node21 bal_tree_d_node22*
bal_tree_d_node31 bal_tree_d_node32 bal_tree_d_node33

lemma *height'_node21*:

$\text{height}\ r > 0 \implies \text{height}(\text{node21}\ l'\ a\ r) = \max(\text{height}\ l')(\text{height}\ r) + 1$
by(*induct l' a r rule: node21.induct*)(*simp_all*)

lemma *height'_node22*:

$\text{height}\ l > 0 \implies \text{height}(\text{node22}\ l\ a\ r') = \max(\text{height}\ l)(\text{height}\ r') + 1$
by(*induct l a r' rule: node22.induct*)(*simp_all*)

lemma *height'_node31*:

$\text{height}\ m > 0 \implies \text{height}(\text{node31}\ l\ a\ m\ b\ r) =$
 $\max(\text{height}\ l)(\max(\text{height}\ m)(\text{height}\ r)) + 1$
by(*induct l a m b r rule: node31.induct*)(*simp_all add: max_def*)

lemma *height'_node32*:

$\text{height}\ r > 0 \implies \text{height}(\text{node32}\ l\ a\ m\ b\ r) =$
 $\max(\text{height}\ l)(\max(\text{height}\ m)(\text{height}\ r)) + 1$
by(*induct l a m b r rule: node32.induct*)(*simp_all add: max_def*)

lemma *height'_node33*:

$\text{height}\ m > 0 \implies \text{height}(\text{node33}\ l\ a\ m\ b\ r) =$

$\max (\text{height } l) (\max (\text{height } m) (\text{height } r)) + 1$
by(*induct l a m b r rule: node33.induct*)(*simp-all add: max_def*)

lemmas *heights = height'_node21 height'_node22*
height'_node31 height'_node32 height'_node33

lemma *height_del_min:*
 $\text{del_min } t = (x, t') \implies \text{height } t > 0 \implies \text{bal } t \implies \text{height } t' = \text{height } t$
by(*induct t arbitrary: x t' rule: del_min.induct*)
(auto simp: heights split: prod.splits)

lemma *height_del:* $\text{bal } t \implies \text{height}(\text{del } x \ t) = \text{height } t$
by(*induction x t rule: del.induct*)
(auto simp: heights max_def height_del_min split: prod.splits)

lemma *bal_del_min:*
 $\llbracket \text{del_min } t = (x, t'); \text{bal } t; \text{height } t > 0 \rrbracket \implies \text{bal } (\text{tree}_d \ t')$
by(*induct t arbitrary: x t' rule: del_min.induct*)
(auto simp: heights height_del_min bals split: prod.splits)

lemma *bal_tree_d_del:* $\text{bal } t \implies \text{bal}(\text{tree}_d(\text{del } x \ t))$
by(*induction x t rule: del.induct*)
(auto simp: bals bal_del_min height_del height_del_min split: prod.splits)

corollary *bal_delete:* $\text{bal } t \implies \text{bal}(\text{delete } x \ t)$
by(*simp add: delete_def bal_tree_d_del*)

18.4 Overall Correctness

interpretation *Set_by_Ordered*
where *empty = Leaf and isin = isin and insert = insert and delete = delete*
and *inorder = inorder and inv = bal*
proof (*standard, goal_cases*)
 case 2 **thus** ?*case* **by**(*simp add: isin_set*)
next
 case 3 **thus** ?*case* **by**(*simp add: inorder_insert*)
next
 case 4 **thus** ?*case* **by**(*simp add: inorder_delete*)
next
 case 6 **thus** ?*case* **by**(*simp add: bal_insert*)
next
 case 7 **thus** ?*case* **by**(*simp add: bal_delete*)
qed *simp+*

end

19 2-3 Tree Implementation of Maps

theory *Tree23_Map*

imports

Tree23_Set

Map_by_Ordered

begin

fun *lookup* :: ('a::cmp * 'b) tree23 $\Rightarrow$ 'a $\Rightarrow$ 'b option **where**

lookup Leaf *x* = None |

lookup (Node2 *l* (*a*,*b*) *r*) *x* = (case *cmp* *x* *a* of

LT $\Rightarrow$ *lookup* *l* *x* |

GT $\Rightarrow$ *lookup* *r* *x* |

EQ $\Rightarrow$ Some *b*) |

lookup (Node3 *l* (*a1*,*b1*) *m* (*a2*,*b2*) *r*) *x* = (case *cmp* *x* *a1* of

LT $\Rightarrow$ *lookup* *l* *x* |

EQ $\Rightarrow$ Some *b1* |

GT $\Rightarrow$ (case *cmp* *x* *a2* of

LT $\Rightarrow$ *lookup* *m* *x* |

EQ $\Rightarrow$ Some *b2* |

GT $\Rightarrow$ *lookup* *r* *x*))

fun *upd* :: 'a::cmp $\Rightarrow$ 'b $\Rightarrow$ ('a*'b) tree23 $\Rightarrow$ ('a*'b) up_{*i*} **where**

upd *x* *y* Leaf = Up_{*i*} Leaf (*x*,*y*) Leaf |

upd *x* *y* (Node2 *l* *ab* *r*) = (case *cmp* *x* (fst *ab*) of

LT $\Rightarrow$ (case *upd* *x* *y* *l* of

T_i *l'* $\Rightarrow$ *T_i* (Node2 *l'* *ab* *r*)

| Up_{*i*} *l1* *ab'* *l2* $\Rightarrow$ *T_i* (Node3 *l1* *ab'* *l2* *ab* *r*)) |

EQ $\Rightarrow$ *T_i* (Node2 *l* (*x*,*y*) *r*) |

GT $\Rightarrow$ (case *upd* *x* *y* *r* of

T_i *r'* $\Rightarrow$ *T_i* (Node2 *l* *ab* *r'*)

| Up_{*i*} *r1* *ab'* *r2* $\Rightarrow$ *T_i* (Node3 *l* *ab* *r1* *ab'* *r2*))) |

upd *x* *y* (Node3 *l* *ab1* *m* *ab2* *r*) = (case *cmp* *x* (fst *ab1*) of

LT $\Rightarrow$ (case *upd* *x* *y* *l* of

T_i *l'* $\Rightarrow$ *T_i* (Node3 *l'* *ab1* *m* *ab2* *r*)

| Up_{*i*} *l1* *ab'* *l2* $\Rightarrow$ Up_{*i*} (Node2 *l1* *ab'* *l2*) *ab1* (Node2 *m* *ab2* *r*)) |

EQ $\Rightarrow$ *T_i* (Node3 *l* (*x*,*y*) *m* *ab2* *r*) |

GT $\Rightarrow$ (case *cmp* *x* (fst *ab2*) of

LT $\Rightarrow$ (case *upd* *x* *y* *m* of

T_i *m'* $\Rightarrow$ *T_i* (Node3 *l* *ab1* *m'* *ab2* *r*))

$$\begin{aligned}
& | \text{Up}_i \text{ m1 ab' m2} \Rightarrow \text{Up}_i (\text{Node2 l ab1 m1}) \text{ ab' (Node2 m2} \\
& \text{ab2 r)}) | \\
& \text{EQ} \Rightarrow \text{T}_i (\text{Node3 l ab1 m (x,y) r}) | \\
& \text{GT} \Rightarrow (\text{case upd x y r of} \\
& \quad \text{T}_i \text{ r'} \Rightarrow \text{T}_i (\text{Node3 l ab1 m ab2 r'}) \\
& \quad | \text{Up}_i \text{ r1 ab' r2} \Rightarrow \text{Up}_i (\text{Node2 l ab1 m}) \text{ ab2 (Node2 r1 ab' r2)}))
\end{aligned}$$

definition $\text{update} :: 'a::\text{cmp} \Rightarrow 'b \Rightarrow ('a*'b) \text{ tree23} \Rightarrow ('a*'b) \text{ tree23}$ **where**
 $\text{update a b t} = \text{tree}_i(\text{upd a b t})$

fun $\text{del} :: 'a::\text{cmp} \Rightarrow ('a*'b) \text{ tree23} \Rightarrow ('a*'b) \text{ up}_d$ **where**
 $\text{del x Leaf} = \text{T}_d \text{ Leaf} |$
 $\text{del x (Node2 Leaf ab1 Leaf)} = (\text{if } x=\text{fst ab1} \text{ then } \text{Up}_d \text{ Leaf else } \text{T}_d(\text{Node2 Leaf ab1 Leaf})) |$
 $\text{del x (Node3 Leaf ab1 Leaf ab2 Leaf)} = \text{T}_d(\text{if } x=\text{fst ab1} \text{ then Node2 Leaf ab2 Leaf}$
 $\quad \text{else if } x=\text{fst ab2} \text{ then Node2 Leaf ab1 Leaf else Node3 Leaf ab1 Leaf ab2 Leaf}) |$
 $\text{del x (Node2 l ab1 r)} = (\text{case cmp x (fst ab1) of}$
 $\quad \text{LT} \Rightarrow \text{node21 (del x l) ab1 r} |$
 $\quad \text{GT} \Rightarrow \text{node22 l ab1 (del x r)} |$
 $\quad \text{EQ} \Rightarrow \text{let (ab1',t)} = \text{del_min r in node22 l ab1' t}) |$
 $\text{del x (Node3 l ab1 m ab2 r)} = (\text{case cmp x (fst ab1) of}$
 $\quad \text{LT} \Rightarrow \text{node31 (del x l) ab1 m ab2 r} |$
 $\quad \text{EQ} \Rightarrow \text{let (ab1',m')} = \text{del_min m in node32 l ab1' m' ab2 r} |$
 $\quad \text{GT} \Rightarrow (\text{case cmp x (fst ab2) of}$
 $\quad \quad \text{LT} \Rightarrow \text{node32 l ab1 (del x m) ab2 r} |$
 $\quad \quad \text{EQ} \Rightarrow \text{let (ab2',r')} = \text{del_min r in node33 l ab1 m ab2' r'}$
 $\quad \quad \text{GT} \Rightarrow \text{node33 l ab1 m ab2 (del x r)})$

definition $\text{delete} :: 'a::\text{cmp} \Rightarrow ('a*'b) \text{ tree23} \Rightarrow ('a*'b) \text{ tree23}$ **where**
 $\text{delete x t} = \text{tree}_d(\text{del x t})$

19.1 Functional Correctness

lemma lookup_map_of :

$\text{sorted1}(\text{inorder t}) \Longrightarrow \text{lookup t x} = \text{map_of (inorder t) x}$

by $(\text{induction t}) (\text{auto simp: map_of_sims split: option.split})$

lemma inorder_upd :

$\text{sorted1}(\text{inorder t}) \Longrightarrow \text{inorder}(\text{tree}_i(\text{upd x y t})) = \text{upd_list x y (inorder t)}$

by $(\text{induction t}) (\text{auto simp: upd_list_sims split: up}_i.\text{splits})$

corollary *inorder_update*:

$sorted1(inorder\ t) \implies inorder(update\ x\ y\ t) = upd_list\ x\ y\ (inorder\ t)$
by(*simp add: update_def inorder_upd*)

lemma *inorder_del*: $\llbracket bal\ t ; sorted1(inorder\ t) \rrbracket \implies$

$inorder(tree_d\ (del\ x\ t)) = del_list\ x\ (inorder\ t)$

by(*induction t rule: del.induct*)

(*auto simp: del_list_simps inorder_nodes del_minD split: prod.splits*)

corollary *inorder_delete*: $\llbracket bal\ t ; sorted1(inorder\ t) \rrbracket \implies$

$inorder(delete\ x\ t) = del_list\ x\ (inorder\ t)$

by(*simp add: delete_def inorder_del*)

19.2 Balancedness

lemma *bal_upd*: $bal\ t \implies bal\ (tree_i(upd\ x\ y\ t)) \wedge height(upd\ x\ y\ t) = height\ t$

by (*induct t*) (*auto split: up_i.split*)

corollary *bal_update*: $bal\ t \implies bal\ (update\ x\ y\ t)$

by (*simp add: update_def bal_upd*)

lemma *height_del*: $bal\ t \implies height(del\ x\ t) = height\ t$

by(*induction x t rule: del.induct*)

(*auto simp add: heights_max_def height_del_min split: prod.split*)

lemma *bal_tree_d_del*: $bal\ t \implies bal(tree_d(del\ x\ t))$

by(*induction x t rule: del.induct*)

(*auto simp: bals bal_del_min height_del height_del_min split: prod.split*)

corollary *bal_delete*: $bal\ t \implies bal(delete\ x\ t)$

by(*simp add: delete_def bal_tree_d_del*)

19.3 Overall Correctness

interpretation *Map_by_Ordered*

where *empty* = *Leaf* **and** *lookup* = *lookup* **and** *update* = *update* **and** *delete* = *delete*

and *inorder* = *inorder* **and** *inv* = *bal*

proof (*standard, goal_cases*)

case 2 **thus** ?*case* **by**(*simp add: lookup_map_of*)

```

next
  case 3 thus ?case by(simp add: inorder_update)
next
  case 4 thus ?case by(simp add: inorder_delete)
next
  case 6 thus ?case by(simp add: bal_update)
next
  case 7 thus ?case by(simp add: bal_delete)
qed simp+

end

```

20 2-3-4 Trees

```

theory Tree234
imports Main
begin

```

```

class height =
fixes height :: 'a  $\Rightarrow$  nat

```

```

datatype 'a tree234 =
  Leaf ( $\langle \rangle$ ) |
  Node2 'a tree234 'a 'a tree234 ( $\langle -, -, - \rangle$ ) |
  Node3 'a tree234 'a 'a tree234 'a 'a tree234 ( $\langle -, -, -, - \rangle$ ) |
  Node4 'a tree234 'a 'a tree234 'a 'a tree234 'a 'a tree234
    ( $\langle -, -, -, -, -, - \rangle$ )

```

```

fun inorder :: 'a tree234  $\Rightarrow$  'a list where
inorder Leaf = [] |
inorder(Node2 l a r) = inorder l @ a # inorder r |
inorder(Node3 l a m b r) = inorder l @ a # inorder m @ b # inorder r |
inorder(Node4 l a m b n c r) = inorder l @ a # inorder m @ b # inorder
n @ c # inorder r

```

```

instantiation tree234 :: (type)height
begin

```

```

fun height_tree234 :: 'a tree234  $\Rightarrow$  nat where
height Leaf = 0 |
height (Node2 l _ r) = Suc(max (height l) (height r)) |
height (Node3 l _ m _ r) = Suc(max (height l) (max (height m) (height r)))

```

```
|
height (Node4 l _ m _ n _ r) = Suc(max (height l) (max (height m) (max
(height n) (height r))))
```

```
instance ..
```

```
end
```

Balanced:

```
fun bal :: 'a tree234 ⇒ bool where
bal Leaf = True |
bal (Node2 l _ r) = (bal l & bal r & height l = height r) |
bal (Node3 l _ m _ r) = (bal l & bal m & bal r & height l = height m &
height m = height r) |
bal (Node4 l _ m _ n _ r) = (bal l & bal m & bal n & bal r & height l =
height m & height m = height n & height n = height r)
```

```
end
```

21 2-3-4 Tree Implementation of Sets

```
theory Tree234_Set
```

```
imports
```

```
Tree234
```

```
Cmp
```

```
../Data_Structures/Set_by_Ordered
```

```
begin
```

21.1 Set operations on 2-3-4 trees

```
fun isin :: 'a::cmp tree234 ⇒ 'a ⇒ bool where
isin Leaf x = False |
isin (Node2 l a r) x =
(case cmp x a of LT ⇒ isin l x | EQ ⇒ True | GT ⇒ isin r x) |
isin (Node3 l a m b r) x =
(case cmp x a of LT ⇒ isin l x | EQ ⇒ True | GT ⇒ (case cmp x b of
LT ⇒ isin m x | EQ ⇒ True | GT ⇒ isin r x)) |
isin (Node4 t1 a t2 b t3 c t4) x =
(case cmp x b of
LT ⇒
(case cmp x a of
LT ⇒ isin t1 x |
EQ ⇒ True |
GT ⇒ isin t2 x) |
```

```

EQ ⇒ True |
GT ⇒
  (case cmp x c of
    LT ⇒ isin t3 x |
    EQ ⇒ True |
    GT ⇒ isin t4 x))

datatype 'a upi = Ti 'a tree234 | Upi 'a tree234 'a 'a tree234

fun treei :: 'a upi ⇒ 'a tree234 where
treei (Ti t) = t |
treei (Upi l a r) = Node2 l a r

fun ins :: 'a::cmp ⇒ 'a tree234 ⇒ 'a upi where
ins x Leaf = Upi Leaf x Leaf |
ins x (Node2 l a r) =
  (case cmp x a of
    LT ⇒ (case ins x l of
      Ti l' => Ti (Node2 l' a r)
      | Upi l1 b l2 => Ti (Node3 l1 b l2 a r)) |
    EQ ⇒ Ti (Node2 l x r) |
    GT ⇒ (case ins x r of
      Ti r' => Ti (Node2 l a r')
      | Upi r1 b r2 => Ti (Node3 l a r1 b r2))) |
ins x (Node3 l a m b r) =
  (case cmp x a of
    LT ⇒ (case ins x l of
      Ti l' => Ti (Node3 l' a m b r)
      | Upi l1 c l2 => Upi (Node2 l1 c l2) a (Node2 m b r)) |
    EQ ⇒ Ti (Node3 l a m b r) |
    GT ⇒ (case cmp x b of
      GT ⇒ (case ins x r of
        Ti r' => Ti (Node3 l a m b r')
        | Upi r1 c r2 => Upi (Node2 l a m) b (Node2 r1 c r2)) |
      EQ ⇒ Ti (Node3 l a m b r) |
      LT ⇒ (case ins x m of
        Ti m' => Ti (Node3 l a m' b r)
        | Upi m1 c m2 => Upi (Node2 l a m1) c (Node2 m2 b
r)))) |
ins x (Node4 t1 a t2 b t3 c t4) =
  (case cmp x b of
    LT ⇒
      (case cmp x a of
        LT ⇒

```

```

      (case ins x t1 of
        Ti t => Ti (Node4 t a t2 b t3 c t4) |
        Upi l y r => Upi (Node2 l y r) a (Node3 t2 b t3 c t4)) |
    EQ => Ti (Node4 t1 a t2 b t3 c t4) |
    GT =>
      (case ins x t2 of
        Ti t => Ti (Node4 t1 a t b t3 c t4) |
        Upi l y r => Upi (Node2 t1 a l) y (Node3 r b t3 c t4))) |
    EQ => Ti (Node4 t1 a t2 b t3 c t4) |
    GT =>
      (case cmp x c of
        LT =>
          (case ins x t3 of
            Ti t => Ti (Node4 t1 a t2 b t c t4) |
            Upi l y r => Upi (Node2 t1 a t2) b (Node3 l y r c t4)) |
          EQ => Ti (Node4 t1 a t2 b t3 c t4) |
          GT =>
            (case ins x t4 of
              Ti t => Ti (Node4 t1 a t2 b t3 c t) |
              Upi l y r => Upi (Node2 t1 a t2) b (Node3 t3 c l y r))))

```

hide_const insert

definition insert :: 'a::cmp => 'a tree234 => 'a tree234 **where**
 insert x t = tree_i(ins x t)

datatype 'a up_d = T_d 'a tree234 | Up_d 'a tree234

fun tree_d :: 'a up_d => 'a tree234 **where**
 tree_d (T_d t) = t |
 tree_d (Up_d t) = t

fun node21 :: 'a up_d => 'a => 'a tree234 => 'a up_d **where**
 node21 (T_d l) a r = T_d(Node2 l a r) |
 node21 (Up_d l) a (Node2 lr b rr) = Up_d(Node3 l a lr b rr) |
 node21 (Up_d l) a (Node3 lr b mr c rr) = T_d(Node2 (Node2 l a lr) b (Node2 mr c rr)) |
 node21 (Up_d t1) a (Node4 t2 b t3 c t4 d t5) = T_d(Node2 (Node2 t1 a t2) b (Node3 t3 c t4 d t5))

fun node22 :: 'a tree234 => 'a => 'a up_d => 'a up_d **where**
 node22 l a (T_d r) = T_d(Node2 l a r) |
 node22 (Node2 ll b rl) a (Up_d r) = Up_d(Node3 ll b rl a r) |
 node22 (Node3 ll b ml c rl) a (Up_d r) = T_d(Node2 (Node2 ll b ml) c (Node2

$rl\ a\ r)) \mid$
 $node22\ (Node4\ t1\ a\ t2\ b\ t3\ c\ t4)\ d\ (Up_d\ t5) = T_d(Node2\ (Node2\ t1\ a\ t2)\ b\ (Node3\ t3\ c\ t4\ d\ t5))$

fun $node31 :: 'a\ up_d \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a\ up_d$ **where**
 $node31\ (T_d\ t1)\ a\ t2\ b\ t3 = T_d(Node3\ t1\ a\ t2\ b\ t3) \mid$
 $node31\ (Up_d\ t1)\ a\ (Node2\ t2\ b\ t3)\ c\ t4 = T_d(Node2\ (Node3\ t1\ a\ t2\ b\ t3)\ c\ t4) \mid$
 $node31\ (Up_d\ t1)\ a\ (Node3\ t2\ b\ t3\ c\ t4)\ d\ t5 = T_d(Node3\ (Node2\ t1\ a\ t2)\ b\ (Node2\ t3\ c\ t4)\ d\ t5) \mid$
 $node31\ (Up_d\ t1)\ a\ (Node4\ t2\ b\ t3\ c\ t4\ d\ t5)\ e\ t6 = T_d(Node3\ (Node2\ t1\ a\ t2)\ b\ (Node3\ t3\ c\ t4\ d\ t5)\ e\ t6)$

fun $node32 :: 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ up_d \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a\ up_d$ **where**
 $node32\ t1\ a\ (T_d\ t2)\ b\ t3 = T_d(Node3\ t1\ a\ t2\ b\ t3) \mid$
 $node32\ t1\ a\ (Up_d\ t2)\ b\ (Node2\ t3\ c\ t4) = T_d(Node2\ t1\ a\ (Node3\ t2\ b\ t3\ c\ t4)) \mid$
 $node32\ t1\ a\ (Up_d\ t2)\ b\ (Node3\ t3\ c\ t4\ d\ t5) = T_d(Node3\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Node2\ t4\ d\ t5)) \mid$
 $node32\ t1\ a\ (Up_d\ t2)\ b\ (Node4\ t3\ c\ t4\ d\ t5\ e\ t6) = T_d(Node3\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Node3\ t4\ d\ t5\ e\ t6))$

fun $node33 :: 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ up_d \Rightarrow 'a\ up_d$ **where**
 $node33\ l\ a\ m\ b\ (T_d\ r) = T_d(Node3\ l\ a\ m\ b\ r) \mid$
 $node33\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Up_d\ t4) = T_d(Node2\ t1\ a\ (Node3\ t2\ b\ t3\ c\ t4)) \mid$
 $node33\ t1\ a\ (Node3\ t2\ b\ t3\ c\ t4)\ d\ (Up_d\ t5) = T_d(Node3\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Node2\ t4\ d\ t5)) \mid$
 $node33\ t1\ a\ (Node4\ t2\ b\ t3\ c\ t4\ d\ t5)\ e\ (Up_d\ t6) = T_d(Node3\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Node3\ t4\ d\ t5\ e\ t6))$

fun $node41 :: 'a\ up_d \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a\ up_d$ **where**
 $node41\ (T_d\ t1)\ a\ t2\ b\ t3\ c\ t4 = T_d(Node4\ t1\ a\ t2\ b\ t3\ c\ t4) \mid$
 $node41\ (Up_d\ t1)\ a\ (Node2\ t2\ b\ t3)\ c\ t4\ d\ t5 = T_d(Node3\ (Node3\ t1\ a\ t2\ b\ t3)\ c\ t4\ d\ t5) \mid$
 $node41\ (Up_d\ t1)\ a\ (Node3\ t2\ b\ t3\ c\ t4)\ d\ t5\ e\ t6 = T_d(Node4\ (Node2\ t1\ a\ t2)\ b\ (Node2\ t3\ c\ t4)\ d\ t5\ e\ t6) \mid$
 $node41\ (Up_d\ t1)\ a\ (Node4\ t2\ b\ t3\ c\ t4\ d\ t5)\ e\ t6\ f\ t7 = T_d(Node4\ (Node2\ t1\ a\ t2)\ b\ (Node3\ t3\ c\ t4\ d\ t5)\ e\ t6\ f\ t7)$

fun $node42 :: 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ up_d \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a\ up_d$ **where**
 $node42\ t1\ a\ (T_d\ t2)\ b\ t3\ c\ t4 = T_d(Node4\ t1\ a\ t2\ b\ t3\ c\ t4) \mid$

$node42 (Node2\ t1\ a\ t2)\ b\ (Up_d\ t3)\ c\ t4\ d\ t5 = T_d(Node3\ (Node3\ t1\ a\ t2\ b\ t3)\ c\ t4\ d\ t5) \mid$
 $node42 (Node3\ t1\ a\ t2\ b\ t3)\ c\ (Up_d\ t4)\ d\ t5\ e\ t6 = T_d(Node4\ (Node2\ t1\ a\ t2)\ b\ (Node2\ t3\ c\ t4)\ d\ t5\ e\ t6) \mid$
 $node42 (Node4\ t1\ a\ t2\ b\ t3\ c\ t4)\ d\ (Up_d\ t5)\ e\ t6\ f\ t7 = T_d(Node4\ (Node2\ t1\ a\ t2)\ b\ (Node3\ t3\ c\ t4\ d\ t5)\ e\ t6\ f\ t7)$

fun $node43 :: 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ up_d \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a\ up_d$ **where**
 $node43\ t1\ a\ t2\ b\ (T_d\ t3)\ c\ t4 = T_d(Node4\ t1\ a\ t2\ b\ t3\ c\ t4) \mid$
 $node43\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Up_d\ t4)\ d\ t5 = T_d(Node3\ t1\ a\ (Node3\ t2\ b\ t3\ c\ t4)\ d\ t5) \mid$
 $node43\ t1\ a\ (Node3\ t2\ b\ t3\ c\ t4)\ d\ (Up_d\ t5)\ e\ t6 = T_d(Node4\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Node2\ t4\ d\ t5)\ e\ t6) \mid$
 $node43\ t1\ a\ (Node4\ t2\ b\ t3\ c\ t4\ d\ t5)\ e\ (Up_d\ t6)\ f\ t7 = T_d(Node4\ t1\ a\ (Node2\ t2\ b\ t3)\ c\ (Node3\ t4\ d\ t5\ e\ t6)\ f\ t7)$

fun $node44 :: 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ tree234 \Rightarrow 'a \Rightarrow 'a\ up_d \Rightarrow 'a\ up_d$ **where**
 $node44\ t1\ a\ t2\ b\ t3\ c\ (T_d\ t4) = T_d(Node4\ t1\ a\ t2\ b\ t3\ c\ t4) \mid$
 $node44\ t1\ a\ t2\ b\ (Node2\ t3\ c\ t4)\ d\ (Up_d\ t5) = T_d(Node3\ t1\ a\ t2\ b\ (Node3\ t3\ c\ t4\ d\ t5)) \mid$
 $node44\ t1\ a\ t2\ b\ (Node3\ t3\ c\ t4\ d\ t5)\ e\ (Up_d\ t6) = T_d(Node4\ t1\ a\ t2\ b\ (Node2\ t3\ c\ t4)\ d\ (Node2\ t5\ e\ t6)) \mid$
 $node44\ t1\ a\ t2\ b\ (Node4\ t3\ c\ t4\ d\ t5\ e\ t6)\ f\ (Up_d\ t7) = T_d(Node4\ t1\ a\ t2\ b\ (Node2\ t3\ c\ t4)\ d\ (Node3\ t5\ e\ t6\ f\ t7))$

fun $del_min :: 'a\ tree234 \Rightarrow 'a * 'a\ up_d$ **where**
 $del_min\ (Node2\ Leaf\ a\ Leaf) = (a,\ Up_d\ Leaf) \mid$
 $del_min\ (Node3\ Leaf\ a\ Leaf\ b\ Leaf) = (a,\ T_d(Node2\ Leaf\ b\ Leaf)) \mid$
 $del_min\ (Node4\ Leaf\ a\ Leaf\ b\ Leaf\ c\ Leaf) = (a,\ T_d(Node3\ Leaf\ b\ Leaf\ c\ Leaf)) \mid$
 $del_min\ (Node2\ l\ a\ r) = (let\ (x,l') = del_min\ l\ in\ (x,\ node21\ l'\ a\ r)) \mid$
 $del_min\ (Node3\ l\ a\ m\ b\ r) = (let\ (x,l') = del_min\ l\ in\ (x,\ node31\ l'\ a\ m\ b\ r)) \mid$
 $del_min\ (Node4\ l\ a\ m\ b\ n\ c\ r) = (let\ (x,l') = del_min\ l\ in\ (x,\ node41\ l'\ a\ m\ b\ n\ c\ r))$

fun $del :: 'a::cmp \Rightarrow 'a\ tree234 \Rightarrow 'a\ up_d$ **where**
 $del\ k\ Leaf = T_d\ Leaf \mid$
 $del\ k\ (Node2\ Leaf\ p\ Leaf) = (if\ k=p\ then\ Up_d\ Leaf\ else\ T_d(Node2\ Leaf\ p\ Leaf)) \mid$
 $del\ k\ (Node3\ Leaf\ p\ Leaf\ q\ Leaf) = T_d(if\ k=p\ then\ Node2\ Leaf\ q\ Leaf\ else\ if\ k=q\ then\ Node2\ Leaf\ p\ Leaf\ else\ Node3\ Leaf\ p\ Leaf\ q\ Leaf) \mid$

$del\ k\ (Node4\ Leaf\ a\ Leaf\ b\ Leaf\ c\ Leaf) =$
 $T_d(if\ k=a\ then\ Node3\ Leaf\ b\ Leaf\ c\ Leaf\ else$
 $\quad if\ k=b\ then\ Node3\ Leaf\ a\ Leaf\ c\ Leaf\ else$
 $\quad if\ k=c\ then\ Node3\ Leaf\ a\ Leaf\ b\ Leaf$
 $\quad else\ Node4\ Leaf\ a\ Leaf\ b\ Leaf\ c\ Leaf) \mid$
 $del\ k\ (Node2\ l\ a\ r) = (case\ cmp\ k\ a\ of$
 $\quad LT \Rightarrow node21\ (del\ k\ l)\ a\ r \mid$
 $\quad GT \Rightarrow node22\ l\ a\ (del\ k\ r) \mid$
 $\quad EQ \Rightarrow let\ (a',t) = del_min\ r\ in\ node22\ l\ a'\ t) \mid$
 $del\ k\ (Node3\ l\ a\ m\ b\ r) = (case\ cmp\ k\ a\ of$
 $\quad LT \Rightarrow node31\ (del\ k\ l)\ a\ m\ b\ r \mid$
 $\quad EQ \Rightarrow let\ (a',m') = del_min\ m\ in\ node32\ l\ a'\ m'\ b\ r \mid$
 $\quad GT \Rightarrow (case\ cmp\ k\ b\ of$
 $\quad\quad LT \Rightarrow node32\ l\ a\ (del\ k\ m)\ b\ r \mid$
 $\quad\quad EQ \Rightarrow let\ (b',r') = del_min\ r\ in\ node33\ l\ a\ m\ b'\ r' \mid$
 $\quad\quad GT \Rightarrow node33\ l\ a\ m\ b\ (del\ k\ r))) \mid$
 $del\ k\ (Node4\ l\ a\ m\ b\ n\ c\ r) = (case\ cmp\ k\ b\ of$
 $\quad LT \Rightarrow (case\ cmp\ k\ a\ of$
 $\quad\quad LT \Rightarrow node41\ (del\ k\ l)\ a\ m\ b\ n\ c\ r \mid$
 $\quad\quad EQ \Rightarrow let\ (a',m') = del_min\ m\ in\ node42\ l\ a'\ m'\ b\ n\ c\ r \mid$
 $\quad\quad GT \Rightarrow node42\ l\ a\ (del\ k\ m)\ b\ n\ c\ r) \mid$
 $\quad EQ \Rightarrow let\ (b',n') = del_min\ n\ in\ node43\ l\ a\ m\ b'\ n'\ c\ r \mid$
 $\quad GT \Rightarrow (case\ cmp\ k\ c\ of$
 $\quad\quad LT \Rightarrow node43\ l\ a\ m\ b\ (del\ k\ n)\ c\ r \mid$
 $\quad\quad EQ \Rightarrow let\ (c',r') = del_min\ r\ in\ node44\ l\ a\ m\ b\ n\ c'\ r' \mid$
 $\quad\quad GT \Rightarrow node44\ l\ a\ m\ b\ n\ c\ (del\ k\ r)))$

definition $delete :: 'a::cmp \Rightarrow 'a\ tree_{234} \Rightarrow 'a\ tree_{234}$ **where**
 $delete\ x\ t = tree_d(del\ x\ t)$

21.2 Functional correctness

21.2.1 Functional correctness of isin:

lemma $sorted(inorder\ t) \Longrightarrow isin\ t\ x = (x \in elems\ (inorder\ t))$
by $(induction\ t)\ (auto\ simp: elems_simps1\ ball_Un)$

lemma $isin_set: sorted(inorder\ t) \Longrightarrow isin\ t\ x = (x \in elems\ (inorder\ t))$
by $(induction\ t)\ (auto\ simp: elems_simps2)$

21.2.2 Functional correctness of insert:

lemma $inorder_ins:$
 $sorted(inorder\ t) \Longrightarrow inorder(tree_i(ins\ x\ t)) = ins_list\ x\ (inorder\ t)$
by $(induction\ t)\ (auto,\ auto\ simp: ins_list_simps\ split: up_i.splits)$

lemma *inorder_insert*:

$\text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{insert } a \ t) = \text{ins_list } a \ (\text{inorder } t)$

by(*simp add: insert_def inorder_ins*)

21.2.3 Functional correctness of delete

lemma *inorder_node21*: $\text{height } r > 0 \implies$

$\text{inorder } (\text{tree}_d (\text{node21 } l' \ a \ r)) = \text{inorder } (\text{tree}_d \ l') @ a \ \# \ \text{inorder } r$

by(*induct l' a r rule: node21.induct*) *auto*

lemma *inorder_node22*: $\text{height } l > 0 \implies$

$\text{inorder } (\text{tree}_d (\text{node22 } l \ a \ r')) = \text{inorder } l @ a \ \# \ \text{inorder } (\text{tree}_d \ r')$

by(*induct l a r' rule: node22.induct*) *auto*

lemma *inorder_node31*: $\text{height } m > 0 \implies$

$\text{inorder } (\text{tree}_d (\text{node31 } l' \ a \ m \ b \ r)) = \text{inorder } (\text{tree}_d \ l') @ a \ \# \ \text{inorder } m @ b \ \# \ \text{inorder } r$

by(*induct l' a m b r rule: node31.induct*) *auto*

lemma *inorder_node32*: $\text{height } r > 0 \implies$

$\text{inorder } (\text{tree}_d (\text{node32 } l \ a \ m' \ b \ r)) = \text{inorder } l @ a \ \# \ \text{inorder } (\text{tree}_d \ m') @ b \ \# \ \text{inorder } r$

by(*induct l a m' b r rule: node32.induct*) *auto*

lemma *inorder_node33*: $\text{height } m > 0 \implies$

$\text{inorder } (\text{tree}_d (\text{node33 } l \ a \ m \ b \ r')) = \text{inorder } l @ a \ \# \ \text{inorder } m @ b \ \# \ \text{inorder } (\text{tree}_d \ r')$

by(*induct l a m b r' rule: node33.induct*) *auto*

lemma *inorder_node41*: $\text{height } m > 0 \implies$

$\text{inorder } (\text{tree}_d (\text{node41 } l' \ a \ m \ b \ n \ c \ r)) = \text{inorder } (\text{tree}_d \ l') @ a \ \# \ \text{inorder } m @ b \ \# \ \text{inorder } n @ c \ \# \ \text{inorder } r$

by(*induct l' a m b n c r rule: node41.induct*) *auto*

lemma *inorder_node42*: $\text{height } l > 0 \implies$

$\text{inorder } (\text{tree}_d (\text{node42 } l \ a \ m \ b \ n \ c \ r)) = \text{inorder } l @ a \ \# \ \text{inorder } (\text{tree}_d \ m) @ b \ \# \ \text{inorder } n @ c \ \# \ \text{inorder } r$

by(*induct l a m b n c r rule: node42.induct*) *auto*

lemma *inorder_node43*: $\text{height } m > 0 \implies$

$\text{inorder } (\text{tree}_d (\text{node43 } l \ a \ m \ b \ n \ c \ r)) = \text{inorder } l @ a \ \# \ \text{inorder } m @ b \ \# \ \text{inorder } (\text{tree}_d \ n) @ c \ \# \ \text{inorder } r$

by(*induct l a m b n c r rule: node43.induct*) *auto*

lemma *inorder_node44*: $\text{height } n > 0 \implies$
 $\text{inorder } (\text{tree}_d (\text{node44 } l \ a \ m \ b \ n \ c \ r)) = \text{inorder } l \ @ \ a \ \# \ \text{inorder } m \ @ \ b$
 $\# \ \text{inorder } n \ @ \ c \ \# \ \text{inorder } (\text{tree}_d \ r)$
by(*induct* $l \ a \ m \ b \ n \ c \ r$ *rule*: *node44.induct*) *auto*

lemmas *inorder_nodes* = *inorder_node21* *inorder_node22*
inorder_node31 *inorder_node32* *inorder_node33*
inorder_node41 *inorder_node42* *inorder_node43* *inorder_node44*

lemma *del_minD*:
 $\text{del_min } t = (x, t') \implies \text{bal } t \implies \text{height } t > 0 \implies$
 $x \# \text{inorder}(\text{tree}_d \ t') = \text{inorder } t$
by(*induction* t *arbitrary*: t' *rule*: *del_min.induct*)
(auto simp: inorder_nodes split: prod.splits)

lemma *inorder_del*: $\llbracket \text{bal } t ; \text{sorted}(\text{inorder } t) \rrbracket \implies$
 $\text{inorder}(\text{tree}_d (\text{del } x \ t)) = \text{del_list } x \ (\text{inorder } t)$
by(*induction* t *rule*: *del.induct*)
(auto simp: inorder_nodes del_list_simps del_minD split: prod.splits)

lemma *inorder_delete*: $\llbracket \text{bal } t ; \text{sorted}(\text{inorder } t) \rrbracket \implies$
 $\text{inorder}(\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$
by(*simp* *add*: *delete_def* *inorder_del*)

21.3 Balancedness

21.3.1 Proofs for insert

First a standard proof that *ins* preserves *bal*.

instantiation $up_i :: (\text{type})\text{height}$
begin

fun *height_up_i* :: $'a \ up_i \Rightarrow \text{nat}$ **where**
 $\text{height } (T_i \ t) = \text{height } t \mid$
 $\text{height } (Up_i \ l \ a \ r) = \text{height } l$

instance ..

end

lemma *bal.ins*: $\text{bal } t \implies \text{bal } (\text{tree}_i(\text{ins } a \ t)) \wedge \text{height}(\text{ins } a \ t) = \text{height } t$
by (*induct* t) (*auto*, *auto split: up_i.split*)

Now an alternative proof (by Brian Huffman) that runs faster because two properties (balance and height) are combined in one predicate.

```
inductive full :: nat => 'a tree234 => bool where
  full 0 Leaf |
   $\llbracket \text{full } n \text{ } l; \text{full } n \text{ } r \rrbracket \implies \text{full } (\text{Suc } n) (\text{Node2 } l \text{ } p \text{ } r) \mid$ 
   $\llbracket \text{full } n \text{ } l; \text{full } n \text{ } m; \text{full } n \text{ } r \rrbracket \implies \text{full } (\text{Suc } n) (\text{Node3 } l \text{ } p \text{ } m \text{ } q \text{ } r) \mid$ 
   $\llbracket \text{full } n \text{ } l; \text{full } n \text{ } m; \text{full } n \text{ } m'; \text{full } n \text{ } r \rrbracket \implies \text{full } (\text{Suc } n) (\text{Node4 } l \text{ } p \text{ } m \text{ } q \text{ } m' \text{ } q' \text{ } r)$ 
```

```
inductive_cases full_elims:
```

```
  full n Leaf
  full n (Node2 l p r)
  full n (Node3 l p m q r)
  full n (Node4 l p m q m' q' r)
```

```
inductive_cases full_0_elim: full 0 t
```

```
inductive_cases full_Suc_elim: full (Suc n) t
```

```
lemma full_0_iff [simp]: full 0 t  $\longleftrightarrow$  t = Leaf
```

```
  by (auto elim: full_0_elim intro: full.intros)
```

```
lemma full_Leaf_iff [simp]: full n Leaf  $\longleftrightarrow$  n = 0
```

```
  by (auto elim: full_elims intro: full.intros)
```

```
lemma full_Suc_Node2_iff [simp]:
```

```
  full (Suc n) (Node2 l p r)  $\longleftrightarrow$  full n l  $\wedge$  full n r
```

```
  by (auto elim: full_elims intro: full.intros)
```

```
lemma full_Suc_Node3_iff [simp]:
```

```
  full (Suc n) (Node3 l p m q r)  $\longleftrightarrow$  full n l  $\wedge$  full n m  $\wedge$  full n r
```

```
  by (auto elim: full_elims intro: full.intros)
```

```
lemma full_Suc_Node4_iff [simp]:
```

```
  full (Suc n) (Node4 l p m q m' q' r)  $\longleftrightarrow$  full n l  $\wedge$  full n m  $\wedge$  full n m'  $\wedge$  full n r
```

```
  by (auto elim: full_elims intro: full.intros)
```

```
lemma full_imp_height: full n t  $\implies$  height t = n
```

```
  by (induct set: full, simp_all)
```

```
lemma full_imp_bal: full n t  $\implies$  bal t
```

```
  by (induct set: full, auto dest: full_imp_height)
```

lemma *bal_imp_full*: $\text{bal } t \implies \text{full } (\text{height } t) \ t$
by (*induct* *t*, *simp_all*)

lemma *bal_iff_full*: $\text{bal } t \longleftrightarrow (\exists n. \text{full } n \ t)$
by (*auto elim!*: *bal_imp_full full_imp_bal*)

The *insert* function either preserves the height of the tree, or increases it by one. The constructor returned by the *insert* function determines which: A return value of the form $T_i \ t$ indicates that the height will be the same. A value of the form $Up_i \ l \ p \ r$ indicates an increase in height.

primrec *full_i* :: $\text{nat} \Rightarrow 'a \ \text{up}_i \Rightarrow \text{bool}$ **where**
full_i *n* ($T_i \ t$) $\longleftrightarrow \text{full } n \ t \mid$
full_i *n* ($Up_i \ l \ p \ r$) $\longleftrightarrow \text{full } n \ l \wedge \text{full } n \ r$

lemma *full_i_ins*: $\text{full } n \ t \implies \text{full}_i \ n \ (\text{ins } a \ t)$
by (*induct rule*: *full.induct*) (*auto*, *auto split*: *up_i.split*)

The *insert* operation preserves balance.

lemma *bal_insert*: $\text{bal } t \implies \text{bal } (\text{insert } a \ t)$
unfolding *bal_iff_full insert_def*
apply (*erule exE*)
apply (*drule full_i_ins* [*of* _ _ *a*])
apply (*cases ins a t*)
apply (*auto intro*: *full.intros*)
done

21.3.2 Proofs for delete

instantiation *up_d* :: $(\text{type})\text{height}$
begin

fun *height_{up_d}* :: $'a \ \text{up}_d \Rightarrow \text{nat}$ **where**
height ($T_d \ t$) = *height* *t* $\mid$
height ($Up_d \ t$) = *height* *t* + 1

instance ..

end

lemma *bal_tree_d_node21*:
 $\llbracket \text{bal } r; \text{bal } (\text{tree}_d \ l); \text{height } r = \text{height } l \rrbracket \implies \text{bal } (\text{tree}_d \ (\text{node21 } l \ a \ r))$
by(*induct l a r rule*: *node21.induct*) *auto*

lemma *bal_tree_d_node22*:

$\llbracket \text{bal}(\text{tree}_d\ r); \text{bal}\ l; \text{height}\ r = \text{height}\ l \rrbracket \implies \text{bal}\ (\text{tree}_d\ (\text{node22}\ l\ a\ r))$
by(*induct l a r rule: node22.induct*) *auto*

lemma *bal_tree_d_node31*:
 $\llbracket \text{bal}\ (\text{tree}_d\ l); \text{bal}\ m; \text{bal}\ r; \text{height}\ l = \text{height}\ r; \text{height}\ m = \text{height}\ r \rrbracket$
 $\implies \text{bal}\ (\text{tree}_d\ (\text{node31}\ l\ a\ m\ b\ r))$
by(*induct l a m b r rule: node31.induct*) *auto*

lemma *bal_tree_d_node32*:
 $\llbracket \text{bal}\ l; \text{bal}\ (\text{tree}_d\ m); \text{bal}\ r; \text{height}\ l = \text{height}\ r; \text{height}\ m = \text{height}\ r \rrbracket$
 $\implies \text{bal}\ (\text{tree}_d\ (\text{node32}\ l\ a\ m\ b\ r))$
by(*induct l a m b r rule: node32.induct*) *auto*

lemma *bal_tree_d_node33*:
 $\llbracket \text{bal}\ l; \text{bal}\ m; \text{bal}(\text{tree}_d\ r); \text{height}\ l = \text{height}\ r; \text{height}\ m = \text{height}\ r \rrbracket$
 $\implies \text{bal}\ (\text{tree}_d\ (\text{node33}\ l\ a\ m\ b\ r))$
by(*induct l a m b r rule: node33.induct*) *auto*

lemma *bal_tree_d_node41*:
 $\llbracket \text{bal}\ (\text{tree}_d\ l); \text{bal}\ m; \text{bal}\ n; \text{bal}\ r; \text{height}\ l = \text{height}\ r; \text{height}\ m = \text{height}\ r; \text{height}\ n = \text{height}\ r \rrbracket$
 $\implies \text{bal}\ (\text{tree}_d\ (\text{node41}\ l\ a\ m\ b\ n\ c\ r))$
by(*induct l a m b n c r rule: node41.induct*) *auto*

lemma *bal_tree_d_node42*:
 $\llbracket \text{bal}\ l; \text{bal}\ (\text{tree}_d\ m); \text{bal}\ n; \text{bal}\ r; \text{height}\ l = \text{height}\ r; \text{height}\ m = \text{height}\ r; \text{height}\ n = \text{height}\ r \rrbracket$
 $\implies \text{bal}\ (\text{tree}_d\ (\text{node42}\ l\ a\ m\ b\ n\ c\ r))$
by(*induct l a m b n c r rule: node42.induct*) *auto*

lemma *bal_tree_d_node43*:
 $\llbracket \text{bal}\ l; \text{bal}\ m; \text{bal}\ (\text{tree}_d\ n); \text{bal}\ r; \text{height}\ l = \text{height}\ r; \text{height}\ m = \text{height}\ r; \text{height}\ n = \text{height}\ r \rrbracket$
 $\implies \text{bal}\ (\text{tree}_d\ (\text{node43}\ l\ a\ m\ b\ n\ c\ r))$
by(*induct l a m b n c r rule: node43.induct*) *auto*

lemma *bal_tree_d_node44*:
 $\llbracket \text{bal}\ l; \text{bal}\ m; \text{bal}\ n; \text{bal}\ (\text{tree}_d\ r); \text{height}\ l = \text{height}\ r; \text{height}\ m = \text{height}\ r; \text{height}\ n = \text{height}\ r \rrbracket$
 $\implies \text{bal}\ (\text{tree}_d\ (\text{node44}\ l\ a\ m\ b\ n\ c\ r))$
by(*induct l a m b n c r rule: node44.induct*) *auto*

lemmas *bals* = *bal_tree_d_node21 bal_tree_d_node22*
bal_tree_d_node31 bal_tree_d_node32 bal_tree_d_node33

bal_tree_d-node41 bal_tree_d-node42 bal_tree_d-node43 bal_tree_d-node44

lemma *height_node21:*

$height\ r > 0 \implies height(node21\ l\ a\ r) = \max\ (height\ l)\ (height\ r) + 1$
by(*induct l a r rule: node21.induct*)(*simp_all add: max.assoc*)

lemma *height_node22:*

$height\ l > 0 \implies height(node22\ l\ a\ r) = \max\ (height\ l)\ (height\ r) + 1$
by(*induct l a r rule: node22.induct*)(*simp_all add: max.assoc*)

lemma *height_node31:*

$height\ m > 0 \implies height(node31\ l\ a\ m\ b\ r) =$
 $\max\ (height\ l)\ (\max\ (height\ m)\ (height\ r)) + 1$
by(*induct l a m b r rule: node31.induct*)(*simp_all add: max_def*)

lemma *height_node32:*

$height\ r > 0 \implies height(node32\ l\ a\ m\ b\ r) =$
 $\max\ (height\ l)\ (\max\ (height\ m)\ (height\ r)) + 1$
by(*induct l a m b r rule: node32.induct*)(*simp_all add: max_def*)

lemma *height_node33:*

$height\ m > 0 \implies height(node33\ l\ a\ m\ b\ r) =$
 $\max\ (height\ l)\ (\max\ (height\ m)\ (height\ r)) + 1$
by(*induct l a m b r rule: node33.induct*)(*simp_all add: max_def*)

lemma *height_node41:*

$height\ m > 0 \implies height(node41\ l\ a\ m\ b\ n\ c\ r) =$
 $\max\ (height\ l)\ (\max\ (height\ m)\ (\max\ (height\ n)\ (height\ r))) + 1$
by(*induct l a m b n c r rule: node41.induct*)(*simp_all add: max_def*)

lemma *height_node42:*

$height\ l > 0 \implies height(node42\ l\ a\ m\ b\ n\ c\ r) =$
 $\max\ (height\ l)\ (\max\ (height\ m)\ (\max\ (height\ n)\ (height\ r))) + 1$
by(*induct l a m b n c r rule: node42.induct*)(*simp_all add: max_def*)

lemma *height_node43:*

$height\ m > 0 \implies height(node43\ l\ a\ m\ b\ n\ c\ r) =$
 $\max\ (height\ l)\ (\max\ (height\ m)\ (\max\ (height\ n)\ (height\ r))) + 1$
by(*induct l a m b n c r rule: node43.induct*)(*simp_all add: max_def*)

lemma *height_node44:*

$height\ n > 0 \implies height(node44\ l\ a\ m\ b\ n\ c\ r) =$
 $\max\ (height\ l)\ (\max\ (height\ m)\ (\max\ (height\ n)\ (height\ r))) + 1$
by(*induct l a m b n c r rule: node44.induct*)(*simp_all add: max_def*)

lemmas *heights* = *height_node21* *height_node22*
height_node31 *height_node32* *height_node33*
height_node41 *height_node42* *height_node43* *height_node44*

lemma *height_del_min*:
 $\text{del_min } t = (x, t') \implies \text{height } t > 0 \implies \text{bal } t \implies \text{height } t' = \text{height } t$
by(*induct* *t* *arbitrary*: *x* *t'* *rule*: *del_min.induct*)
(auto *simp*: *heights* *split*: *prod.splits*)

lemma *height_del*: $\text{bal } t \implies \text{height}(\text{del } x \ t) = \text{height } t$
by(*induction* *x* *t* *rule*: *del.induct*)
(auto *simp* *add*: *heights* *height_del_min* *split*: *prod.split*)

lemma *bal_del_min*:
 $\llbracket \text{del_min } t = (x, t'); \text{bal } t; \text{height } t > 0 \rrbracket \implies \text{bal } (\text{tree}_d \ t')$
by(*induct* *t* *arbitrary*: *x* *t'* *rule*: *del_min.induct*)
(auto *simp*: *heights* *height_del_min* *bals* *split*: *prod.splits*)

lemma *bal_tree_d_del*: $\text{bal } t \implies \text{bal}(\text{tree}_d(\text{del } x \ t))$
by(*induction* *x* *t* *rule*: *del.induct*)
(auto *simp*: *bals* *bal_del_min* *height_del* *height_del_min* *split*: *prod.split*)

corollary *bal_delete*: $\text{bal } t \implies \text{bal}(\text{delete } x \ t)$
by(*simp* *add*: *delete_def* *bal_tree_d_del*)

21.4 Overall Correctness

interpretation *Set.by_Ordered*
where *empty* = *Leaf* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* =
delete
and *inorder* = *inorder* **and** *inv* = *bal*
proof (*standard*, *goal_cases*)
 case 2 **thus** ?*case* **by**(*simp* *add*: *isin_set*)
next
 case 3 **thus** ?*case* **by**(*simp* *add*: *inorder_insert*)
next
 case 4 **thus** ?*case* **by**(*simp* *add*: *inorder_delete*)
next
 case 6 **thus** ?*case* **by**(*simp* *add*: *bal_insert*)
next
 case 7 **thus** ?*case* **by**(*simp* *add*: *bal_delete*)
qed *simp*+

end

22 2-3-4 Tree Implementation of Maps

```

theory Tree234_Map
imports
  Tree234_Set
  ../Data_Structures/Map_by_Ordered
begin

```

22.1 Map operations on 2-3-4 trees

```

fun lookup :: ('a::cmp * 'b) tree234  $\Rightarrow$  'a  $\Rightarrow$  'b option where
  lookup Leaf x = None |
  lookup (Node2 l (a,b) r) x = (case cmp x a of
    LT  $\Rightarrow$  lookup l x |
    GT  $\Rightarrow$  lookup r x |
    EQ  $\Rightarrow$  Some b) |
  lookup (Node3 l (a1,b1) m (a2,b2) r) x = (case cmp x a1 of
    LT  $\Rightarrow$  lookup l x |
    EQ  $\Rightarrow$  Some b1 |
    GT  $\Rightarrow$  (case cmp x a2 of
      LT  $\Rightarrow$  lookup m x |
      EQ  $\Rightarrow$  Some b2 |
      GT  $\Rightarrow$  lookup r x)) |
  lookup (Node4 t1 (a1,b1) t2 (a2,b2) t3 (a3,b3) t4) x = (case cmp x a2 of
    LT  $\Rightarrow$  (case cmp x a1 of
      LT  $\Rightarrow$  lookup t1 x | EQ  $\Rightarrow$  Some b1 | GT  $\Rightarrow$  lookup t2 x) |
    EQ  $\Rightarrow$  Some b2 |
    GT  $\Rightarrow$  (case cmp x a3 of
      LT  $\Rightarrow$  lookup t3 x | EQ  $\Rightarrow$  Some b3 | GT  $\Rightarrow$  lookup t4 x))

fun upd :: 'a::cmp  $\Rightarrow$  'b  $\Rightarrow$  ('a*'b) tree234  $\Rightarrow$  ('a*'b) upi where
  upd x y Leaf = Upi Leaf (x,y) Leaf |
  upd x y (Node2 l ab r) = (case cmp x (fst ab) of
    LT  $\Rightarrow$  (case upd x y l of
      Ti l'  $\Rightarrow$  Ti (Node2 l' ab r)
      | Upi l1 ab' l2  $\Rightarrow$  Ti (Node3 l1 ab' l2 ab r)) |
    EQ  $\Rightarrow$  Ti (Node2 l (x,y) r) |
    GT  $\Rightarrow$  (case upd x y r of
      Ti r'  $\Rightarrow$  Ti (Node2 l ab r')
      | Upi r1 ab' r2  $\Rightarrow$  Ti (Node3 l ab r1 ab' r2))) |
  upd x y (Node3 l ab1 m ab2 r) = (case cmp x (fst ab1) of

```

$$\begin{aligned}
& LT \Rightarrow (\text{case upd } x \ y \ l \text{ of} \\
& \quad T_i \ l' \Rightarrow T_i \ (\text{Node3 } l' \ ab1 \ m \ ab2 \ r) \\
& \quad | \ Up_i \ l1 \ ab' \ l2 \Rightarrow \ Up_i \ (\text{Node2 } l1 \ ab' \ l2) \ ab1 \ (\text{Node2 } m \ ab2 \ r)) \ | \\
& EQ \Rightarrow T_i \ (\text{Node3 } l \ (x,y) \ m \ ab2 \ r) \ | \\
& GT \Rightarrow (\text{case cmp } x \ (\text{fst } ab2) \text{ of} \\
& \quad LT \Rightarrow (\text{case upd } x \ y \ m \text{ of} \\
& \quad \quad T_i \ m' \Rightarrow T_i \ (\text{Node3 } l \ ab1 \ m' \ ab2 \ r) \\
& \quad \quad | \ Up_i \ m1 \ ab' \ m2 \Rightarrow \ Up_i \ (\text{Node2 } l \ ab1 \ m1) \ ab' \ (\text{Node2 } m2 \\
& \quad \quad ab2 \ r)) \ | \\
& \quad EQ \Rightarrow T_i \ (\text{Node3 } l \ ab1 \ m \ (x,y) \ r) \ | \\
& \quad GT \Rightarrow (\text{case upd } x \ y \ r \text{ of} \\
& \quad \quad T_i \ r' \Rightarrow T_i \ (\text{Node3 } l \ ab1 \ m \ ab2 \ r') \\
& \quad \quad | \ Up_i \ r1 \ ab' \ r2 \Rightarrow \ Up_i \ (\text{Node2 } l \ ab1 \ m) \ ab2 \ (\text{Node2 } r1 \ ab' \\
& \quad \quad r2)))) \ | \\
& \text{upd } x \ y \ (\text{Node4 } t1 \ ab1 \ t2 \ ab2 \ t3 \ ab3 \ t4) = (\text{case cmp } x \ (\text{fst } ab2) \text{ of} \\
& \quad LT \Rightarrow (\text{case cmp } x \ (\text{fst } ab1) \text{ of} \\
& \quad \quad LT \Rightarrow (\text{case upd } x \ y \ t1 \text{ of} \\
& \quad \quad \quad T_i \ t1' \Rightarrow T_i \ (\text{Node4 } t1' \ ab1 \ t2 \ ab2 \ t3 \ ab3 \ t4) \\
& \quad \quad \quad | \ Up_i \ t11 \ q \ t12 \Rightarrow \ Up_i \ (\text{Node2 } t11 \ q \ t12) \ ab1 \ (\text{Node3 } t2 \ ab2 \\
& \quad \quad \quad t3 \ ab3 \ t4)) \ | \\
& \quad \quad EQ \Rightarrow T_i \ (\text{Node4 } t1 \ (x,y) \ t2 \ ab2 \ t3 \ ab3 \ t4) \ | \\
& \quad \quad GT \Rightarrow (\text{case upd } x \ y \ t2 \text{ of} \\
& \quad \quad \quad T_i \ t2' \Rightarrow T_i \ (\text{Node4 } t1 \ ab1 \ t2' \ ab2 \ t3 \ ab3 \ t4) \\
& \quad \quad \quad | \ Up_i \ t21 \ q \ t22 \Rightarrow \ Up_i \ (\text{Node2 } t1 \ ab1 \ t21) \ q \ (\text{Node3 } t22 \ ab2 \\
& \quad \quad \quad t3 \ ab3 \ t4))) \ | \\
& \quad EQ \Rightarrow T_i \ (\text{Node4 } t1 \ ab1 \ t2 \ (x,y) \ t3 \ ab3 \ t4) \ | \\
& \quad GT \Rightarrow (\text{case cmp } x \ (\text{fst } ab3) \text{ of} \\
& \quad \quad LT \Rightarrow (\text{case upd } x \ y \ t3 \text{ of} \\
& \quad \quad \quad T_i \ t3' \Rightarrow T_i \ (\text{Node4 } t1 \ ab1 \ t2 \ ab2 \ t3' \ ab3 \ t4) \\
& \quad \quad \quad | \ Up_i \ t31 \ q \ t32 \Rightarrow \ Up_i \ (\text{Node2 } t1 \ ab1 \ t2) \ ab2(*q*) \ (\text{Node3 } \\
& \quad \quad \quad t31 \ q \ t32 \ ab3 \ t4)) \ | \\
& \quad \quad EQ \Rightarrow T_i \ (\text{Node4 } t1 \ ab1 \ t2 \ ab2 \ t3 \ (x,y) \ t4) \ | \\
& \quad \quad GT \Rightarrow (\text{case upd } x \ y \ t4 \text{ of} \\
& \quad \quad \quad T_i \ t4' \Rightarrow T_i \ (\text{Node4 } t1 \ ab1 \ t2 \ ab2 \ t3 \ ab3 \ t4') \\
& \quad \quad \quad | \ Up_i \ t41 \ q \ t42 \Rightarrow \ Up_i \ (\text{Node2 } t1 \ ab1 \ t2) \ ab2 \ (\text{Node3 } t3 \ ab3 \\
& \quad \quad \quad t41 \ q \ t42))))))
\end{aligned}$$

definition $\text{update} :: 'a::\text{cmp} \Rightarrow 'b \Rightarrow ('a*'b) \text{tree234} \Rightarrow ('a*'b) \text{tree234}$
where
 $\text{update } x \ y \ t = \text{tree}_i(\text{upd } x \ y \ t)$

fun $\text{del} :: 'a::\text{cmp} \Rightarrow ('a*'b) \text{tree234} \Rightarrow ('a*'b) \text{up}_d$ **where**
 $\text{del } x \ \text{Leaf} = T_d \ \text{Leaf} \ |$
 $\text{del } x \ (\text{Node2 } \text{Leaf } ab1 \ \text{Leaf}) = (\text{if } x = \text{fst } ab1 \text{ then } \text{Up}_d \ \text{Leaf} \text{ else } T_d(\text{Node2}$

$Leaf\ ab1\ Leaf)) \mid$
 $del\ x\ (Node3\ Leaf\ ab1\ Leaf\ ab2\ Leaf) = T_d(\text{if } x=fst\ ab1\ \text{then } Node2\ Leaf\ ab2\ Leaf$
 $\text{else if } x=fst\ ab2\ \text{then } Node2\ Leaf\ ab1\ Leaf\ \text{else } Node3\ Leaf\ ab1\ Leaf\ ab2\ Leaf) \mid$
 $del\ x\ (Node4\ Leaf\ ab1\ Leaf\ ab2\ Leaf\ ab3\ Leaf) =$
 $T_d(\text{if } x = fst\ ab1\ \text{then } Node3\ Leaf\ ab2\ Leaf\ ab3\ Leaf\ \text{else}$
 $\text{if } x = fst\ ab2\ \text{then } Node3\ Leaf\ ab1\ Leaf\ ab3\ Leaf\ \text{else}$
 $\text{if } x = fst\ ab3\ \text{then } Node3\ Leaf\ ab1\ Leaf\ ab2\ Leaf$
 $\text{else } Node4\ Leaf\ ab1\ Leaf\ ab2\ Leaf\ ab3\ Leaf) \mid$
 $del\ x\ (Node2\ l\ ab1\ r) = (\text{case } cmp\ x\ (fst\ ab1)\ \text{of}$
 $LT \Rightarrow node21\ (del\ x\ l)\ ab1\ r \mid$
 $GT \Rightarrow node22\ l\ ab1\ (del\ x\ r) \mid$
 $EQ \Rightarrow \text{let } (ab1',t) = del_min\ r\ \text{in } node22\ l\ ab1'\ t) \mid$
 $del\ x\ (Node3\ l\ ab1\ m\ ab2\ r) = (\text{case } cmp\ x\ (fst\ ab1)\ \text{of}$
 $LT \Rightarrow node31\ (del\ x\ l)\ ab1\ m\ ab2\ r \mid$
 $EQ \Rightarrow \text{let } (ab1',m') = del_min\ m\ \text{in } node32\ l\ ab1'\ m'\ ab2\ r \mid$
 $GT \Rightarrow (\text{case } cmp\ x\ (fst\ ab2)\ \text{of}$
 $LT \Rightarrow node32\ l\ ab1\ (del\ x\ m)\ ab2\ r \mid$
 $EQ \Rightarrow \text{let } (ab2',r') = del_min\ r\ \text{in } node33\ l\ ab1\ m\ ab2'\ r' \mid$
 $GT \Rightarrow node33\ l\ ab1\ m\ ab2\ (del\ x\ r))) \mid$
 $del\ x\ (Node4\ t1\ ab1\ t2\ ab2\ t3\ ab3\ t4) = (\text{case } cmp\ x\ (fst\ ab2)\ \text{of}$
 $LT \Rightarrow (\text{case } cmp\ x\ (fst\ ab1)\ \text{of}$
 $LT \Rightarrow node41\ (del\ x\ t1)\ ab1\ t2\ ab2\ t3\ ab3\ t4 \mid$
 $EQ \Rightarrow \text{let } (ab',t2') = del_min\ t2\ \text{in } node42\ t1\ ab'\ t2'\ ab2\ t3\ ab3\ t4$
 $\mid$
 $GT \Rightarrow node42\ t1\ ab1\ (del\ x\ t2)\ ab2\ t3\ ab3\ t4) \mid$
 $EQ \Rightarrow \text{let } (ab',t3') = del_min\ t3\ \text{in } node43\ t1\ ab1\ t2\ ab'\ t3'\ ab3\ t4 \mid$
 $GT \Rightarrow (\text{case } cmp\ x\ (fst\ ab3)\ \text{of}$
 $LT \Rightarrow node43\ t1\ ab1\ t2\ ab2\ (del\ x\ t3)\ ab3\ t4 \mid$
 $EQ \Rightarrow \text{let } (ab',t4') = del_min\ t4\ \text{in } node44\ t1\ ab1\ t2\ ab2\ t3\ ab'\ t4' \mid$
 $GT \Rightarrow node44\ t1\ ab1\ t2\ ab2\ t3\ ab3\ (del\ x\ t4)))$

definition $delete :: 'a::cmp \Rightarrow ('a*'b) \ tree234 \Rightarrow ('a*'b) \ tree234$ **where**
 $delete\ x\ t = tree_d(del\ x\ t)$

22.2 Functional correctness

lemma $lookup_map_of$:

$sorted1(inorder\ t) \Longrightarrow lookup\ t\ x = map_of\ (inorder\ t)\ x$
by $(induction\ t)\ (auto\ simp: map_of_simps\ split: option.split)$

lemma $inorder_upd$:

$sorted1(inorder\ t) \implies inorder(tree_i(upd\ a\ b\ t)) = upd_list\ a\ b\ (inorder\ t)$
by(*induction* t)
 $(auto\ simp: upd_list_simps, auto\ simp: upd_list_simps\ split: up_i.splits)$

lemma *inorder_update*:
 $sorted1(inorder\ t) \implies inorder(update\ a\ b\ t) = upd_list\ a\ b\ (inorder\ t)$
by(*simp* *add*: *update_def* *inorder_upd*)

lemma *inorder_del*: $\llbracket bal\ t ; sorted1(inorder\ t) \rrbracket \implies$
 $inorder(tree_d\ (del\ x\ t)) = del_list\ x\ (inorder\ t)$
by(*induction* t *rule*: *del.induct*)
 $((auto\ simp: del_list_simps\ inorder_nodes\ del_minD\ split: prod.splits)[1])+$

lemma *inorder_delete*: $\llbracket bal\ t ; sorted1(inorder\ t) \rrbracket \implies$
 $inorder(delete\ x\ t) = del_list\ x\ (inorder\ t)$
by(*simp* *add*: *delete_def* *inorder_del*)

22.3 Balancedness

lemma *bal_upd*: $bal\ t \implies bal\ (tree_i(upd\ x\ y\ t)) \wedge height(upd\ x\ y\ t) = height\ t$
by (*induct* t) (*auto*, *auto* *split*: *up_i.split*)

lemma *bal_update*: $bal\ t \implies bal\ (update\ x\ y\ t)$
by (*simp* *add*: *update_def* *bal_upd*)

lemma *height_del*: $bal\ t \implies height(del\ x\ t) = height\ t$
by(*induction* $x\ t$ *rule*: *del.induct*)
 $(auto\ simp\ add: heights\ height_del_min\ split: prod.split)$

lemma *bal_tree_d_del*: $bal\ t \implies bal(tree_d(del\ x\ t))$
by(*induction* $x\ t$ *rule*: *del.induct*)
 $(auto\ simp: bals\ bal_del_min\ height_del\ height_del_min\ split: prod.split)$

corollary *bal_delete*: $bal\ t \implies bal(delete\ x\ t)$
by(*simp* *add*: *delete_def* *bal_tree_d_del*)

22.4 Overall Correctness

```
interpretation Map_by_Ordered
where empty = Leaf and lookup = lookup and update = update and delete
= delete
and inorder = inorder and inv = bal
proof (standard, goal_cases)
  case 2 thus ?case by(simp add: lookup_map_of)
next
  case 3 thus ?case by(simp add: inorder_update)
next
  case 4 thus ?case by(simp add: inorder_delete)
next
  case 6 thus ?case by(simp add: bal_update)
next
  case 7 thus ?case by(simp add: bal_delete)
qed simp+

end
```

23 1-2 Brother Tree Implementation of Sets

```
theory Brother12_Set
imports
  Cmp
  Set_by_Ordered
begin
```

23.1 Data Type and Operations

```
datatype 'a bro =
  N0 |
  N1 'a bro |
  N2 'a bro 'a 'a bro |

  L2 'a |
  N3 'a bro 'a 'a bro 'a 'a bro

fun inorder :: 'a bro  $\Rightarrow$  'a list where
inorder N0 = [] |
inorder (N1 t) = inorder t |
inorder (N2 l a r) = inorder l @ a # inorder r |
inorder (L2 a) = [a] |
inorder (N3 t1 a1 t2 a2 t3) = inorder t1 @ a1 # inorder t2 @ a2 # inorder
```

t3

```
fun isin :: 'a bro  $\Rightarrow$  'a::cmp  $\Rightarrow$  bool where  
isin N0 x = False |  
isin (N1 t) x = isin t x |  
isin (N2 l a r) x =  
  (case cmp x a of  
    LT  $\Rightarrow$  isin l x |  
    EQ  $\Rightarrow$  True |  
    GT  $\Rightarrow$  isin r x)
```

```
fun n1 :: 'a bro  $\Rightarrow$  'a bro where  
n1 (L2 a) = N2 N0 a N0 |  
n1 (N3 t1 a1 t2 a2 t3) = N2 (N2 t1 a1 t2) a2 (N1 t3) |  
n1 t = N1 t
```

hide.const (**open**) *insert*

locale *insert*
begin

```
fun n2 :: 'a bro  $\Rightarrow$  'a  $\Rightarrow$  'a bro  $\Rightarrow$  'a bro where  
n2 (L2 a1) a2 t = N3 N0 a1 N0 a2 t |  
n2 (N3 t1 a1 t2 a2 t3) a3 (N1 t4) = N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4) |  
n2 (N3 t1 a1 t2 a2 t3) a3 t4 = N3 (N2 t1 a1 t2) a2 (N1 t3) a3 t4 |  
n2 t1 a1 (L2 a2) = N3 t1 a1 N0 a2 N0 |  
n2 (N1 t1) a1 (N3 t2 a2 t3 a3 t4) = N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4) |  
n2 t1 a1 (N3 t2 a2 t3 a3 t4) = N3 t1 a1 (N1 t2) a2 (N2 t3 a3 t4) |  
n2 t1 a t2 = N2 t1 a t2
```

```
fun ins :: 'a::cmp  $\Rightarrow$  'a bro  $\Rightarrow$  'a bro where  
ins x N0 = L2 x |  
ins x (N1 t) = n1 (ins x t) |  
ins x (N2 l a r) =  
  (case cmp x a of  
    LT  $\Rightarrow$  n2 (ins x l) a r |  
    EQ  $\Rightarrow$  N2 l a r |  
    GT  $\Rightarrow$  n2 l a (ins x r))
```

```
fun tree :: 'a bro  $\Rightarrow$  'a bro where  
tree (L2 a) = N2 N0 a N0 |  
tree (N3 t1 a1 t2 a2 t3) = N2 (N2 t1 a1 t2) a2 (N1 t3) |  
tree t = t
```

definition *insert* :: 'a::cmp $\Rightarrow$ 'a bro $\Rightarrow$ 'a bro **where**
insert x t = tree(ins x t)

end

locale *delete*
begin

fun *n2* :: 'a bro $\Rightarrow$ 'a $\Rightarrow$ 'a bro $\Rightarrow$ 'a bro **where**
n2 (N1 t1) a1 (N1 t2) = N1 (N2 t1 a1 t2) |
n2 (N1 (N1 t1)) a1 (N2 (N1 t2) a2 (N2 t3 a3 t4)) =
 N1 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) |
n2 (N1 (N1 t1)) a1 (N2 (N2 t2 a2 t3) a3 (N1 t4)) =
 N1 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) |
n2 (N1 (N1 t1)) a1 (N2 (N2 t2 a2 t3) a3 (N2 t4 a4 t5)) =
 N2 (N2 (N1 t1) a1 (N2 t2 a2 t3)) a3 (N1 (N2 t4 a4 t5)) |
n2 (N2 (N1 t1) a1 (N2 t2 a2 t3)) a3 (N1 (N1 t4)) =
 N1 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) |
n2 (N2 (N2 t1 a1 t2) a2 (N1 t3)) a3 (N1 (N1 t4)) =
 N1 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) |
n2 (N2 (N2 t1 a1 t2) a2 (N2 t3 a3 t4)) a5 (N1 (N1 t5)) =
 N2 (N1 (N2 t1 a1 t2)) a2 (N2 (N2 t3 a3 t4) a5 (N1 t5)) |
n2 t1 a1 t2 = N2 t1 a1 t2

fun *del_min* :: 'a bro $\Rightarrow$ ('a $\times$ 'a bro) option **where**
del_min N0 = None |
del_min (N1 t) =
 (case *del_min* t of
 None $\Rightarrow$ None |
 Some (a, t') $\Rightarrow$ Some (a, N1 t')) |
del_min (N2 t1 a t2) =
 (case *del_min* t1 of
 None $\Rightarrow$ Some (a, N1 t2) |
 Some (b, t1') $\Rightarrow$ Some (b, n2 t1' a t2))

fun *del* :: 'a::cmp $\Rightarrow$ 'a bro $\Rightarrow$ 'a bro **where**
del _ N0 = N0 |
del x (N1 t) = N1 (del x t) |
del x (N2 l a r) =
 (case *cmp* x a of
 LT $\Rightarrow$ n2 (del x l) a r |
 GT $\Rightarrow$ n2 l a (del x r) |
 EQ $\Rightarrow$ (case *del_min* r of
 None $\Rightarrow$ N1 l |

$Some (b, r') \Rightarrow n2\ l\ b\ r')$

fun $tree :: 'a\ bro \Rightarrow 'a\ bro$ **where**
 $tree\ (N1\ t) = t$ |
 $tree\ t = t$

definition $delete :: 'a::cmp \Rightarrow 'a\ bro \Rightarrow 'a\ bro$ **where**
 $delete\ a\ t = tree\ (del\ a\ t)$

end

23.2 Invariants

fun $B :: nat \Rightarrow 'a\ bro\ set$
and $U :: nat \Rightarrow 'a\ bro\ set$ **where**
 $B\ 0 = \{N0\}$ |
 $B\ (Suc\ h) = \{ N2\ t1\ a\ t2 \mid t1\ a\ t2.$
 $t1 \in B\ h \cup U\ h \wedge t2 \in B\ h \vee t1 \in B\ h \wedge t2 \in B\ h \cup U\ h\} \mid$
 $U\ 0 = \{\}$ |
 $U\ (Suc\ h) = N1\ 'B\ h$

abbreviation $T\ h \equiv B\ h \cup U\ h$

fun $Bp :: nat \Rightarrow 'a\ bro\ set$ **where**
 $Bp\ 0 = B\ 0 \cup L2\ 'UNIV$ |
 $Bp\ (Suc\ 0) = B\ (Suc\ 0) \cup \{N3\ N0\ a\ N0\ b\ N0 \mid a\ b.\ True\}$ |
 $Bp\ (Suc\ (Suc\ h)) = B\ (Suc\ (Suc\ h)) \cup$
 $\{N3\ t1\ a\ t2\ b\ t3 \mid t1\ a\ t2\ b\ t3.\ t1 \in B\ (Suc\ h) \wedge t2 \in U\ (Suc\ h) \wedge t3 \in$
 $B\ (Suc\ h)\}$

fun $Um :: nat \Rightarrow 'a\ bro\ set$ **where**
 $Um\ 0 = \{\}$ |
 $Um\ (Suc\ h) = N1\ 'T\ h$

23.3 Functional Correctness Proofs

23.3.1 Proofs for isin

lemma

$t \in T\ h \Longrightarrow sorted(inorder\ t) \Longrightarrow isin\ t\ x = (x \in elems(inorder\ t))$
by(*induction h arbitrary: t*) (*fastforce simp: elems_simps1 split: if_splits*)+

lemma *isin_set*: $t \in T\ h \Longrightarrow$

$sorted(inorder\ t) \Longrightarrow isin\ t\ x = (x \in elems(inorder\ t))$
by(*induction h arbitrary: t*) (*auto simp: elems_simps2 split: if_splits*)

23.3.2 Proofs for insertion

lemma *inorder_n1*: $\text{inorder}(n1\ t) = \text{inorder}\ t$
by(*induction* *t* *rule*: *n1.induct*) (*auto simp*: *sorted_lems*)

context *insert*
begin

lemma *inorder_n2*: $\text{inorder}(n2\ l\ a\ r) = \text{inorder}\ l\ @\ a\ \# \text{inorder}\ r$
by(*cases* (*l,a,r*) *rule*: *n2.cases*) (*auto simp*: *sorted_lems*)

lemma *inorder_tree*: $\text{inorder}(\text{tree}\ t) = \text{inorder}\ t$
by(*cases* *t*) *auto*

lemma *inorder_ins*: $t \in T\ h \implies$
 $\text{sorted}(\text{inorder}\ t) \implies \text{inorder}(\text{ins}\ a\ t) = \text{ins_list}\ a\ (\text{inorder}\ t)$
by(*induction* *h* *arbitrary*: *t*) (*auto simp*: *ins_list_simps* *inorder_n1* *inorder_n2*)

lemma *inorder_insert*: $t \in T\ h \implies$
 $\text{sorted}(\text{inorder}\ t) \implies \text{inorder}(\text{insert}\ a\ t) = \text{ins_list}\ a\ (\text{inorder}\ t)$
by(*simp add*: *insert_def* *inorder_ins* *inorder_tree*)

end

23.3.3 Proofs for deletion

context *delete*
begin

lemma *inorder_tree*: $\text{inorder}(\text{tree}\ t) = \text{inorder}\ t$
by(*cases* *t*) *auto*

lemma *inorder_n2*: $\text{inorder}(n2\ l\ a\ r) = \text{inorder}\ l\ @\ a\ \# \text{inorder}\ r$
by(*induction* *l a r* *rule*: *n2.induct*) (*auto*)

lemma *inorder_del_min*:
 $t \in T\ h \implies (\text{del_min}\ t = \text{None} \longleftrightarrow \text{inorder}\ t = []) \wedge$
 $(\text{del_min}\ t = \text{Some}(a, t') \longrightarrow \text{inorder}\ t = a\ \# \text{inorder}\ t')$
by(*induction* *h* *arbitrary*: *t a t'*) (*auto simp*: *inorder_n2* *split*: *option.splits*)

lemma *inorder_del*:
 $t \in T\ h \implies \text{sorted}(\text{inorder}\ t) \implies \text{inorder}(\text{del}\ x\ t) = \text{del_list}\ x\ (\text{inorder}\ t)$
by(*induction* *h* *arbitrary*: *t*) (*auto simp*: *del_list_simps* *inorder_n2*)

inorder_del_min[*OF UnI1*] *inorder_del_min*[*OF UnI2*] *split*: *option.splits*)

lemma *inorder_delete*:

$t \in T\ h \implies \text{sorted}(\text{inorder}\ t) \implies \text{inorder}(\text{delete}\ x\ t) = \text{del_list}\ x\ (\text{inorder}\ t)$

by(*simp add: delete_def inorder_del inorder_tree*)

end

23.4 Invariant Proofs

23.4.1 Proofs for insertion

lemma *n1_type*: $t \in Bp\ h \implies n1\ t \in T\ (Suc\ h)$

by(*cases h rule: Bp.cases*) *auto*

context *insert*

begin

lemma *tree_type*: $t \in Bp\ h \implies \text{tree}\ t \in B\ h \cup B\ (Suc\ h)$

by(*cases h rule: Bp.cases*) *auto*

lemma *n2_type*:

$(t1 \in Bp\ h \wedge t2 \in T\ h \longrightarrow n2\ t1\ a\ t2 \in Bp\ (Suc\ h)) \wedge$

$(t1 \in T\ h \wedge t2 \in Bp\ h \longrightarrow n2\ t1\ a\ t2 \in Bp\ (Suc\ h))$

apply(*cases h rule: Bp.cases*)

apply (*auto*)[2]

apply(*rule conjI impI | erule conjE exE imageE | simp | erule disjE*)+

done

lemma *Bp_if_B*: $t \in B\ h \implies t \in Bp\ h$

by (*cases h rule: Bp.cases*) *simp_all*

An automatic proof:

lemma

$(t \in B\ h \longrightarrow \text{ins}\ x\ t \in Bp\ h) \wedge (t \in U\ h \longrightarrow \text{ins}\ x\ t \in T\ h)$

apply(*induction h arbitrary: t*)

apply (*simp*)

apply (*fastforce simp: Bp_if_B n2_type dest: n1_type*)

done

A detailed proof:

lemma *ins_type*:

shows $t \in B\ h \implies \text{ins}\ x\ t \in Bp\ h$ **and** $t \in U\ h \implies \text{ins}\ x\ t \in T\ h$

proof(*induction h arbitrary: t*)

```

case 0
{ case 1 thus ?case by simp
next
  case 2 thus ?case by simp }
next
case (Suc h)
{ case 1
  then obtain t1 a t2 where [simp]: t = N2 t1 a t2 and
    t1: t1 ∈ T h and t2: t2 ∈ T h and t12: t1 ∈ B h ∨ t2 ∈ B h
  by auto
  { assume x < a
    hence ?case ⟷ n2 (ins x t1) a t2 ∈ Bp (Suc h) by simp
    also have ...
    proof cases
      assume t1 ∈ B h
      with t2 show ?thesis by (simp add: Suc.IH(1) n2_type)
    next
      assume t1 ∉ B h
      hence 1: t1 ∈ U h and 2: t2 ∈ B h using t1 t12 by auto
      show ?thesis by (metis Suc.IH(2)[OF 1] Bp_if_B[OF 2] n2_type)
    qed
    finally have ?case .
  }
}
moreover
{ assume a < x
  hence ?case ⟷ n2 t1 a (ins x t2) ∈ Bp (Suc h) by simp
  also have ...
  proof cases
    assume t2 ∈ B h
    with t1 show ?thesis by (simp add: Suc.IH(1) n2_type)
  next
    assume t2 ∉ B h
    hence 1: t1 ∈ B h and 2: t2 ∈ U h using t2 t12 by auto
    show ?thesis by (metis Bp_if_B[OF 1] Suc.IH(2)[OF 2] n2_type)
  qed
}
}
moreover
{ assume x = a
  from 1 have t ∈ Bp (Suc h) by(rule Bp_if_B)
  hence ?case using ⟨x = a⟩ by simp
}
}
ultimately show ?case by auto
next
case 2 thus ?case using Suc(1) n1_type by fastforce }

```

qed

lemma *insert_type*:

$t \in B\ h \implies \text{insert } x\ t \in B\ h \cup B\ (\text{Suc } h)$

unfolding *insert_def* **by** (*metis ins_type(1) tree_type*)

end

23.4.2 Proofs for deletion

lemma *B_simps[simp]*:

$N1\ t \in B\ h = \text{False}$

$L2\ y \in B\ h = \text{False}$

$(N3\ t1\ a1\ t2\ a2\ t3) \in B\ h = \text{False}$

$N0 \in B\ h \longleftrightarrow h = 0$

by (*cases h, auto*)**+**

context *delete*

begin

lemma *n2_type1*:

$\llbracket t1 \in Um\ h; t2 \in B\ h \rrbracket \implies n2\ t1\ a\ t2 \in T\ (\text{Suc } h)$

apply(*cases h rule: Bp.cases*)

apply *auto*[2]

apply(*erule exE bexE conjE imageE | simp | erule disjE*)**+**

done

lemma *n2_type2*:

$\llbracket t1 \in B\ h; t2 \in Um\ h \rrbracket \implies n2\ t1\ a\ t2 \in T\ (\text{Suc } h)$

apply(*cases h rule: Bp.cases*)

apply *auto*[2]

apply(*erule exE bexE conjE imageE | simp | erule disjE*)**+**

done

lemma *n2_type3*:

$\llbracket t1 \in T\ h; t2 \in T\ h \rrbracket \implies n2\ t1\ a\ t2 \in T\ (\text{Suc } h)$

apply(*cases h rule: Bp.cases*)

apply *auto*[2]

apply(*erule exE bexE conjE imageE | simp | erule disjE*)**+**

done

lemma *del_minNoneN0*: $\llbracket t \in B\ h; \text{del_min } t = \text{None} \rrbracket \implies t = N0$

by (*cases t*) (*auto split: option.splits*)

lemma *del_minNoneN1* : $\llbracket t \in U\ h; \text{del_min}\ t = \text{None} \rrbracket \implies t = N1\ N0$
by (*cases h*) (*auto simp: del_minNoneN0 split: option.splits*)

lemma *del_min_type*:

$t \in B\ h \implies \text{del_min}\ t = \text{Some}\ (a, t') \implies t' \in T\ h$
 $t \in U\ h \implies \text{del_min}\ t = \text{Some}\ (a, t') \implies t' \in Um\ h$

proof (*induction h arbitrary: t a t'*)

case (*Suc h*)

{ **case** 1

then obtain *t1 a t2* **where** [*simp*]: $t = N2\ t1\ a\ t2$ **and**

t12: $t1 \in T\ h\ t2 \in T\ h\ t1 \in B\ h \vee t2 \in B\ h$

by *auto*

show *?case*

proof (*cases del_min t1*)

case *None*

show *?thesis*

proof *cases*

assume $t1 \in B\ h$

with *del_minNoneN0*[*OF this None*] 1 **show** *?thesis* **by** (*auto*)

next

assume $t1 \notin B\ h$

thus *?thesis* **using** 1 *None* **by** (*auto*)

qed

next

case [*simp*]: (*Some bt'*)

obtain *b t1'* **where** [*simp*]: $bt' = (b, t1')$ **by** *fastforce*

show *?thesis*

proof *cases*

assume $t1 \in B\ h$

from *Suc.IH*(1)[*OF this*] 1 **have** $t1' \in T\ h$ **by** *simp*

from *n2_type3*[*OF this t12*(2)] 1 **show** *?thesis* **by** *auto*

next

assume $t1 \notin B\ h$

hence *t1*: $t1 \in U\ h$ **and** *t2*: $t2 \in B\ h$ **using** *t12* **by** *auto*

from *Suc.IH*(2)[*OF t1*] **have** $t1' \in Um\ h$ **by** *simp*

from *n2_type1*[*OF this t2*] 1 **show** *?thesis* **by** *auto*

qed

qed

}

{ **case** 2

then obtain *t1* **where** [*simp*]: $t = N1\ t1$ **and** *t1*: $t1 \in B\ h$ **by** *auto*

show *?case*

proof (*cases del_min t1*)

case *None*

```

    with  $\text{del\_minNoneN0}[OF\ t1\ None]\ 2$  show  $?thesis$  by(auto)
  next
    case [simp]: (Some bt')
    obtain  $b\ t1'$  where [simp]:  $bt' = (b, t1')$  by fastforce
    from  $\text{Suc.IH}(1)[OF\ t1]$  have  $t1' \in T\ h$  by simp
    thus  $?thesis$  using 2 by auto
  qed
}
qed auto

lemma  $\text{del\_type}$ :
   $t \in B\ h \implies \text{del}\ x\ t \in T\ h$ 
   $t \in U\ h \implies \text{del}\ x\ t \in U\ m\ h$ 
proof (induction  $h$  arbitrary:  $x\ t$ )
  case (Suc  $h$ )
  { case 1
    then obtain  $l\ a\ r$  where [simp]:  $t = N2\ l\ a\ r$  and
       $lr: l \in T\ h\ r \in T\ h\ l \in B\ h \vee r \in B\ h$  by auto
    { assume  $x < a$ 
      have  $?case$ 
      proof cases
        assume  $l \in B\ h$ 
        from  $n2\_type3[OF\ \text{Suc.IH}(1)[OF\ this]\ lr(2)]$ 
        show  $?thesis$  using  $\langle x < a \rangle$  by(simp)
      next
        assume  $l \notin B\ h$ 
        hence  $l \in U\ h\ r \in B\ h$  using  $lr$  by auto
        from  $n2\_type1[OF\ \text{Suc.IH}(2)[OF\ this(1)]\ this(2)]$ 
        show  $?thesis$  using  $\langle x < a \rangle$  by(simp)
      qed
    } moreover
    { assume  $x > a$ 
      have  $?case$ 
      proof cases
        assume  $r \in B\ h$ 
        from  $n2\_type3[OF\ lr(1)\ \text{Suc.IH}(1)[OF\ this]]$ 
        show  $?thesis$  using  $\langle x > a \rangle$  by(simp)
      next
        assume  $r \notin B\ h$ 
        hence  $l \in B\ h\ r \in U\ h$  using  $lr$  by auto
        from  $n2\_type2[OF\ this(1)\ \text{Suc.IH}(2)[OF\ this(2)]]$ 
        show  $?thesis$  using  $\langle x > a \rangle$  by(simp)
      qed
    } moreover
  }

```

```

{ assume [simp]:  $x=a$ 
  have ?case
  proof (cases del_min r)
    case None
    show ?thesis
  proof cases
    assume  $r \in B\ h$ 
    with del_minNoneN0[OF this None] lr show ?thesis by (simp)
  next
    assume  $r \notin B\ h$ 
    hence  $r \in U\ h$  using lr by auto
    with del_minNoneN1[OF this None] lr(3) show ?thesis by (simp)
  qed
next
case [simp]: (Some br')
obtain b r' where [simp]:  $br' = (b, r')$  by fastforce
show ?thesis
proof cases
  assume  $r \in B\ h$ 
  from del_min_type(1)[OF this] n2_type3[OF lr(1)]
  show ?thesis by simp
next
  assume  $r \notin B\ h$ 
  hence  $l \in B\ h$  and  $r \in U\ h$  using lr by auto
  from del_min_type(2)[OF this(2)] n2_type2[OF this(1)]
  show ?thesis by simp
qed
qed
} ultimately show ?case by auto
}
{ case 2 with Suc.IH(1) show ?case by auto }
qed auto

```

lemma tree_type: $t \in T\ (h+1) \implies \text{tree } t : B\ (h+1) \cup B\ h$
by (auto)

lemma delete_type: $t \in B\ h \implies \text{delete } x\ t \in B\ h \cup B\ (h-1)$
unfolding delete_def
by (cases h) (simp, metis del_type(1) tree_type Suc_eq_plus1 diff_Suc_1)

end

23.5 Overall correctness

```

interpretation Set_by_Ordered
where empty = N0 and isin = isin and insert = insert.insert
and delete = delete.delete and inorder = inorder and inv =  $\lambda t. \exists h. t \in B\ h$ 
proof (standard, goal_cases)
  case 2 thus ?case by(auto intro!: isin_set)
next
  case 3 thus ?case by(auto intro!: insert.inorder_insert)
next
  case 4 thus ?case by(auto intro!: delete.inorder_delete)
next
  case 6 thus ?case using insert.insert_type by blast
next
  case 7 thus ?case using delete.delete_type by blast
qed auto

end

```

24 1-2 Brother Tree Implementation of Maps

```

theory Brother12_Map
imports
  Brother12_Set
  Map_by_Ordered
begin

fun lookup :: ('a  $\times$  'b) bro  $\Rightarrow$  'a::cmp  $\Rightarrow$  'b option where
  lookup N0 x = None |
  lookup (N1 t) x = lookup t x |
  lookup (N2 l (a,b) r) x =
    (case cmp x a of
      LT  $\Rightarrow$  lookup l x |
      EQ  $\Rightarrow$  Some b |
      GT  $\Rightarrow$  lookup r x)

locale update = insert
begin

fun upd :: 'a::cmp  $\Rightarrow$  'b  $\Rightarrow$  ('a $\times$ 'b) bro  $\Rightarrow$  ('a $\times$ 'b) bro where
  upd x y N0 = L2 (x,y) |
  upd x y (N1 t) = n1 (upd x y t) |
  upd x y (N2 l (a,b) r) =

```

```

(case cmp x a of
  LT  $\Rightarrow$  n2 (upd x y l) (a,b) r |
  EQ  $\Rightarrow$  N2 l (a,y) r |
  GT  $\Rightarrow$  n2 l (a,b) (upd x y r))

```

definition *update* :: 'a::cmp $\Rightarrow$ 'b $\Rightarrow$ ('a $\times$ 'b) bro $\Rightarrow$ ('a $\times$ 'b) bro **where**
update x y t = tree(upd x y t)

end

context *delete*
begin

fun *del* :: 'a::cmp $\Rightarrow$ ('a $\times$ 'b) bro $\Rightarrow$ ('a $\times$ 'b) bro **where**
del _ N0 = N0 |
del x (N1 t) = N1 (del x t) |
del x (N2 l (a,b) r) =
 (case cmp x a of
 LT $\Rightarrow$ n2 (del x l) (a,b) r |
 GT $\Rightarrow$ n2 l (a,b) (del x r) |
 EQ $\Rightarrow$ (case del_min r of
 None $\Rightarrow$ N1 l |
 Some (ab, r') $\Rightarrow$ n2 l ab r'))

definition *delete* :: 'a::cmp $\Rightarrow$ ('a $\times$ 'b) bro $\Rightarrow$ ('a $\times$ 'b) bro **where**
delete a t = tree (del a t)

end

24.1 Functional Correctness Proofs

24.1.1 Proofs for lookup

lemma *lookup_map_of*: $t \in T \ h \Longrightarrow$
 $sorted1(inorder \ t) \Longrightarrow lookup \ t \ x = map_of \ (inorder \ t) \ x$
by(induction h arbitrary: t) (auto simp: map_of_simps split: option.splits)

24.1.2 Proofs for update

context *update*
begin

lemma *inorder_upd*: $t \in T \ h \Longrightarrow$
 $sorted1(inorder \ t) \Longrightarrow inorder(upd \ x \ y \ t) = upd_list \ x \ y \ (inorder \ t)$
by(induction h arbitrary: t) (auto simp: upd_list_simps inorder_n1 inorder_n2)

```

lemma inorder_update:  $t \in T\ h \implies$ 
   $sorted1(inorder\ t) \implies inorder(update\ x\ y\ t) = upd\_list\ x\ y\ (inorder\ t)$ 
by(simp add: update_def inorder_upd inorder_tree)

end

```

24.1.3 Proofs for deletion

```

context delete
begin

```

```

lemma inorder_del:
   $t \in T\ h \implies sorted1(inorder\ t) \implies inorder(del\ x\ t) = del\_list\ x\ (inorder\ t)$ 
by(induction h arbitrary: t) (auto simp: del_list_simps inorder_n2
  inorder_del_min[OF UnI1] inorder_del_min[OF UnI2] split: option.splits)

```

```

lemma inorder_delete:
   $t \in T\ h \implies sorted1(inorder\ t) \implies inorder(delete\ x\ t) = del\_list\ x\ (inorder\ t)$ 
by(simp add: delete_def inorder_del inorder_tree)

```

```

end

```

24.2 Invariant Proofs

24.2.1 Proofs for update

```

context update
begin

```

```

lemma upd_type:
   $(t \in B\ h \longrightarrow upd\ x\ y\ t \in Bp\ h) \wedge (t \in U\ h \longrightarrow upd\ x\ y\ t \in T\ h)$ 
apply(induction h arbitrary: t)
  apply (simp)
apply (fastforce simp: Bp_if_B n2_type dest: n1_type)
done

```

```

lemma update_type:
   $t \in B\ h \implies update\ x\ y\ t \in B\ h \cup B\ (Suc\ h)$ 
unfolding update_def by (metis upd_type tree_type)

```

```

end

```

24.2.2 Proofs for deletion

context *delete*

begin

lemma *del_type*:

$t \in B \ h \implies \text{del } x \ t \in T \ h$

$t \in U \ h \implies \text{del } x \ t \in Um \ h$

proof (*induction h arbitrary: x t*)

case (*Suc h*)

{ **case** 1

then obtain $l \ a \ b \ r$ **where** $[simp]: t = N2 \ l \ (a, b) \ r$ **and**

$lr: l \in T \ h \ r \in T \ h \ l \in B \ h \vee r \in B \ h$ **by** *auto*

{ **assume** $x < a$

have *?case*

proof *cases*

assume $l \in B \ h$

from $n2_type3[OF \ Suc.IH(1)[OF \ this] \ lr(2)]$

show *?thesis* **using** $\langle x < a \rangle$ **by** (*simp*)

next

assume $l \notin B \ h$

hence $l \in U \ h \ r \in B \ h$ **using** *lr* **by** *auto*

from $n2_type1[OF \ Suc.IH(2)[OF \ this(1)] \ this(2)]$

show *?thesis* **using** $\langle x < a \rangle$ **by** (*simp*)

qed

} **moreover**

{ **assume** $x > a$

have *?case*

proof *cases*

assume $r \in B \ h$

from $n2_type3[OF \ lr(1) \ Suc.IH(1)[OF \ this]]$

show *?thesis* **using** $\langle x > a \rangle$ **by** (*simp*)

next

assume $r \notin B \ h$

hence $l \in B \ h \ r \in U \ h$ **using** *lr* **by** *auto*

from $n2_type2[OF \ this(1) \ Suc.IH(2)[OF \ this(2)]]$

show *?thesis* **using** $\langle x > a \rangle$ **by** (*simp*)

qed

} **moreover**

{ **assume** $[simp]: x = a$

have *?case*

proof (*cases del_min r*)

case *None*

show *?thesis*

```

proof cases
  assume  $r \in B\ h$ 
  with  $\text{del\_minNoneN0}[OF\ this\ None]\ lr$  show  $?thesis$  by (simp)
next
  assume  $r \notin B\ h$ 
  hence  $r \in U\ h$  using  $lr$  by auto
  with  $\text{del\_minNoneN1}[OF\ this\ None]\ lr(3)$  show  $?thesis$  by (simp)
qed
next
  case [simp]: (Some  $br'$ )
  obtain  $b\ r'$  where [simp]:  $br' = (b, r')$  by fastforce
  show  $?thesis$ 
  proof cases
    assume  $r \in B\ h$ 
    from  $\text{del\_min\_type}(1)[OF\ this]\ n2\_type3[OF\ lr(1)]$ 
    show  $?thesis$  by simp
  next
    assume  $r \notin B\ h$ 
    hence  $l \in B\ h$  and  $r \in U\ h$  using  $lr$  by auto
    from  $\text{del\_min\_type}(2)[OF\ this(2)]\ n2\_type2[OF\ this(1)]$ 
    show  $?thesis$  by simp
  qed
qed
} ultimately show  $?case$  by auto
}
{ case 2 with  $\text{Suc.IH}(1)$  show  $?case$  by auto }
qed auto

lemma delete_type:
   $t \in B\ h \implies \text{delete}\ x\ t \in B\ h \cup B(h-1)$ 
unfolding delete_def
by (cases  $h$ ) (simp, metis  $\text{del\_type}(1)\ \text{tree\_type}\ \text{Suc\_eq\_plus1}\ \text{diff\_Suc\_1}$ )

end

```

24.3 Overall correctness

```

interpretation Map_by_Ordered
where empty = N0 and lookup = lookup and update = update.update
and delete = delete.delete and inorder = inorder and inv =  $\lambda t. \exists h. t \in B\ h$ 
proof (standard, goal_cases)
  case 2 thus  $?case$  by (auto intro!: lookup_map_of)
next

```

```

    case 3 thus ?case by(auto intro!: update.inorder_update)
next
    case 4 thus ?case by(auto intro!: delete.inorder_delete)
next
    case 6 thus ?case using update.update_type by (metis Un_iff)
next
    case 7 thus ?case using delete.delete_type by blast
qed auto

end

```

25 AA Tree Implementation of Sets

```

theory AA_Set
imports
  Isin2
  Cmp
begin

```

```

type_synonym 'a aa_tree = ('a,nat) tree

```

```

fun lvl :: 'a aa_tree  $\Rightarrow$  nat where
  lvl Leaf = 0 |
  lvl (Node lv _ _ _) = lv

```

```

fun skew :: 'a aa_tree  $\Rightarrow$  'a aa_tree where
  skew (Node lva (Node lvb t1 b t2) a t3) =
    (if lva = lvb then Node lva t1 b (Node lva t2 a t3) else Node lva (Node lvb
  t1 b t2) a t3) |
  skew t = t

```

```

fun split :: 'a aa_tree  $\Rightarrow$  'a aa_tree where
  split (Node lva t1 a (Node lvb t2 b (Node lvc t3 c t4))) =
    (if lva = lvb  $\wedge$  lvb = lvc (* lva = lvc suffices *)
    then Node (lva+1) (Node lva t1 a t2) b (Node lva t3 c t4)
    else Node lva t1 a (Node lvb t2 b (Node lvc t3 c t4))) |
  split t = t

```

```

hide_const (open) insert

```

```

fun insert :: 'a::cmp  $\Rightarrow$  'a aa_tree  $\Rightarrow$  'a aa_tree where
  insert x Leaf = Node 1 Leaf x Leaf |
  insert x (Node lv t1 a t2) =

```

```

(case cmp x a of
  LT ⇒ split (skew (Node lv (insert x t1) a t2)) |
  GT ⇒ split (skew (Node lv t1 a (insert x t2))) |
  EQ ⇒ Node lv t1 x t2)

```

```

fun del_max :: 'a aa_tree ⇒ 'a aa_tree * 'a where
del_max (Node lv l a Leaf) = (l,a) |
del_max (Node lv l a r) = (let (r',b) = del_max r in (Node lv l a r', b))

```

```

fun sngl :: 'a aa_tree ⇒ bool where
sngl Leaf = False |
sngl (Node _ _ Leaf) = True |
sngl (Node lva _ (Node lvb _ _)) = (lva > lvb)

```

```

definition adjust :: 'a aa_tree ⇒ 'a aa_tree where
adjust t =
(case t of
  Node lv l x r ⇒
    (if lvl l >= lv-1 ∧ lvl r >= lv-1 then t else
     if lvl r < lv-1 ∧ sngl l then skew (Node (lv-1) l x r) else
     if lvl r < lv-1
     then case l of
        Node lva t1 a (Node lvb t2 b t3)
          ⇒ Node (lvb+1) (Node lva t1 a t2) b (Node (lv-1) t3 x r) |
        _ ⇒ t (* unreachable *))
    else
    if lvl r < lv then split (Node (lv-1) l x r)
    else
    case r of
      Leaf ⇒ Leaf (* unreachable *) |
      Node lvb t1 b t4 ⇒
        (case t1 of
          Node lva t2 a t3
            ⇒ Node (lva+1) (Node (lv-1) l x t2) a
              (split (Node (if sngl t1 then lva-1 else lva) t3 b t4))
          | _ ⇒ t (* unreachable *)))

```

```

fun delete :: 'a::cmp ⇒ 'a aa_tree ⇒ 'a aa_tree where
delete _ Leaf = Leaf |
delete x (Node lv l a r) =
  (case cmp x a of
    LT ⇒ adjust (Node lv (delete x l) a r) |
    GT ⇒ adjust (Node lv l a (delete x r)) |

```

$EQ \Rightarrow (\text{if } l = \text{Leaf then } r$
 $\text{else let } (l', b) = \text{del_max } l \text{ in adjust } (\text{Node } lv \ l' \ b \ r)))$

25.1 Functional Correctness

25.1.1 Proofs for insert

lemma *inorder_split*: $\text{inorder}(\text{split } t) = \text{inorder } t$
by(*cases t rule: split.cases*) (*auto*)

lemma *inorder_skew*: $\text{inorder}(\text{skew } t) = \text{inorder } t$
by(*cases t rule: skew.cases*) (*auto*)

lemma *inorder_insert*:
 $\text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{insert } x \ t) = \text{ins_list } x \ (\text{inorder } t)$
by(*induction t*) (*auto simp: ins_list_simps inorder_split inorder_skew*)

25.1.2 Proofs for delete

lemma *del_maxD*:
 $\llbracket \text{del_max } t = (t', x); t \neq \text{Leaf} \rrbracket \implies \text{inorder } t' @ [x] = \text{inorder } t$
by(*induction t arbitrary: t' rule: del_max.induct*)
(*auto simp: sorted_lems split: prod.splits*)

lemma *inorder_adjust*: $t \neq \text{Leaf} \implies \text{inorder}(\text{adjust } t) = \text{inorder } t$
by(*induction t*)
(*auto simp: adjust_def inorder_skew inorder_split split: tree.splits*)

lemma *inorder_delete*:
 $\text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{delete } x \ t) = \text{del_list } x \ (\text{inorder } t)$
by(*induction t*)
(*auto simp: del_list_simps inorder_adjust del_maxD split: prod.splits*)

25.2 Overall correctness

interpretation *Set_by_Ordered*
where *empty* = *Leaf* **and** *isin* = *isin* **and** *insert* = *insert* **and** *delete* = *delete*
and *inorder* = *inorder* **and** *inv* = $\lambda_.$ *True*
proof (*standard, goal_cases*)
 case 1 **show** ?*case* **by** *simp*
next
 case 2 **thus** ?*case* **by**(*simp add: isin_set*)
next
 case 3 **thus** ?*case* **by**(*simp add: inorder_insert*)

```

next
  case 4 thus ?case by(simp add: inorder_delete)
qed auto

end

```

26 AA Implementation of Maps

```

theory AA_Map
imports
  AA_Set
  Lookup2
begin

fun update :: 'a::cmp  $\Rightarrow$  'b  $\Rightarrow$  ('a*'b) aa_tree  $\Rightarrow$  ('a*'b) aa_tree where
  update x y Leaf = Node 1 Leaf (x,y) Leaf |
  update x y (Node lv t1 (a,b) t2) =
    (case cmp x a of
      LT  $\Rightarrow$  split (skew (Node lv (update x y t1) (a,b) t2)) |
      GT  $\Rightarrow$  split (skew (Node lv t1 (a,b) (update x y t2))) |
      EQ  $\Rightarrow$  Node lv t1 (x,y) t2)

fun delete :: 'a::cmp  $\Rightarrow$  ('a*'b) aa_tree  $\Rightarrow$  ('a*'b) aa_tree where
  delete _ Leaf = Leaf |
  delete x (Node lv l (a,b) r) =
    (case cmp x a of
      LT  $\Rightarrow$  adjust (Node lv (delete x l) (a,b) r) |
      GT  $\Rightarrow$  adjust (Node lv l (a,b) (delete x r)) |
      EQ  $\Rightarrow$  (if l = Leaf then r
        else let (l',ab') = del_max l in adjust (Node lv l' ab' r)))

```

26.1 Functional Correctness Proofs

```

theorem inorder_update:
  sorted1(inorder t)  $\implies$  inorder(update x y t) = upd_list x y (inorder t)
by (induct t) (auto simp: upd_list_simps inorder_split inorder_skew)

```

```

theorem inorder_delete:
  sorted1(inorder t)  $\implies$  inorder (delete x t) = del_list x (inorder t)
by(induction t)
  (auto simp: del_list_simps inorder_adjust del_maxD split: prod.splits)

```

```

interpretation Map_by_Ordered

```

```

where empty = Leaf and lookup = lookup and update = update and delete
= delete
and inorder = inorder and inv =  $\lambda\_.$  True
proof (standard, goal_cases)
  case 1 show ?case by simp
next
  case 2 thus ?case by (simp add: lookup_map_of)
next
  case 3 thus ?case by (simp add: inorder_update)
next
  case 4 thus ?case by (simp add: inorder_delete)
qed auto

end

```

27 Splay Tree Implementation of Sets

theory *Splay_Set*

imports

~~/*src/HOL/Library/Tree*

Set_by_Ordered

Cmp

begin

function *splay* :: '*a*::*linorder* $\Rightarrow$ '*a* *tree* $\Rightarrow$ '*a* *tree* **where**

```

splay a Leaf = Leaf |
splay a (Node t1 a t2) = Node t1 a t2 |
a < b  $\Longrightarrow$  splay a (Node (Node t1 a t2) b t3) = Node t1 a (Node t2 b t3) |
x < a  $\Longrightarrow$  splay x (Node Leaf a t) = Node Leaf a t |
x < a  $\Longrightarrow$  x < b  $\Longrightarrow$  splay x (Node (Node Leaf a t1) b t2) = Node Leaf a
(Node t1 b t2) |
x < a  $\Longrightarrow$  x < b  $\Longrightarrow$  t1  $\neq$  Leaf  $\Longrightarrow$ 
  splay x (Node (Node t1 a t2) b t3) =
  (case splay x t1 of Node t11 y t12  $\Rightarrow$  Node t11 y (Node t12 a (Node t2 b
t3))) |
a < x  $\Longrightarrow$  x < b  $\Longrightarrow$  splay x (Node (Node t1 a Leaf) b t2) = Node t1 a (Node
Leaf b t2) |
a < x  $\Longrightarrow$  x < b  $\Longrightarrow$  t2  $\neq$  Leaf  $\Longrightarrow$ 
  splay x (Node (Node t1 a t2) b t3) =
  (case splay x t2 of Node t21 y t22  $\Rightarrow$  Node (Node t1 a t21) y (Node t22 b
t3)) |
a < b  $\Longrightarrow$  splay b (Node t1 a (Node t2 b t3)) = Node (Node t1 a t2) b t3 |
a < x  $\Longrightarrow$  splay x (Node t a Leaf) = Node t a Leaf |

```

$a < x \implies x < b \implies t2 \neq \text{Leaf} \implies$
 $\text{splay } x \text{ (Node } t1 \text{ a (Node } t2 \text{ b } t3)) =$
 $(\text{case splay } x \text{ t2 of Node } t21 \text{ y } t22 \Rightarrow \text{Node (Node } t1 \text{ a } t21) \text{ y (Node } t22 \text{ b } t3)) \mid$
 $a < x \implies x < b \implies \text{splay } x \text{ (Node } t1 \text{ a (Node Leaf b } t2)) = \text{Node (Node } t1 \text{ a Leaf) b } t2 \mid$
 $a < x \implies b < x \implies \text{splay } x \text{ (Node } t1 \text{ a (Node } t2 \text{ b Leaf)) = Node (Node } t1 \text{ a } t2) \text{ b Leaf} \mid$
 $a < x \implies b < x \implies t3 \neq \text{Leaf} \implies$
 $\text{splay } x \text{ (Node } t1 \text{ a (Node } t2 \text{ b } t3)) =$
 $(\text{case splay } x \text{ t3 of Node } t31 \text{ y } t32 \Rightarrow \text{Node (Node (Node } t1 \text{ a } t2) \text{ b } t31) \text{ y } t32)$

apply(*atomize_elim*)

apply(*auto*)

apply (*subst (asm) neq_Leaf_iff*)

apply(*auto*)

apply (*metis tree.exhaust less_linear*)+

done

termination *splay*

by *lexicographic_order*

lemma *case_tree_cong*:

$\llbracket x = x'; y = y'; z = z' \rrbracket \implies \text{case_tree } x \text{ y } z = \text{case_tree } x' \text{ y}' \text{ z}'$

by *auto*

lemma *splay_code*: $\text{splay } (x::\dots::\text{cmp}) \text{ t} = (\text{case } t \text{ of Leaf} \Rightarrow \text{Leaf} \mid$

$\text{Node } al \text{ a } ar \Rightarrow (\text{case cmp } x \text{ a of}$

$\text{EQ} \Rightarrow t \mid$

$\text{LT} \Rightarrow (\text{case } al \text{ of}$

$\text{Leaf} \Rightarrow t \mid$

$\text{Node } bl \text{ b } br \Rightarrow (\text{case cmp } x \text{ b of}$

$\text{EQ} \Rightarrow \text{Node } bl \text{ b (Node } br \text{ a } ar) \mid$

$\text{LT} \Rightarrow \text{if } bl = \text{Leaf then Node } bl \text{ b (Node } br \text{ a } ar)$

$\text{else case splay } x \text{ bl of}$

$\text{Node } bll \text{ y } blr \Rightarrow \text{Node } bll \text{ y (Node } blr \text{ b (Node } br \text{ a } ar)) \mid$

$\text{GT} \Rightarrow \text{if } br = \text{Leaf then Node } bl \text{ b (Node } br \text{ a } ar)$

$\text{else case splay } x \text{ br of}$

$\text{Node } brl \text{ y } brr \Rightarrow \text{Node (Node } bl \text{ b } brl) \text{ y (Node } brr \text{ a } ar))) \mid$

$\text{GT} \Rightarrow (\text{case } ar \text{ of}$

$\text{Leaf} \Rightarrow t \mid$

$\text{Node } bl \text{ b } br \Rightarrow (\text{case cmp } x \text{ b of}$

```

EQ ⇒ Node (Node al a bl) b br |
LT ⇒ if bl = Leaf then Node (Node al a bl) b br
      else case splay x bl of
            Node bll y blr ⇒ Node (Node al a bll) y (Node blr b br) |
GT ⇒ if br = Leaf then Node (Node al a bl) b br
      else case splay x br of
            Node bll y blr ⇒ Node (Node (Node al a bl) b bll) y blr))))
by(auto cong: case_tree_cong split: tree.split)

```

definition *is_root* :: 'a ⇒ 'a tree ⇒ bool **where**
is_root x t = (case t of Leaf ⇒ False | Node l a r ⇒ x = a)

definition *isin* t x = *is_root* x (splay x t)

hide_const (open) *insert*

fun *insert* :: 'a::cmp ⇒ 'a tree ⇒ 'a tree **where**
insert x t =
 (if t = Leaf then Node Leaf x Leaf
 else case splay x t of
 Node l a r ⇒
 case cmp x a of
 EQ ⇒ Node l a r |
 LT ⇒ Node l x (Node Leaf a r) |
 GT ⇒ Node (Node l a Leaf) x r)

fun *splay_max* :: 'a tree ⇒ 'a tree **where**
splay_max Leaf = Leaf |
splay_max (Node l b Leaf) = Node l b Leaf |
splay_max (Node l b (Node rl c rr)) =
 (if rr = Leaf then Node (Node l b rl) c Leaf
 else case splay_max rr of
 Node rrl m rrr ⇒ Node (Node (Node l b rl) c rrl) m rrr)

definition *delete* :: 'a::linorder ⇒ 'a tree ⇒ 'a tree **where**
delete x t =
 (if t = Leaf then Leaf
 else case splay x t of Node l a r ⇒
 if x = a
 then if l = Leaf then r else case splay_max l of Node l' m r' ⇒ Node l'
 m r
 else Node l a r)

27.1 Functional Correctness Proofs

lemma *splay_Leaf_iff*: $(\text{splay } a \ t = \text{Leaf}) = (t = \text{Leaf})$
by(*induction a t rule: splay.induct*) (*auto split: tree.splits*)

lemma *splay_max_Leaf_iff*: $(\text{splay_max } t = \text{Leaf}) = (t = \text{Leaf})$
by(*induction t rule: splay_max.induct*)(*auto split: tree.splits*)

27.1.1 Proofs for isin

lemma
 $\text{splay } x \ t = \text{Node } l \ a \ r \implies \text{sorted}(\text{inorder } t) \implies$
 $x \in \text{elems } (\text{inorder } t) \longleftrightarrow x=a$
by(*induction x t arbitrary: l a r rule: splay.induct*)
(*auto simp: elems_simps1 splay_Leaf_iff ball_Un split: tree.splits*)

lemma *splay_elemsD*:
 $\text{splay } x \ t = \text{Node } l \ a \ r \implies \text{sorted}(\text{inorder } t) \implies$
 $x \in \text{elems } (\text{inorder } t) \longleftrightarrow x=a$
by(*induction x t arbitrary: l a r rule: splay.induct*)
(*auto simp: elems_simps2 splay_Leaf_iff split: tree.splits*)

lemma *isin_set*: $\text{sorted}(\text{inorder } t) \implies \text{isin } t \ x = (x \in \text{elems } (\text{inorder } t))$
by (*auto simp: isin_def is_root_def splay_elemsD splay_Leaf_iff split: tree.splits*)

27.1.2 Proofs for insert

lemma *inorder_splay*: $\text{inorder}(\text{splay } x \ t) = \text{inorder } t$
by(*induction x t rule: splay.induct*)
(*auto simp: neq_Leaf_iff split: tree.split*)

lemma *sorted_splay*:
 $\text{sorted}(\text{inorder } t) \implies \text{splay } x \ t = \text{Node } l \ a \ r \implies$
 $\text{sorted}(\text{inorder } l \ @ \ x \ \# \ \text{inorder } r)$
unfolding *inorder_splay*[*of x t, symmetric*]
by(*induction x t arbitrary: l a r rule: splay.induct*)
(*auto simp: sorted_simps sorted_Cons_le sorted_snoc_le splay_Leaf_iff split: tree.splits*)

lemma *inorder_insert*:
 $\text{sorted}(\text{inorder } t) \implies \text{inorder}(\text{insert } x \ t) = \text{ins_list } x \ (\text{inorder } t)$
using *inorder_splay*[*of x t, symmetric*] *sorted_splay*[*of t x*]
by(*auto simp: ins_list_simps ins_list_Cons ins_list_snoc neq_Leaf_iff split: tree.split*)

27.1.3 Proofs for delete

lemma *inorder_splay_maxD*:

*splay_max t = Node l a r $\implies$ sorted(inorder t) $\implies$
inorder l @ [a] = inorder t $\wedge$ r = Leaf*

by (*induction t arbitrary: l a r rule: splay_max.induct*)
(*auto simp: sorted_lems splay_max_Leaf_iff split: tree.splits if_splits*)

lemma *inorder_delete*:

sorted(inorder t) $\implies$ inorder(delete x t) = del_list x (inorder t)

using *inorder_splay[of x t, symmetric] sorted_splay[of t x]*

by (*auto simp: del_list_simps del_list_sorted_app delete_def
del_list_notin_Cons inorder_splay_maxD splay_Leaf_iff splay_max_Leaf_iff
split: tree.splits*)

27.1.4 Overall Correctness

interpretation *Set.by_Ordered*

where *empty = Leaf and isin = isin and insert = insert*

and *delete = delete and inorder = inorder and inv = $\lambda_.$ True*

proof (*standard, goal_cases*)

case 2 **thus** ?case **by** (*simp add: isin_set*)

next

case 3 **thus** ?case **by** (*simp add: inorder_insert del: insert_simps*)

next

case 4 **thus** ?case **by** (*simp add: inorder_delete*)

qed *auto*

end

28 Splay Tree Implementation of Maps

theory *Splay_Map*

imports

Splay_Set

Map_by_Ordered

begin

function *splay* :: '*a::linorder* $\Rightarrow$ ('*a*'b*) tree $\Rightarrow$ ('*a*'b*) tree **where**

splay x Leaf = Leaf |

x = fst a $\implies$ splay x (Node t1 a t2) = Node t1 a t2 |

*x = fst a $\implies$ x < fst b $\implies$ splay x (Node (Node t1 a t2) b t3) = Node t1
a (Node t2 b t3)* |

x < fst a $\implies$ splay x (Node Leaf a t) = Node Leaf a t |

$x < fst\ a \implies x < fst\ b \implies splay\ x\ (Node\ (Node\ Leaf\ a\ t1)\ b\ t2) = Node\ Leaf\ a\ (Node\ t1\ b\ t2) \mid$
 $x < fst\ a \implies x < fst\ b \implies t1 \neq Leaf \implies$
 $splay\ x\ (Node\ (Node\ t1\ a\ t2)\ b\ t3) =$
 $(case\ splay\ x\ t1\ of\ Node\ t11\ y\ t12 \Rightarrow Node\ t11\ y\ (Node\ t12\ a\ (Node\ t2\ b\ t3))) \mid$
 $fst\ a < x \implies x < fst\ b \implies splay\ x\ (Node\ (Node\ t1\ a\ Leaf)\ b\ t2) = Node\ t1\ a\ (Node\ Leaf\ b\ t2) \mid$
 $fst\ a < x \implies x < fst\ b \implies t2 \neq Leaf \implies$
 $splay\ x\ (Node\ (Node\ t1\ a\ t2)\ b\ t3) =$
 $(case\ splay\ x\ t2\ of\ Node\ t21\ y\ t22 \Rightarrow Node\ (Node\ t1\ a\ t21)\ y\ (Node\ t22\ b\ t3)) \mid$
 $fst\ a < x \implies x = fst\ b \implies splay\ x\ (Node\ t1\ a\ (Node\ t2\ b\ t3)) = Node\ (Node\ t1\ a\ t2)\ b\ t3 \mid$
 $fst\ a < x \implies splay\ x\ (Node\ t\ a\ Leaf) = Node\ t\ a\ Leaf \mid$
 $fst\ a < x \implies x < fst\ b \implies t2 \neq Leaf \implies$
 $splay\ x\ (Node\ t1\ a\ (Node\ t2\ b\ t3)) =$
 $(case\ splay\ x\ t2\ of\ Node\ t21\ y\ t22 \Rightarrow Node\ (Node\ t1\ a\ t21)\ y\ (Node\ t22\ b\ t3)) \mid$
 $fst\ a < x \implies x < fst\ b \implies splay\ x\ (Node\ t1\ a\ (Node\ Leaf\ b\ t2)) = Node\ (Node\ t1\ a\ Leaf)\ b\ t2 \mid$
 $fst\ a < x \implies fst\ b < x \implies splay\ x\ (Node\ t1\ a\ (Node\ t2\ b\ Leaf)) = Node\ (Node\ t1\ a\ t2)\ b\ Leaf \mid$
 $fst\ a < x \implies fst\ b < x \implies t3 \neq Leaf \implies$
 $splay\ x\ (Node\ t1\ a\ (Node\ t2\ b\ t3)) =$
 $(case\ splay\ x\ t3\ of\ Node\ t31\ y\ t32 \Rightarrow Node\ (Node\ (Node\ t1\ a\ t2)\ b\ t31)\ y\ t32)$
apply(*atomize_elim*)
apply(*auto*)

apply (*subst (asm) neq_Leaf_iff*)
apply(*auto*)
apply (*metis tree.exhaust surj_pair less_linear*)+
done

termination *splay*
by *lexicographic_order*

lemma *splay_code*: $splay\ (x:::cmp)\ t = (case\ t\ of\ Leaf \Rightarrow Leaf \mid$
 $Node\ al\ a\ ar \Rightarrow (case\ cmp\ x\ (fst\ a)\ of$
 $EQ \Rightarrow t \mid$
 $LT \Rightarrow (case\ al\ of$
 $Leaf \Rightarrow t \mid$
 $Node\ bl\ b\ br \Rightarrow (case\ cmp\ x\ (fst\ b)\ of$

```

EQ ⇒ Node bl b (Node br a ar) |
LT ⇒ if bl = Leaf then Node bl b (Node br a ar)
      else case splay x bl of
        Node bll y blr ⇒ Node bll y (Node blr b (Node br a ar)) |
GT ⇒ if br = Leaf then Node bl b (Node br a ar)
      else case splay x br of
        Node brl y brr ⇒ Node (Node bl b brl) y (Node brr a ar))) |
GT ⇒ (case ar of
  Leaf ⇒ t |
  Node bl b br ⇒ (case cmp x (fst b) of
    EQ ⇒ Node (Node al a bl) b br |
    LT ⇒ if bl = Leaf then Node (Node al a bl) b br
          else case splay x bl of
            Node bll y blr ⇒ Node (Node al a bll) y (Node blr b br) |
    GT ⇒ if br = Leaf then Node (Node al a bl) b br
          else case splay x br of
            Node bll y blr ⇒ Node (Node (Node al a bl) b bll) y blr))))
by(auto cong: case_tree_cong split: tree.split)

```

definition *lookup* :: ('a*'b)tree ⇒ 'a::linorder ⇒ 'b option **where** *lookup* t
x =
 (case splay x t of Leaf ⇒ None | Node _ (a,b) _ ⇒ if x=a then Some b
 else None)

hide_const (open) *insert*

fun *update* :: 'a::linorder ⇒ 'b ⇒ ('a*'b) tree ⇒ ('a*'b) tree **where**
update x y t = (if t = Leaf then Node Leaf (x,y) Leaf
 else case splay x t of
 Node l a r ⇒ if x = fst a then Node l (x,y) r
 else if x < fst a then Node l (x,y) (Node Leaf a r) else Node (Node l a
 Leaf) (x,y) r)

definition *delete* :: 'a::linorder ⇒ ('a*'b) tree ⇒ ('a*'b) tree **where**
delete x t = (if t = Leaf then Leaf
 else case splay x t of Node l a r ⇒
 if x = fst a
 then if l = Leaf then r else case splay_max l of Node l' m r' ⇒ Node l'
 m r
 else Node l a r)

28.1 Functional Correctness Proofs

lemma *splay_Leaf_iff*: (splay x t = Leaf) = (t = Leaf)

by(*induction x t rule: splay.induct*) (*auto split: tree.splits*)

28.1.1 Proofs for lookup

lemma *splay_map_of_inorder*:

*splay x t = Node l a r $\implies$ sorted1(inorder t) $\implies$
 map_of (inorder t) x = (if x = fst a then Some(snd a) else None)*

by(*induction x t arbitrary: l a r rule: splay.induct*)
 (*auto simp: map_of_simps splay_Leaf_iff split: tree.splits*)

lemma *lookup_eq*:

sorted1(inorder t) $\implies$ lookup t x = map_of (inorder t) x

by(*auto simp: lookup_def splay_Leaf_iff splay_map_of_inorder split: tree.split*)

28.1.2 Proofs for update

lemma *inorder_splay*: *inorder(splay x t) = inorder t*

by(*induction x t rule: splay.induct*)
 (*auto simp: neq_Leaf_iff split: tree.split*)

lemma *sorted_splay*:

*sorted1(inorder t) $\implies$ splay x t = Node l a r $\implies$
 sorted(map fst (inorder l) @ x # map fst (inorder r))*

unfolding *inorder_splay*[*of x t, symmetric*]

by(*induction x t arbitrary: l a r rule: splay.induct*)
 (*auto simp: sorted_lems sorted_Cons_le sorted_snoc_le splay_Leaf_iff split: tree.splits*)

lemma *inorder_update*:

sorted1(inorder t) $\implies$ inorder(update x y t) = upd_list x y (inorder t)

using *inorder_splay*[*of x t, symmetric*] *sorted_splay*[*of t x*]

by(*auto simp: upd_list_simps upd_list_Cons upd_list_snoc neq_Leaf_iff split: tree.split*)

28.1.3 Proofs for delete

lemma *inorder_splay_maxD*:

*splay_max t = Node l a r $\implies$ sorted1(inorder t) $\implies$
 inorder l @ [a] = inorder t $\wedge$ r = Leaf*

by(*induction t arbitrary: l a r rule: splay_max.induct*)
 (*auto simp: sorted_lems splay_max_Leaf_iff split: tree.splits if_splits*)

lemma *inorder_delete*:

sorted1(inorder t) $\implies$ inorder(delete x t) = del_list x (inorder t)

using *inorder_splay*[*of x t, symmetric*] *sorted_splay*[*of t x*]

```

by (auto simp: del_list_simps del_list_sorted_app delete_def
  del_list_notin_Cons inorder_splay_maxD splay_Leaf_iff splay_max_Leaf_iff
  split: tree.splits)

```

28.1.4 Overall Correctness

```

interpretation Map_by_Ordered
where empty = Leaf and lookup = lookup and update = update
and delete = delete and inorder = inorder and inv =  $\lambda\_.$  True
proof (standard, goal_cases)
  case 2 thus ?case by (simp add: lookup_eq)
next
  case 3 thus ?case by (simp add: inorder_update del: update_simps)
next
  case 4 thus ?case by (simp add: inorder_delete)
qed auto

end

```

29 Bibliographic Notes

Red-black trees The insert function follows Okasaki [5], the delete function Kahrs [3, 4].

2-3 trees Equational definitions were given by Hoffmann and O’Donnell [2] (only insertion) and Reade [8]. Our formalisation is based on the teaching material by Turbak [11].

1-2 brother trees They were invented by Ottmann and Six [6, 7]. The functional version is due to Hinze [1].

Splay trees They were invented by Sleator and Tarjan [10]. Our formalisation follows Schoenmakers [9].

References

- [1] R. Hinze. Purely functional 1-2 brother trees. *J. Functional Programming*, 19(6):633–644, 2009.
- [2] C. M. Hoffmann and M. J. O’Donnell. Programming with equations. *ACM Trans. Program. Lang. Syst.*, 4(1):83–112, 1982.
- [3] S. Kahrs. Red black trees. <http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html>.

- [4] S. Kahrs. Red-black trees with types. *J. Functional Programming*, 11(4):425–432, 2001.
- [5] C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.
- [6] T. Ottmann and H.-W. Six. Eine neue Klasse von ausgeglichenen Binärbäumen. *Angewandte Informatik*, 18(9):395–400, 1976.
- [7] T. Ottmann and D. Wood. 1-2 brother trees or AVL trees revisited. *Comput. J.*, 23(3):248–255, 1980.
- [8] C. Reade. Balanced trees with removals: An exercise in rewriting and proof. *Sci. Comput. Program.*, 18(2):181–204, 1992.
- [9] B. Schoenmakers. A systematic analysis of splaying. *Information Processing Letters*, 45:41–50, 1993.
- [10] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. ACM*, 32(3):652–686, 1985.
- [11] F. Turbak. CS230 Handouts — Spring 2007, 2007. <http://cs.wellesley.edu/~cs230/spring07/handouts.html>.