Thys/BinHeap.thy

Binomial Heaps — Correctness and Complexity

Numerical Method


Only use trees $t_i$ of size $2^i$
Only use trees $t_i$ of size $2^i$
E.g., to store 0b11001 elements: $[t_0, 0, 0, t_3, t_4]$

Meld: Addition with carry.

---

Only use trees $t_i$ of size $2^i$
E.g., to store 0b11001 elements: $[t_0, 0, 0, t_3, t_4]$

Meld: Addition with carry.

---

**Organization of Trees**

```haskell```
datatype 'a tree = Node (rank: nat) (root: 'a)
  (children: 'a tree list)
```

Only use trees $t_i$ of size $2^i$
E.g., to store 0b11001 elements: $[t_0, 0, 0, t_3, t_4]$

Meld: Addition with carry.

Linking two trees of size $2^i$: Yields size $2^{i+1}$
Organization of Trees

**datatype** `a tree = Node (rank: nat) (root: 'a)` (children: 'a tree list)

Node with rank \(i\) has successors \([t_{i-1}, \ldots, t_0]\) with ranks \([i-1, \ldots, 0]\)

\[
btree\_invar\ (\text{Node} \ r \ uu \ c) = \ \\
(Ball \ (\text{set} \ c) \ btree\_invar \land \text{map} \text{ rank} \ c = \text{rev} \ [0..<r])
\]

Linking two Trees

Given two trees of rank \(i\), join them to tree of rank \(i+1\).

**Organization of Trees**

**datatype** `a tree = Node (rank: nat) (root: 'a)` (children: 'a tree list)

Node with rank \(i\) has successors \([t_{i-1}, \ldots, t_0]\) with ranks \([i-1, \ldots, 0]\)

\[
btree\_invar\ (\text{Node} \ r \ uu \ c) = \ \\
(Ball \ (\text{set} \ c) \ btree\_invar \land \text{map} \text{ rank} \ c = \text{rev} \ [0..<r])
\]

Tree has exactly \(2^{\text{rank } t}\) nodes

\[
btree\_invar \ t \implies |t| = 2^{\text{rank } t}
\]

Linking two Trees

Given two trees of rank \(i\), join them to tree of rank \(i+1\).

Idea: Insert one tree under root of other tree
Heap Datatype

Using sparse representation for binary numbers:

\[ [t_0, 0, 0, t_3, t_4] \text{ represented as } [ (0, t_0), (3, t_3), (4, t_4) ] \]

\textbf{type synonym} 'a heap = 'a tree list

Heap Datatype

Using sparse representation for binary numbers:

\[ [t_0, 0, 0, t_3, t_4] \text{ represented as } [ (0, t_0), (3, t_3), (4, t_4) ] \]

\textbf{type synonym} 'a heap = 'a tree list

Inserting a Tree

\textbf{ins_tree} t [] = [t]

\textbf{ins_tree} t1 (t2 # rest) =

(if \text{rank} t1 < \text{rank} t2 then t1 # t2 # rest

else \text{ins_tree} (\text{link} t1 t2) \text{ rest})

Ranks in ascending order
Inserting a Tree

\[
\text{ins_tree} \; t \; \[] = [t] \\
\text{ins_tree} \; t_1 \; (t_2 \; \# \; \text{rest}) = \\
\text{if} \; \text{rank} \; t_1 \; < \; \text{rank} \; t_2 \; \text{then} \; t_1 \; \# t_2 \; \# \; \text{rest} \\
\quad \text{else} \; \text{ins_tree} \; (\text{link} \; t_1 \; t_2) \; \text{rest}
\]

Intuition: Handle a carry

Merge

\[
\text{merge} \; t_{s1} \; \[] = t_{s1} \\
\text{merge} \; [] \; t_{s2} = t_{s2} \\
\text{merge} \; (t_1 \; \# \; t_{s1}) \; (t_2 \; \# \; t_{s2}) = \\
\quad \text{if} \; \text{rank} \; t_1 \; < \; \text{rank} \; t_2 \; \text{then} \; t_1 \; \# \; \text{merge} \; t_{s1} \; (t_2 \; \# \; t_{s2}) \\
\quad \text{else} \; \text{if} \; \text{rank} \; t_2 \; < \; \text{rank} \; t_1 \; \text{then} \; t_2 \; \# \; \text{merge} \; (t_1 \; \# \; t_{s1}) \; t_{s2} \\
\quad \text{else} \; \text{ins_tree} \; (\text{link} \; t_1 \; t_2) \; (\text{merge} \; t_{s1} \; t_{s2})
\]

Find/Delete Minimum Element

All trees are min-heaps
Smallest element may be any root node

\[
ts \neq [] \implies \text{find.min} \; ts = \text{Min} \; (\text{set} \; (\text{map} \; \text{root} \; ts))
\]

Intuition: Addition of binary numbers
Find/Delete Minimum Element

All trees are min-heaps
Smallest element may be any root node
\( ts \neq [] \implies \text{find_min } ts = \text{Min } (\text{set } (\text{map root } ts)) \)

Similar: \( \text{get_min} : 'a \text{ tree list } \Rightarrow 'a \text{ tree } \times 'a \text{ tree list} \)
Returns tree with minimal root, and other trees

Delete via merge
\( \text{delete_min } ts = \)
\( \text{(case get_min } ts \text{ of} \)
\( (\text{Node } xa x ts_1, ts_2) \Rightarrow \text{merge } (\text{rev } ts_1) ts_2 \)\)

Note the \( \text{rev}! \)

Complexity

Recall: \( |t| = 2^{\text{rank } t} \)
Recall: $|t| = 2^{\text{rank } t}$
Similarly for heap: $2^{\text{length } h} \leq |h|$

Recall: $|t| = 2^{\text{rank } t}$
Similarly for heap: $2^{\text{length } h} \leq |h|$
Complexity of operations: linear in length of heap

Recall: $|t| = 2^{\text{rank } t}$
Similarly for heap: $2^{\text{length } h} \leq |h|$
Complexity of operations: linear in length of heap

Find/Delete Minimum Element

All trees are min-heaps
Smallest element may be any root node
$ts \neq [] \implies \text{find.min } ts = \text{Min (set (map root ts))}$
Similar: $\text{get.min}:`a\text{ tree list} \Rightarrow `a\text{ tree } \times `a\text{ tree list}$
Returns tree with minimal root, and other trees
Find/Delete Minimum Element

All trees are min-heaps
Smallest element may be any root node
\( ts \neq \emptyset \implies \text{find.min} \ ts = \text{Min} \ (\text{set} \ (\text{map} \ \text{root} \ ts)) \)

Similar: \( \text{get.min} \) - a tree list \( \Rightarrow \) 'a tree \times 'a tree list
Returns tree with minimal root, and other trees
Delete via merge
\( \text{delete.min} \ ts = \)
\( \begin{cases} \text{get.min} \ ts \ \\ \text{(Node xa x ts}1, ts2) \Rightarrow \text{merge} \ (\text{rev} \ ts1) \ ts2 \end{cases} \)

Note the rev!

Complexity of Merge

\( \text{merge} \ (t1 \neq ts1) \ (t2 \neq ts2) = \)
\( \begin{cases} \text{(if rank t1 < rank t2 then t1 \neq merge ts1} (t2 \neq ts2) \\ \text{else if rank t2 < rank t1 then t2 \neq merge (t1 \neq ts1) ts2} \\ \text{else ins.tree} \ (\text{link} \ t1 \ t2) \ (\text{merge} \ ts1 \ ts2) \end{cases} \)

Complexity of \( \text{ins.tree} \) call depends on length of result of recursive call.
Naive: \( \text{length} \ (\text{merge} \ ts1 \ ts2) \leq \text{length} \ ts1 + \text{length} \ ts2 \)

Yields (roughly) \( m \ n = m \ (n-2) + n \)
**Complexity of Merge**

\[
merge (t_1 \# ts_1) (t_2 \# ts_2) = \\
\begin{align*}
& \text{if rank } t_1 < \text{rank } t_2 \text{ then } t_1 \# \text{merge } ts_1 (t_2 \# ts_2) \\
& \text{else if rank } t_2 < \text{rank } t_1 \text{ then } t_2 \# \text{merge } (t_1 \# ts_1) ts_2 \\
& \text{else } \text{ins_tree} (\text{link } t_1 t_2) (\text{merge } ts_1 ts_2)
\end{align*}
\]

Idea: Estimate cost and length of result:

\[
\text{t}\_\text{ins_tree} t ts + \text{length} (\text{ins_tree} t ts) = 2 + \text{length } ts \\
\text{length} (\text{merge } ts_1 ts_2) + \text{t}\_\text{merge } ts_1 ts_2 \\
\leq 2 * (\text{length } ts_1 + \text{length } ts_2) + 1
\]

Yields desired linear bound!

**Complexity of Merge**

\[
merge (t_1 \# ts_1) (t_2 \# ts_2) = \\
\begin{align*}
& \text{if rank } t_1 < \text{rank } t_2 \text{ then } t_1 \# \text{merge } ts_1 (t_2 \# ts_2) \\
& \text{else if rank } t_2 < \text{rank } t_1 \text{ then } t_2 \# \text{merge } (t_1 \# ts_1) ts_2 \\
& \text{else } \text{ins_tree} (\text{link } t_1 t_2) (\text{merge } ts_1 ts_2)
\end{align*}
\]

Idea: Estimate cost and length of result:

\[
\text{t}\_\text{ins_tree} t ts + \text{length} (\text{ins_tree} t ts) = 2 + \text{length } ts \\
\text{length} (\text{merge } ts_1 ts_2) + \text{t}\_\text{merge } ts_1 ts_2 \\
\leq 2 * (\text{length } ts_1 + \text{length } ts_2) + 1
\]

Yields desired linear bound!

**Complexity of Merge**

\[
\text{merge (t_1 \# ts_1) (t_2 \# ts_2) = } \\
\text{(if rank } t_1 < \text{rank } t_2 \text{ then } t_1 \# \text{merge } ts_1 (t_2 \# ts_2) } \\
\text{else if rank } t_2 < \text{rank } t_1 \text{ then } t_2 \# \text{merge } (t_1 \# ts_1) ts_2 \\
\text{else ins_tree (link } t_1 t_2) (\text{merge } ts_1 ts_2)
\]

Complexity of \text{ins_tree} call depends on length of result of recursive call.

\text{Naive}: \text{length} (\text{merge } ts_1 ts_2) \leq \text{length } ts_1 + \text{length } ts_2

Yields (roughly) \( m n = m (n-2) + n \) quadratic!
subsection: Binomial Tree and Heap Datatype:
datatype a tree = Node (rank: nat) (root: 'a) (children: 'a tree list')
type_synonym a heap = "a tree list"

subsection: Multiset of elements:
fun meet_tree :: "a:list order tree => 'a multiset" where
"meet_tree (Node _ c) = (c#F) ++ ( bene c meet_tree t)"
definition meet_heap :: "a:inorder heap => 'a multiset" where
"meet_heap c = ( bene c meet_heap c) meet_heap t"

lemma meet_tree.simps[ass]:
"meet_tree (Node r c) = (c#F) ++ meet_heap c"
unfolding meet_heap_def by auto
declare meet_tree.simps[del]
proofs are straightforward and automatic.

subsection: Binomial Tree and Heap Datatype:
datatype 'a tree = Node (rank: nat) (root: 'a) (children: 'a tree list)
type synonym 'a heap = 'a tree list

subsection: Multiset of elements:
fun mset_tree :: 'a:linorder tree => 'a multiset
  where
  "mset_tree (Node _ a c) = {#a} + "mset c <mset_tree t"

definition mset_heap :: 'a:linorder heap => 'a multiset
  where
  "mset_heap c = {#c} <mset c <mset_tree t"

lemma mset_tree_single[simp]:
  "mset_tree (Node r a c) = {#a} + mset_heap c"

unfolding mset_heap_def by auto

declare mset_heap_def [simp del]

lemma mset_tree_nonempty[simp]:
  "mset_tree t = {}" 

subsection: Binomial Tree and Heap Datatype:
datatype 'a tree = Node (rank: nat) (root: 'a) (children: 'a tree list)
type synonym 'a heap = 'a tree list

subsection: Multiset of elements:
fun mset_tree :: 'a:linorder tree => 'a multiset
  where
  "mset_tree (Node _ a c) = {#a} + "mset c <mset_tree t"

definition mset_heap :: 'a:linorder heap => 'a multiset
  where
  "mset_heap c = {#c} <mset c <mset_tree t"

lemma mset_tree_single[simp]:
  "mset_tree (Node r a c) = {#a} + mset_heap c"

unfolding mset_heap_def by auto

declare mset_heap_def [simp del]

lemma mset_tree_nonempty[simp]:
  "mset_tree t = {}" 

definition bheap_invar :: 'a:linorder heap => bool
  where
  "bheap_invar c = (\forall c. \forall t. btree_invar t \land (strictlyAscending (map rank c)))"

text: Ordering (heap) invariant:

fun otree_invar :: 'a:linorder tree => bool
  where
  "otree_invar (Node r a c) = (\forall c. otree_invar c \land a < c \land r < t)"

definition oheap_invar :: 'a:linorder heap => bool
  where
  "oheap_invar c = (\forall c. otree_invar c)"

definition invar :: 'a:linorder heap => bool
  where
  "invar t = bheap_invar t \land oheap_invar t"

text: THe children of a node are a valid heap:

lemma children_oheap_invar:
  "children oheap_invar (Node _ a c (rev t)) = c" 
  by (auto simp: oheap_invar_def)
lemma merge_same2[simp]: "merge \[ t_s \] t_s = t_s" by (cases t_s) auto

lemma merge_rank_bound:
  assumes "t'_s \subseteq set (merge t_s t_s)"
  assumes "t'_s \subseteq set (merge t_s t_s)"
  shows "rank t'_s \leq rank t_s"
  using assms
  apply (induction t_s t_s arbitrary: t'_s rule: merge.induct)
  apply (auto split: if_splits simp: ine_tree_rank_bound)
from bhheap_invar rs have
  
  
  
  
  
  
  unfolding bhheap_invar_def by auto
  
  
  
  
  
  
  also have \( \{ (\text{key}, \text{value}) \mid \text{key} : \text{nat} \} \) : \( \text{list} (\text{nat} \times \text{nat}) \)
  
  
  
  
  
  
  done
  
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
  
  
  
  
  
  
  done
fun t_fold f x s = \{ f x, f s \} \uplus f (t_fold f x s) \uplus f (t_fold f s)

text (Estimation for constant function is enough for our purposes):

lemma t_fold_const_bound:
- shows "t-fold \{ f \} \uplus f s = K ^ \text{length} + 1"
  - by (induction \text{ arbitrary} : s) auto

lemma t_fold_bounded_bound:
- assumes "x y : set \{ t \} : |t x s| \leq K"
  - shows "t-fold \{ f \} \uplus f s \leq K ^ \text{length} + 1"
  - using assms
  - apply (induction \text{ arbitrary} : s)
  - auto

interpretation birheap:
- Priority, Queue for "[]" is [] ins find_min_delete_min inorder_heap
  proof (unfold locales, goal_cases)
  - case 1 then show "case by simp" next
  - case 2 then show "case by simp" next
  - case 3 then show "case by simp" next
  - case 4 then show "case by simp" next

subsection (Instantiating the Priority Queue Locale)

subsection (Combined Find and Delete Operation)
- We define an operation that returns the minimum element and a heap with this element removed.
  definition pop_min : '(a,l) heap \times '(a,l) heap
  where "\{ pop_min ts = (case get_min ts of (Node x, ts), ts) = (ts, x) \times l, l \in lorder_heap \}"

lemma pop_min_min_right:
-