CS 423: Theory of Algorithms **Problem Set 3**
Spring, 1999
Tarjan

Problem Set No. 3 Solutions

1. The potential function is zero initially and always non-negative, so the total amortized time is an upper bound on the total actual time. Consider a link that makes a node x a child of a node y. The only node whose potential changes is y, whose potential can go up by at most $\lg n$ (since it must be between zero and $\lg n$). Since the actual time of the link is $O(1)$, the amortized time of the link is $O(\log n)$.

   Consider a find with compression along a path of k nodes. The compression can only cause the potential to drop, not increase, since nodes can only lose descendants, not gain them, as a result of the find. Let $x_1, x_2, \ldots, x_k$ be the successive nodes along the find path, with x_k the root. Call a node x_i *heavy* if $i \neq 1$, $i \neq k$, and $s(x_{i-1}) \geq s(x_i)/2$ before the find. As a result of the find $s(x_i)$ drops by at least (the original value of) $s(x_{i-1})$, and hence by at least a factor of 2 if x_i is heavy. This causes the potential of x_i to drop by at least 1. Since $1 \leq s(x_1) \leq s(x_2) \leq \ldots \leq s(x_k) \leq n$, and since if $i \neq 1, i \neq k$, and x_i is not heavy then $2s(x_{i-1}) \leq s(x_i)$, all but at most $2 + \lg n$ nodes on the find path must be heavy. If we measure the time of the find plus the compression as 1 per node, then the amortized time of the find is at most $k - (k - (\lg n + 2)) = \lg n + 2 = O(\log n)$.

2. (a) We assume that each node x initially has $p(x) = x$. We define a *compress* operation that performs path compression while preserving *sum* values, as follows:

   compress(x) :
   if $p(p(x)) \neq p(x)$ **then begin**
   compress $(p(x))$; *value*$(x) \leftarrow$ *value* $(x) +$ *value* $(p(x))$;
   $p(x) \leftarrow p(p(x))$; **end**

   This recursive procedure follows the path from x through ancestors to the root and then walks back down the path, replacing each value by the sum of values up to but not including the root value, and making each node point directly to the root. This preserves sums along paths. The time required for a compression increases by only a constant factor as compared to path compression as used in disjoint set union.

   We perform a *sum* operation using *compress*, as follows:

   sum (x) :
   if $p(x) = x$ **then return** *value* (x)
   else begin *compress* (x); **return** *value* $(x) +$ *value* $(p(x))$ **end**

If *sum* (x) performs a compression, x is a child of the root after the compression, which means that the value returned by *sum* (x) is correct.

We perform a *graft* operation as a combination of a *compress* and a *link* operation, as follows:

link $(x, y) : p(x) \leftarrow y$

graft (x, y) :
  if $p(y) = y$ then *link* (x, y)
  else begin *compress* (y); *value* $(x) \leftarrow$ *value* (x) + *value* (y); *link* $(x, p(y))$ **end**

This implementation preserves tree roots but not the structure of the trees, because it compresses paths. With this implementation, m *sum* and k intermixed *graft* operations translate into most $m + k$ *compress* and k *link* operations, for a total time of $\mathrm{O}((m + n)\log n) = \mathrm{O}(m \log n)$, since $m \geq n$ and $k \leq n$, applying the bound for path compression with naive linking.

(b) To allow the use of linking by rank, we must modify the implementation of *link* so that no matter how the link is performed, *sum* values are correct. The following version of *link* performs linking by rank while maintaining correct *sum* values, assuming all node ranks are initialized to zero:

link (x, y) :
  if $r(x) < r(y)$ **then** $p(x) \leftarrow y$
  else if $r(x) = r(y)$ **then begin** $p(x) \leftarrow y; r(y) \leftarrow r(y) + 1$ **end**
  else begin *value* $(x) \leftarrow$ *value* (x) + *value* (y); *value* $(y) \leftarrow$ *value* (y) − *value* (x);
    $p(y) \leftarrow x$ **end**

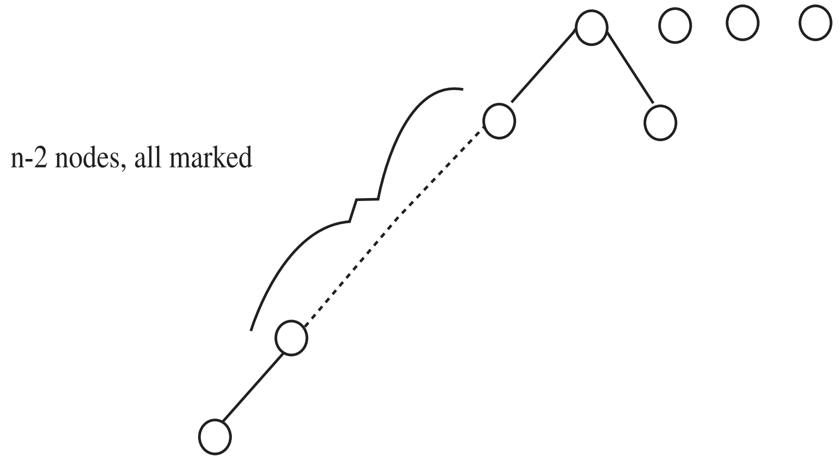
The value updates in the case $r(x) > r(y)$ give the following relationships: *new value* (x) = *old value* (x)+ *old value* (y); *new value* (y) = − *old value* (x). The new sum of values along a path to the new root x avoiding y is the old sum plus the old value of y, as desired; the new sum of values along a path to x through y is the old sum of values along the same path stopping at y, also as desired. In either case, path sums are correct.

Linking by rank does not preserve the roots of trees, so we must also modify *graft*, which in general must do two compressions. The following implementation of *graft* works:

graft (x, y) : **local** z;
  if $p(x) = x$ **then** $z \leftarrow x$
  else begin *compress* $(x); z \leftarrow p(x)$ **end**;
  if $p(y) = y$ **then** *link* (z, y)
  else begin *compress* (y); *value* $(z) \leftarrow$ *value* (z) + *value* (y);
    link $(z, p(y))$ **end**

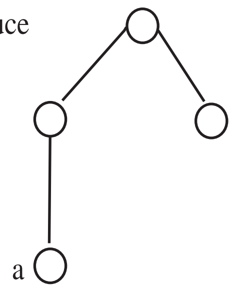
With this implementation, m sums and k grafts translate into at most $m + 2k$ compressions and k links, with a total time bound of $\mathrm{O}(m\alpha(n))$, applying the bound for path compression with linking by rank.

3. We show how to do $\mathrm{O}(n)$ heap operations to build the following tree:

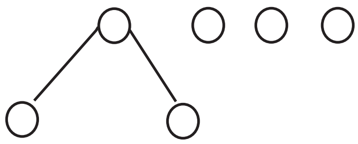


We can then do a decrease key on the item in the bottommost node. This will take $\Omega(n)$time, since cuts will cascade all the way to the root.

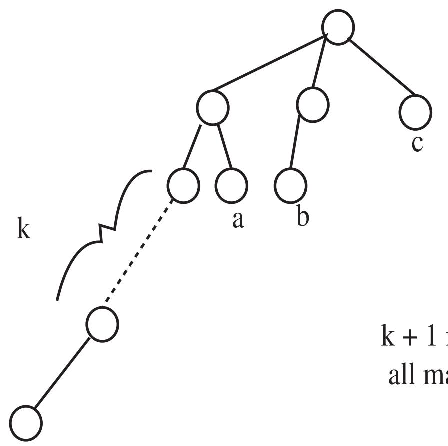
To get started, we do five inserts into an empty heap and a delete min to produce



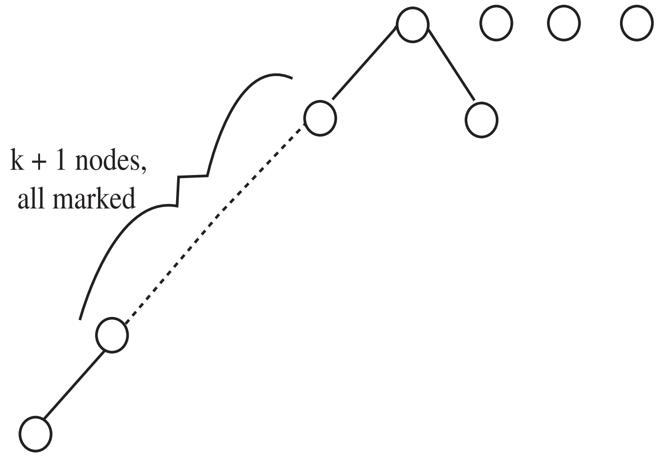
, then a decrease key on *a* and two more inserts to produce



Given a structure as above with a path of *k* nodes, we expand it by one as follows: Two fresh inserts and one delete min will produce



Doing decrease keys on nodes a, b, and c results in



as desired.

4.(a) (b) I will describe a solution to the problem assuming that the items are selected from a totally ordered set and stored in the internal nodes of the tree in symmetric order. Essentially the same solution works if the the items are stored in the leaves, with search keys in the non-spine internal nodes, or if the items are searched by position, and subtree sizes are stored in the non-spine internal nodes.

To search for an item, we walk concurrently up from the bottom of the left spine and the bottom of the right spine, going up one black node and zero or one red node at a time, until reaching a node on the left spine that is not smaller than the desired item or a node on the right spine that is not larger than the desired item. We must reach such a node, since the root, which is on both spines, passes the test for one of the spines. We then search down for the item in the standard way. (This search will go down one step on the left or right spine and then branch into a subtree whose root is off of the spine.)

Insertion or deletion works the same way: first we search as above for the location of the item to be inserted or deleted, and then we actually do the insertion or deletion in the standard bottom-up fashion. In the case of a deletion, we may have to search down the tree to find an item to swap for the item to be deleted, so that the node actually deleted has no children or one child. (See red-black tree deletion, Section 14.4 in the text.)

To bound the times of these operations, we use the following result:

Lemma 1 (see Exercise 14.1-4 in the text) *A red-black tree with black root all of whose paths down from the root contain i black nodes itself contains between $2^i - 1$ and $4^i - 1$ nodes (inclusive).*

Proof. Let $L(i)$ be the minimum number of nodes in such a tree and $U(i)$ the maximum number of nodes in such a tree. We have $L(1) = 1$ and $L(i) \geq 1 + 2L(i-1)$ for $i \geq 2$, from which $L(i) \geq 2^i - 1$ follows by induction. We also have $U(1) = 3$ and $U(i) \leq 3 + 4U(i-1)$ for $i \geq 2$, from which $U(i) \leq 4^i - 1$ follows by induction. ■

Now consider searching for the item that happens to be the k^{th} smallest. By the lemma, after walking up past at most $\log_2 k$ black nodes from the bottom of the left spine we will have reached the root of a subtree containing the desired item;searching down into this tree involves walking down past at most $\log_2 k$ black nodes. Hence the search time is $\mathrm{O}(\log k)$. The symmetric argument applies to a search for the item that happens to be the k^{th} largest. This gives the desired time bound for searching; the pointer structure described in the problem supports the search strategy described above.

The time for insertion or deletion is $\mathrm{O}(\log k)$ for the initial search plus $\mathrm{O}(1)$ per rebalancing step (the various update cases for insertion and deletion); performing a rotation at any position in the tree takes $\mathrm{O}(1)$ time, even with the pointers reversed along the left and right spines, as long as we keep track of the nodes traversed along the spines, by for example putting them into two stacks. With the potential function defined in the problem, each nonterminating rebalancing step in either an insertion or deletion decreases the potential by at least one, as can be easily checked. (Recoloring steps are the only nonterminating steps.) A terminating rebalancing step increases the potential by at most a constant since it affects only a constant number of nodes. It follows that the amortized rebalancing time after an insertion or deletion is $\mathrm{O}(1)$, and the total amortized time for an insertion or deletion is $\mathrm{O}(\log k)$.