SKEW HEAPS: Self-Adjusting Heaps

In this handout we describe the *skew heap* data structure, a self-adjusting form of heap related to the leftist heap of Crane and Knuth. Skew heaps, in contrast to leftist heaps, use less space and are easier to implement, and yet in the amortized sense they are as efficient, to within a constant factor, as leftist heaps.

We will consider the following priority queue operations:

makeheap(h):	Create a new, empty heap, named h.
findmin(h):	Return a minimum key in h. If h is empty, return the special key "null". (This operation does not change the set of keys stored in h.)
insert(k, h):	Insert key k in heap h. (Duplicate keys are allowed in the heap. That is, the key to be inserted is treated as "new".)
deletemin(h):	Delete a minimum key from heap h and return it. If the heap is initially empty, return null.
union(h_1, h_2):	Return the heap formed by joining (i.e., taking the union of) heaps h_1 and h_2. This operation distroys h_1 and h_2.

THE DATA STRUCTURE

There are several ways to implement heaps in a self-adjusting fashion. The one we shall discuss is called *skew heaps* as proposed by Sleator and Tarjan, and is analogous to leftist heaps. A skew heap is a *heap-ordered binary tree*. That is, it is a binary tree with one key in each node so that for each node x other that the root, the key at node x is no less than the key at the parent of x. To represent such a tree we store in each node x its associated key, denoted *key*(x) and two pointers *left*(x) and *right*(x), to its left child and right child, respectively. If x has no left child we define *left*$(x) = \Lambda$; if x has no right child we define *right*$(x) = \Lambda$. Access to the tree is by a pointer to its root; we represent an empty tree by a pointer to Λ.

With this representation we can carry out the various heap operations as follows. We perform *makeheap*(h) in $O(1)$ time by initializing h to Λ. Since heap order implies that the root is a minimum key in the tree, we can carry out *findmin*(h) in $O(1)$ time by returning the key at the root; returning null if the heap is empty. We perform *insert* and *deletemin* using *union*. To carry out *insert*(k, h), we make k into a one-node heap and Union it with h. To carry out *deletemin*(h), if h is not empty we replace h by the Union of its left and right subtrees and return the key at the original root. (If h is originally empty we return null.)

To perform *union*(h_1, h_2), we form a single tree by traversing down the right paths of h_1 and h_2, merging them into a single right path with keys in nondecreasing order. First assume the left subtrees of nodes along the merge path do not change. (See Figure 1(a).) The time for the Union operation is bounded by a constant times the length of the merge path. To make Union efficient, we must keep right paths short. In leftist heaps this is done by maintaining the invariant that, for any node x, the right path descending from x is a shortest path down to a missing node. Maintaining this invariant requires storing at every node the length of a shortest path down to a missing node; after the merge we walk back up the merge path, updating the shortest path lengths and swapping left and right children as necessary to maintain the leftist property. The length of the right path in a leftist heap of n nodes is at most $\log n$, implying an $O(\log n)$ worst-case time bound for each of the heap operations, where n is the number of nodes in the heap or heaps involved.

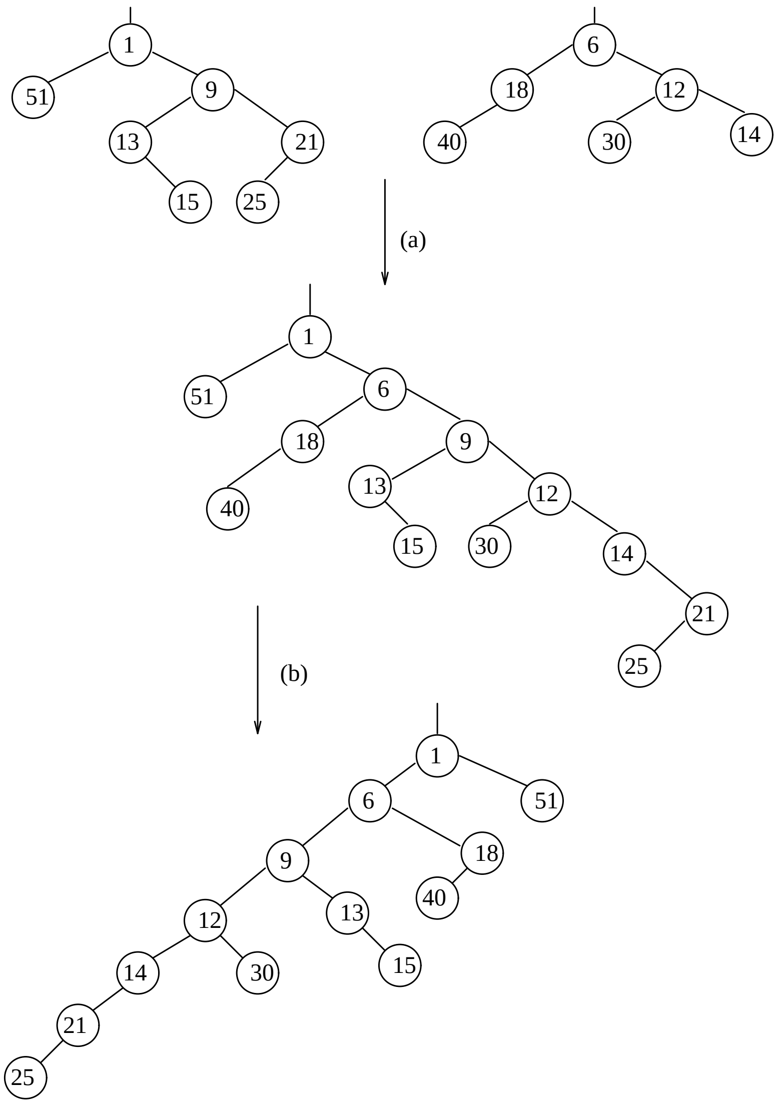


Figure 1: A Union of two skew heaps.
(a) Merge of the right paths.
(b) Swapping of children along the path formed by the merge.

In our self-adjusting version of this data structure, we perform the Union operation by merging the right paths of the two trees and then swapping the left and right children of *every* node on the merge path except the lowest. (See Figure 1(b).) This makes the potentially long right path formed by the merge into a left path. We call the resulting data structure a *skew heap*.

Exercise 1: Write a simple recursive algorithm for the Union operation on skew heaps. □

THE AMORTIZED ANALYSIS

In our analysis of skew heaps we shall use the following general approach. We associate with each collection S of skew heaps a real number $\Phi(S)$ called the potential of S. For any sequence of m operations with running times $t_1, t_2, \cdots, t_m$, we define the *amortized time* a_i of the i^{th} operation to be $a_i = t_i + \Phi_i - \Phi_{i-1}$, where, Φ_i, for $i = 1, 2, \cdots, m$, is the potential of the skew heaps after the i^{th} operation and Φ_0 is the potential of the skew heaps before the first operation. The total running time of the sequence of the operations is then

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i - \Phi_i + \Phi_{i-1}) = \Phi_0 - \Phi_m + \sum_{i=1}^{m} a_i$$

That is, the total running time equals the total amortized time plus the net decrease in potential from the initial to the final collection of the skew heaps. In our analysis of skew heaps the potential will be zero initially and will remain nonnegative. This implies that the total amortized time is an upper bound on the actual running time over the entire sequence of operations.

We shall define the potential of a single skew heap; the potential of a collection of skew heaps is the sum of the potentials of its members. Intuitively, a heap with high potential is one subject to unusually time-consuming operations; the extra time spent corresponds to a drop in potential.

Definition 1: For any node x in a binary tree, we define the *weight* of x, denoted $wt(x)$, to be the number of descendents of x, including x itself.

We use weights to partition nonroot nodes into two classes:

Definition 2: A nonroot node x is *heavy* if $wt(x) > wt(parent(x)) / 2$ and *light* otherwise. A root node is considered neither heavy nor light.

The following facts immediately follow from the above definitions. (Try to prove them yourself.)

Fact 1: Of the children of any node, at most one is heavy. □

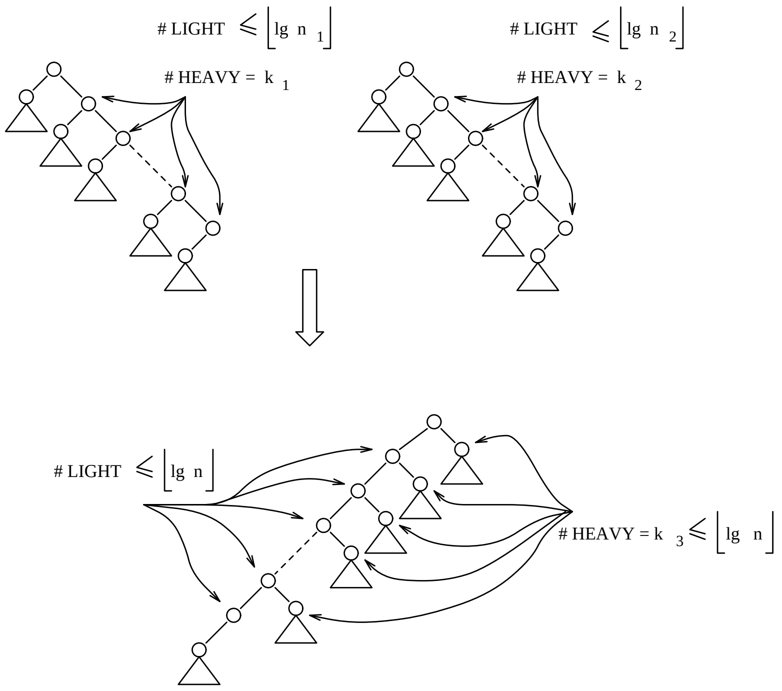
Fact 2: On any path from a node x down to a descendent y, there are at most $\lfloor \log (wt(x) / wt(y)) \rfloor$ light nodes, not counting x. In particular, any path in an n-node tree contains at most $\lfloor \log n \rfloor$ light nodes. □

Definition 3: A nonroot node is called *right* if it is the right child of its parent; it is called *left* otherwise.

Definition 4: We define the potential of a skew heap to be the total number of right heavy nodes it contains.

Suppose we begin with no heaps and carry out a sequence of skew heap operations. The initial potential is zero and the final potential is nonnegative, so the total amortized time is an upper bound on the total actual time of the operations. The amortized time of a *make-heap* or *findmin* operation is $O(1)$, since these operations require $O(1)$ time and do not change the potential. The amortized time of the other operations depend on the amortized time for the *union* operation.

Consider a Union of two heaps h_1 and h_2, containing n_1 and n_2 nodes, respectively. Let $n = n_1 + n_2$ be the total number of nodes in the two heaps. As a measure of the time for Union, we shall charge one per node on the merge path. Thus the amortized time of Union is the number of nodes on the merge path plus the change in potential. By Fact 2, the number of light nodes on the right paths of h_1 and h_2 is at most $\lfloor \log n_1 \rfloor$ and $\lfloor \log n_2 \rfloor$, respectively. Thus the total number of light nodes on the two paths is at most $2\lfloor \log n \rfloor - 1$. (Check this.) (See Figure 2.)



# LIGHT $\leqslant \lfloor \lg n_1 \rfloor$
# HEAVY = k_1
# LIGHT $\leqslant \lfloor \lg n_2 \rfloor$
# HEAVY = k_2
# LIGHT $\leqslant \lfloor \lg n \rfloor$
# HEAVY = $k_3 \leqslant \lfloor \lg n \rfloor$

Figure 2: Analysis of right heavy nodes in *union*.

Let k_1 and k_2 be the number of heavy nodes on the right paths of h_1 and h_2, respectively, and let k_3 be the number of nodes that *become* right heavy children of nodes on the merge path. By Fact 1, every node counted by k_3 corresponds to a light node on the merge path. Thus Fact 2 implies that $k_3 \le \lfloor \log n \rfloor$.

The number of nodes on the merge path is at most $2 + \lfloor \log n_1 \rfloor + k_1 + \lfloor \log n_2 \rfloor + k_2 \le 1 + 2\lfloor \log n \rfloor + k_1 + k_2$. (The first "2" counts the roots of h_1 and h_2.) Therefore the running time of the Union operation is $t \le 1 + 2\lfloor \log n \rfloor + k_1 + k_2$. The change in potential caused by the Union is $\Delta\Phi = k_3 - k_1 - k_2 \le \lfloor \log n \rfloor - k_1 - k_2$. Thus the amortized time of the Union, $t + \Delta\Phi$, is at most $3\lfloor \log n \rfloor + 1$.

We summarize the above result in the following theorem.

Theorem: The amortized time of an *insert*, *deletemin*, or *union* skew heap operation is $O(\log n)$, where n is the number of keys in the heap or heaps involved in the operation. The amortized time of a *makeheap* or *findmin* operation is $O(1)$.

Proof: The analysis above gives the bound for *findmin*, *makeheap*, and *union*; the bound for *insert* and *deletemin* follows immediately from that of *union*. □

Bibliography:

To see further improvements to the basic structure described above, the interested reader is referred to:
Sleator, Tarjan, *"Self-Adjusting Heaps,"* SIAM J. Computing, 15(1), 1986, pp. 52-69.

Bernard Chazelle’s papers sited below describe a novel self adjusting priority queue that allows a controled measure of error in order to accelerate performance! He shows many applications, including the fastest deterministic minimum spanning tree algorithm todate:
"The Soft Heap: An Approximate Priority Queue with Optimal Error Rate", JACM 47(6), Nov. 2000, pp. 1012-1027.
"A Minimum Spanning Tree Algorithm with inverse-Ackermann Type Complexity" JACM 47(6), Nov. 2000, pp. 1028-1047.