LEFTIST HEAPS

Leftist heaps are a data structure for representing priority queues. They were discovered by Clark A. Crane [1971] (also described in Knuth, Art of Computer Programming, vol. 3, 1973, pp. 151-152) and have the following nice properties:

- INSERT and DELETEMIN operations can be performed in $O(\log n)$ time in the worst case — as with standard heaps.
- In addition, the UNION operation joins two leftist heaps with $n$ and $m$ nodes, respectively, into a single leftist heap in $O(\log(\max(m, n)))$ time in the worst case. The two old heaps are destroyed as a result. (Other names used in the literature for UNION are meld, join, merge.)
- The rearrangement of nodes following an INSERT, DELETEMIN or UNION operation involves changing pointers, not moving records. In contrast, in the array representation of standard heaps, upheap and downheap involves exchanging contents of array elements. The difference could be significant if the elements in the priority queue are sizeable objects (e.g. themselves arrays), in which case we can no longer assume that exchanging two array entries takes constant time.†

Definition 1: The distance of a node $m$ in a tree, denoted $\text{dist}[m]$, is the length of the shortest path from $m$ down to a descendant node that has at most one child.

Definition 2: A leftist heap is a binary tree such that for every node $m$,
(a) $\text{key}[m] \leq \text{key}[\text{lchild}[m]]$ and $\text{key}(m) \leq \text{key}[\text{rchild}[m]]$, and
(b) $\text{dist}[\text{lchild}[m]] \geq \text{dist}[\text{rchild}[m]]$.

In the above definition, $\text{key}[m]$ is the key stored at node $m$. We assume that there is a total ordering among the keys. We also assume that $\text{key}[\text{nil}] = \infty$ and $\text{dist}[\text{nil}] = -1$.

Definition 3: The right path of a tree is the path $m_1, m_2, \cdots, m_k$ where $m_1$ is the root of the tree, $m_{i+1} = \text{rchild}[m_i]$ for $1 \leq i < k$, and $\text{rchild}[m_k] = \text{nil}$.

Figure 1 below shows two examples of leftist heaps.

Here are a few simple results about leftist heaps that you should be able to prove easily:

Fact 1: The left and right subtrees of a leftist heap are leftist heaps. □

Fact 2: The distance of a leftist heap’s root is equal to the length of the tree’s right path. □

Fact 3: For any node $m$ of a leftist heap, $\text{dist}[m] = \text{dist}[\text{rchild}[m]] + 1$ (where, as usual, we take $\text{dist}[\text{nil}] = -1$). □

† Note, however, that in this case we could still use the array representation for heaps, now storing in its entries pointers to the heap’s nodes rather than the (large) nodes themselves. Then upheap and downheap can be done by exchanging pointers, while leaving the nodes themselves fixed.
In each node we record its key at the top half and its distance at the bottom half. The right path of $T_1$ is 1, 5, 8 while the right path of $T_2$ is 1.

In the examples above, $T_2$ illustrates the fact that leftist heaps can be unbalanced. However, in INSERT, DELETEMIN and UNION, all activity takes place along the right path which, the following theorem shows, is short.

**Theorem:** If the length of the right path in a leftist tree $T$ is $k$ then $T$ has at least $2^{k+1} - 1$ nodes.

**Proof:** By induction on the height $h$ of $T$.

**Basis** ($h=0$): Then $T$ consists of a single node and its right path has length $k=0$. Indeed, $T$ has $1 \geq 2^1 - 1$ nodes, as wanted.

**Inductive Step** ($h>0$): Suppose the theorem holds for all leftist heaps that have height $< h$ and let $T$ be a leftist heap of height $h$. Further, let $k$ be the length of $T$’s right path and $n$ be the number of nodes in $T$. Consider two cases:

**Case 1:** $k=0$ (i.e. $T$ has no right subtree). But then clearly $n \geq 1 = 2^1 - 1$, as wanted.

**Case 2:** $k>0$. Let $T_L, T_R$ be the left and right subtrees of $T$; $n_L$, $n_R$ be the number of nodes in $T_L$ and $T_R$; and $k_L$, $k_R$ be the lengths of the right paths in $T_L$ and $T_R$ respectively. By Fact 1, $T_R$ and $T_L$ are both leftist heaps. By Facts 2 and 3, $k_R = k - 1$, and by definition of leftist tree $k_L \geq k_R$. Since $T_L, T_R$ have height $< h$ we get, by induction hypothesis, $n_R \geq 2^k - 1$ and $n_L \geq 2^{k-1} - 1$. But $n = n_L + n_R + 1$ and thus, $n \geq 2^k - 1 + 2^{k-1} + 1 = 2^{k+1} - 1$. Therefore $n \geq 2^{k+1} - 1$, as wanted. □

From this we immediately get

**Corollary:** The right path of a leftist heap with $n$ nodes has length $\leq \lceil \log(n+1) \rceil - 1$. □

Now let’s examine the algorithm for joining two leftist heaps. The idea is simple: if one of the two trees is empty we’re done; otherwise we want to join two non-empty trees $T_1$ and $T_2$ and we can assume, without loss of generality, that the key in the root of $T_1$ is $\leq$ the key in the root of $T_2$. Recursively we join $T_2$ with the right subtree of $T_1$ and we make the resulting leftist heap into the right subtree of $T_1$. If this has made the distance of the right subtree’s root longer than the distance of the left subtree’s root, we simply interchange the left and right children of $T_1$’s root (thereby making what used to be the right
subtree of \( T_1 \) into its left subtree and vice-versa). Finally, we update the distance of \( T_1 \)'s root. The following pseudo-code gives more details.

We assume that each node of the leftist heap is represented as a record with the following format

| lchild | key  | dist | rchild |

where the fields have the obvious meanings. A leftist heap is specified by giving a pointer to its root.

/* The following algorithm joins two leftist heaps whose roots are pointed at by \( r_1 \) and \( r_2 \), and returns a pointer to the root of the resulting leftist heap. */

function UNION(\( r_1 \), \( r_2 \))
if \( r_1 = \text{nil} \) then return \( r_2 \)
else if \( r_2 = \text{nil} \) then return \( r_1 \)
else
    if key[\( r_1 \)] > key[\( r_2 \)] then \( r_1 \leftarrow r_2 \)
    rchild[\( r_1 \)] \( \leftarrow \) UNION(\( rchild[\( r_1 \)] \), \( r_2 \))
    if \( d(\text{rchild}[\( r_1 \)]) > d(\text{lchild}[\( r_1 \)]) \)
        then \( \text{rchild}[\( r_1 \)] \leftarrow \text{lchild}[\( r_1 \)] \)
    dist[\( r_1 \)] \( \leftarrow d(\text{rchild}[\( r_1 \)]) + 1 \)
    return \( r_1 \)
end {UNION}

function \( d(x) \) /* returns \( \text{dist}(x) \) */
if \( x = \text{nil} \) then return \(-1\)
else return \( \text{dist}[x] \)
end {d}

What is the complexity of this algorithm? First, observe that there is a constant number of steps that must be executed before and after each recursive call to UNION. Thus the complexity of the algorithm is proportional to the number of recursive calls to UNION. It is easy to see that, in the worst case, this will be equal to \( p_1 + p_2 \) where \( p_1 \) (respectively \( p_2 \)) is 1 plus the length of the right path of the leftist heap whose root is pointed at by \( r_1 \) (respectively \( r_2 \)). Let the number of nodes in these trees be \( n_1 \), \( n_2 \). By the above Corollary we have \( p_1 \leq \lfloor \log (n_1 + 1) \rfloor \), \( p_2 \leq \lfloor \log (n_2 + 1) \rfloor \). Thus \( p_1 + p_2 \leq \log n_1 + \log n_2 + 2 \). Let \( n = \max(n_1, n_2) \). Then \( p_1 + p_2 \leq 2 \log n + 2 \). Therefore, UNION is called at most \( 2 \log n + 2 \) times and the complexity of the algorithm is \( O(\log (\max(n_1, n_2))) \) in the worst case.

Figure 2 below shows an example of the UNION operation.

Armed with the UNION algorithm we can easily write algorithms for INSERT and DELETEMIN:
Figure 2. The UNION operation.

**INSERT**($e, r$) \{ $e$ is an element, $r$ is pointer to root of tree \}
1. Let $r'$ be a pointer to the leftist heap containing only $e$
2. \textbf{return} UNION ($r', r$).

**DELETEMIN**($r$)
1. $min \leftarrow$ element stored at $r$ (root of leftist heap)
2. $r \leftarrow$ UNION (lchild[$r$], rchild[$r$])
3. \textbf{return} $min$.

By our analysis of the worst case time complexity of UNION it follows immediately that the complexity of both these algorithms is $O(\log n)$ in the worst case, where $n$ is the number of nodes in the leftist heap.

In closing, we note that INSERT can be written as in the heap representation of priority queues, by adding the new node at the end of the right path, percolating its value up (if necessary), and switching right and left children of some nodes (if necessary) to maintain the properties of the leftist heap after the insertion. As an exercise, write the INSERT algorithm for leftist heaps in this fashion. On the other hand, we cannot use the idea of percolating values down to implement DELETEMIN in leftist heaps the way we did in standard heaps: doing so would result in an algorithm with $O(n)$ worst case complexity. As an exercise, construct a leftist heap where this worst case behaviour would occur.