NOTES ON LEFTIST TREES

by Vassos Hadilacos

Leftist trees are a data structure for representing priority queues. They were invented by Clark Crane and have the following nice properties:

- INSERT and EXTRACTMIN operations can be performed using time in $O(\log n)$ in the worst case — as with heaps.
- In addition, we can merge two leftist trees with n_1 and n_2 nodes respectively into a single leftist tree using time in $O\big(\log(\max(n_1, n_2))\big)$ in the worst case.

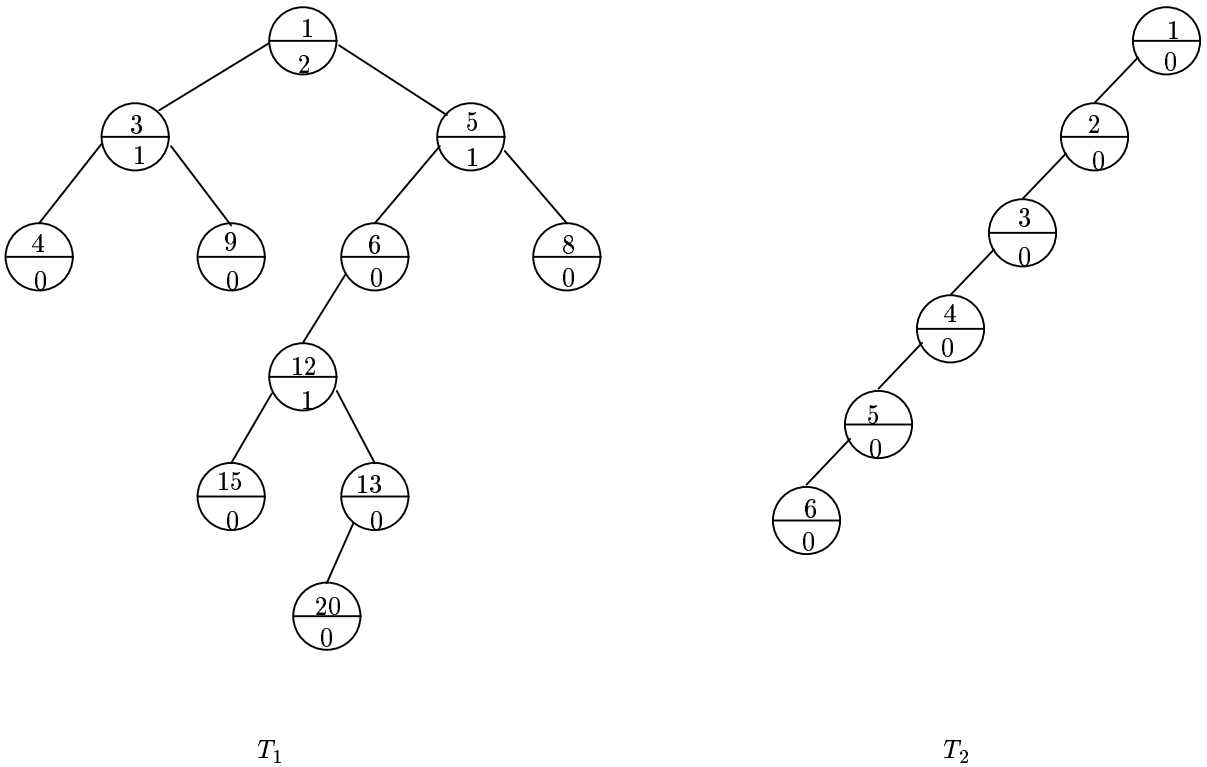
Definition. *The distance of a node in a binary tree is the length of the shortest path from the node to a descendant that has at most one child.*

Definition. *A leftist tree is a binary tree every node u of which contains a key, $key(u)$, and the node's distance, $dist(u)$, so that*

(a) $key(u) \leq \min\big(key(lchild(u)), key(rchild(u))\big)$, *and*

(b) $dist(lchild(u)) \geq dist(rchild(u))$.

In the above definition we assume that $key(\Lambda) = \infty$ and $dist(\Lambda) = -1$, where Λ denotes a left or right child that does not exist (i.e., is nil). Two leftist trees are shown in Figure 1. Each node contains a key (written on the upper half) and its distance (written on the lower half).



T_1 T_2

Figure 1

Definition. *The right path of a binary tree is the path $u_1, u_2, \ldots, u_k$, where u_1 is the root of the tree, $u_{i+1} = rchild(u_i)$ for $1 \leq i < k$, and $rchild(u_k) = \Lambda$.*

Here are a few simple properties of leftist trees that you should be able to prove easily:

Fact 1. *The left and right subtrees of a leftist tree are leftist trees.*

Fact 2. *The distance of a leftist tree's root is equal to the length of the tree's right path.*

Fact 3. *For any node u of a leftist tree, $dist(u) = dist(rchild(u)) + 1$.*

Tree T_2 in Figure 1 illustrates the fact that leftist trees can be unbalanced. However, in INSERT, EXTRACTMIN and MERGE, all activity takes place along the right path which, as the following theorem shows, is short.

Theorem. *Let T be a nonempty leftist tree, k be the length of its right path, and n is the number of its nodes. Then $n \geq 2^{k+1} - 1$.*

PROOF: By (complete) induction on the height h of T. Since T is nonempty, $n \geq 1$ and $h \geq 0$. Suppose the theorem holds for all leftist trees of height less than h.

Case 1: $k = 0$. We have $n \geq 1 = 2^1 - 1$. So, in this case $n \geq 2^{k+1} - 1$, 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. Since T_L and T_R are the subtrees of T, they are both leftist trees (by Fact 1), and have height less than h. Moreover, since $k > 0$, T_R is nonempty, and since T is a leftist tree so is T_L. Therefore, the induction hypothesis applies to T_L and T_R, and we have $n_L \geq 2^{k_L+1} - 1$ and $n_R \geq 2^{k_R+1} - 1$. By Fact 3, $k_R = k - 1$. By the definition of leftist tree and Fact 2, $k_L \geq k_R$, and so $k_L \geq k - 1$. We have, $n = n_L + n_R + 1 \geq (2^{k_L+1} - 1) + (2^{k_R+1} - 1) + 1 \geq (2^k - 1) + (2^k - 1) + 1 = 2^{k+1} - 1$, as wanted. □

Corollary. *The right path of a leftist tree with n nodes has length at most $\lceil \log_2(n+1) \rceil - 1$.*

Now let's examine the algorithm for merging two leftist trees. The idea is simple: if one of the two trees is empty we're done; otherwise we want to merge 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 less than or equal to the key in the root of T_2. Recursively we merge T_2 with the right subtree of T_1, and we make the resulting leftist tree 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 pseudocode below gives more details.

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

lchild	key	dist	rchild

where the fields have the obvious meanings. A leftist tree is specified by giving a pointer to its root. The following algorithm merges two leftist trees whose roots are pointed at by r_1 and r_2, and returns a pointer to the root of the resulting leftist tree.† (See Figure 2 for an example of merging two leftist trees.)

† We use "$r_1 \leftrightarrow r_2$" as an abbreviation for "$temp := r_1$; $r_1 := r_2$; $r_2 := temp$".

```
MERGE(r1, r2)
    if r1 = Λ then return r2
    elsif r2 = Λ then return r1
    else
        if key(r1) > key(r2) then r1 ↔ r2 end if
        rchild(r1) := MERGE(rchild(r1), r2)
        if rchild(r1) ≠ Λ and (lchild(r1) = Λ or dist(rchild(r1)) > dist(lchild(r1))) then
            rchild(r1) ↔ lchild(r1)
        end if
        if rchild(r1) = Λ then dist(r1) := 0
        else dist(r1) := dist(rchild(r1)) + 1
        end if
        return r1
    end if
end MERGE
```

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 MERGE. Thus the complexity of the algorithm is proportional to the number of recursive calls to MERGE. 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 tree whose root is pointed at by r_1 (respectively, r_2). Let the number of nodes in these trees be n_1, and n_2. By the above corollary we have $p_1 \leq \lceil \log(n_1 + 1) \rceil$, and $p_2 \leq \lceil \log(n_2 + 1) \rceil$. 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, MERGE 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.

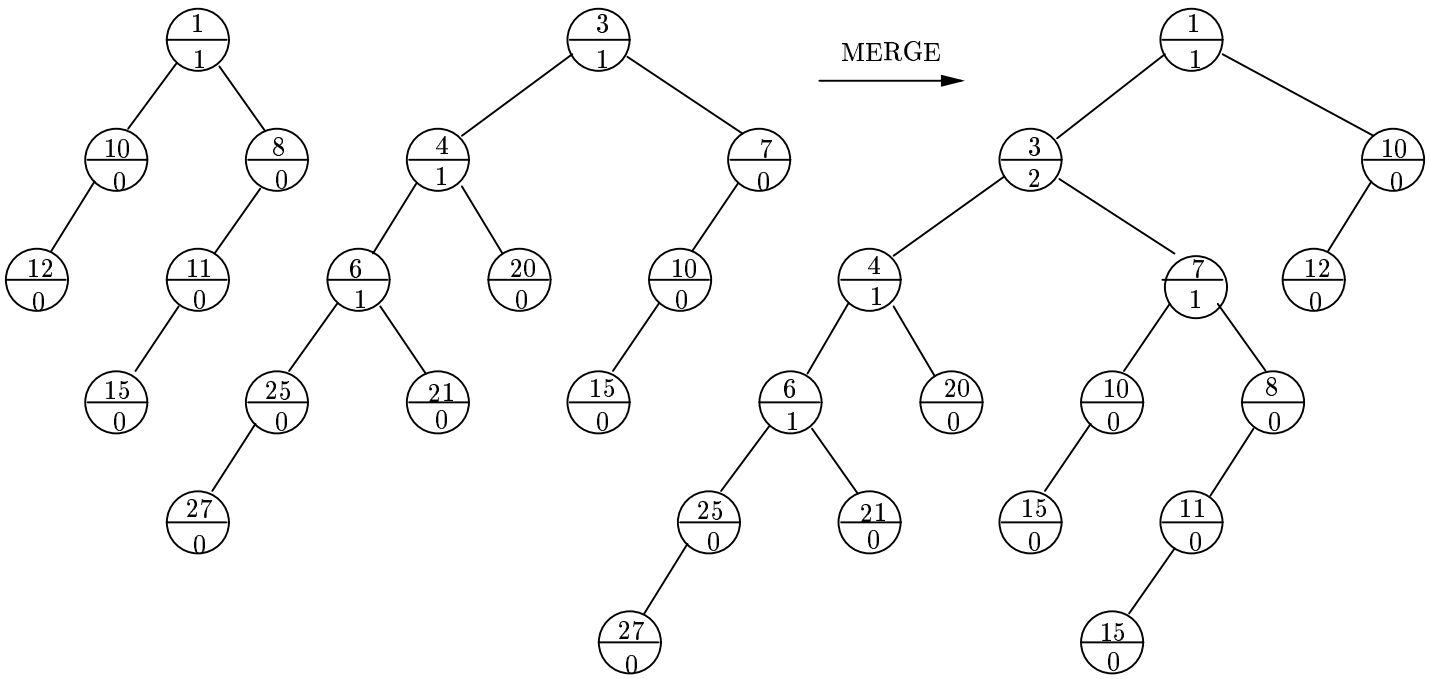


Figure 2: Merging two leftist trees

Using the MERGE algorithm we can easily write algorithms for INSERT and EXTRACTMIN:

INSERT(e, r) { e is an element, r is pointer to root of tree}

1. Let r' be a pointer to the leftist tree containing only e.
2. MERGE(r', r).

EXTRACTMIN(r)

1. min := element stored at r (root of leftist tree);
2. r := MERGE($lchild(r)$, $rchild(r)$);
3. **return** min.

By our analysis of the worst case time complexity of MERGE, 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 tree.

The INSERT operation can also be implemented 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 node (if necessary) to maintain the properties of leftist trees after the insertion. As an exercise, write the INSERT algorithm for leftist trees in this fashion. On the other hand, we *cannot* use the idea of percolating values down to implement EXTRACTMIN in leftist trees the way we did in heaps: doing so would result in an algorithm with $\Theta(n)$ worst case complexity. As an exercise, construct a leftist tree where this worst case behaviour would occur.

Also, it is possible to implement the MERGE operation in an iterative, rather than recursive, way. Roughly speaking, the idea is this: We traverse the right paths of the two leftist trees we wish to merge, merging them into a single right path, called the *merge path*. The nodes of the two right paths are added to the merge path in increasing key order. As a node is added to the merge path, we also "hang" its left subtree on the left side of the merge path (which, recall, is a right path). When we have completed the merging of the two right paths in this way, we will have constructed a binary tree containing the nodes of the two trees, and satisfying the heap-order property. However, the tree is not necessarily a leftist tree because for some nodes u on the merge path it may be that $dist(lchild(u)) < dist(rchild(u))$. So, we traverse the merge path once again but now in *bottom-up* fashion, exchanging the left and right subtrees of nodes when necessary, and updating the *dist* field of each node appropriately, so that at the end we have a leftist tree with the keys of the two trees that were merged. As an exercise, you should write the pseudocode for this algorithm, analyse its complexity, and prove that it produces *exactly* the same leftist tree as the recursive implementation of MERGE.