Lecture 14: Leftist and Skew Heaps

Reading: Sections 6.6 and 6.7 in Weiss.

Leftist Heaps: The standard binary heap data structure is an simple and efficient data structure for the basic priority queue operations `insert(x)` and `x = extractMin()`. It is often the case in data structure design that the user of the data structure wants to add additional capabilities to the abstract data structure. When this happens, it may be necessary to redesign components of the data structure to achieve efficient performance.

For example, consider an application in which in addition to insert and extractMin, we want to be able to *merge* the contents of two different queues into one queue. As an application, suppose that a set of jobs in a computer system are separate queues waiting for the use of two resources. If one of the resources fails, we need to merge these two queues into a single queue.

We introduce a new operation `Q = merge(Q1, Q2)`, which takes two existing priority queues Q_1 and Q_2, and merges them into a new priority queue, Q. (Duplicate keys are allowed.) This operation is *destructive*, which means that the priority queues Q_1 and Q_2 are destroyed in order to form Q. (Destructiveness is quite common for operations that map two data structures into a single combined data structure, since we can simply reuse the same nodes without having to create duplicate copies.)

We would like to be able to implement `merge()` in $O(\log n)$ time, where n is the total number of keys in priority queues Q_1 and Q_2. Unfortunately, it does not seem to be possible to do this with the standard binary heap data structure (because of its highly rigid structure and the fact that it is stored in an array, without the use of pointers).

We introduce a new data structure called a *leftist heap*, which is fairly simple, and can provide the operations `insert(x)`, `extractMin()`, and `merge(Q1,Q2)`. This data structure has many of similar features to binary heaps. It is a binary tree which is *partially ordered*, which means that the key value in each parent node is less than or equal to the key values in its children's nodes. However, unlike a binary heap, we will not require that the tree is complete, or even balanced. In fact, it is entirely possible that the tree may be quite unbalanced.

Leftist Heap Property: Define the *null path length*, npl(v), of any node v to be the length of the shortest path to a descendent has either 0 or 1 child. The value of npl(v) can be defined recursively as follows.

$$\text{npl}(v) = \begin{cases} -1 & \text{if } v = \textit{null}, \\ 1 + \min(\text{npl}(v.\textit{left}), \text{npl}(v.\textit{right})) & \text{otherwise.} \end{cases}$$

We will assume that each node has an extra field, `v.npl` that contains the node's npl value. The *leftist property* states that for every node v in the tree, the npl of its left child is at least as large as the npl of its right child. We say that the keys of a tree are *partially ordered* if each node's key is greater than or equal to its parent's key.

Leftist heap: Is a binary tree whose keys are partially ordered (parent is less than or equal to child) and which satisfies the leftist property (npl(*left*) $\geq$ npl(*right*)). An example is shown in the figure below, where the npl values are shown next to each node.

Note that any tree that does not satisfy leftist property can always be made to do so by swapping left and right subtrees at any nodes that violate the property. Observe that this does not affect the partial ordering property. Also observe that satisfying the leftist heap property does not imply that the tree is balanced. Indeed, a degenerate binary tree in which

[1]Copyright, David M. Mount, 2001

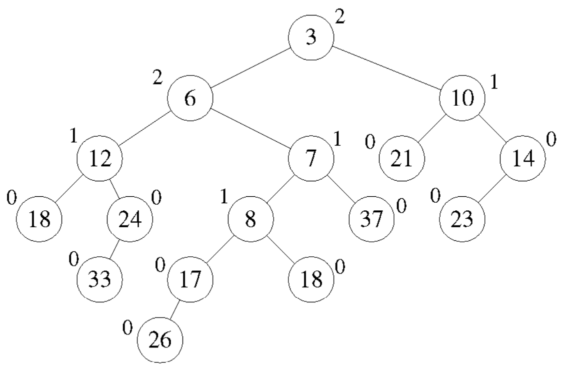

Figure 40: Leftist heap structure (with npl values shown).

is formed from a chain of nodes each attached as the left child of its parent does satisfy this property.

The key to the efficiency of leftist heap operations is that there exists a short ($O(\log n)$ length) path in every leftist heap (namely the rightmost path). We prove the following lemma, which implies that the rightmost path in the tree cannot be of length greater than $O(\log n)$.

Lemma: A leftist heap with $r \geq 1$ nodes on its rightmost path has has at least $2^r - 1$ nodes.

Proof: The proof is by induction on the size of the rightmost path. Before beginning the proof, we begin with two observations, which are easy to see: (1) the shortest path in any leftist heap is the rightmost path in the heap, and (2) any subtree of a leftist heap is a leftist heap. For the basis case, if there is only one node on the rightmost path, then the tree has at least one node. Since $1 \geq 2^1 - 1$, the basis case is satisfied.

For the induction step, let us suppose that the lemma is true for any leftist heap with strictly fewer than r nodes on its rightmost path, and we will prove it for a binary tree with exactly r nodes on its rightmost path. Remove the root of the tree, resulting in two subtrees. The right subtree has exactly $r - 1$ nodes on its rightmost path (since we have eliminated only the root), and the left subtree must have a at least $r - 1$ nodes on its rightmost path (since otherwise the rightmost path in the original tree would not be the shortest, violating (1)). Thus, by applying the induction hypothesis, it follows that the right and left subtrees have at least $2^{r-1} - 1$ nodes each, and summing them, together with the root node we get a total of at least

$$2(2^{r-1} - 1) + 1 = 2^r - 1$$

nodes in the entire tree.

Leftist Heap Operations: The basic operation upon which leftist heaps are based is the merge operation. Observe, for example, that both the operations `insert()` and `extractMin()` can be implemented by using the operation `merge()`. (Note the similarity with splay trees, in which all operations were centered around the splay operation.)

The basic node of the leftist heap is a `LeftHeapNode`. Each such node contains an data field of type `Element` (upon which comparisons can be made) a left and right child, and an npl value. The constructor is given each of these values. The leftist heap class consists of a single `root`, which points to the root node of the heap. Later we will describe the main procedure `merge(LeftHeapNode h1, LeftHeapNode h2)`, which merges the two (sub)heaps rooted at h_1

and h_2. For now, assuming that we have this operation we define the main heap operations. Recall that merge is a destructive operation.

Leftist Operations

```
    void insert(Element x) {
        root = merge(root, new LeftHeapNode(x, null, null, 0))
    }

    Element extractMin() {
        if (root == null) return null       // empty heap
        Element minItem = root.data         // minItem is root's element
        root = merge(root.left, root.right)
return minItem
    }

    void merge(LeftistHeap Q1, LeftistHeap Q2) {
        root = merge(Q1.root, Q2.root)
    }
```

Leftist Heap Merge: All that remains, is to show how `merge(h1, h2)` is implemented. The formal description of the procedure is recursive. However it is somewhat easier to understand in its nonrecursive form. Let h_1 and h_2 be the two leftist heaps to be merged. Consider the rightmost paths of both heaps. The keys along each of these paths form an increasing sequence. We could merge these paths into a single sorted path (as in merging two lists of sorted keys in the merge-sort algorithm). However the resulting tree might not satisfy the leftist property. Thus we update the npl values, and swap left and right children at each node along this path where the leftist property is violated. A recursive implementation of the algorithm is given below. It is essentially the same as the one given in Weiss, with some minor coding changes.

Merge two leftist heaps

```
    LeftHeapNode merge(LeftHeapNode h1, LeftHeapNode h2) {
        if (h1 == null) return h2              // if one is empty, return the other
        if (h2 == null) return h1
        if (h1.data > h2.data)                 // swap so h1 has the smaller root
            swap(h1, h2)

        if (h1.left == null)                   // h1 must be a leaf in this case
            h1.left = h2
        else {                                 // merge h2 on right and swap if needed
            h1.right = merge(h1.right, h2)
            if (h1.left.npl < h1.right.npl) {
                swap(h1.left, h1.right)        // swap children to make leftist
            }
            h1.npl = h1.right.npl + 1          // update npl value
        }
        return h1
    }
```

For the analysis, observe that because the recursive algorithm spends $O(1)$ time for each node on the rightmost path of either h_1 or h_2, the total running time is $O(\log n)$, where n is the total number of nodes in both heaps.

This recursive procedure is a bit hard to understand. A somewhat easier *2-step interpretation* is as follows. First, merge the two right paths, and update the npl values as you do so. Second, walk back up along this path and swap children whenever the leftist condition is violated. The figure below illustrates this way of thinking about the algorithm. The recursive algorithm just combines these two steps, rather than making the separate.

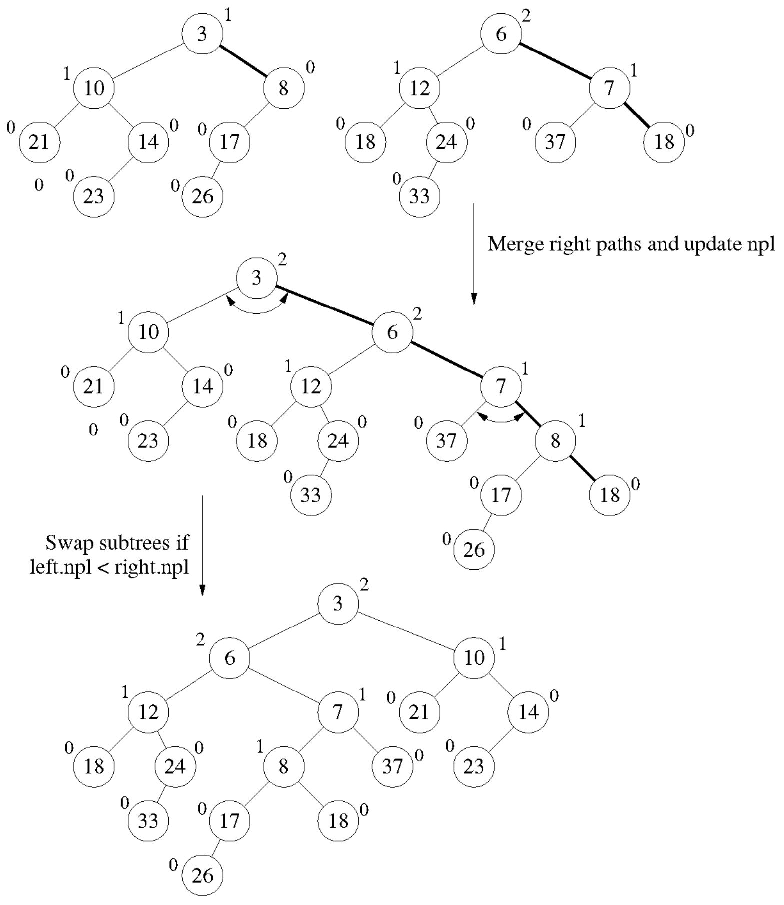
Merge right paths and update npl
Swap subtrees if left.npl < right.npl

Figure 41: Example of merging two leftist heaps (2-step interpretation.

Skew Heaps: Recall that a splay tree is a self-adjusting balanced binary tree. Unlike the AVL tree, which maintains balance information at each node and performs rotations to maintain balance, the splay tree performs a series of rotations to continuously "mix-up" the tree's structure, and hence achieve a sort of balance from an amortized perspective.

A *skew heap* is a self-organizing heap, and applies this same idea as in splay trees, but to the leftist heaps instead. As with splay trees we store no balance information with each node (no npl values). We perform the merge operation as described above, but we swap the left and right children from *every* node along the rightmost path. (Note that in the process the old rightmost path gets flipped over to become the leftmost path.) An example is given in Weiss, in Section 6.7.

It can be shown that if the heap contains at most n elements, then a sequence of m heap operations (insert, extractMin, merge) starting from empty heaps takes $O(m \log n)$ time. Thus the average time per operation is $O(\log n)$. Because no balance information is needed, the code is quite simple.