Lecture 23 — Augmenting Balanced Trees and Leftist Heaps (DRAFT)

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — Nov 15, 2011

Today:

- Independence
- Continue on Augmenting Balanced Trees (see notes from class 21)
- Priority Queues and Leftists Heaps

1 Independence

In problem 5c of Exam 2 we asked whether the $O(\log n)$ bounds on the expected size of a subtree in a treap implied $O(\log n)$ bounds with high probability. When discussing the depth of a node in a subtree we argued that $O(\log n)$ expected bounds implied high-probability bounds so the question is whether this carries over to subtree size. Recall that

$$A_{i,j} = \begin{cases} 1 & \text{if } j \text{ is an ancestor of } i \\ 0 & \text{otherwise} \end{cases}$$

where i and j are the locations of keys in the sorted order. Here, we assume a node is an ancestor of itself (i.e., $A_{ii} = 1$). Therefore for the depth of a node (as previously shown) we have

$$D_i = \left(\sum_{j=1}^{n} A_{i,j} \right) - 1$$

where we remove the 1 from the sum since we don't want to include the node itself, and for the size we have

$$S_i = \sum_{j=1}^{n} A_{j,i}$$

where we include the node itself. We also previously analyzed that $\mathbf{Pr}\left[A_{i,j}\right] = 1/(|j-i|+1)$ so we can plug this in and get:

$$\mathbf{E}\left[D_i\right] = H_i + H_{n-i+1} - 2 = O(\log n)$$

and

$$\mathbf{E}\left[S_i\right] == H_i + H_{n-i+1} - 1 = O(\log n)$$

We have already argued that $\mathbf{E}\left[D_i\right] = O(\log n)$ implies that with high probability D_i is $O(\log n)$. To do this we argued that the terms in the sum $\sum_{j=1}^{n} A_{i,j}$, the $A_{i,j}$ are independent. This is what allowed us to apply Chernoff bounds.

order	$A_{3,1}$	$A_{2,1}$	$A_{3,1}\|A_{2,1}$	$A_{1,3}$	$A_{1,2}$	$A_{1,3}\|A_{1,2}$
1,2,3	1	1	1	0	0	-
1,3,2	1	1	1	0	0	-
2,1,3	0	0	-	0	1	0
2,3,1	0	0	-	0	1	0
3,1,2	0	1	0	1	0	-
3,2,1	0	0	-	1	1	1
Pr $[\ldots]$	2/6 = 1/3	3/6 = 1/2	2/3	2/6 = 1/3	3/6 = 1/2	1/3

Figure 1: Considering all 6 possible priority orderings of three keys. The first column represents the ordering: e.g., 1,2,3 indicates that key 1 has the highest priority, key 2 the second highest, and key 3 the lowest. We see that $\mathbf{Pr}[A_{3,1}] = \frac{1}{3}$ while $\mathbf{Pr}[A_{3,1}|A_{2,1}] = \frac{2}{3}$ so $A_{2,1}$ and $A_{3,1}$, which are added to determine the subtree size, are not independent. On the other hand $\mathbf{Pr}[A_{1,3}] = \mathbf{Pr}[A_{1,3}|A_{1,2}] = \frac{1}{3}$ so $A_{1,2}$ and $A_{1,3}$, which are added to determine node depth, are independent.

The question is whether this is also true for the sum $\sum_{j=1}^{n} A_{j,i}$. By meta-reasoning it cannot be the case that with high probability subtree sizes are $O(\log n)$. This is because in every treap many of the nodes have subtrees larger than $O(\log n)$. In particular the size of the root is n and all but some nodes near the leaves have size greater than $O(\log n)$. This must imply that the terms in the sum are not independent.

To see that they are indeed not independent consider just three keys. We have that $S_1 = 1 + A_{2,1} + A_{3,1}$. So the question is whether $A_{2,1}$ and $A_{3,1}$ are independent, i.e. is the probability of one affected by the outcome of the other. The answer is that they are **not** independent. Informally this is because if 1 wins against 2 ($A_{2,1}$) then it is more likely to win against 2 and 3 ($A_{3,1}$). More formally, it is not hard to verify that $\mathbf{Pr}[A_{3,1}] = \frac{1}{3}$ but that $\mathbf{Pr}[A_{3,1}|A_{2,1}] = \frac{2}{3}$ (see Figure 1). Recall that the notation $\mathbf{Pr}[X|Y]$ means the probability that X is true given that Y is true, and that X and Y are only independent if $\mathbf{Pr}[X] = \mathbf{Pr}[X|Y]$. Therefore the probability of $A_{3,1}$ depends on the outcome of $A_{2,1}$ and the two random variables are not independent.

2 Priority Queues

We have already discussed and used priority queues in a few places in this class. We used them as an example of an abstract data type. We also used them in priority first graph search to implement Dijkstra's algorithm, and the A* variant for shortest paths, and Prim's algorithm for minimum spanning trees. As you might have seen in other classes, a priority queue can also be used to implement an $O(n \log n)$ work (time) version of selection sort, often referred to as heapsort. The sort can be implemented as:

```
fun sort(S) =
let
  (* Insert all keys into priority queue *)
  val pq = Seq.iter Q.insert Q.empty S

  (* remove keys one by one *)
  fun sort' pq =
```

```
    case (PQ.deleteMin pq) of
      NONE => S.empyt
    | SOME(v,pq') => Seq.cons(v,sort'(pq'))
in
  sort' pq
end
```

As covered in Exam 1, with a meld operation the insertion into a priority queue can be implemented in parallel using a reduce instead of iter.

```
(* Insert all keys into priority queue *)
val pq = Seq.reduce Q.meld Q.empty (Seq.map Q.singleton S)
```

However, there is no real way to implement removing the keys in parallel unless we use something more powerful than a heap.

Priority queues also have applications elsewhere, including

- Huffman Codes
- Clustering algorithms
- Event simulation
- Kinetic algorithms

Today we will discuss how to implement a meldable priority queue that supports insert, deleteMin, and meld all in $O(\log n)$ time, where n is the size of the resulting priority queue. The structure is called a leftist heap. There are several other ways to achieve these bounds, but leftists heaps are particularly simple and elegant.

Before we get into leftist heaps lets consider some other possibilities and what problems they have.

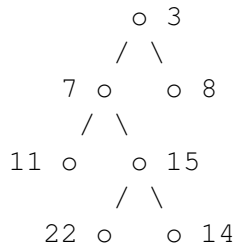
We could use a sorted array.

We could use an unsorted array.

We could use a BST.

2.1 Leftist Heaps

The goal of leftists heaps is to make the `meld` fast, and in particular run in $O(\log n)$ work. As a reminder a *min-heap* is a rooted tree such that the key stored at every node is less or equal to the keys of all its descendants. Similarly a max-heap is one in which the key at a node is greater or equal to all its descendants. For example the following is a min-heap

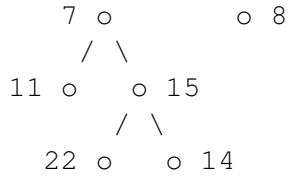


There are two important properties of a min-heap:

1. The minimum is always at the root.
2. The heap only maintains a partial order on the keys (unlike a BST that maintains the keys in a total order).

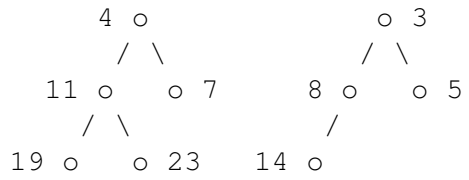
The first property allows us to access the minimum quickly, and the second will give us more flexibility that available in a BST.

Lets consider how to implement the three operations `deleteMin`, `insert` and `meld` on a heap. To implement `deleteMin` we can simply remove the root. This would leave:

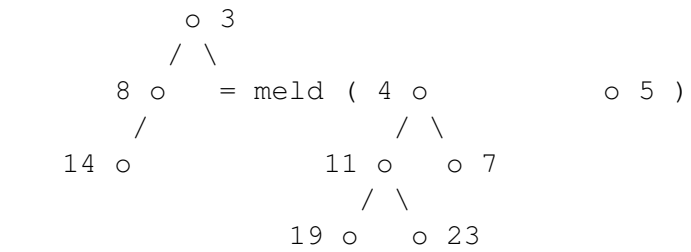


This is simply two heaps, which we can use `meld` to join. To implement `insert`(Q, v) we can just create a singleton node with the value v and then meld it with the heap for Q.

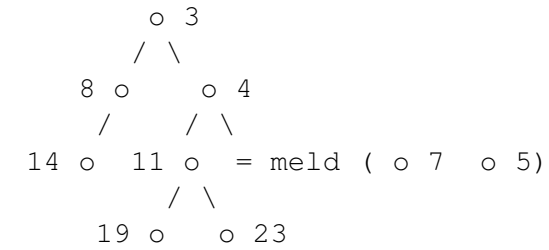
This basically means that the only operation we need to care about is the `meld` operation. Lets consider the following two heaps



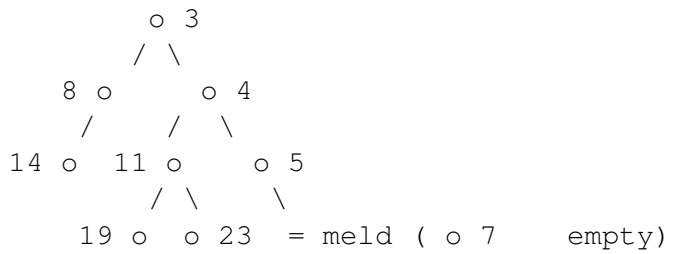
If we meld these two min-heaps, which value should be at the root? Well it has to be 3 since it is the minimum value. So what we can do is select the tree with the smaller root and then recursively meld the other tree with one of its children. In our case lets meld with the right child. So this would give us:



If we apply this again we get



and one more time gives:



Clearly if we are melding a heap A with an empty heap we can just use A. This leads to the following code

```
1  datatype PQ = Leaf | Node of (key, PQ, PQ)
2  fun meld(A, B) =
3    case (A, B) of
4      (_, Leaf) ⇒ A
5    | (Leaf, _) ⇒ B
6    | (Node(k_A, L_A, R_A), Node(k_B, L_B, R_B)) ⇒
7        case Key.compare(k_A, k_B) of
8          LESS ⇒ Node(k_A, L_A, meld(R_A, B))
9        | _ ⇒ Node(k_B, L_B, meld(A, R_B))
```

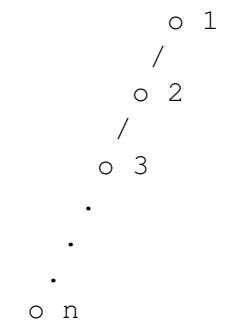
This code traverses the right spine of each tree (recall that the right spine of a binary tree is the path from the root to the rightmost node). The problem is that the tree could be very imbalanced and in general we can't put any useful bound on the length of these spines—in the worst case all nodes could be on the right spine. In this case the *meld* function could take $\Theta(|A| + |B|)$ work. It turns out there is a relatively easy fix to this problem. The idea is to keep the trees so that the trees are always deeper on the left than the right. In particular we define the rank of a tree as the length of the right spine. More formally:

```
1  rank(Leaf) = 0
2  rank(Node(_, _, R) = 1 + rank(R)
```

Now we require that for all nodes of a leftist heap (v, L, R)

$$rank(L) \geq rank(R)$$

This is why the tree is called leftist: the rank of all left branches are always at least as great as that of the right branch. Note that this definition allows the following unbalanced tree.



This is OK since we will only ever traverse the right spine of a tree, which in this case has length 1. We can now bound the rank.

Theorem 2.1. *For any leftist heap T $\texttt{rank}(T) \leq \log |T|$.*

This will be proven in next lecture. To make use of ranks we add a rank field to every node and make a small change to our code to maintain the leftist property:

```
1  datatype PQ = Leaf | Node of (int, key, PQ, PQ)

2  fun rank Leaf = 0
3    | rank (Node(r, _, _, _)) = r

4  fun makeLeftistNode (v, L, R) =
5    if (rank(L) < rank(R))
6    then Node(1 + rank(L), v, R, L)
7    else Node(1 + rank(R), v, L, R)

8  fun meld(A, b) =
9    case (A, B) of
10     (_, Leaf) ⇒ A
11   | (Leaf, _) ⇒ B
12   | (Node(_, k_A, L_A, R_A), Node(_, k_B, L_B, R_B)) ⇒
13       case Key.compare(k_A, k_B) of
14         LESS ⇒ makeLeftistNode(k_A, L_A, meld(R_A, B))
15       | _ ⇒ makeLeftistNode(k_B, L_B, meld(A, R_B))
```

Note that the only real difference is that we now use `makeLeftistNode` to create a node and it makes sure that the rank of the left child is at least as large as the rank of the right child by switching the two children if necessary. It also maintains the rank value on each node. We now have

Theorem 2.2. *If A and B are leftists heaps then the $\texttt{meld}(A, B)$ algorithm runs in $O(\log(|A|) + \log(|B|))$ work and returns a leftist heap containing the union of A and B.*

Proof. The code for `meld` only traverses the right spines of A and B and does constant work at each step. Therefore since both trees are leftist by Theorem 2.1 the work is bounded by $O(\log(|A|) + \log(|B|))$ work. To prove that the result is leftist we note that the only way to create a node in the code is with `makeLeftistNode`. This routine guarantees that the rank of the left branch is at least as great as the rank of the right branch. □