# Lecture 24 — More Leftist Heaps and Sorting Lower Bounds (DRAFT)

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

*Lectured by Kanat Tangwongsan — November 16, 2011*

**Today:**
- Finishing Leftist Heaps
- Lower Bounds for Sorting and Merging

## 1 Recap: Leftist Heaps

Let's take another look at leftist min heaps. Last time, we saw that leftist heaps give a priority-queue implementation with fast `meld` operation, which supports combining leftist heaps of sizes $n$ and $m$ in $O(\log n + \log m)$ work.

### 1.1 How the leftist heap got its name?

Invented by Clark Allen Crane in around 1971, a leftist heap is a binary tree—not a binary search tree—that satisifies the heap property and an additional property known as the leftist property. The heap property means that in a min heap, the value at any node is at least the values at the two children. Note that with heap property alone, we can identify the minimum value very quickly in $O(1)$ since the minimum value is at the root of the tree. But all update operations can take arbitrary long.

This is where the "leftist" idea comes in, with the goal of creating more structure and ensuring that all update operations we care about can be supported efficiently. The leftist property requires that for each node in the heap, the "rank" of the left child must be at least the "rank" of the right child. As we defined last time, the rank of a node $x$ is

$$\mathsf{rank}(x) = \#\text{ of nodes on the right spine of the subtree rooted at } x,$$

and the rank of a leaf is 0. That is, if $L(x)$ and $R(x)$ are the left and child children of $x$, then we have:

**Leftist Property:** For all node $x$ in a leftist heap, $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x))$

At an intuitive level, the leftist property implies that most of entries (mass) will pile up to the left, making the right spine of such a heap relatively short. We'll make this idea precise in the following lemma; following that, we'll see how we can take advantage of this fact to support fast meld operations.

**Lemma 1.1.** *In a leftist heap with $n$ entries, the rank of the root node is at most* $\log_2(n+1)$.

*Proof.* We'll first prove a claim that relates the number of nodes in a leftist heap to the rank of the heap.

**Claim:** If a heap has rank $r$, it contains at least $2^r - 1$ entries.

To prove this claim, let $n(r)$ denote the number of nodes in the smallest leftist heap with rank $r$. It is not hard to convince ourselves that $n(r)$ is a monotone function; that is, if $r' \geq r$, then $n(r') \geq n(r)$. With that, we'll establish a recurrence for $n(r)$. By definition, a rank-0 heap has 0 node. We can establish a recurrence for $n(r)$ as follows: Let's look at a the smallest-sized heap whose root node $x$ has rank $r$. First, the right child of $x$ must necessarily have rank $r - 1$—by the definition of rank. Moreover, by the leftist property, the rank of the left child of $x$ must be at least the rank of the right child of $x$, which in turn means that $\mathsf{rank}(L(x)) \geq \mathsf{rank}(R(x)) = r - 1$. Therefore, the size of the tree rooted $x$ is $n(r) = 1 + |L(x)| + |R(x)|$, so then

$$\begin{aligned} n(r) &\geq 1 + n(\mathsf{rank}(L(x))) + n(\mathsf{rank}(R(x))) \\ &\geq 1 + n(r-1) + n(r-1) = 1 + 2 \cdot n(r-1). \end{aligned}$$

Unfolding the recurrence, we get $n(r) \geq 2^r - 1$, which proves the claim.

To conclude the lemma, we'll simply apply the claim: Consider a leftist heap with $n$ nodes and suppose it has rank $r$. By the claim it must be the case that $n \geq n(r)$, because $n(r)$ is the fewest possible number of nodes in a heap with rank $r$. But then, by the claim above, we know that $n(r) \geq 2^r - 1$, so

$$n \geq n(r) \geq 2^r - 1 \implies 2^r \leq n + 1 \implies r \leq \log_2(n+1).$$

This concludes the proof that the rank of this heap is $r \leq \log_2(n+1)$. □

In words, this lemma says *leftist heaps have a short right spine,* about $\log n$ in length. To get good effiency, we should take advantage of it. Notice that unlike the binary search tree property, the heap property gives us a lot of freedom in working with left and right child of a node (in particular, they don't need to be ordered in any specific way). Since the right spine is short, our meld algorithm should, when possible, try to work down the right spine. With this rough idea, if the number of steps required to meld is proportional to the length of the right spine, we have an efficient algorithm that runs in about $O(\log n)$ work. Indeed, this is precisely what the algorithm we saw last time did: the meld algorithm below effectively traverses the right spines of the heaps $a$ and $b$. (Note how `meld` is called only with either $(R_a, b)$ or $(a, R_b)$.)

```
datatype PQ = Leaf | Node of (int, key, PQ, PQ)
fun rank Leaf = 0
  | rank (Node(r, _, _, _)) = r
fun makeLeftistNode (v, L, R) =
  if (rank(L) < rank(R))
  then Node(1 + rank(L), v, R, L)
  else Node(1 + rank(R), v, L, R)
fun meld(a, b) =
  case (a, b) of
      (_, Leaf) ⇒ a
    | (Leaf, _) ⇒ b
    | (Node(_, k_a, L_a, R_a), Node(_, k_b, L_b, R_b)) ⇒
        case Key.compare(k_a, k_b) of
            LESS ⇒ makeLeftistNode(k_a, L_a, meld(R_a, b))
          | _ ⇒ makeLeftistNode(k_b, L_b, meld(a, R_b))
```

Notice the use of the function *makeLeftistNode*: the role of it is to ensure that the resulting heap satisfies the leftist property assuming the two input heaps $L$ and $R$ did. The function can also be viewed as swapping the left and right children if the original ordering violates the leftist property.

**Performance of Meld:** As we observed already, the *meld*algorithm only traverses the right spines of the two heaps, advancing by one node in one of the heaps. Therefore, on input heaps $A$ and $B$, the process takes at most $\mathsf{rank}(A) + \mathsf{rank}(B)$ steps, which by the lemma we just proved, is at most $\log_2(|A| + 1) + \log_2(|B| + 1)$. Since these steps take constant work, we have the following theorem:

**Theorem 1.2.** *If $A$ and $B$ are leftists heaps then the meld$(A, B)$ algorithm runs in $O(\log(|A|)+\log(|B|)$ work and returns a leftist heap containing the union of $A$ and $B$.*

### 1.2 Summary of Priority Queues

Already, we have seen a handful of data structures that can be used to implement a priority queue. Let's look at the performance guarantees they offer.

| **Implementation** | *insert* | *findMin* | *deleteMin* | *meld* |
|---|---|---|---|---|
| (Unsorted) Sequence | $O(n)$ | $O(n)$ | $O(n)$ | $O(m + n)$ |
| Sorted Sequence | $O(n)$ | $O(1)$ | $O(n)$ | $O(m + n)$ |
| Balanced Trees (e.g. Treaps) | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(m \log(1 + \frac{n}{m}))$ |
| Leftist Heap | $O(\log n)$ | $O(1)$ | $O(\log n)$ | $O(\log m + \log n)$ |

Indeed, a big win for leftist heap is in the super fast *meld* operation—logarithmic as opposed to roughly linear in other data structures.

## 2 Lower Bounds

After spending time formulating a concrete problem, we might wonder how hard the problem actually is. In this course thus far, our focus has been on obtaining efficient algorithms for certain problems. For a problem $P$, we try to design efficient algorithms to solve it. The existence of an algorithm gives an upper bound on the complexity of the problem $P$. In particular, an algorithm $A$ with work (either expected or worst-case) $O(f(n))$ is a constructive proof that $P$ can be solved provided $O(f(n))$ work. This is essentially the upper bound part of the question.

In this lecture, we'll turn the tables, showing that certain problems cannot be solved more efficiently than a given bound. This is the lower bound part of the question. In general, this is a harder task: To establish a lower bound, we have to argue that *no algorithm, however smart, can possibly do better than what we claim*; it is no longer sufficient to exhibit an algorithm $A$ and analyze its performance.

### 2.1 Sorting and Merging Lower Bounds

Before we look at lower bounds for sorting and merging, let us review the (upper) bounds we have for various sorting algorithms we've covered:

| **Algorithm** | Work | Span |
|---|---|---|
| Quick Sort | $O(n \log n)$ | $O(\log^2 n)$ |
| Merge Sort | $O(n \log n)$ | $O(\log^2 n)$ |
| Heap Sort | $O(n \log n)$ | $O(n \log n)$ |
| Balanced BST Sort | $O(n \log n)$ | $O(\log^2 n)$ |

Notice that in this table, all algorithms have $O(n \log n)$ work—and except for heap sort, every algorithm is very parallel ($\log^2 n$ span). Can we sort in less than $O(n \log n)$ work? Probably. But we'll show that in the *comparison-based model*, we need $\Omega(n \log n)$ comparisons to sort $n$ entries. In the comparison-based model, we have no domain knowledge about the entries and the only operation we have to determine the relative order of a pair of entries $x$ and $y$ is a comparision operation, which returns whether $x < y$. More precisely, we'll prove the following theorem:

**Theorem 2.1.** *For a sequence $\langle x_1, \ldots, x_n \rangle$ of $n$ distinct entries, finding the permutation $\pi$ on $[n]$ such that $x_{\pi(1)} < x_{\pi(2)} < \cdots < x_{\pi(n)}$ requires, in the worst case, at least $\frac{n}{2} \log(\frac{n}{2})$ queries to the $<$ operator.*

Since each comparison takes at least constant work, this implies an $\Omega(n \log n)$ lower bound on the work required to sort a sequence of length $n$ in the comparison model.

What about merging? Can we merge sorted sequences faster than resorting them? As seen in previous lectures, we can actually merge two sorted sequences in $O(m \log(1 + n/m))$ work, where $m$ is the length of the shorter of the two sequences, and $n$ the length of the longer one. We'll show, however, that in the comparision-based model, we cannot hope to do better:

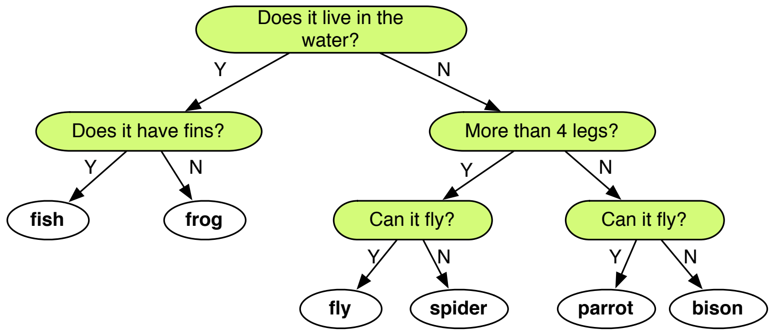
**Theorem 2.2.** *Merging two sorted sequences of lengths $m$ and $n$ ($m \leq n$) requires at least*

$$m \log_2(1 + \tfrac{n}{m})$$

*comparison queries in the worst case.*

## 2.2 Decision Trees or The 20 Questions Game

Let's play game. Suppose I think of an animal but you know for fact it's one of the following: a fish, a frog, a fly, a spider, a parrot, or a bison. You want to find out what animal that is by answering the fewest number of Yes/No questions (you're only allowed to ask Yes/No questions). What strategy would you use? Perhaps, you might try the following reasoning process:



Interestingly, this strategy is optimal: there is no way you could have asked any 2 Yes/No questions to tell apart the 6 possible answers. If we can ask only 2 questions, any strategy that is determistic and computes the output using only the answers to these Yes/No questions can distinguish between only $2^2 = 4$ possibilities. Thus, using 3 questions is the best one can do.

Determining the minimum number of questions necessary in the worst case in at the crux of many lower-bound arguments. For starters, we describe a way to represent a deterministic strategy for playing such a game in the definition below.

**Definition 2.3** (Binary Decision Trees)**.** A *decision tree* is a tree in which

- each leaf node is an answer (i.e. what the algorithm outputs);
- each internal node represents a query—some question about the input instance—and has $k$ children, corresponding to one of the $k$ possible responses $\{0, \ldots, k-1\}$;
- and the answer is computed as follows: we start from the root and follow a path down to a leaf where at each node, we choose which child to follow based on the query response.

The cruical observation is the following: if we're allowed to make at most $q$ queries (i.e., ask at most $q$) questions, the number of possible answers we can distinguish is the number of leaves in a binary tree with depth at most $q$; this is at most $2^q$. Taking logs on both sides, we have

$$\boxed{\text{If there are } N \text{ possible outcomes, the number of questions needed is at least } \log_2 N.}$$

## 2.3 Warm-up: Guess a Number

As a warm-up question, if you pick a number $a$ between 1 and $2^{20}$, how many Yes/No questions do I need to ask before I can zero in on $a$? By the calculation above, since there are $N = 2^{20}$ possible outcomes, I will need *at least*

$$\log_2 N = 20$$

questions in the worst case.

## 2.4 A Sorting Lower Bound

Let's turn back to the classical sorting problem. We will prove Theorem 2.1. This follows almost immediately from our observation about $k$-ary decision trees. There are $n!$ possible permutations, and to narrow it down to one permutation which orders this sequence correctly, we'll need $\log(n!)$ queries, so the number of comparison queries is at least

$$\begin{aligned}
\log(n!) &= \log n + \log(n-1) + \ldots \log 1 \\
&\geq \log n + \log(n-1) + \cdots + \log(n/2) \\
&\geq \tfrac{n}{2} \cdot \log(n/2).
\end{aligned}$$

We can further improve the constants by applying Stirling's formula instead of this crude approximation. Remember that *Stirling's formula* gives the following approximation:

$$n! = \left(\frac{n}{e}\right)^n \sqrt{2\pi n}\left(1 + \Theta(n^{-1})\right) > \left(\frac{n}{e}\right)^n,$$

so $\log_2(n!) > n\log_2(n/e)$.

### 2.5 A Merging Lower Bound

Closely related to the sorting problem is the merging problem: given two sorted sequences $A$ and $B$, the merging problem is to combine these sequences into a sorted one. To apply the argument we used for sorting, we'll need to count how many possible outcomes the comparsion operation can produce.

Suppose $n = |A|$, $m = |B|$, and $m \le n$. We'll also assume that these sequences are made up of unique elements. Now observe that we have not made any comparision between elements of $A$ and $B$. This means any interleaving sequence $A$'s and $B$'s elements is possible. Therefore, the number of possible merged outcomes is the number of ways to choose $n$ positions out from $n + m$ positions to put $A$'s elements; this is simply $\binom{n+m}{n}$. Hence, we'll need, in the worst case, at least $\log_2\binom{n+m}{n}$ comparison queries to merge these sequences.

The following lemma gives a simple lower bound for $\binom{n}{r}$, so that we can simplify $\log_2\binom{n+m}{n}$ to an expression that we recognize.

**Lemma 2.4** (Binomial Lower Bound)**.**

$$\binom{n}{r} \ge \left(\frac{n}{r}\right)^r.$$

*Proof.* First, we recall that

$$\binom{n}{r} = \frac{n!}{r!(n-r)!} = \frac{n(n-1)(n-2)\ldots(n-r+1)}{r(r-1)(r-2)\ldots 1} = \prod_{i=0}^{r-1}\frac{n-i}{r-i}.$$

We'll argue that for $0 \le i < \min(r, n)$, $\frac{n-i}{r-i} \ge \frac{n}{r}$. Notice that

$$\frac{n-i}{r-i} \ge \frac{n}{r} \iff r(n-i) \ge n(r-i) \iff rn - ri \ge nr - ni \iff (n-r)i \ge 0.$$

Therefore, we have $\binom{n}{r} \ge \prod_{i=0}^{r-1}\frac{n-i}{r-i} \ge (n/r)^r$. □

With this lemma, we conclude that the number of comparison queries needed to merge sequences of lengths $m$ and $n$ ($m \le n$) is at least

$$\log_2\binom{n+m}{m} \ge m\log_2\left(1 + \tfrac{n}{m}\right),$$

proving Theorem 2.2