1 Overview

We start a new topic, amortized analysis. In this lecture, we will explain what amortization is and apply it to the analysis of two data structures: Binomial Heaps and Fibonacci Heaps.

2 Amortization

2.1 Definition

Suppose our data structure supports operations A, B, C. We say that the "amortized costs" of these operations are $t_a, t_b, t_c$ if any sequence of $n_a$ A operations, $n_b$ B operations, and $n_c$ C operations takes time at most $n_a t_a + n_b t_b + n_c t_c$.

2.1.1 Potential functions

A common way to prove amortized bounds is via the potential function method:

- Define potential $\Phi : \{\text{state of data structure}\} \rightarrow \mathbb{R}_{\geq 0}$
- $\Phi(\text{empty structure}) > 0$
- We perform $k$ operation with actual times $t_1, \ldots, t_k$ and that the states are $S_0, S_1, \ldots, S_k$. The amortized cost of an operation $i$ is defined to be:
  
  $$t_i + \Phi(S_i) - \Phi(S_{i-1})$$

- So the total amortized cost is bounded by
  
  $$\sum_i (t_i + \Delta \Phi(i)) = \sum_i t_i + \Phi(S_k) \geq \sum_i t_i$$

Therefore, the common way to compute an amortized cost is to compute the worst $t_i + \Phi(S_i) - \Phi(S_{i-1})$ at any step.

2.2 Example: Heaps

Heaps are a data structure $H$ which support:

- $\text{insert}(x)$ (the values $x$ are comparable)
• deleteMin(): returns the min \( x \in H \) then deletes it.
• decreaseKey(p, k): replaces key at item \( p \) with \( k \) which must be at most \( p \)’s current key.

Dijkstra’s algorithm, which uses heaps, has a runtime \( O(n(t_{\text{insert}} + t_{\text{deleteMin}}) + m t_{\text{decreaseKey}}) \).

The existing implementations of heaps are the following:

• Binary Heaps, by Williams [1], which achieve \( O(\lg n) \) worst case for \( t_{\text{insert}} \), \( t_{\text{deleteMin}} \), and \( t_{\text{decreaseKey}} \).
• Binomial Heaps, by Vuillemin [2], which achieve \( O(1) \) amortized cost for \( t_{\text{insert}} \) and \( O(\lg n) \) worst-case for \( t_{\text{deleteMin}} \) and \( t_{\text{decreaseKey}} \).
• Fibonacci Heaps, by Fredman and Tarjan [3], which achieve \( O(1) \) amortized cost for \( t_{\text{insert}} \) and \( t_{\text{decreaseKey}} \) and \( O(\lg n) \) amortized cost for \( t_{\text{deleteMin}} \).

In this lecture, we will not cover binary heaps because they are not relevant to the Fibonacci Heaps. We will cover Binomial Heaps, since Fibonacci Heaps are a more optimal version of Binomial Heaps. Related work also include:

• Kaplan, Tarjan and Zwickl [4] constructed heaps so that there is only one tree and not a forest.
• Brodal [5] obtained the same bounds as Fibonacci Heaps but in worst case instead of in amortized cost. His solution has been described as complicated, and uses extendable arrays.
• Brodal, Lagogiannis, Tarjan [6] obtained the same bounds in worst case in the pointer machine model.

We note that we cannot improve the bound of \( O(\lg n) \) for \( t_{\text{deleteMin}} \) in the comparison model while keeping other operations constant time, since otherwise we would be able to sort in \( o(n \lg n) \) time, which is impossible.

3 Binomial Heaps

3.1 Definition

In binomial heaps, each item is a node. We maintain these nodes in a forest as follow:

• If \( x \) is the parent of \( y \), then \( \text{key}(x) \leq \text{key}(y) \).
• For a tree in our heap, its ”rank” is the degree of its root.
• A tree with rank \( k \) has \( 2^k \) nodes in it
• For each \( k \), we will have at most 1 tree of rank \( k \).
• Roots of trees of rank \( k \) have \( k \) subtrees, these are trees of ranks \( 0, 1, \ldots, k - 1 \).

The roots of the trees are kept in a linked list, as shown in Figure 1.
3.2 Operations on Binomial Heaps

- decreaseKey: Change the key, then keep swapping upward until we satisfy the property of parents having lower keys than children.

- insert(x): Add a new singleton tree to forest, then repeatedly merge trees of equal rank until we have trees that are all of different ranks. This can be seen as a binary addition: for example, if we have trees of rank 4, 2, and 1 and we had a singleton, then $1011 + 0001 = 1100$. So we then have trees of rank 4 and 3.

- deleteMin: Remove root of some tree, insert all its children as roots in the main forest. Now merge trees of equal rank the same way as we did in insert(x).

3.3 Worst Case Analysis

In the worst case, all operations take $\Theta(\lg n)$ time. In the case of decreaseKey, we could decrease a leaf in the tree of highest rank and have to swap until the root. Since the biggest tree has height $\Theta(\lg n)$, this would take $\Theta(\lg n)$ time in the worst case. In the case of insert and deleteMin, we could have trees of rank 1, 2, $\ldots$, $m$. We would then have to repeatedly merge $\Theta(\lg n)$ times to get a tree of rank $m + 1$.

3.4 Amortized analysis

We now study the amortized cost and let the potential function be $\Phi(\text{data structure state}) = \text{number of trees}$. Let $T$ be the number of trees before an insertion and $t$ be the number of trees after an insertion. The actual costs of insertions are $O(T - t + 1)$. Therefore, the amortized cost of insertion is: actual cost + $\Delta\Phi = T - t + 1 + t - T = 1$
4 Fibonacci Heaps

4.1 The structure

The main idea in Fibonacci Heaps is to be lazy. We do not want to do work until we really need to. In the case of Binomial Heaps, there is no reason for insertions to spend time consolidating the trees to maintain the Heap structure, deleteMin can do that. So when we insert an element, we just create a singleton and add it to the structure without doing any merges.

The problem could now come from decreaseKey, in the case where we repeatedly decrease keys that are leaves of a large tree. A really lazy possible decreaseKey is to decrease $x$’s key, and if $x$ is now smaller than parent, then cut $x$’s tree out and place as a top level tree. The problem with this approach is that we could have a tree of rank $k$, with many decreaseKey on the nodes that are neither the root nor a child of root, which would give us a tree of rank $k$ with $k + 1$ leaves.

The fix is as follows. If node $p$ loses one child $x$ due to decreaseKey, then no problem. If $p$ loses a second child, we cut $p$ out of its tree and make $p$’s tree a top level tree. With this fix in hand, it is possible to prove Lemma 2 below.

**Definition 1.** We define the Fibonacci numbers $\{F_k\}_{i=0}^\infty$ via the recurrence

$$F_k = \begin{cases} 0, & \text{if } k = 0 \\ 1, & \text{if } k = 1 \\ F_{k-1} + F_{k-2}, & \text{otherwise} \end{cases}$$

**Lemma 2.** A tree in a Fibonacci heap with rank $k$ has at least $F_{k+2}$ nodes.

4.2 Amortized Analysis

We claim that $t_{\text{insertion}}, t_{\text{decreaseKey}} = O(1)$ and that $t_{\text{decreaseMin}} = O(\log n)$.

**Proof.** Define $\text{mark}(x) = \begin{cases} 1 & \text{if } x \text{ has lost child} \\ 0 & \text{else} \end{cases}$

and the potential function to be $\Phi(\text{state}) = \# \text{ trees} + 2(\# \text{ marked items}) = T(H) + 2M(H)$

- Insertion. actual cost + $\Delta \Phi = O(1) + 1 = O(1)$
- deleteMin. $T + \Delta T(H) + 2\Delta M(H) \leq T + t - T + 0 = O(t)$ where $t$ is the number of trees left after consolidation, and $T$ is before consolidation. That is, we keep merging trees of equal rank until for each rank $k$, at most one tree of rank $k$ exists in our data structure. Now note that a rank $k$ tree has at least $F_{k+2} \geq \sqrt{2}^k$ nodes. Therefore the largest rank of any tree in our data structure can be at most $\log \sqrt{2} n$ (since there are only $n$ nodes in the entire forest), implying $t \leq (\log \sqrt{2} n) + 1 = O(\log n)$.
- decreaseKey. We have two cases.
– Case 1: $x$ does not get cut out.
  \[
  \text{actual cost} + \Delta \Phi = O(1) + 0 = O(1).
  \]
– Case 2: We do some number $c > 0$ of cascaded cuts ($c$ levels of consecutive cuts).
  \[
  \text{actual cost} + \Delta T(H) + 2\Delta M(H) \leq c + c + (-2c + 2) = O(1)
  \]

Next time, we will study Splay trees, which has amortized cost $O(\lg n)$ for all operations. The benefit of Splay trees is that they have very nice properties, such as static optimality. Static optimality takes into account the frequency of searches of items in order to minimize the total cost of all operations.

References


