CS 224: Advanced Algorithms Spring 2017

Lecture 6 — February 14, 2017

Prof. Jelani Nelson *Scribe: David Stoner*

1 Binomial Heap Review

Recall that Binomial heaps are forests (groups of trees) which maintain the following conditions:

- The roots are connected in a doubly linked list.
- At most one tree has rank k for each nonnegative integer k, where a tree's rank is the number of children of its root.
- The children of the root of a rank k tree are roots of rank j trees for $j = 0, 1, \ldots, k-1$. At a consequence, a rank k tree has 2^k nodes.

Additionally, values always are nonincreasing from parent to child.

Insert() in binomial heaps works like binary addition, taking time $O(\log n)$ per insert since at most $\log n$ "carries" are necessary.

Deletemin() only requires the roots to be checked; once a root is removed, the newly added trees are consolidated if necessary. This also takes $O(\log n)$ time.

2 Fibonacci Heaps

One source of inefficiency for Binary heaps stems from the fact that Insert() consolidates the heaps constantly, despite this consolidation only being necessary for Deletemin(). Fibonacci heaps attempt to resolve this issue by making Insert() very lazy, with DeleteMin() doing the consolidation work. We still wish to maintain the same basic structural properties as the binomial heaps. That is, wish for rank k trees to have some number of nodes exponential in k, and for the children of the root to be j trees for $j \in \{0, \ldots, k-1\}$. Also, there should be at most one rank k tree at all times; we end up relaxing this second condition a bit. The Insert() and Deletemin() functions work as follows:

- $Insert\,(x)$: Put x by itself as a rank 0 tree in the top level structure.
- $DeleteMin\,()$: Search the roots for the min, and delete it as with binomial heaps. Afterwards, consolidate all of the trees as with Binomial heaps so that there is at most 1 tree of each rank.

One idea for the $decKey\,()$ function would be to cut the queried element out from its tree. However, this is undesirable since we may end up with trees of small size and large rank. Instead, we come up with a coloring scheme which prevents trees with large rank from becoming too sparse.

$Deckey(x)$: each node stores a mark bit $mark(x)$ which is equal to 1 if x has lost a child to a previous Deckey and 0 otherwise. In order to $Deckey(x)$, we check to see if its parent is marked. If not, then we cut the subtree at x out as usual and mark y. If y is marked, we instead cut out y's subtree and unmark y. This is removing a child from y's parent, so the process will continue to recurse until an unmarked node is reached.

One can show that a rank k tree has at least $F_k \geq c\sqrt{2}^k$ nodes. With this in mind, we analyze the amortized times for Insert, Deletemin, and Deckey via a potential function. Let $\tilde{t} = t_{act} + \Delta\Phi$, with

$$\Phi(H) = T(H) + 2M(H)$$

where H is the current heap, $T(H)$ is the current number of trees, and $M(H)$ is the current number of marked nodes. An insertion takes constant time and increases $\Phi(H)$ by 1, so $\tilde{t}_{Insert} = O(1)$. For $\tilde{t}_{Deletemin}$, let H' denote the heap after the deletion and consolidation. We have:

$$\begin{aligned}\tilde{t}_{Deletemin} &= t_{actual} + \Delta\Phi \\ &= O(\log n) + T(H) + [T(H') - T(H)] \\ &= O(\log n)\end{aligned}$$

Here $T(H') = O(\log n)$ since the heap H' is after consolidation. Finally, suppose c is the number of cascaded cuts required for a Deckey operation. Then:

$$\begin{aligned}\tilde{t}_{Deckey} &= t_{actual} + \Delta\Phi \\ &= c + \Delta T + 2\Delta M \\ &= c + c + 2(1 - c) = O(1)\end{aligned}$$

Splay Trees

Splay trees are self-adjusting data structures introduced by [1]. They support $O(\log n)$ time per insertion, find, and delete functions. Below are some of the milestones achieved by splay trees:

Static Optimality: Consider the static optimality problem, letting $C_T(\sigma)$ be the number of accesses in the tree T. Then:

$$C_{splay}(\sigma) = O\left(n \log n + \min_T C_T(\sigma)\right)$$

Working Set: Let t_j be the number of operations since the last query. Splat trees achieve:

$$C_{splay}(\sigma) = O\left(\sum_{j=1}^{m} \log(t_j + 1) + n \log n\right)$$

Static Finger: For the static finger problem, one of the elements, the finger, is taken to be fixed. The $n \log n$ in both the static and dynamic finger problems accounts for the drop in the potential

function. Here, splay trees get the bound:

$$C_{splay}(\sigma) = O\left(\min_{f}\sum_{j=1}^{m}\log\left(|f - \text{rank order of jth key}| + 2\right) + n\log n\right)$$

Dynamic Finger: In the dynamic finger problem, there i a sequence $\{x_i\}$ of accessed elements, and the finger for each step is the element accessed in the previous step. For this, splay trees ahcieve:

$$C_{splay}(\sigma) = O\left(n\log n + \sum_{j=1}^{m}\left(|x_j - x_{j-1}| + 2\right)\right)$$

Dynamic Optimality Conjecture: There exists a constant c such that $\forall\sigma, C_{splay}(\sigma) = c{*}OPT$, where OPT is the least time achieved by any algorithm which accesses elements by following the path from the root, and which may make rotations for unit cost. Any balanced BST achieves this for $c = O(\log n)$. This is due to the fact that any balanced BST requires $O(\log n)$ per operation. Tango trees showed $C = O(\log\log n)$ [2]

3 Splay Tree Details

Splay Trees make use of the BST rotate() function, which reverses the direction of some parent-child relation, making a vertex the parent of its original parent. The functions of splay trees work as follows:

$insert(x)$: Follow the root to leaf path to find where x should go, and place it there. Then $splay(x)$.

$find(x)$: Follow pointers down to x, then $splay(x)$.

$delete(x)$: First $splay(x)$, then $splay(z)$ where z is the node inheriting x's position when x is deleted.

$splay(x)$: Bring x to the root via a series of rotations. Rotating x over and over again will accomplish this, but it will not balance the tree (consider splaying the lowest vertex in a binary tree which is a path). Instead, $splay(x)$ takes three cases:

Let y be the parent of x, and z the parent of y.

- Case 1: z is null. Then $rotate(x)$.
- Case 2: $x = y.left, y = z.left$ or $x = y.right, y = z.right$. Then rotate y first, then rotate x.
- Case 3: All else. $rotate(x)$ twice.

When splay is defined in this way, $splay(x)$ balances the tree as it sends x to the root. Assign each item weight w, and define $S(x)$ to be the sum over y in x's subtree of $w(y)$. The potential function

we will use to analyze the splay trees is of the form:

$$\Phi(T) = \sum_{x \in T} \log(S(x))$$

References

[1] Sleator, Daniel Dominic and Tarjan, Robert Endre. Self-Adjusting Binary Search Trees. *JACM*, 1985.

[2] Demaine, Erik D. et al. Dynamic Optimality - Almost. *FOCS*, 2004.