Lecture 3, Sep 18, 2023

Binomial Heaps

- What if we had 2 priority queues, and we wanted to merge them in an efficient way?
  - Using a binary heap, this would take $n \log n$ time; can we find a way to do this in $\log n$ time?
- We will present a new data structure, a binomial heap, which can do all operations in $O(\log n)$ time (at the cost of having $O(\log n)$ time to view the max element, as opposed to constant time)

Definition

A *binomial tree* is recursively defined as follows:

1. A binomial tree of height 0, B_0 consists of a single node
2. A binomial tree of height k, B_k consists of two binomial trees B_{k-1}, where one becomes the parent of the other

Alternatively, the root B_k has k children, with the first child being a binomial tree B_0, the second being B_1, etc. until the k-th child B_{k-1}.

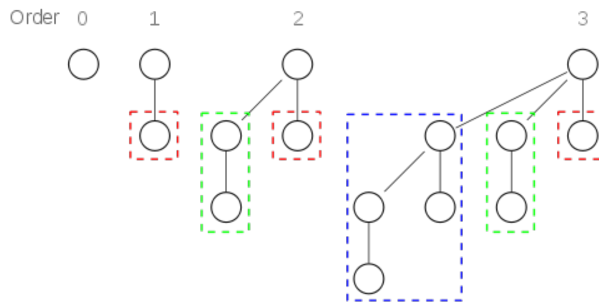

Figure 1: Example binomial trees.

- A binomial tree B_k has:
  - Height k
  - 2^k nodes
  - $\binom{k}{d}$ nodes at depth d (hence why it's called a binomial tree)

Definition

A *binomial forest* of size n is denoted F_n, which contains a sequence of B_k trees with strictly decreasing ks and n nodes in total.

- Example: if we want a forest F_7, we can decompose it into 3 binomial trees with 4, 2, and 1 nodes respectively, so the forest contains B_2, B_1, B_0
  - This is equivalent to expressing n in binary
  - e.g. $9 = 1001_2 = 2^3 + 2^0$ so F_9 contains B_3, B_0
  - Let $\alpha(n)$ be the number of 1s in the binary representation of n, then F_n has $\alpha(n)$ trees
- F_n has $n - \alpha(n)$ edges (since each node comes with 1 edge, except for roots, and there are $\alpha(n)$ trees)

Definition

A *min-binomial heap* of n elements is a binomial forest F_n such that each binomial tree B_k within the forest satisfies the min-heap property, i.e. a parent is smaller than both its children.

- A binomial heap of N elements can be built in $O(n)$ operations from scratch
  - One key comparison is needed per binomial heap edge
- But how do we actually put this in memory?
  - Different nodes have different number of children, so using pointers directly will be inefficient
  - For every node, we will have 3 pointers: one to its parent, one to its leftmost child, and one to its next sibling
    * To access the children of a node, we access the leftmost child first, and then go through the chain of sibling pointers
    * The sibling pointers of root nodes connect between different trees, pointing to the next bigger B_k tree
  - Finally, we need to keep track of the head of the data structure, which is a pointer to the smallest B_k tree
  - All the pointers make the binomial heap much less efficient in memory than binary heaps
- Two important properties to note:
  - We can merge two min-heap ordered B_k trees into a single min-heap ordered B_{k+1} tree with just a single key comparison, by making the tree with the smaller root the parent tree
  - Deleting the root of a min-heap ordered B_k tree gives a min binomial heap, since the children are binomial trees $B_0, B_1, \ldots, B_{k-1}$
- We can now implement the $\textsc{Union}(T, Q)$ operation, which merges two heaps
  - The operation works like binary addition:
    * Start with the B_0 trees in both heaps
    * If there is a B_k in one heap but not the other heap, simply add this tree to the result
    * If there are B_k trees in both heaps, merge them into a B_{k+1} tree and make it a "carry"; merge this carry with the existing B_{k+1} trees of the two heaps
  - Since each B_k tree merge takes constant time and both input heaps have at most $\log n$ trees, the complexity is $O(\log n)$
- An insert operation is equivalent to merging with a heap of size 1
- For $\textsc{Min}(T)$, we have to scan the root of every B_k tree; this gives us $O(\log n)$ complexity
- For $\textsc{ExtractMin}(T)$, find the B_k root that contains the minimum and remove it; this gives us another binomial heap made of its children, so we can simply merge it into the original heap