CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 7 — February 7, 2023

Prof. Jelani Nelson *Scribe: Chris Liu, Serena Zhang*

1 Overview

Next many (5) lectures: Data Structures
Today: Heaps (Priority Queues)
Store database of key-value pairs subject to:

1. *insert(k, v)*: adds key-value pair (k, v) to database of items
2. *decKey(*v, k)*: reduces old key of v to new key k, do nothing if k is bigger than old key
3. *delMin()*: return item with smallest key, delete it from database

2 Runtime of Algorithms that use Heaps

Assume there are n vertices and m edges in the graph.
Dijkstra : $n \cdot t_I + n \cdot t_{DM} + m \cdot t_{DK}$
Prim : $n \cdot t_I + n \cdot t_{DM} + m \cdot t_{DK}$

DS Name	t_I	t_{DM}	t_{DK}
Binary	$\log n$	$\log n$	$\log n$
Binomial	1^*	$\log n$	$\log n$
Fibonacci	1^*	$\log n^*$	1^*

Note: n is the size of the heap. * means amortized.
More Recent:

- Brodal '96 matched Fibonacci bounds in all worst case without amortization. Caveat is it's not a "pointer-machine," meaning it doesn't store items in a tree/graph.
- Brodal, Lagogianis, Tarjan STOC '12 matched Fibonacci bounds in all worst case and using "pointer-machine."

3 Analysis based on Potential Function

Recall that potential function $\Phi : \{state\} \to \mathbb{R}$. We define Φ-cost = true cost + $\Delta\Phi$. We assume that $\Phi \geq 0$ and Φ of an empty heap is 0. Total Φ-cost = total cost + $\Phi_{final} - \Phi_{initial}$ = total cost + $\Phi_{final} \geq$ total cost. We conclude that Φ-cost of an operation is a valid upper bound on amortized cost of that operation.

4 Binomial Heaps

- Forest of trees
- Each tree is heap-ordered, $key(x) \geq key(parent(x))$
- Define rank of a tree is degree of root
- Invariant: at most one tree of each rank

Note that largest rank $\leq \log n$ in Binomial Heaps. And a rank k tree will have rank j trees hanging off the root for $j = 0, 1, ..., k-1$. A rank k tree has 2^k items.

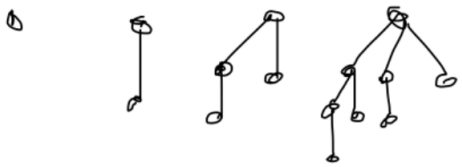
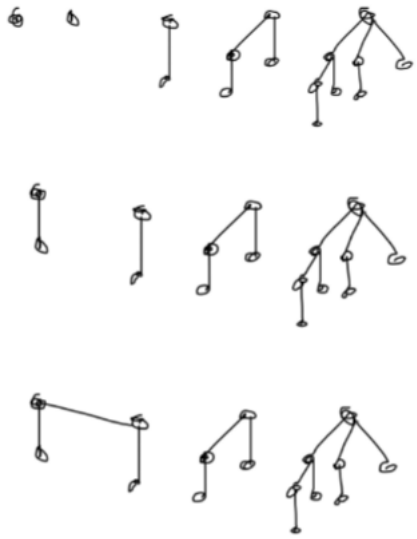


Figure 1: Binomial Heap: forest of trees

Consider adding an item to the binary heap shown above. When adding a new node to the heap, we create a rank 0 tree but there is already a rank 0 tree so we merge them such that the heap order is preserved. Now there are two rank 1 trees so we merger again. Repeat the process above and we will eventually merge two rank 3 trees into one rank 4 tree. Now if we add a new item, there would be a rank 0 tree and a rank 4 tree.



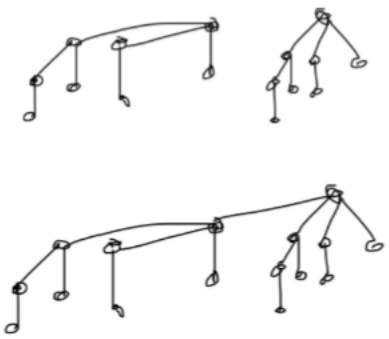


Figure 2: Add an item to Binary Heap

We can think of this process as binary addition. We represent the heap as a binary number, each digit represents a rank (rightmost digit represents rank 0 and so on, for example, 1101 means there is a rank 0 tree, a rank 2 tree, and a rank 3 tree in the heap). Adding a new node to binary heap is like adding 1 to the binary number that represents the heap.

4.1 Implementation

- *insert*: add the new node as a rank 0 tree, then keep merging
- *decKey*: decrement key, if it's smaller than its parent, keep bubbling it up until the heap property is not violated
- *delMin*: query every root to get the min node, return and delete the min node. Now we have k trees, iteratively merge them.

Note that merging k trees can again be thought of as binary addition. Instead of adding 1 to a binary number, we are now adding an arbitrary binary number of the binary number that represents the heap.

4.2 Analysis

We define potential function $\Phi(H) :=$ number of trees in H, denote as $T(H)$.

- *insert*: $\underbrace{\text{actual cost}}_{1+(T-t)} + \underbrace{\Delta\Phi}_{t-T} = \mathcal{O}(1)$
  T is the number of trees before operation, t is the number of trees after operation. $T-t$ tells us how many "carry bits" there were.
- *decKey*: $\underbrace{\text{actual cost}}_{\log n} + \underbrace{\Delta\Phi}_{0} = \mathcal{O}(\log n)$
  $\mathcal{O}(\log n)$ to bubble up the decremented node, and no trees were created or destroyed.
- *delMin*: $\underbrace{\text{actual cost}}_{\log n} + \underbrace{\Delta\Phi}_{\log n} = \mathcal{O}(\log n)$
  $\mathcal{O}(\log n)$ to scan through all roots since there are at most $\mathcal{O}(\log n)$ different ranks, $\mathcal{O}(\log n)$ to merge the trees.

5 Fibonacci Heaps

5.1 Implementation

- *insert*: add a new rank 0 tree
- *decKey*: if decreasing the key of x violates the heap property, cut the edge between x and its parent and make x the root of a new tree
- *delMin*: scan through roots of all trees and find the min root, return and delete from tree, then consolidate by merging trees with equal ranks

The problem with *decKey* is that number of items in rank k trees is too small (not 2^k). So we fix it by doing the following : the first time a node x loses a child, no problem, but the second time it loses a child, cut x off from its parent

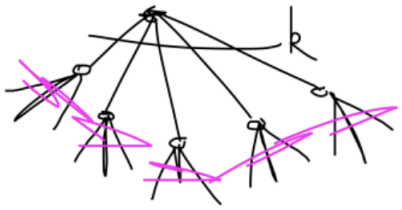


Figure 3: Rank k Tree with bad *decKey*

If we use the *decKey* function defined above, we would keep cutting subtrees and leave the rank k tree with very few nodes. So we construct a better *decKey*:

- each node maintains a "mark" bit
- mark = 1 if x has lost one child
- when x loses 2 children, make x its own tree, then set mark = 0

5.2 Minimum Possible Size of a Tree as a Function of its Rank

Claim 5.1. rank k tree has size $\geq F_{k+2}$ (F_{k+2} is the $(k+2)$th Fibonacci number)

Proof. For the rank 0 and rank 1 tree, the size is at least $1 = F_2$ and $2 = F_3$ respectively. For a tree of rank k where $k > 1$, suppose the children of the root x are $y_0, \cdots y_{k-1}$ (sorted in increasing order by when they became children of x). For $i > 2$, at the point in time that i became a child of x, x had rank at least i, so y_i must have as well (we only make a node a parent of another if they are both roots of equal rank). Since then y_i must have lost at most 1 child, and thus has rank at least $i-1$. Thus if S_i is the minimum size of a subtree whose root has i children, then $S_k \geq 2 + \sum_{i=0}^{k-2} S_i$ (the "2+" comes from the root, and the child y_0). The subtree sizes are minimized when all $\geq$ are equalities, in which we get $S_k - S_{k-1} = S_{k-2}$, or rearranging, $S_k = S_{k-1} + S_{k-2}$, which is the Fibonacci sequence recurrence. □

Note : $F_k = F_{k+1} + F_{k+2} \geq 2F_{k-2} \geq 2^{\frac{k}{2}} = \sqrt{2}^k$, so F_k grows exponentially.

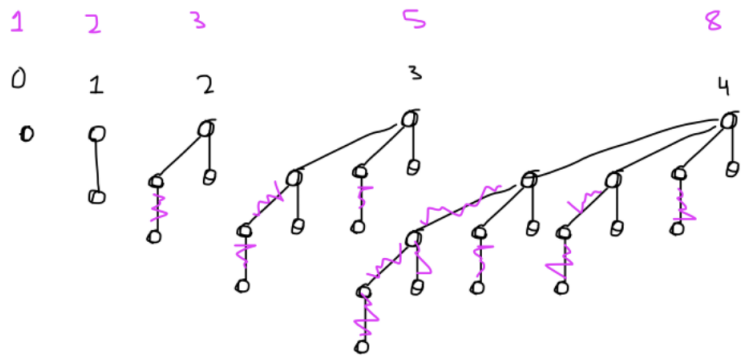


Figure 4: Minimum Possible Size of a Tree

Black numbers are ranks, pink numbers are minimum possible number of nodes in the tree. Edges crossed off are the ones that can be cut off without changing the rank of the trees.

5.3 Analysis

Define $\Phi(H) := T(H) + 2 \cdot M(H)$, where $M(H)$ is the number of marked items.

- *insert*: $\mathcal{O}(1) + \underbrace{\Delta T(H)}_{1} + 2\underbrace{\Delta M(H)}_{0} = \mathcal{O}(1)$
- *decKey*: Say there are C cascading cuts. $\mathcal{O}(1) + C + \underbrace{\Delta T(H)}_{C} + 2\underbrace{\Delta M(H)}_{-(C-1)+1} = \mathcal{O}(1)$

  Since when after C cuts, C new trees were created, $C-1$ nodes got unmarked, node at the "top" got marked.
- *delMin*: $T(H) + \underbrace{\Delta T(H)}_{\mathcal{O}(\log n) - T(H)} + 2\underbrace{\Delta M(H)}_{0} = \mathcal{O}(\log n)$

  Actual cost is $T(H)$ to scan through all roots. Max number of new trees you can create is $\log n$ since the max rank is $\log n$. Change in the number of marked nodes is zero because *delMin* doesn't mark or unmark any nodes.

References

[1] Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.