LECTURE 4- QUAKE HEAPS & SCAPEGOAT TREES

CS-GY 6043 Design and Analysis of Algorithms II

Scribe Notes by Parikshit Solunke

Contents

1 Quake Heaps

An Amortized Data Structure that is an alternative to Fibonacci Heaps. A collection of tournament trees with distinct heights (distinct heights are only maintained after extract-min, in other situations multiple trees in the heap are allowed to have the same height)[1].

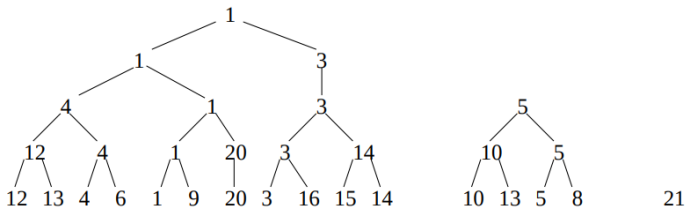


Figure 1: A quake heap with 3 tournament trees of distinct heights

1.1 Tournament Trees

1.1.1 Structure

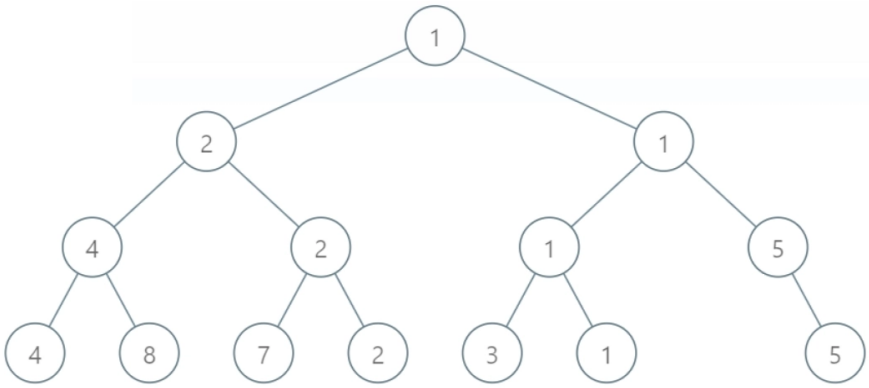


Figure 2: Tournament Tree (Min)

- "Smallest node wins" structure.
- All data stored in leaves at the lowest level, edges cannot skip levels
- Degree of a node can be either 1 or 2
- Every node knows its topmost position
- Smallest element essentially lives in a complete path from leaf to the root

1.1.2 Tournament Tree Operations

- **Link:** Assumes the two trees have the same height h. Simply join the two roots, and duplicate the smaller element to make it the root of the new tree. Constant time operation, the new tree will thus have height $h + 1$.

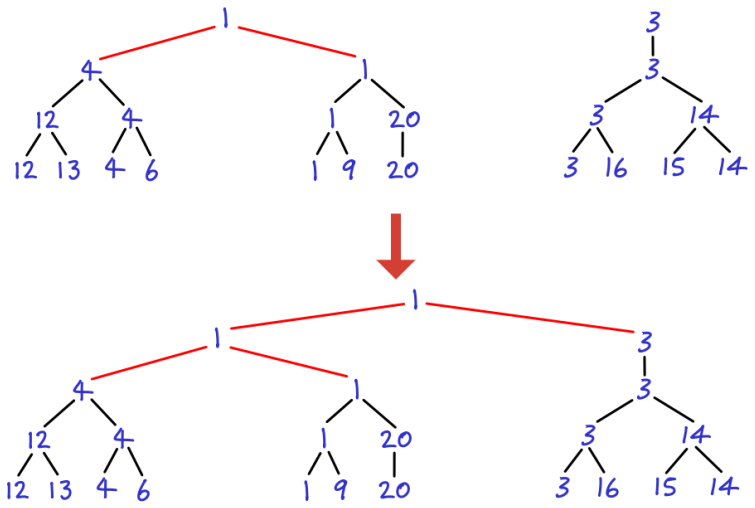


Figure 3: Linking two tournament trees[2]

- **Cut:** Simply remove the entire subtree at the edge. O(1) time. The cut operation is applied to non-root nodes, at the highest level that particular element exists.

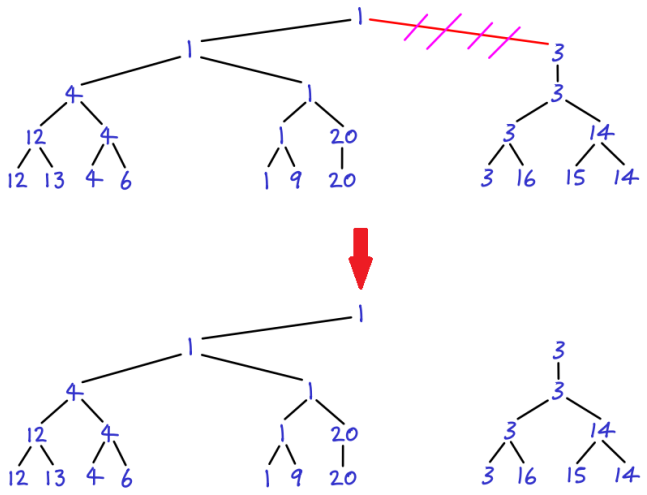


Figure 4: Cutting at an edge

1.2 Quake Heap Operations

We keep track of number of elements at each level. To avoid repeatedly storing the same node we can simply keep track of the topmost node where each element appears.

We will focus on Insert, ExtractMin, and DecreaseKey operations. The potential function

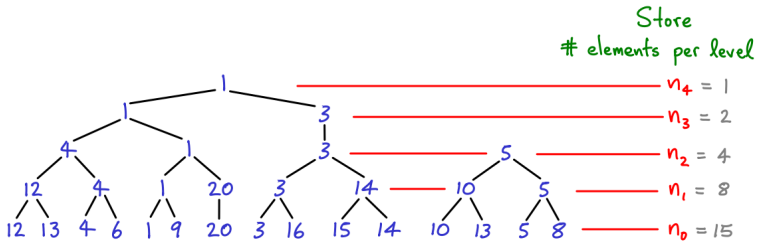


Figure 5: Storing number of elements at every level

is given by $\Phi(H) = N + f.T + g.2B$ where N is the number of Nodes, T is the number of Trees and B is the number of nodes with a single child. f and g are constants.

1.2.1 Insert

Insertion is lazy. Simply add the element as a single-node tree to the root list The actual cost is $O(1)$. The number of trees (T) increases by one. The change in potential is $\Delta\Phi = \Phi(\text{after}) - \Phi(\text{before}) = (N+1) + (T+1) + 2B - (N+T+2B) = 2$. The amortized cost is:

$$\hat{C}(Insert) = C(Insert) + \Delta\Phi = O(1) + 2 = O(1)$$

1.2.2 DecreaseKey

Simply cut the subtree rooted at the highest node storing the element in question and then reduce the key. (yields 1 new tree). This is also a constant time operation because the real cost is $O(1)$ and change in potential will be 3. (As both T and B increase by 1 when we cut out an entire subtree).

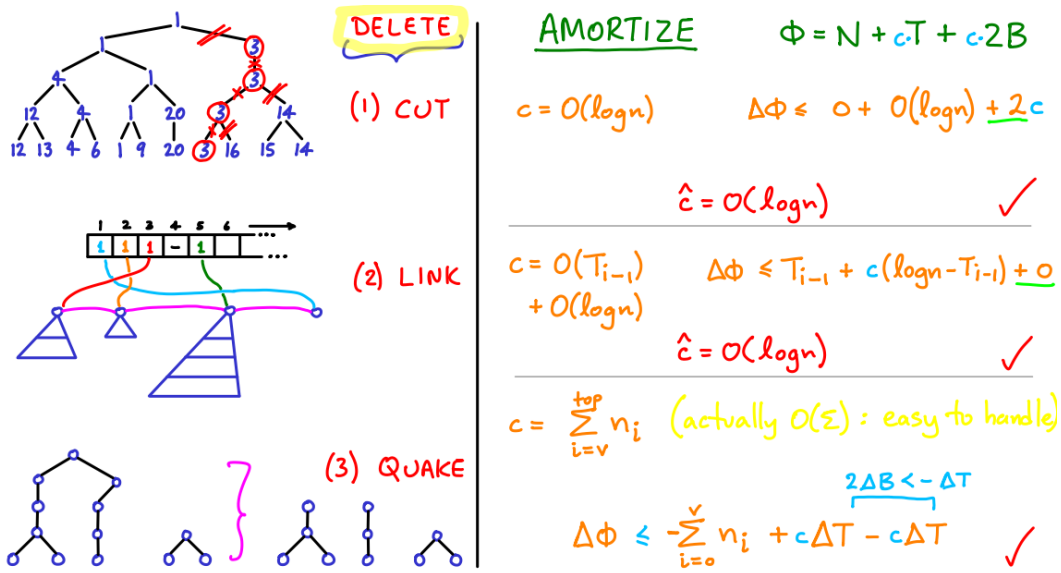
1.2.3 ExtractMin

Invariant: $\frac{n_{i+1}}{n_i} \leq \alpha$ $(0.5 < \alpha < 1)$ where n_i is the number of nodes at level i in the data structure and α is a fixed constant.
The ExtractMin Operation can be broken down into roughly 3 steps:

1. remove the path of nodes storing minimum (yields multiple new trees)
2. while there are 2 trees of the same height: link the 2 trees (reduces the number of trees by 1 while adding an extra node to the tree)
3. Quake - if $n_{i+1} > \alpha n_i$ for some i then: let i be the smallest such index, remove all nodes at $heights > i$ (increases the number of trees)

Amortized cost of the first two operations is $O(\log n)$ and quake is essentially free. (Highly recommmend looking at slides 40-86 of `https://www.eecs.tufts.edu/~aloupis/comp150/classnotes/Quake-heap.pdf`)

Here the c in the potential function i.e. c.T and c.2B is a constant and can be ignored



2 Scapegoat Trees

A "scapegoat tree" is a type of self-balancing binary search tree designed to maintain a balance between efficient search operations and relatively simple insertion and deletion procedures. They aim to provide the efficiency of a balanced binary search tree, like an AVL tree or a Red-Black tree, while minimizing the amount of restructuring (rotations or rebalancing) needed after each insertion or deletion. The structure of a scapegoat tree ensures that, while individual operations may temporarily unbalance the tree, the amortized cost of all operations remains efficient.
Here's a high-level overview of how a scapegoat tree works:

1. Insertions and deletions are initially performed in a way similar to a regular binary search tree, without strict adherence to height balance.
2. When the height of the tree exceeds a certain balance threshold (typically a constant factor times the logarithm of the number of nodes), a rebalancing operation is triggered.
3. During rebalancing, the scapegoat tree identifies a "scapegoat" node that is responsible for the height imbalance.
4. The subtree rooted at the scapegoat node is then rebuilt to restore the balanced property.

2.1 Invariants

No flags are used; Instead we keep only 2 measures $\Rightarrow$ n and q where
n = number of items in tree and q is an overestimate of n such that the following invariant is maintained at all times : $\frac{q}{2} \leq n \leq q$ and $height \leq \log_{\frac{3}{2}} q$

2.2 Operations

2.2.1 Insert

1. Do standard BST insert
2. Increment q and n by 1
3. Check invariant 2, IF $newht > \log_{\frac{3}{2}} q$, there is a Scapegoat Node present on the path from the newly inserted node to the root(proof below); Find Scapegoat Node W and rebalance subtree rooted at W. A Scapegoat is a node with $\frac{size(W.child)}{size(W)} > \frac{2}{3}$

   Claim: For a scapegoat tree with n nodes suppose p is a freshly inserted node satisfying:

   $$depth(p) > \log_{3/2} q$$

   then an ancestor of p is a scapegoat.

   Proof: Suppose that no node from p to the root is a scapegoat. This means that for every node u from the root down to p we have:

   $$\frac{size(u.child)}{size(u)} \leq \frac{2}{3}$$

   $$n = size(root)$$

   $$n \geq \frac{3}{2} size(root.child)$$

   $$n \geq \left(\frac{3}{2}\right)^2 size(root.child.child)$$

   $$....$$

   $$n \geq \left(\frac{3}{2}\right)^{depth(p)} size(p)$$

   but given p is the newly inserted node, it's size is 1. Therefore we get:

   $$n \geq \left(\frac{3}{2}\right)^{depth(p)}$$

   As q is an *overestimate* of n,

   $$q \geq \left(\frac{3}{2}\right)^{depth(p)}$$

   But from this, we get the contradiciton:

   $$depth(p) \leq \log_{\frac{3}{2}} q$$

   Therefore there has to be some node on the path from the newly inserted node p to root that is a scapegoat

2.2.2 Delete

1. decrement n by 1
2. BST Delete
3. IF $q > 2n$, Destroy ENTIRE tree, rebuild, and set $q = n$

2.3 Amortization

As rebalance and destroy operations are expensive, we need to pay for them in the insert and delete operations. We will use the accounting method to analyse the cost:

Lemma: *Starting with an empty tree, any sequence of m dictionary operations (find, insert,and delete) to a scapegoat tree can be performed in time O(m log m).*
Proof:

1. **Insert Operation:**
   The actual cost of inserting a node into a scapegoat tree is $O(logn)$, in addition we put away 1 extra unit cost per node touched for insert operation as "credit" for future rebuilding operations. This is complicated by the fact that subtrees of various sizes can be rebuilt. Intuitively, after any subtree of size k is rebuilt, it takes an additional $O(k)$ (inexpensive) insert/delete operations to force this subtree to become unbalanced and hence to be rebuilt again. We charge the expense of rebuilding to against these "cheap" insertions. Thus the amortized cost of a single insert is $O(\log n)$ and thus the cost of m inserts would be $O(m \log n)$
2. **Delete Operation:** The cost of a BST delete operation is $O(\log n)$. In order to rebuild the entire tree due to deletions, at least half the entries since the last full rebuild must have been deleted. (The value $q - n$ is the number of deletions, and a rebuild is triggered when $q > 2n$, implying that $q - n > n$) Thus, the $O(n)$ cost of rebuilding the entire tree can be amortized against the time spent processing the (inexpensive) deletions by paying with an additional credit for every node touched in the delete operation and another extra credit on the side. Thus the amortized cost of a single delete is $O(\log n)$ and thus the cost of m deletes would be $O(m \log n)$
   see: [3]

References

[1] T. M. Chan, "Quake heaps: A simple alternative to fibonacci heaps," in *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*. Springer, 2013, pp. 27–32.

[2] G. Aloupis, "Quake Heaps lecture notes," https://www.eecs.tufts.edu/~aloupis/comp150/classnotes/Quake-heap.pdf.

[3] D. Mount, "Scapegoat Trees Lecture notes," https://www.cs.umd.edu/class/fall2020/cmsc420-0201/Lects/lect12-scapegoat.pdf.