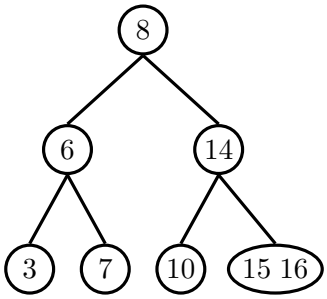
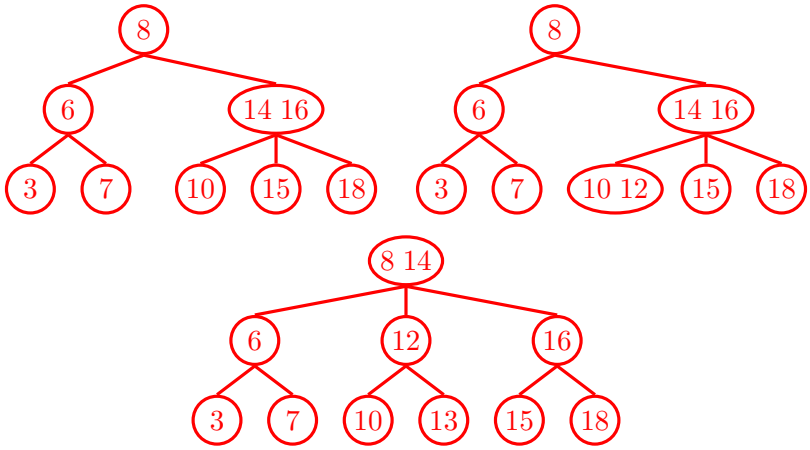
CSM 61B Heaps

Spring 2019 Mentoring 8: November 5, 2018

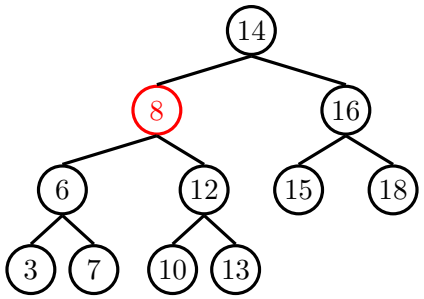
1 2-3 Forever



1.1 Draw what the 2-3 tree would look like after inserting 18, 12, and 13.



1.2 Now, convert the resulting 2-3 tree to a left-leaning red-black tree.



2 Min-Heapify This

2.1 In general, there are 4 ways to heapify. Which 2 ways actually work?

- Level order, bubbling up
- Level order, bubbling down

- Reverse level order, bubbling up
- Reverse level order, bubbling down

Only level order, bubbling up and reverse level order, bubbling down work as they maintain heap invariant. Namely, that every node is either larger (in a max heap) or smaller (in a min heap) than all of its children.

Meta: Students often ask about the runtime of these methods. The specific runtime for heapification is hard to prove, but specifically level order, bubbling up will take $O(N \log(N))$ and reverse level order, bubbling down takes $O(N)$, so reverse level order + bubbling down is faster. Intuition on why it is faster: the majority of the nodes are near the bottom of the tree, so a lot of the nodes bubble down really quickly.

2.2 Are the values in an array-based min-heap sorted in ascending order?

Not necessarily. We can have higher or lower priority items loaded on one branch of the tree.

2.3 Is an array that is sorted in descending order also a max-oriented heap?

True, the heap invariant holds.

3 K Largest Items

3.1 The largest item in a heap must appear in position 1, and the second largest must appear in position 2 or 3. Give the list of positions in a heap where the kth largest can appear for $k \in \{2, 3, 4\}$. Assume values are distinct.

$k = 2$ can be in $\{2, 3\}$. $k = 3$ can be in $\{2 \ldots 7\}$. $k = 4$ can be in $\{2 \ldots 15\}$.

Consider complete binary trees with the largest values contained on one branch of the tree for a lower bound and consider how far the kth element can be from the root for an upper bound.

4 Amortized Analysis

4.1 Give the *amortized runtime analysis* for push and pop for the priority queue below.

```
class TwinListPriorityQueue<E implements Comparable> {
    ArrayList<E> L1, L2;
    void push(E item) {
        L1.push(elem);
        if (L1.size() >= Math.log(L2.size())) {
            L2.addAll(L1);
            mergeSort(L2);
            L1.clear();
        }
    }
    E pop() {
        E min1 = getMin(L1);
        E min2 = L2.poll();
        if (min1.compareTo(min2) < 0) {
            L1.remove(min1);
            return min1;
        } else {
            L2.remove(min2);
            return min2;
        }
    }
}
```

Let N be the number of elements in the priority queue. Then the amortized runtime for `push` is in $O(N)$ as the cost for every $\log N$ insertions is in $O(\log N \cdot 1 + 1 \cdot N \log N)$ which simplifies to $O(N)$. Note that the size of `L1` is always constrained to be in $O(\log N)$.

The amortized runtime for `pop` is also in $O(N)$. `getMin` on the unsorted list, `L1`, is in $O(\log N)$, as with `L1.remove(min1)`. Polling from the front of `L2` is in $\Theta(1)$. The most expensive component is `L2.remove(min2)` which is in $O(N)$.

5 Kelp!

5.1 You have been hired by Alan to help design a priority queue implementation for *Kelp*, the new seafood review startup, ordered on the timestamp of each `Review`.

Describe a data structure that supports the following operations.

- `insert(Review r)` a `Review` in $O(\log N)$.
- `edit(int id, String body)` any one `Review` in $\Theta(1)$.
- `sixtyOne()`: return the sixty-first latest `Review` in $\Theta(1)$.
- `pollSixtyOne()`: remove and return the sixty-first latest `Review` in $O(\log N)$.

Maintain a max-heap called `firstSixtyOne` with 61 `Reviews`, a min-heap called `olderReviews` with all the rest, and a `HashMap` mapping any given integer `id` to its corresponding `Review`.