CSM 61B
Spring 2019

Heaps & Tries
Mentoring 8: March 18, 2019

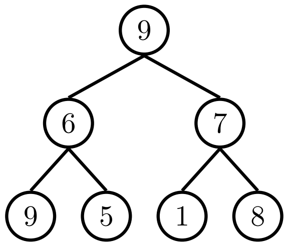
1 Heaps of Fun

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

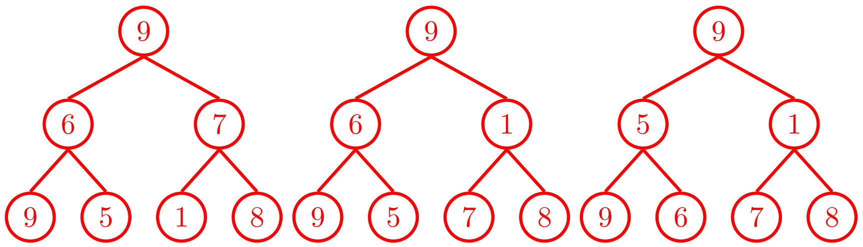
- Level order, bubbling up
- Level order, bubbling down
- Reverse level order, bubbling up
- Reverse level order, bubbling down

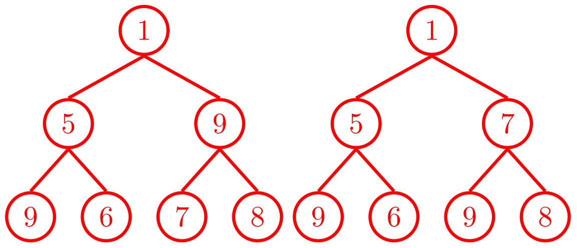
Only level order, bubbling up and reverse level order, bubbling down work as they maintain heap invariant. Namely, that every node is either larger (in a max heap) or smaller (in a min heap) than all of its children.

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.

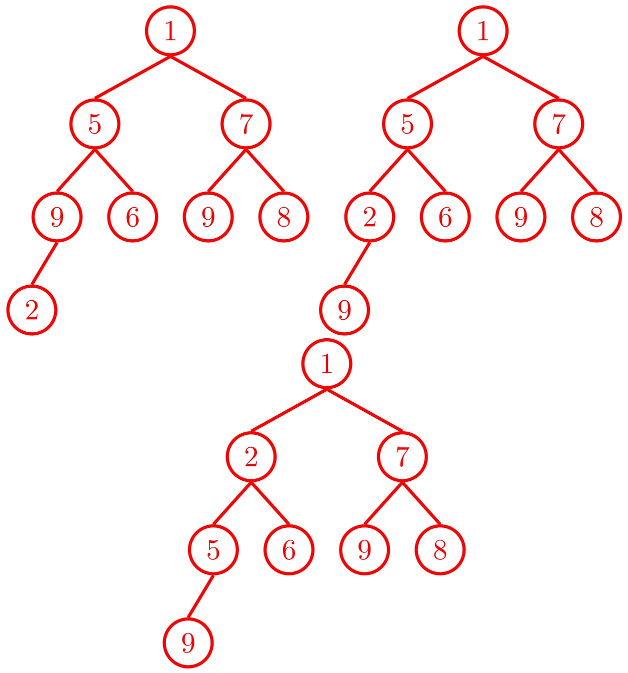


1.2 (a) Show the heapification of the tree. (note we want to create a min-heap)

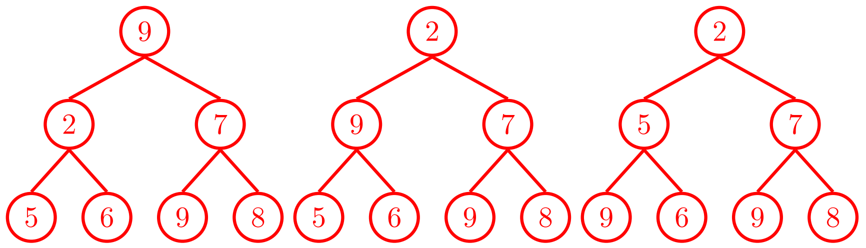




(b) Now, insert the value 2.



(c) Finally, remove the value 1.



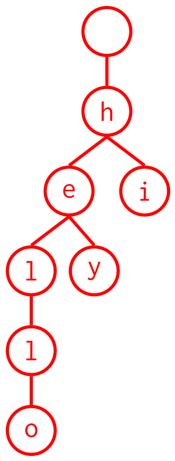
1.3 The largest item in a heap must appear in position 1, and the second largest must appear in position 2 or 3. Give the list of positions in a heap where the kth largest can appear for $k \in \{2, 3, 4\}$. Assume values are distinct.

$k = 2$ 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.

2 Tries

2.1 Draw the trie that results from inserting `"hi"`, `"hello"`, and `"hey"`.



2.2 Given a list of words (possibly repeated), devise a strategy to efficiently return a list of all the words that start with a given prefix.

Put all the names into a trie, lookup the prefix in the trie, and iterate across all the children rooted at that node.

2.3 Given a dictionary of n words, where the average length of the words is l, how long would it take to insert all of the words into a Trie? Express your answer in terms of a tight asymptotic bound.

Runtime would be $\Theta(nl)$.
For each word in our dictionary, we must either traverse through l nodes if the suffix already exists in the trie or add l nodes if a suffix does not exist in the trie. Both cases involve iterating l times and since there are n words, the total runtime of adding all the words from the dictionary into the Trie would be $\Theta(nl)$.

2.4 With that same Trie already containing those n words, how long would it take to check whether or not a new word of length l was in our initial dictionary? Express the runtime in both Big Omega and Big O notation.

Runtime would be $O(l)$ and $\Omega(1)$. Worst case, all letters of our new word exist in that sequence inside our Trie, so we must iterate over every letter of our new word. Best case, the first letter of our new word, is not the first letter of any word in our initial dictionary (ie. given a new word `"hello"` for a dictionary of `{"asdf", "cat", "pidgeon"}`, the starting node of our trie would not have any edge for the character `h` so we immediately return **`false`**).

3 Hashing Again

3.1 You are a software engineer for a newspaper company! Your users are complaining about how slowly your website loads. After performing some performance profiling, you realize that the database queries are slowing the system down.

To fix the issue, you decide to implement a cache that contains the most recently accessed articles. The cache is only fast if it's small so you can only store a maximum of N articles. You want to keep only the N most recent articles that people have read. If a new, unique article is accessed, then the oldest article should be replaced.

Describe how you would implement this cache. What combinations of data structures would you use to build this efficiently?

Use a `HashMap` with references to nodes in a doubly linked list. The `HashMap` will map the name of the article to a tuple containing the article as well as a reference to a node in a doubly linked list. If the article we are trying to access is in the `HashMap`, we deliver the article, and then retrieve its respective node in the linked list and move it to the front. If the article is not in the `HashMap`, and the size of the `HashMap` is still less than N, we can fetch from the main server and create a new node to append to the front of the linked list. If the `HashMap` is at capacity, we look at the tail of the linked list and delete that article from both the linked list as well as the `HashMap`. We then add the new article into the `HashMap` and to the front of the linked list. In Java there's actually a data structure for this: `java.util.LinkedHashMap`!

4 Extra Practice: Trie-ing Extra Hard

You're living in the future in the late 1990's. Everyone owns a cell phone — the handheld kind — and SMS is the bee's knees. However, cell phones only have 9 keys while there are 26 letters in the alphabet. Your company is on the verge of developing a new algorithm for faster texting called "Text on 9 keys", or "T9". Each keypress maps to one of three or four letters in the alphabet.

Given a trie containing the dictionary of words, define a procedure, `getWords`, that returns the set of matching words for a given key-press sequence.

```
public interface TrieNode {
    public char getCharacter();
    public boolean isWord();
    public String getWord();
    public Map<Character,TrieNode> getChildren();
}
```

```
public class T9 {
    private static char[][] KEY_MAPPINGS = {
        {}, {}, // keys 0 and 1 don't map to any characters
        {'a', 'b', 'c'},
        {'d', 'e', 'f'},
        {'g', 'h', 'i'},
        {'j', 'k', 'l'},
        {'m', 'n', 'o'},
        {'p', 'q', 'r', 's'},
        {'t', 'u', 'v'},
        {'w', 'x', 'y', 'z'}
    };
    public static Set<String> getMatches(int[] keyPresses, TrieNode words) {
        return getMatches(keyPresses, words, 0);
    }

    private static Set<String> getMatches(int[] keyPresses, TrieNode wordNode, int index) {
        Set<String> matches = new HashSet<String>();
        if (index == keyPresses.length) {
            if (wordNode.isWord()) {
                matches.add(wordNode.getWord());
            }
            return matches;
        }
        Map<Character,TrieNode> children = wordNode.getChildren();
        for (Character c : KEY_MAPPINGS[keyPresses[index]]) {
            if (children.containsKey(c)) {
                matches.addAll(getMatches(keyPresses, children.get(c), index + 1));
            }
        }
        return matches;
    }
}
```