CSM 61B Spring 2019

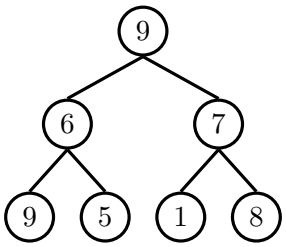
Heaps & Tries

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



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.

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.

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.

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.

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?

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) {
```