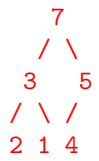
CS61B Spring 2016 Secret Section 5 Worksheet

4/4

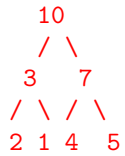
1 Heap Refrehser

(a) Give the array that results if we use transform the array [9,3,7,2,1,4,5] into a binary max heap.
The array stays the same if using bottom-up heapify

(b) Remove 9. What does the heap look like now? What is the runtime for removal?
$O(\log n)$



(c) Insert 10. What does the heap look like now? What is the runtime for insertion?
$O(\log n)$



(d) Check if the heap contains 6. What is the runtime for checking if a heap contains an element?
Does not contain, $O(n)$

2 Bored of Hashing?

Based on http://inst.eecs.berkeley.edu/ cs61b/fa13/samples/test2-soln.pdf, see link for a simpler version of this

(a) Suppose we are keeping a HashSet of 8-puzzle boards for some homework in some Berkeley CS class. The boards use a hashcode that takes the tile values of the upper left tile and bottom right tiles (0 for the blank space) and returns their sum. *Assume that the boards are evenly distributed by this hashcode.* Using **N** = the number of boards in the HashSet, what is the worst case big-O time for adding a new board? See if you can derive the worst case tilde notation time as well.
$O(N)$
Tilde: $\frac{N}{15}$
The lowest possible hashcode is $0+1=1$ and the highest is $7+8=15$. The hash function will distribute the boards evenly, but only into buckets $1-15$, so bucket iteration time grows with $\frac{1}{15}N$

(b) (**Optional**) Now suppose we discover our HashSet needs to concurrently store 8-puzzle boards of varying sizes (So there might be 1000×1000 boards alongside 3×3 boards). *You can assume the square of the board sizes in the HashSet are evenly distributed.* Using the same hashcode and assumptions from part (a), how does the worst-case big-O time change?
$O(1)$
The hashcodes should evenly distribute across all buckets now thanks to the varying board sizes.
More rigorous rationale:
Let $n = m^2 - 1$, or the maximum tile value. Expected value of the hashcode $= 2 * E(\textbf{corner tile value}) = 2 * \frac{n(n+1)}{2}\frac{1}{n} = O(n) = O(m^2)$
So the hashcode grows quadratically with board size, and with evenly-distributed m^2 you can conclude the above.
Note: if the assumption was just that m was evenly distributed, the hashcodes would no longer be a uniform distribution - the distribution would look something more like the graph of $y = \frac{1}{\sqrt{2x}}$

3 Yelplet

In this problem we will be making a class for a microscopic version of Yelp! This version only keeps track of business and the days that they are open.

To satisfy users, try to make `isBusinessOpen()` and `getBusinessesByDay()` as efficient as possible (imagine when a lot of businesses have been added already).

Assumptions you can make:

- All argument inputs will be sane (eg. non-null, and inputs to the day argument will always be a valid day)
- You have a helper method `int dayToIndex(String day)` which takes in "Monday" as a string and returns 0, "Tuesday" as a string and returns 1, etc. to "Sunday" returning 6.

Note: No need to use a HashMap for mapping days to businesses since there are a constant number of days. A 2D List (of LinkedLists for faster appending) will have no collisions, and possibly less calculation/space.

```
public class Yelplet {
    final static int NUM_DAYS = 7;

    HashMap<String, HashSet<String>> bizToDays;
    List<List<String>> dayToBiz;

    public Yelplet() {
```

```
        bizToDays = new HashMap<String, HashSet<String>>();
        dayToBiz = new ArrayList<List<String>>(NUM_DAYS);

        for (int i = 0; i < NUM_DAYS; i++) {
            dayToBiz.add(new LinkedList<String>());
        }
    }

    public void addBusiness(String name, List<String> daysOpen) {
        bizToDays.put(name, new HashSet<String>(daysOpen));
        // Or fill in a new HashSet via for loop

        for (String day: daysOpen) {
            dayToBiz.get(dayToIndex(day)).add(name);
        }
    }

    public boolean isBusinessOpen(String name, String day) {
        HashSet<String> days = bizToDays.get(name);

        return days != null && days.contains(day);
    }

    public List<String> getBusinessesByDay(String day) {
        return dayToBiz.get(dayToIndex(day));
    }
```

4 Huffman Coding

Huffman coding is a technique used for converting words to short codes when compressing data. The input is typically given as a list of words and frequencies:

Swag: 5
Best: 6
Is: 7
CS61B: 15
The: 6

Let's say each word-frequency pair is a node. When running Huffman coding, you first find the two nodes A, B with the least frequencies. You then create a new node C with children A, B, whose frequency is the sum of the two. You repeat this process until all the nodes are connected.
(Note: The newly-created node C contains a frequency and should be included when searching for the least frequent nodes. The newly-connected nodes A, B are removed from the search since they are now combined under C)

What data structure(s) would you use to run Huffman coding?
A priority queue of binary trees is the ideal way to do this

Run Huffman coding using these data structure(s) until the final state.
Answers may vary slightly due to the tie between "Best" and "The". An example series of steps:

```
- (15) (7) (6) (6) (5)

- (15) (7) (6) (11)
                / \
              (6) (5)

- (15)   (13)    (11)
         / \     / \
       (7) (6) (6) (5)

Final Tree:

      (39)
     /    \
 (CS61B)   (24)
          /    \
       (13)     (11)
       /   \    /   \
    (Is) (The)(Best)(Swag)
```

Let's say we want to use your data structure, in its final state, to find words with frequencies in the range $6 \leq f \leq 7$. With that constraint, use an arbitrary traversal/search and give the search's resulting word order.
Answers may vary depending on search.
Preorder (left-leaning) DFS example: "Is", "The", "Best"