Name: Stewart Dulaney **ID:** 1545566

Problem 1:

Recall that a map (or dictionary in some languages) associates a key with a corresponding value or values. in stl, the std::map implements this behavior using a Binary Search Tree, which implies that keys are in some sorted order. So, if we search the std::map for some key, i.e. someTreeMap[key], this will find the key in logarithmic time and retrieve the associated value. On the hand, stl also implements a map using Hash Tables, std::unordered_map. The implication here is that they key is the item that will be hashed to a corresponding bucket. In this case, someHashMap[key] will hash key to a bucket, and we can retrieve the value corresponding to that key in constant time, assuming little to no collisions.

Suppose we created the following object for an application:

```
struct BadMovie {
        int databaseID;
        string name;
        string director;
        int runtimeInSeconds;
        int rating;
};
```

We decided that a Hash Table is the most appropriate data structure for our purposes. However, in our experiments, hashing on the databaseID alone results in many collisions. So, we decided to write a custom hashing function that incorporates all of the member variables to determine the appropriate bucket. Complete such a hash function below, you may assume that the constant ARRAY_SIZE is defined.

```
int hash(const BadMovie &bm) {
    int total = 0;
    total += bm.databaseID;
    total += bm.runtimeInSeconds;
    total += bm.rating;
    for ( int i=0; i < bm.name.length(); i++ ){
        total += (i+1) * bm.name[i];
    }
    for ( int i=0; i < bm.director.length(); i++){
        total += (i+1)* bm.director[i];
    }
    total %= ARRAY_SIZE;
    return total;
}
```

Name: Stewart Dulaney **ID:** 1545566

Problem 2:
Recall that the Tree based heap has a few challenges in locating the appropriate node/location for extraction/insertion, and how to swap with parent nodes in the reheapification process of insertion. To get around all these issues, we make use of the fact that a Heap is by definition a complete tree to encode the Heap as an array instead. So, what would have been some costly traversals, in the case of the Tree, boils down to some arithmetic to access the appropriate array elements. The question remains however whether or not the performance of the array based Heap warrants the seemingly complicated implementation.

a. Below is the pseudocode for Extracting the largest item from a Max Heap. What is the Big-O for this operation assuming array based implementation?

1. **If the tree is empty, return error.**
2. **Otherwise, the top item in the tree is the biggest value. Remember it for later.**
3. **If the heap has only one node, then delete it and return the saved value.**
4. **Copy the value from the right-most node in the bottom-most row to the root node.**
5. **Delete the right-most node in the bottom-most row.**
6. **"Sifting DOWN": Repeatedly until the value is greater than or equal to both of its children.**
    a. **Compare with its children**
    b. **swap with the larger of its two children**
7. **Return the saved value to the user.**

O (log n)

b. Below is the pseudo code for Inserting an item into the Max Heap. What is the Big-O for this operation?

1. **If the tree is empty, create a new root node & return.**
2. **Otherwise, insert the new node in the bottom-most, left-most position of the tree (so it's still a complete tree).**
3. **Reheapification: Repeatedly until the value is less than its parent**
    a. **Compare the new value with its parent's value.**
    b. **If the new value is greater than its parent's value, then swap them.**

O (log n)

Name: Stewart Dulaney **ID:** 1545566

Problem 4::
Consider the following array-based maxheap that has two operations, extractMax() and insert(int num).

15	10	14	7	9	8	11	4	3	5	6

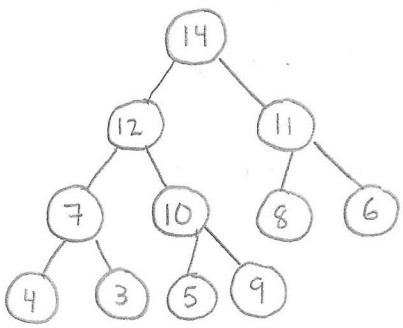
a. How does the maxheap look after calling extractMax()?

14	10	11	7	9	8	6	4	3	5	

b. How does it look after calling insert(12)?

14	12	11	7	10	8	6	4	3	5	9
0	1	2	3	4	5	6	7	8	9	10

c. Draw the equivalent complete binary tree of this maxheap.



Problem 5:
a. Suppose we have a maxheap which can be visualized with a ternary tree (up to three children). If we wanted to convert this to an array based maxheap, how would we find the left, middle, right child for any particular node? The parent for any particular node?

```
int leftChildIndex = 3 * parentIndex + 1;
int middleChildIndex = 3 * parentIndex + 2;
int rightChildIndex = 3 * parentIndex + 3;
```

```
int parentIndex = (childIndex - 1) / 3;
// Note this formula works for left, middle, and right children b/c
// of integer division.
```

Name: Stewart Dulaney **ID:** 1545566

b. What is the complexity of insertion into this maxheap? Complexity of extraction?

$O(\log_3 n)$ insertion

$O(\log_3 n)$ extraction

Problem 6:
You are hired to design a website called *brotionary.com*, a *dictionary.com* variant. You are given a list of **N** dictionary words (and their definitions) in a text file, and would like to preprocess it, such that you can readily provide information to users who visit your website and look up words. Assume that your dictionary is not going to be updated once it is preprocessed. We still want your system to "scale" -- that is, it should be able to efficiently take care of **a lot of queries** that may come in.

Assume a Word structure like the following is used to store each word and definition.

```
struct Word {
        string word;
        string definition;
};
```

a. First of all, the users should be able to look up words. Which of the following options would you take, and why?

Option A: Store the Words in a binary search tree.
Option B: Store the words in a hash table.

Option B b/c (1) a hash table is more efficient for searching (O(n)) than a BST (O(log n)) (2) we don't need to sort the words which would be one reason to use a BST and (3) since we know the dictionary is not going to be updated we can choose a hash table size that we know will give us the load factor we want and we'll know this won't be affected by more inserts.

Name: Stewart Dulaney **ID:** 1545566

b. You want to add functionality that prints words and their definitions within a certain range of **P** words. e.g all the words between and including aardvark and awkward. Which of the following options would you take, and why?

Option A: Add a sorted singly linked list with pointers to Words in the structure used in part a. Each time a range [x:y] is specified, we search for word x in the list, and then traverse the list to print each word, until we hit word y.
Option B: Add a sorted vector with pointers to Words in the structure used in part a. Each time a range [x:y] is specified, we search word x in the vector, and then traverse the vector to print each word, until we hit word y.

Both options have the same time complexity to search for word x (O(n)) and then traverse the data structure until we hit the word y (worst case O(n)) b/c both data structures are already sorted. In this case, I'd take Option B b/c vectors are generally easier to program than linked lists.

Name: Stewart Dulaney **ID:** 1545566

c. In the right corner of the website, you want to display "**K** most popular words", where **K** is some integer, which are determined by queries that were received in the past hour, and is updated every hour. Assume queries are made on **M** distinct words, where **M** >>**K**. Which option would you take, and why?

Option A: In the beginning of every hour, create a hash table (initially empty) that stores (word, count)-pairs. Each time a query for a word x comes in, look up x in the hash table (or add a new one if one does not exist), and increase the count for x. At the end of the hour, iterate through all (word, count)-pairs in the hash table and store them in a maxheap, using their counts as the keys. Then extract **K** words from the heap.
Option B: In the beginning of every hour, create a vector (initially empty) that stores (word, count)-pairs. Each time a query for a word x comes in, look up x in the vector (or add a new one if one does not exist), and increase the count for x. At the end of the hour, sort the pairs in the vector in the decreasing order of their counts, and print the first **K** words

Option A:

$O(1) + O(1) = O(1)$ hash table search + insertion (Query)

$O(M \log M)$ iterate over hash table, insert each item into maxheap (Sort)

$O(K \log M)$ extract K words from maxheap (Display)

Option B:

$O(M) + O(1) = O(M)$ vector search + insertion (Query) } Happens M times

$O(M \log M)$ sort vector using Merge Sort (Sort) } Happen once

$O(K \cdot 1) = O(K)$ print First K words in vector (Display) } Happen once

I'd take Option A b/c it is more eppicient for the Query step, which happens M times. Even though Option A is less eppicient for the Display step, this step only happens once.