Instructor:

- Mr. Nazeef Ul Haq

Registration No._____

Name: _____

Guide Lines/Instructions:

You may talk with your fellow CS261-ers about the problems. Howevesr:

- Try the problems on your own *before* collaborating.
- Write up your answers yourself, in your own words. You should never share your typed-up solutions with your collaborators.
- If you collaborated, list the names of the students you collaborated with at the beginning of each problem.

Today's Task:

- Design of Balanced BST
- Implementation of Hash Tables

Part 1: Design of Data Structures(RB Tree)

1. Implement **RB Tree** class in C++ which must have following functions.

```
class RbTree: BST {
public:
        RBTree(void);   // constructor
        RBTree(int arr[], int size);          // constructor to build tree from array
        ~ RBTree (void);          // destructor
        //override the following functions
        void visualizeTree(Node * T); provide visualization of tree on console
        Node* Insert(int x)
        Node * Delete(int x)

private:s
        Node* root;
};
class Node{
        int data;
        Node *parent;
        Node *left;
        Node *right;
        bool color;  //1 for red, 0 for black, only use it for RB Tree
};
```

2. Implement **AVL Tree** class in C++ which must have following functions.

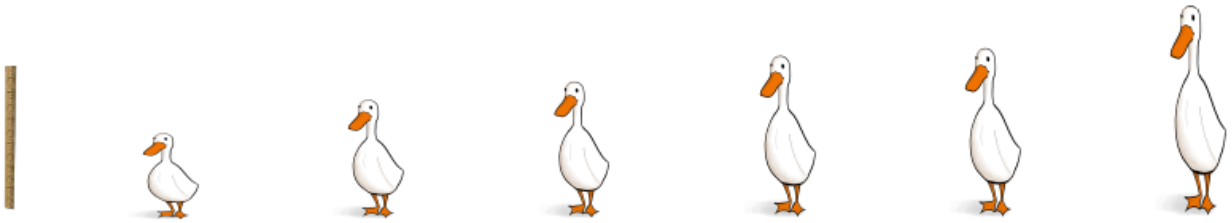
```
class AvlTree: BST {
```

```
public:
        AvlTree(void);  // constructor
        AvlTree (int arr[], int size);        // constructor to build tree from array
        ~ AvlTree (void);          // destructor
        //override the following functions
        void visualizeTree(Node * T); provide visualization of tree on console
        Node* Insert(int x)
        Node * Delete(int x)

private:
        Node* root;
};
```

Part 2: Application of Custom Data Structures for algorithms

3. **The impossible job interview:** You're interviewing for your dream job at an ecological ethical tech company with healthy snacks. You already passed 28 stages of interviews, and your final interviewer asks you to design a binary search tree data structure that performs INSERT operations in $O(\sqrt{\log n})$ time using a comparison-based algorithm. Design such a data structure or prove that this is impossible. [**We are expecting**: If possible: An English description of the algorithm and a run time analysis. If impossible: A formal proof that this is impossible.]
4. Suppose that n ducks are standing in a line, ordered from shortest to tallest (not necessarily of unique height).



You have a measuring stick of a certain height, and you would like to identify a duck which is the same height as the stick, or else report that there is no such duck. The only operation you are allowed to do is compareToStick(j), where $j \in \{0, \ldots, n-1\}$, which returns taller if the j'th duck is taller than the stick, shorter if the j'th duck is shorter than the stick, and the same if the j'th duck is the same height as the stick. You forgot to bring a piece of paper, so you can only remember a constant number of integers in $\{0, \ldots, n-1\}$ at a time.

   (a) Give an algorithm which either finds a duck the same height as the stick, or else returns "No such duck," in the model above which uses O(log(n)) comparisons. [We are expecting: Pseudocode AND an English description of your algorithm. You do not need to justify the correctness or runtime.]
   (b) Prove that any algorithm in this model of computation must use $\Omega(\log(n))$ comparisons. [We are expecting: A short but convincing argument.]

5. [**Goose**!] A goose comes to you with the following claim. They say that they have come up with a new kind of binary search tree, called gooseTree, even better than red-black trees! More precisely, gooseTree is a data structure that stores comparable elements in a binary search tree. It might also store other auxiliary information, but the goose won't tell you how it works. The goose claims that gooseTree supports the following operations:

- gooseInsert(k) inserts an item with key k into the gooseTree, maintaining the BST property. It does not return anything. It runs in time O(1).
- gooseSearch(k) finds and returns a pointer to node with key k, if it exists in the tree. It runs in time O(log(n)).
- gooseDelete(k) removes and returns a pointer to an item with key k, if it exists in the tree, maintaining the BST property. It runs in time O(log(n)).

Above, n is the number of items stored in the gooseTree. The goose says that all these operations are deterministic, and that gooseTree can handle arbitrary comparable objects. You think the goose's logic is a bit loosey-goosey. How do you know the goose is wrong?

Notes:

- You may use results or algorithms that we have seen in class without further justification.
- Since the gooseTree is still a kind of binary search tree, you can access the root of gooseTree by calling gooseTree.root().

[**We are expecting:** Formally prove that the goose is wrong by showing that we can solve an algorithmic problem that we know the lower bound for with this data structure.]

Part 3: Hash Tables Implementation

Hash tables are among the most common and most useful general-purpose data structures. A hash table is a generalization of an array. An array can be thought of as a data structure that maps integers to items of a given type. A hash table can map arbitrary (constant) items of one type (known as keys) to items (not necessarily constant) of another type (known as values). Hash tables are most often used to map strings to data, and that's what we'll be doing here. You can think of this as if you had an array which was indexed by a string value instead of by an integer. Note that hash tables are not the only way to build such a mapping, but they are one of the most efficient.

Hash functions

The basic idea behind a hash table is this. You take your key and process it with a special function called a hash function. This function will transform any key into a non-negative integer with a bound which is proportional to the size of the table. In our case it will transform a string into a value between 0 and 127. Ideally, a hash function will map different keys to different hash values so that a random population of keys will give a uniform distribution of hash values.

The data structure

The kind of hash table we are implementing in this lab manual is essentially an array of objects. In fact, we define it as a class which contains an array of objects as its only field.

The reason we use a class containing an array instead of just using a plain array is that in a real hash table implementation, there would be more fields. For instance, a real hash table would have (at a minimum):

- a field representing the total number of entries in the hash table
- a field representing the size of the array

Every time a key/value pair was added to the hash table, the field representing the total number of entries would be incremented. Once the number of entries was sufficiently larger than the size of the array (say, more than twice as large), then a larger array would be allocated and all the entries in the original array would be copied to the new array (this is called rehashing). Doing this ensures that hash table lookups are fast.

For the purposes of this assignment, though, we will just use an array inside a class. The main reason for this is that it gives you experience working with more complex data structures.

The array indices are the same as the hash values (so in our case the array length is exactly 128), and the lists associated with each index are initially empty (set to NIL).

Adding items to a hash table

We add items to a hash table as a "key/value pair". Let's say our keys represent movie names and our values represent the rating of the movie in stars (0-5). A typical key/value pair might be ("Movie Title", 5). First, we compute the hash value for "Movie Title" (you'll see how below) and get (say) 47. The 47th location in the node pointer array will initially be empty (i.e. it will contain a NIL value). We then create a new list at this index. The object being inserted in the list here will have two fields: a field called key(string), a field called value which holds the int value. We put the (address of the) key string into the key field (here, "Movie Title"), and the value into the value field (here, 5). We then link it to location 47 in the hash table array by putting a pointer to the node in the node pointer array at location 47. Later, we might want to add another entry, say ("Movie Title 2", 0). First, we compute the hash value of " Movie Title 2". Now one of two things could happen:

If the hash value is not 47, we create another list with the appropriate key and value fields and add it to the place in the array corresponding to the hash value, just as we did for the first item.

If the hash value is 47 again (which ideally it won't be), we create a new object. Then we have to add it to the old list at array position 47. We do this by inserting the new object into the old list at any point (the order doesn't matter). Usually we either put it at the front or the back of the list.

Now you can see why hash values should be distributed randomly over the set of all possible keys: if all keys hashed to 47, we would end up with a list at one location in the array and nothing anywhere else. In that case, looking up elements would take time proportional to the number of entries. If most of the lists are either empty or have only one or two values, lookup will require a constant (small) amount of time. Of course, as the number of elements in the hash table increases in size, eventually all the slots will be occupied even with a perfect hash function. At this point (as mentioned above) you should probably create a bigger hash table.

Retrieving items from a hash table

To retrieve an item from a hash table means to retrieve the value given the key. The way this is done is as follows:

- Compute the hash value for the key.
- Find the location in the list corresponding to the hash value.
- Search the list at that location for the key. In general, there will be a very small number of items in the list (usually one or none), so the search will not take long.
- If the key is found, return the value. If not, return a "not found" value (this will become clearer later). If there was no list at that location in the array, also return the "not found" value.

As we said above, this means that our list objects have to have two elements:

- the key,
- the value associated with the key

Program to write
Your program will use a hash table to count the number of occurrences of words in a text file. The basic algorithm goes as follows. For each word in the file:

Look up the word in the hash table.

- If it isn't there, add it to the hash table with a value of 1. In this case you'll have to create a new object and link it to the hash table as described above.
- If it is there, add 1 to its value. In this case you don't create a new object.
- At the end of the program, you should output all the words in the file, one per line, with the count next to the word e.g.

cat 2
hat 4
green 12
eggs 3
ham 5
algorithmic 14
deoxyribonucleic 3
dodecahedron 400

The hash function
The hash function you will use is extremely simple: it will go through the string a character at a time, convert each character to an integer, add up the integer values, and return the sum modulo 128.

(Don't hard-code the number 128 into your code, use a variable)
This will give an integer in the range of [0, 127]. This is a poor hash function; for one thing, anagrams will always hash to the same value. However, it's very simple to implement. Note that a value of type char in a python can also be treated as if it were an int, because internally it's a one-byte integer represented by the ASCII character encoding. If you like, you can convert the char to an int explicitly. However, a string of characters is not an int, and you can't simply convert it into one (mainly because most of the keys will be words, not string representations of numbers). You also shouldn't try to type cast the string to an int. A string is an array of chars, which is not the same as a single char; keep that in mind. So you'll need a loop to go through the string character-by-character.

Note: Explore the anagrams yourself.

Other details
Since a value associated with a key will always be positive, if you search for a key and don't find it in a hash table, just return 0. That's the special "not found" value for this hash table.

For simplicity, the program assumes that the input file has one word per line (we've written this code for you). The order of the words you output isn't important.

What to Submit:
1. Create a class KeyNode containing following attributes

    a. Key(String)
    b. Value(int)
2. Create a class MyHashTable, with the following attributes.
    - Array to list of KetNode objects
    - Size of HashTable
    - KeysOccupied(initially zero)
3. MyHashTable Contain the following functions
    a. Constructor(hsize) – create the array of size hsize
    b. GetHashTableSize()
    c. GetNumberOfKeys()
    d. HashFunction() – logic to calculate hash value
    e. UpdateKey(key, value)
    f. SearchKey(key)—returns value
    g. Rehash()—in case of hash table is full
4. Now use the MyHashTable to write the above program to count the words occurrence.
5. Write the perfect hash function for your task, justify your answer.