Objectives:
In this lab, students will practice:

1. Implementation of Hash Tables with collision resolution strategies
2. Implementation of huffman encoding

Question 1 (Huffman encoding):
Write a C++ program to perform Huffman encoding on a given string. Your program should include functions to build the Huffman tree, generate Huffman codes for each character, and encode the input string. Provide a brief explanation of how the Huffman encoding algorithm works.

Additional Instructions:
Use a basic struct to represent a node in the Huffman tree.
Utilize a simple approach for building the Huffman tree based on character frequencies.
Display the Huffman codes for each character in the input string.
Example:
For the input string "abracadabra", your program should output the Huffman codes for each character and the
encoded string.
Output:

```
Character 'a' : Code 0
Character 'b' : Code 10
Character 'c' : Code 110
Character 'd' : Code 111
Encoded String: 0101100110111000111010
```

Code:

```
#include<iostream>
#include<vector>
#include<fstream>
using namespace std;
struct node{
	char data;
	int frequency;
	node *left;
	node *right;
	node()
	{
		left=NULL;
		right=NULL;
	}
};
```

```
int main()
{
        string text;
        vector<node*>v;
        ifstream file(“abc.txt”);
        while(getline(file,text))
        {
                cout<<text;
        }
        cout<<endl;
        file.close();
        int len=text.length();
        for(int i=0;i<len;i++)
        {
                bool b=false;
                for(int a=0;a<v.size();a++)
                {
                        if(v[a]->data==text[i])
                        {
                                v[a]->frequency;
                                b=true;
                        }
                }
                if(b==false)
                {
                        node *temp=new node();
                        temp->data=text[i];
                        temp->frequency=1;
                        v.push_back(temp);
                }
        }
        for(int a=0;a<v.size();a++)
        {
                cout<<v[a]->data<<“  ”;
                cout<<v[a]->frequency;
                cout<<endl;
        }
}
```

Question 2: (HashMap with Linear Probing)

Implement a Node struct which represents an item in a hash table.

```
template<class v>
Struct HashItem
{
        int key;
        v value;
        int status;
};
```

Implement a HashItem struct which represents an item in a hash array. It consists of 3 variables; key(int), value(generic), and status. status variables can have 0, 1 or 2. Whereas 0 means empty, 1 means deleted, 2 means occupied. Status variables will be used by 'get' and 'delete' methods of HashMaps implemented in HashMap class. The default value assigned to a HashItem is 0 (empty). Now implement a HashMap class (template based):

template <class v>
class HashMap
{
protected:
HashItem<v>* hashArray;
int capacity;
int currentElements;
virtual int **getNextCandidateIndex**(int key, int i) // The reason why getNextCandidateIndex is pure virtual, because this function will return nextCandidateIndex according to scheme while insertion when collision will occur so the implementation may vary. A collision occurs when a hash function will give an index value which is already occupied. In this case different hashing schemes can be applied.
In Linear probing we simply add 1 in our answer of the hash function to see if the next index is vacant and we will keep checking till a vacant index is found.
void **doubleCapacity()** // A method which doubles the capacity of hashArray and rehashes the existing items. Use getNextCandidateIndex method to resolve collisions.

public:
HashMap(); // Default constructor and should assign capacity 10 to hashArray
HashMap(int const capacity); // parameterized constructor that creates hashArray of size capacity. If capacity is less than 1 return error via assert(capacity>1)

void **insert**(int const key, v const value, HashItem<v>* hashArray);

1. 1. The insert method inserts the value at its appropriate location. Find the first candidate index of the key using: index= key mod capacity
2. To resolve hash collisions, it will use the function getNextCandidateIndex(key, i) to get the next candidate index. If the candidate index also has a collision, then getNextCandidateIndex will be called again with an increment in i. getNextCandidateIndex will be continued to call until we find a valid index. Initially I will be 1.
3. If the loadFactor becomes 0.75, then it will call the doubleCapacity method to double the capacity of the array and rehash the existing items into the new array.

bool **deleteKey**(int const key) const;   //this method deletes the given key. It returns true if the key was found. If the key was not found it returns false. When the key is found, simply set the status of the hashitem containing the key to deleted (value of 1).

v* **get**(int const key) const;   // this method returns a pointer to the corresponding value of the key. If the key is not found, it returns nullptr.

~HashMap(); //Destructor
};