Q.16. Comparison of Hash Tables and Binary Search Tree

The objective of this assignment is to compare the performance of open hash tables, closed hash tables, and binary search trees in implementing search, insert, and delete operations in dynamic sets. You can initially implement using C but the intent is to use C++ and object-oriented principles.

Generation of Keys

Assume that your keys are character strings of length 10 obtained by scanning a text file. Call these as tokens. Define a token as any string that appears between successive occurrences of a forbidden character, where the forbidden set of characters is given by:

$$F = \{comma, period, space\}$$

If a string has less than 10 characters, make it up into a string of exactly 10 characters by including an appropriate number of trailing *'s. On the other hand, if the current string has more than 10 characters, truncate it to have the first ten characters only.

From the individual character strings (from now on called as tokens), generate a positive integer (from now on called as keys) by summing up the ASCII values of the characters in the particular token. Use this integer key in the hashing functions. However, remember that the original token is a character string and this is the one to be stored in the data structure.

<u>Methods to be evaluated</u>

The following four schemes are to be evaluated.

1. Open hashing with multiplication method for hashing and unsorted lists for chaining
2. Open hashing with multiplication method for hashing and sorted lists for chaining
3. Closed hashing with linear probing
4. Binary search trees

Hashing Functions

For the sake of uniformity, use the following hashing functions only. In the following, *m* is the hash table size (that is, the possible hash values are, 0, 1,..., *m* - 1**), and *x* is an integer key.**

1.
   Multiplication Method
   $h(x) = Floor(m*Fraction(k*x))$

   where k $= \frac{\sqrt{(5)}-1}{2}$, the Golden ratio
2.
   Linear Probing
   $h(x, i) = (h(x, 0) + i)$ ***mod*** $m; i = 0,..., m - 1$

Inputs to the Program

The possible inputs to the program are:

- *m*: Hash table size. Several values could be given here and the experiments are to be repeated for all values specified. This input has no significance for binary search trees.
- *n*: The initial number of insertions of distinct strings to be made for setting up a data structure.
- *M*: This is a subset of the set {1, 2, 3, 4} indicating the set of methods to be investigated.
- *I*: This is a decimal number from which a ternary string is to be generated (namely the radix-3 representation of *I*). In this representation, assume that a **0** represents the **search** operation, a **1** represents the **insert** operation, and a **2** represents the **delete** operation.
- A text file, from which the tokens are to be picked up for setting up and experimenting with the data structure

What should the Program Do?

1. For each element of *M* and for each value of *m*, do the following.
2. Scan the given text files and as you scan, insert the first *n* distinct strings scanned into an initially empty data structure.

3. Now scan the rest of the text file token by token, searching for it or inserting it or deleting it, as dictated by the radix-3 representation of *I*. Note that the most significant digit is to be considered first while doing this and you proceed from left to right in the radix-3 representation. For each individual operation, keep track of the number of probes.
4. Compute the average number of probes for a typical successful search, unsuccessful search, insert, and delete.

<u>Solution :</u>

<u>Binary Search Tree</u>

```
#include<bits/stdc++.h>
using namespace std;
struct bst
{
        int data;
        bst *left;
        bst *right;
};
struct bst *create (int data)
{
        struct bst *new_node=(struct bst *)malloc(sizeof(struct bst ));
        new_node->data=data;
        new_node->left=NULL;
        new_node->right=NULL;
        return new_node;
}
struct bst *insert(bst *node,int data)
{
          if(node==NULL)
                return create(data);
          if (node->data>data)
                node->left=insert(node->left,data);
          else if(node->data<data)
                node->right=insert(node->right,data);
        return node;
}
int search(struct bst *node,int d)
{       if(node==NULL)
         return 0;
        if(node->data==d)
         return 1;
        else if (node->data>d)
         return search(node->left,d);
        else
         return search(node->right,d);
}
void inorder(bst *node)
{
        if(node!=NULL)
        {
                inorder(node->left);
                cout<<node->data<<" ";
                inorder(node->right);
        }
}
int main()
{
        bst *root;
        root=NULL;
                fstream file;
        string word,t,q,j;
        t="file.txt";
        int k=0,asc,sum=0,size_of_ht=0,s,r;
        file.open(t.c_str());
        while(file>>word)
        {
                q=q+word+" ";
        }
        int z=strlen(q.c_str());
        int arr[z];
        int p=0;
        for(int i=0;i<=z;i++)
        {
                if(q[i]==' '||q[i]==','||q[i]=='.'||k==10)
                {
                        int a=strlen(j.c_str());
                        if(a==10)
                        {
                                arr[p]=sum;
                                sum=int (q[i]);
                                j=q[i];
                                k=0;
                                p++;
                        }
                        else if(a<10)
                        {
                                int count=10-a;
                                sum=sum+count*42;
                                arr[p]=sum;
                                j="";
                                k=0;
                                sum=0;
                                p++;
                        }
                }
                else
                {               sum=sum+int(q[i]);
                                j=j+q[i];
                                k++;
                }
        }
             int arr1[p-1];
                         for(int i=0;i<p;i++)
                                 {
                                  arr1[i]=arr[i];
                                 }
        for(int i=0;i<p;i++)
         {
                root=insert(root,arr1[i]);
         }

        int x=50;
        int booli=search(root,x);
        if(booli==1)
                cout<<"present"<<endl;
        else
                cout<<"Not present"<<endl;
         inorder(root);


}
```

Hashing Open Addressing

```
#include<bits/stdc++.h>
using namespace std;

insert (int hash[],int data,int r,int p)
{
  if( hash[r]==-1)
    hash[r]=data;
  else
    {
        r=(r+1)%p;
        insert(hash,data,r,p);
         }


}
int check (int hash[],int data,int r,int p)
{   int x=r;
        if( hash[r]==-1)
        return 0;
        else
           {
                if (hash[r]==data)
                return 1;
                else
                {
                        r=(r+1)%p;
                        while(r!=x)
                        {
                          if (hash[r]==data)
                        return 1;
                        r=(r+1)%p;
                                }
                                if(r==x)
                                return 0;
                        }
                }
}

int main ()
{   int r,s,is;
    fstream file;
        string word,t,q,j;
        t="file.txt";
        int k=0,asc,sum=0;
        file.open(t.c_str());
        while(file>>word)
        {
                q=q+word+" ";
        }
        int z=strlen(q.c_str());
        int arr[z];
        int p=0;
        for(int i=0;i<=z;i++)
        {
                if(q[i]==' '||q[i]==','||q[i]=='.'||k==10)
                {
                        int a=strlen(j.c_str());
                        if(a==10)
                        {
                                arr[p]=sum;
                                sum=int (q[i]);
                                j=q[i];
                                k=0;
                                p++;
                        }
                        else if(a<10)
                        {
                                int count=10-a;
                                sum=sum+count*42;
                                arr[p]=sum;
                                j="";
                                k=0;
                                sum=0;
                                p++;
                        }
                }
                else
                {               sum=sum+int(q[i]);
                                j=j+q[i];
                                k++;
                }
        }
                int arr1[p-1];
                        for(int i=0;i<p;i++)
                                arr1[i]=arr[i];
        int hash[p];
        for(int i=0;i<p;i++)
          hash[i]=-1;
        for (int i=0;i<p;i++)
        {
         r=arr1[i]%p;
         insert(hash,arr1[i],r,p);
        }
        string boot;
         cin>>boot;
         int sl=strlen(boot.c_str());
         int sums=0;
         for(int i=0;i<sl;i++)
         { sums=sums+(int) boot[i];
         }
         sums=sums+(10-sl)*42;
         cout<<"sum is="<<sums<<endl;
        r=sums%p;
        is=check(hash,sums,r,p);
        /*for (int i=0;i<p;i++)
          cout<<hash[i]<<"\t";
          cout<<endl;*/
          if(is==1)
            cout<<"present"<<endl;
          else
                cout<<"not present"<<endl;
}
```

Hashing Chaining

```
#include<bits/stdc++.h>
using namespace std;
struct bst
{
        int data;
        bst *left;
        bst *right;
};
struct bst *create (int data)
{
        struct bst *new_node=(struct bst *)malloc(sizeof(struct bst ));
        new_node->data=data;
        new_node->left=NULL;
        new_node->right=NULL;
        return new_node;
}
struct bst *create1 (int data)
```

```
{
            struct bst *new_node=(struct bst *)malloc(sizeof(struct bst ));
            new_node->data=data;
            cout<<new_node->data<<endl;;
            new_node->left=NULL;
            new_node->right=NULL;
            return new_node;
}
struct bst *insert(bst *node,int data)
{
               if(node==NULL)
                        return create(data);
               else
                        node->left=insert(node->left,data);
            return node;
}
struct bst *insert1(bst *node,int data)
{
               if(node==NULL)
                        return create1(data);
               else
                        node->right=insert1(node->right,data);
            return node;
}
int search(struct bst *node,int d)
{
            node=node->right;
            if(node==NULL)
             return 0;
            else if(node->data==d)
                return 1;
            else
             return search(node,d);
// return 0;
}
int main ()
{    int s,r,y;
             bst *root;
             bst *root2;
             bst *root1;
             bst *root3;
             root=NULL;
             int a[7]={50,700,76,85,92,73,101};
             for(int i=0;i<7;i++)
             {
                        root=insert(root,i);
             }
             for(int i=0;i<7;i++)
             {   s=a[i]%7;
                                    root2=root;
                  while(root2->data!=s)
                  { root2=root2->left;
                          }
            root2=insert1(root2,a[i]);

        }
        root3=root;
       while(root3->left!=NULL)
        { cout <<root3->data<<"correspondings elements are ";
          root2=root3;
          while(root2->right!=NULL)
          {
                cout <<root2->data<<" ";
                root2=root2->right;
                }
         root3= root3->left;
         cout <<endl;
        }
        int x=700;
         r=x%7;
         cout<<r<<endl;
         root1=root;
             while(root1->data!=r)
                 root1=root1->left;

         if(search(root1,x)==1)
          cout<<"present";
         else
          cout<<"not present";
}
```

Chaining

```
#include<bits/stdc++.h>
using namespace std;
struct ht
{
        int data;
        ht *left;
        ht *right;
};
struct ht *create (int data)
{
        struct ht *new_node=(struct ht *)malloc(sizeof(struct ht ));
        new_node->data=data;
        new_node->left=NULL;
        new_node->right=NULL;
        return new_node;
}
struct ht *create1 (int data)
{
        struct ht *new_node=(struct ht *)malloc(sizeof(struct ht ));
        new_node->data=data;
        new_node->left=NULL;
        new_node->right=NULL;
        return new_node;
}
struct ht *insert(ht *node,int data)
{
          if(node==NULL)
                return create(data);
          else
                node->left=insert(node->left,data);
        return node;
}
struct ht *insert1(ht *node,int data)
{
          if(node==NULL)
                return create1(data);
          else
                node->right=insert1(node->right,data);
        return node;
}
int search(struct ht *node,int d)
{
        node=node->right;
        if(node==NULL)
         return 0;
        else if(node->data==d)
           return 1;
        else
         return search(node,d);
}

int main()
{
        fstream file;
        string word,t,q,j;
        t="file.txt";
        int k=0,asc,sum=0,size_of_ht=0,s,r;
        file.open(t.c_str());
        while(file>>word)
        {
                q=q+word+" ";
        }
        int z=strlen(q.c_str());
        int arr[z];
        int p=0;
        for(int i=0;i<=z;i++)
        {
                if(q[i]==' '||q[i]==','||q[i]=='.'||k==10)
                {
                        int a=strlen(j.c_str());
                        if(a==10)
                        {
                                arr[p]=sum;
                                sum=int (q[i]);
```

```
                                    j=q[i];
                                    k=0;
                                    p++;
                        }
                        else if(a<10)
                        {
                                    int count=10-a;
                                    sum=sum+count*42;
                                    arr[p]=sum;
                                    j="";
                                    k=0;
                                    sum=0;
                                    p++;
                        }
            }
            else
            {                       sum=sum+int(q[i]);
                                    j=j+q[i];
                                    k++;
            }
}
        int arr1[p-1];
                        for(int i=0;i<p;i++)
                                    {
                                     arr1[i]=arr[i];
                                    }

 ht *root;
 ht *root2;
 ht *root1;
 root=NULL;
 for(int i=0;i<p;i++)
 {
            root=insert(root,i);
 }
 float g_ratio=(sqrt(5)-1)/2;
 for(int i=0;i<p;i++)
 {          float g=(g_ratio*arr1[i]);
                        g=g-floor(g);
                        s=floor(p*g);
                        root2=root;
      while(root2->data!=s)
      { root2=root2->left;
              }
    root2=insert1(root2,arr1[i]);
 }
 /*while(root->left!=NULL)
 { cout <<root->data<<"correspondings elements are ";
   root2=root;
   while(root2->right!=NULL)
   {
            cout <<root2->data<<" ";
            root2=root2->right;
```

```
                  }
            root= root->left;
            cout <<endl;
           }
           cout<<root->data<<endl;*/
           string boot;
           cin>>boot;
           int sl=strlen(boot.c_str());
           int sums=0;
           for(int i=0;i<sl;i++)
           { sums=sums+(int) boot[i];
           }
           sums=sums+(10-sl)*42;
           cout<<"sum is="<<sums<<endl;
            r=p*((g_ratio*sums)-floor(g_ratio*sums));
            root1=root;
                while(root1->data!=r)
                      root1=root1->left;
            if(search(root1,sums)==1)
             cout<<"present";
            else
             cout<<"not present";
}
```

Linear Probing

```
#include<bits/stdc++.h>
using namespace std;
int counter;

insert (int hash[],int data,int r)
{
  if( hash[r]==-1)
     hash[r]=data;
  else
     {
            r=(r+1)%10;
            counter++;
            insert(hash,data,r);
             }

}

int main ()
{   int i,r,s,j;
    int a[7]={50,700,76,85,92,73,101};
            int hash[10]={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
            for (i=0;i<7;i++)
            {
             r=a[i]%10;
             insert(hash,a[i],r);
            }
```

```
                    cout<<counter<<endl;
}
```

Double Hashing

```
#include<bits/stdc++.h>
using namespace std;
int counter=0;

insert (int hash[],int data,int k)
{ int r=((data%10)+k*(7-(data%7)))%10;
  if( hash[r]==-1)
     hash[r]=data;
  else
     {
          counter++;
          k++;
          insert(hash,data,k);
           }


}int main ()
{   int i,k;
    int a[7]={50,700,76,85,92,73,101};
          int hash[10]={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
          for (i=0;i<7;i++)
          {
           k=0;
           insert(hash,a[i],k);
          }
          cout<<counter<<endl;
}
```

Quadratic Probing

```
#include<bits/stdc++.h>
using namespace std;
int counter;

insert (int hash[],int data,int k)
{ int r=((data%10)+k*k)%10;
  if( hash[r]==-1)
     hash[r]=data;
  else
     {
          counter++;
          k++;
          insert(hash,data,k);
           }


}int main ()
```

```
{   int i,s,j,k;
    int a[7]={50,700,76,85,92,73,101};
            int hash[10]={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
            for (i=0;i<7;i++)
            {
             k=0;
             insert(hash,a[i],k);
            }
                        cout<<counter<<endl;
}
```

Q.17. Hash header file

Create a header file random.h (Header file for random number class) write program to demonstrate the implementation for random number class, also design a header file UniformRandom.h that uses as random number class using standard library.

Solution:

random.h

```
#include <ctime>

#include <cstdlib>

int getRandomNumber()

{

            srand(time(0));

            return rand() % 100+1;

}
```

RandomNumber.cpp

```
#iinclude <iostream>

#include "random.h"

using namespace std;

int main(){

            int theNumber  =  getRandomNumber();
```

```
        cout << theNumber << endl;

        system("pause");

        return 0;

}
```

Q.18. Computation on Binary Tree

This assignment is an example of the kind of computation one does on binary trees. You have to use C++ for the development of a program that does the following. The program takes commands from users, as usual; the following commands are supported:

- exit: quits the program
- newtree: asks the user to enter an *inorder sequence* and a *preorder sequence* (of strings) of a binary tree. As we have discussed in class, these two orders are sufficient to uniquely reconstruct the tree. Once the user has entered the two sequences, the tree is the "current tree", and the old tree (from a previous typing of newtree command) is discarded.
- preorder: prints the current tree in preorder
- inorder: prints the current tree in inorder
- postorder: prints the current tree in postorder
- levelorder: prints the current tree in levelorder
- ver: prints the current tree vertically in a "pretty format."
- hor: prints the current tree horizontally in a "pretty format"
- sym: prints the current tree symmetrically in a "pretty format

Solution :

```
#include <iostream>
#include <vector>
#include <unordered_map>
using namespace std;

// Data structure to store a Binary Tree node
struct Node
{
   int key;
   Node *left, *right;
};

// Function to create a new binary tree node having given key
Node* newNode(int key)
{
   Node* node = new Node;
   node->key = key;
   node->left = node->right = nullptr;

   return node;
}

// Recursive function to perform inorder traversal of a binary tree
void inorderTraversal(Node* root)
{
   if (root == nullptr)
      return;

   inorderTraversal(root->left);
   cout << root->key << ' ';
   inorderTraversal(root->right);
}
```

```
// Recursive function to perform postorder traversal of a binary tree
void preorderTraversal(Node* root)
{
  if (root == nullptr)
    return;

  cout << root->key << ' ';
  preorderTraversal(root->left);
  preorderTraversal(root->right);
}

// Recursive function to perform post-order traversal of the tree
void postorder(Node *root)
{
  // if the current node is empty
  if (root == nullptr)
    return;

  // Traverse the left subtree
  postorder(root->left);

  // Traverse the right subtree
  postorder(root->right);

  // Display the data part of the root (or current node)
  cout << root->data << " ";
}


// Function to print all nodes of a given level from left to right
bool printLevel(Node* root, int level)
{
  if (root == nullptr)
    return false;

  if (level == 1)
  {
    cout << root->key << " ";

    // return true if at-least one node is present at given level
    return true;
  }

  bool left = printLevel(root->left, level - 1);
  bool right = printLevel(root->right, level - 1);
```

```
    return left || right;
}

// Function to print level order traversal of given binary tree
void levelOrderTraversal(Node* root)
{
    // start from level 1 -- till height of the tree
    int level = 1;

    // run till printLevel() returns false
    while (printLevel(root, level))
        level++;
}




// Recursive function to construct a binary tree from given
// inorder and preorder sequence
Node* construct(int start, int end, vector<int> const &preorder,
                int &pIndex, unordered_map<int,int> &map)
{
    // base case
    if (start > end)
        return nullptr;

    // The next element in preorder[] will be the root node of subtree
    // formed by inorder[start, end]
    Node *root = newNode(preorder[pIndex++]);

    // get the index of root node in inorder[] to determine the
    // boundary of left and right subtree
    int index = map[root->key];

    // recursively construct the left subtree
    root->left = construct(start, index - 1, preorder, pIndex, map);

    // recursively construct the right subtree
    root->right = construct(index + 1, end, preorder, pIndex, map);

    // return current node
```

```
  return root;
}

// Construct a binary tree from inorder and preorder traversals
// This function assumes that the input is valid
// i.e. given inorder and preorder sequence forms a binary tree
Node* construct(vector<int> const &inorder, vector<int> const &preorder)
{
  // get number of nodes in the tree
  int n = inorder.size();

  // create a map to efficiently find the index of any element in
  // given inorder sequence
  unordered_map<int,int> map;
  for (int i = 0; i < n; i++)
    map[inorder[i]] = i;

  // pIndex stores index of next unprocessed node in preorder sequence
  // start with root node (present at 0'th index)
  int pIndex = 0;

  return construct(0, n - 1, preorder, pIndex, map);
}

int main()
{
  vector<int> inorder = { };
  vector<int> preorder = { };

 cout << "Enter commands:";
 cin >> input_string;

if(input_string == "newtree" )
{
```

```
cout << "Enter inorder:";
for(int i = 0; i < n ; i ++){
        cin >> input_no;
        inorder.insert(inorder.end(), input_no )
}

cout << "Enter preorder:";
for(int i = 0; i < n ; i ++){
        cin >> input_no;
        preorder.insert(preorder.end(), input_no )
}

  Node* root = construct(inorder, preorder);

}
// traverse the constructed tree
else if(input_string == “preorder” ){
   cout << "\nPreorder : "; preorderTraversal(root);

}
else if(input_string == “inorder” ){
   cout << "Inorder  : "; inorderTraversal(root);

}
else if(input_string == “postorder” ){
    postorder(root);
}
else if(input_string == “levelorder” ){
    levelOrderTraversal(root);
}
else if(input_string == “hor” ){
    levelOrderTraversal(root);
}
else if(input_string == “exit” ){
   exit();
```

```
}
   return 0;
}
```