Name: Prashant Kumar

Roll No.: 222CS2097

Semester: 1st(IS)

Q.10. 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.

Methods to be evaluated

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.

Code:

```
1. //Hash Header File.......hash.h//
2. #include<iostream>
3. using namespace std;
4. #include<string>

5. template <typename v>
6. class mapNode{
7. public:
8. string key;
9. v value;
10.     mapNode* next;
11.     mapNode(string key,v value){
12.     this->key=key;
13.     this->value=value;
14.     next = NULL;
15.     }
16.     ~mapNode(){
17.     delete next;
18.     }
19.     };

20.     template <typename v>
21.     class ourmap{
22.     mapNode<v>** buckets;
23.     int count;
24.     int numbucket;
25.     public:
26.     ourmap(){
27.     count=0;
28.     numbucket=5;
```

```
29.     buckets=new mapNode<v>*[numbucket];
30.     for(int i=0;i<numbucket;i++){
31.     buckets[i]=NULL;
32.     }
33.     }
34.     ~ourmap(){
35.     for(int i=0;i<numbucket;i++){
36.     delete buckets[i];
37.     }
38.     delete []buckets;
39.     }
40.     private:
41.     int getBucketIndex(string key){
42.     int hashcode=0;
43.     int curr_co=1;
44.     for(int i=key.length()-1;i>=0;i--){
45.     hashcode+=key[i]*curr_co;
46.     hashcode=hashcode%numbucket;
47.     curr_co*=37;
48.     curr_co=curr_co%numbucket;
49.     }
50.     return hashcode%numbucket;
51.     }
52.     public:
53.     void insert(string key,v value){
54.     int bucketindex=getBucketIndex(key);
55.     mapNode<v>* head=buckets[bucketindex];
56.     while(head!=NULL){
57.     if(head->key==key){
58.     head->value=value;
59.     return;
60.     }
61.     head=head->next;
```

```
62.     }
63.     head=buckets[bucketindex];
64.     mapNode<v>* node=new mapNode<v>(key,value);
65.     node->next=head;
66.     buckets[bucketindex]=node;
67.     count++;
68.     }

69.     int size(){
70.     return count;
71.     }
72.     v getValue(string key) {
73.     int bucketIndex = getBucketIndex(key);
74.     mapNode<v>* head = buckets[bucketIndex];
75.     while (head != NULL) {
76.     if (head->key == key) {
77.     return head->value;
78.     }
79.     head = head->next;
80.     }
81.     return 0;
82.     }
83.     v remove (string key){
84.     int bucketindex=getBucketIndex(key);
85.     mapNode<int>* prev=NULL;
86.     mapNode<int>* head=buckets[bucketindex];
87.     while(head!=NULL){
88.     if(head->key==key){
89.     if(prev==NULL){
90.     buckets[bucketindex]=head->next;
91.     }
92.     else{
93.     prev->next=head->next;
```

```
94.      }
95.      v value=head->value;
96.      head->next=NULL;
97.      delete head;
98.      count--;
99.      return value;
100.     }
101.     prev = head;
102.     head = head->next;
103.     }
104.     return 0;
105.     }
106.     };
```

Result:

```
{abc0:1}
{abc1:2}
{abc2:3}
{abc3:4}
{abc4:5}
{abc5:6}
{abc6:7}
{abc7:8}
{abc8:9}
{abc9:10}
10
CURRENT SIZE OF MAP IS:12
GETVALUE AT A GIVEN KEY:8
REMOVED VALUE :8
AFTER REMOVE abc7 size is:11
```

Q.11. 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.

Code:

```
1. //Random.h
2. #include <ctime>
3. #include <cstdlib>
4. int getRandomNumber()
5. {
6. srand(time(0));
7. return rand() % 1000+1;
8. }
```

```
1. //RandomNumber.cpp
2. #include <iostream>
3. #include "Random.h"
4. using namespace std;
5. int main()
6. {
7. int theNumberOne = getRandomNumber();
8. cout << theNumberOne << endl;
9. system("pause");
10.     return 0;
11.     }
```

Result:

81

Press any key to continue . . .

Q.12. 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"

Code:

```
#include <iostream>
#include<queue>
using namespace std;
template <typename T>
class BinaryTreeNode{
 public:
 T data;
 BinaryTreeNode<T>* left;
 BinaryTreeNode<T>* right;
};
//To initialise a BinaryTreeNode
BinaryTreeNode<char>* newNode(char data)
```

```
13.
14.{
15. BinaryTreeNode<char>* Node = new BinaryTreeNode<char>();
16. Node->data = data;
17. Node->left = NULL;
18. Node->right = NULL;
19. return (Node);
20.}
21.int search(char arr[], int strt, int end, char value) {
22. int i;
23. for (i = strt; i <= end; i++)
24. {
25. if (arr[i] == value)
26. return i;
27. }
28. return i;
29.}
30.//to build a tree
31.BinaryTreeNode<char>* createTree(char in[], char pre[], int
   inStrt, int inEnd)
32.{
33. static int preIndex = 0;
34. if (inStrt > inEnd)
35. return NULL;
36. BinaryTreeNode<char>* tNode = newNode(pre[preIndex++]);
37. if (inStrt == inEnd)
38. return tNode;
39. int inIndex = search(in, inStrt, inEnd, tNode->data);
40. tNode->left = createTree(in, pre, inStrt, inIndex - 1);
41. tNode->right = createTree(in, pre, inIndex + 1, inEnd);
```

```
42.  return tNode;
43. }
44.
45. //Inorder Traversal
46. void printInorder(BinaryTreeNode<char>* node) {
47.  if (node == NULL)
48.  return;
49.  printInorder(node->left);
50.  cout<<node->data<<" ";
51.  printInorder(node->right);
52. }
53. //Preorder Traversal
54. void printPreorder(BinaryTreeNode<char> *root) {
55.  if(root==NULL) {
56.  return;
57.  }
58.  cout<<root->data<<" ";
59.  printPreorder(root->left);
60.  printPreorder(root->right);
61. }
62. //Postorder Traversal
63. void printPostorder(BinaryTreeNode<char> *root) {
64.  if(root==NULL) {
65.  return;
66.  }
67.  printPostorder(root->left);
68.  printPostorder(root->right);
69.  cout<<root->data<<" ";
70. }
71. //Levelwise Traversal
```

```
72. void printLevelWise(BinaryTreeNode<char> *root) {
73.  if(root == NULL){
74.  return;
75.  }
76.
77.  queue <BinaryTreeNode<char>*> pendingNode;
78.  pendingNode.push(root);
79.
80.  while(pendingNode.size() != 0){
81.
82.  int size1 = pendingNode.size();
83.  for(int i=0;i<size1;i++){
84.  BinaryTreeNode<char> *front = pendingNode.front();
85.  pendingNode.pop();
86.  cout << front -> data << " ";
87.  if(front->left != NULL){
88.  pendingNode.push(front->left);
89.  }
90.  if(front -> right != NULL){
91.  pendingNode.push(front->right);
92.  }
93.  }
94.  cout << endl;
95.
96.  }
97.
98. }
99. void findMinMax(BinaryTreeNode<char> *node, int *min, int *max,
    int hd) {
100.    if (node == NULL)
```

```
101.  return;
102.   if (hd < *min) *min = hd;
103.   else if (hd > *max) *max = hd;
104.   findMinMax(node->left, min, max, hd-1);
105.   findMinMax(node->right, min, max, hd+1);
106.  }
107.  void printVerticalLine(BinaryTreeNode<char> *node, int
   line_no, int hd) {
108.   if (node == NULL)
109.  return;
110.   if (hd == line_no)
111.   cout<<node->data <<" ";
112.   printVerticalLine(node->left, line_no, hd-1);
113.   printVerticalLine(node->right, line_no, hd+1);
114.  }
115.
116.  //Vertical Order Traversal
117.  void verticalOrder(BinaryTreeNode<char> *root)
118.  {
119.   int min = 0, max = 0;
120.   findMinMax(root, &min, &max, 0);
121.   for (int line_no = min; line_no <= max; line_no++) {
122.   printVerticalLine(root, line_no, 0);
123.   cout<<endl;
124.   }
125.  }
126.  //To check Whether Tree is symmetric or not
127.  bool isSymmetricHelper(BinaryTreeNode<char>* left_part,
   BinaryTreeNode<char>*
128.  right_part) {
```

```
129.   if(left_part == NULL || right_part == NULL) {
130.   return left_part == right_part;
131.   }
132.   if(left_part->data != right_part->data) {
133.   return false;
134.   }
135.   if(isSymmetricHelper(left_part->left, right_part->right) &&
136.  isSymmetricHelper(left_part->right, right_part->left)) {
137.   return true;
138.   }
139.   return false;
140.  }
141.  bool isSymmetric(BinaryTreeNode<char>* root) {
142.   if(root == NULL)
143.  return true;
144.   return isSymmetricHelper(root, root);
145.  }
146.  int main()
147.  {
148.  //Inorder and Preorder char sequence to create tree
149.   char in[] = { 'D', 'B', 'E', 'A', 'F', 'C' };
150.   char pre[] = { 'A', 'B', 'D', 'E', 'C', 'F' };
151.
152.   int len = sizeof(in) / sizeof(in[0]);
153.   BinaryTreeNode<char>* root = createTree(in, pre, 0, len - 1);
154.   int choice=0;
155.
156.   do
157.   {
158.   cout<<"\n!!!Computation on Binary Tree!!! ";
```

```
159.    cout<<"\nPress 0 to Exit ";
160.    cout<<"\nPress 1 to print Preorder ";
161.    cout<<"\nPress 2 to print Inorder ";
162.    cout<<"\nPress 3 to print PostOrder ";
163.    cout<<"\nPress 4 to print Level order ";
164.    cout<<"\nPress 5 to print Vertical order ";
165.    cout<<"\nPress 6 to check Symmetric ";
166.    cout<<"\nEnter the choice: ";
167.    cin>>choice;
168.
169.    switch(choice)
170.    {
171.    case 0:
172.    cout<<"\nExit from program" << endl;
173.    break;
174.    case 1:
175.    cout<<"PreOrder Traversal: ";
176.    printPreorder(root);
177.    cout<<endl;
178.    break;
179.    case 2:
180.    cout<<endl<<"InOrder Traversal: ";
181.    printInorder(root);
182.    cout<<endl;
183.    break;
184.    case 3:
185.    cout<<endl<<"PostOrder Traversal: ";
186.    printPostorder(root);
187.    cout<<endl;
188.    break;
```

```
189.    case 4:
190.
191.    cout<<endl<<"Level Order Traversal: "<<endl;
192.    printLevelWise(root);
193.    break;
194.    case 5:
195.    cout<<endl<<"Vertical Order Traversal: "<<endl;
196.    verticalOrder(root);
197.    break;
198.    case 6:
199.    cout<<endl<<"Check is tree symmetric: ";
200.    if(isSymmetric(root)) {
201.    cout<<"Symmetric";
202.    } else {
203.    cout<<"Not symmetric";
204.    }
205.    cout<<endl;
206.    break;
207.    default:
208.    cout<<"\nInvalid Choice!";
209.    break;
210.    }
211.
212.    } while(choice!=0);
213.    return 0;
214.  }
```

Result:

```
!!!Computation on Binary Tree!!!
Press 0 to Exit
Press 1 to print Preorder
Press 2 to print Inorder
Press 3 to print PostOrder
Press 4 to print Level order
Press 5 to print Vertical order
Press 6 to check Symmetric
Enter the choice: 1
PreOrder Traversal: A B D E C F

!!!Computation on Binary Tree!!!
Press 0 to Exit
Press 1 to print Preorder
Press 2 to print Inorder
Press 3 to print PostOrder
Press 4 to print Level order
Press 5 to print Vertical order
Press 6 to check Symmetric
Enter the choice: 2

InOrder Traversal: D B E A F C

!!!Computation on Binary Tree!!!
Press 0 to Exit
Press 1 to print Preorder
Press 2 to print Inorder
Press 3 to print PostOrder
Press 4 to print Level order
Press 5 to print Vertical order
Press 6 to check Symmetric
Enter the choice: 3

PostOrder Traversal: D E B F C A
```

```
!!!Computation on Binary Tree!!!
Press 0 to Exit
Press 1 to print Preorder
Press 2 to print Inorder
Press 3 to print PostOrder
Press 4 to print Level order
Press 5 to print Vertical order
Press 6 to check Symmetric
Enter the choice: 4

Level Order Traversal:
A
B C
D E F

!!!Computation on Binary Tree!!!
Press 0 to Exit
Press 1 to print Preorder
Press 2 to print Inorder
Press 3 to print PostOrder
Press 4 to print Level order
Press 5 to print Vertical order
Press 6 to check Symmetric
Enter the choice: 5

Vertical Order Traversal:
D
B
A E F
C

!!!Computation on Binary Tree!!!
Press 0 to Exit
Press 1 to print Preorder
Press 2 to print Inorder
Press 3 to print PostOrder
Press 4 to print Level order
Press 5 to print Vertical order
Press 6 to check Symmetric
Enter the choice: 6

Check is tree symmetric: Not symmetric
```