COMP 202 FALL 2020 HOMEWORK #6 PART 1 & 2

COMP-202 FALL 2020

Homework 6

I have completed this assignment individually, without support from anyone else. I hereby accept that only the below listed sources are approved to be used during this assignment:

(i) Course book

(ii) All material that is made available to me by professor (e.g. via Blackboard for this course website, email from professor / TA)

(iii) Notes taken by me during lecture

I have not used, accessed or taken any unpermitted information from any other source. Hence all effort belongs to me.

Sinan Cem Erdoğan

68912

```
int hashFirst(key)
   return key.id % (10)

int hashSecond(Student key)

    string = key.name;

    charAtLast = string.charAt(string.length() - 1)
    charAtFirst = string.charAt(0);

    return (chatAtFirst + chatAtLast + key.id) * key.gpa) % 10
```

Hash Functions Comparison

I used 15 sample input for two functions.

Output for hashFirst:

```
[0] -> NULL
[1] -> NULL
[2] -> Key - Value -> Key - Value ->
[3] -> Key - Value ->
[4] -> Key - Value -> Key - Value ->
[5] -> Key - Value -> Key - Value -> Key - Value -> Key - Value -> Key - Value ->
[6] -> Key - Value -> Key - Value -> Key - Value ->
[7] -> NULL
[8] -> Key - Value -> Key - Value ->
[9] -> NULL
```

Output for hashSecond:

```
[0] -> Key - Value -> Key - Value ->
[1] -> NULL
[2] -> Key - Value ->
[3] -> Key - Value ->
[4] -> Key - Value -> Key - Value ->
[5] -> Key - Value ->
[6] -> Key - Value -> Key - Value -> Key - Value ->
[7] -> Key - Value -> Key - Value -> Key - Value ->
[8] -> Key - Value ->
[9] -> Key - Value ->
```

It is clearly seen that hashSecond function is better to distribute the hashNode to the bucketArray indexes. Thus, the number of collisoin is smaller than the first one.

Number of collision in first: 9
Number of collision in first: 6

In the function hashFirst we use commonly used method of mod of array size. It is not that much good way to distribute. Thus we have a good number of collision.

In the function hashSecond we use different information about students to distribute the students in a more uniform way. Probability that there is relation between a student's name, id and, gpa is low. So we use the 3 information about student to get a more uniform distribution.

```
int size()
 return size
```
Time Complexiy of returning the size is O(1).
```
boolean isEmpty()
 return size() == 0
```
Time Complexiy of checkhing if the BST is empty is O(1).
```
int hashFirst(key)
 return key.id
```
Time Complexiy of returning the Student's id is O(1).

```
int hashSecond(key)
 string = key.name
 chatAtLast = string.charAt(string.length() - 1)
 chatAtFirst = string.charAt(0)
 return (chatAtFirst + chatAtLast + key.id) * key.gpa)
```
Time Complexiy of returning and calculating the mixed value composed of a Student's informations is O(1).
```
int getBucketIndex(key)

 if (hashing = 0)
  return hashFirst(key) % numBuckets
 else if (hashing == 1)
  return hashSecond(key) % numBuckets
 else
  return 0
```

Time Complexiy of calculating and returning the index is O(1).
```
Advisor remove(key)
 bst.delete(key)

 index = getBucketIndex(key)
 currentNode = bucketArray[index]

 if (currentNode != null & currentNode.key = key)
  bucketArray[index] = currentNode.next
  size--
  return currentNode.value

  previousNode = null;
 while (currentNode != null && currentNode.key != key)
  previousNode = currentNode
  currentNode = currentNode.next

 if(currentNode != null)
  previousNode.next = currentNode.next
  size--
  return currentNode.value

 return null
```
The expected runnig time of removing an element from a hashMap is O(1).In the worst case, it is O(n). Worst case occurs when all the keys insterted to the map collides. In this case searhing for the key takes O(n) time.

```
Advisor get(key)
 index = getBucketIndex(key)

 for (node = bucketArray[index] node != null node = node.next)
  if(node.key == key)
   return node.value

 return null
```

The expected runnig time of getting an element from a hashMap is O(1).In the worst case, it is O(n). Worst case occurs when all the keys insterted to the map collides. In this case searhing for the key takes O(n) time.

```
boolean isPresent(key)

 index = getBucketIndex(key)
 for(node = bucketArray[index] node != null node = node.next)
  if(node.key.id = key.id)
   return true



  return false;
```

The expected runnig time of checking if an element present in hashMap is O(1).In the worst case, it is O(n). Worst case occurs when all the keys insterted to the map collides. In this case searhing for the key takes O(n) time.

```
void add(key,value)
 bst.add(key)

 if(!isPresent(key))
 size++
 hashNode = new HashNode(key,value)
 index = getBucketIndex(key)

 if(bucketArray[index] == null)
  bucketArray[index] = hashNode

 else
  node = bucketArray[index]
  while(node.next != null)
   node = node.next

  node.next = hashNode
 return
```

The expected runnig time of adding an element to hashMap is O(1).In the worst case, it is O(n). Worst case occurs when all the keys insterted to the map collides. In this case searhing for the key takes O(n) time.

```
void printSorted()
 bst.printBst()
```

Since we traverse over the tree to convert it to string, time complexity for printing the sorted hashMap is O(n).

```
void add(data)
 root = recursiveAdd(root, data)

TreeNode recursiveAdd(TreeNode root, Student data)
 if(root = null)
  root = new TreeNode(data)
  return root

 if(root.getData().id > data.id)
  root.setLeft(recursiveAdd(root.getLeft(),data))

 else if (root.getData().id < data.id)
  root.setRight(recursiveAdd(root.getRight(),data))

 return root
```

For adding a Student to the BST, we need to first search for the location of the new Student. Search in BST in the worst case is O(n). In general it is O(h). (h: Height of the BST)

So adding in the worst case is O(n). In general it is O(h).

```
 TreeNode delete(data)
  root = recursiveDelete(root,data)
  return root

TreeNode recursiveDelete(root, data)
 if (root == null)
  return root

 if (data.id < root.data.id)
  root.left = recursiveDelete(root.left, data);
 else if (data.id > root.data.id)
  root.right = recursiveDelete(root.right, data);

 else
  if (root.left = null)
   return root.right
  else if (root.right = null)
   return root.left
   root.data = findMin(root.right)
   root.right = recursiveDelete(root.right, root.data)

 return root
```

For deleting a Student from the BST, we need to first search for the location of the Student. Search in BST in the worst case is O(n). In general it is O(h). (h: Height of the BST)

So adding in the worst case is O(n). In general it is O(h).

```
Student findMin(root)
 min = root.data
 while (root.left != null)
  min = root.left.data
  root = root.left
 return min
```

For finding a Student with min key, we need to first search for the location of the Student. Search in BST in the worst case is O(n). In general it is O(h). (h: Height of the BST)

So finding min key in the worst case is O(n). In general it is O(h).

```
String toStringInorder(root)
 result = "";
 if (root = null)
  return result
 result += toStringInorder(root.getLeft())
 result += root.getData() + "\n"
 result += toStringInorder(root.getRight())
 return result

void printBst()
 print(toStringInorder(root))
```

We traverse over the tree to get the information. Time complexity converting BST to string is O(n).

Space Complexity for this assignment:

All functions we call , use O(1) space of memory. They have just a few pointers to do their action.

However, we create a n-sized HashMap and a n-sized BST. We allocate space from the memory for each. So the space complexity is O(n) + O(n) which is just O(n).