COL 106
ASSIGNMENT 3

NAME: RUDRESH RAJ VERMA

ENTRY NO.: 2018CS10376

Pair Class :

I have created this class to store the key of <String, String> form. I have also implemented a method called "toString()" which converts my key to a string containing first name and last name of the key and it returns the combination along with a space bar between the two.

Student class:

I have created a student class of type String which implements the Student_ interface of type String. I have implemented all the methods of the Student_ interface and also I have overridden a method "toString()" in it which returns the combination of the first name and last name of the student along with a space bar between the two.

Double Hashing:

I have created a class called **DoubleHashMap<K, T>** which is a generic class of type <K,T> where K is the key type and T is the value type. This class implements the MyHashMap_ interface which is also generic.
Since the key was generic, I override the toString() function in my defined class of Pair. And I write key.toString() whenever I want to compare my key to the string representation of a student object "obj"(obj.toString()).
I created an array of type T in this class.
the following are the implemented methods in it:

1. **insert():** First I calculated the index where I want to store the data. If collision occurs, I use the next hash function which is actually (first_hash_function + some_integer_i* second_hash_function) until I find an empty spot(the integer_i will keep on increasing). The time complexity for the above operation is O(1) but in worst case can be O(tableSize), where 'tableSize' is the size of the array created. Also the table size must be greater than twice the number

of elements that are to be inserted in the array and the value of 'tableSize' must be prime.

This function returns an integer which is actually 'how many times the index is calculated for inserting the value corresponding to the key'. There is also a situation where the pair(key,value) we want to insert already matches with a value in the table. In that case, we can't just insert the duplicate again in the table, instead we will return -1.

2. **contains():** This function works the same as insert but this is of type boolean. I just keep searching until I find or I go through all of the table. Since tablesize is a prime.So it is guaranteed that we will go through all spots in table once.

When we moved through the complete table but could not find the value related to the given key, then we will return 'true' else 'false'.
Time complexity : O(1) , Worst Case : O(tablesize).

3. **update():** Works same as contains but this returns an integer. If I find such an index which contains the value related to the key, I just put the updated object's data in the found node's data field using the key provided to find the corresponding node and return the number of times an index is calculated to find the value to be updated else I return -1.

Time complexity: O(1) ; Worst Case : O(tablesize).

4. **address():** address function also works same as contains but this is of type String. It just returns the index of the final calculated hash function output if the table contains the value related to the key else it returns null.

Time complexity: O(1) ; Worst Case: O(n).

5. **get():** this function returns the value of type T. It works the same way as other functions. First find the index. If the place is null or filled with the value not related to the key, find the next index until the concerned value is found. I can calculate the index only the number of times equal to the table size as it will cover all the indices. If under this number of counts, I found the value, then return it else return null.

6. **delete():** this function returns an integer. It works the same way as other functions. First find the index. If the place is null or filled with the value not related to the key, find the next index until the concerned value is found. I can calculate the index only the number of times equal to the table size as it will cover all the indices. If under this number of counts, I found the value, then delete it and return the number equals to the number of times the index is calculated to reach this value else return -1.

Binary Search Tree

in order to make things convenient for me, I separately implemented a generic Binary Search Tree<K,T> where K is the type class for key and T for value.

Note: What I am doing is that, I directly compare a key.toString() form with the value.toString() form. In more detail, I am creating a string comprising first name and last name with a space bar between them for both key and value. And if the two Strings are equal, then the value must correspond to that key only.In order to compare the keys, I implemented the toString() function in my Pair implementation.

1. insert(): This function adds/inserts a new node maintaining the BST order. I implemented this function iteratively, i.e., by going through the nodes that come in the path of my binary search. This function returns the total nodes touched while inserting the given element. This function returns the value of nodes touched if the node is inserted. But if there is already a node present with the given key, then we cant again insert the node with the same key, so we will return -1.
Adding a node on average case takes O(log(n)) time.
Because at max we will be traveling the tree's height. However the worst case can be O(n) since if we keep adding elements in strictly increasing order our tree will actually be a chain and it will take O(n) time to travel till leaf.

2. contains(): This function again is iterative. I have defined

toString() in the Pair class, so I compare key.toString() and value.toString() (where value is the value of the node in which we are searching the key). I return 0 if the key to compare and key I carry match, <0 if my key is smaller than the key at the node or >0 for vice-versa. So in this way I can locate the exact node I'm looking for. Here again, I will return true if I find the value related to the key else false.
Again the average complexity can be O(log(n)) and the worst case can be O(n) due the same above discussed reasons.

3. delete(): In delete I have made three cases:

Case 1: if the node I want to delete has no child.
Case 2: if it has one child.
Case 3: if it has two children.

Case 1:
if the found node is root, then make the root null.
if it is not a root, then make it null.

Case 2:
If it is the root of the tree, then make its child the root.
If it is not the root, then directly attach its child to its parent, i.e.,
parent.left = deleteNode.left (if there is left child)
parent.right = deleteNode.right (if there is right child)

Case 3:
In this case, we have to find the In Order Successor of the **deleteNode(**deleteNode is the name of the node which I want to delete**)**.

What is In Order Successor?
Ans: The **In Order Successor** is the node which is just larger than the deleteNode.
Or we can simply say that the In Order Successor is the node that appears just after the node which is being deleted if the BST nodes are

written in ascending order.

getSuccessor():

I have written a method for finding the In Order Successor.
First I will move to the first right of the deleteNode and then keep on proceeding to left node line until I find a null node. Then the node just before the null node is my In Order Successor or simply Successor node.
After finding the successor node, I will assign the right node of the Successor to its directly linked parent node, so that the Successor gets detached from its parent. Then, I will just swap the deleteNode with the Successor node. So the Successor will take the place of deleteNode without disturbing the order of BST.

How will I do the swapping?

Ans: First, I have found the successor node. Then I will make the right child of deleteNode the right child of the successor node.
Then there are two cases:
First, if the deleteNode is attached to the right of the parent node of the deleteNode. Then I will make Successor the right child of the parent node. And then the left child of the Successor will be attached to the left of the Successor. The same is the procedure for the other case where deleteNode is attached to the left of the parent node.
Now this function returns the number of nodes touched while performing the delete operation.

How do I count the touched nodes?

Ans: First I will start from the root. I will keep on iterating on the nodes and increasing the number of count until I find the deleteNode. After that, when I move to the direct right child of the deleteNode, then I again increase the count by one. I will further increase the count while going to the successor. After that I will again increase the count when the node which I m transferring from the Successor to its parent is not null. Then I will return this final count.
Average time complexity to find 'In Order Successor' in O(log(n)) and again the worst case can be O(n).
Therefore average time complexity for delete is O(log(n)) and on similar basis, worst time complexity is O(n).

4. address(): I just used a string to track my movements and appended "L" and "R" accordingly and then returned the string.
If we could find the element, then we will return its address else return null.
Average time complexity is O(log(n)) and on similar basis worst is O(n).

7. get(): this function returns the value related to the key passed as the parameter. The logic is same as other functions. It will first find the node whose value I want and then return it. If the demanding node is not found in the tree, then return null.
8. update(): This function returns the integer value. This function first finds the concerned node with the same logic used in other functions, if found update its value with the value passes as the parameter along with key and return the nodes touched, else return -1.

Separate Chaining:

I have created a generic class named **ChainHashMap<K,T>** which implements the generic interface **MyHashMap_<K,T>.**
I then created an array of BSTs here and then initialized all its elements as a new empty BST.
Due to the creation of BST, it made my work easy here.

1. insert(): Just calculate the index using hash function and insert at BST_table[index]. Average time Complexity is O(1 + log(n)). Worst case is O(n + n) = O(n).

2. delete(): Same as above, just calculate the index using hash function and delete from the BST_table[index]. Average time Complexity is O(1 + log(n)). Worst case is O(n + n) = O(n).

3. contains(): Just calculate the index using hash function, then just check if the BST BST_table[index] contains this key or not. Average

time complexity O(1 + log(n)). Worst case time complexity is O(n + n) = O(n).

4. address(): Same as above, just calculate the index using hash function, then just pass the key to the address function of the BST object BST_table[index] with key as parameter. Then return the net address in the form of string. The net address should be the index number, then append '-' and then the address obtained from the BST_table[index].

5. update(): Same as the rest functions, after finding the index, update in the relevant BST.
6. get(): after finding the index, call the get function of the relevant BST with the given key as the parameter.

NotFoundException class: I have created it but not implemented it as I don't feel the need to use it. I am doing the same work by returning null or -1.

Driver Class/Assignment3 :

Finally I have created a driver class which will implement the main class.

I have passed all the three arguments in main class and then read all the three arguments which are given while running the driver class. I then stored them in the suitable variables.

Then I have checked whether the mode given is "DH" or "SCBST" and then created the map accordingly with the given table size. After that, I have read my input file which is passed as the third argument. I read it line by line and on the basis of first word of the line, I have performed my operations and printed the output on the console.