COL106 Assignment3

Additional Classes:

Pair Class : objects of this class store a key of K type and a value of T type.

Pair2 Class : objects of this class store two strings : fname and lname.

Node class : it contains two objects of node itself (corresponding to left and right) and one object of Pair class.

HashTable class : it contains all the methods of Double Hashing.

HashTablesep class : it contains all the methods of Separate chaining.

DOUBLE HASHING:

In double hashing approach, I used an array of the objects of Pair class and size N.

Time Complexicities of all methods are given at the end of this report.

1. **Insert method:**
   In this method, I created an object of Pair2 class "k" and store the values of the key which I have taken in the constructor. After this I created a string s = fname+lname and get the values of hash1 and hash2 using djb2 and sdbm.
   Define a variable k2 (index) which is initially equal to hash1. If array[k2] is not null then we will replace k2 with (hash1 + i*hash2)%size. If it is null then place a Pair of key, value at that index. It returns the value of i.
2. **Update method:**

   Same as insert method but in this case, we will check whether the key of array[k2] is same as the given key or not. If yes, then replace the value with the new value otherwise k2 = (hash1 + i*hash2)%size. It returns the value of i.

3. **Delete method:**

   Initial part of this method is same as update method. The difference is that if array[k2] is same as the given key then assign null value to array[k2]. Is not then assign k2 with (hash1 + i*hash2)%size.

4. **Boolean contains method:**
   Same as update method. We will check whether the key of array[k2] is same as the given key or not. If yes then return true. Else replace k2 with (hash1 + i*hash2)%N where i = i+1.
5. **get method :**
   get will return an object of T type. Here if T is Student type object and not equals to null then we will print all the information of that student corresponding to the given key.

6. **Address method :** it will return the final hash value (index) of the given key. Procedure is same as contains method.

All the methods will print "E" if there is any kind of error while executing these.

<u>SEPARATE CHAINING USING BINARY SEARCH TREE:</u>

In separate chaining approach, generally the size of the hashtable should be less than total number of extries. I created an array of the objects of node class which is working here as a hashtable. Each node contains the value of the node (i.e. an object of Pair class which contains K type key and T type object) and the address of the nodes which are either on the left, or on the right to that node. By this arrangement of the nodes, a Binary Search Tree has been formed.

(LIMITATIONS : due to a lot of confusions, I calculated the hash function using both first and last name. i.e. if first name of two students are same then I arranged them according to their last name. Simply I join fname and lname for both and compare with lexicographic order.)

1. **Insert Method :** In this method first of all, I calculated the hash value of corresponding to the key. We say "root" to the first node of each index. If the root is null, then we assign the key and value pair to that node. If the root is not empty then we will compare the left and right nodes of the root. If the given key is smaller than the corresponding root, then we will go to the left of the node. Otherwise go to the right. If the new key is same as the given key. Then we will assign the given key value pair to that node.

2. **Update Method :** It compares the pair of the object stored in the first node with the pair of the object stored in the node initially. If equal, the object previously stored is replaced by obj. if not, the key stored in the node is compared to the given key. If it is greater than the given key, the function is repeated for the binary tree considering the right node of this node as the root. If not,the function is repeated for the binary tree considering the left node of this node as the root. the iteration is continued till the same key is found and the obj is updated there. The return type is integer and it returns the number of nodes touched. The average case time complexity is O(log n) and the worst case is O(n).

3. **Delete Method :** the given key and the value is compared with the key and value of each node .if they are equal, three cases arise:
    1. ***Node to be deleted is leaf:*** Simply remove from the tree.
    2. ***Node to be deleted has only one child:*** Copy the child to the node and delete the child
    3. ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

I assumed all cases and these subcases. . If the element which is to be deleted is not a root, then go for other root(either left or right, depends on the comparison of their first and last names. And without any recursion, I first checked all 3 cases for the root. For root complete code is separate with if else loop. After this I consider many cases like if child has subchildren or not and arrange them accordingly to the position of that children. In the 3rd case, we have to go to the left most element of the right branch. I simply use while loop for this The average case time complexity is O(log n) and the worst case is O(n).

4. **Contains method :** simply it iterates in a BST such that if the given key is greater than the iterated key then iteration will go to the right otherwise left. In this process, if the iterated key is same as the given key then method returns true. If the whole iteration is completed, then it will return false.
5. **Get method:** Similar to contains method. It will return an object of T type when the given key is matched with the iterated key. Before proceeding this method, we will check whether the given key contains or not. If not then simply it will print an "E".
6. **Address method:** Procedure is same as contains method. Initially it prints the index value + "-". After this, Whenever the iterator changes its position either to left or to right. It will print "L" or "R" corresponding to the direction.

All the methods will print "E" if there is any kind of error while executing these.

<u>**TIME COMPLEXITY:**</u>

Double Hashing:

let the number of elements be N and the hash table size be B

The probability of a collision on our initial probe is N/B. Assuming a collision, our

first rehash will try one of B - 1 buckets, of which N - 1 are filled, so the probability

of at least two collisions is N*(N-1)/B*(B-1).

Similarly, the probability of at least i collisions

is

:: N*(N-1)......(N-i+1)/(B*(B-1).....(B-i+1))

If B and N are large, this probability approximates (N/B)i

. The average number of probes is one (for the successful insertion) plus the sum over all i ≥ 1 of the

probability of at least i collisions, that is, approximately B/(B-N)

Now the load factor is N/B that is @ hence the average time complexity is O(1/(1-@)).

is N/B has a constant upper bound then we can say that the **average time complexity** is O(1).

insert()=O(1)

update =O(1)

contains=O(1)

get=O(1)

delete=O(1)

address=O(1)

Worst TIme Complexity

The worst time complexity is clearly O(N). Because if we choose all the elements such that all have the same hash value for both the functions, then clearly at every insertion we have to traverse through all the elements inserted earlier. This would be true for all the other functions because, they first reach that point and perform a constant time complexity opeartion after that.

insert()=O(N)

update =O(N)

contains=O(N)

get=O(N)

delete=O(N)

address=O(N)

Separate Chaining

Time complexity for insertion in a binary tree is O(H)

where h is the height of the root we can see it as that to insert we have to reach at the

leaf node. Starting from the root we need count the downward .The maximum number of steps would be equal to height of tree. Hence the time complexity of O(N/H).

For the rest of the operations that is update ,contains, get, delete, address we need to find the element and then do some constant time operation other

than for delete . So we can similarly say that The time complexity of the other operations would be O(H).

let the size of hash table is p and there are n elements to be inserted in the table

average time complexity=

average number of elements in each cell would be n/p now from the above the insertion and other operation take O(H) time. The average height would be O(log n/p)

beacuse in average case the insertion is assumed to be random.

Hence the **average time complexity** of the various operations would be would O(log n).

insert()=O(log n)

update =O(log n)

contains=O(log n)

get=O(log n)

delete=O(log n)

address=O(log n)

If we maintain some upper bound for N/p then average insertion time would be O(1).

Worst Time Complexity

if all the keys are such that the hash function places them in the same index then the number of elements in that index would be N.

Now the maximum Height can be N-1 if all the elements are inserted in increasing or decreasing order.

Hence the worst case time complexity would be O(N).

insert()=O(N)

update =O(N)

contains=O(N)

get=O(N)

delete=O(N)

address=O(N)

Thanks
Manoj Kumar
2018CS50411
Group - 4