COMP SCI 7201

Algorithm and Data Structure Analysis

Name :- Vandit Jyotindra Gajjar
ID :- a1779153

Quiz - 2

Question 2 (2.1)

Part 1

⟹ The table

y	h(y)
Alice	3
Bob	6
Carol	2
Dave	3
Eve	7
Trent	6
Walter	0

— . The hash table has length of 10. [indexed from 0 to 9]

P.T.O

—. Output of the hash table after insertion using chaining method.

Index	Value
0	Walter
1	–
2	Carol
3	Alice → Dave
4	–
5	–
6	Bob → Trent
7	Eve
8	–
9	–

P.T.O.

— . Output of the hash-table after insertion using Linear Probing method.

Index	Value
0	Walter
1	-
2	Carol
3	Alice
4	Dave
5	-
6	Bob
7	Eve
8	Trent
9	-

P.T.O

[a1779153]

(2.2)

—. As per the lecture slides, One of the methods to determine the height of a node, here we can use the coin flip method till "head" occurs.

—. In such case, Probability of flipping a head is 1/9. So the equation will be as follows:

→ Prob. of height of the new node:

[prob. of "tails"] ^ [height - 1] * [prob. of "heads"]

- Prob. of height = [1/10] ^ [height - 1] * [9/10]

- Prob. of height = 9/[10 ^ height]

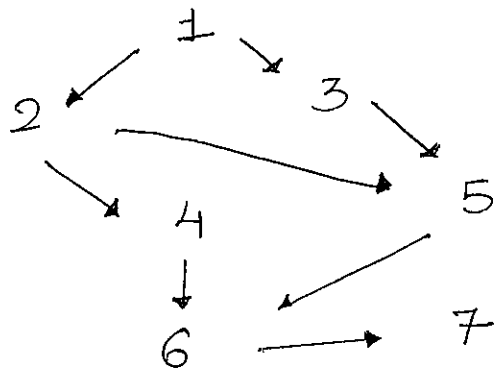
→ As the coin is biased the prob of getting a head is high. So, the height of an element would be less as head will occur in very less trails. So, if the height of an element is less, then it will take more number of iterations to find the value in list. For searching we need to go through more elements. Now, this will increases the insertion cost because we need to search for the correct location for the element before insertion.

P.T.O.

[a1779153]

(2.3)

—. Generating a random directed graph with n nodes and m edges.



→ we considering that each node in the graph has required 1 storage unit.

(*) Adjancy List

—. For n node and m edge, the graph requires n+m memory unit. Here, we need n storage unit array to store node, after that nodes will be connected to linked list. So, here the linked list will use m storage unit.

~. Adjancy list table

Node	Linked list
1	2 → 3
2	4 → 5
3	5
4	6
5	6
6	7
7	-

The total storage unit will be
$m+n = 8+7 = 15$ unit

P.T.O

[a1779153]

(*) Adjancy Matrix

—. For a matrix, we will require n * n storage units.

the matrix will be as follows:

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	1	0	0
4	0	0	0	0	0	1	0
5	0	0	0	0	0	1	0
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0

The total storage unit will be $n * n = 7 * 7$

$= 49$ unit

P.T.O

[a1779153]

Question 1

Part 1 For ADSA Binary Search Tree the ith Largest integer.

(*) Note :- first determine ith largest number index ;

if the index < number of nodes for left subtree and Current
↳ number present in left subtree.

if the index > number of nodes for left subtree and Current
↳ modify index val and remove left subtree for search.

↻ repeat this until you find the node.

(*) Sample Code style

In :- root , int i (index) Out: int result

logic :- findILargest (Node Curr, int nodeIndex)
.... if nodeIndex == Curr → numLeftNode + 1
↳ Out = Curr > Val
return
.... if nodeIndex < Curr → numLeftNode + 1
↳ findILargest (Curr → Left , nodeIndex)
.... if nodeIndex > Curr → numLeftNode + 1
↳ findLargest (Curr → right, nodeIndex - [Curr ↓ numLeftNode+1])

(*) Time Complexity O(logn)
average

(*) Time Complexity O(n)
worst

P.T.O

[a1779153]

Part 2 For ADSA Binary Search Tree insertion

(*) Note :- First we traverse BST and using recursion call, divide the tree for focusing in half. By performing insertion, we need to increase the counter for the nodes, in which path we traversed.

(*) Sample Code style

In :- root, int num Out :- Node Curr

logic :- insertNode (Node Curr, int num)

```
if Curr == 0 or Null :
    ↳ root = new Node val
    ↳ return Curr
if Curr → Val == num :
    ↳ return Curr
if Curr → Val > num :
    ↳ Curr → left = insertNode(Curr → left, number)
    ↳ Curr → numLeftNode ++
// Left traverse
// update
else :
    ↳ Curr → right = insertNode(Curr → right, number)
    ↳ Curr → numRightNode ++
// right traverse
// update
return Curr
// return the Curr Node
```

(*) Time Complexity : O(n) example 1 2 3 4
worst

(*) Time Complexity : O(log(n))
average
where n is the node

P.T.O

For ADSA Binary search Tree Deletion

Part 3

(*) Sample code style

In :- root, int num Out :- void

Logic :- deleteNode (Node Curr, int num)

if Curr == 0 or Null :

↳ then we return Curr

else if [Curr → val > num] : // And logic

&&

[Curr → left != Null or 0]

// left traverse ↳ Curr → left = deleteNode (Curr → left, num)

↳ Curr → numLeftNode --

// update

else if [Curr → val < num] // And logic

&&

[Curr → ~~left~~ right != Null or 0] :

// right traverse ↳ Curr → right = deleteNode (Curr → right, num)

↳ Curr → numRightNode --

// update

if Curr → left == Null or 0 :

↳ Curr → right → tmpLeft

↳ remove Curr

↳ return tmpLeft

else if Curr → right == Null or 0 :

↳ Curr → left → tmpRight

↳ remove Curr

↳ return tmpRight

return Curr

// The logic for in case left/right deleted node has 0/1 child node

(*) Time Complexity average O(log(n)) where

(*) Time Complexity worst O(n) n is the number of node.

- - - - - - - - - End - - - - - - - -

Date 13/05/2020

Sign Vandit