COL106 Minor 2

ARPIT SAXENA

TOTAL POINTS

42 / 70

QUESTION 1

Q1 11 pts

1.1 Q1-a 8 / 8

✓ + 1 pts **Correct Insertion/Probes for 873/837: 3**

✓ + 1 pts **Correct Insertion/Probes for 9734/8843: 3, 1**

✓ + 1 pts **Correct Insertion/Probes for 280/640 : 0**

✓ + 1 pts **Correct Probes for deleting 9734/8843 : 3, 1**

✓ + 2 pts **Probes for searching 143/323 : 3, 1, 4**

✓ + 2 pts **Correct Insertions/Probes for inserting 14/32: 0, 3, 1**

- 1.5 pts Did not listed probes

+ 0 pts Incorrect/Unattempted

1.2 Q1-b 1.5 / 3

✓ + 1.5 pts **Null slot found then stop search**

+ 1.5 pts Probed the slots in correct order untill repeated slot is found (when table is full and element not present) or element found

+ 0 pts Incorrect/unattempted

QUESTION 2

2 2 5 / 5

✓ + 2 pts **Main final structure**

- 0.5 pts Not Setting parents field of p and g

✓ + 2 pts **Correctly attaching old children of n**

- 0.5 pts Not setting parent field of "Old Children of n"

✓ + 1 pts **Correctly setting new parent of n**

+ 0 pts Incorrect Solution

QUESTION 3

3 10 pts

3.1 Q3-Delete 0 / 6

✓ + 0 pts **Incorrect/Unattempted**

+ 1 pts Correctly handles no children case

+ 1 pts Correctly handles the case when the key values matched, left subtree is NULL and right subtree is not NULL

+ 2 pts Correctly handles the case when the key values matched, left subtree is not NULL and right subtree is NULL ; also include recursing again on left subtree

+ 2 pts Correctly handles 2 children case : Includes finding the inorder successor/predecessor and deleting that predecessor/successor and recursed again on root->left

+ 1 pts Incorrect handling of 2 children case

+ 1 pts Only deleted the first key found

- 0.5 pts Not mentioned about how to delete the successor or predecessor or incorrect successor deletion

- 1 pts Didn't recurse on root->left after deleting

+ 2 pts Incomplete/Partial Correct/ Specific Case only

3.2 Q3-Search 4 / 4

+ 0 pts Incorrect/Unattempted

✓ + 1 pts **Correct base case i.e. termination condition**

✓ + 0.5 pts **Implemented only the Binary Search**

✓ + 1 pts **Correctly search duplicates keys**

✓ + 1 pts **return the duplicates keys**

✓ + 0.5 pts **pseudo code attempt**

+ 0 pts Wrong search

storing is not allowed.

QUESTION 4

4 Q4 2 / 5

✓ + 2 pts **- Used global lock/ synchronized**

- Mentioned about locking the root

+ **1 pts** - Incorrect usage of per node lock - leading to inconsistent results.
- Incomplete answer regarding lock/ synchronized. But in the right direction.

+ **0 pts** No explanation of how to implement synchronization or entirely incorrect. Not mentioning anything about how to synchronize.
Not answered.

+ **3 pts** Justification - the deadlock issue if the root is not locked

QUESTION 5

5 Q5 **3 / 5**

+ **0 pts** Incorrect/Unattempted
+ **1 pts** lock whole list without proper explanation
+ **1.5 pts** lock whole list with proper explanation

✓ + **1 pts** **Correct and precise synchronization for search with incorrect explanation**

+ **2 pts** Correct and precise synchronization for search with correct explanation

✓ + **1 pts** **Correct and precise synchronization for insert with incorrect explanation**

+ **1.5 pts** Correct and precise synchronization for insert with correct explanation

✓ + **1 pts** **Correct and precise synchronization for delete with incorrect explanation**

+ **1.5 pts** Correct and precise synchronization for delete with correct explanation

QUESTION 6

6 Q6 **7 / 10**

+ **0 pts** Incorrect or Unattempted
- **0.5 pts** Didn't observe that all leaves have the same height

✓ + **1.5 pts** **Part A: Observed that minimum height will happen when each node is 4-node.**

✓ + **2 pts** **Part A: Derived Correct expression for maximum number of keys for a given height**

✓ + **0.5 pts** **Part A: Derived Correct expression for minimum height given number of keys n**

✓ + **1.5 pts** **Part B: Observed that maximum height will happen when each node is 2-node.**

✓ + **1.5 pts** **Part B: Derived correct expression for maximum height given number of keys n**

+ **1.5 pts** Part C: Observed that to find a key, number of nodes examined is at most the maximum height

+ **1.5 pts** Part C: Observed that maximum number of comparisons per node are 2 (for 3-nodes and 4-nodes).

- **1 pts** PartC: not mentioned why 2 comparison in part c

+ **0 pts** Didn't notice that n denotes number of keys, not number of nodes

- **1 pts** PartC: maximum height not mentioned

QUESTION 7

7 Q7 **6 / 6**

✓ + **6 pts** **Correct**

+ **2 pts** If algorithm partially or fully correct but not O(h)

+ **3 pts** If algorithm not written but worked out by example

+ **0 pts** If algorithm not written or incorrect

+ **1 pts** Partial marks (Brute-force or almost brute force kind of approach or just writing select root as the one of the roots and attach remaining as sub-tree). A recursive approach on nodes is brute-force.

+ **0 pts** Click here to replace this description.

QUESTION 8

8 Q8 **0.5 / 6**

+ **3 pts** correct explanation
+ **3 pts** correct answer
+ **0 pts** Incorrect or blank

✓ + **0.5 pts** **explanion provided but not appropriate**

QUESTION 9

9 Q9 **0 / 6**

✓ + **0 pts** **Incorrect/Didn't attempt**

+ **1 pts** Recognition that Preorder traversal breaks into Root followed by left subtree and right subtree

+ **2 pts** Recognition that elements in the left subtree are smaller than the root and, scanning to get there

+ **2 pts** Proper book keeping in the pseudocode

+ **1 pts** Proper justification

QUESTION 10

10 Q10 **5 / 6**

✓ + **4 pts** **Correct non recursive algorithm**

✓ + **1 pts** **Time complexity O(n) provided algorithm is correct**

+ **1 pts** Argument for correctness

+ **1 pts** Correct recursive solution

+ **0 pts** Incorrect/ Unattempted

+ **0 pts** Does not find depth of binary tree, stops after finding depth of first leaf

gradescope

Please write your entry number and name on each sheet.

Entry Number: 2018MT10742 **Name:** ARPIT SAXENA

B

COL 106 MINOR EXAM II
SEMESTER I 2019-2020
1 hour

Please do not allow any bag, phone or other electronic device near you. Keep your ID card next to you on the desk. Maximum marks available for questions are listed in []. Write answers in the provided space. Justify all answers.

1. (11)

a) [8] Given the Hash function h below, list the table slots touched/probed and show the status of the Hash table after each listed operation (in order from left to right, starting with an empty hash table). The table has 5 slots. Assume open addressing with h_i as given: (Deletion is by marking as deleted.)

$$h = h_0 = (\sum \text{digits}) \% 5$$
$$h_i = (h_0 + 3*i) \% 5$$

Insert:837	Insert:8843	Insert:640	Delete:8843	Search 323	Insert:32
		640	640		640
	8843	8843	~~8843~~(Deleted)		32
837	837	837	837		837

Slots probed:

3 | 3,1 | 0 | 3,1 | 3,1,4 | 0,3,1 |

Rough space

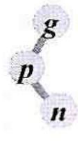
18 % 5 = 3
23 % 5 = 3
(3 + 3) % 5
10 % 5 = 0
23 % 5 = 3
8 % 5 = 3
5 % 5 = 0

b) [3] How do you conclude that a searched key does or does not exist in such a Hash table?

We first compute the hash h (= h_0 = h_k (say)) and check index h in the table.
If the searched key is there, we're done. Else, if the slot has been marked deleted or there is some other k in it, we calculate h_{k+1} and check there. (and recur)
Else, we have concluded that the key does not exist

~~For~~

2. [5] Write code (precise syntax not required) to tri-node restructure the following configuration. Assume references left, right, parent and value are stored for each node. Assume *p.parent.left* == *p* and *n.parent.right* == *n*. (No references should be assumed to be null.)




```
Restructure(Node n):
Node p = n.parent, g = p.parent;
n.parent = g.parent;
if g.parent != null:
  g.parent.left == g ? g.parent.left = n : g.parent.right = n
else:   // g.parent == null => g is root.
  root = n
  p.right = n.left; n.left.parent = p; g.left = n.right; n.right.parent = g
n.left = p; p.parent = n
n.right = g; g.parent = n
```

3. (10) Consider a binary search tree that allows keys to be repeated in multiple nodes such that *keys equal to any node's key are always in that node's left subtree*. Write the pseudo-code for search and delete for this tree that each take time O(h) for a tree with height h. The search function must return all instances and the delete function must delete all instances of the given key.

~~Suppose consume is a function which accumulates values to return~~

[6] Delete(root, key):

[4] Search(root, key):

```
list = []
curr = root
while curr != null :
    if curr.key == key
        list.append(curr.value)
    if key <= curr.key :
        curr = curr.left
    else:
        curr = curr.right
return list
```

4. [5] Consider a *Red-Black tree* that supports multiple threads inserting, searching or deleting in parallel. (Asume each thread uses the standard algorithms to perform these operations.) What synchronization is required to ensure that the threads do not interfere with each other's operation?

The answer is written by considering the equivalency of 2-4 tree and red black tree. We have locks for each node. In searching, lock the root, then determine which path to take. Then go to the corresponding child, lock it and then release parent's lock, and recur.

For insert, we acquire lock as above. The parent lock is only released after we know it's corr. 2-4 child along whose path we have to insert is not a 4-node. We acquire child's lock, and recur.

For delete, lock is released if we know corr. 2-4 node of parent is not a 2-node

5. [5] You need to implement a *skip-list* that supports multiple threads inserting, searching or deleting in parallel. (You may assume that each thread uses the standard algorithms to perform these operations.) What synchronization is required to ensure that the threads do not interfere with each other's operation?

We need to only ensure the nodes in the current interval we're looking at are locked. Nodes outside of this interval can be accessed by other threads too. Consider tower of nodes in arrays, we can lock the beginning array and end array of the interval. So, another thread which wants to access towers in between them would be suspended, as only way to node next to the beginning array is through that array.

ease write your entry number and name on each sheet.

Entry Number: 2018MT10742 **Name:** ARPIT SAXENA

6. [4+3+3] Show that the height of a 2-4 tree with n keys is (a) no less than floor(0.5 $\log_2 n$) and (b) no more than $\log_2 n$. (c) Also show that the number of comparisons required to find a key is no more than 2 $\log_2 n$. (5 extra marks to show the bound to be 1.3 $\log_2 n$)

(a) Minimum height of a 2-4 tree will be obviously, when ~~there~~ all nodes are 4-nodes.

For that, a level at depth 0 has 1 nodes $\equiv$ 3 keys

at depth 1 has 4 nodes $\equiv 3 \times 4$ keys

$\vdots$

depth i has 4^i nodes $\equiv 3 \times 4^i$ keys

$\therefore$ Total keys for such a tree of height h =

$$3(1+4+\cdots+4^{h}) = \cancel{3(1+4^h} \; \frac{3(4^{h+1}-1)}{4-1} = 4^{h+1}-1$$

~~Any 2-4 tr~~

$$n \leq 4^{h+1}-1$$

$$\Rightarrow h+1 \geq \log_4 (n+1)$$

$$\Rightarrow h \geq 0.5 \log_2 (n+1) - 1 > \lfloor 0.5 \log_2 n \rfloor$$

$$\Rightarrow h > \lfloor 0.5 \log_2 n \rfloor$$

(b) For max. height, each node is a 2 node

$\Rightarrow$ Total keys for height h tree $= 1+2+\cdots+2^h = 2^{h+1}-1$

$$n \geq 2^{h+1}-1$$

$$\Rightarrow h \leq \log_2 (n+1) - 1 < \lfloor \log_2 (n) \rfloor$$

$$\Rightarrow h < \lfloor \log_2 (n) \rfloor$$

(c)

7. [6] Provide an O(h) algorithm to merge two heaps with 2^h keys each. (Assume that keys in the two heaps are comparable to each other and h is an integer.)

~~define merge (a, b, newheap) // a, b are nodes~~

~~if a.key > b a.priority >~~

Suppose we have a given place in heap to add a new node to, and two nodes a and b, ~~then~~ say a has greater priority than b. Then a goes to ~~new~~ the given place in heap. Let a's current two children be a_l and a_r. In the heap, we make b (along with it's subtree) the left child of a. Now, we set a and b as a_l and a_r, ~~new~~ place in new heap as now a's right child and recur until a and b both are null.

To the above method, we supply roots of the both the heaps initially. We note that by the given condition, the ~~tre~~ binary trees of both heaps are complete (~~at~~ assuming 2^h-1 keys each)

Note that is O(h) as in each step of recursion, we descend one level and do constant time operations in each step.

8. [6] In an AVL tree with with 20 nodes having numbers 1 to 20, respectively, as keys, which numbers may not appear in the root?

We know that for an AVL tree with height h,

~~n_{min}~~ min. nodes, $n_{min} = F_{h+1}$, where $F_0 = 0, F_1 = 1$, ~~$F_n = F_{n-1} + F_{n-2} + 1$~~

Suppose ~~root~~ has height h.

Then it's children have height $h-1$ and $h-1$ / $h-2$ resp.

~~min max~~

$h = 5$, then height ~~5~~ 4 child has ≥ 12 nodes

3 ≥ 7 nodes

$\Rightarrow$ root has to be 8 or 13

$h = 4$, then height 3 child has ≥ 7 nodes

2 $\geq$

ease write your entry number and name on each sheet.

Entry Number: 2018MT10742 **Name:** ARPIT SAXENA

9. [6] Given the keys of a Red-black tree ordered according to the pre-order traversal order, provide an algorithm to re-construct the Red-black tree. (You do not need to re-construct the nodes' original colors.)

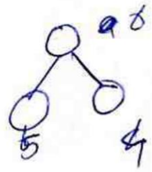
Assuming we have at our disposal a rbtree implementation which has ~~an~~ insert implemented.

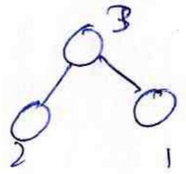
```
let ~~rb~~ tree = new rbtree()
for key in pre-order_traversal:
    tree.insert(key)
```

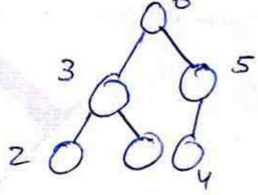
10. [6] Provide *non-recursive* pseudo-code to compute the depth of a binary tree.

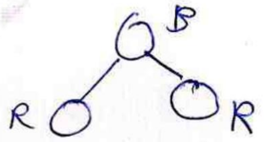
```
if (root == null) return -1
queue.insert(root)
depth = -1
curr_nodes = 1
while queue is not empty:
    new_nodes = 0; depth++;
    for i = 1 .. curr_nodes:
        node = queue.pop()
        ~~if node != null:~~
            if node.left != null:
                queue.insert(node.left)
                new_nodes++
            if node.right != null:
                queue.insert(node.right)
                new_nodes++
    curr_nodes = new_nodes

return depth
```

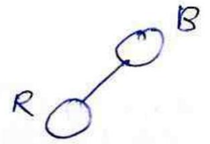
ROUGH

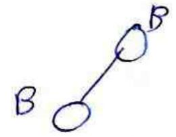


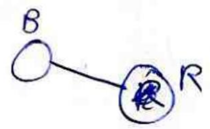


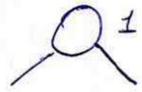












1,