CSCI-GA.1170-001 Homework 6

Ankit Sati

TOTAL POINTS

52.5 / 46

QUESTION 1

1 Rotate that BST! 19.5 / 15

part b

✓ **- 1 pts** *Missing update of u wrt original v's parent*

part d

✓ **- 0.5 pts** *missing/incorrect running time*

part e

✓ **+ 4 pts** *correct algorithm*

✓ **+ 1 pts** *correct running time*

✓ **+ 1 pts** *correctness*

QUESTION 2

2 Comparing Binary Search Trees 15 / 13

✓ **- 0 pts** *Correct*

✓ **- 0.5 pts** *Missing/Incorrect running time in part (a)*

✓ **- 0.5 pts** *Missing/Incorrect runtime in part (b)*

✓ **- 0.5 pts** *Missing/Incorrect runtime in part (c)*

✓ **+ 2 pts** *Correct algorithm in part (d)*

✓ **+ 1.5 pts** *Correctness in part (d)*

1 Missing bound on the number of rotations

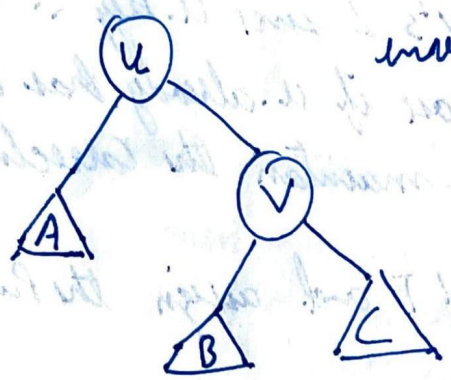
QUESTION 3

3 2-3 Trees 18 / 18

✓ **- 0 pts** *6-3 a,b,c,d Correct*

Q.1.

a)



invocation Call. (T, u)

(b) Left Rotate (T, u):

```
if U. Right ≠ NIL:
    temp = u. Right
    u.Right = V
    V. Left = Temp.
else:
    u . Right = V
if T . Root = V;
    V = T . root
    V.P = U
else. if P = V.P.
    V.P = U.
Return T;
```

Run-Time = O(1)

The Run time is Constant time

There are only 2 operations happening that of Comparison & assignment so both of them will happen in Constant time O(1).

Correctness:

① The algorithm interchanges the parents of u & v's & sets u.key ≤ v.key
v will became the right child of u. In the case if u already has a right child it becomes the left child of v to maintain the correctness

② We also take care of the case when v is root of T and assign the Parent to both u & v.

As mentioned before that the algorithm runs in O(1) so the correctness have been justified.

c.

DELETE ROTATE (T, z)

v = z.p.
v.left = z.left.
Left Rotate (T.v)
Transplant (T.z)

Runtime = Constant O(1)

DELETE ROTATE (T, z):
while deg (z) > 1:
Left Rotate (T. z. left)
Transplant (T. z)

Runtime = O(n)

Correctness.

~~Base case: We need to check for all nodes smaller than z and z.~~

→ Transplant can delete a node only if there are less than 2 children.
→ So we keep pushing down until we have 2 children.
→ As soon as we reach this, the loop terminates & we know our algorithm is true.

d) Assumption

u.right ≠ nil.

Rotate Successor. (T, u):

min = u.right

while = min.left ≠ Nil:

Left Rotate (T, min.left)

Return u.right.

Correctness.

⇒ We are trying to find the successor of node u.

- As per the rule, it should lie in the sub-tree of u-right.
- As we have assumed that u.right ≠ NIL, we know that u.right exist and is the successor to u.
- If u.right has no children, then we just RETURN u.right, but if child is found, then we find the smallest key moving up the left nodes. using Left-ROTATE.

⇒ [The loop terminates, when all the left nodes are traversed]

- v.right will now be the smallest element in the subtree.
- It will be the inorder successor of node u.
- ∴ it will RETURN v.Right

Runtime :- O(logn).

n → number of nodes in the sub-tree of u.right.

Since we have used a WHILE loop to traverse till the end of u.right the runtime will be O(logn).

(e) ROTATE SEARCH Part (T, x)

if T.root.key = x:
    Return T

— else if T.root.key < x:
    subnode. = Rotate Search. (T.root.right.x)

— else if T.root.key > x.:
    subnode = ROTATE SEARCH (T.root.left.x)

RETURN ROTATE (T.subnode.root)

Correctness:

① Base Case → If the key of the root matches x, we return the tree

② → All the other cases we need to check. Root.key. Post this we recursively traverse the subtree where the root is equal to x.

→ That tree is then returned & the ROTATE is applied on node. which will make the target node. the root of the subtree to which it is returned to finally.

→ By doing this the target node will keep moving up untill root.key = x & ROTATE function returns a tree.

→ Recursive calls are made until we have located the target node.

→ Since there exist a target node, we can state that the algorithm must end.

→ Hence, we have justified the correctness.

Runtime:

O(n) when we have the entire tree one side.

O(log n) → This is for the balanced tree

Answer → O(n) → worst Case.

1 Rotate that BST! 19.5 / 15

part b

✓ **- 1 pts** *Missing update of u wrt original v's parent*

part d

✓ **- 0.5 pts** *missing/incorrect running time*

part e

✓ **+ 4 pts** *correct algorithm*

✓ **+ 1 pts** *correct running time*

✓ **+ 1 pts** *correctness*

Problem 2.

(a) FillTree (T, u, A, start)

If u = NIL.

Return Start.

Else.

x = FILLTREE(T. u. left, A, start)

u. key = A[x]

x + 1.

S = FILLTREE (T, u. right, (A, x))

Return S.

Base Case. → u = NILL. (ie. Simply return S)

Proof by Induction

Algorithm is true for trees 0 to k-1

For step k,

→ The Algo calls on the left subtree with keys less than k. which is assumed correct.

So A[start, x-1] placed. & u. key = A[x].

(b) TransformBalanced.(T, N)

1. Start
2. BST T inputs having n nodes.
3. Traverse BST in order & store result in Array. (This array will have the in order traversal
4. Make middle value as root node.
5. Recursively do the same for left & right subtree.
6. ~~Get the middle part of left subtree~~
6. Take the middle of left half & make it left child of the root node.
7. " " " " right " " " " right " " " "
8. Recursively store the items in T.
9. Return T.
10. End.

Time Complexity = O(n)

Correctness.

→ Traverse the tree downwards to find left & right node on both side.
→ Array will store value one node towards the leaf.
→ Let the program terminate the algorithm is correct.

(c) IS Equivalent (T, T', n):

1. Array A = In Order Traversal (T).
2. For i in A:
   if Search (T', i) = True:
      Continue
   else.
      Return False.
3. Return True.

Run Time = $O(n \log n)$.

Correctness :- The algorithm stores element from one tree & iteratively checks each element with the other tree.

→ Even if a single element is missing return false.
→ Only if the loop finishes we return true

1. START
2. Initialize a new head_node= NULL and prev_node= NULL, head=NULL
3. Now, recursively travel the right sub-tree of the root to reach the right most node. Name this as temp.
4. Now, make rightNode as temp->right and leftNode as temp->left.
5. Check if the head_node is NULL. If it is, then head_node= temp;
                                    head=head_node;
                                    temp.right= NULL;
                                    prev_node= head_node;
      else
            prev_node->right = temp;
            temp->right= NULL;
            prev_node= temp;
6. Now, recursively do this for left sub-tree of the root.
7. head contains the left transformed tree.
8. STOP

Time Complexity of the above algorithm will be O(n) since we are traversing the tree and visiting every node.

Correctness of algo
- Algo travreses the tree to find every leaf in node
- Finally it reduces to 2 elements
- Loop ends

2 Comparing Binary Search Trees 15 / 13

✓ **- 0 pts** *Correct*

✓ **- 0.5 pts** *Missing/Incorrect running time in part (a)*

✓ **- 0.5 pts** *Missing/Incorrect runtime in part (b)*

✓ **- 0.5 pts** *Missing/Incorrect runtime in part (c)*

✓ **+ 2 pts** *Correct algorithm in part (d)*

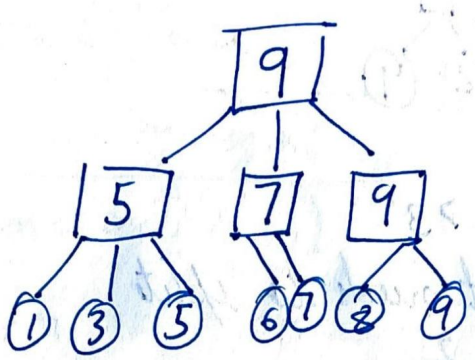
✓ **+ 1.5 pts** *Correctness in part (d)*

1 Missing bound on the number of rotations

Problem 3 - 2-3 Trees.

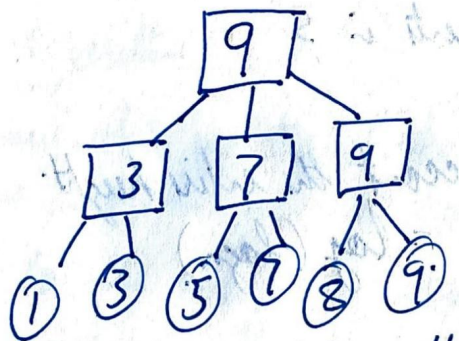
(a) $x = 6$

INSERT (6)



→ 6 should be inserted under 9 as $6 > 5$.
→ 9 has a total of 4 nodes so it need to split & give up a child

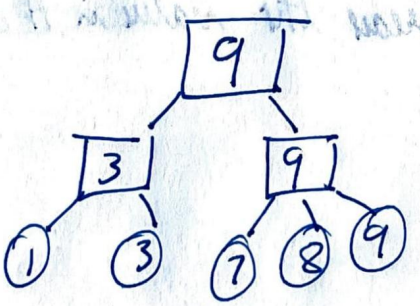
DELETE(6)



→ 6 is deleted then 7 is left with only 1 node.
→ It checks with the other siblings with unbalanced degree 3 & takes a child to balance the tree.

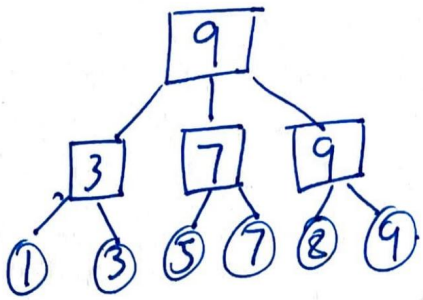
(b) $x = 5$

DELETE (5)



→ 5 is deleted in a single step.
→ The degree still remains greater than 1 so no change is needed.

Insert(5):



→ 5 is put under 9 since 5 > 3.
→ 9 will have 4 nodes so it needs to split
→ Post split all nodes are ≥ 2.

(c) We need to store all the occurrences of a node as well as the number of nodes appeared in the augmented 2-3 tree.

⇒ Number of elements in tree
= No of distinct elements in S.
= $\ln n$.

Since we need to take in account the entire height.
= $\log(\log n)$

Algorithm

1. → Insert elements from S by traversing down the tree & increment the count the leaf node based upon occurrence.

2. → Initialize the Array A.

3. → Traverse to each leaf & increase the value in A as many times as the count of that leaf node.

4. Return Array A.

Running time

T = Insertion + Traversal of tree

$= O(n \log \log n) + O(\log n \log \log \log n)$

$= O(n \log \log n)$

Height = $h = \log(\log n)$

Insertion = $n \cdot h = n \log \log n$.

Traversal = $\log(N) \cdot h$.

$= \log n \times \log \log n$

Since we are traversing the height, it will be $\log n$ for each node.

Correctness of Algorithm

→ For every element of S, we travel the tree & insert it. If it is already present then we just increment the count.

→ While adding to the array the elements are in sorted order. & we also append each element as many times as it counts.

→ Hence the array that we return after the two steps runs in $(n \log \log n)$ time.

→ As both loops terminate after returning the sorted array, hence our algorithm will also terminate.

Hence Proved

d) As the question describes, an 2-3 augmented tree.

① Leaf nodes will contain the book entries [Book #, Name, Book Price]

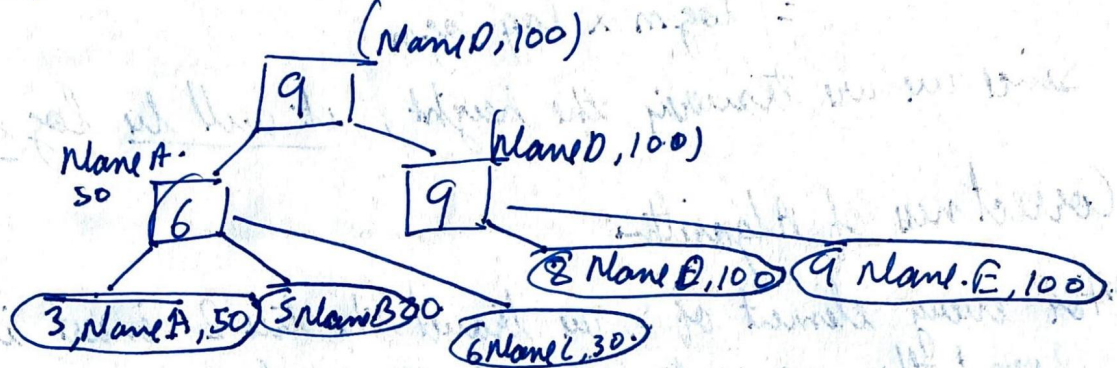
② Internal nodes will contain the max number of books among the sub-tree as value is added with book name & price among the sub tree

= [Book.Number] (Book name with max price)

Let the entries be.

(3, NameA, 50), (9, NameF, 80), (6 NameB, 30) (5, NameC, 20)

(8, NameD, 100)



Above Mention 2-3 tree shows

→ leaf nodes contain records.

→ Internal nodes contain the max book numbers.

→ Finally the max priced book in its sub-tree.

⇒ Indexing Operation → The indexing of the books is dependent upon an unique element that is Book Numbers.

⇒ Report Book Name with max price.

→ Just need to return the book name sorted in the root node. as that is storing max priced Book Name. for the whole tree.

Related time =. O(1) time.

⇒ Report price of a given book number.

Time = O(log n.

→ Just like creating a book, we search for book by the book number.

→ Report Book name with largest price Book I.D in Range (low, high) in (O log (n):

- Internal nodes having value less than low are ignored.
- As we traverse down the tree we keep track of variable with max. price.
- All the values in siblings between (low, high) are compared & stored in our variable
- while we traverse to the next internal node having value greater than high.
- Any other node with value > high are ignored.
- Since these nodes will have values which are greater than a value which is is greater than high.

(low . . . . . high < x < y) → 2nd internal Node } x ignore.

1st

⇒ both traversing down.

~~Booking Operation~~

⇒ Mapping of operations

→ Create a book is O(log n)

Compare the book numbers & search for an internal node that can store this entry.

This takes O(log n) time as height of tree = log n. & we travel down.

The max book number & max priced book values are updated after the newer entry to compare its values with its sibling under same parent.

All of the data are completed under log n. time

∴ The creation of book is O(log n) time justified. (Similar to INSERT)

⇒ Closing a book is log n.

→ We first search the book with the book number which takes (log n) time.

→ After finding the leaf, we delete that leaf node & update the values of its parent internal nodes.

∴ Justification remains the same for opening the book i.e time (log n)

⇒ Add Subtract price of a given book.

→ Again we need to locate the book with book number, taking O(log n) time & then we update its book price value by adding / subtracting given value.

→ After this, the price is compared with sibling & we update the max priced values of its parent internal node.

Time = O(log n)

If we keep traversing down this path we will reach end & return the stored value.

Runtime = $O(\log n)$ + Comparison $O(1)$

$= \boxed{O(\log n)}$.

3 2-3 Trees **18 / 18**

✓ **- 0 pts** *6-3 a,b,c,d Correct*