1. Show how OS-SELECT(T.root, 18) operates on the red-black tree T of Figure 14.1 in CLRS.

A picture containing sword, various, different

Description automatically generated

* OS-SELECT(T.root, 18) – R = 13. 13 < 18, so we will go to the right.
* OS-SELECT(X.Right [41], 5) – R = 6. 6 > 5. Will go left
* OS-SELECT(X.Left [30], 5) – R = 2. 2 < 5. Will go left.
* OS-SELECT(X.LEFT [28], 1) – R = 2. 2 = 2. 28 was found.

1. Show how OS-RANK(T, x) operates on the red-black tree T of Figure 14.1 and the node x with x.key = 38 in CLRS.

A picture containing sword, various, different

Description automatically generated



* OS-RANK(T, 38) – R = 1 + 1 = 2. Y = X
* Until Y reaches the root: At node 38:
  + Y is the right child
  + Y moves up to the parent which is Y.key = 30
* Y is still not the root: At node 30
  + Y is the **not the right** child of root.
  + R = 2 + 1 + 1 = 4
  + Y moves up to the parent which is Y.key = 41
* Y is not the root: At node 41
  + Y is the right child
  + R = 4 + 12 + 1 = 17
  + Y moves up to its parent which is the root
* R = 17

1. Given an element x in an n-node order-statistic tree and a natural number i, how can we determine the ith successor of x in the linear order of the tree in O(log n) time?

An order-statistic tree is an augmented Red-Black tree with a size and key value added to each node. To find the ith successor to x, we would need to traverse down the tree and use the size of each node as the rank. It would behave similarly to OS-Select and OS-Rank in which we would have to determine to travel left or right based on the size of each node related to the list of successors.

* OS-Rank determines the rank of x through an in-order walk of T
* OS-Select returns a pointer to the node containing the ith smallest key

We can use OS-Rank to find the ith successor then use OS-Select to find the location of that node with the ith successor.

* Call OS-RANK on the tree and have it find key i
* Take rank value of OS-Rank and use it as the key for OS-Select to find the ith successor

1. In class we augmented the red-black tree so each node included its size. Suppose we instead stored in each node its rank in the subtree of which it is the root. (The size of each node is not stored.)

* Show how to maintain this information during insertion
* Write the methods OS-SELECT(x, i)
* Write the methods OS-RANK(T, x)
* State the worst case run time of inserting an item, OS-SELECT, and OS-RANK using Theta notation.

Which is more efficient: augmenting the data structure by adding at each node the size of the subtree of which it is the root, or augmenting the data structure by adding at each node the rank of that node of the subtree of which it is the root?

**Tree for reference using the ranks of subtrees**

Diagram, schematic

Description automatically generated

**Tree after Left Rotation Tree after Right rotation**

A picture containing text, device

Description automatically generated A picture containing text, indoor, black

Description automatically generated

* Because we’re storing the size of each subtree, every node would be calculating the sum of its children’s ranks
* Every leaf would begin with the rank of 1 similar to before.
  + When inserting a new node into the tree, we would insert the new node at the bottom and add +1 to every parent we traversed down to find the spot to insert the new node
* Because this is a red-black tree, the rank will also be changed based due to right or left rotations
  + For a right rotation, the ranks will be changed based on the nodes being moved
    - The node being rotated up to become the new parent node has their rank increased by 1 + parent.rank. This is because this node will take an additional subtree when becoming the new parent
    - The parent being rotated down will decrement its rank by parent.right.rank + 1 because it is losing one of its subtrees when rotating down
  + The left rotation will behave similarly to the right rotation.
    - The right node being rotated up has their rank increased by 1 + parent.rank because they have to add up the rank of the previous parent’s subtrees
    - The parent node being rotated down gest its rank reduced by 1 + parent.left.rank

OS-SELECT(x, i)

OS-Rank(T, x)

1. Write the pseudo code for how to find the maximum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

Provide the CPU running time, and the number of disk accesses using Big-Oh notation.

* The keys are stored as x.key1 ≤ x.key2 ≤ x.key3 ≤ x.key4 ≤⋯≤ x.keyn
  + The maximum element located within a node would be at the furthest right location of the list of keys.
* However, there is a possibility that the node has children which contains keys greater than the parent node’s maximum key values
  + We can memorize the previous maximum value of a node as the predecessor of a key when we check the node’s children for the next maximum value

B-MAX-KEY(T.root, 0)

B-MAX-KEY(x, MAX)

i = 1

**while** i ≤ x.n

if (MAX < xi) //Record the max value of the node

MAX = xi

i = i + 1

**if** x.leaf //Final level reached, every node on this path has been checked

RETURN MAX

**else** Disk-Read(x.ci)

//Take the max value and use it to recursively check the node’s children if they have the max value

K = B-MAX-KEY(X.ci, MAX)

If (MAX < k)

RETURN MAX = k

* CPU RUNNING TIME-Would execute for every disk access we perform and we would hae to check every key in the node of rank t. O(t logt n)
* Disk Accesses-We’re searching the children based on the link associated with the key with the maximum value. We would traverse down each level based on the height of the tree. This would cost O(logt n)

B-TREE-PREDECESSOR(x, k)

//Work similarly to Tree-Search except we keep track of the predecessor node before each jump

i = 1

**while** i ≤ x.n and k > x.keyi

i = i + 1

**if** i ≤ x.n and k == x.keyi

return (x, k)

**elseif** x.leaf

return NIL

**else** Disk-Read(x.ci)

k = B-TREE-PREDECESSOR(X.ci, k)

1. Insert the following keys: 40, 41, 42, 45, 44, 43 in this order into the b-tree of degree 2 below using the algorithm discussed in class. Show the tree after each item is inserted.

How many disk accesses occurred when 40 was inserted? Give an exact number (note that the root is always in main memory.)

Diagram

Description automatically generated

Tree after inserting 40: 31 < 40 < 67 -> 35 < 40 < 48 -> 47 < 40. The key would be moved to the furthest level below and be at level 3. 35 < 40 < 48, so 40 would be inserted into the middle node (36, 37)

Box and whisker chart

Description automatically generated

Chart, box and whisker chart

Description automatically generated

Inserting 41: 31 < 41 < 67 -> 35 < 41 < 48 -> 40 < 41. Similarly to 40, 41 would be moved down to level 3. 35 < 41 < 48, so it would be moved to the node containing (36, 37, 40) which is a full node which will initiate a pre-emptive split.

* (36, 37, 40) will split. 37 moves up into the parent: (35, 37, 48). 36 and 40 form their own nodes where 41 will form a pair with 40

Graphical user interface, application, Teams

Description automatically generated

Inserting 42: 40 < 42 < 48 -> 41 < 42, so 42 would be placed as (37, 41, 42). There is no size violation for this insertion.

Chart, box and whisker chart

Description automatically generated

Inserting 45: 31 < 45 < 67 -> 40 < 45 < 48, so it would be inserted into node (37, 41, 42). This initiated a split.

* 41 moves up to (35, 40, 48) which initiates a split
  + 40 moves up to (31, 67, 81) which initiates a split
  + 67 moves up and becomes the new root. 35 and 48 become child nodes of 67.
    - 35 takes the children of 32 and 36
    - 81 takes the children of (75, 68) and (95, 98)
  + 41 forms a pair with 48, making (41, 48)
    - 37 and 42 split to form their own nodes which are children of (41, 48)
  + Pair (41, 48) will hold the children of 37, 42, (49, 57)
  + 41 < 45 < 48, so 45 forms a pair with 42, making (42, 45)

Diagram

Description automatically generated

Inserting 44: 44 < 67 -> 40 < 44 -> 41 < 44 < 48 -> 42 < 44 < 45, so 44 gets inserted into (42, 45). There is no size violation.

Chart, box and whisker chart

Description automatically generated

Inserting 43:

* 43 < 67 -> 40 < 43 -> 41 < 43 < 48 -> (42, 44, 45) initiates a split
* 44 moves up to (41, 48)
  + 42 and 45 split to form children nodes under (41, 44, 48)
* 41 < 43 < 44, so 43 gets inserted into 42
  + 43 and 42 form a pair of (42, 43)

Diagram

Description automatically generated

1. This question is very similar to the question from week 2.

In week 1, when you asked for stock advice, you didn’t think you would get so many suggestions. Due to not wanting to lose most of your money in broker fees,1 you don’t want to invest less than 100$ per stock, so you decide to buy the B < N stocks that did the best in the past 6 months time. Since the advice keeps flowing in and you decide to invest your money as soon as you get paid your wages for being a TA, you want to quickly be able to determine the best stocks to buy in O(B + log N) time and insert a new stock suggestion in O(log N) time.2

Specifically, you want your data structure to implement the following operations:

* Insert(s, r) where s is the stock name and r is how well the stock performed in the last 6 months in O(log N) time3
* Top Stocks(B) where you return as a linked list the top B stocks4 for any B in the range [1 . . . N] in O(B + log N) time (Small amount of extra credit will be given for O(B) time.)
* Average return(B) where you return the average return of the top B stocks in O(log N) time.

For this question, you may assume the stock names you have already removed any duplicate names.

//Insert stock in O(log n) time

Insert(s, r)

r D T:root

if r:n = = 2t 1

s D ALLOCATE-NODE./

T:root D s

s:leaf D FALSE

s:n D 0

s:c1 D r

B-TREE-SPLIT-CHILD.s; 1/

B-TREE-INSERT-NONFULL.s; k/

else B-TREE-INSERT-NONFULL.r; k/

1. Having done so well on the stock market, your time is not your own. Perhaps you shouldn’t have started your own firm. You have so many meetings (and you still have to attend class...)

You need to know how much time you have to study in between your meetings. You decide to create your own system for letting you know how much free time you have during any interval of time s to e. You will represent date/time by Unix epoch time https://en.wikipedia.org/wiki/Unix\_time which is just an integer.

You data structure must be able to:

1. Add Meeting(m, s, e) insert a new meeting, m that starts at s and ends at e in time O(log n), where n is the number of meetings/classes you have already entered.5
2. Check Schedule(s, e) determine if you can fit in a new meeting given the starting, s, and ending time, e in O(log n) time
3. Study Time(s, e) determine how much time you have to study between your meetings given a starting time s and time and ending time e in O(log n) time.

9. (3 bonus points) Think of a good 6 exam question or homework assignment question for the material covered in Lecture 5.