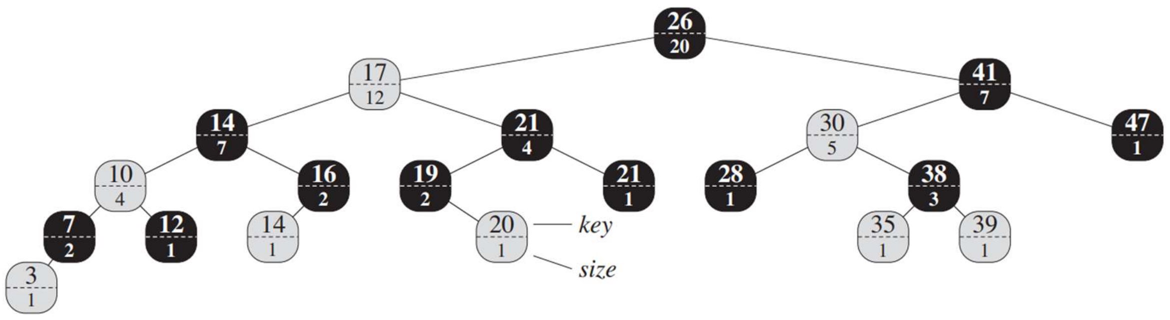
Brandon Vo

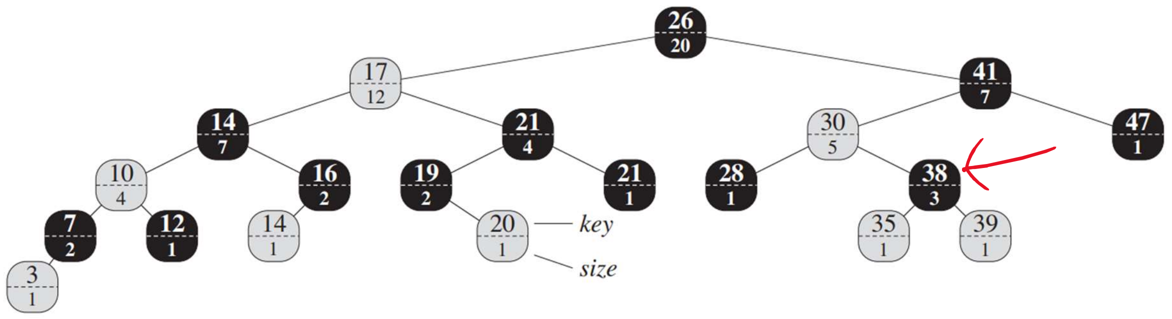
1. Show how OS-SELECT(T.root, 18) operates on the red-black tree T of Figure 14.1 in CLRS.



- 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 right.
- OS-SELECT(X.Left [38], 3) – R = 2. 3 > 2. Will go right
- OS-SELECT(X.Right [39], 1) – R = 1. 1 == 1. OS-Select located the node with value 39 and size 1

Brandon Vo

2. 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.



- 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

Brandon Vo

3. 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 i^{th} 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.

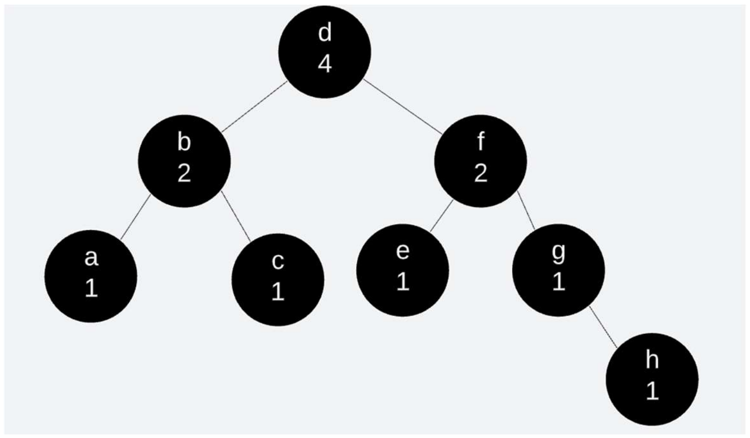
- Call OS-RANK on the tree and have it find the rank of i
    - OS-RANK counts the rank as though it counts all previous nodes in an in-order distribution, so OS-Rank would be counting the rank based on the size of all previous nodes behind it
    - The ith successor would be located in the right subtree which normally isn't counted by rank. We need to add I to track the successor to take into account the right subtree
- Take value of OS-Rank and use it as the key for OS-Select to find the ith successor
- OS-SELECT(T, OS-RANK(T, X) + i)
- This would cost O(2 log n) time to call OS-Rank and OS-Select, making the function cost O(log n) time overall

Brandon Vo

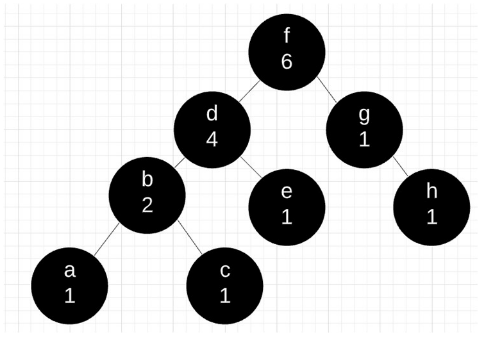
4. In class we augmented the red-black tree so each node included its size. Suppose we instead stored in each node its rank in the subtree of which it is the root. (The size of each node is not stored.)
- Show how to maintain this information during insertion
- Write the methods OS-SELECT(x, i)
- Write 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?

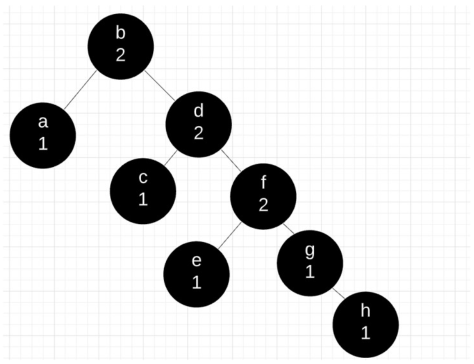
Tree for reference using the ranks of subtrees



Tree after Left Rotation



Tree after Right rotation



- 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 count the rank only from the left subtree
    - During our insertion, we increment the rank of every node by +1 if our new node would be located in its left subtree

    - There is no increment of rank if the new node would be located in the right subtree
    - Once we reach a nil node, we insert the new node and give it a rank of 1, then we rebalance the augmented red-black tree.
- Because this is a red-black tree, the rank will also be changed based due to right or left rotations
    - In a Red-Black tree rotation, only the ranks of two nodes are changed
    - Right rotation
        - The parent node is being rotated to the right where it loses its left subtree. Its rank decreases by parent.rank = parent.rank - (parent.left.rank + 1)
        - The left node keeps its left subtree when being rotated up. The left node will keep its rank after the right rotation.
    - Left Rotation
        - The right node will gain a left subtree after rotation, so its rank increases by right.rank = right.rank + right.p.rank + 1
        - The parent node being rotated down will inherit a right subtree which doesn't count towards its rank. It still keeps its left subtree, so the parent node keeps its rank after the left rotation
- OS-Select and OS-Rank will be changed such that R will just use x.size
    - The original functions used r = x.left.size to measure the size of the left subtree
    - But for our version, x.size is already the same as x.left.size + 1
    - OS-Rank will also be changed such that r = r + y.p.left.size + 1 into r = r + y.size because for our modified ranking tree, y.p will already be holding the size of the left subtree
    - Otherwise, the comparison and while loop remain the same
- OS-Select and OS-Rank were already calculating a node's rank during their loop invariants, the efficiency only changes by a constant time amount if we use an augmented tree with ranks instead of size.
- Both OS-Select and OS-Rank will be $\theta(\log n)$

```
OS-SELECT(x, i)
r = x.size          //Modified to add the right size as well
if i == r
    return x
elseif i < r
    return OS-SELECT(x.left,i)
else return OS-SELECT(x.right, i - r)
```

```
OS-Rank(T, x)
r = x.size          //Modified to add the size of the right side as well
y = x
```

Brandon Vo

```
**while** y $\neq$ T.root
    **if** y == y.p.right
        r = r + y.p.size    //Changed from r = r + y.p.left.size + 1
    y = y.p
**return** r
```

Brandon Vo

5. 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 $x.key_1 \leq x.key_2 \leq x.key_3 \leq x.key_4 \leq \cdots \leq x.key_n$
    - The maximum element located within a node would be at the furthest right location of the list of keys.
    - Where every non-root node has x.n keys
    - So we can just look for the last key in the rightmost leaf node in a B-Tree for the node with the maximum key
- B-TREE-PREDECESSOR:
    - Brightspace mentions that you don't need to find the key in the B-tree, so assuming that we start with a pointer to our given node
    - To find the predecessor, we look for the nearest key preceding our given key
        - If it's a leaf, we check the key right before the one we have
        - If our given key is the first element in a node, we must check the parent for the predecessor
        - If that parent is also the first node in its tree, we must check the parent after that and keep repeating until we find a node that we can find the predecessor of
    - We can use B-MAX-KEY to search for the predecessor of a parent node with a key
    - Otherwise, search and look for the key and move up if the given key does not have children

Brandon Vo

```
B-MAX-KEY(T.root)


B-MAX-KEY(x)
If x == T.NIL
    Print "There is no tree"
if x.leaf //Final level reached, every node on this path has been checked
    RETURN X.key[X.n]      //Return the last key in the node
Else
//Take the max value and use it to recursively check the node's children if they have the max value
    Disk-Read(x.c_i[X.n + 1])
    Return B-MAX-KEY(X.c_i[X.n+1])  //Move to the next node in the children
```

- CPU RUNNING TIME-We would only have to check the last key in each node for the maximum and see if that key has a link to another node. $O(\log_t n)$
- Disk Accesses- To acquire the maximum key, we must traverse the entire B-tree which requires a disk read for each node until we reach the leaf at the rightmost node. This would require the height of the B tree $O(\log_t n)$

Brandon Vo

```
//Assuming x is the pointer to the node we're finding the predecessor of
B-TREE-PREDECESSOR(x)
//Work similarly to Tree-Search except we keep track of the predecessor node before each jump
If x.leaf            //Find the matching key
    Mark i as the index of array X where the key is located
//X is a leaf and is found somewhere in the list after the first spot, take the previous node before k as
the predecessor
    If i > 1 return X.key[i-1]
//Element does not have children, we must go upward
    Else
        If x.p == NIL
            Print "There is only one element.  Predecessor not found"
        Disk-Read(y.c[i])
        Y = X.p
        J = 1
        //Scan the node to find the key that has x.key
        While (Y.key[j] ≠ x.key[i] )
            J++
            DISK-READ(Y.c[j]        //We have to read the child node of every key until we
find x.key[i]
        If (j == 1) Y = Y.p
//If this parent is alone, move into the next parent similar to binary search trees
        Else    Return Y.key[j-1]        //Found the preceding parent
else    //Element has left children, find the largest in the subtree
    DISK-READ-(x.c[i])
    Return B-MAX-KEY(x.c[i])
```

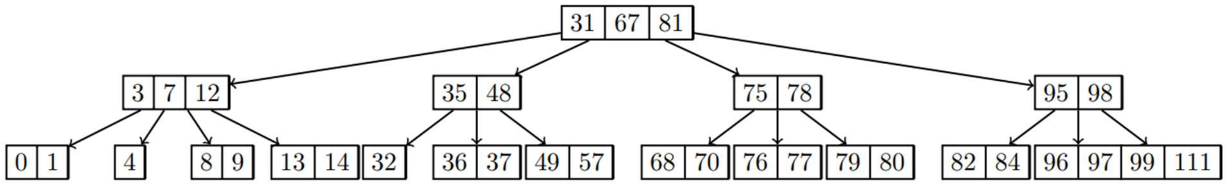
- CPU Run Time-Assuming we start the node with the given key, would have to find the key or node that precedes the given key.

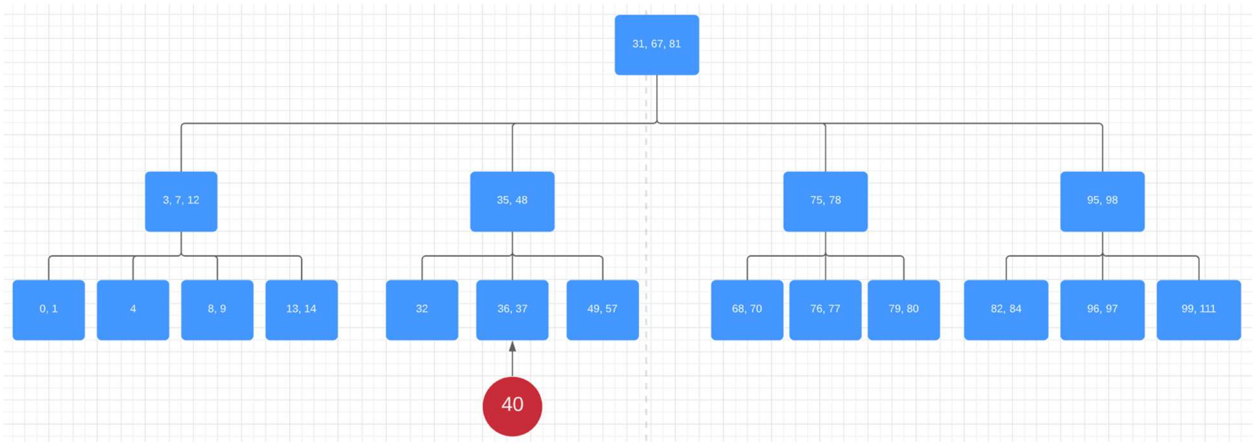
  - During each access, we simply need to check which key is behind our given key k. Since we already start with k, we can check the preceding key or have to recursive and check each parent for a preceding key $O(t \log_t n)$
- Disk Accesses-In worst case, we would either have to find the maximum of our given key's sub-tree for the predecessor or we would have to recursively check the parents to find the predecessor. Both scenarios would require us to recursively check up or down the B tree and possibly check an entire list located in a node to see if the parent contains our node
  - $O(\log_t n)$

Brandon Vo

6. 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.)

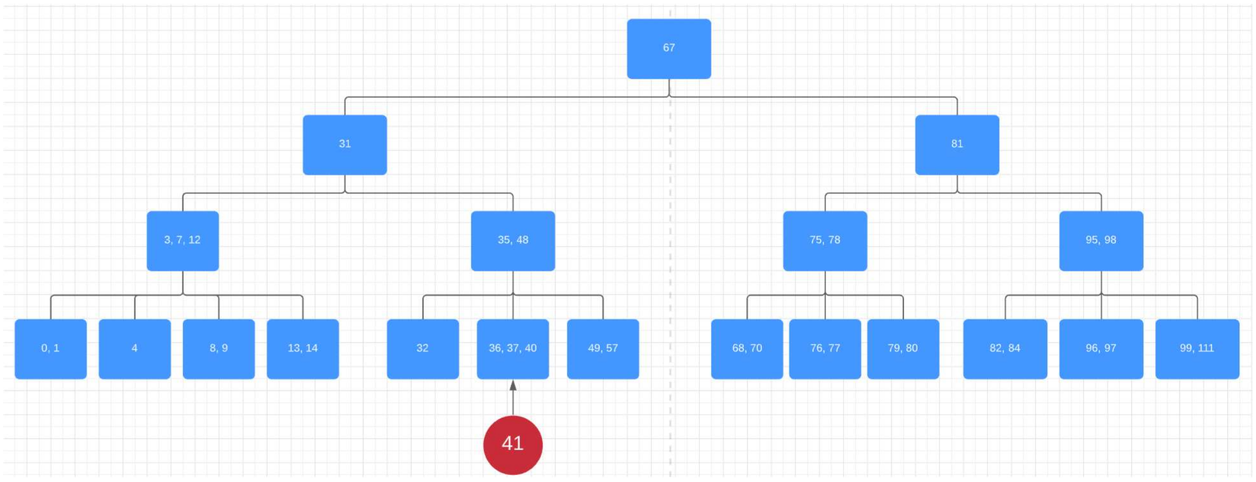




Tree after inserting 40: 40 < 67 (Root node always in memory) -> 31 < 40 (1 disk read) -> 35 < 40 < 48 (2 disk read) -> 47 < 40 (3 disk reads). The key would be moved to the furthest level below and be at level 3. 35 < 40 < 48, so 40 would be inserted into the middle node (36, 37)

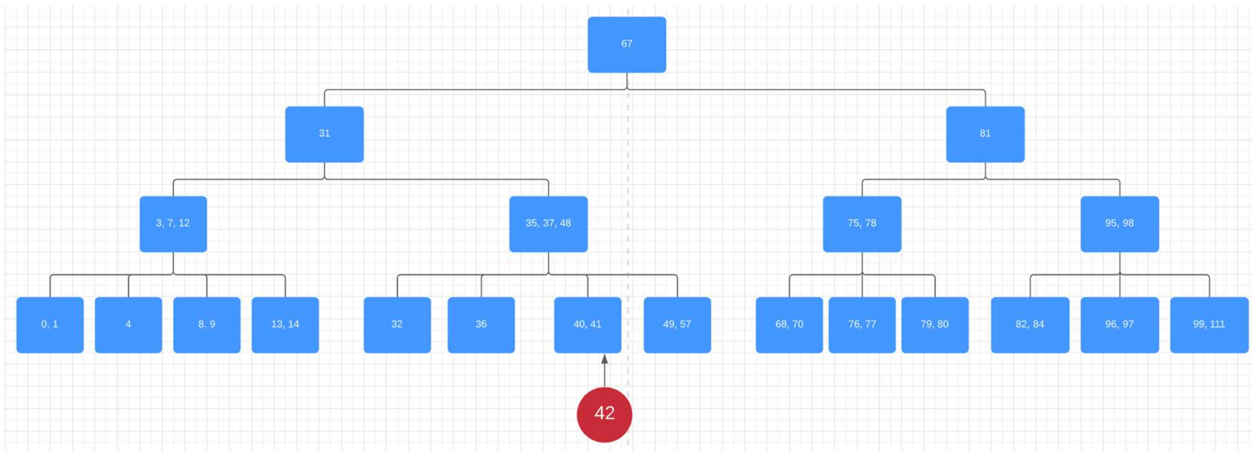
- Root node is permanently kept in memory, so there is no disk access required for the root
- When the root splits, it requires 3 disk-writes
- To insert 40, we have to travel from the root to the third level. We would need a disk-access to check 31 from 67, to access (35, 48), and to access (36, 67)
- Inserting 40 would require 3 disk accesses
- However, when traversing down the list, it will notice that the root node (31, 67, 81) is full and initiate a pre-emptive split
    - 67 becomes the new root node
    - 31 and 81 will become their own children nodes

Brandon Vo



Inserting 41: 41 < 67 -> 41 > 31 -> 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, 48). 36 and 40 form their own nodes
- 37 < 41 < 48, 41 forms a pair with the node of 40



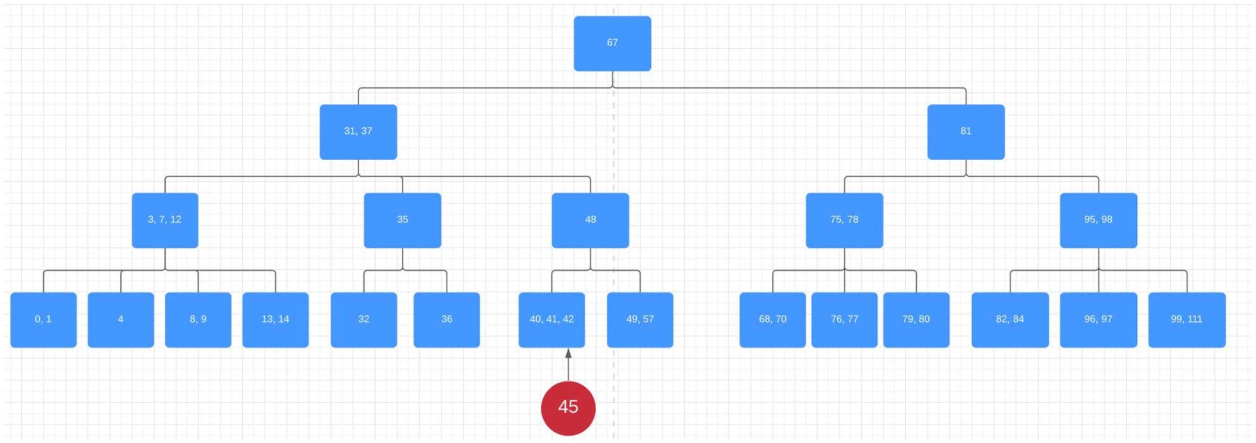
Inserting 42: 42 < 67 -> 31 > 42

- It sees (35, 37, 48) as full and initiated a pre-emptive split
- 47 moves up to 31. 35 and 38 form their own child nodes under (31, 47)

37 > 42 -> 42 < 48 -> 41 < 42

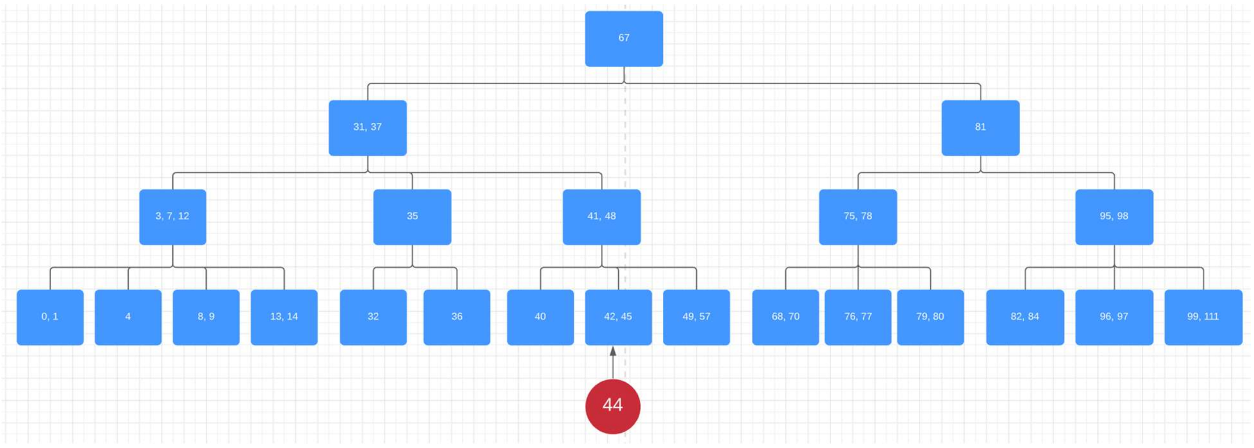
- 42 will be inserted into the node (40, 41)

Brandon Vo



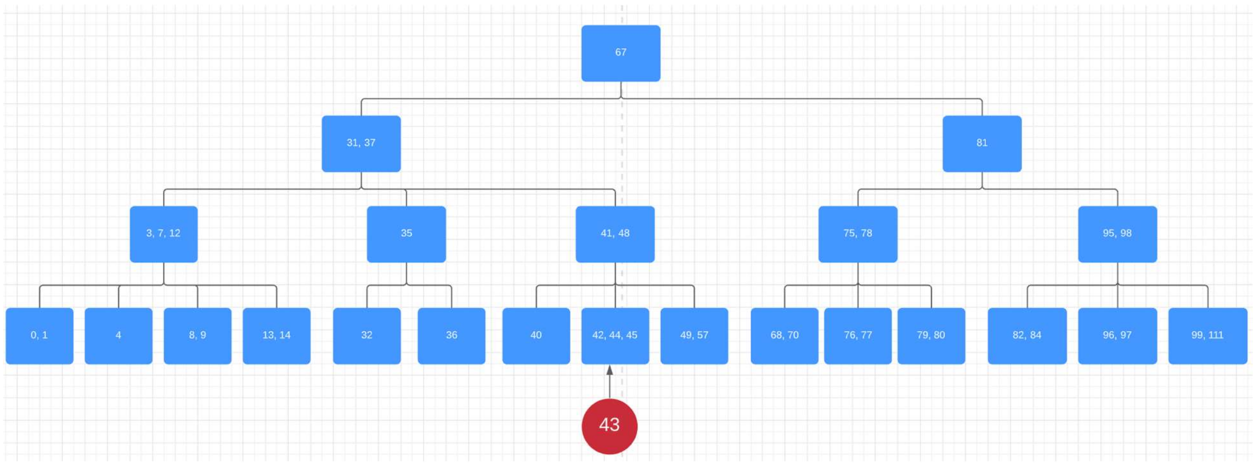
Inserting 45: 45 < 67 -> 37 < 45 -> 45 < 48

- Sees (40, 41, 42) and initiates a pre-emptive split
- 41 moves up to 48
- 40 and 42 form their own independent nodes
- 41 < 45 < 48
- 45 forms a pair with the node containing 42



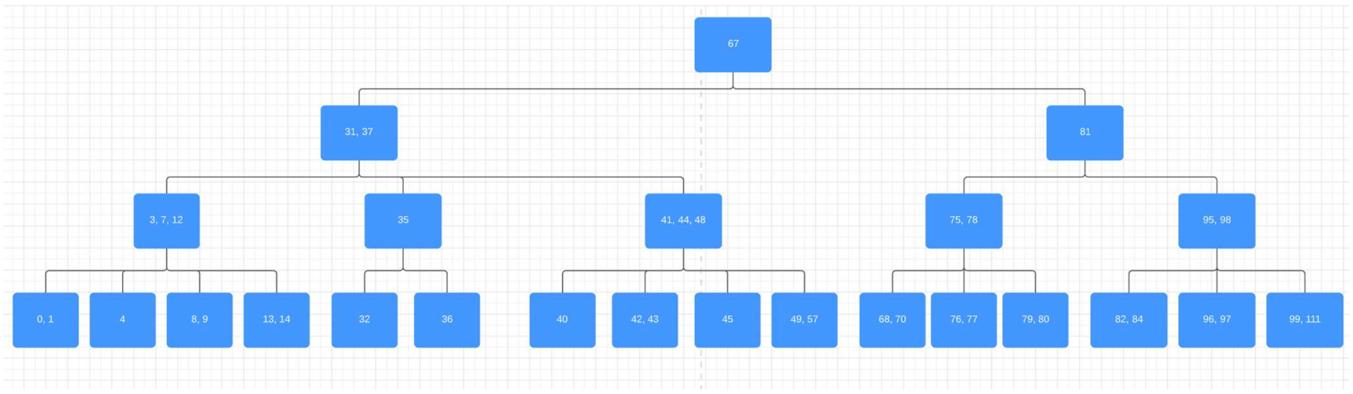
Inserting 44: 44 < 67 -> 37 < 44 -> 41 < 44 < 48. 44 gets inserted into pair (42, 45). No splits are necessary.

Brandon Vo



Inserting 43: 43 < 67 -> 37 < 43 -> 41 < 43 < 48

- Will see (42, 44, 45) and do a pre-emptive split
- 44 moves up to (41, 48)
- 42 and 45 form their own independent nodes



Brandon Vo

7. 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:

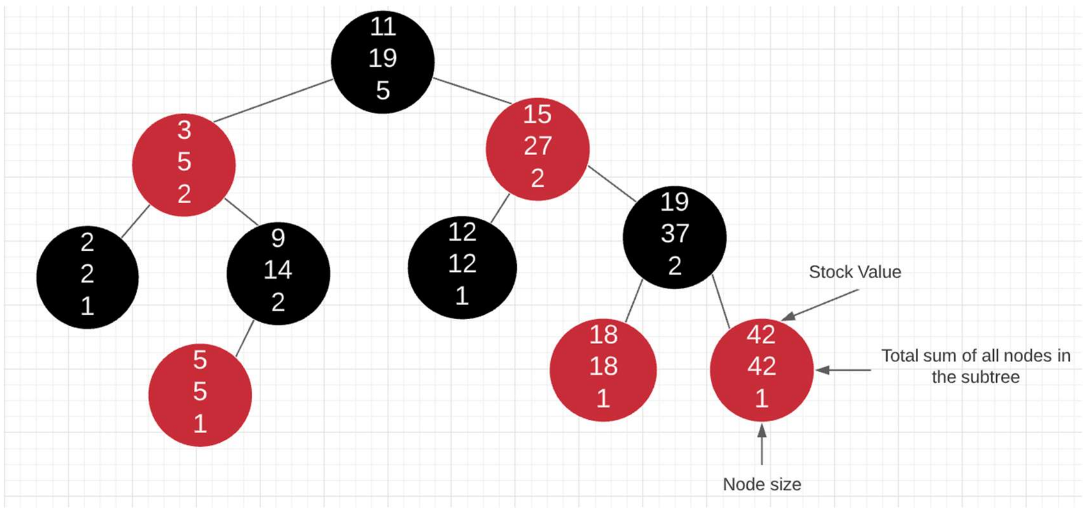
- Insert(s, r) where s is the stock name and r is how well the stock performed in the last 6 months in O(log N) time[3]
- Top Stocks(B) where you return as a linked list the top B stocks[4] for any B in the range [1 . . . N] in O(B + log N) time (Small amount of extra credit will be given for O(B) time.)
- 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.

- We can construct an augmented Red-Black tree using ranks and use that to find the k^th^ largest element in the tree
    - Then, we can find the largest element in the tree, the B^th^ element which would be located somewhere within the right subtree of the root node.
    - Once we have located the kth largest element and the largest element, we can do a post-order traversal, adding every element to our linked list B until we reach the B element.
- Whenever we insert an item, we will also attach a rank to that node based on its position. This rank gets updated during rotations as well.
- Top-Stocks(B)
    - Each node will be marked with a predecessor
    - The predecessor will lead to the next smallest value in the tree
    - We can start with the largest value at the far right of the tree and keep checking the predecessors until B items are added to the list
    - We can use a special pointer to reach the largest node in O(1) then recurse on the predecessors of that node in O(B) time
- Insert(s, r) – Would operate exactly the same as a standard red-black tree insertion
    - Would traverse down to the leaves of the red-black tree and insert on a nil node then rebalance the tree if necessary
    - We'll add a predecessor and successor node to each node
        - For each insertion, based on what direction, we compare based on values to see if it would be a new predecessor or new successor to keep track of the order of nodes
        - If we traverse to the left, we check if the new node's value is closer to the parent than it is with the predecessor's value and replace if necessary

        - If we traverse to the left, we check based on successor to see if the new node's predecessor would be the node we're checking
        - These comparisons and corrections run in O(1) time and do not change if a Red-Black tree rotation occurs
    - Because we're counting the size based on the left subtree, for every parent node that holds the new node in its left subtree, we increment the size by +1
        - Parent nodes holding the new node in the right subtree aren't affected
        - The size can be incremented as we traverse the tree, making it O(1) time to update the nodes as we go down
    - The operation needed to insert and rebalance the tree would remain at O(log n) time
    - The addition of an augmented variable to the node will still take only O(1) time to add to the node
- Average_Return(T, B)
    - We can further augment each node with an additional variable that describes the average of all nodes in its subtrees
    - Because we already have the rank, we can use total and rank to keep track of the averages in a node's subtree
    - To find the average return of the top B stocks, we start at the largest stock which is located in the rightmost subtree and travel backwards from there
    - Traversing for B will behave similarly to Top_Stocks except we'll be focusing on adding the total based on the rank found rather than traversing through the entirety of the B subtree
        - After adding up a subtree, we move up to the parent and compare the size with the number of B nodes we need to add up
        - If X.size matches the number of B nodes we need to add, we add the total and calculate the average
        - If x.size is less than the number of B stocks we need to add, we can add the parent and the entire subtree and check the next parent
        - If x.size is greater than the number of B stocks we need to add, we can't add the entire subtree, so we must start over predecessor node of the parent and count from there until we find the remaining B nodes
    - It takes O(log n) time to move to the rightmost subtree. It takes another O(log n) time to travel backwards until we count the sum of all B nodes because we're going back up. It would take O(2 log n) time to add the sum of all B stocks and count the average which is still O(log n)

Brandon Vo


11
19
5
3
5
2
15
27
2
2
2
1
9
14
2
12
12
1
19
37
2
5
5
1
18
18
1
42
42
1
Stock Value
Total sum of all nodes in the subtree
Node size

Brandon Vo

```
Top_Stocks(T, B)
X = T.root.max_node      //Root keeps a special pointer to the largest node in the tree to reach it in O(1) time
Create STOCKS as an empty linked list

For i = 1 to B
    Insert X to the linked list STOCKS
    X = X.predecessor
Return STOCKS
//Get the list of B stocks backwards.  This takes O(B) time
```

Brandon Vo

```
//Insert stock in O(log n) time
//Adding T to identify the tree
Insert(T, s, r)
y = T.nil // y is the parent of x
x = T.root
Create node z
while x ≠ T.nil
    y = x
    if r < x.return_value     //Compare based on stock value
        if x.predecessor == NIL or r > x.predecessor.return_value
            x.predecessor = z
            z.successor = x  //Used only for tracking
        X.size++          //Increment the size as this new node is in the left subtree
        x = x.left
    else
        if x.successor == NIL or x.sucessor.return_value < r
            z.predecessor = x        //Closest smallest value to z
            x.successor = z          //Only used for referencing
        x = x.right              //Don't count size for right subtrees

if T.root ≠ NIL
    if T.root.max_node == NIL or T.root.max_node.return_value < r
//The root will keep a special pointer to the furthest right node in the tree
//This is to make accessing the top B node O(1) time
        T.root.max_node = z
z.p = y
if y == T.nil
    T.root = z //tree T was empty
```

Brandon Vo

```
else if r < y.return_value
        y.size++            //Left subtree, increment size
        y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
z.size = 1                  //Leaf nodes start with a size of 1
z.return_value = r          //Start with the total sum as just the stock value
z.name = s
RB-INSERT-FIXUP(T, z)
//RB-INSERT-FIXUP will work the same except for its rotations


LEFT-ROTATE(T,x)
y =x.right
x.right = y.left
if y.left ≠ T.nil
        y.left.p = x
y.p = x.p
if x.p == T.nil
        T.root =y
elseif x == x.p.left
 x.p.left = y
else
        x.p.right = y
y.left = x
y.size = y.size + x.size            //Add the rank of the parent
//X's rank doesn't change as it retains its left subtree
```

Brandon Vo

//Right rotate would be symmetrical to left rotate except if x is the parent node and y is the left subchild, then the size would be calculated differently.

//For right rotate, x.size = x.size – x.left.size. As the parent node loses its left subtree

//For right rotate, y.size wouldn't change as the left child of the parent retains its left subtree after rotation

Brandon Vo

```
//Count the average return
Average_Return(T, B)
Total = 0          //In case B is across multiple different subtrees
AVG = B            //Counter for B
X = T.root.max_node    //Move to the largest node in the furthest right subtree so that we can count only the top B stocks
//Start backtracking and adding up the total
While X ≠ T.nil and AVG > 0
//If X is smaller than the total number of B stocks we need, the node and the entire left subtree
        If AVG > X.size
            Total = Total + X.total
            B = B – X.size
            X = X.P  //Move up to the next parent
        If AVG == 1
            //Add the parent only
            Total = Total + X.return_value
            AVG--
            Break
        Else    //AVG < X.size
            //We need to add a specific amount of nodes in the sibling
            //Enter the predecessor of the parent
            //Will terminate in the sibling subtree, making this O(2 log n)
            Total = Total + X.p.return_value
            AVG--
            If AVG > 1
                X = X.p.predecessor
            Else    Break
EndWhile
Return Total / B
```

Brandon Vo

8. 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:

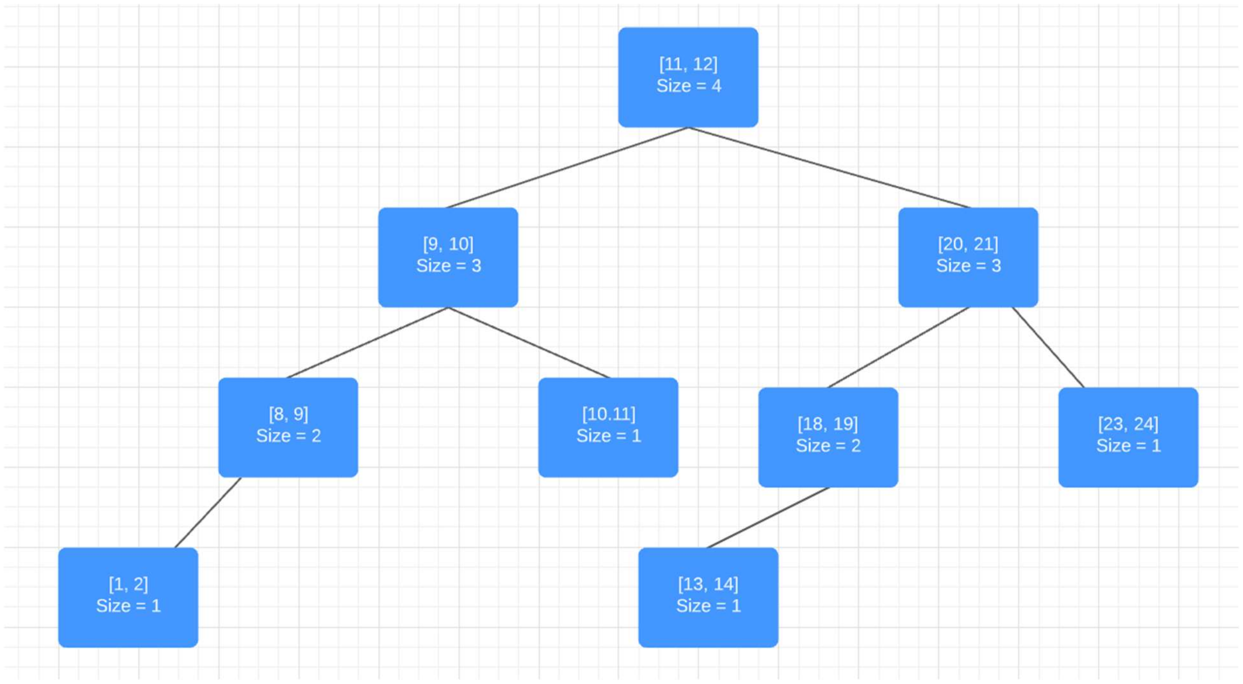
(a) Add Meeting(m, s, e) insert a new meeting, m that starts at s and ends at e in time O(log n), where n is the number of meetings/classes you have already entered.[5]
(b) Check_Schedule(s, e) determine if you can fit in a new meeting given the starting, s, and ending time, e in O(log n) time
(c) Study 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.

- We create an interval tree. Each node will contain a class and meeting and hold the interval of time needed for each one
    - To take into account how much time is being spent, we will augment the interval nodes with a variable called size.
    - Size will contain the total amount of time passed within the left subtree and the parent node.
        - In other words, size represents the amount of time passed from the lowest point in time up to the current time in our node
    - Size = (X.int.high – X.int.low) + X.left.size
    - We're not adding X.right.size to the parent node because we want to keep track of how much time is being spent up until to the time of the parent node.
        - Nodes in the right subtree count the time spent after the parent node
- Because you can't be in two meetings or classes at the same time, I'm assuming that there is no overlap allowed between any two nodes

- Add_Meeting(m, s, e)
    - Inserting a meeting will work exactly the same as a standard red-black tree insertion
    - The difference is we'll be using s as x.low and making comparisons based on that
        - In addition, because we're calculating for any collisions based on time, x.high and e will represent the time when a meeting ends
    - We will traverse and insert based on x.low and s
    - After inserting a new node and doing any rotations if necessary, we can recalculate X.size = X.left + X.right + (x.int.high – x.int.low) in O(1) time
    - Inserting a new meeting will still just take O(log n) time to travel down the Interval tree to find the proper place to insert the new node
- Check_Schedule(s, e)

  - Since we're checking based on starting and ending time, will behave similarly to interval-search
  - We would have to travel down the interval tree, searching for an interval that overlaps with (s, e)
  - If s == X.int.high or e == X.int.low, then this doesn't count as an overlap
    - Instead, we'll check for the other end of the interval and see if the other end intersects with another interval in the child subtrees
    - If there is no intersection and we reach the predecessor or successor interval, then there is no overlap
  - It would take O(log n) time because the time needed to confirm that no intervals overlap would terminate once we reach the final level of the tree and T.nil
- Study time(s, e)
  - Determines the amount of free time we have between meetings
  - We're not concerned with how many meetings there are, only how much time is spent on meeting in general
  - We can augment an interval tree with their size similar to augmented Red-Black trees
    - Size will be represented as how much time is spent from each interval
    - Parent nodes will add up the size from their left subtree to count the total amount of time spent between multiple meetings
  - (s – e) represents the total amount of time spent from s to e, which we can use to represent the total free time we have
    - We can search for the closest node for s and the closest node for e and use their augmented size to identify where the lowest node interval and the highest node intervals are located
    - When we find the interval closely matching s, then we take the largest parent that would contain s and e and subtract our total time from (Parent.size – X.size)
      - This is because X contains time spent for a time interval outside of (s, e)
      - (Parent.size – X.size) represents the time spent between X and the parent
      - (s – e) – (Parent.size – X.size) represents the amount of free time we have between X and the parent
    - Same procedure will be used for e but it will use (s – e) – (Y.size) because our size does not count the right subtree
      - Y.size already represents the time spent from Parent to Y
      - (s – e) – (Y.size) represents the amount of free time we have from Parent to Y
    - Because we're searching for the lowest node and the highest node closest to s and e, it would take O(2 log n) time to find the closest intervals which is still O(log n) time

Brandon Vo


[11, 12]
Size = 4
[9, 10]
Size = 3
[20, 21]
Size = 3
[8, 9]
Size = 2
[10.11]
Size = 1
[18, 19]
Size = 2
[23, 24]
Size = 1
[1, 2]
Size = 1
[13, 14]
Size = 1

Brandon Vo

```
//Insert a new meeting at O(log n) time
Add_Meeting(T, m, s, e)
Y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    //Check for the 3 cases

    if s < x.low       //Changed to use the low value
        x.size = x.size + (e-s) //Update the size based on the additional time spent
        x = x.left
    else x = x.right
m.p = y
if y == T.nil
    T.root = m
Else if s < y.low  //Changed to use the low value
    y.left = m
else y.right = m
m.left = T.nil
m.right = T.nil
m.color = RED
m.int.low = s             //Time when it begins
m.int.high = e            //Add the time when it ends
m.size = (s – e)  //Add the total time spent
RB-INSERT-FIXUP(T, m)
```

Brandon Vo

```
Check_Schedule(T, s, e)
X = T.root
while x ≠ T.nil
//For this function, overlap will not include scenarios where s == x.int.high and e == x.int.low
    if (s, e) overlaps with (x.low, x.high)
        Return false
    Else if s == x.high
        If X.right == Nil Return True
        Else
            X = X.right
            While X.left ≠ Nil        //Check the successor node for overlap
                If E overlaps with (X.low, X.high)
                    Return false
                X = X.left
            Return True

    Else if e == x.int.low
        If X.left == Nil Return True
        Else
            X = X.left
            While X.right ≠ Nil       //Check the predecessor node for overlap
                If s overlaps with (X.low, X.high)
                    Return false
                X = X.right
            Return True               //No overlap found
    Else
        If s > x.low    X = X.right   //Located in the right subtree
        Else    X = X.left            //Located in the left subtree
```

Brandon Vo

```
// Determine how much free time is between s and e in O(log n) time.

Study_time(T, s, e)

//First, we find the lowest node closest to s

MIN = 0

Parent = X        //Used to keep track of the largest parent node that holds s and e's respective nodes in
its subtree

Total = s – e     //Measure the amount of free time as the total amount of time between out interval


//Must find the lowest x.low that is as close to s as it could be

while x ≠ T.nil

    if (s is between x.low and x.high)          //We've found a node where s is located in

        Total = Total – (Parent.size – X.size)

//Count the time between the last node and the highest parent node

//This gives us the total time from our X node to our Parent node

        Break

    Else if s == X.high          //We can't include X

        Total = Total – (Parent.size - X.right.size)

        Break

    Else if   X.left ≠ NIL and |X.left.low – s| < |X.low – s|

        X = X.left

    Else if   X.right ≠ NIL and |X.right.low – s| < |X.low – s|

        Parent = X

        X = X.right

    Else      //We're as close as we can get to s

        Total = Total + |X.low – s| - (Parent.size - X.size)

//There's a gap of free time which we need to add back to the total

        Break
```

Brandon Vo

```
//Now, we find the closest node that matches e
Y = T.root
While Y ≠ T.nil
        If (e is between x.low and x.high)
                Total = Total – Y.size
                Break
        Else If (e == x.low)
//The node we're in shouldn't be counted, count the left subtree instead
                Total = Total - Y.left.size
                Break
        Else if Y.left ≠ NIL and |Y.left.high – e| < |Y.high – e|
                Y = Y.left
        Else if Y.right ≠ NIL and |Y.right.high – e| < |Y.high – e|
                Y = Y.right
        Else    //We're as close as we can get to a node with e
                Total = Total + |e - Y.high| – Y.size  //There's a gap of free time which we need to add
back to the total
                Break
Return Total
```

Brandon Vo

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

Design an algorithm for an augmented Red-Black tree that allows a node to store the black-height and the total number of red nodes within its subtrees.