Solutions for HW5 - CS 6033 Fall 2021

Q1 → Statistic Tree - Select

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

(a) OS-SELECT(T.root, 18)
We begin with x as the root, whose key is 26, and with i = 18. Since the size of 26's left subtree is 12, its rank is 12 + 1 = 13. So, we know that the node with rank 18 is the 18 - 13 = 5th smallest element in 26's right subtree. Then we come to the next recursive call OS-SELECT(x.right, 18-13).

(b) OS-SELECT(T.root.right, 5)
Now x is the node with key 41, and i = 5. Since the size of 41's left tree is 5, r = 5 + 1 = 6 > 5, so we come to the next recursive call OS-SELECT(x.left, 5).

(c) OS-SELECT(T.root.right.left, 5)
Now x is the node with key 30. Since the size of 30's left tree is 1, 1 + 1 = 2 < 5, we come to next recursive call OSSELECT(x.right, 5 - 2).

(d) OS-SELECT(T.root.right.left.right, 3)
Now x is the node with key 38 and i = 3. Since the size of 38's left tree is 1, r = 1 + 1 = 2 < 3, we come to next recursive call OS-SELECT(x.right, 3 - 2).

(e) OS-SELECT(T.root.right.left.right.right, 1)
Now the x is the node with key 39, r = 1 = i, so return this node, this is the 18th key.

Q2 → Statistic Tree - Rank

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.

We begin with the target node, whose key is 38, compute r = 38's left size + 1 = 1 + 1 = 2. Then set y point's to the target node, and then we come into the while loop.

(a) Because the node 38 is node 30's right child: r = r + "30's left size" + 1 = 2 + 1 + 1 = 4, and set y as node 38's parent (node key = 30). Then, we come into the next loop.
(b) Because node 30 is node 41's left child. we don't need to change the value of r and set y as node 30's parent (node key = 41). Then, we come into the next loop.
(c) Because the node 41 is node 26's right child: r = r + "26's left size" + 1 = 4 + 12 + 1 = 17. Then, we come into next loop, set y as node 41's parent (node key = 26).

Because node y (node key = 26) is the root, the loop ends. Return the r. So the rank of key 38 is 17.

Q3 → Statistic Tree - Use

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?

Answer: The i^{th} successor of node x should have a rank equal to the sum of i and the rank of node x. So, we find the rank of node x in $O(\log(n))$ time (using $OS{-}RANK(T,x)$) and then find the element which has rank $r{=}i{+}r_x$ (using $OS{-}SELECT(T.root,r)$), where r_x is the rank of node x. Since both methods take $O(\log(n))$ the total runtime of $OS{-}SUCCESSOR(T,x,i)$ should be $O(\log(n))$.

```
OS−SUCCESSOR(T,x,i):
    r = i + OS−RANK(T,x)
    if r ≥ T.root.size
        return None
    else
        return OS−SELECT(T.root,r)
```

Q4 → Statistic Tree - Variant

4. In class we augmented the red-black tree so each node included it's 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?

Answer: In this new augmentation, we store the rank of the element in its subtree (which is *x.sub_rank* =number of nodes in the left subtree+1) instead of the size of the element (which is *x.size* =*x.left.size*+*x.right.size*+1).

•Show how to maintain this information during insertion

During insertion, we need to update the value of *sub_rank* as follows (changes to the code have been highlighted):

RB−INSERT(T, z):

…

…

if z.key < *x.key*

x.sub_rank += *1*

x = *x.left*

…

…

…

z.sub_rank = *1*

RB−INSERT−FIXUP(T, z)

LEFT−ROTATE(*T*,*x*):

…

…

…

y.sub_rank =*y.sub_rank* + *x.sub_rank*

RIGHT−ROTATE(*T*,*x*):

…

...
...

y.sub_rank =y.sub_rank − x.sub_rank

The value of *x.sub_rank* will not change as the left subtree of node *x* does not change in either rotation.

•Write the methods *OS−SELECT*(*x*,*i*)

```
OS−SELECT(x,i):
        if i == x.sub_rank return x
        elseif i < x.sub_rank return OS−SELECT(x.left,i)
        else return OS−SELECT(x.right,i−x.sub_rank)
```

•Write the methods *OS−RANK*(*T*,*x*)

```
OS−RANK(T,x):
        rank_of_node = x.sub_rank
        current_node = x
        while current_node != T.root:
                if current_node == current_node.p.right:
                        rank_of_node += current_node.parent.sub_rank
                current_node = current_node.parent
        return rank_of_node
```

•State the worst-case run time of inserting an item, OS-SELECT, and *OS−RANK* using Theta notation.

The run time for insertion will remain $\Theta(\log(n))$ as our additions take constant time and will not impact the run time. *OS−SELECT* also takes $\Theta(\log(n))$ as we have only slightly changed the structure. *OS−RANK* is also $\Theta(\log(n))$ as we start at a node in the tree, and traverse up till the root, in a singular path.

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?

On comparing the runtimes of the methods in both cases, we see that there is no obvious improvement in one case over the other. So, both approaches have the same efficiency.

Q5 → B Tree - maximum and predecessor

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.

Finding the maximum key in a B-tree is quite similar to finding maximum key in a BST. We need to find the right most leaf for the given root and return the last key.

```
B-TREE-DIND-MAX(x)
    if x == NIL
        return NIL
    else if x.leaf # this is the left most node
        return  x.key[n] # return the last key
    else:
        DISK-READ(x.c[n+1]) # go to next level
        return B-TREE-FIND-MAX(x.c[n+1])
```

Finding the predecessor of a given key $x.key_i$ is according to the following rules:
a. If x is not a leaf, return the maximum key the subtree rooted at $x.c_i$
b. If x is a leaf and $i > 1$, return the $((i-1)^{th}$ key of x
c. Otherwise, we need to look for the last node (from the bottom up): z is current target node($x.key_i$ is the smallest key in the subtree root at z) , y is the parent of z.
When $j > 1$, the last node is y, and $x.key_i$ is the leftmost key in tree root at $y.c_j$, so we can return y.$key[j-1]$.
When $j = 1$, means current node z is still the first child of y, so last node should be in upper level. So we need go up one level. And if there is no more level upper, (y is the root of the tree), $x.key_i$ is the smallest key in the whole tree, so we return NIL.

```
B-TREE-FIND-PREDECESSOR(x, i)
```

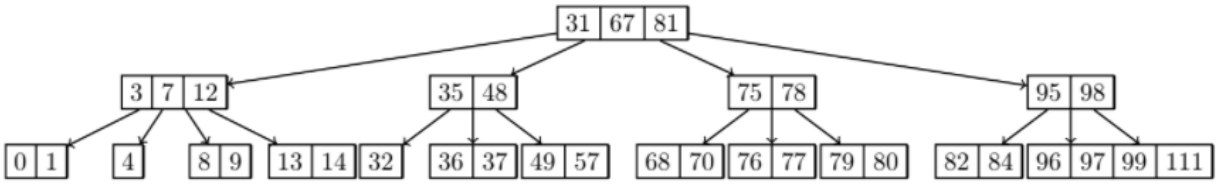
```
if x.leaf = False: # x is not a leaf
    DISK-READ(x.c[i])
    return B-TREE-FIND-MAX(x.c[i])
else if i > 1 # x is a leaf and i > 1
    return x.key[i-1]
else # x is a leaf and i = 1
    z = x
    while True # we are looking for the largest key that  before node z
        y = z.p
         j = 1
         DISK-READ(y)

         while y.c[j] != x
            j = j + 1

        if j == 1  # z is still the first node, go back to higher layer
           if y.p = NIL: # y is root, no more upper level
                return NIL
            z = y # go upper a level
         else
            return y.key[j-1]
```

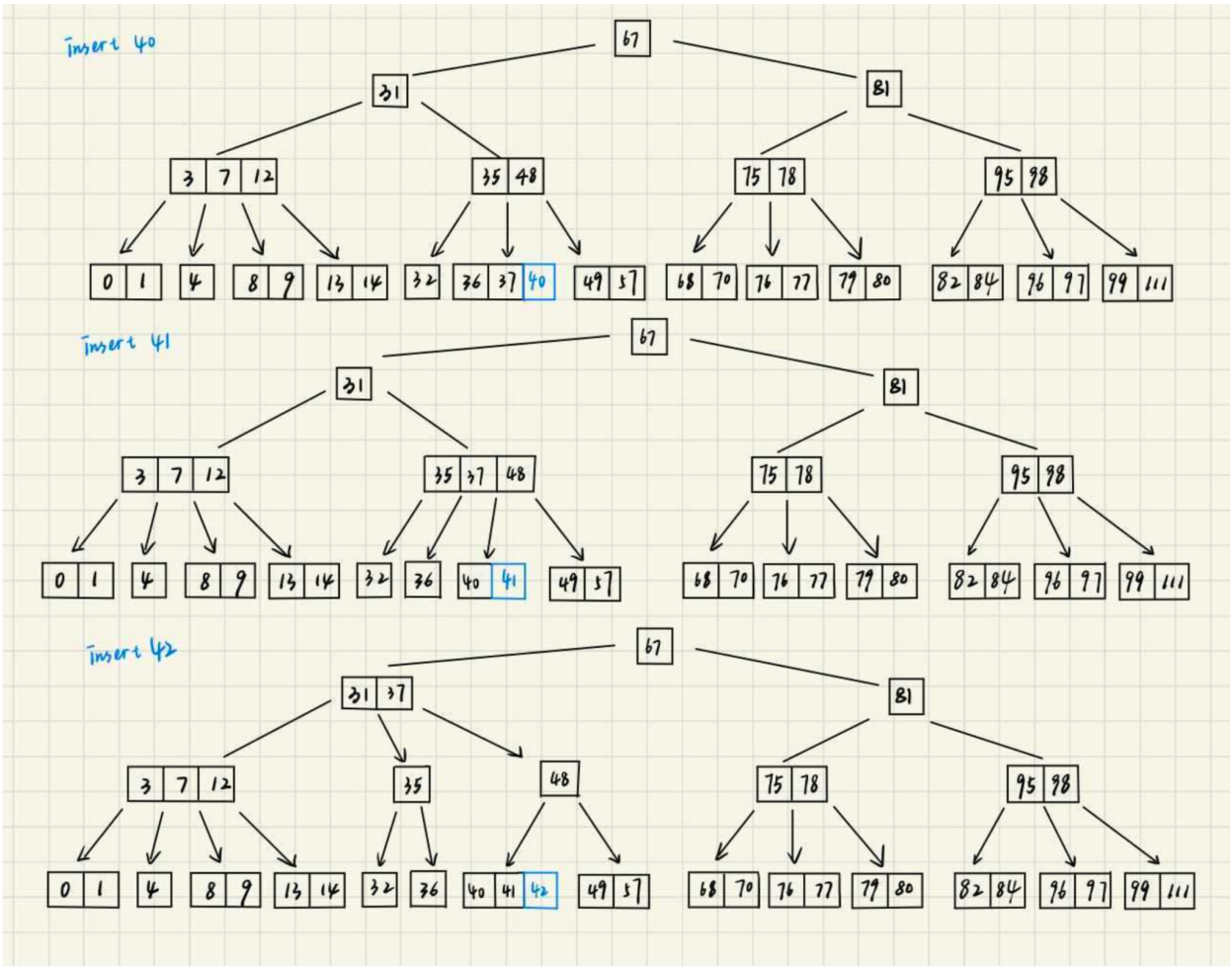
For the running time, since we may need to check every level to find the predecessor, the CPU time is O(h). The number of disk access is also O(h).

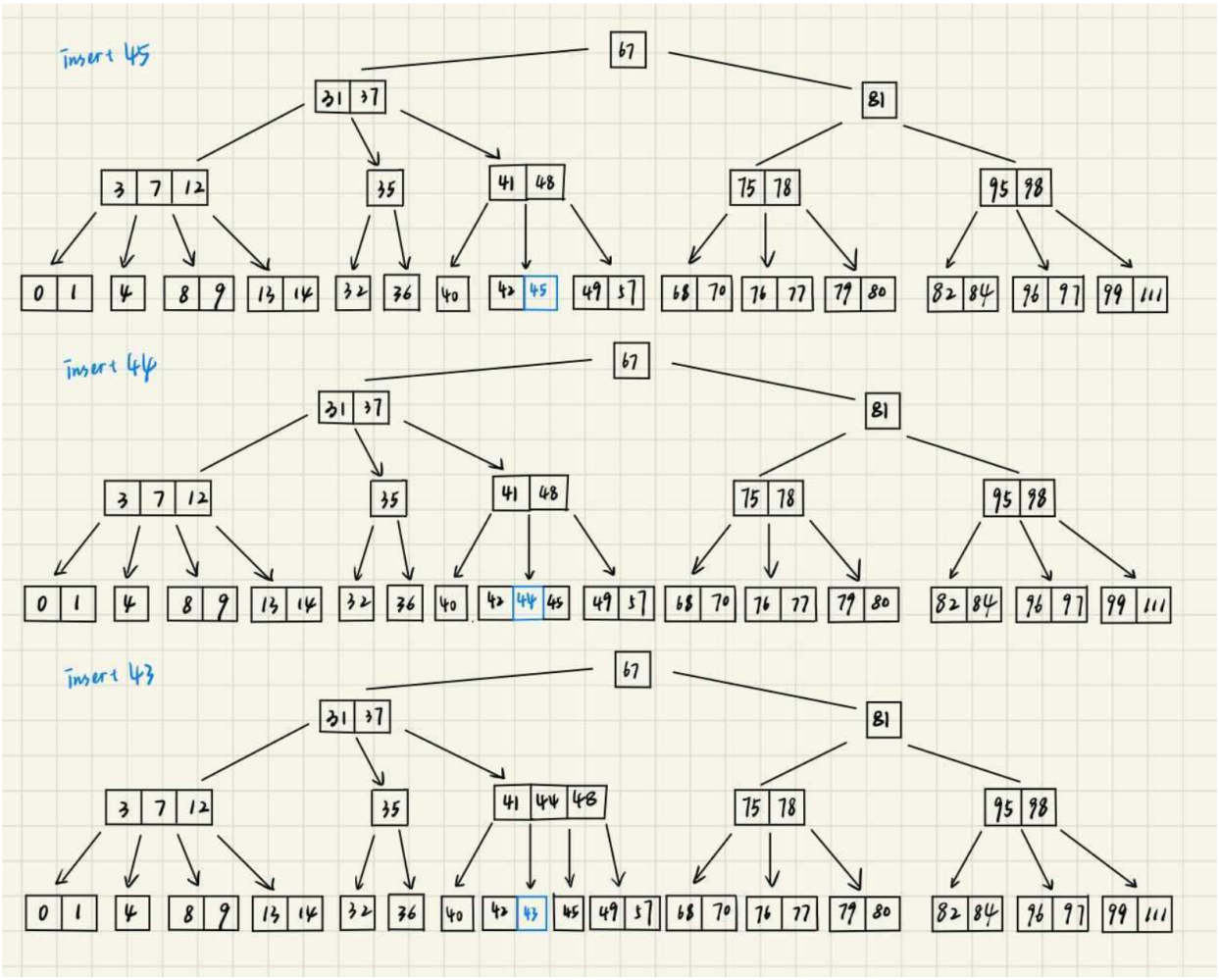
Q6 → B Tree - Insertion

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.)




Insert 40
67
31
81
3 7 12
35 48
75 78
95 98
0 1
4
8 9
13 14
32
36 37 40
49 57
68 70
76 77
79 80
82 84
96 97
99 111
Insert 41
67
31
81
3 7 12
35 37 48
75 78
95 98
0 1
4
8 9
13 14
32
36
40 41
49 57
68 70
76 77
79 80
82 84
96 97
99 111
Insert 42
67
31 37
81
3 7 12
35
48
75 78
95 98
0 1
4
8 9
13 14
32
36
40 41 42
49 57
68 70
76 77
79 80
82 84
96 97
99 111



FYI: the insertion code is on Page 494-496 in CLRS.

Since the B-Tree used "preemptive" splits root is in the memory, we need to split the root (includes 3 writes.) And then need to search for 3 nodes, finally we need to write the leaf that we inserted our key back into the disk(1 writes). So there are 7 accesses in total.

Q7 → R-B Tree application

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 $[1 \ldots N]$ 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 will be using a Red-Black Tree with the following augmentations to each node x:

- x.return → Represents the return of the stock at node x.
- x.total → Represents the total return in the subtree rooted at x.
- x.size → Represents the total items in the subtree rooted at x.
- x.predecessor → It is a pointer to the node that would come before x when it is arranged in order.

For T.nil, x.total = 0, x.size = 0 and x.predecessor = T.nil.

(a) We will add a couple of lines of code to the existing INSERT and LEFT-ROTATE code to make it work with our augmentation.

```
INSERT(T, x):
    ... // Remains the same
    x.size = x.left.size + x.right.size + 1
    x.total = x.left.total + x.right.total + x.return
    pre = x
    while pre ≠ pre.p.right and pre ≠ T.nil:
        pre = pre.p
    x.predecessor = pre.p
    ... // Remains the same
LEFT_ROTATE(T, x):
    ... // Remains the same
```

```
    y.size = x.size
    x.size = x.left.size + x.right.size + 1
    y.total = x.total
    x.total = x.left.total + x.right.total + x.return
    pre = x
    while pre ≠ pre.p.right and pre ≠ T.nil:
        pre = pre.p
    x.predecessor = pre.p
    pre = y
    while pre ≠ pre.p.right and pre ≠ T.nil:
        pre = pre.p
    y.predecessor = pre.p
```

(b) We will start at the maximum value node of the tree and work our way backward in the tree using the predecessor of each node and add it into our final array until it is size B. It will take us O(log N) to get to the maximum value node and O(1) to add each node to our array. Since we have to add B nodes, it will take us O(B) to add the nodes. This leads to our runtime of O(B + log N). For the extra credit, we make an augmentation to the tree itself by storing a pointer to the maximum value node of the tree. That eliminates the need to do the O(log N) walk that we do in our regular solution and transforms it into an O(1) operation.

```
TOP_STOCKS(T, B):
    arr = [] // Array to return
    x = T.root
    while x.right ≠ T.nil: // To find the max value stock
        x = x.right
    for a = 1 to B: // To append all the required stocks
        arr.append(x)
        x = x.predecessor
    return arr
```

(c) Since we only care about the top B stocks, it means that the stocks should exist on the right subtree. The important part is figuring out when should we traverse the left subtree. That only occurs when B > ceil(N/2). When it does, we update how many stocks should we check for in the left subtree since all the nodes in the right subtree and the current node have been accounted for. If we do not have to go down the left subtree, we might have to go down the right subtree. Since we are choosing a path to be just once per level, the time complexity will be O(log N).

```
AVERAGE_RETURN(T, B):
    total_return = T.root.total // Sum of returns for all stocks
    Q = new queue() // Queue to keep track of nodes to check
    Q.enqueue(T.root)
    num_stocks = B // Tracker to check what number of stocks to consider
    while Q is not empty:
        x = Q.dequeue()
```

```
        if x.size == num_stocks: // Entire tree is in range
            pass
        else if x.right.size ≤ num_stocks: // Entire left subtree and
current node not in range
            total_return = total_return - x.return - x.left.total
            Q.enqueue(x.right)
        else if x.right.size + 1 == num_stocks: // Entire left subtree not
in range
            total_return = total_return - x.left.total
        else: // Some some nodes of the left subtree are in range
            num_stocks = num_stocks - (x.right.size + 1)
            Q.enqueue(x.left)
    return total_return / B
```

Q8 → Augmented R-B Tree

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 will be using a Red-Black Tree with the following augmentations to each node x:

- x.int → Represents the interval for that node. It will have a low and high value stored as x.int.low and x.int.high respectively. The key of the tree will be x.int.low
- x.max → Represents the maximum value of any interval endpoint stored in the subtree rooted at x.
- x.min → Represents the minimum value of any interval endpoint stored in the subtree rooted at x.
- x.total_time → Represents the total time spent in meetings for all given intervals stored in the subtree rooted at x.

For T.nil, int = [0,0]; max = 0, min = 0 and total_time = 0.

(a) We will add a couple of lines of code to the existing INSERT and LEFT-ROTATE code to make it work with our augmentation.

```
INSERT(T, x):
    ... // Remains the same
    x.max = max(x.int.high, x.left.max, x.right.max)
    x.min = min(x.int.low, x.left.min, x.right.min)
    x.total_time = x.left.total_time + x.right.total_time + (x.int.high -
x.int.low)
   ... // Remains the same
LEFT_ROTATE(T, x):
    ... // Remains the same
    y.total_time = x_total_time
    x.max = max(x.int.high, x.left.max, x.right.max)
    x.min = min(x.int.low, x.left.min, x.right.min)
    x.total_time = x.left.total_time + x.right.total_time + (x.int.high -
x.int.low)
    y.max = max(y.int.high, y.left.max, y.right.max)
    y.min = min(y.int.low, y.left.min, y.right.min)
```

Since all the lines of code that have been added run in O(1) time, the running time of INSERT will still be O(log n).

(b) The search for an interval that overlaps with the interval [s, e] starts with x at the root of the tree and proceeds downward. It terminates when either it finds an overlapping interval or x points to the sentinel T.nil. Since each iteration of the basic loop takes O(1) time, and since the height of an n-node red-black tree is O(log n), therefore, the check_schedule function runs in O(log n).

```
CHECK_SCHEDULE(T, s, e):
    x = T.root
    while x ≠ T.nil and s ≥ x.int.high and x.int.low ≥ e:
        if x.left ≠ T.nil and x.left.max ≥ s:
            x = x.left
        else x = x.right
    if x == T.nil return True
    else return False
```

(c) We keep track of a running total of the time spent in meetings and subtract nodes as well as their children's trees if they don't fit our given interval. If the entire subtree of a child node is within range, we don't need to go down that path. We only need to traverse down the path where nodes may lie in range. Therefore, we may have to traverse the tree at every level for a maximum of 2 nodes per level in the worst case. Therefore, the running time will be O(log n).

```
STUDY_TIME(T, s, e):
    meeting_time = T.root.total_time // sum of time to be spent in meetings
    Q = new queue() // Queue to keep track of nodes to check
```

```
    Q.enqueue(T.root)
    while Q is not empty:
        x = Q.dequeue()
        if s ≥ x.int.high or x.int.low ≥ e: // Checking if we are out of
interval
            meeting_time = meeting_time - (x.int.high - x.int.low)
        // The node is in the interval
        if (s ≥ x.left.max or e ≥ x.left.min) and x.left ≠ T.nil: // The
entire left tree is out of range
            meeting_time = meeting_time - x.left.total_time
        else if s ≥ x.left.min and x.left.max ≥ e and x.left ≠ T.nil: //
The entire left tree is in range
            pass
        else if x.left ≠ T.nil: // Some of the nodes of the left tree are
in range
            Q.enqueue(x.left)
        if (s ≥ x.right.max or e ≥ x.right.min) and x.right ≠ T.nil: // The
entire right tree is out of range
            meeting_time = meeting_time - x.right.total_time
        else if s ≥ x.right.min and x.right.max ≥ e and x.right ≠ T.nil: //
The entire right tree is in range
            pass
        else if x.right ≠ T.nil: // Some of the nodes of the right tree are
in range
            Q.enqueue(x.right)
    return (s - e) - meeting_time
```