3-1 a. What data structures covered in lectures should be used for the range index. Choose the simplest.

1. Min-Heap 2. Max-Heap 3. Binary Search tree 4. AVL Trees 5. B-Trees

Binary Search Tree since you can visit the min & max values in $\Theta(\log n)$ time

AVL is a balanced BST $< \Theta(2 \lg n)$ lec 6

b. How much time will it take to insert a key in the range index?

1. $O(1)$ 2. $O(\log(\log(N)))$ 3. $O(\log N)$ 4. $O(\log^2 N)$ 5. $O(\sqrt{N})$

Lec 6 says insert is $\Theta(\lg(n))$

c. How much time will it take to find the min key in the range index?

1. $O(1)$ 2. $O(\log(\log(N)))$ 3. $O(\log N)$ 4. $O(\log^2 N)$ 5. $O(\sqrt{N})$

Lec 6

d. Time to find the max key in range index

(H) $(\lg n)$ lec 6 notes

e. Assuming $l < h$, and both l and h exist in the index, COUNT(l, h) is

1. rank(l) − rank(h) − 1 2. r(l) − r(h) 3. r(l) − r(h) + 1 4. r(h) − r(l) + 1

5. r(h) − r(l) 6. r(h) − r(l) + 1 7. r(h) + r(l) − 1 8. r(h) + r(l) 9. r(h) + r(l) + 1

rank l = k → $k < h$

rank(h) − rank(l) = k → $l \le k < h$

rank(h) − rank(l) + 1 = k → $l \le k \le h$

f. Assuming $l < h$ and h exists in the index, but l does not COUNT(l, h) is

$l \le k \le h$

rank(h) = k → $k \le h$

rank(h) − rank(l) = $l \le k \le h$ since rank(l) counted l last time & not this time

$l \le k \le h$

g. Assuming $k < h$, and l exists in the index, but h does not
COUNT(l,h) is ✓

rank(h) counts l, but we don't want to include l in our final number
$rank(h) - (rank(l) - 1) = rank(h) - rank(l) + 1$
↑ to remove the bias from counting l

h. Assuming $k < h$, and neither l nor h exist in the index, COUNT(l,h) is

$rank(h) - rank(l)$
$\le \rightarrow <$ when the indexed term is absent

i. In order to respond to RANK() queries in sub-linear time, each node node in the tree will be augmented with an extra field, node.δ. Keep in mind that for good augmentation, the extra information for a node should be computed in O(1) time, based on other properties of the node, and on the extra information stored in the node's subtree. The meaning of node.δ is
1. min key in subtree rooted at node 2. max key in subtree rooted at node
3. height of subtree rooted @ node (4) # nodes in the subtree rooted at node
5. depth of node 6. Σ(keys in subtree rooted at node)

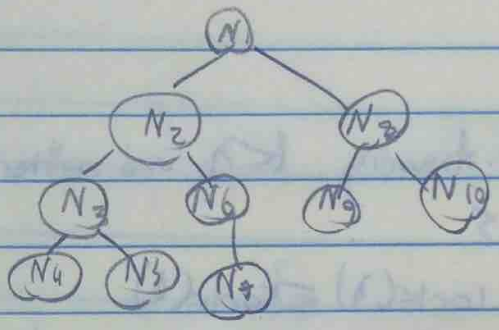
4 since it is useful for computing the ranks, and ranks cannot be computed using the provided info, but ranks can be sped up using 4.

j. How many extra bits of storage per node does the augmentation above require?

1. O(1) 2. O(log(log N)) → 6. O(N)

the whole tree

O(log(N)) since top node. 0 has N in its subtree, and that is the largest value that can be returned. It takes $\log_2 N$ bits to store the numbers 0–N.



The following questions refer to the tree below

k. $N_4.\delta$ is

1. 0 2. 1 3. 2 4. key of N_4

the node itself

l. $N_3.\delta$ is

1. 1 2. 2 3. 3 4. key of N_4 5. key of N_5 6. Σ keys $N_3 \ldots N_5$

m. $N_2.\delta$ is 1. 2 2. 3 3. 4 4. 0 5. key of N_4 6. key of N_7 & keys

n. $N_1.\delta$ is
1. 3 2. 6 3. 7. (4.) e 5. key of N_1 6 key of N_8
7. Σ keys $N_1 \ldots N_{16}$

o. Which of the following functions need to be modified to update δ? If a function does not apply to the tree for the range index, it doesn't need to be modified.

Rotate-Right & Rotate-Left are the functions used to insert, & delete in the tree, so if we get these two to update δ, we won't need to update the hieghts of subtrees, or δ. Rebalance also directly changes the subtree heights, so δ must be updated there as well

1. INSERT F
2. DELETE F
3. ROTATE-LEFT T
4. ROTATE-RIGHT T
5. REBALANCE T
6. HEAPIFY F

p. What is the running time of a COUNT() implementation based on RANK()?

$T(\text{COUNT}) \approx T(\text{RANK}) + T(\text{RANK})$
$= 2\,T(O(\log(N))) = \boxed{O(\log(N))}$

$\log(N)$ calls to $\cdot\delta$, on the path from root to bottom leaf on a path

q. LCA means

Lowest common ancestor

r. running time of LCA(a,b) for the trees used by the range index is

1. O(1) ... 5. Θ(√N)

O(log(N)) since worst case scenario it descends to the bottom of the tree, which is height log(N)

s. Assuming ADD-key runs in O(1) time, and that LIST returns a list of L keys, the running time of NODE-LIST when called on line 3 of LIST is

Worst case scenario NODE-LIST descends the height of the tree, O(log(N)) add O(L) to return the nodes from the list

4. Assuming that ADD-KEY runs in O(1) time, and that LIST returns a list of L keys, the running time of LIST is (.OC1) Z ... 10.

$T = LCA + NODE\text{-}LIST$

$= O(\log(N)) + O(\log(N)) + O(L)$

$= 2O(\log(N)) + O(L)$

$= O(\log(N)) + O(L)$

4. Prove that LCA is correct.

LCA returns the ~~least~~ greatest common ancestor if it exists, or NIL if it doesn't

Case 1: Suppose the LCA exists

Then $\exists$ a node $A \neq A$ is the deepest node that has q and h as descendents from the definition of the lowest common ancestor.

Suppose that LCA returns a node $B \neq A$

then $q \leq B.key \leq h$, which is the condition for returning B in the LCA algorithm

and A must be lower than B since it is the LCA

since the AVLs are balanced BSTs we know A must be in the subtree rooted at B.

Case 1: A is in the subtree rooted at B.left

This leads to a contradiction since all elements in B.left $< B.$

so for A to be in B.left $h \not< B.key$, violating the condition for

Case 2: Symmetric about $q >$

So when a lowest common ancestor exists it is returned

Case 2: LCA does not exist

$\Rightarrow \nexists$ node $A \neq l \leq A.key = h$

So either C1 $l < X.key$ for $X \in Nodes$ or (C2 $h > X.key$ for $X \in Nodes$

These are the conditions for returning a non NIL node, so one will not be returned.

If C1 is true the LCA will return left node, working the bottom node, when node = node.left since $l < node.key$ still, and the left of a ~~bottom~~ node w/ no children is NIL

For C2 similar reasoning with $>$ and right node. Then NIL is returned

So LCA is correct

3-2a. Run the code under the python profiler with the command below, and identify the method that takes up most of the CPU time. If two methods have similar CPU usage times ignore the simple one.

running

intersects

b. How many times is the method called?

18,759,314

c. The x coordinates of points of interest in the input are (T/F)

1. x coordinates of the left endpoints of horizontal wires		T
2, * c	right	T
	mid points	F
	where — cross	F
	Vertical wires	T

The endpoints of wires are points of interest, since these are the inputs

d. When the sweepline hits the x coordinate of the left endpoint of a horizontal wire.

1. wire is added to range index 2. wire is removed from the range index 3. range query is performed 4. nothing happens

1. the wire is added to the range index since the wire is inside darker after the left endpoint is reached

e. When the sweep line hits the x-coordinate of the right endpoint of a horizontal wire

1. wire ..

It is removed, since the line returns to its original size/color

f. When .. mid point

..

Nothing happens, the sweepline doesn't even stop at midpoints

g. .. Vertical wire

..

A range query is performed. I chose this since it is the only remaining unused solution.

h. What's a good invariant for the sweep-line algorithm?

1. range index holds all the horizontal wires to the left of the sweep line
2. " " stabbed by "
3. " " right of "
4. " wires " left of "
5. " " right "

Since the sweepline moves on when all stabbed points have been visited.

i. When a wire is added to the range index, what is its corresponding key?

1. x coordinate of wire's midpoint 2. y 3. segment's length
4. x coordinate of the pt that will remove the wire from the idx

Horizontal wires are added to the range index (constant y value)

j. Run your modified code under the python profiler again, using the same test case as before, and identify the method that takes up the most CPU time.

count

k. How many times is the method called?

20,000

Q. Modify circuit2.py to implement a data structure that has better asymptotic running time for the operator dove. Keep in mind the tool has 2 usage scenarios:

In problem 1 we noted that AVL trees should be used for this.

See circuit2.py