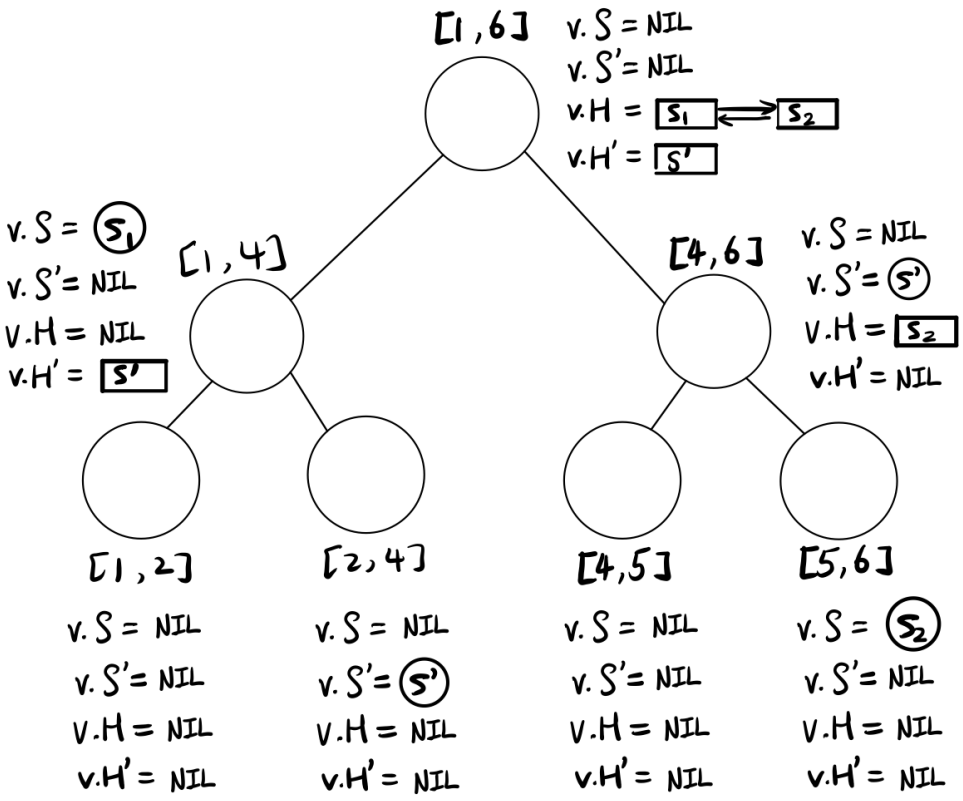
CSC265 Homework 3 | Fall 2021
Name: Haojun Qiu | Student Number: 1006900622 |
Who I discussed with: Runshi Yang, Ted Deng, Darcy Wang

Q1

I denote the interval they represents using mathematical interval notation instead of attributes $v.l$ and $v.r$.



Q2

I will use the fact known from observation that, nodes in T at the same depth, represents a sequence of intervals that is **sorted** increasingly by entire interval, from left to right. For example, for any nodes v_1, v_2 at the same depth of tree T such that v_1 is positioned in the left side of v_2, then $v_1.r \leq v_2.l$.

(a) Upper bound on number of nodes v contains an arbitrary $s \in S$ in $v.S$:

Let $s \in S$. There are maximum two nodes v in the same depth of tree T such that s is a node in $v.S$, for any depth of the tree.

Suppose for a contradiction that there are three or more nodes at the same depth such that node s in their S attributes. Among them, denote the node representing leftmost interval and rightmost interval as L, R, respectively; and fix M to be arbitrary one such that $L.r \leq M.l < M.r \leq R.l$ (i.e., M is bounded between L and R). By definition of $v.S$ we know that none of them are siblings of each other, since otherwise they contains the interval their parent node represents thus would not be in their S attribute. That is, they have different parent nodes, which implies that M's parent node P represents the interval that is also bounded between L and R (since it is bounded by the intervals their parents who has the same depth with P). Then, s contains intervals L and R represents implies it contains the interval P represents, resulting the contradiction that s in $M.S$. Hence we conclude there are at most 2 nodes at the same depth such that node s in their S attributes.

Hence, we conclude that the upper bound on the maximum number of nodes v in T such that s is a node in $v.S$, is $O(h)$, where h is the height of T.

(b) Lower bound on number of nodes v contains an arbitrary $s \in S$ in $v.S$:.

The intuition is that, we make s's projection onto x-axis as big as possible but not covering the entire interval so that node s not in $T.root.S$. Consider T to be a full binary tree with height h, where $X = \{x_1, x_2, \ldots, x_{i-1}, x_i\}$ is as defined with T (as in question) to represents the intervals of all leaves represents, and s to be the line segment such that $s.l = x_2$ and $s.r = x_{i-1}$, i.e., the projection of s onto x-axis is $[x_2, x_{i-1}]$. An example of such (T, s) pair with height 3 is in Figure 1 (a), where I highlighted nodes v such that $v.S$ contains node s.

Notice that its projection contains $[x_2, x_3], [x_3, x_4], \ldots, [x_{i-2}, x_{i-1}]$, so s would contain the node representing $[x_2, x_3]$ and its uncle, and its uncle's uncle, $\ldots$, and $[x_{i-2}, x_{i-1}]$ and its uncle and its uncle's uncle, until the node whose grandparent is the root. Those are nodes in T whose S attribute contains node s. By the definition of data structure it is all nodes v in T such that node s in $v.S$, which is in total $2(h-1)$ nodes and so in $O(h)$ as desired.

(c) Upper bound on number of nodes v contains an arbitrary $s \in S$ in $v.H$:

Let $s \in S$. Similarly, I claimed that there are at most two nodes v in the same depth of tree T such that s is a node in $v.H$, for any depth of the tree.

Suppose for a contradiction that there are three or more nodes in the same depth such that node s is in their H attributes. Among them, denote the node representing leftmost interval and rightmost interval as A and C respectively, and fix B to be arbitrary one such that $A.r \leq B.l < B.r \leq C.l$ (i.e., B is bounded between A and C). It implies that there are some proper descendants of A, B, C, say L, M, R, contains node s in their S attributes (i.e., node s is in $L.S$, $M.S$, and $R.S$). Since L and R are descendants of A and C. We know that $L.r \leq A.r$ and $C.l \leq R.l$, hence $L.r \leq B.l < B.r \leq R.l$, which means that interval that B represents is bounded by which of L and R. And since node s in L and R implies it contains them, it also contains the interval set between them, B. Then, since M is the proper descendant of B, s must also contains the interval that s's parent node represents, so node s would not be in $M.S$, reaching a contradiction.

Hence, we conclude that the upper bound on the maximum number of nodes v in T such that s is a node in $v.H$, is $O(h)$, where h is the height of T.

(d) Lower bound on number of nodes v contains an arbitrary $s \in S$ in $v.H$:

Consider the same input family defined in b), the (T,s) pair with height h. Figure 1 (b) shows an example such (T, s) pair with height 3 that has $2(h-1)+1 = 2(3-1)+1 = 5$ nodes (highlighted in green) with node s in their H attributes.

We showed what node in T has node s their S attribute, hence we know that their parent node should have their H attribute contains node s, as well as the root, so there will be $2(h-1)+1$ nodes, and in $O(h)$ as wanted.

(e) Good upper bound on number of nodes the entire data structure contain:

Firstly, given there are N leaves in T, where T is a tree as specified in handout, there will be at most $2N-1$ nodes in T (with counting nodes in v' attributes where v is a node in T). This can be shown by induction on $N \in \mathbb{N}$.

Base cases: Let $N = 0$, then there is no leave so at most $0 > 2(0) - 1$ nodes. Let $N = 1$, there is one leav, by defnition of T, it must be the root, so T has 1 node, which is $2(1) - 1$.

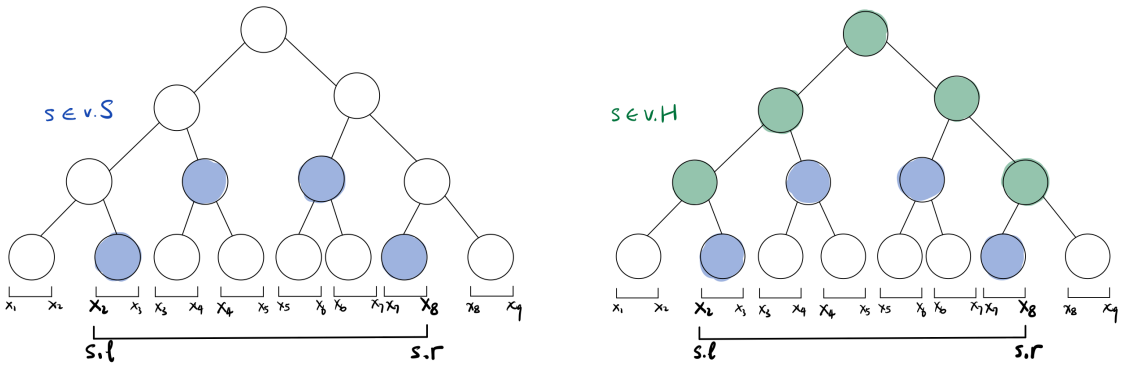
Let $N = 2$, , it has one root with two children, so in total $3 = 2(N) - 1$ nodes.

Induction step: Let $N \in \mathbb{N}$ and assume $N \geq 1$. We assume the statement works for such tree with N leaves. We show that the statement works for such tree T with $N+1$ leaves. We extract two leaves with the same parent from T (valid since $N + 1 \geq 2$) to form a new tree called T'. We know that T' has N leaves since we extract two from T and the extracted two leaves becomes a new leave, so by induction hypothesis it contains at most $2N - 1$ nodes. Hence, T will at most contain $2N - 1 + 2 = 2(N+1) - 1$ nodes, as wanted.

Since we know that T has $N - 1$ leaves, it has $2(N-1) - 1$ nodes, i.e.$|T| = O(N)$.

Furthermore, we count the sum of nodes in $v.S$, $v.S'$, $v.H$, $v.H'$ for all v in T. By the upper bound given in (a) and (c), each $s \in S$ can be in at most $O(h) = O(\log N)$ (we calculate number of nodes in the tree, and it is a red-black tree) node v's S and H attribute, $v \in T$. By the same technique we can have shown it is true for $s' \in S$, so each s' can be at most $O(h) = O(\log N)$ node v's S and H attribute, $v \in T$. So, $\sum_{v \in T} \big(|v.S| + |v.S'| + |v.H| + |v.H'|\big)$ at most is $O\big(|S|h + |S'|h + |S|h + |S'|h\big) = O(2(n+n')h) = O(2N' \log N)$ which is in $O(N' \log N)$, we know that $N \leq 2N'$ so we can also write $O(N' \log N')$. So, combing with $|T| \in$

$O(N) = O(N')$, the total number of nodes in the data structure is $O(N' \log N')$.



(a) (b)

Figure 1: Q2 b) and d) example input pair for lower bound

Q3

In include the code for main algorithm for helping my explanation. And, I will assume that for all $v.S$ $v.S'$ $v.H$, $v.H'$ we have a attribute allows getting cardinality in constant time, it helps when I need to call COUNT1 and COUNT2 from Homework 1.

Writting out function specification helps for knowing what recursion calls do exactly.

Specification: If T is in data structure given in handout and v is a node in T, then COUNT(T, v) counts all the intersections between $s \in S$ and $s' \in S'$ that happens between interval $[v.l, v.r]$, where both $s \in S$ and $s' \in S'$ (fully or partially) resides in $(v.l, v.r)$ and do NOT fully contain v's parent node's interval, if its parent $\neq$ NIL.

COUNT(T, v)

1 $L =$ COUNT$(T, v.left)$ // recurse on the left child
2 $R =$ COUNT$(T, v.right)$ // recurse on the right child
3 $E =$ EDGECASE-HELPER(v)
4 $A =$ COUNT1$(v.S, |v.S|, v.S', |v.S'|)$
5 $B =$ COUNT2$(v.H, |v.H|, v.S', |v.S'|)$
6 $C =$ COUNT2$(v.H', |v.H'|, v.S, |v.S|)$
7 **return** $L + R + E + A + B + C$

We need to justify the algorithm does what its specification says, since if it does it is obvious that the call COUNT$(T, T.root)$ will output the correct value (we observe that if s and s' partially or fully resides in $(v.l, v.r)$, it fully resides in $[v.l, v.r]$ since we assume $s.l < s.r$).

I will walk through the algorithm line by line, putting explanation of algorithm and justification of specification together, and just postpone what the EDGECASE-HELPER(v) does and why it correctly helps us return the right value.

To justify the function follows the specification, we shows that assuming recursion call follows the specification implies the main call follows the specification (kinda of induction step in a formal proof). I will not show the base case as it is more intuitively correct, than the induction step.

Firstly, by the function specification, assuming that two recursive calls returns the correct value, then L equals to the number of intersection between $s \in S$ and $s' \in S'$ that happens between $v.l$ and $v.r$, and similarly, R equals to the number of intersection between $s \in S$ and $s' \in S'$.

When we summing the two together, we have two edge cases that makes our current counting from recursion step $(L + R)$ over count some intersection and miss to count some. So, we use a helper function to resolve the issue. EDGECASE-HELPER(v) is the one for edge case counting. Let's look in detail what are the two edge cases.

- Figure 2 (a) shows when we over count the intersection. Notice that, by specification, the intersection between s and s' will be counted in recursive call COUNT$(T, v.right)$ since none of them fully contain the

interval v represents. And it is also being counted in $\text{COUNT}(T, v.right)$ for the same reason. So we count it twice. So, we need to minus 1 in total count.

- Figure 2 (b) shows when we under count the intersection. Notice that by function specification, such pair of (s, s') do not fully or partially resides in the interior the same side (any of $(v.left.l, v.left.r)$ and $(v.left.l, v.left.r)$), so both recursion call do not count the intersection. So, we need to add 1 in this case, in total count.

So, we also identify what $\text{EDGECASE-HELPER}(v)$ should do, it returns the result that helps resolve all over/under counting previously in $L + R$, when summing $L + R + E$.

What's the algorithm to identify the edge cases? We set a counter first. The we iterate through through all elements in $v.left.S$ with $v.right.S'$ (and $v.left.S'$ and $v.right.S'$). Since S is sorted by $s.b$, S' by $s'.b$ and disjoint so they are sorted at line $x = v.left.r = v.right.l$ (the mid/partitioned line) as well, and we do an in-order tree walk at parallel, comparing their the $y - value$ at $x = v.left.r = v.right.l$. If we find that a pair (s, s') has the same y-value at $x = v.left.r$, we further investigate it's case 1 or case 2, and add 1 to or minus one from the counter, correspondingly. After going through all, we have resolved all the edge case counting at one recursion level. Notice that edge case helper will be called on each node being recursed on, so we resolve all edge case counting.

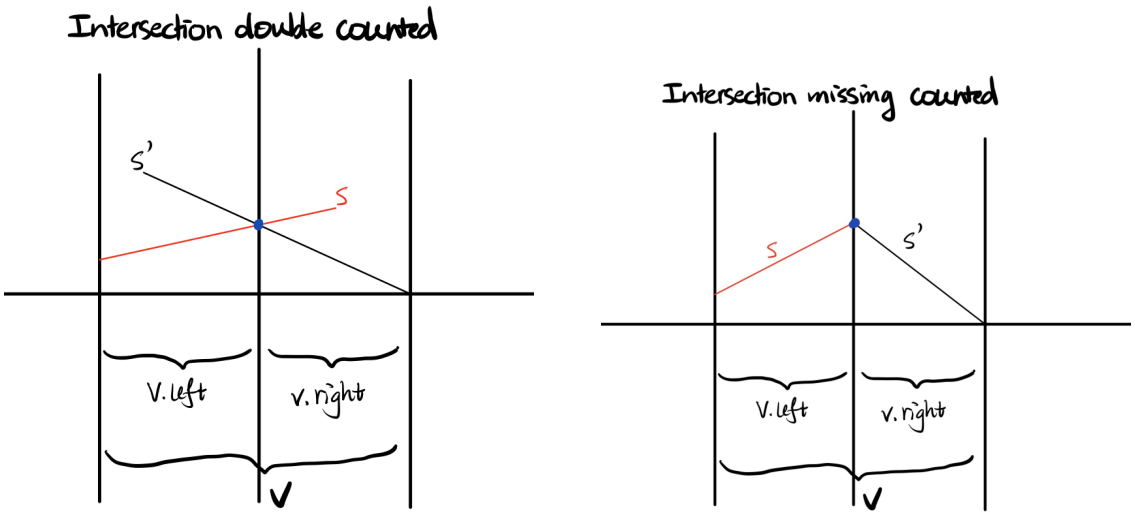
Then, what we remains to count are intersection between $v.l$ and $v.r$ between s and s' such that s fully contains $v.left$ and $v.right$'s parent, which is just v.

If a pair of s and s' fully contain $[v.l, v.r]$, then their intersection, if exists, will be counted at line 4 (since $s \in S'$ and $s' \in S'$).

If a pair of s and s' such that s fully contains $[v.l, v.r]$, but not for s', then it will be counted at line 5 (since $s \in S$ and $s' \in H'$).

If a pair of s and s' such that s' fully contains $[v.l, v.r]$, but not for s, then it will be counted at line 6 (since $s' \in S'$ and $s \in H$).

Then, as a result, the summation $L + R + E + A + B + C$ should be all intersection happens between $[v.l, v.r]$ for all $s \in S$ and $s \in S'$ such that they fully or partially resides in $(v.l, v.r)$ as wanted.



(a) (b)

Figure 2: Q3 Two edge cases counting

Running time Analysis:

Define $Y_v = |v.S| + |v.H| + |v.S'| + |v.H'|$. Notice that for every node we do operation from lines 4 - 7. Fix any node $v \in T$, lines 4 - 7 together takes at most $O(Y_v \log(Y_v))$ steps on it (since COUNT1 is bounded by

COUNT2 and runtime for COUNT2 is $O(|S_1|\log(|S_2|))$ where S_1, S_2 are input, and we know that for all input carnality from lines 7-8, they are a term in summation Y_v so less than or equal to it).

Then we know that the total running time is (I drop the big O notation at times to make calculation more clear)

$$\begin{aligned}\sum_{v\in T} Y_v \log(Y_v) &\le \max_{v\in T}\{Y_v\}\cdot\sum_{v\in T}\log Y_v \\ &\le \max_{v\in T}\{Y_v\}\cdot\log\left(\sum_{v\in T} Y_v\right) \\ &= O(N'\log N'\cdot\log(N'\log N') \\ &= O\Big(N'\log N'\cdot\big(\log(N')+\log(\log N')\big)\Big) \\ &= O(N'\log^2 N'),\end{aligned}$$

as wanted. The first line to the second in the inequality comes from the generalization of the fact that $(a\log a)+(b\log b)\le a\log(a+b)+b\log(a+b)\le(a+b)\log(a+b)$ for non-negative $a,b\in\mathbb{R}$.

Q4

We divide entire algorithm into few important steps.

1. Inserting proper nodes v in T.

Since we have add a line segment s, $s.l$ and $s.r$ should be added in X. If $s.l$ or $s.r$ is previously in X, then there is nothing to do in this step. If $s.l$ or $s.r$ previously not in X, then we add new leaves to T since it means there is a partition of interval (or expansion of interval if $s.l < T.root.l$ or $T.root.r < s.r$, or both).

- If $T.root.l < s.l$, we go down from the left most path of T, find the leftmost node and insert two children to it, where the left one represents interval $[s.l, x_1]$ and right one represents interval $[x_1, x_2]$, where x_1 and x_2 are in X before adding $s.l$ and $s.r$. Also, we set the $v.l$ to be equal to $s.l$ for every node v on the left most path, while traversing downwards.
- If $T.root.r < s.r$, it is similar to first case, we traversing down to the rightmost node to do the operation.
- If $s.l$ partition an existing interval (that being represented by a leavl in T), i.e., $x_a < s.l < x_{a+1}$ for some a, then we find the leave representing interval $[x_a, x_{a+1}]$ and"partition it" by adding two children to this node where left one represents interval $[x_a, s.l]$ and right one represents interval $[s.l, x_{a+1}]$.
- If s, r partition an existing interval, same to above case.

We color newly inserted node in T to be red.

The runtime of this step will be $O(\log N')$, since it actually do the search and then insert the node, taking time proportional to height $= \log N$ and we know $N \le 2N'$.

2. Fix up (red-black property) using RB-Insert-Fixup from CLRS 13.

If newly inserted node has black parent, then we are done for this step since we did not violate RB tree property. If newly inserted node has red parent, we have RB-Insert-Fixup(T, z) from CLRS 13.3 to help us recover the RB tree property. We can use it even we inserting two nodes at the same time (by input any of two to be the z parameter) since we always starts violating RB property as in case 1 specified in CLRS, that is, the parent and uncle are both red and grandparent is black, and the RB-Insert-Fixup fix it through recoloring parent and grand parent which does not matter if we call it after inserting one red node or two. And then the algorithm moves problem up to its grandparent, finishing fix-up via rotations.

One alternation to original to RB-Insert-Fixup(T, z) is its helper rotation algorithm. Our rotation should also update the interval that standard rotation algorithm ruin. But this can be done easily as interval changes happens only if descendants change so only two rotation pivots (original pivot ad pivot after rotation) need to be updated on their interval, by setting $v.l$ and $v.r$ to be $v.left.l$ and $v.right.r$ respectively.

The runtime of this step will be $O(\log N')$, since RB-INSERT-FIXUP(T, z) takes $O(\log |T|)$ and $|T| \leq 2N - 1 < 2N \leq 4N'$, and the updated version of rotation should still takes constant time so does not affect asymptotic runtime for RB-INSERT-FIXUP(T, z).

3. Fix up T's property (some nodes' S, S', H, H' attributes) that is disordered during rotation.

When rotation happens, except the interval of two pivot nodes need to be updated, we also update S S', H and H' attributes of the rotation pivot's descendants and ancestor, if needed. Update S when the node now contains interval s and its parent does not, node with interval expanded (pivot) to interval that original pivot represents, so it needs to inherits the attribute's from previous pivot point. The node with interval cut (pivot node before rotation), now contains less S, so going over its S attribute and delete them. Hence the current pivot node and all its proper ancestors need to update the their H. This in all, as we traverse at most $h = \log N'$ nodes upward, and do deletion on $v.H$ using linked list deletion takes at most $O(N')$ times.

The runtime of this step will be $O(N' \log N')$

4. adding s into proper $v.S$ and $v.H$

a) Add s into proper $v.S$. We recurse down to the bottom of the tree, i.e., the leaves. Then, we check if s contains the interval the node represents and does not contain its parent. If true, then adding it to .. and stop, there is no need to check if s should be in their parents/ancestors' $v.S$. If false, then we moves up to its parent, do the same operation. This kind of recursing down first and then do operation each node upwards is implemented by calling the recursive call first before checking checking containment and insertion. After putting s into correct $v.S$, it will be easier for us to adding s into proper $v.H$.

b) Add s into proper $v.H$. We recurse down to the bottom of the tree, the leaves, then after that we moves up one layer at the time checking if its child has s in its $v.S$, deciding whether to insert s into $v.H$. Since we works from bottom up, we don't need to worry about missing to add s in any $v.H$ if it should be (as $v.H$ depends on descendants only).

In a), we at most traverse all the nodes, and at each node the operation is constant time checking of interval containment, and $O(\log N')$ time for inserting node into S, which together, takes $O(N' \log N')$.

In b), we traverse all the nodes in the tree and for every node we do $O(\log N')$ operation (since searching/checking if a node in $v.S$ takes time proportional to height, and insertion to linked list takes constant time).

The runtime of this step will be $O(N' \log N')$, since it actually do the search and then insert the node, taking time proportional to height $= \log N$ and we know $N \leq 2N'$.

The runtime for the entire algorithm thus is, summing up runtime for steps 1 - 4 (as the are differen subroutine executing in order), $O(\log N') + O(\log N') + O(N' \log N') + O(N' \log N') = O(N' \log N')$, as wanted.

The algorithm do what is done in step 1 - 4, then the correctness follows. Step 1 makes sure that leaves of T should change as we change set X. Step 2 makes sure that the RB property is not violated after insertion in step 1. Step 3 recover any disorder for the T's property resulting from rotation. Step 4 adding new node s into its proper position in T. That is all we need to do, the resulting data sturture will represents the sets $S \cup s$ (done by step 1 and 4), and satisfies all the specified properties (done by 2 and 3).

Q5

Again, we divide entire algorithm into few steps. And, I do not choose the augment the data structure for deletion as I will show the algorithm given can also reach the runtime required. So, the runtime for Q3 and Q4 at all.

1. Delete proper v in T

We know that for deleting an s, its left and right endpoints should possibly be deleted from X. If there is another line segment in S or S' that has same end point as s (for one side or both), then that point, do not need to be deleted from S, otherwise we should delete it from X and also need to change T. Finding out wether there is a overlapping end point with s's end point can be done by traversing through $T.root$'s S, S', H, H' attributes, which at most takes $O(N' \log N')$ time, since it is the maximum of the total number of nodes in the data structure and in each node we do constant operation.

Then, if the endpoint of s is deleted from X, we delete the proper leaves, and updating deleted node's parent's

interval. Fining the proper leave can be done through recursively search on interval, and we still do the recursion downwards first and do operation of updating upwards. Notice that at maximum, to deleted one end point takes time proportion to height, as we traverse nodes proportional to height constant number of times, and operation on them is also constant time, updating the interval. Figure 3 shows an example of when leave deletion is needed in step 1. We want to delete x_5 from X. The four nodes with red cross is nodes to be deleted, and upward arrow indicate moving interval information upwards, and the right highlighting of path is the traversal path of the deletion.

We do not update S, S' H, H' in this step since the update is not necessary in the traversing path, we do it separately.

If deletion happens we passed a value that is the color of deleted node, for fix-up in step 2.

2. Fix up (red-black property) using RB-DELETE-FIXUP **from CLRS 13**

We possibily have deleted node from step 1. If that is the case, we call RB-DELETE-FIXUP to fix up the RB property. The runtime will be $O(logN'$

3. Fix up T's property (some nodes' S, S', H, H' attributes) that is disordered during rotation

We use the same algorithm from Q4 step 3. The runtime would be $O(N' \log N')$

4. Adding s into proper $v.S$ and $v.H$.

From previous step, we pass a pointer of the node to s to this step.

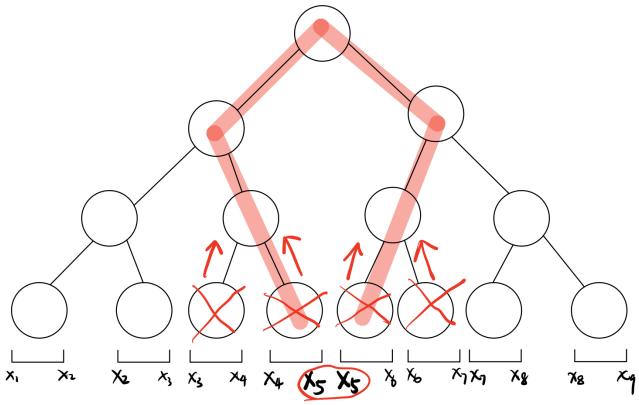
a) Then, we delete s from all $v.S$. Since we can do a recursive call downwards, deleting s from the node. Here we need to use RB-DELETE with an alternation, that is returning whether a node is deleted. If the deletion returns true, then we stop since we know that s would not in its descendants' S attribute, otherwise we check if s's projection onto x-axis overlap with the current node's interval, if true then we just recurse downward since it means that some children of it has S attribute contains s, otherwise we just stop. This in total takes at most $O(N' \log N')$. The deletion and fix up together takes $O(\log N')$, and we at most traverse all the nodes once to do this operation.

b) To delete s from $v.H$, we also use recursion. Starting from the root downwards, for a input node v we first checks whether s fully contains the interval $[v.l, v.r]$ or not overlap with $[v.l, v.r]$ at all. If true we just stop since in first case, either $s \in v.S$ or $s \in a.S$ for some proper ancestor a of v; and in second case, s would not be any of its proper descendant's S attribute thus not in its H attribute. Otherwise, we know $[s.l, s.r]$ overlaps with $[v.l, v.r]$ but not fully contain, that means it must be in $v.H$ since it must be in $u.S$ for some proper descendant u of v. Then we delete s in $v.H$, using the deletion for linked list.

The deletion algorithm takes at most $O(N')$ times. By observation, the nodes that part b) traverse would be proportional to number of nodes $v \in T$ such that $s \in v.H$. The reason is firstly we must go one level deeper after deletion since we need to check if nodes further down contain s in their H. However, if we encounter node does not contain s in H, we stop, so in this case only one depth deeper. Also, notice that we will follow a path from the root downwards, since for any v we have the relation that if $v.H$ contains s so does $v.p.H$. Hence, we know that if there is a node with $s \in v.H$ and $v \neq T.root$, $s \in T.root.H$. With "following a path from the root" and "going only one level deeper if child don't have it" and bound given in Q2 c), this together implies that the number of nodes we traverse in part b) is proportional to number of nodes $v \in T$ such that $s \in v.H$, $O(logN')$.

Together, the running time for entire algorithm is $O(N' \log N')$, by just summing up runtime from step 1 to 4, as they are executed in order, separately.

The correctness of the algorithm, again follows if step 1-4 is done. Since we preseve the RB property as well as the T's additional property (S, S', H, H'). Also, all noes of s is deleted from step 4.



(a)

Figure 3: Example of deletion of in Q4 part 1