

Week 6. Lecture Notes

Topics: Randomly built BST.
Red Black Tree.
Augmentation of data structure.
Interval trees.

Randomized BST Sort

Rand. BST-Sort ($A[1, \dots, n]$)

1. Random permutation on A
2. BST-Sort (A)

The expected time to build the tree is asymptotically the same as the running time of Randomized Quicksort, which is $O(n \log n)$.

Node depth

The depth of a node = the number of comparisions made during TREE-INSERT

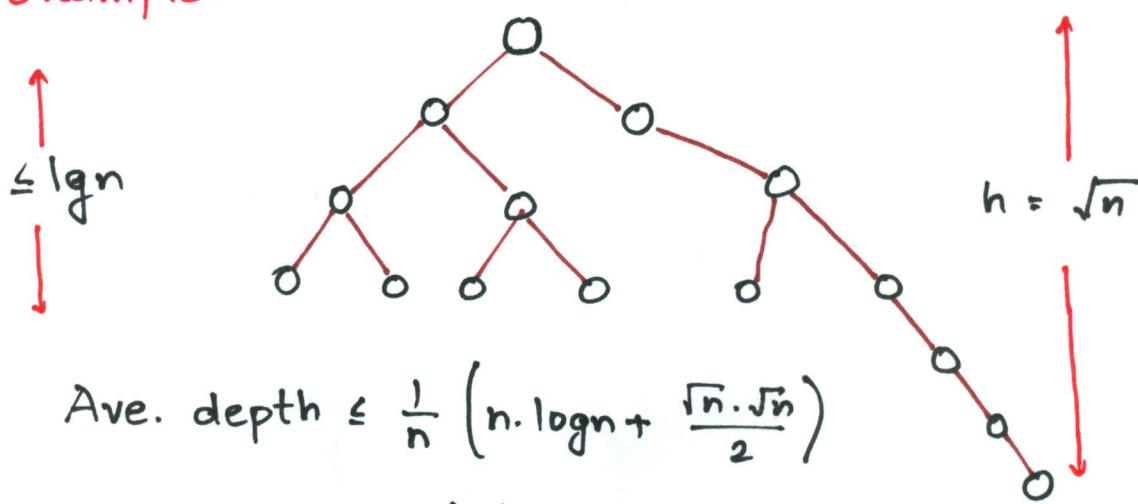
Assuming all input permutations are equally-likely, we have

$$\begin{aligned} \text{average node depth} &= \frac{1}{n} E \left[\sum_{i=1}^n [\# \text{comparisons to insert node } i] \right] \\ &= \frac{1}{n} O(n \lg n) \quad (\text{quick sort analysis}) \\ &= O(\lg n) \end{aligned}$$

Expected Tree height.

Average node depth of a randomly built BST = $O(\lg n)$, does not necessarily mean that its expected height is also $O(\lg n)$ (although it is)

Example:



$$\begin{aligned} \text{Ave. depth} &\leq \frac{1}{n} \left(n \cdot \lg n + \frac{\sqrt{n} \cdot \sqrt{n}}{2} \right) \\ &= O(n) \end{aligned}$$

Height of a randomly built binary search tree

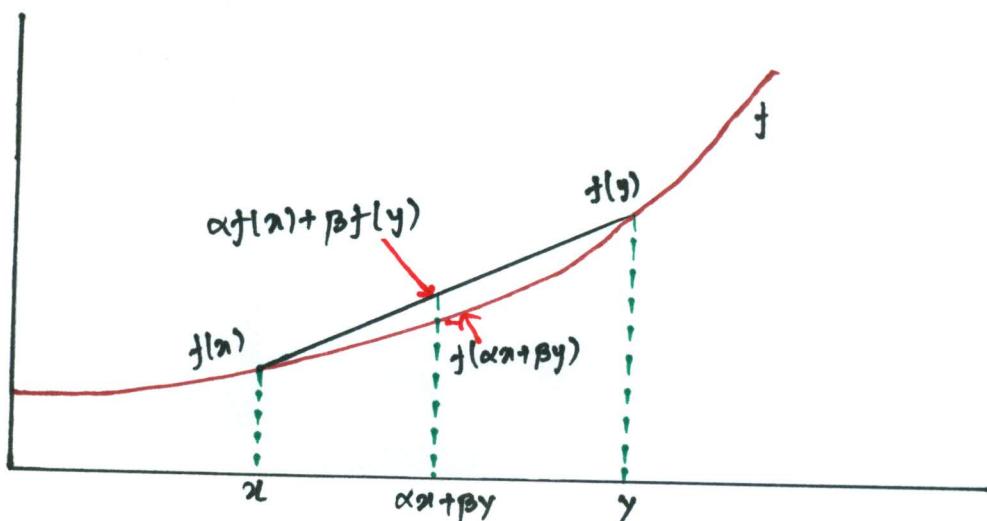
Outline of the analysis:

- Prove Jensen's inequality, which says that $f(E[x]) \leq E[f(x)]$, for any convex function f and random variable X .
- Analyze the exponential height of a randomly built BST on n nodes, which is the random variable $Y_n = 2^{X_n}$, where X_n is the random variable denoting the height of the BST.
- Prove that $2^{E[X_n]} \leq E[2^{X_n}] = E[Y_n] = O(n^3)$ and hence that $E[X_n] = O(\log n)$.

Convex Functions

A function $f: R \rightarrow R$ is convex if for all $\alpha, \beta \geq 0$ such that $\alpha + \beta = 1$, we have

$$f(\alpha x + \beta y) \leq \alpha f(x) + \beta f(y) \text{ for all } x, y \in R$$



Convexity Lemma

Lemma: Let $f: \mathbb{R} \rightarrow \mathbb{R}$ be a convex function, and let $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$ be a set of non-negative constants such that $\sum_k \alpha_k = 1$. Then, for any set $\{x_1, x_2, \dots, x_n\}$ of real numbers, we have

$$f\left(\sum_{k=1}^n \alpha_k x_k\right) \leq \sum_{k=1}^n \alpha_k f(x_k)$$

Proof: We prove by induction on n .

for $n=1$, we have $\alpha_1 = 1$, and hence $f(\alpha_1 x_1) \leq \alpha_1 f(x_1)$

Inductive step:

$$\begin{aligned} f\left(\sum_{k=1}^n \alpha_k x_k\right) &= f\left(\alpha_n x_n + (1-\alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1-\alpha_n) f\left(\sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} x_k\right) \\ &\leq \alpha_n f(x_n) + (1-\alpha_n) \sum_{k=1}^{n-1} \frac{\alpha_k}{1-\alpha_n} f(x_k) \\ &= \sum_{k=1}^n \alpha_k f(x_k). \end{aligned}$$

$$\therefore f\left(\sum_{k=1}^n \alpha_k x_k\right) \leq \sum_{k=1}^n \alpha_k f(x_k)$$

Jensen's Inequality

Lemma: Let f be a convex function, and let X be a random variable. Then

$$f(E[X]) \leq E[f(X)]$$

Proof:

$$\begin{aligned} f(E[X]) &= f\left(\sum_{k=-\infty}^{\infty} k \cdot \Pr\{X=k\}\right) \\ &\leq \sum_{k=-\infty}^{\infty} f(k) \cdot \Pr\{X=k\} \\ &= E[f(X)] \end{aligned}$$

\hookrightarrow Definition of expectation

\hookrightarrow Convexity lemma
(generalized)

Analysis of BST height

Let X_n be the random variable denoting the height of a randomly built binary search tree on n nodes, and let $Y_n = 2^{X_n}$ be its exponential height.

If the root of the tree has rank k , then

$$X_n = 1 + \max\{X_{k-1}, X_{n-k}\}$$

since each of the left and right subtrees of the root are randomly built.

Hence, we have

$$Y_n = 2 \cdot \max\{Y_{k-1}, Y_{n-k}\}$$

Define the indicator random variable Z_{nk} as

$$Z_{nk} = \begin{cases} 1, & \text{if root has rank } k; \\ 0, & \text{otherwise.} \end{cases}$$

Thus

$$\Pr\{Z_{nk} = 1\} = E[Z_{nk}] = 1/n$$

and

$$Y_n = \sum_{k=1}^n Z_{nk} (2 \cdot \max\{Y_{k-1}, Y_{n-k}\}).$$

Exponential Height Recurrence

We have.

$$T_n = \sum_{k=1}^n Z_{nk} (2 \cdot \max\{T_{k-1}, T_{n-k}\})$$

$$\Rightarrow E[T_n] = E\left[\sum_{k=1}^n Z_{nk} (2 \cdot \max\{T_{k-1}, T_{n-k}\})\right]$$

↳ Taking expectations of both sides

$$= \sum_{k=1}^n E[Z_{nk} (2 \cdot \max\{T_{k-1}, T_{n-k}\})]$$

↳ Linearity of expectation

$$= 2 \sum_{k=1}^n E[Z_{nk}] \cdot E[\max\{T_{k-1}, T_{n-k}\}]$$

↳ Independence of the rank
of root from ranks of
subtree roots.

$$\leq \frac{2}{n} \sum_{k=1}^n E[T_{k-1} + T_{n-k}]$$

↳ The max of two non-negative
numbers is atmost their sum.

$$\text{and } E[Z_{nk}] = 1/n$$

$$= \frac{4}{n} \sum_{k=0}^{n-1} E[T_k]$$

↳ Each term appears twice
and re-index.

Solving the recurrence

Use substitution to show that $E[Y_n] \leq cn^3$ for some positive constant c , which we can pick sufficiently large to handle the initial conditions.

We have,

$$\begin{aligned} E[Y_n] &= \frac{4}{n} \sum_{k=0}^{n-1} E[Y_k] \\ &\leq \frac{4}{n} \sum_{k=0}^{n-1} ck^3 \quad \rightarrow \text{Substitution} \\ &\leq \frac{4c}{n} \int_0^n x^3 dx \quad \rightarrow \text{Integral method} \\ &= \frac{4c}{n} \left(\frac{n^4}{4} \right) \quad \rightarrow \text{Solving the integral} \\ &= cn^3 \end{aligned}$$

Putting it all together, we have

$$2^{E[X_n]} \leq E[2^{X_n}]$$

Jensen's inequality, since $f(x) = 2^x$ is convex

So,

$$\begin{aligned} 2^{E[X_n]} &\leq E[2^{X_n}] \\ &= E[Y_n] \quad \rightarrow \text{Definition} \\ &\leq cn^3 \quad \rightarrow \text{just showed above} \end{aligned}$$

$$E[X_n] \in 3 \log n + O(1) \quad \rightarrow \text{Taking log both sides}$$

Hence,

$$E[X_n] \in 3 \log n + O(1)$$

Post Mortem

Q. Does the analysis have to be this hard?

Q. Why bother with analyzing exponential height?

Q. Why not just develop the recurrence on

$$x_n = 1 + \max \{ x_{k-1}, x_{n-k} \}$$

directly.

Answer:

The inequality $\max\{a,b\} \leq a+b$, provides a poor upper bound, since the R.H.S. approaches the L.H.S. slowly as $|a-b|$ increases.

The bound

$$\max \{ 2^a, 2^b \} \leq 2^a + 2^b$$

allows the R.H.S. to approach the L.H.S. far more quickly as $|a-b|$ increases.

By using the convexity of $f(x) = 2^x$ via Jensen's inequality, we can manipulate the sum of exponentials, resulting in a tight analysis.

Balanced Search Trees

Balanced Search tree: A search-tree data structure for which a height of $O(\log n)$ is guaranteed when implementing a dynamic set on n items.

Examples: AVL Trees, 2-3 trees
2-3-4 trees, B-trees.

Red-Black Trees

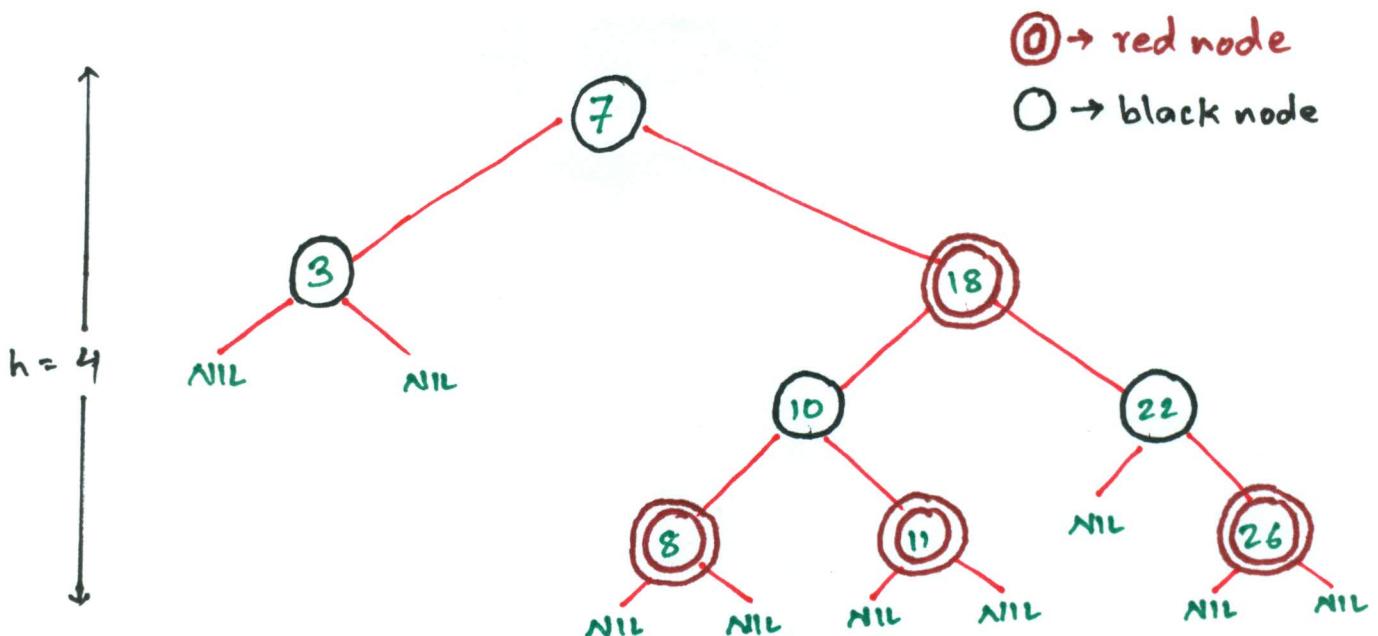
Red-Black Trees

This data structure requires an extra one-bit field - color, in each node.

Red-black properties:

1. Every node is either red or black
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black
4. All simple paths from any node α to a descendant leaf have the same number of black nodes = black-height (α).

Example of a red-black tree



1. Every node is either red or black
2. The root and leaves (NIL's) are black
3. If a node is red, then its parent is black
4. All simple paths from any node x to a descendant leaf have the same number of black nodes = black height (x)

Height of a red black tree

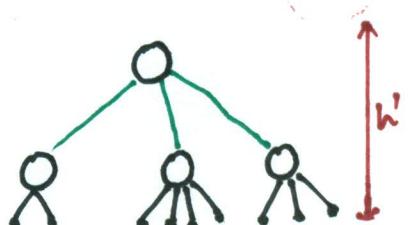
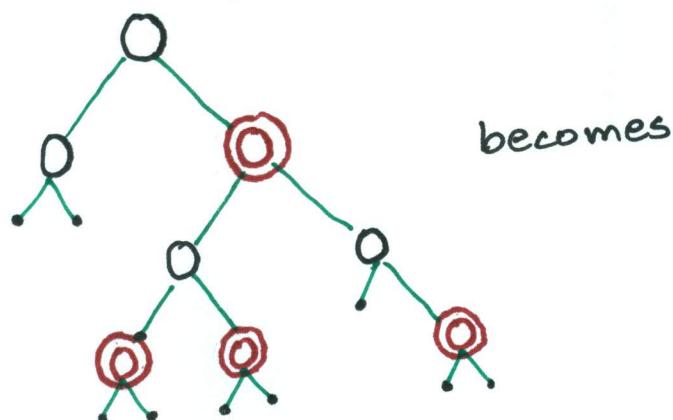
Theorem:

A red black tree with n keys has height $h \leq 2 \log(n+1)$.

Proof:

- Intuition: Merge red nodes into their black parents.

So,



- This process produces a tree in which each node has 2, 3, or 4 children.
- The 2-3-4 tree has uniform depth h' of leaves
- We have $h' \geq h/2$, since atmost half the leaves on any path are red
- The number of leaves in each tree is $n+1$
 - $\Rightarrow n+1 \geq 2^{h'}$
 - $\Rightarrow \log(n+1) \geq h' \geq h/2$
 - $\Rightarrow h \leq 2 \log(n+1)$

Query Operations

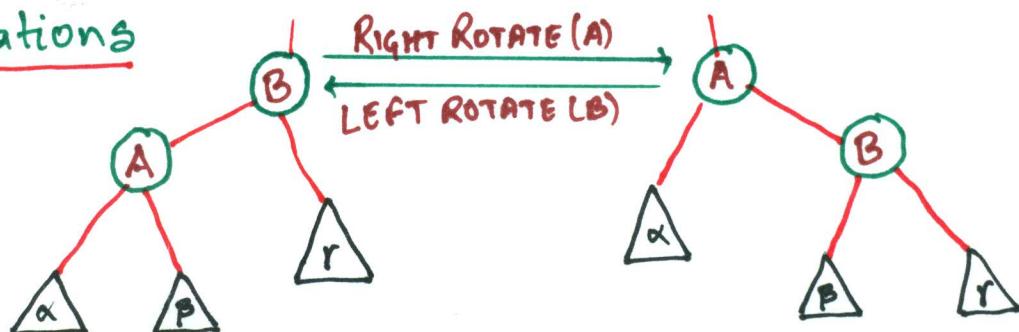
Corollary: The queries SEARCH, MIN, MAX, SUCCESSOR and PREDECESSOR all run in $O(\log n)$ time on a red-black tree with n -nodes.

Modifying Operations

The operations INSERT and DELETE cause modifications to the red black tree:

- the operation itself
- color changes
- restructuring the links of the tree:
"rotation"

Rotations



Rotations maintain the inorder ordering of keys:

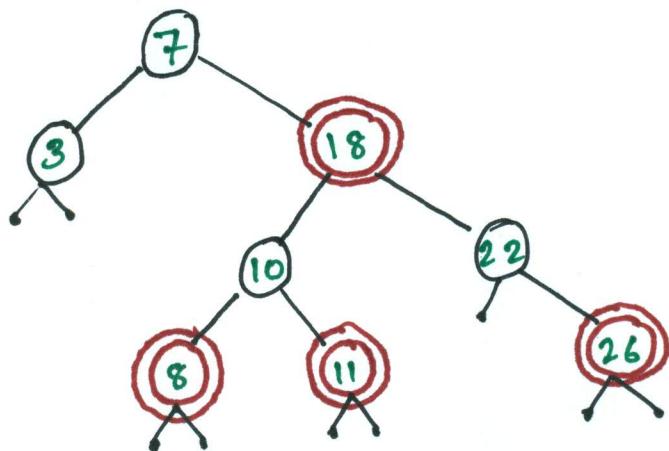
$$a \in \alpha, b \in \beta, c \in r \Rightarrow a \leq A \leq b \leq B \leq c$$

A rotation can be performed in $O(1)$ time.

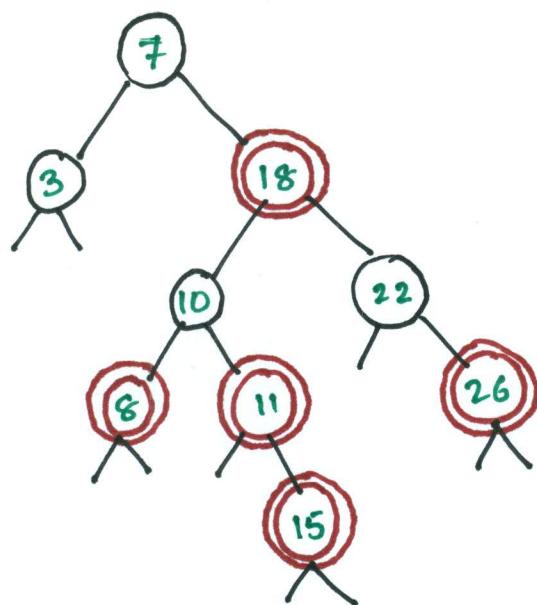
Insertion into a red-black tree

IDEA: Insert α in tree. Color α red. Only red-black property 3 might be violated. Move the violation up the tree by recoloring until it can be fixed with rotations and recoloring

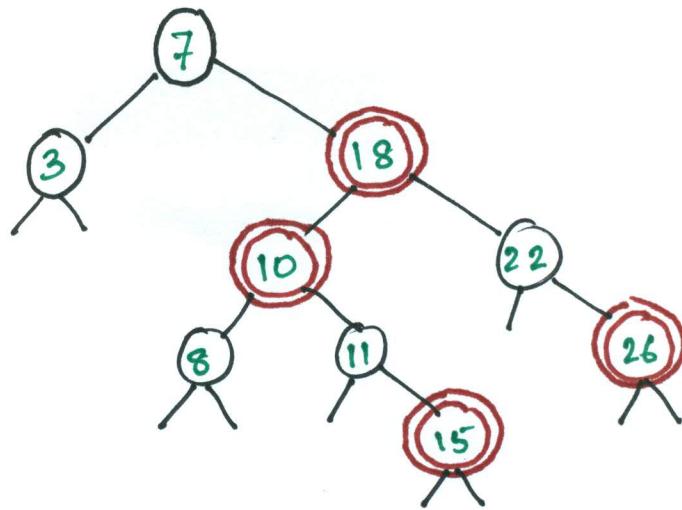
Example:



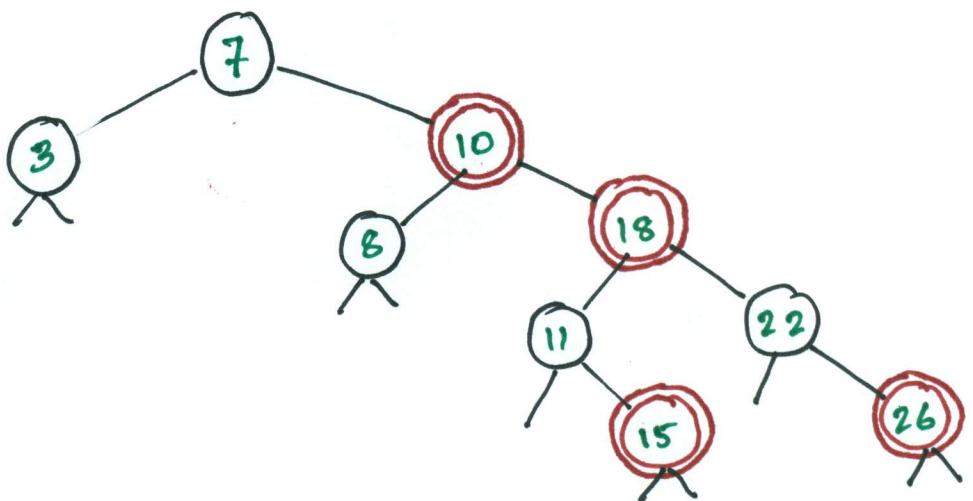
1. Insert $\alpha = 15$.



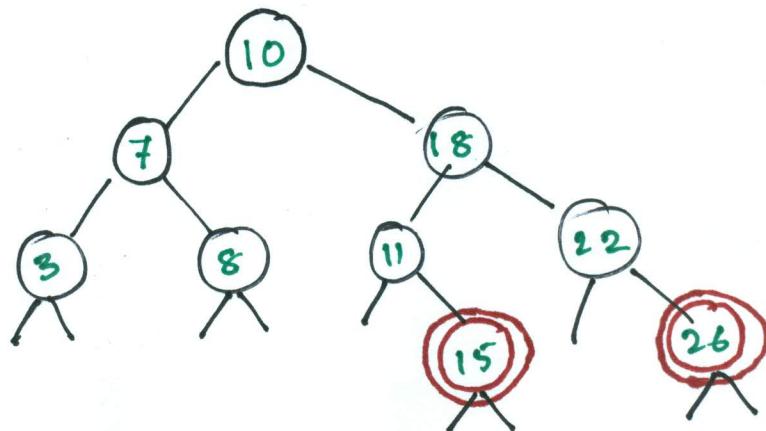
2. Recolor, moving the violation up the tree



3. RIGHT-ROTATE(10)



4. LEFT-ROTATE(7) and recolor



Pseudo code

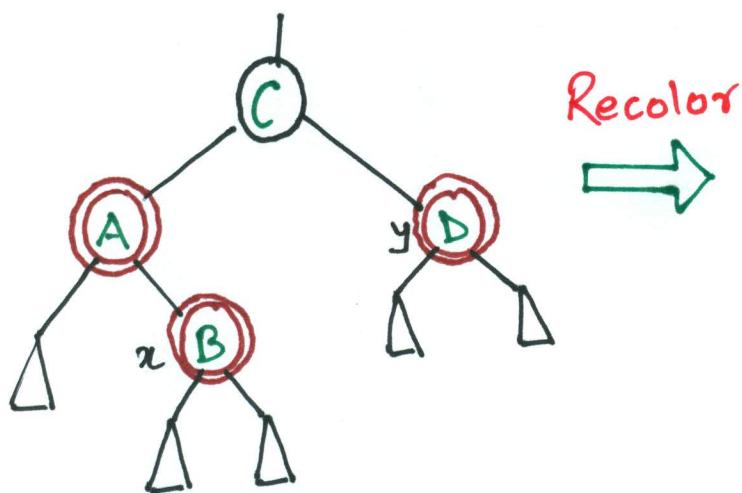
1. RB - INSERT (T, x)
2. TREE - INSERT (T, x)
3. $\text{color}[x] \leftarrow \text{RED}$ ► only RB property 3 can be violated
4. while $x \neq \text{root}[T]$ and $\text{color}[\text{p}[x]] = \text{RED}$
5. do if $\text{p}[x] = \text{left}[\text{p}[\text{p}[x]]]$
6. then $y \leftarrow \text{right}[\text{p}[\text{p}[x]]]$ • y = aunt / uncle of x
7. if $\text{color}[y] = \text{RED}$
8. then <Case 1>
9. else if $x = \text{right}[\text{p}[x]]$
10. then <Case 2> ► Case 2 falls into case 3
11. <case 3>
12. else <"then" clause with "left" and "right" swapped>
13. $\text{color}[\text{root}[T]] \leftarrow \text{BLACK}$

Graphical Notation

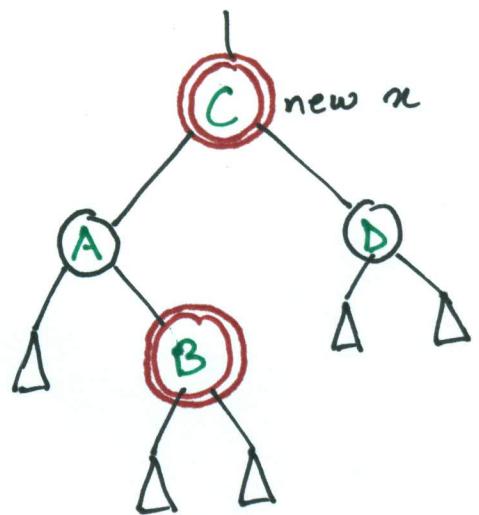
Let  denote a subtree with a black root

All 's have the same black height

Case 1

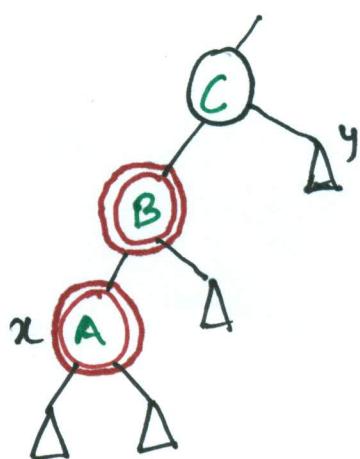
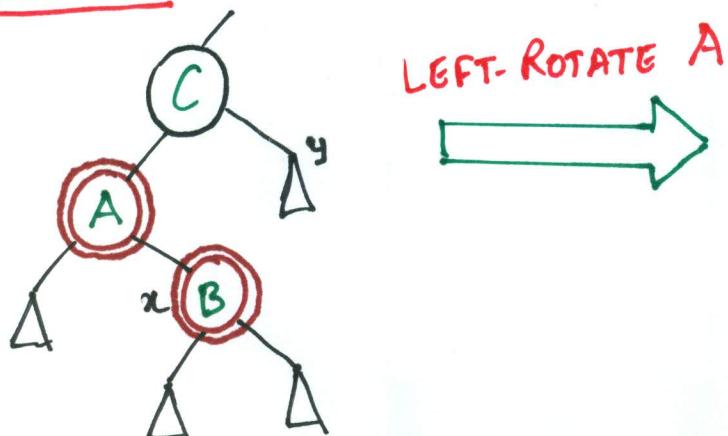


(or, children of A are swapped)



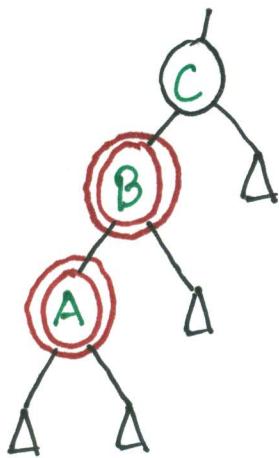
Push C's black onto A and D, and recurse since C's parent may be red.

Case 2

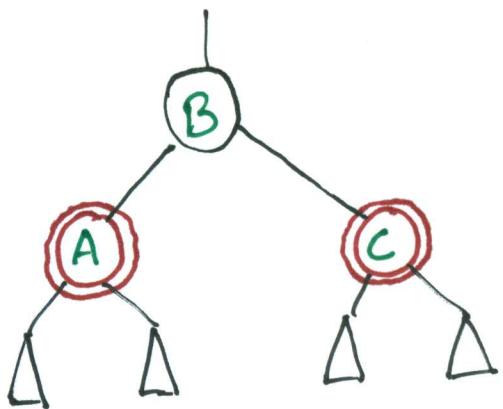


Transform to Case 3

Case 3



RIGHT-ROTATE (c)



Done! No more violations of RB property 3 are possible.

Analysis

- Go up the tree performing Case 1, which only recolor nodes
- If case 2 or Case 3 occurs, perform 1 or 2 rotations, and terminate.

Running Time:

$O(\log n)$ with $O(1)$ rotations

Note:

RB-DELETE - takes same asymptotic running time.

Dynamic Order Statistics

OS-SELECT (i, S) : returns the i^{th} smallest element in the dynamic set S

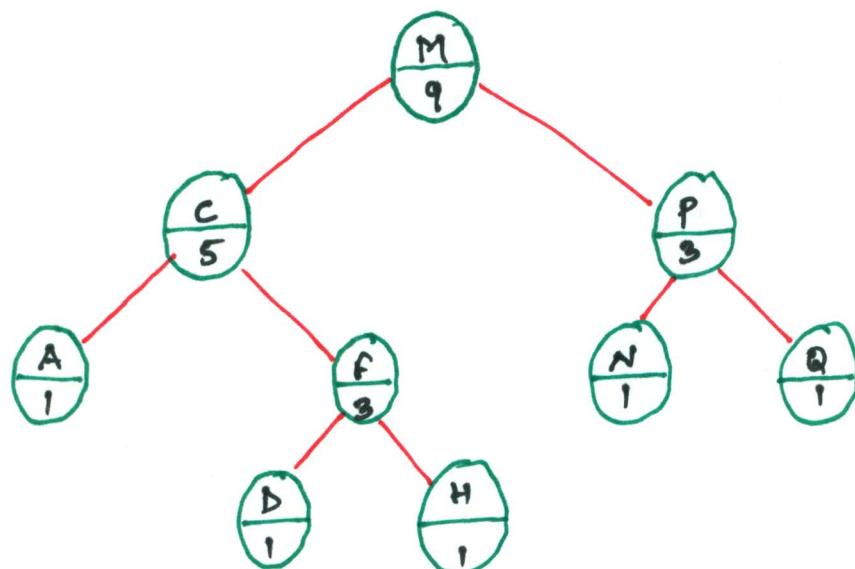
OS-RANK (x, S) : returns the rank of $x \in S$ in the sorted order of S 's elements.

IDEA: Use a red-black tree for the set S , but keep subtree sizes in the node.

Notation for nodes:



Example of an OS tree



$$\text{size}[x] = \text{size}[\text{left}[x]] + \text{size}[\text{right}[x]] + 1$$

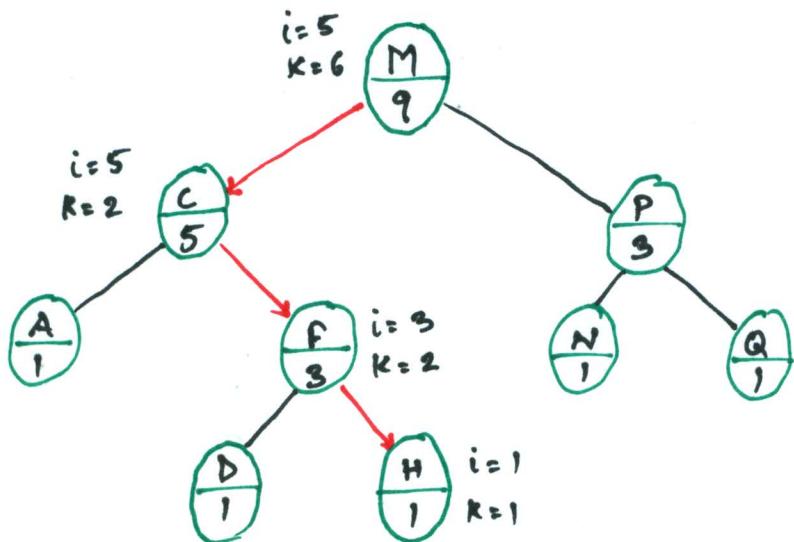
SELECTION

Implementation Trick: Use a sentinel (dummy record) for NIL such that $\text{size}[\text{NIL}] = 0$.

1. OS-SELECT(α, i)
2. $K \leftarrow \text{size}[\text{left}[\alpha]] + 1$ $\rightarrow K = \text{rank}(\alpha)$
3. if $i = K$ then return α
4. if $i < K$
then return OS-SELECT(left[α], i)
5. else return OS-SELECT(right[α], $i - K$)

Example:

OS-SELECT(root, 5)



Running Time: $O(n) = O(\log n)$ for red-black trees.

Data Structure Maintenance

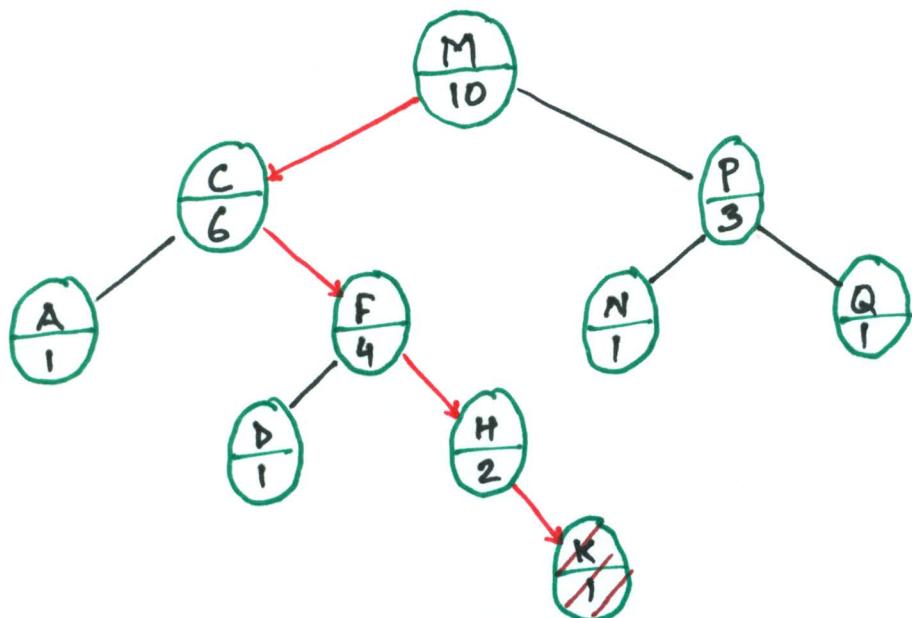
- Q. Why not keep the ranks themselves in the nodes instead of subtree sizes?
- A. They are hard to maintain when the red-black tree is modified.

Modifying operations: INSERT and DELETE

Strategy: Update subtree sizes when inserting or deleting.

Example of insertion

INSERT ("K")



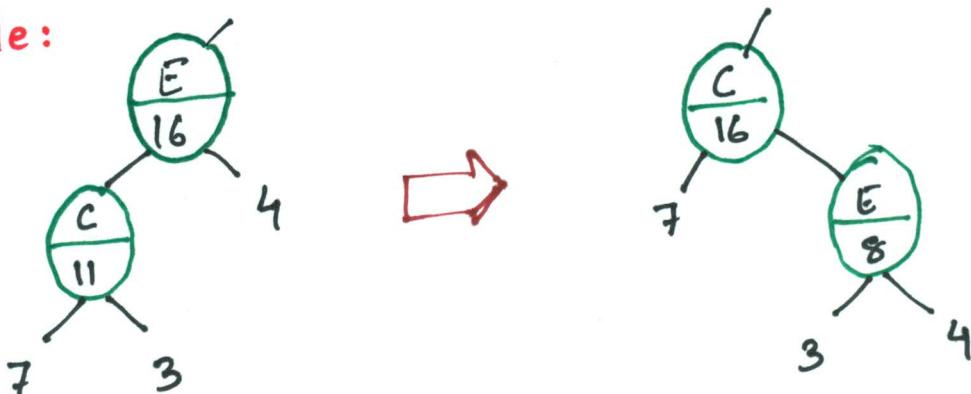
Handling rebalancing

Don't forget that RB-INSERT and RB-DELETE may also need to modify the red-black tree in order to maintain balance.

Recolorings: no effect on subtree sizes.

Rotations: fix up subtree sizes in $O(1)$ time

Example:



∴ RB-INSERT and R.B.DELETE run in $O(\log n)$ time.

Data-structure augmentation

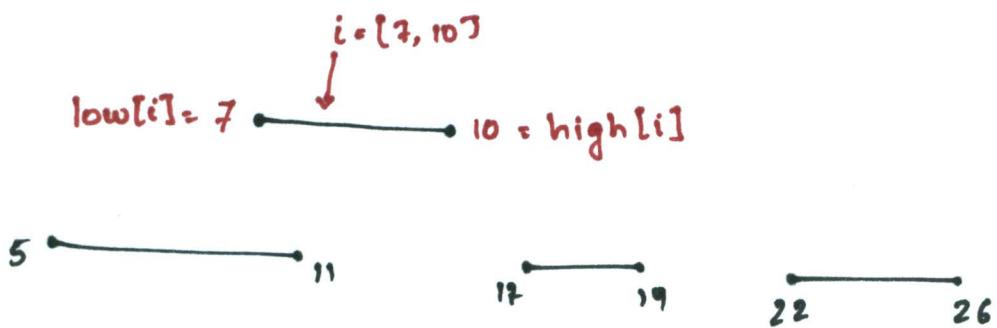
Methodology: (e.g., order-statistics trees)

1. Choose an underlying data structure (red-black trees)
2. Determine additional information to be stored in the data structure (subtree sizes)
3. Verify that this information can be maintained for modifying operations (RB-INSERT, RB-DELETE - don't forget rotations)
4. Develop new dynamic-set operations that use the information (OS-SELECT and OS-RANK)

These steps are guidelines, not rigid rules.

Interval Trees

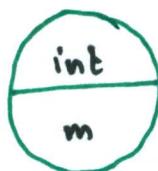
Goal: To maintain a dynamic set of intervals, such as time intervals.



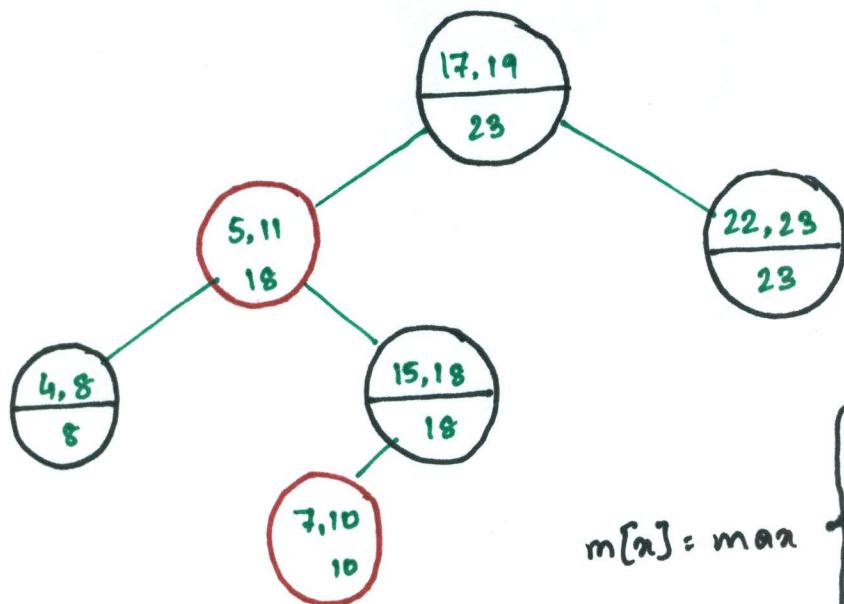
Query: For a given query interval i , find an interval in the set that overlaps i .

Following the methodology

1. Choose an underlying data structure.
 - Red black tree keyed on low(left) endpoint.
2. Determine additional information to be stored in the data structure.
 - Store in each node α the largest value $m[\alpha]$ in the subtree rooted at α , as well as the interval $int[\alpha]$ corresponding to the key.



Example interval tree

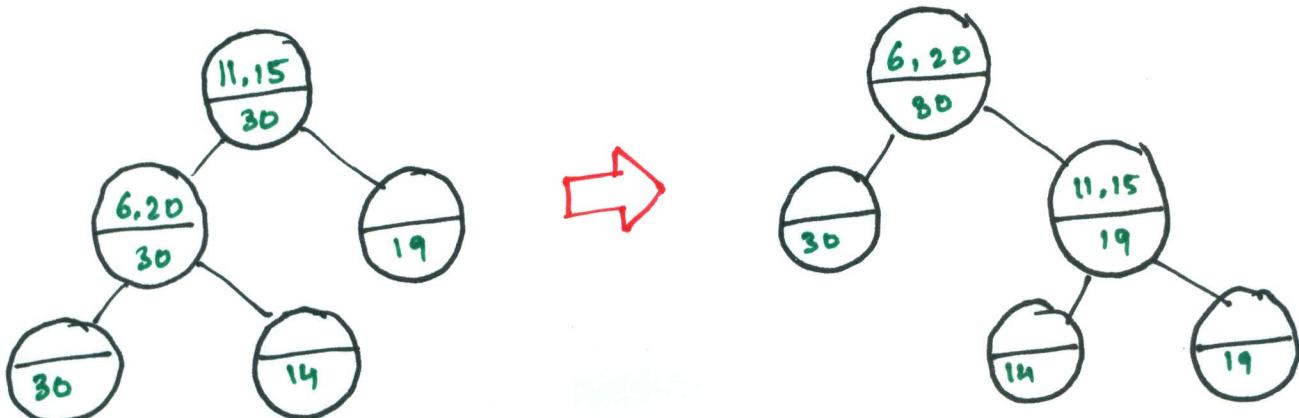


$$m[a] = \max \begin{cases} \text{high[int[a]]} \\ m[\text{left}[a]] \\ m[\text{right}[a]] \end{cases}$$

Modifying operations

3. Verify that this information can be maintained for modifying operations.

- **INSERT** : fix m's on the way down
- **ROTATION**: fixup = $O(1)$ time per rotation.



Total insert time: $O(\log n)$;
Delete similar.

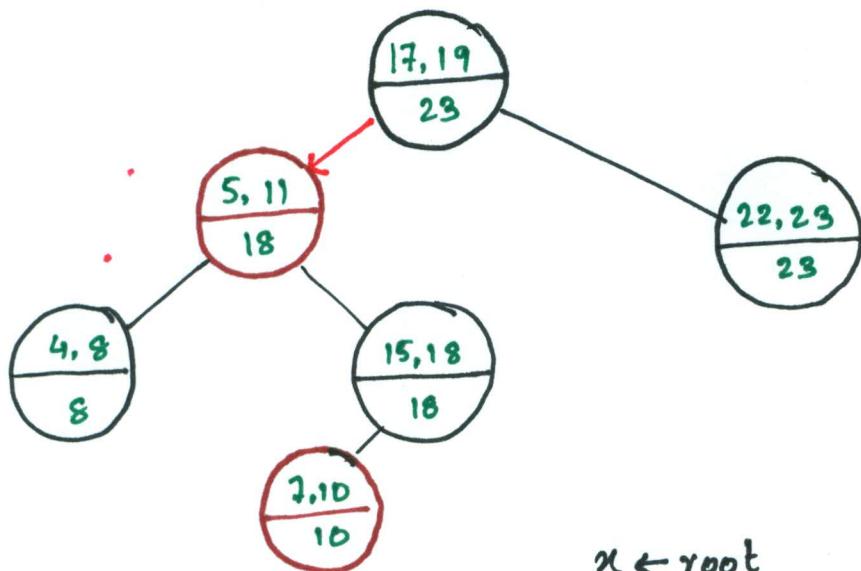
New operations

4. Develop new dynamic-set operations that use the information.

INTERVAL-SEARCH (i)

1. $\alpha \leftarrow \text{root}$
2. while $\alpha \neq \text{NIL}$ and ($\text{low}[i] > \text{high}[\text{int}[\alpha]]$
or $\text{low}[\text{int}[\alpha]] > \text{high}[i]$)
3. do $\triangleright i$ and $\text{int}[\alpha]$ don't overlap
4. if $\text{left}[\alpha] \neq \text{NIL}$ and $\text{low}[i] \leq \text{m}[\text{left}[\alpha]]$
5. then $\alpha \leftarrow \text{left}[\alpha]$
6. else $\alpha \leftarrow \text{right}[\alpha]$
7. return α

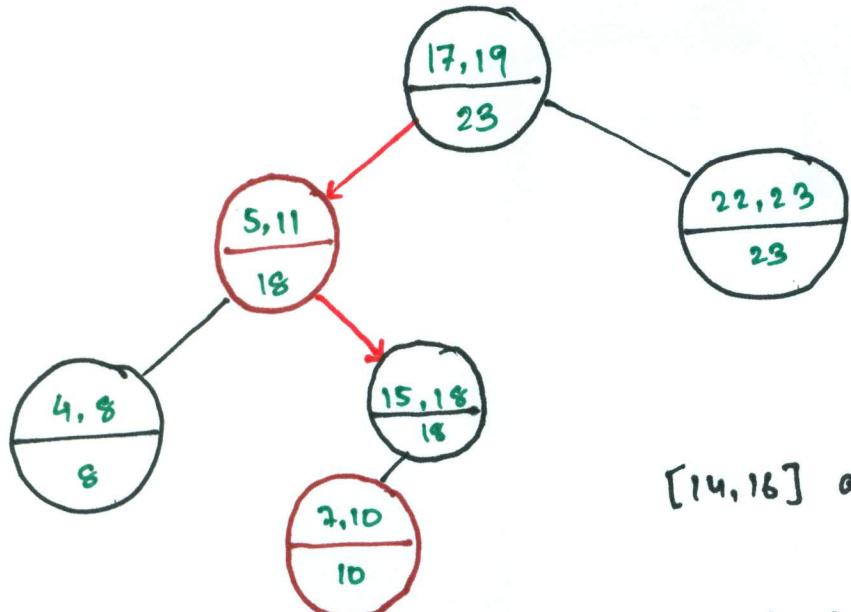
Example 1: INTERVAL-SEARCH ([14, 16])



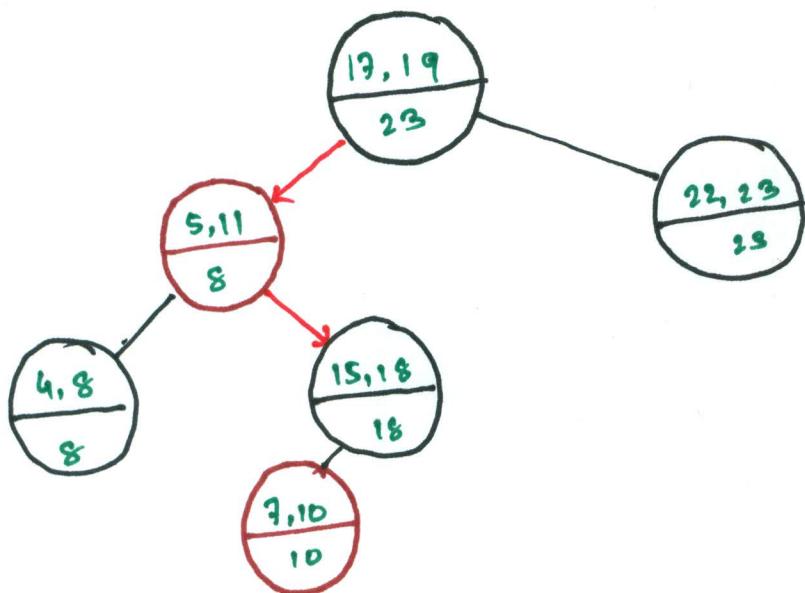
$\alpha \leftarrow \text{root}$

$[14, 16]$ and $[17, 19]$ don't
overlap

$14 < 18 \Rightarrow \alpha \leftarrow \text{left}[\alpha]$



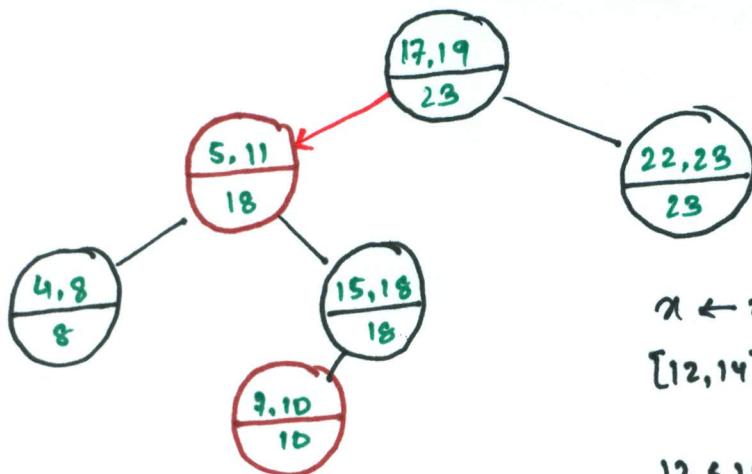
$[14,16]$ and $[5,11]$ don't overlap
 $14 > 8 \Rightarrow x \leftarrow \text{right}[x]$



$[14,16]$, and $[15,18]$ overlap
return $[15,18]$

Example 2: INTERVAL-SEARCH ($[12, 14]$)

1.

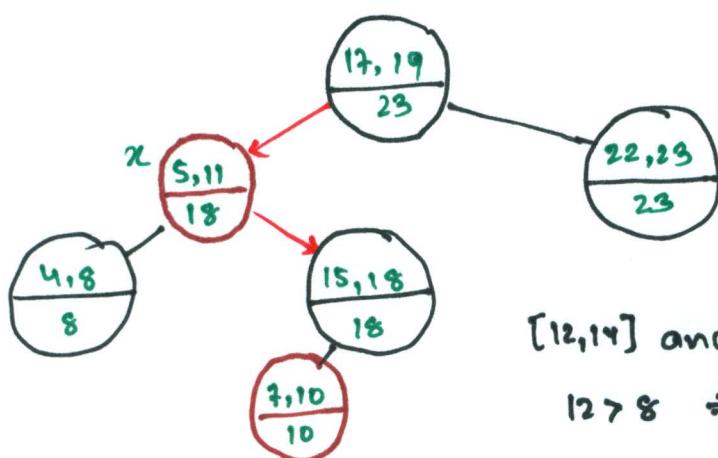


$\alpha \leftarrow \text{root}$

$[12, 14]$ and $[17, 19]$ don't overlap

$12 \leq 18 \Rightarrow \alpha \leftarrow \text{left}[\alpha]$

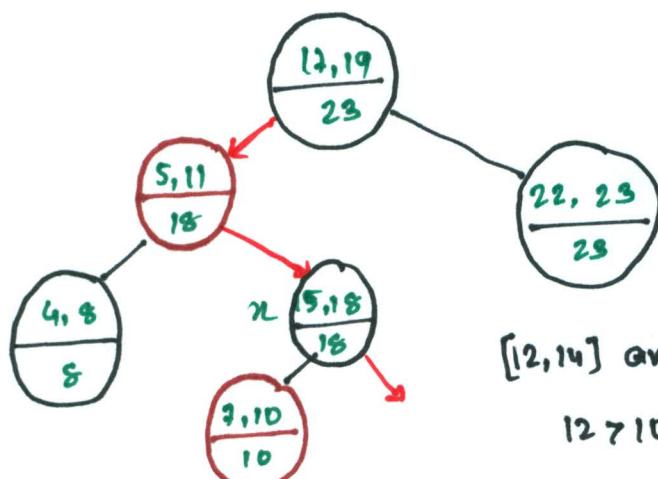
2.



$[12, 14]$ and $[5, 11]$ don't overlap

$12 > 8 \Rightarrow \alpha \leftarrow \text{right}[\alpha]$

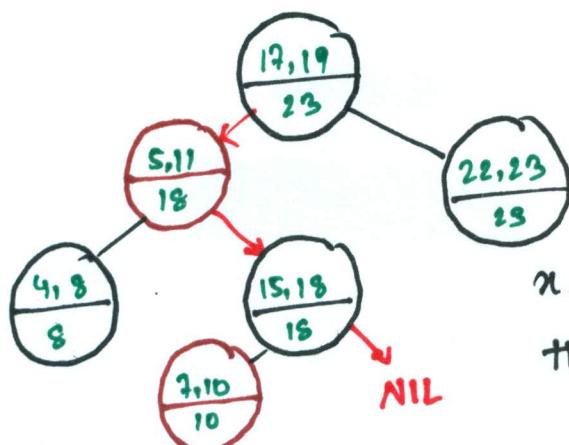
3.



$[12, 14]$ and $[15, 18]$ don't overlap

$12 > 10 \Rightarrow \alpha \rightarrow \text{right}[\alpha]$

4.



$\alpha = [\text{NIL}] \Rightarrow$ no interval
that overlaps
 $[12, 14]$ exist.

Analysis

Time = $O(n) = O(\log n)$, since INTERVAL-SEARCH does constant work at each level as it follows a simple path down the tree.

List all overlapping intervals:

- Search, list, delete, repeat
- Insert them all again at the end

Time = $O(k \log n)$, where k is the total number of overlapping intervals.

This is an output-sensitive bound.

Best algorithm to date: $O(k + \log n)$

Correctness

Theorem: Let L be the set of intervals in the left subtree of node x , and let R be the set of intervals in x 's right subtree.

If the search goes right, then

$$\{i' \in L : i' \text{ overlaps } i\} = \emptyset$$

If the search goes left, then

$$\{i' \in L : i' \text{ overlaps } i\} = \emptyset$$

$$\Rightarrow \{i' \in R : i' \text{ overlaps } i\} = \emptyset$$

In other words, it's always safe to take ~~any~~ only 1 of the two children: we will either find something or nothing was to be found.

Correctness Proof

Proof: Suppose first that search goes right.

- If $\text{left}[x] = \text{NIL}$, then we are done, since $L = \emptyset$
- Otherwise, the code dictates that we must have $\text{low}[i] > m[\text{left}[x]]$. The value $m[\text{left}[x]]$ corresponds to the right endpoint of some interval $j \in L$, and no other interval in L can have a larger right endpoint than $\text{high}(j)$



- Therefore, $\{i' \in L : i' \text{ overlaps } i\} = \emptyset$.

Next suppose that the search goes left, and assume that

$$\{i' \in L : i' \text{ overlaps } i\} = \emptyset$$

- Then, the code dictates that $\text{low}[i] \leq m[\text{left}[x]] = \text{high}[j]$ for some $j \in L$
- Since $j \in L$, it does not overlap i , and hence $\text{high}[i] < \text{low}[j]$
- But, the binary-search tree property implies that for all $i' \in R$, we have $\text{low}[j] \leq \text{low}[i']$
- But then $\{i' \in R : i' \text{ overlaps } i\} = \emptyset$.