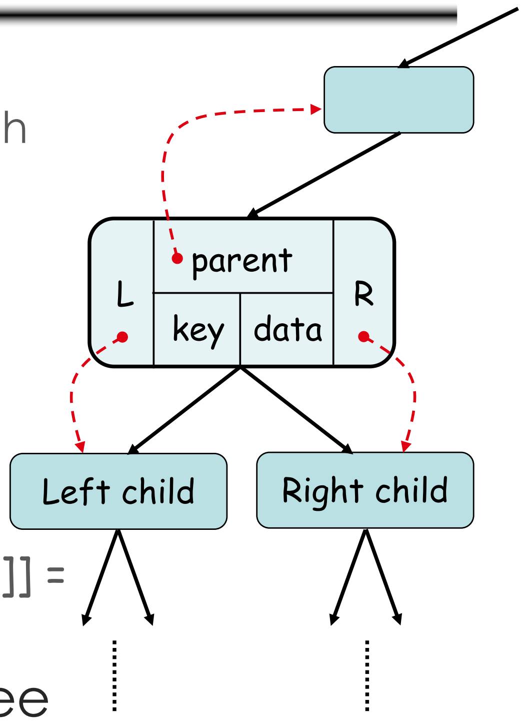
CSC 611: Analysis of Algorithms

Lecture 9

Red-Black Trees, OS Trees, and Interval Trees

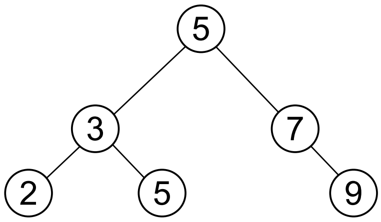
Binary Search Trees

- Tree representation:
  - A linked data structure in which each node is an object
- Node representation:
  - Key field
  - Satellite data
  - Left: pointer to left child
  - Right: pointer to right child
  - p: pointer to parent (p [root [T]] = NIL)
- Satisfies the binary search tree property



Binary Search Tree Example

- Binary search tree property:
  - If **y** is in left subtree of **x**, then **key [y] ≤ key [x]**
  - If **y** is in right subtree of **x**, then **key [y] ≥ key [x]**



Binary Search Trees

- Support many dynamic set operations
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- Running time of basic operations on binary search trees
  - On average: **Θ(lgn)**
    - The expected height of the tree is **lgn**
  - In the worst case: **Θ(n)**
    - The tree is a linear chain of **n** nodes

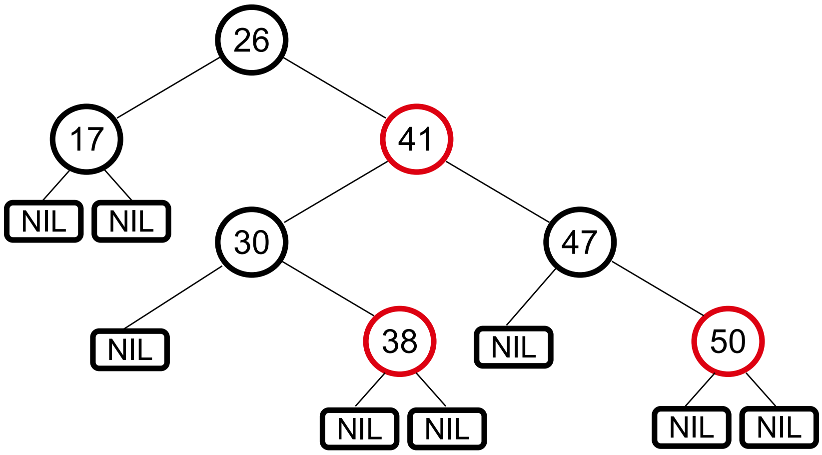
Red-Black Trees

- "Balanced" binary trees guarantee an $O(\lg n)$ running time on the basic dynamic-set operations
- Red-black tree
  - Binary tree with an additional attribute for its nodes: **color** which can be **red** or **black**
  - Constrains the way nodes can be colored on any path from the root to a leaf
    - Ensures that no path is more than twice as long as another ⇒ the tree is balanced
  - The nodes inherit all the other attributes from the binary-search trees: **key**, **left**, **right**, **p**

Red-Black Trees Properties

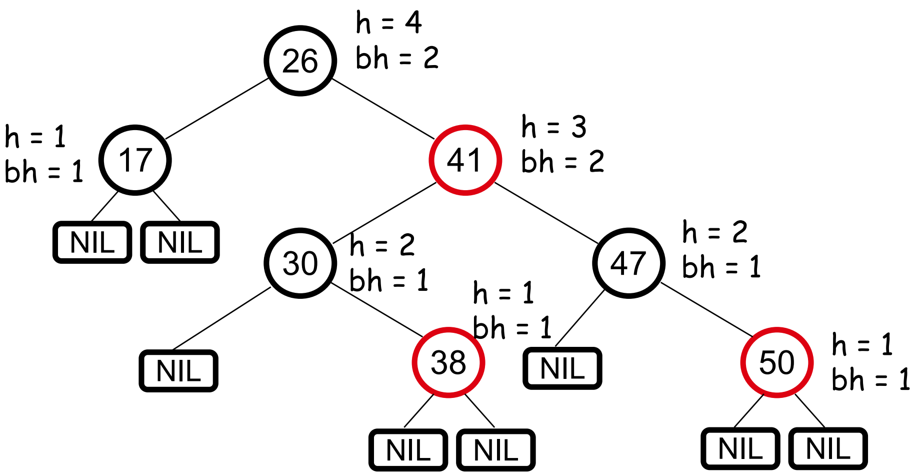
1. Every **node** is either **red** or **black**
2. The **root** is **black**
3. Every *leaf* (NIL) is **black**
4. If a node is red, then both its children are black
   - No two red nodes in a row on a simple path from the root to a leaf
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Example: RED-BLACK TREE



- For convenience we use a sentinel NIL[T] to represent all the NIL nodes at the leafs
  - NIL[T] has the same fields as an ordinary node
  - *Color*[NIL[T]] = *BLACK*
  - The other fields may be set to arbitrary values

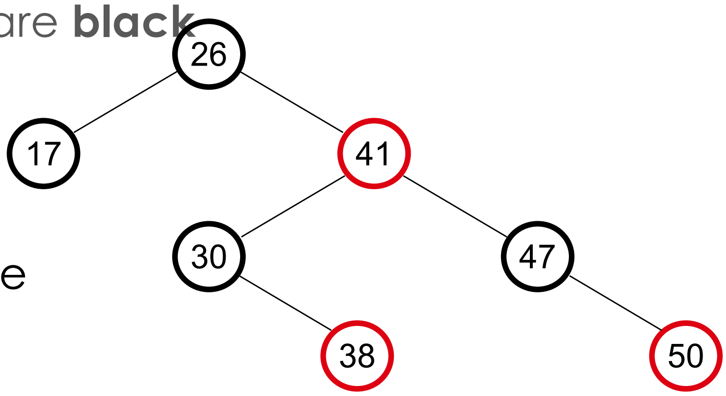
Black-Height of a Node



- **Height of a node:** the number of edges in a longest path to a leaf
- **Black-height** of a node **x**: bh(x) is the number of black nodes (including NIL) on a path from **x** to leaf, not counting **x**

Properties of Red-Black Trees

- **Claim**
  – Any node with height **h** has black-height ≥ **h/2**
- **Proof**
  – By property 4, there are at most **h/2** **red** nodes on the path from the node to a leaf
  – Hence at least **h/2** are **black**



Property 4: if a node is red then both its children are black

Properties of Red-Black Trees

Claim

The subtree rooted at any node **x** contains at least $2^{bh(x)} - 1$ internal nodes

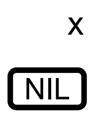
Proof: By induction on height of **x**

Basis: **height[x]** = 0 ⇒

x is a leaf (**NIL[T]**) ⇒

bh(x) = 0 ⇒

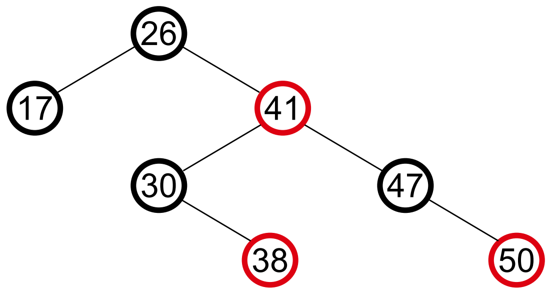
# of internal nodes: $2^0 - 1 = 0$



Properties of Red-Black Trees

Inductive step:

- Let **height**(x) = h and **bh(x)** = b
- Any child **y** of **x** has:
  - bh (y) = b (if the child is **red**), or
  - bh (y) = b - 1 (if the child is **black**)



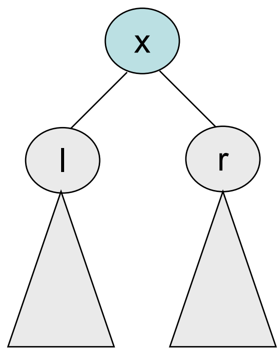
Properties of Red-Black Trees

- Want to prove:
  - The subtree rooted at any node **x** contains at least $2^{bh(x)} - 1$ internal nodes
- Assume true for children of **x**:
  - Their subtrees contain at least $2^{bh(x) - 1} - 1$ internal nodes
- The subtree rooted at **x** contains at least:

  $(2^{bh(x) - 1} - 1) + (2^{bh(x) - 1} - 1) + 1 =$

  $2 \cdot (2^{bh(x) - 1} - 1) + 1 =$

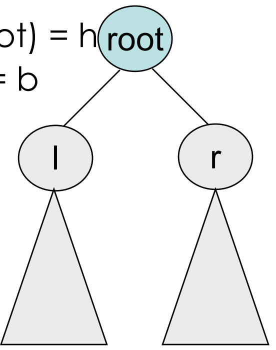
  $2^{bh(x)} - 1$ internal nodes



Properties of Red-Black Trees

Lemma: A red-black tree with **n** internal nodes has height at most **2lg(n + 1)**.

Proof:



$n \geq 2^b - 1 \geq 2^{h/2} - 1$

number n of internal nodes

since $b \geq h/2$

- Add 1 to all sides and then take logs:

$$n + 1 \geq 2^b \geq 2^{h/2}$$
$$\lg(n + 1) \geq h/2 \Rightarrow$$
$$h \leq 2\lg(n + 1)$$

Operations on Red-Black Trees

- The non-modifying binary-search tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in **O(h)** time
  - They take **O(lgn)** time on red-black trees
- What about TREE-INSERT and TREE-DELETE?
  - They will still run in **O(lgn)**
  - We have to guarantee that the modified tree will still be a red-black tree

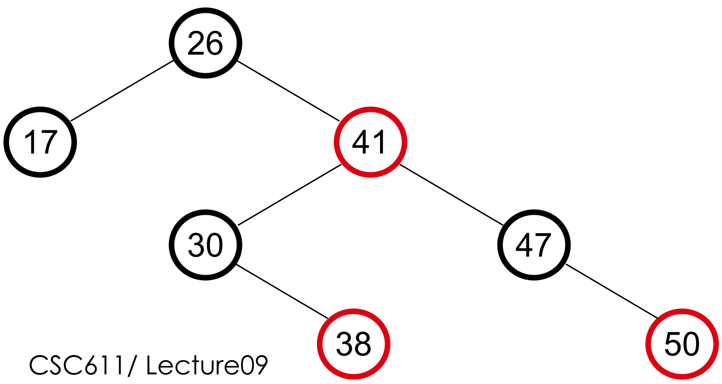
Insertion

- Goal:
  - Insert a new node z into a red-black tree
- Idea:
  - Insert node z into the tree as for an ordinary binary search tree
  - Color the node **red**
  - Restore the red-black tree properties
    - Use an auxiliary procedure RB-INSERT-FIXUP

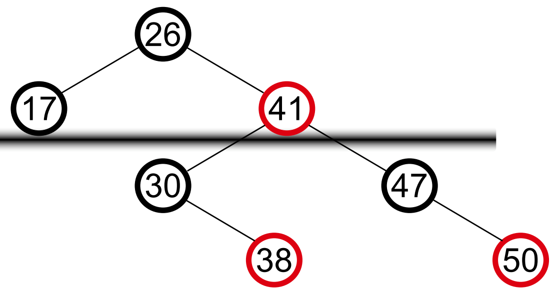
INSERT

INSERT: what color to make the new node?

- Red? Let's insert 35!
  - Property 4: if a node is red, then both its children are black
- Black? Let's insert 14!
  - Property 5: all paths from a node to its leaves contain the same number of black nodes



DELETE



DELETE: what color was the node that was removed? **Red?**

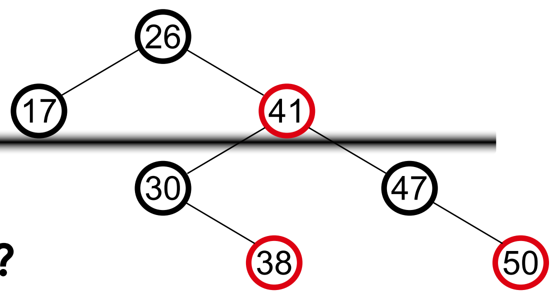
1. Every **node** is either **red** or **black** OK!
2. The **root** is **black** OK!
3. Every **leaf** (NIL) is **black** OK!
4. If a node is red, then both its children are black

OK! Does not change any black heights

OK! Does not create two red nodes in a row

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

DELETE



DELETE: what color was the node that was removed? **Black?**

1. Every **node** is either **red** or **black** OK!
2. The **root** is **black**
3. Every **leaf** (NIL) is **black** OK!
4. If a node is red, then both its children are black

Not OK! If removing the root and the child that replaces it is **red**

Not OK! Could change the black heights of some nodes

Not OK! Could create two red nodes in a row

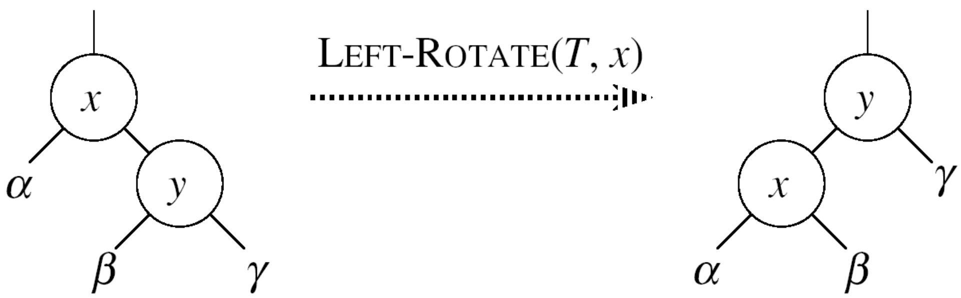
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Rotations

- Operations for restructuring the tree after insert and delete operations on red-black trees
- Rotations take a red-black tree and a node within the tree and:
  - Together with some node re-coloring they help restore the red-black tree property
  - Change some of the pointer structure
  - Do not change the binary-search tree property
- Two types of rotations:
  - Left & right rotations

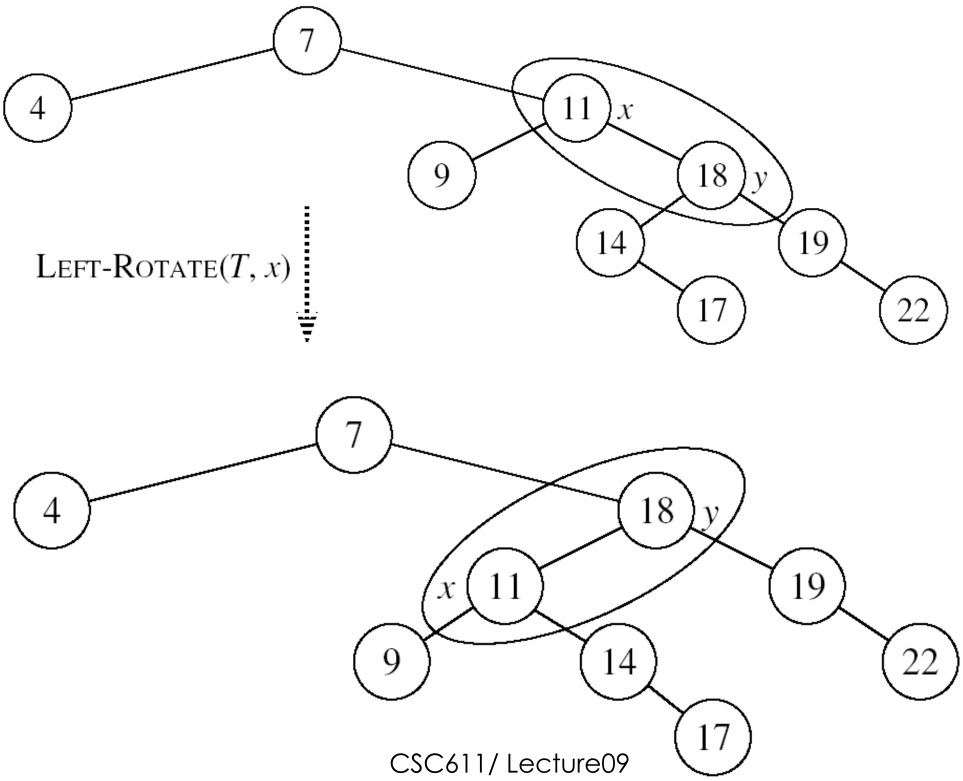
Left Rotations

- Assumption for a left rotation on a node **x**:
  - The right child of **x** (**y**) is not NIL



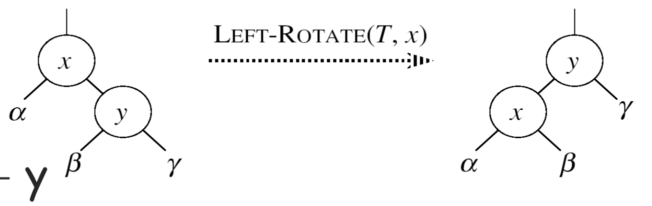
- Idea:
  - Pivots around the link from **x** to **y**
  - Makes **y** the new root of the subtree
  - **x** becomes **y**'s left child
  - **y**'s left child becomes **x**'s right child

Example: LEFT-ROTATE



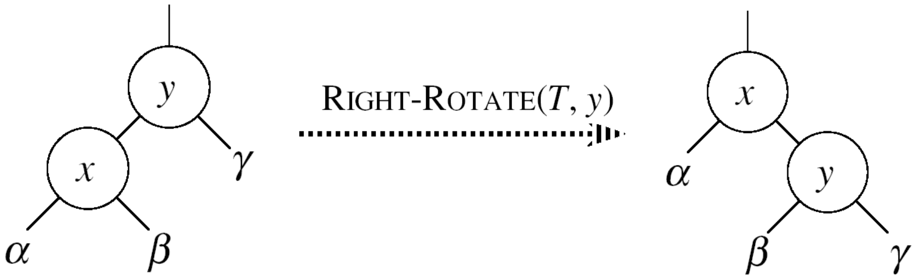
LEFT-ROTATE(T, x)

```
1.  y ← right[x]                 ►Set y
2.  right[x] ← left[y]           ► y's left subtree becomes x's right subtree
3.  if left[y] ≠ NIL
4.     then p[left[y]] ← x       ► Set the parent relation from left[y] to x
5.  p[y] ← p[x]                  ► The parent of x becomes the parent of y
6.  if p[x] = NIL
7.     then root[T] ← y
8.     else if x = left[p[x]]
9.             then left[p[x]] ← y
10.            else right[p[x]] ← y
11. left[y] ← x                  ► Put x on y's left
12. p[x] ← y                     ► y becomes x's parent
```



Right Rotations

- Assumption for a right rotation on a node **x**:
  - The left child of **y** (**x**) is not NIL

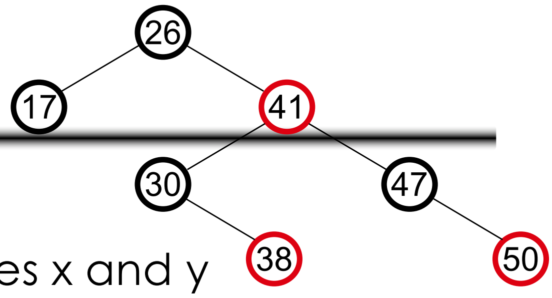


- Idea:
  - Pivots around the link from **y** to **x**
  - Makes **x** the new root of the subtree
  - **y** becomes **x**'s right child
  - **x**'s right child becomes **y**'s left child

Insertion

- Goal:
  - Insert a new node z into a red-black tree
- Idea:
  - Insert node z into the tree as for an ordinary binary search tree
  - Color the node **red**
  - Restore the red-black tree properties
    - Use an auxiliary procedure RB-INSERT-FIXUP

RB-INSERT(T, z)




```
y ← NIL
x ← root[T]
```

- Initialize nodes x and y
- Throughout the algorithm y points to the parent of x

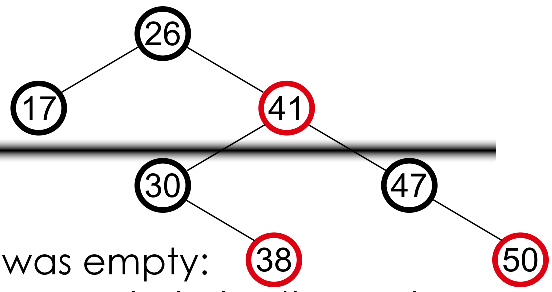
```
while x ≠ NIL
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
```

- Go down the tree until reaching a leaf
- At that point y is the parent of the node to be inserted

```
p[z] ← y
```

- Sets the parent of z to be y

RB-INSERT(T, z)




```
if y = NIL
    then root[T] ← z
```

The tree was empty: set the new node to be the root

```
    else if key[z] < key[y]
            then left[y] ← z
            else right[y] ← z
```

Otherwise, set z to be the left or right child of y, depending on whether the inserted node is smaller or larger than y's key

```
left[z] ← NIL
right[z] ← NIL
color[z] ← RED
```

Set the fields of the newly added node

```
RB-INSERT-FIXUP(T, z)
```

Fix any inconsistencies that could have been introduced by adding this new red node

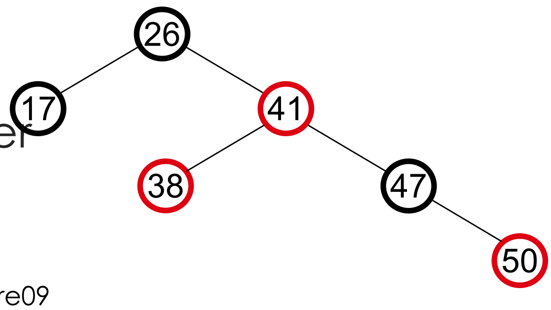
RB Properties Affected by Insert

1. Every **node** is either **red** or **black** — OK!
2. The **root** is **black** — If z is the root ⇒ not OK
3. Every **leaf** (NIL) is **black** — OK!
4. If a node is red, then both its children are black

If p(z) is red ⇒ not OK
z and p(z) are both red

OK!

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

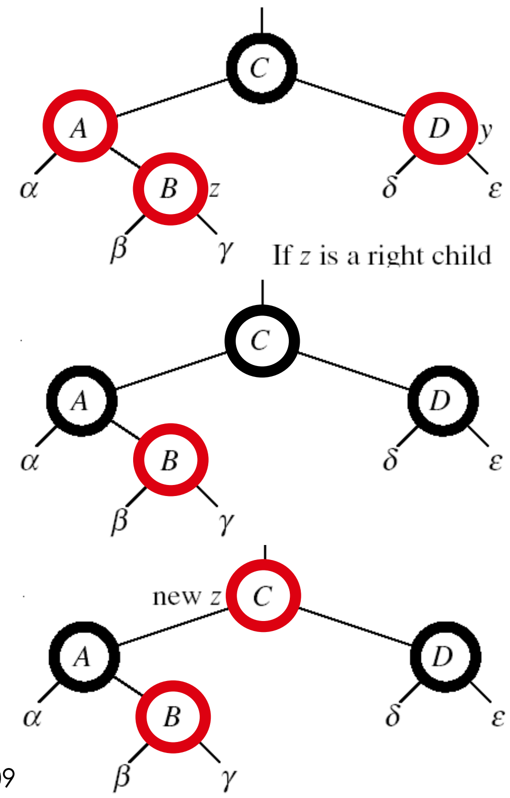


RB-INSERT-FIXUP – Case 1

z's "uncle" (y) is **red**

Idea: (z is a right child)

- **p[p[z]]** (z's grandparent) must be black: **z** and **p[z]** are both red
- Color **p[z] black**
- Color **y black**
- Color **p[p[z]] red**
  - Push the **red** node up the tree
- Make **z = p[p[z]]**

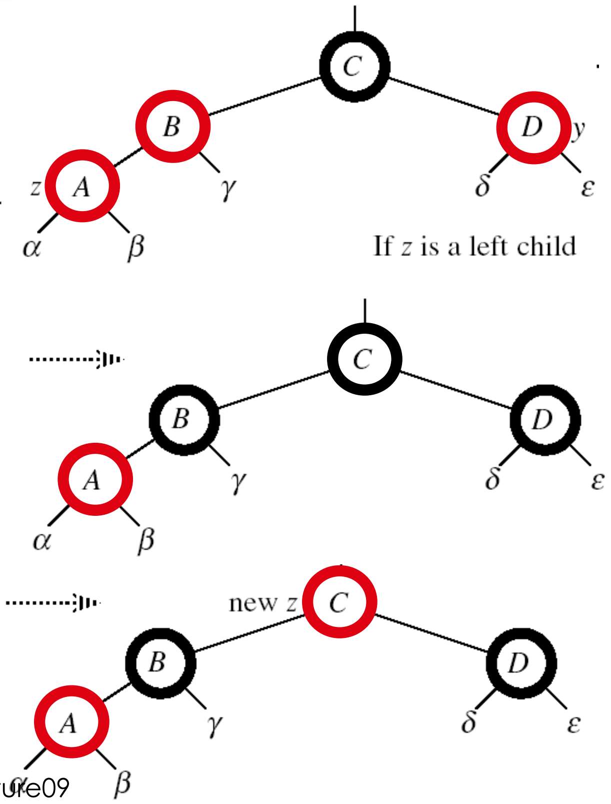


RB-INSERT-FIXUP – Case 1

z's "uncle" (y) is **red**

Idea: (**z** is a left child)

- **p[p[z]]** (**z**'s grandparent) must be black: **z** and **p[z]** are both red
- color[**p[z]**] ← **black**
- color[**y**] ← **black**
- color **p[p[z]]** ← **red**
- **z** = **p[p[z]]** Case1
  - Push the **red** node up the tree



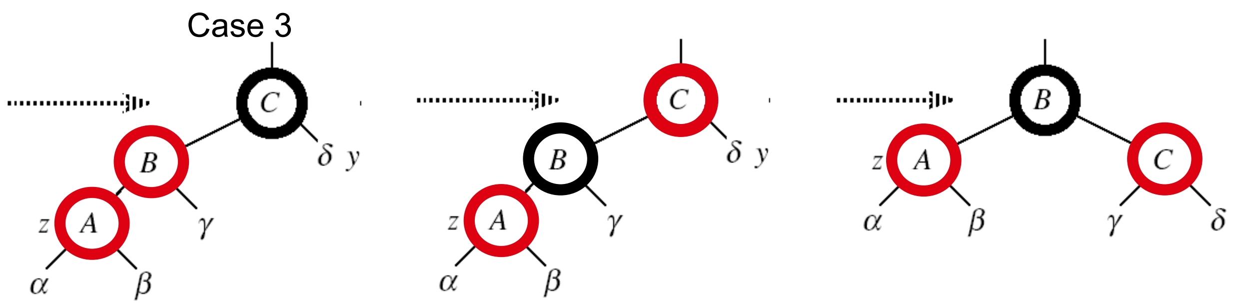
RB-INSERT-FIXUP – Case 3

Case 3:

- **z**'s "uncle" (y) is **black**
- **z** is a left child

Idea:

- color[p[z]] ← black
- color[p[p[z]]] ← red
- RIGHT-ROTATE(T, p[p[z]]) Case3
- No longer have 2 reds in a row
- p[z] is now black



RB-INSERT-FIXUP – Case 2

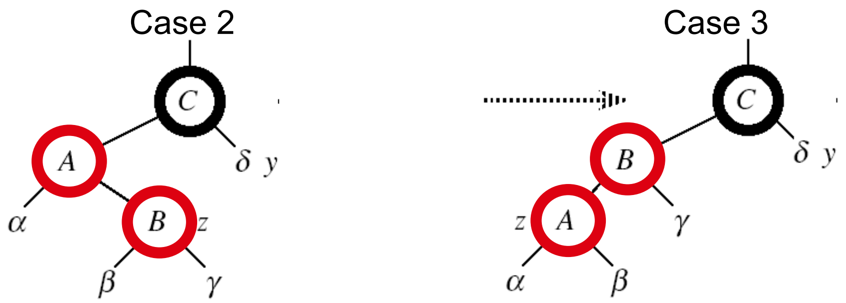
Case 2:

- **z**’s “uncle” (**y**) is **black**
- **z** is a right child

Idea:

- z ← p[z]
- LEFT-ROTATE(T, z) Case2

⇒ now z is a left child, and both z and **p[z]** are red ⇒ case 3



RB-INSERT-FIXUP(T, z)

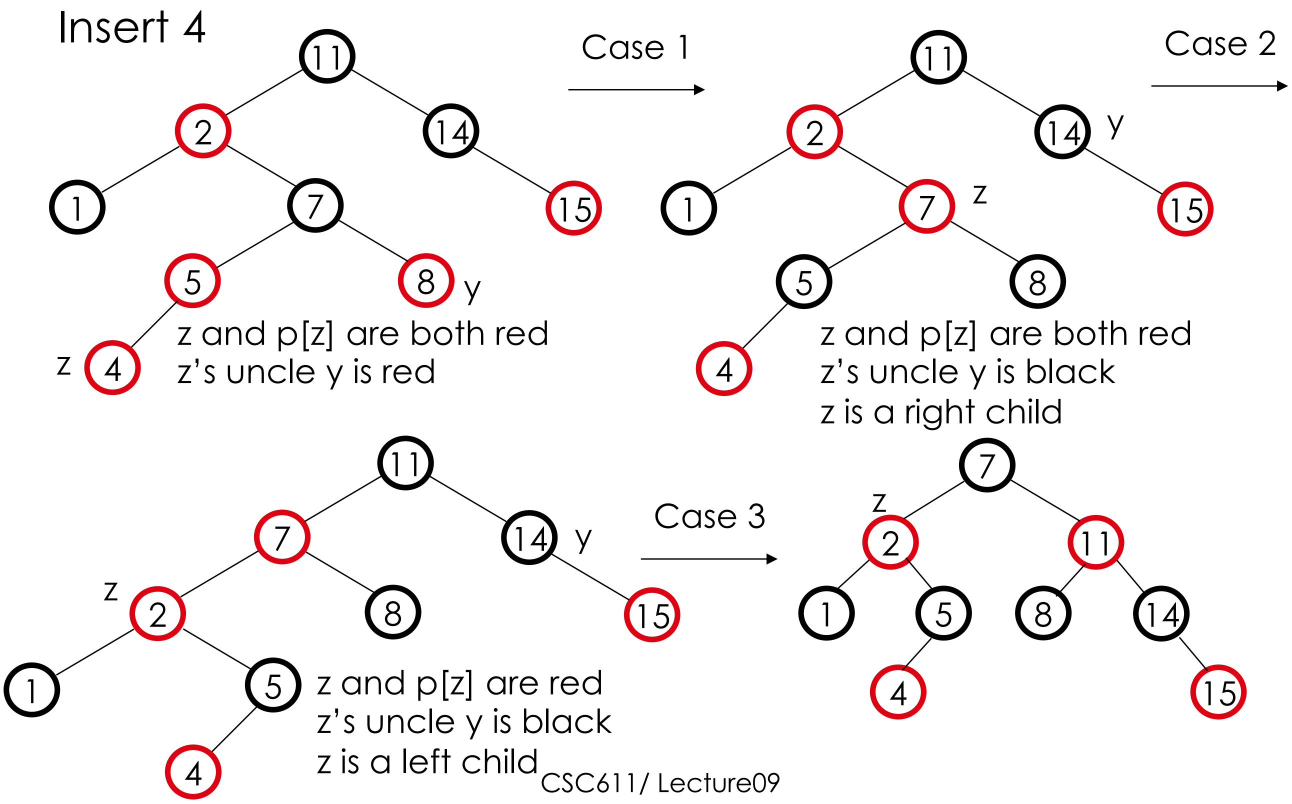
```
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
        then y ← right[p[p[z]]]
            if color[y] = RED
                then Case1
            else if z = right[p[z]]
                then Case2
            Case3
    else (same as then clause with "right"
          and "left" exchanged)
color[root[T]] ← BLACK
```

The while loop repeats only when case1 is executed: O(lgn) times

Set the value of x’s “uncle”

We just inserted the root, or the red node reached the root

Example



Analysis of RB-INSERT

- Inserting the new element into the tree $O(\lg n)$
- RB-INSERT-FIXUP
  - The while loop repeats only if CASE 1 is executed
  - The number of times the while loop can be executed is $O(\lg n)$
- Total running time of RB-INSERT: $O(\lg n)$

Red-Black Trees - Summary

- Operations on red-black trees:
  - SEARCH $O(h)$
  - PREDECESSOR $O(h)$
  - SUCCESOR $O(h)$
  - MINIMUM $O(h)$
  - MAXIMUM $O(h)$
  - INSERT $O(h)$
  - DELETE $O(h)$
- Red-black trees guarantee that the height of the tree will be $O(\lg n)$

Augmenting Data Structures

- Let's look at two new problems:
  - Dynamic order statistic
  - Interval search
- It is unusual to have to design all-new data structures from scratch
  - Typically: store additional information in an already known data structure
  - The augmented data structure can support new operations
- We need to correctly maintain the new information without loss of efficiency

Dynamic Order Statistics

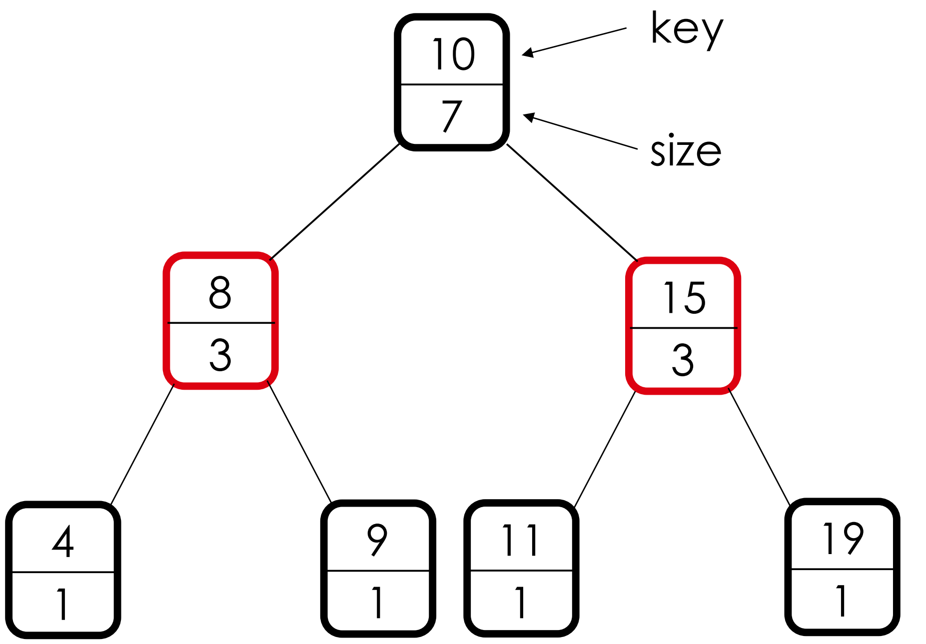
- *Def.:* the **i**-th order statistic of a set of **n** elements, where i ∈ {1, 2, …, n} is the element with the **i**-th smallest key.
- We can retrieve an order statistic from an unordered set:
  - Using: RANDOMIZED-SELECT
  - In: O(n) time
- We will show that:
  - With red-black trees we can achieve this in ***O*(lgn)**
  - Finding the **rank** of an element takes also ***O*(lgn)**

Order-Statistic Tree

- *Def.:* **Order-statistic tree:** a red-black tree with additional information stored in each node
- Node representation:
  - Usual fields: key[x], color[x], p[x], left[x], right[x]
  - Additional field: size[x] that contains the number of (internal) nodes in the subtree rooted at **x** (including **x** itself)
- For any internal node of the tree:

size[x] = size[left[x]] + size[right[x]] + 1

Example: Order-Statistic Tree



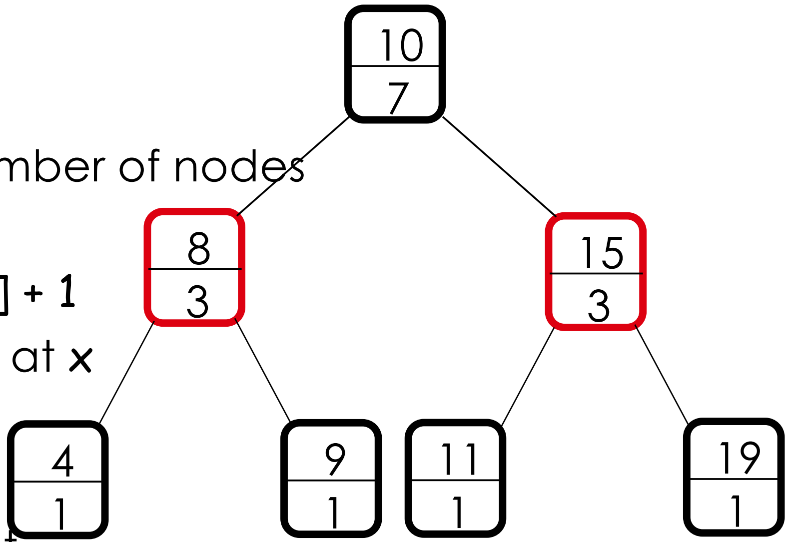
OS-SELECT

Goal:

- Given an order-statistic tree, return a pointer to the node containing the **i**-th smallest key in the subtree rooted at **x**

Idea:

- **size[left[x]]** = the number of nodes that are smaller than **x**
- **rank'[x] = size[left[x]] + 1** in the subtree rooted at **x**
- If **i = rank'[x]** Done!
- If **i < rank'[x]**: look left
- If **i > rank'[x]**: look right



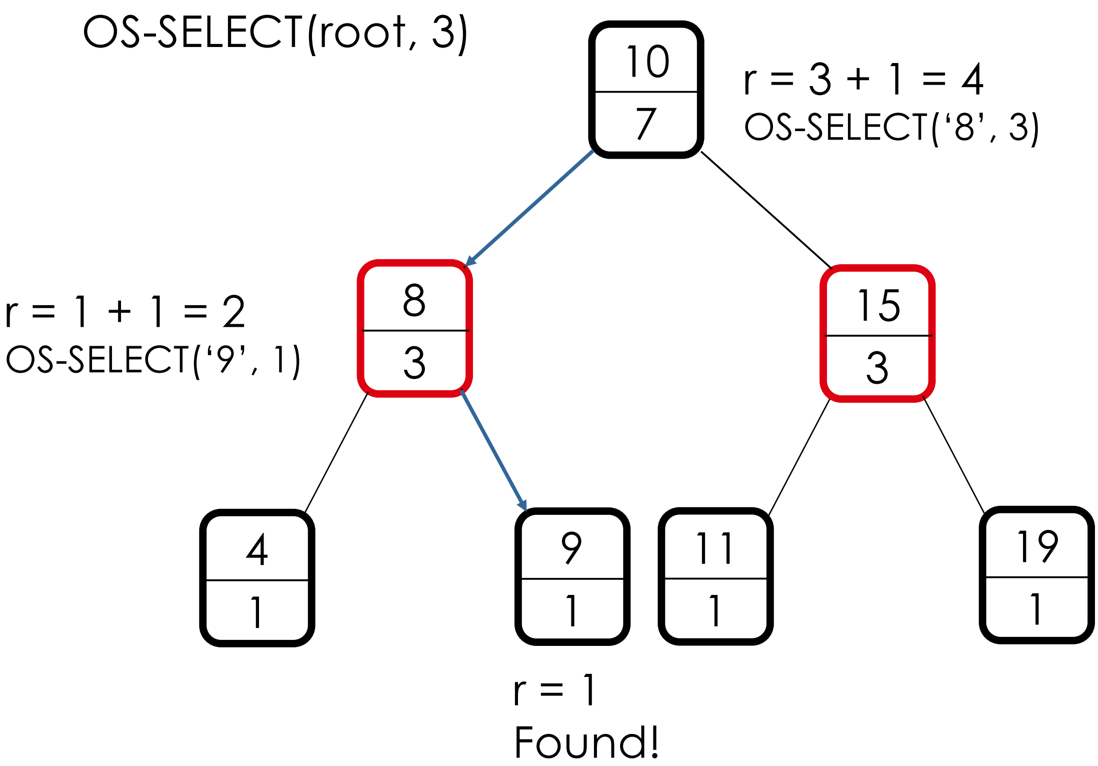
OS-SELECT(x, i)

```
r ← size[left[x]] + 1        ▶ compute the rank of x
                               within the subtree rooted at x
if i = r
   then return x
elseif i < r
        then return OS-SELECT(left[x], i)
else return OS-SELECT(right[x], i - r)
```

Initial call: OS-SELECT(root[T], i)

Running time: *O*(lgn)

Example: OS-SELECT



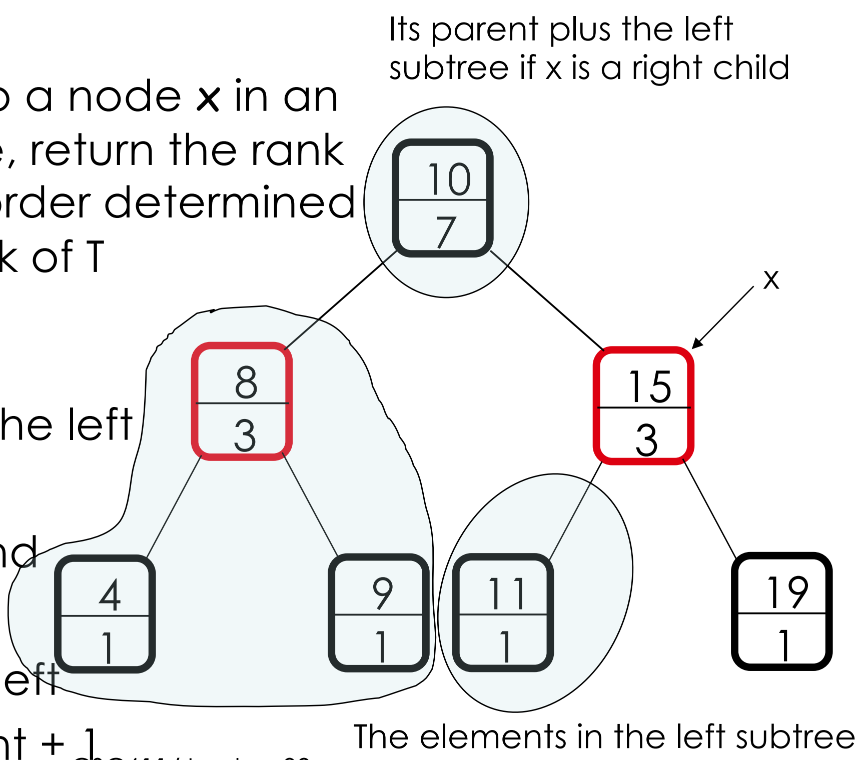
OS-RANK

Goal:

- Given a pointer to a node **x** in an order-statistic tree, return the rank of **x** in the linear order determined by an inorder walk of T

Idea:

- Add elements in the left subtree
- Go up the tree and if a right child: add the elements in the left subtree of the parent + 1



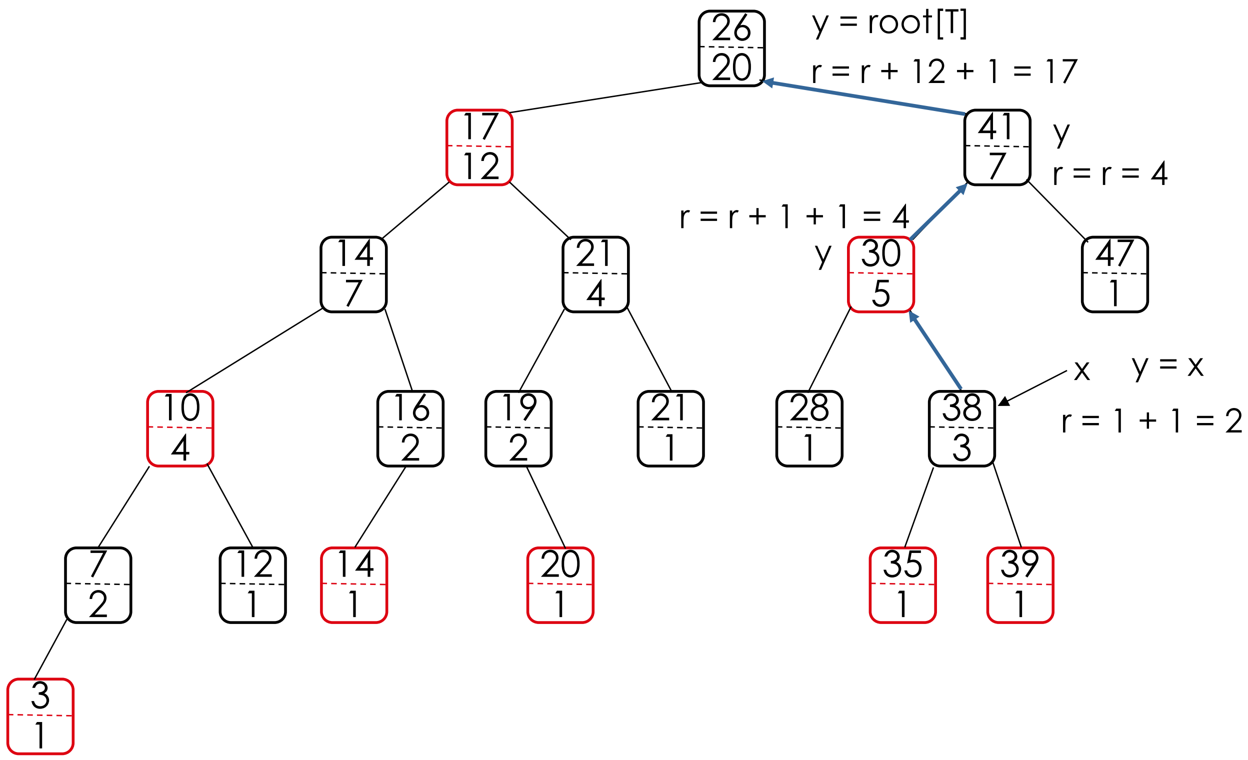
OS-RANK(T, x)

```
r ← size[left[x]] + 1          Add to the rank the elements in its left subtree + 1 for itself
y ← x                          Set y as a pointer that will traverse the tree
while y ≠ root[T]
    do if y = right[p[y]]
        then r ← r + size[left[p[y]]] + 1
    y ← p[y]
return r
```

If a right child add the size of the parent's left subtree + 1 for the parent

Running time: O(lgn)

Example: OS-RANK



Maintaining Subtree Sizes

- We need to maintain the **size** field during INSERT and DELETE operations
- Need to maintain them efficiently
- Otherwise, might have to recompute all **size** fields, at a cost of $\mathbf{\Omega(n)}$

Maintaining ***Size*** for OS-INSERT

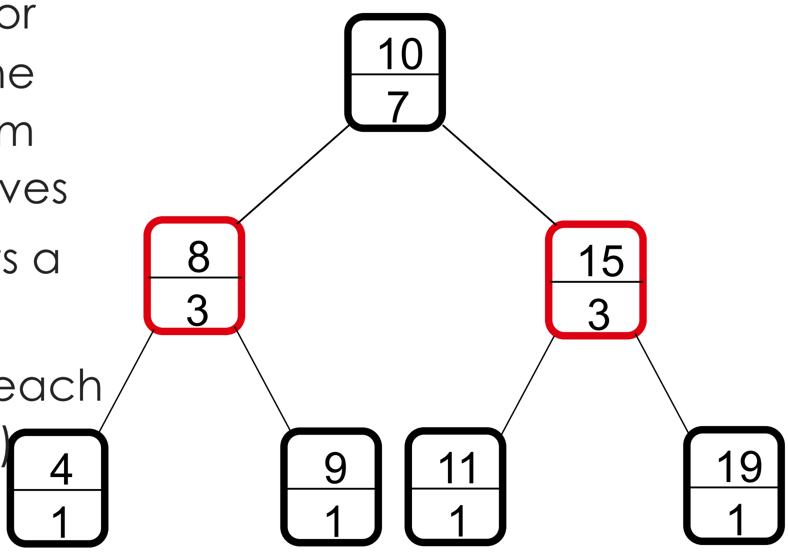
- Insert in a red-black tree has two stages
    1. Perform a binary-search tree insert
    2. Perform rotations and change node colors to restore red-black tree properties

OS-INSERT

Idea for maintaining the ***size*** field during insert

Phase 1 (going down):

- Increment size[x] for each node x on the traversed path from the root to the leaves
- The new node gets a size of 1
- Constant work at each node, so still O(lgn)

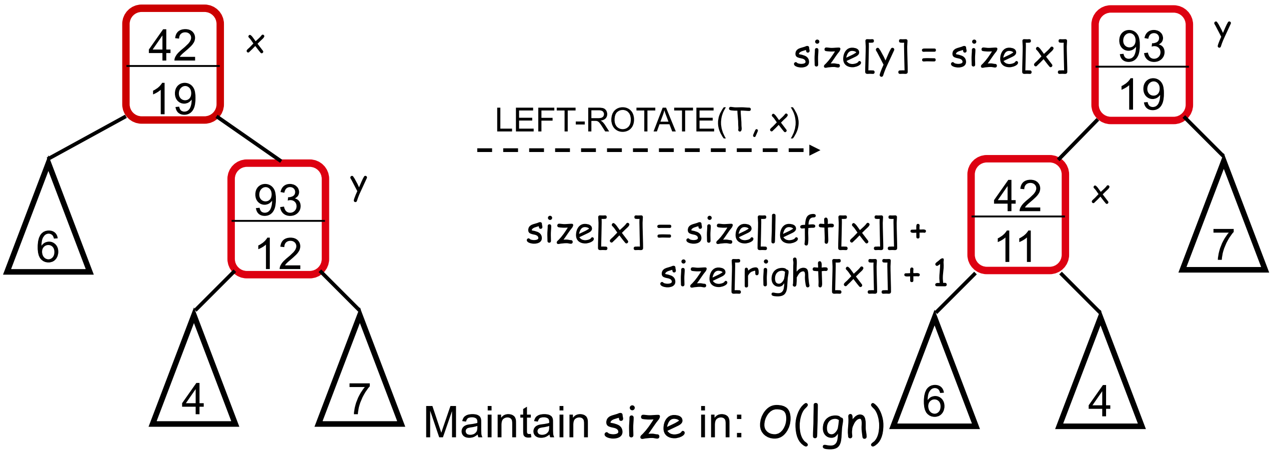


OS-INSERT

Idea for maintaining the ***size*** field during insert

Phase 2 (going up):

- During RB-INSERT-FIXUP there are:
  - *O*(lgn) changes in node colors
  - At most two rotations Rotations affect the subtree sizes !!



Maintain ***size*** in: *O*(lgn)

Augmenting a Data Structure

1. Choose an underlying data structure

   ⇒ Red-black trees

2. Determine additional information to maintain

   ⇒ size[x]

3. Verify that we can maintain additional information for existing data structure operations

   ⇒ Shown how to maintain size during modifying operations

4. Develop new operations

   ⇒ Developed OS-RANK and OS-SELECT

Augmenting Red-Black Trees

Theorem: Let **f** be a field that augments a red-black tree. If the contents of **f** for a node can be computed using only the information in **x**, **left[x]**, **right[x]** ⇒ we can maintain the values of **f** in all nodes during insertion and deletion, without affecting their **$O(\lg n)$** running time.

Examples

1. Can we augment a RBT with **size[x]**?

   Yes: **size[x] = size[left[x]] + size[right[x]] + 1**

2. Can we augment a RBT with **height[x]**?

   Yes: **height[x] = 1 + max(height[left[x]], height[right[x]])**

3. Can we augment a RBT with **rank[x]**?

   No, inserting a new minimum will cause all **n** rank values to change

Interval Trees

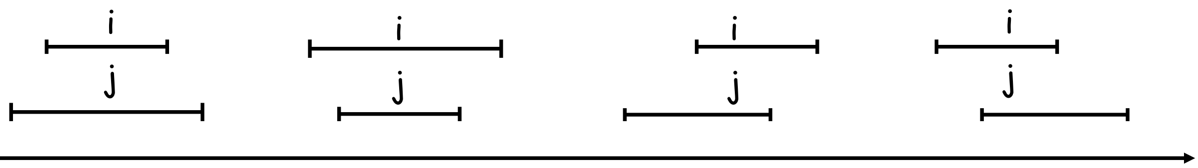
Def.: **Interval tree** = a red-black tree that maintains a dynamic set of elements, each element **x** having associated an interval **int[x]**.

- Operations on interval trees:
  - INTERVAL-INSERT(T, x)
  - INTERVAL-DELETE(T, x)
  - INTERVAL-SEARCH(T, i)

Interval Properties

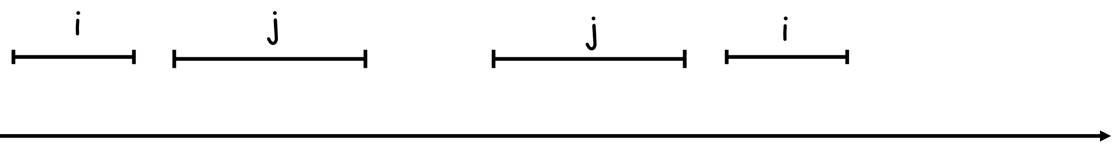
- Intervals **i** and **j** overlap iff:

  low[i] ≤ high[j] and **low[j] ≤ high[i]**



- Intervals **i** and **j** do not overlap iff:

  high[i] < low[j] or high[j] < low[i]



Interval Trichotomy

- Any two intervals **i** and **j** satisfy the **interval trichotomy**: exactly one of the following three properties holds:

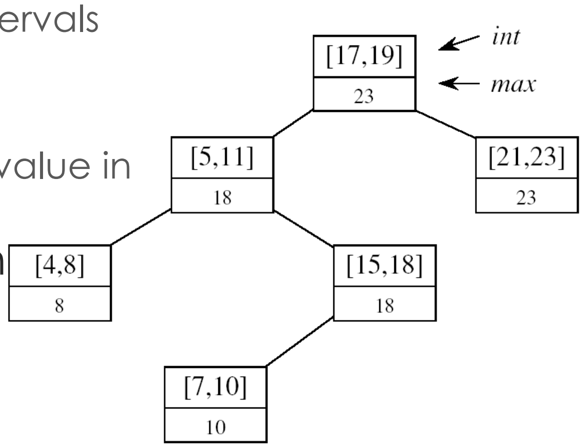
a) i and j overlap,

b) i is to the left of j (high[i] < low[j])

c) i is to the right of j (high[j] < low[i])

Designing Interval Trees

1. Underlying data structure
   - Red-black trees
   - Each node **x** contains: an interval int[x], and the key: low[int[x]]
   - An inorder tree walk will list intervals sorted by their low endpoint
2. Additional information
   - max[x] = maximum endpoint value in subtree rooted at **x**
3. Maintaining the information



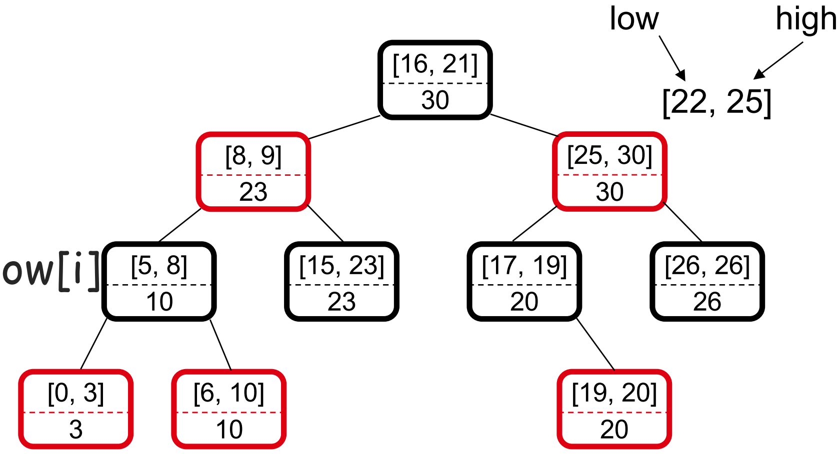
$$\text{max}[x] = \max \begin{cases} \text{high}[\text{int}[x]] \\ \text{max}[\text{left}[x]] \\ \text{max}[\text{right}[x]] \end{cases}$$

Constant work at each node, so still *O(lgn)* time

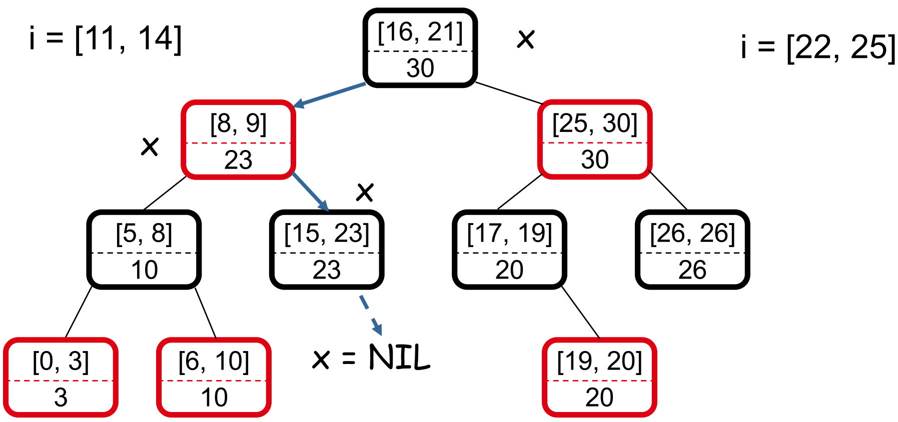
Designing Interval Trees

4. Develop new operations

- INTERVAL-SEARCH(T, i):
  - Returns a pointer to an element **x** in the interval tree T, such that **int[x]** overlaps with **i**, or **NIL** otherwise
- Idea:
- Check if **int[x]** overlaps with i
- **Max[left[x]] ≥ low[i]**
  - Go left
- Otherwise, go right



Example



INTERVAL-SEARCH(T, i)

1. x ← root[T]
2. **while** x ≠ nil[T] and i does not overlap int[x]
3. **do if** left[x] ≠ nil[T] and
   max[left[x]] ≥ low[i]
4. **then** x ← left[x]
5. **else** x ← right[x]
6. **return** x

Theorem

At the execution of interval search: if the search goes right, then either:
- There is an overlap in right subtree, or
- There is no overlap in either subtree

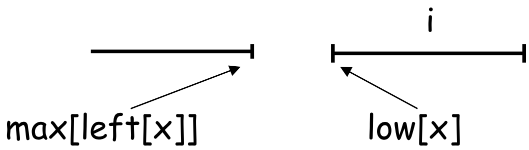
- Similar when the search goes left
- It is safe to always proceed in only one direction

Theorem

- **Proof:** If search goes right:
  - If there is an overlap in right subtree, done
  - If there is no overlap in right ⇒ show there is no overlap in left
  - Went right because:

left[x] = nil[T] ⇒ no overlap in left, or

max[left[x]] < low[i] ⇒ no overlap in left



Theorem - Proof

If search goes left:

- If there is an overlap in left subtree, done
- If there is no overlap in left, show there is no overlap in right
- Went left because:

  low[i] ≤ max[left[x]] = high[j] for some j in left subtree