ALGORITHMS – COMTEK3,CCT3 & ESD3

ORDER STATISTIC, BINARY SEARCH TREES AND DATA AUGMENTATION

Ramoni Adeogun
Associate Professor & Head AI for Communications
Wireless Communication Networks Section (WCN)
Department of Electronic Systems
Email: ra@es.aau.dk

AALBORG UNIVERSITET

Agenda

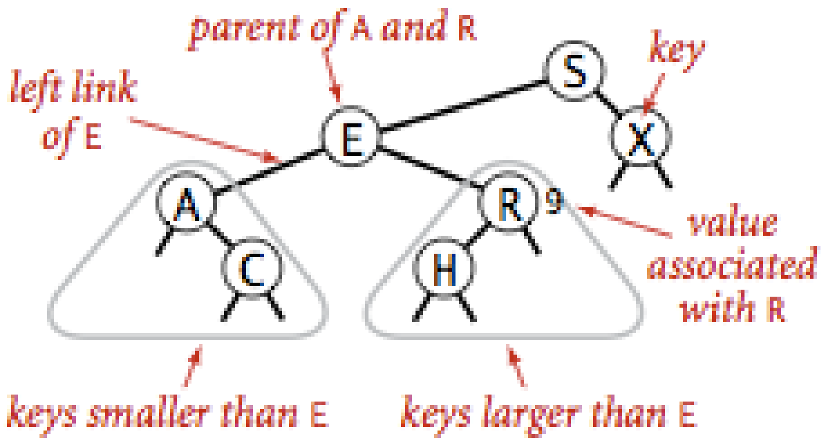
- **Order statistics**
  - Find maximum and minimum
  - Find median

- **Binary search trees**
  - Structure of trees
  - Operations to add, remove and search

- **Red-black trees**
  - Structure of trees
  - Operations to add, remove and search
  - Property maintenance

- **Data structure augmentation**

- Exercises



Anatomy of a binary search tree

Image source: https://algs4.cs.princeton.edu/32bst/

Order Statistics

- The ith ***order statistic*** of a set of n elements is the ith smallest element.

- Find the ith smallest (i.e. of rank i) of n elements, $A[1], A[2], \ldots, A[n]$, mutually comparable.
  - $i = 1$: the minimum, $i = n$: the maximum
  - Find both minimum and maximum
  - $i = (n+1)/2$ is the unique median when n is odd, otherwise $i = \lfloor (n+1)/2 \rfloor$ is the lower median and $i = \lceil (n+1)/2 \rceil$ the upper median

- A simple solution is to sort and pick element i, but then we are doing unnecessary work.

- Count comparisons between elements and assume the elements are distinct, for simplicity.

Find maximum

- Initialize *max* to $A[1]$ and compare it to $A[2]$. If $A[2]$ is larger it becomes the new *max*.
- The procedure of comparing *max* to a new element and possibly updating *max* is repeated until all n elements have been examined.
- This takes $n - 1$ comparisons between elements, which cannot be reduced, since each element but one must be compared to *max* or to some element that is smaller than *max*.

Find maximum and minimum

- To find both the largest and the smallest of n elements:
  - we can first find the largest in $n - 1$ comparisons, and then the smallest among the remaining $n - 1$ elements in $n - 2$ comparisons;
  - leading to a total of $2n - 3$ comparisons in total.
  - Can we do any faster? Oh yes!!
- Solve the problem for n elements by
  - removing two arbitrary elements, x and y, and solving the problem recursively for the remaining $n - 2$ elements.
  - The algorithm will return the largest and smallest elements among these $n - 2$ elements.
  - To get the maximum and minimum:
    - compare the larger of x and y to largest of $n - 2$ and the smaller to the minimum of $n - 2$ elements.
- The number of comparisons needed is hence expressed by the recurrence:

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ T(n-2) + 3 & \text{if } n > 2 \end{cases}$$

$T(n) = T(n-4) + 3 + 3 = T(n-6) + 3 + 3 + 3 \approx 3n/2$, or more precisely:

$$T(n-2) + 3 = \begin{cases} 3(n-2)/2 + T(2) = 3n/2 - 2 & \text{for even } n \\ 3(n-1)/2 + T(1) = (3n+1)/2 - 2 & \text{for odd } n \end{cases}$$

Selection in O(n)

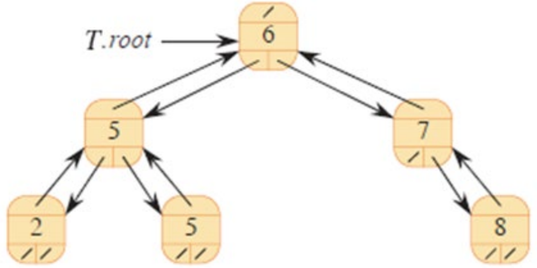
- More difficult problem than finding a minimum:
  - e.g., find an element with a rank around $n/2$ in an n −element array
  - This can be used to partition a file in smaller *quantiles* (when only a small part of the data is needed for later processing).
  - We will study an algorithm to do this in linear time –
- **RANDOMIZED-SELECT:**
  - similar to Quicksort and has $O(n)$ average case time
  - calls RANDOMIZED-PARTITION:
    - randomly chooses a pivot element $A[q]$ and splits $A[p, \ldots, r]$ so that elements less than the pivot are placed in $A[p, \ldots, q - 1]$,
    - and elements larger are placed in $A[q + 1, \ldots, r]$.
    - If the rank k (relative to p) of the pivot is greater than i,
      - the search continues among the smaller elements,
      - otherwise among the larger elements.

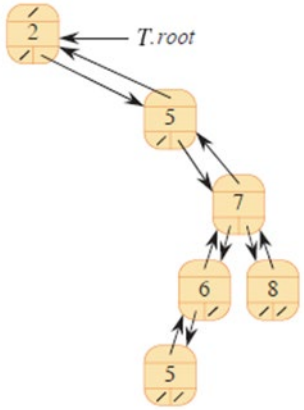
```
RANDOMIZED-SELECT(A, p, r, i)
if p == r
    return A[p]      // 1 ≤ i ≤ r − p + 1 when p == r means that i = 1
q = RANDOMIZED-PARTITION(A, p, r)
k = q − p + 1
if i == k
    return A[q]      // the pivot value is the answer
elseif i < k
    return RANDOMIZED-SELECT(A, p, q − 1, i)
else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

Binary Search Trees

Binary Search Trees

- **Binary search trees** are an important data structure for dynamic sets.
  - Accomplish many dynamic-set operations in $O(h)$ time, where h = height of the tree
  - We represent a binary tree by a linked data structure in which each node is an object.
    - T:root points to the root of tree T
    - Each node contains the attributes
      - key (and possibly other satellite data).
      - left: points to left child.
      - right: points to right child.
      - p: points to parent. T.root.p = NIL





- Stored keys must satisfy the binary-search-tree property.
  - If y is a node in the left subtree of x, then $y.key \leq x.key$
  - If y is a node in the right subtree of x, then $y.key > x.key$

INORDER-TREE-WALK

- The binary-search-tree property allows us to print keys in a binary search tree in order, recursively, using an algorithm called an **INORDER-TREE-WALK**.
  - Elements are printed in monotonically increasing order

- How INORDER-TREE-WALK works:
  - Check to make sure that x is not NIL.
  - Recursively, print the keys of the nodes in x's left subtree.
  - Print x's key.
  - Recursively, print the keys of the nodes in x's right subtree.

- **Example**: Apply INORDER-TREE-WALK to the trees on the previous slide

- **Time:**
  - Intuitively, the walk takes $\Theta(n)$. Why?

```
INORDER-TREE-WALK(x)
if x ≠ NIL
    INORDER-TREE-WALK(x.left)
    print x.key
    INORDER-TREE-WALK(x.right)
```

Operations in binary search trees

- Types of operations
  - Looking for a specific key
  - Minimum
  - Maximum
  - Next
  - Previous

- All can be carried out in $O(h)$ time with h as the height of the tree.

- In worst case, the tree is just n elements high, and in best $\log_2(n)$

Search in a binary tree

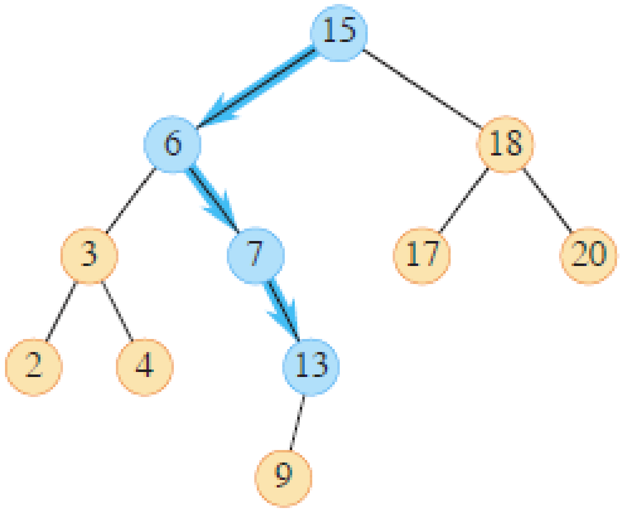
Goal: Search for a node in a BST

Inputs:

- x – a pointer to the root
- k – key for the node to search

Examples:

- k = 13: 15 → 6 → 7 → 13
- k = 4: 15 → 6 → 3 → 4




```
TREE-SEARCH(x, k)
if x == NIL or k == x.key
    return x
if k < x.key
    return TREE-SEARCH(x.left, k)
else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
while x ≠ NIL and k ≠ x.key
    if k < x.key
        x = x.left
    else x = x.right
return x
```

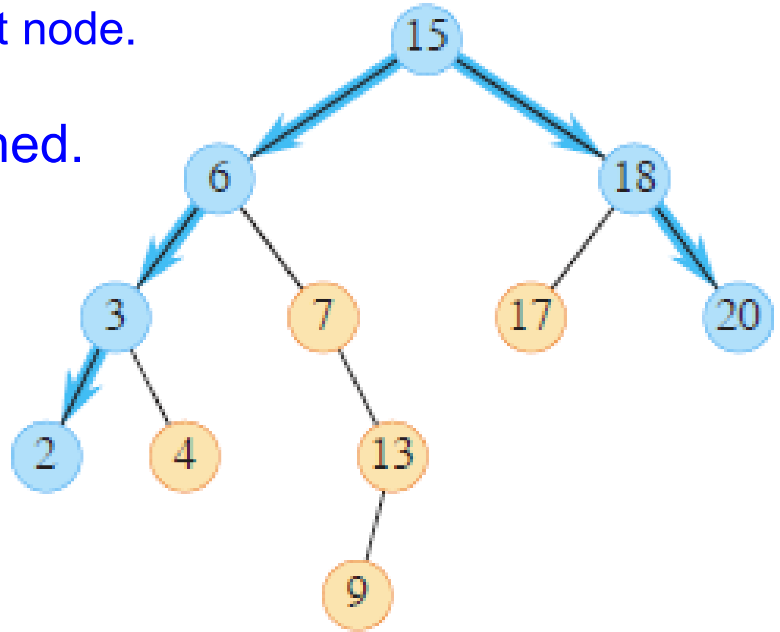
Minimum and maximum

- The binary-search-tree property guarantees that
  - the minimum key of a binary search tree is located at the leftmost node, and
  - the maximum key of a binary search tree is located at the rightmost node.

- Traverse the appropriate pointers (left or right) until NIL is reached.

```
TREE-MINIMUM(x)
while x.left ≠ NIL
    x = x.left
return x
```

```
TREE-MAXIMUM(x)
while x.right ≠ NIL
    x = x.right
return x
```



- Both procedures visit nodes that form a downward path from the root to a leaf.
  - Both procedures run in $O(h)$ time, where h is the height of the tree.

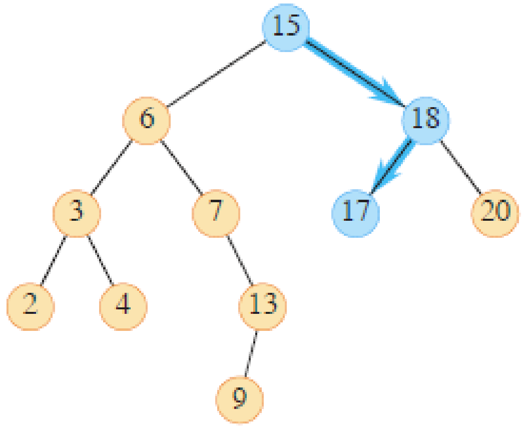
Successor and predecessor

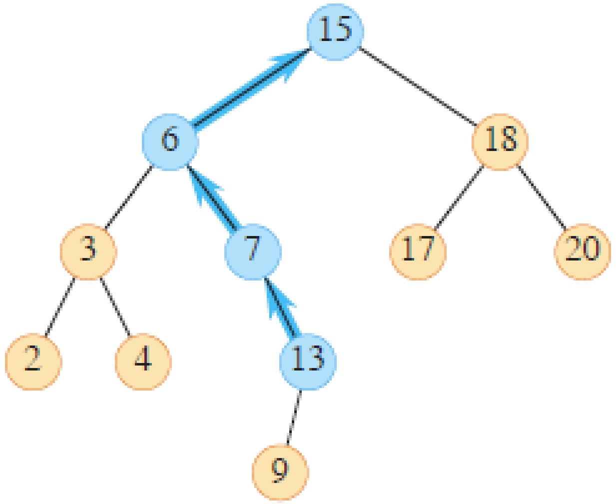
Assuming that all keys are distinct, the successor of a node x is the node y such that y.key is the smallest key > x.key. If x has the largest key in BST, then we say that x's successor is NIL.

There are two cases:

1. If node x has a non-empty right subtree, then x's successor is the minimum in x's right subtree.
2. If node x has an empty right subtree, notice that:
   - If we move to the left up the tree (move up through right children), we're visiting smaller keys.
   - x's successor y is the node that x is the predecessor of (x is the maximum in y's left subtree).

```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)   //
  else // find the lowest ancestor of x whose
      y = x.p
      while y ≠ NIL and x == y.right
          x = y
          y = y.p
      return y
```





- Note: TREE-PREDECESSOR is symmetric to TREE-SUCCESSOR

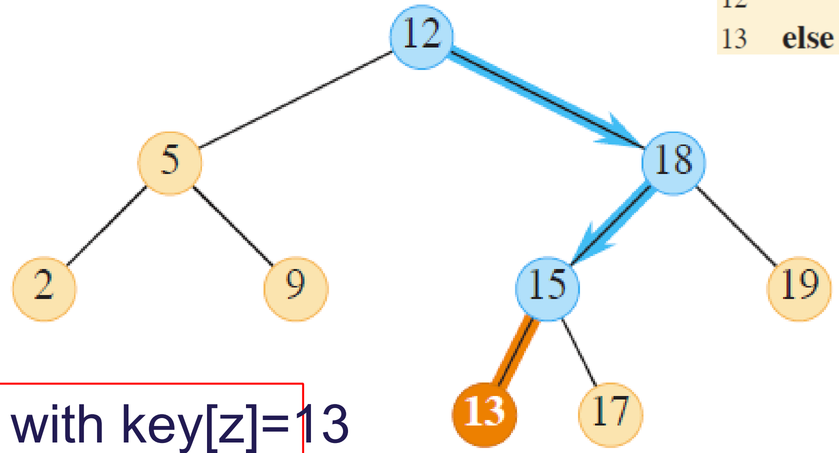
Insert into a BST

- Simple to insert, but we must retain the property of the binary tree

- Go through the tree and find an empty spot that matches criteria
  - Keys must obey binary tree property

- Update pointers

```
TREE-INSERT(T, z)
x = T.root               // node being compared with z
y = NIL                  // y will be parent of z
while x ≠ NIL            // descend until reaching a leaf
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                  // found the location—insert z with parent y
if y == NIL
    T.root = z           // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
```



Example: Insert node z with key[z]=13

Deleting nodes from a BST

- Conceptually, deleting node z from binary search tree T has three cases:
  1. If z has no children, just remove it
  2. If z has just one child, then make that child take z's position in the tree, dragging the child's subtree along.
  3. If z has two children, then find z's successor y and replace z by y in the tree.
     - y must be in z's right subtree and have no left child.
     - The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree.

```
TRANSPLANT(T, u, v)
if u.p == NIL
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
if v ≠ NIL
    v.p = u.p
```

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    if y ≠ z.right
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

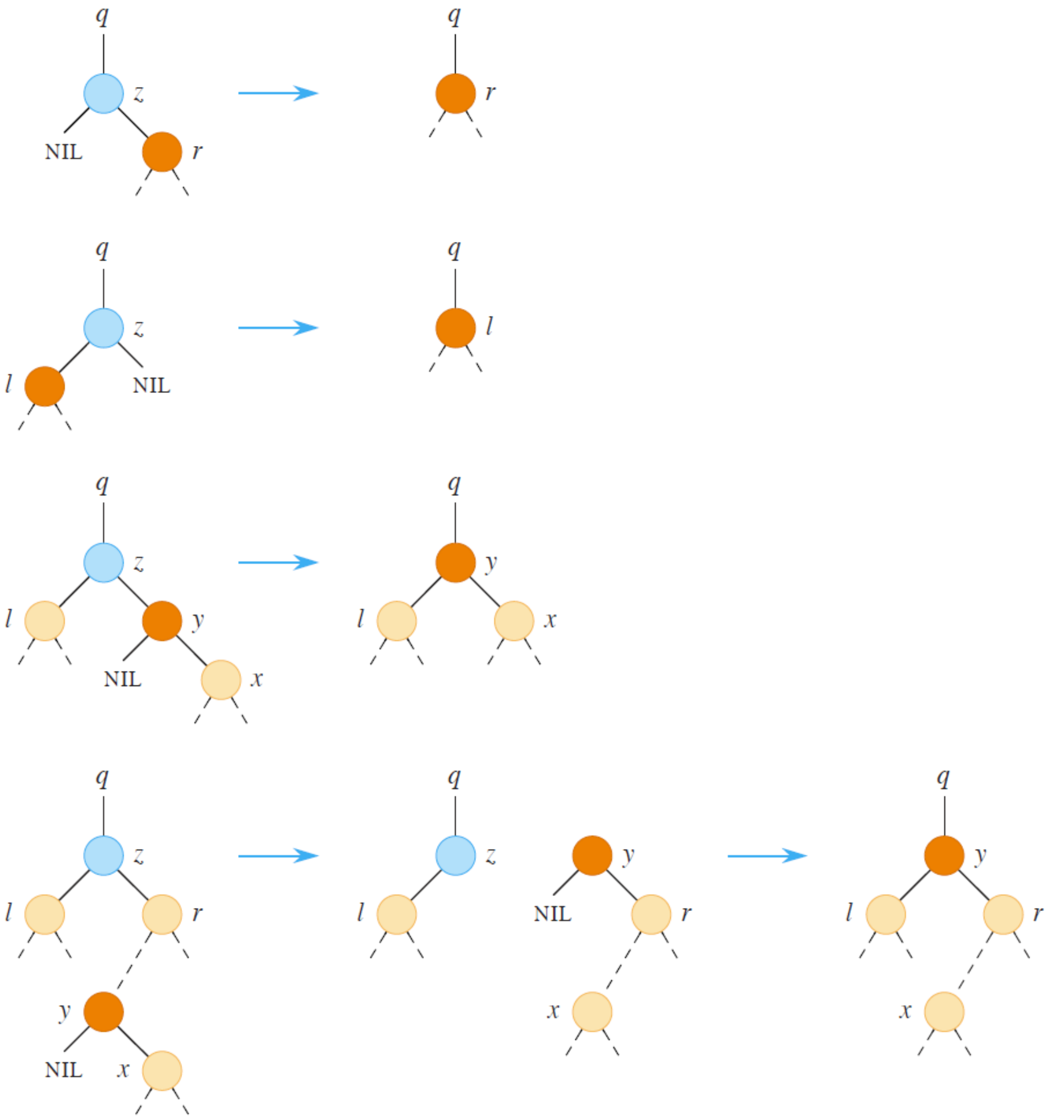
Deleting from a BST (2)

1. If z has no left child, replace ´ by its right child. The right child may or may not be NIL.

2. If z has just one child, and that child is its left child, then replace ´ by its left child.

3. Otherwise, z has two children. Find z's successor y. y must lie in z's right subtree and have no left child
   I. If y is z's right child, replace z by y and leave y's right child alone.

   II. Otherwise, y lies within z's right subtree but is not the root of this subtree. Replace y by its own right child. Then replace z by y.



Red Black Trees

Red-black Trees

- A red-black tree is a binary search tree with an extra attribute: an attribute color, which is either red or black.

- All leaves are empty (nil) and colored black.
  - We use a single sentinel, T.nil, for all the leaves of red-black tree T .
  - T.nil.color is black.
  - The root's parent is also T.nil.

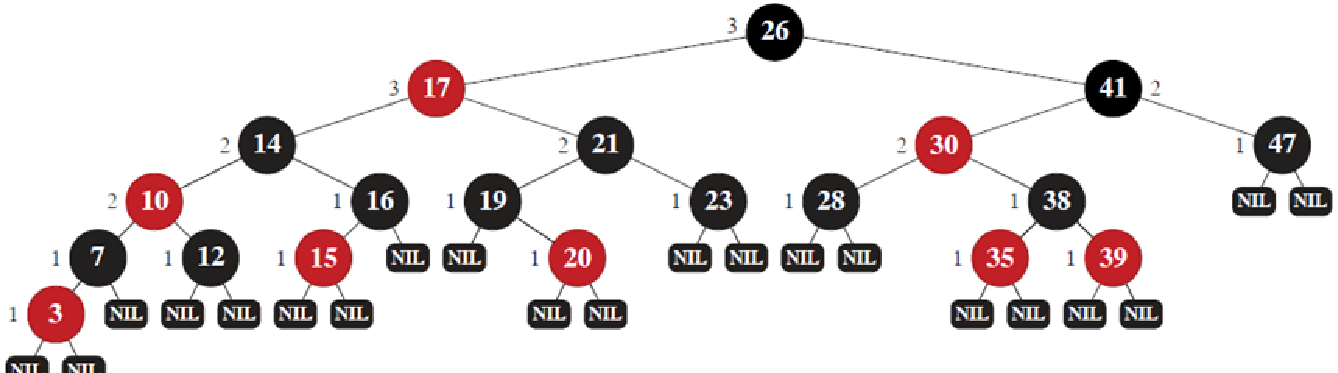
- All other attributes of binary search trees are inherited by red-black trees (key, left, right, and p). We don't care about the key in T.nil.

Red-black tree properties

1. Every node is **either red or black**.
2. The **root is black**.
3. Every **leaf (T.nil) is black**.
4. If a node **is red, then both its children are black =>** no two reds 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**.

Height of a red-black tree

- **Height of a node** is 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 T.nil) on the path from x to leaf, not counting x.
  - By property 5, black-height is well defined.



(a)

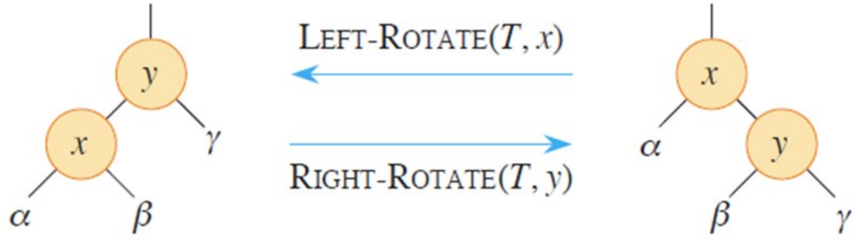
- **Claim**: Any node with height h has black-height $\geq h/2$
  - Proof: By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ nodes are black

Operations on red-black trees

- The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(height)$ time. Thus, they take $O(\log_2(n))$ time on red-black trees.
- Insertion and deletion are not so easy.
  - If we insert, what color to make the new node?
    - Red? Might violate property 4.
    - Black? Might violate property 5.
  - If we delete, thus removing a node, what color was the node that was removed?
    - Red? OK, since we won't have changed any black-heights, nor will we have created two red nodes in a row. Also, cannot cause a violation of property 2, since if the removed node was red, it could not have been the root.
    - Black? Could cause there to be two reds in a row (violating property 4), and can also cause a violation of property 5. Could also cause a violation of property 2, if the removed node was the root and its child—which becomes the new root—was red.

Rotations

- The basic tree-restructuring operation needed to maintain red-black trees as balanced binary search trees.
- Changes the local pointer structure. (Only pointers are changed.)
- Have both left rotation and right rotation. They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree.




```
LEFT-ROTATE(T, x)
  y = x.right
  x.right = y.left
  if y.left ≠ T.nil
      y.left.p = x
  y.p = x.p
  if x.p == T.nil
      T.root = y
  elseif x == x.p.left
      x.p.left = y
  else x.p.right = y
  y.left = x
  x.p = y
```

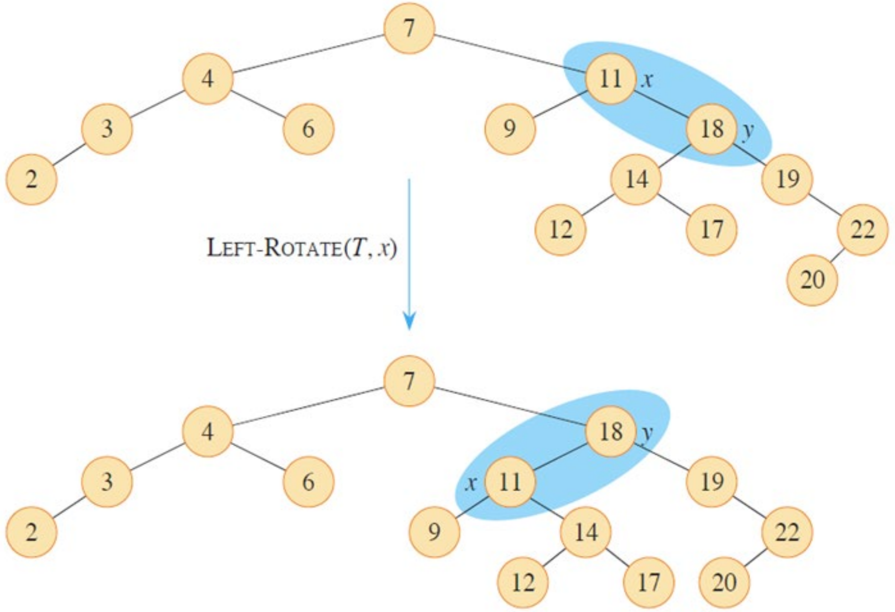
- The pseudocode for LEFT-ROTATE assumes that
  - $x.right \neq T.nil$, and
  - root's parent is $T.nil$.
- Pseudocode for RIGHT-ROTATE is symmetric: exchange left and right everywhere

Rotations: Example

- **Before rotation**: keys of x's left subtree $\leq$11 $\leq$ keys of y's left subtree $\leq$ 18 $\leq$ keys of y's right subtree.
- Rotation makes y's left subtree into x's right subtree.
- **After rotation**: keys of x's left subtree $\leq$ 11 $\leq$ keys of x's right subtree $\leq$ 18 $\leq$ keys of y's right subtree.

Time: $O(1)$ for both LEFT-ROTATE & RIGHT-ROTATE

- a constant number of pointers are modified.



Insertion

- Similar to insertion in regular binary-search-tree insertion.

- RB-INSERT ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.

- Which property might be violated
  1. OK
  2. If z is the root, then there's a violation. Otherwise, OK.
  3. OK.
  4. If $z.p$ is red, there's a violation: both z and $z.p$ are red.
  5. OK.

- Remove the violation by calling RB-INSERT-FIXUP.

```
RB-INSERT(T, z)
x = T.root                    // node being compared with z
y = T.nil                     // y will be parent of z
while x ≠ T.nil               // descend until reaching the sentinel
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y                       // found the location—insert z with parent y
if y == T.nil
    T.root = z                // tree T was empty
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil                // both of z's children are the sentinel
z.right = T.nil
z.color = RED                 // the new node starts out red
RB-INSERT-FIXUP(T, z)         // correct any violations of red-black properties
```

RB-Insert Fix-up: Example

Loop invariant:

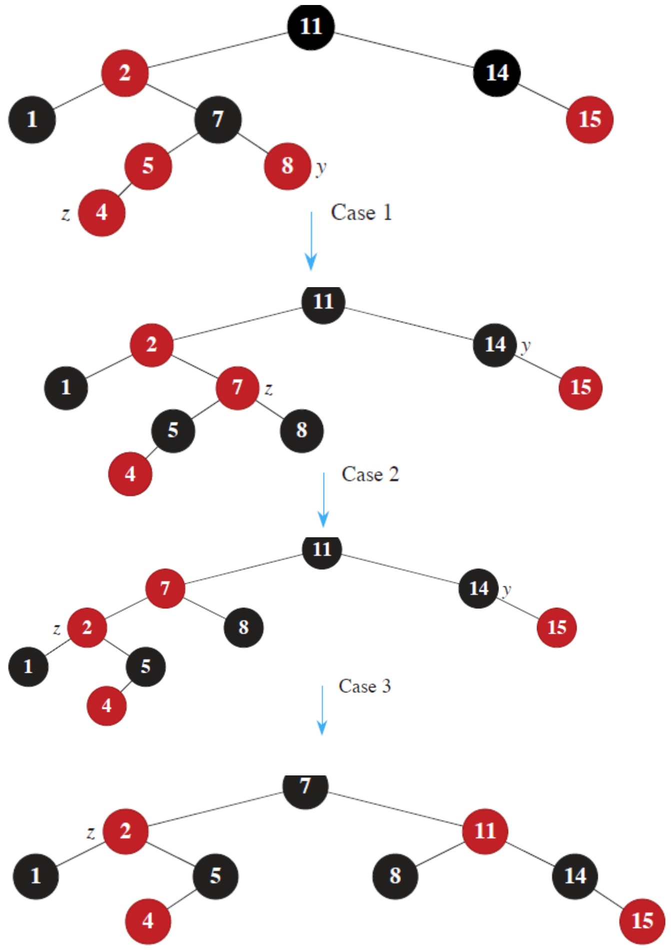
- Node z is red

- If $z.p$ is the root, then $z.p$ is black.

- There is at most one red-black violation:
  - Property 2: z is a red root, or
  - Property 4: z and $z.p$ are both red.

- Analysis:
- $O(\log_2(n))$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left            // is z's parent a left child?
3          y = z.p.p.right             // y is z's uncle
4          if y.color == RED           // are z's parent and uncle both red?
5              z.p.color = BLACK       ⎫
6              y.color = BLACK         ⎬ case 1
7              z.p.p.color = RED       ⎪
8              z = z.p.p               ⎭
9          else
10             if z == z.p.right
11                 z = z.p             ⎫ case 2
12                 LEFT-ROTATE(T, z)   ⎭
13             z.p.color = BLACK       ⎫
14             z.p.p.color = RED       ⎬ case 3
15             RIGHT-ROTATE(T, z.p.p)  ⎭
16     else // same as lines 3–15, but with "right" and "left" exchanged
17         y = z.p.p.left
18         if y.color == RED
19             z.p.color = BLACK
20             y.color = BLACK
21             z.p.p.color = RED
22             z = z.p.p
23         else
24             if z == z.p.left
25                 z = z.p
26                 RIGHT-ROTATE(T, z)
27             z.p.color = BLACK
28             z.p.p.color = RED
29             LEFT-ROTATE(T, z.p.p)
30 T.root.color = BLACK
```



Deletion in RB-Trees

- Based on the TREE-DELETE procedure for BSTs:

```
RB-DELETE-FIXUP(T, x)
1  while x ≠ T.root and x.color == BLACK
2      if x == x.p.left                  // is x a left child?
3          w = x.p.right                 // w is x's sibling
4          if w.color == RED
5              w.color = BLACK              ⎫
6              x.p.color = RED              ⎪
7              LEFT-ROTATE(T, x.p)          ⎬ case 1
8              w = x.p.right                ⎭
9          if w.left.color == BLACK and w.right.color == BLACK
10             w.color = RED                ⎫ case 2
11             x = x.p                      ⎭
12         else
13             if w.right.color == BLACK
14                 w.left.color = BLACK     ⎫
15                 w.color = RED            ⎪
16                 RIGHT-ROTATE(T, w)       ⎬ case 3
17                 w = x.p.right            ⎭
18             w.color = x.p.color          ⎫
19             x.p.color = BLACK            ⎪
20             w.right.color = BLACK        ⎬ case 4
21             LEFT-ROTATE(T, x.p)          ⎪
22             x = T.root                   ⎭
23     else // same as lines 3–22, but with "right" and "left" exchanged
24         w = x.p.left
25         if w.color == RED
26             w.color = BLACK
27             x.p.color = RED
28             RIGHT-ROTATE(T, x.p)
29             w = x.p.left
30         if w.right.color == BLACK and w.left.color == BLACK
31             w.color = RED
32             x = x.p
33         else
34             if w.left.color == BLACK
35                 w.right.color = BLACK
36                 w.color = RED
37                 LEFT-ROTATE(T, w)
38                 w = x.p.left
39             w.color = x.p.color
40             x.p.color = BLACK
41             w.left.color = BLACK
42             RIGHT-ROTATE(T, x.p)
43             x = T.root
44 x.color = BLACK
```

```
RB-TRANSPLANT(T, u, v)
if u.p == T.nil
    T.root = v
elseif u == u.p.left
    u.p.left = v
else u.p.right = v
v.p = u.p
```

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)          // replace z by its right child
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)           // replace z by its left child
else y = TREE-MINIMUM(z.right)            // y is z's successor
    y-original-color = y.color
    x = y.right
    if y ≠ z.right                        // is y farther down the tree?
        RB-TRANSPLANT(T, y, y.right)      // replace y by its right child
        y.right = z.right                 // z's right child becomes
        y.right.p = y                     //     y's right child
    else x.p = y                          // in case x is T.nil
    RB-TRANSPLANT(T, z, y)                // replace z by its successor y
    y.left = z.left                       // and give z's left child to y,
    y.left.p = y                          //     which had no left child
    y.color = z.color
if y-original-color == BLACK              // if any red-black violations occurred,
    RB-DELETE-FIXUP(T, x)                 //     correct them
```

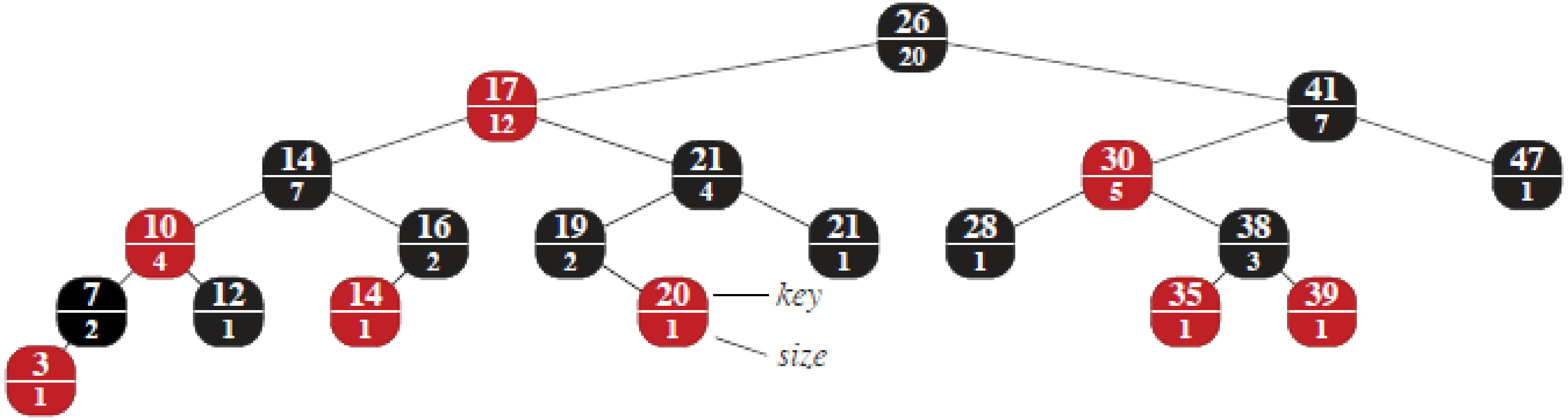
Dynamic Order Statistics and Data Augmentation

Augmenting Data Structures

- Suppose we have a base data structure D that efficiently handles a standard set of operations. **e.g., D is a RBT that supports operations SEARCH, INSERT, and DELETE**
- In some applications, we desire a data structure that support an additional set of operations. For example, **the order-statistics operations SELECT and RANK**
- So, how do we efficiently implement the new operations without degrading the existing ones?
- By augmenting the data structure:
  - Choose underlying data structure, for instance a red-black tree.
  - Determine additional information to be maintained, for instance sizes of subtrees.
  - Verify that additional information is updated correctly for the operations on the data structure.
  - Develop new operations.

Dynamic Order Statistics

- Require other operations in addition to standard dynamic set operations
  - OS-SELECT(x, i) – returns ith smallest key in subtree rooted at x
  - OS-RANK(T, x) – returns rank of x in the linear order determined by an inorder traversal of T
- *Idea:* Store sizes of subtrees in the nodes in a red-black tree => **order-statistic tree** .



Operations

- OS-SELECT for retrieving an element with a given rank
- OS-RANK for determining the rank of an element
- Both OS-SELECT and OS-RANK takes O(log n) time.

```
OS-SELECT(x, i)
r = x.left.size + 1      // rank of x within the subtree rooted at x
if i == r
    return x
elseif i < r
    return OS-SELECT(x.left, i)
else return OS-SELECT(x.right, i - r)
```

```
OS-RANK(T, x)
r = x.left.size + 1              // rank of x within the subtree rooted at x
y = x                            // root of subtree being examined
while y ≠ T.root
    if y == y.p.right            // if root of a right subtree ...
        r = r + y.p.left.size + 1   // ... add in parent and its left subtree
    y = y.p                      // move y toward the root
return r
```

Summary

- **Binary search trees** are an important data structure for dynamic sets.
  - Accomplish many dynamic-set operations in $O(h)$ time, where h = height of tree.
  - T.root points to the root of tree T .
  - Each node contains the attributes: key (and possibly other satellite data), left - points to left child, right - points to right child and p - points to parent. T.root.p = NIL
- **Red black trees**:
  - A variation of binary search trees with colours – red or black attribute.
  - Balanced: height is $O(\log_2(n))$ where n is the number of nodes.
  - Operations will take $O(\log_2(n))$ time in the worst case.
- **Order Statistic:**
  - ith order statistic is the ith smallest element of a set of n elements
  - Applications in finding minimum, maximum, median, selection, etc
- **Data augmentation**:
  - Principle used in applications in which we desire a data structure that support an additional set of operations beyond a base structure