CSC 317: Data Structures and Algorithm Analysis

Dilip Sarkar

Department of Computer Science
University of Miami

Outline I

Trees are Special Type of Graphs

What is a tree?

Tree: A tree is a *connected* graph with n nodes/vertices and $(n-1)$ links/edges.

Labeled tree: A **labeled** tree has a label associated with every node.

Ordered tree: A tree with a node designated as the root node.

Path: A path from a vertex a vertex v_i to another vertex v_j is a unique sequence of edges.

Path length: The number of edges on the path from vertex v_i to v_j is the **length** of path.

Distance: *Path length* is also know as **distance** between two vertices.

Parent/child relationship: If two nodes v_p and v_c are directly connected by an edge and the distance of v_c from the root vertex is greater than v_p, then

v_p is the parent of v_c and
v_c is a child of v_p.

Note: A node have a unique parent, but a parent may have zero or more children.

Oriented tree: If the children of an ordered tree has *left/right* relationship, the tree is an oriented tree.

k-ary tree: In a k-ary tree the maximum number of children of one or more nodes is k.

Binary search trees I

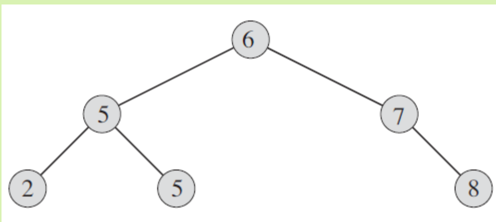


Figure: A binary search tree

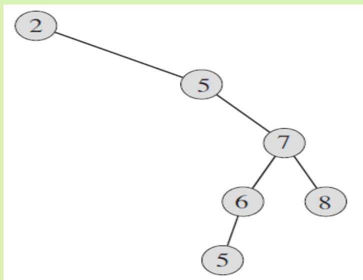


Figure: Another binary search tree

Binary search tree: An oriented **2-ary tree** is a *binary search tree* if label/key of the parent node v_p is

- greater than that of its left child v_{lc} AND
- smaller than that of its right child v_{rc}.

With three keys (for examaple: 1, 2, and 3) how many binary search trees are possible?

Operations on search trees.

Tree-Search(x, k)

Tree-Minimum(x, k)

Tree-Maximum (x, k)

Tree-Successor(x, k)

Tree-Insert(T, z)

Tree-Delete(T, z)

Binary search trees: Dynamic operations

```
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)
```

Figure: Recursive binary tree search algorithm

```
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
```

Figure: Iterative binary tree search algorithm

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

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

Figure: Algorithms to find minimum and maximum keys

```
TREE-SUCCESSOR(x)
if x.right ≠ NIL
    return TREE-MINIMUM(x.right)
y = x.p
while y ≠ NIL and x == y.right
    x = y
    y = y.p
return y
```

Figure: Algorithm to find successor of a key

Theorem 12.2: We can implement the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR so that each one runs in $O(h)$ time on a binary search tree of height h.

Binary search trees: INSERT operation

```
TREE-INSERT(T, z)
1   y = NIL
2   x = T.root
3   while x ≠ NIL
4       y = x
5       if z.key < x.key
6           x = x.left
7       else x = x.right
8   z.p = y
9   if y == NIL
10      T.root = z  // tree T was empty
11  elseif z.key < y.key
12      y.left = z
13  else y.right = z
```

Figure: Algorithm for inserting a new element

- Lines 3 to 8 search for the key z, which is **NOT** in the tree..

- The search ends at a node that has one child or none.
- The new element is to be *attached* to the node where the search ended.

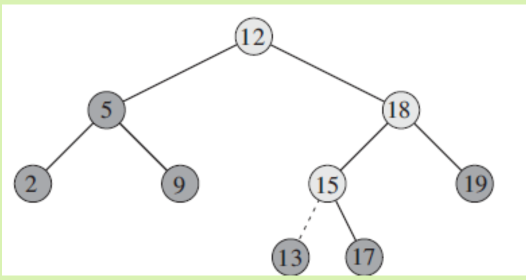


Figure: Example of an insert operation on a binary search tree .

- The key of the element to be inserted is 13.
- Note that 13 is not in the tree and search ends at node with key 15.

Binary search trees: DELETE operation I

- Find the node to be deleted
  - **The node z has no child**. Delete it by removing pointer to it.
  - **The node z has one child**. Elevate the child to the position of z.

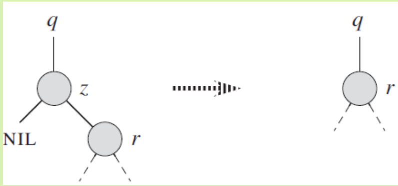


Figure: The node containing z to be deleted and it has no child.

Figure: The node z has only left-child.

- **The node z has two children**.

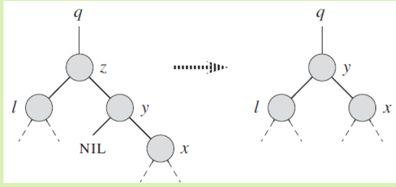


Figure: The node containing z has both children but right child has only right-child.

- Replace NIL pointer with left-pointer of z and replace z with the right child.
- A symmetric case is possible, where left child has only a left-child. (Figure is not shown.)

Binary search trees: DELETE operation II

- The node z has both hildren and both child has two children.

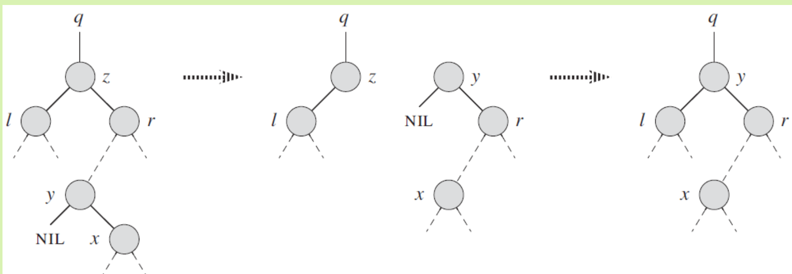


Figure: The node z has both child and both of them has two children.

- For DELETETE operation requires moving subtrees around within the binary search tree.
- Which is transplanting subtrees from one location to another.
- Let us see, an algorithm to replace subtree rooted at node u with the subtree rooted at node v

```
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
```

Binary search trees: DELETE operation III

```
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.p ≠ z
        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
```

Figure: Algorithm for delete operation.

Theorem 12.3 We can implement the dynamic-set operations INSERT and DELETE so that each one runs in $O(h)$ time on a binary search tree of height h.

Number of different binary trees with n Keys

Let b_n denote the number of different binary trees with n nodes.

It can be shown that $b_0 = 1$, and

$$b_n = \sum_{n=0}^{n-1} b_k b_{n-1-k}$$

Because consider a sequnece of length n; we use the first k keys to build the left subtrees, use the $(k+1)$th key as the root and the remaining $n-1-k$ keys to build right subtrees.

We will get, $b_n = b_k b_{n-1-k}$ binary trees.

When we consider all possible values for k, we get

$$b_n = \sum_{n=0}^{n-1} b_k b_{n-1-k}$$

Solution to the equation above is

$$b_n = \frac{1}{n+1}\binom{2n}{n}.$$

However average height of a node in a randomly built tree $O(\lg n)$ and this is the reason for expected running time for Quicksort is $O(n \lg n)$.

Properties of red-black trees

- Dynamic operations on a binary-seardh tree, in the worst case, is $O(h)$ time.
- This would be good if depth $h = O(\lg n)$ for a tree with n nodes.
- This means the tree was height-balanced.
- A binary (search) tree is height-balanced, if for all nodes *nd* in the (search) tree $|h_l(nd) - h_r(nd)| \leq c$, for a given constant c.
- AVL-trees and red-black trees are height-balanced binary search trees.
- In this section we study **red-black trees**

- A red-black tree is a binary-search tree
- Each node contains the attributes *color, key, left, right* and *p*

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 children are black
5. For each node, all simple paths from the node to the descendants leaves contain the same number of **black** nodes.

A red-black tree

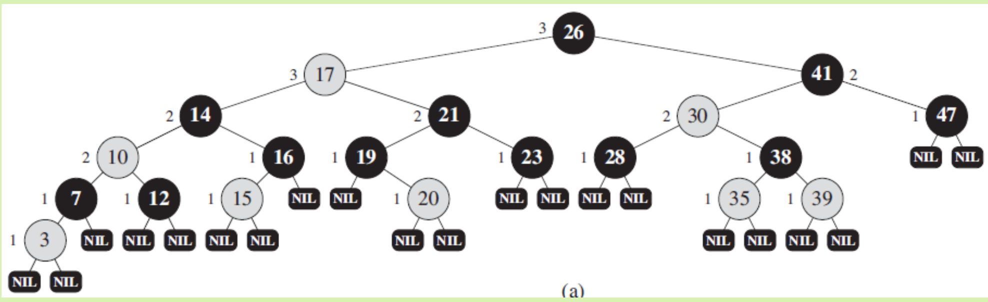


Figure: A red-black tree

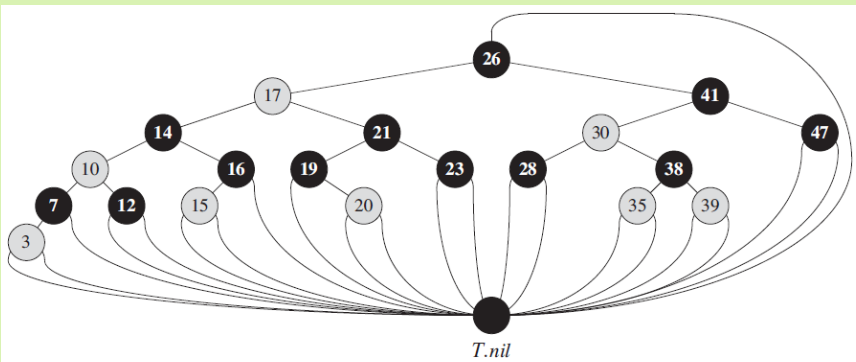


Figure: Each NIL replaced by the single sentient $T.nil$

A red-black tree

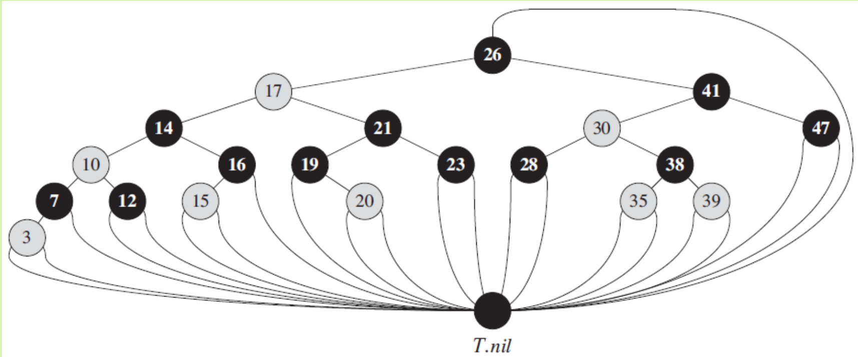


Figure: Each NIL replaced by the single sentient $T.nil$

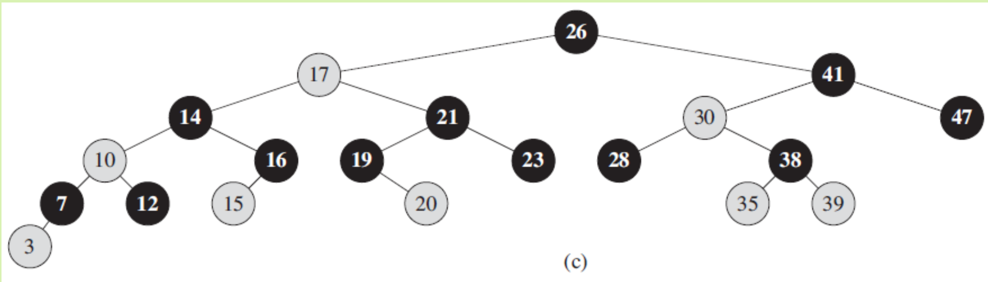


Figure: Same red-black tree with leaves and root's parent omitted

Height of a red-black tree with n nodes

Lemma (13.1)

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

Proof.

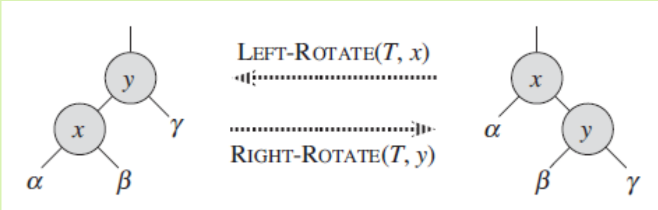
Let $bh(x)$ denote the height of the node x of a red-black tree.

The proof is by induction. It is shown that for a subtree rooted at x has at least $(2^{bh(x)} - 1)$ internal nodes. That is, $n \geq (2^{bh(x)} - 1)$

For a leaf, $T.nil$, we have at least $(2^{bh(x)-1} = 2^0 - 1 = 0$. Now it is easy to show that if the statement is true for x, then it is true for the node's parent.

$\square$

Rotation Operations



LEFT-ROTATE(T, x)

```
y = x.right                  // set y
x.right = y.left             // turn y's left subtree into x's right subtree
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p                    // link x's parent to y
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                   // put x on y's left
x.p = y
```

Figure: Left-Rotation Algorithm

An example of LEFT-ROTATE(T, x)

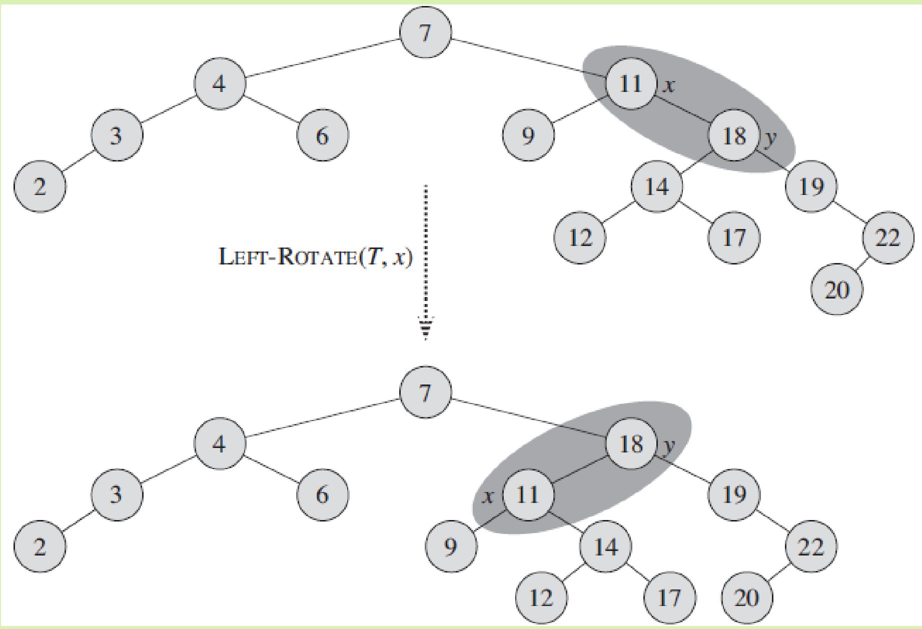


Figure: An example of LEFT-ROTATE(T, x)

Red-Black tree Insert Operation I

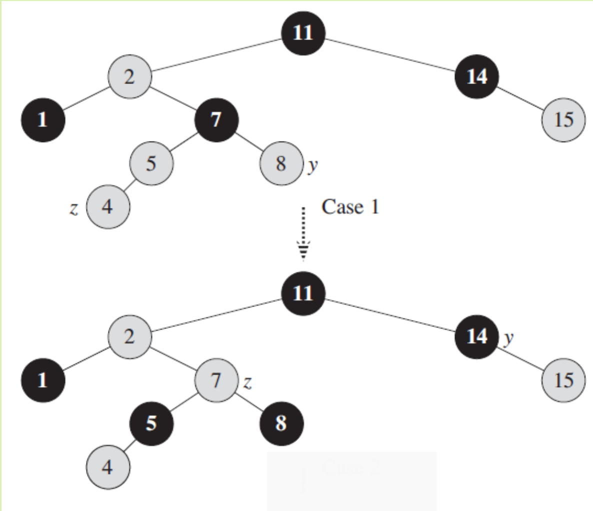


Figure: Red-black tree insert operation: Case 1

Because z and its parent $z.p$ are red, violation of property 4.
Since z's uncle y is red, Case 1 applies
Recolor nodes and move pointer z up the tree.
The resulting tree is shown below.

Red-Black tree Insert Operation II

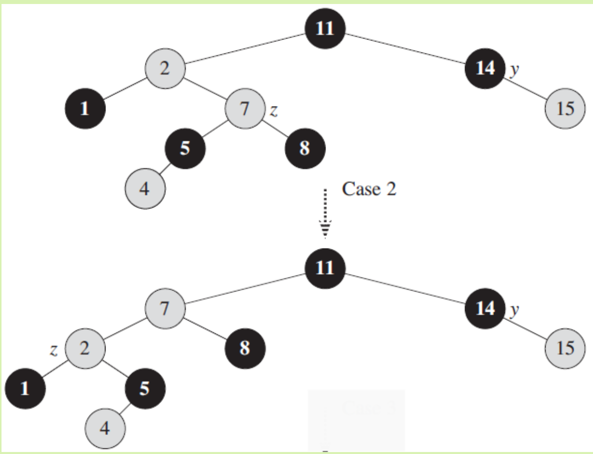


Figure: Red-black tree insert operation: Case 2

Because z and its parent $z.p$ are red, violation of property 4.
Since z's uncle y is black, Case 2 applies
Since z is the right the right child of $z.p$, perform a left rotation.
The resulting tree is shown below.

Red-Black tree Insert Operation III

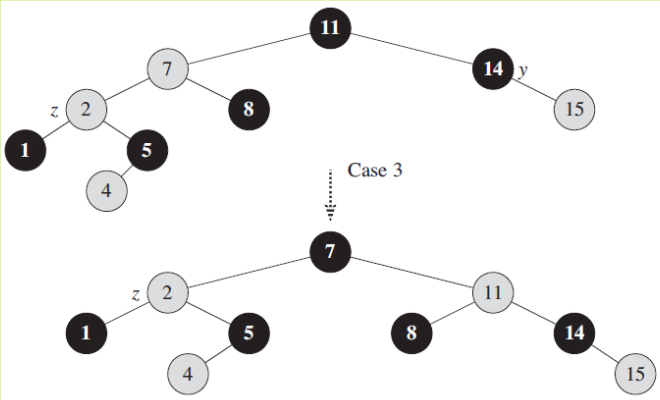


Figure: Red-black tree insert operation: Case 3

Because z is the left child of its parent parent, and Case 3 applies.

Since z's uncle y is black, Case 3 applies.

Recolor node 7 and 11

Right rotate to obtain the final red-black tree

Red-Black tree Insert Operation IV

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK            // case 1
            y.color = BLACK              // case 1
            z.p.p.color = RED            // case 1
            z = z.p.p                    // case 1
        else if z == z.p.right
                z = z.p                  // case 2
                LEFT-ROTATE(T, z)        // case 2
            z.p.color = BLACK            // case 3
            z.p.p.color = RED            // case 3
            RIGHT-ROTATE(T, z.p.p)       // case 3
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

Figure: RED-BLACK-INSERT-FIXUP ALGORITHM