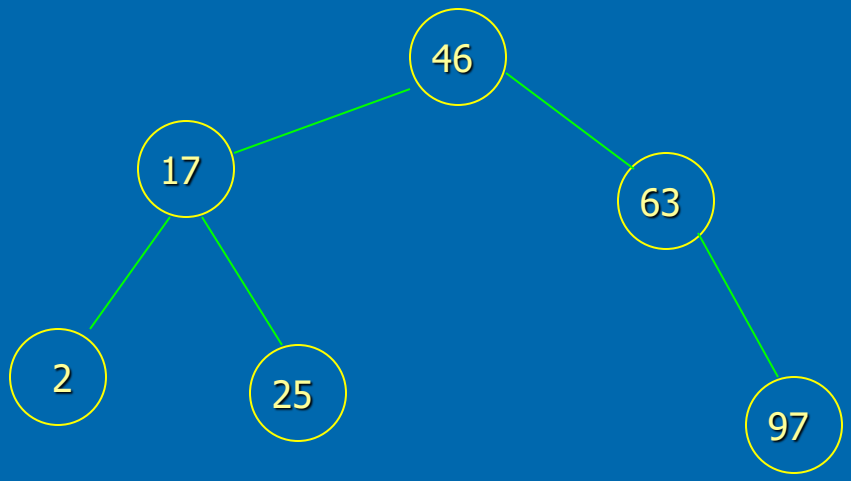
Lecture # 10

- Binary Search Trees
- Red-Black Trees

Binary Search Tree

- Consider the following Tree



- The value in each node is greater than the value in its left child and less than the value in its right child (if it exists). A binary tree having this property is called a binary search tree (BST).

- In-order Traversal is an algorithm which visits each node of a tree after its left sub tree and before its right sub tree. Therefore, in-order traversal shows the values stored in a binary search tree in sorted order.

- The in-order procedure is called recursively twice for each element (once for the left child and once for its right child), and the element is visited right between them.

  Therefore, the construction time is equal to $\Theta(n)$.

- The running time of the search operation is O(h), where h is the height of the tree. Therefore, if the tree is *balanced* and in its *minimum height* (i.e. if the tree is almost complete), then the worst-case run time, being directly proportional to the height of the tree, is log n.
- For example, for a binary search tree with n elements, with n = 1000, it needs about 10 comparisons for the search operation;
- However, if the binary search tree is unbalanced, the running time of a search operation is longer.

Insert

- And inserting characters in the order C, E, M, O, P, T, U will generate the following BST in worst unbalanced fashion.

- For trees that degenerate into linked lists as in the figure previously, search degenerates into linear search so that computing time in that case is ***O*(n)** for such BSTs.

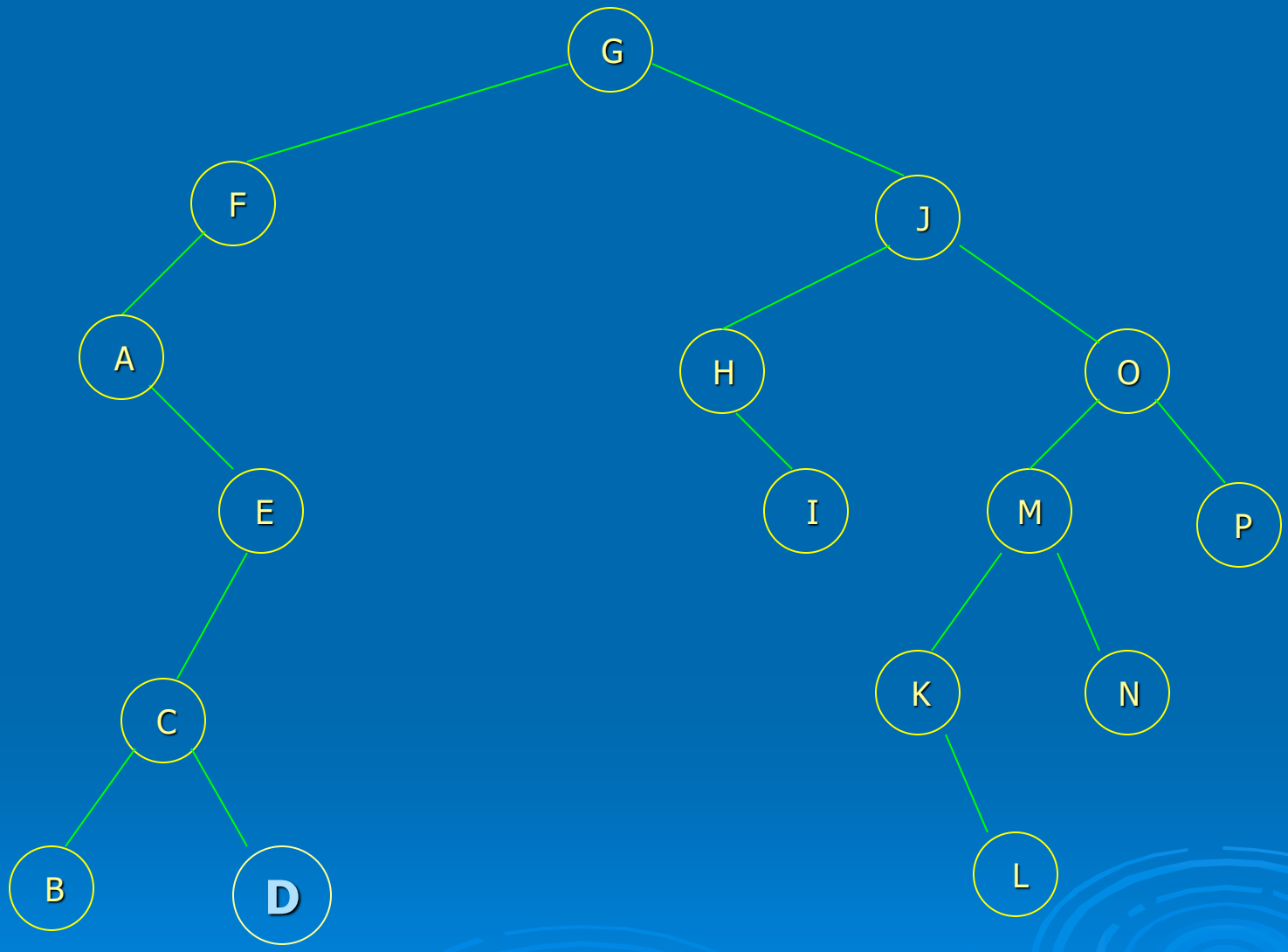
Deletion from a BST

- To delete a node X from BST we consider three cases

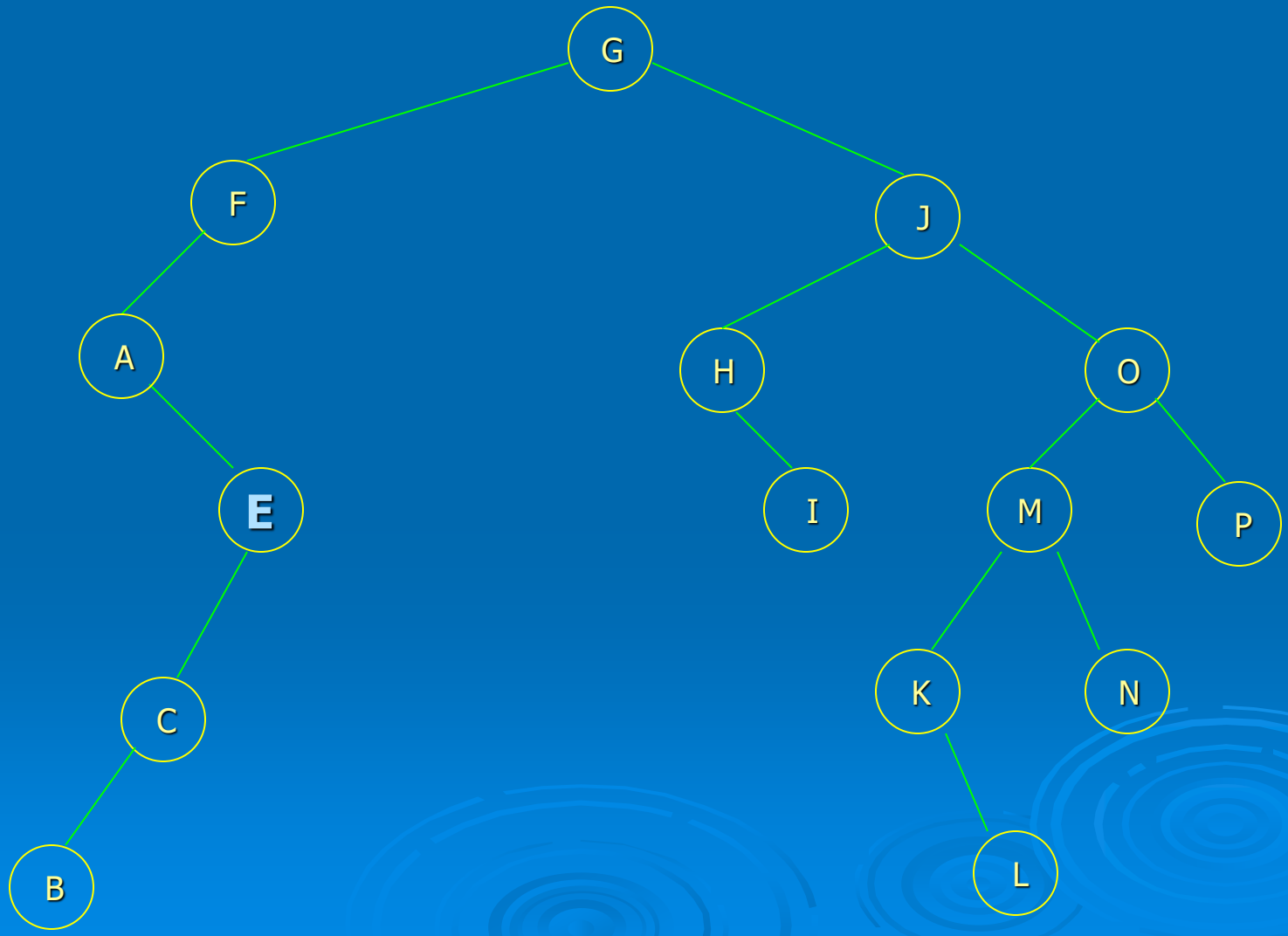
1. X is a leaf
2. X has only one child
3. X has two children

- The First case is very easy. We simply make the appropriate pointer in X's parent null pointer i.e. left or right pointer according to the shape or situation of the tree.

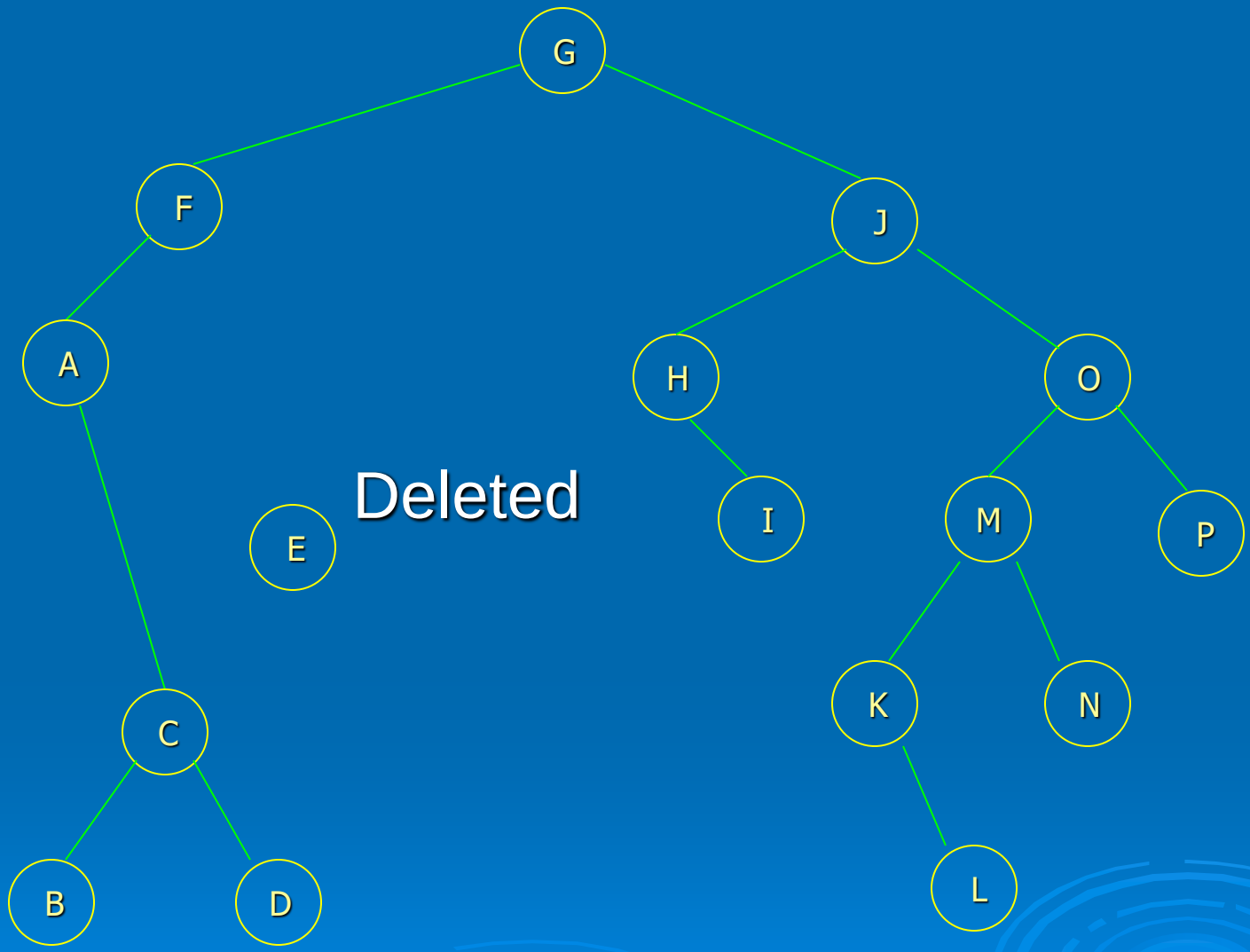
- Consider the Example……..


G
F
J
A
H
O
E
I
M
P
C
K
N
B
D
L

- For example to delete D in the above BST we can simply make the right pointer in its parent C a NULL.



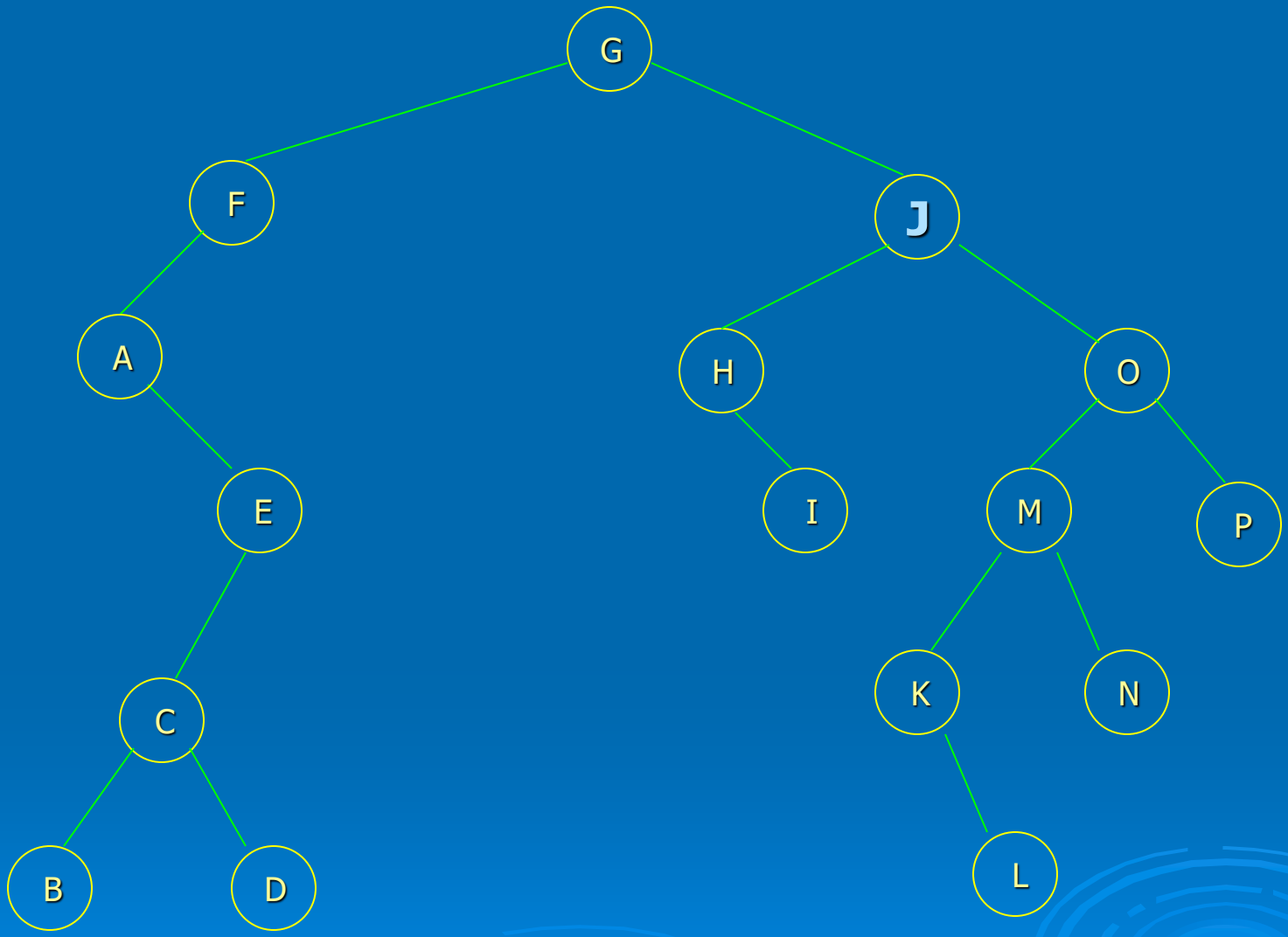
- The Second case, where the node X has exactly one child. Here we need to only set the appropriate pointer in X's parent to point to child. For example we can delete the node E in BST by simply setting the right pointer of its parent A to point to the node C and then delete X as follows.


G
F
J
A
H
O
Deleted
E
I
M
P
C
K
N
B
D
L

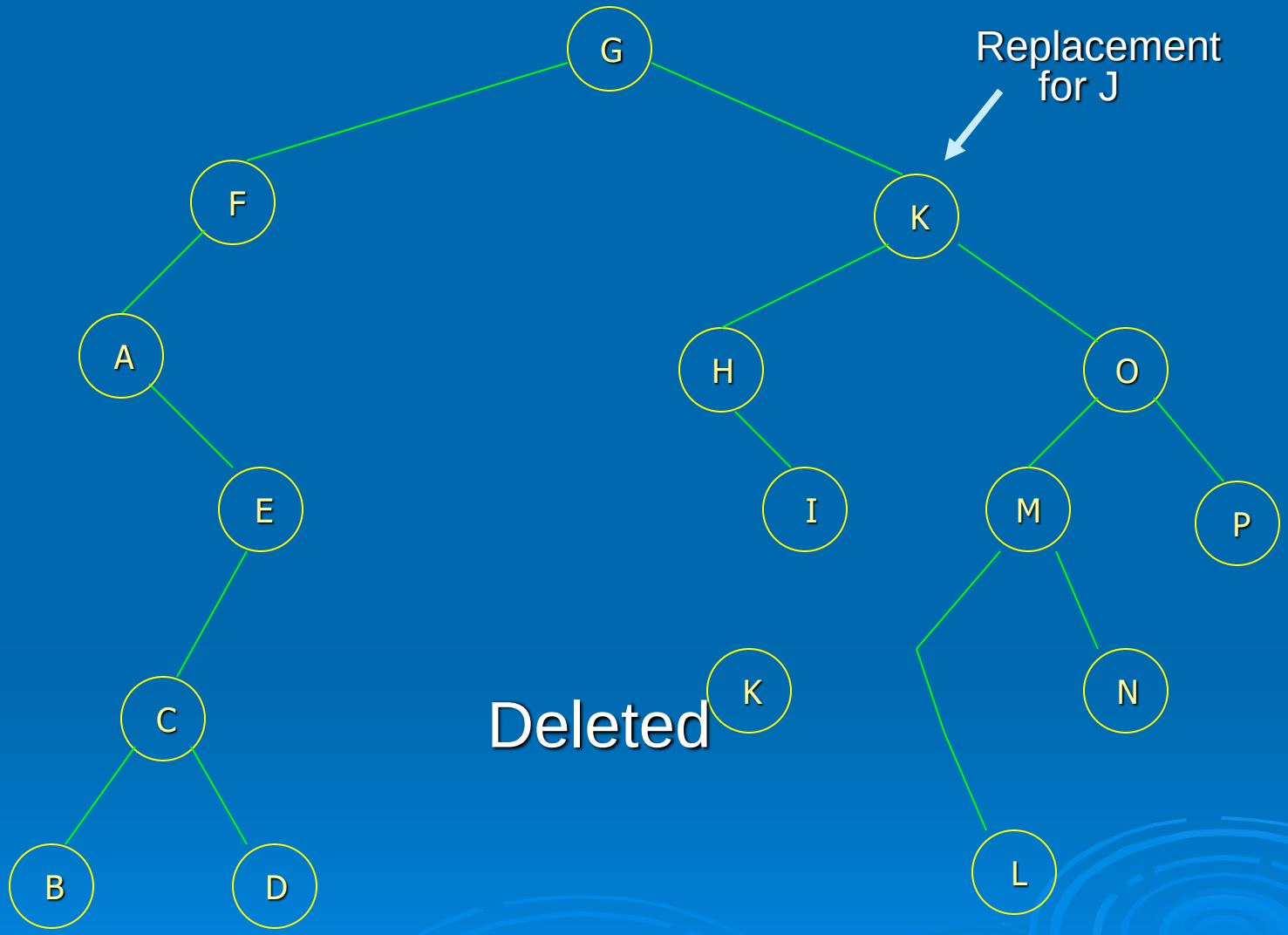
- In third case, in which X has 2 children we can replace the value stored in node X by its in-order successor or predecessor and then delete that successor or predecessor.

- To consider this we consider the following BST and we want to delete the node J.

- So we can replace it with its immediate in-order successor and we can locate it by starting from right child of J. and in our example the immediate in-order successor of J is K.


G
F
J
A
H
O
E
I
M
P
C
K
N
B
D
L

After deletion we have….



- The computing time for delete and insert is $O(h)$.

Red-Black Trees

- A Red-Black Tree is a Binary Search Tree with one extra bit of storage per node i.e. its color which cab be either red or black.

- Red-Black Trees are one of many search tree schemes that are "***balanced***" in order to guarantee the basic dynamic set operations take O(lg n) time in worst case.

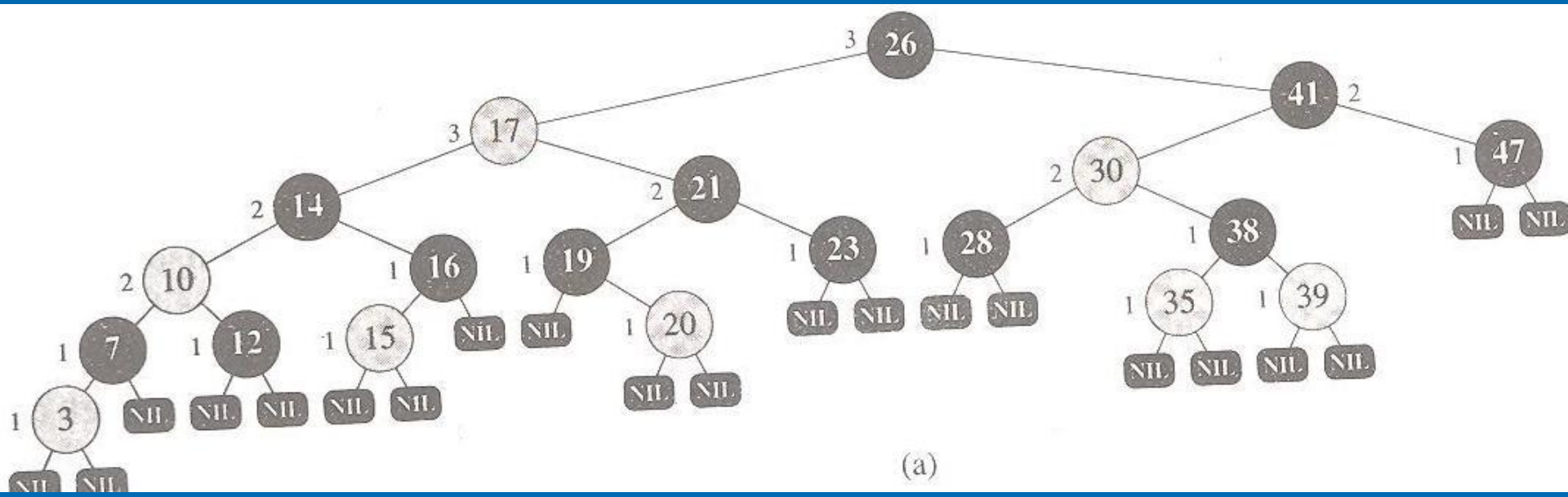
Red-Black Trees

- *Red-black trees*:
  - Binary search trees augmented with node color
  - Operations designed to guarantee that the height
    $h = O(\lg n)$
- First: we describe the properties of red-black trees
- Then: prove that these guarantee $h = O(\lg n)$
- Finally: describe operations on red-black trees

Red-Black Properties

- The *red-black properties*:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every "real" node has 2 children
  3. If a node is red, both children are black
     - Note: can't have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

Red-Black Tree



Red-Black Trees

- *Black-height: The number of* black nodes on any path from but not including a node x down to a leaf, denoted by **bh(x)** i.e. all descending paths from the node have the same number of black nodes.

- We define the *black-height* of a red-black tree to be the black-height of its root.

Height of Red-Black Trees

- *What is the minimum black-height of a node with height h?*
- Ans: a height-*h* node has black-height $\geq h/2$

- Root has a maximum height starting from leaf which has height 0. (leaf is at 0 level)
- Similarly leafs has height = 0 and has black-height = 0
- **Theorem:** A red-black tree with *n* internal nodes has height $h \leq 2\ lg(n + 1)$

  (internal nodes does not include leafs)

RB Trees: Proving Height Bound

- Prove: *n*- internal node RB tree has height

  $h \leq 2 \lg(n+1)$

- Claim: A sub tree rooted at a node *x* contains at least $2^{bh(x)} - 1$ internal nodes
  - Proof by *induction* on height *h*
  - Base step: *x* has height 0 (i.e., NULL leaf node)
    - So…subtree contains $2^{bh(x)} - 1$
      $= 2^0 - 1 = 1 - 1$
      = 0 internal nodes (TRUE)

RB Trees: Proving Height Bound

- Inductive proof that sub tree at node *x* contains at least $2^{bh(x)} - 1$ internal nodes
  - Inductive step: *x* has positive height and 2 children
    - Each child has black-height of **bh(*x*)** or **bh(*x*)-1** *(Why?)*
    - The height **(h)** of a child = (height of *x*) - 1
    - So the sub trees rooted at each child contain at least $2^{bh(x) - 1} - 1$ internal nodes
    - Thus subtree at *x* contains
      $(2^{bh(x) - 1} - 1) + (2^{bh(x) - 1} - 1) + 1$ *(+1 for node x itself)*
      $= 2 \cdot 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$ nodes

RB Trees: Proving Height Bound

- Thus at the root of the red-black tree:

  $n \geq 2^{bh(root)} - 1$ *(at least)*

  $n \geq 2^{h/2} - 1$ (a height-h node has black-height $\geq h/2$)

  $\lg(n+1) \geq h/2$ *(How?)*

  $h \leq 2 \lg(n + 1)$

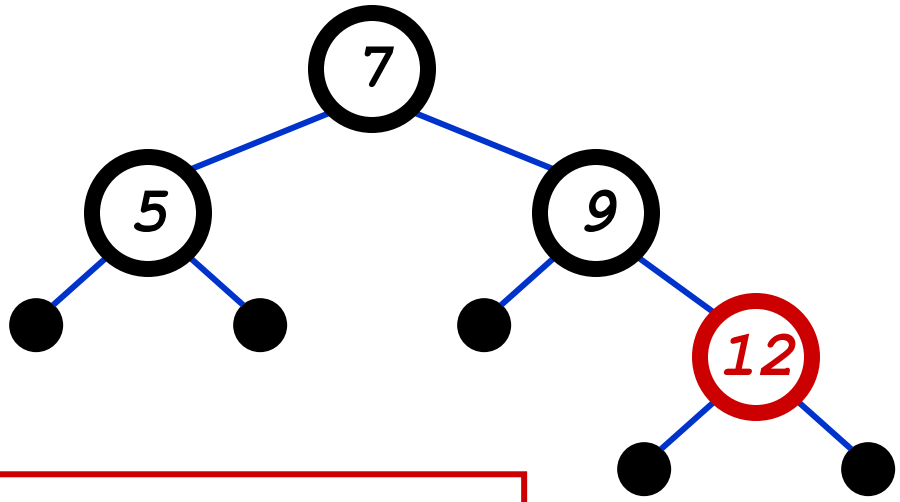
  Thus $h = O(\lg n)$

RB Trees: Worst-Case Time

- So we've proved that a red-black tree has $O(\lg n)$ height.
- Corollary: These operations take $O(\lg n)$ time:
  - Minimum(), Maximum()
  - Successor(), Predecessor()
  - Search()
- Insert() and Delete():
  - Will also take $O(\lg n)$ time
  - *But will need special care since they modify tree*

Red-Black Trees: An Example

- *Color this tree:*

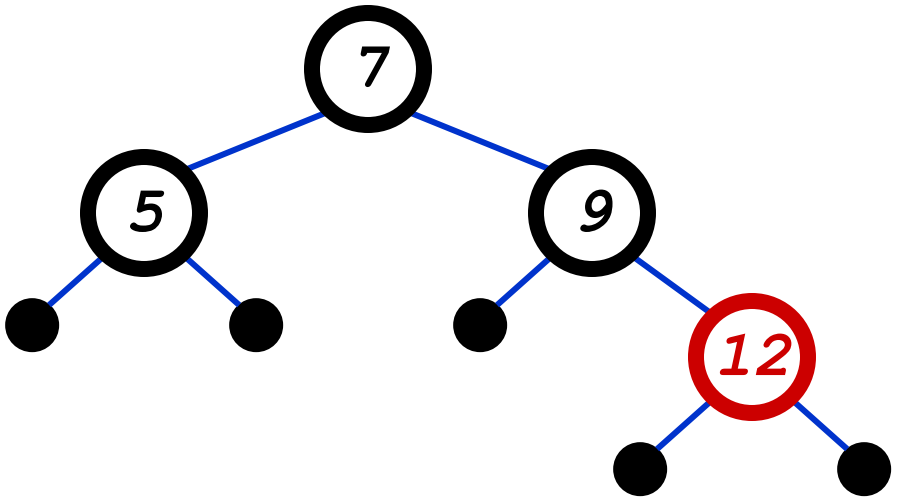


Red-black properties:

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

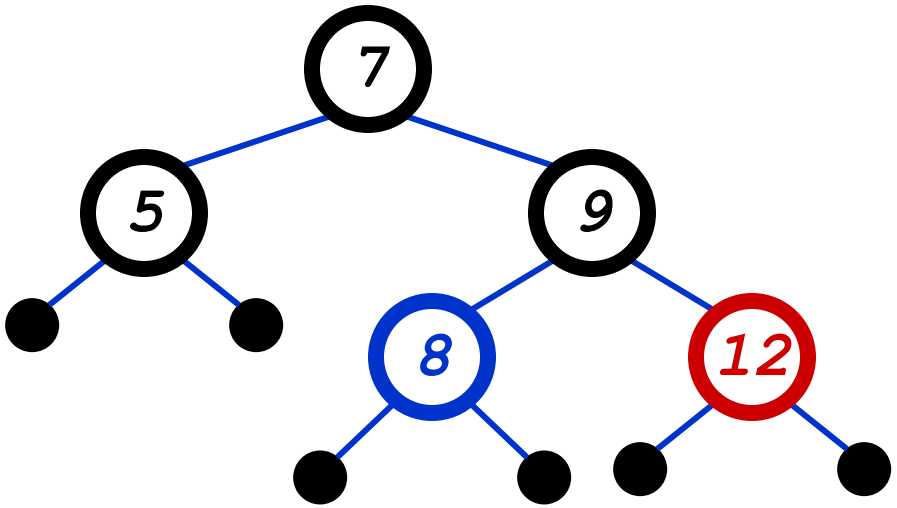
- Insert 8
  - *Where does it go?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

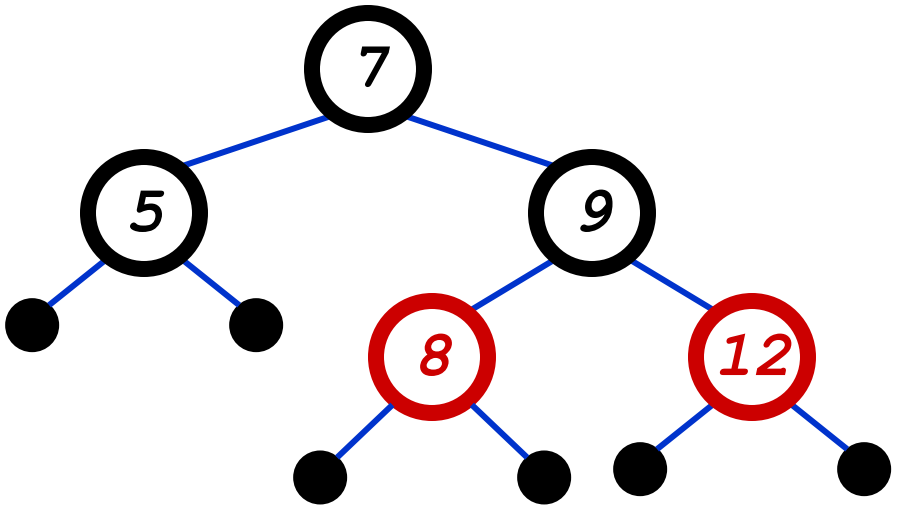
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

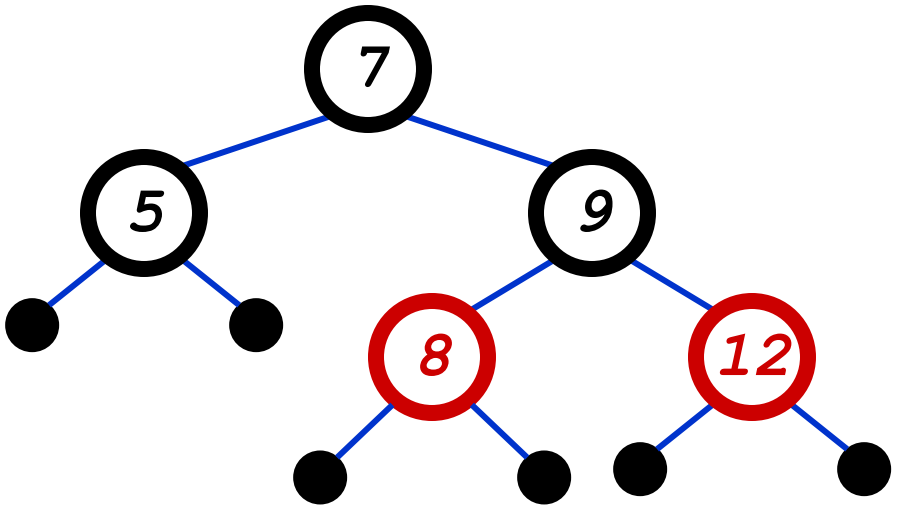
- Insert 8
  - *Where does it go?*
  - *What color should it be?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

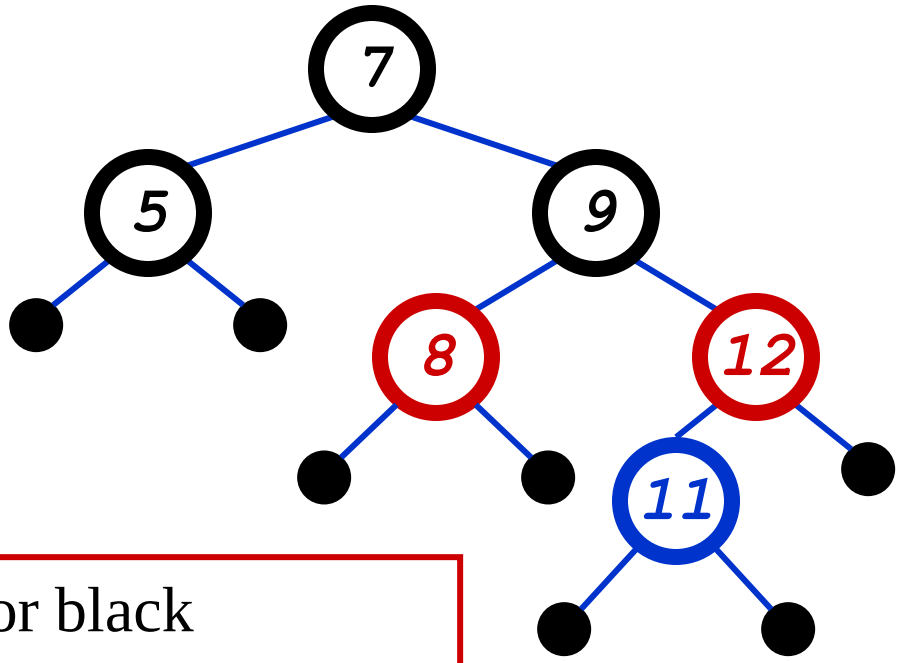
- Insert 11
  - *Where does it go?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

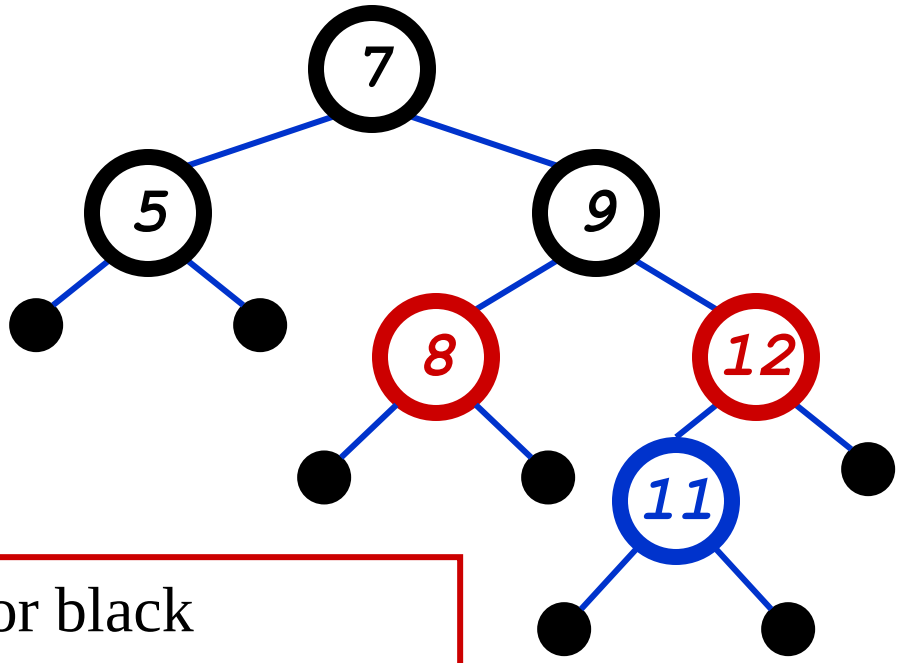
- Insert 11
  - *Where does it go?*
  - *What color?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

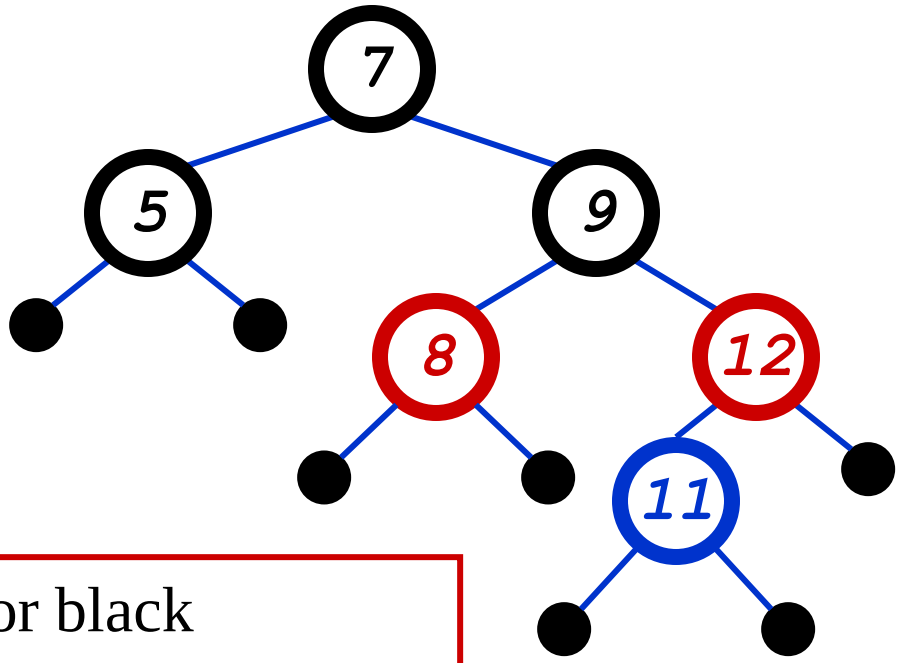
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

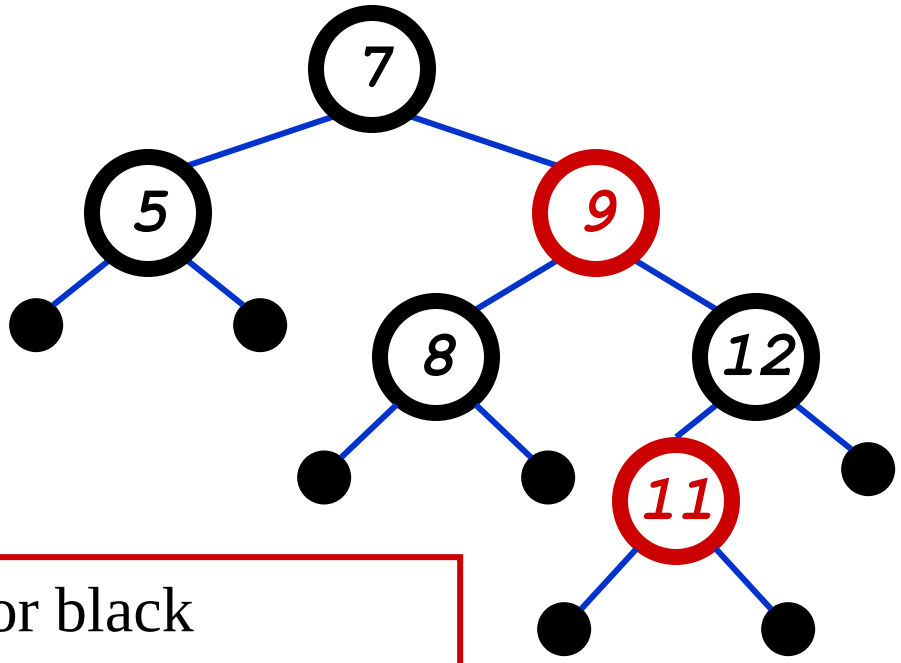
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Can't be red! (#3)
    - Can't be black! (#4)



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

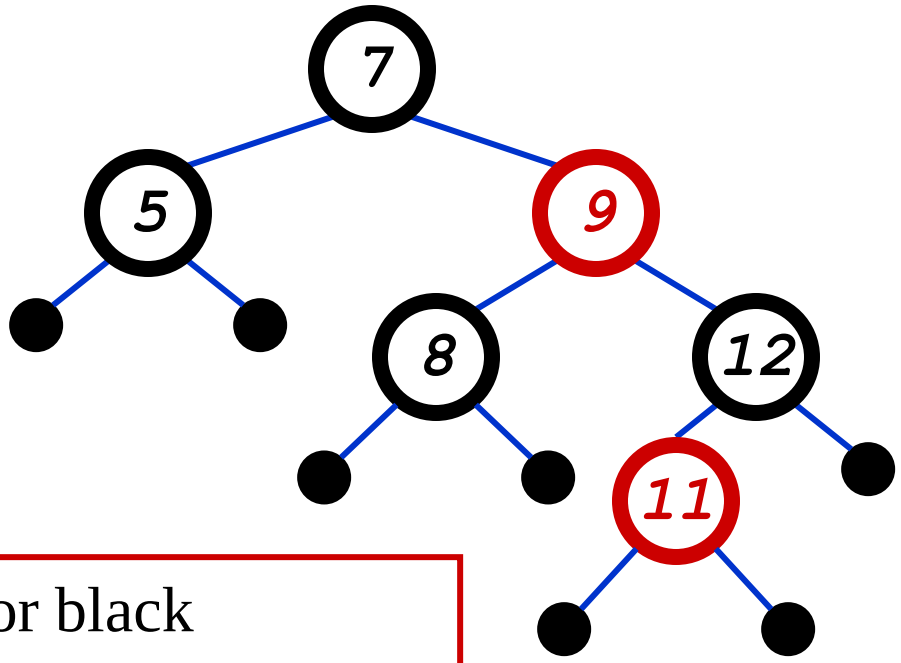
- Insert 11
  - *Where does it go?*
  - *What color?*
    - Solution: recolor the tree



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

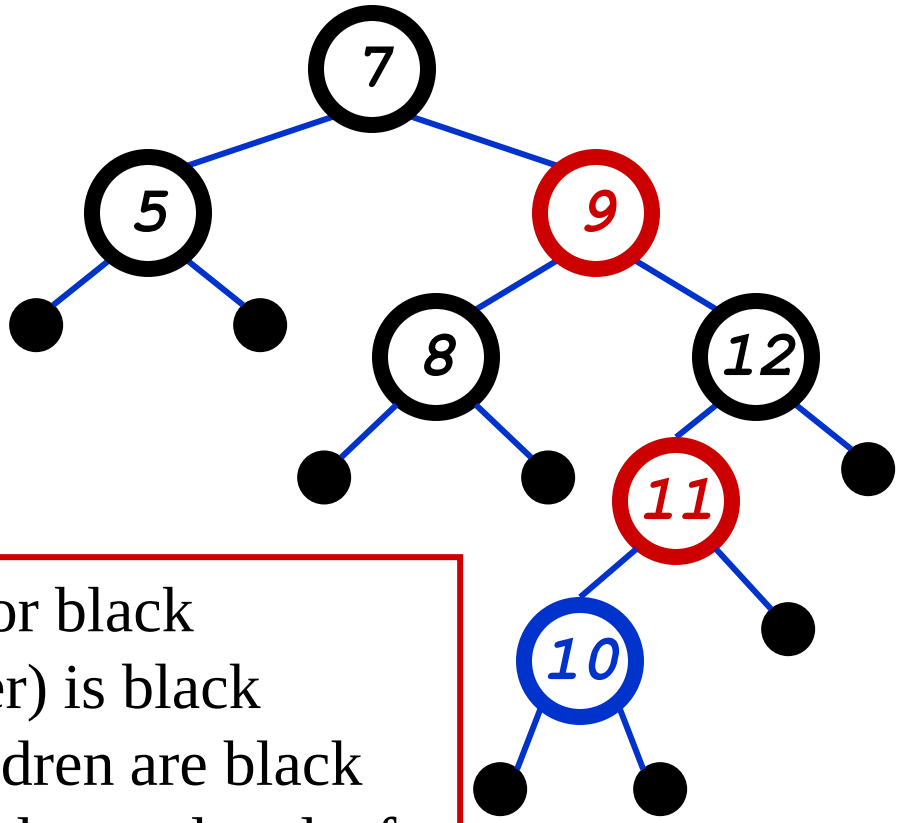
- Insert 10
  - *Where does it go?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

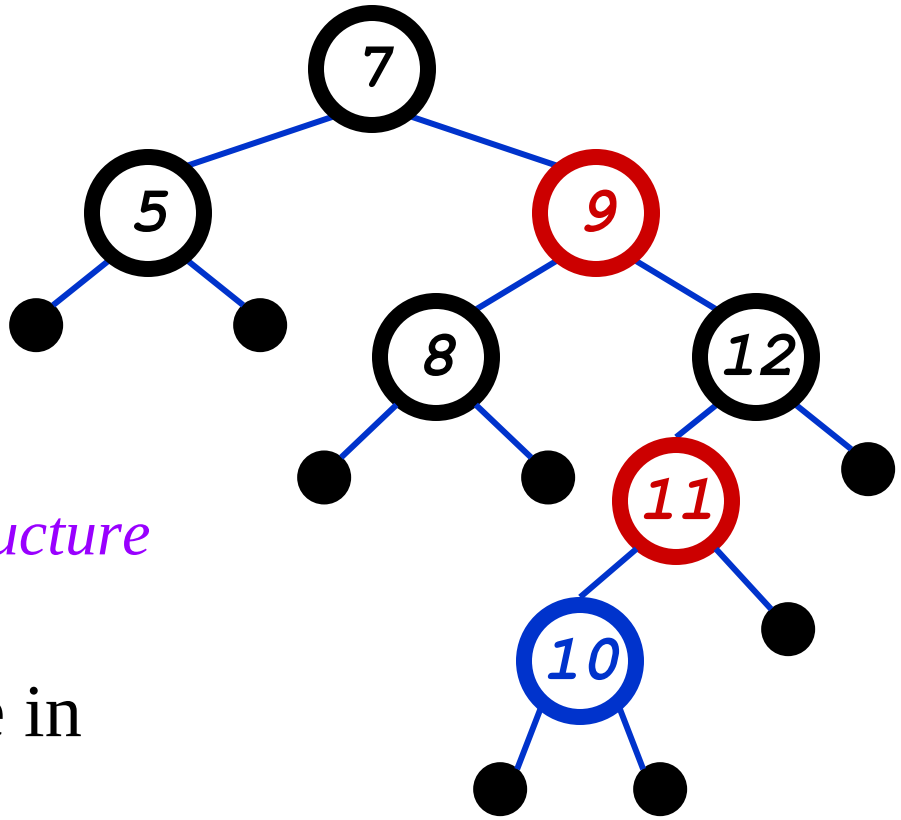
- Insert 10
  - *Where does it go?*
  - *What color?*



1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

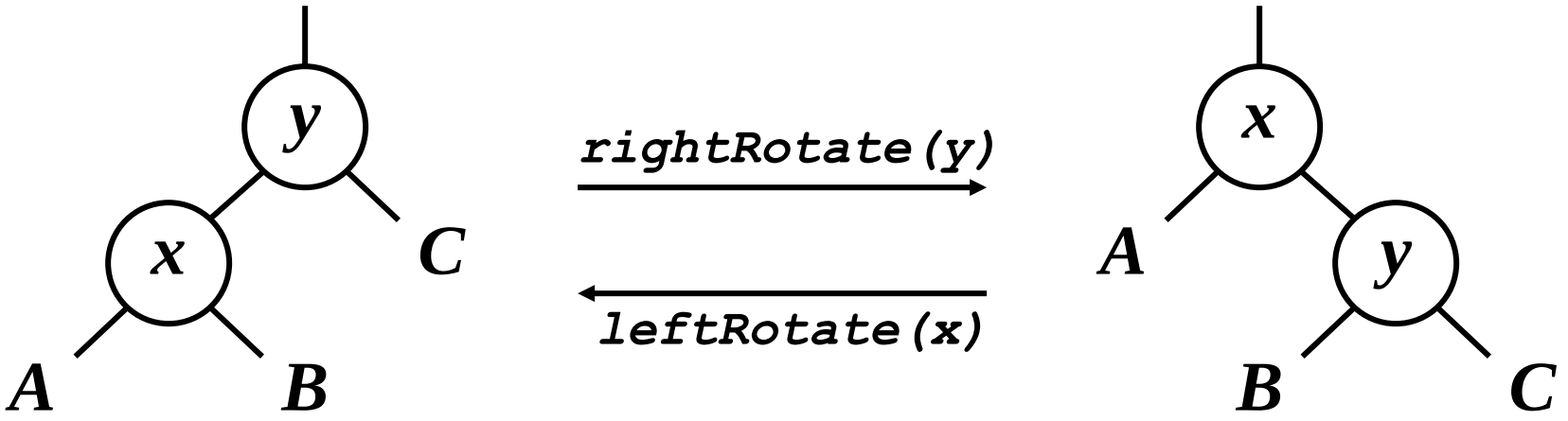
Red-Black Trees: The Problem With Insertion

- Insert 10
  - *Where does it go?*
  - *What color?*
    - A: no color! Tree is too imbalanced
    - Must *change tree structure to allow re-coloring*
  - Goal: restructure tree in O(lg *n*) time



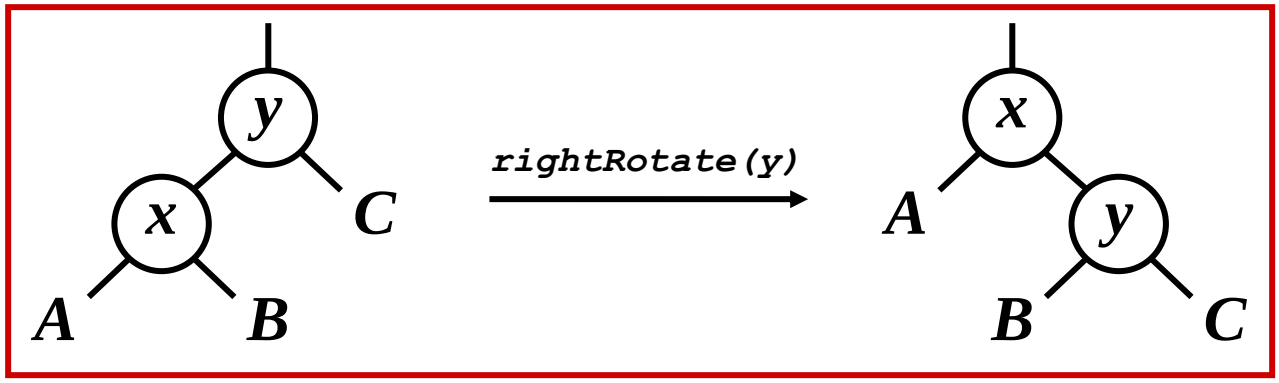
RB Trees: Rotation

- Our basic operation for changing tree structure is called *rotation*:



- *Does rotation preserve inorder key ordering?*
- *What would the code for* **`rightRotate()`** *actually do?*

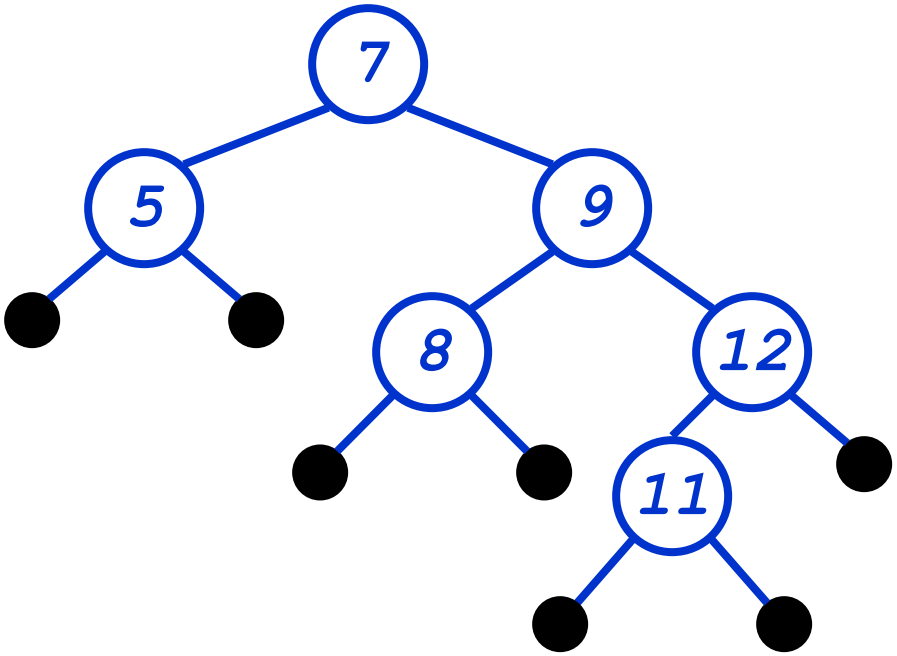
RB Trees: Rotation



- Answer: A lot of pointer manipulation
  - *x* keeps its left child
  - *y* keeps its right child
  - *x*'s right child becomes *y*'s left child
  - *x*'s and *y*'s parents change

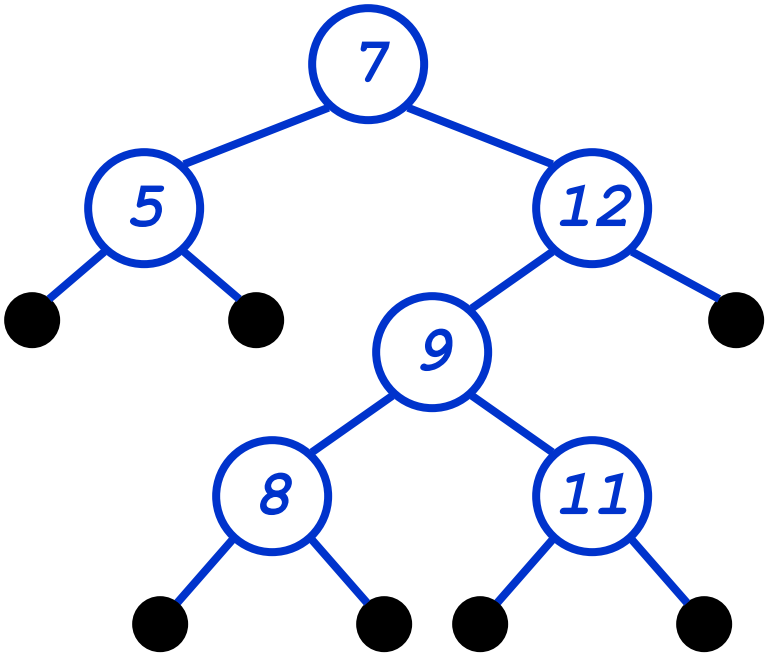
Rotation Example

- Rotate left about 9:



Rotation Example

- Rotate left about 9:



Red-Black Trees: Insertion

- Insertion: the basic idea
  - Insert *x* into tree, color *x* red.
  - Only r-b property 3 might be violated (if p[*x*] red)
    - If so, move violation up tree until a place is found where it can be fixed
  - Total time will be O(lg *n*)

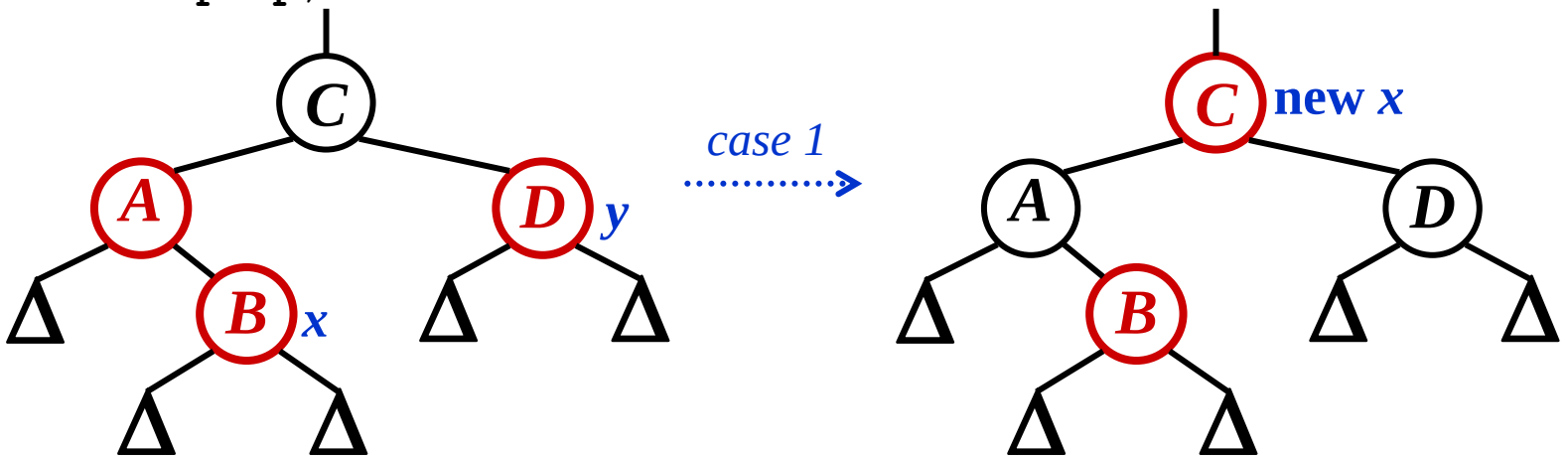
p[*x*] means parent of x

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

RB Insert: **Case 1**

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```

- Case 1: "uncle" is red
- In figures below, all Δ's are equal-black-height subtrees

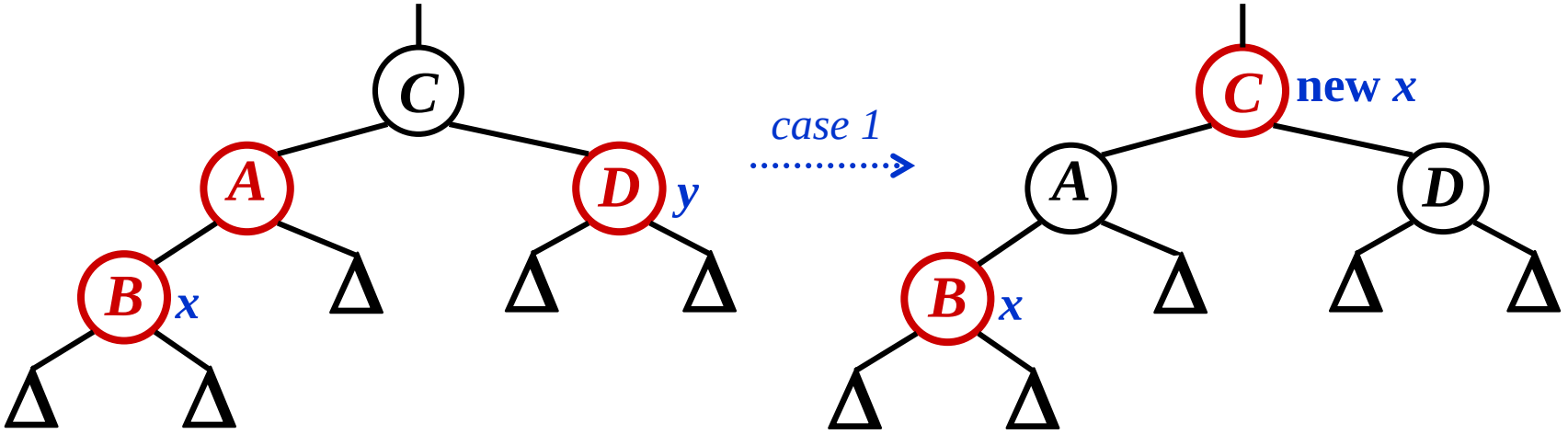


Change colors of some nodes, preserving #4: all downward paths have equal b.h.
x : represents the pointer to the node to be added.
p : represents the pointer to parent node
y : represents the pointer to the uncle node

RB Insert: Case 1

```
if (y->color == RED)
    x->p->color = BLACK;
    y->color = BLACK;
    x->p->p->color = RED;
    x = x->p->p;
```

- Case 1: "uncle" is red
- In figures below, all Δ's are equal-black-height sub trees

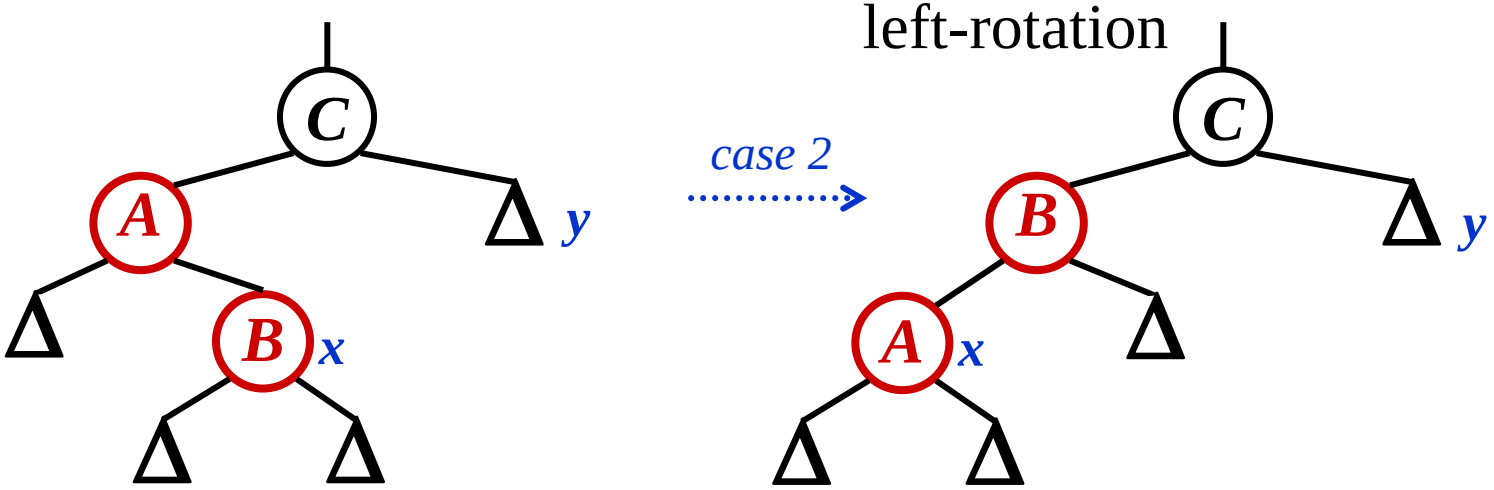


Same action whether x is a left or a right child

RB Insert: Case 2

```
if (x == x->p->right)
    x = x->p;
    leftRotate(x);
// continue with case 3 code
```

- Case 2:
  - "Uncle" is black
  - Node *x* is a right child
- Transform to case 3 via a left-rotation

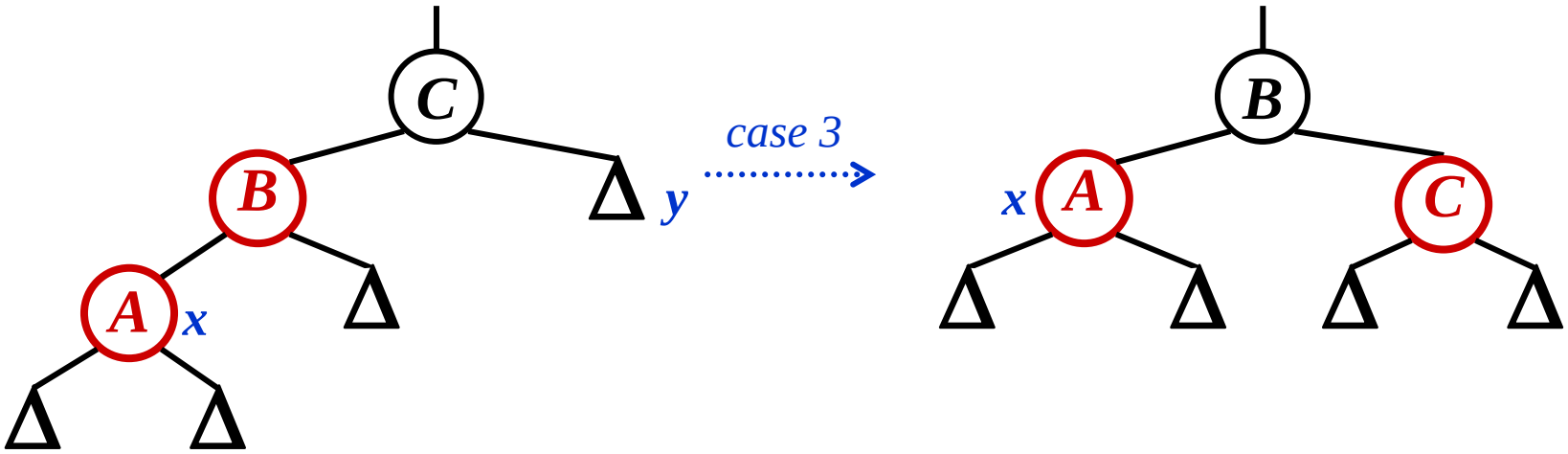


Transform case 2 into case 3 (x is left child) with a left rotation
This preserves property 4: all downward paths contain same number of black nodes

RB Insert: Case 3

```
x->p->color = BLACK;
x->p->p->color = RED;
rightRotate(x->p->p);
```

- Case 3:
  - "Uncle" is black
  - Node *x* is a left child
- Change colors; rotate right



Perform some color changes and do a right rotation
Again, preserves property 4: all downward paths contain same number of black nodes

RB Insert: Cases 4-6

- Cases 1-3 hold if *x*'s parent is a left child of its parent.

- If *x*'s parent is a right child of its parent, cases 4-6 are symmetric (swap left for right)

Red-Black Trees: **Deletion**

- We will not cover RB – Tree delete in class
  - You should think and design an algorithm working on same lines that we followed in insertion process

Thank You …