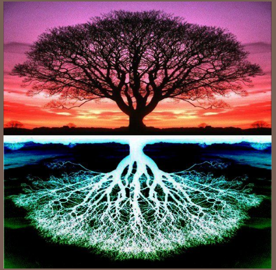
Object-oriented programming and data-structures



CORNELL UNIVERSITY FOUNDED A.D. 1865

CS/ENGRD 2110
SUMMER 2018

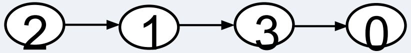
Lecture 9: Trees

http://courses.cs.cornell.edu/cs2110/2018su

Data Structures

- There are different ways of storing data, called data structures
- Each data structure has operations that it is good at and operations that it is bad at
- For any application, you want to choose a data structure that is good at the things you do often

Recall: ArrayList/LinkedList

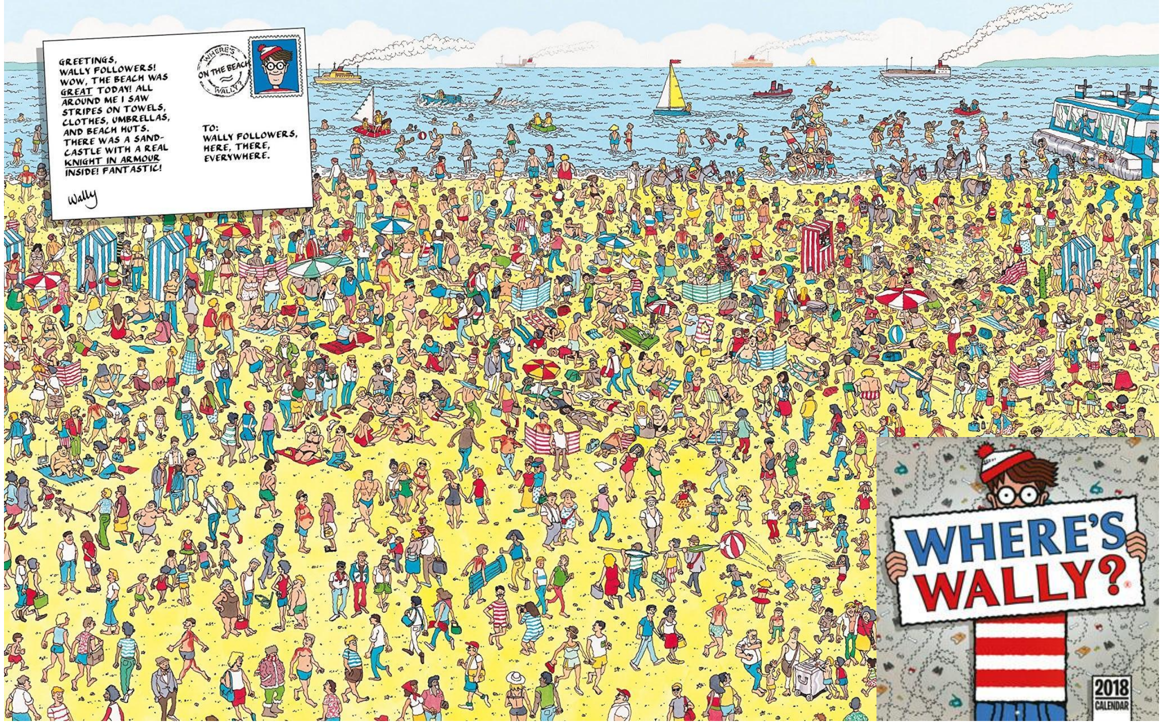
Data Structure	add(val x)	lookup(int i)
Array 2 1 3 0	$O(n)$	$O(1)$
Linked List 	$O(1)$	$O(n)$

The Problem of Search

Search is the problem of finding an element in a datastructure when you don't know where it is stored

ex: does this array contain element x?

Is Wally enrolled in this class?



GREETINGS,
WALLY FOLLOWERS!
WOW, THE BEACH WAS GREAT TODAY! ALL AROUND ME I SAW STRIPES ON TOWELS, CLOTHES, UMBRELLAS, AND BEACH HUTS. THERE WAS A SANDCASTLE WITH A REAL KNIGHT IN ARMOUR INSIDE! FANTASTIC!

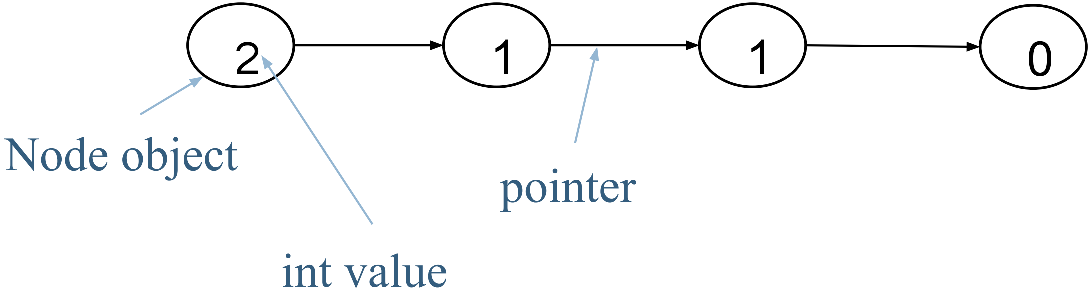
Wally

TO:
WALLY FOLLOWERS,
HERE, THERE,
EVERYWHERE.

Introducing Trees

We have already seen linked lists

But linked lists have O(n) complexity for searching elements

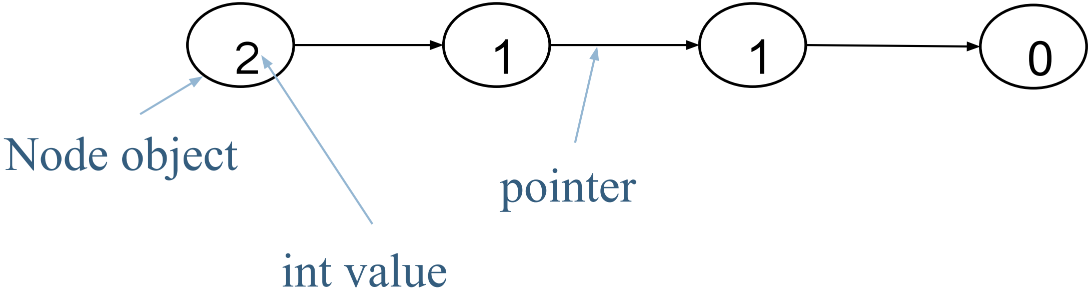


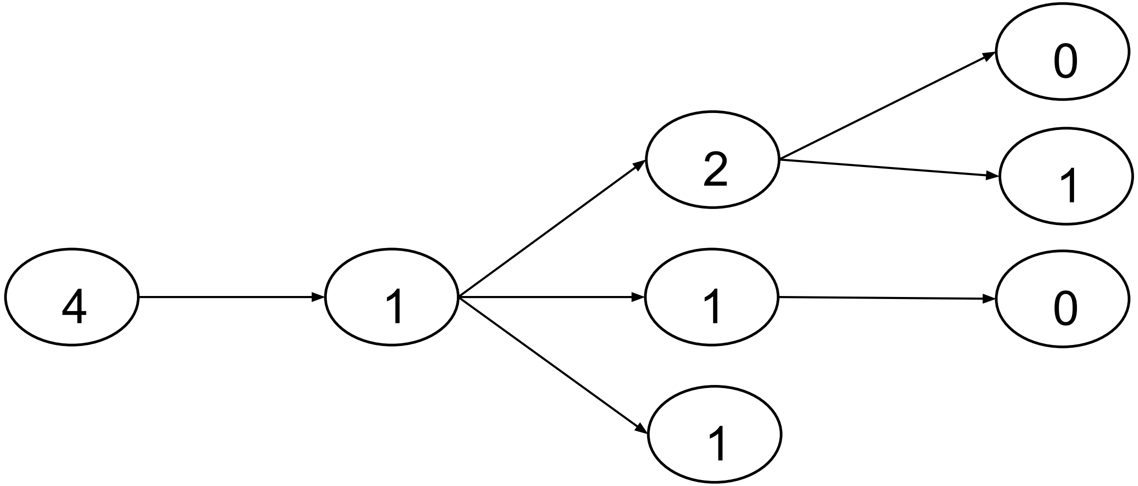
Introducing Trees

We have already seen linked lists

But linked lists have O(n) complexity for searching elements

Today, we look at **trees**. (Specific) trees have O(lg n) complexity for searching elements

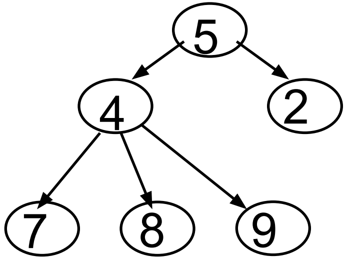




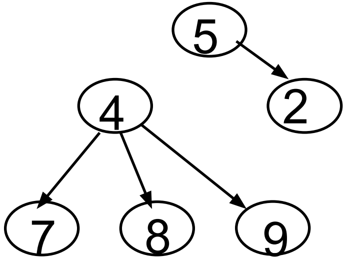
Botanic lesson: what is a tree?

Tree: data structure with nodes, similar to linked list

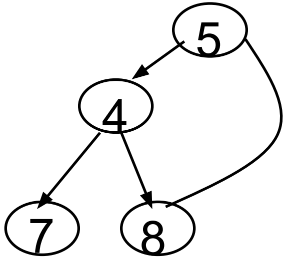
- Each node may have zero or more *successors* (children)
- Each node has exactly one *predecessor* (parent) except the *root*, which has none
- All nodes are reachable from *root*



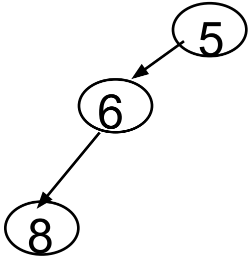
A tree



Not a tree

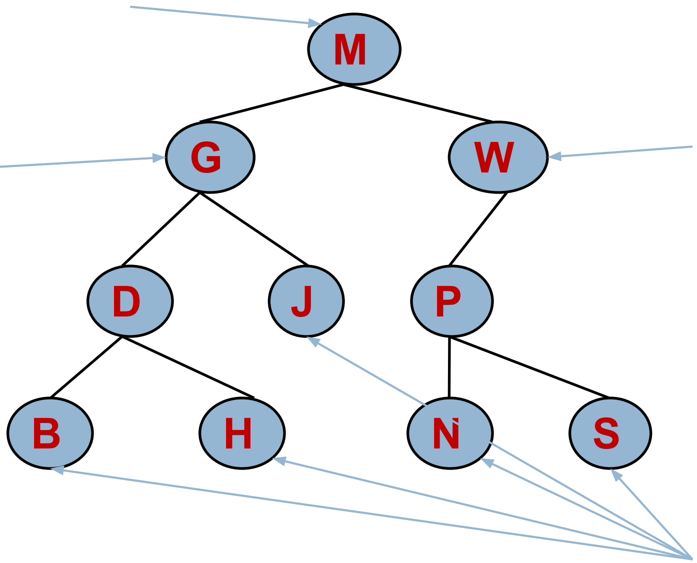


Not a tree



A tree

Tree Terminology



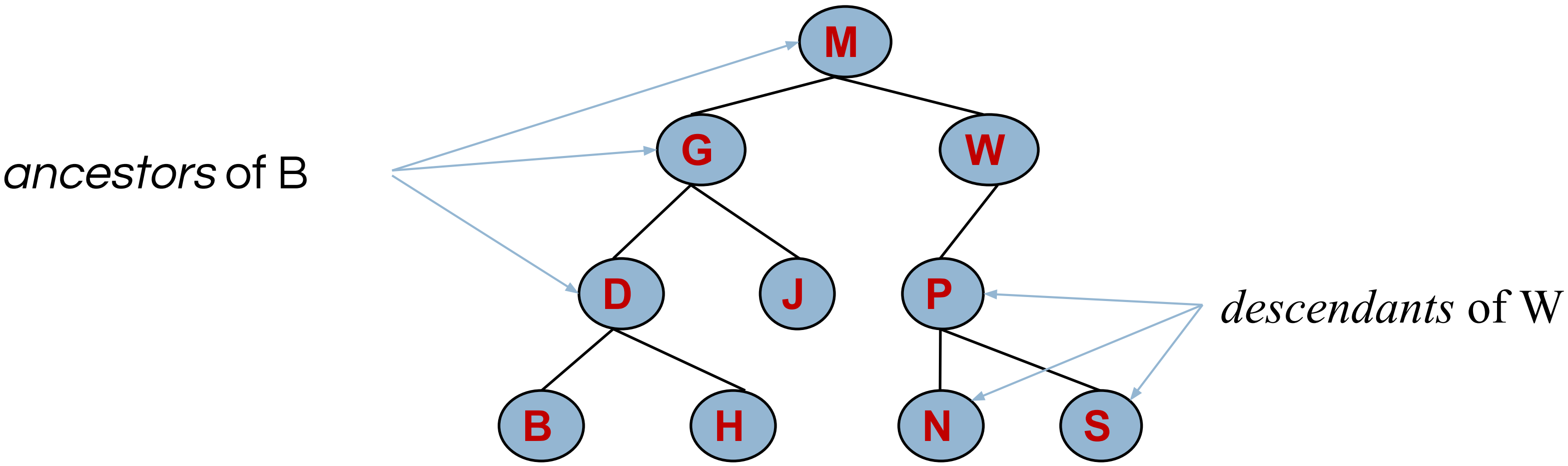
the *root* of the tree
(no parents)

child of M

child of M

the *leaves* of the tree
(no children)

Tree Terminology

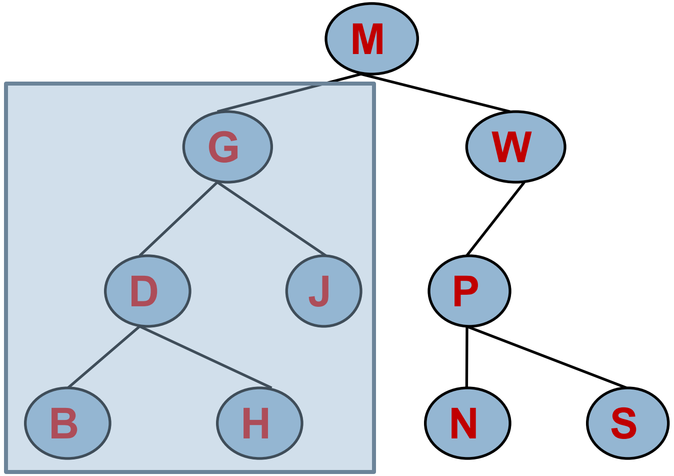


ancestors of B

descendants of W

Tree Terminology

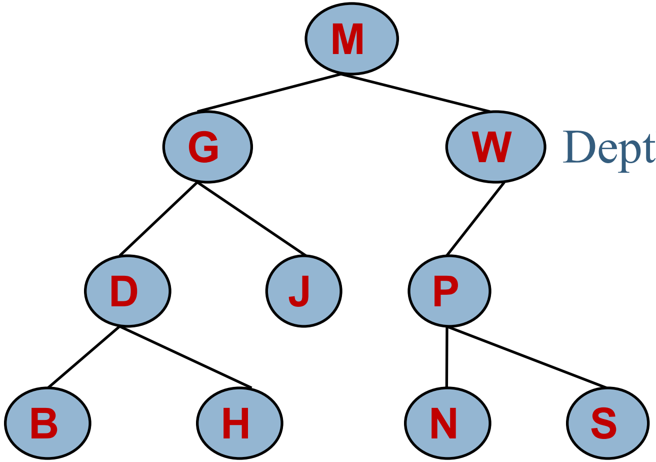
subtree of M



Tree Terminology

A node's *depth* is the length of the path to the root.

A tree's (or subtree's) *height* is the length of the longest path from the root to a leaf.

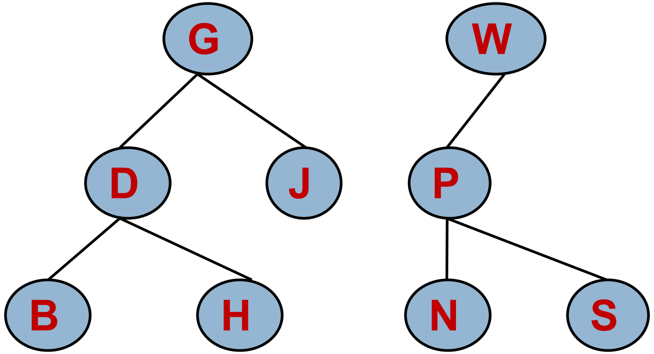


Depth 1, height 2.

Depth 3, height 0.

Tree Terminology

Multiple trees: a *forest*.

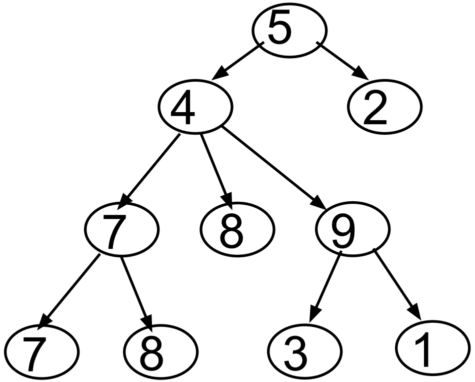


Class for general tree nodes

Class for general tree nodes

```
class GTreeNode<T> {
    private T value;
    private Set<GTreeNode<T>> children;
    //appropriate constructors, getters,
    //setters, etc.
}
```

Parent contains a list of its children

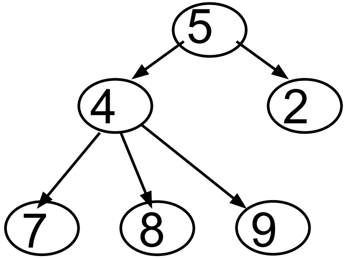


General tree

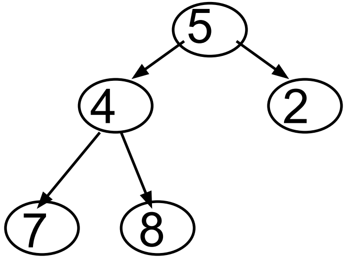
Binary Trees

A *binary tree* is a particularly important kind of tree where every node has at most two children.

In a binary tree, the two children are called the *left* and *right* children.



Not a binary tree
(a *general* tree)



Binary tree

Class for binary tree node

Class for binary tree node

```
class TreeNode<T> {
  private T value;
  private TreeNode<T> left, right;

  /** Constructor: one-node tree with datum x */
  public TreeNode (T v) { value= v; left= null; right= null;}

  /** Constr: Tree with root value x, left tree l, right tree r */
  public TreeNode (T v, TreeNode<T> l, TreeNode<T> r) {
     value= v; left= l; right= r;
  }
}
```

Either might be null if the subtree is empty.

Binary versus general tree

In a binary tree, each node has up to two pointers: to the left subtree and to the right subtree:

- One or both could be **null**, meaning the subtree is empty (remember, a tree is a set of nodes)
- Binary trees are used for searching

In a general tree, a node can have any number of child nodes (and they need not be ordered)

- Very useful in some situations ...
- ... one of which may be in an assignment!

Useful facts about binary trees

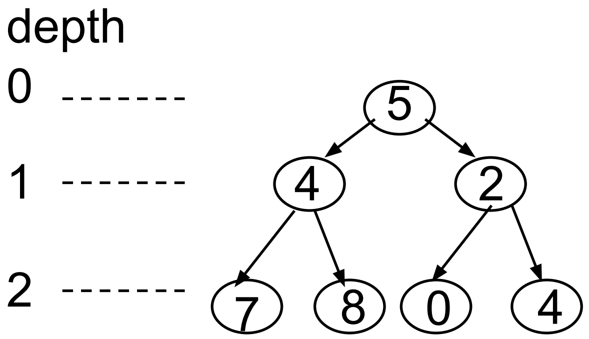
Max # of nodes at depth d: 2^d

If height of tree is h

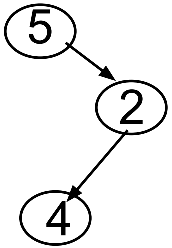
- min # of nodes: $h + 1$
- max #of nodes in tree:
- $2^0 + \ldots + 2^h = 2^{h+1} - 1$

Complete binary tree

- All levels of tree down to a certain depth are completely filled



Height 2,
maximum number of nodes



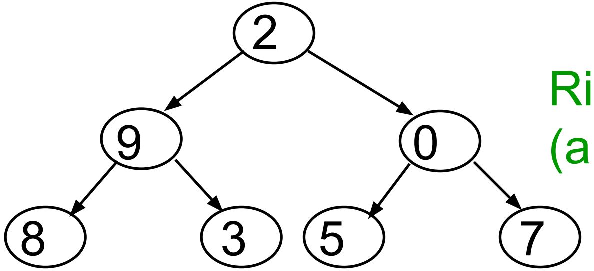
Height 2,
minimum number of nodes

A Tree is a Recursive Concept

A binary tree is either null or an object consisting of a value, a left binary tree, and a right binary tree.

A Tree is a Recursive Concept

A binary tree is either null or an object consisting of a value, a left binary tree, and a right binary tree.

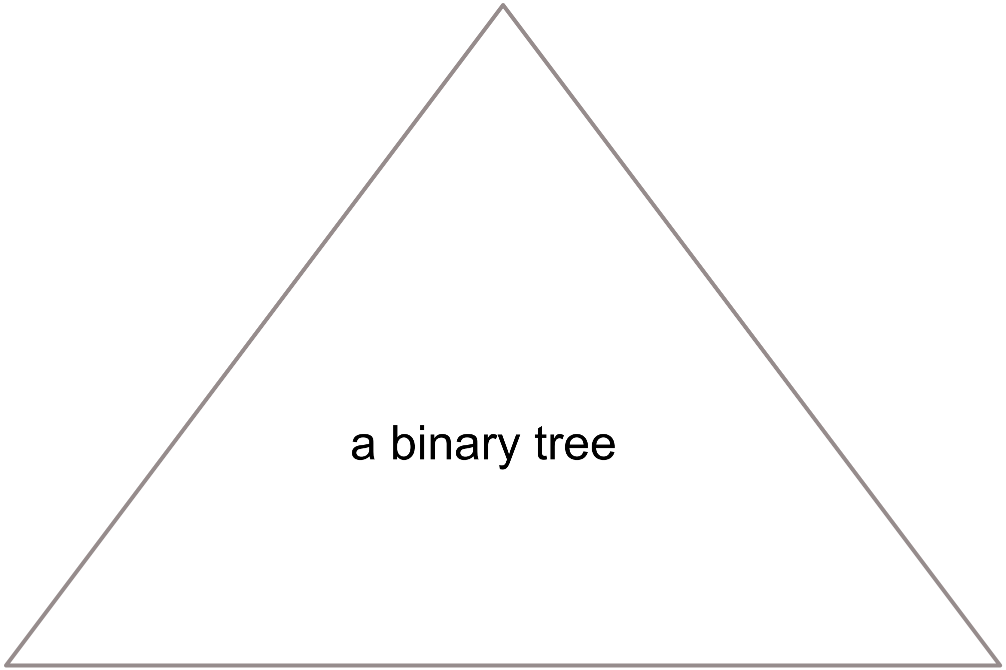


Binary tree

Right subtree
(also a binary tree)

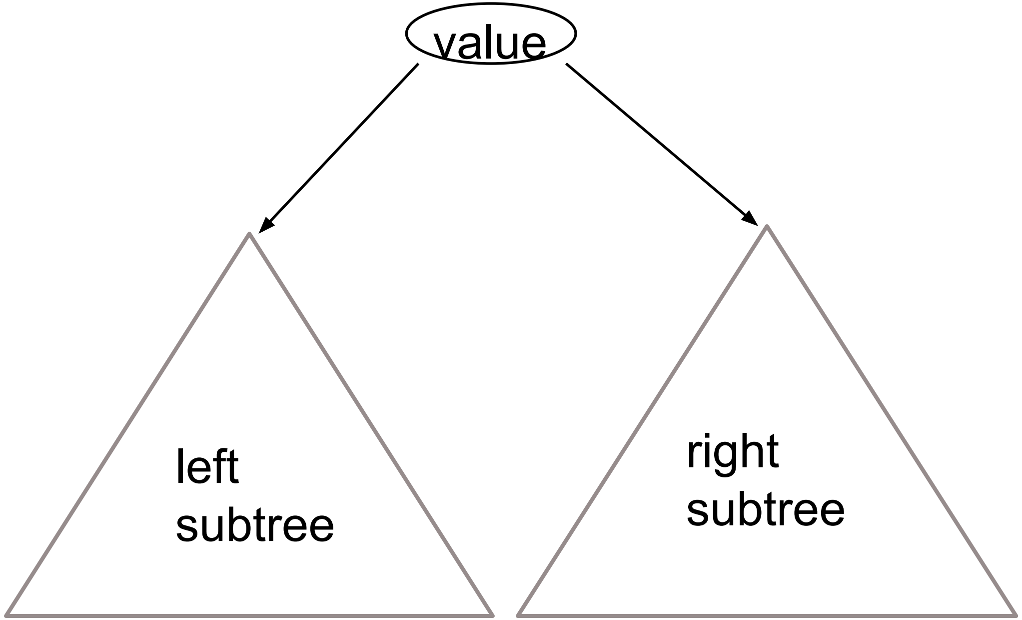
Left subtree,
which is a binary tree too

Looking at trees recursively

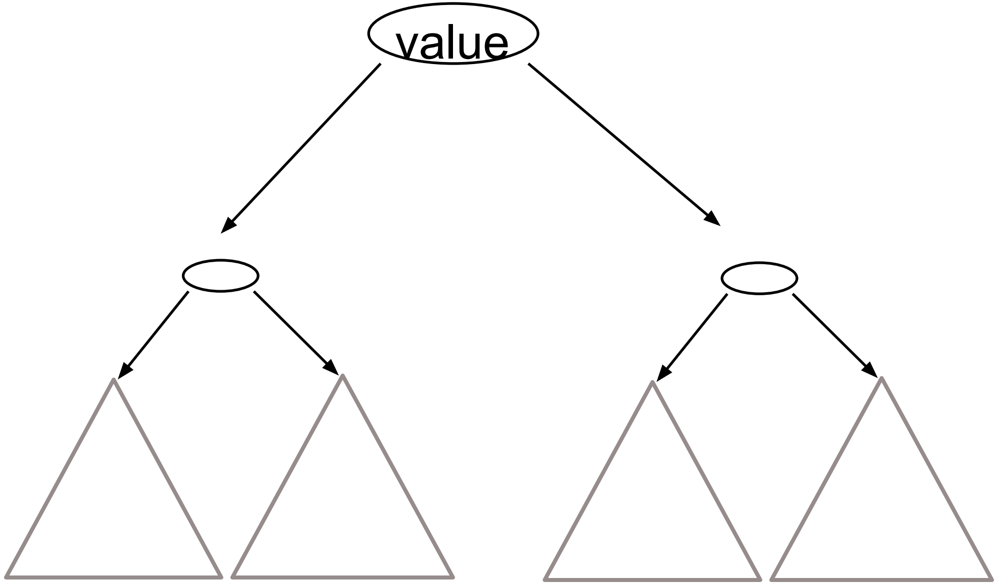


a binary tree

Looking at trees recursively



Looking at trees recursively



Recall: recursive functions

Base case:

If the input is “easy,” just solve the problem directly.

Recursive case:

Get a smaller part of the input (or several parts).
Call the function on the smaller value(s).
Use the recursive result to build a solution for the full input.

Recursive Functions on Binary Trees

Base case:

- empty tree (null)
- or, possibly, a leaf

Recursive case:

- Call the function on each subtree.
- Use the recursive result to build a solution for the full input.

Go through the tutorial http://www.cs.cornell.edu/courses/JavaAndDS/recursion/recursionTree.html

Tree traversals

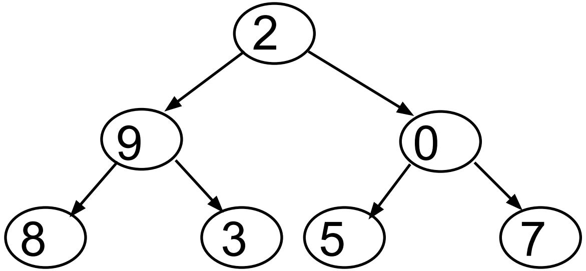
- "Walking" over the whole tree is a tree traversal
  - Done often enough that there are standard names
- In-order traversal
  - Process left subtree / Process root / Process right subtree
- Pre-order traversal
  - Process root / Process left subtree / Process right subtree
- Post-order traversal
  - Process left subtree / Process right subtree / Process root
- Level-order traversal
  - Not recursive: uses a queue (we'll cover this later)

Note: Can do other processing besides printing

Searching in a Binary Tree

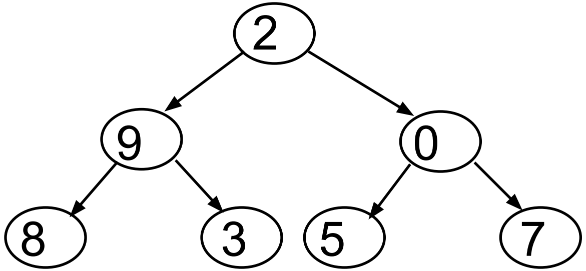
Analog of linear search in lists: given tree and an object, find out if object is stored in tree

Easy to write recursively, harder to write iteratively



Searching in a Binary Tree

```
/** Return true iff x is the datum in a node of tree  t*/
public static boolean treeSearch(T x, TreeNode<T> t) {
  if (t == null) return false;
  if (x.equals(t.value)) return true;
  return treeSearch(x, t.left) || treeSearch(x, t.right);
}
```



Analog of linear search in lists: given tree and an object, find out if object is stored in tree

Easy to write recursively, harder to write iteratively

Have we made search faster?

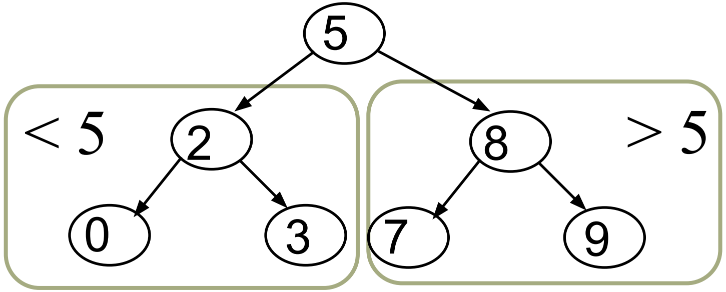
- What is the complexity of search on a tree?

Have we made search faster?

- What is the complexity of search on a tree?
- Bad news: it’s still O(n) in the worst-case
- There is no constraints on the positions of the elements in the tree, so have to go through the whole tree
- To improve the complexity of search, we want to impose some kind of structure on the positions of elements in the tree

Binary Search Tree (BST)

- A Binary Search Tree is a binary tree that is **ordered** and **has no duplicate values**
  - All nodes in the **left** subtree have values that are **less** than the value in that node
  - All values in the **right** subtree are greater

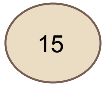


A BST is the key to making search way faster.

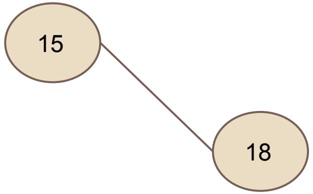
Building a BST

- To insert a new item:
  - Pretend to look for the item
  - Put the new node in the place where you fall off the tree

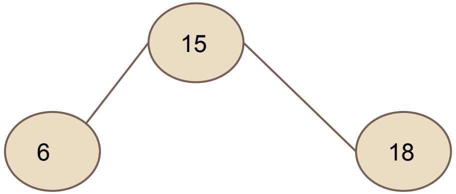
Building a BST



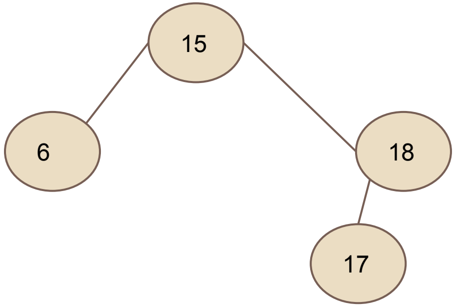
Building a BST



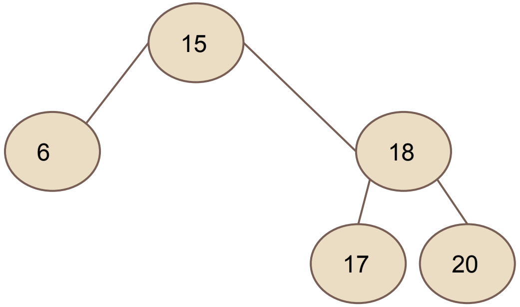
Building a BST



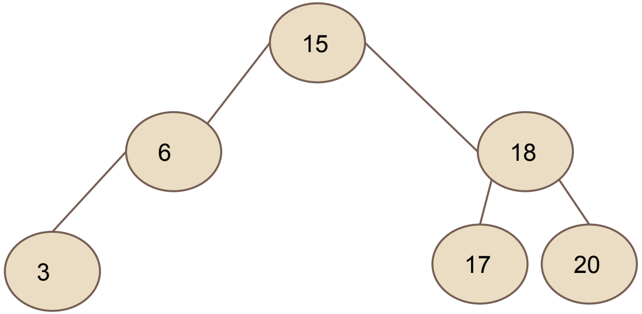
Building a BST



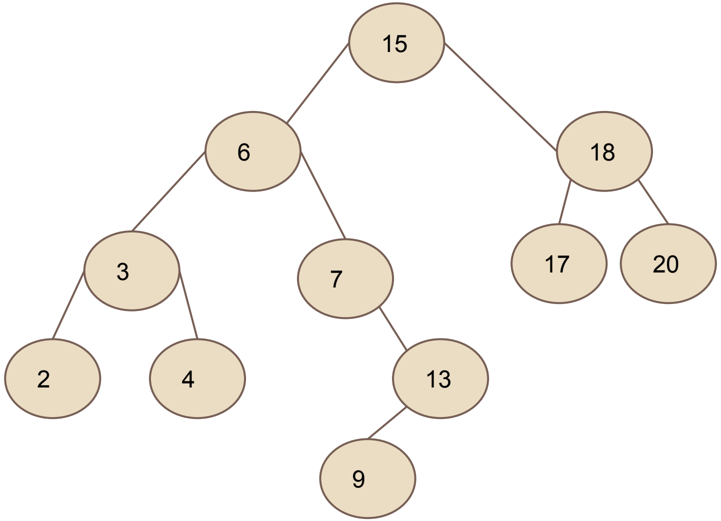
Building a BST



Building a BST



Building a BST



Sorting

Because of ordering rules for a BST, it's easy to print the items in alphabetical order

- Recursively print left subtree
- Print the node
- Recursively print right subtree

Sorting

Because of ordering rules for a BST, it's easy to print the items in alphabetical order

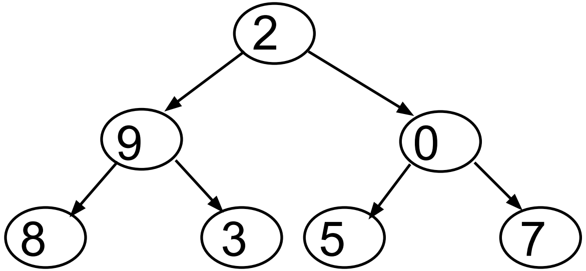
- Recursively print left subtree
- Print the node
- Recursively print right subtree

```
/** Print BST t in alpha order */
private static void print(TreeNode<T> t) {
   if (t== null) return;
   print(t.left);
   System.out.print(t.value);
   print(t.right);
}
```

Searching in a Binary Tree

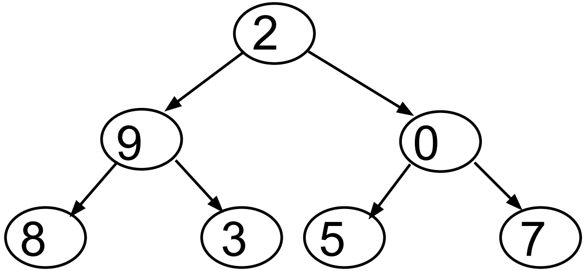
Analog of linear search in lists: given tree and an object, find out if object is stored in tree

Easy to write recursively, harder to write iteratively



Searching in a Binary Tree

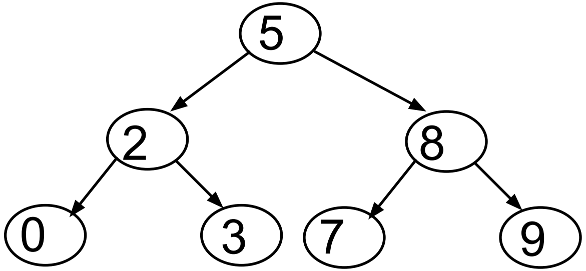
```
/** Return true iff x is the datum in a node of tree  t*/
public static boolean treeSearch(T x, TreeNode<T> t) {
   if (t == null) return false;
   if (x.equals(t.value)) return true;
   if (x < t.value) return treeSearch(x,t.left)
   else return treeSearch(x, t.right);
}
```



Analog of linear search in lists: given tree and an object, find out if object is stored in tree

Easy to write recursively, harder to write iteratively

Binary Search Tree (BST)



Compare binary tree to binary search tree:

```
boolean searchBT(n, v):
  if n==null, return false
  if n.v == v, return true
  return searchBST(n.left, v)
      || searchBST(n.right, v)
```

2 recursive calls

```
boolean searchBST(n, v):
  if n==null, return false
  if n.v == v, return true
  if v < n.v
    return searchBST(n.left, v)
  else
    return searchBST(n.right, v)
```

1 recursive call

Binary Search Tree (BST)

- What is the complexity of search in a binary search tree?

Binary Search Tree (BST)

- What is the complexity of search in a binary search tree?
- Unlike binary tree, structure allows you to explore a single branch in the tree
- Becomes O(depth)

Binary Search Tree (BST)

- What is the complexity of a binary search tree?
- Unlike binary tree, structure allows you to explore a single branch in the tree
- Becomes O(depth)

Data Structure	add(val x)	lookup(int i)	search(val x)
Array	$O(n)$	$O(1)$	$O(n)$
Linked List	$O(1)$	$O(n)$	$O(n)$
Binary Tree	$O(1)$	$O(n)$	$O(n)$
BST	$O(depth)$	$O(depth)$	$O(depth)$

Other operations

- Binary Search Trees aren't just useful for search operations

- They support efficient implements of
  - Finding the minimum/maximum of a collection of elements
  - Given an element, finding its predecessor/successor

Finding the Minimum

- Recall that elements that are smaller than the root node are to the left side of the tree.

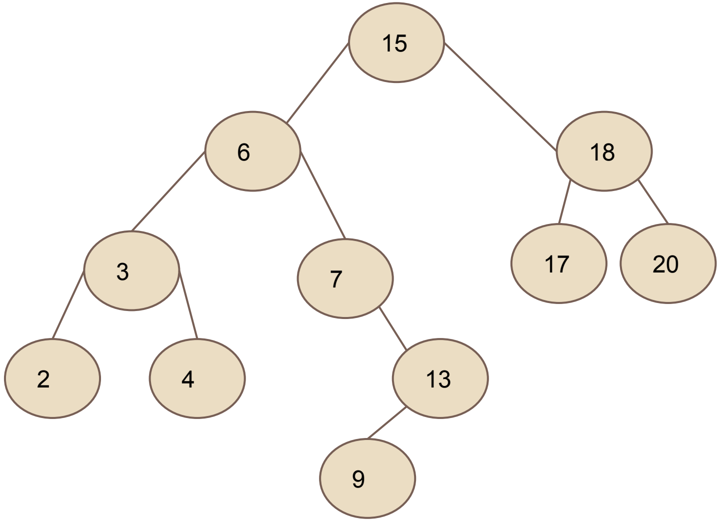
- Where do you think the smallest element of the binary tree is going to be?

Finding the Minimum

- Recall that elements that are smaller than the root node are to the left side of the tree.

- Where do you think the smallest element of the binary tree is going to be?

- It will be the left-most element of the tree



Finding the Maximum

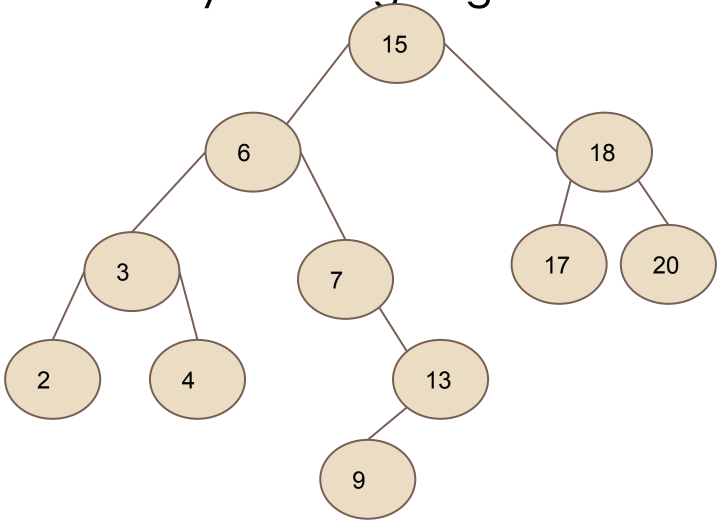
- Recall that elements that are larger than the root node are to the left side of the tree.
- Where do you think the largest element of the binary tree is going to be?

Finding the Maximum

- Recall that elements that are larger than the root node are to the left side of the tree.

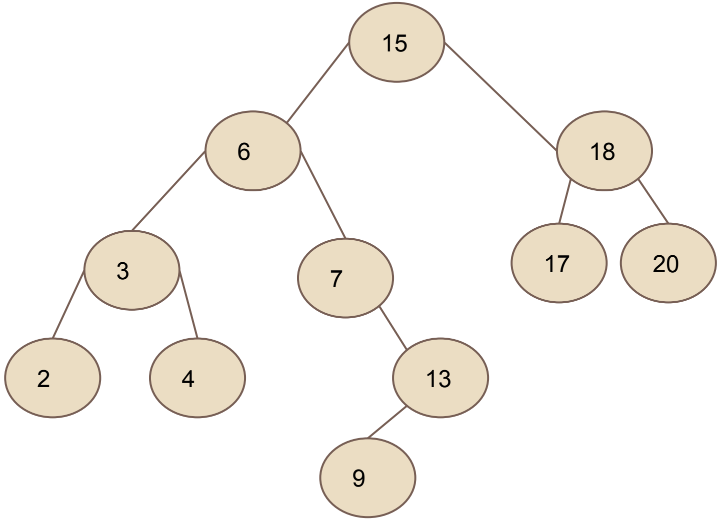
- Where do you think the largest element of the binary tree is going to be?

- It will be the right-most element of the tree



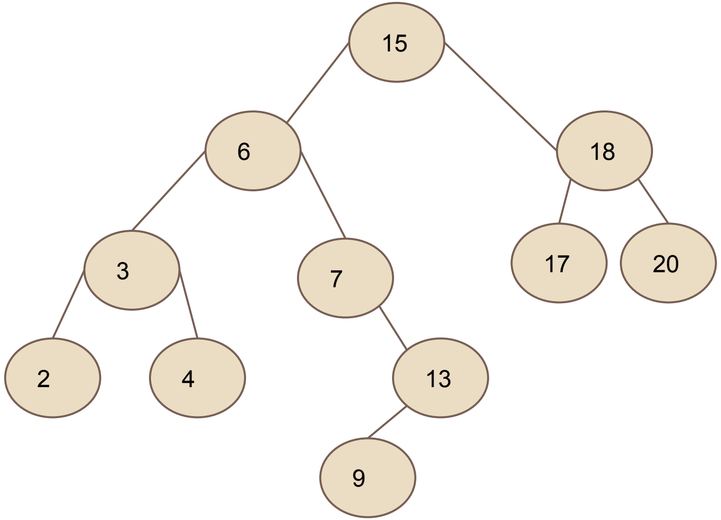
Finding the Successor

- Where is the **successor** of an element going to be in a BST?
  - Successor = successor of x is the node with the smallest key greater than x.
- Successor of 15 is:
  - 17
- Successor of 13 :
  - 15



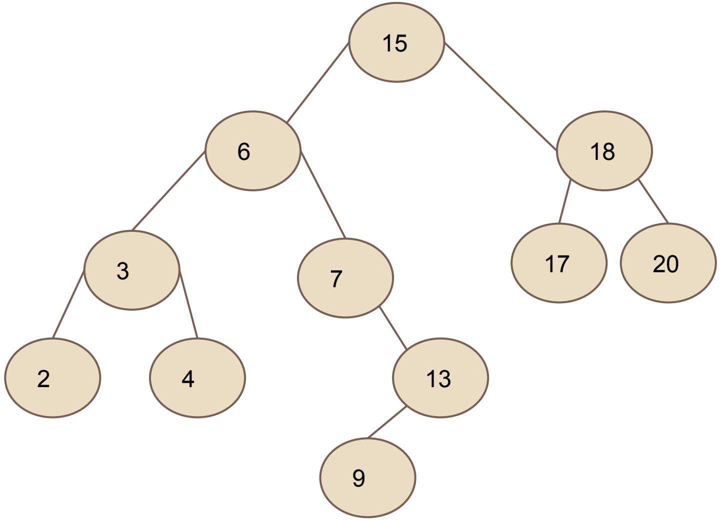
Finding the Successor

- To find the successor of **x**:
  - Two cases:
    - x has a right subtree: the minimum of the right subtree is x's successor
    - x has no right subtree: successor is the lowest ancestor of x whose left child is also an ancestor of x.



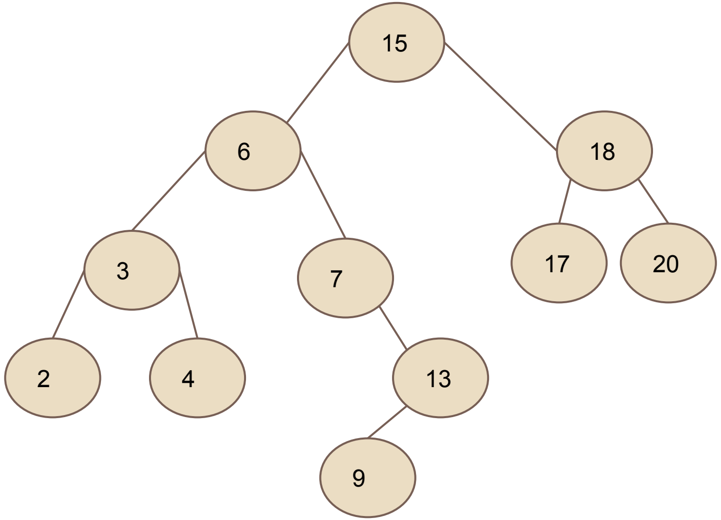
Finding the Successor

- To find the successor of **x**:
  - Two cases:
    - 15 has a right subtree and 17 is the minimum of that subtree
    - 13 has no right subtree, and the first element whose left child (6) is an ancestor of 13, is 15.



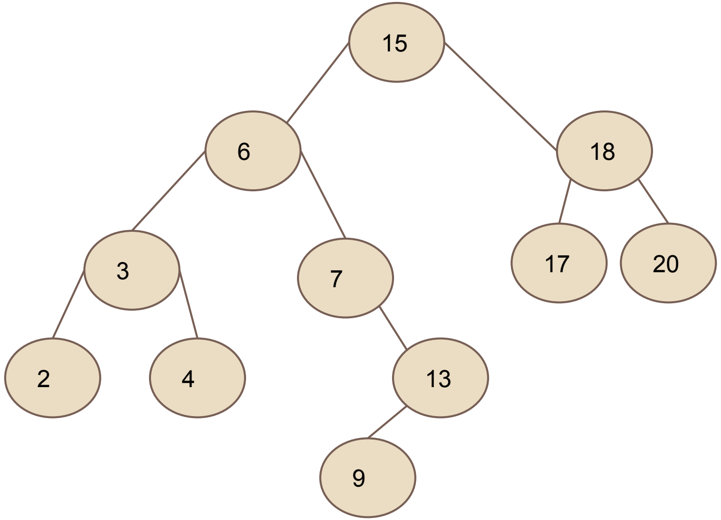
Finding the Predecessor

- Where is the **predecessor** of an element going to be in a BST?
  - Predecessor = predecessor of x is the node with the greatest key smaller than x.
- Predecessor of 15 is:
  - 13
- Predecessor of 7 :
  - 6



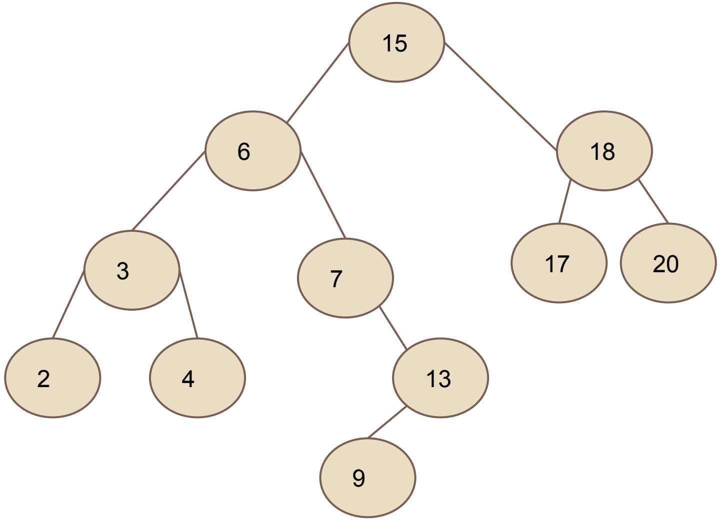
Finding the Predecessor

- To find the predecessor of **x**:
  - Two cases:
    - x has a left subtree:the maximum of the left subtree is x's predecessor
    - x has no right subtree: predecessor is the lowest ancestor of x whose rightchild is also an ancestor of x.



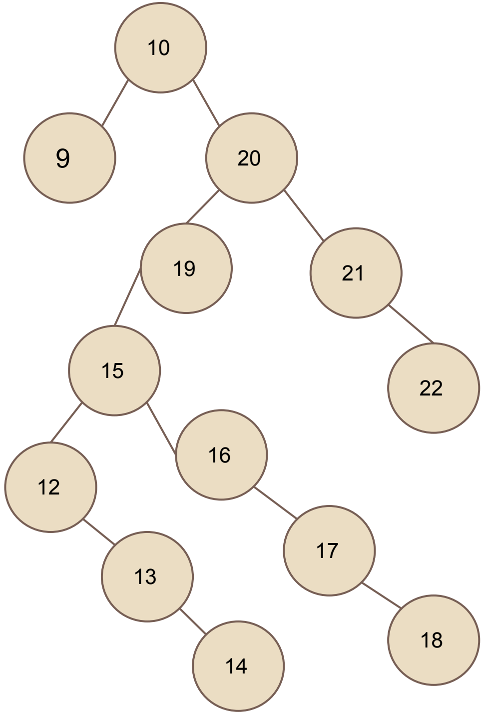
Finding the Predecessor

- To find the predecessor of **x**:
  - Two cases:
    - 15 has a left subtree and 13 is the maximum of that subtree
    - 7 has no left subtree, and the first element whose right child is an ancestor of 7, is 6.



Deleting

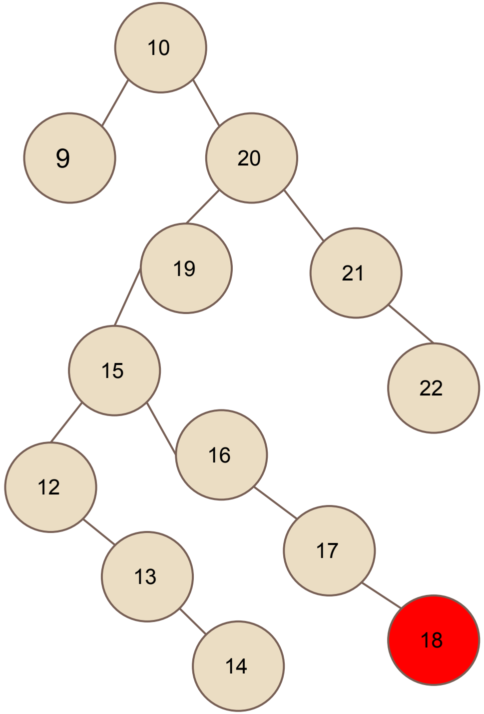
- To delete a node in a BST, distinguish between three cases:
  - Case 1: The node has no children
  - Case 2: The node has one child
  - Case 3: The node has two children



Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 1: The node has no children

Consider deleting node 18

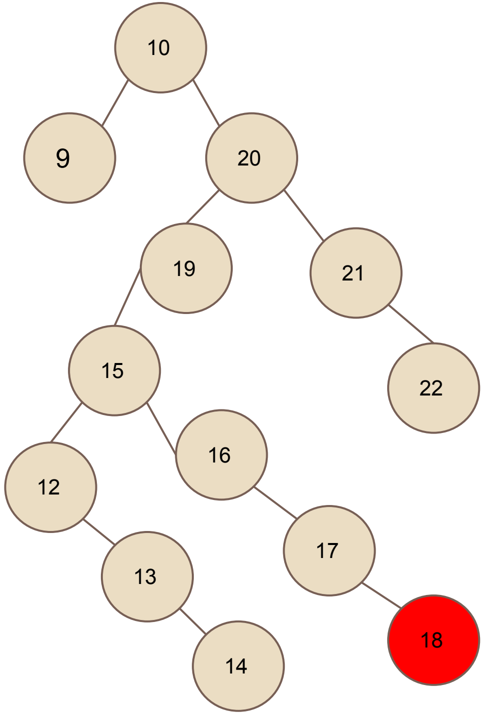


Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 1: The node has no children

Consider deleting node 18

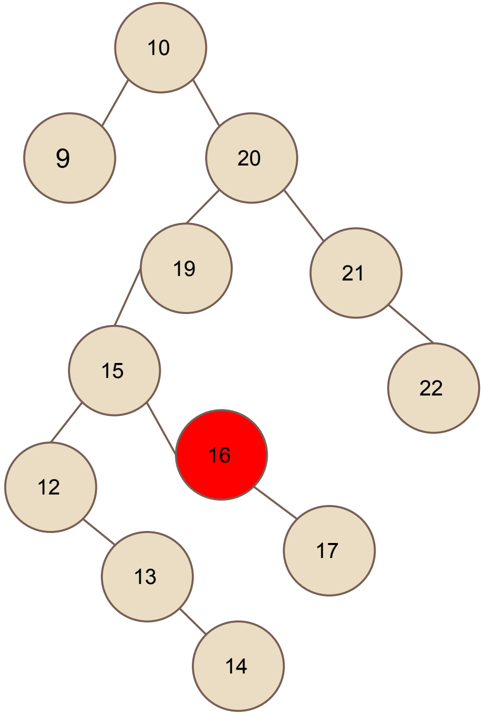
Simply remove 18 from the tree, setting the right (or left) pointer of its parent to null



Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 2: The node has one child

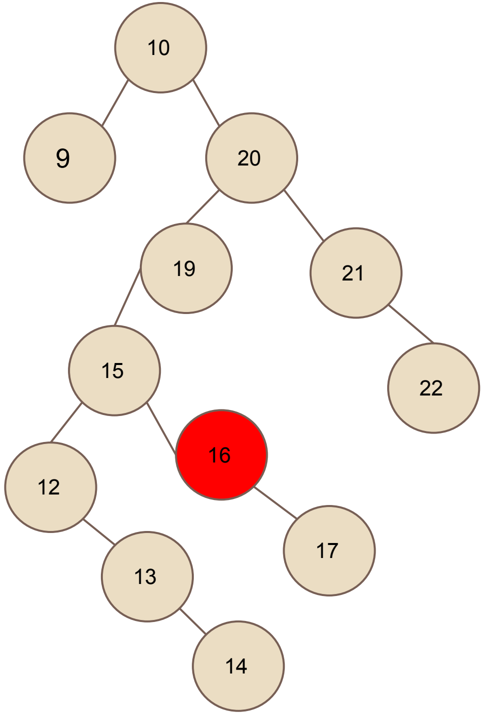
Consider deleting node 16



Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 2: The node has one child

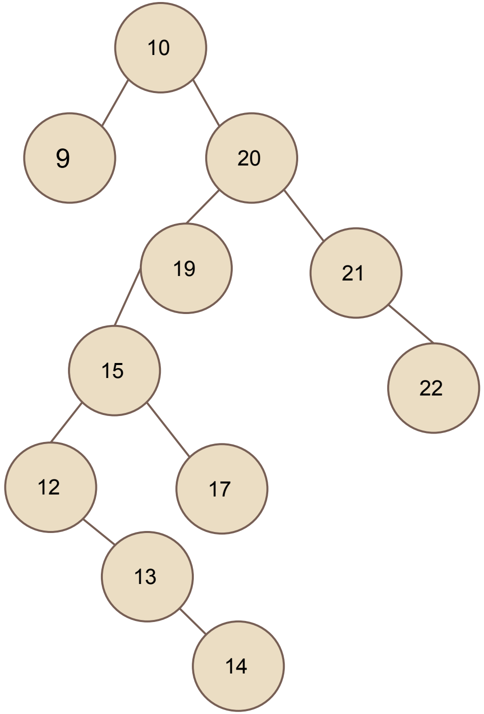
Remove node from tree and set the right (/left) pointer of its parent to the child subtree of the node being deleted



Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 2: The node has one child

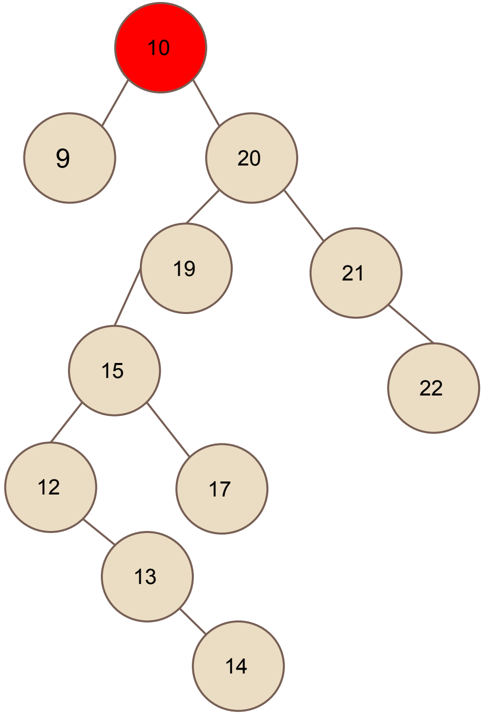
Remove node from tree and set the right (/left) pointer of its parent to the child subtree of the node being deleted



Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 3: The node has two children

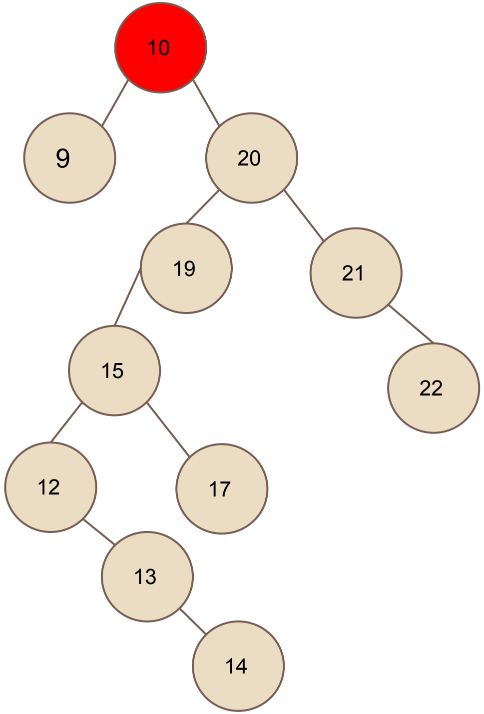
More complicated. Proceed in several steps.



Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 3: The node has two children

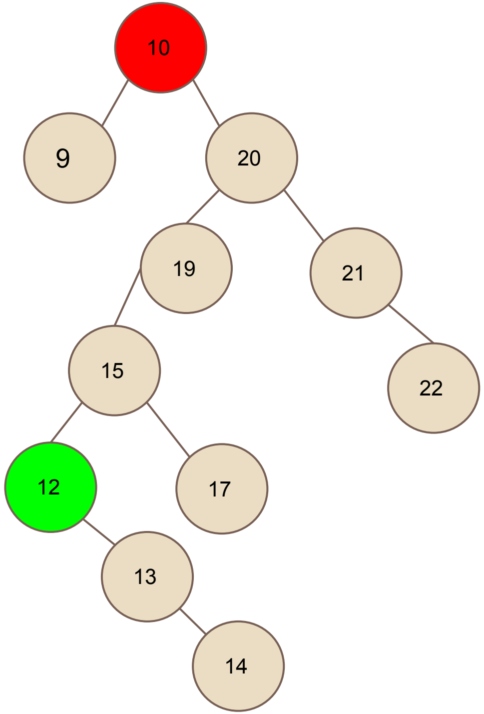
Step 1: find the successor of 10 in the tree.



Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 3: The node has two children

Step 1: find the successor of 10 in the tree. Smallest value that's greater than 10.

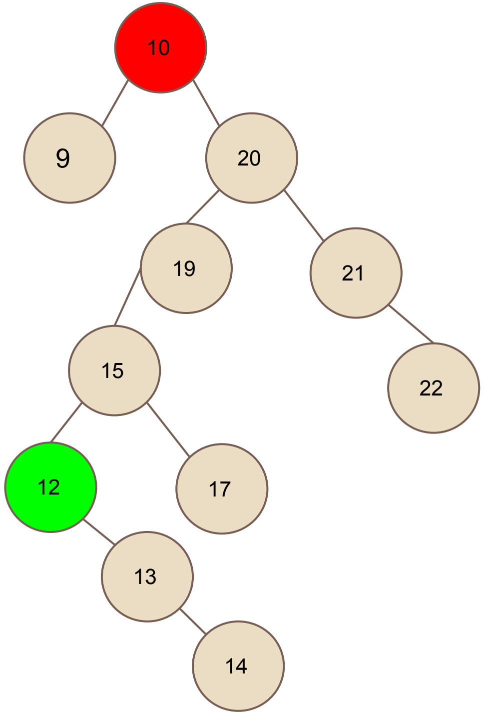


Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 3: The node has two children

Step 1: find the successor of 10 in the tree. Smallest value that's greater than 10.

Step 2: replace the value to be deleted by its successor

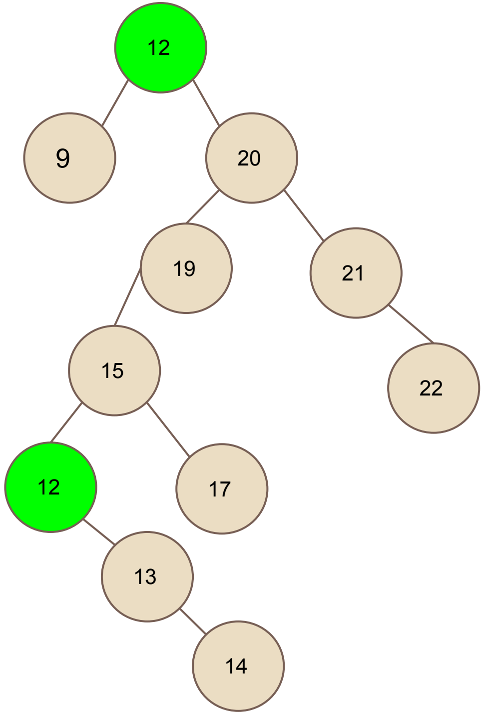


Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 3: The node has two children

Step 1: find the successor of 10 in the tree. Smallest value that's greater than 10.

Step 2: replace the value to be deleted by its successor



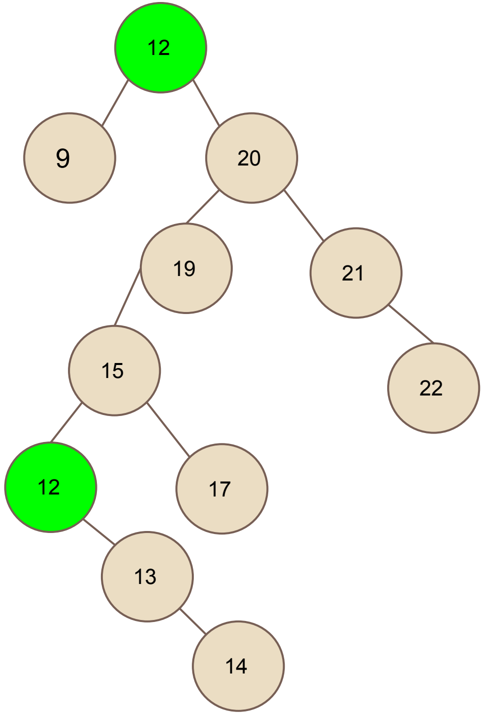
Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 3: The node has two children

Step 1: find the successor of 10 in the tree. Smallest value that's greater than 10.

Step 2: replace the value to be deleted by its successor

Step 3: delete the successor by applying *Case 2*



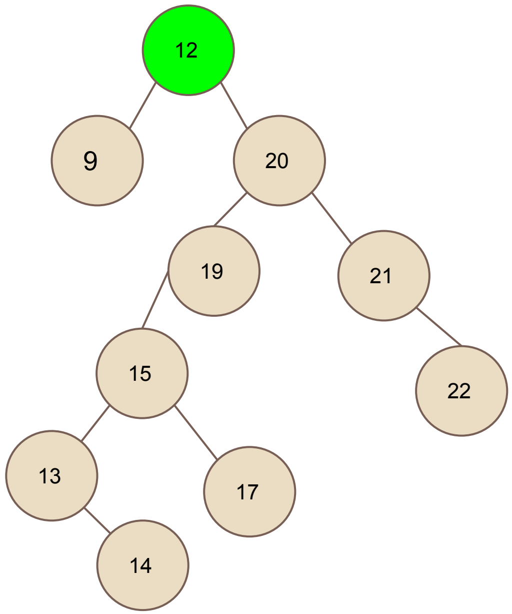
Deleting

- To delete a node in a BST, distinguish between three cases:
  - Case 3: The node has two children

Step 1: find the successor of 10 in the tree. Smallest value that's greater than 10.

Step 2: replace the value to be deleted by its successor

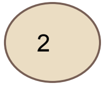
Step 3: delete the successor by applying *Case 2*



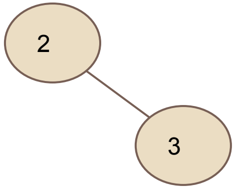
Are we done?

- We wanted an efficient way to do search.
- We know that Binary Search Tree Search has complexity O(height).
- Is that good enough?

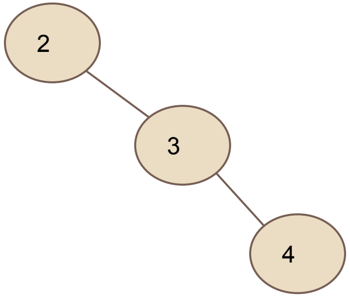
Inserting in Sorted Order



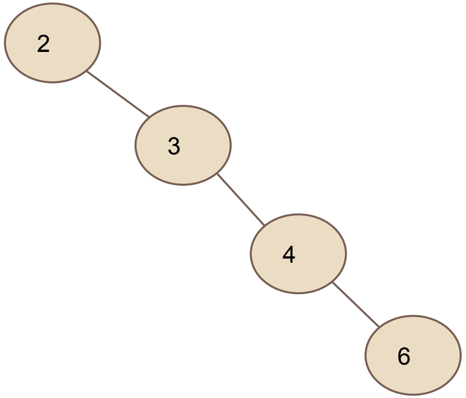
Inserting in Sorted Order



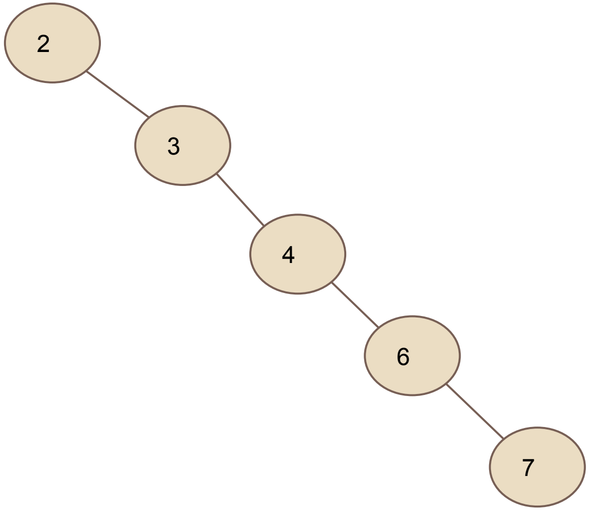
Inserting in Sorted Order



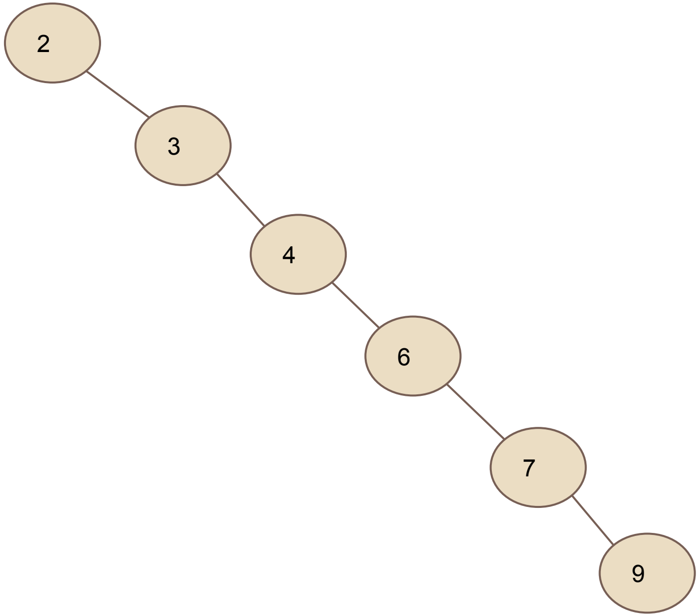
Inserting in Sorted Order



Inserting in Sorted Order



Inserting in Sorted Order



Insertion Order Matters

- A *balanced* binary tree is one where the two subtrees of any node are about the same size.
- Searching a binary search tree takes O(depth) time, where h is the height of the tree.
- But if you insert data in sorted order, the tree becomes imbalanced, so searching is O(n) again
- So we haven’t found a way to improve our worst-case complexity!
- Need a way to ensure tree remains balanced

Balancing a BST

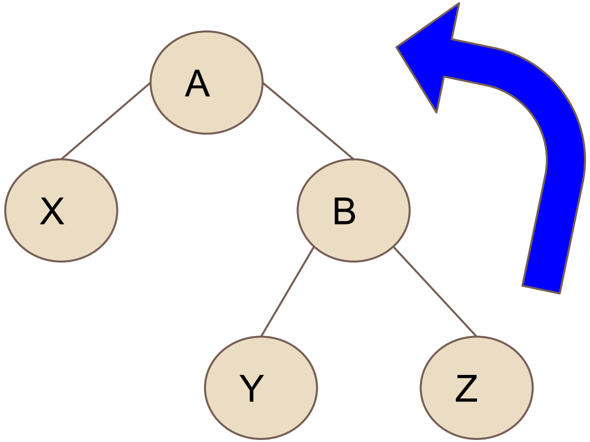
- Balancing a BST is necessary to achieve good performance.

- To balance a tree, we will either:
  - Left-rotate a tree
  - Right-rotate a tree

- Left-rotation
  - Shortens right-subtree by 1, lengthens left subtree by 1
- Right rotation does the opposite

Left Rotation

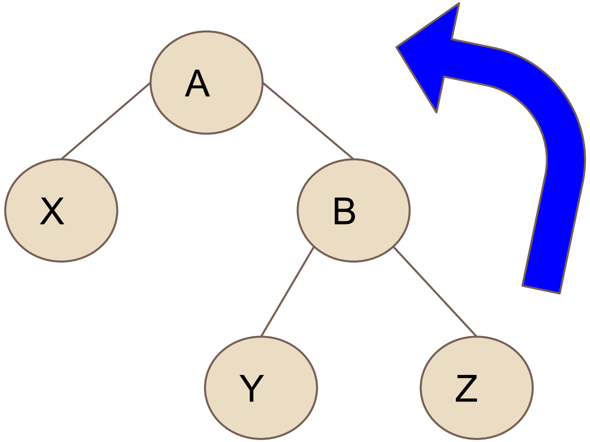
- Left-rotation rotates the right subtree of a BST to the left.



Left Rotation

- Left-rotation rotates the right subtree of a BST to the left.

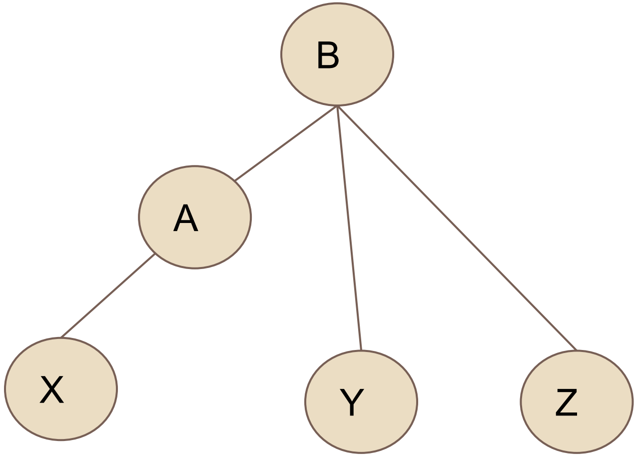
Place the root of the right subtree as the new root of the tree.



Left Rotation

- Left-rotation rotates the right subtree of a BST to the left.

Place the root of the right subtree as the new root of the tree.



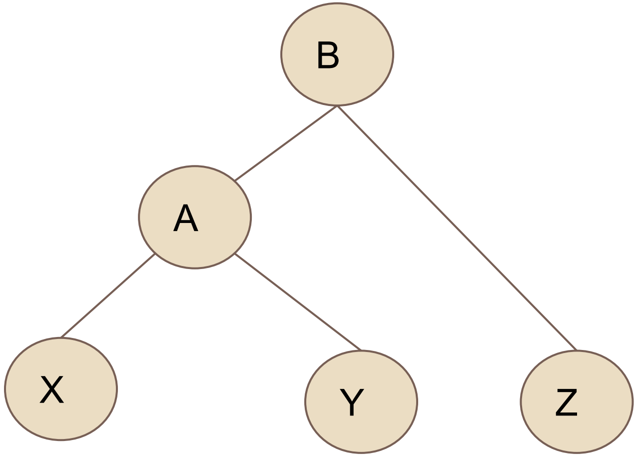
Left Rotation

- Left-rotation rotates the right subtree of a BST to the left.

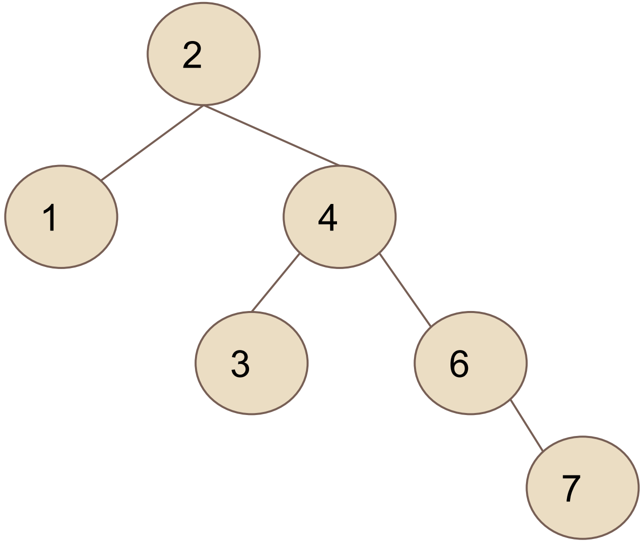
Place the root of the right subtree as the new root of the tree.

Move the left subtree of the new root as the right subtree of the old root.

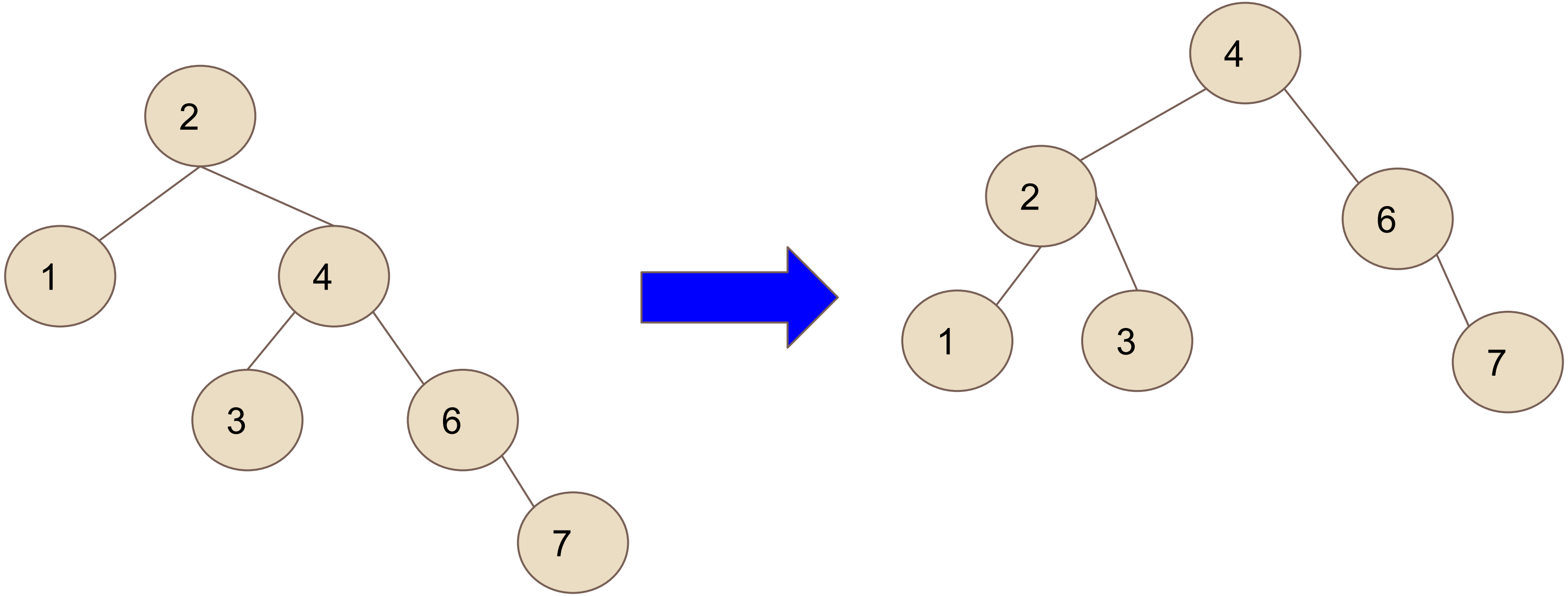
To help you understand why that works, remember the ordering relationships on subtrees!



Left Rotation

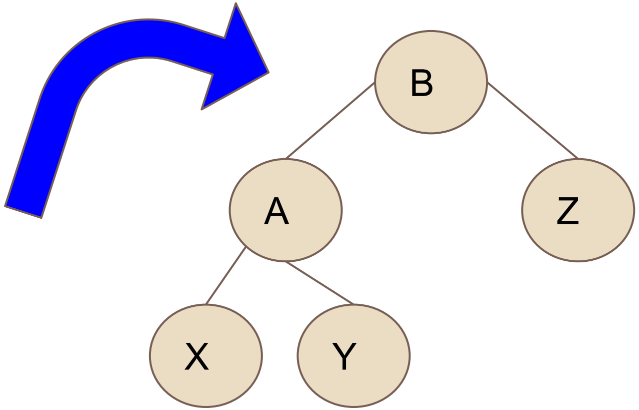


Left Rotation



Right Rotation

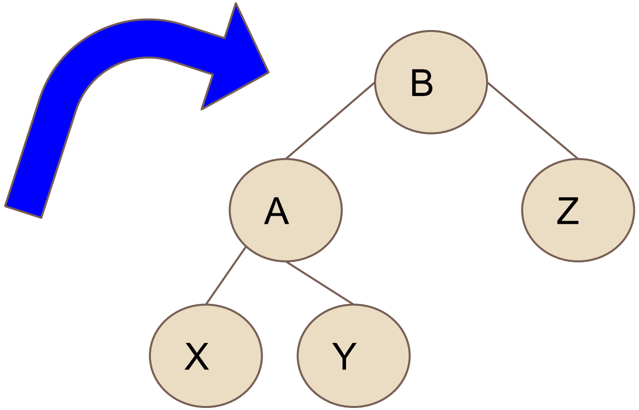
- Right-rotation rotates the left subtree of a BST to the right.



Right Rotation

- Right-rotation rotates the left subtree of a BST to the right.

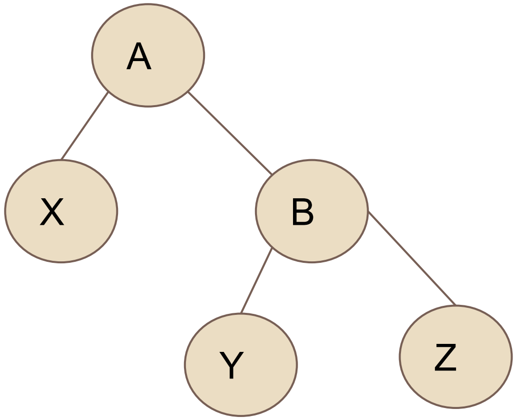
  Inverse of left: make left subtree the root, placing B as the right subtree of A, and placing the right subtree of A as the new left subtree of B



Right Rotation

- Right-rotation rotates the left subtree of a BST to the right.

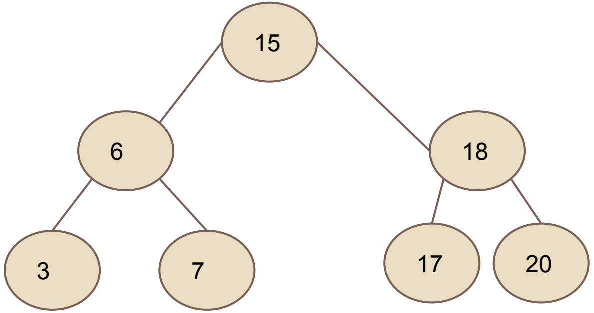
  Inverse of left: make left subtree the root, placing B as the right subtree of A, and placing the right subtree of A as the new left subtree of B



Next Class

A BST works great as long as it's *balanced*.

There are kinds of trees that can *automatically* keep themselves balanced as you insert things!



We'll be looking at Red-Black trees, which is the datastructure that **TreeSet** in Java uses.

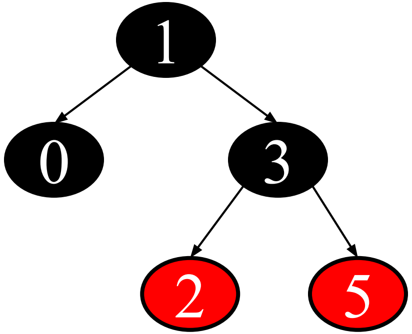
Balanced Search Trees

- Goal is to ensure that the height of the tree is always O(log n)
    - This enables search/insert/delete/min/max/pred/succ to also be O(log n)

- Note: O(log n) is the best you can do for binary trees
    - all operations must at least go down one full branch
    - you need at least O(log n) levels to store n elements

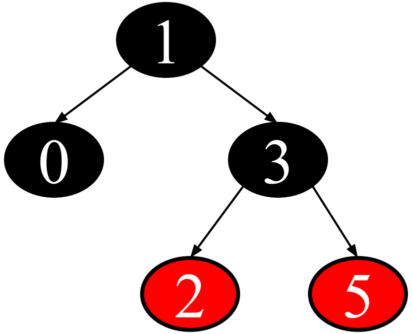
Red-Black Trees

- Self-balancing BST
- Each node has one extra bit of information "colour"
- Constraints on how nodes can be coloured enforces approximate balance



Why red-black?

- Different explanations:
  - Option 1: they only had red and black pens at the time
  - Option 2: red was the nicest colour that the Xerox Parc printer could print

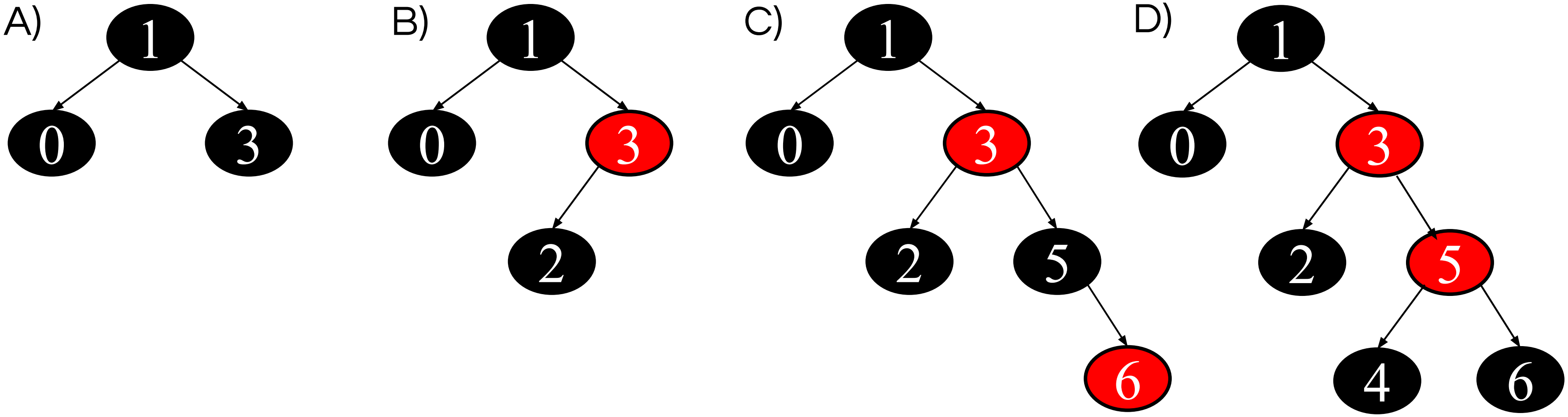


Red-Black Trees

1) A red-black tree is a binary search tree.
2) Every node is either red or black.
3) The root is black.
4) If a node is red, then its (non-null) children are black.
5) For each node, every path to a descendant null node contains the same number of black nodes.

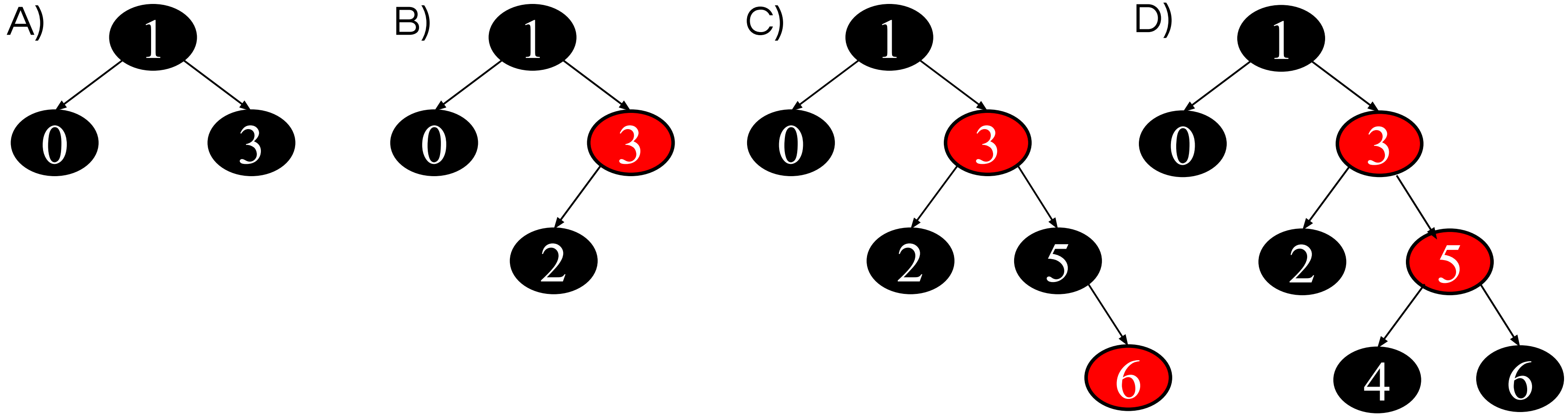
RB Tree Quiz

- Which of the following are red-black trees?



RB Tree Quiz

- Which of the following are red-black trees?



A) YES

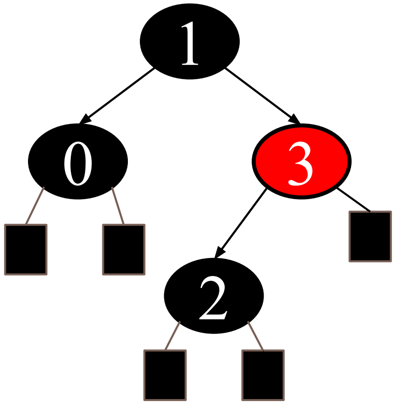
B) NO

C) YES

D) NO

Warning

- You will sometimes see this invariant:
  - All leaves (nil) of a Red-Black tree are black
- And see red-black trees drawn like this:
  - With NIL leaves
  - It makes implementing the functionality easier
- For simplicity, we don't represent them in this class



Is this magic?

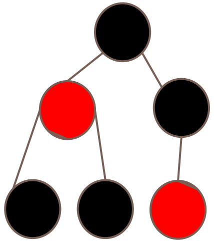
- Red-Black tree invariants can appear quite random
    - But they are key to guaranteeing that the tree is “mostly” balanced

- Intuitively:
    - Property 5: (each branch contains the same number of black nodes) ensures that the tree is perfectly balanced if it does not contain red nodes
    - Property 4 ensures that there can never be two consecutive red nodes in a branch. This guarantees that, for a tree with k black nodes, there can be at most k red nodes. So adding the red nodes only increases the height by a factor of two.

- A subtree can therefore have, at most, a height twice greater than the other subtrees.

Proving that height is O(log n)

- Let BH(x) be the number of black nodes on every x-to-leaf path.

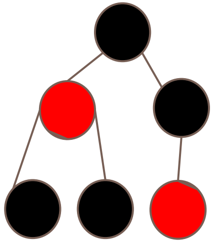


BH(x) = 2

- **Lemma 1: A subtree rooted at x has at least 2^BH(x) - 1 nodes**

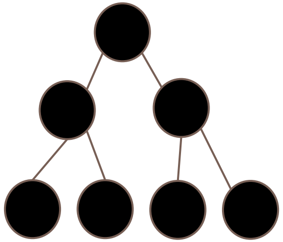
Proving that height is O(log n)

- Let BH(x) be the number of black nodes on every x-to-leaf path.



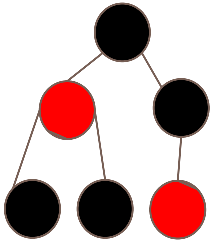
BH(x) = 2

- **Lemma 1: A subtree rooted at x has at least 2^BH(x) - 1 nodes**
  - Suppose that x's subtree has only black nodes. By Property 5, the tree is complete



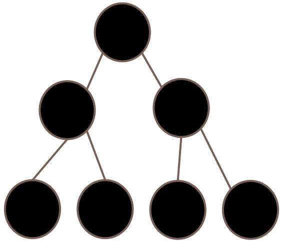
Proving that height is O(log n)

- Let BH(x) be the number of black nodes on every x-to-leaf path.



BH(x) = 2

- **Lemma 1: A subtree rooted at x has at least 2^BH(x) - 1 nodes**
  - Suppose that x's subtree has only black nodes. By Property 5, the tree is complete
  - A **complete** tree has 2^(height + 1) - 1 nodes (recall the formula). So 2^BH(x) -1 nodes
    If red nodes are included, BH(x) doesn't change
    So the number of nodes is still at least 2^BH(x) -1



Proving that height is O(log n)

1) If a node is red, then its (non-null) children are black.

2) For each node, every path to a descendant null node contains the same number of black nodes.

- **Lemma 2: Let h be the height of the tree. Then BH(root) >= h/2**

Proving that height is O(log n)

1) If a node is red, then its (non-null) children are black.

2) For each node, every path to a descendant null node contains the same number of black nodes.

- **Lemma 2: Let h be the height of the tree. Then BH(root) >= h/2**
  - By property 4, a red node cannot be the parent of another red node. So red and black nodes must be interleaved. Because red nodes can't be consecutive, each root-to-leaf path can never have more than h/2 red nodes. So BH(root) >= h/2

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  -

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  - n >= 2^BH(root) -1 (Lemma 1)

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  - n >= 2^BH(root) -1 (Lemma 1)
  - n >= 2^(h/2) -1 >= 2^BH(root) -1 (by Lemma 2: BH(root) > h/2)

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  - $n >= 2^{BH(root)} - 1$ (Lemma 1)
  - $n >= 2^{(h/2)} - 1 >= 2^{BH(root)} - 1$ (by Lemma 2: $BH(root) > h/2$)
  - $n + 1 >= 2^{(h/2)}$

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  - n >= 2^BH(root) -1 (Lemma 1)
  - n >= 2^(h/2) -1 >= 2^BH(root) -1 (by Lemma 2: BH(root) > h/2)
  - n + 1 >= 2^(h/2)
  - log(n+1) >= log(2^(h/2))

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  - n >= 2^BH(root) -1 (Lemma 1)
  - n >= 2^(h/2) -1 >= 2^BH(root) -1 (by Lemma 2: BH(root) > h/2)
  - n + 1 >= 2^(h/2)
  - log(n+1) >= log(2^(h/2))
  - log(n+1) >= h/2

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  - n >= 2^BH(root) -1 (Lemma 1)
  - n >= 2^(h/2) -1 >= 2^BH(root) -1 (by Lemma 2: BH(root) > h/2)
  - n + 1 >= 2^(h/2)
  - log(n+1) >= log(2^(h/2))
  - log(n+1) >= h/2
  - 2log(n+1) >= h
  -

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  - n >= 2^BH(root) -1 (Lemma 1)
  - n >= 2^(h/2) -1 >= 2^BH(root) -1 (by Lemma 2: BH(root) > h/2)
  - n + 1 >= 2^(h/2)
  - log(n+1) >= log(2^(h/2))
  - log(n+1) >= h/2
  - 2log(n+1) >= h
  - 2log(2n) > 2log(n+1) >= h

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  - n >= 2^BH(root) -1 (Lemma 1)
  - n >= 2^(h/2) -1 >= 2^BH(root) -1 (by Lemma 2: BH(root) > h/2)
  - n + 1 >= 2^(h/2)
  - log(n+1) >= log(2^(h/2))
  - log(n+1) >= h/2
  - 2log(n+1) >= h
  - 2log(2n) > 2log(n+1) >= h
  - 2log(2) + 2log(n) > 2log(n+1) >= h

Proving that height is O(log n)

- **Theorem: The height h of a Red-Black tree is O(log n)**
  - n >= 2^BH(root) -1 (Lemma 1)
  - n >= 2^(h/2) -1 >= 2^BH(root) -1 (by Lemma 2: BH(root) > h/2)
  - n + 1 >= 2^(h/2)
  - log(n+1) >= log(2^(h/2))
  - log(n+1) >= h/2
  - 2log(n+1) >= h
  - 2log(2n) > 2log(n+1) >= h
  - 2log(2) + 2log(n) > 2log(n+1) >= h
  - O(1) + c*log(n) > h

h is log(n)

Red-Black Trees are popular

- They underpin the datastructure in **Java Treeset**
- The C++ STL library uses them internally to implement **Set** and **Map**
- They are used to schedule processes in the Linux Kernel
  - Specifically in the **Completely Fair Scheduler** (CFS)
- They are used to **manage memory** allocated to processes in the Linux Kernel

Class for a RBNode

Class for a RBNode

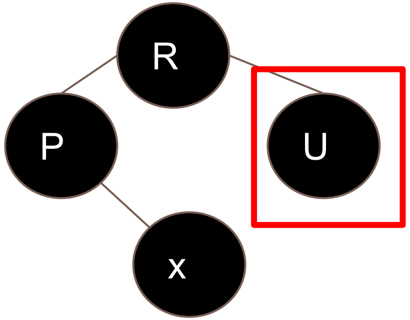
```
class RBNode<T> {
   private T value;
   private Colour colour;
   private RBNode<T> parent;
   private RBNode<T> left, right;

   /** Constructor: one-node tree with value x */
   public RBNode (T v, Colour c) { value= d; colour= c; }


   ...
}
```

Insertion

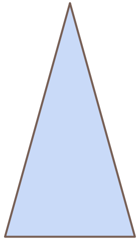
- High-level idea
  - Insert a node in the tree as you would in a BST and **mark it as red**
  - This may violate the RB-tree invariants
    - There may be two consecutive red nodes, causing the tree to be unbalanced.
  - Must "fix" the tree by **rotating** the subtrees appropriately
  - Rotating the subtrees may create new violations. Continue recursively until invariant has been restored.

Insertion

- Let’s define the notion of an uncle node:
  - An uncle node for x is the sibling of the parent of x



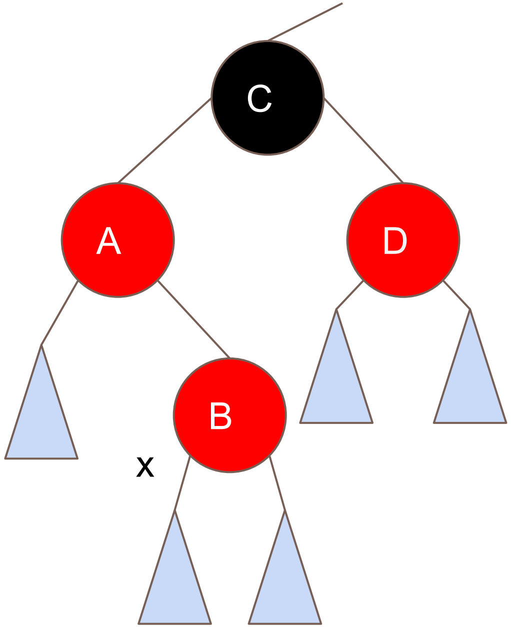
- Let’s write a subtree consisting of black root as



- Insertion can only violate Property 4. Once node has been inserted into appropriate position, must fix the tree

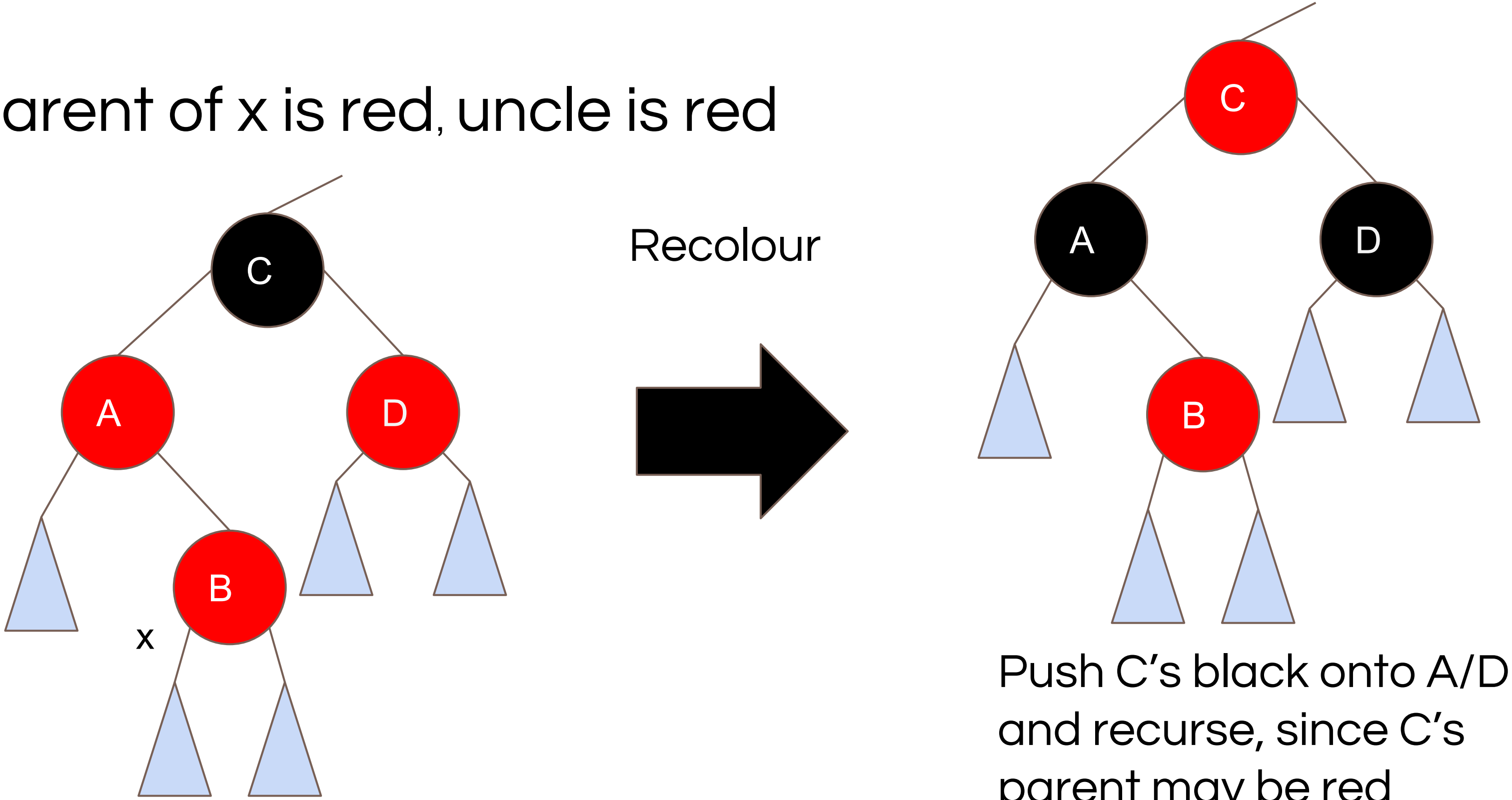
Case 1

- Parent of x is red, uncle is red



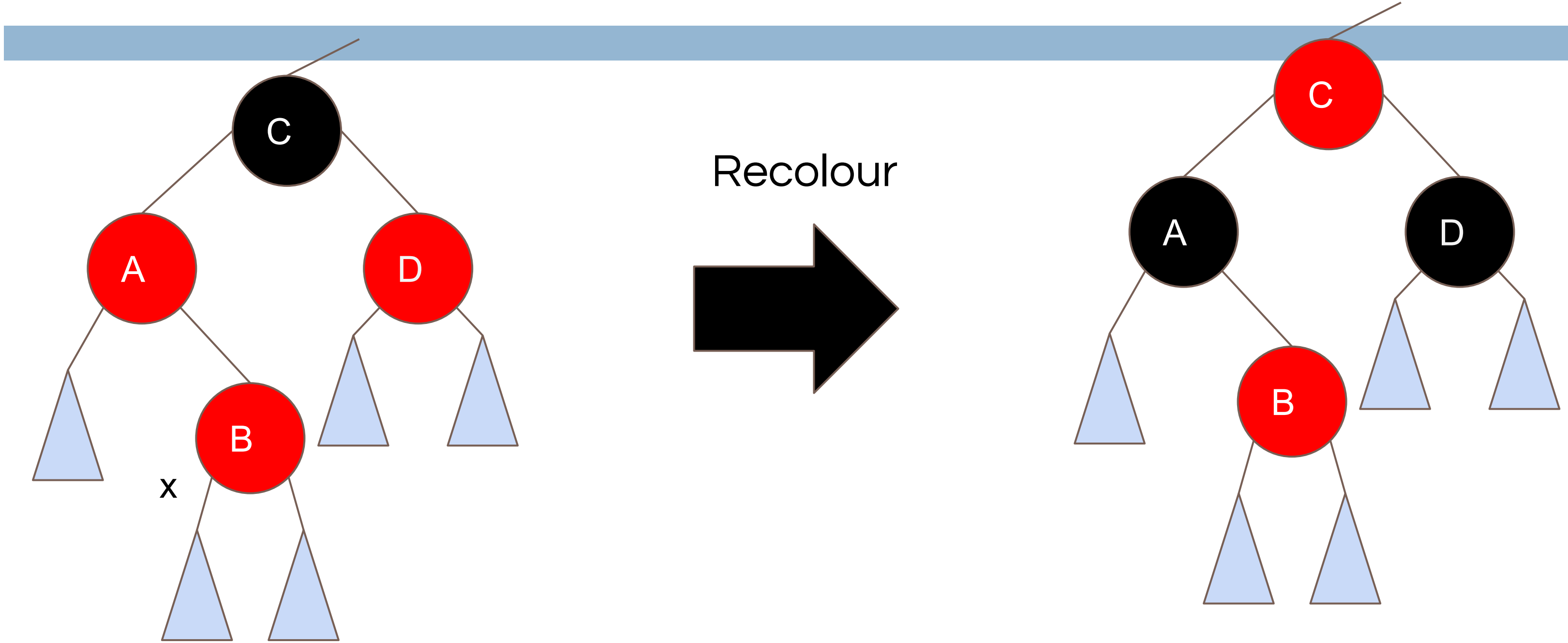
Case 1

- Parent of x is red, uncle is red



Push C’s black onto A/D and recurse, since C’s parent may be red

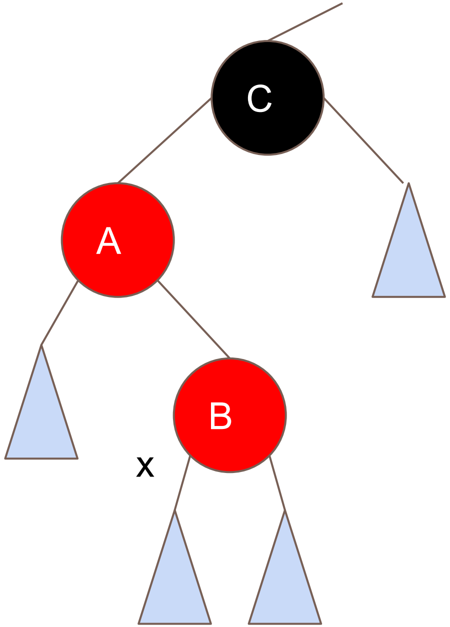
Case 1



Intuitively: A and D are both new inserted nodes inserted on both sides of the subtrees, so it’s “safe” to mark them black without rotating. However, the subtree rooted at the parent of C, may still be unbalanced by the insertion of B, hence why we mark C red.

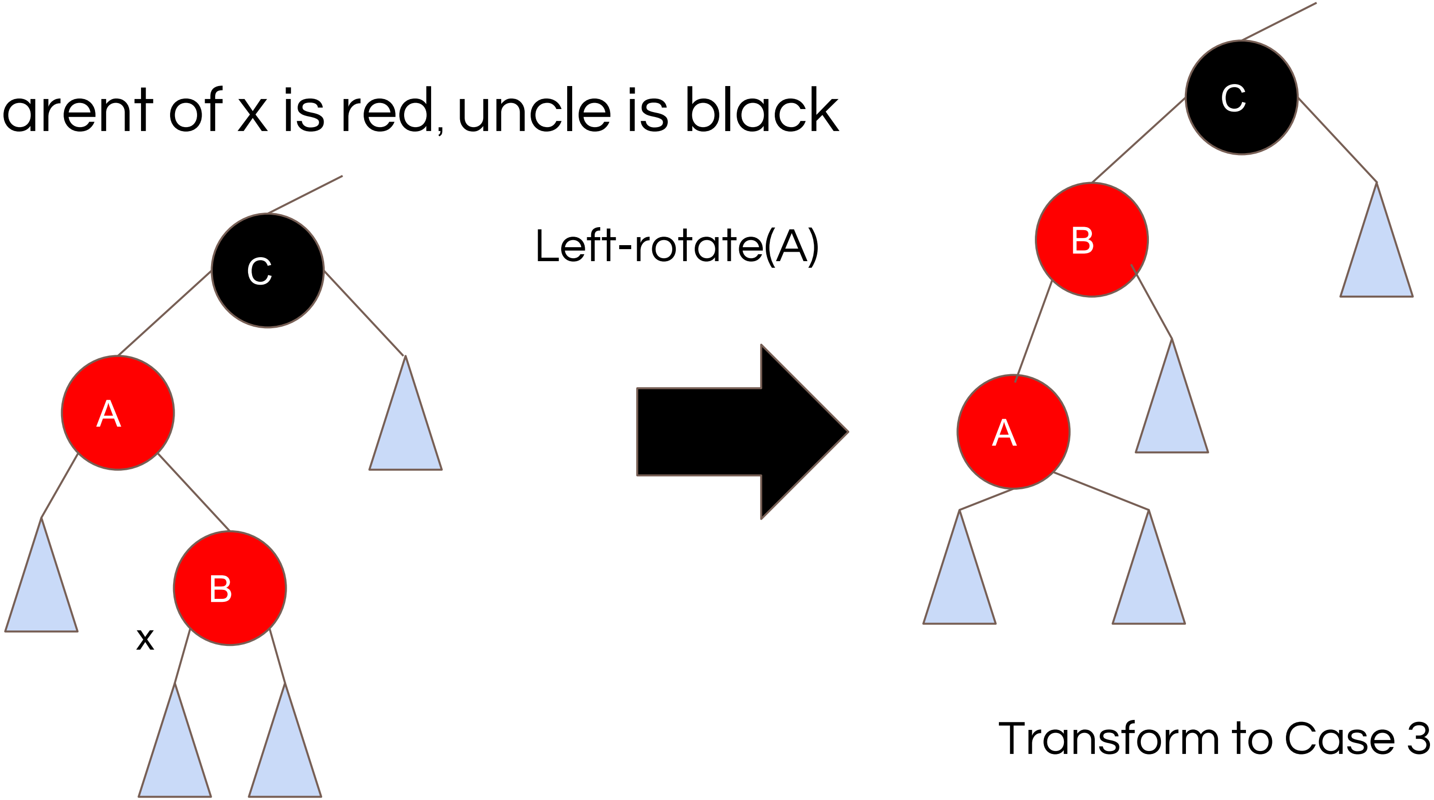
Case 2

- Parent of x is red, uncle is black



Case 2

- Parent of x is red, uncle is black

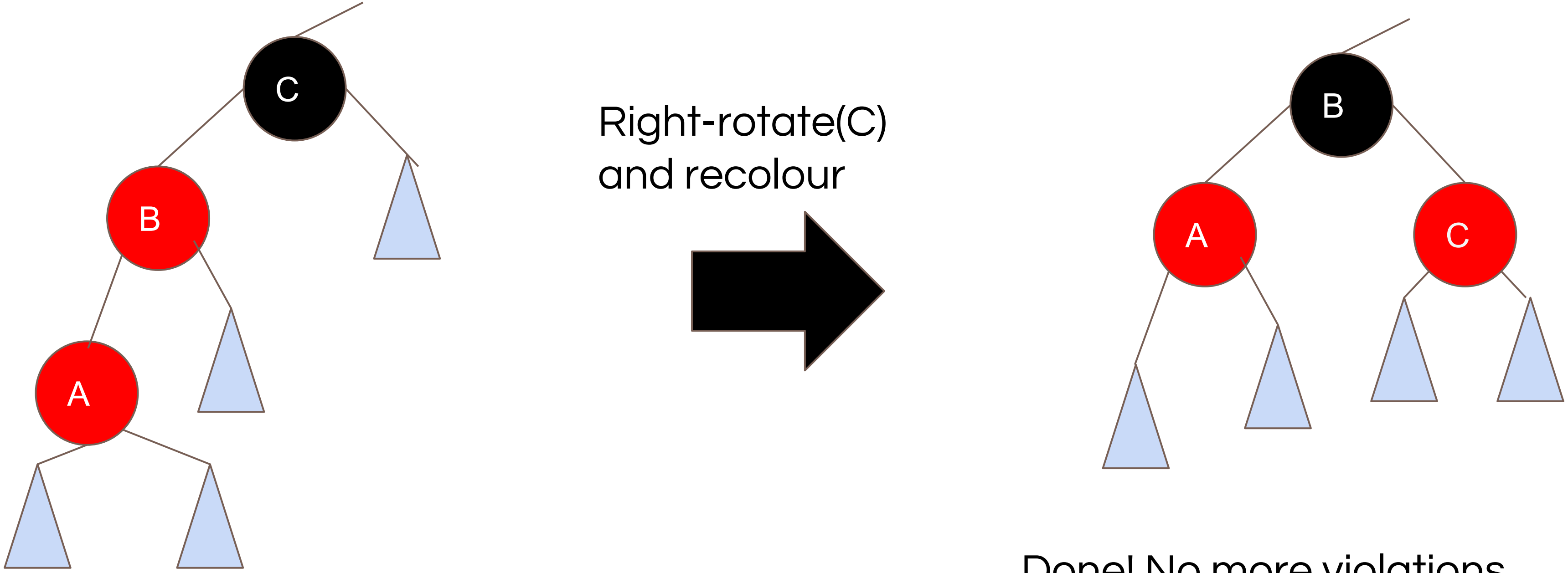


Left-rotate(A)

Transform to Case 3

Case 3

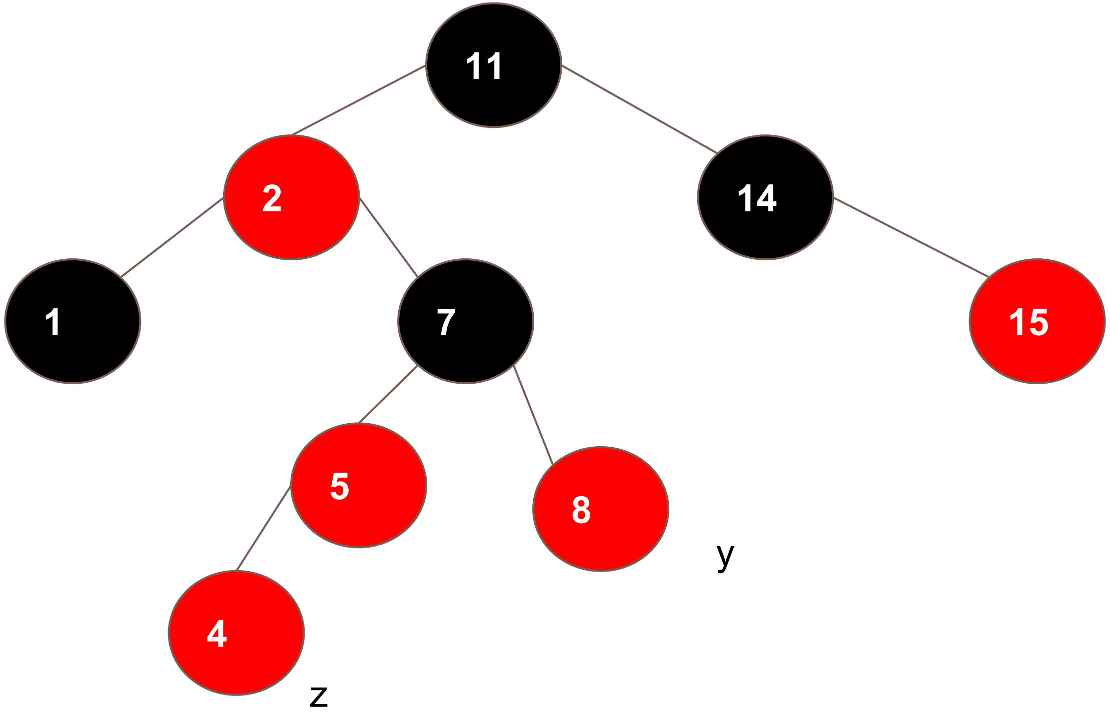
- Parent of x is red, uncle is black



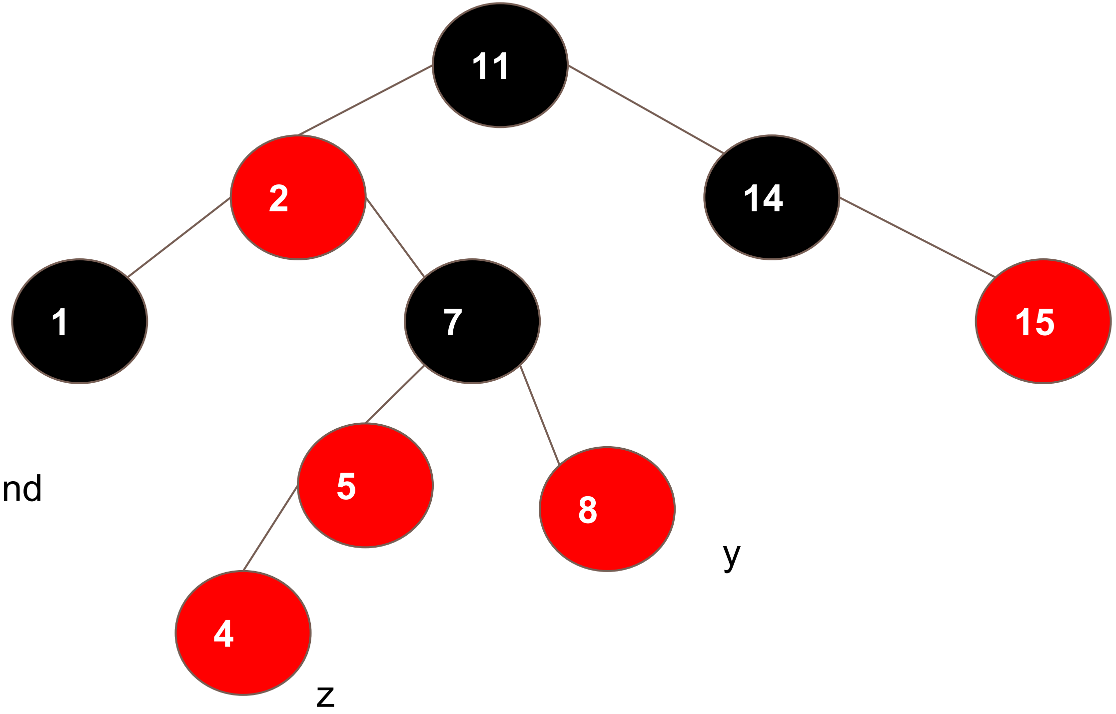
Right-rotate(C)
and recolour

Done! No more violations
are possible

An example

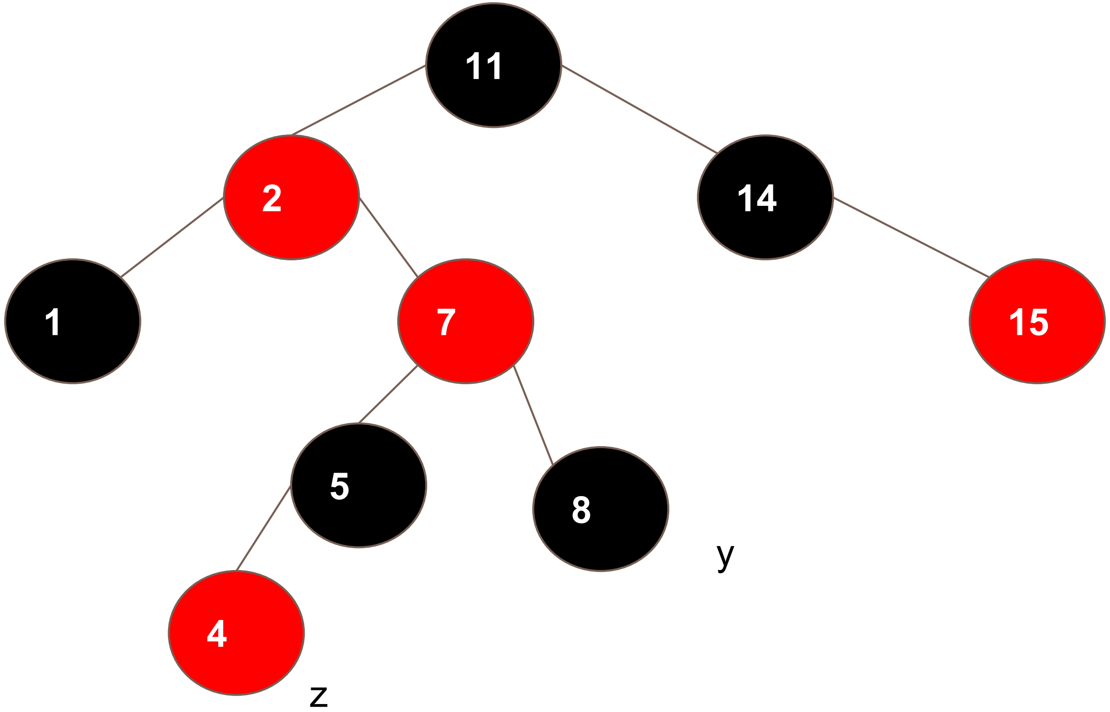


An example

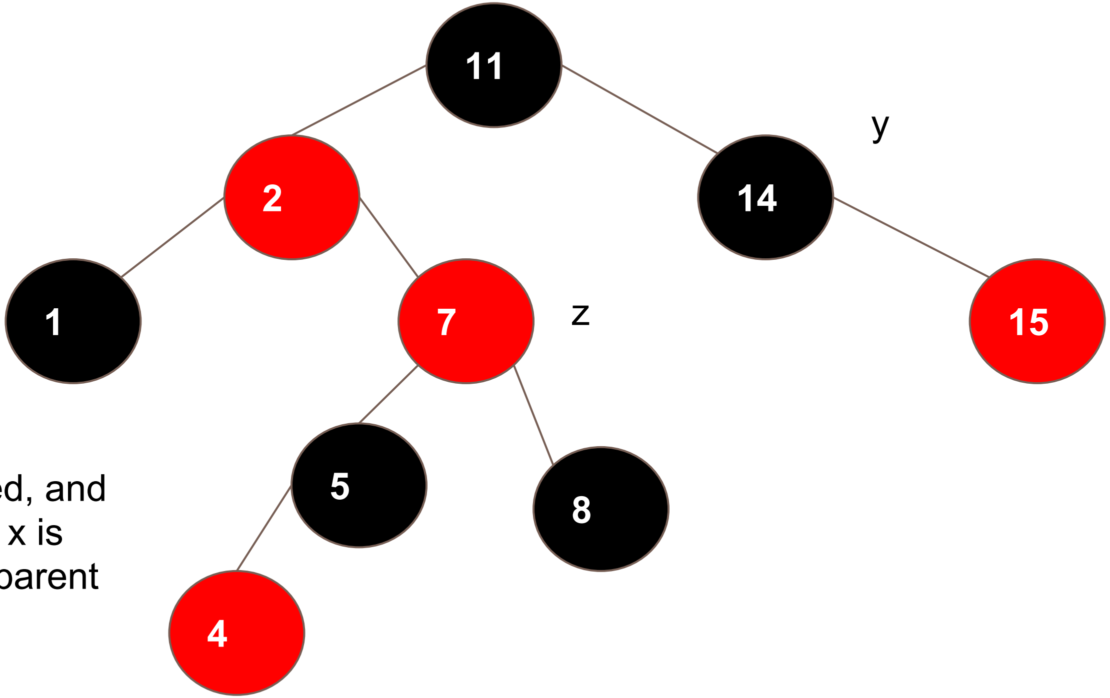


Parent of z is red, and
uncle y is red.
Case 1

An example



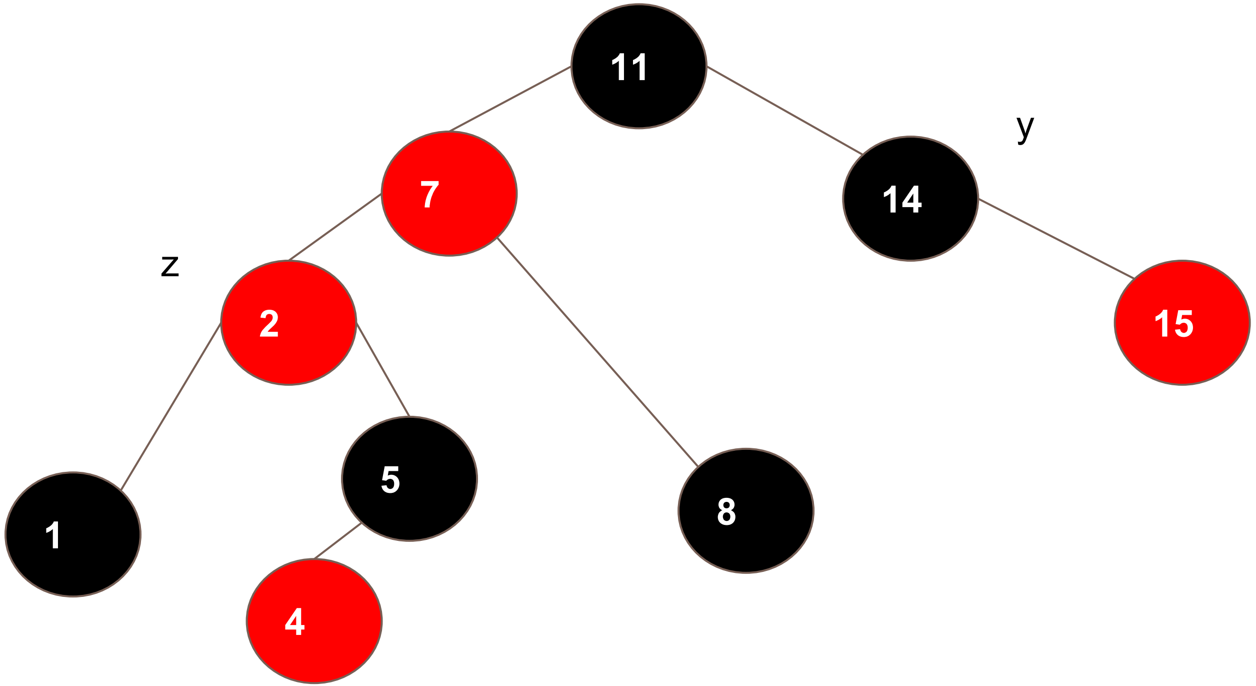
An example



The parent of z is red, and the uncle y is black. x is the right child of its parent so we left rotate the subtree at root 2.

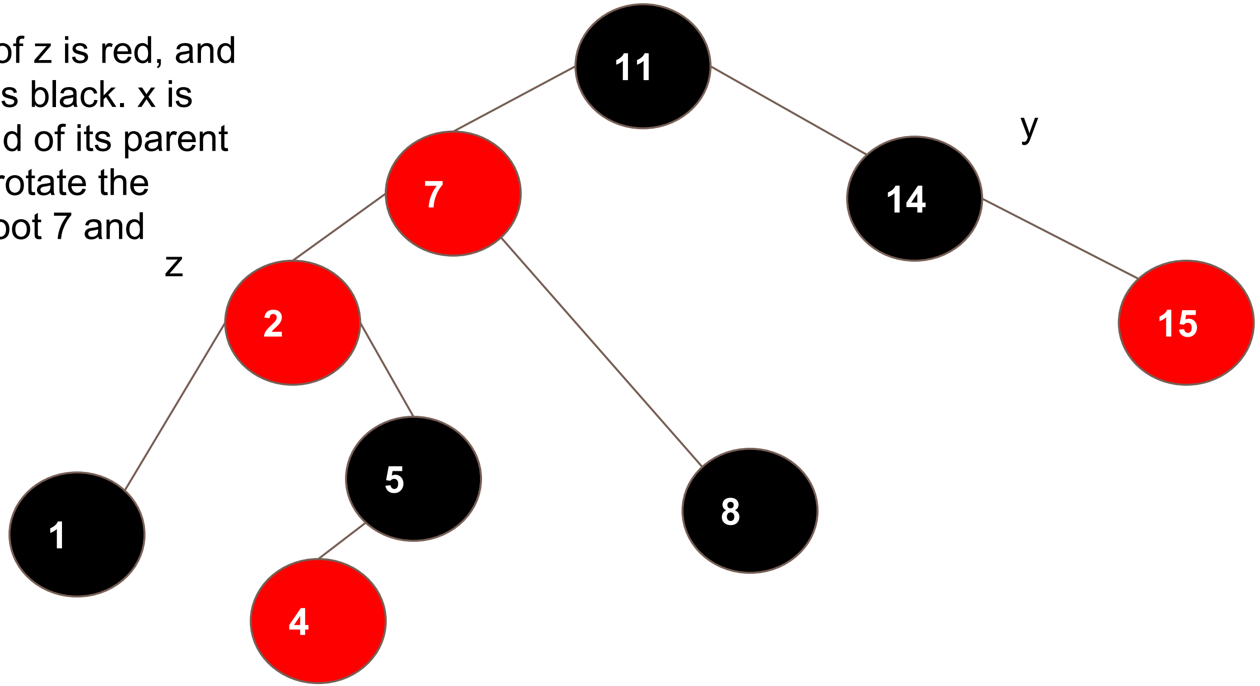
Case 2

An example



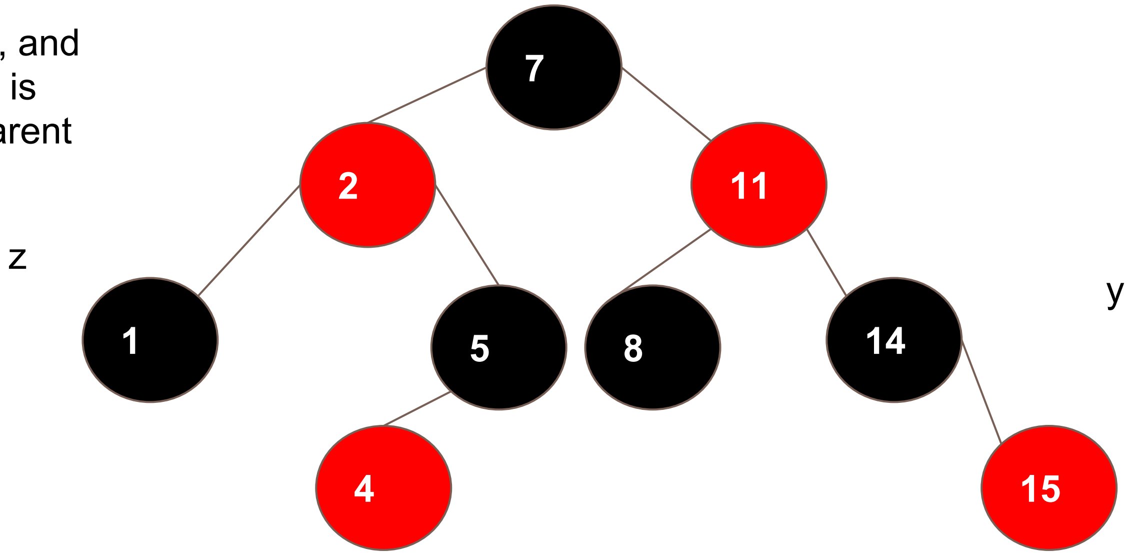
An example

The parent of z is red, and the uncle y is black. x is the right child of its parent so we right rotate the subtree at root 7 and **Case 3**



An example

The parent of z is red, and the uncle y is black. x is the right child of its parent so we right rotate the subtree at root 7



Pseudocode

```
Fix-Tree(T, z)
While z.p.colour == Red
  If z.p == z.p.p.left
          y = z.p.p.right
      If y.colour == red
          z.p.colour = black                 // Case 1
          y.colour = black;                  // Case 1
          z.p.p.colour = red                 // Case 1
          z = z.p.p                          // Case 1
      Else  if z == z.p.right                // Case 2
          z = z.p                            // Case 2
          LEFT-ROTATE(T,z)                   // Case 2
          Z.p.colour = black                 // Case 3
          Z.p.p.colour = red                 // Case 3
          RIGHT-ROTATE(T,z, p.p)             // Case 3
  else (same as then clause but with "right and
     "Left" exchanged)
```

Plenty of other trees in the forest

- Balanced Trees are a huge part of computer science
  - 2-3 Trees, AVL Trees, AA Trees
  - Tango Trees, Scapegoat Trees, Weight-Balanced Trees
  - B-Trees, B+Trees, Splay Trees

- Have slightly different properties but follow the core logic of RB trees
  - Splay Trees allow "recently" accessed items to be retrieved more efficiently at the cost of doing rotations on search/succ/pred
  - B-Trees are very shallow but wide, and can store multiple values per node
    - This is node to better align with the memory hierarchy in databases
  - AVL trees have slightly cheaper search but more expensive inserts

Next Class

- We’ll move on to another useful abstraction:
  - Priority Queues
  - Heaps

- These datastructures can also be implemented with trees :-)