CCCS 224
Binary Trees
(Java)

Outline

- Tree Stuff
  - Trees
  - Binary Trees
  - Implementation of a Binary Tree
- Tree Traversals – Depth First
  - Preorder
  - Inorder
  - Postorder
- Breadth First Tree Traversal
- Binary Search Trees

Tree Stuff

Trees:

- Another Abstract Data Type
- Data structure made of nodes and pointers
  - Much like a linked list
    - The difference between the two is how they are organized.
  - A linked list represents a linear structure
    - A predecessor/successor relationship between the nodes of the list
  - A **tree** represents a **hierarchical relationship** between the nodes (ancestral relationship)
    - A node in a tree can have several successors, which we refer to as children
    - A nodes predecessor would be its parent

Tree Stuff

Trees:

General Tree Information:

Top node in a tree is called the **root**

- the root node has no parent above it…cuz it's the root!

Every node in the tree can have "children" nodes

- Each child node can, in turn, be a parent to its children and so on

Nodes having no children are called **leaves**

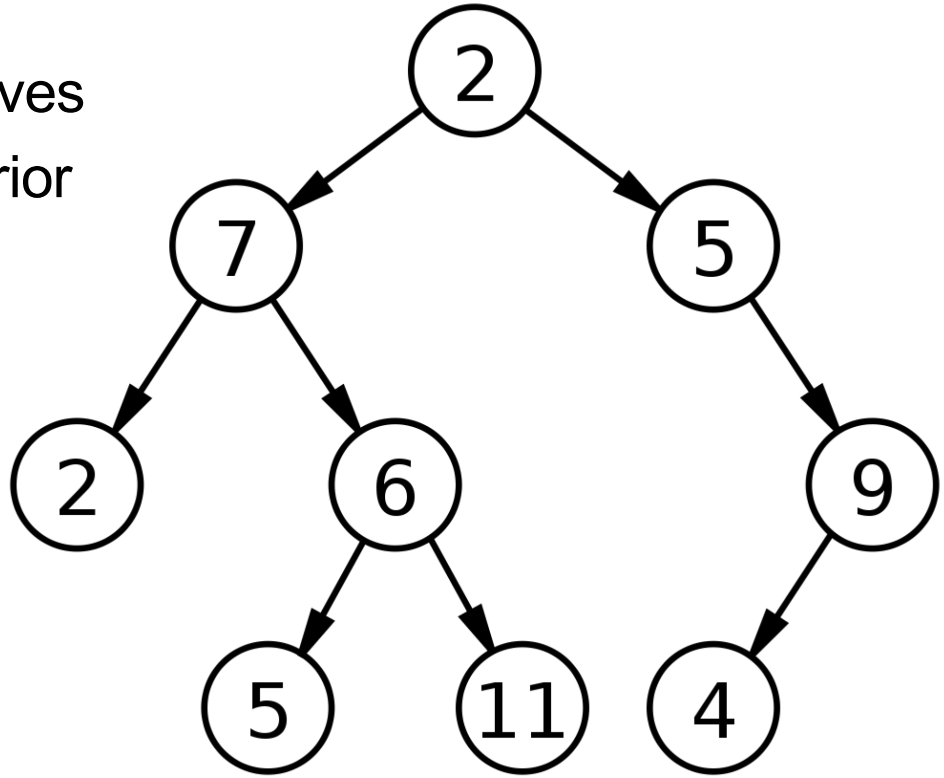
Any node that is not a root or a leaf is an **interior node**

The **height** of a tree is defined to be the length of the longest path from the root to a leaf in that tree.

- A tree with only one node (the root) has a height of zero.

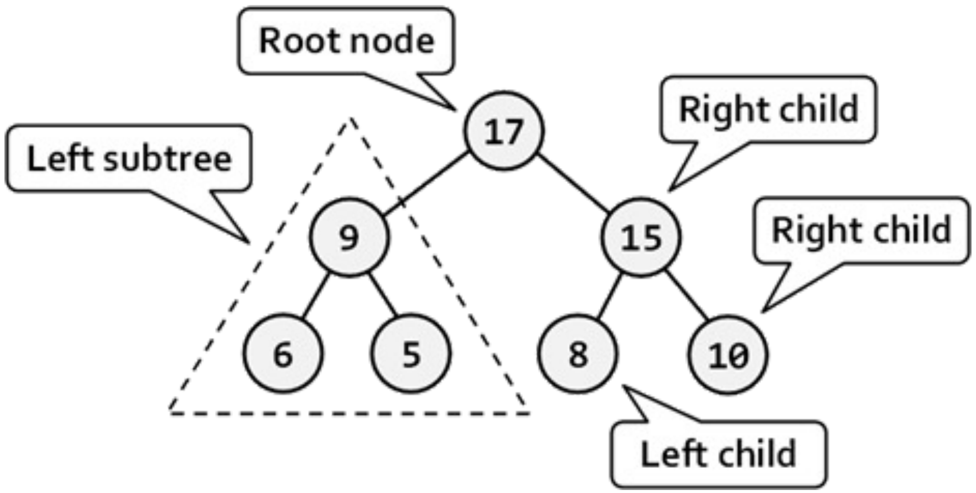
Tree Stuff

- Trees:
  - Here's a picture of a tree
    - 2 is the root
    - 2, 5, 11, and 4 are leaves
    - 7, 5, 6, and 9 are interior nodes



Tree Stuff

- Trees:
  - Subtree:
    - A subtree of a tree, T, is a tree consisting of a node, in T, and all of its children (descendants).



Tree Stuff

<u>Binary Trees</u>:

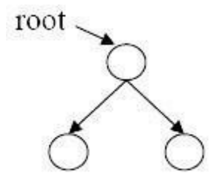
A tree in which each node can have a <u>maximum of two children</u>

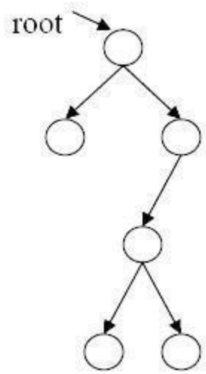
Each node can have no child, one child, or two children

And a child can only have one parent

Pointers help us to identify if it is a right child or a left are

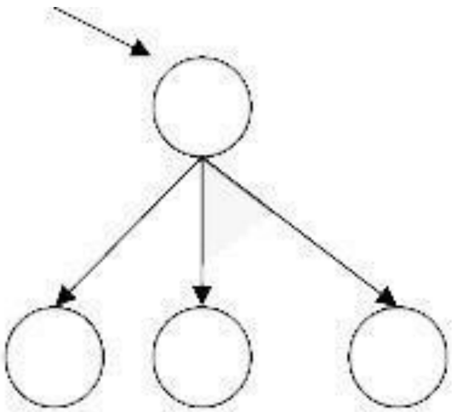
Examples of two Binary Trees:

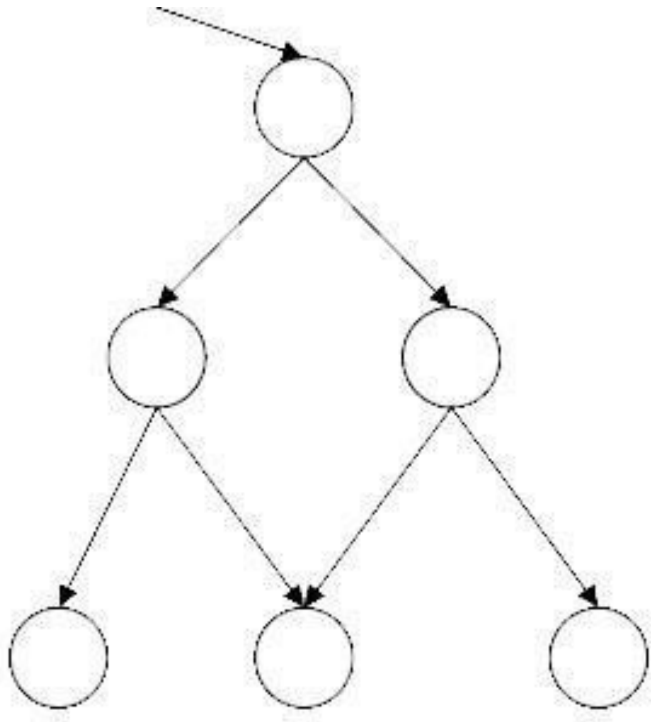




Tree Stuff

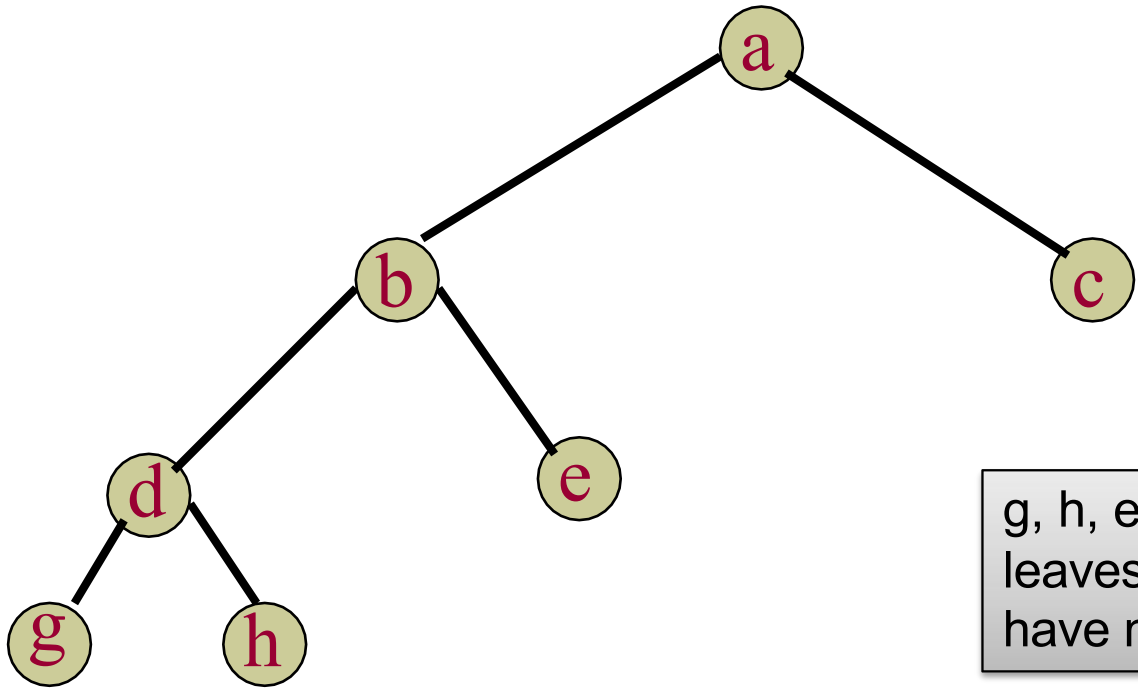
- Examples of trees that are NOT Binary Trees





Tree Stuff

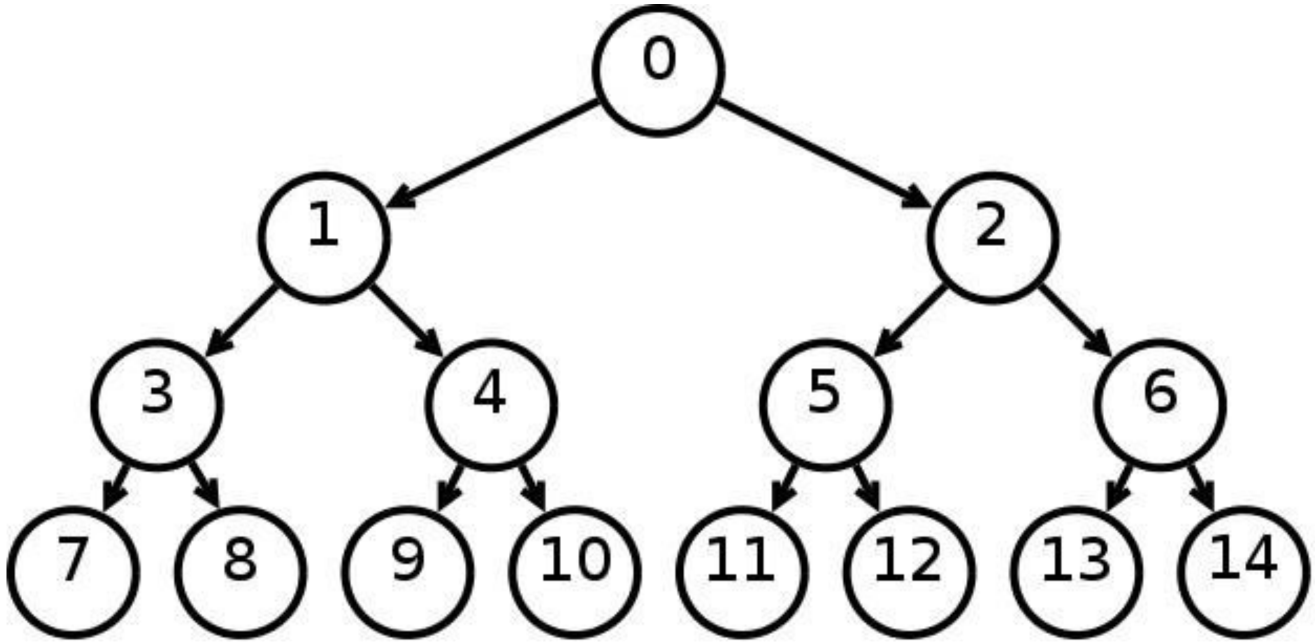
- More Binary Tree Goodies:
  - A **<u>full</u>** binary tree:
    - Every node, other than the leaves, has two children



g, h, e, and c are leaves: so they have no children.

Tree Stuff

- More Binary Tree Goodies:
  - A **complete** binary tree:
    - Every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



Tree Stuff

More Binary Tree Goodies:

The root of the tree is at level 0

The level of any other node in the tree is one more than the level of its parent

Total # of nodes (n) in a complete binary tree:

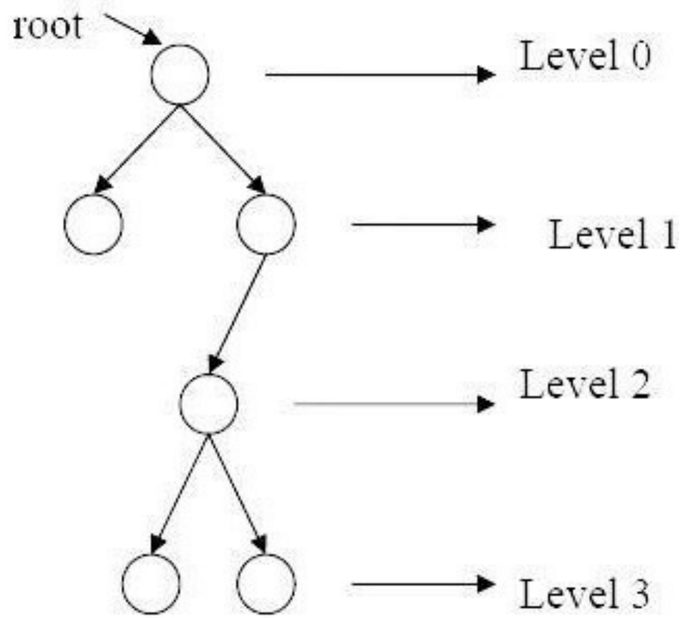
$n = 2^{h+1} - 1$ (maximum)

Height (h) of the tree:

- $h = \log_2(n + 1) - 1$

If we have 15 nodes

- $h = \log(16/2) = \log(8) = 3$



Tree Stuff

Implementation of a Binary Tree:

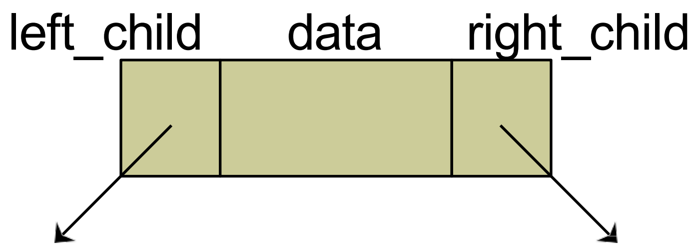
A binary tree has a natural implementation using linked storage

Each node of a binary tree has both left and right subtrees that can be reached with pointers:

```
class intBSTnode {

        private int data;
        private intBSTnode left, right;
    // Constructors go here
    ...

 }
```



Tree Traversals – Depth First

Traversal of Binary Trees:

We need a way of zipping through a tree for searching, inserting, etc.

But how can we do this?

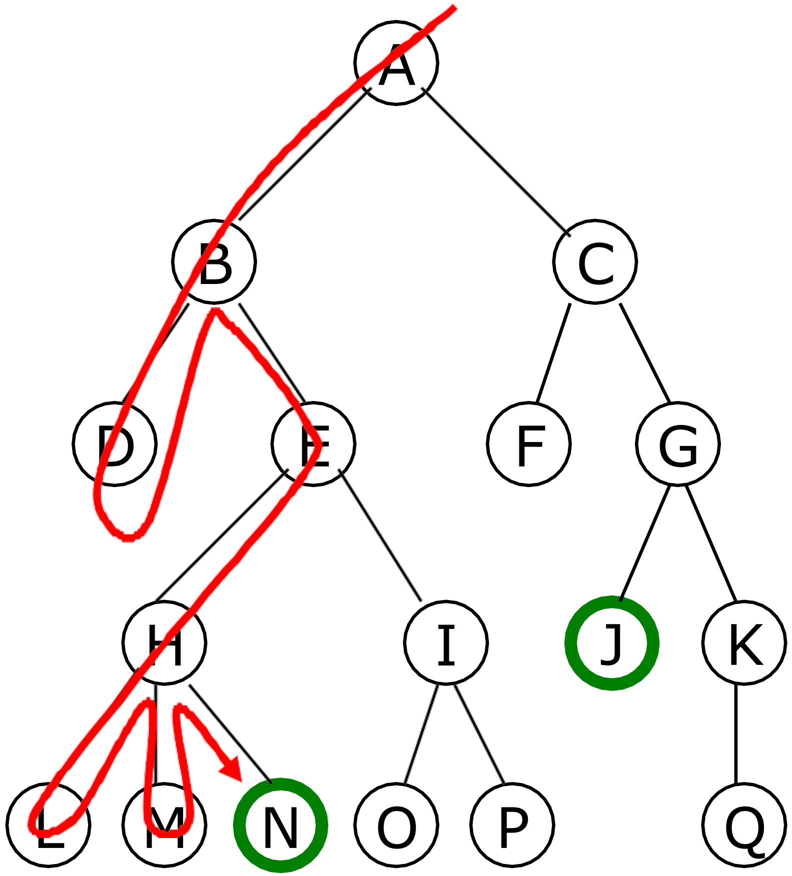
If you remember…

- Linked lists are traversed from the head to the last node …sequentially
- Can't we just "do that" for binary trees.?.
  - NO! There is no such natural linear ordering for nodes of a tree.

Turns out, there are THREE ways/orderings of traversing a binary tree:

Preorder, Inorder, and Postorder

Tree Traversals – Depth First



A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path

For example, after searching A, then B, then D, the search backtracks and tries another path from B

- Node are explored in the order A B D E H L M N I O P C F G J K Q
- N will be found before J

Tree Traversals – Depth First

Traversal of Binary Trees:

There are 3 ways/orderings of traversing a binary tree (all 3 are depth first search methods):

Preorder, Inorder, and Postorder

These names are chosen according to the step at which the root node is visited:

- With **preorder** traversal, the root is visited before its left and right subtrees.
- With **inorder** traversal, the root is visited between the subtrees.
- With **postorder** traversal, the root is visited after both subtrees.

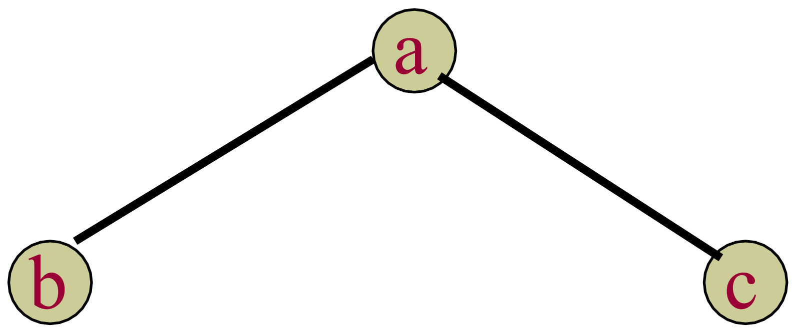
Tree Traversals - Preorder

- Preorder Traversal
  - the root is visited before its left and right subtrees
    - For the following example, the “**visiting**” of a node is
    - **represented by printing** that node
  - Code for Preorder Traversal:

```
void preorder (intBSTnode p) {
       if (p != null) {
           System.out.println(" " + p.data);
           preorder(p.left);
           preorder(p.right);
       }
}
```

Tree Traversals - Preorder

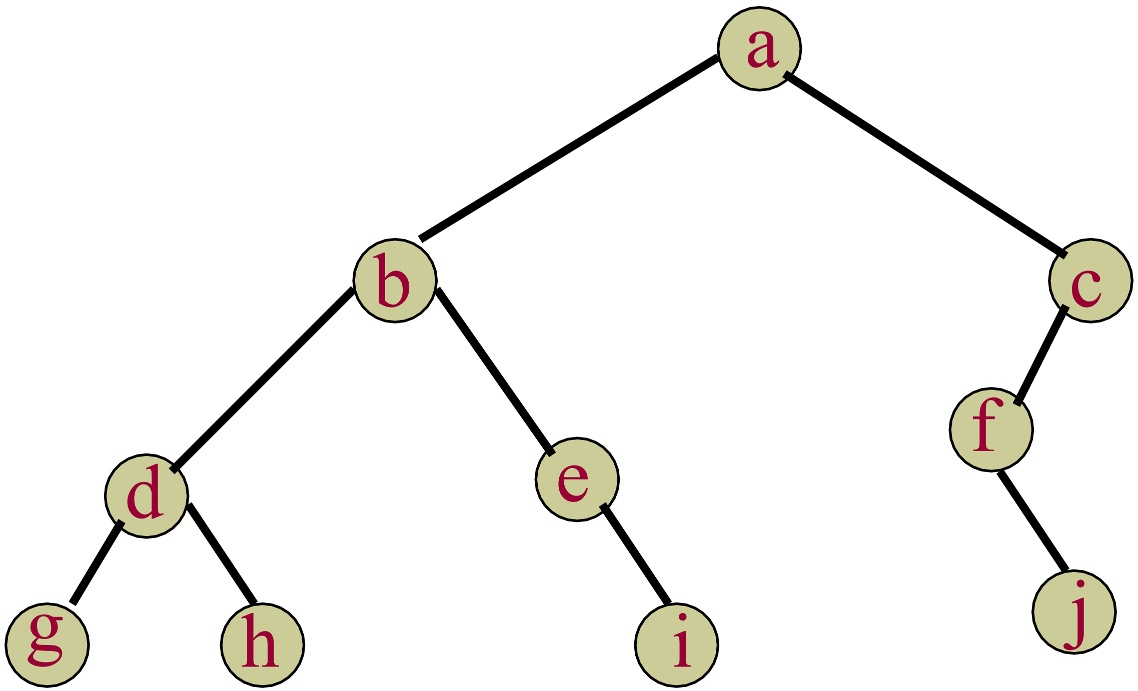
- Preorder Traversal – Example 1
  - the root is visited before its left and right subtrees



a b c

Tree Traversals - Preorder

- Preorder Traversal – Example 2



Order of Visiting Nodes: a b d g h e i c f j

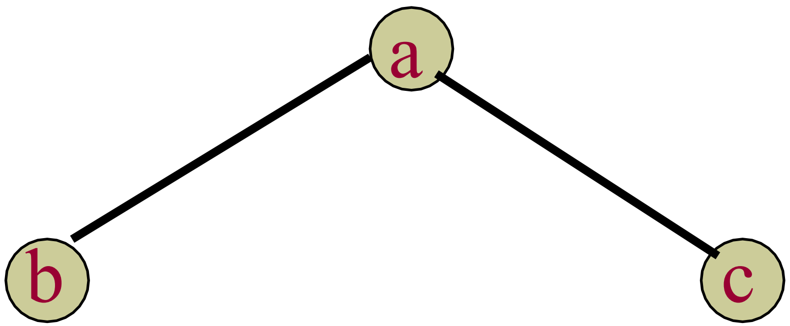
Tree Traversals - Inorder

- Inorder Traversal
  - the root is visited between the left and right subtrees
    - For the following example, the "**visiting**" of a node is
    - **represented by printing** that node
  - Code for Inorder Traversal:

```
void inorder (intBSTnode p) {
        if (p != null) {
                inorder(p.left);
                System.out.println(" " + p.data);
                inorder(p.right);
        }
}
```

Tree Traversals - Inorder

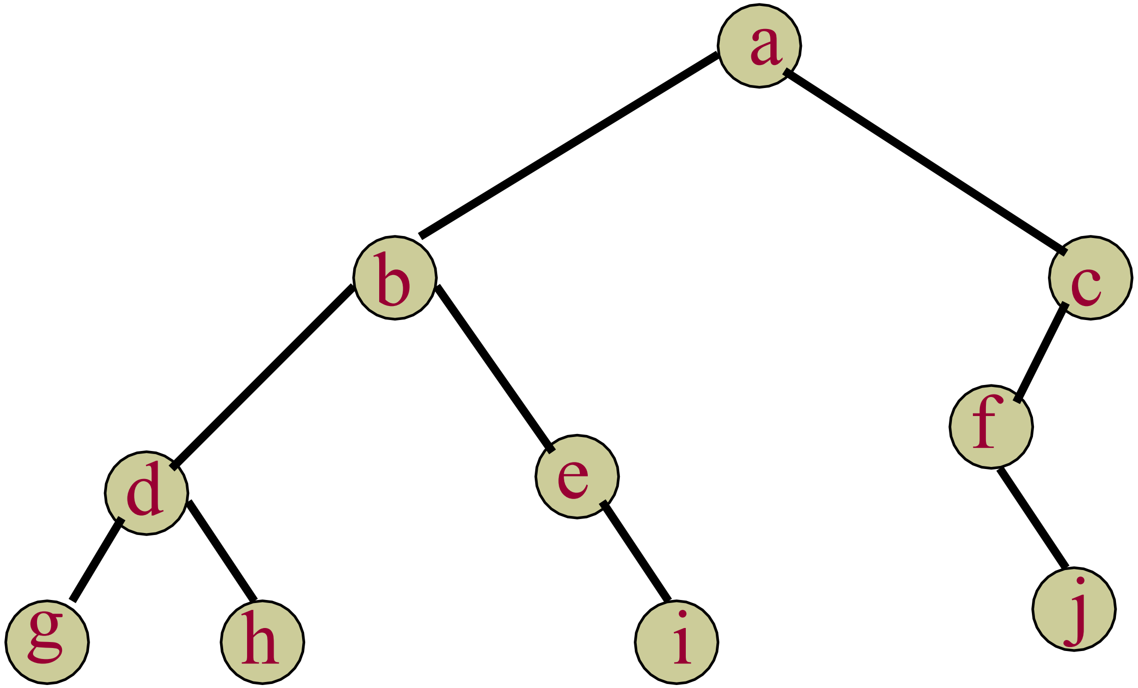
- Inorder Traversal – Example 1
  - the root is visited between the subtrees



b a c

Tree Traversals - Inorder

- Inorder Traversal – Example 2



Order of Visiting Nodes: g d h b e i a f j c

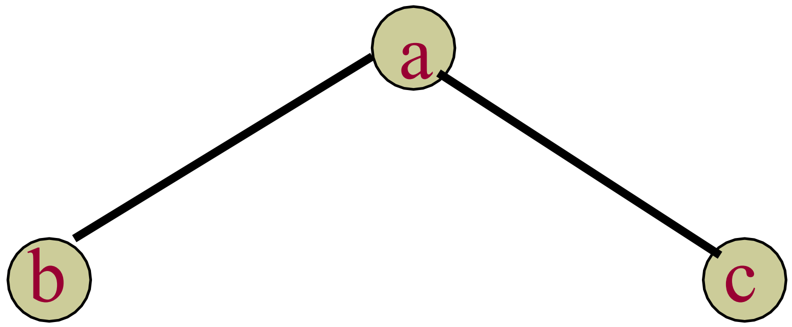
Tree Traversals – Postorder

- Postorder Traversal
  - the root is visited after both the left and right subtrees
    - For the following example, the "**visiting**" of a node is
    - **represented by printing** that node
  - Code for Postorder Traversal:

```
void postorder (intBSTnode p) {
       if (p != null) {
              postorder(p.left);
              postorder(p.right);
              System.out.println(" " + p.data);
       }
}
```

Tree Traversals – Postorder

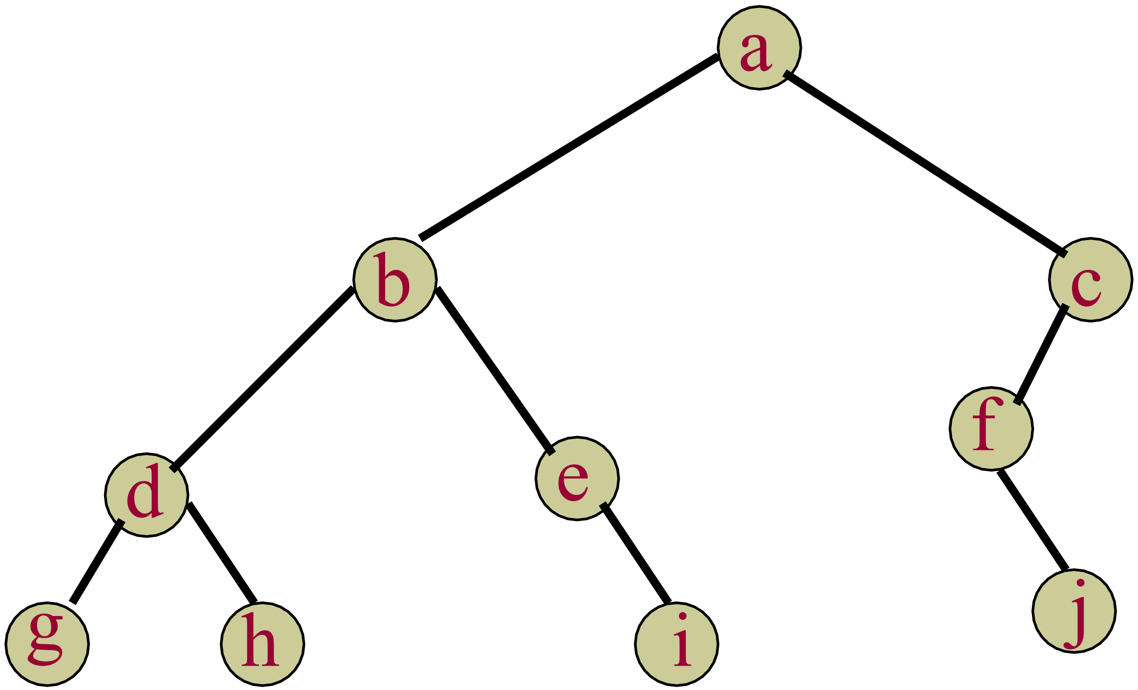
- Postorder Traversal – Example 1
  - the root is visited after both subtrees



b c a

Tree Traversals – Postorder

- Postorder Traversal – Example 2

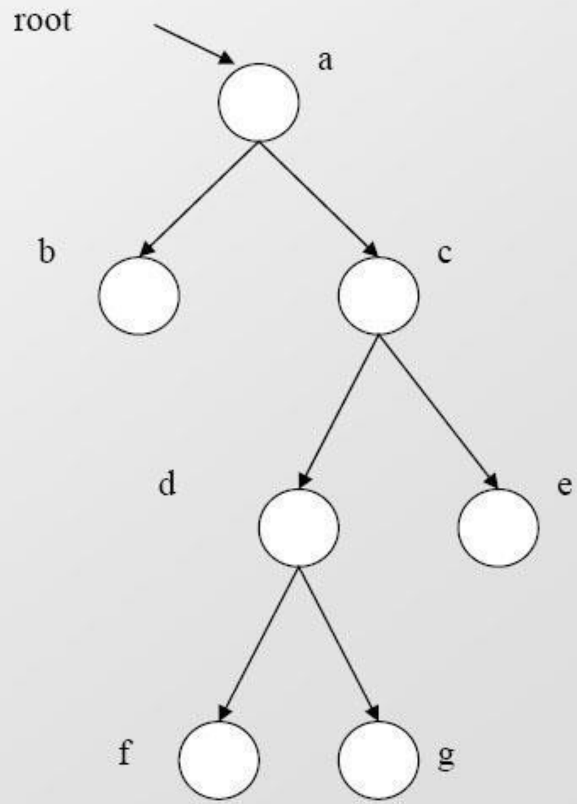


Order of Visiting Nodes: g h d i e b j f c a

Tree Traversals

- Final Traversal example

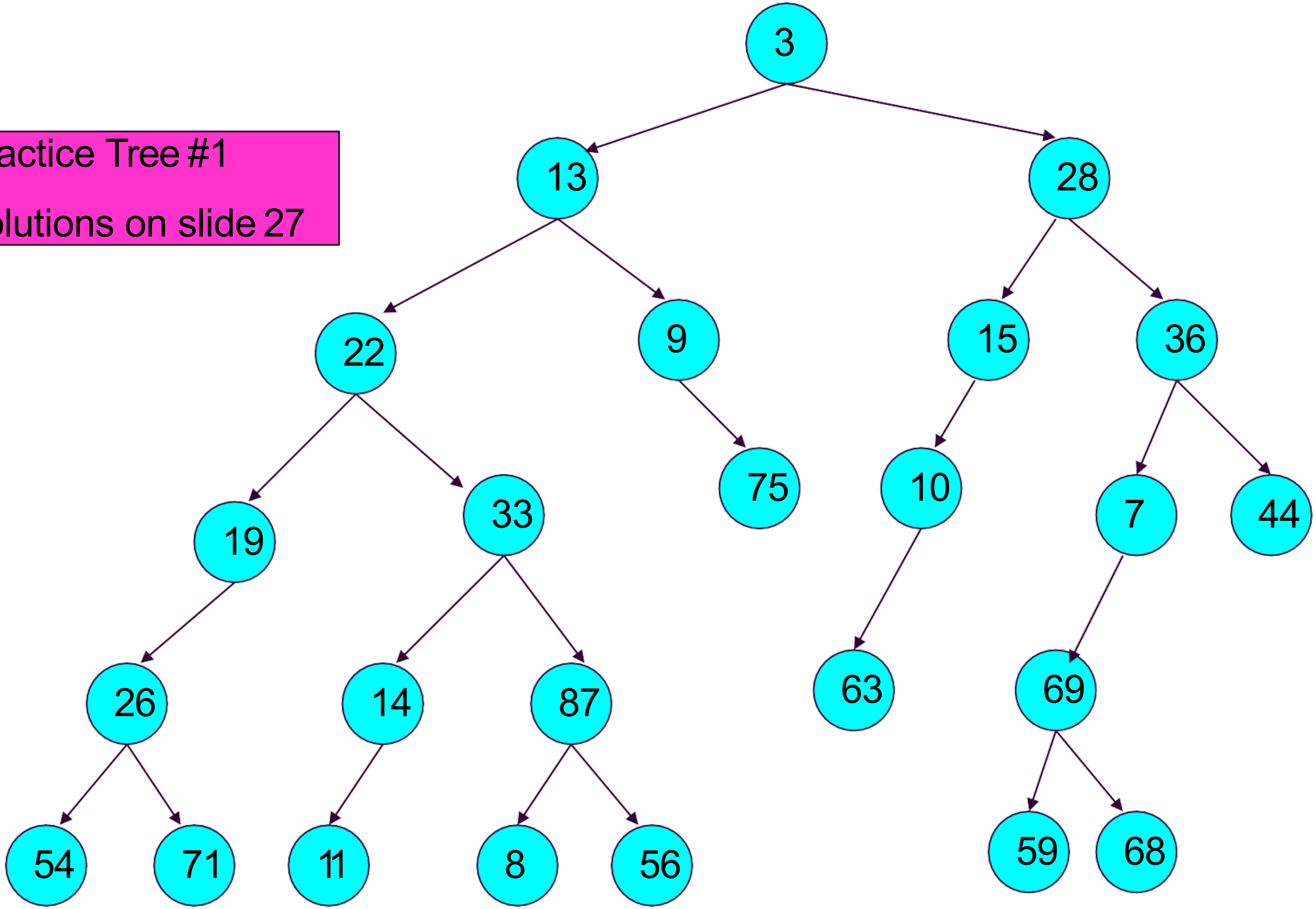
- Preorder: a b c d f g e
- Inorder: b a f d g c e
- Postorder: b f g d e c a



Binary Tree Traversals – Practice Problems

Practice Tree #1

Solutions on slide 27



Practice Problem Solutions – Tree #1

- Preorder Traversal:

3, 13, 22, 19, 26, 54, 71, 33, 14, 11, 87, 8, 56, 9, 75, 28, 15, 10, 63, 36, 7, 69, 59, 68, 44

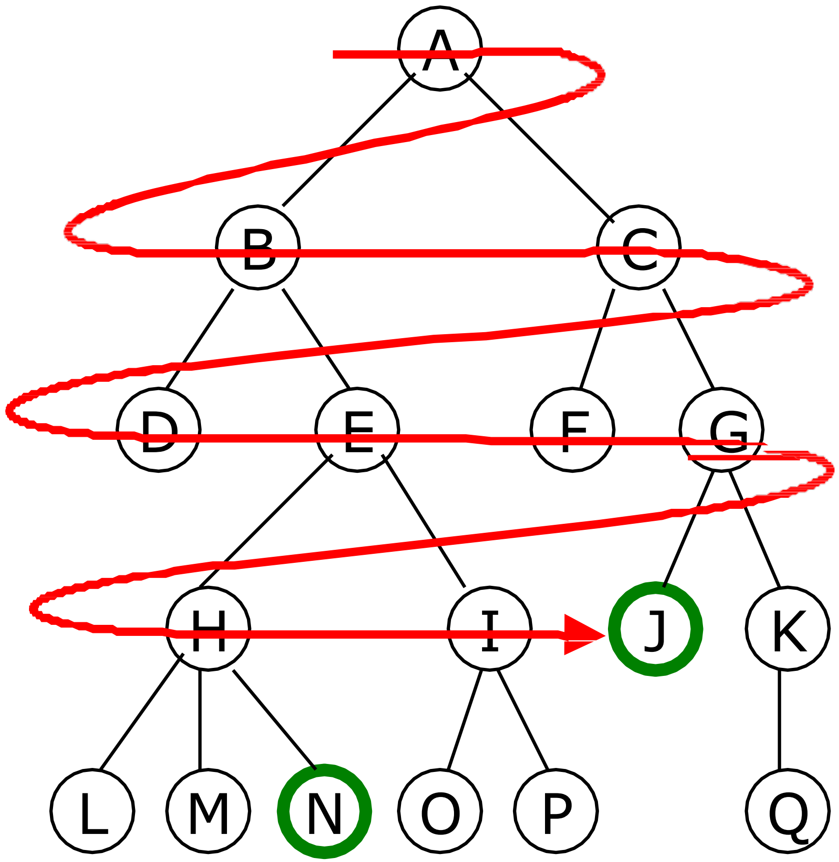
Inorder Traversal:

54, 26, 71, 19, 22, 11, 14, 33, 8, 87, 56, 13, 9, 75, 3, 63, 10, 15, 28, 59, 69, 68, 7, 36, 44

Postorder Traversal:

54, 71, 26, 19, 11, 14, 8, 56, 87, 33, 22, 75, 9, 13, 63, 10, 15, 59, 68, 69, 7, 44, 36, 28, 3

Breadth-First Traversal



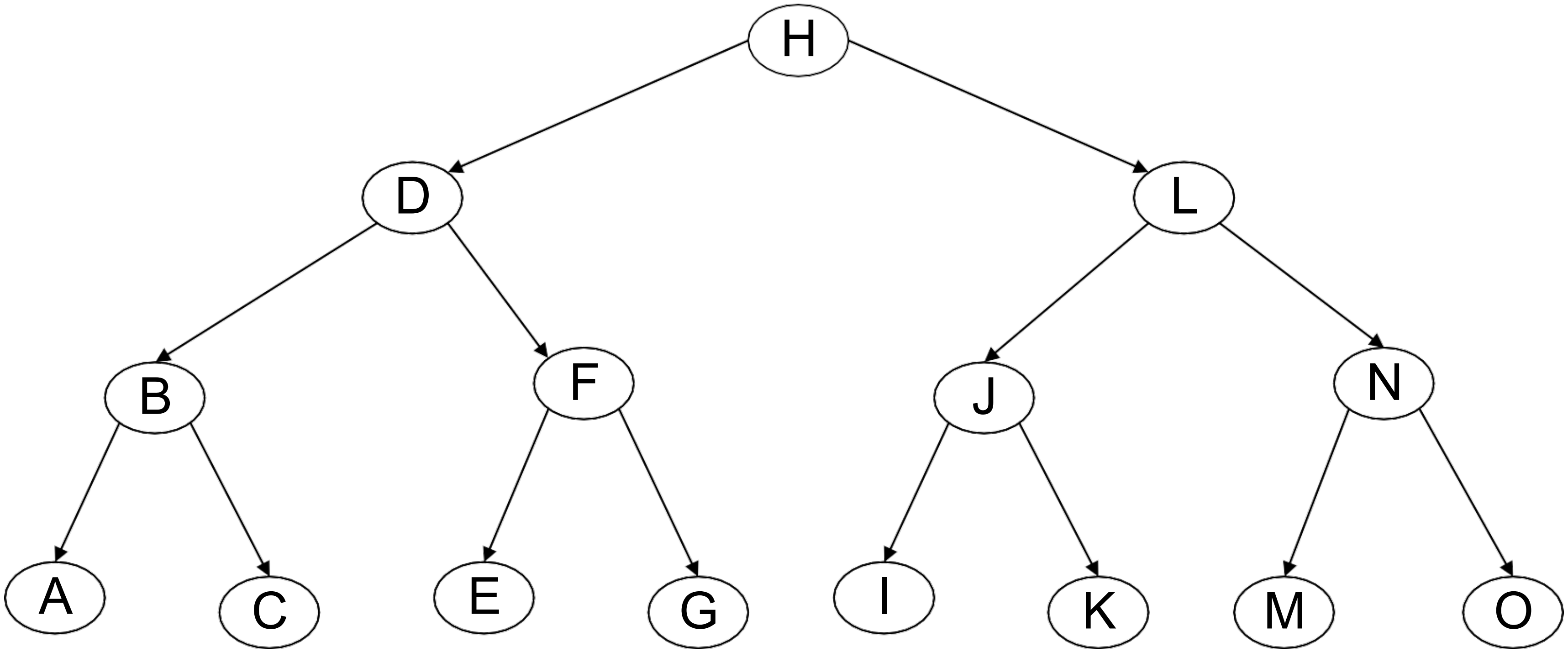
A **breadth-first** search (BFS) explores nodes **nearest** the root before exploring nodes further away

For example, after Searching A, then B, then C, the search proceeds with D, E, F, G

Node are explored in the order A B C D E F G H I J K L M N O P Q

J will be found before N

Breadth-First Traversal



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

Breadth-First Traversal

Coding the Breadth-First Traversal

Let's say you want to Traverse and Print all nodes?

Think about it, how would you make this happen?

SOLUTION:

1) Enqueue the root node.
2) **while** (more nodes still in queue){

    Dequeue node at front (of queue)

    Print this node (that we just dequeued)

    Enqueue its children (if applicable): **left then right**

    …continue till no more nodes in queue

    }

Breadth-First Traversal-**Algorithm**

Algorithm:
For each node, first the node is visited and then its child nodes are put in a FIFO queue.

```
procedure BFS(G, root) is
let Q be a queue
label root as explored
Q.enqueue(root)
while Q is not empty do
v := Q.dequeue()
if v is the goal then
return v
for all edges from v to w in G.adjacentEdges(v) do
if w is not labeled as explored then
label w as explored
w.parent := v
Q.enqueue(w)
```

Breadth-First Traversal- implementation

```
public static void traverseBinaryTree(Node root) {
    if (root == null) {
        System.out.println("Tree is empty");
        return;
    }
    Queue queue = new Queue();
    queue.push(root);

    while (!queue.isEmpty()) {
        Node node = queue.pop();
        System.out.println(node.data);
        if (node.left != null) {
            queue.push(node.left);
        }
        if (node.right != null) {
            queue.push(node.right);
        }
    }
}
```

Search & Insert

Binary Search Tree

Binary Search Trees

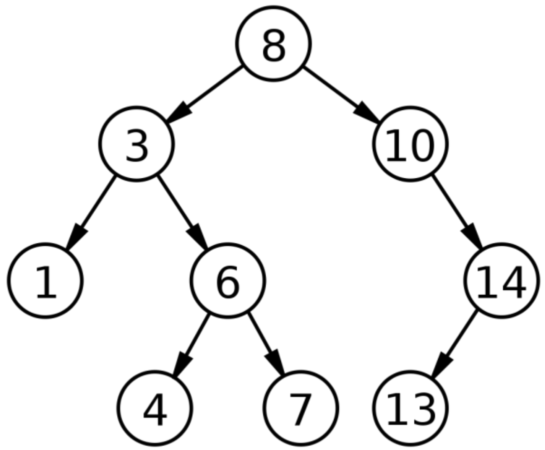
We've seen how to traverse binary trees

But it is not quite clear how this data structure helps us

- **What is the purpose of binary trees?**

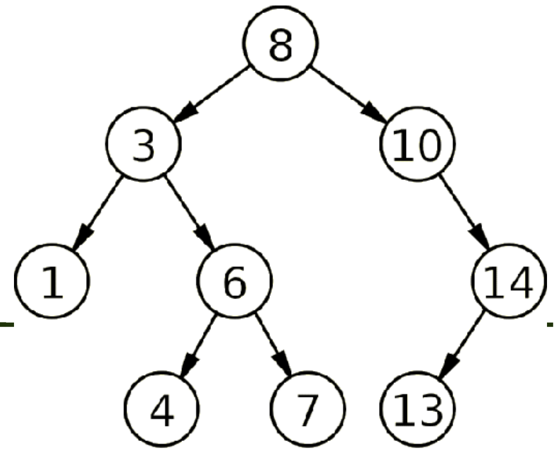
What if we added a restriction…

Consider the following binary tree:



What pattern can you see?

Binary Search Tree



Binary Search Trees

What pattern can you see?

For each node N, all the values stored in the left subtree of N are LESS than the value stored in N.

Also, all the values stored in the right subtree of N are GREATER than the value stored in N.

Why might this property be a desireable one?

- **Searching for a node is super fast!**

Normally, if we search through n nodes, it takes O(n) time

But notice what is going on here:

- This **ordering property** of the tree **tells us where to search**
- We choose to look to the left **OR** look to the right of a node
- We are **HALVING** the search space …**O(log n)** time

Binary Search Tree

Binary Search Trees

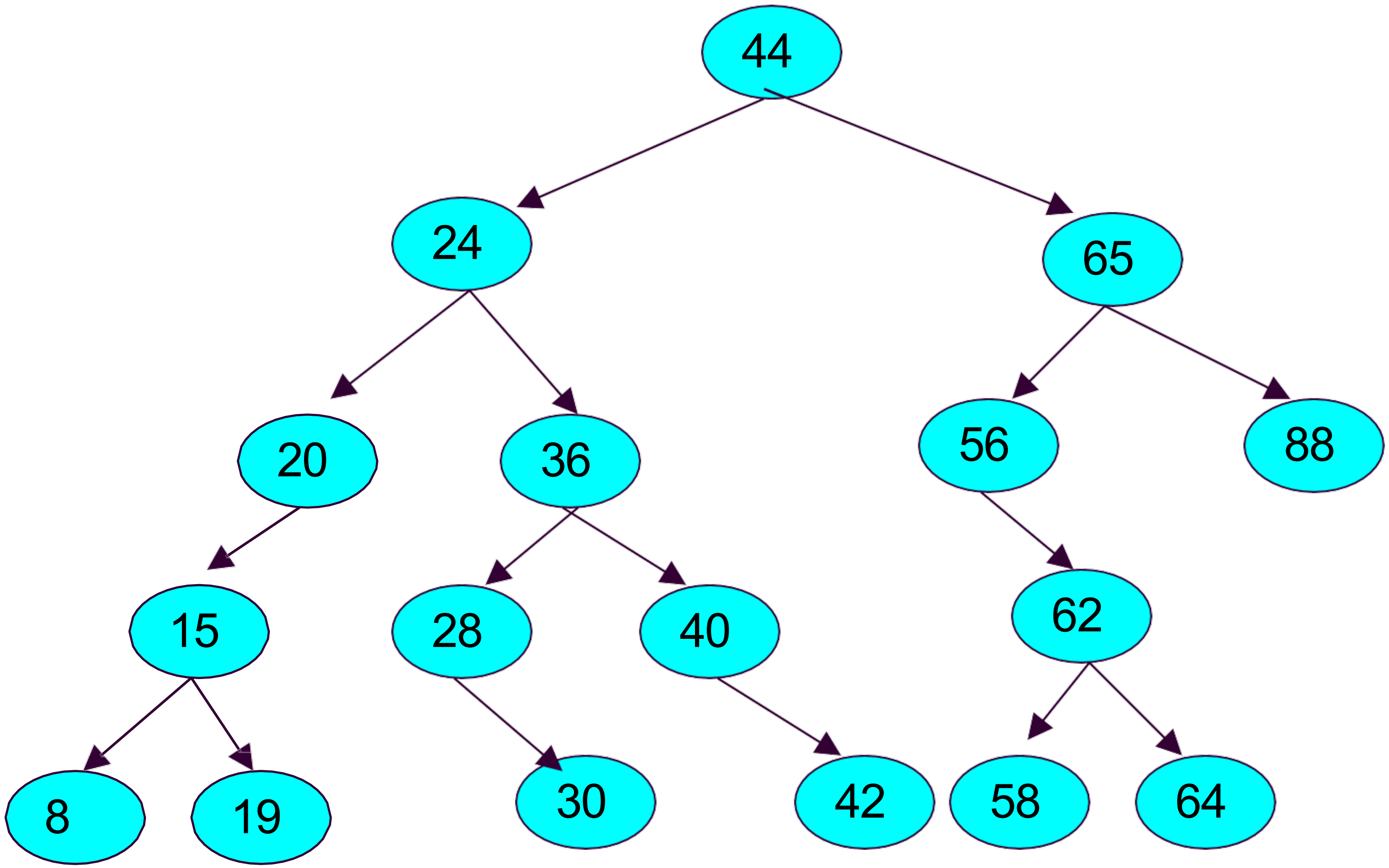
Details:

ALL of the data values in the left subtree of each node are **smaller** than the data value in the node itself (root of said subtree)

ALL of the data values in the right subtree of each node are **larger** than the data value in the node itself (root of the subtree)

Both the left and right subtrees, of any given node are themselves binary search trees.

Binary Search Tree



A Binary Search Tree

Binary Search Tree: Searching

- Binary Search Trees
  - Searching for a node:
    - Algorithm:

1) **IF** the tree is NULL, return false.
   ELSE
2) Check the root node. If the value we are searching for is in the root, return 1 (representing "found").
3) If not, if the value is less than that stored in the root node, recursively search in the left subtree.
4) Otherwise, recursively search in the right subtree.

Binary Search Tree: Searching

- Here is the **`search`** method:

```
public intBSTnode search(intBSTnode p, int key) {
   while (p != null) {
      if (key == p.data)        // the current node has the key value
         return p;              // we found it!  Return this node.
      else if (key < p.data)    // the key is less than current node
         p = P.left;            // search to the left side
      else                      // key is greater than current node
         p = P.right;           // so search to the right side
   }
   return null;   // this "return null" only happens when the value
                  // is NOT found (and returned) within the above
                  // while loop.  In that case, we return null,
                  // indicating that the "key" was not found.
}
```

Binary Search Tree: Searching

- Searching for a node (using **Recursion**):

```
boolean recursiveSearch(int data) {
   return(recursiveSearch(root, data));
}

boolean recursiveSearch(intBSTnode p, int data) {
   if (p == null) {    // meaning, there is no node!
      return(false);
   }

   if (data == p.data) {
      return(true);
   }
   else if (data < p.data) {
      return(recursiveSearch(p.left, data));
   }
   else {
      return(recursiveSearch(p.right, data));
   }
}
```

Binary Search Tree: Search

Search of an **<u>Arbitrary</u>** Binary Tree

- We've seen how to search for a node in a binary search tree
- Now consider the problem if the tree is NOT a binary search tree
  - It <u>does not have the ordering property</u>
- You could simply perform one of the traversal methods, checking each node in the process
  - Unfortunately, this won't be O(log n) anymore
  - It degenerates to O(n) since we possibly check all nodes
- The following slide shows another way to do this

Binary Search Tree: Searching

- Search of an **Arbitrary** Binary Tree:
  - The whole purpose here is to be comfortable with trees and binary search trees

```
boolean searchTree(int data) {
   return(searchTree(root, data));
}

boolean searchTree(intBSTnode p, int data) {
   if (p == null) {    // meaning, there is no node!
      return(false);
   }

   if (data == p.data){
      return(true);
   }

   return (searchTree(p.left, data) || searchTree(p.right, data));
}
```

Binary Search Tree: Creation

Insertion into a Binary Search Tree

Before we can <u>insert</u> a <u>**node**</u> into a BST, what is the one obvious thing that we must do?

We have to actually <u>**create**</u> the node that we want to insert

- And save appropriate data value(s) into it

Binary Search Tree: Creation

- Here's the node class with constructors:

```
class intBSTnode {
        private int data;
        private intBSTnode left, right;

   // Constructors
   intBSTnode() {
      left = null;
      right=null;
   }
   intBSTnode(int newData) {
      this(newData, null, null);
   }
   intBSTnode(int newData, intBSTnode lt, intBSTnode rt) {
      data = newData;
      left = lt;
      right = rt;
   }
}
```

Binary Search Tree: Creation

Creating a Binary Search Tree

- We need to make a class for the actual tree
- The name of this class will be **`intBST`**
  - "*Integer* Binary Search Tree"
  - This class will have one **`intBSTnode`** variable
    - It will be called "**`root`**"
    - Why?
      - Because it will be a reference to the root of the tree!
  - From a main method, we can then make a new integer Binary Search Tree (a new **`intBST`**)
  - Finally, within this **`intBST`** class, we will have many methods to insert nodes, delete nodes, traverse the tree, count the nodes, sum the nodes, etc.

Binary Search Tree: Creation

- Creating a Binary Search tree
  - Here is the **`intBST`** class:

```
class intBST {

        private intBSTnode root;// this is the root of the tree

    // Constructor    ...just makes an empty (null) root for tree
    intBST() {
        root = null;
    }

    // Method to SEARCH for a node
    intBSTnode search(intBSTnode p, int key) {
        // see next page
    }

    // Other methods here

}
```

Binary Search Tree: Insertion

Insertion (of nodes) into a Binary Search Tree

Now that we have nodes, it is <u>**time to insert**</u>!

BSTs <u>must maintain</u> their <u>ordering property</u>

<u>Smaller items</u> to the <u>left of any given root</u>

And <u>greater items</u> to the <u>right of that root</u>

So when we insert, we MUST follow these rules

You simply start at the root and either

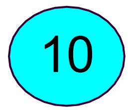
1) Go right if the new value is greater than the root
2) Go left if the new value is less than the root

<u>Keep doing this</u> till you come to an <u>empty position</u>

An example will make this clear…

Binary Search Tree: Insertion

- Insertion into a Binary Search Tree
  - Let's assume we insert the following data values, in their order of appearance into an initially empty BST:
    - 10, 14, 6, 2, 5, 15, and 17



Step 1:

Create a new node with value 10

Insert node into tree

The tree is currently empty

New node becomes the root

Binary Search Tree: Insertion

- Insertion into a Binary Search Tree
- 10, 14, 6, 2, 5, 15, and 17

Step 2:

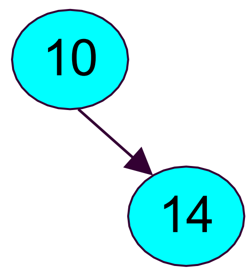
Create a new node with value 14

This node belongs in the right subtree of node 10

Since 14 > 10

The right subtree of node 10 is empty

So node 14 becomes the right child of node 10



Binary Search Tree: Insertion

- Insertion into a Binary Search Tree
  - 10, 14, 6, 2, 5, 15, and 17

Step 3:

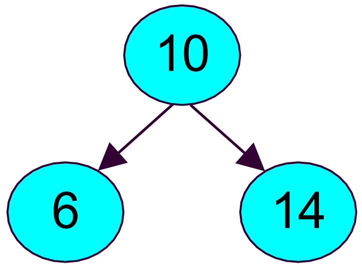
Create a new node with value 6

This node belongs in the left subtree of node 10

Since 6 < 10

The left subtree of node 10 is empty

So node 6 becomes the left child of node 10



Binary Search Tree: Insertion

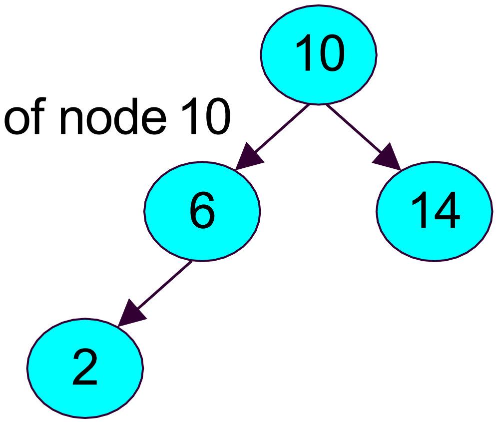
- Insertion into a Binary Search Tree
  - 10, 14, 6, 2, 5, 15, and 17

Step 4:

Create a new node with value 2

This node belongs in the left subtree of node 10

- Since 2 < 10



The root of the left subtree is 6

The new node belongs in the left subtree of node 6

- Since 2 < 6

So node 2 becomes the left child of node 6

Binary Search Tree: Insertion

- Insertion into a Binary Search Tree
  - 10, 14, 6, 2, 5, 15, and 17

Step 5:

Create a new node with value 5

This node belongs in the left subtree of node 10

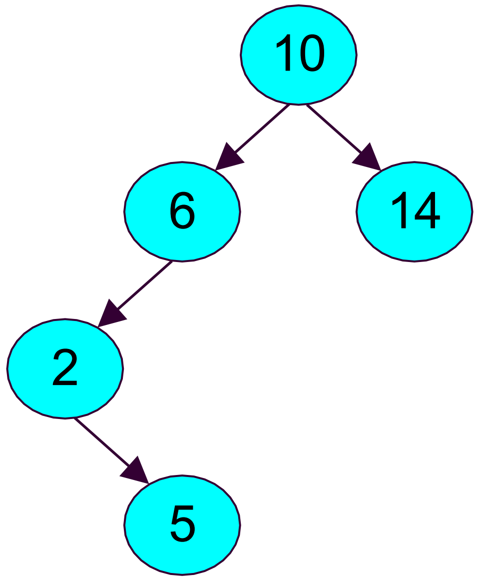
- Since 5 < 10

The new node belongs in the left subtree of node 6

Since 5 < 6

And the new node belongs in the right subtree of node 2

Since 5 > 2



Binary Search Tree: Insertion

- Insertion into a Binary Search Tree
  - 10, 14, 6, 2, 5, 15, and 17

Step 6:

Create a new node with value 15

This node belongs in the right subtree of node 10

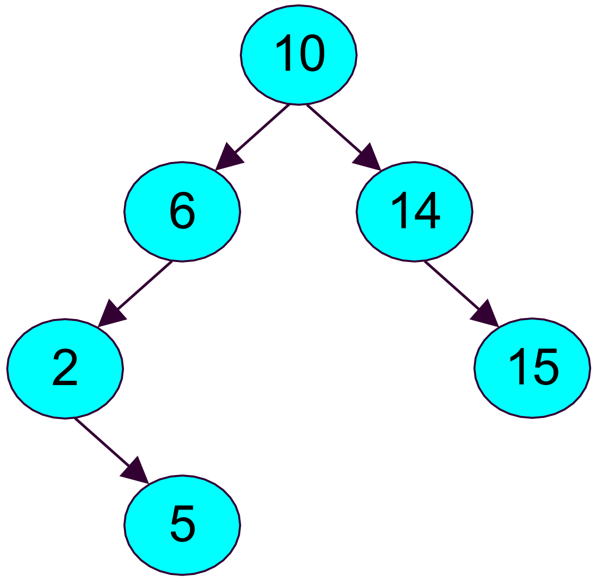
- Since 15 > 10

The new node belongs in the right subtree of node 14

- Since 15 > 14

The right subtree of node 14 is empty

So node 15 becomes right child of node 14



Binary Search Tree: Insertion

- Insertion into a Binary Search Tree
  - 10, 14, 6, 2, 5, 15, and 17

Step 7:

Create a new node with value 17

This node belongs in the right subtree of node 10

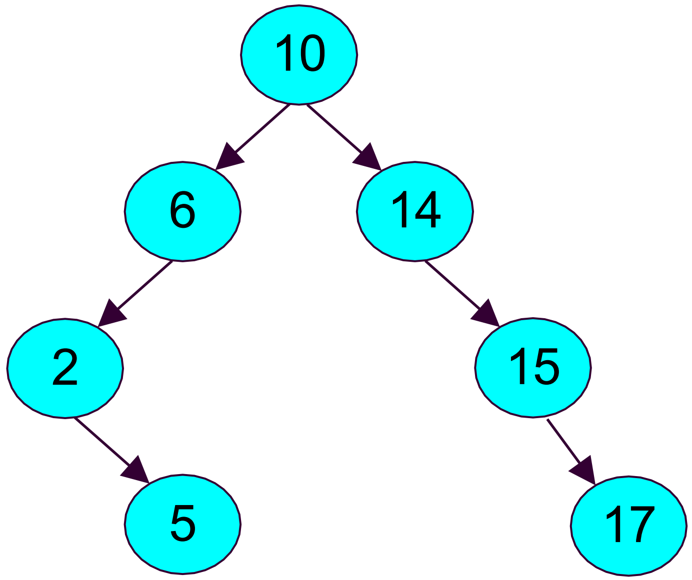
- Since 17 > 10

The new node belongs in the right subtree of node 14

Since 17 > 14

And the new node belongs in the right subtree of node 15

Since 17 > 15



Binary Search Tree: Insertion

Insertion into a Binary Search Tree

Here's our basic plan to do this recursively:

1) If the tree is empty, just return a pointer to a node containing the new value
   - cuz this value WILL be the ROOT
2) Otherwise, see which subtree the node should be inserted into
   - How?
   - Compare the value to insert with the value stored at the root.
3) Based on this comparison, **<u>recursively</u> either insert into the right subtree, or into the left subtree.**

Binary Search Tree: Insertion

- Insertion into a Binary Search Tree

```
void insert(int data) {
   root = insert(root, data);
}

intBSTnode insert(intBSTnode p, int data) {
   if (p == null) {
      p = new intBSTnode(data);
   }
   else {
      if p.data < p.data) {
         p.left = insert(p.left, data);
      }
      else {
         p.right = insert(p.right, data);
      }
   }
   return p; // in any case, return the new pointer to the caller
}
```

Binary Search Tree: Insertion

- Insertion into a Binary Search Tree
  - Here is a sample main method:

```
Public stati void main(String arg[]) throws IOException {
   intBST myBST = new intBST();

   myBST.insert(20);
   myBST.insert(18);
   myBST.insert(33);
}
```

Deletion

Binary Trees: Deletion

Deletion From a Binary Search Tree

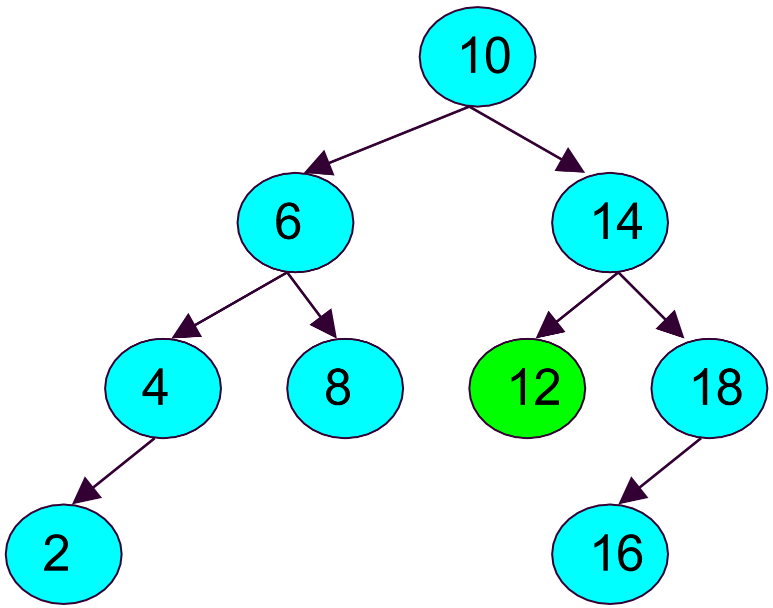
There are 3 possible cases:

1) Deleting of a leaf node
2) Deleting a node with one child
3) Deleting a node with two children

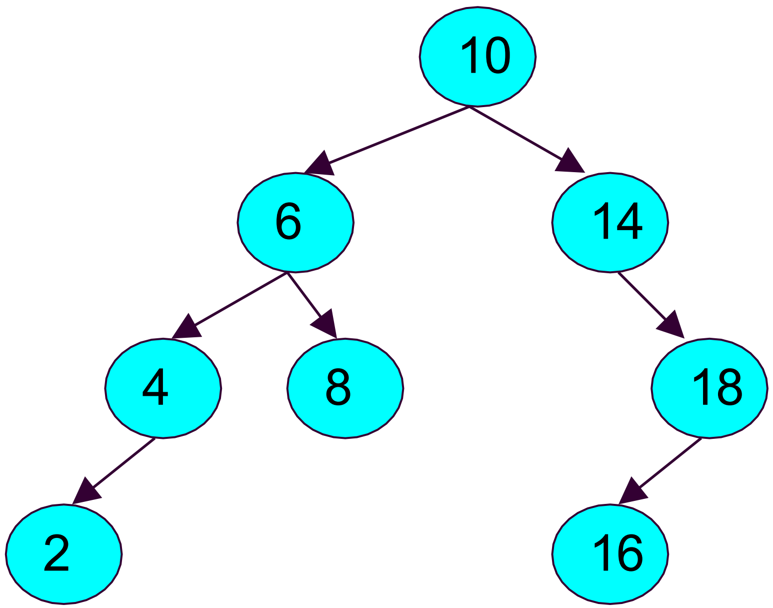
- We examine each case separately

Binary Trees: Deletion

- Case 1: Deleting a Leaf Node
  - This one is pretty easy



Initial BST with node 12 marked for deletion



BST after deletion of node 12

Binary Trees: Deletion

Case1: Deleting a Leaf Node

We start by identifying the parent of the node we wish to delete **Which we actually do in ALL three cases**

Just set the appropriate node to NULL:

`Parent.left = null;` or

`Parent.right = null;`

So now instead of pointing to the toBeDeleted node

The parent simply points to `null`

this signifies that the parent no longer has that child

The garbage collector then deletes the node from memory

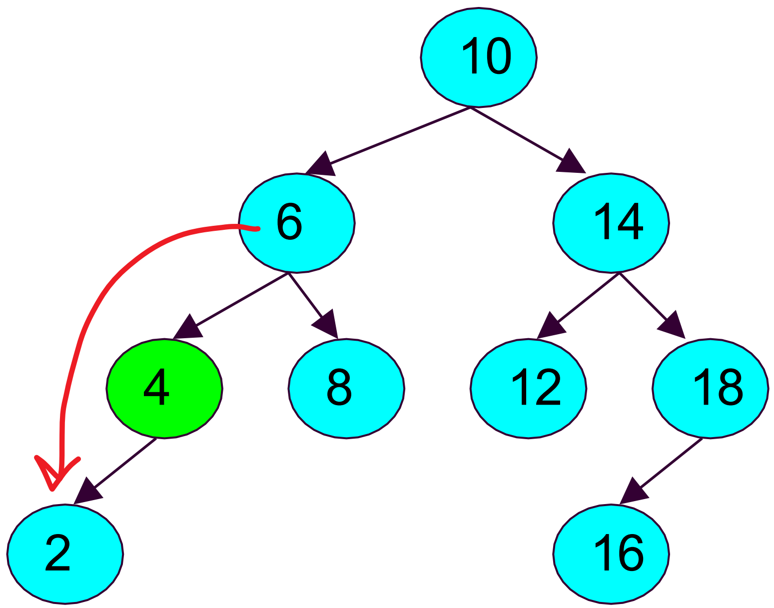
Binary Trees: Deletion

Case2: Deleting a Node with One Child

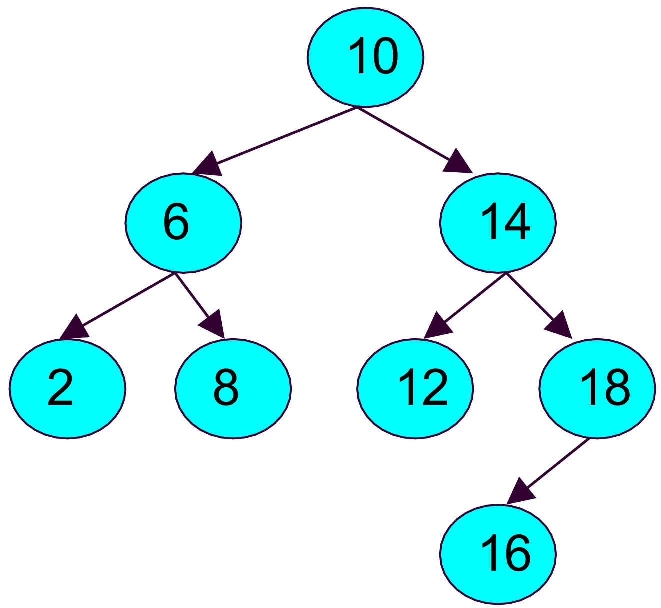
- Again, we start by finding the parent of the node we want to delete
- The parent’s pointer to the node is changed to now point to the deleted node’s child
- This has the effect of lifting up the deleted node’s child by one level in the tree
- Notice that it makes no difference whether the only child is a left child or a right child

Binary Trees: Deletion

- Case 2: Deleting a Node with One left Child
  - `Parent.left = parent.left.left;`



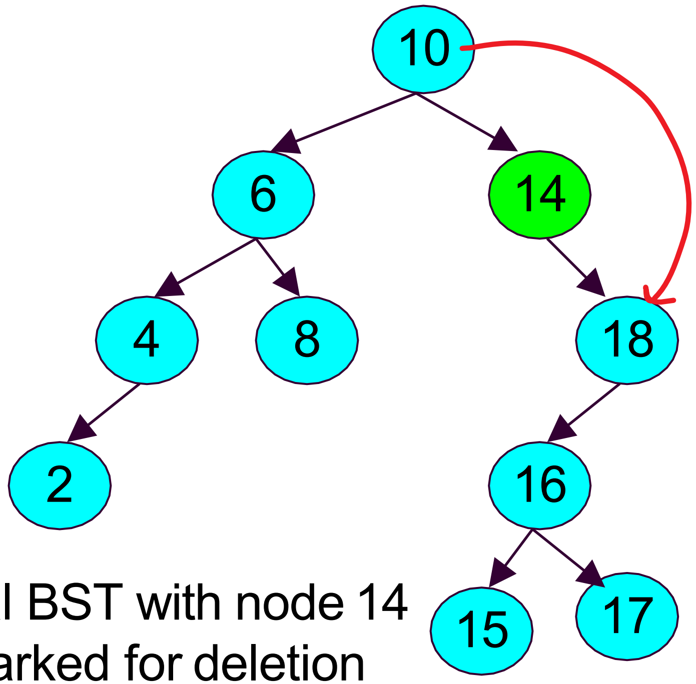
Initial BST with node 4 marked for deletion



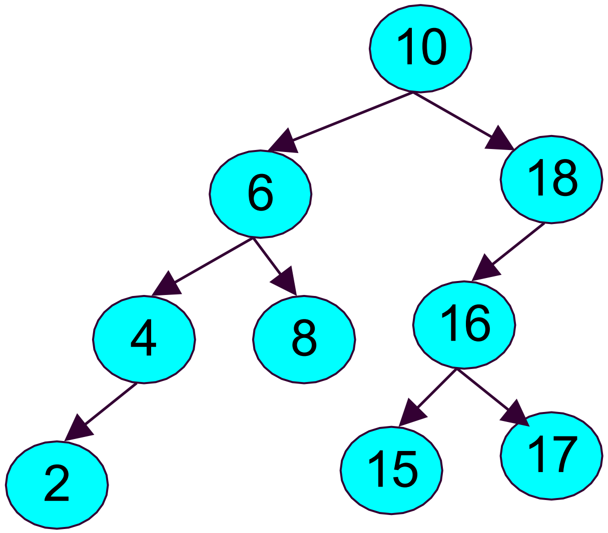
BST after deletion of node 4. Node 2 has taken the place of the deleted node

Binary Trees: Deletion

- Case 2: Deleting a Node with One right Child
  - `Parent.right = parent.right.right;`



Initial BST with node 14 marked for deletion



BST after deletion of node 14. Node 18 has taken the place of the deleted node and the entire subtree moved up one level.

Binary Trees: Deletion

Case 3: Deleting a Node with two children

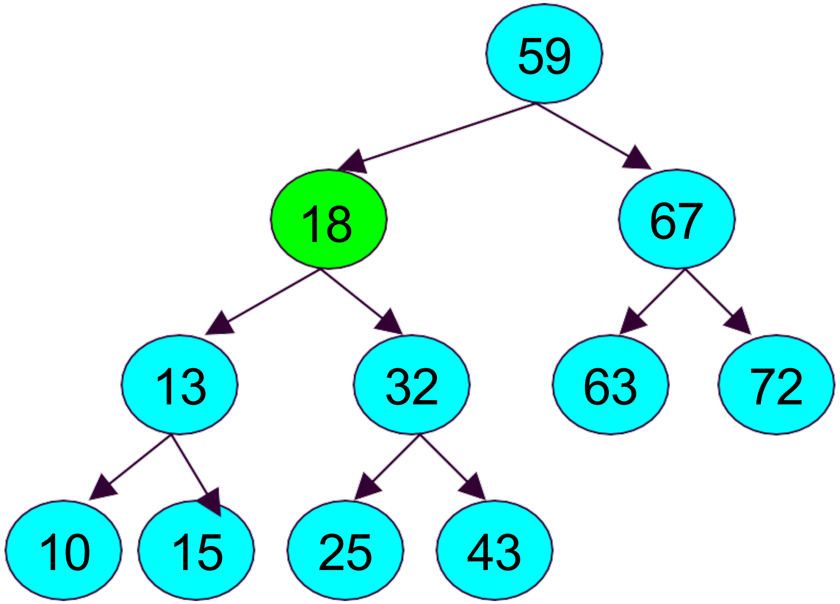
Consider this example tree

We want to delete node 18

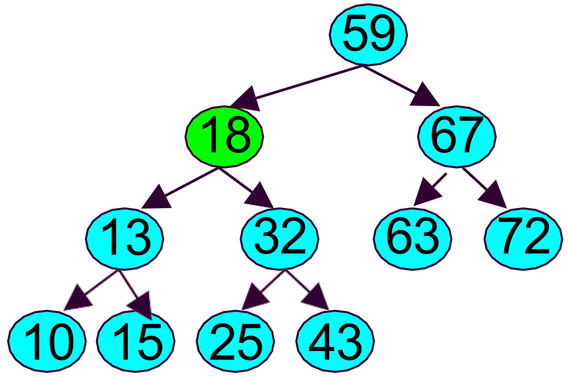
since node 18 can't use its left pointer to point to more than one child

can't point to both node 13 and node 32

What node could I replace the 18 with and still maintain the binary search tree property?



Binary Trees: Deletion



Deleting a Node with two children

Remember:

All the nodes to the left of 18 MUST be smaller than 18

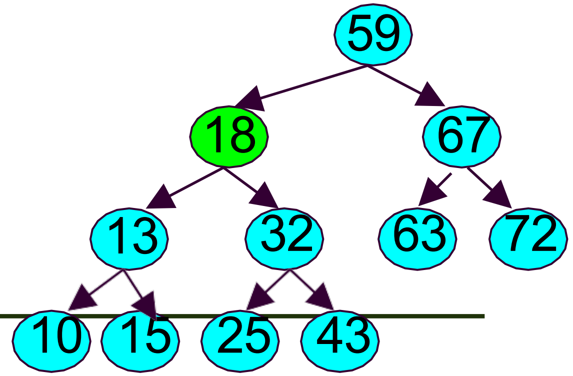
All the nodes right of 18 MUST be greater than 18

Thus, if we delete 18

There are only two nodes we could put at 18's position without causing serious repercussions:

1) The **maximum value in the left subtree** of node 18
   - Which is 15
2) The **minimum value in the right subtree** of node 18
   - Which is 25

Binary Trees: Deletion



- Deleting a Node with two children
  - Thus, if we delete 18
    - There are two possible nodes that could go into 18's position:

1) Node 15 (greatest value in left subtree)
2) Node 25 (smallest value in right subtree)

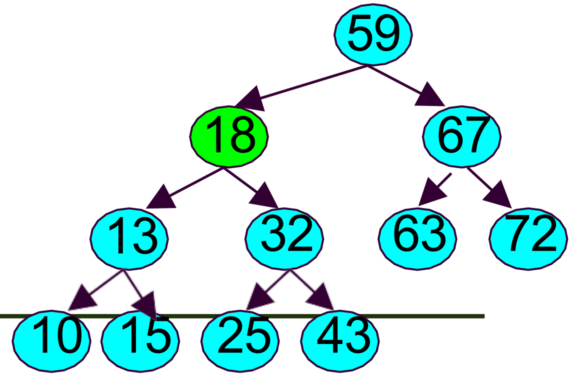
We simply pick one of these to put at 18's position

- We essentially **<u>copy</u>** the node to 18's position

Then we have to delete the actual node that we just copied

- Meaning, if we copy node 15 into 18's positon
- We will have two 15s
- So we now need to delete the leaf node 15

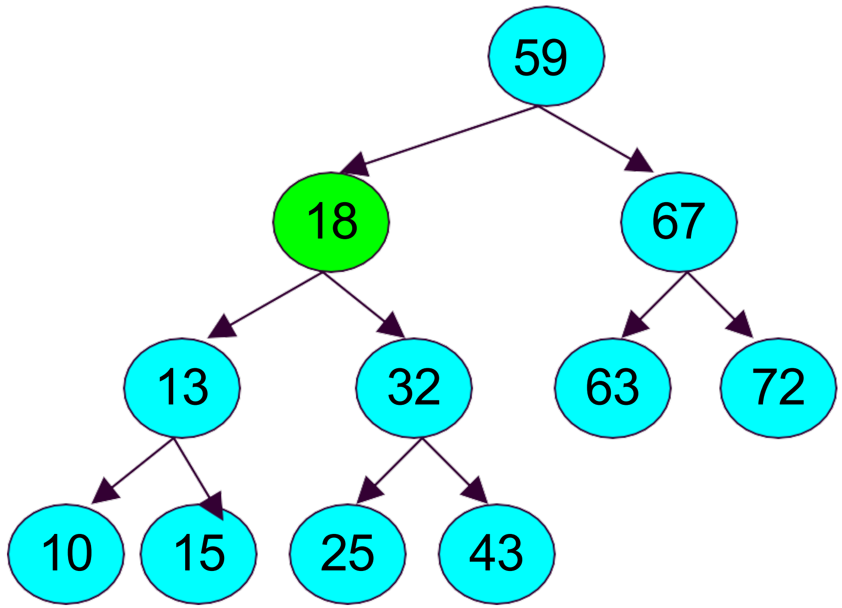
Binary Trees: Deletion



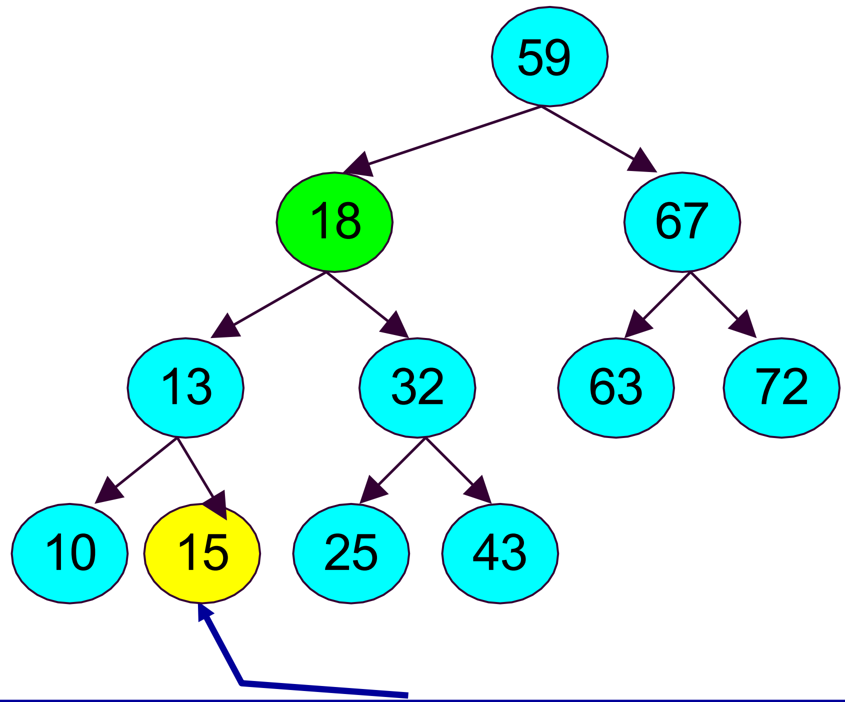
- Deleting a Node with two children
  - <u>We are **guaranteed** that this node</u>
    - Node 15 in this example
  - <u>Has AT MOST only one child</u>
    - Meaning it will be easy to delete!
  - Why is that? How is this guarantee true?
    - The <u>greatest node in a left subtree</u> cannot have two children
      - If it did, its right child would be greater than it
    - Similarly, the <u>smallest node in a right subtree</u> cannot have two children
      - If it did, its left child would be smaller than it

Binary Trees: Deletion

- Deleting a Node with two children (Example)



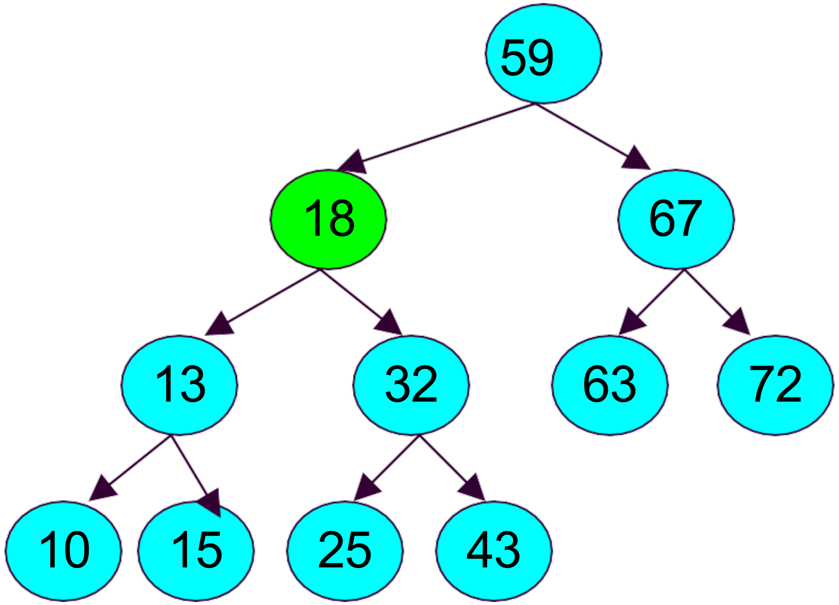
Initial BST with node 18 marked for deletion. Note that this node has two children with values 13 and 32.

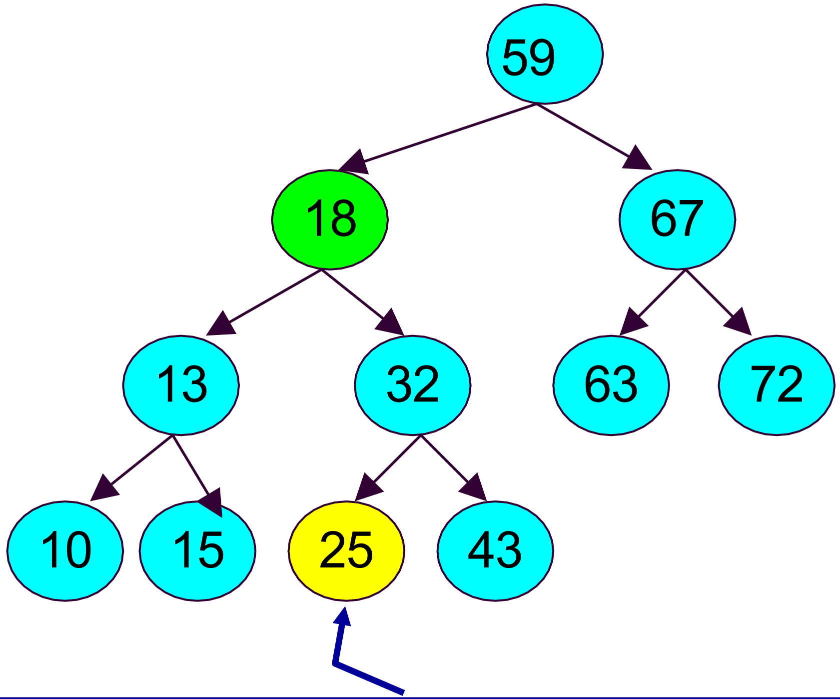


This node contains the logical **predecessor** of the node to be deleted. Note that it is the **greatest node** in the **left subtree** of the node to be deleted.

Binary Trees: Deletion

- Deleting a Node with two children (Example)



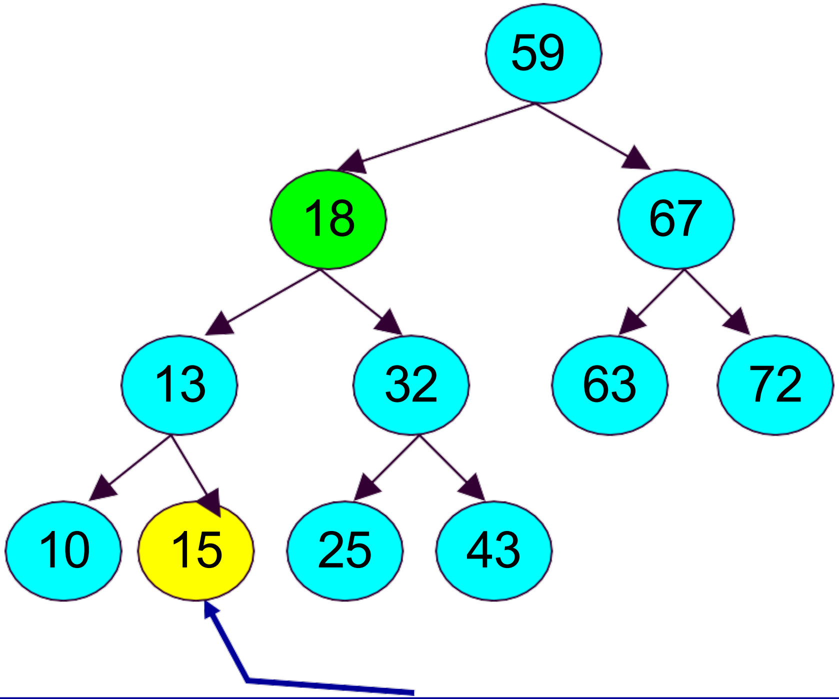


Initial BST with node 18 marked for deletion. Note that this node has two children with values 13 and 32.

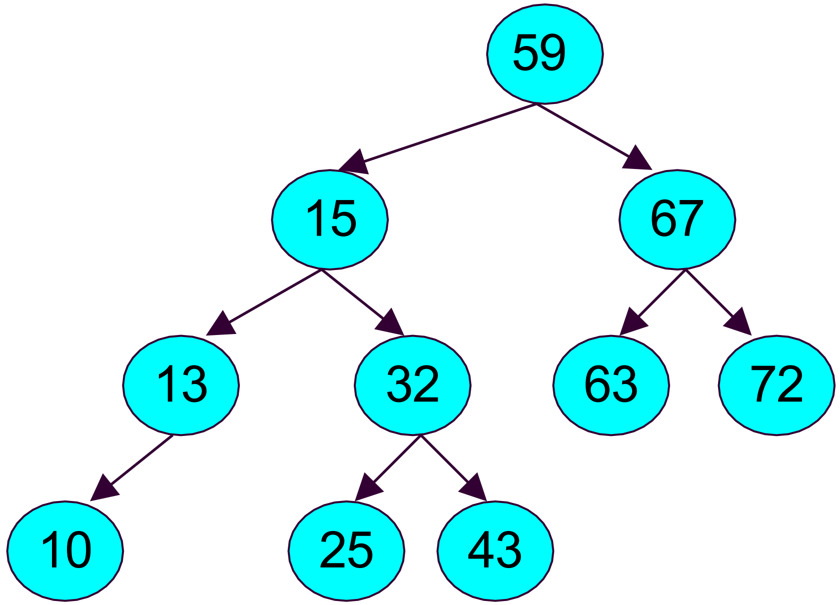
This node contains the logical **successor** of the node to be deleted. Note that it is the **smallest node** in the **right subtree** of the node to be deleted.

Binary Trees: Deletion

- Deleting a Node with two children (Example)



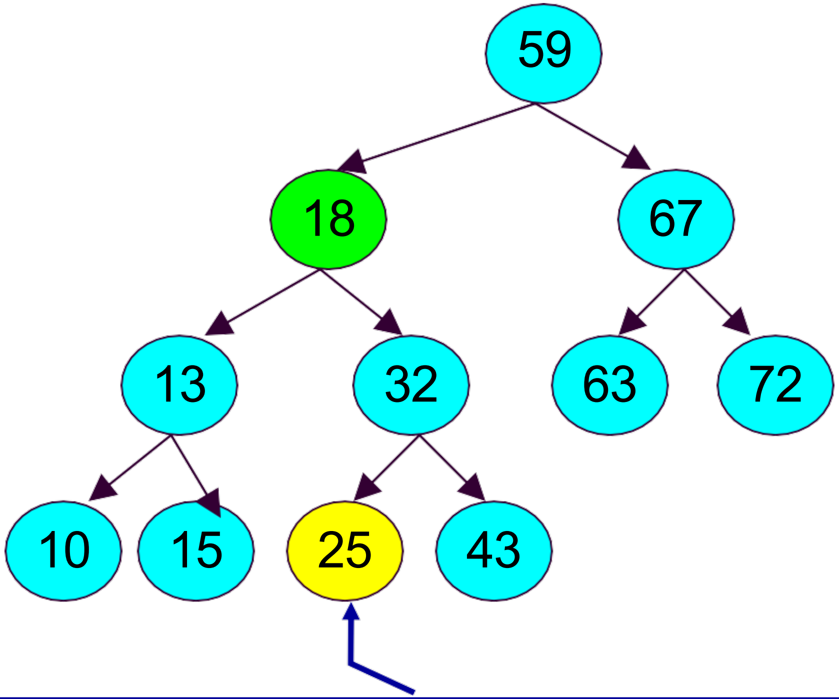
This node contains the logical **predecessor** of the node to be deleted. Note that it is the **greatest node** in the **left subtree** of the node to be deleted.



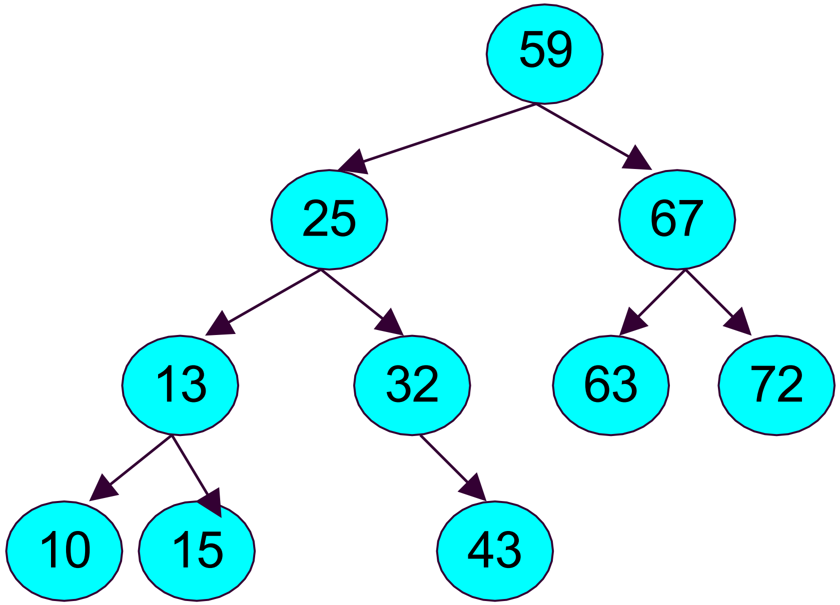
The BST after the deletion of node 18 using the replacement by the logical predecessor node.

Binary Trees: Deletion

- Deleting a Node with two children (Example)



This node contains the logical **successor** of the node to be deleted. Note that it is the **smallest node** in the **right subtree** of the node to be deleted.



The BST after the deletion of node 18 using the replacement by the logical successor node.

Binary Trees: Deletion

Deleting a Node with two children

- For the previous example,
  - The nodes that we copied to 18's position were both leaf nodes
  - How did this help?
    - Once copied over, we need to delete those nodes
    - Since they are leaves, this process is easy
  - But what if they are not leaf nodes?
    - Meaning, they have one child
      - Remember, we are guaranteed that they have AT MOST one kid
    - It is still easy!
    - We would simply be deleting a node with one child
    - Which simply "lifts" up that subtree one level

```
Node deleteRec(Node root, int key)
    {
        if (root == null)  /* Base Case: If the tree is empty */
            return root;
        if (key < root.key)  /* Otherwise, recur down the tree */
            root.left = deleteRec(root.left, key);
        else if (key > root.key)
            root.right = deleteRec(root.right, key);

        // if key is same as root’s key, then This is the node to be deleted
        else {
            // node with only one child or no child
            if (root.left == null)
                return root.right;
            else if (root.right == null)
                return root.left;

       // node with two children: Get the smallest in the right subtree
            root.key = minValue(root.right);

            // Delete the inorder successor
            root.right = deleteRec(root.right, root.key);
        }

        return root;
    }
```

```
int minValue(Node root)
    {
        int minv = root.key;
        while (root.left != null) {
            minv = root.left.key;
            root = root.left;
        }
        return minv;
    }
```

Practice Problems

Binary Trees: Practice Problems

Write a method that Sum the values of nodes in a Tree

```
int sumNodes() {
    return(sumNodes(root));
}

int sumNodes(intBSTnode p) {
    if (p == null)
        return(0);
    else {
        return p.data + sumNodes(p.left) + sumNodes(p.right);
    }
}
```

Binary Trees: Practice Problems

Write a method that counts the number of nodes in a binary tree

```
int countNodes() {
   return(countNodes(root));
}

int countNodes(intBSTnode p) {
   if (p == null)
      return(0);
   else {
      return 1 + countNodes(p.left) + countNodes(p.right);
   }
}
```

Binary Trees: Practice Problems

- Print Even Nodes

```
void printEven(intBSTnode p) {
   if (p != null) {
      if (p.data % 2 == 0)
         System.out.println( p.data);
      printEven(p.left);
      printEven(p-.ight);
   }
}
```

- This is basically just a traversal
  - Except we added a condition (IF) statement before tle print statement

Binary Trees: Practice Problems

- Print Odd Nodes

```
void printOdd(intBSTnode p) {
   if (p != null) {
      if (p.data % 2 != 0)
         System.out.println( p.data);
      printOdd(p.left);
      printOdd(p.right);
   }
}
```

- This is basically just a traversal
  - Except we added a condition (IF) statement before the print statement

Binary Trees: Practice Problems

- Print Nodes (in ascending order):

```
void printAsc(intBSTnode* p) {
    if (p != null)
    {
     printAsc(p.left);
    System.out.println(p.data);
       printAsc(p.right);
    }
}
```

The question requested **ascending** order

This requires an **inorder** traversal

Binary Trees: Practice Problems

- Print Nodes (in descending order):

```
void printDes(intBSTnode p) {
   if (p != null) {
      printDes(p.right);
      System.out.println( p.data);
      printDes(p.left);
   }
}
```

The question requested **descending** order

This requires an **inorder** traversal

Binary Trees: Practice Problems

Compute Height

Defined as the length of the longest path from the root to a leaf node

```
int height(intBSTnode root) {
        int leftHeight, rightHeight;
        if (root == null)
                return -1;
        leftHeight = height(root.left);
        rightHeight =height(root.right);

         if (leftHeight > rightHeight)
                return leftHeight + 1;

         return rightHeight + 1;
 }
```

Binary Trees: Practice Problems

Write a recursive method that returns the largest element in a BST

So where is the largest node located

- Either the root or the greatest node in the right subtree

```
intBSTnode largest(intBSTnode BST) {
        //if BST is NULL, there is no node
        if (BST == null)
                return null;
        //If BST's right is NULL, that means BST is the largest
        else if (BST.right == null)
                return BST;

        // SO if BST's right was NOT equal to NULL,
        // There is a right subtree of BST.
        // Which means that the largest value is in this
        // subtree.  So recursively call BST's right.
        else
                return largest(BST.right);
}
```

Binary Trees: Practice Problems

Write a method that counts the number of leaf nodes in BST

```
int numLeaves () {
    return(numLeaves(root));
}

int numLeaves(intBSTnode p) {
    if (p != NULL) {
        if (p.left == NULL && p.right == NULL)
            return 1;
        else
            return numLeaves(p.left) + numLeaves(p.right);
    }
    else
        return 0;
}
```

Binary Trees: Practice Problems

Write a recursive method that counts the number of nodes with a single child

```
int one(intBSTnode p) {
        if (p != NULL) {
                if (p.left == null)
                        if (p.right != null)
                                return 1 + one(p.right);
                else if (p.right == null)
                         if (p.left != null)
                                return 1 + one(p.left);
                else
                        return one(p.left) + one(p.right);
        }
}
        return 0;
```