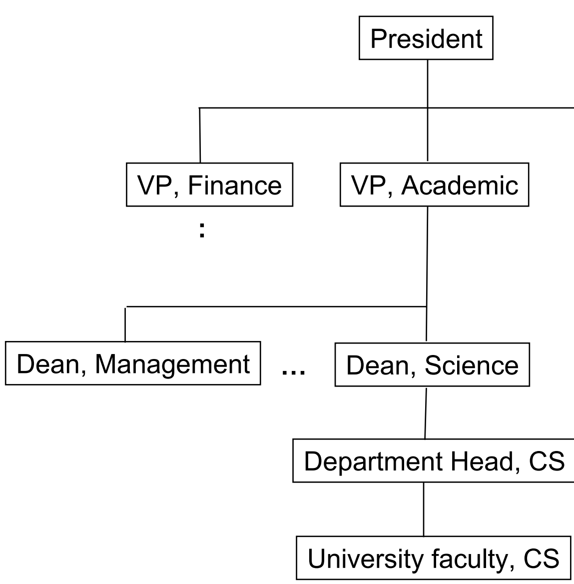
Trees

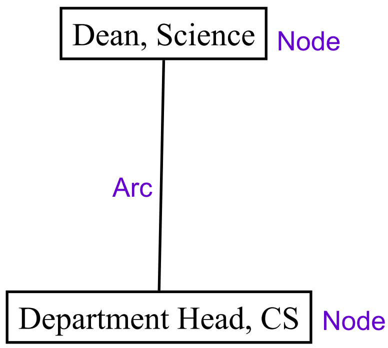
- Lists as an abstract data type (ADT)
- Different list implementations and the tradeoffs of each approach

Trees Can Be Used To Represent A Hierarchy

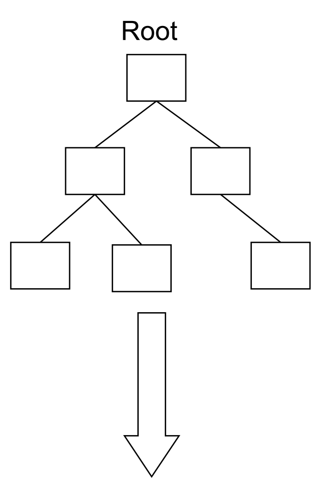


Trees Consist Of Nodes And Arcs

The number of arcs = The number of nodes - 1



Note: Computer People Are Wacky!

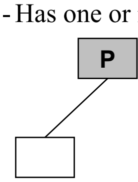


Tree Terminology

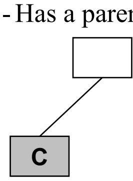
- Parent (descendent)
- Child (ancestor)
- Siblings
- Root
- Left/right sub-tree
- Leaf
- Internal node
- Path length
- The levels of a tree
- Tree height

Tree Terminology

- Parent (ancestor):
  - Has one or more nodes below it. All nodes but the root have one parent.

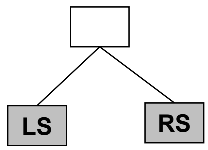


- Child (descendent):
  - Has a parent node above it.

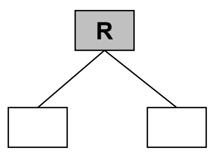


Tree Terminology (2)

- Siblings:
  - Nodes that have the same parent

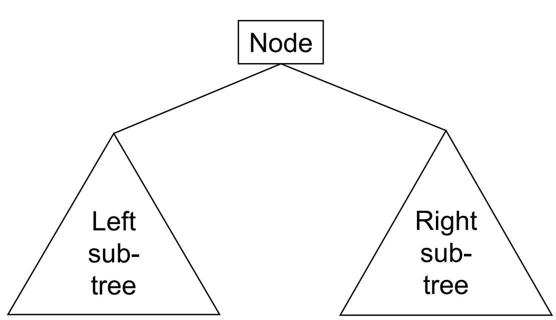


- Root:
  - The parent of all the nodes in a tree.
  - It has no parent



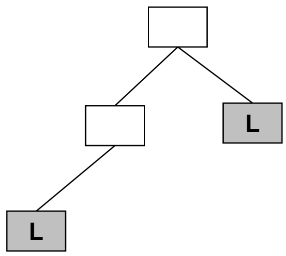
Tree Terminology (3)

- Left sub-tree
  - For a given node, the left sub-tree will consist of the left child node and all the children of the child node.
- Right sub-tree
  - For a given node, the right sub-tree will consist of the right child node and all the children of the child node.



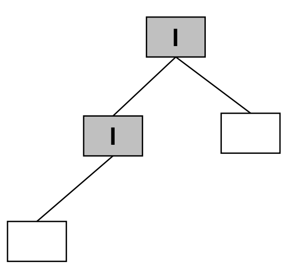
Tree Terminology (4)

- Leaf
  - Has no child nodes

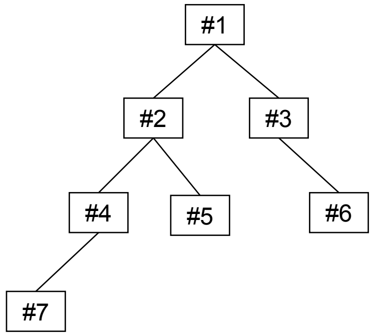


Tree Terminology (5)

- Internal Node
  - Is a non-leaf node
  - Has at least one child node



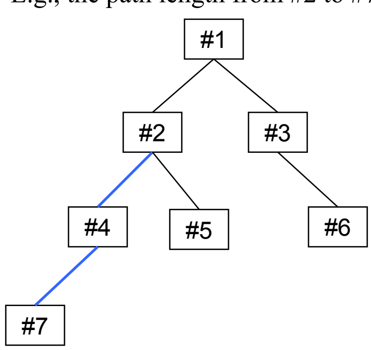
Tree Terminology (6)



- The root is node #1
- #2 is the parent of 4 & 5
- #4 & 5 are the children of 2
- #4 & 5 are siblings
- Leaf nodes: #5, 6, 7
- Internal nodes: #1, 2, 3, 4

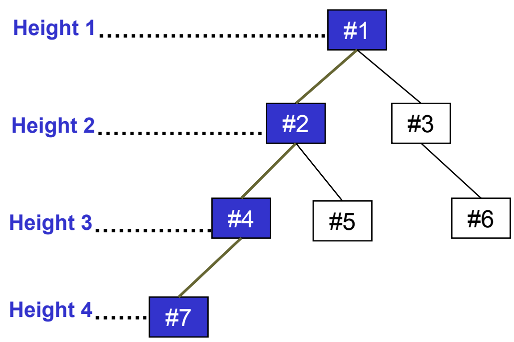
Tree Terminology (7)

- Path length
  - The number of edges that must be traversed to get from one node to another.
  - E.g., the path length from #2 to #7



Tree Terminology (8)

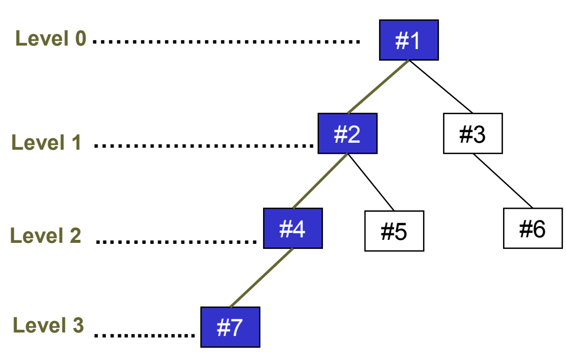
- **Height of a tree:**
  - The number of nodes from the root to the most distant leaf node
- **Levels of a tree**
  - The number of edges from the root to the most distant leaf node
  - It's the path length from the root the most distant node
  - Equal to the height of the tree minus one.



James Tam

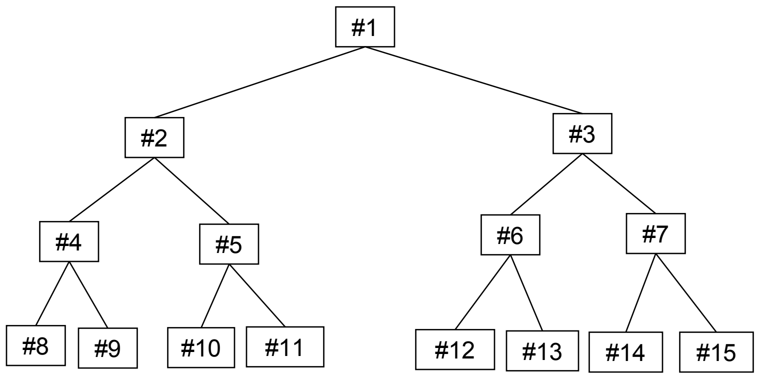
Tree Terminology (9)

- **Height of a tree:**
  - The number of nodes from the root to the most distant leaf node
- **Levels of a tree**
  - The number of edges from the root to the most distant leaf node
  - It's the path length from the root the most distant node
  - Equal to the height of the tree minus one.



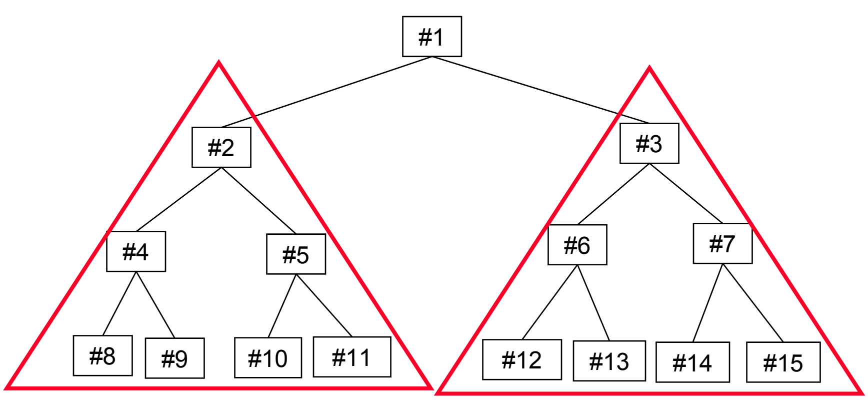
James Tam

Trees Are Recursively Defined



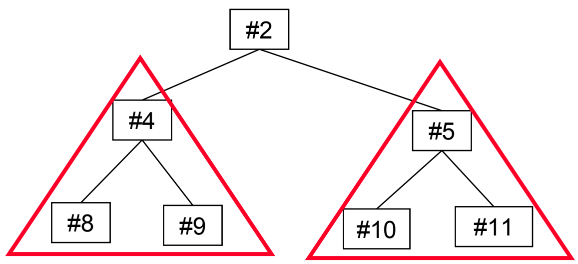
Trees Are Recursively Defined

•The root of the tree consists of the root's left and right sub-trees



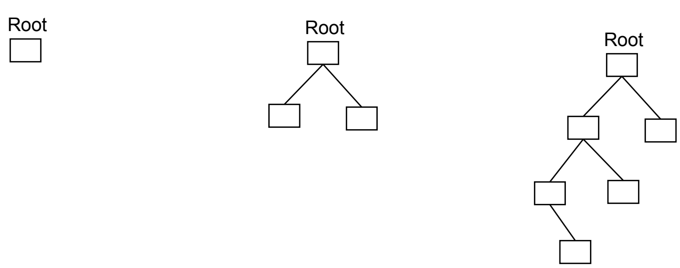
Trees Are Recursively Defined

•Each sub-tree has a top-level node that is composed of it's own left and right sub-trees.



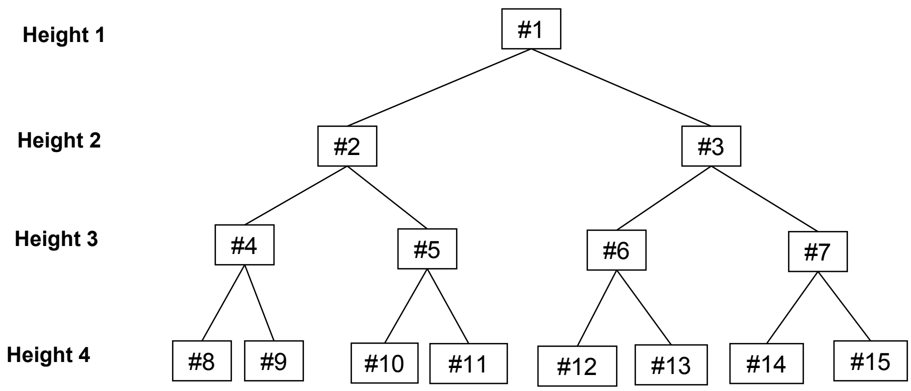
Binary Tree

•A tree with nodes that can have 0, 1 or 2 children.



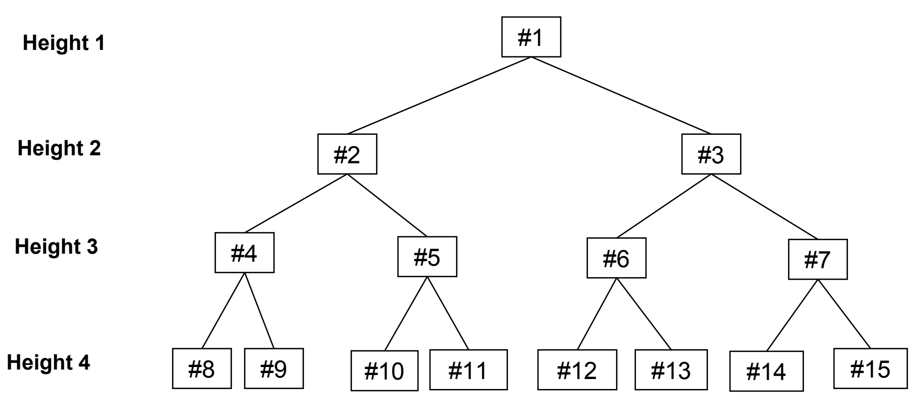
The Max Nodes Per Line

Max nodes for a tree with a given height $= 2^{(height-1)}$



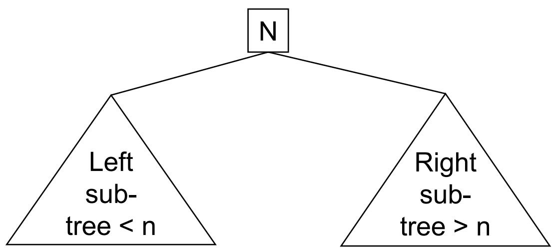
The Max Nodes For A Tree

Max nodes for a tree with a given height $= 2^{height} - 1$

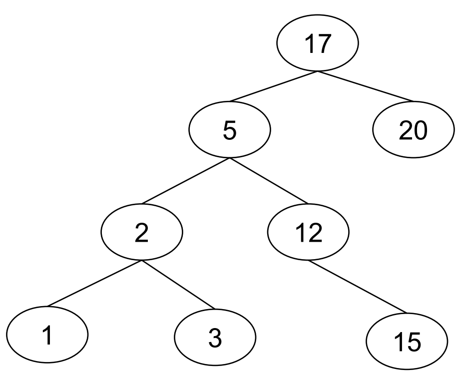


Binary Search Trees

- A binary tree with the property such that all the nodes of the left sub-tree of a particular node will have values less than the value of the parent node. All the nodes of the right sub-tree will have values greater than the value of the parent node.
- Left children < Parent < Right children
- For some trees no duplicates are allowed: adding a duplicate node results in an error condition.



Example Of A Binary Search Tree



Non-Ordered Binary Trees

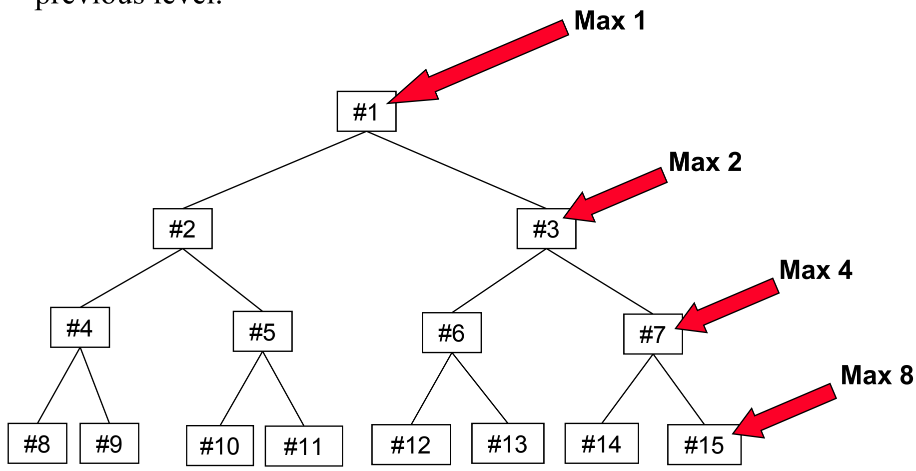
•In this case order doesn't matter so the following trees are identical (not binary search trees)

Binary Tree Implementations

1. Array implementation
2. Linked implementation

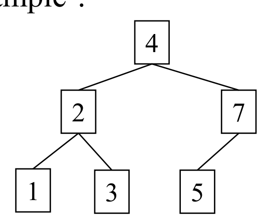
Array Implementations

- Because each node can have at most 2 children, the max nodes for a given level will be double the number of nodes for the previous level.



Array Implementation (2)

- For a given node at index "I", the left child of that node will be at index = (I * 2) and the right child will be at index = (I * 2) + 1
- Example[1]:

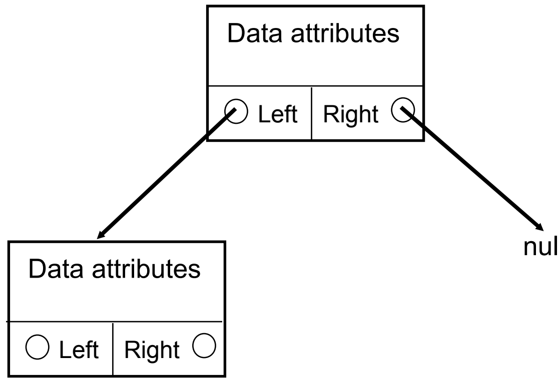


[1]	[2]	[3]	[4]	[5]	[6]		
4	2	7	1	3	5		

From http://pages.cpsc.ucalgary.ca/~marina/331/

Linked Implementation

•A tree is composed of nodes and links between the nodes:



Additional Definitions For Binary Trees

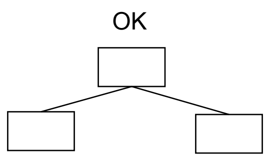
•Full tree

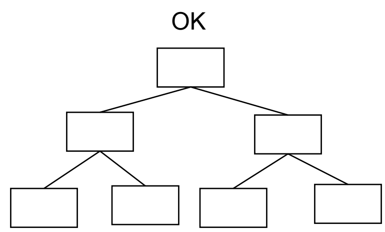
•Degenerate tree

•Balanced tree

Full Binary Tree

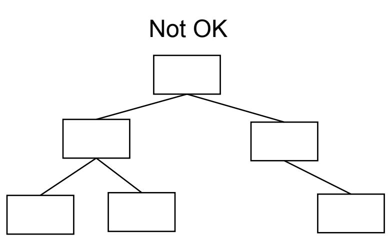
•A tree with height "h" such that all leaves exist only at height "h", all nodes above this level each have two children

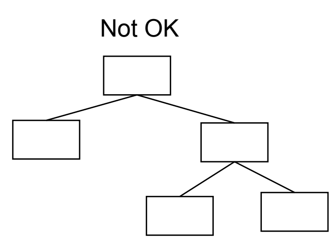




Full Binary Tree (2)

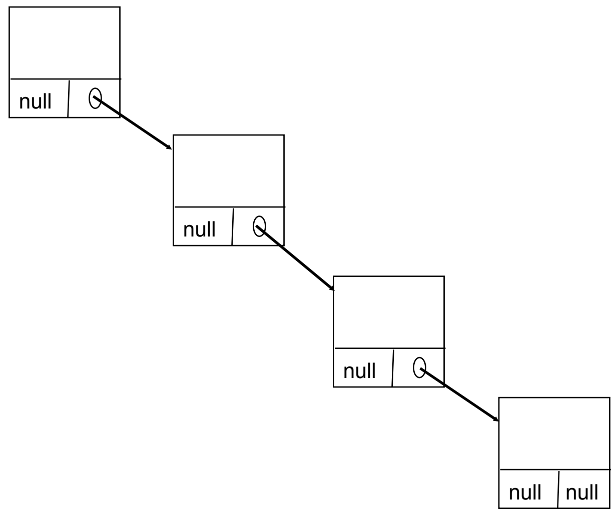
•A tree with height "h" such that all leaves exist only at height "h", all nodes above this level each have two children





Degenerate Binary Tree

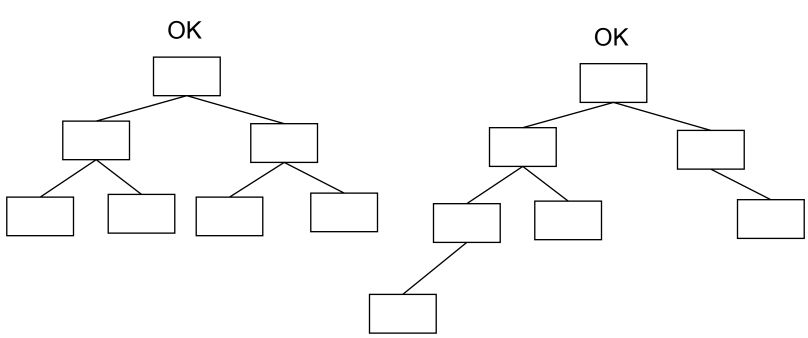
•Looks like a linked list



James Tam

Balanced Binary Tree

•The height difference of the sub-trees of all nodes is either zero or one.

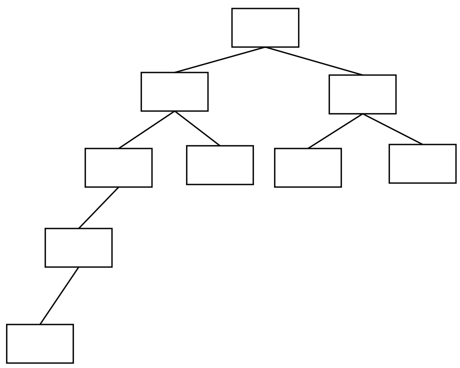


James Tam

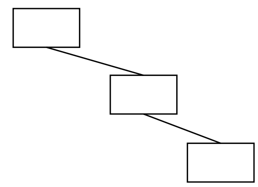
Balanced Binary Tree (2)

•The height difference of the sub-trees of all nodes is either zero or one.

Not OK



Not OK



Operations On Binary Trees

•Insertions

•Search

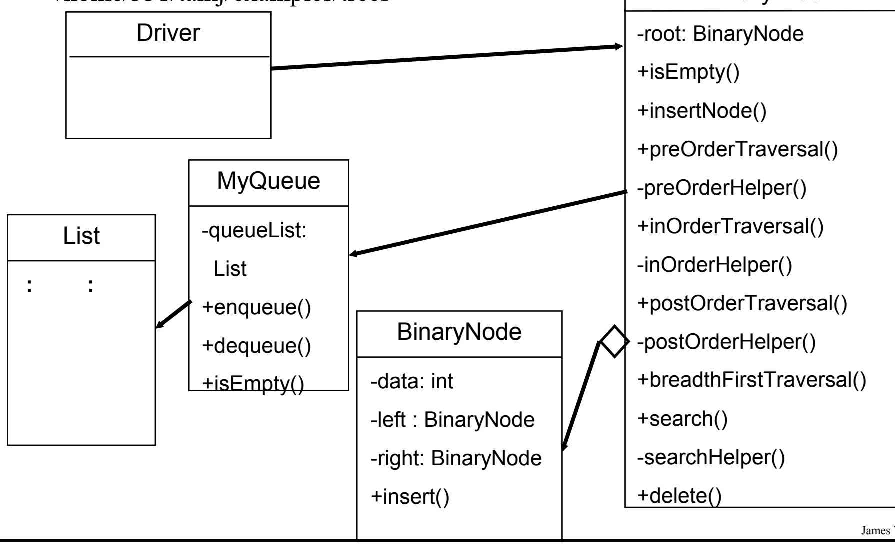
•Traversal

- Depth-first (in order, preorder, post order)
- Breadth first

•Deletions

Example Of A Binary Tree

- The full example can be found in the directory: /home/331/tamj/examples/trees



The Driver Class

```
class Driver
{
   public static void main (String [] args)
   {
      BinaryTree myTree = new BinaryTree ();
      myTree.insertNode(39);
      myTree.insertNode(69);
      myTree.insertNode(94);
      myTree.insertNode(47);
      myTree.insertNode(50);
      myTree.insertNode(72);
      myTree.insertNode(55);
      myTree.insertNode(41);
      myTree.insertNode(97);
      myTree.insertNode(73);
```

The Driver Class (2)

```
myTree.preOrderTraversal();
myTree.inOrderTraversal();
myTree.post orderTraversal();
myTree.breadthFirstTraversal();
System.out.println("Searching tree");
myTree.search(47);
myTree.search(888);
int num;
System.out.println("Deleting from tree");
System.out.print("Data of node to delete: ");
num = Console.in.readInt();
Console.in.readLine();
System.out.println("Deleting " + num);
myTree.delete(num);
myTree.breadthFirstTraversal();
```


The BinaryTree Class

```
public class BinaryTree
{
  private BinaryNode root;

  public BinaryTree ()
  {
    root = null;
  }
  public boolean isEmpty ()
  {
    if (root == null)
      return true;
    else
      return false;
  }
```

Inserting A Node

- Recall (Binary Search Trees):
  - Left children < Parent < Right children
- Nodes must be inserted into their proper (in order) place as they are added

```
(Class Driver)
    myTree.insertNode(39);
    myTree.insertNode(69);
    myTree.insertNode(94);
    myTree.insertNode(47);
    myTree.insertNode(50);
    myTree.insertNode(72);
    myTree.insertNode(55);
    myTree.insertNode(41);
    myTree.insertNode(97);
    myTree.insertNode(73);
```


Inserting A Node (2)

```
(Class BinaryTree)
 public void insertNode (int newValue)
   {
     if (isEmpty() == true)
       root = new BinaryNode (newValue);
     else
       // Call the insert method of the Binary Node class.
       root.insert(newValue);
   }
```

Inserting A Node (3)

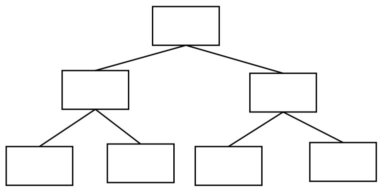
```
(Class Binary Node)
    void insert (int newValue)
    {
        if (newValue < data)
        {
             if (left == null)
                left = new BinaryNode (newValue);
            else
                left.insert(newValue);
        }
         else if (newValue > data)
        {
            if (right == null)
                right = new BinaryNode (newValue);
            else
                right.insert(newValue);
```


Inserting A Node (4)

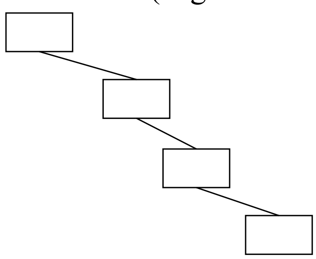
```
         else
        {
            System.out.println("Error: duplicate values for nodes are not " +
                                 "allowed.");
            System.out.println("There is already a node with a value of " +
                                 newValue);
        }
    }
```

Efficiency Of Insertions: Depends Upon The Height

•Best case (full tree) : $O(\log_2 n)$ = height of the tree



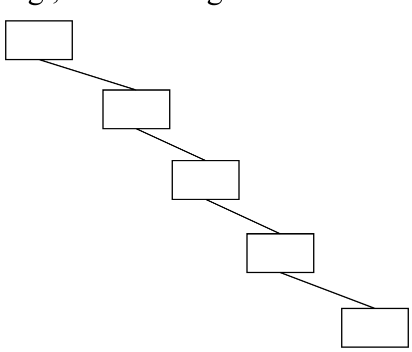
•Worse case (degenerate tree) : $O(n)$ = height of the tree



The Search Time Depends Upon Height

•Maximum height (Degenerate tree) = n

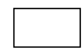
•E.g., n = 5 = height



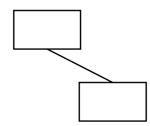
The Search Time Depends Upon Height

- Minimum height[1] = $\lceil \log_2(n+1) \rceil$

e.g., n = 1, min = ceiling $(\log_2(1+1))$
= ceiling (1)
= 1



e.g., n = 2, min = ceiling $(\log_2(2+1))$
= ceiling (1.X)
= 2

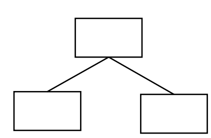


1 This can be proven inductively but this will be left for subsequent courses

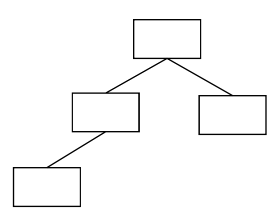
The Search Time Depends Upon Height

- Minimum height[1] = $\lceil \log_2(n+1) \rceil$

e.g., n = 3, min = ceiling $(\log_2(3+1))$
= ceiling (2)
= 2



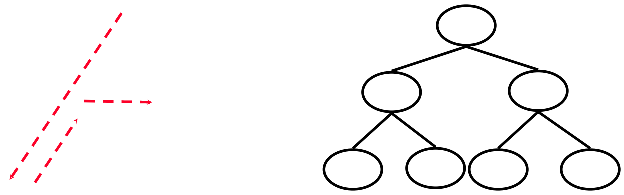
e.g., n = 4, min = ceiling $(\log_2(4+1))$
= ceiling (2.X)
= 3



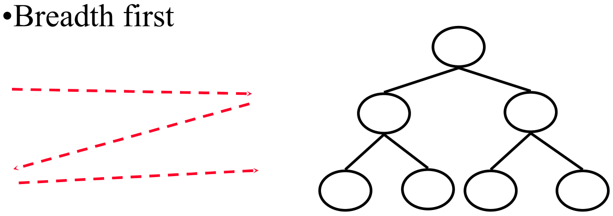
1 This can be proven inductively but this will be left for sub-sesequent courses

Tree Traversals

•Depth first: inorder, preorder, post order

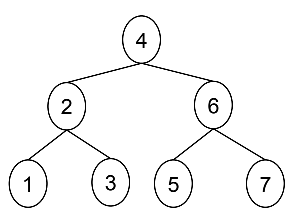


•Breadth first

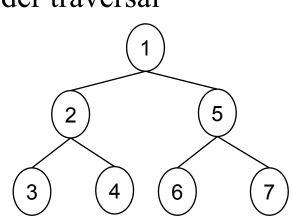


Depth-First Traversals

1. Inorder traversal

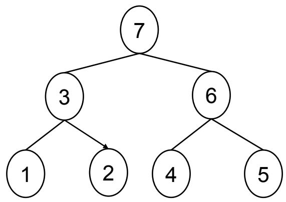


2. Preorder traversal



Depth-First Traversals (2)

3. Post order traversal



Inorder Traversals

(Class Driver)

```
myTree.inOrderTraversal();
```

(Class BinaryTree)

```
public void inOrderTraversal ()
{
   inOrderHelper(root);
   System.out.println();
}
```

Inorder Traversals (2)

(Class BinaryTree)

```
private void inOrderHelper (BinaryNode node)
{
    if (node == null)
        return;
    inOrderHelper(node.getLeft());
    System.out.print(node + " ");
    inOrderHelper(node.getRight());
}
```


Preorder Traversals

(Class Driver)

```
myTree.preOrderTraversal();
```

(Class BinaryTree)

```
public void preOrderTraversal ()
{
    preOrderHelper(root);
}
```

Preorder Traversals (2)

(Class BinaryTree)

```
private void preOrderHelper (BinaryNode node)
{
   if (node == null)
      return;
   System.out.print(node + " ");
   preOrderHelper(node.getLeft());
   preOrderHelper(node.getRight());
}
```


Post order Traversals

(Class Driver)

```
myTree.post orderTraversal();
```

(Class BinaryTree)

```
public void post orderTraversal ()
{
   postorderHelper(root);
}
```

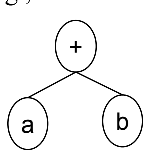
Post order Traversals (2)

```
(Class BinaryTree)
  private void postOrderHelper (BinaryNode node)
  {
     if (node == null)
        return;
     postOrderHelper(node.getLeft());
     postOrderHelper(node.getRight());
    System.out.print(node + " ");
  }
```


Why Bother With Depth-First Traversals?

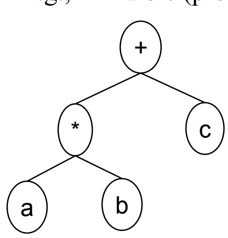
•Inorder traversal is analogous to infix notation

- E.g., a + b



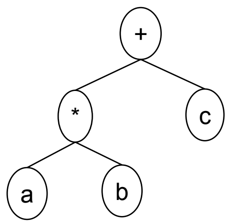
•Preorder traversal is analogous to prefix notation

- E.g., + * a b c (prefix) equivalent to a * b + c (infix)



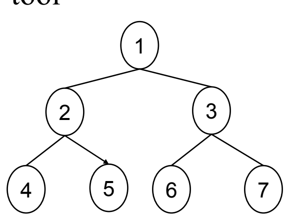
Why Bother With Depth-First Traversals (2)?

- Post order traversal is analogous to postfix notation
  - E.g., a b * c + (postfix) equivalent to a * b + c (infix)



Breadth-First Traversal

- Traverse the tree one level at a time.
- Hint: Since displaying the tree this way corresponds to how it is physically organized, it makes a very good debugging/tracing tool



Breadth-First Traversal (2)

```
public void breadthFirstTraversal ()
{
    BinaryNode tempNode = root;
    MyQueue tempQueue;
    System.out.println("Breadth first traversal");
    if (tempNode != null)
    {
        tempQueue = new MyQueue ();
        tempQueue.enqueue(tempNode);
```


Breadth-First Traversal (3)

```
        while (tempQueue.isEmpty() == false)
        {
           tempNode = (BinaryNode) tempQueue.dequeue();
            System.out.print(tempNode + " ");
            if (tempNode.getLeft() != null)
                tempQueue.enqueue(tempNode.getLeft());
            if (tempNode.getRight() != null)
                tempQueue.enqueue(tempNode.getRight());
        }
    }
    System.out.println();
}
```

Class MyQueue

```
import java.util.LinkedList;
public class MyQueue
{
   private LinkedList queueList;

   public MyQueue ()
   {
      queueList = new LinkedList ();
   }

   public void enqueue (Object newNode)
   {
      queueList.addLast(newNode);
   }
```


Class MyQueue (2)

```
   public Object dequeue ()
   {
      return queueList.removeFirst();
   }

   public boolean isEmpty ()
   {
      if (queueList.size() == 0)
         return true;
      else
         return false;
   }
}
```

Efficiency Of Tree Traversals

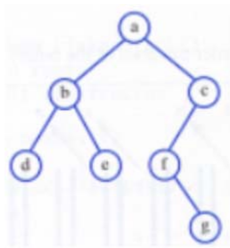
- Since each node must be visited in order to traverse the tree the time taken is *O* (n)

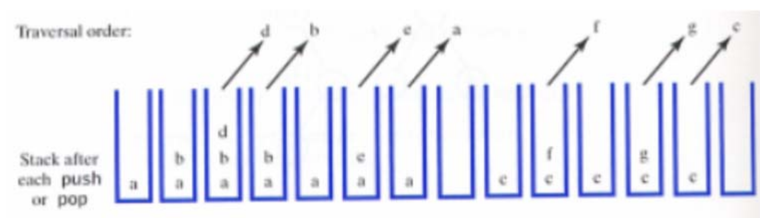
Tree Traversals And Stacks

- Tree traversals must require the use of a stack:
  - Recursively traversing a tree employs the system stack (parameter passing into the recursive methods).
  - Iteratively traversing a tree requires that the programmer create his or her own stack.

Tree Traversals And The System Stack

•Example: An inorder traversal of the following tree.



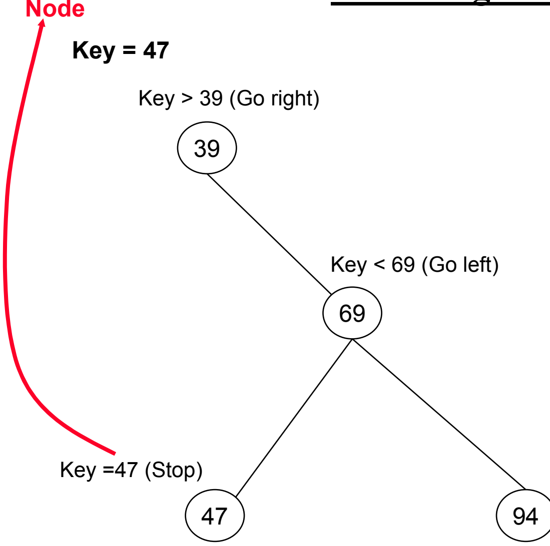


Images from "*Data Abstraction and Problem Solving with Java: Walls and Mirrors*" by Frank M. Carrano and Janet J. Prichard

Searching A Tree

Return address of Node

Key = 47



Searching A Tree (2)

(Class Driver)

```
myTree.search(47);
myTree.search(888);
```

(Class BinaryTree)

```
public void search (int key)
{
    if (searchHelper(root,key) != null)
        System.out.println("Search for node with data value of " + key + " successful");
    else
        System.out.println("Node with data value of " + key + " not found”
                           + " in tree.");
}
```

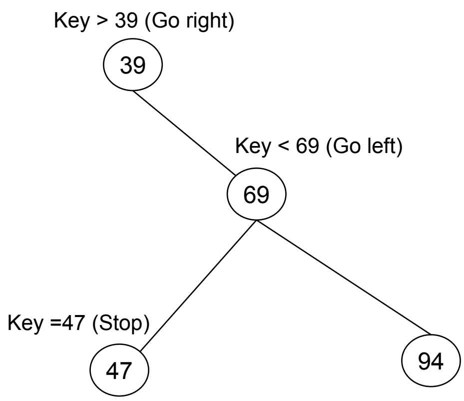

Searching A Tree (3)

(Class BinaryTree)

```
private BinaryNode searchHelper (BinaryNode node, int key)
{
    while (node != null)
    {
       if (key == node.getData())
           return node;
       else if (key < node.getData())
           node = node.getLeft();
       else
           node = node.getRight();
    }
    return null;
}
```

Efficiency Of Searches

•Time in all cases: Equal to the height of the tree

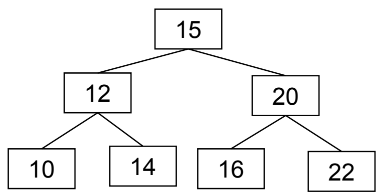


Cases For Deleting Nodes

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

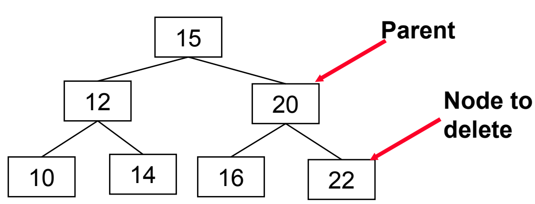
Node To Be Deleted Is A Leaf

- Easiest case e.g., delete 22



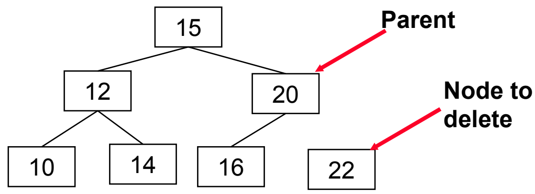
Node To Be Deleted Is A Leaf (2)

- Easiest case e.g., delete 22



Node To Be Deleted Is A Leaf (3)

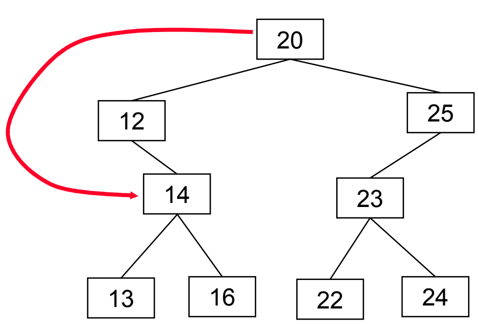
•Easiest case e.g., delete 22



Node To Be Deleted Has One Child

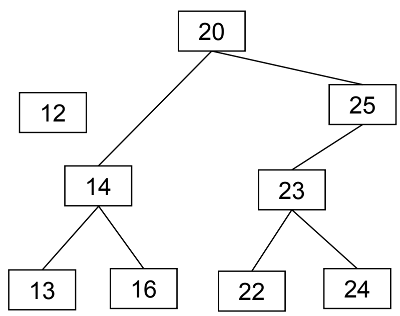
•The node to be deleted either has a left child or a right child (but not both).

•The solution is symmetrical: what applies for the left child also applies for the right and vice versa.



Node To Be Deleted Has One Child (2)

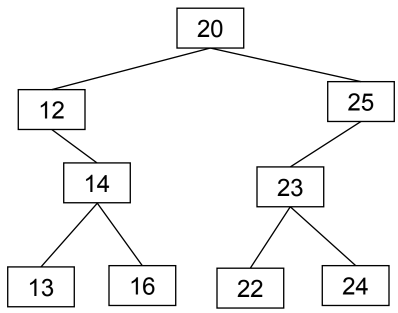
- The node to be deleted either has a left child or a right child (but not both).
- The solution is symmetrical: what applies for the left child also applies for the right and vice versa.



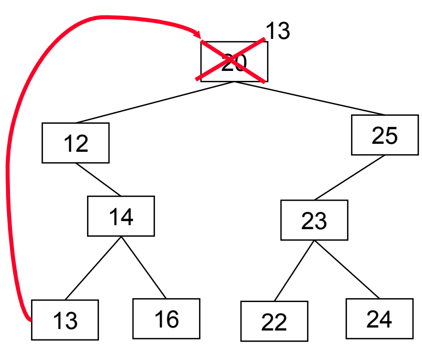
Node To Be Deleted Has Two Children

- More complex: both child nodes cannot be promoted in place of the node to be deleted.
- Instead of deleting the node:
  1. Find another node “A” that is easier to delete than the node to be deleted “D”.
  2. Copy the data from “A” to “D”
  3. Remove node “A” from the tree (setting it’s parent reference to null in Java and by manually de-allocating the memory with languages that don’t have automatic garbage collection).

Not Just Any Node Will Do!

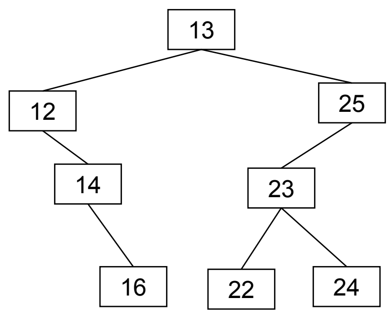


Not Just Any Node Will Do!



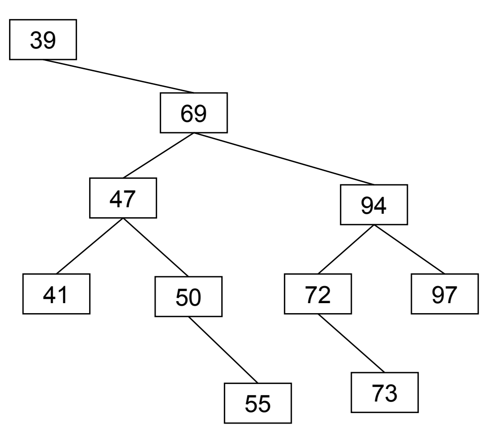
James Tam

Not Just Any Node Will Do!



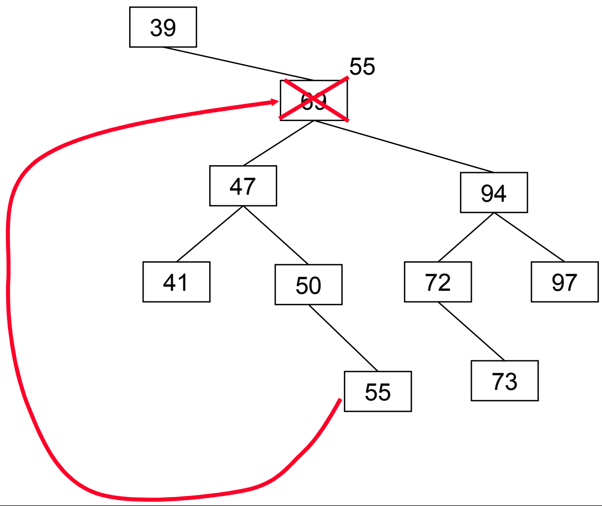
Promote The Next Largest Node

•Example: deleting 69



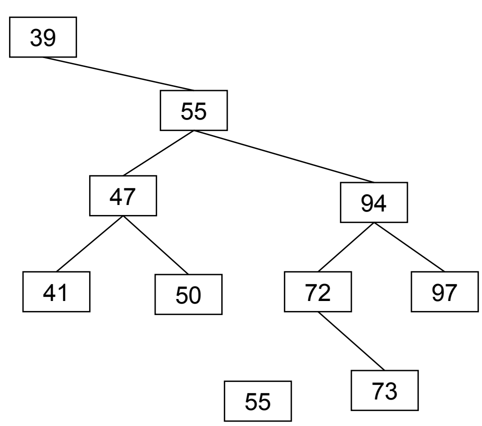
Promote The Next Largest Node

•Example: deleting 69, copy 55



Promote The Next Largest Node

•Example: unlink 55



Deleting A Node

(Class Driver)

```
num = Console.in.readInt();
myTree.delete(num);
```

(Class BinaryTree)

```
public void delete (int key)
  {
    BinaryNode node = null;
    BinaryNode previous = null;
    BinaryNode current = root;
```


Deleting A Node (3)

```
while ((current != null) && (current.getData() != key))
{
    previous = current;
    if (current.getData() < key)
        current = current.getRight();
    else
        current = current.getLeft();
}
node = current;
```

Deleting A Node (4)

```
if ((current != null) && (current.getData() == key))
{
   if (node.getRight() == null)
      node = node.getLeft();
   else if (node.getLeft() == null)
      node = node.getRight();
   else
   {
      BinaryNode temp = node.getLeft();
      BinaryNode prev = node;
      while (temp.getRight() != null)
      {
         prev = temp;
         temp = temp.getRight();
      }
```


Deleting A Node (5)

```
      node.setData(temp.getData());
      if (prev == node)
         prev.setLeft(temp.getLeft());
      else
         prev.setRight(temp.getLeft());
   }
   if (current == root)
      root = node;
   else if (previous.getLeft() == current)
      previous.setLeft(node);
   else
      previous.setRight(node);
}
```

Deleting A Node (6)

```
        else if (root != null)
          System.out.println("Node with data of " + key + " not found.");
        else
          System.out.println("Tree is empty, nothing to delete.");
    }
```


Efficiency Of Deletions

- When the node to be deleted is a leaf: O (1)
- When the node to be deleted has one child: O (1)
- When the node to be deleted has two children: O ($\log_2 n$)

Summary Of Efficiency Of Tree Operations

Operation	Average case	Worse case
Search	*O* ($\log_2 n$)	*O* (n)
Insertion	*O* ($\log_2 n$)	*O* (n)
Deletion	*O* ($\log_2 n$)	*O* (n)
Traversal	*O* (n)	*O* (n)

You Should Now Know

- What is a tree?
- What are different types of trees?
- Common tree terminology.
- How common operations are implemented on a Binary Search Tree.

Sources Of Lecture Material

- *"Data Structures and Abstractions with Java"* by Frank M. Carrano and Walter Savitch
- "*Data Abstraction and Problem Solving with Java: Walls and Mirrors*" by Frank M. Carrano and Janet J. Prichard
- "Data Structures and Algorithms in Java" by Adam Drozdek
- "Java: How to Program (5th Edition)" by Harvey and Paul Deitel
- CPSC 331 course notes by Marina L. Gavrilova http://pages.cpsc.ucalgary.ca/~marina/331/