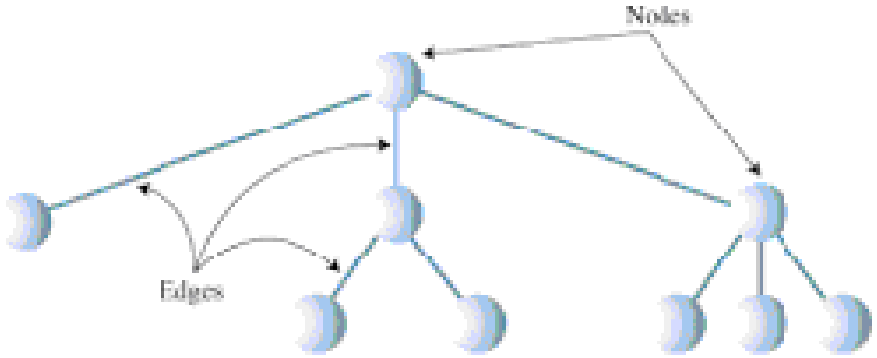
5. Trees

Part -1

© 2008, University of Colombo School of Computing

5.1. Trees, Binary trees and Binary Search trees

- **What Is a Tree?**
  - A tree consists of *nodes* connected by *edges*.



  - In the above picture of the tree, the nodes are represented as circles, and the edges as lines connecting the circles.
  - Trees have been studied extensively as abstract mathematical entities, so there's a large amount of theoretical knowledge about them.
  - A tree is actually an instance of a more general category called a graph.

© 2008, University of Colombo School of Computing

5.1. Trees, Binary trees and Binary Search trees contd…

- **What Is a Tree? contd…**
  - In computer programs, nodes often represent such entities as people, car parts, airline reservations, and etc; in other words, the typical items we store in any kind of data structure.
  - The lines (edges) between the nodes represent the way the nodes are related.

© 2008, University of Colombo School of Computing

5.1. Trees, Binary trees and Binary Search trees contd…

- There are different kinds of trees
  - Binary Tree : each node in a binary tree has a maximum of two children.
  - Multiway trees : more general trees, in which nodes can have more than two children, are called

© 2008, University of Colombo School of Computing

5.1. Trees, Binary trees and Binary Search trees contd…

- **Why Use Binary Trees?**
  - Why might you want to use a tree?
  - Usually, because it combines the advantages of two other structures:
    - an ordered array and
    - a linked list.
  - You can search a tree quickly, as you can an ordered array, and you can also insert and delete items quickly, as you can with a linked list.

© 2008, University of Colombo School of Computing

5.2. Implementation of Binary trees

- **The Node Class**
  - First, we need a class of node objects.
  - These objects contain the data representing the objects being stored (employees in an employee database, for example) and also references to each of the node's two children. Here's how that looks:

© 2008, University of Colombo School of Computing

5.2. Implementation of Binary trees contd…

```
class Node
{
    int iData; // data used as key value
    float fData; // other data
    node leftChild; // this node's left child
    node rightChild; // this node's right child

    public void displayNode()
    {
    // method body
    }
}
```


© 2008, University of Colombo School of Computing

5.2. Implementation of Binary trees contd…

- There are other approaches to designing class Node. Instead of placing the data items directly into the node, you could use a reference to an object representing the data item:

```
class Node
{
        person p1; // reference to person
        object
        node leftChild; // this node's left child
        node rightChild; // this node's right
        child
}
class person
{
        int iData;
        float fData;
}
```

© 2008, University of Colombo School of Computing

5.2. Implementation of Binary trees contd…

- **The Tree Class**
  - We'll also need a class from which to instantiate the tree itself; the object that holds all the nodes.
  - We'll call this class Tree. It has only one field: a Node variable that holds the root.
  - It doesn't need fields for the other nodes because they are all accessed from the root.
  - The Tree class has a number of methods: some for finding, inserting, and deleting nodes, several for different kinds of traverses, and one to display the tree.

© 2008, University of Colombo School of Computing

5.2. Implementation of Binary trees contd…

- Here's a skeleton version:

```
class Tree
{
        private Node root; // the only data field in Tree
        public void find(int key)
        {
        }
        public void insert(int id, double dd)
        {
        }
        public void delete(int id)
        {
        }
        // various other methods
} // end class Tree
```

© 2008, University of Colombo School of Computing

5.2. Implementation of Binary trees contd…

- **The TreeApp Class**
  - Finally, we need a way to perform operations on the tree.
  - Here's how you might write a class with a main() routine to create a tree, insert three nodes into it, and then search for one of them.
  - We'll call this class TreeApp:

© 2008, University of Colombo School of Computing

5.2. Implementation of Binary trees contd…

```
class TreeApp
{
    public static void main(String[] args)
    {
        Tree theTree = new Tree; // make a tree
        theTree.insert(50, 1.5); // insert 3 nodes
        theTree.insert(25, 1.7);
        theTree.insert(75, 1.9);
        node found = theTree.find(25); // find node with key 25
        if(found != null)
        System.out.println("Found the node with key 25");
        else
        System.out.println("Could not find node with key 25");
    } // end main()
} // end class TreeApp
```

© 2008, University of Colombo School of Computing

5.3. Searching a Binary tree

- **Finding a Node**
  - Finding a node with a specific key is the simplest of the major tree operations, so let's start with that.
  - Remember that the nodes in a binary search tree correspond to objects containing information.
  - They could be *person objects*, with an employee number as the key and also perhaps name, address, telephone number, salary, and other fields.
  - Or they could represent car parts, with a part number as the key value and fields for quantity on hand, price, and so on.
  - However, the only characteristics of each node that we can see in the Workshop applet are a number and a color. A node is created with these two characteristics and keeps them throughout its life.

© 2008, University of Colombo School of Computing

5.3. Searching a Binary tree contd…

- **Java Code for Finding a Node**

Here's the code for the find() routine, which is a method of the Tree class:

```
public Node find(int key) // find node with
given key
{ // (assumes non-empty tree)
Node current = root; // start at root
while(current.iData != key) // while no
match,
{
if(key < current.iData) // go left?
current = current.leftChild;
else
current = current.rightChild; // or go right?
if(current == null) // if no child,
return null; // didn't find it
}
return current; // found it
}
```

© 2008, University of Colombo School of Computing

5.3. Searching a Binary tree contd…

- This routine uses a variable current to hold the node it is currently examining.
- The argument key is the value to be found. The routine starts at the root. (It has to; this is the only node it can access directly.) That is, it sets current to the root.
- Then, in the while loop, it compares the value to be found, key, with the value of the iData field (the key field) in the current node. If key is less than this field, then current is set to the node's left child.
- If key is greater than (or equal) to the node's iData field, then current is set to the node's right child.

© 2008, University of Colombo School of Computing

5.4. Ways of traversing a tree

- **Tree-traversal** refers to the process of visiting each node in a tree data structure, exactly once, in a systematic way. Such traversals are classified by the order in which the nodes are visited.

© 2008, University of Colombo School of Computing

5.4. Ways of traversing a tree contd…

- **Traversal methods**

  Compared to linear data structures like linked lists and one dimensional arrays, which have only one logical means of traversal, tree structures can be traversed in many different ways. Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed define the traversal type. These steps (in no particular order) are: performing an action on the current node (referred to as "visiting" the node), traversing to the left child node, and traversing to the right child node.

© 2008, University of Colombo School of Computing

5.4. Ways of traversing a tree contd…

- To traverse a non-empty binary tree in **preorder**, perform the following operations recursively at each node, starting with the root node:
  1. Visit the node.
  2. Traverse the left subtree.
  3. Traverse the right subtree. (This is also called Depth-first traversal.)

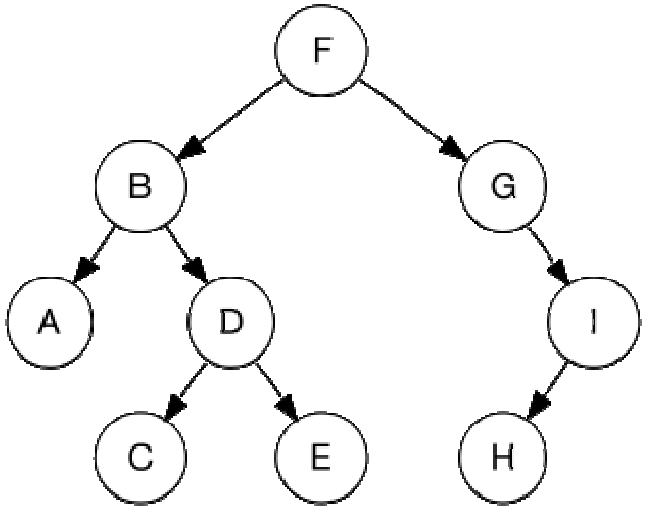
- To traverse a non-empty binary tree in **inorder**, perform the following operations recursively at each node, starting with the root node:
  1. Traverse the left subtree.
  2. Visit the node.
  3. Traverse the right subtree.
- To traverse a non-empty binary tree in **postorder**, perform the following operations recursively at each node, starting with the root node:
  1. Traverse the left subtree.
  2. Traverse the right subtree.
  3. Visit the node.

- Finally, trees can also be traversed in **level-order**, where we visit every node on a level before going to a lower level. This is also called Breadth-first traversal.

© 2008, University of Colombo School of Computing

5.4. Ways of traversing a tree contd…

- **Example**



In this binary search tree,

- Preorder traversal sequence: F, B, A, D, C, E, G, I, H
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I
  - Note that the inorder traversal of this binary search tree yields an ordered list
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F
- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

© 2008, University of Colombo School of Computing

5.4.1. **Breadth-first search**

- **breadth-first search** (**BFS**) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

© 2008, University of Colombo School of Computing

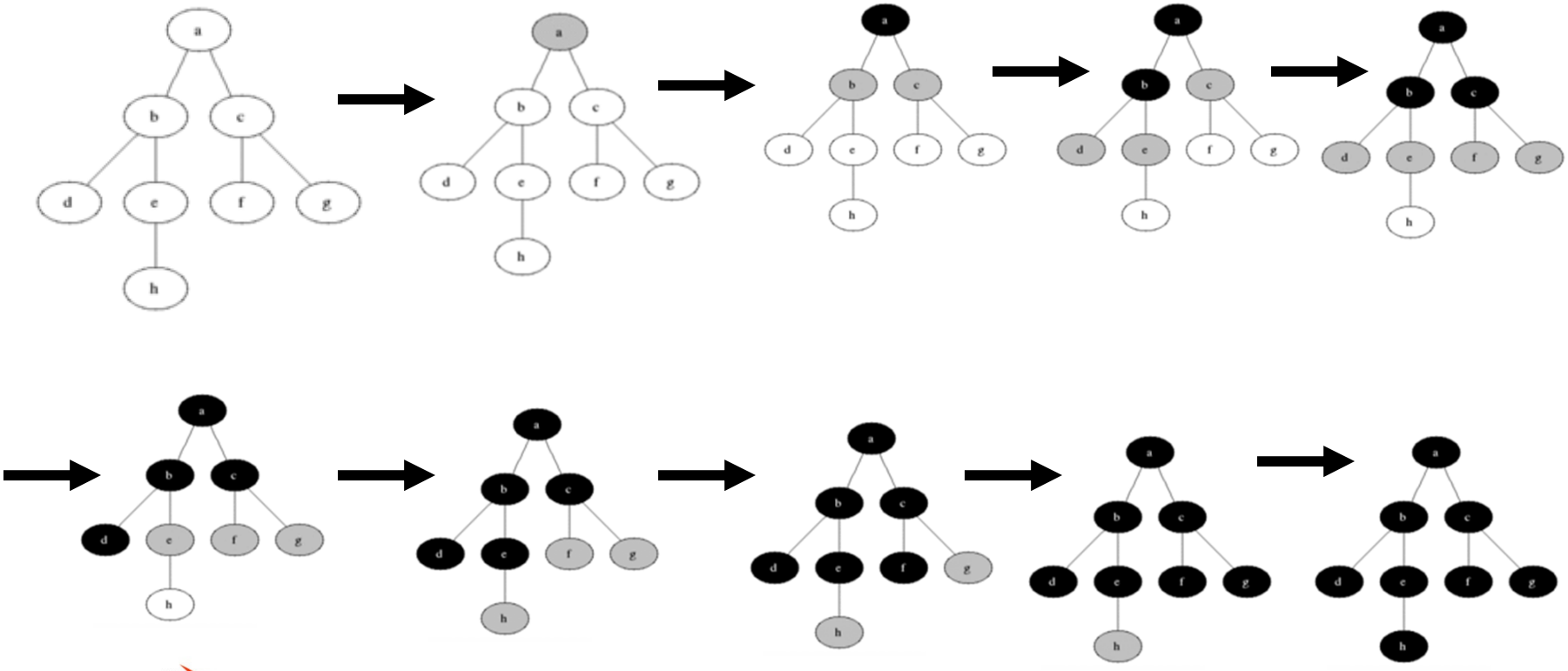
5.4.1. **Breadth-first search**

- **How it works**
  - **Breadth-first search** (BFS) is an uninformed search method that aims to expand and examine all nodes of a graph systematically in search of a solution. In other words, it exhaustively searches the entire graph without considering the goal until it finds it. It does not use a heuristic.
  - From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

© 2008, University of Colombo School of Computing

5.4.1. **Breadth-first search**

- **How it works**



© 2008, University of Colombo School of Computing

5.4.1. **Breadth-first search**

- **Applications of BFS**
  - Breadth-first search can be used to solve many problems in graph theory, for example:
  - Finding all connected components in a graph.
  - Finding all nodes within one connected component
  - Copying Collection, Cheney's algorithm
  - Finding the shortest path between two nodes *u* and *v* (in an unweighted graph)
  - Testing a graph for bipartiteness
  - (Reverse) Cuthill–McKee mesh numbering

© 2008, University of Colombo School of Computing

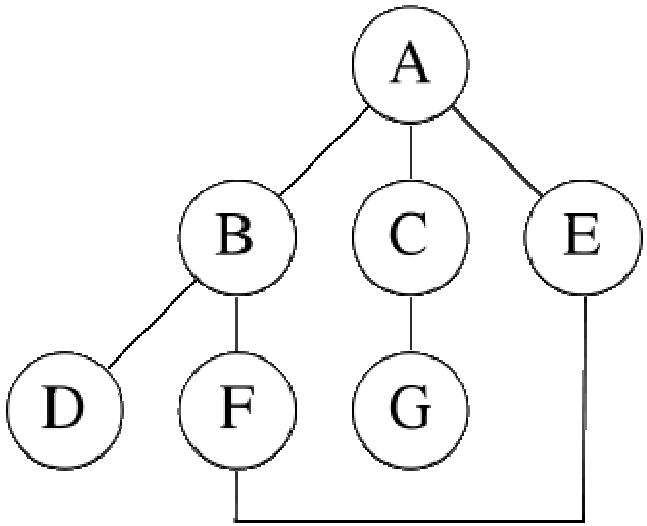
5.4.2. Depth-first search

- **Depth-first search** (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.
- Formally, DFS is an uninformed search that progresses by expanding the first child node of the search tree that appears and thus going deeper and deeper until a goal node is found, or until it hits a node that has no children. Then the search backtracks, returning to the most recent node it hadn't finished exploring. In a non-recursive implementation, all freshly expanded nodes are added to a LIFO stack for exploration.

© 2008, University of Colombo School of Computing

5.4.2. Depth-first search

- **How it works**



See next slide:

© 2008, University of Colombo School of Computing

5.4.2. Depth-first search

- a depth-first search starting at A, assuming that the left edges in the shown graph are chosen before right edges, and assuming the search remembers previously-visited nodes and will not repeat them (since this is a small graph), will visit the nodes in the following order: A, B, D, F, E, C, G.
- Performing the same search without remembering previously visited nodes results in visiting nodes in the order A, B, D, F, E, A, B, D, F, E, etc. forever, caught in the A, B, D, F, E cycle and never reaching C or G.
- Iterative deepening prevents this loop and will reach the following nodes on the following depths, assuming it proceeds left-to-right as above:
  - 0: A
  - 1: A (repeated), B, C, E
- (Note that iterative deepening has now seen C, when a conventional depth-first search did not.)
  - 2: A, B, D, F, C, G, E, F
- (Note that it still sees C, but that it came later. Also note that it sees E via a different path, and loops back to F twice.)
  - 3: A, B, D, F, E, C, G, E, F, B
- For this graph, as more depth is added, the two cycles "ABFE" and "AEFB" will simply get longer before the algorithm gives up and tries another branch.

© 2008, University of Colombo School of Computing

5.4.3. **Stackless Depth-First Traversal**

- Threaded trees allow you traverse the tree by following pointers stored within the tree
- Each node would store pointers to its predecessor and successor
- This would create a lot of overhead with the additional two pointers for a total of 4 pointers per node

© 2008, University of Colombo School of Computing

5.5. Insertion and deletion

- **Inserting a Node**
  - To insert a node we must first find the place to insert it. This is much the same process as trying to find a node that turns out not to exist, as described in the section on Find.
  - We follow the path from the root to the appropriate node, which will be the parent of the new node.
  - Once this parent is found, the new node is connected as its left or right child, depending on whether the new node's key is less than or greater than that of the parent.

© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

- **Java Code for Inserting a Node**
  - The insert() function starts by creating the new node, using the data supplied as arguments.
  - Next, insert() must determine where to insert the new node. This is done using roughly the same code as finding a node, described in the section on find(). The difference is that when you are simply trying to *find* a node and you encounter a null (nonexistent) node, you know the node you are looking for doesn't exist so you return immediately. When you're trying to *insert* a node you insert it (creating it first, if necessary) before returning.

© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

- The value to be searched for is the data item passed in the argument id.
- The while loop uses true as its condition because it doesn't care if it encounters a node with the same value as id; it treats another node with the same key value as if it were simply greater than the key value. (We'll return to the subject of duplicate nodes later in this chapter.)

© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

- A place to insert a new node will always be found (unless you run out of memory); when it is, and the new node is attached, the while loop exits with a return statement.
- Here's the code for the insert() function:

© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

```
public void insert(int id, double dd){
      Node newNode = new Node(); // make new node
      newNode.iData = id; // insert data
      newNode.dData = dd;
      if(root==null) // no node in root
            root = newNode;
      else // root occupied {
      Node current = root; // start at root
      Node parent;
      while(true) // (exits internally)
      {
            parent = current;
            if(id < current.iData) // go left?
            {
```

© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

```
            current = current.leftChild;
            if(current == null) // if end of the line,
            { // insert on left
            parent.leftChild = newNode;
            return;
            }
      } // end if go left
      else // or go right?
      {
            current = current.rightChild;
            if(current == null) // if end of the line
                  { // insert on right
                  parent.rightChild = newNode;
                  return;
                  }
            } // end else go right
            } // end while
      } // end else not root
      } // end insert()
```

© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

- **Deletion**
  - The algorithm to delete an arbitrary node from a binary tree is deceptively complex, as there are many special cases. The algorithm used for the delete function splits it into two separate operations, searching and deletion. Once the node which is to be deleted has been determined by the searching algorithm, it can be deleted from the tree. The algorithm must ensure that when the node is deleted from the tree, the ordering of the binary tree is kept intact.
  - Special Cases that have to be considered:

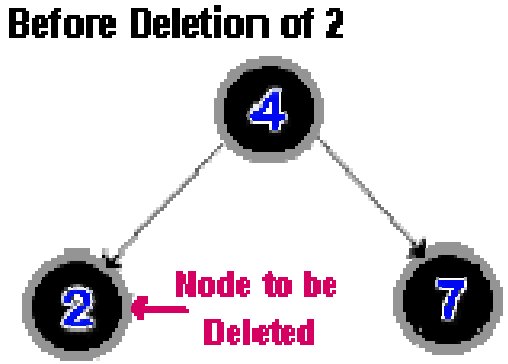
© 2008, University of Colombo School of Computing

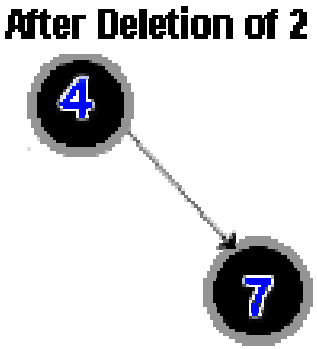
5.5. Insertion and deletion contd…

- **Deletion**

1. *The node to be deleted has no children.*

   In this case the node may simply be deleted from the tree.





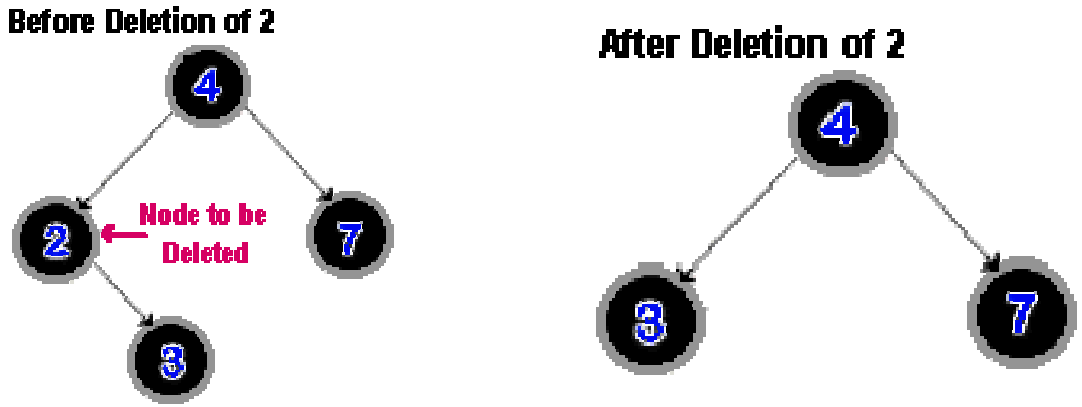
© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

- **Deletion contd…**

  2. The node has one child.

  The child node is appended to its grandparent. (The parent of the node to be deleted.)



© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

- **Deletion contd…**

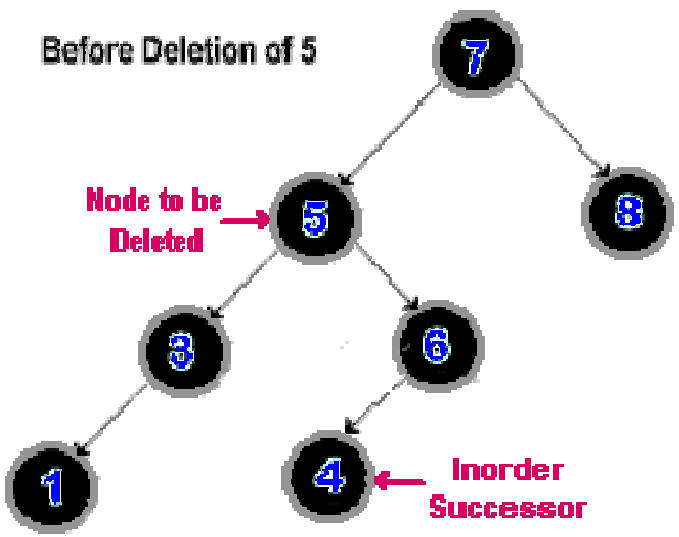
3. The node to be deleted has two children.

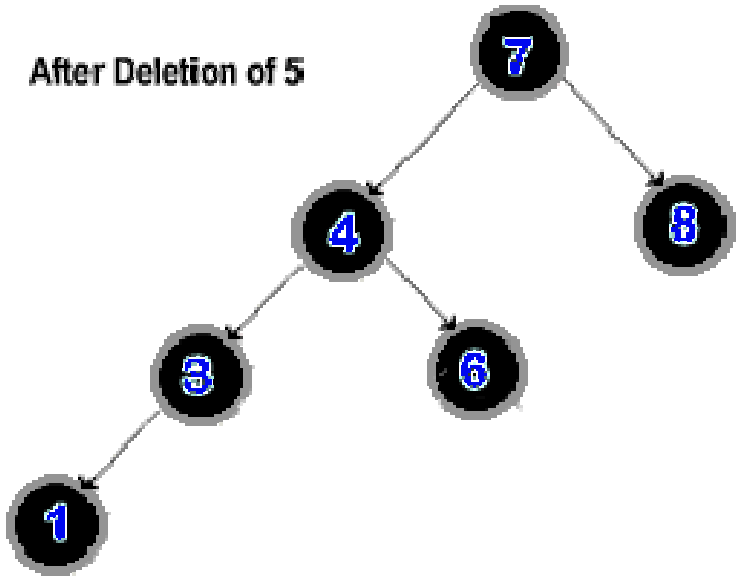
This case is much more complex than the previous two, because the order of the binary tree must be kept intact. The algorithm must determine which node to use in place of the node to be deleted:

© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

(i) Use the inorder successor of the node to be deleted.

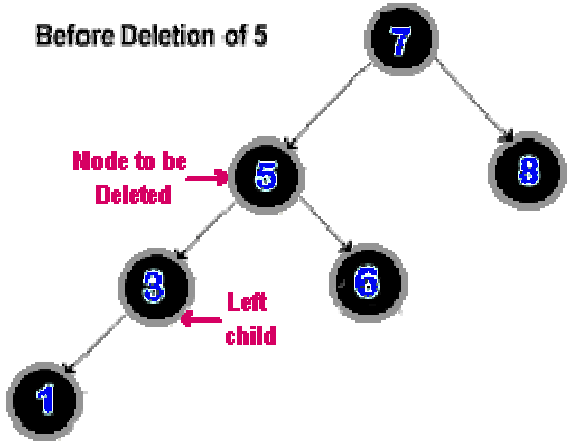


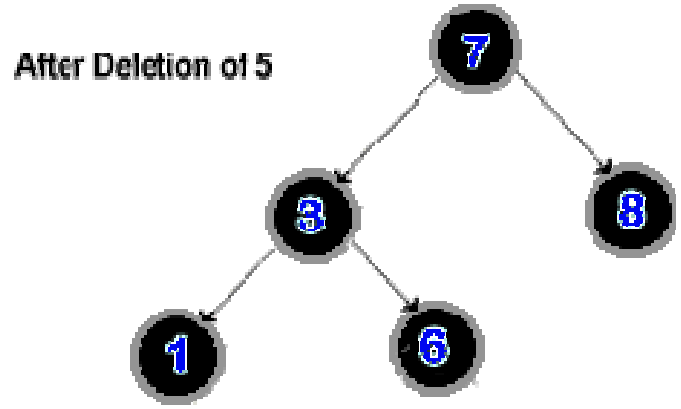


© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

(ii) Else if no right subtree exists replace the node to be deleted with the it's left child.





© 2008, University of Colombo School of Computing

5.5. Insertion and deletion contd…

- Deletion of the root node is also a special case. It can be accomplished using the methods described above, checking for the separate cases with no children, two children, or one.
- ***Complexity***
  - Average case is O(log2n).
  - Worst case is O(n).

© 2008, University of Colombo School of Computing