Lecture 6- Trees and Binary Trees

Data Structure and Algorithm Analysis

Trees

- Linear access time of linked lists is expensive
  - Requires O(N) running time for most of basic operations like search, insert and delete
- Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(log N)?
  - The answer is yes
    - Data structures like binary tree

Trees…

- A tree is a collection of nodes
  - The collection can be empty
- (recursive definition) If not empty, a tree consists of a distinguished node r (the *root*), and zero or more nonempty *sub-trees* $T_1, T_2, \ldots, T_k$, each of whose roots are connected by an *edge* from r

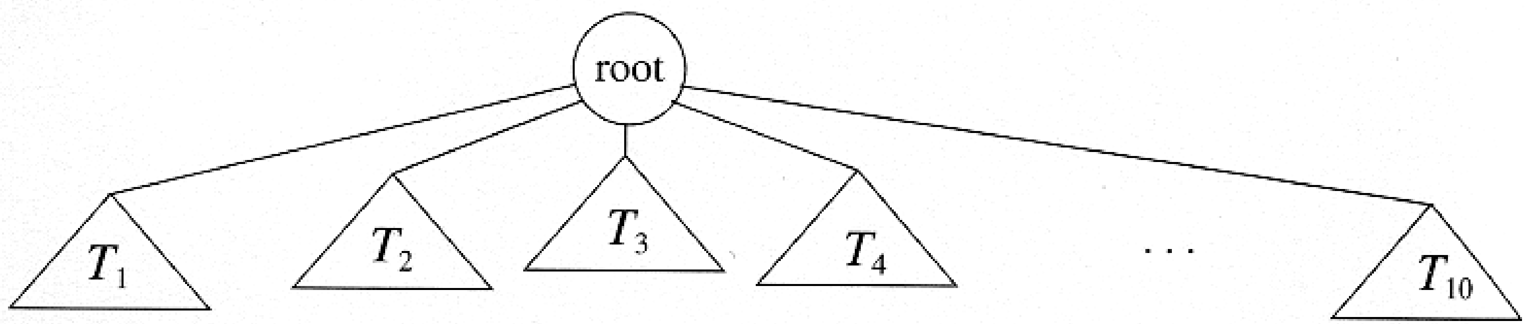
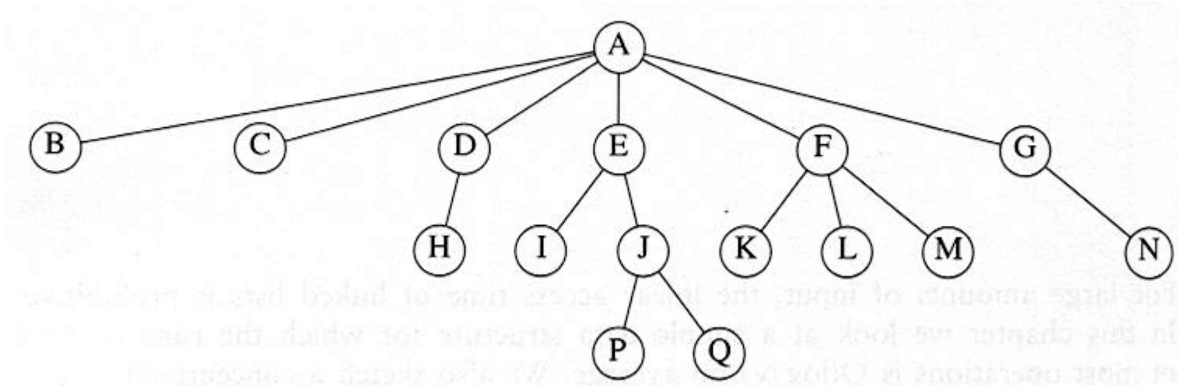


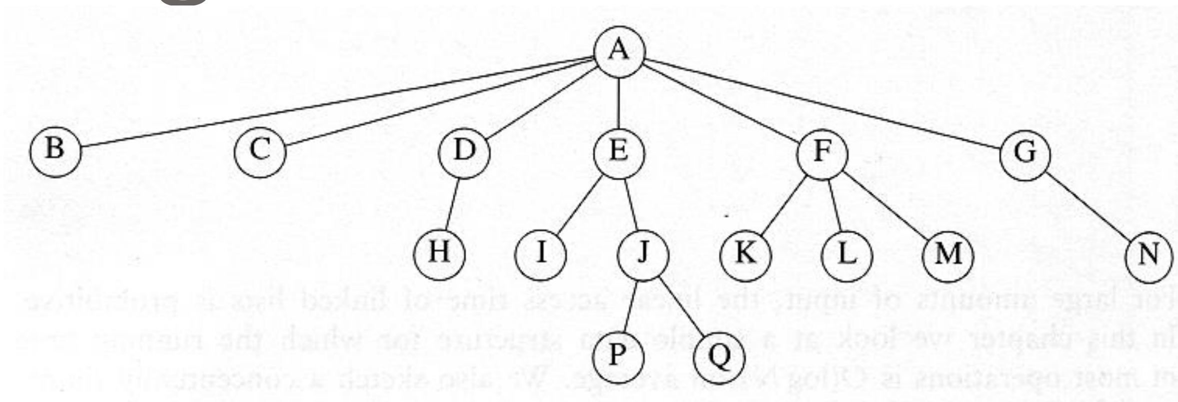
Figure 4.1 Generic tree

Some Terminologies



- *Child* and *parent*
  - Every node except
  - A node can have an arbitrary number of children (A has 6 while D has 1)
- *Leaves/External Nodes*
  - Nodes with no children (B, C, H, I, P, Q, K, L, M, N)
- *Sibling*
  - nodes with same parent (P and Q)
- *Internal node*
  - A node with at least one child (A,D,E,F,G,J)

Some Terminologies…



- *Path*
  - *is a sequence of nodes from root to a node (arbitrary node in the tree).*
- *Length*
  - Number of edges on the path from node x to node y
- *Depth* of a node
  - Number of edges from the root to that node (Depth of C =1)
  - The depth of a tree is equal to the depth of the deepest leaf (=3)

Some Terminologies…

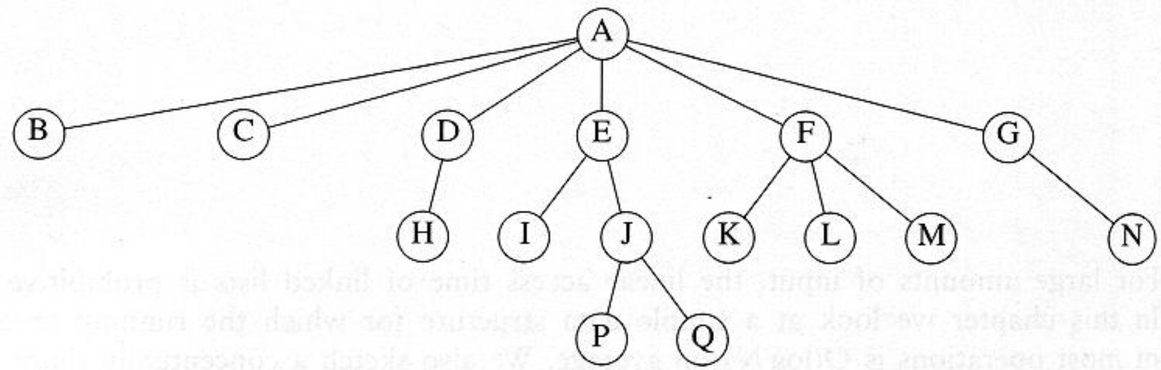
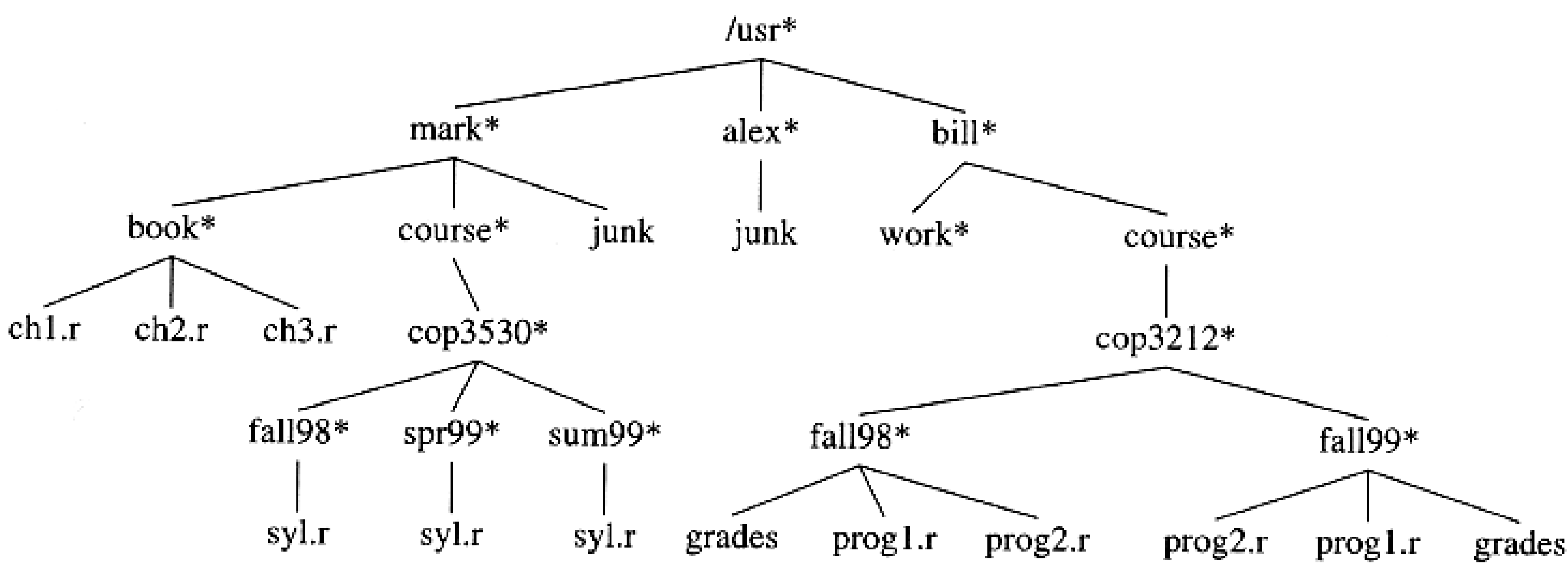


Figure 4.2 A tree

- *Height* of a node
  - length of the longest path from that node to a leaf (E=2)
  - all leaves are at height 0
  - The height of a tree is equal to the height of the root
- *Ancestor* and *descendant*
  - The ancestors of a node are all the nodes along the path from the root to the node.
  - *Descendant* node reachable by repeated proceeding from parent to child.

Example: UNIX Directory

- Tree is useful to represent hierarchical data
- One of its application a file system used by many systems
- The following is an exmple of unix file system



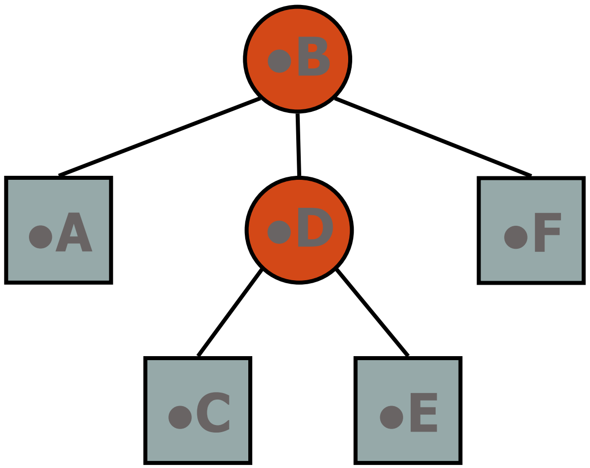
Tree ADT

- We use struct/class to abstract nodes
- Generic methods:
  - integer **size**()
  - boolean **isEmpty**()
  - gisplay**Elements**()
- Accessor methods:
  - Object **root**()
  - Object **parent**(p)
  - display**Children**(p)

- **Query methods:**
  - **boolean isInternal(p)**
  - **boolean isExternal(p)**
  - **boolean isRoot(p)**
- **Update methods:**
  - **swapElements(p, q)**
  - **object replaceElement(p, o)**
- **Additional update methods may be defined by data structures implementing the Tree ADT**

A Tree Representation

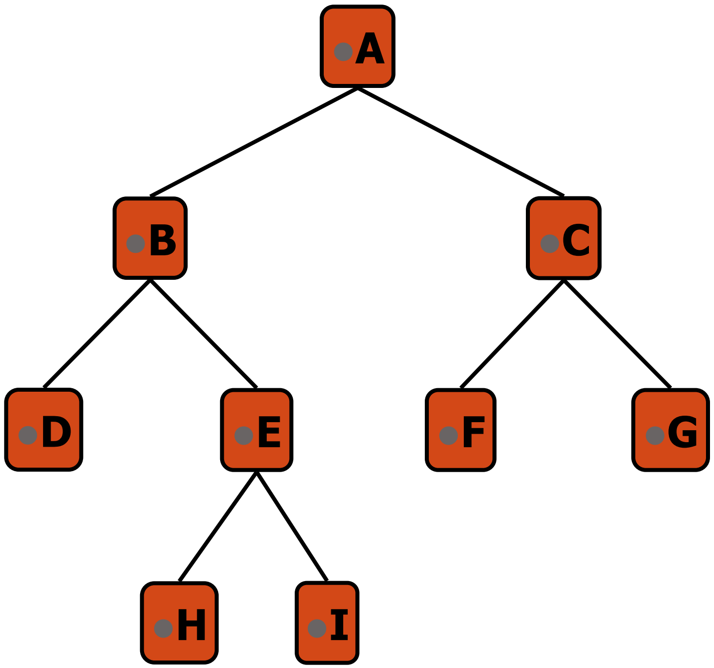
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes



Binary Tree

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (degree of two)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, OR
  - a tree whose root has an ordered pair of children, each of which is a binary tree

- **Applications:**
  - **arithmetic expressions**
  - **decision processes**
  - **searching**

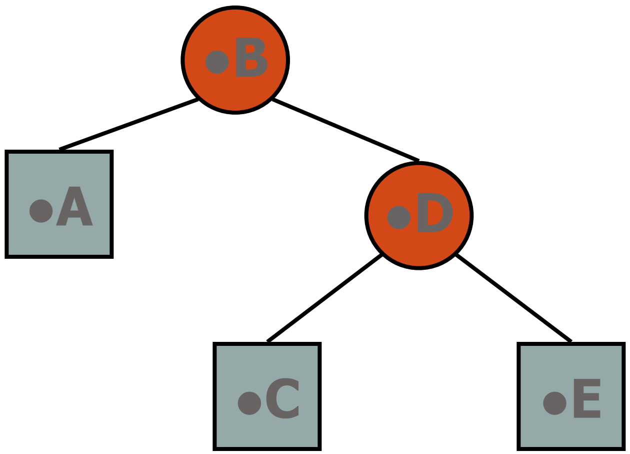


Binary Tree ADT

- The Binary Tree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position leftChild(p)
  - position rightChild(p)
  - position sibling(p)
- Update methods may be defined by data structures implementing the Binary Tree ADT

Data Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node



Example

```
Struct Node{
        Int data;
        Node * parent;
        Node *Lchiled;
        Node * Rchiled;
   }Node *root=NULL;
```

Example: Expression Trees

- One of the application of binary is representing arthematic exression

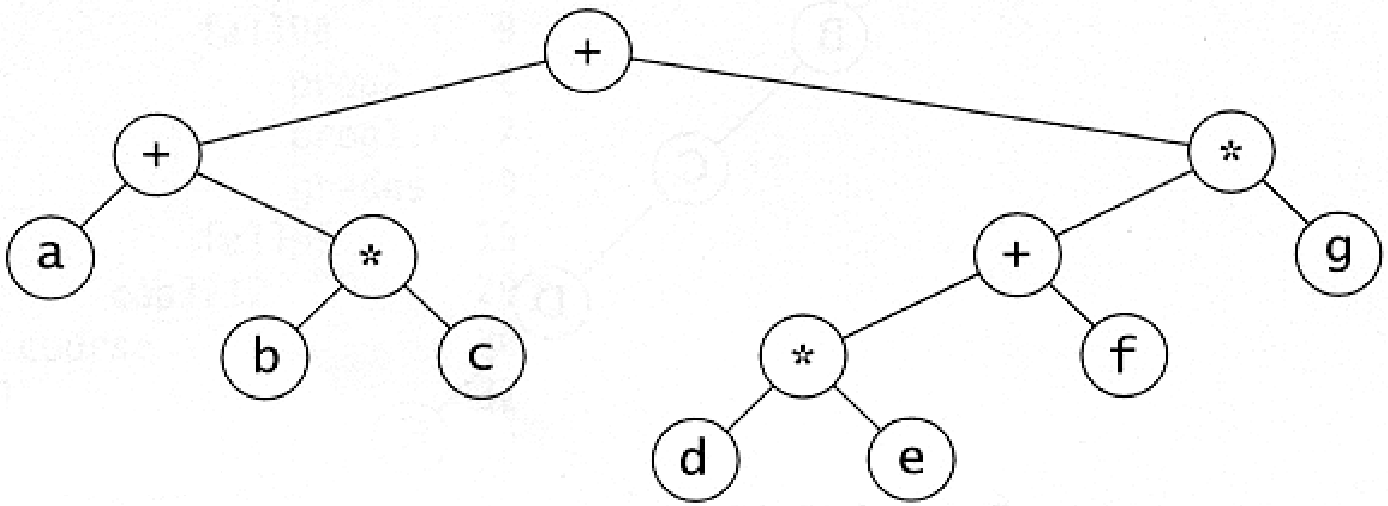


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- Leaves are operands (constants or variables)
- The other nodes (internal nodes) contain operators
- Will not be a binary tree if some operators are not binary

Tree traversal

- Used to print out the data in a tree in a certain order
- Pre-order traversal
  - Print the data at the root
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree

Preorder, Postorder and Inorder

- Preorder traversal
  - node, left, right
  - prefix expression
    - ++a*bc*+*defg

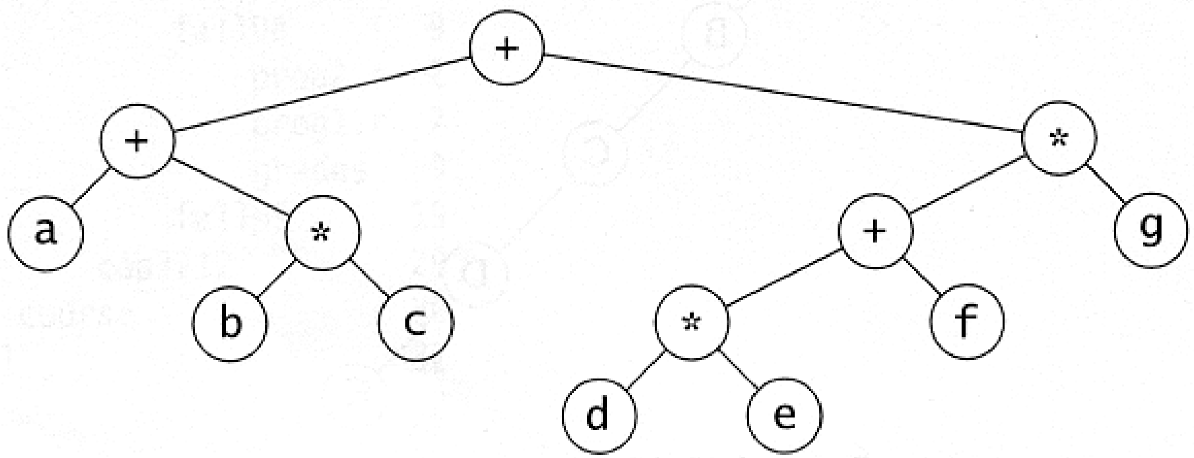


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Preorder, Postorder and Inorder

- Postorder traversal
  - left, right, node
  - postfix expression
    - abc*+de*f+g*+
- Inorder traversal
  - left, node, right.
  - infix expression
    - a+b*c+d*e+f*g

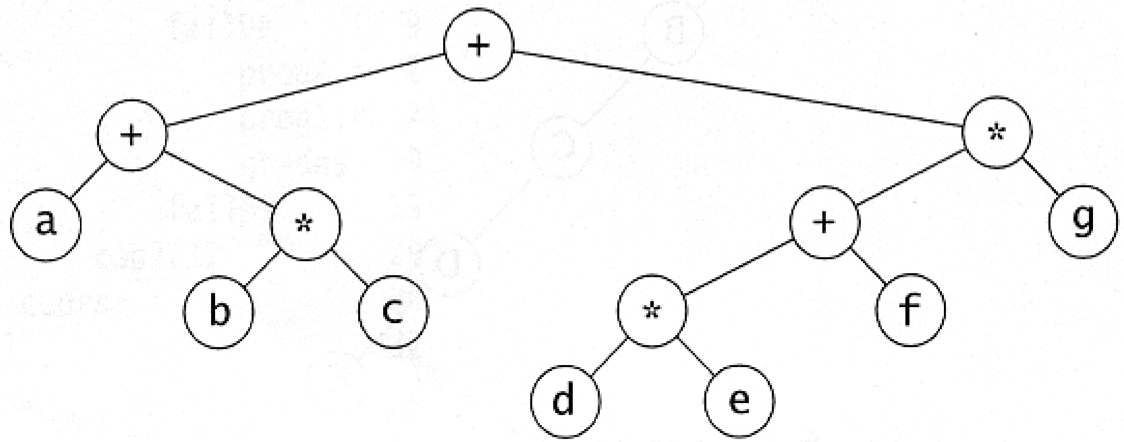


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Preorder, Postorder and Inorder

Algorithm *Preorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** output key(x);
3. *Preorder*(left(x));
4. *Preorder*(right(x));

Algorithm *Inorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** *Inorder*(left(x));
3. output key(x);
4. *Inorder*(right(x));

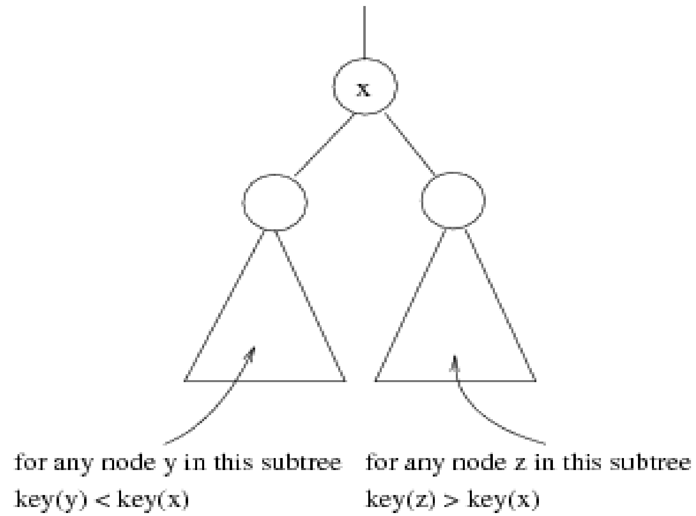
Algorithm *Postorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** *Postorder*(left(x));
3. *Postorder*(right(x));
4. output key(x);

Binary Search Trees

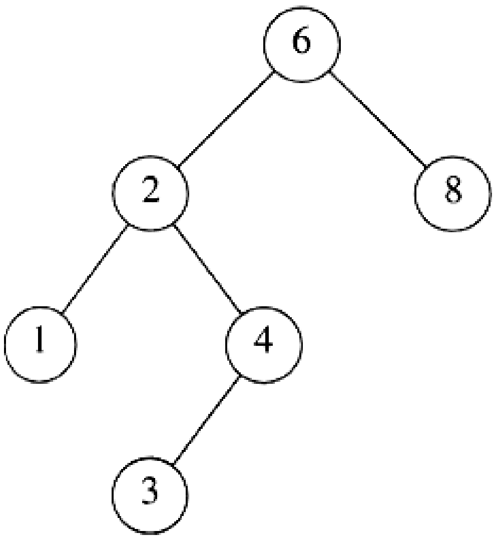
- Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.

Binary search tree property

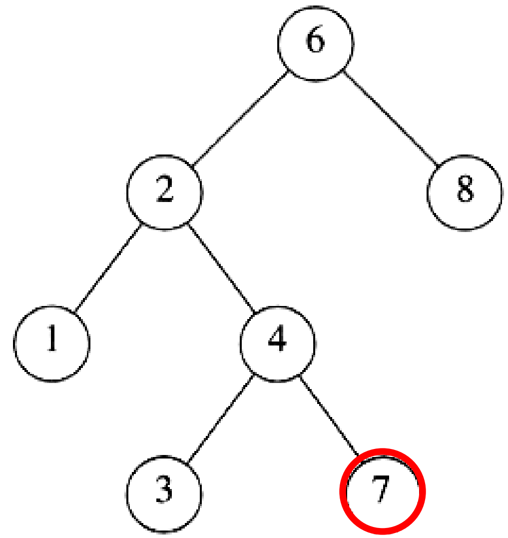
- For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X



Binary Search Trees…



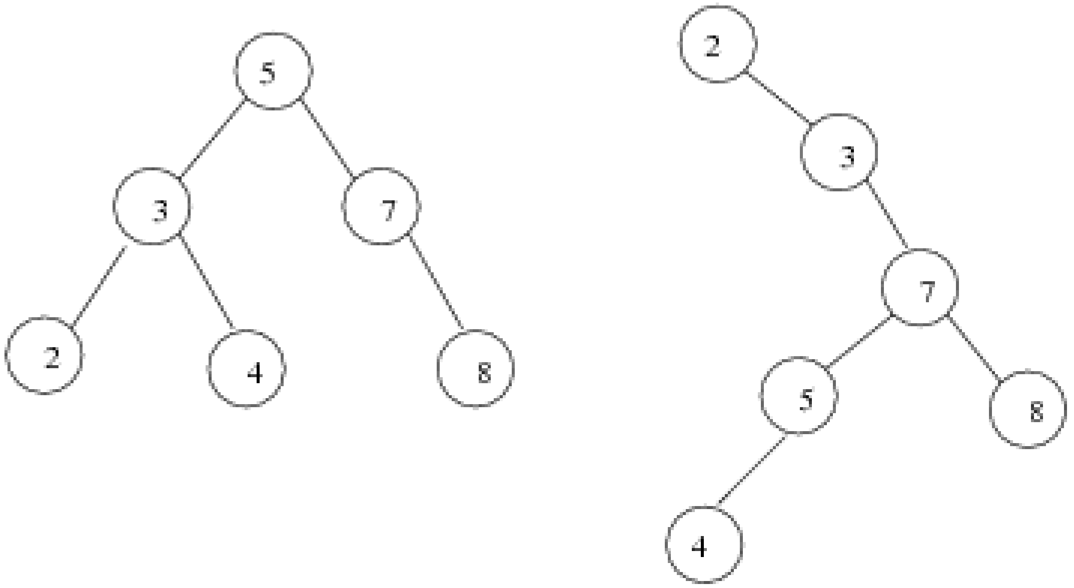
A binary search tree



Not a binary search tree

Binary search trees…

Two binary search trees representing the same set:



- Average depth of a node is O(log N); maximum depth of a node is O(N)

Implementation of BST

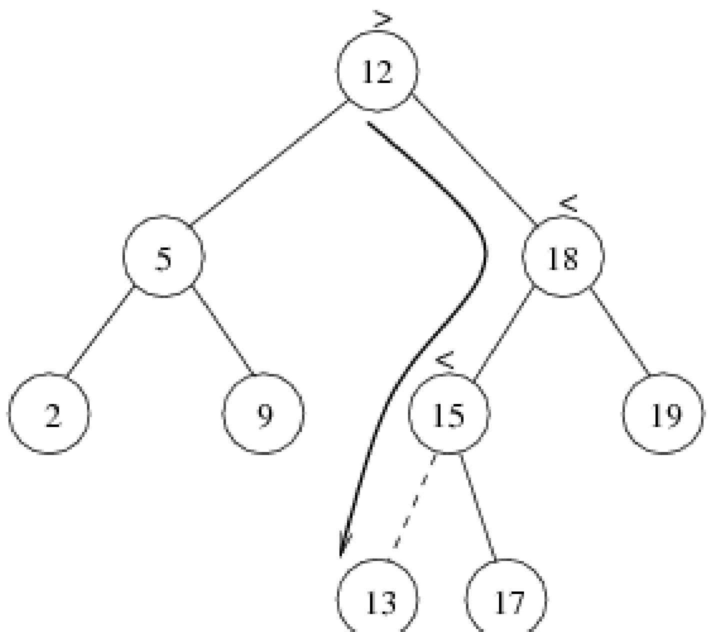
```
Struc node
{
   Int num;
   Node * parent
   Node*left;
   Node * right;
}
Node *root=NULL;
```

Inserting node in BST

- When a new node is inserted the definition of BST should be preserved.
- There are two cases to consider
  - There is no data in the tree (root=null)
    - Root=newnode;
  - There is data
    - Search the appropriate position
    - Insert the node in that position.

Example- insert node13

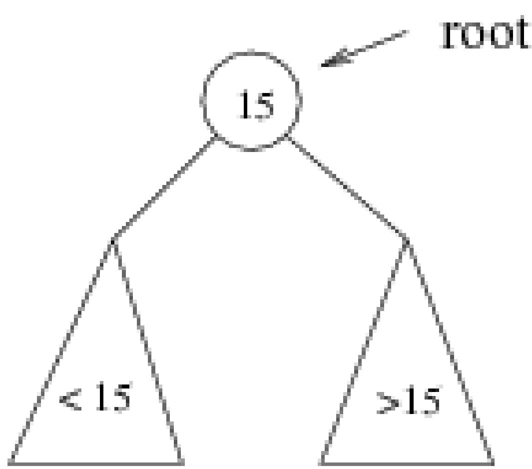
- Proceed down the tree as you would with a find
- If newnode is found, do nothing (or update something)
- Otherwise, insert newnode at the last spot on the path traversed



- Time complexity = O(height of the tree)

Searching BST

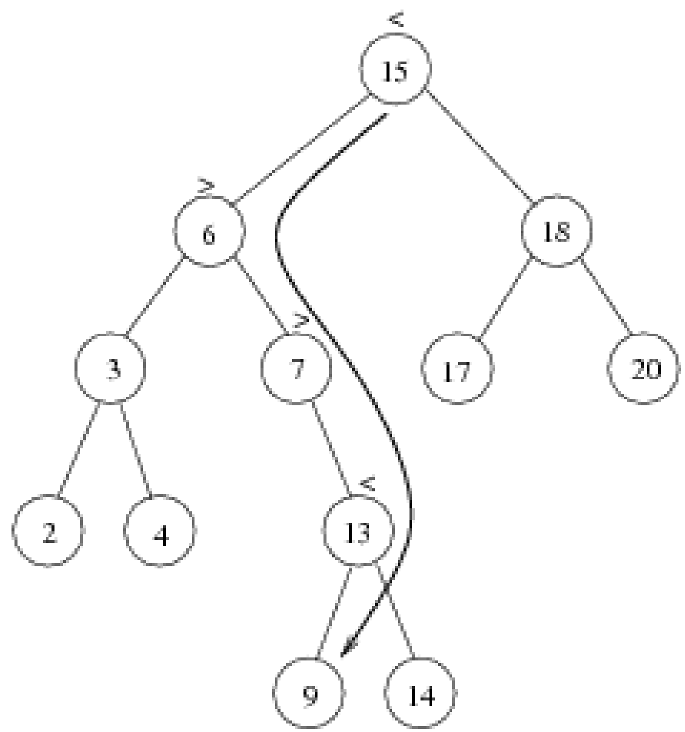
- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



Example: Search for 9 ...

Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!



Searching (Find)

- Find X: return a pointer to the node that has key X, or NULL if there is no such node

- Node *searchBST(node *root, int x)

```
{
  If(root==NULL || root->num==x)
     Return (root)
  Else if(root->num>x)
     Return (searchBST(root->left, x)
  Else
     Return (searchBST(root->right, x))
     }
```

- Time complexity
  - O(height of the tree)

findMin

- Return the node containing the smallest element in the tree
- Start at the root and go left as long as there is a left child. The stopping point is the smallest element

```
Node*findMin(node*root)
{
    If(root==NULL)
        Return Null;
    Ellse if(root->left==Null)
        Return root
    Else
        Return(findMin(root->left)
}
```

- Similarly for findMax
- Time complexity = O(height of the tree)

findMax

- Finds the maximum element in BST
- Start at the root and go right as long as there is a right child. The stopping point is the largest element

```
Node*findMin(node*root)
{
    If(root==NULL)
        Return Null;
    Ellse if(root->right==Null)
        Return root
    Else
        Return(findMin(root->right)
}
```

delete

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - When a node is deleted the definition of a BST should be maintained.
- When a node is deleted four cases should be considered
  - Case1: Deleting a leaf node (a node with no chiled)
  - Case2: Deleting a node having only one child
  - Case3: Deleting a node having two child
  - Case4: Deleting a root node

delete

Three cases:

(1) the node is a leaf

- Delete it immediately

(2) the node has one child

- Adjust a pointer from the parent to bypass that node

- Example delete node 4, make node 2 pointer point to node 3

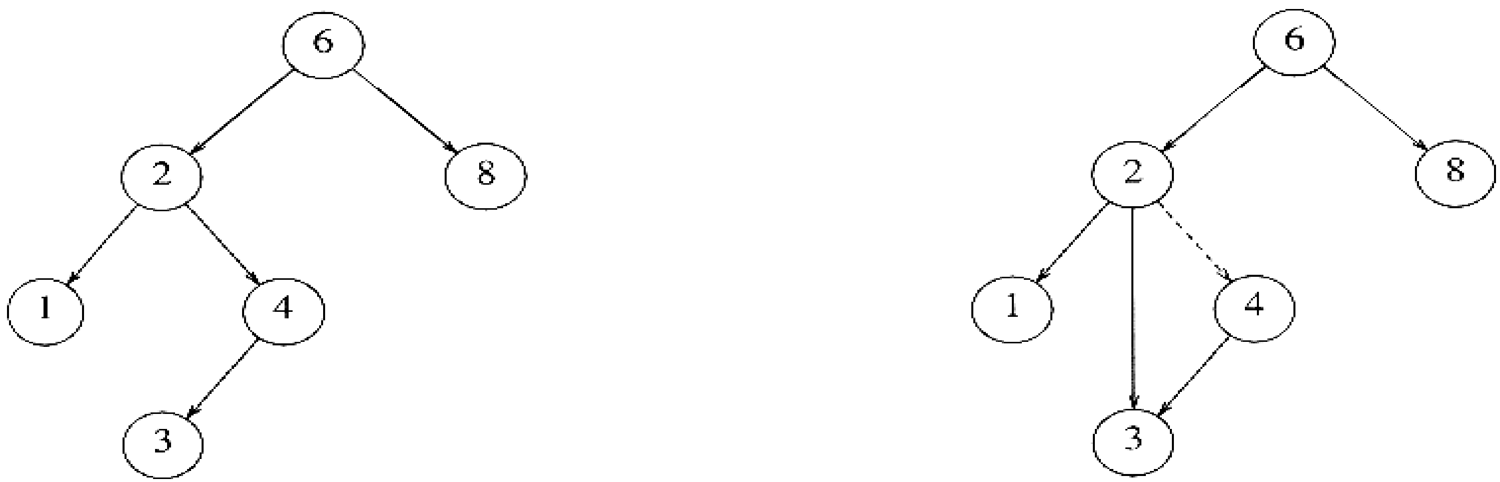


Figure 4.24 Deletion of a node (4) with one child, before and after

delete

(3) the node has 2 children

- Copy the node containing the largest element in the left(or the smallest element in the right)to the node to be deleted
- Delete the copied node

- The picture below shows deleting node2

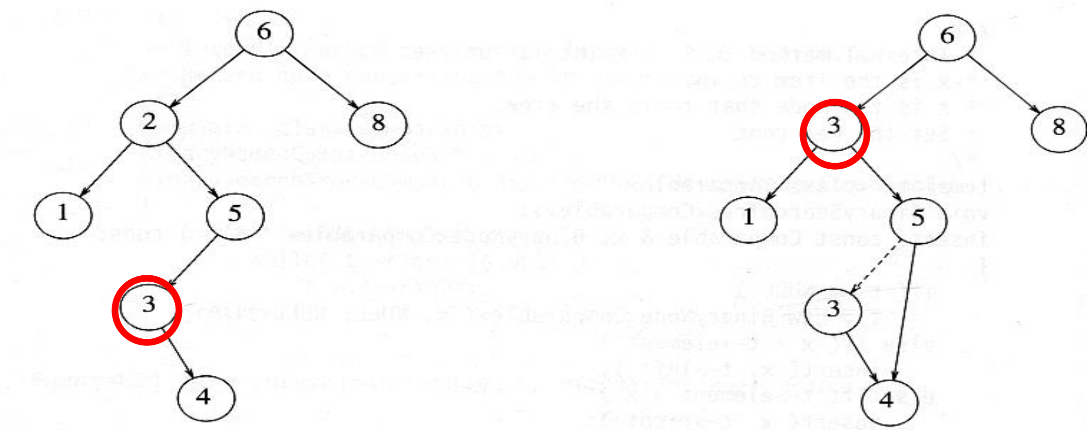


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

Delete the root

- If BST has only one node, make root to point to nothing
  - Root=NULL
- Otherwise,
  - copy the node containing the largest element in the left(or the smallest element in the right)to the node to be deleted
  - Delete the copied node

End of Lecture 6

End of the Course