Chapter 6
Binary Search Tree

By: Dr. Aryaf A. Al-adwan
Faculty of Engineering Technology
Computer and Networks Engineering Dept.
Data Structures Course

Outline

- Trees
- Basic concepts
- Binary tree
- Binary search tree and its operations
- Tree traversal
- BST implementation
- BST Applications

Tree Data Structure

- A tree data structure can be defined recursively as a collection of nodes , starting at a root node.
- A node is an entity that contains a key or value and pointers to its child nodes.

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 *subtrees* T_1, T_2, ..., T_k, each of whose roots are connected by a directed *edge* from r

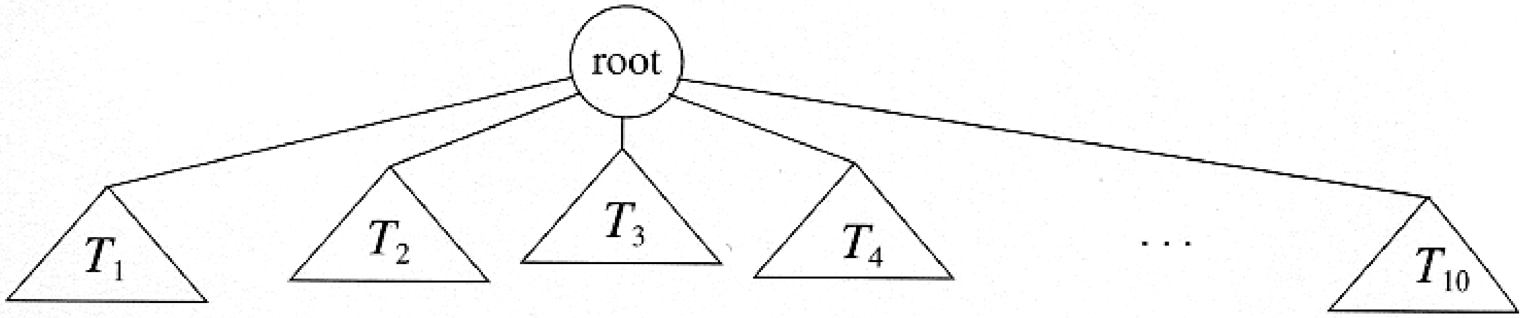


Figure 4.1 Generic tree

Some Terminologies

- *Child* and *Parent*
    - Every node except the root has one parent
    - A node can have an zero or more children
- *Leaves*
    - Leaves are nodes with no children
- *Sibling*
    - nodes with same parent

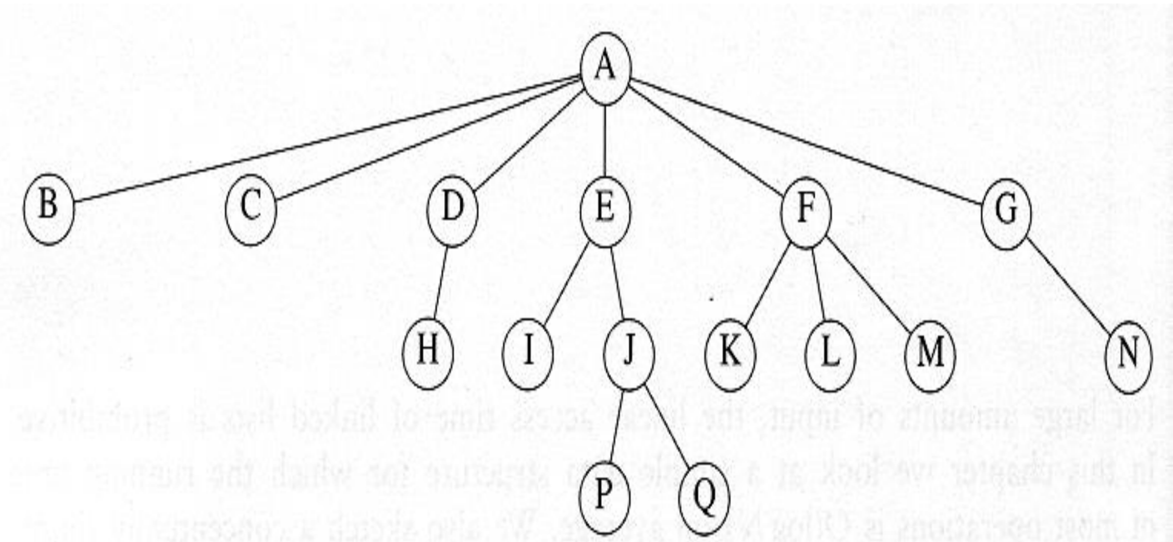
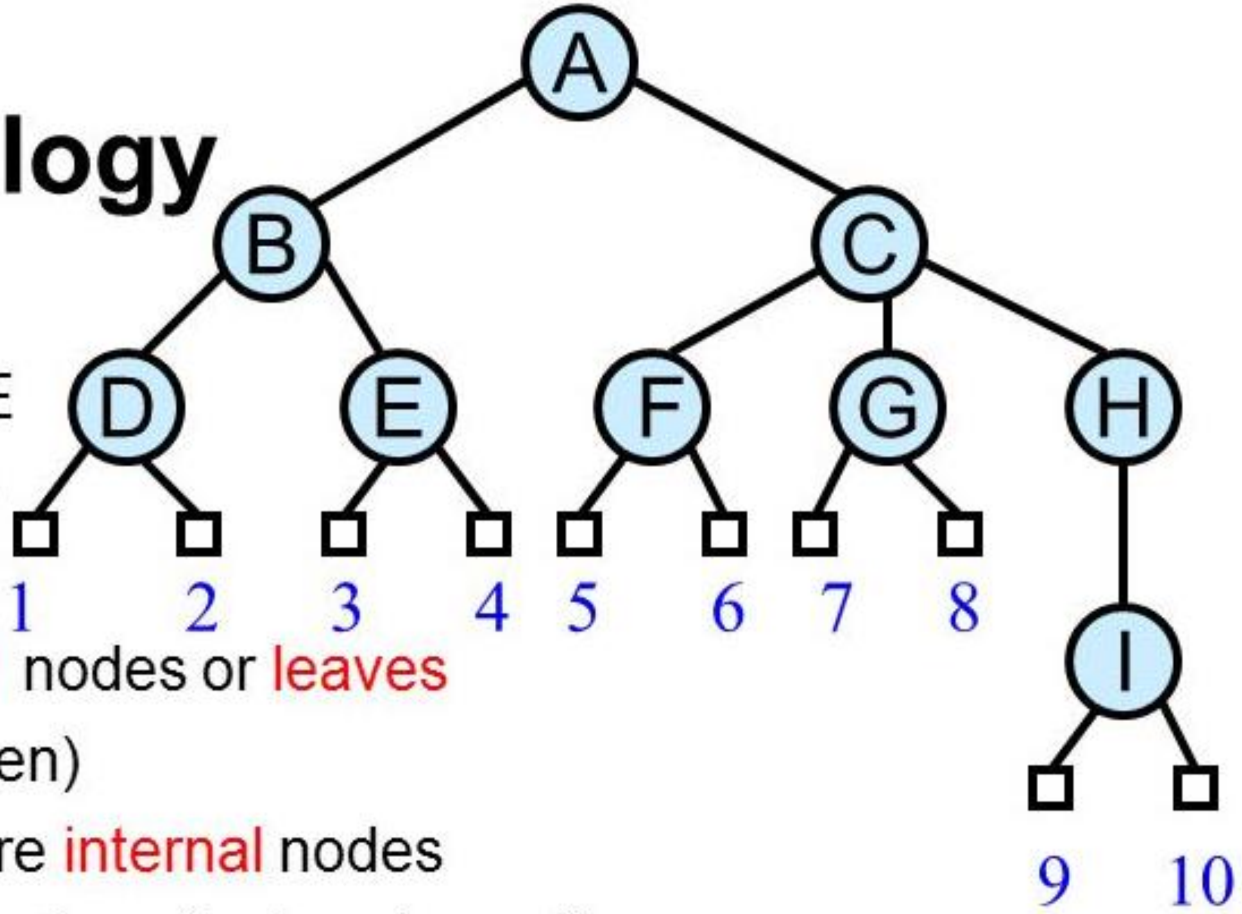


Figure 4.2 A tree

More Terminologies

- ***Path***
  - A sequence of edges
- ***Length of a path***
  - number of edges on the path
- ***Depth* of a node**
  - length of the unique path from the root to that node
- ***Height* of a node**
  - length of the longest path from that node to a leaf
  - all leaves are at height 0
- **The height of a tree** = the height of the root
  = the depth of the deepest leaf
- ***Ancestor* and *descendant***
  - If there is a path from n1 to n2
  - n1 is an ancestor of n2, n2 is a descendant of n1
  - *Proper ancestor* and *proper descendant*

Tree Terminology

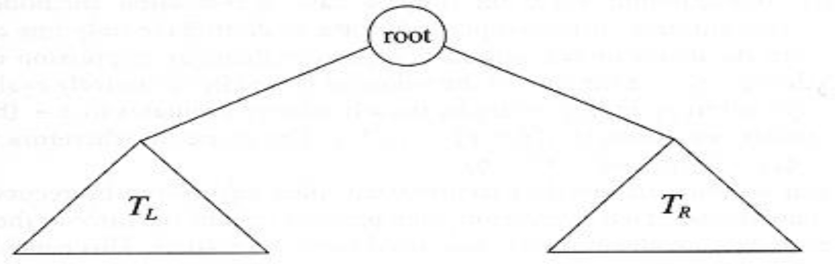


- A is the root node
- B is the parent of D and E
- D and E are children of B
- (C ---- F) is an edge
- 1, 2,…,9, 10 are external nodes or leaves (i.e., nodes with no children)
- A, B, C, D, E, F, G, H, I are internal nodes
- depth (level) of E is 2 (number of edges to root)
- height of the tree is 4 (max number of edges in path from root)
- degree of node B is 2 (number of children)

15/65

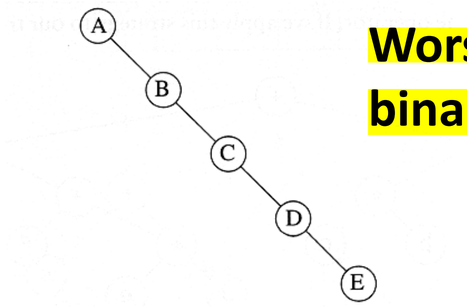
Binary Trees

- A tree in which no node can have more than two children (left child and right child)



Generic binary tree

- The depth of an "average" binary tree is considerably smaller than N, even though in the worst case, the depth can be as large as N – 1.



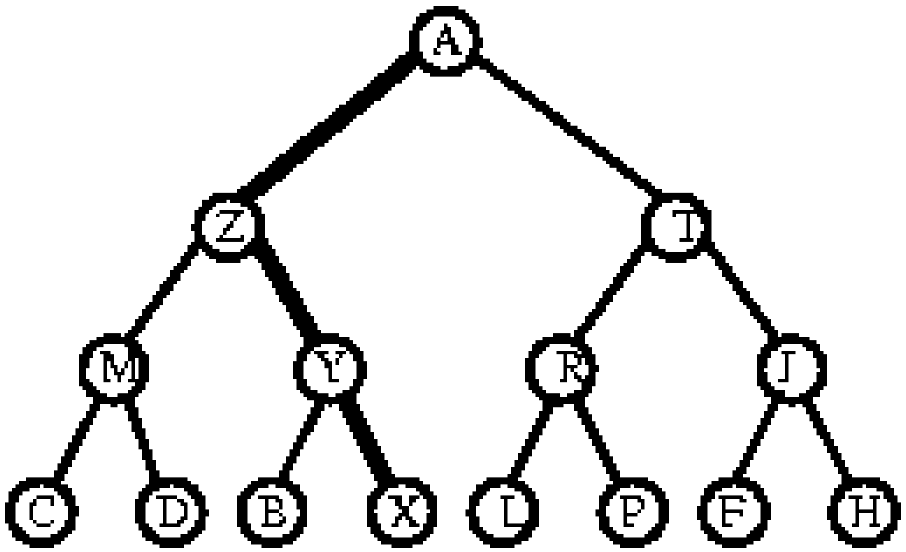
Worst-case binary tree

Types of Binary trees

- Full binary tree
- Complete binary tree
- Balanced binary tree

Full Binary tree

- Every node has exactly two children in all levels except the last level.

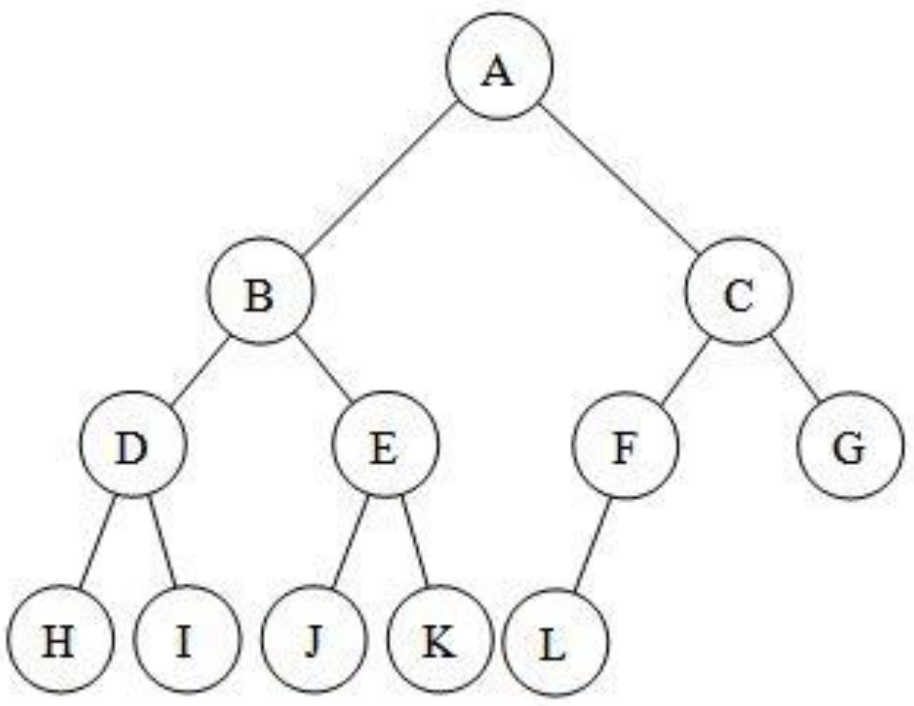


Full Binary tree

- Level i has 2^i nodes
- In a tree of height h
  1. Leaves are at level h
  2. Number of leaves = 2^h
  3. Number of internal nodes = 2^h-1
  4. Number of internal nodes = number of leaves-1
  5. Total number of nodes = $2^{h+1} -1 =n$
- In a tree of n nodes
  1. Number of leaves = n + 1 / 2
  2. Height = $\log_2$ number of leaves

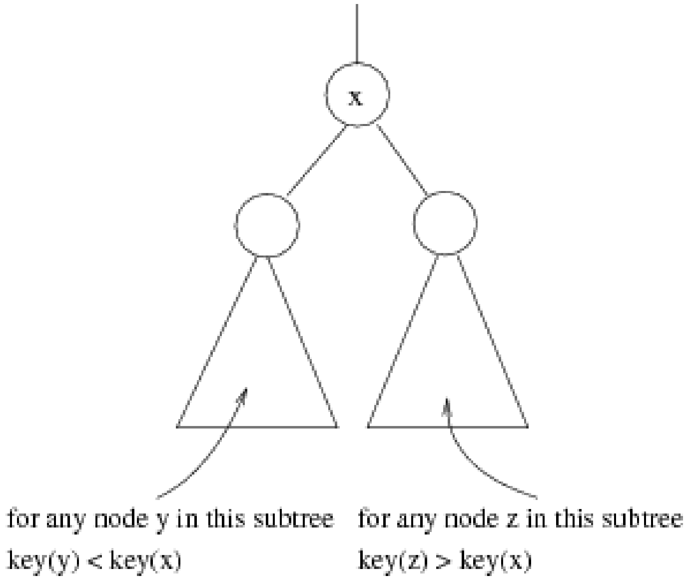
Complete Binary tree

Full up to second last level. And last level is filled from left to right.

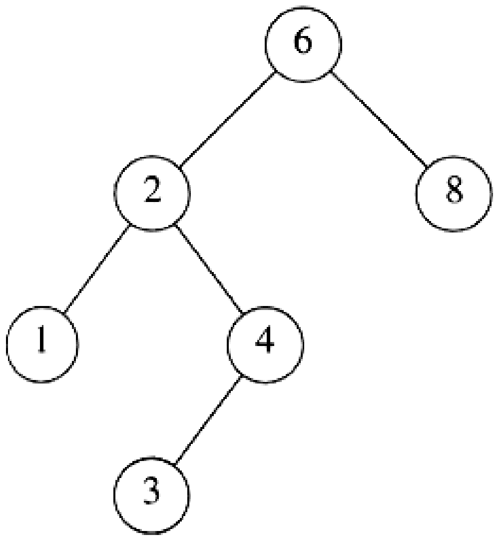


Binary Search Trees (BST)

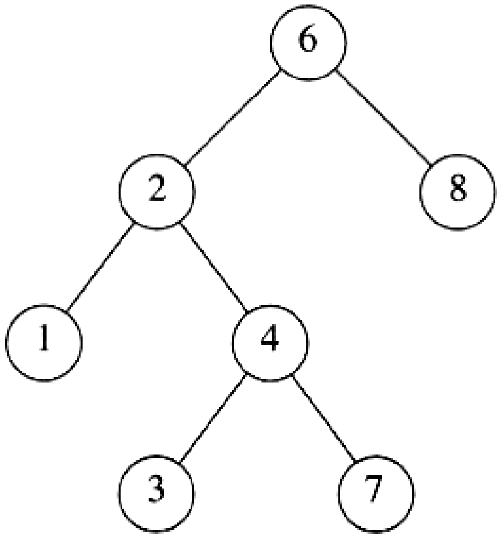
- A data structure for efficient searching, insertion and deletion
- Binary search tree property
  - For every node X
  - All the keys in its left subtree are smaller than the key value in X
  - All the keys in its right subtree are larger than the key value in X
  - Such a property is called an **invariant**



Binary Search Trees



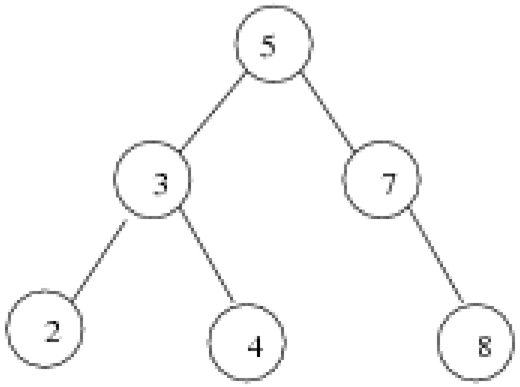
A binary search tree

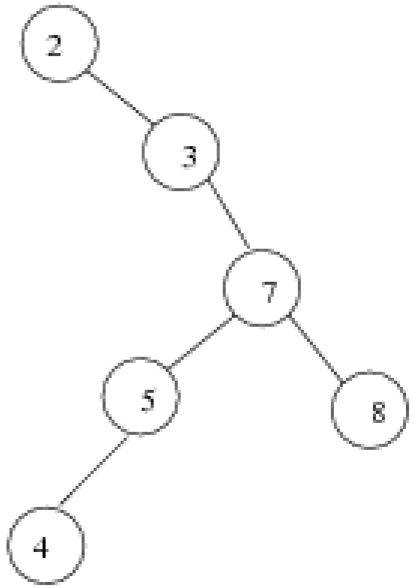


Not a binary search tree

Binary Search Trees

The same set of keys may have different BSTs



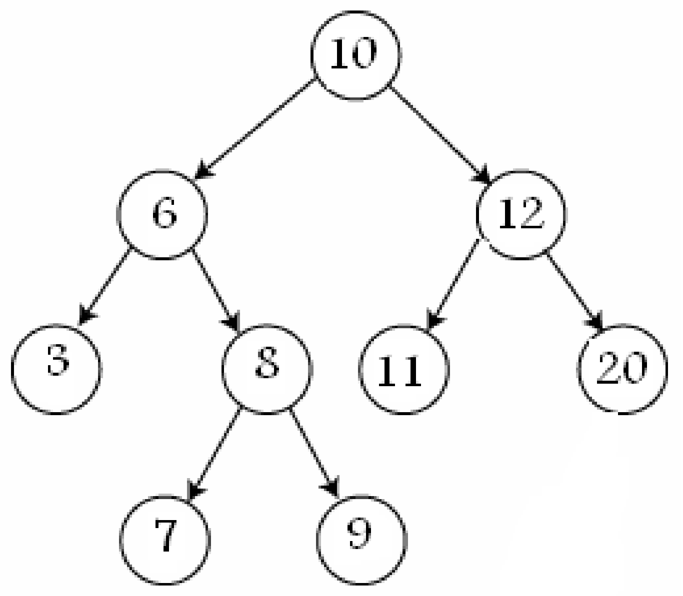


- Average depth of a node is O(log N)
- Maximum depth of a node is O(N)

BST

- Insert the following nodes into binary search tree:

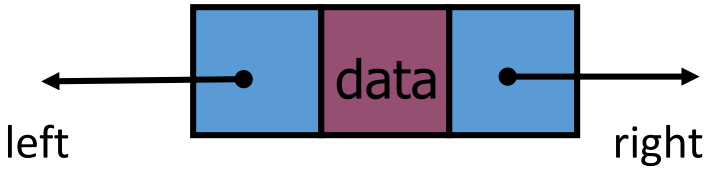
  10 , 6 , 3 , 8 , 7 , 9 , 12 , 11 , 20



BST Implementation

- Using doubly Linked List
- Each node has :

1. Data
2. Right reference
3. Left reference




```
public class TreeNode
{
public int data;
TreeNode  left;
 TreeNode  right;
}
```

BST Class

```
template <class T>
class BinarySearchTree
{
public:
template <class T>
struct node
{
T data;
node *left;
node *right;
};
node <T> *root;
//////////// constructor /////////
BinarySearchTree()
{
root=NULL;
}
public void insert( int item) {   }
public TreeNode  Find(TreeNode  n,int key) {    }
void delet() {   }
public TreeNode FindMin(TreeNode p)   {    }
public void postorder(TreeNode n) {   }
public void preorder(TreeNode n)  {   }
public void inorder(TreeNode n)   {   }
public static void main ( String args [] )  {    }
}
```

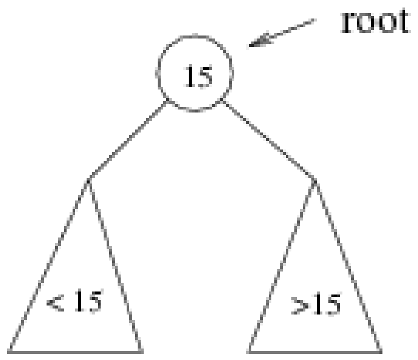
Searching BST

- Algorithm

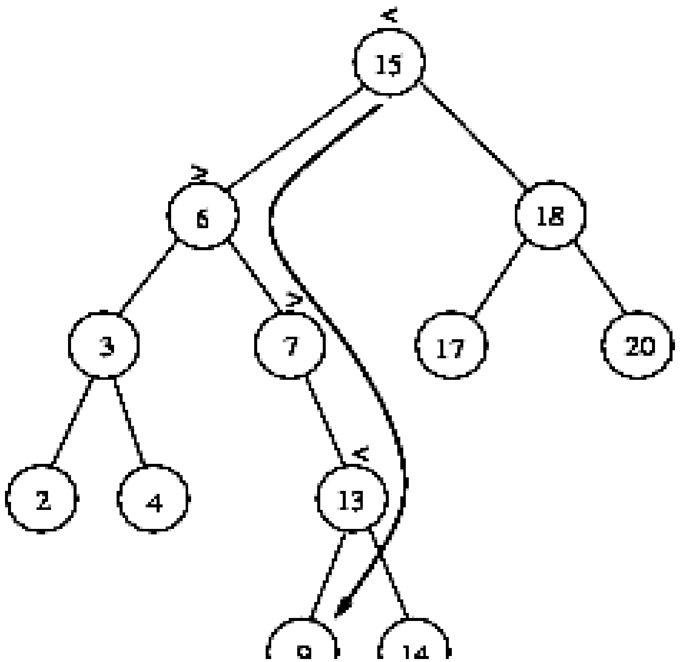
1. Compare the key with the root
   A. If it is equal then it is found
   B. It it is less then repeat in left subtree
   C. If it is greater then repeat in right subtree
2. Otherwise the key not found

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)

```
/////////// this function to search the tree for an element ///////////
node <T> *Find(node <T> *n, int key)
{
node <T> *x;
while (n != NULL)
{
if (n->data == key)      // Found it
{
return n;
}
if (n->data > key)      // In left subtree
{ x=n;
n = n->left;
}
else                    // In right subtree
{ x=n;
n = n->right;
}
}
return x;
}
```

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

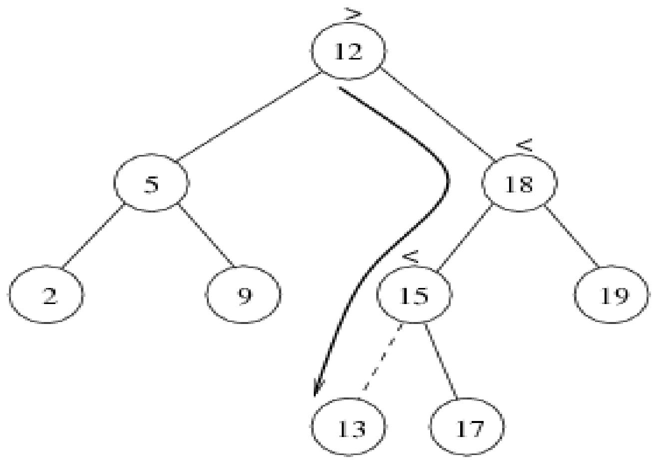
Time complexity: O(height of the tree)

Insertion

- Algorithm

1. Perform search for value x
2. Search will end at node y (if x not in a tree)
3. If x < y then insert x at left subtree for y
4. If x > y then insert x at right subtree of y

- Time complexity = O(height of the tree)



Insert method

```
////////// this is insert function using searching technique //////////
void insert()
{
int numberofnodes;
T num;
cout<<"\nEnter how many elements\n";
cin>>numberofnodes;
cout<<"Enter Elements";
node <T> *n,*x;
for(int i=1;i<=numberofnodes;i++)
{
n=new node <T>;
cin>>n->data;
n->left=n->right=NULL;
num=n->data;
x=Find(root,num);
if(root==NULL) // insert into empty tree
root=n;
else
{
if(n->data<x->data)
x->left=n;
else
x->right=n;
}
}
}
```

Tree Traversal

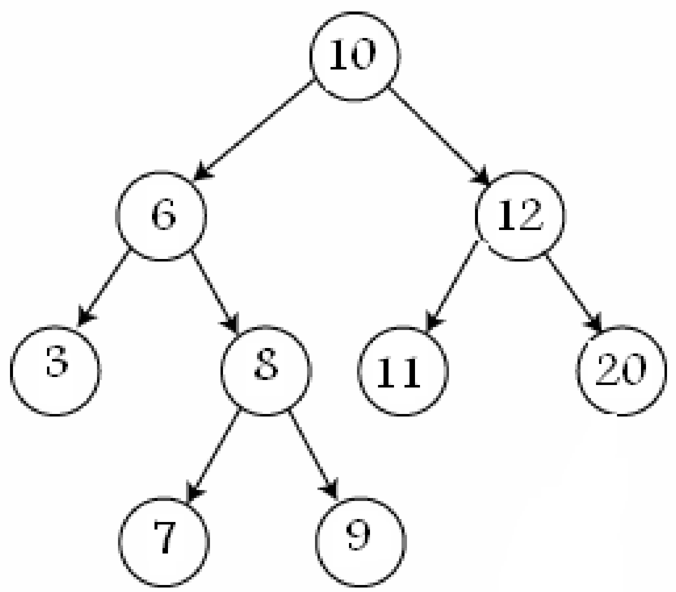
Used to print out the data in a tree in a certain order

- Pre-order traversal
- Post-order traversal
- Inorder traversal

Inorder Traversal of BST

- Inorder traversal of BST prints out all the keys in sorted order
- Algorithm
  1. Visit left subtree
  2. Visit root
  3. Visit right subtree

```
void inorder(node <T> *n)
{
if(n!=NULL)
{
inorder(n->left);
cout<<n->data<<" ";
inorder(n->right);
}
}
```

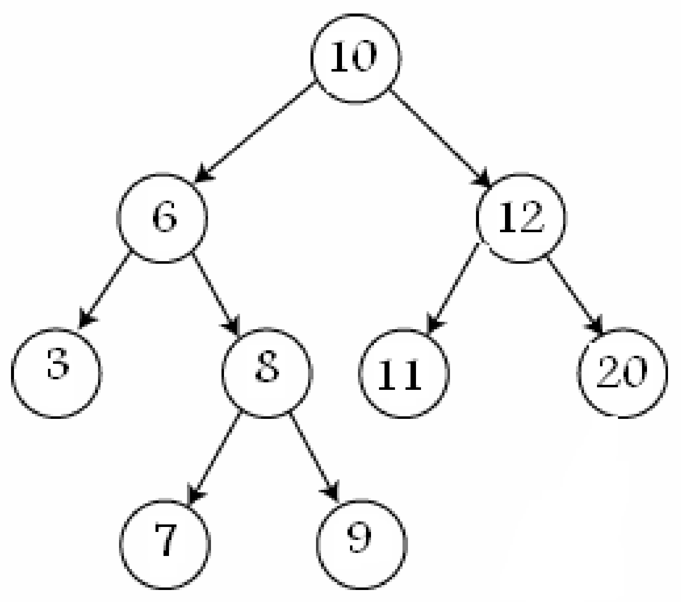


Inorder: 3,6,7,8,9,10,11,12,20

Preorder Traversal of BST

- Inorder traversal of BST prints out all the keys in sorted order
- Algorithm
  1. Visit root
  2. Visit left subtree
  3. Visit right subtree

```
void preorder(node <T> *n)
{
if(n!=NULL)
{
cout<<n->data<<" ";
preorder(n->left);
preorder(n->right);
}
}
```

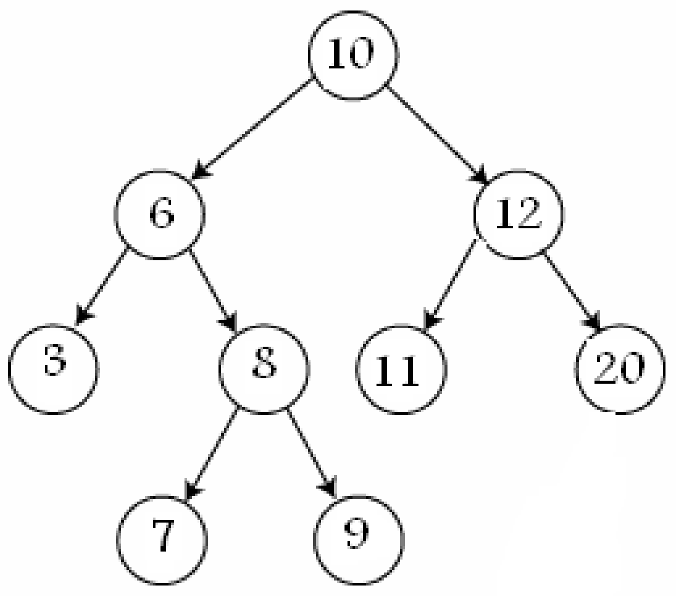


Preorder: 10,6,3,8,7,9,12,11,20

Postorder Traversal of BST

- Inorder traversal of BST prints out all the keys in sorted order
- Algorithm
  1. Visit left subtree
  2. Visit right subtree
  3. Visit root

```
void postorder(node <T> *n)
{
 if(n!=NULL)
{
postorder(n->left);
postorder(n->right);
cout<<n->data<<" ";
}
```



postorder: 3,7,9,8,6,11,20,12,10

findMin/ findMax

- Goal: return the node containing the smallest (largest) key in the tree
- Algorithm: Start at the root and go left (right) as long as there is a left (right) child.
- The stopping point is the smallest (largest) element

```
 node <T> *FindMin(node <T>*p)
{
if(p==NULL)
return NULL;
if(p->left==NULL)
return p;
return FindMin(p->left);
}
```

- Time complexity = O(height of the tree)

Deletion

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - This has to be done such that the property of the search tree is maintained.

Deletion under Different Cases

- Case 1: the node is a leaf
  - Delete it immediately
- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass that node

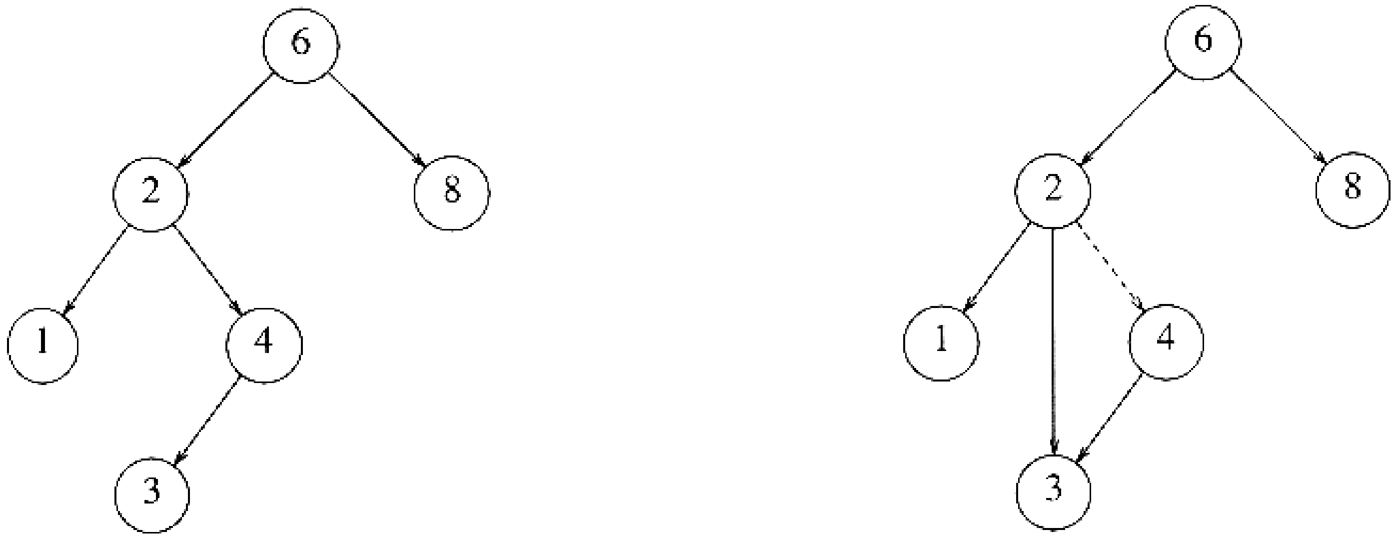


Figure 4.24 Deletion of a node (4) with one child, before and after

Deletion Case 3

- Case 3: the node has 2 children
  - Replace the key of that node with the minimum element at the right subtree
  - Delete that minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

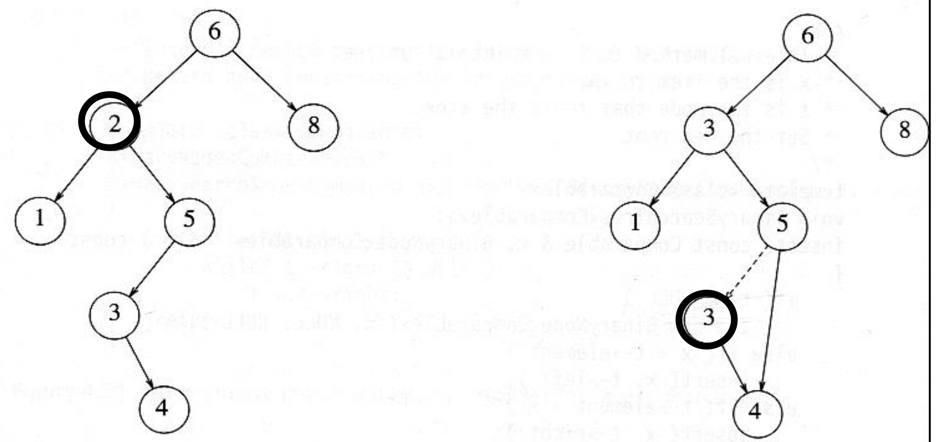


Figure 4.25 Deletion of a node (2) with two children, before and after

- **Time complexity = O(height of the tree)**

Q: How would you find the ‘successor’ (i.e., next greatest number) of a node in a Binary Search Tree?

Example: Expression Trees

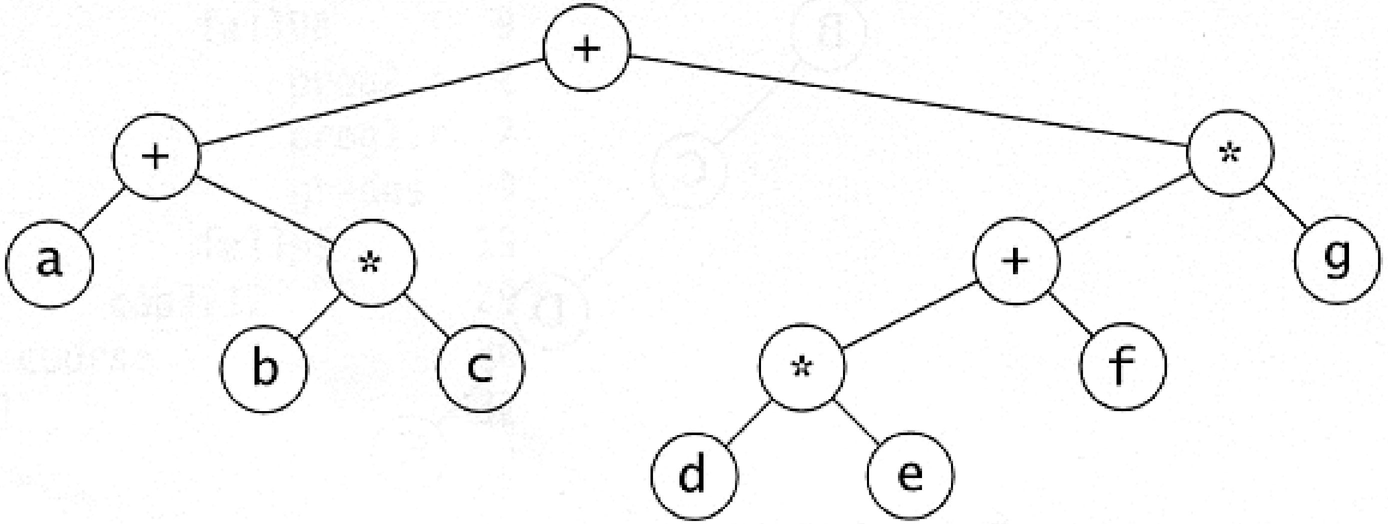


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

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

Preorder

- Print the data at the root
- Recursively print out all data in the left subtree
- Recursively print out all data in the right subtree
- node, left, right
- prefix expression
  - ++a*bc*+*defg

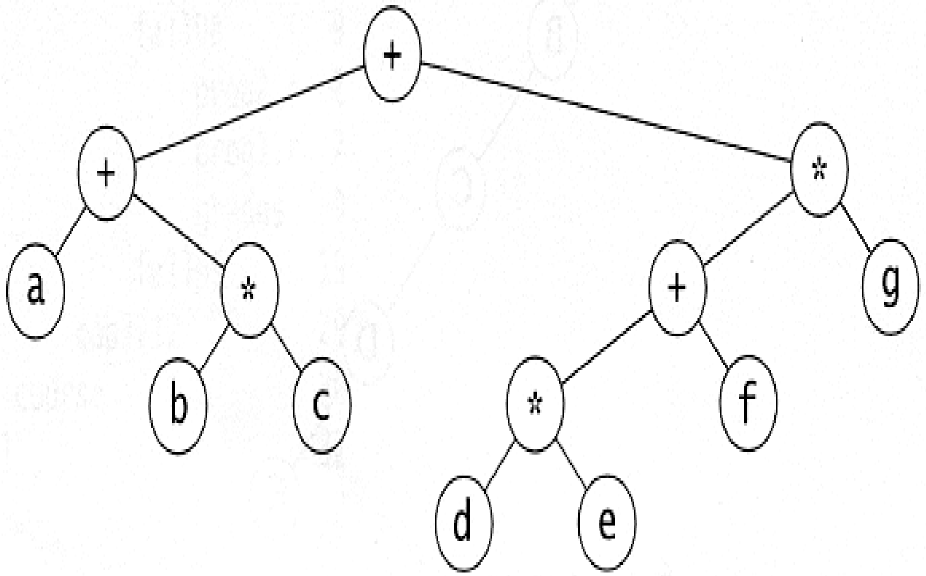


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Postorder

- Postorder traversal
  - left, right, node
  - postfix expression
    - abc*+de*f+g*+

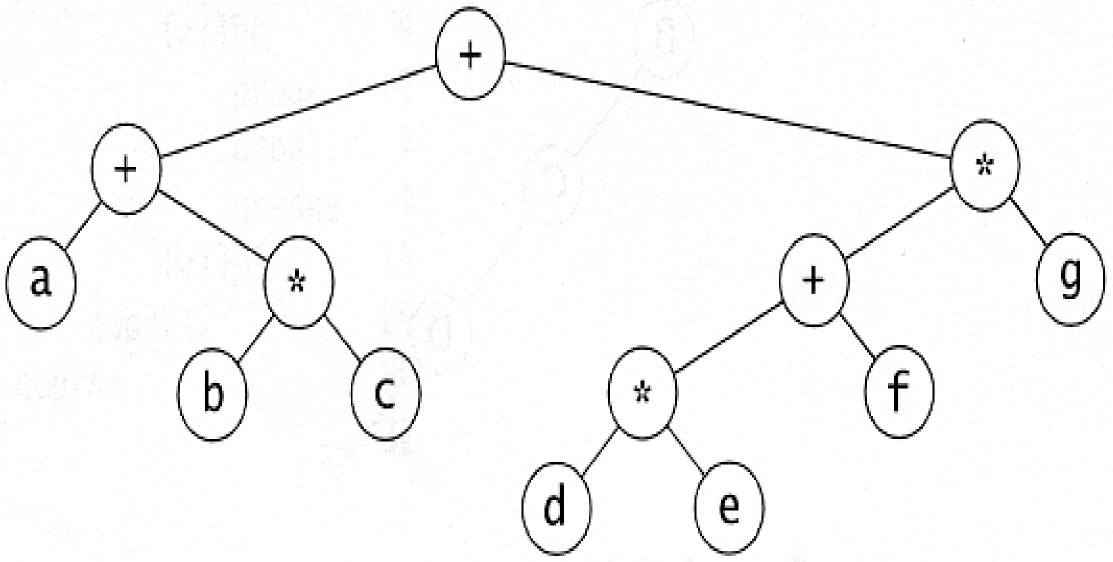


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Inorder

- Inorder traversal
  - left, node, right
  - infix expression
    - a+b*c+d*e+f*g

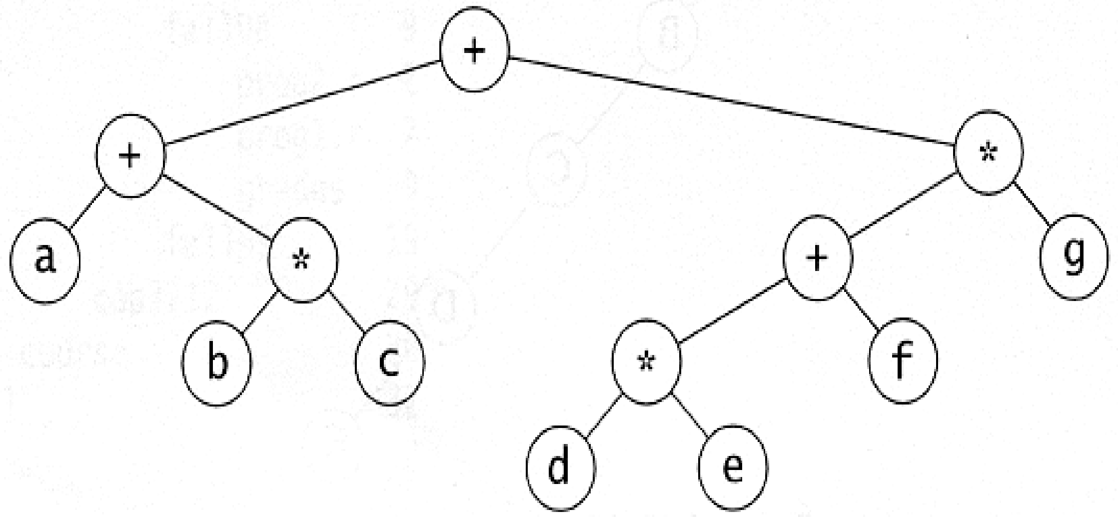


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Example: UNIX Directory

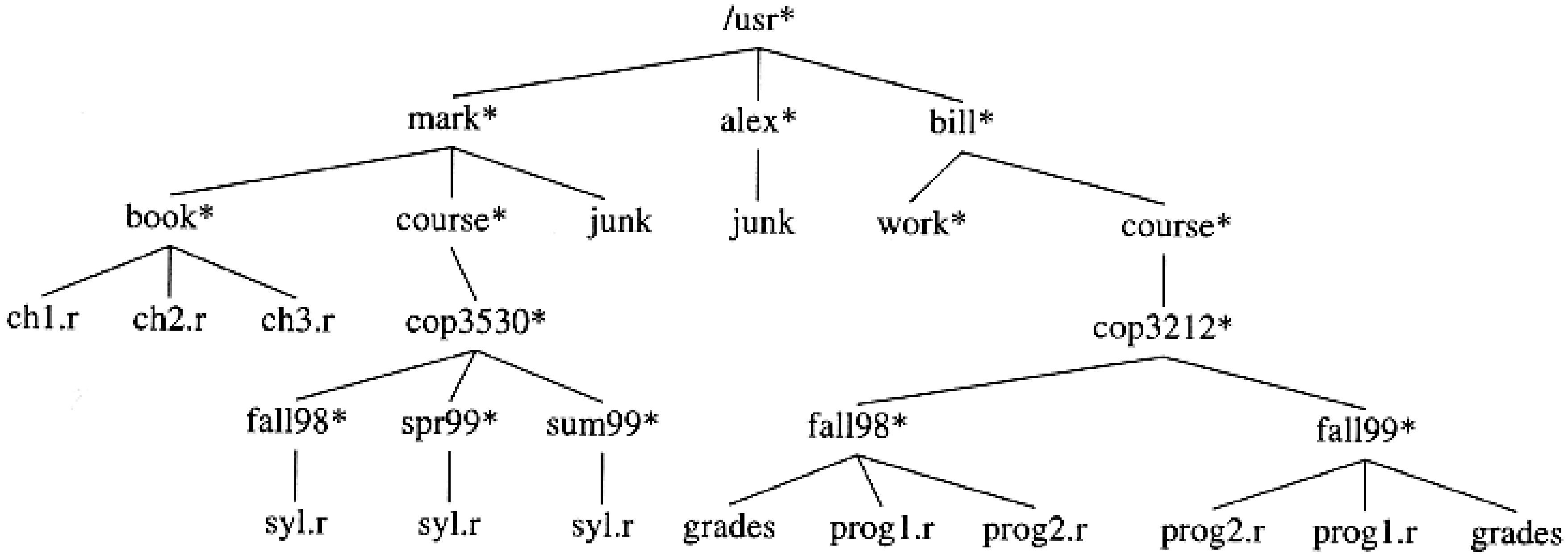


Figure 4.5 UNIX directory

Example: Unix Directory Traversal

PreOrder

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall98
                    syl.r
                spr99
                    syl.r
                sum99
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall98
                    grades
                    prog1.r
                    prog2.r
                fall99
                    prog2.r
                    prog1.r
                    grades
```

PostOrder

```
            ch1.r          3
            ch2.r          2
            ch3.r          4
        book              10
                    syl.r  1
                fall98     2
                    syl.r  5
                spr99      6
                    syl.r  2
                sum99      3
            cop3530       12
        course            13
        junk               6
    mark                  30
        junk               8
    alex                   9
        work               1
                    grades   3
                    prog1.r  4
                    prog2.r  1
                fall98       9
                    prog2.r  2
                    prog1.r  7
                    grades   9
                fall99      19
            cop3212         29
        course              30
    bill                    32
/usr                        72
```

Why Tree Data Structure?

- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

End