Trees

- So far we have discussed mainly linear data structures – strings, arrays, lists, stacks and queues
- Now we will discuss a non-linear data structure called tree.
- Trees are mainly used to represent data containing a hierarchical relationship between elements, for example, records, family trees and table of contents.

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 an *edge* from r

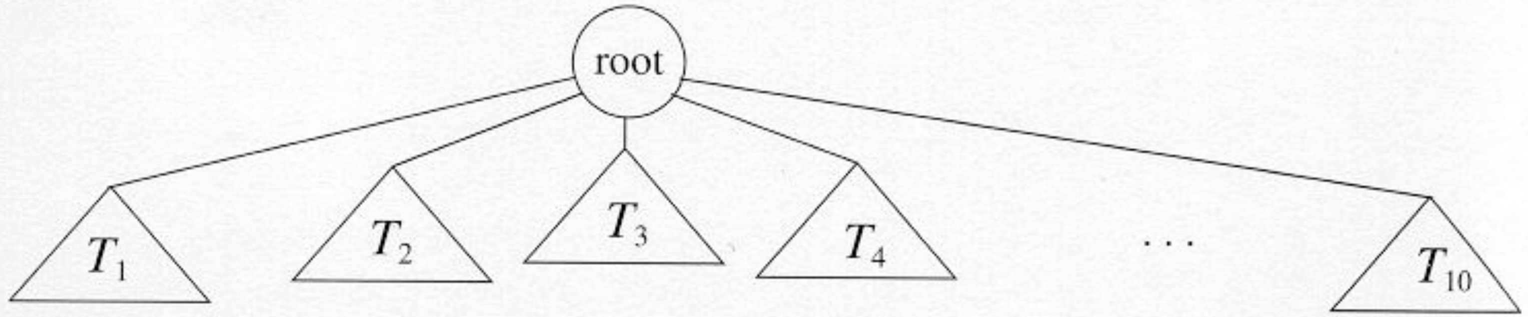


Figure 4.1 Generic tree

Some Terminologies

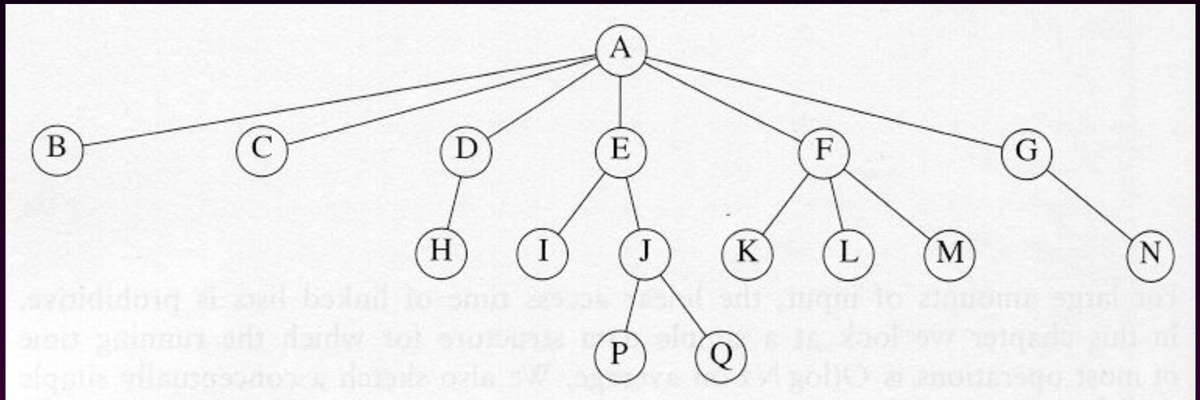


Figure 4.2 A tree

- *Child* and *parent*
  - Every node except the root has one parent
  - A node can have an arbitrary number of children
- *Leaves*
  - Nodes with no children
- *Sibling*
  - nodes with same parent

Some Terminologies

- *Path*
- *Length*
  - number of edges on the path
- *Depth* of a node
  - length of the unique path from the root to that node
  - The depth of a tree is equal to the depth of the deepest leaf
- *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 is equal to the height of the root

Example: UNIX Directory

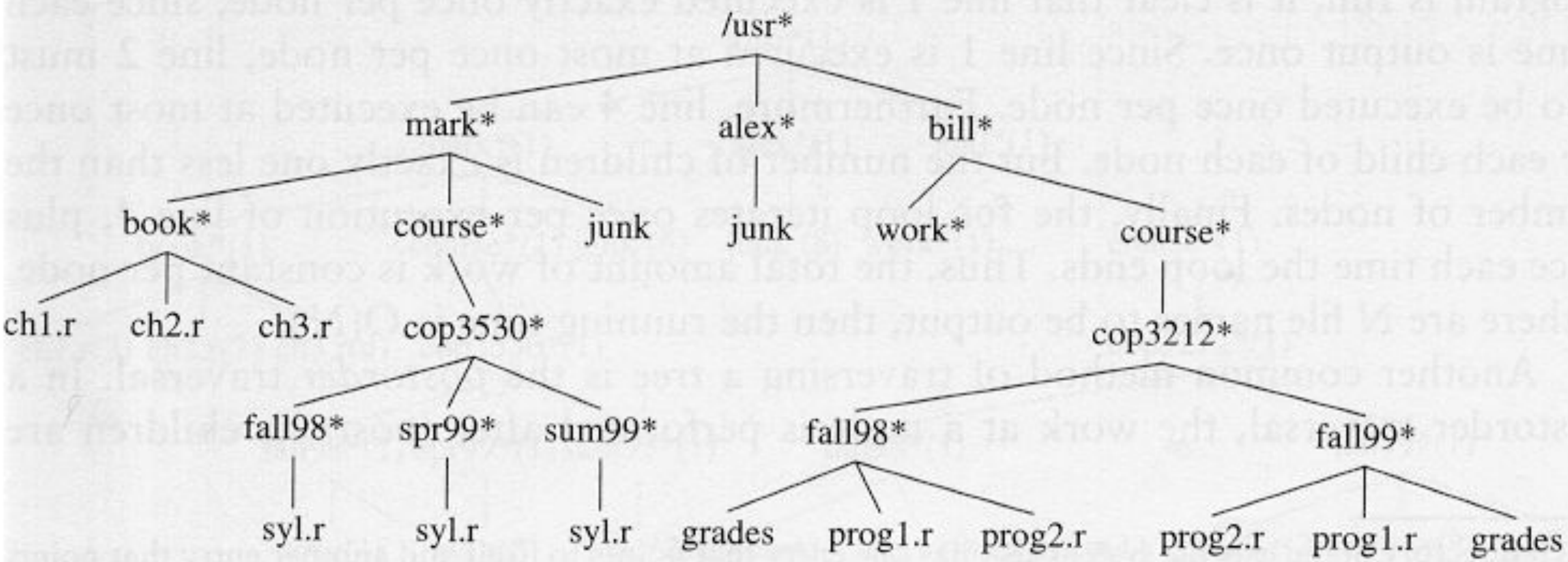
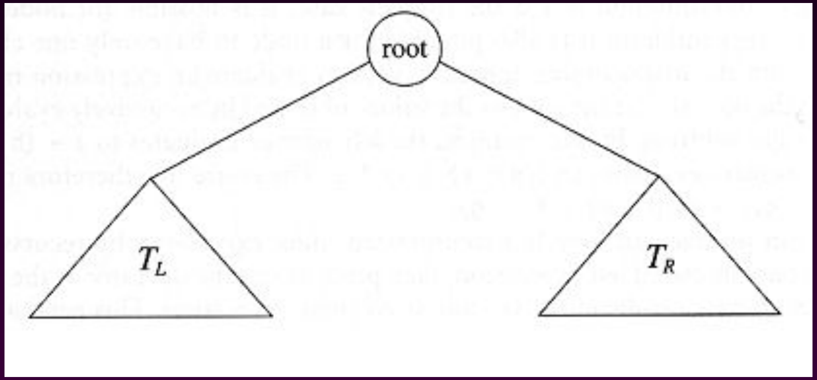


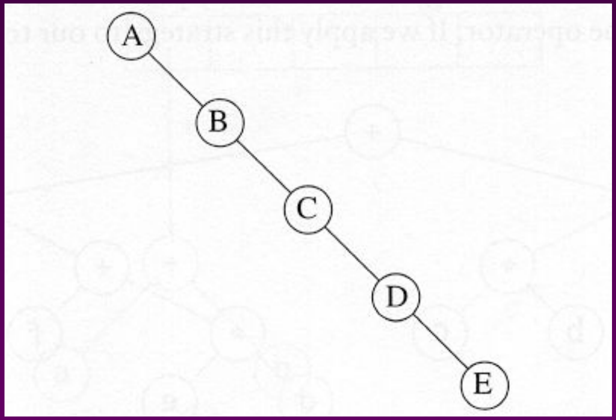
Figure 4.5 UNIX directory

Binary Trees

- A tree in which no node can have more than two children



- 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.



Example: Expression Trees

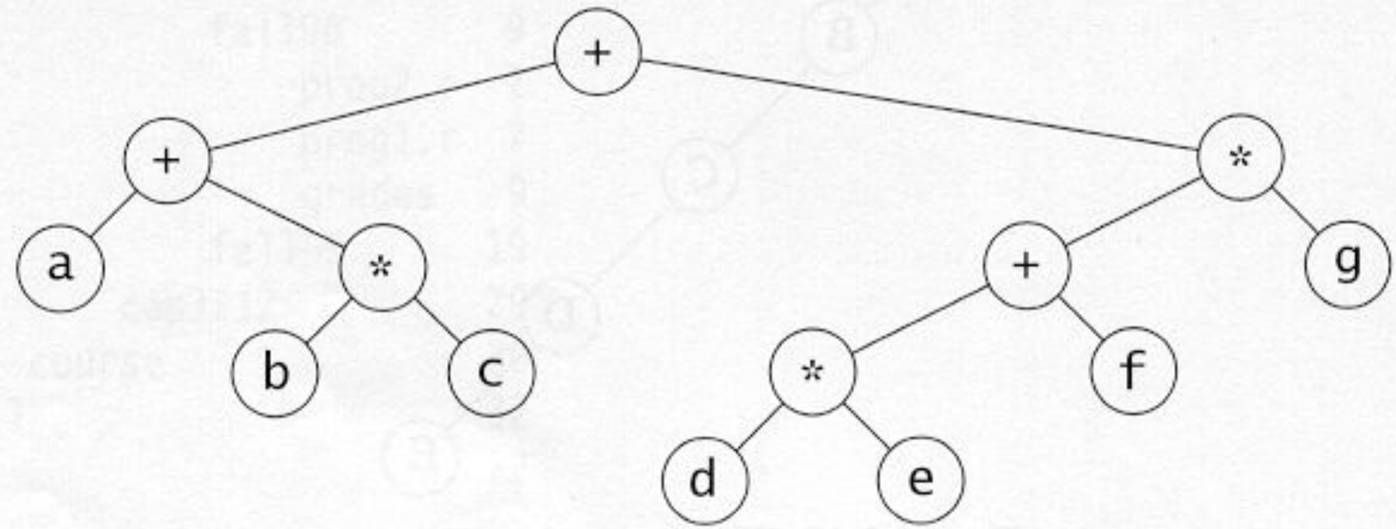


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- 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

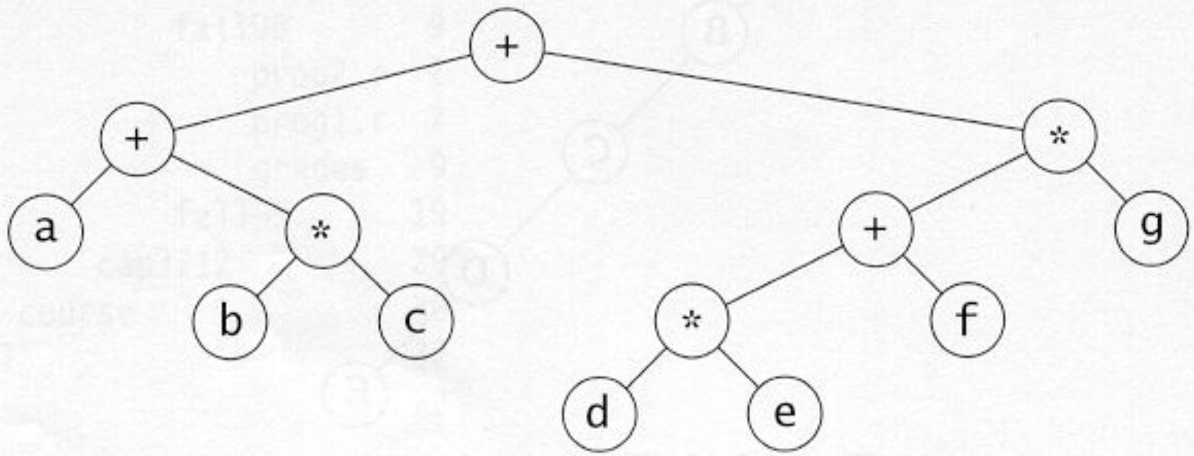


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

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

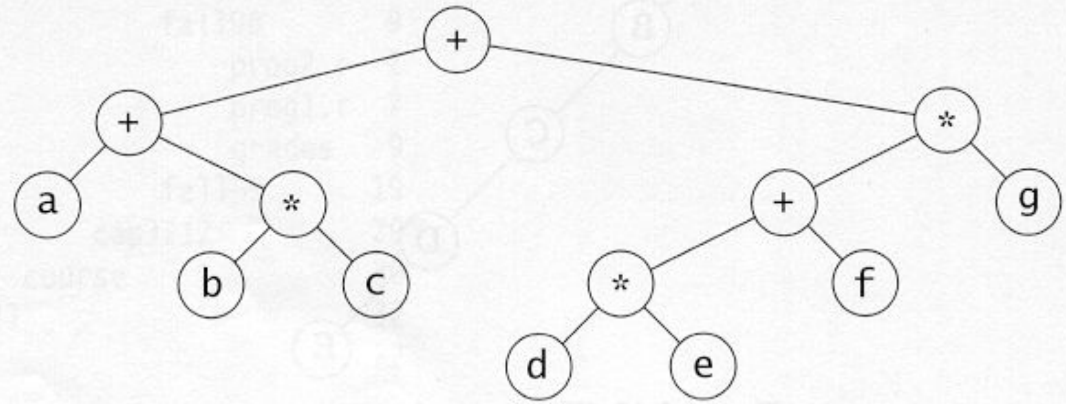


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

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 *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);

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));

Binary Trees

- Possible operations on the Binary Tree ADT
  - parent
  - left_child, right_child
  - sibling
  - root, etc
- Implementation
  - Because a binary tree has at most two children, we can keep direct pointers to them

```
struct BinaryNode
{
    Object      element;        // The data in the node
    BinaryNode *left;           // Left child
    BinaryNode *right;          // Right child
};
```

Properties of Binary Trees

– **Lemma 5.1 [*Maximum number of nodes*]:**

1. The maximum number of nodes on level i of a binary tree is 2^{i-1}, $i \geq 1$.
2. The maximum number of nodes in a binary tree of depth k is 2^k-1, $k \geq 1$.

– **Lemma 5.2 [*Relation between number of leaf nodes and degree-2 nodes*]:**

For any nonempty binary tree, T, if n_0 is the number of leaf nodes and n_2 is the number of internal nodes then $n_0 = n_2 + 1$.

Types of Binary Trees

Full Binary Tree

A Binary Tree is a full binary tree if every node has 0 or 2 children.

Complete Binary Tree

A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible

Perfect Binary Tree

A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.

```
         18
       /    \
     15      20
    /  \
  40    50
 /   \
30    50
```

```
         18
       /       \
     15          30
    /  \        /  \
  40    50   100    40
 /  \    /
8    7   9
```

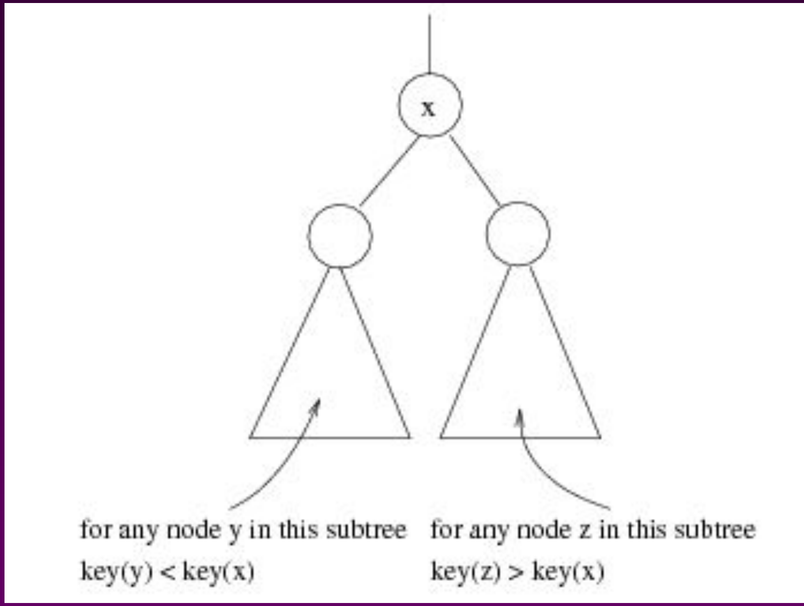
```
         18
       /      \
     15         30
    /  \       /  \
  40    50  100    40
```

Binary Search Trees

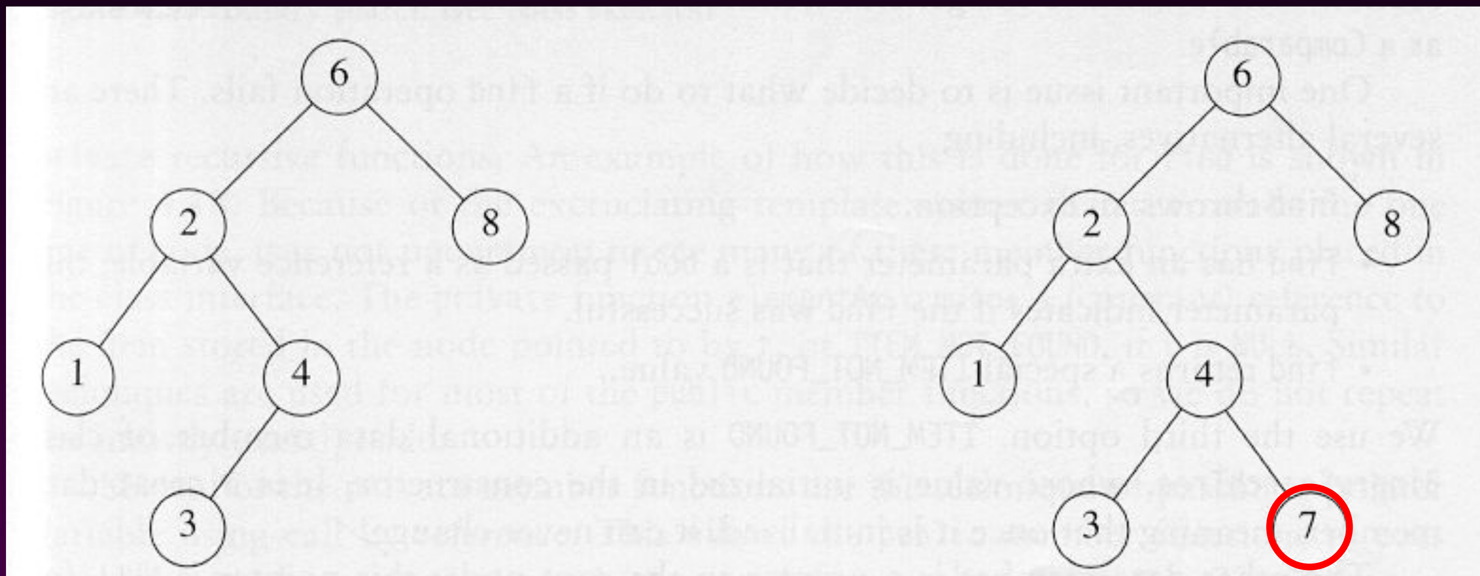
- Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.

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

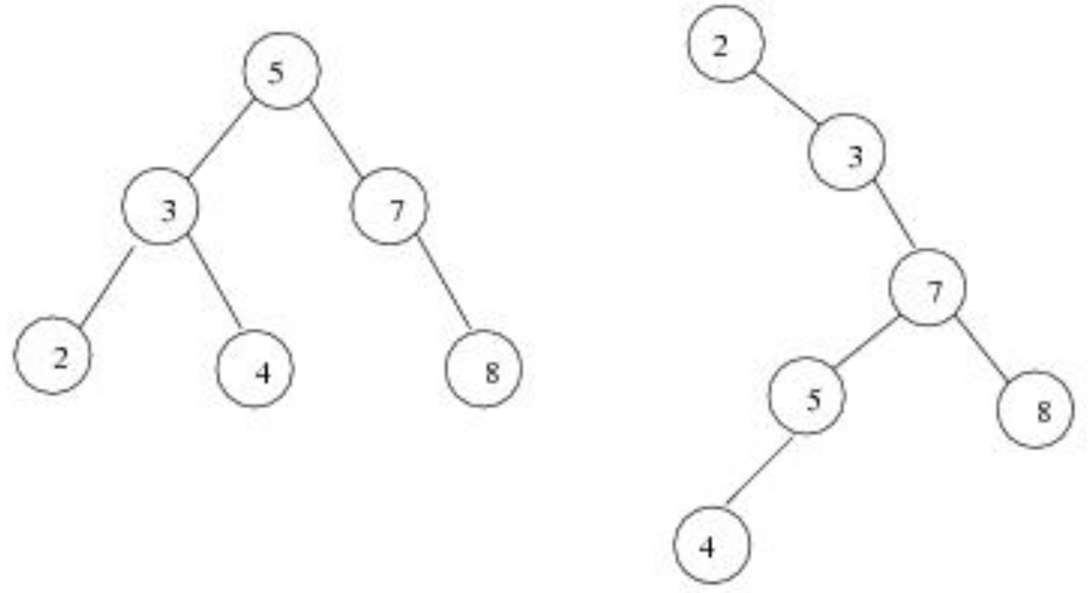


A binary search tree

Not a binary search tree

Binary search trees

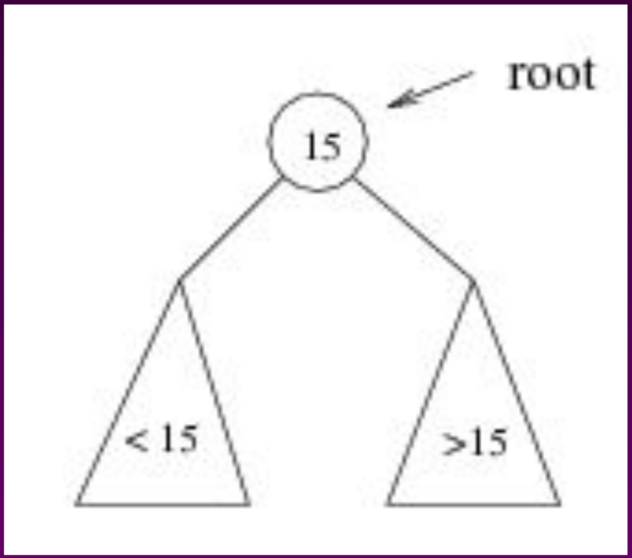
Two binary search trees representing the same set:



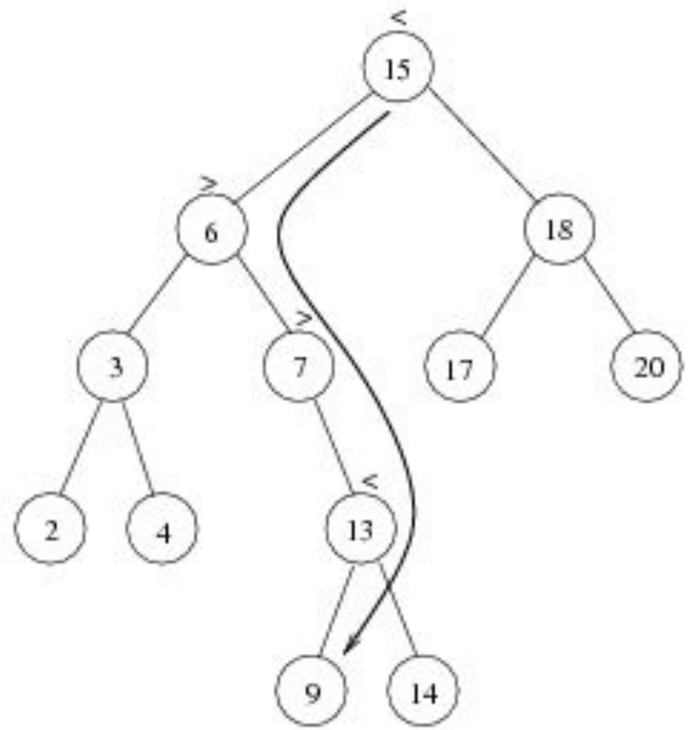
- Average depth of a node is O(log N); maximum depth of a node is O(N)

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)

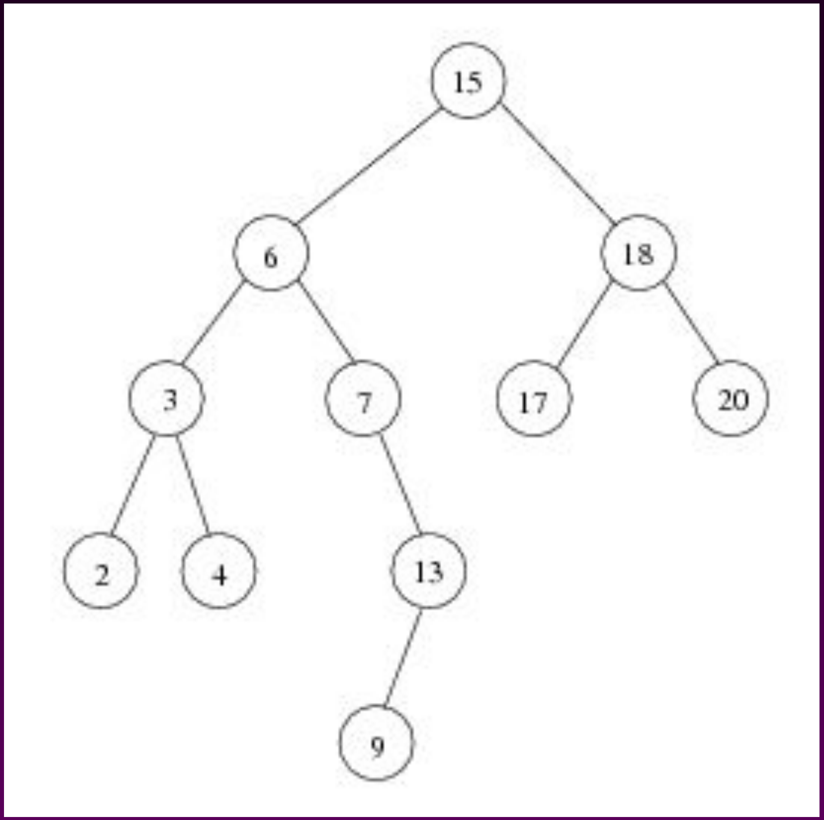
- Find X: return a pointer to the node that has key X, or NULL if there is no such node

```
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::
find( const Comparable & x, BinaryNode<Comparable> *t ) const
{
    if( t == NULL )
        return NULL;
    else if( x < t->element )
        return find( x, t->left );
    else if( t->element < x )
        return find( x, t->right );
    else
        return t;    // Match
}
```

- Time complexity
  - O(height of the tree)

Inorder traversal of BST

- Print out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

findMin/ findMax

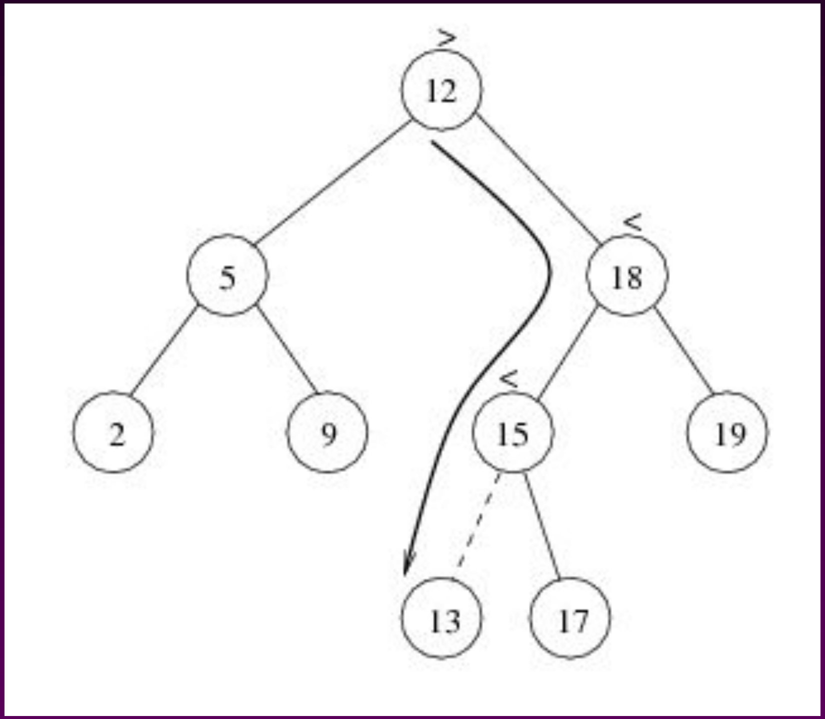
- Return the node containing the smallest element in the tree
- Start at the root and go left as long as there is a left child. The stopping point is the smallest element

```
template <class Comparable>
BinaryNode<Comparable> *
BinarySearchTree<Comparable>::findMin( BinaryNode<Comparable> *t ) const
{
    if( t == NULL )
        return NULL;
    if( t->left == NULL )
        return t;
    return findMin( t->left );
}
```

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

insert

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



- Time complexity = O(height of the tree)

delete

- 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.

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

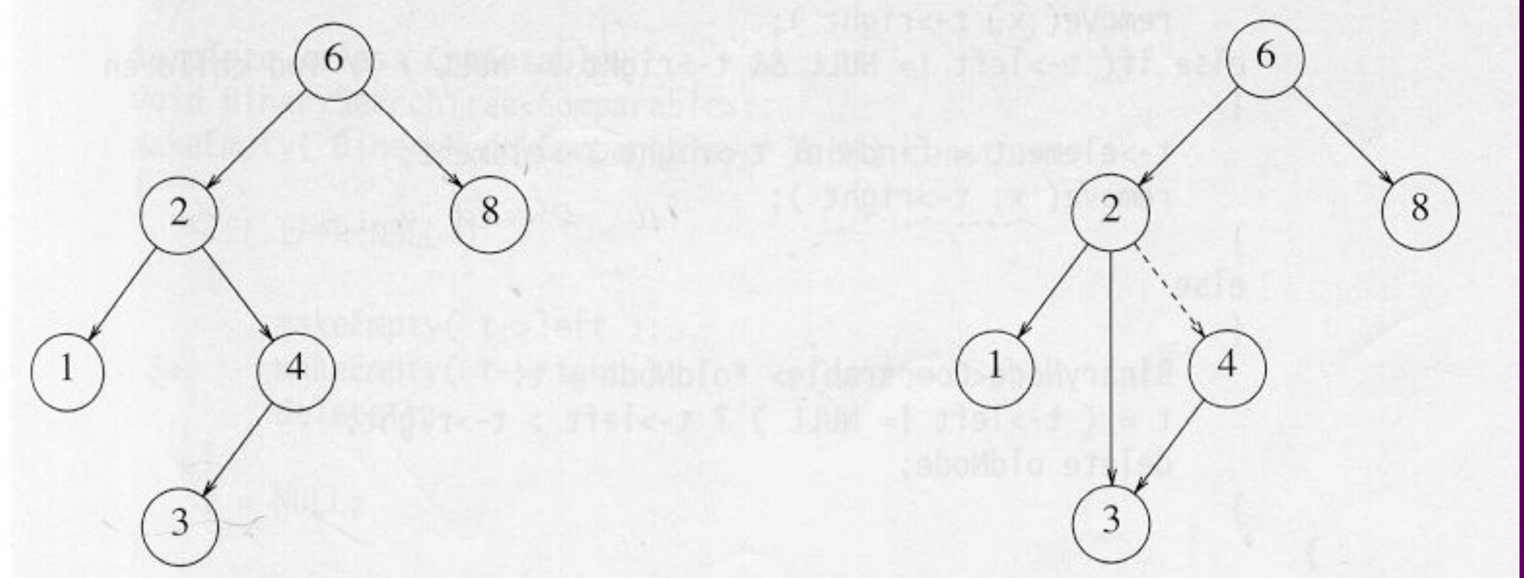


Figure 4.24 Deletion of a node (4) with one child, before and after

delete

(3) the node has 2 children

- replace the key of that node with the minimum element at the right subtree
- delete the 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.

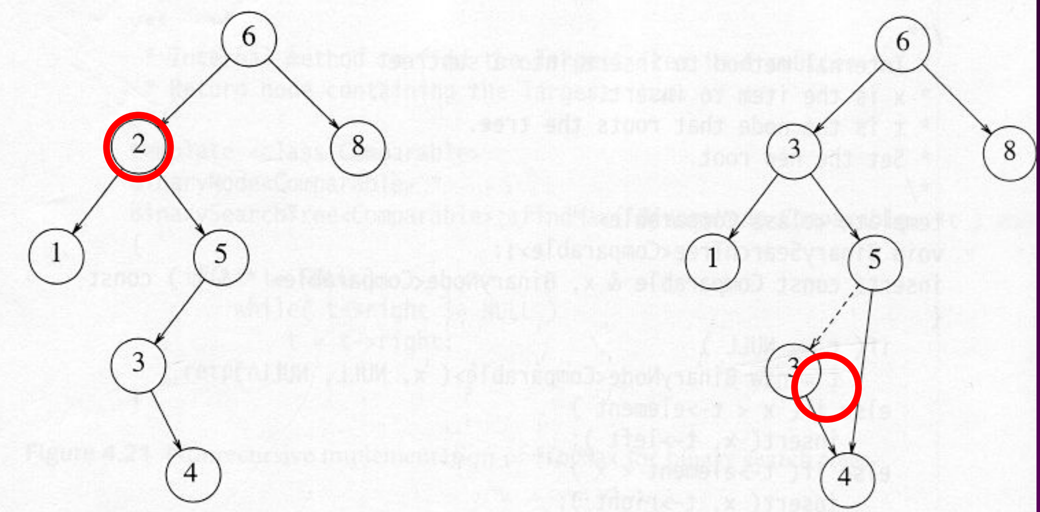


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

Time Complexities

	Array	Linked List	BST
Insert	O(1)	O(1)	O(logn)
Delete	O(n)	O(n)	O(logn)
Search	O(n)	O(n)	O(logn)