Trees

Trees

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

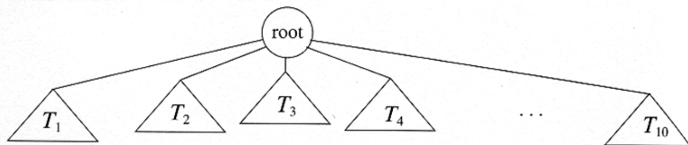
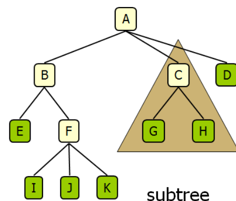


Figure 4.1 Generic tree

Some Terminologies

- *Siblings*: nodes share the same parent
- *Internal node*: node with at least one child (A, B, C, F)
- *External node (leaf)*: node without children (E, I, J, K, G, H, D)



- *Ancestor* of a node: itself, parent, grandparent, etc.
- *Descendant* of a node: itself, child, grandchild, etc.
- *Depth* of a node:
  - length of the unique path (i.e., number of edges) 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
- *Subtree*: tree consisting of a node and its descendants

Example: UNIX Directory

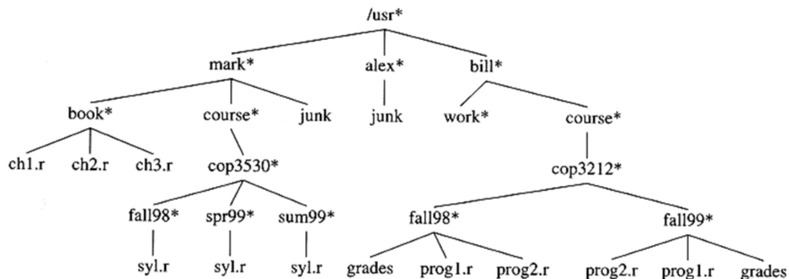
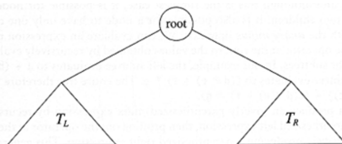


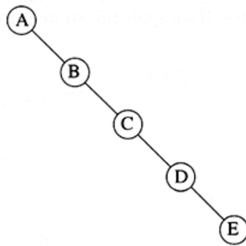
Figure 4.5 UNIX directory

Binary Trees

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

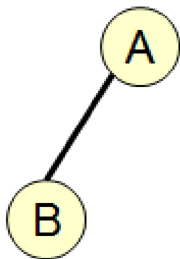


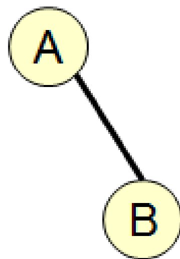
- The depth of an "average" binary tree is considerably smaller than N, even though in the worst case, the depth can be as large as N - 1.



Differences Between A Tree and A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered





- Are different when viewed as binary trees.
- Are the same when viewed as trees.

Example: Expression Trees

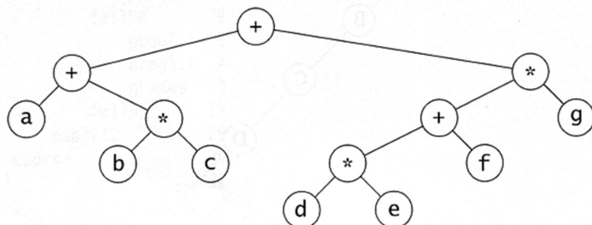
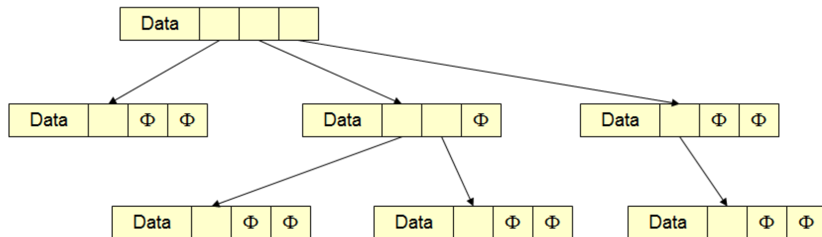


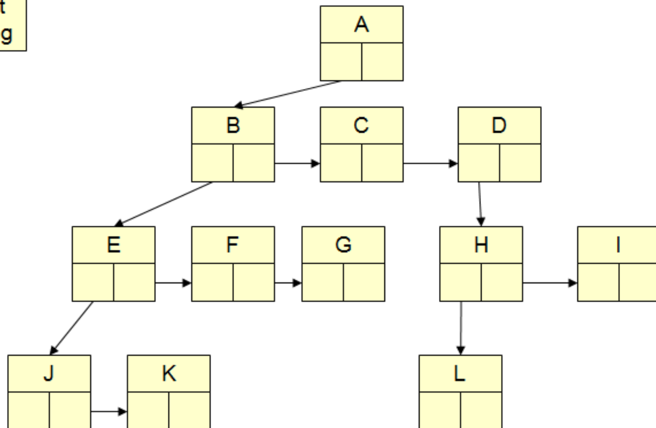
Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

- 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

Pointer Representation of a Tree Node



Left Child, Right Sibling Representation



Example

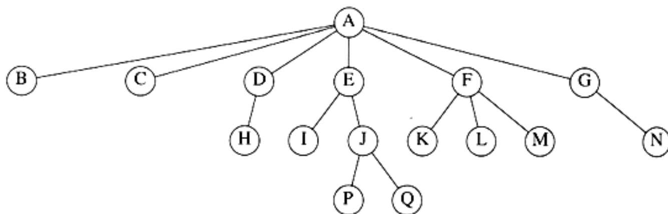


Figure 4.2 A tree

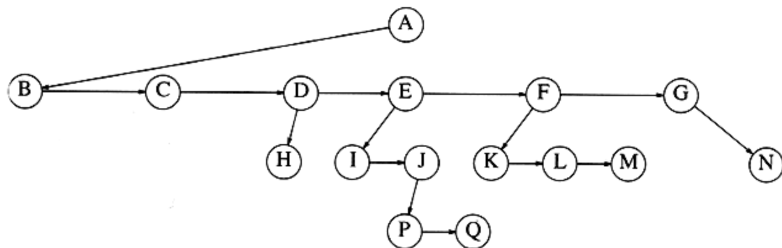
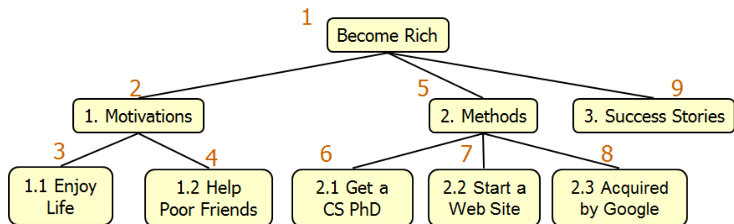


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Tree Traversal - Preorder

- visit the root
- traverse in preorder the children (subtrees)

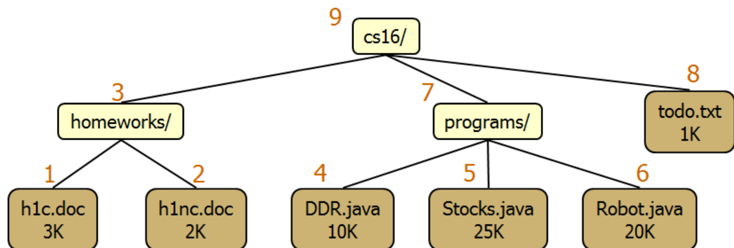
Algorithm *preOrder*(v)
  visit(v)
  for each child w of v
    preorder(w)



Tree Traversal - Postorder

- traverse in postorder the children (subtrees)
- visit the root

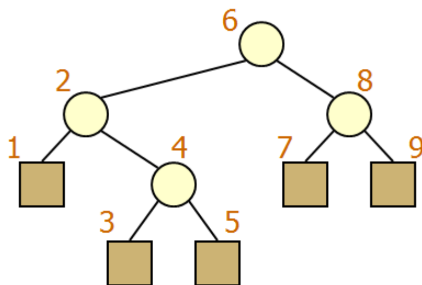
Algorithm *postOrder*(v)
  for each child w of v
    postOrder(w)
  visit(v)



Tree Traversal - Inorder (Binary Tree)

- traverse in inorder the left subtree
- visit the root
- traverse in inorder the right subtree

Algorithm *inOrder*(v)
  if *isInternal* (v)
    inOrder (*leftChild* (v))
  visit(v)
  if *isInternal* (v)
    inOrder (*rightChild* (v))



Inorder Traversal of an Expression Tree

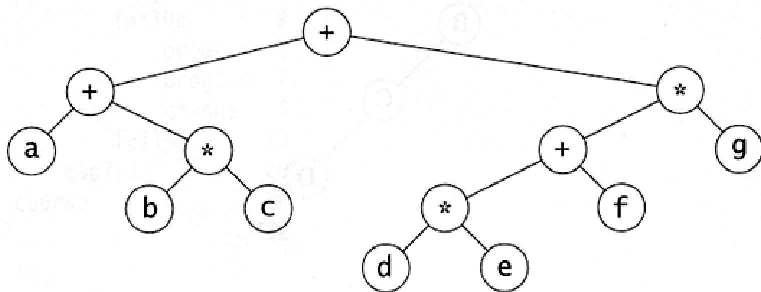
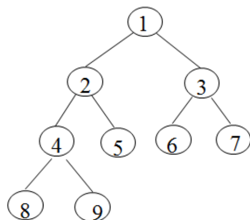


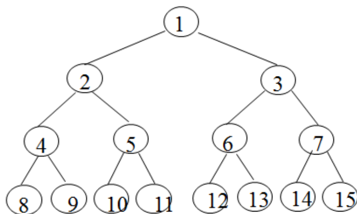
Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Infix expression $a + b * c + d * e + f * g$

Full and Complete Binary Trees



Complete binary tree

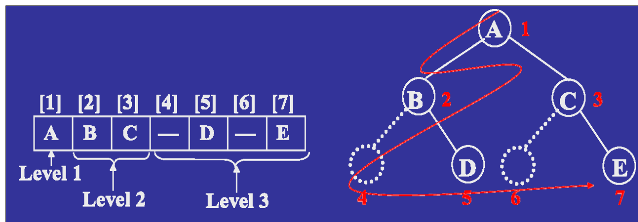


Full binary tree of depth 3

- A full binary tree of a given height k has $2^{k+1} - 1$ nodes.
- Parent of node i is node $i/2$, unless $i = 1$.
- Right child of node i is node $2i + 1$, unless $2i + 1 > n$, where n is the number of nodes.
- Left child of node i is node $2i$, unless $2i > n$, where n is the number of nodes.

Representing Binary Trees Using Arrays

- Waste spaces: in the worst case, a skewed tree of depth k requires $2^k - 1$ spaces, of these, only k spaces will be occupied.
- Good if the tree is almost full (complete).



Threaded Binary Trees

- Using left/right pointers in a node, too many null pointers in current representation of binary trees
- Rules for constructing the *threads*
  - If ptr $\rightarrow$ left_child is null, replace it with a pointer to the node that would be visited before ptr in an inorder traversal
  - If ptr $\rightarrow$ right_child is null, replace it with a pointer to the node that would be visited after ptr in an inorder traversal

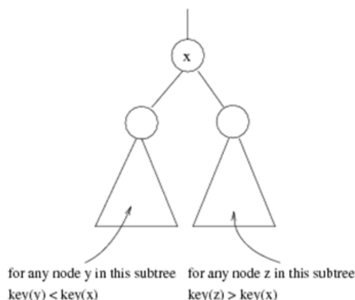
Threaded Binary Trees (Cont'd)

- Inorder traversal of a threaded binary tree
  - By using of threads we can perform an inorder traversal without making use of a stack (simplifying the task)
  - Now, we can follow the thread of any node, ptr, to the "next" node of inorder traversal

1. If ptr $\rightarrow$ right_thread = TRUE, the inorder successor of ptr is ptr $\rightarrow$ right_child by definition of the threads
2. Otherwise we obtain the inorder successor of ptr by following a path of left-child links from the right-child of ptr until we reach a node with left_thread = TRUE

Binary Search Trees

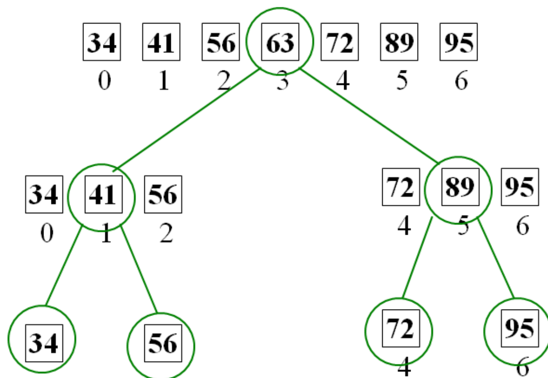
- Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.
- *Binary search tree* property
  - For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X



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

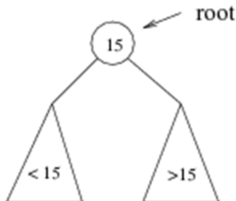
Binary Search Algorithm

Binary Search algorithm of an array of sorted items reduces the search space by one half after each comparison



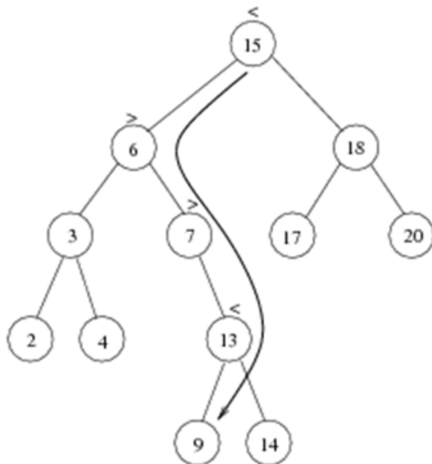
Searching BST

- 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

Example: Search for 9 ...



findMin/ findMax

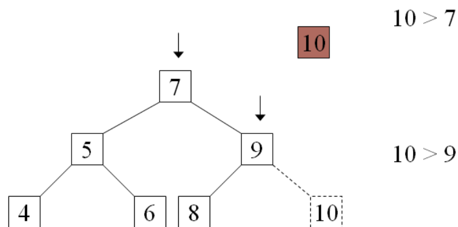
- Return the node containing the smallest element in the tree
- Start at the root and go left as long as there is a left child. The stopping point is the smallest element

```
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

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

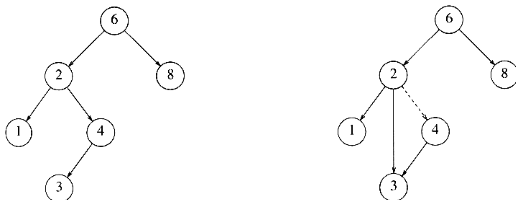


Figure 4.24 Deletion of a node (4) with one child, before and after

Delete (Cont'd)

- 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.

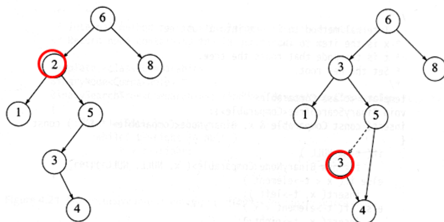


Figure 4.25 Deletion of a node (2) with two children, before and after

- Time complexity = O(height of the tree)

Worst Case

```
BST tree = new BST();
for (int i = 1; i <= 8; i++)
    tree.insert (i);
```

Output:

```
>>>> Items in worst order:
            8
          7
        6
      5
    4
   3
  2
1
```

Better Search Trees

Prevent the degeneration of the BST :

- A BST can be set up to maintain balance during updating operations (insertions and removals)
- Types of ST which maintain the optimal performance:
  - AVL trees
  - splay trees
  - 2-3 (2-3-4) Trees
  - Red-Black trees
  - B-trees