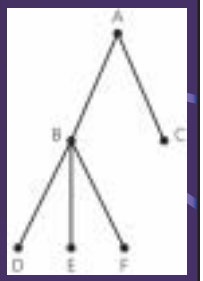
Trees

Terminology

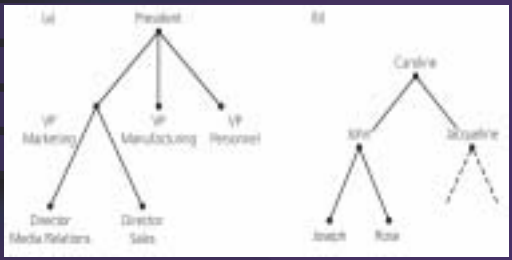
- Trees are hierarchical
  - "parent-child" relationship
    - A is the parent of B
    - B is a child of A
    - B and C are siblings
    - Generalized to ancestor and descendant
  - root: the only node without parent
  - leaf: a node has no children
  - Subtree of node N: A tree that consists of a child (if any) of N and the child's descendants



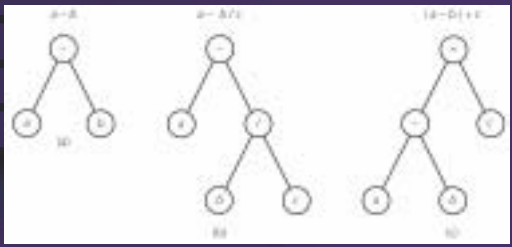
General Tree v.s. Binary Tree

- A general tree T is a set of one or more nodes such that T is partitioned into disjoint subsets:
  - A node r, the root
  - Sets that are general trees, called subtrees of r
- A binary tree is a set T of nodes such that
  - T is empty, or
  - T is partitioned into 3 disjoint subsets:
    - A node r, the root
    - 2 possibly empty sets that are binary trees, called left and right subtrees of r

General Tree v.s. Binary Tree

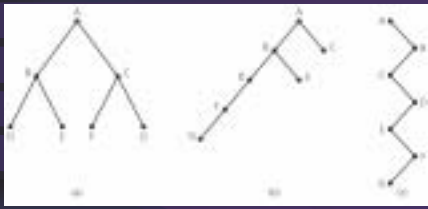


Represent Algebraic Expressions using Binary Tree



Height of Trees

- Trees come in many shapes

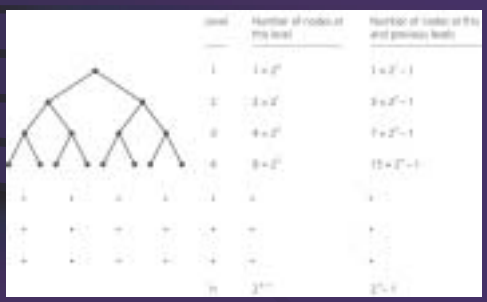


- Height of any tree: number of nodes on the longest path from the root to a leaf

Full Binary Trees

- Full binary tree
  - All nodes that are at a level less than h have two children, where h is the height
- Each node has left and right subtrees of the same height

Full Binary Tree

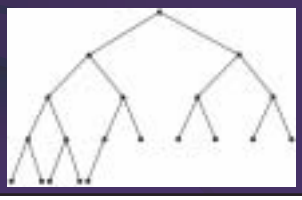


Full Binary Tree

- If the height is h>0
  - The number of leaves is 2^(h-1)
  - The number of nodes is 2^h – 1
- If the number of nodes is N>0
  - The height is $\log_2(N+1)$
  - The number of leaves is (N+1)/2

Complete Binary Trees

- Complete binary tree
  - A binary tree full down to level h-1, with level h filled in from left to right



Balanced Binary Trees

- Balanced binary tree
  - The height of any node's right subtree differs from the height of the node's left subtree by no more than 1
  - A complete binary tree is balanced

ADT Binary Tree

- Operations
  - Create/destroy a tree
  - Determine/change root
  - Determine emptiness
  - Attach/Detach left/right subtree to root
  - Return a copy of the left/right subtree of root
  - Traverse the nodes

Build a Tree

```
Tree1.setRootData('F');
Tree1.attachLeft('G');

Tree2.setRootData('D');
Tree2.attachLeftSubTree(tree1);

Tree3.setRootData('B');
Tree3.attachLeftSubtree(tree2);
Tree3.attachRight('E');

Tree4.setRootData('C);
binTree.createBinaryTree('A', tree3, tree4);
```

Traversal of a Binary Tree

- A binary tree is empty, or root with two binary subtrees

```
traverse(binTree) {
  if (!isEmpty(binTree)){
    traverse(left subtree of binTree's root);
    traverse(right subtree of binTree's root);
  }
}
```

- Only thing missing is to display root's data

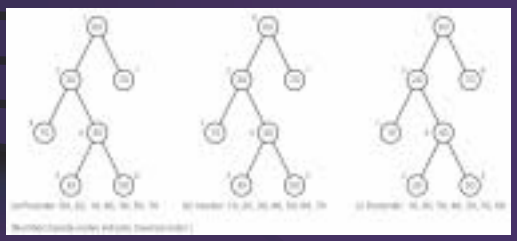
Preorder, inorder, postorder

```
traverse(binTree)
  if (!isEmpty(binTree))
    display data in binTree's root;
    traverse(left subtree of binTree's root);
    traverse(right subtree of binTree's root);
```

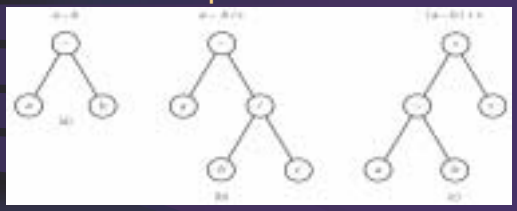
```
traverse(binTree)
  if (!isEmpty(binTree))
    traverse(left subtree of binTree's root);
    display data in binTree's root
    traverse(right subtree of binTree's root);
```

```
traverse(binTree)
  if (!isEmpty(binTree))
    traverse(left subtree of binTree's root);
    traverse(right subtree of binTree's root);
    display data in binTree's root;
```

Traversal Examples



Traversal on Algebraic Expressions

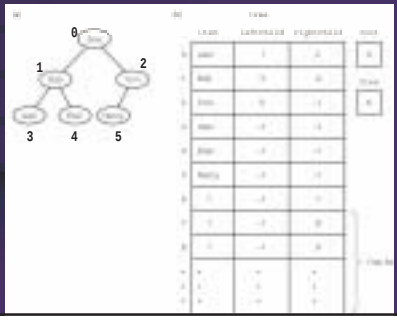


- Preorder, inorder, and postorder print prefix, infix and postfix expressons

Traversal is O(N)

- To traverse a binary tree with N nodes
  - Visits every node just once
  - Each visit performs the same operations on each node, independently of N

Array-Based Implementation of a Binary Tree



Array-Based Implementation

- Represent binary tree by using an array of tree nodes

```
class TreeNode {
private:
  TreeNode();
  TreeNode(TreeItem&
nItem, int left, int
right);
  TreeItem item;
  int leftC;
  int rightC;
  friend class BinaryTree;
};
```

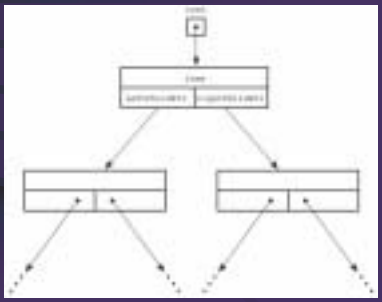
```
class BinaryTree {
public:
  …
private:
  TreeNode[MAX] tree;
  int root;
  int free;
};
```

Array-Based Implementation of a Complete Binary Tree

0	Jane
1	Bob
2	Tom
3	Alan
4	Ellen
5	Nancy
6	
7	

- If nodes numbered according to a level-by-level scheme
  - Root index: 0
  - Given any node tree[i]
    - Left child index: 2*i+1
    - Right chile index: 2*i+2
    - Parent index: (i-1)/2

Pointer-Based Implementation



Pointer-Based Implementation

- Use pointers to link tree nodes

```
class TreeNode {
private:
  TreeNode();
  TreeNode(TreeItem& nItem,
     TreeNode *left=NULL,
     TreeNode *right=NULL);
  TreeItem item;
  TreeNode *leftPtr;
  TreeNode *rightPtr;
  friend class BinaryTree;
};
```

```
class BinaryTree {
public:
  …
private:
  TreeNode *root;
};
```

Pointer-Based Implementation

```
BinaryTree::BinaryTree(TreeItem& rootItem){
  root = new TreeNode(rootItem, NULL, NULL);
}
void BinaryTree::attachLeft(TreeItem& newItem){
  if(root!=NULL && root->leftPtr == NULL)
    root->leftPtr = new TreeNode(newItem,NULL,NULL);
}
void BinaryTree::attachLeftSubtree(BinaryTree& tree){
  if (root!=NULL && root->leftPtr == NULL){
    root->leftPtr = tree.root;
    tree.root = NULL;
  }
}
```

Pointer-Based Implementation

```
void BinaryTree::inorder(TreeNode *treePtr,
                         FunctionType visit){
  if (treePtr!=NULL) {
    inorder(treePtr->leftPtr, visit);
    visit(treePtr->item);
    inorder(treePtr->rightPtr, visit);
  }
}
```

- How about preorder(), postorder()?

Pointer-Based Implementation

```
void BinaryTree::destroyTree(TreeNode *&treePtr){
  if (treePtr!=NULL){
    destroyTree(treePtr->leftPtr);
    destroyTree(treePtr->rightPtr);
    delete treePtr;
    treePtr = NULL;
  }
}
```

- Can we have preorder, or inorder style destroyTree()?

Binary Search Tree

- A deficiency of the ADT binary tree which is corrected by the ADT binary search tree
  - Searching for a particular item
- Node N in Binary Search Tree
  - N's value > all values in its left subtree T_L
  - N's value < all values in its right subtree T_R
  - Both T_L and T_R are binary search trees

The ADT Binary Search Tree

BinarySearchTree

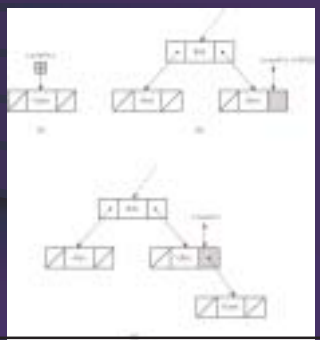
root
left subtree
right subtree

createBinarySearchTree()
destroyBinarySearchTree()
isEmpty()
searchTreeInsert()
searchTreeDelete()
searchTreeRetrieve()
preorderTraverse()
inorderTraverse()
postorderTraverse()

Efficient Search Algorithm

```
search(BinarySearchTreeNode *nd, keyType key)
{
  if (nd == NULL)
    return Not Found
  else if (key == nd->item)
    return Found
  else if (key < nd->item)
    search(nd->left, key);
  else
    search(nd->right, key);
}
```

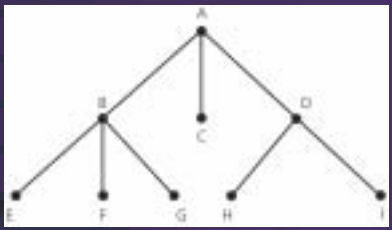
Insertion in Binary Search Tree



Insert a Node

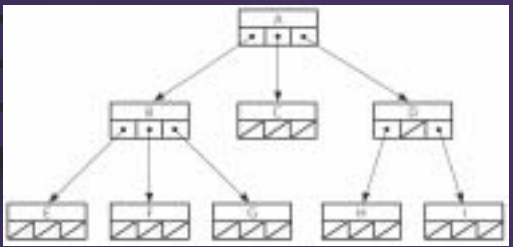
```
insertItem(BinarySearchTreeNode *nd, TreeItem item)
{
  if (nd == NULL)
    nd = new TreeNode(item, NULL, NULL);
  else if (item < nd->item)
    insertItem(nd->left, item);
  else
    insertItem(nd->right, key);
}
```

General Trees



- An n-ary tree is a tree whose nodes each can have no more than n children

General Trees



General Trees