Trees

**Concept of the tree structure**

* The concept of trees consists of abstract trees and tree data structures
* An abstract tree is defined as a collection of nodes connected in a tree structure (or tree topology), together with a set of operations such as traversal, search, insertion and deletion
* A tree structure T consist of a set of nodes V and a set E of node to node relations (called edges), denotes as T = (V,E). Each edge in E represents a connection from a node u to a node v, written as (u,v), and u is called the parent of v, and v the child of u. T satisfies the following conditions:

1. Each node has zero or more children
2. There is a unique node, called root, which is not a child of any node.
3. There is a unique path from the root to any other node.

**Recursive definition of trees**

* T = (V<E) is an empty tree is V is empty
* If V is not empty , V can be partitioned into V0, V1,…Vk, k>=0, and E can be partitioned into E1, E2,…Ek, such that V0 contains only one node, called root; Ti = (Vi,Ei) forms a tree for i=1,…,k. E0 consists of edges from the root to the roots of Ti, i=1,…,k.
* Ti is called a sub-tree of the root node
* Ti is also called a sub-tree of T at ni where ni is the root of Ti.

A tree data structure (or simply tree) is an implementation of an abstract tree in a programming language, in which the parent-child relations of nodes is represented by a specific method.

For example, a tree can be represented by the linked node representation in C using a node structure consisting of a data part and a list of pointers to its children. The following structure defines a node structure of a tree, in which each node is allowed to have at most two children, the left child and the right child.

Typedef struct node{

Int data;

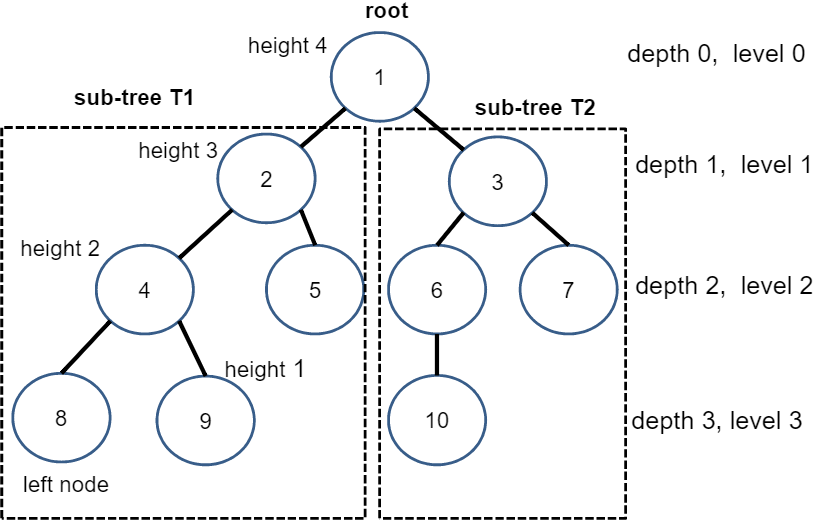
Struct node \*left;

Struct node \*right;

}TNODE;

**Terms and notation**

* Parent and child
  + If node N has a subtree T1, N1 is the root of T1, i.e., (N,N1) is an edge, then N is called parent of N1, and N1 is a child of N.
* Sibling:
  + All child nodes of a parent node are called siblings. Two nodes are called siblings if they have the same parent node.
* Ancestor and descendant:
  + If node N2 is in the subtree of N1, N1 is called an ancestor of N2, N2 is called descendant of N1.
* Leaf node:
  + A node without a child.
* Path:
  + A sequence of different connected edges, the length of a path is the number of edges in the path.
* Depth:
  + The depth of a node is the length of the path from the root to the node. The depth of the root node is zero.
* Level:
  + The set of nodes of the same depth
* Width:
  + The width of a tree is the maximum number of nodes in a level
* Height:
  + The height of a node is the number of nodes in the longest path from the node to a descendent leaf node. The height of a tree is the height of its root.
  + Other texts define height by the number of edges in the longest path



**Types of trees**

**General trees**

* A node in a general tree has zero or more children
* If a node is allowed to have at most two children, then it is called a binary tree
* If a node is allowed to have at most three children, then it is called a ternary tree
* If a node is allowed to have k children it is called a k-way tree or k-tree
* Note: if a node is allowed to have at most one child, then it has a path structure and wont be considered a tree

A simple k-tree node structure of linked node representation:

Typedef struct node{

Int data;

Struct node \*child[k];

}TNODE;

We can also add the parent pointer to parent node for efficient accessing parent node from a child node

Typedef struct node{

Int data;

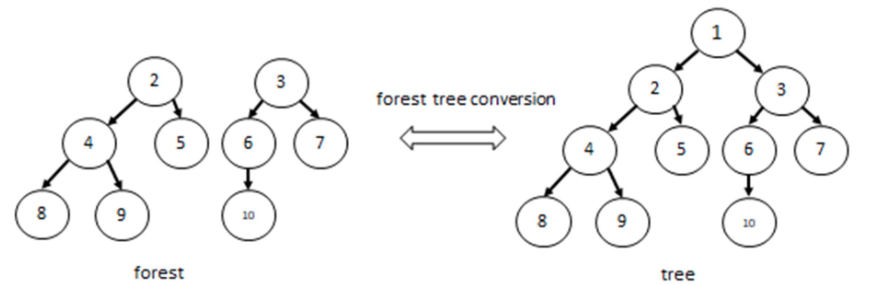
Struct node \*parent;

Struct node \*child[k];

}TNODE;

**Forests**

* A forest is a disjoint union of trees. A forest can also be defined as an ordered set of zero or more general trees.
* A forest can be obtained by deleting the root and the edges connecting the root node to nodes at level 1.
* A forest can be converted to a tree by adding a single node at the root node of the trees.



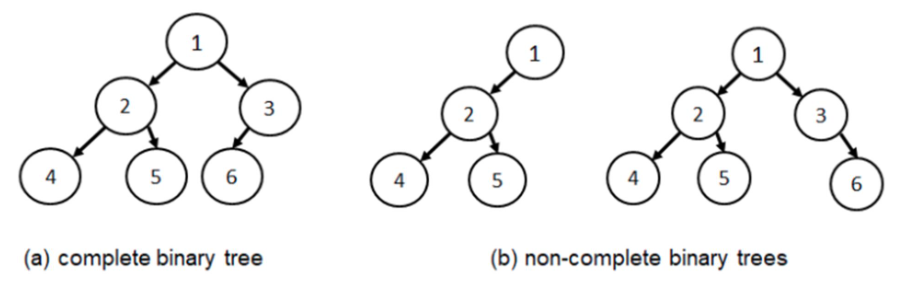
**Binary trees**

* A binary tree is a tree, in which every node is allowed to have two children, commonly referred to as left child (sub-tree), and right child (sub-tree). An empty sub-tree (or NULL child) is allowed, so left child, right child or both can be NULL.

**Complete binary trees**

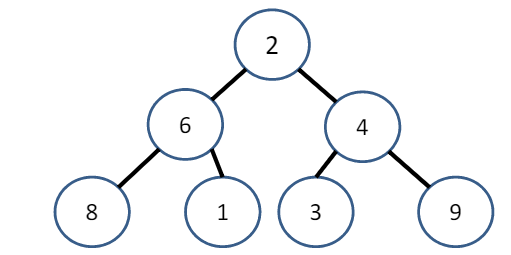
* A complete binary tree is a binary tree such that

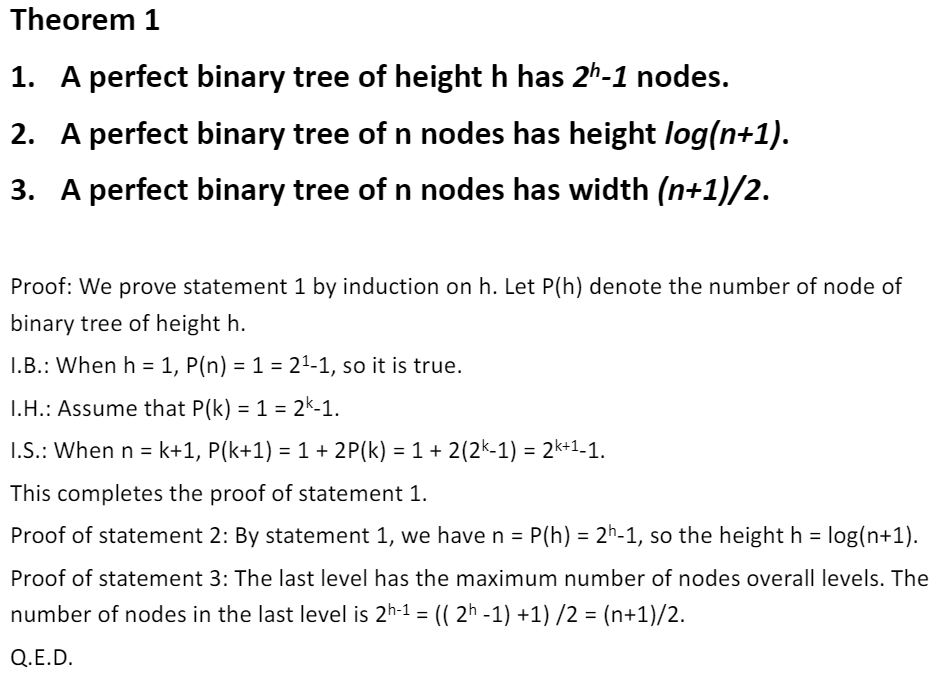
1. Each level except the last two levels is completely filled, i.e., every node in a level has two children.
2. All nodes of the last level align left, i.e., every node the second last level either has a left child or no children

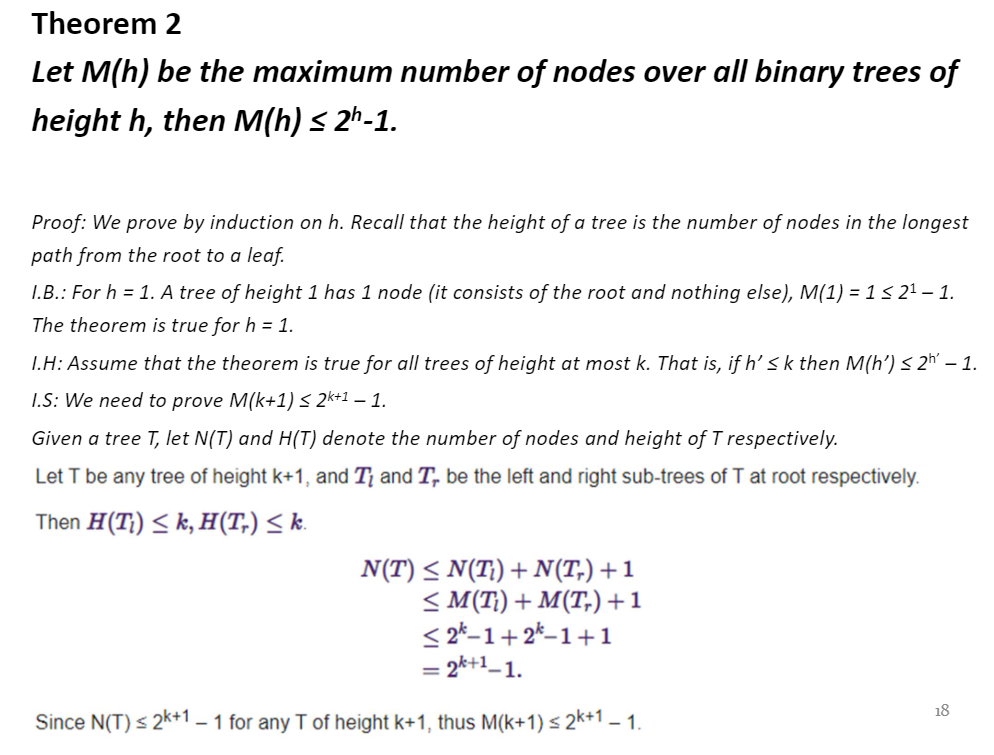


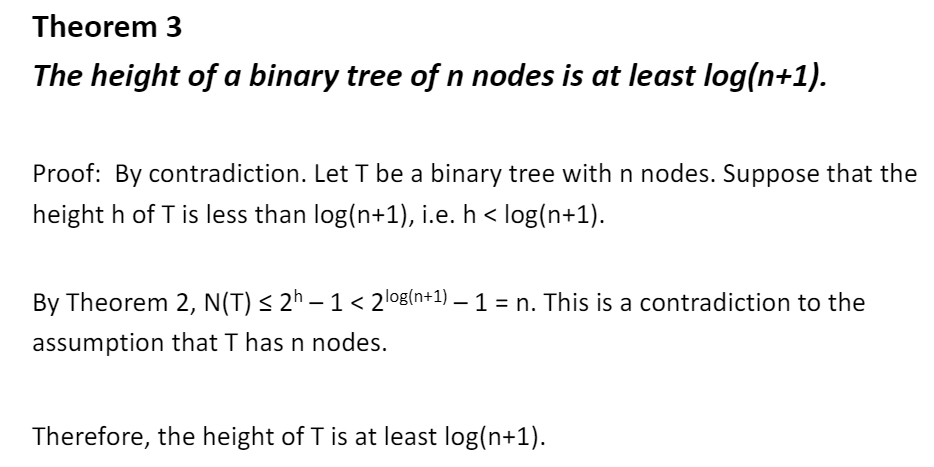
**Perfect binary trees**

* A perfect binary tree is a complete binary tree in which the every node in the second last level has two children. Every level (except the last level) is completely filled.
* A perfect binary tree of height h has 2h-1 nodes
* A perfect binary tree of n nodes has height log2(n+1)



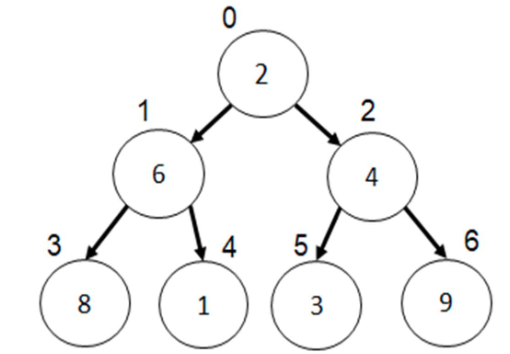






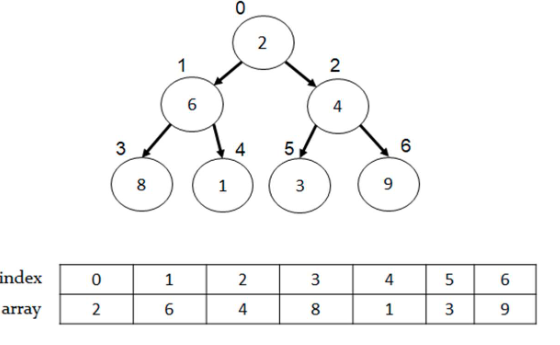
**Breadth-first-ordering of perfect binary trees**

* If nodes of a perfect binary tree is labelled 0,1,…,n-1 in breadth-first-order ,i.e., order nodes level by level starting from level 0, and left to right on each level. Then node I has parent label (i-1)/2 node I has left child label 2\*i+1, right child label 2\*i+2



**Array representation of a binary tree**

* Array representation of a binary tree T uses an array of 2h-1 data elements, where h is the height of the binary tree
* If T is a perfect binary tree, the node I (ith node) is stored at array location of index i.
* If T is not a perfect binary tree, each node has a label I, of its extended perfect binary tree, the data of the node is stored at array location of index i.
* The node at index j has left child at index 2\*j+1, the right child at index 2\*j+2
* The node at index j has parent at index (j-1)/2
* Advantages: no pointers, less overhead, efficient for random access, parent, child
* Disadvantages: waste space if not perfect binary tree



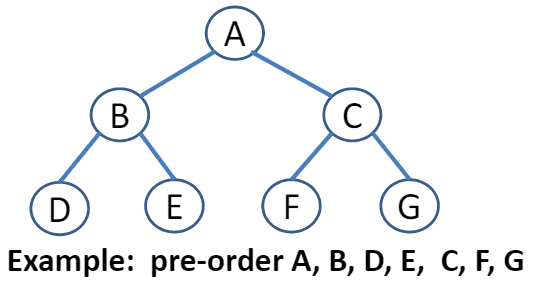
**Tree operations**

**Traversing a binary tree**

* Binary tree traversal is the process of visiting each node in the tree exactly once in a systematic way.
* There are four different algorithms for tree traversals which differ in the order of nodes being visited:
  + Pre-order
  + In-order
  + Post-order
  + Breadth-first-order

Pre-Order

* To traverse a binary tree in pre-order is to visit the tree in order or root, left subtree, right subtree



* Pre-order traversal can be done by recursive algorithm. The algorithm starts with the root node of the tree and continues by
  + Visiting the root node
  + Traversing the left subtree
  + Traversing the right subtree

Pre-order implementation:

Void print\_preorder(node \*root){

If(root){

Printf(“%d”, root->data);

Print\_preorder(root->left);

Print\_preorder(root->right);

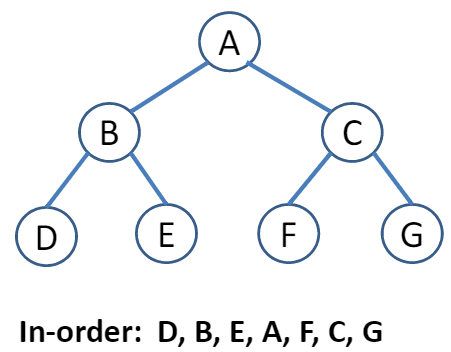
}

}

Time complexity: O(n) Space complexity: O(n) = O(h) where h=height

**In-order**

* To traverse a binary tree in in-order is to visit tree nodes in order of the left subtree, root, right subtree
* Use recursive algorithm, the algorithm starts with the root node of the tree and continues by
  + Traversing the left subtree
  + Visiting the root node
  + Traversing the right subtree



In-order implementation

Void print\_inorder(node \*root){

If(root){

Print\_inorder(root->left);

Printf(“%d”, root->data);

Print\_inorder(root->right);

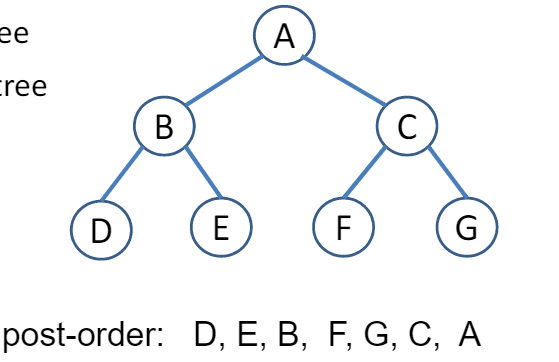
}

}

Time: O(n) space: O(h)

**Post-order**

* To traverse a binary tree in post-order is to visit tree nodes in order of left subtree, right subtree, root
* Use recursive algorithm, starts with the root node of the tree and continues by
  + Traversing the left subtree
  + Traversing the right subtree
  + Visiting the root node



Post-order implementation

Void print\_postoder(node \*root){

If(root){

Print\_postorder(root->left);

Print\_postorder(root->right);

Printf(“%d”, root->data);

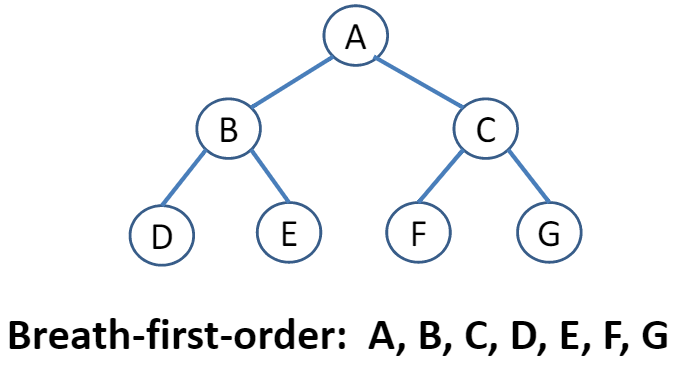
}

}

Time: O(n) space: O(h)

**Breadth-first-order**

* To traverse a binary tree in breadth-first-order is to visit tree nodes by levels from low level to high level. At each level, from left to right.



Breath-first-order implementation

* Use a queue to remember the order of processing

Void print\_bforder(tnode \*root){

If(root==NULL) return;

Qnode \*front = NULL, \*rear = NULL;

Enqueue(&front, &rear, root);

While(front){

If(front->tnp){

Printf(“%d”, front->tnp->data);

Enqueue(&front, &rear, front->tnp->left);

Enqueue(&front, &rear, front->tnp->right);

}

Dequeue(&front, &rear);

}

}

Typedef struct queue\_node{

Tnode \*tnp; //tree node pointer

Struct node \*next;

}qnode

Void enqueuer(qnode \*\*frontp, qnode \*\*rearp, tnode \*newtnode);

Void dequeue( (qnode \*\* front p, qnode \*\*rearp);

Time: O(n), Space: O(n)

**Application of tree traversals – node count**

1. Node count by method of divide and conquer

Int node\_count(node \*root){

If(root==NULL) return 0;

Return 1+node\_count(root->left)+node\_count(root->right);

}

**Search operation**

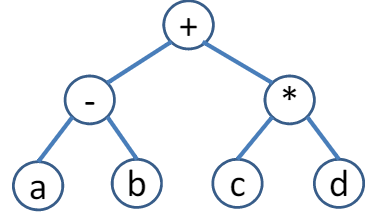
* Searching a tree is to find a node which matches the given search key. Solution: traverse the tree, return the matched node if found, otherwise return NULL
* Depth-first-search (DFS): deepened search as much as possible on each child before going to the next sibling
  + Use pre-order traversal algorithm
  + Can be implemented by recursive function or by iterative algorithm with auxiliary stack
* Breadth-first-search (BFS): use breadth-first traversal algorithm

**Applications of trees**

1. Trees are used to store data in a hierarchical structure, representing the relations of data elements
2. Trees are used to represent a collection of data objects for efficient search
3. Trees are used to implement other types of data structures like hash tables, sets, and maps
4. Binary search trees, self-balancing trees, AVL trees, red-black trees are used to store record data for efficient search, insert and delete operations.
5. Another variation of tree, B-trees are used to store tree structures on disc. They are used to index a large number of records. B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.
6. Trees are used in compiler construction
7. Trees are also used in database design
8. Trees are used in file system directories
9. Binary trees are used to represent algebraic expressions
10. Binary trees are in Huffman trees for encoding and decoding

**Expression trees**

* Binary trees can be used to represent algebraic expressions.
* Ex. (a-b)+(c\*d) can be represented as a binary tree (abstract syntax tree)



* The infix expression (a-b)+(c\*d) can be derived by in-order traversal, add (before left child, add) after right child
* Postfix expression a b – c d \* + can be derived by post-order traversal
* Prefix expression +- a b \* c d can be derived by pre-order traversal

**Huffman trees**

* Huffman coding is an entropy encoding algorithm developed by David A. Huffman, widely used a data compression technique.
* ASCII code uses fixed-length code, i.e., every character has a node of 8 bits.
* The Huffman coding uses a variable-length code table to encode a character or symbol. The variable-length code tables are derived on the basis of the estimated probability of occurrence of the character. The idea of Huffman algorithm is to encode the frequently used characters using a smaller number of bits. Huffman coding is derived by Huffman tree
* Example:

