

# Data Structures and Algorithms

## CSC 731

### Lecture Notes

# Topics

- About the Course
- Review on Data Structures and Algorithm
  - Examples for each type
  - Tree Data Structure
  - Basic Terminology
- Applying theory of Algorithm Analysis
- Tree ADT
- Traversal Algorithms
- Binary Trees
  - Binary Tree Representations
  - Binary Tree ADT
  - Binary Tree Traversals



# ABOUT THIS COURSE (1/2)

- The course is about studying techniques for solving computational problems.
- By “technique” we do not mean a programming style or a programming language but rather the approach or methodology used to solve a problem.

*Example: phone contact lookup*

- suppose **Ferdi** wants to find the name “**Moreen Cloe**” in the phone book.

*One Approach:* is to check each name in sequence, starting with the first name, until “**Moreen, Cloe**” is located.

*Note: No one, however, searches for a name this way*

*Another Approach:* is taking advantage of the fact that the names in the phone book are sorted and opens the book to where he thinks the M’s are located. If he goes too far into the book, he thumbs back a little. He continues thumbing back and forth until he locates the page containing “**Moreen, Cloe**”.

- One may recognize this second approach as a modified binary search and the first approach as a sequential search.



# ABOUT THIS COURSE (2/2)

## *Example: looking up phone contact (cont.)*

- The point to note in the phone look example is that we have two distinct approaches to solving the problem, and the approaches have nothing to do with a programming language or style.
- Nevertheless a computer program is simply one way to implement these approaches.
- In this course we shall discuss various problem-solving techniques and apply those techniques to a variety of problems.
- Applying a technique to a problem results in a step-by-step procedure for solving the problem. *This step-by-step procedure is called an algorithm for the problem.*

The purpose of studying these (several) techniques and their applications is so that, when confronted with a new problem, we have a repertoire of techniques to consider as possible ways to solve the problem.

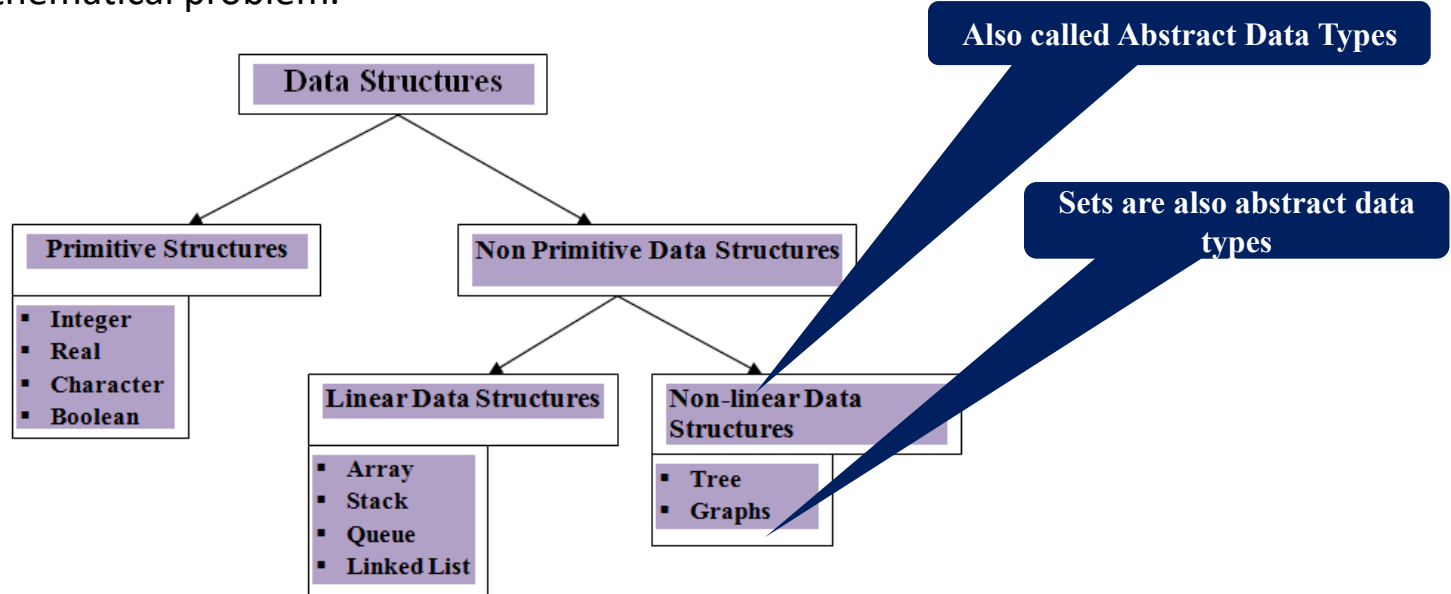


# INTRODUCTION TO THIS COURSE (Cont. )

- You may recognize this second approach as a modified binary search and the first approach as a sequential search

# Review of Data Structures

- Data Structure: A systematic way of organizing and accessing data **needed** to solve a mathematical problem.



- Common programs or algorithms uses at least one of these (List, Stack, Que etc.) linear data structures explicitly.
- However, the stack in particular is always implicitly used in a program, whether or not you declare one.



# Data Structures Operations

## ❑ Data Structures are processed by using certain operations

- **Traversing:** Accessing each record exactly once so that certain items in the record may be processed.
- **Searching:** Finding the location of the record with a given key value, or finding the location of all the records that satisfy one or more conditions.
- **Inserting:** Adding a new record to the structure.
- **Deleting:** Removing a record from the structure

## ❑ Special Data Structure-Operations

- **Sorting:** Arranging the records in some logical order (Alphabetical or numerical order).
- **Merging:** Combining the records in two different sorted files into a single sorted file.



# ALGORITHM

## *Types, Description, efficiency and Order*

- ❑ To manipulate or solve data structure problems, Algorithmic complexity analysis and design come to rescue.
- ❑ With a good algorithm, data can be organized in a data structure in such a way that all items may not be required to say, searched or deleted and the required operations can be done almost instantly.
- ❑ There are different types of Algorithms:
  - **Greedy Algorithms** : finds a locally optimum solution (without any regard for any consequence in future) and hope to find the optimal solution at the global level.
  - **Dynamic Algorithms**: works by remembering the results of the past run and using them to find new results.
  - **Divide & Conquer algorithms**: divides the algorithm into two parts; the first parts divide the problem on hand into smaller subproblems of the same type. Then, in the second part, these smaller problems are solved and then added together (combined) to produce the problem's final solution.
  - More types of Algorithms including **Max-Min Problem, Merge Sort, Binary Sort; Dynamic Programming**, will be discussed in the later parts of the course.
- ❑ Regardless of their types, algorithm can be expressed in different ways
  - Pseudocode
  - By using simple English statements(Steps).
- ❑ Once an algorithm is given for a problem and determined to be correct, the next step is to determine the amount of resources, such as time and space, that the algorithm will require





## *Algorithms: Types, Description*

about techniques for solving problems using a computer



# Algorithm and Programs

- Recall Design and Implementation Phases in System Development Life Cycle (SDLC)

## Algorithms

- Design time
- Domain or knowledge Expert
- language Independent or uses any Language
- H/w or OS independent
- It is Analyzed

## Programs

- Implementation
- Programmers
- Specified Programming Language
- H/w or OS dependent
- Testing is done

- ❑ We design an algorithm to get a solution to a given problem. A problem can be solved in more than one ways.

# Linear Data Structures

- In linear data structure, member elements form a sequence. Such linear structures can be represented in memory by using one of the two basic strategies
- By having the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called arrays. By having relationship between the elements represented by pointers. These structures are called linked lists.

# When to use them

- Arrays are useful when number of elements to be stored is fixed.
- Operations like traversal searching and sorting can easily be performed on arrays.
- On the other hand, linked lists are useful when the number of data items in the collection are likely to change.

# Nonlinear Data Structures

- In nonlinear data structures, data elements are not organized in a sequential fashion. A data item in a nonlinear data structure could be attached to several other data elements to reflect a special relationship among them and all the data items cannot be traversed in a single run.
- Data structures like multidimensional arrays, trees and graphs are some examples of widely used nonlinear data structures.

# Examples

- A Multidimensional Array is simply a collection of one-dimensional arrays.
- A Tree is a data structure that is made up of a set of linked nodes, which can be used to represent a hierarchical relationship among data elements.
- A Graph is a data structure that is made up of a finite set of edges and vertices. Edges represent connections or relationships among vertices that stores data elements.

# Difference

- Main difference between linear and nonlinear data structures lie in the way they organize data elements.
- In linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory.
- In nonlinear data structures, a data element can be attached to several other data elements to represent specific relationships that exist among them.

# Difficult to Implement

- Due to this nonlinear structure, they might be difficult to implement in computer's linear memory compared to implementing linear data structures.
- Selecting one data structure type over the other should be done carefully by considering the relationship among the data elements that needs to be stored.



# A Specific Example

- Imagine that you are hired by company XYZ to organize all of their records into a computer database.
- The first thing you are asked to do is create a database of names with all the company's management and employees.
- To start your work, you make a list of everyone in the company along with their position and other details.

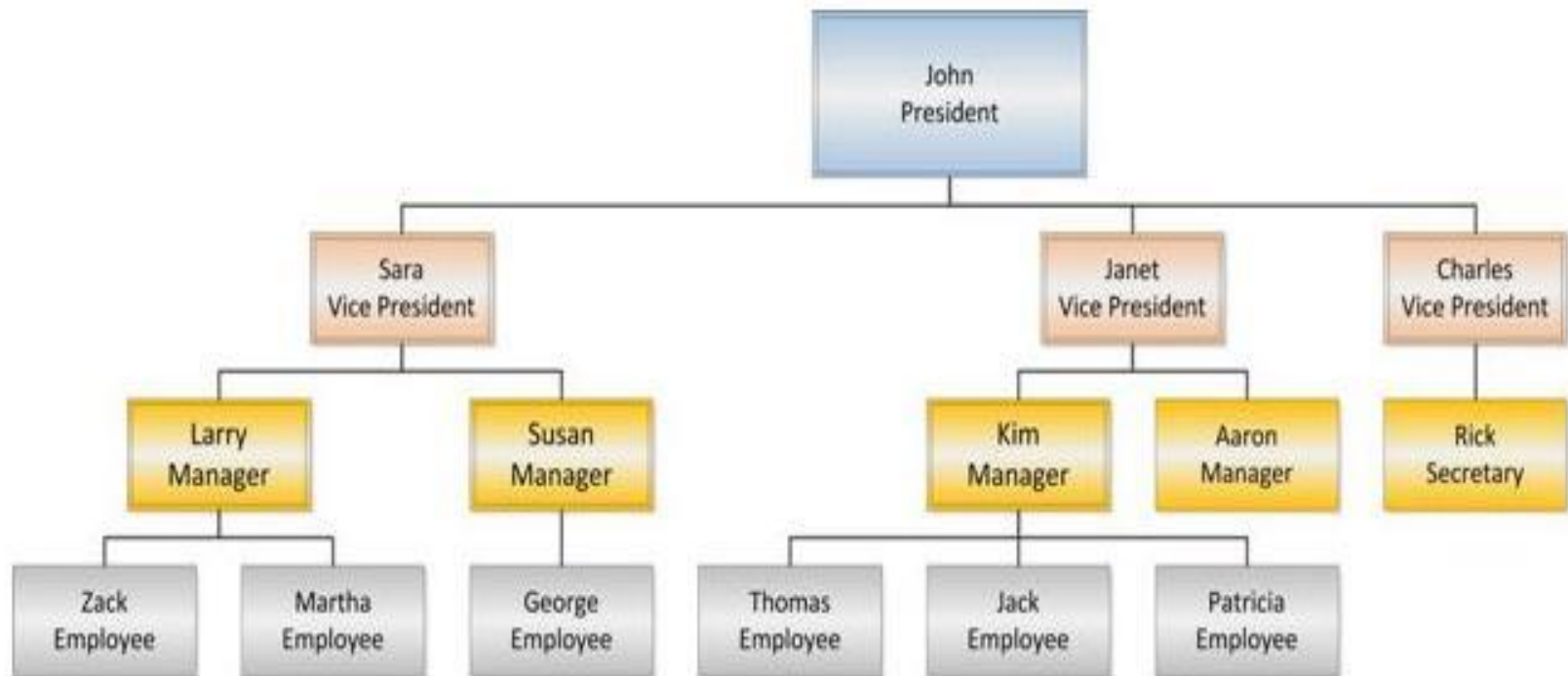
# Employees Table

Name	Position
Aaron	Manager
Charles	VP
George	Employee
Jack	Employee
Janet	VP
John	President
Kim	Manager
Larry	Manager
Martha	Employee
Patricia	Employee
Rick	Secretary
Sarah	VP
Susan	Manager
Thomas	Employee
Zack	Employee

# Disadvantages of Tables

- But this list only shows one view of the company. You also want your database to represent the relationships between management and employees at XYZ.
- Although your list contains both name and position, it does not tell you which managers are responsible for which workers and so on.
- After thinking about the problem for a while, you decide that a tree diagram is a much better structure for showing the work relationships at XYZ.

# Better Representation



# Comparison

- These two diagrams are examples of different data structures.
- In one of the data structures, your data is organized into a list. This is very useful for keeping the names of the employees in alphabetical order so that we can locate the employee's record very quickly.
- However, this structure is not very useful for showing the relationships between employees. A tree structure is much better suited for this purpose.

# Tree Data Structure

- A Tree is a nonlinear data structure that models a hierarchical organization.
- The characteristic features are that each element may have several successors (called its “children”) and every element except one (called the “root”) has a unique predecessor (called its “parent”).
- Trees are common in computer science: Computer file systems are trees, the inheritance structure for C++/Java classes is a tree.

# What is a Tree ?

- In Computer Science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization Charts
  - File Systems
  - Programming Environment

# Tree Definition

- Here is the recursive definition of an (unordered) tree:
  - A *tree* is a pair  $(r, S)$ , where  $r$  is a node and  $S$  is a set of disjoint trees, none of which contains  $r$ .
- The node  $r$  is called the *root* of the tree  $T$ , and the elements of the set  $S$  are called its *subtrees*.
- The set  $S$ , of course, may be empty.
- The elements of a tree are called its *nodes*.



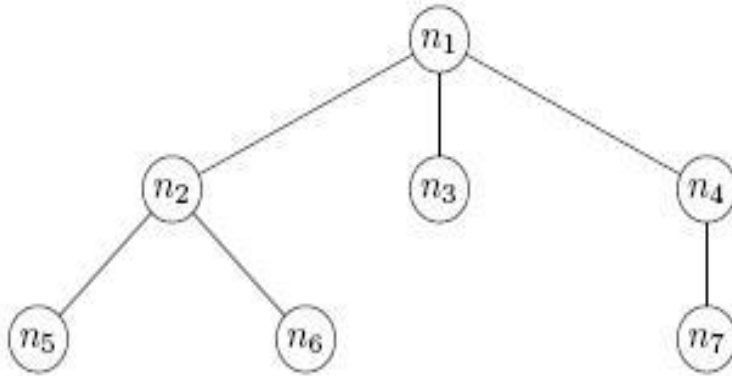
# Root, Parent and Children

- If  $T = (x, S)$  is a tree, then
  - $x$  is the root of  $T$  and
  - $S$  is its set of subtrees  $S = \{T_1, T_2, T_3, \dots, T_n\}$ .
- Each subtree  $T_j$  is itself a tree with its own root  $r_j$ .
- In this case, we call the node  $r$  the *parent* of each node  $r_j$ , and we call the  $r_j$  the *children* of  $r$ . In general.

# Basic Terminology

- A node with no children is called a *leaf*. A node with at least one child is called an *internal node*.
- The Node having further sub-branches is called *parent node*.
- Every node  $c$  other than the root is connected by an edge to some one other node  $p$  called the parent of  $c$ .
- We also call  $c$  a child of  $p$ .

# An Example



- $n_1$  is the parent of  $n_2$ ,  $n_3$  and  $n_4$ , while  $n_2$  is the parent of  $n_5$  and  $n_6$ . Said another way,  $n_2$ ,  $n_3$ , and  $n_4$  are children of  $n_1$ , while  $n_5$  and  $n_6$  are children of  $n_2$ .

# Connected Tree

- A tree is *connected* in the sense that if we start at any node  $n$  other than the root, move to the parent of  $n$ , to the parent of the parent of  $n$ , and so on, we eventually reach the root of the tree.
- For instance, starting at  $n_7$ , we move to its parent,  $n_4$ , and from there to  $n_4$ 's parent, which is the root,  $n_1$ .

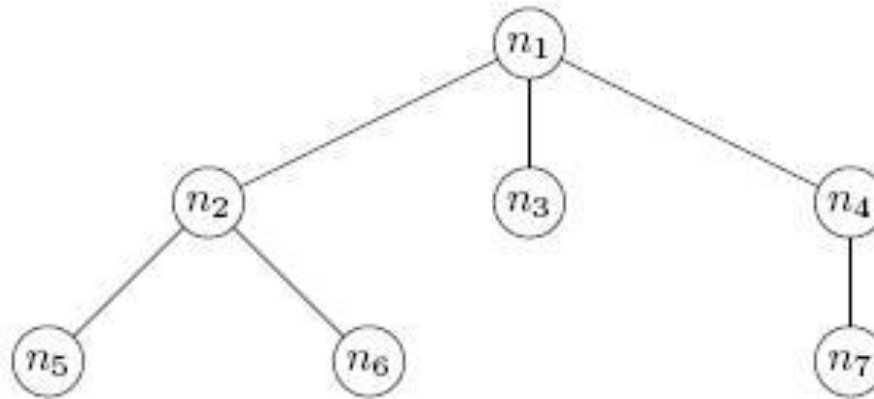
# Ancestors and Descendants

- The parent-child relationship can be extended naturally to ancestors and descendants.
- Informally, the ancestors of a node are found by following the unique path from the node to its parent, to its parent's parent, and so on.
- The descendant relationship is the inverse of the ancestor relationship, just as the parent and child relationships are inverses of each other.

# Path and Path Length

- More formally, suppose  $m_1, m_2, \dots, m_k$  is a sequence of nodes in a tree such that  $m_1$  is the parent of  $m_2$ , which is the parent of  $m_3$ , and so on, down to  $m_{k-1}$ , which is the parent of  $m_k$ . Then  $m_1, m_2, \dots, m_k$  is called a path from  $m_1$  to  $m_k$  in the tree. The path length or length of the path is  $k - 1$ , one less than the number of nodes on the path.

# In our Example



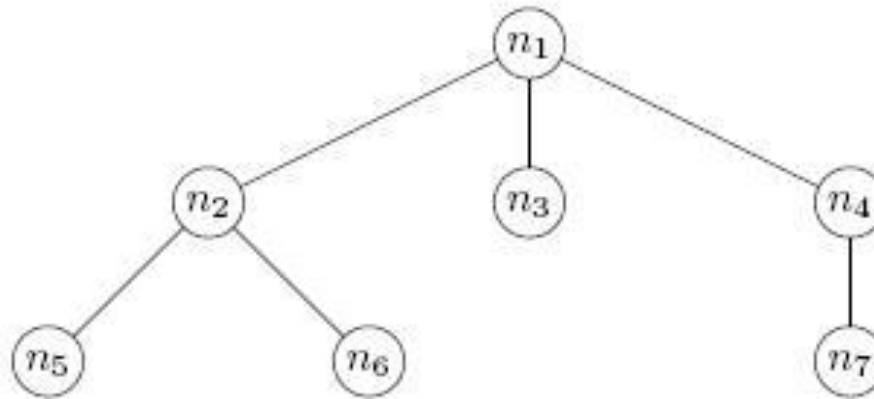
- $n_1, n_2, n_6$  is a path of length 2 from the root  $n_1$  to the node  $n_6$ .

# Height and Depth

- In a tree, the height of a node  $n$  is the length of a longest path from  $n$  to a leaf. The height of the tree is the height of the root.
- The depth, or level, of a node  $n$  is the length of the path from the root to  $n$ .



# In our Example

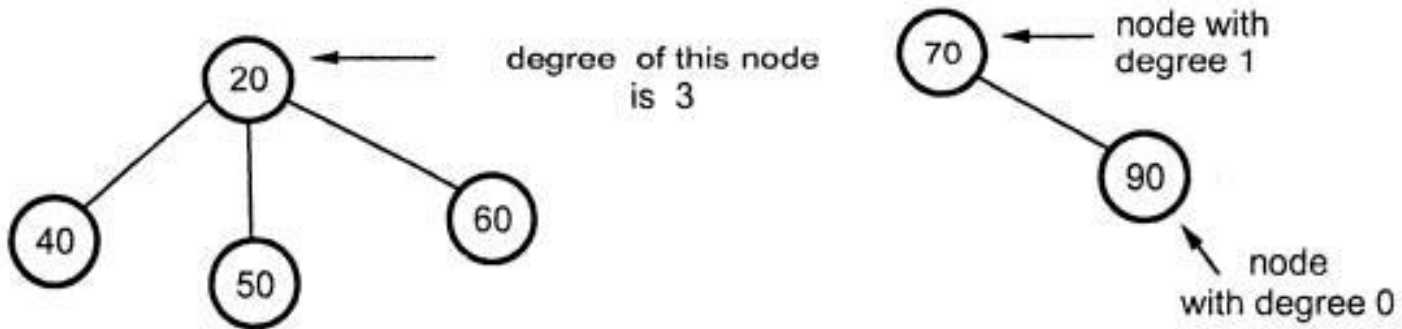


- node  $n_1$  has height 2,  $n_2$  has height 1, and leaf  $n_3$  has height 0. In fact, any leaf has height 0. The height of the tree is 2. The depth of  $n_1$  is 0, the depth of  $n_2$  is 1, and the depth of  $n_5$  is 2.

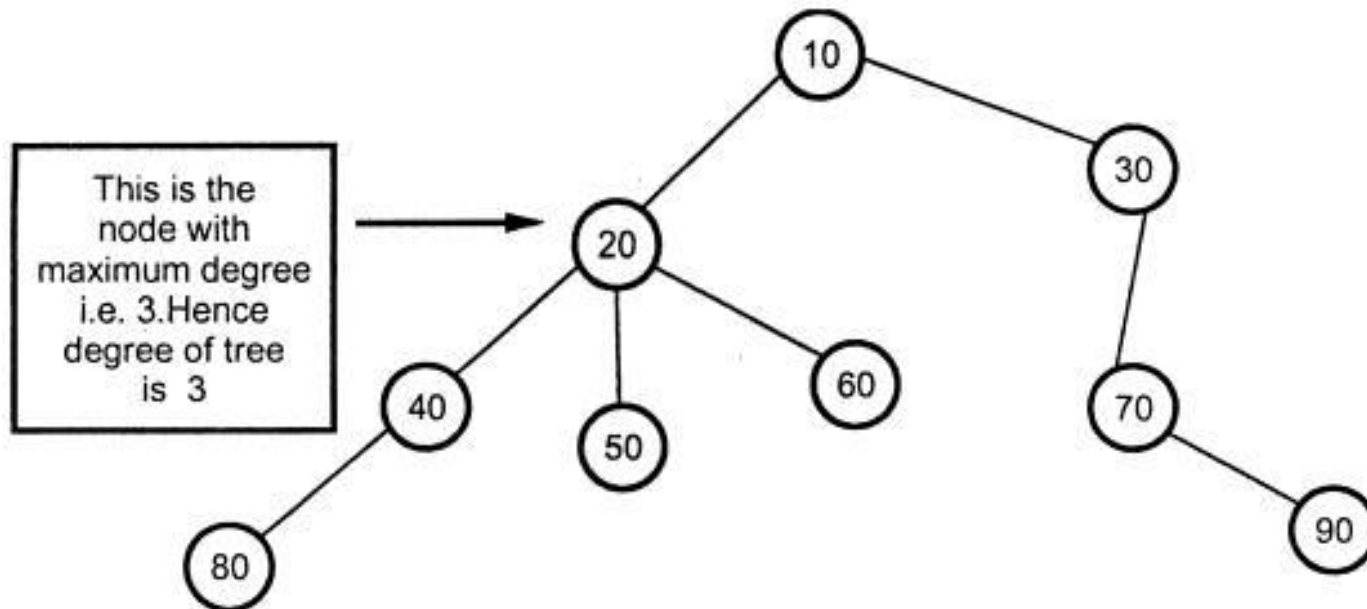
# Other Terms

- The size of a tree is the number of nodes it contains.
- The total number of subtrees attached to that node is called the degree of the node.
- Degree of the Tree is nothing but the maximum degree in the tree.
- The nodes with common parent are called siblings or brothers.

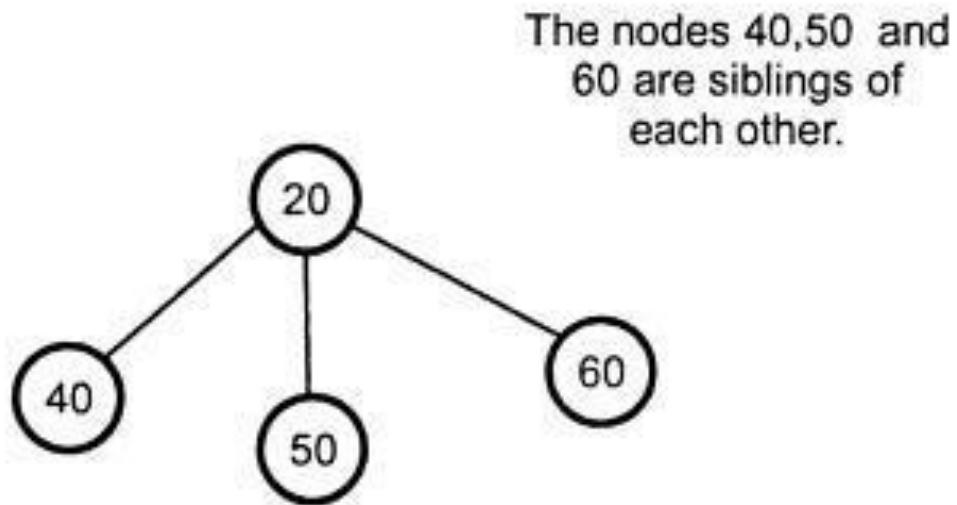
# Degree



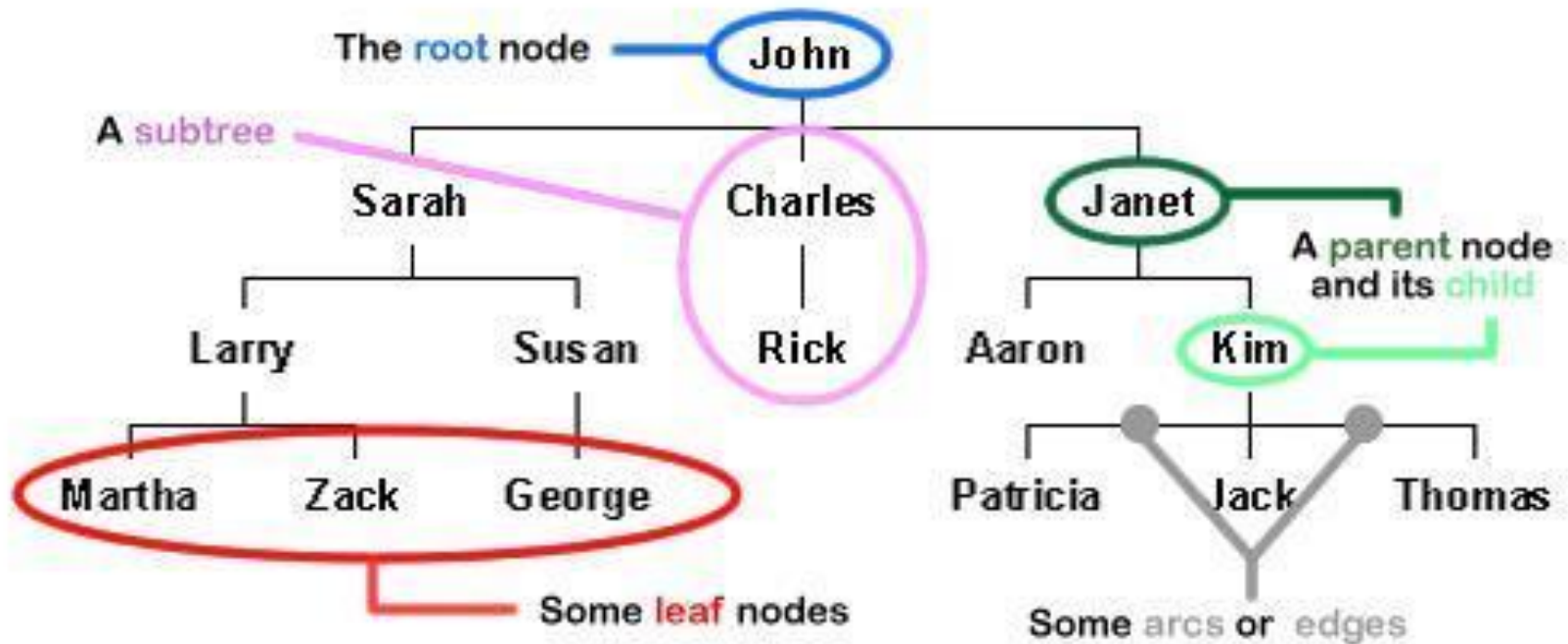
# Degree of the Tree



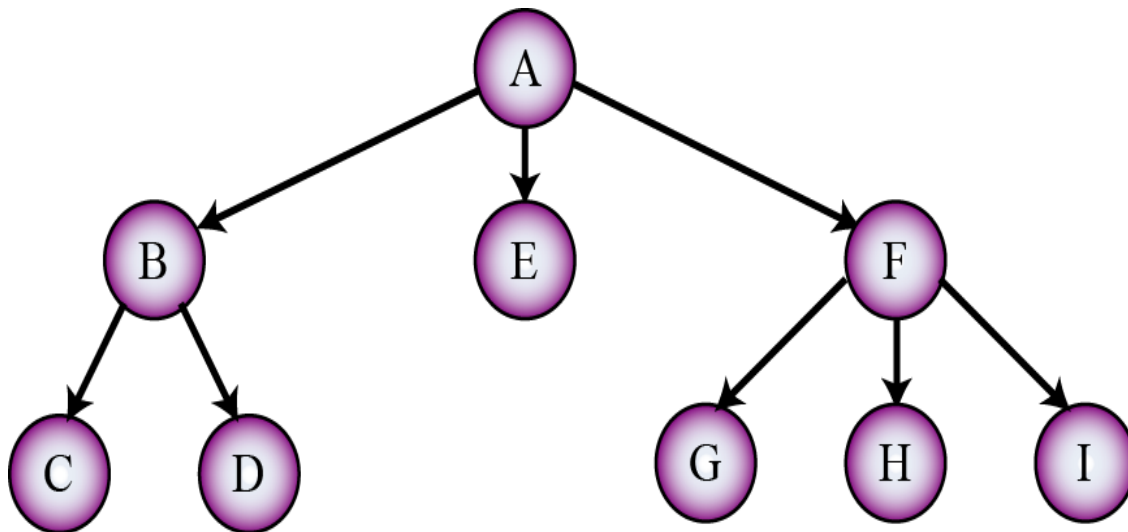
# Siblings



# Terminology Explained



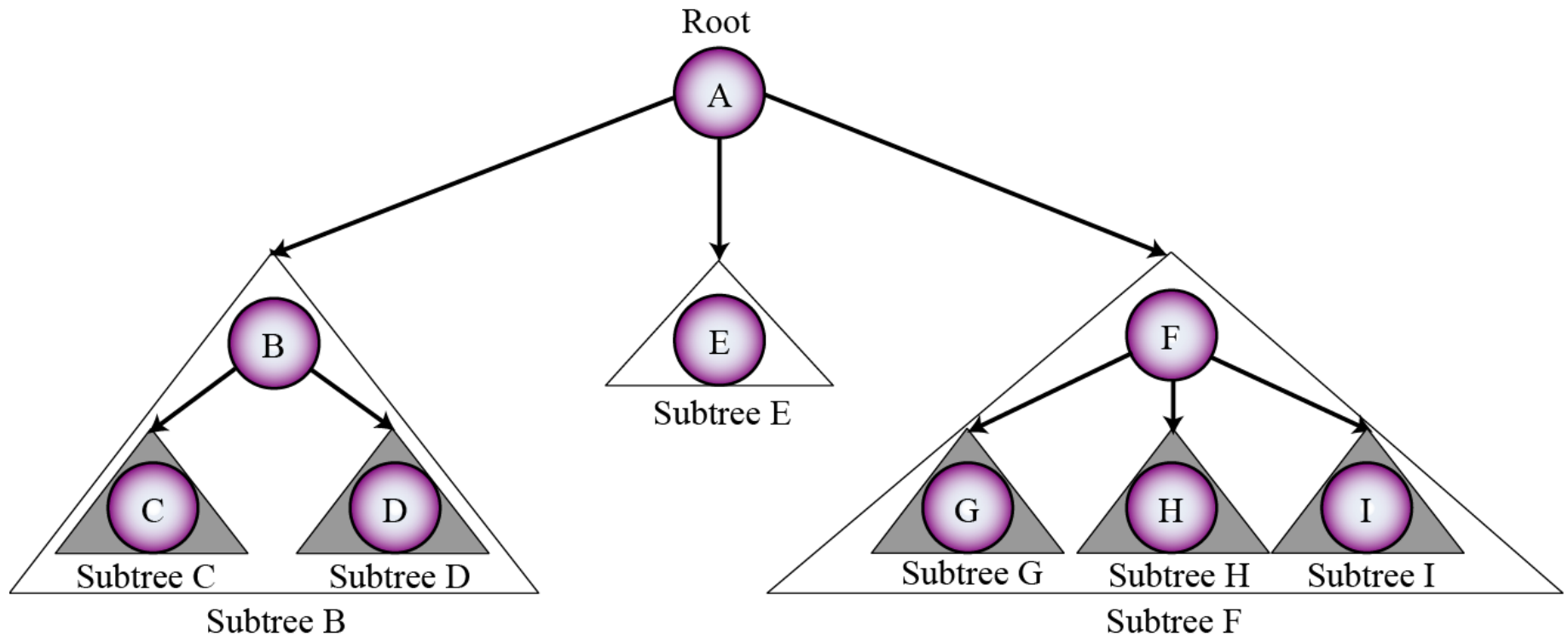
# Another Example



A: root  
B and F: internal nodes  
C, D, E, G, H, and I: leaves

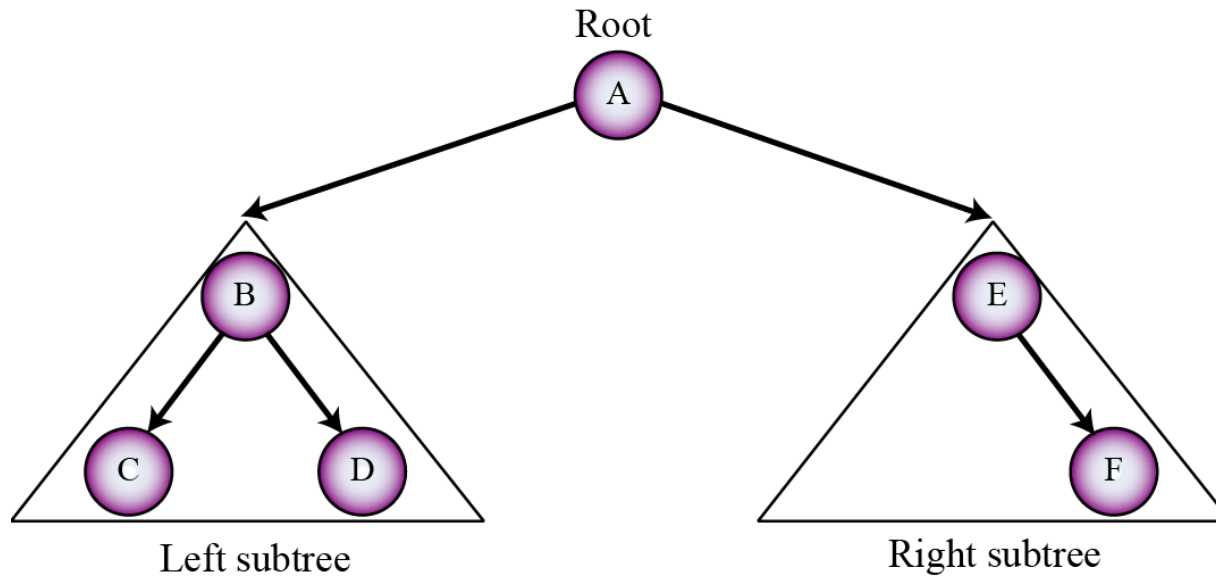
Nodes

# Subtrees





# Binary Tree



- A binary tree is a tree in which no node can have more than two subtrees. In other words, a node can have zero, one or two subtrees.

# Tree ADT

- The tree ADT stores elements at positions, which are defined relative to neighboring positions.
- Positions in a tree are its nodes, and the neighboring positions satisfy the parent-child relationships that define a valid tree.
- Tree nodes may store arbitrary objects.

# Methods of a Tree ADT

- As with a list position, a position object for a tree supports the method: `element()` : that returns the object stored at this position (or node).
- The tree ADT supports four types of methods:
  - Accessor Methods
  - Generic Methods
  - Query Methods
  - Update Methods

# Accessor Methods

- We use positions to abstract nodes. The real power of node positions in a tree comes from the accessor methods of the tree ADT that return and accept positions, such as the following:
  - `root()`: Return the position of the tree's root; an error occurs if the tree is empty.
  - `parent(p)`: Return the position of the parent of `p`; an error occurs if `p` is the root.
  - `children(p)`: Return an iterable collection containing the children of node `p`.
    - Iterable - you can step through (i.e. *iterate*) the object as a collection

# Make a note of it

- If a tree  $T$  is ordered, then the iterable collection,  $\text{children}(p)$ , stores the children of  $p$  in their linear order.
- If  $p$  is an external node, then  $\text{children}(p)$  is empty.
- Any method that takes a position as argument should generate an error condition if that position is invalid.

# Generic Methods

- Tree ADT also supports the following Generic Methods:
  - size(): Return the number of nodes in the tree.
  - isEmpty(): Test whether the tree has any nodes or not.
  - Iterator(): Return an iterator of all the elements stored at nodes of the tree.
  - positions(): Return an iterable collection of all the nodes of the tree.

# Query Methods

- In addition to the above fundamental accessor methods, the tree ADT also supports the following Boolean query methods:
  - isInternal(p): Test whether node p is an internal node
  - isExternal(p): Test whether node p is an external node
  - isRoot(p): Test whether node p is the root node

(These methods make programming with tree easier and more readable, since we can use them in the conditionals of if -statements and while - loops, rather than using a non-intuitive conditional).

# Update Methods

- The tree ADT also supports the following update method:
  - `replace(p, e)`: Replace with `e` and return the element stored at node `p`.

(Additional update methods may be defined by data structures implementing the tree ADT)



# Tree ADT Exceptions

- An interface for the tree ADT uses the following exceptions to indicate error conditions:
  - `InvalidPositionException`: This error condition may be thrown by any method taking a position as an argument to indicate that the position is invalid.
  - `BoundaryViolationException`: This error condition may be thrown by method `parent()` if it's called on the root.
  - `EmptyTreeException`: This error condition may be thrown by method `root()` if it's called on an empty tree.

# Data Structure Manipulation



# Applying the Theory of Algorithm complexity Analysis

## Content

- Analysis of While and For Loop
- Applying the theory of Algorithm Analysis



## Analysis of the **While** and For Loop

Consider a random piece of code below where While loop is used:

```
i = 0;  
while (i < n)  
{  
    stmt;  
    i ++;  
}
```

**Question:** What is the time complexity ?



# Analysis of the **While** and For Loop (cont.)

## solution

```

i = 0; — — — — — 1
while (i < n) — — — n + 1
{
    stmt; — — — — — n
    i + +; — — — — — n
}

```

---


$$f(n) = 3n + 2$$

$$\text{growth} \rightarrow \theta(n)$$



## Analysis of the **While** and For Loop (cont.)

Consider another random piece of code below where *for* loop is used instead:

```
for = (i = 0;    i < n; i + +)  
{  
    stmt; - - - - - n  
    i + +; - - - - - n  
}
```



$$f(n) = 3n + 2$$

$$f(n) = 2n + 1$$

$$\text{growth} \rightarrow \overline{o(n)}$$



# Applying Theory of Algorithm Analysis

When applying the theory of algorithm analysis, one must sometimes be aware of the time it takes to execute the basic operation, the overhead instructions, and the control instructions on the actual computer on which the algorithm is implemented.

By “**overhead instructions**” we mean instructions such as initialization instructions before a loop. The number of times these instructions execute does not increase with input size.

By “**control instructions**” we mean instructions such as incrementing an index to control a loop. The number of times these instructions execute increases with input size.



## Applying Theory of Algorithm Analysis (cont.)

The basic operation, overhead instructions, and control instructions are all properties of an algorithm and the implementation of the algorithm. They are not properties of a problem. This means that they are usually different for two different algorithms for the same problem.



# Applying Theory of Algorithm Analysis

Consider our Array Look up example:

The maximum element in an array can be looked up using a simple piece of code such as this piece of Javascript code. Given an input array A of size n:

```
var M = A[ 0 ];
```

```
for ( var i = 0; i < n; ++i ) {  
  if ( A[ i ] >= M ) {  
    M = A[ i ];  
  }  
}
```



Looking at the for body, we have an array lookup operation and a comparison that happen always: that's two instructions there.

But the **if** loop's body may run or may not run, depending on what the array values actually are.

If it happens to be so that **A[ i ] >= M**, then we'll run these two additional instructions:

1. an array lookup
2. and an assignment



**M = A[ i ]**

1. an array lookup
2. and an assignment



$M = A[i]$

But now we can't define an  $f(n)$  as easily, because our number of instructions doesn't depend solely on  $n$  but also on our input.

➡ For example, for  $A = [1, 2, 3, 4]$  the algorithm will need more instructions than for  $A = [4, 3, 2, 1]$ .

Due to cases of these nature in the above example, when analysing algorithms, we often consider the worst-case scenario.

To ascertain Worst Scenario:

- What's the worst that can happen for our algorithm?
- When does our algorithm need the most instructions to complete?



In this case: when we have an array in increasing order such as  $A = [1, 2, 3, 4]$ .



## Worst-Case Analysis (Cont.)

Algorithmic Example of the Worst case scenario:

Sorting Array in Ascending Order:

The **ascending order** arranges the elements in the lowest to highest order. It is also known as **natural order** or **numerical order**. We can perform sorting in the following ways:

- Using the sort() Method
- Without using the method:
  - Using the for Loop
  - Using the User Defined Method

# Insert An Element

- Given a sorted list/sequence, insert a new element
- Given 3, 6, 9, 14
- Insert 5
- Result 3, 5, 6, 9, 14

# Insert an Element

- 3, 6, 9, 14     insert 5
- Compare new element (5) and last one (14)
- Shift 14 right to get 3, 6, 9, , 14
- Shift 9 right to get 3, 6, , 9, 14
- Shift 6 right to get 3, , 6, 9, 14
- Insert 5 to get 3, 5, 6, 9, 14

# Insert An Element

```
// insert t into a[0:i-1]
int j;
for (j = i - 1; j >= 0 && t < a[j]; j--)
    a[j + 1] = a[j];
a[j + 1] = t;
```

# Insertion Sort

- Start with a sequence of size 1
- Repeatedly insert remaining elements

# Insertion Sort

- Sort 7, 3, 5, 6, 1
- Start with 7 and insert 3  $\Rightarrow$  3, 7
- Insert 5  $\Rightarrow$  3, 5, 7
- Insert 6  $\Rightarrow$  3, 5, 6, 7
- Insert 1  $\Rightarrow$  1, 3, 5, 6, 7



# Insertion Sort

```
for (int i = 1; i < a.length; i++)  
{  
    // insert a[i] into a[0:i-1]  
    // code to insert comes here  
}
```

# Insertion Sort

```
for (int i = 1; i < a.length; i++)  
{// insert a[i] into a[0:i-1]  
    int t = a[i];  
    int j;  
    for (j = i - 1; j >= 0 && t < a[j]; j--)  
        a[j + 1] = a[j];  
    a[j + 1] = t;  
}
```

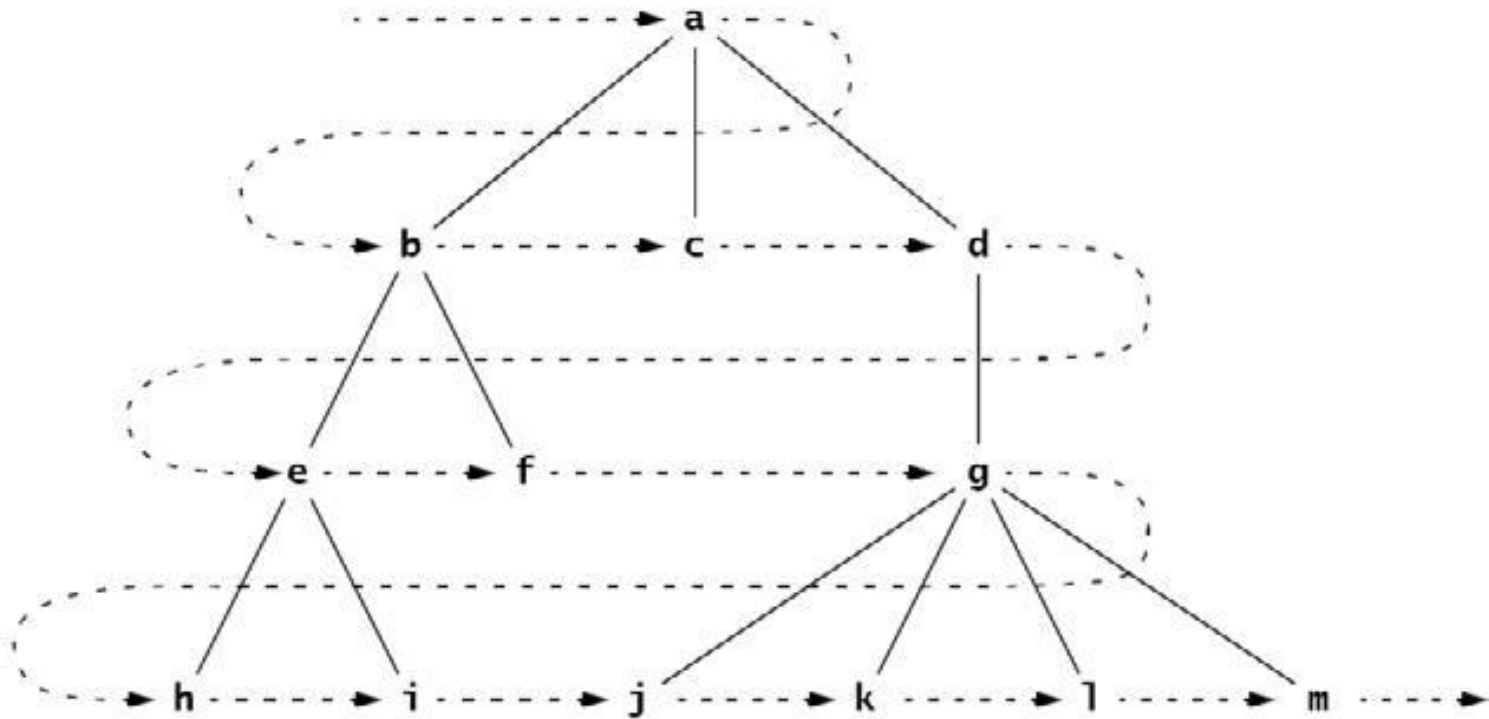
# Traversal Algorithms

- A traversal (circumnavigation) algorithm is a method for processing a data structure that applies a given operation to each element of the structure.
- For example, if the operation is to print the contents of the element, then the traversal would print every element in the structure.
- The process of applying the operation to an element is called visiting the element. So executing the traversal algorithm causes each element in the structure to be visited.

# Level Order Traversal

- The level order traversal algorithm visits the root, then visits each element on the first level, then visits each element on the second level, and so forth, each time visiting all the elements on one level before going down to the next level.
- If the tree is drawn in the usual manner with its root at the top and leaves near the bottom, then the level order pattern is the same left-to-right top-to-bottom pattern that you follow to read English text.

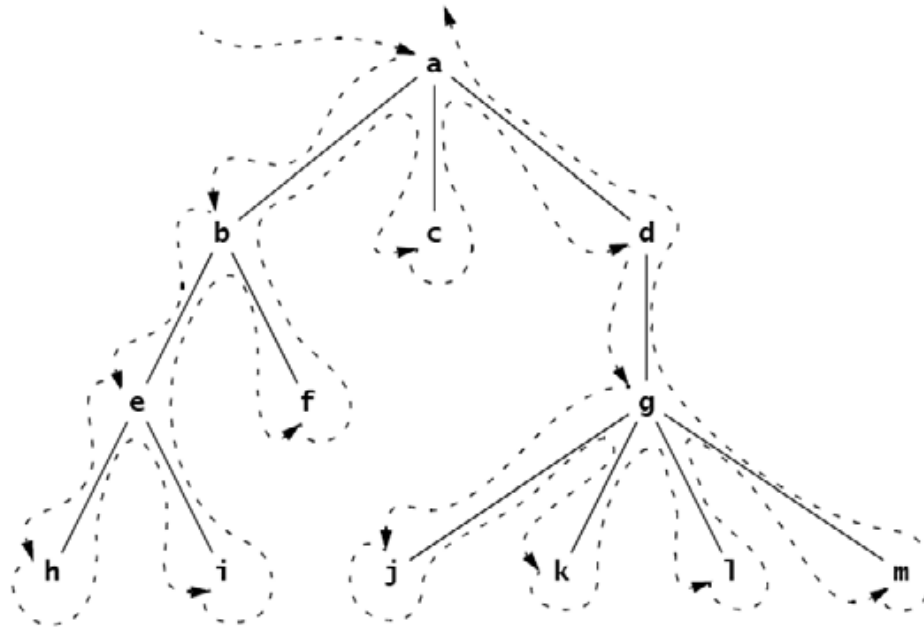
# Level Order Example



# Level order Traversal - Algorithm

- To traverse a nonempty ordered tree:
  1. Initialize a queue.
  2. Enqueue the root.
  3. Repeat steps 4–6 until the queue is empty.
    4. Dequeue node x from the queue.
    5. Visit x.
    6. Enqueue all the children of x in order.

# Preorder Traversal



The preorder traversal of the tree shown above would visit the nodes in this order: a, b, e, h, i, f, c, d, g, j, k, l, m.

Note that the preorder traversal of a tree can be obtained by circumnavigating the tree, beginning at the root and visiting each node the first time it is encountered on the left.

# Preorder Traversal - Algorithm

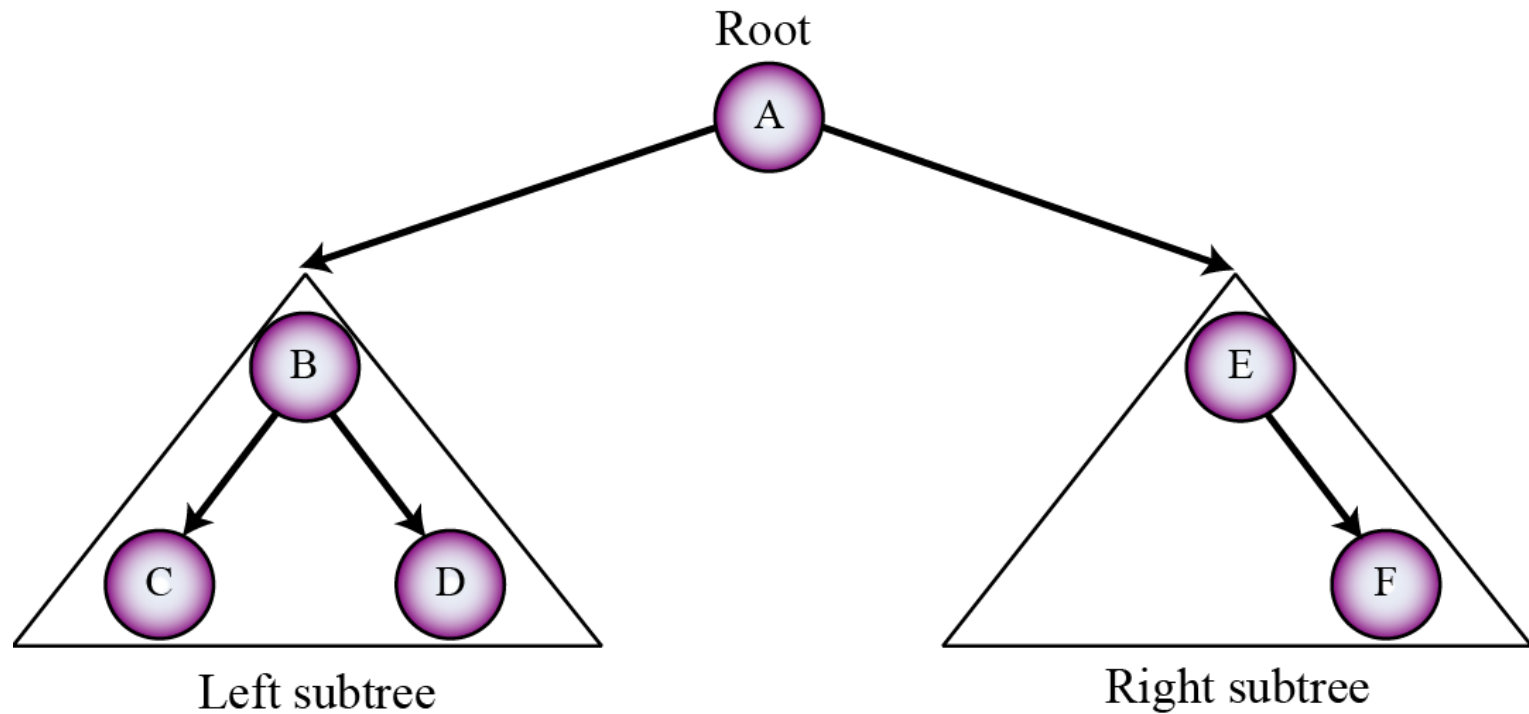
- To traverse a nonempty ordered tree:
  1. Visit the root.
  2. Do a recursive preorder traversal of each subtree in order.
- The postorder traversal algorithm does a postorder traversal recursively to each subtree before visiting the root.



# Binary Trees

- A binary tree is either the empty set or a triple  $T = (x, L, R)$ , where  $x$  is a node and  $L$  and  $R$  are disjoint binary trees, neither of which contains  $x$ .
- The node  $x$  is called the root of the tree  $T$ , and the subtrees  $L$  and  $R$  are called the left subtree and the right subtree of  $T$  rooted at  $x$ .

# Binary Tree



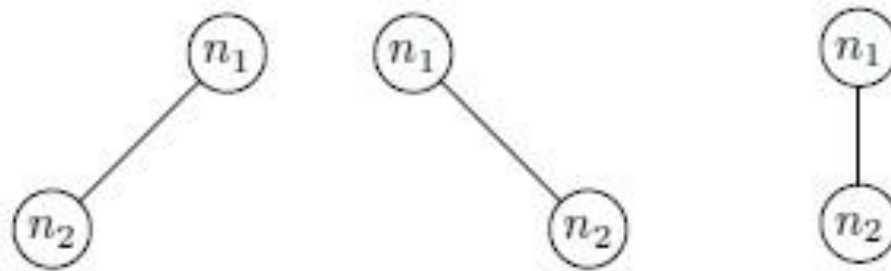
# Terminologies

- The definitions of the terms *size*, *path*, *length* of a path, *depth* of a node, *level*, *height*, *interior* node, *ancestor*, *descendant*, and *subtree* are the same for binary trees as for general trees.

# Trees and Binary Trees - Difference

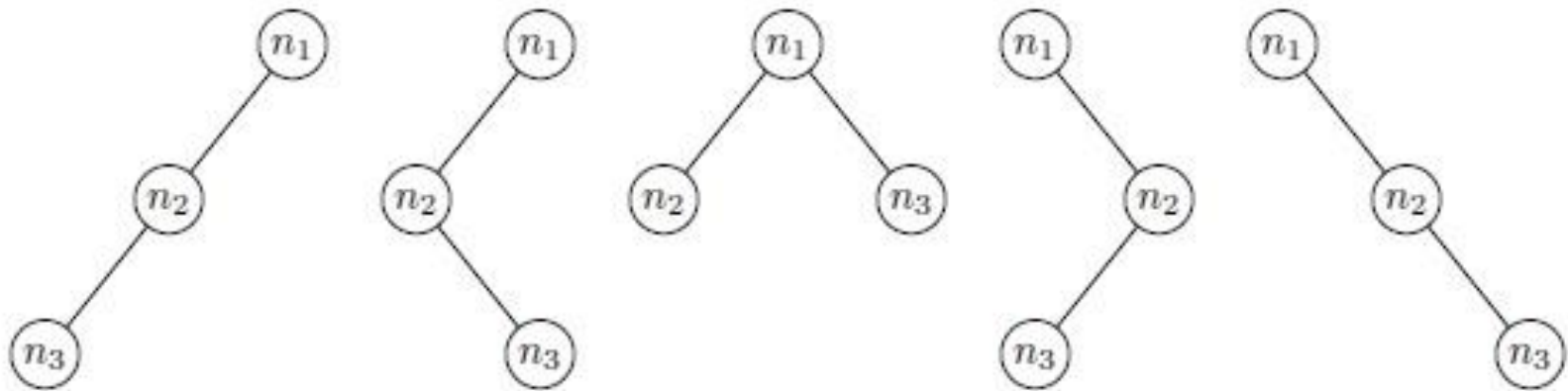
- It is important to understand that while binary trees require us to distinguish whether a child is either a left child or a right child, ordinary trees require no such distinction.
- There is another technical difference. While trees are defined to have at least one node, it is convenient to include the empty tree, the tree Empty tree with no nodes, among the binary trees.

# Difference Continued

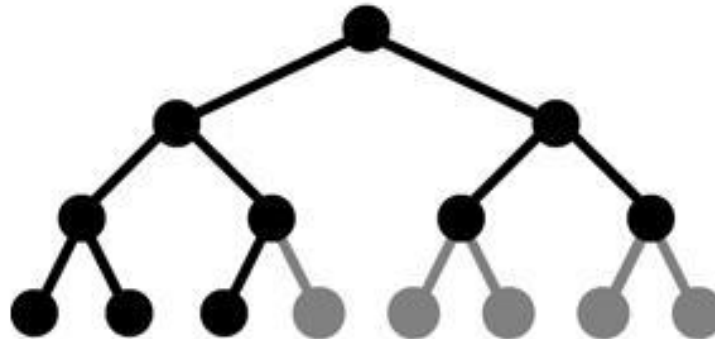
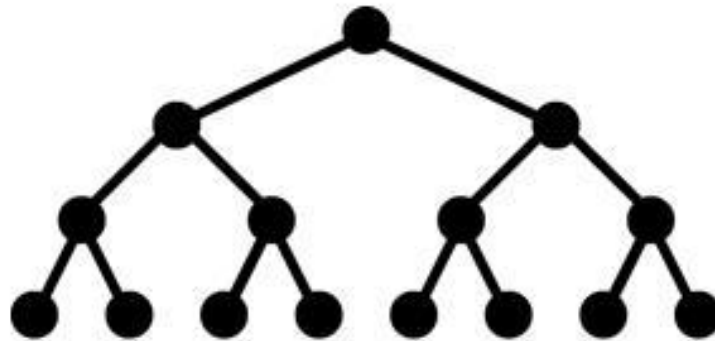


- That is, binary trees are not just trees all of whose nodes have two or fewer children.
- Not only are the two trees in the above Figure different from each other, but they have no relation to the ordinary tree consisting of a root and a single child of the root:

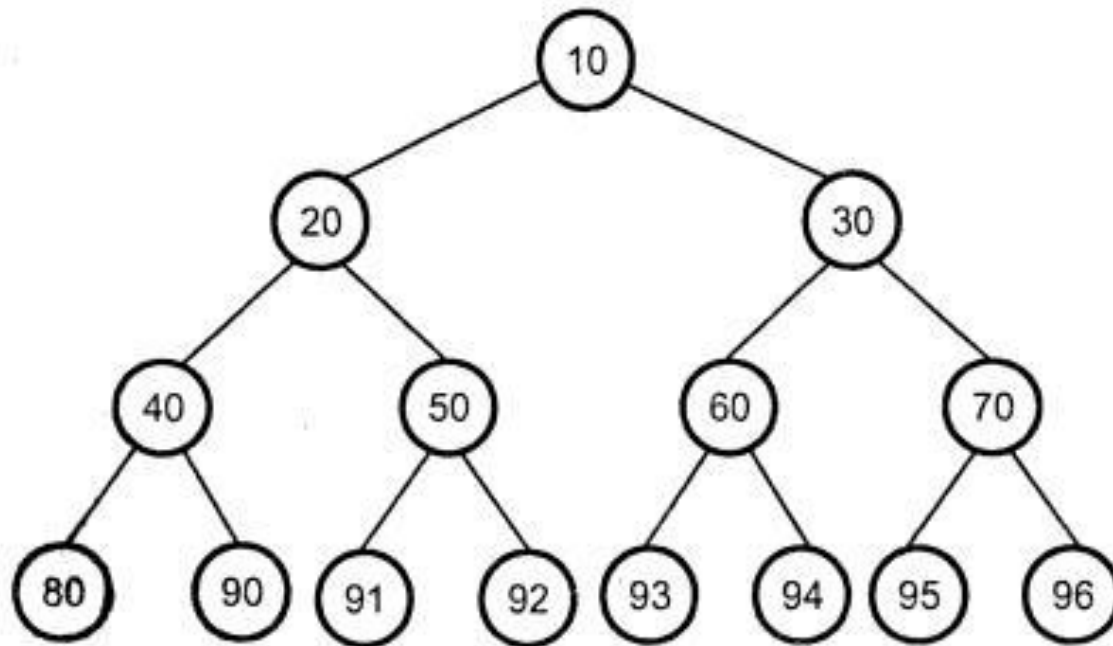
# Five binary trees with three nodes



# Full and Complete Binary Trees

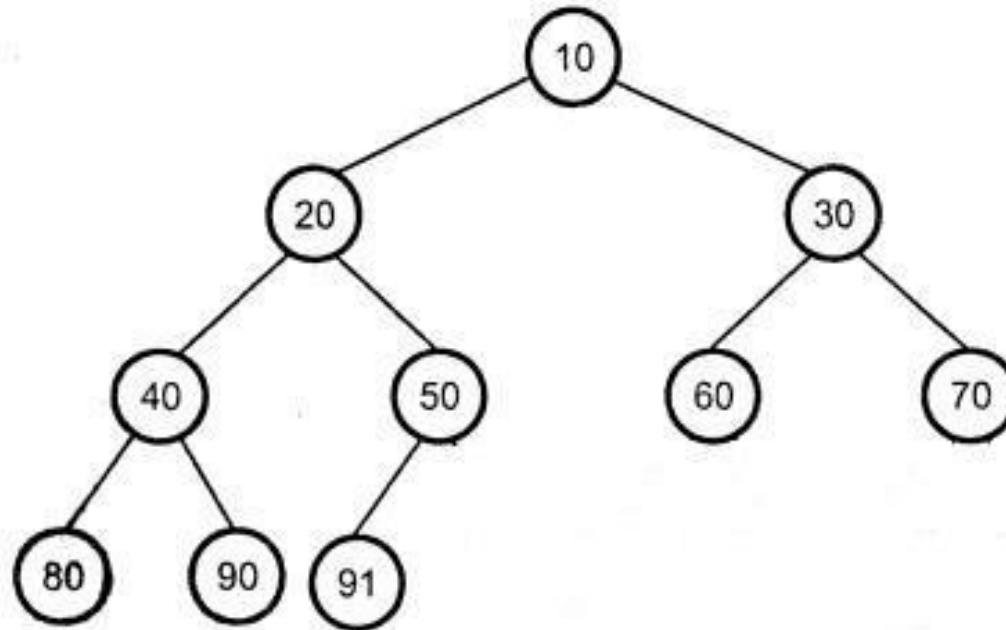


# Full Binary Tree





# Complete Binary Tree



# Full Binary Trees

- The full binary tree of height  $h$  has  $l = 2^h$  leaves and  $m = 2^h - 1$  internal nodes.
- The full binary tree of height  $h$  has a total of  $n = 2^{h+1} - 1$  nodes.
- The full binary tree with  $n$  nodes has height  $h = \lg(n+1) - 1$ .

# Complete Binary Tree

- In a complete binary tree of height  $h$ ,  
$$h + 1 \leq n \leq 2^{h+1} - 1 \text{ and } h = \lfloor \lg n \rfloor$$

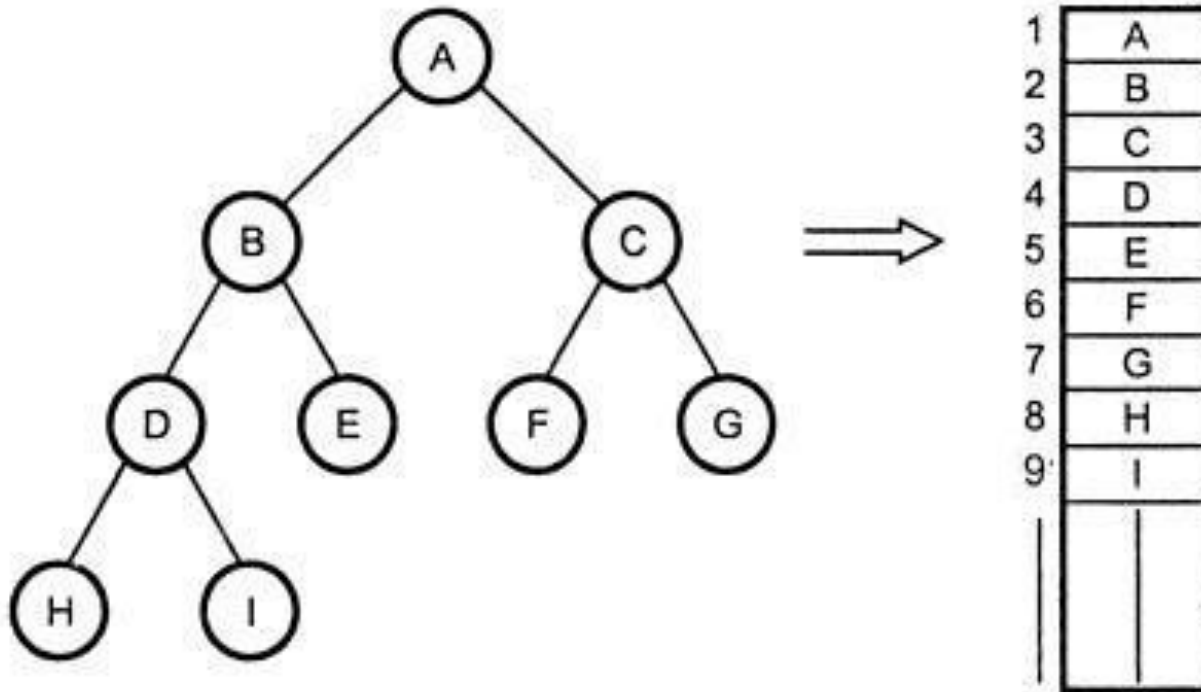
# Make a Note

- Complete binary trees are important because they have a simple and natural implementation using ordinary arrays.
- The *natural mapping* is actually defined for any binary tree: Assign the number 1 to the root; for any node, if  $i$  is its number, then assign  $2i$  to its left child and  $2i+1$  to its right child (if they exist).
- This assigns a unique positive integer to each node. Then simply store the element at node  $i$  in  $a[i]$ , where  $a[]$  is an array.

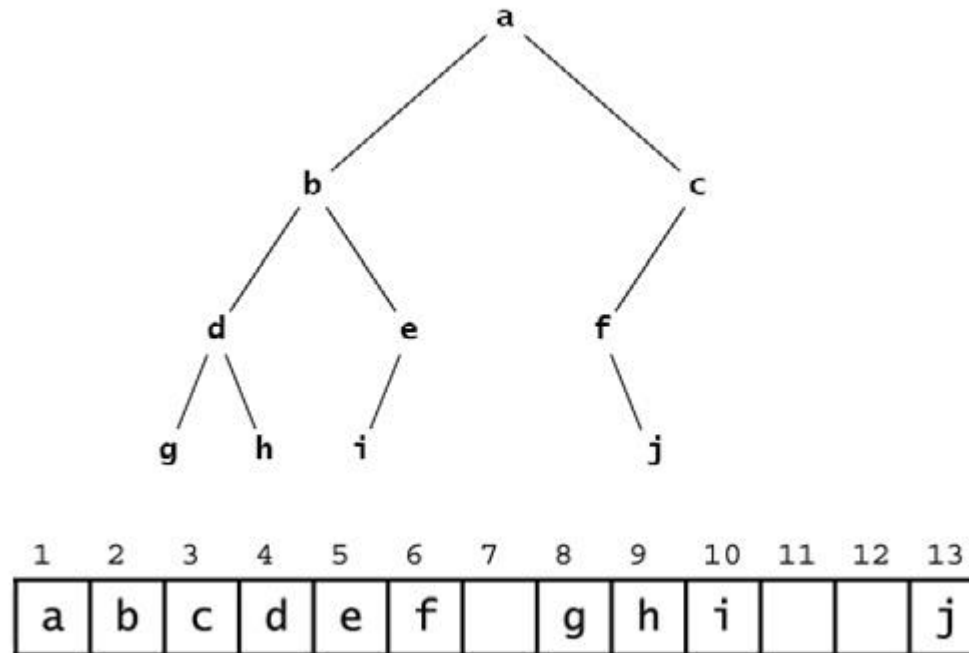
# Binary Tree Representations

- If a complete binary tree with  $n$  nodes (depth =  $\log n + 1$ ) is represented sequentially, then for any node with index  $i$ ,  $1 \leq i \leq n$ , we have:
  - $parent(i)$  is at  $i/2$  if  $i \neq 1$ . If  $i=1$ ,  $i$  is at the root and has no parent.
  - $leftChild(i)$  is at  $2i$  if  $2i \leq n$ . If  $2i > n$ , then  $i$  has no left child.
  - $rightChild(i)$  is at  $2i+1$  if  $2i+1 \leq n$ . If  $2i+1 > n$ , then  $i$  has no right child.

# Array Implementation



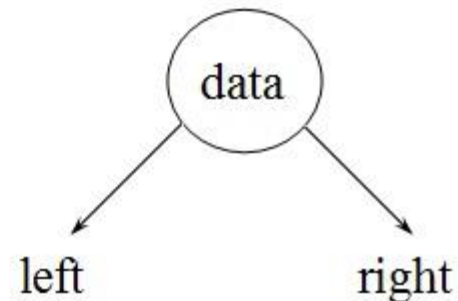
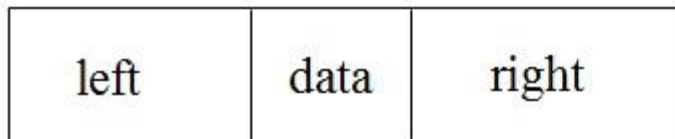
# The Disadvantage



- Figure above shows the incomplete binary tree and the natural mapping of its nodes into an array which leaves some gaps.

# Linked List Implementation

```
typedef struct tnode *ptnode;  
typedef struct tnode  
{  
    int data;  
    ptnode left, right;  
};
```

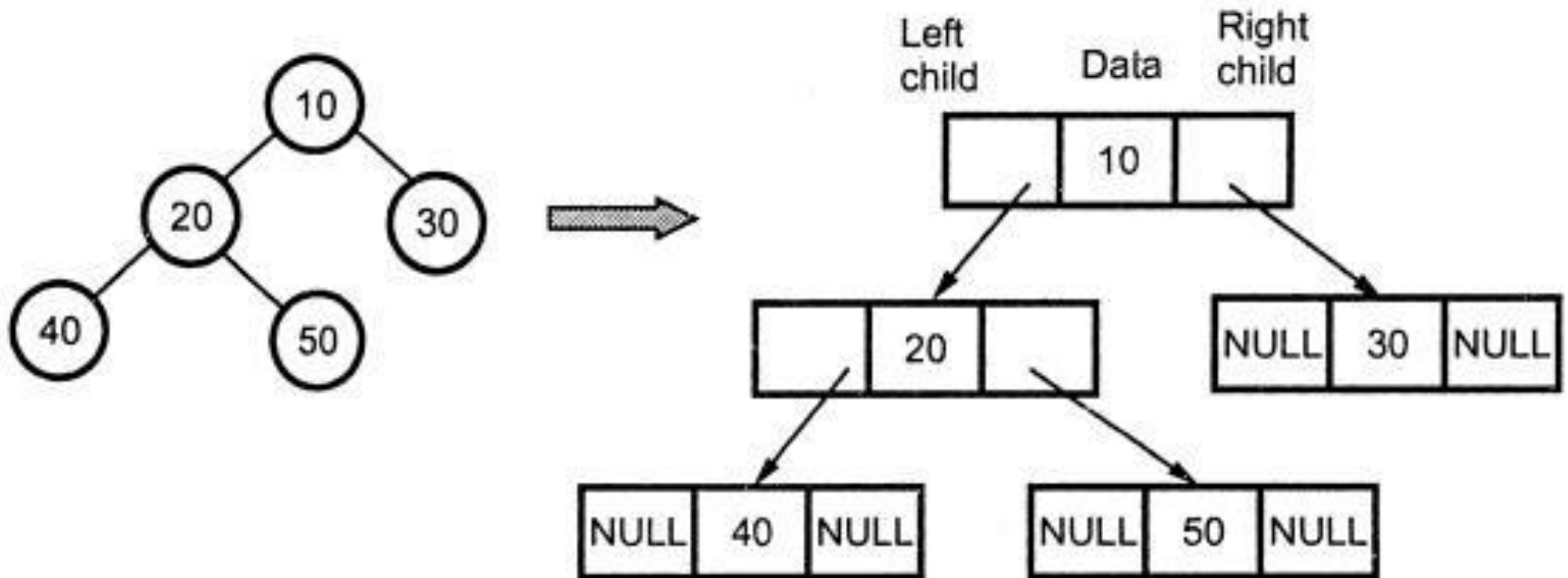




# Full and Complete Binary Trees

- A binary tree is said to be full if all its leaves are at the same level and every interior node has two children.
- A complete binary tree is either a full binary tree or one that is full except for a segment of missing leaves on the right side of the bottom level.

# Linked List Implementation



# Include One more Pointer

- A natural way to implement a *tree*  $T$  is to use a *linked structure*, where we represent each *node*  $p$  of  $T$  by a *position* object with the following fields:
  - A link to the *parent* of  $p$ , A link to the *LeftChild* named *Left*, a link to the *RightChild* named *Right* and the *Data*.

Parent  
Data  
Left      Right

# Array Implementation

- Advantages
  - Direct Access
  - Finding the Parent / Children is fast
- Disadvantages
  - Wastage of memory
  - Insertion and Deletion will be costlier
  - Array size and depth

# Linked List Implementation

- Advantages
  - No wastage of memory
  - Insertion and Deletion will be easy
- Disadvantages
  - Does not provide direct access
  - Additional space in each node.

# Binary Tree ADT

- The Binary Tree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT, in addition to that it supports the following additional accessor methods:
  - position left(p): return the left child of p, an error condition occurs if p has no left child.
  - position right(p): return the right child of p, an error condition occurs if p has no right child.
  - boolean hasLeft(p): test whether p has a left child
  - boolean hasRight(p): test whether p has a right child

# Binary Tree ADT (Cont.)

- Update methods may be defined by data structures implementing the Binary Tree ADT.
- Since Binary trees are ordered trees, the iterable collection returned by method `children(p)` (inherited from the Tree ADT), stores the left child of `p` before the right child of `p`.

# Binary Tree Traversals

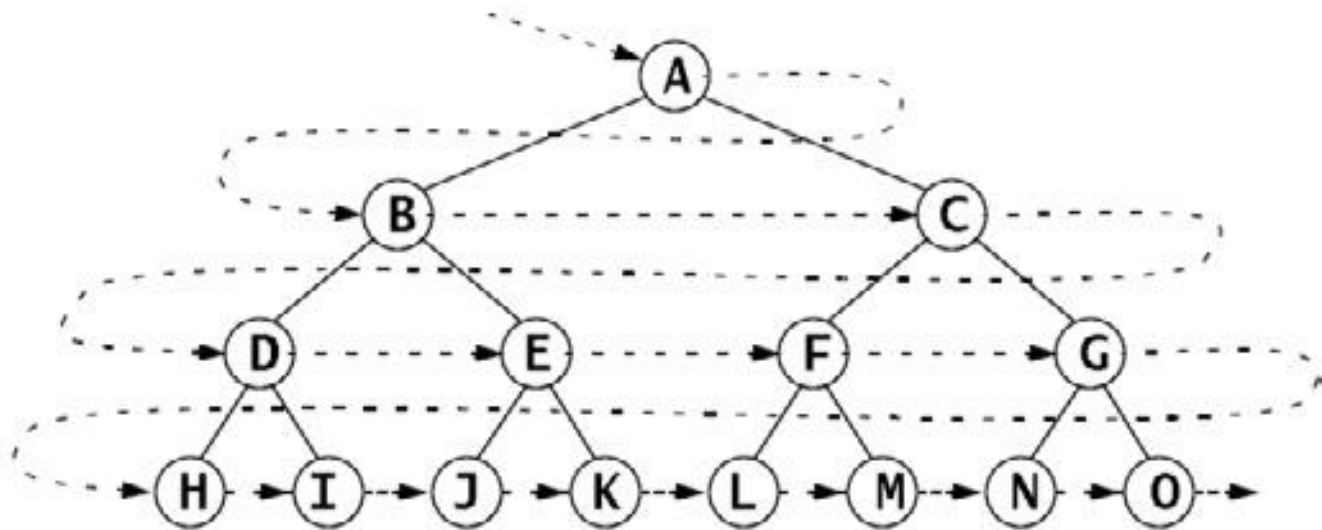
- The three traversal algorithms that are used for general trees (see Chapter 10) apply to binary trees as well: the preorder traversal, the postorder traversal, and the level order traversal.
- In addition, binary trees support a fourth traversal algorithm: the inorder traversal. These four traversal algorithms are given next.



# Level Order Traversal

- To traverse a nonempty binary tree:
  1. Initialize a queue.
  2. Enqueue the root.
  3. Repeat steps 4–7 until the queue is empty.
    4. Dequeue a node  $x$  from the queue.
    5. Visit  $x$ .
    6. Enqueue the left child of  $x$  if it exists.
    7. Enqueue the right child of  $x$  if it exists.

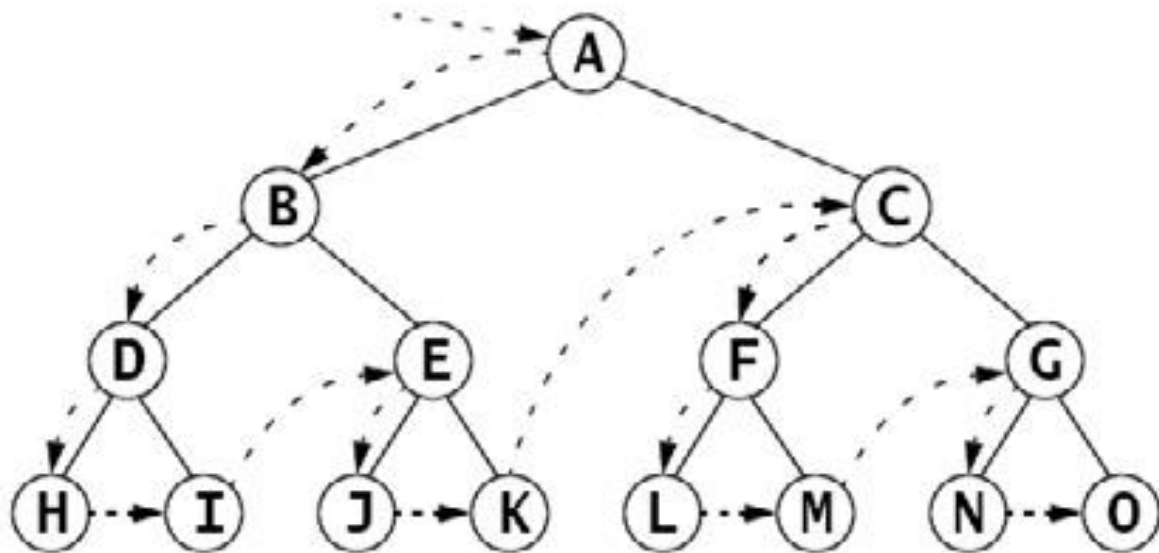
# Level Order



# Preorder Traversal

- To traverse a nonempty binary tree:
  1. Visit the root.
  2. If the left subtree is nonempty, do a preorder traversal on it.
  3. If the right subtree is nonempty, do a preorder traversal on it.

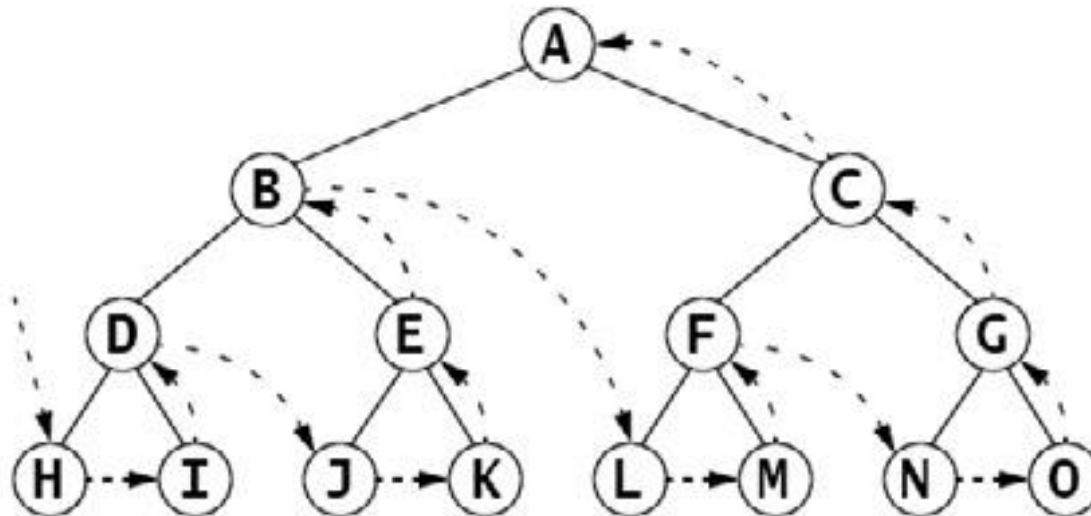
# Preorder



# Postorder Traversal

- To traverse a nonempty binary tree:
  1. If the left subtree is nonempty, do a postorder traversal on it.
  2. If the right subtree is nonempty, do a postorder traversal on it.
  3. Visit the root.

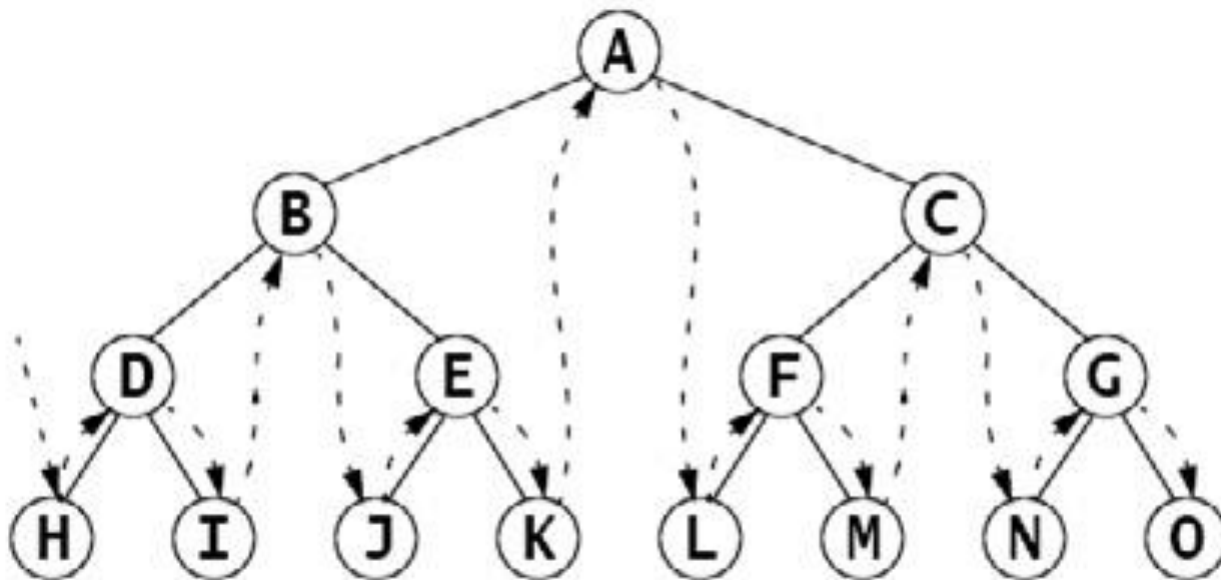
# Postorder



# Inorder Traversal

- To traverse a nonempty binary tree:
  1. If the left subtree is nonempty, do a preorder traversal on it.
  2. Visit the root.
  3. If the right subtree is nonempty, do a preorder traversal on it.

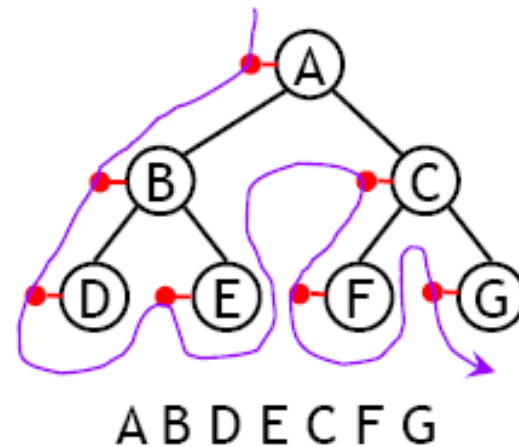
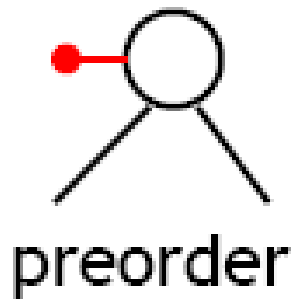
# Inorder



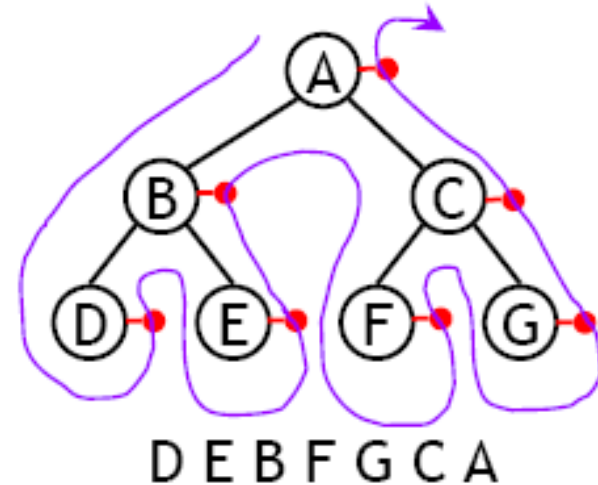
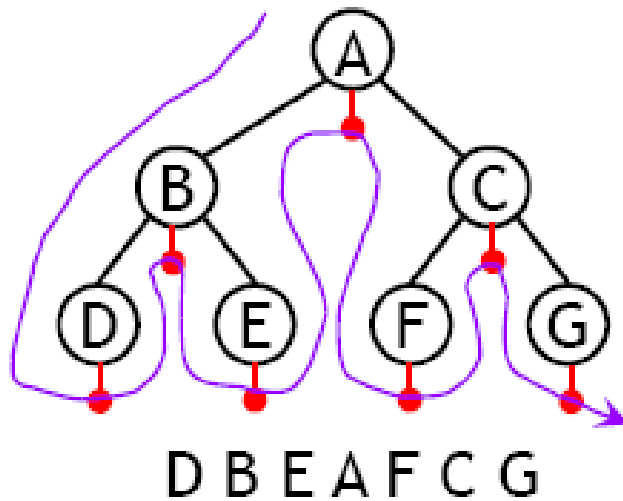
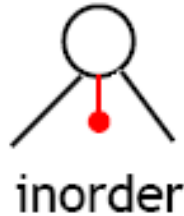


# Traversal Using Flag

- The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a “flag” attached to each node, as follows:
- To traverse the tree, collect the flags:



# Inorder and Postorder



# Interaction

