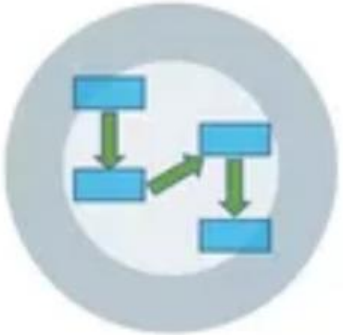
Abstract Data Types

Data Structures and Algorithms



Arfan Shahzad

{ arfanskp@gmail.com }

Data Structures and Algorithms

BS(IT)-03 Session 2021-25			
Sr. No.	**C. Code**	**Course Title**	**Cr. Hr.**
1	CSI-401	Data Structure and Algorithms	4(3-1)
2	ELE-401	Digital Logic Design	3(2-1)
3	ENG-422	Technical Writing	3(3-0)
4	CSI-408	Database Systems	4(3-1)
5	CSI-407	Computer Networks	4(3-1)
6	ISL-411	Translation of Holly Quran-II	1(1-0)
		Total	**18**

Data Structures and Algorithms

Data Structures and Algorithms
Course Contents:
Abstract data types, complexity analysis, Big Oh notation, Stacks (linked lists and array implementations), Recursion and analyzing recursive algorithms, divide and conquer algorithms, Sorting algorithms (selection, insertion, merge, quick, bubble, heap, shell, radix, bucket), queue, dequeuer, priority queues (linked and array implementations of queues), linked list & its various types, sorted linked list, searching an unsorted array, binary search for sorted arrays, hashing and indexing, open addressing and chaining, trees and tree traversals, binary search trees, heaps, M-way tress, balanced trees, graphs, breadth-first and depth-first traversal, topological order, shortest path, adjacency matrix and adjacency list implementations, memory management and garbage collection

Data Structures and Algorithms

Reference Material:
1. Data Structures and Algorithms in C++ by Adam Drozdek 2. Data Structures and Algorithm Analysis in Java by Mark A. Weiss 3. Data Structures and Abstractions with Java by Frank M. Carrano & Timothy M. Henry 4. Data Structures and Algorithm Analysis in C++ by Mark Allen Weiss 5. Java Software Structures: Designing and Using Data Structures by John Lewis and Joseph Chase

Abstraction

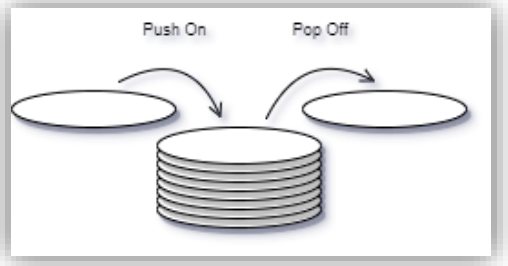
- The concept of ***abstraction*** is to ***extract a complicated system down to its most fundamental parts*** and ***describe these parts in a simple and precise way***.
- Applying the ***abstraction*** paradigm to the ***data structures*** gives rise to ***Abstract Data Types*** **(ADTs)**.

Abstract Data Types

- An ***ADT is a mathematical model of a data structure*** that specifies the ***type of data stored***, the ***operations supported on them***, and ***the types of parameters of the operations***.
- An **ADT specifies *what* each operation does**, but **not *how* it does it**.

Abstract Data Types cont...
Stack ADT

- It is named ***stack*** as it behaves like a real-world stack, for example, a pile of plates, etc.
- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be **LIFO (Last In First Out)** or **FILO (First In Last Out)**.
- Formally, a stack is an abstract data type (ADT) that supports the following two methods:



Abstract Data Types cont...
Stack ADT

- **push(*e*)**: Insert element *e*, to be the top of the stack.
- **pop()**: Remove from the stack and return the top element on the stack; an error occurs if the stack is empty.

- Additionally, let us also define the following methods:

Abstract Data Types cont...
Stack ADT

- **size():** Return the number of elements in the stack.
- **isEmpty():** Return a Boolean indicating if the stack is empty.
- **top():** Return the top element in the stack, without removing it; an error occurs if the stack is empty.

Abstract Data Types cont...
Queue ADT

- Formally, the queue abstract data type defines a collection that ***keeps objects in a sequence***, where element **access and deletion** are ***restricted to the first element in the sequence***, which is called the ***front*** of the queue, and **element insertion** is ***restricted to the end of the sequence***, which is called the ***rear*** of the queue.
- This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in first-out (**FIFO**) principle.

Abstract Data Types cont...
Queue ADT

- The ***queue*** abstract data type (ADT) supports the following two fundamental methods:
- **enqueue(*e*):** Insert element *e* at the rear of the queue.
- **dequeue():** Remove and return from the queue the object at the front; an error occurs if the queue is empty.

Abstract Data Types cont...
Queue ADT

- Additionally, the queue ADT includes the following supporting methods:
- **size():** Return the number of objects in the queue.
- **isEmpty():** Return a Boolean value that indicates whether the queue is empty.
- **front():** Return, but do not remove, the front object in the queue; an error occurs if the queue is empty.

Abstract Data Types cont...
Deque ADT

- A **double-ended queue** (**deque**) is an ADT that generalizes a queue, for which ***elements can be added to or removed from either*** **the front (head)** or **back (tail)**.
- **Note:** As a verb **dequeue** is to remove an item from a queue.
- The fundamental methods of the **deque** ADT are as follows:

Abstract Data Types cont...
Deque ADT

- **addFirst():** Insert a new element at the beginning of the deque.
- **addLast():** Insert a new element at the end.
- **removeFirst():** Remove and return the first element of the deque.
- **removeLast():** Remove and return the last element of the deque.

Abstract Data Types cont...
Deque ADT

- Additionally, the deque ADT also include the following methods:
- **getFirst():** Return the first element of deque; an error occurs if deque is empty.
- **getLast():** Return the last element of deque; an error occurs if deque is empty.
- **size():** Return the number of elements of the deque.
- **isEmpty():** Determine if the deque is empty.

Abstract Data Types cont...
Array list ADT

- An **array** is a collection of same data items stored contiguously.
- As an ADT, an ***array S*** has the following methods (besides the standard **size()** and **isEmpty()** methods):

Abstract Data Types cont...
Array list ADT

- As an ADT, an ***array list S*** has the following methods (besides the standard **size()** and **isEmpty()** methods):
- **get(*i*):** Return the element of *S* with index *i*; an error condition occurs if *i* < 0 or *i* > size() - 1.
- **set(*i*, *e*):** Replace with *e* and return the element at index *i*; an error condition occurs if *i* < 0 or *i* > size() - 1.
- **add(*i*, *e*):** Insert a new element *e* into *S* to have index *i*; an error condition occurs if *i* < 0 or *i* > size().
- **remove(*i*):** Remove from *S* the element at index *i*; an error condition occurs if *i* < 0 or *i* > size() - 1.

Abstract Data Types cont...
Array list ADT

- **add(*i*, *e*):** Insert a new element *e* into *S* at index *i*; error occurs if *i* < 0 or *i* > size().
- **get(*i*):** Return the element of *S* with index *i*; error occurs if *i* < 0 or *i* > size() - 1.
- **set(*i*, *e*):** Replace with *e* and return element at index *i*; error occurs if *i* < 0 or *i* > size() - 1.
- **remove(*i*):** Remove from *S* the element at index *i*; error occurs if *i* < 0 or *i* > size() - 1.

Abstract Data Types cont...
Tree ADT

- A ***tree*** is an abstract data type that stores elements hierarchically.
- With the exception of the top element, each element in a tree has a ***parent*** element and zero or more ***children*** elements.
- A tree is usually visualized by placing elements inside ovals or rectangles, and by drawing the connections between parents and children with straight lines.

Abstract Data Types cont…
Tree ADT

- We typically call the top element the ***root*** of the tree, but it is drawn as the highest element, with the other elements being connected below (just the opposite of a botanical tree).

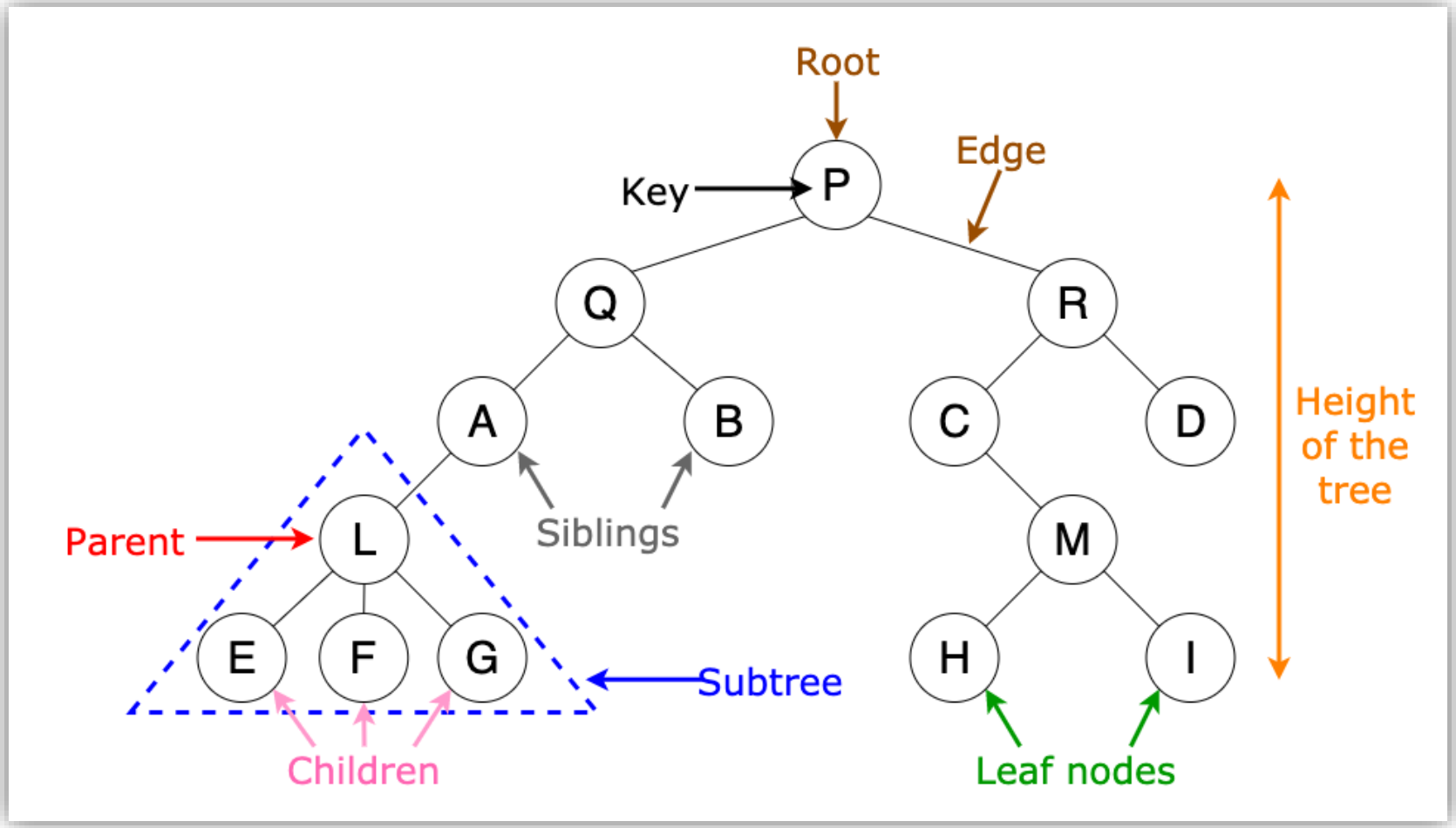
Abstract Data Types cont...
Tree ADT

- A ***tree*** is an abstract data type that stores elements **hierarchically**.
- With the exception of the top element, **each element in a tree has a *parent* element** and **zero or more *children* elements**.
- We typically call **the top element** the ***root*** of the tree, but it is drawn as the **highest element**, with the other elements being connected below (just the opposite of a botanical tree).

Abstract Data Types cont...
Tree ADT

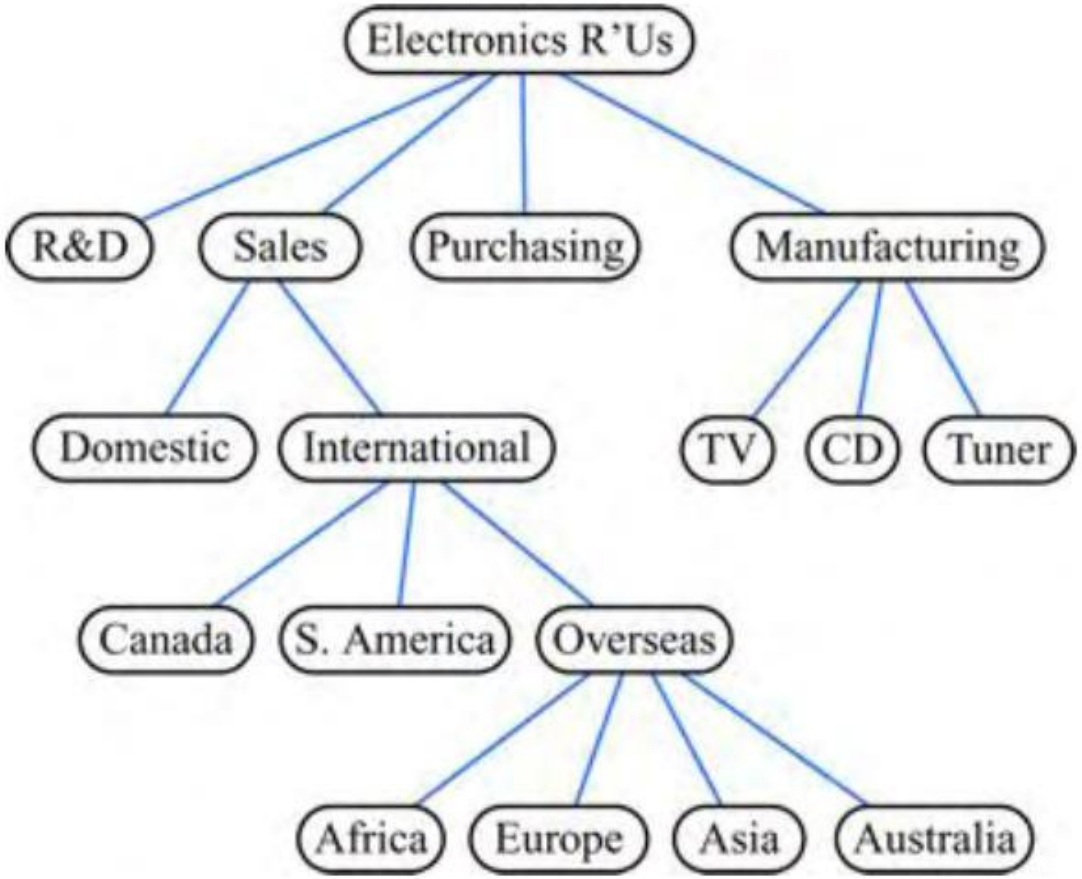
- A tree is usually visualized by **placing** ***elements*** (called ***nodes***) inside ***ovals*** or ***rectangles***.
- Elements (nodes) ***have connections between parents and children*** with ***straight lines*** **(called** ***edges/ links*****)**.

Abstract Data Types cont...
Tree ADT



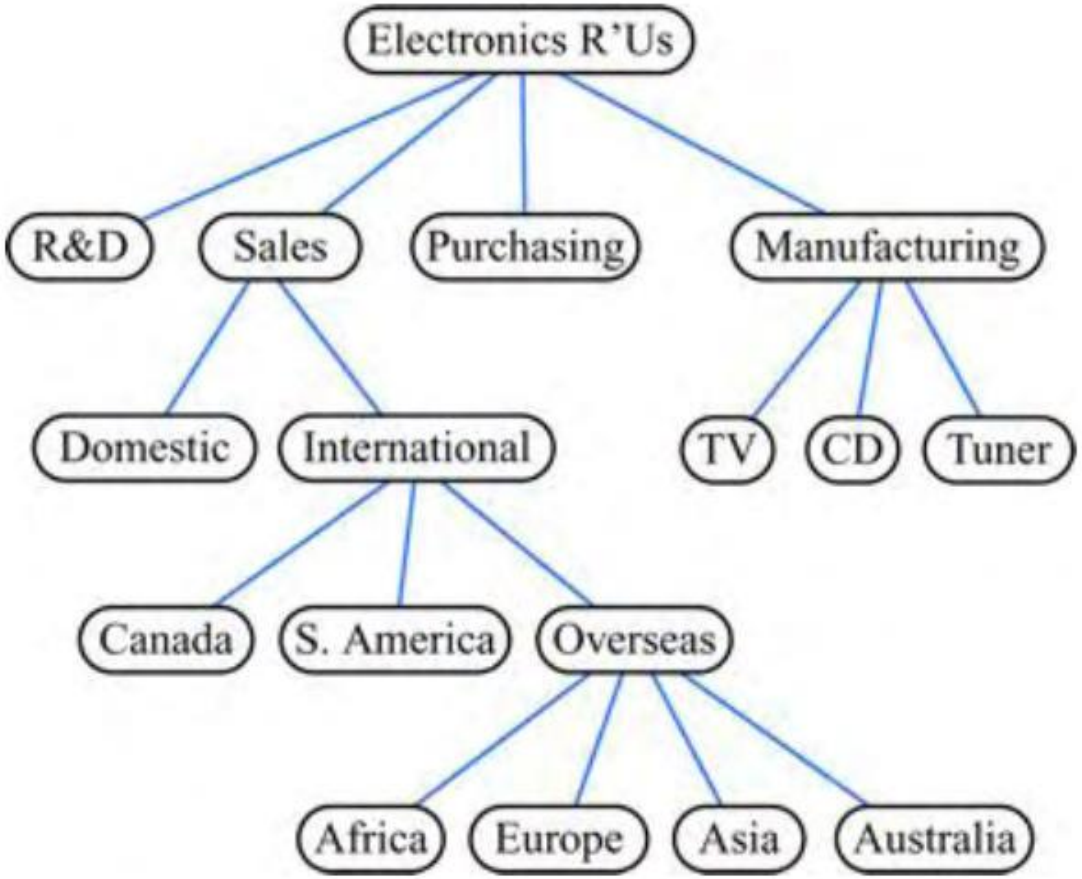
Abstract Data Types cont...
Tree ADT

- A tree with ***17 nodes*** representing the organization of a ***fictitious corporation***.
- The ***root*** stores ***Electronics R'Us***.



Abstract Data Types cont...
Tree ADT

- The ***children*** of the root store ***R&D***, ***Sales***, ***Purchasing***, and ***Manufacturing***.
- The ***internal nodes*** store ***Sales***, ***International***, ***Overseas***, ***Electronics R'Us***, and ***Manufacturing***.



Abstract Data Types cont...
Tree ADT (Formal Tree Def.)

- Formally, we define a ***tree T*** as a **set of *nodes*** storing elements such that the nodes have a ***parent-child*** relationship, that satisfies the following properties:

1. If ***T* is nonempty**, it has a **special node**, called the ***root*** of *T*, that **has no parent**.

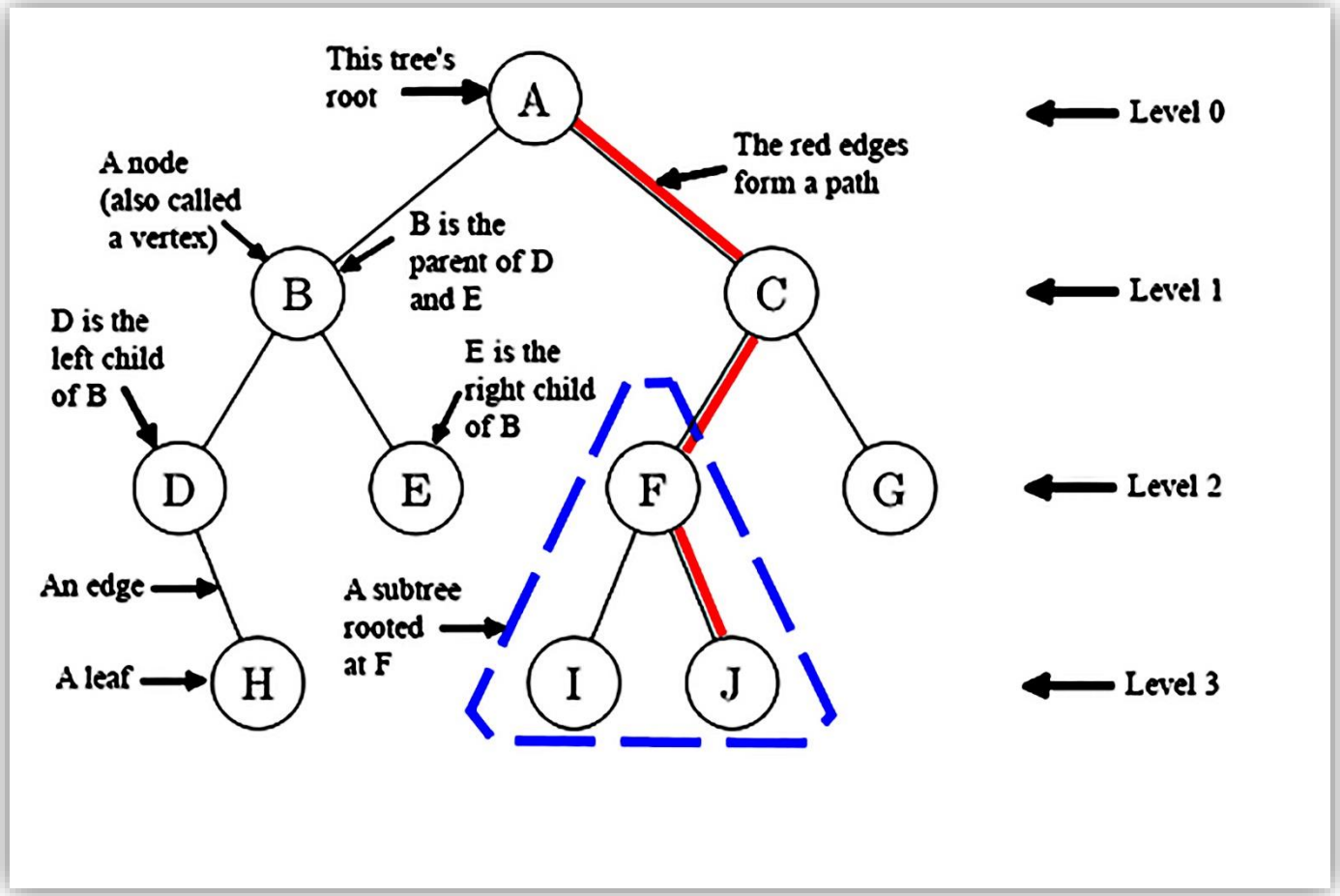
Abstract Data Types cont...
Tree ADT (Formal Tree Def.)

2. Each node **except the root node** has ***one edge upward to*** *another* ***node*** called ***parent***.
3. The ***node below a given node connected by its edge downward*** is called its ***child*** node.
4. Note that according to our definition, ***a tree can be empty***, meaning that **it doesn't have any nodes**.

Abstract Data Types cont...
Tree ADT (Formal Tree Def.)

5. Two nodes that are ***children of the same parent*** are ***siblings***.
6. A node *v* is ***external*** if ***v*** **has no children**. External nodes are also known as ***leaf nodes***.
7. A node *v* is ***internal*** if it **has one or more children**.
8. The ***sequence of nodes along the edges of a tree*** is called ***path***. There is a **path from every node to the root**. This path is found by simply following successive parent links to the root.

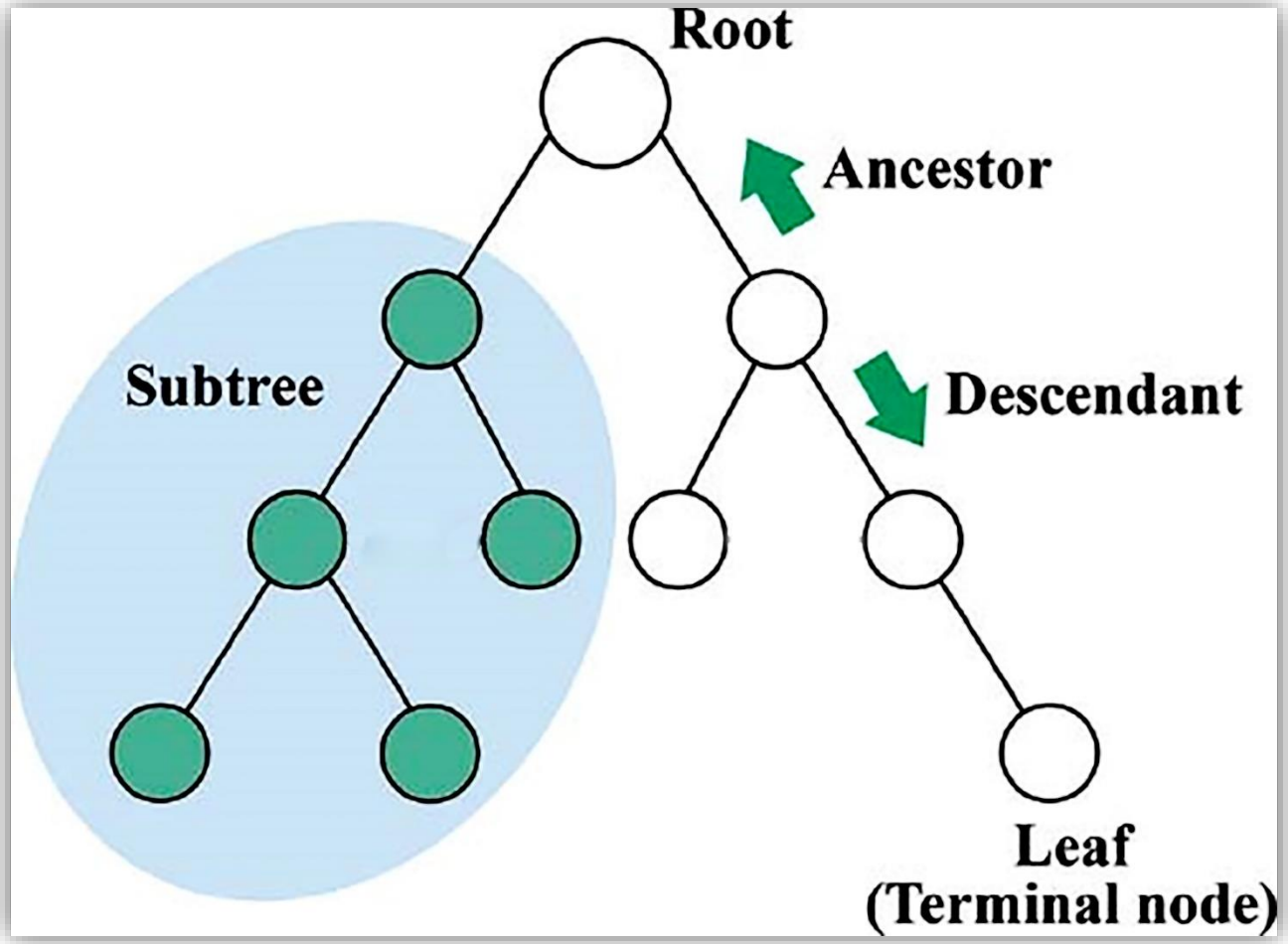
Abstract Data Types cont...
Tree ADT (Formal Tree Def.)



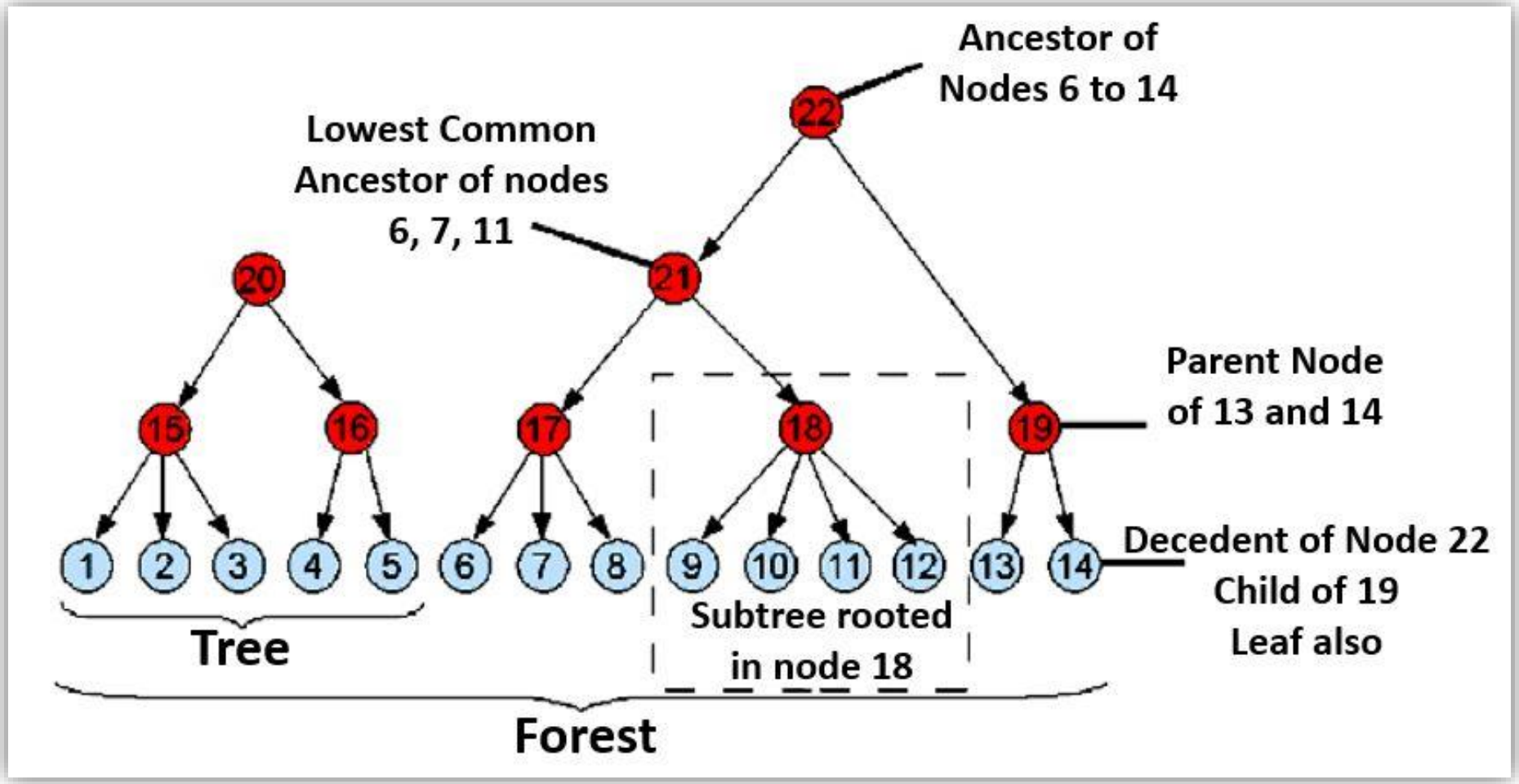
Abstract Data Types cont...
Tree ADT (Formal Tree Def.)

9. Besides ***parent-child relationship***, tree nodes have ***ancestors*** & ***descendants***.
10. An ***ancestor*** of a node is any ***other node on the path*** from the ***node to the root (upwards)***.
11. Conversely, we say that a node ***v is a descendent of a node u*** **if *u* is an ancestor of *v***.
12. The ***descendants*** of a node called ***subtree***.

Abstract Data Types cont...
Tree ADT (Formal Tree Def.)



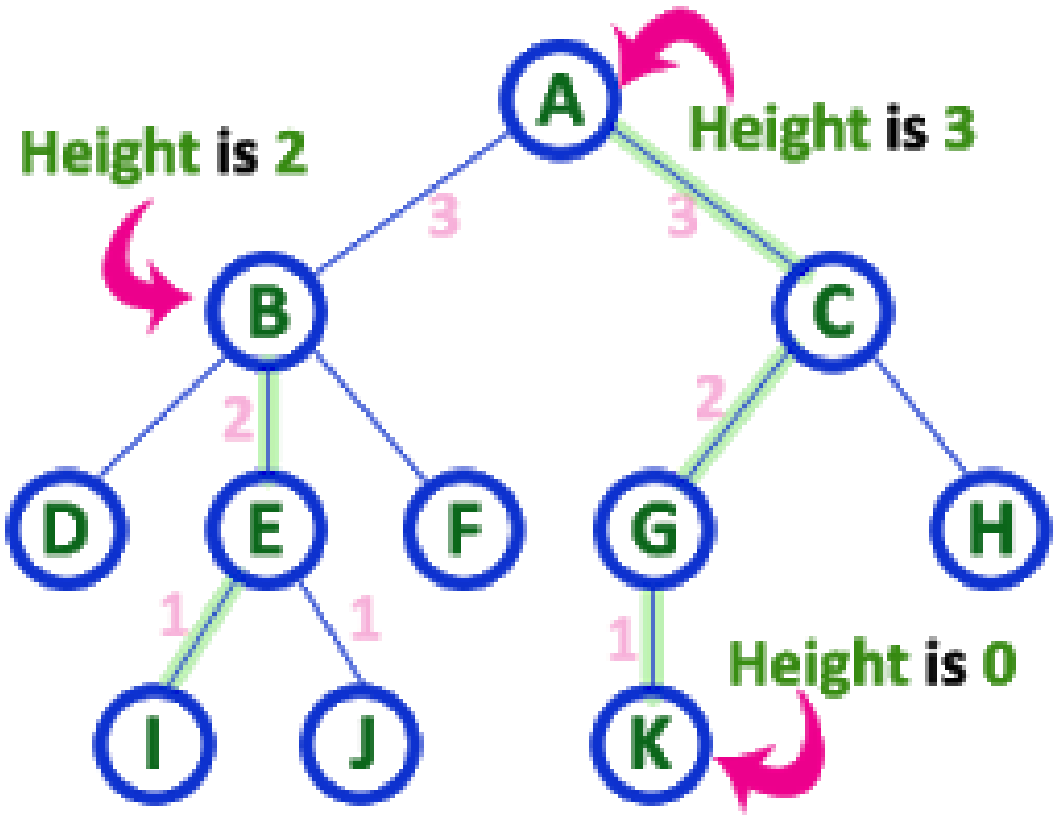
Abstract Data Types cont...
Tree ADT (Formal Tree Def.)



Abstract Data Types cont...
Tree ADT (Formal Tree Def.)

13. The ***height*** **of a node N** is the ***length of the LONGEST path from N to a leaf node***. We usually talk about the ***height of a tree***, which is the ***height of the root node***, the ***longest path from the root to a leaf***. All leaf nodes have height 0.

Abstract Data Types cont...
Tree ADT (Formal Tree Def.)



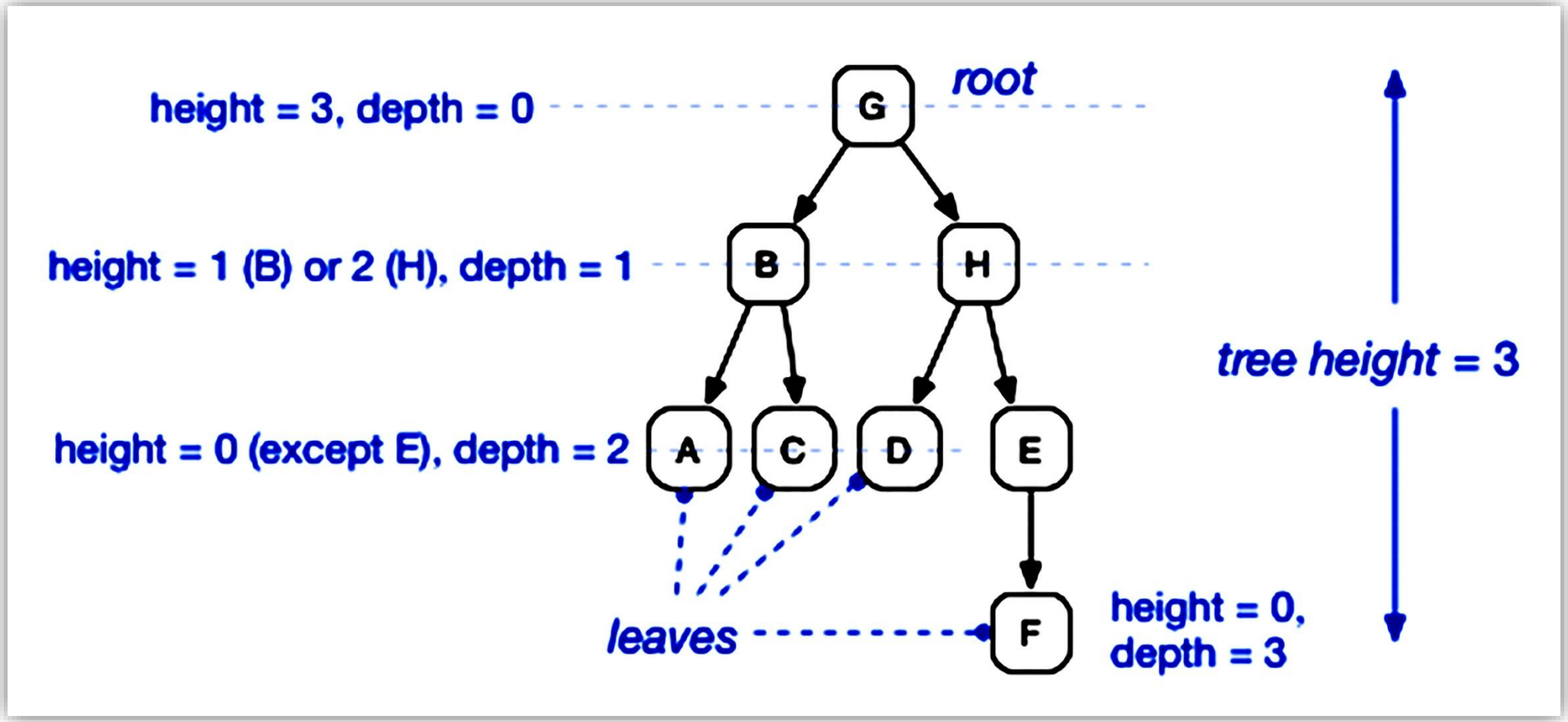
Here Height of tree is 3

- In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.
- In any tree, 'Height of Tree' is the height of the root node.

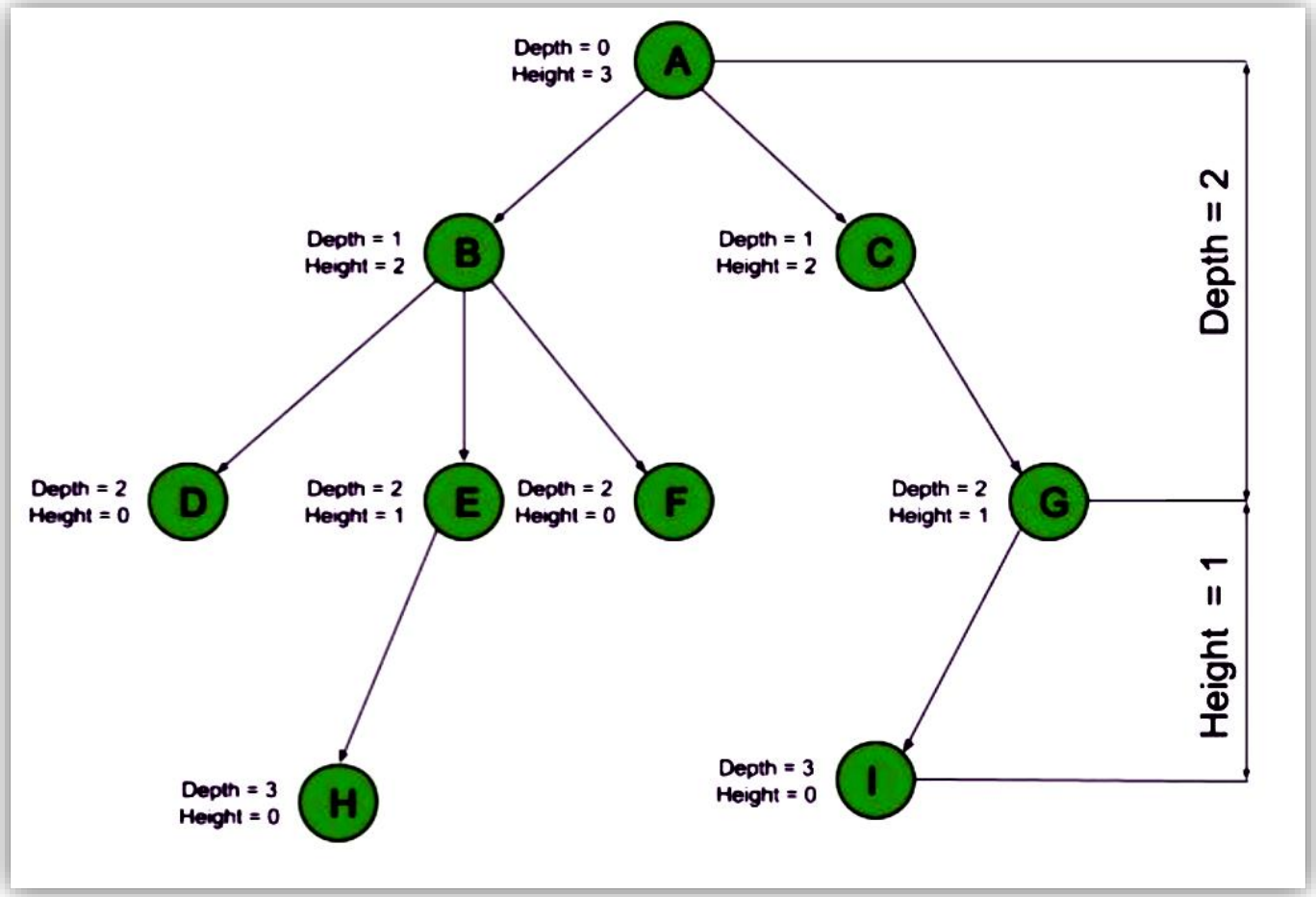
Abstract Data Types cont...
Tree ADT (Formal Tree Def.)

14. We also use the terms ***depth***/ ***level*** of a node. The ***depth or level of a node N is the length of the path from the root to N***. The depth or level of the root is 0.

Abstract Data Types cont...
Tree ADT (Formal Tree Def.)



Abstract Data Types cont...
Tree ADT (Formal Tree Def.)



Abstract Data Types cont...
Tree ADT (Formal Tree Def.)

15. ***Key*** represents a **value** of a node ***based on which a search operation is to be carried out for a node***.

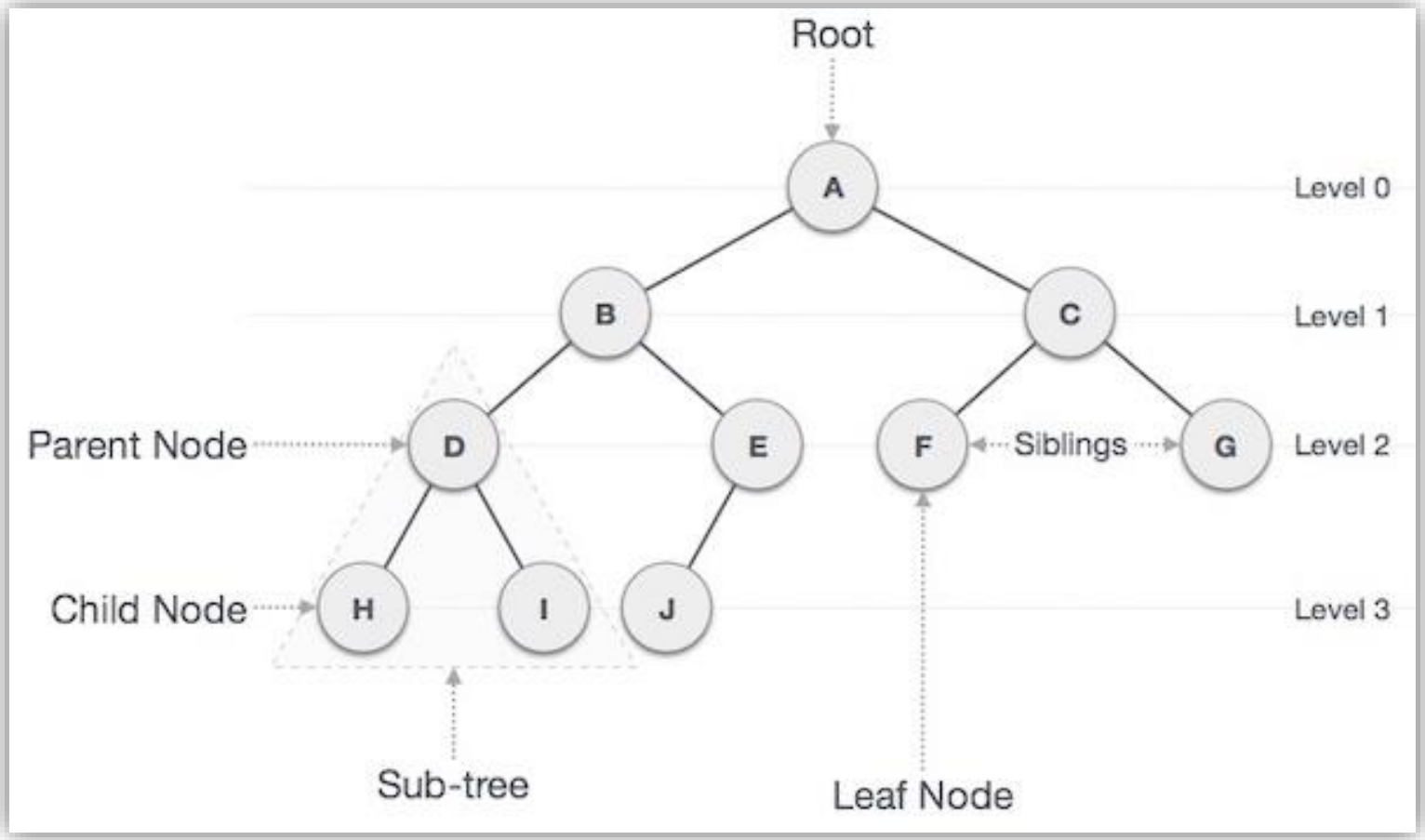
16. ***Passing through nodes in a specific order*** is called ***traversing***.

17. Checking the value of a node when control is on the node is called ***visiting***.

Abstract Data Types cont…
Tree ADT (Ordered/ Binary Tree)

- An ***ordered tree*** contains nodes (elements) which can be ordered according to a specific criteria. Often it is a ***binary tree.***
- A ***binary tree has a special condition*** that ***each node can have a maximum of two children*** (conveniently called the left and right child)..

Abstract Data Types cont...
Tree ADT (Ordered/ Binary Tree)



Abstract Data Types cont...
Tree ADT (Ordered/ Binary Tree)

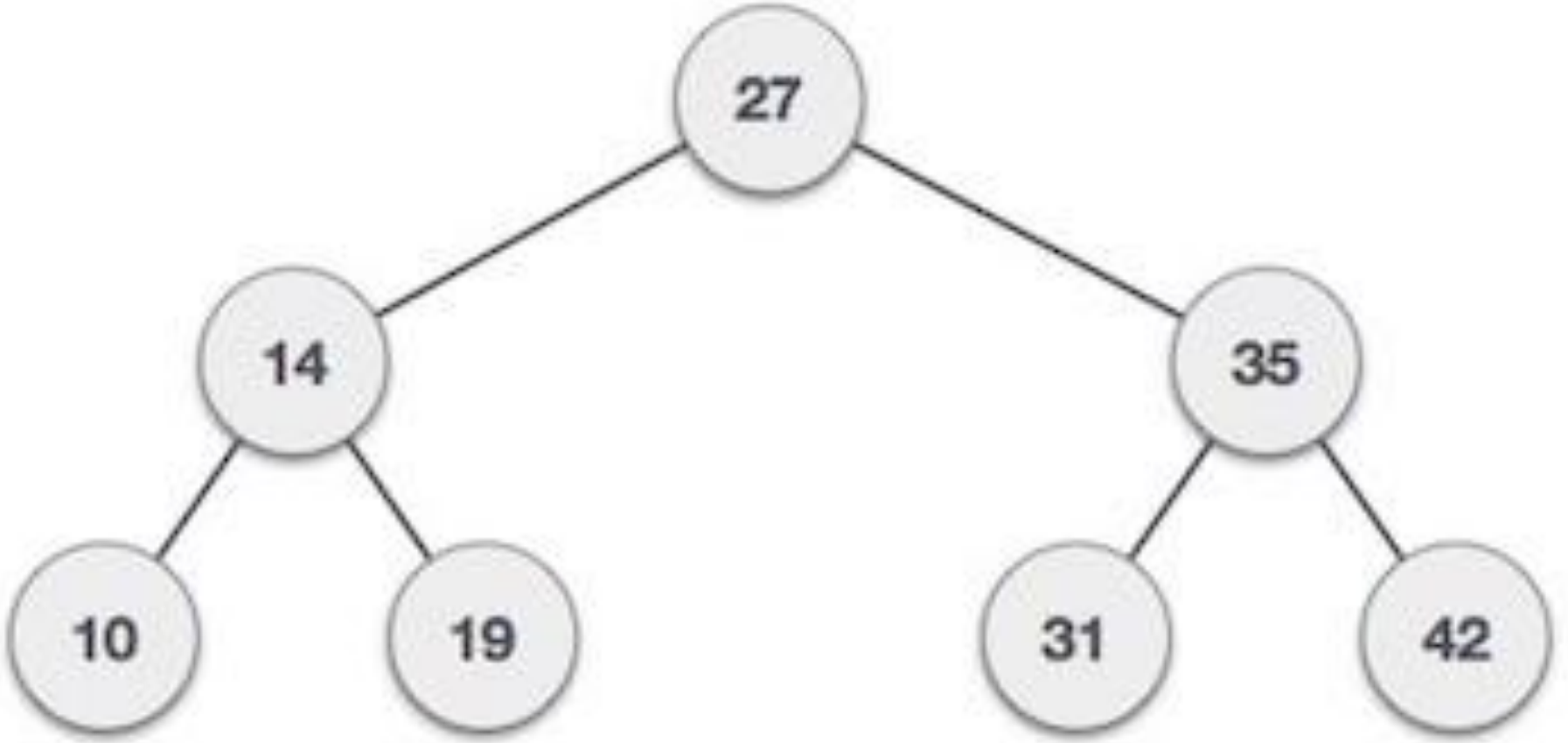
- A binary tree has the benefits of both an ***ordered array*** and a *linked list*.
- **The *search of binary tree is as quick as in a sorted array*.**
- And the ***insertion or deletion operation in binary tree are as fast as in linked list***.

Abstract Data Types cont...
Tree ADT (Ordered/ Binary Tree)

- A ***binary tree*** sometimes called **Binary Search Tree (BST)** is a tree in which all the nodes follow the below-mentioned properties:

1. The ***left sub-tree of a node*** has a ***key less than or equal to its parent node's key***.
2. The ***right sub-tree of a node*** has a ***key greater than or equal to its parent node's key***.

Abstract Data Types cont...
Tree ADT (Ordered/ Binary Tree)



Abstract Data Types cont…
Tree ADT (Ordered/ Binary Tree)

- Thus, Binary Search Tree divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as:

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

Abstract Data Types cont…
Tree ADT (Basic Functions)

- The ***tree ADT stores elements at positions***, which, as with positions in a list, are ***defined relative to neighboring positions (left right)***.
- The ***positions*** in a tree are its ***nodes***, and ***neighboring positions satisfy the parent-child relationships that define a valid tree (just we have discussed in a binary tree)***.

Abstract Data Types cont...
Tree ADT (Basic Functions)

- Therefore, we use the terms "**position**" and "**node**" interchangeably for trees.
- As with a list position, a position object for a tree supports the method:
- **element():** return the object stored at this position.

Abstract Data Types cont...
Tree ADT (Basic Functions)

- The real power of node positions in a tree, however, comes from the ***accessor methods*** of the tree ADT that ***return and accept positions***, such as the following:
- **root():** return the tree's root; an error occurs if the tree is empty.
- **parent (*v*):** return the parent of *v*; an error occurs if *v* is the root.

Abstract Data Types cont…
Tree ADT (Basic Functions)

- **children(*v*):** return an iterable collection containing the children of node *v*.
- If a tree *T* is ordered, then the iterable collection, **children(*v*)**, ***stores the children of v in order***.
- If ***v* is an external node**, then **children(*v*)** is ***empty***.

Abstract Data Types cont…
Tree ADT (Basic Functions)

- As an abstract data type, a binary tree is a specialization of a tree that supports ***four additional accessor methods***:
- **left(*v*):** Return the left child of *v*; an error condition occurs if *v* has no left child.
- **right(*v*):** Return right child of *v*; an error condition occurs if *v* has no right child.
- **hasLeft(*v*):** Test whether *v* has a left child.
- **hasRight(*v*):** Test whether *v* has a right child.

Abstract Data Types cont…
Tree ADT (Basic Functions)

- In addition to the above fundamental accessor methods, we also include the following ***query methods***:
- **isInternal(*v*):** Test whether node *v* is internal.
- **isExternal(*v*):** Test whether node *v* is external.
- **isRoot(*v*):** Test whether node *v* is the root.

Abstract Data Types cont…
Tree ADT (Basic Functions)

- There are also a number of ***generic methods*** a tree should probably support that are not necessarily related to its tree structure, including the following:
- **size():** return the number of nodes in the tree.
- **isEmpty():** Test whether the tree has any nodes or not.