Trees-part1

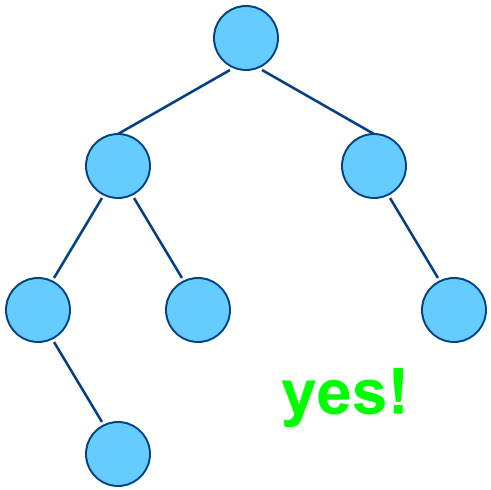
Objectives

- Understand tree terminology
- Understand and implement tree traversals
- Define the binary search tree property
- Implement binary search trees

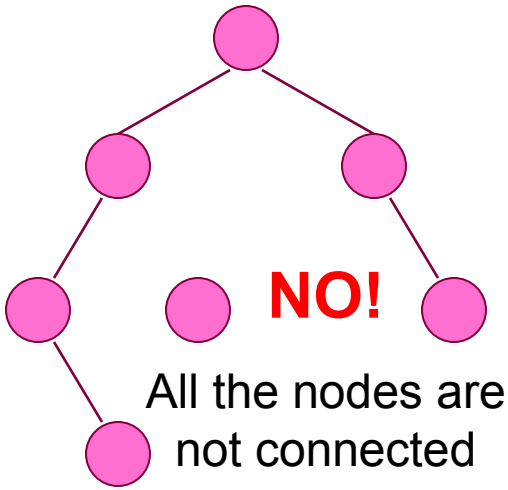
Trees

- A set of nodes with a single starting point
  - called the **root**
- Each node is connected by an edge to some other node
- A tree is a connected **graph**
  - There is a path to every node in the tree
- There are no cycles in the tree.
  - It can be proved by MI that a tree has one less edge than the number of nodes.
- Usually depicted with the root at the top

Is it a Tree?

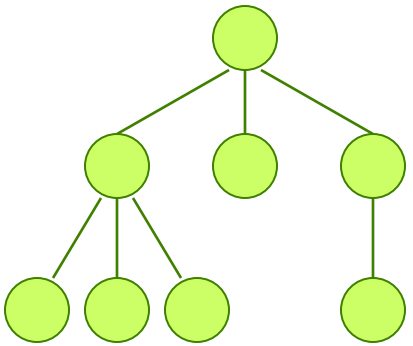


yes!

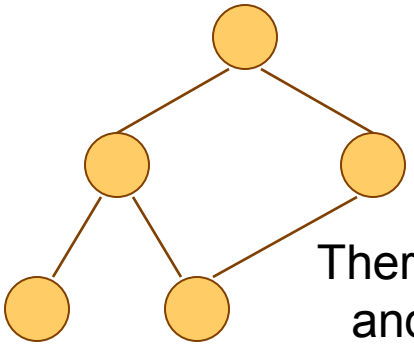


NO!

All the nodes are not connected

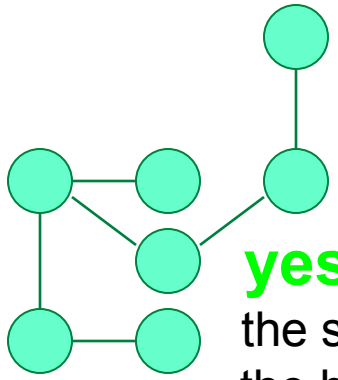


yes! (but not a binary tree)



NO!

There is a cycle and an extra edge (5 nodes and 5 edges)

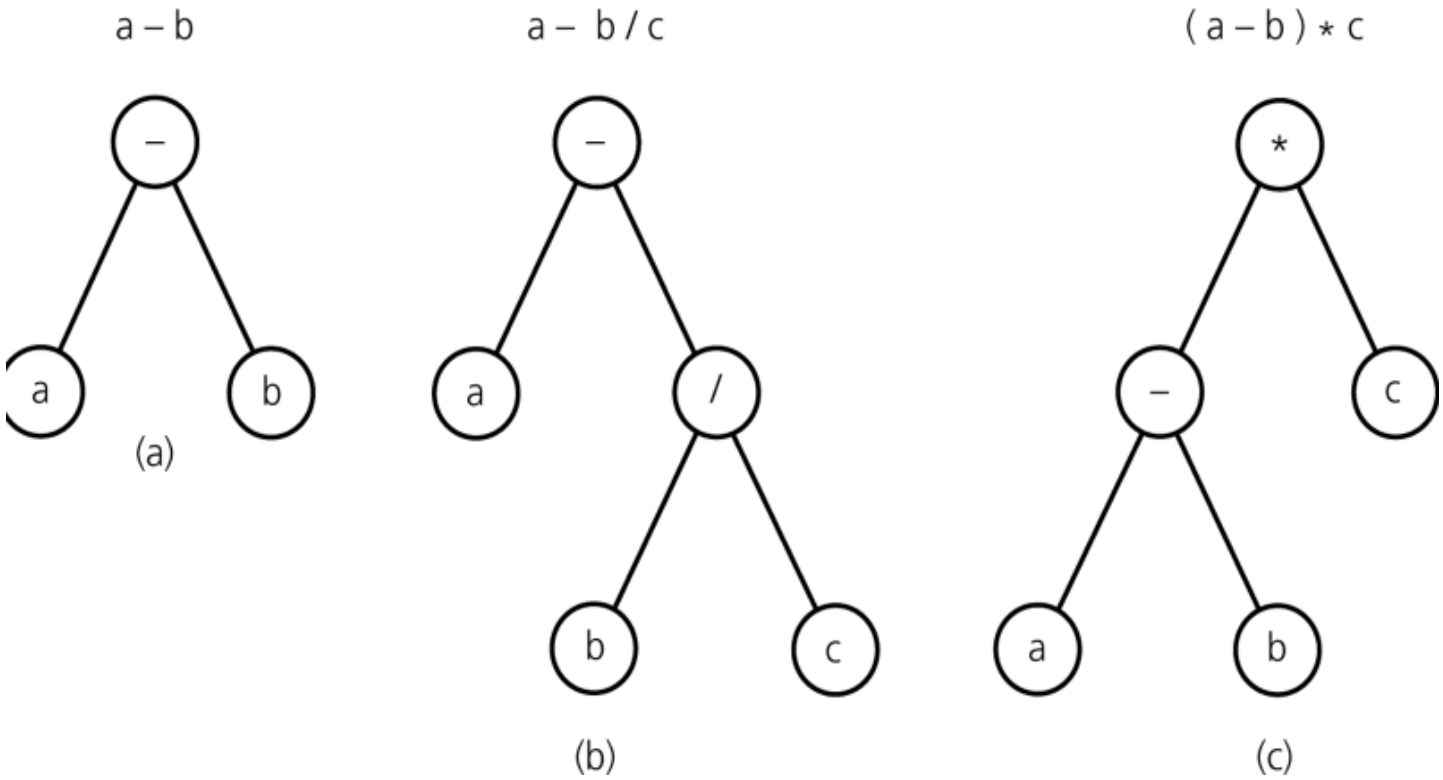


yes! (it's actually the same graph as the blue one) – but usually we draw tree by its "levels"

Examples of trees

- directory structure
- family trees:
  - all descendants of a particular person
  - all ancestors born after year 1800 of a particular person
- evolutionary tress (also called phylogenetic trees)
- algebraic expressions

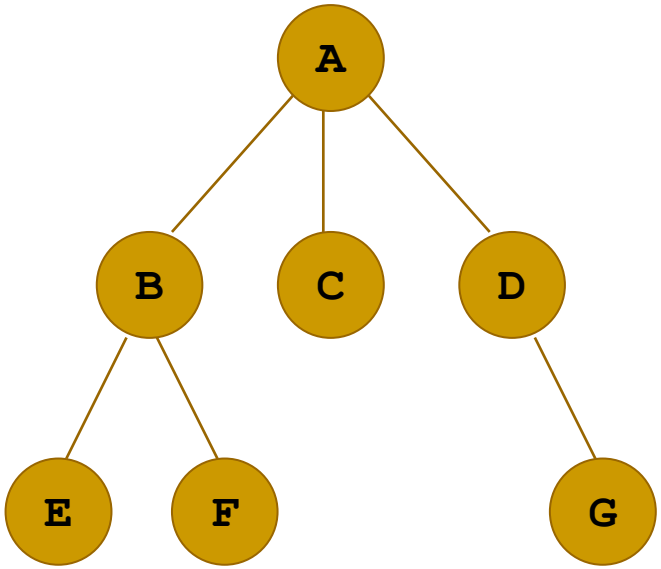
Examples of trees



Binary trees that represent algebraic expressions

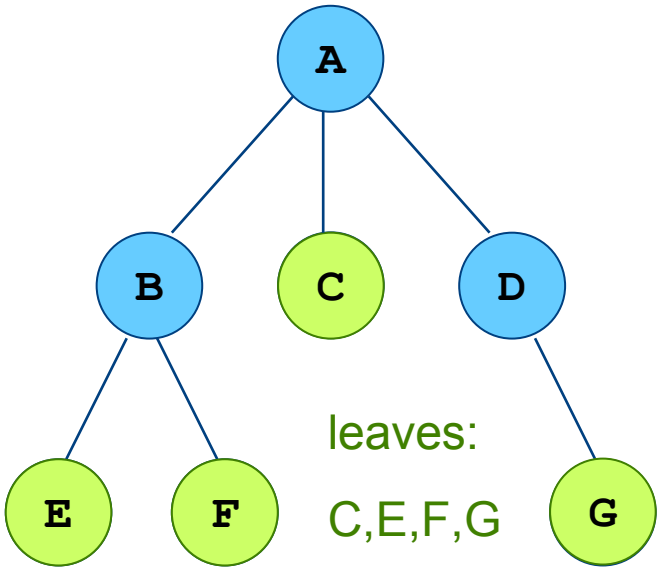
Tree Relationships

- If there is an edge between two nodes *u* and *v*, and *u* is "above" *v* in the tree (closer to the root), then *v* is said to be a **child** of *u,* and *u* the **parent** of *v*
  - A is the parent of B, C and D
- This relationship can be generalized (transitively)
  - E and F are **descendants** of A
  - D and A are **ancestors** of G
  - B, C and D are **siblings**

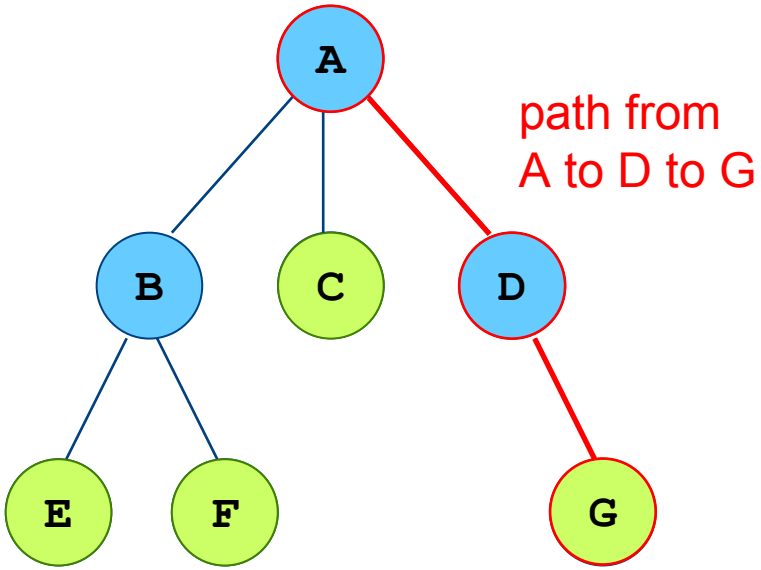


Tree Terminology Example

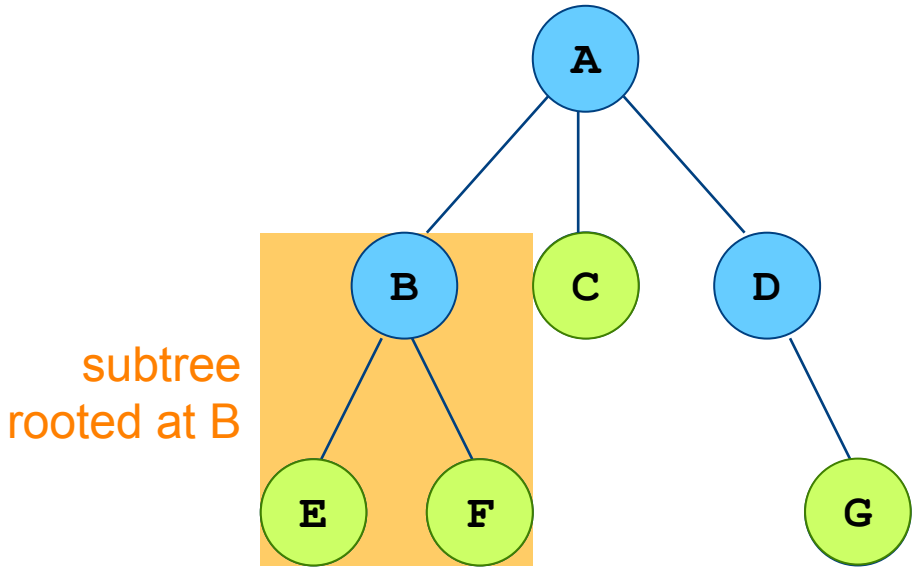
- ✓A **leaf** is a node with no children



A **path** [a **branch**] is a sequence of nodes $v_1 \ldots v_n$ where v_i is a parent of v_{i+1} ($1 \leq i \leq n\text{-}1$) [and v_1 is a root and v_n is a leaf]



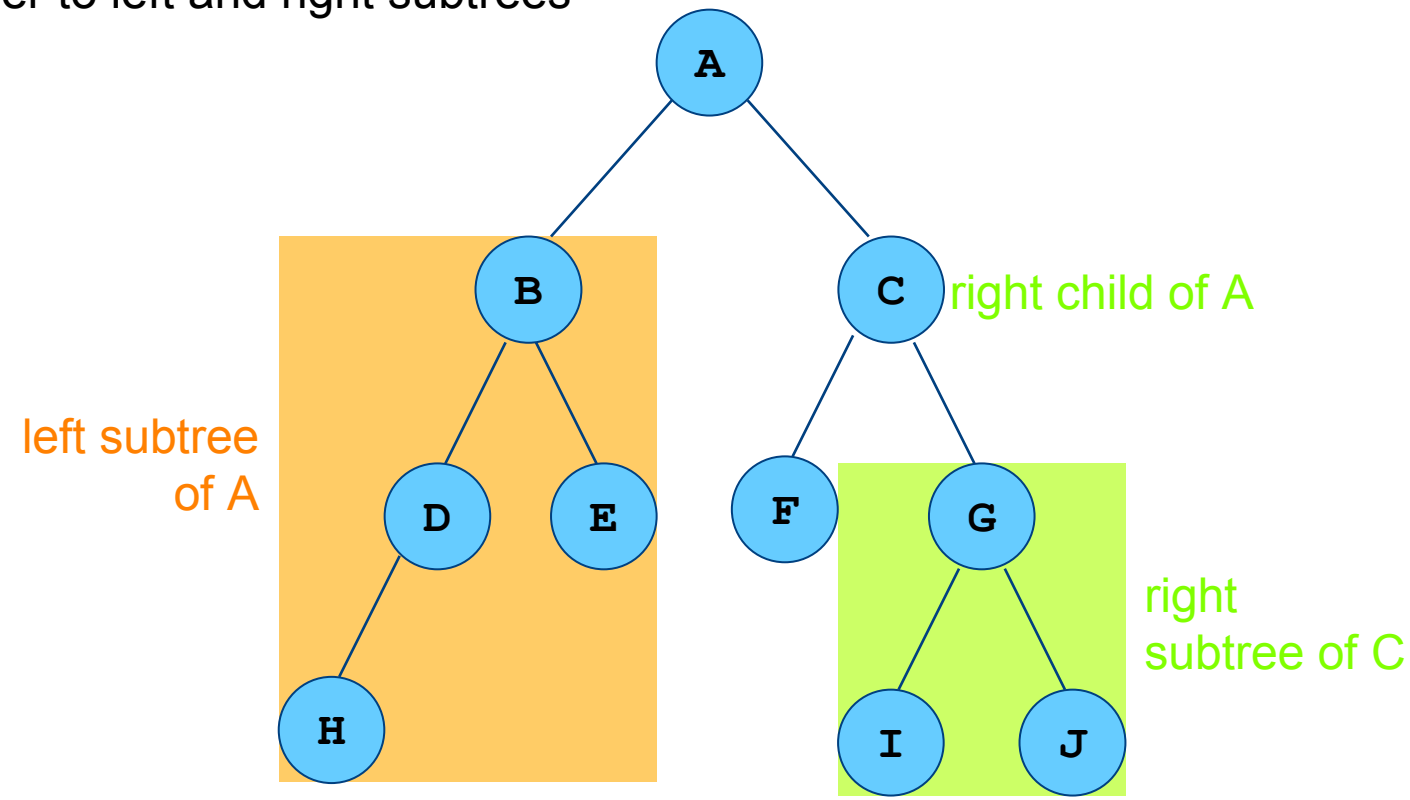
A **subtree** is any node in the tree along with all of its descendants



A **binary** tree is a tree with at most two children per node

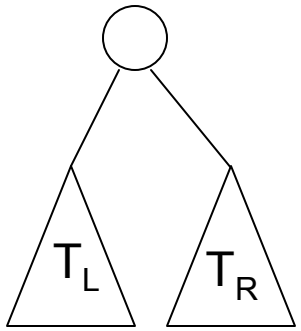
•The children are referred to as left and right (i.e., children are usually ordered)

•We can also refer to left and right subtrees of a node



Prove that the number of edges in a tree is one less than the number of nodes.

- Prove by induction on the number of nodes.
- Base of induction is true for a tree with one node.
- **Induction hypothesis**: the theorem is true for any tree with M < N number of nodes.
- **Induction step**: prove that the theorem is true for a tree with N nodes



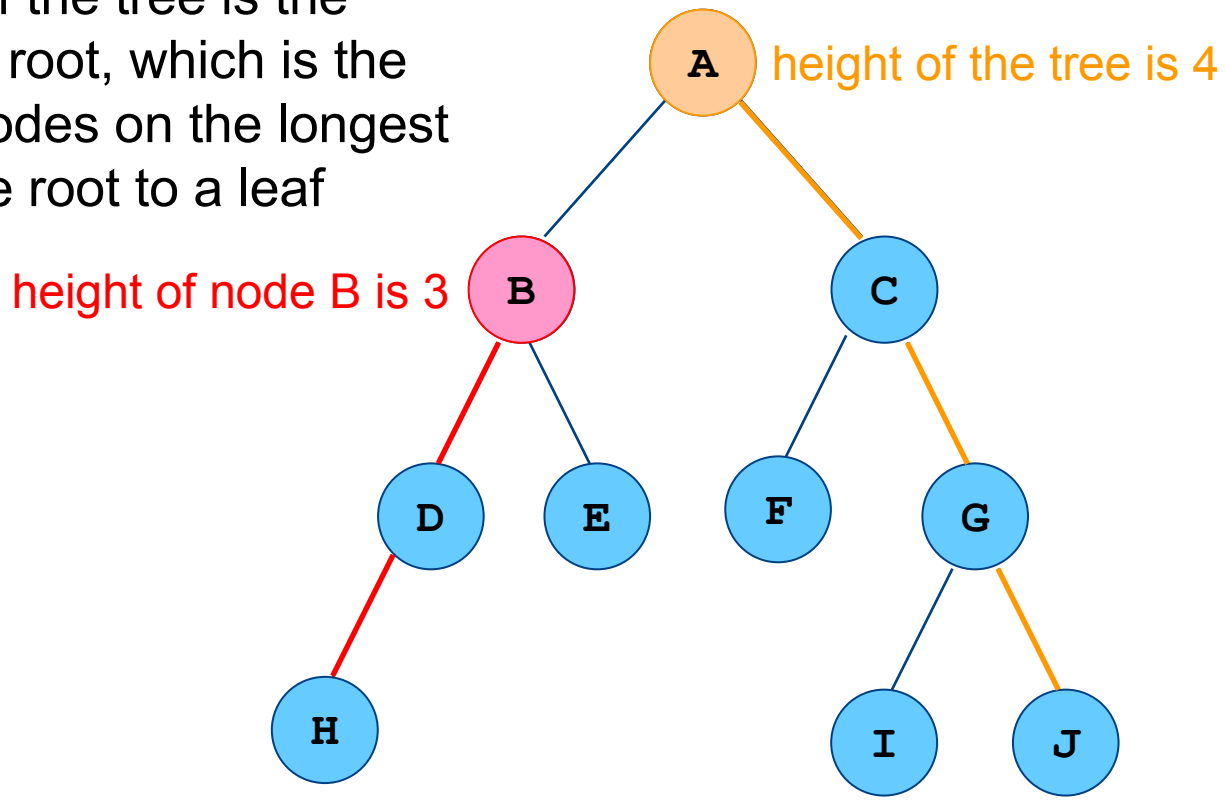
- Let N and E be the number of nodes and edges of tree T respectively. Let N_L (N_R) and E_L(E_R) be the number of nodes and edges of T_L (T_R) respectively.

- $N=N_L+N_R+1$ and $E=E_L+E_R+2$ (there is no edge running between T_R and T_L)
- Based on induction hypothesis: $E_L=N_L-1$ and $E_R=N_R-1$ (because T_R and T_L have less than N nodes).
- Hence $E=N_L-1 + N_R-1 + 2 = N_L+N_R = N - 1$

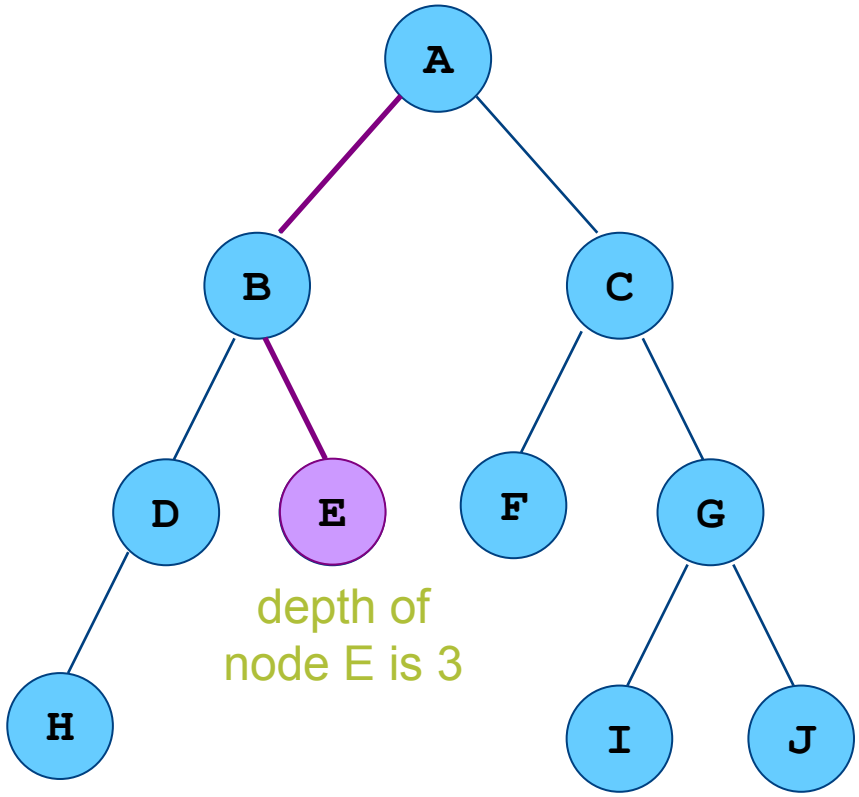
Measuring Trees

- The **height** of a node *v* is the number of nodes on the longest path from *v* to a leaf
  - The height of the tree is the height of the root, which is the number of nodes on the longest path from the root to a leaf
- The **depth** of a node *v* is the number of nodes on the path from the root to *v*
  - This is also referred to as the **level** of a node
- Note that there are slightly different formulations of the height of a tree
  - Where the height of a tree is said to be the length (the number of edges) on the longest path from node to a leaf

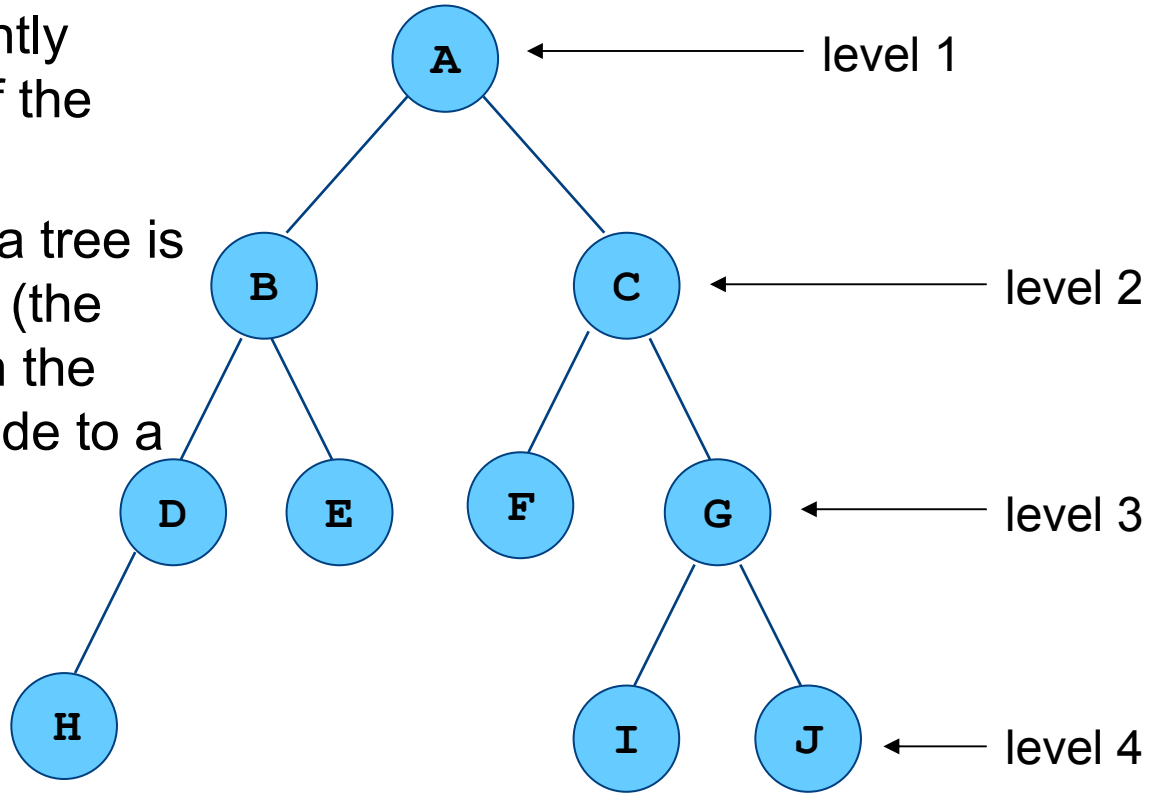
- The **height** of a node *v* is the number of nodes on the longest path from *v* to a leaf
  - The height of the tree is the height of the root, which is the number of nodes on the longest path from the root to a leaf



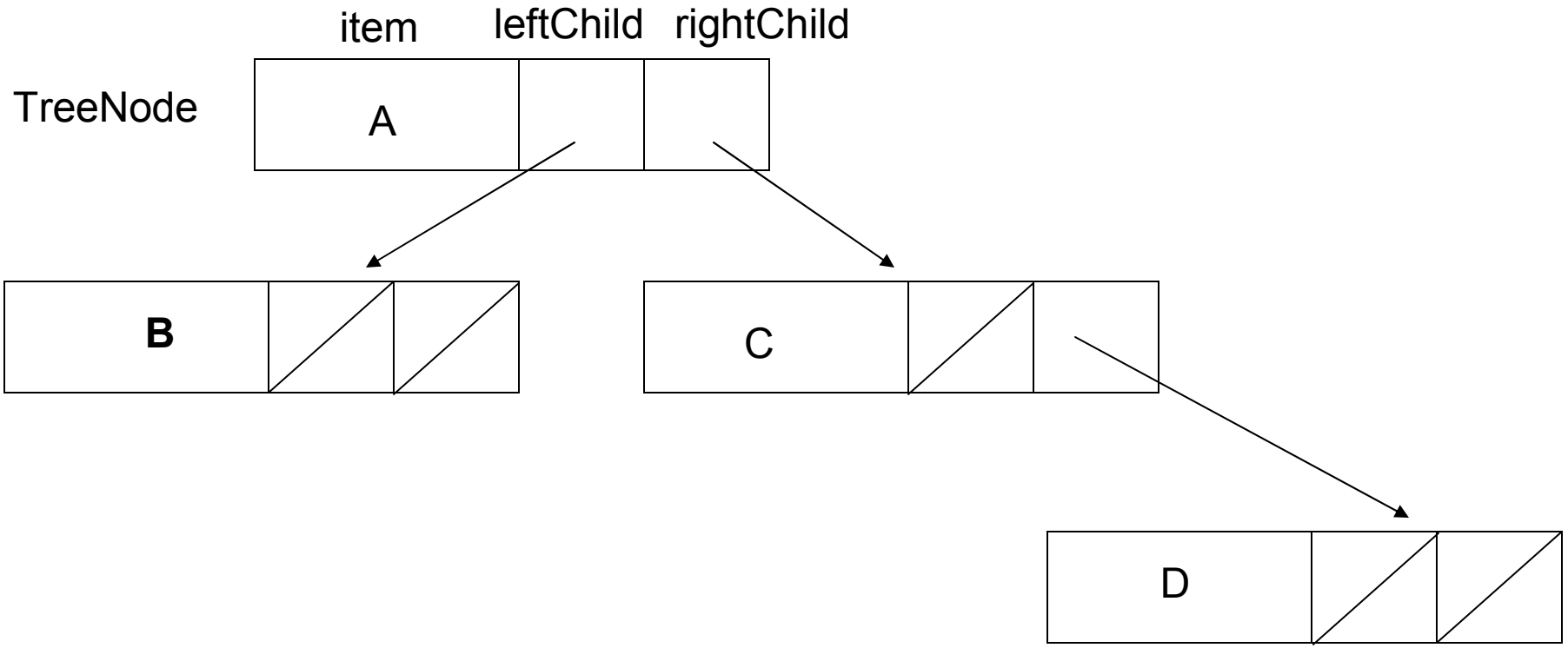
- The **depth** of a node *v* is the number of nodes on the path from the root to *v*
  - This is also referred to as the **level** of a node



- All the nodes in the same distance from root is called to be in the same level. The root of the tree is on level 1
- Note that there are slightly different formulations of the height of a tree
  - Where the height of a tree is said to be the length (the number of edges) on the longest path from node to a leaf



- Representation of a binary tree is very much like linked list.
- Each node has two references one to the right child and one to the left child.



Representation of binary trees

```
public class TreeNode {
  private Comparable item;
  private TreeNode leftChild;
  private TreeNode rightChild;

  public TreeNode(Comparable newItem) {
  // Initializes tree node with item and no children (a leaf).
    item = newItem;
    leftChild  = null;
    rightChild = null;
  }  // end constructor

  public TreeNode(T newItem,
                  TreeNode left, TreeNode right) {
  // Initializes tree node with item and
  // the left and right children references.
    item = newItem;
    leftChild  = left;
    rightChild = right;
  }  // end constructor

  public Comprable getItem() {
  // Returns the item field.
    return item;
  }  // end getItem
```

```
  public void setItem(Comprable newItem) {
  // Sets the item field to the new value newItem.
    item  = newItem;
  }  // end setItem

  public TreeNode getLeft() {
  // Returns the reference to the left child.
    return leftChild;
  }  // end getLeft

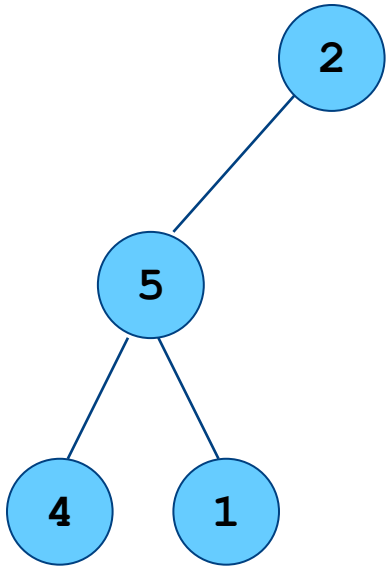
  public void setLeft(TreeNode left) {
  // Sets the left child reference to left.
    leftChild  = left;
  }  // end setLeft

  public TreeNode getRight() {
  // Returns the reference to the right child.
    return rightChild;
  }  // end getRight

  public void setRight(TreeNode right) {
  // Sets the right child reference to right.
    rightChild  = right;
  }  // end setRight
}  // end TreeNode
```

Representing a tree

```
TreeNode root=new
  TreeNode(new Integer(2));
root.setLeft(
  new TreeNode(new Integer(5),
    new TreeNode(new Integer(4)),
    new TreeNode(new Integer(1))));
```



Java Generics: Generic Classes

- ADT developed in this text relied upon the use of `Object` class or the `Comparable` interface.
- Problems with this approach
  - Items of any type could be added to same ADT instance
  - ADT instance returns objects
    - Cast operations are needed:

    e.g `Integer I = (Integer) stack.pop();`
    - May lead to class-cast exceptions
- Avoid these issues by using Java generics
  - To specify a class in terms of a data-type parameter

Example – Nodes with a type

```
//defining a generic class. T is the input data type
public class Node <T>
{
     private T item;
     private Node<T> next;
     public Node (T x, Node <T> n){
             item = x; next = n;
     }
     public T getItem()
     {
             return item;
     }
}

//creating an object of a generic class.
Node intNode= new Node <Integer> (new Integer(1), null);
Node stringNode= new Node <String> (new String ("Jhon"), intNode);
```

Things you cannot do with generics

- When creating an object of a generic class the input data type must be a defined class (not a primitive data type)

```
Node <int> intNode=new Node<int> (1, null);
```

- Java does not allow generic types to be used in array declaration.

```
T[] items = new T[10];
```

- The alternative is to use either the ArrayList or Vector class in Java.

```
Vector<T> items = new Vector<T> ();
ArrayList<T> items = new ArrayList<T> ();
```

Stack Implementation using Java generic class.

```
public class Stack <T>{
    private ArrayList<T> items;
    private int top, capacity;
    public Stack(int c){
        items = new ArrayList<T>(c);
        top = -1;
        capacity = c;
    }
    public boolean isEmpty(){
        return top == -1;
    }

    public T pop() throws StackException{
        if(isEmpty())
            throw new StackException("Stack is empty");
        return items.get(top--);

    }

    public void push (T x) throws StackException{
        if(top==capacity-1)
            throw new StackException("Stack is full");
        items.add(++top, x);
    }
}
```

```
public static void main(String args[]){
   Stack <Integer> s = new Stack <Integer> (100);
   for(int i=0; i<100; i++)
      s.push(new Integer(i));
   for(int i=0; i<100; i++){
      Integer x = s.pop();  //no casting is required
      System.out.println(x);
   }
}
```