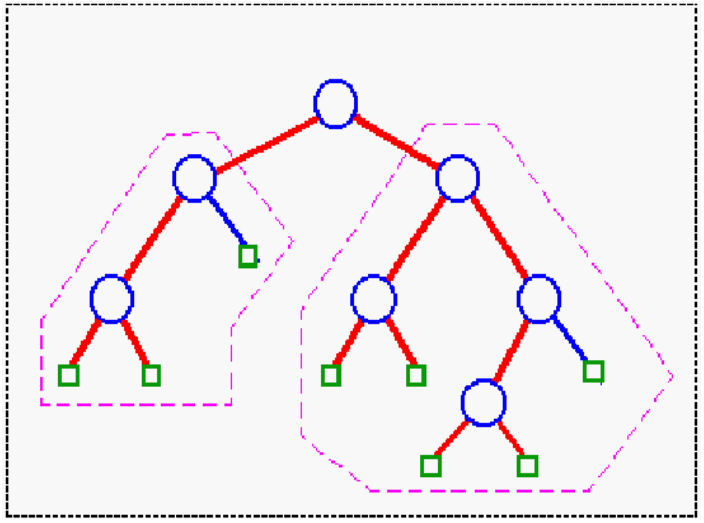
Trees

School of Computer Science and Informatics
University College Dublin, Ireland

Trees: Introduction

- A Tree is a **hierarchical ADT** where data is related in terms of parent-child relationships.
  - Each element (node) in the tree has **at most 1 parent**.
  - Each element (node) may have **0 or more children**.
  - Each tree will include exactly one element (node), known as the **root**, which has no parent.

- Trees can be defined **recursively**:
  - A tree T consists of a root node, *r*, plus a set of subtrees whose roots are children of *r*.

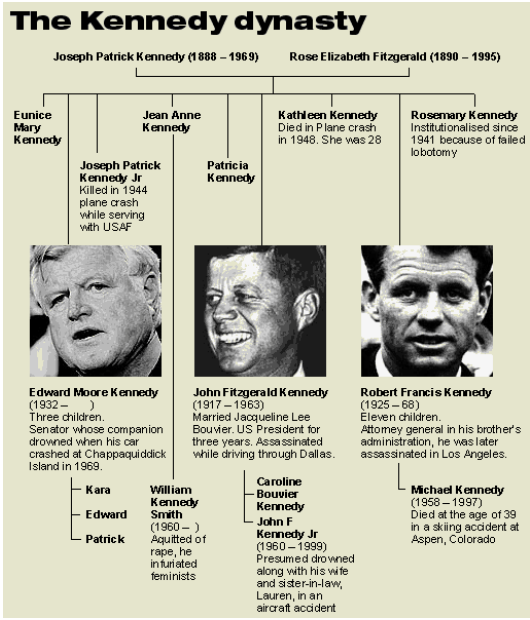


Trees: Introduction

- A Tree is a **hierarchical ADT** where data is related in terms of parent-child relationships.
  - Each element (node) in the tree has **at most 1 parent**.
  - Each element (node) may have **0 or more children**.
  - Each tree will include exactly one element (node), known as the **root**, which has no parent.

- Trees can be defined **recursively**:
  - A tree T consists of a root node, *r*, plus a set of subtrees whose roots are children of *r*.

- Trees occur throughout the real world:
  - Family Trees

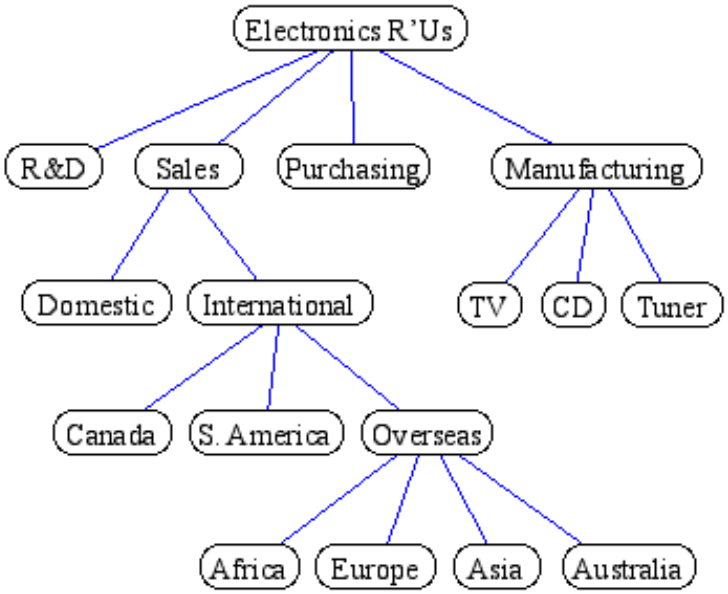


Trees: Introduction

- A Tree is a **hierarchical ADT** where data is related in terms of parent-child relationships.
  - Each element (node) in the tree has **at most 1 parent**.
  - Each element (node) may have **0 or more children**.
  - Each tree will include exactly one element (node), known as the **root**, which has no parent.

- Trees can be defined **recursively**:
  - A tree T consists of a root node, *r*, plus a set of subtrees whose roots are children of *r*.

- Trees occur throughout the real world:
  - Family Trees
  - Company Structures

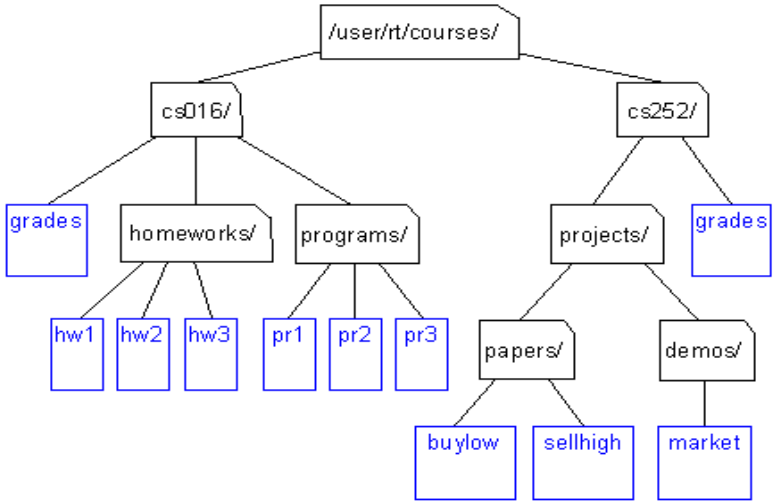


Trees: Introduction

- A Tree is a **hierarchical ADT** where data is related in terms of parent-child relationships.
  - Each element (node) in the tree has **at most 1 parent**.
  - Each element (node) may have **0 or more children**.
  - Each tree will include exactly one element (node), known as the **root**, which has no parent.

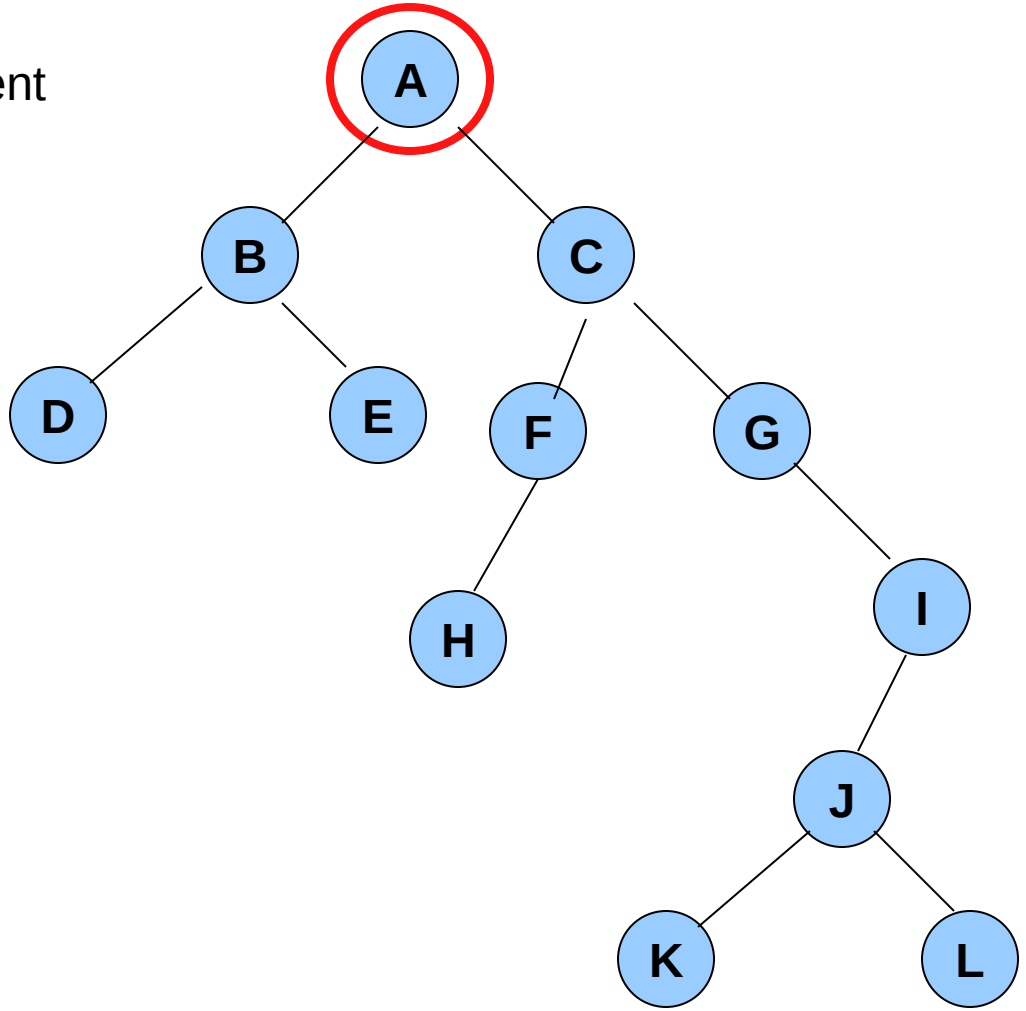
- Trees can be defined **recursively**:
  - A tree T consists of a root node, *r*, plus a set of subtrees whose roots are children of *r*.

- Trees occur throughout the real world:
  - Family Trees
  - Company Structures
  - File Systems



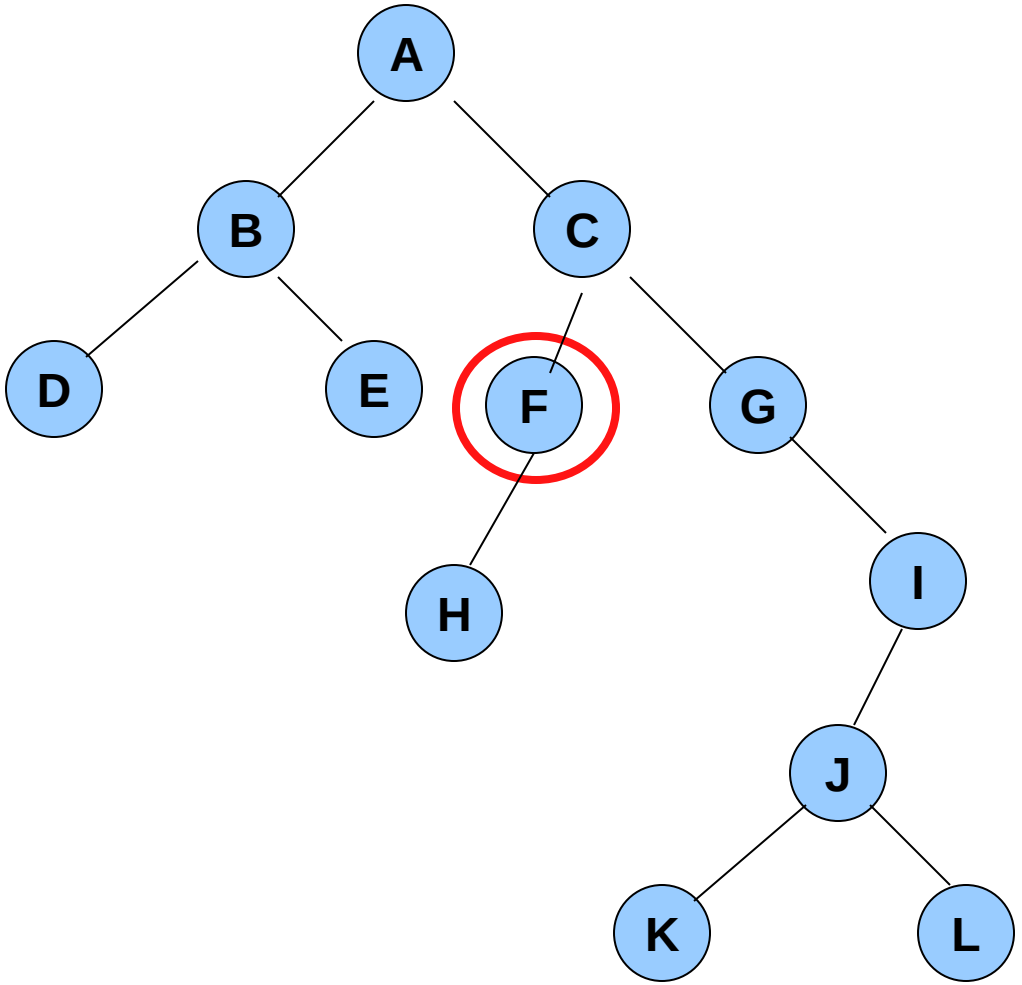
Trees: Terminology

- **Root** of tree: A
  - The only node with no parent



Trees: Terminology

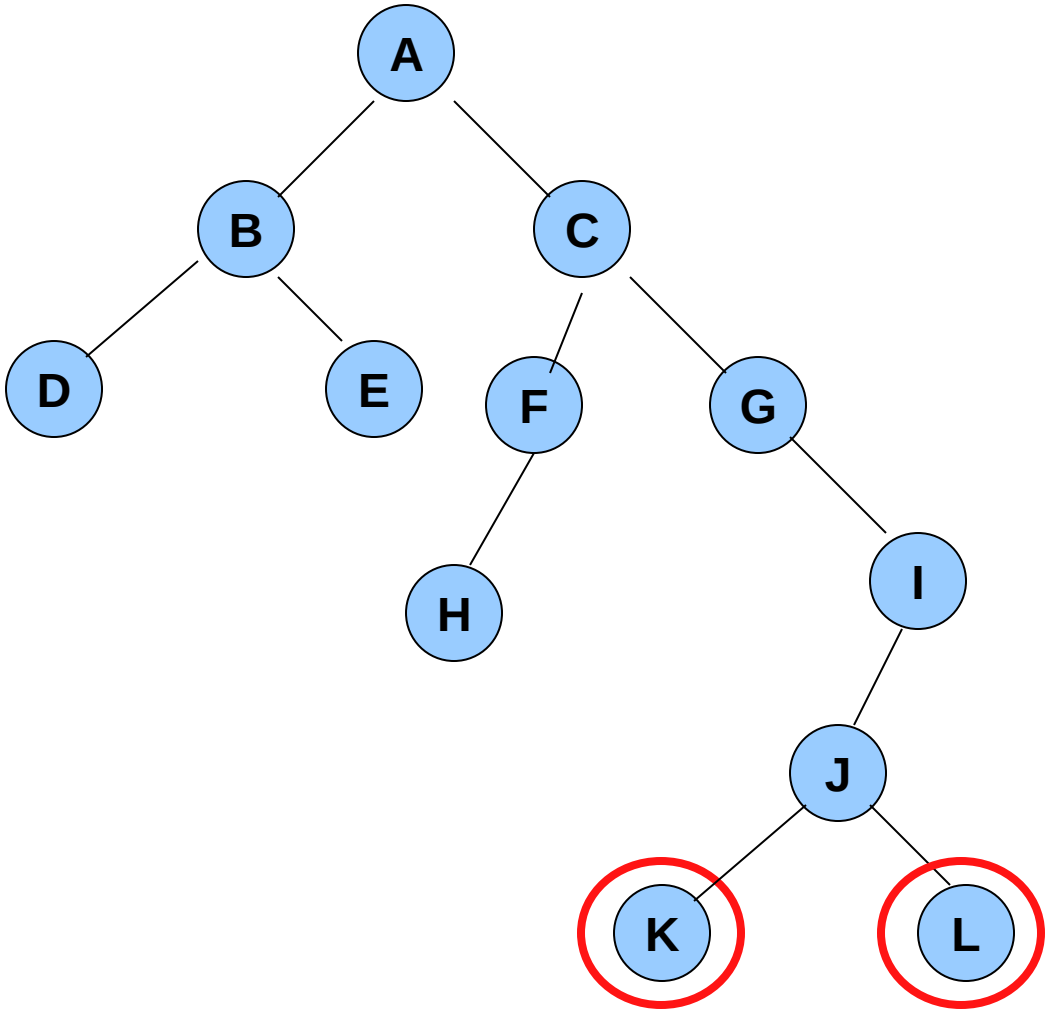
- **Root** of tree: A
- **Parent** of H: F
  - C is the parent of F
  - A is the parent of C



Trees: Terminology

- **Root** of tree: A
- **Parent** of H: F

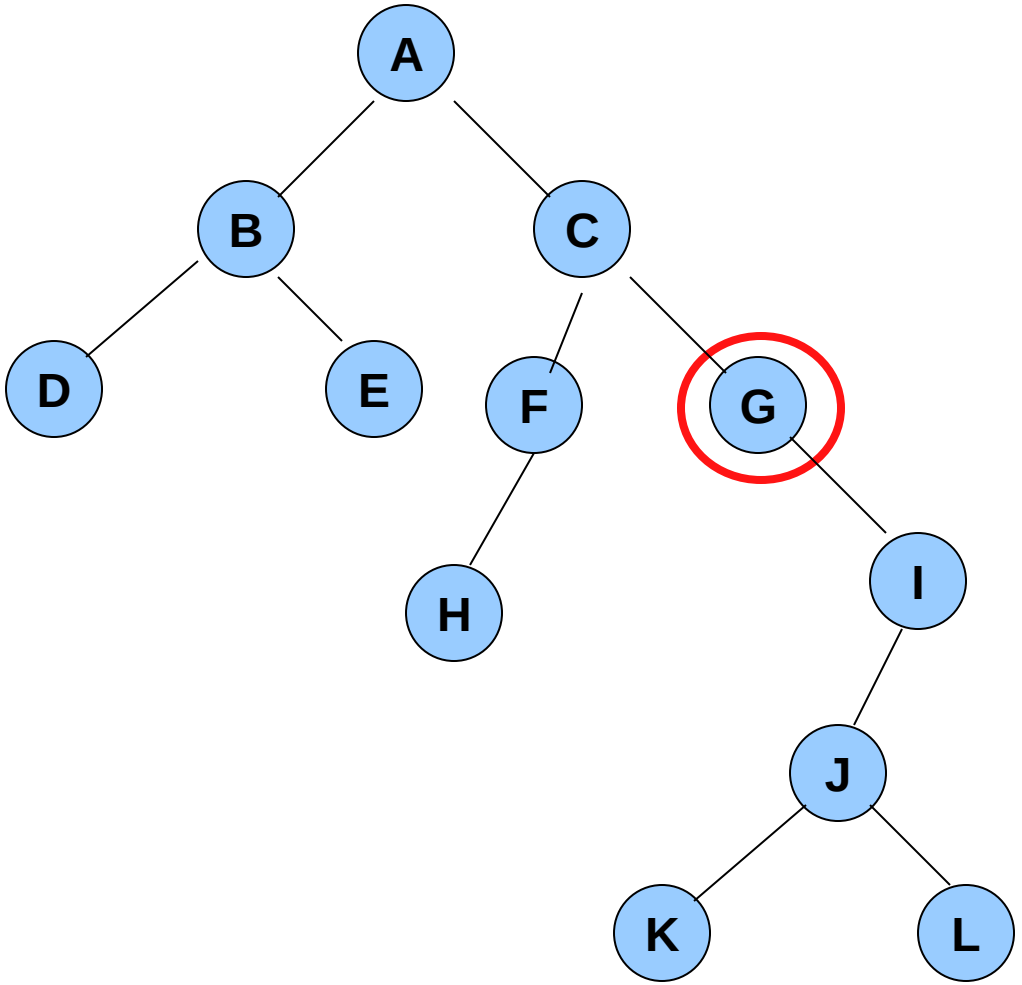
- **Children** of J: K and L
  - K is a child of J
  - L is a child of J



Trees: Terminology

- **Root** of tree: A
- **Parent** of H: F
- **Children** of J: K and L

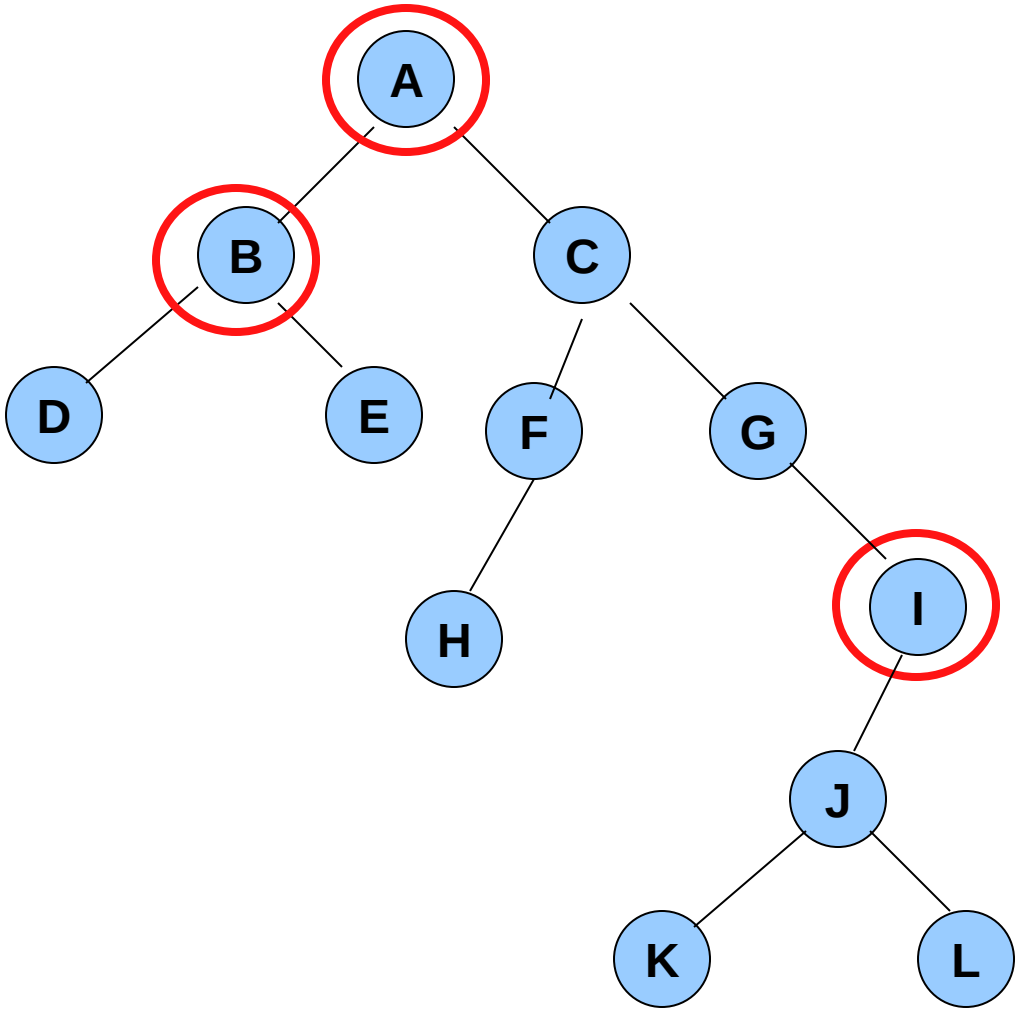
- **Sibling** of F: G
  - Sibling of G: F
  - J does not have a sibling!



Trees: Terminology

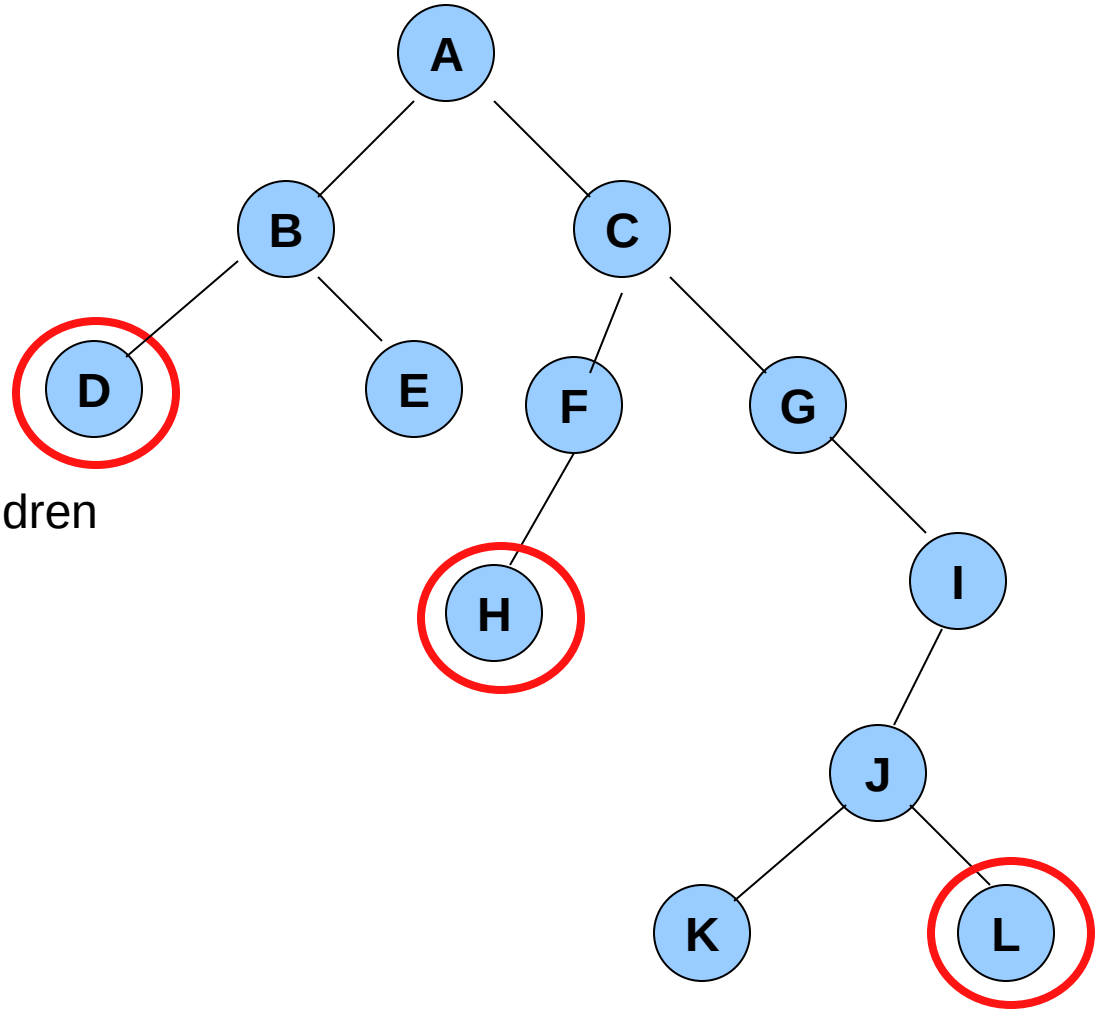
- **Root** of tree: A
- **Parent** of H: F
- **Children** of J: K and L
- **Sibling** of F: G

- **Internal Nodes**: A, B, I
  - Any node that has children



Trees: Terminology

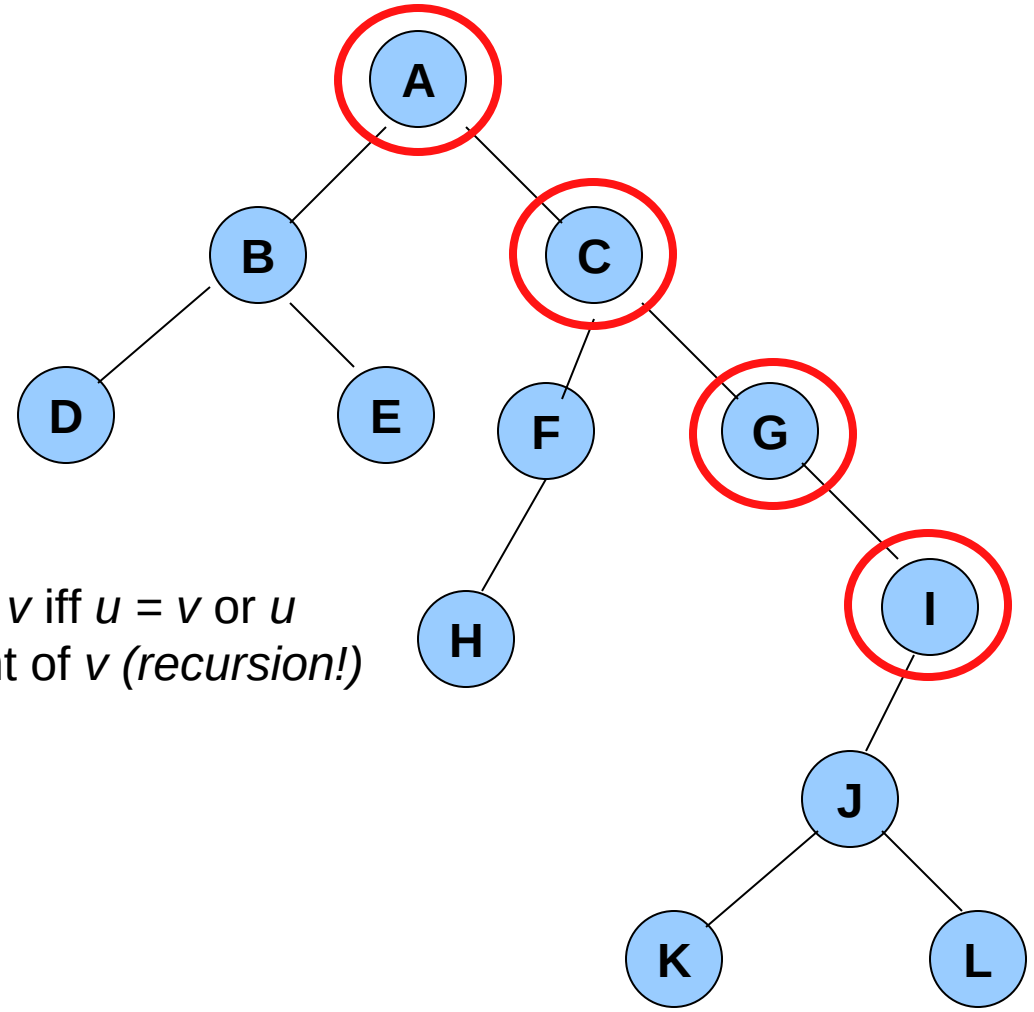
- **Root** of tree: A
- **Parent** of H: F
- **Children** of J: K and L
- **Sibling** of F: G
- **Internal Nodes**: A, B, I
- **External Nodes**: D, H, L
  - Any node that has no children
  - Also known as *leaves*



Trees: Terminology

- **Root** of tree: A
- **Parent** of H: F
- **Children** of J: K and L
- **Sibling** of F: G
- **Internal Nodes**: A, B, I
- **External Nodes**: D, H, L

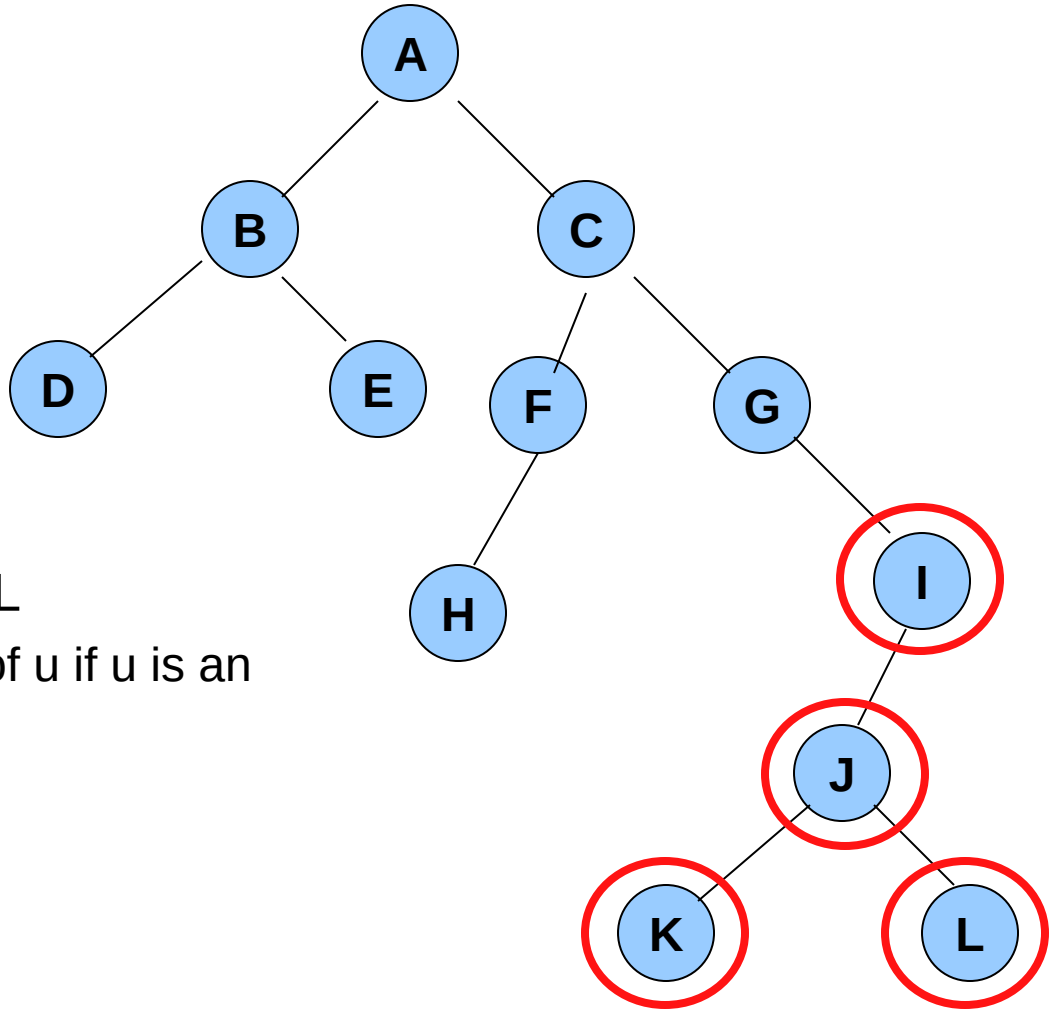
- **Ancestor** of I: A, C, G, or I
  - A node *u* is an ancestor of *v* iff *u* = *v* or *u* is an ancestor of the parent of *v (recursion!)*



Trees: Terminology

- **Root** of tree: A
- **Parent** of H: F
- **Children** of J: K and L
- **Sibling** of F: G
- **Internal** Nodes: A, B, I
- **External** Nodes: D, H, L
- **Ancestor** of I: A, C, G, or I

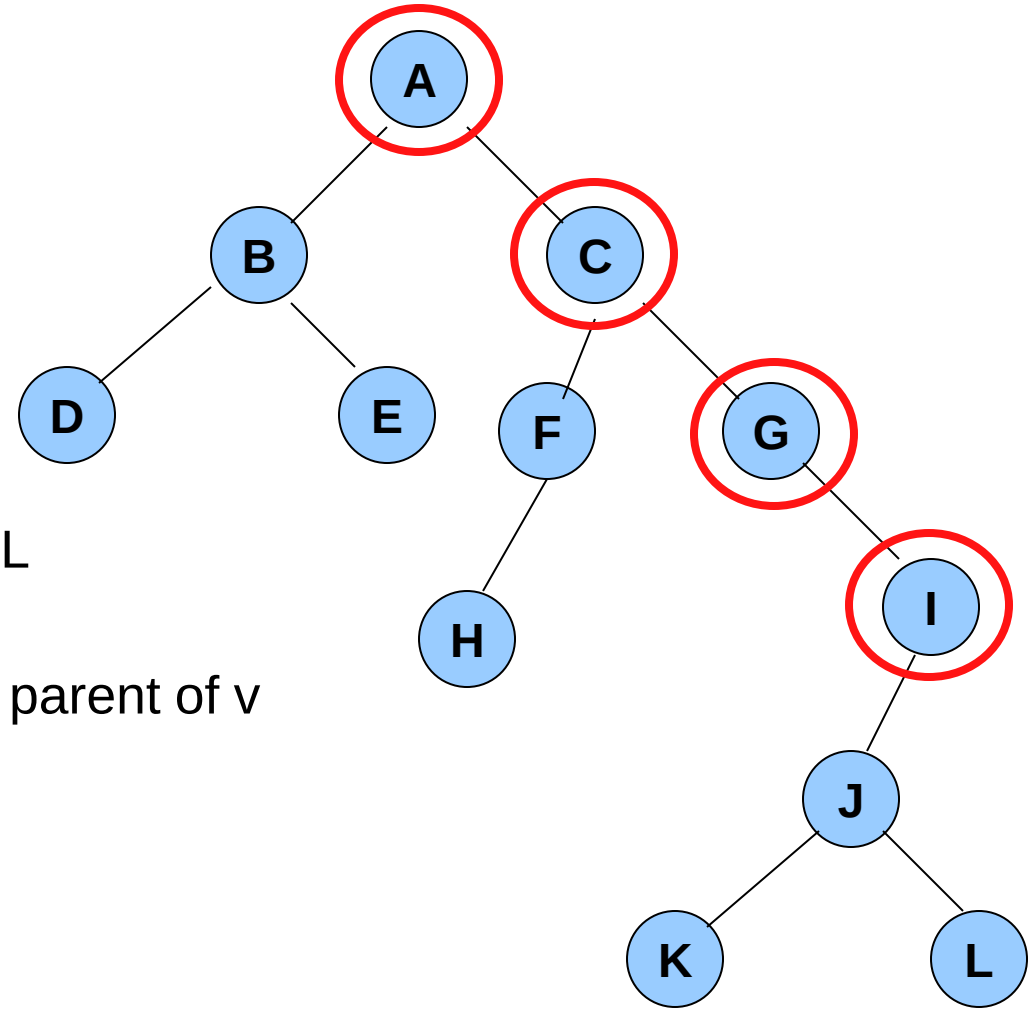
- **Descendent** of I: I, J, K, or L
  - A node v is a descendent of u if u is an ancestor of v



Trees: Terminology

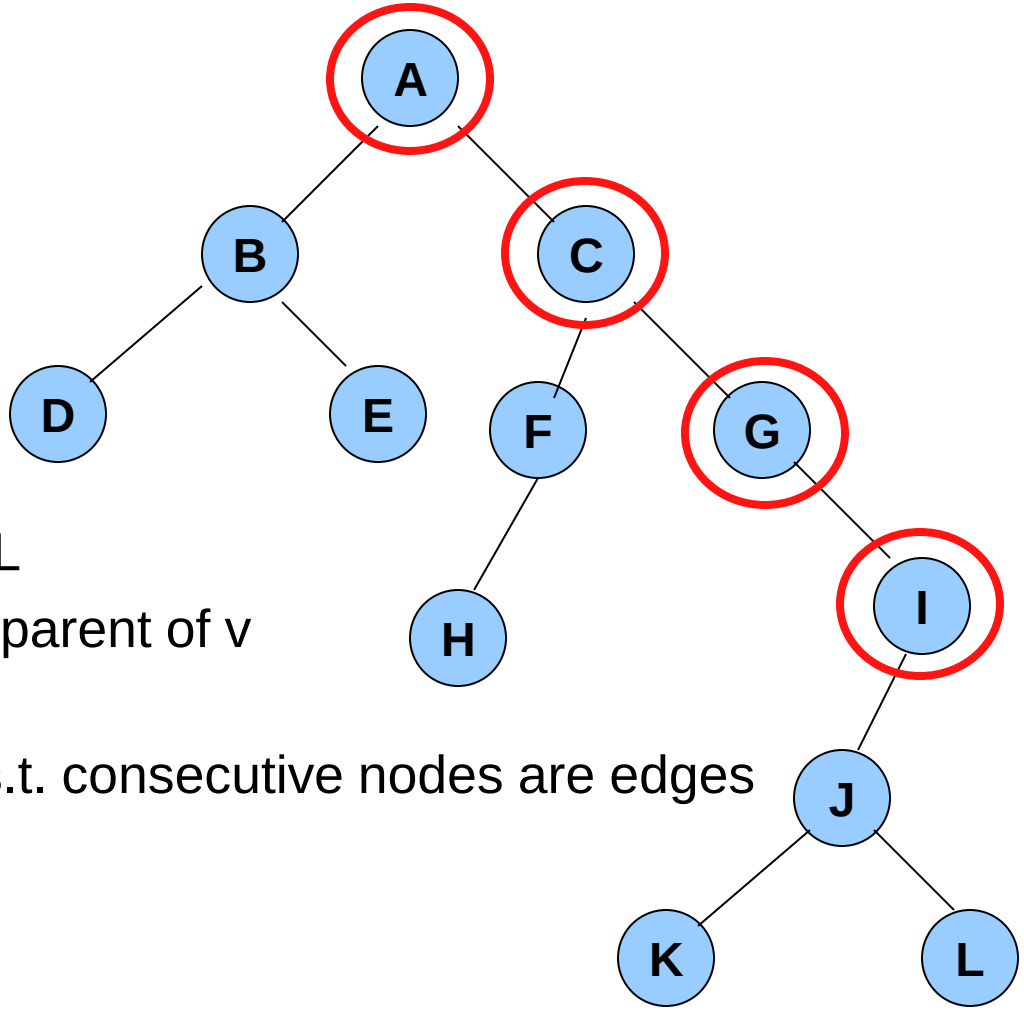
- **Root** of tree: A
- **Parent** of H: F
- **Children** of J: K and L
- **Sibling** of F: G
- **Internal** Nodes: A, B, I
- **External** Nodes: D, H, L
- **Ancestor** of I: A, C, G, or I
- **Descendent** of I: I, J, K, or L

- **Edge**: pair (u,v) s.t. u is the parent of v
  - E.g. (A, B), (G, I)
  - (A, G) is not an edge!!!



Trees: Terminology

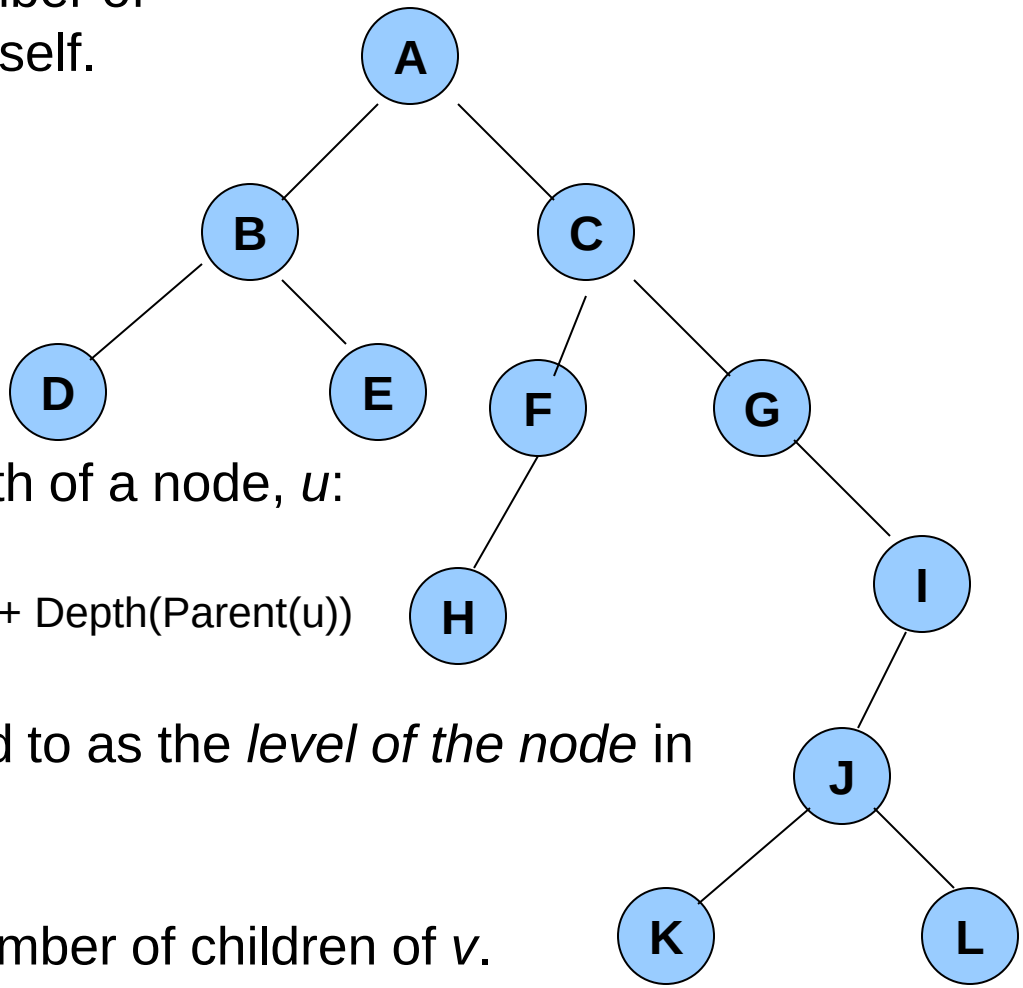
- **Root** of tree: A
- **Parent** of H: F
- **Children** of J: K and L
- **Sibling** of F: G
- **Internal** Nodes: A, B, I
- **External** Nodes: D, H, L
- **Ancestor** of I: A, C, G, or I
- **Descendent** of I: I, J, K, or L
- **Edge**: pair (u,v) s.t. u is the parent of v



- **Path**: sequence (n_1, ..., n_i) s.t. consecutive nodes are edges
  - E.g. (A, C, G, I)
  - (A, B, G) is not an path!!!

Trees: Properties

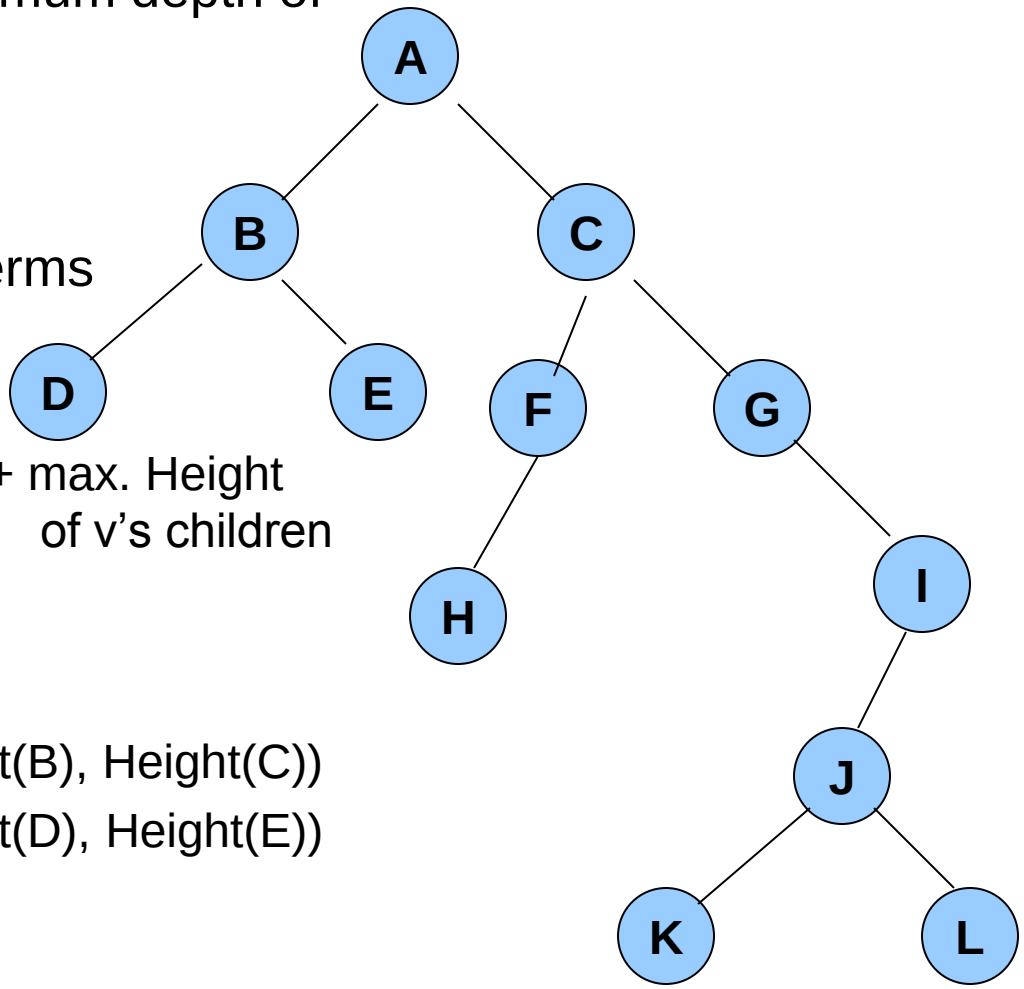
- **Depth** of a node, *v*: the number of ancestors of *v* excluding *v* itself.
  - Depth(A) = 0
  - Depth(G) = 2
  - Depth(E) = 2
  - Depth(K) = 5



- Recursive definition for depth of a node, *u*:
  - *u* is the root: Depth(u) = 0
  - *u* is not the root: Depth(*u*) = 1 + Depth(Parent(u))
- Depth is sometimes referred to as the *level of the node* in the tree.
- **Degree** of a node, *v*: the number of children of *v*.
  - Degree(A) = 2, Degree(G) = 1, Degree(E) = 0

Trees: Properties

- **Height** of a tree T: the maximum depth of an external node of T.
  - Height(T) = 5
- Mathematically defined in terms of the height of a node, *v*:
  - v is external: Height(v) = 0
  - v is internal: Height(v) = 1 + max. Height of v's children
- Example:
  - Height(A) = 1 + max(Height(B), Height(C))
  - Height(B) = 1 + max(Height(D), Height(E))
  - Height(D) = Height(E) = 0
  - ...



Tree ADT

- Trees make use of the **Position ADT** and have the following operations:
  - **root()** returns the Position for the root
  - **parent(p)** returns the Position of p's parent
  - **children(p)** returns an Iterator of the Positions of p's children
  - **isInternal(p)** does p have children?
  - **isExternal(p)** is p a leaf?
  - **isRoot(p)** is p==root()?
  - **size()** number of nodes
  - **isEmpty()** tests whether or not the tree is empty
  - **iterator()** returns an Iterator of every element in the tree
  - **positions()** returns an Iterator of every Position in the tree
  - **replace(p, e)** replaces the element at Position p with e

Tree Interface

```
public interface Tree<T> {
    public Position<T> root();
    public Position<T> parent(Position<T> p);
    public Iterator<Position<T>> children(Position<T> p);
    public boolean isInternal(Position<T> p);
    public boolean isExternal(Position<T> p);
    public boolean isRoot(Position<T> p);
    public int size();
    public boolean isEmpty();
    public Iterator<T> iterator();
    public Iterator<Position<T>> positions();
    public T replace(Position<T> p, T t);
}
```