# EECS 114:
# Engineering Data Structures and Algorithms
## Lecture 5


Instructor: Ryan Rusich

E-mail: rusichr@uci.edu

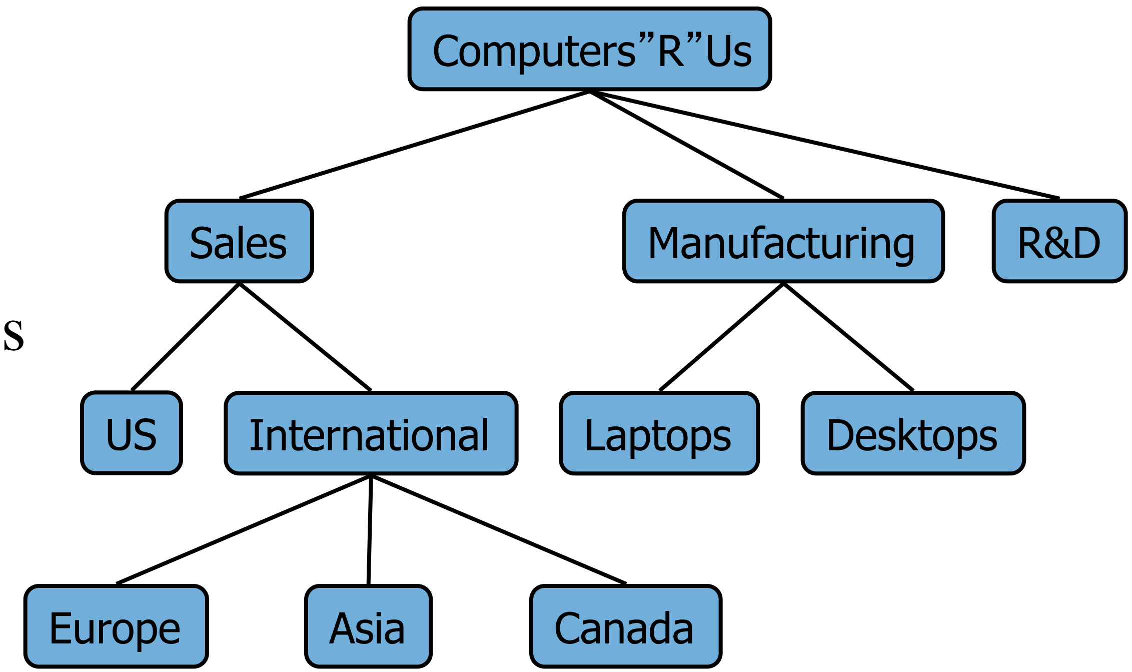
Office: EH 2204

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

# Trees

# What is a Tree

- A tree is an abstract model that captures hierarchical structure
- Tree - A connected acyclic graph of nodes



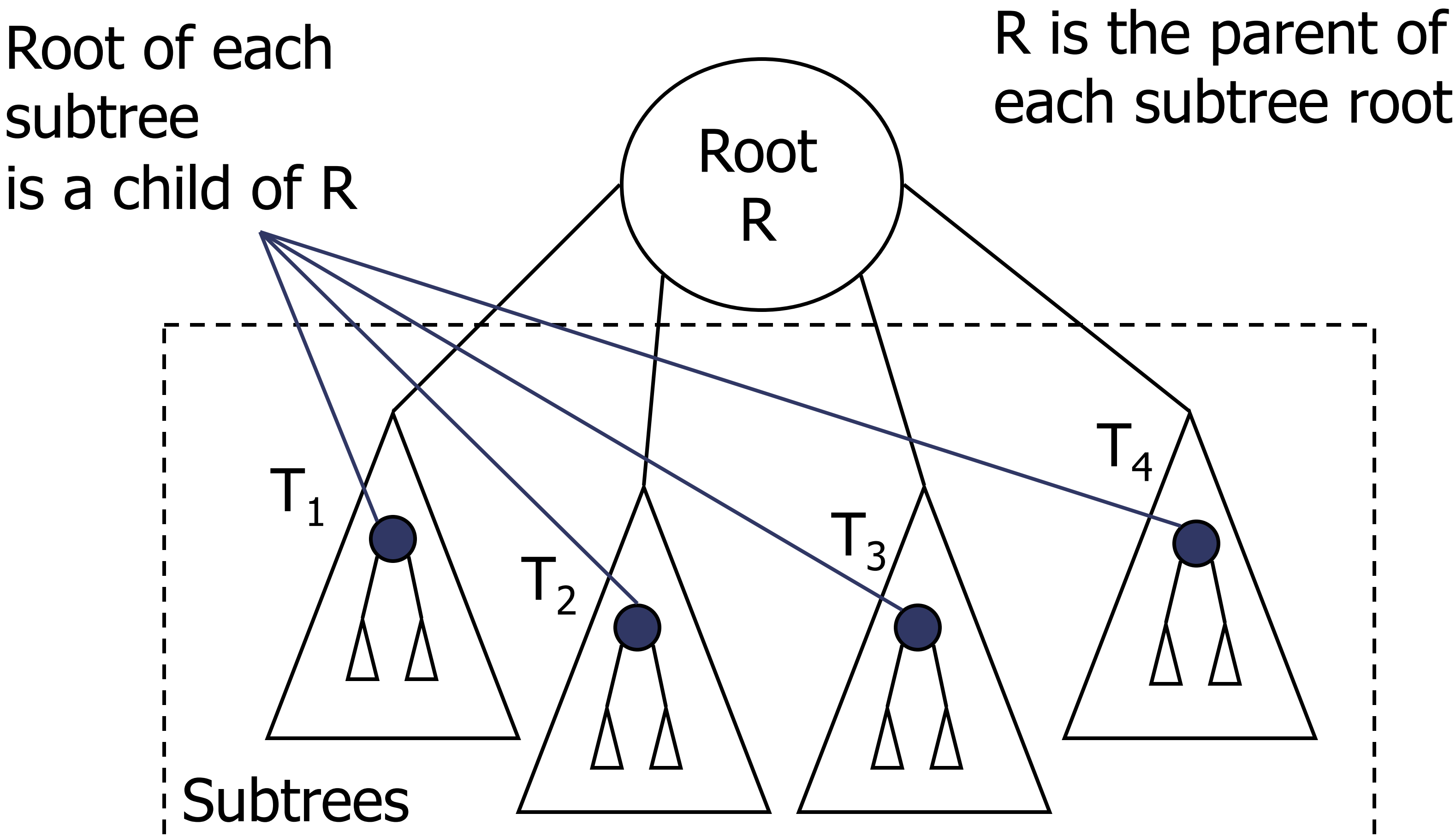
# Trees

- Examples:
  - Organizational charts
  - File systems – Unix, Windows
  - Genealogy (family tree)

# Recursive Definition of a Tree

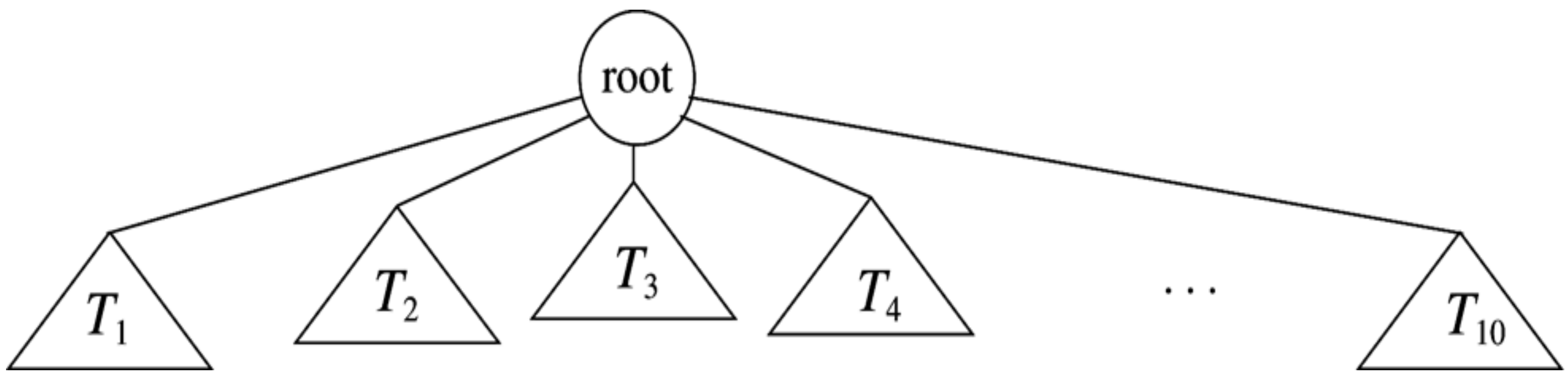
- A tree is a collection of tree nodes
  - One node is called the root
  - root has a rank of zero
- A tree can be empty, otherwise a tree consists of a node called a root (R) and zero or more non-empty subtrees each of whose roots are connected by an edge to the root.

# Recursive Definition of a Tree



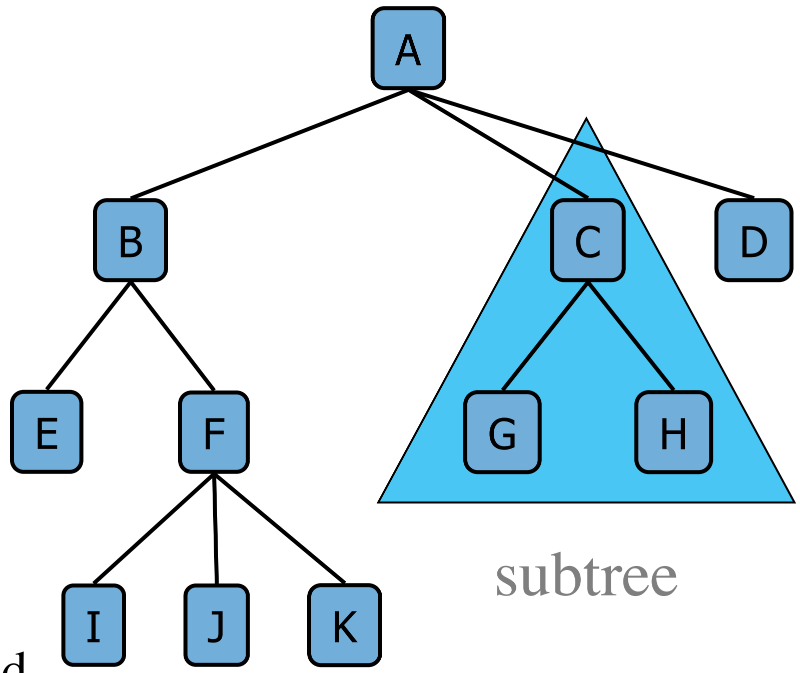
# Tree: Nodes and Edges

- A tree has **N nodes** and **N-1 edges** because each edge connects some node to its parent and every node except the root has exactly one parent



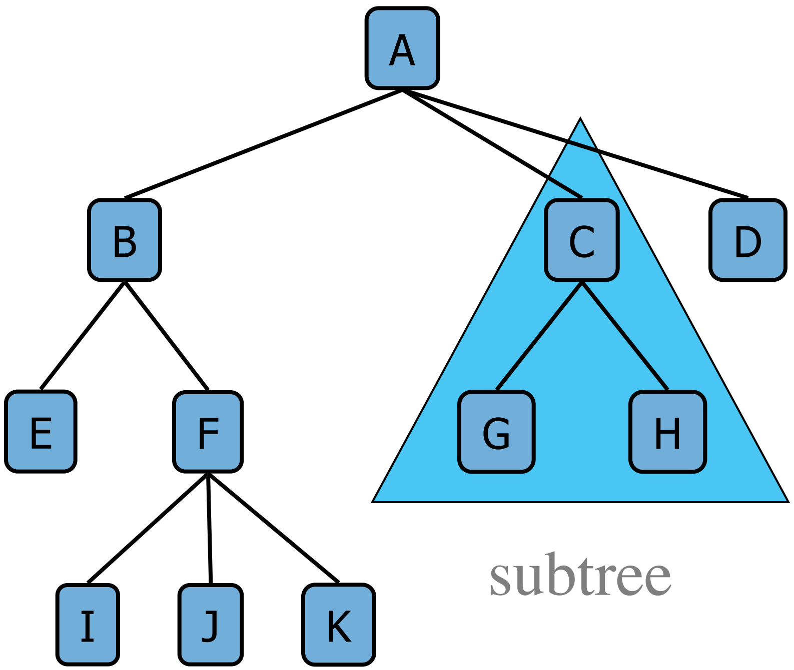
# Tree Terminology

- **Root:** node without parent (A)
- **Internal node:**
  - node with <u>at least one child</u>
  - (A, B, C, F)
- **External node (or leaf ):**
  - <u>node without children</u>
  - (E, I, J, K, G, H, D)
- **Ancestors of a node:**
  - parent,
  - grandparent,
  - grand-grandparent, etc.
- **Subtree**: tree consisting of a node and its descendants



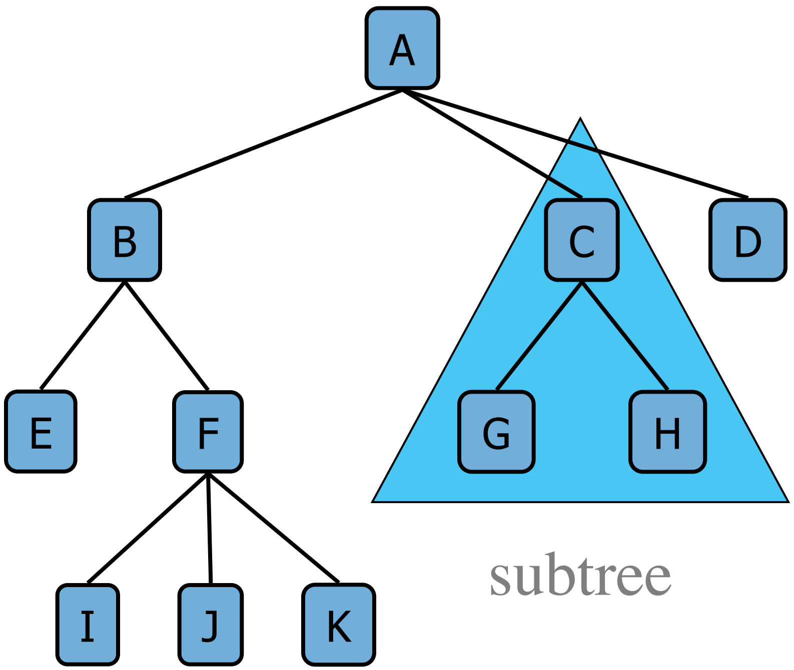
# Tree Terminology

- Descendant of a node:
  - child,
  - grandchild,
  - grand-grandchild, etc.
- Siblings:
  - nodes with the same parent



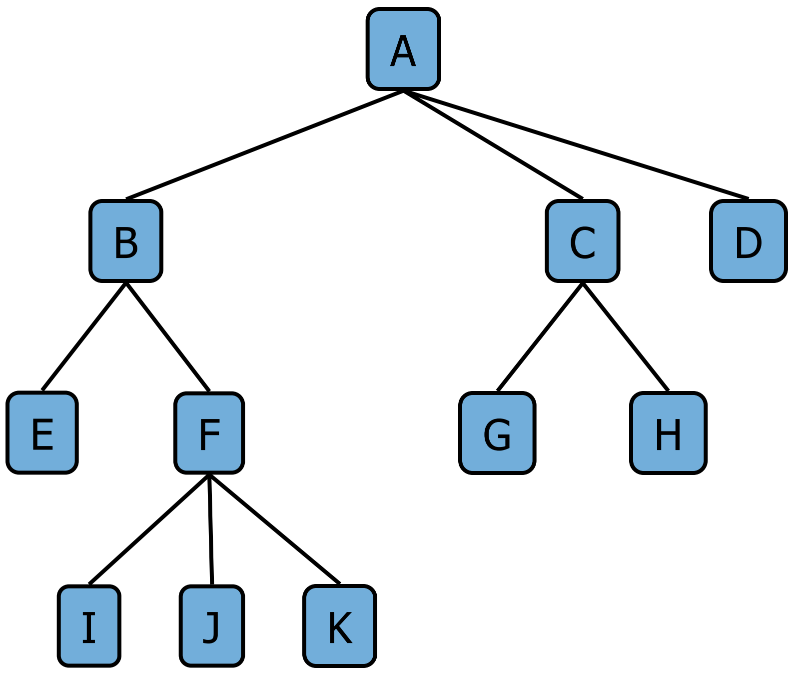
# Tree Terminology

- Depth of a node:
  - number of ancestors
- Height of root:
  - maximum depth of any node.
  - The height of a tree is the number of edges on the longest path from the root to a leaf
  - Can similarly find the height of any node



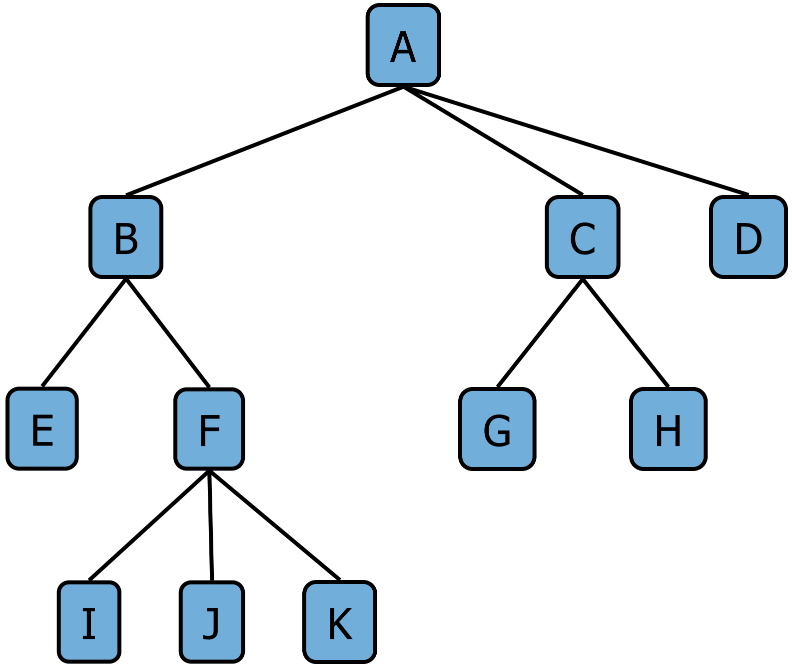
# Tree Terminology

- **Path** - a sequence of nodes $n_1, n_2, \ldots, n_k$ such that $n_i$ is the parent of $n_{i+1}$.
- Path length - the number of edges on the path $(k-1)$
- There is a path of length zero from every node to itself, i.e., shortest path.
- There is exactly <u>one</u> path from the root to each node (acyclic)



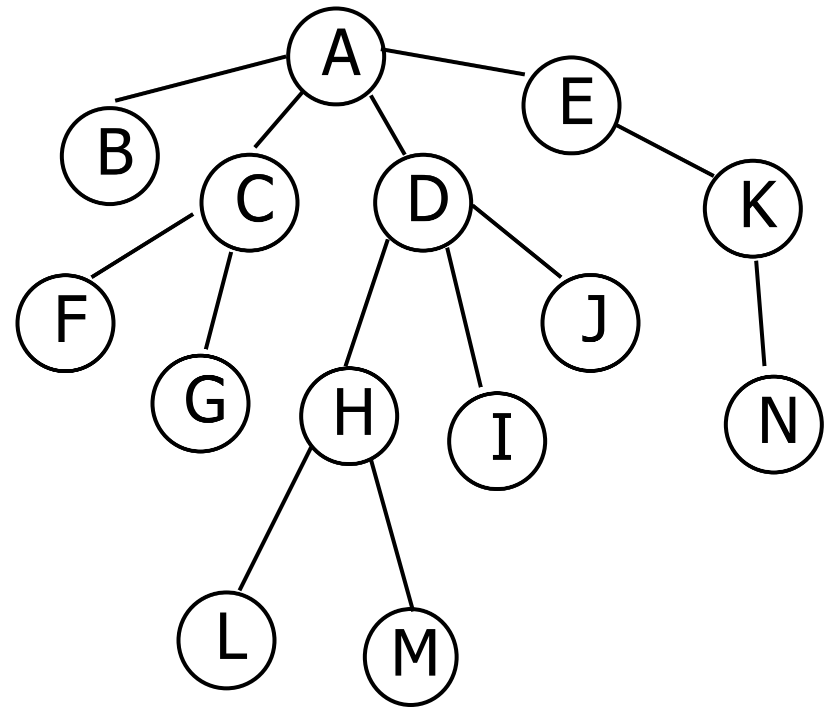
# Tree Terminology

- If there is a path from $\mathbf{n_1}$ to $\mathbf{n_2}$ then $\mathbf{n_1}$ is an ancestor of $\mathbf{n_2}$ and $\mathbf{n_2}$ is a descendant of $\mathbf{n_1}$



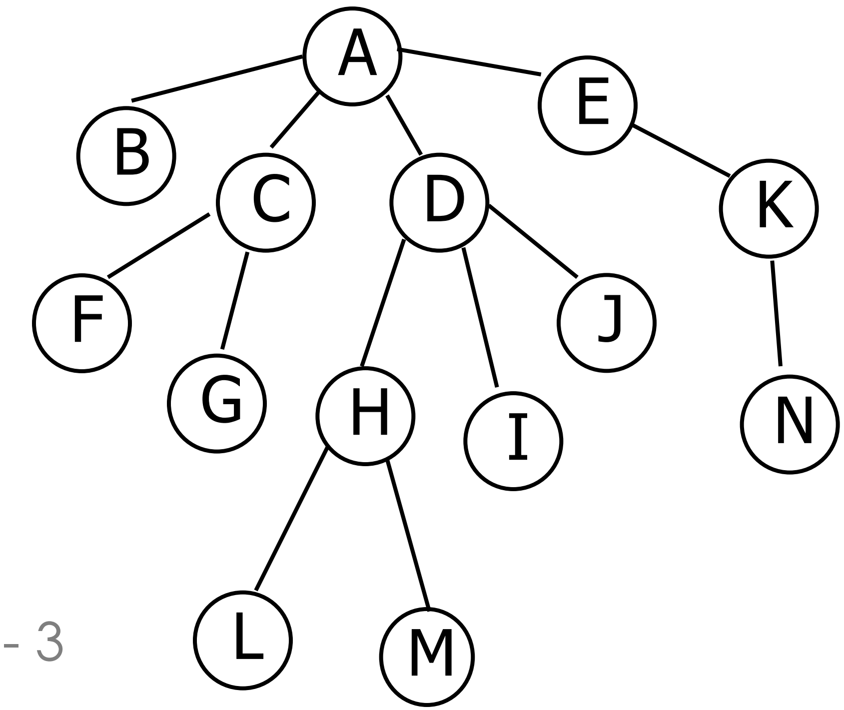
# Tree Terminology

- In class exercise - what is/are the…
  - Root
  - Leaves
  - Height of H
  - Depth of H
  - Ancestors of H
  - Descendants of H
  - Path from A to M
  - Length of path from A to M
  - Internal nodes

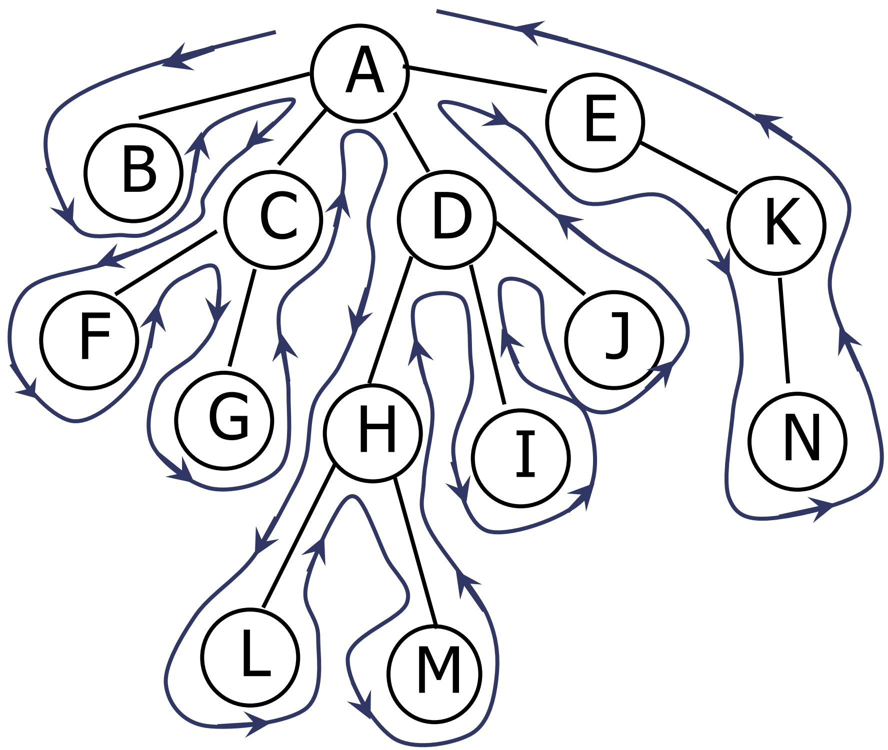


# Tree Terminology

- In class exercise - what is/are the…
  - Root - A
  - Leaves - B F G L M I J N
  - Height of H - 1
  - Depth of H - 2
  - Ancestors of H - A D
  - Descendants of H - L M
  - Path from A to M - A D H M
  - Length of path from A to M - 3
  - Internal nodes - A C D H E K



# Tree Traversals



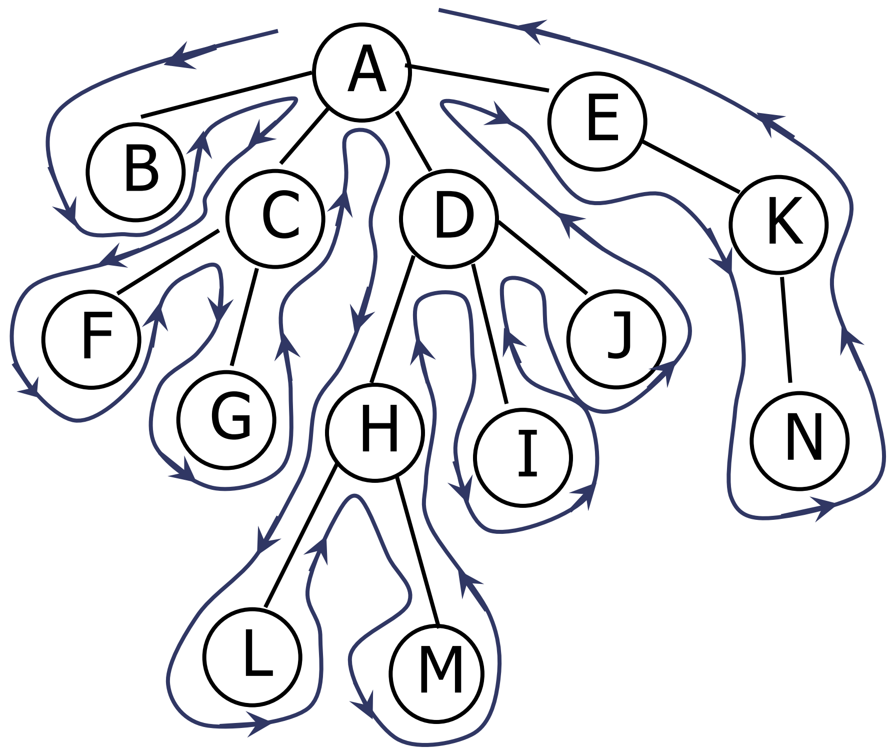
# Tree Traversal

- Definition: a **traversal** of a tree **T** is a systematic way of accessing, or "visiting", all of the nodes in **T**.

# Preorder Traversal

- In a **preorder traversal**, a node is visited before its descendants
- The **preorder** listing of the nodes of **T** is the root of **T** followed by the nodes of $\mathbf{T_1}$ in **preorder**, then the nodes of $\mathbf{T_2}$ in **preorder**, and so on up to the nodes of $\mathbf{T_k}$ in **preorder**.
- Intuition: List the node the <u>first time</u> it is visited

# Preorder Traversal

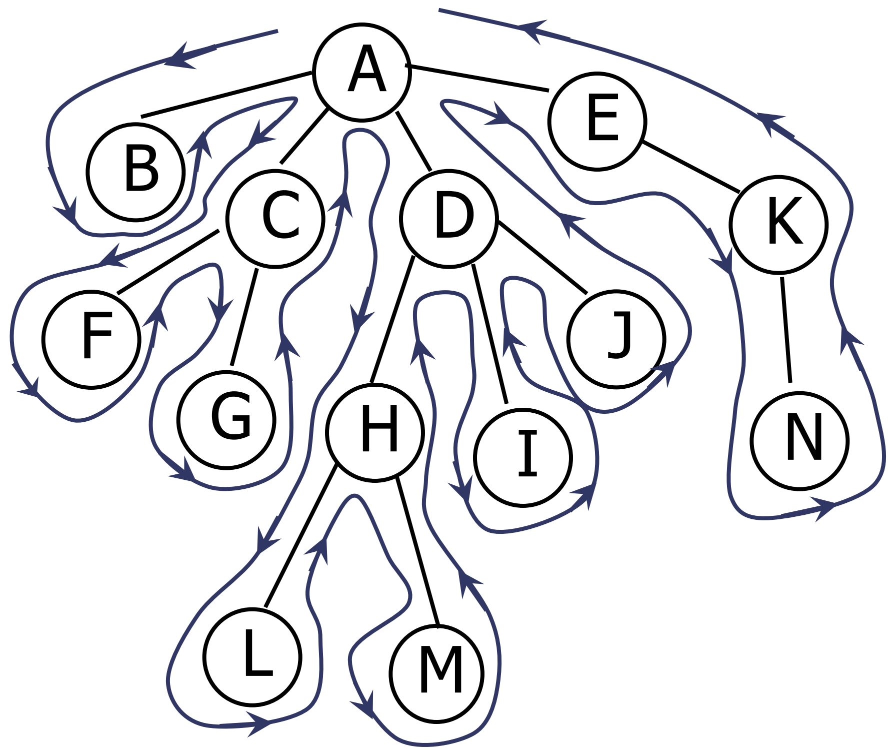



A B C F G D H L M I J E K N

# Inorder Traversal

- In a **inorder traversal**, a node is visited between its descendants.
- The **inorder** listing of the nodes of **T** are the nodes of $\mathbf{T_1}$ **inorder**, followed by the **root**, followed by the nodes of $\mathbf{T_2, \ldots, T_k}$ each group of nodes in **inorder**.
- Intuition: List the node the <u>second time</u> it is passed

# Inorder Traversal



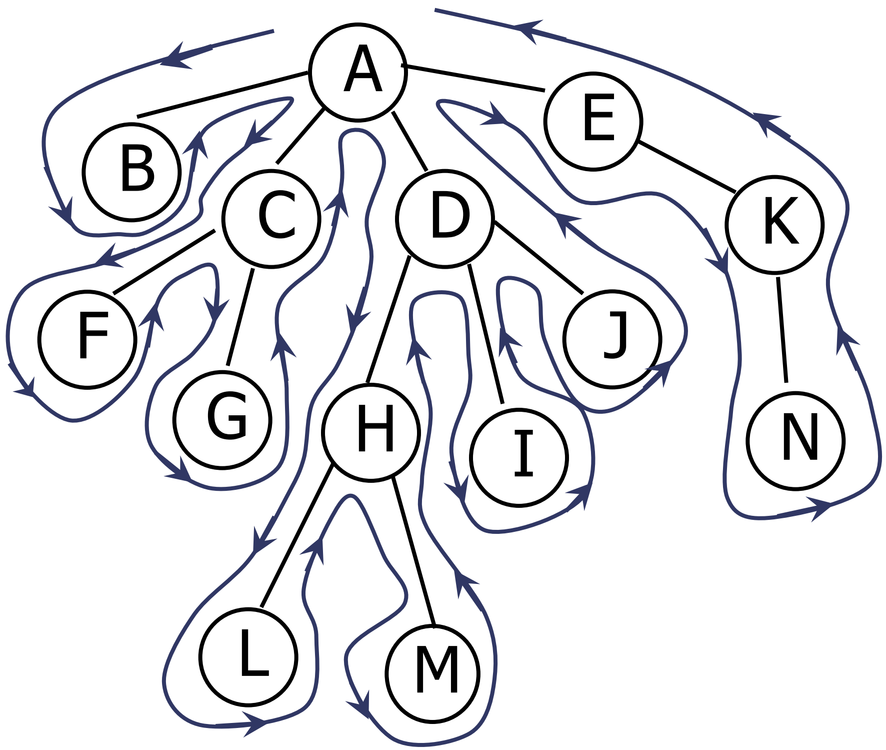

B A F C G L H M D I J N K E

# Postorder Traversal

- In a **postorder traversal**, a node is visited after its descendants.
- The **postorder** listing of the nodes of **T** is the nodes of $\mathbf{T_1}$ in **postorder** then the nodes of $\mathbf{T_2}$ in **postorder** and so on up to $\mathbf{T_k}$ all followed by the node **n.**
- Intuition: List the node the <u>last time</u> it is passed.

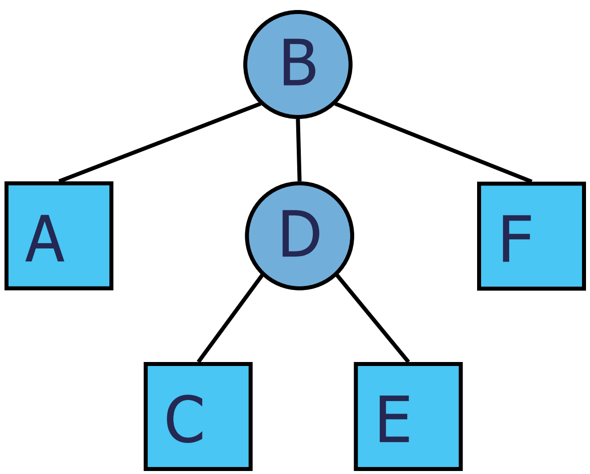
# Postorder Traversal

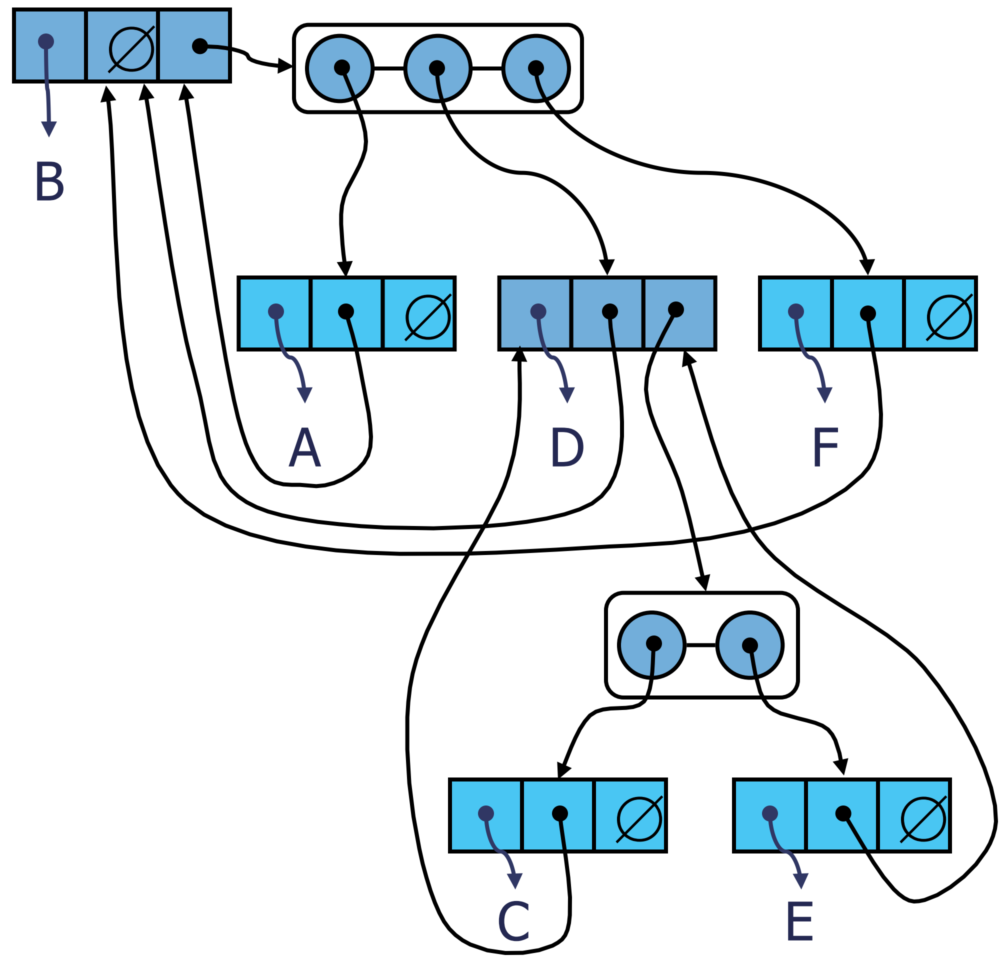
In class exercise



# Data Structure for Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes



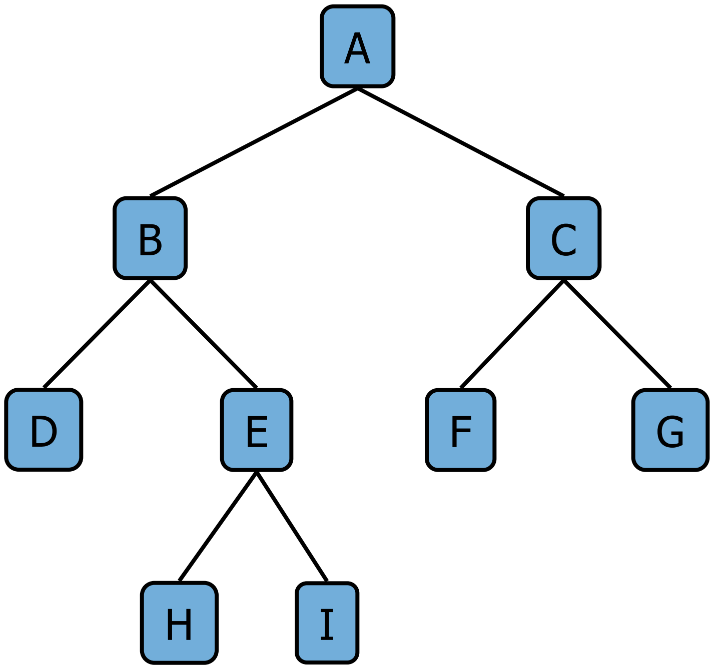



# Binary Tree

- A binary tree is a tree with the following properties:
  - Each internal node has at most two children
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a binary tree

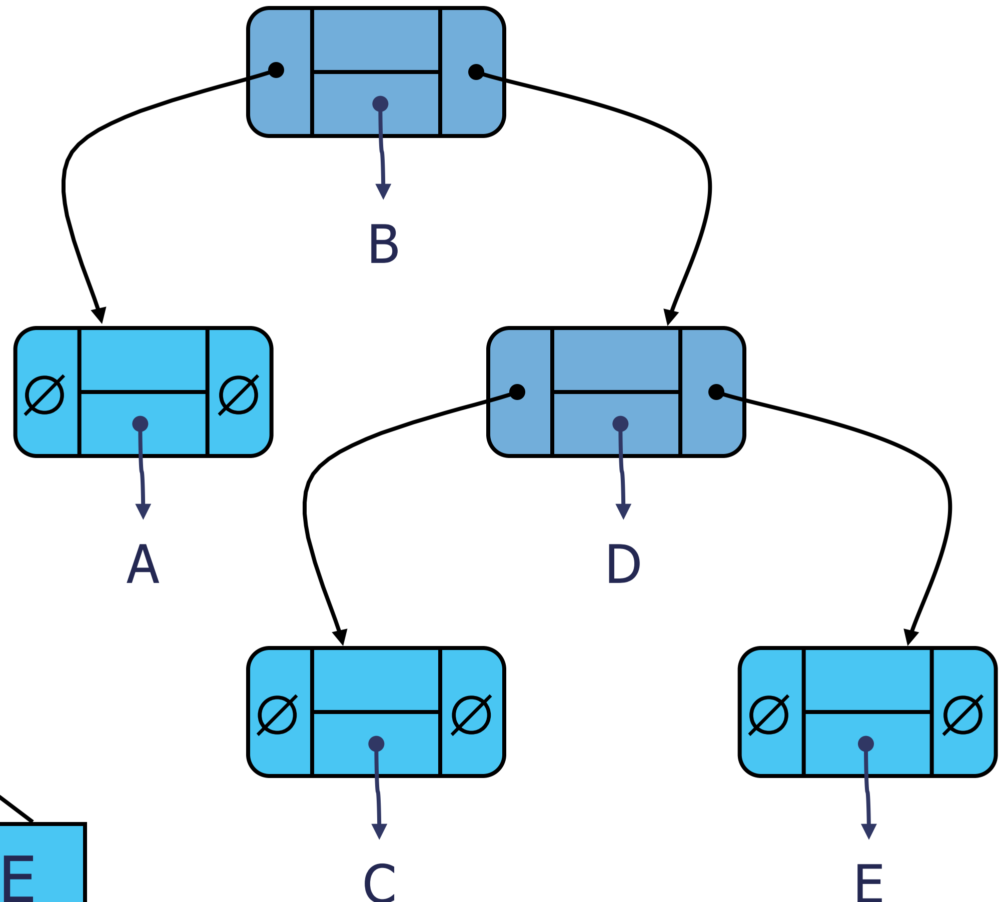
Applications:

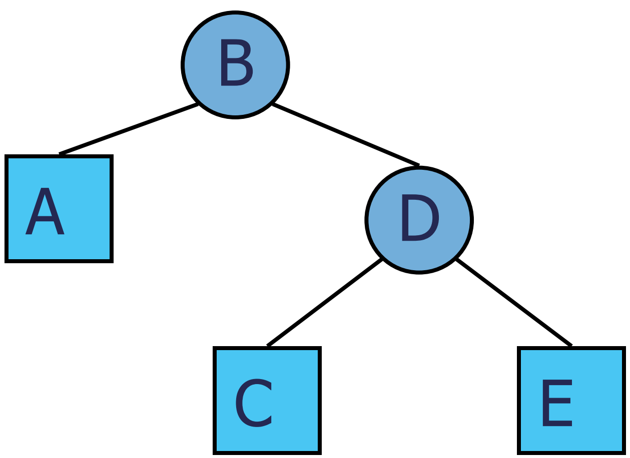
- arithmetic expressions
- decision processes
- searching



# Data Structure for Binary Trees

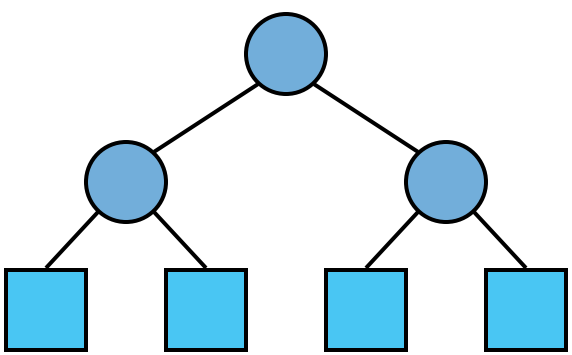
- A node is represented by an object storing
    - Element
    - Left child node
    - Right child node

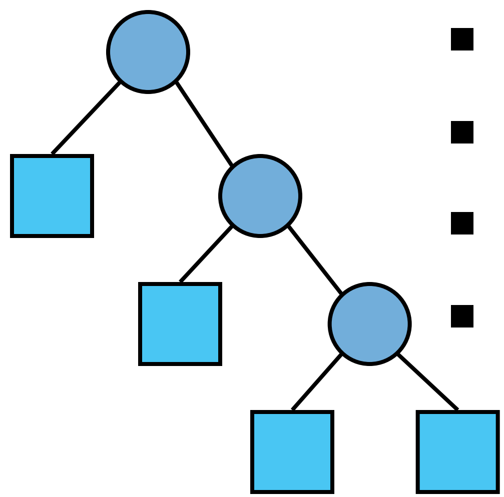


# Properties of Binary Trees

- Notation
  - $n$ number of nodes
  - $e$ number of external nodes
  - $i$ number of internal nodes
  - $h$ height



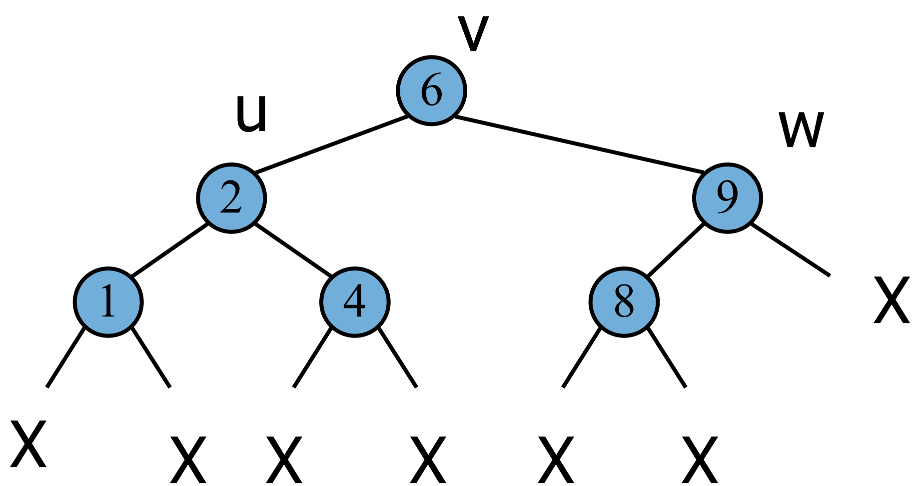


Properties:

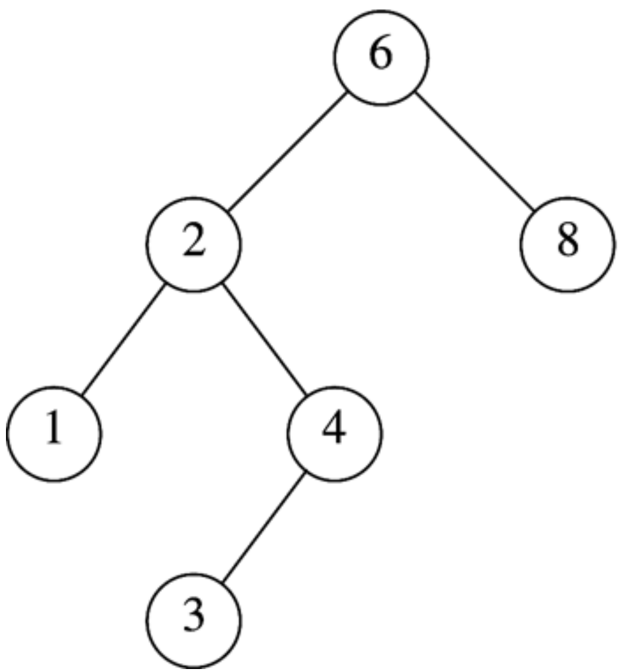
- $e \leq i + 1$
- $n \leq 2e - 1$
- $h \leq i$
- $h \leq (n - 1)/2$
- $e \leq 2^h$
- $h \geq \log_2 e$
- $h \geq \log_2 (n + 1) - 1$

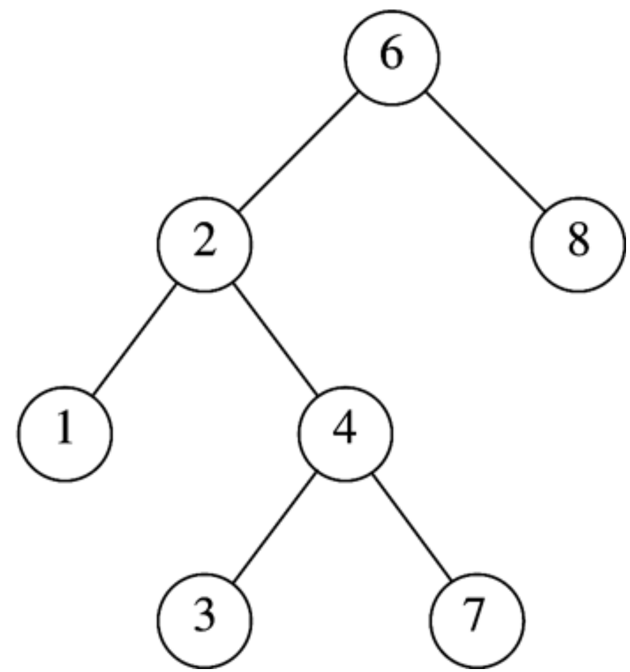
# Binary Search Tree

- A binary search tree is a binary tree storing keys (or key-element pairs) satisfying the following property:
  - Let ***u***, ***v***, and ***w*** be three nodes such that ***u*** is in the left subtree of ***v*** and ***w*** is in the right subtree of ***v***.
  - ***key***(***u***) < ***key***(***v***) < ***key***(***w***)



# Binary Search Tree?



# Preorder Traversal

```
void preorder(Node curr)
        if ( curr != null )
                print curr.value
                preorder (curr.left)
                preorder (curr.right)
```

# Postorder Traversal

```
void postorder(Node curr)
	if ( curr != null)
		postorder (curr.left)
		postorder (curr.right)
		print curr.value
```

# Inorder Traversal

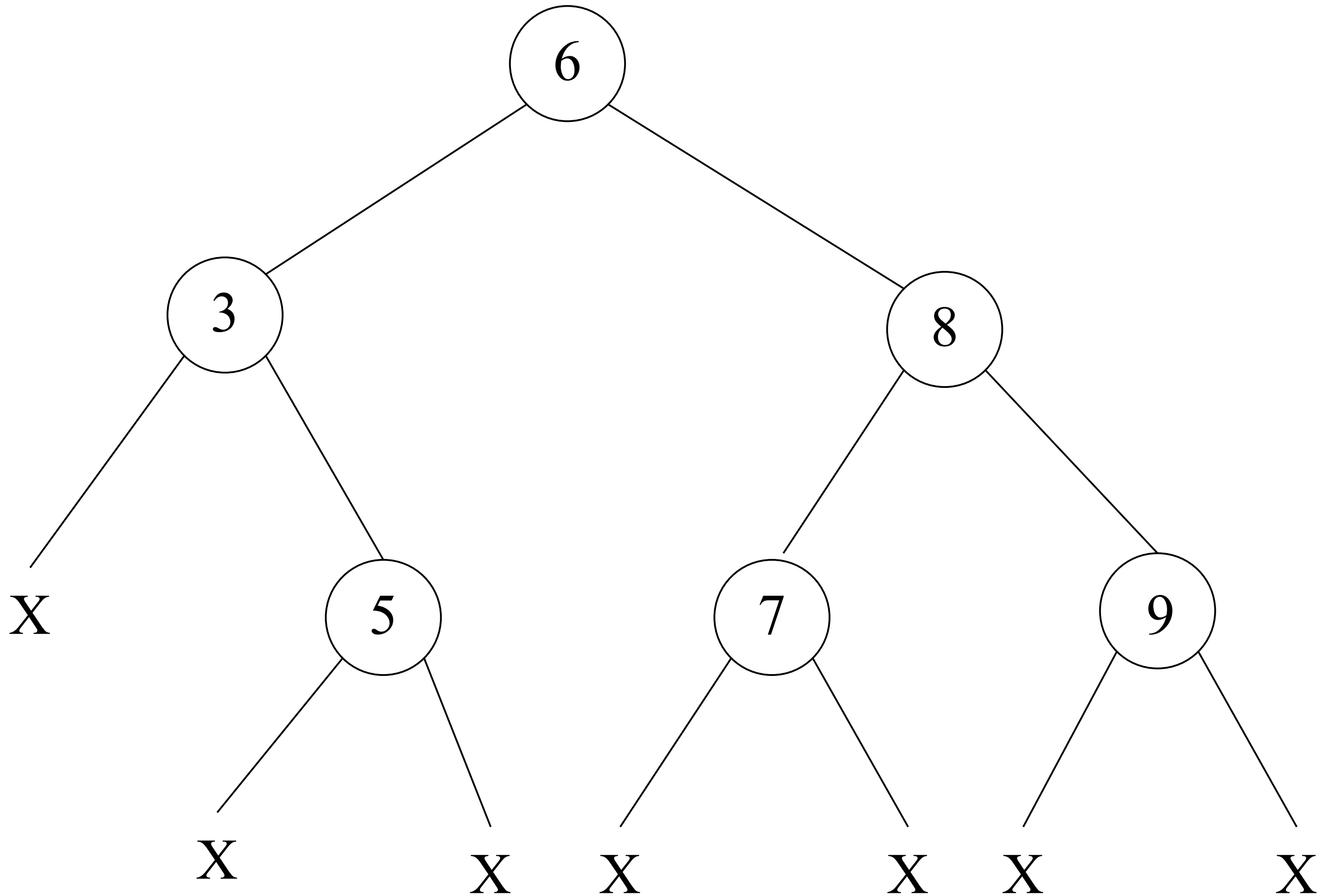
```
void inorder(Node curr)
        if ( curr != null)
                inorder (curr.left)
                print curr.value
                inorder (curr.right)
```

# Inorder Traversal – C++

```
void inorder(Node* curr)
        if ( curr )
                inorder (curr->left)
                print curr->value
                inorder (curr->right)
```

# Inorder Traversal

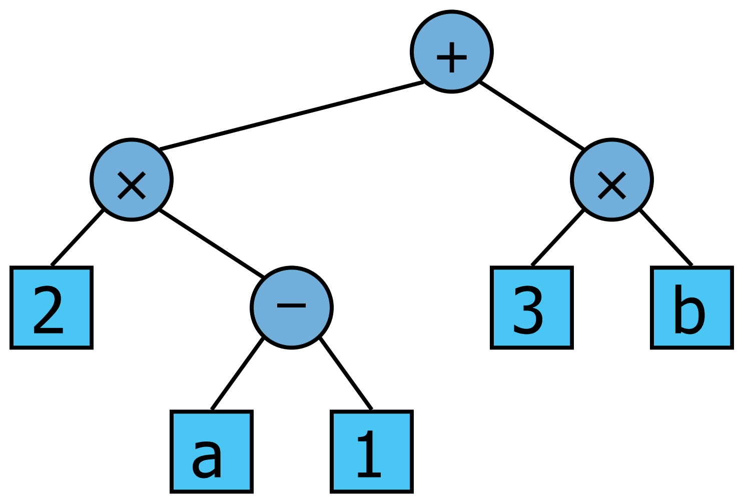
**3 5 6 7 8 9**



# Arithmetic Expression Tree

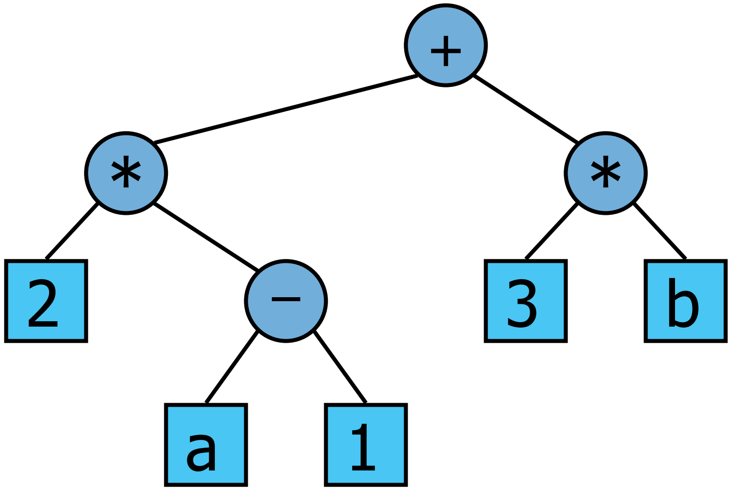
- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression:

$$(2 \times (a - 1) + (3 \times b))$$



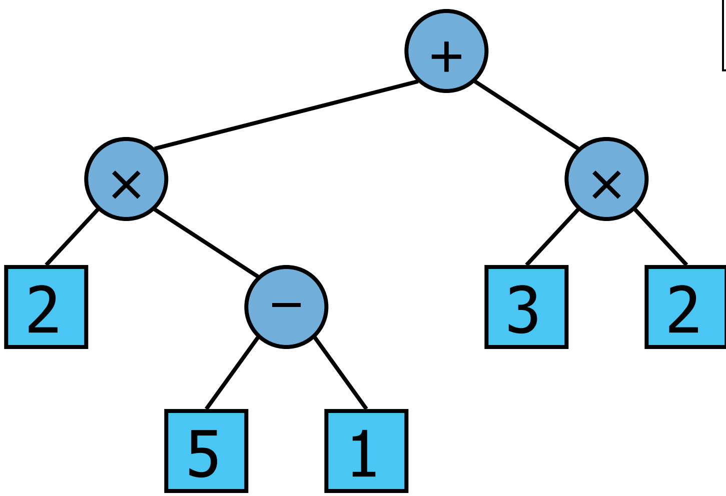
# Arithmetic Expression Tree

- In class exercise - give postfix notation by doing postorder traversal



# Evaluate Arithmetic Expressions

- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees




**Algorithm** *evalExpr*($v$)
    **if** *isExternal* ($v$)
        **return** *v.element* ()
    **else**
        $x \leftarrow$ *evalExpr*(*leftChild* ($v$))
        $y \leftarrow$ *evalExpr*(*rightChild* ($v$))
        $\Diamond \leftarrow$ operator stored at $v$
        **return** $x \Diamond y$

# Creating an Expression Tree

- Given an infix expression, use the stack based algorithm to convert infix to postfix
- Convert postfix expression to a tree

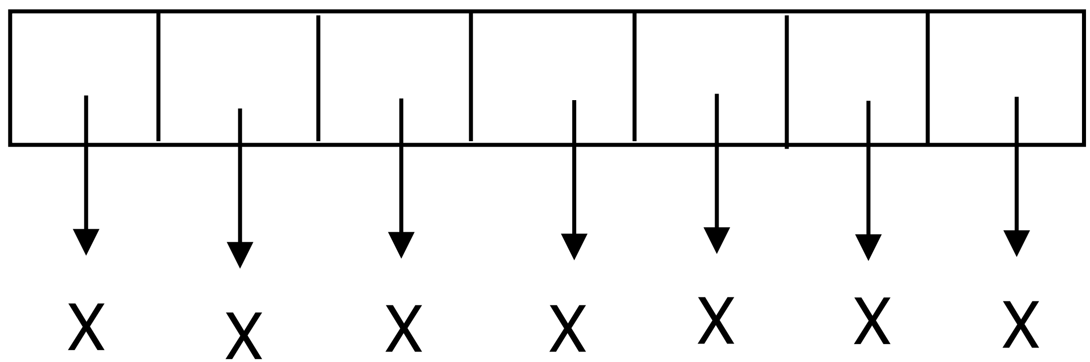
# Creating an Expression Tree

- Algorithm
  - Make a stack of node pointers
  - Operands - push a new tree onto the stack
  - Operators - pop two trees from the stack. Use the operator as the root of a new tree with the popped trees as children. Push a new tree onto the stack

# Creating an Expression Tree

(a + b) * (c * (d + e)) —> ab+cde+**

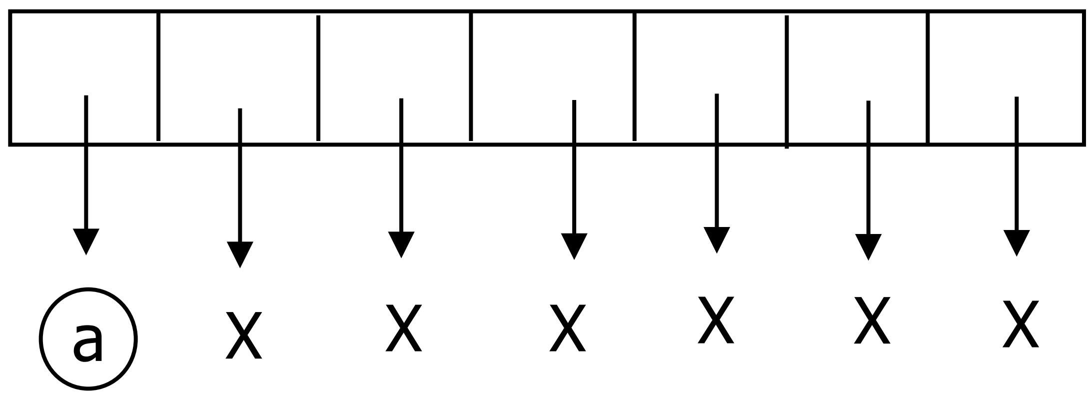
Stack of nodes:



# Creating an Expression Tree

(a + b) * (c * (d + e)) —> ab+cde+**

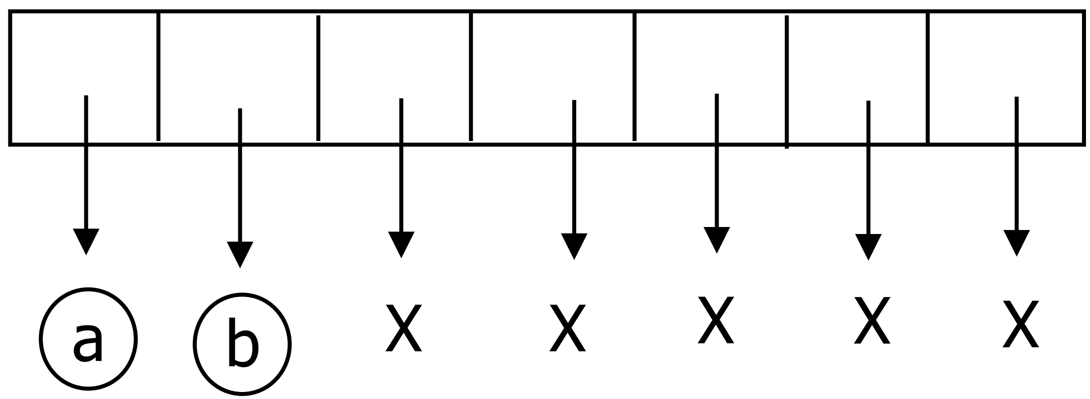
Stack of nodes:



# Creating an Expression Tree

(a + b) * (c * (d + e)) —> ab+cde+**

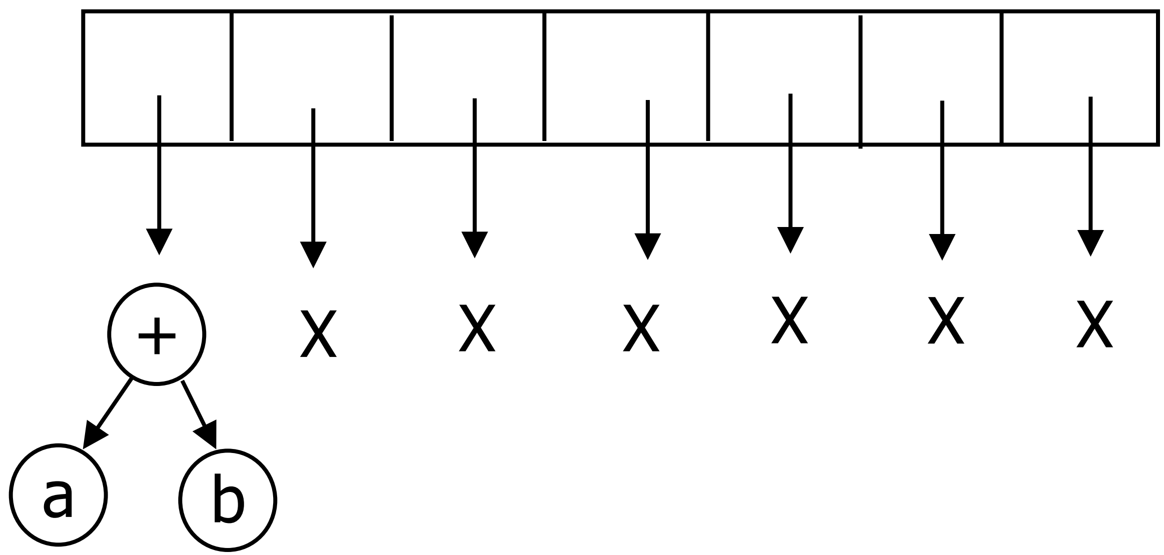
Stack of nodes:



# Creating an Expression Tree

(a + b) * (c * (d + e)) —> ab+cde+**

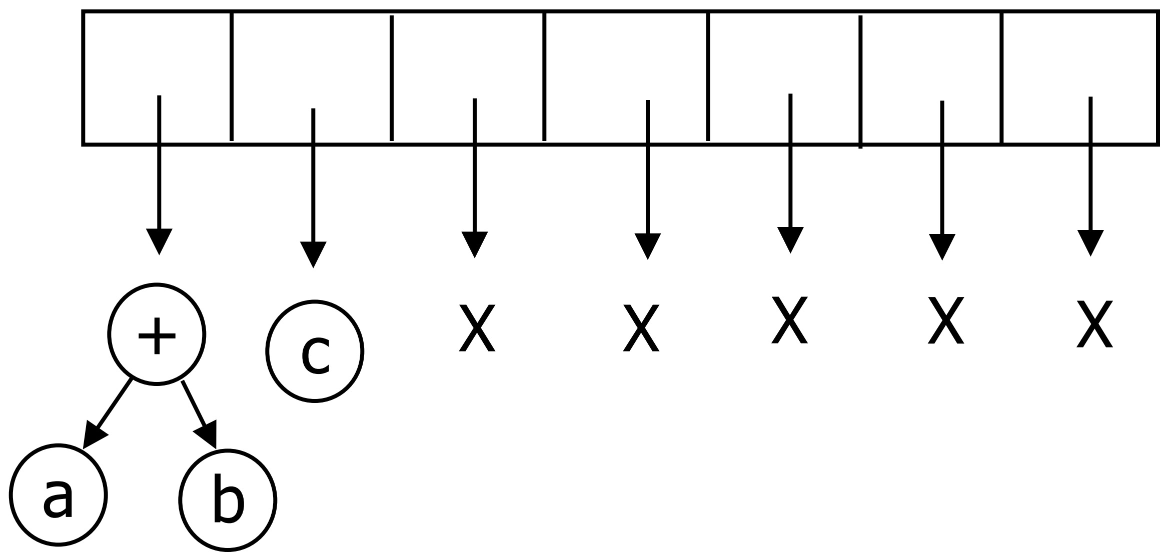
Stack of nodes:



# Creating an Expression Tree

(a + b) * (c * (d + e)) —> ab+cde+**

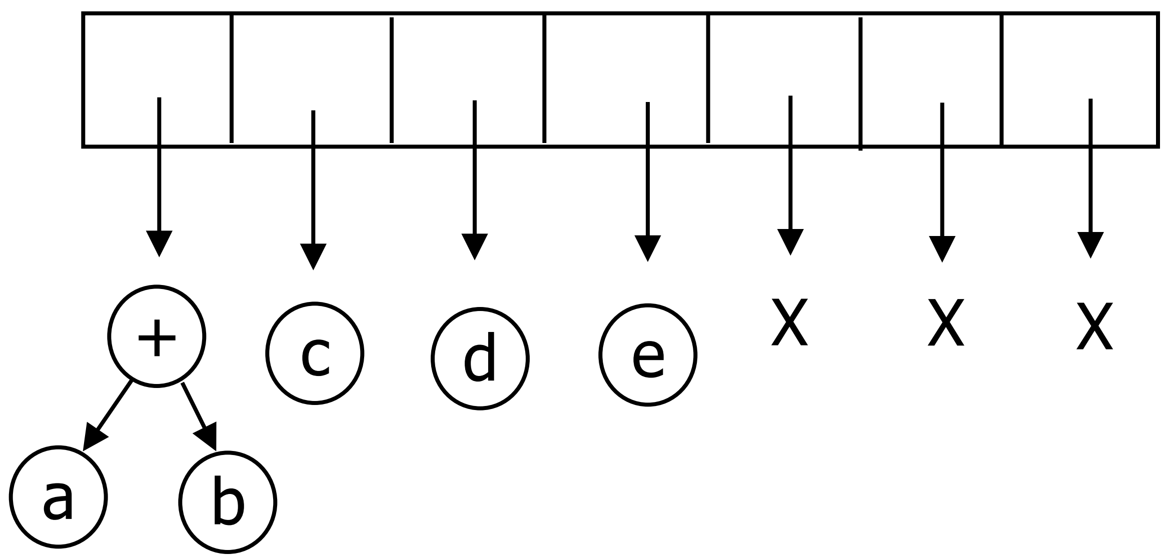
Stack of nodes:



# Creating an Expression Tree

(a + b) * (c * (d + e)) —> ab+cde+**

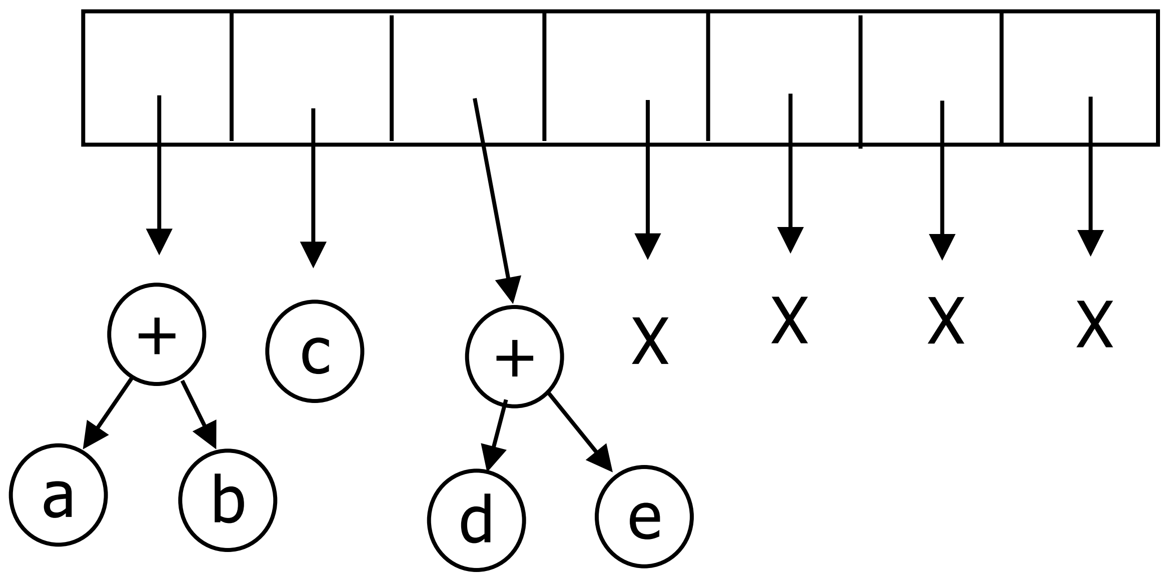
Stack of nodes:



# Creating an Expression Tree

(a + b) * (c * (d + e)) —> ab+cde+**

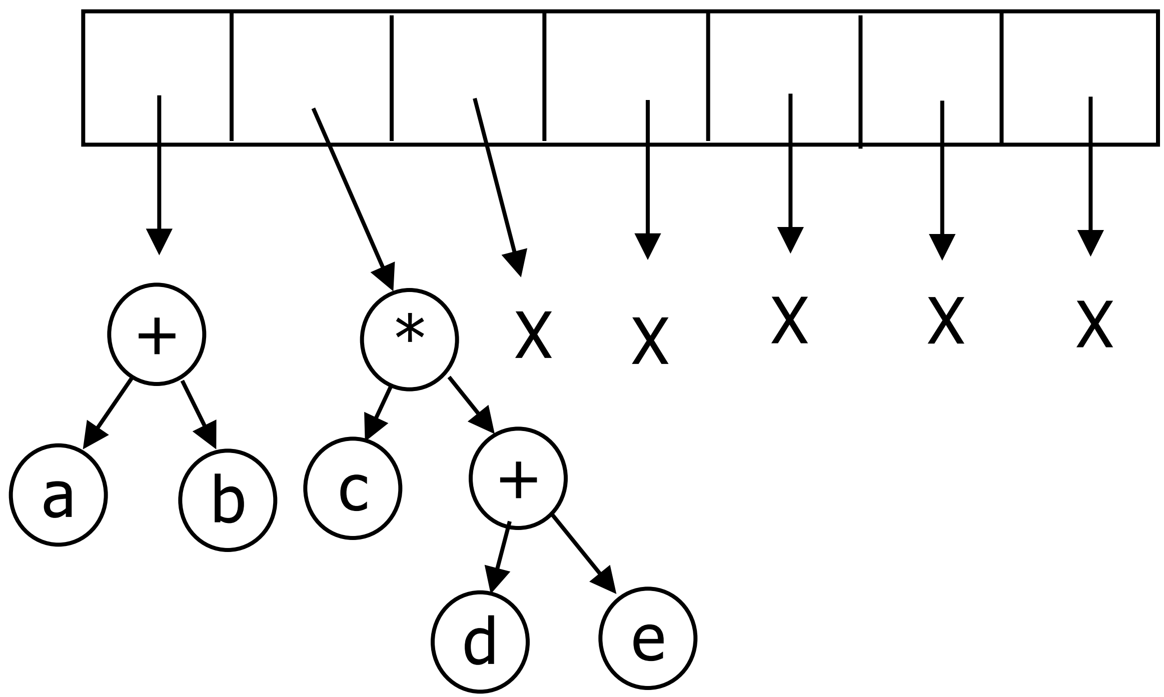
Stack of nodes:



# Creating an Expression Tree

(a + b) * (c * (d + e)) —> ab+cde+**

Stack of nodes:



# Creating an Expression Tree

(a + b) * (c * (d + e)) —> ab+cde+**

Stack of nodes: