Trees

- Linked lists:

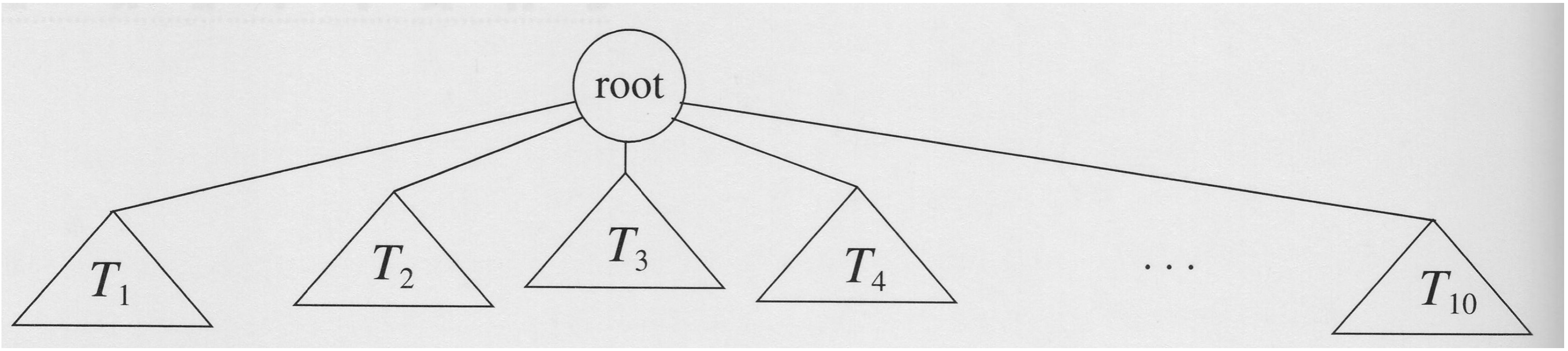
  linear access time to a node => O(N)

- Need a data structure for which the running time of most operations is O(log N) in average.

  => Binary Search Trees (BST)

Trees: Preliminaries

- Tree: collection of nodes
- A nonempty tree consists of:
  - a root node: *r*
  - zero or more nonempty (sub)trees $T_1, T_2, \ldots, T_k$ each of whose roots are connected by an edge from *r*.
- *r* is the parent of each subtree root.
- Each subtree is a child of *r*.

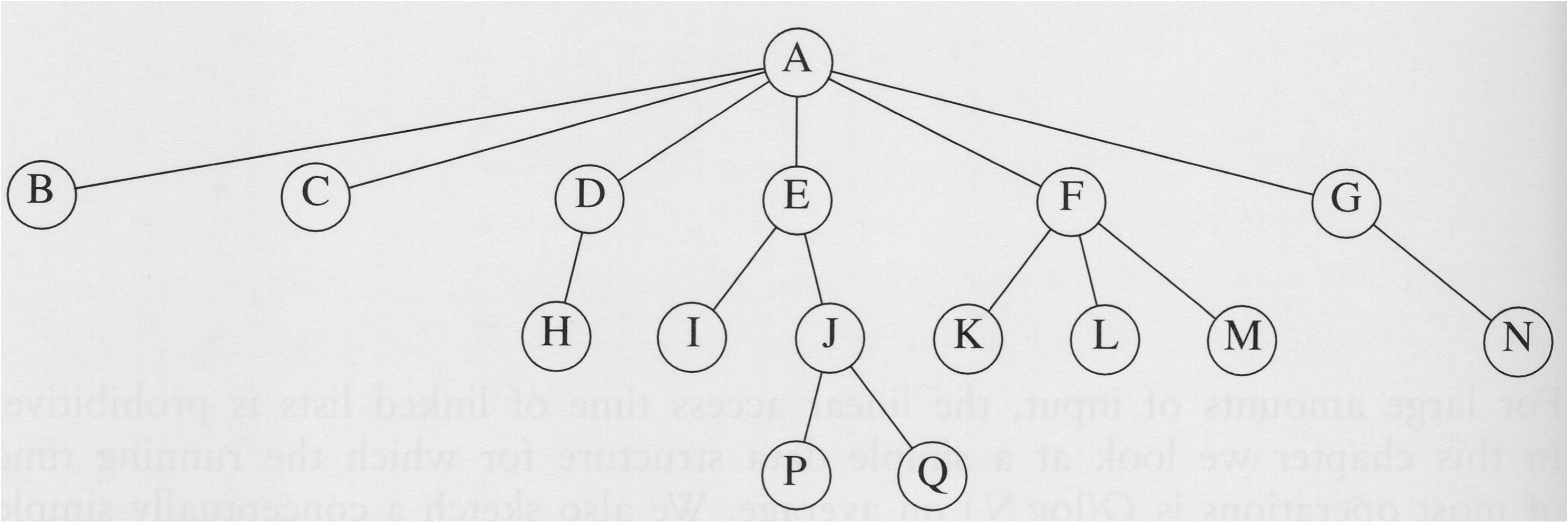


Trees: Preliminaries

- N node tree => N-1 edges

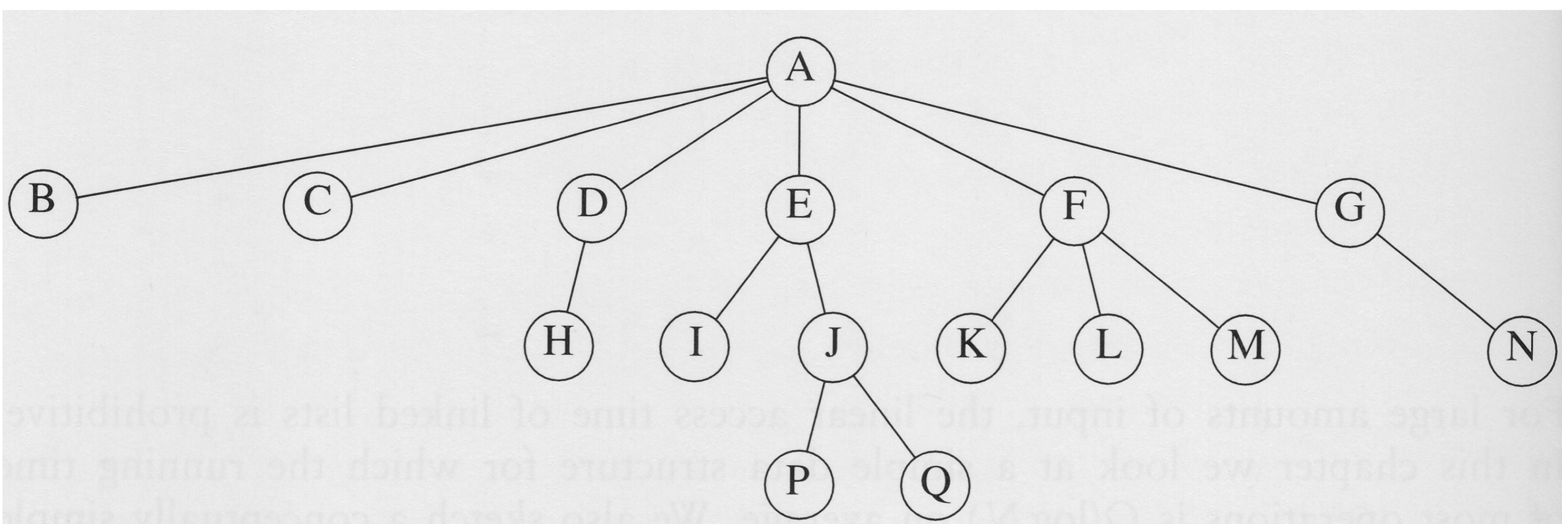
  Proof: each edge connects some node to its parent, every node except the root has one parent.

- Leaves: nodes with no children
- Siblings: nodes with the same parent



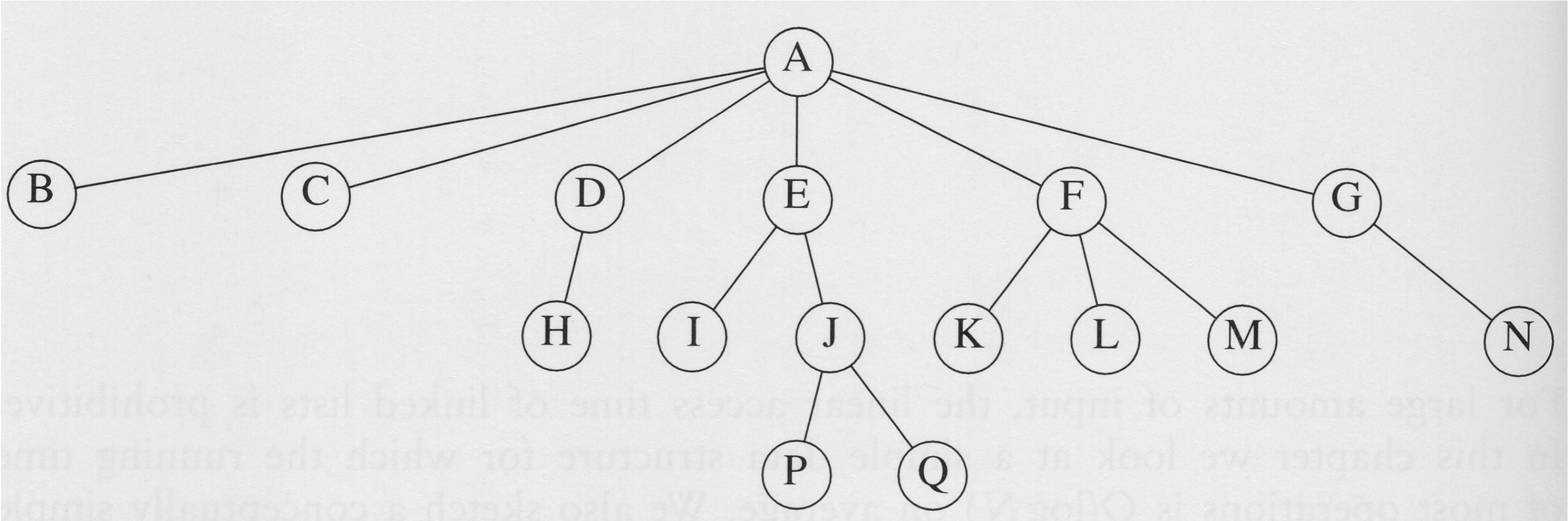
Trees: Preliminaries

- Path from n_1 to n_k: a sequence of nodes n_1, n_2, …, n_k such that n_i is the parent of n_{i+1}, $1 <= i < k$.
- In a tree there is exactly one path from the root to each node.
- Path length = number of edges on the path



Trees: Preliminaries

- Depth of node n_i = length of the longest path from the root to n_i.
- Depth of root = 0
- Height of n_i = length of the longest path from n_i to a leaf.
- Height of all leaves = 0
- Height of the tree = height of the root
- n_1 is an ancestor of n_2 and n_2 is a descendant of n_1 if there is a path from n_1 to n_2.
- If $n_1 \neq n_2$ then n_1 is a proper ancestor of n_2 and n_2 is a proper descendant of n_1 .

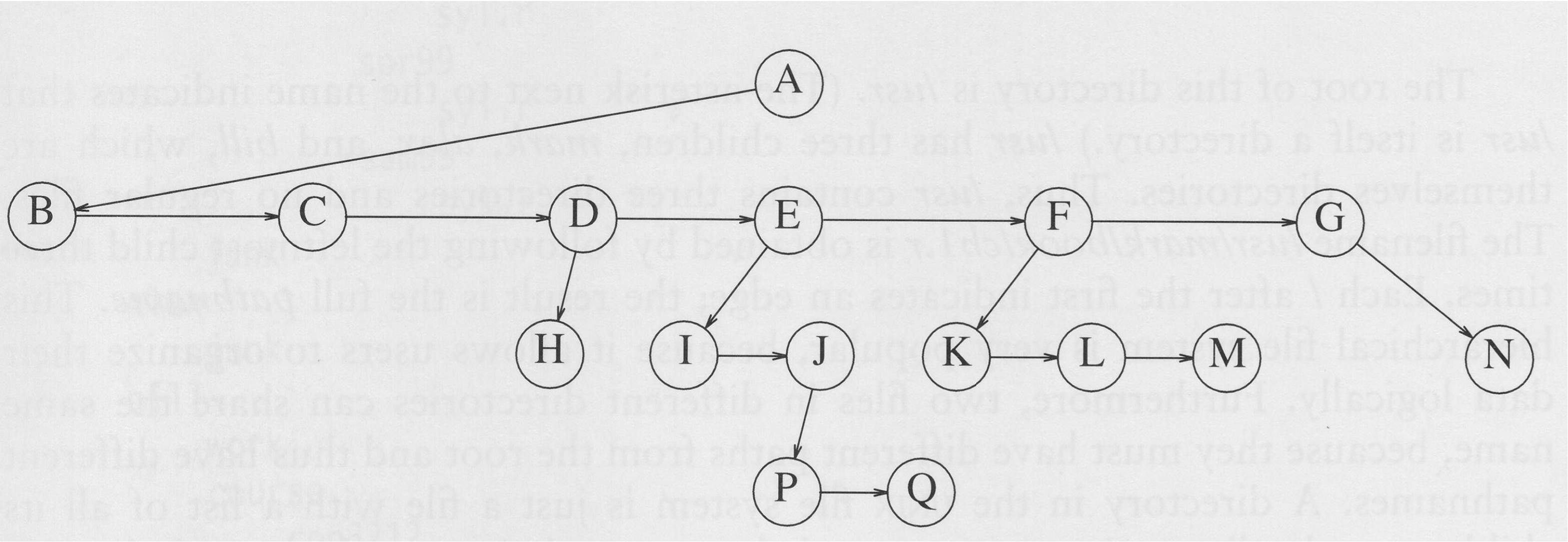


Implementation of Trees

- Implementation 1: have in each node besides its data a link to each child of the node.

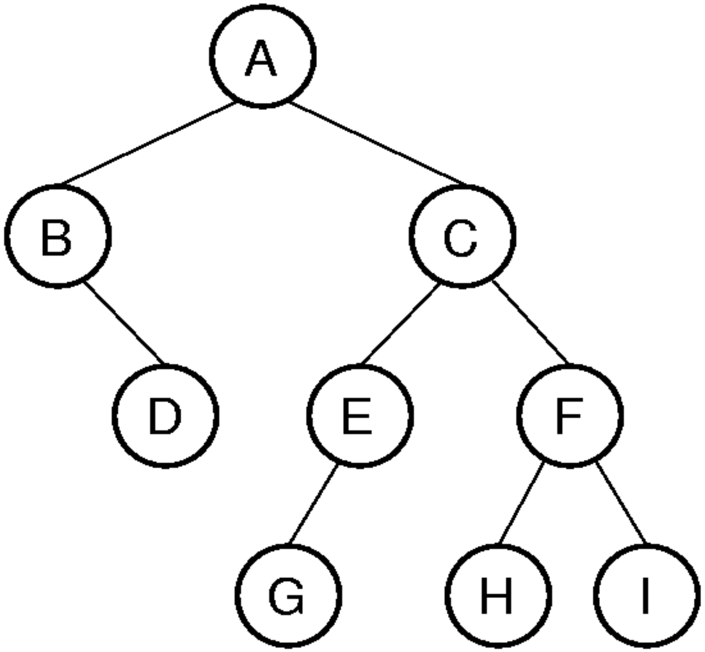
  => not efficient, waste of space if variable number of children per node
- Implementation 2: keep children of each node in a linked list of tree nodes.

```
struct TreeNode
{
        Object element;
        TreeNode *firstChild;
        TreeNode *nextSibling;
};
```



Preorder Traversal

- Preorder traversal: Visit each node before visiting its children.
- => A B D C E G F H I



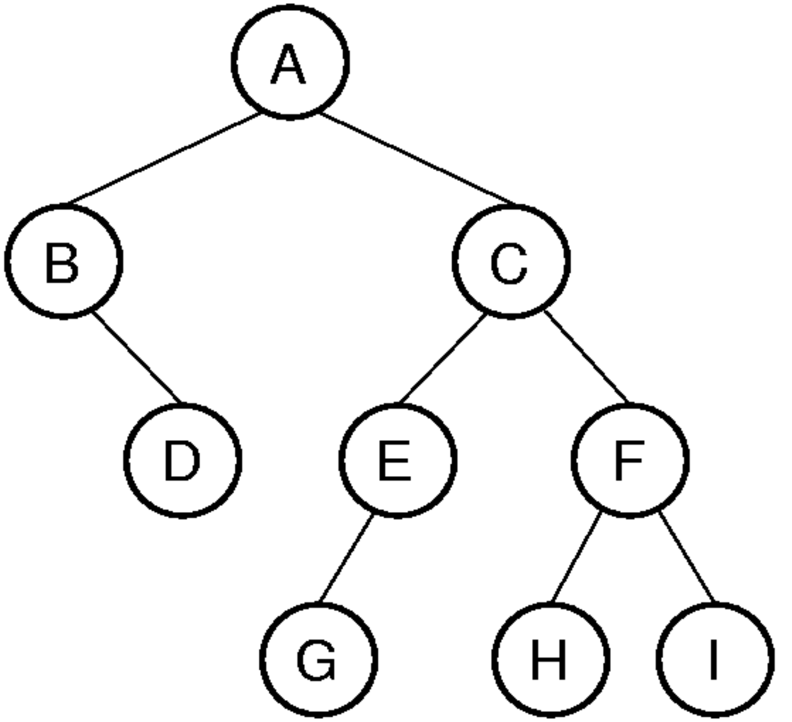

```
template <class Elem>
void preorder(BinNode<Elem>*
    subroot)
{
  if (subroot == NULL) return;
    // Empty

  visit(subroot);
  preorder(subroot->left());
  preorder(subroot->right());
}
```

- **Example: list a directory in a hierarchical file system**

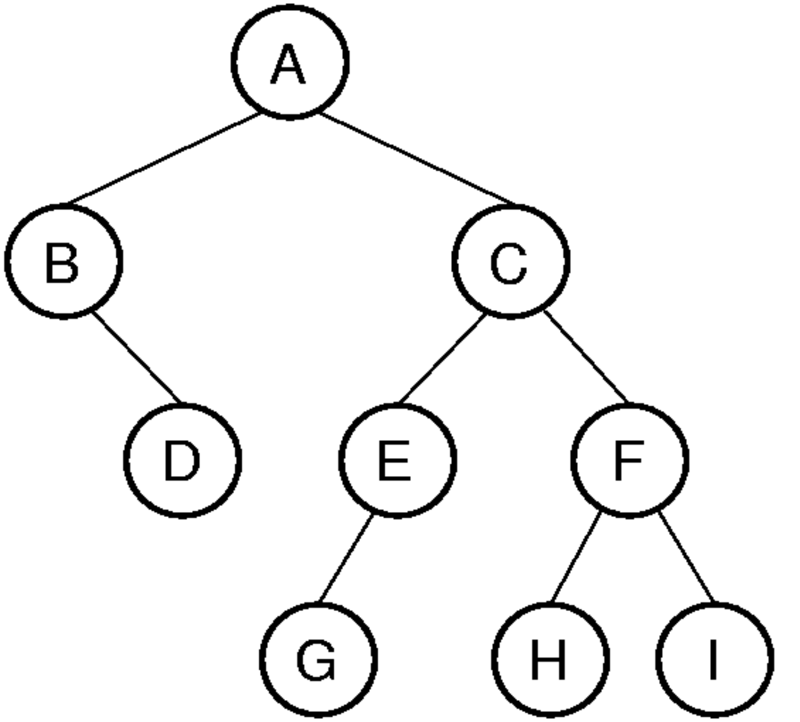
Postorder Traversals

- Postorder traversal: Visit each node after visiting its children.
- => D B G E H I F C A
- Example: calculate the size of a directory



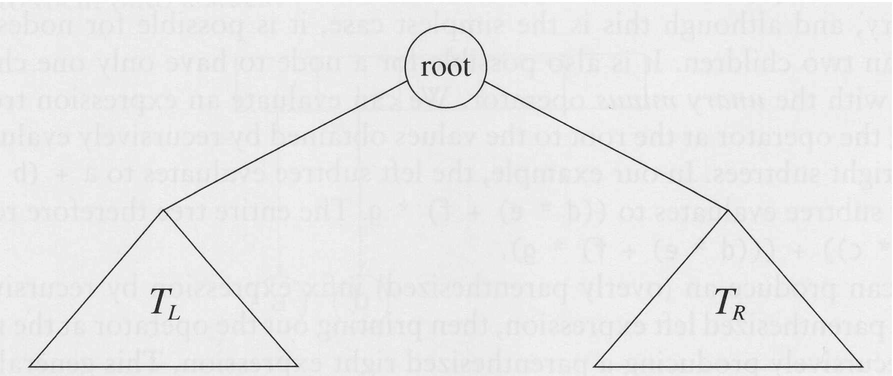
Inorder Traversals

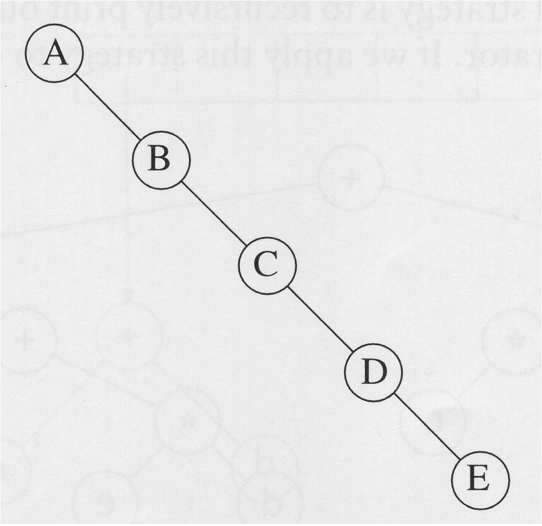
- Inorder traversal: Visit the left subtree, then the node, then the right subtree.
- => B D A G E C H F I



BinaryTrees

- Binary Tree: no node can have more than two children.
- Properties:
  - Average depth of binary tree is $O(N^{1/2})$
  - Average depth of a Binary Search Tree is $O(\log N)$.
- Maximum depth = N-1



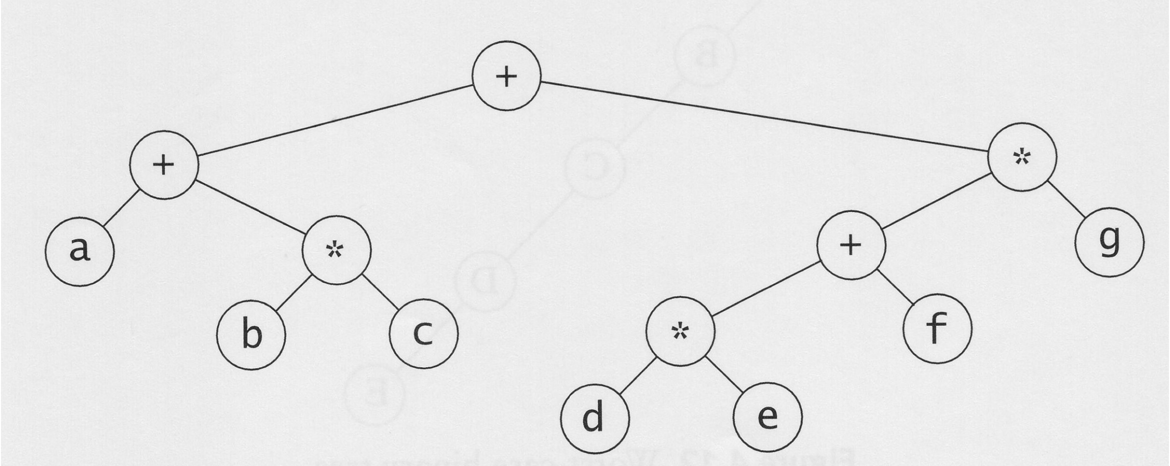


Binary Trees: Implementation

```
struct BinaryNode
  {
        Object element;
        BinaryNode *left;
        BinaryNode *right;
  };
```

Example: Expression Trees

- Expression tree:
  leaves <-> operands;
  internal nodes <-> operators
- Example: (a + b * c) + ((d * e + f) * g)
- Inorder traversal => infix representation
- Postorder traversal => postfix representation:
  a b c * + d e * f + g * +
- Preorder traversal => prefix representation:
  + + a * b c * + * d e f g

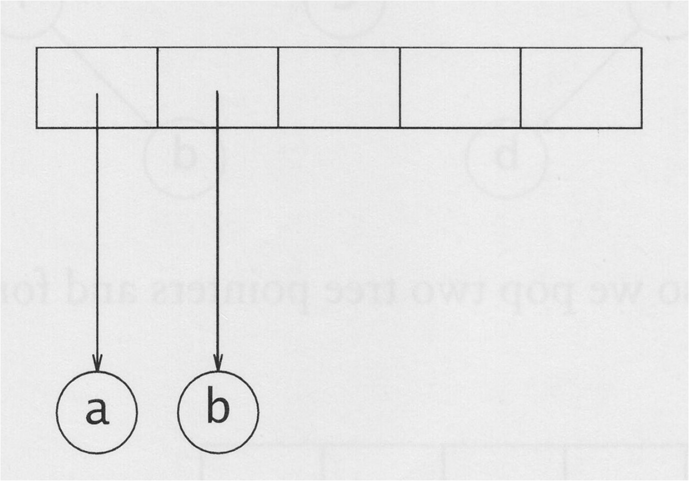


Constructing an Expression Tree

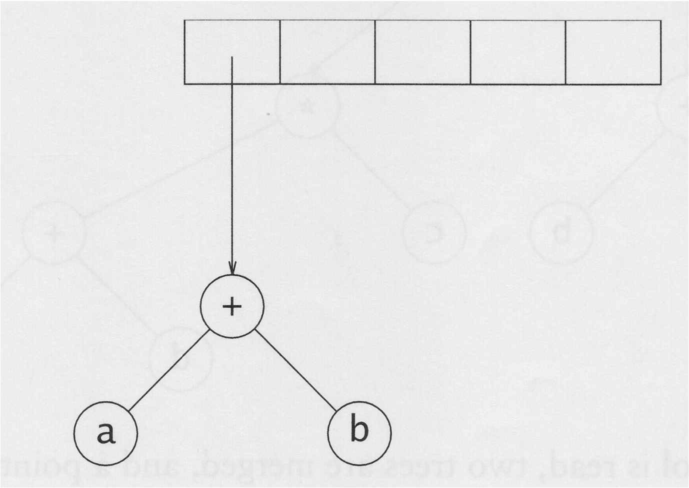
- Problem: Convert a postfix expression into an expression tree
- Idea:
  - Read one symbol at the time
  - If (operand) then create a one-node tree and push a pointer into a stack
  - If (operator) then pop two pointers from stack T1, T2 and form a new tree (root = operator; left = T1 and right =T2). A pointer to root is pushed into the stack.

Constructing an Expression Tree

a b + c d e + * *

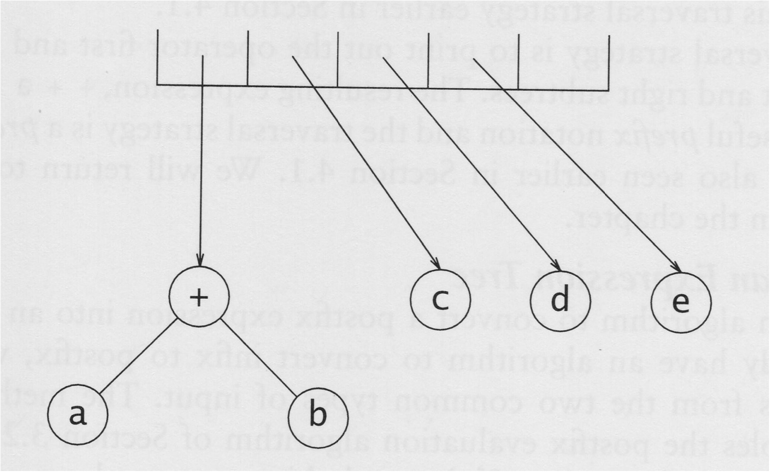


a b + c d e + * *

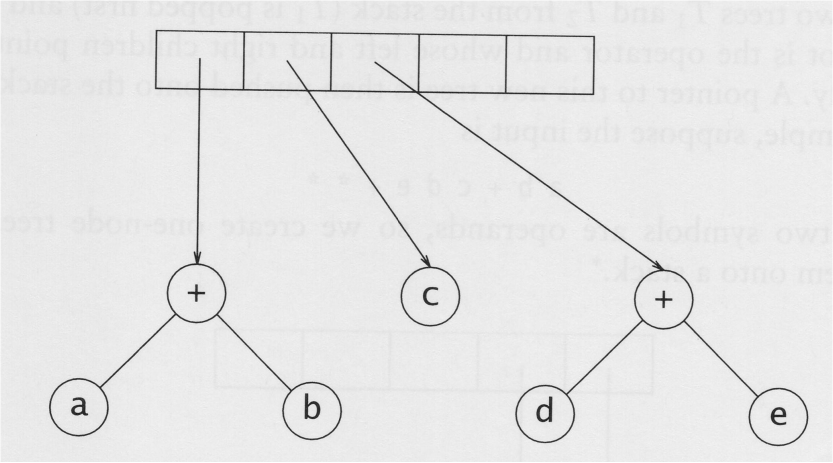


Constructing an Expression Tree

a b + c d e + * *

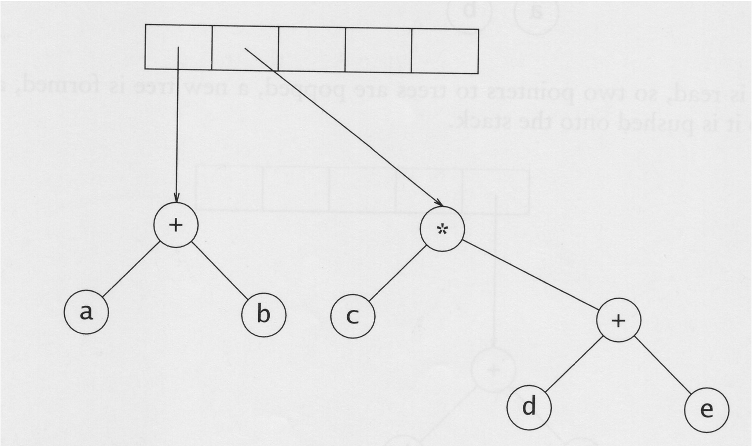


a b + c d e + * *



Constructing an Expression Tree

a b + c d e + * *



a b + c d e + * *