Data Structures

17. Binary Tree Implementation

Tree ADT

- Data Type: Any type of objects can be stored in a tree
- Accessor methods
  - `root()` – return the root of the tree
  - `parent(p)` – return the parent of a node
  - `children(p)` – return the children of a node
- Query methods
  - `size()` – return the number of nodes in the tree
  - `isEmpty()` – return true if the tree is empty
  - `elements()` – return all elements
  - `isRoot(p)` – return true if node p is the root
- Other methods
  - Tree traversal, Node addition/deletion, create/destroy

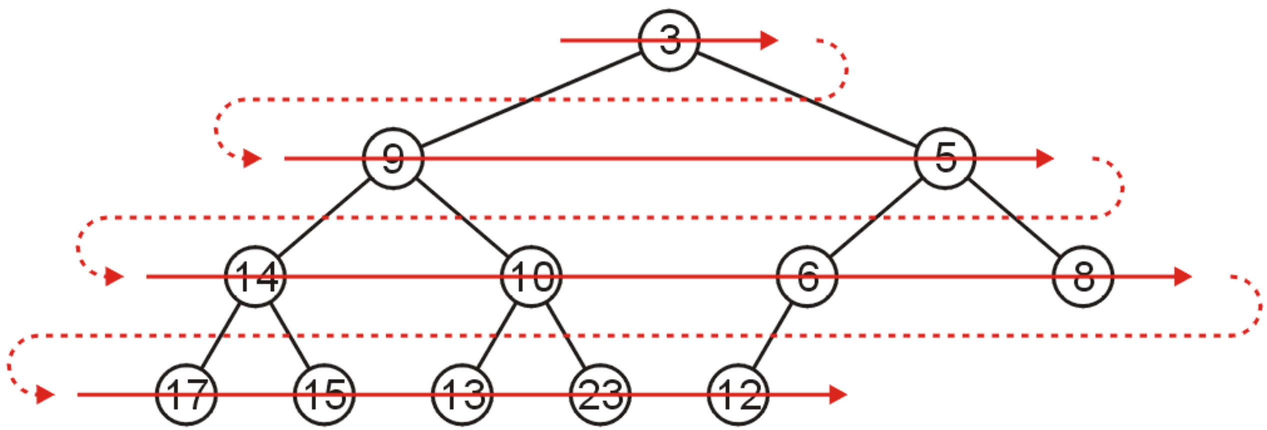
Binary Tree Storage

- Contiguous storage
- Linked list based storage

Contiguous Storage

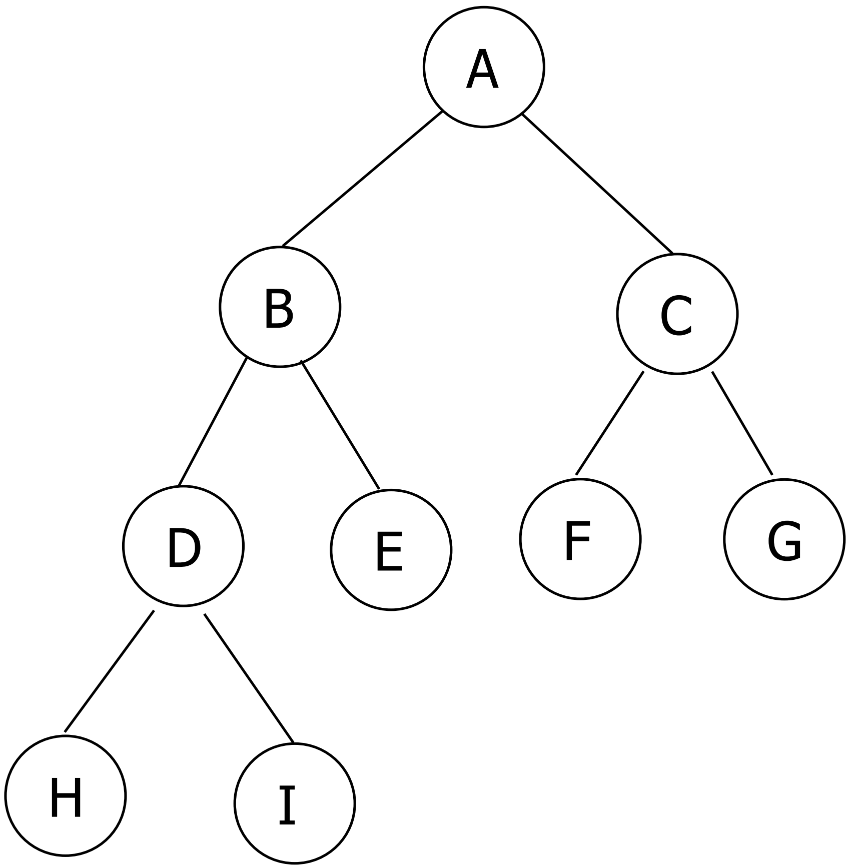
Array Storage (1)

- We can store a binary tree as an array
- Traverse tree in breadth-first order, placing the entries into array
  - Storage of elements (i.e., objects/data) starts from root node
  - Nodes at each level of the tree are stored left to right



	3	9	5	14	10	6	8	17	15	13	23	12				

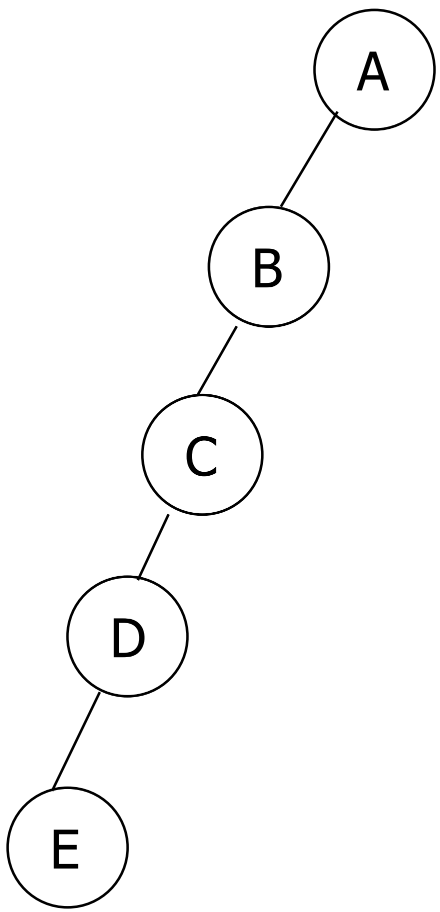
Array Storage Example (1)



[1]	A
[2]	B
[3]	C
[4]	D
[5]	E
[6]	F
[7]	G
[8]	H
[9]	I

Array Storage Example (2)

- Unused nodes in tree represented by a predefined bit pattern



[1]	A
[2]	B
[3]	-
[4]	C
[5]	-
[6]	-
[7]	-
[8]	D
[9]	-
…	…
[16]	E

Exercise

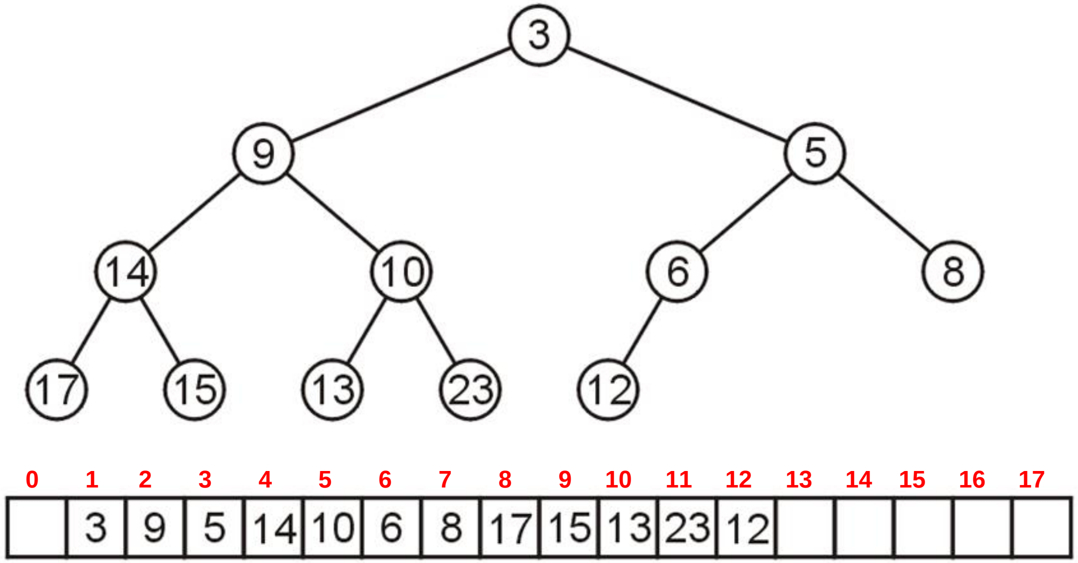
- If the following is stored in an array, what does the tree look like? (Draw it)
- ABCDE-F

- Solution:

```
        A(0)
       /    \
    B(1)  C(2)
   /    \       \
D(3)  E(4)    F(6)
```

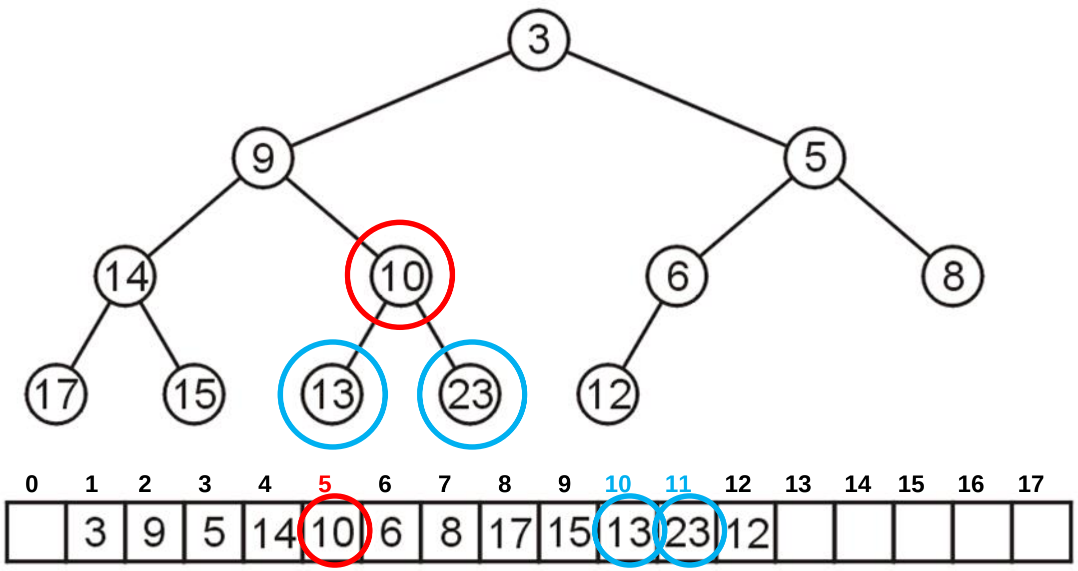
Array Storage (3)

- The children of the node with index `k` are in `2k` and `2k + 1`
- The parent of node with index `k` is in `k ÷ 2`



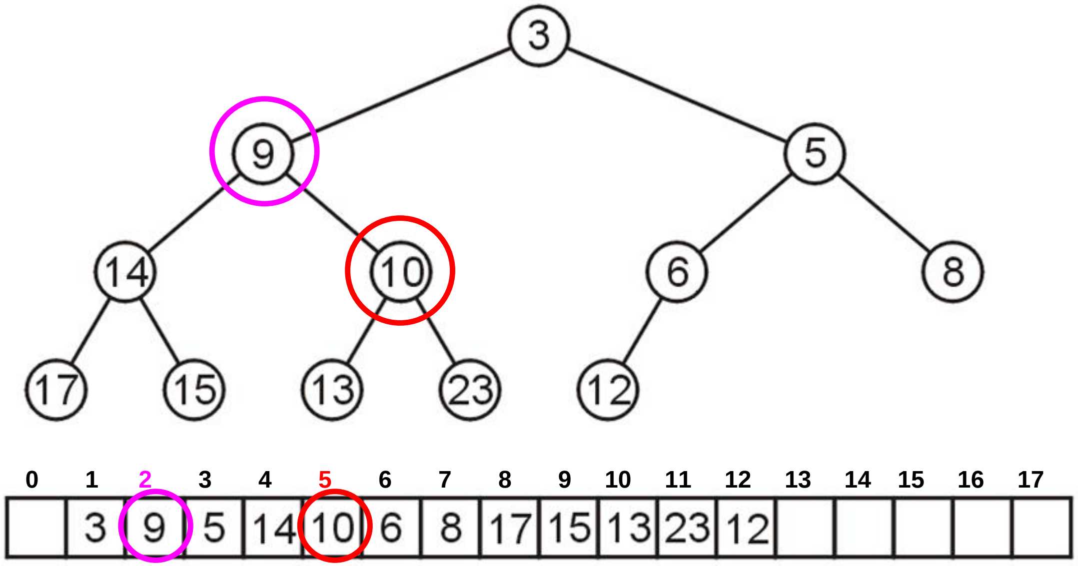
Array Storage Example (3)

- Node 10 has index 5
  - Its children 13 and 23 have indices 10 and 11, respectively



Array Storage Example (4)

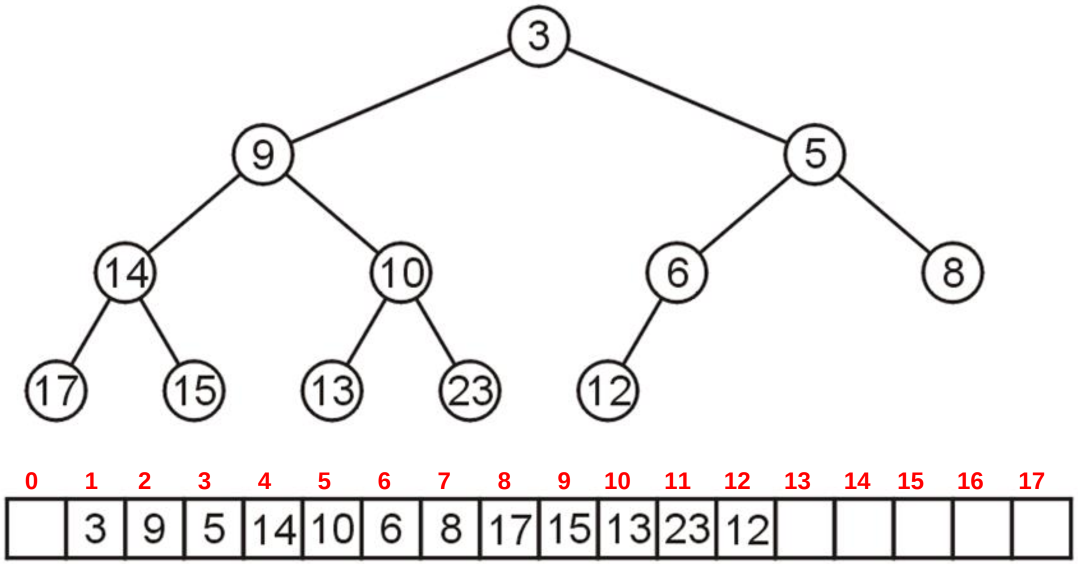
- Node 10 has index 5
  - Its children 13 and 23 have indices 10 and 11, respectively
  - Its parent is node 9 with index 5/2 = 2



Array Storage (4)

- Why array index is not started from 0
  - In C++, this simplifies the calculations

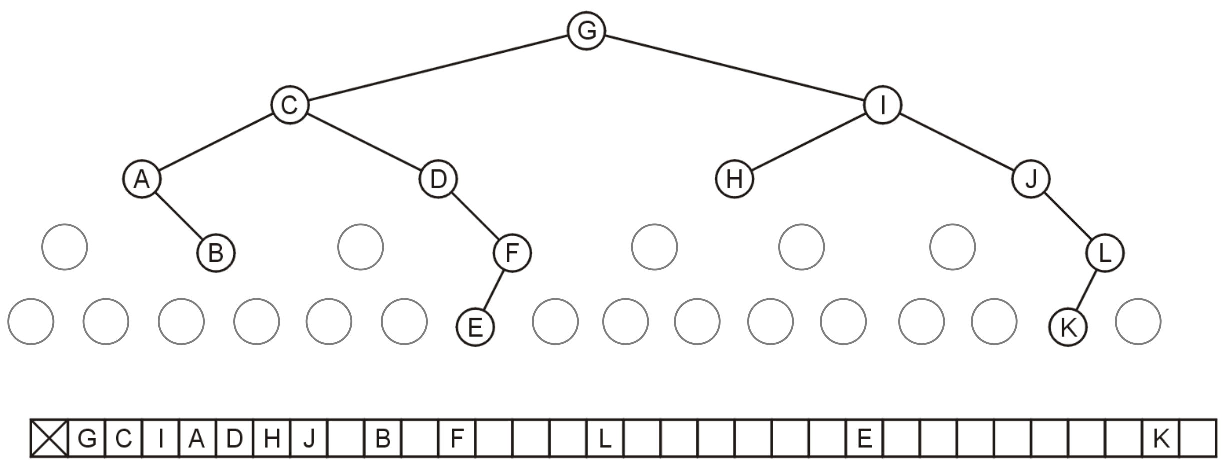
```
parent = k >> 1;
left_child = k << 1;
right_child = left_child | 1;
```



Array Storage: Disadvantage

- Why not store any tree as an array using breadth-first traversals?
  - There is a significant potential for a lot of wasted memory

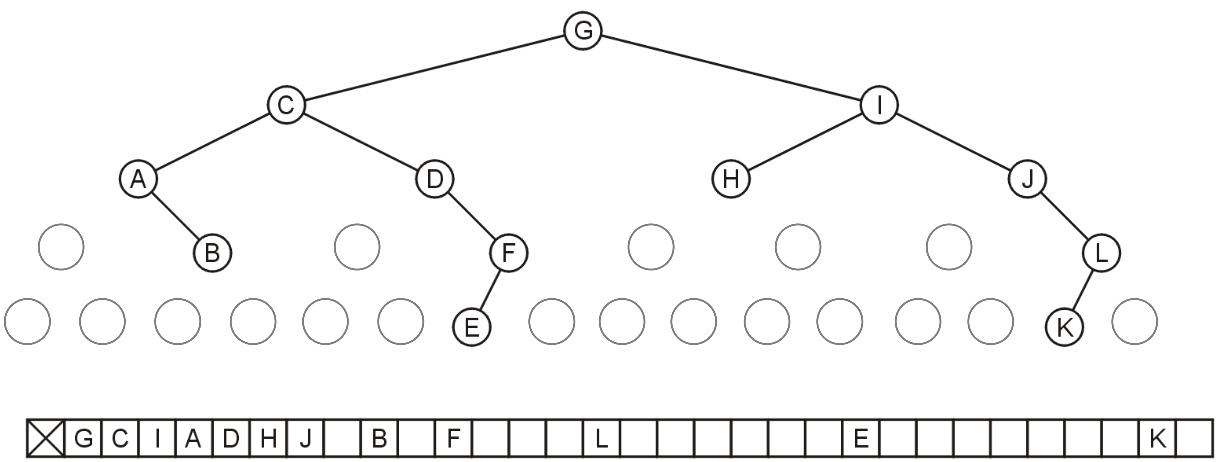
- Consider the following tree with 12 nodes
  - What is the required size of array?



Array Storage: Disadvantage

- Why not store any tree as an array using breadth-first traversals?
  - There is a significant potential for a lot of wasted memory

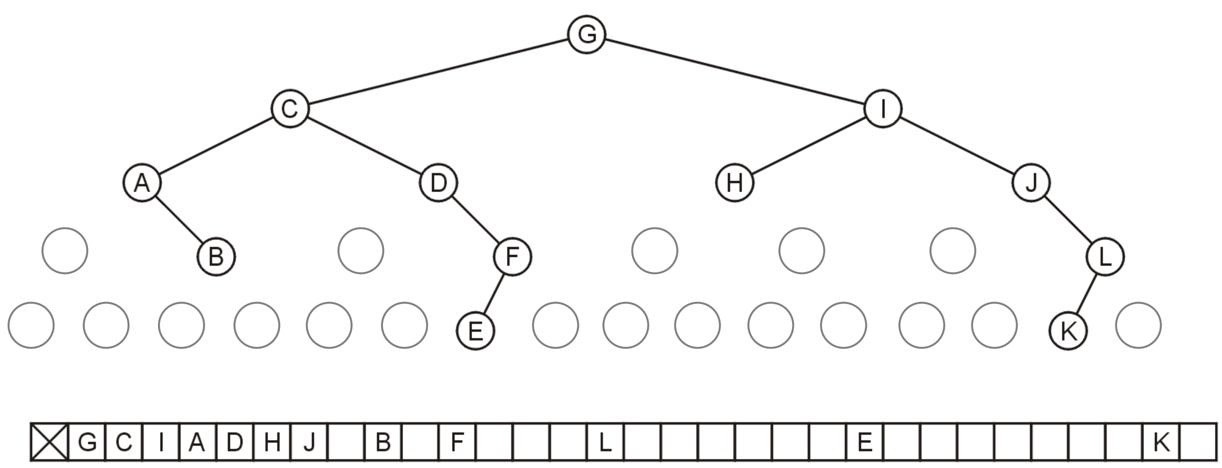
- Consider the following tree with 12 nodes
  - What is the required size of array? **32**
  - What will be the array size if a child is added to node K?



Array Storage: Disadvantage

- Why not store any tree as an array using breadth-first traversals?
  - There is a significant potential for a lot of wasted memory

- Consider the following tree with 12 nodes
  - What is the required size of array? **32**
  - What will be the array size if a child is added to node K? **double**



Implementation Exercise

- How to write code for implementing a tree in an array?
- One possible approach:
- Start with an empty array representing the size of the tree.
- First add the root node in tree[0] (or tree[1], if you want to use the shift operators).
- Then write a set_left and set_right function to add a left child and right child to the root node.
- Then keep making calls to set_left and set_right to set children of all subsequent nodes.

Solution

```
// C++ implementation of tree using array
// numbering starting from 0 to n-1.
#include<bits/stdc++.h>
using namespace std;
char tree[10];
int root(char key) {
if (tree[0] != '\0')
		cout << "Tree already had root";
else
		tree[0] = key;
return 0;
}

int set_left(char key, int parent) {
if (tree[parent] == '\0')
		cout << "\nCan't set child at "
		<< (parent * 2) + 1
		<< " , no parent found";
else
		tree[(parent * 2) + 1] = key;
return 0;
}

}
```

```
int set_right(char key, int parent) {
if (tree[parent] == '\0')
		cout << "\nCan't set child at "
		<< (parent * 2) + 2
		<< " , no parent found";
else
		tree[(parent * 2) + 2] = key;
return 0;
}

int print_tree() {
cout << "\n";
for (int i = 0; i < 10; i++) {
		if (tree[i] != '\0')
		cout << tree[i];
		else
		cout << "-";
}
return 0;
}

// Driver Code
int main() {
root('A');
set_left('B',0);
set_right('C', 0);
set_left('D', 1);
set_right('E', 1);
set_right('F', 2);
print_tree();
return 0;
```

```
         A(0)
        /   \
     B(1)   C(2)
    /   \       \
 D(3)   E(4)    F(6)
```

Practice exercise 1

- Write an alternate print function that prints the tree visually as a tree, something like this:

```
      A(0)
    /   \
   B(1)  C(2)
 /   \      \
D(3)  E(4)   F(6)
```

Practice Exercise 2

- Using the recursive definition of a perfect binary tree, determine, for a given array representing a tree, whether it is a perfect binary tree.

- Recursive definition:
  - A binary tree of height h = 0 is perfect.
  - A binary tree with height h > 0 is perfect
    - If both sub-trees are prefect binary trees of height h – 1

- Write a recursive function bool isPerfect(char*, int) which should take a char array representing the tree and return a Boolean value of true if the tree is perfect and false if it is not.

Linked List Storage

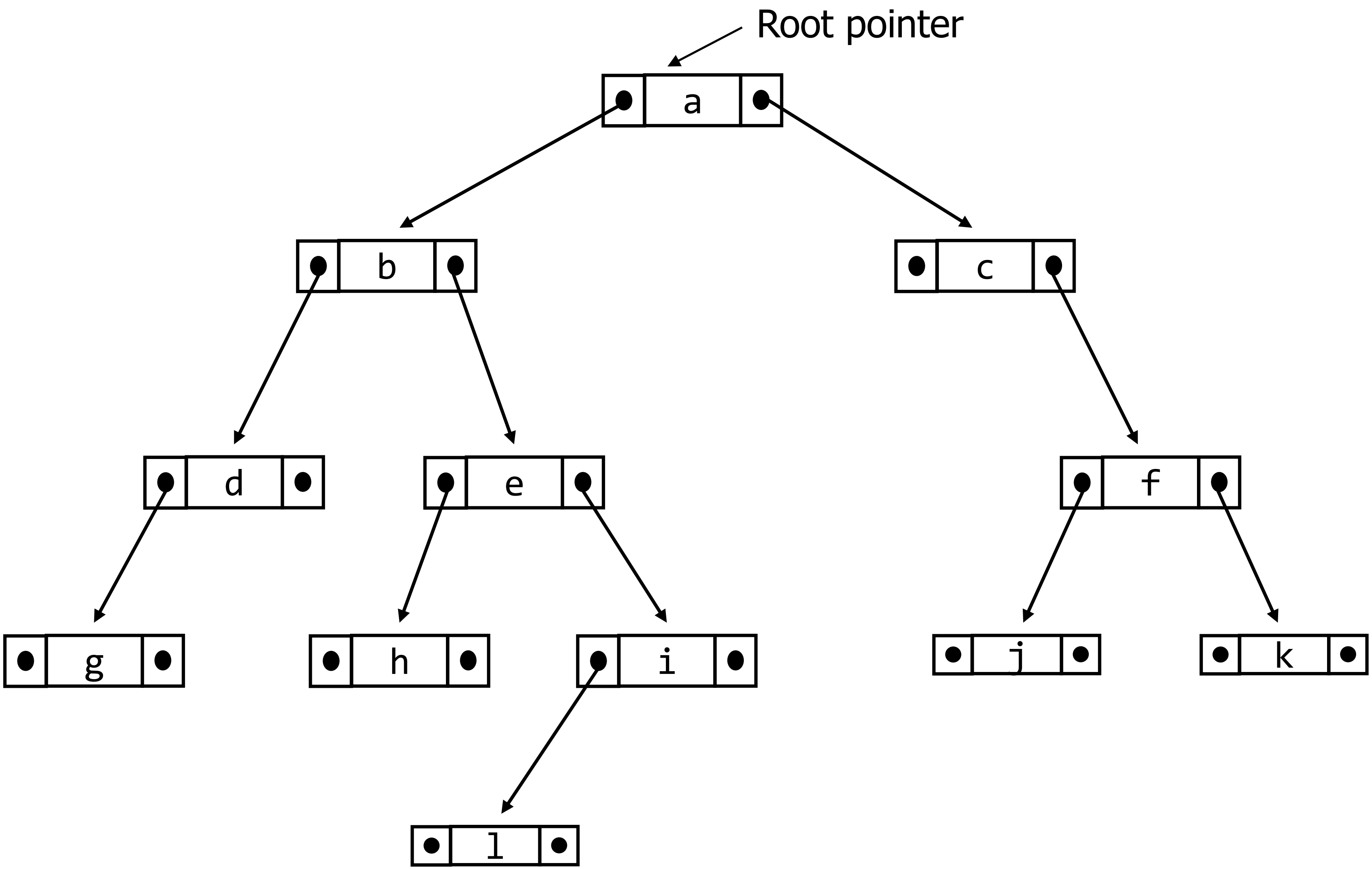
As Linked List Structure (1)

- We can implement a binary tree by using a class which:
  - Stores an element
  - A left child pointer (pointer to first child)
  - A right child pointer (pointer to second child)

```
class Node{
   Type value;
   Node *LeftChild,*RightChild;
}root;
```

- The root pointer points to the root node
  - Follow pointers to find every other element in the tree
- Leaf nodes have `LeftChild` and `RightChild` pointers set to `NULL`

As Linked List Structure: Example



Tree Traversal

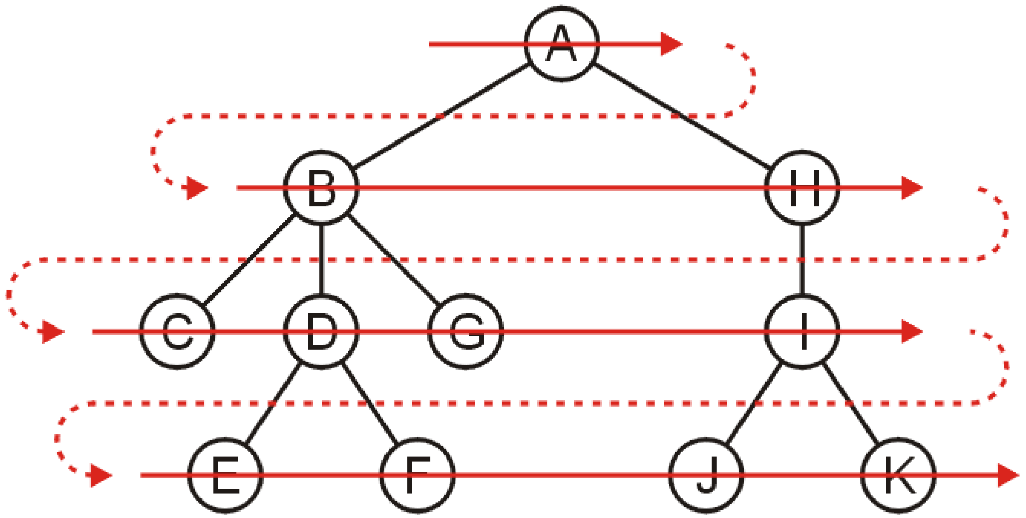
Tree Traversal

- To traverse (or walk) the tree is to visit each node in the tree exactly once
  - Traversal must start at the root node
    - There is a pointer to the root node of the binary tree

- Two types of traversals
  - Breadth-First Traversal
  - Depth-First Traversal

Breadth-First Traversal (For Arbitrary Trees)

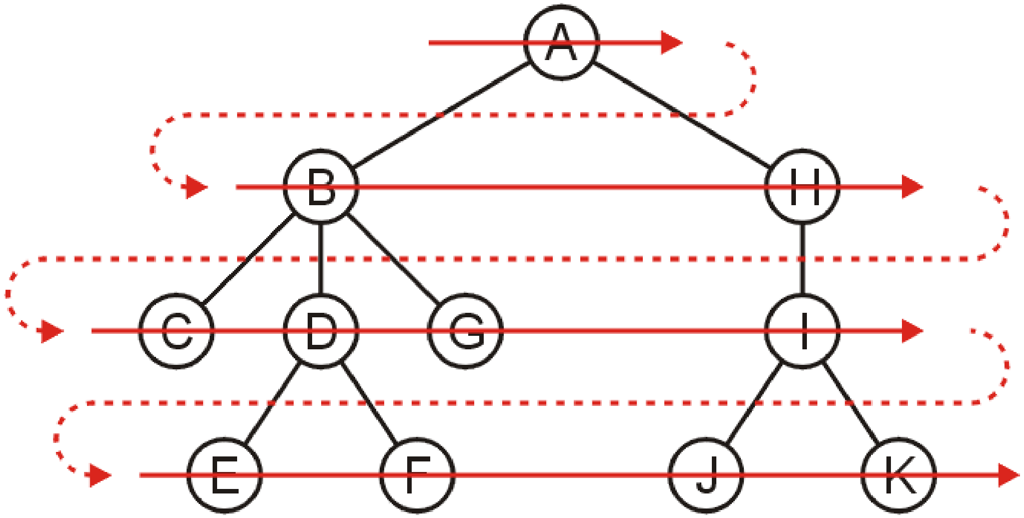
- All nodes at a given depth `d` are traversed before nodes at `d+1`
- Can be implemented using a queue



- Order: A B H C D G I E F J K

Breadth-First Traversal – Implementation

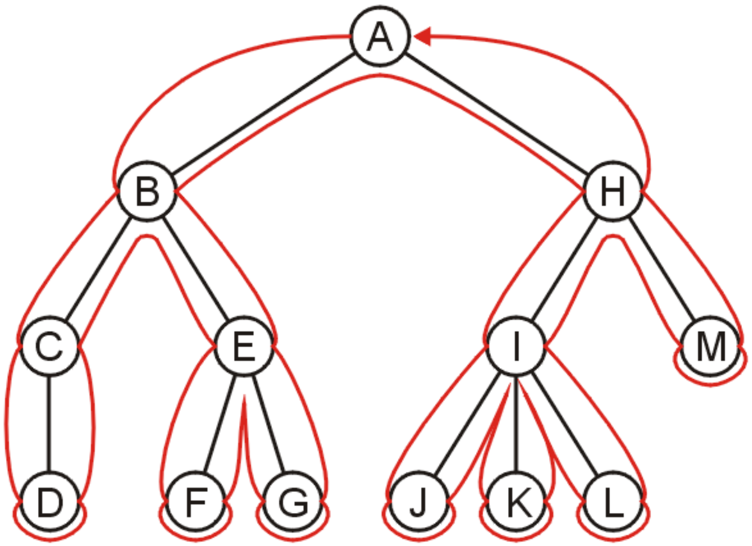
- Create a queue and push the root node onto the queue
- While the queue is not empty:
  - Enqueue all children of the front node onto the queue
  - Dequeue the front node



Depth-First Traversal (For Arbitrary Trees)

- Traverse as much as possible along the branch of each child before going to the next sibling
  - Nodes along one branch of the tree are traversed before backtracking

- Each node could be visited multiple times in such a scheme
  - The first time the node is approached (before any children)
  - The last time it is approached (after all children)



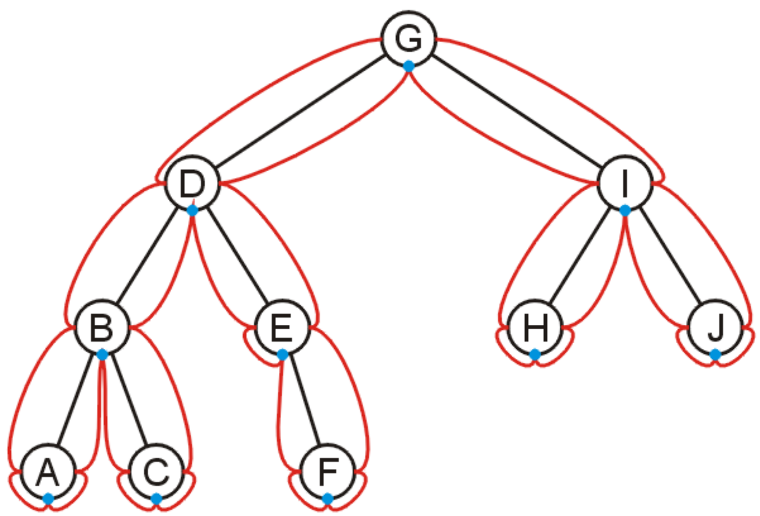
Depth-First Tree Traversal (Binary Trees)

- For each node in a binary tree, there are three choices
  - Visit the node first
  - Visit one of the subtrees first
  - Visit both the subtrees first

- These choices lead to three commonly used traversals
  - Inorder traversal: (Left subtree) visit Root (Right subtree)
  - Preorder traversal: visit Root (Left subtree) (Right subtree)
  - Postorder traversal: (Left subtree) (Right subtree) visit Root

Inorder Traversal

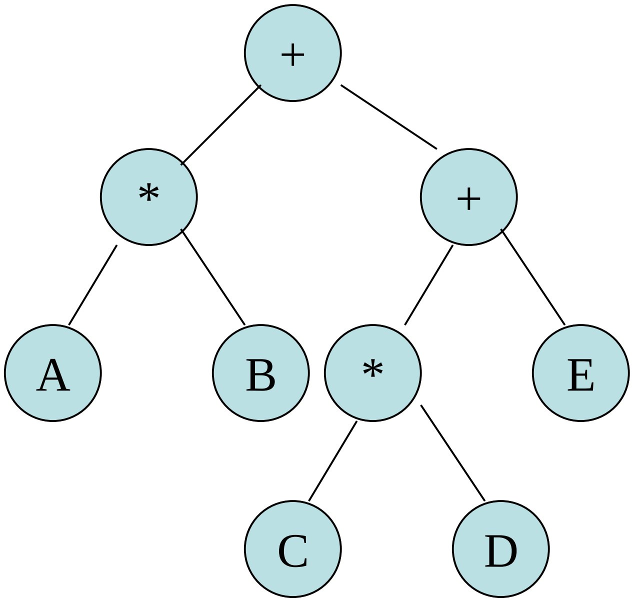
- Algorithm
  1. Traverse the left subtree in inorder
  2. Visit the root
  3. Traverse the right subtree in inorder



A, B, C, D, E, F, G, H, I, J

Inorder Traversal

- Algorithm
  1. Traverse the left subtree in inorder
  2. Visit the root
  3. Traverse the right subtree in inorder



- Example
  - Left + Right
  - [Left * Right] + [Left + Right]
  - (A * B) + [(Left * Right) + E)
  - (A * B) + [(C * D) + E]

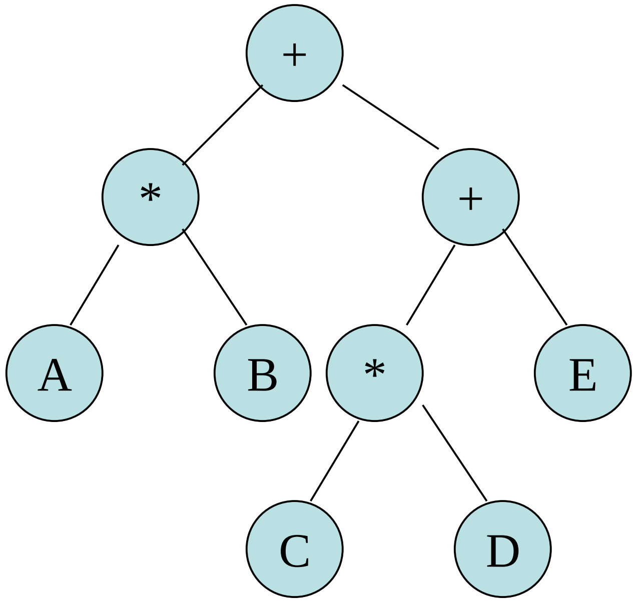
Inorder Traversal – Implementation

```
void inorder(Node *p) const
{
   if (p != NULL)
   {
      inorder(p->leftChild);
      cout << p->info << " ";
      inorder(p->rightChild);
   }
}

void traverse_inorder () {
   . . .
   inorder (root);
}
```

Preorder Traversal

- Algorithm
  1. Visit the node
  2. Traverse the left subtree
  3. Traverse the right subtree



- Example
  - + Left Right
  - + [* Left Right] [+ Left Right]
  - + (* AB) [+ * Left Right E]
  - +*AB + *C D E

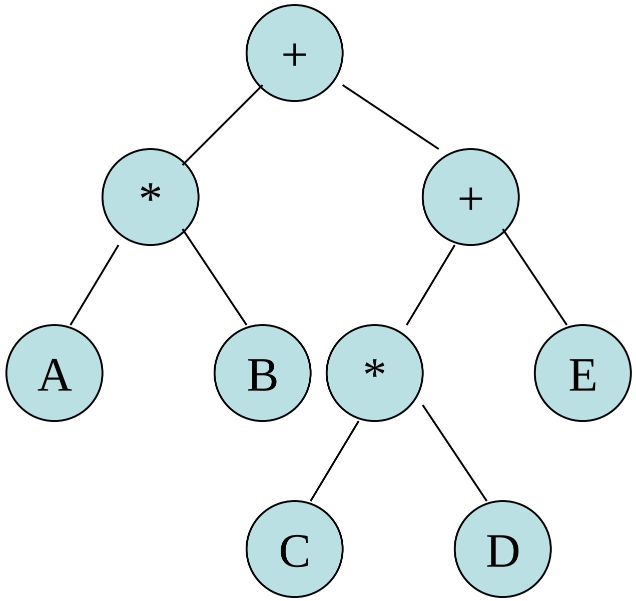
Preorder Traversal – Implementation

```
void preorder(Node *p) const
{
   if (p != NULL)
   {
      cout << p->info << " ";
      preorder(p->leftChild);
      preorder(p->rightChild);
   }
}

void traverse_preorder () {
   . . .
   preorder (root);
}
```

Postorder Traversal

- Algorithm
  1. Traverse the left subtree
  2. Traverse the right subtree
  3. Visit the node



- Example
  - Left Right +
  - [Left Right *] [Left Right+] +
  - (AB*) [Left Right * E +]+
  - (AB*) [C D * E +]+
  - AB* C D * E + +

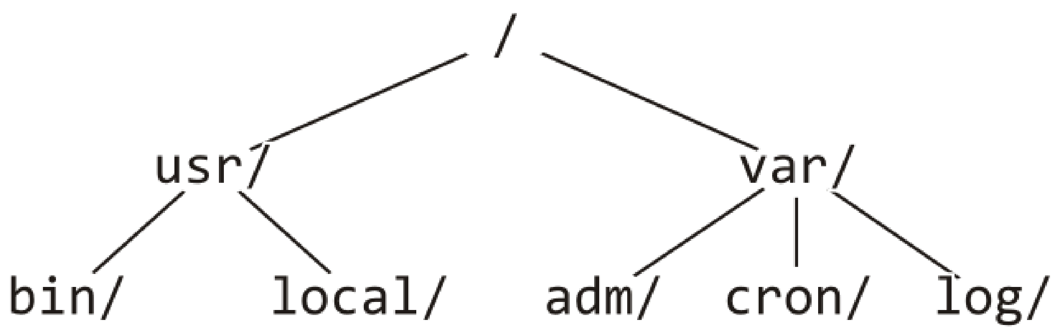
Postorder Traversal – Implementation

```
void postorder(Node *p) const
{
   if (p != NULL)
   {
      postorder(p->leftChild);
      postorder(p->rightChild);
      cout << p->info << " ";
   }
}

void traverse_postorder () {
   . . .
   postorder (root);
}
```

Example: Printing a Directory Hierarchy

- Consider the directory structure presented on the left
  - Which traversal should be used?



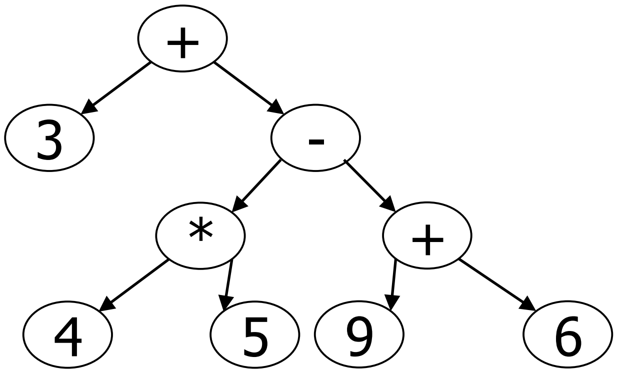

```
/
   usr/
      bin/
      local/
   var/
      adm/
      cron/
      log/
```

Expression Tree

Expression Tree

- Each algebraic expression has an inherent tree-like structure

- An expression tree is a binary tree in which
  - The parentheses in the expression do not appear
    - ➢ Tree representation captures the intent of parenthesis
  - The leaves are the variables or constants in the expression
  - The non-leaf nodes are the operators in the expression
    - ➢ Binary operator has two non-empty subtrees
    - ➢ Unary operator has one non-empty subtree



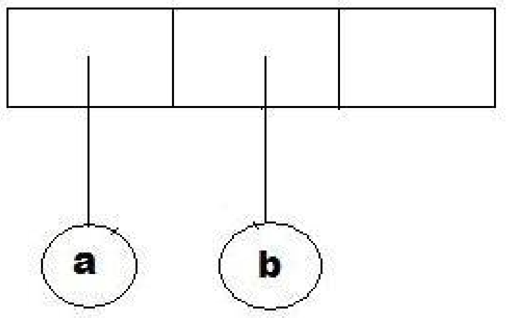
Convert Postfix into Expression Tree – Algorithm

```
while(not the end of the expression)
{
    if(the next symbol in the expression is an operand)
    {
      create a node for the operand ;
      push the reference to the created node onto the stack ;
    }
    if(the next symbol in the expression is a binary operator)
    {
      create a node for the operator ;
      pop from the stack a reference to an operand ;
      make the operand the right subtree of the operator node ;
      pop from the stack a reference to an operand ;
      make the operand the left subtree of the operator node ;
      push the reference to the operator node onto the stack ;
   }
}
```

Convert Postfix into Expression Tree – Example (1)

```
while(not the end of the expression)
{
   if(the next symbol is an operand)
   {
      create a node for the operand ;
      push the reference to the created node onto the stack;
   }
   if(the next symbol is a binary operator)
   {
      create an operator node;
      pop operant from the stack;
      make the operand the right subtree ;
      pop operand from the stack;
      make the operand the left subtree ;
      push the operator node onto the stack;
  }
}
```

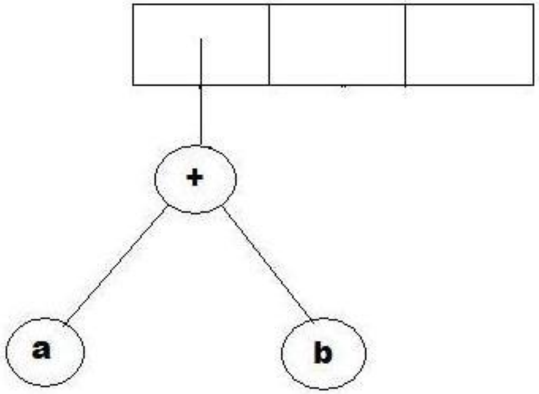
Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example (2)

```
while(not the end of the expression)
{
   if(the next symbol is an operand)
   {
      create a node for the operand ;
      push the reference to the created node onto the stack ;
   }
   if(the next symbol is a binary operator)
   {
      create an operator node;
      pop operant from the stack;
      make the operand the right subtree ;
      pop operand from the stack;
      make the operand the left subtree ;
      push the operator node onto the stack;
  }
}
```

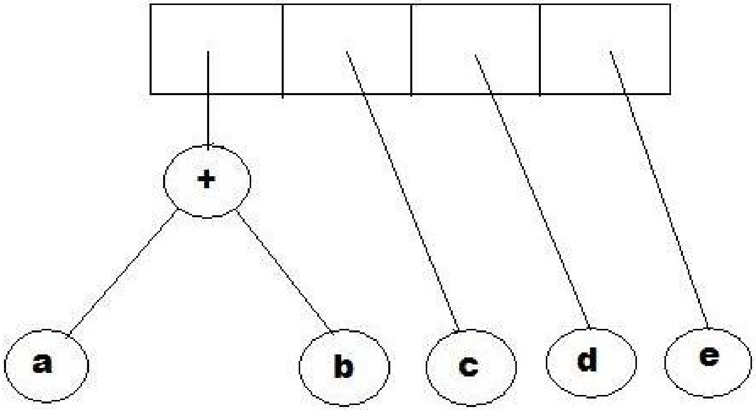
Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example (3)

```
while(not the end of the expression)
{
  if(the next symbol is an operand)
  {
     create a node for the operand ;
     push the reference to the created node onto the stack;
  }
  if(the next symbol is a binary operator)
  {
     create an operator node;
     pop operant from the stack;
     make the operand the right subtree ;
     pop operand from the stack;
     make the operand the left subtree ;
     push the operator node onto the stack;
 }
}
```

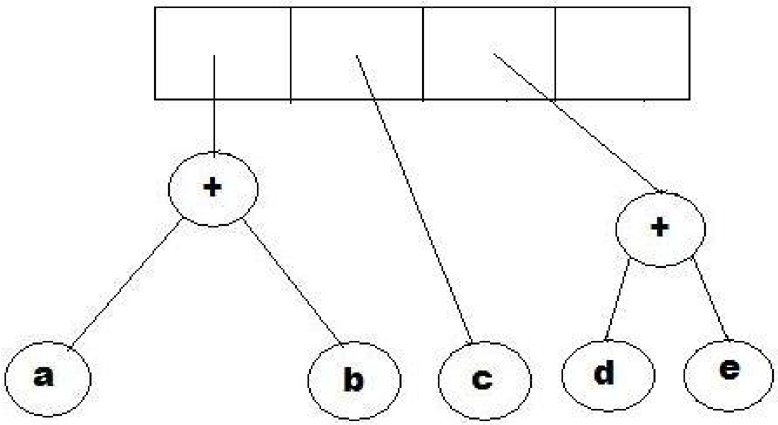
Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example (4)

```
while(not the end of the expression)
{
   if(the next symbol is an operand)
   {
      create a node for the operand ;
      push the reference to the created node onto the stack ;
   }
   if(the next symbol is a binary operator)
   {
      create an operator node;
      pop operant from the stack;
      make the operand the right subtree ;
      pop operand from the stack;
      make the operand the left subtree ;
      push the operator node onto the stack;
  }
}
```

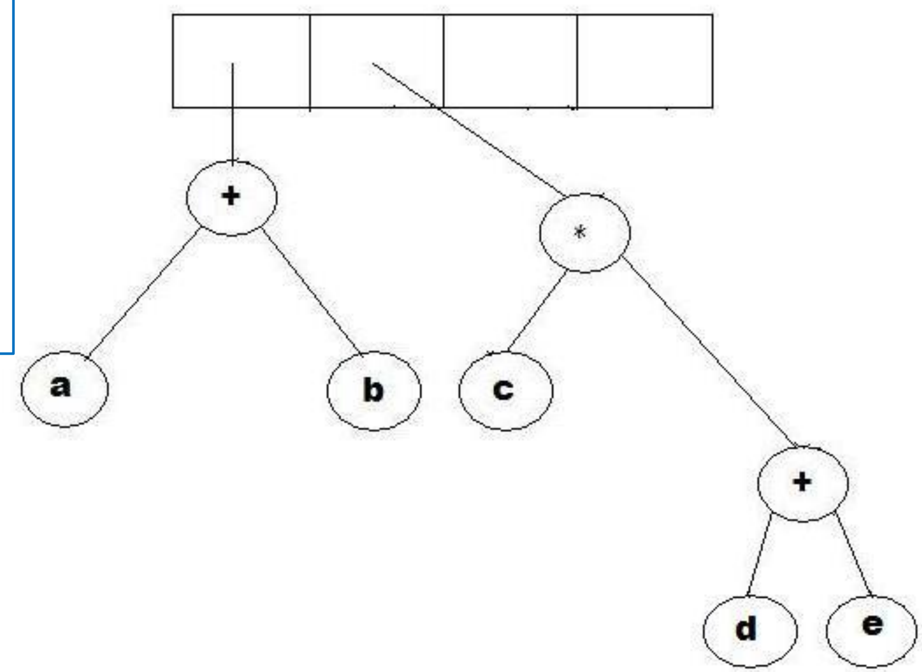
Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example (5)

```
while(not the end of the expression)
{
   if(the next symbol is an operand)
   {
      create a node for the operand ;
      push the reference to the created node onto the stack ;
   }
   if(the next symbol is a binary operator)
   {
      create an operator node;
      pop operant from the stack;
      make the operand the right subtree ;
      pop operand from the stack;
      make the operand the left subtree ;
      push the operator node onto the stack;
  }
}
```

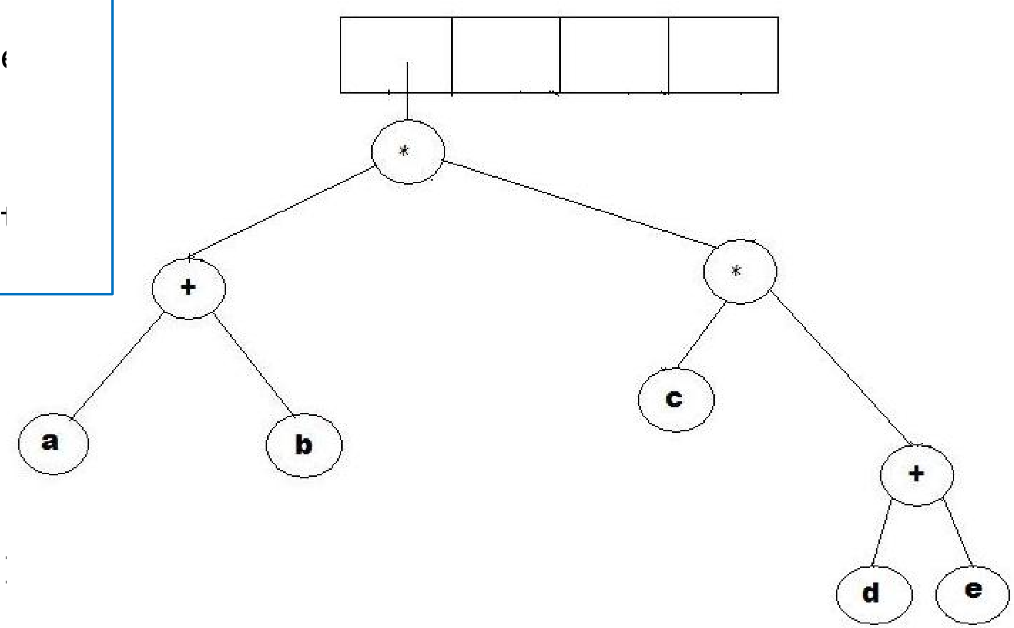
Example:
a b + c d e + * *



Convert Postfix into Expression Tree – Example (6)

```
while(not the end of the expression)
{
   if(the next symbol is an operand)
   {
      create a node for the operand ;
      push the reference to the created node onto the stack ;
   }
   if(the next symbol is a binary operator)
   {
      create an operator node;
      pop operant from the stack;
      make the operand the right subtre
      pop operand from the stack;
      make the operand the left subtree
      push the operator node onto the s
  }
}
```

Example:
a b + c d e + * *



Why Expression Tree?

- Expression trees impose a hierarchy on the operations
  - Terms deeper in the tree get evaluated first
  - Establish correct precedence of operations without using parentheses

- A compiler will read an expression in a language like C++/Java, and transform it into an expression tree

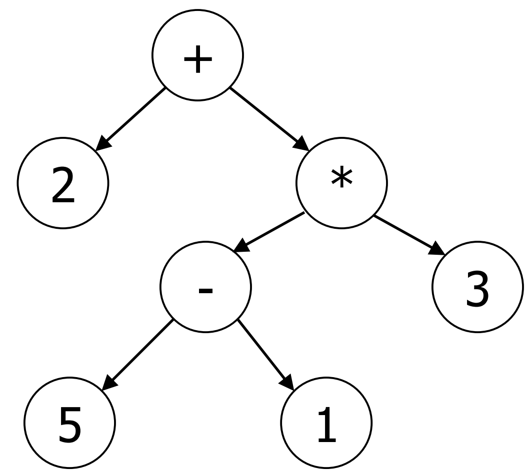
- Expression trees can be very useful for:
  - Evaluation of the expression
  - Generating correct compiler code to actually compute the expression's value at execution time

Evaluating an Expression Tree

- Perform a post-order traversal of the tree
  - Ask each node to evaluate itself
- An operand node evaluates itself by just returning its value
- An operator node has to apply the operator
  - To the results of evaluations from its left subtree and right subtree

Order of evaluation: **3** **1** **2**

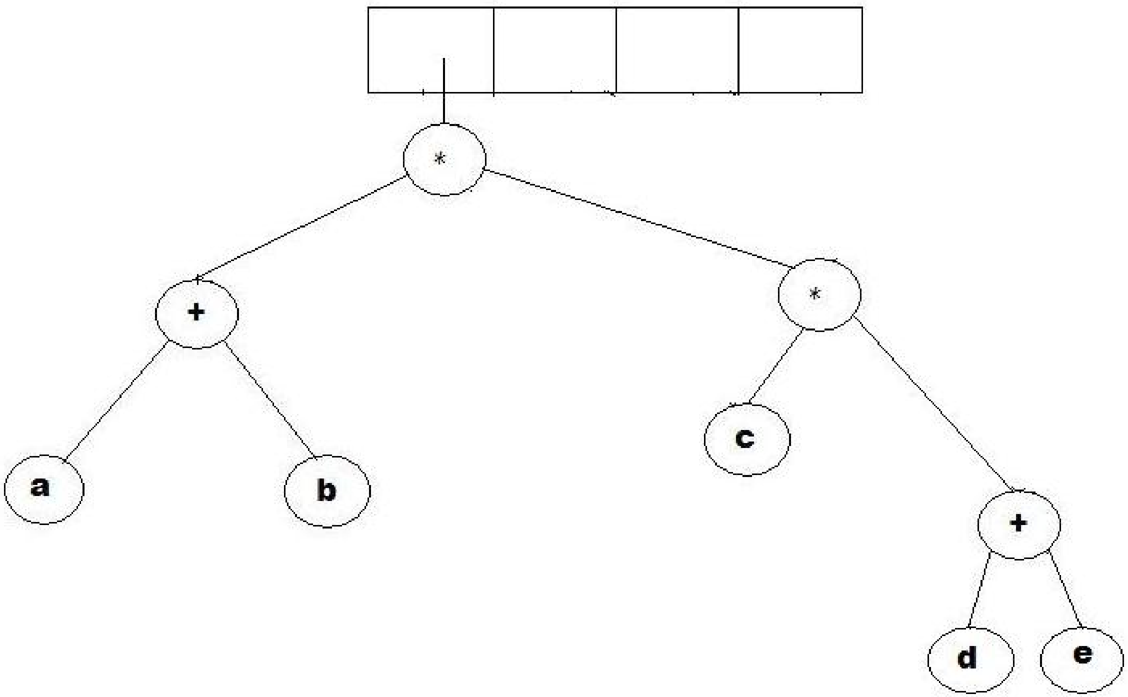
(2 + ((5 – 1) * 3))



Evaluating an Expression Tree – Example

- Expression: a b + c d e + * *

  1 2 + 3 4 5 + * *



Evaluating an Expression Tree - Implementation

```
evaluate(ExpressionTree t){
    if(t is a leaf)
        return value of t's operand ;
    else{
        operator =  t.element ;
        operand1 = evaluate(t.left) ;
        operand2 = evaluate(t.right) ;
        return(applyOperator(operand1, operator, operand2) ;
    }
}
```

Any Question So Far?