DATA STRUCTURES AND ALGORITHMS

Lecture 11: Trees Implementation

Lecturer: Mohsin Abbas

National University of Modern Languages, Islamabad

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

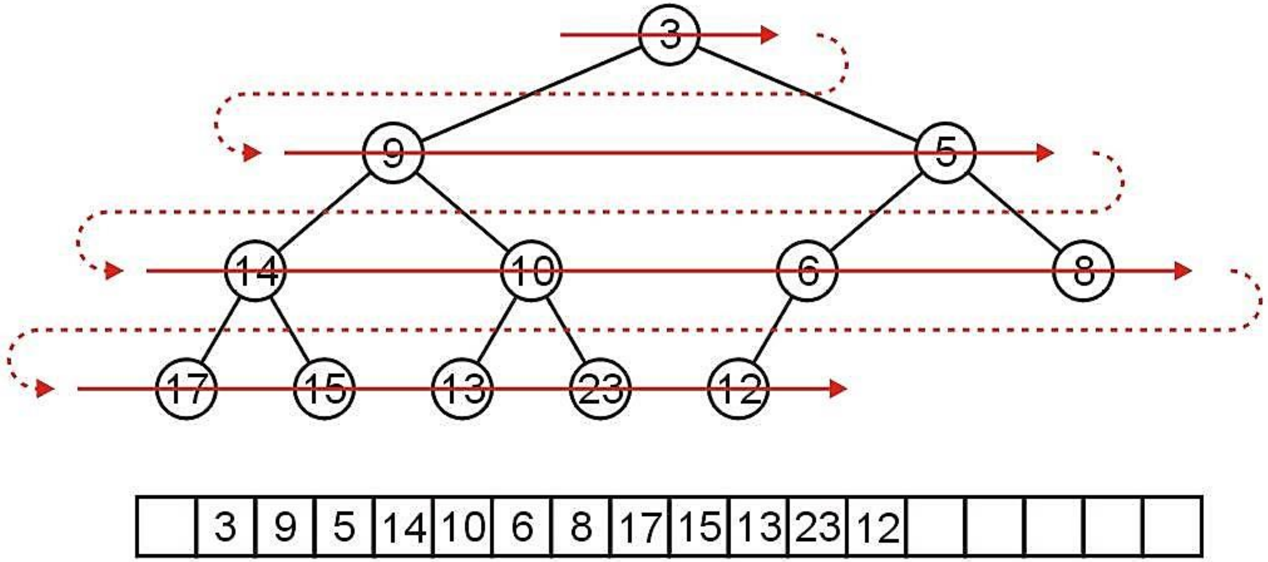
- Contiguous storage
- Linked list based storage

CONTIGUOUS STORAGE

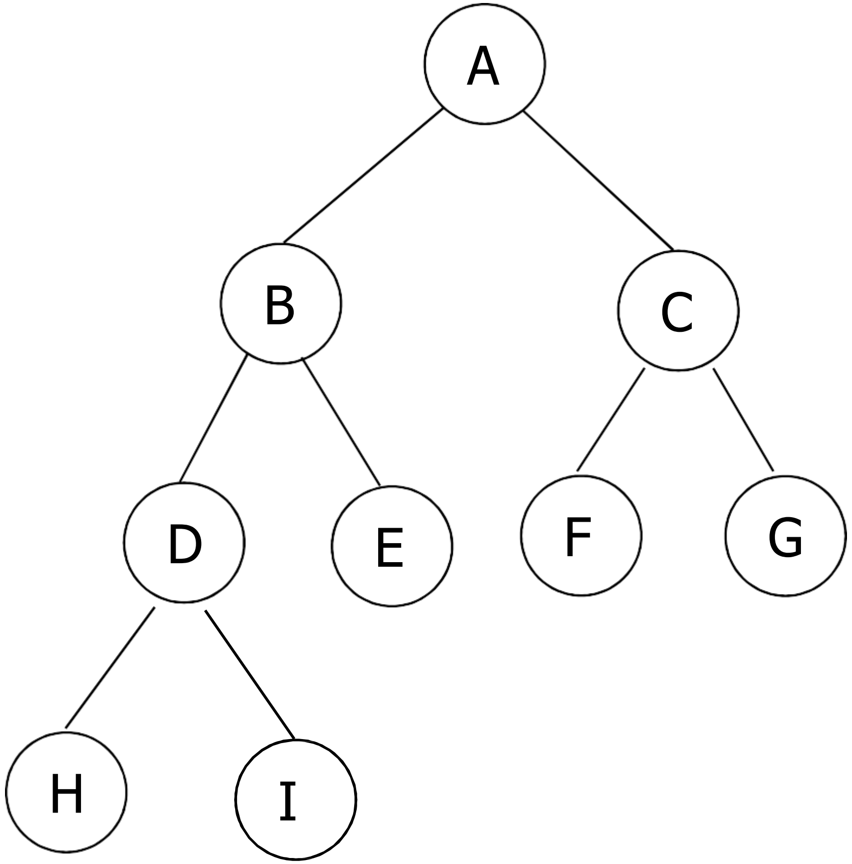
ARRAY STORAGE

- We are able to 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



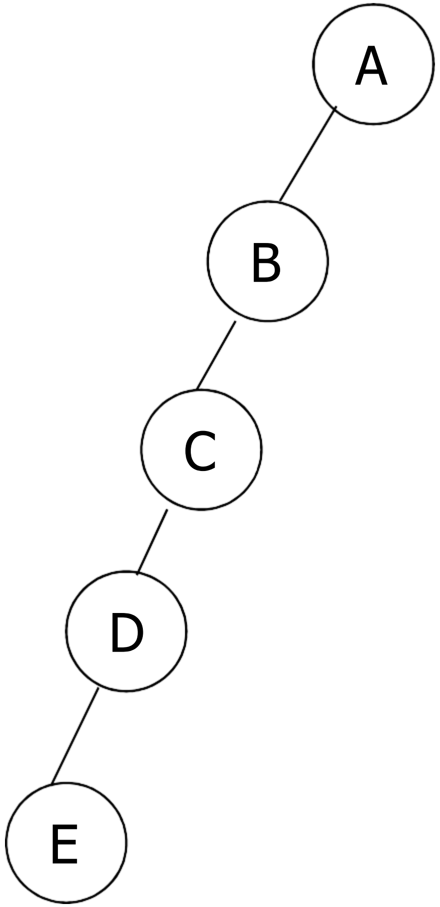
ARRAY STORAGE EXAMPLE



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

ARRAY STORAGE EXAMPLE

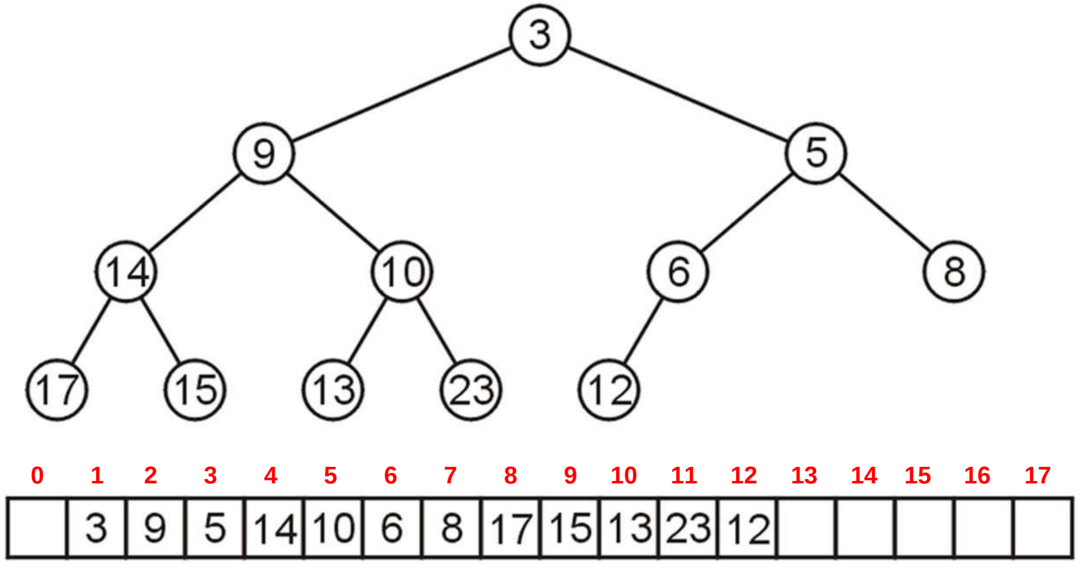
- Unused nodes in tree represented by a predefined bit pattern



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

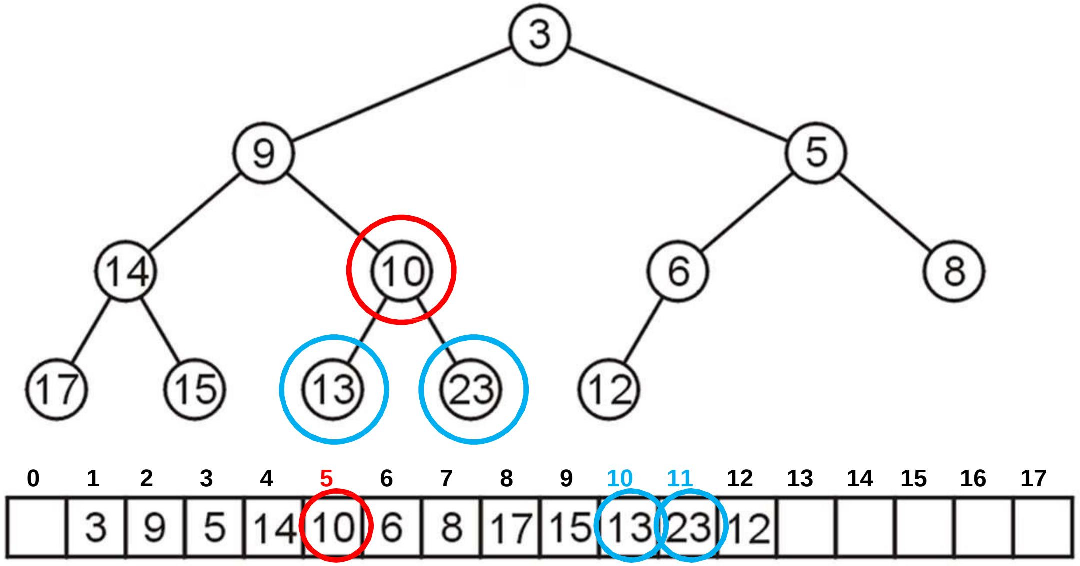
ARRAY STORAGE

- 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



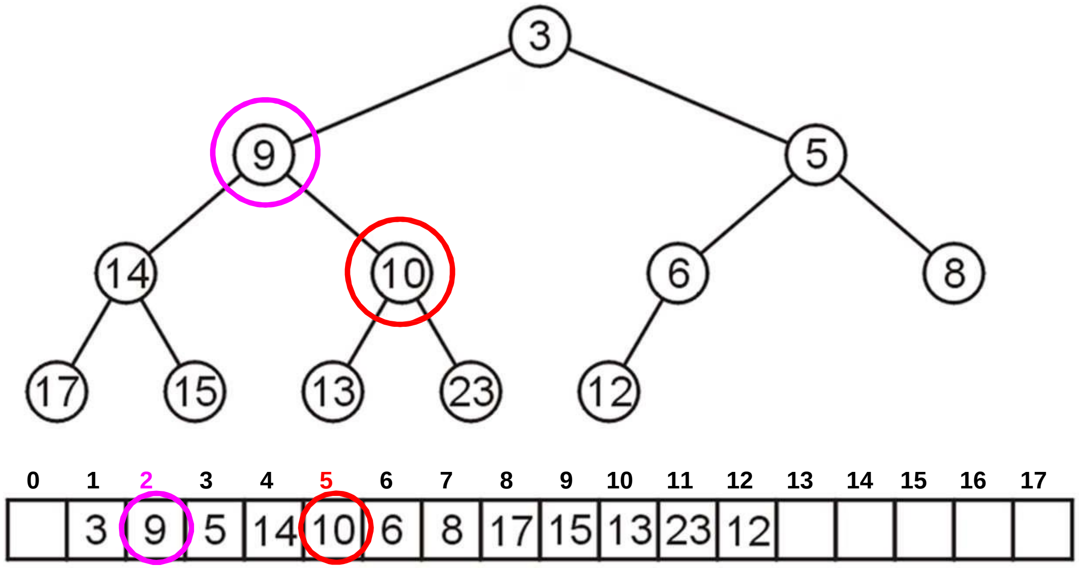
ARRAY STORAGE EXAMPLE

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



ARRAY STORAGE EXAMPLE

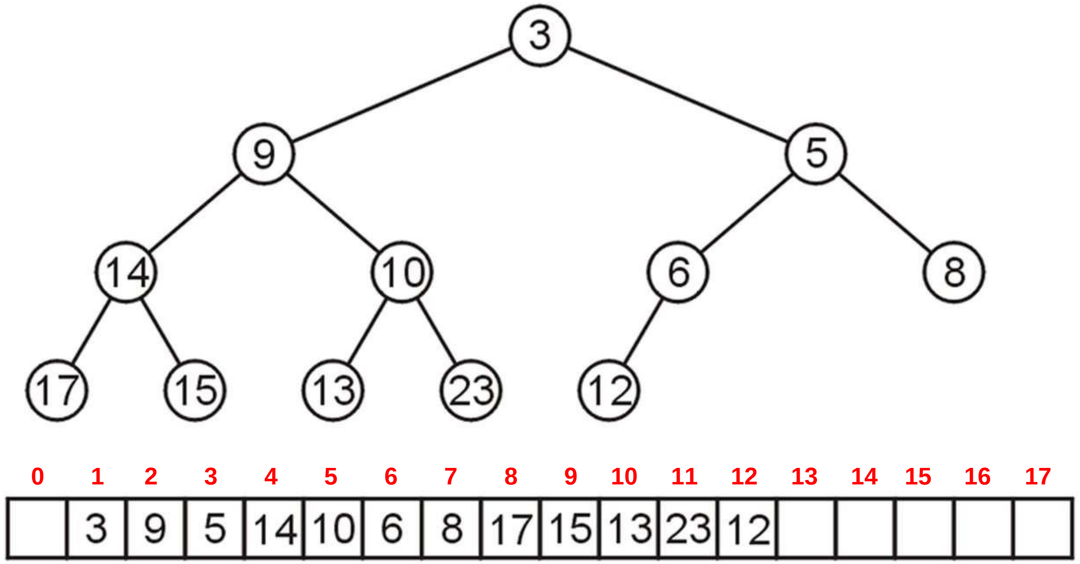
- Node 10 has index 5
  - Its children 13 and 23 have indices 10 and 11, respectively
  - Its parent is node 9 with index 5/2 = 2



ARRAY STORAGE

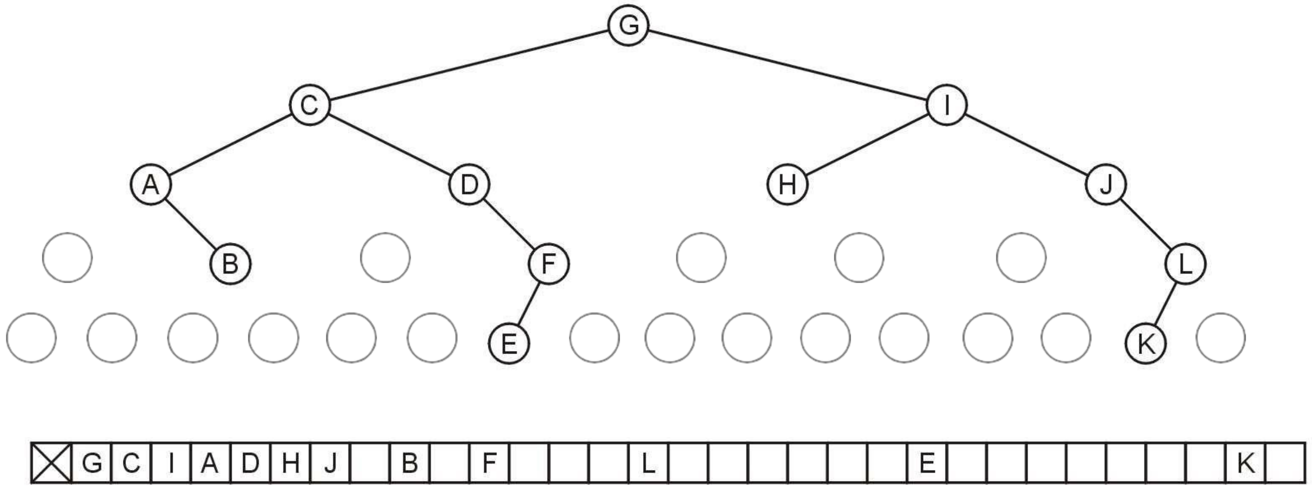
- Why array index is not started from 0
- So having the root at 0 rather than at 1 costs you an extra add to find the left child, and an extra subtraction to find the parent.
  - In C++, this simplifies the calculations

	Root at 0	Root at 1
Left child	Index*2 + 1	Index*2
Right child	Index*2 + 2	Index*2 + 1
Parent	(Index-1) / 2	Index / 2



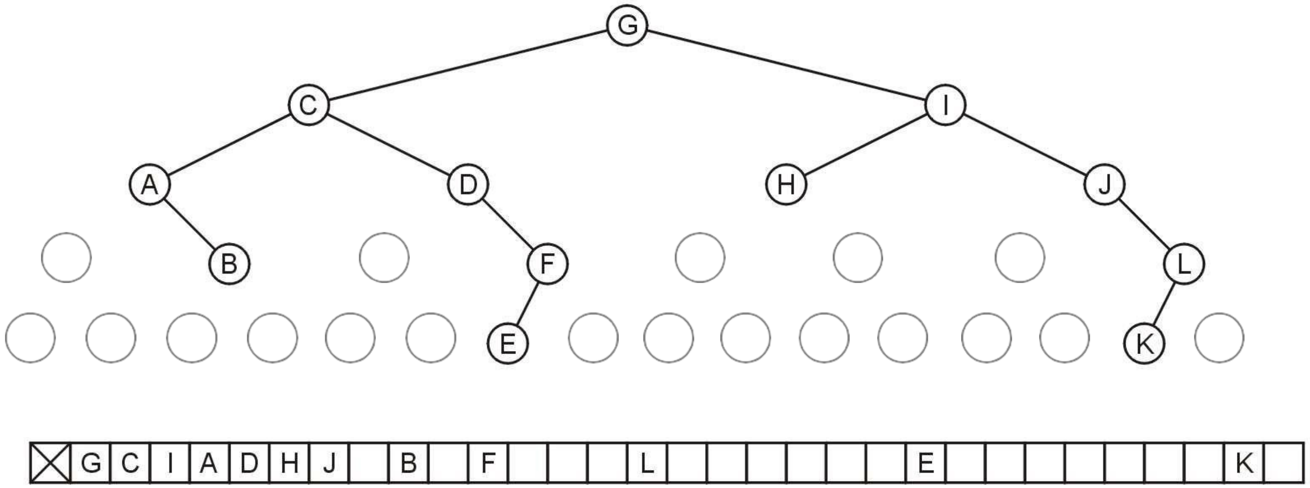
ARRAY STORAGE: DISADVANTAGE

- 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

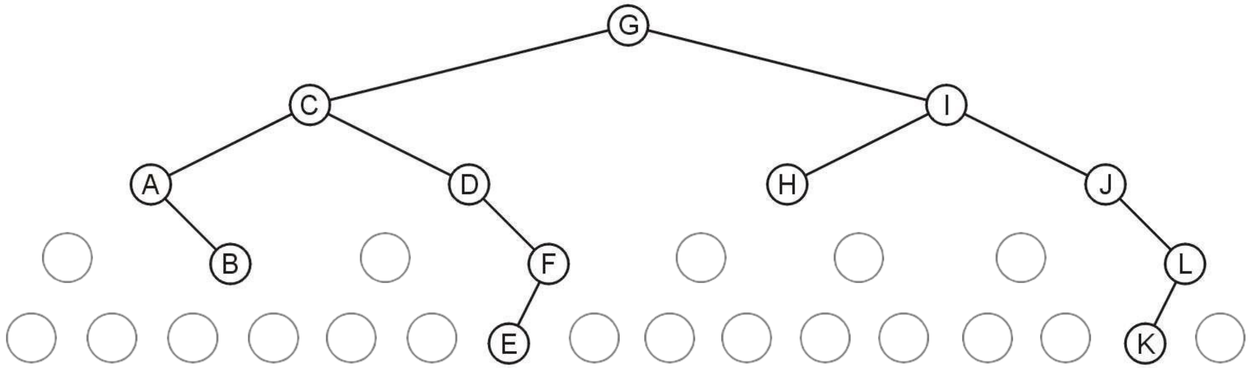
- Why not store any tree as an array using breadth-first traversals?
  - There is a significant potential for a lot of wasted memory
- Consider the following tree with 12 nodes
  - What is the required size of array? **32**
  - What will be the array size if a child is added to node K?



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**



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

LINKED LIST STORAGE

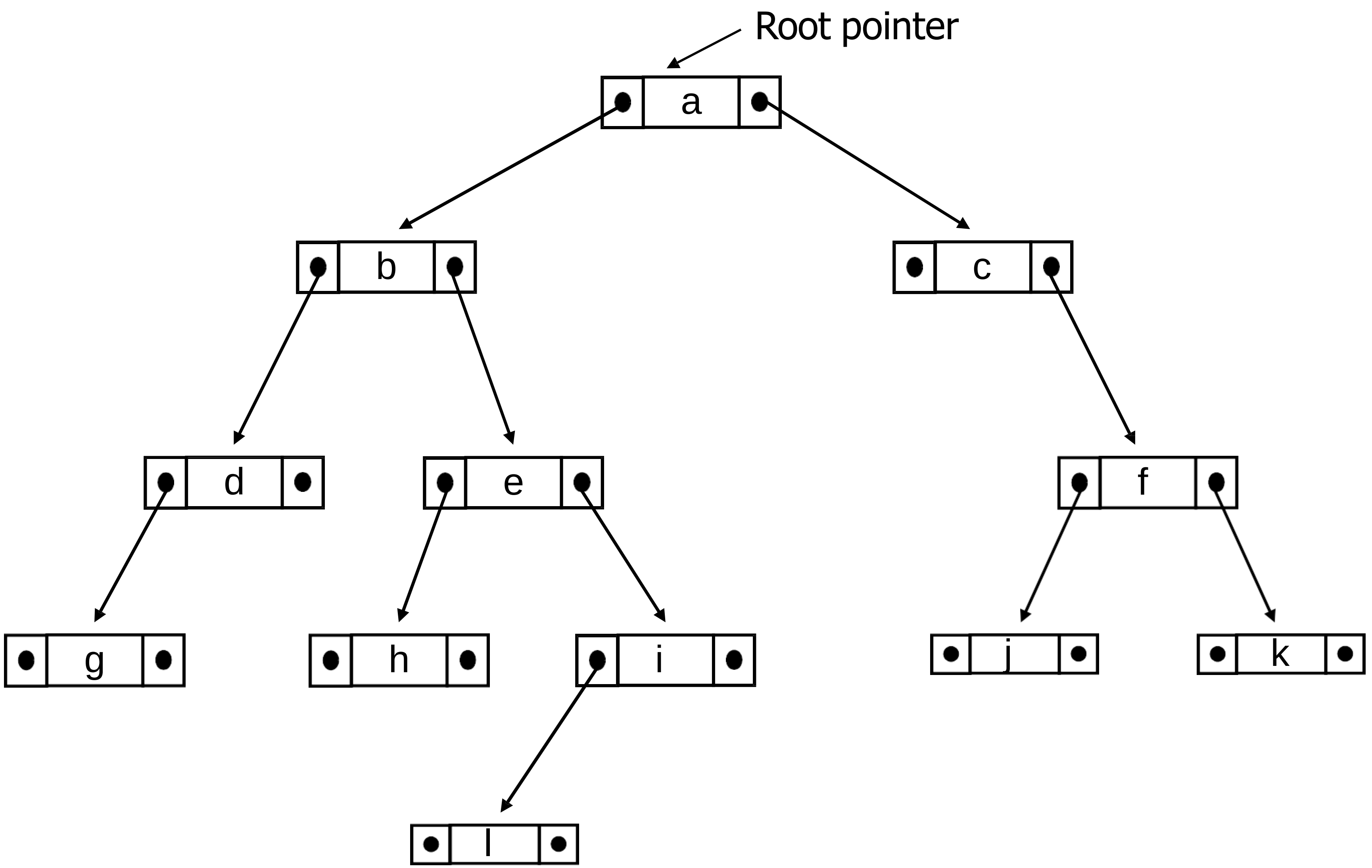
AS LINKED LIST STRUCTURE

- 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;
};
```

- 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`

ARRAY STORAGE EXAMPLE



TREE TRAVERSAL

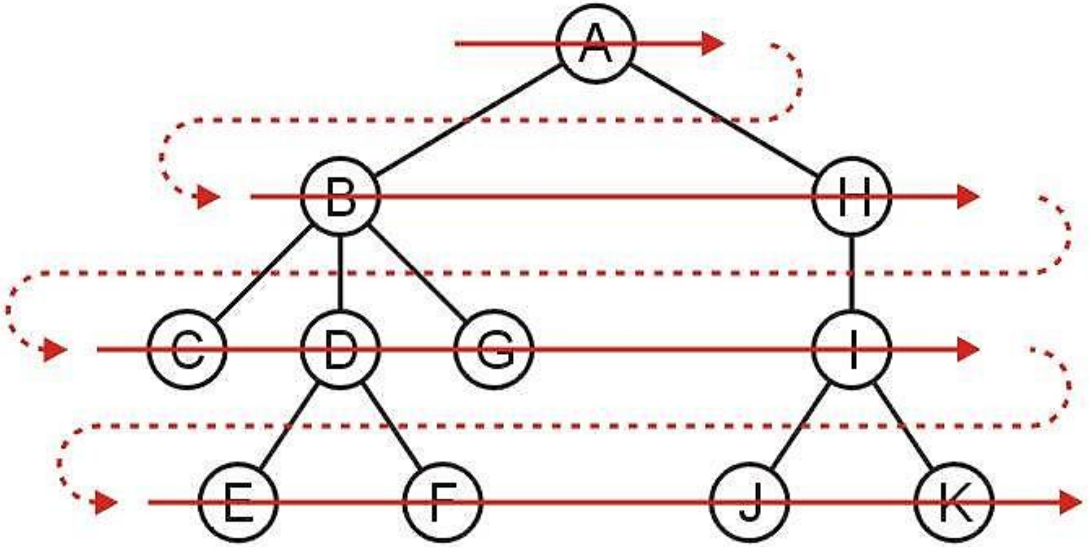
TREE TRAVERSAL

- 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)

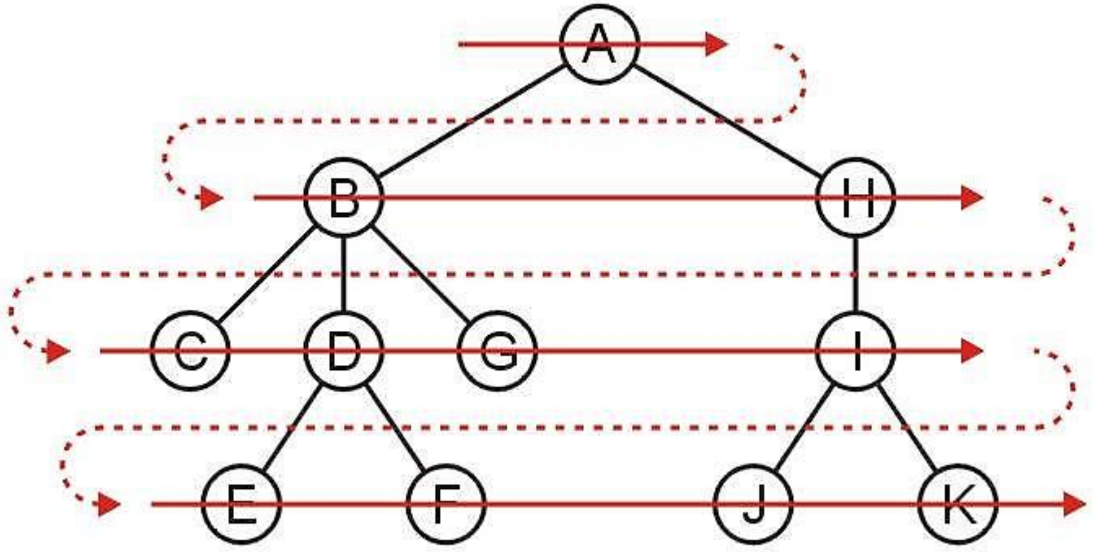
- All nodes at a given depth d are traversed before nodes at d+1
- Can be implemented using a queue



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

BREADTH-FIRST TRAVERSAL – IMPLEMENTATION

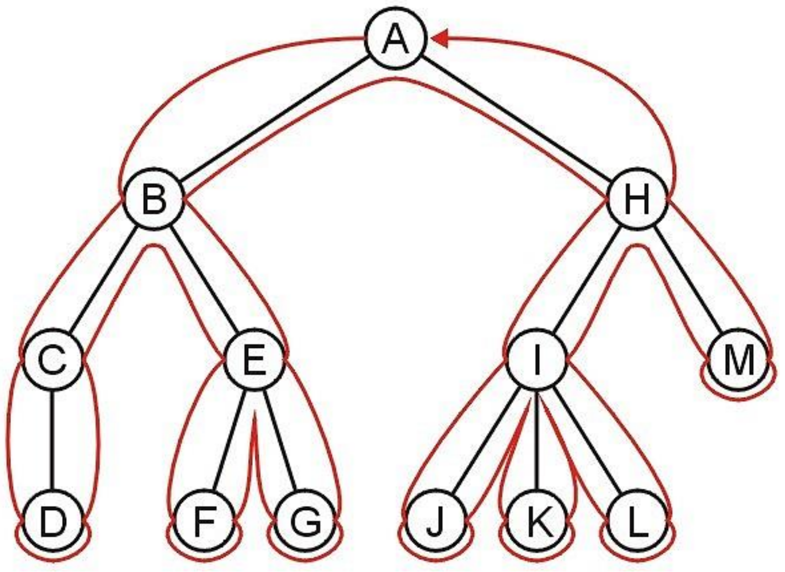
- Create a queue and push the root node onto the queue
- While the queue is not empty:
  - Enqueue all children of the front node onto the queue
  - Dequeue the front node



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)



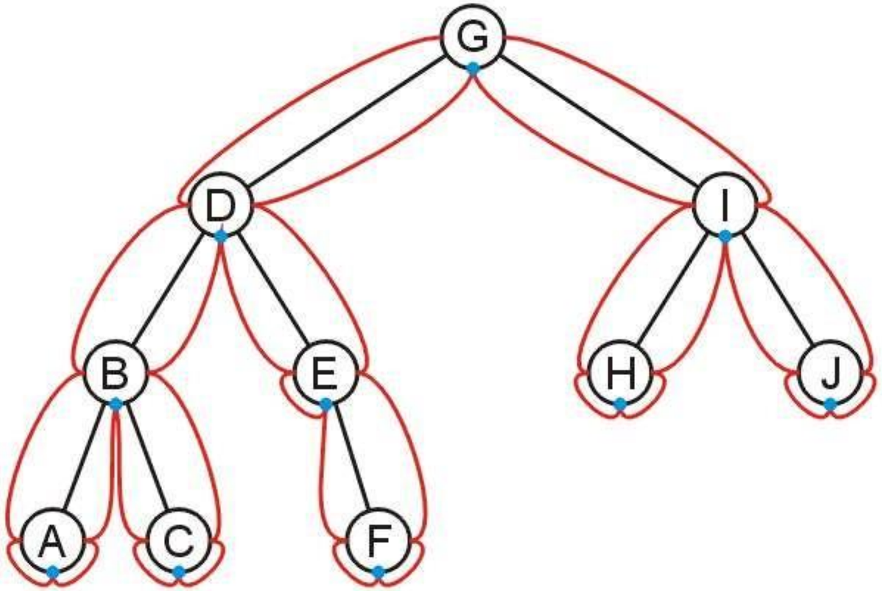
DEPTH-FIRST TREE TRAVERSAL (BINARY TREES)

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

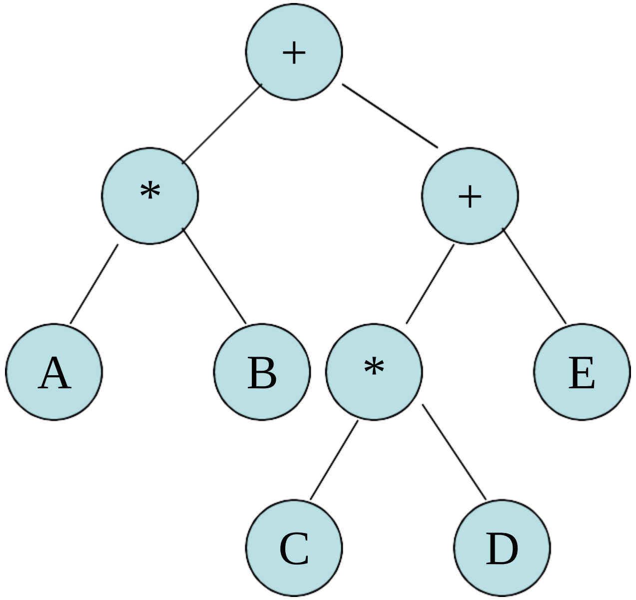
- Algorithm
  1. Traverse the left subtree in inorder
  2. Visit the root
  3. Traverse the right subtree in inorder



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

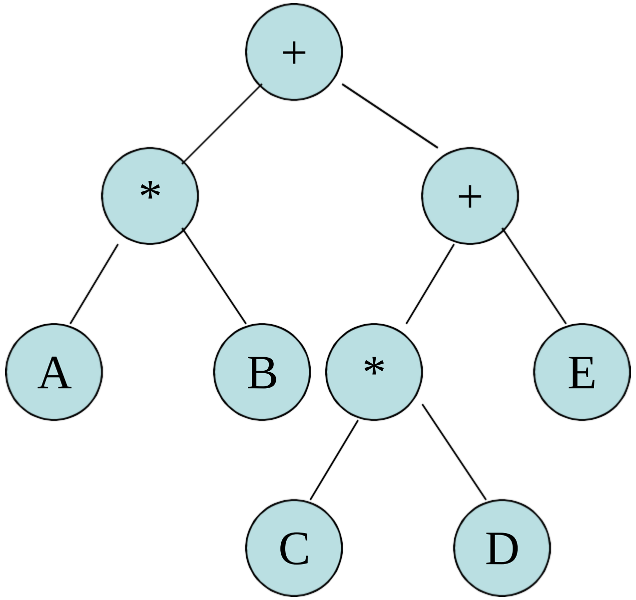
```
void inorder(Node *p) const
{
   if  (p != NULL)
   {
      inorder(p->leftChild);
      cout << p->info << "  ";
      inorder(p->rightChild);}
}

void main () {
   .  .  .
   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



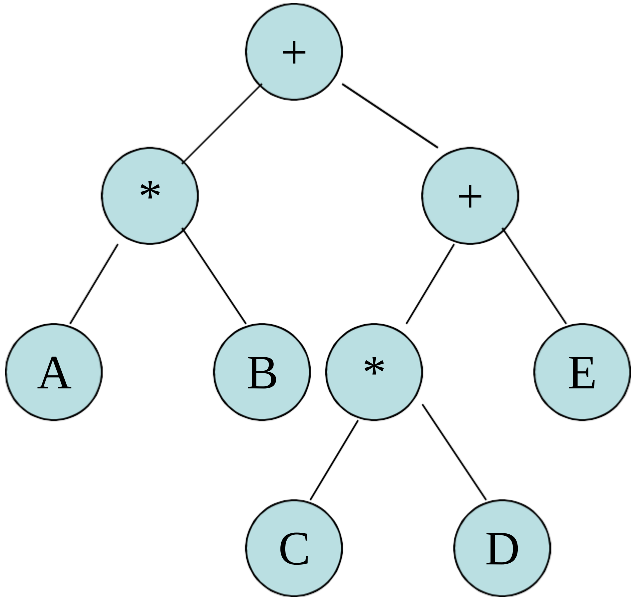
PREORDER TRAVERSAL – IMPLEMENTATION

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

void main () {
   .  .  .
   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

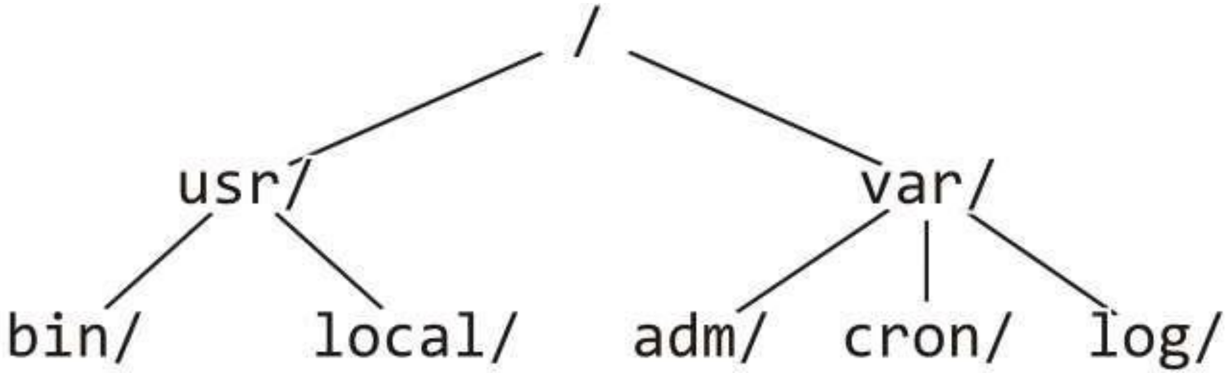
```
void postorder(Node *p) const
{
   if  (p != NULL)
   {
      postorder(p->leftChild);
      postorder(p->rightChild);
      cout << p->info << "  ";
   }
}

void main () {
   .  .  .
   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/
```



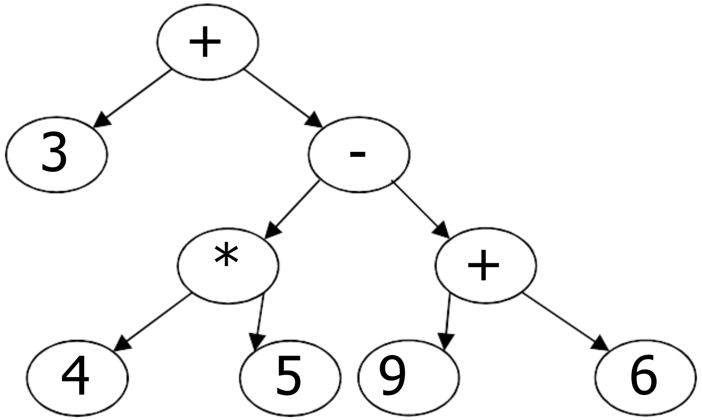
EXPRESSION TREE

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



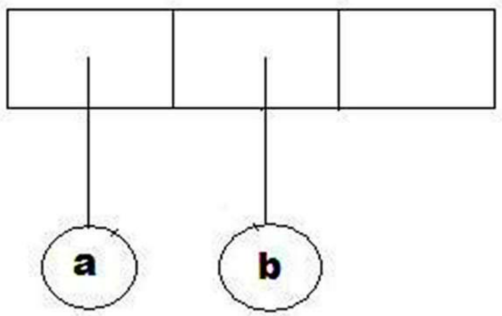
CONVERT POSTFIX INTO EXPRESSION TREE – ALGORITHM

```
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

```
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 operand 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;
  }
}
```

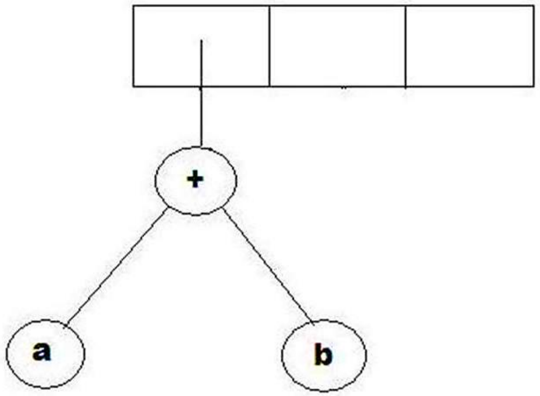
Example:
a b + c d e + * *



CONVERT POSTFIX INTO EXPRESSION TREE – EXAMPLE

```
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 operand 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;
   }
}
```

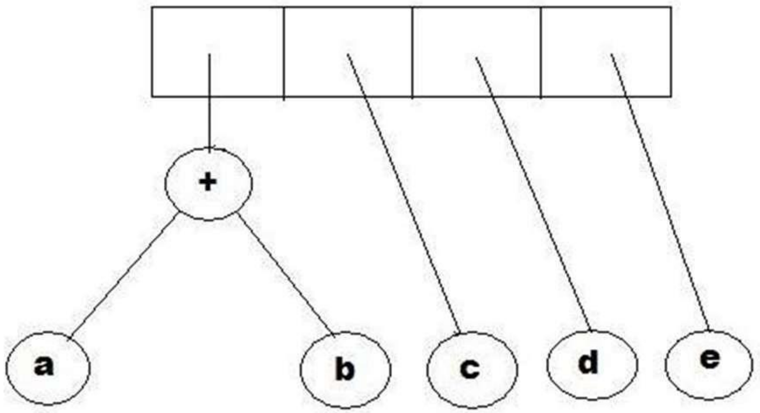
Example:
a b + c d e + * *



CONVERT POSTFIX INTO EXPRESSION TREE – EXAMPLE

```
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 operand 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;
  }
}
```

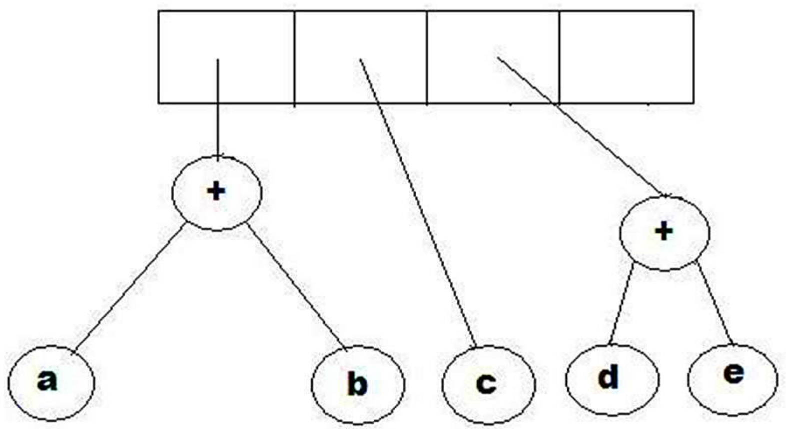
Example:
a b + c d e + * *



CONVERT POSTFIX INTO EXPRESSION TREE – EXAMPLE

```
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 operand 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;
   }
}
```

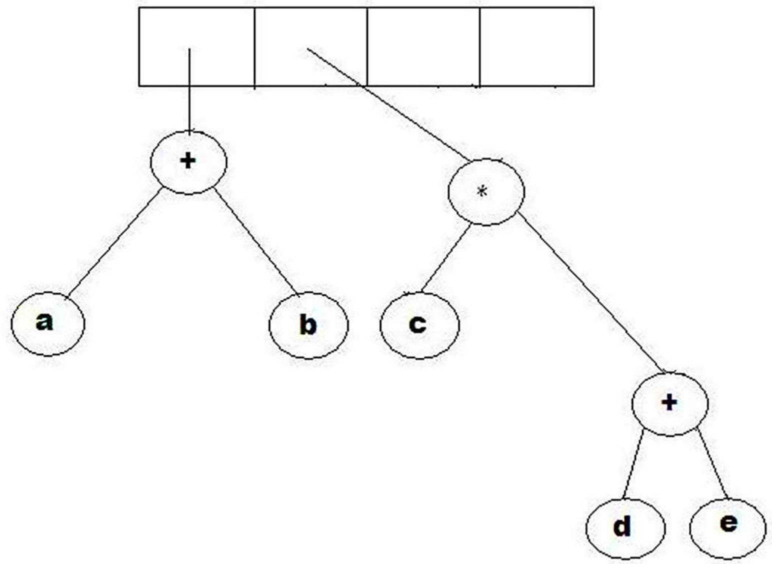
Example:
a b + c d e + * *



CONVERT POSTFIX INTO EXPRESSION TREE – EXAMPLE

```
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 operand 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;
   }
}
```

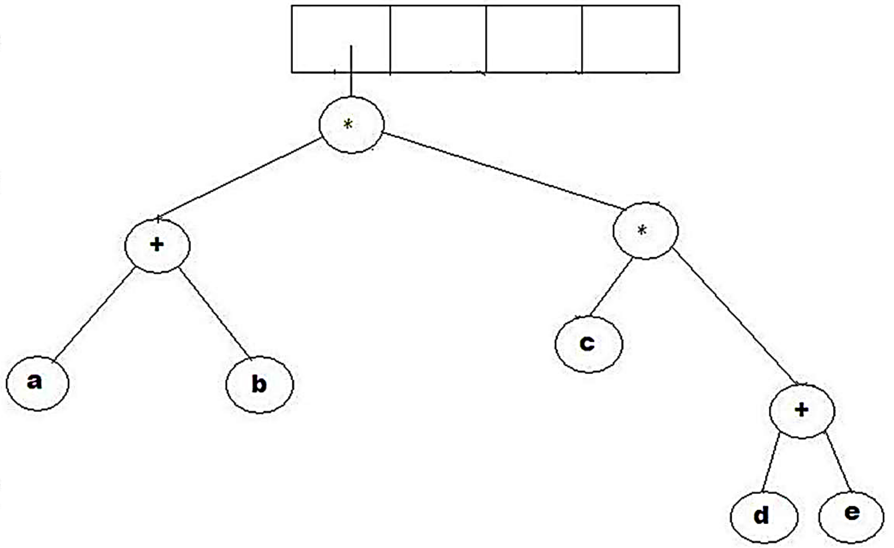
Example:
a b + c d e + * *



CONVERT POSTFIX INTO EXPRESSION TREE – EXAMPLE

```
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 operand 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 + * *



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

CONCLUSION

- In this lecture we have studied:
  - Tree Storage
    - Array
    - Linked List
  - Tree Traversal
    - In order
    - Pre order
    - Post order
  - Expression Tree

Question?