Algorithms and Data Structures 2

4. Trees

Trees

- A *Tree* is a non-linear structure defined by the concept that each *node* in the tree, other than the first node or *root* node, has exactly one parent

- For trees, the operations are dependent upon the type of tree and it's use

Definitions

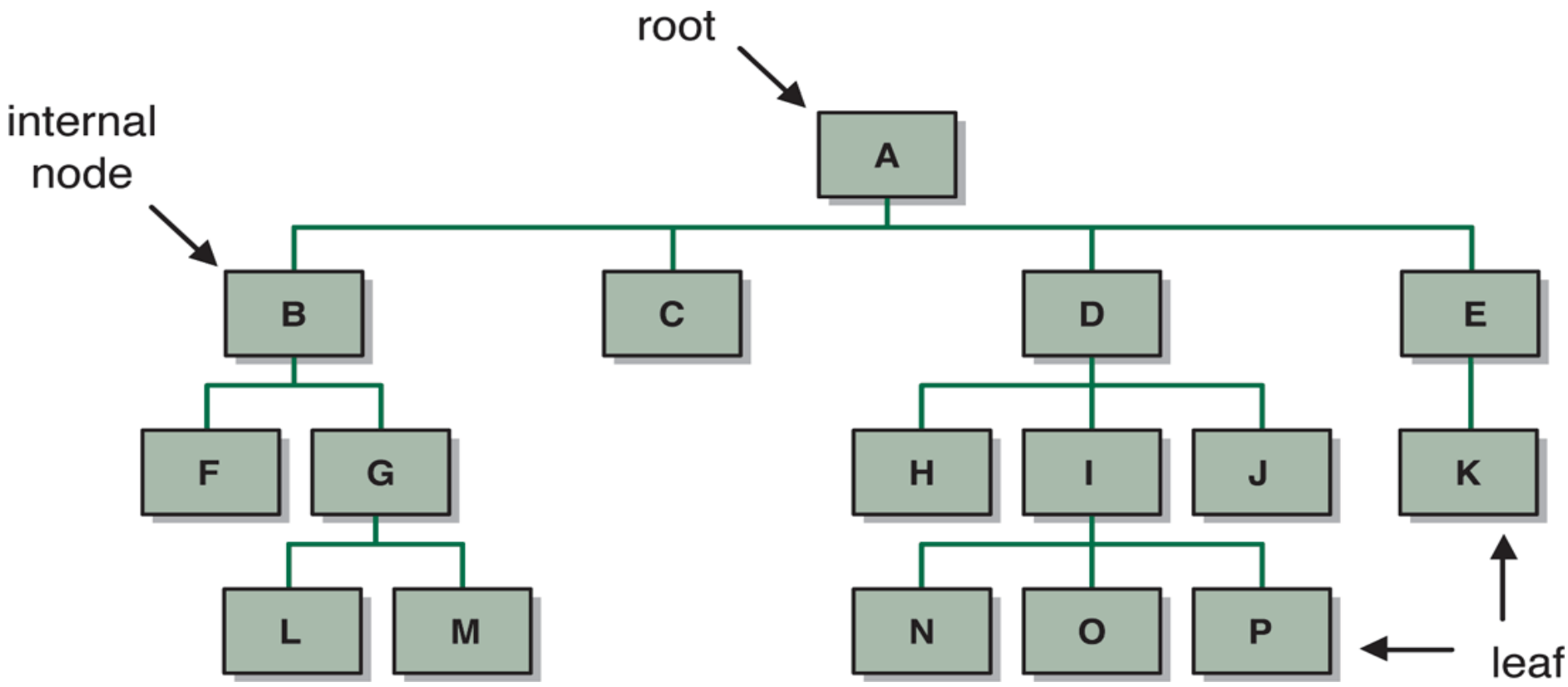
- In order to discuss trees, we must first have a common vocabulary
- We have already introduced a couple of terms:
  - *node* which refers to a location in the tree where an element is stored, and
  - *root* which refers to the node at the base of the tree or the one node in the tree that does not have a parent

Trees, in this context, are generally viewed upside down with the root at the top.

Definitions

- Each node of the tree points to the nodes that are directly beneath it in the tree
- These nodes are referred to as its *children*
- A child of a child is then called a *grandchild*, a child of a grandchild called a *great-grandchild*
- A node that does not have at least one child is called a *leaf*
- A node that is not the root and has at least one child is called an *internal node*

Tree terminology



Definitions

- Any node below another node and on a path from that node is called a *descendant* of that node
- Any node above another node on a connecting path from the root to that node is called an *ancestor* of that node
- All children of the same node are called *siblings*
- A tree that limits each node to no more than n children is called an n-ary tree

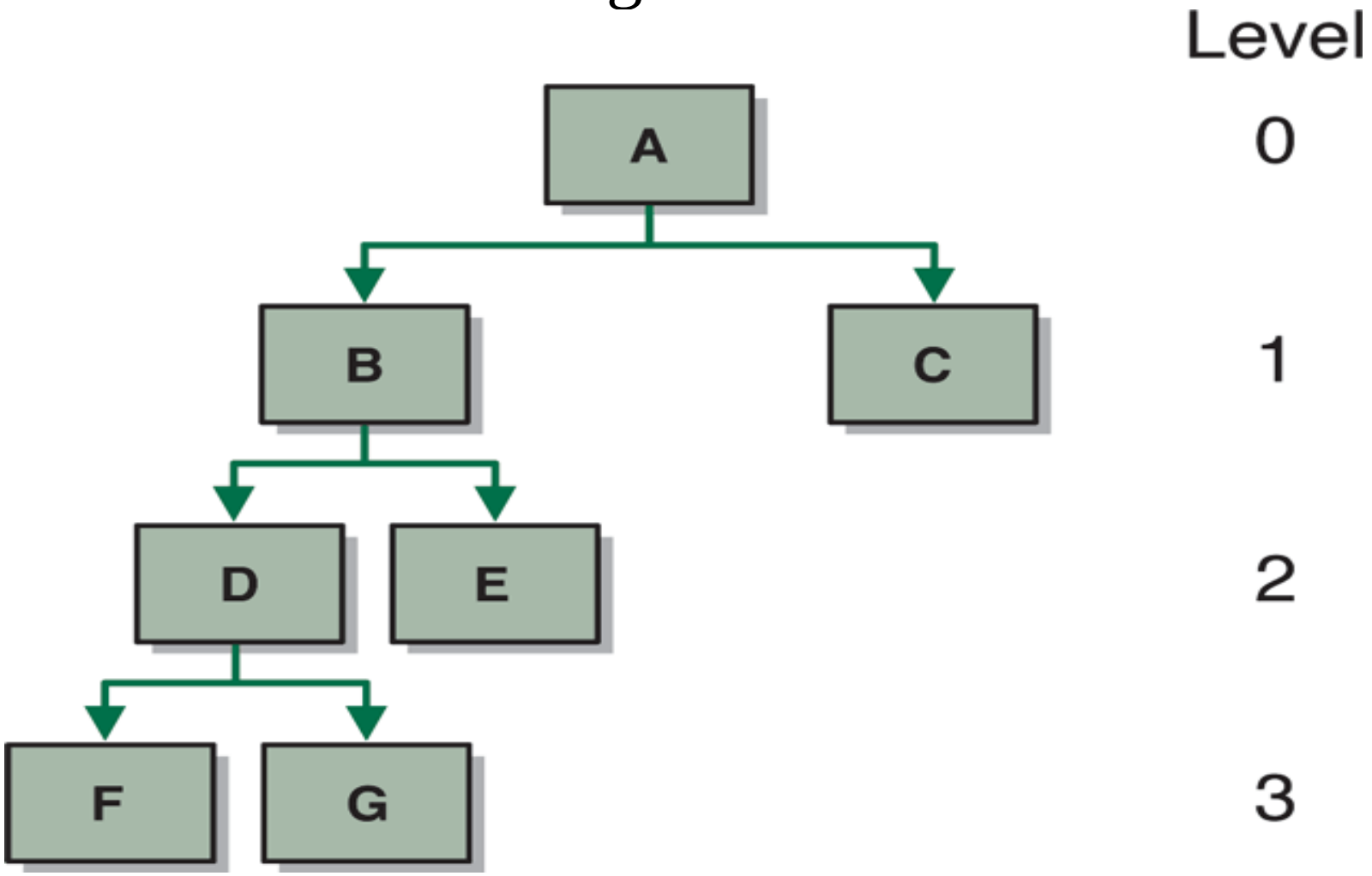
Definitions

- Each node of the tree is at a specific *level* or *depth* within the tree
- The level of a node is the length of the path from the root to the node
- This *pathlength* is determined by counting the number of links that must be followed to get from the root to the node
- The root is considered to be level 0, the children of the root are at level 1, the grandchildren of the root are at level 2, and so on

Definitions

- The *height* or *order* of a tree is the length of the longest path from the root to a leaf
- Thus the height or order of the tree in the next slide is 3
- The path from the root (A) to leaf (F) is of length 3
- The path from the root (A) to leaf (C) is of length 1

Path length and level

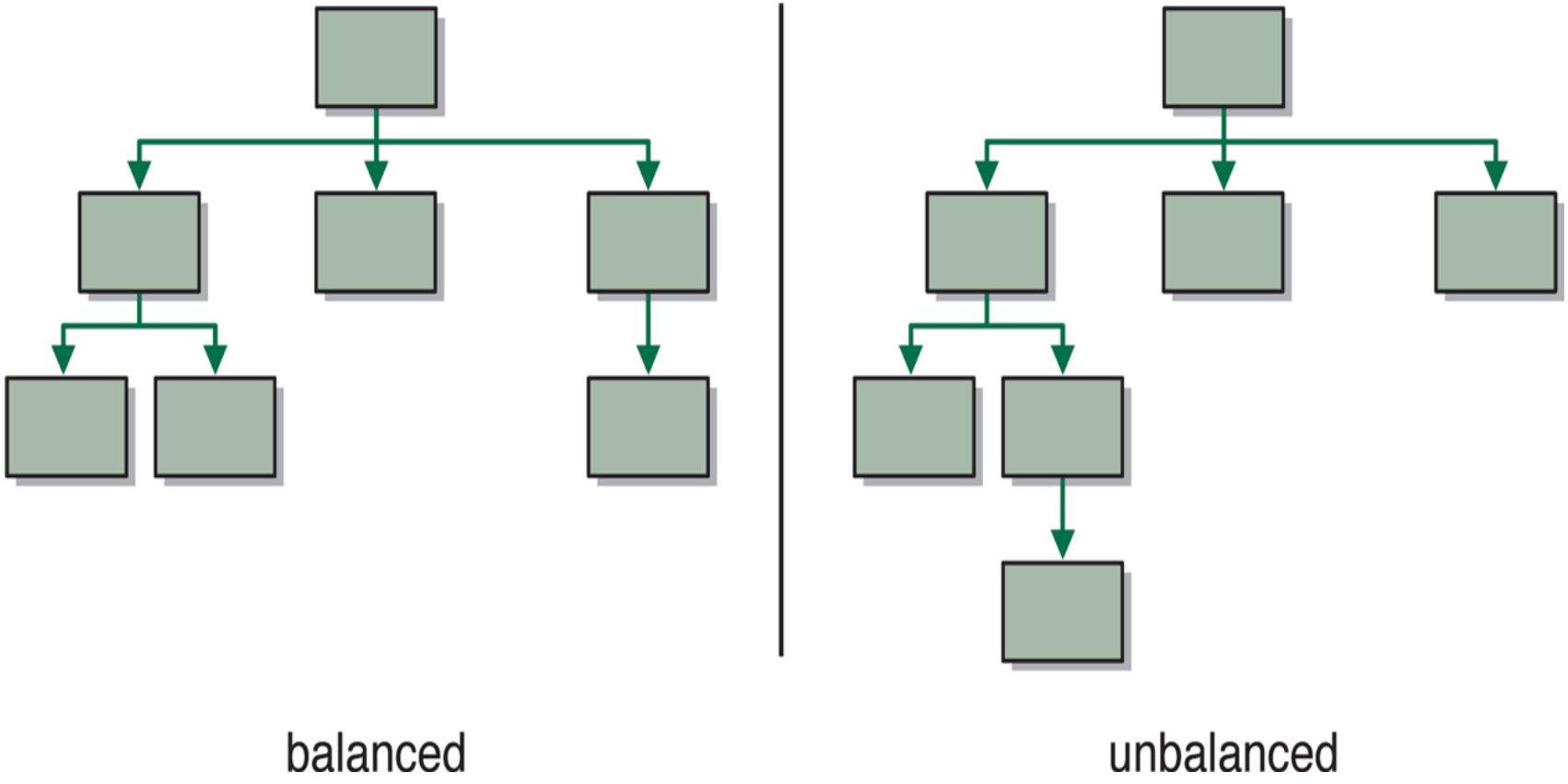


Definitions

- A tree is considered to be *balanced* if all of the leaves of the tree are at roughly the same depth
- While the use of the term “roughly” may not be intellectually satisfying, the actual definition is dependent upon the algorithm being used
- Some algorithms define balanced as all of the leaves being at level h or h-1 where h is the height of the tree (where $h = \log_N n$ for an N-ary tree)

Other algorithms use much less restrictive definitions.

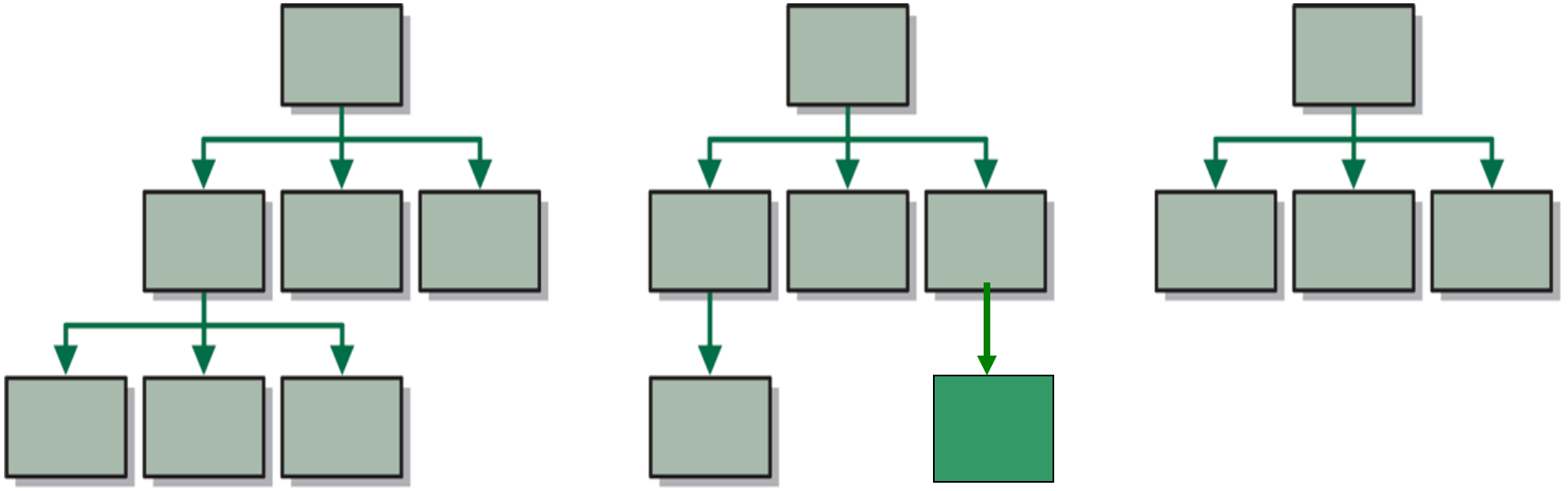
Balanced and unbalanced trees



Definitions

- The concept of a *complete* tree is related to the balance of a tree
- A tree is considered *complete* if it is balanced and all of the leaves at level h (i.e. the lowest level) are on the left side of the tree
- While a seemingly arbitrary concept, this definition has implications for how the tree is stored in certain implementations
- Trees a and c on the next slide are complete while tree b is not

Some complete trees



Implementing Trees with Links

- While it is not possible to discuss the details of an implementation of a tree without defining the type of tree and its use, we can look at general strategies for implementing trees

- The most obvious implementation of tree is a linked structure

- Each node could be defined as a `treeNode` structure, as we did with the `linearNode` class for linked lists

Implementing Trees with Links

- Each node would contain a pointer to the element to be stored in that node as well as pointers for each of the possible children of the node

- Depending on the implementation, it may also be useful to store a pointer in each node to its parent

Implementing Trees with Arrays

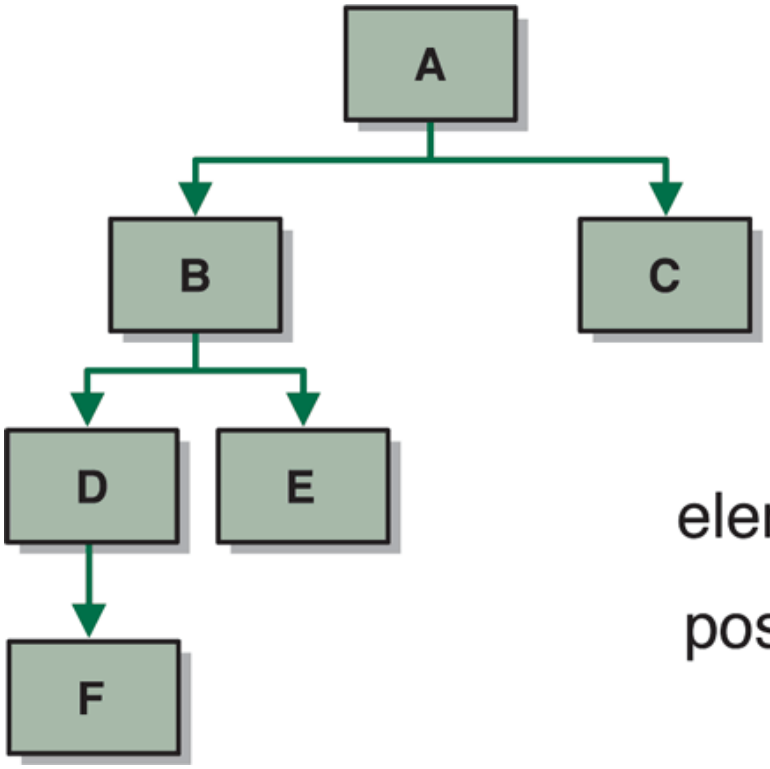
- For certain types of trees, specifically binary trees, **a computational strategy** can be used for storing a tree using an array

- For any element stored in position n of the array, that elements left child will be stored in position ((2*n) + 1) and that elements right child will be stored in position (2*(n+1))

Implementing Trees with Arrays

- This strategy can be managed in terms of capacity in much the same way that we did for other array-based collections
- Despite the conceptual elegance of this solution, it is not without drawbacks
- For example, if the tree that we are storing is not complete or relatively complete, we may be wasting large amounts of memory allocated in the array for positions of the tree that do not contain data

Computational strategy for array implementation of trees



element	A	B	C	D	E			F
position	0	1	2	3	4	5	6	7

Implementing Trees with Arrays

- A second possible array implementation of trees is modeled after the way operating systems manage memory
- Instead of assigning elements of the tree to array position by location in the tree, array positions are allocated contiguously on a first come first served basis
- Each element of the array will be a node class similar to the `treeNode` structure that we discussed earlier

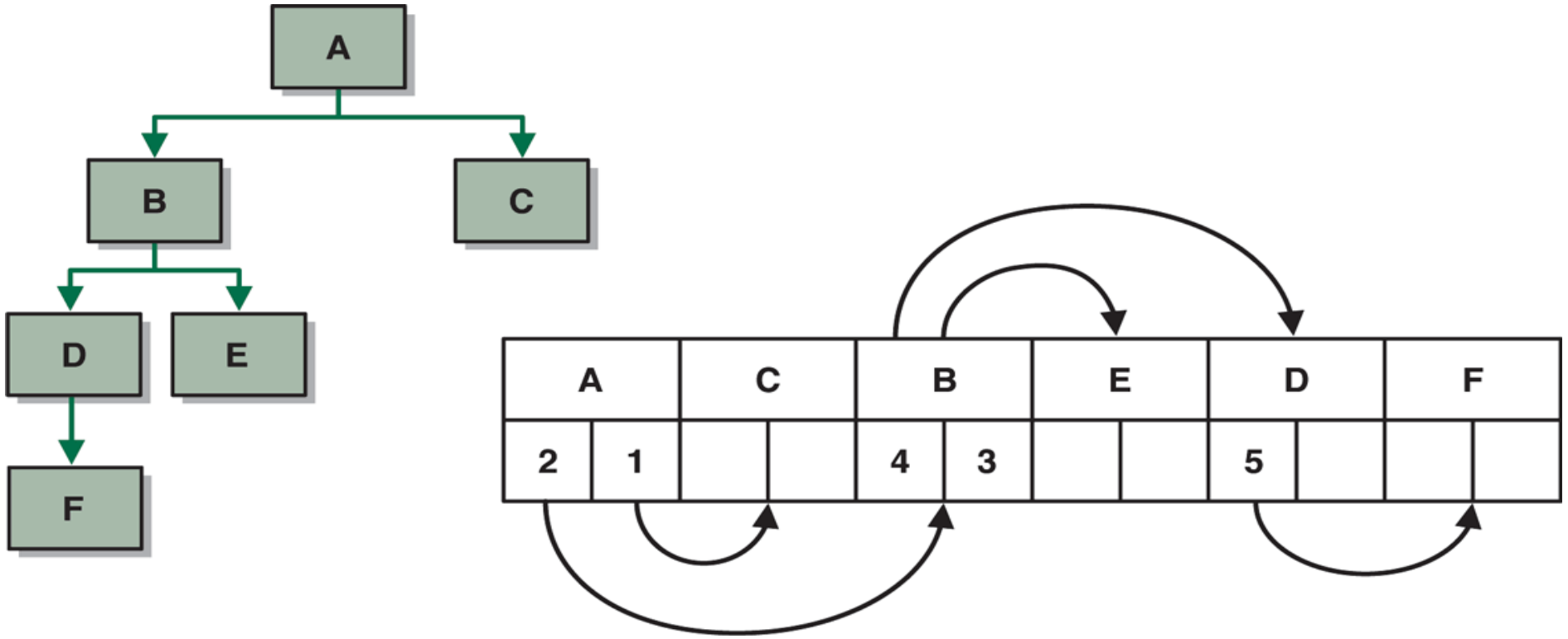
Implementing Trees with Arrays

- However, instead of storing node reference variables as pointers to its children (and perhaps its parent), each node would store the array index of each child (and perhaps its parent)

- This approach allows elements to be stored contiguously in the array so that space is not wasted

- However, this approach increases the overhead for deleting elements in the tree since either remaining elements will have to be shifted to maintain contiguity or a free-list will have to be maintained.

Simulated link strategy for array implementation of trees



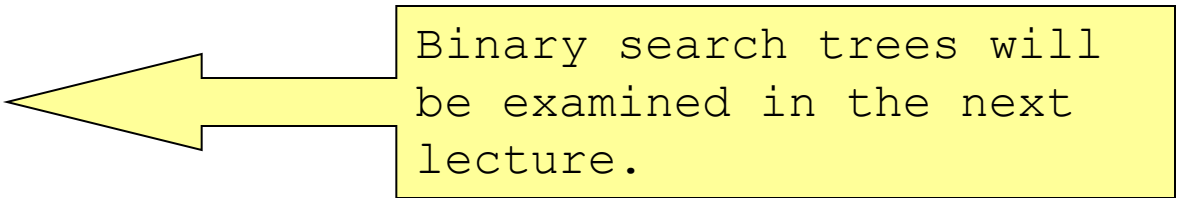
Analysis of Trees

- Trees are a useful and efficient way to store and search large numbers of items in memory. Typically there are more efficient than linear data structures.

- They can be used to implement other structures such as lists, stacks and queues.

- In our analysis of a linear search , we found it to be O(n)

- However, if we implemented the list using a balanced *binary search tree*, a binary tree with the added property that the left child is always less than the parent which is always less than or equal to the right child, then we could improve the efficiency of the algorithm to O(log n).



Analysis of Trees

- This is due to the fact that the height or order of such a tree will always be $\log_2 n$ where n is the number of elements in the tree
- This is very similar to our discussion of binary search in the last lecture.
- In fact, for any balanced N-ary tree with n elements, the tree's height will be $\log_N n$
- With the added ordering property of a binary search tree, you are guaranteed to at worst search one path from the root to a leaf

Tree Traversals

- There are three basic algorithms for traversing a tree:
  - Preorder traversal
  - Inorder traversal
  - Postorder traversal

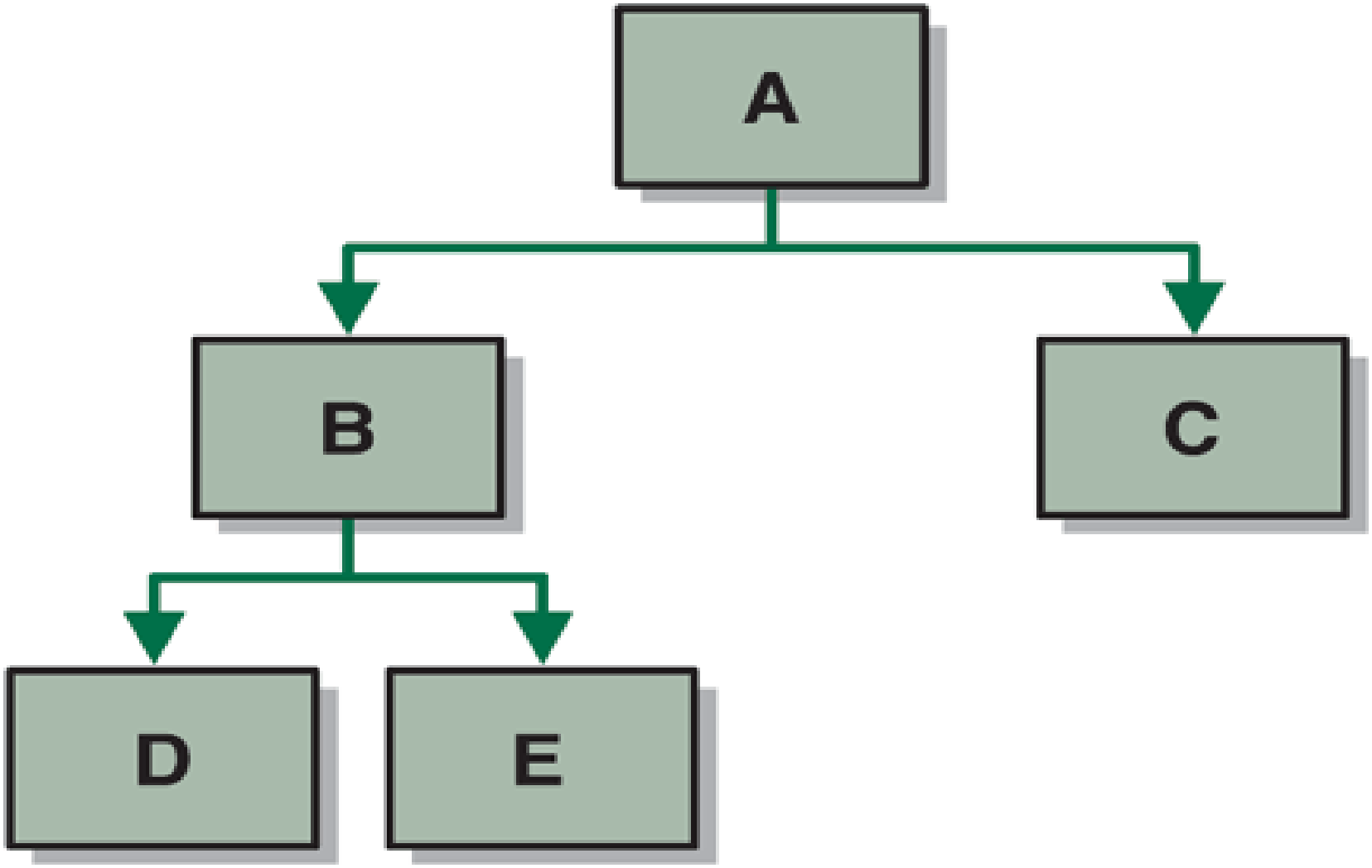
Tree traversal : visit and process in turn, every node in in the tree.

Preorder traversal

- Preorder traversal is accomplished by visiting each node, followed by its children, starting with the root

- Given the binary tree on the next slide, a preorder traversal would produce the order:

  A B D E C


A
B
C
D
E

Preorder traversal

- Stated in pseudocode, the algorithm for a preorder traversal of a binary tree is:

```
Traverse :
    Visit node
    Traverse(left child)
    Traverse(right child)
```

Inorder traversal

- Inorder traversal is accomplished by visiting the left child of the node, then the node, then any remaining child nodes starting with the root.

- An inorder traversal of the previous tree produces the order:

  D B E A C

Inorder traversal

- Stated in pseudocode, the algorithm for an inorder traversal of a binary tree is:

- Traverse :
  Traverse(left child)
  Visit node
  Traverse(right child)

Postorder traversal

- Postorder traversal is accomplished by visiting the children, then the node starting with the root

- Given the same tree, a postorder traversal produces the following order:

  D E B C A

Postorder traversal

- Stated in pseudocode, the algorithm for a postorder traversal of a binary tree is:

- Traverse :
  Traverse(left child)
  Traverse(right child)
  Visit node

Implementing Binary Trees

- As an example of possible implementations of trees, lets explore a simple implementation of a binary tree
- Having specified that we are implementing a binary tree, we can identify a set of possible operations that would be common for all binary trees
- Notice however, none of these operations add any elements to the tree
- It is not possible to define an operation to add an element to the tree until we know more about how the tree is to be used

Trees Operations

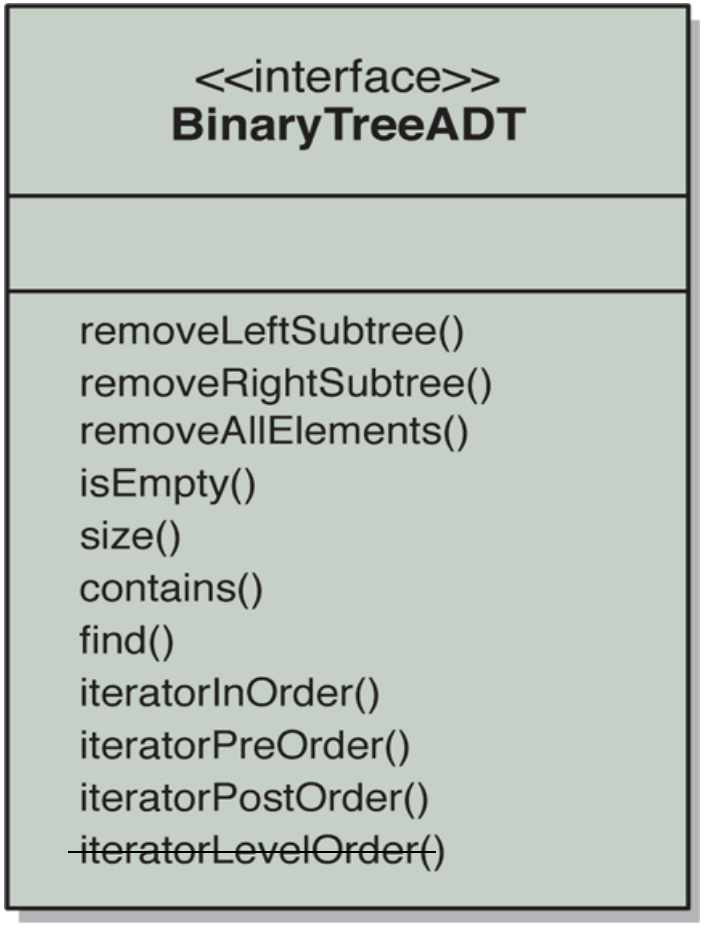
- Unlike lists where we were able to abstract a set of operations that were common for all list, it is virtually impossible to define a common set of operations for all trees.

- For trees, the operations are dependent upon the type of tree and it’s use. A tree is an abstract structure because it is a data structure that is not inherently defined within a programming language.

The operations on a binary tree

Operation	Description
`removeLeftSubtree`	Removes the left subtree of the root.
`removeRightSubtree`	Removes the right subtree of the root.
`removeAllElements`	Removes all of the elements from the tree.
`isEmpty`	Determines if the tree is empty.
`size`	Determines the number of elements in the tree.
`contains`	Determines if the specified target is in the tree.
`find`	Returns a reference to the specified target if it is found.
`tostring`	Returns a string representation of the tree.
`iteratorInOrder`	Returns an iterator for an inorder traversal of the tree.
`iteratorPreOrder`	Returns an iterator for a preorder traversal of the tree.
`iteratorPostOrder`	Returns an iterator for a postorder traversal of the tree.
`iteratorLevelOrder`	Returns an iterator for a level-order traversal of the tree.

UML description of the BinaryTreeADT interface



Note that this ADT is supplied using Java code. This is because C is a procedural language and as such does not support the idea of ADTs as interfaces.

Therefore the ADT here can only be used as a guide to help us implement a binary tree in C.

BinaryTreeADT

```
import java.util.Iterator;


public interface BinaryTreeADT {


 //  Should be implemented to remove the left subtree of the root
   //  of the binary tree.
   public void removeLeftSubtree();


//  Should be implemented to remove the right subtree of the root
//  of the binary tree.
   public void removeRightSubtree();
```

```
//  Should be implemented to remove all elements from the binary
    //  tree.
    public void removeAllElements();

    //  Should be implemented to return true if the binary tree is
    //  empty and false otherwise.
    public boolean isEmpty();

    //  Should be implemented to return the number of elements in the
    //  binary tree.
    public int size();

    //  Should be implemented to return true if the binary tree
    //  contains an element that matches the specified element and
    //  false otherwise.
    public boolean contains (Object targetElement);

    //  Should be implemented to return a reference to the specified
    //  element if it is found in the binary tree.
    public Object find (Object targetElement);

    //  Should be implemented to return the string representation of
    //  the binary tree.
    public String toString();
```

```
 //  Should be implemented to perform an inorder traversal on the
   //  binary tree by calling a recursive inorder method
   //  that starts with the root.
   public Iterator iteratorInOrder();

   //  Should be implemented to perform a preorder traversal on the
   //  binary tree by calling a recursive preorder
   //  method that starts with the root.
   public Iterator iteratorPreOrder();

   //  Should be implemented to perform a postorder traversal on the
   //  binary tree by calling a recursive postorder
   //  method that starts with the root.
   public Iterator iteratorPostOrder();


}  // end interface BinaryTreeADT
```

Linked BinaryTree

- Lets examine a linked implementation of a binary tree. Implementing this in C will require the use of pointers.

- Our implementation will need to keep track of the node that is the root of the tree as well as the count of elements in the tree

```
int treeCountounter = 0;

struct BinaryTreeNode *root = NULL;
```

Linked BinaryTree

- We will need a structure to represent each node in the tree.

- Since this is a binary tree, we will create a BinaryTreeNode ***structure*** that contain a reference to the element stored in the node as well as references for each of the children

```
struct BinaryTreeNode {
        struct data *element;
        struct BinaryTreeNode *left;
        struct BinaryTreeNode *right;
}

struct BinaryTreeNode *root = NULL;

int treeCounter = 0;
```

Linked BinaryTree

- Lets examine some of the functions required for a linked BinaryTree
- Keep in mind that each node of a tree represents a sub-tree

The removeLeftSubtree Method

```
//=================================================
   //  Removes the left subtree of this binary tree.
   //=================================================

   public void removeLeftSubtree()
   {
      if (root->left != NULL)
              treeCounter = treeCounter - leftNumChildren(root);

      root->left = NULL;

   }  // method removeLeftSubtree
```

The inOrder Method

- The inOrder function is called by passing in the root as the first parameter and then uses recursion to return the remaining items.

- The preOrder and post Order functions work in the same way.

The inorder Function

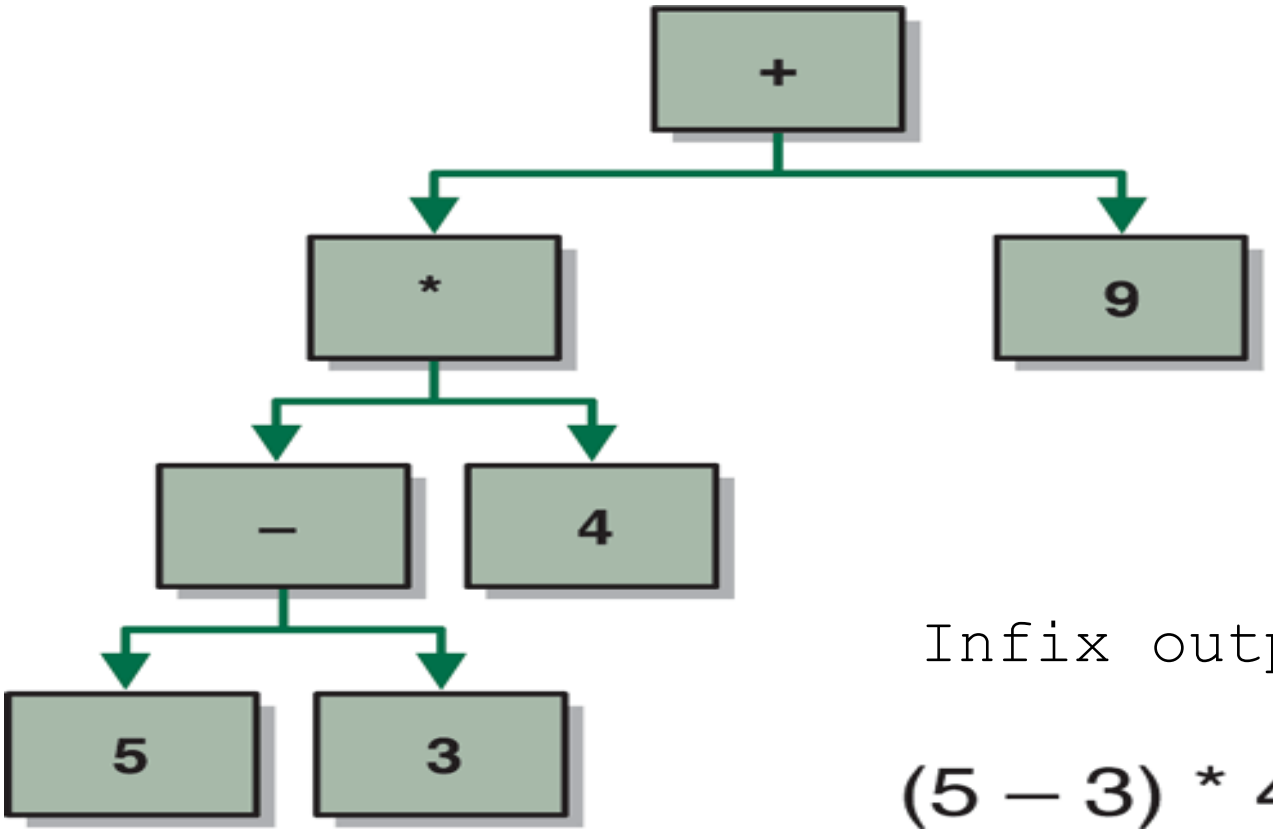
```
void inorder (struct BinaryTreeNode *node)
   {
      if (node != NULL)
        {
          inorder (node->left);
          displayOnScreen(node->data);
          inorder (node->right);
      }//if

   }  // method inorder
```

Expression Trees

- Now lets look at an example using a binary tree
- Last semester, we used a stack to evaluate postfix expressions
- Now we modify that algorithm to construct an expression tree (note that for the sake of simplicity, we will not evaluate the expression). The expression is input as a postfix expression and is output as an infix expression.

An example expression tree



Infix output :

$(5 - 3) * 4 + 9$

Postfix input into stack : 5 3 - 4 * 9 +

Expression Tree

The data structures used in this project are :-

1. Stack

When the expression is input, it is stored in a stack which is implemented as an array of characters.

2. Binary Tree

The characters are popped from the stack and are inserted into their correct position in the tree to produce the tree on the previous slide.

Expression Tree

- Assumptions
  - For the sake of simplicity, it is assumed that the numbers in the expression are between 0 and 9.
  - In the following code, for the sake of space, I haven't inserted the header files or the function prototypes. You will need to add these if you wish to implement this code.

How to create and process the expression

1. Add the postfix expression to the stack
2. Take the top element in the stack (i.e. '+')
   a) Construct the data part of the node (i.e. `Operator = '+'` and `isOperator = true`)
   b) Make the new `BinaryTreeNode` by calling `malloc` and add the data part to this new node. This node will be the `root` node.
3. Take the next top element from the stack (i.e. '9') and create a new node using steps a) and b) above. Make this new node the right child of the root.
4. Take the next top element from the stack (i.e. '*') and create a new node using steps a) and b) above. Make this new node the left child of the root.

5. Now the first 3 nodes are in the tree.

6. For the remaining nodes,
   a) take the next two top elements in turn from the stack
   b) find the left most element in the tree (parent node)
   c) make the first top element the right child of the parent node
   d) make the next top element the left child of the parent node.

Expression Trees: Data Structures

```
struct Node {
        char Operator;
        bool isOperator;
};

struct BinaryTreeNode
{
        struct Node data;
        struct BinaryTreeNode *left;
        struct BinaryTreeNode *right;
};


//Global Variables

int treeCounter = 0, stackCounter = 0, answer;

char stack[10];

struct BinaryTreeNode *root = NULL;
```

main() function

```
void main()
{
	struct Node aNode;
	struct BinaryTreeNode *parent, *leftChild, *rightChild;
	char anOperator;
	bool finished = false;
	int answer;

//input the postfix expression into the stack.
	inputExpressionForStack();
```

```
//move from stack into the binary tree
do
{
if (root == NULL) //get parent, its right and left children
   {
        if (stackNotIsEmpty()) {
               anOperator = stackPop();
               aNode = constructNode(anOperator);
               parent = createNode (aNode);
               root = parent;
        }
               else finished = true;

        if (stackNotIsEmpty()) {
               anOperator = stackPop();
               aNode = constructNode(anOperator);
               rightChild = createNode(aNode);
               parent -> right = rightChild;
        }
               else finished = true;
```

```
if (stackNotIsEmpty()) {
                    anOperator = stackPop();
                    aNode = constructNode(anOperator);
                    leftChild = createNode(aNode);
                    parent -> left = leftChild;
               }
                    else finished = true;
          }

// Now we have created the root node and its left and right
//children. The remaining code adds on any other subtrees
```

```
else //there are already 3 nodes in the tree
          {
                parent = findLeftMostNode(root);
                if (stackNotIsEmpty()) {
                      anOperator = stackPop();
                      aNode = constructNode(anOperator);
                      rightChild = createNode(aNode);
                      parent -> right = rightChild;
               }
                      else finished = true;

               if (stackNotIsEmpty()) {
                      anOperator = stackPop();
                      aNode = constructNode(anOperator);
                      leftChild = createNode(aNode);
                      parent -> left = leftChild;
               }
                      else finished = true;
          }//end else
         }while (finished == false);
//NOW DISPLAY THE CONTENT OF THE TREE AS A REGULAR MATHS
EXPRESSION
   displayInOrder(root);
}
```

Input Expression and push onto the Stack

```
void inputExpressionForStack()
{
   int index;   char anOperator;

//make stack
       for (index = 0; index < STACKSIZE; index++)
              stack[index] = ' ';

       printf("Enter a new postfix expression : ");
       printf("Enter the integer or operator (+,-,*,/) on
separate lines. Note that symbol ! will quit\n");

       do
       {
              anOperator = getchar();
              fflush(stdin);
              if (anOperator != '!') stackPush(anOperator);
       }while (anOperator != '!');
}
```

Stack Operations

```
void stackPush(char aCharacter)
{
	stack[stackCounter++] = aCharacter;
}

char stackPop ()
{
	char top;
	if (stack[stackCounter] == ' ') stackCounter--;
	top = stack[stackCounter--];
	return top;
}

bool stackNotIsEmpty()
{
	if (stackCounter < 0)
		return false;
	else
		return true;
}
```

Input the data part of the node

```
struct Node constructNode (char aCharacter) {
        struct Node temp;
         temp.Operator = aCharacter;
        if ((aCharacter == '+') || (aCharacter == '-') ||
(aCharacter == '*') || (aCharacter == '/'))
          temp.isOperator = true;
  else
          temp.isOperator = false;
  return temp;
}
```

Create a new node for the tree

```
struct BinaryTreeNode *createNode (struct Node aNode) {
	struct BinaryTreeNode *pointer;

	pointer = (struct BinaryTreeNode *)malloc(sizeof
(struct BinaryTreeNode));
	pointer->data = aNode;
	pointer->left = NULL;
	pointer->right = NULL;

	if (isempty()) //need to assign the root pointer
		root = pointer;
	treeCounter++;

	return pointer;
}
```

Check if tree is empty and the display function

```
bool isempty() {
          return (treeCounter==0);
   }


void displayInOrder (struct BinaryTreeNode *tree)
{
       if (tree != NULL)
       {
              displayInOrder(tree->left);
              printf(" %c ", tree->data.Operator);
              displayInOrder(tree->right);
       }
}
```

Find the address of the leftmost node in the tree

```
struct BinaryTreeNode *findLeftMostNode (struct BinaryTreeNode
*pointer) {
	struct BinaryTreeNode *temp;


	if (pointer->left == NULL)
		return pointer;
	else
		temp = findLeftMostNode(pointer->left);
}
```