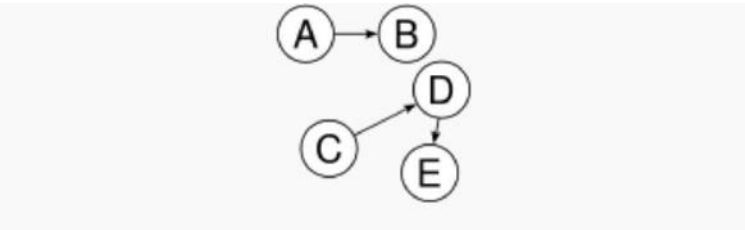
Module-4

Nonlinear Data Structures

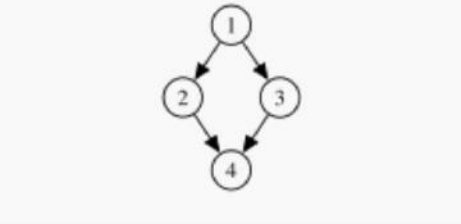
Teaching Hours: 10

Trees

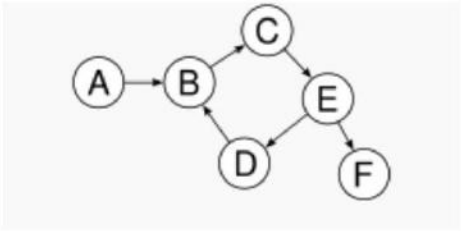
A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy



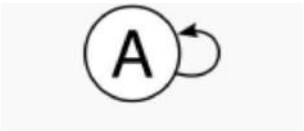
Not a tree: two non-connected parts, A→B and C→D→E



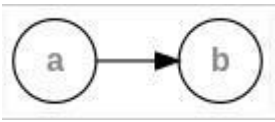
Not a tree: undirected cycle 1-2-4-3



Not a tree: cycle B→C→E→D→B



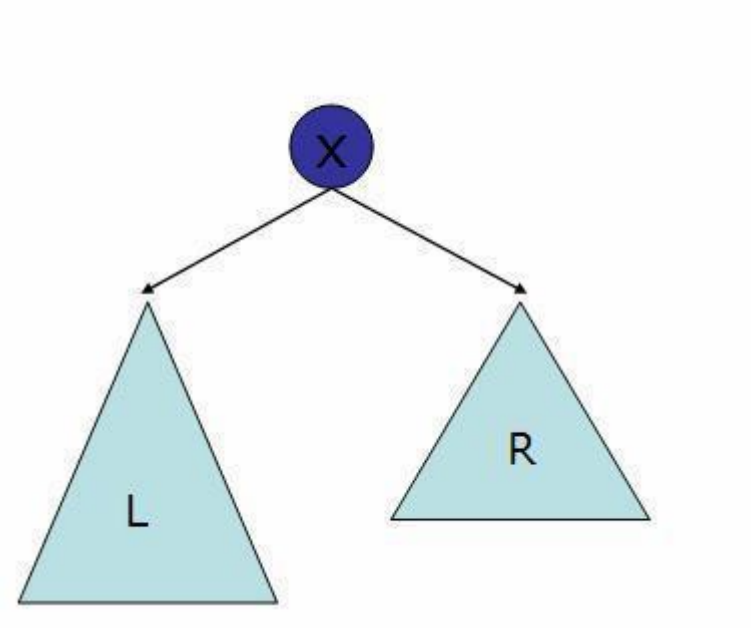
Not a tree: cycle A→A



1. ***Types***

Binary Trees

We will see that dealing with **binary** trees, a tree where each node can have no more than two children is a good way to understand trees.



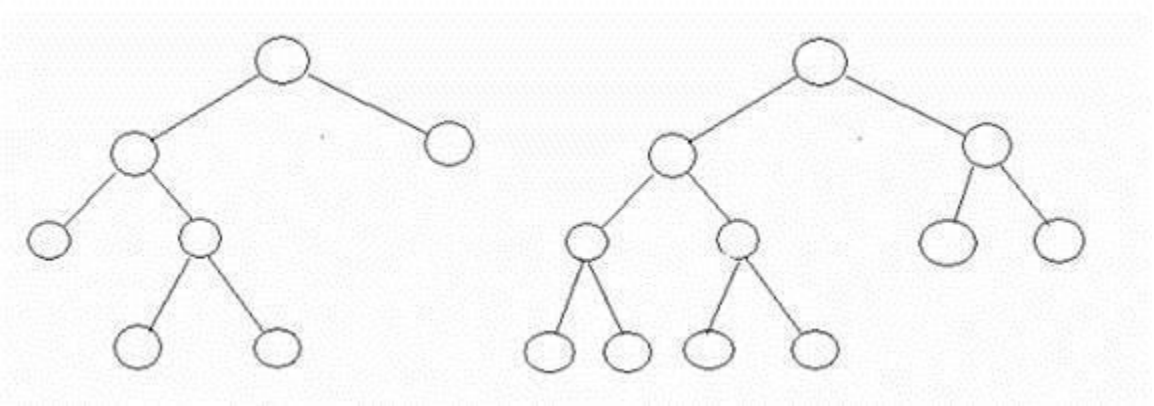
Here is a Java prototype for a tree node:

```
public class BNode
{
  private Object data;
  private BNode left, right;
  public BNode()
  {
    data=left=right=null;
  }
  public BNode(Object data)
  {
    this.data=data;
    left=right=null;
  }
}
```

A binary tree in which each node has exactly zero or two children is called **a full binary tree**. In a full tree, there are no nodes with exactly one child.

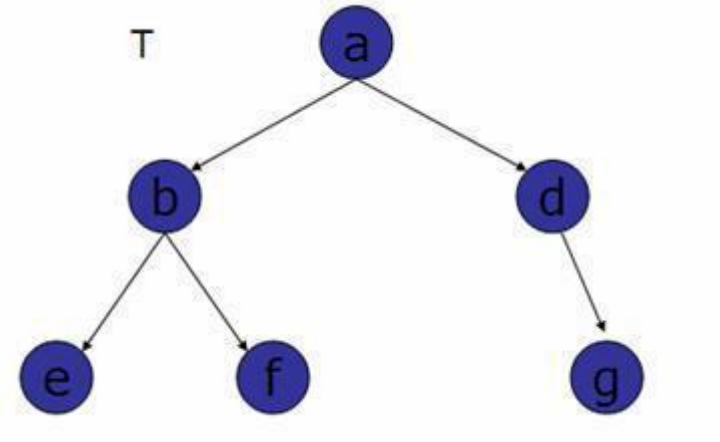
A complete binary tree is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree of the height h has between 2^h and $2^{(h+1)}$-1 nodes. Here are some examples:

full tree **complete tree**



Binary Search Trees

Given a binary tree, suppose we visit each node (recursively) as follows. We visit left child, then root and then the right child. For example, visiting the following tree



In the order defined above will produce the sequence {e, b,f,a,d,g} which we call flat(T). A binary search tree (BST) is a tree, where flat(T) is an ordered sequence. In other words, a binary search tree can be ―searched‖ efficiently using this ordering property. A ―balanced‖ binary search tree can be searched in O(log n) time, where n is the number of nodes in the tree.

2. Traversal methods

Trees can be traversed in *pre-order*, *in-order*, or *post-order*.[1] These searches are referred to as *depth-first search* (DFS), as the search tree is deepened as much as possible on each child before going to the next sibling.

For a binary tree, they are defined as display operations recursively at each node, starting with the root, whose algorithm is as follows:

The general recursive pattern for traversing a (non-empty) binary tree is this: At node N you must do these three things:

(L) recursively traverse its left subtree. When this step is finished you are back at N again.

(R) recursively traverse its right subtree. When this step is finished you are back at N again.

(N) Actually process N itself.

We may do these things *in any order* and still have a legitimate traversal. If we do (L) before (R), we call it left-to-right traversal, otherwise we call it right-to-left traversal.

Pre-order

1. Display the data part of the root (or current node).
2. Traverse the left subtree by recursively calling the pre-order function.
3. Traverse the right subtree by recursively calling the pre-order function.

In-order

1. Traverse the left subtree by recursively calling the in-order function.
2. Display the data part of the root (or current node).
3. Traverse the right subtree by recursively calling the in-order function.

In a search tree, in-order traversal retrieves data in sorted order.

Post-order

1. Traverse the left subtree by recursively calling the post-order function.
2. Traverse the right subtree by recursively calling the post-order function.
3. Display the data part of the root (or current node).

The trace of a traversal is called a sequentialisation of the tree. The traversal trace is a list of each visited root. No one sequentialisation according to pre-, in- or post-order describes the underlying tree uniquely. Given a tree with distinct elements, either pre-order or post-order paired with in-order is sufficient to describe the tree uniquely. However, pre-order with post-order leaves some ambiguity in the tree structure.

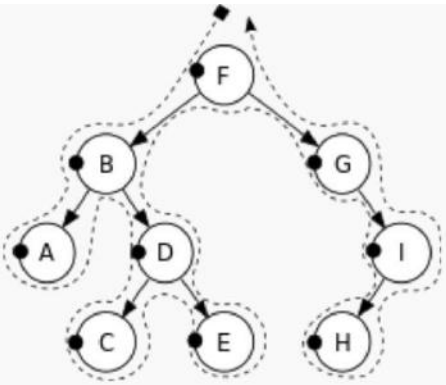
Generic tree

To traverse any tree with depth-first search, perform the following operations recursively at each node:

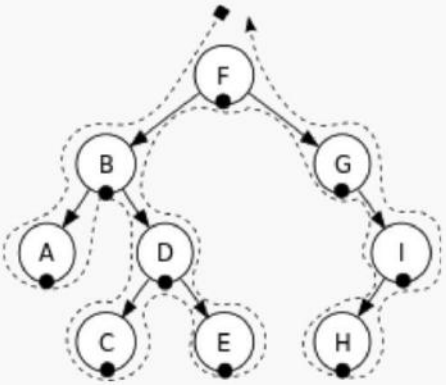
1. Perform pre-order operation.
2. For each *i* from 1 to the number of children do:
    1. Visit *i*-th, if present.

2. Perform in-order operation.
3. Perform post-order operation.

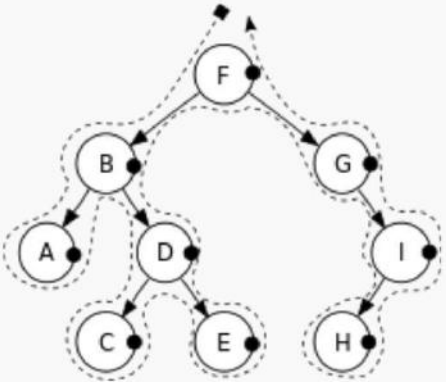
Depending on the problem at hand, the pre-order, in-order or post-order operations may be void, or you may only want to visit a specific child, so these operations are optional. Also, in practice more than one of pre-order, in-order and post-order operations may be required. For example, when inserting into a ternary tree, a pre-order operation is performed by comparing items. A post-order operation may be needed afterwards to re-balance the tree.



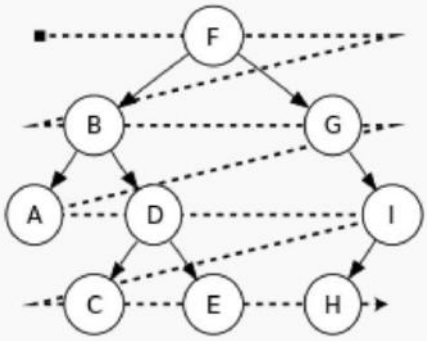
Pre-order: F, B, A, D, C, E, G, I, H.



In-order: A, B, C, D, E, F, G, H, I.



Post-order: A, C, E, D, B, H, I, G, F.



Level-order: F, B, G, A, D, I, C, E, H

3. Binary Search Trees

Binary search trees (**BST**), sometimes called **ordered** or **sorted binary trees**, are a particular type of containers: data structures that store "items" (such as numbers, names etc.) in memory. They allow fast lookup, addition and removal of items, and can be used to implement either dynamic sets of items, or lookup tables that allow finding an item by its *key* (e.g., finding the phone number of a person by name).

Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, based on the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array, but slower than the corresponding operations on hash tables.

Several variants of the binary search tree have been studied in computer science; this article deals primarily with the basic type, making references to more advanced types when appropriate.

Operations

Binary search trees support three main operations: insertion of elements, deletion of elements, and lookup (checking whether a key is present).

Searching

Searching a binary search tree for a specific key can be a programmed recursively or iteratively.

We begin by examining the root node. If the tree is *null*, the key we are searching for does not exist in the tree. Otherwise, if the key equals that of the root, the search is successful and we return the node. If the key is less than that of the root, we search the left subtree. Similarly, if the key is greater than that of the root, we search the right subtree. This process is repeated until the key is found or the remaining subtree is *null*. If the searched key is not found before a *null* subtree is reached, then the key is not present in the tree. This is easily expressed as a recursive algorithm:

```
1 def search_recursively(key, node):
2     if node is None or node.key == key:
3         return node
4     elif key < node.key:
5         return search_recursively(key, node.left)
6     else:  # key > node.key
7         return search_recursively(key, node.right)
```

The same algorithm can be implemented iteratively:

```
 1 def search_iteratively(key, node):
 2     current_node = node
 3     while current_node is not None:
 4         if key == current_node.key:
 5             return current_node
 6         elif key < current_node.key:
 7             current_node = current_node.left
 8         else:  # key > current_node.key:
 9             current_node =
current_node.right 10 return None
```

These two examples rely on the order relation being a total order.

If the order relation is only a total preorder a reasonable extension of the functionality is the following: also in case of equality search down to the leaves in a direction specifiable by the user. A binary tree sort equipped with such a comparison function becomes stable.

Because in the worst case this algorithm must search from the root of the tree to the leaf farthest from the root, the search operation takes time proportional to the tree's *height* (see tree terminology). On average, binary search trees with n nodes have O(log n) height.[a] However, in the worst case, binary search trees can haveO(n) height, when the unbalanced tree resembles a linked list (degenerate tree).

Insertion

Insertion begins as a search would begin; if the key is not equal to that of the root, we search the left or right subtrees as before. Eventually, we will reach an external node and add the new key-value pair (here encoded as a record 'newNode') as its right or left child, depending on the node's key. In other words, we examine the root and recursively insert the new node to the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root.

Here's how a typical binary search tree insertion might be performed in a binary tree in C++:

```
void insert(Node*& root, int key, int value) {
  if (!root)
    root = new Node(key, value);
  else if (key < root->key)
    insert(root->left, key, value);
  else // key >= root->key
    insert(root->right, key, value);
}
```

The above *destructive* procedural variant modifies the tree in place. It uses only constant heap space (and the iterative version uses constant stack space as well), but the prior version of the tree is lost. Alternatively, as in the following Python example, we can reconstruct all ancestors of the inserted node; any reference to the original tree root remains valid, making the tree a persistent data structure:

```
 def binary_tree_insert(node, key, value):
     if node is None:
         return NodeTree(None, key, value, None)
     if key == node.key:
         return NodeTree(node.left, key, value, node.right)
     if key < node.key:
         return NodeTree(binary_tree_insert(node.left, key, value), node.key, node.value, node.right)
     else:
         return NodeTree(node.left, node.key, node.value, binary_tree_insert(node.right, key, value))
```

The part that is rebuilt uses O(log *n*) space in the average case and O(*n*) in the worst case.

In either version, this operation requires time proportional to the height of the tree in the worst case, which is O(log *n*) time in the average case over all trees, but O(*n*)time in the worst case.

Another way to explain insertion is that in order to insert a new node in the tree, its key is first compared with that of the root. If its key is less than the root's, it is then compared with the key of the root's left child. If its key is greater, it is compared with the root's right child. This process continues, until the new node is compared with

a leaf node, and then it is added as this node's right or left child, depending on its key: if the key is less than the leaf's key, then it is inserted as the leaf's left child, otherwise as the leaf's right child.

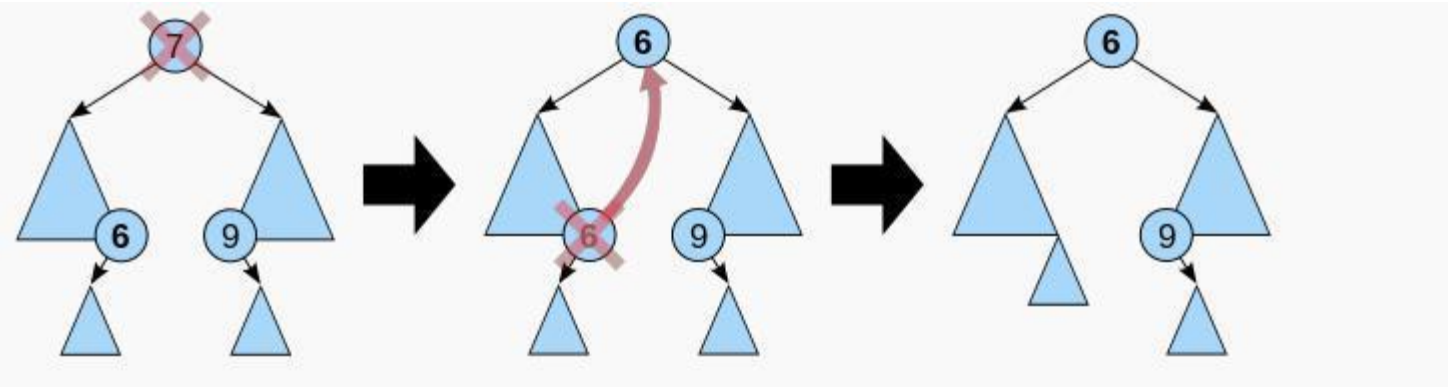
There are other ways of inserting nodes into a binary tree, but this is the only way of inserting nodes at the leaves and at the same time preserving the BST structure.

Deletion

There are three possible cases to consider:

- Deleting a node with no children: simply remove the node from the tree.
- Deleting a node with one child: remove the node and replace it with its child.
- Deleting a node with two children: call the node to be deleted *N*. Do not delete *N*. Instead, choose either its in-order successor node or its in-order predecessor node, *R*. Copy the value of *R* to *N*, then recursively call delete on *R* until reaching one of the first two cases. If you choose in-order successor of a node, as right sub tree is not NIL (Our present case is node has 2 children), then its in-order successor is node with least value in its right sub tree, which will have at a maximum of 1 sub tree, so deleting it would fall in one of the first 2 cases.

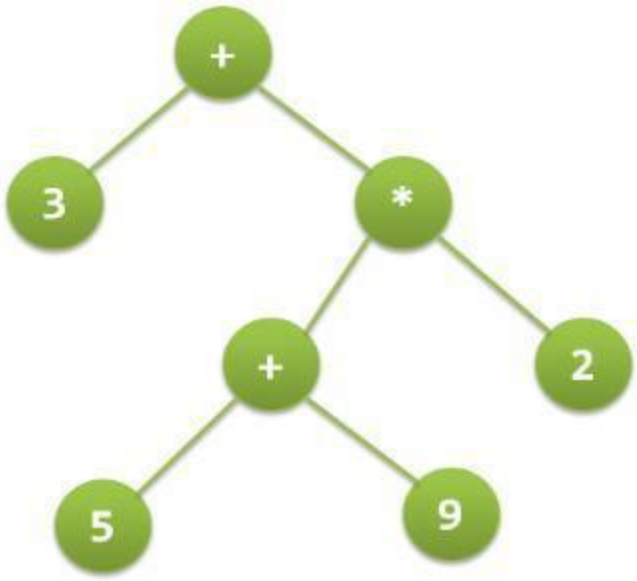
Broadly speaking, nodes with children are harder to delete. As with all binary trees, a node's in-order successor is its right subtree's left-most child, and a node's in-order predecessor is the left subtree's right-most child. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.



Deleting a node with two children from a binary search tree. First the rightmost node in the left subtree, the inorder predecessor **6**, is identified. Its value is copied into the node being deleted. The inorder predecessor can then be easily deleted because it has at most one child. The same method works symmetrically using the inorder successor labelled **9**

4. Expression tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree:

```
Let t be the expression tree
If  t is not null then
    If t.value is operand then
        Return t.value
    A = solve(t.left)
    B = solve(t.right)

    // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

Construction of Expression Tree:

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

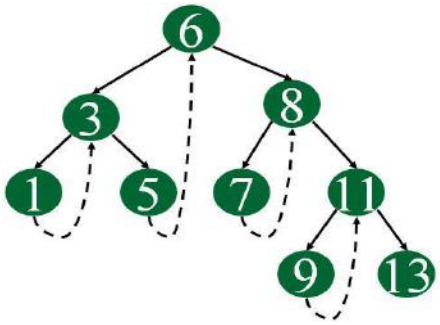
1) If character is operand push that into stack
2) If character is operator pop two values from stack make them its child and push current node again. At the end only element of stack will be root of expression tree.

5. Threaded binary tree

Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A

binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).
There are two types of threaded binary trees.
Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.
The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



C representation of a Threaded Node
Following is C representation of a single threaded node.

```
struct Node

{

   int data;

   Node *left, *right;

   bool rightThread;

}
```

Run on IDE

Since right pointer is used for two purposes, the boolean variable rightThread is used to indicate whether right pointer points to right child or inorder successor. Similarly, we can add leftThread for a double threaded binary tree.

Inorder Taversal using Threads

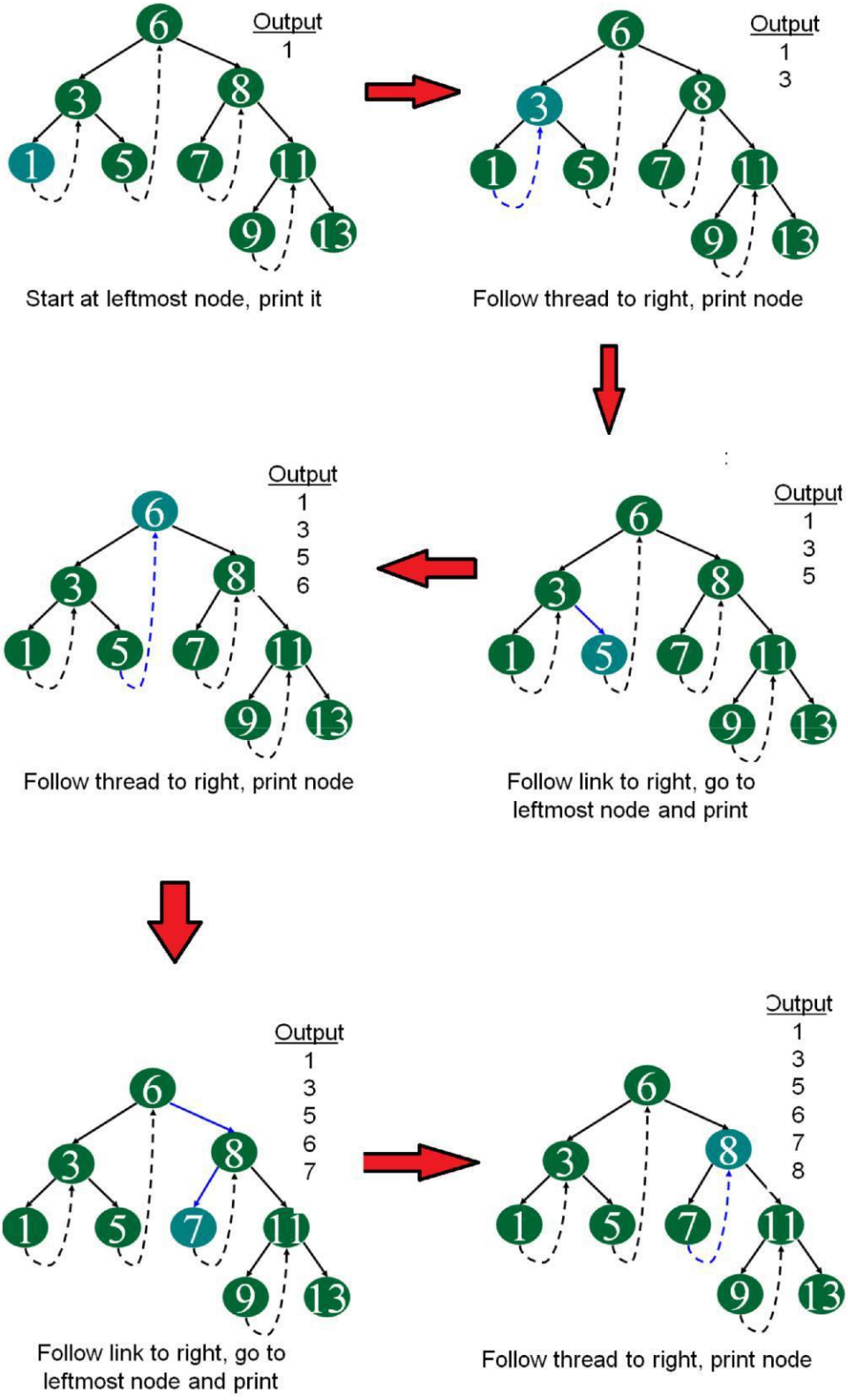
Following is C code for inorder traversal in a threaded binary tree.

```
// Utility function to find leftmost node in atree rooted with n

struct Node* leftMost(struct Node *n)

{

   if (n == NULL)
```

```
    return NULL;

  while (n->left != NULL)
     n = n->left;

  return n;
}

// C code to do inorder traversal in a threadded binary
tree void inOrder(struct Node *root)
{
  struct Node *cur = leftmost(root);
  while (cur != NULL)
  {
     printf("%d ", cur->data);

     // If this node is a thread node, then go to
     // inorder successor
     if (cur->rightThread)
        cur = cur->right;
     else // Else go to the leftmost child in right subtree
        cur = leftmost(cur->right);
  }
}
```

Run on IDE

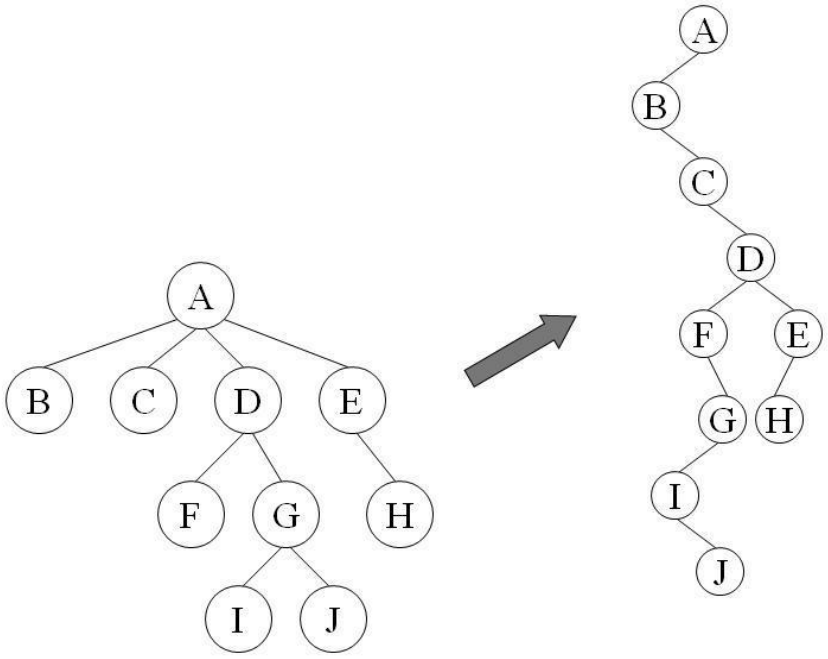
Following diagram demonstrates inorder order traversal using threads.



continue same way for remaining node.....

6. Conversion of General Trees to Binary Trees

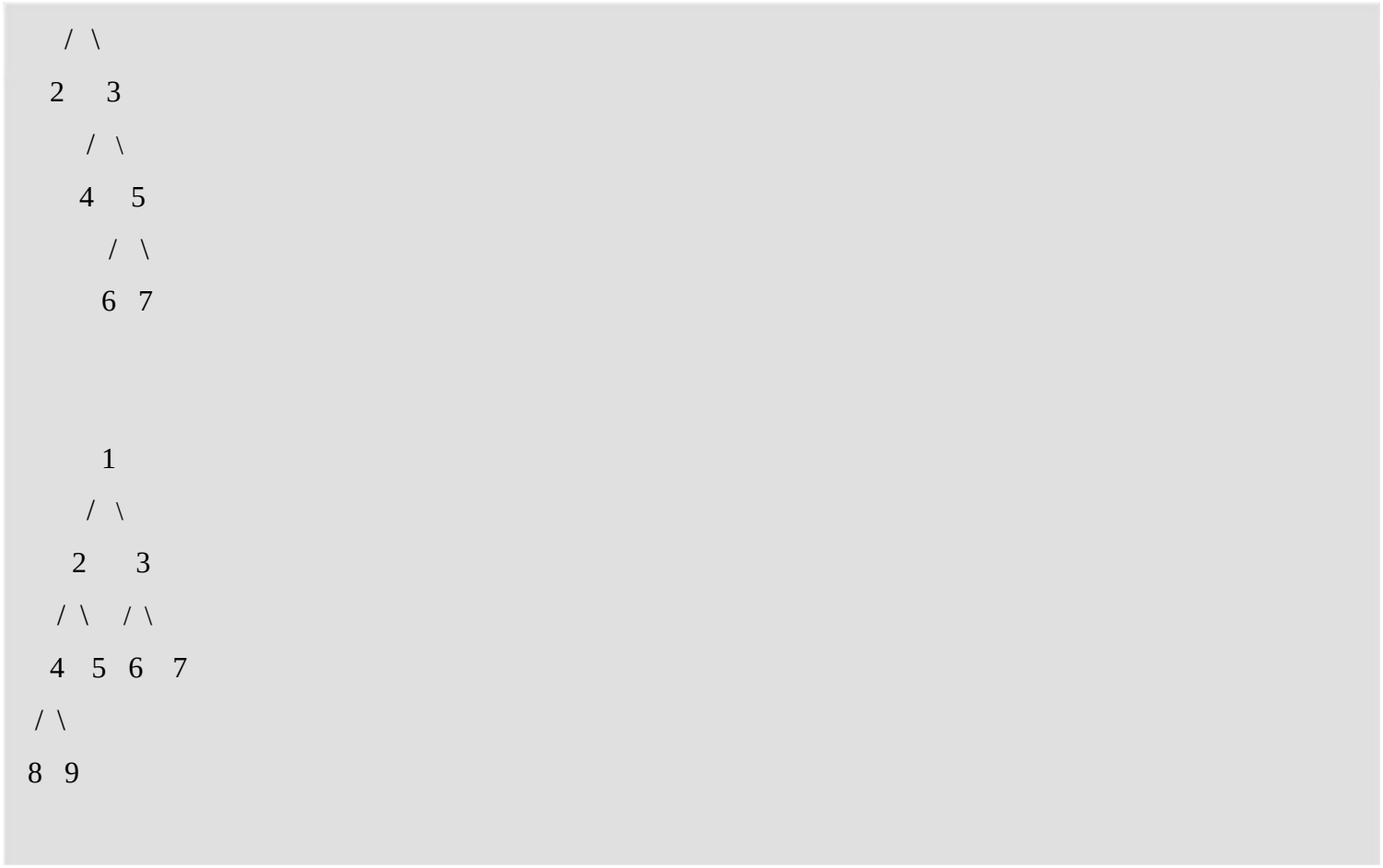
- ***Binary tree left child = leftmost child***
- ***Binary tree right child = right sibling***



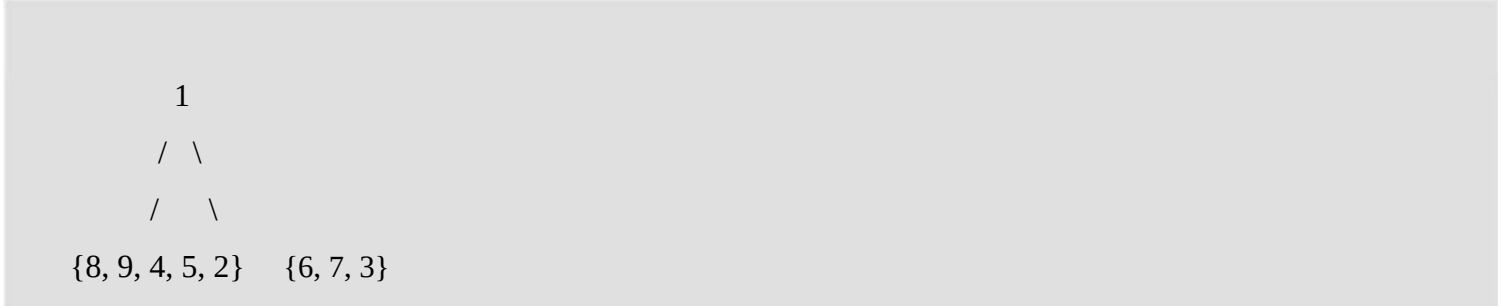
7. Constructing BST from traversal orders

Given two arrays that represent preorder and postorder traversals of a full binary tree, construct the binary tree. A **Full Binary Tree** is a binary tree where every node has either 0 or 2 children Following are examples of Full Trees.

```
     1
   /  \
  2     3
 / \   / \
4  5  6   7



    1
```



It is not possible to construct a general Binary Tree from preorder and postorder traversals (See this). But if know that the Binary Tree is Full, we can construct the tree without ambiguity. Let us understand this with the help of following example.
Let us consider the two given arrays as pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7} and post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1}; In pre[], the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in pre[], must be left child of root. So we know 1 is root and 2 is left child. How to find the all nodes in left subtree? We know 2 is root of all nodes in left subtree. All nodes before 2 in post[] must be in left subtree. Now we know 1 is root, elements {8, 9, 4, 5, 2} are in left subtree, and the elements {6, 7, 3} are in right subtree.



We recursively follow the above approach and get the following tree.

```
      /  \
     2    3
    / \   / \
   4  5  6   7
  /\
 8 9
```

```
/* program for construction of full binary tree */
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

// A utility function to create a node
struct node* newNode (int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
```

```
    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// A recursive function to construct Full from pre[] and post[].
// preIndex is used to keep track of index in pre[].
// l is low index and h is high index for the current subarray in post[]
struct node* constructTreeUtil (int pre[], int post[], int* preIndex,
                                int l, int h, int size)
{
    // Base case
    if (*preIndex >= size || l > h)
        return NULL;

    // The first node in preorder traversal is root. So take the node at
    // preIndex from preorder and make it root, and increment
    preIndex struct node* root = newNode ( pre[*preIndex] );
    ++*preIndex;

    // If the current subarry has only one element, no need to recur
    if (l == h)
        return root;
```

```
    // Search the next element of pre[] in
    post[] int i;
    for (i = l; i <= h; ++i)
        if (pre[*preIndex] ==
            post[i]) break;

    // Use the index of element found in postorder to divide postorder array in
    // two parts. Left subtree and right subtree
    if (i <= h)
    {
        root->left = constructTreeUtil (pre, post, preIndex, l, i, size); root-
        >right = constructTreeUtil (pre, post, preIndex, i + 1, h, size);
    }

    return root;
}

// The main function to construct Full Binary Tree from given preorder and
// postorder traversals. This function mainly uses constructTreeUtil()
struct node *constructTree (int pre[], int post[], int size)
{
    int preIndex = 0;
    return constructTreeUtil (pre, post, &preIndex, 0, size - 1, size);
}
```

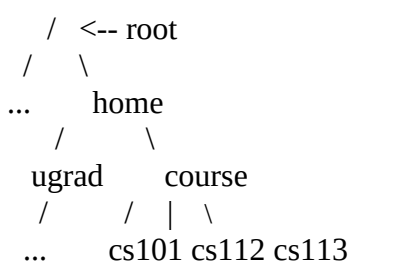
```
// A utility function to print inorder traversal of a Binary Tree
void printInorder (struct node* node)
{
    if (node ==
        NULL) return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7};
    int post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};
    int size = sizeof( pre ) / sizeof( pre[0] );

    struct node *root = constructTree(pre, post, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}
```

8. Applications of Trees

Why Tree?
Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system
———



2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for insertion/deletion.

4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don‘t have an upper limit on number of nodes as nodes are linked using pointers.

As per Wikipedia, following are the common uses of tree.
1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

9. Evaluation of Expression

Consider arithmetic expressions like *(35 - 3*(3+2)) / 4*:

For our discussion we'll consider the four arithmetic operators: +, -, *, /.

We'll study *integer arithmetic*

Here, the /-operator is integer-division (e.g., 2/4 = 0)

Each operator is a *binary* operator

Takes two numbers and returns the result (a number)

There is a natural precedence among operators: \, *, +, -.

We'll use parentheses to force a different order, if needed:

Thus *35 - 3*3+2/4 = 26* (using integer arithmetic)

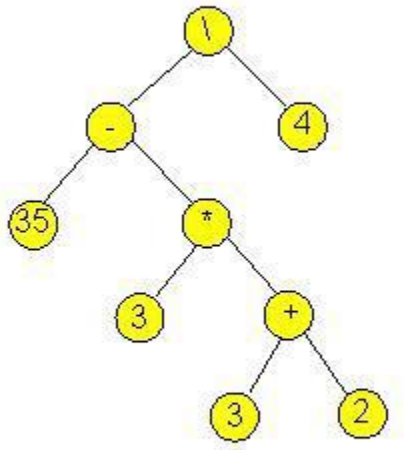
Whereas *(35 - 3*(3+2)) / 4 = 5*

Now let's examine a *computational procedure* for expressions:

At each step, we apply one of the operators.

The rules of precedence and parentheses tell us the order.

The computational procedure can be written as an expression tree:



In an expression tree:

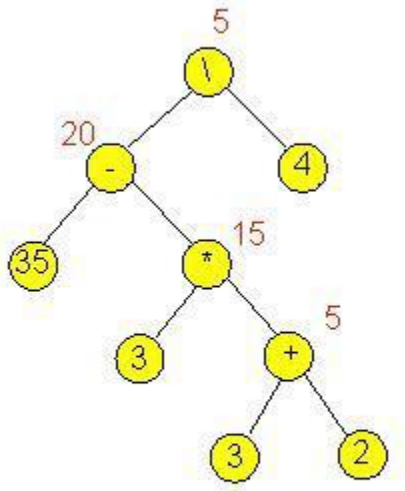
The leaves are numbers (the operands).

The *value* of a leaf is the number.

The non-leaf nodes are operators.

The value of a non-leaf node is the result of the operator applied to the values of the left and right child.

The root node corresponds to the final value of the expression.



10. Tree based Sorting

A binary search tree can be used to implement a simple sorting algorithm. Similar to heapsort, we insert all the values we wish to sort into a new ordered data structure—in this case a binary search tree—and then traverse it in order.

The worst-case time of build_binary_tree is O(n^2)—if you feed it a sorted list of values, it chains them into a linked list with no left subtrees. For example, build_binary_tree([1, 2, 3, 4, 5]) yields the tree (1 (2 (3 (4 (5))))) .

There are several schemes for overcoming this flaw with simple binary trees; the most common is the self-balancing binary search tree. If this same procedure is done using such a tree, the overall worst-case time is O(*n* log *n*), which is asymptotically optimal for a comparison sort. In practice, the added overhead in time and space for a tree-based sort (particularly for node allocation) make it inferior to other asymptotically optimal

sorts such as heapsort for static list sorting. On the other hand, it is one of the most efficient methods of *incremental sorting*, adding items to a list over time while keeping the list sorted at all times.