Lecture 9
Data Structures and Algorithms

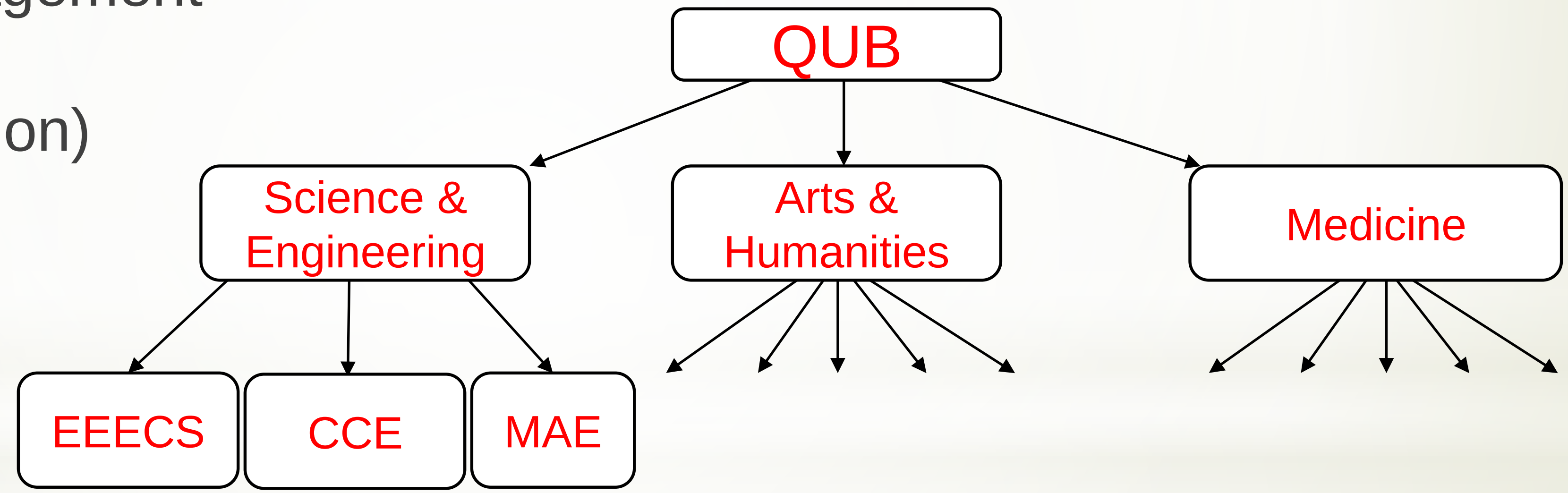
- Tree as data structures
- Tree terminology
- Binary tree and implementation in C++
- Recursive properties of a tree

Trees as Data Structures

Trees are useful for representing hierarchical structure in real life applications:

Example 1

The hierarchical management structure of a system (e.g. a large organisation)



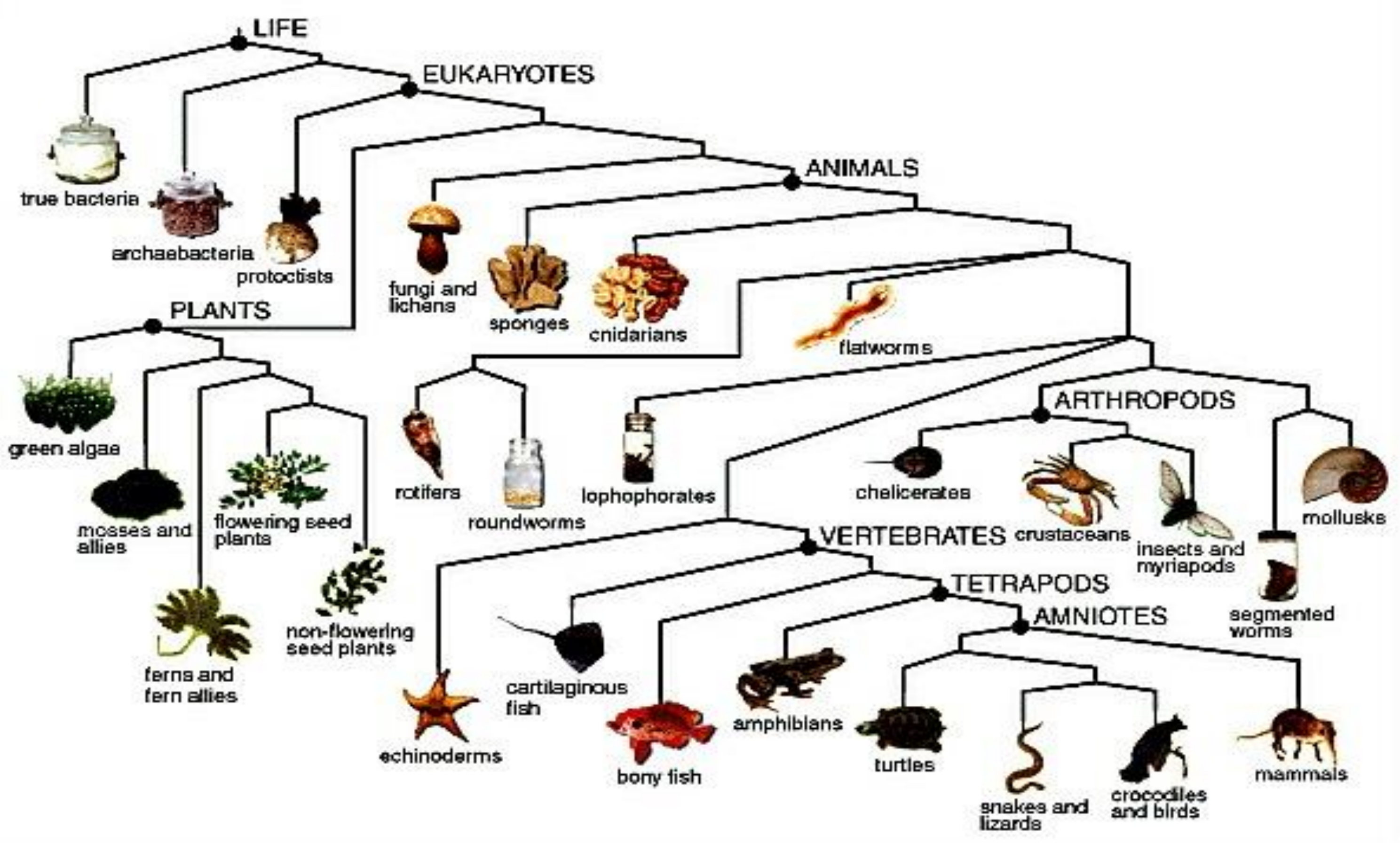
Trees as Data Structures

Trees are useful for representing hierarchical structure in real life applications:

Example 2

The hierarchical classification of objects with some concept of inheritance of properties

(e.g.
- tree of life;
- Class hierarchies in an OO program with inheritance)

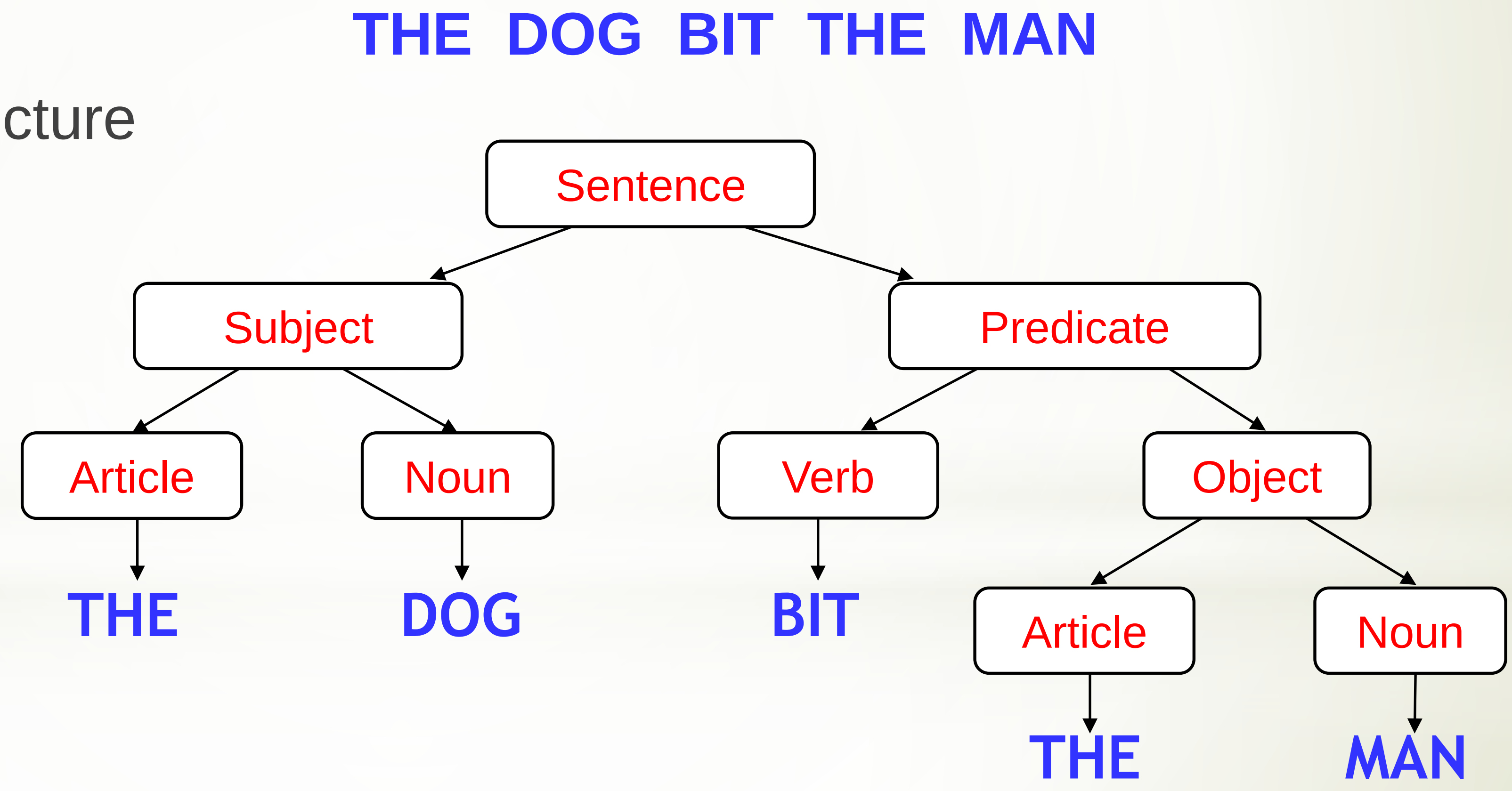


Trees as Data Structures

Trees are useful for representing hierarchical structure in real life applications:

Example 3

The unseen language structure of linear text (natural or computer languages)

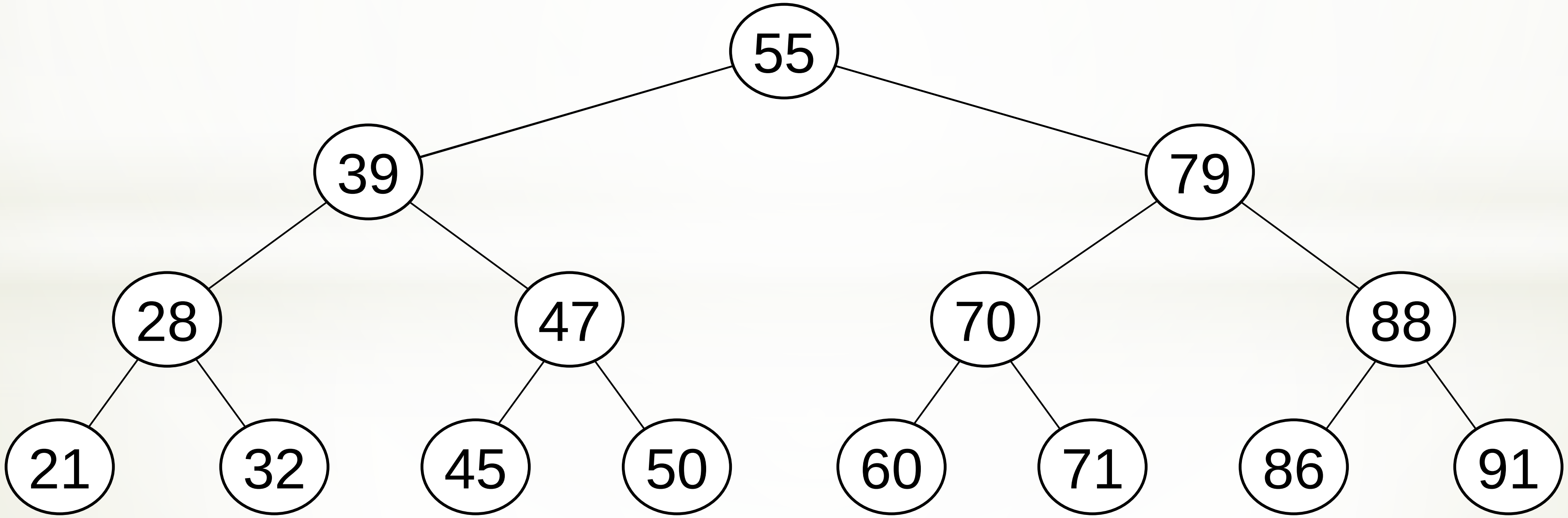


Trees as Data Structures

Trees are useful for representing hierarchical structure in real life applications:

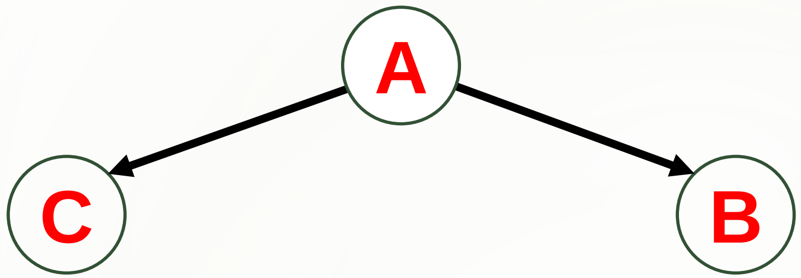
Example 4 The ordered structure of data for fast searching purposes

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
21	28	32	39	45	47	50	55	60	70	71	79	86	88	91



Tree Terminology

Each node in a directed graph has a set of **predecessors** and **successors**



successors (**A**) = {**C**, **B**}

predecessors (**B**) = {**A**}

parent (**C**) = **A**

A **Tree** is a graph in which:

- Each node, apart from the root node, has exactly one predecessor.
- There is exactly one node (the **root** node) which has no predecessors.

The one predecessor of a node is called the **parent** of the node.

A **binary** tree is a tree in which every node has at most two successors

A **leaf** node has no successors

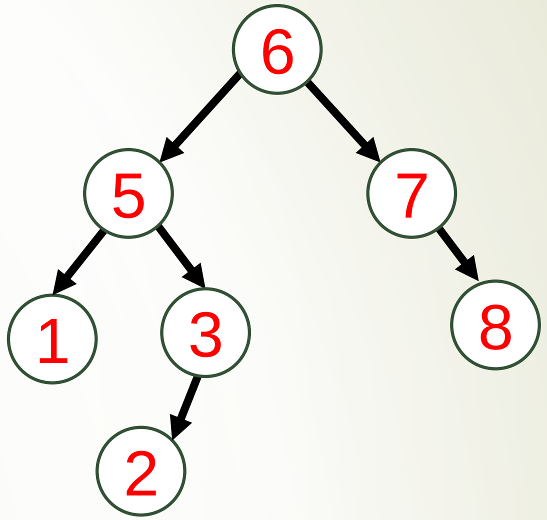
A node in a binary tree comprises: a data item, a left subtree, and a right subtree

A tree is therefore a **recursive** data structure

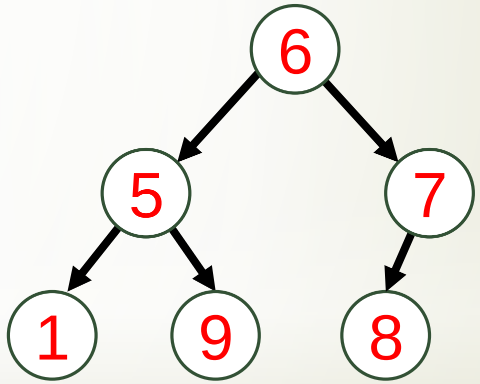
Tree Terminology

The **depth** (or height) of a tree is the number of levels

depth(T) = 1 + max(depth(T.left), depth(T.right))



In a **full** binary tree,
all non-leaf nodes have exactly 2 children

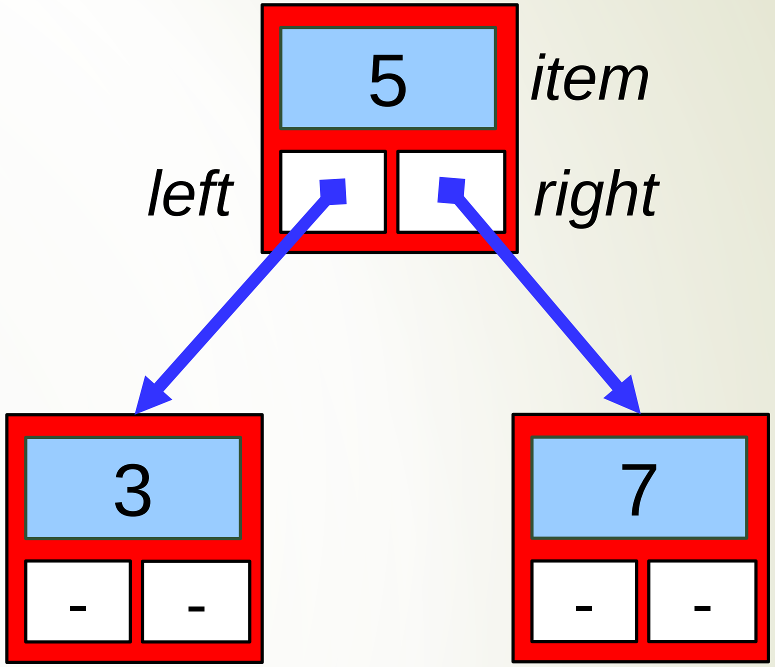


A **complete** binary tree of depth d is filled to the first d-1 levels, and any unfilled nodes are on the right

Implementing a Binary Tree

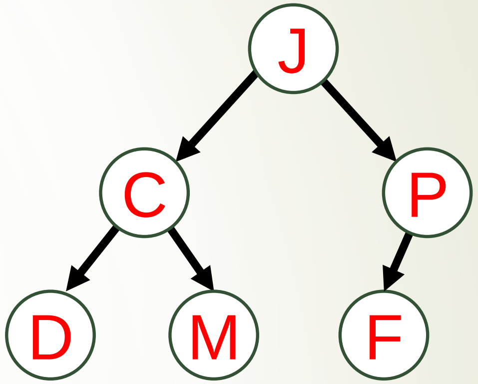
TreeNode.h

```
template <typename T>
class  TreeNode  {
public:
        TreeNode(T i, TreeNode *l, TreeNode* r);
        TreeNode(T i);  // for creating a leaf node
        ~TreeNode();
private:
        T item;
        TreeNode *left, *right;
}
```




```
TreeNode<int> myTree(6) ;    // Calls the 'leaf' constructor
TreeNode<int> myTree(5, new TreeNode<int>(3), new TreeNode<int>(7));
TreeNode<int>* myTree = new TreeNode<int>(6);     // Calls the 'leaf' constructor
TreeNode<int>* myTree = new TreeNode<int>(5, new TreeNode<int>(3), new TreeNode<int>(7));
```

Defining Recursive Properties of a Tree



- How would you **define** the **descendants** of a person? (let's say including the person)?

  For the given tree, **descendants** = { J, C, D, M, P, F }

- For the general case of a tree T:

  descendants(T) =

  if (T is empty): { }

  otherwise: { T→item } ∪ **descendants** (T→left) ∪ **descendants** (T→right)

- The use of a definition within itself is similar to **recursion**.

- It is perfectly safe and well-defined, because there is a base case ("T is empty") which is not recursive.

- ancestors (Node) = { parent (Node) } ∪ ancestors (parent)

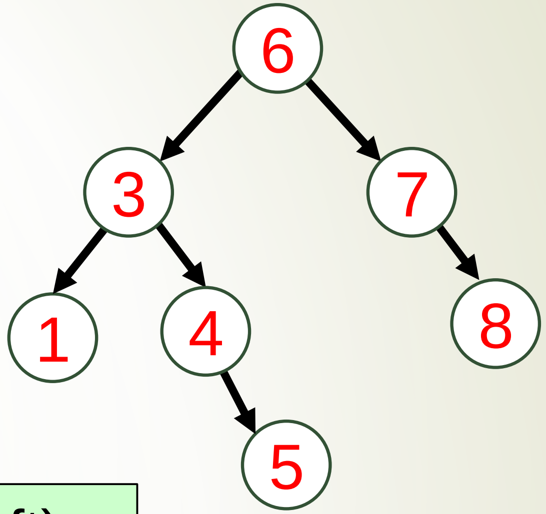
Recursive Data Structures

- When a data structure is inherently recursive (e.g. a tree), we should expect many processing functions to be recursive.
- Even a list can be processed using recursion rather than a for loop, since a list can be defined as an item (head) followed by a (possibly empty) list (tail)
- Hence, many tree processing algorithms are recursive:

```
void traverseTree (TreeNode<T>* t)
{
    if (t == NULL)  // Tree is empty
        // … The base/non-recursive case
    else  {
        process (t->item);
        traverseTree (t->left);
        traverseTree (t->right);
    }
}
```

Where we process the item gives different scanning patterns

Tree Traversals



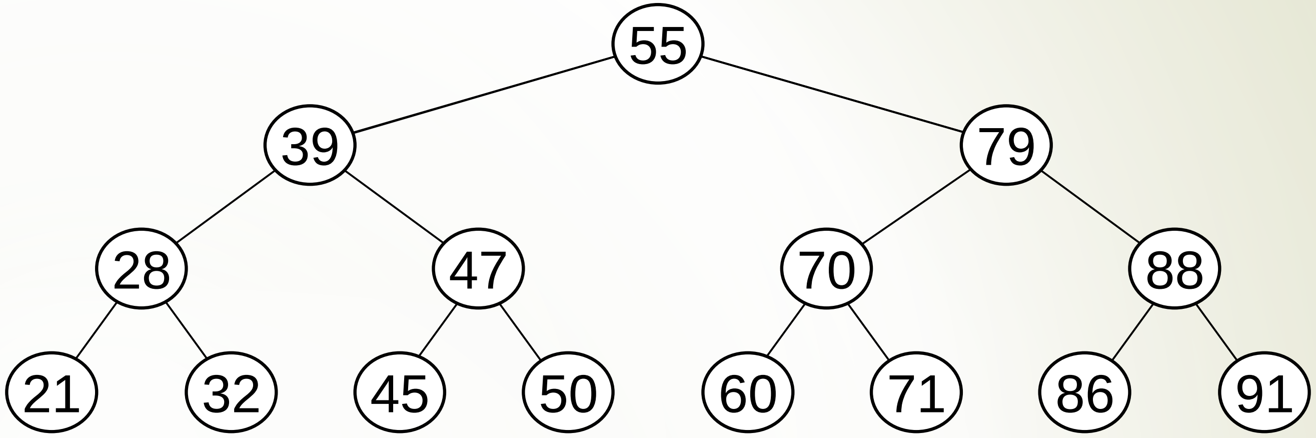
Pre-order	In-order	Post-order
process (t->item); traverseTree (t->left); traverseTree (t->right);	traverseTree (t->left); **process** (t->item); traverseTree (t->right);	traverseTree (t->left); traverseTree (t->right); **process** (t->item);

Pre-order: 6 3 1 4 5 7 8

In-order: 1 3 4 5 6 7 8

Post-order: 1 5 4 3 8 7 6

Building Sorted Trees



- Building a sorted tree one item at a time.

- Algorithm to insert an item X in a tree Tree:

  If (Tree is empty), X becomes the root of Tree (create a leaf node with X)

  Otherwise:

  if X < Tree's item, insert it in left(L); otherwise insert it in right(R)

- To get the above tree, we have to be careful about the order of adding the items.
- E.g. 55, 39, 47, 45, 28, 50, 79, 88, 21, 70, 86, 60, 32, 91, 71 is OK.

Inserting a Value in a Sorted Tree in C++

```
TreeNode<T>* TreeNode<T>::insert (TreeNode<T>* tree, T item)
{
    // Inserts item in tree, and returns the new tree
    if (tree == NULL)
        tree = new TreeNode<T>(item);
    else
        if (item < tree→item)
            tree→left = insert (tree→left, item);
        else
            tree→right = insert (tree→right, item);
    return tree;
}
```

For example:

```
TreeNode<int>* tree = new TreeNode<int>(5);
tree = tree->insert(tree, 7);
tree = tree->insert(tree, 3);
```