Trees

Examples of Tree structure
Definition of trees
Binary tree
Height of tree
Tree traversals
Finding max
Finding sum

Trees

You have seen that using linked lists you can represent an ordered collection of values without using arrays. Although linked lists require more memory space than arrays (as they have to store address at each node), they have definite advantages over arrays. Insertion and deletion of items can be carried out with out involving considerable movement of data.

The ordering relationship amongst a set of values is obtained through use of pointers. However, we need not restrict ourselves to only linear structures. In this chapter we shall extend the use of pointers to define a non-linear structure to model hierarchical relationships, such as a family tree.

In such a tree, we have links moving from an ancestor to a parent, and links moving from the parent to children. We have many other examples of tree-structured hierarchies.

Directory Hierarchies: In computers, files are stored in directories that form a tree. The top level directory represents the root. It has many subdirectories and files. The subdirectories would have further set of subdirectories.

Organization charts: In a company a number of vice presidents report to a president. Each VP would have a set of general managers, each GM having his/her own set of specific managers and so on.

Biological classifications: Starting from living being at the root, such a tree can branch off to mammals, birds, marine life etc.

Game Trees: All games which require only mental effort would always have number of possible options at any position of the game. For each position, there would be number of counter moves. The repetitive pattern results in what is known a game tree.

Tree as a data structure

- A **tree** is a data structure that is made of nodes and pointers, much like a linked list. The difference between them lies in how they are organized:

- The top node in the tree is called the **root** and all other nodes branch off from this one.

- Every node in the tree can have some number of children. Each child node can in turn be the **parent** node to its children and so on.

- Child nodes can have links only from a *single parent.*

- Any node higher up than the parent is called an **ancestor** node.

- Nodes having no children are called **leaves.**

- Any node which is neither a root, nor a leaf is called **an interior node**.

- The **height** of a tree is defined to be the length of the longest path from the root to a leaf in that tree (including the path to root)

- A common example of a tree structure is the binary tree.

Binary Trees

Definition: A **binary tree** is a tree in which each node can have maximum two children. Thus each node can have no child, one child or two children. The pointers help us to identify whether it is a left child or a right child.

Application of a Binary tree

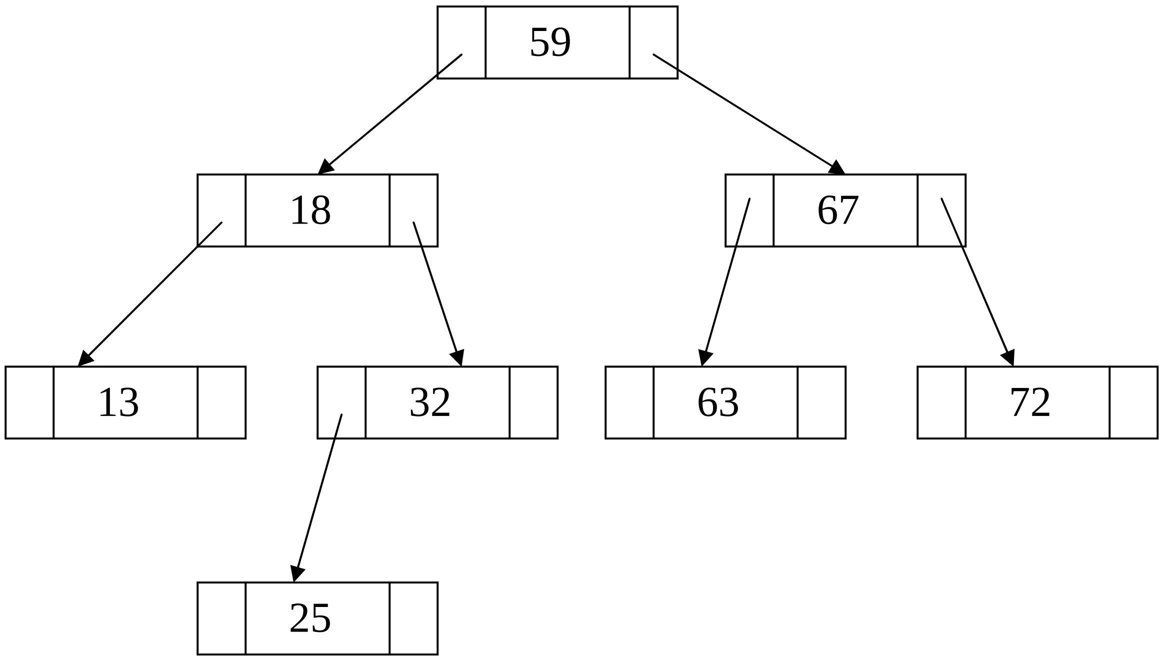
Before we define any formal algorithms, let us look at one possible application of a binary tree.
Consider a set of numbers: 25,63,13, 72,18,32,59,67.

Suppose we store these numbers in individual nodes of a singly linked list. To search for a particular item we have to go through the list, and maybe we have to go to the end of the list as well. Thus if there were n numbers, our search complexity would be O(n).

Is it because the numbers are not in any particular sequence? Now suppose we order these numbers: 13,18,25,32,59,63,67,72. and store these in another linked list.

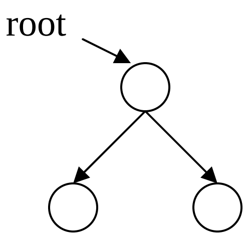
What would be the search complexity now? You may be surprised to discover that it is still O(n). You simply cannot apply binary search on a linked list with O(log n) complexity. You still have to go through each link to locate a particular number. So a linear linked structure is not helping us at all.

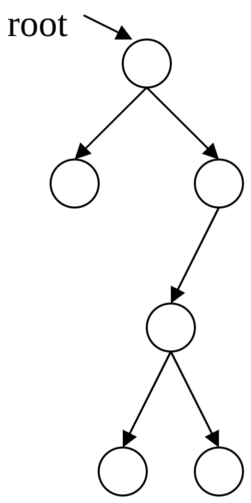
Let us see if we can improve the situation by storing the data using a binary tree structure. Consider the following binary tree where the numbers have been stored in a specific order. The value at any node is more than the values stored in the left-child nodes, and less than the values stored in the right-child nodes.



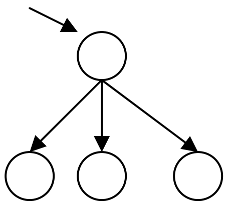
With this arrangement any search is taking at most 4 steps.
For larger set of numbers, if we can come up with a good tree arrangement than the search time can be reduced dramatically.

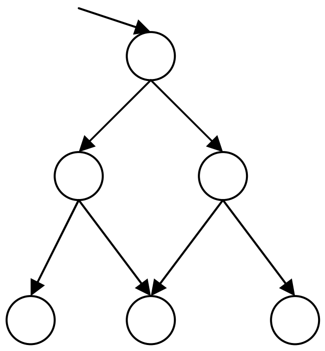
Examples of binary trees:





- The following are NOT binary trees:



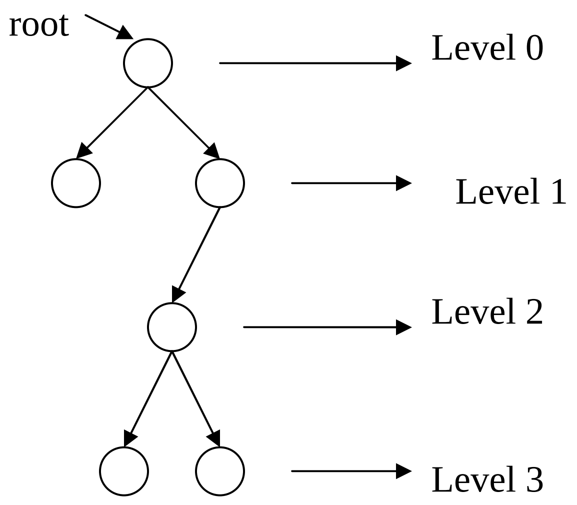


Definitions:

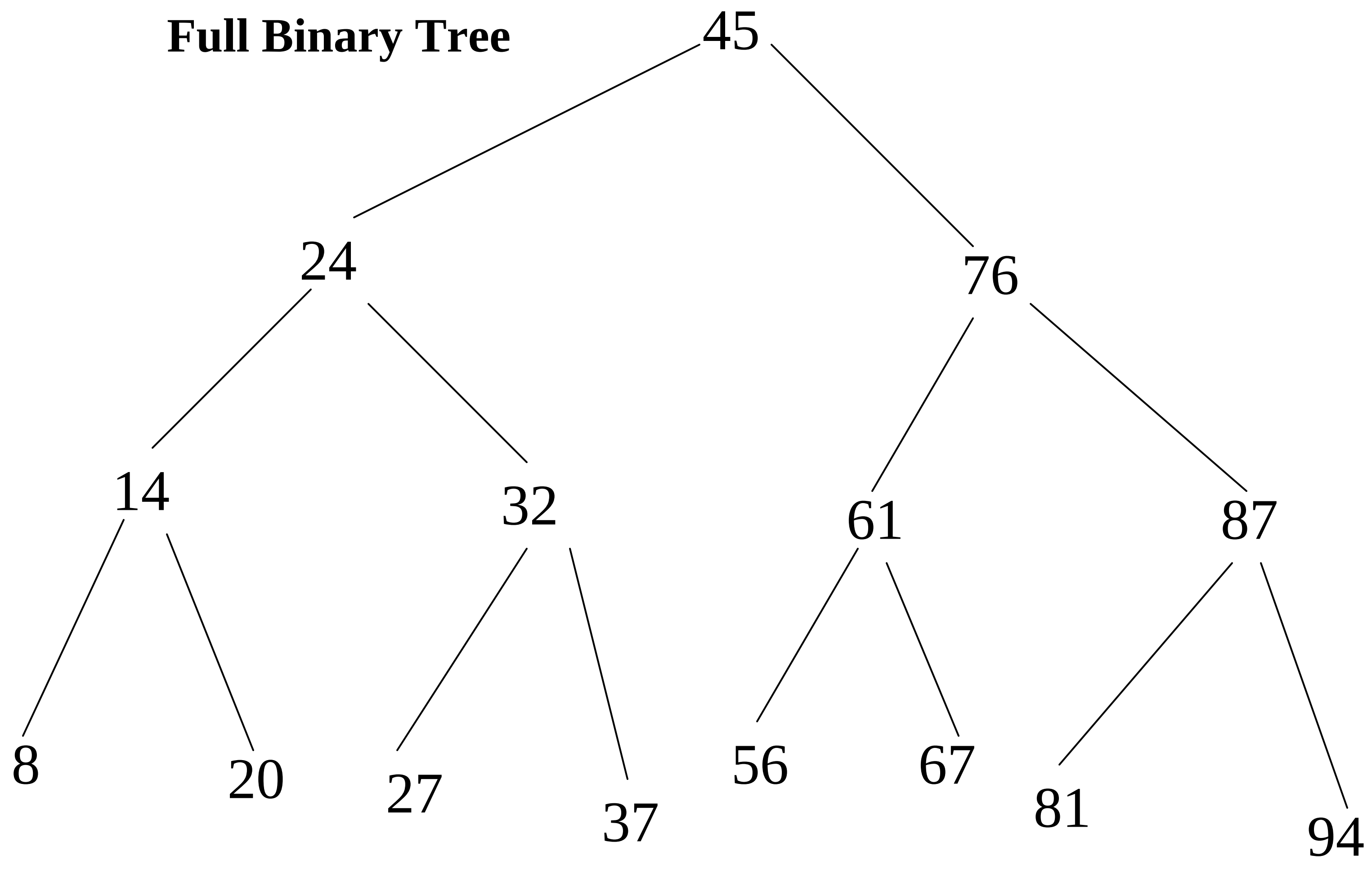
- tree, then n1 is the **parent** of n2 and n2 is the **left** or **right child** of n1.

- The **level** of a node in a binary tree:

    - The root of the tree has level 0
    - The level of any other node in the tree is one more than the level of its parent.



Full Binary Tree



How many nodes?

Level 0 **:** 1 node **(height 1)**
Level 1: 2 nodes (height 2)
Level 3 : 4 nodes (height 3)
Level 3: 8 nodes (height 4)

Total number of nodes

$n = 2^h - 1$ (maximum)

$h = \log (n+1)$

Implementation

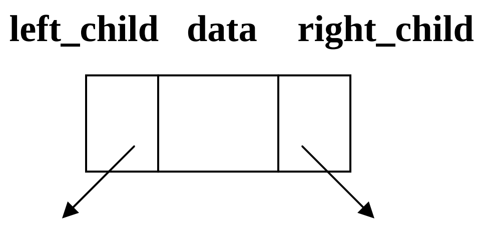
- A binary tree has a natural implementation in linked storage. A tree is referenced with a pointer to its root.

- **Recursive definition** of a binary tree:

  A binary tree is either
    - Empty, or
    - A node (called root) together with two binary trees (called left subtree and the right subtree of the root)

- Each node of a binary tree has both left and right subtrees which can be reached with pointers:

```
struct tree_node{
    int data;
    struct tree_node *left_child;
    struct tree_node *right_child;
};
```



Note the recursive definition of trees. A tree is a node with structure that contains more trees. We have actually a tree located at each node of a tree.

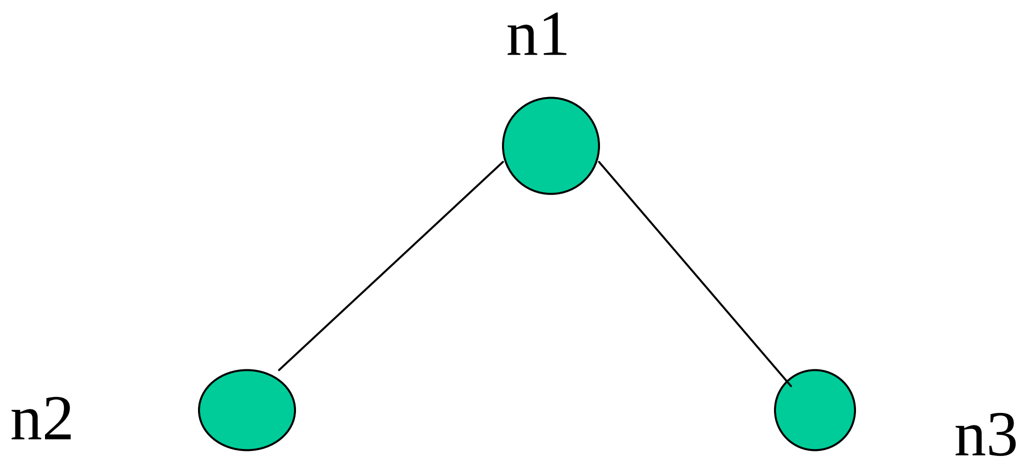
Traversal of Binary Trees

Linked lists are traversed sequentially from first node to the last node. However, there is no such natural linear order for the nodes of a tree. Different orderings are possible for traversing a binary tree. Every node in the tree is a root for the subtree that it points to. There are three common traversals for binary trees:

- **Preorder**
- **Inorder**
- **Postorder**

These names are chosen according to the sequence in which the root node and its children are visited.

Suppose there are only 3 nodes in the tree having the following arrangement:



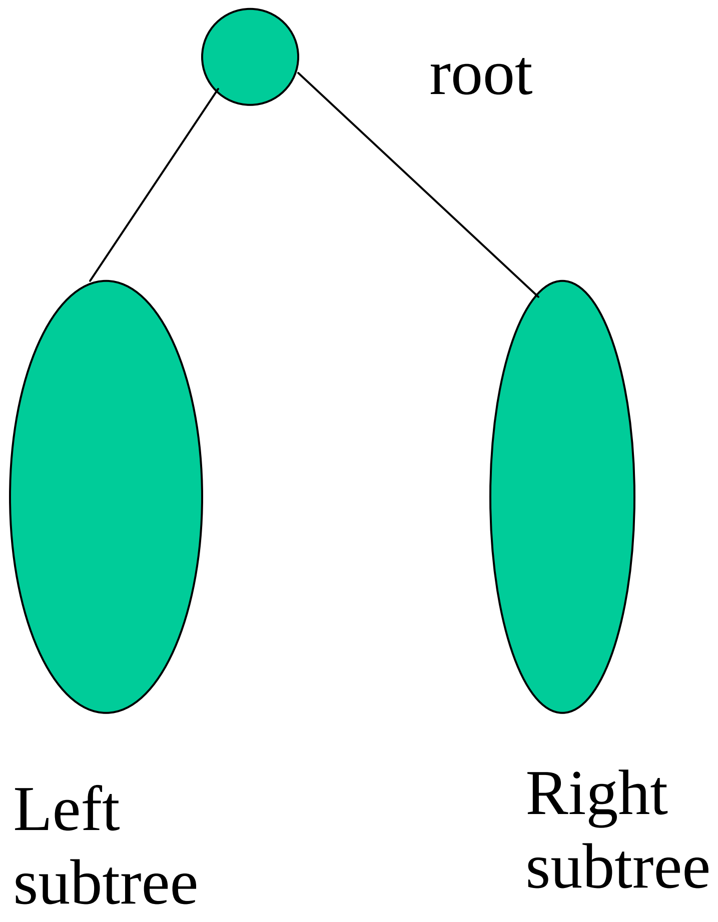
In – order : n2 n1 n3

Pre-order : n1 n2 n3

Post order : n2 n3 n1

- With **inorder** traversal the order is left-child, root node, right-child
- With **preorder** traversal the order is root node, left child , right child.
- With **postorder** traversal the order is left child, right child, root node.

A tree will typically have more than 3 nodes. Instead of nodes n2 and n3 there would be subtrees as shown below:



– With **inorder** traversal the order is left subtree, then the root and finally the right subtree. Thus the root is visited in-*between* visiting the left and right subtrees.

– With **preorder** traversal the root node is visited first, then the nodes in the left subtree are visited followed by the nodes in the right subtrees

– With **postorder** traversal the root is visited *after* both the subtrees have been visited.(left subtree followed by right subtree.

- As the structure of a binary tree is recursive, the traversal algorithms are inherently recursive.

Algorithm for Preorder traversal

In a preorder traversal, we first visit the root node.

If there is a left child we visit the left subtree (all the nodes) in pre-order fashion starting with that left child .

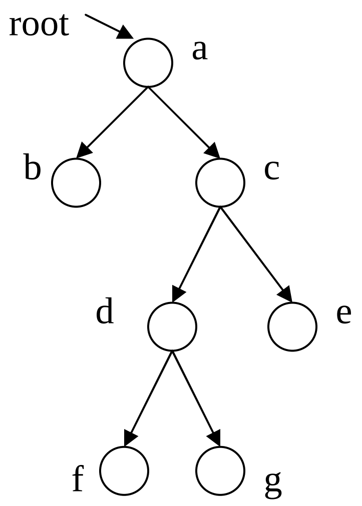
If there is a right child then we visit the right subtree in pre-order fashion starting with that right child.

The function may seem very simplistic, but the real power lies in the recursive formulation. In fact there is a double recursion. The real job is done by the system on the run-time stack. This simplifies coding while it puts a heavy burden on the system.

```
void preorder(struct tree_node * p)
{  if (p !=NULL) {
        printf(“%d\n”, p->data);
        preorder(p->left_child);
        preorder(p->right_child);
    }
}
```

Take a tree of say height 3 with maybe 6 nodes and try to run the above recursion to find out the actual order of printing the nodes.

Example:



<u>**Preorder** Traversal</u> :   a b c d f g e

a(root)

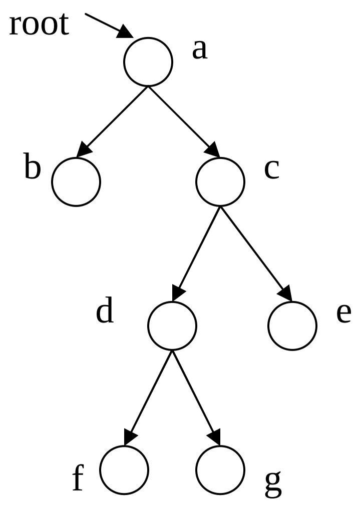
b(left)

c d f g e (right)

Algorithm for Inorder traversal

In the inorder traversal, we first visit its left subtree (all the nodes) , then we visit the root node and then its right subtree.

```
void inorder(struct tree_node *p)
{   if (p !=NULL) {
        inorder(p->left_child);
        printf(“%d\n”, p->data);
        inorder(p->right_child);
    }
}
```



<u>Inorder</u>:   b a f d g c e

b(left)

a(root)

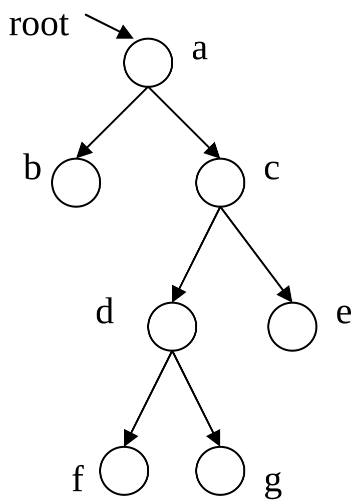
f d g c e(right)

Algorithm for Postorder traversal

In a postorder traversal, we first visit its left subtree (all the nodes) and then visit its right subtree (all the nodes) and then finally we visit the root node.

```
void postorder(struct tree_node *p)
{   if (p !=NULL) {
        postorder(p->left_child);
        postorder(p->right_child);
        printf(“%d\n”, p->data);
    }
}
```

Example:



<u>Postorder</u>: b f g d e c a

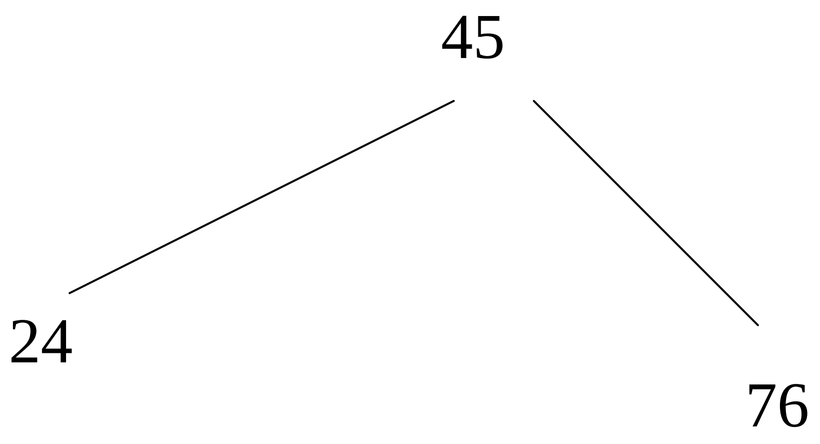
b(left)

f g d e c(right)

a(root)

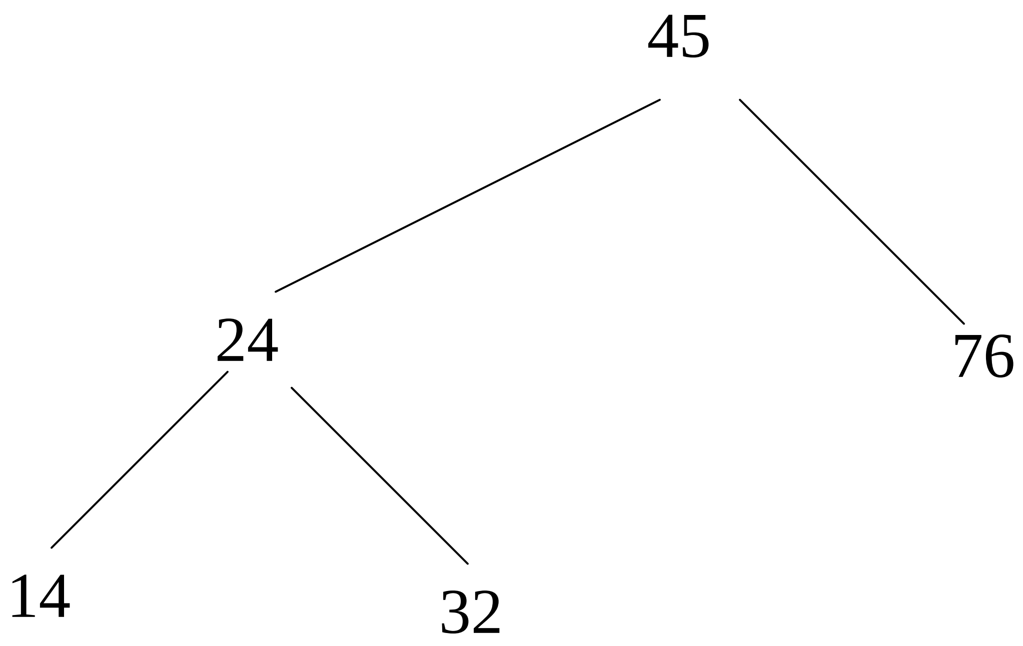
Finding Maximum value in a given tree p

```
int findMax (struct tree_node *p)
{
    int node_data, leftmax, rightmax, max;

    max = -1
//assume all values in the tree are positive
integers

    if (p !=   NULL)
    {  node_data = p -> data;
       leftmax = findMax(p -> left_child);
      rightmax = findMax(p->right_child);

       //find the largest of the tree values.

      if (leftmax > rightmax)
        max = leftmax;
          else
        max = rightmax;
          if (node_data > max)
        max = node_data;
    }
    return max;
```



node_value = 45
leftmax = 24 (max of 24, -1, -1)
rightmax = 76 (max of 76, -1, -1)

max = max of 45, 24, 76
        = 76



max of left subtree
leftmax = max (24,12,32)
        = 24
rightmax = 76
max of tree = max (45, leftmax, rightmax)
                    = 76

Finding sum of values of all the nodes of a tree

To find the sum, add to the value of the current node, the sum of values of all nodes of left subtree and the sum of values of all nodes in right subtree.

```
int sum(struct tree_node *p)
{
  if ( p!= NULL)
    return(p->data + sum(p->left_child)
                + sum(p->right_child));
  else
    return 0;
}
```