CONTENTS

CHAPTER – 26

BINARY TREES

INTRODUCTION

A ***binary tree*** is a tree data structure in which each node has at most two children. Typically the child nodes are called left and right. **Binary trees** are commonly used to implement binary search trees and binary heaps. The binary tree is a useful data structure for rapidly storing sorted data and rapidly retrieving stored data.

The root node of a tree is the node with no parents. There is at most one root node in a rooted tree. A leaf node has no children. The depth of a node n is the length of the path from the root to the node. The set of all nodes at a given depth is sometimes called a level of the tree. The root node is at depth zero.

The height of a tree is the depth of its farthest leaf. A tree with only a root node has a height of zero. Siblings are nodes that share the same parent node. A rooted **binary tree** is a rooted tree in which every node has at most two children. The root is starting or topmost node in the tree.

The typical graphical representation of a **binary tree** is essentially that of an upside down tree. It begins with a root node, which contains the original key value. The root node has two child nodes; each child node might have its own child nodes. Ideally, the tree would be structured so that it is a perfectly balanced tree, with each node having the same number of child nodes to its left and to its right. A perfectly balanced tree allows for the fastest average insertion of data or retrieval of data. The worst case scenario is a tree in which each node only has one child node.

The node storing 10, represented here merely as 10, is the root node, linking to the left and right child nodes, with the left node storing a lower value than the parent node, and the node on the right storing a greater value than the parent node. Notice that if one removed the root node and the right child nodes, that the node storing the value 6 would be the equivalent a new, smaller, binary tree. The structure of a binary tree makes the insertion and search functions simple to implement using recursion.

```
struct node
{
  int key_value;
  node *left;
  node *right;
};
```