So far all the data structures that we have encountered have been either of linear or tabular format. These data structures are generally not suitable to the representation of data which have a hierarchical organization (e.g., the structure of a company, the taxonomy of species in biology, etc.). E.g., if we need to represent the branches of the Federal Government, then we obtain a non-linear structure which resembles the one in Figure 1. Each department in the organization is further subdivided into sub-departments.

In order to represent this sort of data organization, we need to adopt a new type of data structure, called tree.

1 Introduction

Trees are structures that are frequently used in various problem solving activities. Examples of situations where trees can be useful are

- representation of arithmetic expressions;
- description of possible moves in a game playing program;
- to efficiently maintain indexed sets of information;

2 Basic Concepts

A tree is a structure composed by a collection of interconnected elements. Each element in the tree is called node. A link between two nodes is called edge. Each edge links two nodes. The edges are assumed to be directed, which means that they go from one node to the other. If we have an edge going from node $a$ to node $b$, then we say that the edge is an incoming edge for node $b$; similarly for an edge from node $a$ to node $b$ we say that the edge is outgoing for node $a$.

The nodes are organized into a hierchical structure. At the top of the tree lies a distinguished node called root of the tree. For example, in the tree structure depicted in Figure 1, the nodes are the different departments and the root is represented by the top node called Federal Government. The root is the only node which does not have incoming edges. Each other node in the tree has exactly one incoming edge. Each node can have an arbitrary number of outgoing edges. See Figure 2.
If $n$ is a node, and this is connected to the nodes $n_1, \ldots, n_k$ via outgoing edges, then $n_1, \ldots, n_k$ are all children of $n$. In Figure 2, nodes $N_1, N_2, N_3$ are children of node $N$. In turn, $N$ is called the parent of $N_1, N_2, N_3$. $N$ is also in this case the root of the tree (as $N$ does not have any incoming edge). Observe that, since each node (except for the root) has exactly one incoming edge, then this means that each node in the tree (with the exception of the root) has exactly one parent. The nodes in the tree which do not have any outgoing edges are called leaves of the tree. In Figure 2, node $N_2$ is a leaf as it does not have any edge leaving the node.

A node which is neither a root nor a leaf is called internal. Thus internal nodes are all those nodes that have a parent and at least one child.

![Figure 2: Nodes in a Tree](image)

Similarly, in Figure 1, the root is the node called Federal Government; the children are the various nodes called Defense, Education, etc. In the figure, the node called Army is a leaf, since it does not have any children. The node called Defense is an internal node as it has both a predecessor (a parent) and successors (its children).

Nodes in the tree can be arranged in levels. The root is the only node at level 0. Level 1 contains all the children of the root node. Level 2 contains all the children of the nodes at level 1, and so on. Figure 3 illustrates the concept in an example. The height of a tree is the number of different levels that are present in the tree.

Any node $n$ in a tree can be reached starting from the root and following the edges. The sequence of edges traversed to go from the root to a given node $n$ represent a path in the tree. In each tree there is exactly one path from the root to any given node. Figure 3 shows the path from the root to the node $k$.

According to this definition, it is easy to see that if a node $n$ belongs to the level number $i$ of a tree, then this means that the unique path from the root to node $n$ contains exactly $i$ edges. Also, according to this we can redefined the notion of height of a tree as follows: the height of a tree is the length of the longest path from the root present in the tree.

Children of a given node are ordered from left to right; for example in Figure 2, $N_1$ is the first child, $N_2$ is the second child, and $N_3$ is the third child of $N$.

Another important concept is that of subtree. Given a tree and a node $n$ in the tree, the set of all nodes which have $n$ as ancestor represents a tree structure; this is called the subtree rooted in $n$. Figure 4 shows some examples of subtrees. This is very important, as it allows to provide a recursive construction of trees: a tree is either composed by zero nodes or it is composed by a node (the root) which has a number of children, and these children in turn represent other (smaller) trees.

**Question:** Assume that each node in a tree can have at most 2 children. What is the maximum number of nodes in a tree with two levels? What is the maximum number of nodes in a tree with 3 levels? What is the
maximum number of nodes in a tree with $n$ levels?

3 Binary Trees

Given a tree, in the definition given before we did not place any restriction on the number of children each node can have. For an arbitrary tree, the degree of the tree is the maximum number of children that each node is allowed to have in the tree.

A commonly used type of tree is the Binary Tree. A Binary tree is characterized by the fact that each node can have at most two children (i.e., the degree of the tree is two). When dealing with binary trees, we identify the first child of a node as the left child and the second child as the right child. Alternatively, if we use the recursive decomposition of trees described earlier, we can view a binary tree as either an empty tree (i.e., a tree with zero nodes), or as composed by a root element and the remaining nodes are partitioned in two binary trees (which are called the left and right subtree of the original tree).

For example, if we consider the tree representing an arithmetic expression (Figure 5), this is commonly a binary tree. In the figure, the root is the node $+$, and the figure identifies the left and right subtree. These are, in turn,
binary trees as well.

![Binary Tree Diagram]

Figure 5: Binary Tree

In this section we will study a specification of an ADT describing binary trees.

### 3.1 Properties

Before getting into the details of the specification and implementation of an ADT for binary trees, let us study some of the properties of binary trees:

- **Property 1:** if we consider a binary tree containing \( n \) nodes, then this tree contains exactly \( n - 1 \) edges;
- **Property 2:** a binary of height \( h \) has a number of nodes which is greater or equal than \( h \) and less or equal than \( 2^h - 1 \).
- **Property 3:** if we have a binary tree containing \( n \) nodes, then the height of the tree is at most \( n \) and at least \( \lceil \log_2(n + 1) \rceil \).

If we consider a binary tree of height \( h \) which has been completely filled (i.e., each node not at level \( h \) has two children), then such tree contains exactly \( 2^h - 1 \) nodes; this tree is called the **Complete binary tree** of order \( h \).

### 3.2 Specification

**Set of values:** the set of all possible values is represented by the set of all possible binary trees over a given domain. For example, we can consider binary trees over **characters**, where each node in the tree contains exactly one character. Figure 6 illustrates some valid values belonging to this data type (observe that case (i) is the tree which contains zero nodes, i.e., the empty tree).

![Binary Trees Illustration]

Figure 6: Example of binary trees
For the sake of simplicity, we consider each tree characterized by its root node. Thus, to describe a tree we will just need to provide its root node (the outgoing edges will allow to access, from the root, the rest of the tree).

**Operations:** There are various set of operations that one can envision for a tree ADT. The list that follows is just indicative and could be extended and/or modified in different ways.

- **Constructors:** the only constructor needed is the one to create a new object of type tree. Initially the tree created is empty.

  - **CreateTree**
    - preconditions: none
    - postconditions: returns a new object of type binary tree containing, as value, the empty tree (i.e., the tree containing zero elements);

- **Destructors:** the only destructor needed is used to destroy a tree which is not needed any longer.

  - **DeleteTree**
    - preconditions: receives an existing tree as input
    - postconditions: none

- **Inspectors:** the following inspectors are included in the type specification:

  1. **GetLeft** which returns the subtree rooted in the left child of current tree (i.e., it returns a reference to the left child of the root of the current tree).
     - preconditions: receives a non-empty tree as input
     - postconditions: returns the subtree rooted in the left child of the root of the given tree.
  2. **GetRight** which returns the subtree rooted in the right child of the current tree (i.e., it returns a reference to the right child of the root of the current tree).
     - preconditions: receives a non-empty tree as input, with the assumption that the root has a right child;
     - postconditions: returns the subtree rooted in the right child of the root of the given tree.
  3. **EmptyTree** used to verify whether a given tree is empty or not
     - preconditions: receives a tree as input
     - postconditions: produces true (i.e., 1 in C) if the tree is empty, false (i.e., 0 in C) otherwise
  4. **EqualTrees** used to verify whether two trees have the same value (i.e., they contain the same elements and in the same order).
     - preconditions: receives two trees as input
     - postconditions: returns true if the values of the two trees contain the same elements and in the same order, false otherwise.
  5. **GetRoot** returns the information which is currently stored in the root of the given tree;
     - preconditions: receives a tree T
     - postconditions: if T is not empty, then it returns the value stored in the root of T, otherwise it gives an error.

- **Modifiers:** the following modifiers are included

  1. **InsertNode** inserts a new node in the tree;
     - preconditions: receives as input 3 arguments: the value x to store in the new node, the tree T in which the insertion has to take place, and a parameter which has 0 or 1 as possible values.
– postconditions: if \( x \) is of the correct type (i.e., it has the same type as all the other elements in \( T \)), then a new node with value \( x \) is created; if \( T \) is not empty and the parameter has value 0, then the new node becomes the left child of the root of \( T \); if \( T \) is not empty and the parameter has value 1, then the new node becomes the right child of the root of \( T \). If an insertion is attempted with parameter 0 and the root of \( T \) already has a left child, then an error will occur. Analogously if the parameter is 1 and the root of \( T \) already has a right child. If the initial tree is empty, then the parameter is ignored and the new node becomes the root of the tree.

2. **DeleteSubtree** remove a subtree rooted in a given node
   – preconditions: receives as input a nonempty tree \( T \) and a parameter (which can have 0, 1, or 2 as possible values).
   – postconditions: if the parameter is 2, then the whole tree \( T \) is removed. If the parameter is 0 then the subtree rooted in the left child of the root of \( T \) is removed. If the parameter is 1, then the subtree rooted in the right child of the root of \( T \) is removed. The operation returns the modified tree as result.

3. **AssignTree** assign the value of a tree to another tree object
   – preconditions: receives as input two tree objects;
   – postconditions: copies the value of the second tree in the first tree object (thus making them identical—i.e., they have the same value)

### 3.3 Implementation Using Linked Structures

The intuition behind this implementation is to organize the elements in the tree as a collection of structures linked to each others. Each node in the tree is represented by a separate structure, and each node contains pointer to its children (two pointers at most, since we are dealing only with binary trees). If one desires to have access to the parent of a node, then an additional pointer can be maintained in each node, which points to the parent of that node.

The whole tree, as well as any subtree, can be simply represented as a pointer to the root of the tree. Thus the tree is encoded as a pointer to a node. An empty tree is represented by a NULL pointer.

The interface of the ADT is stored in the file `tree.h`, and it contains the following:

```c
/*-----------------------------------*/
/* BINARY TREES ADT */
/*-----------------------------------*/
#include "genlib.h"

typedef int base_type;

typedef struct tree_type *Tree;

/*-----------------------------------*/
/* Constructors */
/* create an empty tree */
/*-----------------------------------*/

Tree CreateTree();

Tree CreateNodeTree(base_type elem);

/*-----------------------------------*/
/* Destructors */
/*-----------------------------------*/
```
/* destroy an existing tree*/
/*---------------------------------*/

void DestroyTree(Tree t);

/*---------------------------------*/
/* Inspectors */
/*---------------------------------*/

/* Is the current tree empty? */
bool EmptyTree(Tree t);

/* Who is the left child? */
Tree GetLeft(Tree t);

/* Who is the right child? */
Tree GetRight(Tree t);

/* What is inside the root? */
base_type GetRoot(Tree t);

/* Are two trees identical? */
bool EqualTrees(Tree t1, Tree t2);

/*---------------------------------*/
/* Modifiers */
/*---------------------------------*/

/* Insert new node */
/* params: current tree */
/* position for insertion */
/* content of the node */
/*---------------------------------*/
Tree InsertNode(Tree t, int position, base_type value);

/*---------------------------------*/
/* Delete Subtree */
/* params: current tree */
/* position of subtree */
/*---------------------------------*/
Tree DeleteSubtree(Tree t, int position);

The implementation of the ADT is stored in the file tree.c.
To make some of the operations to manage the tree simpler, we have provided an additional constructor in the implementation, called `CreateNodeTree` which creates a new tree which already contains one node (while instead the `CreateTree` operation creates a tree with zero nodes inside).

```c
/*-----------------------------------------------*/
/* Binary Tree ADT: Implementation */
/* using linked structures */
/*-----------------------------------------------*/
#include "tree.h"

/*------------------------------*/
/* Type Definitions */
/*------------------------------*/

/* Node of the tree */
typedef struct tree_type
{
    base_type value; /* content of the node */
    struct tree_type *left; /* left child */
    struct tree_type *right; /* right child */
} Node;

/*------------------------------*/
/* OPERATIONS */
/*------------------------------*/

/*-----------------------*/
/* CreateTree: */
/* creates empty tree */
/*-----------------------*/

Tree CreateTree()
{
    return NULL;
}

/*-----------------------*/
/* CreateNodeTree: */
/* takes a value and */
/* creates a tree with */
/* one node */
/*-----------------------*/

Tree CreateNodeTree(base_type elem)
{
    Tree temp;
    temp = GetBlock (sizeof(Node));
    temp->value = elem;
```
temp->left = temp->right = NULL;
return temp;
}

/*-----------------------*/
/* DestroyTree: */
/* takes a tree and */
/* destroys it */
/*-----------------------*/
void DestroyTree(Tree t)
{
    if (t == NULL)
        return;
    else
    {
        DestroyTree(t->left);
        DestroyTree(t->right);
        free(t);
    }
}

/*-----------------------*/
/* EmptyTree: */
/* takes a tree and */
/* verifies if it is */
/* empty */
/*-----------------------*/
bool EmptyTree (Tree t)
{
    return (t == NULL);
}

/*-----------------------*/
/* GetLeft: */
/* takes a tree and */
/* returns the left */
/* child */
/*-----------------------*/
Tree GetLeft(Tree t)
{
    if (t == NULL)
        Error("Cannot take left child of an empty tree\n");
    else
    {
        return t->left;
    }
}
/*-----------------------*/
/* GetRight: */
/* takes a tree and */
/* returns the right */
/* child */
/*-----------------------*/

Tree GetRight(Tree t)
{
    if (t == NULL)
        Error("Cannot take right child of an empty tree\n");
    else
        return t->right;
}

/*-----------------------*/
/* GetRoot: */
/* takes a tree and */
/* returns the content */
/* of the root */
/*-----------------------*/

base_type GetRoot(Tree t)
{
    if (t == NULL)
        Error("Cannot get root of an empty tree\n");
    else
        return t->value;
}

/*-----------------------*/
/* EqualTrees: */
/* takes two trees and */
/* checks if they are */
/* equal */
/*-----------------------*/

bool EqualTrees(Tree t1, Tree t2)
{
    if (t1 == NULL && t2 != NULL)
        return FALSE;
    if (t1 != NULL && t2 == NULL)
        return FALSE;
    if (t1 == NULL && t2 == NULL)
        return TRUE;
    if (t1->value == t2->value)
return (EqualTrees(t1->left,t2->left) &&
          EqualTrees(t1->right,t2->right));
else
  return FALSE;
}

/*-----------------------*/
/* InsertNode: */
/* takes a tree, a value*/
/* and a position (0,1)*/
/* and inserts new node */
/* in the indicated pos.*/
/*-----------------------*/

Tree InsertNode(Tree t, int position, base_type elem) {
    Tree new;

    /* create new node */
    new = GetBlock (sizeof(node));

    new->value = elem;
    new->left = new->right = NULL;

    /* if t is empty, then new node is the root */
    if (t == NULL)
        return new;

    /* test if I can insert a new child */

    if (position == 0 && t->left != NULL)
        Error("The left tree already exists\n");
    if (position == 1 && t->right != NULL)
        Error("The right tree already exists\n");

    if (position == 0) /* insert left */
        t->left = new;
    else
        if (position == 1) /* insert right */
            t->right = new;
    else
        Error("Incorrect position\n");
    return t;
}

/*-----------------------*/
/* DeleteSubtree: */
/*  takes a tree,   */
/*  and a position (0,1)*/
/*  and removes the */
/*  indicated subtree */
/*/-----------------------*/

Tree DeleteSubtree(Tree t, int position)
{
    if (t == NULL)
        Error("Cannot delete from empty tree\n");

    if (position == 0) /* destroy left */
    {
        DestroyTree(t->left);
        t->left = NULL;
    }

    else if (position == 1) /* destroy right */
    {
        DestroyTree(t->right);
        t->right = NULL;
    }

    else
        Error("Incorrect position \n");

    return t;
}
3.4 Implementation using Arrays

Representing linear structures (lists, stacks, etc.) using arrays was a quite natural operation, considering that the array itself is an inherently linear organization (a sequence of variables).

Trees are not linear. Thus representing them using arrays requires a proper mapping from the tree structure into a linear organization. In general representing trees using arrays is feasible (without excessive complications) only if we know a priori what is the maximum number of edges coming out of each node. Binary trees fall in this class, as we know that each node is going to have at most two successors.

Consider the tree in Figure 7(i). The nodes in the tree has been numbered consecutively, starting from 1 (root), 2 and 3 have been assigned to the children of the root, 4, . . . , 7 have been used for the grandchildren of the root and so on. The intuition, as shown in Figure 7(i), is that since

- the first $k$ levels of the tree can contain at most $1 + 2 + 4 + \ldots + 2^{k-1}$ nodes, i.e., $2^k - 1$ nodes;
- the level $k$ of the tree will contain at most $2^k$ nodes;

then we can conclude that by assigning a consecutive number to each node, moving down level by level, the nodes at level $k$ will use the numbers from $2^k$ to $2^{k+1} - 1$.

This is suggesting an easy idea for representing trees as arrays; we consider the above numbering scheme, and if a node is assigned node $j$, then we just simply store that node in the location $a[j]$ of the array. This is show in Figure 7(ii).

![Figure 7: Trees to Arrays](image)

Thus, if we are positioned in a node which is lying in position $i$ of the array, then we can easily identify:

- the left child will be in position $2 \times i$;
- the right child will be in position $2 \times i + 1$;
- its parent (if the node is not the root) is in position $i/2$;
- the root is in position 1

We assume that if a node does not exist, then the corresponding entry in the array is filled with a default value (e.g., $-1$).
The major problem is that the array required to store even a relatively small tree can be very large. If we consider the tree containing \( k \) nodes \( n_1, n_2, \ldots, n_k \) organized as follows: \( n_1 \) is the root, \( n_2 \) is the right child of \( n_1, n_3 \) is the right child of \( n_2, \ldots, n_k \) is the right child of \( n_{k-1} \), then we can easily see that we will need an array of size \( 2^k - 1 \) (e.g., for \( k = 11 \) we need an array with \( 2^{10} = 1024 \) positions—thus we have 1024 positions used to store a tree which contains only 11 nodes...).

The header file \texttt{tree.h} will contain:

```c
/*-------------------------------------------------*/
/* BINARY TREES ADT */
/*-------------------------------------------------*/

#include "genlib.h"

typedef int base_type;
typedef struct tree_type *Tree;

/*-------------------------------------------------*/
/* Constructors */
/*-------------------------------------------------*/

Tree CreateTree();
Tree CreateNodeTree(base_type elem);

/*-------------------------------------------------*/
/* Destructors */
/*-------------------------------------------------*/

void DestroyTree(Tree t);

/*-------------------------------------------------*/
/* Inspectors */
/*-------------------------------------------------*/

/* Is the current tree empty? */
bool EmptyTree(Tree t);
/* Who is the left child? */
Tree GetLeft(Tree t);
/* Who is the right child? */
Tree GetRight(Tree t);
/* What is inside the root? */
base_type GetRoot(Tree t);
```
Are two trees identical?

bool EqualTrees(Tree t1, Tree t2);

Modifiers

Insert new node
params: current tree
position for insertion
content of the node

Tree InsertNode(Tree t, int position, base_type value);

Delete Subtree
params: current tree
position of subtree

Tree DeleteSubtree(Tree t, int position);

The implementation of the ADT is stored in the file tree.c and contains the following:

```
#include "tree.h"

#define FILLER -1
#define MAXTREE 1024

typedef struct tree_type
{
  base_type space[MAXTREE];
  int root;
} wholetree;
```
/*------------------------------------*/
/* OPERATIONS */
/*------------------------------------*/
/*-----------------------*/
/* CreateTree: */
/* creates a tree with */
/* nothing inside */
/*-----------------------*/

Tree CreateTree()
{
    Tree new;
    int i;

    new = GetBlock(sizeof(wholetree));
    new->root = 1;

    for (i = 1; i < MAXTREE; i++)
        new->space[i] = FILLER;

    return new;
}

/*-----------------------*/
/* CreateTree: */
/* takes a value and */
/* creates a tree with */
/* one node containing */
/* such value */
/*-----------------------*/

Tree CreateNodeTree(base_type elem)
{
    Tree new;
    int i;

    new = GetBlock(sizeof(wholetree));
    new->space[1] = elem;
    new->root = 1;

    for (i = 2; i < MAXTREE; i++)
        new->space[i] = FILLER;

    return new;
}
/* destroys it */
/*/-----------------------*/

void DestroyTree(Tree t)
{
    free(t);
}

/*/-----------------------*/
/*/ EmptyTree: */
/*/ takes a tree and */
/*/ verifies if it is */
/*/ empty */
/*/-----------------------*/

bool EmptyTree (Tree t)
{
    return (t->space[t->root] == FILLER);
}

/*/-----------------------*/
/*/ GetLeft: */
/*/ takes a tree and */
/*/ returns the left */
/*/ child */
/*/-----------------------*/

Tree GetLeft(Tree t)
{
    if (EmptyTree(t))
        Error("Cannot get left of empty tree\n");
    else
    {
        t->root = 2*t->root;
        return t;
    }
}

/*/-----------------------*/
/*/ GetRight: */
/*/ takes a tree and */
/*/ returns the right */
/*/ child */
/*/-----------------------*/

Tree GetRight(Tree t)
{
    if (EmptyTree(t))
        Error("Cannot get right of empty tree\n");
    else
    {
        t->root = 2*t->root+1;
    }
}
return t;

/*-----------------------*/
/* GetRoot: */
/* takes a tree and */
/* returns the content */
/* of the root */
/*-----------------------*/

base_type GetRoot(Tree t)
{
    if (EmptyTree(t))
        Error("Cannot get root of an empty tree\n");
    else
        return t->space[t->root];
}

/*-----------------------*/
/* EqualTrees: */
/* takes two trees and */
/* checks if they are */
/* equal */
/*-----------------------*/

bool EqualTrees(Tree t1, Tree t2)
{
    return aux_equal_trees(t1,t2,1);
}

bool aux_equal_trees(Tree t1, Tree t2, int index)
{
    if (EmptyTree(t1) && !EmptyTree(t2))
        return FALSE;
    if (!EmptyTree(t1) && EmptyTree(t2))
        return FALSE;
    if (EmptyTree(t1) && EmptyTree(t2))
        return TRUE;

    if (t1->space[index] == t2->space[index])
        return (aux_equal_trees(t1,t2,2*index) &&
                aux_equal_trees(t1,t2,2*index+1));
    else
        return FALSE;
}

/*-----------------------*/
/* InsertNode: */
/* takes a tree, a value*/
Tree InsertNode(Tree t, int position, base_type elem)
{
    if (EmptyTree(t))
    {
        t->space[t->root] = elem;
        return t;
    }

    if (position == 0) /* insert left */
    {
        if (2*t->root < MAXTREE)
            t->space[2*t->root] = elem;
        else
            Error("Out of space\n");
    }
    else if (position == 1) /* insert right */
    {
        if (2*t->root+1 < MAXTREE)
            t->space[2*t->root+1] = elem;
        else
            Error("Out of space\n");
    }
    else
        Error("Incorrect position\n");

    return t;
}

Tree DeleteSubtree(Tree t, int position)
{
    if (EmptyTree(t))
        Error("Cannot delete from empty tree\n");

    if (position == 0) /* destroy left */
    {
        t->space[2*t->root] = FILLER;
    }
    else if (position == 1) /* destroy right */
    {

3.5 An Application

Consider the problem of managing a distribution network. We have a source which produces a certain resource (e.g., petroleum). A network distributes the resource from the source to a number of destinations (points of consumption). It is common that the structure of the distribution network is a tree structure, where the root is the source, the leaves are the consumption points, and the intermediate nodes are possible substations.

Along the path, the resource may experience loss or degradations of one of its characteristics. E.g., if the network distributes petroleum, then it is possible to have pressure drop along the lines. Between the source and the consumption points we can tolerate only a certain degree of degradation (called tolerance). To guarantee that degradation does not exceed the tolerance, we can place signal boosters in some of the substations of the network. The booster will restore the original value of the pressure, eliminating the degradation accumulated to that point.

Consider for example the network in Figure 8.

![Distribution Network](image)

The root (p) is the source. The leaves (w,x,y,z) are the consumption points. All other nodes are substations. A substation may or may not contain a booster. The numbers on the edges are used to measure the degradation along that segment of the network. Thus, going from q to t we will experience a degradation of 2 points. Degradation is an additive property. Thus, going from p to u we will experience a total degradation of 3 + 1 = 4.

The problem is to determine the best placement of boosters at the substations. We want to place the minimum number of boosters that will guarantee that the resource will get to each consumption point with a degradation which is not superior than the tolerance.

In the example, if the tolerance is 4, if we do not put any booster in the path from the source to w then we will violate our safety property, because on such path we have a total degradation of 1 + 2 + 3 = 6 which is greater than the tolerance (4). If we place a booster in the substation s, then that path will become safe, since up to s we have a total degradation of 3 (below tolerance), and at s all the degradation is removed by the booster; this allows the resource to get to the consumption point w with a degradation of 3 (due to the degradation between s and w).

The solution strategy is as follows:
let us denote with \( \text{degradeFromParent}(i) \) the degradation which occurs between node \( i \) and its parent. In the example we have that \( \text{degradeFromParent}(w)=3 \), \( \text{degradeFromParent}(s)=2 \), and \( \text{degradeFromParent}(q)=1 \).

- let us denote with \( \text{degradeToLeaf}(i) \) the maximum degradation experience between node \( i \) and any of its leaf successors. In the example, we have that \( \text{degradeToLeaf}(q)=5 \) and \( \text{degradeToLeaf}(r)=4 \).

In general we can compute the \( \text{degradeToLeaf} \) as follows:

\[
\text{degradeToLeaf}(i) = \max_{j \text{ child of } i} \left\{ \text{degradeToLeaf}(j) + \text{degradeFromParent}(j) \right\}
\]

The value of this function for each node in the tree can be easily computed by starting from the leaves (whose value is zero) and then progressively moving towards the root.

If during the computation we determine that

\[
\text{degradeToLeaf}(j) + \text{degradeFromParent}(j) > \text{tolerance}
\]

then we need to place a booster at node \( j \).

In the example, when we compute \( \text{degradeToLeaf}(s) + \text{degradeFromParent}(s) \) we obtain 5 which violates a tolerance of 4. If we place a booster at node \( s \), then the degradation above \( s \) disappears and we will get a \( \text{degradeToLeaf}(q) = 3 \).

Thus the pseudocode for computing the \( \text{degradeToLeaf} \) values and placing boosters is as follows:

\[
\text{degradeToLeaf}(i) = 0;
\]

\[
\text{for each } j \text{ child of } i \text{ do}
\]

\[
\text{if } (\text{degradeToLeaf}(j) + \text{degradeFromParent}(j) > \text{tolerance}) \text{ then}
\]

\[
\text{place a booster at } j
\]

\[
\text{degradeToLeaf}(i) = \max\{\text{degradeToLeaf}(i), \text{degradeFromParent}(j)\};
\]

\[
\text{else}
\]

\[
\text{degradeToLeaf}(i) = \max\{\text{degradeToLeaf}(i), \text{degradeFromParent}(j) + \text{degradeToLeaf}(j)\};
\]

It is possible to prove that this produces the optimal placement of boosters in the network.

### 3.6 Traversing Trees

A typical problem is that of traversing a binary tree, i.e., scanning all the nodes of the tree in a predefined order.

We can start by distinguishing two forms of tree traversal:

1. **Depth-first traversal**: the processing proceeds along a path from the root to one child to the most distant descendant of that first child before processing a second child. I.e., all successors of a child are processed before moving to another child.

2. **Breadth-first traversal**: the processing proceeds horizontally from the root to all of its children, then to its children's children and so forth until all nodes have been processed.

#### 3.6.1 Depth-First Traversal

There are three typical orders for (depth-first) traversing a tree:

- **PreOrder**;
- **PostOrder**;
- **InOrder**

Consider the example of the Figure 9. The tree contains the description of an arithmetic expression. Let us consider the binary tree to have base type \texttt{char} (i.e., the value in each node of the tree is a single character).
PreOrder: Visiting a tree in preorder involves exploring the tree as follows:

1. first of all visit the root of the tree (and read its content);
2. visit in PreOrder the left subtree of the root;
3. visit in PreOrder the right subtree of the root;

In the example of Figure 9, a PreOrder visit will thus produce the following sequence of values:

```
/ - ∧ b 2 * * 4 a c * 2 a
```

This visit can be implemented as follows, as part of the implementation using linked lists:

```c
void PreOrderTraverse (Tree t)
{
    if (!EmptyTree(t) && (t != NULL))
    {
        printf("%c", t->value);
        PreOrderTraverse(t->left);
        PreOrderTraverse(t->right);
    }
}
```

PostOrder: Visiting a tree in postorder involves exploring the tree as follows:

1. first visit the whole left subtree in PostOrder;
2. then visit the whole right subtree in PostOrder;
3. the visit the root of the tree;

In the example of Figure 9, a PostOrder visit will thus produce the following sequence of values:

```
b 2 ∧ 4 a * c * − 2 a * /
```

This visit can be implemented as follows, as part of the implementation using linked lists:
void PostOrderTraverse (Tree t)
{
    if (!EmptyTree(t) && (t!= NULL))
    {
        PostOrderTraverse(t->left);
        PostOrderTraverse(t->right);
        printf('%c',t->value);
    }
}

InOrder: Visiting a tree in inorder involves exploring the tree as follows:

1. first visit the whole left subtree in InOrder;
2. the visit the root of the tree;
3. then visit the whole right subtree in InOrder;

In the example of Figure 9, a InOrder visit will thus produce the following sequence of values:

\[ b \wedge 2 - 4 \times a \times c / 2 \times a \]

This visit can be implemented as follows, as part of the implementation using linked lists:

void InOrderTraverse (Tree t)
{
    if (!EmptyTree(t) && (t!= NULL))
    {
        InOrderTraverse(t->left);
        printf('%c',t->value);
        InOrderTraverse(t->right);
    }
}

3.6.2 Breadth-First Traversal

In the breadth-first traversal of a binary tree, we process all of the children of a node before proceeding with the next level. In other words, given a root at level \( n \), we process all nodes at level \( n \) before proceeding with the nodes at level \( n + 1 \).

While depth-first traversal are made according to the recursive structure of a tree (i.e., a tree is a root with two other trees attached below), breadth-first is instead ignoring this structuring and cutting the tree horizontally by levels. This implies that recursive solutions are not going to help in writing a breadth-first procedure. The order in which nodes are considered matches well with the FIFO order provided by a queue (in which nodes at level \( n \) are inserted in the queue before any node of level \( n + 1 \); as consequence all nodes of level \( n \) are going to be encountered while dequeueing the queue before encountering any node of level \( n + 1 \)). The breadth-first traversal of the tree in figure 9 will produce

\[- \wedge \times 2 \times a \times b \times 2 \times c \times 4 \times a\]

The following is a procedure for performing breadth-first traversal (it assumes the availability of a queue ADT—the base type for the queue is Tree):

void BreadthFirst (Tree t)
{
    Queue q;
    q = CreateQueue();
    if (EmptyTree(t))
        return;
    else
Enqueue(q,t);
while (! EmptyQueue(q))
{
    t = FrontQueue(q);
    Dequeue(q);
    printf("%d",GetRoot(t));
    if (! EmptyTree(GetLeft(t)))
        Enqueue(q,GetLeft(t));
    if (! EmptyTree(GetRight(t)))
        Enqueue(q,GetRight(t));
}

4 Binary Search Trees

A binary search tree is a data structure which is used to store information identified by unique keys. Each piece of information is uniquely associated to a key. You can think about keys as non-negative integer numbers.

The problem is to maintain the set of data in such a way to allow fast retrieval of information; given a key we would like to quickly discover whether such key has been used, and in this case we want to extract the corresponding piece of information.

Binary search trees are commonly used to solve this problem. The keys are stored in the nodes of a binary tree (we can assume the existence of an external table which maps each key to the corresponding data). Keys are inserted in the tree according to the following rule: to insert the key $k$ in the tree (see also Figure 10(i)):

Tree insertBST (Tree t, int key)
{
    Tree parent,root;

    if (EmptyTree(t))
        return CreateTree(key);

    root = t;
    while (! EmptyTree(t))
    {
        parent = t;
        if (key == GetRoot(t))
            return root;
        if (key < GetRoot(t))
            t = GetLeft(t);
        if (key > GetRoot(t))
            t = GetRight(t);
    }
    if (key < GetRoot(parent))
        insertNode(parent,0,key);
    else
        insertNode(parent,1,key);
    return root;
}

Thus, to summarize, for each node $N$ (containing the key $k$) in a binary search tree:

- all the keys in the left subtree of $N$ have value smaller than $k$;
- all the keys in the right subtree of $N$ have value bigger than $k$. 
Figure 10(ii) illustrates an example of search tree.

In order to search a key in the tree, we need to move left or right depending on the result of the comparison between the key and the root of the current subtree.

```c
void Search(Tree t, int Key) {
    if (EmptyTree(t))
        printf("Key not present");
    else
        if (GetRoot(t) == Key)
            printf("FOUND!!!");
        else
            if (RootInfo(t) > Key)
                Search(GetLeft(t), Key);
            else
                Search(GetRight(t), Key);
}
```

The more complex operation on binary search trees is the deletion of a key. Since we do not place any restrictions on the insertion of elements (any key can be inserted in the tree), then we should also not place any restriction on which key can be removed. This may potentially lead to complications, since we may be trying to remove a node which has successors, thus leaving a “hole” in the tree. E.g., if we try to remove node 15 from the tree in Figure 7, we need to make sure that we are not loosing access to node 17.

There are four possible cases which may occur:

1. the node to be removed does not have any child; in this case the node can be safely removed without endangering access to other nodes in the tree; we should only make sure that the connection between the node and its parent is removed (e.g., set the parent->left or parent->right value to NULL);
2. if the node has a right subtree but no left subtree, then we can simply attach the right subtree to the node’s parent (see Figure 11(i)).

3. if the node has a left subtree but no right subtree, then we can simply attach the left subtree to the node’s parent;

4. if the node to be deleted has both left and right subtrees, then the simple way to proceed is the following (see also Figure 11(ii)):
   - find the largest node in the left subtree of the node to be deleted (in Figure 11(ii) 20 is the maximum value in the left subtree of 23);
   - copy such value in the node to be delete (in the example, 20 is written in the root of the tree);
   - delete the largest node from the left subtree (in the example, remove the node containing 20 from the subtree rooted in 18);

Note that the last step is a recursive call to the procedure which deletes nodes from a binary search tree. Nevertheless, in this recursive case we are guaranteed that the node to be removed does not have a right subtree (i.e., it will be either case 1 or case 3 above).

4.1 Cost of Searching

If we consider searching one key in a given tree:

**Best Case:** The best case is the one in which we search for a given key and this key is present in the root of the tree. One comparison is sufficient to produce the result.

**Worst Case:** the tree is completely unbalanced and is reduced to a straight sequence of nodes. In this case, if there are \( n \) nodes in the tree, then we may take up to \( 2 \ast n \) comparisons to produce an answer.

What about the case in which we have a given tree and we can make any arbitrary number of queries in it? What is the best case?

It turns out that the best case is the one in which all the paths in the tree have the same length. This occurs if the tree is a complete tree (each internal node has exactly two successors, and the leaves belongs all to the same level). In this case, it is easy to observe that each search will take up to \( \lg n \) comparisons, if \( n \) is the number of nodes in the tree.
4.2 Binary Search Tree ADT

4.2.1 Specification

Set of values: the set of all possible values is represented by the set of all possible binary search trees over a given domain.

The content of each node is a non-negative integer number representing the value of the key. Keys are stored in the tree according to the conditions described above.

Operations:

- **Constructors**: the only constructor needed is the one to create a new object of type tree. Initially the tree created is empty.

  CreateSTree
  
  - preconditions: none
  - postconditions: returns a new object of type binary tree containing, as value, the empty tree (i.e., the tree containing zero elements);

- **Destructors**: the only destructor needed is used to trash a tree which is not needed any longer.

  DeleteSTree
  
  - preconditions: receives an existing tree as input
  - postconditions: none

- **Inspectors**: the following inspectors are included in the type specification:

  1. **EmptySTree** used to verify whether a given tree is empty or not
     
     - preconditions: receives a tree as input
     - postconditions: produces true (i.e., 1 in C) if the tree is empty, false (i.e., 0 in C) otherwise
  
  2. **EqualSTrees** used to verify whether two trees have the same value (i.e., they contain the same elements and in the same order).
     
     - preconditions: receives two trees as input
     - postconditions: returns true if the values of the two trees contain the same elements and in the same order, false otherwise.

  3. **FindKey** determines whether a key is present in the search tree
     
     - preconditions: receives a tree \( T \) and a key \( k \)
     - postconditions: it returns the subtree rooted in the searched key (empty tree if the key is not in the tree)

- **Modifiers**: the following modifiers are included

  1. **InsertKey** inserts a new key in the tree;
     
     - preconditions: receives in input a tree and a key
     - postconditions: inserts the key in the tree according to the rule of search trees; returns the resulting tree

  2. **DeleteKey** remove a key from the tree
     
     - preconditions: receives a tree and a key
     - postconditions: deletes the key from the tree; returns the modified tree
4.2.2 Implementation using Linked Structures

We are going to develop a library that implements binary search trees using linked structures. We assume that we are going to use this library as an alternative to hash tables to store data. This means that each node in the tree is going to contain two things: a key (an integer number) and a data (of type base_type). We will use the functionalities of binary search trees described earlier to manage the keys in the tree. Finding a key will imply giving a key and return the associated data.

The file bst.h contains:

```c
/*-----------------------------------------------*/
/* BINARY Search TREES ADT */
/*-----------------------------------------------*/
#include "genlib.h"

typedef int base_type;

typedef struct tree_type *BST;

/*-----------------------------------------------*/
/* Constructors */
/* create an empty tree */
/*-----------------------------------------------*/
BST CreateSTree();

/*-----------------------------------------------*/
/* Destructors */
/* destroy an existing tree*/
/*-----------------------------------------------*/
void DeleteSTree(BST t);

/*-----------------------------------------------*/
/* Inspectors */
/*-----------------------------------------------*/
/* Is the current tree empty? */
bool EmptySTree(BST t);

/* Are two trees identical? */
bool EqualTrees(BST t1, BST t2);

/* Find a key in the tree */
BST FindKey (BST t, int key);

/* Get Root of the tree */
base_type GetSRoot (BST t);
```
The file bst.c contains the implementation of the ADT and contains the following code.

```c
#include "bst.h"

/* Node of the tree */
struct tree_type
{
    int key;       /* content of the node */
    base_type data; /* the data associated to the key */
    BST left;      /* left child */
    BST right;     /* right child */
};

typedef struct tree_type Node;
```

```c
/* OPERATIONS */
/* CreateTree: */
/* takes a value and */
```
/* creates a tree with */
/* one node containing */
/* such value */
/*-----------------------*/

BST CreateSTree()
{
    return NULL;
}

/*-----------------------*/
/* DeleteSTree: */
/* takes a tree and */
/* destroys it */
/*-----------------------*/

void DeleteSTree(BST t)
{
    if (t == NULL)
        return;
    else
    {
        DeleteSTree(t->left);
        DeleteSTree(t->right);
        free(t);
    }
}

/*-----------------------*/
/* EmptySTree: */
/* takes a tree and */
/* verifies if it is */
/* empty */
/*-----------------------*/

bool EmptySTree (BST t)
{
    return (t == NULL);
}

/*-----------------------*/
/* EqualSTrees: */
/* takes two trees and */
/* checks if they are */
/* equal */
/*-----------------------*/

bool EqualSTrees(BST t1, BST t2)
{
    if (t1 == NULL && t2 != NULL)
        return FALSE;
if (t1 != NULL && t2 == NULL)
    return FALSE;
if (t1 == NULL && t2 == NULL)
    return TRUE;
if (t1->key == t2->key && t1->data == t2->data)
    return (EqualSTrees(t1->left,t2->left) &&
            EqualSTrees(t1->right,t2->right));
else
    return FALSE;
}

/*--------------------------------------------------------------*/
/* FindKey: */
/* takes a tree and a */
/* value and determines*/
/* the position of the */
/* value in the tree */
/* (empty tree if value*/
/* not in the tree) */
/*--------------------------------------------------------------*/
BST FindKey (BST t, int key)
{
    if (t == NULL)
        return NULL;
    if (key == t->key)
        return t;
    if (key < t->key)
        return FindKey(t->left,key);
    else
        return FindKey(t->right,key);
}

/*--------------------------------------------------------------*/
/* GetSRoot */
/* takes a tree and */
/* returns the data in */
/* the root */
/*--------------------------------------------------------------*/
base_type GetSRoot (BST t)
{
    if (t == NULL)
        Error("Accessing an empty tree");

        return (t->data);
}

/*--------------------------------------------------------------*/
/* InsertNode: */
/* takes a tree, a value*/
/* and a position (0,1)*/
/* and inserts new node */
/* in the indicated pos.*/
/*****************************/

BST InsertKey(BST t, int key, base_type data)
{
    BST new;
    BST root, parent;

    new = malloc(sizeof(Node));
    if (new == NULL)
        Error("Cannot create new node\n");

    new->key = key;
    new->data = data;
    new->left = new->right = NULL;

    if (t == NULL)
        return new;

    root = t;
    parent = NULL;
    while (t != NULL)
    {
        parent = t;
        if (key == t->key)
            return root;
        if (key < t->key)
            t = t->left;
        else
            t = t->right;
    }

    if (key < parent->key)
        parent->left = new;
    else
        parent->right = new;
    return root;
}
/*****************************/

/* DeleteKey */
/* takes a tree, */
/* and a key and removes*/
/* the key from tree */
/*****************************/

BST DeleteKey(BST t, int key)
{
    BST parent, root;

if (t == NULL)
    return NULL; /* tree empty, nothing to remove */

if (! FindKey(t,key))
    return t; /* key not in tree - nothing to remove */

parent = NULL;
root = t;
while (key != t->key)
{
    parent = t;
    if (key < t->key)
        t = t->left;
    else
        t = t->right;
}

/* case 1: remove a leaf */
if (t->left == NULL && t->right == NULL)
{
    if (parent == NULL) /* remove the only node in tree */
    {
        free(t);
        return NULL;
    }
    else
    {
        if (t == parent->left)
            parent->left = NULL;
        else
            parent->right = NULL;
        free(t);
        return root;
    }
}

/* case 2: remove a node with only a left child */
if (t->left != NULL && t->right == NULL)
{
    if (parent == NULL) /* node is the root */
    {
        return t->left;
    }
    else
    {
        if (t == parent->left)
            parent->left = t->left;
        else
            parent->right = t->left;
        free(t);
        return root;
    }
}
/* case 3: remove a node with only a right child */
if (t->right != NULL && t->left == NULL)
{
    if (parent == NULL) /* node is the root */
    {
        return t->right;
    }
    else
    {
        if (t == parent->left)
            parent->left = t->right;
        else
            parent->right = t->right;
        free(t);
        return root;
    }
}

/* case 4: remove a node which has two children */
if (t->left != NULL && t->right != NULL)
{
    BST max;
    /* step 1: find max in the left subtree */
    max = t->left;
    while (max->right != NULL)
        max = max->right;
    /* copy the max of the left subtree in t */
    t->key = max->key;
    t->data = max->data;
    /* delete the max from the left subtree */
    t->left = DeleteKey(t->left, max->key);
    return root;
}

How can we extend this to implement also the operations FindMax and FindMin?

5 Additional Considerations

5.1 Some Definitions

Some additional terminology related to trees:

- a binary tree is complete if all the leaves of the tree are either:
  - all at the same level;
the are at two adjacent levels;
and, furthermore, all the leaves on the bottommost level are placed as far to the left as possible. Figure 12(i) illustrates a complete tree, while 12(ii) and 12(iii) are not complete trees (the first does not have all leaves pushed to the left, the second does not have all the leaves in two adjacent levels).

Figure 12: Complete and Incomplete Trees

- Consider a binary search tree. Is there any form of traversal that will provide you automatically with the sorting of the elements present in the tree? E.g., if we insert, in the following order, 10, 5, 18, 7, 2, 24, 20, 11 in a binary search tree, is there a traversal that will give us 2, 5, 7, 10, 11, 18, 20, 24?
If we study the cost of performing the various operations on the binary search trees, we can conclude the best behaviour can be achieved when the tree has a number of level which is proportional to \( \log_{2} n \). In this case, searching a key will take at most \( \log_{2} n \) steps, as well as inserting it.

The problem occurs when the depth of the tree is not \( \log_{2} n \). E.g., in the worst case we can have that all the nodes of the tree are on the same branch (of length \( n \)). Thus it would be nice to be able to maintain during insertion a balanced tree. A binary tree is said to be balanced if, for every node \( K \) in the tree, the height of its two subtrees differs at most of one. For example, in Figure 12(i) the tree is balanced (e.g., the two subtrees of node \( a \) have height 3 and 2); the tree in Figure 12(iii) instead is not balanced: the left subtree of node \( c \) has height 4 while the right subtree has height 2.

AVL trees are a a special kind of search trees in which there is guarantee that after each insertion the tree is balanced (and, additionally the whole resulting tree has minimum height). This means that the heights of the two subtrees of each node are either equal or they differ by one.

More formally, an AVL tree is a binary search tree which is either empty or it satisfies the following conditions:

- the left subtree is an AVL tree
- the right subtree is an AVL tree
- the difference between the heights of the left and right subtrees is at most one;

For example, the first two trees in figure 13 are AVL trees, while the third is not.

![Figure 13: AVL Trees](image)

### 6.1 Representation of AVL Trees

Let us represent in C AVL trees similarly as ordinary binary trees. Each node in the tree is going to be represented by the following structure:

```c
struct avl_node
{
    int value;  /* value stored in the node */
    struct avl_node *left; /* left subtree */
    struct avl_node *right; /* right subtree */
    int bal;     /* parameter */
};
```

The field `bal` can contain one of three possible values: 0 or 1 or 2. This is used to keep track of how the tree rooted in such node is balanced. We will use the following convention:

- `bal=0` implies that the left subtree of the node has height one greater than the right subtree;
• $\text{bal} = 1$ implies that the left and right subtree have the same height;
• $\text{bal} = 2$ implies that the right subtree has height one greater than the left subtree.

6.2 Balancing an AVL Tree

Insertion or deletion of elements may lead an AVL tree to lose its properties. E.g., inserting a new node can create situations where one subtree of a node has height which is 2 or more greater than the height of the other subtree—thus violating the AVL conditions.

Let us define a tree to be left-high if its left subtree has height greater than its right subtree. Symmetrically, let us call a tree right-high if its right subtree has height greater than its left subtree.

We can recognize four cases of non-balanced situations:

1. Left of left: a subtree of a tree which is left-high has become itself left-high (e.g., Figure 14(i));
2. Right of right: a subtree of a tree which is right-high has also become right high (e.g., Figure 14(ii));
3. Right of Left: a subtree of a tree which is left-high has become right high (e.g., Figure 14(iii));
4. Left of right: a subtree of a tree which is right-high has become left-high (e.g., Figure 14(iv)).

Each of the above four cases can be fixed according to a well-defined set of transformations. These transformations are meant to be applied to the smallest subtree which is not AVL.

The transformation rules are the following:

1. Left of left: the tree can be balanced using a rotate right transformation; this is realized by making the left child into the root of the tree, and the previous root into the right child. The process is described in Figure 15 with an example.
2. Right of right: the process is symmetrical to the one described in the previous point; a rotation to the left is needed to rebalance the tree—as in Figure 16.
3. Right of left: this situation requires the use of two rotations to re-establish the balance in the tree. The first is a left rotation which takes place inside the left subtree (the one which is right-high); the second is a right rotation which is applied to the whole unbalanced tree. Figure 17 illustrates the scheme and one example.
4. Left of right: it is performed in a manner symmetrical to what described in the previous case. Figure 18 illustrates an example.
6.3 Insertion in AVL Trees

The insertion of a node in the AVL tree proceeds is composed by two steps:

1. the new node is inserted in the tree exactly as in a normal binary search tree;

2. the newly modified tree has to be transformed in order to guarantee that it still satisfies the AVL property.

Let us consider the path that goes from the root of the tree to the newly added node \( N \). Let us consider the node \( M \) in such path which is closer to \( N \) and has a value in the \( \text{bal} \) field different from 1 (i.e., either one of its subtree has height greater than the other). This node is called pivot node.

We have three possible cases that can occur after the first step (insertion of the new node):

1. there is no pivot node. This means that for each node in the path, the insertion of the new node \( N \) has not lead to violation of the AVL property. Figure 19(i) and 19(ii) illustrates two examples of this situation (in the first case we insert the value 40 while in the second one with insert the value 55). In this case, the only operation required after the insertion is the update of the \( \text{bal} \) field of all the nodes in the path from the root to \( N \).

2. there is a pivot node \( P \), and the subtree of \( P \) to which the new node \( N \) belongs has height smaller than the other subtree. Once again no special actions are required, since the insertion of the new node has “better balanced” the tree. Figure 20(i) and 20(ii) illustrates two examples (first has insertion of node 5, second shows the insertion of node 45).

3. the last case is the one in which the pivot node \( P \) exists, and the subtree of \( P \) where \( N \) has been inserted already has height greater than the other subtree. Thus after this insertion, one subtree of \( P \) has height two greater than the height of the other subtree. Figure 21 illustrates a situation like this; if we add node 5, then the node containing 40 would be the pivot node and the insertion would create an unbalanced tree. Thus if we insert a node we need to modify the structure of the tree in order to recreate a balanced situation. There are two cases that can be considered, one called single rotation and the other called double rotation. The two cases are illustrated here below:

- we have a situation like the one in Figure 22. \( P_1 \) is the pivot, \( P_2 \) is the child of the pivot in the subtree with greater height, \( T_1, T_2, T_3 \) are the various subtrees as in the figure. Since \( P_1 \) is the pivot, we
are guaranteed that $T_1, T_2, T_3$ all have the same height. $A$ identifies the newly added node. Single rotation transforms the tree as shown in the figure, by lifting the child of the pivot and pushing down the pivot itself. The resulting tree is balanced. The case of Figure 21 falls in this class; Figure 23 shows the tree immediately after inserting the node and after the single rotation.

- single rotation does not satisfy all the cases. In fact, if (as in Figure 22) the new node is attached under $T_2$ instead of $T_1$, then the rotation will not produce a balanced tree.

The new case is illustrated in Figure 24. In this case we need to move one step closer to the new node in order to partition the tree in a number of components adequate to rebuild a balanced tree. In the Figure, $P_1$ is the pivot, $A$ is the new node; the subtrees $T_1, T_4$ have the same height; the subtrees $T_2, T_3$ have the same height; the height of $T_1$ is one greater than the height of $T_2$. 

Figure 16: Left Rotation
Figure 17: Right of Left

Figure 18: Left of Right
Figure 19: Case 1 of Insertion

Figure 20: Case 2 of Insertion
Figure 21: Case 3 of Insertion

Figure 22: Single rotation
Figure 23: Case 3 of Insertion

Figure 24: Double rotation