Tree ADT

November 1, 2004

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.

Figure 1: Modules of the Federal Government

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 – Graphs

A graph is a structure composed of a collection of interconnected elements. Each element in the graph is called *node*. A link between two nodes is called *edge*. Each edge links two nodes. Depending on the type of edges, we distinguish between *Directed* and *Undirected* graphs:

- in a directed graph, each edge has a *direction*, which means that the edge goes from one node to the other (e.g., think of them as one-way streets going from one node to another). We will typically draw directed edges as arrows, showing the direction of the edge. Figure 2 shows some sample directed graphs, where circles represent nodes and arrows represent edges.
- in an undirected graphs, the edges do not have any directions (i.e., intuitively, they can be traversed in either direction when moving from a node to another). Figure 3 shows some examples of undirected graphs.
Let us focus on directed graphs. If we have an edge going from node $a$ to node $b$, then we say that the edge is an \textit{incoming} edge for node $b$; similarly for an edge from node $a$ to node $b$ we say that the edge is \textit{outgoing} for node $a$.

Given a node $a$ in a directed graph, its \textit{in-degree} is the number of edges that enter the node; its \textit{out-degree} is instead the number of edges that are leaving the node. For example, in the graph on the right of Figure 2, node 3 has an in-degree equal to 2 and an out-degree equal to 1.

3 Trees

A tree is a special type of graph. The nodes are organized into a hierarchical structure. At the top of the tree lies a distinguished node called \textit{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 4.

More precisely, a tree can be formally defined as follows: a tree is a directed graph which satisfies the following properties:

- There exists exactly one node (called the \textit{root} of the tree) which has in-degree equal to zero.
- all the other nodes in the tree have in-degree equal to one.

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 \textit{children} of $n$. In Figure 4, nodes $N1, N2, N3$ are children of node $N$. In turn, $N$ is called the \textit{parent} of $N1, N2, N3$. $N$ 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 \textit{leaves} of the tree. In Figure 4, node $N2$ 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 \textit{internal}. Thus internal nodes are all those nodes that have a parent and at least one child.

Similarly, in Figure 1, the root is the node called \textit{Federal Government}; the children are the various nodes called \textit{Defense, Education}, etc. In the figure, the node called \textit{Army} is a leaf, since it does not have any children. The node called \textit{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 5 illustrates the concept in an example. The *height* of a tree is the number of different levels that are present in the tree.

![Diagram of a tree with levels](image)

Figure 5: Paths and Levels in a 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 5 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 4, N1 is the first child, N2 is the second child, and N3 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 6 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?

4 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 7), 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.

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

4.1 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 tree 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 \( \lceil \lg_2(n + 1) \rceil \).

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 \( h \).

### 4.2 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 8 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).

![Figure 8: Example of binary trees](image)

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 nodes);

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

**DeleteTree**

- preconditions: operates on a given tree
- 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: operates on the current, non-empty tree
     - 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: operates on the current, non-empty tree
     - 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: operates on a binary tree
     - postconditions: produces true if the tree is empty, false 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: operates on a binary tree and receives as input another binary tree
     - 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: operates on the current tree T
     - postconditions: if T is not empty, then it returns the value stored in the root of T, otherwise it gives an error

  6. **GetSize** returns the number of nodes in the tree
     - preconditions: operates on the current tree T
     - postconditions: computes how many nodes are there in the tree

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

  1. **InsertNode** inserts a new node in the tree;
     - preconditions: operates on a binary tree T and receives as input 2 arguments: the value x to store in the new node 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: operates on a binary, non-empty, tree and receives as input 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. **CopyTree** creates a copy of an existing binary tree
   - preconditions: operates on a binary tree
   - postconditions: creates an exact copy of a given binary tree

### 4.3 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 object, and each node contains references to its children (two references 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 reference can be maintained in each node, which maintains a reference to the parent of that node.

The whole tree, as well as any subtrees, can be simply represented as a reference to the object which represents the root of the tree. Thus the tree is encoded as a reference to a node. An empty tree is represented by a nil reference.

In terms of Java implementation, we encode the representation of each node of the tree. The node is capable of storing a data (for the sake of this example, each node contains an integer, called key) and the references to two other nodes (representing the two children of this node). Methods are provided to access and modify each component of a node.

```java
/**
 * Class used to represent one node of a binary tree
 * the node can store a character
 *
 * @author Enrico Pontelli
 ***/

public class Node {
    // data members (private)
    private int key; // the key stored in the node
    private BinaryTree left; // left child
    private BinaryTree right; // right child

    // OPERATIONS

    // CONSTRUCTOR
    /**
     * Creates a new node, storing a given data
     * the new node has no children to start with
     *
     * @param d the character to be stored in the node
     **/
    public Node(int d) {
        key = d;
    }
}
```
left = new BinaryTree();
right = new BinaryTree();
}

// INSPECTORS

/***
* Read the data stored in the node
* @return the character stored in the node
***/
public int getData()
{
    return key;
}

/***
* Read the left child of the node
* @return the reference to the left child
***/
public BinaryTree getLeftChild()
{
    return left;
}

/***
* Read the right child of the node
* @return the reference to the right child
***/
public BinaryTree getRightChild()
{
    return right;
}

// MODIFIERS

/***
* Modify the data part of the node
* @param d the new integer to be stored in the node
***/
public void setData(int d)
{
    key = d;
}

/***
* Modify the left child part of the node
* @param l the new left child
***/
/**
 * Modify the right child part of the node
 * @param r the new right child
 */
public void setRightChild(BinaryTree r)
{
    right = r;
}
}

Now we can proceed in constructing a class used to encode a binary tree. The binary tree will be represented as a reference to the Node object representing the root of the tree.

/**
 * Class representing the binary tree ADT
 * @author Enrico Pontelli
 */
public class BinaryTree
{
    // data members
    protected Node root;  // root of the tree

    // CONSTRUCTOR
    public BinaryTree()
    {
        root = null;
    }

    public BinaryTree(Node n)
    {
        root = n;
    }

    // INSPECTORS

    /**
     * Determines the left subtree of the tree
     * @return the left subtree
     */
    public BinaryTree getLeft()
    {
        BinaryTree t = root.getLeftChild();
        return t;
    }
}
return t;
}

/** *
 * Determines the right subtree of the tree
 * @return the right subtree
 **/ public BinaryTree getRight()
{
    BinaryTree t = root.getRightChild();
    return t;
}

/** *
 * Determines whether the tree is empty or not
 * @return true if the tree is empty
 **/ public boolean emptyTree()
{
    return (root == null);
}

/** *
 * Determines whether two trees are identical
 * @param t another tree
 * @return true if the two trees are equal
 **/ public boolean equals(BinaryTree t)
{
    if ( root == null && t.emptyTree() )
        return true;
    if ( (t.emptyTree() && root != null) ||
         (root == null && !t.emptyTree()) )
        return false;
    return ( t.getRoot() == root.getData() &&
             t.getLeft().equals(getLeft()) &&
             t.getRight().equals(getRight()) );
}

/** *
 * Determines the content of the root of the tree
 * @return the content of the root
 **/ public int getRoot()
```java
/**
 * Determines the size of the tree
 * @return the number of nodes in the tree
 */
public int getSize()
{
    if (root == null)
        return 0;

    return (1+getLeft().getSize() + getRight().getSize());
}

/**
 * produces a string representation of the tree
 * @return a string representation of the tree
 */
public String toString()
{
    if (root == null)
        return " ";
    else
        return root.getData() + " ( " + getLeft().toString() + " ) ( " + getRight().toString()+ " )" ;
}

// MODIFIERS

/**
 * insert a new node in the tree
 * @param c the integer (key) to be inserted in the new node
 * @param flag the position for the insertion (0=left, 1=right)
 */
public void insertNode(int c, int flag)
{
    BinaryTree b;
    Node n;

    if (flag != 0 && flag != 1)
    {
        System.out.println("ILLEGAL FLAG IN OPERATION");
        return;
    }

    n = new Node(c);
    if (root == null)
    { 
```
root = n;
}
else if (flag == 0)
{
if (root.getLeftChild().emptyTree())
    root.setLeftChild(new BinaryTree(n));
else
    System.out.println("Left Tree Already Present");
else
{
if (root.getRightChild().emptyTree())
    root.setRightChild(new BinaryTree(n));
else
    System.out.println("Right Tree Already Present");
}
}

/***
 * delete a subtree
 *
 * @param flag the position for the deletion (0=left, 1=right)
 ***/
public void deleteSubtree(int flag)
{
    if (flag != 0 && flag != 1)
    {
        System.out.println("Incorrect Flag");
        return;
    }
    if (flag == 0)
    {
        root.setLeftChild(new BinaryTree());
    }
    else
    {
        root.setRightChild(new BinaryTree());
    }
}

/***
 * creates a copy of the tree
 *
 * @return a copy of the tree
 ***/
public BinaryTree copyTree()
{
    if (root == null)
    return new BinaryTree();
}
Node n = new Node(root.getData());
n.setLeftChild(getLeft().copyTree());
n.setRightChild(getRight().copyTree());
return (new BinaryTree(n));
}

The intuition behind this implementation of Trees is the following. We have two types of entities: nodes and trees. A Node contains a data item (e.g., a character in our example) and the references to the two subtrees that are attached to this node. On the other hand, a tree is simply represented by the node which constitutes its root. This is illustrated in the Figure 9.