The ADT Binary Tree

The Binary Tree is a more general ADT than the linear list: it allows one item to have two immediate successors.

**Definition**

The **ADT Binary Tree** is a finite set of nodes which is either empty or consists of a data item (called the **root**) and two disjoint binary trees (called the **left** and **right subtrees** of the root), together with a number of access procedures.

The Binary Tree is a more general ADT than the linear list: it allows one item to have two immediate successors.

**Applications**

- representing 1:2 relations;
- representing sets;
- evaluating expressions.

Nodes with no successors are called **leaves**. The roots of the left and right subtrees of a node “i” are called the “**children** of i”; the node i is their **parent**; they are **siblings**. A child has one parent. A parent has at most two children.

The data organisations presented so far are linear in that items are one after another. The Binary Tree is a more general form of ADT, in which data are organised in a non linear, hierarchical form whereby one item can have more than one immediate successor.

A Binary Tree is a “position-oriented” ADT, as lists, stacks, queues. However, since it is not linear as the ADT lists we have seen so far, we will not reference items in a binary tree by using a position number. The Binary Search Tree is a value-oriented ADT whose elements are organised on the basis of their values.

All trees are **hierarchical** in nature. Intuitively, hierarchical means that a “parent-child” relationship exists between the nodes in the tree. If there is a link between a node “n” and a node “m”, and “n” is above node “m” in the tree, then “n” is the **parent** of “m”, and node “m” is a **child** of “n”. Children of the same parent are called **siblings**. Each node in a tree has at most one parent, and exactly one node, called the **root** of the tree, has no parent. A node that has no children is called a **leaf** of the tree.

The parent-child relationship between the nodes can be generalised to the relationships **ancestor** and **descendant**. The root of a tree is an ancestor of every node in the tree. A **subtree** in a tree is any node in the tree together with all its descendants. A **subtree of a node** r is a subtree rooted at a child of the node r.

To give a formal definition of what a tree is, we say that a binary tree is a set of node which is either empty, or is partitioned into three disjoint subsets: (i) a single node “r”, the **root**; and (ii) two (possibly empty) sets that are binary trees, called the **left** and the **right subtrees** of r. Each node in a binary tree has therefore no more than two children.

You have already seen uses of binary trees in the logic course. These are the trees used to define if a logical expression is a well-formed formula of propositional logic or predicate logic. Evaluation of expressions is a typical application of binary trees.

The nodes of a tree are all of the same type. We still assume the type to be “Object” (in general).
Example of Binary Tree

- Root: Rector
- Siblings: Dean of Science and Dean of Engineering
- Leaves: Head of Maths, Head of Physics, Head of Civil Eng, Head of Mech Eng
This slide provides basic access procedures for general binary trees. Additional access procedures are given in the next slide. What we have here are just the procedures related to the root of a tree, and the creation of a tree. Note that in the case of trees, we can either create an empty tree, or also create just a single node tree, or (as shown in the next slide) create a tree by passing a root item and the two sub-trees.

The implementation of general binary trees can be quite complex. We will therefore introduce example implementations for general binary trees only for some of the access procedures. We will instead concentrate on, and show in more detail in the next lecture, a particular type of binary tree, called a binary search tree.

Procedures are again defined here only in terms of their main post-conditions. Exceptions cases need to be included as part of the post-conditions for some of these procedures, as we will explain in the next slide.
This slide provides the remaining access procedures for general binary trees.

These procedures are defined here only in terms of their main post-conditions. Exception cases need to be included as part of the post-conditions to all but the first access procedure given here. The circumstances for exception are when the tree is empty, as for each of these procedures there will be no root node to attach left/right children or left/right subtrees.

Note that two other access procedures might be useful in managing binary trees. These are “getRightSubtree”, and getLeftSubtree”, which respectively return the left and right subtree of a binary tree’s root without modifying the tree.
In this slide I have listed some of the main axioms that the access procedures for an ADT binary tree have to satisfy.

Note that, the procedures for constructing a tree are also attachLeft, attachRight, attachLeftSubtree, attachRightSubtree. A full axiomatic definition of binary search trees will also need to include axioms for the access procedures given in this slide, but applied to a binary tree constructed by applying these other constructor operations.
Additional Definitions

1. A path of a tree $T$ to a subtree $T_k$ is the sequence $T_1, T_2, \ldots, T_k$ of trees, where for each $1 \leq i < k$,
   \[ T_{i+1} = T_i.\text{detachLeftSubtree} \text{ or } T_{i+1} = T_i.\text{detachRightSubtree}. \]

2. The height of a tree is defined as
   \[ \text{height}(T.\text{createTree}( )) = 0; \]
   \[ \text{height}(T.\text{createBTree}(\text{Item},LTree,RTree)) = 1 + \max(\text{height}(LTree),\text{height}(RTree)); \]

3. Shortest Path in a tree is defined as
   \[ \text{ShortestPath}(T.\text{createTree}( )) = 0 \]
   \[ \text{ShortestPath}(T.\text{createBTree}(\text{Item},LTree,RTree)) = 1 + \min(\text{ShortestPath}(LTree),\text{ShortestPath}(RTree)); \]

4. A perfectly balanced tree (or full) is a tree whose height and its shortest path have the same value.

Trees come in many shapes. Two given trees might be different from each other even though they might contain the same number of nodes. The height of a tree is the number of nodes on the longest path from the root to a leaf. A different definition of height could be considered which uses recursion. This is the definition given in this slide.

A perfectly balanced tree (or full tree) of height $h$ is a tree in which every node that is at a level less than $h$ has two children. This means that each node has left and right subtrees of the same height. In terms of number of leaves, among all binary tree of height $h$, a full binary tree is a tree with as many leaves as possible and all the leaves are at the same “level” in the tree. An equivalent characterisation is given in this slide. A perfectly balanced tree is a tree whose height and shortest path have the same value.
Theorem
The number of nodes in a perfectly balanced tree of height $h \ (\geq 0)$ is $2^h - 1$ nodes.

Proof
The proof is by induction on $h$:
Base Case: $h=0$. Empty tree is balanced; $2^0 - 1 = 0$;
Inductive Hypothesis: suppose the theorem holds for $k$, with $0 \leq k \leq h$.
In this case a perfectly balanced tree has $2^k - 1$ nodes.
We want to show it holds for a perfectly balanced tree with height $k+1$.
A perfectly balanced tree of height $k+1$ consists of a root node and 2 subtrees each of height $k$. Total number of nodes is:
\[
(2^k - 1) + 1 + (2^k - 1) = \\
= 2*2^k - 1 = 2^{k+1} - 1
\]
So a perfectly balanced tree of height $h$ has $2^h - 1$ nodes
Additional Definitions (continued)

A perfectly balanced tree with \( n \) nodes has a height of \( h = \log_2(n+1) \)

eg.: for \( n=7 \)

A Skewed Tree
(or degenerate tree)

Definitions:
The level of a node is 1 if the node is the root of the tree; otherwise it is defined to be 1 more than the level of its parent.

A complete tree of height \( h \) is a tree which is full down to level \( h-1 \), with level \( h \) filled in from left to right.

This slide completes the set of definitions of main features of binary trees. In particular, we have seen in the previous slide a theorem that proves the relation between the height of a full (or perfectly balanced) binary tree and its nodes. This relation can be used also to define the height of a given full tree composed of a given number of nodes. We have also given here an example of a full tree.

Another type of tree is the skewed tree, which is essentially a linked list, as shown in this slide.

Other important definitions on trees is that of “level of a node” and that of “complete tree”. A level of a node is a recursive definition, which assigns value 1 to the root node and assigns value \( = (1+ \text{level of its parent node}) \) to any node that is not a root node.

A complete binary tree is a particular type of tree. We normally say that a binary tree is a complete tree of height \( h \), if it is a perfectly balanced tree of height \( h-1 \), and its nodes at level \( h \) are completed started from left to right. What does this mean exactly?

The fact that it’s perfectly balanced of height \( h-1 \), means that all its nodes with level less than \( h-1 \) have two children. So the nodes with level \( h-1 \) may or may not have children. However, if a node with level \( h-1 \) has children, then all nodes to its left with the same level must have two children each, and if in particular it has only one child then it must be its left child. I’ve given here an example of a complete tree of height 3:
Methods for “visiting” each node of a tree.

Given the recursive definition of a binary tree, we could think of using a recursive traversal algorithm:

```
pseudocode
traverse(binaryTree)
    if (binaryTree is not empty)
        { traverse(Left subtree of binaryTree’s root);
        traverse(Right subtree of binaryTree’s root); }
```

When shall we visit the root node r?

- visit r before traversing both r’s subtrees;
- visit r after it has traversed r’s left subtree; but before traversing r’s right subtree
- visit r after it has traversed both of r’s subtrees.

Programs that use tree structures often need to visit and process each node of a given tree. Methods for traversing or visiting each node of a tree are called “tree traversals”. For example, suppose that we have a tree where each node contains an integer, and we want to print a list of all the integers in the tree. Then for the purpose of this example, we can assume that visiting a node simply means displaying the data portion of the node. With the recursive definition of a binary tree in mind, we can construct a recursive traversal algorithm as shown in this slide. A binary tree is in fact either empty or it is of the form root node “r”, left sub-tree and right sub-tree as shown in the slide. If the tree is empty, then the traversal algorithm is not supposed to take any action (empty tree would therefore be the base case of our algorithm). If the tree is not empty, the traversal algorithm must perform three tasks: visit the root node, traverse the left sub-tree and right sub-tree of the root node. The general form of a traversal algorithm should therefore be as shown above in pseudocode.

However, the algorithm given here is not complete as it does not include any operation on (or visiting) the root node r (which in the recursive execution means operation for visiting each node of the tree). We can have three choices, given the form of traversal algorithm sketched here:

1) we could visit the root node “r” before the algorithm traverses both of r’s subtrees;
2) we could visit the root node r after the algorithm has traversed r’s left subtree but before traversing r’s right subtree;
3) we could visit the root node r after the algorithm has traversed both of r’s subtrees.

These three different ways correspond to three different traversal algorithms for binary trees, called respectively pre-order, in-order and post-order traversal, which are defined in the next slide.
The three different choices of when to visit the root node give rise to three different traversal algorithms.

The pre-order traversal is when the root node is visited first before its’ left sub-tree and right sub-tree are processed (or traversed). This is why in the pseudocode given here the operation “Display the data in the root node” is performed before the two recursive calls of preorder. The order of access in this case would be “root, left, right”.

The in-order traversal is when the root node is visited after the root’s left sub-tree has been visited and before the root’s right sub-tree is traversed. This is why in the pseudocode given here the operation “Display the data in the root node” is performed between the two recursive calls of inorder. The order of access in this case would be “left, root, right”.

The post-order traversal is when the root node is visited after the root’s left and right sub-trees have been visited. In this case, the operation “Display the data in the root node” is performed after the two recursive calls of postorder. The order of access in this case would be “left, right, root”.

---

**Pre-order:**
Visit the root node before traversing the subtrees

```
preorder(binaryTree)
if (binaryTree is not empty)
{ Display the data in the root node;
  preorder(Left subtree of binaryTree’s root);
  preorder(Right subtree of binaryTree’s root); }
```

**In-order:**
Visit the root after traversing left sub-tree and before right sub-tree.

```
inorder(binaryTree)
if (binaryTree is not empty)
{ inorder(Left subtree of binaryTree’s root);
  Display the data in the root node;
  inorder(Right subtree of binaryTree’s root); }
```

**Post-order:**
Visit the root after traversing left sub-tree and before right sub-tree.

```
postorder(binaryTree)
if (binaryTree is not empty)
{ postorder(Left subtree of binaryTree’s root);
  postorder(Right subtree of binaryTree’s root);
  Display the data in the root node; }
```
Examples:

**Display the nodes of the tree:**

- **Pre-order:** 14, 17, 9, 53, 30, 50, 11;
- **In-order:** 9, 17, 30, 53, 50, 14, 11;
- **Post-order:** 9, 30, 50, 53, 17, 11, 14;

**Tree representation of the algebraic expression: (a+(b-c))*d.**

- **Pre-order:** * + a – b c d.
- **In-order:** a + b – c * d.
- **Post-order:** a b c - + d *

This gives the **prefix notation** of expressions;

This requires bracketing for representing sub-expressions.

It is called **infix notation** of expressions;

This gives the **postfix notation** of expressions.

Here we show two examples of tree traversal.

Our second example is a tree representation of an algebraic expression. Traversing the tree in pre-order gives a sequence that is called **prefix notation** of a given algebraic expression, where the operators precede their operands. Traversing the tree in inorder gives a sequence called the **infix notation** of an algebraic expression, which however needs bracketing in order to preserve the fact that subtrees are sub expressions of the given one. Finally, traversing the tree in postorder gives a sequence called **postfix notation**, where operators appear after the operands.

Note that infix notation, although conventional, is inconvenient for automatic execution, because:

- brackets may be needed to clarify expressions
- the order in which the operators/operands are required is not the order in which they appear.
Pre-order, in-order, and post-order tree traversals are **depth-first** traversals, as (except for the root in pre-order) nodes further away from the root node (i.e. deepest nodes) are visited first.

**Recursive algorithms**: the run-time implementation of recursion keeps track of the progress through the tree.

**Iterative algorithms**: must keep track of higher-level nodes explicitly, for example, using a stack. Nodes can be pushed during progress away from the root, and can be popped to move back towards the root.

**Breadth-first** traversal visits all the nodes at one level, before moving to another level. Root first, then the child nodes of the root, then the children of the children in the order in which the root’s children were visited.

A **queue** of children of a node must be built up when the node is visited, so that the correct ordering is maintained at the next level.

Breadth-first: 14, 17, 11, 53, 30, 50
A Static Implementation of Complete Trees

```
tree[i] = node numbered i
```

**Advantages:**

1. Data from the root always appears in the [0] position of the array;
2. Suppose that a non-root node appears at position [i]. Then its parent node is always at location [(i-1)/2] (using integer division).
3. Suppose that a node appears at position [i] of the array. Then its children (if they exist) always appear at locations [2i+1] for the left child and location [2i+2] for the right child.

For all the ADTs considered so far in this part of the course, we have provided a definition of the interface for the ADT, which defines the access procedures for the ADT, and various classes that implement this interface as example implementations (e.g., static and dynamic implementation of an ADT). In the case of trees, we have seen so far that we can have different types of trees and that more specific access procedures might be useful for some type of trees but not for others. It is therefore more difficult to present example implementation of binary trees using this same distinction (basic interface and different implementation classes). We will still see, however, examples of static vs dynamic implementation of binary trees.

As for static implementation, we consider here the case of array-based implementation of just complete binary trees, since this implementation is easier when the binary tree is complete. In a complete tree, all the levels are full except maybe for the lowest level. At the lowest level, the nodes are as far left as possible.

The whole tree can be stored in a simple array, starting by storing the root’s item in the [0] location of the array, then taking the two nodes at the next level in the tree and placing them left node first into the array at locations [1] and [2] respectively, and continuing in this way until all the nodes have been stored in the array. The simple representation is due to the fact that the tree is a complete tree, and therefore we have define specific formulae for accessing in the array to the parent (or children) node(s) of a given node. These formulae are given in this slide. They also make it easy to implement algorithms for traversing the tree. A class `TreeArrayBased` will then have at least two private instance variables: (a) the array itself, and (b) a second instance variable that keeps track of how much of the array is used (eg.int free = 6). The actual links between the nodes are not stored. Instead, they exist via the formulas given in this slide, which determine where an element (child or parent of a node) is stored in the array based on the element position in the tree.

Non complete binary trees could also be implemented using an array, but the array data structure is more complex than the one given here, since it is necessary to keep explicit reference to the children of a node when and if they exist.
A binary tree can be represented by its individual nodes, where each node is an object of a new binary tree node class, similar to the class “node” used for the linked lists, etc. The basic idea in a dynamic implementation of a binary tree is that each node of the tree can be stored as an object of a class called “TreeNode”. This class contains private instance variables that reference other nodes in the tree. Since these variables are private the class TreeNode should also include access procedures for getting or for changing the values of the attributes of a node. We’ll see in the next slide the full implementation of the class TreeNode.

The entire tree is represented as a reference to the root node. If the tree is empty, the root is null. The constructor BinaryTree( ) given here implements the access procedure createEmptyTree( ) given in slide 3.

In the next slide we see the basic methods of the class TreeNode and some example implementations of other access procedures for the ADT binary tree.
The class TreeNode

```java
public TreeNode(Object newItem, TreeNode leftChild, TreeNode rightChild) {
    item = newItem;
    left = leftChild;
    right = rightChild;
} // end constructor.
```

Getting and setting data and links:

```java
public Object getItem() {
    // returns the item field of the particular node.
}

public void setItem(Object newItem) {
    //Sets the item field to the new value newItem;
}

public TreeNode getLeft() {
    // returns the reference to the left child;
}

public void setLeft(TreeNode leftChild) {
    //Sets the left reference to leftChild;
}

public TreeNode getRight() {
    // returns the reference to the right child;
}

public void setRight(TreeNode rightChild) {
    //Sets the right reference to rightChild;
}
```

This slide provides the list of the basic methods that a class TreeNode should have, in terms of their respective post-conditions. Other constructors could also be defined, which for instance, create a TreeNode with just the item and no children. How would you implement such a constructor?

Given this new class for binary tree nodes, how would we implement the access procedures for a binary tree? As shown in the previous slide we would need a class called “BinaryTree”, which includes just the instance variable “root”, which refers to the root node of the entire tree. Some example implementations of some access procedures for the BinaryTree class are given in the next slide.
In some previous slide we have given a default constructor for the class BinaryTree that is supposed to create an empty binary tree. Other constructors can be defined, for instance, to create a binary tree with just one node (an example implementation is given at the top of this slide), or to create a tree from a reference to a root node (an example implementation is given on the top of the next slide). This last type of constructor could be very useful to define implementations of other access procedures, as shown. If used as an auxiliary constructor, it should be defined as protected in order to prevent client classes from using it directly. This is because client classes are in general not able to access node references directly, as node references are private to the class BinaryTree.

In the implementation of the access procedure “getRootItem” we need to check that the binary tree is not empty. If you look at the axioms given in slide 4, the call of this method on an empty tree should flag an error. An example implementation of TreeException can be the following:

```java
class TreeException extends java.lang.RuntimeException {
    public TreeException(String s){
        super(s);
    }
}
```

An example implementation is also given for the access procedure “getLeftTree”. This is supposed to return the left subtree of a binary tree’s root. To do so, we could use the protected constructor of BinaryTree and create a new binary tree that has the root’s left node as its root node.

What would the implementation of the access procedure “getRightTree” look like? This is left as exercise.
Some Methods of the Class BinaryTree (2)

```java
protected BinaryTree(TreeNode rootNode) {
    root = rootNode;
} // end protected constructor

public BinaryTree detachLeftSubtree() throws TreeException {
    if (isEmpty()) { throws new TreeException("TreeException: Empty tree"); }
    else { BinaryTree leftTree;
            leftTree = new BinaryTree(root.getLeft());
            root.setLeft(null);
            return leftTree;
    } // end getLeftTree
}

public BinaryTree attachLeftSubtree(BinaryTree leftTree) throws TreeException {
    if (isEmpty()) { throws new TreeException("TreeException: Empty tree"); }
    else { if (root.getLeft() != null) { throws new TreeException("Left subtree already exists"); }
            else { root.setLeft(leftTree.root);
                    leftTree.makeEmpty(); }
    } // end attachLeftSubtree
}
```

In this slide we give other example implementations of access procedures for a binary tree. Note that in the implementation of attachLeftSubtree we have used a method “makeEmpty()”. This can be defined as an auxiliary method for the class BinaryTree, which empties a given tree, by just setting its root node equal to null. In our example here, the use of leftTree.makeEmpty() is in order to guarantee that the left tree passed as parameter to the method attachLeftSubtree is now only referenced by the root node of the tree.

The remaining methods of the class BinaryTree can be implemented in a way similar to that shown in these last two slides. It is therefore left to you as a little exercise.
Summary

- Binary trees provide a hierarchical organisation of data, important in applications.

- The implementation of a binary tree is usually reference based. If the tree is complete, an efficient array-based implementation is possible.

- Traversing a tree is a useful operation; intuitively, it means to visit every node in the tree.