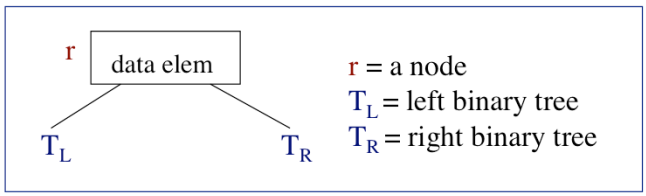
Course: Programming II - Abstract Data Types

The ADT Binary Tree

Definition

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



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.

Uni4: ADT Trees

Slide Number 1

The data structures presented so far are linear in that items are one after another. The Binary Tree, and we'll see later towards the end of this unit also general trees, are ADTs with a more complex structure, in which data are organised in a non linear, hierarchical form whereby one item can have zero, one or more than one immediate successor.

We are going to see two types of trees: binary trees and binary search trees, and only briefly present the notion of general trees and hint a possible implementation. Binary trees are "position-oriented" ADTs (as lists, stacks, queues) whose nodes, however, will be referred as left and right subtree's roots instead of position number. The Binary Search Trees (discussed in the next unit) are instead a value-oriented ADT whose elements are organised on the basis of their key values. General trees can be classified as unordered trees, where children of a node do not reflect the order in which they were added to the structure, and ordered tree where a child of a node is always inserted as its right most child.

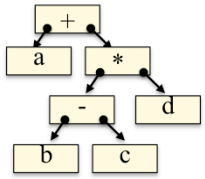
All trees are **hierarchical** in nature. Intuitively, hierarchical means that a "parent-child" relationship exists between the nodes in the tree. If there is a link between a node "n" and a node "m", and "n" is above node "m" in the tree, then "n" is the **parent** of "m", and node "m" is a **child**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

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.

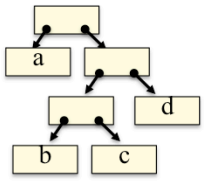
To give a formal definition of what a tree is, we say that a binary tree is a set of node which is either empty, or is partitioned into three disjoint subsets: (i) a single node "r", the **root**; and (ii) two (possibly empty) sets that are binary trees, called the **left** and the **right subtrees**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

Applications of Binary Trees

- Expression trees (e.g., in compiler design) for checking that the expressions are well formed and for evaluating them.
  - leaf nodes are operands (e.g. constants)
  - non leaf nodes are operators



- Huffman coding trees, for implementing data compression algorithms:
  - each leaf is a symbol in a given alphabet
  - code of the symbol constructed following the path from root to the leaf (left link is 0 and right link is 1)



a = 0, b = 100, c = 101, d = 11

There are many uses of binary search trees. One use is in the *expression tree,* which is a central data structure of in compile design. Expression trees are in fact used by compilers to recognize that arithmetic expressions are well formed and also to evaluate an arithmetic expression by recursively iterating through the tree. In an expression tree a leaf is an operand and a node that is not a leaf is an operator. Therefore, we can evaluate an expression by applying the root operator to the values obtained by recursively evaluating the left and right subtrees. In the case of binary trees, operators can be at most binary. More general expression tress can be constructed using general trees instead of binary trees, such as for instance for checking the well–formedness of an XML document (as it is the case in a DOM (document object model) library.

Another use of binary trees is in the Huffman coding algorithm. Huffman coding trees are binary trees where leaves are symbols of a given alphabet. The code of a symbol can be constructed by following the path from the root to the leaf of the symbol and concatenating a zero for each left link traversed and a 1 for each right link traversed. An example is given in this slide. Other uses of trees are for general trees where each node can have more than two children. Examples are given at the end of this lecture when we will briefly introduce general trees.

Some definitions

1. Nodes are arranged in levels. 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.

2. The height of a tree T is the number of levels in the tree. It can be defined, recursively, as

```
height(T.createTree( ) ) = 0;
height(T.createBTree(Item, LTree, RTree)) = 1+
                max(height(LTree), height(RTree));
```

3. The shortest path in a tree T is defined as

```
ShortestPath(T.createTree( )) = 0
ShortestPath(T.createBTree(Item, LTree, RTree)) = 1+
      min((ShortestPath(LTree), ShortestPath(RTree));
```

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

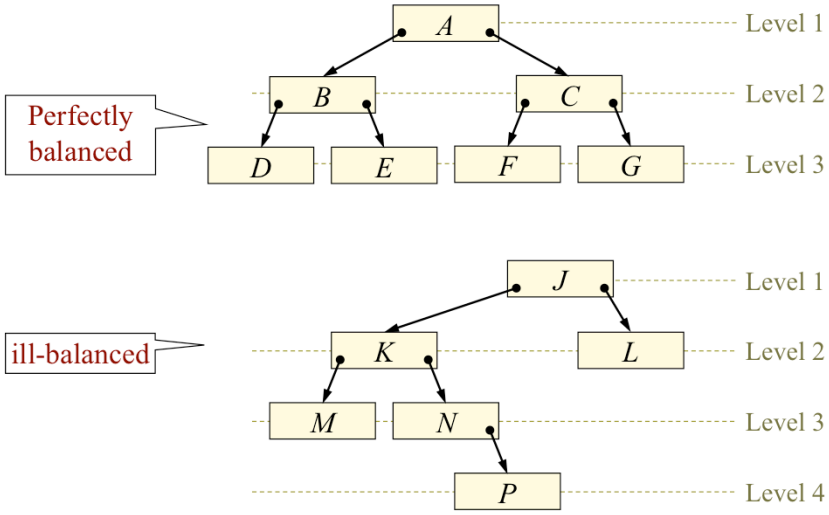
Trees come in many shapes. Two given trees might be different from each other even though they might contain the same number of nodes. We consider the *height* of a tree to be given by the number of levels in the tree. We can assume that nodes in a binary tree are organised in levels, each level representing the hierarchical position in the tree. So for instance the root node is at level 1, whereas the children of a root node is at level 2 and so on. If we consider the expression tree given in the previous slide this has 4 levels. Using this notion of level instead of the conventional notion of depth of a node, we can provide a recursive definition of height of a tree, as shown in this slide. So again, according to this definition, the height of the expression tree given in the previous slide is 4. We can think of the height of a tree as given by the number of nodes along the longest path from the root of the tree and a leaf.

Note that, alternative definitions are often given of depth of a node (number of links from the root to the node) and height of a node, number of links from the node to the deepest leaf in the tree. In this case the height of a tree is given by the height of its root. In this case the depth of an empty tree is -1 and the depth of a tree with just the root is 0 and so on. This is similar to our definition above just starting counting from -1 in the base case of an empty tree.

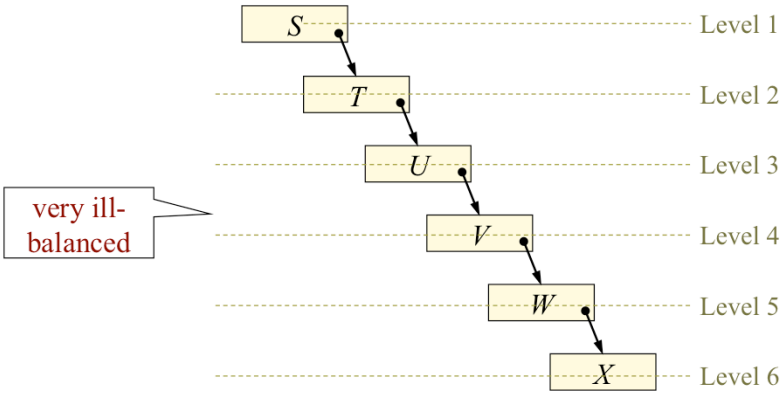
Consider now the definition of shortest path. The shortest path in the expression tree given in the previous slide is 2.

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. So in the previous slide the expression tree has shortest path 2 and height 4. We say in this case that the tree is not a perfectly balanced tree.

Example trees



Example trees



Theorem

The number of nodes in a perfectly balanced tree of height h (≥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 $0 \leq k$.

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:

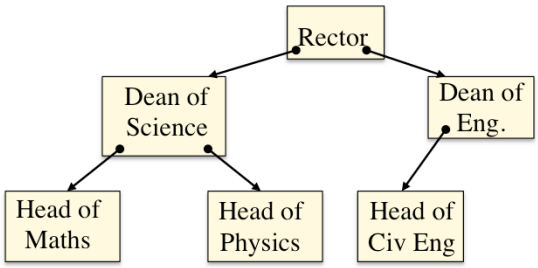
$$(2^k - 1) + 1 + (2^k - 1) =$$
$$= 2*2^k - 1 = 2^{k+1} - 1$$

So a perfectly balanced tree of height h has 2^h - 1 nodes

The relationship between height and number of nodes in a perfectly balance tree can be used also to define the height of a given full tree composed of a given number "n" of nodes. The height is the rounded up value of $\log_2(n+1)$.

Additional Definition

Definition:
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.



A complete tree

The height of a complete or perfectly balanced tree with n nodes is: $h = \log_2(n+1)$ (rounded up).

This slide completes the set of definitions of main features of binary trees. In particular, 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 to its next-to-last level, and its leaves on the last level are filled from left to right. What does this mean exactly?

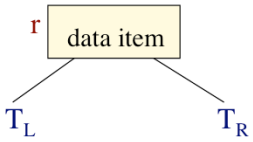
The fact that it's perfectly balanced up to height h-1, means that all its nodes with level less than h-1 have two children. So the nodes with level h-1 may or may not have children. However, if a node with level h-1 has children, then all nodes to its left with the same level must have two children each, and if in particular it has only one child then it must be its left child. The slide shows an example of a complete tree of height 3.

Course: Programming II - Abstract Data Types

Tree Traversals

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); }
```

Not complete: it doesn't include operation for visiting the root r

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.

Uni4: ADT Trees

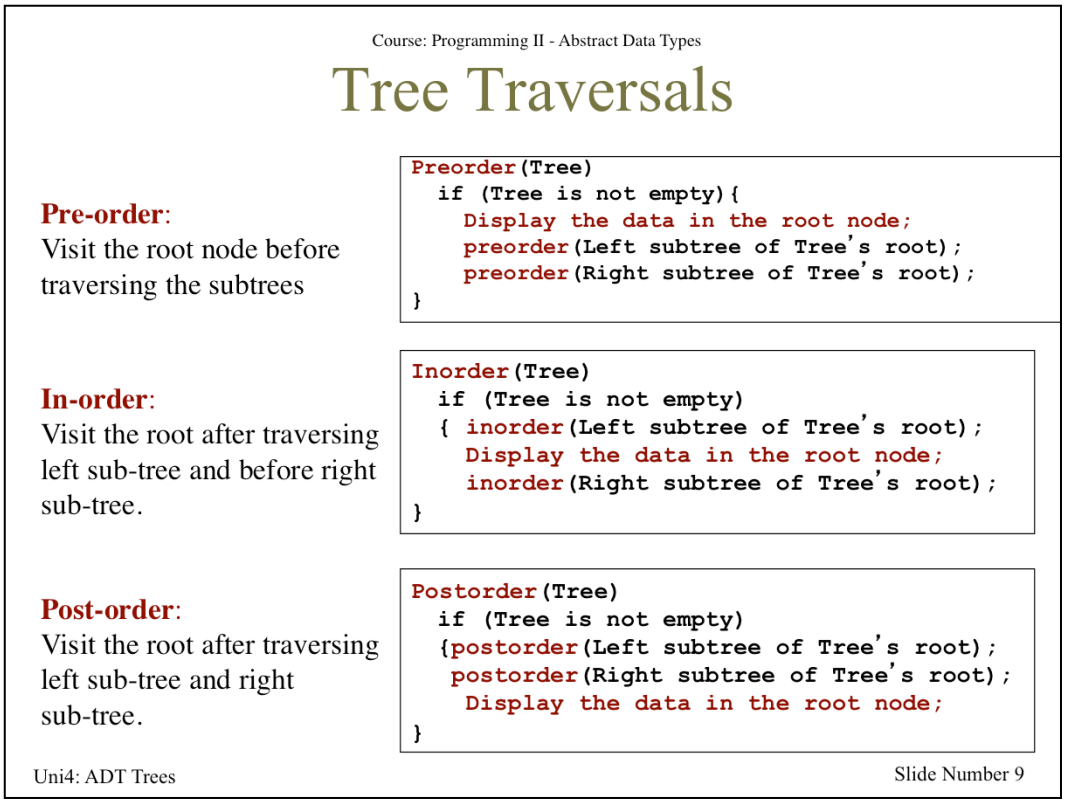
Slide Number 8

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 part 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.

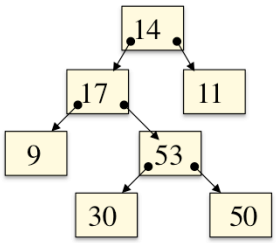
The pre-order traversal is when the root node is visited first before its' left sub-tree and right sub-tree are processed (or traversed). This is why in the pseudocode given here the operation "**Display the data in the root node**" is performed before the two recursive calls of preorder. The order of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

Examples of Tree Traversals

Display the nodes of the tree:

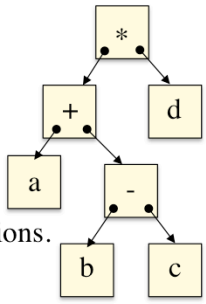


Pre-order: 14, 17, 9, 53, 30, 50, 11;

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

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

Algebraic expression tree: (a+(b-c))*d.



Pre-order: * + a – b c d .
This gives the prefix notation of expressions;

In-order: a + b – c * d
This requires bracketing for representing sub-expressions.
It is called infix notation of expressions;

Post-order: a b c - + d *
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.

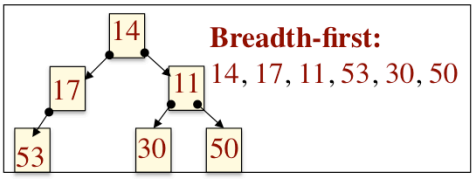
Depth-first and Breadth-first traversals

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. They explore one subtree before exploring another.

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.



Access Procedures

createEmptyTree( )
// post: creates an empty tree

createBTree(rootElem)
// post: creates a one-node binary tree whose root contains rootElem.

createBTree(rootElem leftTree, rightTree)
// post: creates a binary tree whose root contains rootElem, and has leftTree and
// post: rightTree, respectively, as its left and right subtrees.

attachLeft(newElem)
// post: Attaches a left child containing newElem to the root of a binary tree

attachRight(newElem)
// post: Attaches a right child containing newElem to the root of a binary tree

attachLeftTree(leftTree)
// post: Attaches leftTree as the left subtree of the root of a binary tree.

attachRightTree(rightTree)
// post: Attaches rightTree as the right subtree of the root of a binary tree.

This and the next slide provide main access procedures for binary trees, grouped into procedures for constructing a tree structure, for checking properties of a tree and for selecting parts of a tree. As basic Java constructor of a class that implements a binary tree, we can either create an empty tree, or create a single node tree, or create a tree by passing a root element and the two sub-trees.

Procedures are again defined in this slide and the next only in terms of their main post-conditions. For some of these procedures exceptions cases need to be included.

Access Procedures

isEmpty( )
// post: determines whether a tree is empty

getRootElem( )
// post: retrieves the data element in the root of a non-empty binary tree.

getLeftSubtree( )
// post: returns the left subtree of a binary tree's root.

getRightSubtree( )
// post: returns the right subtree of a binary tree's root.

detachLeftSubtree( )
// post: detaches and returns the left subtree of a binary tree's root.

detachRightSubtree( )
// post: detaches and returns the right subtree of a binary tree's root.

Axioms for ADT Tree

These are only some of the axioms that the access procedures have to satisfy, where Elem is an element, aTree, LTree and RTree are given Binary Trees:

1. (aTree.createTree( )).isEmpty = true
2. (aTree.createBTree(Elem, LTree, RTree)).isEmpty() = false
3. (aTree.createTree( )).getRootElem( ) = error
4. (aTree. createBTree(Elem, LTree, RTree)). getRootElem( ) = Item
5. (aTree.createTree( )).detachLeftSubtree( ) = error
6. (aTree.createBTree(Elem, LTree, RTree)). detachLeftSubtree( ) = LTree
7. (aTree.createTree( )).detachRightSubtree( ) = error
8. (aTree.createBTree(Elem, LTree, RTree)). detachRightSubtree( ) = RTree

In this slide I have listed only some of the main axioms that the access procedures for an ADT binary tree have to satisfy.

Note that, the access procedures for constructing a tree are also *attachLeft*, *attachRight*, *attachLeftTree*, *attachRightTree*. 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 using these other operations. So, for instance, we could also have axioms of the form:

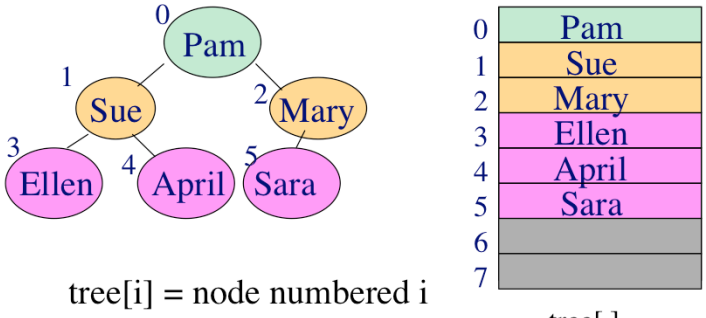
(aTree.attachRightTree(rightTree)).detachRightSubtree( ) = rightTree.

(aTree.attachLeftTree(leftTree)).detachLeftSubtree( ) = leftTree.

(aTree.attachRightTree(rightTree)).detachLeftSubtree( ) = aTree.getLeftSubtree().

(aTree.attachLeftTree(leftTree)).detachRightSubtree( ) = aTree.getRightSubtree().

A Static Implementation of Complete Trees



tree[i] = node numbered i

Invariants:

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.

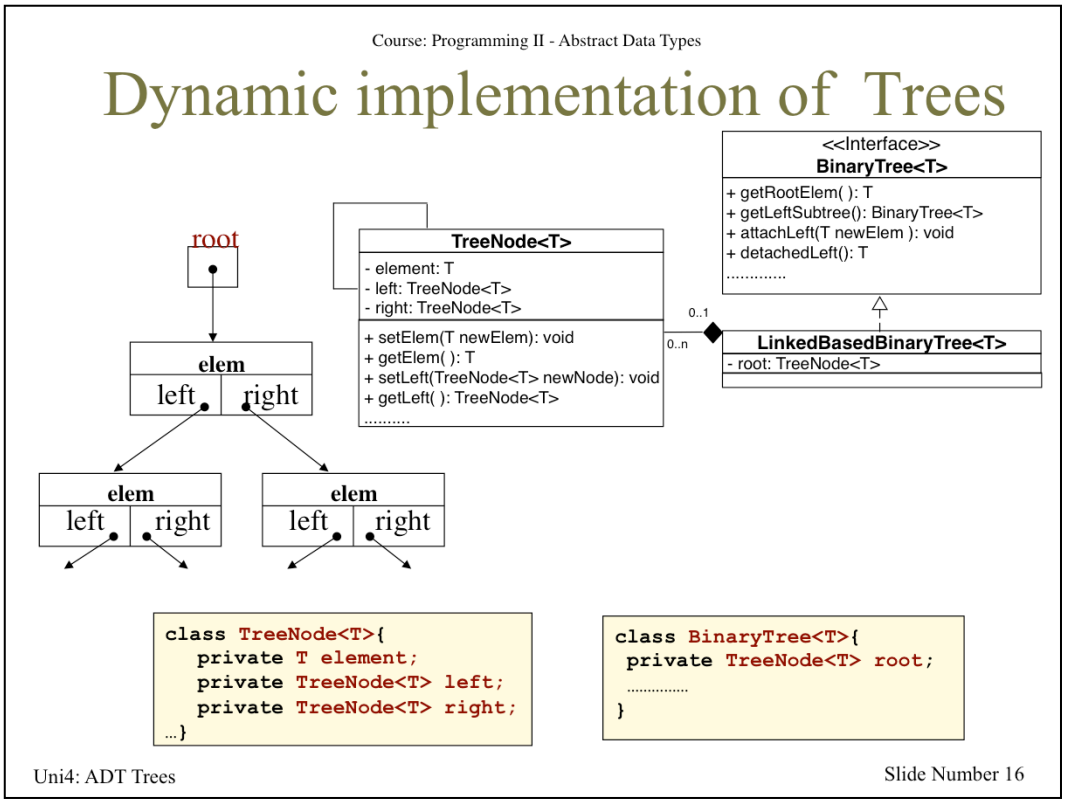
For all the ADTs considered so far in this part of the course, we have provided a definition of the interface for the ADT, which defines the access procedures for the ADT, and various classes that implement this interface as example implementations (e.g., static and dynamic implementation of an ADT). 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. We will consider the access procedures defined in the previous slides as some of the key operation on binary trees. 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 can define specific formulae for accessing the parent (or children) node(s) of a given node. 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 (e.g. int free = 6). The actual links between the nodes are not needed. Instead, links between parent and children nodes can be generated using the formulae given in this slide.

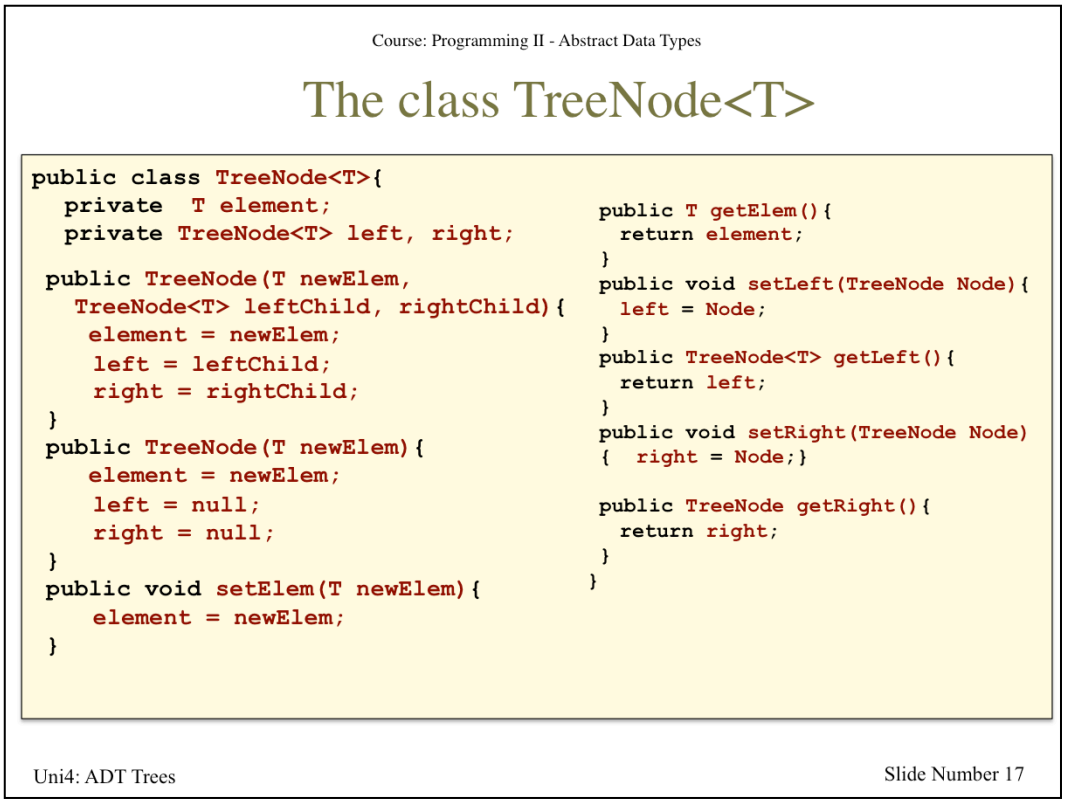
Non complete binary trees could also be implemented using an array, but the array data structure is more complex than the one given here, since in this case it would be necessary to keep explicit reference to the children of a node.



A binary tree can store elements in the tree by using individual nodes, where each node is an object of a binary tree node class. This class contains private instance variables that reference other nodes in the tree. Since these variables are private the class Node 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`.

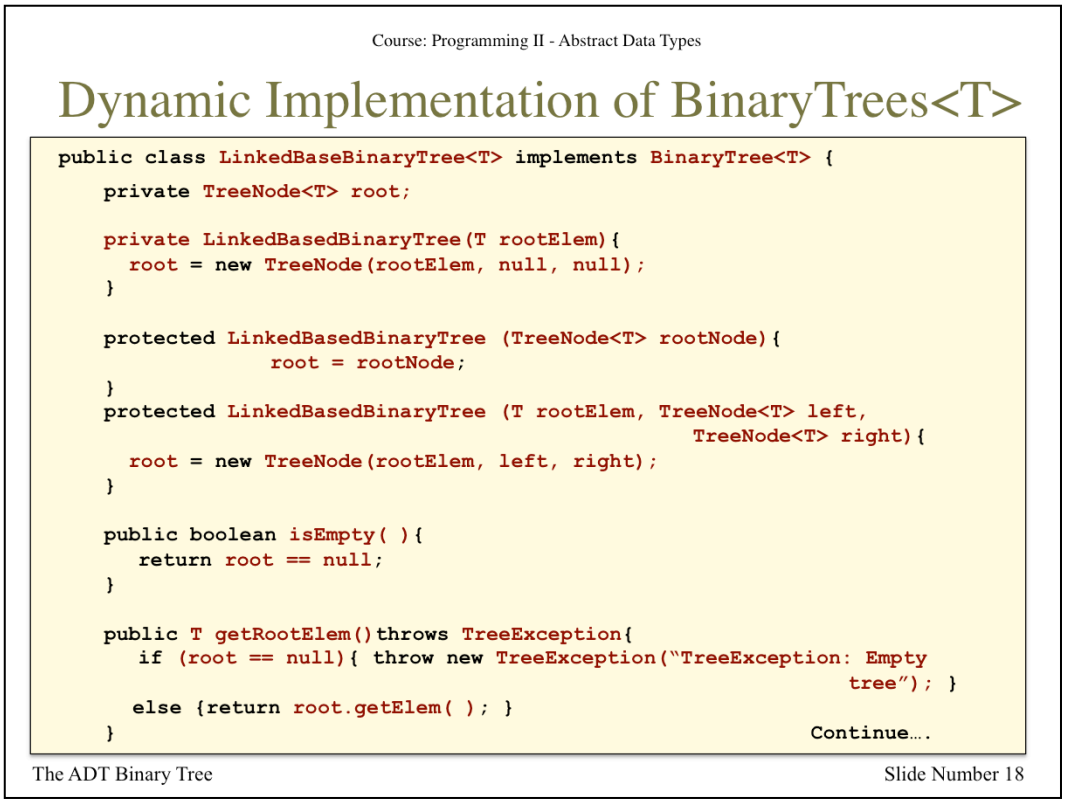
The entire tree is represented as a reference to the `root` node. If the tree is empty, the root is `null`. The default constructor `BinaryTree( )` of the class `BinaryTree<T>` implements the access procedure `createEmptyTree( )` given in a previous slide.

In the next slide we see the basic methods of the class `TreeNode` and some example implementations of some of the access procedures for a binary tree, including more specific constructors.



This slide provides the list of the basic methods that a class `TreeNode<T>` should have and different types of constructors you can implement.

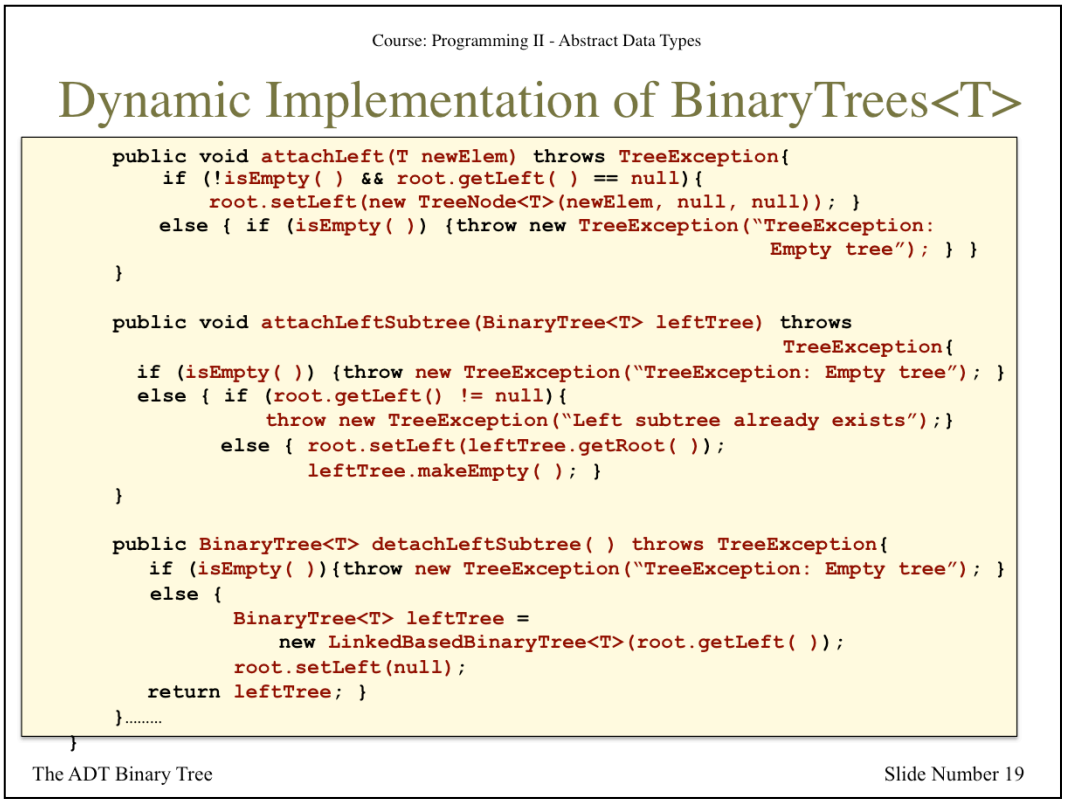
Given this new class for binary tree nodes, how would we implement the access procedures for a binary tree? As shown in the previous slide we would need a class called `LinkedBasedBinaryTree<T>`, which includes just the instance variable "`root`", that refers to the root node of the entire tree. Some example implementations of some of the access procedures for the `LinkedBasedBinaryTree` class are given in the next slide.



This class includes in addition to the default constructor, which does not need to be specified, also other two constructors that respectively create a binary tree with just one node (an example implementation is given at the top of this slide), and create a tree from a reference to a root node and two already constructed trees. If used as an auxiliary constructor, it could be defined as protected in order to prevent client classes from using it directly.

In the implementation of the access procedure "getRootElem" we need to check that the binary tree is not empty. If you look at the axioms given in slide 14, the call of this method on an empty tree should flag an error. An example implementation of TreeException can be the following:

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



This slide continues the implementation of the class `LinkedBasedBinaryTree`. It shows some of the key access procedures. The symmetric operation of the right subtrees are omitted as similar to the one for the right subtree that are illustrated here.

Note that in the implementation of `attachLeftSubtree` we have used a method "`makeEmpty( )`". This can be defined as an auxiliary method of `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. Also the method `getRoot()` retuns a TreeNode object. It is therefore not a public access procedure of the `BinaryTree<K>` interface but an auxiliary procedure of the class `LinkedBasedBinaryTree`.

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.

As for the method "`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. Essentially the implementation of the `getLeftTree` access procedure would be very similar to the implementation of the `detachLeftSubtree` but without setting to null the left reference of the root node.

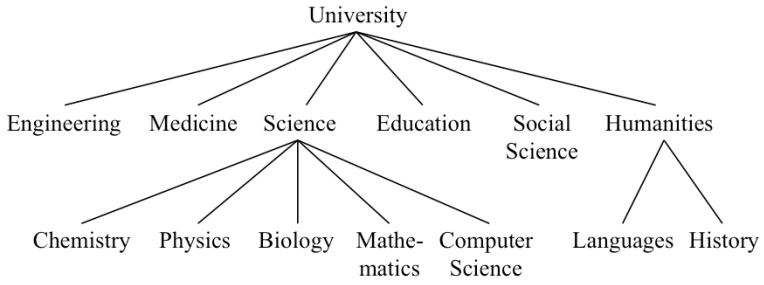
What would the implementation of the access procedure "`getRightTree`" look like? This is left as exercise.

General Trees

A **general tree** is a hierarchical collection of elements, and is a generalization of the binary trees.

Example applications:

Organization tree



A general tree is also a hierarchical collection of elements, organised in level as we have seen for binary trees. But in this case a node may have more than 2 children. General trees are therefore generalizations of binary trees. We will briefly see some example applications of general trees and mention what data structure could be used to implement general trees.

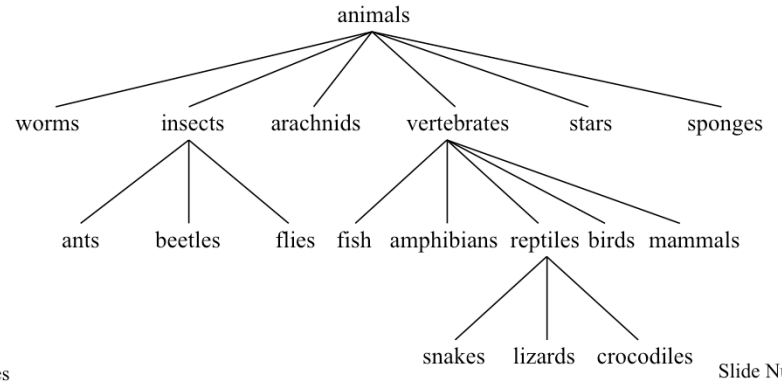
For instance a typical use of general trees is for representing internal hierarchical organizational structure of companies and/or universities (i.e. departments are parts of faculties and different faculties for a university).

General Trees

A **general tree** is a hierarchical collection of elements, and is a generalization of the binary trees.

Example applications:

Taxonomy tree



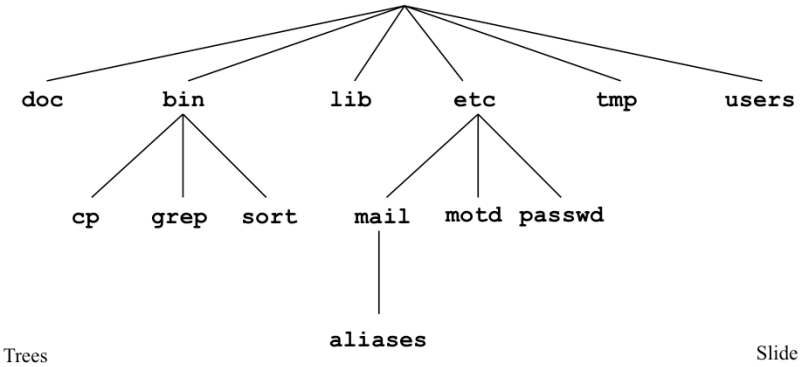
Another example of general trees is for instance that of a taxonomy tree. In this case, for instance, we also need to represent hierarchical structures of different species of animals.

General Trees

A **general tree** is a hierarchical collection of elements, and is a generalization of the binary trees.

Example applications:

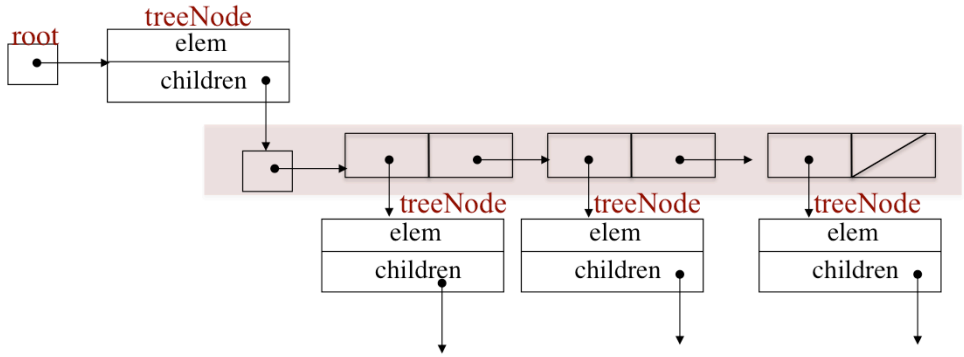
Files hierarchy



In the context of computer science a typical application of general trees is that for instance of a file hierarchy structure of a unix account, where documents (plain files) are organised into folders (directories), and folders can themselves include documents or other folders. In such a tree documents are modeled by the leaves of the tree and folders are modeled by parent nodes.

The question that we need to ask is how to implement a general tree. We will see an example implementation in the next slide in terms of the data structure that can be used to store a general tree. You are supposed to practice implementing yourself such structures since they are based on what we have covered so far in the lectures.

Dynamic Implementation of General Trees




```
class TreeNode<T>{
  private T element;
  private List<TreeNode> children;
…}
```

```
class GeneralTree<T>{
 private TreeNode<T> root;
 ..............
}
```

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. Three different traversal algorithms can be used.
- General trees can be implemented making use of a linked list to store the children of a parent node.