Algorithms and Data Structures 2010-2011

Lesson 3: *trees and visits*

Luciano Bononi
<bononi@cs.unibo.it>
http://www.cs.unibo.it/~bononi/
(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)

International Bologna Master in Bioinformatics

University of Bologna

19/04/2011, Bologna

Outline of the lesson

- **Trees**
  - Basics
  - Rooted trees
  - Binary tress
  - Binary tree ADT
  - Tree traversal (visits)
  - Non-recursive implementation of visits
  - Binary Search Trees (BSTs)

Motivation

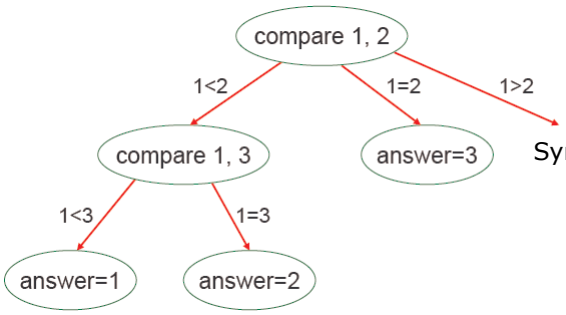
- Let's start with a real world problem:

  given three apparently identical coins, two are of the same weight, while the other is different.

  How is it possible to find out the different one while using only comparisons?

- **Solution:**
  - number the coins, and then compare them

First example: **decision tree**



Symmetric to the case "1<2"

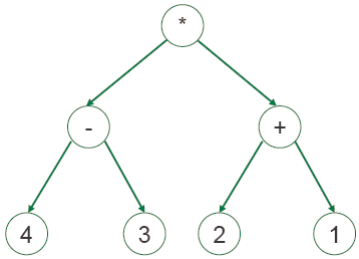
This structure is a tree: a decision tree

Second example: mathematical expressions

- Given the mathematical expression:

  (4-3)*(2+1)

  it can be represented as:



Tree: **formal definition**

- A tree is a set of **nodes** (vertices) connected by **edges** (links) such that there is **exactly one** way to get from any node to any other node
- Example: which of the following are trees?

YES!

No, it is a graph

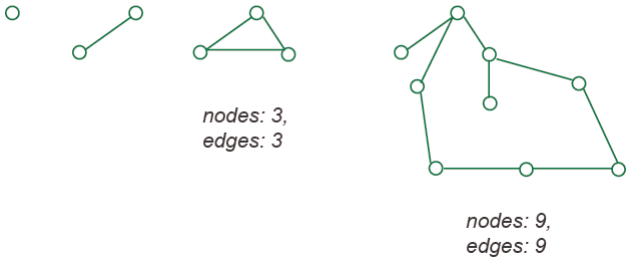
No, it is a forest (i.e. multiple trees)

Theorem

- **THEOREM**:

  every non-empty tree with **n** nodes has exactly **n-1** edges

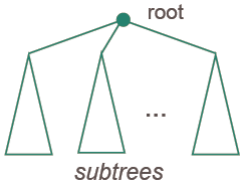
  - How to prove? It is simple, by induction (n ≥1)
  - This theorem can be also used to demonstrate that a given data structures is NOT a tree



Rooted tree

- A tree is called a **rooted tree** if one of its nodes is distinguished as **root**

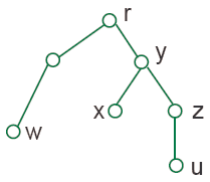
- **This definition can be used in a recursive way:**
  - a rooted tree consists of a root node and a finite set of sub-trees, which are themselves rooted trees



Trees: **terminology**

A little of terminology about trees:

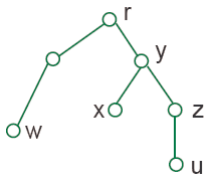
- r is **root**
- y is a **parent** of x (and of z); r is a **parent** of y
- r, y and x are **ancestors** of x
- r and y are **proper ancestors** of x
- x and z are **children** of y
- x, y, z and u are **descendants** of r
- x and z are **siblings**
- all ancestors of u form a **path** from u to the root
  - <u, z, y, r>
- a node without children is called **leaf**
  - in our example: w, x and u are leaves
  - others are called **internal nodes**



Sub-trees

A sub-tree is:

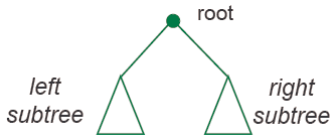
- a node **n** plus all its descendants, n is the root of the sub-tree
- in the example: y is the root of a sub-tree



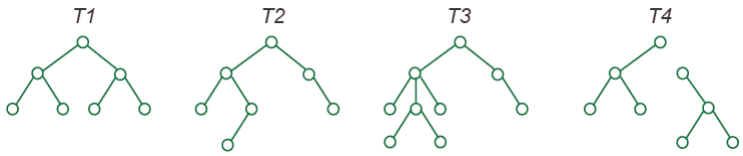
- DEFINITION: an **ordered tree** consists of a root node and a finite sequence of sub-trees, which are themselves ordered trees
- IMPORTANT NOTES:
  - the order is very important!
  - also in this case the definition is recursive

Binary tree

- A **binary tree** is either empty or it consists of a root node and two sub-trees(left and right) which are themselves binary trees



- Which are binary trees?



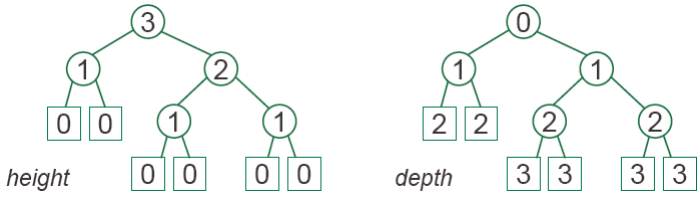
Binary tree

- Are these two binary trees the same?

- Obviously they are **not** the same!
- Remember that binary trees are **ordered** trees!

Height and **deep** in binary trees

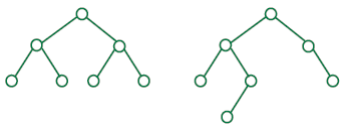
- The **height** of a node **n** in a binary tree is the **number of edges on the longest path** from **n** to an external node
  - the height of a rooted tree is the height of its root
  - the height of an external node is 0
- The **depth** of a node is the **length of the path to the root**
  - the root has depth 0; depths of external nodes can be different



Complete binary trees

- A binary tree whereby if the height is h, and all levels, except possibly level h, are **completely full**. If the bottom level is incomplete, then it has all nodes to **the left side**
- That is the tree has been filled in the level order from left to right

Complete



NOT complete

- Given a **complete** binary tree T with **l** leaves, what is the height **h** of the tree? What is the mathematical function that links the number of leaves in the tree and its height?

Binary tree ADT: some **operations**

Creates a new BinTree	Make(T)
Creates a new Node	NewNode(l, val, r)
True if BinTree is **empty**, false otherwise	Empty(T)
Gets the left child of a node	GetLeftChild(T, x)
Gets the right child of a node	GetRightChild(T, x)
Puts the left child of a node	PutLeftChild(T, x, y)
Puts the right child of a node	PutRightChild(T, x, y)
Puts the root of the BinTree	PutRoot(T, x)

...

Exercise: number of nodes in a tree

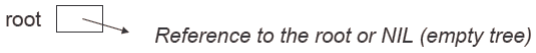
- **EXERCISE**:

  given a binary tree T, count the number of nodes in the tree

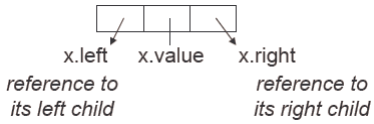
```
function count (T, node)
    if (node == NULL) then
            sum := 0;
    else
            sum := 1 + count(T, GetLeftChild(T, node)) +
                    count(T, GetRightChild(T, node));
    return(sum);
```

Pointer-based implementation of binary tree

- A tree is referenced by its **root**



- Node **x** is composed by three parts



- In this way, all nodes can be organized into a tree via its pointers: it is quite similar to a linked list, but in this case we have not a sequence of nodes
- What is the cost of the operations seen before? **O(1)**
- What is the cost of searching a given element in the bintree?

Pointer-based implementation of binary tree

- **PROBLEM**:

  given a tree T, how is possible to find the parent of a node x?
- **SOLUTION**: in the proposed implementation the solution is to visit all the nodes in the tree
- What is the cost of the GetParent(T, x) operation?
  - **O(n)**, where **n** is the number of nodes of t
- **ALTERNATIVE APPROACH**:
  - add in node **x** a reference (a pointer) to its parent **: O(1)**

Traversing binary trees

- **PROBLEM**:

  how to visit each node exactly once given a binary tree?

- Starting from the root, you can have **three choices**:
  - visiting the root **itself**;
  - visiting the root's **left sub-tree**;
  - visiting the root's **right sub-tree**;

- Three different orders —three main paradigms:
  - **pre-order traversal**
  - **in-order traversal**
  - **post-order traversal**

Binary trees: **pre-order visit**

```
function preorder(T, node)
    if (node <> NULL) then {
        visit(T, node);              // visits the node, i.e. prints the data
        preorder(T, GetLeftChild(T, node));
        preorder(T, GetRightChild(T, node));
    }
```

- **What is the cost of the pre-order traversing?**

- **Is it possible to further reduce the cost of the operation?**

Binary trees: **in-order visit**

```
function inorder(T, node)
    if (node <> NULL) then {
        inorder(T, GetLeftChild(T, node));
        visit(T, node);              // visits the node, i.e. prints the data
        inorder(T, GetRightChild(T, node));
    }
```

- **What is the cost of the in-order traversing?**
- **Is it possible to further reduce the cost of the operation?**

Binary trees: **post-order visit**

```
function postorder(T, node)
    if (node <> NULL) then {
        postorder(T, GetLeftChild(T, node));
        postorder(T, GetRightChild(T, node));
        visit(T, node);              // visits the node, i.e. prints the data
    }
```

- **What is the cost of the post-order traversing?**
- **Is it possible to further reduce the cost of the operation?**

Binary trees: **non recur**sive implementation of visits

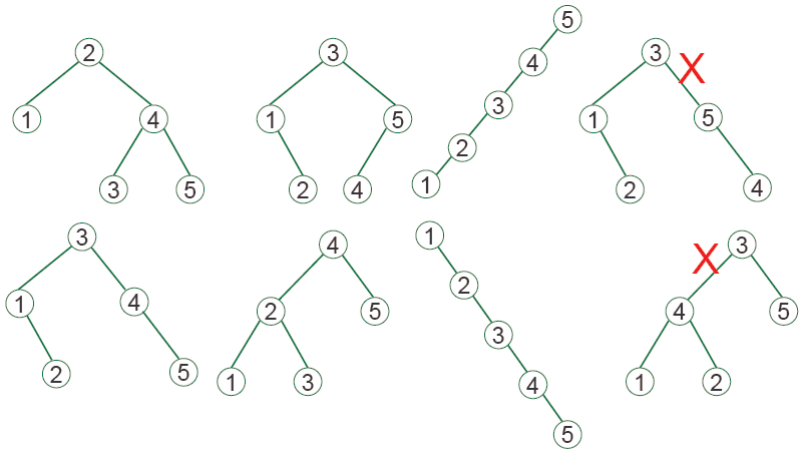
- The preorder(), inorder() and postorder() function seen in the previous slides are **recursive**
- Is it possible to write a **non-recursive** implementation of such functions?
- **EXERCISE**: let's start with preorder()
- **SUGGESTION**: use a stack data structure to simulate the recursion
- **EXERCISE**: inorder() and postorder()

Binary Search Trees (BST)

- **DEFINITION**: for every node **x** in the tree, the value of the entry at **x** is *greater* than the values of all the entries in the left sub-tree of **x**, and *smaller* than the value of all the entries in the right sub-tree of **x**
- For the same sequence, we can have different BSTs
- If we do **in-order traversal** on a BST, we exactly get the **ordered sequence** of all keys!

Binary Search Trees (BST): **examples**

- Given a key set {1, 2, 3, 4, 5}



Binary Search Trees (BST)

PROBLEMS

- what is the cost of:
  - searching a value
  - inserting a new node
  - deleting a node

References

- Part of this material is inspired / taken by the following freely available resources:
  - http://www.cs.rutgers.edu/~vchinni/dsa/
  - http://www.cs.aau.dk/~luhua/courses/ad07/

Algorithms and Data Structures 2008 - 2009

Lesson 3: *trees and visits*

Luciano Bononi
<bononi@cs.unibo.it>
http://www.cs.unibo.it/~bononi/
(slide credits: these slides are a revised version of slides created by Dr. Gabriele D'Angelo)

International Bologna Master in Bioinformatics

University of Bologna

19/04/2011, Bologna