02393 Programming in C++ Module 11
Recursive Data-Structures: Trees

Teacher: Alberto Lluch Lafuente

Sebastian Mödersheim (slides author)

April 25, 2016

Lecture Plan

#	Date	Topic	Chapter *
1	1.2	Introduction	1
2	8.2	Basic C++	1
3	15.2	Data Types	2
4	22.2	Libraries and Interfaces	3
5	29.2		
6	7.3	Classes and Objects	4.1, 4.2 and 9.1, 9.2
7	14.3	Templates	4.1, 11.1
		Påskesferie	
8	4.4	Inheritance	14.3, 14.4, 14.5
9	11.4	Recursive Programming	5
10	18.4	Linked Lists	10.5
11	25.4	Trees	13.1-13.3
12	2.5	Graphs	16
13	9.5	Summary	
	17.5	Exam	

* Recall that the book uses sometimes ad-hoc libraries that are slightly different with respect to the standard libraries (e.g. strings and vectors).

Summary

- Recursive programming... when?
  - ★ Problem is recursively/inductively defined (e.g. Fibonacci);
  - ★ Problem on a recursive data structure (e.g. lists, sets, trees);
- Recursive data structures:
  - ★ Linked lists: single- and doubly-linked lists;
  - ★ Sets, multi-sets;
  - ★ (Binary) trees;
- Abstract Data Type vs Concrete Data Structures:
  - ★ Abstract Data Type: a new type being specified, e.g. a class `Set`;
  - ★ Concrete Data Structure: how we implement it, e.g. `Set` implemented with linked lists... or binary trees... or...;

Today

- Class `Tree` of (ordered) binary trees
  - ★ Inserting an element;
  - ★ Finding an element;
  - ★ Size of a tree;
  - ★ Height of a tree;
  - ★ Traversing the tree (enumerating all elements) in different orders;
- Class `Set` implemented with binary trees

Trees

What is a binary tree?

- A binary tree can be empty;
- A binary tree can be formed by a node and two sub-trees;

Some notation:

- The topmost node of a tree is called the *root*; nodes at the bottom (with empty sub-trees) are called *leafs*, all other nodes are called *internal* nodes;
- The *size* of a tree is the number nodes it has, the *height* is the maximal distance from the root to a leaf.

What is an ordered binary tree or (search tree)?

- An empty binary tree is a search tree;
- A non empty binary tree is a search tree if:
  - ★ all nodes in the left sub-tree are smaller than the root;
  - ★ all nodes in the right sub-tree are greater than the root;
  - ★ both sub-trees are search trees;

Such trees provide efficient insertion/deletion/search methods.

From nodes in a list to nodes in a tree

Possible implementation of tree nodes

```
struct Node{
  int content;
  Node *left;
  Node *right;
}
```

A tree could be then just a pointer to a `Node`, as we did for lists.

Class of Trees

Another possible recursive definition of trees

```
class Tree{
public:
...
private:
  bool empty;
  int content;
  Tree *left;
  Tree *right;
}
```

Here a flag `empty` is used to denote empty trees. Every tree (node) will be a class. We will use this representation in the examples.

Class of Trees: Methods

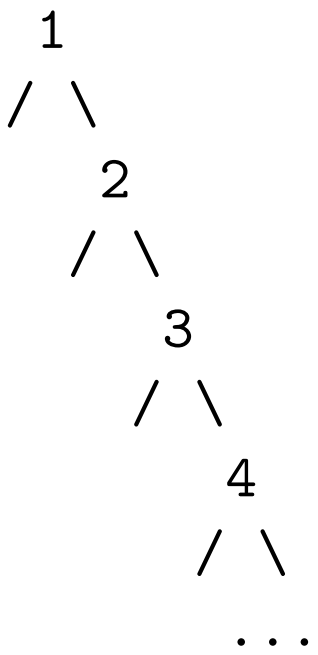
Most methods we need to implement can be implemented using recursion.
Consider, for example, the size of a tree. A recursive formulation could be based on the idea that:

- the size of an empty tree is 0;
- the size of a non empty tree is 1 (for the root node) plus the size of its sub-trees

All other methods that we will see can exploit the recursive structure of trees.
Some of them are not easy to implement without recursion.

Problem with simple binary trees

Inserting 1, 2, 3, 4...

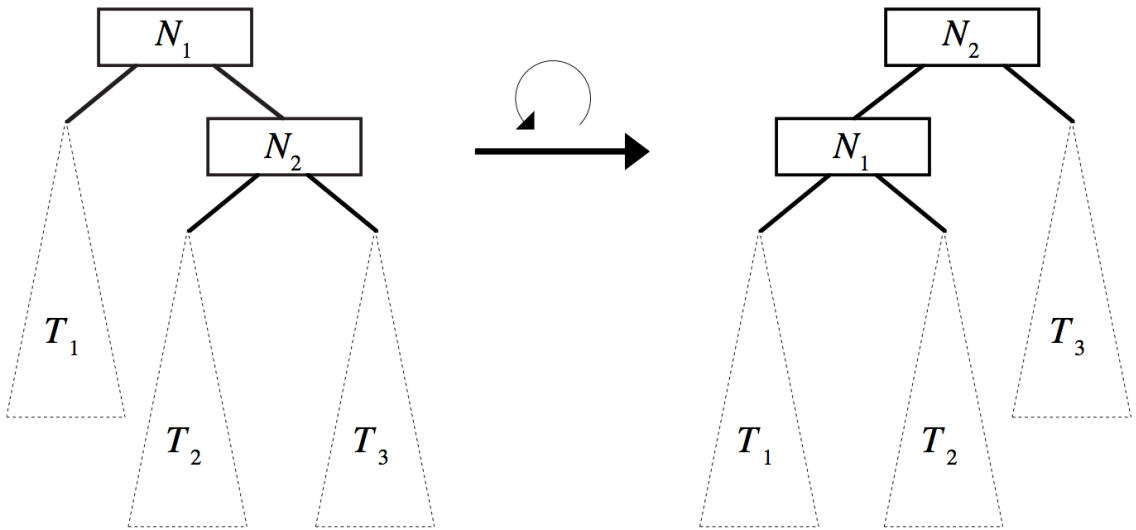


... the tree degenerates into a list :(

Balanced trees

- A family of solutions: AVL (book), B-trees, Red-Black tres, etc.
- Main idea: make the height of the tree be $O(\log n)$.

Main mechanism to re-balance: rotations



Idea: inserting/deleting nodes may unbalance the trees. To re-balance, rotations can be applied.

Re-balancing requires to re-arrange some few links in the tree (a bit like when reversing a linked list).

Complexity discussion

Consider the implementations of sets that we have seen so far and the complexity of these operations:

	(0)	(1)	(2)	(3)
`insert`				
`find`				

(0) Using arrays.

(1) Using Linked-lists.

(2) Using simple binary search trees.

(3) Using balanced binary search trees.