02393 Programming in C++

Module 12: Trees

Alceste Scalas <alcsc@dtu.dk>, **Giovanni Meroni** <giom@dtu.dk>

Slides based on previous versions by Andrea Vandin, Alberto Lluch Lafuente, Sebastian Mödersheim

22 November 2022

Course plan

Module no.	Date	Topic	Book chapter*
0 and 1	30.08	Welcome & C++ Overview	1
2	06.09	Basic C++ and Data Types	2.1, 2.3 - 2.5, 11.1, 11.3
3	13.09	Enumerations and Structures & *LAB DAY*	1.5
4	20.09	Memory Management	12.1, 11.2, 11.3
5	27.09	Libraries and Interfaces	2.2, 2.6 - 2.8, 3.1 - 3.3, 4.1 - 4.3
6	04.10	Classes and Objects	5.1, 6.1, 11.2, 12.1, 12.4, 12.7
7	11.10	Templates	5.1, 14.2
		Autumn break	
8	25.10	Inheritance	19.1 - 19.3
9	01.11	*LAB DAY*	*Previous exams*
10	08.11	Recursive Programming	7
11	15.11	Linked Lists	12.2, 12.3
12	22.11	Trees	16.1 - 16.4
13	29.11	Conclusion & *LAB DAY*	*Exam preparation*
	22.12	**Exam (held physically, all aids allowed)**	

* Recall that the book uses some ad-hoc libraries (e.g., for vectors). We will use standard libraries

Outline

Recap: recursive programming and ADTs

Trees

- Using trees to represent expressions
- The arithmetic syntax tree
- Live coding

Lab

Recap: recursive programming, abstract data types

- Recursive programming is useful when dealing with
  - **recursively-defined problems** (e.g. computing the factorial of a number)
  - **recursive data structures**
- Examples of recursive data structures:
  - Linked lists — singly-linked and doubly-linked
  - Sets, multi-sets
  - **Trees** (today!)
- Always keep in mind the difference between **specification** vs. **implementation**!
  - **Abstract Data Type**: a type being specified
    - e.g. a class `MyVector`
  - **Concrete Data Structure**: how we implement a data type
    - e.g. how the `MyVector` class internally stores its data, by using arrays, or linked lists, or. . .

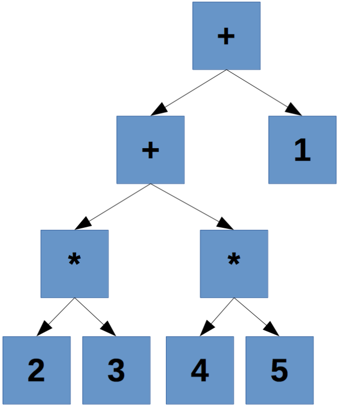
Using trees to represent expressions

How can we represent the following expression? $\Big((2 * 3) + (4 * 5)\Big) + 1$

Using trees to represent expressions

How can we represent the following expression? $\Big((2 * 3) + (4 * 5)\Big) + 1$

We can use a **syntax tree**!

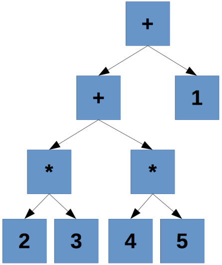


Using trees to represent expressions (cont'd)

We develop a class `ArithmeticSyntaxTree` to represent arithmetic expressions

To represent an expression like $\big((2*3)+(4*5)\big)+1$ we need:

- **constants** (i.e., numbers)
- the **sum** of two arithmetic expressions
- the **product** of two arithmetic expressions



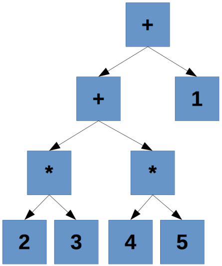
It is a **binary tree**: each node is either **internal** (with 2 children), or is a **leaf**

Using trees to represent expressions (cont'd)

We develop a class `ArithmeticSyntaxTree` to represent arithmetic expressions

To represent an expression like $\Big((2 * 3) + (4 * 5)\Big) + 1$ we need:

- **constants** (i.e., numbers)
- the **sum** of two arithmetic expressions
- the **product** of two arithmetic expressions



It is a **binary tree**: each node is either **internal** (with 2 children), or is a **leaf**

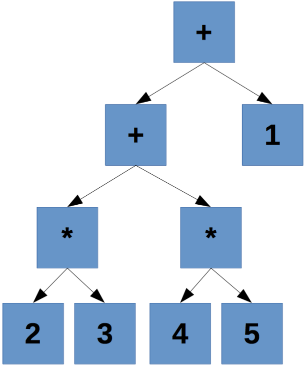
Today we address:

- the **size** of a tree
- the **height** of a tree
- the **number of leaves** of a tree
- **traversing a tree** in different ways

The arithmetic syntax tree

An **arithmetic syntax tree** is made of two kind of **nodes**:

- **internal nodes** with two descendant trees (representing **operators**)
- **leaf nodes** without descendant trees (representing **constants**)



Some more terminology:

- **root node**: the topmost node of a tree
- **size of a tree**: the number of nodes in a tree
- **height of a tree**: length of the longest path from the root to a leaf

Representing a tree using `structs`

Possible implementation of the **arithmetic syntax tree nodes**

```
enum NodeType { Const, Add, Mult };

struct Node {
    NodeType type;
    int value;      // Only used if type == Const
    Node *left;     // Only used if type != Const
    Node *right;    // Only used if type != Const
}
```

A tree is then just a pointer to a `Node`, i.e. `Node*` (as we did for lists)

Representing a tree using classes

Another possible (recursive) definition of **arithmetic syntax tree nodes**

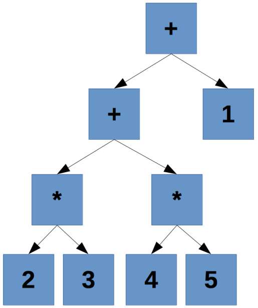
```
enum NodeType { Const, Add, Mult };

class Tree {
public:
  // Some methods...
private:
  NodeType type;
  int value;      // Only used if type == Const
  Tree *left;     // Only used if type != Const
  Tree *right;    // Only used if type != Const
}
```

We will use this representation in the examples

Methods of the `Tree` class

Most methods we need can be **easily implemented using recursion**



Consider, e.g., the **size** **(number of nodes) of a tree**. A recursive formulation:

- **size** of a leaf node $= 1$
- **size** of an internal node $= 1 +$ **size**s of its descendant trees

Non-recursive implementations are possible, but more complicated!

Live coding

Live coding

Implementation of `ArithmeticSyntaxTree`

Lab

Today's lab begins now. Tasks:

- make sure C++ works on your computer, request help if it doesn't
- begin working on **Assignment 11**
- ask questions if something is unclear (including previous assignments)