Deseño de Linguaxes de Programación

Work report for languages C, Java, Ruby and OCaml

Rodríguez Arias, Alejandro
alejandro.rodriguez.arias@udc.es

Bouzas Quiroga, Jacobo
jacobo.bouzas.quiroga@udc.es

Wednesday 4th October, 2017

Contents

1 Introduction

This document is meant to display the work made on implementing the same data structure, a binary search tree, and associated algorithms in an array of assorted programming languages, following a reference Pascal implementation.

2 Language: C

Language Version	GNU C90
Compiler	GCC 5.4.0
Operating System	Ubuntu 16.04.2 LTS 64 bits

C is an imperative procedural programming language that supports structured programming and recursion. C has a weak and static typing and provides a low-level access to memory, allowing us to make a dinamic and manual memory management. C gives us a lot of control flow tools for our problem:

- Executing a set of statements only if some condition is met (`if-else`).
- Executing a set of statements zero or more times, until some condition is met (`while(condition)`).
- Allows a variable to be tested for equality against a list of values in order to choose an execution branch (`switch-case`).
- Executing a set of distant statements, after which the flow of control usually returns (subroutines).

The C library `stdlib` provides us functions to make a manual memory management of the dinamic memory. In our problem we use two of that functions:

- The `malloc(size)` function allocates 'size' bytes and returns a pointer to the allocated memory.
- The `free` function frees the memory space pointed by the pointer.

For the data structure we use `struct`. A user defined data type available in C that allows to combine data items of different kinds. Structures are used to represent a record.

Memory Management

The main differences when it comes to adapt Pascal code to C code are Pascal abstractions in memory management and reference parameters, since C has a lower level memory management. To inicialize a pointer in Pascal is enough to call the `new` function with the pointer as a parameter, but in order to initialize a pointer in C setting the pointer to the returned value of the `malloc` function is needed. The `malloc` function requires the size of the pointer in bytes to allocate the memory.

Differences within reference parameters

Pascal has a `VAR` keyword to pass arguments by reference but in C language every argument is passed by value, so passing a pointer to the memory address of the argument is needed in order to modify it.

Unit in C

C needs two files to make a library: a code file (.c) and a header file (.h). The header file provides a program with library functions. Pascal has the header and source code in the same file.

3 Language: Java

4 Language: Ruby

Language Version	Ruby 2.4.2
Interpreter	IRB (Interactive Ruby Shell) 0.9.6
Operating System	MS Windows 10 Education 64 bits

Ruby is an scripting, imperative, reflective, object-oriented programming language with strong and dynamic typing. Dynamic memory management is automatic. Ruby give us a lot of control flow tools for our problem:

- Executing a set of statements only if some condition is met (`if-else`).
- Executing a set of statements zero or more times, until some condition is met (`while(condition)`).
- Allows a variable to be tested for equality against a list of values in order to choose an execution branch (`switch-case`).
- Executing a set of distant statements, after which the flow of control usually returns (subroutines).

Memory Management

Ruby dynamic memory management is done with an automatic garbage colector that frees allocated memory when it's unused. For the data structure we use a constant `Struct` with the necessary fields for our problem. All structures on our code are created by that constant struct.

In Ruby, type data declarations aren't done in compiling time: they are assigned during run time so we don't need to create any type in order to solve our problem (excluding `tNodeT struct`).

Differences within reference parameters

Pascal has a `VAR` keyword to pass arguments by reference but in Ruby 'pass by object reference' is used:

- Inside the function, any of the object's members can have new values assigned to them and these changes will persist after the function returns.
- Inside the function, assigning a whole new object to the variable causes the variable to stop referencing the old object. But after the function returns, the original variable will still reference the old object.

Therefore Ruby functions return the modified input parameter so setting the input argument on the new function in order to update it correctly is needed. We have to do it that way because when the input argument is an empty tree we need to reassign it to a new node. This logic is maintained in the whole code.

Differences within the recursive remove function

In the recursive remove function a nested function is declared, but Ruby doesn't allow nested function so it can't use the local variables of the function that contains it. We have to pass a variable called aux as an input parameter. Besides a reference to the node is passed as an input parameter in order to avoid losing the original node reference, since aux is being changed to always be the father of the leaf node and so the value of its child (leaf node) is changed into `null`.

Improvement

Ruby's module is a class that can't be instaced or have instance methods which complicates the management of the nodes. If we use a class instantiable for the management of nodes it would be easy to modify the values with getters and setters, simplifying the program logic.

5 Language: OCaml

Language Version	OCaml 4.02.3
Interpreter	The OCaml toplevel, version 4.02.3
Operating System	Debian GNU/Linux 9

Objective Caml, developed by INRIA, is the main implementation of the Caml programming language and it's a multiparadigm language which allows for object orientation, functional and imperative constructs while featuring lazy evaluation and strong static inferred typing. It can be both compiled and interpreted. Caml is, in turn, a member of the ML family of programming languages.

Some prominent features of OCaml include a static type system, type inference, parametric polymorphism, tail recursion, pattern matching, first class lexical closures, functors, exception handling, and automatic garbage collection and polish notation on function application. The provided compiler is optimized for execution speed, making it able to compete with lower level languages in terms of efficiency of the resulting code.

Memory Management

The OCaml language has fully automatic memory management and garbage collection. On implementing the binary search tree, mutable types where used, specifically the `ref` type, which is the standard way to handle object references in OCaml. `ref`s are similar to Pascal pointers but don't allow for `null` values. `ref`s are overwritten with the `:=` operator and accessed through the `!` operator. A tree type was defined as a tuple containing a value and two references to similar trees in the following manner:

```
type int abb = Null | Node of (int * int abb ref * int abb ref);;
```

This type can express the values `Null`, in the case of an empty tree, and `Node (int, ref, ref)` for an existing tree.

Differences with Pascal implementation

Modifications to mutable OCaml types are permanent. Because of this, our 'pointers' behave in a way similar to Pascal's `var` arguments. Directing a reference to a different object from inside a function will be reflected in the caller's scope so function headers can be adapted from Pascal practically verbatim.

In the imperative version of the deletion function a very slight change was introduced to ease the readability of the resulting code. Where the number of subtrees of the current node is calculated in the original code in order to choose an execution branch with a `switch-case` construct, an OCaml pattern-matching was placed instead, avoiding the need to calculate this number explicitly. Pattern-matching was also used in small auxiliary functions from the specification as it was deemed the most natural way to write them in OCaml.

Improvement

The recursive nature of binary search trees make them an ideal candidate for a purely functional, immutable implementation; favored in the OCaml ecosystem. This would allow for cleaner code, as OCaml syntax leans heavily on the functional end of things through pattern-matching and similar mechanism, and also to take advantage of the tail-recursion capabilities provided by the compiler.

A very simple change in the type definition is enough to make the developed module polymorphic, by substituting `int` with `'a` and adding the additional parameter where needed