**COURSE: CSC 316 FUNCTIONAL PROGRAMMING**

**PROJECT: 1.3 MULTI-TYPE TREE NODE EDITING**

Preface: Courier [New] font text for executable commands.

Bold and italicized font are for function names.

**PART 2: PROJECT DEVELOPMENT REPORT**

**Analysis Process**

The project requires the creation of a Binary tree that takes values of possibly different types.

The Nodes of the tree are to be distinct and labeled so that it helps identify the type of operations that are carried out on the nodes i.e. each node of the same type have operations that can be carried out specifically on them.

The nodes though of possibly different types should have a relationship among themselves which should be pre-defined and stored and can be edited.

Though we are talking about a binary tree, it can happen that a node has more than two children, this is achieve by using container nodes (the right child of a node become a container node if its parent is to have other children ) which are roots to sub trees of nodes related to their parents.

Generic operations should be created that invoke specific operations that effects it taking into consideration all constraints for that particular situation.

There should be a table that holds all the operations that can be carried out on the Nodes and or tree both generic and node specific functions.

The tables should be separated into sub tables one that maps a function to a type and the other sub tables are for the different types of Node holding all the operations that can be carried out on that particular type.

**PROBLEMS AND APPROACHES TO THEM:**

* Our first approach of the project was that the tree created should be a multi-way tree since a node had the possibility of having more than two children this was because we had not clearly grabbed the concept of the container node(the special node) before.
* Another problem was when performing an operation on a tree we did not know which node we are dealing with since operations are type specific therefore we came up with the idea of creating functions that determines the specific type of the node and hence call the specific operation that can handle the node type. for Example deleting a node: though there is a delete function which calls delete course, ***delete\_lecturer***... depending on the type of the node hence we created functions ***get\_node\_label*** and ***get\_label*** where ***get\_label*** is called when you are at the node itself but ***get\_node\_label*** gets some arbitrary values of a node then it searches for the node and so then returns the type.
* Another problem was providing a table that holds the functions in order to give the user the possibility to modify the functions

We then had many different approaches in handling the table including

Firstly using a list which was not efficient at all where the time complexity of getting an element **O(k)** where k > 1 and is the position of the function on the list.

Secondly we then came up with a tree where the complexity **O(k)** was **nlog(n).**

Lastly we came up with the better ones being a Map and a hashtable which had a complexity **O(k) = constant** but we agreed on using a hash table.

* Another problem was we created one hashtable where we stored all the operations we were dealing with irrespective of the type of node it was to handle hence posing a difficulty in knowing and selecting a procedure for a specific type, so we had to create sub hashtable for the each specific node type that held the operations for permissible on the node type.

**Design**

* We started by creating the tree node type (using ocaml's variant), then the functions to work with the tree, and finally the table to hold the functions. We had to consider the fact that nodes of different types are treated differently (and we used polymorphism to deal with this, where once we call a function it first checks on the type of the node and calls the corresponding function for that type) and also the fact that the user can modify the functions that we provided. We agreed on using hash tables to save them, enabling the user to modify them, but for that we had to create another hashtable of type functions (the variant ***'a type\_func*** ).
* We also decided in order to make some of our types more Dynamic, to use polymorphic variants to deal with that but it was difficult as the idea came up late and trying to modify our whole code would have been such a draw back since the dateline was fast approaching.
* We had to learn lots of ocaml's concepts to implement our project (from variable to module).
* We can say we went deeply into ocaml. We made lots of mistake (they help us to know what to do while facing certain situation) so now it is quite easy for us to turn our algorithms into ocaml's code.

**Implementation process**

Most of the implementation we used the Ocaml utop to do most of our code testing and running. We also used text editors for our code typing and editing (and also maintenance). So the code from our paper drafts were more concretely turned into the program using the above mentioned tools

Our main approach to the implementation was the divide and conquer procedure. Breaking our project into simple sub units and implementing the subunits until finally we arrived at the solution proper.

We first created the multi-type binary tree which we tested with the data types in ocaml (int, float and string) and when we were sure of its working,

We proceeded to creating our own types and the main type (variant that held the types (***diffType***)).

Next we created the basic functions that could be applied on the tree but at first we were not concerned with the constraints or managing the functions according to the different types the node held but rather the functions were more general like insert, delete etc.

Furthermore we then went along to try and implement the container concept in the specification which we did, by extending the right node to serve as one when we need to use a container.

We then went ahead to try and make our functions more specific and ensuring that when we call the general purpose functions, they should call the appropriate function to effect the change for the particular type of node.

Another thing was that we decided and implemented everything inside a module as a good and easy way to manipulate the bulk program.

Next we implemented the Hashtable to solve the problem of the table, for each type there is a table that hold all the possible operations that can be performed on that particular type of node and we also created a main hashtable to hold the other tables to ease interfacing(accessing) the other tables.

Lastly we had to modify some of our functions to be able to call needed functions from the hashtable rather than the module since there is the possibility that the user can modify the functions in the hashtable

To crown our project we created a means (user interface) by which any user can easily manipulate the project.

**Part 3: USE OF PROGRAM.**

Firstly copy the source code to a preferred location (say desktop),

Open you terminal,

Change the directory to the directory where the file is found,

Launch the ocaml utop (works for all Operating systems)

1. Type #use "MultTypeTree.ml" ;;
2. Next type open MultTypeTree ;;
3. Next type init ();;
4. Lastly type let root = run Leaf (\*Note root is the name of the tree you want to create so you can change the root to something pleasant, and run is a function that is responsible for the interfacing\*)

You can now follow the guide to do what you want to do.

Last Note is that after you have gone through what you want to do the next time

Type let root = run root ;; to effect the change on that same tree.

**ADMINISTRATIVE REPORT**

**Member participation**

We tried to develop the algorithm together as a group but individually we had some specific tasks we worked on.

Asnel Christian was the main coder for our project and he worked on the module, edit functions and the hashtables.

Kefeh Collins worked on the documentation, on the insert functions and part of the edit functions.

Chawa Elcid worked on the relation mapping and function to get relations with different nodes.

Happi Yvan worked on the tree creation and functions to extract information from the tree.