Final project report

Daniela Alvarado Pereda

A01329233

05/12/2017

Estructura de datos

ITESM Campus Puebla

Introduction

For the final project for the Data Structures course, a program that implements an AVL tree must be programmed using C. The purpose of the program is to help someone who can benefit from this type of system to organize and manage certain entities.

The high school robotics team from ITESM Campus Puebla has several tool containers (chests, cabinets, boxes), each tool has a label with a number that depends on when the tool was bought. The team could use this program to organize and manage data regarding their tools.

Taking this into account, the entities or nodes that the program will manage will be tools, which have a unique numerical label or “key”, a type (screwdriver, saw, driller, etc.) and a brand. The following section details the implementation of the described system.

Process of implementation

Structures

The system was implemented in C, first, two structs were created, a Node and a Queue. In addition to the aforementioned attributes of a node, an integer called “enabled” was added, because an entity can be disabled by the user. The Queue is an auxiliary structure that will be used to store pointers to nodes so they can be sorted in different configurations.

```
5  struct Node{
6      int key;
7      char* type;
8      char* brand;
9      int enabled;
10     struct Node* left;
11     struct Node* right;
12 };
13
14 struct Queue{
15     struct Node** array;
16     int back;
17 };
```

Some functions were programmed for the menu interface of the system, this functions take care of tasks such as periodically printing menus, clearing the screen, and controlling the flow of execution of the program.

Basic AVL tree implementation

Next, the basic functions to manage trees were created, these functions execute tasks such as inserting a node, creating a node and finding a node’s father. After the tree implementation was functional, the specific functions for AVL trees were implemented, like one to check if the tree is balanced and another one to balance it. To support the AVL tree implementation, a helper function was created in order to determine the tree’s height.

Node management

To begin with the implementation of managements of nodes, a “checkNode” function was created to determine the state of a node, it returns 1 if the node exists and is enabled, 0 if it doesn’t exist and -1 if it exists but is disabled.

Next, some auxiliary functions were created to perform tasks such as print a node, receive string and integer inputs from the console and return a pointer to a node given its key.

After that, the main functions to manage an individual node were created. The function “delete” sets a node “enabled” attribute to false (0). The “edit” function lets the user edit the node’s type and brand. The “recover” function sets a node “enabled” attribute to true (1).

Tree printing

The tree can be printed in six different ways: ascending and descending orders based on either the key, type or brand of the node. To print the nodes according to their keys, recursive inOrder and reverse inOrder algorithms were used.

To print the tree based on any of the other fields, every element of the tree was inserted into a queue and sorted alphabetically using a bubble sort algorithm. Helper functions were created in order to fill and print the queue.

Search

The elements in the tree can be searched by their key or looking for a string in any of their other fields, the match can be exact or partial. To look for an exact key match a search algorithm that uses the advantages of the binary tree was implemented, it compares the key of the current node to the key to match and it moves to the right or left child of the node according to the result.

Variations of the inOrder and reverse inOrder traversals were used to find keys lesser or equal, or greater or equal than a number entered by the user.

For the string search, the tree was traversed to look for partial or exact matches. The C functions from string.h “strcmp” and “strstr” were very useful to make the implementation of these tasks easier.

Saving and loading files

To manage files, auxiliary functions were created to open files, read string and integer input from a file and delete a tree.

In order to save a file, the height of the tree had to be obtained to write a level traversal to a file. After each element is written to the file, a “-1” is printed on the last line, to indicate that there are no more entities left. This choice seemed the best since because of the nature of the elements, there cannot be a negative key value for a node.

To load a file, the current tree has to be deleted first, assuring that all of its dynamically allocated elements are freed. Then, a loop reads each entity until it finds the “-1” that indicates the end of the file.

Difficulties

Output

The first significant difficulty presented itself when trying to output the list of elements in the tree using bar separators (“+--“). This was difficult because some of the functions that listed nodes were iterative while others were recursive, because of this, it was difficult to find a unifying way to do it.

In the end, an extra parameter was added to the “printNode” function, that would determine if the upper bar was printed or not. This way, it was printed for the first element of the list but not for the rest. The first element was identified in a different way depending if the function was recursive or iterative.

Reading and writing files

Reading and writing files was initially difficult because the newline character wouldn’t create a new line on the saved file. The character had to be “\r\n” instead of “\n” in order for a newline to be written in the backup file.

This caused issues with reading the file too, because an extra character had to be disposed after each line, and this caused conflict at the end of the file. In the end, it was discovered that even if a new line wasn’t created, the newline character was still there.

Concatenating strings

Since every string read by the program is allocated dynamically, there were conflicts when trying to use strcat function to concatenate filenames with the extension “.txt”. Because of this, it was decided to use files without extension to save backups for the information generated during the execution of the program.

Conclusion

Even though there were difficult aspects regarding the implementation of the system, it was also made easy because many of the functions were reused from previous assignments and activities. This remarks the importance of dividing code into functions and make them modular and reusable.

The different structures studied during the course made this project possible, effective and more efficient. Even though theoretical knowledge is essential, it is also important to recognize the practical applications that it has, and how it can improve processes and make people's lives easier.

References and used resources

C++ reference. Retrieved from: http://en.cppreference.com/w/

cplusplus.com. Retrieved from: http://www.cplusplus.com/

C Tutorial. Retrieved from: https://www.tutorialspoint.com/cprogramming/