Hunter Negron

Main

This section of the code functions the same as the main method in a C program. It is both the entrance and exit point for the program. All of the test cases are located here. I tried to make the comments as big and clear is possible since some of it can be dense. It starts by making a new tree, then it adds values to it, then prints it out. Various values are then tested for containment in the tree. Two of the test cases test values that should be in the tree, and two of the test cases test values that should not be in the tree. Following every call to the “contains” method, is a compact if-statement that merely displays the value and a message that says whether it was found or not. Following these, there is a series of four tests for the “delete” method. They use the same exact numbers as those in the test cases for containment. After each removal, the tree is printed out. Deleting a value not found in the tree does not affect the tree at all. Finally, the main section executes the program.

New Tree

NewTree starts by saving all of the arguments to the stack. Then, 3 words are allocated because that is the canonical size for a node (value, left child, right child). The value parameter is then placed as the value for this new node. One more word is allocated for the binary tree itself, that will hold the address of the root node. The address for this binary tree is then returned after the stack is adjusted.

AddNode

AddNode starts by saving all of the arguments to the stack. Then, a loop is used to iteratively find the location for where the new value ought to be placed. It traverses to the left or right node depending on its value in relation to existing nodes, and finds the first null space that maintains the structure of the binary tree.

Once a spot is found, 3 words are allocated for the new node and then the value is added to it. It finishes by correctly adjusting the stack.

Contains

Contains starts by saving all of the arguments to the stack. It then uses a loop to find the spot where the value in question ought to be found. If it gets to this spot and the pointer is not null, then the value is found and 1 is returned. If a null pointer is reached, then there must not be a value there so a 0 is returned. The function ends by putting this return value into `$v0` and adjusting the stack.

DeleteNode

DeleteNode takes a tree and a value. After saving the appropriate arguments to the stack, it tests for containment and immediately returns a fail value (0) if not found, if so, it calls DelRec to get a pointer to the updated root node after the value in question is removed and returns a 1 for success. It finishes by appropriately adjusting the stack.

DelRec

DelRec is a recursive function that takes a node and a value. It is only used as a helper function for DeleteNode. It is only called when the value exists inside the tree. It works by considering all possible cases. First, if both children are null, the root is the value in question, so null is returned because there is nothing else to return. In the case that either one of the children are null, merely the non-null child is "pulled up" into the position of the root node, replacing it in memory, by a recursive call that properly adjusts the tree below the node in question This way, it is properly found in the tree. In the case that both children are not null, the largest element smaller than the value in question replaces the root value. In this case, the memory of the former leaf node must be freed.

Possible Optimizations

The number of instructions is reduced in multiple methods by only saving those arguments that will need to be accessed in the future and there is a possibility that they will be overwritten.

Methods have multiple ways of ending, mostly with minute differences. Therefore, the similar parts are put under one label and jumped to when appropriate.

In cases where the number zero is used. The zero register is used directly rather than loading the immediate value zero into a temporary register.

Memory usage is reduced by only using recursive calls when it makes sense to do so. Since every function call uses up stack space, iterative implementations reduce memory using when it will not inconvenience the programming or readability.