Operativsystemer og multiprogrammering
Handin 1

Malte Stær Nissen and
Jacob Daniel Kirstejn Hansen

February 14, 2011

Contents

1 Assignment 1

We have in the following subsections outlined the each of our implemented functions, and their help functions, in order of implementation.

1.1 `out_of_memory()`

This function isn't really necessary for the actual implementation, but we used it to debug our other functions. It simply prints a message, notifying us that we're out of memory.

1.2 `insert(tnode_t** tree, int data)`

We start off creating a private variable called root, so we have a starting point. Thereby saying that we expect the insert function to be called on the root of a tree and not e.g a leaf. We'd also have to use this function, as there isn't any build_tree function. So to build a tree we have to make repeatedly calls to insert.

We have used a recursive strategy, instead of a iterating strategy with a while loop. We have chosen to do so because a binary search tree is recursively defined, therefore our solution is intuitively easier to understand - though it probably isn't the fastest one.

The actual implementation is quite simple: We check if the root is nul, if so we just allocate memory and check if we're out of memory. If we're not out of memory, we insert the data argument in the node at this location and update it's children to null. Notice that this is the only case where we actually insert and update children; When the node is null.

If we reach the second if statement, the tree isn't empty. We check if the data value of the leftchild is less than or equal to the data we want to insert, if so we make a recursive call to insert this time with the left child as argument, thereby we have chosen to go down the left tree.

If we reach the last case, the data of the right child is greater than the data we want to inser. Therefore it's not necessary to make another if-statement, we simply make a recursive call to insert on the right child, thereby going down the right tree.

1.3 `print_inorder(tnode_t* tree)`

In this function we again have a recursive strategy. We always print out the left-most node first, so our first if statement checks if the left child is null, if not we keep making recursive calls, with the left child untill it's null. We then print the left-most node's data. If the right child, that is the brother/sister of the left-most node is make a recursive call to print_inorder with this right child - thereby checking the first if statement again..

1.4 `size(tnode_t* tree)`

We start off checking if the tree is null, if so we return 0 - as there aren't any elements. The only setback of this check is that it is always checked, as we again are using a recursive strategy. In retrospect we should have used an incremental strategy to save work, but we found the recursive implementation easier to read. We basicly do the exact same thing as print_inorder, expect we don't print the data, but increment a counter and make recursive calls - thereby summing each subtree.

1.5 `to_array_help(tnode_t* tree, int* pos, int* array)`

This function is used to help the to_array function by means of recursivly walking through the tree inorder, and incrementing the position variable it has as argument from the to_array function and inserting the data at that position, by having a pointer point to it.

1.6 `*to_array(tnode_t* tree)`

We start by making a call to size and save it in a variable, then allocate memory on the stack for the number of nodes (the call to size) times what an integer demands of memory. We then call to_array_help in order to create the actual array and then return a pointer to the first data element of the array.

2 Assignment 2

We have in this section created a new modulus function to help wrap a double linked list, in order to make it circular. Again the functions are outlined below.

2.1 `mod(int x, int y)`

Our modulus function works the same way as the original implemented modulus function of C, except our function handles negative numbers in a different way. The actual coding is self-explanatory so we won't go into detailt explaining it. What it does is if we call e.g mod(-2,6), we get 4. If we call mod(-1,6) we get 5, if we call mod(0,6) we get 0.

2.2 `tree2list(tnode_t* tree)`

We start off creating a dlist and setting it to null as we have no elements. We then make a container that is a pointer to a pointer to allocate memory. Again we create a variable, count, to store our size of the tree in and allocate memory according to the number of nodes times what an integer requires of memory. But before that we make sure that we have elements by checking if count is greater than 0, if not we simply return dlist, which at this point still is null. If count is greater than 0, we then create a pointer to a data_array by making a call to our to_array function. We then iterate a number of times equal to the number of elements in our tree and create them as dlists and save them in our container. Now that we have all our elements created and we know their positions, we simply iterate through all of them and create pointers to the next and previous one. This trick works only because we have created our own modulus function that wraps around negative numbers aswell. Lastly we set dlist to point at the first element in our circular double linked list and return it.

3 Assignment 3

We have to this assignment created an aditional function called compare, to compare two integers. We did this to check our insert2 function.

3.1 `insert2(tnode_t2 tnode, void* data, int (*comp)(void*, void*))`

This function is really simply now that we've already made the insert function. The only difference is in the comparing because insert2 inserts in the left child tree if the data is only less than, and not greater than as in insert. So instead we just use the comparing function argument. Besides that, it's the exact same implementation as insert.

3.2 compare(void* x, void* y)

This is an example function used to test our insert2 function. It is simply implemented with if-statements so it follows the guidelines of the problem thesis.

4 Tests

We have made tests for all the functions in a main function. The code should be self-explanatory.