Assignment 3 (MacOS version)

Due Tuesday, November 21 at 11:55pm

with an automatic extension to Sunday, November 26 at 11:55pm

This assignment requires you to write two assembly language functions, one to insert employee records into a binary search tree and one to remove employee records from the binary search tree in alphabetical order. You are being given the C versions of these two functions, as well as the C code that uses these functions.

You should perform the following series of steps:

Step 1

If you are reading this document, it means that you have already downloaded and unzipped the file containing the source code and input files. You should have the following files: assignment3.c, bin.s, binary_search_tree.c, binary_search_tree.h, input_file.txt, and makefile.

Compile the provided code (in a shell) by typing "make". This produces an executable file called "assign3". To run the executable, type:

./assign3 input_file.txt output_file.txt

This should create a file named output_file.txt whose first line is:

Abarry, Abdou, 599872805, 194617

and whose last line is:

Zeledon, Rodrigo, 1144067779, 70062

Step 2

Read the file binary_search_tree.h closely. You'll see that it defines a struct type EMPLOYEE, which has an 8-byte (i.e. long int) "id" field, an 8-byte "salary" field, a 100-byte "first" field (a character array for holding the first name), and a 100-byte "last" field (for holding the last name).

Thus, if a register such as %rcx points to an EMPLOYEE struct (which starts with the "id" field), the fields of the EMPLOYEE would be accessed as offsets from %rcx as follows (where you should fill in the offset in each "___" based on writing and running a tiny C program, like my offset.c on Brightspace, that tells you the offsets).

EMPLOYEE (216 bytes)

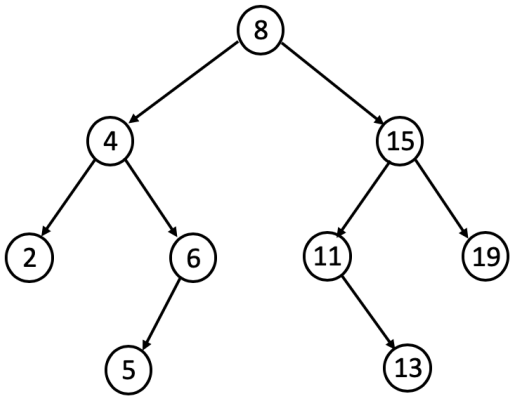
last (100 bytes)	___(%rcx)
first (100 bytes)	___(%rcx)
salary (8 bytes)	___(%rcx)
id (8 bytes)	___(%rcx)

You will also see in binary_search_tree.h the definition of a NODE struct type, which will be used for the binary search tree. A NODE has a "person" field of type EMPLOYEE (above), and two pointer fields, "left" and "right", each of type NODE *. If, for example, %rcx points to a NODE, then the fields of the NODE would be accessed as offsets from %rcx as follows (where you should fill in the offsets in "___").

NODE (232 bytes)

right (8 bytes)	___(%rcx)
left (8 bytes)	___(%rcx)
person (216 bytes)	___(%rcx)

The NODE type will be used to define the nodes of a binary search tree. As you probably recall, a binary search tree is a tree where the label associated with each internal node is greater than any label found in its left subtree (if there is one) and less than any label found in its right subtree (if there is one). For example, the following tree is a binary search tree.



In this assignment, the employee names (not numbers, as shown above) will be the basis for the ordering of the nodes in the tree. Note that the code for this assignment makes no attempt to balance the tree. That

is, it makes no attempt to ensure that the paths from the root to the leaves are roughly the same length (which would lead to a more efficient search since that would minimize the height of the tree).

Step 3

Read binary_search_tree.c very closely. As indicated by the code and comments in the file, the following functions are defined:

- new_node(): This function takes an employee record as a parameter, allocates a new node (of type NODE) by calling malloc(), populates the node with the employee data, and returns a pointer to the node.
- insert_node(): This function take a node containing employee data as a parameter and inserts the node into the binary search tree whose root is pointed to by the global variable "root". The insertion, of course, maintains the tree as a binary search tree. The employee's name is used as the basis for determining where in the tree the node is inserted.
- remove_smallest(): This function removes the smallest (alphabetically) node from the binary search tree. The node removed, of course, is the leftmost node in the tree. The function returns the removed node.
- insert_employee(): This function takes an employee record as a parameter and calls new_node() and insert_node() to insert a new node containing the specified employee data into the binary search tree.
- remove_employee(): This function takes a pointer to an employee record as a parameter. It calls remove_smallest() to remove the smallest (alphabetically) node from the binary search tree, and copies the employee data from that node into the record pointed to by the parameter.

Step 4

Read assignment3.c closely. As the comments and code indicates, the main() function uses the functions in binary_search_tree.c to sort employee data. In particular, main() expects two command-line parameters in argv[]: the name of a file to read unsorted employee data from and the name of a file to write the sorted employee data to. It reads the employee data into an EMPLOYEE array named unsorted_employees. It then sorts the employee records alphabetically by putting each employee record into a node and inserting the node into a binary search tree. Once all the employees are in the binary search tree, the smallest (alphabetically) employee record is repeatedly removed from the tree and placed into the array sorted_employees (in successive positions), until the tree is empty. At this point, the employee records in sorted_employees have been sorted and are then written to the specified output file.

Step 5

Review the calling convention information for macOS that was provided on Brightspace. Be sure to also review the assembly language programs written in class that demonstrate the use of calling conventions.

Step 6

This is the step where you write the assembly code for the insert_node() and remove_smallest() functions (only). You should put your assembly code in the file bin.s that has been provided for you. You should:

1. Comment out the code for insert_node() in binary_search_tree.c (or simply rename the function to another name). Commented-out C code for insert_node() has already been put in bin.s. Your job is to translate it into assembly code. You can insert the assembly code corresponding to a line of C code right after that line of C code, so that commented-out C code and assembly code is interleaved.
2. Once you have finished writing the assembly code for insert_node(), you should debug it before moving on to writing remove_smallest().
3. Repeat step 1 for remove_smallest().

Here are some helpful hints:

- The "root" global variable, which is already defined in binary_search_tree.c, can be accessed in assembly language as "_root(%rip)".
- The strcmp() function, which is called by insert_node(), expects its two parameters to be addresses – in this case, the addresses of character arrays containing employee names. In order to pass these addresses to strcmp(), use the "leaq" instruction to compute each address.
- Make sure, in insert_node(), to save (push) the caller-saved registers that you are using before you call strcmp(), and then restore (pop) them after the call (in reverse order).
- If you are using any callee-saved registers in insert_node() or remove_smallest(), you'll need to save those registers prior to the first time you use them and restore those registers after you're done using them.
- As discussed in class, keep %rsp aligned on 16-byte boundaries by always decrementing %rsp by a multiple of 16. Since the push instruction pushes 8 bytes onto to the stack, always either perform an even number of pushes or, if you have an odd number of pushes, subtract 8 from %rsp after doing a push.

Step 7

When your assembly code in bin.s is working, submit only the bin.s file by uploading it to Brightspace.