Computer Systems Organization
Fall 2024

Programming Assignment 1
Due Monday, October 7 at 11:59pm

In this assignment, you will be building binary search trees and a circular doubly-linked list. Here are the steps of the assignment:

1. Create a file called assignment 1.c . This is the file in which you will be putting all of your code.

2. At the top of the file, put the following #include commands:

```
#include <stdio.h>   // to use printf and scanf
#include <stdlib.h>   // to use malloc
#include <string.h>   // to use strlen, strcmp, and strcpy
```

3. Using typedef, define a struct type, **PERSONNEL_REC**, that contains the following fields:
   - **last_name** and **first_name**, both strings (of type char *).
   - **middle_initial** of type char.
   - **age** and **salary** of type int
   - **id_num** of type long.

4. Write the following functions that each takes two pointers, p1 and p2, to a PERSONNEL_REC (i.e. of type PERSONNEL_REC *) as parameters and returns an int:
   - **compare_id_number()**: returns a positive number if p1's id_num is greater than p2's id_num, returns a negative number if p1's id_num is less than p2's id_num, and returns 0 otherwise. It doesn't matter what actual value the positive and negative numbers have, as long as they are positive and negative, respectively.
   - **compare_name()**: compares the names in the records pointed to by p1 and p2, using the built-in strcmp() function, as follows:
     - Compare the last names using strcmp(). If the last names are not the same, return the value that strcmp() returned. Otherwise, compare the first names using strcmp() and, if they are not the same, return the value that strcmp() returned. Otherwise, compare the middle initials (which are chars, not strings) and return a positive number if p1's middle initial is greater, return a negative number of p2's middle initial is greater. If the middle initials are the same, return the result of using compare_id_number() to compare the id numbers, above.
   - **compare_age()**: returns a positive number if p1's age is greater than p2's age, a negative number if p1's age is less than p2's age, and otherwise returns the result of using compare_name() to campare the names, above.

- **compare_salary()**: returns a positive number if p1's salary is greater than p2's salary, a negative number if p1's salary is less than p2's salary, and otherwise returns the result of using compare_name to campare the names, above.

5. Write a function **new_record()** that takes a last name, first name, middle initial, age, id number, and salary as parameters and returns a pointer of type PERSONNEL_REC *. It should use malloc() to allocate space for a new PERSONNEL_REC, populate the fields of the record with the values of the corresponding parameters, and return the address of the new record. Important: malloc() should also be used here to allocate space to hold the strings for the first and last names. To allocate the char arrays of the right size for the first and last names, you can use the built-in strlen() function (don't forget to allocate an extra byte for the terminating 0). To copy the first and last names from the parameters into the corresponding fields of the record, use the strcpy() function.

6. Write a function **read_record()** that performs the following actions:
   - Uses scanf() to read a last name, first name, middle initial, age, id number, and salary from the terminal (standard input), in that order. Your code does not need to read input from a file.
   - If scanf() returns EOF, indicating that the end of the input has been reached, then read_record() should return NULL.
   - Otherwise, read_record() should create a new PERSONNEL_REC using malloc() and populate the record with the read-in values by calling new_record(), above, and then return a pointer to the new record.

   You can assume that the last name and first name are less than 100 characters each. As will be discussed in class, using scanf() this way is susceptible to a buffer-overflow attack, but in this first assignment, that's OK.

7. Define a function **print_record()** that takes a pointer to a personnel record and prints all the fields of the record using printf.

8. The rest of the assignment will be about creating binary search trees and a circular doubly-linked list, where each node in a binary search tree and each cell in the list points to a PERSONNNEL_REC.

   Using typedef, define a struct type **NODE** for use in a tree, where a NODE contains the following fields:
   - **record** that is a pointer to a PERSONNEL_REC.
   - **left** and **right** that are each a pointer to a NODE (representing the left and right children of the node).

9. In the binary search trees to be implemented here, the value of the left child of a parent node is less than or equal to (<=) the value of the parent node and the value of the right

child is strictly greater (>) than the value of the parent. There will be four binary search trees, for ordering the personnel records by id number, name, age, and salary.

Declare four global variables, each of type NODE * representing the root of a binary search tree: **name_root**, **age_root, id_num_root**, and **salary_root**. These global variables should all be initialized to NULL.

10. Define a function **insert_personnel_record()** that takes the following parameters:
    - A pointer to a root. Important: Since a root is of type NODE *, a pointer to a root is of type NODE **
    - A pointer to a PERSONNEL_REC
    - A pointer to a comparison function of the type of the comparison functions above (e.g. compare_name() and compare_id_number()).

    The insert_personnel_record() function should create a new NODE using malloc, set the record field of the node to point to the passed-in personnel record, and insert the node into the tree rooted at the passed-in root in the correct place, using the passed-in comparison function. You don't need to do anything to ensure the tree is balanced.

11. Define a function **traverse_and_print_records()** that takes a pointer to a tree (so of type NODE *) as a parameter and traverses the tree in order, printing out the record encountered at each node by calling print_record(). Note that in-order traversal is most easily specified recursively using the following algorithm:

```
traverse(p) = if p is null, then return
              else:
                traverse(p->left)
                visit p
                traverse(p->right)
```

    where p is a pointer to a node. Visiting p, in this case, entails printing out the personnel record found at p.

12. A circular doubly-linked list is a linked list with the following properties:
    - Each cell in the list has a **next** field that points to the next cell in the list.
    - Each cell in the list also has a **prev** field that points to the previous cell in the list.
    - The next field of the last cell in the list points to the first cell in the list, and the prev field of the first cell points to the last cell.

    Using typedef, define a struct type **CELL** that contains the following fields:
    - **record** that is a pointer to a PERSONNEL_REC
    - **next** and **prev**, each of which is a pointer to a CELL.

There will only be one circular doubly-linked list in this assignment, so declare a single global variable **head** of type CELL * and initialize it to NULL. Notice that there is no need for a tail variable, since – once cells are added to the list – head points to the first cell and head->prev points to the last cell of the list.

13. Define a function **insert_record_in_list()** that takes a pointer to a PERSONNEL_REC as a parameter. It should allocate a new CELL using malloc, set the record field of the new cell to point to the passed-in personnel record, and add the cell to the end of the list.

14. Define a function **insert_from_tree_into_list()** which takes a pointer to a tree (of type NODE *) as a parameter and traverses the tree using an in-order traversal (see the above recursive in-order algorithm). At each node that it encounters in the tree, it inserts the personnel record at that node into the doubly-linked list by called insert_record_in_list(). No new personnel records should be created (so no calls to malloc()) and no nodes should be deleted from the tree. When the function completes, the cells in the linked list should point to the same personnel records as the nodes in the tree.

15. Using #define, define the constant **FORWARD** to be 0 and define the constant **BACKWARD** to be 1.

    Then, define a function **print_list()** that take a parameter **direction** of type integer. If direction is FORWARD, the personnel records in the doubly-linked list should be printed in order, i.e. starting at the head and going to the end of the list. If direction is BACKWARD, the personnel records should be printed out in reverse order, i.e. starting at the end of the list and going backwards to the head of the list.

16. Define a function **print_n_records()** that takes an integer **n** as a parameter and prints the first n personnel records in the doubly-linked list. Note that if n is greater than the number of the cells in the list, some of the records in the list should be printed more than once because the list is circular.

17. Finally, define the **main()** function that performs the following actions:
    - Repeatedly reads in a PERSONNEL_REC by calling read_record() and adds the record to each of the following binary search trees by calling insert_personnel_record() multiple times:
        - The tree rooted at name_root and ordered by name (using compare_name(), above)
        - The tree rooted at id_num_root and ordered by id number (using compare_id_number())
        - The tree rooted at age_root and ordered by age (using compare_age())
        - The tree rooted at salary_root and ordered by salary (using compare_salary())

      The reading should stop when there is no more input (i.e. when read_record() returns NULL). Once a personnel record is read in, it should not be duplicated. A single personnel record will be pointed to by one node in each of the four trees above.

- For each of the above trees ordered by name, id number, age, and salary, print out an appropriate header (e.g. “SORTED BY NAME”) and then print out the personnel records in the tree by calling traverse_and_print_records().
- Call insert_from_tree_into_list(), passing in name_root, to put the personnel records from the tree ordered by name onto the doubly-linked list. At the end of the call, the list will contain personnel records ordered by name.
- After printing a header (e.g. “Printing list in forward direction”), call print_list() to print the personnel records in the list in the forward direction.
- After printing a header (e.g. “Printing list in backward direction”), call print_list() to print the personnel records in the list in the backward direction.
- After printing a header (e.g. “Printing 120 records”), call print_n_records() to print the first 120 records in the list.

Running and testing your program

First – and do this only once – perform the following steps:

a. Download one of the following compressed files that has been posted under Assignment 1 on Brightpace to the same directory where your code is:
   - a1_macOS.tgz (for Mac)
   - a1_linux.tgz (for Linux)
   - a1_cygwin.tgz (for Cygwin on Windows)

b. Decompress the file you downloaded by typing the following command into a shell (i.e. a Mac terminal, a Linux shell, or a Cygwin shell):

   tar -xzvf a1_*XXX*.tgz

   where *XXX* is either “macOS”, “linux”, or “Cygwin”. In the directory, you should see two new files, input.txt and ben_assign1 (on Windows it will be called ben_assign1.exe).

Next, to compile your program each time, in a shell type

```
gcc -o assign1 assignment1.c
```

To run your program using the input contained in input.txt, above, type

```
./assign1 < input.txt
```

You should compare the output of your program to the output produced by the professor’s version of the program. To run the professor’s program, type

```
./ben_assign1 < input.txt
```

If you have questions about the assignment that are not specific to your code, you can post them on the class forum for Assignment 1 and the professor will answer. If you are stuck and need help with your code, email the class tutor that you have been assigned to. Do not email the professor.