CP264: Fall 2021
Final Exam

Final Details:

Date: Thursday, December 16, 2021

Time: 8:30 – 10:30 am (+ 30 minutes for technical issues).

Format: Take-home, open book

Total Grade: 25

STOP:
READ THE POLICIES AND INSTRUCTIONS CAREFULLY BEOFORE YOU PROCEED

Final Exam Guidelines:

1- The final is designed to be completed in 2 hours. To give you extra time, an additional 30 minutes are added to accommodate for any technical difficulties.

2- No late submissions will be accepted for the final. Therefore, it is recommended that you upload your solution after each question. Note, that your new submission will override your previous submission permanently.

3- The final exam is open book, open notes and open internet access. You can search online for syntax and technical issues, but you cannot seek assistance of a third entity, whether in person or over the internet.

4- If you have any questions, post it in the forums. Do not post screenshots of your solution. I will be checking the forum sometime around 9:30 am.

5- If you have any comments or notes that you need to inform the instructor about relevant to your solution, write it on the designated box on to top of the final.c file.

6- If you encounter any technical issues during the final exam, you can contact the ICT support team which will be available during the exam time. The email address: examquestions@wlu.ca

Submission Rejection:

Your submission would be rejected in the following three scenarios:

1- Providing final.c without the academic honesty certification

2- Providing a submission through email.

3- Providing a submission, using any means, outside the final exam window.

Grading Guidelines:

1- The first and second questions are worth 8 pts, while the third question is worth 9 pts.

2- If your code does not match the expected output:

   a. If it is a minor formatting issue: -1 pt

   b. If it is an invalid output: -2 pts.

3- If your program crash, for any reason, then you lose 50% of the mark for each task that causes a crash. To avoid such scenarios, you can inform the instructor on commenting some lines in the testing file or in your own solution.

4- The instructor will be inspecting your code. Therefore, you do not necessarily receive full mark with 100% match. The instructor will be checking for major issues, not minor coding issues. Major issues include, using illegal operations (even if they produce correct output) and not following task guidelines.

5- Be considerate of code readability (e.g. naming variables) and to some extent efficiency. You will not lose marks on efficiency, unless you make something excessive, like providing an O(n^3) solution which could be solved in O(n).

Deductions:

There is a 2 pts deduction (Max 6 pts) for each of the following violations:

1- Failing to provide student credentials in “final.c” file.

2- Having any compilation error. (2 pts for each error).

3- Having warnings. Each warning would be counted as -0.25 pts (max 2 pts).

4- Including a header file other than the ones provided to you.

5- Failing to submit only one file which is: “final.c”

Files and Coding Instructions:

1- Create a C project in eclipse called *final*.

2- Download the final.zip files and extract the files into to a folder in your machine. Drag and drop the files into the final project in eclipse.

3- Rename the file: “final_template.txt” to “final.c”.

4- Open the file and **Enter your credentials on top of file**

5- Before submission, type the following certification on top of the file:

"I certify that I completed this final exam according to academic honesty guidelines. I attest that I did not use any external help, neither in person nor virtually. I understand that plagiarizing will lead to my failure in the course, in addition to other penalties according to the University policies".

6- Do not edit any file, other than "final.c". However, you may comment some lines in the testing file for your own testing purposes.

7- Do not include any other files to your project, because it is redundant.

8- When you are done, you need to submit only your "final.c". Do not export the project or upload a .zip file.

9- For every function that you complete, test it using "final_test.c" file. The results should match those presented in "final_output.txt".

Final Exam Structure:

The file "data_structure.h" and "datda_structure.c" contains specific implementations of some ADTs that would be used in the final. These have been tested and verified to work for the final exam purposes. Most of these functions are exactly as you studied in the course, but some have been altered slightly without changing its functionality. Also, some functions have been removed because they are not relevant to the final exam. All comments and docstrings have been removed to save space.

The file "final.h" contains the function prototypes and structs for the tasks that you will be working with during the final exam. You should not alter this file.

The file "final.c" is where all your solution should be placed. Each task should be placed in the designated area. You may add up to one helper functions for each task. These helper functions should be placed in the same area as the relevant task. You can either place it above the function or add a prototype on top of the file.

The file "final_test.c" contains the testing methods. It is similar to how we have been using the testing file in the course.

Task 1: Sorting Stacks

Implement the following function:

```
Stack* sort_stacks(Stack *stacks[5], const int size)
```

The function receives an array of five Stack pointers (Note: this is an array of Stack pointers, not an array of Stacks). The input parameter size determines how many stacks from the array should be used. For instance, if the value of size is 3, then only the first three stacks from the array should be used.

The objective of the function is to sort the data in the stacks. The result is stored in a new stack which is returned by the function.

The sorting works by comparing the top of the stacks, the highest value is popped out and pushed into the output stack. The above sorting step is repeated until there are no more data in all stacks.

Assume the following:

1- The input stacks will always contain at least one non-empty stack. You don't have to worry about the scenario when all input stacks are empty.
2- All stacks contain Data items which are integers.
3- Every input stack has data items sorted in descending order, i.e., the top of the stack is the highest value and the bottom of the stack is the lowest value.
4- The input stacks may have equal or different sizes
5- The input array of pointers will have at least 2 stacks. This means the minimum value of size is 2. Therefore, you do not have to worry about a scenario in which all pointers are NULL.

The output stack should have a capacity equal to the number of data items in all input stacks. The data will be sorted in ascending order, i.e., the top of the stack is the lowest value and the bottom of the stack is the highest value.

Testing the function will produce the following output:

```
-----------------------------
Start testing: sort_stacks:

Case 1:
Stack Size = 2 <12-10>
Stack Size = 2 <13-11>
output:
Stack Size = 4 <10-11-12-13>

Case 2:
Stack Size = 5 <14-13-12-11-10>
Stack Size = 3 <15-14-13>
output:
Stack Size = 8 <10-11-12-13-13-14-14-15>

Case 3:
Stack Size = 4 <13-12-11-10>
Stack Size = 6 <10-9-8-7-6-5>
Stack Size = 5 <15-13-11-9-7>
output:
```

```
Stack Size = 15 <5-6-7-7-8-9-9-10-10-11-11-12-13-13-15>

End Testing: sort_stacks
-----------------------------
```

Task 2: Deleting Middle of a Linked List

Implement the following function:

```
int delete_mid_list(List* list);
```

The function deletes the middle element of a given linked list. The index of the deleted element is returned.

For instance, if the list contains three data elements, then the function deletes the element at index 1 and returns the integer number 1.

If the list contains two data items, then the function deletes the element at index 0, and returns the integer number 0.

Note that you don't have a delete function in the provided linked list implementation. Therefore, you need to perform manual deletion which adjusts the order of nodes, the head pointer and linked list size accordingly.

If the given linked list is empty, the function should return -1 and print the following error message:

```
"Error(delete_mid_list): empty list\n"
```

If tested, the function should produce the following:

```
-----------------------------
Start testing: delete_mid:

Create Linked list with 5 elements
10-->20-->30-->40-->50-->NULL

calling delete_mid_list:
Index of deleted item = 2, list after deletion:
10-->20-->40-->50-->NULL

calling delete_mid_list:
Index of deleted item = 1, list after deletion:
10-->40-->50-->NULL

calling delete_mid_list:
Index of deleted item = 1, list after deletion:
10-->50-->NULL

calling delete_mid_list:
Index of deleted item = 0, list after deletion:
50-->NULL

calling delete_mid_list:
Index of deleted item = 0, list after deletion:
NULL
```

```
calling delete_mid_list:
Error(delete_mid_list): empty list
Index of deleted item = -1, list after deletion:
NULL

destroy linked list
End Testing: delete_mid
------------------------------
```

Task 3: Multi-Data Structures

Throughout this course, we have been using data.h and data.c files. The implementation focused on defining Process as Data or integers as Data. In this task, you will define a new data struct that works for both Process and int.

The file final.h defines the following structure:

```
typedef struct {
      void *item;      //pointer to data item
      char type; //'i' = integer, 'p' = process
} MData;
```

The struct MData could either contain an integer or a Process. To distinguish between the two scenarios, the data member `type` is initialized to `'i'` if it stores an integer and is initialized to `'p'` if it stores a Process. The `item` data member is a generic pointer, that could point to a Process or an integer.

You need to implement the following four functions:

1- `MData* create_mdata(void *item, char type);`
The function receives a pointer to a Process or to an integer along with a type. If the item pointer is NULL, the function should print an error message and return a NULL pointer.

Just as all create functions, the function needs to use malloc to create an MData pointer and return it. You need to be careful about the malloc size of the pointer. You can't use malloc(sizeof(MData)) because you do not know the size of the void*. Therefore, you need to specify a size depending on the scenario, i.e. Process or integer.

If the MData pointer that you plan to return is called d, then d->item should also be dynamically allocated. You should not use pointer assignment, i.e. d->item = item.

If the `type` is any value other than `i` or `p`, the function prints an error message and returns a NULL pointer.

2- `void destroy_mdata(MData **d);`
The destroy function should set all numbers to 0, all strings to an empty string, deallocate any dynamically allocated data and set all pointers to NULL. The function asserts that the given mdata is not NULL.

3- `void print_mdata(MData *d);`

The function prints the type and the contents of the MData. See the sample output. If the pointer is NULL, the function prints <NULL MData>. If the type of mdata is any value other than i or p, the function prints an error message.

4- `MData* copy_mdata(MData *d);`

The function creates a duplicate copy of the given mdata. The function asserts that the given pointer is not NULL and if the `type` is other than `i` or `p`, it prints an error message and returns a NULL pointer.

Hint: In your implementation, you may directly call functions defined in Process.h.

Testing the above functions would produce the following output:

```
------------------------------
Start: Testing MData:

Testing create_mdata:
MData of type p created successfully
MData of type i created successfully
Error(create_mdata): invalid data type
Error(create_mdata): invalid pointer

Testing print_mdata:
type = process, value = [30](1000010000,1)
type = integer, value = 10
Error(print_mdata): unsupported data
<NULL MData>

Testing copy_mdata:
Copy data of type p: type = process, value = [30](1000010000,1)
Copy data of type i: type = integer, value = 10
Copy data of type f: Error(copy_mdata): unsupported data
<NULL MData>

Testing destroy_mdata:
mdata of type p successfully destroyed
mdata of type i successfully destroyed

End: Testing MData
------------------------------
```

Good Luck

It was wonderful having you in my class