Homework - Homework 3

Topics: Lists (single-linked), proper code testing, time complexity, run with Valgrind, draw the relevant nodes and the readjustment of links.

Task 1 (12 pts)

See 2320_H3.pdf for this problem.
Here is a docx version of the document, for your convenience. If you are using this, save it as a pdf when finished and submit the pdf.

Task 2 (10 pts.) - Time complexity

For each function discussed in Task 3, specify the time complexity (in Theta notation) that can be attained for that function, using lists **as defined** in list_hw.c.
You must give the complexity of the functions that you wrote, as you wrote them. For example if your implementation is inefficient you should give the run time of your implementation (not that of some other efficient implementation).
Also if your function calls a helper function that you wrote or if it calls some other already provided list function, you should take in consideration the time complexity of that function as well in your answer.

Remember that the time complexity describes the worst-case behavior.
A brief justification is sufficient. You do NOT need to use the table method.
Also note that these points will not be distributed equally per function, but given as a total. Therefore if any of the above functions requires a more complicated time analysis it may get more points than the others.
Put your answer in the 2320_H3.pdf file.

Task 3 (78 pts.) - Single-linked list

Summary: For this task You are given 3 files (list_hw.h, list_hw.c, and instructor_client.c). Download them, and compile and run them on omega as given. They should run fine. File list_hw.c has 4 stubs (place-holders) for functions that you will have to implement. When compiling with the original list_hw.c, it will call these functions, but they will not do anything. After you implement them, they should do what is required.

Provided files:

- list_hw.c - The function stubs (where you will write your code) are at the end.
- list_hw.h - header file. Do not modify it.
- instructor_client.c - client program. Do not modify it.
- sample run - See the expected behavior of your program and compilation instructions here.
- 2320_H3.pdf - Fill-in the written answers for all tasks in here. Required test cases for Task 1 are also here.
  Here is a docx version of the document, for your convenience. If you are using this, save it as a pdf when finished and submit the pdf.
- data_1.txt - This file shows how the build_list_of_ints() function will read data and stop.

Below are given 4 functions that you must implement in list_hw.c and the requirements that your code must meet (e.g. no memory errors, keep the function signature, etc.). The list_hw.c file already has stubs for these functions. You just need to go in there and put the actual code for them.

1. (15 points) **struct node * sublist(struct node * A, struct node * pos_list)**. This function returns a list that contains the values that appear in list A at positions given in pos_list
   - Requirement 1: the values should appear in the result in the order given by pos_list. For example if
     A: 15 -> 100 -> 7 -> 5 -> 100 -> 7 -> 30 and

pos_list : 3 -> 0 -> 6 -> 4
The resulting list will be A[3] -> A[0] -> A[6] -> A[4] , which gives values: 5 -> 15 -> 30 -> 100.

- Requirement 2: the result should be a deep copy, so that any future changes in list A cannot possibly affect the result, and any future changes in the result cannot possibly affect list A. (List A should remain as it was after building the sublist.)
- Requirement 3: DO not copy the nodes in an array and then build a list from an array. When found the node in A, make a new node with the same value and insert it in the result list.
- Requirement 4: DO not make a copy of a list and change the data there. Build the result list by starting with an empty list and then creating new nodes and inserting them at the end of it.
- The list pos_list CAN have repetitions. E.g.:
  list A: 10 -> 5
  pos_list: 0->0->1->-0->1->1->0.
  produces list: 10->10->5->10->5->5->10
  (This example also shows that the size of A is unrelated to the size of pos_list.)

2. (12 points) **void delete_occurrences(struct node * A, int V)**. This function should delete (and free) all nodes in list A that contain value V. For example if
   A: 15 -> 100 -> 7 -> 5 -> 100 -> 7 -> 30 and we delete value 7, list A will become:
   A: 15 -> 100 -> 5 -> 100 -> 30

3. (13 points)**void swap_first_third(struct node * A)**. This function should swap the first and the third nodes in list A by adjusting the links, not copying the items. For this function you need to:
   1. Write the code in list_hw.c
   2. Make a drawing like the one did in class to show how the links are adjusted. In order to make it easy to correct the drawing, INCLUDE A COPY OF **void swapFirstThird(list A)** NEXT TO THE DRAWING.

   Special cases:
   1. If the list is empty or null, no swap is needed.
   2. If the list has only one item, no swap is needed.
   3. If the list has 2 items, they should be swapped: if A is 3 -> 7 , it becomes 7 -> 3.

4. (10 points) **student_test_sublist()** - your code to test the `sublist()` function. This code should also not have memory errors. You can use in list_hw.c the two methods from the instructor_client.c that make lists from array and from user (build_list_of_ints() and array_2_list(int arr[], int sz) .
   Put your answers in 2320_H3.pdf file indicating if your function passes or fails that test. Your function "failed a test case" if it did not have code to deal with that case and as a result of that, it either crashed or it produced an unexpected effect, e.g.: it used a pointer that was not initialized, or it accessed the wrong data.
   If you did not write the code for a required case you will not get any credit for that test case.

5. (8 points) quality of code - good variable names, comments, code is not convoluted.

6. You must keep the signature for the given functions as shown here. We will use test code that will call your functions expecting this format. - NO CREDIT WILL BE GIVEN IF THE CODE DOES NOT COMPILE DUE TO CHANGE IN FUNCTION SIGNATURES OR SYNTAX ERRORS.

7. You cannot use arrays to make working with lists easier. E.g., do not copy the data in an array and then access the numbers from the array. The function "array_2_list" should only be used to create lists that are used for test cases in "run_stduent_test". It should NOT be used in the other functions (e.g. sublist()) to bypass the work that should normally be done with lists, such as creating a list by repeated insertions.

8. (**20 pts**) Your program should not have any memory leaks or memory related errors, even if the program 'computes the correct results'.
   If you are using a function already provided in the list_hw.c file and that results in a memory error, you need to address it: either do not use the provided function (write your own) or call the provided function only for cases that it handles correctly. You should still NOT modify any of the provided functions.
   You can check for memory leaks using the Valgrind program on omega. To run your code with it and get a report of memory leaks at the end, instead of:

`./instr`

run:

```
valgrind --leak-check=yes ./instr
```

will show a report at the end similar to:

```
==1210==
==1210== HEAP SUMMARY:
==1210==     in use at exit: 0 bytes in 0 blocks
==1210==   total heap usage: 142 allocs, 142 frees, 2,272 bytes allocated
==1210==
==1210== All heap blocks were freed -- no leaks are possible
==1210==
==1210== For counts of detected and suppressed errors, rerun with: -v
==1210== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 4 from 4)
----------------------------------------------------------------
```

In the above report:

1. On the last line, you must make sure you get *0 errors from 0 contexts*.
2. You do not need to handle suppressed errors. E.g. it is fine if you get: *(suppressed: 4 from 4)*
3. under "total heap usage: " you have as many "allocs" as "frees"

Here are examples of a memory errors reported by Valgrind. Your program should NOT have any of these:

1. memory leak:

   ```
   ==27047== 400 (16 direct, 384 indirect) bytes in 1 blocks are definitely lost in loss record 3 of 3
   ```

2. invalid pointer (memory) use:

   ```
   ==9814==  Invalid write of size 1
   ==9814==    at 0x804841E: main (example2.c:6)
   ==9814==  Address 0x1BA3607A is 0 bytes after a block of size 10 alloc'd
   ```

Read more about using Valgrind and understanding the errors on this page from Cprogramming.com.
See the Code & Programming Requirements page for instructions on how to compile and run in order to see LINE NUMBERS for the memory errors that Valgrind reports.
For compilation instructions see sample run .

How to submit

15 points penalty for wrong filenames, name of folder, type of archive (must be ZIP).
The assignment should be submitted via Canvas.
Place all your files in a folder called 2320_HW3. Zip that folder (it will produce a file called 2320_HW3.zip). Submit `2320_HW3.zip`. No other forms of compression accepted, contact the instructor or TA if you do not know how to produce .zip files.
The zipped directory should contain the following documents:

- **2320_H3.pdf**. A PDF file containing your solutions for each task. **Include your name and student ID at the top of this document. It may be printed and graded on paper.**
  If your code needs other commands (or arguments) than the ones shown in the sample run file, in order to compile or run include those commands in 2320_H3.pdf. If missing: 10 points penalty per assignment and the assignment will not be graded until you give that information to the TA.
- **list_hw.c**
  DO NOT SUBMIT (and do not modify): `instructor_client.c` ,`list_hw.h`.

Programs must have the .c extensions and must run on omega.uta.edu (when compiled with the list_hw.h and instructor_client.c).

PARTIAL CREDIT : Partial credit will NOT be given for programs that have compilation or runtime errors. You are responsible

for debugging your own code.

IMPORTANT: Pay close attention to all specifications on this page, including file names and submission format. Even in cases where your answers are correct, points will be taken off liberally for non-compliance with the instructions given on this page (such as wrong file names, wrong compression format for the submitted code, and so on). The reason is that non-compliance with the instructions makes the grading process significantly (and unnecessarily) more time consuming. Contact the instructor or TA if you have any questions.

Back to Homework page .