G51PGA 2017 Coursework 4

Introduction

This coursework focuses on implementing four algorithms which employ linked lists, queues and stacks. There are four tasks to complete, details of which are given below. You should complete all four tasks. Links to external pages are provided in the details below which you will need to follow and read to complete each task.

Overview

As with the last coursework, you will implement a number of library functions. For each task, an executable program is provided which will include and use the library functions which you implement. You will not need to add any files to your forked repository for this coursework, as all the required files are present. For each task, you will add function implementations to certain source files in your repository. A Makefile is also provided to build and test your library functions. All tasks will be compiled with the gcc -ansi -pedantic-errors switches.

Assessment

This coursework is worth 30% of your final course grade. The points awarded for each task are as follows:

Compilation
Implementation
Execution
Total

Task 1

2
1
4
7

Task 2

2
1
4
7

Task 3
2
1
10
13

Task 4
2
1
6
9

For each task, you are awarded two points if your program compiles correctly: one point if it compiles without errors, and a further one point if it compiles without errors and warnings. If your program follows the implementation instructions given in the task details below, you are awarded one point.

For each of Tasks 1 and 2, if your program is implemented so that no memory leaks occur when it is executed, you are awarded two points. If your program produces the correct output, you are awarded a further two points.

For Task 3, there are five test cases which your program will be tested on. These test cases are described in the task details below. For each test case, if your program is implemented so that no memory leaks occur when it is executed with the test case, you are awarded one point. If your program produces the correct output with the test case, you are awarded a further one point.

For Task 4, you will be awarded points for each of the first three test cases used in Task 3. For each of these three test cases, if your program is implemented so that no memory leaks occur when it is executed with the test case, you are awarded one point. If your program produces the correct output with the test case, you are awarded a further one point.

The contribution towards your final course grade is calculated as follows:

floor(30 * (task1_points_awarded + task2_points_awarded + task3_points_awarded + task4_points_awarded) / 36)

Your provisional score for each task can be viewed on GitLab after every push to projects.cs.nott.ac.uk. Your final score will be based on the code in the last commit pushed to projects.cs.nott.ac.uk before the coursework deadline. Commits pushed after the coursework deadline will be disregarded. After the coursework deadline, your code will undergo further review and, based on this review, your provisional score may go up or down. This further review will include checks for code plagiarism and for trivial implementations e.g. implementations just containing an empty main function or clearly not written following the task guidelines. Final scores will be published on the G51PGA moodle pages around a week after the coursework deadline.

Your repository contains a file called .gitlab-ci.yml. This file is used during the assessment

process and must not be removed or edited in any way. Any tampering with this file will result in a score of zero for this coursework.

Task 1

Overview

Your task is to implement the insertion sort algorithm to sort data stored in a linked list.
A test program task1_test.c and linked list library files linked_list.h and linked_list.c are provided for you. You must not remove or edit these files in any way.

Details

The insertion sort algorithm is a basic sorting algorithm which can be used to sort an array or list. Pseudocode for the insertion sort algorithm was given in lectures. In this task, you will implement a function insertion_sort which implements the insertion sort algorithm for linked lists.
The interface for the insertion_sort function is given in the sort.h file, and a skeleton implementation and further notes are provided in the sort.c file. Your task is to complete the implementation of the insertion_sort function in sort.c. You should only edit the sort.c file, and must not edit the sort.h file.
You do not need to implement a linked list library, as a full implementation is provided in the linked_list.h and linked_list.c files. You must not edit these two files in any way. You should use this implementation when implementing your insertion_sort function.
A test program, task1_test.c, is provided which will include your implementation of insertion_sort and use your implementation to sort a linked list. You must not edit the file task1_test.c. To compile your implementation and the test program, type:
$ make task1_test
To compile your implementation and the test program and then run the test program, type:
$ make task1_test_run
A typical output from a correctly working implementation is:

```
$ make task1_test_run
./task1_test harmonious pencil push naive jittery greedy billowy practise lock star appear record precede pen lace
appear
billowy
greedy
harmonious
jittery
lace
lock
naive
pen
pencil
practise
precede
```

push
record
star
Your implementation should handle dynamically allocated memory correctly i.e. free all dynamically allocated memory. As discussed in lectures, a good tool for assessing if a program has handled dynamic memory allocation correctly is valgrind. To check your implementation using valgrind you can type:
$ make task1_test_memcheck
If your program has correctly handled dynamic memory allocation, the last line of output should read:
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Task 2

Overview
In this task you will implement the merge sort algorithm to sort data stored in a linked list.
A test program task2_test.c and linked list library files linked_list.h and linked_list.c are provided for you. You must not remove or edit these files in any way.

Details
The merge sort algorithm is an efficient sorting algorithm which can be used to sort an array or list. In this task, you will implement a function merge_sort which implements the merge sort algorithm for linked lists.
Similar to Task 1, the interface for the merge_sort function is given in the sort.h file, and a skeleton implementation and further notes are provided in the sort.c file. Your task is to complete the implementation of the merge_sort function in sort.c. You should only edit the sort.c file, and must not edit the sort.h file.
As with Task 1, you do not need to implement a linked list library, as a full implementation is provided in the linked_list.h and linked_list.c files. You must not edit these two files in any way. You should use this implementation when implementing your merge_sort function.
You are free to implement either a top-down or bottom-up merge sort with or without recursion. Extra functions can be added to sort.c, and in the case of merge sort, you will probably want to implement an additional function to merge two sorted lists, as well as the merge_sort function itself.
A test program, task2_test.c, is provided which will include your implementation of merge_sort and use your implementation to sort a linked list. You must not edit the file task2_test.c. To compile your implementation and the test program, type:
$ make task2_test
To compile your implementation and the test program and then run the test program, type:
$ make task2_test_run
A typical output from a correctly working implementation is:
$ make task1_test_run
./task2_test harmonious pencil push naive jittery greedy billowy practise lock star appear record precede pen lace

appear
billowy
greedy
harmonious
jittery
lace
lock
naive
pen
pencil
practise
precede
push
record
star

Your implementation should handle dynamically allocated memory correctly i.e. free all dynamically allocated memory. To check your implementation using valgrind you can type:

$ make task2_test_memcheck

If your program has correctly handled dynamic memory allocation, the last line of output should read:

ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Task 3

Overview

In this task you will implement the shunting yard algorithm to convert a mathematical expression in infix notation to postfix notation.

A test program task3_test.c and linked list, stack and queue library files linked_list.h, linked_list.c, stack.h, stack.c, queue.h, and queue.c are provided for you. You must not remove or edit these files in any way.

Details

The shunting yard algorithm is a method for converting mathematical expressions from infix notation to postfix notation (also known as reverse Polish notation). In this task, you will implement a function infix_to_postfix which implements the shunting yard algorithm.

The interface for the infix_to_postfix function is given in the rpn.h file, and a skeleton implementation and further notes are provided in the rpn.c file. Your task is to complete the implementation of the infix_to_postfix function in rpn.c. You should only edit the rpn.c file, and must not edit the rpn.h file.

You do not need to implement a linked list, stack or queue library, as a full implementation is provided in the linked_list.h, linked_list.c, stack.h, stack.c, queue.h, and queue.c files. You must not edit these files in any way. You should use this implementation when implementing your infix_to_postfix function.

Your implementation should support the following five operators:

Operator
Symbol
Precedence
Associativity

addition
+
1
left

subtraction
-
1
left

multiplication
*
2
left

division
/
2
left

exponentiation
^
3
right

Operator precedence is given in the third column of this table, where a higher value indicates a higher precedence. Operator associativity is given in the fourth column of this table.

In addition to these operators, your implementation should support nested left and right parentheses (and).
Your implementation should accept integer and floating point numbers as operands. You can assume that valid operands will contain the characters 0, 1, ..., 9, ., + or -.
In order to implement the shunting yard algorithm, I would also suggest implementing several helper functions in rpn.c:

is_left_associative - returns true if an operator is left associative

is_operator - returns true if a character is an operator

operator_precedence - returns the precedence of an operator

test_precedence - compares the precedence of two operators and returns -1, 0, or 1 if the first operator has lower, equal or higher precedence compared to the second operator

is_number - returns true if a character is a number (you will also need to consider the previous and next characters to deal with signed numbers correctly)

As a first step, you could implement the shunting yard algorithm so that it correctly converts all infix expressions with positive numbers but without spaces. This will allow your implementation to pass test case 1 (see below for more details on these test cases) and gain the points for this test case. You could then improve your implementation to correctly parse infix expressions containing spaces (test case 2), and gain the points for this test case. You could then improve your implementation to correctly parse negative and positively signed numbers (test case 3) and increase your grade for this task. The final two test cases test whether your implementation correctly handles invalid infix expressions with mismatched parentheses and invalid operands respectively.
A test program, task3_test.c, is provided which will include your implementation of infix_to_postifx and use your implementation to convert the five test case mathematical expressions from infix to postfix. You must not edit the file task3_test.c. To compile your implementation and the test program, type:
$ make task3_test
To compile your implementation and the test program and then run the test program with the test cases, type:
$ make task3_tests_run
This will run the test program with the five test cases. The first test case tests whether your implementation parses all operators and brackets correctly. The second test case tests whether your implementation can also handle expressions with white space correctly. The third test case tests whether your implementation can parses positive and negative numbers correctly. The last two test cases test whether your implementation handles incorrect input correctly. A typical output from a correctly working implementation is:
$ make task3_tests_run

...

*** RUNNING TEST ***
$ ./task3_test "5+2-3*4/2^((1.43+2.57)/3)"
5
2
+
3
4
*
2
1.43
2.57
+
3
/
^
/
-
*** TEST COMPLETE ***

*** RUNNING TEST ***
$ ./task3_test "4 * (3.332 + 5.668) / 3^ 2.0"
4
3.332
5.668
+
*
3
2.0
^
/
*** TEST COMPLETE ***

*** RUNNING TEST ***
$ ./task3_test "-5 + 6 / (+2.76 + 1.24 + +1 ^ 2) - -10.2"
-5
6
+2.76
1.24
+
+1
2
^

+
/
+
-10.2
-
*** TEST COMPLETE ***

*** RUNNING TEST ***
$ ./task3_test "((1+7)-(1*6)"
error: unable to parse expression.
*** TEST COMPLETE ***

*** RUNNING TEST ***
$ ./task3_test "a+b"
error: unable to parse expression.
*** TEST COMPLETE ***
Your implementation should handle dynamically allocated memory correctly i.e. free all dynamically allocated memory. To check your implementation using valgrind you can type:
$ make task3_tests_memcheck
If your program has correctly handled dynamic memory allocation, the last line of output for each of the five test cases should read:
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

Task 4

Overview
In this task you will implement an algorithm to evaluate a mathematical expression in postfix notation.
A test program task4_test.c and linked list, stack and queue library files linked_list.h, linked_list.c, stack.h, stack.c, queue.h, and queue.c are provided for you. You must not remove or edit these files in any way.

Details
The algorithm to evaluate a mathematical expression in postfix notation was presented in lectures and is given on this page. In this task, you will implement a function evaluate_rpn which implements this algorithm.
The interface for the evaluate_rpn function is given in the rpn.h file, and a skeleton implementation and further notes are provided in the rpn.c file. Your task is to complete the implementation of the evaluate_rpn function in rpn.c. You should only edit the rpn.c file, and must not edit the rpn.h file.
As with Task 3, you do not need to implement a linked list, stack or queue library, as a full implementation is provided in the linked_list.h, linked_list.c, stack.h, stack.c, queue.h, and queue.c files. You must not edit these files in any way. You should use this implementation when implementing your evaluate_rpn function.

Your implementation should support all the operators, parentheses and operands described in Task 3. You should assume a precision of eight decimal places in your calculations.
As a suggestion, you might want to implement an evaluate_expression function in rpn.c as well as the evaluate_rpn function. The evaluate_expression function would take two operands and an operator and return the result of calculating this expression.
A test program, task4_test.c, is provided which will include your implementation of evaluate_rpn and use your implementation (as well as your implementation of infix_to_postfix from Task 3) to convert and evaluate the five test case mathematical expression described in Task 3 from infix to post. You must not edit the file task4_test.c. To compile your implementation and the test program, type:
$ make task4_test
To compile your implementation and the test program and then run the test program with the test cases, type:
$ make task4_tests_run
This will run the test program with the five test cases as described in Task 3 above. A typical output from a correctly working implementation is:
$ make task4_tests_run
...

*** RUNNING TEST ***
$ ./task4_test "5+2-3*4/2^((1.43+2.57)/3)"
2.23779683

*** TEST COMPLETE ***

*** RUNNING TEST ***

$ ./task4_test "4 * (3.332 + 5.668) / 3 ^ 2.0"
4.00000000
*** TEST COMPLETE ***

*** RUNNING TEST ***
$ ./task4_test "-5 + 6 / (+2.76 + 1.24 + +1 ^ 2) - -10.2"
6.40000000
*** TEST COMPLETE ***

*** RUNNING TEST ***
$ ./task4_test "((1+7)-(1*6)"
error: unable to parse expression.
*** TEST COMPLETE ***

*** RUNNING TEST ***
$ ./task4_test "a+b"
error: unable to parse expression.
*** TEST COMPLETE ***
Your implementation should handle dynamically allocated memory correctly i.e. free all

dynamically allocated memory. To check your implementation using valgrind you can type:
$ make task4_tests_memcheck
If your program has correctly handled dynamic memory allocation, the last line of output for each of the five test cases should read:
ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)