Assignment 3

CS 1037
Computer Science Fundamentals II

Due date: November 1, 2024, at 23:59

Learning Outcomes

Upon completion of this assignment, students will be able to:

- Record data processing using singly linked lists
- Apply doubly linked lists
- Implement linked list queues
- Implement linked list stacks
- Apply queues and stacks in expression evaluations

Introduction

This assignment focuses on developing your skills in C programming by working with singly and doubly linked lists, queues, and stacks and implementing an expression evaluator. You will create functions to handle list operations and stack and queue manipulations. This assignment will enhance your problem-solving abilities and provide hands-on experience with fundamental C programming principles.

Question 1

Record Data Processing using a Singly Linked List (12 points) – Develop two C programs, *myrecord_sllist.h* and *myrecord_sllist.c*, to implement a singly linked list for storing and processing student mark records. In *myrecord_sllist.h*, define the necessary data structures for the singly linked list, including the record and node structures and the list itself. Ensure that the appropriate function prototypes for manipulating the list are also declared.

In *myrecord_sllist.c*, implement the functions that will manage the list, including adding new records in a sorted manner, searching for records by name, deleting records, and cleaning up the list. The program should handle these operations efficiently, with proper memory management and error handling, to ensure reliable performance.

Files Provided

- *myrecord_sllist.h:* Header file containing structure and function declarations
- *myrecord_sllist_ptest.c:* Main function for testing your implementation

Implementation Details

The *myrecord_sllist.c* will implement the following functions:

- Insert a record into the linked list:
    - `void sll_insert(Record **head, char id[], float mark);`
- Search for a record by ID:
    - `Record* sll_search(Record *head, char id[]);`
- Delete a record by ID:

    - `void sll_delete(Record **head, char id[]);`
- Import data from a file into the linked list:
    - `void import_data(const char *filename, Record **head);`
- Process data (count, mean, standard deviation, and median):
    - `void process_data(Record *head);`

Refer to *myrecord_sllist.h* for detailed function specifications.

Compilation

```
gcc myrecord_sllist.c myrecord_sllist_ptest.c -o q1
```

Testing

You are tasked with writing your own testing main function and saving it in a file called *myrecord_sllist_ptest.c*. Use the test data provided in marks.txt alongside your *myrecord_sllist_ptest.c* file to test your implementation. The screen output from running your test should match the example provided in Appendix A.

Question 2

Record Data Processing using a Doubly Linked List (12 points) – Develop two C programs, *dllist.h* and *dllist.c*, to implement a doubly linked list for storing and managing character data values. In *dllist.h*, define the necessary data structures for the doubly linked list, including the node and list structures. The node should hold the character data and pointers to both the previous and next nodes. The list structure should track the number of nodes and pointers to the list's first and last nodes. Ensure that the appropriate function prototypes for managing the list are also declared.

In *dllist.c*, implement the functions to manage the doubly linked list. These functions should include creating new nodes, inserting nodes at both the start and end of the list, deleting nodes from the start or end, and cleaning up the entire list by freeing all allocated memory. To ensure reliable functionality, the program must handle these operations efficiently, with **proper memory management and error handling**.

Files Provided

- **dllist.h**: Header file containing structure and function declarations
- **dllist_ptest.c**: Main function for testing your implementation

Implementation Details

Create a C file named dllist.c that implements the following functions:

- Create a new node:
    - `Node* new_node(char data);`
- Insert a node at the beginning of the list:
    - `void dll_insert_start(Node **head, char data);`

- Insert a node at the end of the list:
  - `void dll_insert_end(Node **head, char data);`
- Delete a node from the beginning of the list:
  - `void dll_delete_start(Node **head);`
- Delete a node from the end of the list:
  - `void dll_delete_end(Node **head);`

Refer to *dllist.h* for detailed function specifications.

Compilation

```
gcc dllist.c dllist_ptest.c -o q2
```

Testing

You are tasked with writing your own testing main function and saving it in a file called *dllist_ptest.c*. Use the test data provided in marks.txt alongside your *dllist_ptest.c* file to test your implementation. The screen output from running your test should match the example provided in Appendix A.

IMPORTANT NOTE

In the following tasks, you will use the NODE structure and utility functions from *common.h* and *common.c* to implement linked list-based queues and stacks effectively. The NODE structure represents the nodes in a linked list, while functions like *new_node()*, *clean()*, and *display()* make it easier to create nodes, manage memory, and debug. The mytype() function helps identify different types of characters, which is especially helpful for expression evaluation. These utilities are crucial for performing operations like node creation, memory cleanup, and list visualization, and are key for building your queue (Question 3), stack (Question 4), and expression evaluator (Question 5).

Question 3

Linked List Queue Implementation (8 points) – Develop two C programs, *queue.h* and *queue.c*, utilizing *common.h* and *common.c* to implement a linked list-based queue for managing character data values.

In *queue.h*, define the queue structure, leveraging the node structure and common operation functions provided in *common.h*. The queue structure should track the number of nodes and pointers to the front and rear of the queue. Ensure that appropriate function prototypes for queue operations are declared.

In *queue.c*, implement the functions required to manage the queue using the utilities in *common.c*. These functions should include initializing the queue, enqueueing elements, dequeueing elements, and cleaning up the queue by freeing all nodes. Proper memory management and error handling must ensure efficient functionality.

Files Provided

- **queue.h**: Header file containing structure and function declarations
- **queue_ptest.c**: Main function for testing your implementation

Implementation Details

Create a C file named queue.c that implements the following functions:

- Initialize the queue:
    - `void queue_init(Queue *q);`
- Add an element to the queue:
    - `void enqueue(Queue *q, char data);`
- Remove an element from the queue:
    - `char dequeue(Queue *q);`

Refer to *queue.h* for detailed function specifications.

Compilation

```
gcc queue.c queue_ptest.c -o q3
```

Testing

Please test your programs using the provided public test file *queue_ptest.c*. The screen output should match the example shown in Appendix A.

Question 4

Stack Data Processing using a Linked List (8 points) – Develop two C programs, *stack.h* and *stack.c*, to implement a linked list-based stack for storing and managing data.

In *stack.h*, define the necessary data structures for the stack, including the node structure and the stack structure itself. The stack structure should track the stack's length and include a pointer to the top node of the linked list. Ensure that the appropriate function prototypes for managing the stack are declared.

In *stack.c*, implement the functions required to manage the stack. These functions should include pushing a node onto the stack, popping the top node from the stack, and cleaning up the entire stack by freeing all allocated memory. Each function should handle operations efficiently, with proper memory management and error handling, to ensure the stack operates reliably.

Files Provided

- **stack.h:** Header file containing structure and function declarations
- **stack_ptest.c**: Main function for testing your implementation

Implementation Details

Create a C file named stack.c that implements the following functions:

- Initialize the stack:
  - `void stack_init(Stack *s);`
- Push an element onto the stack:
  - `void push(Stack *s, char data);`
- Pop an element from the stack:
  - `char pop(Stack *s);`

Compilation

```
gcc stack.c stack_ptest.c -o q4
```

Testing

Please test your programs using the provided public test file *stack_ptest.c*. The screen output should match the example shown in Appendix A.

Question 5

Expression Evaluation Using Linked List Queue and Stack (20 points) – Develop two C programs, *expression.h* and *expression.c*, to evaluate arithmetic infix expressions. Use the queue data structure from Question 3 to represent the postfix expression and the stack data structure from Question 4 to evaluate the postfix expression.

In *expression.h*, define the necessary function prototypes to convert infix expressions to postfix and evaluate both postfix and infix expressions. The functions should utilize the queue to represent the postfix expression and the stack to perform the evaluation. Ensure proper handling of arithmetic operators and their precedence.

In *expression.c*, implement the functions to manage the conversion of infix expressions to postfix using the queue, evaluate the postfix expression using the stack, and return the value of the evaluated infix expression. These functions should handle all operations efficiently and accurately, ensuring the correct precedence of arithmetic operators. Proper error handling and memory management must be implemented to ensure reliable functionality.

Files Provided

- **expression.h**: Header file containing structure and function declarations

- **expression_ptest.c**: Main function for testing your implementation

Implementation Details

Create a C file named expression.c that implements the following functions:

- Convert infix to postfix:
  - `void infix_to_postfix(const char *infix, Queue *postfix_queue);`
- Evaluate postfix expression:
  - `int evaluate_postfix(Queue *postfix_queue);`
- Evaluate infix expression:
  - `int evaluate_infix(const char *infix);`

Compilation

```
gcc expression.c queue.c stack.c expression_ptest.c -o q5
```

Testing

Please test your programs using the provided public test file *expression_ptest.c*. The screen output should match the example shown in Appendix A.

What to submit?

- Submit your completed files as follows:
  Question 1:
  1. *myrecord_sllist.h*
  2. *myrecord_sllist.c*
  3. *myrecord_sllist_ptest.c*

  Question 2:
  4. *dllist.h*
  5. *dllist.c*
  6. *dllist_ptest.c*

  Question 3
  7. *common.h*
  8. *common.c*
  9. *queue.h*
  10. *queue.c*

  Question 4
  11. *stack.h*
  12. *stack.c*

  Question 5
  13. *expression.h*
  14. *expression.c*

  through the designated dropbox on OWL Brightspace.
- Do not submit any other files, including header files or executables.

- Ensure your code compiles and runs without errors on the university lab computers.

Submission Platform

- All submissions must be made through OWL Brightspace. Please do not email your files to the instructors or TAs, as emailed submissions will not be marked.

Deliverables

  - Place all required .c files into a single folder.
  - Name the folder with your student number (e.g., 12345678).
  - Compress the folder into a .zip file with the same name (e.g., 12345678.zip).

Resubmissions: You may resubmit your assignment as many times as needed before the deadline. Your last submitted version will be considered for grading. In addition, any resubmissions made after the deadline will incur late penalties.

Compatibility Reminder: Ensure your code compiles and runs on the university lab computers. Code that only works on your local IDE may result in lost marks.

Grading Criteria – Total: 70 points

Question 1: Record Data Processing by Singly Linked List (12 points)

- sll_insert function implementation: 3 points
- sll_search function implementation: 3 points
- sll_delete function implementation: 3 points
- process_data function implementation: 3 points

Question 2: Doubly Linked List (12 points)

- dll_insert_start function implementation: 3 points
- dll_insert_end function implementation: 3 points
- dll_delete_start function implementation: 3 points
- dll_delete_end function implementation: 3 points

Question 3: Linked List Queue (8 points)

- queue_init function implementation: 2 points
- enqueue function implementation: 3 points
- dequeue function implementation: 3 points

Question 4: Linked List Stack (8 points)

- stack_init function implementation: 2 points
- push function implementation: 3 points

- pop function implementation: 3 points

Question 5: Expression Evaluation (20 points)

- infix_to_postfix function implementation: 7 points
- evaluate_postfix function implementation: 7 points
- evaluate_infix function implementation: 6 points

Code Quality (10 points)

- Proper use of C programming conventions and best practices (4 points)
- Code organization and readability (3 points)
- Appropriate comments and documentation (3 points)
- Correctness and Efficiency (10 points)
- Code compiles without errors (2 points)
- Passes all provided test cases (4 points)
- Efficient implementation (e.g., proper use of pointers, avoiding unnecessary computations – 4 points)

Additional Rules

- Do not upload the .exe files! Penalties will be applied for this.
- Submit the assignment on time. Late submission policy will be applied.
- Forgetting to submit is not a valid excuse for submitting late.
- Assignment files must NOT be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.

Files to submit

Remember **you must do** all the work on your own. **Do not copy** or even look at the work of another student. All submitted code will be run through similarity-detection software.

Your submission will be evaluated using the grading criteria. To maximize your score, ensure that your implementation addresses all the aspects outlined above.

Good Luck ☺

Appendix A

Compilation and Output for Question 1

```
compile command: gcc myrecord_sllist.c myrecord_sllist_ptest.c -o q1
```

test run command: q1

```
-------------------
Test: ssl_insert

given linked list:length 0
sll_insert(A9 90.0): length 1 A9 90.0
sll_insert(A8 80.0): length 2 A8 80.0 A9 90.0
sll_insert(A7 70.0): length 3 A7 70.0 A8 80.0 A9 90.0
sll_insert(A6 60.0): length 4 A6 60.0 A7 70.0 A8 80.0 A9 90.0

-------------------
Test: ssl_search

given linked list:length 10 A0 100.0 A1 10.0 A2 20.0 A3 30.0 A4 40.0 A5 50.0
A6 60.0 A7 70.0 A8 80.0 A9 90.0
sll_search(A1): A1 10.0
sll_search(A3): A3 30.0
sll_search(A5): A5 50.0
sll_search(A10): not found

-------------------
Test: ssl_delete

given linked list:length 10 A0 100.0 A1 10.0 A2 20.0 A3 30.0 A4 40.0 A5 50.0
A6 60.0 A7 70.0 A8 80.0 A9 90.0
sll_delete(A2): length 9 A0 100.0 A1 10.0 A3 30.0 A4 40.0 A5 50.0 A6 60.0 A7
70.0 A8 80.0 A9 90.0
sll_delete(A4): length 8 A0 100.0 A1 10.0 A3 30.0 A5 50.0 A6 60.0 A7 70.0 A8
80.0 A9 90.0
sll_delete(A6): length 7 A0 100.0 A1 10.0 A3 30.0 A5 50.0 A7 70.0 A8 80.0 A9
90.0
sll_delete(A8): length 6 A0 100.0 A1 10.0 A3 30.0 A5 50.0 A7 70.0 A9 90.0
```

---

test run command q1 with data file

```
data processing test run command: q1 data

-------------------
Test: import_data

Import data from file
length:10
A0,100.0
A1,10.0
A2,20.0
A3,30.0
A4,40.0
```

```
A5,50.0
A6,60.0
A7,70.0
A8,80.0
A9,90.0

Test: process_data

count     10.0
mean      55.0
stddev    28.7
median    55.0

Sort records by scores
A0,100.0
A9,90.0
A8,80.0
A7,70.0
A6,60.0
A5,50.0
A4,40.0
A3,30.0
A2,20.0
A1,10.0
```

Compilation and Output for Question 2

```
compile command: gcc dllist.c dllist_ptest.c -o q2
```

`test run command: q2`

```
------------------
Test: new_node

new_node(A): A
new_node(B): B
new_node(C): C
new_node(D): D

------------------
Test: dll_insert_start

given dll: length 0 forward
dll_insert_start(A): length 1 forward A
dll_insert_start(B): length 2 forward B A
dll_insert_start(C): length 3 forward C B A
dll_insert_start(D): length 4 forward D C B A

------------------
Test: dll_insert_end

given dll: length 0 forward
dll_insert_end(A): length 1 forward A
dll_insert_end(B): length 2 forward A B
dll_insert_end(C): length 3 forward A B C
dll_insert_end(D): length 4 forward A B C D
```

```
resulted dll: length 4 backward D C B A

------------------
Test: dll_delete_start

given dll: length 4 forward D C B A
dll_delete_start(A): length 3 forward C B A
dll_delete_start(B): length 2 forward B A
dll_delete_start(C): length 1 forward A
dll_delete_start(D): length 0 forward

------------------
Test: dll_delete_end

given dll: length 4 forward D C B A
dll_delete_end(A): length 3 forward D C B
dll_delete_end(B): length 2 forward D C
dll_delete_end(C): length 1 forward D
dll_delete_end(D): length 0 forward
```

Compilation and Output for Question 3

```
compile command: gcc common.c queue.c queue_ptest.c -o q3
```

`test run command: q3`

```
------------------
Test: enqueue

enqueue((): length 1 data (
enqueue(2): length 2 data ( 2
enqueue(+): length 3 data ( 2 +
enqueue(3): length 4 data ( 2 + 3
enqueue()): length 5 data ( 2 + 3 )

------------------
Test: dequeue

given queue: length 5 data ( 2 + 3 )
dequeue() (: length 4 data 2 + 3 )
dequeue() 2: length 3 data + 3 )
dequeue() +: length 2 data 3 )
dequeue() 3: length 1 data )
dequeue() ): length 0 data
```

Compilation and Output for Question 4

```
compile command: gcc common.c stack.c stack_ptest.c -o q4
```

`test run command: q4`

```
------------------
Test: push

push(1): length 1 data 1
push(2): length 2 data 2 1
push(+): length 3 data + 2 1
push(3): length 4 data 3 + 2 1
push(a): length 5 data a 3 + 2 1

------------------
Test: pop

given stack: length 5 data a 3 + 2 1
pop() a: length 4 data 3 + 2 1
pop() 3: length 3 data + 2 1
pop() +: length 2 data 2 1
pop() 2: length 1 data 1
pop() 1: length 0 data
```

Compilation and Output for Question 5

```
compile command: gcc common.c stack.c queue.c expression.c expression_ptest.c
-o q5
```

test run command: q5

```
------------------
Test: infix_to_postfix

infix_to_postfix(1+2): 1 2 +
infix_to_postfix((1+2*3)): 1 2 3 * +
infix_to_postfix(10-((3*4)+8)/4): 10 3 4 * 8 + 4 / -

------------------
Test: evaluate_postfix

evaluate_postfix(1 2 +): 3
evaluate_postfix(1 2 3 * +): 7
evaluate_postfix(10 3 4 * 8 + 4 / -): 5

------------------
Test: evaluate_infix

evaluate_infix(1+2): 3
evaluate_infix((1+2*3)): 7
evaluate_infix(10-((3*4)+8)/4): 5
```