Stacks and Queues

Assigned: Monday, October 19
Due: Thursday, October 29, before midnight
Value: 20 points (for successfully submitting a functionally correct program to your Assembla repo before the deadline). Late submissions will receive a 0.

Executive Summary

In this lab, you will learn about the data structures stack and queue, and implement your own ones. Then you'll use your stack to write a postfix calculator, and use your queue to implement a log file management program.

Purpose

- Increase your comfort level with dynamic allocation and linked data structures
- Introduce data structures stack and queue
- Apply stack and queue to solve real world problems

Deliverables

In this assignment, you'll be designing a set of functions that other programs can use by including your .h file in them. For example, if somebody wants to use your stack implementation, they'll put assignment-5_stack.c in their source directory and include assignment-5_stack.h.

Lab_5_1

- assignment-5_stack.h
- assignment-5_stack.c
- assignment-5_calculator.h
- assignment-5_calculator.c

Lab_5_2

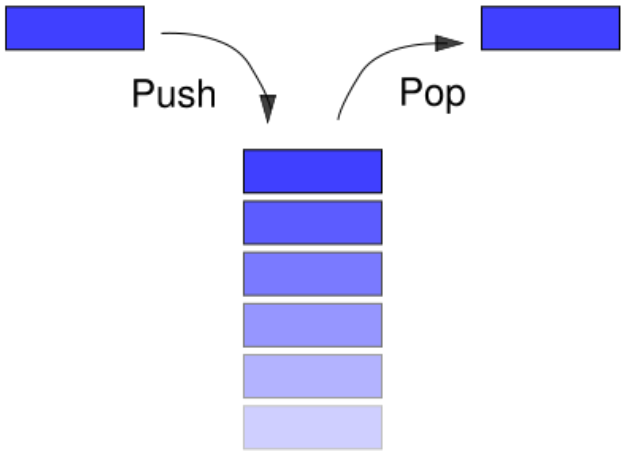
- assignment-5_queue.h
- assignment-5_queue.c
- assignment-5_logMgr.h
- assignment-5_logMgr.c

A test file have been provided to you. Passing these test cases does not mean your program is correct. You should construct your own test cases to fully test your program.

- assignment-5_test.c

Stack

Stack is a last in first out (LIFO) abstract data type and linear data structure. It has two fundamental operations *push* and *pop*.



In this assignment, your stack should hold *double* numbers. You must use a linked list design to implement your stack. Your implementation must support the operations listed below:

- initialize_stk: create a new stack *on the heap* and return a pointer to the new stack
  - e.g. Stack* my_stack = initialize_stk();
- size: return the number of elements on the specified stack
  - e.g. size(my_stack);
- push: add an element to the specified stack
  - e.g. push(my_stack, 1.5);
- pop: return and remove the last element added to the specified stack
  - e.g. double result = pop(my_stack);
- peek: simply return the last element added to the specified stack
  - e.g. double result = peek(my_stack);
- search: search the specified stack and return the position of the target, if not found, return 0. The top of the stack (last pushed) is position `size`, and the bottom of the stack (first pushed) is position 1. In the figure below, 17 is position 1; 12 is position 5.
  - e.g. int position = search(my_stack, 1.5);

- printStack: print all the elements on the specified stack
  - e.g. if the stack is like this:

printStack(my_stack); prints the stack (last element pushed should be printed first). Would print out the following line on the console (terminated with a new line): *12, 25, 123, 5, 17*

You are **<u>required</u>** to use the given *linked list* design where each node contains a *double* number and a pointer to the next node.

Postfix Calculator

Postfix notation is a mathematical notation wherein every operator follows all of its operands.

e.g. 3 4 + is the same as 4+3, which results in 7
e.g. 3 4 + 7 * is read as: 7*(3+4), which results in 49

In this part of assignment 5, you will use the stack you constructed to implement a calculator that takes command in postfix notation and output the proper result. Your calculator should support the basic arithmetic operations (addition (+), subtraction (-), multiplication (*) and division (/)).

The input commands will be given as an expression string array, each expression represents a set of data and operations that are to be executed separately. Your calculator will be used like this:

```
void main(){
     char expr1[][MAX_EXPR_LENGTH] = {“2”, “3”, “-”};
     assert( eval( expr1, 3) == 1 );
}//Pay attention to the order of the operands, the example calculates 3-2 instead of 2-3
```

You must implement the function `eval` in the file assignment-5_calculator.c

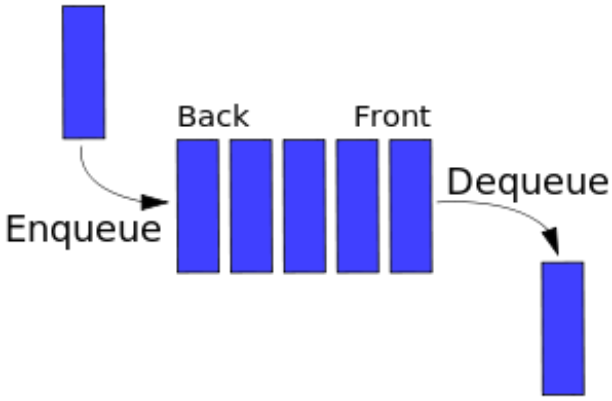
The user will pass two parameters into your calculator: an expression array and the number of elements in the expression. You can add functions as you like, but you must use your stack to implement the calculator. You should assume that after an expression is fully evaluated, there will only be one number left on the stack, and that is the result.

Implementation: Create an empty stack. Traverse the expression array, and each time you see an operand, push it on the stack. Each time you see an operator, ***pop*** two values from the stack, apply the operator on these two values, and push the result back to the stack. At the end of these operations, one element (the result of the expression), should be on the stack.

Hint: You can use the *atof* function in standard C library which parses the C string str interpreting its content as a floating point number and returns its value as a double. On success, the function returns the converted floating point number as a double value. If no valid conversion could be performed, a zero value (0.0) is returned.

Queue

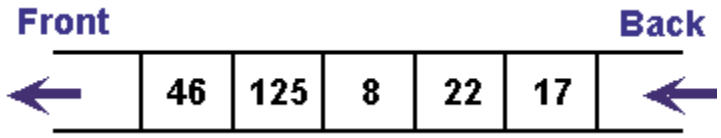
Queue is a first in first out (FIFO) abstract data type and linear data structure. The first element added to the queue will be the first one to be removed. The basic operations are *enqueue* and *dequeue*.



In this assignment, your queue should hold *long integer* values. You must use a linked list design to implement your queue. Your implementation must support the operations listed below:

- initialize_q: create a new queue *on the heap* and return a pointer to the new queue
  - e.g. Queue* my_Q = initialize_q();
- enqueue: add an element to the back of the specified queue
  - e.g. enqueue(my_Q, 3);

- dequeue: return and remove the front element in the specified queue
  - e.g. long result = dequeue(my_Q);
- insert: insert an element at a specified location; return 1 if success, 0 otherwise
  - e.g. insert(my_Q, 3, 0); //the front-most element is now 3
  - e.g. insert(my_Q, 18, 2); //the third element is now 18
- printQueue: print all the elements in the queue
  - e.g. if the queue is like this:



printQueue(my_Q) prints the queue (the first element that was enqueued should be printed first). Should output the following to the console (terminated with a new line): *46, 125, 8, 22, 17*

You are **<u>required</u>** to use the *linked list* design where each node contains a *long integer* number and a pointer to the next node.

Simple Log File Manager

Assume that you are given a number of server log files. Each log file contains millions of lines. Each line contains two fields: timestamp and visited_page. Each log file is sorted ascendingly according to timestamp.

You will implement a simple log file manager that tells the user how many log entries are in the past time period. To make this lab easier, we will use long integer as timestamps, and each entry will only contain its timestamp.

Your manager is composed of a queue and a set of functions:

- *createLogMgr* function that takes a time period and returns a pointer to the log manager;
- *add_to_log* function that takes a pointer to the manager, and a timestamp *T*. It adds an entry with timestamp=*T* to the queue, and returns the number of entries within the time period ending at *T,* i.e., the number of entries that have a timestamp between *T-WINDOW_SIZE* (inclusive) and *T* (inclusive).

Here's how your manager will be tested:

```
#define WINDOW_SIZE 2          //how long the time period is
int main() {
 log_manager *log = createLogMgr( WINDOW_SIZE );
 assert( add_to_log( log, 1 ) == 1 );         //current timestamp is 1
```

```
    assert( add_to_log( log, 2 ) == 2 );        //current timestamp is 2, 2-1 <= 2
    assert( add_to_log( log, 3 ) == 3 );        //current timestamp is 3, 3-1 <= 2
    assert( add_to_log( log, 4 ) == 3 );        //current timestamp is 4, 4-1 > 2,
                                                //the first entry with timestamp=1 is evicted from the
                                                queue. Since 4-2>=2 the second entry with timestamp=2
                                                is safe.
    assert( add_to_log( log, 4 ) == 4 );        //current timestamp is 4, 4-2<=2, so add this entry to log
    assert( add_to_log( log, 10 ) == 1 );       //current timestamp is 10, 10-2=8
                                                //so all entries whose timestamp is smaller than 8 are
                                                evicted from the queue
}
```

Requirements

Setup

Either use the provided solution project that pushed to your repo, or follow these steps to manually create the project:

1. Create a Project named `Lab5` in Visual Studio, following the same setup as in previous labs.
2. Download the starter files, **place them under the Lab5 folder that was created**
3. From the Solution Explorer, add all the downloaded files to your project, so your file structure will be `<repo>/Lab5/<individual .c or .h files>`

Implementation Notes

1. Implementation of the stack, heap, calculator, and log manager functions should go in the appropriate .c file. You must implement all of these functions. You cannot change the name of these functions, or their parameters or return type.
2. You should use the struct declaration provided in the header files. They can potentially make your life easier.
3. You should safeguard your data structures. Popping an empty stack and dequeueing an empty queue should not be allowed. You can insert element to the front and back of the queue, but “inserting to position 5” on a queue of size 2 does not make sense and should not be allowed.
4. Your stack and queue are expected to grow very large, so arrange your design accordingly. All the operations should finish within a reasonable time period.
5. Since you’ll be calling *malloc* to allocate memory, you must call *free* when the memory you allocated is no longer in use. Your program should not cause any memory leak.
6. Make sure you remove extraneous printf statements from your code, that were used for debugging purposes, before you submit. Aside from the printStack and printQueue functions, your functions should not be outputting anything to the standard output.

Grading

Your grade is spread evenly over the following parts

1. Stack implementation
2. Queue implementation
3. Stack Calculator
4. Log Manager

FAQs

1. **What is malloc and how do I invoke it?**
   Malloc is used to dynamically allocate memory (also referred to as heap memory)
   ```
   /* Allocate space for an array with ten elements of type int. Some programmers place an optional "(int *)" cast before malloc. */
   int * array = malloc(10 * sizeof(int));
   ```

2. **How do I create a new stack on the heap?**
   You first need to allocate space: call malloc and pass to it the size of a Stack object.
   ```
   Stack* stk = malloc(sizeof(Stack));
   ```

   You must then initialize the components of the created object appropriately. For example, to set the size to zero, you can do:

   `(*stk).size = 0;` [dereference the pointer, then access the member 'size']

   An equivalent shorthand syntax is `stk->size = 0;`

3. **Why is there an error in Visual Studio when I invoke malloc?**
   Don't worry about it being underlined with red. The compiler will correctly cast the return void* from malloc to the type being assigned.

3. **How do I check if a pointer ptr is not NULL? Dereferencing it doesn't work**
   If you try to check **(*ptr!=0)** , it may not work depending on the type of object that ptr is pointing to. The simplest way is to check **if(!ptr)**

4. Can I assume that for the log_Mgr subsequent timestamps will not be less than the current timestamp?
   Yes

5. **Can I assume that the input to the postfix calculator input will be well-formed?**
   Yes, you do not need to do any error checking for the postfix input. Also, you do not need to worry about division by 0.