CSCI 2270 – Data Structures

Instructor: Asa Ashraf

Assignment 5

Due: Sunday, October 3 2021, 11:59PM

Assignment 5 - Stacks and Queues

OBJECTIVES

1. **Create, add to, delete from, and work with a stack implemented as a linked list**
2. **Create, add to, delete from, and work with a queue implemented as an array**

Overview

Stacks and Queues are both data structures that can be implemented using either an array or a linked list. You will gain practice with each of these in the following two mini-projects. The first is to build a simple sentence rearranger using a linked list based stack and the second is to simulate a waitlist using a circular array based queue.

SentenceReverser

In this section, we will implement a linked list based stack. We will use this stack implementation to parse and unravel nested sentences and produce them as an output. We know that stacks are great for reversing strings, and parsing nested structures as you may have witnessed with previous week's recitation (gold problem) - balanced parentheses.

Your task will be to implement the stack using a linked list and an accompanying driver file to test your implementation.

How to Evaluate

The input will be nested sentences which are separated by closing and opening square brackets. We will evaluate sentences by pushing each word onto a stack until we encounter a "]" bracket. After we get a "]" bracket we subsequently pop from the stack until we get the first "[". This marks the end of one sentence.

We then print all the sentences encountered by push and pop operation along with the count of words in the sentence. We must also evaluate whether every "[" is matched with a "]" bracket. Else the output is appended with "**err: Invalid String**".

You may think of the provided sentence as a nested representation. The idea will be to start at the deepest nested level and process them first. For sentences on the same level, you should start by processing from bottom to top.

Consider illustration below:

```
|-- AB
|    |
|-- CD
|    |
|    |-- FJ
|    |    |
|    |-- KL
|    |    |
|    |    |-- M
|    |    |
|    |    |-- NO
```

Input: "[AB CD [FJ KL [M NO]]]" .

You will need to store each word/token as a separate string element in a temporary string array.

[	AB	CD	[	FJ	KL	[	M	NO	]	]	]
0	1	2	3	4	5	6	7	8	9	10	11

Then, within the **evaluate()** method, you must iterate through each element in the array from the beginning and follow the step to evaluate, as mentioned above and depicted below.

Instruction	Stack
[| AB | CD | [| FJ | KL | [| M | NO |] |] |] 0	[Push [
[| AB | CD | [| FJ | KL | [| M | NO |] |] |] 1	AB [Push AB

CSCI 2270 – Data Structures (Fall 2021)

Instructor: Asa Ashraf

[| AB | CD | [| FJ | KL | [| M | NO |] |] |] 2	CD AB [Push CD
[| AB | CD | [| FJ | KL | [| M | NO |] |] |] 3	[CD AB [Push [
[| AB | CD | [| FJ | KL | [| M | NO |] |] |] 4	FJ [CD AB [Push FJ
[| AB | CD | [| FJ | KL | [| M | NO |] |] |] 5	KL FJ [CD AB

[

Push KL

[	AB	CD	[	FJ	KL	[	M	NO	]	]	]

6

[
KL
FJ
[
CD
AB
[

Push [

[	AB	CD	[	FJ	KL	[	M	NO	]	]	]

7

M
[
KL
FJ
[
CD
AB
[

Push B

[	AB	CD	[	FJ	KL	[	M	NO	]	]	]
								8			

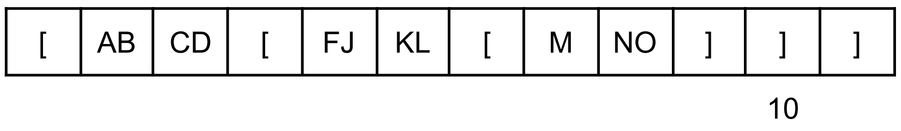
NO
M
[
KL
FJ
[
CD
AB
[

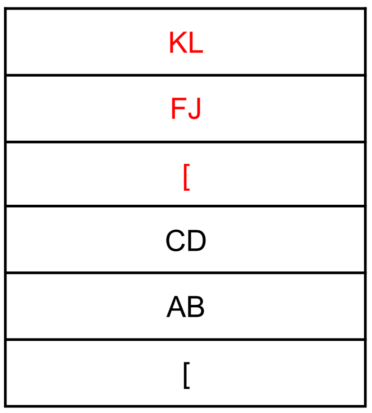
Push NO

[	AB	CD	[	FJ	KL	[	M	NO	]	]	]
									9		

NO
M
[
KL
FJ
[
CD
AB
[

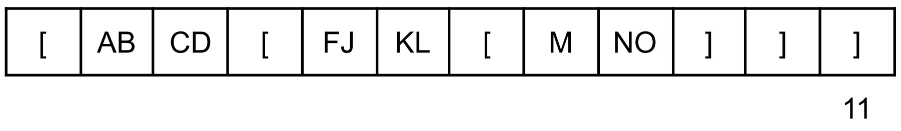
Pop from stack till you encounter ‘[’. Popped elements will be NO, M. Pop ‘[‘ too.
Result: "NO M"

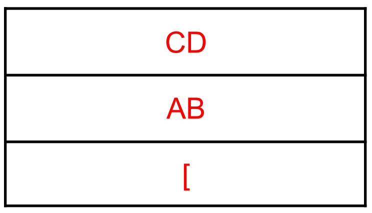




Pop from stack till you encounter ‘[’. Popped elements will be KL FJ. Pop ‘[‘ too.

Result: "KL FJ"





Pop from stack till you encounter ‘[’. Popped elements will be CD AB. Pop ‘[‘ too.

Result: "CD AB"

So, the final output will be:

```
NO M - 2
KL FJ - 2
CD AB - 2
```

Here, the format is: `<sentence> - <word count>`

Refer to Appendix 2 for another example.

Class Specifications

The node structure for the linked list is defined in SentenceReverser.hpp.

```
struct Word
{
    string word; // the string to store
    Word* next; // the pointer to the next node
};
```

The SentenceReverser class definition is provided in the file SentenceReverser.*hpp* in Canvas. *Do not modify this file or your code won't work on Coderunner!* Fill in the file SentenceReverser.*cpp* according to the following specifications.

Word* stackHead;

- ➔ Points to the top of the stack

SentenceReverser();

- ➔ Class constructor; set the stackHead pointer to NULL

~SentenceReverser();

- ➔ Destroy the stack. Take special care not to leave any memory leaks.

bool isEmpty();

- ➔ Returns true if the stack is empty else false.

void push(string word);

- ➔ Push a word to the stack

string pop();

- ➔ Pop and return the string on the top of the stack. Do not leave any memory leaks.
- ➔ Print **"Stack empty, cannot pop a word."** if the stack is empty; return empty string.

Word* peek();

- ➔ Returns the top of the stack. Print **"Stack empty, cannot peek."** if the stack is empty.

bool evaluate(std::string* s, int size);

- ➔ This function will accept an array of strings and the length of the array. This function will evaluate the expression and will print sentences with their count as described and demonstrated in the illustration above.
- ➔ After processing the entire input array, you should also print “**err: Invalid String**” if "**[**" does not match a subsequent "**]**", i.e., the brackets are not balanced.

Driver code

- ➔ The driver code should start with printing **"Enter the sentences separated by '[' and ']'.**
- ➔ It will keep taking input from the user and will store the input in a string array. Prompt using "**#>** " to each line of the console while accepting input. *Refer to the example runs.*
- ➔ If user inputs "**end**" it will stop accepting the input and will call the evaluate function with the string array s.
- ➔ If input is empty print “**No sentences: Nothing to evaluate**”.
- ➔ Print the output as mentioned below.
- ➔ Free any unused memory

Example Runs:

1.

```
Enter the sentences separated by ‘[‘ and ‘]’.
#> [
#> one
#> sentence
#> [
#> two
#> sentence
#> ]
#> is
#> This
#> ]
#> end
sentence two - 2
This is sentence one - 4
```

2.

```
Enter the sentences separated by ‘[‘ and ‘]’.
#> [
#> [
#> one
#> sentence
#> ]
#> ]
#> end
sentence one - 2
```

3.

```
Enter the sentences separated by ‘[‘ and ‘]’.
#> [
#> one
#> [
#> by
#> the
#> end
err: Invalid String
```

4.

```
Enter the sentences separated by ‘[‘ and ‘]’.
#> [
#> one
#> [
#> by
#> ]
#> the
#> end
by - 1
err: Invalid String
```

WaitlistQueue Class

(Beware of edge cases that arise from the array being circular)

The queue data structure organizes elements in order to provide a FIFO (First In First Out) order. Often, any application that relies on asynchronous processing or buffering, will maintain a queue to process the elements. An example would be a simple course registration waitlist, where each course maintains its own waitlist of students to allocate empty seats in the course on a first come first serve basis.

In this section, you will build a queue using the **circular array implementation** to emulate a simple course registration waitlist. Implement the methods of **WaitlistQueue** according to the following specifications.

DATA

std::string* queue

➔ A dynamically allocated array of strings to act as a container for the circular queue. You will use **qCapacity** to determine the size of this container.

int qCapacity

➔ The total capacity of the dynamically allocated array, queue. This value will be passed in a parameterized constructor as an argument.

int qFront

➔ Index in the array that keeps track of the index at which dequeue will happen next, i.e., it refers to the front (or first) element in the queue.

int qEnd

➔ Index in the array that keeps track of the tail element in the queue.

FUNCTIONS

WaitlistQueue(int qSize=10)

➔ Constructor. Initialize **qFront, qEnd** to **-1** and **qCapacity** to **qSize.** You should, then, allocate memory for the **queue** container according to **qSize**.

~WaitlistQueue()

➔ Destructor. Deallocate the memory for **queue** container.

void enqueue(std::string value)

➔ If the queue is not full, then add the **value** to the end of the queue and modify **qFront** and/or **qEnd** appropriately, else print "*Waitlist is full. Cannot enqueue.*"

void dequeue()

- ➔ Remove the first element from the queue if the queue is not empty and modify **queueFront** and/or **queueEnd** appropriately. Otherwise, print *"Waitlist is empty. Cannot dequeue."*

- ➔ Take care of the edge case where there is only 1 element in the queue to be dequeued. In such a case, you should appropriately reset **qFront** and **qEnd**. *(Check sample runs)*

std::string peek()
- ➔ If the queue is empty then return "*<EMPTY QUEUE>*". Otherwise, return the first element in the queue.

int size()
- ➔ Return the number of active elements in the queue. Not to be confused with **qCapacity**.

int capacity()
- ➔ Return the capacity of the queue container, i.e., **qCapacity**.

bool isEmpty()
- ➔ Return true if the queue is empty, false otherwise.

bool isFull()
- ➔ Return true if the queue is full, false otherwise.

void printInfo()
- ➔ This function has been implemented for you and is meant for debugging and grading purposes.
- ➔ This function prints the results of **peek()**, **qFront**, **qEnd**.

void resize(int newSize)
- ➔ Resize the queue container to this **newSize** (>= 1). You must start at the front of the original **queue** and copy elements till the end of the original **queue**, or until the new queue is full.

- ➔ You must update **queue**, **qFront**, **qEnd** and **qCapacity** appropriately. Take care of any possible memory leaks.

buff05	buff10	buff25	buff30
qEnd		*qFront*	

After **resize(6)**:

buff25	buff30	buff05			
qFront		*qEnd*			

Below are a few sample runs for WaitlistQueue.cpp:

PROGRAM main()	**OUTPUT**

`WaitlistQueue q(5);`
 	
`q.enqueue("buff_24");`
`cout << q.peek() << endl;`
 	
`q.enqueue("buff_25");`
`q.enqueue("buff_26");`
`q.printInfo();`
 	
`q.dequeue();`
`q.dequeue();`
`q.printInfo();`
	`buff_24`
	`peek()=buff_24, qFront=0, qEnd=2`
	`peek()=buff_26, qFront=2, qEnd=2`

PROGRAM main()	**OUTPUT**

`WaitlistQueue q(5);`
 	
`q.enqueue("buff_24");`
`q.enqueue("buff_25");`
`q.enqueue("buff_26");`
`q.printInfo();`
 	
`q.dequeue();`
`q.dequeue();`
`q.printInfo();`
 	
`q.dequeue();`
`q.printInfo();`
	`peek()=buff_24, qFront=0, qEnd=2`
	`peek()=buff_26, qFront=2, qEnd=2`
	`peek()=<EMPTY QUEUE>, qFront=-1, qEnd=-1`

<u>WaitlistDriver</u>

In this section, you will write an interactive driver to work with your queue. The specifications are as follows:

- **INITIALIZATION**: Read the initial queue size as a command-line argument. Your program will receive exactly 1 command-line argument (apart from the filename), the initial size of the queue (int).
  - Print the following message after instantiating the queue:

    ```
    Initial queue capacity = <capacity>
    ```

- **OPTIONS**: The user will input the following options:
  - **`enqueue`**
    - Request the user for the value to be enqueued
    - Call the **`enqueue()`** method on this value

      ```
      #> enqueue buff01
      ```

  - **`dequeue`**
    - Simply call the **`dequeue()`** method
  - **`peek`**
    - Simply print the output of the **`peek()`** method.

      ```
      #> peek
      buff01
      ```

  - **`resize`**
    - Request the user for the new size of the queue
    - Call the **`resize()`** function after printing the following message:

      ```
      #> resize <new_capacity>
      Resizing from <old_capacity> to <new_capacity>
      ```

  - **`quit`**
    - Prior to exiting the program, print the queue from front to back separated by a space. *Example*:

      ```
      Contents of the queue:
      buff01 buff03 buff05
      ```

    - Exit the program.

- **AFTER EACH OPTION**: You must also call **`printInfo()`** method after each option's execution.
- **TIP**: We recommend using the stream extraction operator (**>>**) rather than the getline() method to get input.

Below are a few sample runs for the Driver:

```
./Q2Driver 5
```

```
Initial queue capacity = 5
#> enqueue 2
peek()=2, qFront=0, qEnd=0
#> enqueue 3
peek()=2, qFront=0, qEnd=1
#> enqueue 4
peek()=2, qFront=0, qEnd=2
#> enqueue 5
peek()=2, qFront=0, qEnd=3
#> dequeue
peek()=3, qFront=1, qEnd=3
#> dequeue
peek()=4, qFront=2, qEnd=3
#> enqueue 6
peek()=4, qFront=2, qEnd=4
#> enqueue 7
peek()=4, qFront=2, qEnd=0
#> dequeue
peek()=5, qFront=3, qEnd=0
#> quit
peek()=5, qFront=3, qEnd=0
Contents of the queue:
5 6 7
```

```
./Q2Driver 2
```

```
Initial queue capacity = 2
#> enqueue 10
peek()=10, qFront=0, qEnd=0
#> enqueue 20
peek()=10, qFront=0, qEnd=1
#> dequeue
peek()=20, qFront=1, qEnd=1
```

```
#> enqueue 30
peek()=20, qFront=1, qEnd=0
#> enqueue 40
Waitlist is full. Cannot enqueue.
peek()=20, qFront=1, qEnd=0
#> resize 5
Resizing from 2 to 5
peek()=20, qFront=0, qEnd=1
#> enqueue 40
peek()=20, qFront=0, qEnd=2
#> enqueue 50
peek()=20, qFront=0, qEnd=3
#> peek
20
peek()=20, qFront=0, qEnd=3
#> quit
peek()=20, qFront=0, qEnd=3
Contents of the queue:
20 30 40 50
```

APPENDIX 1: Debugging hints

Compilation for gdb:

```
g++ -std=c++11 -g WaitlistDriver.cpp -o Q2Driver
```

Run using valgrind:

```
valgrind --leak-check=yes  ./Q2Driver
```

APPENDIX 2: Example for Q1

For question 1, here is another illustration:

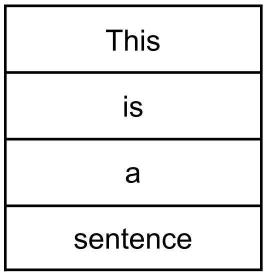
```
sentence another yet is This
|
|-- string another by separated
|   |
|   |-- sentence a is This
|   |
|-- sentence another is This
```

Input: [sentence another yet is This [string another by separated [sentence a is This] sentence another is This]]

Output:

```
This is a sentence - 4
This is another sentence separated by another string - 8
This is yet another sentence - 5
```

Push each word onto the stack until you encounter **]**

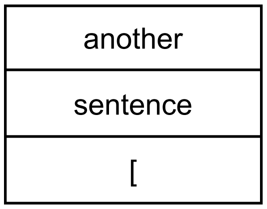


[
separated
by
another
string
[
This
is
yet
another
sentence
[

Start popping until you get **[.** Pop the **[**, as well.

Stack becomes:

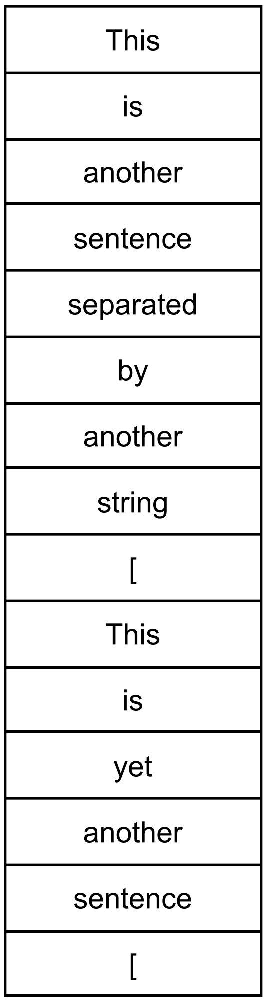
separated
by
another
string
[
This
is
yet



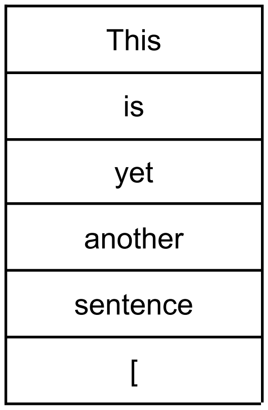
<u>Output</u>: `This is a sentence - 4`

<u>Explanation</u>**:** We popped until we hit the first opening square brackets. We appended to a temporary string and tracked the number of peek & pop instructions.

Now, we continue our process, and push words until we hit a closing **].**



Now, we come across a closing **].** Therefore we peek and pop to get this sentence, resulting in:



Output: **`This is another sentence separated by another string - 8`**

Now, we continue our process, and we encounter another closing **]**. Thus, we continue to peek and pop until we hit the opening **[**.

Output: **`This is yet another sentence - 5`**

Overall the complete output will be:

```
This is a sentence - 4
This is another sentence separated by another string - 8
This is yet another sentence - 5
```