



CSCI 2270 – Data Structures

Instructors: Zagrodzki, Ashraf

Assignment 5

Due Sunday, October 11 2020, 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 calculator using a linked list stack and the second is to simulate a job scheduling system using a circular array based queue.

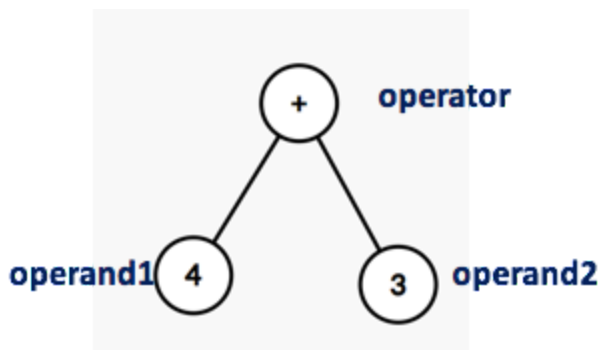
Stack calculator

In this assignment we will implement a linked list based stack. We will use this stack implementation to evaluate prefix expressions. In the following section we will discuss what is a prefix expression and how we can evaluate them using stack. Your task will be to implement the logic into c++ code. **We will consider only two binary operators “+” and “*” for prefix notations.**

Prefix Notation

In general the way we express a binary operator in mathematical expression is infix. For example “4 + 3”. In this expression the order is “<operand1> operator <operand2>”. The prefix notation for the will be “+ 4 3”. The order here is “operator <operand1> <operand2>”.

A visual representation of the operation is given here. Note though “4 + 3” and “ + 4 3” are different strings but both of them should be evaluated to 7.



Even if the computation tree is deeper we can follow the same paradigm to represent the expression. For example consider the expression-

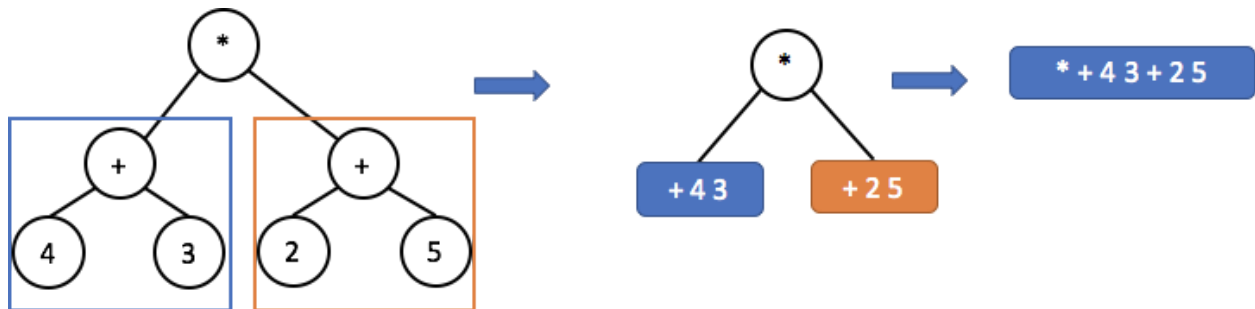


CSCI 2270 – Data Structures Fall 2020

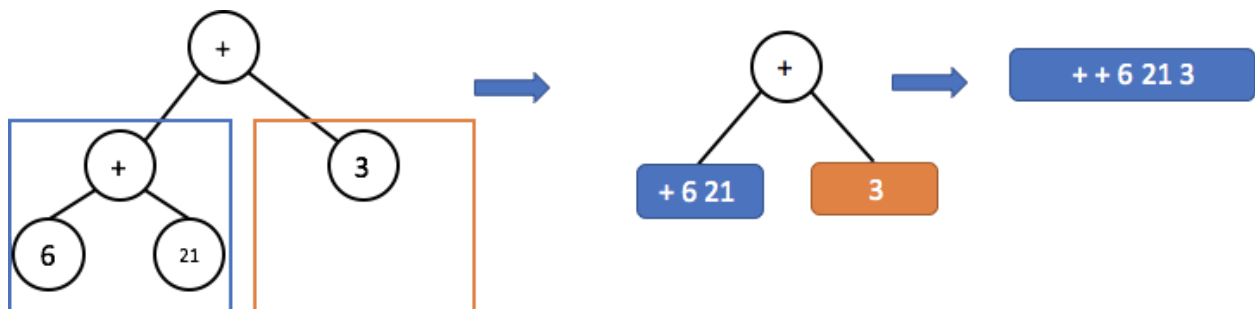
Instructors: Zagrodzki, Ashraf

“(4+3) + (2+5)”.

The computation tree of the same is given below. For each subexpression we will follow the same procedure “**operator <operand1> <operand2>**”. So the subexpression “4 + 3” will be “+ 4 3” and “2 + 5” will be “+ 2 5”. Now these two subexpressions “+ 4 3” and “+ 2 5” are the operands for “*”. So the final expression become “* + 4 3 + 2 5”.



Again note the expression “* + 4 3 + 2 5” evaluates to be 49 which is the same as its infix counterpart “(4+3) + (2+5)”. Here is another example for (6 + 21) + 3



How to Evaluate

To evaluate a prefix expression we will use stack. **We will parse the expression from the back.** If we encounter a number we will push it to the stack. If we encounter an operator (“+” or “*”) we will take two elements from the stack top and perform the operation. We will push back the result again in the stack. To determine whether an element is operator or operand we will



CSCI 2270 – Data Structures Fall 2020

Instructors: Zagrodzki, Ashraf

use **isNumber** function. Note this function is provided in the cpp file so you do not need to implement **isNumber** function.

Consider the example “ * + 4 3 + 2 5”. Our code will accept the notation as an array of strings-



*	+	4	3	+	2	5
Index:0	1	2	3	4	5	6

Instruction	Stack
Index: 0 1 2 3 4 5 6 * + 4 3 + 2 5 ↑	5 Push 5
Index: 0 1 2 3 4 5 6 * + 4 3 + 2 5 ↑	2 5 Push 2
Index: 0 1 2 3 4 5 6 * + 4 3 + 2 5 ↑	7 Pop two elements. Perform the operation '+'. Push the result.
Index: 0 1 2 3 4 5 6 * + 4 3 + 2 5 ↑	3 7 Push 3
Index: 0 1 2 3 4 5 6 * + 4 3 + 2 5 ↑	4 3 7



CSCI 2270 – Data Structures Fall 2020

Instructors: Zagrodzki, Ashraf

	Push 4									
Index: 0 1 2 3 4 5 6 <table><tr><td>*</td><td>+</td><td>4</td><td>3</td><td>+</td><td>2</td><td>5</td></tr></table> 	*	+	4	3	+	2	5	<table><tr><td>7</td></tr><tr><td>7</td></tr></table> <p>Pop two elements. Perform the operation +. Push the result.</p>	7	7
*	+	4	3	+	2	5				
7										
7										
Index: 0 1 2 3 4 5 6 <table><tr><td>*</td><td>+</td><td>4</td><td>3</td><td>+</td><td>2</td><td>5</td></tr></table> 	*	+	4	3	+	2	5	<table><tr><td>49</td></tr></table> <p>Pop two elements. Perform the operation *. Push the result.</p>	49	
*	+	4	3	+	2	5				
49										

Class Specifications

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

```
struct Operand
{
    float number; // the number to store
    Operand* next; // the pointer to the next node
};
```

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

Operand* stackHead;

→ Points to the top of the stack

StackCalculator();

→ Class constructor; set the stackHead pointer to NULL

~StackCalculator();

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

bool isEmpty();

→ Returns true if the stack is empty



CSCI 2270 – Data Structures Fall 2020

Instructors: Zagrodzki, Ashraf

void push(float num);

→ Push a float element to the stack

void pop();

→ Pop the element out of the stack. Do not leave any memory leak. Print **"Stack empty, cannot pop a job."** if the stack is empty.

Operand* peek();

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

Operand* getStackHead() { return stackHead; }

→ Returns the stackHead. **This is already implemented.**

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

→ This function will accept an array of strings and the length of the array. The array represents the prefix expression. This function will evaluate the expression and will store the ultimate result in the stack.

→ If the function encounters any other operator than "+" or "*" it should print **"err: invalid operation"** and return false.

→ If the array does not have the correct number of operands it should print **"err: not enough operands"** and return false.

→ If everything goes well it will return true.

Driver code

The driver code should start with printing **"Enter the operators and operands ('+', '*') in a prefix format"**

→ It will keep taking input from the user and will store the input to a string array. Print **"#> "** to each line of the console while accepting input. Refer to the example run.

→ If user inputs **"="** it will stop accepting the input and will call the evaluate function with the string array.

→ If the expression is not valid it should print **"Invalid expression"**

→ For valid expression it should print the result with **"Result= "**

→ Free any unused memory

Example Runs:

1.

```
Enter the operators and operands ('+', '*') in a prefix format
#> *
#> +
```



CSCI 2270 – Data Structures Fall 2020

Instructors: Zagrodzki, Ashraf

```
#> 4
#> 3
#> +
#> 2
#> 5
#> =
Result= 49
```

2.

```
Enter the operators and operands ('+', '*') in a prefix format
#> +
#> +
#> 14
#> 12
#> 3
#> =
Result= 29
```

3.

```
Enter the operators and operands ('+', '*') in a prefix format
#> *
#> +
#> 4
#> 3
#> =
err: not enough operands
```

4.

```
Enter the operators and operands ('+', '*') in a prefix format
#> *
#> 12
#> -
#> 20
#> 10
#> =
err: invalid operation
```



CSCI 2270 – Data Structures Fall 2020

Instructors: Zagrodzki, Ashraf

5.

```
Enter the operators and operands ('+', '*') in a prefix format
#> *
#> +
#> 4
#> 3
#> 2
#> 5
#> =
Invalid expression
```

JobQueue Class

****Beware of edge cases that arise from the array being circular****

In this class you will build a queue using the circular array implementation. Implement the methods of **JobQueue** according to the following specifications.

std::string queue[SIZE]

→ A circular queue of strings in the form of an array. **SIZE** is initialized to a default value of 20 in *JobQueue.hpp*

int queueFront

→ Index in the array that keeps track of the index at which dequeue will happen next

int queueEnd

→ Index in the array that keeps track of the first available empty space (in case the queue is full, queueEnd points to queueFront).

JobQueue()

→ Constructor--Set **queueFront**, **queueEnd** and **counter** to 0

bool isEmpty()

→ Return true if the queue is empty, false otherwise

bool isFull()

→ Return true if the queue is full, false otherwise

void enqueue(std::string job)



CSCI 2270 – Data Structures Fall 2020

Instructors: Zagrodzki, Ashraf

- If the queue is not full, then add the **job** to the end of the queue and modify **queueFront** and/or **queueEnd** if appropriate, else print "Queue full, cannot add new job"

void dequeue()

- Remove the first job from the queue if the queue is not empty and modify **queueFront** and/or **queueEnd** if appropriate. Otherwise print "Queue empty, cannot dequeue a job"

int queueSize()

- Return the number of jobs in the queue.

std::string peek()

- If the queue is empty then print "Queue empty, cannot peek" and return an empty string. Otherwise, return the first job in the queue.

JobQueue main driver file

Your program will start by displaying a menu by calling the **menu()** function which is included in the provided skeleton code. The user will select an option from the menu to decide upon what the program will do, after which the menu will be displayed again. Below are the specifics of the menu (*Tips: use **getline** for all inputs, and you are required to create and write the main function. the example is shown in the starter code.*)

Option 1: Add jobs into the queue - This is an enqueue operation

- This option prompts the user to enter the number of jobs being created using the below print statement

```
cout << "Enter the number of jobs to be created:" << endl;
```

- Then prompt the user to enter each job using the below print statement

```
cout << "job" << <JOB_NO> << ":" << endl;
```

- Create (**enqueue**) all the jobs into the queue

Option 2: Dispatch jobs from the queue - This is a dequeue operation

- This option prompts the user to enter the number of jobs being dispatched using the below print statement

```
cout << "Enter the number of jobs to be dispatched:" << endl;
```




CSCI 2270 – Data Structures Fall 2020

Instructors: Zagrodzki, Ashraf

- Dispatch (**dequeue**) jobs from the queue. If the number of jobs to be dispatched is greater than the total number of jobs in the queue, then dispatch (**dequeue**) all the available jobs in the queue and notify the user with a below statement

```
cout<< "No more jobs to dispatch from queue" << endl;
```

This message should never be printed more than once.

- For each job dispatched, print the following

```
cout << "Dispatched: " << job << endl;"
```

where **job** is the string you are dequeuing.

Option 3: Return the queue size and exit

- This option prompts the user to return the number of jobs in the queue using the below print statement, before exiting the program.

```
cout << "Number of jobs in the queue:" <job_count>;
```

Below is a sample output

```
*-----*
Choose an option:
1. Add jobs into the queue
2. Dispatch jobs from the queue
3. Return the queue size and exit
*-----*
1
Enter the number of jobs to be created:
3
job1:
Sort
job2:
Save
job3:
Load
*-----*
Choose an option:
1. Add jobs into the queue
```



CSCI 2270 – Data Structures Fall 2020

Instructors: Zagrodzki, Ashraf

```
2. Dispatch jobs from the queue
3. Return the queue size and exit
*-----*
2
Enter the number of jobs to be dispatched:
2
Dispatched: Sort
Dispatched: Save
*-----*
Choose an option:
1. Add jobs into the queue
2. Dispatch jobs from the queue
3. Return the queue size and exit
*-----*
3
Number of jobs in the queue: 1
```