Notes on Assignment 5

CSCI 2270: Data Structures

Section: 202
TA: Sanskar Katiyar

Section: 305
TA: Varad Deshmukh

RPNCalculator.cpp

In this problem we are going to implement a Reverse Polish Notation Calculator (RPNCalculator). RPN avoids ambiguity in arithmetic operations. The notation we generally use is called infix notation, in which we fix ambiguous order of operations using parentheses. (Check the example on writeup)

We can use a stack to evaluate a RPN expression. In order to do that, we will need to implement the following methods in **`RPNCalculator.cpp`**

`RPNCalculator();`
`~RPNCalculator();`
`bool isEmpty();`
`void push(float num);`
`void pop();`
`Operand* peek();`
`bool compute(std::string symbol);`

You are also provided with an Operand data type already implemented. Compare this structure to the Node data type we implemented previously for linked list.

Some tips to approach the problem

1. Recall that implementing a stack using a linked list is simply a variation on regular linked list operations:
    a. Push: Insertion only at head in a LL
    b. Pop: Deletion only at head in a LL
    c. Peek: Return contents of the head node in a LL

2. Use the class access specifier before each function name. Thus your functions should look like `void RPNCalculator::push(float num)`.

3. Include the **`RPNCalculator.hpp`** file in this source file.

4. Take note of the arguments and return specifications for each function.

5. Refer to recitation code examples on StackLL, Linked List.

RPNCalculator()

The constructor sets the **`stackHead`** to NULL, which is, essentially just a single statement. It starts the stack as empty.

~RPNCalculator()

The destructor frees the memory allocated to the stack. This is similar to how we delete a linked list. In a linked list, we just delete the **`head`** node, one at a time, until the **`head`** is NULL. [Example, Slide #53]

We already have a member function which deletes one node at a time: ***which one?***

bool isEmpty()

How do you check if a linked list is empty? `[head == NULL]`

What is the equivalent of **`head`** ***here?*** `[stackHead]`

Remember to return the correct boolean value based on the result.

void push(float num)

This operation is equivalent to inserting a node at head in a linked list. Create a new node of type **`Operand`** with the passed argument `num`. [Example, Slide #32].

void pop()

This operation is equivalent to deleting the head node in a linked list. [Example, Slide #37]

Check if the stack is empty prior to deletion. If the stack is empty, remember to print the required message.

Operand peek()*

Return the contents (here: node itself) at the top (here: stackHead) of the stack. [Example, Slide #43]

bool compute(std::string symbol)

While you should follow the instructions in the writeup, here is a sequential template if you are having trouble starting with this solution:

1. **Check if the stack is empty.**
   If it is, then print an error message and conclude the execution of the function by returning false. Use one of the member functions you have already implemented.

2. **Check if the symbol is valid or not.**
   Remember to compare the symbol using strings (“”) and not characters (‘’).

   If symbol is not among the valid operators, then print an error message and conclude the execution of the function by returning false.

3. **Read the first operand**
   If you reach this stage, this implies that there is at least one operand on the stack. However, we cannot be certain about a second operand.

   In order to check whether there exists a second operand, we will need to pop the first operand. But before we pop the top operand we must store it in a temporary variable since popping is equivalent to deletion. And we will need to utilize this operand later.

   Use **`peek()->number`** to get the floating number stored in the top node. And then delete the top node.

4. **Check if the stack is empty.**
   If it is, then there is just one operand in the stack. We can’t utilize this. Thus, push the previously read floating number back to the stack.

   Then print an error message and conclude the execution of the function by returning false.

5. **Read the second operand**
   If we reach this stage then we are certain that there exists a valid operation and two valid operands on the stack. Thus we can go through with our execution of the function.

   Similar to the first operand, use **`peek()->number`** to get the floating number stored in the top node. And then delete the top node.

6. **Return the computation**

Finally, based on the symbol passed (if statement) and the read operands (in step 3, 5) compute the correct operation and push the result back to the stack.

Remember to return true to mark successful execution of the operation.

RPNCalculatorDriver.cpp

The starter code should give you some hints about what you ought to do.

Start by creating a RPNCalculator object. This will be your stack. After that follow the instructions mentioned as TODOs in the starter code.

Remember you can now use only the implemented methods in the RPNCalculator class to modify the stack: pop, push, peek, isEmpty, compute.

This while loop is responsible for reading one operand/operator at a time.

1. If you read "=" then you should come out of the loop.
2. If you read an operand (utilize the **`isNumber(x)`** function, already implemented) then push it onto the stack.
3. Otherwise, you can assume it is an operator. Thus you should invoke the compute() method.

Once you are out of the loop, you must check the status of the stack for the validity of the entire operation.

1. If the stack is empty, print the required error message and conclude the program.
2. *A valid stack should have just one resultant operand.*
   Thus, use the peek, pop operations in correct order to read the top element and then check if the stack is empty.

   If the stack is not empty then print an error message and conclude the program. Otherwise, print the result of the operand.

ProducerConsumer.cpp

In this problem, we will be implementing a Producer-Consumer system (fancy term for a queue). Recall that the queue may be implemented either as an array or a linked list. In this program, you'll be implementing the queue of string data using a **circular array**.

`ProducerConsumer();`
`~ProducerConsumer();`
`bool isEmpty();`
`bool isFull();`
`void enqueue(string item);`
`void dequeue();`
`std::string peek();`
`int queueSize();`

Some tips to approach the problem

1. You will obviously be creating a regular array of type `string`, and creating two indices --- `queueFront` and `queueEnd` to manipulate the front and end of the queue.

2. When you are trying to enqueue something, increment `queueEnd`. If you are trying to dequeue something, increment `queueFront`. Both indices will move in the same direction (always forward).

3. When you dequeue something, don't try to delete the element from the array using the **delete** instruction. *This was a common mistake observed in the array halving question in the midterm exam.* As long as the element index is outside of the queueFront and queueEnd bounds, it is safe to assume the element is invalid.

4. What makes the array "**circular**" is what happens to queueFront or queueEnd, when the indices hit the bounds of the array. This is where you will be using the modulo operator: % (more on this later). Take a look at the [Example, Slide #63], however there is a slight implementation difference. Specifically, in what bounds queueFront and queueEnd represent.

5. Similar to the first problem, your implemented functions should be scoped using the class name -- **`void ProducerConsumer::enqueue(std::string element)`**.

6. ***Remember the roles of queueFront and queueEnd***.
    a. queueFront represents the starting position of the queue. The element at queueFront will be valid (unless queueFront = queueEnd= 0).
    b. queueEnd represents the ***next*** available slot in the queue. When you are about to enqueue, add an element to the *current position of the queueEnd,* and then increment queueEnd.

ProducerConsumer();

All you need to do here is set the *queueFront* and *queueEnd* to 0. *Take a note of the **counter** variable in the class and reason why is it needed?*

bool isEmpty();

The *counter* variable in your class definition keeps track of the number of filled entries in the queue. So the queue is empty if the *counter* equals __? Use this as a condition check to return **`True`** or **`False`**.

bool isFull();

Again, use the *counter* variable. If the queue is full, *counter* equals __? (HINT: What is the maximum size of the array?) Use this as a condition check to return **`True`** or **`False`**.

void enqueue(string item);

To add to the queue,

1. Check if the queue is already full. You can use the function **`isFull()`** and don't add anything if it returns **`True`**.
2. You can add directly to *queueEnd* otherwise (simple array insertion).
3. In a ***non-circular array*** implementation, you would increment the queueEnd by 1 after the element has been added:

queueEnd = queueEnd + 1;

- Say your *queueEnd* is 7 currently, and the array SIZE is 8 (indexed from 0 to 7). With the above addition, your *queueEnd = 8* would be accessing an invalid index in the array.

- You want your *queueEnd* to loop back to the start of the array to make use of any empty space in the beginning. So after inserting at index 7, you want queueEnd to become 0.

HINT: Make use of % operator on top of the *queueEnd* = *queueEnd* + *1;*
HINT: 8 % 8 = 0.

4. **Change counter value!** Your **`isFull`** and **`isEmpty`** use it.

void ***dequeue();***

To remove from the queue,

1. Check if the queue is empty using the **`isEmpty()`** function and don't dequeue anything if it returns **`True.`**
2. *queueFront* points to the start of the queue. To dequeue the element array[*queueFront*], modify *queueFront* appropriately. Don't forget to use % operator!
3. Don't use the delete operator!
4. Change counter value! Your **`isFull`** and **`isEmpty`** use it.

std::string ***peek();***

This should print the element at the front of the queue. Make sure that the edge cases are handled (queue is empty)!

ProducerConsumerDriver.cpp

This file implements the main function that makes calls to the ProducerConsumer functions you have implemented.

- This can be done using a **`while`**() loop that loops infinitely.
- Inside this, you want to input a choice from the user, and based on the choice, have different **if()** blocks, which either call the enqueue sequence, or the dequeue sequence or the queue size sequence. You'll need a break statement in the queue size sequence since you need to exit.
- IMPORTANT: Don't use "**`cin >>`**" to get the input, use **`getline`** everywhere!
- IMPORTANT: **`getline`** will store the data in the string. Where needed, use **`stoi()`** to convert from string to integer.

NOTE: If your program is stuck after a certain operation then take a look at your destructor. Since it might have an infinite loop. Remember, destructors are called automatically.