CMP 210 – Data Structures and Algorithms
Special Section – Spring 2023
Assignment - 02

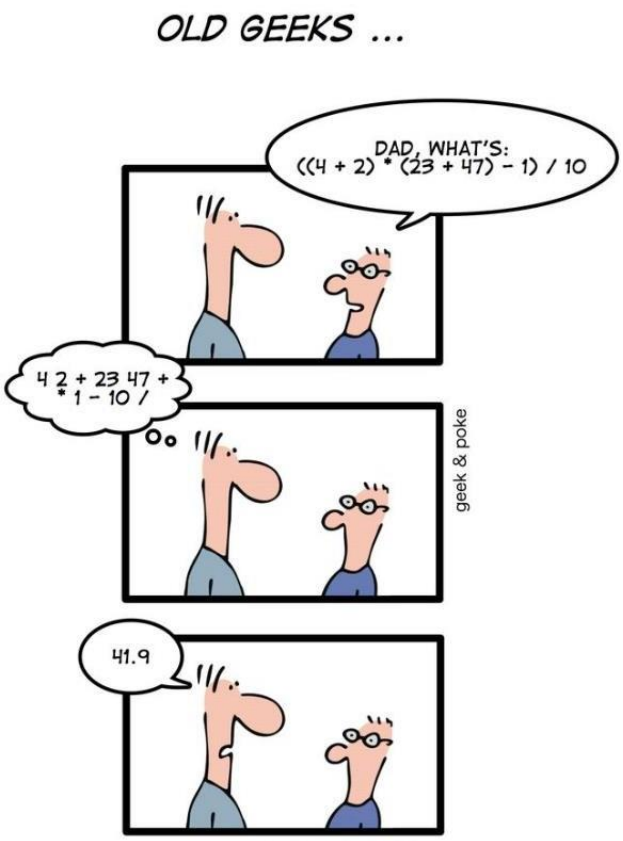
Issue Date: **Tuesday – November 21st, 2023**
Submission Deadline: **Wednesday – November 29th, 2023 (Till 12:00 am)**

Instructions!

1. You are required to do this assignment on your own. Absolutely **NO** collaboration is allowed, if you face any difficulty feel free to discuss with me.
2. Cheating will result in a **ZERO** for the assignment. (Finding solutions online is cheating, copying someone else's solution is cheating). Also, do not hand your work over to another student to read/copy. If you allow anyone to copy your work, in part or in whole, you are liable as well.
3. Hard **DEADLINE** of this assignment is **Wednesday, November 29th, 2023**. No late submissions will be accepted after due date and time so manage emergencies beforehand.

RPN Calculator: [30 Marks]

Reverse Polish notation (RPN), also known as polish postfix notation or simply postfix notation, is a mathematical notation in which operators follow their operands, in contrast to prefix or Polish notation (PN), in which operators precede their operands. The notation does not need any parentheses for as long as each operator has a fixed number of operands.



In Reverse Polish notation, the operators follow their operands. For example, to add 3 and 4 together, the expression is 3 4 + rather than 3 + 4. Initially It looks a bit weird, but it's actually pretty easy to understand and use. We have studied the stack based method in the class as well. Stacks are great for doing this job. For example, If the user enters a valid integer, you push that integer onto the stack. If the user enters a valid operator, you pop two integers off the stack, perform the requested operation, and push the result back onto the stack and so on.

Your task is to implement a basic RPN calculator that supports integer operands like 1, 64738, and -42 as well as the (binary) integer operators +, -, *, /, and %. Note that we are dealing with integers only so / stands for integer division and % stands for integer remainder. Both of these should behave just like they do in C++. Apart from the usual working of the calculator, you also have to include some extra operators to get information about results and the current state of the stack. For example, for symbol ? (that's a question mark), you should print the current state of the stack followed by a new line. For the symbol ^ (that's a caret), you should print only the top most element of the stack (not the entire stack) followed by a new line.

Note that there are a number of error conditions that your program must deal with gracefully for full credit. We'll give you two examples for free, you'll have to figure out any further error conditions for yourself:

- If the user enters *blah* or *1.5* or anything else that doesn't make sense for an integer calculator, your program should make it clear that it can't do anything helpful with that input; but it should not stop at that point.
- If the user requests an operation for which there are not enough operands on the stack, your program should notify the user of the problem but leave the stack unchanged; again, it should certainly not stop at that point.

Of course this means that you'll have to print error messages to the user. Error messages must be descriptive enough to make sense out of the errors. Furthermore, all error messages must start with the symbol # (that's a hash sign) and be followed by a new line! Remember to write a proper main program making it easy to test the functionality of your RPN calculator. No marks shall be given without this driver program.

FIFO Page Replacement Algorithm [20 Marks]

In an operating system, *paging* is a method to manage the computer's memory more efficiently. Think of it like organizing a big book into smaller pages. Each page has a specific size, and the computer's memory is divided into these fixed-size pages. Here's how it works: When a program or application runs on your computer, it needs space in memory to store its data and instructions. Instead of finding one large, continuous space in memory (which can be tricky), paging divides memory into these manageable pages. Each page is like a little box that can hold a certain amount of data. When a program needs memory, the operating system assigns it one or more of these pages. If a program needs more space, it gets more pages. Paging makes memory management more flexible and efficient. It's like having lots of sticky notes (pages) that can be moved around in a notebook (memory) to accommodate different things you're working on. This way, the computer can use its memory more effectively and keep everything organized. In simple terms, paging in an operating system is like using sticky notes to organize and manage the computer's memory, making it easier to handle multiple tasks and programs.

Here's how it works:
When a new page is requested, and it's not already in the computer's memory, we have what's called a "page fault." To handle this, the computer's operating system replaces one of the pages that's already in memory with the new one. Different page replacement algorithms suggest different ways to decide which page should be replaced. But they all share a common goal: to reduce the number of page faults, which slow down the computer.

One of the simplest page replacement algorithms is called "First In First Out" or FIFO. In this algorithm, the operating system keeps track of all the pages currently in memory in a queue, which is like a line of pages waiting their turn. The oldest page, the one that came in first, is at the front of the queue. When a new page needs to be loaded, the operating system selects the page at the front of the queue to be replaced. Here's an example: Let's say we have a sequence of page references: 1, 3, 0, 3, 5, 6, and we have space for 3 pages in memory. At first, all the memory slots are empty. So, when pages 1, 3, and 0 are requested by a program they come in, they are put into the empty slots, resulting in 3 page faults. When again page 3 is requested by a program, it's already in memory, so there's no page fault, hence it results into a page hit. But then, page 5 is requested by a program, and it's not in memory, so operating system loads it from the hard disk and it replaces the oldest page in memory, which is page 1. This causes another page fault. Finally, page 6 is requested by the program, and since it's also not in memory, it loaded by the operating system which replaces the oldest page remaining in memory, which is page 3, causing 1 more page fault. So, in total, there were 5 page faults and one page hit in a sequence of six page requests by a program namely 1, 3, 0, 3, 5, 6 with a queue size (available memory) of three pages.

Your task is to create a program that simulates this FIFO-based page replacement algorithm. In your program, you'll ask the user for the size of the memory queue and then keep asking the user for the page numbers they want to load into the queue. You'll need to determine whether each page is a hit (already in memory) or a fault (needs to be loaded), and you'll display the queue after each operation. The program should continue until the user decides to quit, and then it should display the total number of page faults and page hits.

Sum Triangle [15 Marks]

You are given and integer array and you have to print the sum in form of a triangle. First level of triangle is going to have all the elements of array. Second level will have 1 less element and so on, and the element at that level will be the sum of two consecutives elements in previous level as show below.

Input: A = {10, 20, 30, 40, 50}

Output:

[480]

[200, 280]

[80, 120, 160]

[30, 50, 70, 90]

[10, 20, 30, 40, 50]

Binary Numbers [15 Marks]

Given a number N, your task is to use a queue data structure to print all possible binary numbers with decimal values from 1 to N. For instance, if the input is 4, the expected output would be: 1, 10, 11, 100. Here's an intriguing approach that employs a queue data structure to achieve this:

1. Begin with an empty queue of strings.
2. Add the first binary number, "1," to the queue.
3. Remove and print the front element of the queue.
4. Extend the front item by appending "0" to it and place it back in the queue.
5. Extend the front item by appending "1" to it and place it back in the queue.
6. Repeat steps 3 to 5 until you reach the desired end value.

Example Simulation of Queue:

Let's perform a dry run of the given task for N = 4 to see how it generates binary numbers using a queue:

1. Initialize an empty queue: **Queue = []**
2. Add the first binary number, "1," to the queue: **Queue = ["1"]**
3. Remove and print the front element of the queue:
   - Printed: "1"
   - Queue: **Queue = []**
4. Extend the front item by appending "0" to it and place it back in the queue: **Queue = ["10"]**
5. Extend the front item by appending "1" to it and place it back in the queue: **Queue = ["10","11"]**
6. Repeat steps 3 to 5 until you reach the desired end value.

Iteration 1:

- Remove and print the front element of the queue:
  Printed: "10"
  Queue: **Queue = ["11"]**
- Extend the front item by appending "0" to it and place it back in the queue:
  Queue = ["11", "100"]
- Extend the front item by appending "1" to it and place it back in the queue:
  Queue = ["11", "100", "101"]

Iteration 2:

- Remove and print the front element of the queue:
  Printed: "11"
  Queue: **Queue = ["100", "101"]**
- Extend the front item by appending "0" to it and place it back in the queue:
  Queue = ["100", "101", "110"]
- Extend the front item by appending "1" to it and place it back in the queue:
  Queue = ["100", "101", "110", "111"]

Iteration 3:

- Remove and print the front element of the queue:
  Printed: "100"
  Queue: **Queue = ["101", "110", "111"]**
- Terminate the loop as 100 which equivalent to 4 is printed.