SGSITS
Dept. of Computer Engineering
CO24007: Data Structure
Lab Assignment 3 B
Stacks and Queues

Date of submission: Monday 21st August, 2017

Stacks

1. **Three Stacks in an Array**

   Create a data structure ***threeStacks*** that represents three stacks. Implementation of ***threeStacks*** should use only one array, i.e., all three stacks should use the same array for storing elements. Following functions must be supported by ***threeStacks***:

   - **push(int x, int stackId)** — pushe x to stack with id *stackId*
   - **pop(int stackId)** — pop an element from stack with id *stackId* and return the popped element.

   Challenge: Can you efficiently consume space of the original array, so that every byte of the array can be consumed? (You may need additional arrays for bookkeeping).

2. **Equalizing Stacks**

   Consider that you have 3 stacks, each stores positive integers. Your task is to find out number of minimum pops that are required collectively so that sum of elements of each stack is same as the other two.

   For example: if three stacks are (3 2 1 1 1), (4 3 2) and (1 1 4 1) (with top be the left most element). If one element is removed from first two stacks and two elements from the third, they become (2 1 1 1), (3 2) and (4 1) and sum of elements is each stack becomes 5. Thus final answer is 4.

3. **Implement Stack with a Linked List**

   Implement a stack ADT with a linked list.

4. **Balanced parentheses**

   Given a C source code file, write a program to examine whether the pairs and the orders of "{“,”}”,”(“,”)”,”[“,”]” are correct in the file. Your program should print "Balanced" if all brackets and parenthesis are mathched correct. For example, the program should print "Not balanced" for following code:

```
#include <stdio.h>

main( //because ')' is missing
{
  int array[100],i;

  printf("Enter 100 numbers:\n");

  //take input in the array
  for (i=0;i<100;i++)
  {
    scanf("\%d",array[i]);
  }
}
```

5. **Infix Evaluation**
   You can combine the algorithms for converting between infix to postfix and for evaluating postfix to evaluate an infix expression directly. To do so you need two stacks: one to contain operators and the other to contain operands. When an operand is encountered, it is pushed onto the operand stack. When an operator is encountered, it is processed as described in the infix to postfix algorithm. When an operator is popped off the operator stack, it is processed as described in the postfix evaluation algorithm: The top two operands are popped off the operand stack, the operation is performed, and the result is pushed back onto the operand stack. Write a program to evaluate infix expressions directly using this combined algorithm.

Queues

6. **Queue via Stacks**
   Implement a Queue ADT using two stacks that support following operations:

   - *enqueue()*: put an element at the end
   - *dequeue()*: take out an element from the front and return it
   - *front()*: return the element on the front without returning it
   - *isEmplty()*: return 1 if the queue is empty, 0 otherwise

   Create a menu based interface that allows you to execute above operations, until user exists.

7. **Circular Queue: A Marathon**
   Suppose there is circle, and it has N petrol pumps. The petrol pumps are numbered from 0 to N-1. For each petrol pump, you know the amount of petrol available on it and the distance of next petrol pump.
   Assuming your car run 1 Km in 1 litre of petrol, you need to find out the petrol pump, from where you should start so that you can finish the circle. If tank of you car becomes empty before reaching the next petrol pump, you can not go any further.
   For example, if there are 3 petrol pumps with (petrolInLitres, distanceToNextPetrolPump) as (1,5), (10,3) and (3,4). Then you should start from second petrol pump (having with index 1).