Shri G. S. Institute of Technology and Science
Department Of Computer Engineering
CO 24007: DATA STRUCTURES
Lab Assignment # 03 (Stack & Queue)
Marks: 20 points

Submission Date: 20 Aug 2017@23:59
Demo Date: 21 Aug – 26 Aug 2017
Late Submission: Not allowed
No copying allowed. If found then students involved in copying will fail in this course.

1. Design and write a function for the stack and queue as abstract data type.

2. Write a function that reverses the order of words in a sentence. For eg, given the input : "life is like a box of chocolates" , your code should produce the output : "chocolates of box a like is life".

   For this, you have to implement the 'reverseWordsOfSentence' function using stack as a ADT.

   string reverseWordsOfSentence(string sentence) { }

   [Hint: One possible solution to this problem can be to push all the individual words on a stack, and then pop them off from the stack, thus reversing their order.]

3. There are several notations to write an algebraic expression. We will deal with two such notations - prefix notation and infix notation.

   Infix notation: Operators are written in-between their operands. This is the usual way we write expressions. An expression such as A *(B+C) is usually taken to mean something like: "First add B and C together, then multiply the result by A to give the final answer.

   Prefix notation: Operators are written before their operands. The expression given above is equivalent to *A+BC. Operators are evaluated left-to-right and brackets are superfluous(so they are removed)

   Your task is to write a function prefix_to_infix()

   This function will take a string containing the expression in prefix notation(assume exactly no spaces, operators to be binary and operators/operands to be single character long only). The function should return a string that contains the infix form of the expression(without any spaces).

Important Note: In infix notation, every group of operator and its two operands should enclosed by brackets. So (A+B/C) or (A+B)/C are wrong, while ((A+B)/C) or (A+(B/C)) are correct.

Example:

prefix_to_infix("/+2b-3a") should return "((2+b)/(3-a))"

4. You will be given a 10 X 10 matrix, where every element represents a block of the city. The city will have three types of blocks : free block, dead-ends, and target. Free blocks are represented as single digit numbers from 0-9 (repetitions allowed), dead-ends as 'x' and target as 't'. Note that there will be exactly one target block in the city i.e., only one element with 't' in the matrix. Also , you will be given a 10 × 10 matrix in which a traversal exists from the initial block (0,0) to the target block. [So , it is also guaranteed that block (0,0) doesn't contain 'x']
(Note : Block (i,j) means the element in the (i+1)th row and (j+1)th column of the given matrix)

   Initially , all blocks are marked unvisited .
   In the beginning, we add block (0,0) to the queue .[By adding block (i,j) to the queue, we mean that x-coordinate of block (i,j) i.e. i needs to be pushed to the queue of x and the y-coordinate of block i.e. j to the queue of y].We also mark this block as visited .You need to perform the following sequence of operations in order :

   0. If you are at target block, stop ; else go to step 1.
   1. Look right: if the block is either free or target (not 'x') and also unvisited, add it to the queue.*
   2. Look down: if the block is either free or target (not 'x') and also unvisited, add it to the queue.*
   3. Look left: if the block is either free or target (not 'x') and also unvisited, add it to the queue.*
   4. Look up: if the block is either free or target (not 'x') and also unvisited, add it to the queue.*
      (Skip any of the corresponding steps 1,2,3 or 4 if right,down,left and upper blocks do not exist respectively.)
   5. Dequeue the head element.
   6. If there are element(s) in the queue : (This is true for the given test cases)
      Go to next block on the front(head) of the queue. Then go to step 0.

   * Note that when you add a block to the queue in your code, you also need to set the block to be a visited one.
   (Definitely the elements of the queue in Step-5 would not be empty as the test cases provided are the cases which definitely have a possible traversal.)

   You need to write a function "move" which just adds the elements to the queue (x-coordinates to queue of x and y-coordinates to queue of y checking if the element is visited or not (based on the "visited" list)). Note that after you push block(i,j) to the queue, you need to mark it as visited by updating "visited" list in your code and also just push the corresponding character in that block to the "answer" list. This vector contains

the order in which the blocks are pushed to the queue and will be used to verify the correctness of the order of elements added to the queue by you.

5. In this assignment we have to illustrate the use of queues. This example refers to what we call a Mumbai Vada Pav Restaurant. Vada Pav, by the way, is a popular dish in western India, and of course, this restaurant like all other restaurants sells not just vada pav but many other items. Let us begin with an overview of the restaurant. The restaurant has fixed food items in the menu and there is one counter in which customers queue up to place orders, we assume that each customer can place order for at most ten items and each customer must specify the quantity to be purchased for that item.

   Next, let us look at the behavior of the customer. Every customer arrives at stands in queue, places order on his or her turn, gets a token ID and the cost to be paid, pays that cost, collects the token ID and waits at a different table to collect the order when it is ready. We will need some data structures to represent various entity. So we know that the restaurant maintains information about complete details of the menu. Menu has multiple items. So I will assume that we have a food ID where the name of the food item like Samosa, we have rate per item like 15.0 rupees. We will use structure foodInfo to maintain this information. So this is an example of typical data in the structure foodInfo. You've food ID, your food name, rate etc.

   Now, we will have to use queues, the first queue which is very obvious is a queue for customers. As we have already observed, a customer comes in a queue and when his or her turn comes, places order by giving multiple food IDs and quantity, then at the end gets the token-id and cost to be pay. So at the end of this queue customer has got a token-id and presumably he has paid the cost. Next, we will presume that the person at the counter passes on that order to the kitchen and kitchen prepares that item or that order and sends it for another queue which we call order dispatch queue. In this order dispatch queue, we'll simply have token-id which implies that once an order comes in the order dispatch queue that particular token-id order is ready to be collected by the customer. As I mentioned, kitchen itself will have to store information about orders. So we assume the following. We assume that each order placed by customer send to kitchen. Now, what is the pertinent information for us at this juncture? The pertinent information is that token-id and time required for preparation of that order. Of course, there will be many more details, but we will ignore those at the movement. We presumed that kitchen will prepare the ordered items. Different time will be needed to prepare an order and the orders may not be prepared necessarily in the sequence of their arrival.

   So, as I mentioned, for now, we use a simple array to represent kitchen operations. Each element of the array has token-id and time for preparation, which again we will arbitrarily assign in this demonstration. We will store every element of order queue in this array as an order is placed. So at the end, data structure in the kitchen array will have all orders that have been placed. For now, we will allocate an arbitrary time for order preparation for every order. Again as I mentioned last time in real life when we are simulating a true restaurant, we will have to generate a random number for the order preparation time based on various parameters like the items ordered etc. Let us demonstrate this entire process.

So here is the person from the restaurant standing here waiting for customers. The customers come one by one.

Let's assume that there are four customers who have joined this queue. The first customer is placing the order. When the order is placed, the customer will specify food ID and quantity. He gets let's say token number 1001 and cost is let's say 165. Order is placed. Now preparation is started in kitchen, because the order is sent immediately to that array for kitchen, which stores all the orders. This customer comes and waits at this table, okay. Waiting for the order to be dispatch. The next customer same thing happens. Note that the second customer has a smaller order cost but there are many items to be prepared. Obviously, this may take much longer to prepare than other orders. Here is a third customer. He has only one food item. Here is a fourth customer. He has only one food item. At the end all four people have collected their coupons, their orders have been placed and they are waiting at the table. As we notice, all item are being prepared in the kitchen.

Now this is a place where in our demonstration we shall assume some arbitrary time for order preparation. According to which the orders will be dispatched. Let us look at the kitchen order dispatch queue. Let's assume that the first order to be prepared is 1001 that comes in the kitchen dispatch queue and this person who is standing with token number T1001 is happy to receive this order. Let's assume that the second order which comes in the queue is 1003. We have already indicated that different orders may take different preparation time. So, this time, it is T1003 who gets the order fulfilled. Next, comes let's say T1002 and this customer now gets his order. Last is T1004.

In short, we noticed that the orders may be placed in a certain sequence such as token number 1001, 1002, 1003, 1004, 1005 etc, but the kitchen dispatch will depend on upon the time required for preparation of each order. Once again, we have taken all images from open clipart. They are all in the public domain, so you can feel free to use this. This session to conclude has indicated the organization of a typical restaurant where customers queue up to place orders and then wait for their orders to be fulfilled. We have demonstrated that we can do that using two queues. The first queue is the customer queue and the next queue is the order dispatch queue. Again in this illustration, we will assume arbitrary times for completing the order preparation and we have used arbitrary item codes and costs. Frankly, the item codes and cost are not relevant in this demonstration because we are concentrating on a use of queues.

```
struct customerInfo{
   int custID;
   int arrivalTime;
};

struct foodInfo{
   int foodID;
   string foodName;
   float rate;
};
```

```
struct orders {
    int tokenID;
    float cost;
};

struct kitchen {
  struct orders order;
  int preparationTime;
};

struct restaurant {
        static int custID, ordersInKitchen, tokenID, foodIndex;
        struct foodInfo food[100];
        struct orders order;
        struct kitchen k[100];
        queue<customerInfo> customerQueue;
        queue<int> orderDispatch;
}

void restaurant_loadMenu() {
    setFoodItems(1,"Vada Pav",10.0);
    setFoodItems(2,"Uttappa",18.0);
    setFoodItems(3,"Samosa",15.0);
    setFoodItems(4,"Mendu Vada",10.0);
    setFoodItems(5,"Missal Pav",20.0);
    setFoodItems(6,"Masala Dosa",25.0);
    setFoodItems(7,"Bhel Puri",30.0);
    setFoodItems(8,"Sev Puri",30.0);
    setFoodItems(9,"Pakoda",22.0);
    setFoodItems(10,"Bhajia",16.0);
    setFoodItems(11,"Sukhi Baji",19.0);
    setFoodItems(12,"Shira",26.0);

}

void setFoodItems(int id, string name, float rate) {
    food[foodIndex].foodID = id;
    food[foodIndex].foodName = name;
    food[foodIndex].rate = rate;
    foodIndex++;
}
```