Implementation of stack using an array

We have seen in the lecture class how a stack can be implemented with push and pop functions to handle integers. In this section we shall show how to handle a situation when we have to deal both with integers and character information.

Let us consider a factory where goods of varying quality are being produced in batches. We assume that all goods produced in a particular batch have the same quality. As the items are being produced, we want to store the information regarding the number of items produced and the quality of that batch ('H' for high quality, 'G' for good quality, 'A' for average quality etc.) on a stack using PUSH function. To deliver the items to a client, we want to use the POP function. The POP function is supposed to remove the items from the stack. We are also interested to find out if the stack is full or empty at any point in time using the functions ISFULL and ISEMPTY.

We start by declaring the following structure which has two arrays, one storing the number of items being produced in every batch, while the other storing the quality of the goods in that batch. The variable *top* indicates the position of the last item in an array.
The variable MAXSTACK defines the maximum size of the stack.

```
#define MAXSTACK 20
```

//Here is the structure declaration:

```
    struct arraystack {
      int items[MAXSTACK];
      char quality[MAXSTACK];
      int top;
};
```

Calling the stack functions from the main program

```
int main( )
{
```

// declare other variables….
// Declare the name of the stack structure as "factory"

```
struct arraystack factory ;
```

//initialize stack top to -1

```
factory.top = -1;
```

//jprod is number of items in a batch
//qlty is the quality of items in the batch ('H','G' etc)

```
. . . . . .
```
//read from file corresponding value of jprod and qlty
//then use the following statement to store the data on
//the stack "factory"
……..
…….…….
// Typical calls to push and pop functions
```
push( factory, jprod, qlty);

. . . . . .
. . . . . .
result = isEmpty(factory);

. . . . . .
. . . . . .
pop(factory,  &item, &qout );

// print qout and item


. . . . . .
```

//Function definitions
// We use the 'arrow' symbol to indicate component of a structure that is being referenced
//through a pointer. Thus if *produce is a pointer to the structure in any function call then
//we can write

```
//   produce-> top
```
to denote
```
(*produce ).top
```

//Precondition: Enter a valid pointer name which should not be NULL
//Postcondition: Returns 1 if stack is empty , otherwise returns 0.

```
int isEmpty( struct arraystack *produce )
{
      return ( produce-> top < 0 );
}
```

//Precondition:Enter a valid pointer name which should not be NULL
//Postcondition: Returns 1 if stack is full

```
int isFull( struct arraystack *produce )
{
      return ( produce->top >= MAXSTACK - 1 );
}
```

//Precondition: Enter a valid pointer name which should not be NULL along with item number and quality
//Postcondition: Pushes item number and its quality on the stack and updates the depth of stack

```
void push( struct arraystack *produce, int x, char q )
{
       if ( produce->top >= MAXSTACK - 1 )
         {
            printf( "\n% Stack is full.\n");
         }
          else {
               produce -> top = produce -> top + 1;
               produce -> items[ produce -> top ] = x;
               produce -> quality[ produce -> top ] = q;
              }
}
```

//Precondition:Enter a valid pointer name which should not be NULL along with xx that
//stores the integer type address of the variable item code and qq stores the char type
//address of the variable quality
//Postcondition: Pops item and its quality from the stack

```
void  pop( struct arraystack *produce ,int *xx, char *qq)
{
       int x = 0;
       if ( produce->top < 0 )
       {
       printf("\nStack is empty");
       }
        else {
              *xx = produce -> items[ produce ->top ];
              *qq = produce -> quality[ produce ->top ];
              produce -> top = produce -> top - 1;
              }
}
```

Implementation code for Queues:

We are now in a position to look at implementation issues related to Queues.
In the following implementation, we consider a queue where each object specifies the number of items produced in a factory of a certain quality. The quality is described by one of the characters V,G,A or P (V: very good, G: good, A: average, P: poor). The structure for the object is as follows;

```
typedef struct objectType {
     char quality;
     int items;
} *Object;
```

The structure of the queue includes the object, the front and rear indices and the maximum size of the queue.

```
typedef struct queueType {
     Object list; //list- ptr to circular array of Objects
     int front;
     int rear;
     int maxLen;  //maxLen- the length of the array
} *Queue;
```

The queue is initialized by dynamically creating space to hold maximum number of specified objects. Both the front and the rear indices are set to -1.

```
Queue initQueue(int maxsize)
{
     Queue newQ = (Queue)malloc(sizeof(struct queueType));
     newQ->list = (Object)calloc(maxsize, sizeof(struct
                    objectType));
     newQ->front = -1;
     newQ->rear = -1;
     newQ->maxLen = maxsize;
     return newQ;
}

int isQueueEmpty(Queue q)
{
     return (q->front == -1);
}
```

If
(front index is at the beginning, and rear index is at the end) OR
(front index is one more than rear index)
then return true (queue is full)

else return false (queue is not full)

```
int isQueueFull(Queue q)
{
     return (q->front==0 && q->rear==q->maxLen-1)
     || (q->front==q->rear+1);
}

void enqueue(Queue q, Object objInsert)
{
     int index;
     if( ! isQueueFull(q) ) {
          if( isQueueEmpty(q) )
               q->front = 0;
          if( q->rear == q->maxLen-1){
                    q->rear = 0;
          q->list[0] = *objInsert;
          }
          else {
               index = ++q->rear;
          q->list[index] = *objInsert;
          }
     }
}
```

To hold dequeued object temporarily, request memory to allocate space to hold item of type Object. If queue will be empty after this dequeue then set both indices to -1. If item corresponds to last location of queue front should be set to first location.

```
Object dequeue(Queue q)
{
  Object item = (Object)malloc(sizeof(struct objectType));


  if( ! isQueueEmpty(q) ) {
          *item = q->list[q->front];

          if( q->front == q->rear )
               q->rear = q->front = -1;

          else if( q->front == q->maxLen-1 )
               q->front = 0;
          else
               q->front++;
          return item;
  }
```

```
        else
                return NULL;
}
```