Data Structure using C++

Lecture 04

Reading Material

Data Structures and algorithm analysis in C++ Chapter. 3 3.1, 3.2, 3.2.1

Summary

- Infix to Postfix Example 1:
- Infix to Postfix Example 2:
- Postfix Evaluation
- Stack
- Operations on a stack
- Representing stacks
- Converting an expression from index to postfix
- Infix, Postfix , and prefix Expressions

Infix to Postfix Example 1:

A + B * C - D / E

Infix	Stack (bot->top)	Postfix
a) A + B * C - D / E		
b) + B * C - D / E		A
c) B * C - D / E	+	A
d) * C - D / E	+	A B
e) C - D / E	+ *	A B
f) - D / E	+ *	A B C
g) D / E	+ -	A B C *
h) / E	+ -	A B C * D
i) E	+ - /	A B C * D
j)	+ - /	A B C * D E
k)		A B C * D E / - +

Infix to Postfix Example 2:

A * B - (C + D) + E

Infix	Stack (bot->top)	Postfix
a) A * B - (C - D) + E	empty	empty
b) * B - (C + D) + E	empty	A
c) B - (C + D) + E	*	A
d) - (C + D) + E	*	A B
e) - (C + D) + E	empty	A B *
f) (C + D) + E	-	A B *
g) C + D) + E	- (	A B *
h) + D) + E	- (	A B * C
i) D) + E	- (+	A B * C
j)) + E	- (+	A B * C D
k) + E	-	A B * C D +
l) + E	empty	A B * C D + -
m) E	+	A B * C D + -
n)	+	A B * C D + - E
o)	empty	A B * C D + - E +

Postfix Evaluation:

Operand: push

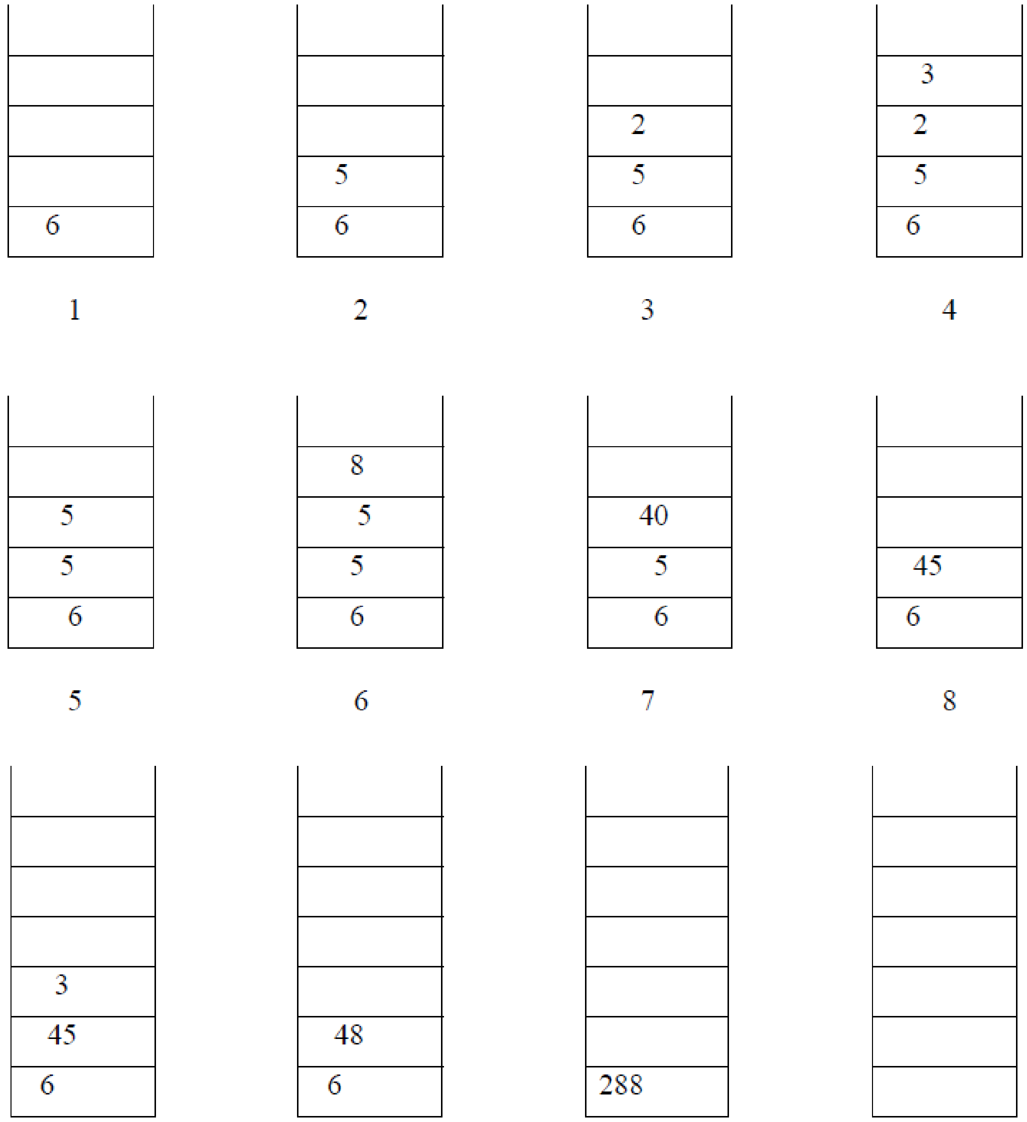
Operator: pop 2 operands, do the math, pop result
back onto stack

1 2 3 + *

Postfix	Stack(bot -> top)
a) 1 2 3 + *	
b) 2 3 + *	1
c) 3 + *	1 2
d) + *	1 2 3
e) *	1 5 // 5 from 2 + 3
f)	5 // 5 from 1 * 5

Example

6 5 2 3 + 8 * + 3 + *



The process stops when there are no more operator left in the string. The result of evaluating the expression is obtained just by popping off the single element. More examples will be done in the lecture and recitation labs.

A stack can also be used to convert an infix expression in standard form into postfix form. We shall assume that the expression is a legal one (i.e. it is possible to evaluate it). When an operand is read, it will be placed on output list (printed out straight away). The operators are pushed on a stack. However, if the priority of the top operator in the stack is higher than the operator being read, then it will be put on output list, and the new operator pushed on to the stack. The priority is assigned as follows.

Queues:

A queue is an ordered collection of items from which items may be deleted at one end (collect the front of the queue) and into which items may be inserted at the other end (called the nearer of the queue). The figure illustrates a queue containing there elements A, B and C. A is at the front of the queue and C is at the rear.

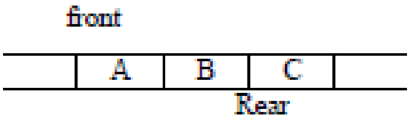


Figure -1-

In figure -2- an element has been deleted from the queue. Since may be deleted only from the front of the queue, A is removed and B is now at the front.

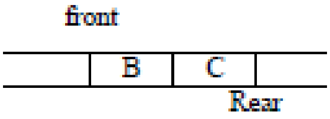


Figure -2-

In figure -3- , when items D and E are inserted , they may be inserted at the rear of the queue.

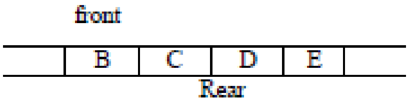


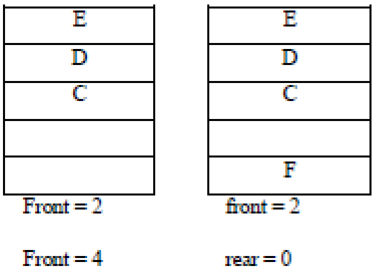
Figure -3-

Initially, the queue is empty. Then H, B, and C have been inserted, then tow items have been deleted into two new items D and E have been inserted. Although there are two spaces in the queue array but, we cannot insert a new element to this queue, because of the overflow case is notified when

q. rear = = maxqueue -1

A more efficient queue representation is to treat the array of the queue as a circuited rather than a straight line. That means even if the last element of the array is occupied, anew value can be inserted in the first element of the array as long as that element is empty.

Unfortunately, it is difficult under this representation to determine when the queue is empty. The condition q. rear < q. front is no longer valid as a test for the empty queue.



One may to solve this problem is to establish the conversation that the value of front is the array index immediately preceding the first element of the queue rather than the index of the first element itself. Since rear is the index of last element of the queue, the condition q. front = = q. rear.

The first element inserted into a queue is the first element to be removed. For this reason a queue is sometimes called a FIFO (first – in first – out) list. Three primitive operations can be applied to a queue. The operation insert (q,x) inserts item x at the rear of the queue q. the operation.

X = remove (a) deletes the front element from the queue q and sets x to its contents.

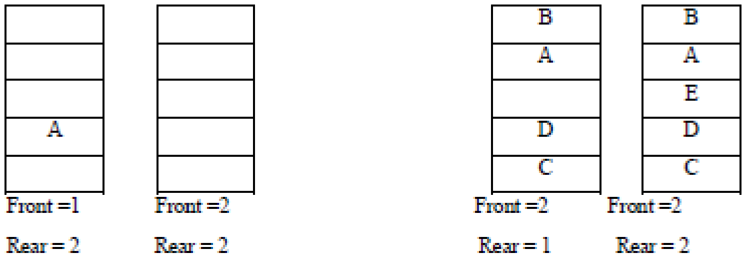
The third operation, empty (a), returns false or true depending on whether or not the queue contains any element.

The insert operation can always be performed since there is no limit to the number of elements a queue may contain, the remove operation, however, can be applied only if the queue is nonempty; there is no way to remove an element from a queue containing no element. The result of an illegal attempt is called underflow. The empty operation is, of course, always application.

Implies that the queue is empty. A queue of integers may therefore be declared and initialized by:

```
# define queue {
            int items [MAXCUEUE];
            int front , rear;
            };
             Struct queue q;
             q. front = q. rear = MAXCUEUE – 1;
```

this solution leads a new problem that we cannot distinguish between full and empty queue.

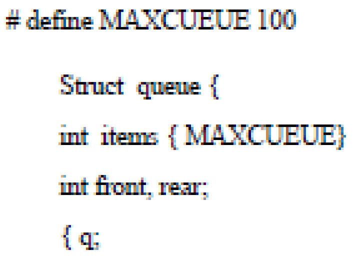


In figure (a) removing the last element, leaving the queue empty. In figure (b) adding an element to the last free slat in the queue leaving the queue full. The value of front and rear in the two situation are identical.

One solution is to scarifies one element of the array and to allow a queue to grow only as large as one less than the size of the array.

Implementation of queues:

The queue can be represented using an array to hold the elements of the queue and to use two variables, front and rear, to hold the positions within the array of the first and last elements of the queue. Of course using an array, to hold queue introduce the possibility of overflow, if the queue should grow larger than the size of the array.

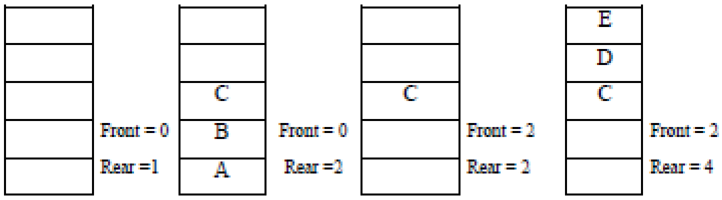


```
# define MAXCUEUE 100

    Struct  queue {
    int  items { MAXCUEUE}
    int front, rear;
    { q;
```

Initially, q. rear is set to -1 , and q. front is set to 0. The queue is empty whenever q. rear < q. front, and the number of elements in the queue at any time is equal to the value of q. rear –q. front +1.

Let us examine what might happen under this representation, the following figure illustrates an array of five elements used to represent q queue.


Front = 0
Rear =1
C
B
A
Front = 0
Rear =2
C
Front = 2
Rear = 2
E
D
C
Front = 2
Rear = 4