

CHAPTER 6







STACK AND QUEUE

“The art of programming is the art of organizing complexity”. - W. W. Dijkstra

In this chapter, we discuss some elementary data structures like stack and queue along with their properties, different operations that are performed on them, algorithms of their different operations and applications of them.

In 1946, Alan M. Turing first proposed the stack, in the computer design. In 1955, Klaus Samelson and Friedrich L. Bauer of Technical University Munich proposed the idea of the stack and filed a patent in 1957. The Australian Charles Leonard Hamblin in the first half of 1957 developed the same concept, independently.

KEY FEATURES

	Stack
	Evaluation of Expression
	Queue
	Circular Queue
	Dequeue
	Priority Queue

STACK

A stack is a one of the most important and useful non-primitive linear data structure in computer science. Real-life examples of the stack are a stack of books, a stack of plates, a stack of cards, a stack of coins, etc.

Definition: A stack is a sequential collection of elements into which new elements are inserted and from which, elements are deleted only at one end.

As all the insertion and deletion in a stack is done from the top of the stack, the lastly added element will be first to be removed from the stack. That is the reason why stack is also called **Last-In-First-Out (LIFO)** data structure. Note that the most frequently accessed element in the stack is the top most elemental, whereas the least accessible element is the bottom of the stack.

In the stack, the top variable is used to point the top of the stack. The following tasks are performed by the top variable:

- To keep track, how many cells are used,
- Whether the stack is full or empty
- Insert new element in the stack
- Delete elements from the stack

Operations on Stack

The stack is an abstract data type since it is defined in terms of operations on it and its implementation is hidden. Therefore, we can implement a stack using either array or linked list. The stack includes a finite sequence of the same type of items with the different operations described in table 6.1.

Table 6.1: Operations on Stack

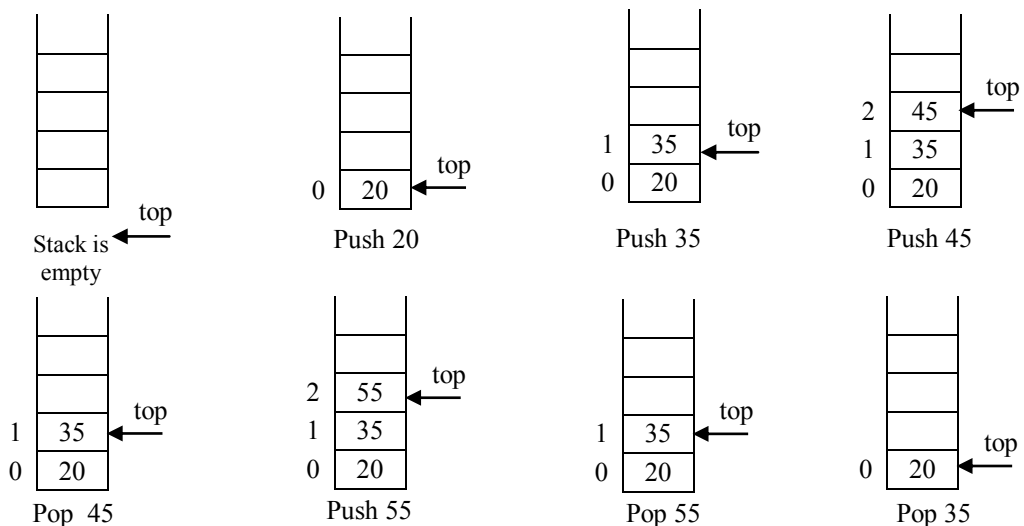
Operation	Description
Creation	This operation creates a stack and initializes the stack.
Iseempty	This operation checks whether the stack is empty or not.
Isefull	This operation checks whether the stack is full or not.
Insertion (Push)	This operation inserts an item only at the top of the stack when the stack is not full.
Deletion (Pop)	This operation deletes an item only from the top of the stack when the stack is not empty.
Peek	This operation returns the value of the top of the stack without removing the element from the stack.

The insertion (or addition) operation is referred as push, and the deletion (or remove) operation as pop. A stack is said to be empty if the stack contains no elements. At this point, the top of the stack is present at the bottom of the stack. In addition, it is full when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point, the top pointer is at the highest location of the stack. If the pop operation in the stack when it is in the empty state then underflow occurs. Similarly, if the push operation is done in a full stack, then stack overflow happens.

Stack as an ADT promotes data abstraction and focuses on what operations it does rather than how it does (implements). Therefore, the stack can be implemented with an array or with the linked list.

Stack representation with Array

The array can be used to implement a stack of fixed size, therefore only fixed a number of data items can be pushed and popped. Consider the following figure 6.1, the stack of size 4; therefore, maximum only 4 data items can be inserted. The top index always keeps track of the last inserted element of the stack, which is the top of the stack. Initially, the top is initialized by -1 (for the zero-based array) when there are no items in the stack, i.e. stack is empty.

**Figure 6.1:** Stack Operations

When a new item is inserted in the stack, the top is incremented by 1 before the item is stored in the stack. Once all the four items are inserted then pop is 3, as shown in the figure. Now, if we try to insert next item, it leads to an overflow condition. It indicates that the stack is full and we cannot insert the new item. If the size of the array is MAXSIZE then when the top equals to MAXSIZE -1 then the stack is full.

When an item is deleted from the stack, the top is decremented by 1, after the item is removed from the queue. Now, if top = -1 then if we try to delete an item, it results in underflow condition. It indicates that the stack is empty and we cannot delete an item.

Therefore, it is required to check these conditions whenever push and pop operations take place.

Algorithm to insert (push) onto the stack

Algorithm: PUSH (STACK, ITEM)

[STACK is an array of MAXSIZE and ITEM is an item to be pushed onto stack]

1. [Check for stack overflow]
 - If TOP = MAXSIZE - 1 then
 - a) Print: Overflow
 - b) Return
2. [Increase top by 1]
 - Set TOP = TOP + 1
3. [Insert item in new top position]
 - Set STACK[TOP] = ITEM
4. Return

Algorithm to deletes (pop) the top element from the stack.

Algorithm: POP (STACK, ITEM)

[STACK is an array and ITEM is an item to be popped from stack]

1. [Check for stack underflow]
 - If TOP = -1 then
 - a) Print: Underflow
 - b) Return
2. [Assign top element to item]
 - Set ITEM = STACK[TOP]
3. [Decrease top by 1]
 - Set TOP = TOP - 1
4. Return

The stack can be represented using the following structure:

```
struct STACK
{
    int a[MAXSIZE];
    int top;
};
```

Stack Representation with Linked List

Another way to represent stack is by using the singly linked list, which is also known as **Linked Stack**. A linked list is a dynamic data structure and each element of a linked list is a node that contains a value and a link to its neighbor. The link is a pointer to another node that contains a value and another pointer to another node and so on.

The linked list header acts as the top of the Stack. All push or pop operations are taking place at the front of the linked list. Each operation always changes the header of the linked list. When the stack is empty then HEAD is null. If the stack has at least one node, the first node is the top of the stack. In the push operation, it needs to add a new node to the front of the list. The pop operation removes the first node of the linked list when the stack is not empty.

Algorithm of the push operation using linked list

Algorithm: PUSH (HEAD, ITEM)

[HEAD is a pointer to the first node and ITEM is an item to be pushed onto stack]

1. [Create the new node]
 Allocate memory for NEW node
2. If NEW = NULL then
 - i) Print: Out of memory
 - ii) Return
3. Set NEW→DATA = ITEM
4. Set NEW→LINK = HEAD
5. Set HEAD = NEW
6. Return

Algorithm of pop operation using linked list

Algorithm: POP (HEAD, ITEM)

[HEAD is a pointer to the first node and ITEM is an item to be popped from stack]

1. [Whether List is empty]
 If HEAD = NULL then
 - i) Print: Stack is underflow
 - ii) Return
2. ITEM = HEAD→DATA
3. Set P = HEAD
4. HEAD = HEAD→LINK
5. Set P→LINK = NULL
6. De-allocate memory for node P
7. Return

Comparisons of stack representation using linked list over array

- The array is fixed size, therefore, a number of elements will be limited in the stack. Since linked list is dynamic and can be changed easily, so the number of elements can be changed.

- The pointers in linked list consume additional memory compared to an array.
- In array and linked list push, pop operations can be done in $O(1)$.

Applications of Stack

The stack is a very useful data structure. Most of the modern computers and microprocessor provide an inbuilt hardware stack. Even there are stack-oriented computer architectures.

1. A very important application of stack is to implement recursive function call and processing of function calls such as passing arguments.
2. Evaluation of Arithmetic expressions.
3. To simulate recursion.
4. The scope rule and block structure can also be implemented using the stack. Stacks are used in the development of Compilers, System programs, Operating systems and in many elegant application algorithms.
5. Stack is used to implement different algorithms, Depth first search, Quicksort, Mergesort etc.

Processing of Function calls

One of the most important applications of the stack is the processing of subprogram calls and their termination.

The program must remember the place where the call was made: so that it can return back after the subprogram is complete.

Suppose we have three subprograms called $A()$, $B()$, $C()$ and one main program and $main()$ invokes $A()$, $A()$ invokes $B()$ and $B()$ in turn invokes $C()$. Then B will not have finished its work until $C()$ has finished and returned. Similarly $main()$ is the first to start work, but it is the last to be finished.

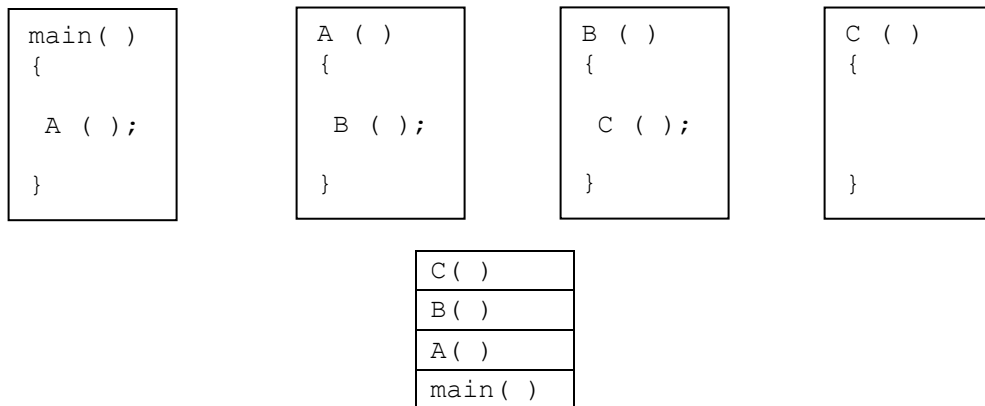


Figure 6.2: Function call processing

These function calls are pushed into the stack according to the order in which they were called along with their parameters and processed according to the order in which they are popped from the stack. The stack, which is used in function-call, is generally implemented in physical memory and in the reverse order. That means the stack top locates the last address.

Evaluation of Arithmetic Expressions

An expression is defined as a number of operands or data items combined with several operators. An Arithmetic expression consists of arithmetic operators and operands.

There are three types of notations in an arithmetic expression.

- i) Infix Notation
- ii) Prefix Notation
- iii) Postfix Notation

Infix Notation

Most usually, in arithmetic expressions, the binary operator appears between its two operands. This is called infix notation. The general form of the infix notation is:

Op1 operator Op2 where Op1 and Op2 are two operands

Example:

a + b

Prefix Notation

In prefix notation, binary operators appear before its two operands. This notation is also known as **Polish notation**. In prefix notation, the operations that are to be performed is absolutely determined by the positions of the operators and operands in the expression. Therefore, parentheses are never used when writing expressions in prefix notation. The general form of the prefix notation is:

operator Op1 Op2 where Op1 and Op2 are two operands

Example:

+ a b

Postfix Notation

In postfix notation, binary operators appear after its two operands. This notation is also known as **Reverse Polish notation**. In postfix notation, the operations are to be performed is absolutely determined by the positions of the operators and operands in the expression. Therefore, parentheses are never used when writing expressions in postfix notation. The general form of the postfix notation is:

Op1 Op2 operator where Op1 and Op2 are two operands

Example:

a b +

We are familiar with the conventional infix notation. However, postfix notation is most suitable for a computer to calculate any expression and it is the universally accepted notation for designing Arithmetic Logic Unit (ALU) of CPU. Therefore, it is necessary for us to study the postfix notation. Postfix expression is the way the computer looks towards any arithmetic expression.

Disadvantages of infix expression

- The infix expression is evaluated by traversing in more than one passes.

- The infix expression does not exclusively define the order in which the operators are to be evaluated. In infix expression, the operators are evaluated on basis of operator precedence convention.
- In infix notation, one can increase the operator precedence by using parentheses. Therefore, parentheses can change the order of evaluation in an infix expression.
- While storing the fully parenthesized expression is wasteful, since the parentheses in the expression need to be stored to evaluate the expression properly.

Advantages of postfix expression

- The postfix expression is evaluated by traversing only one pass. Therefore, evaluation of postfix expression is faster than infix expression.
- Postfix expression is already arranged according to the operator precedence. Therefore, one never needs to look at the precedence of the operator during evaluation
- In postfix expression, parentheses are never used.

The Computer usually evaluates an expression written in infix notation in two steps:

- i) Converts the infix expression to equivalent postfix expression
- ii) Evaluates the postfix expression in a single pass

Converting infix expression to postfix form

The order of evaluation can be fixed by assigning a priority to each operator. The operators within parentheses having the highest priority will be evaluated first. When an expression has two operators with same priority then the expression is evaluated according to its associativity (left to right or right to left) order. In table 6.2, the priorities of different operators are specified.

Table 6.2: Priority of Operators in the order of evaluation

Operator	Description	Priority
+ -	Unary operator	5
^	Power operator	4
* / %	Multiplication, Division, Remainder	3
+ -	Addition, Subtraction	2
< <= > >= == !=	Relational operators	1

There is an algorithm to convert an infix expression to the equivalent postfix expression. A stack is used here to store operators rather than operand. The purpose of the stack is to reverse the order of the operators in the expression.

Algorithm converts an infix expression to the equivalent postfix expression.

Algorithm: POSTFIX (Q, P)

[Q is a given infix expression and P is a postfix expression]

1. Push "(" onto stack & add ")" to the end of Q.
2. Scan Q from left to right and repeat step 3 to 6
for each element (symbol) of Q while the stack is not empty
3. If the element is an operand then add it to P.

4. If the element is left parenthesis "(" then push it onto the stack.
 5. If the element is an operator then:
 - a) Repeatedly pop from stack (until the element on top of the stack has higher or same precedence than the operator currently scanned) and add it to P.
 - b) Add the operator to stack.
 6. If the element is a right parenthesis ")" then:
 - a) Repeatedly pop from stack and add to P each operator until a left parenthesis "(" is found
 - b) Pop the left parenthesis from the stack.
- [End of Loop]
7. Return

Example:

Find the postfix expression of the following infix expression:

$$Q = A + (B * C - (D / E \uparrow F) * G) * H$$

Serial Number	Symbol Scanned	Stack	Postfix Expression (P)
Initial		(
1	A	(A
2	+	(+	A
3	((+ (A
4	B	(+ (A B
5	*	(+ (*	A B
6	C	(+ (*	A B C
7	-	(+ (-	A B C *
8	((+ (- (A B C *
9	D	(+ (- (A B C * D
10	/	(+ (- (/	A B C * D
11	E	(+ (- (/	A B C * D E
12	↑	(+ (- (/↑	A B C * D E
13	F	(+ (- (/↑	A B C * D E F
14)	(+ (-	A B C * D E F ↑ /
15	*	(+ (- *	A B C * D E F ↑ /
16	G	(+ (- *	A B C * D E F ↑ / G
17)	(+	A B C * D E F ↑ / G * -
18	*	(+ *	A B C * D E F ↑ / G * -
19	H	(+ *	A B C * D E F ↑ / G * - H
20)	STACK EMPTY	A B C * D E F ↑ / G * - H * +

Postfix expression A B C * D E F ↑ / G * - H * +

Example:

Find the postfix expression of the following infix expression:

$$Q = (a + b * x) / (!a - d). s - c \wedge y$$

Serial Number	Symbol Scanned	STACK	Postfix Expression
Initial		(
1	(((
2	A	((A
3	+	((+	A
4	B	((+	a b
5	*	((+ *	a b
6	X	((+ *	a b x
7)	(a b x * +
8	/	(/	a b x * +
9	((/ (a b x * +
10	A	(/ (a b x * + a
11	!	(/ (!	a b x * + a
12	-	(/ (-	a b x * + a !
13	D	(/ (-	a b x * + a ! d
14)	(/	a b x * + a ! d -
15	.	(.	a b x * + a ! d - /
16	S	(.	a b x * + a ! d - / s
17	-	(-	a b x * + a ! d - / s .
18	C	(-	a b x * + a ! d - / s . c
19	\wedge	(- \wedge	a b x * + a ! d - / s . c
20	Y	(- \wedge	a b x * + a ! d - / s . c y
21)	stack empty	a b x * + a ! d - / s . c y \wedge -

The equivalent postfix expression of the given infix expression is $a b x * + a ! d - / s . c y \wedge -$

The time complexity of evaluation algorithm is $O(n)$ where n is a number of characters in input expression.

Evaluation of a Postfix Expression

Algorithm finds the value of an arithmetic expression P written in postfix notation.

Algorithm: EVALUATION (P)

[P is a postfix expression]

1. Add a right parenthesis ")" at the end of P.
2. Read P from left to right and repeat step 3 and 4 for each element of P until the ")" is found.
3. If an operand is found, put it onto the stack.

```

4. If an operator # is found then
    a) Pop the two top elements of the stack,
       Where A is the top element and B is the next to top element
    b) Evaluate  $R = B \# A$ 
    c) Push R onto the stack
    [End of If]
  [End of Loop]
5. Set Result equals to the top element on stack
6. Return

```

Example:

Find the value of following postfix expression:

5 3 2 * 8 + *

Serial Number	Symbol Scanned	Stack	Output
1	5	5	
2	3	5 3	
3	2	5 3 2	
4	*	5 6	
5	8	5 6 8	
6	+	5 14	
7	*	70	
			70

The time complexity of evaluation algorithm is $O(n)$ where n is a number of characters in input expression.

Converting infix expression to prefix form

There is an algorithm to convert an infix expression to the equivalent prefix expression. A stack is used here to store operators rather than operand.

Algorithm converts an infix expression to the equivalent prefix expression.

Algorithm: PREFIX (A, B)

[A is a given infix expression and B is a prefix expression]

```

1. Push ")" onto STACK, and add "(" to end of the A
2. Scan A from right to left and repeat step 3 to 6
   for each element(symbol) of A while the STACK is not empty
3. If the element is an operand then add it to B
4. If the element is a right parenthesis then push it onto STACK
5. If the element is an operator then:
   a) Repeatedly pop from STACK and add to B, each operator (on the
      top of STACK) which has same or higher precedence than the operator.
   b) Add operator to STACK
6. If the element is a left parenthesis then

```

```

    a) Repeatedly pop from the STACK and add to B (each operator on
top of stack until a left parenthesis is found)
    b) Pop the left parenthesis from stack.
    [End of loop]
7. Return

```

We can also convert an infix expression to prefix expression in three steps. In the first step reverse the infix expression and convert left parenthesis to right parenthesis and vice versa. Then in the second step, the modified infix expression is converted into postfix expression which algorithm is discussed above. Finally, the postfix expression is reversed to get the equivalent prefix expression.

QUEUE

The queue is also another useful non-primitive linear data structure in computer science. A real-life example of the queue is line or sequence of people or vehicles awaiting their turn to be attended to or to proceed.

Definition: A queue is a homogeneous collection of elements in which deletions can take place only at the front end, known as dequeue and insertions can take place only at the rear end, known as enqueue.

The element to enter the queue first will be deleted from the queue first. That is why a queue is called **First-In-First-Out (FIFO)** system.

The concept of the queue can be understood by our real life problems. For example, a customer comes and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of the queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service center.

Operations on Queue

The queue is an abstract data type since it is defined in terms of operations on it and its implementation is hidden. Therefore, the queue can be implemented with an array or with the help of the linked list. The queue includes a finite sequence of the same type of items with the different operations described in table 6.3.

Table 6.3: Operations on Queue

Operation	Description
Creation	This operation creates a queue and initialization is done here.
isempty	This operation checks whether the queue is empty or not.
isfull	This operation checks whether the queue is full or not.
Insertion (Enqueue)	This operation inserts an item only at the rear of the queue when the queue is not full.
Deletion (Dequeue)	This operation deletes an item only from the front of the queue when the queue is not empty.
Peek	This operation returns the value of the front of the queue without removing the element from the queue.

Table 6.4: Difference between stack and queue

Stack	Queue
In the stack, items are inserted and deleted at the one end of the list	In the queue, items are inserted at one end (called rear) and deleted at another end (called the front)
Stack is Last-in-First-out system	The queue is the First-in-First-out system

Queue Representation with Array

The array can be used to implement a queue of fixed size, therefore only fixed a number of data items can be inserted and deleted. Consider the following example, the queue of size 5, therefore, maximum only 5 data items can be inserted. The front index always keeps track of the last deleted item from the queue and rear index always keep track of the last inserted item in the queue. Initially, front and rear both are initialized by -1 (for the zero-based array) when there are no items in the queue, i.e. the queue is empty.

When a new item is inserted in the queue, the rear is incremented by 1 before the item is stored in the queue. Once all the five items are inserted, then rear is 4, as shown in the figure 6.4. Now, if we try to insert next item, it leads to an overflow condition. It indicates that the queue is full and we cannot insert the new item.

Algorithm to insert an item to rear of a queue by using an array

Algorithm: ENQUEUE (Q, ITEM)

[Q is an array represent queue and ITEM is deleted item]

1. [check overflow]
 - If $\text{Rear} = \text{MAX} - 1$ then
 - a) Print: Queue is Full
 - b) Return
2. Set $\text{Rear} = \text{Rear} + 1$
3. $Q[\text{Rear}] = \text{ITEM}$
4. Return

When an item is deleted from the queue, the front is incremented by 1, before the item is removed from the queue. Now, if $\text{front} = \text{rear}$ then if we try to delete an item, it results in underflow condition. It indicates that the queue is empty and we cannot delete an item. Whenever the queue is found empty, then to reuse the empty slots at the front of the queue we can reset the front and rear by -1.

Algorithm to delete from the front of a queue by using an array

Algorithm: DEQUEUE (Q, ITEM)

[Q is an array represent queue and ITEM is inserted item]

1. [Check underflow]
 - If $\text{Rear} = \text{Front}$ then
 - a) Print: Queue is Empty
 - b) Return
2. Set $\text{Front} = \text{Front} + 1$
3. $\text{ITEM} = Q[\text{Front}]$
4. If $\text{Rear} = \text{Front}$ then

```
Set Front = Rear = -1
5. Return
```

Therefore, it is required to check overflow and underflow conditions whenever insertions and deletions operations take place.

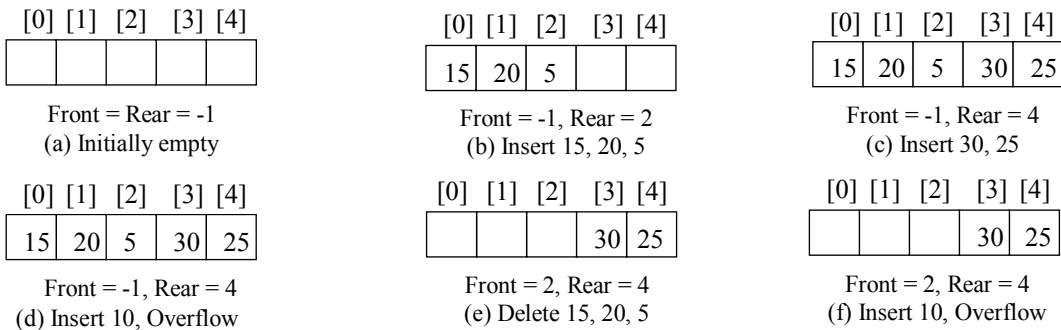


Figure 6.4: Different operations on Queue

Queue Representation with Linked List

Singly linked list can be used to represent a queue, which is also known as **Linked Queue**. In this representation, any number of data items can be inserted and deleted. The front and rear pointers always keep track of the first node and the last node in the linked list respectively. Initially, front and rear are initialized by null (i.e. front = rear = null), when there are no items in the queue, that means the queue is empty. The linked list header acts as the front of the queue. All deletion operations take place at the front of the list. All insertion operations take place at the end of the list. If the queue contains a single element then front and rear points to head/new node (i.e. front = rear = head).

When a new item is inserted in the queue, a new node is inserted at the end of the linked list, the rear points to the new node. When an item is deleted from the queue, the node from the front of the queue is deleted. Now, if front = null then if we try to delete an item, it results in underflow condition. It indicates that the queue is empty and we cannot delete an item. Whenever the queue is found empty, we can reset the front and rear by null.

Algorithm to insert an item to rear of a queue using linked list

Algorithm: ENQUEUE (FRONT, REAR, ITEM)

[FRONT points to the first node and REAR points to the last node of the linked list. ITEM is the inserted value]

1. Allocate memory for NEW node.
2. If NEW = NULL then Print: Out of memory and Return
3. Set NEW → DATA = ITEM
4. Set NEW → LINK = NULL
5. If REAR = NULL then
 - Set FRONT = REAR = NEW
- Else

```

        Set REAR → LINK = NEW
        Set REAR = NEW
    [End of If]
6. Return

```

Algorithm to delete from the front of a queue using linked list

Algorithm: DEQUEUE (FRONT, REAR, ITEM)

[FRONT points to the first node and REAR points to the last node of the linked list. ITEM is the deleted value]

1. If FRONT = NULL then Print: Stack is underflow and Return
2. Set P = FRONT
3. Set ITEM = P → DATA
4. Set FRONT = P → LINK
5. If FRONT = NULL then Set REAR = NULL
6. Set P → LINK = NULL
7. De-allocate memory for node P
8. Return

Comparisons of queue representation using linked list over the array

- The array is fixed size, therefore, a number of elements will be limited in the queue. Since linked list is dynamic and can be changed easily, so the number of elements can be changed.
- The pointers in linked list consume additional memory compared to an array.
- In array implementation, sometimes dequeue operation not possible, although there are free slots. This drawback can be overcome in linked list representation.
- In array and linked list enqueue and dequeue operations can be done in O(1).

Application of Queue

- A major application of the queue is in simulation [see Kruse for example].
- In operating systems, queues are used for process management, I/O request handling, etc.
Examples: Print queue of DOS, Message queue of Unix IPC.
- Queues are also used in some elegant algorithms like graph algorithms (breadth first search), radix sort etc.
- Different types of customer service software are designed using a queue for proper service to the customer.

Example: Railway ticket reservation system

Drawbacks of Linear Queue

The linear queue, when represented using an array, suffers from drawbacks. Once the queue is full, even though few elements are deleted from the front end and some free slots are created, it is not possible to insert new elements, as the rear has already reached the queue's rear most position. Consider the figure, the queue of size 5 and the front is 2, rear is 4. Now, we are not able to insert new data item into the queue, although there are free slots (first and second location) in the front of the queue, because of rigid rule followed by linear queue (insertion can be done at the rear end of the queue). It is also known as a boundary case problem.

This drawback can be overcome in two different ways. The first solution is by left shifting all elements after every deletion. However, this is not suitable since after every deletion, the entire elements required shifting left and front and rear should be readjusted according to that.

The second solution is by implementing a circular queue and it is a suitable method to overcome the above drawback.

CIRCULAR QUEUE

A circular queue (also known as a circular buffer) is a linear data structure that uses a single, fixed-size buffer as if it were connected end-to-end. A circular queue is just one of the efficient ways to implement a queue. It also follows First-in-First-out (FIFO) principle.

Circular Queue Representation with Array

In array representation, the queue is considered as circular queue when the positions 0 and MAX-1 are adjacent. It means when rear (or front) reaches MAX-1 position then increment in rear (or front) causes rear (or front) to reach the first position that is 0.

One solution of the problem is if not all the elements of the array can be used to accommodate queue elements; in particular, an array of size n can accommodate a maximum of $n - 1$ elements and one of the slot always remains unused.

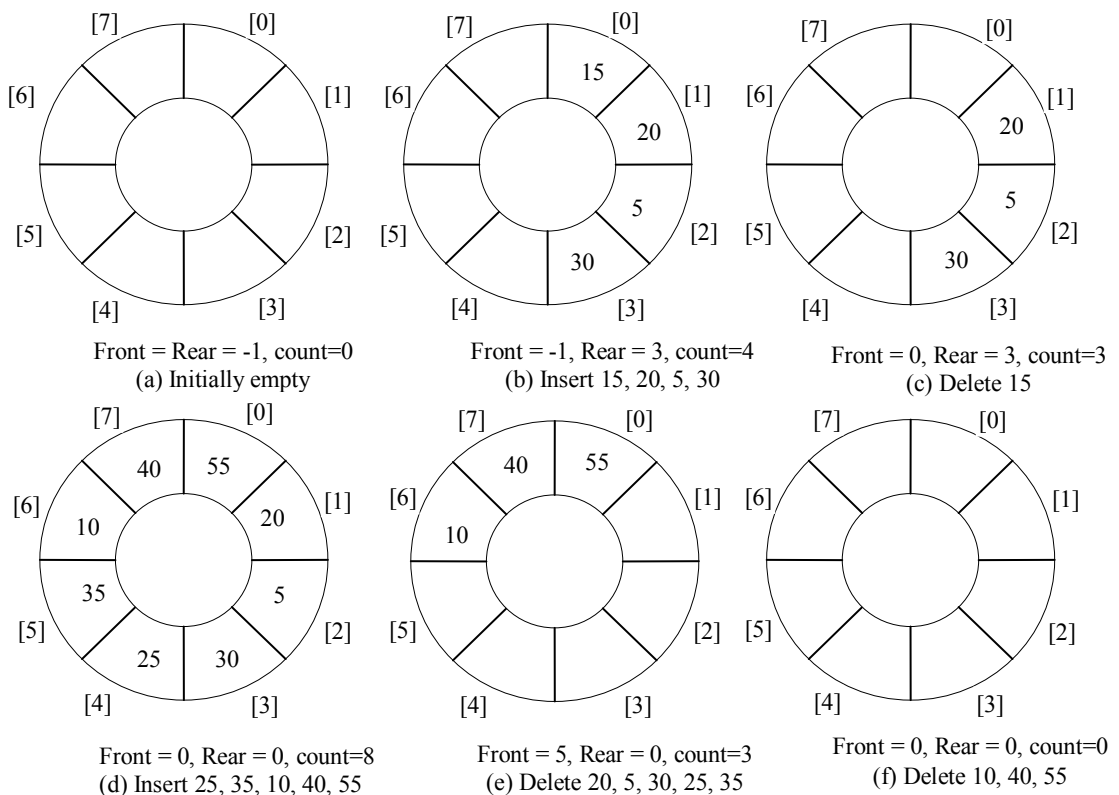


Figure 6.5: Different operations on Circular Queue

Consider the circular queue with $n = 8$ in figure 6.5.

Another solution is that if an extra variable counter is used for the identification of the empty queue and full queue. At first, the variable counter initialized by zero. When an element is inserted into the circular queue, the counter is increased by one and when an element is removed from the circular queue, the counter is decreased by one. Now, if the value of the counter is zero that means, a circular queue is empty and if the value of the counter equals to n (the size of the array) then the circular queue is full.

Note: Circular Queue's circularity is only logical. There cannot be a physical circularity in main memory.

Algorithm to insert an item to the rear of a circular queue.

Algorithm: ENQUEUE (Q, ITEM)

[Q is an array represent circular queue and ITEM is inserted item]

1. [Check for overflow]
 - If Count = MAX then
 - a) Print: Queue is Full
 - b) Return
2. Set $Rear = (Rear + 1) \bmod MAX$
3. Set $Q[Rear] = ITEM$
4. Set Count = Count + 1
5. Return

Algorithm to delete from the front of a circular queue

Algorithm: DEQUEUE (Q, ITEM)

[Q is an array represent queue and ITEM is deleted item]

1. [Check for underflow]
 - If Count = 0 then
 - a) Print: Queue is Empty
 - b) Return
2. Set $Front = (Front + 1) \bmod MAX$
3. $ITEM = Q[Front]$
4. Set Count = Count - 1
5. Return

DEQUEUE

A Double-ended Queue (Deque often abbreviated to Deque) is a linear list that generalizes a queue, for which elements can be inserted or deleted from either the front end or from the rear end but not in the middle. It is also often called a head-tail linked list.

There are two variations of a Dequeue

- i) **Input-restricted Dequeue:** Input-restricted dequeue is dequeue, where insertions can be made at only one end of the list but deletions can be made from both ends of the list.
- ii) **Output-restricted Dequeue:** Output-restricted dequeue is dequeue, where deletions can be

made from only one end of the list but insertions can be made at both ends of the list.

Both the basic and most common list types in computing, queues and stacks can be considered specializations of dequeue and can be implemented using dequeue.

Note: Deque is sometimes written dequeue, but this use is generally deprecated in technical writing because dequeue is also a verb meaning, "to remove from a queue".

Operations on Dequeue

The operations are supported by Dequeue as follows:

Table 6.5: Operations on Dequeue

Operation	Description
Insertion at Rear	This operation inserts the element into dequeue at rear end.
Insertion at Front	This operation inserts the element into dequeue at front end.
Deletion from Rear	This operation removes an element from dequeue from rear end.
Deletion from Front	This operation removes an element from dequeue from front end.

PRIORITY QUEUE

A Priority Queue is a collection of elements such that each element has been assigned a priority and the elements are arranged on the basis of priority. The order in which elements are deleted and processed comes from the following rules:

- i) The element having a higher priority is processed before any elements of lower priority.
- ii) The two elements that have the same priority are processed according to the order in which that are inserted into the priority queue.

There are two types of priority queue:

- i) **Ascending priority queue:** In this type of the priority queue, the elements can be inserted arbitrarily, but only the element with the smallest priority can be removed.
- ii) **Descending priority queue:** In this type of the priority queue, the elements can be inserted arbitrarily, but only the element with the highest priority can be removed.

Here we discuss the operations and algorithms for descending priority queue.

Abstract Data Type

A priority queue is an abstract concept. A priority queue can be implemented with an array, a linked list or a heap. Priority Queue supports the operations as follows:

Table 6.6: Operations on Priority Queue

Operation	Description
Insert	This operation inserts the element into the priority queue with an associated priority
Peek	This operation returns the element that has the highest priority.
Extract-Maximum or Remove	This operation removes the element from the priority queue that has the highest priority.

Representation of Priority Queue

The priority queue can be represented using a (unsorted or sorted) linked list or array. For an unsorted array or for a linked list, insertion operation is done at the end or the head of the linked list, or at the end of the array, therefore, it runs in $O(1)$. In the peek operation, to return the highest-priority element, it required searching (linear search) the highest-priority element in the entire array or the linked list, therefore it runs in $O(n)$. To remove the highest-priority element, it required searching (linear search) the highest-priority element in the entire array or the linked list. In an array, we also have to shift array contents to fill the gap. Therefore, it runs in $O(n)$.

However, a linked list or unsorted array are great for inserting elements, but not good at searching high-priority elements (or removing them). An alternative is to sort the array (or List or linked list) according to the priority, with the highest priority at the end of the array. For a sorted array or for a linked list, in insertion operation at first it needs to search for the correct position using binary search in an array ($O(\log n)$) and linear search in a linked list ($O(n)$). Once the correct position is found, insert the element thereby keeping the entries in the array or linked list in order (as in insertion sort). In an array, this involves shifting all elements to the right of the inserted element over by one position. Therefore, in either case, the cost is $O(n)$. In peek operation, to return the highest-priority element, it required returning the last element of the array or linked list as the highest-priority element is always at the end. Therefore, it runs in $O(1)$. To remove the highest-priority element, it required removing the last element of the array or linked list as the highest-priority element is always at the end. In the array, by decreasing the size counter. Therefore, it takes $O(1)$.

A sorted array or linked list is fast at looking up and removing high-priority elements, but pays with linear insertion cost.

Priority Queue Representation with Heap

The priority queue can be represented efficiently using a max-heap; the representation would be as follows:

Insert Operation

Since we maintain the property of the complete binary tree, insert the element as a new leaf, as far to the left as possible, i.e. at the end of the array; increment the size of the heap. After insertion, it may violate the heap property when the newly added element has higher priority than its parent. Therefore, to restore the heap condition shift-up through the heap with that element. The running time of the insert operation on a n element heap is $O(\log n)$.

Algorithm to insert an item to Priority Queue

Algorithm: Insert (Q, N, ITEM)

[Q is an array represent priority queue, N is the number of items and ITEM is an item to be inserted into priority queue]

1. Set $N = N + 1$
2. Set $I = N$ and $J = I/2$
3. Repeat step 4 and 5 while $I > 1$ and $Q[J] < ITEM$
4. Set $Q[I] = Q[J]$
5. Set $I = J$ and $J = I/2$

```

    [End of loop]
6. Set Q[I] = ITEM
7. Return

```

Peek Operation

To return the highest-priority element, it required returning the root element of the heap as the highest-priority element will always be at the root of the max-heap. Therefore, it runs in $O(1)$.

Remove Operation

To remove the highest-priority element, it required removing the top element of the heap; decrement the size of the heap, and then shift-down through the heap with that item to restore the heap condition. The running time of remove operation on an n element heap is $O(\log n)$.

Algorithm to delete an item from Priority Queue

Algorithm: Remove (Q, N, ITEM)

[Q is an array represent priority queue, N is the number of items and ITEM is an item to be removed from priority queue]

```

1. [check whether queue is empty]
   If  $N < 1$  then Print: Queue Underflow and Return
2. Set ITEM = Q[1]
3. Set Q[1] = Q[N]
4. Set  $N = N - 1$ 
5. Call Heapify(Q, 1)
6. Return

```

In the following table (Table 6.7) the time complexity of different operations on priority queue are given where different data structures are used. From the table, it is observed that the time complexity of Heap data structure is the best choice for implementing a priority queue.

Table 6.7: Time complexity of different operation with different representation

Operation	Priority queue representation with		
	Unsorted array or linked list	Sorted array or linked list	Heap
Insert	$O(1)$	$O(n)$	$O(\log n)$
Peek	$O(n)$	$O(1)$	$O(1)$
Remove	$O(n)$	$O(1)$	$O(\log n)$

Summary

- A stack is an ordered collection of elements into which new elements may be inserted and from which elements may be deleted only at one end called the top of the stack.
- A queue is a homogeneous collection of elements in which deletions can take place only at the front end, known as dequeue and insertions can take place only at the rear end, known as enqueue.

- A Double-ended Queue (Deque often abbreviated to Deque) is a linear list that generalizes a queue, in which elements can be inserted or deleted from either the front end or from the rear end but not in the middle.
- A Priority Queue is a collection of elements such that each element has been assigned a priority and the elements are arranged on the basis of priority.

Exercises

- What is a queue? Write an algorithm to insert an element in such a queue.
- Why is the queue data structure called the FIFO?
- Define circular queue. Write an algorithm to insert an item in the circular queue.
- What are the disadvantages of the linear queue? How can we overcome these disadvantages in case of the circular queue? Explain with an example.
- What is priority queue? Implement the operations of the priority queue.
- What is input restricted dequeue?
- Write an algorithm to convert an infix expression to its corresponding postfix expression, using the stack.
- Write the differences between stack and queue.
- Write short notes on Dequeue- operations and applications.
- Evaluate following expression.
 - $10+3-2-8/2*6-7$
 - $(12-(2-3)+10/2+4*2)$
- Convert following infix expression to postfix expression:
 - $((a+b)/d-((e-f)+g)$
 - $12/3*6+6-6+8/2$
- Convert following infix expression to prefix expression:
 - $((a+b)/d-((e-f)+g)$
 - $12/3*6+6-6+8/2$
- Explain application of Stack.
- Choose the correct alternative in each of the following:
 - Reverse Polish notation is often known as
 - Infix
 - Prefix
 - Postfix
 - none of these
 - The postfix equivalent of the prefix $*+ab-cd$ is
 - $ab+cd-*$
 - $abcd+-*$
 - $ab+cd*-$
 - $ab+-cd*$
 - The following sequence of operations is performed on a stack: push(1), push(2), pop, push(1), push(2), pop, pop, pop, push(2), pop. The sequence of popped out values are:
 - 2, 2, 1, 1, 2
 - 2, 2, 1, 2, 2
 - 2, 1, 2, 2, 1
 - 2, 1, 2, 2, 2
 - The initial configuration of queue is a, b, c, d (a is at the front). To get the configuration d, c, b, a one needs a minimum of
 - 2 deletions and 3 additions
 - 3 deletions and 2 additions
 - 3 deletions and 3 additions
 - 3 deletions and 4 additions
 - A linear list that allows elements to be added or removed at either end but not in the middle is called:
 - Stack
 - Deque
 - Deque
 - Priority queue

- vi) If we evaluate the following post-fix expression, $23\ 5\ 7\ * -12\ +$, the result will be
 a) 12 b) 0 c) -12 d) 35
- vii) The evaluation of the postfix expression $3\ 5\ 7\ * + 12\ %$ is
 a) 2 b) 3 c) 0 d) 3.17
- viii) The integers 1, 2, 3, 4 are pushed into the stack in that order. They may be popped out of the stack in any valid order. The integers, which are popped out produce a permutation of the members 1, 2, 3, 4. Which of the following permutation can never produce in such a way?
 a) 1, 2, 3, 4 b) 4, 2, 3, 1 c) 4, 3, 2, 1 d) 3, 2, 4, 1
- ix) The prefix expression for the infix expression $a * (b + c) / e - f$ is
 a) $/ * a + bc - ef$ b) $- / * a b c e f$ c) $- / * a + b c e f$ d) none of these
- x) A stack is implemented using an array with the following declaration:
`int stack[100]; int stack_top=0;`
 To perform the POP operation, which of the following is correct?
 a) $x = \text{stack}[\text{stack_top}++]$ b) $x = \text{stack}[++\text{stack_top}]$
 c) $x = \text{stack}[\text{stack_top}--]$ d) $x = \text{stack}[--\text{stack_top}]$
- xi) Translating the infix expression $P = A + (B * C -> D / (E + F) * G) * H$ into postfix notation, we get,
 a) $ABC * DEF / + G * - H +$ b) $ABC * + DEF + / - G * H -$
 c) $ABC * DEF + / G * - H +$ d) None of these
- xii) The number of stacks required to implement mutual recursion is
 a) 3 b) 2 c) 1 d) none of these
- xiii) Queue can be used to implement?
 a) Radix sort b) Quick sort c) Recursion d) Depth first search
- xiv) Stack is useful for implementing
 a) Radix sort b) Recursion c) Breadth first search d) Depth first search
- xv) The postfix expression for the infix expression $A + B * (C + D) / F + D * E$ is
 a) $AB + CD + * F / D + E *$ b) $ABCD + * F / + DE * +$
 c) $A * B + CD / F * DE ++$ d) $A + * BCD / F * DE ++$
- xvi) Stack is sometimes called a _____
 a) Push down list b) Pushdown array c) Pop down list d) Pop up array
- xvii) The prefix expression for the infix expression : $a + b * c / d$
 a) $+ ab * / cd$ b) $+ * ab / cd$ c) $+ a * b / cd$ d) none
- xviii) Which of the following is not the operation on stack?
 a) Push b) Pop c) Peep d) Enqueue
- xix) Which of the following is related to Queue?
 a) Round Robin algorithm b) Traffic Control System c) All d) None
- xx) Which of the following is not a application of Stack?
 a) Evaluation of Police notation b) Tower of Hanoi c) Stack Machine d) None