KIET Group of Institutions, Ghaziabad
Department of Computer Application
(An ISO – 9001: 2015 Certified & 'A' Grade accredited Institution by NAAC)

NOTES

Subject Name: Data Structures & Analysis of Algorithms **Subject Code**: KCA-205 **Semester**: II

UNIT-2

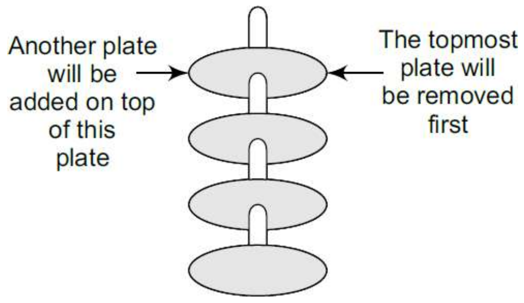
Stacks: Abstract Data Type, Primitive Stack operations: Push & Pop, Array and Linked Implementation of Stack in C, Application of stack: Prefix and Postfix Expressions, Evaluation of postfix expression, Iteration and Recursion- Principles of recursion, Tail recursion, Removal of recursion Problem solving using iteration and recursion with examples such as binary
search, Fibonacci numbers, and Hanoi towers.

Queues: Operations on Queue: Create, Add, Delete, Full and Empty, Circular queues, Array and linked implementation of queues in C, Dequeue and Priority Queue.

Searching: Concept of Searching, Sequential search, Index Sequential Search, Binary Search. Concept of Hashing & Collision resolution Techniques used in Hashing.

What is Stack?

Stack is an important data structure which stores its elements in an ordered manner. You must have seen a pile of plates where one plate is placed on top of another. Now, when you want to remove a plate, you remove the topmost plate first. Hence, you can add and remove an element (i.e., a plate) only at/from one position which is the topmost position.


Another plate will be added on top of this plate
The topmost plate will be removed first

A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

Now the question is **where do we need stacks in computer science**? The answer is in function calls. Stacks can be implemented using either arrays or linked lists. In the following sections, we will discuss both array and linked list implementation of stacks.

ARRAY REPRESENTATION OF STACKS

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.

- If TOP = NULL, then it indicates that the stack is empty and
- if TOP = MAX–1, then the stack is full.

A	AB	ABC	ABCD	ABCDE					
0	1	2	3	**TOP = 4**	5	6	7	8	9

The stack shows that TOP = 4, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

OPERATIONS ON A STACK

A stack supports three basic operations:

- push
- pop
- peek.

- ✓ The push operation adds an element to the top of the stack and
- ✓ the pop operation removes the element from the top of the stack.
- ✓ The peek operation returns the value of the topmost element of the stack.

Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. However, before inserting the value, we must first check if TOP=MAX–1, because if that is the case, then the stack is full and no more insertions can be done. If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.

1	2	3	4	5					
0	1	2	3	**TOP = 4**	5	6	7	8	9

To insert an element with value 6, we first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP]. Thus, the updated stack becomes as below:

1	2	3	4	5	6				
0	1	2	3	4	**TOP = 5**	6	7	8	9

Algorithm to insert an element in a stack

```
Step 1: IF TOP = MAX-1
            PRINT OVERFLOW
            Goto Step 4
[END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

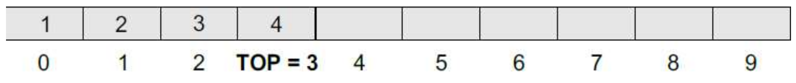
Algorithm shows the algorithm to insert an element in a stack. In Step 1, we first check for the OVERFLOW condition. In Step 2, TOP is incremented so that it points to the next location in the array. In Step 3, the value is stored in the stack at the location pointed by TOP.

Pop Operation

The pop operation is used to delete the topmost element from the stack. However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack given below:

1	2	3	4	5					
0	1	2	3	**TOP = 4**	5	6	7	8	9

To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP. Thus, the updated stack becomes as shown below



```
Step 1: IF TOP = NULL
            PRINT UNDERFLOW
      [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

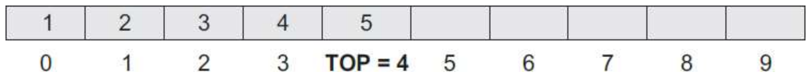
In Step 1, we first check for the UNDERFLOW condition. In Step 2, the value of the location in the stack pointed by TOP is stored in VAL. In Step 3, TOP is decremented.

Peek Operation

Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.

```
Step 1: IF TOP = NULL
            PRINT STACK IS EMPTY
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

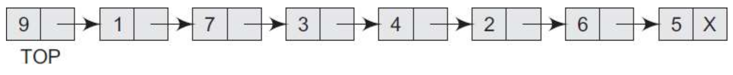
However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned. Consider the stack given below



Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

LINKED REPRESENTATION OF STACKs

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used. The storage requirement of linked representation of the stack with n elements is O(n), and the typical time requirement for the operations is O(1).
In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty. The linked representation of a stack is shown in below:

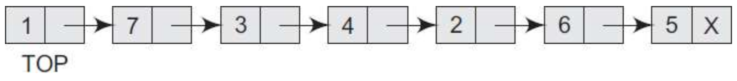


OPERATIONS ON A LINKED STACK

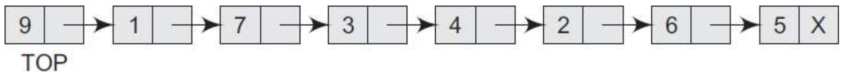
A linked stack supports all the three stack operations, that is, push, pop, and peek.

Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown below:



To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP. Thus, the updated stack becomes as shown below:

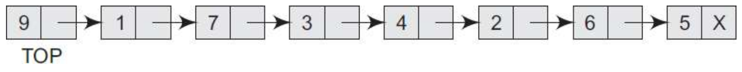


```
Step 1: Allocate memory for the new node and name it as NEW_NODE
Step 2: SET NEW_NODE->DATA=VAL
Step 3: IF TOP = NULL
                SET NEW_NODE->NEXT=NULL
                SET TOP =NEW_NODE
        ELSE
                SET NEW_NODE->NEXT=TOP
                SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

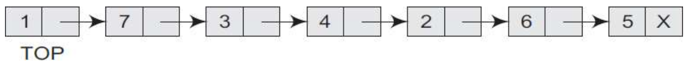
In Step 1, memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked list. This is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

Pop Operation

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack shown below:



In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown below:



```
Step 1: IF TOP = NULL
                PRINT UNDERFLOW
                Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP->NEXT
Step 4: FREE PTR
Step 5: END
```

In Step 1, we first check for the UNDERFLOW condition.
In Step 2, we use a pointer PTR that points to TOP. In Step 3, TOP is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

APPLICATIONS OF STACKS

In this section we will discuss typical problems where stacks can be easily applied for a simple
and efficient solution. The topics that will be discussed in this section include the following:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

Reversing a List

A list of numbers can be reversed by reading each number from an array starting from the first index and pushing it on a stack. Once all the numbers have been read, the numbers can be popped one at a time and then stored in the array starting from the first index.

Implementing Parentheses Checker

Stacks can be used to check the validity of parentheses in any algebraic expression. For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket. For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid. Look at the program below which traverses an algebraic expression to check for its validity.

Evaluation of Arithmetic Expressions

Polish Notations

Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions. But before learning about prefix and postfix notations, let us first see what an infix notation is.

Infix Notation

While writing an arithmetic expression using infix notation, the operator is placed in between the operands. For example, A+B; here, plus operator is placed between the two operands A and B. Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer needs a lot of information to evaluate the expression. Information is needed about operator precedence and associativity rules, and brackets which override these rules. So, computers work more efficiently with expressions written using prefix and postfix notations.

Postfix Notation

In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as A+B in infix notation, the same expression can be written as AB+ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation. The expression (A + B) * C can be written as:

[AB+]*C = AB+C* in the postfix notation

Example: Convert the following infix expressions into postfix expressions.

(a) (A–B) * (C+D)
[AB–] * [CD+]
AB–CD+*

(b) (A + B) / (C + D) – (D * E)
[AB+] / [CD+] – [DE*]
[AB+CD+/] – [DE*]
AB+CD+/DE*–

A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands. For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication.

Thus we see that in a postfix notation, operators are applied to the operands that are immediately left to them. In the example, AB+C*, + is applied on A and B, then * is applied on the result of addition and C.

Prefix Notation

Prefix notation is also evaluated from left to right, the only difference between a postfix notation and a prefix notation is that in a prefix notation, the operator is placed before the operands. For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.

While evaluating a prefix expression, the operators are applied to the operands that are present immediately on the right of the operator. Like postfix, prefix expressions also do not follow the rules of operator precedence and associativity, and even brackets cannot alter the order of evaluation.

Convert the following infix expressions into prefix expressions.

(a) (A + B) * C
(+AB)*C
*+ABC

(b) (A–B) * (C+D)
[–AB] * [+CD]
*–AB+CD

(c) (A + B) / (C + D) – (D * E)
[+AB] / [+CD] – [*DE]
[/+AB+CD] – [*DE]
–/+AB+CD*DE

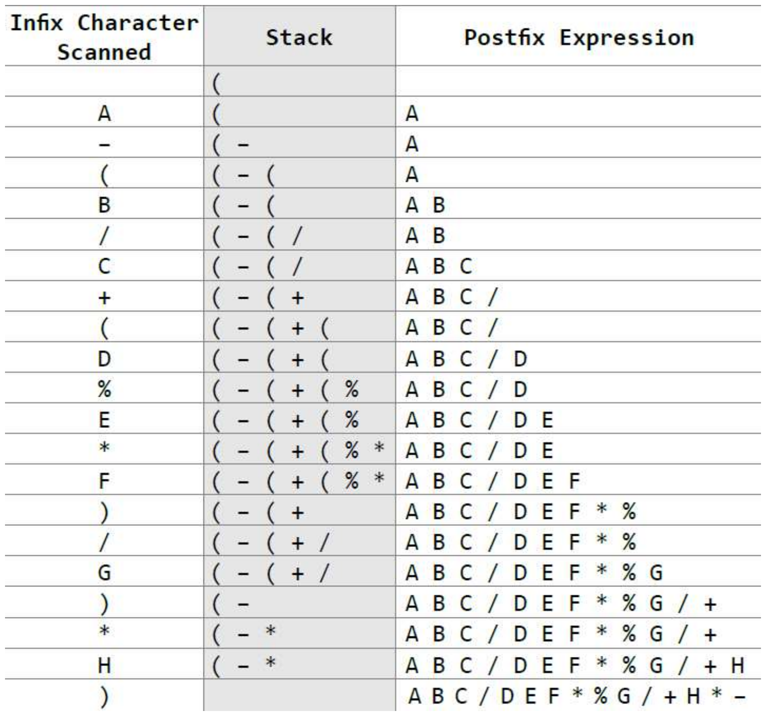
Conversion of an Infix Expression into a Postfix Expression

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only +, –, *, /, % operators. The precedence of these operators can be given
as follows:
Higher priority *, /, %
Lower priority +, –
No doubt, the order of evaluation of these operators can be changed by making use of parentheses. For example, if we have an expression A + B * C, then first B * C will be done and the result will be added to A. But the same expression if written as, (A + B) * C, will evaluate A + B first and then the result will be multiplied with C.

The **algorithm** given below transforms an infix expression into postfix expression:

```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
        IF a ( is encountered, push it on the stack
        IF an operand (whether a digit or a character) is encountered, add it to the postfix
        expression.
        IF a ")" is encountered, then
        a. Repeatedly pop from stack and add it to the postfix expression until a ( is
           encountered.
        b. Discard the ( . That is, remove the ( from stack and do not add it to the postfix
           expression
          IF an operator 0 is encountered, then
        a. Repeatedly pop from stack and add each operator (popped from the stack) to the
           postfix expression which has the same precedence or a higher precedence than 0
        b. Push the operator 0 to the stack
        [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is
        empty
Step 5: EXIT
```

Convert the following infix expression into postfix expression
(a) A – (B / C + (D % E * F) / G)* H



Infix Character Scanned	Stack	Postfix Expression
	(	
A	(	A
-	(-	A
(	(- (	A
B	(- (	A B
/	(- (/	A B
C	(- (/	A B C
+	(- (+	A B C /
(	(- (+ (	A B C /
D	(- (+ (	A B C / D
%	(- (+ (%	A B C / D
E	(- (+ (%	A B C / D E
*	(- (+ (% *	A B C / D E
F	(- (+ (% *	A B C / D E F
)	(- (+	A B C / D E F * %
/	(- (+ /	A B C / D E F * %
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

Evaluation of a Postfix Expression

PPT

Conversion of an Infix Expression into a Prefix Expression

The corresponding prefix expression is obtained in the operand stack.
For example, given an infix expression (A – B / C) * (A / K – L)

Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.

(L – K / A) * (C / B – A)

Step 2: Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1.

The expression is: (L – K / A) * (C / B – A)
Therefore, [L – (K A /)] * [(C B /) – A]
= [LKA/–] * [CB/A–]
= L K A / – C B / A – *

Step 3: Reverse the postfix expression to get the prefix expression. Therefore, the prefix expression is
*** – A / B C – /A K L**

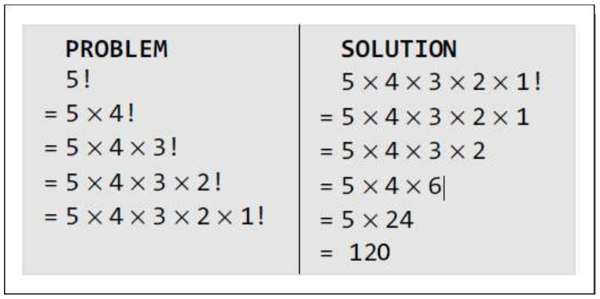
Evaluation of a Prefix Expression

PPT

Recursion

A *recursive function* is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.

To understand recursive functions, let us take an example of calculating factorial of a number. To calculate n!, we multiply the number with factorial of the number that is 1 less than that number.
In other words, n! = n * (n–1)!
Let us say we need to find the value of 5!
5! = 5 * 4 * 3 * 2 * 1
= 120
This can be written as
5! = 5 * 4!, where 4!= 4 * 3!
Therefore,
5! = 5 * 4 * 3!
Similarly, we can also write,
5! = 5 * 4 * 3 * 2!
Expanding further
5! = 5 * 4 * 3 * 2 * 1!
We know, 1! = 1



PROBLEM	SOLUTION
5!	5 × 4 × 3 × 2 × 1!
= 5 × 4!	= 5 × 4 × 3 × 2 × 1
= 5 × 4 × 3!	= 5 × 4 × 3 × 2
= 5 × 4 × 3 × 2!	= 5 × 4 × 6
= 5 × 4 × 3 × 2 × 1!	= 5 × 24
	= 120

Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. every recursive function must have a base case and a recursive case. For the factorial function,

Base case is when n = 1, because if n = 1, the result will be 1 as 1! = 1.

Recursive case of the factorial function will call itself but with a smaller value of n, this case can be given as
factorial(n) = n × factorial (n–1)

Greatest Common Divisor

The greatest common divisor of two numbers (integers) is the largest integer that divides both the numbers. We can find the GCD of two numbers recursively by using the Euclid's algorithm that states

```
GCD (a, b) = b, if b divides a
        GCD (b, a mod b), otherwise
```

GCD can be implemented as a recursive function because if b does not divide a, then we call the same function (GCD) with another set of parameters that are smaller than the original ones.

Here we assume that a > b. However if a < b, then interchange a and b in the formula given above.

Working

```
Assume a = 62 and b = 8
GCD(62, 8)
        rem = 62 % 8 = 6
        GCD(8, 6)
                rem = 8 % 6 = 2
                GCD(6, 2)
                        rem = 6 % 2 = 0
                Return 2
        Return 2
Return 2
```

Types of Recursion

Recursion is a technique that breaks a problem into one or more sub-problems that are similar to the original problem. Any recursive function can be characterized based on:

- whether the function calls itself directly or indirectly (*direct or indirect recursion*)
- whether any operation is pending at each recursive call (*tail recursive* or not)

Direct Recursion

A function is said to be directly recursive if it explicitly calls itself. For example, consider the code shown below. Here, the function Func() calls itself for all positive values of n, so it is said to be a directly recursive function.

```
int Func (int n)
{
        if (n == 0)
                return n;
        else
                return (Func (n–1));
}
```

Indirect Recursion

A function is said to be indirectly recursive if it contains a call to another function which ultimately calls it. Look at the functions given below. These two functions are indirectly recursive as they both call each other

```
int Funcl (int n)
{
        if (n == 0)
                return n;
        else
                return Func2(n);
}

int Func2(int x)
{
        return Func1(x–1);
}
```

Tail Recursion

A recursive function is said to be *tail recursive* if no operations are pending to be performed when the recursive function returns to its caller. when the called function returns, the returned value is immediately returned from the calling function. Tail recursive functions are highly desirable because they are much more efficient to use as the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

Non-Tail Recursive Functions

```
int Fact(int n)
{
if (n == 1)
        return 1;
else
        return (n * Fact(n–1));
}
```

In above algo, the factorial function that we have written is a nontail- recursive function, because there is a pending operation of multiplication to be performed on return from each recursive call.

However, the same factorial function can be written in a tail recursive manner as shown below:

Tail Recursive Functions

```
int Fact(n)
{
        return Fact1(n, 1);
}
int Fact1(int n, int res)
{
        if (n == 1)
                return res;
        else
                return Fact1(n–1, n*res);
}
```

Tower of Hanoi

PPT

Points to Remember

- A stack is a linear data structure in which elements are added and removed only from one end, which is called the top. Hence, a stack is called a LIFO (Last-In, First-Out) data structure as the element that is inserted last is the first one to be taken out.

- In the computer's memory, stacks can be implemented using either linked lists or single arrays.

- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself. They are implemented using system stack.

QUEUE

What is Queue?

A queue is a Data Structure in which element at the first position is served first.
Example:

- ➢ People moving on an escalator. The people who got on the escalator first will be the first one to step out of it.
- ➢ People waiting for a bus. The first person standing in the line will be the first one to get into the bus.
- ➢ People standing outside the ticketing window of a cinema hall. The first person in the line will get the ticket first and thus will be the first one to move out of it.

A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT. Queues can be implemented by using either arrays or linked lists.

ARRAY REPRESENTATION OF QUEUEs

Queues can be easily represented using linear arrays. As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively. The array representation of a queue is shown below:

12	9	7	18	14	36				
0	1	2	3	4	5	6	7	8	9

Operations on Queues

Insertion

In above figure, FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR. The queue after addition is shown below:

12	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Here, FRONT = 0 and REAR = 6.

Before inserting an element in a queue, we must check for overflow conditions. An overflow will occur when we try to insert an element into a queue that is already full.
When REAR = MAX – 1, where MAX is the size of the queue, we have an overflow condition. Note that we have written MAX – 1 because the index starts from 0.

```
Step 1: IF REAR = MAX-1
                Write OVERFLOW
                Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
                SET FRONT = REAR =
        ELSE
                SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Deletion

If we want to delete an element from the queue, then the value of FRONT will be incremented. Deletions are done from only this end of the queue. The queue after deletion will be as shown below:

	9	7	18	14	36	45			
0	1	2	3	4	5	6	7	8	9

Here, FRONT = 1 and REAR = 6.

Before deleting an element from a queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If FRONT = –1 and REAR = –1, it means there is no element in the queue.

```
Step 1: IF FRONT = -1 OR FRONT > REAR
                Write UNDERFLOW
        ELSE
                SET VAL=QUEUE[FRONT]
                SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

LINKED REPRESENTATION OF QUEUEs

We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size. If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted. And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.

In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.

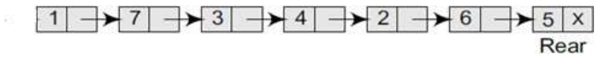
In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty.

Operations on Linked Queues

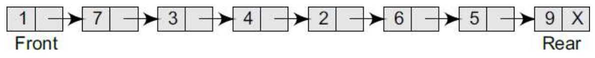
A queue has two basic operations: insert and delete. The insert operation adds an element to the end of the queue, and the delete operation removes an element from the front or the start of the queue. Apart from this, there is another operation peek which returns the value of the first element of the queue.

Insert Operation

The insert operation is used to insert an element into a queue. The new element is added as the last element of the queue. Consider the linked queue shown below:



To insert an element with value 9, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part. The new node will then be called both FRONT and rear. However, if FRONT!= NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear. Thus, the updated queue becomes as shown below:

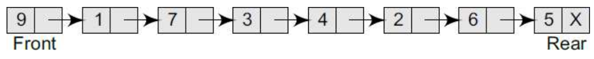


```
Step 1: Allocate memory for the new node and name it as PTR
Step 2: SET PTR->DATA = VAL
Step 3: IF FRONT = NULL
                SET FRONT = REAR = PTR
                SET FRONT->NEXT = REAR->NEXT = NULL
        ELSE
                SET REAR->NEXT = PTR
                SET REAR = PTR
                SET REAR->NEXT = NULL
        [END OF IF]
Step 4: END
```

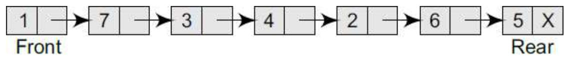
Above algorithm shows the insertion an element in a linked queue. In Step 1, the memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked queue. This is done by checking if FRONT = NULL. If this is the case, then the new node is tagged as FRONT as well as REAR. Also NULL is stored in the NEXT part of the node (which is also the FRONT and the REAR node). However, if the new node is not the first node in the list, then it is added at the REAR end of the linked queue (or the last node of the queue).

Delete Operation

The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT. However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed. Consider the queue shown below:



To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue. Thus, the updated queue becomes as shown below:



```
Step 1: IF FRONT = NULL
                Write Underflow
                Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

In Step 1, we first check for the underflow condition. If the condition is true, then an appropriate message is displayed, otherwise in Step 2, we use a pointer PTR that points to FRONT. In Step 3, FRONT is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

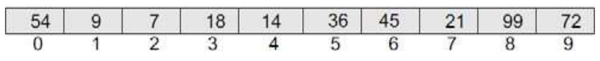
TYPES OF QUEUES

A queue data structure can be classified into the following types:

1. Circular Queue
2. Deque
3. Priority Queue
4. Multiple Queue

Circular Queues

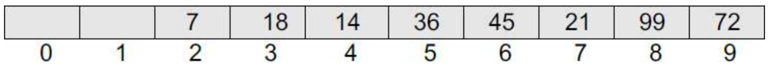
In linear queues, we have discussed so far that insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT. Look at the queue shown below:



Here, FRONT = 0 and REAR = 9.

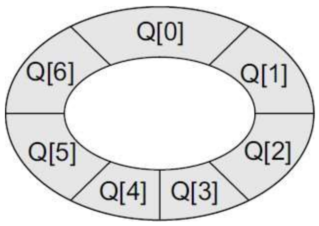
Insertion

Now, if you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted. Consider a scenario in which two successive deletions are made. The queue will then be given as shown below:



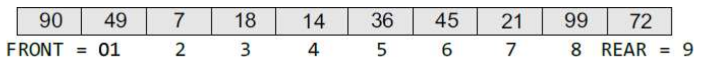
Here, front = 2 and REAR = 9.
Suppose we want to insert a new element in the queue shown above. Even though there is space available, the overflow condition still exists because the condition
rear = MAX – 1 still holds true. This is a major drawback of a linear queue.

To resolve this problem, we have two solutions. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But this can be very time-consuming, especially when the queue is quite large. The second option is to use a circular queue. In the circular queue, the first index comes right after the last index. Conceptually, you can think of a circular queue as shown below:

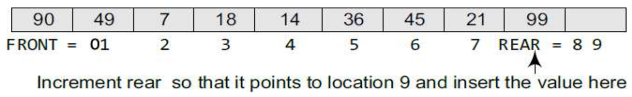


The circular queue will be full only when front = 0 and rear = Max – 1. A circular queue is implemented in the same manner as a linear queue is implemented. The only difference will be in the code that performs insertion and deletion operations. For insertion, we now have to check for the following three conditions:

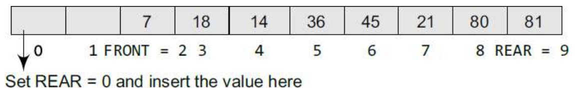
If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given below which illustrates this point.



If rear != MAX – 1, then rear will be incremented and the value will be inserted as below:


90 49 7 18 14 36 45 21 99
FRONT = 01 2 3 4 5 6 7 REAR = 8 9
Increment rear so that it points to location 9 and insert the value here

If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there, as shown below:



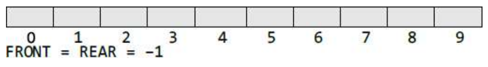
```
Step 1: IF FRONT = 0 and Rear = MAX - 1
            Write OVERFLOW
            Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR =0
        ELSE IF REAR = MAX - 1 and FRONT !=0
            SET REAR =0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

In Step 1, we check for the overflow condition. In Step 2, we make two checks. First to see if the queue is empty, and second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end. In Step 3, the value is stored in the queue at the location pointed by REAR.

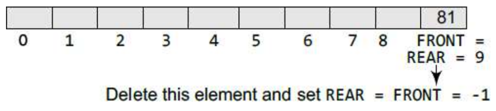
Deletion

To delete an element, again we check for three conditions.

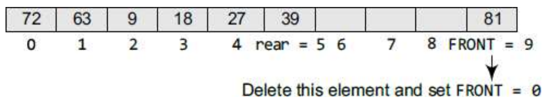
If front = –1, then there are no elements in the queue. So, an underflow condition will be reported. Figure shown below shows underflow.



If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1. This is shown below:



If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0. This is shown below:

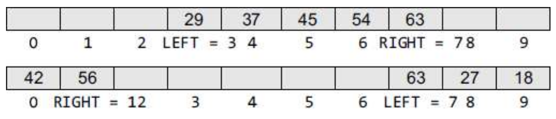


```
Step 1: IF FRONT = -1
                Write UNDERFLOW
                Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
                SET FRONT = REAR = -1
        ELSE
                IF FRONT = MAX -1
                SET FRONT = 0
        ELSE
                SET FRONT = FRONT + 1
        [END of IF]
        [END OF IF]
Step 4: EXIT
```

Deques

A deque is a list in which the elements can be inserted or deleted at either end. It is also known as a *head-tail linked list* because elements can be added to or removed from either the front (head) or the back (tail) end.

However, no element can be added and deleted from the middle. In the computer's memory, a deque is implemented using either a circular array or a circular doubly linked list. In a deque, two pointers are maintained, LEFT and RIGHT, which point to either end of the deque. The elements in a deque extend from the LEFT end to the RIGHT end and since it is circular, Dequeue[N–1] is followed by Dequeue[0]. Consider the deques shown below.


29
37
45
54
63
0 1 2 LEFT = 3 4 5 6 RIGHT = 7 8 9
42
56
63
27
18
0 RIGHT = 1 2 3 4 5 6 LEFT = 7 8 9

There are two variants of a double-ended queue. They include

Input restricted deque In this dequeue, insertions can be done only at one of the ends, while deletions can be done from both ends.

Output restricted deque In this dequeue, deletions can be done only at one of the ends, while insertions can be done on both ends.

Priority Queues

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed. The general rules of processing the elements of a priority queue are

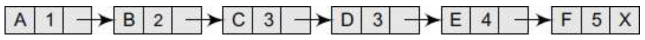
- An element with higher priority is processed before an element with a lower priority.

- Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

A priority queue can be thought of as a modified queue in which when an element has to be removed from the queue, the one with the highest-priority is retrieved first. The priority of the element can be set based on various factors. Priority queues are widely used in operating systems to execute the highest priority process first. The priority of the process may be set based on the CPU time it requires to get executed completely. For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed. However, CPU time is not the only factor that determines the priority, rather it is just one among several factors. Another factor is the importance of one process over another. In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

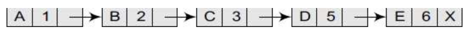
Implementation of a Priority Queue
There are two ways to implement a priority queue. We can either use a sorted list to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority or we can use an unsorted list so that insertions are always done at the end of the list. Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed.

Linked Representation of a Priority Queue
In the computer memory, a priority queue can be represented using arrays or linked lists. When a priority queue is implemented using a linked list, then every node of the list will have three parts:
(a) the information or data part,
(b) the priority number of the element, and
(c) the address of the next element. If we are using a sorted linked list, then the element with the higher priority will precede the element with the lower priority. Consider the priority queue shown below:

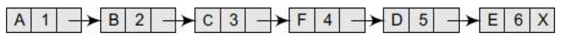


Lower priority number means higher priority. For example, if there are two elements A and B, where A has a priority number 1 and B has a priority number 5, then A will be processed before B as it has higher priority than B. The priority queue shown above is a sorted priority queue having six elements. From the queue, we cannot make out whether A was inserted before E or whether E joined the queue before A because the list is not sorted based on FCFS. Here, the element with a higher priority comes before the element with a lower priority. However, we can definitely say that C was inserted in the queue before D because when two elements have the same priority the elements are arranged and processed on FCFS principle.

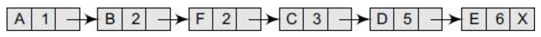
Insertion When a new element has to be inserted in a priority queue, we have to traverse the entire list until we find a node that has a priority lower than that of the new element. The new node is inserted before the node with the lower priority. However, if there exists an element that has the same priority as the new element, the new element is inserted after that element. For example, consider the priority queue shown below:



If we have to insert a new element with data = F and priority number = 4, then the element will be inserted before D that has priority number 5, which is lower priority than that of the new element. So, the priority queue now becomes as shown below:



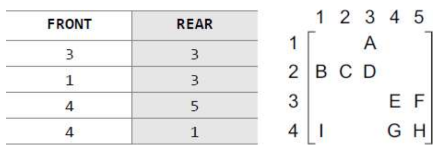
However, if we have a new element with data = F and priority number = 2, then the element will be inserted after B, as both these elements have the same priority but the insertions are done on FCFS basis as shown below:



Deletion: Deletion is a very simple process in this case. The first node of the list will be deleted and the data of that node will be processed first.

Array Representation of a Priority Queue

When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers. We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space. Look at the two-dimensional representation of a priority queue given below. Given the front and rear values of each queue, the two-dimensional matrix can be formed as shown below:

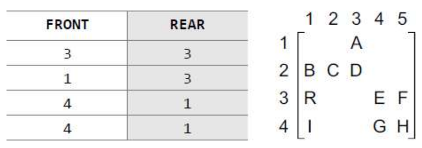


FRONT	REAR
3	3
1	3
4	5
4	1

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & & & A & & \\ 2 & B & C & D & & \\ 3 & & & & E & F \\ 4 & I & & & G & H \end{array}$$

FRONT[K] and REAR[K] contain the front and rear values of row K, where K is the priority number. Note that here we are assuming that the row and column indices start from 1, not 0. Obviously, while programming, we will not take such assumptions.

Insertion To insert a new element with priority K in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element. For example, if we have to insert an element R with priority number 3, then the priority queue will be given as shown below:



FRONT	REAR
3	3
1	3
4	1
4	1

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & & & A & & \\ 2 & B & C & D & & \\ 3 & R & & & E & F \\ 4 & I & & & G & H \end{array}$$

Deletion To delete an element, we find the first nonempty queue and then process the front element of the first non-empty queue. In our priority queue, the first non-empty queue is the one with priority number 1 and the front element is A, so A will be deleted and processed first. In technical terms, find the element with the smallest K, such that FRONT[K] != NULL.

APPLICATIONs OF QUEUES

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.