Syllabus (University of Rajasthan)

Data Structure (Using C/C++)

Code-301

Max Marks: 100

Part-I (Very Short Answer) consists 10 questions of **two marks** each with two questions form each unit. Maximum limit for each question is up to 40 words.

Part-II (Short Answer) consists 5 questions of **four marks** each with one question from each unit. Maximum limit for each question is up to 80 words.

Part-III (Long Answer) consists 5 questions of **twelve marks** each with one question from each unit with internal choice.

UNIT – I

Introduction to Algorithm Design: Algorithm, its characteristics, efficiency of algorithms, analyzing Algorithms and problems.

Linear Structure: Arrays, records, stack, operation on stack, implementation of stack as an array, queue, types of queues, operations on queue, implementation of queue.

UNIT– II

Linked Structure: List representation, Polish notations, operations on linked list –get node and free node operation, implementing the list operation. inserting into an ordered linked list, deleting, circular linked list, doubly linked list, implementation of stack and queues using linked list.

UNIT – III

Tree Structure: Concept and terminology, Types of tress, Binary search tree, inserting, deleting and searching into binary search tree, implementing the insert , search and delete algorithms, tree traversals, Huffman's algorithm.

Contents

UNIT – I

Chapter ▸ 1

Introduction

An information is a collection of data that has been translated into a form that is more convenient to move or to process or can say data are simply values or sets of values. For e.g. an student's name is a data which may be divided into further subitems-first name, middle name and last name. Collections of data are frequently organized in a hierarchy of fields, records and files. A field is an item of stored data field could be name, a date, an address, a description, an quantity etc. A record is the collection of fields that relate to a single entity (means something that has certain attributes or properties which may be assigned values) for e.g. we could have a student record that includes fields for the students name, address, date of birth etc. Records can be classified into fixed length record and Variable-length records. In fixed-length records, all records contains the same data items with the some amount of data space assigned to each data item. In variable-length records, file records may contains different lengths. Student records usually have variable length since different students take different number of courses. A file is a collection of related records. For example a student file might include all of the records of student enrolled in a school. Within a file all records have the same structures. That is every record in the file contains the same fields. Only the data stored in the fields of different record will be different.

1.1 What is Data Type?

A data type is a set of values (e.g. integer, boolean, float). A Data type is type or collection of operations that manipulates on type.

A data item or element is a piece of information or a record. A data item is said to be member of data type. A simple data item contain no subparts (e.g. integer). An aggregated data item may contain several piece of information (e.g. Payroll record, city database record).

1.1.1 Abstract Data Type

Abstract data type is set of values and associated operations that may be performed on that values. Abstract data structure or type is define indirectly, only by the operations that may be performed on it and by mathe-matical constraints on effects of those operations". The classic example of Abstract data types is set of integers and associate operations that may be performed on integers such as addition, substraction, multiplication etc.

1.2 Data Structure

There is a concept of data management which is a complex task that includes activities like data collection, organization of data into appropriate structure and developing and maintaining routines for quality assurance.

A data structure is an organization of the data to solve a problem in such a way that data can be access efficiently by a problem. Program is an implentation of an algorithm in some programming language Algorithm is just an outline, the essence of a computational procedure step by step instruction.

Data structures are used in almost every program or software system. Some common example of data structure are arrays, linked lists, queues, stacks, binary tree and hash tables. Data structures are widely applied in the following area:

- Compiler design
- Operating System
- Statistical Analysis
- Artificial Intelligence
- DBMS
- Simulation
- Graphics

1.3 Elementary Data Structures Organisation

Data structure is basic building blocks of program. A program built using improper data structure may not work as expected. So its mandatory to choose most appropriate data structure according to need.

1.4 Classification of Data Structure

As we discussed above any thing that store data is called data structure hence integer, float, Boolean,char, pointer etc all are data structure. They are known as Primitive data structures or Built-in-data type. But often these limited datatypes aren't enough and a programmer wants to build their own datatypes. Users can define their own data types to handle such limitations of buit-in-type data structures and are called User-defined Data structures.

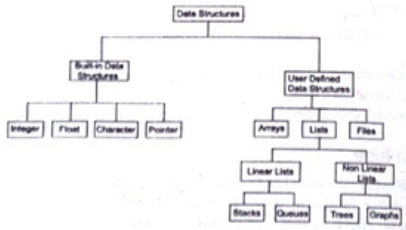


Figure 1.1 Classification of Data Structures Organisation

1.4.1 Built in Data structure

- Integer -This data structure used to store integeral data type values may be of different size and may be not be allowed to contain negative values.
- Float
- Character
- Pointer

1.4.2 User-defined Data structure

a) **Arrays:** An array is collection of similar data types store in common variable. The collection forms a data structure where data items are stored linearly one after another in memory. An array has a set of homogeneous data items. Each elements is reference by index or subscript. For e.g. A[1], A[2], A[3]...A[N]. Index is usually a number to address the item in array.

b) **Lists:** Array has a major drawback you must know the maximum number of item in your array when you create it.This presents problem in program in which the maximum number of item cannot be predicted accurately when the program starts up. So we use a structure called List to overcome this problem. List is very flexible dynamic data structure; item may be added or delete from it .A programmer need not worry about how many item will program be accommodate , this allows us to write robust program which require less maintence. A very common source of program maintence is need to increase the capacity of a program to handle larger collection of data. In List each item has allocated space as it is added to List. List is further categories in two type according to their structure (a) Linear List (b) Non-Linear List

- **Linear List:** A linear data structure traverses the data element sequentially in which only one data item can directly be reached. Linked List, Stack, Queue etc.

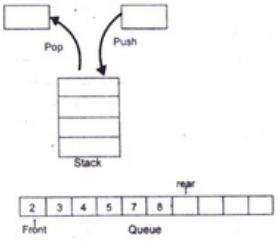
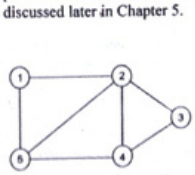


Figure 1.2 User-defined Data structure

- **Non Linear List:** Every data item is attached to several other data items in a way that is specific for reflecting relationship.The data items are not sequentially arranged. Moreover removing one of the links could divide the data structure in two disjoint pieces. e.g. Trees, Graphs etc.
- **Tree:** Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree or simply tree. Tree will discussed later in Chapter 4.
- **Graph:** A graph is a datastructure consist of finite set of vertices or nodes, connected by edges. Graph will be discussed later in Chapter 5.



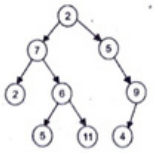


Figure 1.3 Graph Tree

(c) **Files:** File is collection of records. A student record may comprise item of information such as student's name, roll no, registration number, age, grade etc. Individual items of information in record are often called as field of record.

Operations on file:

- Create a file
- Insert a record
- Search a record
- Delete a record
- Modify a record
- List a record

File is organized logically as sequence of record. These records are mapped into disk blocks. Records are organized in file by following method:

(1) **Heap File Organization:** Any record can be placed anywhere in the file where there is space for record. There no specific ordering of record. Typically there is only single file for any relation.

(2) **Sequential File Organization:** Records are stored in sequential order according to their "search key" of each record.

(3) **Hashed File Organization:** A hash function is computed on some other attribute of each record. The result of hash function specifies in which block of the file record should be places.

1.5 Algorithm Design of Complexity

An algorithm is defined as step-by-step procedure or method for solving a problem by a computer in a finite number of steps. Steps of an algorithm definitions may include branching or repetition depending upon what problem the algorithm is being developed for while defining an algorithm steps are written in human understandable language and independent of any programming language.

1.6 Important feature of a working algorithm are as follow

1. Finiteness: An algorithm must always terminate after a finite number of steps.
2. Definiteness: Each step of an algorithm must be precisely defined the actions to be carried out must be rigorously and unambiguously specified for each case.
3. Input: An algorithm has zero or more inputs, i.e. quantities which are given to it initially before the algorithm begins.
4. Ouput: An algorithm has one or more outputs i.e. quantities which have a specifical relation to the inputs.
5. Effectiveness: An algorithm is also generally expected take effective. This means that all of the operations to be performed in the algorithm must be sufficiently basic that they can in principle be done exactly and in a finite length of time.

In practice We not only want algorithm but also we want good algorithm in some loosely-defined a esthetic sense. One criteria of goodness is the length of time taken to perform the algorithm termed as time complexity.

1.7 Algorithm Efficiency

There are many approaches (algorithms) to solve a problem. How do we choose among them?

Before designing an algorithm our main aim are:

(1) To design an algorithm that is easy to understand code and debug.
(2) To design an algorithm that to make efficient use of computer's resources.

1.8 Different Approaches to Designing an Algorithm

A complex algorithm is often divided into smaller units called modules. This process of dividing algorithm into modules is called modularization that makes the complex algorithm simpler to design and implement.

There are two approaches to design an algorithm - top-down approach and bottom-up approach.

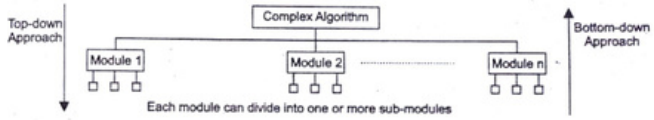


Figure 1.4 Different Approaches to Designing an Algorithm

1.8.1 Top-down Approach

Top-down design approach start by dividing the complex algorithm into one or more modules. These modules can further be decomposed into one or more sub-modules, their process of decomposition is iterated until the desired level of module complexity is achieved.

1.8.2 Bottom-up Approach

Reverse of top-down approach. Here we start with desiging the most basic or concrete modules and then procced towards designing higher level modules. In this approach sub-modules are grouped together to form a higher level module. All the higher level modules are clubbed together to form even higher level modules.

1.9 Complexity of an algorithm

Complexity of an algorithm is a measure of the amount of time /space required by an algorithm for an input of a given size.

There are two main complexity measures of the efficiency of an algorithm.

Time complexity: It describe the amount of time an algorithm takes in term s of the amount of input of an algorithm. The required time is expressed by the number of constant time operation needed to complete the algorithm.

Space complexity: It describe the amount of memory required during the program execution as the function input. Generally the space needed.

1.9.1 Worst-case, Average-case, Best-case and Amortised Time complexity

Maximum number of steps(longest running time) taken to perform an operation to complete the algorithm. In other words, the worst-case complexity measures the resources(i.e. running time, memory) an algorithm require the worst case. It gives the upper bound on the resources required by the algorithm.

Best-case: The best-case complexity of the algorithm is the function defined by the minimum number of steps taken to perform an operation to complete the algorithm.

Average-case: The average case of complexity is to complete the operation on algorithm within average number of steps. Average case running time assumes that the input of a given size are equally likely.

Amortised Time Complexity: Amortised running time refers to the time required to perform a sequence of (related) operation averaged over all operations performed. Amortized analysis guarantees the average performance of each operation in the worst case.

1.9.2 Time –Space Trade-Off

The best algorithm is one which require less memory space and takes less time to complete its execution. But practically desigining such an algorithm is not trivial task.There can be more than one algorithm to solve a particular problem. One may require less memory space, while other may require less CPU time to execute.Thus it is uncommon to sacrifice one thing for the other. Hence there exists a time-space trade-off among algorithms.

1.9.3 Big O Notation

The Big O notation, where O stands for 'order of' is concerned with what happens for large values of n. If f(n) and g(n) are the functions defined on a positive integer number n, then

f(n) = Og((n))

That is f of n is Big-O of g of n if and only if positive constants c and n exist, such that f(n)≤ cg(n). It means that for a large amounts of data f(n) will grow no more than a constant factor than g(n). Hence g provides an upper bound. Note that here c is a constant which depends on the following factors:

- The programming language used.
- The quality of the compiler or interpreter.
- The CPU speed.
- The size of the main memory and the access time to it.
- The knowledge of the programmer and
- The algorithm itself which may require simple but also time-consuming machine instructions.

Big O notation provide a strict upper bound for f(n). This mean that the function f(n) can do better but not worse than the specified value . Big O notation is simply written as f(n) ϵ O(g(n)) or as f(n)=O(g(n)).

F(n) ≤ cg(n) , c>0, n≥ n_0 , then f(n)=O(g(n)) and g(n) is an asymptotically tight upper bound for f(n).

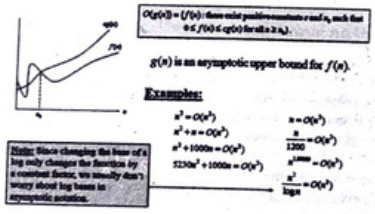


Figure 1.5 The (Big) O Notation

Categories of Algorithm

According to the Big O notation we have five different categories of algorithm:

- Constant time algorithm: running time complexity given as O(1).
- Linear time algorithm : running time complexity given as O(n).
- Logarithmic time algorithm: running time complexity given as O(log n)
- Polynomial time algorithm : running time complexity given as O(k^n) where k>1
- Exponential time algorithm : running time complexity given as O(2^n).

Example: Show that n = O(n log n)

Solution: By definition , we have

O≤ h(n) ≤ cg(n)

Substituting n as h(n) and nlogn as g(n)

O≤ n≤ nlogn

Dividing by nlogn, we get

O/n log n ≤ n/n log n ≤ c n log n/n log n

O ≤ 1/log n ≤ c

We know that 1/log n ->0 as n-> ∞

To determine the value of c , it is clearly evident that 1/nlogn is greatest when n=2. Therefore ,

O≤ 1/log2≤c=1. Hence c=1

To determine the value n0 we can write

O ≤ 1/log n_0 ≤ 1

O≤ 1 ≤ log n_0

Now log n_0 =1 , when n_0=2

Hence , O≤ n ≤ cn logn when c=1 and ¥ n ≥ n_0=2.

Limitations:

Big O faces certain limitation which as shown below:

- Many algorithm are simply too hard to analyse mathematically.
- There may not be sufficient information to calculate the behavior of the algorithm in the average case.
- Big O analysis only tells us how the algorithm grows with the size of the problem not how efficient it is as it does not consider the programming effort.
- It ignores important constants for example if one algorithm take O(n^2) time to execute and the other takes O(100000n^2) time to execute , then as per Big O notataion both algorithm have equal time complexity. In real-time systems this may be serious consideration.

1.10 Omega Notations

The Omega notation provides a tight lower bound for f(n). This means that the function can never do better than the specified value but it may do worse.

Ω notation is simply written as , f(n) ϵ Ω (g(n)), where n is the problem size and

Ω (g(n)) = {h (n): positive constants c>0, n_0 such that 0≤cg(n) ≤h(n), n ≥n_0}.

Example of functions in Ω(n^2) include: n^2, $n^{2.9}$, n^3+n^2, n^3

Example of functions in Ω(n^3) include: n, $n^{2.9}$, n^2

Example: Show that $5n^2$+10n = Ω (n^2)

Solution By definition , we have

O≤cg(n) ≤ h(n)

Substituting n^2 as g(n) and h(n) as $5n^2$+10n

O≤ n^2 ≤ $5n^2$+10n

Dividing by n^2

0/n^2≤ cn^2/n^2 ≤ $5n^2$/n^2+10n/n^2

0≤ c ≤ 5+10/n

Now, lim 5+10/n=5

Therefore,0≤c≤5,

Hence ,c=5

Now to determine the value n_0

$0 \le 5 \le 5+10/n_0$

$-5 \le 5-5 \le 5+10/n_0-5$

$-5 \le 0 \le 10/n_0$

So $n_0=1$ as lim $1/n=0$

Hence, $5n^2+10n = \Omega(n^2)$ for $c=5$

1.11 Theta Notations (Θ)

Theta notation provides an asymptotically tight bound for f(n). Θ notation is simply written as,

$f(n) \in \Theta(g(n))$, where n is the problem size and

$\Theta(g(n)) = \{h(n)$: positive constant c_1, c_2 and n_0 such that $0 \le c_1 g(n) \le h(n) \le c_2 g(n)$, $n \ge n_0\}$.

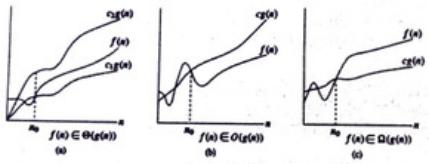


Figure 1.6 Theta Notations (Θ)

Hence we can say that Θ(g(n)) comprises a set of all the functions h(n) that are between $c_1g(n)$ and $c_2g(n)$ for all values), $n \ge n_0$

- The best case in Θ notation is not used .
- Worst case Θ describes asymptotic bounds for worst case combination of input values.
- If we simply write Θ it means same as worst case Θ.

1.12 Other Useful Notations

There are other useful notatations like little o notation and little ώ notation which have been discussed below:

(1) Little o Notation

This notation provides a non-symptotically tight upper bound for f(n). To express a function using this notation we write

$f(n) \in o(g(n))$ where

$o(g(n))=\{h(n)$: positive constants c, n_0 such that for any c?0, $n_0>0$, and $0 \le h(n) \le cg(n)$, $n \ge n_0\}$

Example of functions in $o(n^3)$ include: $n^{2.9}$, $n^3/\log n$, $2n^2$

Example of functions not in $o(n^3)$ include: $3n^3$, n^3, $n^3/1000$.

Example: Show that $n^3/1000 \neq o(n^3)$

Solution By definition we have

$o \le h(n) \le cg(n)$, for any constant $c>0$

$o \le n^3/1000 \le cn^3$

This is contradiction with selecting any $c<1/1000$.

(2) Little Omega Notation(ω)

This notation provides a non-symptotically tight lower bound for f(n). It can be simply written as $f(n) \in \dot{\omega}(g(n))$, where

$\dot{\omega}(g(n))=\{h(n)$: positive constants c, n_0 such that for any $c>0, n_0>0$ and $0 \le cg(n) < h(n)$, $n \ge n_0\}$

This is unlike the Ω notation where we say for some c>0 (not any). For example $5n^3 = \Omega(n^3)$

Is asymptotically tight upper bound but $5n^2 = \dot{\omega}(n^3)$ is non-asymptoticallytight bound for f(n).

■ ■ ■

Questions

1. Define data structure. Give some example.
2. In how many ways can you categorize data structures? Explain each of them.
3. Write a short note on abstract data structure.
4. Write a short note on different operation that can be performed on data structure.
5. Write short note on
   (a) Data type (b) File
   (b) Record (d) Field
6. What is an algorithm?
7. Explain the features of good algorithm.
8. State down the algorithm efficiency.
9. What do you understand by Complexity?
10. Write short note on
    (a) Time complexity (b) Space complexity
11. Discus the best case, worst case, average case and amortized of an algorithm.
12. What do you understand by time-space trade-off?
13. Explain Big O notation.
14. Discuss the significance and limitations of the big O notation.
15. Explain Ω notation.
16. Explain Θ notation.
17. Differentiate between Big O notation and little o notation.

Exercise

1. Show that $n^2+50n=O(n2)$
2. Show that $n^2+ n^2 +n^2= 3n^2=O(n^2)$
3. Show that $n= O(n \log n)$
4. Which notation provides a strict upper bound.
5. Prove that running time $T(n) = n^3 + 20n + 1$ is not $O(n^2)$
6. Prove that running time $T(n) = n^3 + 20n$ is $\Omega(n^2)$
7. Prove that running time $T(n) = n^3 + 20n + 1$ is $O(n^3)$

Chapter ▸ 2

Linear Structure

2.1 Introduction

An array is an aggregate data structure that is designed to store a group of objects of same type or different types. The way to storing the objects should be linear that's why comes in category of linear data structure. Linear data structure means, have a linear relationship between the elements represented by means of sequential memory locations. Besides Array linked list also come in category of linear data structure. Here we have a linear relationship between the elements represented by means of pointers or links.

The operation one normally performs on linear data structure are as follow : -

1. **Traversal:** Processing each element in the array/list.
2. **Searching:** finding the location of the element with a given value or the record with a given key.
3. **Insertion:** Adding a new element to the array/list.
4. **Deletion:** Removing an element form either an array or list.
5. **Sorting:** Arranging the elements in same type of order. (ascending & descending etc.)
6. **Merging:** Combining two lists/array into a single list/array.

2.2 Linear Array

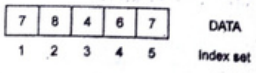
A linear array is a list of finite number of 'n' homogeneous (siniliar type) data element stored in successive memory locations. The element of array are referenced respectively by an index set constiting consecutive numbers.

Size of an array means total number of elements in an array. If not explicity given we assume the index set consists of integers 1, 2, 3 …….. n. length of an array can be calculated by an airthmatic formula.

Length = UB-LB+1

Where UB is the largest index called upper bound and LB is the smallest index called lower bound of the array.

For e.g. We have an array such as



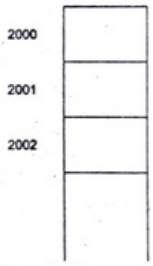
Length = UB-LB +1
= 5-1+1
= 5

This array has five data elements.

Each programming language has its own rules of declaring arrays. Each such declaration must given implicitely or explicitely three items of information (1) name of the array (2) data type of the array and (3) index set of the array.

2.3 Representation of Linear Array in Memory

Let LA be a linear array in the memory of computer. Memory of the computer simply a sequence of addressed locations as pictured in figure.



Computer Memory

Elements of LA stored in successive memory cells. First element address denoted by Base (LA) which we have to remember to calculate the address of any elements of LA.

LOC (LA (k) = Base (LA) + w (k-lower bound)

Where w is the number of words per memory cell for the array k is subscript.

Example: Consider on array (CAR, which records the number of Cars Sold each year from 1981 through 2001 Suppose CAR appears in memory as pictured shown. That is Base (CAR)=300, and w = 4 words per memory cell/or CAR, then-

LOC (CAR [1981]=300
LOC (CAR [1982] = 304
The address of the array
element for the year k=1999
can be obtained by using formula.

LOC [CAR (1999)]= BASE [CAR] + w (1999-lower bound)
= 300 + 4 (1999-1981)
= 300 + 4 (18)
= 372.

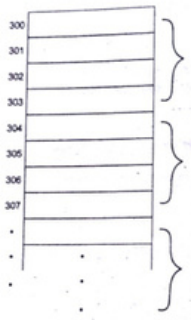


Figure 2.1 Representation of Linear Array in Memory

2.4 Traversing Linear Array

Traversing a linear array means moving through sequentially node by node. Processing the data element of a node may be complex but general pattern is as follows:

- Begin at the first node,
- Repeat until there are no more nodes.
- Process the accessing node.
- Move to the next node.

Algorithm: Traversing a linear Array

(Here LA is a linear array with lower bound LB and upper bound UB. Traverses LA means process to each element of LA).

1. [Initialize counter] Set K : = LB
2. Repeat Step 3 and 4 while K ≤ UB
   [Visit node]. Apply PROCESS to LA [K]
   [Increase counter] Set K=K+1
   [End of Step 2 loop]
3. Exit

2.5 Inserting and Deleting

Let LA be a linear array. Operation of adding one more data element in the LA is termed as ''Insertion'' whereas "Deleting" refers to the operation of removing one of the data element from the LA.

Inserting an data element at the ''end'' of a linear array can be easily done because memory space for array is enough to accommodate the new one. But if we want to add an element in between the two data element or in middle of the array then on the average half of the elements must be moved downward to new locations to accomodate the new element and keep the order of the other elements.

Algorithm: Insertion in a Linear Array.

Here LA is a linear array with N elements and k is positive integer such that K ≤ N. This algorithm inserts an element ITEM into the Kth position in LA.

Insert (LA, N, K, ITEM)

1. Set J=N (Initialise counter)
2. Repeat steps 3 & 4 while J ≥ = K
3. Set LA [J+1] = LA [J] (Move Jth element upward)
4. Set J= J-1 (Decrease counter)
   [End d step 2 loop]
5. Set LA [k] = Item (Insert element)
6. Set N=N +1 (Reset N]
7. Exit

Example:

Insert

A	B	C	D	E	G	H	I			
0	1	2	3	4	5	6	7	8		

Insert f in between E & G

A	B	C	D	E	G	H	I			
0	1	2	3	4	5	6	7			

New LA

A	B	C	D	E	F	G	H	I		
0	1	2	3	4	5	6	7	8	9	

Algorithm: Deletion in A Linear array.

Here LA is a linear array, array with N elements and K is positive integer such that K < N. This algorithm deletes the kth element from the LA.

Delete (LA, N, K, Item)

1. Set ITEM = LA [k]
2. Repeat for J= K to N-1
   Set LA [J] = LA [J+1] [move J+1 element upward)
   [End of step 2 loop]
3. Set N= N+1 [Reset the number N of elements in LA]
4. Exit

Example:

	1	2	3	4	5	6	7	
	11	22	33	44	55	66	77	
Step 1	11	22	33	55	55	66	77	
Step 2	11	22	33	55	66	66	77	
Step 3	11	22	33	55	66	77		

SORTING AND SEARCHING will be discussed in later chapter.

half of the elements must move down word to new locations to accommodate the new element.

Similarly deleting in the middle would require that each subsequent element be mored upward in order to ''fill up'' the array. Deleting an element at the ''end'' of an array has no complexity.

A program to insert a number at a given location in an array

```
# include <stdio.h>
# include <conio.h>
int main()
{   int i,n, num,pos, arr[10];
  clrscr();
  printf("\  n  Enter the number of elements in the array");
  scanf("%d", &arr[10]);
    for(i=0;i<n: i++)
{
   printf(" \n arr[%d]=",i);
   scanf(" %d", &arr[i]);
     }
    printf(" \n enter the number to be inserted : ");
    scanf(" %d", &num);
    printf("\  n enter the position at which the number has to be added : ");
    scanf(" %d", &pos);
for(i=n-1;i>=pos; i--)
  arr[i+1]=arr[i];
arr[pos]=num;
n=n+1;
printf(" \ n the array after insertion of %d is :  ", num);
for(i=0;i<n; i++)
  printf("\n arr[%d] = %d", i,arr[i]);
getch();
return 0;
}
```

Write a program t o delete a number from a given location in an array

```
# include <stdio.h>
# include <conio.h>
int main()
{
int i,n, num,pos,arr[10];
 clrscr();
 printf(" \ n  Enter the number of elements in the array");
 scanf("%d", &arr[10]);
    for(i=0;i<n: i++)
{
  printf("\ n arr[%d]=",i);
  scanf(" %d", &arr[i]);
   }
```



```
       printf(" \n enter the position from which the number has to be deleted
: ");
       scanf(" %d", &pos);
  for(i=pos;i<n-1; i++)
    arr[i]=arr[i+1];
  n--;
  printf(" \ n the array after deletion of %d is :  ");
  for(i=0;i<n; i++)
    printf("\n arr[%d] = %d", i,arr[i]);
  getch();
  return 0;
  }
```

2.6 Multidimensional Array

The linear array discussed so far are also called ''one dimensional array where accessing its elements involves a single subscript which can be either represent a row or column index.

1	2	3	4	5	6
11	22	33	44	55	

Arrays can have more than one dimension such arrays are called multidimensional arrays. They are very similar to standard array with exception that they have multiple subscript (set of square brackets after the array identifier).

Two Dimensional Arrays

A two-dimensional array A is a collection of **m** rows and **n** columns containing data elements such that each element is specified by a pair of integers (such as J, K) called subscript.

$1 \le J \le m$ and $1 \le K \le n$.

A Common declaration of a two dimensional array as

A [J, K]

How to compute the data element of given position in multidimensional array?

Let 'two-dimensional array mx n. The computer keeps. track of Base (A)- The address of the first element A [1,1] of A and computes the address LOC (A [J,K]) of A (J,K) using the formula.

(Column-major order) **LOC (A [J, K] = Base (A) + w (M (K-1) + (J-1)]**

or the formula

(Row-major order) **LOC (A [J,K]= Base (A) + w [N (J-1) + (K-1)]**

Again w denotes the numbers of words per memory location for the array A. Note that formula are linear in J and K

Example: Consider the 25x4 matrix array SCORE. Suppose Base (SCORE)= 200 and there are w= 4 words per memory cell. Furthermore suppose the programming language stores two- dimensional arrays using row-major order. Then the address of SCORE [12,3] the third test of the twelveth student follows:

LOC(SCORE [12,3])= 200+4 [4(12- 1) + (3- 1)]
= 200+4 [46] = 384

2.7 Record

Record is a collection of pieces of information pertaining to a single object. The object may be physical or conceptual. For example a student record may comprise items of information such as student's name, roll number, registration number, age, grade in semester 1, grade in semester 2, etc. A bank account record (a bank account is a conceptual object) may comprise account holder's name, present balance, overdraft limit, etc. The individual items of information in a record are often called fields of the records. This notion of a record also provides for the existance of multiple instances of a type of record, i.e., there may be several student records (each about a distinct student object), or several bank account records (each about a distinct account). The sizes of individual instances of records of the same type may be same or allowed to vary.

2.7.1 Representing a record in computer

Each field of a record may be represented (i.e. held in memory and processed) in a computer system by the data abstraction mechanisms provided by the programming languages. Data abstraction mechanisms mean built-in data types in a language and user defined data types. Further most programming languages also allow user to define composite data types (viz., struct and union in C, class in C++, record in Pascal). In these mechanisms when a record is identified its constituent fields can be identified (accessed). So records can be represented in a computer using such mechanisms.

2.8 Stack/Queue

Besides Array and linked lists, stack and Queue are also linear data structure. Stack and queen also used to store data elements in sequential order in the memory space of computer.

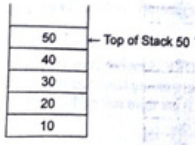
A stack is a linear data structure in which addition or deletion of data element only at one and that's why considered as restricted typed data structure (addition of data element to end and removing of data element to other end) Stack are also called Last-in-first-Out (LIFO) list means last item to be added is the first item to be removed. For example, the tennis balls in their container. Ball remove which was last insert and no two balls on be remove at same time.

A queue is a linear list in which addition or deletion of data element only at one end. As we wait bus stand is an example of queue. First person in the line is the first person to leave thus queue are also called as first-in-first out lists.

2.8.1 Stack is Abstract Data Type and Linear Data Structure

In stack addition of new element or deletion of existing element always takes place at a same end. This end is known as the top of the stack. That means that is possible to remove elements from a stack in reverse order from the insertion of elements into the stack. Also called as LIFO (last in first out). One other way of describing the stack is as last in, first out (LIFO) abstract data type and linear data structure-

Example: 10, 20, 30, 40, 50



Structure of Stack

10 is the first data element which first enter in stack, then 20, 30, ... so on 50 is on the top of the stack.. 50 be the first which removed from the stack, which enter last in the stack.

2.8.2 Operations on Stack

The stack is basically performed two operations PUSH and POP.

Push and Pop are the operations that are provided for insertion of an element into the stack and the removal of an element from the stack respectively.

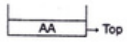
PUSH: PUSH operation performed for the adding item to the stack.
POP: Pop operation performed for removing item from a stack.

PUSH operation:

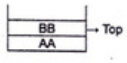
Example: Suppose the following 6 elements are pushed, in order, into an empty stack.

AA, BB, CC, DD, EE, FF

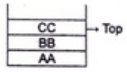
Step 1: Push AA



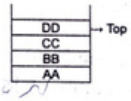
Step 2: Push BB



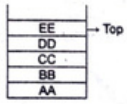
Steps 3: Push CC



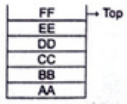
Step 4: Push DD



Steps 5: Push EE



Step 6: Push FF



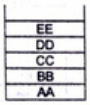
Right most element in the above array placed at the top of the stack.

Write a program to perform Push,Pop operation

```
# include <stdio.h>
# include <conio.h>
#define MAX 10
int st[MAX], top=-1;
void push(int st[],int val);
int pop(int, st[]);
void display(int st[]);
int main()
{
  int val ,option;
  clrscr();
  do
{
  printf("\ n  *****MAIN MENU*****");
  printf("\n 1. PUSH ");
  printf(" \n 2. POP");
printf("\ n 3 DISPLAY");
printf(" \n 4 EXIT");
printf(" n\ Enter your option: ");
scanf("%d ", &option);
switch(option)
{
  Case 1:
    printf("\ n Enter the number to be pushed on stack :");
    scanf(" %d", &val);
    push(st, val);
    break;
  Case 2:
    val=pop(st);
    if(val!=-1)
    printf("\n The value deleted from the stack is :%d", val);
    break;
  Case 3:
    display(st);
    break;
}while(option!=4);
getch();
return 0;

}
void push (int st[], int val)
{
  if(top = =MAX-1)
  {
    printf(" \n STACK OVERFLOW");
}
else
{
  top++;
  st[top]=val;

}
}
int pop(int st[])
{
int  val;
  if (top = = -1)
    {
    printf(" \n STACK UNDRFLOW");
    return -1;
    }
    else
    {      val = st[top];
    top--;
    return;
}

}
void display (int st[])
{
  int i;
  if (top = = -1)
    printf(" \n STACK IS EMPTY");
  else
{
  for(i=top;i>=0;i--)
    printf(" n%d ", st[i]);
}
}
}
```

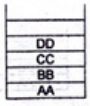

POP operation

Example: Suppose have a stack of 5 elements, pop the DD element

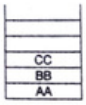


for pop out DD, use have to pop first EE from the stack because deletion always done at top of stack

step 1



step 2

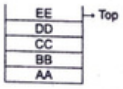


Example: Operations to Perform

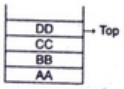
1. Delete EE
2. Delete CC
3. Add GG

1. FF is on the top of stack, This mean EE connot be deleted before FF is deleted CC cannot deleted before DD, EE, FF are deleted. The elements may be popped from the stack only in the reverse order of that in which they were pushed onto the stack.

POP FF

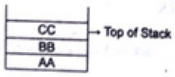


POP EE

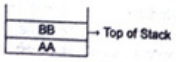


2. Now DD is on top of stack. Our aim is to delete CC.

POP DD

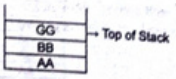


POP CC



3. GG is to insert in the stack.

Push GG



2.8.3 Over Flow Stack

Each stack has deserved amount of memory space. The condition, when stack is fully occupied, no more memory space to hold the new one element, such stack is called overflow stack.

Example: Suppose a stack of 6 elements overflow will occur when stack contains more than 6 elements.

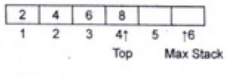
2.8.4 Underflow Stack

If POP operation apply on an empty stack, underflow condition met. Empty stack has no element to pop out. Such stack is called underflow stack.

2.8.5 Array Representation of Stack

Stacks may be represented in the computer in various ways, linear array and linked list. Here we discuss array representation of stack. Let us assume that the name of the linear array be array be arr and this array can hold maximum 6 integer numbers.

There are two representation ways of stack.



(i) Horizontal Representation

(ii) Vertical Representation

Top variable, whichs contains the location of the top element of the stack; and a variable MAXSTK, which gives the maximum number of elements that can be hold by the stack. The Condition TOP = 0 or TOP = NULL will indicate that the stack is empty. In the above array TOP=4, the stack has four element 2, 4, 6, 8 and since MAXSTK=6, there is room for 2 more data elements in the stack.

2.8.6 Application of stack

In this section we discussed about the problems where stacks can be easily applied for a simple and efficient solution.

- Reversing a List
- Paranthesis checker
- Evaluation of a Airthmatic Expression
- Conversion of an infix expression into a postfix expression
- Conversion of an infix expression into a prefix expression
- Recursion

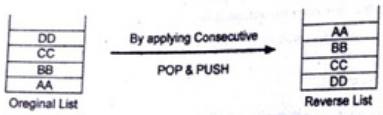
(a) Reversing a List

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

Reversing a list

Example: We have a stack of 4 element having order DD, CC, BB, AA. We reverse the order.

For this purpose, we use two stacks.



Algorithm: PUSH (STACK, TOP, MAXSTK, ITEM)

1. [Stack already filled?]
   If Top = MAXSTK, then : Print : Overflow, and Return
2. Set Top = Top +1 [Increase Top by 1]
3. Set Stack [Top] = Item [Insert Item in new Top position]
4. Return

Algorithm: POP (Stack, TOP, ITEM)

This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?]
2. IF Top=0, then : Print : Underflow, and Return
3. Set Top = Top-1 [Decrease Top by 1]
4. Return

(b) Paranthesis Checker

In algebraic expression , for every open bracket there is a corresponding closing bracket. For example (A+B) is invalid but an expression {A+(B-C)} is valid.

Write a program to check nesting of parantheses using a stack.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 10
int top=-1;
int stk[MAX];
void push(char);
char pop;
void main();
{
 char exp[MAX],temp;
 int i, flag=-1;
 clrscr();
 printf(" Enter an expression :");
 gets(exp);
 for(i=0; i<strlen(exp); i++)
    {
 if (exp[i]=='('|| exp[i]=='{'|| exp[i]=='[')
```



```
    Push(exp[i]);
  if (exp[i]==')'|| exp[i]=='}'||exp[i]==']')
    If(top==-1)
      flag=0;
    else
    {
      temp=pop();
      if(exp[i]==')'&&(temp=='{'||temp=='['))
        flag=0;
      if(exp[i]=='}'&&(temp=='('||temp=='['))
        flag=0;
      if(exp[i]==']'&&(temp=='{'||temp=='('))
        }
      }
    if(top>=0)
    flag=0;
    if(flag==1)
    printf("\ n Valid expression");
    else
    printf(" \n Invalid expression");
    }
void push(char c)
{
 if(top==(MAX-1))
   printf(" stack is Overflow");
 else
 {
   top=top+1;
   stk[top]=c;
}
}
char pop()
{
 if (top==-1)
   printf(" stack overflow");
 else
   return (stk[top--1]);

   }
```

(c) Evaluation of Airthmatic Expression

- **Polish Notation**

Infix,Postfix and Prefix notations are three different type of notations to write an algebraic expression.

- **Infix Notation**

Operators are written between operands . This is the usual way in which we write.An expression such as A*(B+C).is usually taken to mean something like " First add B and C together, then multiply result to A. Infix notation needs extra information to make the order of evaluation of the operators clear: rules

built into the language about operator precedence and associativity and brackets ( ) to allow users to override these rules. For example, the usual rules for associativity say that we perform operations from left to right so the multiplication by A is assumed to come before the division by D. Similarly, the usual rules for precedence say that we perform multiplication and division before we perform addition and subtraction.

- **Postfix Notation** (also known as Reverse Polish Notation): Operators are written after their operands. The infix expression given above is equivalentto `A B C + * D /` The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order. Because the "+" is to the left of the "*" in the example above, the addition must be performed beforethemultiplication. Operators act on values immediately to the left of them. For example the "+" above uses the "B" and "C". We can add (totally unnecessary) brackets to make this explicit: `((A(BC+)*)D/)` Thus, the "*" uses the two values immediately preceding: "A", and the result of the addition. Similarly, the "/" uses the result of the multiplication and the "D".

- **Prefix Notation(also known as "Polish Notation")**

Operators are written before their operands. The expressions given above are equivalent to `/ * A + B C D` As for Postfix, operators are evaluated left-to-right and brackets are superfluous. Operators act on the two nearest values on the right. I have again added (totally unnecessary) brackets to make this clear:

`(/(*A(+BC))D)`

Although Prefix "operators are evaluated left-to-right", they use values to their right and if these values themselves involve computations then this changes the order that the operators have to be evaluated in. In the example above although the division is the first operator on the left it acts on the result of the multiplication and so the multiplication has to happen before the division (and similarly the addition has to happen before the multiplication).

(d) Conversion of an Infix Expression into a Postfix Expression

We have an algebraic expression written in infix notation may conatain parentheses, operands, and operators. For simplicity we use +,-,*,/,% operators.The precedence of such operators are as follow:

High Priority *,/,%
Low Priority +,-

The precedence works, we have an expression like A+B*C , then first B*C is calculated then A is added into the result. But if same expression written as (A+B)*C , will evaluate A+B first and then the result will be multiplied with C.

One more thing is important that if same operators are appear then precedence are performed from left-to-right.

Stack is temporarily used to hold operators. The postfix expression is obtained from left-to-right using operands from the infix expression and the operators which are removed from the stack.

Example: Convert the following infix expression into postfix notation using stack

(a) A—(B/C+(D%E*F)/G)*H

Solution:

Conversion of an infix expression into a post fix expression.

Infix character scanned	Stack	Post fix Expression
	(	
A	(	A
-	(-	A
C	(-(	A
B	(-(	AB
/	(-(/	AB
C	(-(/	ABC
+	(-(+	ABC/
C	(-(+(	ABC/
D	(-(+(	ABC/D
%	(-(+(%	ABC/D
E	(-(+(%	ABC/DE
*	(-(+(%*	ABC/DE
F	(-(+(%*	ABC/DEF
'	(-(+	ABC/DEF*%
/	(-(+/	ABC/DEF*%
G	(-(+/	ABC/DEF*%G
'	(-	ABC/DEF*%G/+
*	(-*	ABC/DEF*%G/+
H	(-*	ABC/DEF*%G/+H
'		ABC/DEF*%G/+H

(e) Conversion of Infix Expression into a Prefix Expression

Work same as above method done.

Write a program to convert an infix expression into its equivalent postfix notation.

```
# include <stdio.h>
# include <conio.h>
# include <ctype.h>
# include <string.h>
# define MAX 10
char st[MAX];
int top=-1;
void push(char st[],char);
char pop(char st[]);
void IfixtoPostfix(char source[],char target[]);
int getpriority(char);
```

```
int main()
{
 char infix[100], postfix[100];
 clrscr();
 printf(" \n Entry any Infix Expression :");
 gets(Infix);
 strcpy(postfix, " ");
 InfixtoPostfix(infix, postfix);
 printf("\n The corresponding postfix expression is :" );
 puts(postfix);
getch();
 return 0;
}
Void InfixtoPostfix( char source[], char target[])
{
 int i=0,j=0;
 char  temp;
 strcpy(target , " ");
 while(source[i]!='\0')
 {
  if(source[i]==')')
  {
   push(st, source[i])
   i++;
  }
  else if(source[i]= =')')
  {
   while ((top!=1) && (st[top]!='('))
  {
   target[j]=pop(st);
   j++;
  }
  if(top= = -1)
  {
   printf(" \n INCORRECT EXPRESSION");
   exit(1);
  }
  temp=pop(st); // remove left parathensis
  i++;
 }
 else if(isdigit (source[i]) || isalpha(source[i])
 {
  target[j]= source[i];
  j++;
  i++;
 }
 else if (source[i]= = '+' || source[i]= = '*' || source[i] = = '/' ||
souce[i] = = '%')
 {
while  (  (top!=  -1)  &&  (st[top]!=  '(')  &&  (getpriority(st[top])>
```



```
getpriority(source[i])))
        {
         target[j] = pop(st);
         j++;
        }
        push(st, source[i]);
        i++;
      }
      else
      {
       printf(" \n INCORRECT ELEMENT IN EXPRESSION");
       exit();
      }
    }
  while ((top!= -1) && (st[top]!='('))
  {
   target[j]= pop(st);
   j++;
  }
  target[j]='\0';
  }
  Int getPriority(char op)
  {
   if(op= = '/' || op = ='*' || op= = '%')
    return 1;
   else if (op = ='+' || op = ='-')
    return 0;
  }
  Void push(char st[], char val)
  {
   if ( top= = MAX-1)
    printf("\n STACK IS OVERFLOW");
   else
   {
    top++;
    st[top]=val;
   }
  }
  char pop(char st[])
  {
   char val =' ';
   if (top = = -1)
    printf("\n STACK UNDERFLOW");
   else
   {
    val=st[top];
    top--;
   }
   return val;
  }
```

(f) 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 doesnot require a call to itself. Recursive function make a use of Stack to temporarily store the return address and local variables of the calling function.

To understand recursive functions, let us take an example of 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 6!
6!=6 ×5 ×4 ×3× 2 ×1
= 720

This can be written as 6!=6 ×5! Or 6!= 6×5×4!
Every recursive solution has two major cases:
Base case: in which the problem is simple enough to be solved directly without making any further calls to the same function.
Base case is when n=1,because if n=1, the result will be 1 as 1!=1

Recursive case

In which first problem is divide into simpler sub-parts . Second the function calls itself but sub-parts the problem obtained in the first step. For eg. 6!=6×5!

2.9 Queue

Queue is also an abstract data type or a linear data structure in which the first element is inserted from one end called REAR (also called tail) and the deletion of existing element takes place from the other and called as FRONT (also called head). This makes queue as FIFO data structure which means that element inserted first will also be removal first.

Queue abound in everyday life. The automobiles waiting to pass through an intersection form a queue, in which the first can in line is the first car through; As important example of time of a queue in computer science occurs in a time sharing system in which propgrams with the same priority form a queue while waiting to be executed.

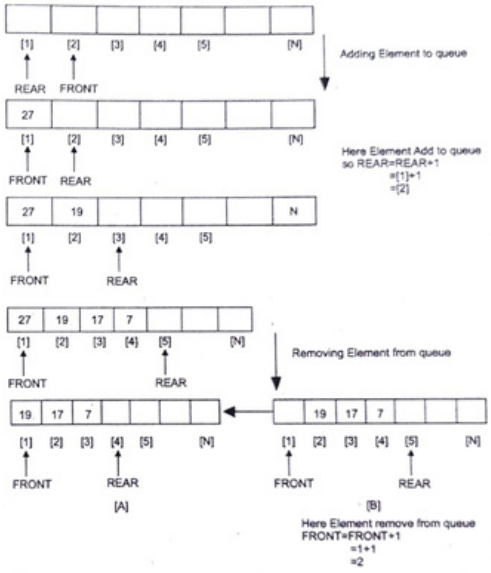
Representation of Queues: Queue may be represented by linear array one-way tests unless otherwise stated or implied, each of our queue will be maintained by a linear array dueue and two paints the location of the rear element of the queue; and the REAR, containing is empty.

Let take an array queue with N elements.The figure on next page also indicate the way elements will be deleted from the queue and the way new element will be added to the queue. Whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment.

FRONT = FRONT + 1

Similarly, whenever an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment.

REAR = REAR +1



Algorithm: Q Insert [Queue, N, FRONT, REAR, ITEM]

This procedure inserts an element ITEM into a queue

1. [Queue already filled?]
   If Front = I and REAR=N, or if Front = Rear+1, then : write, overflow and Return.
2. [Find new value of REAR]
   If Front = Null, then : [Queue initially empty]
   Set Front=1 and REAR=1
   else if REAR= N, then
   Set REAR=1
   Else
   Set REAR = REAR +1
   [End of if structure]
3. Set queue [REAR] = Item [this insert & new element]
4. Return

Algorithm: Q DELETE [QUEUE, N, FRONT, REAR, ITEM]

This procedure deletes an element from a queue and assings it to the variable ITEM.

1. [queue already empty ?]
   IF FRONTC= NULL, then : write : Underflow and Return
2. Set ITEM = QUEUE [FRONT]
3. [Find new value of FRONT]
   IF FRONT = REAR then : [Queue has only one element to start]
   Set FRONT = NULL and REAR= NULL
   Else if FRONT = N then :
   Set FRONT=1
   else
   Set FRONT = FRONT +1
   [End of if structure]
4. Return.

When we remove element from queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram. In [A] approach we remove the element at FRONT position and then one by one move all the other elements on position forward. In [B] approach we remove the element from FRONT position and then move FRONT to the next position.

In approach [A] there is overhead of shifting the elements one position forward every time we remove the first element. In approach [B] there is no such overhead, but whenever we move one position ahead, after removal of first element, the size of queue is reduced by one space each time.

Write a program to implement a linear Queue.

```
# include <stdio.h>
# include <conio.h>
# include <string.h>
# define MAX 10
int queue[MAX]
int front=-1,rear=-1;
void insert (void);
int delete_element(void);
int main()
{
  int option, val;
  clrscr();
  do
  {
    printf(" \n\n *****MAIN MENU*****");
    printf("\n 1. Insert an element");
    printf("\n 2. Delete an element");
printf("\n 3. Peek");
    printf("\n 4. Display the queue");
     printf("\n 5. EXIT");
    printf("\n 6. Enter your option :");
    scanf(" %d", &option);
    switch(option)
    {
     case 1:
```



```
        insert();
        break;
      case 2:
        val=delete_element();
        if (val!=-1)
          printf(" \n The first value is queue is : %d", val);
        break;
      case 3:
        val=peek();
        if(val!= -1)
          printf("\nThe first value in queue is : %d", val);
        break;
      case 4:
        display();
        break;
}
}while (option!=4)
getch();
return 0;
}
void insert()
{
  int num;
  printf("\n Enter the number to be inserted in the queue :");
  scanf(" %d", &num)
if(rear==MAX-1)
  printf("\n OVERFLOW");
else if (front==-1 && rear==-1)
    front=rear=0;
else
  rear++;
queue[rear]=num;

}
int delete_element()
{
  int val;
  if (front== -1 ||  front>rear)
  {
    printf("\n UNDERFLOW");
    return -1;
}
val=queue[front];
front++;
if(front>rear)
  front = rear=-1
{
  if( front=rear= -1)
  return 0;
}
```

```
}
int peek()
{
  if (front== -1 ||  front>rear)
  {
    printf("\n Queue Is Empty ");
return -1;
}
else
  {
    return queue[front];
}
}
void display()
{
 int i;
 printf("\n ");
 if (front== -1||  front>rear)
   printf(" \n Queue Is Empty");
 else
 {
   for(i=front;i<=rear;i++)
   {
    printf("\t %d", queue[i]);
   }

}
```

2.10 Types of Queue

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

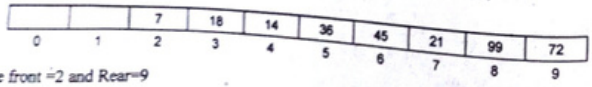
1. Circular Queue
2. Deque
3. Priority Queue
4. Multiple Queue

All these types are discussed in detail in the following section.

Circular Queue

In linear queue, insertion of new element can be done only at one end called as Rear and deletion of element can be done from other end called the Front. Now if you want to insert new element in full occupied queue will not be possible.

Consider a linear queue as shown below



Here front =2 and Rear=9

Suppose we want to insert a new element in the queue. Even though there is a space available , but still there is condition of Overflow . To solve this problem we have two solution .First shift the element to left

side so that vacant space is efficiently utilized. But this method is time-consuming,especially when queue is quite large.

The second option is to join the Rear to Front ends of queue to make queue as circular queue.

Circular queue is a linear data structure . It follows FIFO principle.

Figure 2.2 Circular Queue

1. In circular queue the last node is connected back to first node to make it circular
2. Circular queue follows First In First Out principle.
3. elements are added at rear end and the elements are deleted at front end.
4. Bothe Rear and Front pointers points to the beginning of circular queue points.
5. It is also called as "Ring Buffer".
6. Circular queue is full only when Front=0 and Rear=Max-1

Write a program to implement a circular queue

```
# include <stdio.h>
# include <conio.h>
# define MAX 10
int queue[MAx]
int front=-1,rear=-1;
void insert (void)
int delete_element(void);
int peek(void);
void display(void);
int main()
{
  int option,val;
clrscr();
do
{
  printf("\n ***** MAIN MENU ***** ");
  printf(" \n 1. Insert an element ");
  printf(" \n 2. Delete an element ");
  printf(" \n 3. Peek ");
  printf(" \n 4. Display the Queue ");
printf(" \n 5. EXIT ");
printf(" \n Enter your option :  ");
scanf("%d", &option);
```

```
    switch(option)
    {
     case 1:
      insert();
      break;
     case 2:
      val=delete_element();
      if(val!=1)
      printf(" \n The number deleted is : %d", val);
      break;
     case 3:
      Val=peek();
      if (val!=-1)
      printf("\n The first value in Queue is : %d", val);
      break;
     case 4:
      display();
      break;
    }
   }while(option!=5)
   getch();
   return 0;
   }
   void insert()
   {
    int num;
    printf("\n Enter the number to be inserted in the queue :");
    scanf(" %d", &num)
  if(front==0 && rear==MAX-1)
   printf("\n OVERFLOW");
  else if (front = = -1 && rear = = -1)
  {
   front=rear=0;
   queue[rear]=num;
  }
  else if (rear=MAX-1 && front!=0)
  {
   rear=0;
   queue[rear]=num;

 }
 else
 {
  rear++;
  queue[rear]=num;

}

}
int delete_element()
{
```



```
  int val;
  if (front = = -1 && rear = =  -1)
  {
   printf("\n UNDERFLOW");
   return -1;
  }
 val=queue[front];
 if(front==rear)
  front=rear= -1;
 else
 {
  if front=rear= -1
   front=0;
  else
   front+;
 }
 return val;
 }
 int peek()
 {
  if (front== -1 && rear== -1)
  {
   printf("\n Queue Is Empty ");
 return -1;
 }
 else
  {
   return queue[front];
 }
 }
 void display()
 {
  int i;
  printf("\n ");
  if (front = = -1 && rear = = -1)
   printf(" \n Queue Is Empty");
  else
  {
   if(front<rear)
   {
    for(i=front;i<=rear;i++)
     printf("\t %d", queue[i]);
   }
   else
   {
    for(i=front;i<MAX;i++)
     printf("\t %d", queue[i]);
    for(i=0;i<=rear;i++)
     printf("\t %d", queue[i]);
 }
```

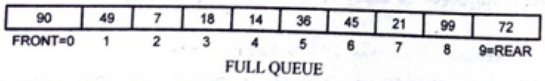
```
}
}
```

Algorithm to insert an element in Circular Queue

```
Step1: IF FRONT=0 and REAR=MAX-1
          WRITE "OVERFLOW"
          GOTO STEP4
Step2: 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]
Step3: SET QUEUE [REAR]=VAL
Step4: EXIT
   Algorithm to delete an element from a Circular Queue
Step1: IF FRONT=-1
        WRITE "UNDERFLOW"
        GOTO Step4
       [END OF IF]
Step2: SET VAL=QUEUE[FRONT]
Step3: IF FRONT=REAR
       SET FRONT=REAR=-1
     ELSE
       IF FRONT=MAX=-1
      ELSE
        SET FRONT=FRONT+1
      [END OF IF]
    [END OF IF]
Step4: EXIT
```

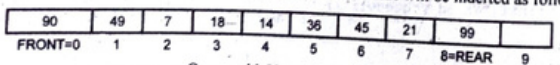
For insertion, three conditions have to check:

1. If Front=0 and rear= Max-1 , then the cicular queue is full



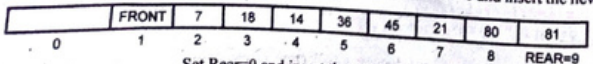
FULL QUEUE

2. If Rear != Max-1 then Rear will be incremented and the calue will be inderted as follow



Queue with Vaccant location

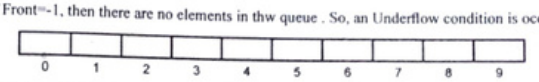
3. If Front !=0 and Rear= Max-1, then the queue is not full. So , set Rear=0 and insert the new element.



Set Rear=0 and insert the value here

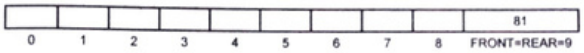
To delete an element, again we have to check three conditions:

1. If Front=-1, then there are no elements in thw queue . So, an Underflow condition is occur.



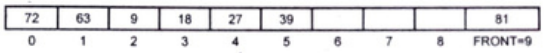
FRONT=REAR=-1 ↳ Empty Queue

2. If the queue is not empty and Front=rear, then after deleting the element at the Front of the queue becomes empty and so Front and Rear are set to -1.As shown in figure.



Delete this element and3 set Rear=Front=-1 ↳ Queue with single element

3. If the queue is not empty and Front= Max-1, then after deleting the element at the front , Front is set to 0. As shown in given figure

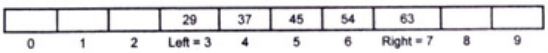


rear ↳ Delete this element and set FRONT=0

Algorithm

Dequeue

A dequeue is double –ended-queue where element can be inserted or deleted at either end. It is also known as head-tail linked list because elements can be added or removed from either the front (head) or the back (tail) end.



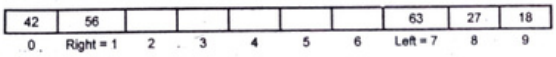


Figure 2.3 Double-Ended Queue

No element can be inserted or deleted from the middle . In the computer's memory 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. This type of data structure have a sub-type as follow:-

1. Input restricted deque: In this deque insertions can be done only at one ends, while deletions can be done from both ends.
2. Output restricted deque: In this deque, deletions can be done only at one of the ends while insertions can be done on both ends.

Priority Queue

The priority queue is a data structure having a collection of elements which are associated with specific ordering. There are two types of priority queue:

1. Ascending
2. Descending

Application of Priority queue

1. The typical example of priority queue is in scheduling the jobs in operating system. Typically operating system allocates priority to jobs. The jobs are placed in the queue and the position of the the jobs in the priority queue determines theior priority. In operating system there are three kinds of jobs. There are real time jobs , foreground job and background jobs. The operating system always schedules the the real time jobs first. If there is no real time job pending then it schedule foreground jobs. Lastly if no real time or foreground jobs are pending then operating system schedules the background jobs.
2. In network communication to manage limited bandwidth for transmission the priority queue is used.
3. In simulation modeling to manage the discrete events the priority queue is used.

Multiple Queue

When we implement a queue using an array , the size of the array must be known in advance .If the queue is allocated less space, then frequent overflow condition will be encountered . To deal with this problem, the code will have to be modified to reallocate more space for the array.

If allocate more space for the queue then it will result in wastage of the memory .Thus, there lies a tradeoff between the frequency of overflow and the space allocated.

So a better solution is have a multiple queue or to have more than one queue in the same array of sufficient size.

Application of Queue

1. Queue are widely uswd as waiting lists for a single shared resource like printer,disk CPU.
2. Used to transfer data asynchronously(data not necessarily received at same rate are sent) between two processes (I/O buffers).
3. Used as buffers on MP3 players and portable CD players, iPod playlist.
4. 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.

■ ■ ■

Very Short Questions

1. Define Array.
2. What is linear array?
3. State the list the operartions performed over array.
4. State down the representing way of array in memory.
5. Write down the algorithm of insertion and deletion of data element in array.
6. What do you understand by two-dimensional array?
7. An arrary int marks[ ]={99,67,78,56,88,90,34,85}, calculate the address of marks[4] if the base address=1000.
8. Consider a 20×5 two dimensional array mark which has its base address =1000 and the size of an element =2. Now compute the address of the element , marks[18][4] assuming that the elements are stored in row major order.
9. Let Age[5] be an array of integers such that
   Age[0]=2, Age[1]=5,Age[2]=3,Age[3]=1,Age[4]=7
   Show the memory representation of the array and calculate its length
10. Consider a two dimensional array arr [10][10] which has its base address =1000 and the size of an element =2. Now compute the address of the element , marks[8][5] assuming that the elements are stored in columa major order.