Data Structure Using C

This download is Sponsored by:

Contributed By:

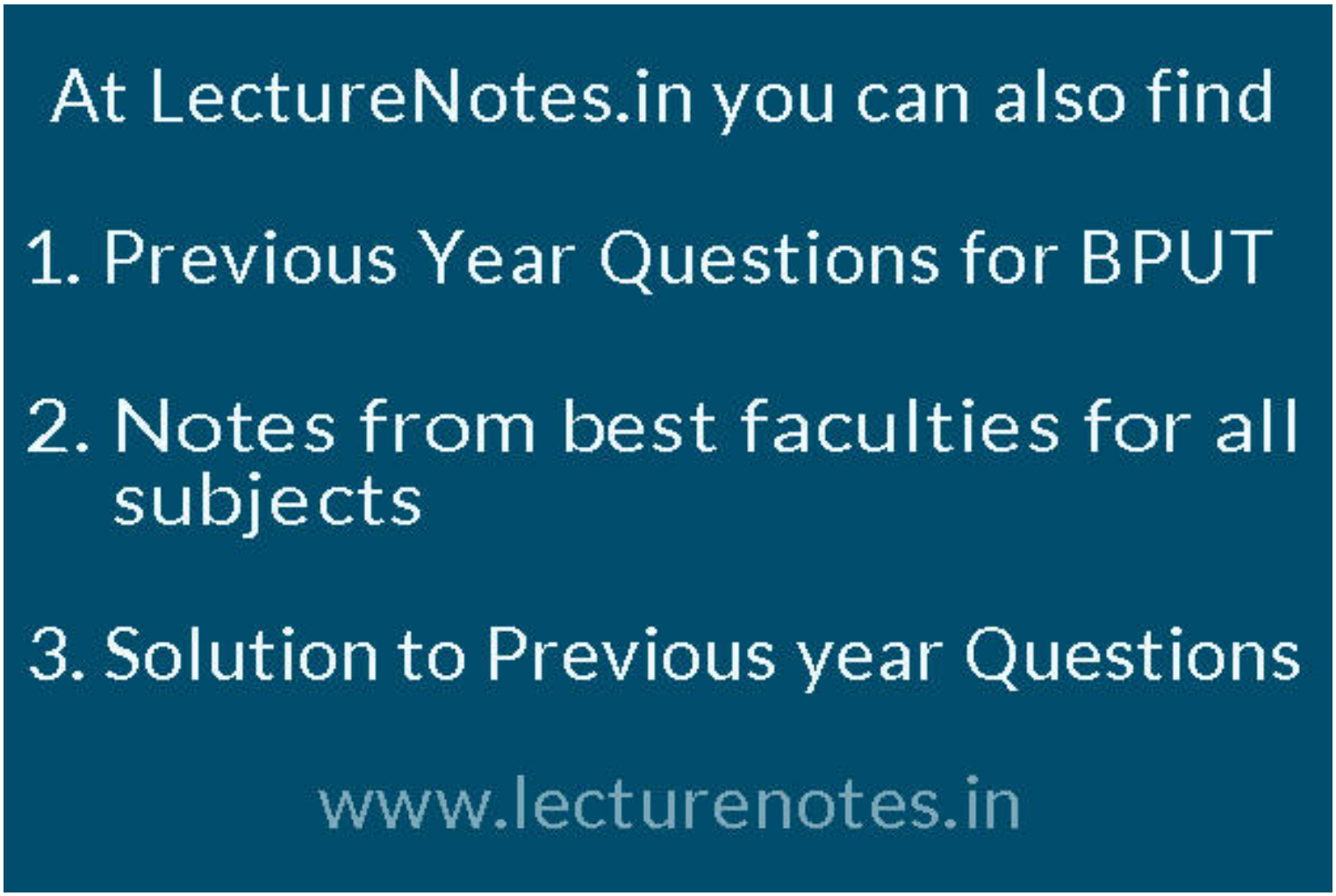
Nihar Ranjan Rout

Gandhi Institute For Technology, GIFT

Disclaimer

This document may not contain any originality and should not be used as a substitute for prescribed textbook. The information present here is uploaded by contributors to help other users. Various sources may have been used/referred while preparing this material. Further, this document is not intended to be used for commercial purpose and neither the contributor nor LectureNotes.in is accountable for any issues, legal or otherwise, arising out of use of this document. The contributor makes no representation or warranties with respect to the accuracy or completeness of the contents of this document and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. By proceeding further, you agree to LectureNotes.in Terms of Use.

This document was downloaded by: amreen bano on 17th Sep, 2017.



Data Structure Using C

Topic:

Introduction To Data Structure

Contributed By:

Nihar Ranjan Rout

Gandhi Institute For Technology, GIFT

INTRODUCTION TO DATA STRUCTURE:

Computer Science is the study of data, its representation and transformation by computer. For every data object, we consider the class of operations to be forced and then the way to represent the object so that these operations may be efficiently carried out. The two techniques required for this is :-

i) Devise alternative forms of data representation.

ii) Analyze the algorithm which operates on the structure.

These includes data structures, data types and data representations.

Data :

→ Data are simply values or set of values. A data item refers to a single unit of values. By data we mean known facts that can be recorded and that help implicit meaning.

Datatype and its necessity :

→ A datatype is a term which refers to the kind of data that a variable may hold. Every programming language has some built-in-datatypes. This means that the language allows variables to name data of that type and provides a set of operations which meaningfully manipulates these variables.

→ If an application needs to use a datatype other than the primitive datatypes of the language i.e. a datatype for which values and operations are not defined in the language itself, then it is programmer's responsibility to specify the values and operations for that datatype and implement it.

GIFT CENTRAL LIBRARY BHUBANESWAR

Abstract Data type :

→ Abstract data type is a mathematical model or concept that defines a data type logically. It specifies a set of data and collection of operations that can be performed on that data. This definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

implementing the operations. It is called abstract because it gives an implementation development view. The process of providing only the essentials and hiding the details is known as abstraction.

Data Structures :

1. The logical or mathematical model of a particular organization of data is called a data structure. Data structure is a programming construct used to implement an ADT. It is the physical implementation of ADT.
2. A data structure that is implementing an ADT consists of collection of variables for storing the data specified in the ADT and the algorithms for implementing the operations specified in ADT. Data structure is the actual representation of data and the algorithms to manipulate the data.

TYPES OF DATA STRUCTURES:

1. Data Structures are broadly classified into two categories.
   i) Primitive (using basic datatypes)
   ii) Non Primitive (using derived data type or user defined data type).

i) Primitive Data Structures:

→ The primitive data structures are typically those data structures that directly operated upon the fundamental data types. e.g. int, char etc in case of 'C' language. So, the data structures that use primary data type is of primitive type.

ii) Non Primitive Data Structures:

→ The data structures operated upon derived or user defined datatype is non primitive type. Non-primitive data structures can be represented in two categories.

ie:- a) Linear type (Array, Linked List, Stacks and Queues)

b) Non Linear type (Tree, Graph, Hash Table, Sets).

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

a) Linear Data Structures:

1. A data structure is said to be linear if its elements form a sequence or linear list. or A data structure which allows representing the data in sequential order and allows the operations accordingly, is of linear type. A list which shows the relationship of adjacency between elements is said to be linear data structure.

2. The linear data structures are, array, linked list, stack and Queue. There are two basic ways of representing such linear structure in memory. i.e:- i) array and ii) Linked list.

3. The linear relationship between the elements represented by means of sequential memory location. These linear structures are called array.

4. The linear relationship between the elements represented by means of pointer or links are called linked list.

Eg:- Suppose a file contains customers name and his/her sales person.

	Customer	Sales Person
1.	A	SMITH
2.	B	RAY
3.	C	JONES
4.	D	RAY
5.	E	SMITH
6.	F	JONES
7.	G	RAY

Figure -1.

	Customer	Link
1.	A	3
2.	B	2
3.	C	1
4.	D	2
5.	E	3
6.	F	1
7.	G	2

Sales person
JONE
RAY
SMITH

Figure -2

	Customer	Link
1.	A	5
2.	B	4
3.	C	6
4.	D	7
5.	E	8
6.	F	0
7.	G	0

Sales Person
JONE
RAY
SMITH

Figure -3 - Linked list representation

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

5. Stack is the last-in-first-out (LIFO) linear list in which insertion and delision can takeplace only one end called top of the stack.

stack of dishes

New dishes are inserted only at the top of the stack and dishes can be deleted only from the top of the stack.

6. Queue is a first-in-first-out (FIFO) linear list in which deletion can take place only at one end of the list which is called front and the insertion can takesplace only at other end of the list called rear end.

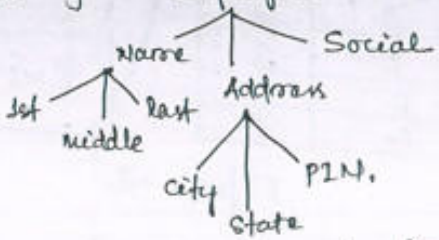
Eg:- The first person in line is the first person to board the bus.

b. Non-Linear Data Structure:

1. The data structure represented in non-sequential order is of non-linear type. The data structure that comes under non-linear type are trees and graphes.

2. In tree, data frequently contains hierarchical relationship between the various elements. It consists of a special element called root and from root many elements are connected like branches, sub-branches and so on, upto leaf.

3. Eg:- Employee



3. Graph:- In Graph, data sometimes containing a relationship between pairs of elements, which is not necessarily hierarchical in nature. It maintains the random relationship between various elements.

Eg:- Suppose an airline files, only between cities connected by lines.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

Algorithm:

1. An algorithm is any well formed defined computational procedure that takes set of values as input and produces set of values as output.
2. A well defined set of instructions to solve a particular problem in finite number of steps is called Algorithm.

Characteristics of Algorithm:

1. Every algorithm should have the following characteristics, i.e.
   - (i) Input: Every algorithm should take zero or more input.
   - (ii) Output: Every algorithm must yeild at least one output.
   - (iii) Definiteness: Every step of an algorithm should be very clear and unambiguous.
   - (iv) Finiteness: Every algorithm must end with a finite numbes of steps. i.e. Every algorithm must have countable numbers of steps.
   - (v) Effectiveness: The steps present in the algorithm should be such that one can implement it by taking the help of pen and paper.

Analyzing an algorithm:

1. Analysing of an algorithm means predicting resources that the algorithm requires. Resources such as:- memory, logic, gets etc. But most often analyzing an algorithm is in computational time that we want to measure. The space needed by each of these algorithm is seen to be the sum of the following components.
   a) A fixed part: It is independent of input and output, which includes the space for code, simple variable etc.

```
Eg:- Algorithm Rsum(a,n)
     {
       if n<=0 then return 0.0;
       else
         return Rsum(a,n-1) + a[n];
     }
```

In fixed part, the space needed to store the program, variable 'n' always take fixed byte and 'a' is the array.

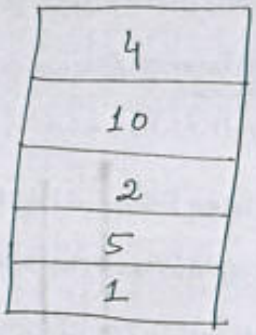
GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

b) A Variable Part: It consist of the space needed by reference variable and recursion stack space.

Eg:- Suppose a[5] is an array, then take a[1] = 4, a[2] = 10, a[3] = 2, a[4] = 5, a[5] = 1.

Rsum(a,5)
↓
Rsum(a,4) +1
↓
Rsum(a,3) +5
↓
Rsum(a,2) +2
↓
Rsum(a,1) +10
↓
Rsum(a,0) + 4



4 +10 = 14 +2 = 16 +5
= 21 +1 = 22

The time T(P) taken by a program 'P' is the sum of the compile time and the recum time. But compile time is fixed for a program and recun time is variant.

2. Once an algorithm is designed, there are two issues which are to be properly dealt with. One issue is validation of the algorithm that means whether the algorithm is valid or not and the other issue is evaluating the complexity of the algorithm.

3. It is necessary to show that designed algorithm yields correct output for all possible combination of input values. This is also called "program proving" or "program verification".

4. One important topic in the study of algorithms is to determine the amount of time and storage it may require for execution. These results are useful for comparing algorithms and predicting the performance of the algorithm in the best, worst and average case.

5. When a program is recun on a computer, two of the most important considerations are :-
   i) Time complexity
   ii) Space complexity.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Criteria for developing an algorithm:

1. It should correctly work under all possible conditions.
2. It should solve the problem according to the given conditions.
3. It should be clearly written by following the top down strategy.
4. It should be efficient to use the time and resources.
5. It should have sufficient documentation so that any one can understand it.
6. It should be easy to modify.
7. It should be independent of running on a particular computer.

Best Case, Worst Case and Average Case:

1. The Best Case time complexity is the minimum amount of time that can required by an algorithm for an input of size 'n'. Thus, it is the function defined by the minimum no. of steps taken on any instant of size 'n'.
2. The Worst case time complexity is the function defined by the maximum amount of the time needed by an algorithm for an input size 'n'.
3. The average case time complexity is the education of an algorithm having typical input data of size 'n', Thus it is the function defined by the minimum no. of steps taken on arrray instance of size 'n',

Eg:-

```
Sum(a,n)
{ s=0;              - 1
  for(i=1 to n)     - n+1
  { s=s+a[i];       - n
  }
  return s;         - 1
}
                    2n + 3
```

CENTRAL LIBRARY BHUBANESWAR

Asymptotic Notation:

1. The Notation, which we use to describe the asymptotic running time of an algorithm are defined in terms of functions, whose domains are the set of natural numbers and real numbers.
   i) The Natural number set is denoted as:-

$$N = \{0, 1, 2 \cdots\}$$

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

iii) The positive integer set is denoted as :-

$$N^+ = \{1, 2, 3 \cdots \infty\}$$

iii) The real number set is denoted as 'R'.

iv) The positive real number set is denoted as 'R^+'.

v) The non-negative real set is denoted as R^*.

2. The different types of notations are :-

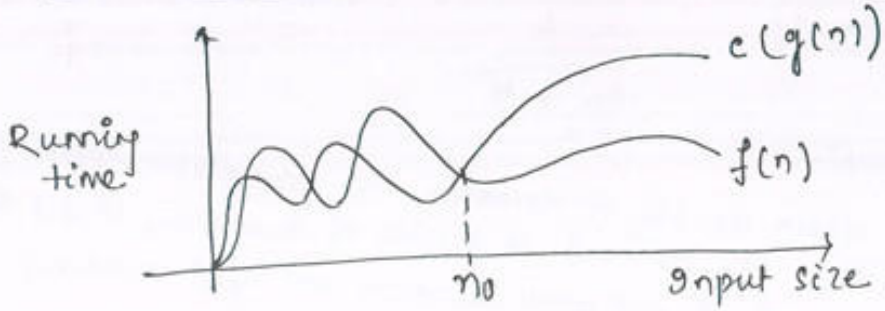
i) Big oh('O') notation
ii) Small oh('o') notation.
iii) Big omega (Ω) notation.
iv) Small omega (ω) notation.
v) Theta(θ) notation.

i) Big Oh('O') Notation :

1. The upper bound for the function is provided by the Big oh(O) notation. We can say, the running time of an algorithm $O(g(n))$, if whenever input size is equal to or exceeds, some threshold n_0', its running time can be bounded by some positive constant 'c' time $g(n)$.

2. Let $f(n)$ and $g(n)$ are two functions from set of natural numbers to set of non-negative real numbers and $f(n)$ is said to be $O(g(n))$. That is, $\boxed{f(n) = O(g(n))}$ iff $\exists$ a natural number n_0' and a positive constants $c > 0$, such that

$$\boxed{f(n) \leq c(g(n)),\ \forall\ n \geq n_0}$$



Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Examples :

1. $f(n) = 2n^2 + 7n - 10$

$\Rightarrow f(n) = O(g(n))$, where $g(n) = n^2$

i) let $n = 5$, $c = 3$

$\Rightarrow f(n) \leq c(g(n)) \Rightarrow 2n^2 + 7n - 10 \leq cn^2$

$\Rightarrow 2 \times 25 + 7 \times 5 - 10 \leq 3 \times 25 \Rightarrow 50 + 35 - 10 \leq 75$

$\Rightarrow 75 \leq 75$.

So, it is $O(g(n)) = O(n^2)$.

ii) let $n = 4$, $c = 3$

$\Rightarrow f(n) \leq c(g(n)) \Rightarrow 2n^2 + 7n - 10 \leq cn^2$

$\Rightarrow 2 \times 16 + 7 \times 4 - 10 \leq 3 \times 16 \Rightarrow 32 + 28 - 10 \leq 48 \Rightarrow 50 \leq 48$

So, it is not in $O(g(n))$

iii) let $n = 6$, $c = 3$

$\Rightarrow f(n) \leq c(g(n)) \Rightarrow 2n^2 + 7n - 10 \leq cn^2$

$\Rightarrow 2 \times 36 + 7 \times 6 - 10 \leq 3 \times 36 \Rightarrow 72 + 42 - 10 \leq 108$

$\Rightarrow 114 - 10 \leq 108 \Rightarrow 104 \leq 108$.

So, $f(n) = O(g(n)) = O(n^2)$.

2. If $3n + 2 = O(g(n))$

Here $f(n) = 3n + 2$

$3n + 2 \leq c * n$.

$3n + 2 \leq 4 * n$ ($\because c = 4$)

If $n = 2$, then $6 + 2 \leq 8 \Rightarrow 8 \leq 8$.

So, $3n + 2 = O(n)$, for $n \geq 2$.

ii) $10n^2 + 4n + 2 = O(g(n))$

$10n^2 + 4n + 2 \leq c * n^2$

let $c = 11$, $10n^2 + 4n + 2 \leq 11n^2$

If $n = 5$, $10 \times 25 + 4 * 5 + 2 \leq 11 * 25$

$= 250 + 20 + 2 \leq 275$

$= 272 \leq 275$. So $10n^2 + 4n + 2 = O(n^2)$, $n \geq 5$.

G.I.F.T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

3. $O(1)$ = A computing time ie:- constant.
$O(n)$ = Linear
$O(n^2)$ = Quadratic
$O(n^3)$ = Cubic
$O(2^n)$ = Exponential
Seven computing times in decreasing order
ie:- $O(1)$, $O(\log n)$, $O(n)$, $O(n\log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$.

iii) Small oh (o) Notation:

1. The functions in small oh (o) notation are the smaller function in Big oh (O) notation. We use small oh (o) notation to denote an upper bound that is not asymptotically tight. This notation is defined as :- $f(n) = o(g(n))$ iff $\exists$ any positive constant $c>0$ and $n_0>0$, such that $f(n) < c(g(n))$, $\forall\ n>n_0$

2. The defenition of O-notation and o-notation are similar. The main difference is that in $f(n) = O(g(n))$, the bound $f(n) \le c(g(n))$, holds for some constant $c>0$, but in $f(n) = o(g(n))$, the bound $f(n) < c(g(n))$ hold for all constant $c>0$.

3. In this notation, the function $f(n)$ becomes insignificant relative to $g(n)$ as it approaches infinity.
ie:- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

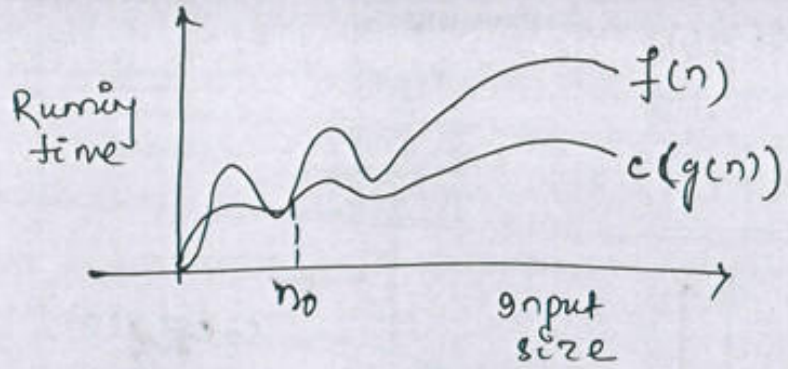
iv) Big Omega (Ω) Notation:

1. The lower bound for the function is provided by Big Omega (Ω) Notation. We can say, the running time of an algorithm $\Omega(g(n))$, if whenever input size is equal to or exceeds some thresholds value n_0, its running time can be denoted by some positive constant 'c' times $g(n)$.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

2. Let $f(n)$ and $g(n)$ are 2 functions from set of natural numbers to set of non-negative real numbers and $f(n)$ said to be $\Omega(g(n))$, i.e:- $\boxed{f(n) = \Omega(g(n))}$, iff $\exists$ a natural number n_0 and a constant $c > 0$, such that

$$\boxed{f(n) \geq c(g(n)),\ \forall\ n \geq n_0}$$



Examples:

1. $f(n) = n^2 + 3n + 4 \Rightarrow f(n) = \Omega(g(n))$, where $g(n) = n^2$

Let $c = 1$, $n = 1 \Rightarrow f(n) \geq c(g(n))$

$\Rightarrow n^2 + 3n + 4 \geq cn^2 \Rightarrow 1 + 3 + 4 \geq 1 \Rightarrow 8 \geq 1$

$\Rightarrow f(n) = \Omega(g(n))$ (proved)

G.I.F.T. CENTRAL LIBRARY BHUBANESWAR

iv) Small omega (ω) Notation:

1. The function small omega (ω) notation are the larger functions of Big omega (Ω) Notation. We use, a notation to denote a lower bound that is not asymptotically tight, we define this notation as:- $\boxed{f(n) = \omega(g(n))}$ $\exists$ some positive constant $c > 0$ and $n_0 > 0$, such that $\boxed{f(n) > c(g(n)),\ \forall\ n \geq n_0}$.

The relation $f(n) = \omega(g(n))$ implies that $\boxed{\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty}$

v) Theta (Θ) Notation:

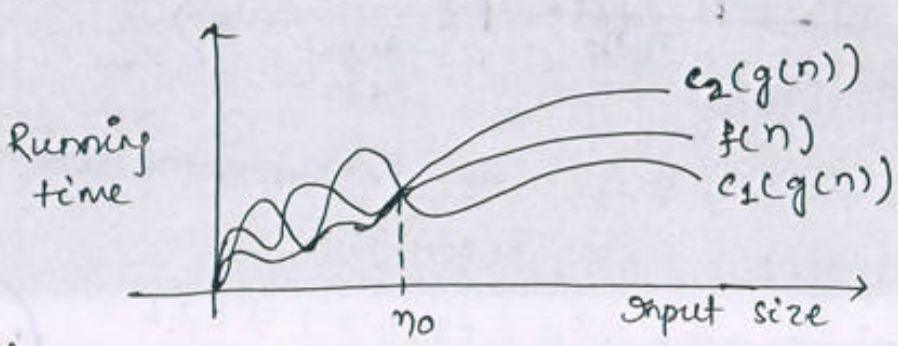
1. For a given function $g(n)$ we denoted by $\Theta(g(n))$, the set of functions $\boxed{f(n) = \Theta(g(n))}$, iff $\exists$ some constant c_1, c_2 and n_0 such that $\boxed{c_1(g(n)) \leq f(n) \leq c_2(g(n)),\ \forall\ n \geq n_0}$

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

2. For all values of 'n' to the right of n_0, the values of $f(n)$ lies at or above $c_1(g(n))$ and at or below $c_2(g(n))$. In other words, $\forall n \geq n_0$, the function $f(n)$ is equal to $g(n)$ to within a constant factor. We say that $g(n)$ is asymptotically tight bound for $f(n)$.

3. The definition of $\Theta(g(n))$ requires that every member $f(n) \in \Theta(g(n))$ be asymptotically non-negative. i.e. that $f(n)$ be non-negative whenever 'n' is sufficiently large.

So, $f(n) = \Theta(g(n)) \Rightarrow \lim_{n \to \infty} \frac{f(n)}{g(n)} = \text{constant 'c'}$



Example:-

$f(n) = 3n+2$

$c_1(g(n)) \leq 3n+2$

let $c_1 = 3$, $3n \leq 3n+2$

if $n=2$, then $3*2 \leq 3*2+2 \Rightarrow 6 \leq 8$.

$\therefore 3n \leq 3n+2$, $n \geq 2$.

$3n+2 \leq c_2(g(n))$

$3n+2 \leq 4*n$ $(\because c_2 = 4)$

if $n=2$, then $3*2+2 \leq 4*2 \Rightarrow 8 \leq 8$.

$\therefore 3n+2 \leq 4*n$, $n \geq 2$.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Array:

1. An array is a collection of similar data elements present in continuous memory locations referred by a unique name. The data type of the elements may be any valid data type like char, int or float. The elements of array share the same variable name, but each element has a different index number known as subscript.

2. An array can be one dimensional array, two dimensional array and multidimensional array. Two dimensional array has 2 subscripts one for row and other one for column. It is also called matrix.

One dimensional array:

1. An array is defined as :-

datatype arrayname[size];

which has one subscript.

Eg:- int a[5];

The initialization of the array is :-

i) int a[5] = {1, 2, 3, 4, 5};

ii) int a[] = {1, 2, 3, 4, 5};

CENTRAL LIBRARY RHUBANESWAR

2. A linear array is a list of a finite number 'n' of homogeneous data element, such that

i) The elements of the array are referenced respectively by an index set consisting of 'n' consecutive numbers.

ii) The elements of the array are stored respectively in successive memory location.

3. The number 'n' of elements is called the length or size of the array. The length of an array can be obtained from the index by the formula :-

Length = Upperbound (UB) − Lowerbound (LB) + 1

where UB = Largest index called upper bound.

LB = Smallest index called lower bound

When lowerbound = 1, then **Length = Upperbound (UB)**

Eg:-

3	5	6	7	9	10
0	1	2	3	4	5
(LB)					(UB)

Length = UB − LB + 1 = 5 − 0 + 1 = 6.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Representation of Linear array in Memory / Finding Address of an element in an 1-D array :

1. Let 'LA' be a linear array in the memory of the computer.
   LOC(LA[K]) = Address of the element LA[K] of the array LA.
2. Base(LA) called the base address of the array LA. Elements of 'LA' are stored in successive memory cells. So, we need to keep track of the address of the 1st element of the array LA, which is the Base(LA).
3. Using this base address, the computer calculate the address of any element of 'LA' by the following formula :-

$$\boxed{LOC(LA[K]) = Base(LA) + W(K - LB)}$$

where W = no. of words per memory cell for an array LA.
or size/width of each element.

Eg :-

A[10]	0	1	2	3	4	5 (K)	6	7	8	9
	5	9	2	11	3	8	7	6	4	1
	101	103	105	107	109	111	113	115	117	119
	LB									UB

Here, base address Base(LA) = 101.
Lowerbound (LB) = 0, Upperbound (UB) = 9.
Size/Width of each element W = 2 bytes,
'K' is the position = 5

So, Address of an element with index K :

$$Loc(LA[K]) = Base(LA) + W(K - LB)$$
$$= 101 + 2 * (5 - 0)$$
$$= 111.$$

So Address of LA[5] = 111.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

Operation on Data Structure:

1. The operation performed on any linear structure whether it be an array or linked list include the following.

i) Creation: Creation of new data structures by considering a set of data.

ii) Insertion: Adding a new data at various positions of existing data structure.

iii) Deletion: Deleting an existing data from the data structure.

iv) Search: Finding the existance and location of the elements with a given value or the records with a given key.

v) Traversal: Visiting each field present in the data structure exactly once for certain processing purpose.

vi) Sorting: Arranging the data or records or elements in a particular order.

vii) Merging: Joining or combining of records from multiple data structures to produce a single file in sorted order.

Memory Representation Data Structures:

1. Any data structures can be represented in two ways in the memory. ie. i) Sequential representation and ii) Linked representation

i) Sequential Representation:

→ A sequential representation maintains the data in continuous memory locations which takes less time to retrieve the data but leads to time complexity during insertion and deletion operations. The major drawback of this is it takes much time for insertion and deletion operations unnecessarily and increasing the complexity of algorithm.

ii) Linked Representation:

→ It maintains the list by means of a link or a pointer between the adjacency elements which need not be stored in continuous memory location. During insertion and deletion operations, links will be created or removed between the elements which takes less time when compared to the corresponding operations of sequential representation. It takes more memory space, which is the drawback of this.

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Entity:

1. An entity is a thing or object in the real world that is distinguishable from all other objects.
   Eg:- Each person in an enterprise is an entity.
2. An entity is represented by a set of attributes.
3. An entity set is a set of entities of the same type that share the same properties or attributes.
   Eg:- The set of all persons who are customers of a given bank can be defined as the entity set customer.
4. Each entity has a value for each of its attributes.
   Eg:- 123 for customer ID is the attributes of the entity set.
5. For each attributes there is a set of permitted values called the domain or value set of the entity set.

Information:

1. Data with given attributes or meaningful and processed data is known as information. field is the single elementary unit of information representing an attributes of an entity.

Record:

1. It is the collection of field values of a given entity.

File:

1. It is the collection of records of the entity in a given entity set.

Space Complexity and Time Complexity:

1. Space Complexity of an algorithm is the amount of memory, it needs to run to completion.
2. The time complexity of an algorithm is the amount of computer time, it needs to run to completion.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Data Structure Using C

Topic:

Operations On Array

Contributed By:

Nihar Ranjan Rout

Gandhi Institute For Technology, GIFT

Operations on One Dimensional Array :

1. The operations on one dimensional array are :-
   i) Creation : Create or entering elements into Array
   ii) Traversal : Processing each element
   iii) Search : Searching for an element.
   iv) Insertion : Inserting an element at required position.
   v) Deletion : Deletion of an element.
   vi) Sorting : Arranging in ascending or descending order
   vii) Merging : It is done in two ways
      a) Join/concatenate two ways into 3rd Array and then sort.
      b) Merge directly into 3rd Array where two arrays are already sorted.

i) Creation of an Array :

Algorithm :

```
CREATE (LA, LB, UB)
    Step 1: Let LA[10], LB, UB, K
    Step 2: LB:=1, UB:=10
    Step 3: Display "Enter 10 Elements"
    Step 4: Repeat, for K:= LB to UB incremented by 1
                INPUT LA[K]
            [End of For]
    Step 5: Exit.
```

The 'C' function for this is :-

```
/* Assume LB and UB are declared as global variables */
    Void CREATION (int LA[])
    {  int K;
       printf ("Enter elements into array");
       for (K = LB; K <= UB; K++)
          scanf ("%d", &LA[K]);
    }
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

(ii) Traversing an Array:

Algorithm:

```
TRAVERSE (LA, LB, UB)
  Step 1: Repeat for K := LB to UB by 1
              Apply PROCESS to LA[K]
          [End of For]
  Step 2: Exit.
```

Here, PROCESS can be visiting each element and do calculation on it. In 'C' Language LB is 0 by default and UB is Last position of Array.

The 'C' function for Traversing a Linear Array :-

```
/* UB representing Upper Bound is declared as global */
Void Traverse(int LA[ ])
  { int K;
   for(K = LB; K <= UB; K++)
     { printf("%d", LA[K]);
     }
  }
```

(iii) Searching for an element in an array:

Algorithm:

```
SEARCH (LA, LB, UB, ITEM)
  Step 1: Let K, Set K := LB
  Step 2: Repeat, while (K <= UB)
      Step 2.1: If LA[K] = ITEM, then
                    Display "Item Found at", K
                [End of If]
      Step 2.2: Set K := K+1
                [End of While]
  Step 3: Exit
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

The 'C'-function for this is :-

```
void search (int LA[], int ITEM)
{
  int K=0;
  for (K=LB; K<=UB; K++)
  {
    if (LA[K]==ITEM)
    {
      printf("Item found at %d", K);
    }
  }
}
```

(iv) Insertion of new element in an array :

1. Inserting an element at the end of the linear array can be easily done, provided the memory space allocated for the array is large enough to accomodate the additional element.
2. If we insert an element in the middle of the array, then on the average half of the element must be moved downward to new location to accomodate the new element and give the order of the other elements.

Algorithm:

INSERTION (LA, LB, UB, ITEM, POS)

Step 1: Set K

Step 2: Repeat, for K := UB to POS, decreasing by 1

LA[K+1] := LA[K]

[End of For]

Step 3: LA[POS] := ITEM

Step 4: Set UB := UB + 1.

Step 5: EXIT

The 'C'-function for this is :-

```
void insertion(int LA[], int ITEM, int POS)
{
  int K;
  for (K=UB; K>=POS; K--)
  {
    LA[K+1] = LA[K];
  }
  LA[POS] = ITEM;
  UB++;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Deletion of an element from an array:

1. Deletion of an element in the middle of the array or somewhere at the array would required that each subsequent element be moved one location upward or under to fill up the array.

Algorithm:

1. Deletion of an element when the position is given is:-

```
DELETION_POS (LA, LB, UB, POS)
  Step 1: Let K
  Step 2: Repeat for K := POS to UB incremented by 1.
              LA[K] := LA[K+1]
          [End of For]
          UB := UB-1
  Step 3: Exit
```

The C-function for this is:-

```
void DELETION_POS(int LA[], int POS)
{ int K;
  for (K= POS; K<= UB; K++)
  { LA[K]:= LA[K+1];
  }
   UB--;
}
```

2. Deletion of an element when the item is given.

```
DELETION_ITEM (LA, LB, UB, ITEM)
  Step 1: let K, S=0, T
  Step 2: Repeat for K:= LB to UB incremented by 1.
        Step 2.1: If (LA[K] = ITEM), Then
            Step 2.1.1: Display "Item found at", K.
            Step 2.1.2: Repeat for T:= K to UB, incremented by 1
                Step 2.1.2.1: LA[T]:= LA[T+1]
                [end of For]
            Step 2.1.3: UB := UB-1.
            Step 2.1.4: S=1
            [End of If]
        [End of For]
  Step 3: If (S=0), Then
        Step 3.1: Display "Item to be deleted is not found".
        [End of If]
  Step 4: Exit
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

The 'C'- function for this is :-

```
void DELETION-ITEM (int LA[ ], int ITEM)
{ int K, S = 0, T;
  for(K = LB; K<=UB; K++)
  { if (LA[K] == ITEM)
    { printf("Item found at %d", K);
      for(T = K; T<= UB; T++)
      { LA[T] = LA[T+1];
      }
      UB--;
      S = 1;
    }
  }
  if (S==0)
  { printf("Item to be deleted is not found");
  }
}
```

Sorting the elements of an array:

Algorithm:

```
SORTING (LA, LB, UB)
  Step 1: Let I, J, K, T
  Step 2: Set I := 1, J := 2.
  Step 3: Repeat for K := UB - 1 to 1 by decrementing by 1.
      Step 3.1: Repeat for I := 1, J := 2 to K by 1.
          Step 3.1.1: If LA[I] > LA[J], then
                      T := LA[I]
                      LA[I] := LA[J]
                      LA[J] := T
                  [End of IF]
      [End of For]
  [End of for]
  Step 4: Exit
```

CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

The C-function for this is :-

```
void SORTING(int LA[])
{ int I, J, K, T;
  for (K = UB - 1; K > 0, K--)
  {
    for (I = 0, J = 1; J <= K; I++, J++)
    { if (LA[I] > LA[J])
      {
        T = LA[I];
        LA[I] = LA[J];
        LA[J] = T;
      } } }
}
```

Merging of Two Arrays :

1. Merging means storing the elements of two or more arrays together into 3rd array in sorted array. It is an operation when we need to compact the elements from two different arrays into a single array.
2. Merging is possible in two ways. That is :-
   i) when the elements of two arrays given are in unsorted manner.
   ii) when the elements of two arrays given are in sorted manner.

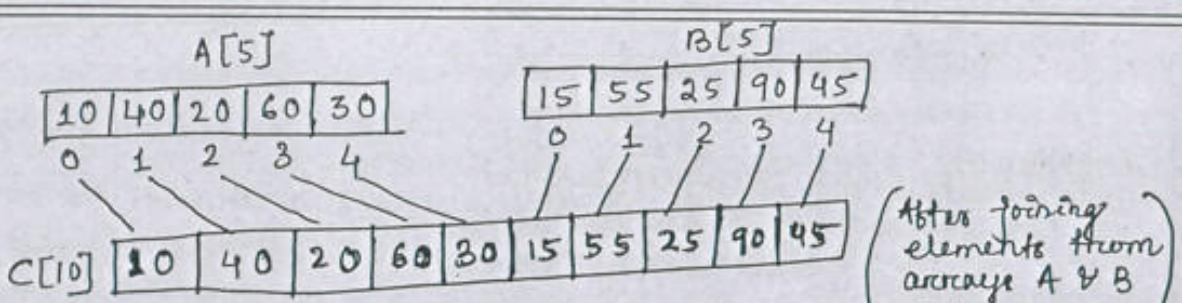
i) When the elements of two arrays given are in unsorted manner :

1. Here we need to join the elements of both the arrays into 3rd array and then apply sorting method to the 3rd array or resultant array.

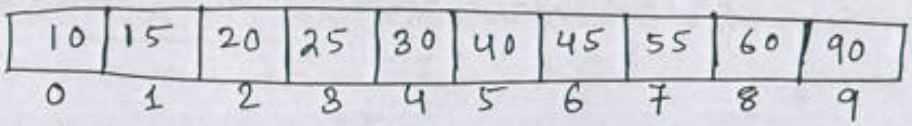
Eg:- Let us consider A[5] contains {10, 40, 20, 60, 30} and B[5] contains {15, 55, 25, 90, 45}.

Assuming C[10] is the resultant array, we need to join all the elements of array A and B together to store in third array. ie C[10] now contains {10, 40, 20, 60, 30, 15, 55, 25, 90, 45}. as shown in the figure.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout



After applying sorting on array C, we get



Algorithm : (Method - I)

MERGE (A, B, C, LB1, UB1, LB2, UB2, LB3, UB3)

Step 1: Let I, K
Step 2: Set K := LB3
Step 3: Repeat for I := LB1 to UB1 incrementing by 1
C[K] := A[I]
K := K + 1
[End of For]
Step 4: Repeat for I = LB2 to UB2 incrementing by 1
C[K] := B[I]
K := K + 1
[End of For]
Step 5: Call SORTING (C, LB3, UB3)
Step 6: Exit

The 'c' function for this is :-

```
void MERGE (int A[], int B[])
{ int C[10], I, K;
  K = 0;
 for (I = 0; I < 5; I++)
  { C[K] = A[I];
    K++;
  }
 for (I = 0; I < 5; I++)
  { C[K] = B[I];
    K++;
  }
```

CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

SORTING(C, 10);

}

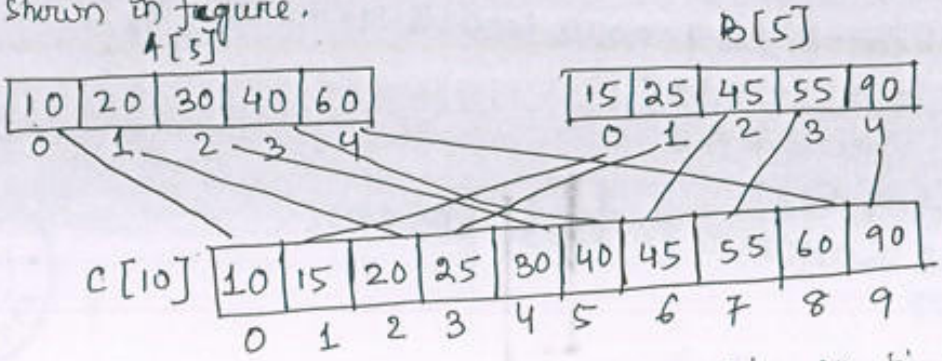
ii) When the elements of two arrays given are in sorted manner :

1. Assume two arrays A and B are given with elements in sorted order. We need to compare elements between both arrays A & B, and always store the smaller element into the 3rd array. Here if some elements are retained either in array A or in B, then they need to be stored directly into to the at the end of 3rd array.

Eg:- Let us consider A[5] = {10, 20, 30, 40, 60} and
B[5] = {15, 25, 45, 55, 90}

Now we need to compare the elements of array 'A' with elements of array 'B' and the smaller element will be copied into array C. If any element are left out from either array A or array B, they will be copied into array 'C' at the end. So, finally we get the elements in array C in sorted order.

They are :- C[10] = {10, 15, 20, 25, 30, 40, 45, 55, 60, 90} as shown in figure.



By comparing the elements of array 'A' with 'B' always smaller elements is stored into array 'C'. Finally left over element i.e. 90 is stored directly into array 'C'.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm:

```
MERGE (A, B, C, LB1, UB1, LB2, UB2, LB3, UB3)
  Step 1: Let I, J, K.
  Step 2: Set I:= LB1, J:= LB2, K:= LB3
  Step 3: Repeat While I <= UB1 AND J<= UB2
          Step 3.1: If A[I] < B[J], then
                      C[K]:= A[I]
                      K:= K+1
                      I:= I+1
                    Else
                      C[K]:= B[J]
                      J:= J+1
                      K:= K+1
                  [End of IF]
          [End of While]
  Step 4: Repeat While I <= UB1
                      C[K]:= A[I]
                      K:= K+1, I:= I+1.
          [End of While]
  Step 5: Repeat, while J <= UB2
                      C[K]:= B[J]
                      K:= K+1, J:= J+1
          [End of While]
  Step 6: Exit
```

The 'C'-function for this is :—

```
void MERGE (int A[], int B[])
{
  int C[10], I, J, K;
  I=0, J=0, K=0;
  While ( I<5 && J<5)
  { if (A[I] < B[J]
      { C[K] = A[I];
```

G.I.F. CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE


```
        K++;
        I++;
    }
    else
    {
        C[K] = B[J];
        K++;
        J++;
    }
    while (I<5)
    {
        C[K] = A[I];    /* To copy rest of the elements if
                           left out in array A */
        I++;
        K++;
    }
    while (J<5)
    {
        C[K] = B[J];    /* To copy rest of the elements if
                           left out in array B */
        J++;
        K++;
    }
    for (K=0; K<10; K++)
    {
        printf("%d", C[K]);
    }
}
```

Multidimensional Array :-

1. While discussing Multidimensional array, we will concentrate on the concepts related to 2-D array or matrix. In general, a two dimensional array is row and column arrangement of elements.
2. A two dimensional MXN array, A is the collection of M x n data elements let each element is specified by a pair of integers such as i, j called subscript with the property that $1 \leq i \leq M$ and $1 \leq j \leq N$.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Memory Representation of 2-D Array:

1. The programming language will store the array in two different ways, that is:- i) Row-Major Order (Row by Row) and ii) Column Major order (column by column)

i) Row-Major Order (Row by Row):

1. In Row-Major order, elements of 2D array are stored on a row-by-row basis beginning from first row. i.e. the majority is given to the rows present in the matrix.

7	9	20	1	11	5	2	17	13	15	21	19
0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3

[Row Major Order Representation]

2. Address of an element when represented in row-major order can be calculated by using the following formula:-

Address of A[i][j] = B + W * [(N * (i - LBR)) + (j - LBC)]

Eg:- Address of A[1][3]

= 100 + 2 * (4 * 1 - 0) + (3 - 0)]

= 100 + 2 * [(4 * 1) + 3]

= 100 + 2 * 7

= 114

In 'C' language as LBR = 0 and LBC = 0, so the above formula can also be represented as per the following:-

Address A[i][j] = B + W * (N * i + j)

ii) Column Major Order (Column by Column):

1. In column-Major order, elements of 2D array are stored on a column by column basis beginning from first column. i.e. the majority is given to the columns present in the matrix.

Eg:- The representation of column-major order is:—

7	11	13	9	5	15	20	2	21	1	17	19
0,0	0,1	0,2	1,0	1,1	1,2	2,0	2,1	2,2	3,0	3,1	3,2

[Column-Major Order Representation]

CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

2. Address of an element when represented in column-major order can be calculated by using the following formula :-

$$\text{Address of } A[i][j] = B + W * [(i - LBR) + M * (j - LBC)]$$

eg :- Address of A[1][3]

$= 100 + 2 * [(1-0) + 3 * (3-0)]$

$= 100 + 2 * [1 + 9]$

$= 100 + 20$

$= 120$

In C Language, when LBR = 0 and LBC = 0, then

$$\text{Address of } A[i][j] = B + W * [i + M * j]$$

A table that shows the address of each location of matrix 'A' in row major order and in column major order is given below.

Row Major Order	Address	Column Major Order	Address
A[0][0]	100	A[0][0]	100
A[0][1]	102	A[1][0]	102
A[0][2]	104	A[2][0]	104
A[0][3]	106	A[0][1]	106
A[1][0]	108	A[1][1]	108
A[1][1]	110	A[2][1]	110
A[1][2]	112	A[0][2]	112
A[1][3]	114	A[1][2]	114
A[2][0]	116	A[2][2]	116
A[2][1]	118	A[0][3]	118
A[2][2]	120	A[1][3]	120
A[2][3]	122	A[2][3]	122

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Sparse Matrix:

1. A matrix with relatively high proportion of zero or null entries is called sparse matrix. A M×N matrix is said to be sparse matrix if many of its elements are zero.

Eg:-

$$\begin{bmatrix} 0 & 0 & 0 & 24 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9 & 0 & 0 \\ 0 & 0 & 0 & 0 & 18 & 0 & 0 \\ 0 & 0 & 0 & 0 & 8 & 0 & 0 \end{bmatrix}_{6\times 7}$$

It is a sparse matrix of 6×7 order with 5 non-zero elements out of 42 entries.

2. A matrix that is not sparse is called dense matrix. The diagonal, trigonal, lower triangular and upper triangular matrices fits into the category of sparse matrix.

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$
Diagonal matrix

$$\begin{bmatrix} 2 & 1 & 0 & 0 \\ 8 & 5 & 8 & 0 \\ 0 & 6 & 7 & 2 \\ 0 & 0 & 4 & 2 \end{bmatrix}$$
Trigonal matrix

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 3 & 5 & 0 & 0 \\ 8 & 6 & 7 & 0 \\ 4 & 1 & 4 & 9 \end{bmatrix}$$
Lower triangular matrix

$$\begin{bmatrix} 2 & 1 & 8 & 6 \\ 0 & 5 & 3 & 4 \\ 0 & 0 & 7 & 2 \\ 0 & 0 & 0 & 9 \end{bmatrix}$$
Upper triangular matrix.

Array Representation of Sparse Matrix:

1. The alternate data structure that we consider to represent a sparse matrix is a triplet. The triplet is a two dimensional array having t+1 rows and 3 columns, where 't' is the total number of non-zero entries.
2. The first row of the triplet contains number of rows, columns and non-zero entries available in the matrix in its 1st, 2nd and 3rd column respectively.
3. In addition, we need to record the size of matrix ie:- no. of rows and no. of columns and non-zero elements, for this purpose, the first element of array of triplets is used where the first field stores no. of rows and 2nd field stores no. of columns and 3rd field stores no. of non-zero elements.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

4. The possible operations on a sparse matrix are :-

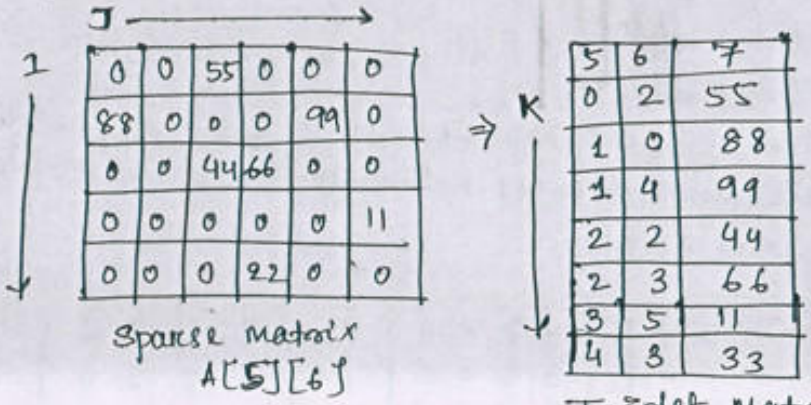
(i) To store the non-zero entries of a sparse matrix into alternate triplet representation

(ii) To display the original matrix from a given triplet representation of matrix.

(iii) To store transpose of the triplet representation.

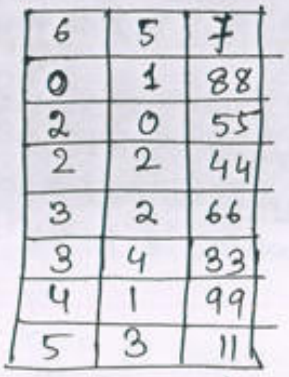
(iv) To display the transpose of original sparse matrix using transpose of triplet.

Eg :- Consider a matrix A[5][6], which contains zeros and non-zeros. Let NZ is a variable contains nonzeros count.

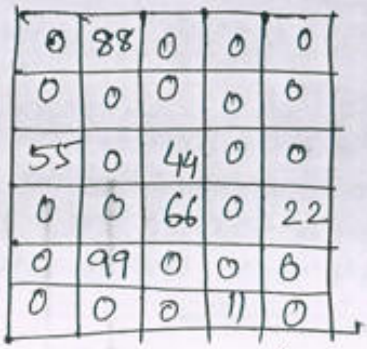


Sparse matrix A[5][6]

Triplet Matrix SP[8][3]



Transpose of triplet matrix TP[8][3]



Transpose of original matrix.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

Algorithm for operations on a Sparse matrix :

Step 1: Let A[10][10], SP[50][3], TP[50][3], I, J, M, N, NZ
Step 2: NZ := 0
Step 3: Display "Enter the row and column size"
Step 4: Input M, N.
Step 5: Repeat, for I := 0 to M-1 Incrementented by 1.
  Step 5.1: Repeat for J := 0 to N-1 Incrementented by 1:
    Input A[I][J]
    If A[I][J] != 0, then
      NZ := NZ + 1
    [End of If]
  [End of step 5.1]
Step 6: If NZ >= (M*N)/2, then Display "Matrix is not sparse"
Step 7: else Display "Matrix is sparse".
  [Calling the procedure to store Non-zero elements from Matrix A into alternate data structure Triplet Matrix SP]
  Step 7.1: CALL ALTERNATE (A, SP, M, N, NZ)
    [Calling the procedure to display original matrix using Triplet Matrix SP]
  Step 7.2: CALL DISPLAY_ORIGINAL (SP, M, N)
    [Calling procedure to transpose of Triplet Matrix Sp into alternate Matrix TP]
  Step 7.3: CALL TRANSPOSE (SP, TP, NZ, N)
    [Calling procedure to display transpose of original matrix using triplet TP]
  Step 7.4: CALL DISPLAY_TRANSPOSE (TP, M, N, NZ)
    [End of step-6]
Step 8: Exit.

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm to Represent a Sparse Matrix in Triplet format :

ALTERNATE (A, SP, M, N, NZ)

Step 1: Let I, J, K := 1.

Step 2: SP[0][0] := M, SP[0][1] := N, SP[0][2] := NZ.

Step 3: Repeat for I := 0 to M-1 incerrented by 1.

Step 3.1: Repeat for J := 0 to N-1 incerrented by 1.

If A[I][J] != 0 then

SP[K][0] := I

SP[K][1] := J

SP[K][2] := A[I][J]

K := K+1.

[End of If]

[End of Step 3.1]

Step 4: Repeat, for K := 0 to NZ incerrented by 1

Display SP[K][0], SP[K][1], SP[K][2]

[End of For]

Step 5: Exit.

C-function:

```
Void Alternate (int A[10][10], int SP[50][3])
{ int I, J, K = 1;
  SP[0][0] = M, SP[0][1] = N, SP[0][2] = NZ;
  for (I = 0; I < M; I++)
  { for (J = 0; J < N; J++)
    { if (A[I][J] != 0)
      { SP[K][0] = I;
        SP[K][1] = J;
        SP[K][2] = A[I][J];
        K++;
      }
    }
  }
  for (K = 0; K <= NZ; K++)
  { printf ("%d %d %d\n", SP[K][0], SP[K][1], SP[K][2]);
  }
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm to Display Original Matrix Using Triplet (SP):

```
DISPLAY_ORIGINAL(SP, M, N)
    Step 1: Let I, J, K
    Step 2: K := 1
    Step 3: Repeat For I := 0 to M-1 Increased by 1.
        Step 3.1: Reapeat For J := 0 to N-1 Increased by 1.
                If (SP[K][0] = I AND SP[K][1] = J), then
                    Display SP[K][2].
                    K := K+1.
                Else
                    Display "0".
                [End of If]
        [End of For Step 3.1]
    [End of For - Step 3]
Step 4: Exit.
```

C - Function:

```
void Display_original(int SP[50][3])
{ int I, J, K;
  K = 1;
  for(I = 0; I < M; I++)
  { for(J = 0; J < N; J++)
    { if(SP[K][0] == I && SP[K][1] == J)
      { printf("%d", SP[K][2]);
        K++;
      }
      else
      { printf(" 0");
      }
      printf("\n");
    }
}
```

G.I.F.T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm to find transpose of triplet matrix (SP) into alternate matrix (TP):

```
TRANSPOSE(SP, TP, NZ, N)
 step 1: Let J, K, T:=1.
 step 2: TP[0][0]:=SP[0][1], TP[0][1]:=SP[0][0],
         TP[0][2]:=SP[0][2].
 step 3: Repeat for J:=0 to N-1 incremented by 1.
            If (SP[K][1]=J), then
               TP[T][0]:=SP[K][1]
               TP[T][1]:=SP[K][0]
               TP[T][2]:=SP[K][2]
               T:=T+1
            [End of If]
         [End of for - step 3.1]
      [End of for - step 3]
Step 4: Repeat for T:=0 to NZ-1 incremented by 1.
           Display TP[T][0], TP[T][1], TP[T][2]
        [End of for]
Step 5: Exit.
```

c-function:

```
void Transpose(int SP[50][3], int TP[50][3])
{ int J, K, T=1;
  TP[0][0]=SP[0][1];
  TP[0][1]=SP[0][0];
  TP[0][2]=SP[0][2];
  for(J=0; J<N; J++)
    { for(K=1; K<NZ; K++)
        { if(SP[K][1]==J)
            { TP[T][0]=SP[K][1];
              TP[T][1]=SP[K][0];
              TP[T][2]=SP[K][2];
              T++;
            }
        }
    }
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
for (T=0; T<=NZ; T++)
  {
   printf("%d %d %d", TP[T][0], TP[T][1], TP[T][2]);
  }
}
```

Algorithm to display transpose of original matrix using transpose of triplet (TP):

```
DISPLAY_TRANSPOSE(TP, M, N, NZ)
  Step 1: Let I, J, T.
  Step 2: T := 1
  Step 3: Repeat for J := 0 to N-1 incremented by 1
     Step 3.1: Repeat for I := 0 to M-1 incremented by 1.
          If (TP[T][0] = J AND TP[T][1] = I), then
               Display TP[T][2]
               T := T+1.
          Else
               Display "0"
          [End of If]
     [End of for - step 3.1]
  [End of for - step-3]
Step 4: Exit.
```

C - function:

```
Void Display_Transpose (int TP[50][3])
  { int I, J, T;
    T = 1;
    for (J=0; J<N; J++)
     { for (I=0; I< M; I++)
        { if (( TP[T][0] == J) && (TP[T][1] == I))
           { printf("%d", TP[T][2]);
             T++;
           }
          else
           { printf("0");
           }
     } } }
```

G.I.F.T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE


```
    Printf("\n");
  }
}
```

Complete C- program for operations on Sparse Matrix :

```
#include<stdio.h>
void Alternate(int [10][10], int [50][3]);
void Display-Original(int [50][3]);
void Transpose(int [50][3], int [50][3]);
void Display-Transpose(int [50][3]);
int M,N,NZ;
main()
{ int A[10][10], SP[50][3], TP[50][3], I,J;
  NZ=0;
  Printf("enter row and column size of Matrix");
  scanf("%d %d", &M, &N);
  Printf("enter elements now");
  for(I=0; I<M; I++)
    { for(J=0; J<N; J++)
      { scanf("%d", &A[I][J]);
        if(A[I][J] != 0)
          { NZ++;
          }
      }
    }
if(NZ>=(M*N)/2)
  { printf("Matrix is not sparse");
    exit(0);
  }
else
  { printf("Matrix is sparse");
    /* calling the function to store Non-Zero elements from
       Matrix A into alternate Triplet Matrix Sp*/
    Alternate(A, SP);
    /* calling the procedure to display original matrix using
       Triplet Matrix SP*/
    Display-Original(SP);
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
/* calling the function to transpose of Triplet Matrix sp into alternate
   matrix TP*/
 Transpose (SP, TP);
/* calling function to display transpose of original matrix using
   Triplet TP*/
   Display_Transpose(TP);
}
void Alternate(int A[][10], int SP[][3])
{ int I, J, K = 1;
   SP[0][0] = M, SP[0][1] = N, SP[0][2] = NZ;
   for(I=0; I<M; I++)
    { for(J=0; J<N; J++)
      { if(A[I][J] != 0)
        { SP[K][0] = I;
          SP[K][1] = J;
          SP[K][2] = A[I][J];
          K++;
        }
      }
    }
   for(K=0; K<=NZ; K++)
    { printf("%d %d %d %d\n", SP[K][0], SP[K][1], SP[K][2]);
    }
}
Void Display_Original (int SP[50][3])
  { int I, J, K;
    K = 1;
    for(I=0; I<M; I++)
     { for(J=0; J<N; J++)
       { if(SP[K][0] == I && SP[K][1] == J)
         { printf("%d", SP[K][2]);
           K++;
```

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
        else
            { printf("0");
            }
        printf("\n");
    }
}
Void Transpose(int SP[50][3], int TP[50][3])
{ int J, K, T = 1;
  TP[0][0] = SP[0][1];
  TP[0][1] = SP[0][0];
  TP[0][2] = SP[0][2];
  for(J=0; J<N; J++)
  { for(K=1; K<NZ; K++)
    { if(SP[K][1] == J)
      { TP[T][0] = SP[K][1];
        TP[T][1] = SP[K][0];
        TP[T][2] = SP[K][2];
        T++;
      }
    }
  for(T=0; T<=NZ; T++)
  { Printf("%d%d%d", TP[T][0], TP[T][1], TP[T][2]);
  }
}
Void Display_Transpose(int TP[50][3])
{ int I, J, T;
  T = 1;
  for(J=0; J<N; J++)
  { for(I=0; I<M; I++)
    { if((TP[T][0] == J) && (T[T][1] == I))
      { printf("%d", TP[T][2]);
        T++;
      }
      else { printf("0");
           }
      printf("\n");
    }
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Data Structure Using C

Topic:

Stack

Contributed By:

Nihar Ranjan Rout

Gandhi Institute For Technology, GIFT

Stack:

1. It is a non-primitive linear data structure, which works on the concept of Last In First Out (LIFO). It is an ordered list, where insertion and deletion can be done at one end only, which is called TOP element of the stack.

eg:- i) stack of plates one above the other
ii) stack of books kept one above the others.

Operation of stack:

1. There are various operation done in stack. That is :-
   i) PUSH: Insertion of an element at the TOP of the stack.
   ii) POP: Deletion of an element from the TOP of the stack.
   iii) PEEP: If we want to know about the information stored at some location in the stack then PEEP operation is required.

G.I.F.T CENTRAL LIBRARY BHUBANESWAR

Memory Representation of stack: Array Representation

1. An one dimensional array can be considered for representing a stack in memory, while representing a stack in the form of an array we take a variable TOP that identifies top most element of the stack.
2. Initially, when the stack is empty TOP = -1. The operation of pushing an ITEM into a stack and the operation of removing an ITEM from the stack is done by PUSH and POP.
3. In executing the procedure PUSH, one must first test whether there is room in the stack for new ITEM, if not then we have the condition known as "OVERFLOW". The condition for overflow is TOP = size - 1, where size is the maximum number of elements that can be hold by the array that represents the stack in memory.
4. In executing the procedure POP, we ~~first~~ must first test whether there is an element in the stack to be deleted, if not then we have the condition known as 'UNDERFLOW". The condition for UNDERFLOW is TOP = -1.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Linked Representation of Stack :

The concept of dynamic memory allocation is implemented by using linked list to represent stack in linked format.

Algorithm For PUSH Operation :

Let stack[size] is an array, TOP = topmost element and ITEM = element to be pushed into the stack. Then the algorithm for PUSH operation is :-

```
PUSH(STACK, TOP, SIZE, ITEM)
    Step 1: If (TOP = SIZE-1), then
                Display "Overflow"
                EXIT
    Step 2: Else
                TOP := TOP + 1
                STACK[TOP] := ITEM
            [End of If]
    Step 3: Exit
```

Algorithm for POP Operation :

```
    Step 1: Let ITEM
    Step 2: If (TOP < 0), then
                Display "underflow"
                EXIT.
    Step 3: else
                ITEM := STACK[TOP]
                display "popped Item is"; ITEM
                TOP := TOP - 1
            [End of If]
    Step 4: Exit.
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm for PEEP:

Step 1: If TOP = -1, then print "underflow" and return.
Step 2: Read the location value 'i'
Step 3: Return the ith element from the top of the stack
Return (STACK[TOP - i + 1])

Algorithm for Update:

When the contained of some location in a stack is to be changed "Update" operation is required.

Update (STACK, ITEM)
Step 1: If TOP = -1, then print "Underflow" and return.
Step 2: Read the location value 'i'.
Step 3: STACK[TOP - i + 1] = ITEM
Step 4: Return.

'C' Program for stack operation:

```
#include<stdio.h>
# define SIZE 10.
int stack[SIZE], TOP = -1, i;
void push();
void pop();
void display();
main()
{ int ch;
  do
  { printf("[1]. PUSH [2]. pop [3]. Display [4]. Exit");
    printf("\n enters your choice [1-4]:");
    scanf("%d", &ch);
    switch(ch)
    {
     case 1:
            push();
            break;
     case 2:
            pop();
            break;
     case 3:
            display();
            break;
     case 4:
            break;
     default:
            printf("Invalid option");
    }
    while(ch != 4);
}
```

GIFT CENTRAL LIBRARY RHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
void push()
  { if (TOP == SIZE-1)
      { printf ("stack overflow");
        return;
      }
    else
      { TOP ++;
        printf("enter the element to push");
        scanf("%d", &stack[TOP]);
      }
  }
void pop()
  { if (TOP == -1)
      { printf("stack underflow");
        return;
      }
    printf("the pop element is :%d", stack[TOP]);
    TOP --;
  }
void display()
  {
    if (TOP == -1)
    { printf ("stack is empty or underflow");
      return;
    }
    printf("the elements in stack are: TOP");
    for (i= TOP; i>=0; i--)
        printf("%d", stack[i]);
  }
```

Application of stack:

1. Stacks are used to pass parameters between functions or a call to a function the parameter and local variable are stored on a stack.
2. High level programming language, such as 'pascal', 'c', that provides for recursion used stack for book keeping. That is:- in each recursive call, there is need to save the current values of parameter local variables and return address.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

Conversion and Evaluation of Arithmetic Expressions:

Arithmetic Expression Notations:

1. In any arithmetic expression, each operator is placed in between two operands i.e. mathematical representation, then this way of representing the arithmetic expression is called as infix expression e.g. A+B.

2. Apart from usual mathematical representation of an arithmetic expression, an expression can also be represented in the following two ways.
   i.e. (i) Polish or Prefix Notation.
   (ii) Reverse Polish or Postfix Notation.

(i) **Polish or Prefix Notation:**
→ The notation, in which the operator symbol is placed before its operands, is referred as polish or prefix notation, eg:- +AB

(ii) **Reverse Polish or Postfix Notation:**
→ The notation, in which the operator symbol is placed after its operands, is referred as reverse polish or postfix notation. Eg:- AB+.

Mathematical Procedure for Conversion:

1. The possible conversions are:-
   (i) Infix to Prefix
   (ii) Prefix to Infix
   (iii) Infix to Postfix
   (iv) Postfix to Infix
   (v) Prefix to Postfix
   (vi) Postfix to prefix.

G.I.F.T CENTRAL LIBRARY BHUBANESWAR

2. Standard Arithmetic Operators and Precedence levels are:-
   ^ (exponential) —: Higher level
   *, /, % —: Middle level
   +, - —: lower level.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

i) Infix to Prefix :

→ Identify the innermost brackets.
→ Identify the operator according to the priority of evaluation.
→ Represent the operator and corresponding operator in prefix notation.
→ Continue this process until the equivalent prefix expression is achieved.

Eg:- (A + B * (C - D ^ E) / F)
= (A + B * (C - [^ D E]) / F)
= (A + B * [- C ^ D E] / F)
= (A + [* B - C ^ D E] / F)
= A + [/ * B - C ^ D E F]
= + A / * B - C ^ D E F.

ii) Prefix to Infix :

→ Identify the operator from right to left order.
→ The two operands which immediately follows the operator are for evaluation.
→ Represent the operator and operands in infix notation.
→ Continue this process until the equivalent infix expression is achieved.
→ If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Eg:- + A / * B - C ^ D E F
= + A / * B - C [D ^ E] F
= + A / * B [C - D ^ E] F
= + A / [B * (C - D ^ E)] F
= + A [B * (C - D ^ E) / F]
= A + B * (C - D ^ E) / F.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

(iii) Infix to Postfix :

→ Identify the inner most branches.
→ Identify the operator according to the priority of evaluation
→ Represent the operator and corresponding operator in postfix notation.
→ Continue this process until the equivalent postfix expression is achieved.

eg:- (A + B * (C - D ^ E) / F)
= (A + B * (C - [DE^]) / F)
= (A + B * [CDE^-] / F)
= (A + [BCDE^-*] / F)
= A + [BCDE^-*F/]
= ABCDE^-*F/+

(iv) Postfix to Infix :

→ Identify the operator from left to right order.
→ The two operands which immediately proceeds the operator are for evaluation.
→ Represent the operator and operands in infix notation.
→ Continue this process until the equivalent infix expression is achieved.
→ If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

eg:- AB&CDE^-*F/+
= AB&C[D^E]-*F/+
= AB[C-D^E]*F/+
= A[B*(C-D^E)]F/+
= A[B*(C-D^E)/F]+
= A+B*(C-D^E)/F.

G I F T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

v) Prefix to Postfix:

→ Identify the operator from ~~right~~ left to ~~left~~ right order.
→ The two operands which immediately follows the operator are for evaluation.
→ Represent the operator and operands in Postfix notation.
→ Continue this process until the equivalent Postfix expression is achieved.
→ If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Eg:- +A/*B-C^DEF
= +A/*B-C[DE^]F
= +A/*B[CDE^-]F
= +A/[BCDE^-*]F
= +A[BCDE^-*F/]
= ABCDE^-*F/+

vi) Postfix to Prefix:

→ Identify the operator from left to right order
→ The two operands which immediately preceeds the operator are for evaluation.
→ Represent the operator and operands in prefix notation.
→ Continue this process until the equivalent prefix expression is achieved.
→ If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Eg:- ABCDE^-*F/+
= ABC[^DE]-*F/+
= AB[-C^DE]*F/+
= A[*B-C^DE]F/+
= A[/*B-C^DEF]+
= +A/*B-C^DEF.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Importance of Postfix Expression:

1. INFIX notation which is rather complex as using this notation one has to remember a set of rules. The rules include BODMAS and ASSOCIATIVITY.
2. In case of POSTFIX notation, the expression, which is easier to work or evaluate the expression as compared to the INFIX expression. In a POSTFIX expression, operands appear before the operator, there is no need to follow the operator preceedence and any other rules.

Conversion of an INFIX expression into Postfix expression using Stack:

G.I.F.T. CENTRAL LIBRARY BHUBANESWAR

Algorithm:

[Assume Q is an Infix expression and P is the corresponding Postfix Notation]

Step 1: PUSH "(" onto STACK and Add ')' at the end of Q.

Step 2: Repeatedly scan from Q Until STACK is empty.

Step 2.1: If Q[I] is an operand, Then Add it to P[J].

Step 2.2: Else If Q[I] is '(', Then PUSH Q[I] onto STACK.

Step 2.3: Else If Q[I] is an operator, Then

Step 2.3.1: While STACK[TOP] is an operator and has higher or equal priority compared to Q[I].

pop operators from STACK and add to P[J].

[End of while]

Step 2.3.2: Push operator Q[I] into STACK

Step 2.4: Else If Q[I] is an ')', Then

Step 2.4.1: Repeatedly pop operators from STACK and add to P[J] until '(' is encountered.

Step 2.4.2: Remove '(' from STACK.

[End of If]

Step 3: Exit. [End of step-2]

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

Procedure for Conversion:

Eg:- Q = (B * C - (D / E ^ F))

Symbol Scanned from Q	Stack	Postfix Expression P
(	(	
B	(	B
*	(*	B
C	(*	BC
-	(-	BC *
(	(- (	BC *
D	(- (	BC * D
/	(- (/	BC * D
E	(- (/	BC * DE
^	(- (/ ^	BC * DE
F	(- (/ ^	BC * DEF
)	(-	BC * DEF ^ /
)		BC * DEF ^ / -

Evaluate:

Evaluation of Postfix expression using Stack:

[Assume 'P' is a postfix expression]

Step 1: Add ')' at the end of P.

Step 2: Repeatedly scan 'P' from left to right until ')' encountered.

Step 2.1: If P[I] is an operand, then Push the operand onto STACK

Step 2.2: else if P[I] is an operator (X) then

Step 2.2.1: Pop two elements from stack (1st element is A and 2nd is B)

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Step 2.2.2: Evaluate B(X)A
Step 2.2.3: Push result back to STACK
[End of If]
[End of step 2]
Step 3: VALUE := STACK[TOP]
step 4: Display VALUE
step 5: Exit

Eg:- Evaluate the following Postfix expression using stack:-
P = BC * DEF ^/-
Where B = 5, C = 6, D = 24, E = 2, F = 3.
So, P = 5, 6, *, 24, 2, 3, ^, /, -

Symbol scanned from Postfix Notation	Stack	A	B	B(X)A
5	5			
6	5, 6			
*	30	6	5	5*6 = 30
24	30, 24			
2	30, 24, 2			
3	30, 24, 2, 3			
^	30, 24, 8	3	2	2^3 = 8
/	30, 3	8	24	24/8 = 3
-	27	3	30	30-3 = 27.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

'c' program for conversion of an infix expression to postfix expression:

```
#include<stdio.h>
int top=-1;
void push(char);
char pop(void);
int op(char);
int priority(char);
char Q[100], P[100], S[100];
main()
{
  int i, j, k, r;
  printf("enter a string");
  scanf("%s", Q);
  for(i=0; Q[i]!='\0'; i++).
   Q[i]=')';
   top++;
   S[top]='(';
  i=0; j=0;
  while(top!=-1)
   {
     /* if an operand is found */
     else if(op Q[i]==2)
       { push(Q[i]);
  /* if an operator is found at Q[i] then Repeatedly
   pop from the top of stack for each operator which
   has same or higher precedence than the operator found*/
  else if(op Q[i]==3)
   { while(op(S[top])==3 && priority(S[top])>=priority(Q[i]))
     {
       P[j]=pop();
        j++;
     }
    push(Q[i]);
  }
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
/* If a right parenthesis is found, then repeatedly pop from stack
   and add it to 'p' until a left parenthesis is found. Remove it
   from the stack and do not add it to p */
  else if(op(Q[i]) == 4)
  { while(op(S[top]) != 2)
    {
      P[j] = pop();
      j++;
    }
    top--;
  } i++;
  P[j] = '\0';
  Printf("\n\t%c\t", Q[i-1]);
  for(r=0; r<=top; r++)
      Printf("%c", S[r]);
  Printf("\t%s", P);
 }
}
int op(char x)
{ if(x == '(')
    return 2;
  else if(x == '+' || x == '-' || x == '*' || x == '/' || x == '^')
    return 3;
  else if(x == ')')
    return 4;
  else
    return 1;
}
int priority(char x)
{ if(x == '+' || x == '-')
    return 1;
```

G.I.T. CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
else if(x=='*' || x=='/')
        return 2;
 else
        return 3;
}

void push(char x)
 {
  top++;
  S[top]=x;
}
char pop(void)
  {
   char x;
   x = S[top];
   top--;
   return x;
}
```

C - program for Post-fix Evaluation :

```
#include<stdio.h>
#include<math.h>
int op(char);
void push(int);
int pop(void);
int top = -1;
int stack[100], A,B,R, Value;
char P[100];
main()
{
   int i,j;
    printf(" enter the postfix expression ");
    scanf("%s", P);
    for(i=0; P[i] != '\0'; i++)
      P[i] = ')';
    for(i=0; P[i] != '\0'; i++)
      {
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE


```
if (op[P[i]] == 1)
{
  printf("enter value of %c", P[i]);
  scanf("%d", &value);
  push(value);
}
else if (op[P[i]] == 3)
{
  A = pop();
  B = pop();
  switch (P[i])
  {
    case '^' : R = pow(B,A);
               break;
    case '*' : R = B*A;
               break;
    case '/' : R = B/A;
               break;
    case '+' : R = B+A;
               break;
    case '-' : R = B-A;
               break;
  }
  push(R);
}
printf("Result is %d", stack[0]);
}
void push(int x)
{ top ++;
  stack[top] = x;
}
int pop(void)
{
  int R;
  R = stack[top];
  top --;
  return R;
}
int op(char x)
{ if (x == '+' || x == '-' || x == '*' || x == '/' || x == '^')
    return 3;
  else return 1;
}
```

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Conversion of infix expression to prefix expression using stack:
Assume 'Q' is an infix expression. Consider there are two stacks S1 and S2 exist. The following algorithm converts the infix expression into its equivalent prefix notation.

Algorithm:

Step 1: Add left parenthesis '(' at the beginning of the expression 'Q'.
Step 2: PUSH ')' onto stack S1.
Step 3: Repeatedly scan Q in right to left order, until stack S1 is empty.
  Step 3.1: If Q[I] is an operand, then PUSH it onto stack S2.
  Step 3.2: Else if Q[I] is ')', then PUSH it onto stack S1.
  Step 3.3: Else if Q[I] is an operator (OP), then
    Step 3.3.1: Set X := POP(S1)
    Step 3.3.2: Repeat while X is an operator AND (Precedence(X) > Precedence(OP))
      PUSH(X) onto stack S2
      Set X := POP(S1)
    [End of while-Step-3.3.2]
    Step 3.3.3: PUSH(X) onto stack S1
    Step 3.3.4: PUSH(OP) onto stack S1.
  Step 3.4: Else if Q[I] is '(', then
    Step 3.4.1: Set X := POP(S1)
    Step 3.4.2: Repeat, while (X != ')') (until right parenthesis found)
      Step 3.4.2.1: PUSH(X) onto stack S2
      Step 3.4.2.2: Set X := POP(S1)
    [End of while-Step 3.4.2]
  [End of If-Step 3.1]
[End of loop-step-3]
Step 4: Repeat, while stack S2 is not empty.
  Step 4.1: Set X := POP(S2)
  Step 4.2: Display X.
[End of while]
Step 5: Exit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Procedure of Conversion :

Eg :- Q = (A+B*C*(M*N^P+T)-G+H

Symbol Scanned from Q	Stack S1	Stack S2.
H	)	H
+	)+	H
G	)+	HG
-	)+-	HG
)	)+-)	HG
T	)+-)+	HGT
+	)+-)+	HGT
P	)+-)+	HGTP
^	)+-)+^	HGTP
N	)+-)+^	HGTPN
*	)+-)+*	HGTPN^
M	)+-)+*	HGTPN^M
(	)+-	HGTPN^M*+
*	)+-*	HGTPN^M*+
C	)+-*	HGTPN^M*+C
*	)+-**	HGTPN^M*+C
B	)+-**	HGTPN^M*+CB
+	)+-+	HGTPN^M*+CB**
A	)+-+	HGTPN^M*+CB**A-
C		HGTPN^M*+CB**A+-+

So, the prefix Notation, We can get by poping all the symbols from stack S2. ie:- +-+A**BC+*M^NPTGH.

G I F T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Evaluation of prefix expression :

[Assume P is a Prefix Expression]

Step 1 : Add ')' at beginning of the prefix expression.

Step 2 : Repeatedly scan from P in right to left order until ')' encountered.

Step 2.1 : If P[I] is an operand, then
PUSH the operand onto STACK

Step 2.2 : Else if P[I] is operator (OP), then

Step 2.2.1 : Pop two elements from STACK
(1st element is A and 2nd element is B)

Step 2.2.2 : Evaluate A (OP) B.

Step 2.2.3 : push result back to STACK

[End of If]

[End of loop - step-2]

Step 3 : VALUE := STACK[TOP]

Step 4 : Display VALUE

Step 5 : Exit.

Example :

Evaluate the following prefix Expression using stack :

P = (-, *, 3, +, 16, 2, /, 12, 6)

Symbol scanned from Prefix Expression in Right to Left order	Stack	A	B	A (OP) B
6	6			
12	6, 12			
/	2	12	6	12/6 = 2
2	2, 2			
16	2, 2, 16			
+	2, 18	16	2	16+2 = 18
3	2, 18, 3			
*	2, 54	3	18	3*18 = 54
-	52	54	2	54-2 = 52

Finally, the value in the stack is 52.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

'c' program for conversion from infix to prefix expression:-

```
#include<stdio.h>
char s[50], s2[50], Q[50], P[50];
int t1=-1, t2=-1;
void s1push(char);
void s1pop(void);
void s2push(char);
char s2pop(void);
int op(char);
int priority(char);
main()
{
   int i, k, r;
   char x;
   printf("enter a string");
   scanf("%s", Q);
   for(i=0; Q[i]!='\0'; i++)
   for(i; iy=0; i--)
        Q[i+1]=Q[i];
   Q[0]='(';
   Printf("%s", Q);
   t1=0;
   S1[t1]=')';
   for(i=0; Q[i]!='\0'; i++)
   i--;
   while(t1!=-1)
   {
      /* if an operand is found */
      if(op(Q[i])==1)
      { s2push(Q[i]);
      }
      /* if an right parenthesis is found */
      else if(op(Q[i])==4)
      { s1push(Q[i]);
      }
```

G I F T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE


```
/* If an operator is found */
else if(op(a[i]) == 3)
 { x = s1pop();
   while(op(x) == 3 && priority(x) > priority(a[i]))
      { s2.push(x);
        x = s1.pop();
      }
   s1.push(x);
   s1.push(a[i]);
 }
/* If left parenthesis found */
  else if(op(a[i]) == 2)
   { while(op(s1[t1]) != 4)
       { x = s1.pop();
         s2.push(x);
       }s1.pop();
   }
 i--;
}
 printf("\n\n\n the prefix notation is");
  while(t2 != -1)
    { x = s2.pop();
      printf("%c", x);
    }
}
int op(char x)
 { if(x == '(')
      return 2;
   else if(x == '+' || x == '-' || x == '*' || x == '/' || x == '^')
      return 3;
   else if(x == ')')
      return 4;
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
    else return 1;
}
int priority (char x)
{ if(x=='+' || x=='-')
      return 1;
  else if(x=='*' || x=='/')
      return 2;
  else return 3;
}
void s1push(char x)
{ t1++;
  s1[t1]=x;
}
char s1pop(void)
{ char x;
  x=s1[t1];
  t1--;
  return x;
}
void s2push(char x)
{ t2++;
  s2[t2]=x;
}
char s2pop(void)
{ char x;
  x=s2[t2];
  t2--;
  return x;
}
```

G I F T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

'C' program for prefix Evaluation:

```
#include<stdio.h>
#include<math.h>
int op(char);
int calc(char, int, int);
void push(int);
int pop(void);
int s[50];
char p[50];
int top = -1;
main()
{
  int i, k, a, b, c;
  printf("enter prefix notation");
  scanf("%s", p);
  for(i=0; p[i] != '\0'; i++)
  for( ; i>=0; i--)
  p[i+1] = p[i];
  p[0] = '(';
  for(i=0; p[i] != '\0'; i++)
  i--;
  while(p[i] != '(')
  { if(op(p[i])==2)
    {
      printf("enter the value of %c", p[i]);
      scanf("%d", &k);
      push(k);
    }
    else
    {
      a = pop();
      b = pop();
      c = calc(p[i], b, a);
      push(c);
    }
    i--;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
printf("the evaluated value is %d", S[top]);
}
int op(char x)
{
  if (x == '+' || x == '-' || x == '*' || x == '/' || x == '^')
      return 1;
  else if (x == '(')
      return 3;
  else
      return 2;
}
int calc(char x, int a, int b)
{ int c;
  switch(x)
  {
    case '+': c = b+a; break;
    case '-': c = b-a; break;
    case '*': c = b*a; break;
    case '/': c = b/a; break;
    case '^': c = pow(b,a); break;
  }
    return c;
}
void push(int x)
{
  top++;
  S[top] = x;
}
int pop(void)
{
  int t;
  t = S[top];
  top--;
  return t;
}
```

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Data Structure Using C

Topic:

Queue

Contributed By:

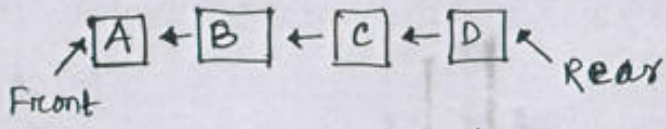
Nihar Ranjan Rout

Gandhi Institute For Technology, GIFT

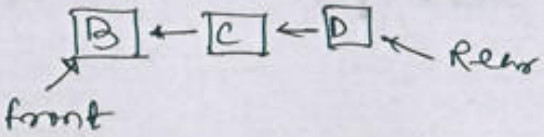
Queue:

1. Queue is a linear non-primitive data structure, works on the concept First in first out (FIFO), where insertion takes place at one end called REAR end and Deletion takes place at other end called front end.

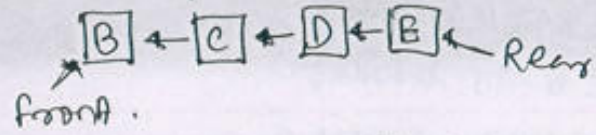
Eg:- Consider below a list of elements exist.



After deleting an element from front end :-



After inserting element 'E' at Rear end :-



Memory Representation of Queue:

1. Array Representation :-

The concept of static memory allocation can be implemented by using ~~linked list to represent~~ a one dimensional array to represent a queue in memory. We consider two variables FRONT and REAR which holds the index of front and rear end of the queue.

2. Linked Representation:

The concept of dynamic memory allocation is implemented by using linked list to represent queue in linked format.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

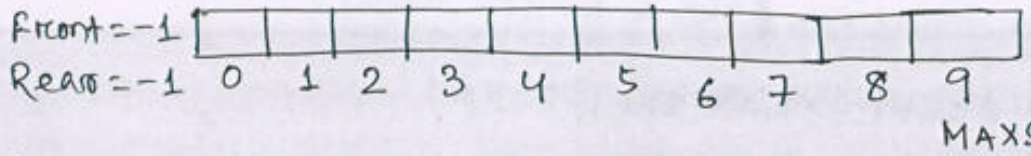
Types of Queues :

1. Linear Queue : Insertion Operation and Deletion operation :

   1. In the Queue, while inserting a new element, one must test whether there is a room available at Rear end or not if room is available then the element can be inserted. otherwise Overflow situation occurs ie:- Queue is full.

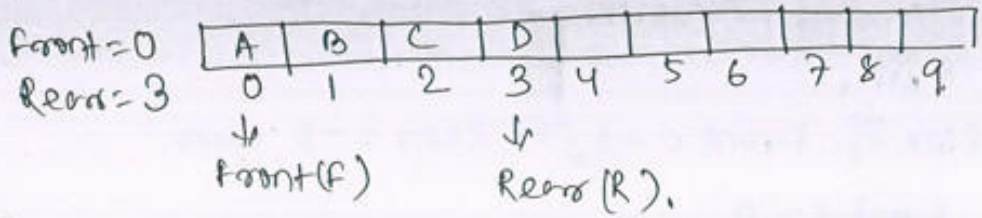
   2. Deletion of an element can be done from the room that is identified by FRONT and FRONT will identify the next room of the queue. Underflow situation occurs when no element is present in the queue. ie:- when front and rear identifies NULL and deletion operation is performed on it.

Eg:- Let Q[MAXSIZE] is an array that represents a queue in the memory and at the beginning, it is empty, and MAXSIZE = 10, ie:- Front = Rear = NULL or -1,

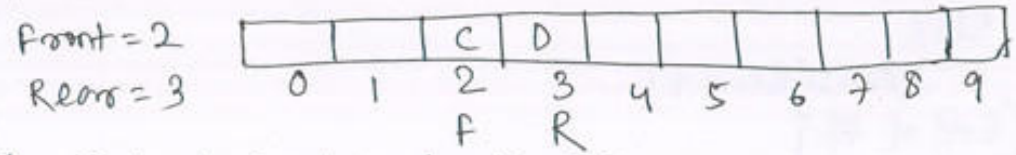


G F T CENTRAL LIBRARY RHUBANESWAR

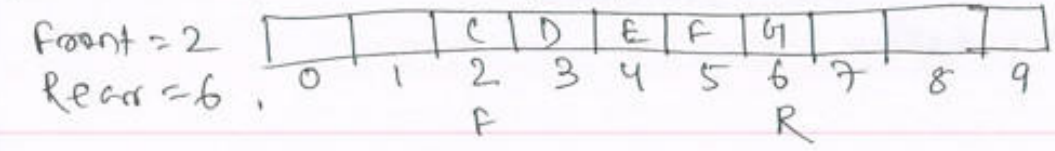
i) Insert 4 elements A, B, C, D.



ii) Delete 2 elements



iii) Insert 3 elements E, F, G.



Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

From the example, it is clear that-

i) For each insertion in the queue the value of REAR is incremented by 1 ie:- REAR = REAR + 1.

ii) For each deletion in the queue the value of front is incremented by FRONT = FRONT + 1.

Note:

1. After all the elements inserted into the Queue, Rear will identify position at MAXSIZE - 1. ie:- Queue is full.
2. Even though the memory locations are vacant at FRONT end, where Rear identified MAXSIZE-1 position, we can't insert a new element.
3. When Front = Rear = Null (ie:- queue is empty), we can't perform deletion operation.
4. When Front = Rear only one element is present in Queue.
5. When front is at MAXSIZE - 1, means Queue having one element. After deletion Front need to be reset to NULL.

Algorithm for insertion into a linear Queue:

QINSERT (QUEUE[MAXSIZE], ITEM)

Step 1: If Rear = MAXSIZE-1, then
Display "Overflow or Queue full"
Exit.

Step 2: Else if front = -1, or Rear = -1, then
front := 0
Rear := 0

Step 3: Else
Rear := Rear + 1
[End of if]

Step 4: QUEUE[Rear] := ITEM

Step 5: Exit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

C-function:

```
void insertion(int Q[], int item)
{
  if(rear == maxsize-1)
  {
    printf("\n queue overflow");
    exit(0);
  }
  else if(front == -1 || rear == -1)
  {
    front = 0;
    rear = 0;
  }
  else
  {
    rear = rear + 1;
  }
  Q[rear] = item;
}
```

G I P T CENTRAL LIBRARY BHUBANESWAR

Algorithm for deletion in a linear queue:

Assume QUEUE[MAXSIZE] is an array which represents a queue, FRONT and REAR identifies the index of front and rear end respectively. ITEM holds the element to be deleted from Queue.

QDELETE (QUEUE[MAXSIZE])

Step 1: Set ITEM

Step 2: If front = -1 or Rear = -1, then
Display "Underflow" or Queue empty"
Exit
[End of If]

Step 3: ITEM := QUEUE[front]

Step 4: Display "Deleted Item is", ITEM

Step 5: Else if front = Rear, then [when queue has only one element]
front := -1
Rear := -1

step 6: Else
front := front + 1.

step 7: Exit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

C-function:

```
void deletion (int a[])
{
  int item;
  if (front == -1 || rear == -1)
  { printf ("\n Queue underflow")
    exit(0);
  }
  item = a[front];
  if (front == rear)
  { front = -1;
    rear = -1;
  }
  else.
  { front = front + 1;
  }
  printf ("\n deleted item is %d", item);
}
```

C-Program for implementing Linear queue:

```
#include<stdio.h>
#include<process.h>
#define maxsize 10
int a[maxsize], front = -1, rear = -1, item;
void insertion(int[], int);
void deletion(int[]);
void display(int[]);
main()
{
  int option;
  do
  {
    printf("\n 1. insert\n 2. delete\n 3. display\n 4. exit\n\n
            enter option");
    scanf("%d", &option);
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
switch(option)
{
 case 1: printf("\n enter item");
         scanf("%d", &item);
         insertion(Q, item);
          break;
 case 2: deletion(Q);
          break;
 case 3: display(Q);
          break;
 case 4: exit(0);
          break;
 default: printf("\n Invalid entry");
}
}while(option!=4);
}

void insertion(int Q[], int item)
{ if(rear == maxsize-1)
  { printf("\n queue overflow");
    exit(0);
  }
 else if(front == -1 || rear == -1)
  {
     front = 0;
     rear = 0;
  }
 else
    { rear = rear + 1;
    }
   Q[rear] = item;
}
```

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
void deletion(int Q[])
{ int item;
  if(front == -1 || rear == -1)
  { printf("\n Queue Underflow");
    exit(0);
  }
  item = Q[front];
  if(front == rear)
  { front = -1;
    rear = -1;
  }
  else { front = front + 1;
  }
  printf("\n deleted item = %d", item);
}
void display(int Q[])
{ int i;
  if(front != -1)
  { printf("\n items at queue\n");
    for(i = front; i <= rear; i++)
      printf("\t%d", Q[i]);
  }
}.
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Circular Queue:

1. It is a queue of circular nature, in which it is possible to insert a new element at the first location, when the last location of the queue is occupied.

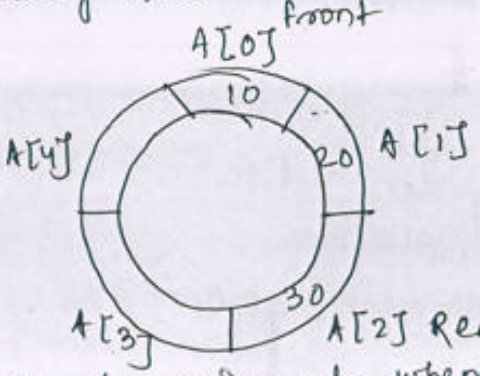
In other words, if an A[MAXSIZE] represents a circular queue, then after inserting an element at (MAXSIZE - 1)th location of the array, the next element will be inserted at the very first location at index 0.

Advantage of circular queue over linear queue:

→ In case of linear queue, if Rear is at MAXSIZE - 1, then queue is considered as full. Here even though memory locations are vacant at beginning still we can not use them, as for each insertion rear is incremented and here REAR goes out of bounds. This drawback can be overcome by circular queue representation.

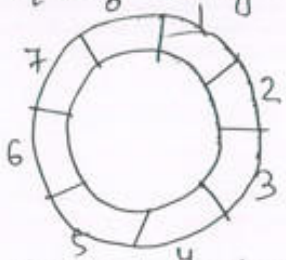
Circular Queue Representation:

Eg:- Let an array A[5] is maintained in form of circular queue.

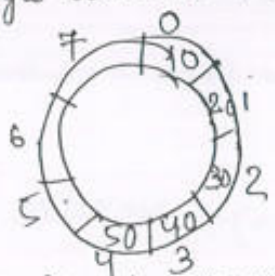


Here overflow situation arises only when Rear is before the front position.

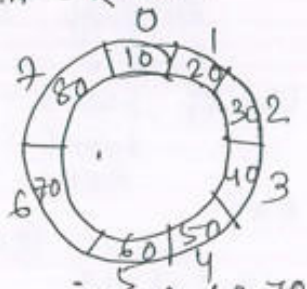
Eg:- When Queue is empty, the front = NULL or -1 and when Queue is full, the front = 0 and Rear = MAXSIZE - 1. When the queue having a single element the front = Rear.



Queue is empty
Front = Rear = -1.

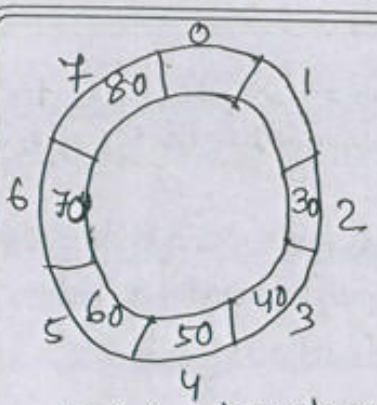


insert 10, 20, 30, 40, 50
front = 0
Rear = 4

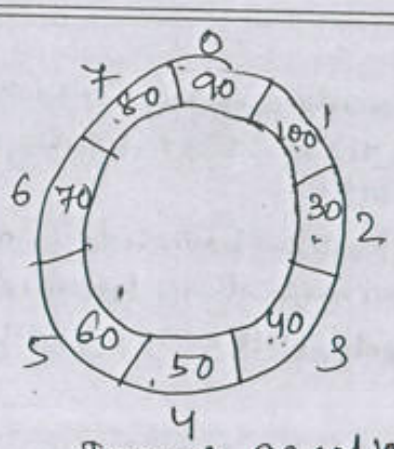


insert 60, 70, 80
front = 0, rear = 7.
Queue is full.

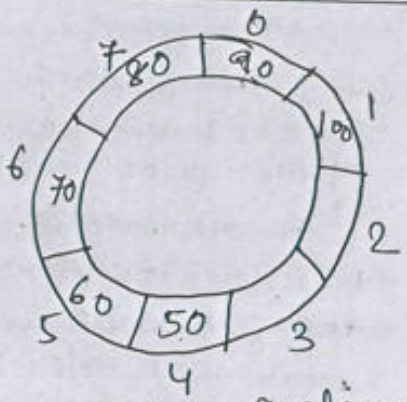
Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout



Delete two elements
Front = 2
Rear = 7



Insert 90 and 100
front = 2
Rear = 1.



Delete 2 elements
front = 4
rear = 1.

iii) If Rear = MAXSIZE - 1, and free locations exist, then
Rear = 0.

iii) If front = MAXSIZE - 1 and Rear is less than N, then
Front = 0.

During each insertion, to find the next Rear the formula is :-

Rear = (Rear + 1) % MAXSIZE

Similarly during each deletion to find the next front position the formula is :-

Front = (front + 1) % MAXSIZE.

Algorithm for Insertion in a circular queue :

CQINSERT(QUEUE[MAXSIZE], ITEM)
Step 1: If (front = 0 and Rear = MAXSIZE - 1) or
(front = (Rear + 1) % MAXSIZE), then
Display "Overflow"
Exit.
Step 2: Else If front = -1 or Rear = -1 then
front := 0
Rear := 0.
Step 3: Else
Rear := (Rear + 1) % MAXSIZE
[End of If]
Step 4: QUEUE[Rear] := ITEM
Step 5: Exit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

C - function:

```
void insertion(int a[], int item)
{
  if((front == 0 && rear == maxsize-1) || (front == (rear+1)%maxsize))
  {
    printf("\n Queue overflow");
    return;
  }
  else if(front == -1 || rear == -1)
  {
    front = 0;
    rear = 0;
  }
  else
  {
    rear = (rear+1)%maxsize;
  }
  a[rear] = item;
}
```

Algorithm for deletion in a circular queue:

CQDELETION(QUEUE, FRONT, REAR)

Step 1: Let ITEM

Step 2: If front = -1 or Rear = -1, then
Display "Queue is empty or Underflow"
Exit
[End of if]

Step 3: ITEM := QUEUE[Front]

Step 4: Display "Deleted Item is", ITEM.

Step 5: If front := Rear, then
front := -1
Rear := -1

Step 6: Else
front := (front + 1)% MAXSIZE
[End of If]

Step 7: Exit.

G.I.F.T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

c-function:

```
Void deletion(int Q[])
{ int item;
  if(front == -1 || rear == -1)
  { printf("\n Queue Underflow");
    return;
  }
  item = Q[front];
  if(front == rear)
  { front = -1;
    rear = -1;
  }
  else
  { front = (front + 1) % maxsize;
  }
  printf("\n deleted item is %d", item);
}
```

'C' Program: implementing circular queue and its operation:

```
#include<stdio.h>
#define maxsize 5
int Q[maxsize], front = -1, rear = -1, item;
void insertion(int [], int);
void deletion(int []);
void display(int []);
main()
{ int option;
  do
  { printf("\n 1. insert\n 2. delete\n 3. display\n 4. exit\n\n enter option");
    scanf("%d", &option);
    switch(option)
    { case 1: printf("\n enter item");
              scanf("%d", &item);
              insertion(Q, item);
              break;
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
        case 2: deletion(a);
                break;
        case 3: display(a);
                break;
        case 4: exit(0);
                break
        default: printf("\n Invalid option");
    }
  }
  while(option!=4);
}
void insertion(int a[], int item)
{
  if((front==0 && rear==maxsize-1) || (front==(rear+1)%maxsize))
    {
      printf("\n queue overflow");
      return;
    }
  else if(front==-1 || rear==-1)
    {
      front=0;
      rear=0;
    }
  else
    {
      rear=(rear+1)%maxsize;
    }
  a[rear]=item;
}
void deletion(int a[])
  { int item;
    if(front==-1 || rear==-1)
      { printf("\n Queue Underflow");
        return;
      }
    item=a[front];
    if(front==rear)
      { front=-1;
        rear=-1;
      }
```

GIFT CENTRAL LIBRARY BHUBANESWAR

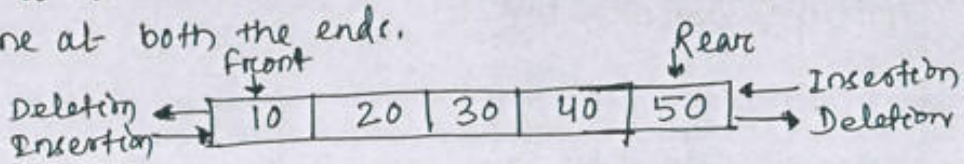
Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
else
 {
   front = (front +1) % maxsize;
 }
 printf("\n deleted item = %d", item);
}
void display(int q[])
{
  int i;
  if (front != -1)
   {
     if (front <= rear)
     {
       printf("\n items at queue\n");
       for( i= front; i<=rear; i++)
            printf("\t%d", q[i]);
     }
     else
      {
         printf("\n items at queue\n\n");
         for(i= front; i<maxsize; i++)
              printf("\t%d", q[i]);
          for(i=0; i<=rear; i++)
              printf("\t%d", q[i]);
      }
   }
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Double Ended Queue (DEQUE)

1. It is a kind of queue, in which the elements can be added or removed at either ends but not in the middle. In other words, it is a linear list in which insertion and deletion can be done at both the ends.



2. There are 2 forms of Deques such as :-
   a) Input Restricted Deque and b) Output Restricted Deque

a) Input Restricted Deque
→ It is a deque, which restricts insertion at only one end, but allows deletion at both ends of the list. So in input restricted deque, the following operations are possible :-
   i) Deletion at front end
   ii) Deletion at Rear end
   iii) Insertion at front or Rear end.

G I P T CENTRAL LIBRARY BHUBANESWAR

b) Output Restricted Deque
→ It is a deque, which restricts deletion at only end but allows insertion at both ends of the list. So, in output restricted deque, the following operations are possible :-
   i) Insertion at front end
   ii) Insertion at Rear end
   iii) Deletion at front or Rear end.

3. Insertion and deletion is allowed at both end of Deque, so the operations possible in deque are :-
   i) Insertion at Rear end (similar to linear queue)
   ii) Deletion at front end (similar to linear queue)
   iii) Insertion at front end
   iv) Deletion at Rear end.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

'C' - function for insertion at rear end of deque:

```
void dqinsert_rear(int Q[], int ITEM)
  {if (rear == Maxsize-1)
     { printf("Deque is full");
     }
   else
     {
       Rear = Rear+1;
       Q[Rear] = ITEM;
     }
  }
```

'C' - function for insertion at front end of deque:

```
void dqinsert_front(int Q[], int Front, int Rear, int ITEM, int maxsize)
 {if (Front ==0)
    { printf("Deque is full");
    }
  else
    {
      front = Front -1;
      Q[FRONT] = ITEM;
    }
 }
```

'C' - function for deletion at front end of deque:

```
void dqdelete_Front(int Q[])
 { int ITEM;
   if(Front == -1)
    { printf("Deque is empty");
    }
   else if (front == Rear)
      { ITEM = Q[front];
        front = Rear = -1;
        printf("Deleted item is %d", ITEM);
      }
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
else
{
    ITEM = Q[front];
    front = front+1;
    printf("Deletem Item is %d", ITEM);
}
}
```

C- function for deletion at rear end of deque :

```
void dqdelete_rear(int Q[])
{
int ITEM;
if(front == -1 || Rear == -1)
{
    printf("Deque is empty");
    exit(0);
}
ITEM = Q[Rear];
if(front == Rear)
{
    front = Rear = -1;
}
else
{
    Rear = Rear - 1;
}
printf("Deleted Item is %d", ITEM);
}
```

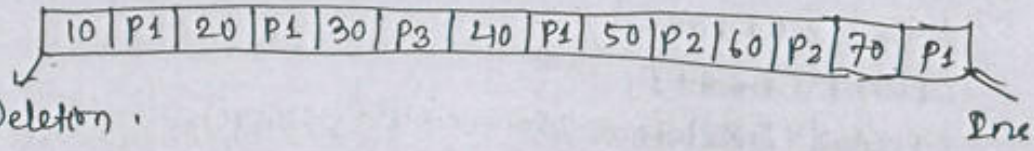
GIFT CENTRAL LIBRARY BHUBANESWAR

Priority Queue :

1. A priority Queue is a collection of elements such that each element has a priority level and according to the priority level, the elements are deleted. In the queue, an element of higher priority is processed before any element of lower priority. When two or more elements have same priority levels then they are processed according to the order in which they

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

are added to the queue.



2. Priority queues are classified into two categories, they are :-

   i) Ascending priority queue.
   ii) Descending priority queue.

Ascending priority queue : It is a collection of elements in which the items or elements are inserted randomly and from which the smallest element can be deleted.

Descending Priority queue : It is a collection of elements in which the items or elements are inserted randomly and from which the largest element can be deleted.

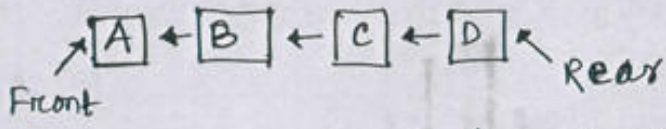
—X—

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

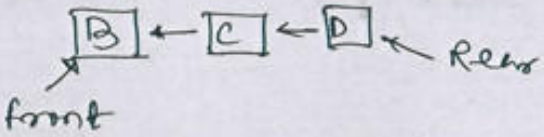
Queue:

1. Queue is a linear non-primitive data structure, works on the concept First in first out (FIFO), where insertion takes place at one end called REAR end and Deletion takes place at other end called front end.

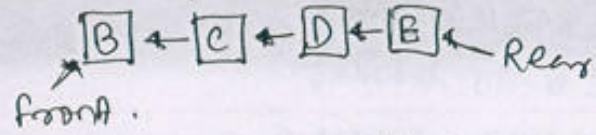
Eg:- Consider below a list of elements exist.



After deleting an element from front end :-



After inserting element 'E' at Rear end :-



Memory Representation of Queue:

1. Array Representation :-

The concept of static memory allocation can be implemented by using ~~linked list to represent~~ a one dimensional array to represent a queue in memory. We consider two variables FRONT and REAR which holds the index of front and rear end of the queue.

2. Linked Representation:

The concept of dynamic memory allocation is implemented by using linked list to represent queue in linked format.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

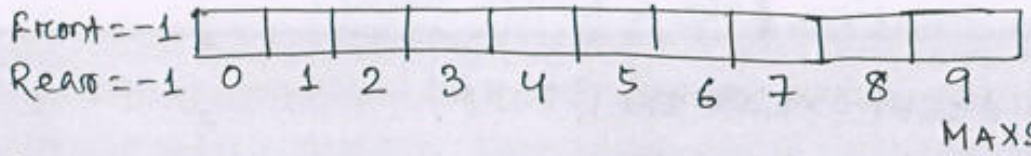
Types of Queues :

1. Linear Queue : Insertion Operation and Deletion operation :

   1. In the Queue, while inserting a new element, one must test whether there is a room available at Rear end or not if room is available then the element can be inserted. otherwise Overflow situation occurs ie:- Queue is full.

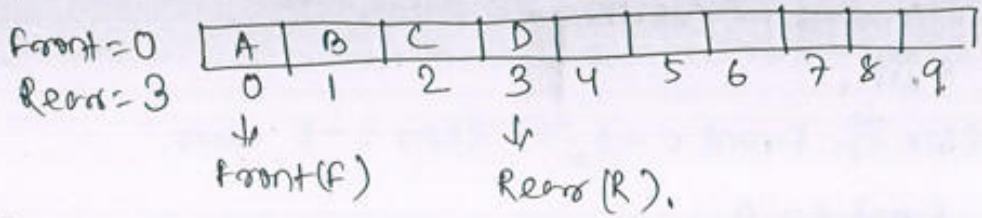
   2. Deletion of an element can be done from the room that is identified by FRONT and FRONT will identify the next room of the queue. Underflow situation occurs when no element is present in the queue. ie:- when front and rear identifies NULL and deletion operation is performed on it.

Eg:- Let Q[MAXSIZE] is an array that represents a queue in the memory and at the beginning, it is empty, and MAXSIZE = 10, ie:- Front = Rear = NULL or -1,

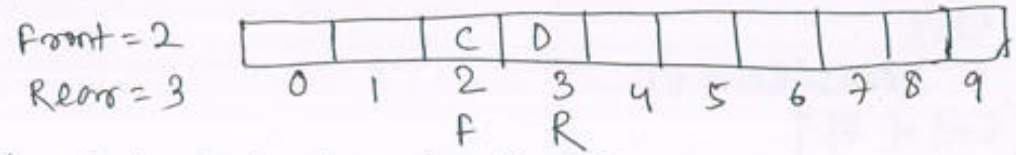


G F T CENTRAL LIBRARY RHUBANESWAR

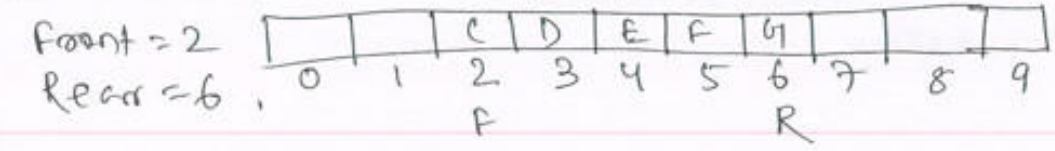
i) Insert 4 elements A, B, C, D.



ii) Delete 2 elements



iii) Insert 3 elements E, F, G.



Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

From the example, it is clear that-

i) For each insertion in the queue the value of REAR is incremented by 1 ie:- REAR = REAR + 1.

ii) For each deletion in the queue the value of front is incremented by FRONT = FRONT + 1.

Note:

1. After all the elements inserted into the Queue, Rear will identify position at MAXSIZE - 1. ie:- Queue is full.
2. Even though the memory locations are vacant at FRONT end, where Rear identified MAXSIZE-1 position, we can't insert a new element.
3. When Front = Rear = Null (ie:- queue is empty), we can't perform deletion operation.
4. When Front = Rear only one element is present in Queue.
5. When front is at MAXSIZE - 1, means Queue having one element. After deletion Front need to be reset to NULL.

Algorithm for insertion into a linear Queue:

QINSERT (QUEUE[MAXSIZE], ITEM)

Step 1: If Rear = MAXSIZE-1, then
Display "Overflow or Queue full"
Exit.

Step 2: Else if Front = -1, or Rear = -1, then
Front := 0
Rear := 0

Step 3: Else
Rear := Rear + 1
[End of if]

Step 4: QUEUE[Rear] := ITEM

Step 5: Exit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

C-function:

```
void insertion(int Q[], int item)
{
  if(rear == maxsize-1)
  {
    printf("\n queue overflow");
    exit(0);
  }
  else if(front == -1 || rear == -1)
  {
    front = 0;
    rear = 0;
  }
  else
  {
    rear = rear + 1;
  }
  Q[rear] = item;
}
```

G I P T CENTRAL LIBRARY BHUBANESWAR

Algorithm for deletion in a linear queue:

Assume QUEUE[MAXSIZE] is an array which represents a queue, FRONT and REAR identifies the index of front and rear end respectively. ITEM holds the element to be deleted from Queue.

QDELETE (QUEUE[MAXSIZE])

Step 1: Set ITEM

Step 2: If front = -1 or Rear = -1, then
Display "Underflow" or Queue empty"
Exit
[End of If]

Step 3: ITEM := QUEUE[front]

Step 4: Display "Deleted Item is", ITEM

Step 5: Else if front = Rear, then [when queue has only one element]
front := -1
Rear := -1

step 6: Else
front := front + 1.

step 7: Exit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

C-function:

```
void deletion (int a[])
 { int item;
   if (front == -1 || rear == -1)
    { printf ("\n Queue underflow")
      exit(0);
    }
   item = a[front];
   if (front == rear)
     { front = -1;
       rear = -1;
     }
   else.
     { front = front + 1;
     }
   printf ("\n deleted item is %d", item);
}
```

C-Program for implementing Linear queue:

```
#include<stdio.h>
#include<process.h>
#define maxsize 10
int a[maxsize], front = -1, rear = -1, item;
void insertion(int [], int);
void deletion (int[]);
void display (int []);
main()
 {
   int option;
    do
     {
       printf("\n 1. insert\n 2. delete\n 3. display\n 4. exit\n\n
                enter option");
        scanf("%d", &option);
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
switch(option)
{
 case 1: printf("\n enter item");
         scanf("%d", &item);
         insertion(Q, item);
          break;
 case 2: deletion(Q);
          break;
 case 3: display(Q);
          break;
 case 4: exit(0);
          break;
 default: printf("\n Invalid entry");
}
}while(option!=4);
}

void insertion(int Q[], int item)
{ if(rear == maxsize-1)
  { printf("\n queue overflow");
    exit(0);
  }
 else if(front == -1 || rear == -1)
  {
     front = 0;
     rear = 0;
  }
 else
    { rear = rear + 1;
    }
   Q[rear] = item;
}
```

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
void deletion(int Q[])
{ int item;
  if(front == -1 || rear == -1)
  { printf("\n Queue Underflow");
    exit(0);
  }
  item = Q[front];
  if(front == rear)
  { front = -1;
    rear = -1;
  }
  else { front = front + 1;
  }
  printf("\n deleted item = %d", item);
}
void display(int Q[])
{ int i;
  if(front != -1)
  { printf("\n items at queue\n");
    for(i = front; i <= rear; i++)
      printf("\t%d", Q[i]);
  }
}.
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

Circular Queue:

1. It is a queue of circular nature, in which it is possible to insert a new element at the first location, when the last location of the queue is occupied.

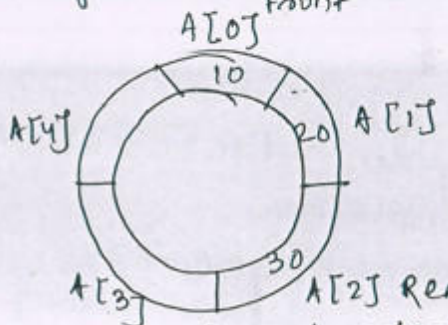
   In other words, if an A[MAXSIZE] represents a circular queue, then after inserting an element at (MAXSIZE - 1)th location of the array, the next element will be inserted at the very first location at index 0.

Advantage of circular queue over linear queue:

→ In case of linear queue, if Rear is at MAXSIZE - 1, then queue is considered as full. Here even though memory locations are vacant at beginning still we can not use them, as for each insertion rear is incremented and here REAR goes out of bounds. This drawback can be overcome by circular queue representation.

Circular Queue Representation:

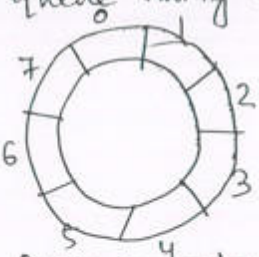
Eg:- Let an array A[5] is maintained in form of circular queue.



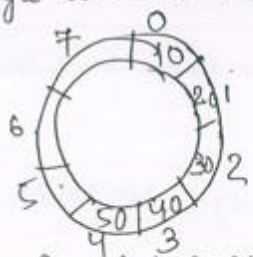
G.I.F.T. CENTRAL LIBRARY BHUBANESWAR

Here overflow situation arises only when Rear is before the front position.

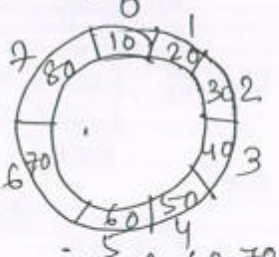
Eg:- When Queue is empty, the front = NULL or -1 and when Queue is full, the front = 0 and Rear = MAXSIZE - 1. When the queue having a single element the front = Rear.



Queue is empty
Front = Rear = -1.

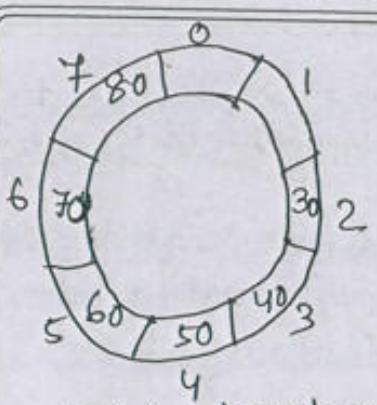


insert 10, 20, 30, 40, 50
front = 0
Rear = 4

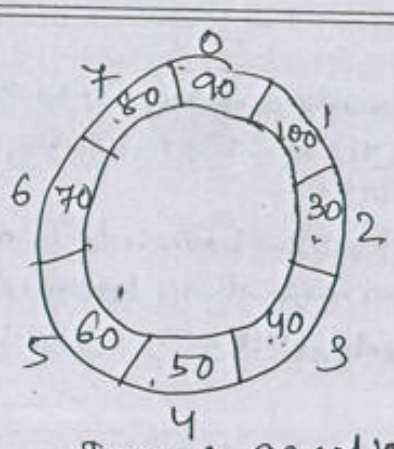


insert 60, 70, 80
front = 0, rear = 7.
Queue is full.

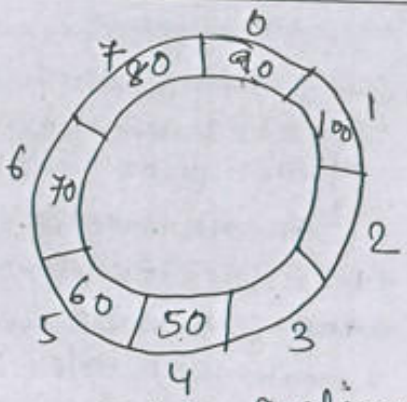
Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout



Delete two elements
Front = 2
Rear = 7



Insert 90 and 100
front = 2
Rear = 1.



Delete 2 elements
front = 4
rear = 1.

iii) If Rear = MAXSIZE - 1, and free locations exist, then
Rear = 0.

iii) If front = MAXSIZE - 1 and Rear is less than N, then
Front = 0.

During each insertion, to find the next Rear the formula is :-

Rear = (Rear + 1) % MAXSIZE

Similarly during each deletion to find the next front position the formula is :-

Front = (front + 1) % MAXSIZE.

Algorithm for Insertion in a circular queue :

CQINSERT(QUEUE[MAXSIZE], ITEM)
Step 1: If (front = 0 and Rear = MAXSIZE - 1) or
(front = (Rear + 1) % MAXSIZE), then
Display "Overflow"
Exit.
Step 2: Else If front = -1 or Rear = -1 then
front := 0
Rear := 0.
Step 3: Else
Rear := (Rear + 1) % MAXSIZE
[End of If]
Step 4: QUEUE[Rear] := ITEM
Step 5: Exit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

C - function:

```
void insertion(int Q[], int item)
{
  if((front == 0 && rear == maxsize-1) || (front == (rear+1)%.maxsize))
  {
    printf("\n Queue overflow");
    return;
  }
  else if(front == -1 || rear == -1)
  {
    front = 0;
    rear = 0;
  }
  else
  {
    rear = (rear+1)%. maxsize;
  }
  Q[rear] = item;
}
```

Algorithm for deletion in a circular queue:

CQDELETION(QUEUE, FRONT, REAR)

Step 1: Let ITEM

Step 2: If Front = -1 or Rear = -1, then
Display "Queue is empty or Underflow"
Exit
[End of if]

Step 3: ITEM := QUEUE[Front]

Step 4: Display "Deleted Item is", ITEM.

Step 5: If Front := Rear, then
Front := -1
Rear := -1

Step 6: Else
Front := (Front + 1)% MAXSIZE
[End of If]

Step 7: Exit.

G.I.F.T. CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

c-function:

```
Void deletion(int Q[])
{ int item;
  if(front == -1 || rear == -1)
  { printf("\n Queue Underflow");
    return;
  }
  item = Q[front];
  if(front == rear)
  { front = -1;
    rear = -1;
  }
  else
  { front = (front + 1) % maxsize;
  }
  printf("\n deleted item is %d", item);
}
```

'C' Program: implementing circular queue and its operation:

```
#include<stdio.h>
#define maxsize 5
int Q[maxsize], front = -1, rear = -1, item;
void insertion(int [], int);
void deletion(int []);
void display(int []);
main()
{ int option;
  do
  { printf("\n 1. insert\n 2. delete\n 3. display\n 4. exit\n\n enter option");
    scanf("%d", &option);
    switch(option)
    { case 1: printf("\n enter item");
              scanf("%d", &item);
              insertion(Q, item);
              break;
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
        case 2: deletion(a);
                break;
        case 3: display(a);
                break;
        case 4: exit(0);
                break
        default: printf("\n Invalid option");
    }
  }
  while(option!=4);
}
void insertion(int a[], int item)
{
  if((front==0 && rear==maxsize-1) || (front==(rear+1)%maxsize))
    {
      printf("\n queue overflow");
      return;
    }
  else if(front==-1 || rear==-1)
    {
      front=0;
      rear=0;
    }
  else
    {
      rear=(rear+1)%maxsize;
    }
  a[rear]=item;
}
void deletion(int a[])
  { int item;
    if(front==-1 || rear==-1)
      { printf("\n Queue Underflow");
        return;
      }
    item=a[front];
    if(front==rear)
      { front=-1;
        rear=-1;
      }
```

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

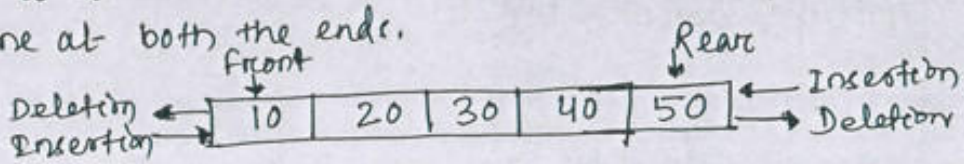
Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE


```
else
 {
   front = (front +1) % maxsize;
 }
 printf("\n deleted item = %d", item);
}
void display (int Q[])
{
  int i;
  if (front != -1)
   {
     if (front <= rear)
     {
       printf("\n items at queue\n");
       for( i = front; i<=rear; i++)
            printf("\t%d", Q[i]);
     }
     else
      {
         printf("\n items at queue\n\n");
         for (i = front; i<maxsize; i++)
              printf("\t%d", Q[i]);
          for (i=0; i<=rear; i++)
              printf("\t%d", Q[i]);
      }
   }
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Double Ended Queue (DEQUE)

1. It is a kind of queue, in which the elements can be added or removed at either ends but not in the middle. In other words, it is a linear list in which insertion and deletion can be done at both the ends.



2. There are 2 forms of Deques such as :-
   a) Input Restricted Deque and b) Output Restricted Deque

a) Input Restricted Deque
→ It is a deque, which restricts insertion at only one end, but allows deletion at both ends of the list. So in input restricted deque, the following operations are possible :-
   i) Deletion at front end
   ii) Deletion at Rear end
   iii) Insertion at front or Rear end.

G I P T CENTRAL LIBRARY BHUBANESWAR

b) Output Restricted Deque
→ It is a deque, which restricts deletion at only end but allows insertion at both ends of the list. So, in output restricted deque, the following operations are possible :-
   i) Insertion at front end
   ii) Insertion at Rear end
   iii) Deletion at front or Rear end.

3. Insertion and deletion is allowed at both end of Deque, so the operations possible in deque are :-
   i) Insertion at Rear end (similar to linear queue)
   ii) Deletion at front end (similar to linear queue)
   iii) Insertion at front end
   iv) Deletion at Rear end.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

'C' - function for insertion at rear end of deque:

```
void dqinsert_rear(int Q[], int ITEM)
  { if (rear == Maxsize-1)
     { printf("Deque is full");
     }
    else
     { Rear = Rear + 1;
       Q[Rear] = ITEM;
     }
  }
```

'C' - function for insertion at front end of deque:

```
void dqinsert_front(int Q[], int Front, int Rear, int ITEM, int maxsize)
 { if(Front == 0)
    { printf("Deque is full");
    }
   else
    { front = Front - 1;
      Q[FRONT] = ITEM;
    }
 }
```

'C' - function for deletion at front end of deque:

```
void dqdelete_Front(int Q[])
 { int ITEM;
   if(Front == -1)
    { printf("Deque is empty");
    }
   else if (front == Rear)
     { ITEM = Q[front];
       Front = Rear = -1;
       printf("Deleted item is %d", ITEM);
     }
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
else
{
    ITEM = Q[front];
    front = front+1;
    printf("Deletem Item is %d", ITEM);
}
}
```

C- function for deletion at rear end of deque :

```
void dqdelete_rear(int Q[])
{
int ITEM;
if(front == -1 || Rear == -1)
{
    printf("Deque is empty");
    exit(0);
}
ITEM = Q[Rear];
if(front == Rear)
{
    front = Rear = -1;
}
else
{
    Rear = Rear - 1;
}
printf("Deleted Item is %d", ITEM);
}
```

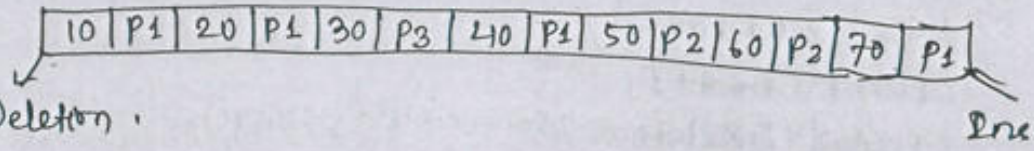
GIFT CENTRAL LIBRARY BHUBANESWAR

Priority Queue :

1. A priority Queue is a collection of elements such that each element has a priority level and according to the priority level, the elements are deleted. In the queue, an element of higher priority is processed before any element of lower priority. When two or more elements have same priority levels then they are processed according to the order in which they

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

are added to the queue.



2. Priority queues are classified into two categories, they are :-

   i) Ascending priority queue.
   ii) Descending priority queue.

Ascending priority queue : It is a collection of elements in which the items or elements are inserted randomly and from which the smallest element can be deleted.

Descending Priority queue : It is a collection of elements in which the items or elements are inserted randomly and from which the largest element can be deleted.

—X—

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Data Structure Using C

Topic:

Linked List

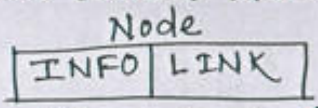
Contributed By:

Nihar Ranjan Rout

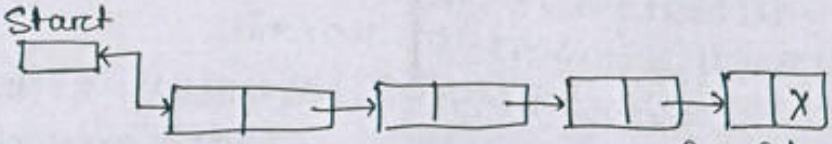
Gandhi Institute For Technology, GIFT

LINKED LIST:

1. Linked List is a linear collection of data elements called nodes, each of which is a memory location that contains two parts. That is :-
   i) Information of element or data.
   ii) Link field contains the address of next node or points to the next element stored in the list.



[Structure of a node in linked list]



2. Here start is an external pointer which identified address of 1st Node. i.e. the beginning of the linked list. The address of Link part of last node contains NULL as no further nodes exist.
3. The INFO part contains the value and the link points to the next node. To create a node of linked list, self referenced structure is needed.

CENTRAL LIBRARY RHUBANESWAR

Dynamic Storage Management:

i) Static Memory Allocation:

Static memory allocation is the one in which the required memory is allocated at compile time.
Eg:- arrays are the best example of static memory allocation.

Drawbacks of static memory Allocation:

Static memory allocation has the following drawbacks.
i) Size is fixed before execution.
ii) Insertion and deletion is a time taking process as the elements need to be shifted towards left or right.
iii) There may be a possibility that memory may get wasted or memory is deficit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

ii) Dynamic Memory Allocation :-

Dynamic memory allocation in which the required memory is allocated at runtime. Eg:- Linked lists are the best example of dynamic memory allocation. The drawbacks of static memory allocation can be overcome by dynamic memory allocation.

Advantages of Linked List :

1. A Linked List has the following advantages over arrays.
   i) The allocated size can be increased or decreased as per the requirement at run time.
   ii) Memory can be utilized efficiently as the programmer can choose exact amount of memory needed.
   iii) The operations like insertion and deletion are less time consuming as compared to arrays.

Disadvantages of Linked List :

1. A linked list has few disadvantages, they are :-
   i) It takes more memory space because each node contains a link or address part.
   ii) Moving to a specific node directly like array is not possible.

How to declare structure of a node in 'c'

```
struct node
  { int info;
    struct node *link; // self referencing structure.
  };
struct node *p;
p = (struct node*) malloc (sizeof (struct node));
```

Note :- If a pointer to a structure is declared as a structure element of same structure, called self referencing structure.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Basic Operations on Linked List :

→ The basic operations that can be performed on a linked list are as follows :-

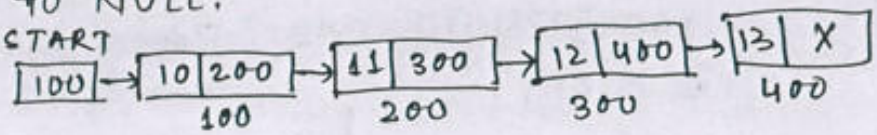
i) Creation
ii) Insertion
iii) Deletion
iv) Traversing
v) Searching
vi) Sorting
vii) Concatenation

G.I.F.T. CENTRAL LIBRARY BHUBANESWAR

Types of Linked List :

1. Singly Linked List :

1. It is a collection of nodes where each node contains the address of next node. It is also referred as one-way list as traversing is possible in only one direction ie :- from START to NULL.



Memory Allocation :

1. Assume a set of free memory blocks in the form of nodes exist in the memory of the computer.
2. Assume AVAIL is a pointer which holds the address first free node in the memory.
3. Assume each node is divided into two parts INFO and LINK parts.

Suppose START, PTR and FRESH are the pointers where,

i) START identifies the address of first node of linked list.
ii) FRESH identifies the new node to be added to a linked list.
iii) PTR will be used to move between linked lists,

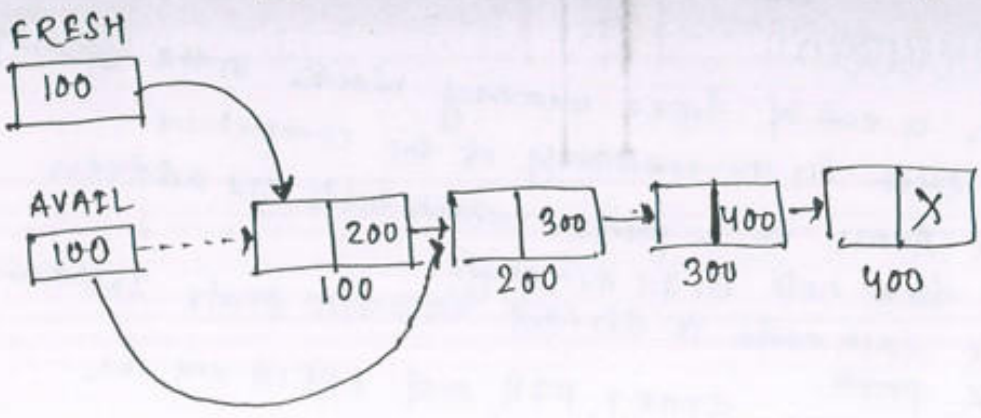
Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

→ When AVAIL = NULL, it indicates that no free memory blocks exist in the memory. At this point of time, if we try to create a new node overflow situation occurs.

→ When START = NULL, it indicates that linked list has no nodes or linked list doesn't exist. At this point of time if we try to delete a node from linked list it leads to underflow situation.

4. The following pseudocode checks for availability of memory and a new node is ~~pre~~ prepared if free memory is available otherwise displays an overflow message.

```
Step 1: If AVAIL = NULL, Then
            Display "Insufficient Memory or Overflow"
        Exit.
Step 2: Else
        FRESH := AVAIL          [FRESH is assigned with node as pointed by AVAIL]
        AVAIL := LINK[AVAIL]    [AVAIL now holds address of next available free node]
        INFO[FRESH] := ITEM     [INFO part of FRESH is updated with ITEM]
        LINK[FRESH] := NULL     [Address or LINK part is assigned with NULL]
[End of IF]
```



Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

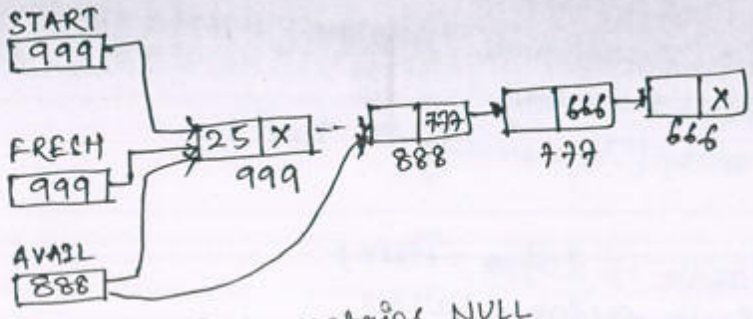
Operations on a single linked list :

1. Insertion of a node in a single linked list.
   a) Insertion at the beginning
   b) Insertion at the end
   c) Insertion at a specific location
   d) Insertion after a specific location
   e) Insertion after specified node.
2. Deletion of a node in a single linked list.
   a) Deletion at the beginning
   b) Deletion at the end
   c) Deletion of a node at specific location.
   d) Deletion of a node based on a given item or INFO
3. Searching for a node based on given item
4. Concatenation of two linked lists.
5. Sorting the node INFO in a single linked list.
6. Reversing the single linked list.

GIFT CENTRAL LIBRARY BHUBANESWAR

1. Insertion of a node in a single linked list :

a) Insertion at the beginning (when list is empty):



START initially contains NULL

After creation operation :-

i) START points to node with address 999

ii) AVAIL points to the next free node with address 888

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm:

CREATION (AVAIL, INFO, LINK, START, FRESH, ITEM)

Step 1: If AVAIL = NULL, Then
Display "Insufficient Memory or Overflow"
Exit.

Step 2: Else
FRESH := AVAIL
AVAIL := LINK[AVAIL]
INFO[FRESH] := ITEM
LINK[FRESH] := NULL
START := FRESH
[End of IF]

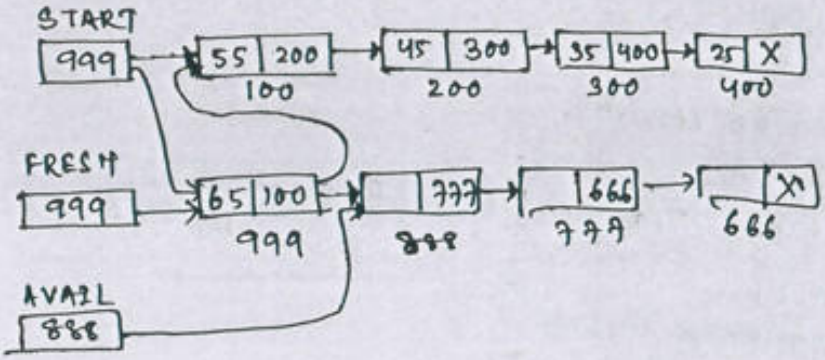
Step 3: Exit

C-function:

```
struct node *creati-list (struct node *start)
{ struct node *fresh;
  int item;
  printf("\n enter item:");
  scanf("%d", &item);
  fresh = (struct node *) malloc (sizeof (struct node));
  if (fresh == NULL)
     printf("\n memory is full");
  else
  { fresh -> info = item;
    fresh -> link = NULL;
    start = fresh;
  }
  return start;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

b) Insertion at the beginning (when linked list has one or more nodes):



START initially contains the 1st node address ie :- 100,
After insertion operation :-

i) START points to node with address 999
ii) Node with address 999 point to the address 100
iii) AVAIL points to the next free node with address 888.

Algorithm :

INSERTFIRST(INFO, LINK, START, AVAIL, ITEM)

Step 1: If AVAIL = NULL, then
Display "Memory insufficient or overflow"
Exit.
[End of If]

Step 2: If START = NULL, then
START := FRESH

Step 3: Else
FRESH := AVAIL
AVAIL := LINK[AVAIL]
INFO[FRESH] := ITEM
LINK[FRESH] := START.
START := FRESH
[End of IF]

Step 3: Exit.

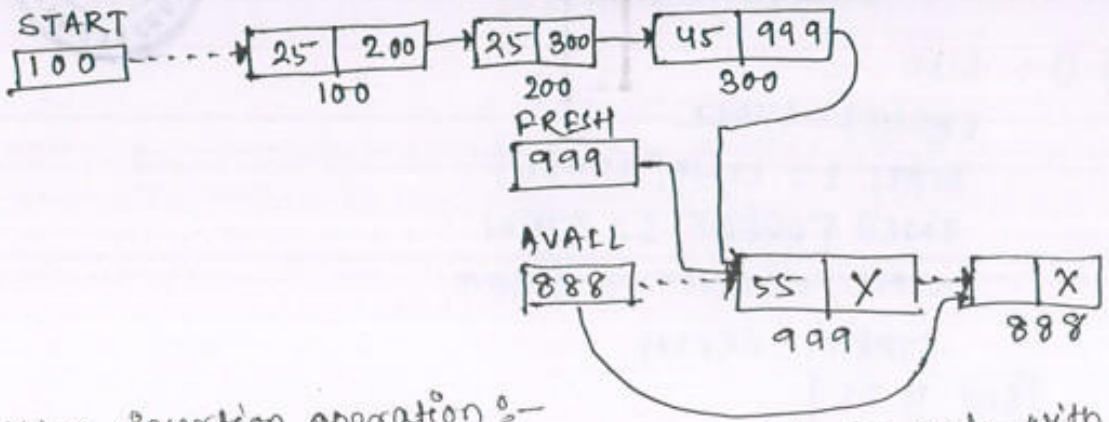
GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

C-function :

```
struct node * insert_first(struct node *start)
{
  struct node *fresh;
  int item;
  printf("\n enter an item");
  scanf("%d", &item);
  fresh = (struct node *) malloc (sizeof (struct node));
  if (fresh == NULL)
      printf("\n memory is full");
  else
    { fresh -> info = item;
      fresh -> link = NULL;
      if (start == NULL)
       { start = fresh;
       }
      else
        { fresh -> link = start;
          start = fresh;
        }
    }
    return start;
}
```

c) Insertion at the end of a single Linked list :



After insertion operation :-
The last node with address 300 points to node with address 999 and link part of Node with address 999 sets to NULL and AVAIL points to the next free node with address 888.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm:

INSERTEND (INFO, LINK, START, AVAIL, ITEM)

Step 1: Set PTR

Step 2: If AVAIL = NULL, then Display "Memory Insufficient or overflow" Exit.

Step 3: Else

FRESH := AVAIL

AVAIL := LINK[AVAIL]

INFO[FRESH] := ITEM

LINK[FRESH] := NULL

Step 3.1: If START = NULL, Then

START := FRESH

Step 3.2: else

PTR := START

Repeat while LINK[PTR] != NULL

PTR := LINK[PTR]

[End of while]

LINK[PTR] := FRESH

[End of step 3.1]

[End of step-2]

Step 4: Exit.

C-function:

```
struct node *insert_last(struct node *start)
{ struct node *fresh, *ptr;
  int item;
  printf("\n enter item");
  scanf("%d", &item);
  fresh = (struct node*) malloc (sizeof (struct node));
  if(fresh == NULL)
      printf("\n memory is full");
  else
  { fresh -> info = item;
    fresh -> link = NULL;
    if (start == NULL)
    { start = fresh;
    }
    else
    { for (ptr = start; ptr -> link != NULL; ptr = ptr -> link);
      ptr -> link = fresh;
    }
  } return start;
```

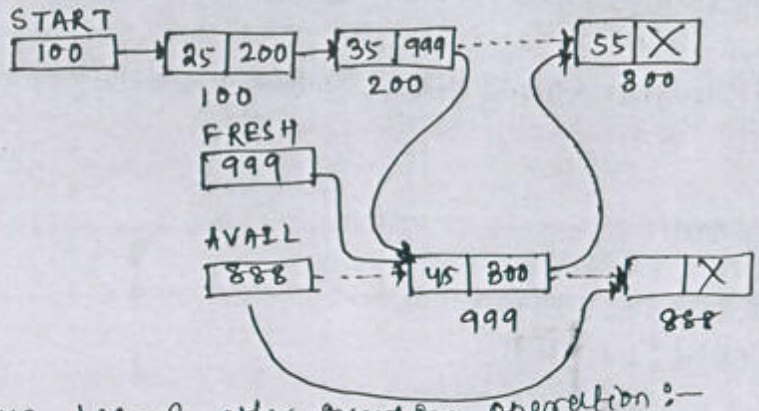
GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

4) Insertion at specific location of a Single Linked list :



When LOC = 3, after insertion operation :-

ie the 2nd node with address 200 points to new node with address 999 and the link part of node with address 999 points to node with address 300. So, now the new node is inserted at location 3 and Avail points to the next free node having address 888.

Algorithm :

INSERTLOC (INFO, LINK, START, AVAIL, LOC, ITEM, PTR, PREV)

Step 1 : Let I := 1

Step 2 : If AVAIL = NULL, then Display "memory insufficient"
Exit.

Step 3 : Else
FRESH := AVAIL
AVAIL := LINK[AVAIL]
INFO[FRESH] := ITEM
LINK[FRESH] := NULL

Step 3.1 : If START = NULL, Then Display "List doesn't exist"

Step 3.2 : Else
Step 3.2.1 : PTR := START
Step 3.2.2 : Repeat, while I < LOC AND PTR != NULL
PREV := PTR
PTR := LINK[PTR]
I := I + 1
[End of while]
Step 3.2.3 : If PTR = NULL, then Display "Location not found"
Step 3.2.4 : Else if PTR = START, then
LINK[FRESH] := START
START := FRESH.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Step 3.2.5 : Else

LINK[PREV] := FRESH

LINK[FRESH] := PTR

[End of IF - Step 3.2.3]

[End of Step-3.1]

[End of step-2]

Step 4 : Exit.

C-function:

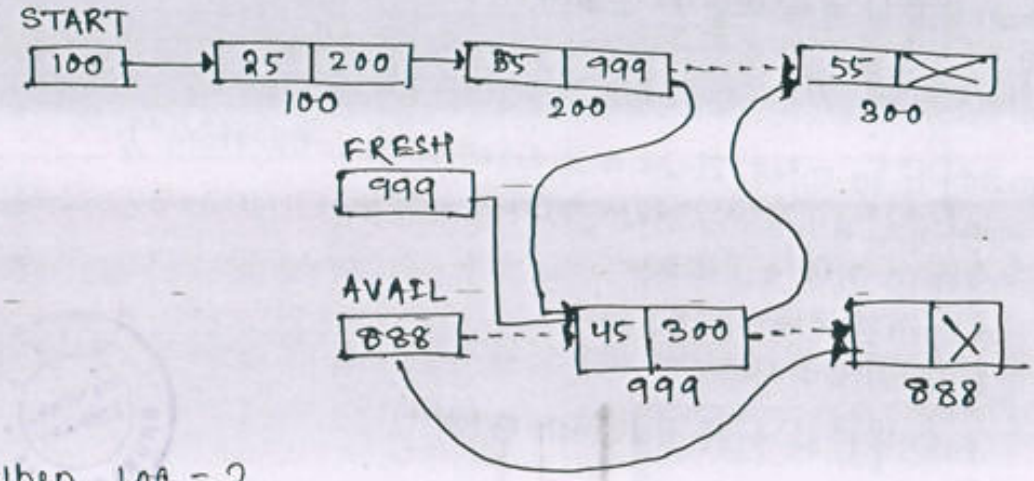
```
struct node *insert_loc(struct node *start)
{ struct node *fresh, *ptr, *prev;
  int item, loc, i;
  fresh = (struct node *) malloc(sizeof(struct node));
  if(fresh == NULL)
  { printf("\n memory is full");
  }
  else
  { printf("\n enter item and location for insertion");
    scanf("%d%d", &item, &loc);
    fresh->info = item;
    fresh->link = NULL;
    if(start == NULL)
    { printf("\n list does not exist");
    }
    else
    { ptr = start;
      i=1;
      while(i<loc && ptr != NULL)
      { prev = ptr;
        ptr = ptr->link;
        i++;
      }
      if(ptr == NULL)
      { printf("\n Invalid entry of location");
      }
```

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
else if (ptr == start)
{
    // insertion at beginning
    fresh -> link = ptr;
    start = fresh;
}
else
{
    prev -> link = fresh;
    fresh -> link = ptr;
}
}
return start;
}
```

e) Insertion after specific location in a single linked list :



When LOC = 2,

After insertion operation :-

The 2nd node with address 200 points to new node with address 999 and the link part of node with address 999 points to node with address 300. So, now the new node inserted after location 2 and AVAIL points to the next free node with address 888.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm:

INSERTAFTERLOC(INFO, LINK, START, AVAIL, LOC, ITEM, PTR)

Step 1: Let I.
Step 2: If AVAIL = NULL, then Display "Memory Insufficient"
Exit.
Step 3: Else
FRESH := AVAIL
AVAIL := LINK[AVAIL]
INFO[FRESH] := ITEM
LINK[FRESH] := NULL
Step 3.1: If START = NULL, then Display "list does not exist".
Step 3.2: Else
Step 3.2.1: PTR := START
Step 3.2.2: Repeat for I := 1 to LOC - 1 increasing by 1.
PTR := LINK[PTR]
[End of FOR]
Step 3.2.3: If PTR = NULL, Then
Display "Invalid entry for location"
Step 3.2.4: Else
LINK[FRESH] := LINK[PTR]
LINK[PTR] := FRESH
[End of IF - Step 3.2.3]
[End of IF 3.1 step]
[End of IF - step 2]
Step 4: Exit.

C-function:

```
struct node *insert_after_loc(struct node *start)
{
  struct node *fresh, *ptr;
  int item, loc, i;
  fresh = (struct node *) malloc(sizeof(struct node));
  if (fresh == NULL)
  {
    printf("\n memory is full");
  }
```

GIFT CENTRAL LIBRARY BHUBANESWAR

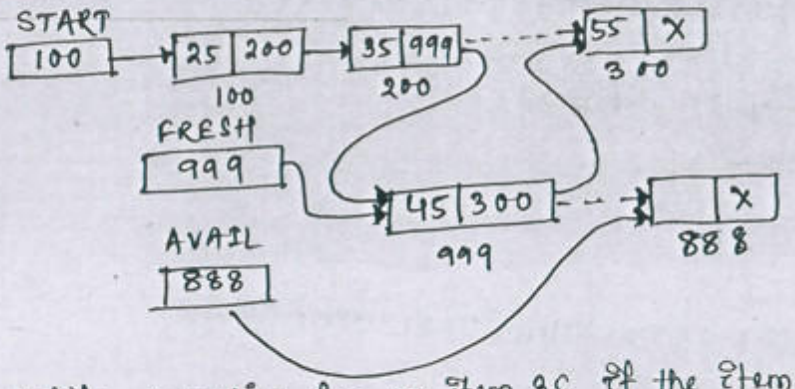
Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
else
{
  printf("\n enter item");
  scanf("%d", &item);
  printf("\n enter location");
  scanf("%d", &loc);
  fresh -> info = item;
  fresh -> link = NULL;
  if (start == NULL)
    printf("list doesn't exist");
  else
  {
    ptr = start;
    i = 1;
    while (i < loc && ptr != NULL)
    {
      ptr = ptr -> link;
      i++;
    }
    if (ptr == NULL)
    {
      printf("\n Invalid entry of location");
    }
    else
    {
      fresh -> link = ptr -> link;
      ptr -> link = fresh;
    }
  }
}
return start;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

f) Insertion of a node After specified node in single linked list :



While searching for an item 35, if the item is available then insertion after that node is accomplished as follows :-
Here, the node with address 200 contains item ie:- 35, so node with address 200 points to new node with address 999 and the link part of node with address 999 points to the node with address 300. So, now the new node is inserted after the node having info part with value 35, and AVAIL points to the next free node with address 888.

Algorithm :

INSERTAFTERNODE (INFO, LINK, START, ITEM, PTR, AVAIL, NEWITEM, FRESH)

Step 1 : START
Step 2 : If AVAIL = NULL, then Display "memory insufficient or Overflow"
Exit
Step 3 : Else
FRESH := AVAIL
AVAIL := LINK[AVAIL]
INFO[FRESH] := NEWITEM
LINK[FRESH] := NULL
Step 3.1 : If START = NULL, then Display "list doesn't exist"
Step 3.2 : Else
Step 3.2.1 : PTR := START
Step 3.2.2 : Repeat while PTR != NULL AND ITEM != INFO[PTR]
PTR := LINK[PTR]
[End of while]
Step 3.2.3 : If PTR = NULL then
Display "Item doesn't exist".

G P T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Step 3.2.4: Else

LINK[FRESH] := LINK[PTR]

LINK[PTR] := FRESH.

[End of IF - step 3.1]

[End of IF - step 2]

Step 4: Exit.

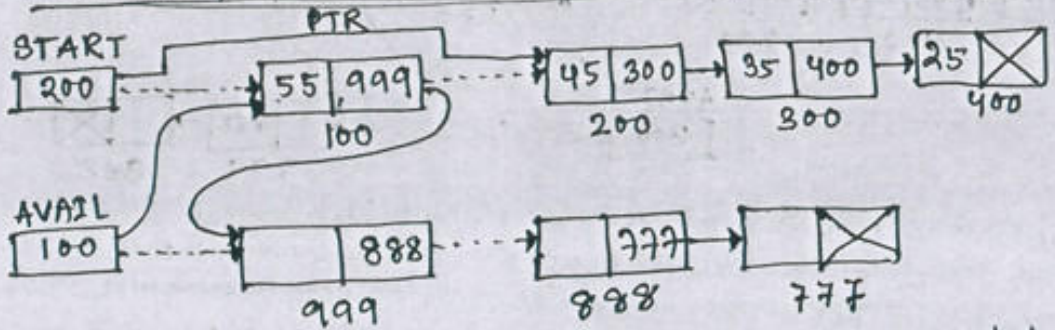
C - Program:

```
struct node * insert_after_item(struct node *start)
{ struct node *fresh, *ptr;
  int item, newitem;
  fresh = (struct node *) malloc(sizeof(struct node));
  if (fresh == NULL)
      printf("\n memory is full");
  else
  {
    printf("\n enter item to search and new item to insert");
    scanf("%d %d", &item, &newitem);
    fresh -> info = newitem;
    fresh -> link = NULL;
    if (start == NULL)
    { printf("\n list doesn't exist");
    }
    else
    {
        ptr = start;
        while (item != ptr -> info && ptr != NULL)
        { ptr = ptr -> link;
        }
        if (ptr == NULL)
          { printf("\n item to search not exist");
          }
        else { fresh -> link = ptr -> link;
               ptr -> link = fresh;
        }
  } return start;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

2. Deletion from a Linked List :

a) Deletion of first node from a single linked list :



Here the first node i.e. node with address 100 to be deleted. So, START will be updated with address of 2nd node which is present in the link part of first node as pointed by START. PTR is a pointer that points to first node with address 100 is added to the free or avail list at the beginning by updating link part of PTR with address of node as pointed by AVAIL and AVAIL is assigned with the address 100 as pointed by PTR.

Algorithm :

DELETION FIRST(INFO, LINK, START, AVAIL, PTR)

Step 1: If START = NULL, then
Display " List doesnot exist or Underflow"

Step 2: Else
PTR := START
START := LINK[PTR]
LINK[PTR] := AVAIL
AVAIL := PTR
[End of IF]

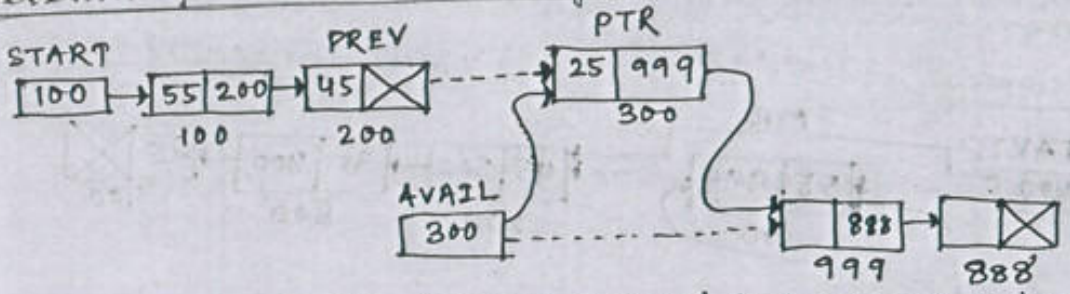
Step 3: Exit.

C - function :

```
struct node *delete_first(struct node *start)
{ struct node *ptr;
  if(start == NULL)
  { printf("\n memory underflow - list is empty");
  }
  else
  { ptr = start;
    start = start->link;
    free(ptr);
  } return start;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

b) Deletion of last node from single linked list :



Here, the last node is pointed by PTR and last but one node is pointed by PREV. As the node pointed by PREV will become the last node, the link part of PREV with address 200 is updated with NULL. The node with address 300 as pointed by PTR is added at the beginning of free or avail list.

Algorithm :

DELETIONEND (START, LINK, INFO, PTR, PREV, AVAIL)

Step 1: If START = NULL, Then Display "List is empty",

Step 2: Else

Step 2.1 : PTR := START.

Step 2.2 : Repeat While LINK[PTR] != NULL

PREV := PTR

PTR := LINK[PTR]

[End of While]

Step 2.3 : LINK[PREV] := NULL

Step 2.4 : LINK[PTR] := AVAIL

Step 2.5 : AVAIL := PTR

[End of If]

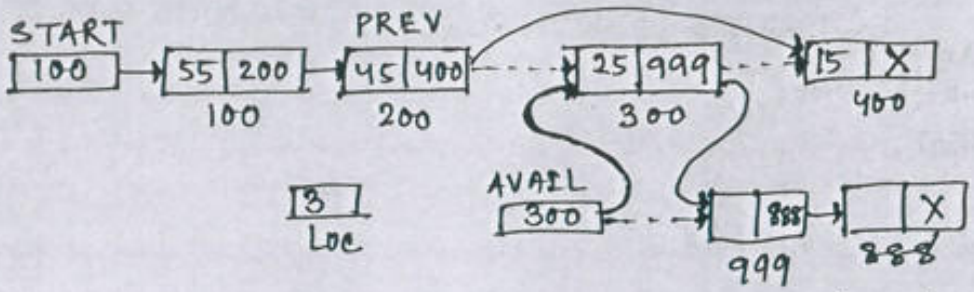
Step 3 : Exit.

C - function :

```
struct node *delete_last(struct node *start)
{ struct node *ptr, *prev;
  if(start == NULL)
  { printf("\n memory underflow — list is empty");
  }
  else {
       ptr = start;
       while(ptr -> link != NULL)
       { prev = ptr;
         ptr = ptr -> link;
       }
       prev -> link = NULL;
       free(ptr);
  } return start;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

c) Deletion of node from a specific location:



Here loc is 3. i.e. node with address 300 is to be deleted. The link part of node with address 200 is updated with the link part of node with address 300 i.e. 400. Now the node pointed by PTR is added at the beginning of AVAIL list.

Algorithm:

```
DELETION LOC (INFO, LINK, START, PTR, PREV, LOC)
 Step 1: let I := 1
 Step 2: PTR := START
 Step 3: Repeat, while I < LOC AND PTR != NULL
            PREV := PTR
            PTR := LINK[PTR]
            I := I+1
         [End of While]
 Step 4: If PTR = NULL, then Display "Location Doesn't exist".
 Step 5: Else if PTR = START, then START := LINK[PTR]
 Step 6: Else LINK[PREV] := LINK[PTR]
         [END of IF]
 Step 7: LINK[PTR] := AVAIL
 step 8: AVAIL := PTR
 Step 9: Exit.
```

GIFT CENTRAL LIBRARY BERHAMPUR

c-function:

```
struct node *delete_loc (struct node *start)
{ struct node *ptr, *prev;
  int loc, i;
  if (start == NULL)
  { printf("\n memory underflow - list is empty");
  }
```

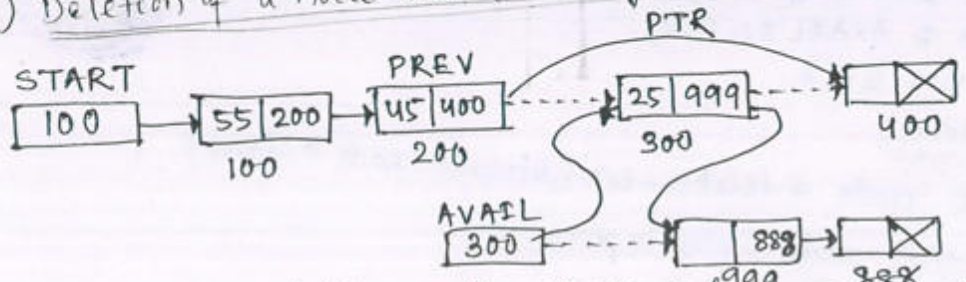

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE


```
else
{
  printf("\n enter the location or node no. for deletion");
   scanf("%d", &loc);
   ptr = start;
   i=1;
  while(i<loc && ptr!=NULL)
   {prev = ptr;
     ptr = ptr -> link;
     i++;
   }
  if(ptr==NULL)
        printf("\n location entered not exist");
   else if(ptr == start)
     {
       start = start -> link;
        free(ptr);
     }
   else
     {
       prev -> link = ptr -> link;
       free(ptr);
     }
  }
  return start;
}
```

d) Deletion of a node based on a given item:



Here we want to delete a node whose info part contains the value 25. PTR is pointing to the node with address 300 whose info part contains the item given i.e. 25 and PREV is pointing to node with address 200. The link part of node as pointed by PREV is updated with link part of node as pointed by PTR. Now, the node with address 300 as pointed by PTR is added at the beginning of free or avail list.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm :

DELETION ITEM (LINK, INFO, START, PTR, PREV, AVAIL, ITEM)

Step 1 : PTR := START

Step 2 : Repeat while PTR != NULL AND ITEM != INFO[PTR]

PREV := PTR

PTR := LINK[PTR]

[End of while]

Step 3 : If PTR = NULL, then Display "Item Not found"

Step 4 : Else If PTR = START, then START := LINK[PTR]

Step 5 : Else LINK[PREV] := LINK[PTR]

[End of If]

Step 6 : LINK[PTR] := AVAIL

Step 7 : AVAIL := PTR

Step 8 : Exit.

C-function :

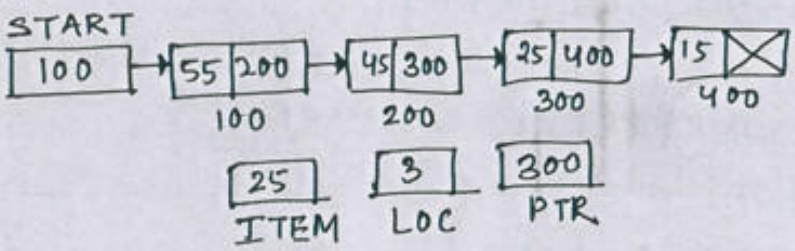
```
struct node *delete_item(struct node *start)
{ struct node *ptr, *prev;
  int item;
  if (start == NULL)
  { printf("\n memory underflow - list is empty");
  }
  else
  { printf("\n enter the item for deletion");
    scanf("%d", &item);
    ptr = start;
    while (item != ptr->info && ptr != NULL)
    { prev = ptr;
      ptr = ptr->link;
    }
    if (ptr == NULL)
      printf("\n location entered not exist");
    else if (ptr == start)
    { start = start->link;
      free(ptr);
    }
```

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
    else
    {
      prev->link = ptr->link;
      free(ptr);
    }
  return start;
}
```

3. Search of a Node in a single Linked List :-



Algorithm :

SEARCHNODE(INFO, LINK, START, PTR, ITEM)

Step 1: Let LOC := 0, F := 0

Step 2: PTR := START

Step 3: Repeat while PTR != NULL

Step 3.1: Set LOC := LOC + 1

Step 3.2: If ITEM = INFO[PTR], then

F := 1

BREAK

[END of IF]

PTR := LINK[PTR]

[END of While]

Step 4: If F = 0, Then Display "Item Not Found"

Step 5: Else Display "Item Found at Location" LOC

[End of If]

Step 6: Exit.

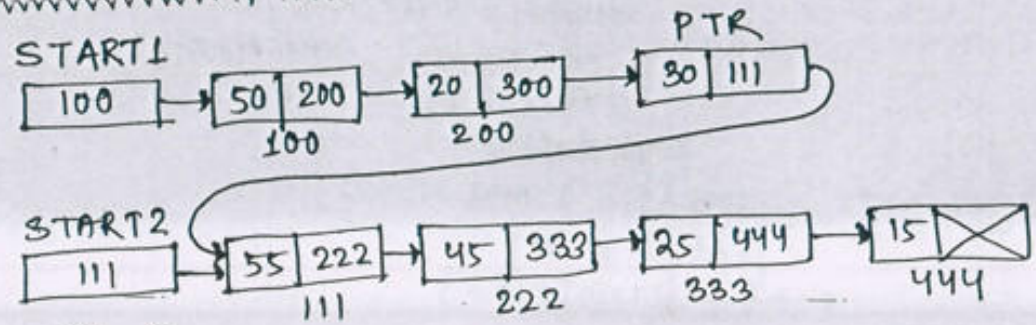
C-function :

```
void searching (struct node * start)
{ int item, loc = 1;
  struct node *ptr;
  printf("\n enter the item to search:");
  scanf("%d", &item);
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
ptr = start;
while(ptr != NULL && item != ptr->info)
{
  ptr = ptr->link;
   loc ++;
}
if(ptr == NULL)
    printf("\n item not found");
else
  {
    printf("\n item found at location = %d", loc);
  }
}
```

4. Concatenation of two single linked lists :



Algorithm :

Step 1 : Let PTR
Step 2 : PTR := START 1
Step 3 : Repeat While LINK[PTR] != NULL
PTR := LINK[PTR]
[End of While]
Step 4 : LINK[PTR] := START 2
Step 5 : START 2 := NULL
Step 6 : EXIT.

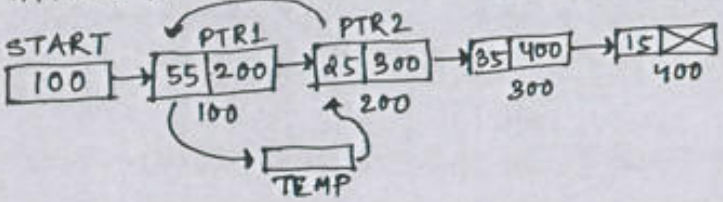
G.I.F.T. CENTRAL LIBRARY BHUBANESWAR

C - function :

```
struct node* concatenate (struct node *start1, struct node *start2)
{ struct node *ptr;
    ptr = start1;
    while (ptr->link != NULL)
    { ptr = ptr->link;
    } ptr->link = start2;
      start2 = NULL;
      return start1;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

5. Sorting of a single linked list :



Algorithm:

```
SORTING(START, INFO, LINK)
  Step 1: Let PTR1, PTR2, TEMP
  Step 2: PTR1 := START
  Step 3: Repeat while LINK[PTR1] != NULL
      Step 3.1: PTR2 := LINK[PTR1]
      Step 3.2: Repeat while PTR2 != NULL
          Step 3.2.1: If INFO[PTR1] > INFO[PTR2], then
                        TEMP := INFO[PTR1]
                        INFO[PTR1] := INFO[PTR2]
                        INFO[PTR2] := TEMP
                      [End of IF]
                    PTR2 := LINK[PTR2]
                  [End of while]
                PTR1 := LINK[PTR1]
              [End of while]
  Step 4: Exit
```

'c' function:

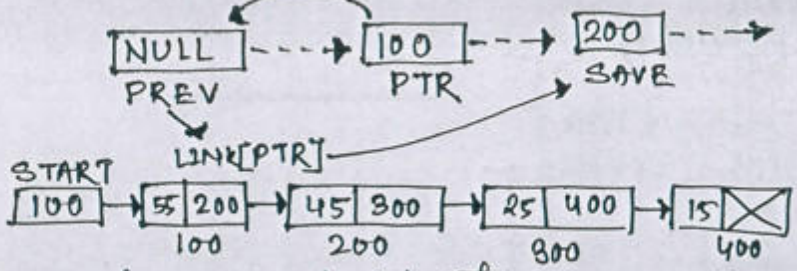
```
void sorting(struct node *start)
{ struct node *ptr1, *ptr2;
  int temp;
  ptr1 = start;
  while(ptr1 -> link != NULL)
  {
    ptr2 = ptr1 -> link;
    while (ptr2 != NULL)
    { if (ptr1 -> info > ptr2 -> info)
      { temp = ptr1 -> info;
        ptr1 -> info = ptr2 -> info;
        ptr2 -> info = temp;
      }
      ptr2 = ptr2 -> link;
    }
    ptr1 = ptr1 -> link;
  }
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

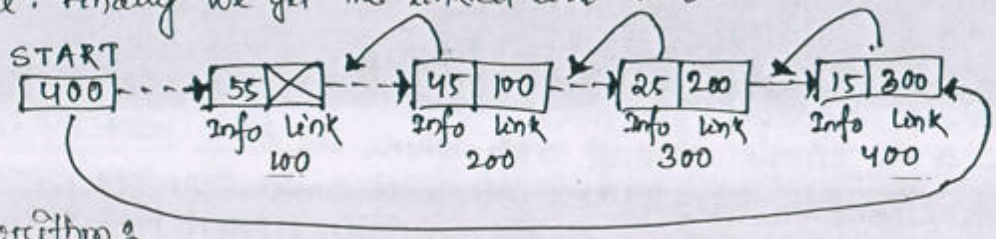
Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

6. Reversing a single Linked List :

→ Reversing of single linked list means pointing the START pointer to the last node, then the last node link part have to point to its previous node and that node have to point to its previous node and so on till 1st node, where the 1st node link part will point to NULL.



→ Here for each node identified by PTR it need to connect in reverse order e.g. when PTR = 100. LINK[PTR] points to NULL as it becomes last node. Then PTR = SAVE to point to the next node. Finally we get the linked list as :-



Algorithm :

REVERSE(START, INFO, LINK)
Step 1 : Let PREV, SAVE, PTR
Step 2 : PTR := START.
Step 3 : PREV := NULL
Step 4 : Repeat while PTR != NULL
SAVE := LINK[PTR]
LINK[PTR] := PREV
PREV := PTR
PTR := SAVE
[End of while]
Step 5 : START := PREV
Step 6 : Exit.

GIFT CENTRAL LIBRARY BHUBANESWAR

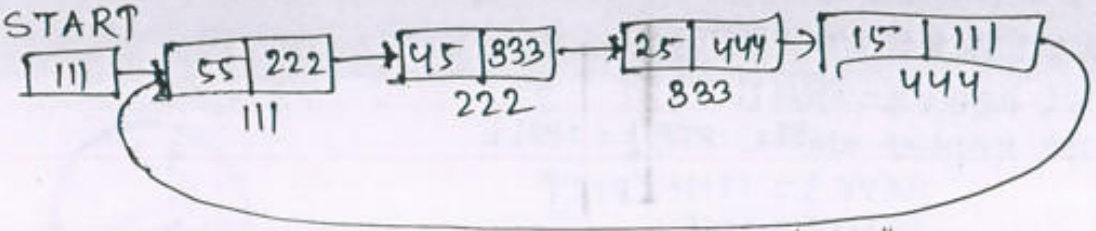
Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

'C'- function:

```
struct node *reverse(struct node *start)
{
  struct node *prev, *ptr, *save;
   ptr = start;
   prev = NULL;
  while(ptr != NULL)
  {
     save = ptr → link;
     ptr → link = prev;
     prev = ptr;
     ptr = save;
  }
  start = prev;
  return start;
}
```

Circular Singly Linked List:

1. It is a single linked list where the link part of last node contains the address of first node. In other words, when we traverse a circular single linked list, the first node can be visited after the last node.

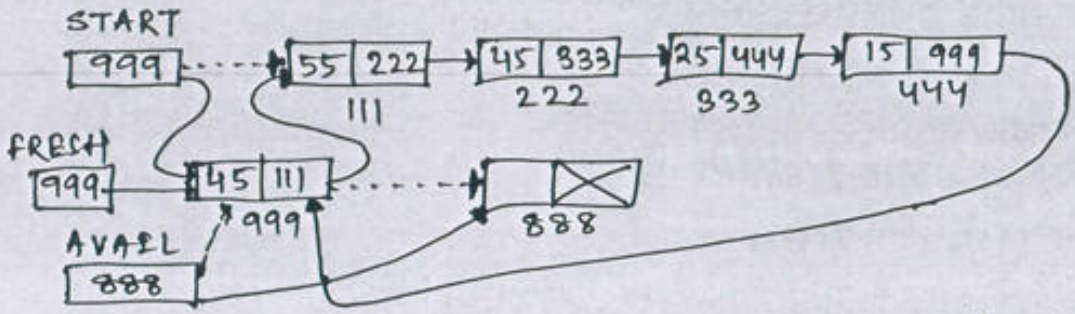


operations on circular singly linked list:-

1. Insertion at the beginning
2. Insertion at the end
3. Deletion at the beginning
4. Deletion at the end.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

1. Insertion of a node at the beginning:



Here, the FRESH node with address 999 is to be inserted. The link part of fresh node is updated with 111 which will become the 2nd node. The link part of last node i.e. node with address 444 is updated with address of fresh node 999. START is assigned with the address of fresh node 999.

Algorithm:

INSERTFIRST(START, INFO, LINK, AVAIL, FRESH, ITEM)

Step 1: If AVAIL = NULL, Then Display "memory Insufficient"

Step 2: Else

FRESH := AVAIL

AVAIL := LINK[AVAIL]

INFO[FRESH] := ITEM

LINK[FRESH] := NULL

Step 2.1: If START = NULL Then

START := FRESH

LINK[FRESH] := START

Step 2.2: Else

LINK[FRESH] := START

PTR := START

Repeat while LINK[PTR] != START

PTR := LINK[PTR]

[End of while]

LINK[PTR] := FRESH

START := FRESH

[End of IF]

Step 3: Exit.

G.I.F.T CENTRAL LIBRARY BHUBANESWAR

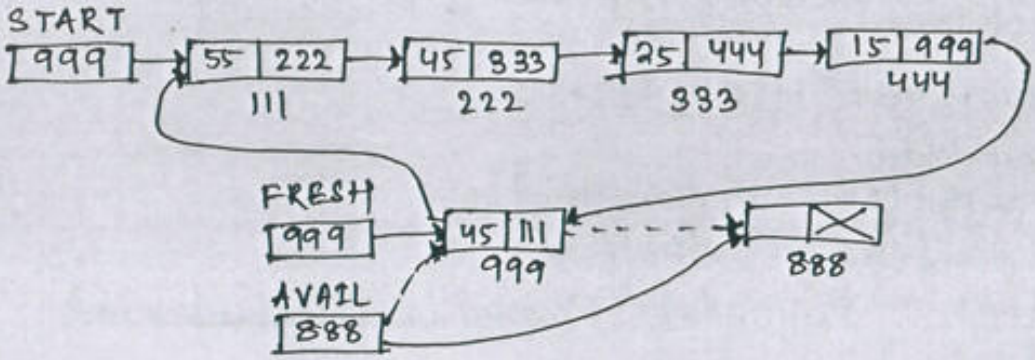
Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

'C' - Function :

```
struct node *insert_first(struct node *start)
{
  struct node *fresh, *ptr;
  int item;
  printf("\n enter an item");
  scanf("%d", &item);
  fresh = (struct node*) malloc(sizeof(struct node));
  if (fresh == NULL)
  {
    printf("\n memory is full - overflow");
    exit(0);
  }
  else
  {
    fresh -> info = item;
    fresh -> link = NULL;
    if(start == NULL)
    {
      start = fresh;
      fresh -> link = start;
    }
    else
    {
      for (ptr = start; ptr -> link != start; ptr = ptr -> link);
      ptr -> link = fresh;
      fresh -> link = start;
      start = fresh;
    }
  }
  return start;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

2. Insertion of a node at the end :



Algorithm :

INSERTEND (START, INFO, LINK, AVAIL, FRESH, ITEM)

Step 1 : If AVAIL = NULL, then Display "Memory Insufficient"

Step 2 : Else

FRESH := NULL

AVAIL := LINK[AVAIL]

INFO[FRESH] := ITEM

LINK[FRESH] := NULL

Step 2.1 : If START = NULL, then

START := FRESH

LINK[FRESH] := START

Step 2.2 : Else

PTR := START

Repeat while LINK[PTR] != START

PTR := LINK[PTR]

[End of while]

LINK[PTR] := FRESH

LINK[FRESH] := START.

[End of If - Step 2.1]

[End of If - Step - 1]

Step 3 : - Exit.

G.I.F.T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

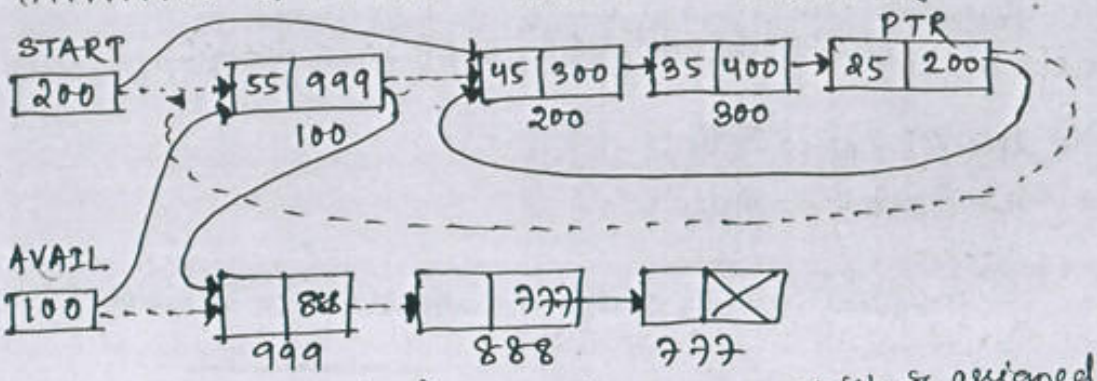
Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

C-function:

```
struct node * insert_last(struct node *start)
{ struct node *fresh, *ptr;
  int item;
  printf("\n enter an item");
  scanf("%d", &item);
  fresh = (struct node*) malloc (sizeof(struct node));
  if (fresh == NULL)
      printf("\n memory is full");
  else
   { fresh -> info = item;
     fresh -> link = NULL;
    if (start == NULL)
    { start = fresh;
      fresh -> link = start;
    }
    else
    { for(ptr = start; ptr -> link != start; ptr = ptr -> link)
        ptr -> link = fresh;
        fresh -> link = start;
    }
  }
  return start;
}.
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

3. Deletion of a node from the beginning:



Here, the link part of 1st node is 200 which is assigned to START and by moving to the last node, the link part of last node updated with address 200 as pointed by START. Now the 1st node with address 100 is added at the beginning of free or avail list.

Algorithm:

DELETE FIRST (START, INFO, LINK, AVAIL)

Step 1: If START = NULL, then
Display "List doesnot exist".

Step 2: Else if START = LINK[START], then
LINK[START] := AVAIL
AVAIL := START
START := NULL

Step 3: Else
PTR := START
Repeat While LINK[PTR] != START
PTR := LINK[PTR]
[End of While]
LINK[PTR] := LINK[START]
PTR := START
START := LINK[START]
LINK[PTR] := AVAIL
AVAIL := PTR
[End of IF]

Step 4: Exit.

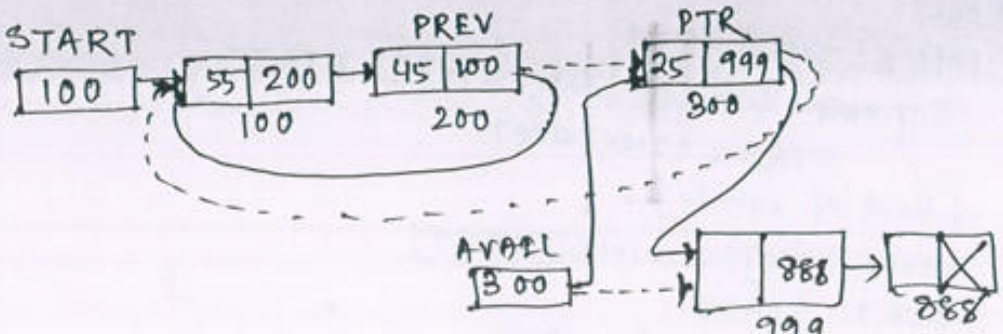
G.F.T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

'C' - function :

```
struct node *delete_first(struct node *start)
{ struct node *ptr;
  if (start == NULL)
  {
    printf("\n memory underflow - list is empty");
  }
  else
  {
    ptr = start;
    while (ptr -> link != start)
    {
      ptr = ptr -> link;
    }
    ptr -> link = start -> link;
    ptr = start;
    start = start -> link;
    free(ptr);
  }
  return start;
}
```

4. Deletion of a node from the end :



Here, the link part of last but one node as pointed by PREV with address 200 is updated with address of first node 100. Now the last node as pointed by PTR is added at the beginning of free or avail list.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm:

DELETE END (START, INFO, LINK, AVAIL)

Step 1: If START = NULL, Then Display "List Not Exist"

Step 2: Else if START = LINK[START], Then
LINK[START] := AVAIL
AVAIL := START
START := NULL

Step 3: Else
PTR = START
Repeat, while LINK[PTR] != START
PREV := PTR
PTR := LINK[PTR]
[End of while]
LINK[PREV] := START
LINK[PTR] := AVAIL
AVAIL := PTR
[End of IF]

Step 4: Exit.

GIFT CENTRAL LIBRARY BHUBANESWAR

'C'- Function:

```
struct node *delete_last(struct node *start)
{ struct node *ptr, *prev;
  if(start == NULL)
  { printf("\n memory underflow - list is empty");
  }
  else if(start == start->link)
  { ptr = start;
    start = NULL;
    free(ptr);
  }
  else{ ptr = start;
        while(ptr->link != start)
        { prev = ptr;
          ptr = ptr->link;
        }
        prev->link = start;
        free(ptr);
      return start;
}
```

Advantages of circular singly linked list:

(i) Nodes can be accessed easily
(ii) Deletion of nodes is easier
(iii) Concatenation and splitting of circular linked list can be done efficiently.

Drawbacks of circular singly linked list:

(i) It may enter into an infinite loop.
(ii) Head node is required to indicate the START or END of the circular linked list.
(iii) Backward traversing is not possible

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Data Structure Using C

Topic:

Doubly Linked List

Contributed By:

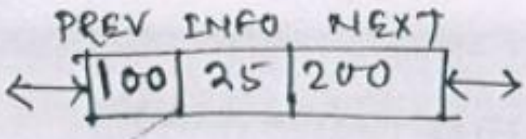
Nihar Ranjan Rout

Gandhi Institute For Technology, GIFT

Doubly Linked List :

1. A doubly linked list is a collection of nodes where each node contains the address of next as well as address of previous node along with INFO part to hold the information. In other words, it is also referred as two-way list as traversing is possible in both the directions.

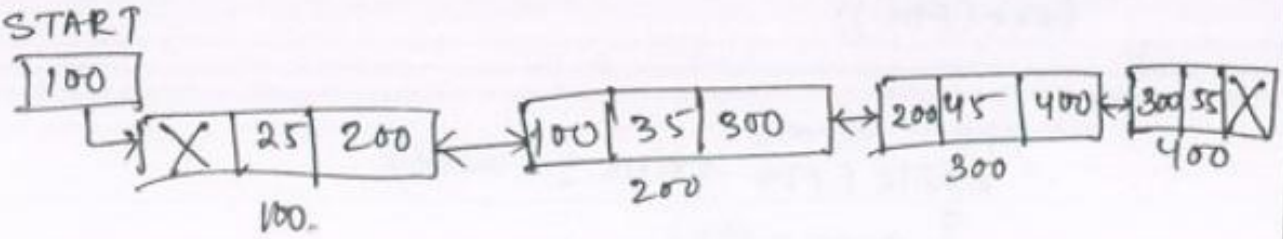
Eg:-



2. The structure of a node in 'C', is:-

```
struct node
{
    int info
    struct node *prev, *next;
};
```

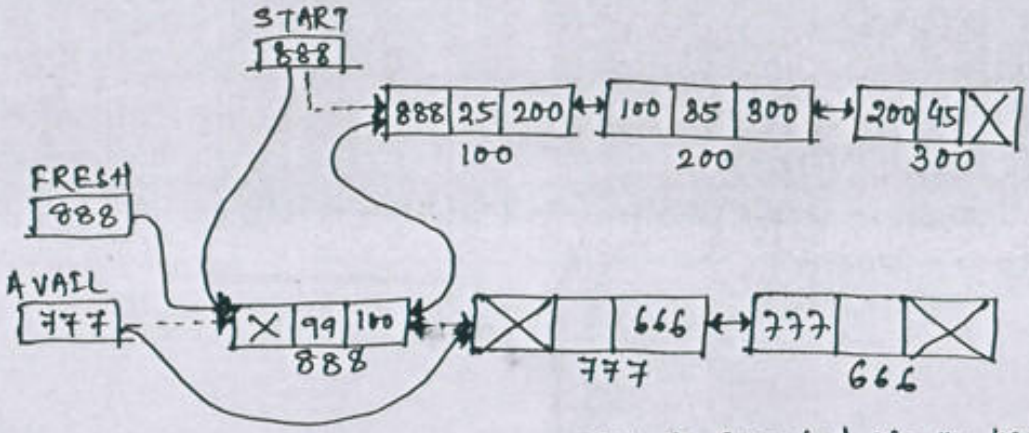
Eg:-



Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Operations on doubly linked list :

1. Insertion of a node at the beginning :



Here, the fresh node with address 888 is inserted at the beginning of the doubly linked list. The NEXT part of fresh node is updated with the address contained in START. i.e.. 100 and START is updated with 888, the address of fresh node.

Algorithm :

INSERT FIRST(START, INFO, NEXT, PREV, FRESH, AVAIL)

Step 1 : If AVAIL = NULL, Then Display "Insufficient Memory"

Step 2 : Else

FRESH := AVAIL

AVAIL := NEXT[AVAIL]

PREV[AVAIL] := NULL

NEXT[FRESH] := NULL

PREV[FRESH] := NULL

INFO[FRESH] := ITEM

Step 2.1 : If START = NULL, then START := FRESH

Step 2.2 : Else

NEXT[FRESH] := START

PREV[START] := FRESH

START := FRESH

[End of IF - step 2.1]

[End of IF - Step-1]

Step-3 : Exit

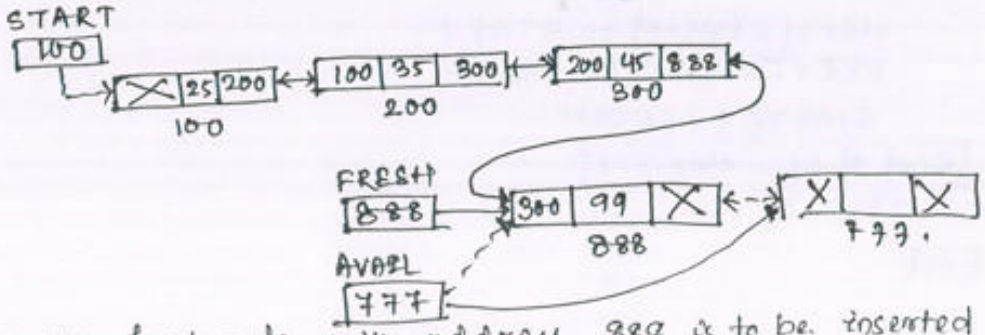
G I F T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

c-function:

```
struct node *insert_first(struct node *start)
{
    struct node *fresh;
    int item;
    printf("\n enter item");
    scanf("%d", &item);
    fresh = (struct node*) malloc (sizeof (struct node));
    if(fresh == NULL)
        printf("\n memory is full");
    else
    {
        fresh -> info = item;
        fresh -> next = NULL;
        fresh -> prev = NULL;
        if(start == NULL)
        {
            start = fresh;
        }
        else
        {
            fresh -> next = start;
            start -> prev = fresh;
            start = fresh;
        }
    }
    return start;
}
```

2. Insertion of a node at the end:



Here, the fresh node with address 888 is to be inserted at the end. PTR is made to point the last node with address 300. Now the NEXT part of node with address 300 as pointed by PTR is assigned with address of fresh ie:- 888 and PREV part of fresh node is updated with address 300 as pointed by PTR.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm:

INSERT END (START, INFO, NEXT, PREV, FRESH, AVAIL, PTR)

Step 1: If AVAIL = NULL, Then Display "Insufficient Memory"

Step 2: Else

FRESH := AVAIL
AVAIL := NEXT[AVAIL]
PREV[AVAIL] := ITEM
NEXT[FRESH] := NULL
PREV[FRESH] := NULL
INFO[FRESH] := ITEM

Step 2.1: If START = NULL, Then START := FRESH

Step 2.2: Else

PTR := START
While NEXT[PTR] != NULL
PTR := NEXT[PTR]
[End of while]
NEXT[PTR] := FRESH
PREV[FRESH] := PTR

[End of IF - Step 2.1]

[End of If - Step 1]

Step 3: Exit

'C' Program:

```
struct node *insert_last(struct node *start)
{
  struct node *fresh, *ptr;
  int item;
  printf("\n enter an item");
  scanf("%d", &item);
  fresh = (struct node *) malloc(sizeof(struct node));
  if (fresh == NULL)
        printf("\n memory is full");
  else
    { fresh -> info = item;
      fresh -> next = NULL;
      fresh -> prev = NULL;
      if(start == NULL)
      { start = fresh;
      }
```

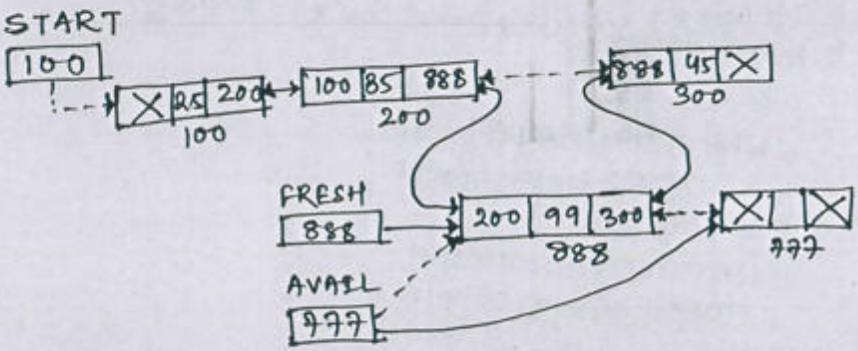
GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE


```
else
{
  for(ptr = start; ptr -> next != NULL; ptr = ptr -> next)
    ptr -> next = fresh;
    fresh -> prev = ptr;
}
return start;
}
```

3. Insertion of a node at specific location :



Algorithm :

INSERT_AT_LOC (START, INFO, NEXT, PREV, FRESH, AVAIL, PTR)

Step 1 : If AVAIL = NULL, Then Display "Insufficient Memory".

Step 2 : Else

FRESH = AVAIL
AVAIL := NEXT[AVAIL]
PREV[AVAIL] := NULL
NEXT[FRESH] := NULL
PREV[FRESH] := NULL
INFO[FRESH] := ITEM

Step 2.1 : If START = NULL, Then START := FRESH

Step 2.2 : Else

Step 2.2.1 : Set I := 1, PTR1 := START
Step 2.2.2 : Repeat While I < LOC AND PTR1 != NULL
Set PTR := PTR1
Set PTR1 := NEXT[PTR1]
Set I := I + 1
[End of While]

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

step 2.2.3: If PTR1 = NULL, Then
Display "Location doesnot exist".
step 2.2.4: Else If PTR1 = START, Then
NEXT[FRESH] := START
PREV[START] := FRESH
step 2.2.5: Else
NEXT[PTR] := FRESH
PREV[FRESH] := PTR
NEXT[FRESH] := PTR1
PREV[PTR1] := FRESH
[End of If - step 2.2.3]
[End of If - step 2.1]
[End of If - step 1]
Step 3: Exit.

'C' function:

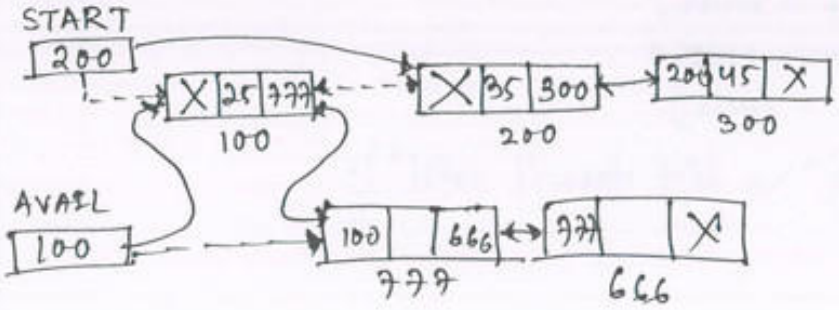
```
struct node *insert_loc(struct node *start)
{ struct node *fresh, *ptr, *ptr1;
  int item, loc, c;
  fresh = (struct node*) malloc(sizeof(struct node));
  if(fresh == NULL)
  { printf("\n memory is full");
  }
  else
  { printf("\n enter item and location for insertion");
    scanf("%d%d", &item, &loc);
    fresh -> info = item;
    fresh -> next = NULL;
    fresh -> prev = NULL;
    if(start == NULL)
    { printf("\n list doesn't exist");
    }
```

G I F T CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
else
{
  ptr1 = start;
  i=1;
  while(i<loc && ptr1 != NULL)
  {
    ptr = ptr1;
    ptr1 = ptr1 -> next;
    i++;
  }
  if(ptr1 == NULL)
  { printf("\n invalid entry of location");
  }
  else if(ptr1 == start)
  { // insertion at beginning
    fresh -> next = ptr;
    ptr -> prev = fresh;
    start = fresh;
  }
  else
  { ptr -> next = fresh;
    fresh -> prev = ptr;
    fresh -> next = ptr1;
    ptr1 -> prev = fresh;
  }
}
return start;
}
```

4. Deletion of a node from the beginning:



Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm:

DELETE FIRST (START, PREV, NEXT, PTR, AVAIL)

Step 1: If START = NULL, Then
Display "Linked List Doesn't Exist or Underflow".

Step 2: Else
PTR := START
START := NEXT[PTR]
NEXT[PTR] := AVAIL
PREV[AVAIL] := PTR
AVAIL := PTR
[End of If]

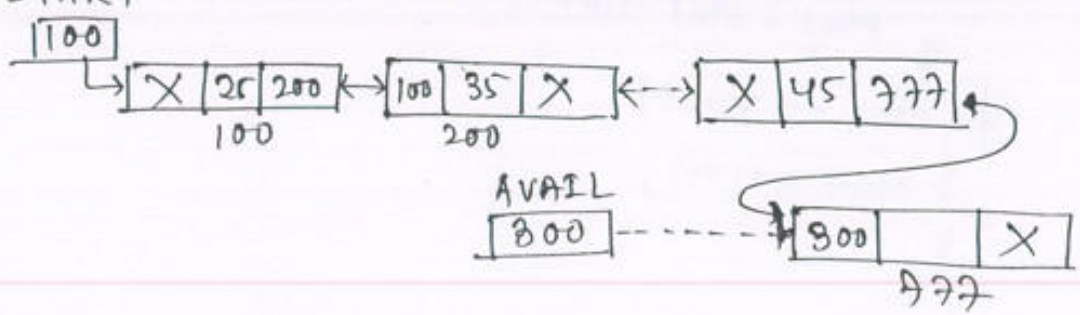
Step-3: Exit.

'C'-function:

```
struct node *delete_first(struct node *start)
{
  struct node *ptr;
  if(start == NULL)
  {
    printf("\n memory underflow - list is empty");
  }
  else
  {
    ptr = start;
    start = start -> next;
    start -> prev = NULL;
    free(ptr);
  }
  return start;
}
```

G I F T CENTRAL LIBRARY BHUBANESWAR

5. Deletion of a node from the end:



Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm:

DELETEEND(START, PREV, NEXT, PTR, PTR1, AVAIL)

Step 1: If START = NULL, then
Display "Linked list Doesn't Exist or Underflow"

Step 2: Else

Step 2.1: PTR1 = START

Step 2.2: Repeat while NEXT[PTR1] != NULL
Set PTR := PTR1
Set PTR1 := NEXT[PTR1]
[End of while]

Step 2.3: NEXT[PTR] := NULL
PREV[PTR1] := NULL
NEXT[PTR1] := AVAIL
PREV[AVAIL] := PTR1
AVAIL := PTR1

[End of If]

Step 3: Exit

C-function:

```
struct node *delete_last(struct node *start)
{
  struct node *ptr, *ptr1;
  if (start == NULL)
  { printf("\n memory underflow - list is empty");
  }
  else
  { ptr1 = start;
    while (ptr1 -> next != NULL)
    { ptr = ptr1;
      ptr1 = ptr1 -> next;
    }
    if (ptr1 == start)
    { start = NULL;
    }
```

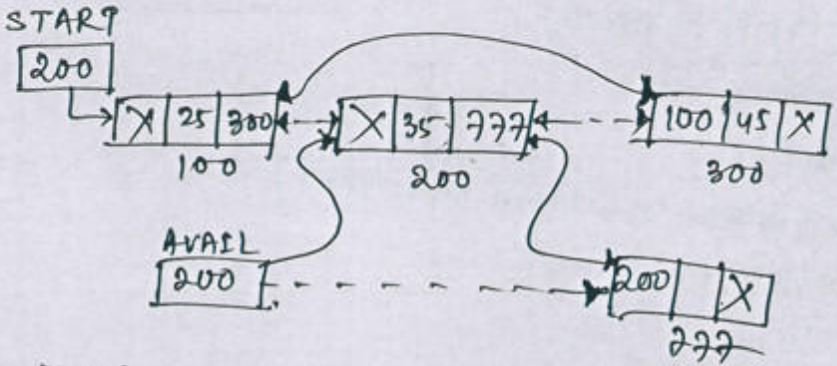

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE


```
    else
    {
        ptr -> next = NULL;
    }
    free(ptr1);
}
return start;
}
```

6. Deletion of a node from Specific position :



GIFT CENTRAL LIBRARY BHUBANESWAR

Algorithm :

DELETEPOS (START, LOC, PTR, PTR1, ITEM, AVAIL)

Step 1 : Let I := 1

Step 2 : If START = NULL, then Display "List not exist"

Step 3 : Else

Step 3.1 : PTR1 := START

Step 3.2 : Repeat while I < LOC AND PTR1 != NULL

PTR := PTR1

PTR1 := NEXT[PTR1]

I := I + 1

[End of while]

Step 3.3 : If PTR1 = NULL, Then

Display "Location entered not exist"

Step 3.4 : Else If PTR1 = START, Then

START := NEXT[START]

PREV[START] := NULL

Step 3.5 : Else If NEXT[PTR1] = NULL, Then

NEXT[PTR] := NULL

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Step 3.6: Else
NEXT[PTR] := NEXT[PTR1]
PREV[NEXT[PTR1]] := PTR
[End of step 3.3]
Step 3.7: NEXT[PTR1] := AVAIL
PREV[AVAIL] := PTR1
PREV[PTR1] := NULL
AVAIL := PTR1
[End of step-2]

Step 4: Exit.

C-function:

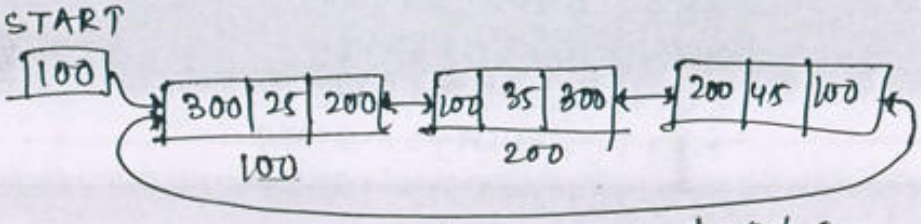
```
struct node *delete_loc(struct node *start)
{ struct node *ptr, *ptr1;
  int loc, i;
  if(start == NULL)
  { printf("\n memory underflow - list is empty");
  }
  else
  { printf("\n enter the location/node no. for deletion");
    scanf("%d", &loc);
    ptr1 = start;
    i = 1;
    while( i<loc && ptr1 != NULL)
    {
      ptr = ptr1;
      ptr1 = ptr1 -> next;
      i++;
    }
  if(ptr1 == NULL)
      printf("\n location entered not exist");
  else if (ptr1 == start)
    { start = start -> next;
      start -> prev = NULL;
      free(ptr1);
    }
  else if (ptr1 -> next == NULL)
      ptr -> next = NULL;
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
else
  {
    ptr -> next = ptr1 -> next;
    ptr1 -> next -> prev = ptr;
    free (ptr1);
  }
}
return start;
}.
```

Circular Linked List (Doubly)

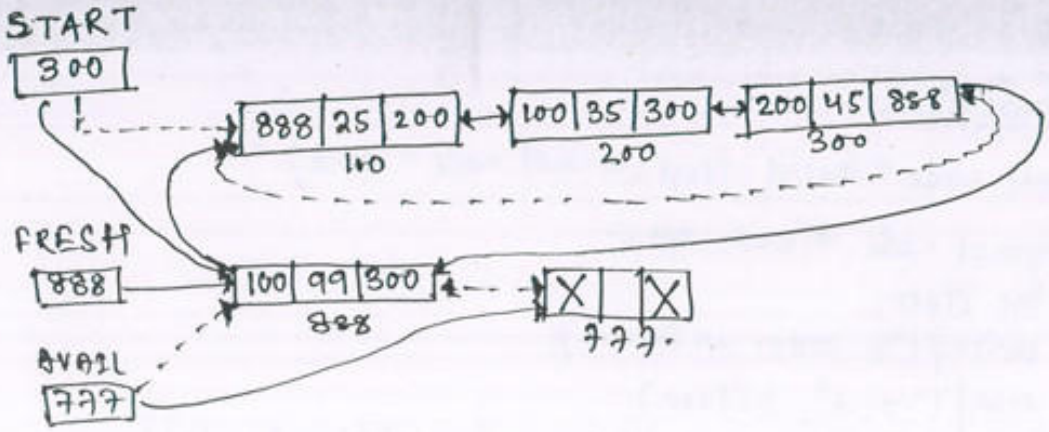
1. The doubly linked list of circular in nature is referred as circular doubly linked list i.e. In a circular doubly linked list, the NEXT part of last node contains the address of first node and the PREV part of first node contains address of last node.



GIFT CENTRAL LIBRARY BHUBANESWAR

Operation on circular doubly linked list :

1. Insertion of a node at the beginning :



Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Algorithm:

INSERT_FIRST(START, FRESH, INFO, PREV, NEXT, PTR, AVAIL)

Step 1: If AVAIL = NULL, then Display "Insufficient memory or overflow".

Step 2: Else

Step 2.1: FRESH := AVAIL
AVAIL := NEXT[AVAIL]
PREV[AVAIL] = NULL
PREV[FRESH] = NULL
NEXT[FRESH] = NULL

Step 2.2: If START = NULL, then
START := FRESH
PREV[FRESH] := START
NEXT[FRESH] := START

Step 2.3: Else
PTR := PREV[START]
NEXT[PTR] := FRESH
PREV[FRESH] := PTR
NEXT[FRESH] := START
PREV[START] := FRESH
START := FRESH

[End of IF]

[END of IF]

Step 3: Exit.

'C'-function:

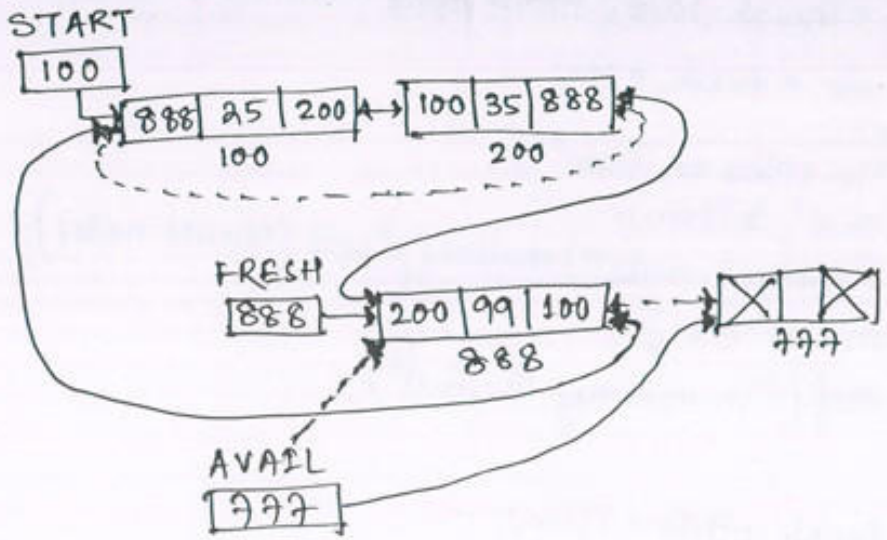
```
struct node *insert_first(struct node *start)
{ struct node *fresh, *ptr;
  int item;
  printf("\n enter an item");
  scanf("%d", &item);
  fresh = (struct node*) malloc (sizeof(struct node));
  if (fresh == NULL)
  { printf("\n memory is full");
  }
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
else
{
    fresh -> info = item;
    fresh -> next = NULL;
    fresh -> prev = NULL;
    if (start == NULL)
    {
        start = fresh;
        start -> prev = start;
        start -> next = start;
    }
    else
    {
        ptr = start -> prev;
        ptr -> next = fresh;
        fresh -> prev = ptr;
        fresh -> next = start;
        start -> prev = fresh;
        start = fresh;
    }
}
return start;
}
```

GIFT CENTRAL LIBRARY RHUBANESWAR

2. Insertion of a node at the end :



Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

Algorithm:

```
INSERTEND(START, FRESH, INFO, PREV, NEXT, PTR, AVAIL)
Step 1: If AVAIL = NULL, then display "Insufficient memory or Overflow"
Step 2: Else
    Step 2.1: FRESH := AVAIL
              AVAIL := NEXT[AVAIL]
              PREV[AVAIL] = NULL
              PREV[FRESH] = NULL
              NEXT[FRESH] = NULL
    Step 2.2: If START = NULL, then
                 START := FRESH
                 PREV[FRESH] := START
                 NEXT[FRESH] := START

    Step 2.3: Else
                 PTR := PREV[START]
                 NEXT[PTR] := FRESH
                 PREV[FRESH] := PTR
                 NEXT[FRESH] := START
                 PREV[START] := FRESH
          [End of IF]
    [End of IF]
Step 3: Exit.
```

C-function:

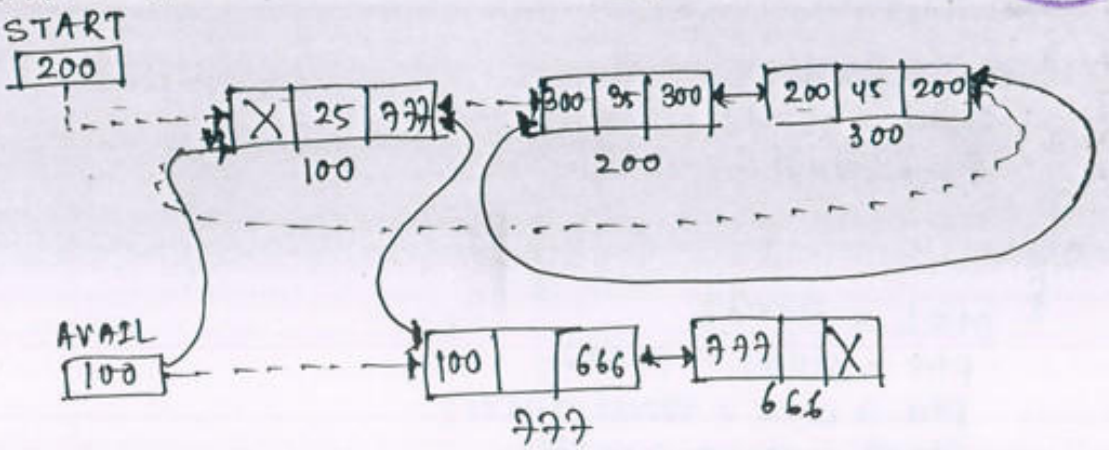
```
struct node *insert_last(struct node *start)
{ struct node *fresh, *ptr;
  int item;
  printf("\n enter an item");
  scanf("%d", &item);
  fresh = (struct node*) malloc(sizeof(struct node));
  if (fresh == NULL)
  { printf("\n memory is full");
  }
  else
  { fresh->info = item;
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
fresh → next = NULL;
fresh → prev = NULL;
if (start == NULL)
{
    start = fresh;
    start → prev = start;
    start → next = start;
}
else
{
    ptr = start → prev;
    ptr → next = fresh;
    fresh → prev = ptr;
    fresh → next = start;
    start → prev = fresh;
}
return start;
}
```

GIFT CENTRAL LIBRARY BHUBANESWAR

3. Deletion of a node from the beginning :



Algorithm :

DELETEFIRST(START, PTR, PTR1, PREV, NEXT, AVAIL)

Step 1 : If START = NULL, Then Display "List Not Exist"

Step 2 : Else If NEXT[START] = START, Then

```
PTR := START
START := NULL
NEXT[PTR] := AVAIL
PREV[AVAIL] := PTR
AVAIL := PTR
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

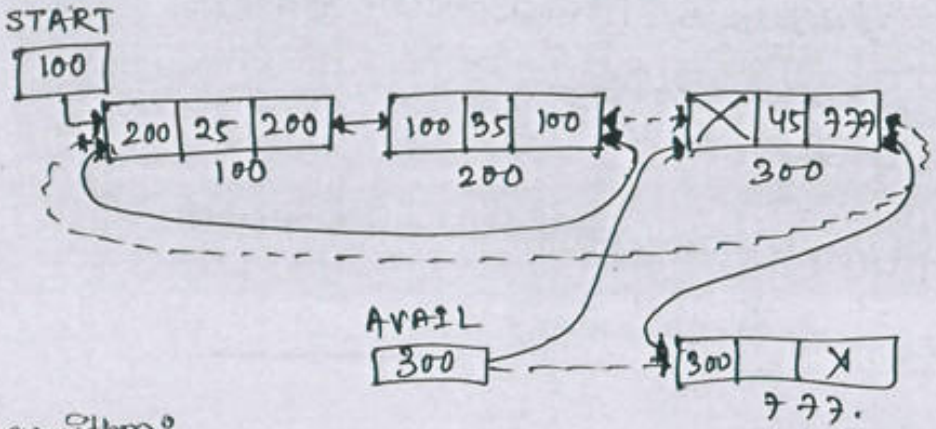
Step 3: Else
PTR := PREV[START]
PTR1 := START
NEXT[PTR] := NEXT[PTR1]
START := NEXT[PTR1]
PREV[START] := PTR
NEXT[PTR1] := AVAIL
PREV[AVAIL] := PTR1
AVAIL := PTR1
[End of If]
Step 4: Exit.

'C'-function:

```
struct node *delete_first(struct node *start)
{
  struct node *ptr, *ptr1;
  if(start == NULL)
  {
    printf("\n memory underflow - list is empty");
  }
  else if(start == start->next)
  {
    ptr = start;
    start = NULL;
    free(ptr);
  }
  else
  {
    ptr1 = start;
    ptr = start->prev;
    ptr->next = start->next;
    start = start->next;
    start->prev = ptr;
    free(ptr1);
  }
  return start;
}
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

4. Deletion of a node from the end :



Algorithm:

DELETE END (START, PTR, PTR1, PREV, NEXT, AVAIL)

Step 1: If START = NULL, Then Display "List not exist"

Step 2: Else If START = NEXT[START], Then

PTR := START
START := NULL
NEXT[PTR] := AVAIL
PREV[AVAIL] := PTR
AVAIL := PTR

Step 3: Else

PTR1 := PREV[START]
PTR := PREV[PTR1]
NEXT[PTR] := START
PREV[START] := PTR
NEXT[PTR1] := AVAIL
PREV[AVAIL] := PTR1
AVAIL := PTR1.

[End of IF]

Step 4: Exit.

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

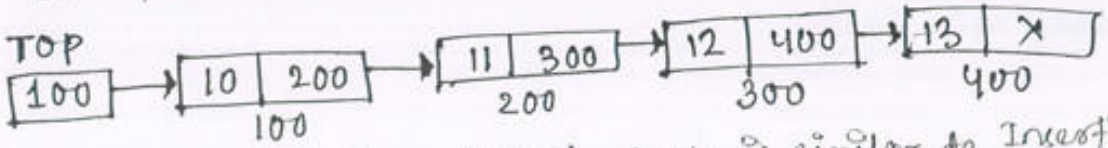
C-function:

```
struct node *delete_last(struct node *last)
{ struct node *ptr, *ptr1;
  if(start == NULL)
  { printf("\n memory underflow - list is empty");
  }
  else if(start == start -> next)
  {
     ptr = start;
     start = NULL;
     free(ptr);
  }
  else
  {
     ptr1 = start -> prev;
     ptr = ptr1 -> prev;
     ptr -> next = start;
     start -> prev = ptr;
     free(ptr1);
  }
  return start;
}
```

Applications of Linked list:

1. Linked stack:

→ When the concept of stack is implemented using singly linked list, it is referred as linked stack. In linked stack, assume TOP is the pointer that identifies starting or topmost node.



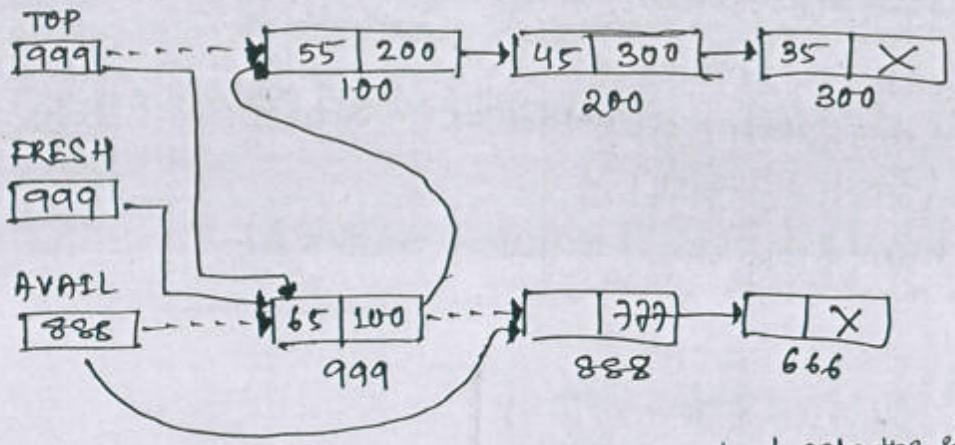
→ PUSH operation in a linked stack is similar to Insertion of a node at the beginning of single linked list. Pop operation in a linked stack is similar to Deletion of a node from the beginning of a single linked list.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

a) Push operation in a linked stack:



Here node with address 999 is to be pushed onto the stack. Initially Top holds the address of node 100. In order to push the fresh node 999, the link part of it is updated with 100 and Top is assigned with the address of fresh 999.

Algorithm:

PUSH(AVAIL, FRESH, LINK, INFO, TOP)
Step 1: If AVAIL = NULL, Then Display "Insufficient memory or Overflow"
Step 2: Else
FRESH := AVAIL
AVAIL := LINK[AVAIL]
LINK[FRESH] := NULL
INFO[FRESH] := ITEM

Step 2.1:
If TOP = NULL, Then TOP := FRESH

Step 2.2:
Else
LINK[FRESH] := TOP
TOP := FRESH
[End of If - Step 2.1]
[End of If - Step 1]
Step 3: Exit.

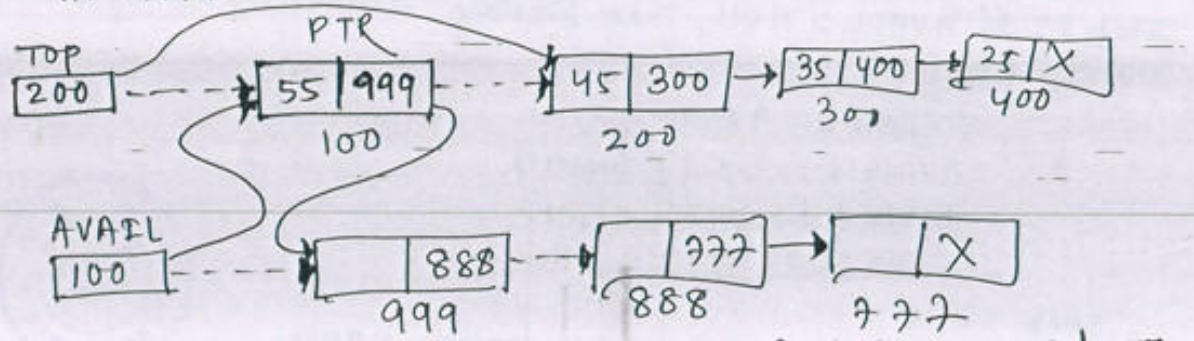
GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

C-function:

```
struct node *push(struct node *top, int item)
{
    struct node *fresh;
    fresh = (struct node*) malloc (sizeof (struct node));
    if(fresh == NULL)
    {
        printf("\n stack is full or overflow");
    }
    else
    {
        fresh -> info = item;
        fresh -> link = top;
        top = fresh;
    }
    return top;
}
```

b) Pop operation on a linked stack:



Here, the top node with address 100 is to be popped. The TOP is updated with address 200. The node 100 is added at the beginning of free or avail list.

Algorithm:

```
POP(TOP, PTR, LINK, AVAIL)
Step 1: If TOP = NULL, Then Display "Underflow"
Step 2: Else
            PTR := TOP
            TOP := LINK[TOP]
            LINK[PTR] := AVAIL
            AVAIL := PTR
        [End of If]
Step 3: Exit.
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

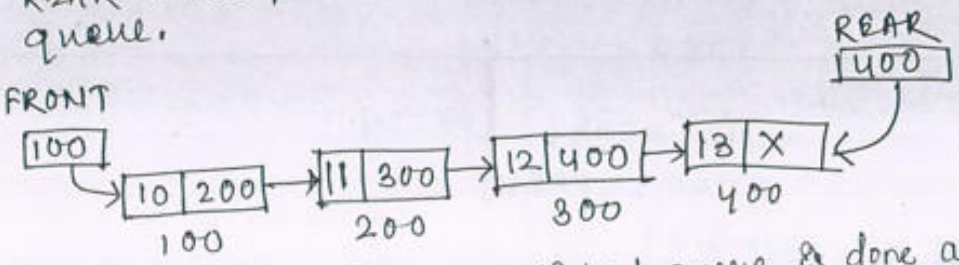
C-function:

```
struct node *pop(struct node *top)
{
  struct node *ptr;
  int pitem;
  if(top == NULL)
  {
    printf("\n stack is empty or underflow");
  }
  else
  {
    pitem = top -> info;
    printf("\n popped item is %d", pitem);
    ptr = top;
    top = top -> link;
    free(ptr);
  }
  return top;
}
```

GIFT CENTRAL LIBRARY BHUBANESWAR

2. Linked Queue :

→ When the concept of queue is implemented using singly linked list, it is referred as linked queue. In linked queue, assume FRONT is the pointer that identifies first node and REAR is the pointer that identifies the last node of linked queue.



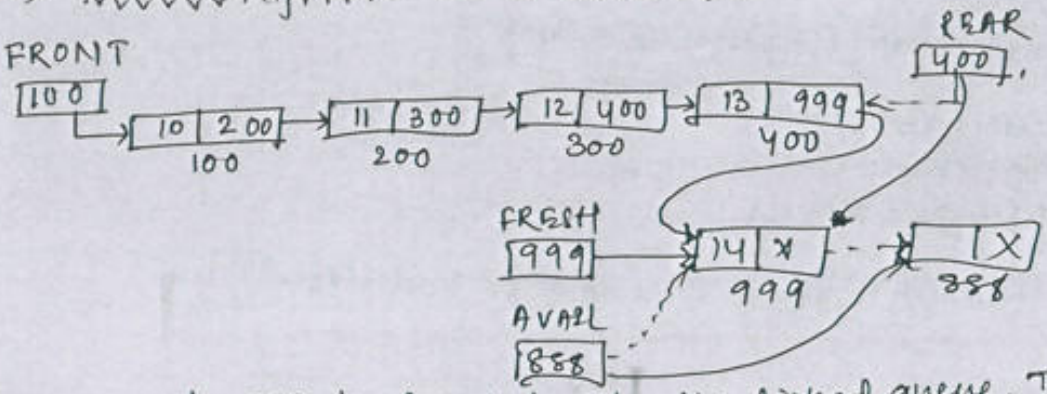
→ Insertion operation in a linked queue is done at the rear end of the linked queue. Insertion operation in a linked queue is similar to insertion at the end of singly linked list.

→ Deletion operation in a linked queue is done at the front end of linked queue. Deletion operation in a linked queue is similar to deletion from the beginning of the singly linked list.

→ A linked queue can be traversed from the node as pointed by FRONT to node as pointed by REAR.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

3 a) Insertion operation in a linked queue :



Assume node 999 is inserted onto the linked queue. The link part of REAR node is updated with the address of fresh node 999 and REAR is made to point to fresh node 999.

Algorithm:

LINKED_QUEUE_INSERTION(AVAIL, FRESH, LINK, FRONT, REAR, INFO, ITEM)

Step 1: If AVAIL = NULL, Then display "overflow"

Step 2: Else

FRESH := AVAIL

AVAIL := LINK[AVAIL]

LINK[FRESH] := NULL

INFO[FRESH] := ITEM

Step 2.1: If FRONT = NULL AND REAR = NULL, Then

FRONT := FRESH

REAR := FRESH

Step 2.2: Else

LINK[REAR] := FRESH

REAR := FRESH

[End of If - step 2.1]

[End of If - step 1]

Step 3: Exit.

C-function:

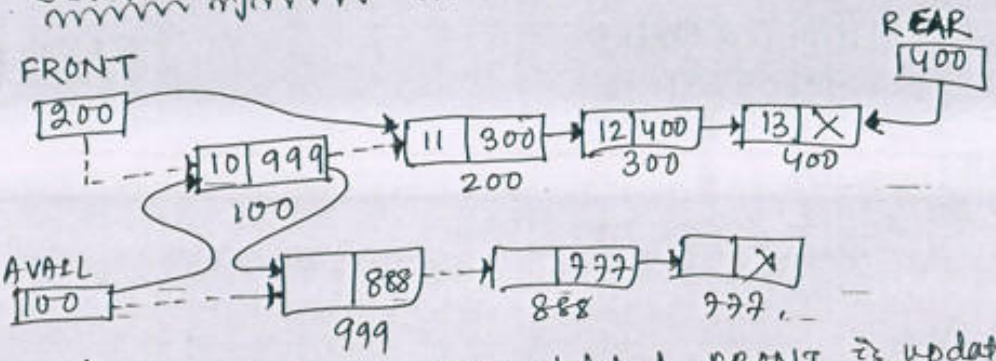
```
Void insertion(int item)
{ struct node *fresh;
  fresh = (struct node *) malloc(sizeof(struct node));
  if(fresh == NULL)
  {
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
        printf("\n queue is full or overflow");
    }
    else
    {   fresh -> info = item;
        fresh -> link = NULL;
        if (front == NULL && rear == NULL)
        {
            front = rear = fresh;
        }
        else
        {
            rear -> link = fresh;
            rear = fresh;
        }
    }
}
```

GIFT CENTRAL LIBRARY BHUBANESWAR

b) Deletion operation in a linked queue:



Here front node 100 is to be deleted. FRONT is updated with address 200. Now node 100 is added at the beginning of free list or avail list.

Algorithm:

LINKED_QUEUE - DELETION (AVAIL, LINK, FRONT, REAR, PTR)

Step 1: If FRONT = NULL AND REAR = NULL, then
Display "underflow"

Step 2: Else If FRONT = REAR, then
FRONT := NULL
REAR := NULL

Step 3: Else
PTR := FRONT
FRONT := LINK[FRONT]
LINK[PTR] := AVAIL
AVAIL := PTR
[End of If]

Step 4: Exit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

C-function:

```
void deletion(void)
{ struct node *ptr;
  int pitem;
  if (front == NULL)
  { printf("\n queue is empty or underflow");
  }
  else
  {
    pitem = front -> info;
    printf("\n deleted item is %d", pitem);
     ptr = front;
    if (front == rear)
      { front = NULL;
        rear = NULL;
      }
    else
      {
        front = front -> link;
        free(ptr);
      }
  } }
```

Polynomial Representation Using Linked List:

1. A single linked list can be used to represent and manipulate polynomials. A polynomial expression is a collection of polynomial terms, where each term contains a coefficient, base and exponent.

Representation of a polynomial term:

1. A node that can be used to represent a polynomial term with one variable, is divided into 3 parts such as :- Coefficient Part, Exponent Part and Link.

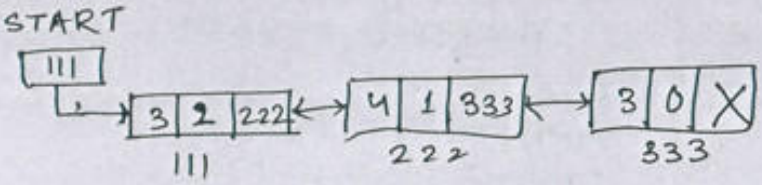
Eg:- To represent a polynomial term $3x^2$, the structure of node is:-

coefficient	exponent	link
3	2	200

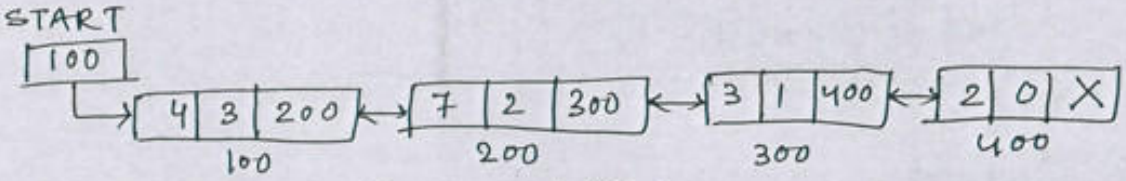
Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Example: Represent the following polynomial expressions in linked list format

i) $3x^2+4x+3$



ii) $4x^3+7x^2+3x+2$



Creating a polynomial using linked list:

Algorithm:

CREATE POLY (START, AVAIL, CO, EXP, LINK, FRESH, PTR, PTR1)

Step 1: If AVAIL = NULL, Then Display "Memory insufficient"

Step 2: Else

FRESH := AVAIL

AVAIL := LINK[AVAIL]

LINK[FRESH] := NULL

Display "Enter Co-efficient & Exponent Value"

READ CO[FRESH], EXP[FRESH]

Step 2.1: If START = NULL, Then START = FRESH

Step 2.2: Else PTR1 := START

Step 2.2.1: Repeat while PTR1 != NULL

If EXP[PTR1] < EXP[FRESH], Then

BREAK

[End of If]

PTR := PTR1

PTR1 := LINK[PTR1]

[End of while]

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Step 2.2.2 : If PTR1 = START, Then
LINK[FRESH] := PTR1
START := FRESH

Step 2.2.3 : Else If PTR1 = NULL, Then
LINK[PTR] := FRESH

Step 2.2.4 : Else
LINK[PTR] := FRESH
LINK[FRESH] := PTR1.

[End of If - step 2.2.2]

[End of If - step 2.1]

step 3 : exit.

C-function :

```
struct node *createpoly(struct node *start)
{ struct node *ptr, *ptr1, *avail, *fresh;
  char ch;
  do
   {
    avail = (struct node *) malloc(sizeof(struct node));
    if(avail == NULL)
     { printf("overflow");
     }
    else
     { fresh = avail;
       avail = avail -> link;
       fresh -> link = NULL;
       printf("\n enter co-efficient & exponent :");
       scanf("%d %d", &fresh->co, &fresh->exp);
       if(start == NULL)
            start = fresh;
       else
          { ptr1 = start;
            while(ptr1 != NULL && ptr1->expy = fresh->exp)
             { ptr = ptr1;
               ptr1 = ptr1 -> link;
             }
```


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

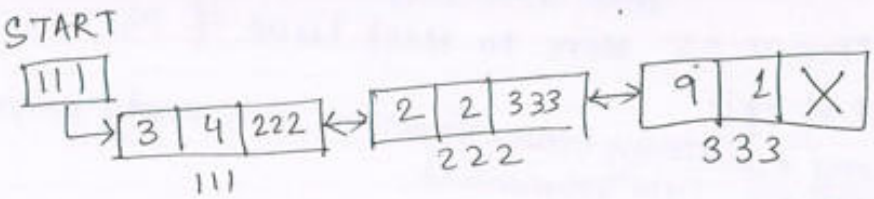
```
if (ptr1 == start)
{
   fresh -> link = ptr1;
    start = fresh;
}
else if (ptr1 == NULL)
{ ptr -> link = fresh;
}
else
{  ptr -> link = fresh;
   fresh -> link = ptr1;
}
}
printf("add more (y/n) ?");
fflush(stdin);
scanf("%c", &ch);
}
while (ch == 'y');
return start;
}
```

GIFT CENTRAL LIBRARY BHUBANESWAR

Polynomial Addition:

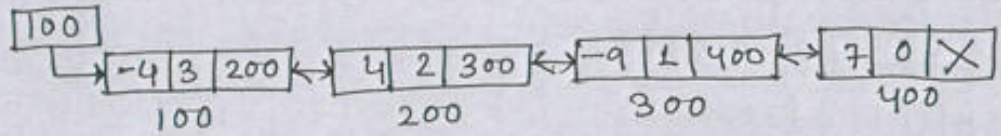
1. Assume P1, P2 are two polynomial expressions given. P3 is the resultant polynomial which represents sum of P1 and P2

$$P_1 = 3x^4 + 2x^2 + 9x$$
$$P_2 = -4x^3 + 4x^2 - 9x + 7$$
$$P_3 = 3x^4 + (-4x^2) + 6x^2 + 7.$$

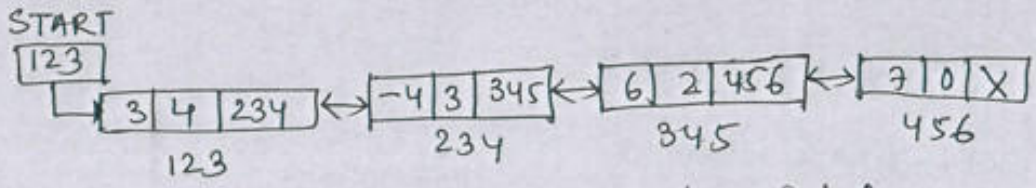


Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

START



START



Procedure for addition of two polynomials :
[Assume P1 and P2 are two polynomial expressions and P3 is the resultant polynomial]

Step 1 : Repeat while P1 and P2 are Not NULL Repeat step 2,3 and 4.

Step 2 : If exponent parts of two terms of P1 and P2 are equal and if the co-efficient terms do not cancel to 0, then

Step 2.1 : Find sum of the co-efficient terms and insert SUM polynomial P3

Step 2.2 : Move to next term of P1

Step 2.3 : Move to next term of P2.

Step 3 : Else if the exponent of the term in first polynomial > exponent of the term in second polynomial, then insert the term from first polynomial in the SUM polynomial.

Step 3.1 : Move to next term of P1.

Step 4 : Else

Step 4.1 : Insert the term from the 2nd polynomial into SUM polynomial

Step 4.2 : Move to Next term of P2.

[End of If]

Step 5 : Copy remaining terms from the non-empty polynomials into the SUM polynomial.

Step 6 : Exit.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Techie Work | Talented Colleagues | Cool Workspace
High Salary | Helpful Leadership
mindfire BURN IGNORANCE

Algorithm for polynomial addition:

GET_NODE (AVAIL): This procedure used to allocate memory for a new node

INSERT_AT_END (P3, FRESH): This procedure used to insert FRESH node at the end of linked list.

ADD_POLY (P1, P2, P3): Here P1 identifies first terms of 1st polynomial, P2 identifies first term of 2nd polynomial and P3 identifies first term of the resultant polynomial.

GET_NODE (AVAIL)

Step 1: Let PTR
Step 2: If AVAIL = NULL, then Display "Overflow"
Step 3: Else
PTR := AVAIL
AVAIL := LINK[AVAIL]
LINK[PTR] := NULL
[End of If]
Step 4: RETURN (PTR)
Step 5: Exit

INSERT_AT_END (P3, FRESH)

Step 1: Let PTR
Step 2: If P3 = NULL, then P3 := FRESH
Step 3: Else
Step 3.1: Set PTR := P3
Step 3.2: Repeat while LINK[PTR] != NULL
PTR := LINK[PTR]
[End of while]
Step 3.3: LINK[PTR] := FRESH
Step 4: Exit

ADD_POLY (P1, P2, P3)

Step 1: Let PTR1, PTR2, FRESH
Step 2: P3 := NULL
Step 3: PTR1 := P1
Step 4: PTR2 := P2

GIFT CENTRAL LIBRARY BHUBANESWAR

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

```
Step 5: Repeat while PTR1 != NULL AND PTR2 != NULL
  Step 5.1: If EXP[PTR1] > EXP[PTR2], then
              FRESH := GET-NODE(AVAIL)
              EXP[FRESH] := EXP[PTR1]
              CO[FRESH] := CO[PTR1]
              INSERT_AT_END(P3, FRESH)
              PTR1 := LINK[PTR1]
  Step 5.2: Else If EXP[PTR2] > EXP[PTR1], then
              FRESH := GET_NODE(AVAIL)
              EXP[FRESH] := EXP[PTR2]
              CO[FRESH] := CO[PTR2]
              INSERT-AT_END(P3, FRESH)
              PTR2 := LINK[PTR2]
  Step 5.3: Else
              FRESH := GET_NODE(AVAIL)
              EXP[FRESH] := EXP[PTR1]
              CO[FRESH] := CO[PTR1] + CO[PTR2]
      Step5.3.1: If CO[FRESH] = 0, then
                   INSERT_AT_END(P3, FRESH)
          [End of If - step 5.3.1]
        PTR1 := LINK[PTR2]
        PTR2 := LINK[PTR2]
      [END of If - step 5.1]
  [End of while - step 5]
Step 6: Repeat while PTR2 != NULL
            FRESH := GET-NODE(AVAIL)
            EXP[FRESH] := EXP[PTR2]
            CO[FRESH] := CO[PTR2]
            INSERT_AT-END (P3, FRESH)
            PTR2 := LINK[PTR2]
        [End of while - step 6]

Step 7: Repeat While PTR1 != NULL
            FRESH := GET-NODE(AVAIL)
            EXP[FRESH] := EXP[PTR1]
            CO[FRESH] := CO[PTR1]
            INSERT - AT-END (P3, FRESH)
            PTR1 := LINK[PTR1]
        [End of while - step 7]
Step 8: RETURN(P3)
Step 9: Exit.
```

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.

Data Structure Using C

Topic:

Dynamic Storage Management

Contributed By:

Nihar Ranjan Rout

Gandhi Institute For Technology, GIFT

Dynamic Storage Management:

1. Basic task of any program is to manipulate data. These data should be stored in memory during their manipulation. There are two memory management schemes for the storage allocations of data:-
   i) Static storage Management
   ii) Dynamic storage Management.

G I E T CENTRAL LIBRARY BHUBANESWAR

2. In case of static storage management scheme, the net amount of memory for various data for a program are allocated before the starting of the execution of the program. Once memory is allocated, it neither can be extended nor can be returned to the memory bank for the use of other programs at the same time.
3. The dynamic storage management scheme allows the user to allocate and reallocate memory as per the necessity during the execution of programs.
4. Principles of dynamic memory management scheme :-
   i) Allocation scheme: A request for a memory block will be serviced. There are two strategies, i.e:- a) fixed block allocation and b) variable block allocation (First fit and its variant, Next fit, Best fit, Worst fit)
   ii) Reallocation scheme: How to return memory block to the memory bank whenever it is no more required. That is:- a) Random reallocation and b) order Reallocation.

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Garbage Collection:

1. Suppose some memory space becomes reusable, a node is deleted from a list or an entire list is deleted from a program, the operating system of a computer may periodically collect all the deleted space onto the free storage list.
2. Any technique which does this collection is called Garbage collection. Garbage collection usually takesplace in two steps. That is:-
   i) First the computer runs through all list, tagging those cells which are currently in use.
   ii) Then the computer runs through the memory collecting all untagged space into the free storage list
3. The Garbage collection may takesplace when there is only some minimum amount of space or no space at all left in free storage list or when the CPU has the time to do the collection.

Compaction:

1. It is a technique for reclaiming the memory is unused for longer period by introducing a program to accomplish this task. The allocation problem becomes simple after compaction process.
2. Compaction works by actually moving one block of data from one location of memory to another. So, as to collect all the free blocks into one large block.

—x—

Downloaded from www.LectureNotes.in. Contributed by Nihar Ranjan Rout

Mindfire is a software service provider,
with unrelenting focus on optimal software development
and delivery.