PNS SCHOOL OF ENGINEERING & TECHNOLOGY

Nishamani Vihar, Marshaghai, Kendrapara

LECTURE NOTES

ON

DATA STRUCTURE

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

3RD SEMESTER

PREPARED BY

MR. BISWARANJAN SWAIN

LECTURER IN COMPUTER SCIENCE & ENGINEERING

UNIT-I

INTRODUCTION

Data

The term ***Data*** is defined as a raw and unstructured fact that needs to be processed to make it meaningful. Data can be simple and unstructured at the same time until it is structured. Usually data contains facts, numbers, symbols, image, observations, perceptions, characters, etc.

Information

The term ***Information*** is defined as a set of data that is processed according to the given requirement in a meaningful way. To make the information useful and meaningful, it must be processed, presented and structured in a given context. Information is processed from data and possess context, purpose and relevance.

Data Type:

Data types are used within type systems, which offer different ways of defining, implementing and using the data. Different languages may use different terminology. Common data types are :

Integers,
Booleans,
Characters,
Floating-point numbers,
Alphanumeric strings.

Classes of data types

There are different classes of data types as given below.

1. Primitive data type
2. Composite data type
3. En-numerated data type
4. Abstract data type
5. Utility data type
6. Other data type

1. Primitive data types:

All data in computers based on digital electronics is represented as bits (alternatives 0 and 1) on the lowest level. The smallest addressable unit of data is usually a group of bits called a byte (usually an octet, which is 8 bits). The unit processed by machine code instructions is called a word (as of 2011, typically 32 or 64 bits). Most instructions interpret the word as a binary number, such that a 32-bit word can represent unsigned integer values from 0 to $2^{32}-1$ or signed integer values from $-2^{31}$to $2^{31}-1$.

Boolean type :

The Boolean type represents the values true and false. Many programming languages do not have an explicit boolean type, instead interpreting (for instance) 0 as false and other values as true.

Numeric types such as:

The integer data types, or "whole numbers" may be subtype according to their ability to contain negative values (e.g. unsigned in C and C++).

Floating point data types, contain fractional values. They usually have predefined limits on both their maximum values and their precision. These are often represented as decimal numbers.

2. Composite / Derived data types :

Composite types are derived from more than one primitive type. This can be done in a number of ways. The ways they are combined are called data structures. Composing a primitive type into a compound type generally results in a new type, e.g. array-of-integer is a different type to integer.

3. Enumerated Type :

This has values which are different from each other, and which can be compared and assigned, but which do not necessarily have any particular concrete representation in the computer's memory; compilers and interpreters can
represent them arbitrarily.

String and text types such as:
Alphanumeric character. A letter of the alphabet, digit, blank space, punctuation mark, etc.
Alphanumeric strings, a sequence of characters. They are typically used to represent words and text.

Character and string types can store sequences of characters from a character set such as ASCII. Since most character sets include the digits, it is possible to have a numeric string, such as "1234".

4. Abstract data types

Any type that does not specify an implementation is an abstract data type. For instance, a stack (which is an abstract type) can be implemented as an array (a contiguous block of memory containing multiple values), or as a linked list (a set of non-contiguous memory blocks linked by pointers).
Examples include:
- A queue is a first-in first-out list. Variations are Deque and Priority queue.
- A set can store certain values, without any particular order, and with no repeated values.
- A stack is a last-in, first out.
- A tree is a hierarchical structure.
- A graph.

Data structure

In Computer Science, data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently. Data structures determine the way in which information can be stored in computer and used. Finding the best data structure when solving a problem is an important part of programming. Programs that use the right data structure are easier to write, and work better.

Data Structure Operations:

The various operations that can be performed on different data structures are as follows:
1. Create– A data structure created from data.
2. Traverse – Processing each element in the list
3. Searching – Finding the location of given element.
4. Insertion – Adding a new element to the list.
5. Deletion – Removing an element from the list.
6. Sorting – Arranging the records either in ascending or descending order.
7. Merging – Combining two lists into a single list.
8. Modifying – the values of DS can be modified by replacing old values with new ones.
9. Copying – records of one file can be copied to another file.
10. Concatenating – Records of a file are appended at the end of another file.
11. Splitting – Records of big file can be splitting into smaller files.

Types of Data Structure :

Basically, data structures are divided into two categories:
- Linear data structure
- Non-linear data structure

Linear data structures:

In linear data structures, the elements are arranged in sequence one after the other. Since elements are arranged in particular order, they are easy to implement. However, when the complexity of the program increases, the linear data structures might not be the best choice because of operational complexities.
Popular linear data structures are:
1. Array Data Structure

In an array, elements in memory are arranged in continuous memory. All the elements of an array are of the same type. And, the type of elements that can be stored in the form of arrays is determined by the programming language.
2. Stack Data Structure

In stack data structure, elements are stored in the LIFO principle. That is, the last element stored in a stack will be removed first.
It works just like a pile of plates where the last plate kept on the pile will be removed first.

3. Queue Data Structure

Unlike stack, the queue data structure works in the FIFO principle where first element stored in the queue will be removed first. It works just like a queue of people in the ticket counter where first person on the queue will get the ticket first.

4. Linked List Data Structure

In linked list data structure, data elements are connected through a series of nodes. And, each node contains the data items and address to the next node.

Non linear data structures

Unlike linear data structures, elements in non-linear data structures are not in any sequence. Instead they are arranged in a hierarchical manner where one element will be connected to one or more elements.

Non-linear data structures are further divided into graph and tree based data structures.

1. Graph Data Structure

In graph data structure, each node is called vertex and each vertex is connected to other vertices through edges.

2. Trees Data Structure

Similar to a graph, a tree is also a collection of vertices and edges. However, in tree data structure, there can only be one edge between two vertices.

Abstract data type:

Abstract data types are like user-defined data types, which define the operations on the values using functions without specifying what is inside the function and how the operations are performed. It is a logical description of how we view the data and the operations that are allowed without knowing how they will be implemented. For example, an abstract stack could be defined by three operations:

1. push, that inserts some data item onto the structure,
2. pop, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and peek, that allows data on top of the structure to be examined without removal.

Abstract data types are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages.

Abstract Data Type in Computer Programming

In the course an abstract data type refers to a generalized data structure that accepts data objects stored as a list with specific behaviors defined by the methods associated with the underlying nature of the list.

Some common ADTs, are –

- Container
- Deque
- List
- Map
- Multimap
- Multiset
- Priority queue
- Queue
- Set
- Stack
- Tree
- Graph

Algorithm:

An algorithm is a step-by-step procedure for calculations. Algorithms are used for calculation, data processing, and automated reasoning. An algorithm is an effective method expressed as a finite list of well-defined instructions for calculating a function. Starting from an initial state and initial input (perhaps empty), the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input.

Complexity of Algorithm and Time, Space tradeoff:

The algorithms are evaluated by the determination of the amount of resources (such as time and storage) necessary to execute them. Most algorithms are designed to work with inputs of arbitrary length. Usually, the
efficiency or running time of an algorithm is stated as a function relating the input length to the number of steps (time complexity) or storage locations (space complexity).

Best, worst and average case complexity:

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size n may be faster to solve than others, we define the following complexities:

- Best-case complexity: This is the complexity of solving the problem for the best input of size n.
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size n.
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are
assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size n.

Time Complexity

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the string representing the input. The time complexity of an algorithm is commonly expressed using big O notation, which excludes coefficients and lower order terms. When expressed this way, the time complexity is said to be described asymptotically, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is $O(n^3)$.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Since an algorithm's performance time may vary with different inputs of the same size, one commonly uses the worst-case time complexity of an algorithm, denoted as T(n), which is defined as the maximum amount of time taken on any
input of size n. Time complexities are classified by the nature of the function T(n).

An algorithm with T(n) = O(n) is called a linear time algorithm, and an algorithm with $T(n) = O(2^n)$ is said to be an exponential time algorithm.

Space complexity

The way in which the amount of storage space required by an algorithm varies with the size of the problem it is solving. Space complexity is normally expressed as an order of magnitude, e.g. O(N^2) means that if the size of the problem (N) doubles then four times as much working storage will be needed.

Question Set:

1. What is ADT and explain ADT with example?
2. Define time complexity and space complexity.
3. Why it is necessary to find time and space complexity of an algorithm?
4. What are the possible operation on data structure?
5. Describe different type of data structure with example.

UNIT-III

ARRAYS

Definition :

• Arrays are defined as the collection of similar type of data items stored at contiguous memory locations.
• Arrays are the derived data type in C programming language which can store the primitive type of data such as int, char, double, float, etc.
• Array is the simplest data structure where each data element can be randomly accessed by using its index number.

For example, if we want to store the marks of a student in 6 subjects, then we don't need to define different variable for the marks in different subject, instead of that, we can define an array which can store the marks in each subject at contiguous memory locations.

The array marks[10] defines the marks of the student in 10 different subjects where each subject marks are located at a particular subscript in the array i.e. marks[0] denotes the marks in first subject, marks[1] denotes the marks in 2nd subject and so on
Array is a list of finite number of n homogeneous data elements i.e. the elements of same data types Such that:

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

• The elements of the array are stored respectively in the successive memory locations.
The number n of elements is called length or size of array. If not explicitly stated, we will assume the index set consists of integers 1, 2, 3 …n. In general the length or the number of data elements of the array can be obtained from the index set by the formula

Length= UB – LB + 1

Where UB is the largest index, called the upper bound, and LB is the smallest index,

called the lower bound. Note that length = UB when LB = 1.

The elements of an array A are denoted by subscript notation à a1, a2, a3…an Or by the

parenthesis notation -> A (1), A (2),…., A(n)

Or by the bracket notation -> A[1], A[2],….,A[n].

We will usually use the subscript notation or the bracket notation.

REPRESENTATION OF LINEAR ARRAYS IN MEMORY:

Let LA is a linear array in the memory of the computer. The memory of computer is simply a sequence of addressed locations.
LOC (LA[k]) = address of element LA[k] of the array LA.
As previously noted, the elements of LA are stored in the successive memory cells.
Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by Base (LA)
And called the base address of LA.

Using base address the computer calculates the
address of any element of LA by the following formula:

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

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

OPERATIONS ON ARRAYS :

Various operations that can be performed on an array

- Traversing
- Insertion
- Deletion
- Sorting
- Searching
- Merging

TRAVERSING LINEAR ARRAY:

Let A be a collection of data elements stored in the memory of the computer.

Suppose we want to print the content of each element of A or suppose we want to count the number of elements of A, this can be accomplished by traversing A, that is, by accessing and processing each element of a exactly ones.

The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure. Other linear structures, such as linked list, can also be easily traversed. On the other hand, traversal of nonlinear structures, such as trees and graph, is considerably more complicated.

Algorithm: (Traversing a Linear Array)

Here LA is a linear array with lower bound LB and upper bound UB.

This algorithm traverses LA applying an operation PROCESS to each element of LA.

1. [Initialize counter] Set k: =LB.
2. Repeat steps 3 and 4 while k <=UB.
3. [Visit Element] Apply PROCESS to LA [k].
4. [Increase Counter] Set k: =k + 1. [End of step 2 loop]
5. Exit.

INSERTION AND DELETION IN LINEAR ARRAY:

Let A be a collection of data elements in the memory of the computer.

Inserting refers to the operation of adding another element to the collection A

Deleting refers to the operation of removing one element from A.

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 accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new location to accommodate the new elements and keep the order of the other elements.

Similarly, deleting an element at the end of the array presents no difficulties, but deleting the element somewhere in the middle of the array requires that each subsequent element be moved one location upward in order to fill up the array.

The following algorithm inserts a data element ITEM in to the Kth position in the linear array LA with N elements.

Algorithm for Insertion: (Inserting into Linear Array)

INSERT (LA, N, K, ITEM)

Here LA is a linear array with N elements and K is a positive integer such that K<=N.
The algorithm inserts an element ITEM into the Kth position in LA.
1. [Initialize counter] Set J: = N.
2. Repeat Steps 3 and 4 while j >= k;
3. [Move jth element downward.] Set LA [J + 1]: =LA [J].
4. [Decrease counter]
Set J: = J-1 [End of step 2 loop]
5. [Insert element] Set LA [K]:=ITEM.
6. [Reset N] Set N:=N+1
7. EXIT.

The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

Algorithm for Deletion: (Deletion from a Linear Array)

DELETE (LA, N, K, ITEM)

Here LA is a Linear Array with N elements and K is the positive integer
such that K<=N. This algorithm deletes the Kth element from LA.
1. Set ITEM: = LA [k].
2. Repeat for J = K to N – 1.
[Move J + 1^{st} element upward] Set LA [J]: = LA [J +1].
[End of loop]
3. [Reset the number N of elements in LA] Set N: = N-1
4. EXIT

MULTIDIMENSIONAL ARRAY:

1. Array having more than one subscript variable is called Multi- Dimensional array.
2. Multi Dimensional Array is also called as Matrix.

Consider the Two dimensional array -
1. Two Dimensional Array requires Two Subscript Variables
2. Two Dimensional Array stores the values in the form of matrix.
3. One Subscript Variable denotes the —Row of a matrix.
4. Another Subscript Variable denotes the —Column of a matrix.

Declaration and Use of Two Dimensional Array :

```
int a[3][4];
```

Use :

```
for(i=0;i<row;i++)
{
     for(j=0;j<col;j++)
    {
          printf("%d",a[i][j]);
    }
}
```

Meaning of Two Dimensional Array :

1. Matrix is having 3 rows (i takes value from 0 to 2)
2. Matrix is having 4 Columns (j takes value from 0 to 3)
3. Above Matrix 3×4 matrix will have 12 blocks having 3 rows & 4 columns.
4. Name of 2-D array is _a_ and each block is identified by the row & column Number.
5. Row number and Column Number Starts from 0.

Cell Location Meaning

a[0][0] 0th Row and 0th Column
a[0][1] 0th Row and 1st Column
a[0][2] 0th Row and 2nd Column
a[0][3] 0th Row and 3rd Column
a[1][0] 1st Row and 0th Column
a[1][1] 1st Row and 1st Column
a[1][2] 1st Row and 2nd Column
a[1][3] 1st Row and 3rd Column
a[2][0] 2nd Row and 0th Column
a[2][1] 2nd Row and 1st Column
a[2][2] 2nd Row and 2nd Column
a[2][3] 2nd Row and 3rd Column

Two-Dimensional Array : Summary with Sample Example
Summary Point Explanation
No of Subscript Variables Required 2
Declaration a[3][4]
No of Rows 3
No of Columns 4
No of Cells 12
No of for loops required to iterate 2

MEMORY REPRESENTATION :

1. 2-D arrays are Stored in contiguous memory location row wise.
2. X 3 Array is shown below in the first Diagram.
3. Consider 3×3 Array is stored in Contiguous memory location which starts from 4000 .
4. Array element a[0][0] will be stored at address 4000 again a[0][1] will be stored to next memory location i.e Elements stored row-wise
5. After Elements of First Row are stored in appropriate memory location , elements of next row get their corresponding memory locations.
6. This is integer array so each element requires 2 bytes of memory.

Array Representation:

- Column-major
- Row-major

Arrays may be represented in Row-major form or Column-major form.

In **Row- major** form, all the elements of the first row are printed, then the elements of the second row and so on up to the last row.

In **Column-major** form, all the elements of the first column are printed, then the elements of the second column and so on up to the last column.

The ‗C' program to input an array of order m x n and print the array contents in row major and column major is given below. The following array elements may be entered during run time to test this program:
Output:
Row Major:

1 2 3

4 5 6

7 8 9

Column Major:

1 4 7

2 5 8

3 6 9

Basic Memory Address Calculation :

a[0][1] = a[0][0] + Size of Data Type
Element Memory Location
a[0][0] 4000
a[0][1] 4002
a[0][2] 4004
a[1][0] 4006
a[1][1] 4008
a[1][2] 4010
a[2][0] 4012
a[2][1] 4014
a[2][2] 4016

Array and Row Major, Column Major order arrangement of 2 d array :

An array is a list of a finite number of homogeneous data elements. The number of elements in an array is called the array length. Array length can be obtained from the index set by the formula

Length=UB-LB+1

Where UB is the largest index , called the upper bound and LB is the smallest index , called the lower bound. Suppose int Arr[10] is an integer array. Upper bound of this array is 9 and lower bound of this array is 0 ,so the length is 9- 0+1=10.

In an array ,the elements are stored successive memory cells. Computer does not need to keep track of the address of every elements in memory. It will keep the address of the first location only and that is known as base address of an array.

Using the base address , address of any other location of an array can be calculated by the computer. Suppose Arr is an array whose base address is Base(Arr) and w is the number of memory cells required by each elements of the array. The address of Arr[k] – k being the index value can be obtained by Using the formula

Address(Arr[k])=Base(Arr)+w(k-LowerBound)

2 d Array :- Suppose Arr is a 2 d array. The first dimension of Arr contains the index set 0,1,2, … row-1 (the lower bound is 0 and the upper bound is row-1) and the second dimension contains the index set 0,1,2,… col-1(with lower bound 0 and upper bound col-1.)

The length of each dimension is to be calculated .The multiplied result of both the lengths will give you the number of elements in the array.

Let's assume Arr is an two dimensional 2 X 2 array .The array may be stored in memory one of the following way :-

- Column by columni,e column major order
- Row by row , i,e in row major order. The following figure shows both representation of the above array.

By row-major order, we mean that the elements in the array are so arranged that the subscript at the extreme right varies fast than the subscript at it's left., while in column-major order , the subscript at the extreme left changes rapidly ,Then the subscript at it's right and so on.

1,1

2,1

1,2

2,2

Column Major Order

1,1

1,2

2,1

2,2

Row major order

Now we know that computer keeps track of only the base address. So the address of any specified location of an array , for example Arr[j,k] of a 2 d array Arr[m,n] can be calculated by using the following formula :- (Column major order)

Address(Arr[j,k])= base(Arr)+w[m(k-1)+(j-1)]

(Row major order)

Address(Arr[j,k])=base(Arr)+w[n(j-1)+(k-1)]

For example

Arr(25,4) is an array with base value 200.w=4 for this array. The address of Arr(12,3) can be calculated using row-major order as

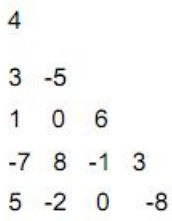
Address(Arr(12,3))=200+4[4(12-1)+(3-1)]
=200+4[4*11+2]
=200+4[44+2]
=200+4[46]
=200+184
=384

Again using column-major order

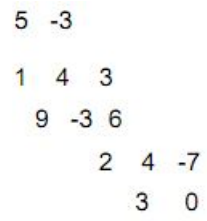
Address(Arr(12,3))=200+4[25(3-1)+(12-1)]
=200+4[25*2+11]
=200+4[50+11]
=200+4[61]
=200+244
=444

SPARSE MATRIX :

Matrix with relatively a high proportion of zero entries are called sparse matrix. Two general types of n-square sparse matrices are there which occur in various applications are mention in figure below(It is sometimes customary to omit blocks of zeros in a matrix as shown in figure below)



Triangular matrix



Tridiagonal matrix

Triangular matrix

This is the matrix where all the entries above the main diagonal are zero or equivalently where non-zero entries can only occur on or below the main diagonal is called a (lower)Triangular matrix.

Tridiagonal matrix

This is the matrix where non-zero entries can only occur on the diagonal or on elements immediately above or below the diagonal is called a Tridiagonal matrix. The natural method of representing matrices in memory as two-dimensional arrays may not be suitable for sparse matrices i.e. one may save space by storing only those entries which may be non-zero.

UNIT-2
STRING

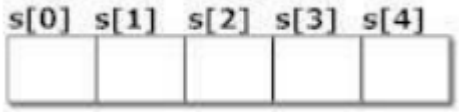
- A string in C is a array of character.
- It is a one dimensional array type of char.
- Every string is terminated by null character('\0') .

 The predefined functions gets() and puts() in C language to read and display string respectively.

Declaration of strings:

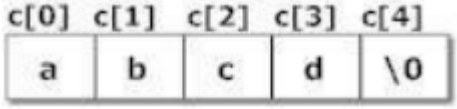
Strings are declared in C in similar manner as arrays. Only difference is that, strings are of char type.

Syntax:
char str_name[str_size];
Example:
char s[5];



Example:
char s[5];

Initialization of strings
char c[]="abcd";



String handling functions

- Many library function are defined under header file <string.h> to perform different tasks.
- Diffrent user defined functios are:
- Strlen()
- Strcpy()
- Strcmp()
- Strcat()

Strlen():

- The strlen( ) function is used to calculate the length of the string.
- It means that it counts the total number of characters present in the string which includes alphabets, numbers, and all special characters including blank spaces.

Strlen():

- The strlen( ) function is used to calculate the length of the string.
- It means that it counts the total number of characters present in the string which includes alphabets, numbers, and all special characters including blank spaces.

Example:

```
char str[] = "Learn C Online";
int strLength;
strLength = strlen(str);  //strLength contains the length of the string i.e. 14
```

Strcpy()

• strcpy function copies a string from a source location to a destination location and provides a null character to terminate the string.

Syntax:
strcpy(Destination_String,Source_String);
Example:
char *Destination_String;
char *Source_String = "Learn C Online";

strcpy(Destination_String,Source_String);
printf("%s", Destination_String);
Output:
Learn C Online

Strcmp()

- Strcmp() in C programming language is used to compare two strings.
- If both the strings are equal then it gives the result as zero but if not then it gives the numeric difference between the first non matching characters in the strings.

Syntax:
int strcmp(string1, string2);
Example:
char *string1 = "Learn C Online";
char *string2 = "Learn C Online";
int ret;
ret=strcmp(string1, string2);
printf("%d",ret);
Output:
0

Strcat()

- The strcat() function is used for string concatenation in C programming language. It means it joins the two strings together Syntax:

strncat(Destination_String, Source_String,no_of_characters);

Example:
char *Destination_String="Visit ";
char *Source_String = "Learn C Online is a great site";
strncat(Destination_String, Source_String,14);
puts(Destination_String);
Output:
Visit Learn C Online

Program to check wheaher a word is polyndrome or not

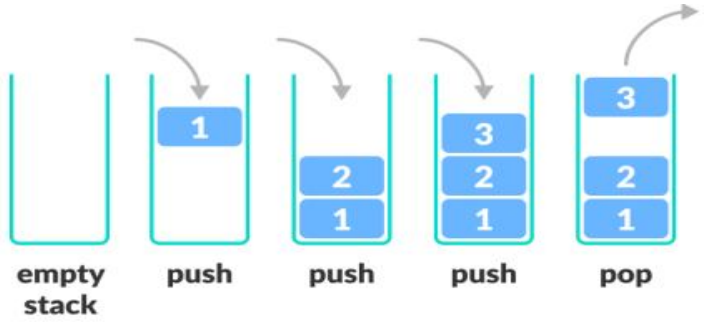
```
#include<stdio.h>
#include<string.h>
void main()
{
char s1[10], s2[10];
int x;
gets(s1);
strcpy(s2,s1);
strrev(s1);
x=strcmp(s1,s2);
if(x==0)
printf("pallindrome");
else
printf("not pallindrome");
getch();}
```

UNIT-4
STACKS & QUEUES

STACK

 Stack is a linear data structure in which an element may be inserted or deleted at one end called TOP of the stack.

 That means the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.



Stack Push and Pop Operations

 Stack is called LIFO (Last-in-first-Out) Str. i.e. the item just inserted isdeleted first.

 There are 2 base operations associated with stack

1. Push :- This operation is used to insert an element into stack.
2. Pop :- This operation is used to delete an element from stack.

Condition also arise :

1. Overflow :- When a stack is full and we are attempting a push operation , overflow condition arises.
2. Underflow :- When a stack is empty , and we are attempting a pop operation then underflow condition arises.

Representation of Stack in Memory :

A stack may be represented in the memory in two ways:

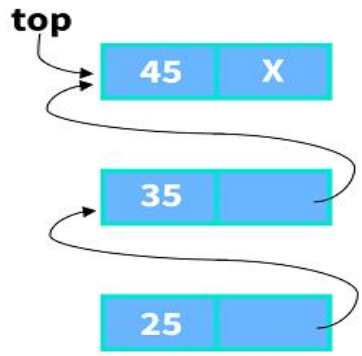
1. Using one dimensional array i.e. Array representation of Stack.
2. Using single linked list i.e. Linked list representation of stack.

Array Representation of Stack :

To implement a stack in memory, we need a pointer variable called TOP that hold the index of the top element of the stack, a linear array to hold the elements of the stack and a variable MAXSTK which contain the size of the stack.

Linked List Representation of Stack :

 Array representation of Stack is very easy and convenient but it allows only to represent fixed sized stack.



 But in several application size of the stack may very during program execution, at that cases we represent a stack using linked list.

 Single linked list is sufficient to represent any Stack.

 Here the 'info' field for the item and 'next' field is used to print the next item.

INSERTION IN STACK (PUSH OPERATION)

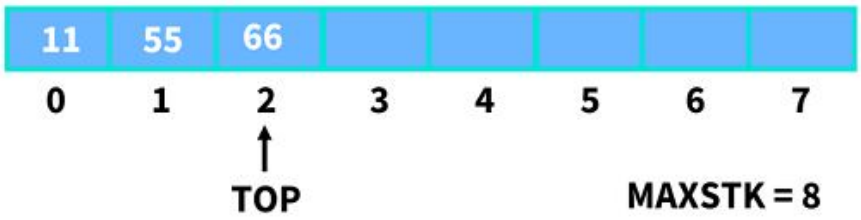
This operation is used to insert an element in stack at the TOP of the stack.

Algorithm :-

PUSH (STACK, TOP, MAXSTK, ITEM)

This procedure pushes an item into stack.

1. If TOP = MAXSTK then print Overflow and Return.
2. Set TOP = TOP+1 (Increase TOP by 1)

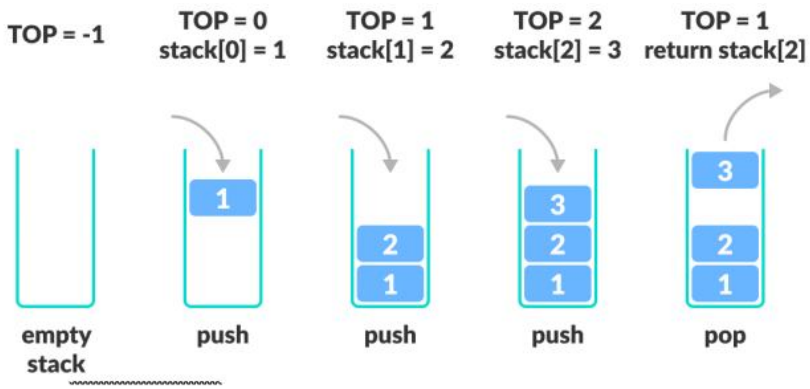


3. Set STACK[TOP] = ITEM (Inserts ITEM in new TOP position)
4. Return

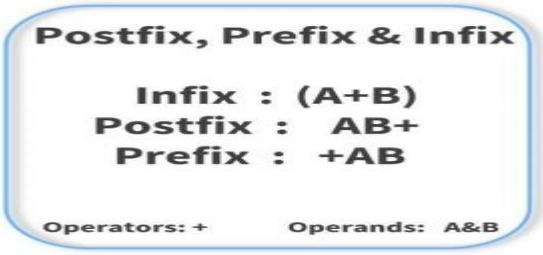
(Where Stack = Stack is a list of linear structure.
TOP = printer variable which contain the location of TOP
element of the stack.
MAXSTK = A variable which contain size of stack
ITEM = Item to be inserted)

DELETION IN STACK (POP OPERATION)

This operation is used to delete an element from stack.



Algorithm :-
POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.
1. If TOP = 0 the print Underflow and Return.
2. Set ITEM = STACK[TOP] (Assigns TOP element to ITEM)
3. Set TOP = TOP-1 (Decreases TOP by 1)
Working of Stack Data Structure
5. Return



ARITHMETIC EXPRESSION
There are 3 notation to represent an arithmetic expression.
1. Infix notation
2. Prefix notation
3. Postfix notation

INFIX NOTATION
The conventional way of writing an expression is called as infix.
Ex: A+B, C+D, E*F, G/M etc.
Here the notation is
<Operand><Operator><Operand>
This is called infix because the operators come in between the operands.

PREFIX NOTATION
This notation is also known as “POLISH NOTATION”
Here the notation is
<Operator><Operand><Operand>
Ex: +AB, -CD, *EF, /GH
POSTFIX NOTATION
This notation is called as postfix or suffix notation where operator is suffixed by operand.
Here the notation is
<Operand ><Operand><Operator>
Ex: AB+, CD-, EF*, GH/
This notation is also known as “ REVERSE POLISH NOTATION.”
CONVERSION FROM INFIX TO POSTFIX EXPRESSION

Algorithm:
POLISH(Q,P)
Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.
1. Push “ (” onto stack and add “) ” to the end of Q.
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty.
3. If an operand is encountered, add it top.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator X is encountered, then:

a) Repeatedly POP from STACK and add to P each operator (on the top of the STACK) which has the same precedence as or higher precedence than the operator X .
b) Add the operator X to STACK.
6. If a right parenthesis is encountered then:
a) Repeatedly POP from STACK and add to P each operator (on the top of the STACK) until a left parenthesis is encountered.
b) Remove the left parenthesis.
[End of if str.]
[End of Step-2 Loop]
7. Exit

Ex: A+(B+C – (D/E+F)*G)*H)

Symbol Scanned	STACK	EXPRESSION (POSTFIX)
	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
C	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+*	ABC*DEF^/G*-
H	(+*	ABC*DEF^/G*-H
)		ABC*DEF^/G*-H*+

Equivalent Postfix Expression ->ABC*DEF^/G*-H*+

Algorithm:
EVALUATION OF POSTFIX EXPRESSION
This algorithm finds the value of an arithmetic expression P written in Postfix notation.
1. Add a right parenthesis ") " at the end of P.
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the ") " is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator X is encountered then:
a) Remove the two top element of STACK, where A is top element and B is the next-to-top element.
b) Evaluate B X A.

c) Place the result of (b) on STACK.
[End of if str.]
5. Set value equal to the top element on STACK.
6. Exit

Ex: 5, 6, 2, +, *, 12, 4, /, -)

Symbol Scanned	STACK
5	5
6	5,6
2	5,6,2
+	5,8
*	40
12	40,12
4	40,12,4
/	40,3
-	37 (Result)
)	

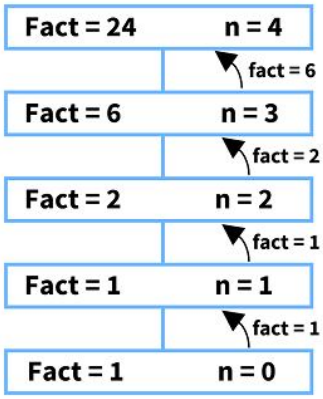
APPLICATIONS OF STACK:

- Recursion process uses stack for its implementation.
- Quick sort uses stack for sorting the elements.
- Evaluation of antithetic expression can be sone by using STACK.
- Conversion from infix to postfix expression
- Evaluation of postfix expression
- Conversion from infix to prefix expression
- Evaluation of prefix expression.
- Backtracking
- Keeptrack of post-visited (history of a web- browsing)

IMPLEMENTATION OF RECURSION

A function is said to be Recursive function if it call itself or it call a function such that the function call back the original function.

This concept is called as Recursion.
The Recursive function has two properties.
I. The function should have a base condition.
II. The function should come closer towards the base condition.
The following is one of the example of recursive function which is described below.



Factorial of a no. using Recursion

The factorial of a no. ‘n’ is the product of positive integers from 1 to n.

n ! = 1 X 2 X 3 X . . . . . .X (n-) X n

Physically proved n ! = n X (n-1) !

The factorial function can be defined as follows.

I. If n=0 then n ! = 1

II. If n>0 then n ! = n.(n-1) !

The factorial algorithm is given below factorial (fact , n)

This procedure calculates n! and returns the value in the variable fact.

1. If n=0 then fact=1 and Return.
2. Call factorial (fact, n-1)
3. Set fact = fact*n
4. Return

Ex: Calculate the factorial of 4.

4 ! = 4 x 3 !

3 ! = 3 x 2 !

1 ! = 1 x 1 = 1

2 ! = 2 x 1 = 2

3 ! = 3 x 2 = 6

4 ! = 4 x 6 = 24

2 ! = 2 x 1 !

1 ! = 1 x 0 !

0 ! = 1

UNIT-5

QUEUE

 Queue is a linear data structure or sequential data structure where insertion take place at one end and deletion take place at the other end.
 The insertion end of Queue is called rear end and deletion end is called front end.
 Queue is based on (FIFO) First in First Out Concept that means the node i.e. added first is processed first.

 HereEnqueue is Insert Operation.
FIFO Representation of Queue
Dequeue is nothing but Delete Operation.
Types of Queue:

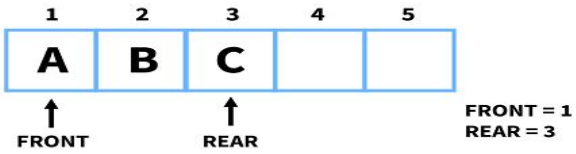
1. General Queue
2. Circular Queue
3. Deque
4. Priority Queue
5.

REPRESENTATION OF QUEUE IN MEMORY

 The Queue is represented by a Linear array “Q” and two pointer variable FRONT and REAR.
 FRONT gives the location of element to be deleted and REAR gives the location after which the element will be inserted.
 The deletion will be done by setting Front = Front + 1
 The insertion will be done by setting Rear = Rear + 1

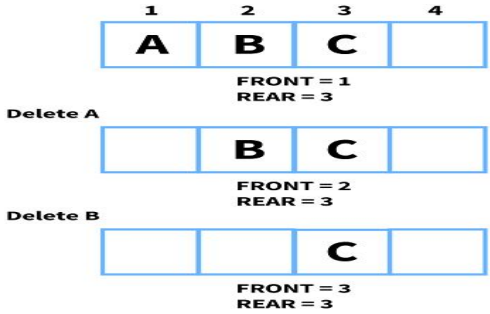
INSERTION IN QUEUE(Enqueue):

Ex:



Algorithm:
Insert (Q, ITEM, Front, Rear)
This procedure insert ITEM in queue Q.

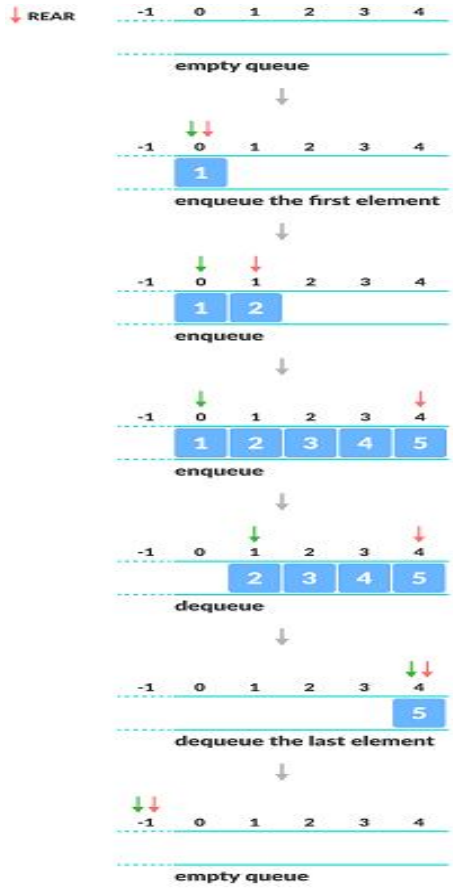
1. If Front = 1 and Rear = Max
2. Print ‘Overflow’ and Exit.
3. If front = NULL than
4. Front = 1
5. Rear = 1
6. else
7. Rear = Rear +1
8. 3. Q [Rear] = ITEM
9. 4. Exit



DELETION IN QUEUE(Dequeue):

Ex:
Algorithm:
Delete (Q, ITEM, FRONT, REAR)
This procedure remove element from queue Q.
1. If Front = Rear = NULL
Print 'Underflow' and Exit.
2. ITEM = Q (FRONT)
3. If Front = Rear
Front = NULL
Rear = NULL
else
Front = Front + 1
4. Exit

Enqueue and Dequeue Operations:



PRIORITY QUEUE

 If is a type of queue in which each element has been assigned a priority such that the order in which the elements are processed according to the elements are processed according to the following rule

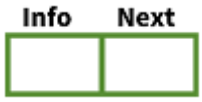
I. This element of high priority is processed first.

II. The element having same priority are processed according to the order in which they are inserted.

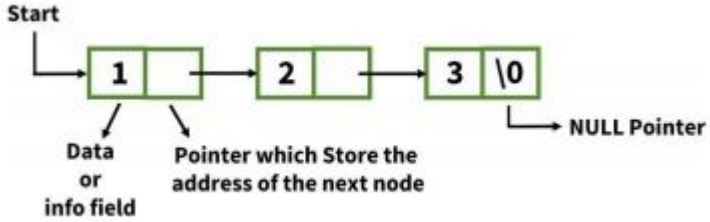
 The lower ranked no. enjoy high priority.

UNIT-6
LINKED LIST

- Linked List is a collection of data elements called as nodes.
- The node has 2 parts
1. Info is the data part
2. Next i.e. the address part that means it points to the next node.



- If linked list adjacency between the elements are maintained by means of links or pointers.
- A link or pointer actually the address of the next node.



- The last node of the list contain '\0' NULL which shows the end of the list.
- The linked list contain another pointer variable 'Start' which contain the address the first node.
- Linked List is of 4 types
1. Single Linked List
2. Double Linked List
3. Circular Linked List
4. Header Linked List

Representation of Linked List in Memory
- Linked List can be represented in memory by means 2 linear arrays i.e. Data or info and Link or address.
- They are designed such that info[K] contains the Kth element and Link[K] contains the next pointer field i.e. the address of the Kth element in the list.
- The end of the List is indicated by NULL pointer i.e. the pointer field of the last element will be NULL or Zero.

SINGLE LINKED LIST

A Single Linked List is also called as one-way list. It is a linear collection of data elements called nodes, where the linear order is given by means of pointers. Each node is divided into 2 parts.

I. Information
II. Pointer to next node.

OPERATION ON SINGLE LINKED LIST

1. TRAVERSAL

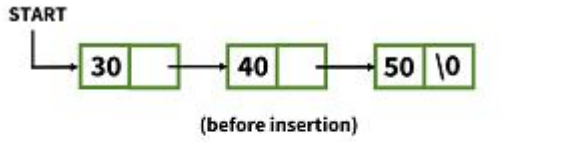
Algorithm:

Display (Start) This algorithm traverse the list starting from the 1st node to the end of the list.

Step-1 : "Start" holds the address of the first node.
Step-2 : Set Ptr = Start [Initializes pointer Ptr]
Step-3 : Repeat Step 4 to 5, While Ptr ≠ NULL
Step-4 : Process info[Ptr] [apply Process to info(Ptr)]

Step-5 : Set Ptr = next[Ptr] [move Print to next node]
[End of loop]
Step-6 : Exit
2. INSERTION
The insertion operation is used to add an element in an existing Linked List.
There are various position where node can be inserted.
a) Insert at the beginning
b) Insert at end
c) Insert at specific location.

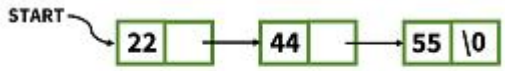
INSERT AT THE BEGINNING
Suppose the new mode whose information field contain 20 is inserted as the first node.



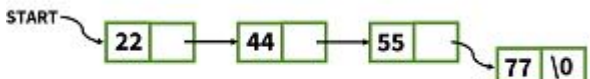
(before insertion)

(After insertion)

Algorithm:
This algorithm is used to insert a node at the beginning of the Linked List.
Start holds the address of the first node.
Step-1 : Create a new node named as P.
Step-2 : If P == NULL then print “Out of memory space” and exit.
Step-3 : Set info [P] = (copies a new data into a new node)
Step-4 : Set next [P] = Start (new node now points to original first node)
Step-5 : Set Start = P
Step-6 : Exit
INSERT AT THE END
 To insert a node at the end of the list, we need to traverse the List and advance the pointer until the last node is reached.
 Suppose the new node whose information field contain 77 is inserted at the last node.



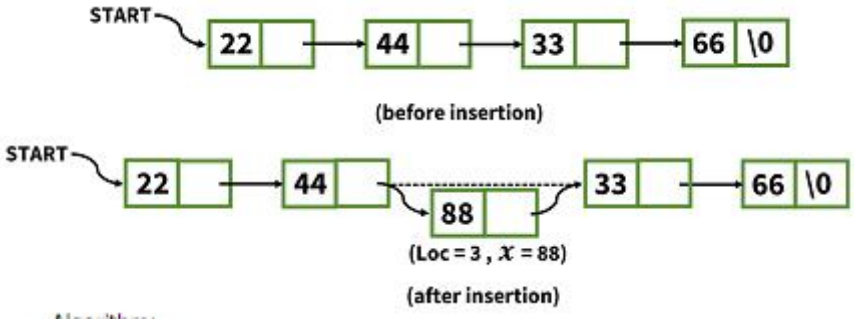
(before insertion)



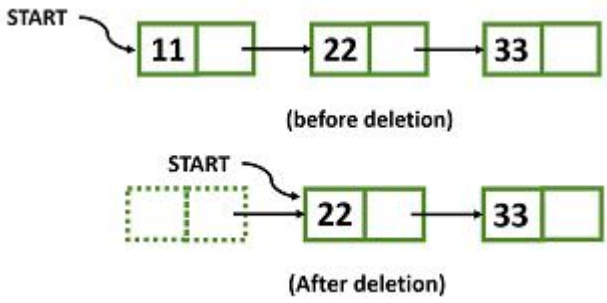
(After insertion)

Algorithm:
This algorithm is used to insert a node at the end of the linked list.
‘Start’ holds the address of the first node.
Step-1 : Create a new node named as P.
Step-2 : If P = NULL then print “Out of memory space” and Exit.
Step-3 : Set info [P] = (copies a new data into a new node)
Step-4 : Set next [P] = NULL
Step-5 : Set Ptr = Start
Step-6 : Repeat Step-7 while Ptr ≠ NULL
Step-7 : Set temp = Ptr

Ptr = next [Ptr]
(End of step-6 loop)
Step-8 : Set next [temp] = P
Step-9 : Exit
INSERT AT ANY SPECIFIC LOCATION
To insert a new node at any specific location we scan the List from the beginning and move up to the desired node where we want to insert a new node.
In the below fig. Whose information field contain 88 is inserted at 3rd location.



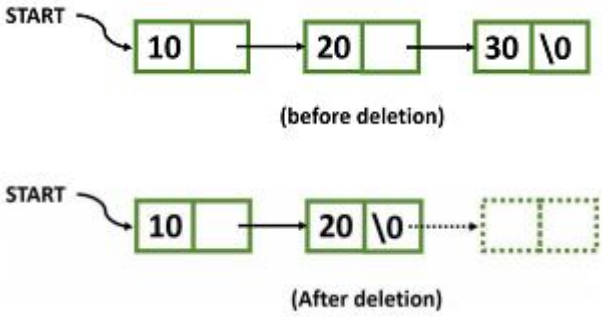
Algorithm:
'Start' holds the address of the 1st node.
Step-1 : Set Ptr = Start
Step-2 : Create a new node named as P.
Step-3 : If P = NULL then write 'Out of memory' and Exit.
Step-4 : Set info [P] = □ (copies a new data into a new node)
Step-5 : Set next [P] = NULL
Step-6 : Read Loc
Step-7 : Set i = 1
Step-8 : Repeat steps 9 to 11 while Ptr ≠ NULL and i < Loc
Step-9 : Set temp = Ptr.
Step-10 : Set Ptr = next [Ptr]
Step-11 : Set i = i + 1
[End of step-7 loop]
Step-12 : Set next [temp] = P
Step-13 : Set next [P] = Ptr
Step-14 : Exit
3. DELETION
The deletion operation s used to delete an element from a single linked list.
There are various position where node can be deleted.
a) Delete the 1st Node



Algorithm:
Start holds the address of the 1st node.
Step-1 : Set temp = Start

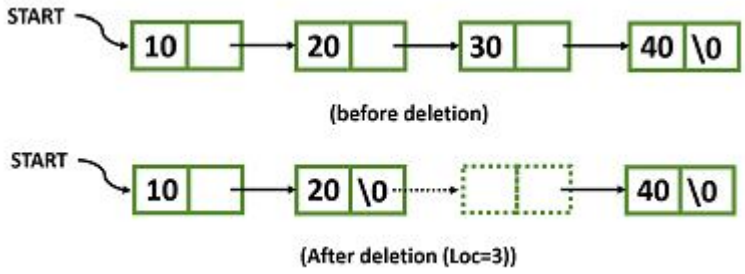
Step-2 : If Start = NULL then write 'UNDERFLOW' & Exit.
Step-3 : Set Start = next[Start]
Step-4 : Free the space associated with temp.
Step-5 : Exit

b) Delete the last node



Algorithm:
Start holds the address of the 1st
node.
Step-1 : Set Ptr = Start
Step-2 : Set temp = Start
Step-3 : If Ptr = NULL then write 'UNDERFLOW' & Exit.
Step-4 : Repeat Step-5 and 6 While next[Ptr] ≠ NULL
Step-5 : Set temp = Ptr
Step-6 : Set Ptr = next[Ptr]
(End of step 4 loop)
Step-7 : Set next[temp] = NULL
Step-8 : Free the space associated with Ptr.
Step-9 : Exit

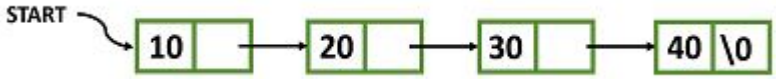
c) Delete the node at any specific location



Algorithm:
Start holds the address of the 1st
node.
Step-1 : Set Ptr = Start
Step-2 : Set temp = Start
Step-3 : If Ptr = NULL then write 'UNDERFLOW' and Exit.
Step-4 : Set i = 1
Step-5 : Read Loc
Step-6 : Repeat Step-7 to 9 while Ptr ≠ NULL and i < Loc
Step-7 : Set temp = Ptr
Step-8 : Set Ptr = next[Ptr]
Step-9 : Set i = i+1
(End of Step 6 loop)
Step-10 : Set next[temp] = next[Ptr]
Step-11 : Free the space associated with Ptr.
Step-12 : Exit

4. SEARCHING

Searching means finding an element from a given list.



Algorithm:
Start holds the address of the 1st
node.
Step-1 : Set Ptr = Start
Step-2 : Set Loc = 1
Step-3 : Read element
Step-4 : Repeat Step-5 and 7 While Ptr ≠ NULL
Step-5 : If element = info[Ptr] then Write 'Element found at position', Loc and Exit.
Step-6 : Set Loc = Loc+1
Step-7 : Set Ptr = next[Ptr]
(End of step 4 loop)
Step-8 : Write 'Element not found'
Step-9 : Exit