

THE DEEPHE

K. SRIVASTAVA
DEEPALEI SRIVASTAVA

BPB PUBLICATIONS



Contents

<u>CHAPTER 1 INTRODUCTION</u>	1-28
<u>1.1 Abstract Data Type</u>	1
<u>1.2 String Operations</u>	2
<u>1.21 Length</u>	2
<u>1.22 Concatenation</u>	3
<u>1.23 Comparison</u>	4
<u>1.24 Copy</u>	5
<u>1.25 Substring</u>	5
<u>1.3 Pattern matching algorithm</u>	6
<u>1.31 First approach</u>	7
<u>1.32 Second Approach</u>	8
<u>1.4 Complexity of Algorithm</u>	10
<u>1.41 Best case</u>	10
<u>1.42 Worst case</u>	11
<u>1.43 Average case</u>	11
<u>1.5 O Notation</u>	11
<u>1.6 Design of Algorithm</u>	12
<u>1.61 Greedy Algorithm</u>	12
<u>1.62 Divide and conquer</u>	12
<u>1.63 Non recursive algorithm</u>	12
<u>1.64 Randomized algorithm</u>	12
<u>1.65 Modular programming approach</u>	12
<u>1.7 Mathematical Notations And Methods</u>	13
<u>1.71 Factorial</u>	13
<u>1.72 Series</u>	13
<u>1.73 Logarithms</u>	14
<u>1.74 Exponents</u>	14
<u>1.75 Modular operation</u>	14
<u>1.76 Fibonacci series</u>	15
<u>1.77 Arithmetic series</u>	15
<u>1.8 Matrices</u>	16
<u>1.81 Matrix Addition</u>	16
<u>1.82 Matrix Multiplication</u>	17
<u>1.83 Lower triangular matrix</u>	17
<u>1.84 Upper triangular matrix</u>	18
<u>1.85 Tridiagonal matrix</u>	18
<u>1.86 Transpose matrix</u>	18
<u>1.9 Recursion</u>	19
<u>1.91 Tail Recursion</u>	22
<u>1.10 Recursion through stack</u>	23
<u>1.11 Removal of recursion through iteration</u>	23
<u>1.12 Tower of hanoi</u>	24
<u>Exercise</u>	28

CHAPTER 2 ARRAYS, POINTERS AND STRUCTURES 29-88

2.1	What is array	29
2.2	Declaration of array	29
2.3	Processing with array	30
2.4	String library functions	30
2.41	strlen ()	31
2.42	strcmp ()	31
2.43	strcpy ()	31
2.44	strcat ()	31
2.5	Initialization of array	31
2.6	Use of array in function	32
2.7	Multidimensional array	33
2.8	What is pointer	35
2.9	Uses of pointer	35
2.10	The & and * operator	35
2.11	Declaration of pointer	36
2.12	Pointer to pointer	39
2.13	Pointer arithmetic	40
2.14	Pointers and functions	43
2.141	Call by value	44
2.142	Call by reference	45
2.15	Pointer and array	49
2.16	Pointer with multidimensional array	55
2.17	Array of pointers	58
2.18	Pointer and string	60
2.19	Two dimensional array of characters	62
2.20	Array of pointer to string	63
2.21	Dynamic memory allocation	64
2.211	sizeof()	65
2.212	malloc()	65
2.213	calloc()	66
2.214	free()	67
2.215	realloc()	67
2.22	What is structure	68
2.23	Another type of declaration	69
2.24	Initialize value to the structure	69
2.25	Another type of initialization	69
2.26	Array of structures	70
2.27	Passing structure to function	70
2.28	Passing array of structure to function	70
2.29	Structure within structure	71
2.30	typedef	72
2.31	union	73
2.32	Pointer to structure	74
	Exercise	79

CHAPTER 3 LINKED LIST89-138

<u>3.1</u>	<u>What is list</u>	89
<u>3.2</u>	<u>Array Implementation of list</u>	89
<u>3.21</u>	<u>Traversing an array list</u>	90
<u>3.22</u>	<u>Searching in an array list</u>	90
<u>3.23</u>	<u>Insertion into an array list</u>	90
<u>3.24</u>	<u>Deletion from an array list</u>	91
<u>3.3</u>	<u>Linked List</u>	95
<u>3.31</u>	<u>Traversing a linked list</u>	96
<u>3.32</u>	<u>Searching into a linked list</u>	96
<u>3.33</u>	<u>Insertion into a linked list</u>	97
<u>3.34</u>	<u>Deletion from a linked list</u>	98
<u>3.4</u>	<u>Reverse linked list</u>	104
<u>3.41</u>	<u>Creation of reverse()</u>	104
<u>3.5</u>	<u>Circular linked list</u>	108
<u>3.51</u>	<u>Creation of circular linked list</u>	108
<u>3.52</u>	<u>Traversal in circular linked list</u>	108
<u>3.53</u>	<u>Insertion into a circular Linked list</u>	109
<u>3.54</u>	<u>Deletion from circular linked list</u>	109
<u>3.6</u>	<u>Sorted linked list</u>	114
<u>3.7</u>	<u>Double linked list</u>	118
<u>3.71</u>	<u>Traversing a doubly linked list</u>	119
<u>3.72</u>	<u>Insertion into a doubly linked list</u>	119
<u>3.73</u>	<u>Deletion from doubly linked list</u>	121
<u>3.8</u>	<u>Polynomial arithmetic with linked list</u>	127
<u>3.9</u>	<u>Creation of polynomial linked list</u>	128
<u>3.10</u>	<u>Addition with polynomial linked list</u>	128
<u>3.101</u>	<u>Implementation</u>	130
	<u>Exercise</u>	136

CHAPTER 4 STACK AND QUEUE139-188

<u>4.1</u>	<u>✓ Stack</u>	139
<u>4.2</u>	<u>✓ Array Implementation of Stack</u>	141
<u>4.21</u>	<u>✓ Push operation on stack</u>	141
<u>4.22</u>	<u>✓ Pop operation on stack</u>	141
<u>4.3</u>	<u>Linked List Implementation</u>	143
<u>4.31</u>	<u>✓ Push operation on Stack</u>	144
<u>4.32</u>	<u>✓ Pop operation on Stack</u>	144
<u>4.4</u>	<u>✓ Queue</u>	146
<u>4.5</u>	<u>✓ Array Implementation of Queue</u>	147
<u>4.51</u>	<u>✓ Add operation in queue</u>	148
<u>4.52</u>	<u>✓ Delete operation in queue</u>	148
<u>4.6</u>	<u>Linked List implementation</u>	151
<u>4.61</u>	<u>Add operation in Queue</u>	151
<u>4.62</u>	<u>Delete operation in Queue</u>	152

<u>4.7</u> ✓ Circular Queue	156
4.71 ✓ Add operation in Circular Queue	157
4.72 ✓ Delete operation in Circular Queue	158
<u>4.8</u> Priority Queue	161
<u>4.9</u> Linked list implementation of priority queue	161
4.91 Operation in priority queue	162
4.911 Add operation in Priority Queue	162
4.912 Delete operation in Priority Queue	162
<u>4.10</u> ↗ Dequeue	165
<u>4.11</u> ↗ Array Implementation of Dequeue	165
4.111 ↗ Add and delete operation in Dequeue	165
<u>4.12</u> Applications of stack	172
4.121 Reversal of string	172
4.122 Checking validity of an expression containing nested parentheses	173
<u>4.13</u> Polish Notation with arithmetic expression	175
<u>4.14</u> Polish Notation	177
<u>4.15</u> Converting infix expression into postfix expression	178
<u>4.16</u> Evaluation of postfix expression	179
<u>4.17</u> Sparse Matrix	184
4.171 3-tuple Method	185
Exercise	187

CHAPTER 5 TREES 189-297

<u>5.1</u> Binary Tree	189
<u>5.2</u> Strictly Binary Tree	191
<u>5.3</u> Complete Binary Tree	191
<u>5.4</u> Extended Binary Tree	192
<u>5.5</u> Algebraic Expression representation in tree	192
<u>5.6</u> Representation of Binary Tree	194
5.61 Linked Representation	194
<u>5.7</u> Traversing in Binary Tree	195
5.71 Preorder Traversal	197
5.72 Inorder Traversal	197
5.73 Postorder Traversal	198
<u>5.8</u> Non recursive functions for traversals	201
5.81 Preorder Traversal	201
5.82 Inorder Traversal	201
5.83 Postorder Traversal	202
<u>5.9</u> Level order traversal	204
<u>5.10</u> Creation of binary tree from preorder and inorder traversals	204
<u>5.11</u> Creation of tree from postorder and inorder traversals	207
<u>5.12</u> Shortcut method of creating the tree from preorder and inorder traversal	208
<u>5.13</u> Binary Search Tree	210
5.131 Search and Insertion Operations	211
5.1311 Creation of find()	212
5.1312 Insertion in Binary Search tree	212

<u>5.132</u>	<u>Deletion operation</u>	213
<u>5.1322</u>	<u>Creation of function case_a()</u>	217
<u>5.1322</u>	<u>Creation of function case_b()</u>	217
<u>5.1323</u>	<u>Creation of function case_c()</u>	218
<u>✓5.133</u>	<u>Traversal in Binary Search Tree</u>	221
<u>5.1331</u>	<u>Preorder Traversal</u>	221
<u>5.1332</u>	<u>Inorder Traversal</u>	221
<u>5.1333</u>	<u>Postorder Traversal</u>	221
<u>5.134</u>	<u>Recursive Function for finding a node in Binary search tree</u>	227
<u>5.14</u>	<u>Threads</u>	227
<u>5.141</u>	<u>Finding inorder successor of a node in in-threaded tree</u>	230
<u>5.142</u>	<u>Finding inorder predecessor of a node in in-threaded tree</u>	231
<u>5.143</u>	<u>Inorder Traversal in in-threaded binary tree</u>	231
<u>5.144</u>	<u>Preorder traversal of in-threaded binary tree</u>	232
<u>5.145</u>	<u>Insertion and deletion in threaded binary tree</u>	233
<u>5.1451</u>	<u>Insertion in a threaded binary search tree</u>	233
<u>5.1452</u>	<u>Deletion from a threaded binary search tree</u>	234
<u>5.15</u>	<u>AVL Tree(Balanced Tree)</u>	242
<u>5.151</u>	<u>Insertion in AVL tree</u>	244
<u>5.152</u>	<u>AVL Rotations</u>	246
<u>5.1521</u>	<u>Left to Left rotation</u>	246
<u>5.1522</u>	<u>Right to Right rotation</u>	247
<u>5.1523</u>	<u>Left to right rotation</u>	248
<u>5.1524</u>	<u>Right to left rotation</u>	250
<u>5.16</u>	<u>Huffman Tree</u>	258
<u>5.161</u>	<u>Huffman Algorithm</u>	259
<u>5.162</u>	<u>Use in application</u>	262
<u>✓5.17</u>	<u>Heap</u>	263
<u>5.171</u>	<u>Insertion in Heap</u>	264
<u>5.172</u>	<u>Deletion in heap</u>	267
<u>5.18</u>	<u>General Tree</u>	273
<u>5.181</u>	<u>Linked representation of General Tree</u>	274
<u>5.19</u>	<u>B Tree</u>	275
<u>5.191</u>	<u>Insertion in B tree</u>	276
<u>5.192</u>	<u>Deletion in B-tree</u>	280
<u>5.20</u>	<u>B+ tree</u>	289
<u>5.201</u>	<u>Insertion in B+ tree</u>	290
<u>5.202</u>	<u>Deletion in B+ tree</u>	290
<u>5.21</u>	<u>Digital Search Tree</u>	290
<u>5.22</u>	<u>Trie</u>	291
<u>5.221</u>	<u>Traversal in trie</u>	292
<u>5.222</u>	<u>Insertion in Trie</u>	292
<u>5.223</u>	<u>Deletion in Trie</u>	293
<u>5.224</u>	<u>Analysis</u>	293
<u>5.23</u>	<u>Optimum Search Tree</u>	293
<u>Exercise</u>	<u>.....</u>	297

CHAPTER 6. GRAPH	298-378
6.1 Undirected Graph	298
6.2 Directed Graph	298
6.3 Representation of Graph	301
6.31 Adjacency Matrix	301
6.32 Adjacency List	304
6.4 Operations on Graph	306
6.41 Insertion in Adjacency Matrix	307
6.411 Node insertion	307
6.412 Edge insertion	308
6.42 Deletion in adjacency matrix	308
6.421 Node deletion	308
6.422 Edge deletion	312
6.43 Insertion in adjacency list	312
6.431 Node insertion	312
6.432 Edge insertion	313
6.44 Deletion in adjacency list	313
6.441 Node deletion	313
6.442 Edge deletion	314
6.5 Path Matrix	320
6.6 Computing Path matrix from powers of adjacency matrix	320
6.7 Warshall's Algorithm	325
6.8 Modified Warshall's Algorithm	329
6.9 Traversal In Graph	334
6.91 Breadth First Search	334
6.911 Breadth First Search through queue	336
6.92 Depth First Search	338
6.921 Depth First Search through stack	339
6.10 Shortest Path Algorithm (Dijkstra)	347
6.11 Spanning Tree	356
6.12 Minimum Spanning Tree	356
6.13 Prim's Algorithm	356
6.14 Kruskal's Algorithm	363
6.15 Topological Sorting	370
Exercise	376
CHAPTER 7 SORTING	379-423
7.1 What is sorting	379
7.2 Efficiency Parameters	381
7.3 Efficiency of sorting	381
7.4 Bubble Sort	382
7.41 Analysis	384
7.5 Selection Sort	385
7.51 Analysis	387
7.6 Insertion Sort	387

<u>7.61</u>	<u>Analysis</u>	389
<u>7.7</u>	<u>Shell Sort</u>	390
<u>7.71</u>	<u>Analysis</u>	392
<u>7.8</u>	<u>Merging</u>	393
<u>7.9</u>	<u>✓ Merge Sort</u>	396
<u>7.91</u>	<u>Analysis</u>	400
<u>✓ 7.10</u>	<u>Radix Sort</u>	400
<u>7.101</u>	<u>Analysis</u>	405
<u>7.11</u>	<u>Address Calculation Sort</u>	405
<u>7.111</u>	<u>Analysis</u>	409
<u>7.12</u>	<u>Quick Sort</u>	409
<u>7.121</u>	<u>Analysis</u>	413
<u>7.13</u>	<u>Binary Tree Sort</u>	413
<u>7.131</u>	<u>Analysis</u>	415
<u>✓ 7.14</u>	<u>Heap Sort</u>	415
<u>7.141</u>	<u>Analysis</u>	421
<u>7.15</u>	<u>Comparison</u>	421
<u>Exercise</u>		422

CHAPTER 8 SEARCHING, HASHING AND STORAGE MANAGEMENT 424

<u>8.1</u>	<u>Sequential searching</u>	424
<u>8.11</u>	<u>Analysis</u>	425
<u>8.2</u>	<u>Binary Search</u>	426
<u>8.3</u>	<u>Hashing</u>	429
<u>8.4</u>	<u>Choosing a hash function</u>	431
<u>8.41</u>	<u>Truncation Method</u>	431
<u>8.42</u>	<u>Mid square Method</u>	431
<u>8.43</u>	<u>Folding Method</u>	432
<u>8.44</u>	<u>Modular Method</u>	432
<u>8.5</u>	<u>Hash function for floating point numbers</u>	433
<u>8.6</u>	<u>Hash function for strings</u>	435
<u>8.7</u>	<u>Collision Resolution (Open Hashing)</u>	436
<u>8.71</u>	<u>Separate chaining</u>	436
<u>8.8</u>	<u>Closed Hashing (Open Addressing)</u>	437
<u>8.81</u>	<u>Linear Probing</u>	437
<u>8.82</u>	<u>Quadratic Probing</u>	438
<u>8.83</u>	<u>Double Hashing</u>	438
<u>8.9</u>	<u>Rehashing</u>	439
<u>8.10</u>	<u>Extendible Hashing</u>	441
<u>8.11</u>	<u>Storage Management</u>	442
<u>8.12</u>	<u>Garbage collection</u>	442
<u>8.13</u>	<u>Dynamic memory management</u>	443
<u>8.14</u>	<u>Method to select free block</u>	443
<u>8.141</u>	<u>First Fit</u>	444
<u>8.142</u>	<u>Best Fit</u>	444
<u>8.143</u>	<u>Worst Fit</u>	445
<u>8.15</u>	<u>Freeing Memory</u>	445

8.16	Boundary Tag Method	445
8.17	Buddy Systems	448
8.171	Binary Buddy System	449
8.172	Fibonacci Buddy System	452
	Exercise	452

Index

Chapter 1

Introduction

Data we get in any form, it can be numeric or character. When we process this data then it becomes information. Let us take a string UP32V-2126, it looks like sequence of character type data, but it represents the car registration number then it is processed data i.e. information. Data can be distinguished in different data types like integer, floating or character type. These data can be collected and organized in some way like records. Let us take we have data "Suresh 28 M". Here "Suresh" is of string data type name, 28 is of integer data type age and M is of character data type gender. We can collect it and organize as record.

Record		
Name	Age	Gender
"Suresh"	28	M

Now we can collect records and store in a file. Basically we are organizing this data in some way so it will be easy to handle.

Data structure is a way to organize the data in some way so we can do the operations on these data in effective way. Some example of data structures are list, stack, queue, tree, graph, hash table etc. We select these data structure depending upon which type of operation is needed with data. Suppose we have a need to handle some processes which are available in queue then we implement this situation with queue data structure. Sometimes a situation can be handled with different data structure then we take the efficiency and storage in consideration. We can implement these data structures and their operations in some generalized manner then they can be used in some other modules also.

We do only some specific operations with data structure. Data structure with these specific operations are called abstract data type. This is basically user defined data type. Here user defines the data structure for organizing his specific data and provides set of operations to handle this data structure.

Abstract Data Type

An abstract data type commonly called as ADT is nothing but a set of operations which is used with the component of the element of that abstract data type. We can simply take the abstract data type of a list.

List ADT-

Component-
Item

Operations-
Insertion
Deletion
Search
Display

Here Item is the component of Abstract data type. List can contain number of items. Operations on these items can be insertion, deletion, search, display. There can be more operations but basically only these operations will be with item.

String Operations-

String is a sequence of consecutive characters which ends with '\0'. In 'C' language character type array is used for storing the string value. There are so many requirement where operations on string are needed. These can be concatenation, comparison, copy, length, substring search etc. Let us take some string operations and see how they can be implemented in programs.

Length-

This operation returns the length of the string.

Length("ChandraVaibhav")

It will return the value 14 which is number of characters in string "ChandraVaibhav".

```
/*Program to find the length of string*/
#include<stdio.h>
main()
{
    char str[25];
    int length;
    printf("Enter the string : ");
    gets(str);
    length=Length(str);
    printf("Length of string is %d\n",length);
}/*End of main()*/
```

```
Length(char *str)
{
    int len=0;
```

```

while(*str != '\0')
{
    len++;
    str++;
}
return(len);
}/*End of Length( )*/

```

Concatenation-

This operation adds one string at the end of another string.

```

char str1[30] = "Kumar";
char str2[30] = "Manish";
Concat(str1,str2);

```

This will concatenate the second string "Manish" into the first string "Kumar". So now the value of string will be "KumarManish".

```

/*program to concatenate the two strings*/
#include<stdio.h>
main()
{
    char str1[30],str2[30];
    printf("Enter the first string : ");
    gets(str1);
    printf("Enter the second string : ");
    gets(str2);
    Concat(str1,str2);
    printf("Now the first string is %s\n",str1);
}/*End of main()*/

```

```

Concat(char *str1, char *str2)
{
    while(*str1 != '\0')
        str1++;
    while(*str2 != '\0')
    {
        *str1 = *str2;
        str1++;
        str2++;
    }
    *str1 = '\0';
}/*End of Concat( )*/

```

Comparison-

This operation is used for comparing two strings.

Compare(string1,string2)

It will return following value-

```

< 0    if string1 < string2
= 0    if string1 == string2
> 0    if string1 > string2

#include<stdio.h>
main()
{
    char str1[30],str2[30];
    int s;
    printf("Enter the first string : ");
    gets(str1);

    printf("Enter the second string : ");
    gets(str2);

    s=Compare(str1,str2);

    if(s==0)
        printf("Strings are same\n");
    else
        if(s>0)
            printf("String 1 is greater than string 2\n");
        else
            printf("String 2 is greater than string 1\n");
}/*End of main() */

int Compare(char *str1,char *str2)
{
    while(*str1!='\0'||*str2!='\0')
    {
        if( *str1 == *str2 )
        {
            str1++;
            str2++;
        }
        else
            return( *str1 - *str2 );
    }
    return 0;
}/*End of Compare()*/

```

Copy-

This operation copies one string to another.

```
char str1[30] = "Suresh";
char str2[30] = "Kumar";
CopyString(str1,str2);
```

This will copy the value "Kumar" of str2 to str1. Now the value of str1 will be "Suresh".

```
#include<stdio.h>
main( )
{
    char str1[30],str2[30];
    int i=0;
    printf("Enter the first string : ");
    gets(str1);

    printf("Enter the second string : ");
    gets(str2);

    CopyString(str1,str2);
    printf("Now the first string is same as the second string \n");
    printf("First string is %s \n",str1);
    printf("Second string is %s \n",str2);
}/*End of main() */
```

```
CopyString(char *str1,char *str2)
{
    while(*str2 != '\0')
    {
        *str1 = *str2;
        str1++;
        str2++;
    }
    *str1 = '\0';
}/*End of CopyString() */
```

Substring-

This operation will return the starting position of string in another string.

```
char str1[30] = "SURESH";
char str2[30] = "ES";
substr(str1,str2);
```

This will return the index value of string "ES" in string "SURESH" which is 4.

Another version of this operation returns the substring from one string.

```
char str[30] = "SURESHKUMAR";
substr(str,5,4);
```

Here 5 is the start position in string str which is 'S' and 4 is the number of characters from substring. So this will return the substring "SHKU".

```
/*Program of substring */
#include<stdio.h>
main()
{
    char string[30];
    int n,s;
    printf("Enter the string : ");
    gets(string);
    printf("Enter the position from where substring starts : ");
    scanf("%d",&s);
    printf("Enter number of characters in the substring : ");
    scanf("%d",&n);
    substr(string,s,n);
    printf("\n");
}/*End of main()*/
```



```
substr(char *string, int s,int n)
{
    int i;
    for(i=0;i<s-1;i++)
    {
        string++;
        if(*string=='\0')
        {
            printf("There are less than %d characters in string\n",s);
            exit(1);
        }
    }
    for(i=1;i<=n;i++)
    {
        printf("%c",*string);
        string++;
    }
}/*End of substr()*/
```

Pattern matching algorithm-

In string manipulation many times such a need arises where we want to find whether a string is in particular string or not. Hence the problem is of matching a pattern of

string in a given text or string. There are two approaches for matching the pattern of string, both have some advantage and disadvantage. Let us take a pattern for matching is-
 S1 = "Kumar"

The given text where we want to match this pattern is-

S2 = "Suresh Kumar Srivastava"

Now we want to find whether string S1 is in string S2 or not and also what is the starting position of S1 in S2. Here length of S1 should be less than or equal to S2.

First approach-

In this approach we match each character of S1 with each character of S2 from 0,1,2,.....N-1 until the string is matched.

Let us take an example, if length of S1 is 5 and S2 is 21 then first we match S1 as-

S1[0][1][2][3][4] with S2[0][1][2][3][4] and after that with-

S2[1][2][3][4][5]

S2[2][3][4][5][6]

.....

.....

S2[16][17][18][19][20]

Here we are comparing the substring of S2 with S1. If any character of S1 is not matched with any character of substring of S2 then we match with next substring of S2.

Let us take X is the substring of S2.

X0 = S2[0].....[4]

X1 = S2[1].....[5]

.....

.....

X16 = S2[16][20]

Here we match each character of S1 with the substring of S2 from X0.....X16

Total number of substrings = length of S2 – length of S1 + 1

Let us take S1 = "Kumar" and S2 = "Suresh Kumar Srivastava"

S1 is the substring of S2 and started from the position 7.

This process can be implemented in program with introducing two loops (loop within loop). Outer loop will be for the text in which we have to search the pattern and inner loop will be for matching the pattern.

```

/* Program for implementing the first approach */
#include<stdio.h>
#include<string.h>
main( )
{
    char text[25],pattern[80];
    int position;

    printf("Enter the text : ");
    gets(text);
    printf("Enter the pattern for matching : ");
    scanf("%s",pattern);
    position=pmatch(text,pattern);
    if(position==0)
        printf("%s does not exist in %s \n",pattern,text);
    else
        printf("%s starts at %d\n",pattern,position);
}/*End of main()*/
int pmatch(char text[],char pattern[])
{
    int matched,i,j,k=0;
    if( strlen(pattern) > strlen(text) )
        return 0;
    for(i=0;i<=strlen(text);i++)
    {
        k=0;
        for(j=i;k<strlen(pattern);j++,k++)
        {
            if(pattern[k]!=text[j])
            {
                matched=0;
                break;
            }
            else
                matched=1;
        }/*End of for */
        if(matched==1)
            return(i+1);
    }/*End of for */
    return 0;
}/*End of pmatch() */

```

Second Approach-

Second approach of pattern matching algorithm uses the table of states which is created with the letters of pattern. Let us take a string which we want to search is-

$S = abbc$

Initially state will be I1. The table created with the pattern string will be-

State	a	b	c	Other
I1	I2	I1	I1	I1
I2	I1	I3	I1	I1
I3	I3	I4	I1	I1
I4	I1	I1	S	I1

Here we can see the table created for the state and how the state will be changed after finding next letter of pattern string. Initial state is I1 and when it gets the letter 'a' from the text then this state changes to I2. Similarly if state I2 gets the letter 'b', it changes to I3, I3 gets letter 'b' it changes to I4 and when I4 gets letter 'c' it changes to S, means it got the pattern string. Now it's index is the position of last letter of pattern found in text. So the starting index of the pattern will be-

$$\text{Current index} - \text{Length of pattern string}$$

In any stage of state if it finds any other letter which is not in pattern then the state will be changed to initial state I1.

Let us take an example-

Text = abdabbcba

Pattern = abbc State = I1 Index = 1

1. Letter = a Index = 2
State = I2
2. Letter = b Index = 3
State = I3
3. Letter = d Index = 4
State = I1
4. Letter = a Index = 5
State = I2
5. Letter = b Index = 6
State = I3
6. Letter = b Index = 7
State = I4
7. Letter = c Index = 8
State = S

So the starting index of the pattern string
 = Current index - Length of pattern string
 = 8 - 4
 = 4

Here table will be in memory and it will be accessed by the algorithm when it scans each letter of the text.

Complexity of Algorithm

An algorithm is a sequence of steps that gives method of solving a problem. It creates the logic of program. Efficiency of algorithm depends on two major criteria, first one is run time of that algorithm and second is the space. Run time of algorithm means the time taken by program for execution. We implement different data structures, some data structure takes more space but improves the run time and some takes less space but it effects run time of algorithm.

We can take a simple case of hash table and linked list. Hash table takes more space but searching is fast, linked list takes less space than hash table but searches sequentially and slow. So we can improve the run time by simply increasing the space at different places.

The major criteria for complexity of any algorithm is comparison of keys and moving of data, means number of times the key is compared and data is moved. Suppose space is fixed for one algorithm then only run time will be considered for obtaining the complexity of algorithm. We takes 3 cases for complexity of algorithms-

1. Best case
2. Worst case
3. Average case

1. Best case-

Generally most of the algorithms behave sometimes in best case. In this case, algorithm searches the element in first time itself. So taking this case for complexity of algorithm doesn't tell too much. Let us take a case of linear search, if it finds the element at first time itself then it behaves as best case.

2. Worst case-

Generally we see the complexity of algorithm in it's worst case behaviour, because it tells the behaviour of algorithm in worst case and we want to improve the worst case behaviour of algorithm. In this case we find the element at the end or when searching of element fails. Let us again take a case of linear search. Suppose the element for which algorithm is searching, is the last element of array or it's not available in array then algorithm behaves as worst case.

Worst case analysis is necessary in the view that algorithm will perform at least up to this efficiency and it will not go for less than this efficiency.

3. Average case-

Analysing the average case behaviour of algorithm is little bit complex than best case and worst case. Here we take the probability with list of data. Average case of algorithm should be the average number of steps but since data can be at any place, so finding exact behaviour of algorithm is difficult. As the volume of data increases average case of algorithm behaves like worst case of algorithm.

O Notation-

O notation is used to measure the performance of any algorithm. Performance of any algorithm depends upon the volume of input data. It is proportional to the input data. O notation is used to define the order of growth for any algorithm. For a particular input size n we know how a particular algorithm behaves but when n increases and it becomes larger, then the same algorithm behaves differently. So O notation defines the order of growth of any algorithm which is useful to measure the performance of algorithm.

Let us take two functions $f(n)$ and $g(n)$. Here $g(n)$ is upper bound of $f(n)$ for some constant multiplier. So we can say $f(n)$ is of $O(g(n))$ and

$$f(n) \leq c g(n) \quad \text{for all } n \geq N$$

or we can say for any value of $n \geq N$, $f(n)$ is bounded by multiplier of $g(n)$.

Let us take a function $f(n) = n^2 + 10n$ and $g(n) = n^2$. Here for all $n \geq 10$, value of $f(n)$ will always less than $2n^2$ which is $g(n)$. So we can tell that $f(n)$ is of $O(n^2)$.

Let us take some situations and behaviour of algorithm in order-

- | | |
|---------------|---|
| $O(1)$ | When we get data in first time itself eg. hash table. |
| $O(n)$ | When all the elements of list will be traversed eg. best case of bubble sort. |
| $O(n^2)$ | When the full list will be traversed for each element eg. worst case of bubble sort. |
| $O(\log n)$ | When we divide list half each time and traverse the middle element eg. binary searching, binary tree traversal. |
| $O(n \log n)$ | When we divide list half each time and traverse that half portion eg. best case of quick sort |

Design of Algorithm-

As we know algorithm is a sequence of steps which creates the logic of program. For

designing of any algorithm some important things should be considered. These are run time, space and simplicity of algorithm. Some places input data also decides in designing of algorithm. Now we will discuss some common approaches for designing algorithm.

1. Greedy Algorithm-

This algorithm works in steps. In each step it selects the best available option until all options finish. This approach is widely used in so many places for designing algorithm eg. shortest path algorithm.

2. Divide and conquer-

Here we divide the big problem into some same type of small problems and we design the algorithm to combine the implementation of these small problems for implementing big problem. Example of this approach is quick sort where we divide the initial list into several small lists, after sorting those smaller lists we combine them and get the initial list sorted.

3. Non recursive algorithm-

We know that recursion is very powerful technique which is supported by C but some compilers don't support this feature. In some places recursion is not as effective and it can be avoided with iteration concept easily. So we have a need to design the algorithm with non recursive approach.

4. Randomized algorithm-

In randomized algorithm, we use the feature of random number instead of fixed number. Performance of some algorithm depends upon the input data. It gives different results with different input data. Let us take a case of quick sort, it behaves $O(n^2)$ in worst case. We take the pivot element for dividing the list. Suppose element of list are already in sorted order and we are taking the pivot as first element then it will behave as $O(n^2)$. Here we can take any element of list randomly for pivot, so it can improve the performance of algorithm. In any way it will not behave worse than $O(n^2)$ because it's a worst case behaviour of quick sort. Suppose with this random selection of pivot it behaves as average case of quick sort then its performance will be of $O(n \log n)$.

5. Modular programming approach-

Here we divide the big problem into smaller ones which are totally different from each other. Then we combine the solution of all smaller problems and we get the solution of big problem. Actually here we can use different algorithm design for all small problems. This approach gives different modules for different problems and makes it easier to handle the big problem.

Mathematical Notations And Methods

Factorial-

Factorial of any number n is the value of product of numbers from n to 1. It's notation is $n!$.

$$n! = n * (n-1) * (n-2) * \dots * 3 * 2 * 1$$

Let us take a number 4, Factorial of 4 is-

$$4! = 4 * 3 * 2 * 1 = 24$$

Factorial of zero(0!) is 1. This operation can be with only positive integer.

So if we are writing program for getting factorial of any number then these things should be under consideration.

Series-

We will use some series in analysis. For summation of these series we use the notation Σ

Let us take a series-

$$x_1 + x_2 + x_3 + \dots + x_n$$

$$= \sum_{i=1}^n x_i$$

Here i is the index for that element which indicates that particular element.

$$x^1 + x^2 + x^3 + \dots + x^n$$

$$= \sum_{i=1}^n x^i$$

$$x_c + \dots + x_d$$

$$= \sum_{i=c}^d x_i$$

Here element starts from the index c and ends at d.

Let us take one more series-

$$cx_1 + cx_2 + cx_3 + \dots + cx_n$$

$$= \sum_{i=1}^n cx_i$$

Here c is constant value in series.

Logarithms-

Logarithms are very useful scientific convention to denote the number. The value of log depends on the base of logarithm. Commonly used bases are 10, 2 and e (natural log) = 2.72. Here generally we will use the log of base 2.

$\log_{10} a \rightarrow$ logarithm of base 10

$\log_e a \rightarrow$ logarithm of base e

$\log_2 a \rightarrow$ logarithm of base 2

Some operations with logarithm are as-

$$\log ab = \log a + \log b$$

$$\log a/b = \log a - \log b$$

$$\log a^x = x \log a$$

$$\log_a a^x = x$$

If base and number are same then value will be the number denoted for exponent.

Exponents-

Some operations with exponents are as-

$$a^{x-y} = a^x / a^y$$

$$a^{xy} = a^x \cdot a^y$$

$$a^{xy} = (a^x)^y$$

$$a^x + a^x = 2a^x$$

$$2^x + 2^x = 2^{x+1}$$

$$a^{-x} = 1/a^x$$

Modular operation-

This operation is useful for getting the remainder value. Let us take an example-

N modulo X

Here N will be divided by X and remainder will be the value of this modulo opera

13 modulo 5 = 3

For modulo we use the notation M.

$13 M 5 = 3$

In 'C', % operator is available for modulus operation.

$$13 \% 5 = 3$$

Fibonacci series:-

This is the series of numbers where first number $F_0 = 0$, second number $F_1=1$ and the next number will be the sum of previous two numbers.

$$F_n = F_{n-1} + F_{n-2}$$

So the series will be as-

$$0, 1, 1, 2, 3, 5, 8, \dots$$

```
/*Program to generate fibonacci series upto some limit*/
#include<stdio.h>
main( )
{
    int F1=0, F2=1, F3, limit;
    printf( " Series up to: " );
    scanf( "%d", &limit );
    if( limit >= 2 )
    {
        printf( "Fibonacci series is:\n" );
        printf( "%d, %d ", F1, F2 );
        F3 = F1 + F2;
        while( F3 <= limit )
        {
            printf( ", %d", F3 );
            F1 = F2;
            F2 = F3;
            F3 = F1 + F2;
        }/* End of while */
    }/* End of if */
}/* End of main() */
```

Arithmetic series:-

This is the series of numbers and two consecutive numbers have equal difference. As example-

$$1, 3, 5, 7, 9, 11, \dots$$

Sum of this series is $S = n/2 [2a + (n-1)d]$

Here 'n' is the total numbers in the series, 'a' is the first number of series and 'd' is the difference between second number and first number.

Matrices-

A matrix is a collection of elements represented in rows and columns. A $m \times n$ matrix is a collection of $m \times n$ elements where m is the number of rows and n is the number of columns in matrix. A $m \times n$ matrix C can be represented as-

$$C_{m \times n} = \begin{bmatrix} C_{11} & C_{12} & \dots & \dots & C_{1n} \\ C_{21} & C_{22} & \dots & \dots & C_{2n} \\ \dots & \dots & \dots & \dots & \dots \\ C_{m1} & C_{m2} & \dots & \dots & C_{mn} \end{bmatrix}$$

This can be represented very easily in two dimensional array, which we will see later in next chapter. Let us take an example of matrix-

$$C_{3 \times 4} = \begin{bmatrix} 4 & 8 & 0 & 3 \\ 9 & 12 & -9 & 1 \\ -4 & 7 & 5 & 2 \end{bmatrix}$$

This is 3×4 matrix which is of 3 rows and 4 columns. Here $C_{11} = 4$, $C_{14} = 3$, $C_{31} = -4$ and $C_{34} = 2$.

Matrix Addition-

Addition of two matrices requires same number of rows and columns in both matrices. Here we add corresponding element of both matrices. Let us take two matrices A & B and add them.

$$A = \begin{bmatrix} 4 & 3 & 2 \\ 5 & 4 & 3 \\ 2 & 6 & 8 \end{bmatrix} \quad B = \begin{bmatrix} 9 & 5 & 6 \\ 4 & 8 & 7 \\ 1 & 3 & 11 \end{bmatrix}$$

$$A + B = \begin{bmatrix} 13 & 8 & 8 \\ 9 & 12 & 10 \\ 3 & 9 & 19 \end{bmatrix}$$

Matrix Multiplication-

Multiplication of matrices requires the number of columns in first matrix should be equal to number of rows in second matrix. Let us take two matrices $A_{m \times n}$ and $B_{n \times p}$. After multiplication it will be in form of $m \times p$ matrix. Here we multiply each row of first matrix with the column of second matrix and add them for getting the element of resultant matrix. Elements of resultant matrix will be as-

$$C_{11} = A_{11} \times B_{11} + A_{12} \times B_{21} + A_{13} \times B_{31} + \dots + A_{1n} \times B_{n1}$$

$$C_{12} = A_{21} \times B_{12} + A_{22} \times B_{22} + A_{23} \times B_{32} + \dots + A_{2n} \times B_{n2}$$

$$C_{ij} = A_{i1} \times B_{1j} + A_{i2} \times B_{2j} + A_{i3} \times B_{3j} + \dots + A_{in} \times B_{nj}$$

Let us take two matrices A and B and multiply them-

$$A_{2 \times 2} = \begin{bmatrix} 4 & 5 \\ 3 & 2 \end{bmatrix} \quad B_{2 \times 3} = \begin{bmatrix} 2 & 6 & 3 \\ -3 & 2 & 4 \end{bmatrix}$$

$$C_{2 \times 3} = \begin{bmatrix} 4 \times 2 + 5 \times (-3) & 4 \times 6 + 5 \times 2 & 4 \times 3 + 5 \times 4 \\ 3 \times 2 + 2 \times (-3) & 3 \times 6 + 2 \times 2 & 3 \times 3 + 2 \times 4 \end{bmatrix}$$

$$C_{2 \times 3} = \begin{bmatrix} -7 & 32 & 34 \\ 0 & 22 & 17 \end{bmatrix}$$

Lower triangular matrix -

In this matrix all the elements above diagonal will be zero or we can say $C_{ij} = 0$ if $i < j$.

$$\begin{bmatrix} C_{11} & 0 & 0 & \dots & 0 \\ C_{21} & C_{22} & 0 & \dots & 0 \\ \dots & \dots & C_{33} & \dots & 0 \\ \dots & \dots & \dots & \dots & 0 \\ C_{n1} & C_{n2} & \dots & \dots & C_{nn} \end{bmatrix}$$

Upper triangular matrix-

In this matrix all the elements below the diagonal matrix will be zero or we can say $C_{ij} = 0$ if $i > j$.

$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1n} \\ 0 & C_{22} & \dots & C_{2n} \\ 0 & 0 & \dots & \\ \dots & & & \\ 0 & 0 & \dots & 0 & C_{nn} \end{bmatrix}.$$

Tridiagonal matrix-

In this matrix all the elements except elements of three diagonal will be zero.

$$\begin{bmatrix} C_{11} & C_{12} & 0 & 0 & 0 & \dots & 0 \\ C_{21} & C_{22} & C_{23} & 0 & 0 & \dots & 0 \\ 0 & C_{32} & C_{33} & C_{34} & 0 & \dots & 0 \\ 0 & 0 & & & & & \\ 0 & \dots & & & & & \\ 0 & \dots & & C_{n-1n-1} & C_{nn-1} & & \\ 0 & 0 & 0 & C_{nn-1} & C_{nn} & & \end{bmatrix}$$

Transpose matrix-

Transpose matrix is defined as the matrix which is obtained by interchanging the rows and columns of a matrix. If a matrix is of $m \times n$ order then its transpose matrix will be of order $n \times m$.

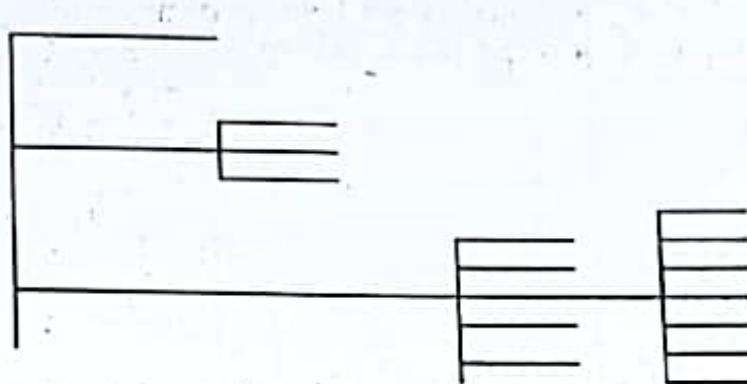
$$B_{2 \times 3} = \begin{bmatrix} 5 & 4 & 3 \\ 9 & 7 & 1 \end{bmatrix}$$

$$B^T = \begin{bmatrix} 5 & 9 \\ 4 & 7 \\ 3 & 1 \end{bmatrix}$$

B^T represents the transpose of matrix B and is of order 3×2 .

Recursion

Recursion is a very powerful technique to write a complicated algorithm in easy way. For solving any big problem we divide it in smaller ones. Then we can group them into problems which have near about same logic. Let us take a simple example of directory structure, if we want to visit any directory structure then we have a need to go by level1, level2,...,level n directory structure and then only we can see the particular file which we want to search. So we can say method for visiting directory structure is same, only levels are different.



So we can use the same logic for all same type of modules. 'C' language supports the technique of recursion which makes it more powerful and unique. The function which call itself (in function body) again and again is known as recursive function. This function will call itself as long as the condition is satisfied.

Ex-

```

main( )
{
    .....
    func();
    .....
}
func()
{
    .....
    .....
    func(); --> recursive call
}
  
```

```

/* Find the factorial of any number */
#include<stdio.h>
main()
{
    int n,value;
  
```

```

printf( "Enter the number: " );
scanf( "%d", &n );
if ( n < 0 )
    printf( "No factorial of negative number\n" );
else
    if ( n == 0 )
        printf( "Factorial of zero is 1\n" );
    else
    {
        value = factorial( n ); /* Function for factorial of number */
        printf( "Factorial of %d = %d\n", n, value );
    }
}

factorial( int k )
{
    int fact = 1;
    if ( k > 1 )
        fact = k * factorial( k-1 ); /* Recursive function call */
    return ( fact );
}

```

After run:

Enter the number: 4
Factorial of 4 = 24

For understanding the recursion process we have a need to understand the work of stack portion of program. Stack contains the address, formal parameter and return value for each function. It behaves as LIFO (Last In First Out) manner. We will see the stack data structure in next chapters.

Let us take a case of finding factorial of number 4. It will call function again and again until the condition when number is greater than 1. After that every called function will return the value to the previous function.

$$\text{Factorial}(4) = 4 * \text{Factorial}(3)$$

$$\rightarrow \text{Factorial}(3) = 3 * \text{Factorial}(2)$$

$$\rightarrow \text{Factorial}(2) = 2 * \text{Factorial}(1)$$

$$\rightarrow \text{Factorial}(1) = 1$$

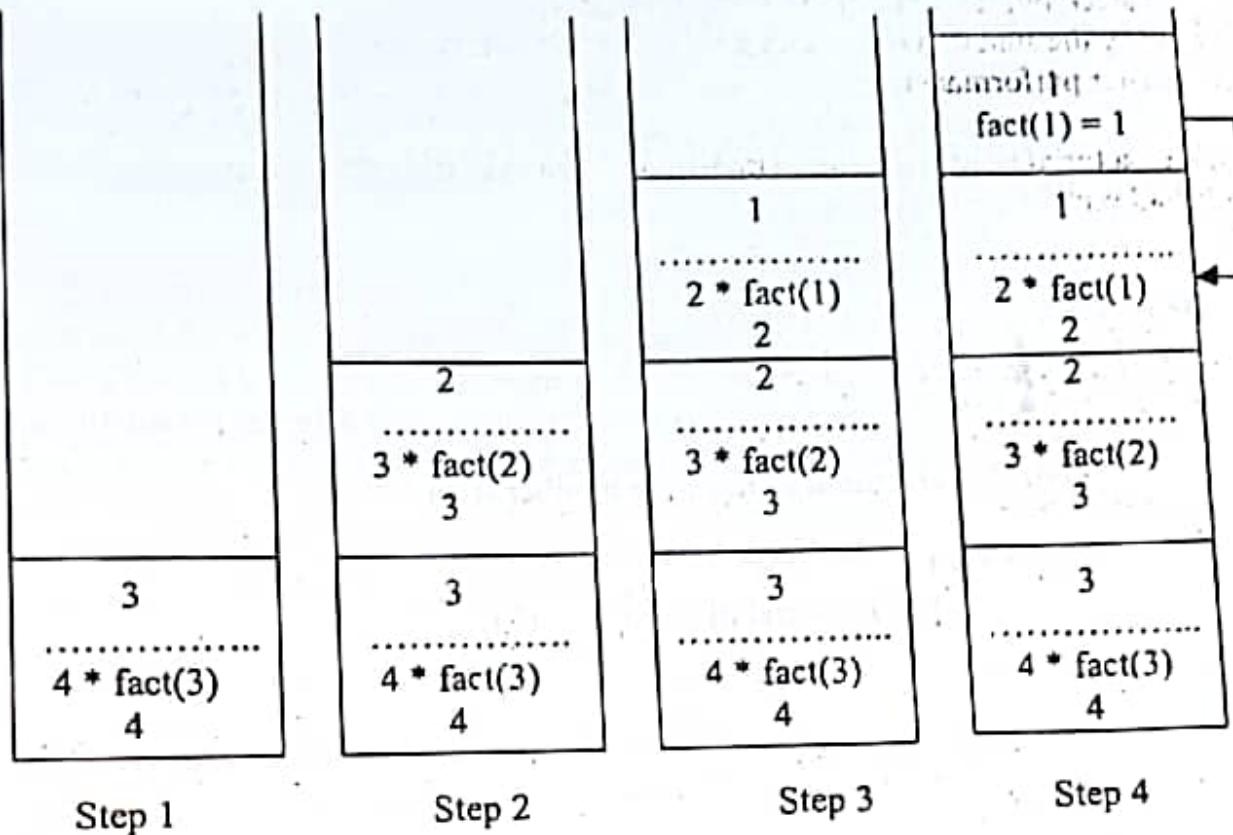
Here the steps for calling function are finished. Now each function will return the value to its previous function.

$$\text{Factorial}(2) = 2 * \text{Factorial}(1) = 2$$

$$\text{Factorial}(3) = 3 * \text{Factorial}(2) = 3 * 2 = 6$$

$$\text{Factorial}(4) = 4 * \text{Factorial}(3) = 4 * 6 = 24$$

The whole process will be implemented in stack as-

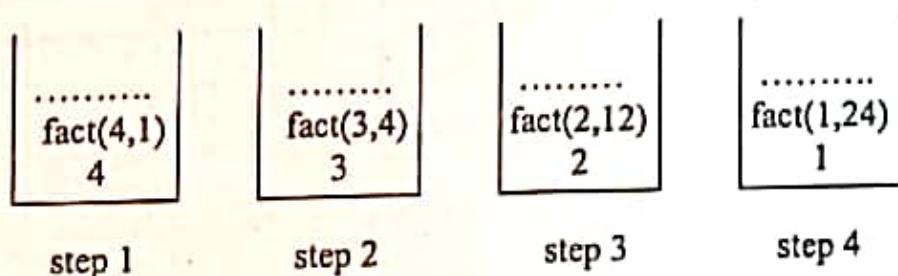


Tail Recursion-

As we know the process in which function calls itself in function body is called as recursion. But when this called function is the last executed statement in the function body then it is called tail recursion. Here we take the return value as one parameter of function itself. We use stack to maintain all the functions but here it will not append new function in stack but it will overwrite the value of previous function with the current one. So we can say the function call time and stack implementation time will be reduced and it will give better performance.

```
/* Find the factorial of any number with the use of tail recursion */
#include<stdio.h>
main( )
{
    int n,value;
    printf( "Enter the number:" );
    scanf( "%d", &n );
    if( n < 0 )
        printf( "No factorial of negative number\n" );
    else
        if( n == 0 )
            printf( "Factorial of zero is 1\n" );
        else
            {
                value = factorial( n;1 ); /* Function for factorial of number */
                printf( "Factorial of %d=%d\n",n,value );
            }
}
factorial( int n, int fact )
{
    if( n == 1 )
        return fact;
    else
        factorial( n-1, n*fact );
}
```

This will be implemented in stack as-



Here we can see tail recursion takes only 4 steps for getting factorial of number 4. It reduces space and time both and improves the performance.

Recursion through stack-

Some programming languages like 'C' provide the facility of recursive function. So we can use it very easily. But in other languages also we can provide recursion technique with stack implementation whenever the situation arises for recursion implementation. Even we can convert recursive function in 'C' into non recursive function through stack implementation. We will see this approach in graph traversal where we use Depth first traversal using recursion as well as stack implementation. We have a need to do following things for using recursion through stack implementation.

1. One stack for each parameter of function.
2. Only one stack for return address.
3. One stack for each local variable of function.

We have a need to take right decision where to use recursion function or stack implementation because most of the places using stack implementation will be complex and takes more execution time but in some situations it can be more useful than recursion also.

Removal of recursion through iteration-

We can use iteration to replace recursion. Let us take a simple example of factorial through recursion and as well as through iteration.

```
int factorial( int no )
{
    int fact = 1;
    if ( no > 1 )
        fact = no * factorial( no - 1 );
    return fact;
}
```

Suppose we want to get the factorial of 4 then the whole process will be as-

```
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1
```

Now values will return back to the previous called function as-

```
factorial(2) = 2 * 1 = 2
factorial(3) = 3 * 2 = 6
factorial(4) = 4 * 6 = 24
```

Here we can see the whole process takes 7 steps. Getting values in stack, calculation and returning values to stack is also time consuming and takes extra spaces. Same thing can be replaced with iteration as-

```
int factorial( int no )
{
    int i, fact = 1;
    for ( i = no; i > 1; i-- )
        fact = fact * i;
    return fact;
}
```

Here we can see recursion is replaced very easily with iteration. Here iteration takes less steps, space and time also.

TOWER OF HANOI-

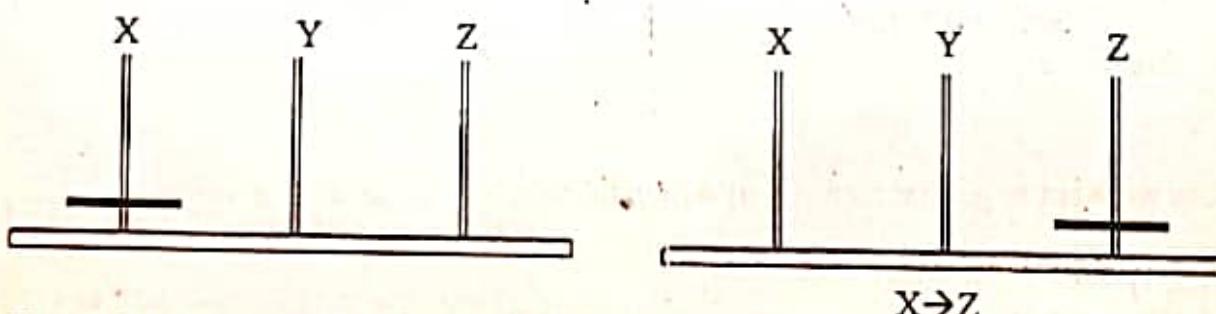
The problem of Tower of Hanoi is to move the disk from one pillar to another with the use of a temporary pillar. Let us take these pillars are X and Y. We want to move the disk from X to Y with the use of TEMP pillar. The conditions for this game are-

1. We can move only one disk from one pillar to another at a time.
2. Larger disk can not be placed on smaller disk.

Let us take the solution of Tower of Hanoi problem for n disk

For n = 1

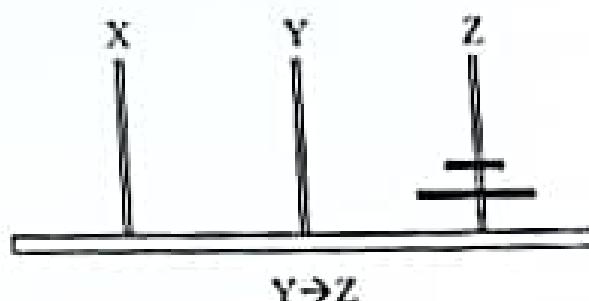
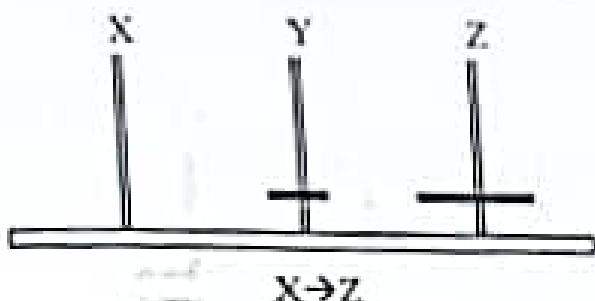
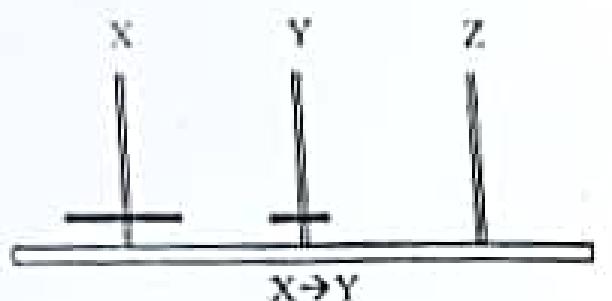
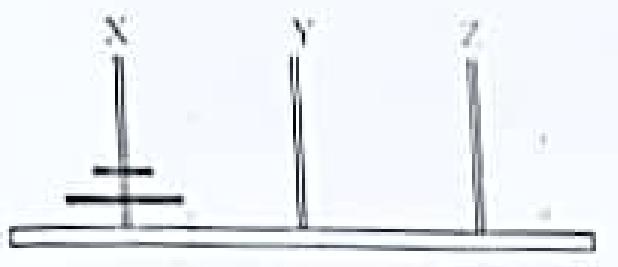
Move disk from Source to Destination



Here X is the source, Z is the destination pillar and Y is the temporary pillar.
 $X \rightarrow Z$ means move the disk from pillar X to pillar Z.

For n = 2

Move top disk from pillar X to Y	$(X \rightarrow Y)$
Move top disk from pillar X to Z	$(X \rightarrow Z)$
Move top disk from pillar Y to Z	$(Y \rightarrow Z)$

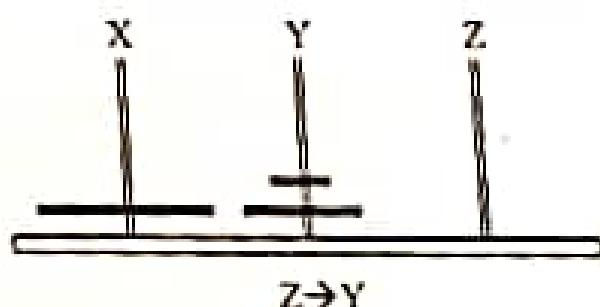
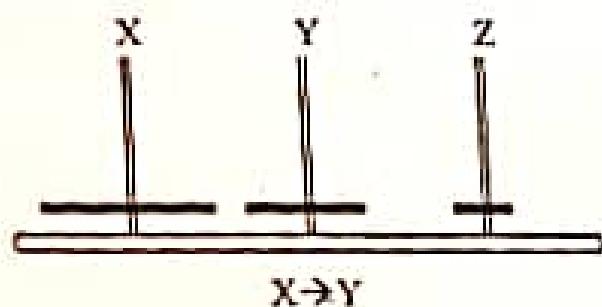
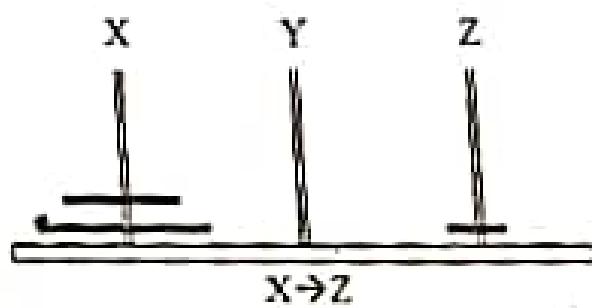
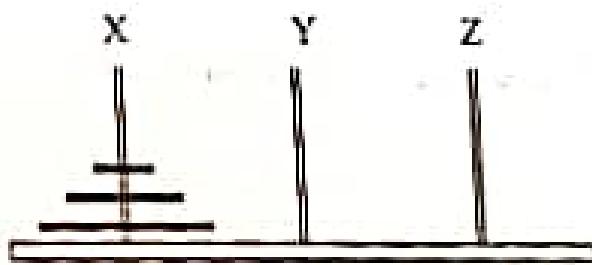


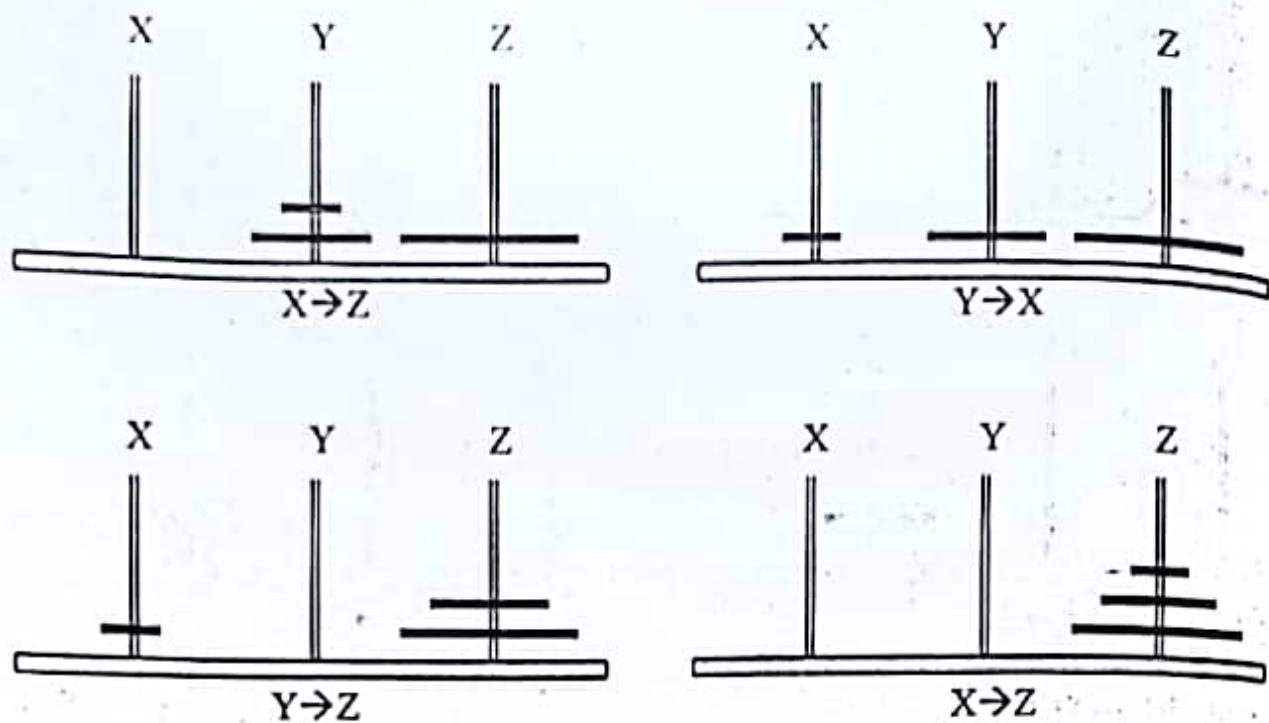
Hence the sequence is-

$X \rightarrow Y$, $X \rightarrow Z$, $Y \rightarrow Z$

For $n=3$

- Move top disk from pillar X to Z
- Move top disk from pillar X to Y
- Move top disk from pillar Z to Y
- Move top disk from pillar X to Z
- Move top disk from pillar Y to X
- Move top disk from pillar Y to Z
- Move top disk from pillar X to Z





Hence the sequence is-

$X \rightarrow Z, X \rightarrow Y, Z \rightarrow Y, X \rightarrow Z, Y \rightarrow X, Y \rightarrow Z, X \rightarrow Z$

Now the problem arises for taking the general solution for n disk. From previous example we see that first we move n-1 disk to another place and then move the largest disk to destination. Hence we can say the problem is-

1. Move upper n-1 disk from Source to Temp.
2. Move largest disk from Source to Destination.
3. Move n-1 disk from Temp to Destination.

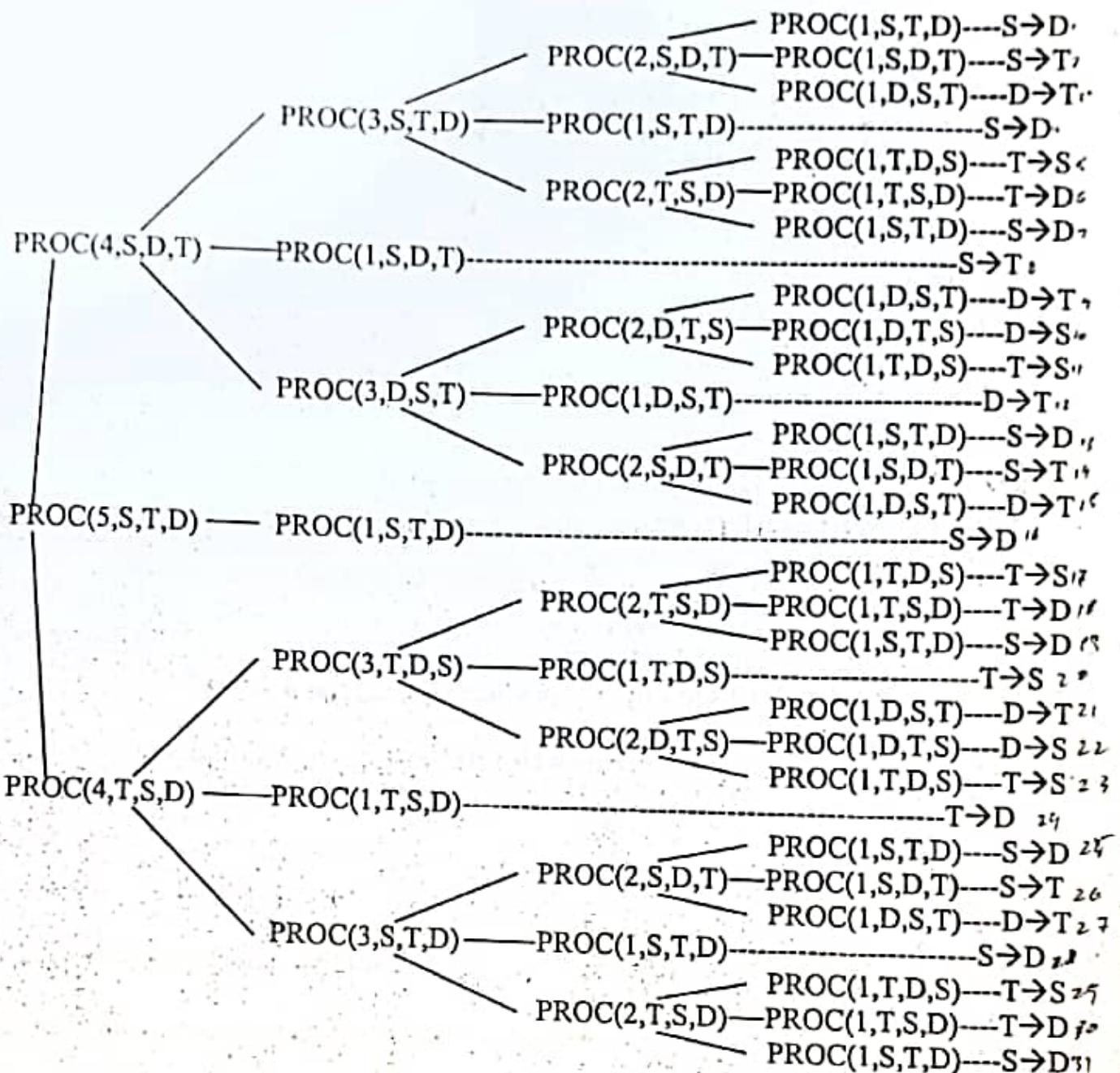
Now we can take the process 3 until only one disk is remaining. So the procedure is-

1. PROC(N-1, S, D, TEMP)
2. PROC(1, S, TEMP,D)
3. PROC(N-1,TEMP,S,D)

Let us take n=5, hence the solution is-

The sequence is-

$S \rightarrow D, S \rightarrow T, D \rightarrow T, S \rightarrow D, T \rightarrow S, T \rightarrow D, S \rightarrow D, S \rightarrow T, D \rightarrow T$
 $D \rightarrow S, T \rightarrow S, D \rightarrow T, S \rightarrow D, S \rightarrow T, D \rightarrow T, S \rightarrow D, T \rightarrow S, T \rightarrow D$
 $S \rightarrow D, T \rightarrow S, D \rightarrow T, D \rightarrow S, T \rightarrow S, D \rightarrow T, D \rightarrow S, T \rightarrow S, T \rightarrow D$
 $S \rightarrow D, S \rightarrow T, D \rightarrow T, S \rightarrow D, T \rightarrow S, T \rightarrow D, S \rightarrow D$



```

/* Program for solution of Tower of Hanoi*/
#include<stdio.h>
main()
{
    char source= 'S',temp= 'T ', dest= 'D';
    int ndisk;
    printf("Enter the number of disks : ");
    scanf( "%d", &ndisk );
    printf ("Sequence is :\n");
    toh( ndisk, source, temp, dest );
}

```

```

toh( int ndisk, char source, char temp, char dest )
{
    if( ndisk > 0 )
    {
        toh( ndisk-1, source, dest, temp );
        printf( "Move Disk %d %c-->%c\n", ndisk, source, dest );
        toh( ndisk-1, temp, source, dest );
    }
}/*End of toh()*/

```

Exercise-

1. Write a program to calculate sum of following series-

3,5,7,9,11,.....:.....95,97

2. Write a program to calculate factorial of given number from iteration, simple as well as through tail recursion depending on user choice.
3. Write a program of fibonacci series which starts and ends with given number.
4. Show the figures of steps to solve the problem tower of hanoi for 3 disk.
5. Write a program of pattern matching which start matching pattern from end of string.
6. Write a program to reverse the string.
7. Write a function Concatrev(str1,str2) which concatenate the reverse of str2 into str1.
8. Write a program to convert the decimal number to binary with iteration and recursion.
9. Write a program to accept a string as input and display all the lines where input string exists for a specific file.

Chapter 2

Arrays, Pointers and Structure

What is Array-

In many situations, it may be possible that it is useful to collect the similar type of data items. C supports concept of array for this purpose. An array is a collection of similar type of data items.

Previously we declared a variable for taking any value, but the variable can hold only one value at a time. Problem arises when we want to store the age of 50 employees or elements of any matrix.

Let us take the example of employee records-

1. Declare 50 different variables to store the age of employees.
2. Assign values to each variable.

It is also not easy to handle these variables in the program.

The concept of array is useful in this situation. An array is a collection of similar type of data items. The type of data items may be char, int or float. The elements of array share the same variable name.

The elements of array are indicated by specifying the array name followed by a subscript in brackets. Let us take an array-

arr[10];

The elements of this array are-

arr[0], arr[1], arr[2].....arr[9]

Here number of subscripts determines the dimension of array. The value in the first bracket is size of array. arr[1] is the second element of the array because in C we start counting from 0.

Hence if the size of array is n, for example-

arr[n];

then arr[0] is the first element and arr[n-1] is the last element of the array.

DECLARATION OF ARRAY

As other variables, we can also declare the array. The array should be declared with the data type, array name and number of subscript in the bracket. By declaring an array, the number of memory locations (size of array) is allocated in memory.

The syntax for declaration of array is-

Storage_class datatype array_name[expression];

Here storage_class may be auto, static or extern, which tells the scope of the array

variable and expression may be a positive constant or constant expression which gives the size of the array. If the storage class is not given then compiler assumes it is an auto class.

The array can be declared as-

```
int age[10];
float sal[10];
char grade[10];
```

Here first one is integer type array. Every element of array can hold an integer value. Second one is the floating type array, can hold a float value and third one is the character type array, can hold one character.

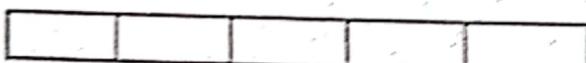
PROCESSING WITH ARRAY

We know the elements of array is stored in contiguous memory locations. For example-

```
int arr[5];
```

This is stored in memory as-

100 102 104 106 108



arr[0] arr[1] arr[2] arr[3] arr[4]

We can take the value in the array as other variables by scanf() as example-

```
scanf( "%d", &arr[1] );
```

By this statement we can take the value in the 2nd element of array.

Similarly we can print the value of the array element by printf() statement.

For example-

```
printf( "%d", arr[1] );
```

By this statement we can print the value of 2nd element of array.

We can not take the value in array which is out of range(size of array).

We can take the value in character type array as-

```
scanf( "%c", addr );
```

Here name of the array addr gives the starting address of character type array.

Similarly we can print the character type array by printf() statement as-

```
printf( "%s", addr );
```

STRING LIBRARY FUNCTIONS-

strlen()

This function returns the length of string i.e. number of characters in the string. strlen("saurabh") return the value 7.

If s1 is the array that contains the name "madhu" then strlen(s1) returns the value 5.

strcmp()

This function is used for comparison of two strings, if the two strings match, strcmp() would return a value 0, otherwise it would return a non-zero value. This function compares the strings character by character.

strcmp(s1,s2) returns a value-

- < 0 when s1 < s2
- = 0 when s1 == s2
- > 0 when s1 > s2

strcpy()

This function is used for copying of one string to another string.

strcpy(str1,str2) copy str2 to str1. Here str2 is the source string and str1 is destination string. If str2 = "rajesh" then this function will copy "rajesh" into str1.

strcat()

This function is used for concatenation of two strings. If first string is "ran" and second string is "jana" then after using this function the resultant string is "ranjana".

strcat(str1,str2) concatenates str2 at the end of str1.

INITIALIZATION OF ARRAY-

Array can also be initialized at the time of declaration. The syntax for initialization of array is-

Storage_class data_type array_name[size] = { value1, value2,value n };

Here storage_class refer to scope of array variable, data_type is the type of array, array_name is the name of array variable, size is the size of array. value1,value2value n are values which is assigned in array one after another.

For example-

```
int marks[5] = { 50, 85, 70, 65, 95 };
float sal[4] = { 2000, 1500.50, 315.625, 1000 };
```

The values assigned in array are as-

marks [0]	[1]	[2]	[3]	[4]
-----------	-----	-----	-----	-----

50	85	70	65	95
----	----	----	----	----

1000	1002	1004	1006	1008
------	------	------	------	------

sal [0] [1] [2] [3]

2000	1500.50	315.625	1000
------	---------	---------	------

4000 4004 4008 4012

The character array is also called string because it's a collection of characters.

In character type array, it is necessary that it should be ended with '\0'. This is automatically added by compiler at the end of character type array. As other array variables we can also initialize the character type array. For example-

char name[6] = "madhu";

The values assigned in array are as-

name [0] [1] [2] [3] [4] [5]

m	a	d	h	u	'\0'
---	---	---	---	---	------

1000 1001 1002 1003 1004 1005

The character type array can also be initialized as-

char name[] = "Atul";
 char name[] = {'d', 'i', 'v', 'y', 'a', '\0'};

USE OF ARRAY IN FUNCTION-

Previously we pass the variables in the function, similarly we can also pass the whole array as actual parameter through function, but the formal parameter should be declared as array variable of same data type as-

```
main()
{
    int arr[10];
    .....
    .....
    func( arr );
}
func( int val[ ] )
{
    .....
    .....
}
```

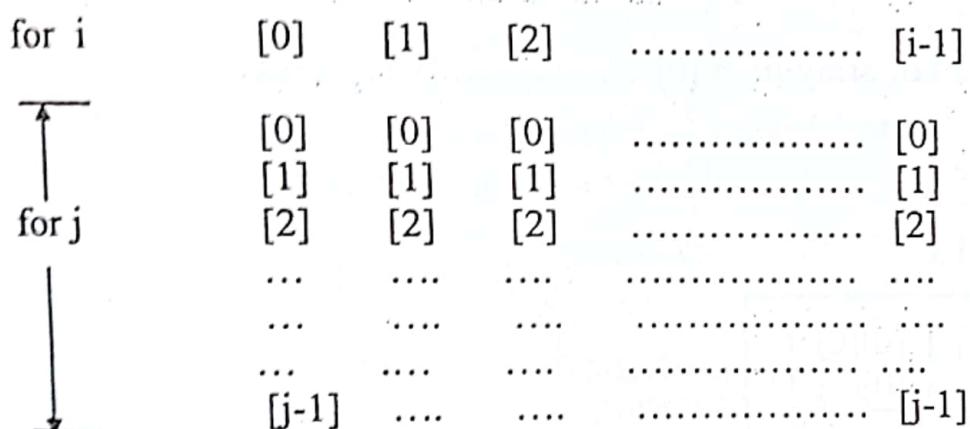
MULTIDIMENSIONAL ARRAY-

As we know one dimensional arrays use one bracket, two dimensional arrays use two brackets, three dimensional arrays use three brackets. Similarly n-dimensional arrays use n brackets. As example-

$\text{arr}[i][j]$ means the starting element of array is $\text{arr}[0][0]$ and the last element of array is $\text{arr}[i-1][j-1]$.

Here in $\text{arr}[i][j]$, i denotes the row in the array and j denotes the column in array or elements in each row.

As example-



Here in first bracket each element stores j elements, hence the total number of elements
 $= i * j$

In array $\text{arr}[i][j]$, elements of first bracket [0].....[i-1] have the address of row.

Ex-

```
int marks[5][3]
```

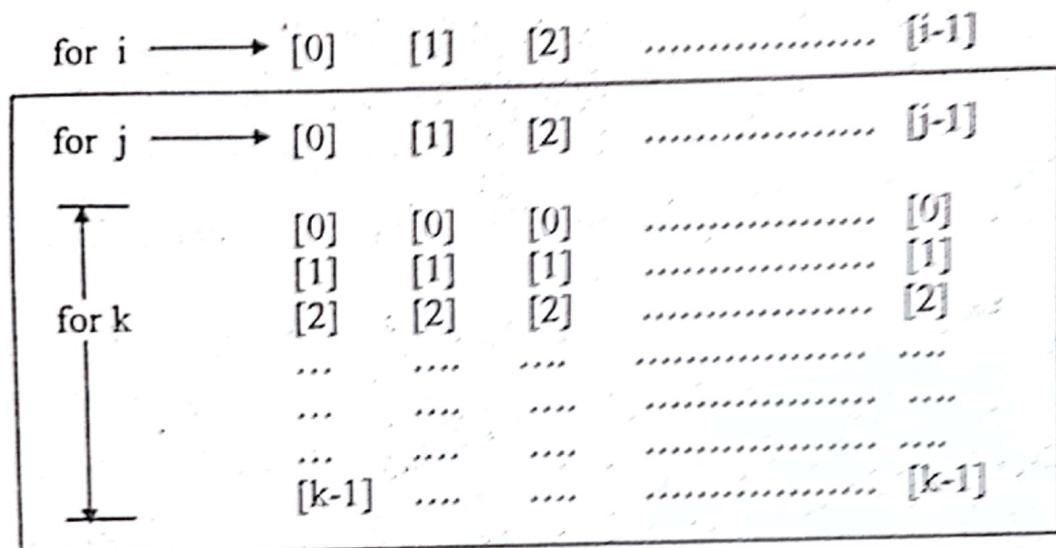
Here in array $\text{marks}[5][3]$, marks of five student in three subjects are stored as-

marks of student	[0][0]	[1][0]	[2][0]	[3][0]	[4][0]
	[0][1]	[1][1]	[2][1]	[3][1]	[4][1]
	[0][2]	[1][2]	[2][2]	[3][2]	[4][2]
	1st	2nd	3rd	4th	5th

Similarly for three dimensional array-

Ex-

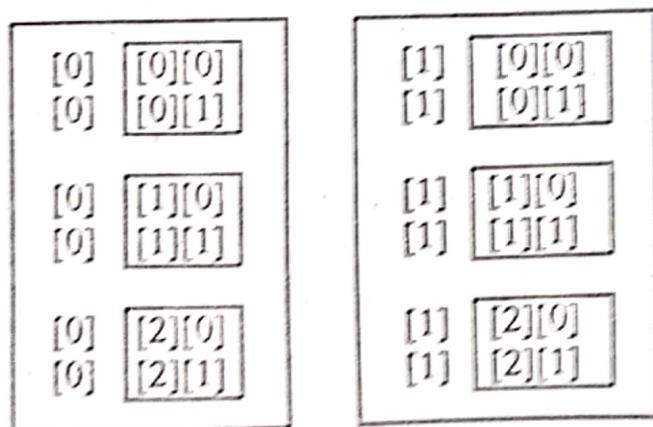
```
arr[i][j][k]
```



Here each element of first bracket of array from [0] [i-1] stores the value of two dimensional array.

Ex-

int arr[2][3][2] are stored as-



Total number of elements in $\text{arr}[i][j][k]$

$$= i * j * k$$

Similarly total number of elements in n -dimensional array $\text{arr}[] [] [] \dots []$

$\longleftrightarrow n \longleftrightarrow$

n = multiplication of number given in each bracket

POINTERS

Provision for pointers is one of the most powerful and useful feature of the C language, which distinguishes it from other languages. The concept of pointers is somewhat complex but is simple when compared to its advantages. Pointer is more difficult to understand but the proper use of it makes the C language excellent.

WHAT IS POINTER-

A pointer is a variable. It contains the memory address (locations) of another variable which is declared as the pointer type (It doesn't take the value of the variable). As an example if one variable is a data type and second variable is the pointer type which points to the first variable's type, then the content of second variable is the address of first variable.



Here the second variable contains the address of first variable. 5 is the value at address 2000, which is the address of first variable.

USES OF POINTERS-

1. Pointers are used for saving memory space.
2. Use of pointer, assigns the memory space and also releases it. This concept helps in making the best use of the available memory (dynamic memory allocation).
3. With pointers, data manipulation is done with address, so the execution time is faster.
4. Two-dimensional and multi-dimensional array representation is easy with pointers.
5. Concept of pointer is used in data structures such as linked list.

THE & AND * OPERATOR-

The "&" is the address operator, it represents the address of the variable.

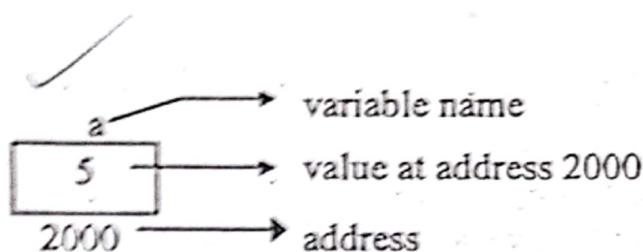
Ex:-

```
#include< stdio. h>
main ()
{
    int a = 5;
    printf (" value of a = %d\n",a);
    printf (" address of a = %u\n",&a);
}
```

The "%u" is used for obtaining the address.

Output:

```
value of a = 5
address of a = 2000
```



Here 'a' is the variable name, 2000 is the address of variable and 5 is the value at address 2000.

The '*' operator is the value at address operator. It represents the value at the specified address.

Ex2-

```
#include<stdio.h>
main()
{
    int a = 5;
    printf("value of a = %d\n", a);
    printf("address of a = %u\n", &a);
    printf("value at address %u = %d\n", &a, *(&a));
}
```

Output:

{ value of a = 5
 address of a = 2000
 value at address 2000 = 5 }

Here '&' is the address of variable 'a' and preceded by *, *(&a)=a means the value at address of variable.

DECLARATION OF POINTER-

Let us see, how we use the pointer in the expression. The & operator represents the address of variable. If we want to give the address of variable to another variable then we can write as-

b = &a;

Here 'b' is the variable which contains the address of the variable 'a'.



Here value of 'a' is 5 and 'b' has the value 2000 which is address of 'a'.

When a pointer variable is declared then an asterisk (*) symbol should precede the variable name. Here b is a pointer type variable, which points to the variable a. Hence it must be declared as -

```
int a = 5;
int *b;
```

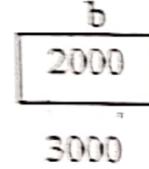
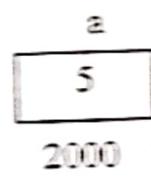
Here 'b' is an int type of pointer. Value at address contained in 'b' is an int. The address of any variable is a whole number. Pointer variable contains the address of any variable, hence pointer variable always contains the whole number. Similarly we can declare character and floating type pointer-

```
char *p;
float *q;
```

char *p means, 'p' contains the address of the variable, which is of character type variable. float *q means, 'q' contains the address of the variable which is of floating type.

Ex 3-

```
#include< stdio. h>
main ()
{
    int a = 5;
    int *b;
    b = &a;
    printf ("value of a = %d\n",a);
    printf ("value of a = %d\n",*( &a));
    printf ("value of a = %d\n",*b);
    printf ("address of a = %u\n",&a);
    printf ("address of a = %u\n",b);
    printf ("address of b = %u\n",&b);
    printf (" value of b = address of a = %u",b);
}
```

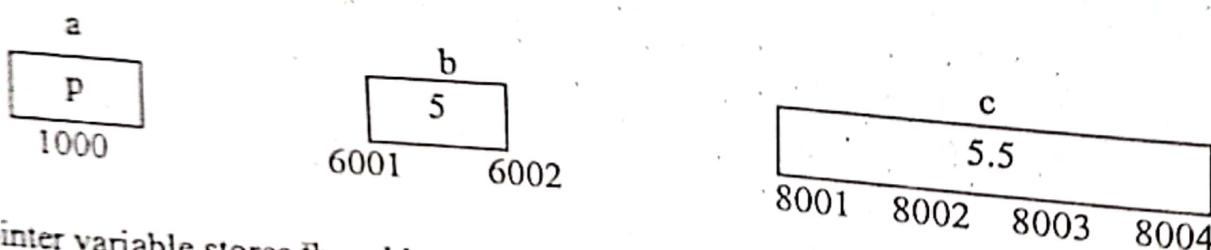


Output-

```
value of a = 5
value of a = 5
value of a = 5
address of a = 2000
address of a = 2000
address of b = 3000
value of b = address of a = 2000
```

Ex 4 -

```
#include< stdio.h>
main()
{
    char a = 'p';
    char *a1;
    int b = 5;
    int *b1;
    float c = 5.5;
    float *c1;
    a1 = &a;
    b1 = &b;
    c1 = &c;
    printf (" value of a = %c\n",a);
    printf (" value of a = %c\n",*( &a)); → value at address
    printf ("value of a = %c\n",*a1);
    printf ("value of b = %d\n",b);
    printf ("value of b = %d\n",*(&b));
    printf ("value of b = %d\n",*b1);
    printf ("value of c = %f\n",c);
    printf ("value of c = %f\n",*(&c));
    printf ("value of c = %f\n",*c1);
    printf ("address of a = value of a1 = %u\n",a1);
    printf ("address of b = value of b1 = %u\n",b1);
    printf (" address of c = value of c1 = %u\n",c1);
}
```



Pointer variable stores the address of the first byte. So in case of int pointer, it stores the address 6001 and in float pointer it stores the address 8001.

Output-

```
value of a = p
value of a = p
value of a = p
value of b = 5
value of c = 5.5
value of c = 5.5
value of c = 5.5
address of a = value of a1 = 1000
```

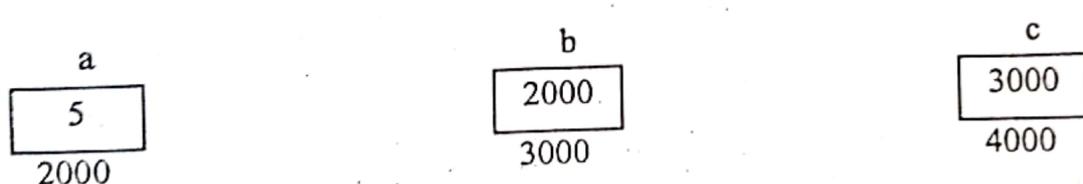
address of b = value of b1 = 5001
 address of c = value of c1 = 8001

POINTER TO POINTER -

We know that pointer is a variable that contains the address of another variable. Similarly another pointer variable can store the address of this pointer variable. Hence we can say this is a pointer to pointer variable. This concept can be extended.

Ex 5-

```
#include< stdio.h>
main ()
{
    int a = 5;
    int *b;
    int **c;
    b = &a;
    c = &b;
    printf ("value of a = %d\n",a);
    printf (" value of a = %d\n",*(&a));
    printf (" value of a = %d\n",*b);
    printf (" value of a = %d\n",**c);
    printf (" value of b = address of a = %u\n",b);
    printf (" value of c = address of b = %u\n",c);
    printf (" address of a = %u\n",&a);
    printf (" address of a = %u\n",b);
    printf (" address of a = %u\n",*c);
    printf (" address of b = %u\n",&b);
    printf (" address of b = %u\n",c);
    printf (" address c = %u\n",&c);
}
```



Here 'b' is a pointer variable which contain the address of the variable 'a'. 'c' is a pointer to pointer variable, which contains the address of the pointer variable 'b'.

Output -

```
value of a = 5
value of b = address of a = 2000
```

value of c = address of b = 3000

address of a = 2000

address of a = 2000

address of a = 2000

address of b = 3000

address of b = 3000

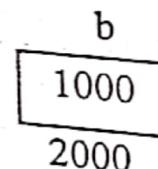
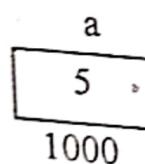
address of c = 4000

POINTER ARITHMETIC -

Arithmetic operations can also be done with the pointer variable. Postfix, prefix, increment and decrement operations are also possible with the pointers. In other variables postfix, prefix, increment, decrement means addition or subtraction by 1 with these variables, but here the postfix, prefix, increment, decrement means addition or subtraction of bytes that pointer data type holds; with the value that the pointer variable contains.

Ex -

```
int a = 5;
int *b;
b = &a;
```



Here $b++$ or $++b$ means $1000 + 2 = 1002$

and $b--$ or $--b$ means $1000 - 2 = 998$

It doesn't affect the address of 'a'.

In other variables

Ex -

```
int a = 5;
a++ or a++ means 5+1=6
and a-- or --a means 5 - 1 = 4
```

Ex 6-

```
/* program to understand the postfix, prefix, increment, decrement in the pointer variable */
#include< stdio. h>
main ()
{
    int a = 5;
    int *b;
    b = &a;
```

```

printf (" address of a = %u\n",b);
printf (" value of b = %u\n",++b);
printf (" value of b = %u\n",b++);
printf (" value of b = %u\n",--b);
printf (" value of b = %u\n",b--);
printf (" value of b = %u\n",b);
}

```

**Output -**

```

address of a = 1000
value of b = 1002
value of b = 1000

```

Explanation -

In first printf the address of a = 1000. In second prefix increment is done with the pointer variable 'b'. So first it is incremented then print the value. Since pointer is int type, hence it is incremented by 2. Hence the address of a = 1002. b++ means first print the value of 'b' then increment 'b' by 2. Now 'b' has the value 1004. --b means first decrement it by 2 then print the value of 'b' that is 1002. b-- means print the value of b then decrement it by 2. Hence at last value of b = 1000.

Similarly the postfix/prefix, increment/decrement operation can be done with the char and float pointer but here 1 is added to or subtracted from the char pointer and 4 is added to or subtracted from the float pointer.

Ex 7-

```

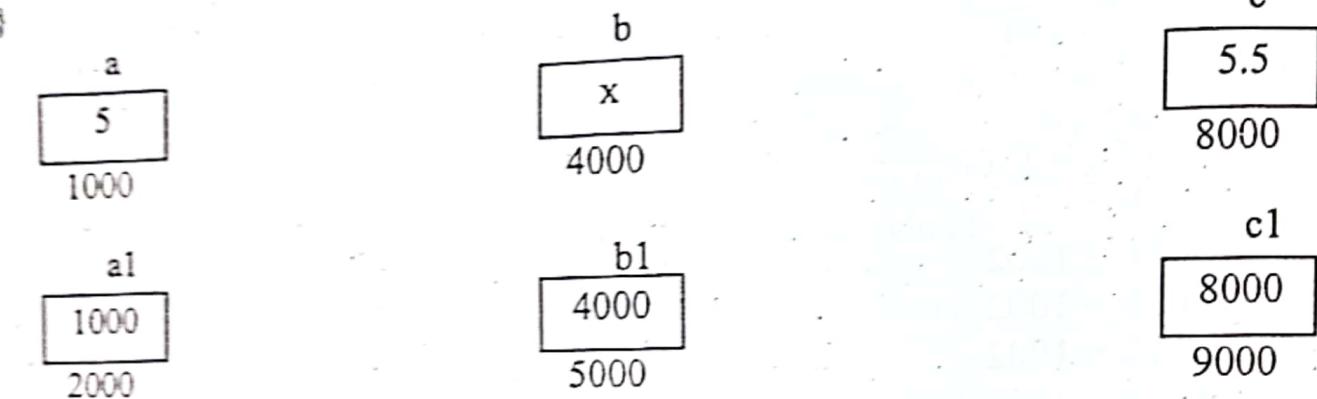
#include< stdio.h>
main ()
{
    int a = 5;
    int *a1;
    char b = 'x';
    char *b1;
    float c = 5.5;
    float *c1;
    a1 = &a;
    b1 = &b;
    c1 = &c;
}

```

```

printf (" address of a = value of a1 = %u\n", a1);
printf (" address of b = value of b1 = %u\n", b1);
printf (" address of c = value of c1 = %u\n", c1);
a1++;
b1++;
c1++;
printf (" Now value of a1 = %u\n", a1);
printf (" Now value of b1 = %u\n", b1);
printf (" Now value of c1 = %u\n", c1);
}

```



$$a1++ = 1000 + 2 = 1002$$

$$b1++ = 4000 + 1 = 4001$$

$$c1++ = 8000 + 4 = 8004$$

Since int holds 2 bytes, char holds 1 byte and float hold 4 bytes.

Output -

address of a = value of a1 = 1000

address of b = value of b1 = 4000

address of c = value of c1 = 8000

Now value of a1 = 1002

Now value of b1 = 4001

Now value of c1 = 8004

These are the operations that can be done with the pointer.



(I) Addition of a number to a pointer variable.

Ex -

```

int a = 5;
int *b;
b = &a;
b = b + 1;
b = b + 2;

```

But here addition means bytes, that pointer data type holds are added number of times that is added to the pointer variable.

$$b = b + 1 = 1000 + 2 = 1002$$

$$b = b + 2 = 1000 + 2 * 2 = 1000 + 4 = 1004$$

(II) Subtraction of a number from a pointer variable.

```
int a = 5;
int *b;
b = &a;
b = b - 1;
b = b - 3;
```

Here subtraction means bytes that pointer data type hold are subtracted number of times that is subtracted from the pointer variable.

$$b = b - 1 = 1000 - 2 = 998$$

$$b = b - 3 = 1000 - 3 * 2 = 1000 - 6 = 994$$



(III) Subtraction of one pointer from another pointer.

This operation is done when the both pointer variables point to the elements of the same array.

These operations can never be done with pointers and gives error -

1. Addition between two pointers.
2. Multiplication between pointer and any number.
3. Division between pointer and any number.

POINTER AND FUNCTIONS -

The arguments or parameters to the functions are passed in two ways –

- ✓ 1. Call by value
- ✓ 2. Call by reference

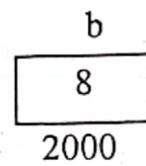
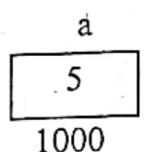
In call by value the values of the variables are passed. In this values of variables are not affected by changing the value of the formal parameter.

Ex 8 -
/* program to explain call by value */
#include< stdio.h>

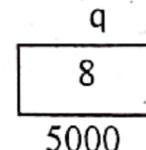
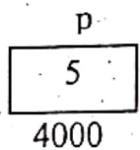
```
main ()
{
    int a = 5;
    int b = 8;
    printf( " Before calling the function a and b are %d , %d\n",a,b);
    value( a , b );
    printf( " After calling the a and b are %d , %d\n",a,b);
}

value ( p , q )
int p , q;
{
    p++;
    q++;
    printf( " In function changes are %d , %d\n", p , q);
}
```

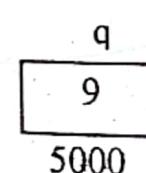
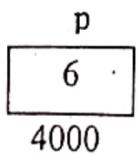
Before calling function -



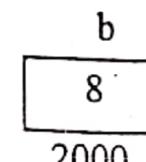
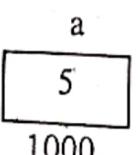
In function -



After incrementing p and q -



After calling function -



Before calling the function value (), the value of a = 5 and b = 8. The variables 'a' and 'b' are passed as the parameter in the function value (). The value of 'a' and 'b' are passed to 'p' and 'q'. But the memory locations are different from the memory location of 'a' and 'b'. Hence when the value of 'p' and 'q' are incremented , there will be no effect on the value of 'a' and 'b'. So after calling the function 'a' and 'b' are same as before calling the function and have the value 5 and 8.

Output -

Before calling the function a and b are 5 , 8

In function changes are 6 , 9

After calling the function a and b are 5 , 8

In call by reference the address of the variables are passed . In this value of variables are affected by changing the value of formal parameter.

Ex 9-

```
/* program to explain call by reference */
#include< stdio.h>
```

```
main ()
{
    int a = 5;
    int b = 8;
    printf (" Before calling the function a and b are %d , %d\n",a,b);
    ref ( &a , &b);
    printf (" After calling the function a and b are %d , %d\n",a,b);

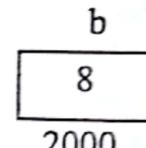
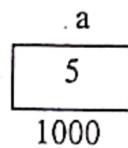
}

ref( p , q )
int *p,*q;
{
    (*p)++;
    (*q)++;
    printf (" In function changes are %d , %d\n" , *p,*q);
}
```

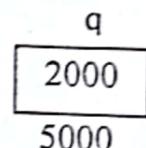
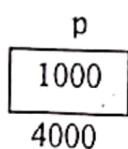
Before calling the function ref (), the value of a = 5 and b= 8. The address of the variables 'a' and 'b' are passed as the parameter in the function ref (). The address of 'a' and 'b' are passed to 'p' and 'q', which is a pointer variable. (*p)++ means value at address is incremented . So the value at address 1000 which is 5,incremented. Similarly (*q)++ means value at address 2000, which is 8,incremented. Now the value of *p = 6 and *q= 9.

Here the address is not changed but value at this address is changed. Hence after calling the function the values of the variables 'a' and 'b' are changed.

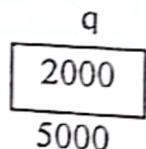
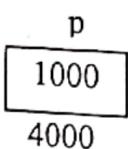
Before calling function -



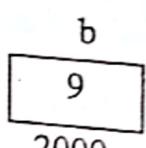
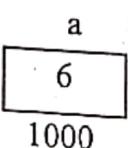
In function -



After (*p)++ and (*q)++ -



After calling function -



Output -

Before calling the function a and b are 5 , 8

In function changes are 6 , 9

After calling the function a and b are 6 , 9

Ex 10-

```
/* program to interchange the value of two variable from call by value */
#include< stdio.h>
main ()
{
    int a = 5;
    int b = 8;
    printf (" Before swapping a = %d , b = %d\n",a,b);
    swap( a , b);
    printf (" After calling swap function a = %d , b = %d\n",a,b);
}
```

```

swap( p , q)
int p , q;
{
    int temp;
    temp = p;
    p = q;
    q = temp;
    printf (" In swapping function p = %d , q = %d\n",p,q);
}

```

Output-

Before swapping a = 5 , b = 8
 In swapping function p = 8 , q = 5
 After calling swap function a = 5 , b = 8

We can see from call by value that we can interchange the value of the variables only in function.

Ex 11-

```

/* program to interchange the value of variables from call by reference */
#include< stdio.h>
main ()
{
    int a = 5;
    int b = 8;
    printf (" Before swapping a = %d , b = %d\n",a,b);
    swap ( &a,&b );
    printf (" After calling swap function a = %d , b = %d\n",a,b);
}

swap ( p , q )
int *p,*q;
{
    int temp;
    temp = *p;
    *p = *q;
    *q = temp;
    printf (" In swapping function p = %d , q = %d\n",*p,*q);
}

```

Output-

Before swapping a = 5 , b = 8
 In swapping function p = 8 , q = 5
 After calling swap function a = 8 , b = 5

We can see from call by reference that we can interchange the value of variable in function. Now the value of variables will also be interchanged after calling the function.

We can also declare the pointer function which can return an address . But we should declare the function as a pointer data type in the main function.

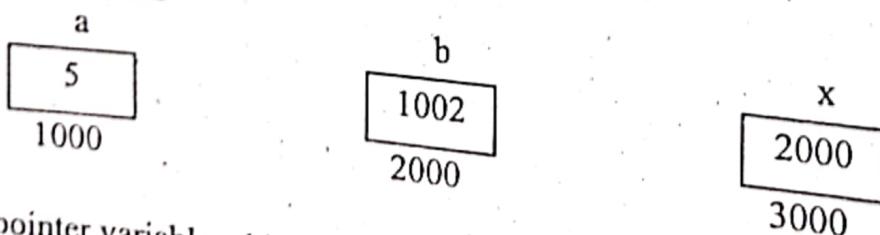
Ex 12-

```
#include<stdio.h>
main ()
{
    int a = 5;
    int *b;
    int *fun( );
    b = &a;
    printf (" Before calling the function address of a = %u\n",b);
    b = fun( &b );
    printf (" After calling the function value of b = %u\n",b);
}

int *fun(int **x)
{
    *x = *x + 1;
    return (*x);
}
```



After calling function -



Here 'b' is a pointer variable which holds the address of 'a'. fun() is a pointer data type function. The address of 'a' is passed to the function fun(). The address of 'a' is passed to the formal parameter 'x' in the function fun(). The value at address 2000 which is 1000 is incremented by 2, since 'x' is an int pointer. Hence after calling the function address is 1002.

Output-

Before calling the function address of a = 1000
After calling the function value of b = 1002

POINTER AND ARRAY-

Array is a collection of similar data type elements. When we declare an array then consecutive memory locations are allocated to array elements. The base address of array is the address of 0th element of array. The name of array also gives the base address of array.

For example-

```
int arr[4] = { 5 , 10 , 15 , 20 }
```

Here arr[4] means that array 'arr' has 4 elements and is of int data type.

arr[0]	arr[1]	arr[2]	arr[3]
1000	1002	1004	1006

Here the base address of array is the address of 0th element of array. Since the array is of type int, hence address of the next element of array is incremented by 2.

Following are the two main points to understand the concept of pointer with array.

1. Elements of array are stored in the consecutive memory locations.
2. When an array pointer is incremented it refers to the next location of its data type.

Let us take an example-

```
int arr[4] = {5 , 10 , 15 , 20 };
```

```
int *a;
```

```
a = arr;
```

This shows 'a' is a pointer variable which holds the base address of the array arr or we can say

```
a = &arr[0]
```

When pointer variable is incremented by 1 then it contains the base address+2 because pointer is of type int. The address of next element will be also base address+2 because elements of array are stored in consecutive memory locations. So we can say when an array pointer is incremented it refers to next immediate location of array.

arr[0]	arr[1]	arr[2]	arr[3]
5	10	15	20

1000	1002	1004	1006
------	------	------	------

a	a+1	a+2	a+3
1000	1002	1004	1006

Hence,

```
a = &arr[0]
a+1 = &arr[1]
a+2 = &arr[2]
a+3 = &arr[3]
```

Ex 13-

```
/*program to print the value and address of the element */
#include< stdio.h>
main ()
{
    int arr[4] = { 5 , 10 , 15 , 20 };
    int i = 0;
    for ( i=0;i<=4;i++)
    {
        printf ("value of arr[%d] = %d\n",i,arr[i]);
        printf (" address of arr[%d] = %u\n",i,&arr[i] );
    }
}
```

arr[0]	arr[1]	arr[2]	arr[3]
5	10	15	20
1000	1002	1004	1006

Output-

```
value of arr[0]=5
address of arr[0]=1000
value of arr[1]=10
address of arr[1]=1002
value of arr[2]=15
address of arr[2]=1004
value of arr[3]=20
address of arr[3]=1006
```

We know that the element of array are stored in consecutive memory location. The array is of 'int' data type. Since 'int' data type hold 2 bytes, hence address of next element of array is incremented by 2.

Ex 14 -

```
/*Use pointer to print the value and address of array elements*/
#include< stdio.h>
main ()
{
    int arr[4] = { 5 , 10 , 15 , 20 };
}
```

```

int i = 0;
int *b;
b = arr; /*we can also write b = &arr[0] */
for ( i = 0;i<=4;i++)
{
    printf ("value of arr[%d]=%d\n",i,*b);
    printf ("address of arr[%d]=%u\n",i,b);
    b=b+1;
}

```

arr[0]	arr[1]	arr[2]	arr[3]
5	10	15	20

1000 1002 1004 1006

b	b+1	b+2	b+3
1000	1002	1004	1006

Here 'b' is a pointer variable. b=arr means the base address (address of the 0th element of the array) are given to the pointer variable 'b'. *b gives the value at the address . 'b' gives the address of the array element. When 'b' is incremented then it holds the address of the next element of array and *b gives the value at this address.

Output -

```

value of arr[0]=5
address of arr[0]=1000
value of arr[1]=10
address of arr[1]=1002
value of arr[2]=15
address of arr[2]=1004
value of arr[3]=20
address of arr[3]=1006

```

We can also pass the whole array to a function.

Ex 15-

```

/*Use the pointer with function to print the value and address of the array element*/
#include< stdio.h>
main ()
{
    int arr[4]={ 5 , 10 , 15 , 20 };
    fun ( arr );
}

```

```

fun( int *a)
{
    int i;
    for( i = 0; i <= 4; i++)
    {
        printf(" value of arr[%d]=%d\n", i, *a);
        printf(" address of arr[%d]=%u\n", i, a);
        a = a + 1;
    }
}

```

Here the base address of array arr[] is passed as the parameter of function fun(). This address is passed to the pointer variable 'a'. When we increment 'a' by 1, this gives the address of next array element. *a gives the value at this address and 'a' gives the address of array element.

Output -

```

value of arr[0]=5
address of arr[0]=1000
value of arr[1]=10
address of arr[1]=1002
value of arr[2]=15
address of arr[2]=1004
value of arr[3]=20
address of arr[3]=1006

```

int arr[4] = { 5, 10, 15, 20 }

We know that the name of array 'arr' is the base address of array. This can also be written as arr+0.

*arr gives the 0th element of the array

or *arr = *(arr+0) = arr[0]

Similarly *(arr+1) gives the first element of the array.

arr[i] is the ith element of the array.

*(arr+i)

*(i+arr)

i [arr]

Ex 16-

```

/* program to print the value of the array element */
#include<stdio.h>
main()
{

```

```

int arr[4] = { 5, 10, 15, 20 };
int i = 0;

```

```

for ( i=0;i<4;i++)
{
    printf( " value of arr[%d]=",i);
    printf( "%d----",arr[i] );
    printf( "%d----",*(arr+i) );
    printf( "%d----",*(i+arr) );
    printf( "%d\n",i[arr] );
    printf( " address of arr[%d]=%u\n",i,&arr[ i ] );
}
}

```

Output -

value of arr[0]=5----5----5----5
 address of arr[0]=1000
 value of arr[1] = 10----10----10----10
 address of arr[1]=1002
 value of arr[2]=15----15----15----15
 address of arr[2]=1004
 value of arr[3]=20----20----20----20
 address of arr[3]=1006

arr[0]	arr[1]	arr[2]	arr[3]
5	10	15	20

1000 1002 1004 1006

Ex 17-

```

/* program to accept 10 numbers and print the total with use of pointer */
#include <stdio.h>
main ()
{
    int i,total;
    int arr[10];
    int *a;
    a = arr;
    for ( i=0;i<10;i++)
    {
        printf( " Enter the number %d",i+1);
        scanf( "%d",&arr[i]);
    }
    for ( i=0;i < 10 ; i++)
    {
        printf( "%d---",*a);
        total = total + *a;
        a=a+1;
    }
    printf( "\n Total = %d\n",total);
}

```

Ex 18-

```

/* program to accept 10 numbers and sort them with use of pointer */
#include<stdio.h>
main()
{
    int i,j;
    int arr[10];
    int *a;
    a=arr;
    for ( i=0;i<10;i++)
    {
        printf( "Enter the number %d=%d",i+1);
        scanf( "%d",&arr[i]);
    }
    printf( "Before sorting:\n");
    for ( i=0; i<10 ; i++)
        printf( "%d ", arr[i] );
    printf( "\n");
    a=arr;
    for ( i=0;i<10;i++)
        for ( j=0;j<10-i-1;j++)
            if ( *(arr+j) > *(arr+j+1))
                swap(arr+j , arr+j+1);
    printf( " After sorting :\n");
    for ( i=0 ; i < 10 ; i++)
        printf( "%d---",arr[i] );
    printf( "\n");
}
swap( int *b , int *c )
{
    int temp;
    temp = *b;
    *b = *c;
    *c = temp;
}

```

Output -

Enter the number 1=10
 Enter the number 2 = 15
 Enter the number 3=9
 Enter the number 4 = 12
 Enter the number 5=19
 Enter the number 6=20
 Enter the number 7 = 13
 Enter the number 8 =22
 Enter the number 9 = 27
 Enter the number 10 = 2

Before sorting:

10---15---9---12---19---20---13---22---27---2

After sorting:

2---9---10---12---13---15---19---20---22---27

POINTER WITH MULTI-DIMENSIONAL ARRAY-

We can also declare two or more dimensional arrays. Let us take two dimension array-
int arr[3][2]

Here arr[0] is an one-dimensional array having 2 elements which are arr[0][0] and arr[0][1]

Similarly other elements are -

arr[1][0]	arr[1][1]
arr[2][0]	arr[2][1]

Two dimensional array is also called a matrix. Here first subscript is a row number and second subscript is a column numbers.

Ex 19-

/* program to print the values and address of the array element*/

```
#include< stdio.h>
```

```
main ()
```

```
{
```

```
    int arr[3][2]= {
```

{10,100},
{20,200},
{30,300}
}

```
    int i, j;
```

```
    for ( i = 0; i < 3; i++ )
```

```
{
```

```
        printf( "address of %d array = %u\n", i, &arr[i] );
```

```
        for ( j=0 ; j < 2 ; j++ );
```

```
            printf( "value = %d\n", arr[i][j] );
```

```
}
```

arr[0][0]	arr[0][1]	arr[1][0]	arr[1][1]	arr[2][0]	arr[2][1]
-----------	-----------	-----------	-----------	-----------	-----------

10	100	20	200	30	300
1000	1002	1004	1006	1008	1110

Here &arr[0] gives the base address of the 0th 1-dimensional array. &arr[1] gives the base address of the 1st 1-dimensional array. We can see that a two-dimensional array is a collection of a number of one dimensional array, which are placed one after another.

Output-

```

address of 0 array = 1000
value = 10
value = 100
address of 1 array = 1004
value = 20
value = 200
address of 2 array = 1006
value = 30
value = 300

```

Let us analyze how the addresses are given in two dimensional array. We know that arr gives the base address of 2-D array. This can also be written as arr[0]. This gives the 0th element of 2-D array. But the 0th element of 2-D array is a 1-D array. Hence arr[0] gives the base address of 0th 1-D array. Similarly arr[1] gives the base address of 1st 1-D array and arr[i] gives the base address of ith 1-D array.

We know that

```

arr = arr[0] = *(arr+0)
arr[1] = *(arr+1)
arr[i] = *(arr+i)

```

Similarly, we can obtain the address of the 2-D array. As example - arr[0][1]

This shows the 1st element of arr[0]. Here arr[0] gives the base address of 0th array+1. We can write it as -

```

arr[0][1] = *(arr[0]+1) = *(*(arr+0)+1)
arr[i][j] = *(arr[i]+j) = *(*(arr+i)+j)

```

Here i is the ith 1-D array and j is the jth element of the ith array.

Ex 20 -

```

/*Program to print the elements of the array*/
#include<stdio.h>
main ()
{
    int arr[3][2] = {
        { 10 , 100 },
        { 20 , 200 },
        { 30 , 300 }
    };
    int i , j;
    for ( i = 0 ; i < 3 ; i++)
    {
        printf( "\n");
        for ( j=0;j< 2;j++)
        {
            printf( "value = %d\t", *(*(arr+i)+j));
        }
    }
}

```

Output-

```
value = 10    value = 100
value = 20    value = 200
value = 30    value = 300
```

Let us take a 3-D array-

```
int arr[2][3][2]
```

Here arr[0] is an array having 3 elements, each element is an array having 2 elements.
Here arr[0] gives the base address of 0th 2-D array and arr[1] gives the base address of 1st 2-D array.

Similarly, we can obtain the address of 3-D array element.

For example-

```
arr[0][2][1]
```

Here first subscript means 0th 2-D array. Second subscript means 2nd 1-D array and third subscript means 1st element of 2nd 1-D array of 0th 2-D array.

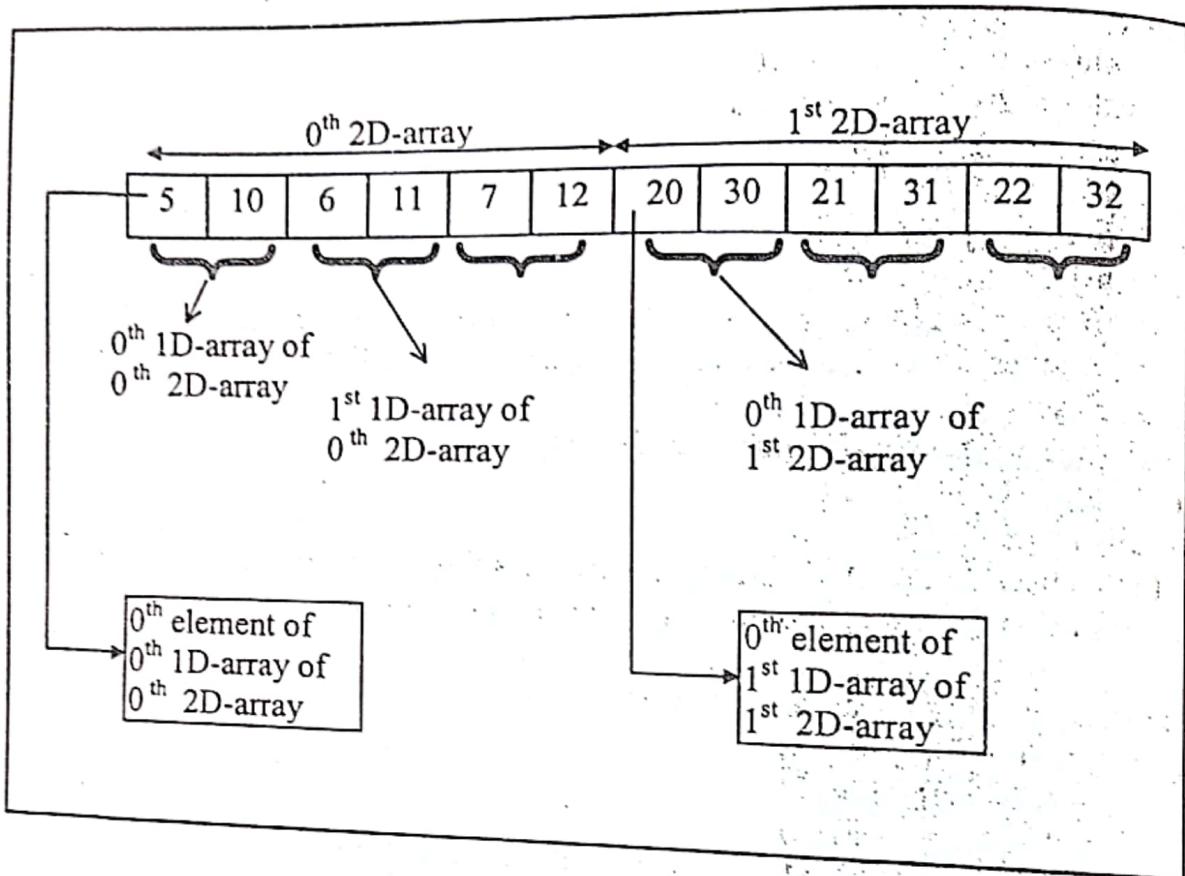
In 3-D array arr[i][j][k] can be written as -

$$\begin{aligned} \text{arr}[i][j][k] &= *(\text{arr}[i][j]+k) \\ &= *(*(\text{arr}[i]+j)+k) \\ &= *(*(*(\text{arr}+i)+j)+k) \end{aligned}$$

This means kth element of the jth 1-D array of the ith 2-D array.

Ex 21-

```
/*program to print the elements of 3-D array */
#include<stdio.h>
main ()
{
    int arr[2][3][2] = {
        {
            { 5 , 10 },
            { 6 , 11 },
            { 7 , 12 },
        },
        {
            { 20 , 30 },
            { 21 , 31 },
            { 22 , 32 },
        }
    };
    int i,j,k;
    for ( i=0;i<2;i++)
        for ( j=0;j < 3 ;j++)
        {
            printf( "\n");
            for ( k=0;k<2;k++)
                printf( " %d\t",*(*(arr+i)+j)+k);
        }
}
```



Here $*(*(*(arr+i)+j)+k)$ means k^{th} element of j^{th} 1-D array of i^{th} 2-D array.

Output-

5	10
6	11
7	12
20	30
21	31
22	32

ARRAY OF POINTERS-

We can also declare an array of pointers. Every element of this array can hold address of any variable. So we can say every element of this array is a pointer variable. It is same as array but it contains the collection of addresses.

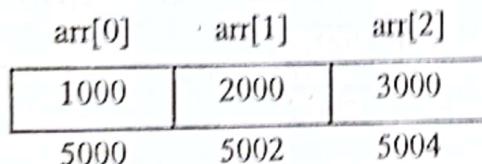
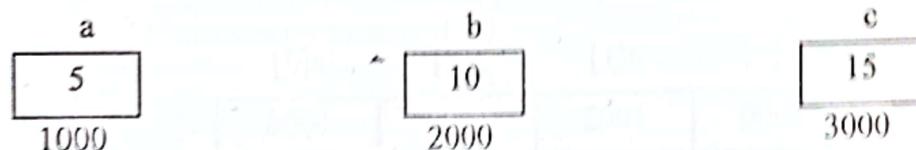
Ex 22-

```
/* program for understanding the concept of array of pointers */
#include<stdio.h>
main ()
{
    int *arr[3];
    int a = 5, b = 10, c = 15, i;
```

```

arr[0] = &a;
arr[1] = &b;
arr[2] = &c;
for ( i=0;i< 3;i++)
{
    printf ("address = %u\n", arr[i]);
    printf (" value = %d \n"*(arr[i]));
}

```



Here arr is declared as an array of pointers. Every element of this array holds the address of variable. arr[i] gives the address of the ith element of 'arr' and *arr[i] gives the value at this address.

Output-

```

address = 1000      value = 5
address = 2000      value = 10
address = 3000      value = 15

```

The array of pointers can also contain the address of another array.

Ex 23-

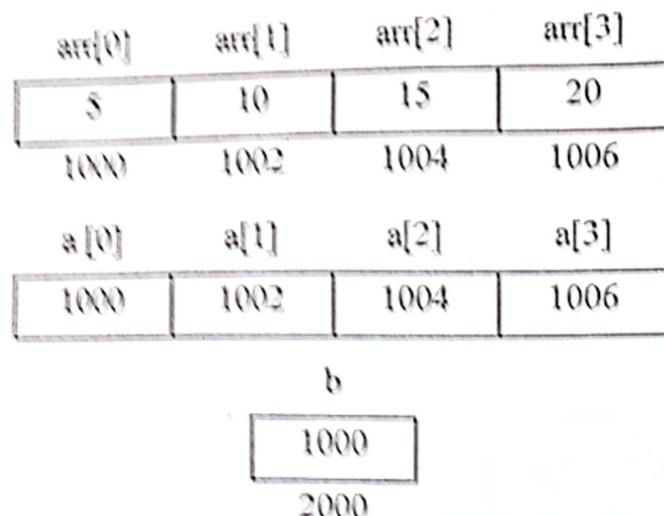
```

/* program for understanding this concept */
#include <stdio.h>
main ()
{
    static int arr[4] = { 5 , 10 , 15 , 20 };
    static int *a[4] = { arr , arr+1 , arr+2 , arr+3 };
    int i;
    int **b;
    b=a;
    for (i = 0 ; i < 4 ; i++)
    {
        printf ("address = %u", *b);
        printf (" value = %d \n", **b);
        b = b + 1;
    }
}

```

Here 'a' is declared as array of pointers. Each element of this array pointer contains the address of each element of array 'arr'.

'b' is declared as pointer to pointer variable, which contains the address of pointer array 'a'.



*b means the value at this address. Here 'b' holds the base address of the array 'a' and every element of the array 'a' is the address of the elements of the array 'arr'. *b gives the value at this address which is 1000. **b gives the value at (*b) means value at the address 1000, which is 5. Increment of 'b' by 1 gives the value of next element of the array 'a'.

Output-

```
address=1000 value=5
address=1002 value=10
address=1004 value=15
address=1006 value=20
```

POINTER AND STRING-

A string is a collection of characters that are stored in the character array. Every string is terminated with '\0' (null character). It is automatically inserted at the end of every string.

Let us take an example -

```
char arr[] = { 'P', 'R', 'A', 'T', 'I', 'B', 'H', 'A', '\0' }
```

Here the end of array with '\0' indicates that this is a string.

P	R	A	T	I	B	H	A	\0
1000	1001	1002	1003	1004	1005	1006	1007	1008

Ex 24 -

```
/*program to print the character of any string and also address of each character. */
#include< stdio.h>
main()
{
    char arr[ ] = "reeta";
    int i;
    for ( i=0;arr[i]!='\0';i++)
    {
        printf (" address = %u\n",&arr[ i]);
        printf ("character = %c\n",arr[ i]);
    }
}
```

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
r	e	e	t	a	\0

1000 1001 1002 1003 1004 1005

Output -

```
address=1000 character=r
address=1001 character=e
address=1002 character=e
address=1003 character=t
address=1004 character=a
```

Ex 25-

```
/*program to print the address and character of the string with using pointer*/
#include< stdio.h>
main ()
{
    char arr[ ] = "reena";
    char *a;
    a = arr;
    while ( *a !='\0' )
    {
        printf ("address= %u\n",a);
        printf ("character = %c\n",*a);
        a = a+1;
    }
}
```

arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]
r	e	e	n	a	\0

Here 'a' is pointer variable which holds the base address of array arr[]. Increment of pointer a with 1 gives the address of next element of character array arr[]. The loop terminates when the last character '\0' comes.

Output -

```
address=1000 character=r
address=1001 character=c
address=1002 character=e
address=1003 character=n
address=1004 character=a
```

TWO-DIMENSIONAL ARRAY OF CHARACTERS-

We can also declare two-dimensional array of characters. As example-

```
char arr[5][10] = {
    "riti",
    "niti",
    "kriti",
    "kittu",
    "nitin"
};
```

Here first subscript of array arr[] means number of strings in the array and second subscript gives the length of string.

riti\0	niti\0	kriti\0	kittu\0	nitin\0
5000	5010	5020	5030	5040

Here 10 bytes are reserved in memory for each string. 5000 is the base address of the first string. Similarly, 5010 is the base address of the second string. We can see the first string takes only 5 bytes, so 5 bytes memory is wasted. Similarly 4th string takes only 6 bytes and 4 bytes memory is wasted.

The total number of bytes occupied by this array is 50 while the strings use only 28 bytes so wastage of memory is of 22 bytes.

Ex 30-

```
/* program to print the strings of the two-dimensional character array*/
#include<stdio.h>
main ()
{
char arr[5][10] = {
    "riti",
    "niti",
    "kriti",
    "kittu",
    "nitin"
};
```

```

    "kriti",
    "kittu",
    "nitin"
};

int i;
for ( i=0;i<5;i++)
{
    printf (" base address = %u",&arr[ i ]);
    printf (" string = %s\n",arr[ i ]);
}
}

```

Here &arr[i] gives the base address of each and arr[i] prints the string.

Output -

base address=5000	string=riti
base address=5010	string=niti
base address=5020	string=kriti
base address=5030	string=kittu
base address=5040	string=nitin

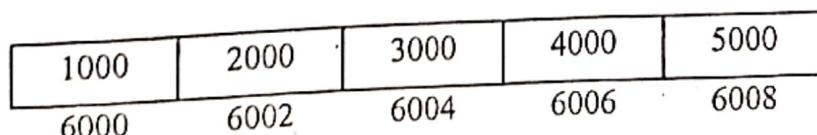
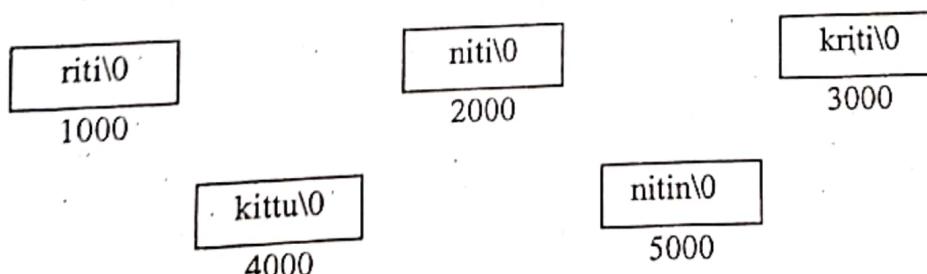
ARRAY OF POINTERS TO STRING-

We can also declare an array as a pointer to string. Each element of this array contains the base address of each string. Let us take an example-

```

char *arr[ ] = {
    "riti",
    "niti",
    "kriti",
    "kittu",
    "nitin"
}

```



Here arr[] is declared as an array of pointer to strings. Each element of this array contains the base address of each string. Here all strings occupy 28 bytes and 10 bytes are occupied by the array of pointers. So the total bytes occupied are 38 bytes. In two-dimensional array, total bytes occupied are 50 bytes. We can see, here saving of 12 bytes.

Ex 31-

```
/* program to print the address and string using array of pointer to string */
#include<stdio.h>
main ()
{
    int i;
    char *arr[ ] = {
        "riti",
        "niti",
        "kriti",
        "kittu",
        "nitin"
    };
    for ( i=0;i<5;i++)
    {
        printf( "address=%u\t", (arr + i));
        printf( "string = %s\n", *(arr + i));
    }
}
```

Output -

```
address=6000 string=riti
address=6002 string=niti
address=6004 string=kriti
address=6006 string=kittu
address=6008 string=nitin
```

DYNAMIC MEMORY ALLOCATION.

In an array, it is must to declare the size of array. There are two possibilities when we declare the array size. The first case is , the records that are stored is less than the size of array, second case, we want to store more records than size of array. In first case, there is a wastage of memory. In second case, we cannot store more records than the size of array. The C language has the facility to allocate memory at the time of execution. The process of allocating memory at the time of execution is called as dynamic memory allocation. The allocation and releasing of this memory space can be done with the use of some built-in-functions which are found in alloc.h header file. Let us take these functions in detail.

1. sizeof () -

sizeof is an unary operator. This operator gives the size of its argument in terms of bytes. The argument can be a variable, array or any datatype (int, float, char etc). This operator also gives the size of any structure . As example-

```
sizeof( int );
```

This gives the bytes occupied by the int datatype, that is 2.

Ex 33- .

```
/*program to understand the sizeof operator */
#include<stdio.h>
main()
{
struct
{
    char name[10];
    int age;
    float sal;
}rec;
int arr[10];
printf(" size of structure = %d",sizeof( rec ));
printf(" size of int = %d",sizeof( int ) );
printf(" size of array = %d",sizeof( arr ) );
}
```

Output -

```
size of structure = 16
size of int = 2
size of array = 20
```

Here rec is a variable of structure type. hence sizeof (rec) gives the total number of bytes occupied by the structure. Total number of bytes occupied by the structure is 16. sizeof (arr) gives value 20 because total number of bytes taken by the array is 20.

2. malloc () -

This function is used to allocate memory space. The malloc () function reserves a memory space of specified size and gives the starting address to pointer variable. This can be written as-

```
ptr = (datatype * ) malloc ( specified size );
```

Here datatype is the type of pointer and specified size is the size which is required to reserve in memory. For example-

```
ptr = ( int * ) malloc ( 10 );
```

This allocates 10 bytes of memory space to the pointer ptr of type int and the base address is stored in the pointer variable ptr.

```
ptr = ( int * ) malloc ( 10 * sizeof ( int ) );
```

This allocates the memory space 10 times, that is hold an int datatype. The base address is stored in the pointer variable ptr.

Ex 34 -

```
/* program to enter the 5 numbers and print them using malloc() */
```

```
#include<stdio.h>
```

```
main ( )
```

```
{
```

```
    int i;
```

```
    int *a;
```

```
    a = ( int * ) malloc ( 5 * sizeof ( int ) );
```

```
    for ( i=0;i<5;i++ )
```

```
{
```

```
        printf ( " number %d=", i+1 );
```

```
        scanf ( "%d", &( a+i ) );
```

```
}
```

```
    for ( i=0;i<5;i++ )
```

```
        printf ( "%d\n", *( a+i ) );
```

```
}
```

3. calloc ()-

The calloc () function is used to allocate multiple blocks of memory. This has two arguments. For example-

```
ptr = ( int * ) calloc ( 5,2 );
```

This allocates 5 blocks of memory, each block contains 2 bytes of memory and the starting address is stored in the pointer variable ptr which is of type int.

The calloc () function is generally used for allocating the memory space for array and structure.

As example-

```
struct record
```

```
{
```

```
    char name[10];
```

```
    int age;
```

```
    float sal;
```

```
};
```

```
int tot-rec=100;
```

```
ptr = ( record * ) calloc ( tot-rec, sizeof ( record ) );
```

This allocates the memory space for 100 blocks and block contains the memory space that is occupied by the structure variable record.

4. free () -

For efficient use of memory space we can also release the memory space that is not required. We can use the free () function for releasing the memory space. For example -

```
free ( ptr );
```

Here ptr is a pointer variable that contains the base address of memory block, that is created by malloc() or calloc ().

5. realloc () -

There are two possibilities when we want to change the size of the block. In first case, we want more memory space in comparison to allocated memory space. In second case, the allocated memory space is more than the required memory space. For changing the size of the memory block we can use the function realloc(). This is known as reallocation of memory. For example-

```
ptr = malloc ( specified size );
```

This statement allocate the memory of the specified size and the base address of this memory block are stored in the pointer variable ptr. If we want to change the size of memory block , then

```
ptr = realloc ( ptr , newsize );
```

From this statement we can allocate the memory space of this newsize and the base address of this memory block is stored in the pointer variable ptr. The base address may or may not be same because it is possible that memory block of newsize may or may not be in the same region. This function moves the content of old block into the new block and the data of the old block is not lost.

Ex 35-

```
/*program to understand the use of realloc function */
#include<stdio.h>
#include<alloc.h>
#include<string.h>
main ()
{
    char *ptr;
    ptr = ( char * ) malloc ( 6 );
    ptr = "ankit";
    printf ( "%s is in memory block\n",ptr );
    ptr = ( char * ) realloc ( ptr,8 );
    printf ( "%s is in memory block\n",ptr );
    strcpy ( ptr , "rishabh" );
```

```

        printf("Now %s is in memory block\n");
    }
}

```

Output :-

ankit is in memory block
 ankit is in memory block
 Now rishabh is in memory block

STRUCTURE

WHAT IS STRUCTURE:-

As we know array is the collection of same type of elements but it may be possible to take a different type of elements together. It is useful to group the different types of data which has some meaningful information. If we want to create a record of any person which contains the name, age and address of that person then we make the structure of the record as-



Here Name, Age and address are the related field of that record.

We can define the structure as-

```

struct tag{
    member1;
    member2;
    .....
    .....
    member m;
};

```

Here struct is a keyword for structure and member1, member2, , member m is the members of structure.
 This can also be defined as-

```

struct {
    member1;
    member2;
    .....
    .....
    member m;
} var;

```

Here var is the structure variable.

Ex-

```
struct {
    char name[20];
    int age;
    char address[20];
}rec;
```

Here name, age and address are the members of this structure. rec is the structure variable. For accessing any member of the structure , we use dot (.) operator. For example, rec.name points to the member name of this structure.

ANOTHER TYPE OF DECLARATION-

We can also declare the structure with the use of structure tag. This can be written as-

```
struct record {
    char name[20];
    int age;
    char address[20];
}rec;
struct record person;
```

Here ,we define another structure variable person which has the same structure as record.

INITIALIZE VALUE TO THE STRUCTURE-

We can also initialize the value to the members of structure as-

```
struct {
    char name[20];
    int age;
    char address[20];
}rec = { "suresh" , 24 , "anpara" };
```

Here rec.name, rec.age, rec.address have values suresh , 24 and anpara.

ANOTHER TYPE OF INITIALIZATION-

Another way for assigning the values to the structure members are as-

```
struct rec{
    char name[20];
    int age;
    char address[20];
};
struct rec person = { "Nirvikar" , 28 , "Allahabad" };
```

ARRAY OF STRUCTURES-

As we know array is collection of same datatype of elements. For example, we can take 10 integer values in the integer type array. Similarly we can declare the array of structures where every element of array is structure type. Array of structure can be declare as-

```
struct rec{  
    char name[20];  
    int age;  
    char address[20];  
} person[10];
```

Here person is an array of 10 elements and each element of person have structure of rec , means each element of person have 3 member element which are name, age and address.

PASSING STRUCTURE TO FUNCTION-

As in function we take the variables as actual parameter and pass them in same datatype in formal parameter. Similarly we can pass the structure variable. For example-

```
struct rec{  
    char name[20];  
    int age;  
    char address[20];  
} var1;  
main()  
{  
    .....  
    .....  
    func( var1 );  
    .....  
}  
func( struct rec var2 )  
{  
    .....  
    .....  
}
```

It is necessary to declare the structure globally.

PASSING ARRAY OF STRUCTURE TO FUNCTIONS-

As we pass the array in the function , similarly we can pass the array of structure to function, where each element of array is structure type. This can be written as-

```

struct rec {
    char name[20];
    int age;
    char address[20];
};

main()
{
    .....
    .....
    struct rec emp[10];
    .....
    func( emp );
    .....
}

func( struct rec person )
{
    .....
    .....
}

```

STRUCTURE WITHIN STRUCTURE-

We can take the member element of structure as any datatype, such as int, float, char etc. Similarly we can take the member of structure as structure type. We can take any structure as a member of structure, i.e. called structure within structure. For example-

```

struct tag1{
    member1;
    member2;
    .....
    struct tag2{
        member1;
        member2;
        .....
        member m;
        }var1;
    .....
    member m;
    }var2;

```

For accessing the member1 of inner structure, we write as-

var2.var1.member1;

Ex-

```
struct rec{  
    char name[20];  
    int age;  
    struct dob{  
        int day;  
        int month;  
        int year;  
    }birthday;  
    char address[20];  
}emp;
```

Here struct rec is the structure for employee record and struct dob represents the birth date of that employee.

TYPEDEF

typedef is used for defining new datatype. The general syntax is-

```
typedef type dataname;
```

Here type is the data type and dataname is name for that type.

```
typedef int mark;
```

Here mark is the another name for int and now we can use mark instead of int in the program as-

```
mark age, class;
```

Similarly we can also use typedef for defining structure-

```
typedef struct{  
    member1;  
    member2;  
    .....  
}dataname;
```

Here dataname is the another name for defining this structure.

```
typedef struct{  
    int class;  
    int marks;  
}rec;  
rec person;
```

Here rec is used for defining the structure and person has the same structure as structure rec.

```
typedef struct{
    int class;
    int marks;
    char name[20];
}rec;
rec person[10];
```

Now person is the array of 10 elements which has same structure as rec.

UNION

union is almost same as the structure. As the structure contains members of different datatypes. In the structure each member has its own memory location, whereas members of unions have same memory location. We can assign values to only one member at a time, so assigning value to another member at that time has no meaning.

When a union is declared ,compiler automatically allocates a memory location to hold the largest data type of members in the union. Thus the union is used for saving memory. The concept of union is useful when it is not necessary to assign the values to all the members of the union at a time. Union can be declared as-

```
union union_name {
    member1;
    member2;
-----
}
```

Here union is the keyword, it is necessary for declaration of union. The declaration of union variable is same as structure

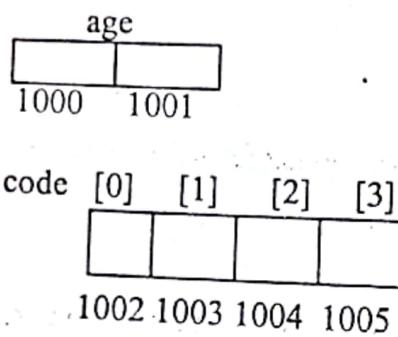
```
union union_name
{
    member1;
    member2;
-----
} variable_name;
```

This can also be declared as-

```
union union_name variable_name;
```

Ex-

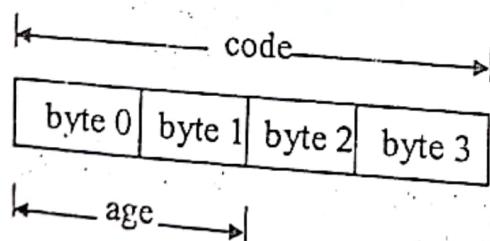
```
struct
{
    int age;
    char code[4];
}
```



Here both the members of structure have different memory location let us take an example of union

example -

```
union emp
{
    int age;
    char code[4];
}
```



In union emp both int age and char code[4] share the same memory location. Here code occupies 4 bytes and age uses only 2 bytes.
We can access the union members same as structure if we access directly then we use the dot(.) operator, if we access through pointer then we use arrow (->) operator.

POINTER TO STRUCTURE.

As we know that pointer is a variable which holds the starting address of another variable, it may be of type int, float or char. Similarly, if the variable is of structure type then for accessing the starting address of the structure, we must declare pointer for a structure variable. These pointers are called structure pointers and can be declared as-

Ex-

```
struct rec {
    char name[10];
    int age;
    int sal;
} data;
struct rec *ptr;
```

Here ptr is a pointer variable which points to the structure rec. ptr can hold the starting address of the structure as-

ptr=&data;

because data is a structure variable.

This can also be written as-

```
struct rec {
    char name[10];
    int age;
    float sal;
} data,*ptr;
ptr=&data;
```

Here data is a structure variable and ptr is a structure pointer that points to the starting address of structure variable data.

There are two ways of accessing the member of structure through the structure pointer. First is to use the arrow operator (\rightarrow), formed by hyphen symbol and greater than symbol. As example for accessing the member element age of the structure we can write as-

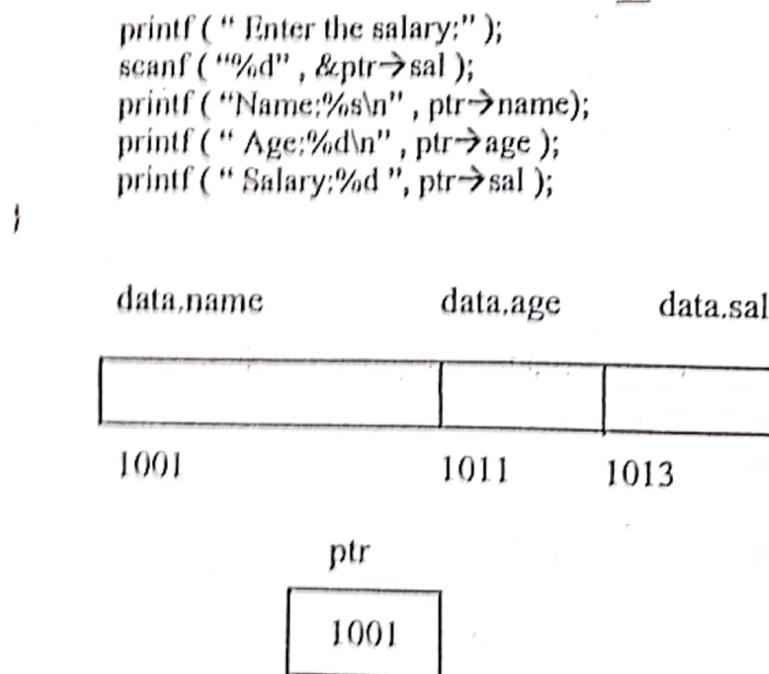
ptr \rightarrow age;

The other way for accessing the member element is as-

(*ptr).age;

Here parenthesis is necessary because the dot operator has higher precedence than the * operator.

```
/*Program to understand pointer to structure*/
#include<stdio.h>
main ()
{
    struct rec {
        char name[10];
        int age;
        int sal;
    } *ptr;
    printf("Enter the name:");
    scanf("%s", ptr $\rightarrow$ name);
    printf("Enter the age:");
    scanf("%d", &ptr $\rightarrow$ age);
```



```

/*Program to assign the value to member of structure with the use of structure pointer */
#include<stdio.h>
#include<stdio.h>
main ()
{
    struct rec {
        char name[10];
        int age;
        int sal;
    } *ptr;
    strcpy(ptr→name, "suresh");
    ptr→age = 24;
    ptr→sal = 5000;
    printf( "Name:%s\n" , ptr→name );
    printf( "Age:%d\n" , ptr→age );
    printf( "Salary:%d " , ptr→sal );
}

```

We can also pass the address of the structure variable (Remember that previously we pass whole structure) to the function,

```

/* program to pass the address of the structure variable */
#include<stdio.h>
struct rec {
    char name[10];
    int age;
    int sal;
};

```

```

main()
{
    static struct rec data = { "sachin", 26, 25000 };
    function( &data );
}

function( ptr )
struct rec *ptr;
{
    printf( "Name:%s\n", ptr->name );
    printf( "Age:%d\n", ptr->age );
    printf( "Salary:%d\n", ptr->sal );
}

```

A pointer can also be used as a member of structure. If ptrmem is a pointer variable and a member of structure variable strvar, then *strvar.ptrmem will access that value which is pointed by ptrmem. If the ptr is a structure pointer and a pointer ptrmem is a member of that structure then *ptr->ptrmem will access that value which is pointed by ptrmem.

As example-

```

struct rec{
    char *name;
    int *age;
    int *sal;
}data, *ptr;

/* program to use a pointer as a member of structure and print the value of members*/
#include<stdio.h>
main()
{
    struct rec{
        char *name;
        int *age;
        int *sal;
    }data,*ptr;
    char name1[10]= "R.O. Verma";
    int age1=50;
    int sal1=17000;
    ptr->name=&name1;
    ptr->age=&age1;
    ptr->sal=&sal1;
    printf( "Name:%s\n", ptr->name );
    printf( "Age:%d\n", *ptr->age );
    printf( "Salary:%d\n", *ptr->sal );
}

```

```
/*Same program without use of arrow operator */
#include<stdio.h>

main()
{
    struct rec{
        char *name;
        int *age;
        int *sal;
    }data;
    char name1[10] = "B.N.S. Srivastava";
    int age1=23;
    int sal1=17000;
    strcpy(data.name , "B.N.S.Srivastava");
    data.age=&age1;
    data.sal=&sal1;
    printf ("Name:%s\n" , data.name );
    printf ("Age.%d\n" , *data.age );
    printf ("Salary:%d\n" , *data.sal );
}
```

/*Program to accept name , class and marks with the use of structure with pointer. Also print the grade which is based on the marks-

1. marks ≥ 75 grade-A
2. marks ≥ 50 and < 75 grade-B
3. marks < 50 grade-C

```
*/
#include<stdio.h>
```

```
main()
{
    struct {
        char name[20];
        int class;
        int marks;
    }*ptr;
    printf ("Name:");
    scanf (" %s", ptr->name );
    printf ("Class:");
    scanf (" %d", &ptr->class );
    printf (" Marks:");
    scanf (" %d", &ptr->marks );
    printf ("Name\tClass\tMarks\tGrade\n");
    printf (" %s\t", ptr->name );
    printf (" %d\t", ptr->class );
    printf (" %d\t", ptr->marks );
    if (ptr->marks < 50 )
        printf (" F\n");
```

```

    else if ( ptr->marks >=50 && ptr->marks < 75 )
        printf( " B\n" );
    else
        printf( " A\n" );
}

```

*/*Program to accept name , class and marks of 10 student with the use of structure with array of pointer and also print them.*/*

```

#include<stdio.h>
main ()
{
    struct {
        char name[20];
        int class;
        int marks;
        }*rec[10];
    int i;
    for ( i=0;i< 10;i++ )
    {
        printf( "Name:" );
        scanf( "%s", rec[i]->name );
        printf( "Class:" );
        scanf( "%d", &rec[i]->class );
        printf( " Marks:" );
        scanf( "%d", &rec[i]->marks );
    }
    printf( "Name\tClass\tMarks\n" );
    for( i=0 ; i<10 ;i++ )
    {
        printf( "%s\t", rec[i]->name );
        printf( "%d\t", rec[i]->class );
        printf( "%d\n", rec[i]->marks );
    }
}

```

Exercise

1.


```

#include<stdio.h>
main()
{
    int arr[4]={2,4,8,16};
    int i=4,j;
    while( i )
    {
        j=arr[--i] + i;
    }
}

```

```

        i--;
    }
    printf( "j=%d\n" , j );
}

```

3.

```

#include<stdio.h>
main()
{
    int i=0;
    char name[10]={‘b’, ‘i’, ‘n’, ‘a’, ‘r’, ‘a’, ‘n’, ‘i’, ‘\0’};
    while( name[i] )
    {
        printf( “ %d\n”, name[ i ] );
        i++;
    }
}

```

3.

```

#include<stdio.h>
main( )
{
    int i=0,sum=0;
    int arr[6]={ 4,2,6,0,5,10};
    while( arr[i] )
    {
        sum=sum+arr[i];
        i++;
    }
    printf( “sum=%d\n”,sum);
}

```

4.

```

#include<stdio.h>
main( )
{
    char name[15] = “Arvind”;
    int i=0;
    while( name[i] )
    {
        printf( “%c\n”,name[i] );
        i = i +6;
    }
}

```

5.

```
#include<stdio.h>
main()
{
    int arr[5] = { 5,10,15,20,25};
    func( arr );
}
func( int arr[ ][ ] )
{
    int i = 5, sum=0;
    while( i > 2 )
    {
        sum = sum+arr[--i];
        printf( " sum = %d\n",sum );
    }
}
```

6.

```
#include<stdio.h>
#include<string.h>
main()
{
    char name1[15] = "Neha";
    char name2[15] = "Deepali";
    if ( !strlen(name1) )
        strcpy( name1, name2 );
    else
        strcat( name1, name2 );
    printf( "Name1=%s\n",name1);
    printf( "Name2=%s\n",name2 );
}
```

7.

```
#include<stdio.h>
main()
{
    int i=5;
    char name1[15] = "appu";
    char name2[15] = "soumya";
    if ( !strcmp( name1,name2 ) )
        strcat( name1, name2 );
    else
        i = strlen( strcat( name1, name2 ) );
    printf( "Name1=%s\n",name1);
    printf( "Name2=%s\n",name2 );
}
```

8.

```
#include<stdio.h>
main()
{
    int arr[3][2] = { 2 , 4,
                      4 , 6,
                      8 , 10
                  };
    printf( "%d\n",arr );
    printf( "%d\n",arr[1] );
    printf( "%d\n",arr[1][2] );
}
```

9.

```
#include<stdio.h>
main()
{
    int i;
    char name[5][15] = {
        "Awadhesh",
        "Reena",
        "Reeta",
        "Ranjana",
        "Mithilesh"
    };
    for( i=0; i < 5; i++ )
        printf("%s\n", name[i] );
}
```

10.

```
#include<stdio.h>
#define A 4
#define B 3
main()
{
    int i,j;
    int arr[A][B] = { 2,4,6,8,10,18,6,4,2,1 };
    for( i=0; i < A ; i++ )
        for(j =0; j < B; j++)
            display( arr[A][B] );
    display( int arr[ ][ ] )
    {
        printf( "%d\n", arr[A-B] + arr[B-1] );
    }
}
```

11.

```
#include<stdio.h>
main( )
{
    int arr[10];
    int *a,*b;
    *a=&x;
    *b=&arr[2];
    x=b-(a++);
    b = b+2;
    printf (" Address of b = %u\n",b);
}
```

12.

```
#include<stdio.h>
main( )
{
    int *a,x=2;
    do{
        a = fun( x );
        printf (" %d\n",*a);
        x++;
    }while (( x<8 ) && ( *a < 1000 ) );
int *f( int x )
{
    int y;
    y = x*200;
    return( &y );
}
```

13.

```
#include<stdio.h>
#include<string.h>
main( )
{
    puts( combine( "Alok", "Saxena" ) );
}
char *combine( char *arr1, char *arr2 )
{
    char str[80];
    int x,y,i,j;
    x=strlen(arr1);
    y=strlen(arr2);
    for ( i = x, j=0; j<y; i++,j++ )
        str[ i ] = arr2[ j ];
```

```

        str[ i ] = '\0';
        return( str );
    }
}

```

14.

```

#include<stdio.h>
char *str = "Saurabh Srivastava";
main()
{
    int i;
    while( str[i++]);
    printf( " length of string = %d\n",i );
}

```

15.

```

#include<stdio.h>
main()
{
    char *arr[4] = { "Rishabh", "is", "an", "Indian" };
    char **x;
    x=&arr[1];
    swap(arr,++x);
    if( strcmp( *x,arr[3] ) )
        printf( "True\n");
    else
        printf( " False\n");
}
swap( char **a, char *b)
{
    char **temp;
    temp = ++a;
    *a=b;
    b=*temp;
}

```

16.

```

#include<stdio.h>
int arr[ ] = { 5,10,15,20 };
main()
{
    int a=2;
    arr[a] = 15;
    func( &a, &arr[a] );
    printf( "%d, %d\n", a, arr[a] );
}

```

```
func( int *x, int *y )
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

17.

```
#include<stdio.h>
main()
{
    int x,y,z;
    int *xx;
    x=5;
    *xx=&x;
    y=x+10;
    z=*xx;
    printf( "%d,%d,%d\n",x,y,z );
}
```

18.

```
#include<stdio.h>
#include<string.h>
typedef struct {
    char *name;
    int *age;
}rec;
main()
{
    display( rec person );
    struct rec person;
    person.name = "Sharaj";
    person.age = 32;
    printf( " Name = %s\n", person.name );
    printf( " Age = %d\n", person.age );
}
display( rec person )
{
    person.name = "Shailesh";
    person.age = 28;
    printf( " Name = %s\n", person.name );
    printf( " Age = %d\n", person.age );
}
```

19.

```

typedef struct {
    int size;
    union {
        struct {
            unsigned char head;
            unsigned char track;
            unsigned char sector;
            }hts;
        int cluster;
        }address;
    struct {
        int total;
        detail *ptrs;
        }FAT[5][2];
}
main()
{
    int y;
    detail *var1,*var2[5];
    detail var3[3] = { 10, {127},20,{63},5,{512} };
    var1=var3;
    var2[0] = var1+1;
    var1->address.cluster = 1020;
    var3[0].size = var1->address.hts.track;
    var3[2].address.hts.track = 2;
    FAT[0][0].ptrs = var1+2;
}

```

What will be the value of followings-

- (i) var1->size
- (ii) ++var1->size
- (iii) (++var1)->size
- (iv) sizeof (detail)
- (v) sizeof (var2)
- (vi) sizeof (FAT)
- (vii) ++FAT[0][0].ptrs->size
- (viii) var2[0]->address.cluster
- (ix) var2[0]->address.hts.track
- (x) (var1+2)->address.cluster

20.

(DOE July 1996)

```

#include<stdio.h>
main()
{

```

```

struct A {
    int marks;
    char grade;
}A1;

A1.marks = 80;
A1.grade = 'A';
printf( " Marks = %d\n", A1.marks );
printf( " grade = %c\n", A1.grade );
struct A B1;
B1 = A1;
printf( " Marks = %d\n", B1.marks );
printf( " Grade = %c\n", B1.grade );
}

```

21.

```

#include<stdio.h>
main()
{
    struct rec{
        char name[20];
        int age;
    }rec1={"Prashant",28};
    func( rec1 );
}

func( struct rec rec1 )
{
    printf( " Name = %s\n", rec1.name );
    printf( " Age = %d\n", rec1.age );
}

```

22.

```

#include<stdio.h>
main()
{
    struct rec {
        char *name;
        int age;
    }*ptr;
    char name1[10] = "Elang Kumaran";
    ptr->name = &name1;
    ptr->age = 25;
    printf( " Name = %s\n", *ptr->name );
    printf( " Age = %d\n", ptr->age );
}

```

23.

```
#include<stdio.h>
main( )
{
    struct {
        char name[10];
        int age;
        float salary;
    }rec;
    printf (" Address of name =%u\n", &rec.name );
    printf (" Address of age =%u\n", & rec.age );
    printf (" Address of salary =%u\n",&rec.salary);
}
```

24.

```
#include<stdio.h>
main( )
{
    union tag {
        char name[15];
        int age;
    }rec;
    rec.name = "Shipra";
    rec.age = 24;
    printf ("Name = %s\n",rec.name );
}
```

Chapter 3

Linked List

What is List-

As the name implies, list is a collection of short pieces of information, such as names of people, usually written with a single item on each line and often ordered in a way that makes a particular item easy to find. It is usually used in the daily life routine. Some other examples of list are birthday list of friends, shopping list.

Here we will take the name and age of persons in the list.

Name	Age
Suresh	26
Ranjana	32
Reeta	30
Reena	28
Ankit	10

We can take three operations on the list-

1. Addition
2. Deletion
3. Searching

Name	Age
Suresh	26
Reeta	32
Reena	28
Ankit	10
Rajesh	37
Madhu	35

Here we have deleted the name and age of Ranjana and added the name and age of Madhu and Rajesh in the list.

There are two ways of maintaining this list in computer memory. First is to take an array to store the elements and second is the linked list.

Array Implementation of List-

Let us take an array of size 10, which has 5 elements.

```
int arr[10];
```

arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	10	20	30	40	50					

Traversing an array List-

In array, we can find the next element through the index number of array because each next element of array has index number incremented by 1 with previous index number. So we can traverse and access the array elements through array index.

Let us take index value is 0.

Now the contents of arr[index] is 10 which is first element of array. We can process the next element of array by incrementing the index by 1.

$$\text{index} = \text{index} + 1$$

Now array[index] is 20 which is second element of array. So we can traverse each element of array by incrementing the index by 1 until the index is not greater than number of elements.

Searching in an array list-

Searching refers to search an element into an array list. For searching the element, we first traverse the array list and with traversing, we compare each element of array with the given element.

We can search the element by another methods, which we will describe in the topic of searching.

Insertion into an array list-

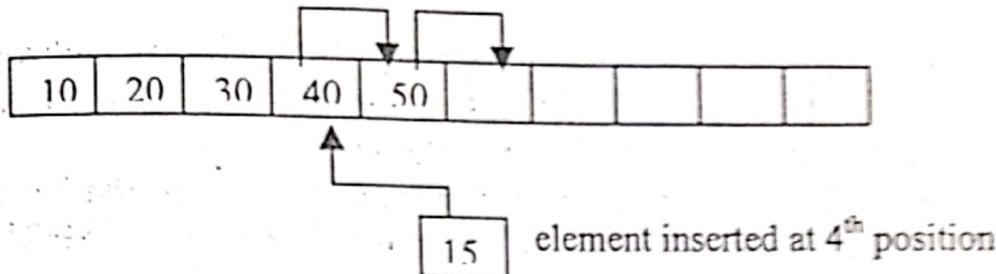
Insertion into an array list may be possible in two ways-

1. Insertion at end
2. Insertion in between

arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	10	20	30	40	50					

Case I

element inserted at 6th position



Case II

Case I-

In the first case we have to set the array index to the total number of element and then insert the element.

index = Total number of elements (i.e. 5)

arr[index] = value of inserted element

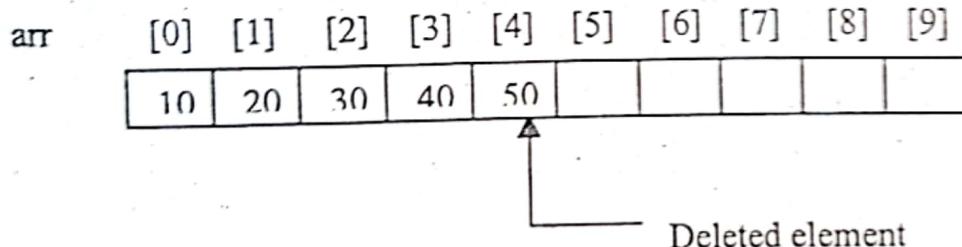
Case II-

In the second case, we have to shift right one position all array elements from last array element to the array element before which we want to insert the element.

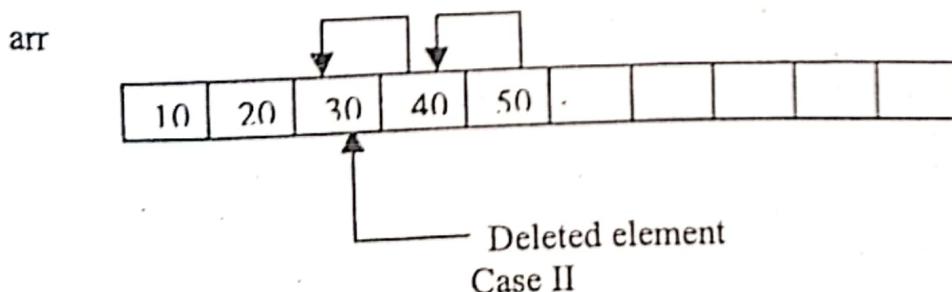
Deletion from an array list-

Deletion into an array list may be possible in two ways.

1. Deletion of the last element
2. Deletion in between



Case I



Case II

Case I-

First traverse the array list and compare array element with the element which you want to delete. If the item is last item of the array then delete that element and decrease the total number of elements by 1.

Case II-

In the second case, first traverse the array list and compare array element with the element which you want to delete then shift left one position from the next element to the last element of array and decrease the total number of elements by 1.

It is necessary to keep the status of total number of elements and size of array. If we want to keep a record such as employee code, name in the array list then we can take array of structure as-

Example-

struct {

```
    int code;
    char name[30];
} emp[10];
```

Here each element of structure array emp[10] has member variables code and name.

- ① The advantage of the array list is that we can easily compute the address of the array through index and we can also access the array element through index.
- ② The disadvantage of the array list is that we have to keep the total number of elements and array size. We cannot take elements more than array size because array size is fixed.
- ③ It also requires much processing in insertion and deletion because of shifting of array elements.

```
/* Program of list using array */
#include<stdio.h>
#define MAX 20
int arr[MAX];
int n; /*Total number of elements in the list */

main()
{
    int choice,item,pos;
    while(1)
    {
        printf("1.Input list\n");
        printf("2.Insert\n");
        printf("3.Search\n");
        printf("4.Delete\n");
        printf("5.Display\n");
        printf("6.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                printf("Enter the number of elements to be entered : ");

```

```

        scanf("%d", &n);
        input(n);
        break;
    case 2:
        insert();
        break;
    case 3:
        printf("Enter the element to be searched : ");
        scanf("%d", &item);
        pos = search(item);
        if(pos >= 1)
            printf("%d found at position %d\n", item, pos);
        else
            printf("Element not found\n");
        break;
    case 4:
        del();
        break;
    case 5:
        display();
        break;
    case 6:
        exit();
        break;
    default:
        printf("Wrong choice\n");
    } /*End of switch */
} /*End of while */
} /*End of main() */

```

```

input()
{
    int i;
    for(i = 0; i < n ; i++)
    {
        printf("Input value for element %d : ", i+1);
        scanf("%d", &arr[i]);
    }
} /*End of input() */

```

```

int search(int item)
{
    int i;
    for(i=0; i < n; i++)
    {
        if(item == arr[i])
            return(i+1);
    }
    return(0); /* If element not found */
} /*End of search() */

```

```

insert()
{
    int temp,item,position;
    if(n == MAX)
    {
        printf("List overflow\n");
        return;
    }
    printf("Enter position for insertion : ");
    scanf("%d", &position);
    printf("Enter the value : ");
    scanf("%d",&item);
    if(position > n+1 )
    {
        printf("Enter position less than or equal to %d\n",n+1);
        return;
    }
    if( position == n+1 ) /*Insertion at the end */
    {
        arr[n] = item;
        n = n+1;
        return;
    }
    /* Insertion in between */
    temp=n-1;
    while( temp >= position-1)
    {
        arr[temp+1] = arr[temp]; /* shifting right */
        temp--;
    }
    arr[position-1] = item;
    n = n +1 ;
}/*End of insert()*/

```

```

del()
{
    int temp,position,item;
    if(n == 0)
    {
        printf("List underflow\n");
        return;
    }
    printf("Enter the element to be deleted : ");
    scanf("%d",&item);
    if(item == arr[n-1]) /*Deletion at the end*/
    {
        n = n-1;
        return;
    }
}

```

```

position=search(item);
if(position == 0)
{
    printf("Element not present in array\n");
    return;
}
/*Deletion in between */
temp=position-1;
while(temp <= n-1)
{
    arr[temp] = arr[temp+1]; /* Shifting left */
    temp++;
}
n=n - 1;
}/*End of del()*/
display()
{
    int i;
    if(n == 0)
    {
        printf("List is empty\n");
        return;
    }
    for(i = 0; i< n; i++)
        printf("Value at position %d : %d\n", i+1, arr[i]);
}/*End of display()*/

```

Linked List

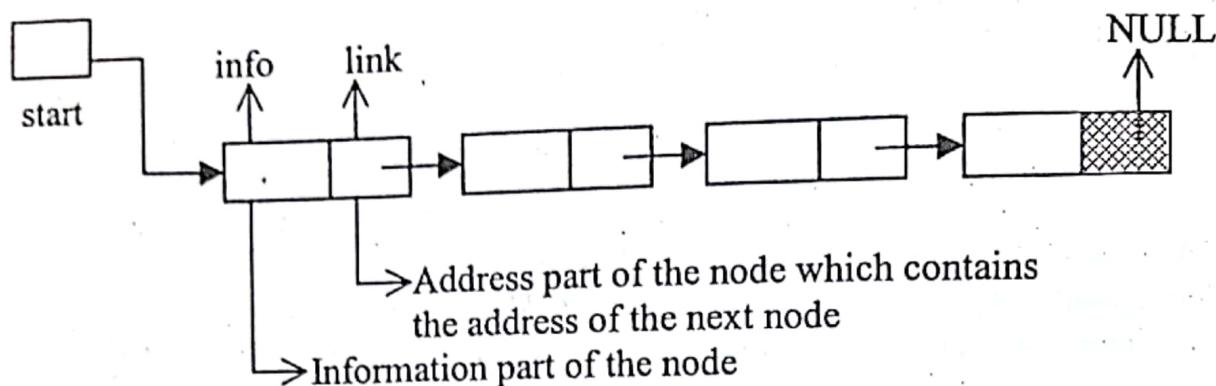
The second way for implementing a list in memory is to use a structure which is self referential structure .One member of this structure is a pointer that points to the structure itself.

```

struct node {
    int data;
    struct node *link;
};

```

Here member of the structure `struct node *link` points to the structure itself. This type of structure is called linked list. A linked list is a collection of elements called nodes. Each node has two parts, first part contains the information field and second part contains the address of the next node .The address part of last node of linked list will have NULL value.



Traversing a Linked List-

In linked list, `info` is the information part, `link` is the address of the next element .We take `ptr` as a pointer variable. Here `start` is a pointer, which points to the first element of the list. Initially we assign the value of `start` to `ptr`. So now `ptr` also points to the first element of linked list. For processing the next element we assign the address of the next element to the `ptr` as –

```
ptr=ptr->link;
```

Now `ptr` has address of the next element .We can traverse each element of linked list through this assignment until `ptr` has `NULL` address which is link part value of last element. So the linked list can be traversed as-

```
while( ptr != NULL )
    ptr=ptr->link;
```

Searching into a Linked List-

Searching refers to search an element in a linked list .For searching the element we first traverse the linked list and with traversing we compare the `info` part of each element with the given element. It can be written as-

```
while( ptr != NULL )
{
    if ( ptr->info == data)
        -----
    else
        ptr=ptr->link;
}
```

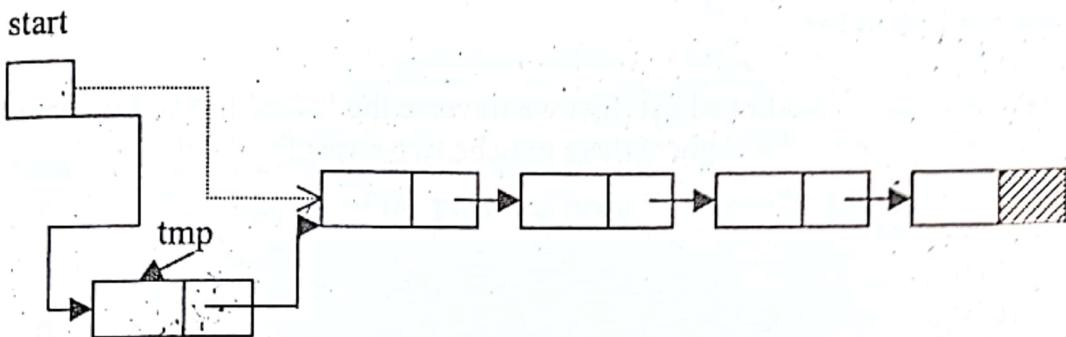
Here `data` is the element which we want to search.

Insertion into a Linked List-

Insertion in a linked list may be possible in two ways-

1. Insertion at beginning
2. Insertion in between

Case 1-



tmp is pointer which points to the node that has to be inserted.

tmp->info=data;

start points to the first element of linked list. For insertion at beginning, we assign the value of start to the link part of inserted node as –

tmp->link=start;

Now inserted node points to the next node which was beginning node of the linked list.

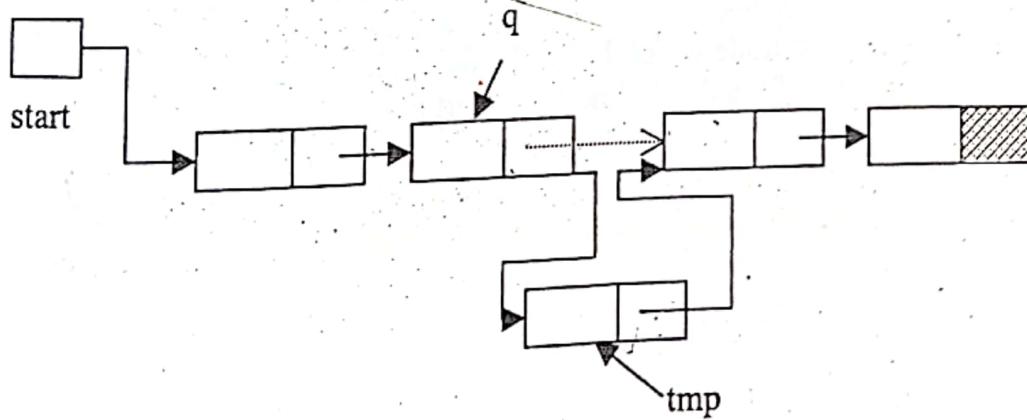
Now inserted node is the first node of the linked list. So start will be reassigned as –

start = tmp

Now start will point to the inserted node which is first node of the linked list.

Dotted line represents the link before insertion.

Case 2-



First we traverse the linked list for obtaining the node after which we want to insert the element. We obtain pointer q which points to the element after which we have to insert new node. For inserting the element after the node, we give the link part of that node to the link part of inserted node and the address of the inserted node is placed into the link part of the previous node.

```

tmp->info=data;
tmp->link=q->link;
q->link=tmp;

```

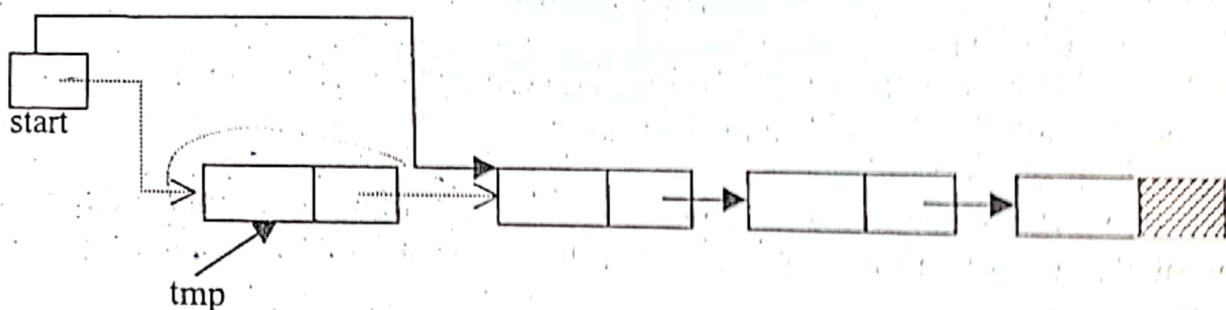
Here q is pointing the previous node. After statement 2, link of inserted node will point to the next node and after statement 3 link of previous node will point to the inserted node.

Deletion from a linked list -

For deleting the node from a linked list ,first we traverse the linked list and compare with each element. After finding the element there may be two cases for deletion-

1. Deletion at beginning
2. Deletion in between

Case 1-



Here start points to the first element of linked list. If element to be deleted is the first element of linked list then we assign the value of start to tmp as-

$\text{tmp} = \text{start};$

So now tmp points to the first node which has to be deleted.

Now we assign the link part of the deleted node to start as-

$\text{start} = \text{start} \rightarrow \text{link};$

Since start points to the first element of linked list, so $\text{start} \rightarrow \text{link}$ will point to the second element of linked list. Now we should free the element to be deleted which is pointed by tmp.

`free(tmp);`

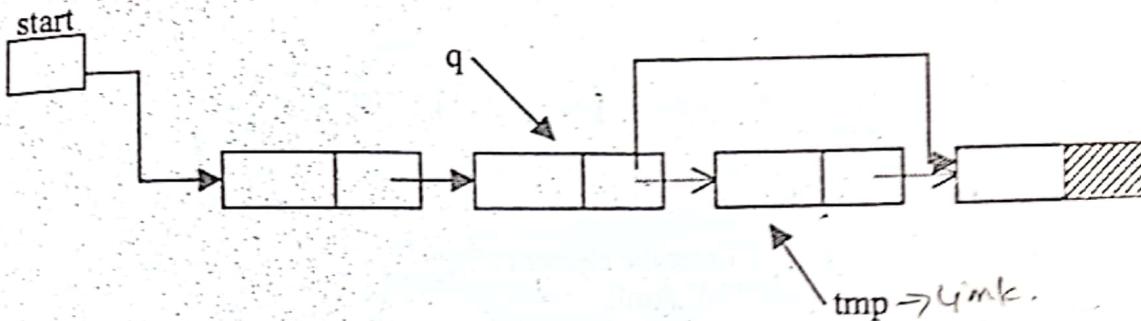
So the whole process for deletion of first element of linked list will be-

```

tmp = start;
start = start->link;
free( tmp );

```

Case 2-



If the element is other than the first element of linked list then we give the link part of the deleted node to the link part of the previous node. This can be as-

```
tmp = q->link;
q->link = tmp->link;
free(tmp);
```

Here q is pointing to the previous node of node to be deleted. After statement 1 tmp will point to the node to be deleted. After statement 2 link of previous node will point to next node of the node to be deleted.

If node to be deleted is last node of linked list then statement 2 will be as-

```
q->link = NULL;
```

```
/* Program of single linked list*/
#include <stdio.h>
#include <malloc.h>

struct node
{
    int info;
    struct node *link;
}*start;

main()
{
    int choice,n,m,position,i;
    start=NULL;
    while(1)
    {
        printf("1.Create List\n");
        printf("2.Add at begining\n");
        printf("3.Add after \n");
        printf("4.Delete\n");
        printf("5.Display\n");
        printf("6.Count\n");
        printf("7.Reverse\n");
        printf("8.Search\n");
        printf("9.Quit\n");
    }
}
```

```

printf("Enter your choice : ");
scanf("%d",&choice);
switch(choice)
{
    case 1:
        printf("How many nodes you want : ");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
            printf("Enter the element : ");
            scanf("%d",&m);
            create_list(m);
        }
        break;
    case 2:
        printf("Enter the element : ");
        scanf("%d",&m);
        addatbeg(m);
        break;
    case 3:
        printf("Enter the element : ");
        scanf("%d",&m);
        printf("Enter the position after which this element is inserted : ");
        scanf("%d",&position);
        addafter(m,position);
        break;
    case 4:
        if(start==NULL)
        {
            printf("List is empty\n");
            continue;
        }
        printf("Enter the element for deletion : ");
        scanf("%d",&m);
        del(m);
        break;
    case 5:
        display();
        break;
    case 6:
        count();
        break;
    case 7:
        rev();
        break;
    case 8:
        printf("Enter the element to be searched : ");
        scanf("%d",&m);
        search(m);
}

```

```

        break;
case 9:
    exit( );
default:
    printf("Wrong choice\n");
}/*End of switch */
}/*End of while */
}/*End of main()*/

```

create_list(int data) ✓

```

{
    struct node *q,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=NULL;
}

```

```

if(start==NULL) /*If list is empty */
    start=tmp;
else
{ /*Element inserted at the end */
    q=start;
    while(q->link!=NULL)
        q=q->link;
    q->link=tmp;
}
}/*End of create_list()*/

```

addatbeg(int data)

```

{
    struct node *tmp;
    tmp=malloc(sizeof(struct node));
    tmp->info=data;
    tmp->link=start;
    start=tmp;
}
}/*End of addatbeg()*/

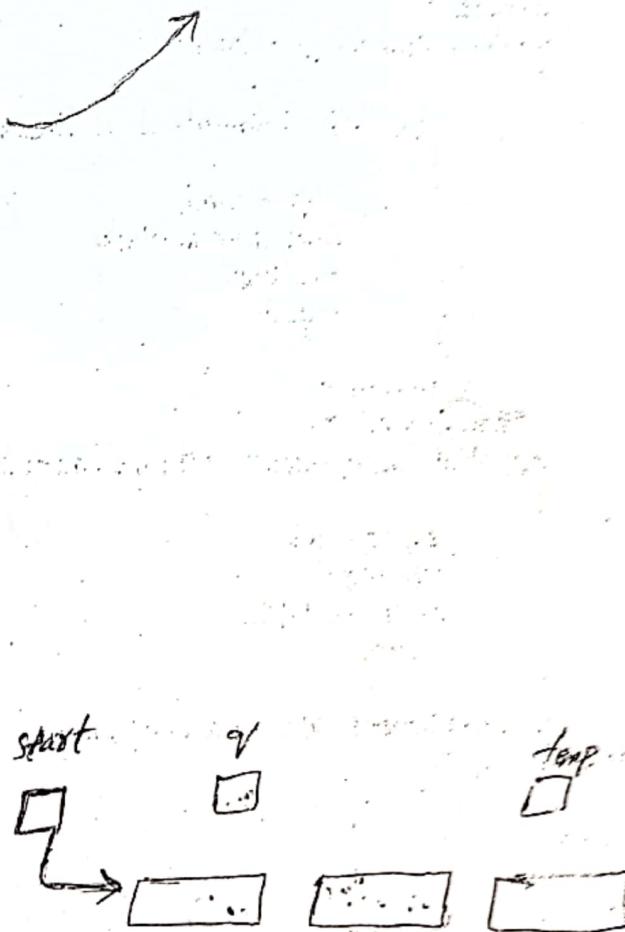
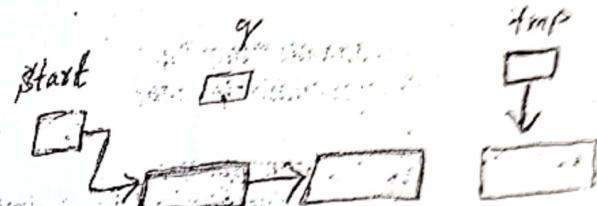
```

addafter(int data,int pos)

```

{
    struct node *tmp,*q;
    int i;
    q=start;
    for(i=0;i<pos-1;i++)
    {
        q=q->link;
        if(q==NULL)
        {
            printf("There are less than %d elements",pos);
            return;
        }
    }
}/*End of for*/

```



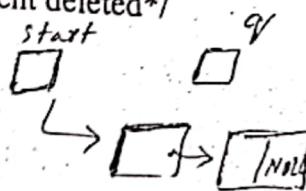
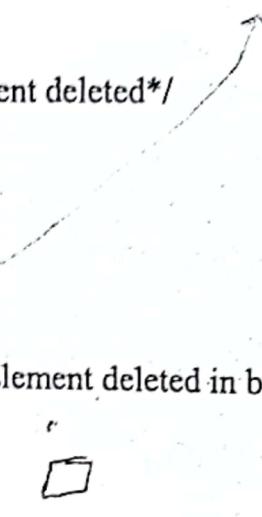
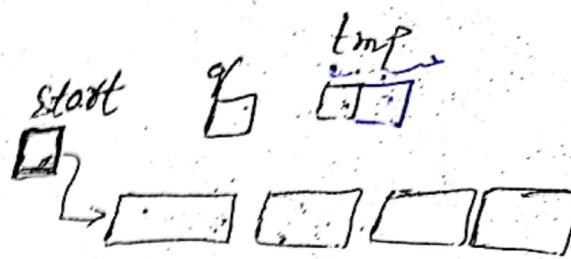
```

tmp=malloc(sizeof(struct node));
tmp->link=q->link;
tmp->info=data;
q->link=tmp;
}/*End of addafter( )*/

del(int data)
{
    struct node *tmp,*q;
    if(start->info == data)
    {
        tmp=start;
        start=start->link; /*First element deleted*/
        free(tmp);
        return;
    }
    q=start;
    while(q->link->link != NULL)
    {
        if(q->link->info==data) /*Element deleted in between*/
        {
            tmp=q->link;
            q->link=tmp->link;
            free(tmp);
            return;
        }
        q=q->link;
    }/*End of while */
    if(q->link->info==data) /*Last element deleted*/
    {
        tmp=q->link;
        free(tmp);
        q->link=NULL;
        return;
    }
    printf("Element %d not found\n",data);
}/*End of del()*/

display()
{
    struct node *q;
    if(start == NULL)
    {
        printf("List is empty\n");
        return;
    }
    q=start;
    printf("List is :\n");
}

```



```

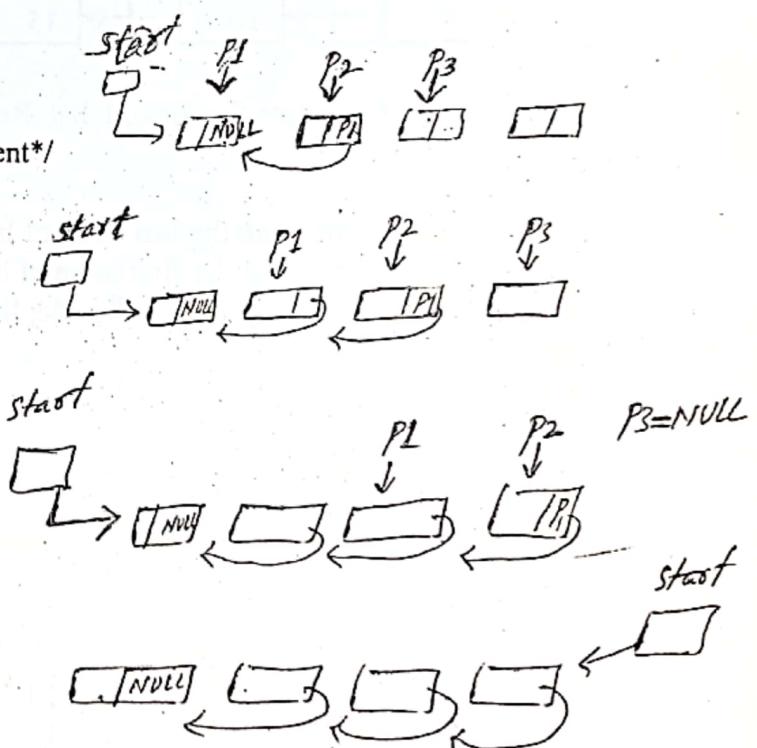
while(q!=NULL)
{
    printf("%d ", q->info);
    q=q->link;
}
printf("\n");
/*End of display() */

count()
{
    struct node *q=start;
    int cnt=0;
    while(q!=NULL)
    {
        q=q->link;
        cnt++;
    }
    printf("Number of elements are %d\n",cnt);
}/*End of count() */

rev()
{
    struct node *p1,*p2,*p3;
    if(start->link==NULL) /*only one element*/
        return;
    p1=start;
    p2=p1->link;
    p3=p2->link;
    p1->link=NULL;
    p2->link=p1;
    while(p3!=NULL)
    {
        p1=p2;
        p2=p3;
        p3=p3->link;
        p2->link=p1;
    }
    start=p2;
}/*End of rev() */

search(int data)
{
    struct node *ptr = start;
    int pos = 1;
    while(ptr!=NULL)
    {
        if(ptr->info==data)
        {
            printf("Item %d found at position %d\n",data,pos);
        }
    }
}

```



```

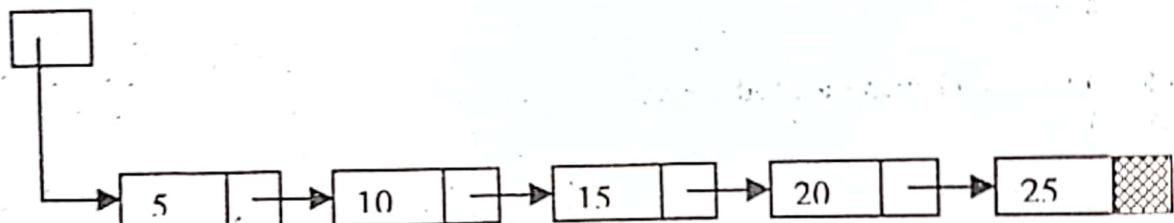
        return;
    }
    ptr = ptr->link;
    pos++;
}
if(ptr == NULL)
    printf("Item %d not found in list\n",data);
/*End of search()*/

```

Reverse linked list

Let us take a linked list-

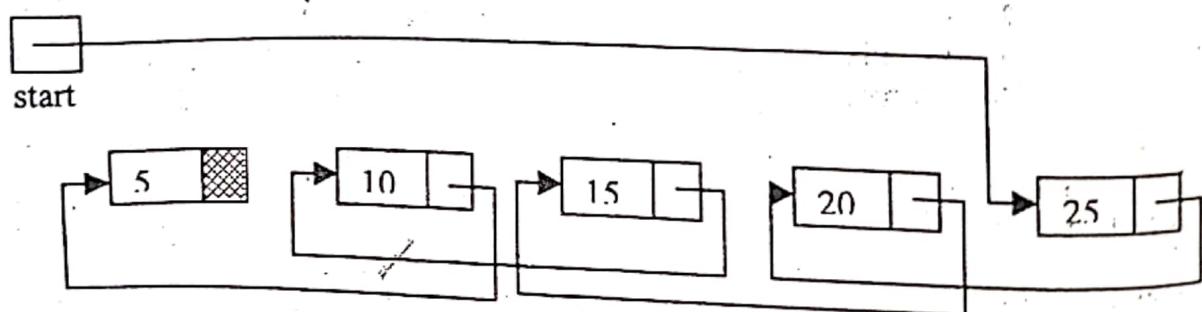
start



Now we want to reverse this linked list. Reverse of this linked list will do the following things-

1. First node will become the last node of linked list.
2. Last node will become the first node of linked list and now start will point to it.
3. Link of 2nd node will point to 1st node, link of 3rd node will point to second node and so on.
4. Link of last node will point to the previous node of last node in linked list.

Now reversed linked list will be as-



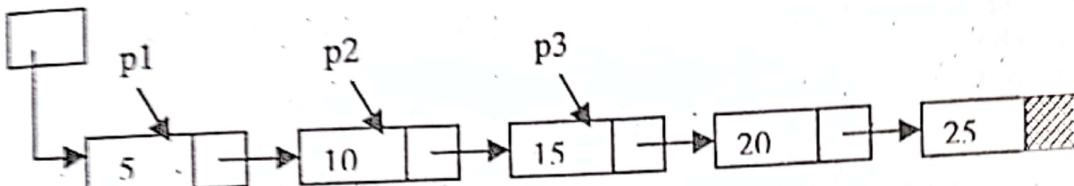
Creation of reverse()

We will take three pointers p1, p2 and p3. Initially p1, p2 and p3 will point to first, second and third node of linked list.

Linked List

```
p1 = start;
p2 = p1->link;
p3 = p2->link;
```

Start



Since in reverse list first node will become the last node , so link part of first node should be NULL.

$p1->link=NULL;$

Now link of second node should point to first node hence

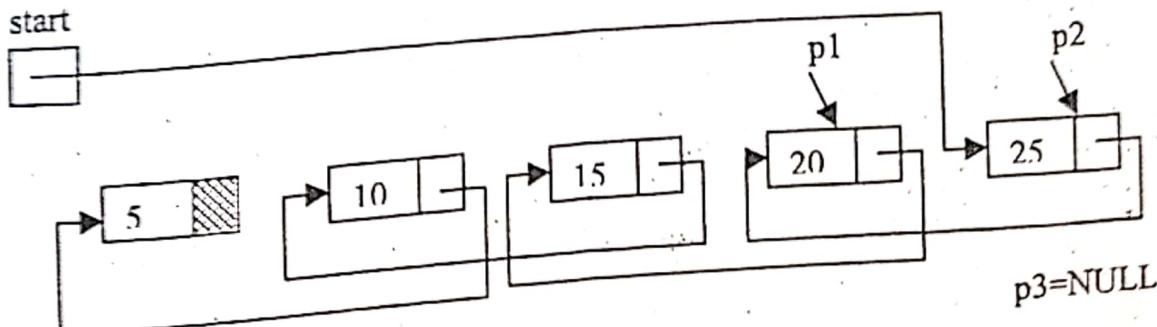
$p2->link=p1;$

Now we will traverse the linked list with p3 pointer and we shift pointers p1 and p2 forward. We assign p1 to the link part of p2, so that link of each node will now point to its previous node.

```
p2->link=p1;
while(p3!=NULL)
{
    p1=p2;
    p2=p3;
    p3=p3->link;
    p2->link=p1;
}
```

When this loop will terminate p3 will be NULL, p2 will point to last node, and link of each node will now point to its previous node. Now start should point to the last node of the linked list which is first node of reversed linked list.

$start=p2;$



If the list contains only one element then there will be a problem in initializing p3, hence we check this condition in the beginning-

```

if( start->link == NULL )
    return;

/* Program of reverse linked list*/
# include <stdio.h>
# include <malloc.h>

struct node
{
    int info;
    struct node *link;
}*start;

main( )
{
    int i,n,item;
    start=NULL;

    printf("How many nodes you want : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter the item %d : ",i+1);
        scanf("%d",&item);
        create_list(item);
    }
    printf("Initially the linked list is :\n");
    display( );
    reverse( );
    printf("Linked list after reversing is :\n");
    display( );
}/*End of main()*/

create_list(int num)
{
    struct node *q,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info=num;
    tmp->link=NULL;

    if(start==NULL)
        start=tmp;
    else
    {
        q=start;

```

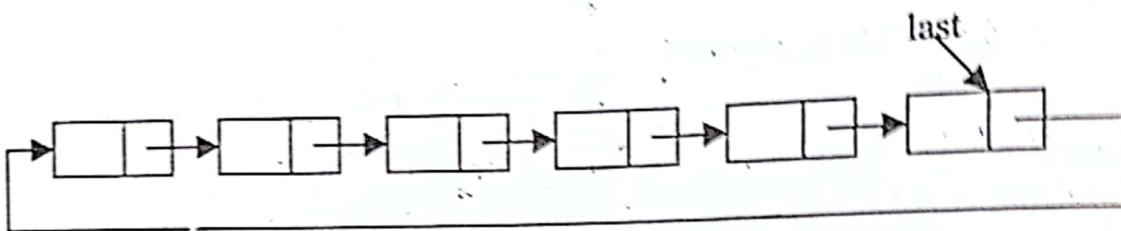
```
        while(q->link!=NULL)
            q=q->link;
            q->link=tmp;
    }
}/*End of create_list()*/
display()
{
    struct node *q;
    if(start == NULL)
    {
        printf("List is empty\n");
        return;
    }
    q=start;
    while(q!=NULL)
    {
        printf("%d ", q->info);
        q=q->link;
    }
    printf("\n");
}/*End of display()*/
reverse()
{
    struct node *p1,*p2,*p3;
    if(start->link==NULL) /*only one element*/
        return;
    p1=start;
    p2=p1->link;
    p3=p2->link;

    p1->link=NULL;
    p2->link=p1;

    while(p3!=NULL)
    {
        p1=p2;
        p2=p3;
        p3=p3->link;
        p2->link=p1;
    }
    start=p2;
}/*End of reverse() */
```

Circular linked list-

As we have seen in a single linked list, for accessing any node of linked list, we start traversing from first node. Suppose we are at the last node of linked list and we want to access the second node of linked list then again we start traversing from first node. So we can not access the previous node because it traverse only in one direction up to the last node. One useful data structure circular linked list is implemented to overcome this problem. It has only small change. In linked list, link part of last node has NULL value, it represents the last node of linked list, but here link part points to the first node of linked list and represents the linked list in circular way.



Now we can access any node of the linked list without going back and start traversal again from first node because it's direction is just like circle and we can traverse from last node to first node. We can see there is no first and last node but conventionally we maintain it. We can maintain the pointer at the last node only. It's link part will point to the first node. It will be very useful for implementation of stack and queue because we have a need of top in stack and rear in queue.

Creation of circular linked list-

Creation of circular linked list is same as single linked list. Only one thing is needed that here last node will always point to first node instead of NULL. So we maintain one pointer last which points to last node of list and link part of this node points to the first node of list. New element can be added as-

```

tmp->link = last->link; /*added at the end of list*/
last->link = tmp;
last = tmp;
  
```

After statement 1 link of added node will point to the first node of list. After statement 2 link of previous node will point to the added node and after statement 3 pointer variable last will point to the added node which is now the last node of list.

Traversal in circular linked list-

Traversal in list has a need of pointer variable which points to first node of list. Here we maintain the pointer last which points to last node. But link part of this last pointer points to first node of list, so we can assign this value to pointer variable which will point to first node. Now we can traverse the list until the last node of list comes.

```
q = last->link;  
while(q != last)  
{  
.....  
q = q->link;  
}
```

Insertion into a circular Linked List-

Insertion in a circular linked list may be possible in two ways-

1. Insertion at beginning
2. Insertion in between

Case 1-

Insertion at beginning is needed to assign the address of inserted node to link part of last node. Insertion at beginning is as-

```
tmp->info = num;  
tmp->link = last->link;  
last->link = tmp;
```

After statement 2 inserted node points to the previous first node of list and now after statement 3 link part of last node will point to the inserted node which is first node of circular linked list.

Case 2-

Insertion in between is same as in single linked list. This can be as-

```
tmp->link = q->link;  
tmp->info = num;  
q->link = tmp;
```

Here q points to the node after which new node will be inserted.

Deletion from circular linked list-

Deletion from circular linked list is little bit different from single linked list. Here we have a need to take care of some more conditions because of it's circular behaviour.

These are the cases for deletion-

1. If list has only one element
2. Node to be deleted is the first node of list
3. Deletion in between
4. Node to be deleted is last node of list

Case 1-

Here we check the condition for only one element of list then assign NULL value to last pointer because after deletion no node will be in list.

```

if( last->link == last && last->info == num) /*Only one element*/
{
    tmp = last;
    last = NULL;
    free(tmp);
}

```

Case 2-
Here we assign the link part of deleted node to the link part of pointer last. So now link part of last pointer will point to the next node which is now first node of list after deletion.

```

q = last->link;
if(q->info == num)
{
    tmp = q;
    last->link = q->link;
    free(tmp);
}

```

After statement 1, q is pointing to the first node of list.

Case 3-

Deletion in between is same as in single linked list. Deletion of node in between will be as-

```

while(q->link != last)
{
    if(q->link->info == num) /*Element deleted in between*/
    {
        tmp = q->link;
        q->link = tmp->link;
        free(tmp);
    }
    q = q->link;
}/*End of while*/

```

First we are traversing the list, when we find the element to be deleted, then q points to the previous node. We assign the link part of node to be deleted to the link part of previous node and then we free the address of node to be deleted from memory.

Case 4-

Deletion of last node requires to assign the link part of last node to the link part of previous node. So link part of previous node will point to the first node of list. Then assign the value of previous node to the pointer variable last because after deletion of last node pointer variable last should point to the previous node.

```

tmp = q->link;
q->link = last->link;
free(tmp);
last = q;

```

Here q is pointing to previous node of last node. After statement 1 tmp will point to last node. After statement 2 link part of previous node will point to the first node of list and after statement 4 pointer variable last will point to the previous node which is now last node of list.

```

/* Program of circular linked list*/
#include <stdio.h>
#include <malloc.h>

struct node
{
    int info;
    struct node *link;
}*last;

main()
{
    int choice,n,m,po,i;
    last=NULL;
    while(1)
    {
        printf("1.Create List\n");
        printf("2.Add at begining\n");
        printf("3.Add after \n");
        printf("4.Delete\n");
        printf("5.Display\n");
        printf("6.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                printf("How many nodes you want : ");
                scanf("%d",&n);
                for(i=0; i < n;i++)
                {
                    printf("Enter the element : ");
                    scanf("%d",&m);
                    create_list(m);
                }
                break;
            case 2:
                printf("Enter the element : ");

```

```

        scanf("%d",&m);
        addatbeg(m);
        break;
    case 3:
        printf("Enter the element : ");
        scanf("%d",&m);
        printf("Enter the position after which this element is inserted : ");
        scanf("%d",&po);
        addafter(m,po);
        break;
    case 4:
        if(last == NULL)
        {
            printf("List underflow\n");
            continue;
        }
        printf("Enter the number for deletion : ");
        scanf("%d",&m);
        del(m);
        break;
    case 5:
        display();
        break;
    case 6:
        exit();
    default:
        printf("Wrong choice\n");
    }/*End of switch*/
}/*End of while*/
}/*End of main()*/
}

create_list(int num)
{
    struct node *q,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info = num;

    if(last == NULL)
    {
        last = tmp;
        tmp->link = last;
    }
    else
    {
        tmp->link = last->link; /*added at the end of list*/
        last->link = tmp;
        last = tmp;
    }
}/*End of create_list()*/
}

```

```

addatbeg(int num)
{
    struct node *tmp;
    tmp = malloc(sizeof(struct node));
    tmp->info = num;
    tmp->link = last->link;
    last->link = tmp;
}/*End of addatbeg( )*/

addafter(int num,int pos)
{
    struct node *tmp,*q;
    int i;
    q = last->link;
    for(i=0; i < pos-1; i++)
    {
        q = q->link;
        if(q == last->link)
        {
            printf("There are less than %d elements\n",pos);
            return;
        }
    }/*End of for*/
    tmp = malloc(sizeof(struct node));
    tmp->link = q->link;
    tmp->info = num;
    q->link = tmp;
    if(q==last) /*Element inserted at the end*/
        last=tmp;
}/*End of addafter( )*/

del(int num)
{
    struct node *tmp,*q;
    if( last->link == last && last->info == num) /*Only one element*/
    {
        tmp = last;
        last = NULL;
        free(tmp);
        return;
    }
    q = last->link;
    if(q->info == num)
    {
        tmp = q;
        last->link = q->link;
        free(tmp);
    }
}

```

*let
Peb*

```

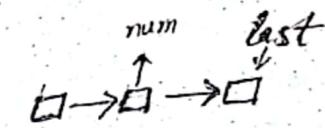
        return;
    }
    while(q->link != last)
    {
        if(q->link->info == num) /*Element deleted in between*/
        {
            tmp = q->link;
            q->link = tmp->link;
            free(tmp);
            printf("%d deleted\n",num);
            return;
        }
        q = q->link;
    }/*End of while*/
    if(q->link->info == num) /*Last element deleted q->link=last*/
    {
        tmp = q->link;
        q->link = last->link;
        free(tmp);
        last = q;
        return;
    }
    printf("Element %d not found\n",num);
}/*End of del()*/
}

display()
{
    struct node *q;
    if(last == NULL)
    {
        printf("List is empty\n");
        return;
    }
    q = last->link;
    printf("List is :\n");
    while(q != last)
    {
        printf("%d ", q->info);
        q = q->link;
    }
    printf("\n%d\n",last->info);
}/*End of display()*/

```

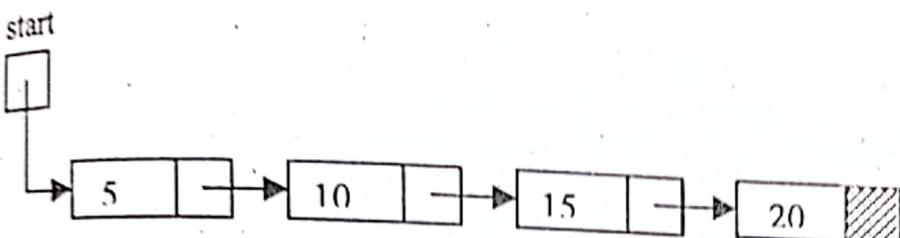
Sorted linked list-

Suppose we want to maintain linked list in sorted order then we have to keep in mind that when we add the element in the linked list then it should be inserted at the proper place.



Every time when we will add the element it will be inserted at the proper place and linked list will be in sorted order.

Let us take a linked list-

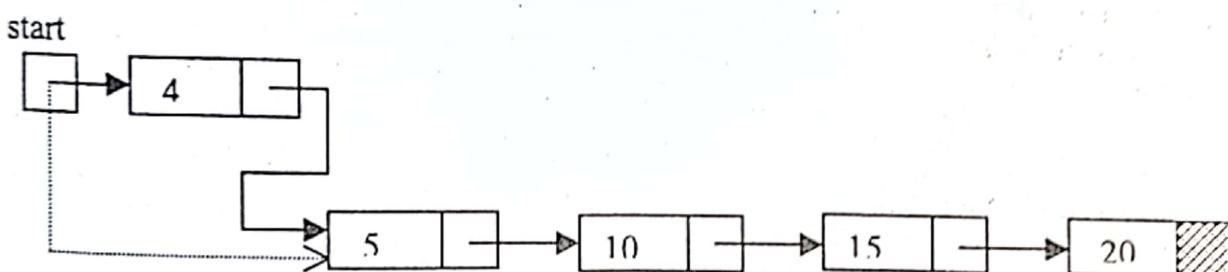


There will be two cases for adding the element-

1. List is empty or element value is less than the value of first node.
2. The element value lies in between or at the end of list.

Case 1-

Added element is 4.



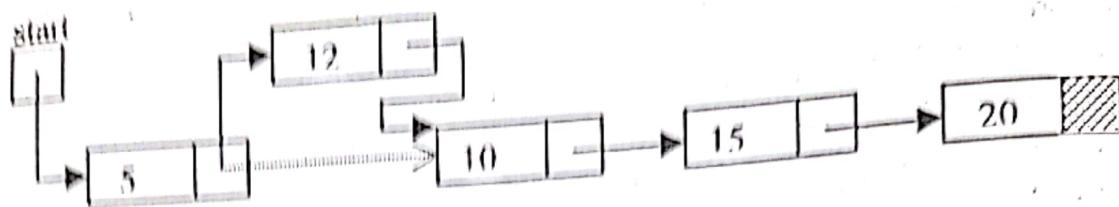
Here in any case if list is empty or element value is less than first node the inserted element will be first node. It is same as insertion of a node at the beginning in single linked list. We assign the value of start to the link part of inserted node, so if list is empty then link part of inserted node will be NULL. We assign the value of inserted node to start because now start will point to the new inserted node.

```

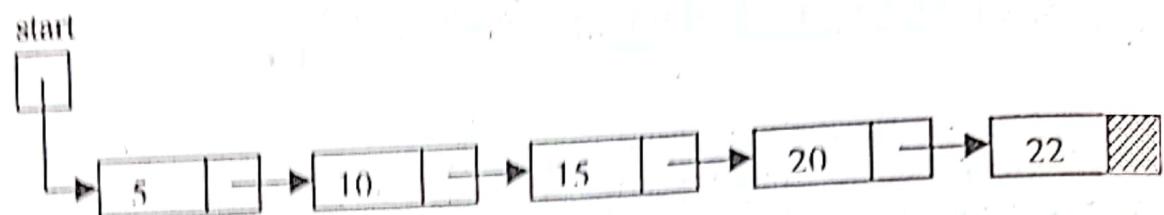
if(start == NULL || num < start->info)
{
    tmp->link=start;
    start=tmp;
}
  
```

Case 2-

This will be same as the insertion of node in between for single linked list.
Added element is 12



Added element is 22.



The other operations traversal, deletion will be same as linked list operations.

```
/* Program of sorted linked list*/
#include <stdio.h>
#include <malloc.h>
struct node
{
    int info;
    struct node *link;
}*start;
main()
{
    int choice,n,m,i;
    start=NULL;
    while(1)
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Exit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
                printf("Enter the element to be inserted : ");
                scanf("%d",&m);
                insert(m);
                break;
            case 2:
                printf("Enter the element to be deleted : ");
        }
    }
}
```

```
        scanf("%d",&m);
        del(m);
        break;
    case 3:
        display();
        break;
    case 4:
        exit();
    default:
        printf("Wrong choice\n");
    }/*End of switch*/
}/*End of while*/
} /*end of main()*/
```

```
insert(int num)
{
    struct node *q,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info=num;

    /*list empty or item to be added in begining */
    if(start == NULL || num < start->info)
    {
        tmp->link=start;
        start=tmp;
        return;
    }
    else
    {
        q=start;
        while(q->link != NULL && q->link->info < num)
            q=q->link;
        tmp->link=q->link;
        q->link=tmp;
    }
}/*End of insert( )*/
```

```
del(int num)
{
    struct node *tmp,*q;
    if(start->info==num)
    {
        tmp=start;
        start=start->link; /*first element deleted*/
        free(tmp);
        return;
    }
    q=start;
```

```

while(q->link->link!=NULL)
{
    if(q->link->info==num) /*element deleted in between*/
    {
        tmp=q->link;
        q->link=tmp->link;
        free(tmp);
        return;
    }
    q=q->link;
}/*End of while */
if(q->link->info==num) /*last element deleted*/
{
    tmp=q->link;
    free(tmp);
    q->link=NULL;
    return;
}
printf("Element %d not found\n",num);
}/*End of del()*/
display()
{
    struct node *q;
    if(start == NULL)
    {
        printf("List is empty\n");
        return;
    }
    q=start;
    printf("List is :\n");
    while(q != NULL)
    {
        printf("%d ", q->info);
        q=q->link;
    }
    printf("\n");
}/*End of display()*/

```

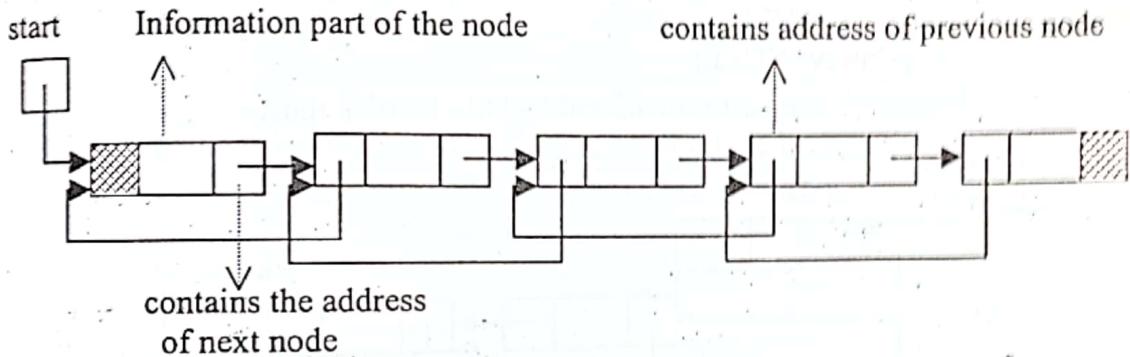
Double linked list-

We have seen in single linked list we can traverse only in one direction because each node has address of next node only. Suppose we are in the middle of linked list and we want to do operation with just previous node then we have no way to go on previous node, we will again traverse from starting node. So this is a drawback of single linked list. We implement another data structure doubly linked list, in this each node has address of previous and next node also. The data structure for doubly linked list will be as-

```
struct node{
    struct node *previous;
    int info;
    struct node *next;
}
```

Here `struct node *previous` is a pointer to structure, which will contain the address of previous node and `struct node *next` will contain the address of next node in the list. So we can traverse in both directions at any time.

The only drawback is that each node has to contain the address information of previous node.



Traversing a doubly linked List-

In doubly linked list `info` is the information part, `next` is the address of the next node and `prev` is the address of previous node. We take `ptr` as a pointer variable. Here `start` points to the first element of list. Initially we assign the value of `start` to `ptr`. So `ptr` also points to the first node of list. For processing the next element we assign the address of next node to `ptr` as –

`ptr=ptr->next;`

Now `ptr` has the address of next node. We can traverse each element of list through this assignment until `ptr` has `NULL` value which is `next` part value of last element. So the doubly linked list can be traversed as-

```
while( ptr != NULL )
    ptr=ptr->next;
```

Insertion into a doubly linked List-

Insertion in a doubly linked list may be possible in two ways-

1. Insertion at begining
2. Insertion in between

Case 1-

start points to the first node of doubly linked list. For insertion at beginning, we assign the value of start to the next part of inserted node and address of inserted node to the prev part of start as -

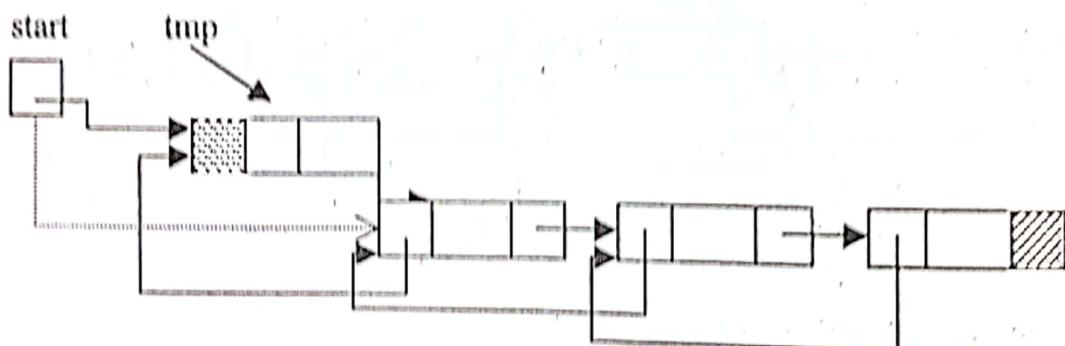
```
tmp->next=start;
start->prev = tmp;
```

Now inserted node points to the next node, which was beginning node of the doubly linked list and prev part of second node will point to the new inserted node. Now inserted node is the first node of the doubly linked list. So start will be reassigned as -

```
start= tmp;
```

Now start will point to the inserted node which is first node of the doubly linked list. Assign NULL to prev part of inserted node since now it will become the first node and prev part of first node is NULL

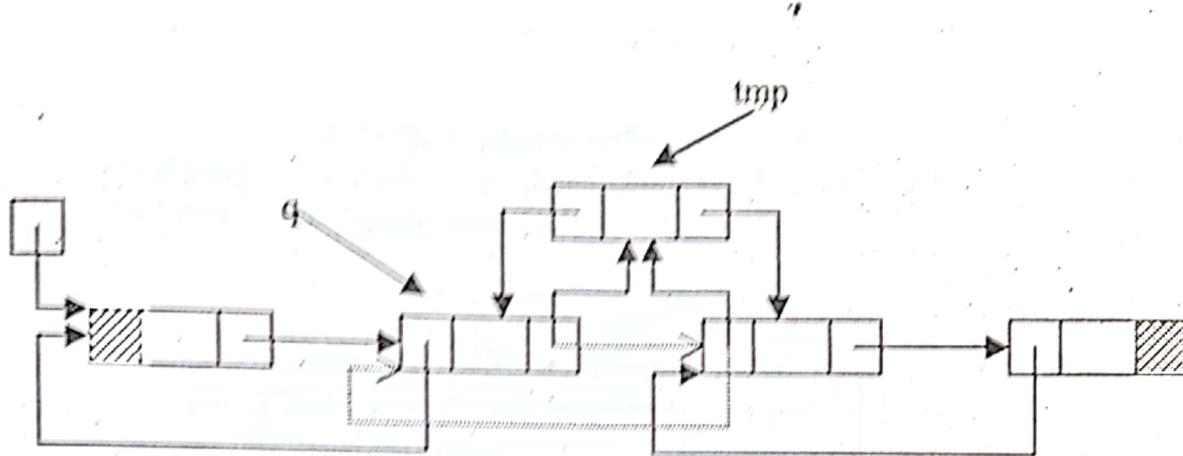
```
tmp->prev=NULL;
```

**Case 2-**

First we traverse the doubly linked list for obtaining the node after which we want to insert the element. For inserting the element after the node we assign the address of inserted node to the prev part of next node. Then we assign the next part of previous node to the next part of inserted node. Address of previous node will be assigned to prev part of inserted node and address of inserted node will be assigned to next part of previous node.

```
q->next->prev=tmp;
tmp->next=q->next;
tmp->prev=q;
q->next=tmp;
```

Here q points to the previous node (after that new node will be inserted). After statement 1, prev part of next node will point to inserted node. After statement 2, next part of inserted node will point to next node. After statement 3, prev part of inserted node will point to its previous node and after statement 4, next part of previous node will point to inserted node.



Deletion from doubly linked list -

For deleting the node from a doubly linked list ,first we traverse the linked list and compare with each element. After finding the element there may be three cases for deletion-

1. Deletion at beginning
2. Deletion in between
3. Deletion of last node

Case 1-

Here start points to the first node of doubly linked list. If node to be deleted is the first node of list then we assign the value of start to tmp as-

```
tmp = start;
```

Now we assign the next part of deleted node to start as-

```
start = start->next;
```

Since start points to the first node of linked list, so start->next will point to the second node of list. Then NULL will be assigned to start->prev. Now we should free the node to be deleted which is pointed by tmp,

```
free( tmp );
```

So the whole process for deletion of first node of doubly linked list will be-

```
tmp = start;
start = start->next; /*first element deleted*/
start->prev = NULL;
free(tmp);
```

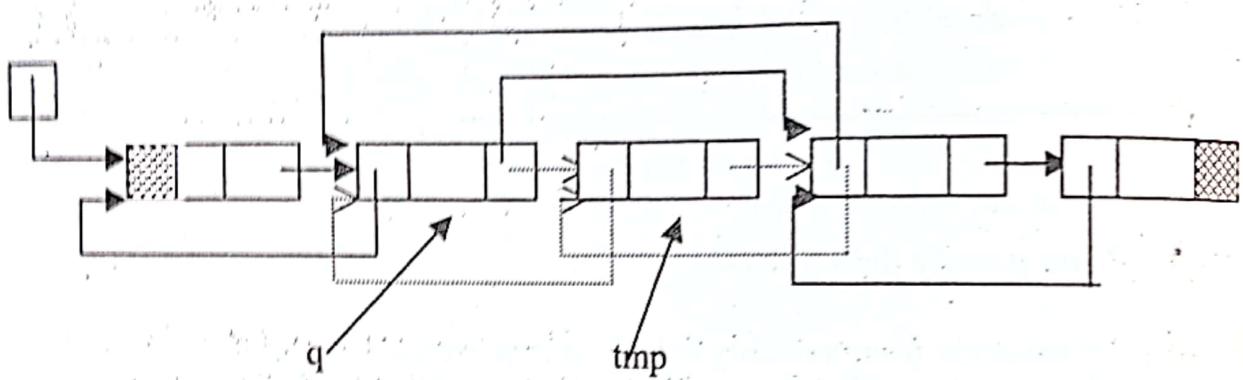
Case 2-

If the element is other than the first element of linked list then we assign the next part of the deleted node to the next part of the previous node and address of the previous node to prev part of next node. This can be as-

```
tmp = q->next;
q->next = tmp->next;
```

```
tmp->next->prev=q;
free(tmp);
```

Here q is pointing to the previous node of node to be deleted. After statement 1 tmp will point to the node to be deleted. After statement 2 next part of previous node will point to next node of the node to be deleted and after statement 3 prev part of next node will point to previous node.



Case 3-

If node to be deleted is last node of doubly linked list then we will just free the last node and next part of second last node will be NULL.

```
tmp=q->next;
free(tmp);
q->next=NULL;
```

Here q is pointing to the previous node of node to be deleted. After statement 1 tmp will point to the node to be deleted. After statement 2 last node will be deleted and after statement 3 second last node will become the last node of list.

```
/* Program of double linked list*/
#include <stdio.h>
#include <malloc.h>

struct node
{
    struct node *prev;
    int info;
    struct node *next;
}*start;

main()
{
    int choice,n,m,po,i;
    start=NULL;
```

```
while(1)
{
    printf("1.Create List\n");
    printf("2.Add at begining\n");
    printf("3.Add after\n");
    printf("4.Delete\n");
    printf("5.Display\n");
    printf("6.Count\n");
    printf("7.Reverse\n");
    printf("8.exit\n");
    printf("Enter your choice : ");
    scanf("%d",&choice);
    switch(choice)
    {
        case 1:
            printf("How many nodes you want : ");
            scanf("%d",&n);
            for(i=0;i<n;i++)
            {
                printf("Enter the element : ");
                scanf("%d",&m);
                create_list(m);
            }
            break;
        case 2:
            printf("Enter the element : ");
            scanf("%d",&m);
            addatbeg(m);
            break;
        case 3:
            printf("Enter the element : ");
            scanf("%d",&m);
            printf("Enter the position after which this element is inserted : ");
            scanf("%d",&po);
            addafter(m,po);
            break;
        case 4:
            printf("Enter the element for deletion : ");
            scanf("%d",&m);
            del(m);
            break;
        case 5:
            display();
            break;
        case 6:
            count();
            break;
        case 7:
            rev();
    }
}
```

```

        break;
    case 8:
        exit( );
    default:
        printf("Wrong choice\n");
    } /* End of switch */
} /* End of while */
} /* End of main() */

```

create_list(int num)

```

{
    struct node *q,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info=num;
    tmp->next=NULL;
    if(start==NULL)
    {
        tmp->prev=NULL;
        start->prev=tmp;
        start=tmp;
    }
    else
    {
        q=start;
        while(q->next!=NULL)
            q=q->next;
        q->next=tmp;
        tmp->prev=q;
    }
} /* End of create_list() */

```

addatbeg(int num)

```

{
    struct node *tmp;
    tmp=malloc(sizeof(struct node));
    tmp->prev=NULL;
    tmp->info=num;
    tmp->next=start;
    start->prev=tmp;
    start=tmp;
} /* End of addatbeg() */

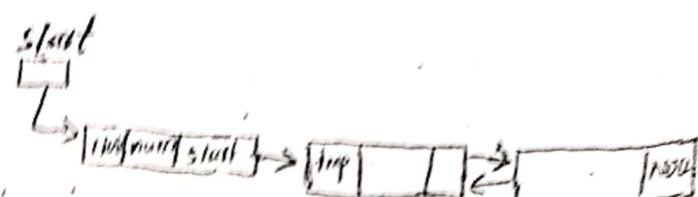
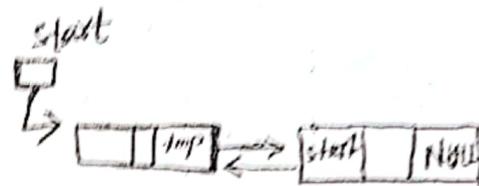
```

addafter(int num,int e)

```

{
    struct node *tmp,*q;
    int i;
    q=start;

```

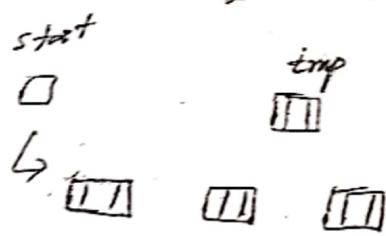


```

for(i=0;i<c-1;i++)
{
    q=q->next;
    if(q==NULL)
    {
        printf("There are less than %d elements\n",c);
        return;
    }
}
tmp=malloc(sizeof(struct node));
tmp->info=num;
q->next->prev=tmp;
tmp->next=q->next;
tmp->prev=q;
q->next=tmp;
/*End of addafter() */

del(int num)
{
    struct node *tmp,*q;
    if(start->info==num)
    {
        tmp=start;
        start=start->next; /*first element deleted*/
        start->prev=NULL;
        free(tmp);
        return;
    }
    q=start;
    while(q->next->next!=NULL)
    {
        if(q->next->info==num) /*Element deleted in between*/
        {
            tmp=q->next;
            q->next=tmp->next;
            tmp->next->prev=q;
            free(tmp);
            return;
        }
        q=q->next;
    }
    if(q->next->info==num) /*last element deleted*/
    {
        tmp=q->next;
        free(tmp);
        q->next=NULL;
        return;
    }
    printf("Element %d not found\n",num);
}/*End of del()*/

```



```

display()
{
    struct node *q;
    if(start==NULL)
    {
        printf("List is empty\n");
        return;
    }
    q=start;
    printf("List is :\n");
    while(q!=NULL)
    {
        printf("%d ", q->info);
        q=q->next;
    }
    printf("\n");
}/*End of display()*/

```

```

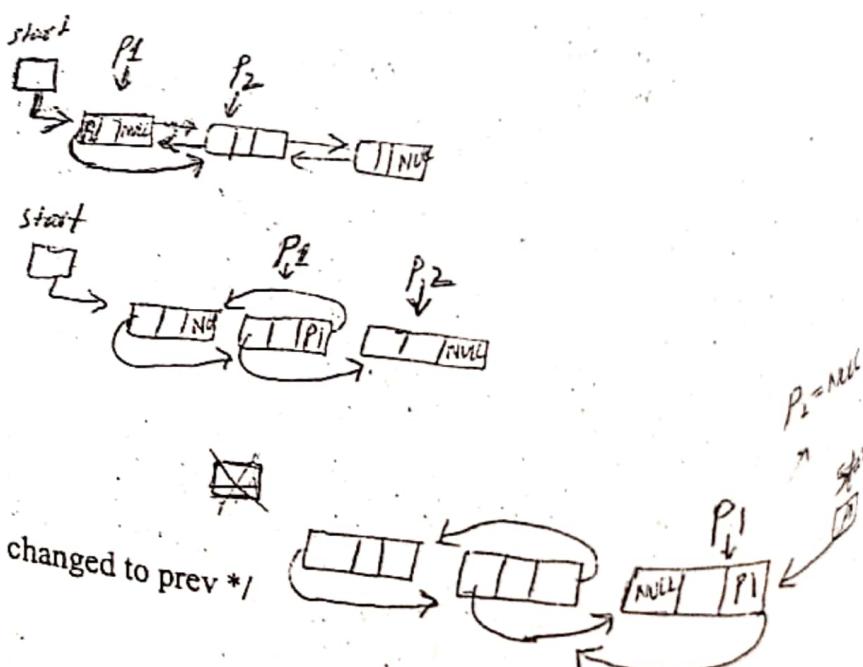
count()
{
    struct node *q=start;
    int cnt=0;
    while(q!=NULL)
    {
        q=q->next;
        cnt++;
    }
    printf("Number of elements are %d\n",cnt);
}/*End of count()*/

```

```

rev()
{
    struct node *p1,*p2;
    p1=start;
    p2=p1->next;
    p1->next=NULL;
    p1->prev=p2;
    while(p2!=NULL)
    {
        p2->prev=p2->next;
        p2->next=p1;
        p1=p2;
        p2=p2->prev; /*next of p2 changed to prev */
    }
    start=p1;
}/*End of rev()*/

```



Polynomial arithmetic with linked list-

One useful application of linear linked list is the representation of polynomial expression. We can use linked list to represent polynomial expression and as well as for arithmetic operations also. Let us take a polynomial expression-

$$5x^4 + x^3 - 6x + 2$$

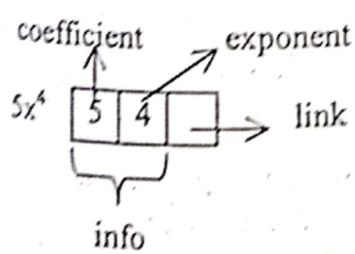
Here we can see every symbol x is attached with two things, coefficient and exponent. As in $5x^4$, coefficient is 5 and exponent is 4. In linked list data structure for representing each node is-

```
struct node{
    int info;
    struct node *link;
}
```

Here info is the information part which can be a record also and link is the pointer which points to the next node. But in polynomial expression info part has a need of only coefficient and exponent and link part will be the same. So we can represent polynomial expression in single linked list where each node of list will contain the coefficient and exponent of each symbol of polynomial expression. So the data structure for polynomial expression will be-

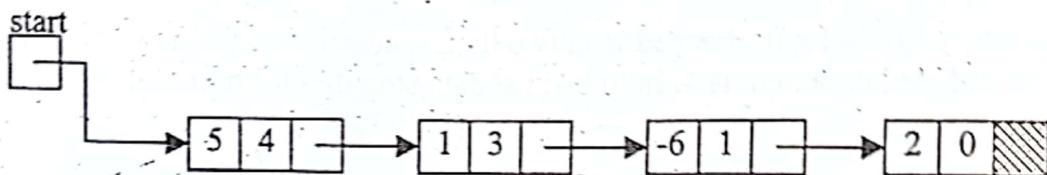
```
struct node{
    int coefficient;
    int exponent;
    struct node *link;
}
```

Let us take the above expression and represent it in nodes-



x^3	1	3	
$-6x$	-6	1	
2	2	0	

Here 2 is represented as $2x^0$ because $x^0 = 1$. For representing the polynomial expression, we will use the descending sorted linked list based on the exponent because it will be easier for arithmetic operation with polynomial linked list. Otherwise, we have a need to traverse the full list for every arithmetic operation. Now we can represent the above polynomial expression as-



Creation of polynomial linked list-

Creation of polynomial linked list will be same as creation of sorted linked list but it will be in descending order and based on exponent of symbol.

```
/*list empty or exp greater than first one */
if(start==NULL || ex > start->expo)
{
    tmp->link=start;
    start=tmp;
    return start;
}
else
{
    ptr=start;
    while(ptr->link!=NULL && ptr->link->expo > ex)
        ptr=ptr->link;
    tmp->link=ptr->link;
    ptr->link=tmp;
    if(ptr->link==NULL) /*item to be added in the end */
        tmp->link=NULL;
}
```

Here in if condition we are checking for the node to be added will be first node or not. In else part we traverse the list and check the condition for exponent then we add the node at proper place in list. If node will be added at the end of list then we assign NULL in link part of added node.

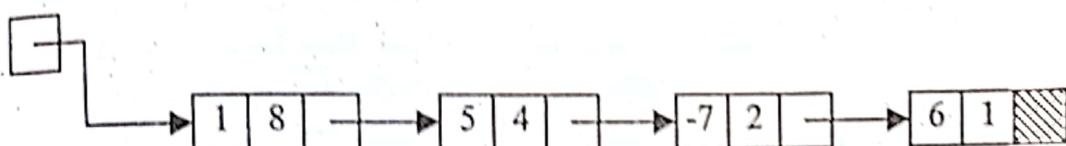
Addition with polynomial linked list-

For addition of two polynomial linked lists, we have a need to traverse the nodes of both the lists. If the node has exponent value higher than another, then that node will be added to the resultant list or we can say that nodes which are unique to both the lists will be added in the resultant list. If the nodes have same exponent value then first the coefficient

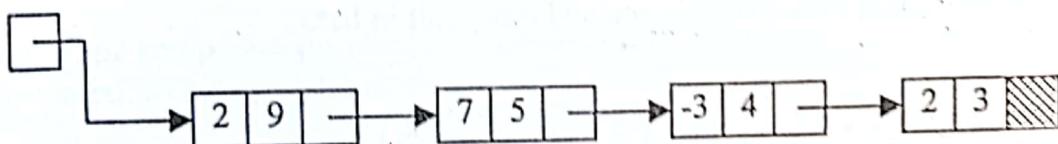
of both nodes will be added then the result will be added to the resultant list. Suppose in traversing one list is finished then remaining node of the another list will be added to the resultant list.

Let us take two polynomial expression lists and add the result in another list.

$$\text{Poly1- } x^8 + 5x^4 - 7x^2 + 6x$$



$$\text{Poly2- } 2x^9 + 7x^5 - 3x^4 + 2x^3$$



We can define the addition of polynomial list as-

Steps-

1. Scan the list1 and list2 one by one
2. Compare the exponent of list1 node with list2 node
 - (i) If one node has higher exponent than another node then add the higher exponent node in list3 and scan the next node in higher exponent node.
 - (ii) If exponents are same, then add the coefficients and add the result in list3 and scan the next nodes in both the lists.
3. Repeat the same process until one list is finished.
4. Add the remaining nodes of another unfinished list in list3.

Let us apply these steps in addition of above two polynomial lists.

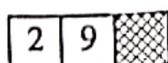
Step 1- Scan the nodes of list1 and list2.

Here exponent of node of list2 is higher than the exponent of node of list1.

$$8 < 9$$

So add the node of list2 in list3.

Lst3-

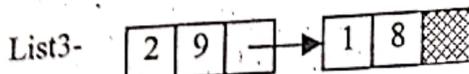


Scan the next node of list2.

Step 2- Compare the exponent of current node of list1 and list2

$$8 > 5$$

Hence add the node of list1 in list3.

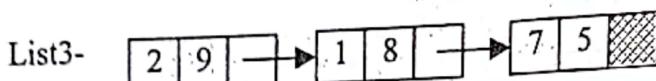


Scan the next node of list1.

Step 3- Compare the exponent of current node of list1 and list2.

$$4 < 5$$

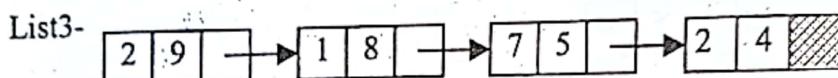
Hence add the node of list2 in list3.



Step 4- Compare the exponent of current node of list1 and list2.

$$\text{Both are equal i.e. } 4$$

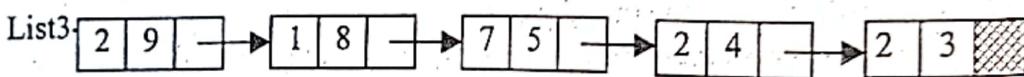
Hence add the coefficient of both nodes and add the result in list3.



Step 5- Compare the exponent of current node of list1 and list2.

$$2 < 3$$

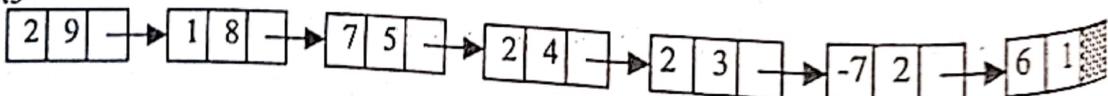
Hence add the node of list2 in list3.



Scan the next node of list2.

Step 6- List2 has no more nodes to scan. Hence add the remaining node of list1 to list3.

List3-



Implementation-

We declare a pointer variable p3_start which will denote the start address of the added polynomial list. We initialize this with NULL.

p3_start=NULL;

Pointer p3 will point to the current node in the added polynomial, and pointers p1 and p2 will point to the current nodes in the polynomials which will be added.

If both the polynomials are empty then the added polynomial will also be empty. So we return p3_start with NULL value, which denotes that the added polynomial list is empty.

```
if(p1==NULL && p2==NULL)
    return p3_start;
```

Now we traverse both polynomials until one polynomial finishes. First, we attach a new node to polynomial list3 and then we assign the coef and expo part of this node based following cases-

1. If $(p1->expo) > (p2->expo)$ then we assign the value of current node of p1 to current node of p3 and now p1 will point to the next node of polynomial1.
2. If $(p2->expo) > (p1->expo)$ then we assign the value of current node of p2 to current node of p3 and now p2 will point to the next node of polynomial.
3. If $(p1->expo) == (p2->expo)$ then we add the coefficients of p1 and p2 nodes and then new value will be assigned to current node of p3. Now p1 and p2 will point to next node of polynomials.

Here p1, p2 and p3 are pointing to the current node of polynomial1, polynomial2 and polynomial3. This can be written as-

```
while(p1!=NULL && p2!=NULL )
{
    tmp=malloc(sizeof(struct node));
    if(p3_start==NULL)
    {
        p3_start=tmp;
        p3=p3_start;
    }
    else
    {
        p3->link=tmp;
        p3=p3->link;
    }
    if(p1->expo > p2->expo)
    {
        tmp->coef=p1->coef;
        tmp->expo=p1->expo;
        p1=p1->link;
    }
    else
        if(p2->expo > p1->expo)
        {
            tmp->coef=p2->coef;
            tmp->expo=p2->expo;
            p2=p2->link;
        }
    else
        if(p1->expo == p2->expo)
        {
```

```

        tmp->coef=p1->coef + p2->coef;
        tmp->expo=p1->expo;
        p1=p1->link;
        p2=p2->link;
    }
}/*End of while*/

```

Here above loop will finish when traversal of any polynomial finishes. Now we have a need to traverse the remaining node of unfinished polynomial list and then it will be added to polynomial p3 as-

```

while(p1!=NULL)
{
    tmp=malloc(sizeof(struct node));
    tmp->coef=p1->coef;
    tmp->expo=p1->expo;
    if (p3_start==NULL) /*poly 2 is empty*/
    {
        p3_start=tmp;
        p3=p3_start;
    }
    else
    {
        p3->link=tmp;
        p3=p3->link;
    }
    p1=p1->link;
}/*End of while */

while(p2!=NULL)
{
    tmp=malloc(sizeof(struct node));
    tmp->coef=p2->coef;
    tmp->expo=p2->expo;
    if (p3_start==NULL) /*poly 1 is empty*/
    {
        p3_start=tmp;
        p3=p3_start;
    }
    else
    {
        p3->link=tmp;
        p3=p3->link;
    }
    p2=p2->link;
}/*End of while*/
p3->link=NULL;

```

Here at the end we assign the NULL value to the link part of p3 which is last node of p3.

```
/* Program of polynomial addition using linked list */
# include <stdio.h>
# include <malloc.h>

struct node
{
    float coef;
    int expo;
    struct node *link;
};

struct node *poly_add(struct node *,struct node *);
struct node *enter(struct node *);
struct node *insert(struct node *,float,int);

main()
{
    struct node *p1_start,*p2_start,*p3_start;

    p1_start=NULL;
    p2_start=NULL;
    p3_start=NULL;

    printf("Polynomial 1 :\n");
    p1_start=enter(p1_start);

    printf("Polynomial 2 :\n");
    p2_start=enter(p2_start);

    p3_start=poly_add(p1_start,p2_start);

    printf("Polynomial 1 is : ");
    display(p1_start);
    printf("Polynomial 2 is : ");
    display(p2_start);
    printf("Added polynomial is : ");
    display(p3_start);
}

/*End of main()*/

struct node *enter(struct node *start)
{
    int i,n,ex;
    float co;
    printf("How many terms u want to enter : ");
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
```

```

        printf("Enter coefficient for term %d : ",i);
        scanf("%f",&co);
        printf("Enter exponent for term %d : ",i);
        scanf("%d",&ex);
        start=insert(start,co,ex);
    }
    return start;
}/*End of enter()*/
struct node *insert(struct node *start,float co,int ex)
{
    struct node *ptr,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->coef=co;
    tmp->expo=ex;

    /*list empty or exp greater than first one */
    if(start==NULL || ex>start->expo)
    {
        tmp->link=start;
        start=tmp;
    }
    else
    {
        ptr=start;
        while(ptr->link!=NULL && ptr->link->expo>ex)
            ptr=ptr->link;
        tmp->link=ptr->link;
        ptr->link=tmp;
        if(ptr->link==NULL) /*item to be added in the end */
            tmp->link=NULL;
    }
    return start;
}/*End of insert()*/
struct node *poly_add(struct node *p1,struct node *p2)
{
    struct node *p3_start,*p3,*tmp;
    p3_start=NULL;
    if(p1==NULL && p2==NULL)
        return p3_start;
    while(p1!=NULL && p2!=NULL )
    {
        tmp=malloc(sizeof(struct node));
        if(p3_start==NULL)
        {
            p3_start=tmp;
            p3=p3_start;
        }

```

```

else
{
    p3->link=tmp;
    p3=p3->link;
}
if(p1->expo > p2->expo)
{
    tmp->coef=p1->coef;
    tmp->expo=p1->expo;
    p1=p1->link;
}
else
    if(p2->expo > p1->expo)
    {
        tmp->coef=p2->coef;
        tmp->expo=p2->expo;
        p2=p2->link;
    }
    else
        if(p1->expo == p2->expo)
        {
            tmp->coef=p1->coef + p2->coef;
            tmp->expo=p1->expo;
            p1=p1->link;
            p2=p2->link;
        }
/*End of while*/
while(p1!=NULL)
{
    tmp=malloc(sizeof(struct node));
    tmp->coef=p1->coef;
    tmp->expo=p1->expo;
    if(p3_start==NULL) /*poly 2 is empty*/
    {
        p3_start=tmp;
        p3=p3_start;
    }
    else
    {
        p3->link=tmp;
        p3=p3->link;
    }
    p1=p1->link;
}/*End of while */
while(p2!=NULL)
{
    tmp=malloc(sizeof(struct node));
    tmp->coef=p2->coef;
    tmp->expo=p2->expo;
}

```

```

        if(p3_start==NULL) /*poly 1 is empty*/
        {
            p3_start=tmp;
            p3=p3_start;
        }
        else
        {
            p3->link=tmp;
            p3=p3->link;
        }
        p2=p2->link;
    }/*End of while*/
    p3->link=NULL;
    return p3_start;
}/*End of poly_add() */

display(struct node *ptr)
{
    if(ptr==NULL)
    {
        printf("Empty\n");
        return;
    }
    while(ptr!=NULL)
    {
        printf("(%.1fx^%d) + ", ptr->coef,ptr->expo);
        ptr=ptr->link;
    }
    printf("\b\b \n"); /* \b\b to erase the last + sign */
}/*End of display()*/

```

Exercise-

1. Compare the advantages and disadvantages of using array list and linked list.
2. What is the advantage of using a circular linked list.
3. Write a program to concatenate one list at the end of other.
4. Write a program of splitting a list into two lists.
5. Write a program of copying one list into other.
6. Write a program to remove first node of the list and insert it at the end.
7. Write a program to remove the last node of the list and insert it in the beginning.

8. Write a program to swap the first and last elements of a linked list.
 (i) by exchanging info part.
 (ii) through pointers.
9. Write a program to count the number of occurrences of an element in the list.
10. Write a program to swap mth and nth elements of a linked list.
11. Write a program to swap mth and m+1th elements of a list
 (i) by exchanging info part
 (ii) thru pointers.
12. Count all non zero elements, odd numbers and even numbers in a list. ✓
- ✓ 13. Find the largest and smallest element of a list, print total of all elements and find out the average.
14. Given a list L1 , delete all the nodes having even number in info part from the list L1 and insert into list L2 and all the nodes having odd numbers into list L3.
15. Construct a linked list in which each node has the following information about a student rollno, name, marks in 3 subjects. Enter records of different students in list. Traverse this list and calculate the total marks, percentage and division of each student. Count how many students have 1st, 2nd, 3rd divisions and how many are fail.
16. Modify the above program so that now the names are inserted in alphabetical order in the list. Make this program menu driven with the following menu-
 - (1) Create list
 - (2) Insert
 - (3) Delete
 - (4) Modify
 - (5) Display record
 - (6) Display result

Delete menu should have the facility of entering name of a student and the record of that student should be deleted
 Display record menu should ask for the roll no of a student and display all information about him.
 Display result should display how many students have got 1,2,3 div and how many are fail. Modify menu has the facility of modifying a record.
- ✓ 17. Given two sorted linked lists, merge them into a third sorted linked list. If an element is present in both the lists ,it should occur only once in the third list.
18. Write a program to multiply the polynomials with a given number.

19. Write a program to multiply two polynomials.

20. Create a linked list with given number in which info part of each node contains the digit of this number. Suppose the number is 54681 then the nodes of linked list should contain 5,4,6,8,1

21. Write a program to delete the Nth node.

22. Write a program to delete all the nodes which have value N

23. Write a program of linked list in which element can be inserted only at the end of the list and deletion can also take place only at the end.

24. Write a program of linked list in which new element can be inserted only at the end and deletion can take place only in the beginning(Queue).

25. Write a program of doubly circular linked list.

26. Write a program to maintain a sorted linked list in descending order.

27. Write a program of doubly linked list to maintain records of students. Search a particular record based on roll number and display the previous and next node of that node.

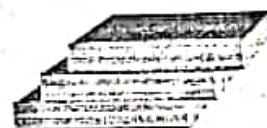
Chapter 4

Stack and Queue

It may be possible that there is a need of a data structure which takes operation on only one end i.e. beginning or end of the list. In linked list and arrays we can take operations on any place of the list. Stack and Queue are data structures which fulfill the requirements to take operation on only at one end.

Stack

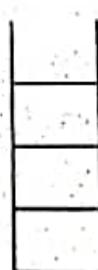
We can see the example of stack in our daily life as stack of CD's, stack of books.



Stack is defined as a list of elements in which we can insert or delete the element only at the top of the stack. You can see that we can take CD or book only from the top. Similarly we can keep one more CD or book only at the top.

There are two operations possible on the stack.

1. Push, when we add the item in stack
2. Pop, when we delete the item from stack.



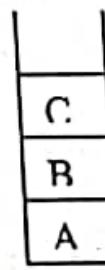
(a) Empty



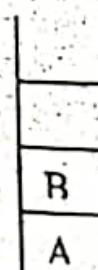
(b) Push A



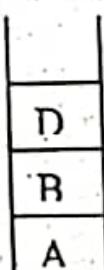
(c) Push B



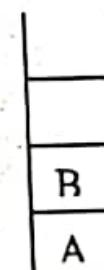
(d) Push C



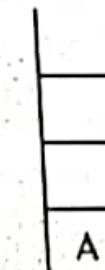
(e) Pop C



(f) Push D



(g) Pop D

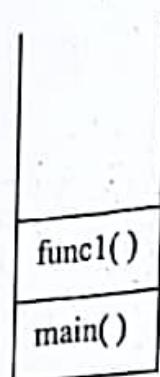
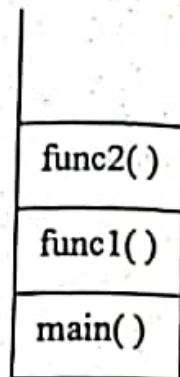
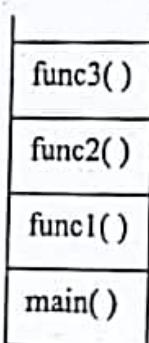
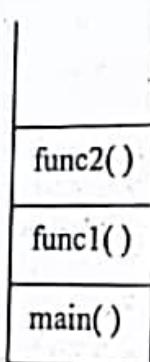
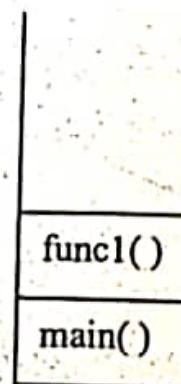
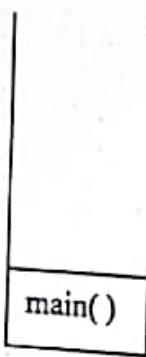


(h) Pop B

In computer implementation, let us take a C program example in which main() function calls func1(), func1() calls func2() and func2() calls func3(). Then it will be implemented in stack as-



(a) Push main()

(b) main() calls func1()
Push func1()(c) func1() calls func2()
Push func2()(d) func2() calls func3()
Push func3()(e) func3() completed
Pop func3()(f) func2() completed
Pop func2()(g) func1() completed
Pop func1()(h) main() completed
Pop main()

Here there will be some other values in stack related with function call in between two function call values but we are taking function call address only in consideration.

Array Implementation of Stack-

Since stack is a collection of same type of elements, so we can take array for implementing stack. In array we can push elements one by one from 0th position, 1st position.....n-1th position.

But in array, any element can be added or deleted at any place and we want to push or pop the element from top of the stack only. So we take a variable **top**, which keeps the position of the top element in array.

It may be possible that a condition arises when there is no place for adding(push) the element in the array. This is called overflow, so first we check the value of top with size of array.

The second possibility arises when there is no element for deleting(pop) from stack. If there is no element in stack then value of top will be -1. So we check the value of top before deleting the element of stack.

top=2

	10	15				
stackarr	[1]	[2]	[3]	[4]	[5]	[6]

Here stack is implemented with array stackarr, size of stackarr is 7. The value of top is 2. So the last element is added on the 2nd position in the array, the elements of stack are stackarr[0]=5, stackarr[1]=10 and stackarr[2]=15.

Push operation on stack-

For pushing the element on stack first we check the condition of overflow then push the element as –

```

if(top == (MAX-1))
    printf("Stack Overflow\n");
else
{
    top=top+1;
    stack_arr[top] = pushed_item;
}

```

Here top always points to last added item of stack.

Pop operation on stack-

For pop operation on stack first we check the condition of underflow then pop the element as –

```

if(top == -1)
    printf("Stack Underflow\n");
else
{
    printf("Popped element is : %d\n", stack_arr[top]);
    top=top-1;
}

```

Here top points to last pushed item of stack. After pop operation now it will point previous pushed item of stack.

/* Program of stack using array*/

```

#include<stdio.h>
#define MAX 5

```

```

int top = -1;
int stack_arr[MAX];

```

```
main()
```

```
{
```

```
    int choice;
```

```
    while(1)
```

```
{

```

```
    printf("1.Push\n");

```

```
    printf("2.Pop\n");

```

```
    printf("3.Display\n");

```

```
    printf("4.Quit\n");

```

```
    printf("Enter your choice : ");

```

```
    scanf("%d",&choice);

```

```
    switch(choice)
    {

```

```
        case 1 :

```

```
            push();

```

```
            break;

```

```
        case 2:

```

```
            pop();

```

```
            break;

```

```
        case 3:

```

```
            display();

```

```
            break;

```

```
        case 4:

```

```
            exit(1);

```

```
        default:

```

```
            printf("Wrong choice\n");
        }
    }
}
```

```
/*End of main()*/
/*End of while*/
/*End of switch*/

```

```

push()
{
    int pushed_item;
    if(top == (MAX-1))
        printf("Stack Overflow\n");
    else
    {
        printf("Enter the item to be pushed in stack : ");
        scanf("%d",&pushed_item);
        top=top+1;
        stack_arr[top] = pushed_item;
    }
/*End of push()*/
}

pop()
{
    if(top == -1)
        printf("Stack Underflow\n");
    else
    {
        printf("Popped element is : %d\n",stack_arr[top]);
        top=top-1;
    }
/*End of pop()*/
}

display()
{
    int i;
    if(top == -1)
        printf("Stack is empty\n");
    else
    {
        printf("Stack elements :\n");
        for(i = top; i >=0; i--)
            printf("%d\n", stack_arr[i] );
    }
/*End of display()*/
}

```

Linked List Implementation

Stack can be implemented through linked list also. For this we will take the structure as -

```

struct node{
    int info;
    struct node *link;
}

```

Push operation on Stack

For pushing the element on stack we follow the insertion operation of the linked list, means we add the element at the start of the list.

```
tmp->info = pushed_item;
tmp->link = top;
top = tmp;
```

Here top always points to the first node of linked list. After statement 3, last pushed will become first node of linked list. Since stack implementation is through linked list we don't check overflow condition.

Pop operation on Stack

For pop operation on stack we delete the first element of linked list because stack is in first out structure.

```
if(top == NULL)
    printf("Stack is empty\n");
else
{
    tmp = top;
    printf("Popped item is %d\n",tmp->info);
    top = top->link;
    free(tmp);
}
```

Here first we check for stack empty condition then we pop the first element of stack. Now top will point to second node of linked list which will become first node of linked list.

```
/* Program of stack using linked list*/
#include<stdio.h>
#include<malloc.h>

struct node
{
    int info;
    struct node *link;
} *top=NULL;

main()
{
    int choice;
```

```

while(1)
{
    printf("1.Push\n");
    printf("2.Pop\n");
    printf("3.Display\n");
    printf("4.Quit\n");
    printf("Enter your choice : ");
    scanf("%d", &choice);

    switch(choice)
    {
        case 1:
            push();
            break;
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            exit(1);
        default :
            printf("Wrong choice\n");
    }/*End of switch */
}/*End of while */
/*End of main() */

```

```

push()
{
    struct node *tmp;
    int pushed_item;
    tmp = (struct node *)malloc(sizeof(struct node));
    printf("Input the new value to be pushed on the stack : ");
    scanf("%d",&pushed_item);
    tmp->info=pushed_item;
    tmp->link=top;
    top=tmp;
}/*End of push()*/

```

```

pop()
{
    struct node *tmp;
    if(top == NULL)
        printf("Stack is empty\n");
    else
    {
        tmp=top;
        printf("Popped item is %d\n",tmp->info);
        top=top->link;
    }
}

```

```

        free(tmp);
    }

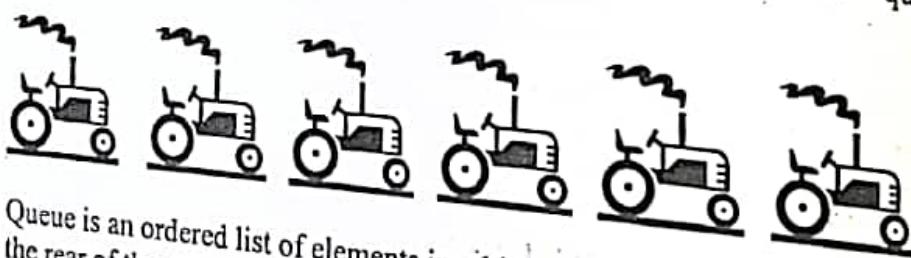
/*End of pop( )*/

display()
{
    struct node *ptr;
    ptr=top;
    if(top==NULL)
        printf("Stack is empty\n");
    else
    {
        printf("Stack elements :\n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->info);
            ptr = ptr->link;
        }
        /*End of while */
    }
    /*End of else*/
}
/*End of display( )*/

```

QUEUE

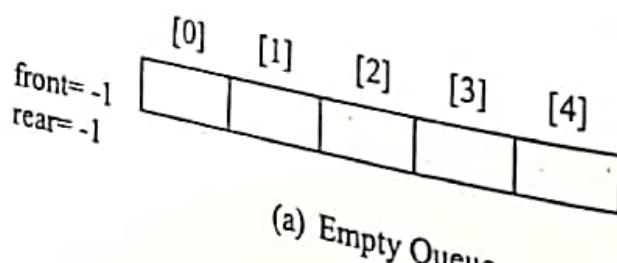
As the name implies we can see the example of queue in daily life as queue of people, queue of cars etc.



Queue is an ordered list of elements in which we can add elements only at one end, called the rear of the queue and delete elements only at the other end called the front of the queue.

We can see in the queue of people and queue of cars that the people or car that comes first in the queue will be out first. It's behaviour seems to be first in first out. So queue is also called FIFO data structure

queue_arr



```

        free(tmp);
    }

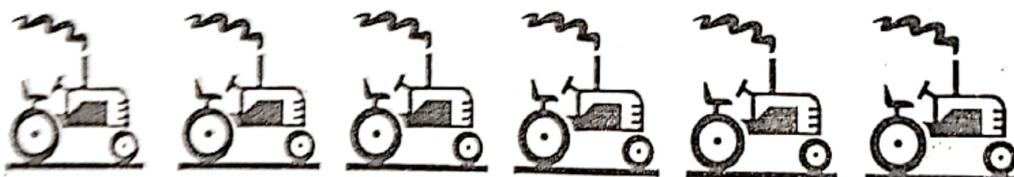
}/*End of pop()*/

display()
{
    struct node *ptr;
    ptr=top;
    if(top==NULL)
        printf("Stack is empty\n");
    else
    {
        printf("Stack elements :\n");
        while(ptr!=NULL)
        {
            printf("%d\n",ptr->info);
            ptr = ptr->link;
        }/*End of while */
    }/*End of else*/
}/*End of display()*/

```

QUEUE

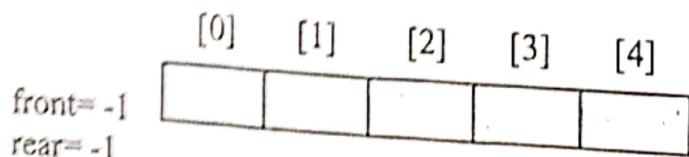
As the name implies we can see the example of queue in daily life as queue of people, queue of cars etc.



Queue is an ordered list of elements in which we can add elements only at one end, called the rear of the queue and delete elements only at the other end called the front of the queue.

We can see in the queue of people and queue of cars that the people or car that comes first in the queue will be out first. Its behaviour seems to be first in first out. So queue is also called FIFO data structure.

queue_arr



(a) Empty Queue

[0]	[1]	[2]	[3]	[4]
5				

front=0

rear=0

(b) Adding an element in queue

[0]	[1]	[2]	[3]	[4]
5	10			

front=0 rear=1

(c) Adding an element in queue

[0]	[1]	[2]	[3]	[4]
5	10	12		

front=0 rear=2

(d) Adding an element in queue

[0]	[1]	[2]	[3]	[4]
	10	12		

front=1 rear=2

(e) Deleting an element from queue

[0]	[1]	[2]	[3]	[4]
	10	12	16	

front=1 rear=3

(f) Adding an element in queue

In computer implementation, there is an example of accessing the printer in multi user environment. If printer is in process and more than one user wants to access the printer then it maintains the queue for requesting user and serves as first in first out manner, means gives the access permission to the user which comes first in the queue.

Array Implementation of Queue

As stack, Queue is also a collection of same type of elements. So we can take array for implementing the queue.

Since we want to add an item in queue at the rear end and delete the item in the queue at the front. We take two variables rear(keeps the status of last added item in queue) and front(keeps the status of first item of queue).

It may be possible that condition arises when there is no place for adding elements in queue. This is called overflow, so first we check the value of rear with size of array.

The second possibility arises when there is no element for deleting from queue. If there is no element in queue then value of front and rear will be -1 or front will be greater than rear. So we check the condition before deleting the element of queue.

queue_arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]
		5	10	15	20		

front=1 rear=4

Here queue is implemented with array queue_arr. Size of the queue is 7. The value of the front is 1 means element will be deleted from the 1st position of queue_arr. Value of the rear is 4 means the element will be added at the 5th position of queue_arr. The elements of the queue are -

queue_arr[1]=5, queue_arr[2]=10
queue_arr[3]=15, queue_arr[4]=20

Add operation in queue-

For adding the element in queue ,first we check the condition of overflow then add the element as -

```
4-1
if (rear == MAXSIZE-1)
    printf("Queue Overflow\n");
else
{
    if (front == -1) /*If queue is initially empty */
        front = 0;

    rear = rear+1;
    queue_arr[rear] = added_item;
}
```

Here initially front and rear are -1. After adding first element both will become 0 and after that only rear will increase when we add element in queue.

Delete operation in queue-

For deleting the element of queue ,first we check the condition of underflow then delete the element as -

```

if(front == -1 || front > rear)
{
    printf("Queue Underflow\n");
    return;
}
else
{
    printf("Element deleted from queue is: %d\n", queue_arr[front]);
    front = front + 1;
}

```

Here underflow condition can arise in both situations if front is -1 or front is greater than rear. For deleting the element, we increase the front value by 1.

It may be possible that a situation arises when rear is at the last position of array and front is not at the 0th position. But we cannot add any element in queue because rear is at the n-1th position.

<code>queue_arr</code>	[0]	[1]	[2]	[3]	[4]	[5]	[6]
		5	10	12	15	4	

`front=2` `rear=6`

There are 2 spaces for adding the elements in queue but we cannot add any element in queue because rear is at the last position of array.

One way is to shift all the elements of array to left and change the position of front and rear but it is not practically good approach.

To overcome this type of problem we use the concept of circular queue.

```
/*Program of queue using array*/  
#include<stdio.h>  
#define MAX 5
```

```
int queue_arr[MAX];  
int rear = -1;  
int front = -1;
```

1

```
int choice;  
while(1)  
{
```

```
printf("1.Insert\n");  
printf("2.Delete\n");
```

```

printf("3.Display\n");
printf("4.Quit\n");
printf("Enter your choice : ");
scanf("%d",&choice);

switch(choice)
{
    case 1:
        insert();
        break;
    case 2:
        delete();
        break;
    case 3:
        display();
        break;
    case 4:
        exit(1);
    default:
        printf("Wrong choice\n");
}
/*End of switch*/
}/*End of while*/
}/*End of main()*/\n

insert()
{
    int added_item;
    if (rear==MAX-1)
        printf("Queue Overflow\n");
    else
    {
        if (front==-1) /*If queue is initially empty */
            front=0;
        printf("Input the element for adding in queue : ");
        scanf("%d", &added_item);
        rear=rear+1;
        queue_arr[rear] = added_item;
    }
}/*End of insert()*/\n

del()
{
    if (front == -1 || front > rear)
    {
        printf("Queue Underflow\n");
        return;
    }
}

```

```

    else
    {
        printf("Element deleted from queue is : %d\n", queue_arr[front]);
        front=front+1;
    }
/*End of del() */

display()
{
    int i;
    if(front == -1)
        printf("Queue is empty\n");
    else
    {
        printf("Queue is :\n");
        for(i=front;i<= rear;i++)
            printf("%d ",queue_arr[i]);
        printf("\n");
    }
/*End of display() */
}

```

Linked List implementation

Queue can also be implemented through linked list. The structure of a node will be as-

```

struct node {
    int data;
    struct node *link;
}

```

Add operation in Queue

For adding the element in queue we add the element at the end of linked list. Here front

will point to the first node of linked list and rear will point to the last node of linked list.

```

tmp->info = added_item;
tmp->link = NULL;
if(front == NULL)           /*If Queue is empty*/
else    front = tmp;
rear->link = tmp;
rear = tmp;

```

Delete operation in Queue

For deleting the element of a queue ,we delete the first node of the linked list.Since queue is a first in first out structure and first node of the linked list will be the first element of the queue.

```
if(front == NULL)
    printf("Queue Underflow\n");
else
{
    tmp = front;
    printf("Deleted element is %d\n",tmp->info);
    front = front->link;
    free(tmp);
}
```

Here front is pointing to the first element of queue. After deleting the element it will point to the next element of queue.

```
/* Program of queue using linked list*/
#include<stdio.h>
#include<malloc.h>

struct node
{
    int info;
    struct node *link;
}*front=NULL,*rear=NULL;

main()
{
    int choice;
    while(1)
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
```

```

        del( );
        break;
    case 3:
        display( );
        break;
    case 4:
        exit(1);
    default :
        printf("Wrong choice\n");
    }/*End of switch*/
}/*End of while*/
}/*End of main()*/
}

insert()
{
    struct node *tmp;
    int added_item;
    tmp = (struct node *)malloc(sizeof(struct node));
    printf("Input the element for adding in queue : ");
    scanf("%d",&added_item);
    tmp->info = added_item;
    tmp->link=NULL;
    if(front==NULL)           /*If Queue is empty*/
        front=tmp;
    else
        rear->link=tmp;
    rear=tmp;
}/*End of insert()*/
}

del()
{
    struct node *tmp;
    if(front == NULL)
        printf("Queue Underflow\n");
    else
    {
        tmp=front;
        printf("Deleted element is %d\n",tmp->info);
        front=front->link;
        free(tmp);
    }
}/*End of del()*/
display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
        printf("Queue is empty\n");
}

```

```

else
{
    printf("Queue elements :\n");
    while(ptr != NULL)
    {
        printf("%d ",ptr->info);
        ptr = ptr->link;
    }
    printf("\n");
}/*End of else*/
}/*End of display( )*/

```

We can implement queue with circular linked list also, here we take only one variable rear.

```

/* Program of queue using circular linked list*/
#include <stdio.h>
#include <malloc.h>

struct node
{
    int info;
    struct node *link;
}*rear=NULL;

main( )
{
    int choice;
    while(1)
    {
        printf("1.Insert \n");
        printf("2.Delete \n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
                del();
                break;
            case 3:
                display();
                break;
        }
    }
}

```

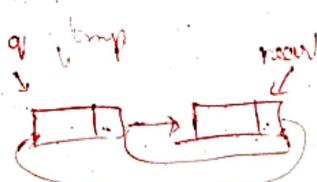
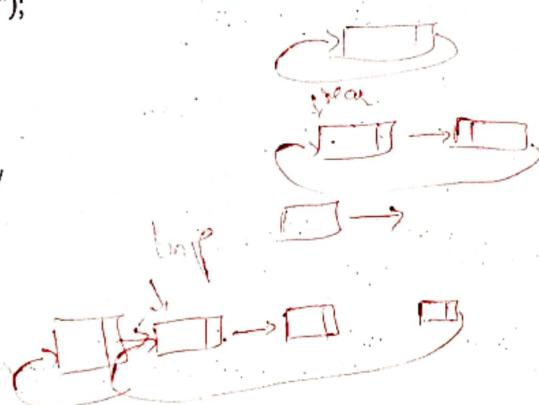
```

case 4:
    exit( );
default:
    printf("Wrong choice\n");
}/*End of switch*/
}/*End of while*/
}/*End of main()*/

insert()
{
    int num;
    struct node *q, *tmp;
    printf("Enter the element for insertion : ");
    scanf("%d",&num);
    tmp= malloc(sizeof(struct node));
    tmp->info = num;

    if(rear == NULL) /*If queue is empty */
    {
        rear = tmp;
        tmp->link = rear;
    }
    else
    {
        tmp->link = rear->link;
        rear->link = tmp;
        rear = tmp;
    }
}/*End of insert()*/
del()
{
    struct node *tmp, *q;
    if(rear==NULL)
    {
        printf("Queue underflow\n");
        return;
    }
    if( rear->link == rear ) /*If only one element*/
    {
        tmp = rear;
        rear = NULL;
        free(tmp);
    }
    q=rear->link;
    tmp=q;
    rear->link = q->link;
    printf("Deleted element is %d\n",tmp->info);
}

```



```

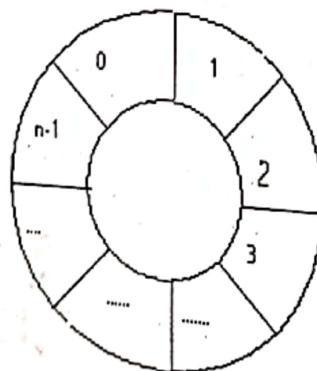
        free(tmp);
    }/*End of del( )*/

display()
{
    struct node *q;
    if(rear == NULL)
    {
        printf("Queue is empty\n");
        return;
    }
    q = rear->link;
    printf("Queue is :\n");
    while(q != rear)
    {
        printf("%d ", q->info);
        q = q->link;
    }
    printf("%d\n", rear->info);
}/*End of display()*/

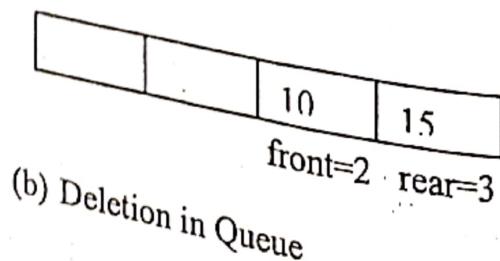
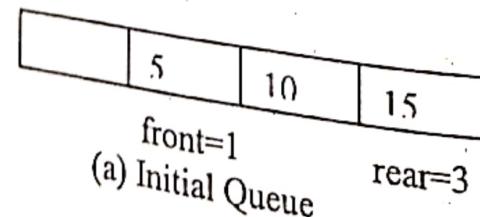
```

Circular Queue-

As in a circle after last element ,first element occurs.



Here after $n-1$ th element, 0th element occurs. Similarly we take assumption that after last element of queue, the first element will occur.



35		10	15
rear=0		front=2	

(c)Addition in Queue

35	20	10	15
rear=1	front=2		

(d)Addition in Queue

35	20		15
rear=1		front=3	

(e) Deletion in Queue

35	20		
front=0	rear=1		

(f) Deletion in Queue

	20		
front=1			
rear=1			

(g)Deletion in Queue

--	--	--	--

front= -1

rear= -1

(h)Deletion in Queue(Queue is empty)

Add operation in Circular Queue-

We check that rear is at the n-1th position .If rear =N-1 then we set the value of rear to 0 and add the element at the 0th position of the array otherwise element will be added same as in simple queue.

Here first we are checking for overflow condition.

```
if((front == 0 && rear == MAXSIZE-1) || (front == rear))
{
    printf("Queue Overflow \n");
    return;
}
```

If queue is initially empty then we set the value 0 to front and rear and then add the element in queue otherwise we increase the value of rear only and then element will be added in queue.

```
if(front == -1) /* If queue is initially empty */
{
    front = 0;
    rear = 0;
}
else
if(rear == MAXSIZE-1) /* rear is at last position of queue */
    rear = 0;
else
    rear = rear+1;
queue_arr[rear] = added_item;
```

Delete operation in Circular Queue

We check that front is at the N-1th position. If front=N-1 then delete the element from queue and set the value of front to 0 as-

```
if(front==MAXSIZE-1)
    front=0;
```

If there is only one element in queue then we delete the element from queue and set the value of front and rear to -1 as-

```
if(front == rear) /* queue has only one element */
{
    front = -1;
    rear = -1;
}
```

All the other conditions will be similar to the previous queue implementation.

```
* Program of circular queue using array*/
#include<stdio.h>
#define MAX 5

int queue_arr[MAX];
int front = -1;
int rear = -1;

main()
{
    int choice;
    while(1)
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1 :
                insert();
                break;
            case 2 :
                del();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(1);
            default:
                printf("Wrong choice\n");
        }
    }
}

/*End of main()*/
insert()
{
    int added_item;
    if((front == 0 && rear == MAX-1) || (front == rear+1))
    {
        printf("Queue Overflow\n");
        return;
    }
}
```

```

if(front == -1) /*If queue is empty */
{
    front = 0;
    rear = 0;
}
else
    if(rear == MAX-1)/*rear is at last position of queue */
        rear = 0;
    else
        rear = rear+1;
printf("Input the element for insertion in queue : ");
scanf("%d", &added_item);
cqueue_arr[rear] = added_item ;
}/*End of insert()*/
}

del()
{
    if(front == -1)
    {
        printf("Queue Underflow\n");
        return ;
    }
    printf("Element deleted from queue is : %d\n",cqueue_arr[front]);
    if(front == rear) /* queue has only one element */
    {
        front = -1;
        rear=-1;
    }
    else
        if(front == MAX-1)
            front = 0;
        else
            front = front+1;
}/*End of del() */

display()
{
    int front_pos = front,rear_pos = rear;
    if(front == -1)
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements :\n");
    if( front_pos <= rear_pos )
        while(front_pos <= rear_pos)
        {
            printf("%d ",cqueue_arr[front_pos]);
            front_pos++;
        }
}

```

```

else
{
    while(front_pos <= MAX-1)
    {
        printf("%d ",cqueue_arr[front_pos]);
        front_pos++;
    }
    front_pos = 0;
    while(front_pos <= rear_pos)
    {
        printf("%d ",cqueue_arr[front_pos]);
        front_pos++;
    }
}
/*End of else */
printf("\n");
/*End of display() */

```

Priority Queue

As we have seen earlier, queue is an ordered list of elements in which we can add the element only at one end called rear of the queue and delete the element only at the other end called front of the queue .But in priority queue every element of queue has some priority and based on that priority it will be processed. So the element of more priority will be processed before the element which has less priority.

Suppose two elements have same priority then in this case FIFO rule will follow, means the element which comes first in the queue will be processed first.

In computer implementation, priority queue is used in the CPU scheduling algorithm, in which CPU has need to process those processes first which have more priority.

Linked list implementation of priority queue

Priority queue can be maintained in other ways also but here we are taking linked list approach. First, we have a need to define the structure of elements which will be used for priority queue-

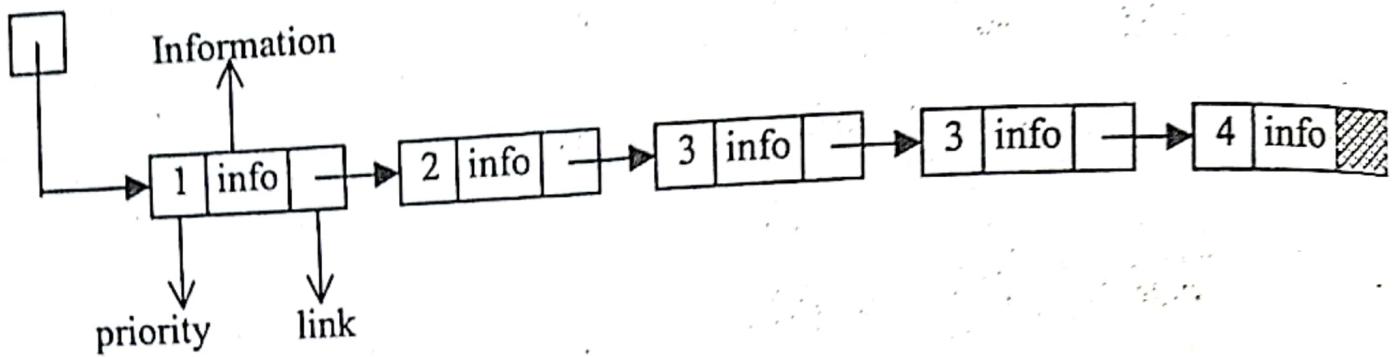
```

struct pq{
    int priority;
    int data;
    struct pq *link;
}

```

Here first member of structure is the priority for that element ,second has information and third is link which has address of next element.

start



Here priority 1 means the highest priority. If priority of an element is 2 then it means it has priority more than the element which has priority 3.

Operation in priority queue

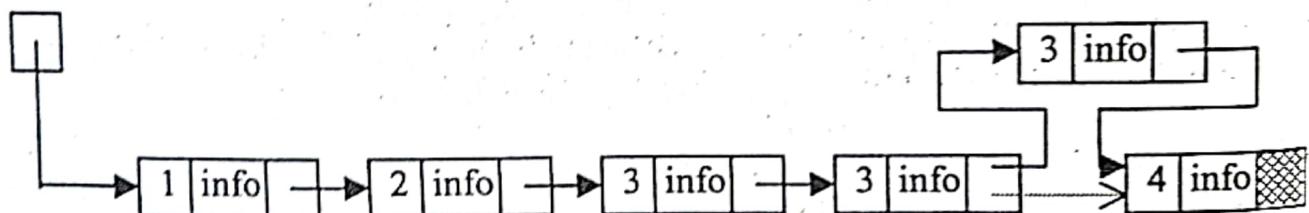
As in queue priority queue has also same two operations ,but in different manner.

1. Add
2. Delete

Add operation in Priority Queue

Add operation in priority queue is same as the insert operation in sorted linked list. Here we insert the new element on the basis of priority of element. The new element will be inserted before the element which has less priority than new element.

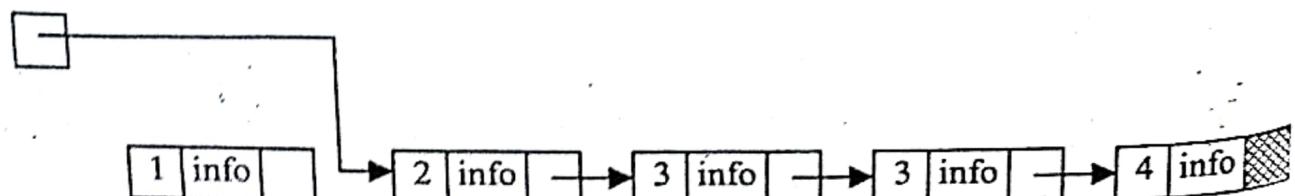
start



Delete operation in Priority Queue

Delete operation will be the deletion of first element of list because it has more priority than other elements of queue.

start



```

/* Program of priority queue using linked list*/
#include<stdio.h>
#include<malloc.h>
struct node
{
    int priority;
    int info;
    struct node *link;
}*front = NULL;

main()
{
    int choice;
    while(1)
    {
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Display\n");
        printf("4.Quit\n");
        printf("Enter your choice : ");
        scanf("%d", &choice);

        switch(choice)
        {
            case 1:
                insert();
                break;
            case 2:
                del();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(1);
            default :
                printf("Wrong choice\n");
        }/*End of switch*/
    }/*End of while*/
}/*End of main()*/
insert()
{
    struct node *tmp,*q;
    int added_item , item_priority;
    tmp = (struct node *)malloc(sizeof(struct node));
    printf("Input the item value to be added in the queue : ");
    scanf("%d", &added_item);
    printf("Enter its priority : ");
}

```

```

scanf("%d",&item_priority);
tmp->info = added_item;
tmp->priority = item_priority;
/*Queue is empty or item to be added has priority more than first item*/
if( front == NULL || item_priority < front->priority )
{
    tmp->link = front;
    front = tmp;
}
else
{
    q = front;
    while( q->link != NULL && q->link->priority <= item_priority )
        q=q->link;
    tmp->link = q->link;
    q->link = tmp;
}/*End of else*/
}/*End of insert()*/



del()
{
    struct node *tmp;
    if(front == NULL)
        printf("Queue Underflow\n");
    else
    {
        tmp = front;
        printf("Deleted item is %d\n",tmp->info);
        front = front->link;
        free(tmp);
    }
}/*End of del()*/



display()
{
    struct node *ptr;
    ptr = front;
    if(front == NULL)
        printf("Queue is empty\n");
    else
    {
        printf("Queue is :\n");
        printf("Priority   Item\n");
        while(ptr != NULL)
        {
            printf("%5d      %5d\n",ptr->priority,ptr->info);
            ptr = ptr->link;
        }
    }/*End of else */
}/*End of display() */

```

DEQUEUE

As we have seen earlier queue is an ordered list of elements in which we can add the element only at one end called the rear of queue and delete the element only at the other end called the front of the queue. But in dequeue, also called double ended queue, as the name implies we can add or delete the element from both sides. Dequeue can be of two types-

1. Input restricted
2. Output restricted

In Input restricted dequeue, element can be added at only one end but we can delete the element from both sides.

Similarly as in Output restricted dequeue, element can be added from both sides but deletion is allowed only at one end.

Array Implementation of Dequeue

Here we are taking array for implementing the dequeue. Similarly as queue it has also two operations.

1. Add
2. Delete

We maintain two pointers LEFT and RIGHT which indicate the left and right position of the dequeue.

dq_arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
			5	10	15			

LEFT - 2

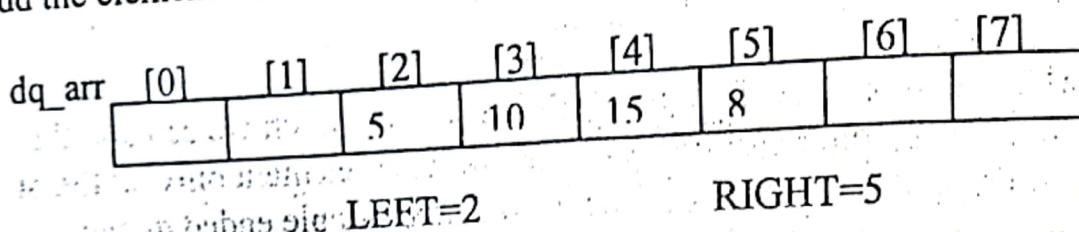
RIGHT - 4

Here left pointer is at position 2 and right pointer is at position 4 in the array

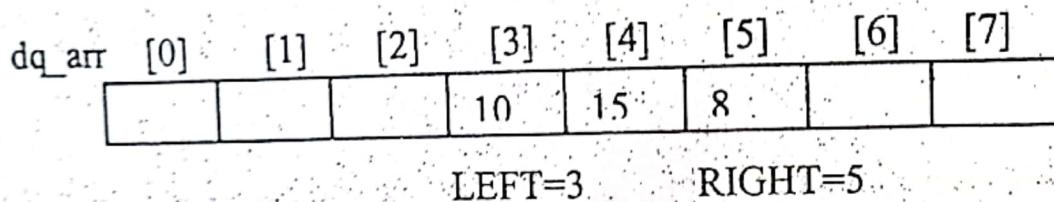
Add and delete operation in Dequeue

We assume this is circular array for operations of addition and deletion. Let us take it's an input restricted dequeue and we can add element only on the right side of queue but we can delete from both sides.

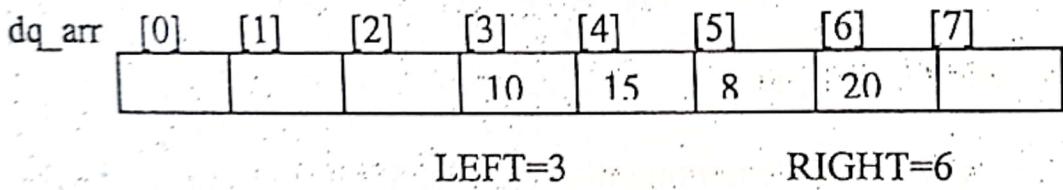
1. Add the element 8 in the queue



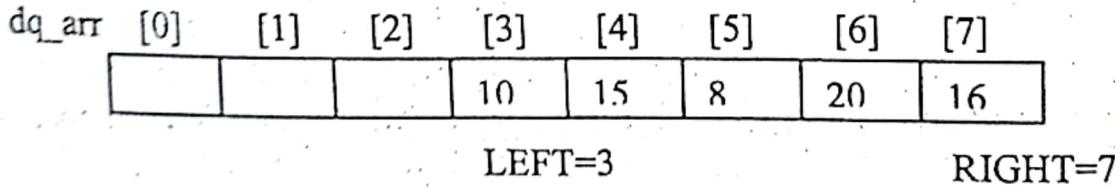
2. Delete the element from left of the queue



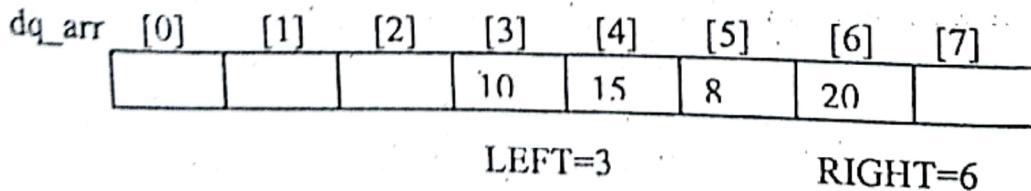
3. Add the element 20 in the queue



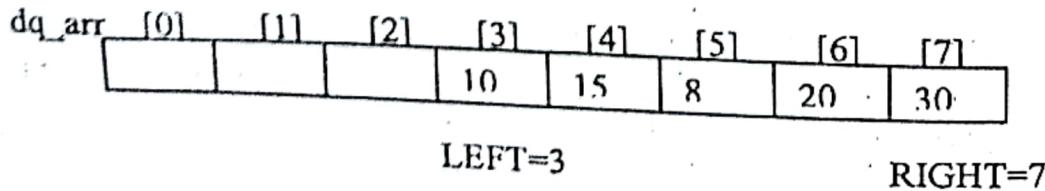
4. Add the element 16 in the queue



5. Delete the element from right of the queue



6. Add the element 30 in the queue



7. Add the element 12 in the queue

dq_arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	12			10	15	8	20	30

RIGHT=0 LEFT=3

8. Add the element 35 in the queue

dq_arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	12	35		10	15	8	20	30

RIGHT=1 LEFT=3

9. Add the element 6 in the queue

dq_arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
	12	35	6	10	15	8	20	30

RIGHT=2 LEFT=3

10. Add the element 45 in the queue

Now it's overflow because RIGHT pointer will become equal to LEFT pointer after

adding one element.

Suppose operation will be on output restricted dequeue , then element can be added from both sides but deletion is allowed only at one end.

/* Program of input and output restricted dequeue using array*/

include<stdio.h>

define MAX 5

```
int deque_arr[MAX];
int left = -1;
int right = -1;
```

```
main()
{
```

```
    int choice;
    printf("1.Input restricted dequeue\n");
    printf("2.Output restricted dequeue\n");
    printf("Enter your choice : ");
    scanf("%d",&choice);
```

```
    switch(choice)
    {
```

```
        case 1 :
```

```
            input_que();
```

```

        break;
case 2:
    output_que();
    break;
default:
    printf("Wrong choice\n");
}/*End of switch*/
}/*End of main()*/



input_que()
{
    int choice;
    while(1)
    {
        printf("1.Insert at right\n");
        printf("2.Delete from left\n");
        printf("3.Delete from right\n");
        printf("4.Display\n");
        printf("5.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                insert_right();
                break;
            case 2:
                delete_left();
                break;
            case 3:
                delete_right();
                break;
            case 4:
                display_queue();
                break;
            case 5:
                exit(1);
            default:
                printf("Wrong choice\n");
        }/*End of switch*/
    }/*End of while*/
}/*End of input_que() */



output_que()
{
    int choice;

```

```

while(1)
{
    printf("1.Insert at right\n");
    printf("2.Insert at left\n");
    printf("3.Delete from left\n");
    printf("4.Display\n");
    printf("5.Quit\n");
    printf("Enter your choice : ");
    scanf("%d",&choice);

    switch(choice)
    {
        case 1:
            insert_right();
            break;
        case 2:
            insert_left();
            break;
        case 3:
            delete_left();
            break;
        case 4:
            display_queue();
            break;
        case 5:
            exit();
        default:
            printf("Wrong choice\n");
    }/*End of switch*/
}/*End of while*/
}/*End of output_que() */

insert_right()
{
    int added_item;
    if((left == 0 && right == MAX-1) || (left == right+1))
    {
        printf("Queue Overflow\n");
        return;
    }
    if(left == -1) /* if queue is initially empty */
    {
        left = 0;
        right = 0;
    }
    else
        if(right == MAX-1) /*right is at last position of queue */
            right = 0;
}

```

```

else
    right = right+1;
printf("Input the element for adding in queue : ");
scanf("%d", &added_item);
deque_arr[right] = added_item ;
/*End of insert_right( )*/

insert_left( )
{
    int added_item;
    if((left == 0 && right == MAX-1) || (left == right+1))
    {
        printf("Queue Overflow \n");
        return;
    }
    if(left == -1)/*If queue is initially empty*/
    {
        left = 0;
        right = 0;
    }
    else
    if(left== 0)
        left=MAX-1;
    else
        left=left-1;
    printf("Input the element for adding in queue : ");
    scanf("%d", &added_item);
    deque_arr[left] = added_item ;
/*End of insert_left( )*/

delete_left( )
{
    if(left == -1)
    {
        prints("Queue Underflow\n");
        return ;
    }
    printf("Element deleted from queue is : %d\n",deque_arr[left]);
    if(left == right) /*Queue has only one element */
    {
        left = -1;
        right=-1;
    }
    else
        if(left == MAX-1)
            left = 0;
}

```

```

        else
            left = left+1;
    }/*End of delete_left()*/
}

delete_right()
{
    if(left == -1)
    {
        printf("Queue Underflow\n");
        return ;
    }
    printf("Element deleted from queue is : %d\n",deque_arr[right]);
    if(left == right) /*queue has only one element*/
    {
        left = -1;
        right=-1;
    }
    else
        if(right == 0)
            right=MAX-1;
        else
            right=right-1;
}/*End of delete_right() */

display_queue()
{
    int front_pos = left,rear_pos = right;
    if(left == -1)
    {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements :\n");
    if( front_pos <= rear_pos )
    {
        while(front_pos <= rear_pos)
        {
            printf("%d ",deque_arr[front_pos]);
            front_pos++;
        }
    }
    else
    {
        while(front_pos <= MAX-1)
        {
            printf("%d ",deque_arr[front_pos]);
            front_pos++;
        }
    }
}

```

```

    front_pos = 0;
    while(front_pos <= rear_pos)
    {
        printf("%d ", deque_arr[front_pos]);
        front_pos++;
    }
}/*End of else */
printf("\n");
}/*End of display_queue() */

```

Applications of stack

Some of the uses of stack are described below-

1. Reversal of a string.
2. Checking validity of an expression containing nested parentheses
3. Conversion of infix expression to postfix and prefix forms.
4. Evaluation of postfix and prefix forms.

Reversal of string

We can reverse a string by pushing each character of the string on the stack .When the whole string is pushed on the stack we will pop the characters from the stack and we will get the reversed string.

```

/* Program of reversing a string using stack */
#include<stdio.h>
#define MAX 20
#include<string.h>

int top = -1;
char stack[MAX];
char pop();
push(char);
main()
{
    char str[20];
    int i;
    printf("Enter the string : ");
    gets(str);

    /*Push characters of the string str on the stack */
    for(i=0;i<strlen(str);i++)
        push(str[i]);
}

```

```

/*Pop characters from the stack and store in string str */
for(i=0;i<strlen(str);i++)
    str[i]=pop();
printf("Reversed string is : ");
puts(str);
}/*End of main()*/
push(char item)
{
    if(top == (MAX-1))
        printf("Stack Overflow\n");
    else
        stack[++top] =item;
}/*End of push()*/
char pop()
{
    if(top == -1)
        printf("Stack Underflow\n");
    else
        return stack[top--];
}/*End of pop()*/

```

Checking validity of an expression containing nested parentheses

We can use stack to check the validity of an expression which uses nested parentheses.

An expression will be valid if it satisfies these two conditions-

1. The total number of left parentheses should be equal to the total number of right parentheses in the expression.
2. For every right parenthesis there should be a left parenthesis of the same type.

Some valid and invalid mathematical expressions are given below:

[19-5*(9+4)	Invalid
(1+5}	Invalid
[5+4*(9-2)]	Valid
{ 3+2-[9%2]]	Invalid
[2+(4*2)-{6%4}]	Valid

The procedure is as-

Take a boolean variable valid which will be true if the expression is valid and will be false if the expression is invalid. Initially let valid is true.

1. Scan the symbols of expression from left to right.
2. If the symbol is a left parenthesis then push it on the stack.

3. If the symbol is right parenthesis

If the stack is empty

valid =false

else

pop an element from stack

If popped parenthesis does not match the parenthesis being scanned

valid=false

4. After scanning all the symbols if stack is not empty then make valid =false

/* Program to check nesting of parentheses using stack */

#include<stdio.h>

#define MAX 20

#define true 1

#define false 0

int top = -1;

int stack[MAX];

push(char);

char pop();

main()

{

char exp[MAX],temp;

int i,valid=true;

printf("Enter an algebraic expression : ");

gets(exp);

for(i=0;i<strlen(exp);i++)

{

if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')

push(exp[i]);

if(exp[i]==')' || exp[i]=='}' || exp[i]==']')

if(top == -1) /* stack empty */

valid=false;

else

{

temp=pop();

if(exp[i]==')' && (temp=='{' || temp=='['))

valid=false;

if(exp[i]=='}' && (temp=='(' || temp=='['))

valid=false;

if(exp[i]==']' && (temp=='(' || temp=='{'))

valid=false;

}/*End of else */

}/*End of for*/

```

if(top>=0) /*stack not empty*/
    valid=false;
if( valid==true )
    printf("Valid expression\n");
else
    printf("Invalid expression\n");
}/*End of main()*/



push(char item)
{
    if(top == (MAX-1))
        printf("Stack Overflow\n");
    else
    {
        top=top+1;
        stack[top] = item;
    }
}

}/*End of push()*/



char pop()
{
    if(top == -1)
        printf("Stack Underflow\n");
    else
        return(stack[top--]);
}/*End of pop()*/

```

Polish Notation with arithmetic expression-

In any language, arithmetic expression support is needed which is same as mathematical expression. But writing data structure for understanding arithmetic expression have so many complications, like scanning direction, operators priority and after that how to decide the priority if any parenthesis is coming. So designing compiler for any language has a need to take care of these complications. But support of arithmetic expression is one of basic requirement of any language. Let us take an arithmetic expression-

$$a + b/3$$

Suppose $a=9$ and $b=6$. Suppose we are taking divide operation first then add. So the resultant value will be-

$$\begin{aligned}
 & 9+6/3 \\
 & = 9+2 \\
 & = 11
 \end{aligned}$$

Now, suppose we are taking add operation first then divide operation. So the resultant value will be-

$$\begin{aligned}
 9+6/3 \\
 =15/3 \\
 =5
 \end{aligned}$$

Here we can see that priority of operation is giving different result. So it is decided that scanning will be from left to right and each operator will have some precedence level. Now suppose we are taking parenthesis in expression. Let us take the same expression with parenthesis.

$$(a+b)/3$$

Here first the operation will be for expression inside the parenthesis. So we have a need to decide the priority for parenthesis. First expressions inside the parenthesis should be evaluated so its priority should be highest.

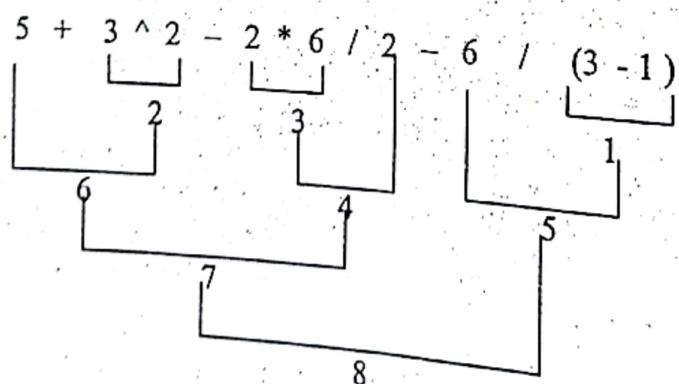
Here we will take five operators '+', '-', '*', '/' and '^' (for exponentiation) and based on that all the expression will be given.

Level 2	\wedge (Exponentiation)
Level 1	$*$ (Multiplication) and $/$ (Division)
Level 0	$+$ (Addition) and $-$ (Subtraction)

Here Level 2 is highest priority and Level 0 is the lowest. We can see '*' and '/' have same precedence level. We are scanning from left to right, so whichever operator in both will come first that will be evaluated first.

Let us take an arithmetic expression-

$$5 + 3^2 - 2 * 6 / 2 - 6 / (3 - 1)$$



Here we can see that the expression inside the parenthesis is evaluated first and after that evaluation is on the basis of operator precedence. We can see each time we have a need to know the information of parenthesis and precedence of operators for evaluating expression and every time we are traversing to different place in expression for evaluation. So now we have a need of the process of evaluation where parenthesis should not come and evaluation should be based on scanning the expression from left to right and the operator which is coming first should be evaluated with its operands.

Polish Notation-

The great polish mathematician Jan Lukasiewich came with new technique for representation of arithmetic expression where operator will be before or after operands called polish notation in his honour.

Now we have three way to represent the arithmetic expression-

Infix-	$A+B$
Prefix-	$+AB$
Postfix-	$AB+$

First one where operator is in between operands is called infix, second one where operator is before operands called prefix or polish notation and third one where operator is after operands is called postfix(suffix) or reverse polish notation. Here we have no need of parenthesis, which will increase the efficiency in evaluation of expression. Let us take some infix expressions and convert them in prefix and postfix expression-

Infix-	$(A + B) / C * D - E$
Prefix-	$\{ +AB \} / C * D - E$
	$\{ / +ABC \} * D - E$
	$\{ * / +ABCD \ } - E$
	$- * / +ABCDE$
Postfix-	$\{ AB+ \} / C * D - E$
	$\{ AB + C / \} * D - E$
	$\{ AB + C / D * \} - E$
	$AB + C / D * E -$

Let us take another expression-

Infix-	$A + B / C - D * E + F$
Prefix-	$A + \{ / BC \} - \{ * DE \} + F$
	$\{ + A / BC \} - \{ + * DEF \}$
	$- + A / BC + * DEF$
Postfix-	$A + \{ BC / \} - \{ DE * \} + F$
	$\{ ABC / + \} - \{ DE * F + \}$
	$ABC / + DE * F + -$

Here we can see the prefix and postfix expression are not a mirror image of each other. In polish notation we don't have a need of any parenthesis and by scanning from left to right we can evaluate the expression one by one. There is no need of traversal in expression at different places for evaluation. So computer takes the input in infix form and convert them in postfix form then it evaluates expression efficiently.

Converting infix expression into postfix expression-

Before converting the infix expression into postfix expression we have a need to take some assumption. Here we are scanning from left to right and operators involved are only '^' (exponentiation), '+', '-', '*' and '/'. We will be in need to implement stack for temporary placement of operators and information of operator is also required. For getting the information of end of arithmetic expression, addition of one unique symbol is needed at the end.

Let us take an array infix has arithmetic expression in infix form and array postfix will contain the arithmetic expression in postfix form. The steps involved to convert the infix expression into postfix expression will be-

Step1-

Add the unique symbol '#' into stack and at the end of array infix.

Step2-

Scan the symbol of array infix one by one from left to right.

Step3-

If symbol is left parenthesis '(' then add it to the stack.

Step4-

If symbol is operand then add it to array postfix.

Step5-

(i) If symbol is operator then pop the operators which have same precedence or higher precedence than the operator which occurred.

(ii) Add the popped operator to array postfix.

(iii) Add the scanned symbol operator into stack.

Step6-

(i) If symbol is right parenthesis ')' then pop all the operators from stack until left parenthesis '(' in stack.

(ii) Remove left parenthesis '(' from stack.

Step7-

If symbol is '#' then pop all the symbols from stack and add them to array postfix except '#'.

Step8-

Do the same process until '#' comes in scanning array infix.

Let us take an infix expression and convert it into postfix-

A * (B + C ^ D) - E ^ F * (G / H)

Initially '#' will be added in stack and at the end of infix expression.
So now the infix expression will be-

A * (B + C ^ D) - (E ^ F) * (G / H) #

Step	Symbol	Operator in stack	Postfix expression
1.	A	#	A
2.	*	# *	A
3.	(# * (A
4.	B	# * (AB
5.	+	# * (+	AB
6.	C	# * (+	ABC
7.	^	# * (+ ^	ABC
8.	D	# * (+ ^	ABCD
9.)	# *)	ABCD^+
10.	-	# -	ABCD^+*
11.	E	# -	ABCD^+* E
12.	^	# - ^	ABCD^+* E
13.	F	# - ^	ABCD^+* EF
14.	*	# - ^	ABCD^+* EF^
15.	(# - ^ (ABCD^+* EF^
16.	G	# - ^ (ABCD^+* EF^ G
17.	/	# - ^ (/	ABCD^+* EF^ G
18.	H	# - ^ (/	ABCD^+* EF^ GH
19.)	# - ^	ABCD^+* EF^ GH
20.	#		ABCD^+* EF^ GH/* -

So now the postfix expression is-

ABCD^+* EF^ GH/* -

We can see that the resultant postfix expression is parenthesis free and operators are in order of sequence of evaluation with operands.

Evaluation of postfix expression-

Evaluation of postfix expression also maintains the stack but here stack contains the operands instead of operators. Whenever any operator occurs in scanning, we evaluate with last two elements of stack. Initially one unique symbol will be added at the end of arithmetic expression for termination of scanning. Here we have no need of information of operator precedence.

Let us take an array infix has arithmetic expression in postfix form. The steps involved to evaluate the postfix expression will be-

Step1-

Add the unique symbol '#' at the end of array postfix.

Step2-

Scan the symbol of array postfix one by one from left to right.

Step3-

If symbol is operand then push it to stack.

Step4-

If symbol is operator then pop last two element of stack and evaluate it as
[top-1] operator [top]

and push it to stack.

Step5-

Do the same process until '#' comes in scanning.

Step6-

Pop the element of stack which will be value of evaluation of postfix arithmetic expression.

Let us take a postfix expression and evaluate it-

4, 5, 4, 2, ^, +, *, 2, 2, ^, 9, 3, /, *, -

Initially '#' will be added at the end of postfix expression, so now the postfix expression will be-

4, 5, 4, 2, ^, +, *, 2, 2, ^, 9, 3, /, *, -

Step	Symbol	Operator in stack
1.	4	4
2.	5	4, 5
3.	4	4, 5, 4
4.	2	4, 5, 4, 2
5.	^	4, 5, 16
6.	+	4, 21
7.	*	84
8.	2	84, 2
9.	2	84, 2, 2
10.	^	84, 4
11.	9	84, 4, 9
12.	3	84, 4, 9, 3
13.	/	84, 4, 3
14.	*	84, 12
15.	-	72
16.	#	

So after evaluation of the postfix expression it's value is 72. Let us take the same postfix expression in infix form and evaluate it.

$$\begin{aligned}
 & 4 * (5 + 4^2) - 2^2 * (9 / 3) \\
 & = 4 * (5 + 16) - 4 * 3 \\
 & = 4 * 21 - 12 \\
 & = 84 - 12 \\
 & = 72
 \end{aligned}$$

We can see the same result but in postfix evaluation we have no need to take care of parenthesis and sequence of evaluation.

/* Program for conversion of infix to postfix and evaluation of postfix.
It will take only single digit in expression */

```

#include<stdio.h>
#include<string.h>
#include<math.h>
#define Blank ''
#define Tab '\t'
#define MAX 50

long int pop();
long int eval_post();
char infix[MAX], postfix[MAX];
long int stack[MAX];
int top;

main()
{
    long int value;
    char choice='y';
    while(choice == 'y')
    {
        top = 0;
        printf("Enter infix : ");
        fflush(stdin);
        gets(infix);
        infix_to_postfix();
        printf("Postfix : %s\n",postfix);
        value=eval_post();
        printf("Value of expression : %ld\n",value);
        printf("Want to continue(y/n) : ");
        scanf("%c",&choice);
    }
    /*End of main()*/
}

infix_to_postfix()
{
    int i,p=0,type,precedence,len;
    char next;

```

```

stack[top] = '#';
len = strlen(infix);
infix[len] = '#';

for(i=0; infix[i] != '#'; i++)
{
    if( !white_space(infix[i]))
    {
        switch(infix[i])
        {

            case '(':
                push(infix[i]);
                break;

            case ')':
                while( (next = pop()) != '(')
                    postfix[p++] = next;
                break;

            case '+':
            case '-':
            case '*':
            case '/':
            case '%':
            case '^':
                precedence = prec(infix[i]);
                while(stack[top] != '#' && precedence <= prec(stack[top]))
                    postfix[p++] = pop();
                push(infix[i]);
                break;

            default: /*if an operand comes */
                postfix[p++] = infix[i];
        }/*End of switch */

    }/*End of if */

}/*End of for */
while(stack[top] != '#') /* Pop remaining operators */
    postfix[p++] = pop();
postfix[p] = '\0'; /*Append '\0' to postfix to make it a string*/
}/*End of infix_to_postfix()*/
/* This function returns the precedence of the operator */
prec(char symbol )
{
    switch(symbol)
    {
        case '(':
            return 0;
        case '+':

```

```
case '-':  
    return 1;  
case '*':  
case '/':  
case '%':  
    return 2;  
case '^':  
    return 3;  
}/*End of switch*/  
}/*End of prec( )*/  
  
push(long int symbol)  
{  
    if(top > MAX)  
    {  
        printf("Stack overflow\n");  
        exit(1);  
    }  
    else  
    {  
        top=top+1;  
        stack[top] = symbol;  
    }  
}/*End of push()*/  
  
long int pop()  
{  
    if (top == -1 )  
    {  
        printf("Stack underflow \n");  
        exit(2);  
    }  
    else  
        return (stack[top--]);  
}/*End of pop()*/  
  
white_space(char symbol)  
{  
    if( symbol == Blank || symbol == Tab || symbol == '\0')  
        return 1;  
    else  
        return 0;  
}/*End of white_space()*/  
  
long int eval_post()  
{  
    long int a,b,temp,result,len;  
    int i;  
    len=strlen(postfix);
```

```

postfix[len]='#';
for(i=0;postfix[i]!='#';i++)
{
    if(postfix[i]<='9' && postfix[i]>='0')
        push( postfix[i]-48 );
    else
    {
        a=pop();
        b=pop();

        switch(postfix[i])
        {
            case '+':
                temp=b+a; break;
            case '-':
                temp=b-a; break;
            case '*':
                temp=b*a; break;
            case '/':
                temp=b/a; break;
            case '%':
                temp=b%a; break;
            case '^':
                temp=pow(b,a);
        }/*End of switch */
        push(temp);
    }/*End of else*/
}/*End of for */
result=pop();
return result;
}/*End of eval_post() */

```

Sparse Matrix-

Matrices, which have more zero entries are called as sparse matrices. Matrix has presentation in row and column only. So it can be presented in two dimensional array. Suppose a matrix has many zero entries and we have work only with non zero entries. So it's a wastage of memory. Now we want to save the memory space which every zero entry is using. So alternate method is to save only non-zero entries which can be through sparse matrix. We will see the 3-tuple method for presenting sparse matrix.

0	1	0
3	5	0
0	0	2

Sparse Matrix

3-tuple Method-

As the name implies, every nonzero entry of sparse matrix represented by three tuples. First tuple represents row, second for column and third for value. Here all the elements of matrix will come sequentially. It can be row major or column major 3-tuple presentation of sparse matrix. First row will represent the number of rows, number of columns and number of non zero elements of matrix. After that each row will represent non zero entries of matrix. Let us take a matrix of 4×5

	0	-4	5	0	1
	9	0	0	2	0
$M_{4 \times 5} = 0$	0	3	0	0	0
	6	0	0	0	12

Now we have to represent this matrix in 3-tuple representation of sparse matrix. Here total number of rows is 4, number of columns are 5 and non zero entries are 8. So sparse matrix will be-

$M =$	Row	Column	Value
	4	5	8
	1	2	-4
	1	3	5
	1	5	1
	2	1	9
	2	4	2
	3	3	3
	4	1	6
	4	5	12

Here it's a row major 3-tuple presentation. So elements are coming sequentially on the basis of row. Suppose we want to represent column major 3-tuple sparse matrix then it will be as-

$M =$	Column	Row	Value
	5	4	8
	1	2	9
	1	4	6
	2	1	-4
	3	1	5
	3	3	3
	4	2	2
	5	1	1
	5	4	12

```

/* Program of sparse matrix for 3-tuple method using array*/
#include<stdio.h>
#define srow 50
#define mrow 20
#define mcolumn 20

main( )
{
    int mat[mrow][mcolumn],sparse[srow][3];
    int i,j,nzero=0,mr,mc,sr,s;
    printf("Enter number of rows : ");
    scanf("%d",&mr);
    printf("Enter number of columns : ");
    scanf("%d",&mc);
    for(i=0;i<mr;i++)
        for(j=0;j<mc;j++)
    {
        printf("Enter element for row %d,column %d : ",i+1,j+1);

        scanf("%d",&mat[i][j]);
    }
    printf("Entered matrix is :\n");
    for(i=0;i<mr;i++)
    {
        for(j=0;j<mc;j++)
        {
            printf("%6d",mat[i][j]);
            if(mat[i][j]!=0)
                nzero++;
        }
        printf("\n");
    }
    sr=nzero+1;
    sparse[0][0]=mr;
    sparse[0][1]=mc;
    sparse[0][2]=nzero;
    s=1;
    for(i=0;i<mr;i++)
        for(j=0;j<mc;j++)
    {
        if(mat[i][j]!=0)
        {
            sparse[s][0]=i+1;
            sparse[s][1]=j+1;
            sparse[s][2]=mat[i][j];
            s++;
        }
    }
}

```

```

printf("Sparse matrix is :\n");
for(i=0;i<sr;i++)
{
    for(j=0;j<3;j++)
        printf("%5d",sparse[i][j]);
    printf("\n");
}
/*End of main( )*/

```

Exercise

1. Write a program to convert an infix expression to prefix and evaluate that prefix expression.
2. Take a stack of size 5. Push the elements 5, 10, 15, 20. Now do the following operations and show the stack in figure.
 - (i) pop
 - (ii) push 2
 - (iii) push 4
 - (iv) push 6
 - (v) pop
3. Write a program of stack where element will be pushed from last position of array.
4. Write a program to implement 2 stacks in one array where first stack will start from 0th position and second stack will start from last position of array.
5. Convert following infix expressions into equivalent postfix and prefix
 - (i) $(a+b)*(c+d)$
 - (ii) $a\%(c-d)+b*e$
 - (iii) $a-(b+c)*d/e$
 - (iv) $h+(j^k)*i\%s$
 - (v) $f^*(g-h)+(j/k)$
6. Convert the following postfix arithmetic expression into infix and evaluate it.
 $4 \ 5 \ + \ 3 \ 6 \ * \ - \ 3 \ 2 \ ^ \ + \ 8 \ 2 \ / \ 3 \ * \ - \ 5 \ 7 \ * \ +$
7. Show the following postfix arithmetic expression evaluation in stack.
 $3 \ 9 \ 6 \ - \ ^ \ 6 \ 2 \ / \ 2 \ * \ + \ 7 \ 3 \ \% \ -$
8. Write a program to implement a stack which contains the address of the pointers for allocation of memory. Do the pop operation on stack and free the popped pointers.
9. Implement a small calculator through stack to calculate basic arithmetic expressions.

188

10. Write a program of stack to display the elements in FIFO manner.
11. Write a program to implement 2 queues in one array where first queue will start from 0th position and second queue will start from last position of array.
12. Let us take a circular queue of size 8-

front = 2 rear = 6 Queue : _ A B C D E _

Do the following operations and show the positions of rear and front after every operation

- (i) Add element F in queue
- (ii) Add element G in queue
- (iii) Delete an element from queue
- (iv) Add element H in queue
- (v) Add element I in queue.
- (vi) Add element J in queue
- (vii) Delete an element from queue

13. Let us take a input restricted dequeue of size 5 implemented in a circular array

left=1 right = 3 dequeue : _ A B C _

Do the following operations on this dequeue

- (i) Add element D in the dequeue .
- (ii) Add element E in the dequeue .
- (iii) Delete an element from the left .
- (iv) Delete an element from the right.

14. Let us take a output restricted dequeue of size 7 implemented in a circular array

left=2 right=5 dequeue : _ B C D E _

Do the following operations on this dequeue

- (i) Add element A from right
- (ii) Add element F from left
- (iii) Add element G from left
- (iv) Delete an element from dequeue

15. Write a program of priority queue implemented through array.
16. Modify the program of priority queue such that the element having least priority is deleted first.

Chapter 7

Sorting

In our daily life, we can see so many applications use the data in sorted order as telephone directory, railway reservation tickets, merit list, roll numbers etc. But how concept of sorting came in picture. What exactly is sorting and why the need of sorting arises. Some years back a big research has gone only on sorting and this can be discussed or a research can be done as a separate topic. Sorting is nothing but storage of data in sorted order, it can be in ascending or descending order. But why data is needed in sorted order. The term, which comes in picture, is searching. We have a need to search the data, particular record, particular telephone number, particular roll number in merit list. There are so many things in our life where searching is needed for getting information. But question remains why sorting? How it is related with searching? Oh yes, answer is very simple, just go to index part of your book, search the word sorting for getting information of page number. No, No, you are going directly to the word starting from 'S' alphabet. Yes because of sorting only. Here words are sorted in ascending order based on alphabets. Suppose here index is not in sorted order and contains 5000 words, then you have only one option for searching a particular word i.e. one by one. Similar case is for telephone directory, bank account number. But this sorting is on a particular key, here key is word. So every record, which is going to be sorted, will contain one key. Suppose we have a record of employees. Every record has information of-

- Employee Number
- Name
- Age
- Designation
- Date of joining

Here we can take Employee Number as a key for sorting. Index contains word and page number and it is sorted based on word as a key. There is a possibility for the same key value for two records. This sorting is called stable, here the record which came first during sorting process will be before the record which has same key value but was next in sorting process.

The sorting methods can be divided in two parts i.e. internal and external. In internal sorting, data that is going to be sorted will be in main memory. In external sort, data will be on auxiliary storage like tape, floppy, disk etc. Here we will focus only on internal sorting.

We can do sorting in two ways, first we can sort on the basis of key where we will move the whole record from one position to another. Let us take an example-

<u>Index</u>	<u>Key</u>	<u>Page No.</u>	<u>Key</u>	<u>Page No.</u>
	Word		Word	
	Tree	322	Graph	415
	Graph	415	Hashing	610
	Sorting	516	List	45
	List	610	Queue	110
	Hashing	110	Sorting	516
	Queue	45	Tree	322

Here we can see sorting is based on key word and we are moving whole record for sorting. But suppose every record has large volume of data then moving of data will be really overhead in sorting.

In second way of sorting we maintain the pointer table, where each entry of table points to the record. Now in sorting we can easily move the pointer position rather than whole record in sorting process. This sorting technique is called sorting by address. Let us take an example-

Pointer Table	Word	Page No.
1000	Tree	322
1002	Graph	415
1004	Sorting	516
1006	List	45
1008	Hashing	610
1010	Queue	110

Now after sorting pointer table will be –

1002
1008
1006
1010
1004
1000

Here we can see records are at the same position but the pointers, which are pointing to records, are in different sequence but still pointing to the same record.

Now we will go to the actual sorting technique but before that I should tell that no sorting technique is best, we cannot say this is better than that, it is totally dependent on type of data, particular situation and complexity. In some particular situation and type of data one

algorithm can perform better than other but in other situation and in complexity it can be worst. We will discuss these things later but one thing is clear before going to implement any sorting technique, all these things should be kept in mind.

Efficiency Parameters-

There are many methods for sorting. There are several different situations where behaviour of particular sorting technique will entirely change. So before going to implement particular sorting technique it is very important to know which technique behaves how in every situation. Without knowing these things we cannot go for best sorting technique for a particular problem. But how we will come to know which sorting technique will be best for this problem. Definitely we will go for some parameters for getting this knowledge. First parameter is execution time of program means how much time is taken for execution of program. Second is the space and third is of course coding time of that program.

Suppose data is in small quantity and doesn't have too much importance and sorting is needed only at few occasions then there is no need to search for best sorting technique and implementing of complicated sorting technique for getting less execution time and less space. Important thing here is only coding time. So we can take any simpler sorting technique.

But we have a need to be very careful in analyzing the situation because may be it's a very small part of system but can be very crucial like satellite launch system. So we should know the positive and also negative impact of that sorting technique in a particular situation.

The execution time does not depend only on the volume of data, it has some other measurement also. The main component in any sorting is the key comparison because most of the time we have a need to compare the key. After that moving a record is also a parameter in execution time. So we can say the number of times the key comparison and moving record is the main component for affecting execution time.

The best way is to go through mathematics and get the equation for each case, best, worst and average. Through equation we can get the information how execution time is varying with different volume of data.

Efficiency of sorting-

As we have seen the efficiency criteria of any sorting technique. Now we will go for efficiency of particular sort.. This we can know by mathematical analysis that it has of $O(n)$, $O(n \log n)$ or $O(n^2)$. Still now there is no sorting technique which is of $O(n)$. All sorting techniques are in between $O(n \log n)$ to $O(n^2)$. We can see clear difference in $O(n \log n)$ and $O(n^2)$ for any number of records.

Let us take the $n=100$

$O(n \log n)$ has execution time only $100 \log 100$ which is very less compared to $O(n^2)$ of 100^{100}

Actually the main factor for the efficiency is type of data means in which order it's coming. Sometimes data is coming in sorted order but particular sorting technique is taking time of $O(n^2)$. But that does not mean that this sorting technique is not good, maybe it will take time of $O(n \log n)$ in worst case situation.

So the need is of exact analysis of the situation, suppose most of the time data is coming in sorted order then we should take the appropriate sorting technique for this situation. Suppose most of the time data is coming as worst case then we should take the sorting technique which behaves very good in worst case. So we can see the implementation of particular sorting technique depends on the situation rather than the order of that technique.

But once sorting technique is implemented, then we should go for the time efficiency of that sorting technique like try to compare keys in a better way, avoiding more operations after matching the key, code optimization technique. So we can improve the time efficiency of sorting technique as well as system because it's affecting system also.

Bubble Sort-

If N elements are given in memory then for sorting we do following steps-

1. First compare the 1st and 2nd element of array if $1st < 2nd$ then compare the 2nd with 3rd.
2. If $2nd > 3rd$ then interchange the value of 2nd and 3rd.
3. Now compare the value of 3rd (which has the value of 2nd) with 4th.
4. Similarly compare until the $N-1$ th element is compared with N th element.
5. Now the highest value element is reached at the N th place.
6. Now elements will be compared until $N-1$ elements.

Ex-

Let us take the elements are-

13	32	20	62	68	52	38	46
----	----	----	----	----	----	----	----

Pass 1-

- (1) Compare 1st and 2nd element, $13 < 32$ No change
- (2) Compare 2nd and 3rd element, $32 > 20$ Interchange

13	20	32	62	68	52	38	46
----	----	----	----	----	----	----	----

- (3) Compare 3rd and 4th element, $32 < 62$ No change
- (4) Compare 4th and 5th element, $62 < 68$ No change
- (5) Compare 5th and 6th element, $68 > 52$ Interchange

13 20 32 62 52 68 38 46

- (6) Compare 6th and 7th element, $68 > 38$ Interchange

13 20 32 62 52 38 68 46

- (7) Compare 7th and 8th element, $68 > 46$ Interchange

13 20 32 62 52 38 46 68

Pass 2-

Now we will show only interchange-

13 20 32 52 62 38 46 68
 13 20 32 52 38 62 46 68
 13 20 32 52 38 46 62 68

Pass 3-

13 20 32 38 52 46 62 68
 13 20 32 38 46 52 62 68

Pass 4-

13 20 32 38 46 52 62 68

```
/* Program of sorting using bubble sort */
#include <stdio.h>
#define MAX 20
main()
{
    int arr[MAX], i, j, k, temp, n, exchanges;
    printf("Enter the number of elements : ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        printf("Enter element %d : ", i+1);
        scanf("%d", &arr[i]);
    }
    printf("Unsorted list is :\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
```

```

printf("\n");

/* Bubble sort*/
for (i = 0; i < n-1 ; i++)
{
    xchanges=0;
    for (j = 0; j <n-1-i; j++)
    {
        if (arr[j] > arr[j+1])
        {
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
            xchanges++;
        }/*End of if*/
    }/*End of inner for loop*/
    if(xchanges==0) /*If list is sorted*/
        break;
    printf("After Pass %d elements are : ",i+1);
    for (k = 0; k < n; k++)
        printf("%d ", arr[k]);
    printf("\n");
}/*End of outer for loop*/

printf("Sorted list is :\n");
for (i = 0; i < n; i++)
    printf("%d ", arr[i]);
printf("\n");
}/*End of main()*/

```

Analysis-

In bubble sort, $n-1$ comparisons will be in 1st pass, $n-2$ in 2nd pass, $n-3$ in 3rd pass and so on. So it's very simple to calculate the number of comparisons. Total number of comparisons will be -

$$(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$$

It's a form of arithmetic progression series. So we can apply formula -

$$\begin{aligned} \text{Sum} &= n/2 [2a + (n-1)d] \\ &= (n-1)/2 [2 \times 1 (n-1-1) \times 1] \\ &= (n-1)/2 [2 + n - 2] \\ &= n(n-1)/2 \end{aligned}$$

which is of $O(n^2)$.

The main advantage is the simplicity of algorithm, additional space requirement is only one temporary variable and it behaves as $O(n)$ for sorted array of element.

Selection Sort-

As the name suggests selection sort is the selection of an element and keeping it in sorted order. If you have a list of elements in unsorted order and you want to make a list of elements in sorted order then first you will take the smallest element and keep in the new list, after that second smallest element and so on until the largest element of list.

Let us take an array arr[0], arr[1].....arr[N-1] of elements. First you will search the position of smallest element from arr[0].....arr[N-1]. Then you will interchange that smallest element with arr[0]. Now you will search position of smallest element (second smallest element because arr[0] is the first smallest element) from arr[1].....arr[N-1], then interchange that smallest element with arr[1]. Similarly the process will be for arr[2].....arr[N-1]. The whole process will be as-

Pass 1 :

1. Search the smallest element from arr[0].....arr[N-1].
2. Interchange arr[0] with smallest element.

Result : arr[0] is sorted.

Pass 2 :

1. Search the smallest element from arr[1].....arr[N-1].
2. Interchange arr[1] with smallest element.

Result : arr[0], arr[1] is sorted.

.....
.....
.....

Pass N-1 :

1. Search the smallest element from arr[N-2] and arr[N-1].
2. Interchange arr[N-2] with smallest element.

Result : arr[0].....arr[N-1] is sorted.

Let us take list of elements in unsorted order and sort them by applying selection sort.

Pass							
1.	75	35	42	13	87	24	64
2.	13	35	42	75	87	24	64
3.	13	24	42	75	87	35	64
4.	13	24	35	75	87	42	64
5.	13	24	35	42	87	75	64
6.	13	24	35	42	57	75	64
7.	13	24	35	42	57	64	75
	13	24	35	42	57	64	75

Now the elements are in sorted order-

13 24 35 42 57 64 75 87

```
/*Program of sorting using selection sort*/
#include <stdio.h>
#define MAX 20

main()
{
    int arr[MAX], i,j,k,n,temp,smallest;
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for (i = 0; i < n; i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d", &arr[i]);
    }
    printf("Unsorted list is : \n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    /*Selection sort*/
    for(i = 0; i< n - 1 ; i++)
    {
        /*Find the smallest element*/
        smallest = i;
        for(k = i + 1; k < n ; k++)
        {
            if(arr[smallest] > arr[k])
                smallest = k ;
        }
        if( i != smallest )
        {
            temp = arr [i];
            arr[i] = arr[smallest];
            arr[smallest] = temp ;
        }
        printf("After Pass %d elements are : ",i+1);
        for (j = 0; j < n; j++)
            printf("%d ", arr[j]);
        printf("\n");
    }/*End of for*/
    printf("Sorted list is : \n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}/*End of main()*/

```

Analysis-

As we have seen selection sort algorithm will search the smallest element in the array and then that element will be at proper position. So in Pass 1 it will compare $n-1$ elements. Same process will be for Pass 2 but this time comparison will be $n-2$ because first element is already at proper position. The elements, which are already at correct position, will not be disturbed. Same thing we will do for other passes. We can easily write function for comparisons as-

$$F(n) = (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$$

This is arithmetic series in decreasing order, so applying the formula-

$$\text{Sum} = n/2 [2a + (n-1)d]$$

Where n = Number of elements in series, a = First element in series and d = difference between second element and first element

or 2nd element - 1st element

Hence,

$$\begin{aligned} F(n) &= (n-1)/2 [2(n-1) + \{ (n-1) - 1 \} \{ (n-2) - (n-1) \}] \\ &= (n-1)/2 [2n - 2 + (n-1-1)(n-2-n+1)] \\ &= (n-1)/2 [2n - 2 + (n-2)(-1)] \\ &= (n-1)/2 [2n - 2 - n + 2] \\ &= n(n-1)/2 \\ &= O(n^2) \end{aligned}$$

Since, selection sort doesn't see the order of elements, so its behaviour is near about same for worst and best case. The best thing with selection sort is that in every pass one element will be at correct position, very less temporary variables will be required for interchanging the elements and it is simple to implement.

Insertion Sort-

The insertion sort inserts each element in proper place. This is same as playing cards, in which we place the cards in proper order. There are n elements in the array and we place each element of array at proper place in the previously sorted element list.

Let us take there are N elements in the array arr. Then process of inserting each element in proper place is as-

Pass 1 - arr[0] is already sorted because of only one element.

Pass 2 – arr[1] is inserted before or after arr[0].

So arr[0] and arr[1] are sorted.

Pass 3 – arr[2] is inserted before arr[0], in between arr[0] and arr[1] or after arr[1].

So arr[0], arr[1] and arr[2] are sorted.

Pass 4- arr[3] is inserted into its proper place in array arr[0], arr[1], arr[2]

So arr[0], arr[1], arr[2], arr[3] are sorted.

Pass N- arr[N-1] is inserted into its proper place in array

arr[0], arr[1], arr[N-2]

So arr[0], arr[1], arr[N-1] are sorted.

The element inserted in the proper place is compared with the previous elements and placed in between the i^{th} element and $i+1^{th}$ element if

element $\geq i^{th}$ element

element $\leq (i+1)^{th}$ element

Ex-

Let us take the elements are-

82	42	49	8	92	25	59	52
----	----	----	---	----	----	----	----

Pass 1-

82	42	49	8	92	25	59	52
----	----	----	---	----	----	----	----

Pass 2-

82	42	49	8	92	25	59	52
----	----	----	---	----	----	----	----

Pass 3-

42	82	49	8	92	25	59	52
----	----	----	---	----	----	----	----

Pass 4-

42	49	82	8	92	25	59	52
----	----	----	---	----	----	----	----

Pass 5-

8	42	49	82	92	25	59	52
---	----	----	----	----	----	----	----

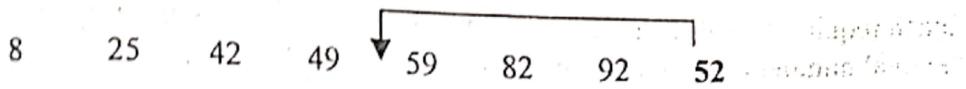
Pass 6-

8	42	49	82	92	25	59	52
---	----	----	----	----	----	----	----

Pass 7-

8	25	42	49	82	92	59	52
---	----	----	----	----	----	----	----

Pass 8-



Sorted elements-



/* Program of sorting using insertion sort */

```
#include <stdio.h>
#define MAX 20
```

```
main()
{
    int arr[MAX], i, j, k, n;
    printf("Enter the number of elements : ");
    scanf("%d", &n);
    for (i = 0; i < n; i++)
    {
        printf("Enter element %d : ", i+1);
        scanf("%d", &arr[i]);
    }
    printf("Unsorted list is :\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    /*Insertion sort*/
    for(j=1;j<n;j++)
    {
        k=arr[j]; /*k is to be inserted at proper place*/
        for(i=j-1;i>=0 && k<arr[i];i--)
            arr[i+1]=arr[i];
        arr[i+1]=k;
        printf("Pass %d, Element inserted in proper place: %d\n", j, k);
        for (i = 0; i < n; i++)
            printf("%d ", arr[i]);
        printf("\n");
    }
    printf("Sorted list is :\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}/*End of main()*/
```

Analysis-

In insertion sort we insert the element before or after and we start comparison from the first element. Since first element has no other elements before it, so it does not require any comparison. Second element requires 1 comparison, third requires 2 comparisons.

fourth requires 3 comparisons and so on. The last element requires $n-1$ comparisons. So the total number of comparisons will be-

$$1 + 2 + 3 + \dots + (n-2) + (n-1)$$

It's a form of arithmetic progression series, so we can apply formula-

$$\begin{aligned} \text{Sum} &= n/2 [2 \times 1 + (n-1) \times 1] \\ &= (n-1)/2 [2 \times 1 + (n-1-1) \times 1] \\ &= (n-1)/2 [2 + n - 2] \\ &= n(n-1)/2 \end{aligned}$$

which is of $O(n^2)$.

It's a worst case behaviour of insertion sort where all elements are in reverse order.

The advantage of insertion sort is it's simplicity and it is very efficient when number of elements to be sorted are very less. Because for smaller file size n the difference between $O(n^2)$ and $O(n \log n)$ is very less and $O(n \log n)$ has complex sorting technique. Insertion sort behaves as of $O(n)$ when elements are in sorted order and also has worth when list of elements are near about sorted.)

Shell Sort (Diminishing Increment Sort) –

D.L.Shell proposed an improvement on insertion sort in 1959 named after him as shell sort. In insertion sort items can be moved only at one position because it compares with only adjacent items. If we takes the item at a particular distance (increments) then we can compare items which are far apart and we can sort them.

Afterward we decrease the increment and do this process again. At last we take the increment 1 and sort them with insertion sort.

Let us take an array from $\text{arr}[0], \dots, \text{arr}[N-1]$ and we take the distance (increments) 4 for grouping together the items then items will be grouped in this order-

First : $\text{arr}[0], \text{arr}[4], \text{arr}[8], \dots$
 Second : $\text{arr}[1], \text{arr}[5], \text{arr}[9], \dots$
 Third : $\text{arr}[2], \text{arr}[6], \text{arr}[10], \dots$
 Fourth : $\text{arr}[3], \text{arr}[7], \text{arr}[11], \dots$

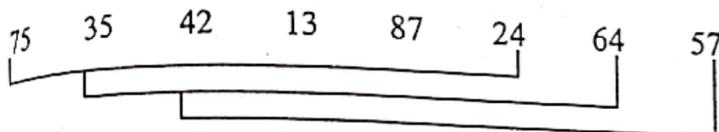
We can see that we have to make the list equal to the increments and it will cover all the items from $\text{arr}[0], \dots, \text{arr}[N-1]$. First we sort this list with insertion sort then decrease the increment and do this process again. At the end list will be maintained with increment 1 and we will sort them with insertion sort.

We take the decreasing increments for making the list so this sorting is also called as diminishing increment sort.

Let us take list of elements in unsorted order and sort them by applying shell sort.

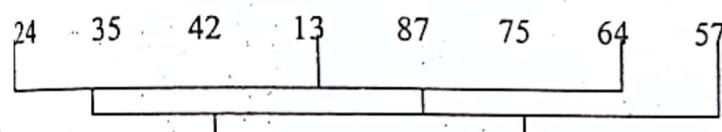
75 35 42 13 87 24 64 57

Pass 1 : (increment = 5)



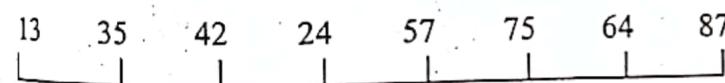
24 35 42 13 87 75 64 57

Pass 2 : (increment = 3)



13 35 42 24 57 75 64 87

Pass 3 : (increment = 1)



Sorted elements-

13 24 35 42 57 64 75 87

The main thing you have in your mind at this time is how the increments should be taken. There is no restriction on sequence of increments but it should be 1 at the last. It will be useful to avoid the increments as power of 2 as 4,2,1 because the items compared at one pass will be compared again at next pass.

```
/* Program of sorting using shell sort */
#include <stdio.h>
#define MAX 20
main()
{
    int arr[MAX], i,j,k,n,incr;
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d",&arr[i]);
    }
}
```

```

        scanf("%d",&arr[i]);
    }
    printf("Unsorted list is :\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\nEnter maximum increment (odd value) : ");
    scanf("%d",&incr);
    /*Shell sort*/
    while(incr>=1)
    {
        for(j=incr;j<n;j++)
        {
            k=arr[j];
            for(i = j-incr; i >= 0 && k < arr[i]; i = i-incr)
                arr[i+incr]=arr[i];
            arr[i+incr]=k;
        }
        printf("Increment=%d \n",incr);
        for (i = 0; i < n; i++)
            printf("%d ", arr[i]);
        printf("\n");
        incr=incr-2; /*Decrease the increment*/
    }/*End of while*/
    printf("Sorted list is :\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
/*End of main()*/
}

```

Analysis-

As we know shell sort is nothing but an improvement on insertion sort. Insertion sort is very efficient when number of elements are less. When N is small then difference between $O(n^2)$ and $O(n \log n)$ will also be less because sorting technique of $O(n^2)$ is simple and has less movement of elements compare to complex sorting technique of $O(n \log n)$. Shell sort takes increment for creating subfile which will be smaller and suitable for fast insertion sort.

So on each increment most of the elements will be in sorted order and even when file size will be larger then also insertion sort will be efficient.

The efficiency of shell sort can be found on some condition with mathematical analysis which is quite difficult. This sorting technique is based on size of increments, means how many increments and value of these increments also. It will be better if increments are not of power of 2 eg. 8 4 2 1 because in the next pass same elements will be compared again. We can take prime number for increment value also. It is found that for a particular sequence of increments it is of $O(n(\log n)^2)$ and running time is xn^y where x is in between 1 to 1.6 and y is near about 1.25.

Merging-

If there are two sorted lists of array then process of combining these sorted lists into sorted order is called merging.

Let us take two arrays arr1 and arr2 in sorted order, then for merging we will combine them in arr3 with sorted order.

arr1 - 5, 8, 22, 30, 36

arr2 - 4, 9, 25, 28, 34, 40, 42

Here arr1 and arr2 are in sorted order, after combining arr3 will be -

4, 5, 8, 9, 22, 25, 28, 30, 34, 36, 40, 42

We can take this process with two approaches. First one, so simple, take the first array, after that take second array and sort them with any sorting.

But it is not useful because both the lists are sorted and we are taking them as unsorted list.

The second approach is to take one element of each array, compare them and then take the smaller one in third array. Repeat this process until the elements of any array are finished. Then take the remaining elements of unfinished array in third array.



Take 4

arr3 -

4



Take 5

arr3 -

4, 5



Take 8

Compare

394

arr3-

4, 5, 8

22	30	36
----	----	----

arr1

9	25	28	34	40	42
---	----	----	----	----	----

arr2

Compare

Take 9

arr3-

4, 5, 8, 9

22	30	36
----	----	----

arr1

25	28	34	40	42
----	----	----	----	----

arr2

Compare

Take 22

arr3-

4, 5, 8, 9, 22

30	36
----	----

arr1

25	28	34	40	42
----	----	----	----	----

arr2

Compare

Take 25

arr3-

4, 5, 8, 9, 22, 25

30	36
----	----

arr1

28	34	40	42
----	----	----	----

arr2

Compare

Take 28

arr3-

4, 5, 8, 9, 22, 25, 28

30	36
----	----

arr1

34	40	42
----	----	----

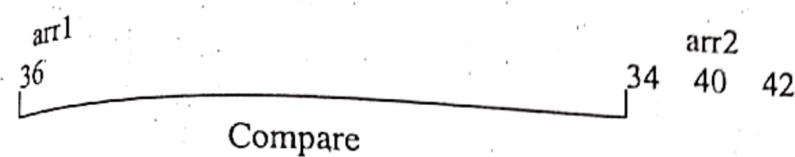
arr2

Compare

Take 30

arr3-

4, 5, 8, 9, 22, 25, 28, 30



Take 34.

arr3-

4, 5, 8, 9, 22, 25, 28, 30, 34

arr1

36

arr2

40 42

Compare

Take 36

arr3-

4, 5, 8, 9, 22, 25, 28, 30, 34, 36

Now take 40, 42 in arr3 because arr1 has no elements to compare

arr3-

4, 5, 8, 9, 22, 25, 28, 30, 34, 40, 42

/* Program of merging two sorted arrays into a third sorted array*/

#include<stdio.h>

main()

{

 int arr1[20], arr2[20], arr3[40];

 int i, j, k;

 int max1, max2;

 printf("Enter the number of elements in list1 : ");

 scanf("%d", &max1);

 printf("Take the elements in sorted order :\n");

 for(i=0; i<max1; i++)

 {

 printf("Enter element %d : ", i+1);

 scanf("%d", &arr1[i]);

 }

 printf("Enter the number of elements in list2 : ");

 scanf("%d", &max2);

 printf("Take the elements in sorted order :\n");

 for(i=0; i<max2; i++)

 {

 printf("Enter element %d : ", i+1);

 scanf("%d", &arr2[i]);

 }

 /* Merging */

 i=0; /* Index for first array */

```

j=0; /*Index for second array*/
k=0; /*Index for merged array*/;

while( (i < max1) && (j < max2) )
{
    if( arr1[i] < arr2[j] )
        arr3[k++]=arr1[i++];
    else
        arr3[k++]=arr2[j++];
}

/*End of while*/
/*Put remaining elements of arr1 into arr3*/
while( i < max1 )
    arr3[k++]=arr1[i++];

/*Put remaining elements of arr2 into arr3*/
while( j < max2 )
    arr3[k++]=arr2[j++];

/*Merging completed*/
printf("List 1 : ");
for(i=0;i<max1;i++)
    printf("%d ",arr1[i]);
printf("\nList 2 : ");
for(i=0;i<max2;i++)
    printf("%d ",arr2[i]);
printf("\nMerged list : ");
for(i=0;i<max1+max2;i++)
    printf("%d ",arr3[i]);
printf("\n");
}/*End of main()*/

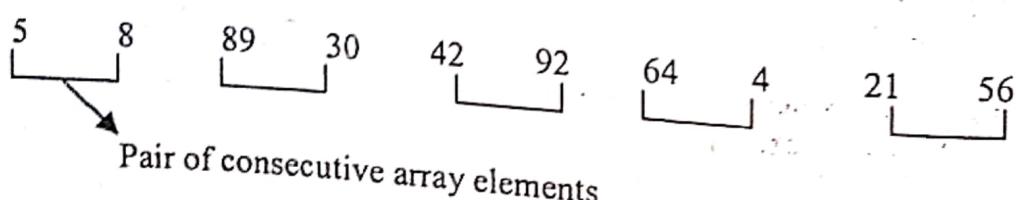
```

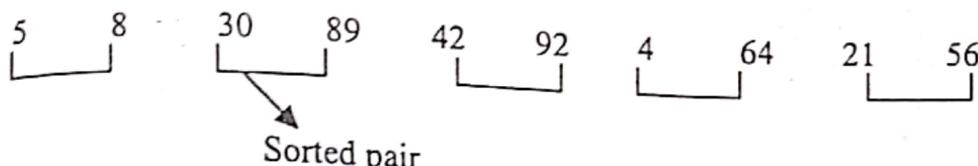
Merge Sort-

Similarly, in second approach of merging, we can take the pair of consecutive elements, merge them in sorted array and then take adjacent pair of array elements and so on until the all elements of array are in single list.

Let us take an array of elements and process the merge sort as –

Pass 1- Merge the two sorted pairs in a sorted pair. Initially we have pair size 1. Merge these pair of size 1 into pairs of size 2.





Pass 2- Similarly merge the pairs of size 2 into pair of size 4.



Here 5, 8 and 30, 89 are in one pair and 21, 56 is not merged because it is single pair and there is no other pair to merge.

Pass 3- Merge the two sorted pairs in a sorted pair.



Pass 4- Merge the two sorted pairs in a sorted pair.

4 5 8 21 30 42 56 64 89 92

Now the sorted list is-

4 5 8 21 30 42 56 64 89 92

```
/* Program of sorting using merge sort without recursion*/
#include<stdio.h>
#define MAX 30

main()
{
    int arr[MAX],temp[MAX],i,j,k,n,size,l1,h1,l2,h2;

    printf("Enter the number of elements : ");
    scanf("%d",&n);

    for(i=0;i<n;i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d",&arr[i]);
    }

    printf("Unsorted list is : ");
    for( i = 0 ; i < n ; i++ )
        printf("%d ", arr[i]);
}
```

```

/*11 lower bound of first pair and so on*/
for(size=1; size < n; size=size*2)
{
    l1=0;
    k=0; /*Index for temp array*/
    while( l1+size < n)
    {
        h1=l1+size-1;
        l2=h1+1;
        h2=l2+size-1;
        if( h2>=n ) /* h2 exceeds the limit of arr */
            h2=n-1;
        /*Merge the two pairs with lower limits l1 and l2*/
        i=l1;
        j=l2;
        while(i<=h1 && j<=h2 )
        {
            if( arr[i] <= arr[j] )
                temp[k++]=arr[i++];
            else
                temp[k++]=arr[j++];
        }
        while(i<=h1)
            temp[k++]=arr[i++];
        while(j<=h2)
            temp[k++]=arr[j++];
        /***Merging completed*/
        l1=h2+1; /*Take the next two pairs for merging */
    }/*End of while*/

    for(i=l1; k<n; i++) /*any pair left */
        temp[k++]=arr[i];

    for(i=0;i<n;i++)
        arr[i]=temp[i];

    printf("\nSize=%d \nElements are : ",size);
    for( i = 0 ; i<n ; i++)
        printf("%d ", arr[i]);
}/*End of for loop */
printf("Sorted list is :\n");
for( i = 0 ; i<n ; i++)
    printf("%d ", arr[i]);
printf("\n");
}/*End of main()*/

```

We can implement merge sort with recursion also.

```

/* Program of sorting using merge sort through recursion*/
#include<stdio.h>
#define MAX 20
int array[MAX];
main()
{
    int i,n;
    printf("Enter the number of elements : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d",&array[i]);
    }
    printf("Unsorted list is :\n");
    for( i = 0 ; i<n ; i++)
        printf("%d ", array[i]);
    merge_sort( 0, n-1 );
    printf("\nSorted list is :\n");
    for( i = 0 ; i<n ; i++)
        printf("%d ", array[i]);
    printf("\n");
}/*End of main()*/

merge_sort( int low, int high )
{
    int mid;
    if( low != high )
    {
        mid = (low+high)/2;
        merge_sort( low , mid );
        merge_sort( mid+1, high );
        merge( low, mid, high );
    }
}/*End of merge_sort*/

merge( int low, int mid, int high )
{
    int temp[MAX];
    int i = low;
    int j = mid + 1 ;
    int k = low ;
    while( (i <= mid) && (j <=high) )
    {
        if(array[i] <= array[j])
            temp[k++] = array[i++];
        else
            temp[k++] = array[j++];
    }
}/*End of while*/

```

```

while( i <= mid )
    temp[k++]=array[i++];
while( j <= high )
    temp[k++]=array[j++];

for(i= low; i <= high ; i++)
    array[i]=temp[i];
}/*End of merge()*/

```

Analysis-

Let us take an array of size n is used for merge sort. Because here we take elements in pair and merge with another pair after sorting. So merge sort requires maximum $\log_2 n$ passes. Each pass, total number of comparison can be maximum n . Hence we can say merge sort requires maximum $n * \log_2 n$ comparisons which is of $O(n\log_2 n)$.
The main disadvantage of merge sort is space requirement It requires extra space of $O(n)$.

Radix Sort-

If we have a list of 100 names and we want to sort them alphabetically then first we take the names which start with alphabet 'A' and then we take the names which start from alphabet 'B' and so on. In list of names the radix is 26(all alphabets). We take all the names in different parts. In second pass we take the names in parts on the basis of second alphabets of names. Similar process we take for other passes.

Similarly if we have list of numbers then there will be 10 parts from 0 to 9 because radix is 10. In first pass we take the numbers in parts on the basis of unit digit(least significant digit). In second pass the base will be tens digit and similarly for other passes. If the largest number has 3 digits then number will be sorted in maximum 3 passes.
Let us take number in unsorted order and sort them by applying radix sort.

Pass 1- 233 124 209 345 498 567 328 163

Numbers	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
233				233						
124					124					
209										209
345						345				
498									498	
567								567		
328									328	
163				163						

Taking the numbers on the basis of unit(first from last) digit

Here we can see after Pass1 unit numbers are in sorted order.

After pass1 numbers are as -

233 163 124 345 567 498 328 209

Pass 2-

Numbers	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
233				233						
163							163			
124			124							
345					345					
567							567			
498										498
328			328							
209	209									

Taking the numbers on the basis of tens(second from last) digit

Now the whole number with unit and tens digit are in sorted order

After pass2 numbers are as -

209 124 328 233 345 163 567 498

Pass 3-

Numbers	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
209			209							
124		124								
328				328						
233			233							
345				345						
163										163
567							567			
498					498					

Now the sorted list is -

124 163 209 233 328 345 498 567

Here we take number of queues based on radix. Suppose radix is 10 then we takes 10 queues. In each pass we insert the element into corresponding queue based on the digit of element. In pass1 it will be inserted based on unit digit, in pass2 based on tens digit and so on. After each pass we concatenate all the queues and in next pass again it will be inserted based on next digit.

```
/*Program of sorting using radix sort*/
#include<stdio.h>
#include<malloc.h>

struct node
{
    int info ;
    struct node *link;
}*start=NULL;

main()
{
    struct node *tmp,*q;
    int i,n,item;

    printf("Enter the number of elements in the list : ");
    scanf("%d", &n);

    for(i=0;i<n;i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d",&item);

        /* Inserting elements in the linked list */
        tmp= malloc(sizeof(struct node));
        tmp->info=item;
        tmp->link=NULL;

        if(start==NULL) /* Inserting first element */
            start=tmp;
        else
        {
            q=start;
            while(q->link!=NULL)
                q=q->link;
            q->link=tmp;
        }
    }/*End of for*/

    printf("Unsorted list is :\n");
    display();
    radix_sort();
}
```

```

printf("Sorted list is :\n");
display();
/*End of main()*/
display()
{
    struct node *p=start;
    while( p !=NULL)
    {
        printf("%d ", p->info);
        p= p->link;
    }
    printf("\n");
}/*End of display()*/
radix_sort()
{
    int i,k,dig,maxdig,mindig,least_sig,most_sig;
    struct node *p, *rear[10], *front[10];
    least_sig=1;
    most_sig=large_dig(start);

    for(k = least_sig; k <= most_sig ; k++)
    {
        printf("PASS %d : Examining %dth digit from right ",k,k);
        for(i = 0 ; i <= 9 ; i++)
        {
            rear[i] = NULL;
            front[i] = NULL ;
        }
        maxdig=0;
        mindig=9;
        p = start ;
        while( p != NULL)
        {
            /*Find kth digit in the number*/
            dig = digit(p->info, k);
            if(dig>maxdig)
                maxdig=dig;
            if(dig<mindig)
                mindig=dig;

            /*Add the number to queue of dig*/
            if(front[dig] == NULL)
                front[dig] = p ;
            else
                rear[dig]->link = p ;
            rear[dig] = p ;
            p=p->link; /*Go to next number in the list*/
        }/*End while */
    }
}

```

```

/* maxdig and mindig are the maximum and minimum
   digits of the kth digits of all the numbers*/
printf("mindig=%d maxdig=%d\n",mindig,maxdig);
/*Join all the queues to form the new linked list*/
start=front[mindig];
for(i=mindig;i<maxdig;i++)
{
    if(rear[i+1]!=NULL)
        rear[i]->link=front[i+1];
    else
        rear[i+1]=rear[i];
}
rear[maxdig]->link=NULL;
printf("New list : ");
display();
/* End for */
}/*End of radix_sort*/

/* This function finds number of digits in the largest element of the list */
int large_dig()
{
    struct node *p=start;
    int large = 0,ndig = 0;

    while(p != NULL)
    {
        if(p->info > large)
            large = p->info;
        p = p->link ;
    }
    printf("Largest Element is %d , ",large);
    while(large != 0)
    {
        ndig++;
        large = large/10 ;
    }
    printf("Number of digits in it are %d\n",ndig);
    return(ndig);
} /*End of large_dig() */

/*This function returns kth digit of a number*/
int digit(int number, int k)
{
    int digit, i ;
    for(i = 1 ; i <=k ; i++)
    {
        digit = number % 10 ;

```

```

        number = number /10 ;
    }
    return(digit);
/*End of digit( )*/
}

```

Analysis-

Radix sort is dependent on three things-

1. Radix (10 for decimal, 26 for alphabets and 2 for binary)
2. Number of digits in largest element
3. Size of array of elements.

Number of passes is the number of digits in largest element i.e. p. In each pass we will compare the digits of elements i.e. maximum value of radix r. So the maximum number of comparisons will be

$$P * r * n$$

Worst case- Suppose the number of digits in largest element is equal to number of elements in array then maximum number of comparisons = $p * n * n$ which is of $O(n^2)$.

Best case- In best case the number of digits in largest element will be $\log_p n$. So the maximum number of comparisons = $p * (\log_p n) * n$

which is of $O(n \log n)$.

The disadvantage of radix sort is extra space requirement. Suppose for particular pass all the elements are in same index then it will be in need of space $p * n$.

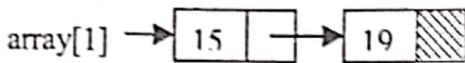
Address Calculation Sort-

We have seen the radix sort where we are using radix for sorting the elements. Here we will use some non decreasing function. Based on this function we will get the key of particular element. If $x < y$ then this non decreasing function will give value $f(x) < f(y)$. Now we will keep that element in sorted linked list corresponding to that particular key. So this sorting is basically based on the address calculated by this function. Let us take list of elements and sort them with address calculation sort-

19	24	49	15	45	33	89	66
----	----	----	----	----	----	----	----

Now we have to create the list. Let's create the list based on the function which gives the key as first digit of element. Obviously this will be non decreasing function. So there will be 10 lists. Here lists are sorted linked list. Starting address of list can be maintained in a array of pointers i.e. array[10]. Let us take first element 19. It's first digit is 1, so it will be in the linked list array[1]. Similarly we can maintain list for other elements.

array[0] → NULL



array[2] → 24 [shaded]

array[3] → 33 [shaded]

array[4] → 45 [shaded]

array[5] → NULL

array[6] → 66 [shaded]

array[7] → NULL

array[8] → 89 [shaded]

array[9] → NULL

/*Program of sorting using address calculation sort*/

#include<stdio.h>

#include<malloc.h>

#define MAX 20

struct node

{

int info;

struct node *link;

};

struct node *head[10];

int n,i,arr[MAX];

main()

{

int i;

printf("Enter the number of elements in the list : ");

scanf("%d", &n);

for(i=0;i<n;i++)

{

printf("Enter element %d : ",i+1);

scanf("%d",&arr[i]);

/*End of for */

printf("Unsorted list is :\n");

```
for(i=0;i<n;i++)
    printf("%d ",arr[i]);
printf("\n");
addr_sort();
printf("Sorted list is :\n");
for(i=0;i<n;i++)
    printf("%d ",arr[i]);
printf("\n");
/*End of main( )*/
addr_sort()
{
    int i,k,dig;
    struct node *p;
    int addr;
    k=large_dig();
    for(i=0;i<=9;i++)
        head[i]=NULL;
    for(i=0;i<n;i++)
    {
        addr=hash_fn( arr[i],k );
        insert(arr[i],addr);
    }
    for(i=0; i<=9 ; i++)
    {
        printf("head(%d) -> ",i);
        p=head[i];
        while(p!=NULL)
        {
            printf("%d ",p->info);
            p=p->link;
        }
        printf("\n");
    }
/*Taking the elements of linked lists in array*/
    i=0;
    for(k=0;k<=9;k++)
    {
        p=head[k];
        while(p!=NULL)
        {
            arr[i++]=p->info;
            p=p->link;
        }
    }
/*End of addr_sort( )*/
}
```

```

/*Inserts the number in sorted linked list*/
insert(int num,int addr)
{
    struct node *q,*tmp;
    tmp= malloc(sizeof(struct node));
    tmp->info=num;
    /*list empty or item to be added in begining */
    if(head[addr] == NULL || num < head[addr]->info)
    {
        tmp->link=head[addr];
        head[addr]=tmp;
        return;
    }
    else
    {
        q=head[addr];
        while(q->link != NULL && q->link->info < num)
            q=q->link;
        tmp->link=q->link;
        q->link=tmp;
    }
}/*End of insert( )*/

/* Finds number of digits in the largest element of the list */
int large_dig( )
{
    int large = 0,ndig = 0 ;

    for(i=0;i<n;i++)
    {
        if(arr[i] > large)
            large = arr[i];
    }
    printf("Largest Element is %d , ",large);
    while(large != 0)
    {
        ndig++;
        large = large/10 ;
    }

    printf("Number of digits in it are %d\n",ndig);
    return(ndig);
} /*End of large_dig( )*/

hash_fn(int number,int k)
{
    /*Find kth digit of the number*/
    int digit,addr,i;
}

```

```

for(i = 1 ; i <=k ; i++)
{
    digit = number % 10 ;
    number = number /10 ;
}
addr=digit;
return(addr);
}/*End of hash_fn( )*/
/* This is another non decresing hash function which can also be used instead of the above hash
function*/
hash_fn(int number,int k)
{
    int addr,i,large=0;
    float tmp;

    for(i=0;i<n;i++)
    {
        if(arr[i] > large)
            large = arr[i];
    }
    tmp=(float)number/large;
    addr=tmp*9;
    return(addr);
}/*End of hash_fn( )*/

```

Analysis-

In address calculation sort we maintain the list and insert the element in particular list as sorted order. So the time requirement is only the insertion time of element in particular list.

Best case- Suppose number of elements to sort is equal to the number of lists then it will behave as of $O(n)$.

Worst case- If number of elements to sort is too much then it will behave as of $O(n^2)$. Because here insertion time of element will matter.

The space needed for address calculation sort will be $2 * n$ and pointers pointing to the lists.

Quick Sort (Partition Exchange Sort) -

The idea behind this sorting is that sorting is much easier in two short lists rather than one long list. C. A. R. Hoare implements the divide and conquer means divide the big

problem into two small problems and then those two small problems into two small ones and so on. As example you have a list of 100 names and you want to list them alphabetically then you will make two lists for names A-L and M-Z from original list. Then you will divide list A-L into A-F and G-L and so on until the list could be easily sorted. Similar policy you will adopt for the list M-Z.

In Quick Sort we divide the original list into two sublists. We choose the item from list called key or pivot from which all the left side of elements are smaller and all the right side of elements are greater than that element. So we can create two lists, one list is on the left side of pivot and second list is on right side of pivot, means pivot will be in its actual position. Hence there are two conditions for choosing pivot-

1. All the elements on the left side of pivot should be smaller or equal to the pivot.
2. All the elements on the right side of pivot should be greater than or equal to pivot.

Similarly we choose the pivot for dividing the sublists until there are 2 or more elements in the sublist.

The process for sorting the elements through quick sort is as-

1. Take the first element of list as pivot.
2. Place pivot at the proper place in list. So one element of the list i.e. pivot will be at it's proper place.
3. Create two sublists left and right side of pivot.
4. Repeat the same process until all elements of list are at proper position in list.

For placing the pivot at proper place we have a need to do the following process-

1. Compare the pivot element one by one from right to left for getting the element which has value less than pivot element.
2. Interchange the element with pivot element.
3. Now the comparison will start from the interchanged element position from left to right for getting the element which has higher value than pivot.
4. Repeat the same process until pivot is at it's proper position.

Let us take a list of element and process through quick sorting-

48	29	8	59	72	88	42	65	95	19	82	68
----	----	---	----	----	----	----	----	----	----	----	----

Here we are taking 48 as pivot and we have to start comparison from right to left. Now the first element less than 48 is 19. So interchange it with pivot i.e. 48.

19	29	8	59	72	88	42	65	95	48	82	68
----	----	---	----	----	----	----	----	----	----	----	----

Now the comparison will start from 19 and will be from left to right. The first element greater than 48 is 59. So interchange it with pivot.

19 29 8 48 72 88 42 65 95 59 82 68

Now the comparison will start from 59 and will be from right to left. The first element less than 48 is 42. So interchange it with pivot.

19 29 8 42 72 88 48 65 95 59 82 68

Now the comparison will start from 42 and will be from left to right. The first element greater than 48 is 72. So interchange it with pivot.

19 29 8 42 48 88 72 65 95 59 88 68

Now the comparison will start from 72 and will be from right to left. There is no element less than 48. So now 48 is at its proper position in the list. So we can divide the list into two sublist, left and right side of pivot.

19 29 8 42 48 88 72 65 95 59 88 68

Sublist 1

Sublist 2

Now we have a need to do the same process for sublists and at the end all the elements of list will be at its proper position.

/*Program of sorting using quick sort through recursion*/

```
#include<stdio.h>
#define MAX 30
```

```
enum bool { FALSE, TRUE };
main()
{
    int array[MAX], n, i;
    printf("Enter the number of elements : ");
    scanf("%d", &n);

    for(i=0; i<n; i++)
    {
        printf("Enter element %d : ", i+1);
        scanf("%d", &array[i]);
    }

    printf("Unsorted list is :\n");
    display(array, 0, n-1);
    printf("\n");
    quick(array, 0, n-1);

    printf("Sorted list is :\n");
    display(array, 0, n-1);
}
```

```

        printf("\n");
    }/*End of main() */

quick(int arr[],int low,int up)
{
    int piv,temp,left,right;
    enum bool pivot_placed=FALSE;
    left=low;
    right=up;
    piv=low; /*Take the first element of sublist as piv */

    if(low>=up)
        return;
    printf("Sublist : ");
    display(arr,low,up);

    /*Loop till pivot is placed at proper place in the sublist*/
    while(pivot_placed==FALSE)
    {
        /*Compare from right to left */
        while( arr[piv]<=arr[right] && piv!=right )
            right=right-1;
        if( piv==right )
            pivot_placed=TRUE;
        if( arr[piv] > arr[right] )
        {
            temp=arr[piv];
            arr[piv]=arr[right];
            arr[right]=temp;
            piv=right;
        }
        /*Compare from left to right */
        while( arr[piv]>=arr[left] && left!=piv )
            left=left+1;
        if(piv==left)
            pivot_placed=TRUE;
        if( arr[piv] < arr[left] )
        {
            temp=arr[piv];
            arr[piv]=arr[left];
            arr[left]=temp;
            piv=left;
        }
    }/*End of while */

    printf("-> Pivot Placed is %d -> ",arr[piv]);
    display(arr,low,up);
    printf("\n");
}

```

```

quick(arr,low,piv-1);
quick(arr,piv+1,up);
/*End of quick( )*/
display(int arr[],int low,int up)
{
    int i;
    for(i=low;i<=up;i++)
        printf("%d ",arr[i]);
}

```

Analysis-

Time requirement of quick sort depends on the position of pivot in the list, how pivot is dividing list into sublists. It may be equal division of list or may be it will not divide also.

Average case- In average case we assume that list is equally divided means list1 is equally divided in two sublists, these two sublists in four sublists and so on. So the total number of elements at particular level will be 2^{l-1} . So total number of steps will be $\log_2 n$.

The number of comparison at any level will be maximum n . So we can say run time of quick sort will be of $O(n \log n)$.

Worst case- Suppose list of elements are already in sorted order. When we find the pivot then it will be first element. So here it produces only 1 sublist which is on right side of first element starts from second element. Similarly other sublists will be created only at right side. The number of comparison for first element is n , second element requires $n-1$ comparison and so on. So the total number of comparison will be-

$$n + (n-1) + (n-2) + \dots + 3 + 2 + 1$$

It's a form of arithmetic progression series. So we can apply formula-

$$\text{Sum} = \frac{n}{2} [2a + (n-1)d]$$

$$= \frac{n}{2} [2 \times 1 + (n-1) \times 1]$$

$$= \frac{n}{2} [2 + n - 1]$$

$$= \frac{n(n-1)}{2}$$

which is of $O(n^2)$.

Binary Tree Sort-

As we have seen the implementation of binary tree where each element will be inserted at proper place, based on its value is greater than or less than the node in traversing, starting from the root. If its value is less than the visiting node value then it will be left child and if its value is greater than the visiting node value then it will be right child of

the visiting node. Now we can go for inorder traversal of constructed binary tree which will give the list of elements in sorted order. So this sorting is based on first creation of binary tree on the basis of given list of elements then in order traversal of constructed binary tree.

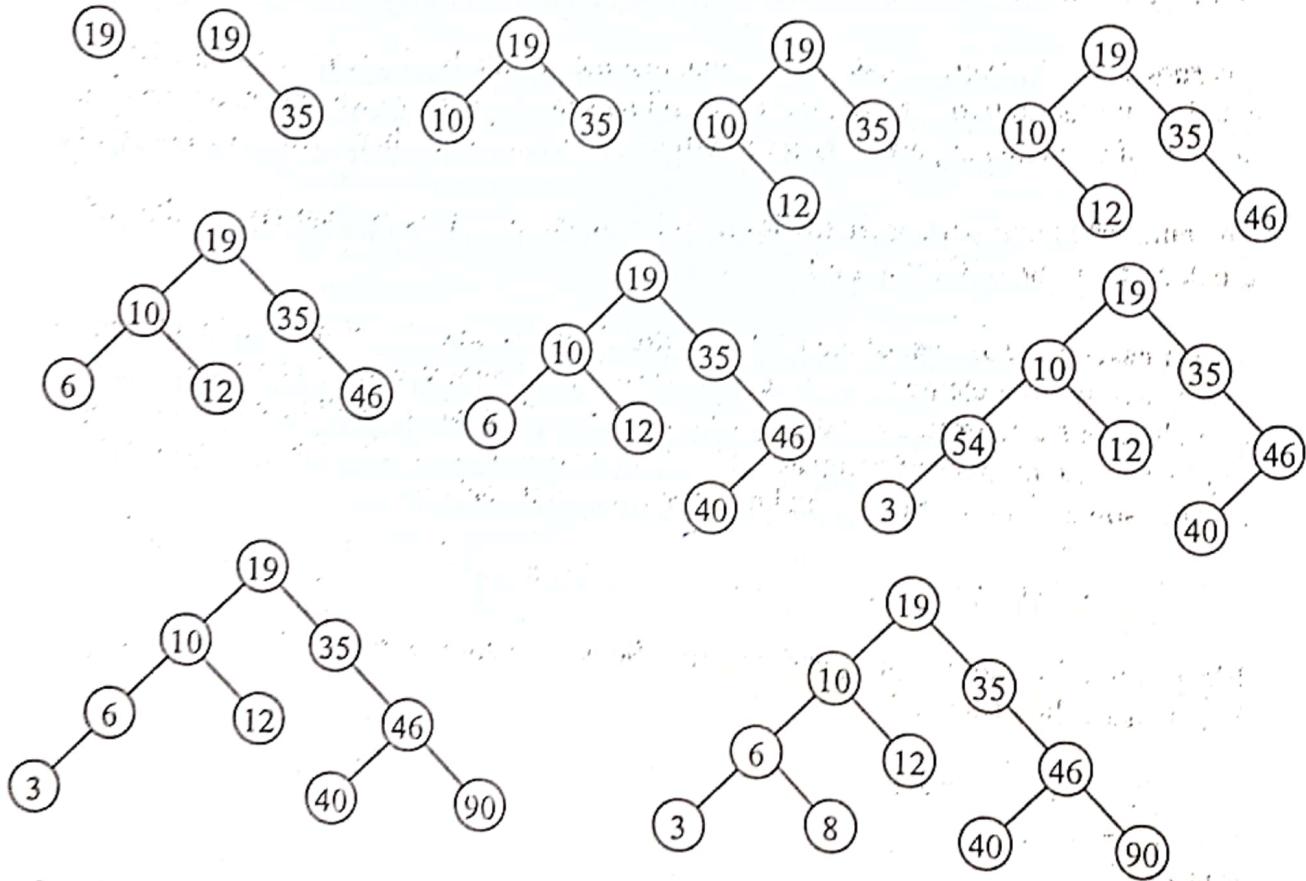
Let us take the list of elements and implement the binary tree sorting-

Let us take the list of elements and implement the binary tree sorting-

19, 35, 10, 12, 46, 40, 6, 90, 3, 8

Step 1- Creation of binary search tree

Binary tree is constructed based on the list of elements.



Step 2- Finding inorder traversal of constructed binary tree
Inorder traversal of this binary tree will be-

3, 6, 8, 10, 12, 19, 35, 40, 46 ,90

Now list of elements are in sorted order.

Program of binary search tree and it's inorder traversal can be seen in tree chapter.

Analysis-

Steps involved in binary tree sort are-

1. Insertion of element at proper place.
2. Inorder traversal of binary tree.

So we can say insertion of element is the basis of time requirement. Comparison of element will start from root then it will go to left or right child for comparing element and so on. So it will go maximum up to the depth of tree which will be $\log_2 n$. There is no need of comparison for the first inserted element. At any level l the number of elements will be 2^l . So now we can easily find the order with mathematical analysis, it will be of $O(n \log n)$.

This was the case where tree has both left and right subtrees. Suppose the binary tree has only left or right subtree, so first element will be root and has no need of comparison, second has a need of 2 comparisons, third will be in need of 3 comparisons and so on. So total number of comparisons will be-

$$0 + 2 + 3 + 4 + \dots + n$$

This is a form of arithmetic progression series. So applying the formula

$$\text{Sum} = n/2 [2a + (n-1)d]$$

$$= (n-1)/2 [2 \times 2 + (n-1) \times 1]$$

$$= (n-1)/2 [4 + n-1 - 1]$$

$$= (n-1)/2 \times (n+2)$$

which is of $O(n^2)$.

The main drawback of the binary tree sort is that it behaves worst if data is already in sorted order or in reverse order.

Heap Sort-

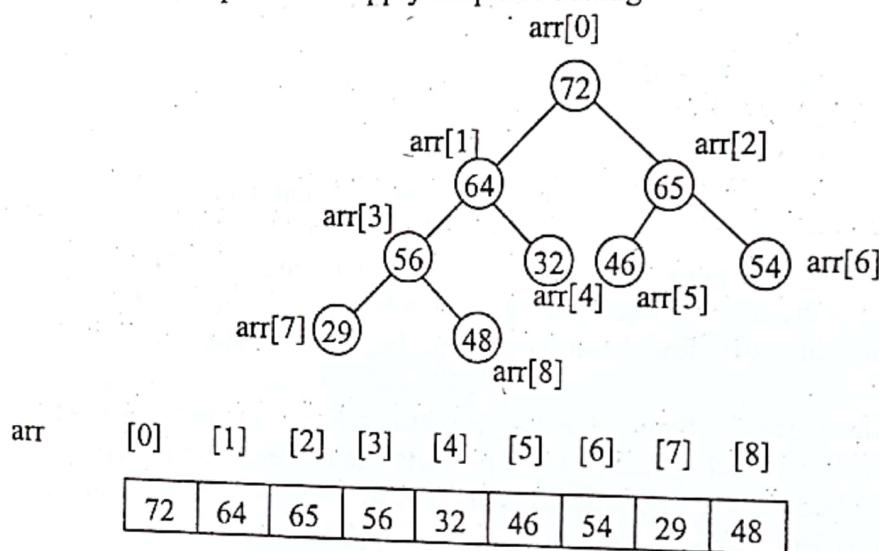
The elements of the heap tree are represented by an array. The root will be largest element of heap tree. Since it is maintained in the array, so the largest value should be the last element of array. For heap sorting we will keep on deleting the root till there is only one element in the tree.

Then the array which represented the heap tree will now contain sorted elements.

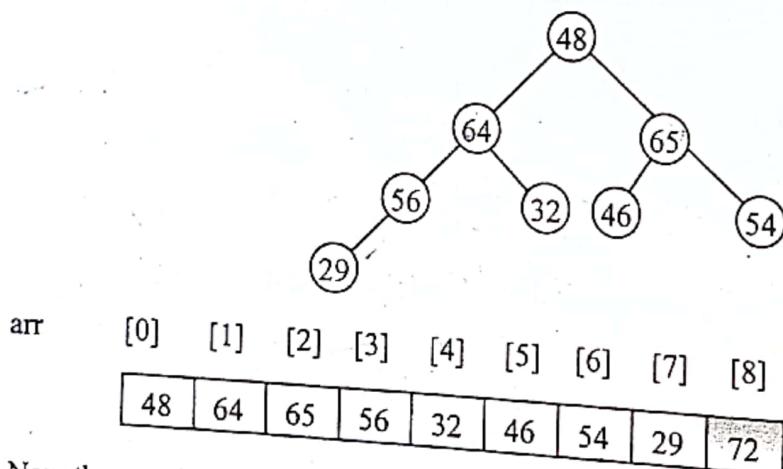
So we do following steps for heap sorting -

1. Replace the root with last node of heap tree.
2. Keep the last node (now root) at the proper position, means do the delete operation in heap tree but here deleted node is root

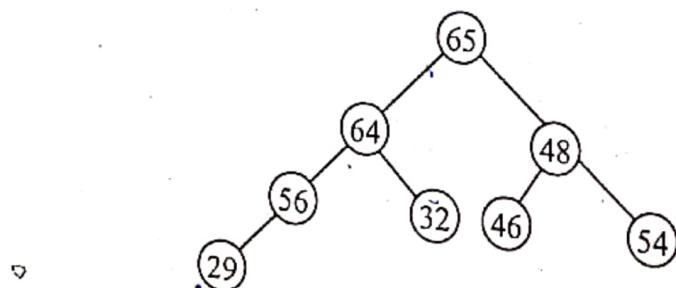
Let us take a heap tree and apply heap tree sorting-



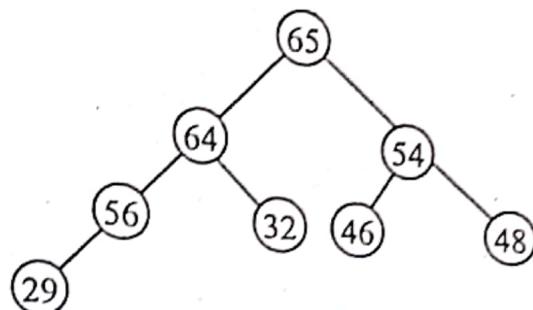
Step 1-



Now the root is at the position of last node and last node at the position of root.
 Here left and right child of 48 is 64 and 65. Both are greater than 48, but right child 65 is greater than left child 64, hence replace it with right child 65.



Here right child of 48 is 54, which is greater than 48, hence replace it with 54.



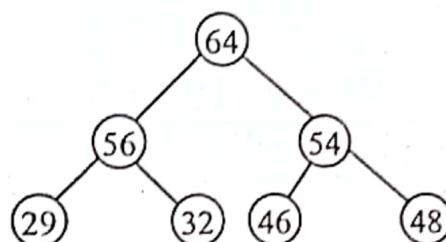
Now the elements of heap tree in array are as-

arr [0] [1] [2] [3] [4] [5] [6] [7] [8]

65	64	54	56	32	46	48	29	72
----	----	----	----	----	----	----	----	----

Now 29 is the last node. So replace it with root 65 and do the same operation.

Step 2-

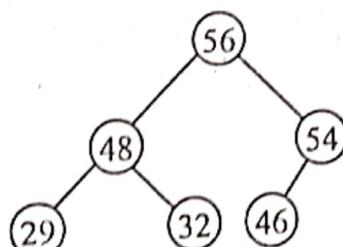


arr [0] [1] [2] [3] [4] [5] [6] [7] [8]

64	56	54	29	32	46	48	65	72
----	----	----	----	----	----	----	----	----

Similarly in next iteration

Step 3-

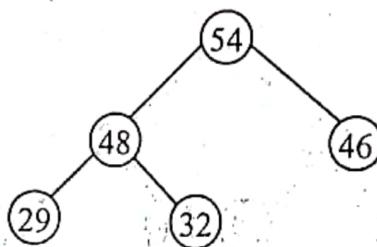


arr [0] [1] [2] [3] [4] [5] [6] [7] [8]

56	48	54	29	32	46	64	65	72
----	----	----	----	----	----	----	----	----

Step 4-

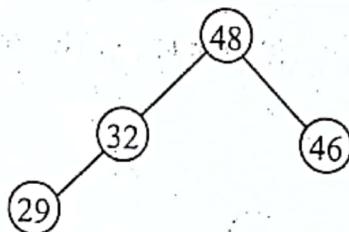
After the first pass, the array and the tree are as follows:



arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	54	48	46	29	32	56	64	65	72

Step 5-

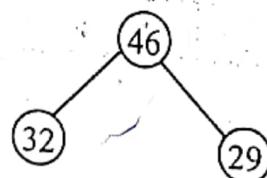
After the second pass, the array and the tree are as follows:



arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	48	32	46	29	54	56	64	65	72

Step 6-

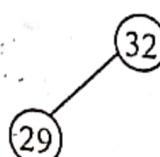
After the third pass, the array and the tree are as follows:



arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	46	32	29	48	54	56	64	65	72

Step 7-

After the fourth pass, the array and the tree are as follows:



arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	32	29	46	48	54	56	64	65	72

Step 8

(29)

arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	29	32	46	48	54	56	64	65	72

Now all the numbers are in sorted order.

```
/* Program of sorting through heapsort*/
#include <stdio.h>

int arr[20],n;

main()
{
    int i;
    printf("Enter number of elements : ");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d",&arr[i]);
    }
    printf("Entered list is :\n");
    display( );

    create_heap( );

    printf("Heap is :\n");
    display( );

    heap_sort( );
    printf("Sorted list is :\n");
    display( );
}/*End of main()*/
display()
{
    int i;
    for(i=0;i<n;i++)
        printf("%d ",arr[i]);
}
```

```

    printf("\n");
}/*End of display( )*/

create_heap()
{
    int i;
    for(i=0;i<n;i++)
        insert(arr[i],i);
}/*End of create_heap()*/

insert(int num,int loc)
{
    int par;
    while(loc>0)
    {
        par=(loc-1)/2;
        if(num<=arr[par])
        {
            arr[loc]=num;
            return;
        }
        arr[loc]=arr[par];
        loc=par;
    }/*End of while*/
    arr[0]=num;
}/*End of insert()*/

heap_sort()
{
    int last;
    for(last=n-1; last>0; last--)
        del_root(last);
}/*End of del_root*/

del_root(int last)
{
    int left,right,i,temp;
    i=0; /*Since every time we have to replace root with last*/
    /*Exchange last element with the root */
    temp=arr[i];
    arr[i]=arr[last];
    arr[last]=temp;

    left=2*i+1; /*left child of root*/
    right=2*i+2; /*right child of root*/
    while( right < last)
    {
}

```

```

        if( arr[i]>=arr[left] && arr[i]>=arr[right] )
            return;
        if( arr[right]<=arr[left] )
        {
            temp=arr[i];
            arr[i]=arr[left];
            arr[left]=temp;
            i=left;
        }
        else
        {
            temp=arr[i];
            arr[i]=arr[right];
            arr[right]=temp;
            i=right;
        }
        left=2*i+1;
        right=2*i+2;
    }/*End of while*/
    if( left==last-1 && arr[i]<arr[left] )/*right==last*/
    {
        temp=arr[i];
        arr[i]=arr[left];
        arr[left]=temp;
    }
}/*End of del_root*/

```

Analysis-

The process of heap sort is the creation of heap and place the node at proper position in heap tree. Here we can see every time one largest element will be at proper place and there will be no movement for already sorted element. Suppose heap tree is a complete binary tree then the sorting time will be of $O(n \log n)$ because the depth of tree will be $\log_2 n$. In the same worst case it will behave as of $O(n \log n)$. The main drawback of heap tree is extra time for creation of heap and too much movement of data. So it's not preferable for small list of data.
 For space requirement point of view it has no need of extra space except temporary variables.

Comparison-

Now we know the worst, average and best case behaviour of all sorting technique analyzed before. But one thing should be clear, no sorting technique is best, choice of algorithm should depend on the situation and the order of the data to be sorted. Now we can list out the best, average and worst case behaviour of sorting techniques in terms of O notation.

Sorting Technique	Best Case	Average Case	Worst Case
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$
Shell	-	-	-
Quick	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Radix	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
Binary Tree	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Heap	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Address Calculation	$O(n)$	$O(n \log n)$	$O(n^2)$

Exercise

1. Show all the passes using bubble sort with following list-
234, 54, 12, 76, 11, 87, 32, 12, 45, 67, 76 ✓
2. Show all the passes using insertion sort with following list-
13, 33, 27, 77, 12, 43, 10, 432, 112, 90 ✓
3. Show all the passes using selection sort with following list-
10, 22, 65, 223, 87, 343, 98, 244, 543, 22, 4 ✓
4. Show all the passes using shell sort with following list-
11, 54, 22, 124, 54, 77, 234, 11, 898, 43, 5, 2, 13 ✓
5. Show all the passes using quick sort with following list-
19, 123, 43, 78, 242, 98, 34, 75, 135, 87, 24 ✓
6. Show all the passes using radix sort with following list-
123, 76, 456, 244, 654, 865, 124, 987, 222, 890
7. Show all the passes using binary tree sort with following list-
12, 33, 11, 65, 22, 87, 76, 33, 34, 98, 30
8. Show all the passes using address calculation sort with following list-
67, 88, 99, 56, 54, 32, 67, 32, 86, 97, 88, 43, 93
9. Show all the passes using merge sort with following list-
194, 34, 12, 756, 54, 1, 88, 54, 897, 23, 96, 34 ✓
10. Write a program of heap sort using min heap.

11. Write a program to do the partition of list using quick sort for partition of list and use insertion sort for sublist.
12. Write a program to use any sorting technique to sort a data file of student records where key is Roll no.
13. Write a program to get the number of comparisons for any sorting technique with different lists.
14. Write a program of selection sort to insert the largest element at the end and compare it with actual selection sorting.

Chapter 5

Tree

We have studied linked list which is better data structure from memory point of view. But the main disadvantage with linked list is that it is a linear data structure. Every node has information of next node only. So the searching in linked list is sequential which is very slow and of $O(n)$. For searching any element in list, we search every element of the list before that element. Now we will study a data structure tree which represents the hierarchical data structure. It is very useful for information retrieval and searching in tree is also very fast and of $O(\log_2 n)$.

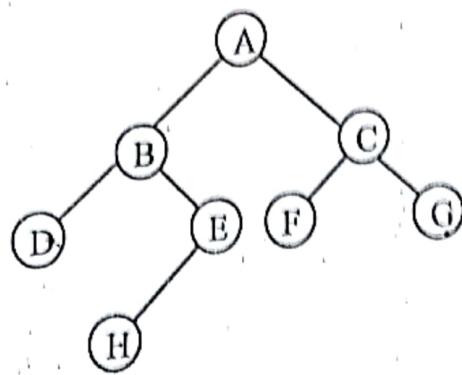
Binary Tree

In linked list, every node has two fields, first represents the data and second has information of next node. But in tree every node has three fields, first represents data, while second and third keep the information of left and right child of node. As in real life every tree starts with root, similarly here also the beginning node is called root. A binary tree can be defined as-

A binary tree is a finite set of nodes.

- (1) It is either empty or
- (2) It consists a node called root with two disjoint binary trees called left subtree and right subtree.

There are several ways to represent tree structure. We will use graph to represent tree in which every node will be represented by circle and a line from one node to other is called edge.



The left and right subtree also represent binary tree. Each element of tree is called a node of the tree. Here 'A' is beginning node of the binary tree, hence it is root. It has left successor 'B' and right successor 'C', these are also called left and right child of father.

Here A is father, B and C are left and right children. Similarly B is father of D and E, C is father of F and G. A node which has no child is called terminal or leaf node. Here nodes D, F, G, H are terminal nodes or leaf nodes.

- If two nodes are left or right child of same father then they are called siblings or brothers. Here B and C are siblings and the nodes other than terminal are called internal nodes. Here A, B, C, E are internal nodes.

The level of every node in binary tree can be defined as-

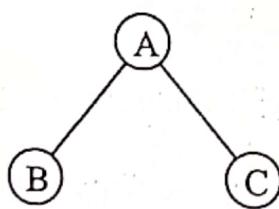
1. Root of tree is defined as level 0.
2. Level number of other nodes is 1 more than level number of its father node.

Here A has level 0, B and C have level 1.

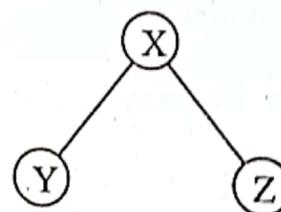
The depth of a tree is 1 more than the largest level number of tree. Depth is also sometimes known as height of the tree.

Here largest level number is 3 which is the level number of node H and depth of tree is $3+1 = 4$ (1 more than largest level).

Two trees are called similar if they have similar data structure and also said to be copies if they have same data at the corresponding nodes.

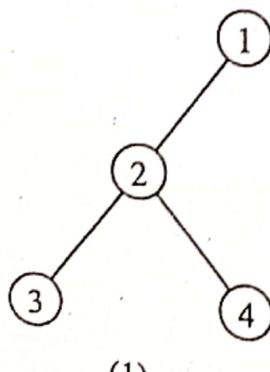


(1)

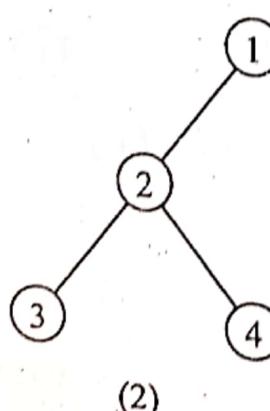


(2)

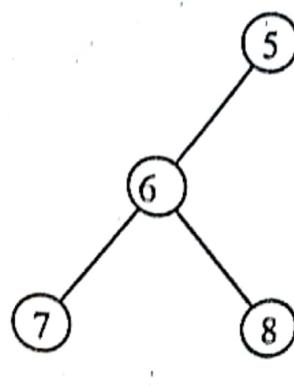
Here (1) and (2) are similar.



(1)



(2)

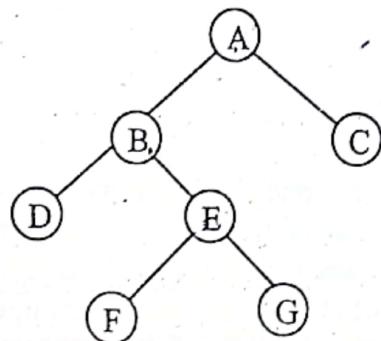


(3)

Here (1) and (2) are copies but (3) is not copy of (1) and (2).

Strictly Binary Tree

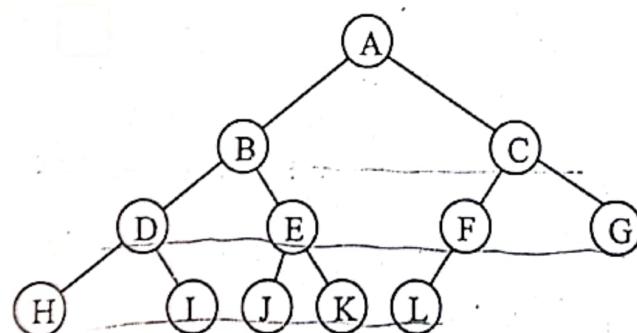
If every internal node (non terminal node) has its non empty left and right children then it is called strictly binary tree.



Here every internal node A, B, E has two non empty left and right child hence it is strictly binary tree.

Complete Binary Tree

We have already seen binary tree where each node has only left and right child but only two children maximum are allowed. So we can say at any level maximum number of nodes will be 2^l only. A complete binary tree is a binary tree where all the nodes have both children except last node. Let us take a complete binary tree.



The main advantage with this structure is that we can easily find the left and right child of any node and similarly parent of any child. Left child and right child of any node will be at position $2N$ and $2N+1$. Similarly, parent of any node N will be $\text{floor}(N/2)$. Here we can see parent of I will be $\text{parent}(9/2)=4$ i.e. D. Similarly left and right child of D will be $2*4=8$ th i.e H and $2*4+1=9$ th i.e. I. We can find out depth of tree as -

Suppose it has N nodes then the depth will be-
 $\text{depth} = \text{floor}(\log_2 N + 1)$

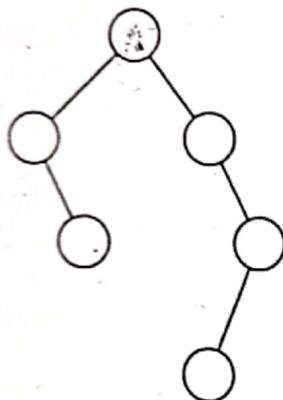
Suppose tree has 1000 nodes then depth will be-

$$\begin{aligned}\text{depth} &= \text{Floor}(\log_2 1000 + 1) \\ &= \text{Floor}(\log_2 2^{10} + 1) \\ &= 10 + 1 \\ &= 11\end{aligned}$$

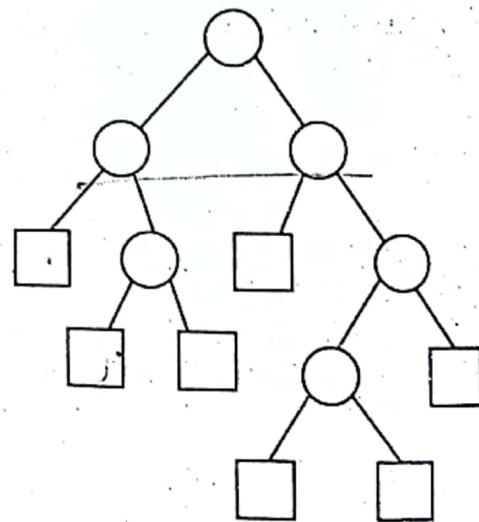
Extended Binary Tree

A binary tree is called extended binary tree if every node of tree has zero or two children. It is also said to be 2-tree.

Most of the nodes in binary tree have one child then we can add one more or two children and can extend it. So it can be converted to extended binary tree. It is very useful for representing algebraic expression, because in algebraic expression left and right child represent the operand and father of children represents the operator.



Binary tree



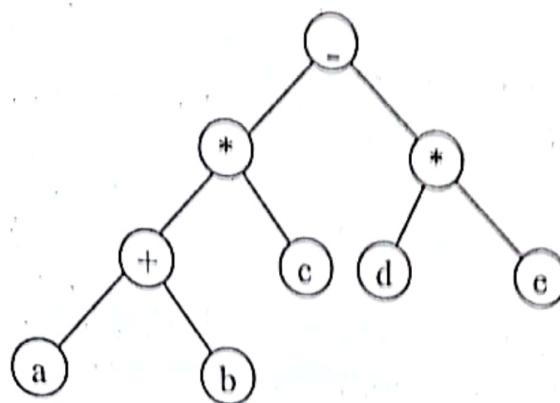
Extended Binary Tree

Here square represents the extended nodes in the original binary tree.

Algebraic Expression representation in tree

Any algebraic expression can be represented in binary tree in which left and right child represent operand of the expression and parent of that child represents the operator. Let us take an algebraic expression,

$$(a+b)*c-(d*e)$$



Every algebraic expression represents a unique tree.

Theorem

- (i) The maximum no of nodes at level i in a binary tree is 2^{i-1} where $i \geq 1$
- (ii) The maximum no of nodes in a binary tree of depth k is $2^k - 1$ where $k \geq 1$

Proof

- (i) The proof is by induction on i .

Induction Base

The root is the only node on level $i=1$.

Hence the maximum no of nodes on level $i=1$ is $2^0 = 2^{1-1}$

Induction Hypothesis

For all $j, 1 \leq j \leq i$

The maximum no of nodes on level j is 2^{j-1}

Induction Step

The maximum no of nodes on level $i-1$ is 2^{i-1} , i.e by induction hypothesis.

Since each node of binary tree has maximum two nodes(child). Hence the maximum no of nodes on level i is 2 times the maximum no of nodes on level $i-1$ or $2^{i-1} \times 2 = 2^i$.

Hence proved.

- (ii) The maximum no of nodes in a binary tree of depth k is

$$\sum_{i=1}^k 2^{i-1}$$

Since each node has maximum 2^{i-1} node at level i .

$$\sum_{i=1}^k 2^{i-1} = 1 + 2 + 2^2 + 2^3 + \dots + 2^{k-1}$$

(represents the geometric progression series)

$$= 1, (2^k - 1) / 2 - 1 = 2^k - 1$$

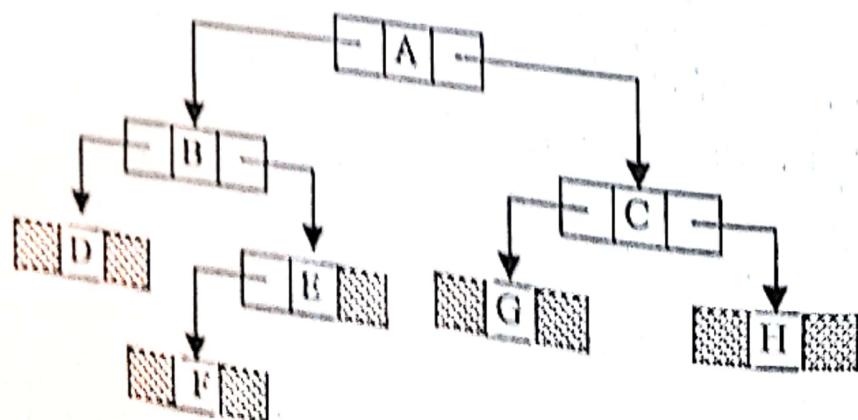
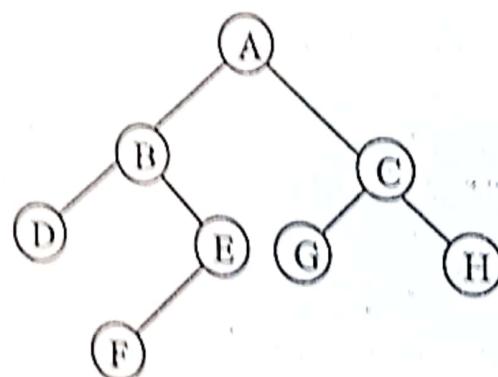
Representation of Binary Tree

As list is implemented in two ways, first in linked list and second through array.
 Similarly binary tree can also be implemented in two ways.

1. Linked representation
2. Array representation

Linked Representation

As in the linked list we take the structure for tree. In which we take three members. First member for data (this can be whole record), second member for left child and third member is for right child. Second and third members are structure pointers which point to the same structure as for tree node.



First we should declare structure for tree node.

```
struct node {
    char data;
    node * lchild;
    node * rchild;
}
```

Here first member data is for information field of node, second member is for left child of node which points to the structure itself, it contains the address of left child. If node has no left child then it should be NULL. Third member is for right child of node which also points to the structure itself, it contains the address of right child. If node has no right child it should be NULL.

Traversing in Binary Tree

In tree creation we take three parameters node, left child and right child, so traversing of binary tree means traversing of node, left subtree and right subtree. If root is denoted as N, left subtree as L and right subtree as R, then there will be six combinations of traversals NNL, NLR, LNR, LRN, RNL, RLN. But only three are standard, NLR(node-left-right), LNR(left-node-right) and LRN(left-right-node) traversal. We can see left subtree is always traversed before right subtree. NLR is called preorder LNR is inorder NNL is postorder.

LRN

These traversals are as -

Preorder(NLR Traversal)

1. Visit the root.
2. Traverse the left subtree of root in preorder.
3. Traverse the right subtree of root in preorder.

Inorder Traversal(LNR Traversal)

1. Traverse the left subtree of root in preorder.
2. Visit the root.
3. Traverse the right subtree of root in preorder.

Postorder(LRN Traversal)

1. Traverse the left subtree of root in postorder.
2. Traverse the right subtree of root in postorder.
3. Visit the root.

We will create the function for preorder traversal as-

```
preorder(ptr)
struct node *ptr;
{
    if (ptr!=NULL)
    {
        printf("%c",ptr→data);
        preorder(ptr→lchild); /* calling with address of left child */
        preorder(ptr→rchild); /* calling with address of right child */
    }
}
```

Here we are calling function recursively for left and right child.

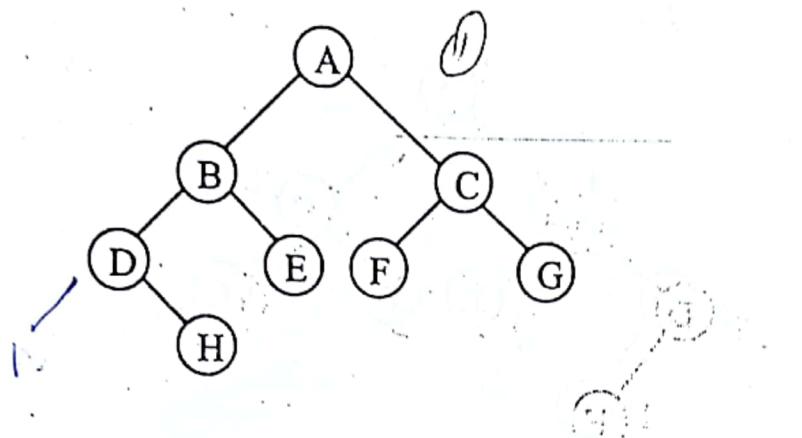
Similarly the function for inorder traversal will be as-

```
inorder(ptr)
struct node *ptr;
{
    if (ptr!=NULL)
    {
        inorder(ptr→lchild);
        printf("%c",ptr→data)
        inorder(ptr→rchild)
    }
}
```

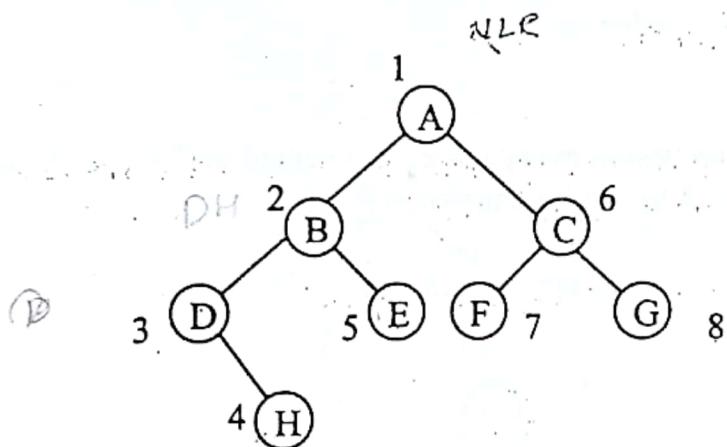
Function for postorder traversal is as-

```
postorder(ptr)
struct node *ptr;
{
    if (ptr!=NULL)
    {
        postorder(ptr→lchild);
        postorder(ptr→rchild);
        printf("%c",ptr→data);
    }
}
```

Let us take a binary tree and apply each traversal.



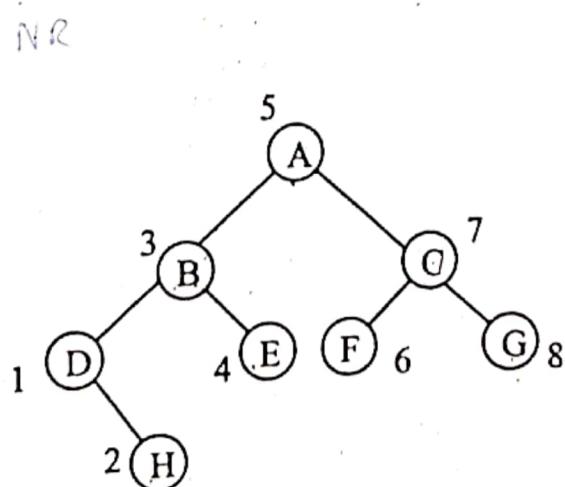
Preorder Traversal



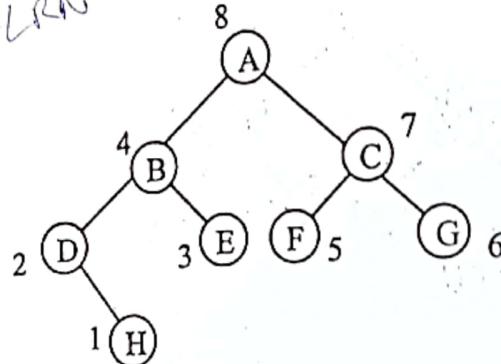
Here numbers represent the sequence of visited node. The nodes are visited in preorder as-

ABDHECFG

Inorder Traversal



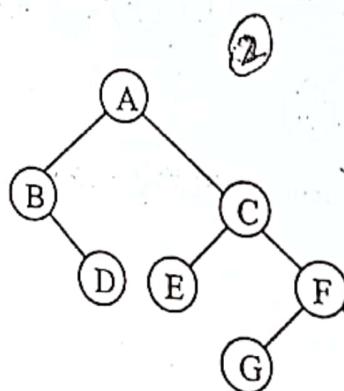
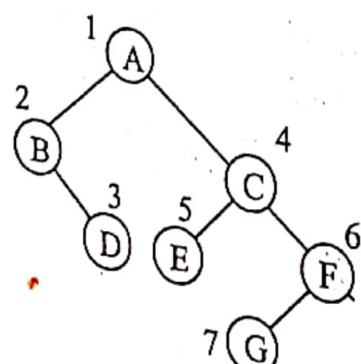
The nodes are visited in inorder as-
DHBEAFCG

Postorder Traversal

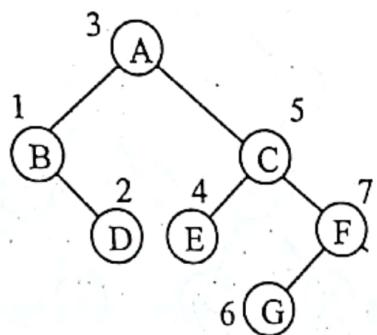
The nodes are visited in postorder as-
HDEBFGCA

Now we can see through recursion every node is traversed and it creates a copy of every call just as factorial program through recursion.

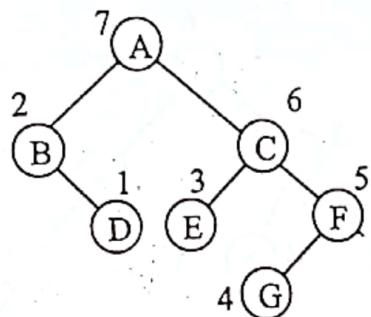
Let us take another binary tree and apply each traversal.

**Preorder Traversal**

The nodes are visited in inorder as -
ABDCEFG

Inorder Traversal

The nodes are visited in inorder as -
BDAECGF

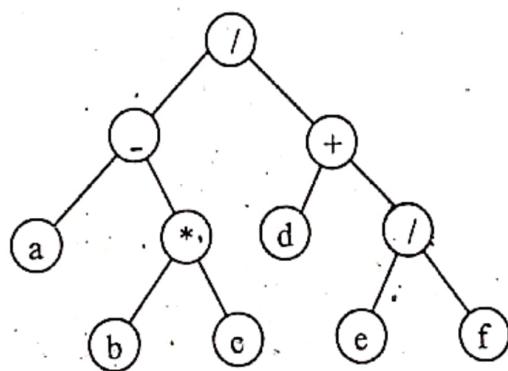
Postorder traversal

The nodes are visited in postorder as -
DBEGFCA

Let us take an algebraic expression in binary tree and apply each traversal.

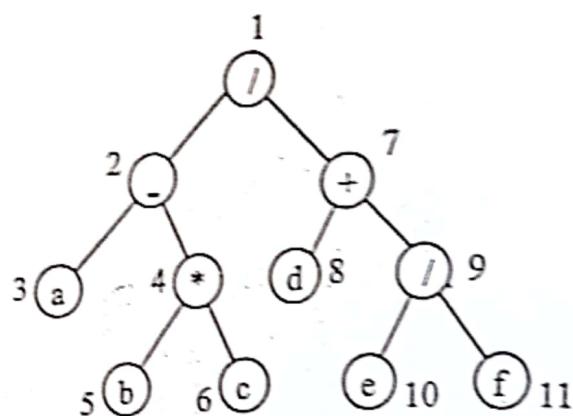
Algebraic expression

$$(a - b * c) / (d + e / f)$$



Binary tree representation of expression

Preorder Traversal

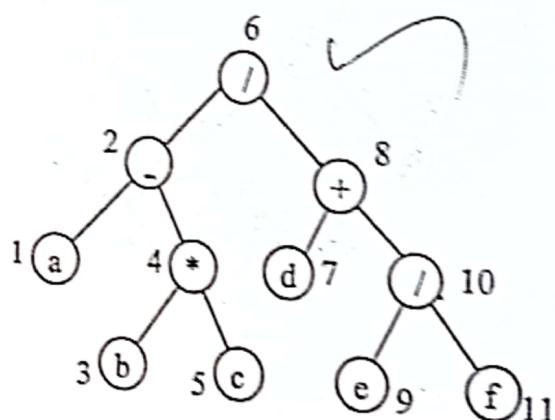


The nodes are visited in preorder as -

$/-a*b*c+d+e/f$

This is same as prefix of algebraic expression.

Inorder Traversal

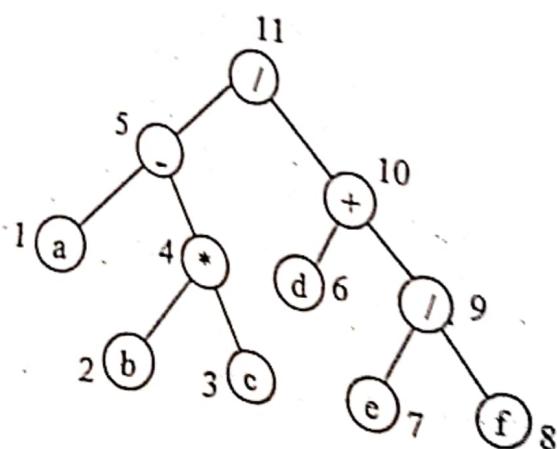


The nodes are visited in inorder as -

$a-b*c/d+e/f$

This is same as infix of algebraic expression.

Postorder Traversal



The nodes are visited in postorder as-

$abc^*def^{+/-}$

This is same as postfix of algebraic expression.

We can also make the tree if preorder and inorder traversals or postorder and inorder traversals are given.

Non recursive functions for traversals-

In non recursive functions we will use an array stack which will be used to keep the addresses of the nodes which are to be traversed.

Preorder traversal-

The procedure for traversing a tree in preorder non recursively is as-

Initially ptr contains the address of root.

1. Push the address of root node on the stack.

2. Pop an address from the stack.

3. If the popped address is not NULL

 Traverse the node

 Push right child of node on stack

 Push left child of node on stack.

4. Repeat steps 2,3 until the stack is not empty.

The non recursive function for preorder traversal is -

```
nrec_pre(struct node *ptr)
{
    stack[++top]=ptr;
    while(top!= -1 )
    {
        ptr=stack[top-];
        if(ptr!=NULL)
        {
            printf("%d ",ptr->info);
            stack[++top]=ptr->rchild;
            stack[++top]=ptr->lchild;
        }/*End of if*/
    }/*End of while */
}/*End of nrec_pre()*/
```

Inorder traversal-

The procedure for traversing a tree in inorder non recursively is as-

Initially ptr contains the address of root.

1. Repeat steps 2,3 while stack is not empty or ptr is not equal to NULL.

2. If ptr is not equal to NULL.

 push ptr on stack.

 ptr=ptr->lchild

3. If ptr is equal to NULL

 pop an address from stack.

 traverse the node at that address.

 ptr=ptr->rchild

The non recursive function for inorder traversal is –

```
nrec_in(struct node *ptr)
{
    while( top! = -1 || ptr!=NULL )
    {
        if(ptr!=NULL)
        {
            stack[++top]=ptr;
            ptr=ptr->lchild;
        }
        else
        {
            ptr=stack[top--];
            printf("%d ",ptr->info);
            ptr=ptr->rchild;
        }
    }/*End of while*/
}/*End of nrec_in()*/
```

Postorder traversal-

The procedure for traversing a tree in postorder non recursively is as-

Here we will take an array stack which will keep the addresses of the nodes. In this traversal we have to take another array flag corresponding to the stack array. The elements of this array will be 1 or -1. We have also taken a new variable top_prev in the function, which we will use to save the previous value of top when an item is popped from the stack.

Steps-

Initially ptr has the address of root.

1. First push NULL on the stack

2. Repeat steps 3, 4, 5 while ptr !=NULL

3. Move on the leftmost path rooted at ptr, and all the nodes which come on this path are to be pushed on stack and the value of flag is made 1 for these nodes. Besides putting these nodes on the stack we also check whether the node has right child or not, if the

node has right child then that right child is also pushed on stack and the value of flag is made -1 for these type of nodes.

4. Save the value of top in top_prev, and then pop an address from the stack and assign that address to ptr.

5. Repeat following steps while flag[top_prev] == 1

Traverse the node whose address is ptr.

Pop another node from the stack.

As soon as we get the value of flag[top_prev] = -1 we will move again to step 3, because we have got a right subtree of a node which again has to be traversed in postorder.

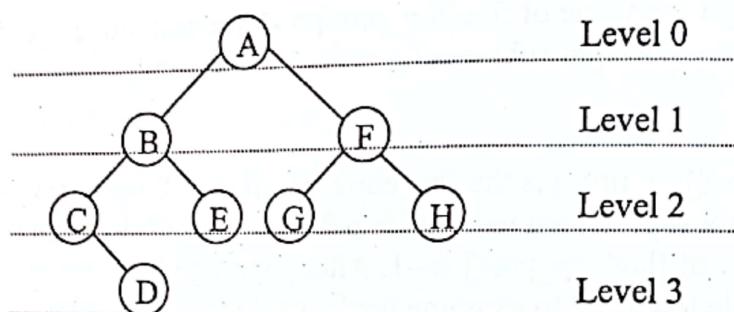
In the step 5, flag[top_prev] is the flag corresponding to the pointer ptr, i.e if the ptr was a node in the leftmost path then value of this flag is 1 and if ptr was a right child of the node then value of flag[top_prev] is -1. After popping ptr, value of top is changed so we need the variable top_prev to examine the flag value corresponding to ptr. The function for non recursive postorder traversal of a tree is as-

```
nrec_post(struct node *ptr)
{
    int flag[MAX];
    int top_prev;
    stack[++top]=NULL;
    do
    {
        while(ptr!=NULL)
        {
            stack[++top]=ptr;
            flag[top]=1;
            if(ptr->rchild!=NULL)
            {
                stack[++top]=ptr->rchild;
                flag[top]=-1;
            }
            ptr=ptr->lchild;
        }
        top_prev=top;
        ptr=stack[top--];
        while( flag[top_prev]==1)
        {
            printf("%d ",ptr->info);
            top_prev=top;
            ptr=stack[top--];
        }
    }while(ptr!=NULL);
}/*End of nrec_post()*/
```

Level order traversal

In level order traversal, we traverse the nodes according to their levels. We start traversing with the level 0, then we traverse all the nodes of level 1, and then all nodes of level 2 and so on. We traverse the nodes of a particular level from left to right.

Let us take a tree and see its level order traversal.



The level order traversal of this tree will be -

A B F C E G H D

We can easily write the function for level order traversal non recursively with the help of a queue which will keep the addresses of the nodes.

```

level_trav(struct node *ptr)
{
    struct node *que[MAX];
    int front=-1,rear=-1;
    front++;
    rear++;
    que[rear]=ptr;
    while(front<=rear) /*Loop until queue is not empty*/
    {
        ptr=que[front++];
        if(ptr!=NULL)
        {
            printf("%d ",ptr->info);
            que[++rear]=ptr->lchild;
            que[++rear]=ptr->rchild;
        }
    }/*End of while */
}/*End of level_trav()*/
  
```

Creation of binary tree from preorder and inorder traversals -

Preorder ABDHECFG
Inorder DHBEAFCG

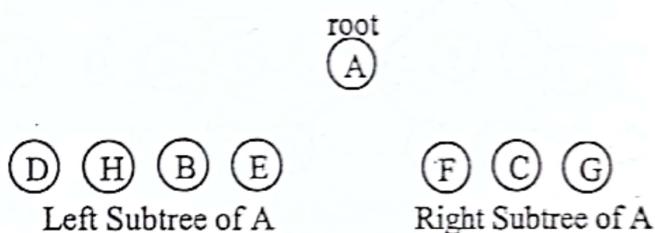
1. In preorder traversal root is the first node. Hence A is the root of the binary tree root



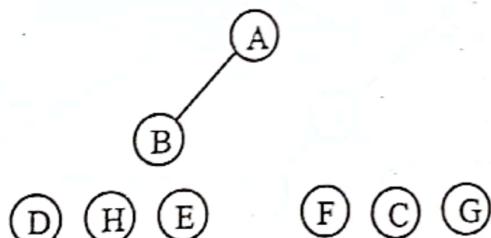
2. Now we can find the node of left subtree and right subtree with inorder sequence. Nodes which are in the left side of root in inorder are nodes of left subtree and nodes which are in the right side of root in inorder are nodes of right subtree.

Nodes of left subtree – DHBE

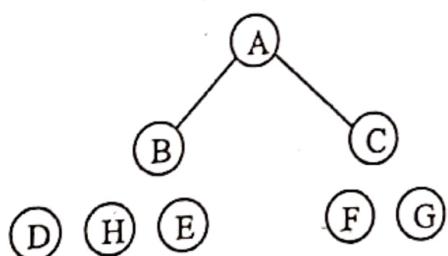
Nodes of right subtree – FCG



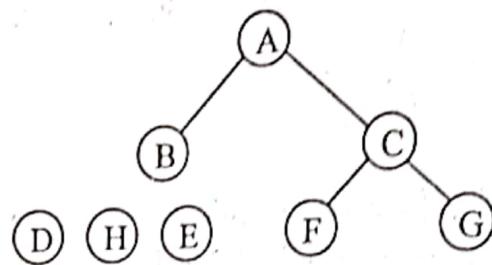
3. The left child of the root F will be the first node in preorder traversal after root A. Hence B is the left child of A.



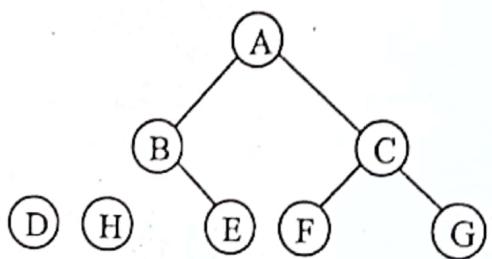
4. The right child of root A will be the first node after nodes of left subtree in preorder traversal. Hence C is the right child of A.



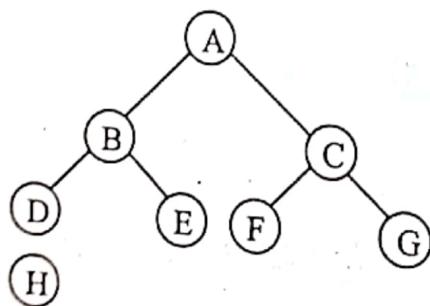
5. In inorder sequence, F is on the left of C and G is on the right side of C. Hence F will be in left subtree of C and G in right subtree of C.



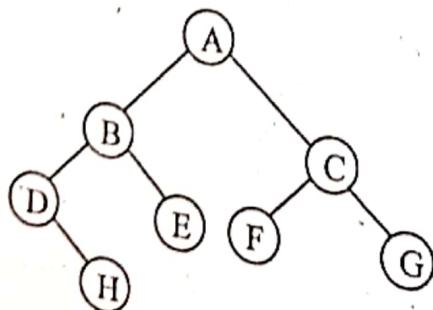
6. Now in inorder sequence, D and H are on the left side of B and E is on the right side of B. So D and H will form left subtree of B and E will be in right subtree of B.



7. In preorder sequence, root is traversed before left and right subtrees. In preorder traversal of this tree, D is coming before H, hence D is the root of left subtree of B and H can be either in left or right subtree of D.



8. To find out whether H is in left or right subtree of D, we look at inorder traversal. Since H is in right side of D, hence it will be in right subtree of D.



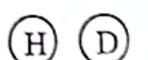
Creation of tree from postorder and inorder traversals

Postorder HDIEBJFKLGC
Inorder HDBIEAFJCKGL

1. In postorder traversal root is the last node. Hence A is the root of the binary tree root



2. From inorder traversal we can find out left and right subtrees of root.

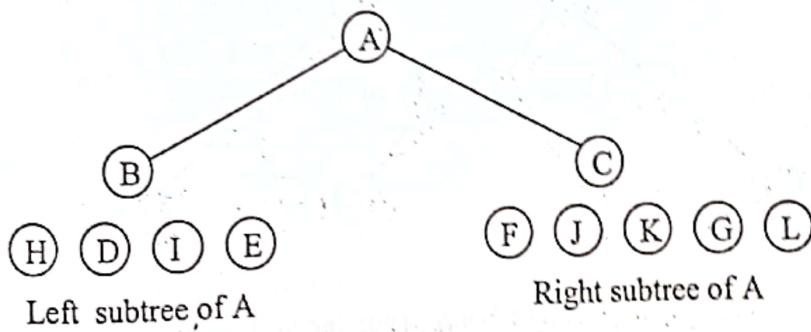


Left subtree of A

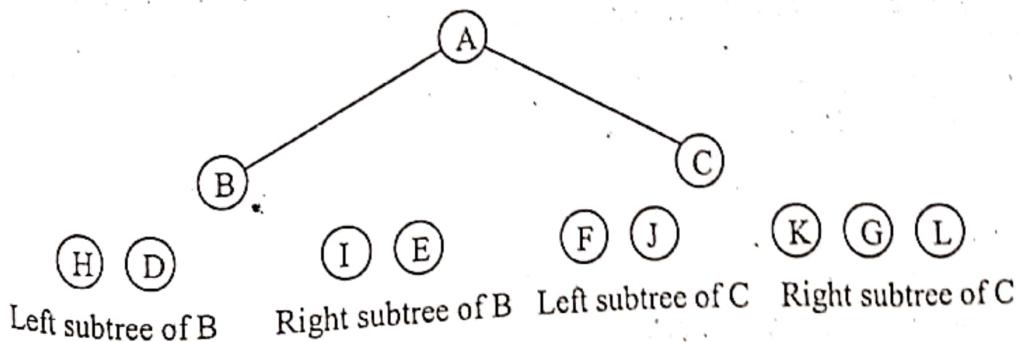


Right subtree of A

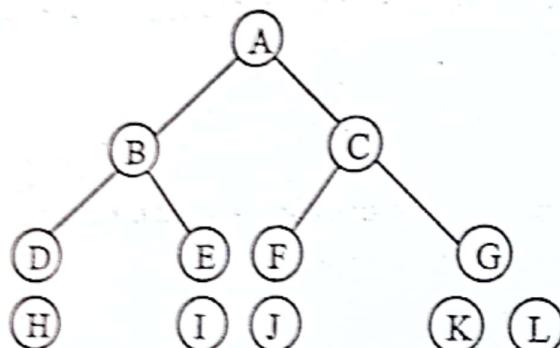
3. Now right child of A will be the node which comes just before node A. Hence C is right child of node A. Left child of A will be the first node before nodes of right subtree in postorder traversal. Hence B is left child of A.



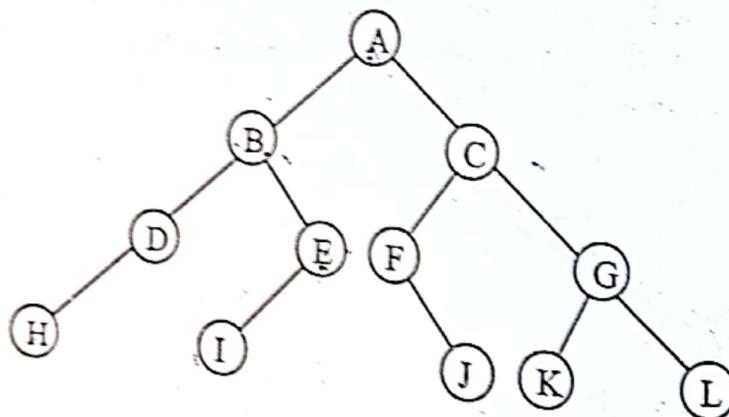
4. Now look at C in the inorder traversal. F and J are to the left of C and K, G, L are to the right of it, hence F, J form left subtree of C and K, G, L form right subtree of C. Now look at B in inorder traversal. H, D are to the left of B and I, E are to the right of B hence H, D form left subtree of B and I, E form right subtree of B.



5. Now look at postorder traversal, node just before B is E, hence E is right child of B and node just before C is G, hence G is right child of C. D is the first node before the nodes of right subtree of B hence D is right child of B; F is the first node before nodes of right subtree of C. Hence F is left child of C.



6. Now look at inorder traversal, H is to the left of D hence it is left child of D, I is to the left of E, hence it is left child of E. J is to the right of F hence it is right child of F. K is to the left of G and L to the right of G hence K is left child of G and L is right child of G.



Shortcut method of creating the tree from preorder and inorder traversal

Creation of tree by this method is very simple and quick. In preorder traversal, scan the nodes one by one and keep them inserting in the tree. In inorder traversal, put a cross mark over the node which has been inserted. To insert a node in its proper position in the tree, we will look at that node in the inorder traversal and insert it according to its position with respect to the crossed nodes.

Preorder A B D G H E I C F J K
 Inorder G D H B E I A C J F K ✓

Step 1 Insert A

Pre A B D G H E I C F J K
 In G D H B E I A C J F K
 A is the first node in preorder traversal hence it is the root of the tree.

Step 2 Insert B
 Pre A B D G H E I C F J K
 In G D H B E I A C J F K

Since B is to left of A in inorder traversal hence it is left child of A.

Step 3 Insert D
 Pre A B D G H E I C F J K
 In G D H B E I A C J F K

Since D is to the left of B in inorder traversal hence D is left child of B.

Step 4 Insert G
 Pre A B D G H E I C F J K
 In G D H B E I A C J F K

Since G is to the left of D in inorder traversal hence G is left child of D.

Step 5 Insert H
 Pre A B D G H E I C F J K
 In G D H B E I A C J F K

Since H is to the left of B and right of D in inorder traversal hence H is right child of D.

Step 6 Insert E
 Pre A B D G H E I C F J K
 In G D H B E I A C J F K

Since E is to the left of A and right of B in inorder traversal hence E is right child of B.

Step 7 Insert I
 Pre A B D G H E I C F J K
 In G D H B E I A C J F K

Since I is to the left of A and right of E in inorder traversal hence I is right child of E.

Step 8 Insert C
 Pre A B D G H E I C F J K
 In G D H B E I A C J F K

Since C is to the right of A in inorder traversal hence C is right child of A.

Step 9 Insert F
 Pre A B D G H E I C F J K
 In G D H B E I A C J F K

Since F is to the right of C in inorder traversal hence F is right child of C.

Step 10 Insert J
 Pre A B D G H E I C F J K
 In G D H B E I A C J F K

Since J is to the right of C and to left of F in inorder traversal hence J is left child of F.

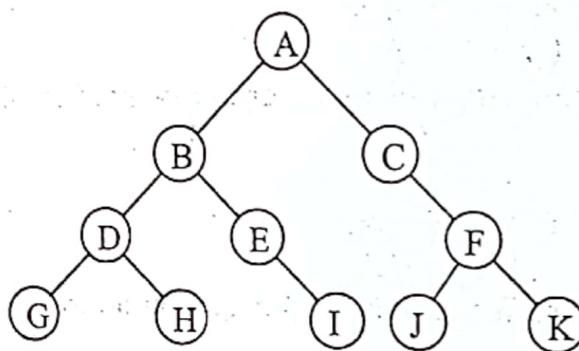
Step 11 Insert K

Pre A B D G H E I C F J K

In ~~G D A B E Y A & J F~~ K
Since K is to the right of F in inorder traversal hence K is right child of F.

Draw the tree with each step and when all the nodes have been inserted, you will get the final tree. Here we have shown all the steps, but while creating tree on your own just write the preorder and inorder traversal once and keep on scanning the nodes in preorder and cross the inserted nodes in inorder and simultaneously construct your tree by inserting nodes.

The tree for the above preorder and inorder traversal will be-

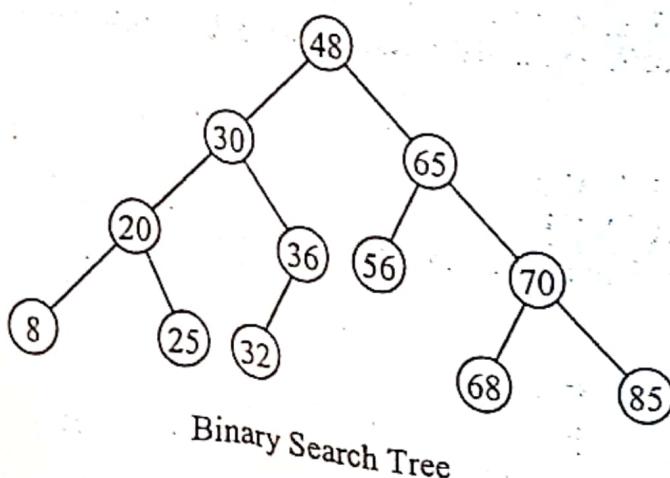


Creation of tree from postorder traversal and inorder traversal by shortcut method is same as creation of tree from inorder and preorder. The only difference is that here we will start scanning the nodes from the right side means the last node in the postorder traversal will be inserted first and first node will be inserted in the last.

Binary Search Tree

A binary search tree is a binary tree in which each node has value greater than every node of left subtree and less than every node of right subtree.

Binary search tree is very useful data structure in which item can be searched in $O(\log_2 N)$ where N is the number of nodes.



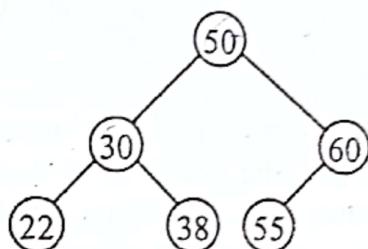
Search and Insertion Operations

Searching and insertion is one time operation in binary search tree because before insertion of any element we first search the exact place for inserting.

Suppose a data is given and we want to search and insert that data. This can be done as -
 1. Compare data with data of root node
 (a) If data < data of node then compare with data of left child node.
 (b) if data > data of node then compare with data of right child node

At last, we find the node which has same value as data or we will reach the exact place where we will insert the node.

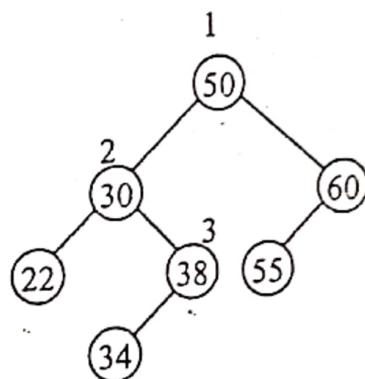
Let us take a binary search tree.



We want to insert a node which has value 34.

Steps-

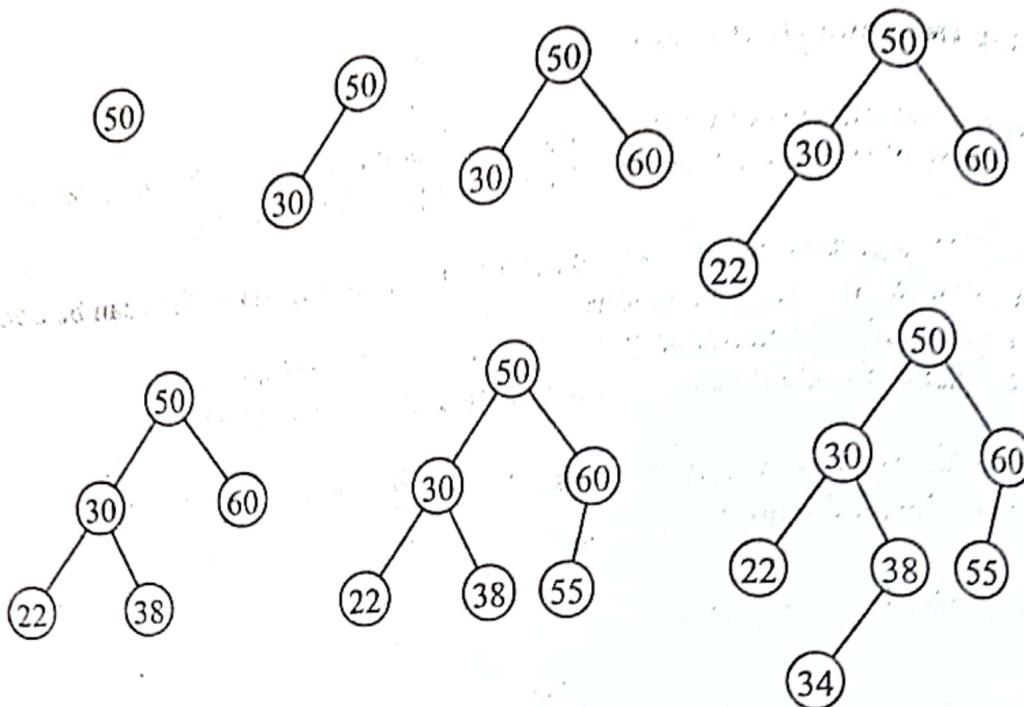
1. Compare 34 with 50. $34 < 50$ so compare with left child of 50 which is 30.
2. Compare 34 with 30. $34 > 30$ so compare with right child of 30 which is 38.
3. Compare 34 with 38. $34 < 38$ so compare with left child of 38 but 38 has no left child so this is the exact place to insert 34. We will insert 34 as a left child of 38.



Here numbers show the path for insertion of new node.

Let us take seven numbers and insert them in binary tree which is initially empty.

50, 30, 60, 22, 38, 55, 34



Creation of find()

This function `find()` is used both in deletion and insertion of nodes in binary search tree. The arguments to this function are the item to be found, address of location pointer and address of parent pointer. Note that we are passing here parent and location pointers by reference, why are we doing so. Generally, we pass parameters by reference when we want the function to return more than one value to the program. Here we want the `find()` function to return the address of item and address of parent also. So we are passing address of pointers to the `find()` function and hence the last 2 arguments to this function are a pointer to pointer. So arguments `loc` and `par` are pointer to pointer.

After calling `find()` function location and parent pointer will be assigned with some values and based on these values we can draw some conclusions.

- (a) If `*loc==NULL` and `*par==NULL` means tree is empty.
- (b) If `*loc==NULL` and `*par!=NULL` means item is not present in tree.
- (c) If `*loc!=NULL` means item is present in tree.
 - If `*loc!=NULL` and `*par==NULL` item is at root.
 - If `*loc!=NULL` and `*par!=NULL` item is at some other place in the tree.

Insertion in Binary Search tree

Insertion in Binary search tree requires the address of parent node where we will insert the new node. So we call the function `find()` as -
`find(item,&parent,&location);`

After calling this function, we get the address of parent node and address of item in location pointer, if item already exists in tree. Since we are not allowing duplicate values,

so first we check this condition-

```
if(location!=NULL)
{
    printf("Item already present");
    return;
}
```

If location is NULL value means added item does not exist in tree. So we can add this in tree.

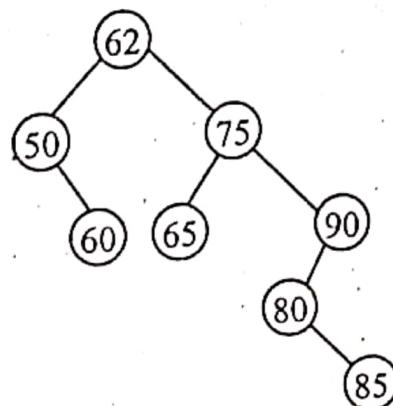
Now we will insert the value at proper position. First we will check that element will be added as a root if tree is empty, otherwise it will be added as left terminal node or right terminal node as-

```
tmp->info=item;
tmp->lchild=NULL;
tmp->rchild=NULL;
if(parent == NULL)
    root=tmp;
else
    if(item<parent->info)
        parent->lchild=tmp;
    else
        parent->rchild=tmp;
```

Since it will be terminal node so left and right child of new node will be NULL. If new node will be added as a root then we assign the address of new node to the root. If new node will be added as left child then we assign the new node address to the lchild part of parent node. If new node will be added as right child then we assign the address of new node to the rchild part of parent node.

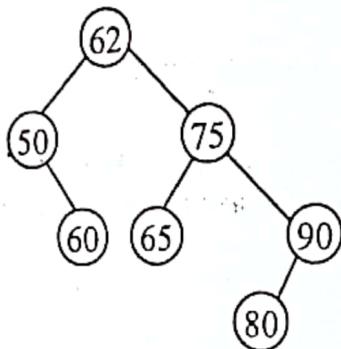
Deletion operation

Let us take binary search tree and apply delete operation



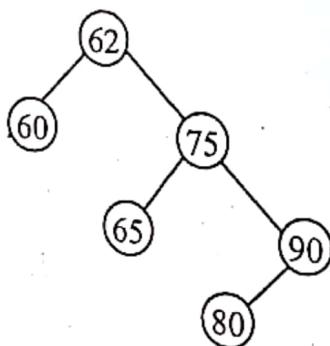
Now we want to delete the item 85. Since 85 has no children. So we can delete it simply by giving NULL value to its parent's right pointer. Here 80 is the parent of 85 and 85 is the right child of 80, so after deleting 85, right pointer of 80 will have NULL value.

Now the binary search tree will be as –



Now we want to delete the item 50. Since 50 has only one child. So we can delete it simply by giving the address of right child to its parent left pointer.
Here 62 is the parent of 50 and 60 is the right child of 50.

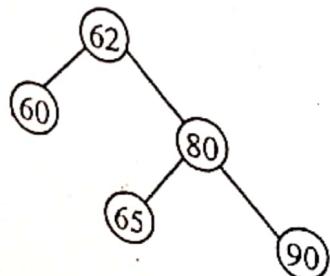
Now the binary search tree will be as –



Now we want to delete the item 75. Since 75 has two children, so first we delete the item which is inorder successor of 75. Here 80 is the inorder successor of 75. We delete 80 by simply giving the NULL value to its parent's left pointer. Here 90 is the parent and 80 is the left child of 90.

After that we replace the item 75 with item 80. We give the address of the left and right pointers of 75 to left and right pointers of 80 and address of 80 to the right pointer of parent 75 which is 62.

Now the binary search tree will be –



Deletion in Binary search tree requires the address of item to be deleted and address of parent node. Then only we can delete the item. So we call the function find() as-
find(item,&parent,&location);

After calling this function, we get the address of item in location pointer and address of parent node. If we get NULL value in location pointer then that item doesn't exist in tree.

There can be 5 possibilities while performing delete operation in binary search tree-

1. There are no nodes in the tree i.e the tree is empty.
2. Node to be deleted is not present in the tree.
3. Node to be deleted is leaf node i.e it has no children.
4. Node to be deleted has only one child.
5. Node to be deleted has two children.

We'll check for these possibilities in our del() function and take appropriate action.

1. First we check the condition for tree empty as-

```
if(root==NULL)
{
    printf("Tree is empty");
    return;
}
```

2. Now we have a need to get the address of item to be deleted and address of it's parent node. So we call find() function as-

```
find(item,&parent,&location);
```

After calling this function, we get the address of item in location pointer and address of parent node in parent pointer. If we get NULL value in location pointer then that item doesn't exist in tree. So we check for item existence as-

```
if(location==NULL)
{
    printf("Item not present in tree");
    return;
}
```

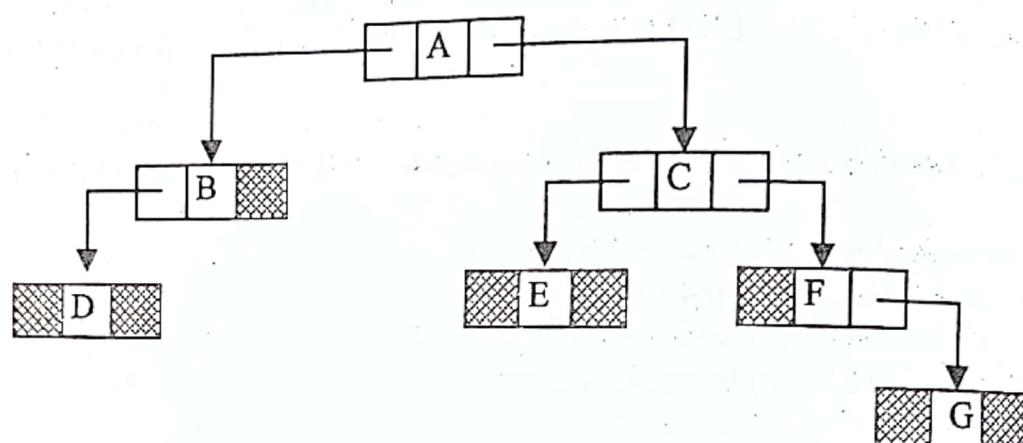
Now we have address of node to be deleted and address of it's parent node. Now deletion can be for three cases-

1. Node to be deleted is leaf node i.e it has no children.

2. Node to be deleted has only one child.

3. Node to be deleted has two children.

We can see these cases below-



Tree T1

Case 1-

If the node is a leaf node i.e it has no children, then its left and right pointers will contain NULL. Here D, E and G are leaf nodes. This will be checked as-

```
if(location->lchild==NULL && location->rchild==NULL)
    case_a(parent,location);
```

Case 2-

If the node has only one child, that child can be either left child or right child. Here node B has only left child so its left pointer is not NULL and right pointer is NULL. So we have a need to check for both conditions as-

```
if(location->lchild!=NULL && location->rchild==NULL)
    case_b(parent,location);
```

Node F has only right child, so its right pointer is not NULL and left pointer is NULL.

```
if(location->lchild==NULL && location->rchild!=NULL)
    case_b(parent,location);
```

Case 3-

Now we have a need to consider the condition when the node has both left and right child. This can be checked as-

```
if(location->lchild!=NULL && location->rchild!=NULL)
    case_c(parent,location);
```

Now we will see the creation of function `case_a()`, `case_b()` and `case_c()`.

Creation of function `case_a()`-

Deleting a leaf node is very simple. First we check if node to be deleted is root then we assign NULL value to root, otherwise if node is left child then NULL will be assigned to lchild part of parent node and if it's right child then NULL will be assigned to rchild part of parent node.

```
if(par==NULL) /*item to be deleted is root node*/
    root=NULL;
else
    if(loc==par->lchild)
        par->lchild=NULL;
    else
        par->rchild=NULL;
```

Suppose we want to delete the node G in tree T1 then we assign NULL to right pointer of its parent F, since G is right child of F. Similarly for deleting node D we assign NULL to left pointer of B since it is left child of B.

Creation of function `case_b()`-

If item to be deleted has only one child, we can delete it by giving address of its child to its parent. First we check that node to be deleted has left child or right child. If it has only left child then we store the address of left child and if it has only right child then we store the address of it's right child in pointer variable child as-

```
if(loc->lchild!=NULL)
    child=loc->lchild;
else
    child=loc->rchild;
```

Now we check that if node to be deleted is root node then we assign value of pointer variable child to root node. So child of deleted node(root) will become the root node. Otherwise, we check that node to be deleted is left child or right child of its parent. If it is left child then we assign the value of child pointer variable to lchild part of its parent. So child of node to be deleted will become the left child of it's parent. If it is right child then we assign the value of child pointer variable to rchild part of its parent. So child of node to be deleted will become the right child of its parent.

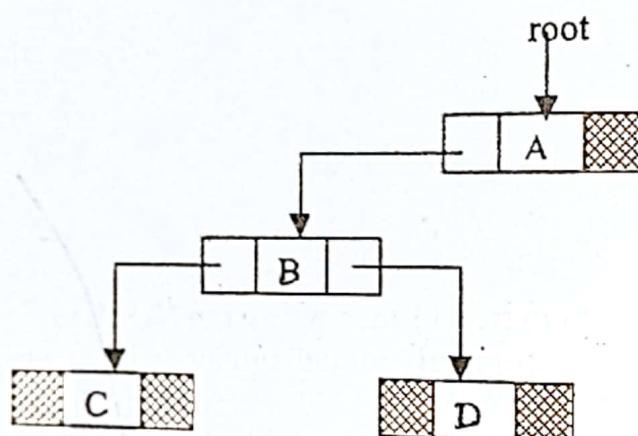
```

if(par == NULL)
    root = child,
else
    if( loc == par->lchild)
        par->lchild = child;
    else
        par->rchild = child;

```

In tree T1, for deleting B, put the address of its child G in the right pointer of its parent C since F is right child of C.

As in case_a() we had treated the condition of node to be deleted being the root node, here also we'll treat it separately, since the value of root will change.

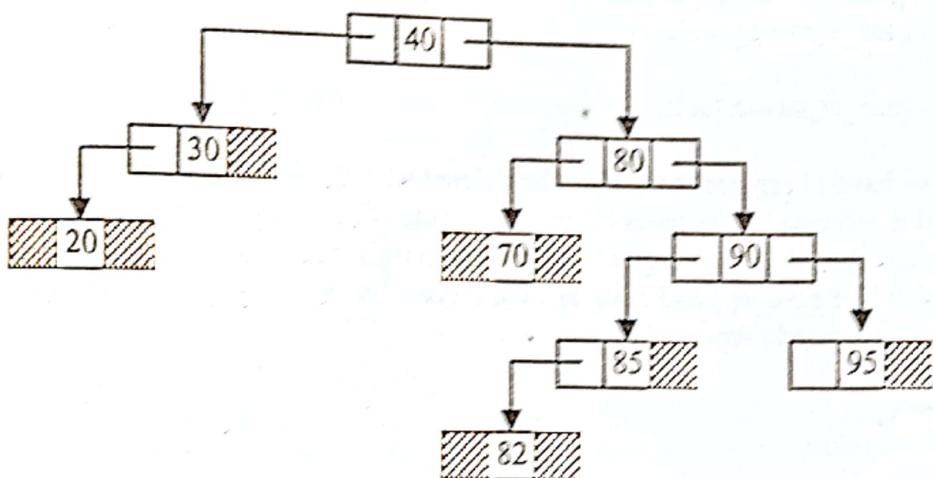


To delete A which is the root node we will assign the value of its child to the root pointer. So now child of the deleted node will become the root node. Hence, after deleting A, root pointer will point to node B.

Creation of function case_c()-

If the node to be deleted has two children then first delete the inorder successor of node and then replace the deleted node with the inorder successor.
Inorder successor of a node is the node which comes just after that node in the inorder traversal of the tree. Inorder successor of any node will be the leftmost node of the right subtree of that node. Inorder successor can have either no child or only one right child. It can't have left child because then that left child will become the inorder successor. Hence inorder successor can't have two children.

So to delete inorder successor we again have to call case_a() or case_b() depending on whether it has no child or one child. For deleting a node by case_a() or case_b() we need to send the address of that node and address of its parent. So to delete inorder successor we need to find out its address and address of its parent. Let us take a tree-



Inorder traversal of this tree is –

20 30 40 70 80 82 85 90 95

Suppose we have to delete node 80. Then we have to find address of its inorder successor and parent of inorder successor. From the inorder traversal we can see that inorder successor of 80 is 82. Now we'll see how we'll find out this in our program.

We know that inorder successor of a node is the leftmost node in the right subtree of that node. So initially we take two pointers ptr and ptrsave. Initialize ptrsave with the address of node to be deleted and ptr with the right child of node because we want to traverse right subtree of that node. To find out leftmost node of this right subtree, we will keep on traversing to left until we reach a node which has NULL in its left pointer, means there is no node left to it and we have reached the leftmost node.

```

ptrsave=loc;
ptr=loc->rchild;
while(ptr->lchild!=NULL)
{
    ptrsave=ptr;
    ptr=ptr->lchild;
}
  
```

When this loop will terminate, ptr will contain address of inorder successor and ptrsave will contain address of parent of inorder successor.

```

suc=ptr;
parsuc=ptrsave;
  
```

Now we have to delete successor from the tree so we'll call case_a() or case_b() depending on whether node has no child or one child.

```

if(suc->lchild==NULL && suc->rchild==NULL)
    case_a(parsuc,suc);
else
    case_b(parsuc,suc);

```

Now we have to replace the node to be deleted with the inorder successor. If node to be deleted is left child of its parent then put address of successor in left pointer of parent of that node otherwise put address of successor in right pointer of parent of that node. Here also like case_a() and case_b() we'll consider the condition of node to be deleted being the root node separately.

```

if(par==NULL)
    root=suc;
else
    if(loc==par->lchild)
        par->lchild=suc;
    else
        par->rchild=suc;

```

We have attached successor with the parent of the deleted node. Now we have to attach the successor with children of the deleted node so that the successor comes at the place of deleted node fully. So we will put the address of left child of node being deleted in the left pointer of successor and similarly for right child.

```

suc->lchild=loc->lchild;
suc->rchild=loc->rchild;

```

Now we see that successor has been removed from its original place in the tree and now it is at the place of deleted node. Note that replacement of node is done by exchanging pointers and not just by copying the contents.

```

case_c(struct node *par,struct node *loc)
{
    struct node *ptr,*ptrsave,*suc,*parsuc;
    ptrsave=loc;
    ptr=loc->rchild;
    while(ptr->lchild!=NULL)
    {
        ptrsave=ptr;
        ptr=ptr->lchild;
    }
    suc=ptr;
    parsuc=ptrsave;
    if(suc->lchild==NULL && suc->rchild==NULL)
        case_a(parsuc,suc);
}

```

```

    else
        case_b(parsuc,suc);

    if(par==NULL) /*if item to be deleted is root node */
        root=suc;
    else
        if(loc==par->lchild)
            par->lchild=suc;
        else
            par->rchild=suc;

        suc->lchild=loc->lchild;
        suc->rchild=loc->rchild;
}

```

Traversal in Binary Search Tree

Traversal in Binary Search Tree is same as preorder, inorder and postorder traversal of binary Tree.

Preorder Traversal

```

if(ptr!=NULL)
{
    printf("%d ",ptr->info);
    preorder(ptr->lchild);
    preorder(ptr->rchild);
}

```

Inorder Traversal

```

if(ptr!=NULL)
{
    inorder(ptr->lchild);
    printf("%d ",ptr->info);
    inorder(ptr->rchild);
}

```

Postorder Traversal

```

if(ptr!=NULL)
{
    postorder(ptr->lchild);
    postorder(ptr->rchild);
    printf("%d ",ptr->info);
}

```

```

/*Insertion, Deletion and Traversal in Binary Search Tree*/
#include <stdio.h>
#include <malloc.h>

struct node
{
    int info;
    struct node *lchild;
    struct node *rchild;
}*root;

main()
{
    int choice,num;
    root=NULL;
    while(1)
    {
        printf("\n");
        printf("1.Insert\n");
        printf("2.Delete\n");
        printf("3.Inorder Traversal\n");
        printf("4.Preorder Traversal\n");
        printf("5.Postorder Traversal\n");
        printf("6.Quit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                printf("Enter the number to be inserted : ");
                scanf("%d",&num);
                insert(num);
                break;
            case 2:
                printf("Enter the number to be deleted : ");
                scanf("%d",&num);
                del(num);
                break;
            case 3:
                inorder(root);
                break;
            case 4:
                preorder(root);
                break;
            case 5:
                postorder(root);
                break;
        }
    }
}

```

```

case 6:
    exit( );
default:
    printf("Wrong choice\n");
}/*End of switch */
}/*End of while */
}/*End of main()*/

```

```
find(int item, struct node **par, struct node **loc)
```

```
{
    struct node *ptr, *ptrsave;
```

(circled)

```
if(root==NULL) /*tree empty*/
{
```

```
    *loc=NULL;
    *par=NULL;
    return;
```

```
}
```

```
if(item==root->info) /*item is at root*/
{
```

```
    *loc=root;
    *par=NULL;
    return;
```

```
}
```

```
/*Initialize ptr and ptrsave*/
if(item<root->info)
```

```
    ptr=root->lchild;
else
```

```
    ptr=root->rchild;
```

```
ptrsave=root;
```

```
while(ptr!=NULL)
```

```
{
    if(item==ptr->info)
    {
        *loc=ptr;
        *par=ptrsave;
        return;
    }
```

```
    ptrsave=ptr;
```

```
    if(item<ptr->info)
        ptr=ptr->lchild;
    else
```

```
        ptr=ptr->rchild;
```

```
}/*End of while */
```

```
*loc=NULL; /*item not found*/
```

(circled)

```
/*End of find()*/

```

(circled)

2d Y

```

insert(int item)
{
    struct node *tmp, *parent, *location;
    find(item, &parent, &location);
    if(location!=NULL)
    {
        printf("Item already present");
        return;
    }

    tmp=(struct node *)malloc(sizeof(struct node));
    tmp->info=item;
    tmp->lchild=NULL;
    tmp->rchild=NULL;

    if(parent==NULL)
        root=tmp;
    else
        if(item<parent->info)
            parent->lchild=tmp;
        else
            parent->rchild=tmp;
}

/*End of insert()*/
}

del(int item)
{
    struct node *parent, *location;
    if(root==NULL)
    {
        printf("Tree empty");
        return;
    }

    find(item, &parent, &location);
    if(location==NULL)
    {
        printf("Item not present in tree");
        return;
    }

    if(location->lchild==NULL && location->rchild==NULL)
        case_a(parent, location);
    if(location->lchild!=NULL && location->rchild==NULL)
        case_b(parent, location);
    if(location->lchild==NULL && location->rchild!=NULL)
        case_b(parent, location);
    if(location->lchild!=NULL && location->rchild!=NULL)
        case_c(parent, location);
}

/*End of del()*/
}
  
```

Delete

```

case_a(struct node *par,struct node *loc)
{
    if(par==NULL) /*item to be deleted is root node*/
        root=NULL;
    else
        if(loc==par->lchild)
            par->lchild=NULL;
        else
            par->rchild=NULL;
}/*End of case_a()*/



case_b(struct node *par,struct node *loc)
{
    struct node *child;

    /*Initialize child*/
    if(loc->lchild!=NULL) /*item to be deleted has lchild */
        child=loc->lchild;
    else /*item to be deleted has rchild */
        child=loc->rchild;
    if(par==NULL) /*Item to be deleted is root node*/
        root=child;
    else
        if( loc==par->lchild) /*item is lchild of its parent*/
            par->lchild=child;
        else /*item is rchild of its parent*/
            par->rchild=child;
}/*End of case_b()*/



case_c(struct node *par,struct node *loc)
{
    struct node *ptr,*ptrsave,*suc,*parsuc;

    /*Find inorder successor and its parent*/
    ptrsave=loc;
    ptr=loc->rchild;
    while(ptr->lchild!=NULL)
    {
        ptrsave=ptr;
        ptr=ptr->lchild;
    }
    suc=ptr;
    parsuc=ptrsave;

    if(suc->lchild==NULL && suc->rchild==NULL)
        case_a(parsuc,suc);
    else
        case_b(parsuc,suc);
}

```

```

if(par==NULL) /* if item to be deleted is root node */
    root=suc;
else
    if(loc==par->lchild)
        par->lchild=suc;
    else
        par->rchild=suc;

    suc->lchild=loc->lchild;
    suc->rchild=loc->rchild;
}/*End of case_c()*/

```

```

preorder(struct node *ptr)
{
    if(root==NULL)
    {
        printf("Tree is empty");
        return;
    }
    if(ptr!=NULL)
    {
        printf("%d ",ptr->info);
        preorder(ptr->lchild);
        preorder(ptr->rchild);
    }
}/*End of preorder()*/

```

```

inorder(struct node *ptr)
{
    if(root==NULL)
    {
        printf("Tree is empty");
        return;
    }
    if(ptr!=NULL)
    {
        inorder(ptr->lchild);
        printf("%d ",ptr->info);
        inorder(ptr->rchild);
    }
}/*End of inorder()*/

```

```

postorder(struct node *ptr)
{
    if(root==NULL)
    {
        printf("Tree is empty");
        return;
    }
}

```

```

if(ptr!=NULL)
{
    postorder(ptr->lchild);
    postorder(ptr->rchild); ✓
    printf("%d ",ptr->info);
}
/*End of postorder()*/

```

Recursive Function for finding a node in Binary search tree

Initially value of ptr will be root.

```

struct node* search(struct node *ptr, int info)
{
    if(ptr!=NULL)
        if(info < ptr->info)
            ptr=search(ptr->lchild,info);
        else if( info > ptr->info)
            ptr=search(ptr->rchild,info);
    return(ptr);
}
/*End of search()*/

```

Threads

In linked representation of binary tree we can see that most of the nodes have NULL value in their left and right pointer fields. It will be useful to use these pointer fields to keep some other information for operations in binary tree. Traversing is the most common operation in binary tree. We can use these pointer fields to contain the address pointer which points to the nodes higher in the tree. Such pointer which keeps the address of the nodes higher in tree is called thread. A binary tree which implements these pointers is called threaded binary tree.

We can have threading corresponding to any of the three traversals i.e postorder, preorder and inorder. There may be two types of inorder threading, one-way inorder threading and two-way inorder threading. In one-way inorder threading, right field of the node will keep the thread pointer which will point to the next node in the sequence of the inorder traversal or we can say that right thread will point to the inorder successor of the node.

In two-way inorder threading left field of node will also keep the thread pointer which will point to the previous node in the sequence of inorder traversal or we can say left thread will point to the inorder predecessor of the node.

If we use right field of node to take the thread then this is called right in-threaded binary tree. When we use left field of node to take the thread then this is called left in-threaded binary tree. If both left and right fields are used for threading then it is called fully threaded binary tree or in-threaded binary tree.

Chapter 6

Graph

A graph G is a collection of two sets V & E where V is the collection of vertices v_0, v_1, \dots, v_{n-1} also called nodes and E is the collection of edges e_1, e_2, \dots, e_n where e_i is an arc which connects two nodes. This can be represented as-

$$G = (V, E)$$

$$V(G) = (v_0, v_1, \dots, v_n) \text{ or set of vertices}$$

$$E(G) = (e_1, e_2, \dots, e_n) \text{ or set of edges}$$

A graph can be of two types-

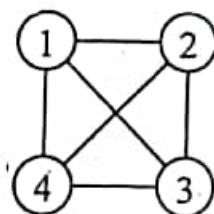
1. Undirected graph
2. Directed graph

Undirected Graph-

A graph, which has unordered pair of vertices, is called undirected graph. Suppose there is an edge between v_0 & v_1 then it can be represented as (v_0, v_1) or (v_1, v_0) also

$$V(G) = \{1, 2, 3, 4\}$$

$$E(G) = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$$



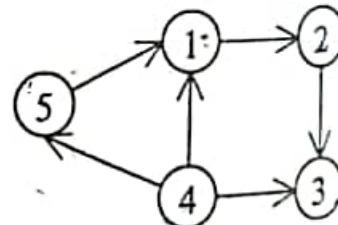
Here we can see this graph has 4 nodes and 6 edges.

Directed Graph-

A directed graph or digraph is a graph which has ordered pair of vertices $\langle v_1, v_2 \rangle$ where v_1 is the tail and v_2 is the head of the edge. In this type of graph each edge has direction, means $\langle v_1, v_2 \rangle$ and $\langle v_2, v_1 \rangle$ will represent different edges. Here the edge means that a direction will be associated with that edge. Directed graph is also known as digraph.

$$V(G) = \{1, 2, 3, 4, 5\}$$

$$E(G) = \{\langle 1, 2 \rangle, \langle 2, 3 \rangle, \langle 4, 3 \rangle, \langle 4, 1 \rangle, \langle 4, 5 \rangle, \langle 5, 1 \rangle\}$$



This graph has 5 nodes and 6 edges.

Weighted graph A graph is said to be weighted if its edges have been assigned some non negative value as weight. A weighted graph is also known as network. Graph G9 is a weighted graph.

Adjacent nodes A node u is adjacent to another node or is a neighbour of another node

If there is an edge from node u to node v . In undirected graph if (v_0, v_1) is an edge then v_0 is adjacent to v_1 and v_1 is adjacent to v_0 . In a digraph if $\langle v_0, v_1 \rangle$ is an edge then v_0 is adjacent to v_1 and v_1 is adjacent from v_0 .

In a digraph the edge $\langle v_0, v_1 \rangle$ is incident on nodes v_0 and v_1 .

Path A path from node u_0 to node u_n is a sequence of nodes $u_0, u_1, u_2, u_3, \dots, u_{n-1}, u_n$ such that u_0 is adjacent to u_1 , u_1 is adjacent to u_2 , u_2 is adjacent to u_3 , \dots , u_{n-1} is adjacent to u_n . In other words we can say that $(u_0, u_1), (u_1, u_2), (u_2, u_3), \dots, (u_{n-1}, u_n)$ are edges.

Length of path Length of a path is the total number of edges included in the path.

Closed path A path is said to be closed if first and last nodes of the path are same.

Simple path Simple path is a path in which all the nodes are distinct with an exception that the first and last nodes of the path can be same.

Cycle Cycle is a simple path in which first and last nodes are the same or we can say that a closed simple path is a cycle.

In a digraph a path is called a cycle if it has one or more nodes and the start node is connected to the last node. In graph G7, path ACBA is a cycle and in graph G9 path ABA is a cycle.

In an undirected graph a path is called a cycle if it has at least three nodes and the start node is connected to the last node. In undirected graph if (u, v) is an edge then $u-v-u$ should not be considered a path since (u, v) and (v, u) are the same edges. So for a path to be a cycle in an undirected graph there should be at least three nodes.

Cyclic graph A graph that has cycles is called a cyclic graph.

Ayclic graph A graph that has no cycles is called an acyclic graph.

Dag A directed acyclic graph is named as dag after its acronym. Graph G5 is an example of a dag.

Degree In an undirected graph, the number of edges connected to a node is called the degree of that node, or we can say that degree of a node is the number of edges incident on it. In graph G2 degree of node A is 1, degree of node B is zero. In graph G3 degree of node A is 3, of node B is 2.

In a digraph, there are two degrees for every node known as indegree and outdegree.

Indegree The indegree of a node is the number of edges coming to that node or in other words edges incident to it. In graph G8, the indegree of nodes A, B, D and G are 0, 2, 6 and 1 respectively.

Outdegree The outdegree of node is the number of edges going outside from that node, or in other words the edges incident from it. In graph G8, outdegrees of nodes A, B, D, F, and G are 3, 1, 6, 3, and 2 respectively.

Source A node, which has no incoming edges, but has outgoing edges, is called a source. The indegree of source is zero. In graph G8, nodes A and F are sources.

Sink A node, which has no outgoing edges but has incoming edges, is called a sink. The outdegree of a sink is zero. In graph G8, node D is a sink.

Pendant node A node is said to be pendant if its indegree is equal to 1 and outdegree is equal to 0.

Reachable If there is a path from a node to any other node then it will be called as reachable from that node.

Isolated node If a node has no edges connected with any other node then its degree will be 0 and it will be called isolated node. In graph G2, node B is an isolated node.

Successor and predecessor In digraph if a node v_0 is adjacent to node v_1 , then v_0 is the predecessor of v_1 , and v_1 is the successor of v_0 . In graph G, node A is predecessor of node B, node B is predecessor of node C, node B is successor of node A and node C is successor of node B.

Connected graph An undirected graph is connected if there is a path from any node of graph to any other node, or any node is reachable from any other node. Graph G1 and G3 are connected graphs while graph G2 is not a connected graph.

Strongly connected A digraph is strongly connected if there is a directed path from any node of graph to any other node. We can also say that a digraph is strongly connected if for any pair of nodes u and v, there is a path from u to v and also a path from v to u. Graph G7 is a strongly connected graph.

Weakly connected A digraph is called weakly connected or unilaterally connected if for any pair of nodes u and v, there is a path from u to v or a path from v to u. If from the digraph we remove the directions and the resulting undirected graph is connected then that digraph is weakly connected. Graph G6 is a weakly connected graph.

Maximum edges in graph In an undirected graph there can be $n(n-1)/2$ maximum edges and in a digraph there can be $n(n-1)$ maximum edges, where n is the total number of nodes in the graph.

Complete graph A graph is complete if any node in the graph is adjacent to all the nodes of the graph or we can say that there is an edge between any pair of nodes in the graph. An undirected complete graph will contain $n(n-1)/2$ edges.

Multiple edge If between a pair of nodes there are more than one edges then they are known as multiple edges or parallel edges. In graph G3, there are multiple edges between nodes A and C.

Loop An edge will be called loop or self edge if it starts and ends on the same node. Graph G4 has a loop at node B.

Multigraph A graph which has loop or multiple edges can be described as multigraph. Graphs G3 and G4 are multigraphs.

Regular graph A graph is regular if every node is adjacent to the same number of nodes. Graph G1 is regular since every node is adjacent to 3 nodes.

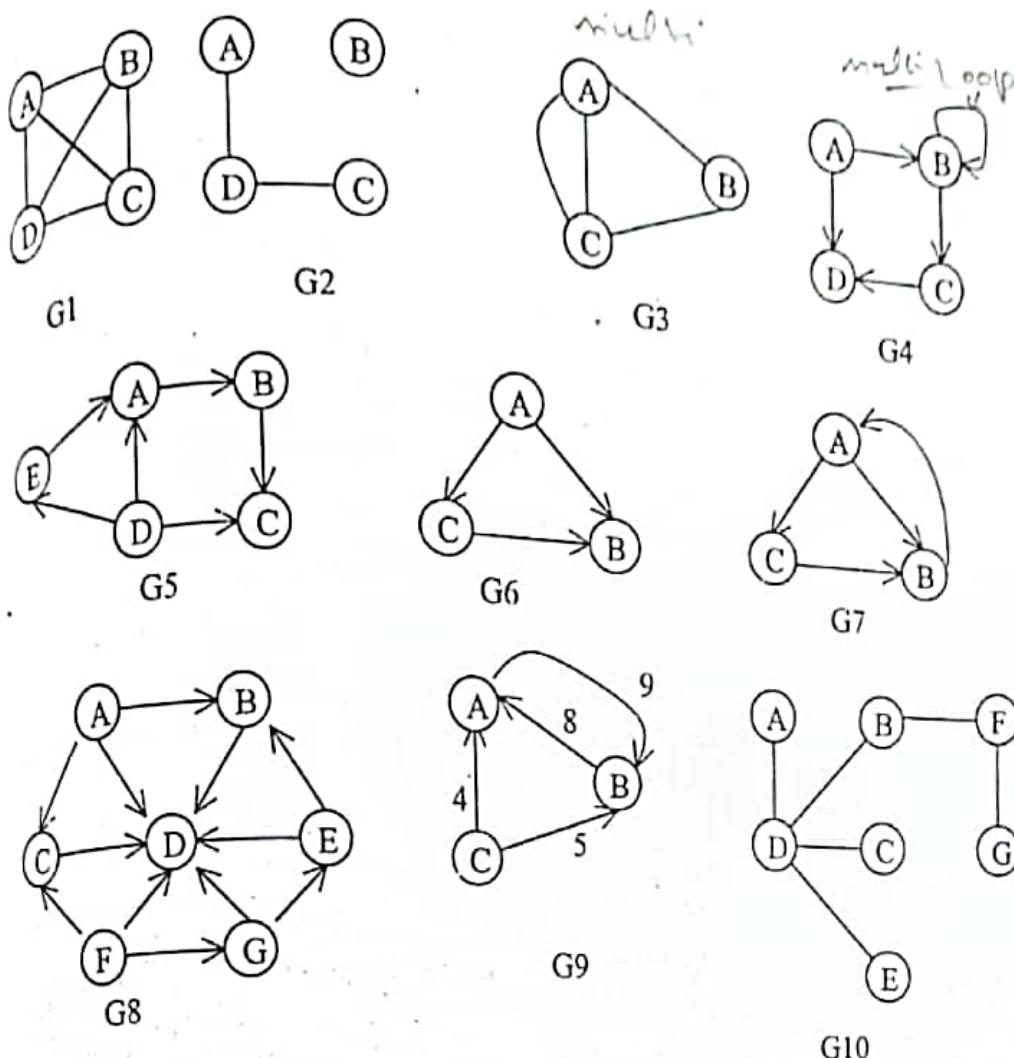
Planar graph A graph is called planar if it can be drawn in a plane without any two edges intersecting. Graph G1 is not a planar graph, while graphs G2, G3, G4 are planar graphs.

Articulation point If on removing a node from the graph the graph becomes disconnected then that node is called the articulation point.

Bridge If on removing an edge from the graph the graph becomes disconnected then that edge is called the bridge.

Biconnected graph A graph with no articulation points is called a biconnected graph.

Tree An undirected connected graph will be called tree if there is no cycle in it. So we can see that the tree structure, which we studied in earlier chapter, is a special form of graph structure. In tree structure, any node can have many children but it will have only one parent, while in graph structure any node can have many children and many parents. Graph G10 is a tree.



Representation of Graph-

We have mainly two components in graph, nodes & edges. Now we have to design the data structure to keep these components in mind. There are two ways for representing the graph in computer memory. First one is the sequential representation and second one is linked list representation.

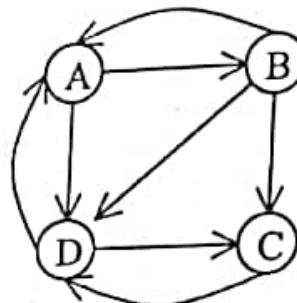
1. Adjacency Matrix-

Adjacency matrix is the matrix, which keeps the information of adjacent nodes. In other words, we can say that this matrix keeps the information that whether this node is adjacent to any other node or not. We know very well that we can represent a matrix in two dimensional array of $n \times n$ or $\text{array}[n][n]$, where first subscript will be row and second subscript will be column of that matrix. Suppose there are 4 nodes in graph then row1 represents the node1, row2 represents the node2 and so on. Similarly column1 represents node1, column2 represents node2 and so on. The entry of this matrix will be as.

$\text{Arr}[i][j]$ = 1 If there is an edge from node i to node j
 = 0 If there is no edge from node i to node j

Hence, all the entries of this matrix will be either 1 or 0.

Let us take a graph-

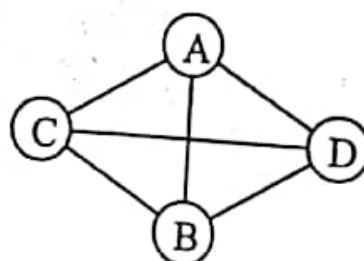


The corresponding adjacency matrix for this graph will be-

$$\begin{array}{c} \begin{matrix} & \text{A} & \text{B} & \text{C} & \text{D} \end{matrix} \\ \text{Adjacency Matrix } \mathbf{A} = \begin{matrix} \text{A} & \left[\begin{matrix} 0 & 1 & 0 & 1 \end{matrix} \right] \\ \text{B} & \left[\begin{matrix} 1 & 0 & 1 & 1 \end{matrix} \right] \\ \text{C} & \left[\begin{matrix} 0 & 0 & 0 & 1 \end{matrix} \right] \\ \text{D} & \left[\begin{matrix} 1 & 0 & 1 & 0 \end{matrix} \right] \end{matrix} \end{array}$$

This adjacency matrix is maintained in the array $\text{arr}[4][4]$. Here the entry of matrix $\text{arr}[0][1] = 1$, which represents there is an edge in the graph from node A to node B. Similarly $\text{arr}[2][0] = 0$, which represents there is no edge from node C to node A.

Let us take an undirected graph-



The corresponding adjacency matrix for this graph will be-

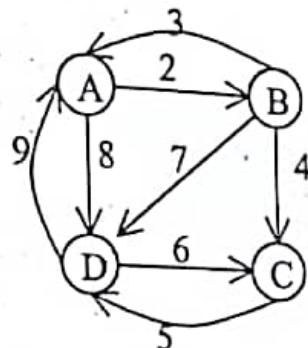
$$\begin{array}{c} \begin{matrix} & \text{A} & \text{B} & \text{C} & \text{D} \end{matrix} \\ \text{Adjacency Matrix } \mathbf{A} = \begin{matrix} \text{A} & \left[\begin{matrix} 0 & 1 & 1 & 1 \end{matrix} \right] \\ \text{B} & \left[\begin{matrix} 1 & 0 & 1 & 1 \end{matrix} \right] \\ \text{C} & \left[\begin{matrix} 1 & 1 & 0 & 1 \end{matrix} \right] \end{matrix} \end{array}$$

Note that the adjacent matrix for an undirected graph will be a symmetric matrix. This implies that for every i and j , $A[i][j] = A[j][i]$ in an undirected graph. In an undirected graph rowsum and columnsum for a node is same and represents the degree of that node and in directed graph rowsum represents the outdegree and columnsum represents the indegree of that node.

Suppose a graph has some weight on its edge then the elements of adjacency matrix will be defined as-

$=$ Weight on edge If there is an edge from node i to node j .
 $A[i][j] = 0$ Otherwise

Let us take the same graph-



The corresponding adjacency matrix will be as-

$$\text{Weighted Adjacency Matrix } W = \begin{array}{c|cccc} & & A & B & C & D \\ \hline A & 0 & 2 & 0 & 8 \\ B & 3 & 0 & 4 & 7 \\ C & 0 & 0 & 0 & 5 \\ D & 9 & 0 & 6 & 0 \end{array}$$

Here all the elements of matrix represent the weight on that edge.

/ Program for creation of adjacency matrix */*

```
#include<stdio.h>
#define max 20
```

```
int adj[max][max]; /* Adjacency matrix */
int n; /* Denotes number of nodes in the graph */
```

```
main()
{
    int max_edges, i, j, origin, destin;
    char graph_type;
    printf("Enter number of nodes : ");
    scanf("%d", &n);
```

```

printf("Enter type of graph, directed or undirected (d/u) : ");
fflush(stdin);
scanf("%c",&graph_type);

if(graph_type=='u')
    max_edges=n*(n-1)/2;
else
    max_edges=n*(n-1);

for(i=1;i<=max_edges;i++)
{
    printf("Enter edge %d ( 0 0 to quit ) : ",i);
    scanf("%d %d",&origin,&destin);
    if( (origin==0) && (destin==0) )
        break;
    if( origin > n || destin > n || origin<=0 || destin<=0 )
    {
        printf("Invalid edge!\n");
        i--;
    }
    else
    {
        adj[origin][destin]=1;
        if( graph_type=='u' )
            adj[destin][origin]=1;
    }
}
/*End of for*/

printf("The adjacency matrix is :\n");
for(i=1;i<=n;i++)
{
    for(j=1;j<=n;j++)
        printf("%4d",adj[i][j]);
    printf("\n");
}
/*End of main()*/

```

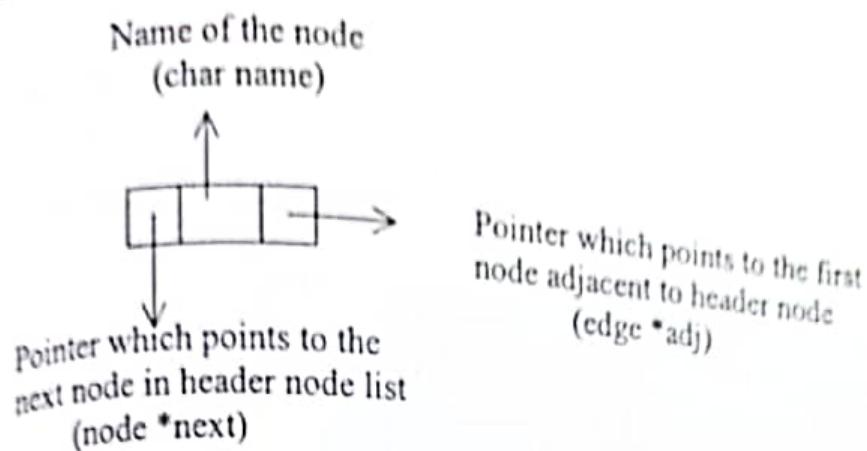
2. Adjacency List-

If the adjacency matrix of the graph is sparse then it is more efficient to represent the graph through adjacency list.

In adjacency list representation of graph, we will maintain two lists. First list will keep track of all the nodes in the graph and second list will maintain a list of adjacent nodes of each node. Suppose there are n nodes then we will create one list which will keep information of all nodes in the graph and after that we will create n lists., where each list will keep information of all adjacent nodes of that particular node.

Each list has a header node, which will be the corresponding node in the first list.

Structure of header node



```
struct node{
    struct node *next;
    char name;
    struct edge *adj;
};
```

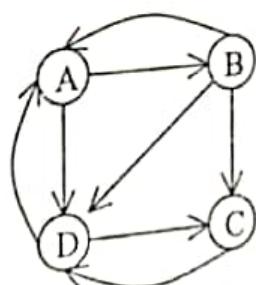
Structure of edge

Name of the destination node
of the edge (char dest)

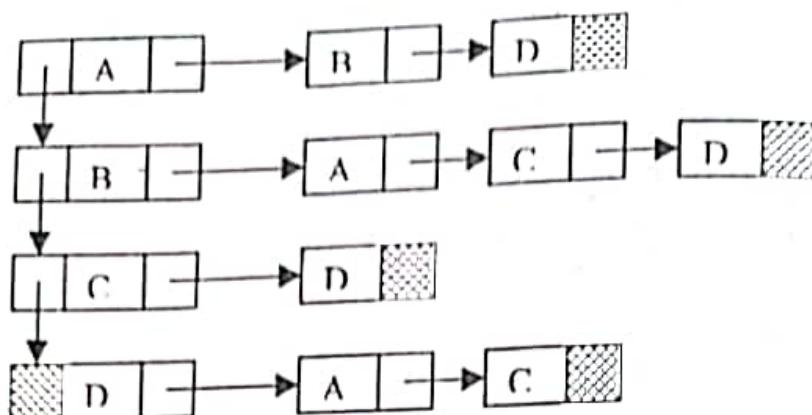
Pointer which points to the next adjacent node of header node
(edge *link)

```
struct edge{
    char dest;
    struct edge *link;
};
```

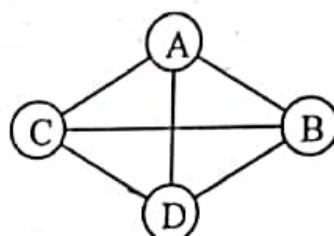
Let us take the same graph-



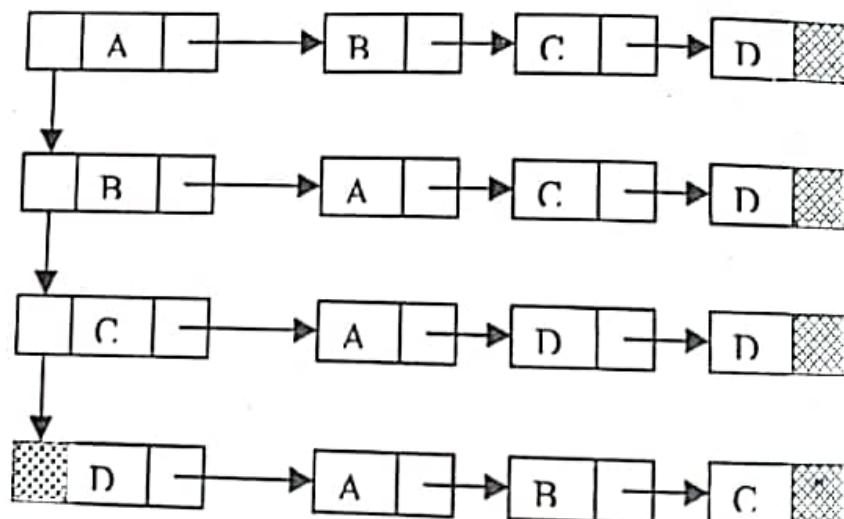
The adjacency list for this graph will be as-



Let us take an undirected graph-



The adjacency list for this graph will be as -



Operations on Graph-

As we have seen, a graph can be represented in two ways-

1. Adjacency Matrix
2. Adjacency list

The two main operations on graph will be-

1. Insertion

2. Deletion

Traversing of graph will be described later.

Here insertion and deletion will be also on two things-

On node

On edge

Now we will describe insertion and deletion operation on adjacency matrix and adjacency

list.

Insertion in Adjacency Matrix-

Node insertion-

Insertion of node requires only addition of one row and one column with zero entries in that row and column. Let us take an adjacency matrix-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

Suppose we want to add one node E in the graph then we have a need to add one row and one column with all zero entries for node E.

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	0	0	1	0
D	1	0	1	0	0
E	0	0	0	0	0

Edge insertion-

We know that entry of adjacency matrix 1 represents the edge between two nodes, and 0 represents no edge between those two nodes. Therefore, insertion of edge requires changing the value 0 into 1 for those particular nodes.

Let us take an adjacency matrix-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

Here we can see that there is no edge between D to B. So adjacency matrix has 0 entry at 4th row 2nd column. Suppose we want to insert an edge between D to B, then we have a need to change the 0 entry into 1 at the position 4th row 2nd column. Now the adjacency matrix will be-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	1	1	0

Deletion in adjacency matrix-

Node deletion-

Deletion of node requires deletion of that particular row and column in adjacency matrix for node to be deleted, because node deletion requires deletion of all the edges which are connected to that particular node.

Let us take an adjacency matrix-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

Suppose we want to delete the node D, then 4th row and 4th column of adjacency matrix will be deleted. Now the adjacency matrix will be-

	A	B	C
A	0	1	0
B	1	0	1
C	0	0	0

Edge deletion-

Deletion of an edge requires changing the value 1 to 0 for those particular nodes. Let us take an adjacency matrix-

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	0	1
D	1	0	1	0

Here we can see that an edge exists between node B to node C. So adjacency matrix has entry 1 at 2nd row 3rd column. Suppose we want to delete the edge which is in between B and C, then we have a need to change the entry 1 to 0 at the position 2nd row, 3rd column. Now the adjacency matrix will be-

	A	B	C	D
A	0	1	0	1
B	1	0	0	1
C	0	0	0	1
D	1	0	1	0

```

/*Program for addition and deletion of nodes and edges of graph using adjacency matrix */
#include<stdio.h>
#define max 20
int adj[max][max];
int n;
main( )
{
    int choice;
    int node,origin,destin;

    create_graph( );
    while(1)
    {
        printf("1.Insert a node\n");
        printf("2.Insert an edge\n");
        printf("3.Delete a node\n");
        printf("4.Delete an edge\n");
        printf("5.Display\n");
        printf("6.Exit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                insert_node( );
                break;
            case 2:
                printf("Enter an edge to be inserted : ");
                scanf("%d %d",&origin,&destin);
                insert_edge(origin,destin);
                break;
            case 3:
                printf("Enter a node to be deleted : ");
                scanf("%d",&node);
                delete_node(node);
                break;
            case 4:
                printf("Enter an edge to be deleted : ");
                scanf("%d %d",&origin,&destin);
        }
    }
}

```

```

        del_edge(origin,destin);
        break;
    case 5:
        display();
        break;
    case 6:
        exit( );
    default:
        printf("Wrong choice\n");
        break;
    }/*End of switch*/
}/*End of while*/
}/*End of main()*/



create_graph()
{
    int i,max_edges,origin,destin;

    printf("Enter number of nodes : ");
    scanf("%d",&n);
    max_edges=n*(n-1); /* Taking directed graph */

    for(i=1;i<=max_edges;i++)
    {
        printf("Enter edge %d( 0 0 ) to quit : ",i);
        scanf("%d %d",&origin,&destin);
        if((origin==0) && (destin==0))
            break;
        if( origin > n || destin > n || origin<=0 || destin<=0)
        {
            printf("Invalid edge!\n");
            i--;
        }
        else
            adj[origin][destin]=1;
    }/*End of for*/
}/*End of create_graph()*/



display()
{
    int i,j;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
            printf("%4d",adj[i][j]);
        printf("\n");
    }
}/*End of display()*/

```

```

insert_node()
{
    int i;
    n++; /*Increase number of nodes in the graph*/
    printf("The inserted node is %d \n",n);
    for(i=1;i<=n;i++)
    {
        adj[i][n]=0;
        adj[n][i]=0;
    }
}/*End of insert_node()*/
delete_node(char u)
{
    int i,j;
    if(n==0)
    {
        printf("Graph is empty\n");
        return;
    }
    if(u>n)
    {
        printf("This node is not present in the graph\n");
        return;
    }
    for(i=u;i<=n-1;i++)
        for(j=1;j<=n;j++)
        {
            adj[j][i]=adj[j][i+1]; /* Shift columns left */
            adj[i][j]=adj[i+1][j]; /* Shift rows up */
        }
    n--; /*Decrease the number of nodes in the graph */
}/*End of delete_node()*/
insert_edge(char u,char v)
{
    if(u > n)
    {
        printf("Source node does not exist\n");
        return;
    }
    if(v > n)
    {
        printf("Destination node does not exist\n");
        return;
    }
}/*End of insert_edge()*/

```

```

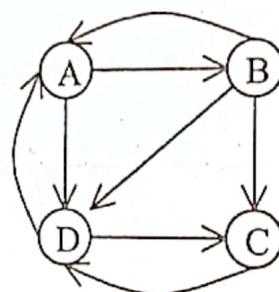
del_edge(char u,char v)
{
    if(u>n || v>n || adj[u][v]==0)
    {
        printf("This edge does not exist\n");
        return;
    }
    adj[u][v]=0;
}/*End of del_edge( )*/

```

Insertion in adjacency list-

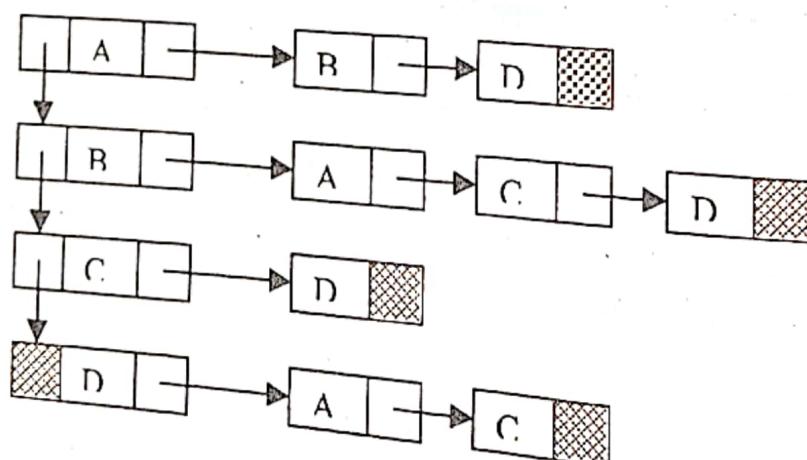
Node insertion-

Insertion of node in adjacency list requires only insertion of that node in header nodes of adjacency list. Let us take a graph G-



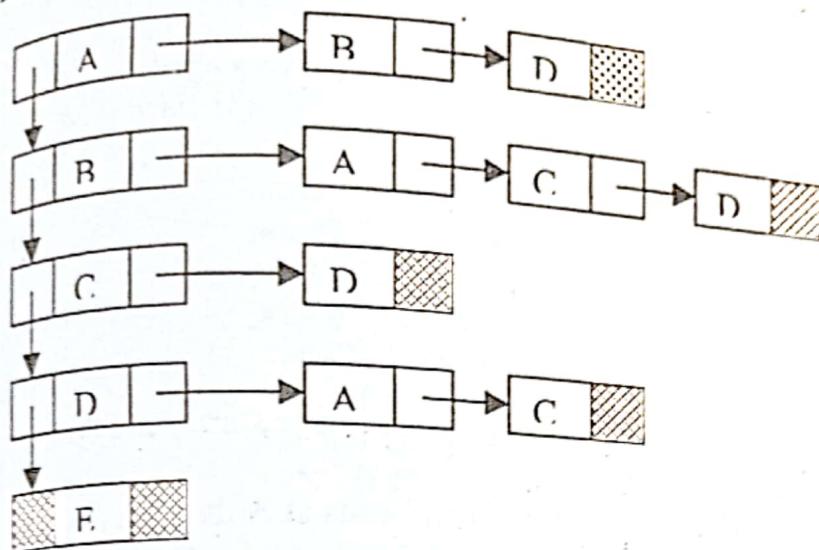
The adjacency list for this graph will be as-

Header Nodes



Suppose we want to insert one node E then it requires addition of node E in header node only. Now the adjacency list will be-

Header Nodes

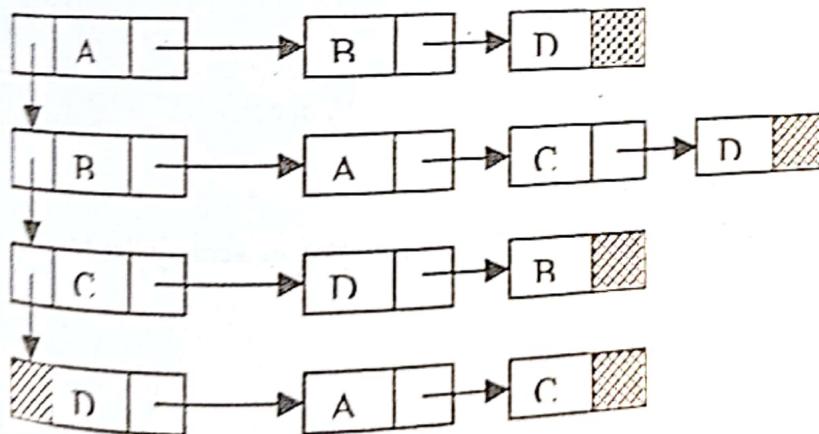


Edge insertion-

Insertion of an edge requires add operation in the list of the starting node of edge. Remember here graph is directed graph. In undirected graph, it will be added in the list of both nodes.

Suppose we want to add the edge which starts from node C and ends at node B, then add operation is needed in the list of C node. Now the adjacency list of graph will be-

Header Nodes



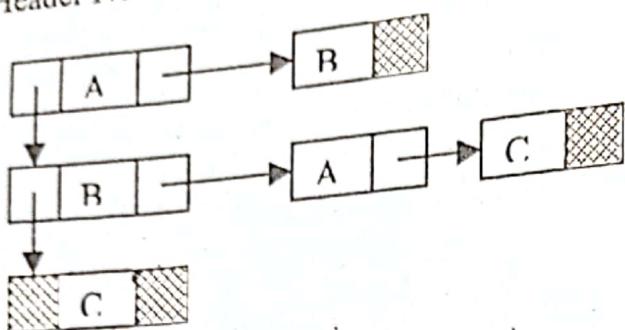
Deletion in adjacency list-

Node deletion-

Deletion of node requires deletion of that particular node from header node and from the entire list wherever it is coming. Deletion of node from header will automatically free the list attached to that node.

Suppose we want to delete the node D from graph then we have a need to delete node D from header node and from the list of A, B and C nodes. Now the adjacency list of graph will be-

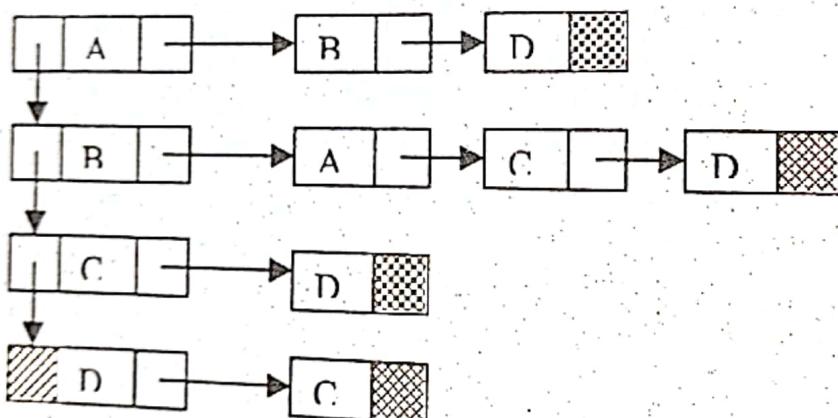
Header Nodes

**Edge deletion-**

Deletion of edge requires deletion in the list of that node where edge starts, and that element of the list will be deleted where the edge ends.

Suppose we want to delete the edge which starts from D and ends at A then we have a need to delete in the list of node D and element in the list deleted will be A. Now the adjacency list of the graph will be-

Header Nodes



```

/* Program for insertion and deletion of nodes and edges in a graph using adjacency list */
#include<stdio.h>
struct edge;
struct node
{
    struct node *next;
    char name;
    struct edge *adj;
}*start=NULL;

struct edge
{
    char dest;
    struct edge *link;
};

struct node *find(char item);
  
```

```
main()
{
    int choice;
    char node,origin,destin;
    while(1)
    {
        printf("1.Insert a node\n");
        printf("2.Insert an edge\n");
        printf("3.Delete a node\n");
        printf("4.Delete an edge\n");
        printf("5.Display\n");
        printf("6.Exit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                printf("Enter a node to be inserted : ");
                fflush(stdin);
                scanf("%c",&node);
                insert_node(node);
                break;
            case 2:
                printf("Enter an edge to be inserted : ");
                fflush(stdin);
                scanf("%c %c",&origin,&destin);
                insert_edge(origin,destin);
                break;
            case 3:
                printf("Enter a node to be deleted : ");
                fflush(stdin);
                scanf("%c",&node);
                /*This fn deletes the node from header node list*/
                delete_node(node);
                /* This fn deletes all edges coming to this node */
                delnode_edge(node);
                break;
            case 4:
                printf("Enter an edge to be deleted : ");
                fflush(stdin);
                scanf("%c %c",&origin,&destin);
                del_edge(origin,destin);
                break;
            case 5:
                display();
                break;
            case 6:
                exit();
        }
    }
}
```

```

        default:
            printf("Wrong choice\n");
            break;
        }/*End of switch*/
    }/*End of while*/
}/*End of main()*/



insert_node(char node_name)
{
    struct node *tmp,*ptr;
    tmp=malloc(sizeof(struct node));
    tmp->name=node_name;
    tmp->next=NULL;
    tmp->adj=NULL;

    if(start==NULL)
    {
        start=tmp;
        return;
    }
    ptr=start;
    while( ptr->next!=NULL)
        ptr=ptr->next;
    ptr->next=tmp;
}/*End of insert_node()*/



delete_node(char u)
{
    struct node *tmp,*q;
    if(start->name == u)
    {
        tmp=start;
        start=start->next; /* first element deleted */
        free(tmp);
        return;
    }
    q=start;
    while(q->next->next != NULL)
    {
        if(q->next->name==u) /* element deleted in between */
        {
            tmp=q->next;
            q->next=tmp->next;
            free(tmp);
            return;
        }
        q=q->next;
    }/*End of while*/
}

```

```
if(q->next->name==u) /* last element deleted */
{
    tmp=q->next;
    free(tmp);
    q->next=NULL;
}

/*End of delete_node( )*/

delnode_edge(char u)
{
    struct node *ptr;
    struct edge *q,*start_edge,*tmp;
    ptr=start;
    while(ptr!=NULL)
    {
        /* ptr->adj points to first node of edge linked list */
        if(ptr->adj->dest == u)
        {
            tmp=ptr->adj;
            ptr->adj=ptr->adj->link; /* first element deleted */
            free(tmp);
            continue; /* continue searching in another edge lists */
        }
        q=ptr->adj;
        while(q->link->link != NULL)
        {
            if(q->link->dest==u) /* element deleted in between */
            {
                tmp=q->link;
                q->link=tmp->link;
                free(tmp);
                continue;
            }
            q=q->link;
        }
        /*End of while*/
        if(q->link->dest==u) /* last element deleted */
        {
            tmp=q->link;
            free(tmp);
            q->link=NULL;
        }
        ptr=ptr->next;
    }
    /*End of delnode_edge( )*/
}

insert_edge(char u,char v)
{
    struct node *locu,*locv;
    struct edge *ptr,*tmp;
```

```

locu=find(u);
locv=find(v);

if(locu==NULL)
{
    printf("Source node not present ,first insert node %c\n",u);
    return;
}
if(locv==NULL)
{
    printf("Destination node not present ,first insert node %c\n",v);
    return;
}
tmp=malloc(sizeof(struct edge));
tmp->dest=v;
tmp->link=NULL;

if(locu->adj==NULL) /* item added at the begining */
{
    locu->adj=tmp;
    return;
}
ptr=locu->adj;
while(ptr->link!=NULL)
{
    ptr=ptr->link;
}
ptr->link=tmp;

}/*End of insert_edge( )*/

struct node *find(char item)
{
    struct node *ptr,*loc;
    ptr=start;
    while(ptr!=NULL)
    {
        if(item==ptr->name)
        {
            loc=ptr;
            return loc;
        }
        else
        {
            ptr=ptr->next;
        }
    }
    loc=NULL;
    return loc;
}/*End of find( )*/

```

```

del_edge(char u,char v)
{
    struct node *locu,*locv;
    struct edge *ptr,*tmp,*q;
    locu=find(u);

    if(locu==NULL )
    {
        printf("Source node not present\n");
        return;
    }
    if(locu->adj->dest == v)
    {
        tmp=locu->adj;
        locu->adj=locu->adj->link; /* first element deleted */
        free(tmp);
        return;
    }
    q=locu->adj;
    while(q->link->link != NULL)
    {
        if(q->link->dest==v) /* element deleted in between */
        {
            tmp=q->link;
            q->link=tmp->link;
            free(tmp);
            return;
        }
        q=q->link;
    }/*End of while*/
    if(q->link->dest==v) /* last element deleted */
    {
        tmp=q->link;
        free(tmp);
        q->link=NULL;
        return;
    }
    printf("This edge not present in the graph\n");
}/*End of del_edge( )*/
display()
{
    struct node *ptr;
    struct edge *q;

    ptr=start;
    while(ptr!=NULL)
    {
        printf("%c ->",ptr->name);
}

```

```

q=ptr->adj;
while(q!=NULL)
{
    printf(" %c",q->dest);
    q=q->link;
}
printf("\n");
ptr=ptr->next;
}
/*End of display()*/

```

Path Matrix-

Let us take a graph G with n nodes v_1, v_2, \dots, v_n . The path matrix or reachable matrix of G can be defined as-

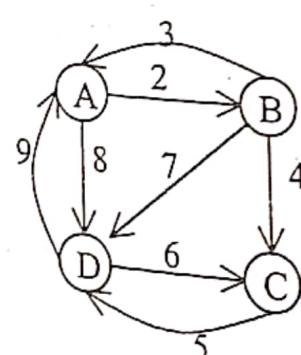
$$P[i][j] = \begin{cases} 1 & \text{if there is a path in between } v_i \text{ to } v_j \\ 0 & \text{Otherwise.} \end{cases}$$

If there is a path from v_i to v_j then it can be a simple path from v_i to v_j of length $n-1$ or less or there can be a cycle of length n or less.

A graph G will be strongly connected if there are no zero entries in path matrix means a path exists between all v_i to v_j and v_j to v_i also

Computing Path matrix from powers of adjacency matrix-

Let us take a graph and compute path matrix for it from its adjacency matrix.



The corresponding weighted adjacency matrix will be as-

$$\text{Weighted Adjacency Matrix } W = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 2 & 0 & 8 \\ B & 3 & 0 & 4 & 7 \\ C & 0 & 0 & 0 & 5 \\ D & 9 & 0 & 6 & 0 \end{array}$$

The adjacency matrix will be

Adjacency Matrix A =

$$A \begin{bmatrix} A & B & C & D \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Path length denotes the number of edges in the path. Adjacency matrix is the path matrix of path length 1. Now if we multiply the adjacency matrix with itself then we get the path matrix of length 2.

$$AM_2 = A^2 = A \begin{bmatrix} A & B & C & D \\ 2 & 0 & 2 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 \end{bmatrix}$$

In this matrix, value of $AM_2[i][j]$ will represent the number of paths of path length 2 from node v_i to v_j . For example here node A has 2 paths of path length 2 to node C, and node D has 2 paths of path length 2 to itself, node C has 1 path of path length 2 to node A.

To obtain the path matrix of length 3 we will multiply the path matrix of length 2 with adjacency matrix.

$$AM_3 = AM_2 * A = A^3 = A \begin{bmatrix} A & B & C & D \\ 1 & 2 & 1 & 4 \\ 3 & 1 & 3 & 3 \\ 0 & 1 & 0 & 2 \\ 3 & 0 & 3 & 1 \end{bmatrix}$$

Here $AM_3[i][j]$ will represent the number of paths of path length 3 from node v_i to node v_j . For example, here node A has 4 paths of path length 3 to node D, and node D has no path of path length 3 to node B.

Similarly, we can find out the path matrix for path length 4.

$$AM_4 = AM_3 * A = A^4 = A \begin{bmatrix} A & B & C & D \\ 6 & 1 & 6 & 4 \\ 4 & 3 & 4 & 7 \\ 3 & 0 & 3 & 1 \\ 1 & 3 & 1 & 6 \end{bmatrix}$$

Let us take a matrix X where

$$X_n = AM_1 + AM_2 + \dots + AM_n$$

Here $X[i][j]$ has the value of number of paths, less than or equal to length n, from node v_i to v_j . Here n is the total number of nodes in the graph.

For the above graph the value of X_4 will be as -

$$X_4 = \begin{bmatrix} 9 & 4 & 9 & 10 \\ 9 & 5 & 9 & 13 \\ 4 & 1 & 4 & 4 \\ 5 & 4 & 5 & 9 \end{bmatrix}$$

From definition of path matrix we know that $P[i][j] = 1$ if there is a path from v_i to v_j , and this path can have length n or less than n.

Now in this matrix, if we replace all nonzero entries by 1 then we will get the path matrix or reachability matrix.

$$P = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

This graph is strongly connected since all the entries are equal to 1.

```
/* Program to find out the path matrix by powers of adjacency matrix */
#include<stdio.h>
#define MAX 20
```

```
int n;
main()
{
    int w_adj[MAX][MAX], adj[MAX][MAX], adjp[MAX][MAX];
    int x[MAX][MAX], path[20][20], i, j, p;

    printf("Input number of vertices : ");
    scanf("%d", &n);
    printf("Enter the weighted adjacent matrix :\n");
    for(i=0; i<n; i++)
        for(j=0; j<n; j++)
            scanf("%d", &w_adj[i][j]);
    printf("The weighted adjacency matrix is :\n");
    display(w_adj);
```

```

/* Converts weighted adjacency matrix to boolean matrix */
to_boolean(w_adj,adj);

printf("The boolean adjacency matrix is :\n");
display(adj);
while(1)
{
    printf("Enter the path length to be searched (0 to quit) : ");
    scanf("%d",&p);
    if(p==0)
        break;

    /* Matrix adjp is equal to adj raised to power p */
    pow_matrix(adj,p,adjp);
    printf("The path matrix for lengths equal to %d is :\n",p);
    display(adjp);
}/*End of while */

for(i=0;i<n;i++)
for(j=0;j<n;j++)
    x[i][j]=0;

/*All the powers of adj will be added to matrix x */
for(p=1;p<=n;p++)
{
    pow_matrix(adj,p,adjp);
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            x[i][j]=x[i][j]+adjp[i][j];
}
printf("The matrix x is :\n");
display(x);

to_boolean(x,path);

printf("The path matrix is :\n");
display(path);
}/*End of main() */

/*This function computes the pth power of matrix adj and stores result in adjp */
pow_matrix(int adj[MAX][MAX],int p,int adjp[MAX][MAX])
{
    int i,j,k,tmp[MAX][MAX];

    /*Initially adjp is equal to adj */
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            adjp[i][j]=adj[i][j];
}

```

```

for(k=1;k<p;k++)
{
    /*Multiply adjp with adj and store result in tmp */
    multiply(adjp,adj,tmp);
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            adjp[i][j]=tmp[i][j]; /* New adjp is equal to tmp */
}
}/*End of pow_matrix()*/
/*This function multiplies mat1 and mat2 and stores the result in mat3 */
multiply(int mat1[MAX][MAX],int mat2[MAX][MAX],int mat3[MAX][MAX])
{
    int i,j,k;

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
    {
        mat3[i][j]=0;
        for(k=0;k<n;k++)
            mat3[i][j] = mat3[i][j]+ mat1[i][k] * mat2[k][j];
    }
}/*End of multiply()*/
/*This fn converts matrix mat into boolean matrix and stores result in boolmat */
to_boolean( int mat[MAX][MAX], int boolmat[MAX][MAX] )
{
    int i,j;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if (mat[i][j] == 0 )
                boolmat[i][j]=0;
            else
                boolmat[i][j]=1;
}/*End of to _boolean()*/
display(int matrix[MAX][MAX])
{
    int i,j;
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
            printf("%4d",matrix[i][j]);
        printf("\n");
    }
}/*End of display()*/

```

```

        for(k=1;k<p;k++)
    {
        /*Multiply adjp with adj and store result in tmp */
        multiply(adjp,adj,tmp);
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                adjp[i][j]=tmp[i][j]; /* New adjp is equal to tmp */
    }
}/*End of pow_matrix( )*/

/*This function multiplies mat1 and mat2 and stores the result in mat3 */
multiply(int mat1[MAX][MAX],int mat2[MAX][MAX],int mat3[MAX][MAX])
{
    int i,j,k;

    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
    {
        mat3[i][j]=0;
        for(k=0;k<n;k++)
            mat3[i][j] = mat3[i][j]+ mat1[i][k] * mat2[k][j];
    }
}/*End of multiply( )*/

/*This fn converts matrix mat into boolean matrix and stores result in boolmat */
to_boolean( int mat[MAX][MAX], int boolmat[MAX][MAX] )
{
    int i,j;
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if (mat[i][j] == 0 )
                boolmat[i][j]=0;
            else
                boolmat[i][j]=1;
}/*End of to_boolean( )*/

display(int matrix[MAX][MAX])
{
    int i,j;
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
            printf("%4d",matrix[i][j]);
        printf("\n");
    }
}/*End of display( )*/

```

Warshall's Algorithm-

As we have seen earlier, we can find the path matrix P of a given graph G with the use of powers of adjacency matrix. But this method is not efficient one. Warshall has given one efficient technique for finding path matrix of a graph which is called Warshall's algorithm.

Let us take a graph G of n vertices $v_1, v_2, v_3, \dots, v_n$. First we will take Boolean matrices P_0, P_1, \dots, P_n where $P_k[i,j]$ is defined as -

$$P_k[i][j] = \begin{cases} 1 & \text{If there is a simple path from vertices } v_i \text{ to } v_j \text{ which does not use any other node except possibly } v_1, v_2, \dots, v_k \text{ or this path does not use any node numbered higher than } k \\ 0 & \text{Otherwise} \end{cases}$$

Or we can say,

$P_0[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any node.

$P_1[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any other nodes except possibly v_1 .

$P_2[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any other nodes except v_1, v_2 .

$P_k[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any other nodes except v_1, v_2, \dots, v_k .

$P_n[i][j] = 1$ If there is a simple path from v_i to v_j which does not use any other nodes except v_1, v_2, \dots, v_n .

Here P_0 represents the adjacency matrix and P_n represents the path matrix.

$P_0[i][j] = 1$, If there is a simple path from v_i to v_j which does not use any node. The only way to go directly from v_i to v_j without using any node is to go directly from v_i to v_j . Hence $P_0[i][j] = 1$ if there is any edge from v_i to v_j . So P_0 will be the adjacency matrix.

$P_n[i][j] = 1$, If there is a simple path from v_i to v_j which does not use any nodes except v_1, v_2, \dots, v_n . There are total n nodes means this path can use all n nodes, hence from the definition of path matrix we observe that P_n is the path matrix.

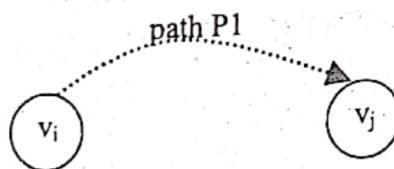
Now we'll see how we can find out the elements of matrix P_k . We know that P_0 is equal to the adjacency matrix. We have to find matrices P_1, P_2, \dots, P_n . First, we have a need to know how to find $P_k[i][j]$ from $P_{k-1}[i][j]$ then we can find all these matrices. So first we will find $P_k[i][j]$ from $P_{k-1}[i][j]$.

There can be two cases for $P_{k-1}[i][j]$

$P_{k-1}[i][j]=1$ or $P_{k-1}[i][j]=0$

(i) If $P_{k-1}[i][j]=1$
 This implies that there is a simple path P_1 from v_i to v_j which does not use any nodes except possibly v_1, v_2, \dots, v_{k-1} or this path does not use any nodes numbered higher than $k-1$. So it is obvious that this path does not use any nodes numbered higher than k also or we can say that this path does not use any other nodes except v_1, v_2, \dots, v_k . So $P_k[i][j]$ will be equal to 1.

Hence if $P_{k-1}[i][j]=1$ then definitely $P_k[i][j]=1$.



(ii) If $P_{k-1}[i][j]=0$

This means path P_1 defined above does not exist.

Now $P_k[i][j]$ can be 0 or 1.

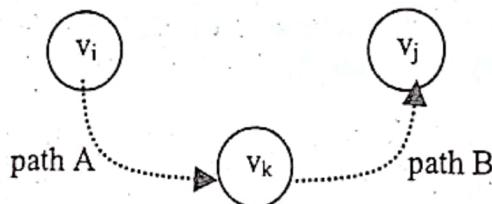
The only case for $P_k[i][j]$ will be 1 for $P_{k-1}[i][j]=0$ is when there is a path from v_i to v_j passing through nodes v_1, v_2, \dots, v_k , but there is no path from v_i to v_j using only nodes v_1, v_2, \dots, v_{k-1} . Hence we see that if we use only nodes v_1, v_2, \dots, v_{k-1} then there is no path from v_i to v_j but if we use node v_k also then we get a path from v_i to v_j . This means that there is a path from v_i to v_j which definitely passes through v_k and all other nodes in the path can be from nodes v_1, v_2, \dots, v_{k-1} .

Now we can break this path, into two paths.

1. Path A from v_i to v_k using nodes v_1, v_2, \dots, v_{k-1} .
2. Path B from v_k to v_j using nodes v_1, v_2, \dots, v_{k-1} .

From path A we can write that $P_{k-1}[i][k]=1$

From path B we can write that $P_{k-1}[k][j]=1$

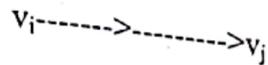


So we can conclude that if $P_{k-1}[i][j]=0$ then $P_k[i][j]$ can be equal to 1 only if $P_{k-1}[i][k]=1$ and $P_{k-1}[k][j]=1$.

So we have two cases where $P_k[i,j] = 1$

1. There is a simple path from v_i to v_j which does not use any other nodes except possibly v_1, v_2, \dots, v_{k-1} , hence

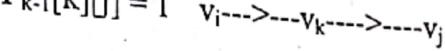
$$P_{k-1}[i][j] = 1$$



2. There is a simple path from v_i to v_k and a simple path from v_k to v_j where each path does not use any other nodes except v_1, v_2, \dots, v_{k-1} , hence

$$P_{k-1}[i][k] = 1$$

$$\text{and } P_{k-1}[k][j] = 1$$



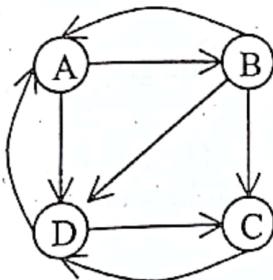
So the element of path matrix P_k can be defined as-

$$P_{k-1}[i][j]$$

$P_k[i][j] = \text{or}$

$$P_{k-1}[i][j] \text{ And } P_{k-1}[k][j]$$

Let us take a graph and find out the values of P_0, P_1, P_2, P_3, P_4 .



The first matrix P_0 is the adjacency matrix.

$$P_0 = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 1 & 0 \end{array}$$

Now we have to find P_1

Now wherever $P_0[i][j]=1$ $P_1[i][j]=1$

If $P_0[i][j]=0$ then see $P_0[i][1]$ and $P_0[1][j]$, if both are 1 then $P_1[i][j]=1$

$$P_1 = \begin{array}{c|cccc} & A & B & C & D \\ \hline A & 0 & 1 & 0 & 1 \\ B & 1 & 0 & 1 & 1 \\ C & 0 & 0 & 0 & 1 \\ D & 1 & 1 & 1 & 1 \end{array}$$

$$\text{Similarly, } P_2 = \begin{array}{c} \begin{matrix} & A & B & C & D \\ A & \left[\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{matrix} \right] \\ B \\ C \\ D \end{matrix} \end{array}$$

$$\text{Similarly, } P_3 = \begin{array}{c} \begin{matrix} & A & B & C & D \\ A & \left[\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{matrix} \right] \\ B \\ C \\ D \end{matrix} \end{array}$$

$$\text{And } P_4 = \begin{array}{c} \begin{matrix} & A & B & C & D \\ A & \left[\begin{matrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{matrix} \right] \\ B \\ C \\ D \end{matrix} \end{array}$$

Here P_0 is the adjacency matrix and P_4 is the path matrix of the graph.

```
/* Program to find path matrix by Warshall's algorithm */
#include<stdio.h>
#define MAX 20
main()
{
    int i,j,k,n;
    int w_adj[MAX][MAX],adj[MAX][MAX],path[MAX][MAX];

    printf("Enter number of vertices : ");
    scanf("%d",&n);
    printf("Enter weighted adjacency matrix :\n");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&w_adj[i][j]);

    printf("The weighted adjacency matrix is :\n");
    display(w_adj,n);

    /* Change weighted adjacency matrix into boolean adjacency matrix */
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(w_adj[i][j]==0)
                adj[i][j]=0;
            else
                adj[i][j]=1;
}
```

```

printf("The adjacency matrix is :\n");
display(adj,n);

for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        path[i][j]=adj[i][j];

for(k=0;k<n;k++)
{
    printf("P%d is :\n",k);
    display(path,n);
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            path[i][j]=( path[i][j] || ( path[i][k] && path[k][j] ) );
}
printf("Path matrix P%d of the given graph is :\n",k);
display(path,n);

/*End of main() */

display(int matrix[MAX][MAX],int n)
{
    int i,j;
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
            printf("%3d",matrix[i][j]);
        printf("\n");
    }
}

/*End of display() */

```

Modified Warshall's Algorithm-

We have seen that Warshall's algorithm gives the path matrix of graph. Now by modifying this algorithm, we will find out the shortest path matrix Q. Here $Q[i][j]$ will represent the length of shortest path from v_i to v_j .

There can be many paths from any node v_i to v_j . Our purpose is to find the length of the shortest path in between these two nodes.

We have seen that weighted matrix can be defined as-

$$W[i][j] = \begin{cases} \text{weight on edge} & \text{if there is an edge from node } i \text{ to node } j. \\ 0 & \text{otherwise} \end{cases}$$

Now we will take weight on each edge as the length of that edge.

As in Warshall's algorithm matrices were $P_0, P_1, P_2, \dots, P_n$, here we will take matrices as $Q_0, Q_1, Q_2, \dots, Q_n$.

length of shortest path from v_i to v_j using nodes $v_1, v_2, v_3, \dots, v_k$,

$$Q_k[i][j] = \begin{cases} \text{length of shortest path from } v_i \text{ to } v_j \text{ using nodes } v_1, v_2, v_3, \dots, v_k \\ \infty \quad (\text{if there is no path from } v_i \text{ to } v_j \text{ using nodes } v_1, v_2, v_3, \dots, v_k) \end{cases}$$

Hence we can say that

$Q_0[i][j]$ = length of an edge from v_i to v_j

$Q_1[i][j]$ = length of shortest path from v_i to v_j using v_1 .

$Q_2[i][j]$ = length of shortest path from v_i to v_j using v_1, v_2 .

$Q_3[i][j]$ = length of shortest path from v_i to v_j using v_1, v_2, v_3

$Q_4[i][j]$ = length of shortest path from v_i to v_j using v_1, v_2, v_3, v_4

We can find out Q_0 from the weighted adjacency matrix by replacing all zero entry by ∞ .

If there are n nodes in the graph then matrix Q_0 will represent the shortest path matrix. So now our purpose is to find out matrices $Q_1, Q_2, Q_3, \dots, Q_n$. We have already found out matrix Q_0 by weighted adjacency matrix. Now if we know how to find out the matrix Q_k from matrix Q_{k-1} then we can easily find out matrices $Q_1, Q_2, Q_3, \dots, Q_n$ also.

Now we'll see how to find out matrix Q_k from matrix Q_{k-1} .

Now we'll see how to find out matrix Q_k from matrix Q_{k-1} .

We have seen in Warshall's algorithm that $P_k[i][j]=1$ if any of these two conditions is true.

1. $P_{k-1}[i][j]=1$ or
2. $P_{k-1}[i][k]=1$ and $P_{k-1}[k][j]=1$

This means there is a path from v_i to v_j using $v_1, v_2, v_3, \dots, v_k$ in two conditions

1. There is a path from v_i to v_j using v_1, v_2, \dots, v_{k-1} (path P1)
or
2. There is a path from v_i to v_k using $v_1, v_2, v_3, \dots, v_{k-1}$ and there is a path from v_k to v_j using $v_1, v_2, v_3, \dots, v_{k-1}$ (path P2)

Here we are dealing with path lengths so lengths of paths will be as -

Length of first path will be $Q_{k-1}[i][j]$

Length of second path will be $Q_{k-1}[i][k]+Q_{k-1}[k][j]$

Now we'll select the smaller one from these two path lengths.

So value of $Q_k[i][j] = \text{Minimum}(Q_{k-1}[i][j], Q_{k-1}[i][k]+Q_{k-1}[k][j])$

Case1-

$Q_{k-1}[i][j]=\infty$ and $Q_{k-1}[i][k]+Q_{k-1}[k][j]=\infty$

Path P1 and path P2 do not exist.

So $Q_{k-1}[i][j]=\text{Minimum}(\infty, \infty)=\infty$

means there will be no path from node i to node j using $v_1, v_2, v_3, \dots, v_k$.

Case 2-

$Q_{k-1}[i][j] = \infty$ and $Q_{k-1}[i][k] + Q_{k-1}[k][j] = b$

There is no path from v_i to v_j using nodes $v_1, v_2, v_3, \dots, v_{k-1}$ but when we include node v_k then there is a path (path P2) of length b.
So $Q_{k-1}[i][j] = \text{Minimum}(\infty, b) = b$

Case 3-

$Q_{k-1}[i][j] = a$ and $Q_{k-1}[i][k] + Q_{k-1}[k][j] = \infty$

There is a path (path P1) of length a from v_i to v_j when we use nodes $v_1, v_2, v_3, \dots, v_{k-1}$ but when we include node v_k then there is no path. This is possible when $Q_{k-1}[i][k]$ or $Q_{k-1}[k][j]$ or both are infinity.

So $Q_{k-1}[i][j] = \text{Minimum}(a, \infty) = a$

Case 4-

$Q_{k-1}[i][j] = a$ and $Q_{k-1}[i][k] + Q_{k-1}[k][j] = b$

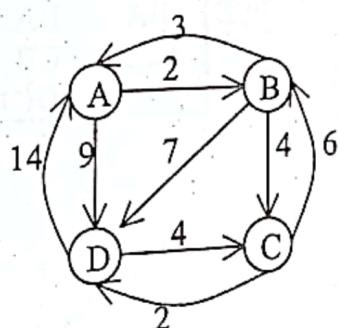
Both path P1 and path P2 exist. Length of the shorter of the two will be the value of $Q_{k-1}[i][j]$.

So $Q_{k-1}[i][j] = \text{Minimum}(a, b)$

We replaced every zero in the weighted matrix by ∞ to obtain Q_0 because if we take zero then whenever we use minimum function we will get zero as the result and we won't be able to compute the shortest path.

In program, we take ∞ to be a very large number.

Now let us take a graph and find out the shortest path matrix for it.



$$K_0 = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

Weighted adjacency matrix for this graph is

$$W = \begin{bmatrix} & & & \\ & & & \\ & & & \\ & & & \end{bmatrix}$$

$$A \quad \begin{bmatrix} 0 & 2 & 7 & 14 \\ 3 & 0 & 4 & 0 \\ 0 & 6 & 0 & 2 \\ 9 & 0 & 4 & 0 \end{bmatrix}$$

$$Q_0 = \begin{bmatrix} A & B & C & D \\ A & \infty & 2 & 0 & 9 \\ B & 3 & \infty & 4 & 7 \\ C & \infty & 6 & \infty & 2 \\ D & 14 & \infty & 4 & \infty \end{bmatrix}$$

	A	B	C	D
A	--	AB	--	AD
B	BA	--	BC	BD
C	--	CB	--	CD
D	DA	--	DC	--

After including node A (k=1)

$$Q_1 = \begin{bmatrix} A & B & C & D \\ A & \infty & 2 & 0 & 9 \\ B & 3 & 5 & 4 & 7 \\ C & \infty & 6 & \infty & 2 \\ D & 14 & 16 & 4 & 23 \end{bmatrix}$$

	A	B	C	D
A	--	AB	--	AD
B	BA	<u>BAB</u>	BC	BD
C	--	CB	--	CD
D	DA	<u>DAB</u>	DC	<u>DAD</u>

After including node B (k=2)

$$Q_2 = \begin{bmatrix} A & B & C & D \\ A & 5 & 2 & 6 & 9 \\ B & 3 & 5 & 4 & 7 \\ C & 9 & 6 & 10 & 2 \\ D & 14 & 16 & 4 & 23 \end{bmatrix}$$

	A	B	C	D
A	<u>ABA</u>	AB	<u>ABC</u>	AD
B	BA	BAB	BC	BD
C	<u>CBA</u>	CB	<u>CBC</u>	CD
D	DA	DAB	DC	<u>DAD</u>

After including node C (k=3)

$$Q_3 = \begin{bmatrix} A & B & C & D \\ A & 5 & 2 & 6 & 8 \\ B & 3 & 5 & 4 & 6 \\ C & 9 & 6 & 10 & 2 \\ D & 13 & 10 & 4 & 6 \end{bmatrix}$$

	A	B	C	D
A	ABA	AB	ABC	<u>ABCD</u>
B	BA	BAB	BC	<u>BCD</u>
C	<u>CBA</u>	CB	CBC	CD
D	<u>DCBA</u>	<u>DCB</u>	DC	<u>DCD</u>

After including node D (k=4)

$$Q_4 = \begin{bmatrix} A & B & C & D \\ A & 5 & 2 & 6 & 8 \\ B & 3 & 5 & 4 & 6 \\ C & 9 & 6 & 6 & 2 \\ D & 13 & 10 & 4 & 6 \end{bmatrix}$$

	A	B	C	D
A	ABA	AB	ABC	<u>ABCD</u>
B	BA	BAB	BC	BCD
C	<u>CBA</u>	CB	<u>CDC</u>	CD
D	<u>DCBA</u>	<u>DCB</u>	DC	DCD

$$Q_1(1,3) = \text{Minimum } [Q_0(1,3), Q_0(1,1)+Q_0(1,3)] = \text{Minimum } (\infty, \infty) = \infty$$

$$Q_1(2,2) = \text{Minimum } [Q_0(2,2), Q_0(2,1)+Q_0(1,2)] = \text{Minimum } (\infty, 3+2) = 5$$

$$Q_2(3,1) = \text{Minimum } [Q_1(3,1), Q_1(3,2)+Q_1(2,1)] = \text{Minimum } (\infty, 6+3) = 9$$

$$Q_3(1,4) = \text{Minimum } [Q_2(1,4), Q_2(1,3)+Q_2(3,4)] = \text{Minimum } (9, 6+2) = 8$$

$$Q_4(3,3) = \text{Minimum } [Q_3(3,3), Q_3(3,4)+Q_3(4,3)] = \text{Minimum } (10, 2+4) = 6$$

```

/*Program for Modified Warshall's algorithm to find shortest path matrix */
#include<stdio.h>
#define infinity 9999
#define MAX 20

main()
{
    int i,j,k,n;
    int adj[MAX][MAX],path[MAX][MAX];

    printf("Enter number of vertices : ");
    scanf("%d",&n);
    printf("Enter weighted matrix :\n");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&adj[i][j]);

    printf("Weighted matrix is :\n");
    display(adj,n);

    /*Replace all zero entries of adjacency matrix with infinity*/
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            if(adj[i][j]==0)
                path[i][j]=infinity;
            else
                path[i][j]=adj[i][j];

    for(k=0;k<n;k++)
    {
        printf("Q%d is :\n",k);
        display(path,n);
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
                path[i][j]=minimum( path[i][j] , path[i][k]+path[k][j] );
    }
    printf("Shortest path matrix Q%d is :\n",k);
    display(path,n);
}/*End of main() */

minimum(int a,int b)
{
    if(a<=b)
        return a;
    else
        return b;
}/*End of minimum()*/

```

```

display(int matrix[MAX][MAX],int n )
{
    int i,j;
    for(i=0;i<n;i++)
    {
        for(j=0;j<n;j++)
            printf("%7d",matrix[i][j]);
        printf("\n");
    }
}/*End of display()*/

```

Traversal In Graph-

Previously we have seen that traversal is nothing but visiting each node in some systematic approach. As we traverse a binary tree in preorder, postorder and inorder manner. Graph is represented by it's nodes and edges. So traversal of each node is the traversing in graph. There are two efficient techniques for traversing the graph. First is depth first search and second is breath first search. We maintain graph in an adjacency matrix or in adjacency list for nodes of graph and edge from one node to other. In breadth first search, we use queue for keeping nodes, which will be used for next processing, and in depth first search we use stack, which keeps the node for next processing.

Traversal in graph is different from traversal in tree or list because of the following reasons-

- (a) There is no first node or root node in a graph, hence the traversal can start from any node.
- (b) In tree or list when we start traversing from the first node, all the nodes are traversed but in graph only those nodes will be traversed which are reachable from the starting node. So if we want to traverse all the reachable nodes we again have to select another starting node for traversing the remaining nodes.
- (c) In tree or list while traversing we never encounter a node more than once but while traversing graph ,there may be a possibility that we reach a node more than once. So to ensure that each node is visited only once we have to keep the status of each node whether it has been visited or not.
- (d) In tree or list we have unique traversals. For example if we are traversing the tree in inorder there can be only one sequence in which nodes are visited. But in graph ,for the same technique of traversal there can be different sequences in which nodes can be visited .

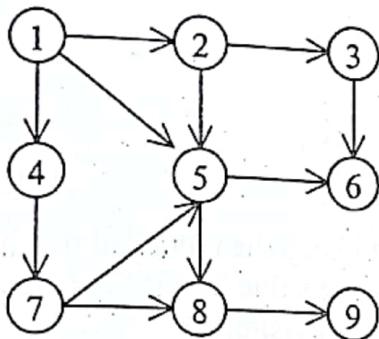
Breadth First Search-

This graph traversal technique uses queue for traversing all the nodes of the graph. In this, first we take any node as a starting node then we take all the nodes adjacent to that starting node. Similar approach we take for all other adjacent nodes, which are adjacent to the starting node and so on. We maintain the status of visited node in one array so that no node can be traversed again.

Suppose V_0 is our starting node and V_1, V_2, V_3 are nodes adjacent to it. V_{11}, V_{12}, V_{13} are nodes adjacent to V_1 . V_{21}, V_{22} are nodes adjacent to V_2 and V_{31} is adjacent to V_3 . So we will traverse V_0 first and then all nodes adjacent to V_2 and V_{31} is adjacent to V_3 . To traverse all nodes adjacent to V_1 i.e. V_{11}, V_{12}, V_{13} and then we will traverse nodes adjacent to V_2 i.e. V_{21}, V_{22} and then nodes adjacent to V_3 i.e. V_{31} . Traversal will be in following order-

$V_0 \ V_1 \ V_2 \ V_3 \ V_{11} \ V_{12} \ V_{13} \ V_{21} \ V_{22} \ V_{31}$

Let us take a graph and apply breadth first traversal to it.



Take the node 1 as starting node and start the traversal of graph.

First we will traverse the node 1. Then we will traverse all nodes adjacent to node 1 i.e. 2, 4, 5. Here we can traverse these three nodes in any order. So we can see that traversal is not unique. Suppose we traverse these nodes in order 2, 4, 5.

So now the traversal is -

1 2 4 5

Now first we traverse all the nodes adjacent to 2, then all the nodes adjacent to 4 then all the nodes adjacent to 5. So first we will traverse 3, then 7 and then 6, 8.

So now the traversal is -

1 2 4 5 3 7 6 8

Now we will traverse one by one all the nodes adjacent to nodes 3, 7, 6, 8. We can see that node adjacent to node 3 is node 6, but it has already been traversed so we will just ignore it and proceed further. Now node adjacent to node 7 is node 8 which has already been traversed so ignore it also. Node 6 has no adjacent nodes. Node 8 has node 9 adjacent to it which has not been traversed, so traverse node 9.

So now the traversal is -

1 2 4 5 3 7 6 8 9

This was the traversal when we take node 1 as the starting node. Suppose we take node 2 as the starting node. Then applying above technique, we will get the following traversal-

2 3 5 6 8 9

We can see that all the nodes are not traversed when starting node is 2. The nodes which are in the traversal are those nodes which are reachable from 2. See the different traversals when we take different starting nodes.

<u>Start Node</u>	<u>Traversal</u>
1	2 4 5 3 7 6 8 9
2	2 3 5 6 8 9
3	3 6
4	4 7 5 8 6 9
5	5 6 8 9
6	6
7	7 5 8 6 9
8	8 9
9	9

Breadth First Search through queue-

Take an array queue which will be used to keep the unvisited neighbours of the node.
 Take a boolean array visited which will have value true if the node has been visited and
 will have value false if the node has not been visited.
 Initially queue is empty and front = -1 and rear = -1
 Initially visited[i] = false where i = 1 to n, n is total number of nodes.

Procedure-

1. Insert starting node into the queue.
2. Delete front element from the queue and insert all its unvisited neighbours into the queue at the end ,and traverse them. Also make the value of visited array true for these nodes.
3. Repeat step 2 until the queue is empty.

Let us take node 1 as the starting node for traversal.

Step 1-

Insert starting node 1 into queue.

Traversed nodes = 1

visited[1]=true

front =0 rear=0 queue=1

Traversal = 1

Step 2-

Delete front element node 1 from queue and insert all its unvisited neighbours 2, 4, 5 into the queue .

Traversed nodes - 2, 4, 5

visited[2]=true, visited[4]=true, visited[5]=true

front =1 rear=3

queue = 2, 4, 5

Traversal = 1, 2, 4, 5

Step 3-

Delete front element node 2 from queue, traverse it's unvisited neighbour node 3 ,and insert it into the queue.
 Traversed nodes - 3
 visited[3] = true
 front = 2 rear = 4 queue = 4,5,3
 Traversal = 1, 2, 4, 5, 3

Step 4-

Delete front element node 4 from queue, traverse it's unvisited neighbour node 7 and insert it into the queue.
 Traversed nodes - 7
 visited[7] = true
 front = 3 rear = 5 queue = 5, 3, 7
 Traversal = 1, 2, 4, 5, 3, 7

Step 5-

Delete front element node 5 from the queue, traverse it's unvisited neighbours nodes 6, 8 and insert them into the queue.
 Traversed nodes - 6, 8
 visited[6] = true, visited[8] = true
 front = 4 rear = 7 queue = 3, 7, 6, 8
 Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 6-

Delete front element node 3 from queue. It has no unvisited neighbours.
 front = 5 rear = 7 queue = 7, 6, 8
 Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 7-

Delete front elemnt node 7 from queue. It has no unvisited neighbours .
 front = 6 rear = 7 queue = 6, 8
 Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 8-

Delete front element node 6 from queue. It has no unvisited neighbours.
 front = 7 rear = 7 queue = 8
 Traversal = 1, 2, 4, 5, 3, 7, 6, 8

Step 9-

Delete front element node 8 from queue, traverse it's unvisited neighbour node 9 and insert it into the queue.
 Traversed nodes - 9
 visited[9] = true
 front = 8 rear = 8 queue = 9
 Traversal = 1, 2, 4, 5, 3, 7, 6, 8, 9

Step 10-

Delete front element node 9 from queue. It has no unvisited neighbours
 front = 8 rear = 7 queue = EMPTY

Traversal=1,2,4,5,3,7,6,8,9

Since front > rear, hence now queue is empty so we will stop our process

As we have seen this process can traverse only those nodes which are reachable from the starting node. If we want to traverse all the nodes then we will take the unvisited node as starting node and again start this process from starting node. This process will continue until all the nodes are traversed.

The function for Breadth first search is as-

```
bfs(int v)
{
    int i,front,rear;
    int que[20];
    front=rear=-1;
    printf("%d ",v);
    visited[v]=true;
    rear++;
    front++;
    que[rear]=v;
    while(front<=rear)
    {
        v=que[front]; /* delete from queue */
        front++;
        for(i=1;i<=n;i++)
        {
            /* Check for adjacent unvisited nodes */
            if( adj[v][i]==1 && visited[i]==false)
            {
                printf("%d ",i);
                visited[i]=true;
                rear++;
                que[rear]=i;
            }
        }
    }/*End of while*/
}/*End of bfs()*/
```

Depth First Search- ✓

In Depth first search technique also we take one node as a starting node. Then go to the path which is from starting node and visit all the nodes which are in that path. When we reach at the last node then we traverse another path starting from that node. If there is no path in the graph from the last node then it returns to the previous node in the path and

traverse another path and so on. This technique uses stack.

Let us take the same graph and starting node as node 1.

First, we will traverse node 1. Then we will traverse any node adjacent to node 1. Here we traverse node 2, then traverse node 3, which is adjacent to node 2, then traverse adjacent node 6. Now there is no node adjacent to node 6, means we have reached the end of the path or a dead end from where we can't go forward. So we will move backward. Till now the traversal is -

1 2 3 6

Now we reach node 3, see if there is any node adjacent to it, and not traversed yet. There is no such node so we will reach node 2 and we see that node 5 is adjacent to it and not traversed yet. So we will traverse node 5 and move along the path which starts at node 5. So we will traverse 8 and then 9. Now there is no node adjacent to node 9 so we have reached the end of the path and now we will move backward. Till now the traversal is-

1 2 3 6 5 8 9

Now we reach node 8 and we see that there is no node adjacent to it and not traversed yet, so we reach node 5, here also we observe the same thing, so we reach node 6 here also there is no untraversed adjacent node, then we reach 3 and then 2. On reaching node 1 we see that node 4 is adjacent to it and has not been traversed. So we traverse it and move along a path which starts at node 4. So we traverse node 7. Now there is no untraversed node adjacent to node 7 so we have reached a dead end. We can't move forward but now we can't move backward also so we will stop our process. The traversal is as-

1 2 3 6 5 8 9 4 7

See the different traversals when we take different starting nodes.

<u>Start Node</u>	<u>Traversal</u>
1	1 2 3 6 5 8 9 4 7
2	2 3 6 5 8 9
3	3 6
4	4 7 5 6 8 9
5	5 6 8 9
6	6
7	7 5 6 8 9
8	8 9
9	9

1	1 2 3 6 5 8 9 4 7
2	2 3 6 5 8 9
3	3 6
4	4 7 5 6 8 9
5	5 6 8 9
6	6
7	7 5 6 8 9
8	8 9
9	9

Depth First Search through stack-

Take an array stack which will be used to keep the unvisited neighbours of the node. Take a boolean array visited which will have value true if the node has been visited and will have value false if the node has not been visited.

Initially stack is empty and top = -1
 Initially visited[i] = false where i = 1 to n, n is total number of nodes.

Procedure-

1. Push starting node into the stack.
2. Pop an element from the stack, if it has not been traversed then traverse it; if it has already been traversed then just ignore it. After traversing make the value of visited array true for this node.
3. Now push all the unvisited adjacent nodes of the popped element on stack. Push the element even if it is already on the stack.
4. Repeat steps 3 and 4 until stack is empty.

Suppose the starting node for traversal is 1.

Step 1-

Push node 1 into stack

top = 0 stack = 1

Step 2-

Pop node 1 from stack, traverse it:

Traversed node - 1

visited[1] = true

Now push all the unvisited adjacent nodes 5, 4, 2 of the popped element on stack.

top = 2 stack = 5, 4, 2

Traversal = 1

Step 3-

Pop the element node 2 from the stack, traverse it and push all its unvisited adjacent nodes 5, 3 on the stack

Traversed node - 2

visited[2] = true

top = 3 stack = 5, 4, 5, 3

Traversal = 1, 2

Step 4-

Pop the element node 3 from the stack, traverse it and push its unvisited adjacent node 6 on the stack

Traversed node - 3

visited[3] = true

top = 3 stack = 5, 4, 5, 6

Traversal = 1, 2, 3

Step 5-
 Pop the element node 6 from stack, traverse it. Node 6 has no adjacent nodes, so nothing is pushed
 Traversed node - 6
 visited [6] = true
 top = 2 stack = 5, 4, 5
 Traversal = 1, 2, 3, 6

Step 6-
 Pop the element node 5 from stack, traverse it and push its unvisited adjacent node 8 on the stack, node 6 is its adjacent node but it has been visited so it is not pushed.
 Traversed node - 5
 visited[5] = true
 top = 2 stack = 5, 4, 8
 Traversal = 1, 2, 3, 6, 5

Step 7-
 Pop the element node 8 from stack, traverse it and push its unvisited adjacent node 9 on the stack
 Traversed node - 8
 visited[8] = true
 top = 2 stack = 5, 4, 9
 Traversal = 1, 2, 3, 6, 5, 8

Step 8-
 Pop the element 9 from the stack and traverse it, 9 has no adjacent nodes
 visited[9]=true
 top=1 stack=5, 4
 Traversal=1,2,3,6,5,8,9

Step 9 -
 Pop the element node 4 from stack, traverse it and push its unvisited adjacent node 7 on the stack
 Traversed node - 4
 visited[4] = true
 top = 1 stack = 5, 7
 Traversal = 1, 2, 3, 6, 5, 8, 9, 4

Step 10-
 Pop the element node 7 from stack, traverse it, it has no unvisited adjacent nodes.
 Traversed node - 7
 visited[7]=true
 top = 0 stack = 5
 Traversal = 1, 2, 3, 6, 5, 8, 9, 4, 7

Step 11-
 Pop the element node 5 from stack, since visited[5] = true, so just ignore it.
 top = -1 Stack = EMPTY

Since the stack is empty so we will stop our process.

The function for Depth First Search is as-

```

dfs(int v)
{
    int i,stack[MAX],top=-1,pop_v,j,t;
    int ch;

    top++;
    stack[top]=v;s
    while (top>=0)
    {
        pop_v=stack[top];
        /*pop from stack*/
        top--;
        if( visited[pop_v]==false)
        {
            printf("%d ",pop_v);
            visited[pop_v]=true;
        }
        else
            continue;
        for(i=n;i>=1;i--)
        {
            if( adj[pop_v][i]==1 && visited[i]==false)
            {
                top++; /* push all unvisited neighbours of pop_v */
                stack[top]=i;
            }
        }
    }
}

/* Program for traversing a graph through BFS and DFS */
#include<stdio.h>
#define MAX 20

typedef enum boolean{false,true} bool;
int adj[MAX][MAX];
bool visited[MAX];
int n; /* Denotes number of nodes in the graph */
  
```

```
main()
{
    int i,v,choice;
    create_graph();
    while(1)
    {
        printf("\n");
        printf("1. Adjacency matrix\n");
        printf("2. Depth First Search using stack\n");
        printf("3. Depth First Search through recursion\n");
        printf("4. Breadth First Search\n");
        printf("5. Adjacent vertices\n");
        printf("6. Components\n");
        printf("7. Exit\n");
        printf("Enter your choice : ");
        scanf("%d",&choice);

        switch(choice)
        {
            case 1:
                printf("Adjacency Matrix\n");
                display();
                break;
            case 2:
                printf("Enter starting node for Depth First Search : ");
                scanf("%d",&v);
                for(i=1;i<=n;i++)
                    visited[i]=false;
                dfs(v);
                break;
            case 3:
                printf("Enter starting node for Depth First Search : ");
                scanf("%d",&v);
                for(i=1;i<=n;i++)
                    visited[i]=false;
                dfs_rec(v);
                break;
            case 4:
                printf("Enter starting node for Breadth First Search : ");
                scanf("%d", &v);
                for(i=1;i<=n;i++)
                    visited[i]=false;
                bfs(v);
                break;
            case 5:
                printf("Enter node to find adjacent vertices : ");
                scanf("%d", &v);
                printf("Adjacent Vertices are : ");
        }
    }
}
```

```

adj_nodes(v);
break;
case 6:
    components();
    break;
case 7:
    exit(1);
default:
    printf("Wrong choice\n");
    break;
}/*End of switch*/
}/*End of while*/
}/*End of main()*/
}

create_graph()
{
    int i,max_edges,origin,destin;

    printf("Enter number of nodes : ");
    scanf("%d",&n);
    max_edges=n*(n-1);

    for(i=1;i<=max_edges;i++)
    {
        printf("Enter edge %d( 0 0 to quit ) : ",i);
        scanf("%d %d",&origin,&destin);

        if((origin==0) && (destin==0))
            break;

        if( origin > n || destin > n || origin<=0 || destin<=0)
        {
            printf("Invalid edge!\n");
            i--;
        }
        else
        {
            adj[origin][destin]=1;
        }
    }/*End of for*/
}/*End of create_graph()*/
}

display()
{
    int ij;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
            printf("%4d",adj[i][j]);
    }
}

```

```

        printf("\n");
    }
/*End of display()*/
dfs_rec(int v)
{
    int i;
    visited[v]=true;
    printf("%d ",v);
    for(i=1;i<=n;i++)
        if(adj[v][i]==1 && visited[i]==false)
            dfs_rec(i);
}
/*End of dfs_rec()*/
dfs(int v)
{
    int i,stack[MAX],top=-1,pop_v,j,t;
    int ch;

    top++;
    stack[top]=v;
    while (top>=0)
    {
        pop_v=stack[top];
        top--; /*pop from stack*/
        if( visited[pop_v]==false)
        {
            printf("%d ",pop_v);
            visited[pop_v]=true;
        }
        else
            continue;

        for(i=n;i>=1;i--)
        {
            if( adj[pop_v][i]==1 && visited[i]==false)
            {
                top++; /* push all unvisited neighbours of pop_v */
                stack[top]=i;
            }
        }
    }
}
/*End of while*/
}/*End of dfs()*/
bfs(int v)
{
    int i,front,rear;
    int que[20];

```

```

front=rear=-1;
printf("%d ",y);
visited[v]=true;
rear++;
front++;
que[rear]=v;
while(front<=rear)
{
    v=que[front]; /* delete from queue */
    front++;
    for(i=1;i<=n;i++)
    {
        /* Check for adjacent unvisited nodes */
        if( adj[v][i]==1 && visited[i]==false)
        {
            printf("%d ",i);
            visited[i]=true;
            rear++;
            que[rear]=i;
        }
    }
}/*End of while*/
}/*End of bfs()*/
```



```

adj_nodes(int v)
{
    int i;
    for(i=1;i<=n;i++)
        if(adj[v][i]==1)
            printf("%d ",i);
    printf("\n");
}/*End of adj_nodes( )*/
```



```

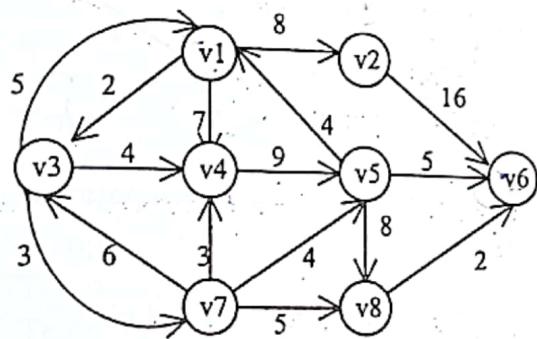
components()
{
    int i;
    for(i=1;i<=n;i++)
        visited[i]=false;
    for(i=1;i<=n;i++)
    {
        if(visited[i]==false)
            dfs_rec(i);
    }
    printf("\n");
}/*End of components()*/

```

Shortest Path Algorithm (Dijkstra)-

There are several cases in graph where we have a need to know the shortest path from one node to another node. General electric supply system and water distribution system also follow this approach. The best example we can take is of a railway track system. Suppose one person wants to go from one station to another then he needs to know the shortest path from one station to another. Here station represents the node and tracks represent edges. In computers, it is very useful in network for routing concepts.

There can be several paths for going from one node to another node, but the shortest path is that path in which the sum of weights of the included edges is the minimum. There are several algorithms to find out the shortest path. Here we will describe the Dijkstra's algorithm. Let us take a graph and find out the shortest path from the source node to destination node.



We label each node with dist, predecessor and status. Dist of node represents the shortest distance of that node from the source node, and predecessor of a node represents the node which precedes the given node in the shortest path from source. Status of a node can be permanent or temporary. In the figure, shaded circles represent permanent nodes. Making a node permanent means that it has been included in the shortest path. Temporary nodes can be relabeled if required but once a node is made permanent, it can't be relabeled.

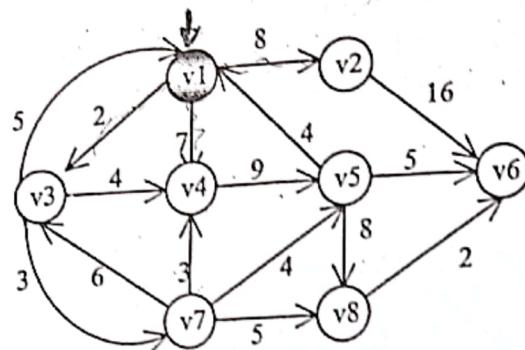
The procedure is as -

1. Initially make source node permanent and make it the current working node. All other nodes are made temporary
2. Examine all the temporary neighbours of the current working node and after checking the condition for minimum weight , relabel the required nodes.
3. From all the temporary nodes, find out the node which has minimum value of dist , make this node permanent and now this is our current working node.
4. Repeat steps 2 and 3 until destination node is made permanent.

Suppose the source node is v1

v1 is the current working node

Node	dist	pred	status
v1	0	0	Perm
v2	∞	0	Temp
v3	∞	0	Temp
v4	∞	0	Temp
v5	∞	0	Temp
v6	∞	0	Temp
v7	∞	0	Temp
v8	∞	0	Temp



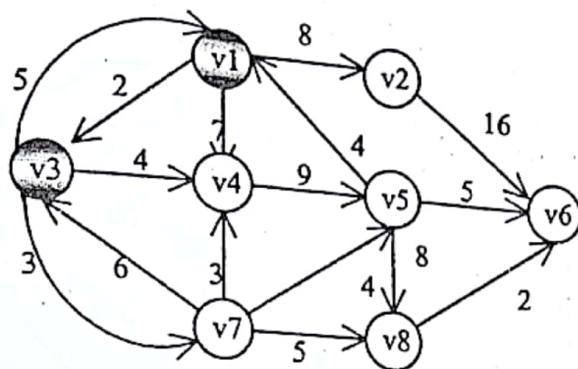
Now we will check adjacent nodes of v1, which are temporary also. Here v2,v3,v4 are adjacent to v1 and are temporary.

$$\begin{aligned}
 v2.dist &> v1.dist + \text{distance}(v1, v2) & \infty > 0 + 8 & \text{relabel } v2 \\
 v3.dist &> v1.dist + \text{distance}(v1, v3) & \infty > 0 + 2 & \text{relabel } v3 \\
 v4.dist &> v1.dist + \text{distance}(v1, v4) & \infty > 0 + 7 & \text{relabel } v4
 \end{aligned}$$

Now we will change the pred and dist of v2, v3 and v4.

Node	dist	pred	status
v1	0	0	Perm
v2	8	v1	Temp
v3	2	v1	Temp
v4	7	v1	Temp
v5	∞	0	Temp
v6	∞	0	Temp
v7	∞	0	Temp
v8	∞	0	Temp

Now from all the temporary nodes find out the node that has smallest distance from source i.e. smallest value of dist. Here v3 has the smallest value of dist in all temporary nodes so make it permanent and now v3 is our current working node.



Adjacent nodes of v_3 are v_4 and v_7 and both are temporary.

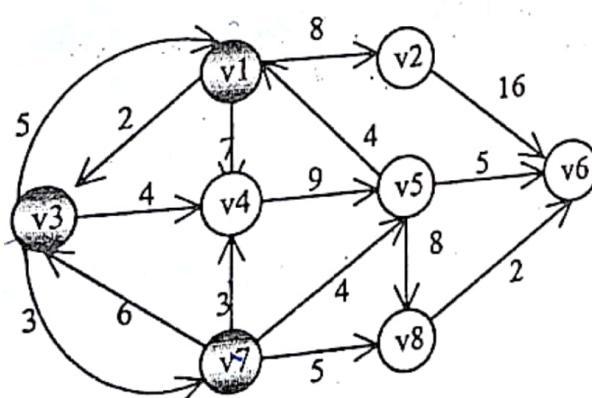
$$v_4.\text{dist} > v_3.\text{dist} + \text{distance}(v_3, v_4) \quad 7 > 2+4 \text{ relabel } v_4$$

$$v_7.\text{dist} > v_3.\text{dist} + \text{distance}(v_3, v_7) \quad \infty > 2+3 \text{ relabel } v_7$$

Now we will change the pred and dist of v_4 and v_7 .

Node	dist	pred	status
v_1	0	0	Perm
v_2	8	v_1	Temp
v_3	2	v_1	Perm
v_4	6	v_3	Temp
v_5	∞	0	Temp
v_6	∞	0	Temp
v_7	5	v_3	Temp
v_8	∞	0	Temp

Now from all the temporary nodes, v_7 has smallest value of dist so make it permanent and now v_7 is our current working node



Adjacent nodes of v_7 are v_3, v_4, v_5 Since v_3 is permanent we will not relabel it,

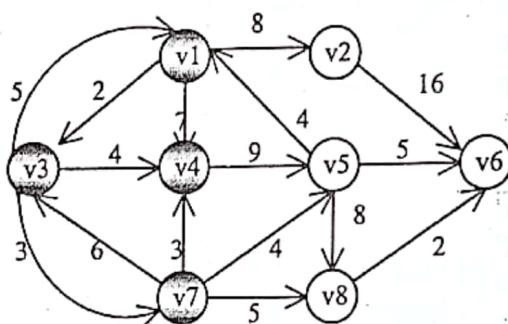
$$v_4.\text{dist} \leq v_7.\text{dist} + \text{distance}(v_7, v_4) \quad 6 \leq 5+3 \text{ Don't relabel } v_4$$

$$v_5.\text{dist} > v_7.\text{dist} + \text{distance}(v_7, v_5) \quad \infty > 5+4 \text{ relabel } v_5$$

Now pred and dist of v5 will be changed.

Node	dist	pred	status
v1	0	0	Perm
v2	8	v1	Temp
v3	2	v1	Perm
v4	6	v3	Temp
v5	9	v7	Temp
v6	∞	0	Temp
v7	5	v3	Perm
v8	∞	0	Temp

Now from all temporary nodes v4 has smallest value of dist so make it permanent. Now v4 is the current working node.

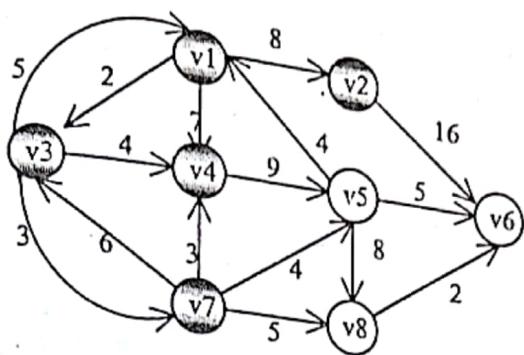


Adjacent nodes of v4 are v5 and v1. Node v1 is already permanent. So we will check for v5 only.

$$v5.\text{dist} < v4.\text{dist} + \text{distance}(v4, v5) \quad 9 < 6+9 \quad \text{Don't relabel v5}$$

Node	dist	pred	status
v1	0	0	P
v2	8	v1	Temp
v3	2	v1	Perm
v4	6	v3	Perm
v5	9	v7	Temp
v6	∞	0	Temp
v7	5	v3	Perm
v8	∞	0	Temp

Now from all temporary nodes v2 has smallest value of dist so make it permanent. Now v2 is our current working node.



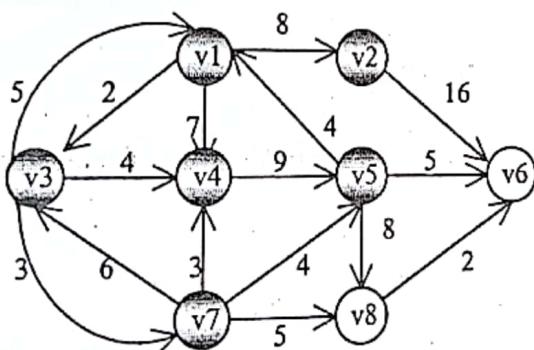
Node adjacent to v2 is v6.

$$v6.dist < v2.dist + \text{distance}(v2, v6) \quad \infty > 8 + 16 \text{ relabel } v6$$

Now pred and dist of v6 will be changed,

Node	dist	pred	status
v1	0	0	Perm
v2	8	v1	Perm
v3	2	v1	Perm
v4	6	v3	Perm
v5	9	v7	Temp
v6	24	v2	Temp
v7	5	v3	Perm
v8	∞	0	Temp

Now from all temporary nodes v5 has smallest value of dist, so make it permanent .Now v5 is our current working node.



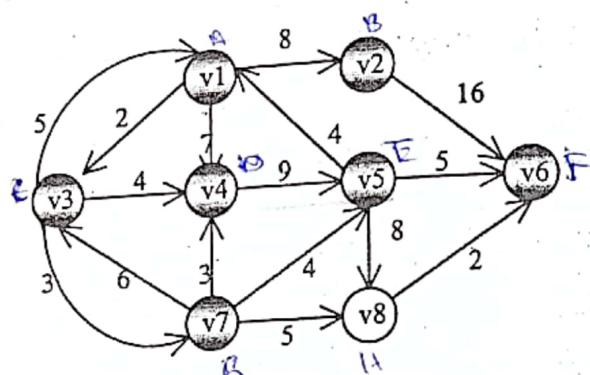
Adjacent nodes of v5 are v6 and v8.

$$\begin{aligned} v6.dist &< v5.dist + \text{distance}(v5, v6) & 24 > 9 + 5 \text{ relabel } v6 \\ v8.dist &< v5.dist + \text{distance}(v5, v8) & \infty > 9 + 8 \text{ relabel } v8 \end{aligned}$$

Now pred and dist of v6 and v8 will be changed.

Node	dist	pred	status
v1	0	0	Perm
v2	8	v1	Perm
v3	2	v1	Perm
v4	6	v3	Perm
v5	9	v7	Perm
v6	14	v5	Temp
v7	5	v3	Perm
v8	17	v5	Temp

Now from all temporary nodes v6 has smallest value of dist so make it permanent. Since v6 is our destination node and it has been made permanent, so we will stop our process.



We can find out the shortest path from the last table. Start from the destination node and keep on seeing the predecessors until we get source node as a predecessor.

pred of v6 is v5
pred of v5 is v7
pred of v7 is v3
pred of v3 is v1

So the path is v1 -- v3 -- v7 -- v5 -- v6

```
/* Program of shortest path between two node in graph using Djikstra algorithm */
#include<stdio.h>
#define MAX 10
#define TEMP 0
#define PERM 1
#define infinity 9999

struct node
{
    int predecessor;
    int dist; /*minimum distance of node from source*/
    int status;
};
```

```
int adj[MAX][MAX];
int n;
void main()
{
    int i,j;
    int source,dest;
    int path[MAX];
    int shortdist,count;

    create_graph();
    printf("The adjacency matrix is :\n");
    display();

    while(1)
    {
        printf("Enter source node(0 to quit) : ");
        scanf("%d",&source);
        printf("Enter destination node(0 to quit) : ");
        scanf("%d",&dest);

        if(source==0 || dest==0)
            exit(1);

        count = findpath(source,dest,path,&shortdist);
        if(shortdist!=0)
        {
            printf("Shortest distance is : %d\n", shortdist);
            printf("Shortest Path is : ");
            for(i=count;i>1;i--)
                printf("%d->",path[i]);
            printf("%d",path[i]);
            printf("\n");
        }
        else
            printf("There is no path from source to destination node\n");
    }/*End of while*/
}/*End of main()*/
create_graph()
{
    int i,max_edges,origin,destin,wt;
    printf("Enter number of vertices : ");
    scanf("%d",&n);
    max_edges=n*(n-1);
    for(i=1;i<=max_edges;i++)
    {
        printf("Enter edge %d(0 to quit) : ",i);
        scanf("%d %d",&origin,&destin);
    }
}
```

```

        if((origin==0) && (destin==0))
            break;
        printf("Enter weight for this edge : ");
        scanf("%d",&wt);
        if( origin > n || destin > n || origin<=0 || destin<=0)
        {
            printf("Invalid edge!\n");
            i--;
        }
        else
            adj[origin][destin]=wt;
    }/*End of for*/
}/*End of create_graph()*/



display()
{
    int i,j;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
            printf("%3d",adj[i][j]);
        printf("\n");
    }
}/*End of display()*/
int findpath(int s,int d,int path[MAX],int *sdist)
{
    struct node state[MAX];
    int i,min,count=0,current,newdist,u,v;
    *sdist=0;
    /* Make all nodes temporary */
    for(i=1;i<=n;i++)
    {
        state[i].predecessor=0;
        state[i].dist = infinity;
        state[i].status = TEMP;
    }

    /*Source node should be permanent*/
    state[s].predecessor=0;
    state[s].dist = 0;
    state[s].status = PERM;

    /*Starting from source node until destination is found*/
    current=s;
    while(current!=d)
    {
        for(i=1;i<=n;i++)
        {
}

```

```

graph LR
    /*Checks for adjacent temporary nodes */
    if ( adj[current][i] > 0 && state[i].status == TEMP )
        newdist=state[current].dist + adj[current][i];
        /*Checks for Relabeling*/
        if( newdist < state[i].dist )
        {
            state[i].predecessor = current;
            state[i].dist = newdist;
        }
    }

}/*End of for*/



/*Search for temporary node with minimum distand make it current node*/
min=infinity;
current=0;
for(i=1;i<=n;i++)
{
    if(state[i].status == TEMP && state[i].dist < min)
    {
        min = state[i].dist;
        current=i;
    }
}
}/*End of for*/



if(current==0) /*If Source or Sink node is isolated*/
    return 0;
state[current].status=PERM;
}/*End of while*/



/* Getting full path in array from destination to source */
while( current!=0 )
{
    count++;
    path[count]=current;
    current=state[current].predecessor;
}

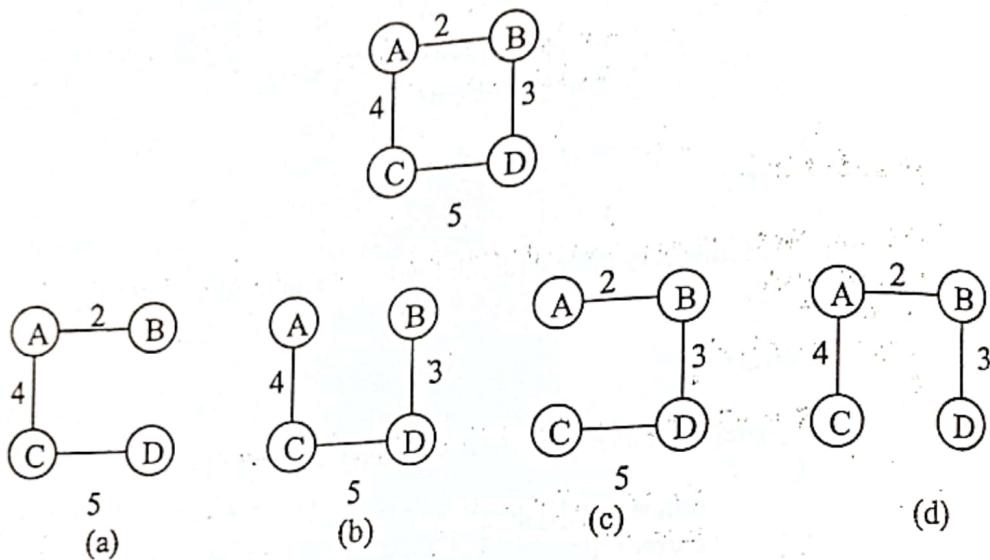
/*Getting distance from source to destination*/
for(i=count;i>1;i--)
{
    u=path[i];
    v=path[i-1];
    *sdist+= adj[u][v];
}

return (count);
}/*End of findpath()*/

```

Spanning Tree-

A spanning tree of a connected graph G contains all the nodes and has the edges, which connects all the nodes. So number of edges will be 1 less than the number of nodes. Let us take a connected graph G -



Here all these trees are spanning tree. The number of edges is 3, which is 1 less than the number of nodes.

If a graph is not connected, i.e. a graph with n nodes has edges less than $n-1$ then no spanning tree is possible.

DFS and BFS spanning trees are the spanning trees which are obtained by DFS and BFS traversals.

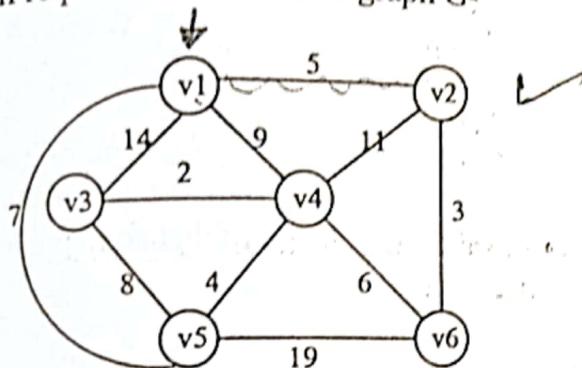
Minimum Spanning Tree-

Minimum spanning tree is the spanning tree in which the sum of weights on edges is minimum. In the above example (d) is the minimum spanning tree with weight 9. There are many ways for creating minimum spanning tree but the most famous methods are Prim's and Kruskal algorithm.

Prim's Algorithm-

In this algorithm, we start with any node and add the other node in spanning tree on the basis of weight of edge connecting to that node. Suppose we start with the node v_1 then we have a need to see all the connecting edges and which edge has minimum weight. Then we will add that edge and node to the spanning tree. So here we will be in need to know the nodes in spanning tree and weights on the edge connecting to other nodes because if two nodes v_1 & v_2 are in spanning tree and both have edge connecting to v_3 then edge which has minimum weight will be added in spanning tree. So we can say it creates growing tree and each step adds one node in spanning tree.

The method of making minimum spanning tree from Prim's algorithm is like Dijkstra's algorithm with some modifications. Here also we will label each node with dist, predecessor and status of each node. Dist of node will represent shortest distance of that node from its predecessor while in Dijkstra's algorithm dist represented distance of node from the source. Status of a node can be permanent or temporary. Whenever a node is made permanent, we include it in the spanning tree and make it the current working node. Once a node is made permanent, it is not relabeled. Only temporary nodes will be relabeled if required. Let us take a graph G-

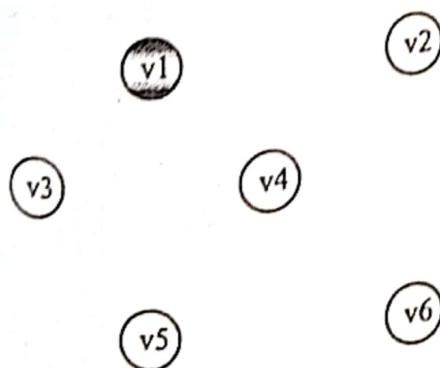


The procedure is same as in Dijkstra's algorithm. In Dijkstra's algorithm we stopped our process when destination node was made permanent, while here we will stop our process when all the nodes will be made permanent.

Initially make all the nodes temporary.

Nodes	Dist	Pred	Status
v1	∞	0	Temp
v2	∞	0	Temp
v3	∞	0	Temp
v4	∞	0	Temp
v5	∞	0	Temp
v6	∞	0	Temp

Make first node v1 permanent.



$v_2.dist > Distance(v_1, v_2)$	$\infty > 5$	Relabel v_2
$v_3.dist > Distance(v_1, v_3)$	$\infty > 14$	Relabel v_3
$v_4.dist > Distance(v_1, v_4)$	$\infty > 9$	Relabel v_4
$v_5.dist > Distance(v_1, v_5)$	$\infty > 7$	Relabel v_5

Nodes	Dist	Pred	Status
v_1	0	0	Perm
v_2	5	v_1	Temp
v_3	14	v_1	Temp
v_4	9	v_1	Temp
v_5	7	v_1	Temp
v_6	∞	0	Temp

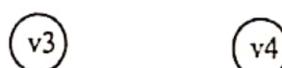
Now from all the temporary nodes v_2 has the minimum value of dist so make it permanent and now v_2 is our current working node.



$v_6.dist > Distance(v_2, v_6)$	$\infty > 3$	Relabel v_6
---------------------------------	--------------	---------------

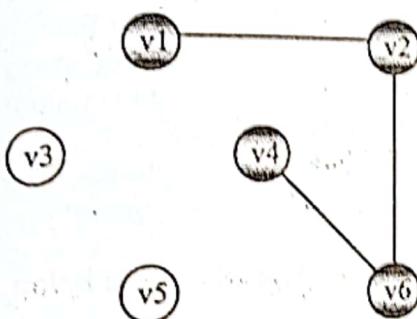
Nodes	Dist	Pred	Status
v_1	0	0	Perm
v_2	5	v_1	Perm
v_3	14	v_1	Temp
v_4	9	v_1	Temp
v_5	7	v_1	Temp
v_6	3	v_2	Temp

From all temporary nodes v_6 has the minimum value of dist so make it permanent and now v_6 is the current working node.



$v_4.dist > Distance(v_6, v_4)$	$9 > 6$	Rlabel v_4
Nodes	Dist	Pred
v_1	0	0
v_2	5	v_1
v_3	14	v_1
v_4	6	v_6
v_5	7	v_1
v_6	3	v_2

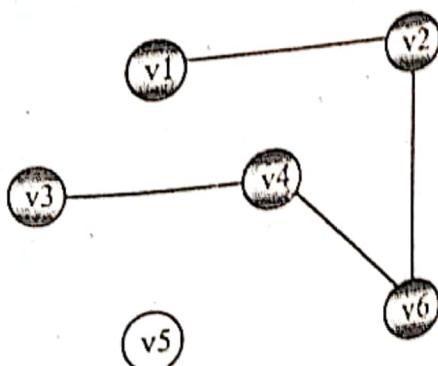
From all temporary nodes v_4 has the minimum value of dist so make it permanent and now v_4 is the current working node.



$v_3.dist > Distance(v_4, v_3)$	$14 > 2$	Rlabel v_3
$v_5.dist > Distance(v_4, v_5)$	$7 > 4$	Rlabel v_5

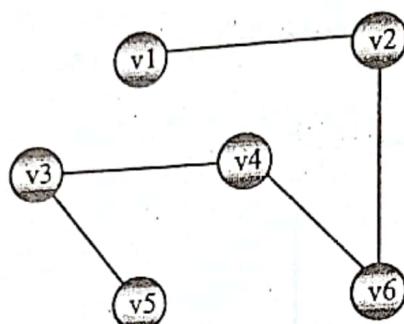
Nodes	Dist	Pred	Status
v_1	0	0	Perm
v_2	5	v_1	Perm
v_3	2	v_4	Temp
v_4	6	v_6	Perm
v_5	4	v_4	Temp
v_6	3	v_2	Perm

From all temporary nodes v_3 has the minimum value of dist so make it permanent and now v_3 is the current working node.



Nodes	Dist	Pred	Status
v1	0	0	Perm
v2	5	v1	Perm
v3	2	v4	Perm
v4	6	v6	Perm
v5	4	v4	Temp
v6	3	v2	Perm

Make v5 permanent.



Now we have a complete minimum spanning tree. The edges that belong to minimum spanning tree are

(v1,v2), (v3,v4), (v3,v5), (v2,v6), (v4,v6)

Weight of minimum spanning tree will be-

$$5 + 2 + 8 + 6 + 3 = 24$$

/ Program for creating minimum spanning tree from Prim's algorithm */*

```

#include<stdio.h>
#define MAX 10
#define TEMP 0
#define PERM 1
#define FALSE 0
#define TRUE 1
#define infinity 9999

struct node
{
    int predecessor;
    int dist; /*Distance from predecessor */
    int status;
};

struct edge
{
    int u;
    int v;
};
  
```

```

int adj[MAX][MAX];
int n;
main()
{
    int i,j;
    int path[MAX];
    int wt_tree,count;
    struct edge tree[MAX];

    create_graph();
    printf("Adjacency matrix is :\n");
    display();
    count = maketree(tree,&wt_tree);
    printf("Weight of spanning tree is : %d\n", wt_tree);
    printf("Edges to be included in spanning tree are :\n");
    for(i=1;i<=count;i++)
    {
        printf("%d->",tree[i].u);
        printf("%d\n",tree[i].v);
    }
/*End of main()*/
create_graph()
{
    int i,max_edges,origin,destin,wt;
    printf("Enter number of vertices : ");
    scanf("%d",&n);
    max_edges=n*(n-1)/2;
    for(i=1;i<=max_edges;i++)
    {
        printf("Enter edge %d(0 0 to quit) : ",i);
        scanf("%d %d",&origin,&destin);
        if((origin==0) && (destin==0))
            break;
        printf("Enter weight for this edge : ");
        scanf("%d",&wt);
        if( origin > n || destin > n || origin<=0 || destin<=0)
        {
            printf("Invalid edge!\n");
            i--;
        }
        else
        {
            adj[origin][destin]=wt;
            adj[destin][origin]=wt;
        }
    }
/*End of for*/
}

```

```

        if(i<n-1)
        {
            printf("Spanning tree is not possible\n");
            exit(1);
        }
    }/*End of create_graph( )*/

    display()
    {
        int i,j;
        for(i=1;i<=n;i++)
        {
            for(j=1;j<=n;j++)
                printf("%3d",adj[i][j]);
            printf("\n");
        }
    }/*End of display( )*/

    int maketree(struct edge tree[MAX],int *weight)
    {
        struct node state[MAX];
        int i,k,min,count,current,newdist;
        int m;
        int u1,v1;
        *weight=0;
        /*Make all nodes temporary*/
        for(i=1;i<=n;i++)
        {
            state[i].predecessor=0;
            state[i].dist = infinity;

            state[i].status = TEMP;
        }
        /*Make first node permanent*/
        state[1].predecessor=0;
        state[1].dist = 0;
        state[1].status = PERM;

        /*Start from first node*/
        current=1;
        count=0; /*count represents number of nodes in tree */
        while( all_perm(state) != TRUE ) /*Loop till all the nodes become PERM*/
        {
            for(i=1;i<=n;i++)
            {
                if( adj[current][i] > 0 && state[i].status == TEMP )

```

```

        if( adj[current][i] < state[i].dist )
        {
            state[i].predecessor = current;
            state[i].dist = adj[current][i];
        }
    }/*End of for*/

    /*Search for temporary node with minimum distance
    and make it current node*/
    min=infinity;
    for(i=1;i<=n;i++)
    {
        if(state[i].status == TEMP && state[i].dist < min)
        {
            min = state[i].dist;
            current=i;
        }
    }/*End of for*/

    state[current].status=PERM;

    /*Insert this edge(u1,v1) into the tree */
    u1=state[current].predecessor;
    v1=current;
    count++;
    tree[count].u=u1;
    tree[count].v=v1;
    /*Add wt on this edge to weight of tree */
    *weight=*weight+adj[u1][v1];

}/*End of while*/
return (count);
}/*End of maketree( )*/
}

/*This function returns TRUE if all nodes are permanent*/
int all_perm(struct node state[MAX])
{
    int i;
    for(i=1;i<=n;i++)
        if( state[i].status == TEMP )
            return FALSE;
    return TRUE;
}/*End of all_perm( )*/

```

Kruskal's Algorithm

In this method initially we take n distinct trees for all n nodes of the graph. Then we take edges in ascending order, whenever we insert an edge two trees will be joined.

First we examine all the edges one by one starting from the smallest edge. To decide whether the selected edge should be included in the spanning tree or not, we'll examine the two nodes connecting the edge. If the two nodes belong to the same tree then we will not include the edge in the spanning tree, since the two nodes are in the same tree, they are already connected and adding this edge would result in a cycle. Since we are making a tree, so there should not be any cycles. We will insert an edge in the spanning tree only if its nodes are in different trees.

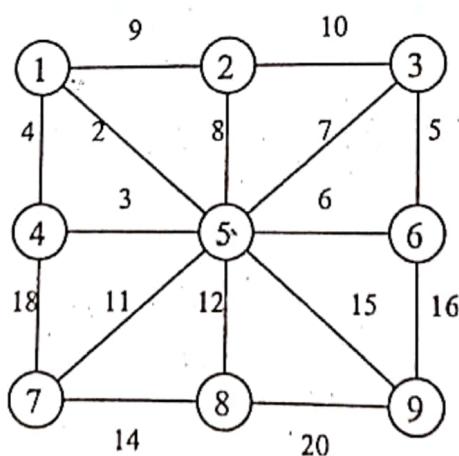
Now we'll see how to decide whether two nodes are in the same tree or not. We will keep record of the father of every node. Since this is a tree, every node will have only one distinct father. Initially father of all nodes will be zero. We will recognize each tree by a root node and a node will be a root if its father is zero. For finding out, to which tree a node belongs, we will find out the root of that tree. So we will traverse all the ancestors of the node till we reach a node whose father is zero. This will be the root of the tree to which the node belongs.

Now we know the root of both nodes of edge, if roots are same means both nodes are in the same tree and are already connected. If the roots are different, then we'll insert this edge into the spanning tree and we'll join the two trees, which are having these two nodes. For joining the two trees, we will make root of one tree as the father of root of another tree.

After joining two trees all the nodes of both trees will be connected and have the same root.

To obtain the edges in ascending order we can insert them in a priority queue.

Let us take a graph and make a minimum spanning tree by Kruskal's algorithm.



Initially father of every node is zero, and hence every node is a tree, and the root of that tree is the node itself.

node	1	2	3	4	5	6	7	8	9
father	0	0	0	0	0	0	0	0	0

Step 1 - Edge selected is 1-5 wt=2
 $n_1=1$ root_{n1}=1 $n_2=5$ root_{n2}=5
Roots are different so edge is inserted in the spanning tree.
Join the two trees father[5]=1
node 1 2 3 4 5 6 7 8 9
father 0 0 0 0 1 0 0 0 0

Step 2 - Edge selected is 4-5 wt=3
 $n_1=4$ root_{n1}=4 $n_2=5$ root_{n2}=1
Edge Inserted, father[1]=4
node 1 2 3 4 5 6 7 8 9
father 4 0 0 0 1 0 0 0 0

Step 3 - Edge selected is 1-4 wt=4
 $n_1=1$ root_{n1}=4 $n_2=4$ root_{n2}=4
Roots are same so edge is not inserted in the spanning tree.
node 1 2 3 4 5 6 7 8 9
father 4 0 0 0 1 0 0 0 0

Step 4 - Edge selected is 3-6 wt=5
 $n_1=3$ root_{n1}=3 $n_2=6$ root_{n2}=6
Edge inserted father[6]=3
node 1 2 3 4 5 6 7 8 9
father 4 0 0 0 1 3 0 0 0

Step 5 - Edge selected is 5-6 wt=6
 $n_1=5$ root_{n1}=4 $n_2=6$ root_{n2}=3
Edge inserted father[3]=4
node 1 2 3 4 5 6 7 8 9
father 4 0 4 0 1 3 0 0 0

Step 6 - Edge selected is 3-5 wt=7
 $n_1=3$ root_{n1}=4 $n_2=5$ root_{n2}=4
Edge not inserted
node 1 2 3 4 5 6 7 8 9
father 4 0 4 0 1 3 0 0 0

Step 7 - Edge selected is 2-5 wt=8
 $n_1=2$ root_{n1}=2 $n_2=5$ root_{n2}=4
Edge inserted, father[4]=2
node 1 2 3 4 5 6 7 8 9
father 4 0 4 2 1 3 0 0 0

Step 8 -

Edge selected is 1-2 wt=9
 n1=1 root_n1=2 n2=2 root_n2=2
 Not inserted
 node 1 2 3 4 5 6 7 8 9
 father 4 0 4 2 1 3 0 0 0

Step 9 -

Edge selected is 2-3 wt=10
 n1=2 root_n1=2 n2=3 root_n2=2
 Edge not inserted
 node 1 2 3 4 5 6 7 8 9
 father 4 0 4 2 1 3 0 0 0

Step 10 -

Edge selected is 5-7 wt=11
 n1=5 root_n1=2 n2=7 root_n2=7
 Edge inserted, father[7]=2
 node 1 2 3 4 5 6 7 8 9
 father 4 0 4 2 1 3 2 0 0

Step 11 -

Edge selected is 5-8 wt=12
 n1=5 root_n1=2 n2=8 root_n2=8
 Edge inserted, father[8]=2
 node 1 2 3 4 5 6 7 8 9
 father 4 0 4 2 1 3 2 2 0

Step 12 -

Edge selected is 7-8 wt=14
 n1=7 root_n1=2 n2=8 root_n2=2
 Edge not inserted
 node 1 2 3 4 5 6 7 8 9
 father 4 0 4 2 1 3 2 2 0

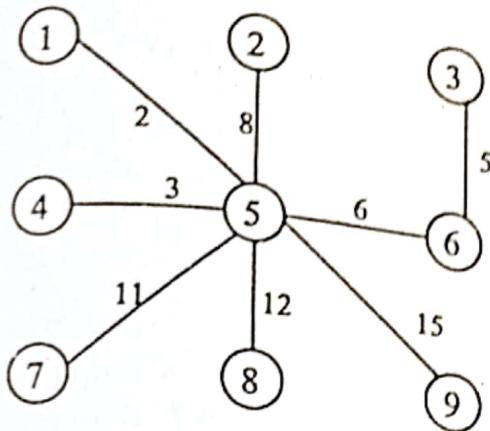
Step 13 -

Edge selected is 5-9 wt=15
 n1=5 root_n1=2 n2=9 root_n2=9
 Edge inserted, father[9]=2
 node 1 2 3 4 5 6 7 8 9
 father 4 0 4 2 1 3 2 2 2

The minimum spanning tree should contain $n-1$ edges where n is the number of nodes in the graph. This graph contains 9 nodes so after including 8 edges in the spanning tree, we will not examine other edges of the graph and stop our process.

We can also see that now there is only one tree which is rooted at node 2. We can take any node of the tree and its root will come out to be node 2.

The resulting minimum spanning tree will be-



Edges included in this spanning tree are (1,5), (4,5), (3,6), (5,6), (2,5), (5,7), (5,8), (5,9)
 Weight of this spanning tree is $2 + 3 + 5 + 6 + 8 + 11 + 12 + 15 = 62$

** Program for creating a minimum spanning tree from Kruskal's algorithm */*

```
#include<stdio.h>
#define MAX 20
```

```
struct edge
{
    int u;
    int v;
    int weight;
    struct edge *link;
}*front=NULL;
```

```
int father[MAX]; /*Holds father of each node */
struct edge tree[MAX]; /* Will contain the edges of spanning tree */
int n; /*Denotes total number of nodes in the graph */
int wt_tree=0; /*Weight of the spanning tree */
int count=0; /* Denotes number of edges included in the tree */
```

```
*Functions */
void make_tree();
void insert_tree(int i,int j,int wt);
void insert_pque(int i,int j,int wt);
struct edge *del_pque();
```

```
main()
```

```
{  

    int i;  

    create_graph();  

    make_tree();  

    printf("Edges to be included in spanning tree are :\n");  

    for(i=1;i<=count;i++)  

    {  

        printf("%d->",tree[i].u);  

    }
```

```

        printf("%d\n",tree[i].v);
    }
    printf("Weight of this minimum spanning tree is : %d\n", wt_tree);
}/*End of main( )*/



create_graph( )
{
    int i,wt,max_edges,origin,destin;

    printf("Enter number of nodes : ");
    scanf("%d",&n);
    max_edges=n*(n-1)/2;
    for(i=1;i<=max_edges;i++)
    {
        printf("Enter edge %d(0 0 to quit): ",i);
        scanf("%d %d",&origin,&destin);
        if( (origin==0) && (destin==0) )
            break;
        printf("Enter weight for this edge : ");
        scanf("%d",&wt);
        if( origin > n || destin > n || origin<=0 || destin<=0 )
        {
            printf("Invalid edge!\n");
            i--;
        }
        else
            insert_pque(origin,destin,wt);
    }/*End of for*/
    if(i<n-1)
    {
        printf("Spanning tree is not possible\n");
        exit(1);
    }
}/*End of create_graph( )*/



void make_tree( )
{
    struct edge *tmp;
    int node1,node2,root_n1,root_n2;

    while( count < n-1 ) /*Loop till n-1 edges included in the tree*/
    {
        tmp=del_pque( );
        node1=tmp->u;
        node2=tmp->v;
        printf("n1=%d ",node1);
        printf("n2=%d ",node2);
}

```

```

        while( node1 > 0 )
        {
            root_n1=node1;
            node1=father[node1];
        }
        while( node2 > 0 )
        {
            root_n2=node2;
            node2=father[node2];
        }
        printf("rootn1=%d ",root_n1);
        printf("rootn2=%d\n",root_n2);

        if(root_n1!=root_n2)
        {
            insert_tree(tmp->u,tmp->v,tmp->weight);
            wt_tree=wt_tree+tmp->weight;
            father[root_n2]=root_n1;
        }
    }/*End of while*/
}/*End of make_tree()*/



/*Inserting an edge in the tree */
void insert_tree(int i,int j,int wt)
{
    printf("This edge inserted in the spanning tree\n");
    count++;
    tree[count].u=i;
    tree[count].v=j;
    tree[count].weight=wt;
}/*End of insert_tree()*/



/*Inserting edges in the priority queue */
void insert_pque(int i,int j,int wt)
{
    struct edge *tmp,*q;

    tmp = (struct edge *)malloc(sizeof(struct edge));
    tmp->u=i;
    tmp->v=j;
    tmp->weight = wt;

    /*Queue is empty or edge to be added has weight less than first edge*/
    if( front == NULL || tmp->weight < front->weight )
    {
        tmp->link = front;
        front = tmp;
    }
}

```

```

else
{
    q = front;
    while( q->link != NULL && q->link->weight <= tmp->weight )
        q=q->link;
    tmp->link = q->link;
    q->link = tmp;
    if(q->link == NULL) /*Edge to be added at the end*/
        tmp->link = NULL;
}
/*End of else*/
}/*End of insert_pque()*/
/*Deleting an edge from the priority queue*/
struct edge *del_pque()
{
    struct edge *tmp;
    tmp = front;
    printf("Edge processed is %d->%d %d\n",tmp->u,tmp->v,tmp->weight);
    front = front->link;
    return tmp;
}/*End of del_pque()*/

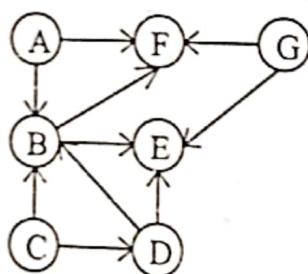
```

Topological Sorting-

There are so many applications where execution of one task is necessary before starting another task. Like understanding of 'C' language, programming is necessary before starting 'Data Structure Through C'. Similarly in booting process of computer or printing process from printer. Same in this book also after understanding of tree, we are going to heap sort or binary tree sort.

In any directed graph which has no cycle, topological sort gives the sequential order of all the nodes $x, y \in G$ and x comes before y in sequential order if a path exists from x to y . So this sequential order will indicate the dependency of one task on another.

Let us take a directed graph G .



Adjacency list of G will be-

A → B, F
 B → E, F
 C → B, D
 D → B, E
 E →
 F →
 G → E, F

Now we have a need to find the way to get topological sorting. We should first take those nodes which have no predecessors. It can be chosen very easily. The nodes which have indegree (number of edges coming to that node) zero will be those nodes. So topological sorting can be defined as-

1. Take all the nodes which have zero indegree.
2. Delete those nodes and edges going from those nodes.
3. Do the same process again until all the nodes are deleted.

This operation will need a queue where all the nodes which have indegree zero will be first added then deleted. Every time we need to get the information of indegree of every

node. One array is also required where all the nodes will be in sequence of deletion from queue. Let us take a graph and find the topological sorting. Here we are taking a queue for maintaining the zero indegree elements and array topSort for sequence of deleted elements from queue.

Step1-

Indegree of nodes are-
 $A=0, B=3, C=0, D=1, E=3, F=3, G=0$

Step2-

(i) Taking all the nodes, which have zero indegree.

A, C, G

(ii) Add all zero indegree nodes to queue

Queue: A, C, G

Front=1, Rear=3

topSort:

(iii) Delete the node A and edges going from A.

Front=2, Rear=3

topSort:A

Queue: C, G

(iv) Now the indegree of nodes will be-

$B=2, D=1, E=3, F=2$

Step3-

(i) Delete the node C and edges going from C

Front=3, Rear=3

topSort:A, C

Queue: G

(ii) Now the indegree of nodes will be-

$B=1, D=0, E=3, F=2$

topSort:A, C

Step 4-

(i) Add D to queue

Queue: G,D

Front=3, Rear=4

(ii) Delete the node G and edges going from G.

Queue: D

Front=4, Rear=4

(iii) Now the indegree of nodes will be-

B=1, E=2, F=2

topSort: A,C

Step 5-

(i) Delete the node D and edges going from D.

Queue:

Front=5, Rear=4

(ii) Now the indegree of nodes will be-

B=0, E=1, F=2

topSort: A,C,G

Step 6-

(i) Add B to queue

Queue: B

Front = 5, Rear = 5

(ii) Delete the node B and edges going from B.

Queue:

Front = 6, Rear = 5

(iii) Now the indegree of nodes will be-

E = 0, F = 1

topSort: A,C,G,D

Step 7-

(i) Add E to queue

Queue: E

Front = 6, Rear = 5

(ii) Delete the node E and edges going from E

Queue:

Front = 7, Rear = 5

(iii) Now the indegree of nodes will be-

F = 0

topSort: A,C,G,D,B

Step 8-

(i) Add F to queue

Queue: F

Front = 7, Rear = 7

(ii) Delete the node F and edges going from F

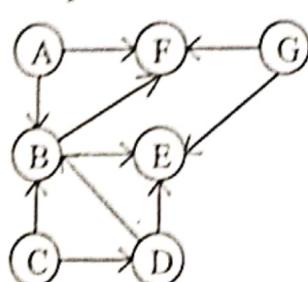
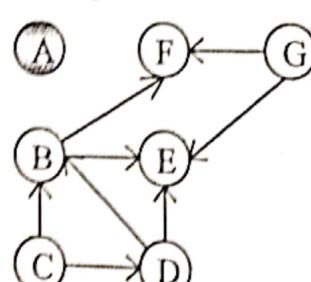
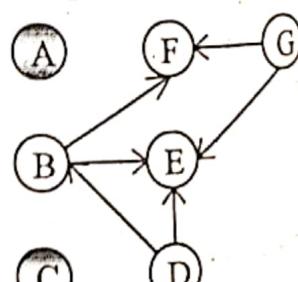
Queue:

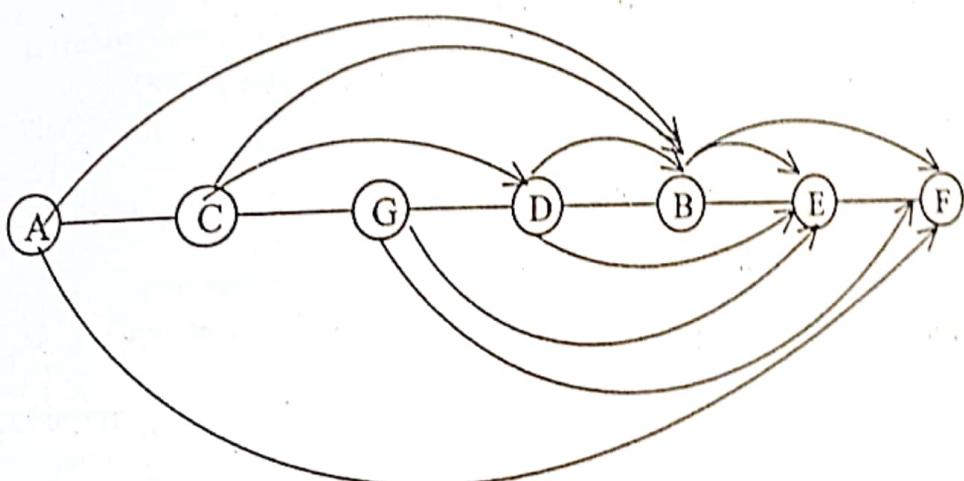
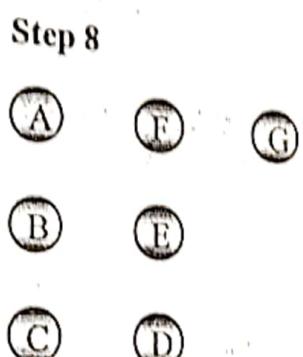
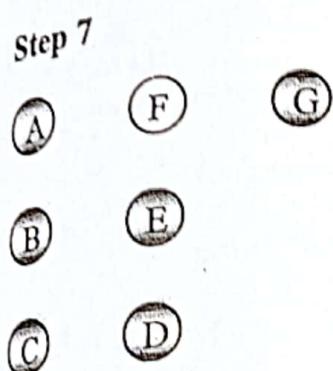
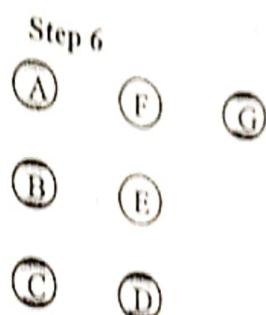
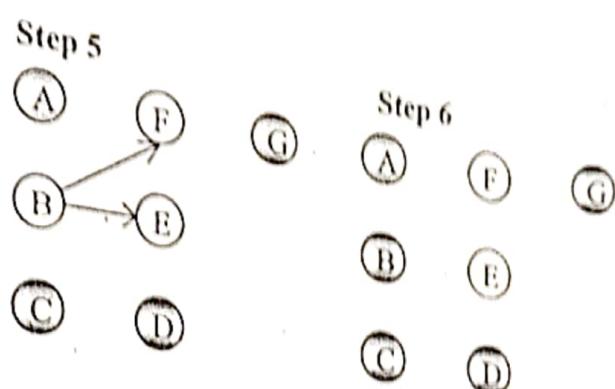
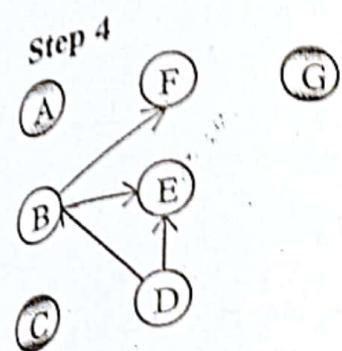
Front = 8, Rear = 7

topSort: A,C,G,D,B,E

topSort: A,C,G,D,B,E,F

Now we have no more nodes in the graph. So the topological sorting of graph will be-
A, C, G, D, B, E, F

Step 1**Step 2****Step 3**



** Program for topological sorting */*

```
#include<stdio.h>
```

```
#define MAX 20
```

```
int n,adj[MAX][MAX];
int front=-1,rear=-1,queue[MAX];
```

```
main()
```

```
{ int i,j=0,k;
    int topsort[MAX],indeg[MAX];
```

```
create_graph( );
```

```

printf("The adjacency matrix is :\n");
display();
/*Find the indegree of each node*/
for(i=1;i<=n;i++)
{
    indeg[i]=indegree(i);
    if( indeg[i]==0 )
        insert_queue(i);
}
while(front<=rear) /*Loop till queue is not empty */
{
    k=delete_queue();
    topsort[j++]=k; /*Add node k to topsort array*/
    /*Delete all edges going from node k */
    for(i=1;i<=n;i++)
    {
        if( adj[k][i]==1 )
        {
            adj[k][i]=0;
            indeg[i]=indeg[i]-1;
            if(indeg[i]==0)
                insert_queue(i);
        }
    }/*End of for*/
}/*End of while*/

printf("Nodes after topological sorting are :\n");
for(i=0;i<j;i++)
    printf( "%d ",topsort[i] );
printf("\n");
}/*End of main()*/



create_graph()
{
    int i,max_edges,origin,destin;

    printf("Enter number of vertices : ");
    scanf("%d",&n);
    max_edges=n*(n-1);

    for(i=1;i<=max_edges;i++)
    {
        printf("Enter edge %d(0 0 to quit): ",i);
        scanf("%d %d",&origin,&destin);

        if((origin==0) && (destin==0))
            break;
    }
}

```

```

if( origin > n || destin > n || origin<=0 || destin<=0)
{
    printf("Invalid edge!\n");
    i--;
}
else
    adj[origin][destin]=1;
}/*End of for*/
}/*End of create_graph( )*/

display()
{
    int i,j;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=n;j++)
            printf("%3d",adj[i][j]);
        printf("\n");
    }
}/*End of display( )*/

insert_queue(int node)
{
    if (rear==MAX-1)
        printf("Queue Overflow\n");
    else
    {
        if (front==-1) /*If queue is initially empty */
            front=0;
        rear=rear+1;
        queue[rear] = node ;
    }
}/*End of insert_queue( )*/

delete_queue()
{
    int del_item;
    if (front == -1 || front > rear)
    {
        printf("Queue Underflow\n");
        return ;
    }
    else
    {
        del_item=queue[front];
        front=front+1;
        return del_item;
    }
}/*End of delete_queue( ) */

```

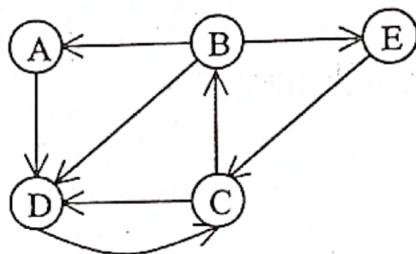
```

int indegree(int node)
{
    int i,in_deg=0;
    for(i=1;i<=n;i++)
        if( adj[i][node] == 1 )
            in_deg++;
    return in_deg;
}/*End of indegree()*/

```

Exercise-

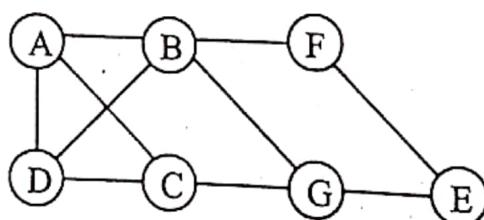
- Ans* 1. Find out the adjacency matrix and adjacency list for the graph given below-



Modify the adjacency matrix and adjacency list after each of the operations given below.

- (i) Delete the edge $D \rightarrow C$.
- (ii) Insert an edge $A \rightarrow E$.
- (iii) Delete the node E from the graph.
- (iv) Insert a new node F in the graph.
- (v) Insert an edge $C \rightarrow F$.

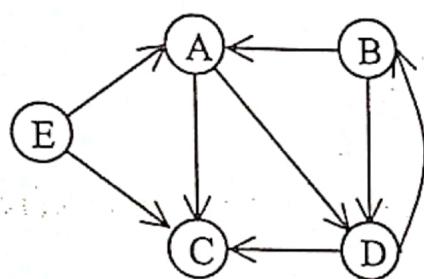
2. Consider the graph given below and answer the following questions



- (a) Find all the simple paths from node A to node E . Compute the lengths of each of these paths and find out the shortest path.
- (b) Find out the degree of each node in the graph. Is there any isolated node in the graph?
- (c) Is the above graph a connected graph?
- (d) Are there any cycles in the above graph, if yes then how many?
- (e) Is the above graph a tree?

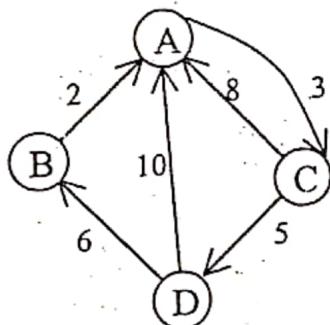
- (f) How many maximum edges can be there in this graph?
 (g) Find the diameter of the above graph. (the diameter of a graph is defined as the length of the longest path in the graph)

3. Consider the directed graph given below and answer the following questions.



- (i) Write down the indegree and outdegree of each node in the graph.
 (ii) Which nodes are sources and sinks in this graph?
 (iii) Is the graph weakly connected or strongly connected?
 (iv) Is the above graph a complete graph?
 (v) Are there any cycles in the above graph?

4. Consider the graph given below.

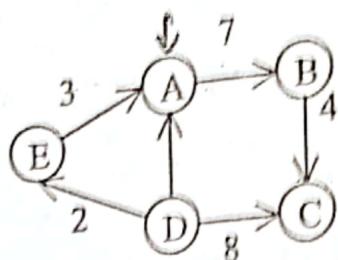


- (i) Find out the adjacency matrix for this graph.
 (ii) Find out the path matrix by powers of adjacency matrix.
 (iii) Find out the path matrix by Warshall's algorithm.

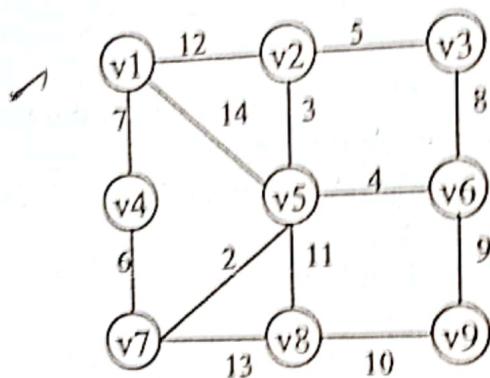
5. Find out the shortest path matrix by modified warshalls algorithm for graph whose weighted adjacency matrix is given below

$$W = \begin{array}{c} \begin{array}{cccc} & A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} \end{array} \left[\begin{array}{cccc} 1 & 0 & 3 & 6 \\ 0 & 5 & 0 & 0 \\ 8 & 0 & 7 & 5 \\ 4 & 2 & 0 & 9 \end{array} \right]$$

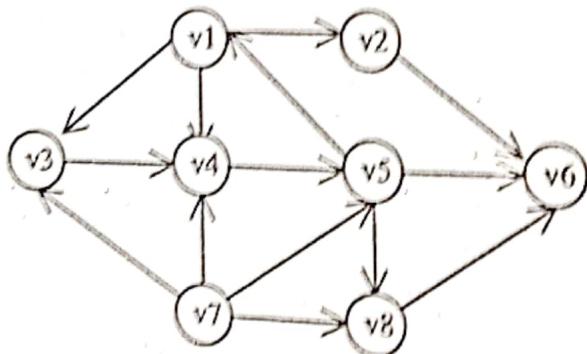
6. Find out the shortest path between all pair of nodes in the given graph by djikstra algorithm.



7. Find out the minimum spanning tree of given graph by prim's and kruskal's algorithm.



8. Find out the DFS and BFS traversals of following graph taking v1 as the starting node.



10. Write a program which takes input as an adjacency matrix and find out whether there are any loops in the graph.

11. Write a program which finds out whether a graph is regular or not.

12. Write a program to find the indegree and outdegree of a node in a directed graph.

13. Write a program to find out the number of source nodes and sink nodes in a graph.

14. Write a program which finds out all the successors and predecessors of a given node.

Chapter 8

Searching, Hashing and Storage Management

Sequential searching-

Sequential searching is nothing but searching an element in linear way. This can be in array or in linked list. We have a need to start the search from beginning and scan the elements one by one until the end of array or linked list. If search is successful then it will return the location of element, otherwise it will return the failure notification. Let us take sequential search in array-

arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	10	15	8	9	25	30			

First we have a need to put unique value at the end of array then the steps can be defined as-

Step1- INDEX = 0

Step2- Scan each element of array one by one.

Step3-

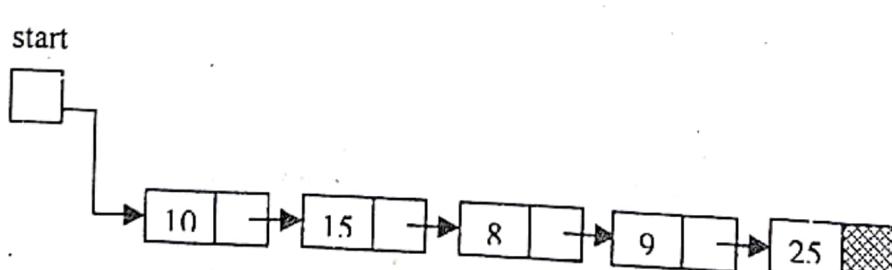
(i) If match occurs then return the index value.

(ii) Otherwise INDEX = INDEX + 1

Step4- Repeat the same process until unique value comes in scanning.

Step5- Return the failure notification.

Let us take sequential search in linked list-



Here we can define the steps as-

Step1- Take a pointer of node type and initialize it with start.
 $\text{ptr} = \text{start};$

Step2- Scan each node of the linked list by traversing the list with the help of ptr.
 $\text{ptr} = \text{ptr} \rightarrow \text{link};$

Step3- If match occurs then return.

Step4- Repeat the same process until NULL comes in scanning.
 Step5- Return the failure notification.

/*Write a program for sequential searching in an array*/

```
#include<stdio.h>
main()
{
    int arr[20],n,i,item;
    printf("How many elements you want to enter in the array : ");
    scanf("%d",&n);

    for(i=0; i < n;i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d", &arr[i]);
    }

    printf("Enter the element to be searched : ");
    scanf("%d",&item);

    for(i=0;i < n;i++)
    {
        if(item == arr[i])
        {
            printf("%d found at position %d\n",item,i+1);
            break;
        }
    }/*End of for*/
    if(i == n)
        printf("Item %d not found in array\n",item);
}
```

Analysis-

As the sequential search behaviour of this algorithm, it searches element one by one in linear manner. In the case of successful search, it will search the element upto the position in the array where it finds the element in array. Suppose it finds the element at the first position of array then it will behave like best case. In the case of unsuccessful search it will search up to the last element of array. But it can be reduced by adding one technique. Suppose all the elements in array are in sorted order then we can check the condition, if element in array is greater then return the search failure notification. So we will be in need to search the element in array up to the position where it finds the array element value less than or equal to the element to be searched.

Binary Search-

The sequential search situation will be in worst case if the element is at the end of the list. Suppose a telephone directory has a name 'Suresh' at the end then sequential search will not be efficient in this condition. For eliminating this problem, we have one efficient searching technique called binary search. The condition for binary search is that all the data should be in sorted array. We compare the element with middle element of the array. If it is less than the middle element then we search it in the left portion of the array and if it is greater than the middle element then search will be in the right portion of the array. Now we will take that portion only for search and compare with middle element of that portion. This process will be in iteration until we find the element or middle element has no left or right portion to search.

For this we will take 3 variables Start, End and Middle, which will keep track of the status of Start, End and Middle value of the portion of the array, in which we will search the element. The value of the middle will be as-

$$\text{Middle} = \text{Start} + \text{End} / 2$$

Let us take a sorted array of 10 elements-

arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	10	18	19	20	25	30	49	57	64	72

Suppose the element we are searching is 49.

Iteration 1-

Start=0	End=9	Middle=(0+9)/2=4								
arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	10	18	19	20	25	30	49	57	64	72

Now we will compare middle element which is 25 with 49
Since 49>25

Hence we will search the element in the right portion of the array.
Hence now

$$\text{Start}=\text{Middle}+1=5$$

End will be same as earlier.

Iteration 2-

Start=5 arr	End=9 [0]	Middle=(5+9)/2=7 [1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	10	18	19	20	25	30	49	(57)	64	72

Since $49 < 57$

Hence we will search the element in the left portion of the array.

Hence now

End=Middle=7

Start will be same as earlier.

Iteration 3-

Start=5 arr	End=7 [0]	Middle=(5+7)/2=6 [1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	10	18	19	20	25	30	(49)	57	64	72

Now we find the index of the element in array and through index we can access information easily.

Now let us take a case where search fails. Suppose we are searching the element 47 which is not present in array.

arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	10	18	19	20	25	30	49	57	64	72

Iteration 1-

Start=0 arr	End=9 [0]	Middle=(0+9)/2=4 [1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	10	18	19	20	(25)	30	49	57	64	72

Now we will compare middle element which is 25 with 47

Since $47 > 25$

Hence we will search the element in the right portion of the array.

Hence now

Start=Middle+1=5

End will be same as earlier.

Iteration 2-

Start=5	End=9	Middle=(5+9)/2=7	arr [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
10	18	19	20	25	30	49	57	64	72			

Since $47 < 57$

Hence we will search the element in the left portion of the array.

Hence now

End=Middle=7

Start will be same as earlier.

Iteration 3-

Start=5	End=7	Middle=(5+7)/2=6	arr [0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
10	18	19	20	25	30	49	57	64	72			

Now the element has no left and right portion to search. So it will return the search failure notation.

```
/*Program for binary search*/
#include <stdio.h>

main( )
{
    int arr[20],start,end,middle,n,i,item;

    printf("How many elements you want to enter in the array : ");
    scanf("%d",&n);
    for(i=0; i < n; i++)
    {
        printf("Enter element %d : ",i+1);
        scanf("%d",&arr[i]);
    }
    printf("Enter the element to be searched : ");
    scanf("%d",&item);
    start=0;
    end=n-1;
    middle=(start+end)/2;
    while(item != arr[middle] && start <= end)
    {
        if(item > arr[middle])
            start=middle+1;
    }
}
```

```

        else
            end=middle-1;
            middle=(start+end)/2;
    }
    if(item==arr[middle])
        printf("%d found at position %d\n",item,middle+1);
    if(start>end)
        printf("%d not found in array\n",item);
/*End of main( )*/

```

Hashing-

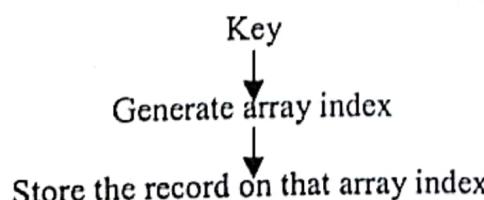
We have seen different searching techniques where search time is basically dependent on the number of elements. Sequential search, binary search and all the search trees are totally dependent on number of elements and so many key comparisons are also involved. Now our need is to search the element in constant time and less key comparisons should be involved. Suppose all the elements are in array size N. Let us take all the keys are unique and in the range 0 to N-1. Now we are storing the records in array based on the key where array index and keys are same. Now we can access the record in constant time and there no key comparisons are involved. Let us take the 5 records where keys are-

9 4 6 7 2

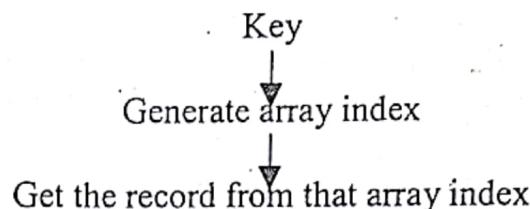
It will be stored in array as-

arr	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		2		4		6	7			9

Here we can see the record which has key value can be directly accessed through array index arr[2]. Now the idea that comes in picture is hashing where we will convert the key into array index and put the records in array and in the same way for searching the record, convert the key into array index and get the records from array. This can be described as-
For storing record



For accessing record-



The generation of array index uses hash function, which converts the keys into array index and the array which supports hashing for storing record or searching record called hash table. So we can say each key is mapped on a particular array index through hash function. If each key is mapped on a unique hash table address then this situation is ideal situation but there may be possibility that our hash function is generating same hash table address for different keys, this situation is called collision. So our hash function should generate unique address but it is not possible, only one thing we can do is to take the good hash function for minimizing collision. But still now we have a need to resolve these collisions with some technique. So we have a need to do two important things-

1. Choosing a good hash function which ensures minimum collision.
2. Resolving the collision.

Let us take a simple hash table.

0	
1	
2	011 Delhi
3	
4	022 Calcutta
5	
6	033 Bombay
7	
8	044 Chennai
9	

Here STD codes of the cities are keys and hash function maps that into the address 2,4,6,8 by just adding the digits of the key.

So we can define hashing as-

$$h(k) \rightarrow a$$

where h is the hash function on particular key and generated address a belongs to hash table.

Choosing a hash function

Now we know the hash function is generating an address location in hash table from key. It works like mapping interface between key and hash table. Basically we have two criteria for choosing good hash function-

1. It should be easy to compute.
2. It should generate unique address but it is an ideal situation so it should generate address with minimum collision.

We have so many methods to choose hash function but we have a need to select the hash function keeping in mind these two criteria. As we can say that take some rightmost digit of key and take it as address or add all the digits of the key and take it as address but what will be implication of this method. Is it collision free. Nobody knows, if we have key in advance then only we can say this will be best hash function for this. But that situation rarely comes. So we have no chance. We will certainly get some collision but that can be minimized through good hash function. We have so many cases for handling key itself.. It can be integer binary string and dangerous floating point number also. Let us consider the case we are taking the address by the modulus operation on key with table size (key modulus size) then here it depends on the size of the table if it's a power of two then sure we will get more collision .So we should take the table size prime number for minimizing collision.

Now we will see some techniques for choosing hash function and analyze them based on easily compatible and minimum collision.

1. Truncation Method-

This is the easiest method for computing key. Here we take only part of the key as address, it can be some rightmost digit or leftmost digit. Let us take some 8 digit keys

82394561, 87139465, 83567271, 85943228

Now we can take the number of rightmost digits or leftmost digits based on the size of hash table. Suppose table size is 100 then take the 2 rightmost digits for getting the hash table address. Suppose we are taking rightmost digits address then the addresses will be 61,65,71 and 28. We can see that it is very easy to compute but chance of collision is more because last two digits can be same in different keys.

2. Mid square Method-

In mid square method we square the key, after getting number we take some digits from the middle of that number as an address. Let us take some 4 digit number as a key-

1337 1273 1391 1026