



NEW
SYLLABUS

DATA STRUCTURE USING C

SHILPA PAWALE



 NIRALI[®]
PRAKASHAN
MANAGERS OF INDIAN LIBRARIES



A Text Book Of

DATA STRUCTURE USING C

**For
BBA (Computer Applications)
Formerly Known as BCA
Semester - III**

As per New Revised Syllabus

SHILPA PAWALE
M.C.S., M.C.A.
Lecturer, P.V.G.'s College of Science, Pune

Price ₹ 290.00



N2193



Syllabus ...

Unit 1: Basic Concept and Introduction to Data Structure	[9 Lectures]
1.1 Pointers and dynamic memory allocation	
1.2 Algorithm-Definition and characteristics	
1.3 Algorithm Analysis	
• Space Complexity	
• Time Complexity	
• Asymptotic Notation	
Introduction to Data structure	
1.5 Types of Data structure	
1.6 Abstract Data Types (ADT)	
Introduction to Arrays and Structure	
1.7 Types of array and Representation of array	
1.8 Polynomial	
• Polynomial Representation	
• Evaluation of Polynomial	
• Addition of Polynomial	
1.9 Self Referential Structure	
Unit 2: Searching and Sorting Techniques	[9 Lectures]
2.1 Linear Search	
2.2 Binary Search (Recursive, Non-Recursive)	
2.3 Bubble Sort	
2.4 Insertion Sort	
2.5 Selection Sort	
2.6 Quick Sort	
2.7 Heap Sort (No Implementation)	
2.8 Merge Sort	
2.9 Analysis of all Sorting Techniques	
Unit 3: Linked List	[10 Lectures]
3.1 Introduction	
3.2 Static and Dynamic Representation	
3.3 Types of linked List	
• Singly Linked list(All type of operation)	
• Doubly Linked list (Create , Display)	
• Circularly Singly Linked list (Create, Display)	
3.4 Circularly Doubly Linked list (Create, Display)	

**Unit 4: Stack and Queue****[9 Lectures]**

- 4.1 Introduction stack
- 4.2 Static and Dynamic Representation
- 4.3 Primitive Operations on stack
- 4.4 Application of Stack
- 4.5 Evaluation of postfix and prefix expression
- 4.6 Conversion of expressions- Infix to prefix and Infix to postfix

Queue

- 4.7 Introduction queue
- 4.8 Static and Dynamic Representation
- 4.9 Primitive Operations on Queue
- 4.10 Application of Queue
- 4.11 Type of Queue
 - Circular Queue
 - De Queue
 - Priority Queue

Unit 5: Trees**[7 Lectures]**

- 5.1 Introduction and Definitions
- 5.2 Terminology
- 5.3 Static and Dynamic Representation
- 5.4 Types of tree
- 5.5 Operations on Binary Tree and Binary Search Tree
- 5.6 Tree Traversal
 - Inorder, Preorder, Postorder (Recursive and Iterative)
- 5.7 AVL Tree

Unit 6: Graphs**[4 Lectures]**

- 6.1 Representation
 - Adjacency Matrix
 - List
- 6.2 In degree, out degree of graph
- 6.3 Graph operation
 - DFS, BFS
- 6.4 Spanning Tree





Contents ...

1. Basic Concept and Introduction to Data Structure	1.1 – 1.42
2. Searching and Sorting Techniques	2.1 – 2.30
3. Linked List	3.1 – 3.54
4. Stack and Queue	4.1 – 4.58
5. Trees	5.1 – 5.60
6. Graphs	6.1 – 6.38
University Question Papers	P.1 – P.62



Chapter 1 ...

Basic Concept and Introduction to Data Structure

Contents ...

- 1.1 Pointers and Dynamic Memory Allocation
 - 1.2 Algorithm-Definition and Characteristics
 - 1.3 Algorithm Analysis
 - 1.3.1 Space Complexity
 - 1.3.2 Time Complexity
 - 1.3.3 Asymptotic Notation
 - 1.4 Introduction to Data structure
 - 1.5 Types of Data structure
 - 1.6 Abstract Data Types (ADT)
 - 1.7 Introduction to Array and Structure
 - 1.8 Types of array and Representation of array
 - 1.9 Polynomials
 - 1.9.1 Polynomial Representation
 - 1.9.2 Evaluation of Polynomial
 - 1.9.3 Polynomial Addition using Array
 - 1.10 Self Referential Structure
 - Practice Questions
 - University Questions and Answers
-

1.1 POINTERS AND DYNAMIC MEMORY ALLOCATION

[April 15, 17, Oct. 17]

1.1.1 Pointers

- Any variable declaration specifies 3 things,
 - (a) Memory space required to store the value of the variable (depending on the data type of a variable).
 - (b) Name of the variable which is associated with the memory location allocated to that variable
 - (c) Value of the variable.

For example: `int a = 30;`

(1.1)



Memory map for 'a' is as shown in Fig. 1.1.

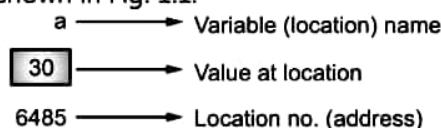


Fig. 1.1: Memory Allocation

- To get the address of a variable '&' operator called as 'address of' operator, is used. For example, `&a` expression gives address of the variable 'a' i.e. 6485.
 - Pointer is a variable which stores the address of another variable.
 - '*' operator, called 'value at address' operator, is used with pointer variable. It is also known as dereferencing or indirection operator.
 - For example,

```
int a = 30;
int *ptr;
ptr = &a;
```
- In above example, 'ptr' is a pointer variable. Statement `int *ptr` tells the compiler that `ptr` is used to store the address of an integer value i.e. it points to an integer.
- Statement `ptr = &a` is used to store the address of integer variable 'a' into `ptr`. The memory map for above example is given in Fig. 1.2.

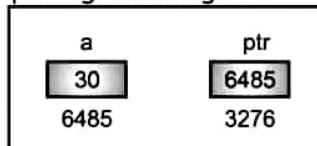


Fig. 1.2: Memory Map

- As pointer is a variable, it also has a datatype and requires a memory space for storage.
- Since memory address is always a positive integer value, printer is a variable of type unsigned integer.
- Note that, it is possible to store address of any variable of any datatype such as char, float, double, structure, union, etc.

For example,

```
char ch = 'A';
char *ptr1;
ptr1=&ch;
```

Here `ptr1` points to a character type variable.

- To get the value of a variable, '*' operator is used.

For example,

```
#include<stdio.h>
main()
{
    int a = 30;
    int *ptr;
    ptr = &a;
    printf("\n Value of a=%d", a);
    printf("\n Value of a=%d", *ptr);
}
```

Output:

```
Value of a = 30
Value of a = 30
```



- It is possible to store address of pointer variable into another pointer variable.

For example,

```
int a = 30;  
int *ptr = &a;  
int **ptr1 = &ptr;
```

The memory map for above example is given in Fig. 1.3.

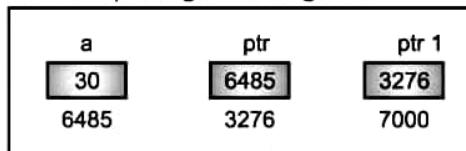


Fig. 1.3: Memory map of pointer

*ptr and **ptr1 will refer to the same value, 30.

1.1.2 Dynamic Memory Allocation

[April 15, 17; Oct. 16, 17]

- Many languages permit a programmer to specify an array's size at run time. Such languages have the ability to calculate and assign, during program execution, the memory space required by the variables in a program.
- The process of allocating memory at run time is known as Dynamic Memory Allocation.
- Although C does not inherently have this facility, there are four library routines known as "memory management functions" that can be used for allocating and freeing memory during program execution.
- They are listed in following Table. These functions help us to build complex application programs that use the available memory intelligently.

malloc()	Allocates a specified number of bytes in memory. Returns a pointer to the beginning of the allocated block.
calloc()	Similar to malloc(), but initializes the allocated bytes to zero. This function also allows us to allocate memory for more than one object at a time.
realloc()	Changes the size of a previously allocated block.
free()	Frees up memory that was previously allocated with malloc(), calloc(), or realloc().

Memory Allocation Process

- Before we discuss these functions, let us look at the memory allocation process associated with a C program.



- Fig. 1.4 shows the conceptual view of storage of a C program in memory.

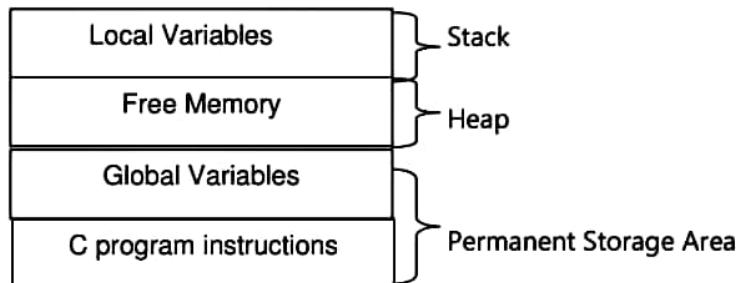


Fig. 1.4: Storage of a C Program

- The program instructions and global and static variables are stored in a region known as permanent storage area and the local variables are stored in another area called as stack.
- The memory space between these two regions is available for dynamic allocation during execution of the program. This free memory region is called the Heap.
- The size of the heap keeps changing when program is executed due to creation and death of the variables that are local to functions and blocks. Therefore, it is possible to encounter memory "Overflow" during dynamic allocation process.
- In such situations, the memory allocation functions return a NULL pointer (when they fail to locate enough memory requested).

1. Allocating a Block of Memory using `malloc()`

- `malloc()` is a standard function which allocates a block of memory in the "heap" and returns a pointer to the new block.
- The prototype for `malloc()` and other heap functions are in **alloc.h**.
- The argument to `malloc()` is the integer size of the block in bytes.
- `malloc()` returns NULL if it cannot fulfill the request.
- The `malloc()` function reserves a block of memory of specified size and return a pointer of type void. This means that we can assign it to any type of pointer.
- It takes the following form;

```
ptr=(cast-type*)malloc (byte-size);
```

- ptr is a pointer of cast-type. The malloc returns a pointer (of cast type) to an area of memory with size byte-size.

• Example :

```
x=(int *)malloc(100 *sizeof (int));
```

- On successful execution of this statement, a memory space equivalent to "100 times of the size of an int" bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer x of type of int. Similarly, the statement ;

`cptr = (char *)malloc(10);`
allocates 10 bytes of space the pointer cptr of type char. This is illustrated as,

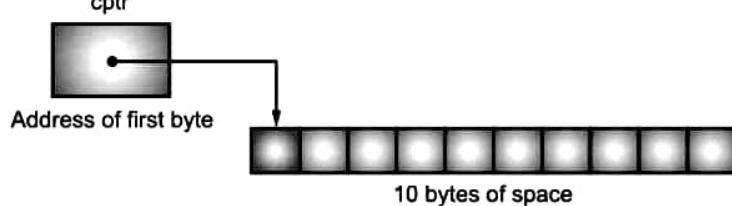


Fig. 1.5: Illustration of pointer

- Note that the storage space allocated dynamically has no name and therefore its contents can be accessed only through a pointer. We may also use **malloc** to allocate space for complex data types such as structures.

- **Example :**

```
st_ptr = (struct stud *)malloc(sizeof(struct stud));
```

where `st_var` is the pointer of type `struct stud`.

- Remember, the malloc allocates a block of contiguous bytes.

2. Allocating Multiple Blocks of Memory using `calloc()`

- `calloc()` is another memory allocation function that is normally used for requesting memory space at runtime for storing derived data types such as arrays and structures.
 - While `malloc()` allocates a single block of storage space, `calloc` allocates multiple blocks of storage; each of the same size, then sets all bytes to zero.
 - The general form of `calloc()` is,
`ptr = (cast_type *)calloc(n, elem-size);`
 - The above statement allocates contiguous space for `n` blocks. All bytes are initialized to zero and the pointer to the first byte of the allocated region is returned. If there is not enough space, a `NULL` pointer is returned.
 - The following piece of code allocates space for a structure variable,

```
 . . . . .
. . . . .
struct student
{
    char Name [25];
    float Age;
    int Roll_No;
};
```



```
typedef struct student STUD;
STUD *stud_ptr;
int SE_IT = 30;
stud_ptr = (STUD *) calloc(SE_IT, sizeof (STUD));
. . . . .
. . . . .
```

- STUD is of type struct student having three members Name, Age, and Roll_No. The calloc() allocates memory to hold data for 30 such records.
- We must be sure that the requested memory has been allocated successfully before using the stud_ptr. This may be done as follows;

```
if (stud_ptr==NULL)
{
    printf ("Available memory not sufficient");
    exit (1);
}
```

3. Releasing the allocated Space using Free()

- Compile-time storage of a variable is allocated and released by the system in accordance with its storage class.
- With the dynamic run-time allocation, it is our responsibility to release the space when it is not required. The release of storage space becomes important when the storage is limited.
- free() function is used to release (free) the block of heap memory allocated dynamically so that it can be used in future.
- The argument to free() is a pointer to a block of memory in the heap, a pointer which some time earlier was obtained via a call to malloc().

```
free(ptr);
```

- Here, ptr is a pointer to a memory block, which has already been created by malloc() or calloc(). Use of an invalid pointer in the call may create problems and cause system crash.
- We should remember two things here,
 1. A block of memory pointed by 'ptr' is released and not the memory allocated to 'ptr' i.e. we can use ptr variable in further program.
 2. To release an array of memory that was allocated by calloc we need only to release the pointer once. It is an error to attempt to release the pointer once. It is an error to attempt to release elements individually.

4. Altering the size of a Block realloc()

- It is likely that we discover later, the previously allocated memory is not sufficient and we need additional space for more elements. It is also possible that the memory allocated is much larger than necessary and we want to reduce it.

In both the cases, we can change the memory size already allocated with the help of the function **realloc()**. The process is called the reallocation of memory.



- For example, if the original allocation is done by the statement,

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

then reallocation of space may be done by the statement,

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

- This function allocates a new memory space of size newsize to the pointer variable *ptr* and returns the pointer to the first byte of the new memory block.
- The newsize may be larger or smaller than the size. Remember, the new memory block may or may not begin at the same place as the old one.
- In case, it is not able to find additional space in the same region, it will create the same in an entirely new region and move the contents of the old block into the new block. This function guarantees that the old data will remain intact.
- If the function is unsuccessful in locating additional space, it returns a NULL pointer and the original block is freed (lost).
- This implies that it is necessary to test the success of operation before proceeding further.
- Write a program to store a character string in a block of memory space created by malloc() and then modify the same to store a larger string.

Program 1.1:

[Oct. 15, April 16]

```
#include<stdio.h>
#include<string.h>
#include<alloc.h>
main( )
{
    char *buffer;
    /*Allocating memory*/
    if ((buffer=(char *) malloc (10))==NULL)
    {
        printf ("malloc failed .\n");
        exit (1);
    }
    printf ("Buffer of size 10 created \n");
    strcpy (buffer, "UNIVERSITY");
    printf ("\nBuffer contains %s \n", buffer);
    /*Reallocation*/
    if ((buffer=(char*) realloc (buffer, 14))==NULL)
    {
        printf ("Reallocation failed.\n");
        exit (1);
    }
}
```



```
    printf ("\nBuffer size is to be modified. \n");
    printf ("\nBuffer still contains %s \n, buffer);
    strcpy (buffer, "PUNEUNIVERSITY");
    printf ("\nBuffer now contains %s \n", buffer);
    /*Freeing memory */
    free (buffer);
}
```

Output:

```
Buffer of size 10 created
Buffer contains UNIVERSITY
Buffer size is to be modified
Buffer still contains UNIVERSITY
Buffer now contains PUNEUNIVERSITY
```

1.2 ALGORITHM-DEFINITION AND CHARACTERISTICS

- There are different ways to solve a given problem and these ways are represented by writing an algorithm.

1.2.1 Definition of Algorithm

[Oct. 17]

- 'Algorithm' is a finite sequence of instructions, which can be carried out to solve a particular problem in order to obtain the desired result.
- The sequence of instructions must posses the following characteristics:
 1. The instructions must be precise and unambiguous.
 2. Every instruction should be performed in finite time.
 3. The algorithm should ultimately terminate (the instructions should not repeat infinitely).
 4. The desired results (i.e. one or more outputs) should be obtained after the algorithm terminates.

1.2.2 Uses of Algorithm

1. It gives a language independent layout of the program.
2. Using the basic layout the program can be developed in any desired language.
3. Representation is in simple English language, so it is very easy to understand.
4. Facilitates easy coding.

1.2.3 Attributes of Algorithms

Algorithms consist of following attributes:

- **Correctness:**
 - Does the algorithm solve the problem it is designed for?
 - Does the algorithm solve the problem correctly?



- **Ease of understanding (clarity):**
 - How easy it is to understand or alter the algorithm?
 - Important for program maintenance.
- **Elegance:**
 - How clever or sophisticated is the algorithm?
 - Sometimes elegance and ease of understanding work at cross-purposes.
- **Efficiency:**
 - How much time and / or space does the algorithm required when executed?
 - Perhaps the most important desirable attribute.

1.2.4 Characteristics of Algorithm

[Oct. 17]

1. A well-defined set of steps to provide a solution of a specific problem.
2. An algorithm should have zero or more input.
3. An algorithm should exhibit at least one output.
4. An algorithm should be finite.
5. Each instruction in an algorithm should be defined clearly.
6. Each instruction used in an algorithm should be basic and easy to perform.

1.2.5 Advantages of Algorithms

1. It gives language independent layout of the program.
2. Using the basic layout the program can be developed in any desired language.
3. Representation is in simple English language so it is very easy to understand.
4. Facilitates easy coding.

1.2.6 Examples of Algorithms

Ex. 1: Write an algorithm to find the volume and total surface area of a cylinder.

Volume = $227 * r * r * h$ Surface Area = $447 * r * (r + h)$

Sol.: Algorithm to find volume.

- Step 1: Start
- Step 2: Read r
- Step 3: Read h
- Step 4: Calculate s = r * r
- Step 5: Calculate x = 227 * s
- Step 6: Calculate a = x * h
- Step 7: Print a
- Step 8: Stop

**Algorithm to find surface area.**

Step 1: Start
Step 2: Read r
Step 3: Read h
Step 4: Calculate $t = r + h$
Step 5: Calculate $s = r * t$
Step 6: Calculate $A = 447 * s$
Step 7: Print A
Step 8: Stop.

Ex. 2: Write an algorithm to input values of x and y to calculate x^y .**Sol.:**

Step 1: Start
Step 2: Read x
Step 3: Read y
Step 4: Initialise $t = x$
Step 5: Repeat steps 6 and 7 until $y = 1$
Step 6: Calculate $y = y - 1$
Step 7: print t
Step 8: Stop.

Ex. 3: Write an algorithm to check if the given number is a Palindrome. A palindrome is a number that reads same forward and backwards. e.g. 12321, 5445.**Sol.:**

Step 1: Start
Step 2: Read no
Step 3: Let temp = no and mirror = 0
Step 4: Repeat steps 4 and 5 until temp is not equal to 0
Step 5: Calculate mirror = temp mod 10 + mirror * 10
Step 6: Calculate temp = temp div 10
Step 7: If no = mirror then print "number is palindrome"
Step 8: If no is not = mirror then print "number is not palindrome"
Step 9: Stop.

1.3 ALGORITHM ANALYSIS

[April 15, 16, Oct 17]

- As there are different ways to solve a problem and some of the solutions may be more efficient than the other, there is need to compare these ways or algorithms to select the best one. This process is called as algorithm analysis.
- There are many criteria upon which an algorithm can be evaluated such as:
 - Does it produce correct output?
 - Does it work as per specifications?



3. Does it contain proper documentation?
 4. Does the algorithm modular so that it can be converted to different subfunctions?
 5. Is the code readable?
- Performance of an algorithm is measured in terms of:
 1. **Space complexity:** The amount of memory required to perform the task.
 2. **Time complexity:** Total time taken by an algorithm to perform a task.
 - Consequently, analysis of algorithms focuses on computation of space and time complexity.
 - Space can be defined in terms of space required to store the instructions and data whereas the time is the computer time an algorithm might require for its execution, which depends on the size of the algorithm and input.
 - Operators which are executed oftenly (within loop) are called active operations.
 - Assignment operations and arithmetic operations are called book-keeping operations and not executed oftenly.
 - For simplicity declaration statement are ignored.
 - Execution time is proportional to the frequency count of an active operation.
 - If there are more than one active operation, the sum of frequency counts result in polynomial. The largest degree variable gives time complexity.

Example: Consider following codes,

1.

```
for(i=1; i<=n; i++)
    a=a - i;
a = a - i executes 'n' times which is a frequency count.
```
2.

```
for (i=1; i<=n; i++)
    for(j=1; j<=m; j++)
        a=a + b;
a = a + b executes 'n * m' times which is a frequency count.
```
3.

```
for(i=1; k <=n; i++)
    a=a + b;
    for(j=1; j<=m; j++)
        for(k=1; k<m; k++)
            a=a*b;
'a = a+b' executes 'n' times and 'a=a*b' executes m*m i.e. m2 times. Therefore, total frequency count is m2+n. Thus, complexity is order to m2.
```

1.3.1 Space Complexity

- Space complexity specifies the amount of computer memory required during the program execution, as a function of the input size.



- The memory requirement is summation of the program space, data space and stack space.
 - (a) **Program space** : This is memory occupied by program itself.
 - (b) **Data space** : This is memory occupied by data members such as constants and variables.
 - (c) **Stack space** : This is stack memory needed to save functions runtime environment while another function is called. This cannot be accurately estimated, since it depends on the runtime call stack, which can depend on the programs' data set. This memory space is crucially important for recursive functions.
- Space complexity measurement, that is, measurement of space requirement of an algorithm can be done at two different times :
 1. Compile time space complexity and
 2. Runtime space complexity

1. Compile time space complexity :

- Compile time space complexity is defined as the storage requirement of a program at compile time.
- This storage requirement can be computed during compile time. The storage needed by the program at compile time can be determined by summing up the storage size of each variable using declaration statements.
- For example, space complexity of a recursive function of calculating factorial of number 'n' depends upon the number n itself.

2. Runtime space complexity :

- If program is recursive or uses dynamic variables or dynamic datastructure then there is a need to determine space complexity at runtime.
- Generally, this dynamic storage size is dependent on some parameters used in a program. It is difficult to estimate memory requirement accurately, as it is also determined by efficiency of compiler.

1.3.2 Time Complexity

[Oct. 16]

- Time Complexity is the time taken by program for execution. What we do is to count the number of algorithm steps.
- We can determine the number of steps needed by a program to solve a particular problem instance in one of the two ways :
 1. Introduce a new variable, count, into the program. This is a global variable with initial value 0. Statements to increment count are introduced in the program. This is done so that each time the statement in original program is executed, count is incremented by step count of that statement.

We measure the run time of an algorithm by counting the number of steps.



2. Compute number of times each statement will be executed manually. Number of times the statement is executed is its **frequency count**. Sum-up of frequency counts of all statements. This sum is the number of steps needed to solve given problem
- There are different types of time complexities which can be analyzed for an algorithm :
 1. **Best case time complexity** : It is a measure of the minimum time that the algorithm will require for an input of size 'n'. The running time of many algorithms varies not only for the inputs of different sizes but also for the different inputs of same size. For example, in the running time of some sorting algorithms, the sorting will depend on the ordering of the input data. Therefore, if an input data of 'n' items is presented in sorted order, the operations performed by the algorithm will take the least time.
 2. **Worst case time complexity** : It is a measure of the maximum time that the algorithm will require for an input of size 'n'. Therefore, if various algorithms for sorting are taken into account and say 'n' input data items are supplied in reverse order for any sorting algorithm, then the algorithm will require n^2 operations to perform the sort which will correspond to the worst case time complexity of the algorithm.
 3. **Average case time complexity** : The time that an algorithm will require to execute a typical input data of size 'n' is known as average case time complexity. We can say that the value that is obtained by averaging the running time of an algorithm for all possible inputs of size 'n' can determine average case time complexity. The computation of exact time taken by the algorithm for its execution is very difficult. Thus, the work done by an algorithm for the execution of the input of size 'n' defines the time analysis as function $f(n)$ of the input data items.

1.3.3 Asymptotic Notation

[Oct. 16, 17]

- An a priori analysis of computing time ignores all the factors which are machine or language dependent and concentrates on determining order of magnitude of algorithm.
- We need to formalize the ideas that an algorithms has running time or storage requirements that are "never more than," "always greater than," or "exactly" some amount.
- We use $T(n)$ for the running time or time complexity and $S(n)$ for the storage requirements or space complexity of an algorithm on an instance of size n .
- The phrase "**never more than**" **expresses an upper bound on $T(n)$ or $S(n)$** . There are several ways you could choose to express this on $T(n)$.
 1. The algorithm never takes more than (some function on n of) operations.
 2. The algorithm has running time that is always less than some function of n .
 3. The algorithm's running time is in the order of some function of n .
 4. The algorithm's time complexity is big-O of some function of n .
- You can make similar statements about space complexity, $s(n)$.
- The phrase "**always less than**" **expresses a lower bound on $T(n)$ or $S(n)$** .



- There are several ways you could choose to express this on $T(n)$:
 1. The algorithm always takes at least (some function of n) operations.
 2. The algorithm has running time that is always greater than some function of n .
 3. The algorithm's time complexity is big-Omega (Ω) of some function of n .
- You can make similar statements about space complexity.
- Let's give the function a name, call it $f(n)$.
- We are familiar with the common functions that will be useful complexity functions.
 - o The constant function $f(n) = c$;
 - o The logarithmic function $f(n) = \lg n$;
 - o The linear function $f(n) = n$;
 - o The linear-log function $f(n) = n \lg n$;
 - o The quadratic function $f(n) = n^2$;
 - o The cubic function $f(n) = n^3$;
 - o The general power function $f(n) = n^k$;
 - o The exponential function $f(n) = 2^n$;
 - o The factorial function $f(n) = n!$;

Omega Notation

$f(n) = \Omega(g(n))$ iff there exist positive constants C and n_0 such that for all $n > n_0$,

$$|f(n)| \geq C |g(n)|$$

(lower bound)

- In some cases the time for an algorithm, $f(n)$ will be such that $f(n) = \Omega(g(n))$ and $f(n) = O(g(n))$ for such circumstances we use the following notation i.e. Θ notation.
- Ω -notation is used to denote lower bounds. We write,

$$T(n) = \Omega(f(n))$$

to denote that $T(n)$ always takes at least roughly $f(n)$ operations. This is stated in English as,

" $T(n)$ is Ω of $f(n)$ "

We need to make this informal notion more precise.

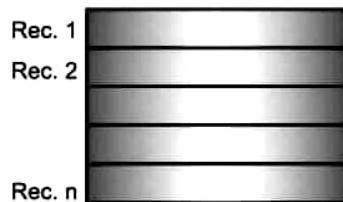
- **Definition 1 :** Let $T(n)$ be the time complexity of an algorithm and let $f(n)$ be a proper complexity functions. We write,

$$T(n) = \Omega(f(n))$$

if and only if there exists a natural number $N > 0$ and a constant $d > 0$ such that

$$d f(n) \leq T(n) \quad \Omega(n) > N$$

- This can be interpreted as saying,
"T grows faster than a constant time f for all sufficiently large n", or
"The graph of T lies above the graph of a constant time f for all sufficiently large n."
- Consider an algorithm of linear search.

**Fig. 1.6: File of Record 'n'**

- In this example, to search a particular record in a file of 'n' records, there are 4 possibilities :
 1. The record is present at 1st position (Best case).
 2. The record is present at last position (Worst case).
 3. The record is somewhere in between 1st and last record (Average case).
 4. Record is absent (Worst case).
- This algorithm will take maximum of 'n' number of comparisons to search the record, if the record is at nth place or if the record is absent. So the maximum time taken by the algo is say 'n comparisons'. Time complexity can be said as O(n).
- Now consider a case if the record is present at the 1st location (Best case). Here minimum time taken by the algorithm is time for one comparison.
- Therefore, the lower bound of an algorithm can be stated as $\Omega(1)$ which is constant.

Θ (Theta) Notation

- $f(n) = \Theta(g(n))$ if there exists positive constants C_1, C_2 and no such that $\Theta(g(n)) \leq f(n) \leq \Theta(g(n))$.
- if $f(n) = \Theta(g(n))$ then $g(n)$ is both upper and lower bound on $f(n)$. This means that the worst and the best cases require the same amount of time to within a constant factor.
- Θ -notation is used to denote both upper and lower bounds. We write,

$$T(n) = \Theta(f(n))$$

to denote that $T(n)$ always takes roughly $f(n)$ operations. This is stated in English as

" $T(n)$ is Θ of $f(n)$ "

We need to make this informal notion more precise.

- **Definition 1 :** Let $T(n)$ be the time complexity of an algorithm and let $f(n)$ be a proper complexity functions. We write,

$$T(n) = \Theta(f(n))$$

if and only if there exists a natural number $N > 0$ and constants $c, d > 0$ such that

$$d f(n) < T(n) < c f(n) \quad n > N$$

- This can be interpreted as saying
 "T grows at the same rate as some constant time f for all sufficiently large n", or
 "The graph of T lies between the graphs of df and cf for all sufficiently large n."



- Consider the algorithm of finding a maximum number from a list of 'n' unsorted numbers. In this algorithm if the maximum number is present at 1st location or at last location, our algorithm is going to perform 'n' comparisons always. So each time our algorithm takes same maximum time and same minimum time i.e. 'n' comparisons. This means that lower bound and upper bound of time complexity is same, so this can be represented by asymptotic notation Θ . So this algo takes.

5
9
7
11
2
1

Fig. 1.7: List

- (1) $O(n)$
 (2) $\Omega(n)$

and hence (3) $\Theta(n)$

```
max (A, n, j)
{ i = 1;
  max = A(i);
  for i = 2 to n do
    begin
      if A(i) > max then
        max = A(i);
        j = i;
    end if
}
```

- Algorithm above has computing time both $O(n)$ and $\Omega(n)$, since the for loop always make $n = 1$ iterations. Thus, we say that its time is $\Theta(n)$.

Big – O Notation

- We can express the time complexity of a program in terms of the order of magnitude of frequency count using the Big O notation. If $f(n)$ represent the computing time of some algorithm and $g(n)$ represent a known standard function like n , n^2 , $n \log n$ etc. Then to write $f(n)$ in $O(g(n))$ means that $f(n)$ of n is equal to biggest order of function $g(n)$.
- Big 'O' notation helps to determine the time as well as space complexity of the algorithm.**
- Using Big 'O' notation, the time taken by the algorithm and the space required to run the algorithm can be ascertained. This information is useful to set algorithms and to develop and design efficient algorithms in terms of time and space complexity.
- $f(n) = O(g(n))$ iff there exist two + ve constants c and n_0 such that $|f(n)| \leq |g(n)|$ for all values $n \geq n_0$.
- When we say that an algorithm has computing time $O(g(n))$, we mean that if the algorithm is run on same computer on the same type of data, but increasing value of n , the resulting time will always be less than some constant time $|g(n)|$.
- If an algorithm has k -statements whose orders of magnitude are,
 $C_1 n^{m_1} + C_2 n^{m_2} + \dots + C_k n^{m_k}$, order of algorithm is $O(n^m)$.
 where $m = \max(m_i), 1 \leq i \leq k$.



- O-notation is used to denote upper bounds. We write,

$$T(n) = O(f(n))$$

to denote that $T(n)$ never takes more than roughly $f(n)$ operations. This is stated in English as :

" $T(n)$ is in the order of $f(n)$ ", or " $T(n)$ is big-O of $f(n)$ "

We need to make this informal notion more precise.

- **Definition 1 :** Let $T(n)$ be the time complexity of an algorithm and let $f(n)$ be a proper complexity functions. We write

$$T(n) = O(f(n))$$

if and only if there exists a natural number $N > 0$ and a constant $c > 0$ such that

$$T(n) \leq c f(n) \quad \forall n > N$$

This can be interpreted as saying

" T grows slower than a constant time f for all sufficiently large n ", or

"The graph of T lies below the graph of a constant time f for all sufficiently large n ."

- **Most common computing time of algorithm :** If complexity of algorithms is

$O(1)$ = Computing time is constant

$O(n)$ = Linear

$O(n^2)$ = Quadratic

$O(n^3)$ = Cubic

$O(2^n)$ = Exponential

$O(\log n)$ = Logarithmic

- Algorithm with exponential running time is not suitable for practical use. $O(\log n)$ is better than $O(n)$. For large n , $O(n \log n)$ is better than $O(n^2)$ but not as good as $O(n)$.
- These can be summarized in following table.

n	$\log n$	$n \log n$	n^2	n^3	2^n
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	4294967296
64	6	384	4096	262144	Too large



1.4 INTRODUCTION TO DATA STRUCTURE

[April 16]

- Data structures is the branch of computer science that give knowledge of how the data should be organized, how the flow of data should be controlled and how data structure should be designed and implemented to reduce the complexity and increase the efficiency of the algorithm.
- The theory of data structures helps us to understand and use the concept of abstraction, analysing problems step by step and to develop algorithms to solve real world problems.
- It enables us to design and implement various data structures for example stacks, queue, linked list, tree, graphs etc.
- Effective use of principles of data structures increase efficiency of algorithms to solve problems like searching, sorting, populating and handling voluminous data.

1.4.1 Need of a Data Structure

- A data structure helps us to understand the relationship of one data element with the other and organize it in a logical or mathematical manner.
- Data structures facilitates us in various ways. They are useful in three categories :
 1. Real world data storage.
 2. Programmer's tools.
 3. Modeling.
- By real world data, we mean data that describes physical entities external to computer. Some examples are a personnel record that describes an actual human being, an inventory record.
- Suppose a programmer wanted to write a card file program. He might need to answer various types of questions like how to store the data in computer's memory? Would written method works for a hundred file cards? A thousand? A million? Would a method permit quick insertion of new cards and deletion of old ones? Would it allow for fast searching for a specified card? Would a program sort cards? The answers to all those questions we could get from the concepts of data structures.
- Some data storage structures are not meant to be accessed by the user, but by the program itself. A programmer uses such structures as tools to facilitate some other operation. Stacks, queues and priority queues are often used in this way.
- The third category of data structures and algorithms is real world modeling. A queue, for example, can model customers waiting in a line at bus stop, whereas a priority queue can model messages waiting to be transmitted over a local area network.

1.4.2 What is Data Structure?

- Basically, a **data structure is an aggregation of atomic and composite data types into a set with defined relationships**. In this definition structure means a set of rules that holds the data together.



- In other words, if we take combination of data types and fit them into a structure such that we can define its relating rules. We can have data structures that consist of other data structures.
- Data structure contains the data object along with the set of operations which will be performed on them. It also contains the information about the manner in which these data objects are related.
- Data object refers to the set of elements which may be finite or may not be finite. For example, set of real numbers which is infinite, whereas set of numeric digits is finite.

Data Object

- A data object represents a container for data values, a place where data values may be stored and later retrieved.
- A data object is characterized by a set of attributes, one of the most important of which is its data type.
- The attributes determine the number and type of values that the data object may contain and also determine the logical organization of those values.
- The data object "alphabets" can be defined as $D = \{A, B, \dots, Z, a, b, \dots, z\}$. Data object "integer" $D = \{-\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$.
- Data object is runtime instance of data structure.
- Data object is runtime grouping of one or more data pieces.
- Some of the data objects that exist during program execution are programmer defined such as variables, constants, arrays, files etc.
- The programmer explicitly creates and manipulating them through declarations and statements in the program.
- System defined data objects are created during program execution without explicit specification by the programmer.

1.5 TYPES OF DATA STRUCTURE

[April 17]

1.5.1 Types of Data Structures

- We defined a data structure as a way of organizing data that specifies :
 - (i) A set of data element i.e. a data object and,
 - (ii) A set of operations which are applied to this data object.
- These two sets form a mathematical construct that may be implemented using a particular programming language. The data structure is independent of their implementation.
- In any programming language, data structure is often called as a data type for defining variables of that types in order to create many instances. The names of these variables are then used in a program. The operations on these variables are performed accordingly.



- The various types of data structures are :
 1. Primitive and Non-primitive data structures.
 2. Linear and Non-linear data structures.
 3. Static and Dynamic data structures.
 4. Persistent and Ephemeral data structures.
 5. Sequential and Direct access data structures.

1. Primitive and Non-Primitive Data Structures :

- **Primitive data structure** defines a set of primitive elements which do not involve any other elements as its subparts, for example, data structure defined for integers and characters. These are generally primary or built-in data types in programming languages.
- **Non-primitive data structures** are those which define a set of derived elements such as array. Array data structure defined in C consists of a set of similar type of elements. While structure is another example of non-primitive data type which consists of a set of elements which may be of different data types.
- Hence, int, char, float and double are primitive data types of language C and array, structure and user defined data types are non-primitive data structures.

2. Linear and Non-linear Data Structures :

- A data structure is said to be **linear** if its elements form a sequence or a linear list.
- In a linear data structure, every data element has unique successor and unique predecessor.
- There are two basic ways of representing linear structures in memory.
- One way is to have the relationship between the elements by means of pointers (links), called as linked lists. And the other way is using sequential organization i.e. arrays.
- **Non-linear data structures** are used to represent the data containing hierarchical or network relationship between the elements.
- Trees and graphs are examples of non-linear data structure. In non-linear data structures, every data element may have more than one predecessor as well as successor. Elements do not form any particular linear sequence.
- The tree drawn in Fig. 1.8 is non-linear data structure.

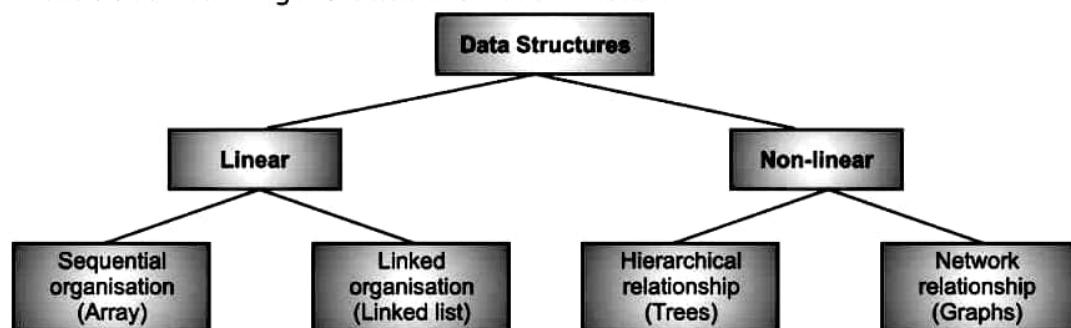


Fig. 1.8: Linear and Non-Linear Data Structure

**3. Static and Dynamic Data Structures :**

- A data structure is referred as **static data structure** if it is created before program execution begins (i.e. during compilation time).
- The variables of static data structure have user specified names.
- Array is static data structure.
- We have another class of variables called dynamic variables, which are created and destroyed dynamically during the execution of a program. These dynamic variables are not referenced by a user defined names always. These are accessed indirectly using their addresses through pointers.
- A data structure which is created at run time, is called as **dynamic data structure**.
- Linked lists is a dynamic data structure when realized using dynamic memory management and pointers.
- Non-linear data structures are generally implemented in a same way as linked list. Hence trees and graphs can be implemented as dynamic data structures whereas arrays is a static data structure.

4. Persistent and Ephemeral Data Structures :

- A data structure that supports operations on the most recent version as well as previous version is termed as **persistent data structure**.
- A persistent data structure is **partially persistent** if any version can be accessed but only the most recent one can be updated; and it is **fully persistent** if any version can both be accessed and updated.
- **An ephemeral data structure** is the one which supports operations only on the most recent version.
- The distinction between ephemeral and persistent data structure is essentially the distinction between functional (also called as effect free) and conventional imperative (also called as effect full) programming paradigms. The functional data structures are persistent and imperative data structures are ephemeral.
- Data structures in **conventional imperative** languages are **ephemeral**.
 - Insertion into a linked list mutates the list.
 - The old version is lost.
 - Pushing onto a stack modifies the stack pointer and writes on the underlying memory. Popping modifies the stack pointer.
- Data structures in **functional languages** are **persistent**.
 - Inserting an element into a list yields a new list. The old versions is still available.
 - Stacks can be implemented so that pushing yields a new stack, leaving the old stack still available.



5. Sequential Access and Direct Access Data Structures :

- This classification is with respect to the access operations associated with data structure.
 - (i) **Sequential** : Sequential access means to access n^{th} element we must access the preceding $(n - 1)$ data elements. Linked list is a sequential access data structure.
 - (ii) **Direct access** : Direct access means any element can be accessed without accessing its predecessor or successor; we can directly access any n^{th} element. Array is an example of direct access data structure.

1.5.2 Implementation of Data Structure

- A data structure is a aggregation of atomic and composite data types into a set with relationship among them defined.
- A data structure DS is a triplet, that is $DS = (D, F, A)$ where D is a set of data object, F is a set of functions and A is a set of rules to implement the functions.
- Let us consider an example of integer data type (int) in C :
 $D = (0, \pm 1, \pm 2, \pm 3, \dots)$
 $F = (+, -, *, /, \%)$
 $A = \text{a set of binary arithmetics rules to perform addition, subtraction, division, multiplication and module operations.}$
- Set of axioms, A define semantics of operations on DS for F.
- An implementation of a data structure DS is a mapping from DS to a set of other data structures E. This mapping specifies how every data object of DS is to be represented by objects of E.
- It require that every function of D must be written using the functions of the implementing data structures E. Thus, we may say that integers are represented by bit strings, boolean is represented by zero and one, an array is represented by a set of sequential locations in memory.
- Implementation of data structure can be viewed in terms of two phases :
 1. Specification and
 2. Implementation
- Such a division of tasks is useful as it helps to control the complexity of the entire process.
- **Phase 1 : Specification** : At the first stage, a data structure should be designed so that we know what it does, not necessarily how it will do it.
- **Phase 2 : Implementation** : At this stage we define all function with respect to description how to manipulate a data structure. This can be done with algorithms so that details of the operation can be understood easily and reader can implement them easily and effectively with the help of any programming language. Either of the one design tools that is, algorithm or flow chart can be used at this phase.



1.5.3 Data Structure Operations

- A Data Structure consist of following operations :
 1. **Inserting** : Adding a new data in the data structure is referred as insertion.
 2. **Deleting** : Removing a data from the data structure is referred as deletion.
 3. **Sorting** : Arranging the data in some logical order (ascending or descending, numerically or alphabetically).
 4. **Searching** : Finding the location of data within the data structure which satisfy the searching condition.
 5. **Traversing** : Accessing each data exactly once in the data structure so that each data item is traversed or visited.
 6. **Merging** : Combining the data of two different sorted files into a single sorted file.

1.6 ABSTRACT DATA TYPES (ADT)

[Oct. 16]

- When an application requires a special kind of data, which is not available as built-in data type, then it is the programmer's burden to implement his own kind of data.
- Here, the programmer has to give more efforts regarding how to store value for that data, what are the operations that meaningfully manipulate variables of that kind of data, amount of memory requirement for storing variables.
- The programmer has to decide this entire thing and accordingly implement it. Programmers own data type is termed as abstract data type.
- The above task can be performed using the dynamic memory management functions in C.
- The functions of C which support memory allocation are **malloc()**, **calloc()**, **alloc()**. Function to release node is **free()**.
- To verify memory allocation process the address returned by memory allocation function is compared with value **NULL**. Non-null address returned indicates that the process is successful.
- A useful tool for specifying the logical properties of a data type is the abstract data type or ADT.
- Fundamentally, a data type is collection of values and a set of operations on those values. That collection and those operations form a mathematical construct that may be implemented using a particular hardware or software data structure.
- **The term 'abstract data type' refers to basic mathematical concept that defines data type.**
- ADT is defined as a mathematical model of the data objects that make up a data type as well as functions that operate on these objects.
- ADT is the specification of logical and mathematical properties of a data type or structure.



- For example : int data type in 'C' Programming provides an implementation of mathematical concept of an integer number.
- The int data type in 'C' can be considered as an implementation of ADT, integer-ADT. Integer-ADT defines the set of numbers given by the union of the set $(-1, -2, -3, \dots, \infty)$ and set of whole numbers $(0, 1, 2, \dots, \infty)$. Integer-ADT also specifies the operations that can be performed on integer number, for example, addition, subtraction, multiplication etc.

Atomic Type

- Generally, a data structure is represented by a memory block which has two parts :
 - Data storage, and
 - Address storage.
- An atomic type data is a data structure that contains only the data items and not the pointers.
- Thus, for a list of data items, several atomic type nodes may exist each with a single data item corresponding to one of the legal data types.
- The list is maintained using a list node which contains pointers to these atomic nodes and a type indicator indicating the type of atomic node to which it points. Whenever a test node is inserted in the list, its address is stored in the next free element of the list of pointers.
- In Fig. 1.9, a list of atomic nodes is maintained using list of nodes. In each node type represents the type of data stored in the atomic node to which the list node points. 1 stands for integer type, 2 for real number and 3 for character type or any different assumption can be made at implementation level to indicate different data types.

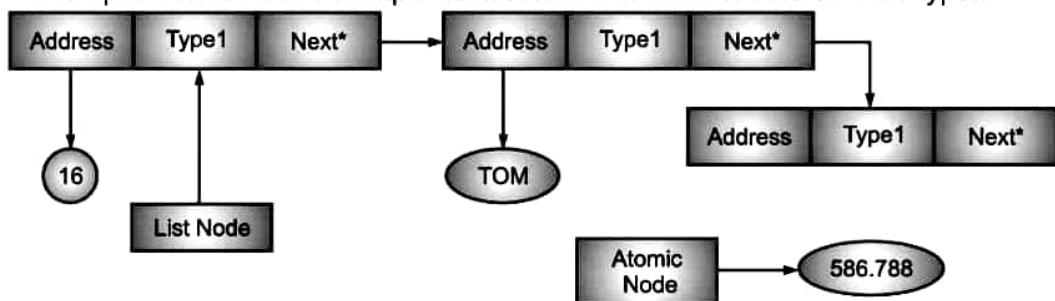


Fig. 1.9: Atomic type

- In computer science, an abstract data type (ADT) is a mathematical model for a certain class of data structures that have similar behavior; or for certain data types of one or more programming languages that have similar semantics.
- The triplet $DS = (D, F, A)$ is referred to as abstract data type. It is abstract because the axioms in the triplet do not give any idea about the representation.



- An abstract data type is defined indirectly, only by the operations that may be performed on it and by mathematical constraints on the effects (and possibly cost) of those operations.
- For example, an abstract stack could be defined by three operations: push, that inserts some data item onto the structure, pop, that extracts an item from it (with the constraint that each pop always returns the most recently pushed item that has not been popped yet), and peek, that allows data on top of the structure to be examined without removal. When analyzing the efficiency of algorithms that use stacks, one may also specify that all operations take the same time no matter how many items have been pushed into the stack, and that the stack uses a constant amount of storage for each element.
- Abstract data types are purely theoretical entities, used to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type of programming languages.
- However, an ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language.
- The ADT is a useful tool for specifying the logical properties of a data type without giving any details about the implementation of operations on that data type.

1.7 INTRODUCTION TO ARRAY AND STRUCTURE

[April 16, Oct. 16]

- Arrays are data types. The simplest form of array is a one dimensional array that may be defined abstractly as a finite, ordered set of homogeneous elements.
- By 'finite' we mean that there is a specific number of elements in the array. This number may be large or small, but it must exist.
- By 'ordered' we mean that the elements of the array are arranged so that there is a zeroth, first, second, third and so on.
- By 'homogeneous' we mean that all the elements in the array must be of the same data type.
- For example, an array may contain all integers or characters but may not contain both.
- **An array is a group of related data items that share a common name.**
- For example, salary[10]. Here, salary is the name of array. 10 is called index number or subscript in bracket. salary[10] represents the salary of the 10th employee.
- The set of complete values is referred to as an array. The individual values are called elements. Arrays can be of any data type.

1.7.1 What is an Array ?

- Arrays are the most common and easy to use data structures.
- An array is most often defined as : **a consecutive set of memory locations.**
- Each memory location has associated index and value pair.



- For each index which is defined, there is a value associated with that index. In mathematical terms it is called as a correspondence or a mapping.
- However, as computer professional we want to provide a more functional definition by giving the operations which are permitted on this data structure.
- With arrays, two operations are associated retrieve and store values.

1.7.2 Definition

- An array is a finite ordered collection of homogeneous data elements which provides random access to the elements.
- Example, In C, `int A[20];`

This declaration will create an array A, which have a capacity to hold 20 integers, where 20 is dimension or size of an array A.

`char B[30];`

This declaration will create an array B, which can hold 30 characters.

1.7.3 Terminology

1. **Size of Array :** Number of elements in an array is called as the size of array. Size of array is also called as length or dimension. Dimension once defined can not be modified after compilation. Hence, arrays are called as static data structures.
2. **Base :** Base of an array is the address of memory location where the first element of the array is located. Base of array is defined when the program is executed. Value of base varies at every run of the program. It is not defined by the programmer.
3. **Type :** Data type represents the kind of information i.e. type of data to be stored is an array.
4. **Index :** An element of an array is referenced by a subscript like `A[i]` and also as `Ai`. This subscript is known as index.
5. **Range of index :** Range of index in 'C' is 0 to $n-1$ for dimension ' n '.

1.7.4 Declaration of Array

- Arrays must be declared before they are used.

Syntax :

`type variable-name[size];`

- Here, type specifies the type of element that will be contained in the array such as int, float or char and size specifies the maximum number of elements that can be stored inside the array.
- For example, `float height[50];` declares height to be an array contain a maximum of 50 real numbers.
- For example, `int group[10];` declares group to be an array contain a maximum of 10 integer numbers.
- For example, `char name[10];` declares name as a character array that can hold a maximum of 10 characters.



- Suppose we store "WELL DONE" string into name as,
`char name[10] = "WELL DONE";`

Then each character of the string in an array is stored in memory as follows:

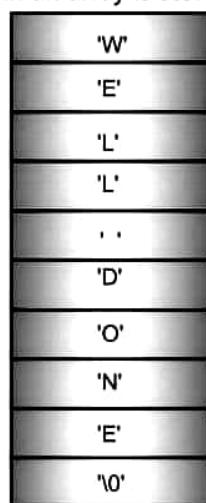


Fig. 1.10: Memory Representation of Character Array

- When the compiler sees a character string, it terminates it with an additional null character. Thus, element[9] holds the null character '\0' at the end. So when declaring character array, we must always allow an extra element space for the null terminator.
There is a difference between a character array and a string. Every string terminates with '\0'.

For e.g. `char str[] = {'H', 'E', 'L', 'L', 'O'}` is not a string as it is not terminated with '\0'. Size of 'str' array is 5.

Now following declaration indicates string,

`char str1 = {'H', 'E', 'L', 'L', 'O', '\0'}`

is a string. The above declaration can be written as,

`char str1[] = "HELLO";`

Size of 'str1' array is 6, one space for '\0' character.

1.7.5 Initialization of Arrays

- Syntax** of initialization of arrays are given below :

`type array-name [size]={list of values};`

Values in the list are separated by commas.

- For example, `int number[3] = {0,0,0};`
The size can be omitted.
- For example, `int number[] = {1,1,1,1};`
Character array can be initialized in the similar manner.
- For example, `char name[] = {'d', 'c', 'm'}; //not a string`



1.7.6 Array as an ADT

- An abstract data type includes:
 1. Declaration of data
 2. A set of rules that put data together.
 3. A set of operations performed on the data.
- Data in an array is a collection of fixed no. of components (or elements) of same type and a set of pairs - index and value.
- Rule for an array is the one-to-one correspondence between element and index.
- The primitive operations defined on an array are create, store and retrieve.
 1. **Create:** Creates an empty array or new array. This function requires type of data and size of an array.
 2. **Retrieve:** Return the value stored at a given index from an array.
 3. **Store:** It is used to enter new index value pairs or data in an array.
- Implementation of above operations are hidden (abstract) from user.
- Thus array is an ADT.

1.7.7 Differentiate between Array and Structure

[Oct. 16]

Array	Structure
1. An array is a collection of related data element of same type.	1. Structure can have elements of different types.
2. An array is a derived data type.	2. A structure is a programmer defined data type.
3. An array behaves like a built-in data types. All we have to do is to declare an array variable and use it.	3. But in case of structure, first we have to design and declare a data structure before the variable of that type are declared and used.
4. Array allocates static memory and uses index or subscript for accessing elements of the array.	4. For structures we can allocate dynamic memory and uses (.) dot operator for accessing the member of a structure.
5. Array name works as a pointer to the first element of it.	5. But it is not in case of structure.
6. The elements of the array are contiguous in memory.	6. The elements of a structure may not be contiguous.

1.8 TYPES OF ARRAY AND REPRESENTATION OF ARRAY

[April 15, 16]

- Array consist of following types :
 1. One-dimensional array,
 2. Two-dimensional array, and
 3. Multi-dimensional array.



1.8.1 One-Dimensional Array

- A one-dimensional array is used when it is necessary to keep a large number of items in memory and reference all the items in a uniform manner.
- A list of items having one variable name, one subscript is known as an One-dimensional array.
- For example, to represent a set of five numbers 35, 40, 20, 57, 19 by an array variable number, declare the variable number as follows : int number[5]; and computer reserves five storage locations as shown below :

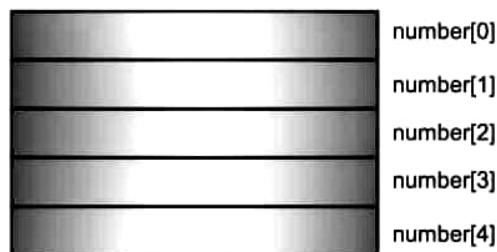


Fig. 1.11: Storage Location

- The smallest index is called its **lower bound** and in C is always 0 and the highest index is called its **upper bound**.
- If lower is the lower bound of an array and upper the upper bound, the number of elements in the array, called its range, is given by upper-lower + 1.
- For example, in the array, a, declared previously, the lower bound is 0, the upper bound is 99 and the range is 100.
- The values to the array elements can be assigned as follows :

```
number [0]=35  
number [1]=40  
number [2]=20  
number [3]=57  
number [4]=19
```

OR

- It can be done as follows:

```
int number[5] = {35, 40, 20, 57, 19};
```

Thus, array number stores the values as shown below :

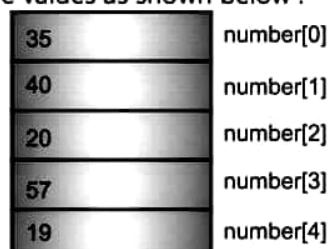


Fig. 1.12: Value Store in Memory

where b is an base address of 'number' array i.e. address of first element.



- Using the base address of an array, it is possible to calculate the address of an element using following formula:

Address of i^{th} element = Base address + Offset of the i^{th} element
 $\text{offset of } i^{\text{th}} \text{ element} = (i - LB) * (\text{Size of each element})$ where, LB is lower bound of an array.

For example, suppose 'C' we declare an array as,

```
int a[5];
```

Let base address is 100. Lower bound of every array in C is 0. Suppose size of integer is 2 bytes.

\therefore Address of 3rd element = $100 + (3 - 0) * 2 = 106$

- In the C language the declaration of arrays is as int a [100]; It specifies an array of 100 integers.
- The two basic operations that access an array are **extraction** and **storing**.
- The extraction operation is a function that accepts an array, a and an index, i and returns an element of the array. In C, this operation is denoted by expression a[i].
- The storing operation accepts an array, a, an index, i, and an element, x. In C this operation is denoted by the assignment statement a[i] = x.
- Before, a value has been assigned to an element of the array, its value is undefined (garbage value) and a reference to it in an expression is illegal.

1.8.2 Two-Dimensional Array

- An array having two dimensions (sizes) is known as a Two-dimensional array.

- Syntax :**

- type array-name [row-size][column-size];
- Here, type specifies the type of element that will be contained in the array such as int, float or char and row-size specifies the number of rows and column-size specifies the number of column, array-name is the name of table or, matrix.
- For example, int v[4][3];
- Two-dimensional array are stored in memory as shown below : (b is base address) Row major representation.

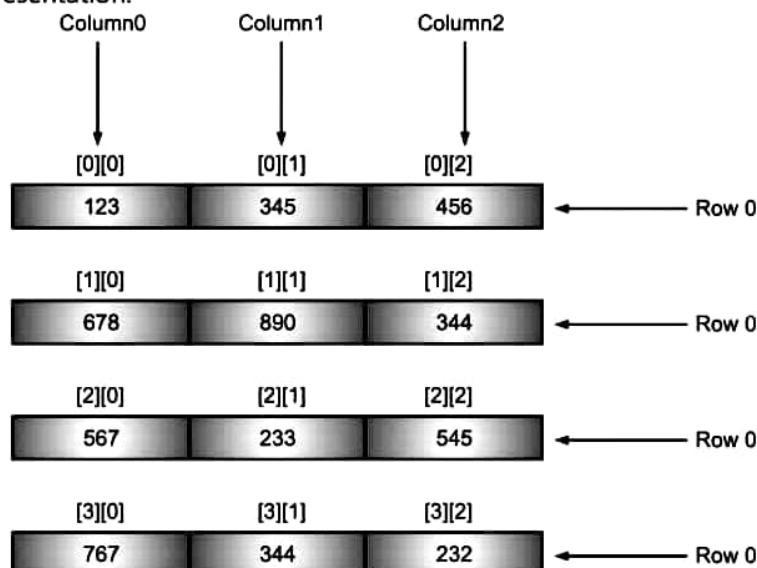


Fig. 1.13: Memory Representation of Two-Dimensional Array



- The above representation of data can be visualized as :

	Col1	Col2	Col3	Where,	v ₁₁ = 123	v ₄₁ = 767
Row1	v ₁₁	v ₁₂	v ₁₃		v ₁₂ = 345	v ₄₂ = 344
Row2	v ₂₁	v ₂₂	v ₂₃		v ₁₃ = 456	v ₄₃ = 232
Row3	v ₃₁	v ₃₂	v ₃₃		v ₂₁ = 678	
Row4	v ₄₁	v ₄₂	v ₄₃	4 × 3	v ₂₂ = 890	
					v ₂₃ = 344	
					v ₃₁ = 567	
					v ₃₂ = 233	
					v ₃₃ = 545	

- The two-dimensional arrays are initialized as follows :

int a[2][3] = {0,0,0,1,1,1};

It initializes the elements of the first row to zero and second row to one.

Or int a[2][3] = {{0,0,0},{1,1,1}};

1.8.3 Multi-Dimensional Array

- An array having more than two dimensions (sizes) is known as a Multi-dimensional array.
- Syntax :** type array-name [s₁] [s₂] [s₃]... [s_n];
Here, type specifies the type of element that will be contained in the array such as int, float or char, array-name is the name of table or matrix.
- For example, int d[3][5][12]; Here, d is three dimensional array to contain 180 integer type elements.
- For example, float table[5][4][5][3]; Here table is a four dimensional array containing 300 elements of floating point type.

Array Representation

- Arrays may be represented in Row-major form or Column-major form.
- In Row-major form, all the elements of the first row are printed, then the elements of the second row and so on upto the last row.
- In Column-major form, all the elements of the first column are printed, then the elements of the second column and so on upto the last column.
- A 2D array's elements are stored in continuous memory locations. It can be represented in memory using any of the following two ways:
 - Column-major order, and
 - Row-major order
- A 2-dimensional array is having two indices, of which first specifies the number of rows and the second specifies the number of columns of an array.
For example, int A[3][4];



- The above declaration tells us that A is 2-D array having 3 rows and 4 columns. The row and column arrangement of 2-D array forms a matrix like structure.

For example,

```
int A[3][4] = {{19, 20, 31, 41},
                {50, 51, 71, 89},
                {90, 91, 95, 99}}
```

};

The array 'A' is represented as follows:

Rows	Columns			
	0	1	2	3
0	19	20	31	41
1	50	51	71	89
2	90	91	95	99

- Actually, the above matrix elements are stored linearly, since memory of computer can only be viewed as sequential units of memory. There are two possible arrangements of elements in memory, namely Row-major order and Column-major order.
- Row-major order is employed by high-level-programming languages like Pascal, C, Ada etc. Column-major order is used in FORTRAN and various dialects of BASIC.

1. Row-major Representation

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

In this method the elements are stored row wise, i.e. n elements of first row are stored in first n locations, n elements of second row are stored in next n locations and so on.

For example, the previous array A[3][4] would be stored in memory in Row-major order with the array cells continuously stored as shown below.

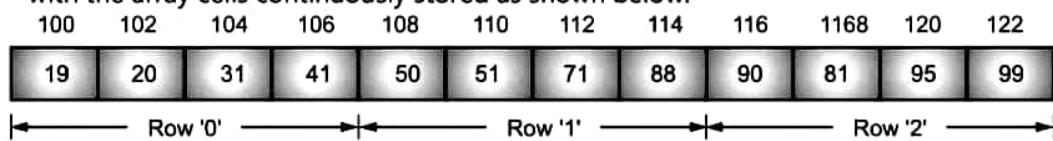


Fig. 1.14: Row-major Representation

The numbers written above the array cells ranging from 100 to 122 shows the memory offset. Here by definition each integer takes two bytes. Hence, the difference between two adjacent offsets is two.

- The linear offset from the beginning of the array to any specific element [row] [column] can be found as:

$$\text{offset} = (\text{row} * \text{n} + \text{column}) * 2$$

where, 'n' is the number of columns in the array.



- 'C' language labels the first element of array index by zero. Row 1, Column 2 in matrix A is represented by A[0][1].

1. The offset for A[0][1] is computed as follows.

$$\begin{aligned}\text{Offset} &= (0 * 4 + 1) * 2 \\ &= 1 * 2 \\ &= 2\end{aligned}$$

Hence, elements 20 (i.e. A[0][1] is at 102, which is nothing but base address offset (i.e. 100 + 2)).

2. The offset for A[2][3] = (2 * 4 + 3) * 2

$$\begin{aligned}&= 11 * 2 \\ &= 22\end{aligned}$$

Element 99 (i.e. A[2][3]) is at 122 which is equal to base address + offset (i.e. 100 + 22)).

2. Column-major Representation

- In column-major form, all the elements of the first column are printed, then the elements of the second column and so on upto the last column.
- The two dimensional array which is stored column by column is said to have column major order on the array.
- In this method the elements are stored column wise, i.e. m elements of first column are stored in first m locations, m elements of second column are stored in next m locations and so on.

For example, the array A[3][4] would be stored in memory in column-major order with the array cells contiguously stored as shown below.

100	102	104	106	108	110	112	114	116	1168	120	122
19	50	90	20	51	91	31	71	95	41	89	99

← Column '0' ← Column '1' → Column '2' → Column '3' →

Fig. 1.15: Column-major Representation

- The numbers written above in array cells shows the memory offset. The difference between two adjacent offsets is two.
- The linear offset from the beginning of the array to any specific element [row] [column] can be found as:

$$\text{offset} = (\text{row} + \text{column} * n) * 2$$

where, 'n' is the number of rows in the array.



1. The offset for A[2][1] (i.e. 91) is computed as follows:

$$\text{offset} = (2 + 1 * 3) * 2$$

$$\text{offset} = 5 * 2 = 10$$

The element 91 (i.e. A[2][1]) is at 110, which is nothing but base address + offset (i.e. 100 + 10).

2. The offset for A[2][2] (i.e. 95) is computed as follows:

$$\text{offset} = (2 + 2 * 3) * 2$$

$$\text{offset} = 8 * 2 = 16$$

The element 95 (i.e. A[2][2]) is at 116, which is nothing but base address + offset (i.e. 100 + 16).

Program 1.2: Program for printing row-major and column-major representation of a matrix array.

```
#include<conio.h>
#include<stdio.h>
main()
{
    int a[3][4];
    int i, j;
    clrscr();
    printf("\n Enter 12 elements:");
    for(i=0; i<=2; i++)
        for(j = 0; j<= 3; j++)
            scanf("%d", & a[i][j]);
    printf("\n Row major array \n");
    for(i=0; i<=2; i++)
    {
        printf("\n Row %d:", i);
        for(j=0; j<=3; j++)
            printf("%d", a[i][j]);
    }
    printf("\n Column major array \n");
    for(i=0; i<=3; i++)
    {
        printf("\n Column %d:", i);
        for(j =0; j < 2; j++)
            printf("%d", a[j][i]);
    }
}
```

**Output:**

```
Enter 12 elements: 1 2 3 4 5 6 7 8 9 10 11 12
Row major array
Row 0 : 1 2 3 4
Row 1 : 5 6 7 8
Row 2 : 9 10 11 12
Column major array
Column 0 : 1 5 9
Column 1 : 2 6 10
Column 2 : 3 7 11
Column 3 : 4 8 12
```

1.9 POLYNOMIALS

[Oct. 16, 17]

1.9.1 Polynomial Representation

- Polynominal consists of following functions :
 1. Polynominal Zero() : return the polynomial $p(x) = 0$
 2. Boolean IsZero(poly) : if (poly) return FALSE else TRUE
 3. Coefficient Coef(poly, expon) : if(expon \in poly) return its coefficient else return zero
 4. Exponent LeadExp(poly) : return the largest exponent in poly
 5. Polynomial Attach(poly, coef, expon) : if(expon \in poly) return error else return the polynomial poly with the term $\langle \text{coef}, \text{expon} \rangle$ inserted
 6. Polynomial Remove(poly, expon) : if(expon \in poly) return the polynomial poly with the term whose exponent is expon deleted else return error
 7. Polynomial SingleMult(poly, coef, expon) : return the polynomial $\text{poly} \cdot \text{coef} \cdot x^{\text{expon}}$
 8. Polynomial Add(poly1, poly2) : return the polynomial $\text{poly1} + \text{poly2}$
 9. Polynomial Mult(poly1, poly2) : return the polynomial $\text{poly1} \cdot \text{poly2}$
- We want to represent the polynomial,

$$A(x) = a_m x^{e_m} + \dots + a_2 x^{e_2} + a_1 x^{e_1}$$



- Where a_i are non-zero coefficients with exponents e_i such that $e_m > e_{m-1} \dots > e_2 > e_1 > = 0$.
- Polynomial can be represented using array and linked list.
- Each term of a polynomial is one structure.
- Structure is a collection of different data elements of different data types i.e. different elements of different datatypes (called member or element of structure) are combined under one name.

For example, structure to store student details is defined as follows:

```
struct student
{
    int roll_no;
    char name[20];
    int age;
};
```

- To represent polynomial using array, we use array of structures i.e. element of an array is a structure.

```
struct student s[10];
```
- Here 'S' is an array of size 10 i.e. we can store details of 10 students. Element of an array is a structure student.

	rollno	name	age
0			
1			
2			
3			
4			
5			
6			
7			
8			
9			

S

- Polynomial can be represented using array as follows:

```
struct term
{
    float coeff;
    int exp;
};

struct term poly[10];
```



- Poly is an array which can store polynomial having maximum 10 terms.

For example, polynomial $3x^5 - 2x^3 + 1$ can be stored as,

	coeff	pw
0	3	5
1	-2	3
2	1	0
3		
4		
5		
6		
7		
8		
9		

- Polynomial can be represented using linked list.

1.9.2 Polynomial Evaluation

[Oct. 17]

The function traversal of singly linked list can be used with few modifications for polynomial evaluation.

```
/* function for evaluating polynomial */
double evaluate (struct node *head, double val)
{
    double result = 0;
    while(head != null)
    {
        result += (head → coef) * pow(val, head → exp);
        head = head → next;
    }
    return result
}
/* pow is library function use to compute  $a^b$  */
main()
{
    struct node * head,
    double ans;
    head = createpoly()
    .
    .
    .
    ans = evaluate (head, 6)
}
```



1.9.3 Polynomial Addition using Array

[April 16, Oct. 16]

- Let two polynomials A and B be,

$$A = 4x^9 + 3x^6 + 5x^2 + 1$$

$$B = 3x^6 + x^2 - 2x$$

- The polynomial A and B are to be added to yield polynomial C. The assumption here is the two polynomials are stored in array.

Program 1.3: To add two single variable polynomials and evaluate them.

```
/* Program to add two single variable polynomials and evaluate them. */
#include <stdio.h>
#define size 10
typedef struct
{
    float coeff;
    int pw;
    int flag;
} POLY;
void read_poly(POLY p[], int n);
int add_poly(POLY p1[], int n1, POLY p2[], int n2, POLY p3[]);
int find_pos(POLY p[], int n, int pw1);
void print_poly(POLY p[], int n);
float calculate_power(float x, int pw1);
float evaluate_poly(POLY p[], int n, float val);
main()
{
    POLY p1[size], p2[size], p3[size];
    int n1, n2, n3;
    float val, res;
    printf("\n Enter the no.of terms for first polynomial : ");
    scanf("%d", &n1);
    printf("\n Enter the terms for first polynomial : ");
    read_poly(p1, n1);
    printf("\n Enter the no.of terms for second polynomial : ");
    scanf("%d", &n2);
    printf("\n Enter the terms for second polynomial : ");
    read_poly(p2, n2);
    printf("\n First polynomial is : ");
    print_poly(p1, n1);
    printf("\n Second polynomial is : ");
    print_poly(p2, n2);
    n3=add_poly(p1, n1, p2, n2, p3);
    printf("\n Addition is : ");
    print_poly(p3, n3);
    printf("\n Enter the value for x : ");
    scanf("%f", &val);
    res=evaluate_poly(p1, n1, val);
    printf("\n After evaluating first polynomial : %2.2f", res);
    res=evaluate_poly(p2, n2, val);
    printf("\n After evaluating second polynomial : %2.2f", res);
```



```
res=evaluate_poly(p3,n3,val);
printf("\n After evaluating third polynomial : %2.2f",res);
}
void read_poly(POLY p[],int n)
{
    int i;
    for(i=0;i<n;i++)
    {
        printf("\n Enter coefficient : ");
        scanf("%f",&p[i].coeff);
        printf("\n Enter power : ");
        scanf("%d",&p[i].pw);
        p[i].flag=0;
    }
}
void print_poly(POLY p[],int n)
{
    int i=0;
    for(i=0;i<n-1;i++)
    {
        if(p[i].coeff!=0)
            printf("%2.2f[x^%d] + ",p[i].coeff,p[i].pw);
    }
    printf("%2.2f[x^%d]\n",p[i].coeff,p[i].pw);
}
int find_pos(POLY p[],int n,int pw1)
    /*to find the index of term in second polynomial */
    /*having same power as that of first polynomial */
{
    int i=0;
    while(i<n)
    {
        if(p[i].pw==pw1)
            return(i);
        i++;
    }
    return(-1);
}
float calculate_power(float x, int n)
{
    int i=0,m;
    float p=1;
    if(n<0)
        m=-n;
    else
        m=n;
    while(i<m)
    {
        p=p*x;
        i++;
    }
    if(n>=0)
```



```
    return(p);
else
return(1/p);
}
int add_poly(POLY p1[],int n1,POLY p2[],int n2, POLY p3[])
{
    int i=0,j=0,k=0,n3;
    do
    {
        j=find_pos(p2,n2,p1[i].pw);
        if(j==-1)      /* second polynomial does not contain term */
                        /*having same power as that of first polynomial */
        {
            p3[k].pw=p1[i].pw;
            p3[k].coeff=p1[i].coeff;
            p1[i].flag=1;
            k++; i++;
        }
        else
        {
            p3[k].pw=p1[i].pw;
            p3[k].coeff=p1[i].coeff+p2[j].coeff;
            p1[i].flag=1;
            p2[j].flag=1;
            i++;
            k++;
        }
    }while(i<n1);
    j=0;
    while(j<n2) /* to copy terms from second polynomial */
                  /*not having same power term in first polynomial */
    {
        if(p2[j].flag==0)
        {
            p3[k].pw=p2[j].pw;
            p3[k].coeff=p2[j].coeff;
            k++;
        }
        j++;
    }
    n3=k;
    return(n3);
}
float evaluate_poly(POLY p[],int n,float val)
{
    int i;
    float res=0;
    for(i=0;i<n;i++)
        res=res+(p[i].coeff * calculate_power(val,p[i].pw));
    return res;
}
```

**Output:**

```
Enter the no.of terms for first polynomial : 3
Enter the terms for first polynomial :
Enter coefficient : 3
Enter power : 2
Enter coefficient : -4
Enter power : 4
Enter coefficient : 1
Enter power : 2
Enter the no.of terms for second polynomial : 2
Enter the terms for second polynomial :
Enter coefficient : 2
Enter power : 2
Enter coefficient : -2
Enter power : 4
First polynomial is : 3.00[x^2] + -4.00[x^4] + 1.00[x^2]
Second polynomial is : 2.00[x^2] + -2.00[x^4]
Addition is : 5.00[x^2] + -6.00[x^4] + 3.00[x^2]
Enter the value for x :2
After evaluating first polynomial : -48.00
After evaluating second polynomial : -24.00
After evaluating third polynomial : -64.00
```

- The two polynomials A and B are stored in two linked lists with pointers Aptr and Bptr pointing to first node of each polynomial respectively.
- To add these two polynomials, let us use the same method as that of paper-pencil method. Let us use these two pointers Aptr and Bptr to move along the terms of A and B.

1.10 SELF REFERENTIAL STRUCTURE

[April 17]

- It is one in which one or more of its components is a pointer to itself.
- It usually require dynamic storage management routines (malloc and free) to explicitly obtain and release memory.
- For example:

```
typedef struct {
    char data;
    struct list *link;
} list;
```
- Each instance of the structure list will have two components.
 1. Data, and
 2. Link.
- Data is a single character and link is a pointer to a list structure.
- Consider these statements, which create three structures and assign values to their respective fields :

```
list item1, item2, item3;
item1.data = 'x';
item2.data = 'y';
item3.data = 'z';
item1.link = item2.link = item3.link = NULL;
```



- Structures item 1, item 2 and item 3 each contain the data item x, y and z respectively.
- We can also store address of one item into the other as follows:

```
item1.link = & item2;  
item2.link = &item3;  
item3.link = NULL;
```

Practice Questions

1. What is data structures?
2. Differentiate between primitive and non-primitive data structures.
3. Explain different types of data structures.
4. Describe the term 'Primitive Data Structure'. Enlist four types of Primitive data structures.
5. Why we required data structures?
6. How to implement a data structure? Explain in brief.
7. What is a pointer?
8. With suitable example describe dynamic memory allocation.
9. Explain the term; Algorithm Analysis.
10. Enlist various Asymptotic notation.
11. What is time complexity and space complexity?
12. Describe the term ADT.
13. How to implement a data structure?
14. Describe the following asymptotic notations :
 - (i) Omega notation
 - (ii) Θ notation
15. Enlist various operation of data structure.
16. Compare space and time complexity.
17. Define array.
18. How to declare and initialize a array ?
19. Enlist various types of array.
20. What is a structure ?
21. What is mean by polynomial ?
22. What is meant by searching ?
23. Explain the term self-referential structures.
24. What is searching ?
25. Compare array and structures.
26. With suitable example describe Linear search.
27. With suitable example describe Binary search.
28. How to add two polynomials ?
29. Describe Internal representation of structure.
30. Compare Linear and Binary search.
31. How to represent a polynomial ?
32. Explain evaluation of polynomials.





Chapter 2...

Searching and Sorting Techniques

- 2.1 Linear Search
- 2.2 Binary Search (Recursive, Non-Recursive)
- 2.3 Introduction To Sorting
- 2.4 Bubble Sort
- 2.5 Insertion Sort
- 2.6 Selection Sort
- 2.7 Quick Sort
- 2.8 Heap Sort (No Implementation)
- 2.9 Merge Sort
- 2.10 Analysis Of All Sorting Techniques
 - Practice questions
 - University Questions and Answers

2.1 LINEAR SEARCH

[Oct. 15, 16, 17; April 17]

Internal and External Keys

- A key may be contained within the record, like name in telephone directory example. Such key may be at specific offset from the start of the record. Such a key is called an **internal key** or an **embedded key**.
- Now consider the index of a book. Here index is a separate table of keys that include page numbers (as pointers to) of the records. Such keys, which are maintained in separate tables with pointers to actual records, are called as **external keys**.
- In day-to-day life there are various applications, in which the process of searching is to be carried such as, searching a name of person from the given list, searching a specific card from the of set cards, etc.
- For example, consider a personal telephone diary. This diary is used to store phone/ mobile numbers of friends, relatives, etc. For searching the phone number of a person one can directly search the number by using indexing in that diary.

**Definition:**

- Searching is a technique of finding an element from the given data list or set of elements like an array, list or trees.
- Searching can be used to find the given data from a sorted or unsorted list.
- While searching, list is searched or some value. For a given field. This field is called as a **key** or **search key**.

For example, a telephone directory records have 3 fields - name, address and telephone number. If search is done on name, then name is search key.

Internal and External searching:

- If the search is applied on the table, which resides at secondary storage (hard disk) is called **external searching**.
- Where as searching of a table which is in primary storage (main memory) is called **internal searching**.
- Internal searching is faster than external searching.

Result of Searching

- A searching algorithm accepts two arguments as parameters, one a target value to be searched and the second the list in which it is to be searched.
- The search algorithm searches a target value in the list, until the target key is found or can conclude that it is not found.

Primary key and Secondary key:

- Sometimes it may be desirable to have more than one key is searching for a given information. For example, suppose there are 10,000 employees in a company. We want to search an employee having name 'Rohan'. It may happen that more than one employee has name 'Rohan'. Therefore, some another key, say designation as 'manager' is used for searching.
- Thus name is a primary key and designation is a secondary key.
- Thus the result of searching can be a 'success' or a 'failure'.
- A successful search is called 'Retrieval'. When a search is successful, we may either return the record or return a pointer to the record.
- When a search fail, we may either print an appropriate message and terminate searching or insert a record with search key as its key.
- A technique of searching the record and if it is not found then insert it, is called search and insert algorithm.
- Searching can be classified into two types:
 1. Linear or sequential search
 2. Binary search.



2.2 LINEAR OR SEQUENTIAL RESEARCH

- Sequential search is performed in a linear way i.e. it starts from the beginning of the list and continues till the item is found or reaches the end of the list.
- The item (key value) to be searched is compared with each element of the list one by one, starting from the first element.

For example, consider following array having 20 values,

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
arr	96	19	5	85	16	29	2	10	22	7	12	66	75	82	95	23	93	11	40	59

Fig. 2.1: Array

Suppose we want to search for value 12. This value is compared with arr[0], arr[1], ..., arr[10]. Since, the value is found at index 10, the search is successful.

Now suppose we want to find value 56. This value is not found, so the search is unsuccessful.

Program 2.1: Program for sequential search:

```
# include <stdio.h>
int main()
{
    int arr[20];
    int i,n, no;
    printf("Enter n: ");
    scanf("%d", &n);
    printf("\nEnter %d numbers:\n", n);
    for(i=0;i<n;i++)
        scanf("%d", &arr[i]);
    printf("\nEnter the number to be searched: ");
    scanf("%d", &no);
    for(i=0;i<n;i++)
    {
        if(arr[i]==no)
        {
            printf("\nNumber is found at location %d.", i+1);
            break;
        }
    }
    if(i==n)
    printf("\nNumber is not found.");
    return 0;
}
```

**Output:**

```
Enter n: 6
Enter 6 numbers:
12 23 1 56 43 9
Enter the number to be searched: 1
Number is found at location 3.
```

- Let, us find out the amount of time the sequential search needs to search a target data. For this we must find the number of comparisons of keys that it makes. In general, for any search algorithm, the computational complexity is computed by considering number of comparisons made.
- The number of comparisons depends on where the target data is stored in the search list. If the target data is placed at the first location, we get it in just one comparison. Two comparisons are needed if target data is in second location, i comparisons if at i^{th} location and n comparisons if at the n^{th} location.
- As the total number of comparisons depends on the position of target data, let us compute average complexity of algorithm. Average complexity is sum of comparisons for each position of target data divided by n .

Hence,

$$\begin{aligned}\text{Average number of comparisons} &= (1 + 2 + 3 + \dots + n) / n \\ &= (\Sigma n) / n \\ &= ((n(n+1)) / 2) \times 1/n \\ &= (n+1) / 2\end{aligned}$$

- Hence, the average number of comparisons done by the sequential search method in case of a successful search is $(n+1)/2$. The unsuccessful search is given by n comparisons. The number of comparisons is of the order of n denoted as $O(n)$.
- The worst-case complexity is n means target data element is at n^{th} location, requires n comparisons. The best-case complexity is 1, as the target data element is at the first location and requires single comparison. Sequential search is suitable when the data is stored in unordered manner, and also suitable when we have no way to directly access the data elements.
- For example, to search the data record stored on the magnetic tape has to be searched sequentially from first location till n^{th} location. The linear list implemented using linked list have no way to access any i^{th} element directly except ($i = 1$). We need to search through whole linked list to search a target data.
- Hence, sequential search is used if data is unsorted and if the storage does not provide direct access to the data.



- **Pros and Cons of Sequential Search:**

Pros:

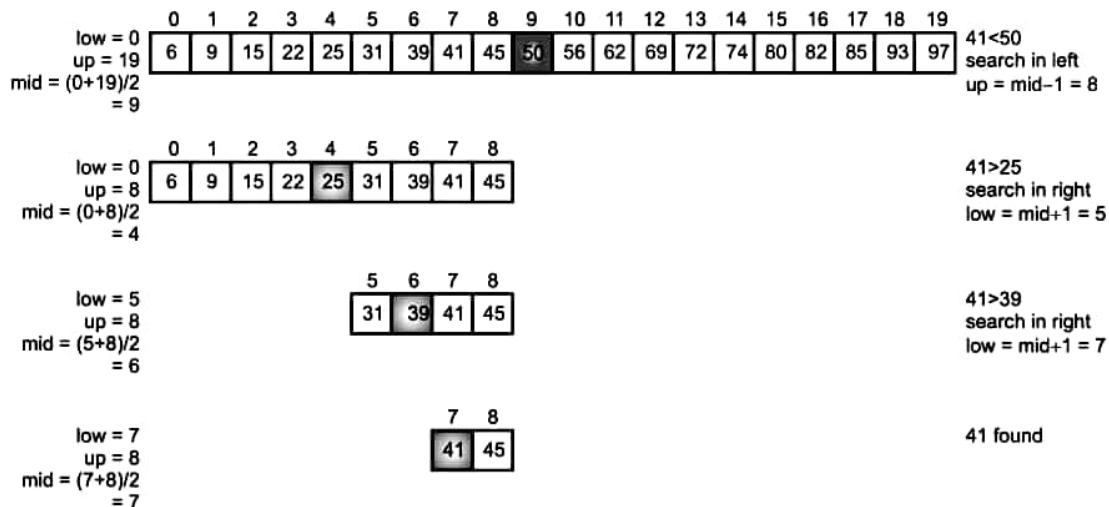
1. A simple and easy method.
2. Efficient for only small lists.
3. Better to use for unsorted data.
4. Suitable for storage structure, which do not support direct access to data example, magnetic tape.
5. Best case one comparison, worst-case n comparisons and average case $(n+1)/2$ comparisons.
6. Complexity is in the order of n denoted as $O(n)$.

Cons:

1. Highly inefficient for large data.
2. Other search techniques such as binary search are found more suitable than sequential search for ordered data.

2.2 BINARY SEARCH (RECURSIVE, NON-RECURSIVE) [April 17, Oct. 17]

- Sequential search is not suitable for larger list. It requires ' n ' comparisons in worst case.
 - **For binary search, the list or array should be sorted in ascending order.**
 - First we compare the item to be searched with the middle element of an array. If the item is found there, search is successful. Otherwise array is divided into two halves, first halve (left) contains all elements smaller than the middle element and the elements in the second halve (right) will be greater than the middle element.
 - If the item to be searched is less than the middle element, it is searched in left half, otherwise it is searched in right half.
 - The process of comparing item with the middle element and dividing the array continues till we find require item or get a portion which does not contain any element.
 - Algorithm for binary search is as follows:
 1. Start
 2. Let ' n ' be the size of the array.
 3. Let 'item' be the item to be search.
 4. Let $low = 0$ which is lower limit of an array.
 upper = $n - 1$ which is upper limit of an array.
 5. Calculate $mid = (low + upper)/2$.
 6. If $item > arr[mid]$
 search will resume in right half which is $arr[mid + 1], \dots, arr[up]$
 So, $low = mid + 1$
 7. If $item < arr[mid]$
 Search will resume in left half which is $arr[low], \dots, arr[mid - 1]$.
 So, $up = mid - 1$
 8. If $item = arr[mid]$, search is successful.
 9. If $low > up$, search is unsuccessful.
- For example, take a sorted array of 20 elements.

**Search 41 (i.e. item = 41)****Fig. 2.2: Binary Search****Program 2.2:** Implementing non-recursive binary search is as follows:

```
# include <stdio.h>
int main()
{
    int arr[20];
    int i, n, no;
    clrscr();
    printf("Enter n: ");
    scanf("%d", &n);
    printf("\nEnter sorted list of %d numbers:\n", n);
    for(i=0;i<n;i++)
        scanf("%d", &arr[i]);
    printf("\nEnter the number to be searched: ");
    scanf("%d", &no);
    bsearch(arr, no, 0, n-1);
    return 0;
}
bsearch(int a[], int no, int lb, int ub)
{
    int mid;
    if(lb > ub)
        printf("Given number is not found.");
}
```



```
else
{
    mid=(lb+ub)/2;
    if(a[mid]==no) // No. found
        printf("Given number is found at %d position.",mid+1);
    else
    {
        if(a[mid] > no) // to recursively search in left part
            bsearch(a, no, lb, mid-1);
        else
            bsearch(a, no, mid+1, ub); // to recursively search in
            //right part
    }
}
```

Output:

```
Enter n: 6

Enter sorted list of 6 numbers:
1 9 12 23 43 56

Enter the number to be searched: 23
Given number is found at 4 position.
```

- The best case of binary search is when item to be searched is present in the middle of the array and in this case the loop is executed only once. Thus best case complexity is O(1).
- The worst case is when the item is not present in the array. In each iteration, the array is divided into half, so if the size of array is 'n', there will be maximum \log^n such divisions. Thus there will be \log^n comparisons and so complexity O($\log n$).
- The average complexity of binary search is O($\log n$).

Pros and Cons of Binary Search:

1. Suitable for sorted data.
2. Efficient for large lists.
3. Not usable for unsorted data.
4. Suitable for storage structure which supports direct access to data.
5. Not usable for storage structure which do not support direct access to data example, magnetic tape and linked list.
6. Complexity is in the order of n denoted as O($\log_2(n)$).
7. Inefficient for small lists.



2.3 INTRODUCTION TO SORTING

- As soon as you create a significant collection of data, you will probably think of reason to sort it. You need to arrange names in alphabetical order, students by grade, customers by zip code, house sales by price, cities in order of increasing population and so on.
- Sorting of data may also be a preliminary step to searching it. Because sorting is so important and potentially so time consuming, that it has been the subject of extensive research in computer science and some sophisticated methods have been developed like bubble sort, heap sort, radix sort and so on.
- Sorting refers to the operation of arranging a set of data in some given order. Order can be ascending or descending.

Types Of sorting:

- The methods of sorting can be divided into two categories.
 - Internal Sorting, and
 - External Sorting.
 - Internal Sorting:** If all the data that is to be sorted can be adjusted at a time in main memory, then internal sorting methods are used.
 - External Sorting:** When the data to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory (hard disk, floppy, tape etc.) then external sorting methods are used.

Stability of Sorting:

- It may happen that the key on which data is being sorted is not unique for each record i.e. two or more records have identical key values.
- A sorting algorithm is said to be stable if it maintains the relative order of the duplicate keys in the sorted output. For example, if record R_i and R_j have equal key values and record R_i precedes record R_j in the input data then R_i should precede R_j in the sorted output data.
- Consider Fig. 2.3, sorting done on Age,

Name	Age	Name	Age	Name	Age
Shraddha	19	Rekha	18	Rekha	18
Amit	20	Deepa	19	Shraddha	19
Kiran	19	Kiran	19	Kiran	19
Rekha	18	Shraddha	19	Deepa	19
Geeta	21	Amit	20	Amit	20
Deepa	19	Geeta	21	Geeta	21

Unsorted list Sorted list
(Unstable sort) Sorted list
(Stable sort)

Fig. 2.3: Stability of Sorting



Indirect Sort (Sort by Address)

- Sorting can be done in two ways, by actually moving the records or by maintaining an auxillary array of pointer and rearranging the pointer in that array.
- The process of sorting by adjusting pointers is called sorting by address or indirect sort because the record is indirectly referred in sorted order.
- If the records are elements of an array then we can store the indices of these elements in an auxillary array.
- Consider Fig. 2.4 in which sorting is done by moving record.

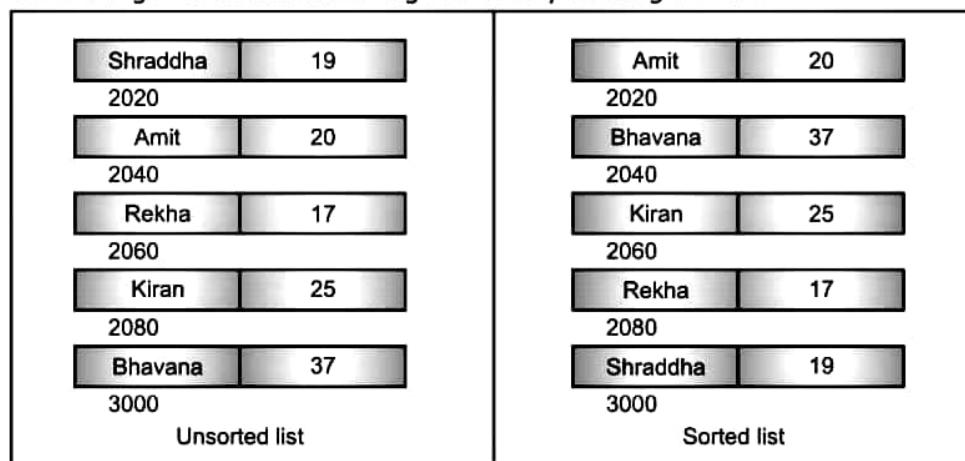


Fig. 2.4: Sorting by moving records

- Consider Fig. 2.5, in which sorting is done on name by rearranging pointers.

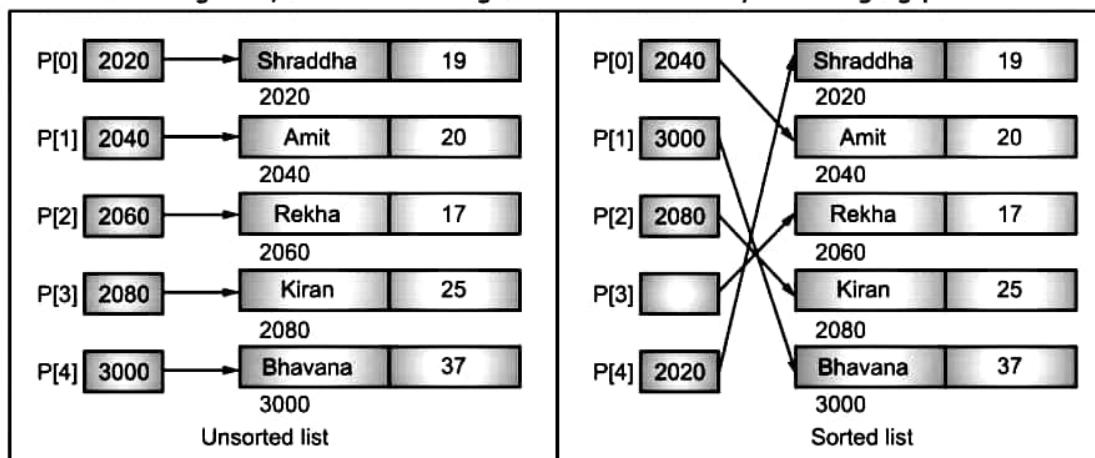


Fig. 2.5: Sorting by rearranging pointers

**In place sort:**

- In place sorting methods generally use the same storage that is occupied by input data to store the output data while sorting. These methods do not need any extra storage except for working storage used to store program variables. For example, bubble sort, insertion sort, selection sort are in place sort methods.
- Other sorting methods may need extra storage to store intermediate results of the sorting and at last sorted data is copied back to original storage.
- For example, merge sort is not in place sort because it requires an extra array of size n to sort an array of size n .

2.4 BUBBLE SORT

[April 15, Oct. 17]

- The algorithm in bubble sort involves two steps, executed over and over until the data is sorted.
 1. Compare two adjacent items.
 2. If necessary, swap (exchange) them.
- In this sorting method, to arrange elements in ascending order, we begin with 0th element and compare it with the 1st element. If it is found to be greater than 1st element then they are interchanged. Compare 1st element with 2nd element and so on. In this way all the elements are compared with their next element and are interchanged if required.
- On completing the first iteration or pass, the largest element gets placed at the last position. Similarly in the second iteration second largest element gets placed at the second last position and so on. As a result after all the iterations the last becomes a sorted list.

Original array	1 st Iteration	2 nd Iteration	3 rd Iteration	4 th Iteration
23	15	15	11	1
15	23	11	1	11
29	11	1	15	15
11	1	23	23	23
1	29	29	29	29



- Contents of array at every pass or iteration is shown in Fig. 2.6.

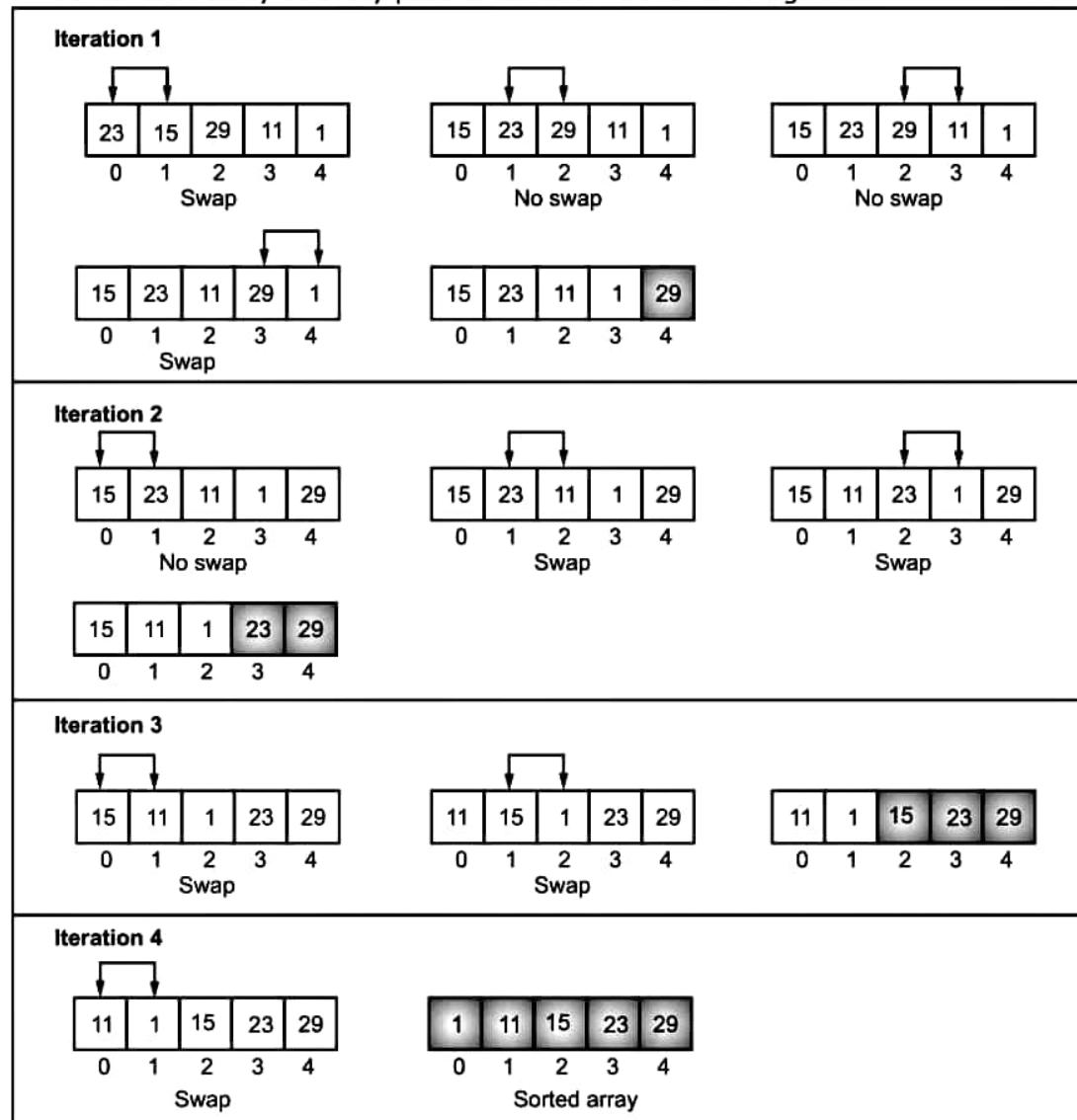


Fig. 2.6: Bubble sort

Program 2.3: Program for Bubble Sort

```
#include <stdio.h>
int main()
{
    int arr[50];
    int i, j, n, temp, k, pass;
```



```
clrscr();
printf("Enter n: \n");
scanf("%d",&n);
printf("Enter array elements:");
for(i=0;i<n;i++)
scanf("%d",&arr[i]);
printf("\nUnsorted array is:\n");
for(i=0;i<n;i++)
printf("%d ",arr[i]);
for(pass=0;pass<n-1;pass++)
{
    for(j=0;j<n-1-pass;j++)
    {
        if(arr[j]>arr[j+1])
        {
            temp=arr[j];
            arr[j]=arr[j+1];
            arr[j+1]=temp;
        }
    }
    printf("\nArray after pass %d:\n",pass+1);
    for(k=0;k<n;k++)
    printf("%d ",arr[k]);
}
printf("\nSorted array is:\n");
for(i=0;i<n;i++)
printf("%d ",arr[i]);
return 0;
}
```

Output:

```
Enter n:
6
Enter array elements:
90 78 56 3 12 9

Unsorted array is:
90 78 56 3 12 9
Array after pass 1:
78 56 3 12 9 90
Array after pass 2:
56 3 12 9 78 90
```



```
Array after pass 3:  
3 12 9 56 78 90  
Array after pass 4:  
3 9 12 56 78 90  
Array after pass 5:  
3 9 12 56 78 90  
Sorted array is:  
3 9 12 56 78 90
```

- **Time Complexity:** The performance of bubble sort in worst case is $n*(n - 1)/2$. This is because in the first pass $n - 1$ comparisons or exchanges are made; in second pass $n - 2$ comparisons are made. This is carried out until the last exchange is made. The mathematical representation for these exchange will be equal to

$$(n - 1)(n - 2) + \dots + (n - (n - 1))$$

Thus, the expression becomes $n*(n - 1)/2$. Thus, the number of comparisons is proportional to (n^2) . The time complexity of bubble sort is $O(n^2)$.

- Bubble sort is stable in place of sorting technique.

2.5 INSERTION SORT

[April 15, Oct. 16]

- The insertion sort is substantially better than bubble sort. It is about twice as fast as the bubble sort.
- Insertion sort is performed by inserting a particular element at the appropriate position.
- In insertion sort, the first iteration starts with comparison of 1st element with the 0th element.
- In the second iteration, 2nd element is compared with the 0th and 1st element. In general in every iteration, an element is compared with all elements.
- If at same point it is found that the element can be inserted at a position then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position. This procedure is repeated for all the elements in the array.
- The advantage of insertion sort is its simplicity. It is very efficiency when number of elements to be sorted are very less.
- A disadvantage of this sorting is the number of movements. The elements of the sorted part also have to move to make place for another element.
- Insertion sort is stable and in-place sorting method.
- Content of array at every iteration is shown in Fig. 2.7.

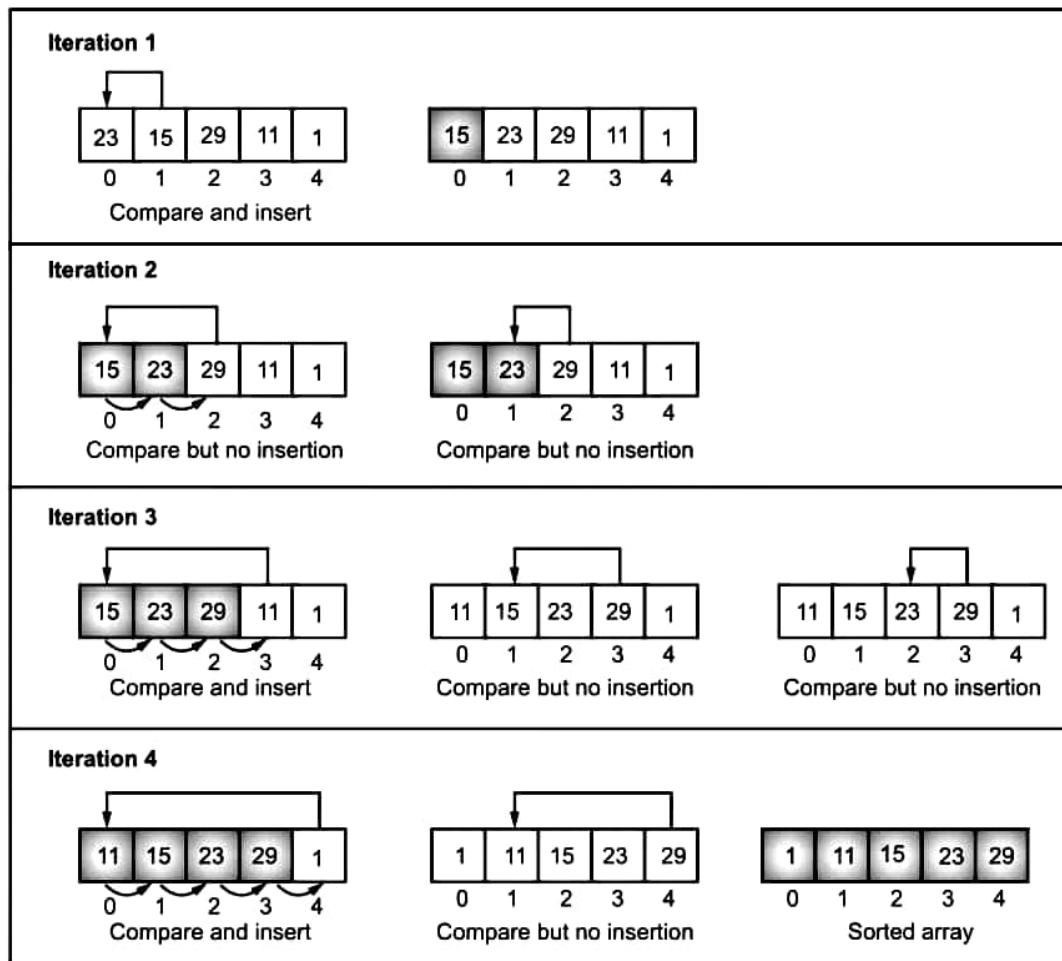


Fig. 2.7: Insertion sort

Program 2.4: Program for insertion sort is as follows:

```
# include <stdio.h>
int main()
{
    int arr[50];
    int i,j,n,temp,k;
    clrscr();
    printf("\nEnter n: ");
    scanf("%d",&n);
    printf("\nEnter %d no. of array elements: ",n);
    for(i=0;i<n;i++)
        scanf("%d",&arr[i]);
    printf("\nUnsorted array is:\n");
```



```
for(i=0;i<n;i++)
    printf("%d ",arr[i]);
for(j=1;j<n;j++)
{
    temp=arr[j];
    printf("\n New Element: %d",temp);
    for(i=j-1;i>=0 && temp<arr[i];i--)
    {
        arr[i+1]=arr[i];
    }
    arr[i+1]=temp;
    printf("\nAfter pass %d:\n",j);
    for(k=0;k<n;k++)
        printf("%d ",arr[k]);
    }
    printf("\nSorted array is:\n");
    for(i=0;i<n;i++)
        printf("%d ",arr[i]);
    return 0;
}
```

Output:

```
Enter n: 6
Enter 6 no. of array elements:
12 4 52 1 7 89
Unsorted array is:
12 4 52 1 7 89
New Element: 4
After pass 1:
4 12 52 1 7 89
New Element: 52
After pass 2:
4 12 52 1 7 89
New Element: 1
After pass 3:
1 4 12 52 7 89
New Element: 7
After pass 4:
1 4 7 12 52 89
New Element: 89
After pass 5:
1 4 7 12 52 89
Sorted array is:
1 4 7 12 52 89
```

**Time Complexity:**

- The outer for loop will always have $n-1$ iterations.
- The iterations of the inner for loop will vary according to the data. For each iteration of outer for loop, the inner for loop executes 0 to i times.
- The best case occurs if the list is in sorted order. In each iteration of outer for loop, element $arr[i]$ is always in proper place, thus only one comparison to be done. Therefore for ' n ' no. of elements, total $n-1$ comparisons to be done i.e. complexity is $O(n)$.
- The worst case occurs when the list is in reverse order. In first iteration 1 comparison is done in second iteration 2 comparisons and in $(n - 1)^{th}$ iteration there will be $(n - 1)$ comparisons. So total no. of comparisons will be,

$$\sum_{i=1}^{n-1} i = 1 + 2 + 3 + \dots + (n - 1) \approx \frac{n(n - 1)}{2}$$

Therefore the complexity is $O(n^2)$.

- For average case, when data is in any order, average no. of comparisons in i^{th} iteration is,

$$\frac{1 + 2 + 3 + \dots + (i - 1) + i + i}{i + 1} = \frac{i}{2} + 1 - \frac{1}{(i + 1)}$$

Therefore total number of comparisons $\approx \frac{n(n - 1)}{4} + n - \log n$. Thus complexity is $O(n^2)$.

2.6 SELECTION SORT

[April 16, Oct. 17]

- The selection sort is the easiest method of sorting. In this sort the data is arranged in ascending order.
- The 0^{th} element is compared with all other elements, if the 0^{th} element is found to be greater than the compared element then they are interchanged. In this way after the first iteration the smallest element is placed at 0^{th} position. The procedure is repeated for 1^{st} element and so on.
- Selection sort is simple to implement and requires only one temporary variable for swapping elements.
- The main advantage of selection sort is that data movement is very less.
- It is not a stable sort. It is an in-place sort.

Original array	1st Iteration	2nd Iteration	3rd Iteration	4th Iteration
23	1	1	1	1
15	23	11	11	11
29	29	29	15	15
11	15	23	29	23
1	11	15	23	29



- Content of array at every iteration are shown in Fig. 2.8.

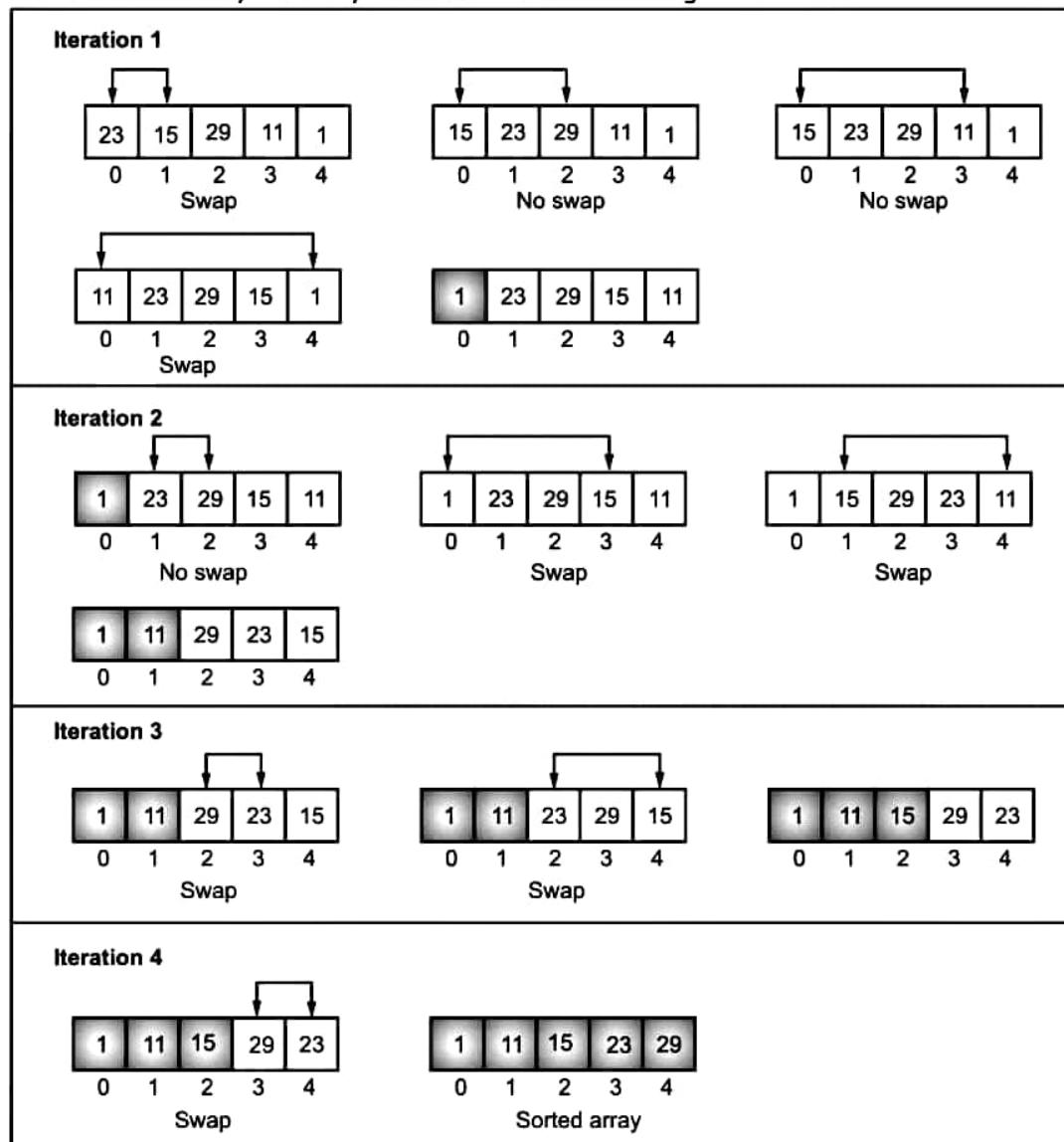


Fig. 2.8: Selection Sort

Program 2.5: Program for insertion sort is as follows:

```
# include <stdio.h>
int main()
{
    int arr[50];
    int i,j,n,temp,k,pass;
    clrscr();
    printf("Enter n: \n");
    scanf("%d", &n);
    for(i=1;i<n;i++)
    {
        pass=0;
        for(j=i;j>0;j--)
        {
            if(arr[j]<arr[j-1])
            {
                temp=arr[j];
                arr[j]=arr[j-1];
                arr[j-1]=temp;
                pass=1;
            }
        }
        if(pass==0)
        {
            break;
        }
    }
    for(i=0;i<n;i++)
    {
        printf("%d ", arr[i]);
    }
}
```



```
scanf("%d",&n);
printf("Enter array elements:");
for(i=0;i<n;i++)
scanf("%d",&arr[i]);
printf("\nUnsorted array is:\n");
for(i=0;i<n;i++)
printf("%d ",arr[i]);
for(pass=0;pass<n-1;pass++)
{
    for(j=pass+1;j<n;j++)
    {
        if(arr[pass]>arr[j])
        {
            temp=arr[pass];
            arr[pass]=arr[j];
            arr[j]=temp;
        }
    }
    printf("\nArray after pass %d:\n",pass+1);
    for(k=0;k<n;k++)
    printf("%d ",arr[k]);
}
printf("\nSorted array is:\n");
for(i=0;i<n;i++)
printf("%d ",arr[i]);
return 0;
}
```

Output:

```
Enter n:
5
Enter array elements:
77 33 44 11 66
Unsorted array is:
77 33 44 11 66
Array after pass 1:
11 77 44 33 66
Array after pass 2:
11 33 77 44 66
Array after pass 3:
11 33 44 77 66
Array after pass 4:
11 33 44 66 77
Sorted array is:
11 33 44 66 77
```

**Another way for selection sort is as follows:**

```
# include <stdio.h>
int main()
{
    int arr[50];
    int i,j,n,temp,k,pass,min;
    clrscr();
    printf("Enter n: \n");
    scanf("%d",&n);
    printf("Enter array elements:");
    for(i=0;i<n;i++)
        scanf("%d",&arr[i]);
    printf("\nUnsorted array is:\n");
    for(i=0;i<n;i++)
        printf("%d ",arr[i]);
    for(pass=0;pass<n-1;pass++)
    {
        min=pass;
        for(j=pass+1;j<n;j++)
        {
            if(arr[min]>arr[j])
                min=j;
        }
        if(pass!=min)
        {
            temp=arr[pass];
            arr[pass]=arr[min];
            arr[min]=temp;
        }
        printf("\nArray after pass %d:\n",pass+1);
        for(k=0;k<n;k++)
            printf("%d ",arr[k]);
    }
    printf("\nSorted array is:\n");
    for(i=0;i<n;i++)
        printf("%d ",arr[i]);
    return 0;
}
```

**Output:**

```
Enter n:  
6  
Enter array elements:  
56 23 44 12 6 9  
Unsorted array is:  
56 23 44 12 6 9  
Array after pass 1:  
6 23 44 12 56 9  
Array after pass 2:  
6 9 44 12 56 23  
Array after pass 3:  
6 9 12 44 56 23  
Array after pass 4:  
6 9 12 23 56 44  
Array after pass 5:  
6 9 12 23 44 56  
Sorted array is:  
6 9 12 23 44 56
```

- **Time Complexity:** The performance of sorting algorithm depends upon the number of iterations and time to compare them. The first element is compared with the remaining $n - 1$ elements in pass 1. Then $n - 2$ elements are taken in pass 2. This process is repeated until the last element is encountered. The mathematical expression for these iterations will be equal to $(n - 1) + (n - 2) + \dots + (n - (n - 1))$. Thus the expression becomes $n*(n - 1)/2$. Thus the number of comparisons is proportional to (n^2) . The time complexity of selection sort is $O(n^2)$.

2.7 QUICK SORT

[April 16, 17]

- Quick sort is very popular sorting method. It was introduced by Hoare in 1962.
- It is very important method of internal sorting. According to this algorithm, it is faster and easier to sort two small arrays than one large one.
- The strategy that quick sort follows is of divide and conquer. In this approach, numbers are divided and again subdivided and division goes on until it is not possible to divide them further.
- The procedure it follows is of recursion. It is also known as **Partition Exchange Sort**.
- Quick sort is not a stable sort.
- This algorithm consider one element at a time called pivot and place it in its correct position in such a way that all elements to the left of the pivot are smaller than the pivot and all elements to the right are greater.



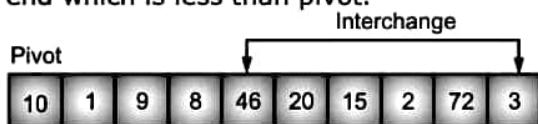
- Place of pivot is gives place for partition i.e. if pivot is place at i^{th} position then array is partitioned as 0 to $i - 1$ and $i + 1$ to n .
- For example, consider on array

10, 1, 9, 8, 46, 20, 15, 2, 72, 3
Pivot

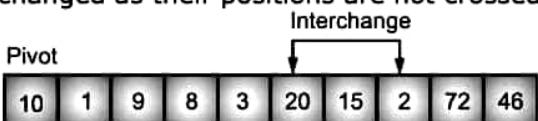


Let, pivot = $a[0] = 10$

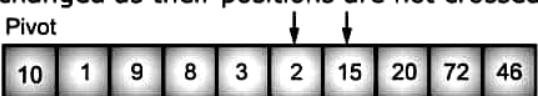
Identify first element from start which is greater than pivot i.e. 10. Also identify first element from end which is less than pivot.



46 and 3 are interchanged as their positions are not crossed each other.



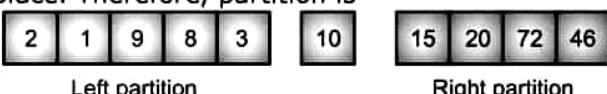
20 and 2 are interchanged as their positions are not crossed each other.



Here 2 and 15 are crossed. Therefore, interchange 10 and 2.



10 is at its right place. Therefore, partition is



Recursively above method is applied on both partitions till further partition is not possible.

Program 2.6: Program for Quick sort.

```
# include <stdio.h>
int main()
{
    int arr[50];
    int i,j,temp,k,n;
    clrscr();
    printf("Enter n: \n");
    scanf("%d",&n);
    printf("Enter array elements:");
    for(i=0;i<n;i++)
        scanf("%d",&arr[i]);
    printf("\nUnsorted array is:\n");
```



```
for(i=0;i<n;i++)
printf("%d ",arr[i]);
quicksort(arr,0,n-1);
printf("\nSorted array is:\n");
for(i=0;i<n;i++)
printf("%d ",arr[i]);
return 0;
}
quicksort(int arr[], int lb, int ub)
{
    int j,i;
    if (lb < ub)
    {
        j=partition(arr,lb,ub); // to find the place for partition
        quicksort(arr,lb,j-1); // sort left part
        quicksort(arr,j+1,ub); //sort right part
    }
}
int partition(int arr[],int lb, int ub)
{
    int down,up,temp,pivot;
    if(lb < ub)
    {
        down=lb;
        up=ub;
        pivot=arr[lb]; // First element of partition is a pivot
        while(down < up)
        {
            while(arr[up]>pivot && up > lb)
                up--;
            while(arr[down]<=pivot && down < ub)
                down++;
            if(down < up)
            {
                temp=arr[down];
                arr[down]=arr[up];
                arr[up]=temp;
            }
        }
        temp=arr[lb];
        arr[lb]=arr[up];
        arr[up]=temp;
        return(up); // up indicates correct place for pivot element
    }
}
```

**Output:**

```
Enter n:  
6  
Enter array elements:  
12 4 55 33 78 3  
Unsorted array is:  
12 4 55 33 78 3  
Sorted array is:  
3 4 12 33 55 78
```

- **Time Complexity:** The efficiency of quick sort depends upon the selection pivot. Assume in which pivot middle is thus the total elements at left and right are equal. After the pass1 is completed there will be two parts having equal number of elements i.e. $n/2$ elements. In this way we will be proceeding further until the list is sorted. The number of comparisons in different passes will be as follows. Pass1 will have n comparisons. Pass2 will have $2*(n - 1)$ comparisons. In the subsequent passes will have $4*(n/4)$, $8*(n/8)$ comparisons and so on. The total comparisons involved in this case would be $0(n) + 0(n) + 0(n) + \dots + s$. The value of expression will be $O(n \log n)$. Thus, time complexity of quick sort is $O(n \log n)$.

2.8 HEAP SORT

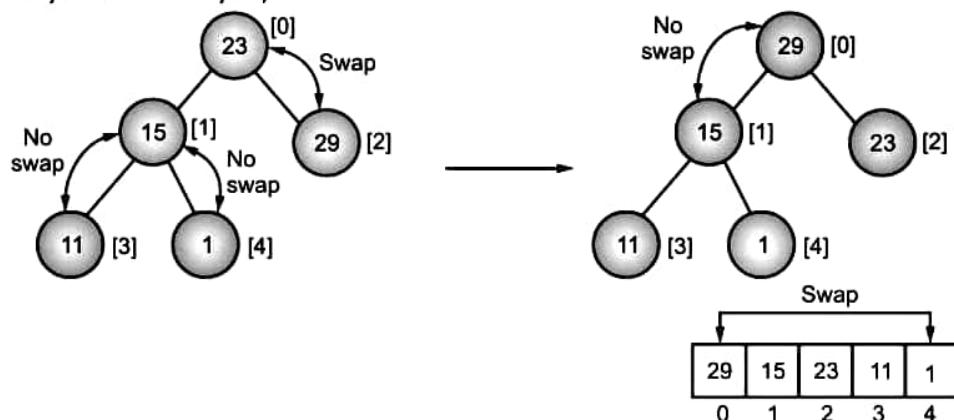
[April 16, Oct. 17]

- The heap sort is based on a tree structure that reflects peaking order in a corporate hierarchy. For e.g., in an organisation when President retires, Vice-President becomes President.
- There are 2 types of heap trees-ascending (min) heap and descending (max) heap.
- An ascending heap is an almost complete binary tree such that the contents of each node is greater than or equal to the content of its father. That is, in min heap, the root contains the largest element of the list or heap.
- Heap sort proceeds in two phases:
 1. The entries are arranged in the list satisfying the requirements of a heap.
 2. We repeatedly remove the top of the heap and promote another entry to take its place.
- In second phase we recall that the root of the tree has the largest key. This key belongs to the end of the list.
- For example, consider an array of to 5 elements,

23	15	29	11	1
0	1	2	3	4



Binary tree of array is,



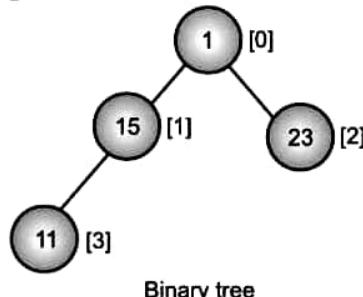
Number in square indicates index of the element. First step is to create a max heap. To do this, children node value is compared with parent node, if larger than parent, then swap it start from end of an array.

Swap first and last element. Then array is

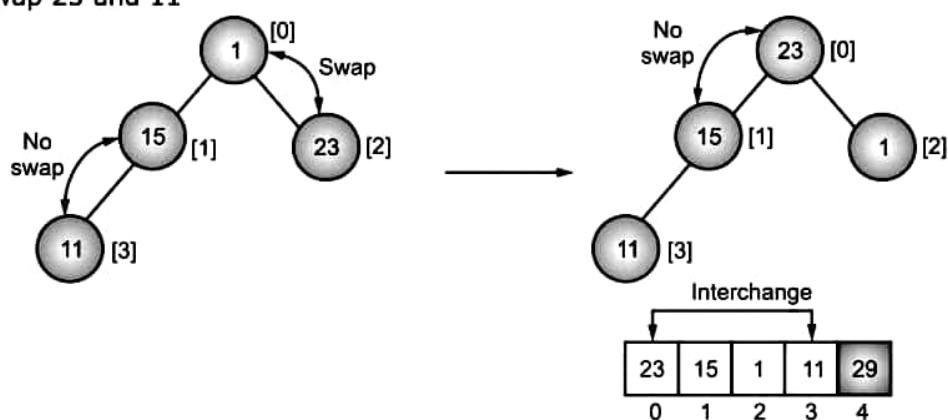
Now apply second step. That is again generate tree without considering '2g' element.

1	15	23	11	29
0	1	2	3	4

Now create a max heap again.



Swap 23 and 11

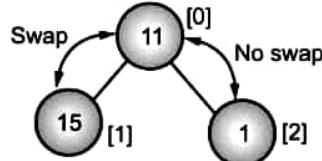




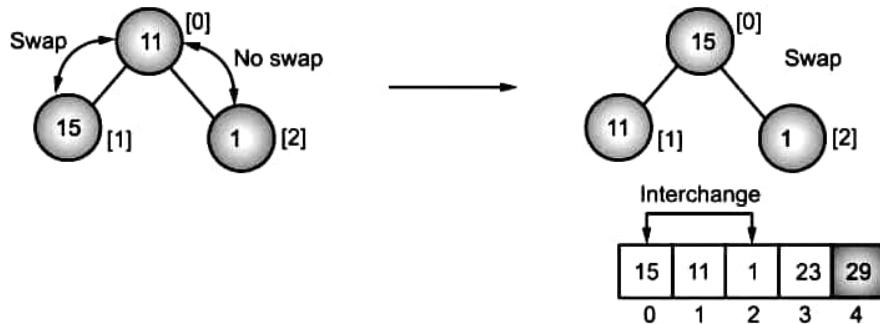
To crate max heap tree

11	15	1	23	29
0	1	2	3	4

Swap 15 and 1



To create max heap tree,



Swap 1 and 11

1	11	15	23	29
0	1	2	3	4

- **Time complexity:** We know that the depth of the complete binary tree is $(\log_2 n)$. In worst case the number of comparisons in each step would be equal to the depth of the tree. The number of comparisons for the above observation would be $O(n \log_2 n)$. Thus the number of comparisons in this method would be $O(n \log_2 n)$.
- Heap sort is good for larger lists but it is not preferable for small list of data.
- It is not a stable sort. But it is in-place sort.

2.9 MERGE SORT

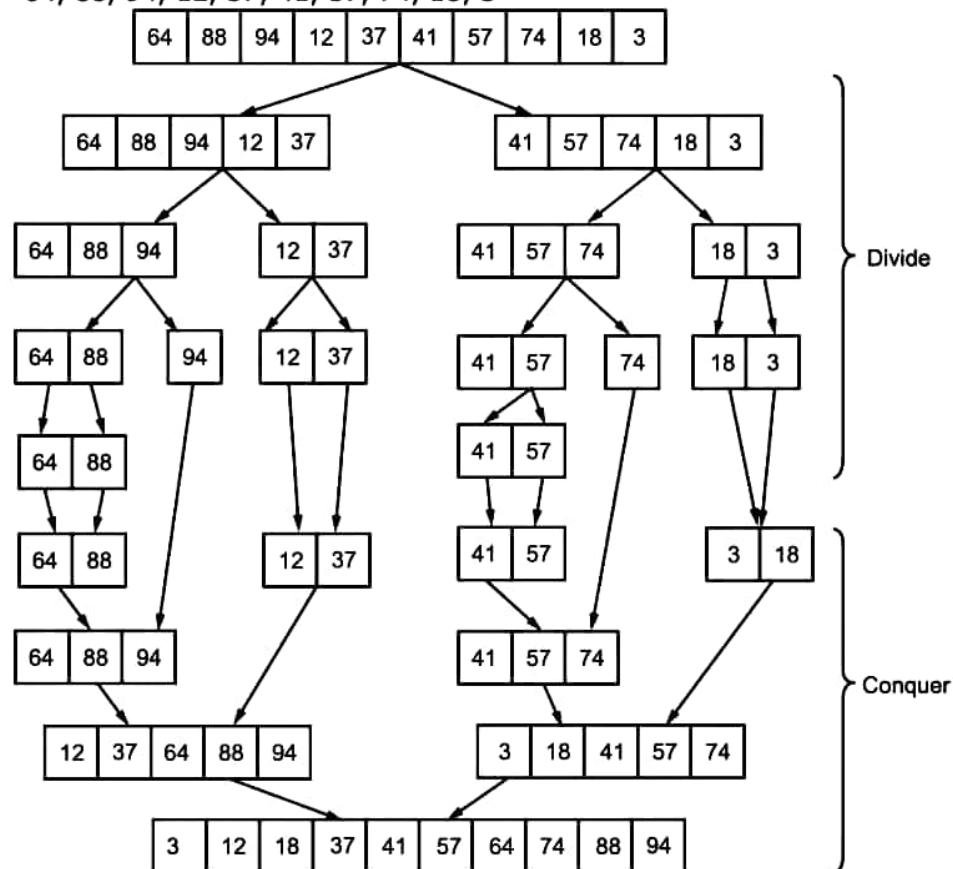
[Oct. 16]

- Merging is the process of combining two sorted list into single sorted list. To perform the merge sort both the sorted lists are compared.
- The smaller of both the elements is stored in the third array. The sorting completes when all the elements from both the lists are placed in the third list.
 $X = 1 \ 9 \ 10 \ 11 \ 46$
 $Y = 0 \ 2 \ 15 \ 20 \ 72$
 $Z = 0 \ 1 \ 2 \ 9 \ 10 \ 11 \ 15 \ 20 \ 46 \ 72$
- Merge sort can be done recursively. The recursive definition is based on the divide and conquer technique. The list is recursively divided into sublist till the single element list is left (which is obviously sorted). Then the sublists are merged repeatedly to get a single sorted list.



- For example, consider an array

64, 88, 94, 12, 37, 41, 57, 74, 18, 3



Program 2.7:

```
# include <stdio.h>
int n;
int main()
{
    int arr[50];
    int i,j,temp,k;
    clrscr();
    printf("Enter n: \n");
    scanf("%d",&n);
    printf("Enter array elements:");
    for(i=0;i<n;i++)
        scanf("%d",&arr[i]);
    printf("\nUnsorted array is:\n");
    for(i=0;i<n;i++)
```



```
printf("%d ",arr[i]);
mergesort(arr,0,n-1);
printf("\nSorted array is:\n");
for(i=0;i<n;i++)
printf("%d ",arr[i]);
return 0;
}

mergesort(int arr[],int lb, int ub) // to partition the list
{
int mid,k;
if(lb < ub)
{
    mid=(lb+ub)/2;
    mergesort(arr,lb,mid);
    mergesort(arr,mid+1,ub);
    merge(arr,lb, mid, ub);
}
}

merge(int arr[], int lb, int mid, int ub)
{
    int i,j,k;
    int temp[50];
    for(i=lb;i<=ub;i++)
    temp[i]=arr[i];
    i=lb;
    j=mid+1;
    k=lb;
    while(i<=mid && j <=ub)
    {
        if(temp[i]<temp[j]) // element of first sublist is smaller
        than second
        {
            arr[k]=temp[i];
            i++;k++;
        }
        else
        {
            arr[k]=temp[j];
            j++;k++;
        }
    }
}
```



```

        while(i<=mid) // To copy remaining elements of first sublist
    {
        arr[k]=temp[i];
        i++;k++;
    }
    while(j<=ub) // To copy remaining elements of second sublist
    {
        arr[k]=temp[j];
        j++;k++;
    }
}

```

Output:

```

Enter n:
6
Enter array elements:
12 56 2 9 33 27
Unsorted array is:
12 56 2 9 33 27
Sorted array is:
2 9 12 27 33 56

```

Time Complexity

- In quick sort also use divide and conquer approach but in this algorithm partitioning is difficult process. Whereas in merge sort combining is difficult.
- In merge sort, 'n' element list is divided repeatedly into half approximately $\log n$ times and at end we get n sublist of size 1.
- In each pass, there will be merging of n elements which is $O(n)$. Therefore, complexity is $O(n \log n)$.
- Merge sort is stable sort but not in place sort as it requires $O(n)$ extra space.

2.10 ANALYSIS OF ALL SORTING TECHNIQUES

Name	Average Case	Worst Case	Method	Advantage/Disadvantage
Bubble Sort	$O(n^2)$	$O(n^2)$	Exchange	1. Straightforward, simple and slow. 2. Stable. 3. Inefficient on large tables. 4. Good for small $n < 100$

contd. ...



Insertion Sort	$O(n^2)$	$O(n^2)$	Insertion	<ul style="list-style-type: none"> 1. Efficient for small list and mostly sorted list. 2. Sort big array slowly. 3. Save memory
Selection Sort	$O(n^2)$	$O(n^2)$	Selection	<ul style="list-style-type: none"> 1. Improves the performance of bubble sort and also slow. 2. Unstable but can be implemented as a stable sort. 3. Quite slow for large amount of data. 4. Good for partially sorted data.
Heap Sort	$O(n \log n)$	$O(n \log n)$	Selection	<ul style="list-style-type: none"> 1. More efficient version of selection sort. 2. No need extra buffer. 3. As efficient as quick sort.
Merge Sort	$O(n \log n)$	$O(n \log n)$	Merge	<ul style="list-style-type: none"> 1. Well for very large list, stable sort. 2. A fast recursive sorting. 3. Both useful for internal and external sorting. 4. It requires an auxiliary array that is as large as the original array to be sorted
Quick Sort	$O(n^2)$	$O(n \log^2 n)$	Insertion	<ul style="list-style-type: none"> 1. Efficient for large list. 2. It requires relative small amount of memory, extension of insertion sort. 3. Not stable but in place.

Practice Questions

1. What is meant by sorting ?
2. Describe internal and external sorting.
3. What is a Bubble sort ?
4. What is a Merge sort ?
5. What is a Quick sort ?



- 6 Consider the following set of 10 numbers
85, 36, 87, 10, 91, 18, 15, 52, 73, 49
Sort the array using
(A) Insertion Sort (B) Quick Sort
7. Differentiate between Internal Sort and External Sort.
8. Explain the Merge Sort with example.
9. Compare the performance of selection and insertion sort.
10. Explain heap sort with example.
11. Create initial Heap for following input elements:
5, 22, 17, 35, 38, 28, 40, 42
-
- ■ ■



Chapter 3...

Linked List

Contents ...

- 3.1 Introduction
- 3.2 Static and Dynamic Representation
- 3.3 Types of Linked List
 - 3.3.1 Singly Linked List (All type of operation)
 - 3.3.2 Doubly Linked List (Create, Display)
 - 3.3.3 Circularly Singly List (Create, Display)
 - 3.3.4 Circularly Doubly Linked List (Create, Display)
 - Practice Questions
 - University Questions and Answers

3.1 INTRODUCTION

- Linked list is a special list of same data elements linked to one another. The logical ordering is represented by having each element pointing to the next element.
- Each element is called a node. Node has two parts: Data part which stores the information and Pointer which points to the next element.

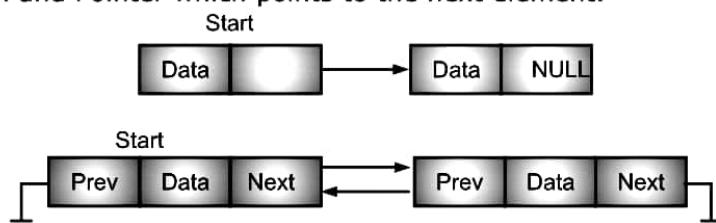


Fig. 3.1: Single linked list and double linked list

- In single linked list, nodes have one pointer (Next) pointing to the next node, whereas nodes of double linked list have two pointers (Prev and Next). Prev points to the previous node and Next points to the next node in the list.

3.1.1 What is Linked List?

[Oct. 16]

- A linked list is ordered collection of data in which each element contains the data and link to its successor (and predecessor).

(3.1)



- In a linked list, each item is allocated space as it is added to the list.
- A link is kept with each item to the next item in the list.
- It is linear data structure.

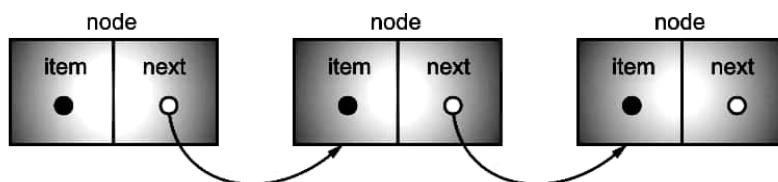


Fig. 3.2: Linear Data Structure

- Each node of the list has two elements:
 1. The item being stored in the list
 2. A pointer to the next item in the list
- As items are added to a list, memory for a node is dynamically allocated. Thus the number of items that may be added to a list is limited only by the amount of memory available.
- A linked list is a non-sequential collection of data items called nodes. Each node in a linked list has basically two fields:
 - o Data field
 - o Link field
- The data field contains an actual value to be stored and processed and the link field contains the address of the next data item in the linked list. The address used to access a particular node is known as a pointer.
 1. **NULL Pointer:** The link field of the last node contains NULL rather than a valid address. It is a null pointer and indicates the end of the list.
 2. **External Pointer:** It is a pointer to the very first node in the linked list, it enables us to access the entire linked list.
 3. **Empty List:** If the nodes are not present in a linked list, then it is called an empty linked list or simply empty list. It is also called the null list. The value of the external pointer will be zero for an empty list.

3.1.2 Advantages

1. **Linked lists are dynamic data structures:** i.e. they can grow or shrink during the execution of a program. Whereas arrays are static i.e. they cannot grow or shrink during the execution.
2. **Efficient memory utilization:** Here, memory is not defined (pre-allocated). Memory is allocated whenever it is required. And it is de-allocated (removed) when it is no longer needed, which is not possible in array.
3. **Insertion and deletions are easier and efficient:** Linked lists provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many complex applications can be easily carried out with linked lists.



3.1.3 Disadvantages

1. **More memory required:** If the number of fields are more, then more memory space is needed.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming.
3. Linked list requires pointer manipulation which is not required for array.

3.1.4 Representation

- Suppose we want to store list of integer numbers, then the linear linked list can be represented in memory with the following declarations:

```
struct node
{
    int num;
    struct node *next;
};

typedef struct node NODE;
NODE *start;
```

- The above declaration defines a new data type, whose each element is of type node.

3.1.5 Operations on Linked List

- The basic operations to be performed on the linked list are as follows:
 1. **Creation:** This operation is used to create a linked list, here the constituent node is created as and when it is required and linked to the list to preserve the integrity of the list.
 2. **Insertion:** This operation is used to insert a new node in the linked list at the specified position. A new node may be inserted.
 - o At the beginning of a linked list
 - o At the end of a linked list
 - o At the specified position in a linked list
 - o If the list itself is empty, then the new node is inserted as a first node.
 3. **Deletion:** This operation is used to delete an item (a node) from the linked list. A node may be deleted from the
 - Beginning of a linked list
 - End of a linked list
 - Specified position in the list
 4. **Traversing:** It is a process of going through all the nodes of a linked list from one end to the other end. If we start traversing from the very first node towards the last node, it is called forward traversing. If the desired element is found we signal operation "**Successful**". Otherwise, we signal it as "**Unsuccessful**".



5. Concatenation: It is the process of appending (joining) the second list to the end of the first list consisting of m nodes. When we concatenate two lists, the second list has n nodes, and then the concatenated list will be having $(m+n)$ nodes. The last node of first list is now pointing to the first node of the second list.

6. Display: This operation is used to print each and every node's information. We access each node from the beginning of the list and output the data.

3.2 STATIC & DYNAMIC REPRESENTATION

3.2.1 Static Representation

- The linked list is maintained by two linear arrays—one is used for data and the other for links both are of same size. Let DATA and LINK be the two arrays, DATA contains the information part, and their corresponding pointers to the next node are stored in the array LINK. A pointer variable HEAD is used to store the first location of the linked list, and a Null pointer is used to denote the end of the list. Since array subscripts are positive integers, Null is represented by '0'. Fig. 3.3 illustrates the static representation of a singly linked list.
- Address of next or successor node is stored in LINK array at same index as that of the current node.
- In static representation, memory utilization is less as array is of fixed size.
- It is suitable when exact number of elements are to be stored.
- Program for static representation linked list (singly) is,

Program 3.1: Static Implementation of Singly Linked List.

```
# include <stdio.h>
struct linked_list
{
    int data;
    int next;
}list[20];
int main()
{
    int i,choice,head=-1,n,no=0,pos,index,j,prev;
    for(i=0;i<20;i++)
    {
        list[i].data=-1;
        list[i].next=-1;
    }
    do
    {
        printf("\n1. Insert\n2. Delete\n3. Display\n4. Exit\n");
        printf("Enter ur choice :");
        scanf("%d",&choice);
```



```
switch(choice)
{
    case 1: printf("\nEnter the number to insert :");
               scanf("%d", &n);
               if(head== -1)
               {
                   list[0].data=n;
                   head=0;
                   no++;
               }
               else
               {
                   printf("\nEnter the position number
                           (start from 0) :");
                   scanf("%d", &pos);
                   if(pos>no)
                       printf("\nWrong position");
                   else
                   {
                       for(index=0;index < 20 && list[index].data !=
                           = -1;index++);
                           if(index==20)
                           {
                               printf("\nArray is full.");
                           }
                           else
                           {
                               list[index].data=n;
                               no++;
                               if(pos==0)
                               {
                                   list[index].next=head;
                                   head=index;
                               }
                               else
                               {
                                   j=0;
                                   prev=0;
                                   i=head;
                                   while(j<pos && i!= -1)
                                   {
                                       if(i==head)
                                           head=i;
                                       else
                                           prev.next=i;
                                       prev=i;
                                       i=list[i].next;
                                       j++;
                                   }
                                   list[index].next=i;
                               }
                           }
                   }
               }
           }
```



```
        prev=i;
        i=list[i].next;
        j++;
    }
    list[prev].next=index;
    list[index].next=i;
}
}
}
}
break;
case 2: printf("\nEnter number to be deleted : ");
scanf("%d",&n);
if(list[head].data==n)
{
    head=list[head].next;
}
else
{
    i=head; prev=head;
    while(i!=-1)
    {
        if(list[i].data==n)
        {
            list[prev].next=list[i].next;
            break;
        }
        else
        {
            prev=i;
            i=list[i].next;
        }
    }
    if(i==-1)
    printf("\n Number not found.");
}
break;
case 3: printf("\n Singly linked list is : ");
for(i=head;i!=-1;i=list[i].next)
{
    printf("%d ",list[i].data);
}
break;
case 4: break;
}
}while(choice!=4);
```

**Output:**

```
1. Insert
2. Delete
3. Display
4. Exit
Enter ur choice : 1
Enter the number to insert : 34

1. Insert
2. Delete
3. Display
4. Exit
Enter ur choice : 1
Enter the number to insert : 56
Enter the position number <start from 0> : 1

1. Insert
2. Delete
3. Display
4. Exit
Enter ur choice :1
Enter the number to insert : 12
Enter the position number <start from 0> : 0

1. Insert
2. Delete
3. Display
4. Exit
Enter ur choice : 3

Singly linked list is : 12 34 56
1. Insert
2. Delete
3. Display
4. Exit
Enter ur choice : 2
Enter the number to insert : 34

1. Insert
2. Delete
3. Display
4. Exit
Enter ur choice : 3
```



```
Singly linked list is : 12 56
1. Insert
2. Delete
3. Display
4. Exit
Enter ur choice :4
```

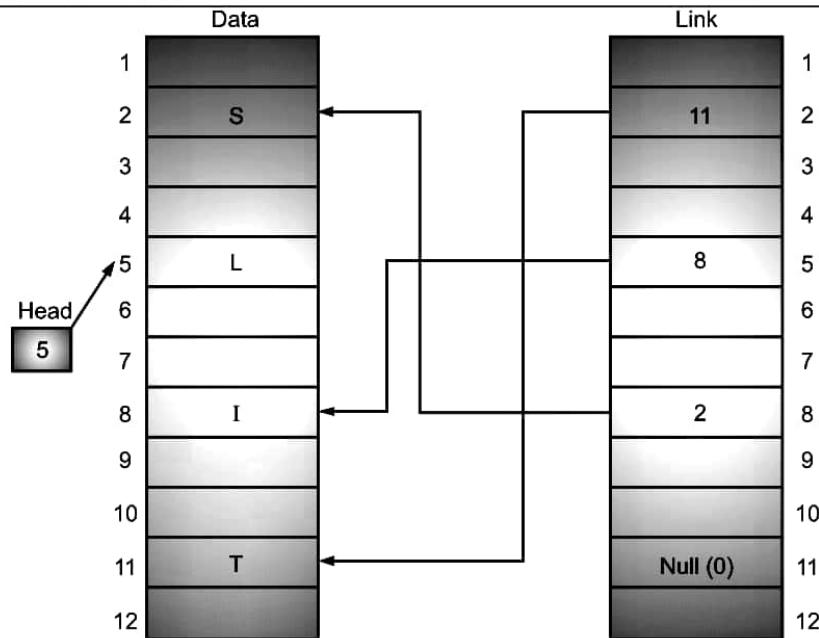


Fig. 3.3: Static representation

Dynamic Representation:

- A linked list is a dynamic data structure i.e. the size of the list grows and shrinks depending upon the operations performed on it.
- When we insert elements in the list, its size increases and when elements are deleted, its size decreases.
- In dynamic representation memory to every node is allocated dynamically using malloc() or calloc() function.
- Every node stores the data and one or more pointers pointing to a node of similar type or of different type.
- Unlike static representation, there is no upper limit on number of nodes in a list.
- As memory is allocated dynamically, it can be released or de-allocated whenever node is deleted.

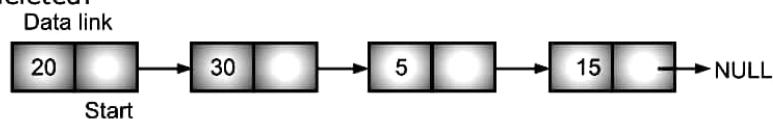


Fig. 3.4: Dynamic Representation of Linked List storing Integer Data



3.3 TYPES OF LINKED LIST

- Basically, there are five types of linked list:

[Oct. 17]

1. Singly Linked List,
2. Doubly Linked List,
3. Circular Linked List, and
4. Circular Doubly Linked List.
5. Generalised Linked List.

1. Singly Linked List: It is one in which all nodes are linked together in some sequential manner. Hence it is called linear linked list. Every node contains a pointer pointing to next node (successor node).

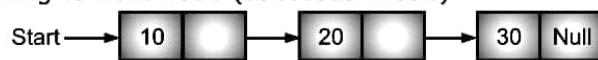


Fig. 3.5 (a): Singly Linked List

2. Doubly Linked List: It is one in which all nodes are linked together by multiple links which help in accessing both the successor node (next node) and predecessor node (previous node) for any arbitrary node within the list. Therefore, each node in a doubly linked list fields (pointers) to the left node (previous) and the right node (next). This helps to traverse the list in the forward direction and backward direction.

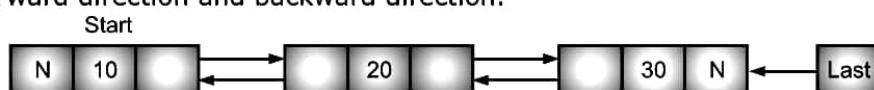


Fig. 3.5 (b): Doubly Linked List

3. Circular Singly Linked List: It is one in which node has no beginning and no end. A singly linked list can be made a circular by simply storing the address of the very first node in the linked field of the last node.

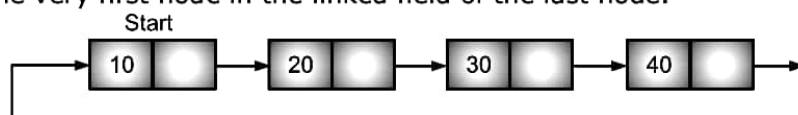


Fig. 3.5 (c): Circular Singly Linked List

4. Circular Doubly Linked List: It is one in which last node's successor pointer points to the first node and first node's predecessor pointer points to last node.

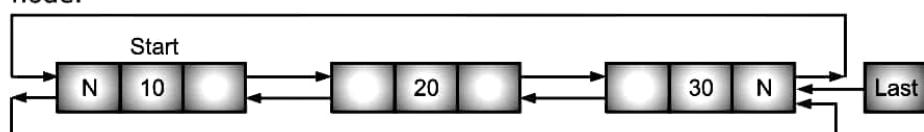


Fig. 3.5 (d): Circular Doubly Linked List



- 5. Generalised Linked List:** Generalised list made up of number of component, some are data elements and other are generalised list (list includes list).

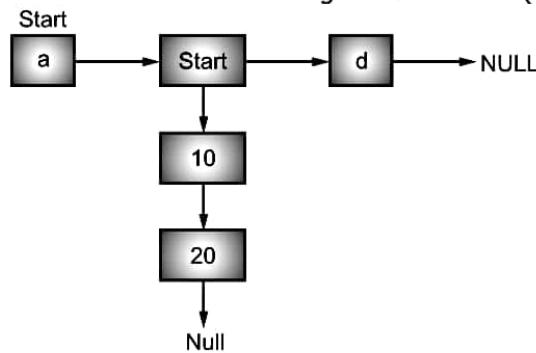


Fig. 3.6: Generalised Linked List

3.3.1 Singly Linked list (All type of operation)

[Oct. 15, April 15, 16, 17]

- A singly linked list is made up of nodes where each node has 2 parts - the first one is the info part that contains the actual data and second one is the link part that points to the next node of the list.

INFO | LINK

- It is a dynamic data structure. It may grow or shrink. Growing or shrinking depends on the operations made.
- In C, a linked list is created using structure, pointers and dynamic memory allocation function **malloc**.

```

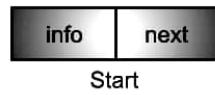
struct node
{
    int info;
    struct node *next;
};

typedef struct node NODE;
NODE *start;
start=(NODE *) malloc (sizeof (NODE));
  
```

- The beginning of the list is marked by a special pointer named 'start' which stores the address of first node.
- The link part of each node points to the next node except the last one which does not point to any node, so store NULL into it.
- The statement,

```
start=(NODE *) malloc (sizeof (NODE));
```

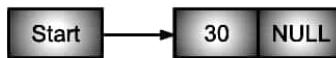
is executed to allocate a block of memory sufficient to store the NODE and assigns address of that block to start.

**Fig. 3.7 (a)**

Now assign values to the respective fields of NODE.

```
Start→num=30;
Start→next=NULL;
```

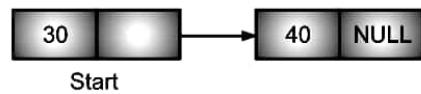
Thus, the node looks like as:

**Fig. 3.7 (b)**

- Any number of nodes can be created and linked to the existing node. Suppose we want to add another node to the above list, then the following statements are required.

```
Start→next=(NODE *) malloc (sizeof (NODE));
Start→next.num=40;
Start→next→next=NULL';
```

Thus, the nodes look like:

**Fig. 3.7 (c)**

- The address of node storing 40 value is stored in link field of the node having value 30.

3.3.1.1 Inserting Nodes

- Steps to insert an element
 - Allocates a node,
 - Assign the data, and
 - Adjust the pointers.

1. Algorithm for inserting a node at the Beginning:

```
Insertfirst(start, item)
1. new node=(node *) malloc (sizeof (node));
   //create new node from memory and assign its address to ptr
   End if
1. set newnode→info= item
2. set newnode→next=start
3. set start=newnode
```

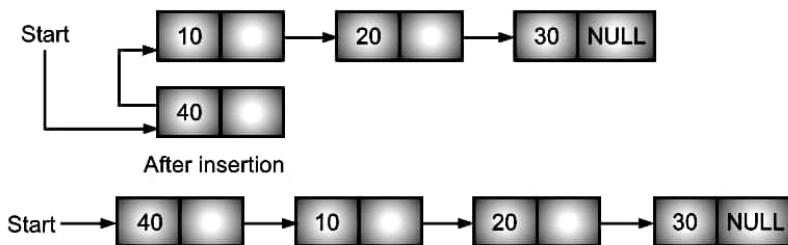


Fig. 3.8: Insert Node at Beginning

- 'C' code for above algorithm:

```
NODE * insertfirst(NODE * start, int num)
{
    NODE *P;
    P= (NODE *) malloc (sizeof(NODE));
    P->info=num; p->next=NULL;
    if (start == NULL)
        P->start=p;
    else
    {
        P->next=start;
        start=p; } return start;
}
```

2. Algorithm for Inserting A Node at the End:

```
insertlast (start, item)
1. ptr=(node * ) malloc (sizeof (node));
2. set ptr->info=item
3. set ptr->next=NULL
4. set loc=start
5. repeat step 6 until loc->next != NULL
6. set loc=loc->next
7. set loc->next=ptr
```

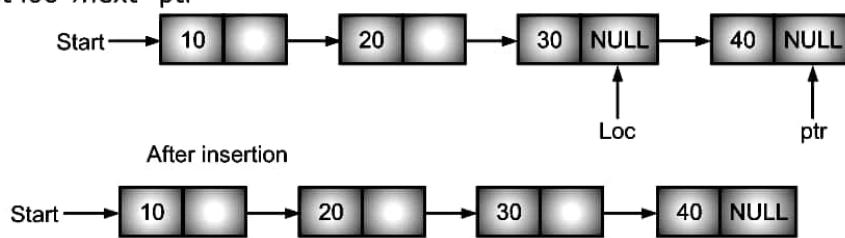


Fig. 3.9: Insert Node at End



- **'C' code for above algorithm:**

```
NODE * insertlast(NODE * start, int item)
{
    NODE *P, *loc;
    P=(NODE *) malloc (sizeof(NODE));
    P->info=item;
    P->next=NULL;
    loc=start;
    while(loc->next!=NULL)
    { loc=loc->next; }
    Loc->next=P;
}
```

3. Algorithm for Inserting A New Node at the specified Position:

Insertloc(start, item, pos)

1. newnode=(node *) malloc(sizeof(node))
2. set new node→info =item
3. initialize the counter i and pointers
(Node * temp)
Set i=0
4. repeat steps 5 and 6 until i < pos and temp! = NULL
5. set temp=temp→next
6. set i=i+1
7. if temp_=NULL then print message "Wrong position"
8. Otherwise,
set newnode p→next=temp→next
set temp→next=newnode

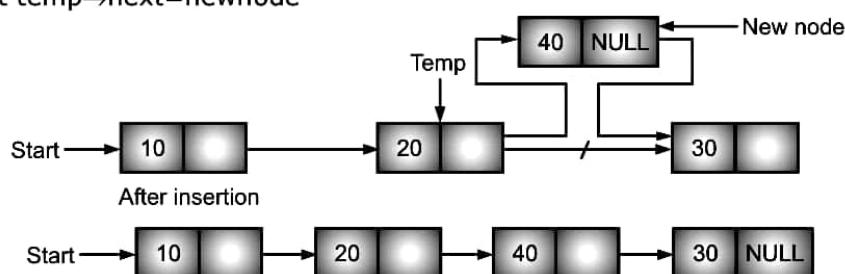


Fig. 3.10: Insert Node in Between



- **'C' code for above algorithm**

```
NODE * insertloc(NODE * start, int item, int pos)
{
    NODE *newnode, *temp;
    int k;
    for(k=0, temp=start; k<loc; k++)
    {
        temp=temp->next;
        if(temp==NULL)
        { printf("wrong Position");
            Return;
        }
    }
    newnode=(NODE *) malloc(sizeof (NODE));
    newnode->info=item;
    newnode->next=temp->next;
    temp->next=newnode;
    return start;
}
```

3.3.1.2 Deleting Nodes

1. Algorithm for deleting the first node:

Deletefirst(start)

1. set Ptr=start
2. set start=start→next, ptr→next=NULL
3. print 'Element deleted is Ptr→info
4. free(Ptr)

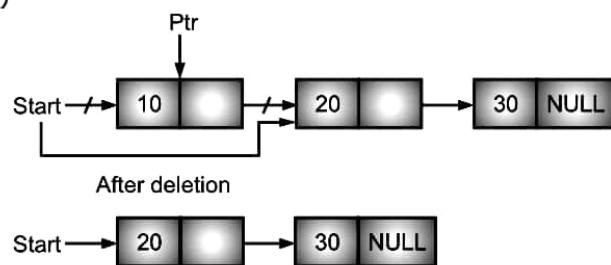


Fig. 3.11: Delete First Node



- **'C' code for above algorithm:**

```
NODE * deletefirst (NODE * start)
{
    NODE *P;
    if(start==NULL)
    { return; }
    else
    {
        P=start;
        start=start->next; p->next=NULL;
        printf("element deleted is %d",P->info);
        free(P);
    }
}
```

2. Algorithm for deleting the Last Node:

```
deletelast(start)
1. if start->next=NULL then
    set Ptr=start
    set start=NULL
    print 'element deleted is ptr->info
    free(Ptr)
    endif
2. set Ptr=start
3. repeat steps 4 and 5 till Ptr->next!=NULL
4. set temp=Ptr
5. set Ptr=Ptr->next
6. set temp->next=NULL
7. free(Ptr)
```

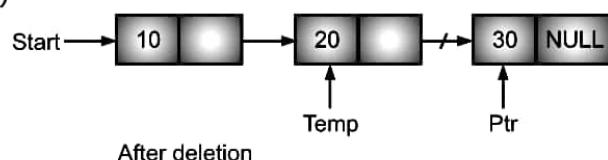


Fig. 3.12: Delete Last Node



- **'C' code for above algorithm:**

```
void deletelast(NODE * start)
{
    NODE *P, *temp;
    If(start==NULL)
    { return; }
    else if (start→next==NULL)
    {
        P=start;
        start=NULL;
        printf("element deleted is=%d",P→info);
        free(P);
    }
    else
    {
        temp=start;
        P=start→next;
        While (P→next!=NULL)
        {
            temp=Ptr;
            P=P→next;
        }
        temp→next=NULL;
        Free(P);
    }
}
```

3. Algorithm for deleting the Node from specified Position:

Deleteloc(start,pos)

1. [initialize the counter i and pointer]
Node * temp, *ptr
Set i=0
Set Ptr=start
2. repeat step 2 to 6 until i <= pos and ptr! = NULL
3. set temp=Ptr



4. set Ptr=Ptr→next
5. set i=i+1
6. if ptr==NULL print message "Wrong position"
7. otherwise print message 'element deleted is ptr→info'

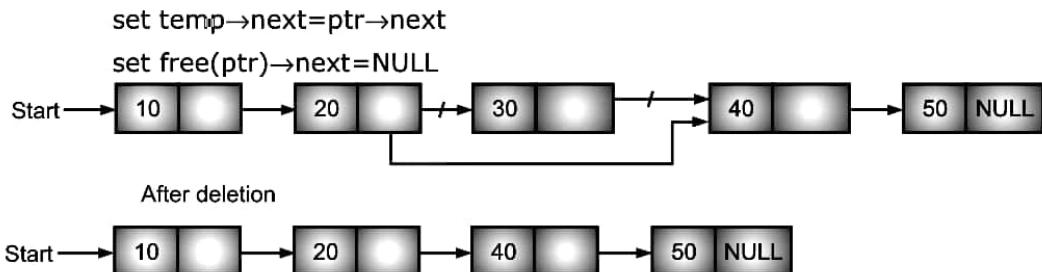


Fig. 3.13: Delete Specified Node

- 'C' code for above algorithm:

```
Void Deleteloc(NODE * start)
{
    node *P, *temp;
    int i, pos;
    printf("enter the position to delete\n");
    scanf("%d",&pos);
    if(start==NULL)
    {
        printf("List isEmpty");
    }
    else
    {
        Ptr=start;
        for(i=1;i<=pos;i++)
        {temp=Ptr;
        Ptr=Ptr→next;
        }
        printf("Deleted no is=%d",Ptr→info);
        temp→next=Ptr→next; ptr→next=NULL;
        free(Ptr);
    }
}
```



3.3.1.3 Advantages

- Advantages of singly linked list are given below:
 1. Accessibility of a node in the forward direction is easier.
 2. Insertion and deletion of nodes are easier.

3.3.1.4 Disadvantages

- Disadvantages of singly linked list are given below:
 1. Accessing the preceding node of a current node is not possible as there is no backward traversal.
 2. Accessing a node is time consuming.

3.3.2 Doubly Linked List (DLL)

[Oct. 15, April 16, 17]

- A doubly linked list may either be linear or circular and it may or may not contain a header node.
- Doubly linked list is also called as **two-way lists**.
- In singly linked list, each node provides information about where the next node is. It has no knowledge about where the previous node is.
- For example, if currently we are at i^{th} node in the list then to access the $(i-1)^{\text{th}}$ node or $(i-2)^{\text{th}}$ node we have to traverse the list right from the first node.
- Also it is not possible to delete i^{th} node given only pointer to i^{th} node. It is also not possible to insert a node before i^{th} node given only pointer to i^{th} node, (There are other ways which are without link manipulations such as using data exchange).
- For handling such difficulties, we can use doubly linked lists in which each node contain two links, one to its predecessor and other to its successor.



Fig. 3.14 (a): Doubly Linked List

- Each node of a doubly linked list has generally three fields from which two are link fields.

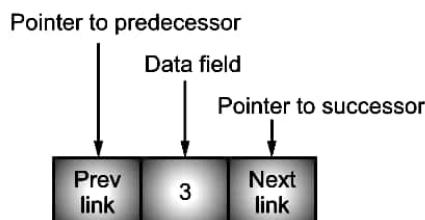


Fig. 3.14 (b): Fields of Doubly Linked List



- We can declare such a node in C as,

```
typedef struct dll_node
{
    int data;
    struct dll_node *prev, *next;
} DLLNODE;
```

3.3.2.1 Creation Of Doubly Linked List

- Creation of DLL has same procedure as that of singly linked list.
 - The only difference is each node must be linked to its predecessor and successor.
- For creating a DLL is as follows:

```
DLLNODE * create()
{
    char ans;
    DLLNODE * temp, * head, newnode;
    head = NULL;
    do
    {
        newnode=(DLLNODE *) malloc(sizeof(DLLNODE));
        printf("\n Enter the value:");
        scanf("%d", & newnode->data);
        newnode->next=newnode->prev=NULL;
        if(head==NULL)
        {
            head=newnode;
            temp=head;
        }
        else
        {
            temp->next=newnode;
            newnode->prev=temp;
            temp=temp->next;
        }
        printf("\m Any node to be added? (Y/N) :");
        scanf("%c", & ans);
    }while(ans=='N' || ans=='n');
    return(head);
}
```



3.3.2.2 Deletion of Node from DLL

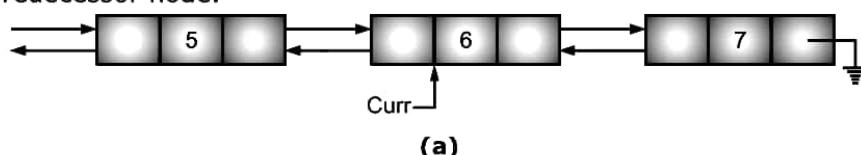
- We studied that to delete node in linked list, we have to follow the two steps given below:
 - Rearranging the pointers prev and next.

temp

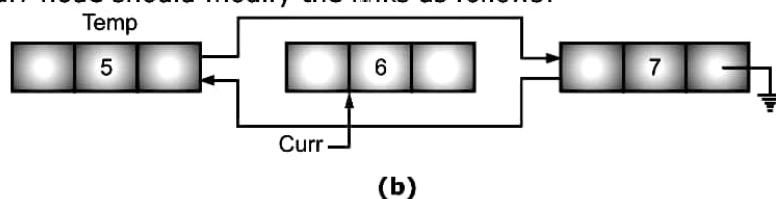
- Release the memory allocated

```
free (curr);
```

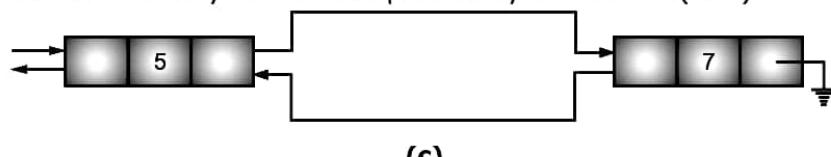
- Here, curr is a pointer to node to be deleted and temp is a pointer to node to the previous node.
- Deleting from a DLL needs the deleted node's predecessor, if any, to be pointed to the deleted nodes successor. Also the successor, if any, should be set to point the predecessor node.



delete *curr* node should modify the links as follows:



- Let us free the memory of the node pointed by curr as free(curr).



- The above figure can be redrawn as,

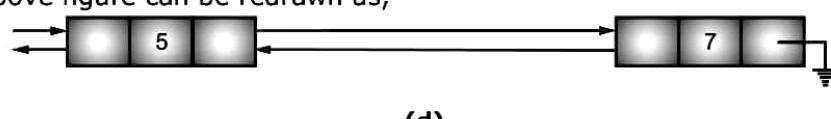


Fig. 3.15: Deletion of Node in DLL

- Deletion of a node can be done in two ways:
 - deletion by value (many nodes can have same value so in this case zero or more nodes can be deleted).
 - deletion by a position (in this case zero or one node can be deleted).



- Function to delete a node by value is as follows:

```
DLLNODE * delete_by_value (DLLNODE * head, int value)
{
    DLLNODE * temp, * templ;
    if(head==NULL)
    {
        printf("\n List is empty.");
        return(head);
    }
    else
    temp=head;
    while(temp!=NULL)
    {
        if(temp->data==value)
        {
            if(temp==head) //first node contains value to be deleted
            {
                head=head->next;
                head->prev=NULL;
                free(temp);
                temp=head;
            }
            else
//if in between node or last node contains value to be deleted
            {
                templ->next=temp->next;
                if(temp->next!=NULL)
                    temp->next->prev=templ;
                free(temp);
                temp=templ->next;
            }
        }
        else
        {
            templ=temp;
            temp=temp->next;
        }
    }
    return(head);
}
```



- Function to delete a node by position is as follows:

```
DLLNODE * delete_by_pos (DLLNODE * head, int pos)
{
    DLLNODE * temp = head;
    int cnt=1;
    if(head==NULL)
    {
        printf("\n List is empty.");
        return(head);
    }
    else
    {
        if(pos==1) //deletion of first node
        {
            head=head->next;
            head->prev=NULL;
            free(temp);
            return(head);
        }
        while(count!=pos&&temp!=NULL)
            //deletion of inbetween or last node
        {
            temp=temp->next;
            count++;
        }
        if(temp==NULL)
        {
            printf("Wrong position");
            return(head);
        }
        else
        {
            (temp->prev)->next=temp->next
            if(temp->next!=NULL) //if temp is not a last node
                (temp->next)->prev=temp->prev
            free(temp);
            return(head);
        }
    }
}
```



3.3.2.3 Insertion of Node in DLL

(a) Insertion at first place:

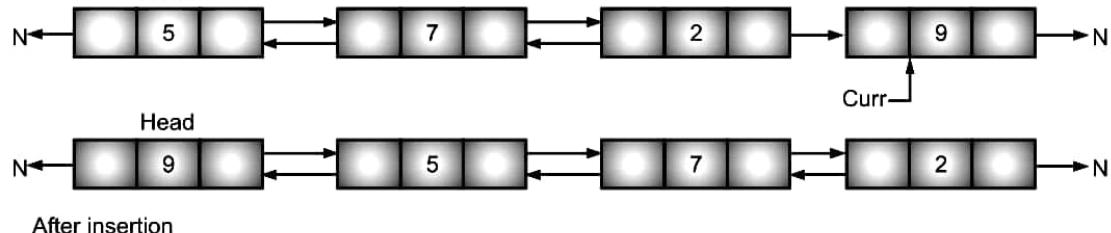
- To insert at first position we need to modify 3 pointers (suppose node to be inserted is pointed by 'curr').

```
curr→next=head;
```

```
head→prev=curr;
```

```
head=curr;
```

Head



After insertion

Fig. 3.16: Insert Node at Beginning

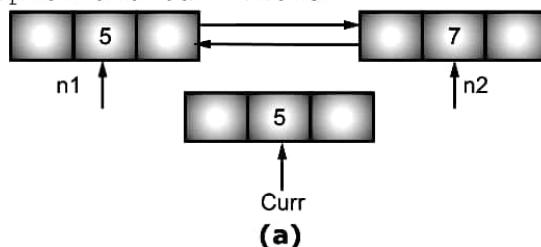
(b) Insertion in between two nodes:

- To insert a node, say 'curr', we have to modify four links. If the node 'curr' is to be inserted in between the two nodes say n1 and n2, we have to modify following links.

$n1 \rightarrow next, n2 \rightarrow prev, curr \rightarrow prev$ and $curr \rightarrow next$

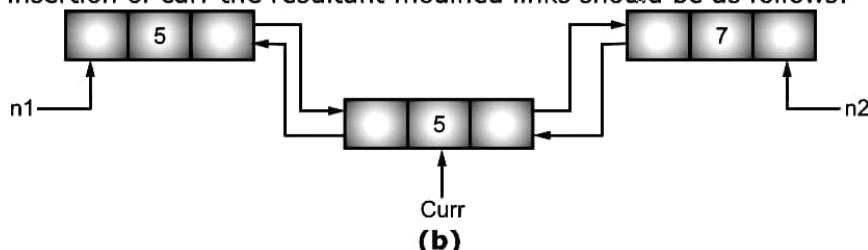
- When the 'curr' node is inserted in between n1 and n2, n1's successor node changes. Hence we need to modify $n1 \rightarrow next$. For node n2, its predecessor changes, hence we need to modify $n2 \rightarrow prev$. The 'curr' is new node to be inserted we need to set its both predecessor and successor by setting the links:

$curr \rightarrow prev$ and $curr \rightarrow next$



Curr
(a)

After insertion of curr the resultant modified links should be as follows:



Curr
(b)

Fig. 3.17: Insert Node in Between



- Hence, to modify the links the statements would be,

(I) To modify $n1 \rightarrow \text{next}$ we can write as,

$n1 \rightarrow \text{next} = \text{curr};$

(II) To modify $n2 \rightarrow \text{prev}$ we can write,

$n2 \rightarrow \text{prev} = \text{curr};$

(III) To set $\text{curr} \rightarrow \text{next}$, we can write,

$\text{curr} \rightarrow \text{next} = n2;$

(IV) To set $\text{curr} \rightarrow \text{prev}$, we can write,

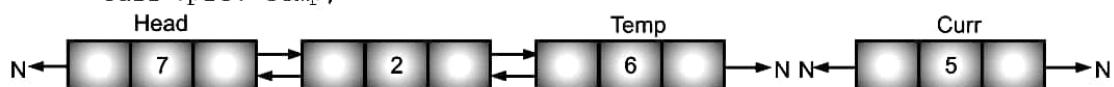
$\text{curr} \rightarrow \text{prev} = n1;$

(c) Insertion at end:

- To insert at end, first take a dummy pointer say 'temp' and move it to point last node.
- Then modify 2 pointers as follows:

$\text{temp} \rightarrow \text{next} = \text{curr};$

$\text{curr} \rightarrow \text{prev} = \text{temp};$



After insertion

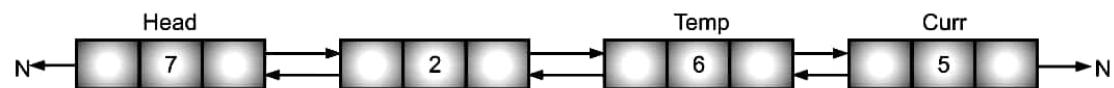


Fig. 3.18: Insert Node at end

- Function for inserting a node into list is:

```
DLLNODE * insert(DLLNODE * head, int value, int pos)
{
    int cnt=1;
    DLLNODE * curr, *temp;
    temp=head;
    if(head==NULL)
    {
        printf("List is empty.");
        return(head);
    }
    else
    {
        curr=(DLLNODE *) malloc(sizeof(DLLNODE));
        curr->data=value;
        curr->prev=NULL;
        curr->next=NULL;
    }
}
```



```
if(pos==1)    //insertion at beginning
{
    curr→next=head;
    head→prev=curr;
    head=curr;
    return(head);
}
else
{
    while(temp!=NULL && cnt < pos)
    {
        temp=temp→next;
        cnt++;
    }
    if(temp==NULL)
    {
        printf("\n Wrong position");
        return(head);
    }
    else
    {
        curr→next=temp→next;
        curr→prev→temp;
        if(temp→next!=NULL) //if temp is not a last node
            (temp→next)→prev=curr;
        temp→next=curr;
        return(head);
    }
}
}
```

3.3.2.4 Traversal of DLL

- As doubly linked list node has 2 pointers, list can be displayed in two way:
 - in forward direction (first to last)
 - in backward direction (last to first)

Display DLL in forward direction:

- To traverse or display DLL in forward direction, we use next pointer.
- First assign head node pointer to a dummy pointer say 'temp'.
- Then print the value of each node by moving temp as `temp=temp→next` till it is not NULL.



- Function to traverse DLL in forward direction is as follows:

```
void display_forward(DLLNODE * head)
{
    DLLNODE *temp;
    temp=head;
    if(head==NULL)
        printf("\n List is empty.");
    else
        printf("\n DLL is: \n");
    {
        while(temp!=NULL)
        {
            printf("%d \t", temp→data);
            temp=temp→next;
        }
    }
}
```

Display DLL is backward direction:

- To traverse or display DLL in backward direction, we use prev pointer.
- First set a dummy pointer, say 'temp' to a head node.
- Move temp to the end of list so that it points to the last node.
- Then use prev pointer as `temp=temp→prev` and print each node value, till temp is not NULL.
- Function to traverse DLL in backward direction is as follows:

```
void display_backward (DLLNODE *head)
{
    DLLNODE *temp=head;
    if(head==NULL)
        print("\n List is empty.");
    else
        printf("DLL is reverse order is:");
    {
        while(temp→next!=NULL) //temp is moved till last node
            tempt=temp→next;
        while(temp!=NULL)
        {
            printf("%d \t", temp→data);
            temp=temp→prev;
        }
    }
}
```



3.3.2.5 Advantages of DLL

1. It is possible to delete i^{th} node given only pointer to i^{th} node.
2. It is also possible to insert at i^{th} place given only pointer to $(i - 1)^{\text{th}}$ or $(i + 1)^{\text{th}}$ node.
3. List can be traversed in both directions.

3.3.2.6 Disadvantages of DLL

1. Extra storage is required to store both pointers.
2. Two pointers to be handled at the time of insertion in or deletion from doubly linked list.

Program 3.2: Doubly linked list.

```
#include<stdio.h>
#include<malloc.h>
typedef struct node
{
    int data;
    struct node *prev;
    struct node * next;
}DLLNODE;
DLLNODE * create(DLLNODE *);
void display_forward(DLLNODE *);
void display_backward(DLLNODE *);
DLLNODE * insert_pos(DLLNODE *,int,int);
DLLNODE * deletenode_pos(DLLNODE *,int);
void main()
{
    DLLNODE *head=NULL;
    int choice,n,no,i,pos;
    clrscr();
    do
    {
        printf("\n1. Create a list \n2. Insert at given position \n3.
Delete a node \n4. Display in forward direction \n5. Display
in backward direction \n6. Exit");
        printf("\nEnter ur choice : ");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1: head=NULL;
                      head=create(head);
                      break;
```



```
case 2: printf("\n Enter the value to insert :");
          scanf("%d",&no);
          printf("\n Enter the position :");
          scanf("%d",&pos);
          if(head==NULL)
              printf("List is empty. Create it first.");
          else
              head=insert_pos(head,no,pos);
          break;
case 3: printf
          ("\nEnter the position of the node
                           to be deleted : ");
          scanf("%d",&pos);
          if(head==NULL)
              printf("Linked list is empty.");
          else
              head=deletenode_pos(head,pos);
          break;
case 4: display_forward(head);
          break;
case 5: display_backward(head);
          break;
case 6: break;
}
}
while(choice != 6);
}
DLLNODE * create(DLLNODE *head)
{
    int no;
    char ch;
    DLLNODE *newnode,*temp;
    do
    {
        printf("\n Enter the value to insert :");
        scanf("%d",&no);
        newnode=(DLLNODE *) malloc(sizeof(DLLNODE));
        if(newnode==NULL)
        {
            printf("\nNew node cannot be created.");
            exit(0);
        }
    }
```



```
else
{
    newnode->data=no;
    newnode->next=NULL;
    newnode->prev=NULL;
    if(head==NULL) // first node
    {
        head=newnode;
        temp=head; // temp points to last node
    }
    else // append node, same as insertion at the end
    {
        temp->next=newnode;
        newnode->prev=temp;
        temp=newnode; //newnode will be now last node
    }
}
printf("\nDo u want to enter another value?(Y/N) :");
scanf(" %c",&ch);
}while(ch=='Y' || ch=='y');
return head;
}

DLLNODE * insert_pos(DLLNODE * head, int no, int pos)
{
    DLLNODE *newnode,*temp;
    int i=1;
    newnode=(DLLNODE *)malloc(sizeof(DLLNODE));
    if(newnode==NULL)
    {
        printf("\nNew node cannot be created.");
        exit(0);
    }
    else
    {
        newnode->data=no;
        newnode->next=newnode->prev=NULL;
        if(pos==1) // insert at first place
        {
            newnode->next=head;
            head->prev=newnode;
            head=newnode;
        }
    }
}
```



```
else
{
    temp=head;
    for(i=1;i<pos-1 && temp!=NULL;i++)
    {
        temp=temp->next;
    }
    if(temp==NULL)
    {
        printf("\nWrong position.");
    }
    else
    {
        newnode->next=temp->next;
        newnode->prev=temp;
        if(temp->next!=NULL)// if temp is not a last node
            (temp->next)->prev=newnode;
        temp->next=newnode;
    }
}
return(head);
}

DLLNODE * deletenode_pos(DLLNODE *head,int pos)
{
    DLLNODE *temp;
    int cnt=1;
    temp=head;
    if(pos == 1) // deletion of first node
    {
        head=head->next;
        head->prev=NULL;
        free(temp);
    }
    else
    {
        while(cnt<pos && temp!=NULL)
        {
            temp=temp->next;
            cnt++;
        }
    }
}
```



```
if(temp==NULL)
    printf("\nWrong position.");
else
{
    (temp->prev)->next=temp->next;
    if(temp->next!=NULL) // temp is not a last node
        (temp->next)->prev=temp->prev;
    free(temp);
}
return(head);
}

void display_forward(DLLNODE *head)
{
    DLLNODE *temp;
    if(head==NULL)
        printf("Linked list is empty.");
    else
    {
        printf("\n Linked list is : ");
        temp=head;
        while(temp!=NULL)
        {
            printf("%d ",temp->data);
            temp=temp->next;
        }
    }
}

void display_backward(DLLNODE *head)
{
    DLLNODE *temp;
    if(head==NULL)
        printf("Linked list is empty.");
    else
    {
        printf("\n Linked list in reverse order is : ");
        while(temp->next!=NULL) // move temp till last node
            temp=temp->next;
        while(temp!=NULL)
        {
            printf("%d ",temp->data);
            temp=temp->prev;
        }
    }
}
```

**Output:**

```
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 1
Enter the value to insert :56
Do u want to enter another value?(Y/N) :y
Enter the value to insert :23
Do u want to enter another value?(Y/N) :y
Enter the value to insert :8
Do u want to enter another value?(Y/N) :y
Enter the value to insert :11
Do u want to enter another value?(Y/N) :n
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 4
Linked list is : 56 23 8 11
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 5
Linked list in reverse order is : 11 8 23 56
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 2
Enter the value to insert :67
Enter the position :1
```



```
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 4

Linked list is : 67 56 23 8 11
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 3
Enter the position of the node to be deleted : 3
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 4

Linked list is : 67 56 8 11
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 6
```

3.3.3 CIRCULAR SINGLY LIST

[April 16]

- The linked lists that we have seen so far are often known as linear linked lists.
- All elements of such a linked list can be accessed by first setting up a pointer pointing to the first node in the list and then traversing the entire list.
- Although a linear linked list is a useful data structure, it has some drawbacks. For example, given a pointer 'curr' to a node in a linear list, we cannot reach any of the nodes that precede the 'curr' node. This drawback can be overcome by making a small change. This change is without any additional data structure.



In linear list, the last nodes link field is set to NULL. Instead of that, store the address of first node of the list in that link field. This will make the last node point to first node of the list. Such a linked list is called as circular linked list.

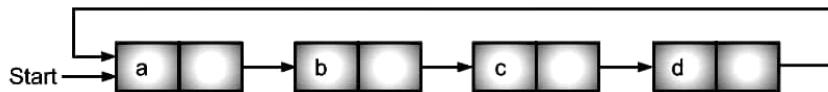


Fig. 3.19: Circular Singly List

- Node structure is same as that of linear singly linked list.
- From any node in such a list, it is possible to reach to any other node in the list. We need not traverse the list again right from the first node.
- Circular linked list is used in many applications:
 1. Circular linked list is used to keep track of free space (unused nodes) in memory. In circular list, traversal can be continued from current node, it helps us to keep the traversal procedure an unending one.
 2. Two of the applications circular list is time slicing and memory management.
- Circular linked lists could be with or without head nodes. Circular lists could be either singly or doubly linked list.

3.3.3.1 Singly Circular Linked List (Circular SLL)

- Let us consider singly linked list without head node as shown below.

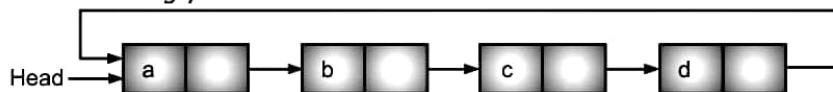


Fig. 3.20 (a)

- In circular singly list, the pointer head points first node of the list. From the last node, we can access the first node.
- We can set pointer head to point to last node instead of first node? It looks pictorially as follows:

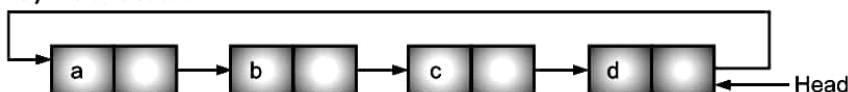


Fig. 3.20 (b)

- Now through head we have access to last node and also ($\text{head} \rightarrow \text{next}$) gives us address of the first node. Remaining behaviour of the list remains same as that of linear singly linked list.

• Inserting Nodes in Circular Singly Linked List:

1. Algorithm for Inserting a node at the Beginning:

Insertfirst(head,item)

1. $\text{Ptr} = (\text{node} *) \text{ malloc } (\text{sizeof} (\text{node}))$
//create new node from memory and assign its address to ptr
 $\text{set } \text{Ptr} \rightarrow \text{num} = \text{item}$



2. if head==NULL then
 set Ptr→next=Ptr
 set head=Ptr
 set last=Ptr
 end if
3. otherwise
 set Ptr→next=head
 set head=Ptr
 set last→next=ptr

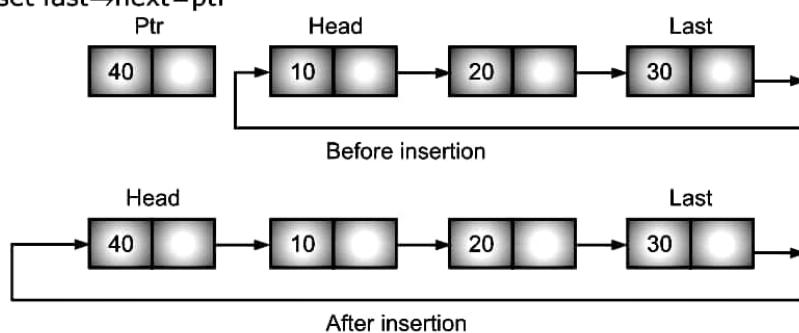


Fig. 3.21: Insert Node at First

- **'C' code for above algorithm:**

```
NODE *insertfirst(NODE * head, int item)
{
    NODE *P, last;
    P=(NODE *) malloc (sizeof(NODE));
    p→info=item;
    if(head==NULL)
    {
        P→next=P;
        head=P;
        last=P;
    }
    else
    {
        last=head;
        while(last→next!=head)
            last=last→next;
        P→next=head;
        head=P;
        last→next=P;
    }
    return(head);
}
```



2. Algorithm for Inserting a new node at the End:

insertlast (head, item)

1. Ptr=(NODE *) malloc (sizeof (NODE));
set Ptr→num=item
2. if head=NULL, then
set Ptr→next=Ptr
set last=Ptr
set start=Ptr
end if
3. otherwise
4. set last→next=Ptr
5. set last=Ptr
6. set last→next=head

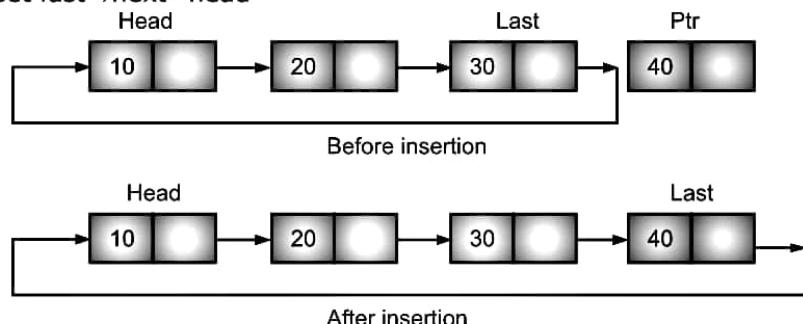


Fig. 3.22: Insert Node at the end

- **'C' code for above algorithm:**

```
NODE *insertfirst(NODE *head, int item)
{
    NODE *P, * last;
    P=(NODE *) malloc (size of(NODE));
    P→info=item;
    if(head==NULL)
    {
        P→next=P;
        head=last=P;
    }
    else
    { last=head;
        while(last→next!=head)
            last=last→next;
        last→next=P;
        last=P;
        last→next=head;
    }
    return(head);
}
```



3. Algorithm for inserting a new node in between (at a given position):

- Algorithm is same as for linear singly linked list. Only difference is to check end of list us condition ($\text{temp} \rightarrow \text{next} \neq \text{head}$).

Deleting a Node:

1. Algorithm for Deleting a node from the Beginning:

```
Deletefirst(head)
1. [check for Underflow]
   if start=NULL then print 'List is Empty'
   exit
endif
2. set Ptr=head
3. set head=head→next
4. print 'Element deleted is Ptr→info'
5. set last→next=head
6. free(Ptr)
```

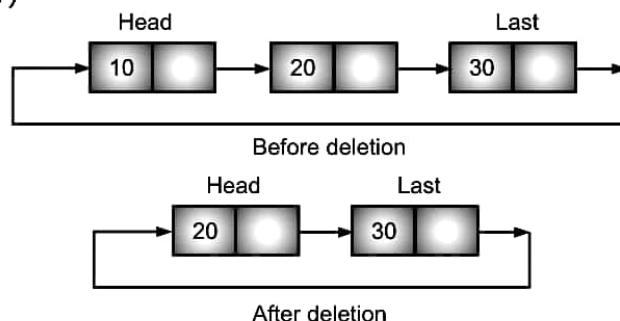


Fig. 3.23: Delete Node from Beginning

- 'C' code for above algorithm:

```
Node * deletefirst (node *head)
{
Node *P;
P=head;
if (P==NULL)
{ printf("list empty"); }
else
{
head=head→next;
printf("element deleted is %d", P→info);
last→next=P;
}
Return head;
}
```



2. Algorithm for Deleting a node from the End:

```

deletelast(head)
1. [check for underflow]
   If head=NULL then print 'list Empty'
   exit
endif
2. set Ptr=head
3. repeat steps 4 and 5 until
   Ptr->next!=head
4. set Ptr1=Ptr
5. set Ptr=Ptr->next
6. print "element deleted is", Ptr->info
7. set ptr1->next=ptr->next
8. set last=ptr1
9. free(ptr)
10. return head

```

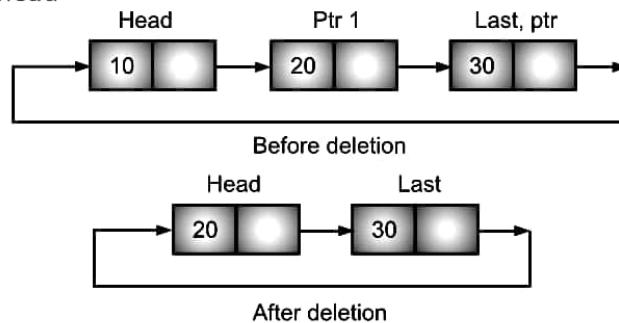


Fig. 3.24: Delete Node from End

- **'C' code for above algorithm:**

```

void deletelast(NODE * head)
{
NODE *P, *q;
P=head;
If (P==NULL)
{ printf("list empty");
}

```



```
else
{
    While(p!=last) //or while(p→next!=head)
    {
        Q=P;
        P=P→next;
    }
    printf("element deleted is=%d",P→info);
    q→next=P→next;
    free(p);
}
Return head;
```

Program 3.3: Circular Singly Linked List.

[Oct. 17]

```
#include<stdio.h>
#include<malloc.h>
typedef struct node
{
    int data;
    struct node * next;
}LNODE;
LNODE * create(LNODE *); // to create a list
void display(LNODE *); // to display list
void display_pos(LNODE *,int); // to traverse list from given position
LNODE * insert_pos(LNODE *,int,int); // to insert a value at given position
LNODE * deletenode_pos(LNODE *, int); // to delete a node by position
void main()
{
    LNODE *head=NULL; // pointer to store address of first node
    int choice,n,no,i,pos;
    clrscr();
    do
    {
        printf("\n1. Create a list \n2. Display \n3. Traverse list
from given position \n4. Insert at given position \n5. Delete
a node by position \n6. Exit");
        printf("\nEnter ur choice : ");
        scanf("%d",&choice);
```



```
switch(choice)
{
    case 1: head=NULL;
              head=create(head);
              break;
    case 2: display(head);
              break;
    case 3: printf("\n Enter the position from
                           where to traverse :");
              scanf("%d",&pos);
              display_pos(head,pos);
              break;
    case 4: printf("\n Enter the value to insert :");
              scanf("%d",&no);
              printf("\n Enter the position :");
              scanf("%d",&pos);
              if(head==NULL)
                  printf("List is empty. Create it first.");
              else
                  head=insert_pos(head,no,pos);
              break;
    case 5: printf("\nEnter the node position to delete : ");
              scanf("%d",&pos);
              if(head==NULL)
                  printf("Linked list is empty.");
              else
                  head=deletenode_pos(head,pos);
              break;
    case 6: break;
}
}
while(choice != 6);
}

LNODE * create(LNODE *head)
{
    int no;
    char ch;
    LNODE *newnode,*temp;
    do
    {
        printf("\n Enter the value to insert :");
        scanf("%d",&no);
        newnode=(LNODE *) malloc(sizeof(LNODE));
        newnode->data=no;
        newnode->next=NULL;
        if(head==NULL)
            head=newnode;
        else
        {
```



```
if(newnode==NULL)
{
    printf("\nNew node cannot be created.");
    exit(0);
}
else
{
    newnode->data=no;
    newnode->next=NULL;
    if(head==NULL) // first node
    {
        head=newnode;
        newnode->next=head;
        temp=head; // temp points to last node
    }
    else // append node, same as insertion at the end
    {
        temp->next=newnode;
        newnode->next=head;
        temp=newnode; //newnode will be now last node
    }
}
printf("\nDo u want to enter another value?(Y/N) :");
scanf(" %c",&ch);
}while(ch=='Y' || ch=='y');
return head;
}
void display(LNODE *head)
{
    LNODE *temp;
    if(head==NULL)
    printf("Linked list is empty.");
    else
    {
        printf("\n Linked list is : ");
        temp=head;
        do
        {
            printf("%d->",temp->data);
            temp=temp->next;
        }while(temp!=head);
    }
}
```



```
void display_pos(LNODE *head,int pos)
{
    int cnt=1;
    LNODE *temp,*temp1;
    if(head==NULL)
        printf("Linked list is empty.");
    else
    {
        temp=head;
        while(cnt<pos)
        {
            temp=temp->next;
            if(temp==head)
                break;
            cnt++;
        }
        if(temp==head && cnt < pos)
        {
            printf("\nWrong position.");
        }
        else
        {
            printf("\n Linked list is from given position: ");
            temp1=temp;
            do
            {
                printf("%d->",temp1->data);
                temp1=temp1->next;
            }while(temp1!=temp);
        }
    }
}

LNODE * insert_pos(LNODE * head, int no, int pos)
                                //insert at given position
{
    LNODE *newnode,*temp1,*last;
    int i=1,flag=0;
    newnode=(LNODE *)malloc(sizeof(LNODE));
    newnode->data=no;
    newnode->next=NULL;
    last=head;
    while(last->next!=head)
        last=last->next;
```



```
if(pos==1) // insert at first place
{
    newnode->next=head;
    head=newnode;
    last->next=head;
    return(head);
}
else
{
    templ=head;
    i=1;
    while(i<pos-1)
    {
        templ=templ->next;
        i++;
        if(templ==head)
        {
            flag=1;
            break;
        }
    }
    if(flag==1)
        printf("\nWrong position.");
    else
    {
        newnode->next=templ->next;
        templ->next=newnode;
    }
}
return(head);
}

LNODE * deletenode_pos(LNODE *head,int pos)
{
    LNODE *temp1,*temp2,*last;
    int cnt=2;
    last=head;
    while(last->next!=head)
    last=last->next;
    temp1=head;
    if(pos==1) //delete head node
    {
        head=head->next;
        last->next=head;
        free(temp1);
    }
}
```



```
else
{
    temp2=head;
    temp1=head->next;
    while(cnt<pos && temp1!=head)
    {
        temp2=temp1;
        temp1=temp1->next;
        cnt++;
    }
    if(temp1==head)
        printf("\nWrong position.");
    else
    {
        temp2->next=temp1->next;
        free(temp1);
    }
}
return(head);
}
```

Output:

```
1. Create a list
2. Display
3. Traverse list from given position
4. Insert at given position
5. Delete a node by position
6. Exit
Enter ur choice : 1
Enter the value to insert :56
Do u want to enter another value?(Y/N) :y
Enter the value to insert :32
Do u want to enter another value?(Y/N) :y
Enter the value to insert :90
Do u want to enter another value?(Y/N) :y
Enter the value to insert :2
Do u want to enter another value?(Y/N) :n
1. Create a list
2. Display
3. Traverse list from given position
4. Insert at given position
5. Delete a node by position
6. Exit
Enter ur choice : 2
```



```
Linked list is : 56→32→90→2→
1. Create a list
2. Display
3. Traverse list from given position
4. Insert at given position
5. Delete a node by position
6. Exit
Enter ur choice : 3
Enter the position from where to traverse :3

Linked list is from given position: 90→2→56→32→
1. Create a list
2. Display
3. Traverse list from given position
4. Insert at given position
5. Delete a node by position
6. Exit
Enter ur choice : 4
Enter the value to insert :34
Enter the position :3
1. Create a list
2. Display
3. Traverse list from given position
4. Insert at given position
5. Delete a node by position
6. Exit
Enter ur choice : 2

Linked list is : 56→32→34→90→2→
1. Create a list
2. Display
3. Traverse list from given position
4. Insert at given position
5. Delete a node by position
6. Exit
Enter ur choice : 5
Enter the node position to delete : 4
1. Create a list
2. Display
3. Traverse list from given position
4. Insert at given position
5. Delete a node by position
6. Exit
Enter ur choice : 2
```



```

Linked list is : 56→32→34→2→
1. Create a list
2. Display
3. Traverse list from given position
4. Insert at given position
5. Delete a node by position
6. Exit
Enter ur choice : 6

```

3.3.4 Circular Doubly Linked List

[April 15, 17, Oct. 17]

- In doubly circular linked list last node's next link is set to the first node of the list and the first nodes previous link is set to the last node of the list. This gives access to the last node directly from first node.

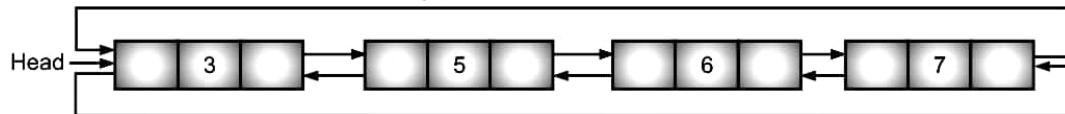


Fig. 3.25: Circular Doubly Linked List

- The operations on circular linked list insert, delete node and also create and traverse do follow the same method as that of linear doubly list except few changes.

Program 3.4: Circular Doubly linked list

```

#include<stdio.h>
#include<malloc.h>
typedef struct node
{
    int data;
    struct node *prev;
    struct node * next;
}DLLNODE;
DLLNODE * create(DLLNODE *);
void display_forward(DLLNODE *);
void display_backward(DLLNODE *);
DLLNODE * insert_pos(DLLNODE *,int,int);
DLLNODE * deletenode_pos(DLLNODE *,int);
void main()
{
    DLLNODE *head=NULL;
    int choice,n,no,i,pos;
    clrscr();
}

```



```
do
{ clrscr();
printf("\n1. Create a list \n2. Insert at given position \n3.
Delete a node \n4. Display in forward direction \n5. Display
in backward direction \n6. Exit");
printf("\nEnter ur choice : ");
scanf("%d",&choice);
switch(choice)
{
case 1: head=NULL;
head=create(head);
break;
case 2: printf("\nEnter the value to insert :");
scanf("%d",&no);
printf("\nEnter the position :");
scanf("%d",&pos);
if(head==NULL)
printf("List is empty. Create it first.");
else
head=insert_pos(head,no,pos);
break;
case 3: printf("\nEnter the position of the
node to be deleted : ");
scanf("%d",&pos);
if(head==NULL)
printf("Linked list is empty.");
else
head=deletenode_pos(head,pos);
break;
case 4: display_forward(head);
break;
case 5: display_backward(head);
break;
case 6: break;
}
getch();
}
while(choice != 6);
}

DLLNODE * create(DLLNODE *head)
{
int no;
char ch;
DLLNODE *newnode,*temp;
```



```
do
{
    printf("\n Enter the value to insert :");
    scanf("%d", &no);
    newnode=(DLLNODE *) malloc(sizeof(DLLNODE));
    if(newnode==NULL)
    {
        printf("\nNew node cannot be created.");
        exit(0);
    }
    else
    {
        newnode->data=no;
        newnode->next=NULL;
        newnode->prev=NULL;
        if(head==NULL) // first node
        {
            head=newnode;
            head->next=head;
            head->prev=head;
            temp=head; // temp points to last node
        }
        else // append node, same as insertion at the end
        {
            temp->next=newnode;
            newnode->prev=temp;
            newnode->next=head;
            head->prev=newnode;
            temp=newnode; //newnode will be now last node
        }
    }
    printf("\nDo u want to enter another value?(Y/N) :");
    scanf(" %c", &ch);
}while(ch=='Y' || ch=='y');
return head;
}
DLLNODE * insert_pos(DLLNODE * head, int no, int pos)
{
    DLLNODE *newnode,*temp,*last;
    int i=1,flag=0;
    newnode=(DLLNODE *)malloc(sizeof(DLLNODE));
```



```
if (newnode==NULL)
{
    printf("\nNew node cannot be created.");
    exit(0);
}
else
{
    newnode->data=no;
    newnode->next=newnode->prev=NULL;
    last=head;
    while(last->next!=head)
        last=last->next;
    if(pos==1) // insert at first place
    {
        newnode->next=head;
        head->prev=newnode;
        head=newnode; //newnode becomes head node
        last->next=head;
        head->prev=last;
    }
    else
    {
        temp=head;
        i=1;
        while(i<pos-1)
        {
            temp=temp->next;
            i++;
            if(temp==head)
            {
                flag=1;
                break;
            }
        }
        if(flag==1)
            printf("\nWrong position.");
        else
        {
            newnode->next=temp->next;
            newnode->prev=temp;
            (temp->next)->prev=newnode;
            temp->next=newnode;
        }
    }
}
return(head);
}
```



```
DLLNODE * deletenode_pos(DLLNODE *head,int pos)
{
    DLLNODE *temp,*last;
    int cnt;
    temp=head;
    last=head;
    while(last->next!=head)
        last=last->next;
    if(pos == 1) // deletion of first node
    {
        head=head->next;
        head->prev=last;
        last->next=head;
        free(temp);
    }
    else
    {
        cnt=2;
        temp=head->next;
        while(cnt<pos && temp!=head)
        {
            temp=temp->next;
            cnt++;
        }
        if(temp==head)
            printf("\nWrong position.");
        else
        {
            (temp->prev)->next=temp->next;
            (temp->next)->prev=temp->prev;
            free(temp);
        }
    }
    return(head);
}
void display_forward(DLLNODE *head)
{
    DLLNODE *temp;
    if(head==NULL)
        printf("Linked list is empty.");
}
```



```
else
{
    printf("\n Linked list is : ");
    temp=head;
    do
    {
        printf("%d ",temp->data);
        temp=temp->next;
    }while(temp!=head);
}
void display_backward(DLLNODE *head)
{
    DLLNODE *last,*temp;
    if(head==NULL)
        printf("Linked list is empty.");
    else
    {
        printf("\n Linked list in reverse order is : ");
        last=head;
        while(last->next!=head) // move last till last node
        last=last->next;
        temp=last;
        do
        {
            printf("%d ",temp->data);
            temp=temp->prev;
        }while(temp!=last);
    }
}
```

Output:

```
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 1
Enter the value to insert :56
Do u want to enter another value?(Y/N) :y
```



```
Enter the value to insert :32
Do u want to enter another value?(Y/N) :y
Enter the value to insert :9
Do u want to enter another value?(Y/N) :y
Enter the value to insert :1
Do u want to enter another value?(Y/N) :n
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 4

Linked list is : 56 32 9 1
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 5

Linked list in reverse order is : 1 9 32 56
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 2
Enter the value to insert :45
Enter the position :2
1. Create a list
2. Insert at given position
3. Delete a node
```



```
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 4
Linked list is : 56 45 32 9 1
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 3
Enter the position of the node to be deleted : 2
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 4

Linked list is : 56 32 9 1
1. Create a list
2. Insert at given position
3. Delete a node
4. Display in forward direction
5. Display in backward direction
6. Exit
Enter ur choice : 6
```

Practice Questions

1. What is Linked list?
2. Define Linked list. Explain its types.
3. Differentiate between Singly and Doubly linked list.
4. Explain insertion and deletion operation with the circular list.
5. Explain the use of pointer head in the linked list.



6. Explain applications of linked list.
7. What is circular list?
8. Write algorithm to insert and delete node of doubly linked list.
9. State advantages of linked lists.
10. State disadvantages of linked list.
11. Enlist various operations of linked list.
12. How to create a node in singly circular list?
13. How to delete node in doubly circular list.





Chapter 4...

Stack and Queue

Contents ...

- 4.1 Introduction
- 4.2 Static and Dynamic Representation of Stack
- 4.3 Primitive Operations on Stack
- 4.4 Applications of Stack
- 4.5 Evaluation of Postfix and Prefix Expression
- 4.6 Conversion of Expressions- Infix to Prefix and Infix to Postfix
- 4.7 Introduction to Queue
- 4.8 Primitive Operations on Queue
- 4.9 Static and Dynamic Representation of Queue
- 4.10 Application of Queue
- 4.11 Type of Queue
 - 4.11.1 Priority Queue
 - 4.11.2 Circular Queue
 - 4.11.3 Double Ended Queue
- 4.12 Difference between Stack and Queue
 - Practice Questions
 - University Questions and Answers

4.1 INTRODUCTION

- A stack is an ordered collection of items in which insertion and deletions are allowed only at one end called top of the stack.
- It is a linear list which is dynamic, constantly changing object.
- The definition specifies that a single end of stack is designated as the stack top. New item may be put on top of stack or item which is at the top of the stack may be removed.

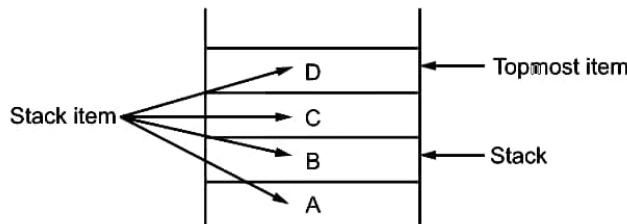


Fig. 4.1: Stack containing stack items
(4.1)



- A stack is a non-primitive linear data structure.
- As all the deletion and insertion in a stack is done from top of the stack, the last added element will be the first to be removed from the stack. That is the reason why stack is also called **Last-In-First-Out (LIFO)** type of list.

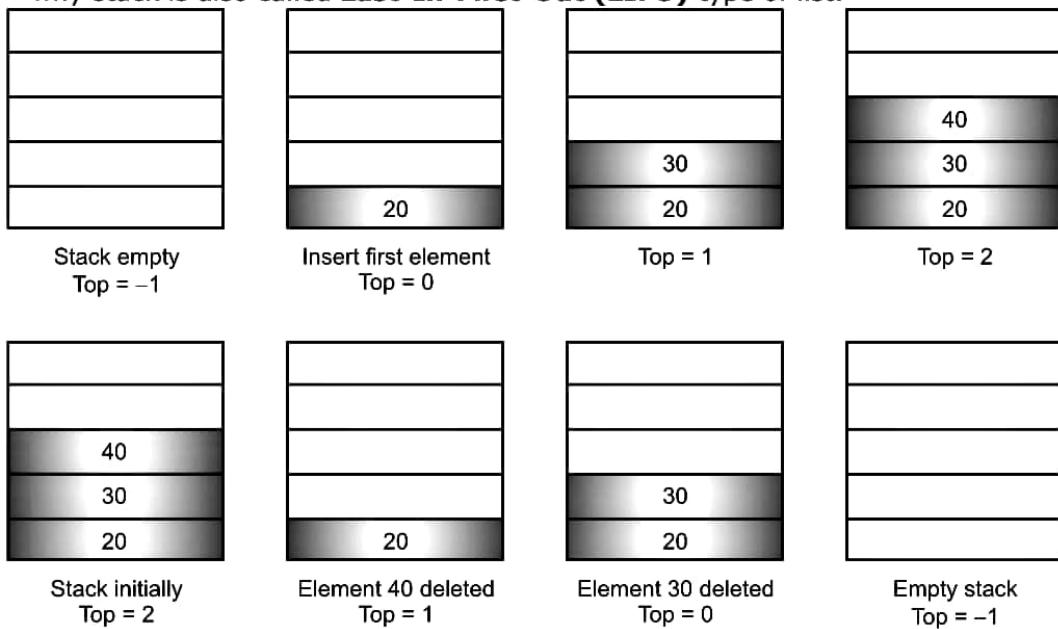


Fig. 4.2: Operation on stack

4.1.1 What Is Stack?

- A stack is an ordered list in which all insertions and deletions are made at one end, called the top.
- Associated with each stack there will be a variable, 'top', which points to the top element in the stack.
- There are two basic operations, which can be performed on a stack: adding an element on the stack and removing an element from the stack.
- Adding an element is called as pushing the element onto the stack. The function, which does this, is called 'push'.
- Removing an element from the stack is called as popping the element from the stack and the function, which does this, is called 'pop'.
- Given a stack $S = (q_1, q_2, \dots, q_n)$, we say that as q_n is bottommost element and q_1 is on top of the stack, and element q_{i-1} is said to be on the top of q_i , $1 < i \leq n$.
- Let $S = (C, B, A)$ then A is bottommost element and C is the topmost element.

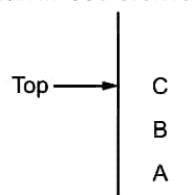


Fig. 4.3: Stack



4.2 STATIC AND DYNAMIC REPRESENTATION OF STACK

- Stack can be represented in two ways:
 1. Static representations, (Representation of stacks using Arrays).
 2. Dynamic representations, (Representation of stacks using Linked List).

4.2.1 Static Representation

[April 15, 17]

- It uses arrays to store stack elements.
- Static implementation is a simple technique but is not a flexible way, as the size of stack to be declared during program design, after that the size cannot be varied.
- Static implementation is not too efficient with respect to memory utilization. As the declaration of array is done before the start of the operation, now if there are too few elements to be stored in the stack the statically allocated memory will be wasted.
- On the other hand, if there are more number of elements to be stored in the stack then we are not able to change the size of array to increase its capacity, so that it can accommodate new elements.

Defining static stack

```
#define MAXSIZE 10
typedef struct stack
{
    int arr[MAXSIZE];
    int top;
} STATIC_STACK;
```

- Here st is a static stack which can store maximum 10 integer values.
- Initially top contains -1.
 - When a new element is pushed it is increased by one first and then value is stored at arr[top].
 - When element is popped, first value is returned as return (arr[top]) and then top is decreased by one.
 - When top reaches MAX-1, stack overflow occurs (i.e. stack is full).
 - When top becomes -1, stack underflow occurs (i.e. stack is empty).
 - In case of static representation of stack, as array is used to store the stack element, there is limit on maximum elements in the stack.

**Program 4.1:** Program for stack representation of an array.

```
# define MAX 50
typedef struct
{
    int data[MAX];
    int top;
}STATIC_STACK;
void initstack(STATIC_STACK *s)
{
    int i;
    for(i=0;i<MAX;i++)
        s->data[i]=-1;
    s->top=-1;
}
int isfull(STATIC_STACK *s)
{
    if(s->top==MAX - 1)
        return 1;
    else
        return 0;
}
int isempty(STATIC_STACK *s)
{
    if(s->top== -1)
        return 1;
    else
        return 0;
}
void push(STATIC_STACK *s, int no)
{
    (s->top)++;
    s->data[s->top]=no;
}
int pop(STATIC_STACK *s)
{
    int no;
    no=s->data[s->top];
    s->top--;
    return(no);
}
```



```
void display(STATIC_STACK *s)
{
    int i;
    printf("\nStack contents are:");
    for(i=s->top;i>=0;i--)
        printf("\n%d",s->data[i]);
}

void main()
{
    STATIC_STACK st;
    int ch,n;
    clrscr();
    initstack(&st); //address of stack is passed
    do
    {
        printf("\n1.PUSH \n2.POP \n3.DISPLAY \n4.EXIT");
        printf("\nEnter ur choice: ");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1: printf("\nEnter a number:");
                      scanf("%d",&n);
                      if(!isfull(&st))
                          push(&st,n);
                      else
                          printf("\nStack is full.");
                      break;
            case 2: if(!isempty(&st))
                      printf("\nPopped number is %d",pop(&st));
                      else
                          printf("\nStack is empty.");
                      break;
            case 3: if(!isempty(&st))
                      display(&st);
                      else
                          printf("\nStack is empty.");
                      break;
            case 4: break;
        }
    }
    while(ch!=4);
}
```

**Output:**

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 1  
Enter a number:12
```

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 1  
Enter a number:67
```

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 1  
Enter a number:89
```

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 1  
Enter a number:12
```

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 3
```

```
Stack contents are:  
12  
89  
67  
12
```

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 2
```



```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 2
```

```
Popped number is 12  
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 2
```

```
Popped number is 89  
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 2
```

```
Popped number is 67  
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 2
```

```
Popped number is 12  
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 2
```

```
Stack is empty.  
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 4
```



4.2.2 Dynamic Representation

- To remove the drawback of static stack (i.e. limit on the no. of elements in the stack), dynamic representation of stack is used.
- Linked list is used to store stack elements.
- It is also called linked list representation and uses pointers to implement the stack type of data structure.
- Pointer is address of element, each element is represented with node and each node is divided into two parts, the first part contains the information of the element and the second part called the link field contains the address of the next node in the list.
- Fig. 4.4 shows the linked list representation of the stack.

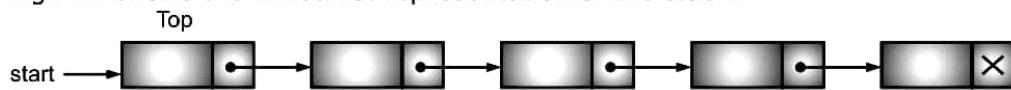


Fig. 4.4: Linked list with 5 nodes

- **Definition of dynamic stack is:**

```
typedef struct node
{
    int data;
    struct node * next;
} DYNAMIC_STACK;
DYNAMIC_STACK * top;
```

Here top is a pointer to a topmost element (node) of a stack.

- When an element is pushed into the dynamic stack, it is inserted at the beginning of the list and becomes a head node i.e. top node.
- When an element is popped from the dynamic stack first node (top node). Value is returned and first node is deleted. In this case next node becomes top node.
- As there is no limit on no. of elements, stack overflow condition occur only when there is no sufficient memory available to a new node.
- Stack underflow condition occurs when top becomes NULL.
- When we create stack to store integer, real characters, string, structure or any user defined typed data.

Program 4.2: Program for dynamic implementation of stack is as follows:

```
#include <stdio.h>
#include <malloc.h>
typedef struct node
{
    int data;
    struct node * next;
} DYNAMIC_STACK;
```



```
void initstack(DYNAMIC_STACK **top)
{
    *top = NULL;
}
int isempty(DYNAMIC_STACK **top)
{
    if (*top==NULL)
        return 1;
    else
        return 0;
}
void push(DYNAMIC_STACK **top, int n)
{
    DYNAMIC_STACK *temp;
    temp=(DYNAMIC_STACK *)malloc(sizeof(DYNAMIC_STACK));
    temp->data=n;
    temp->next=*top;
    *top=temp;
}
int pop(DYNAMIC_STACK **top)
{
    int n;
    DYNAMIC_STACK *temp;
    temp=*top;
    n=temp->data;
    *top=temp->next;
    temp->next=NULL;
    free(temp);
    return(n);
}
void display(DYNAMIC_STACK **top)
{
    DYNAMIC_STACK *temp=*top;
    printf("\nStack contents are:");
    while(temp!=NULL)
    {
        printf("\n%d",temp->data);
        temp=temp->next;
    }
}
```



```
void main()
{
    DYNAMIC_STACK *top;
    int ch,n;
    clrscr();
    initstack(&top); //address of top is passed
    do
    {
        printf("\n1.PUSH \n2.POP \n3.DISPLAY \n4.EXIT");
        printf("\nEnter ur choice: ");
        scanf("%d", &ch);

        switch(ch)
        {
            case 1: printf("\n Enter a number: ");
                      scanf(" %d", &n);
                      push(&top, n);
                      break;
            case 2: if(!isempty(&top))
                      {
                          n=pop(&top);
                          printf("\n Popped number is: %d", n);
                      }
                      else
                          printf("\nSTACK IS EMPTY.");
                      break;
            case 3: if(!isempty(&top))
                      display(&top);
                      else
                          printf("\nSTACK IS ENMPTY.");
                      break;
            case 4: exit(0);
        }
    }while(ch!=4);
}
```

Output:

```
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter ur choice: 1
Enter a number: 12
```



```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 1  
Enter a number: 5
```

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 1  
Enter a number: 78
```

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 1  
Enter a number: 23
```

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice:3
```

```
Stack contents are:  
23  
78  
5  
12
```

```
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 2
```

```
Popped number is: 23  
1.PUSH  
2.POP  
3.DISPLAY  
4.EXIT  
Enter ur choice: 2
```



```
Popped number is: 78
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter ur choice: 2

Popped number is: 5
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter ur choice: 2

Popped number is: 12
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter ur choice: 2

STACK IS EMPTY.
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter ur choice: 4
```

4.3 PRIMITIVE OPERATIONS ON STACK

- A stack consists of following operations:
 1. **CREATE or INITIALIZE:** Creates a new Stack. This operation creates a new stack which is empty.
 2. **PUSH:** The process of adding a new element to the top of stack is called Push operation. Pushing an element in the stack invoke adding of element. As the new element is inserted at the top, after every push operation the top is incremented by one. In case the array is full and no new element can be accommodated, it is called STACK-FULL condition. This condition is called STACK OVERFLOW.
 3. **POP:** The process of deleting an element from the top of stack is called Pop operation. After every pop operation the stack is decremented by one. If there is no element on the stack and the pop is performed then this will result into STACK UNDERFLOW condition.



4. **IS EMPTY:** Checks whether a stack is empty. This operation returns TRUE if the stack is empty and FALSE otherwise. This is required for the pop operation because we cannot pop from an empty stack.
- In case of static representation of stack (using array), one more operation is required i.e. IS_FULL which checks whether stack is full or not. This operation returns TRUE if the stack is full otherwise FALSE. This operation is required to be performed before PUSH operation.

4.3.1 Stack Terminology

1. **Context:** The environment in which a function executes. It includes argument values, local variables and global variables. The entire context except the global variables is stored in a stack frame.
2. **Stack Frame:** The data structure containing all the data (arguments, local variables, return address etc.) needed each time a procedure or function is called.
3. **Maxsize:** This term is not a standard one; we use this term to refer the maximum size of stack in case stack is represented statically using array.
4. **Top:** This term refers to the top of stack. The stack Top is used to check stack overflow or underflow conditions. Initially Top stores -1. This assumption is taken so that whenever an element is added to the stack the Top is first incremented and then the item is inserted into the location currently indicated by the Top.
5. **Stack empty or Underflow:** This is the situation when the stack contains no element. At this point, the top of stack is present at the bottom of the stack.
6. **Stack Overflow:** This is the situation when the stack becomes full and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location of the stack.

4.4 APPLICATIONS OF STACK

- Stack data structure is used in wide range of applications. Few of them are listed below:
 1. Conversion from Infix to Postfix and Prefix expressions.
 2. Evaluating the Postfix Expressions.
 3. Processing of Subprogram Calls.
 4. Reversing a String.
 5. Checking Correctness of Nested Parenthesis.
 6. Simulating Recursion.
 7. Parsing (analyzing the structure) of computer programs.
 8. Backtracking algorithms (often used in optimizations and in games).
 9. Computation like decimal to binary conversion.



4.5 EVALUATION OF POSTFIX AND PREFIX EXPRESSIONS

[April 15, 16]

- The most frequent application is in evaluation of arithmetic expressions. An arithmetic expression is made up of operands, operators and delimiters.
- When higher level programming language came into existence one of the major difficulty faced by computer scientists was to generate machine language instruction, which would properly evaluate any arithmetic expression.
- A complex assignment statement such as,
$$X = (A/B+C*D - F*G/Q)$$

might have several meaning; and even if meaning were uniquely defined, still it is difficult to generate correct and reasonable instruction sequence. Fortunately, the solution we have today is both elegant and simple. As of today, this conversion is considered to be one of the major aspects of compiler writing.

- Let us see what are the difficulties in understanding the meaning of expression. The first problem with understanding the meaning of an expression is to decide in what order the operations are to be carried out. This needs that every language must uniquely define such an order.
- For instance consider the following expression:
$$X=A/B*C-D$$
- Let, A=2, B=3, C=4 and D=5.

One of the meaning which can be drawn from above expression could be,

$$X=(2/3)*(4-5)=-2/3$$

Another way to evaluate the same expression could be,

$$X=(2/(3*4))-5=-58/12$$

- To avoid more than one meaning to be drawn out of one expression, we could specify order by using parentheses as,

$$X=(A/B)*(C-D)$$

- To fix the order of evaluation, let us assign to each operation a priority. Because even though we write expression in parenthesis, still we have question in mind, about whether to evaluate (A/B) first or evaluate (C-D) first.
- Once, the priorities are assigned then within any pairs of parenthesis we can understand that operators with highest priority are to be evaluated first while evaluating expression usually the following operation precedence is used.
- The following operators are written in descending order their (precedence),
 - Exponentiation ^, Unary +, Unary -, and NOT ~
 - Multiplication * and division /
 - Addition + and subtraction -
 - <, ≤ , =, ≠, ≥, >
 - AND
 - OR



- Some integer values as priority can be assigned to them as,

Operator Arithmetic, Boolean and Relational	Priority
\wedge , Unary +, Unary - , ~	6
* /	5
+ -	4
<, ≤ , =, ≠, ≥, >	3
AND	2
OR	1

- Note that, all of the relational operators are having the same priority. Exponentiation, unary operators (\wedge , unary +, - and ~) have top priority. When there are two adjacent operators with same priority, again the question arises, which one to evaluate first.
- For example, The expression, A + B – C means,
 $(A+B)-C$ or $A+(B-C)$?
- This needs deciding whether to evaluate expression from right to left or left to right. The expressions such as $A+B-C$ and $A*B/C$ are evaluated from left to right. Whereas, the expression A^B^C is evaluated as $A^{(B^C)}$ i.e. from right to left. Hence, the operators need to decide a rule for proceeding left to right for all except exponential.
- When we write parenthesized expression, these rules can be overridden. In parenthesized expression, the innermost parenthesized expression is evaluated first. The order of evaluation is either left to right or right to left is called associativity. Exponentiation is right associative and all other are left associative operators.
Let $X=A/B^C+D*E-A*C$
- By using priority and associativity rules, we get,

$$X=A/(B^C)+(D*E)-(A*C)$$
- Still the question is, how can a compiler accept such an expression and produce correct code. The solution is converting of the expression into a form called postfix notation.

**Polish Notation:**

- A polish Mathematician Han Lukasiewicz suggested a notation called polish notation, which gives two alternatives to represent an arithmetic expression. The notations are Postfix and prefix notations.
- The fundamental property of polish notation is that the order in which the operations are to be performed is determined by the positions of the operators and operands in the expression. Hence, the advantage is parenthesis are not required while writing expressions in polish notation.
- If an exp is an expression with operand and operators, the conventional way of writing expression is called infix, because the binary operators occur in between the operands and unary operators precede their operand.
- For example, $((a+b)*c)/d$
- In postfix notation, the operator is written after its operands, whereas in prefix notation the operator precedes its operands.

Infix	Prefix	Postfix
(operand) (operator)	(operator) (operand)	(operand) (operator)
(operand)	(operand)	(operator).
$(A+B)*C$	$*+ABC$	$AB+C*$

- **Example:**

$$X = A/B^C + D * E - A * C$$

- By applying rule of priority and associativity the above expression can be written as:

$$X = ((A/(B^C)) + (D * E)) - (A * C)$$

- To convert into postfix form, take operators to the right side, outside the brackets and rewrite expression without brackets.

$$(((A/(B^C)) + (D * E)) - (A * C))$$

Postfix expression

$$ABC^/DE^*+AC*-$$

- To convert into prefix form, take operators to the left side, outside the brackets and rewrite expression without brackets.

$$X = (((A/(B^C)) + (D * E)) - (A * C))$$

$$X = ((A/(B^C)) + (D * E) - (A * C))$$

Prefix expression:

$$- + / A ^ BC * DE * AC$$



Characteristics of Polish Notation

- The important property of polish notation (prefix or postfix) is that, the order in which the operations are to be performed is determined by the positions of operator and operand.
- **Why prefix and postfix?**
 - The need for parenthesis is eliminated.
 - The priority of operators is no longer relevant.
 - The expression evaluation process is much simpler than attempting a direct evaluation from infix notation.

4.5.1 Evaluation of Postfix Expression

[April 17]

- The postfix expression may be evaluated by making a left to right scan, push operands into the stack and evaluating operators by popping correct number of operands from the stack and finally placing the result again onto stack.
- This evaluation process is much simpler than attempting a direct evaluation from infix notation. Process continues till stack is not empty or on occurrence of a character # which denotes end of expression.
- The following **Evaluate** procedure evaluates the postfix expression E. It is assumed that the last character in E is '#'.
- A procedure NEXT_TOKEN is used to extract from E the next token. A token is either an operand, operator or #. A one dimensional array STACK[n] is used as a stack.
- **Procedure Evaluate (E):**

Begin

```
    Top = -1
    Loop
        X = NEXT_TOKEN (E)
        case
            X= #
                then return
            X is an operand
                then call PUSH(X,STACK,n,top)
            else
                remove the correct number of operands for operator X
                from STACK, perform the operation and store the
                result, if any, onto the stack.
        end case
    end loop
End Evaluate
```

- Let us consider example $E=AB+C*#$. Now let us scan the above expression from left to right character by character.

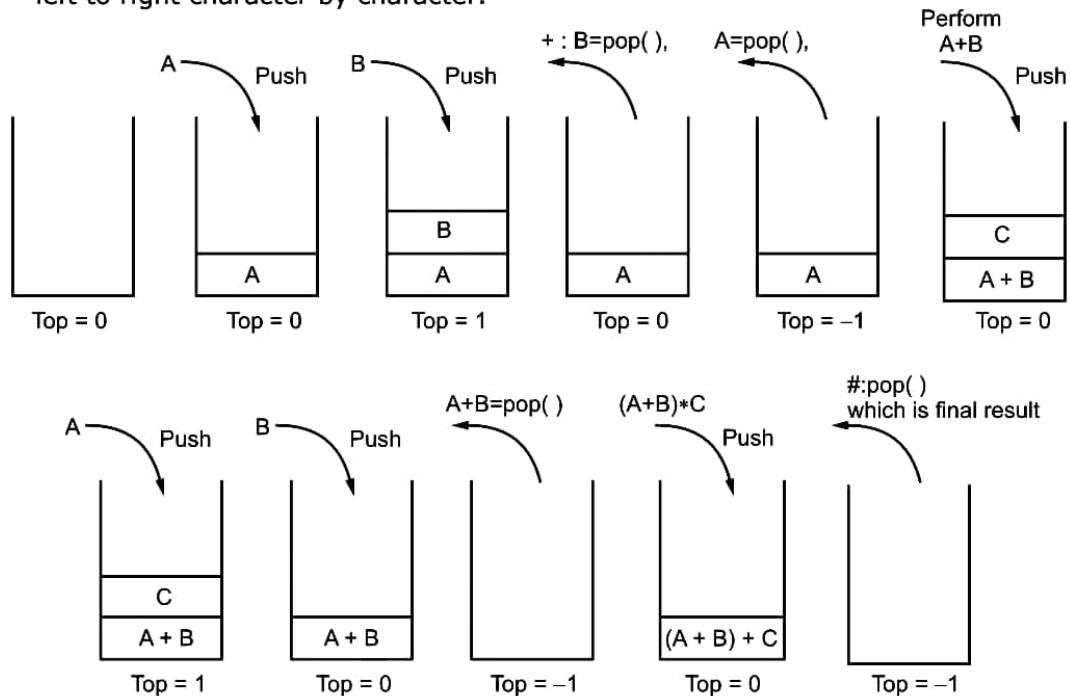


Fig. 4.5: Evaluation of Infix Expression

Example:

- This evaluation process is much simpler than evaluation of infix expression.
Postfix expression: ABC + DE + F * + G/+

Token	Stack	Operand 1	Operand 2	Result	Stack Operation
A	A				Push
B	A B				Push
C	A B C				Push
+	A	B	C	$R_1 = B + C$	OP2= $\text{pop}()$ OP1= $\text{pop}()$ Perform $B+C$, push result into stack Push
		AR ₁			
D	A R ₁ D				Push
E	AR ₁ DE				Push
+		D	E	$R_2 = D + E$	OP2= $\text{pop}()$ OP1= $\text{pop}()$ Perform $D+E$, push result into stack
		AR ₁ R ₂			

contd. ...



F	A R ₁ R ₂ F				push
*		R ₂	F	R ₃ = R ₂ *F	op2=pop() op1=pop() perform R ₂ * F push result into stack
+		R ₁	R ₃	R ₄ =R ₁ +R ₃	op2=pop() op1=pop() perform R ₁ +R ₃ push result into stack
G	A R ₄ G	R ₄	G	R ₅ = R ₄ /G	Push op2=pop() op1=pop() perform R ₄ /G Push result into stack
+		A	R ₅	R ₆ = A+R ₅	op2=pop() op1=pop() Perform A+R ₅ Push result into stack
No symbol					result=pop() i.e. R ₆

Program 4.3: Program for evaluating postfix expression using static stack is as follows:

```
# include<stdio.h>
# include<math.h>
# define MAX 50
typedef struct
{
    int data[MAX];
    int top;
}STATIC_STACK;
void initstack(STATIC_STACK *s)
{
    int i;
    for(i=0;i<MAX;i++)
        s->data[i]=-1;
    s->top=-1;
}
```



```
int isfull(STATIC_STACK *s)
{
    if(s->top==MAX - 1)
        return 1;
    else
        return 0;
}
int isempty(STATIC_STACK *s)
{
    if(s->top== -1)
        return 1;
    else
        return 0;
}
void push(STATIC_STACK *s, int no)
{
    (s->top)++;
    s->data[s->top]=no;
}
int pop(STATIC_STACK *s)
{
    int no;
    no=s->data[s->top];
    s->top--;
    return (no);
}
int evaluate(char *poststr)
{
    STATIC_STACK st;
    int i,value,op1,op2;
    initstack(&st);
    i=0;
    while(poststr[i]!='\0')
    {
        if(isalpha(poststr[i]))
        {
            printf("\nEnter the value for operand %c: ",poststr[i]);
            scanf("%d",&value);
            push(&st,value);
        }
    }
}
```



```
else
{
    if(!isempty(&st))
        op2=pop(&st);
    else
    {
        printf("\nWrong postfix expression.");
        exit(0);
    }
    if(!isempty(&st))
        op1=pop(&st);
    else
    {
        printf("\nWrong postfix expression.");
        exit(0);
    }
    switch(poststr[i])
    {
        case '+' : value=op1+op2;
                    break;
        case '-' : value=op1-op2;
                    break;
        case '*' : value=op1*op2;
                    break;
        case '/' : value=op1/op2;
                    break;
        case '%' : value=op1%op2;
                    break;
    }
    push(&st,value);
}
i++;
}
value=pop(&st);
return(value);
}
void main()
{
    char poststr[20]="";
    int val;
    clrscr();
    printf("\nEnter postfix expression : ");
    scanf("%s",poststr);
    val=evaluate(poststr);
    printf("Result of postfix expression is : %d",val);
}
```

**Output:**

```
Enter postfix expression : ab+c*d/
Enter the value for operand a: 12
Enter the value for operand b: 3
Enter the value for operand c: 54
Enter the value for operand d: 6
Result of postfix expression is : 135
```

4.5.2 Evaluation of Prefix Expansion

- For evaluating prefix expression, read the expression from right to left and to the same procedure as that for evaluating postfix expression. That is if symbol is operand push it into the stack and if it is operator, evaluate it by popping correct number of operands from stack. Push the result into the stack. Continue this for entire expression.

Example: Consider prefix expression + a * b + cd.

Token	Stack	Operand 1	Operand 2	Result	Stack Operation
d	d				Push
c	dc				Push
+		c	d	$R_1 = c + d$	op1=pop() op2=pop() Perform c+d Push result
	R_1				
b	$R_1\ b$				Push
*		b	R_1	$R_2 = b * R_1$	op1=pop() op2=pop() perform b * R_1 Push Result
	R_2				
a	$R_2\ a$				Push
+		a	R_2	$R_3 = a + R_2$	op1=pop() op2=pop() perform a + R_2 Push result
no symbol					resultpop() i.e. R_3

**Program 4.4:** Program for evaluating prefix expression using static stack.

```
# include<stdio.h>
# include<string.h>
# include<math.h>
# define MAX 50
typedef struct
{
    int data[MAX];
    int top;
}STATIC_STACK;
void initstack(STATIC_STACK *s)
{
    int i;
    for(i=0;i<MAX;i++)
        s->data[i]=-1;
    s->top=-1;
}
int isfull(STATIC_STACK *s)
{
    if(s->top==MAX - 1)
        return 1;
    else
        return 0;
}
int isempty(STATIC_STACK *s)
{
    if(s->top==-1)
        return 1;
    else
        return 0;
}
void push(STATIC_STACK *s, int no)
{
    (s->top)++;
    s->data[s->top]=no;
}
int pop(STATIC_STACK *s)
{
    int no;
    no=s->data[s->top];
    s->top--;
    return(no);
}
```



```

int evaluate(char *prestr)
{
    STATIC_STACK st;
    int i,value,op1,op2;
    initstack(&st);
    i=strlen(prestr)-1;
    while(i>=0)
    {
        if(isalpha(prestr[i]))
        {
            printf("\nEnter value for operand %c : ",prestr[i]);
            scanf("%d",&value);
            push(&st,value);
        }
        else
        {
            if(!isempty(&st))
                op1=pop(&st);
            else
            {
                printf("\nWrong prefix expression.");
                exit(0);
            }
            if(!isempty(&st))
                op2=pop(&st);
            else
            {
                printf("\nWrong prefix expression.");
                exit(0);
            }
            switch(prestr[i])
            {
                case '+' : value=op1+op2;
                            break;
                case '-' : value=op1-op2;
                            break;
                case '*' : value=op1*op2;
                            break;
                case '/' : value=op1/op2;
                            break;
            }
        }
    }
}

```



```

        case '%': value=op1%op2;
                     break;
    }
    push(&st,value);
}
i--;
}
value=pop(&st);
return(value);
}
void main()
{
    char prestr[20]="";
    int val;
    clrscr();
    printf("\nEnter prefix expression : ");
    scanf("%s",prestr);
    //To evaluate prefix expression
    val=evaluate(prestr);
    printf("\nResult of prefix expression is : %d",val);
}

```

Output:

```

Enter prefix expression : +*abc
Enter value for operand c : 1
Enter value for operand b : 2
Enter value for operand a : 3
Result of prefix expression is : 7

```

4.6 CONVERSION OF EXPRESSION - Infix to Prefix and Infix to Postfix**4.6.1 Infix to Prefix Conversion**

A + B * C	infix form
A + (B * C)	arenthesize expression
A + (* B C)	convert multiplication
+ A (* B C)	convert addition
+ A * B C	prefix form

Example 4.6: (A * B) + C

Sol.: (* A B) + C
T + C
+ T C
+ * A B C

**Example 4.7: $A / (B ^ C) + D$**

Sol.: $A / (^ B C) + D$
 $A / T + D$
 $(A / T) + D$
 $(/ A T) + D$
 $S + D$
 $+ S D$
 $+ / A T D$
+ / A ^ B C D

Example 4.8: $(A - B / C) * (D * E - F)$

Sol.: $(A - (B / C)) * ((D * E) - F)$
 $(A - (/ B C)) * ((* D E) - F)$
 $(A - T) * (S - F)$
 $(- A T) * (- S F)$
 $Q * P$
 $* Q P$
 $* - A T - S F$
*** - A / B C - * D E F**

Example 4.9: $(A * B + (C / D)) - F$

Sol.: $(A * B + (/ C D)) - F$
 $(A * B + T) - F$
 $((* A B) + T) - F$
 $(S + T) - F$
 $(+ S T) - F$
 $Q - F$
 $- Q F$
 $- + S T F$
- + * A B / C D F

Example 4.10: $A / B ^ C - D$

Sol.: $A / (B ^ C) - D$
 $A / (^ B C) - D$
 $A / T - D$
 $(/ A T) - D$
 $S - D$
 $- S D$
 $- / A T D$
- / A ^ B C D



- **Algorithm to convert infix to prefix**

Steps:

1. Accept infix expression say 'instr'.
2. Initialise a character stack to store operators and an output string 'prestr' to empty.
3. Check whether expression 'instr' is parenthesis balanced. If not stop.
4. Scan 'instr' from right to left and repeat steps 5 to 8 for each symbol of 'instr' until last symbol.
5. If symbol is ')' push it into stack.
6. If symbol is operand, append it to 'prestr'.
7. If symbol is '(', pop operator from stack and append it to 'prestr' until ')' is encountered (do not append ')' to 'prestr').
8. If symbol is operator, pop the operator from stack and append it to 'prestr' till priority of topmost operator is greater than equal to current operator. Push current operator into the stack.
9. Pop operator from stack and append it to 'prestr' till stack is not empty.
10. Print 'prestr' which is reverse order is prefix expression.
11. Stop.

Program 4.5: Infix to Prefix conversion.

```
# include<stdio.h>
# include<string.h>
// Static stack to store operators
typedef struct
{
    char data[MAX];
    int top;
}STATIC_STACK;
void initstack(STATIC_STACK *s)
{
    int i;
    for(i=0;i<MAX;i++)
        s->data[i]=-1;
    s->top=-1;
}
int isfull(STATIC_STACK *s)
{
    if(s->top==MAX - 1)
        return 1;
    else
        return 0;
}
```





```
else if(str[i]==']')
{
    if(isempty(&st) || pop(&st) != '[')
return(0);
    else
i++;
}
else if(str[i]=='}')
{
    if(isempty(&st) || pop(&st) != '{')
return(0);
    else
i++;
}
else
    i++;
}

if(!isempty(&st))
    return(0);
else
    return(1);
}

//Function to check whether character is operator or not
int isoperator(char ch)
{
    char op[8]={'+', '-', '*', '/', '%', '^', '(', ')'};
    int i;
    for(i=0;i<8;i++)
    {
        if(ch==op[i])
            return(1);
    }
    return(0);
}
//Function to get priority of operator
int priority(char ch)
{
    switch(ch)
    {
        case ')' :

```



```
        case '(' : return 0;
        case '+' :
        case '-' : return 1;
        case '*' :
        case '/' :
        case '%' : return 2;
        case '^' : return 3;
    }
    return 0;
}
//Function to convert infix to prefix
void prefix(char infix[],char prefix[])
{
    int i, j = 0;
    STATIC_STACK opstk;
    char symbol,ch;
    initstack(&opstk);
    infix = strrev(infix);
    for (i = 0;i<strlen(infix);i++)
    {
        symbol = infix[i];
        if (isoperator(symbol) == 0)
        {
            prefix[j] = symbol;
            j++;
        }
        else
        {
            if (symbol == ')')
            {
                push(&opstk,symbol);
            }
            else if (symbol == '(')
            {
                while ((ch=pop(&opstk)) != ')')
                {
                    prefix[j] = ch;
                    j++;
                }
            }
        }
    }
}
```



```
else
{
    if (priority(symbol) > priority(stacktop(&opstk)))
    {
        push(&opstk, symbol);
    }
    else
    {
        while (priority(symbol) <= priority(stacktop(&opstk)))
        {
            prefix[j] = pop(&opstk);
            j++;
        }
        push(&opstk, symbol);
    } //end of else.
} //end of else.
} //end of for.
while (!isempty(&opstk))
{
    prefix[j] = pop(&opstk);
    j++;
}
prefix[j] = '\0'; //null terminate string.
prefix = strrev(prefix);
}
void main()
{
    char prestr[20]="", instr[20]="";
    // clrscr();
    printf("\nEnter Infix Expression : ");
    scanf("%s", instr);
    // to check parenthesis balance
    if(!check_para_balance(instr))
        printf("\nExpression is not parenthesis balanced.");
    else
    {
        // To convert infix to postfix expression
        prefix(instr, prestr);
        printf("\nPrefix expression is : %s", prestr);
    }
}
```

Output:

```
Enter Infix Expression : (a+b)*(c-d)
Prefix expression is : *+ab-cd
```



Example: Infix expression: $A + B / (C - D)$

Symbol	poststr	stack	Operation
))	Push
D	D)	Append to prestr
-	D)-	Push
(DC)-	Append to prestr
(DC -		pop operator till ')
/	DC-	/	Push
B	DC-B	/	Append to prestr
+	DC-B/	+	pop '/' & push '//'
A	DC-B/A	+	append to prestr
No symbol	DC-B/A+		pop '+'

Prefix expression = $+ A/B - CD$.

4.6.2 Infix to Postfix Conversion

[April 17, Oct. 17]

$A + B * C$	infix form
$A + (B * C)$	parenthesized expression
$A + (B C *)$	convert the multiplication
$A (B C *) +$	convert the addition
$A B C * +$	postfix form

Convert the expression into postfix form:

Example 3.1: $A + [(B+C) + (D+E) * F]/G$

Sol.: $A + [\{ (B C +) + (D E +) F * \} / G]$

$A + [\{ (B C +) + (D E + F *) \} / G]$

$A + [\{ (B C + (D E + F * +) \} / G]$

$A + [B C + D E + F * + G /]$

A B C + D E + F * + G / + Postfix form

Example 4.2: $(A * B + C)$

Sol.: $(A B *) + C$

$(T) + C$

$T C +$

A B * C +

Postfix form

**Example 4.3: A + B / C - D****Sol.:** A + (B / C) - D

A + (B C /) - D

A + T - D

(A + T) - D

(A T +) - D

S - D

S D -

A T + D -

A B C / + D -**Example 4.4: (A + B) * C / D****Sol.:** (A B +) * C / D

T * C / D

(T * C) / D

(T C *) / D

S / D

S D /

T C * D /

A B + C * D /**Example 4.5: (A + B) * C / D + E ^ F / G****Sol.:** (A B +) * C / D + (E ^ F) / G

T * C / D + (E F ^) / G

T * C / D + S / G

(T * C) / D + S / G

(T C *) / D + S / G

Q / D + S / G

(Q / D) + (S / G)

(Q D /) + (S G /)

Q D / S G / +

T C * D / E F ^ G / +

A B + C * D / E F ^ G / +

Infix Notation	Postfix Form
(A + B) / (C - D)	A B + C D - /
A + [(B + C) + (D + E) * F] / G	A B C + D E + F * G / +
A + (B * C - (D / E ^ F) * G) * H	A B C * D E F ^ / G * - H * +
A - B / (C * D ^ E)	A B C D E ^ * / -

**Algorithm for converting infix to postfix expression:**

This algorithm considers both fully parenthesized or non-parenthesized or non-parenthesized expressions. It also considers priority of the operators.

Steps:

1. Accept infix expression say 'instr'.
2. Initialize a character stack to store operators and an output string 'poststr' to empty.
3. Check whether expression is parenthesis balanced. If not, stop.
4. Scan 'instr' from left to right and repeat steps 5 to 8 for each symbol of instr until last symbol.
5. If symbol is '(' push it into stack.
6. If symbol is operand, append it to 'poststr'.
7. If symbol is ')', pop operator from stack and append it to 'poststr' until a left parenthesis is encountered (do not append left parenthesis to 'poststr').
8. If symbol is operator, pop the operator from stack and append it to 'poststr' till priority of topmost operator is greater than or equal to current operator. Push current operator into the stack.
9. Pop operator from stack and append it to 'poststr' till stack is not empty.
10. Print 'poststr' which is postfix expression.
11. Stop.

Program 4.6: Program for infix to postfix conversion.

```
# include<stdio.h>
# include<math.h>
# define MAX 50
typedef struct
{
    char data[MAX];
    int top;
}STATIC_STACK;
void initstack(STATIC_STACK *s)
{
    int i;
    for(i=0;i<MAX;i++)
        s->data[i]=-1;
    s->top=-1;
}
int isfull(STATIC_STACK *s)
{
    if(s->top==MAX - 1)
        return 1;
    else
        return 0;
}
```





```
else if(str[i]=='])'
{
    if(isempty(&st) || pop(&st) != '[')
        return(0);
    else
        i++;
}
else if(str[i]=='}')
{
    if(isempty(&st) || pop(&st) != '{')
        return(0);
    else
        i++;
}
if(!isempty(&st))
    return(0);
else
    return(1);
}

int isoperator(char ch)
{
    char op[5]={'+', '-', '*', '/', '%'};
    int i;
    for(i=0;i<5;i++)
    {
        if(ch==op[i])
            return(1);
    }
    return(0);
}

int priority(char ch)
{
    switch(ch)
    {
        case '(' : return 0;
        case '+' :
        case '-' : return 1;
        case '*' :
```



```
        case '/':
        case '%': return 2;
        case '^': return 3;
    }
    return 0;
}
void postfix(char *str,char * result)
{
    STATIC_STACK top;
    int i,j=0;
    char ch;
    initstack(&top);
    i=0;
    while(str[i]!='\0')
    {
        if(str[i]=='(')
        push(&top,str[i]);
        else
        if(isoperator(str[i]))
        {
            while(!isempty(&top) &&
                   priority(str[i])<=priority(stacktop(&top)))
                result[j++]=pop(&top);
            push(&top,str[i]);
        }
        else
        if(str[i]==')')
        {
            while(1)
            {
                if(!isempty(&top))
                {
                    ch=pop(&top);
                    if(ch=='(')
                        break;
                    result[j]=ch;
                    j++;
                }
            else
            {
                printf("\nWrong infix expression****");
                exit(0);
            }
        }
    }
}
```



```
else
{
    result[j]=str[i];
    j++;
}
i++;
}
while(!isempty(&top))
{
    ch=pop(&top);
    result[j]=ch;
    j++;
}
result[j]='\0';
}
void main()
{
    char poststr[20]="",instr[20]++;
    int val;
    clrscr();
    printf("\nEnter Infix Expression : ");
    scanf("%s",instr);
    // to check parenthesis balance
    if(!check_para_balance(instr))
        printf("\nExpression is not parenthesis balanced.");
    else
    {
        // To convert infix to postfix expression
        postfix(instr,poststr);
        printf("\nPostfix expression is : %s",poststr);
    }
}
```

Output :

```
Enter Infix Expression : (a+b)*(c-d)
Postfix expression is : ab+cd-*
```



Example: Convert following infix expression to post (A + B) * C/D.

Symbol	poststr	stack	Operation
(A	(Push
A	A		Append to poststr
+	A	(+	Push
B	AB	(+	Append to poststr
)	AB+		pop till '('
*	AB+	*	Push
C	AB+C	*	Append to poststr
/	AB+C*	/	Pop '*' & push '/'
D	AB+C*D	/	Append to poststr
No symbol	AB+C*D/		Pop till stack is not empty

Postfix expression: AB+C*D/

4.7 INTRODUCTION TO QUEUE

- A queue is an ordered collection of homogeneous items from which items may be deleted at one end (called the *front* of the queue) and into which items may be inserted at the other end (called the *rear* of the queue).
- Fig. 4.6 shows a queue containing three elements A, B and C. A is at the front of the queue and C is at the rear.

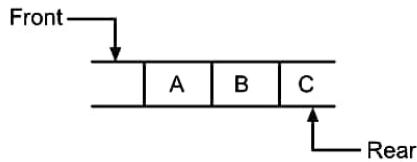


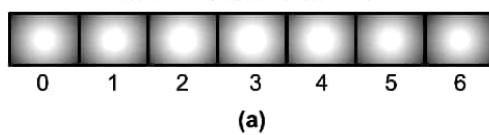
Fig. 4.6: Queue

- A queue is logically a First In First Out (FIFO) type of list.
- Queue means a line, for example at railway reservation booth; we have to get into the reservation queue. New customers got into the queue from the rear end, whereas, the customers who get their seats reserved leave the queue from the front end. It means the customers are serviced in the order in which they arrive the service center.
- Thus, a queue is a non-preemptive linear data structure.



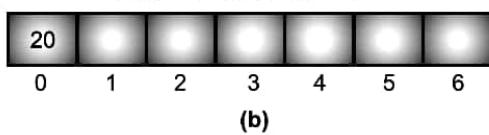
- The following Fig. 4.7 show queue graphically during insertion operation:

Rear = -1 and Front = -1



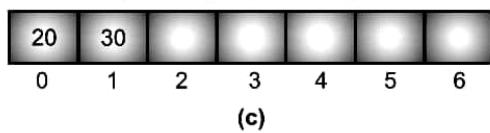
(a)

Rear = 0 and Front = -1



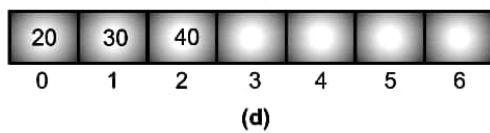
(b)

Rear = 1 and Front = -1



(c)

Rear = 2 and Front = -1



(d)

Fig. 4.7: Insertion Operation in Queue

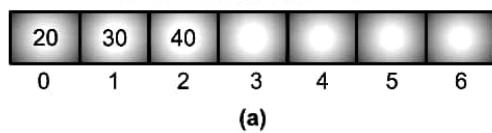
- Whenever, we insert any element in the queue, the value of rear is incremented by one.

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

- When we delete any element is deleted from the queue, front is incremented first. $\text{front} = \text{front} + 1$
- Fig. 4.8 shows queue graphically during deletion operation.

R = 2 and F = 1

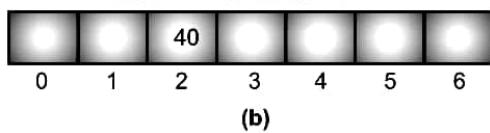
Rear = 2 and Front = -1



(a)

R = 2 and F = 1

Rear = 2 and Front = 1

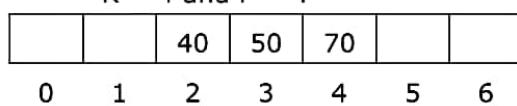


(b)

Fig. 4.8: Deletion Operation in Queue

Now if we insert any element in the queue, the queue will look like,

R = 4 and F = 4



- This is clear from the above figure that whenever an element is removed from the queue, the value of front is incremented by one i.e. $\text{Front} = \text{Front} + 1$

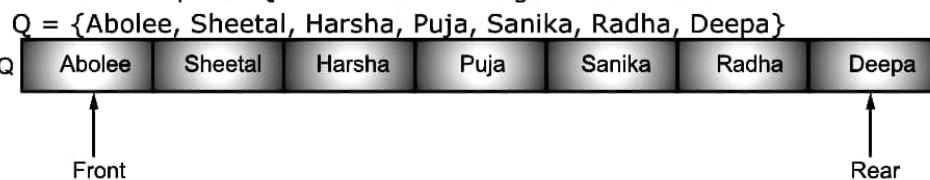
4.7.1 What is Queue

[Oct. 15]

- A queue is a linear list in which data can only be inserted at one end, called the **rear**, and deleted from the other end, called the **front**.
- These restrictions ensure that the data are processed through the queue in the order in which they are received. In simple words, queue is **First In-First Out (FIFO)** structure.
- For a ordered list $L = \{a_1, a_2, a_3, a_4, \dots, a_n\}$ when represented as a queue, a_1 is said to be at the front end and a_n is said to be at the rear end. And also a_{i+1} is said to be behind a_i .



- Let us consider queue Q of students waiting at office counter.

**Fig. 4.9: Queue of Student**

4.8 PRIMITIVE OPERATIONS IN QUEUE

[Oct. 15]

- Queue consist of following operations:
 - Create or Initiative:** Creates a new queue. This operation creates an empty queue. This initialized front and rear.
 - Add or Insert:** Adds an element to the queue. A new element can be added to the queue at the rear.
 - Delete:** Removes an element from the queue. This operation removes the element which is at the front of the queue. This operation can only be performed if the queue is not empty. The result of an illegal attempt to remove an element from an empty queue is called underflow.
 - Is empty:** Checks whether a queue is empty. This operation returns TRUE if the queue is empty and FALSE otherwise.
- In case of static representation of a queue (using array), one more operation is performed `isfull()` which is used to check whether a queue is full or not. This operation returns TRUE if queue is full and FALSE otherwise.

4.9 STATIC AND DYNAMIC REPRESENTATION OF QUEUE

- Queue can be represented using:
 - Static representation, (using array) and
 - Dynamic representation, (using pointers).
 - If queue is implemented using arrays, we must be sure about the exact number of elements we want to store in the queue, because we have to declare the size of the array at design time or before the processing starts.
 - In this case, the beginning of the array will become the front for the queue and the last location of the array will act as rear for the queue.
 - Static queue is defined as,
- ```

#define MAXSIZE 10
typedef struct qnode
{
 int data[MAXSIZE];
 int front, rear;
} STATIC_QUEUE;
STATIC_QUEUE q;

```
- Here `q` is queue which can store maximum 10 numbers.



```
int isfullq(INTQUEUE *q)
{
 if(q->rear==MAX-1)
 {
 if(q->front===-1)
 return 1;
 else
 return 0;
 }
 else
 return 0;
}
void insertq(INTQUEUE *q,int val)
{
 q->rear++;
 q->data[q->rear]=val;
}
int deleteq(INTQUEUE *q)
{
 return(q->data[++(q->front)]);
}
int main()
{
 INTQUEUE q;
 int ch,no,i;
 initq(&q);
 do
 {
 printf("\n1. Insert \n2. Delete \n3. Display \n4. Exit");
 printf("\nEnter ur choice: ");
 scanf("%d",&ch);
 switch(ch)
 {
 case 1: if(!isfullq(&q))
 {
 printf("\nEnter the number: ");
 scanf("%d",&no);
 insertq(&q,no);
 }
 else
 printf("\n QUEUE FULL.");
 break;
 }
 }
}
```



```
case 2: if(!isemptyq(&q))
{
 no=deleteq(&q);
 printf("\n Deleted number is: %d",no);
}
else
 printf("\n QUEUE EMPTY.");
break;
case 3: printf("\n Queue is:");
for(i=q.front+1;i<=q.rear;i++)
 printf("\n%d",q.data[i]);
break;
case 4: exit(0);
}
}while(ch!=4);
```

**Output:**

```
1. Insert
2. Delete
3. Exit
Enter ur choice: 1
Enter the number: 45

1. Insert
2. Delete
3. Exit
Enter ur choice: 1
Enter the number: 23

1. Insert
2. Delete
3. Exit
Enter ur choice: 1
Enter the number: 67

1. Insert
2. Delete
3. Exit
Enter ur choice: 3

Queue is:
45
23
67
```



```
1. Insert
2. Delete
3. Exit
Enter ur choice: 2

Deleted number is: 45
1. Insert
2. Delete
3. Exit
Enter ur choice: 3

Queue is:
23
67
1. Insert
2. Delete
3. Exit
Enter ur choice: 1
Enter the number: 6

1. Insert
2. Delete
3. Exit
Enter ur choice: 3

Queue is:
23
67
6
1. Insert
2. Delete
3. Exit
Enter ur choice: 4
```

- When queue is implemented dynamically, pointers are used.
- Linked list is used to store the queue element. Every node stores the queue element value and a link to a next node.
- Definition for node structure is as follows:

```
typedef struct qnode
{
 int data;
 struct qnode * next;
}DYNAMIC_QUEUE;
DYNAMIC_QUEUE *front, *rear;
```



- Front points to a first node whereas rear points to a last node.
- Initially front and rear stores NULL.
- When a new element is inserted, a node created and appended at the end of list front and rear points to this new node.
- When an element is deleted from list, first node value is returned, front to moved to point next node and first node is deleted.
- When both front and rear points to NULL, the queue is empty.

**Program 4.8:** Dynamic queue implementation.

[Oct. 17]

```
include <stdio.h>
include<string.h>
include<malloc.h>
typedef struct queue
{
 int data;
 struct queue *next;
}INTQUEUE;
void initq(INTQUEUE **front, INTQUEUE **rear)
{
 *front=*rear=NULL;
}
int isemptyq(INTQUEUE **front)
{
 if(*front==NULL)
 return 1;
 return 0;
}
void insertq(INTQUEUE **front, INTQUEUE **rear, int val)
{
 INTQUEUE *temp;
 temp=(INTQUEUE *) malloc(sizeof(INTQUEUE));
 temp->data=val;
 temp->next=NULL;
 if(*rear==NULL)
 {
 *rear=temp;
 *front=temp;
 }
 else
 {
 (*rear)->next=temp;
 *rear=temp;
 }
}
```



```
int deleteq(INTQUEUE **front, INTQUEUE **rear)
{
 int n;
 INTQUEUE *t;
 t=*front;
 n=t->data;
 *front=t->next;

 t->next=NULL;
 free(t);
 if(*front==NULL)
 *rear=NULL;
 return(n);
}

void display(INTQUEUE **front, INTQUEUE **rear)
{
 INTQUEUE *temp;
 temp=*front;
 printf("\nQueue is: ");
 while(temp!=*rear)
 {
 printf("\n%d",temp->data);
 temp=temp->next;
 }
 printf("\n%d",temp->data);
}

int main()
{
 INTQUEUE *front,*rear;
 int ch,no;
 initq(&front,&rear);
 do
 {
 printf("\n1. Insert \n2. Delete \n3. Display \n4. Exit");
 printf("\nEnter ur choice: ");
 scanf("%d",&ch);
 switch(ch)
 {
 case 1: printf("\nEnter the number: ");
 scanf("%d",&no);
 insertq(&front,&rear,no);
 break;
 }
 }
}
```



```
case 2: if(!isemptyq(&front))
{
 no=deleteq(&front,&rear);
 printf("\n Deleted number is: %d",no);
}
else
 printf("\n QUEUE EMPTY.");
break;
case 3: if(!isemptyq(&front))
 display(&front,&rear);
else
 printf("\n QUEUE EMPTY.");
break;
case 4: exit(0);
}
}while(ch!=4);
```

**Output:**

```
1. Insert
2. Delete
3. Exit
Enter ur choice: 1
Enter the number: 23

1. Insert
2. Delete
3. Exit
Enter ur choice: 1
Enter the number: 67

1. Insert
2. Delete
3. Exit
Enter ur choice: 1
Enter the number: 12

1. Insert
2. Delete
3. Exit
Enter ur choice: 3

Queue is:
23
67
12
```



```
1. Insert
2. Delete
3. Exit
Enter ur choice: 2

Deleted number is: 23
1. Insert
2. Delete
3. Exit
Enter ur choice: 3

Queue is:
67
12
1. Insert
2. Delete
3. Exit
Enter ur choice: 1
Enter the number: 89

1. Insert
2. Delete
3. Exit
Enter ur choice: 3

Queue is:
67
12
89
1. Insert
2. Delete
3. Exit
Enter ur choice: 4
```

## 4.10 APPLICATION OF QUEUE

[April 15, 17]

- The most useful application of queues is the simulation of a real world situation, so that it is possible to understand what happens in a real world in a particular situation without actually observing its occurrence.
- Queues are also very useful in a time-sharing computer system where many users share the system simultaneously.
- Whenever, the user requests the system to run a particular program, the operating system adds the request at the end of the queue of the jobs waiting to be executed.



- Whenever, the CPU is free, it executes the job which is at the front of the job queue. Similarly, there are queues for sharing I/O devices. Each device maintains its own queues of requests.
- Another useful operation of queues is in the solution of problems involving searching a non-linear collection of states.
- Queue is used for finding a path using **breadth-first-search** of graphs.
  - Round Robin Technique for processor scheduling is implemented using queues.
  - All types of customer service (like railway ticket reservation) center softwares are designed using queues to store customer's information.
  - Printer server routines are designed using queues. A number of users share a printer using printer server (a dedicated computer to which a printer is connected), the printer server then spools all the jobs from all the users, to the server's hard disk in a queue. From here jobs are printed one-by-one according to their number in the queue.

## 4.11 TYPES OF QUEUE

1. Priority queue
2. Circular queue
3. Doubled ended queue (DEQUE)

### 4.11.1 Priority Queue

[Oct. 16]

- Priority queues are useful data structure in simulations, particularly for maintaining a set of future events ordered by time so that we can quickly retrieve what the next thing to happen is.
- They are called "priority" queues because they enable you to retrieve items neither by the insertion time, nor by a key match, but by which item has the highest priority of retrieval.
- We have the following basic priority queue choices:
  1. **Sorted array or list:** In a sorted array, it is very efficient to find and delete the smallest element. However, maintaining sortedness makes the insertion of new elements slow. Sorted arrays are suitable where there will be a few insertion into the priority queue.
  2. **Binary heaps:** This simple, elegant data structure supports both insertion and extraction in  $O(\log n)$  time each. Heaps maintain an implicit binary tree structure in an array, such that the key of the root of any subtree is less than that of all its descendants. Thus, the minimum key is always at the root of the heap. New keys can be inserted by placing them at an open leaf and relocating the elements upwards until it sits at its proper place in the partial order. Binary heaps are the right answer whenever we know an upperbound on the number of items in we priority queue, since we must specify the array size at creation time.



#### 4.11.1.1 Types of Priority Queue

1. **Ascending Priority Queue:** Elements can be inserted arbitrarily but only the smallest elements can be removed.
2. **Descending Priority Queue:** This allows deletion only of the largest element.

#### 4.11.1.2 Elements of the Priority Queue

- The elements need not be numbers or characters but they may be complex structures that are ordered on several fields like Telephone directory listing comprising of last name, first name, address etc.
- The basis of ordering need not be a part of the elements. It is external value on the basis of which ordering is done.

#### 4.11.1.3 Dynamic Implementation of Priority Queue

- A priority queue can be implemented dynamically in two ways:
  1. Maintaining an ordered linked list of elements, and
  2. Maintaining unordered linked list of elements

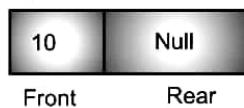
##### 1. Ordered List Implementation:

- In this method the elements are added to the list in such a way that the list is in the sorted order.
- When an element is to be added to the priority queue, it is inserted in its correct position in the queue. For deletion the first element (which is smallest element) is removed.

##### • Example:

1. front = rear = NULL

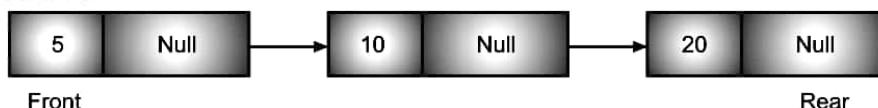
2. Add 10



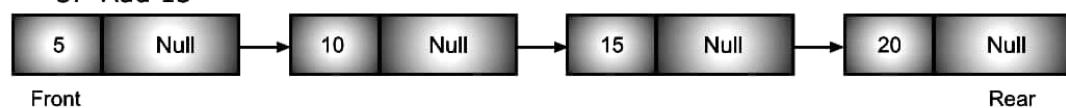
3. Add 5



4. Add 20



5. Add 15

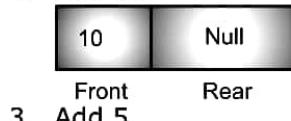




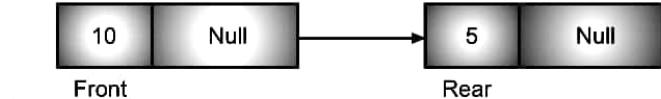
## 2. Unordered List Implementation:

- In this method of implementation, elements are added to the rear of the queue. This is simple insertion process. The insertion is not dependent on sorted order.
- For deletion the entire list has to be examined in order to find the smallest element, which is then removed from the list.
- Example:**

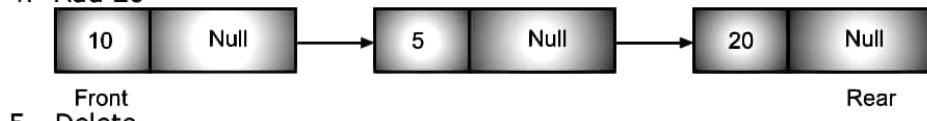
1. front = rear = NULL
2. Add 10



3. Add 5



4. Add 20



5. Delete



- At the time of deletion, the smallest element is deleted from the list.

### 4.11.2 Circular Queue

[Oct. 15]

- When queue is implemented as an array, insertion is not possible after rear reaches the last position of array. There may be vacant positions in the array at the beginning but they cannot be utilized. To overcome this limitation, the concept of circular queue is used.
- In case of circular queue, the array is considered to be logically circular i.e. two ends of it wrap up to make a circle.

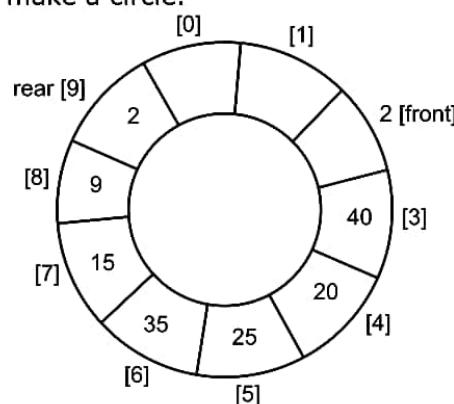


Fig. 4.10: Circular queue

Front = 2, Rear = 9



- When rear reaches to Max-1 i.e. 9, insertion into the queue is made by making rear = 0.
- When front reaches to Max-1, i.e. 9, it is set to 0.
- The circular queue will be empty when front = rear. For e.g. if we delete all elements of above queue one by one, front becomes 9 i.e. equal to rear.
- Similarly, the circular queue will be full when rear becomes equal to front.
- For e.g., when we insert 3 elements into above queue rear becomes 2 which is equal to front.
- Thus, there is conflict. To remove this conflict one way is to allow Max-1 values to be put in the queue i.e. keep one place blank. In this case, queue\_empty condition is same as that of linear queue i.e. front == rear because initially front = rear = -1. But queue-full condition is modified as,  
$$(\text{rear} + 1) \% \text{MAX} == \text{front}$$
- Also front and rear are incremented using following formula,  
$$\text{front} = (\text{front} + 1) \% \text{MAX}$$
  
and  
$$\text{rear} = (\text{rear} + 1) \% \text{MAX}$$
  
Thus front and rear can have values between 0 and MAX-1.

**Program 4.9:** Implementing circular queue using array.

```
include <stdio.h>
define MAX 10
typedef struct queue
{
 int data[MAX];
 int front;
 int rear;
} INTCQUEUE;
void initq(INTCQUEUE *q)
{
 q->front=q->rear=MAX-1;
}
int isemptyq(INTCQUEUE *q)
{
 if(q->front==q->rear)
 return 1;
 return 0;
}
int isfullq(INTCQUEUE *q)
{
 if((q->rear+1)%MAX==q->front)
 return 1;
 return 0;
}
```



```
void insertq(INTCQUEUE *q,int val)
{
 q->rear=(q->rear+1)%MAX;
 q->data[q->rear]=val;
}
int deleteq(INTCQUEUE *q)
{
 q->front=(q->front+1)%MAX;
 return(q->data[q->front]);
}
int main()
{
 int choice;
 INTCQUEUE q;
 int n;
 initq(&q);
 do{
 printf("\n*****MENU*****");
 printf("\n1. Insert \n2. Delete \n3.Exit");
 printf("\nEnter ur choice: ");
 scanf("%d",&choice);
 switch(choice)
 {
 case 1:
 if(!isfullq(&q))
 {
 printf("\nEnter number ");
 scanf("%d",&n);
 insertq(&q,n);
 }
 else
 printf("\n Queue Full.");
 break;
 case 2: if(!isemptyq(&q))
 {
 n=deleteq(&q);
 printf("\n Deleted number is:%d",n);
 }
 else
 printf("\n Queue Empty.");
 break;
 case 3: exit(0);
 }
 }while(choice!=3);
 return 0;
}
```

**Output:**

```
1. Insert
2. Delete
3. Exit
Enter ur choice: 1

Enter number 12

***** MENU *****
1. Insert
2. Delete
3. Exit
Enter ur choice: 23

***** MENU *****
1. Insert
2. Delete
3. Exit
Enter ur choice: 1
Enter number 67

***** MENU *****
1. Insert
2. Delete
3. Exit
Enter ur choice: 45

***** MENU *****
1. Insert
2. Delete
3. Exit
Enter ur choice: 2

Deleted number is: 12
***** MENU *****
1. Insert
2. Delete
3. Exit
Enter ur choice: 2
```



```
Deleted number is: 67
***** MENU *****
1. Insert
2. Delete
3. Exit
Enter ur choice: 2
```

```
Queue Empty
***** MENU *****
1. Insert
2. Delete
3. Exit
Enter ur choice: 3
```

#### 4.11.3 Double Ended Queue

[April 17]

- It is also a homogeneous list of elements in which insertion and deletion operations are performed from both the ends. That is, we can insert or delete elements from the rear end or front end. Hence it is called **Double-Ended Queue**. It is commonly referred to as **deque**.
- There are two types of deques. These two types are due to the restrictions put to perform either the insertions or deletions only at one end.
- Four operations that can be performed on deque are:
  - (i) Insertion at the front end (named as push)
  - (ii) Insertion at the rear end (named as inject)
  - (iii) Deletion from the front end (named as pop)
  - (iv) Deletion from the rear end (named as eject)
- They are:
  1. **Input restricted deques:** Input restricted deques allow insertions at only one end (rear end) of the array or list but deletions allow at both ends. Valid functions in this case are `insert_rearend()`, `delete_frontend()` and `delete_rearend()`.
  2. **Output restricted deques:** Output restricted deques allow deletions at only one end (front end) of the array or list but insertions allow at both ends. Valid functions in this case are `insert_frontend()`, `insert_rearend()` and `delete_frontend()`.



- The variations are shown in Fig. 4.11.

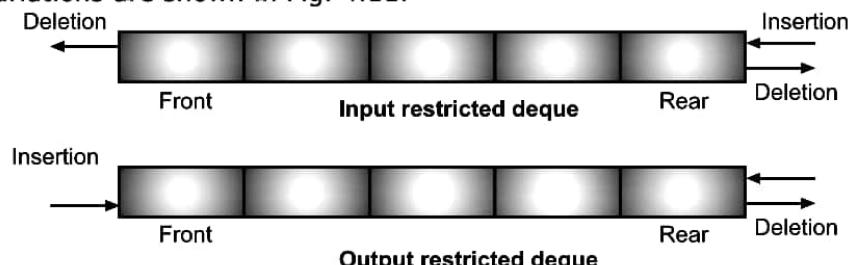


Fig. 4.11: Double Ended Queue

## 4.12 DIFFERENTIATE BETWEEN STACK AND QUEUE

[Oct. 16, 17]

| Stack                                                                                   | Queue                                                                                                               |
|-----------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------|
| 1. Stack works in LIFO manner i.e. element which is inserted last will taken out first. | 1. Queue works in FIFO manner i.e. element which is inserted first will be taken out first.                         |
| 2. In stack, insertions and deletions are made at only end called top                   | 2. In queue, insertions will be done at end called rear and deletions will be done from beginning called front end. |
| 3. Stacks are visualized as vertical collection.                                        | 3. Queues are visualized as horizontal collections.                                                                 |
| 4. Stack operations are called push and pop.                                            | 4. Queue operations are called enqueue and dequeue.                                                                 |
| 5. Stack does not have any types.                                                       | 5. Queue can have different types such as circular queue, priority queue, double-ended queue.                       |

### Practice Questions

- What is meant by stack?
- Define various terms related to stack.
- What is the top of the stack?
- Explain implementation procedure of stack.
- Explain concept of overflow and underflow.
- Explain the push and pop operations.
- What are the applications of stacks?
- Explain importance of polish notations.
- Explain the terms: infix expression, postfix expression.
- Differentiate between static and dynamic implementation of stack.
- Explain role of a stack in calling a sub program.



12. Convert following infix expressions to prefix and postfix format

$$P = (x + y)(a + 7b)^3$$

13. Translate, stepwise following infix expression into its equivalent postfix expression:

$$(A + B \uparrow D) / (E - F) + G$$

$$(A - B) * (D/E)$$

14. Consider the following postfix expression:

(a) 5, 3, +, 2, \*, 6, 9, 7, -, 1, -

(b) 6, 10, +, 12, 8, -, \*, 8, 2, -, 4,  $\uparrow$ , +

Translate each expression into infix notation.

15. Suppose STACK of names is in memory where STACK is allocated  $n = 10$  rooms: Top = 7 STACK: DON, RACHITA, MISTHI, TOM, ALTER, NAFIZ, JEF, DON find the output the following code slices:

(a) Do while (TOP # 0), POP (STACK, Name) (End of loop).

16. Evaluate postfix expression: 5, 6, 2, +, \*, 12, 4, /, -

17. What are advantages and disadvantages of queue?

18. Explain different types of queues.

19. What are the limitations of simple queue?

20. Explain the insertion and deletion operations of a queue.

21. Write an algorithm for insertion and deletion operations performed on circular queue.

22. Differentiate between circular and simple queue.

23. Differentiate between stack and queue.

24. Explain the term of Deque.

25. Write a note on Priority Queue.

26. What is circular queue? How do you represent it?





# Chapter 5...

---

# Trees

---

## Contents ...

- 5.1 Introduction and Definitions
  - 5.1.1 Properties of Trees
  - 5.1.2 Definition of Tree
- 5.2 Tree Terminology
- 5.3 Types of Tree
  - 5.3.1 Binary Tree
  - 5.3.2 Application of Binary Trees
- 5.4 Static and Dynamic Representation of Binary Trees
  - 5.4.1 Array Implementation of Binary Tree (Static Representation)
  - 5.4.2 Linked Implementation of Binary Tree (Dynamic Representation)
  - 5.4.3 Operations on Binary Tree
- 5.5 BST (Binary Search Tree)
- 5.6 Binary Tree Traversal
- 5.7 AVL/High Balanced Tree
  - 5.7.1 Representation of AVL Tree
    - Practice Questions
    - University Questions and Answers

---

## 5.1 INTRODUCTION AND DEFINITIONS

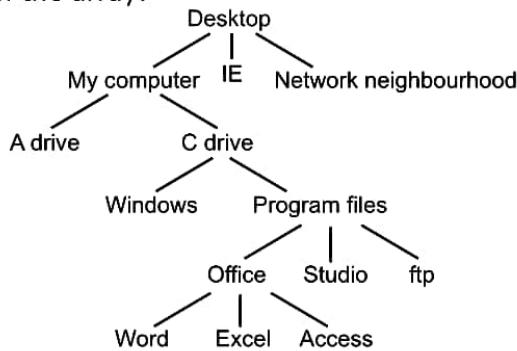
[April 15, Oct. 16]

- Tree, a non-linear data structure which represents hierarchical relationship among its elements.
- Along with information storage, tree as data structure is efficient with respect to operations such as insertion, deletion, searching as compared to linear data structures.
- **For example:** The operating system of a computer organizes files into directories and subdirectories. Directories are also referred as folders. Operating system organizes folders and files using a tree structure. A folder contains other folders (subdirectories) and files. This can be viewed as tree drawn below. The root is desktop.

(5.1)



- There are different ways to represent trees; the first way to representations the nodes as records (structure) are allocated on the heap with pointers to their children or to their parents and second way, the nodes are represented as items in an array. When stored in the array, relationships between them is determined by their positions in the array.



**Fig. 5.1: Tree structure**

- Common operations on trees are:
  - Enumerating all the items;
  - Searching for an item;
  - Adding a new item at a certain position on the tree;
  - Deleting an item;
  - Removing a whole section of a tree (called **pruning**);
  - Adding a whole section to a tree (called **grafting**);
  - Finding the root for any node.
- Common uses for trees are:
  1. To manipulate **hierarchical** data;
  2. To make information easily **searchable**.
  3. To manipulate **sorted lists** of data.

### 5.1.1 Properties of Trees

- Tree is a connected acyclic graph. In many ways a tree is the simplest non-trivial type of graph. It has several 'nice' properties, such as the fact that there exists a unique path between every two vertices.
- The following theorems lists some simple properties of trees. Let T be a tree:
  1. There exists unique path between every two vertices.
  2. The number of vertices is one more than the number of edges in a tree.
  3. A tree with two or more vertices has at least two leaves.



### 5.1.2 Definition of Tree

- A tree is defined recursively, as follow:
  1. A set of zero items is a tree, called the empty tree (or null tree).
  2. If  $T_1, T_2, \dots, T_n$  are  $n$  trees for  $n > 0$  and  $R$  is an item, called a node, then the set  $T$  containing  $R$  and the trees  $T_1, T_2, \dots, T_n$  is a tree. Within  $T$ ,  $R$  is called the root of  $T$  and  $T_1, T_2, \dots, T_n$  are called subtrees.
- The tree in Fig. 5.2 (a) is the empty tree, there are no nodes. The tree in Fig. 5.2 (b) has only one node, the root. The tree in Fig. 5.2 (c) has 16 nodes. The root node has four subtrees. The roots of these subtrees are called the children of the root.
- There are 16 nodes in the tree, so there are 15 non-empty subtrees. The nodes with no subtrees are called terminal node or more commonly, leaves. These are 10 leaves in tree in Fig. 5.2 (c).

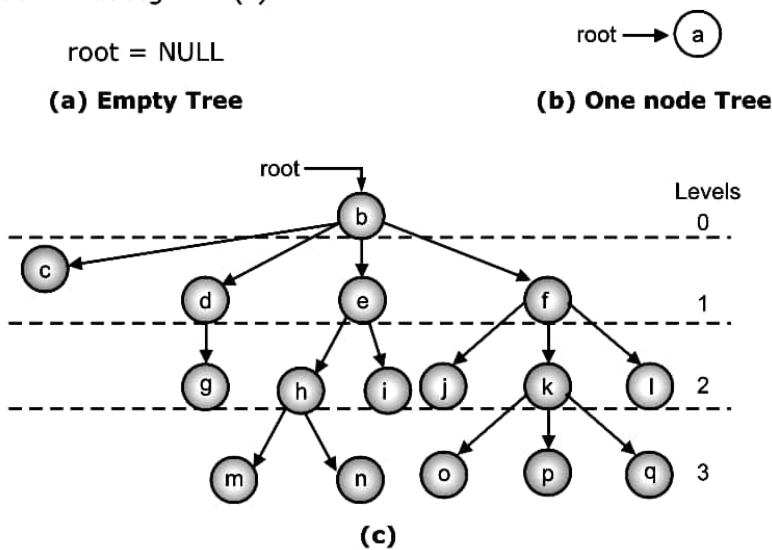
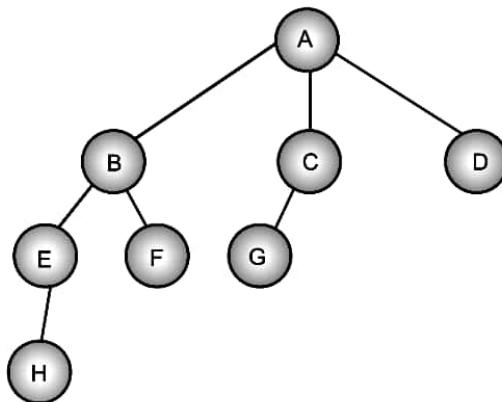


Fig. 5.2: Tree of 10 node

## 5.2 TREE TERMINOLOGY

1. **Root:** It is the mother node of a tree structure. This is the important node of any tree. This node does not have parent. It is the first node in the hierarchical arrangement.
2. **Null Tree:** A tree with no nodes is a Null Tree.
3. **Node:** A node of a tree is an item of information along with the branches to other nodes. The following tree has 8 nodes.

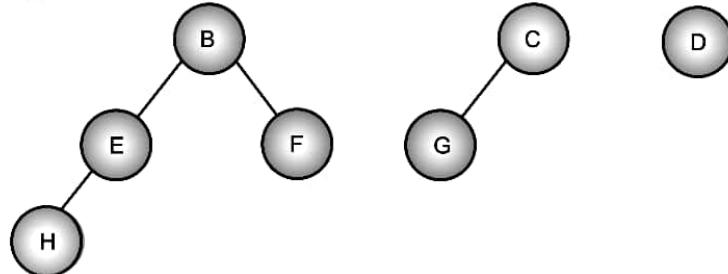


**Fig. 5.3: Tree structure**

4. **Leaf Node:** Leaf node is a terminal node of a tree. It does not have any nodes connected to it. H, F, G, and D are leaf nodes. All other nodes are called non-leaf nodes or internal nodes.
5. **Degree of a node:** The number of sub trees of a node is called its degree. The degree of A is 3, B is 2, D is 0. The degree of the leaf node is zero.
6. **Degree of the Tree:** The degree of a tree is the maximum degree of the nodes in the tree. The degree of the shown tree is 3.
7. **Level:** Level is a rank of tree hierarchy. The whole tree structure is leveled. The level of root node is always at 0. The immediate children of root are at level 1 and their immediate children are at level 2 and so on. The level of above Tree is 3.
8. **Parent Node:** It is a node having other nodes connected to it. These nodes are called the children of that node. The root is the parent of all its sub trees, A, B, and C are parent nodes.
9. **Child Node:** When a predecessor of a node is parent then all successor nodes are called child nodes.
10. **Edge:** Line from node to successor is called edge. For example: line from node A to node B.
11. **Path:** A sequence of consecutive edges is called a path. For example: A, B, E, H or A, C, G.
12. **Height:** The highest number of nodes that is possible in a way starting from the first node (root) to a leaf node is called the height of tree.
13. **Heap:** Heap is a complete binary tree; there are two types of heaps. If the value present at any node is greater than all its children, then the tree is called the max heap or descending heap. In the case of min heap or ascending heap, the value present at any node is smaller than all its children.

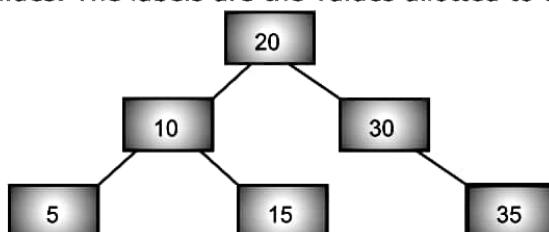


- 14. Forest:** Forest is a set of several trees that are not linked to each other. Forest can be represented as a binary tree, which is built from a forest. Following three trees form the forest if A is removed.



**Fig. 5.4: Forest trees**

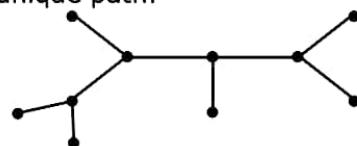
- 15. Descendents:** The descendents of a node are all those nodes which are reachable from that node. For example, E, F, and H are descendents of B.
- 16. Ancestors [April 15]:** The ancestors of a node are all the nodes along the path from the root to that node. B and A are ancestors of E.
- 17. Siblings:** Children of the same parent are called siblings. B, C and D are siblings, E and F are siblings.
- 18. Terminal node:** A node with degree zero is called terminal node or leaf.
- 19. Path length:** It is the number of successive edges from source node to destination node.
- 20. Labeled Trees:** In the labeled tree the nodes are labeled with alphabetic or numerical values. The labels are the values allotted to the nodes in the tree.



**Fig. 5.5: Labeled trees**

### 5.3 TYPES OF TREE

- 1. Free Tree:** A free tree is a connected, a cyclic graph. It is undirected graph. It has no node designated as a root. As it is connected, any node can be reached from any other node by unique path.



**Fig. 5.6: Free Tree**

The tree drawn above is a example of a free tree.



- 2. Rooted Tree:** Unlike free tree, rooted tree is a directed graph in which one node is designated as root, whose incoming degree is zero. And all other nodes incoming degree is one.

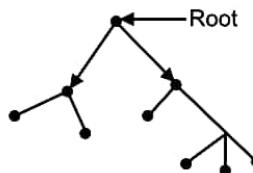


Fig. 5.7: Rooted tree

- 3. Ordered Tree:** In many applications the relative order of the nodes at any particular level assumes some significance. It is easy to impose an order on the nodes at a level by referring to a particular node as the first node, to another node as the second, and so on. Such ordering can be done left to right.

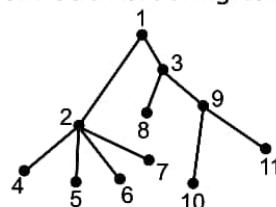


Fig. 5.8: Ordered tree

Just like nodes at each level, we can prescribe order to edges. If in a directed tree, an ordering of node at each level is prescribed, then such a tree is called an ordered tree.

- 4. Positional Trees:** Let us consider m-ary trees in which the m (or fewer) children of any node are assumed to have m distinct positions. If such positions are taken into account, then the tree is called a positional m-ary tree.

In a positional tree, the children of a node are labelled with distinct positive integers. The  $i^{\text{th}}$  child of a node is absent if no child is labelled with integer  $i$ . A m-ary tree is a positional tree in which for every node, all children with labels greater than  $m$  are missing.

- 5. Regular Tree:** A tree in which each branch node vertex has the same out-degree is called as regular tree. If in a directed tree, the outdegree of every node is less than or equal to  $m$ , then the tree is called an m-ary tree. If the outdegree of every node is exactly equal to  $m$  (branch nodes) or zero (leaf nodes) then the tree is called as regular m-ary tree.



**6. Complete Tree:** A tree with  $n$  nodes and depth of  $k$  is complete if its nodes correspond to the nodes which are numbered one to  $n$  in the full tree of depth  $k$ .

A binary tree of height  $h$  is complete if,

- o It is empty or
- o Its left subtree is complete of height  $h-1$  and its right subtree is completely full of height  $h-2$  or
- o Its left subtree is completely full of height  $h-1$  and its right subtree is complete of height  $h-1$ .

A binary tree is completely full if it is of height,  $h$  and has  $(2^{h+1} - 1)$  nodes.

**7. Binary Tree:** A binary tree is a special form of a tree. Binary tree is important and frequently used in various application of computer science.

We have defined  $m$ -ary tree (general tree). For  $m = 2$ , the trees are called binary tree. A binary tree is a  $m$ -ary position tree with  $m = 2$ . In a binary tree no node have more than two children.

**8. Full Binary Tree:** A binary tree is a full binary tree, if it contains maximum possible number of nodes in all levels. Following is a full binary tree of height 2.

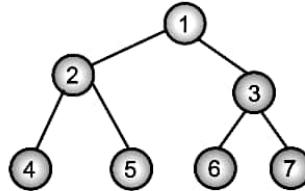


Fig. 5.9: Full binary tree

**9. Complete Binary Tree [April 17]:** A binary tree is said to be complete binary tree, if all its level, except the last level, have maximum number of possible nodes, and all the nodes of the last level appear as far left as possible. Following is a complete binary tree.

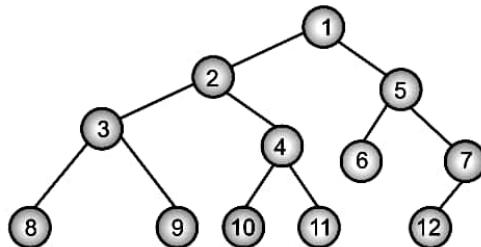


Fig. 5.10: Complete binary tree

**10. Strictly Binary Tree:** A strictly binary tree is a binary tree where all non-leaf nodes have two branches.

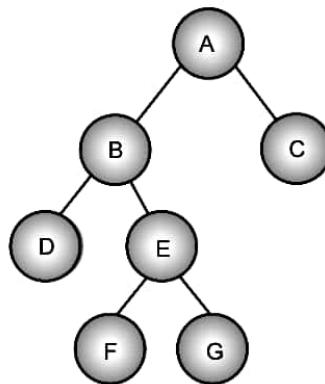


Fig. 5.11: Strictly binary tree

**11. Skewed Binary Tree:** The branches of this tree have either only left branches or only right branches. Accordingly the tree is called left skewed or right skewed tree.

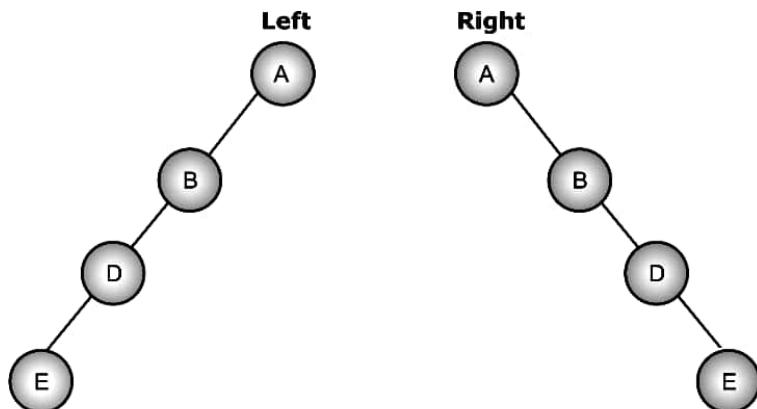


Fig. 5.12: Skewed binary tree

### 5.3.1 Binary Tree

[April 17]

- One of the most commonly used class of tree is the binary tree as shown in Fig. 5.13.

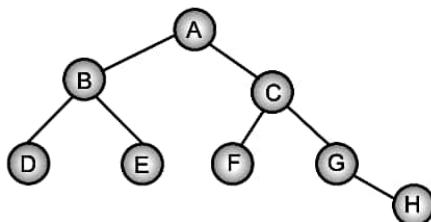
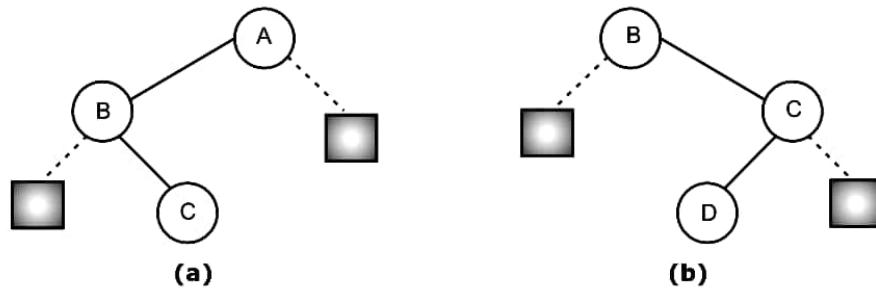


Fig. 5.13: A binary tree

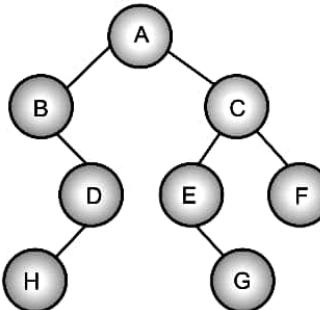
- A binary tree has degree two, each node has at most two children. This makes the implementation of tree easier. Also binary tree has wide range of applications, as we shall study ahead.

- The binary tree is ordered tree. We consider the order of the children significant in a binary tree; hence each child node is either a left node or a right node. Let us formalize these notations in following definition for binary tree.
  - **Definition:** A binary tree is either:
    1. an empty tree; or
    2. consists of a node, called root and two children, left and right, each of which are themselves binary trees.
  - The definition is recursive as we have defined a binary tree in terms of itself. The notion of an empty tree serves as non-recursive part of the definition. Note that in this definition, all the internal nodes of a binary tree are themselves the roots of smaller binary trees.
  - Let us consider the two distinct binary trees in Fig. 5.14. The definition implies that every non-empty node has two children, either of which may be empty. Here the A's right child and B's left child are empty trees (represented by the shaded box).



**Fig. 5.14: Two binary trees**

- Usually, we do not show the empty trees in the binary tree.
  - **Binary Tree** is a special type of tree in which every node or vertex has either no children or one child or two children.
  - A binary tree is an important class of tree data structure in which a node can have at most two children. Child of a node in a binary tree on the left is called "**Left Child**" and the node on the right is called "**Right Child**".



**Fig. 5.15: Binary tree structure**



- A is root node which has two children B and C. Every node in the tree is root of some other sub tree. B and C are sub tree root of node A. E and B has right sub tree rooted as G and D. The H, G and F has no sub trees. They are the leaf nodes in the tree.

### 5.3.2 Applications of Binary Trees

- Binary trees are used to represent non-linear data structure. Binary trees play an important role in software applications. One of the most important applications of binary trees is in the searching algorithms.
- Most efficient and commonly used searching technique known as binary search which uses a special form of binary tree known as **Binary Search Tree**.
- A binary search tree always has two children and the left child node is always lighter than the root node, right child node is always heavier than the root node. Binary search tree brings down the time complexity of algorithm to less than 50%.
- Binary trees may be seen in relational and hierarchical data into the memory. This increases the efficiency of the algorithm which manages this data. Similarly binary trees are used in decisions making, artificial intelligence, compilers, expression evaluation etc.

## 5.4 STATIC AND DYNAMIC REPRESENTATION OF BINARY TREES

- Implementation of binary tree should represent hierarchical relationship between parent node and its left and right children.
- Linked and sequential allocation techniques will be used to represent binary tree structure. The pros and cons of both the allocation techniques we have studied.
- Linked implementation is more popular than the corresponding sequential structure.
- The two main reasons are:
  - A binary tree has a natural implementation in linked storage.
  - The linked structure is more convenient for insertions and deletions.

### 5.4.1 Array Implementation of Binary Tree (Static Representation)

- One of the ways to represent tree using array is to store nodes level by level, starting from the zero level where the root is present. Such representation needs sequential numbering of the nodes, starting with nodes on level zero, then those on level 1 and so on.
- A complete binary tree of height h has  $(2^{h+1} - 1)$  nodes in it. The nodes can be stored in one dimensional array, TREE, with the node numbered at location TREE(i). An array of size  $2^{h+1} - 1$  is needed for the same.
- The root node is stored in the first memory location as the first element in the array.



- Following rules can be used to decide the location of any  $i^{\text{th}}$  node of a tree:
  - For any node with index  $i$ ,  $1 \leq i \leq n$ ,
    - $\text{PARENT}(i) = \lfloor i/2 \rfloor$  if  $i \neq 1$ .  
If  $i = 1$  then it is root which has no parent.
    - $\text{LCHILD}(i) = 2*i$  if  $2i \leq n$ .  
If  $2i > n$ , then  $i$  has no left child.
    - $\text{RCHILD}(i) = 2i + 1$  if  $2i + 1 \leq n$ .  
If  $(2i + 1) > n$ , then  $i$  has no right child.
- Let us, consider following complete binary tree.

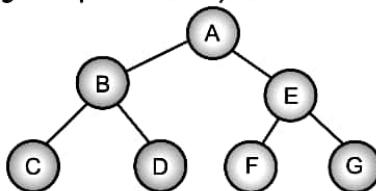


Fig. 5.16: Full binary tree

- The representation of the above binary tree using array is as follows:

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| A | B | E | C | D | F | G | - | - |

- Let us consider one more example.

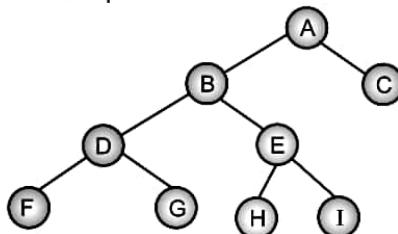


Fig. 5.17: Strictly binary tree

- Now array representation of above tree is as follows:

| Level | 0 | 1 | 2 | 3 |   |   |   |   |   |    |    |    |    |    |    |   |   |   |    |  |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---|---|---|----|--|
|       | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |   |   |   | 20 |  |
|       | A | B | C | D | E | - | - | F | G | H  | I  | -  | -  | -  | -  | - | - | - |    |  |

- Let us consider one more example of skewed tree as follows:

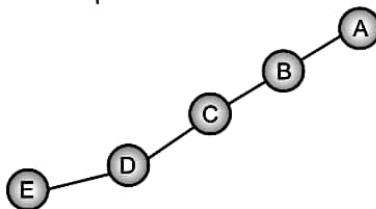


Fig. 5.18: Left skewed tree



- This tree has following array representation:

|   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |         |    |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|---------|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16      | 20 |
| A | B | - | C | - | - | - | D | - | -  | -  | -  | -  | -  | E  | - - - - |    |

- The said representation of binary tree using array seems to be the easiest. Certainly, it can be used for all binary trees. But the method has certain drawbacks. In most of the representations there will be a lot of unused space.
- For complete binary trees the representation is ideal as no space in between the nodes in array is wasted. Certainly the space is wasted as we generally declare array of some arbitrary maximum limit.
- From previous examples we can make out that for the skewed tree, however, less than half of the array is only used and the more is left unused. In the worst case a skewed tree of depth  $k$  will require  $2^{k+1} - 1$  locations of array, and occupying just few of them.
- Major drawback of sequential representation is that data movement of potentially many nodes is needed when insertion or deletion of node occurs. Here the movement of nodes is needed to reflect the change in level number of these nodes.
- These problems can be overcome by use of a linked representation.

#### **Advantage of Binary Trees Array Representation:**

1. Any node can be accessed from any other node by calculating the index.
2. Here, data are stored without any pointers to their successor or ancestor.
3. Programming languages, where dynamic memory allocation is not possible (such as BASIC, FORTRAN), array representation is the only mean to store a tree.

#### **Disadvantages of Binary Tree Array Representation:**

1. Other than full binary trees, majority of the array entries may be empty.
2. It allows only static representation. Array size can not be changed during execution.
3. Inserting a new node to it or deleting a node from it are inefficient with this representation, because these require considerable data, movement up and down the array which demand excessive amount of processing time.

### **5.4.2 Linked Implementation of Binary Trees**

#### **(Dynamic Representation)**

- A binary tree has natural implementation in linked storage.
- In linked organization we wish that all node should be allocated dynamically. Hence, we need each node with data and link fields. Also we need a separate pointer variable to enable us to know where the tree is in memory.
- The most commonly used name to this pointer is root, since it will point to the root of the tree.



- Each node of a binary tree has both a left and a right subtree. Each node will have three fields Lchild, Data and Rchild. Pictorially this node is shown in Fig. 5.19.



Fig. 5.19: Fields of binary tree

- While the node do not provide information about the parent node. But for most of the application this is adequate. If needed, the forth field parent can be included. The binary tree drawn in Fig. 5.20 will have the linked representation as in Fig. 5.21.

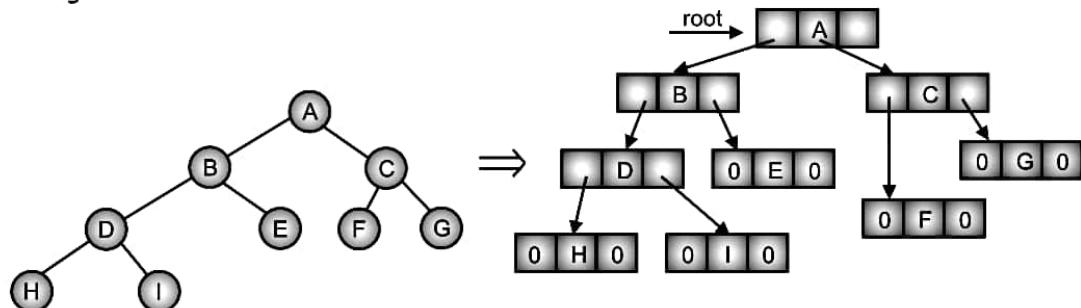


Fig. 5.20: Linked representation of binary tree

- Here, 0 (zero) stored at Lchild or Rchild field represent that the respective child is not present.
- Let us consider one more example,

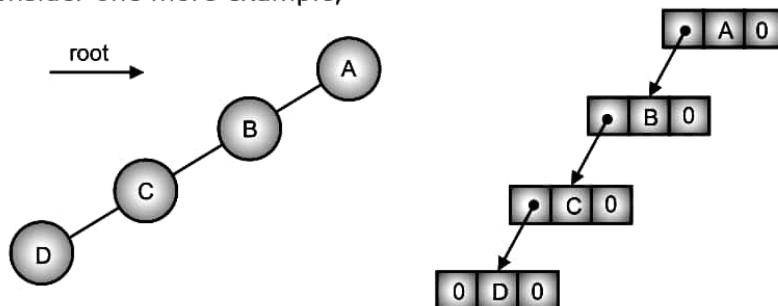


Fig. 5.21: Linked representation of skewed binary tree

- In this node structure Lchild and Rchild are two link fields to store addresses of left child and right child of a node; data is the information content of the node. With this representation, if we know the address of root node (pointer variable 'root') then using it any other node can be accessed.



- **Declaration in 'C':** Each node of binary tree, (as the root of some subtree) has both left and right subtree, which we can access through pointers by declaring as follows:

```
struct treenode
```

```
{
```

```
<data type> Data;
struct treenode *Lchild;
struct treenode *Rchild;
```

```
};
```

`<data type>` is a data type of field Data; the information to be stored.

- For example,

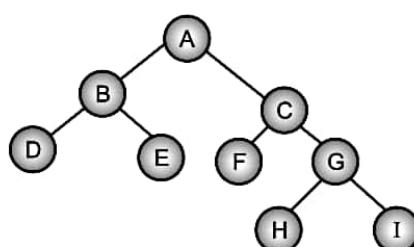
```
typedef struct treenode
```

```
{
```

```
char Data;
TreeNode *Lchild;
TreeNode *Rchild;
```

```
} TreeNode;
```

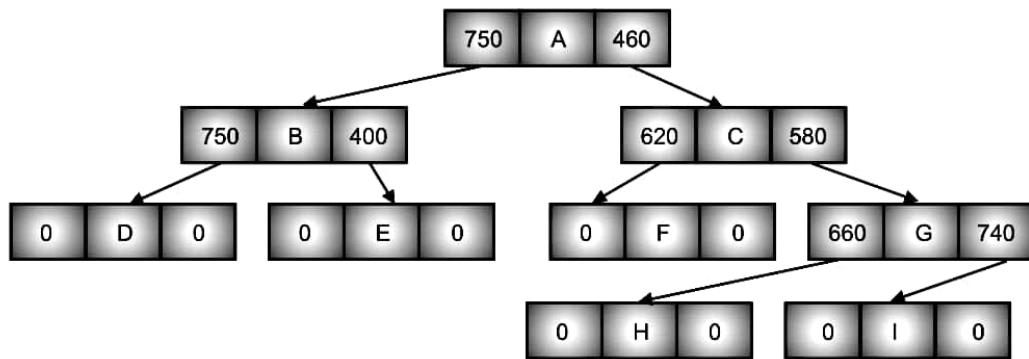
- Using this declaration for linked representation the binary tree representation can be logically viewed as in Fig. 5.22. Physical representation shows the memory allocation of nodes.



(a) A binary view

| Address | Node  |      |        |
|---------|-------|------|--------|
|         | Child | Data | Rchild |
| 500     | 0     | D    | 0      |
| 750     | 500   | B    | 400    |
| 400     | 0     | E    | 0      |
| 890     | 750   | A    | 460    |
| 620     | 0     | F    | 0      |
| 460     | 620   | C    | 580    |
| 660     | 0     | H    | 0      |
| 580     | 660   | G    | 740    |
| 740     | 0     | I    | 0      |

(b) Physical view



(c) Logical view

Fig. 5.22: Physical representation of binary tree

- **Advantages of Linked Binary Tree Representation:**
  1. The drawback of sequential representation are overcome in this representation. We may or may not know the tree depth in advance. Also for unbalanced tree, memory is not wasted.
  2. Insertion and deletion operations are more efficient in this representation.
  3. Useful for dynamic data.
- **Disadvantages of Linked Binary Tree Representation:**
  1. In this representation, there is no direct access to any node. It has to be traversed from root to reach to particular node.
  2. As compared to sequential representation memory needed per node is more. This is due to two link fields (left child and right child for binary trees) in node.
  3. Programming languages not supporting dynamic memory management would not be useful for this representation.

#### 5.4.3 Operations on Binary Tree

- The basic operations on a binary tree can be as listed below:
  1. **Creation:** Creating an empty binary tree to which 'root' points.
  2. **Traversal:** To visit all the node in a binary tree.
  3. **Deletion:** To delete a node from a non-empty binary tree.
  4. **Insertion:** To insert a node into an existing (may be empty) binary tree.
- Few more could be:
  5. **Merge:** To merge two binary trees.
  6. **Copy:** Copy a binary tree.
  7. **Compare:** Compare two binary trees.

**5.5 BST (BINARY SEARCH TREE)**

[April 17, Oct. 17]

- 1. Creating a Binary Search Tree:** A binary tree is a binary tree which is either empty or contains nodes which satisfy the following:

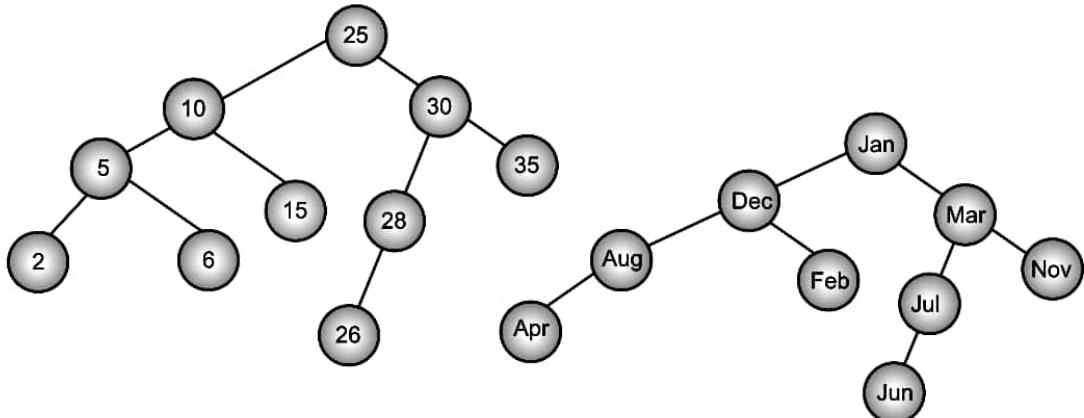
1. All keys in the left sub tree < root
2. All keys in the right sub tree > root
3. The right and left sub trees are also binary search trees.

Definition for BST node is as follows:

```
typedef struct node
{
 int data;
 struct node *left;
 struct node *right;
} BSTNODE;
```

```
BSTNODE *root;
```

- Following are two binary search trees.



**Fig. 5.23: Binary search trees**

**Algorithm to create a binary search tree is:**

1. Initially root=NULL
2. Read a key value and store in node newnode
3. if root==NULL, assign newnode to root  
    goto 6
4. temp=root



5. if key < temp → data  
if temp does not have a left child  
    attach newnode to temp → left  
    goto 6  
else  
    temp=temp → left  
    goto 5  
else  
    if temp does not have right child  
        attach newnode to temp → right  
        goto 6  
    else  
        temp=temp → right  
        goto 5
6. repeat from 2 till all values have been put into the tree
7. stop

```
BSTNODE createBST(BSTNODE *root)
{
 BSTNODE *newnode, *temp;
 char ans;
 do
 {
 newnode=(BSTNODE *) malloc(sizeof(BSTNODE));
 printf("\n Enter the element to be inserted:");
 scanf("%d", &newnode→data);
 newnode→left=newnode→right=NULL;
 if(root==NULL)
 root=newnode;
 else
 {
 temp=root;
 while(temp!=NULL)
 {
 if(newnode→data<temp→data)
 {
 if(temp→left==NULL)
 {
 temp→left=newnode;
 break;
 }
 else
 temp=temp→left;
 }
 }
 }
 }
}
```



```
 else if(newnode->data>temp->data)
 { if(temp->right==NULL)
 {
 temp->right=newnode;
 break;
 }
 else
 temp=temp->right;
 }
 }
 printf("\n Do u want to add more numbers?");
 scanf("%c" & ans);
} while(ans=='y' || ans =='Y')
return(root);
}
```

## 2. Searching an element in a Binary Search Tree Algorithm:

1. Accept key to be search
2. temp=root
3. if temp is NULL or key is found in temp  
    goto 6
4. if key < value in temp  
    temp = temp → left  
    goto 3
5. if key > value in temp  
    temp = temp → right  
    goto 3
6. display temp
7. stop

### • The Non Recursive Function:

```
BSTNODE *search (BSTNODE *root, int key)
{
 BSTNODE *temp=root;
 while (temp != NULL) && (temp → data != NULL)
 {
 if (key < temp → data)
 temp = temp → left;
 else
 temp = temp → right;
 }
 return(temp)
}
```



- **The Search operation can also be written Recursively:**

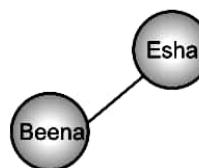
```
BSTNODE *research (BSTNODE *root, int key)
{
 BSTNODE *temp=root;
 if (temp == NULL) || (temp → data == key)
 return(temp);
 else
 if (key < temp → data)
 recsearch (temp → left, key);
 else
 recsearch (temp → right, key);
}
```

### 3. Inserting Data in the Tree:

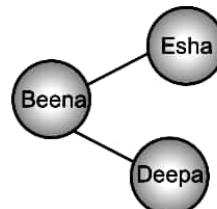
- Let us consider the insertion of the keys: Esha, Beena, Deepa, Gilda, Amit, Geeta, Chetan, into an initially empty tree in given order.



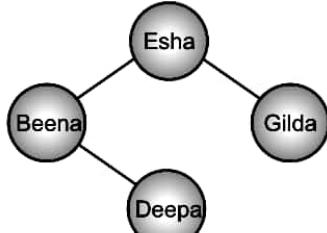
(a) Insert Esha



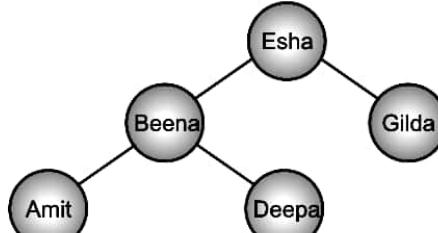
(b) Insert Beena



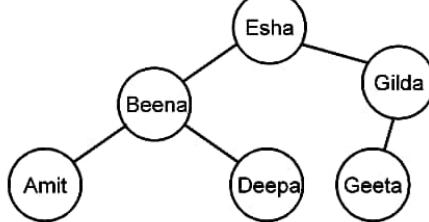
(c) Insert Deepa



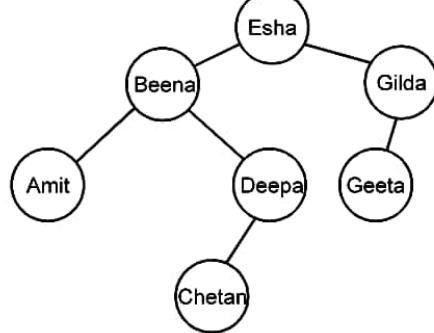
(d) Insert Gilda



(e) Insert Amit



(f) Insert Geeta



(g) Insert Chetan

**Fig. 5.24: Insertion in BST**



- When the first entry, Esha, is inserted, it becomes the root, as shown in Fig. 5.24 (a). Since, Beena is before Esha, insertion goes into left subtree of Esha and so on for all keys.
- When the first node is inserted into an empty tree, is easy, but we need to make root point to the new node. If the tree is not empty, then we must compare the key with one in the root and take decision appropriately. Similar, to Search-BST, Insert-BST function can also be written both recursively as well as non-recursively.

- **Recursive:**

```
BSTNODE *Insert-BST (BSTNODE *root, int n)
{
 if (root == NULL)
 {
 root = (BSTNODE *) malloc (sizeof(BSTNODE));
 root → data = n;
 root → left = root → right = NULL;
 }
 else
 if (n < root → data)
 root → left = Insert-BST (root → left, n);
 else
 root → right = Insert-BST (root → right, n);
 return(root);
}
```

- **Non-Recursive:**

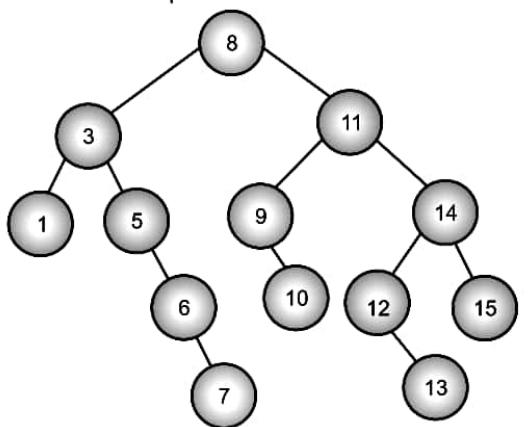
```
BSTNODE *Insert-BST (BSTNODE *root, int n)
{
 BSTNODE *temp, *newnode;
 newnode = (BSTNODE*) malloc (sizeof(BSTNODE));
 newnode → data = n;
 newnode → left = root → right = NULL;
 if (root==NULL)
 root = newnode;
 else
 {
 temp = root;
 while (temp)
 {
 if (n < temp → data) {
```



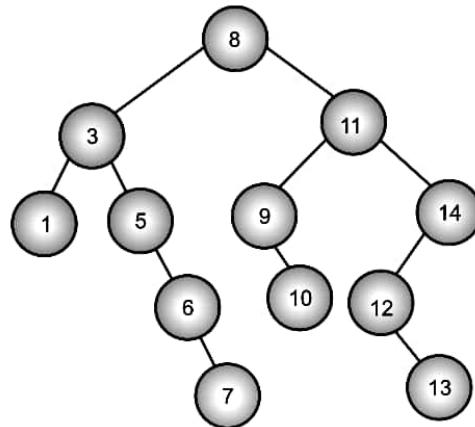
```
 if (temp → left==NULL)
 {
 temp → left = newnode;
 break;
 }
 else
 temp = temp → left;
 else
 if (n > temp → data) {
 if(temp → right==NULL) {
 temp → right = newnode;
 break;
 }
 else
 temp = temp → Rchild;
 }
 }
return root;
}
```

**4. Deleting a node from a Binary Search Tree:**

- (a) A Leaf Node:** In such a case, it is easy to remove it. The corresponding link of the parent node has to be made NULL.



Before delete 15

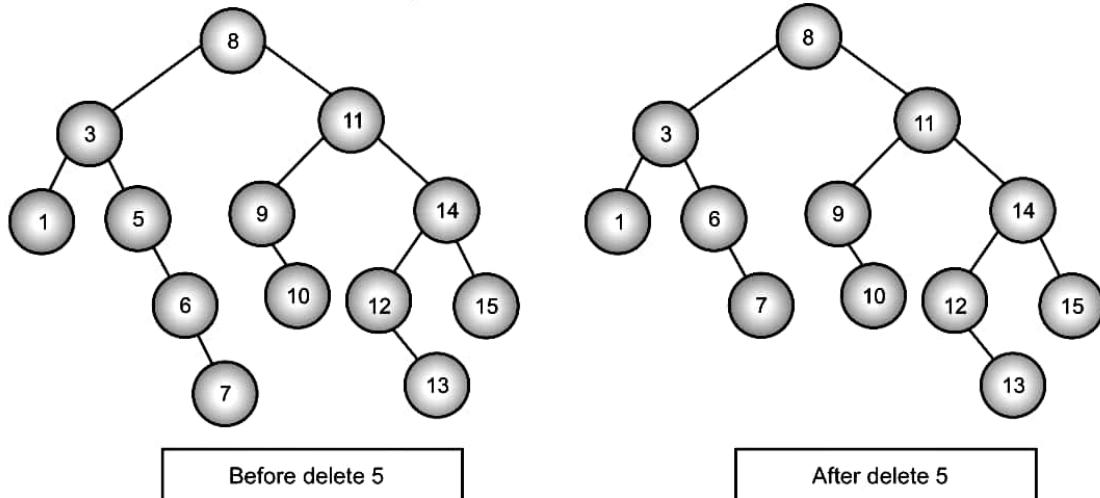


After delete 15

**Fig. 5.25: Deletion of leaf node**

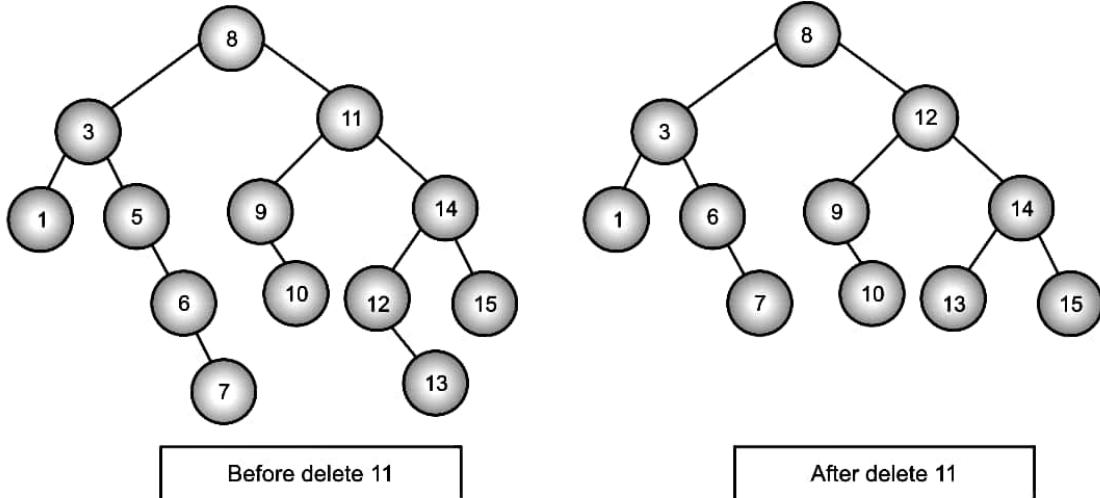


**(b) A Node having one child:** When the node has a single child, the child can be linked to the nodes parent and the node can be removed.



**Fig. 5.26: Deletion of Node having one child**

**(c) A Node having both children:** In this case some rearrangement is needed. One solution is to replace the node with the rightmost node in its left sub tree or with the leftmost node in its right sub tree.



**Fig. 5.27: Deletion of node having both child**

#### Non-recursive function:

```
BSTNODE *non_rec_DeleteBST (BSTNODE *root, int n)
{
 BSTNODE *temp, *parent, *child, *succ, *parsucc;
 temp==root;
 parent=NULL;
```



```
while(temp !=NULL)
{
 if (n == temp → data)
 break;
 parent = temp;
 if(n < temp → data)
 temp = temp → left;
 else
 temp = temp → right;
}
if(temp == NULL)
{
 printf("\nNumber not found.");
 return(root);
}
if(temp→left !=NULL && temp→right !=NULL)
// a node to be deleted has 2 children
{
 parsucc=temp;
 succ=temp→right;
 while(succ→left!=NULL)
 {
 parsucc=succ;
 succ=succ→left;
 }
 temp → data = succ → data;
 temp = succ;
 parent = parsucc;
}
if(temp→left! = NULL) //node to be deleted has left child
 child=temp→left;
else //node to be deleted has right child or no child
 child=temp→right;
if(parent==NULL) //node to be deleted is root node
 root=child;
elseif(temp==parent → left) //node is left child
 parent→left=child;
else //node is right child
 parent→right=child;
free(temp);
return(root);
}
```

**Recursive function for deletion from BST:**

```
BSTNODE *rec_deleteBST(BSTNODE *root, int n)
{
 BSTNODE *temp, *succ;
 if (root==NULL);
 {
 printf("\n Number not found.");
 return(root);
 }
 if(n<root→data) //deleted from left subtree
 root→left=rec_deleteBST(root→left, n);
 else if(n>root→data) //deleted from right subtree
 root→right=rec_deleteBST(root→right, n);
 else //Number to be deleted is found
 {
 if(root→left != NULL && root→right !=NULL)//2 children
 {
 succ=root→right;
 while(succ→left)
 succ=succ→left;
 root→data=succ→data;
 root→right=rec_deleteBST(root→right, succ→data);
 }
 else
 {
 temp=root;
 if(root→left !=NULL) //only left child
 root=root→left;
 elseif (root→right!=NULL) //only right child
 root=root→right;
 else //no child
 root=NULL;
 free(temp);
 }
 }
 return(root);
}
```



### 5. Copying a Tree:

Copying a binary tree creates an exact image of the tree. This can be performed by a recursive method. We will start from root if root is NULL, Null is returned as the copied tree. If it exists a new node is created and it becomes the root of the new tree. If root has a left child, a left child is created for the new tree. The same applies for the right child. The process continues till the entire tree is copied.

#### Function:

```
BSTNODE *teecopy (BSTNODE *root)
{
 BSTNODE *newnode;
 if (root != NULL)
 {
 newnode=(BSTNODE *) malloc (sizeof(BSTNODE));
 newnode -> left = treecopy (root -> left);
 newnode -> right = treecopy (root -> right);
 newnode -> data = root -> left;
 return (newnode);
 }
 else
 return NULL;
}
```

### 6. Comparing Two Binary Search Trees:

In order to check whether two binary trees are the same or not, we shall have to compare each node in one tree to its corresponding node in the other. To do this we will have to use a traversal method so that all nodes in the tree will be visited. Two trees will be identical if their corresponding left and right sub trees are identical. This can be checked in a recursive manner. The function will return true if identical and false otherwise.

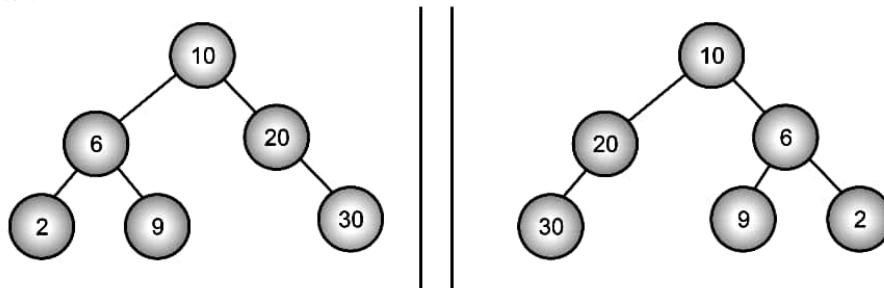
```
int compare(BSTNODE *root1, BSTNODE *root2)
{
 static int equal=0;
 if (root1==NULL && root2==NULL)
 return 1;
 else
 if (root1!=NULL && root2!=NULL)
 if (root1 -> data == root2 -> data)
 if (compare (root1 -> left, root2 -> left))
 equal = compare (root1 -> right, root2 -> right);
 else equal=0;
 return(equal);
}
```



In this function we first compare the data in the root. If it is equal we give a recursive call to the same function for the left sub trees. If the left sub trees are also equal then the right is checked.

### 7. Mirroring a given Tree:

The mirror image of a tree contains its left and right sub trees interchanged as shown.



**Fig. 5.28: Mirroring tree**

In order to mirror a tree, the left and right children of each node have to be interchanged. We will have to begin from the lowest level and move upwards till the children of the root are interchanged.

This can be done by the following recursive function.

```

void mirror (BSTNODE *root)
{
 BSTNODE *temp=root, templ;
 if (temp)
 {
 if (temp → left)
 mirror (temp → left);
 if (temp → right)
 mirror (temp → right);

 /*interchange */
 templ=temp → left;
 temp → left = temp → right;
 temp → right = templ;
 }
}

```

The mirror tree can be displayed by using any tree traversal method.

### 8. Counting the total nodes in a tree:

[Oct. 17]

In order to count the total number of nodes in a tree we have to traverse the tree such that each node is visited. Any one of the traversal methods can be used for this purpose. The function can be written recursively as shown.



```
int countnode(BSTNODE *root)
{
 static int count=0;
 BSTNODE *temp=root;
 if (temp != NULL)
 {
 count++;
 countnode (temp → left);
 countnode (temp → right);
 }
 return count;
}
```

**Program 5.1 Operations on Binary Search Tree**

```
include <stdio.h>
include <malloc.h>
include <stdlib.h>
// stack for BSTNODE pointers
define MAX 50
typedef struct node
{
 int data;
 int flag;
 struct node *lchild;
 struct node *rchild;
}BSTNODE;
// Queue for BSTNODE pointers
typedef struct queue
{
 BSTNODE *data[MAX];
 int front;
 int rear;
}BSTNODE_QUEUE;
void initq(BSTNODE_QUEUE *q)
{
 q→front=q→rear=-1;
}
int isemptyq(BSTNODE_QUEUE *q)
{
 if(q→front==q→rear)
 return 1;
 return 0;
}
```



```
void shiftq(BSTNODE_QUEUE *q)
{
 int to=0,from;
 for(from=q->front+1;from<MAX;from++)
 q->data[to++]=q->data[from];
 q->front=-1;
 q->rear=to-1;
}
int isfullq(BSTNODE_QUEUE *q)
{
 if(q->rear==MAX-1)
 {
 if(q->front== -1)
 return 1;
 else
 {
 shiftq(q);
 return 0;
 }
 }
 else
 return 0;
}
void insertq(BSTNODE_QUEUE *q,BSTNODE *temp)
{
 q->rear++;
 q->data[q->rear]=temp;
}

BSTNODE * deleteq(BSTNODE_QUEUE *q)
{
 return(q->data[++(q->front)]);
}
BSTNODE * rec_insert(BSTNODE *root, int val)
{
 if(root==NULL)
 {
 BSTNODE *newnode;
 newnode=(BSTNODE *) malloc(sizeof(BSTNODE));
 newnode->data=val;
 newnode->flag=0;
 newnode->lchild=newnode->rchild=NULL;
 root = newnode;
 }
}
```



```
 else if(val>root→data)
 root→rchild = rec_insert(root→rchild,val);
 else if (val<root→data)
 root→lchild = rec_insert(root→lchild,val);
 else
 {
 printf("Duplicate nodes are not allowed");
 exit(1);
 }
 return root;
}
BSTNODE * nonrec_insert(BSTNODE *root, int val)
{
 BSTNODE *temp,*temp1;
 temp1=(BSTNODE *) malloc(sizeof(BSTNODE));
 temp1→data=val;
 temp1→lchild=temp1→rchild=NULL;
 if(root==NULL)
 {
 root=temp1;
 return(root);
 }
 else
 {
 temp=root;
 while(1)
 {
 if(val<temp→data)
 {
 if(temp→lchild==NULL)
 {
 temp→lchild=temp1;
 break;
 }
 else
 temp=temp→lchild;
 }
 else if(val>temp→data)
 {
 if(temp→rchild==NULL)
 {
 temp→rchild=temp1;
 break;
 }
 }
 }
 }
}
```



```
 else
 temp=temp->rchild;
 }
 else
 {
 printf("Duplicate nodes are not allowed");
 exit(1);
 }
}
return(root);
}

void rec_inorder(BSTNODE *root)
{
 if(root!=NULL)
 {
 rec_inorder(root->lchild);
 printf("%d ",root->data);
 rec_inorder(root->rchild);
 }
}
BSTNODE * copy_tree(BSTNODE * root)
{
 BSTNODE *newnode;
 if(root!=NULL)
 {
 newnode=(BSTNODE *)malloc(sizeof(BSTNODE));
 newnode->flag=0;
 newnode->lchild=newnode->rchild=NULL;
 newnode->lchild=copy_tree(root->lchild);
 newnode->rchild=copy_tree(root->rchild);
 newnode->data=root->data;
 return(newnode);
 }
 else
 return(NULL);
}
BSTNODE * mirror(BSTNODE *root)
{
 BSTNODE *temp,*temp1;
 temp=root;
```



```
if(temp!=NULL)
{
 if(temp->lchild !=NULL)
 temp->lchild=mirror(temp->lchild);
 if(temp->rchild!=NULL)
 temp->rchild=mirror(temp->rchild);
 temp1=temp->lchild;
 temp->lchild=temp->rchild;
 temp->rchild=temp1;
 return(temp);
}
else
 return(NULL);
}
void levelwise(BSTNODE *root)
{
 int level=0;
 BSTNODE_QUEUE q;
 BSTNODE *temp=root, *dummy=NULL;
 initq(&q);
 insertq(&q,temp);
 insertq(&q,dummy);
 printf("\n level : %d\t",level);
 do
 {
 temp=deleteq(&q);
 if(temp!=dummy)
 printf("%d\t",temp->data);
 if(temp==dummy)
 {
 if(!isemptyq(&q))
 {
 level++;
 printf("\n Level : %d\t",level);
 insertq(&q,dummy);
 }
 }
 else
 {
 if(temp->lchild)
 insertq(&q,temp->lchild);
 }
 }
}
```



```
 if(temp→rchild)
 insertq(&q,temp→rchild);
 }
}while(!isemptyq(&q));
printf("\n The height of the tree is %d",level+1);
}
int cal_height(BSTNODE * root)
{
 if (root==NULL)
 return 0;
 else
 {
 /* compute the depth of each subtree */
 int lDepth = cal_height(root→lchild);
 int rDepth = cal_height(root→rchild);
 /* use the larger one */
 if (lDepth > rDepth)
 return(lDepth+1);
 else return(rDepth+1);
 }
}
int comparetree(BSTNODE *root1,BSTNODE *root2)
{
 static int equal=0;
 if(root1==NULL && root2==NULL)
 return 1;
 else
 if(root1!=NULL && root2 !=NULL)
 {
 if(root1→data==root2→data)
 {
 if(comparetree(root1→lchild,root2→lchild))
 equal=comparetree(root1→rchild,root2→rchild);
 else
 equal=0;
 }
 else
 equal=0;
 }
 return(equal);
}
```



```
int searchBST(BSTNODE *root, int no)
{
 if (root == NULL)
 return (0);
 if (root->data == no)
 return (1);
 if (root->data > no)
 return (searchBST(root->lchild, no));
 else
 return (searchBST(root->rchild, no));
}

int count_total_nodes(BSTNODE *root)
{
 static int cnt=0;
 BSTNODE *temp=root;
 if(temp!=NULL)
 {
 cnt++;
 count_total_nodes(temp->lchild);
 count_total_nodes(temp->rchild);
 }
 return cnt;
}

int count_total_leafnodes(BSTNODE *root)
{
 if(root == NULL)
 return 0;
 if(root->lchild == NULL && root->rchild==NULL)
 return 1;
 else
 return count_total_leafnodes(root->lchild)
 + count_total_leafnodes(root->rchild);
}

int findmin(BSTNODE * root)
{
 while(root->lchild!=NULL)
 root=root->lchild;
 return (root->data);
}
```



```
int findmax(BSTNODE * root)
{
 while(root->rchild!=NULL)
 root=root->rchild;
 return (root->data);
}

//To find address of a node with minimum value
BSTNODE* FindMinnode(BSTNODE *root)
{
 if (root==NULL)
 {
 /* There is no element in the tree */
 return NULL;
 }
 if (root->lchild)
 /* Go to the left sub tree to find the min element */
 return FindMinnode(root->lchild);
 else
 return root;
}

//To find address of a node with maximum value
BSTNODE* FindMaxnode(BSTNODE *root)
{
 if (root==NULL)
 {
 /* There is no element in the tree */
 return NULL;
 }
 if (root->rchild)
 /* Go to the left sub tree to find the min element */
 return(FindMaxnode(root->rchild));
 else
 return root;
}

//Recursive funtion to delete a node from BST
BSTNODE * rec_deleteBST(BSTNODE *root, int val)
{
 BSTNODE *temp;
 if (root==NULL)
 {
 printf("Element Not Found");
 }
```



```
else if(val < root→data)
{
 root→lchild = rec_deleteBST(root→lchild, val);
}
else if(val > root→data)
{
 root→rchild = rec_deleteBST(root→rchild, val);
}
else
{ /* Now We can delete this node and replace with either minimum
 element in the right sub tree or maximum element in the left
 subtree */
 if(root→rchild && root→lchild)
 {
 /* Here we will replace with minimum element in the right
 sub tree */
 temp = FindMinnode(root→rchild);
 root → data = temp→data;
 /* As we replaced it with some other node, we have to
 delete that node */
 root → rchild = rec_deleteBST(root→rchild,temp→data);
 }
 else
 {
 /* If there is only one or zero children then we can
 directly remove it from the tree and connect its parent to
 its child */
 temp = root;
 if(root→lchild == NULL)
 root = root→rchild;
 else if(root→rchild == NULL)
 root = root→lchild;
 free(temp); /* temp is longer required */
 }
}
return root;
}
int main()
{
 BSTNODE *root1=NULL,*root2=NULL;
 int n,no,i,choice;
```



```
do{
 printf("\n1. Create BST");
 printf("\n2. Insert a value in BST using recursive function");
 printf("\n3. Insert a value in BST using non-recursive function");
 printf("\n4. Delete a value from BST");
 printf("\n5. Recursive inorder traversal");
 printf("\n6. Levelwise display of BST");
 printf("\n7. Count total nodes of BST");
 printf("\n8. Count leafnodes of BST");
 printf("\n9. Count height of BST");
 printf("\n10. Copy a BST");
 printf("\n11. Compare two BST");
 printf("\n12. Search value in BST");
 printf("\n13. Find minimum value in BST");
 printf("\n14. Find maximum value in BST");
 printf("\n15. Mirror of BST");
 printf("\n16. Exit");
 printf("\nEnter ur choice :");
 scanf("%d",&choice);
 switch(choice)
 {
 case 1: printf("\nEnter the no. of nodes for tree1 :");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 printf("\nEnter the node value : ");
 scanf("%d",&no);
 root1=nonrec_insert(root1,no);
 // or root1=rec_insert(root1,no);
 }
 break;
 case 2: printf("\nEnter the number to inserted into a BST : ");
 scanf("%d",&no);
 root1=rec_insert(root1,no);
 break;
 case 3: printf("\nEnter the number to inserted into a BST : ");
 scanf("%d",&no);
 root1=nonrec_insert(root1,no);
 break;
 case 4: printf("\nEnter number to delete :");
 scanf("%d",&n);
 root1=rec_deleteBST(root1,n);
 break;
 }
}
```



```
case 5: printf("\nTree is :");
 rec_inorder(root1);
 break;
case 6: printf("\n Levelwise tree is :\n");
 levelwise(root1);
 break;
case 7: printf("\n Total nodes of a BST are :
 %d",count_total_nodes(root1));
 break;
case 8: printf("\n Total leaf nodes of a BST are :
 %d",count_total_leafnodes(root1));
 break;
case 9: printf("\n Height of a BST is :
 %d",cal_height(root1));
 break;
case 10: root2=copy_tree(root1);
 printf("\n Original BST is :\n");
 rec_inorder(root1);
 printf("\n Copy of BST is :\n");
 rec_inorder(root2);
 break;
case 11: printf("\n Enter the no. of nodes for tree2 :");
 scanf("%d",&n);
 for(i=1;i<=n;i++)
 {
 printf("\n Enter the node value : ");
 scanf("%d",&no);
 root2=rec_insert(root2,no);
 }
 printf("\nFirst BST is :\n");
 rec_inorder(root1);
 printf("\nSecond BST is :\n");
 rec_inorder(root2);
 if(comparetree(root1,root2))
 printf("\nTrees are equal.");
 else
 printf("\nTrees are not equal.");
 break;
case 12: printf("\n Enter number to search :");
 scanf("%d",&n);
 if(searchBST(root1,n))
 printf("\n Number is found.");
 else
 printf("\n Number not found.");
 break;
```



```
case 13: printf("\n Minimum value is : %d", findmin(root1));
 break;
case 14: printf("\n Maximum value is : %d", findmax(root1));
 break;
case 15: root1=mirror(root1);
 printf("\n Mirror image of a BST is
 (Mirror image is not a BST) : \n");
 rec_inorder(root1);
 break;
case 16: exit(0);
}
}while(choice!=16);
return 0;
}
```

**Output:**

```
1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit
Enter ur choice :1
Enter the no. of nodes for tree1 :4
Enter the node value : 56
Enter the node value : 23
Enter the node value : 11
Enter the node value : 78
1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
```



```
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit

Enter ur choice :5
Tree is :11 23 56 78
1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit

Enter ur choice :2
Enter the number to inserted into a BST : 44

1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
```



```
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit
Enter ur choice :5
```

```
Tree is :11 23 44 56 78
1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit
```

```
Enter ur choice :4
```

```
Enter number to delete :23
```

```
1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
```



```
14. Find maximum value in BST
```

```
15. Mirror of BST
```

```
16. Exit
```

```
Enter ur choice :5
```

```
Tree is :11 44 56 78
```

```
1. Create BST
```

```
2. Insert a value in BST using recursive function
```

```
3. Insert a value in BST using non-recursive function
```

```
4. Delete a value from BST
```

```
5. Recursive inorder traversal
```

```
6. Levelwise display of BST
```

```
7. Count total nodes of BST
```

```
8. Count leafnodes of BST
```

```
9. Count height of BST
```

```
10. Copy a BST
```

```
11. Compare two BST
```

```
12. Search value in BST
```

```
13. Find minimum value in BST
```

```
14. Find maximum value in BST
```

```
15. Mirror of BST
```

```
16. Exit
```

```
Enter ur choice :6
```

```
Levelwise tree is :
```

```
level : 0 56
```

```
Level : 1 44 78
```

```
Level : 2 11
```

```
The height of the tree is 3
```

```
1. Create BST
```

```
2. Insert a value in BST using recursive function
```

```
3. Insert a value in BST using non-recursive function
```

```
4. Delete a value from BST
```

```
5. Recursive inorder traversal
```

```
6. Levelwise display of BST
```

```
7. Count total nodes of BST
```

```
8. Count leafnodes of BST
```

```
9. Count height of BST
```

```
10. Copy a BST
```

```
11. Compare two BST
```



```
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit
Enter ur choice :7
Total nodes of a BST are : 4

1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit
Enter ur choice :8
Total leaf nodes of a BST are : 2

1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit
Enter ur choice :9
Height of a BST is : 3
```



```
1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit
Enter ur choice :10
```

Original BST is :

11 44 56 78

Copy of BST is :

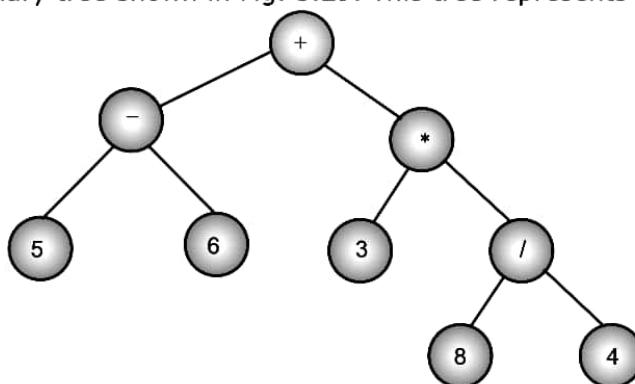
11 44 56 78

```
1. Create BST
2. Insert a value in BST using recursive function
3. Insert a value in BST using non-recursive function
4. Delete a value from BST
5. Recursive inorder traversal
6. Levelwise display of BST
7. Count total nodes of BST
8. Count leafnodes of BST
9. Count height of BST
10. Copy a BST
11. Compare two BST
12. Search value in BST
13. Find minimum value in BST
14. Find maximum value in BST
15. Mirror of BST
16. Exit
Enter ur choice :16
```

## 5.6 BINARY TREE TRAVERSAL

[April 16, 17, Oct. 17]

- Traversal means visiting every node of a binary tree.
  - There are many operations that often are performed on tree such as search a node, print some information, insert a node, delete a node etc.
  - All such operations need traversal through a tree. Hence, traversal is frequently used operation.
  - This operation is used to visit each node (or visit each node exactly once). A full traversal of a tree lists nodes of a tree in a certain linear order. This linear order could be familiar and useful.
  - For example, if the binary tree contains an arithmetic expression then its traversal may give us the expression in infix notation or postfix notation or prefix notation.
  - There are various traversal methods. For a systematic traversal, it is better to visit each node (starting from root) and its both subtrees in same way.
  - In other words, when traversing we want to treat each node and its subtree in the same fashion. **If we let L, D and R stand for moving left, Data and moving right when at node** then there are six possible combinations as: LDR, LRD, DLR, DRL, RDL and RLD.
  - Consider a binary tree shown in Fig. 5.29. This tree represents a binary tree.



**Fig. 5.29: A binary tree representing of an arithmetic expression**

- Let us see the result of each of the six traversals.

$$LDR = 5 - 6 + 3 * 8 / 4$$

LBD: 56-384/\*+

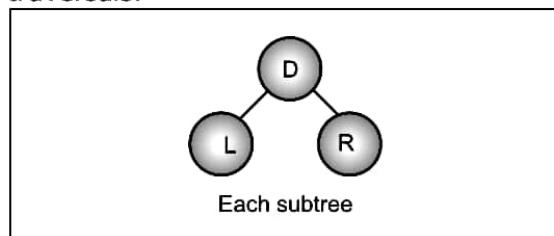
DLR : ± = 5.6 \* 3 / 8.4

DBI : +\*/4.83 - 6.5

BDL : 4 / 8 \* 3 + 6 - E

RDL: 4 / 8 \* 3 + 0 - 5

RLD: 48/3\*65-+





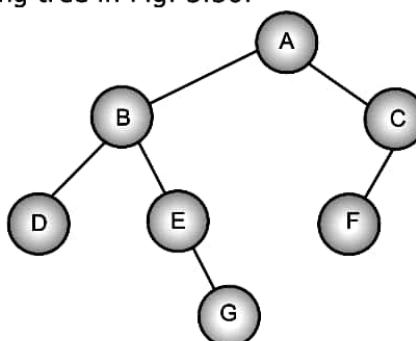
- We can notice that DLR and RLD, LDR and RDL, LRD and DRL are mirror symmetric. If we adopt convention that we traverse left before right then only three traversals are fundamental: LDR, LRD and DLR. These are called as **inorder, postorder and preorder traversals** respectively because there is a natural correspondence between these traversals and producing the infix, postfix and prefix forms of an arithmetic expressions respectively.

- 1. Pre-order Traversal: (DLR):** In this traversal, root is visited first then left subtree in preorder and then right subtree in preorder. It can be defined in following steps.

**Preorder:**

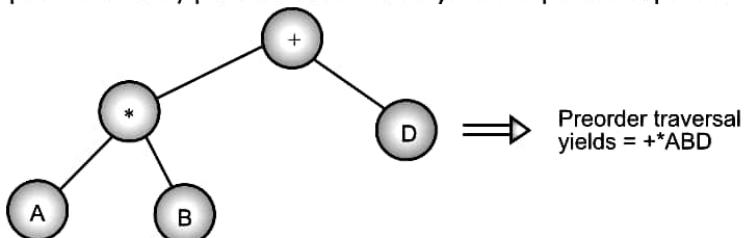
- Visit the root node, say D.
- Traverse the left subtree of node in preorder.
- Traverse the right subtree of node in preorder.

Let us consider following tree in Fig. 5.30.



**Fig. 5.30: Binary tree**

- A preorder traversal of the tree in Fig. 5.30 visits node in a sequence: A B D E G C F. For expression tree, preorder traversal yields a prefix expression.



**Fig. 5.31: Expression binary tree**

- 'C' Function for Preorder Traversal:** Preorder traversal says that 'visit a node, traverse left and continue again. When you cannot continue, move right and begin again or move back, until you can move right and stop'. Let us write a function in C for the same. Preorder function can be written as both recursive and non-recursive.

**Algorithm (recursive):**

```
begin
 if tree not empty
 visit the root
 preorder (left child)
 preorder (right child)
end
```

**Recursive function for Preorder Traversal**

```
void rec_preorder (BSTNODE *root)
{
 if(root)
 {
 printf("%d \t", root→data);
 preorder(root→left);
 preorder(root→right);
 }
}
```

**Non-recursive Algorithm for Preorder Traversal**

1. Start
2. Initialise a stack say 'st' which stores the pointers to a BSTNODE.
3. temp = root
4. Display data stored in temp node.
5. If temp has right child, push its address into the stack.
6. Move temp to left, temp = temp → left.
7. If temp is not NULL, goto step 4.
8. Pop from stack and assign to temp.
9. Continue from step 4 till stack is not empty.
10. Stop.

**Non-recursive function for Preorder Traversal:**

For non-recursive implementation, an explicit stack is used. Stack used to store node addresses is implemented statically as follows:

```
#define MAX 50
BSTNODE *stack[MAX];
int top=-1;
int isempty()
{
 if(top == -1) return(1) else return(0);
}
```



```
int isfull()
{
 if(top==MAX-1)
 return(1);
 else return(0);
}

void push (BSTNODE *item)
{
 if(! isfull())
 {
 top++;
 stack[top]=item;
 }
 else
 {
 printf("Stack overflow");
 exit(0);
 }
}

BSTNODE *pop()
{
 BSTNODE *item;
 if(! isempty())
 {
 item=stack[top];
 top--;
 }
 else
 {
 printf("\n Stack underflow");
 exit(0);
 }
 return(item);
}

void non_rec_preorder(BSTNODE *root)
{
 BSTNODE *temp=root;
 if(root==NULL)
 printf("\n Tree isempty");
```



```
else
{
 push(temp);
 while(! empty())
 {
 temp=pop(); printf("%d \t", temp→data);
 if(temp→right!=NULL) push(temp→right);
 if(temp→left!=NULL) push(temp→left);
 }
}
```

- 2. Inorder Traversal:** In this traversal, the left subtree is visited first in inorder, then root and then the right subtree in inorder.

- It is also called as symmetric traversal.

**Inorder:**

- (a) Traverse the left subtree of root node in inorder.
- (b) Visit the root node.
- (c) Traverse the right subtree of root node in inorder.
- Let us consider the binary tree in Fig. 5.30. A inorder traversal of tree visits node in following sequence.

Sequence: D B E G A F C

- A inorder expression traversal of tree in Fig. 5.31 which is expression tree yields a inorder expression as: A \*B + D.

**Recursive Algorithm:**

```
begin
 if tree is not empty
 inorder (left child)
 visit the root
 inorder (right child).
end
```

- **'C' Function for Inorder Traversal:** Informally the inorder function calls for moving down the tree towards the left until, you can go on farther. Then visit the node, move one node to the right and continue again. If we cannot move, move one node to the right and continue again. If we cannot move to the right go back one more node.



- This can be written as recursive function as:

```
void inorder (BSTNODE *root)
{
 if (root)
 {
 inorder (root → Lchild);
 printf("%d \t", root→data);
 inorder (root → Rchild);
 }
}
```

**Non-recursive Algorithm for Inorder Traversal:**

1. Start
2. Initialise stack say 'st' to store a pointers of BSTNODE.
3. temp=root
4. As long as temp is not NULL
  - (a) push temp in stack
  - (b) move temp to left, temp = temp → left
5. If stack is not empty,
  - (a) pop temp from stack
  - (b) display data of temp
  - (c) Move temp to right, temp = temp → right
6. If stack is not empty goto step 4.
7. Stop.

**Non-recursive function for Inorder Traversal:**

```
void nonrec_inorder (BSTNODE *root)
{
 BTSNODE *temp = root;
 if(temp == NULL)
 printf("\n Tree is empty.");
 else
 {
 do
 {
 while(temp ! = NULL)
 {
 push(temp);
 temp=temp→left;
 }
 }
```



```

 if(! isempty())
 {
 temp=pop();
 printf("%d \t", temp→data);
 temp=temp→right;
 }
} while(! isempty());
}
}

```

**3. Post-order Traversal:** In this traversal, root is visited at the end. Left subtree and then right subtree should be traversed in postorder. This is defined as:

- (i) Traverse the root's left child (subtree) of root node in postorder.
- (ii) Traverse the root's right child (subtree) of root node in postorder.
- (iii) Visit the root 'node'.

Let us consider the binary tree in Fig. 5.30.

The postorder traversals yields the following sequence.

Sequence: D G E B F C A

For the expression tree in Fig. 5.31 the postorder traversal yields a postfix expression as:

= A B \*D +

- **'C' Function for Postorder Traversal:** The postorder traversal say that 'traverse left and continue again. When you cannot continue, move right and begin again or move back, until you can move right and visit the node'.

```

void rec_postorder (BSTNODE *root)
{
 if (root)
 { postorder (root → left);
 postorder (root → right);
 printf ("%c\t", root → data);
 }
}

```

#### Non-recursive Algorithm for Postorder Traversal:

(For this one extra element flag is stored in every node which is initialized to zero)

1. Start
2. Initialise stack say 'st' to stores pointers to the tree node.
3. temp=root
4. Push temp into the stack
5. Move temp to left, temp = temp → left
6. If temp! = NULL, goto step 4
7. temp = pop()



8. If temp's flag is 1
  - (a) display temp's data
  - (b) set temp to NULL
9. Otherwise
  - (a) set temp's flat to 1.
  - (b) push temp into the stack.
  - (c) Move temp to right, temp=temp→right
  - (d) goto step 6
10. If stack is not empty goto step 7
11. Stop

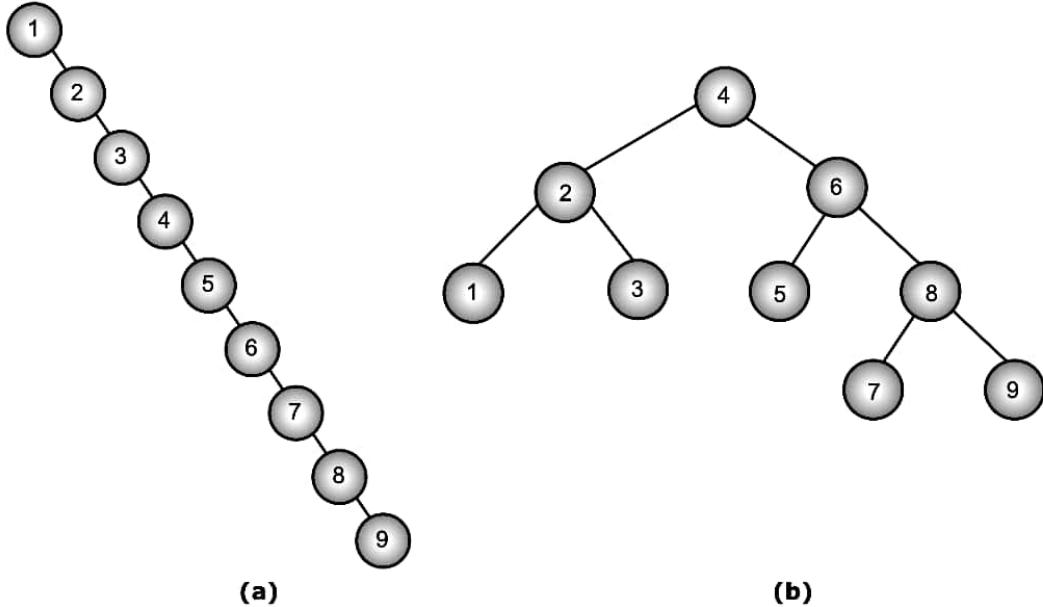
**Non-recursive function for Postorder Traversal:**

```
void nonrec_postorder (BSTNODE *root)
{
 BSTNODE *temp = root;
 if(temp == NULL)
 printf("\n Tree is empty.");
 else
 {
 do
 {
 while(temp != NULL)
 {
 push(temp);
 temp=temp→left;
 }
 temp=pop();
 if(temp→flag==1)
 {
 printf("% d \t", temp→data);
 temp=NULL;
 }
 else
 {
 temp→flag=1;
 temp=temp→right;
 }
 } while(!_isempty());
 }
}
```

## **5.7 AVL / HEIGHT BALANCED TREE**

[Oct. 16, April 16, 17]

- A set of data can produce different binary search trees if the sequence of insertion of elements is different.
  - For example, suppose we have data that consists of numbers from 0 to 9. If we insert the data in sequence {1, 2, 3, 4, 5, 6, 7, 8, 9} then we get a binary search tree shown in Fig. 5.32 (a), while if we insert the data in sequence {4, 2, 6, 1, 3, 5, 8, 7, 9} we get a binary search tree shown in Fig. 5.32 (b).



**Fig. 5.32: AVL tree**

- Average number of comparisons to reach a node for binary search tree (a) is,  
$$(1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9)/9 = \frac{45}{9} = 5.$$
Similarly, average number of comparisons to reach a node for binary search tree (b) is,  $(1 + 2 + 2 + 3 + 3 + 3 + 3 + 4 + 4)/9 = \frac{25}{9} = 2.77.$ Thus, the tree (b) is better than tree (a) from searching point of view.
  - Both the trees have same data but their structure are different because of different sequence of insertion of elements.
  - It is not possible to control the order of insertion, so the concept of height balanced binary search trees came in.
  - One of the more popular balanced trees was introduced in 1962 by **Adelson-Velskii** and **Landis** and was known as AVL Tree. It needs to maintain the binary search tree to be of balanced height.
  - The main aim of AVL tree is to perform efficient search, insertion and deletion operations. Searching is efficient when the height of left and right subtrees of the nodes are almost same.

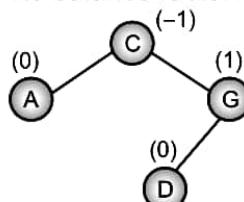
**Definition:**

- An empty binary tree is an AVL tree. A non empty binary tree  $T$  is an AVL tree iff given  $T^L$  and  $T^R$  to be the left and right sub trees of  $T$  and  $h(T^L)$  and  $h(T^R)$  to be the heights of sub trees  $T^L$  and  $T^R$  respectively,  $T^L$  and  $T^R$  are AVL trees and  $|h(T^L) - h(T^R)| \leq 1$ .
- $h(T^L) - h(T^R)$  is known as the **Balance Factor** (BF) and for an AVL tree the balance factor of a node can be either 0, 1 or -1.

**5.7.1 Representation of AVL Search Tree**

[Oct. 17]

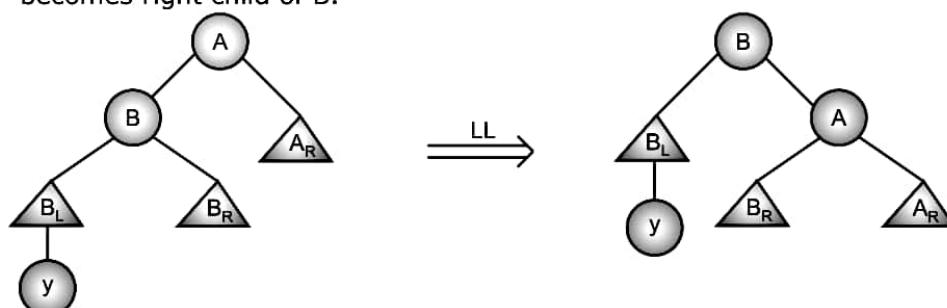
- AVL search trees like binary search trees are represented using a linked representation. However, every node registers its balance factor. The number against each node represents its balance factor.

**Fig. 5.33: AVL tree with BF****1. Insertion and deletion of an AVL search tree:**

- Inserting of an element into an AVL search tree and deletion of AVL tree are similar to that of one used in a binary search tree.
- However, if after insertion of the element or after deleting an element, the balance factor of any node in the tree is affected so as to render the binary search tree unbalanced, we resort to techniques called Rotations to restore the balance of the search tree.
- The node which is nearest to the newly inserted node or deleted node having a balance factor  $\pm 2$  is called the **ancestor node** of that particular subtree. It is denoted by A.
- The rebalancing rotations are classified as LL, LR, RR and RL based on the position of the inserted node with reference to A.

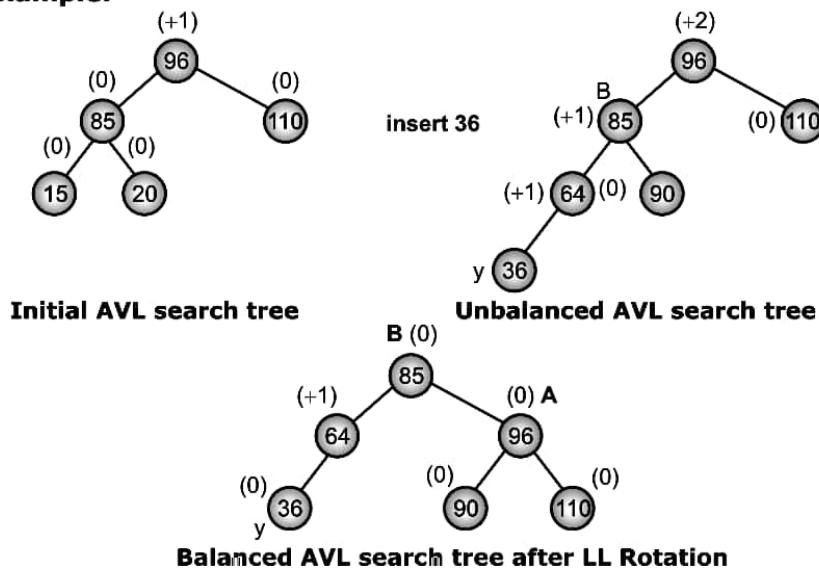
**1. LL Rotation:**

- New node Y is inserted in the left subtree of the left child of A.
- To balance the tree, A's left child B becomes the root of the subtree and A becomes right child of B.

**Fig. 5.34: LL Rotation****Note:**  $\Delta$  represents subtree.



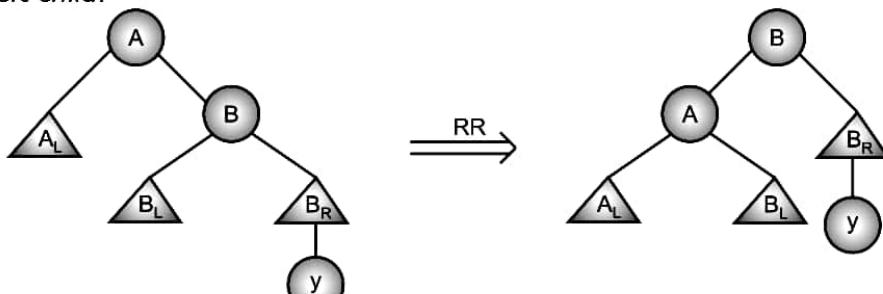
**For example:**



**Fig. 5.35: Insert Item in AVL tree by LL rotation**

## 2. RR Rotation:

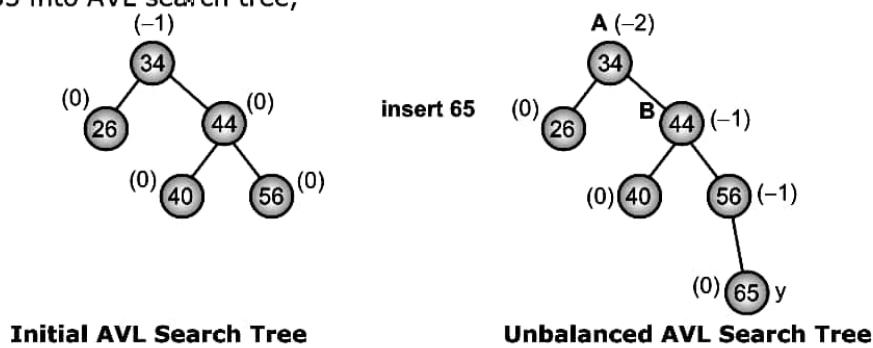
- o New node Y is inserted in the right subtree of the right child of A.
- o The right child of A i.e. B becomes new root of the subtree and A becomes its left child.

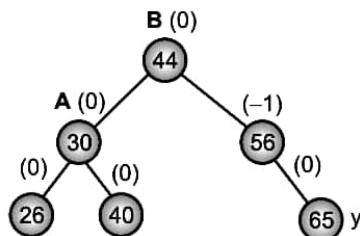


**Fig. 5.36: RR Rotation**

**For example:**

Insert 65 into AVL search tree,





Balanced AVL Search Tree after RR Rotation

Fig. 5.37: Insert item in AVL tree by RR rotation

### 3. LR Rotation:

- New node Y is inserted in the right subtree of the left child of A.
- To balance tree, right child (say c) of left child (say B) of A becomes new root of the subtree and A and B becomes its children.

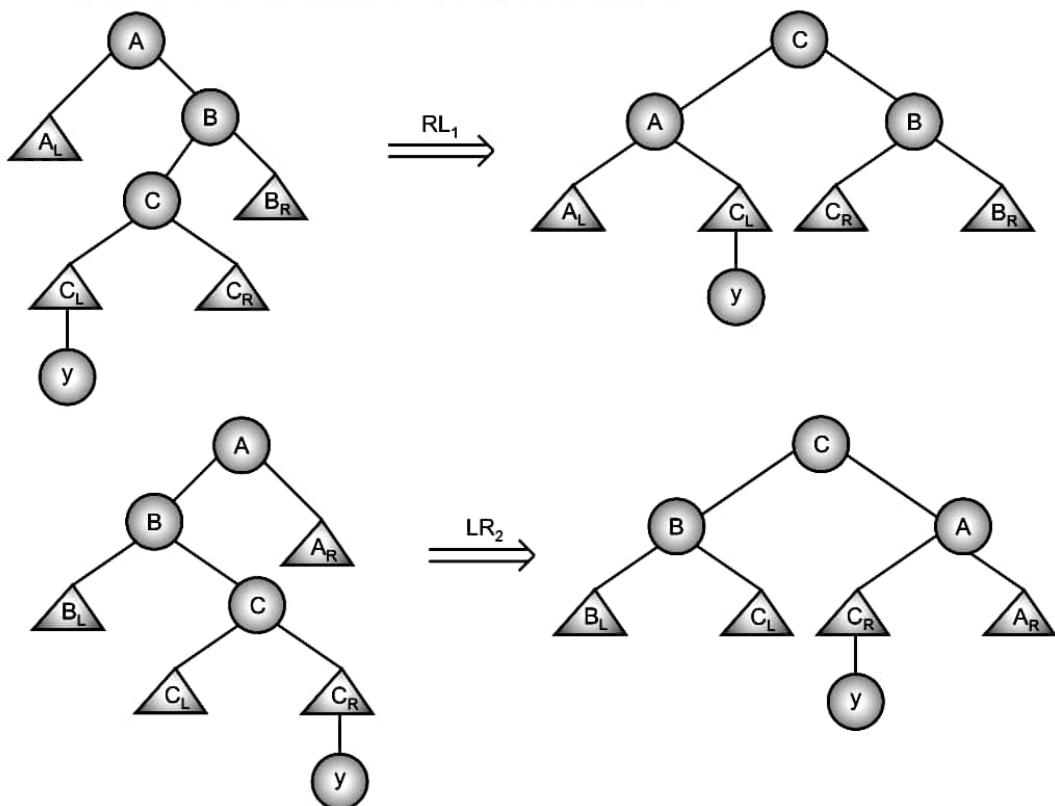


Fig. 5.38: LR Rotation



**For example:**

**Insert 37 into AVL search tree**

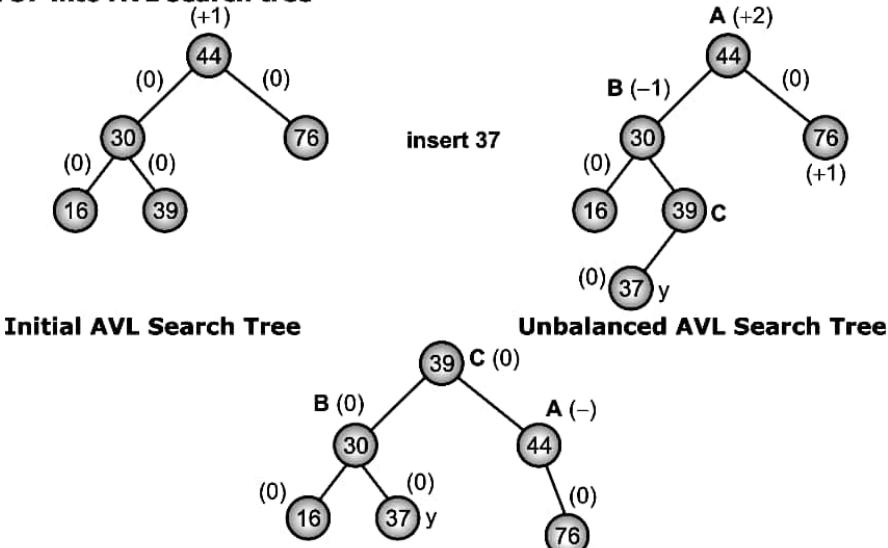


Fig. 5.39: Insert item in AVL tree by LR rotation

#### 4. RL Rotation:

- o New node Y is inserted in the left subtree of the right child of A.
- o To balance the tree, left child (say C) of right child (say B) of A becomes the new root of the subtree and A and B becomes its children.

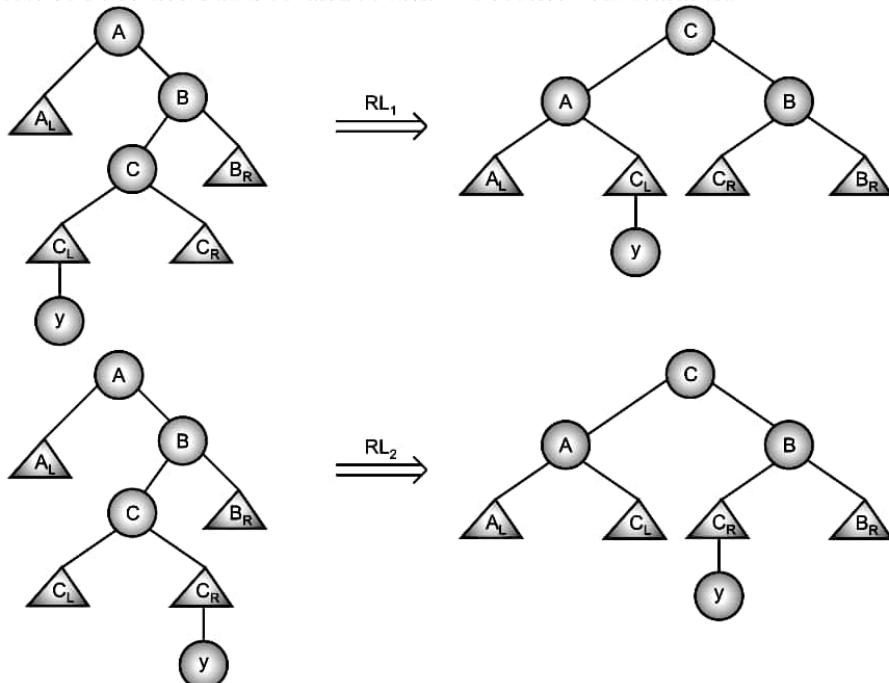
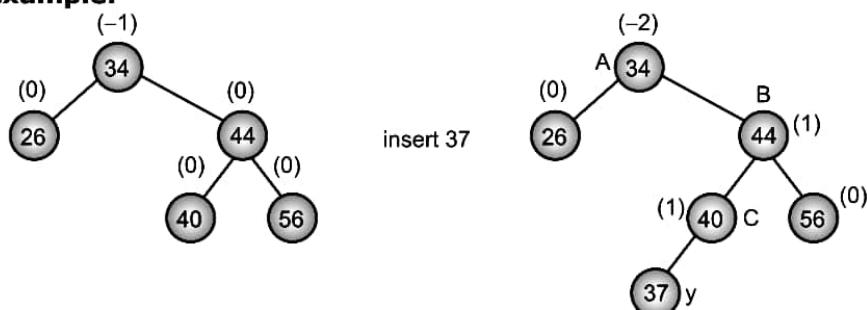


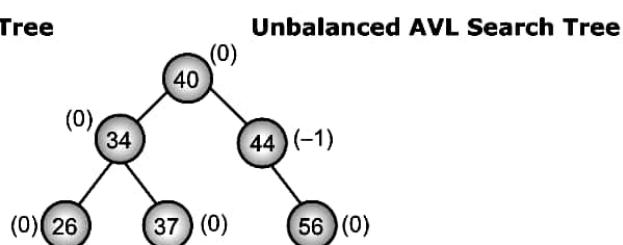
Fig. 5.40: RL rotation



**For example:**



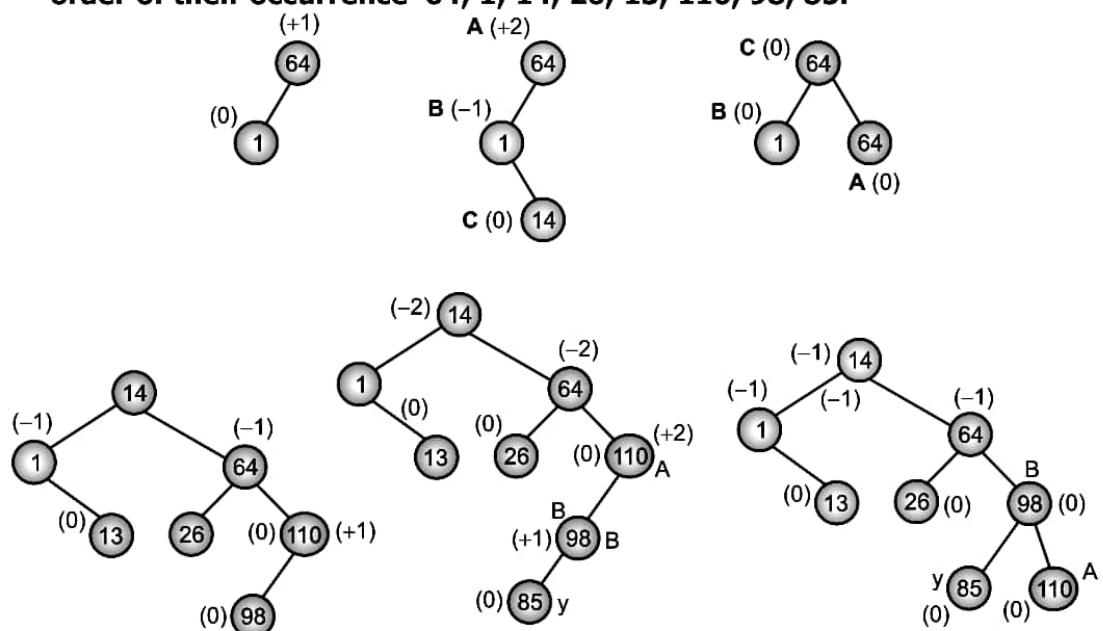
**Initial AVL Search Tree**



**Balanced AVL Search Tree after RL1 Rotation**

**Fig. 5.41**

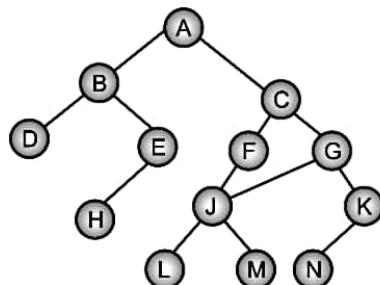
- **Construct an AVL Search Tree by inserting the following elements in the order of their occurrence 64, 1, 14, 26, 13, 110, 98, 85.**



**Fig. 5.42: Insert item in AVL tree by RL rotation**

**Practice Questions**

1. What is a tree?
2. Define Tree?
3. Enlist various types of trees.
4. Explain trees and binary trees.
5. Give the properties of binary tree.
6. Explain the various types of binary trees.
7. Define the following terminologies of tree.
  - (i) Height.
  - (ii) Degree of nodes.
  - (iii) Siblings.
  - (iv) Terminal node.
  - (v) Binary trees?
8. What is meant by traversing? List the methods of node traversing.
9. Define binary search tree. Write a pseudo code for inserting an element in binary search tree.
10. Create a binary search tree of the following data entered as a sequential set  
14, 23, 7, 10, 33, 56, 80, 66, 70.
11. Explain pre-order, in-order and post-order traversal of a tree. Give the pre-order, in-order and post-order listing of the nodes of the following:

**Fig. 5.43**



12. Draw the tree structure for the following expressions.
- (a)  $(2x + y^2 + z)^3 + (3a + 4b + c^2) * (2a + b)$
- (b)  $(a + b + c + d)^3 + (2d + 4e^2 + 5x^3)^4$
- (c)  $(x + 2y + 3z + 4a + 5b + 6c)^4 * (7d + 4z)^2$
13. Write a 'C' Program for preorder traversal of Binary Tree.
14. Write a 'C' Function for inserting a node into Binary Tree.
15. Write a 'C' Program which will satisfy the concept of inorder traversal.
16. Write a function that counts number of nodes in a binary tree.
17. Create a binary search tree for following data:

10, 20, 15, 5, 1, 7, 13

Steps in creation of Binary Search Tree for the data: 10, 20, 15, 5, 1, 7, 13

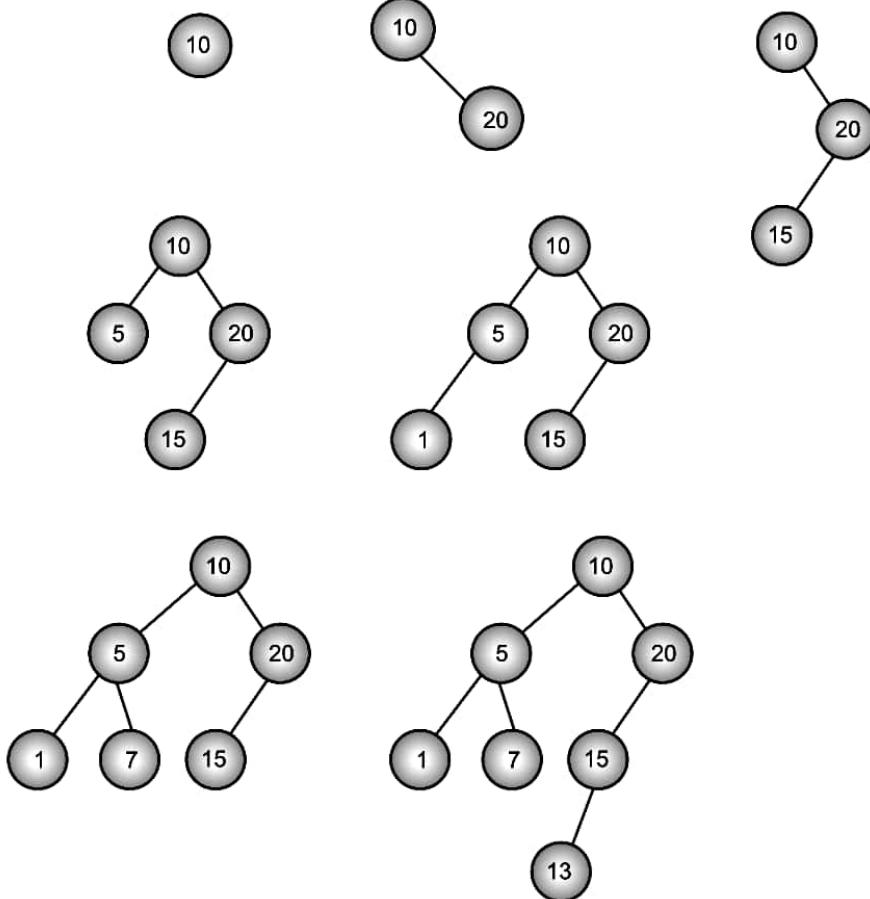


Fig. 5.44



18. Determine the inorder, preorder and postorder traversal of a given binary tree.

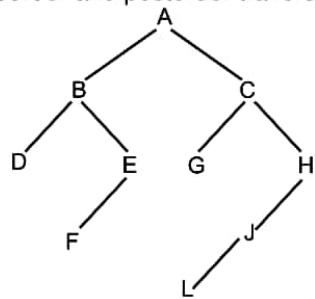


Fig. 5.45





## Chapter 6...

# Graphs

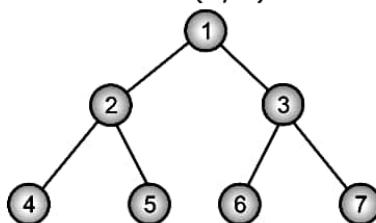
### Contents ...

- 6.1 Introduction
  - 6.1.1 Graph Terminology
- 6.2 Representation of Graphs
  - 6.2.1 Adjacency Matrix (Sequential Representation of Graph)
  - 6.2.2 Adjacency Lists (Dynamic/Linked Representation of Graph)
  - 6.2.3 Inverse Adjacency List
- 6.3 Graph Operations - Traversal
  - 6.3.1 Breadth First Search
  - 6.3.2 Depth First Search
- 6.4 Spanning Trees
  - 6.4.1 Kruskal's Algorithm
  - 6.4.2 Prim's Algorithm
- 6.5 Applications of Graph
  - 6.5.1 Shortest Path
    - Practice Questions
    - University Questions and Answers

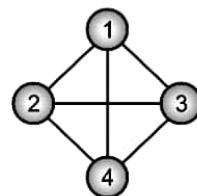
### 6.1 INTRODUCTION

[April 15, 16, 17, Oct. 17]

- A graph  $G$  is a collection of two sets  $V$  and  $E$ .
- A set  $V$  of elements is called nodes or points or vertices.  $V$  is a finite non empty set of vertices.  $E$  is a finite non empty set of edges or arcs connecting a pair of vertices.
- $G$  is represented as  $G = (V, E)$ .



$$\begin{aligned} V(G1) &= \{1, 2, 3, 4, 5, 6, 7\} \\ E(G1) &= \{(1,2), (1,3), (2,4), (2,5), (3,6), (3,7)\} \end{aligned}$$

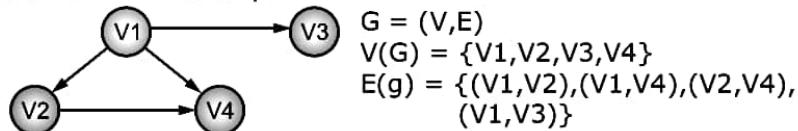


$$\begin{aligned} V(G1) &= \{1, 2, 3, 4\} \\ E(G1) &= \{(1,2), (1,3), (1,4), (2,3), (2,4), (3,4)\} \end{aligned}$$

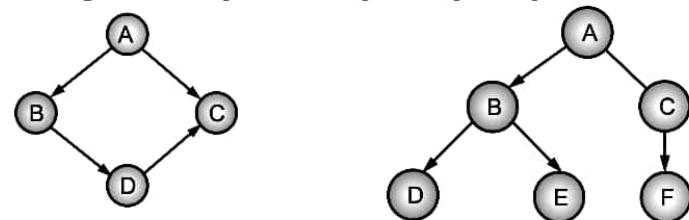
(6.1)



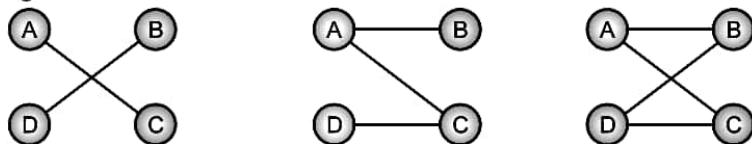
- 1. Directed Graph:** A directed graph G is also called **digraph** which is same as a multigraph except that each edge e in G is assigned a direction or in other words, each edge in G is identified with an ordered pair  $(U,V)$  of nodes in G rather than an unordered pair.



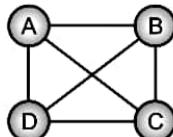
For an edge  $(Vi, Vj)$  in directed graph, vertex  $Vi$  is called the tail and  $Vj$  is the head of the edge.  $Vi$  is adjacent to  $Vj$  and  $Vj$  is adjacent from  $Vi$ .



- 2. Undirected Graph:** An undirected graph G is a graph in which each edge e is not assigned a direction.

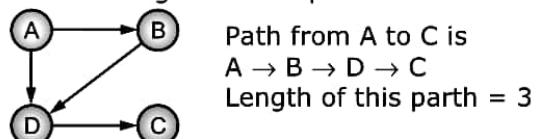


- 3. Connected Graph:** A graph is called connected if there is a path from any vertex to any other vertex.



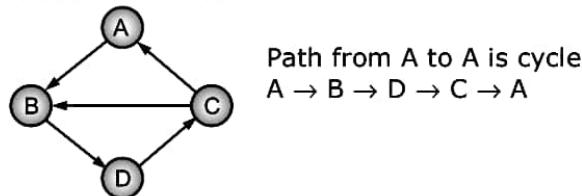
A graph G is connected if and only if there is a simple path between any two nodes in G. A graph G is said to be complete if every node U in G is adjacent to every other node V in G. A complete graph with n nodes had  $n*(n - 1)/2$  edges.

- 4. Path:** A path is a sequence of distinct vertices, each adjacent to the next. The length of such a path is number of edges on that path.

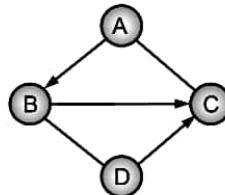




- 5. Cycle:** A path from a node to itself is called a cycle. Thus a cycle is a path in which the initial and final vertices are same.



- 6. Acyclic Graph:** A graph without cycle is called acyclic graph.
- 7. Directed Acyclic Graph (DAG):** A DAG is a graph which is directed containing no cycles.
- 8. Mixed type graph:** The graph in which some edges are directed and some are undirected such a graph is called as a mixed graph.

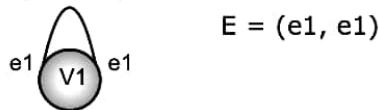


### 6.1.1 Graph Terminology

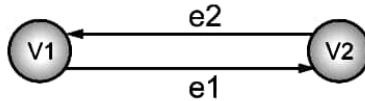
1. **Adjacent Node:** When there is an edge from one node to another then these nodes are called adjacent node.
2. **Incidence:** In an undirected graph the edges ( $e_0, e_1$ ) are incident on nodes. In a direct graph, the edges ( $e_0, e_1$ ) are incident from node  $e_0$  and it is incident to node  $e_1$ .
3. **Length of Path:** Length of path is nothing but total number of edges included in the path from source node to destination node.
4. **Degree of node:** The number of edges connected directly to the node is called as degree of node.
5. **Indegree:** The indegree of a node is the total number of edges coming to that node.
6. **Outdegree:** The outdegree of a node is the total number of edges going outside from the node.
7. **Source:** A node which has only outgoing edges and no incoming edges is called source.
8. **Sink:** A node having only incoming edges and no outgoing edges is called sink node.
9. **Pendant Node:** When indegree of node is one and outdegree is zero then such a node is called pendant node.



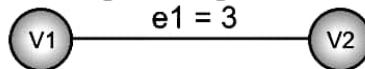
- 10. Reachable:** If a path exists between two nodes, it will be called reachable from one node to other node.
- 11. Articulation Point:** If on removing the node the graph gets disconnected, then that node is called the articulation point.
- 12. Biconnected Graph:** The biconnected graph is the graph which does not contain any articulation point.
- 13. Sling or loop:** An edge of a graph which joins a node to itself is called a sling or loop.



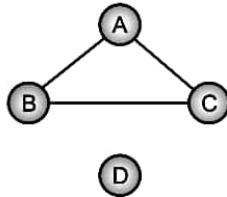
- 14. Parallel Edges:** The two distinct edges between a pair of nodes which are opposite in direction are called as parallel edges.



- 15. Weighted Edges:** An edge which has a numerical value assigned to it is called weighted edge.



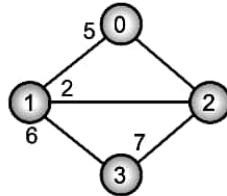
- 16. Isolated Node:** A node which is not an adjacent neighbor to any other node is called an isolated node.



- 17. Isolated Graph:** A graph which has set of empty edges or is containing only isolated nodes is called a NULL graph or isolated graph.



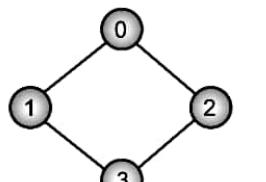
- 18. Weighted Graph:** A weighted graph is a graph in which edges are assigned weights, weights of an edge is also called its cost.



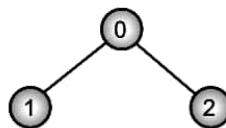


**19. Finite Graph:** A graph which has a finite number of nodes and a finite number of edges is known as finite graph.

**20. Subgraph:** A sub graph of G is a graph  $G'$  such that  $V(G') \subseteq V(G)$  and  $E(G') \subseteq E(G)$ .

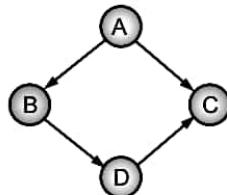


Graph



Subgraph

**21. Directed Acyclic Graph:** It is a diagraph with no cycles.



## 6.2 REPRESENTATION OF GRAPHS

[April 15]

- There are two standard representations of a graph. These are:
  - Sequential, and
  - Linked representation.

### 6.2.1 Adjacency Matrix

#### (Sequential Representation of Graph)

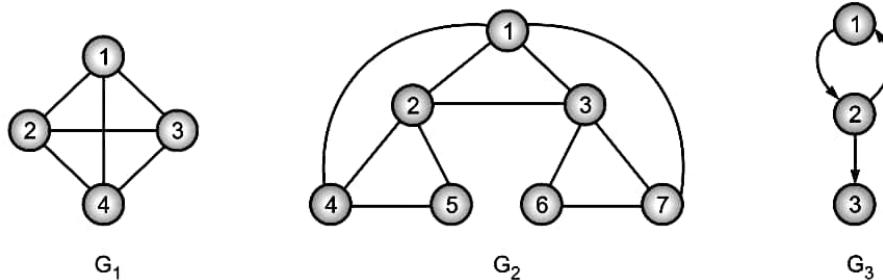
- The graphs when represented using sequential representation using matrices, it is most commonly titled as **Adjacency matrix**.
- One of the most common representations for a graph  $G = (V, E)$  is the adjacency matrix. Suppose,  $V = \{1, 2, \dots, n\}$ . The adjacency matrix for  $G$  is a two dimensional  $n \times n$  matrix  $A$  of booleans. Here,

$$A[i][j] = \begin{cases} 1 & \text{iff the edge } \langle i, j \rangle \text{ exists} \\ 0 & \text{if there exists no edge } \langle i, j \rangle \end{cases}$$

- The adjacency matrix  $A$  has a natural implementation:  
 $A[i][j]$  is 1 (or true) if and only if vertex  $i$  is adjacent to vertex  $j$ . If the graph is undirected then,

$$A[i][j] = A[j][i] = 1$$

- If the graph is directed, we interpret 1 stored at  $A[i][j]$  as indicating that the edge from  $i$  to  $j$  exists and not indicating whether or not the edge from  $j$  to  $i$  exists in graph or not.

**Fig. 6.1: Graph  $G_1$ ,  $G_2$  and  $G_3$** 

- The graphs  $G_1$ ,  $G_2$  and  $G_3$  of Fig. 6.1 are represented using adjacency matrix in Fig. 6.2.

|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 0 | 1 | 1 |
| 3 | 1 | 1 | 0 | 1 |
| 4 | 1 | 1 | 1 | 0 |

$G_1$

|   | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 2 | 1 | 0 | 1 |
| 3 | 0 | 0 | 0 |

$G_3$

|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| 3 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 4 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 6 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 7 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |

$G_2$

**Fig. 6.2: Adjacency matrix for  $G_1$ ,  $G_2$  and  $G_3$  of Fig. 6.1**

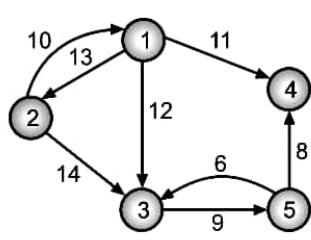
- For weighted graph, the matrix  $A$  is represented as,

$$A[i][j] = \begin{cases} \text{weight} & \text{if the edge } \langle i, j \rangle \text{ exists} \\ 0 & \text{if there exists no edge } \langle i, j \rangle \end{cases}$$

Here, weight is label associated with edge.

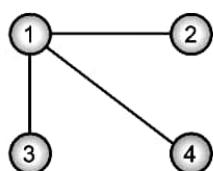


- For example, following is the weighted graph and its associated adjacency matrix.



|   | 1  | 2  | 3  | 4  | 5 |
|---|----|----|----|----|---|
| 1 | 0  | 13 | 12 | 11 | 0 |
| 2 | 10 | 0  | 14 | 0  | 0 |
| 3 | 0  | 0  | 0  | 0  | 9 |
| 4 | 0  | 0  | 0  | 0  | 0 |
| 5 | 0  | 0  | 6  | 8  | 0 |

- We can notice that the adjacency matrix for an undirected graph is symmetric and the adjacency matrix for directed graph need not be symmetric.
- The space needed to represent a graph using adjacency matrix is  $n^2$ , where  $|V| = n$ . When the graph is undirected, we can store only the upper or lower triangle of the matrix, as the matrix is symmetric.
- As we represent edge of graph using adjacency matrix, we can place an edge query such as determine whether a particular edge is in the graph.
- By examining the adjacency matrix, it's easy to check an edge between two vertices by simply looking at the appropriate entry in the matrix.
- Finding all the neighbors of a vertex requires searching a whole row of matrix i.e. it requires  $O(n)$  of time.
- Many of the algorithms need to get number of edges, generating all edges or checking whether, the graph is connected or not. Such queries need to examine  $(n^2 - n)$  entries of matrix as diagonal entries are zero. Hence, we require atleast  $O(n^2)$  of time, even when most of the vertices have few neighbors (that is, when the graph is sparse). Let us consider graph in Fig. 6.3.



|   | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 |
| 2 | 1 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 |
| 4 | 1 | 0 | 0 | 0 |

Fig. 6.3: Graph and its adjacency matrix

#### Program 6.1: Program to Store Graph as Adjacency Matrix

```
Calculating Indegree and Outdegree of node using Adjacency Matrix
#include <stdio.h>
#define MAX 10
/* a function to build an adjacency matrix of the graph*/
void buildadjm(int adj[][MAX], int n)
{
 int i, j;
```



```
for(i=0;i<n;i++)
 for(j=0;j<n;j++)
 {
 printf("Enter 1 if there is an edge from %d to %d,
 otherwise enter 0 \n",i,j);
 scanf("%d",&adj[i][j]);
 }
}
/* Function to compute outdegree of a node*/
int outdegree(int adj[][MAX],int x,int n)
{
 int i, count =0;
 for(i=0;i<n;i++)
 if(adj[x][i] ==1) count++;
 return(count);
}
/* Function to compute indegree of a node*/
int indegree(int adj[][MAX],int x,int n)
{
 int i, count =0;
 for(i=0;i<n;i++)
 if(adj[i][x] ==1) count++;
 return(count);
}
int main()
{
 int adj[MAX][MAX],n,i,j;
 printf("Enter the number of nodes in graph maximum = %d\n",MAX);
 scanf("%d",&n);
 buildadjm(adj,n);
 printf("\nAdjancy matrix is :\n");
 for(i=0;i<n;i++)
 {
 for(j=0;j<n;j++)
 printf("%d\t",adj[i][j]);
 printf("\n");
 }
 printf("\n Indegree of nodes :\n");
 for(i=0;i<n;i++)
 {
 printf("%d -> %d\n",i,indegree(adj,i,n));
 }
}
```



```
printf("\n Outdegree of nodes :\n");
for(i=0;i<n;i++)
{
 printf("%d -> %d\n",i,outdegree(adj,i,n));
}
return(0);
}
```

**Output:**

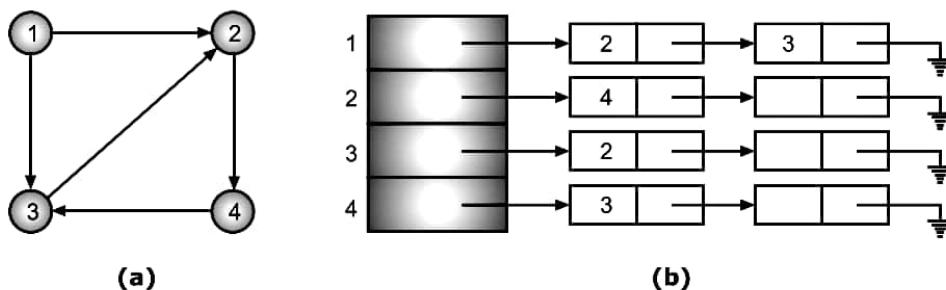
```
Enter the number of nodes in graph maximum = 10
3
Enter 1 if there is an edge from 0 to 0, other enter 0
0
Enter 1 if there is an edge from 0 to 1, other enter 0
1
Enter 1 if there is an edge from 1 to 2, other enter 0
1
Enter 1 if there is an edge from 1 to 0, other enter 0
1
Enter 1 if there is an edge from 1 to 1, other enter 0
0
Enter 1 if there is an edge from 1 to 2, other enter 0
1
Enter 1 if there is an edge from 2 to 0, other enter 0
1
Enter 1 if there is an edge from 2 to 1, other enter 0
1
Enter 1 if there is an edge from 2 to 2, other enter 0
0
Adjancy matrix is :
0 1 1
1 0 1
1 1 0
Indegree of nodes :
0 → 2
1 → 2
2 → 2
Outdegree of nodes :
0 → 2
1 → 2
2 → 2
```



### 6.2.2 Adjacency Lists (Dynamic/Linked Representation of Graph)

[Oct. 16]

- In adjacency matrix of graph in Fig. 6.3, very few entries are non-zero. When we need a list of neighbors of a vertex, we need to transverse a whole row.
- Instead if we keep one list per each vertex, listing all the neighbors for that vertex, a rapid retrieval in time  $O(e + n)$  is possible, where  $e$  is number of edges in graph  $G$  and  $e < (n^2/2)$ . Such a structure which has list for each vertex containing its neighbors, is called as **adjacency list**.
- In this representation, the  $n$  rows of the adjacency list are represented as  $n$  linked lists, one list per vertex of graph.
- The adjacency list for a vertex  $i$  is a list, in some order, of all vertices adjacent to  $i$ . We can represent  $G$  by an array  $\text{Head}$ , where  $\text{Head}[i]$  is a pointer to the adjacency list of vertex  $i$ .
- Each node of the list has atleast two fields: vertex and link. The vertex field contains vertex-id, and link field stores pointer to next node storing another vertex adjacent to  $i$ .
- Fig. 6.4 (b) shows an adjacency list representation for directed graph in Fig. 6.4 (a).



**Fig. 6.4: Graph and its adjacency list representation**

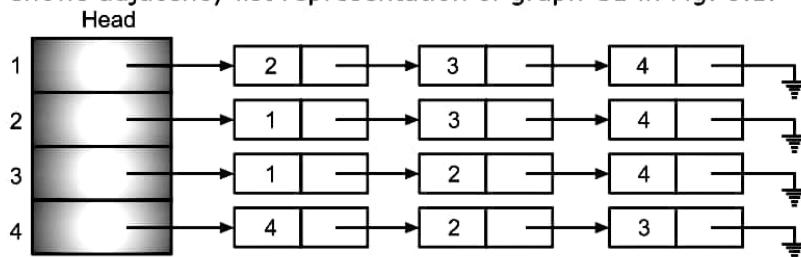
- The graph in Fig. 6.4 (a) is a directed graph. If the graph is a weighted graph, weight field can be added in node structure of list.
- The adjacency list can be declared in C as follows:

```
#define Max_Vertex 20
typedef struct node
{
 int Vertex;
 struct node *Link;
} *GNODE;

GNODE Head[Max_Vertex];
```



- Fig. 6.5 shows adjacency list representation of graph G1 in Fig. 6.1.



**Fig. 6.5: Adjacency list representation of G1 in Fig. 6.1**

- The adjacency list representation of directed graph requires storage proportional to sum of number of vertices plus the number of edges; is often used when the number of edges  $e \ll n^2/2$ . In case of an undirected graph with  $n$  vertices and  $e$  edges, this representation requires  $2e$  list nodes. Both directed and undirected graphs require  $n$  Head Nodes.
- A disadvantage of adjacency list representation is that it may take  $O(n)$  time to determine whether there is an edge from vertex  $i$  to vertex  $j$ , since there can be  $O(n)$  vertices in adjacency list of vertex  $i$ .
- The degree of any vertex in an undirected graph may be determined by counting the number of nodes in its adjacency list. Hence, total edges of  $G$  may be determined in time  $O(n + e)$ .
- The out going degree of any vertex  $i$  may be determined by counting the number of nodes on its adjacency list. To compute incoming degree of vertex, we have to repeatedly access all vertices adjacent to another vertex.
- This is tedious task, hence it is better to keep another set of lists in addition to the adjacency list called as **inverse adjacency lists**.

#### Program 6.2: Program to Store Graph as Adjacency List.

```

Calculating Indegree and Outdegree of node using Adjacency List
#include <stdio.h>
#include <malloc.h>
#define MAX 10
// Node structure of Adjacency List
typedef struct adjnode
{
 int node;
 struct adjnode *next;
} ANODE;

```



```
/* Function to build an adjacency matrix of the graph*/
void buildadjm(int adj[][]MAX], int n)
{
 int i,j;
 for(i=0;i<n;i++)
 for(j=0;j<n;j++)
 {
 printf("Enter 1 if there is an edge from %d to %d,
otherwise enter 0 \n",i,j);
 scanf("%d",&adj[i][j]);
 }
}
// Function to compute outdegree of a node from adjacency list
int outdegreel(ANODE *alist[], int x, int n)
{
 ANODE *temp;
 int cnt=0;
 temp=alist[x];
 while(temp!=NULL)
 {
 cnt++;
 temp=temp->next;
 }
 return(cnt);
}
// Function to compute indegree of a node from adjacency list
int indegreel(ANODE *alist[], int x, int n)
{
 int i,cnt=0;
 ANODE *temp;
 for(i=0;i<n;i++)
 {
 temp=alist[i];
 while(temp!=NULL)
 {
 if(temp->node ==x)
 cnt++;
 temp=temp->next;
 }
 }
 return(cnt);
}
```



```
// Function to convert Adjacency Matrix to Adjacency List
void adjmat_to_adjlist(int adj[][][MAX], int n, ANODE *alist[])
{
 int i, j;
 ANODE *temp, *temp1;
 for(i=0; i<n; i++)
 {
 alist[i]=NULL;
 for(j=0; j<n; j++)
 {
 if(adj[i][j]==1)
 {
 temp=(ANODE *) malloc(sizeof(ANODE));
 temp->node=j;
 temp->next=NULL;
 if(alist[i]==NULL)
 {
 alist[i]=temp;
 temp1=temp;
 }
 else
 {
 temp1->next=temp;
 temp1=temp;
 }
 }
 }
 }
}

int main()
{
 int adj[MAX][MAX], n, i, j;
 ANODE *adjlist[MAX], *temp;
 printf("Enter the number of nodes in graph maximum = %d\n", MAX);
 scanf("%d", &n);
 buildadjm(adj, n);
 printf("\nAdjancy matrix is :\n");
 for(i=0; i<n; i++)
 {
 for(j=0; j<n; j++)
 printf("%d\t", adj[i][j]);
 printf("\n");
 }
 adjmat_to_adjlist(adj, n, adjlist);
 printf("\nAdjancy list is :\n");
}
```



```
for(i=0;i<n;i++)
{
 printf("Node %d ==> ",i);
 for(temp=adjlist[i];temp!=NULL,temp=temp->next)
 printf("%d\t",temp->node);
 printf("\n");
}
for(i=0;i<n;i++)
{
 printf("\n Outdegree of a vertex %d :
 %d",i,outdegreel(adjlist,i,n));
}
for(i=0;i<n;i++)
{
 printf("\n Indegree of a vertex %d :
 %d",i,indegreel(adjlist,i,n));
}
return(0);
}
```

**Output :**

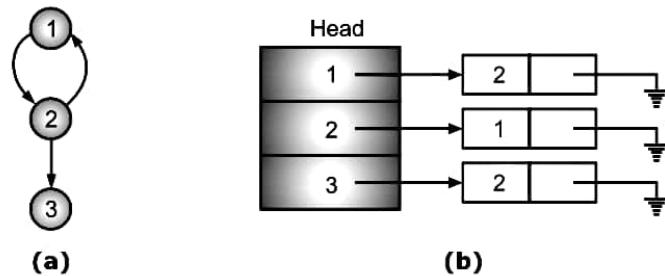
```
Enter the number of nodes in graph maximum = 10
3
Enter 1 if there is an edge from 0 to 0, other enter 0
0
Enter 1 if there is an edge from 0 to 1, other enter 0
1
Enter 1 if there is an edge from 0 to 2, other enter 0
1
Enter 1 if there is an edge from 1 to 0, other enter 0
1
Enter 1 if there is an edge from 1 to 1, other enter 0
0
Enter 1 if there is an edge from 1 to 2, other enter 0
1
Enter 1 if there is an edge from 2 to 0, other enter 0
1
Enter 1 if there is an edge from 2 to 1, other enter 0
1
Enter 1 if there is an edge from 2 to 2, other enter 0
0
Adjancy matrix is :
0 1 1
1 0 1
1 1 0
Adjancy list is :
Node 0 → 1 2
Node 1 → 0 2
Node 2 → 0 1 2
```

```
Outdegree of a vertex 0 : 2
Outdegree of a vertex 1 : 2
Outdegree of a vertex 2 : 3

Indegree of a vertex 0 : 2
Indegree of a vertex 1 : 2
Indegree of a vertex 2 : 3
```

### **6.2.3 Inverse Adjacency List**

- Inverse adjacency lists is a set of lists which contain one list for vertex. Each list contains a node per vertex adjacent to the vertex it represents.
  - The Fig. 6.6 (b) represents inverse adjacency list representation for graph in Fig. 6.6 (a).



**Fig. 6.6: Graph and its inverse adjacency list**

## **Comparison of Sequential and Linked Representation**

- Disadvantage of adjacency matrix representation is that it always requires  $n \times n$  matrix with  $n$  vertices, regardless of the number of edges. If the graph is sparse, many of the entries are null. But as it provides direct access, it is suitable for many applications.
  - Linked representation (Adjacency list) of graph has an advantage of space complexity when graph is sparse, but do not provide direct access.
  - But considering overall performance, matrix representation of graph is more powerful than all.

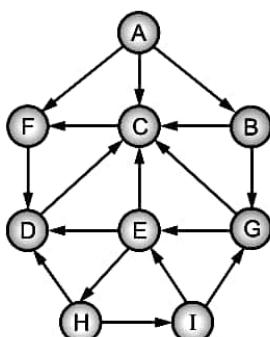
## 6.3 GRAPH OPERATIONS - TRAVERSAL

- In many situations, there is need to visit all the vertices and edges in a systematic fashion, called as graph traversal.
  - There are two techniques for graph traversal:
    1. Breadth First Search (BFS)
    2. Depth First Search (DFS)

### **6.3.1 Breadth First Search**

[Oct. 15, April 15, 16, 17]

- The general idea behind a BFS beginning at a starting node A is as follows:
    - (a) First examine the starting node A.
    - (b) Then examine all the neighbors of A.
    - (c) Then examine all the neighbors of the neighbors of A and so on.
  - Naturally, there is need to keep track of the neighbors of a node and need to guarantee that no node is processed more than once. This is accomplished by using a queue to hold nodes that are waiting to be processed and by using a field STATUS which tells us the current status of any node.
  - The algorithm is as follows: (This algorithm consider that graph is stored using adjacency list).
    1. Initialize all nodes to the ready state (STATUS=1).
    2. Put the starting node A in Queue and change its status to the waiting state (STATUS=2).
    3. Repeat steps 4 and 5 until Queue is empty.
    4. Remove the front node N of Queue. Process N and change the status of N to the processed state (STATUS=3).
    5. Add to the rear of Queue all the neighbors of N that are in the steady state (STATUS=1), and change their status to the waiting state (STATUS=2),  
[end of step 3 loop]
    6. Stop.
  - Consider the given graph G. Suppose starting vertex is A. The BFS traversal is as follows:



**Adjacency List**

- A = B, C, F
- B = C, G
- C = F
- D = C
- E = C, D, H
- F = D
- G = C, E
- H = D, I
- I = E, G



(a) Insert vertex 'A' into the queue.

Queue: A, change status of A to 2.

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

(b) Delete vertex from queue and visit it

Deleted vertex = A, change status of 'A' to 3

Visited vertices = A

Add neighbors (adjacent nodes) of A to queue = B, C, F and change their status to 2

Queue = B C F

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 2 | 2 | 1 | 1 | 2 | 1 | 1 | 1 |

(c) Delete vertex from queue and visit it.

Deleted vertex = B, change status of 'B' to 3.

Visited vertices = A, B

Add neighbors to B to queue having status '1' and change their status to 2 = G.

Queue = C F G

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 3 | 2 | 1 | 1 | 2 | 2 | 1 | 1 |

(d) Delete vertex from queue and visit it.

Deleted vertex = C, change status of 'C' to 3.

Visited vertices = A, B, C

Add neighbors of C to queue having status 1 and change their status to 2 = none.

Queue = F G

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 3 | 3 | 1 | 1 | 2 | 2 | 1 | 1 |

(e) Delete vertex from queue and visit it.

Deleted vertex = F, change status of 'F' to 3.

Visited vertices = A, B, C, F

Add neighbors of F to queue having status and change their status to 2 = D.

Queue = G D

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 3 | 3 | 2 | 1 | 3 | 2 | 1 | 1 |



(f) Delete vertex from queue and visit it.

Deleted vertex = G, change status of 'G' to 3.

Visited vertices = A, B, C, F, G

Add neighbors of G to queue having status 1 and change their status to 2 = D.

Queue = D E

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 3 | 3 | 2 | 2 | 3 | 3 | 1 | 1 |

(g) Delete vertex from queue and visit it.

Deleted vertex = D, change status of 'D' to 3.

Visited vertices = A, B, C, F, G, D

Add neighbors of D to queue having status 1 and change their status to 2 = None.

Queue = E

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 3 | 3 | 3 | 2 | 3 | 3 | 1 | 1 |

(h) Delete vertex from queue and visit it.

Deleted vertex = E, change status of 'E' to 3.

Visited vertices = A, B, C, F, G, D, E

Add neighbors of E to queue having status 1 and change their status to 2 = H.

Queue = H

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 1 |

(i) Delete vertex from queue and visit it.

Deleted vertex = E, change status of 'H' to 3.

Visited vertices = A, B, C, F, G, D, E, H

Add neighbors of H to queue having status 1 and change their status to 2 = I.

Queue = I

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 |

(j) Delete vertex from queue and visit it.

Deleted vertex = I, change status of 'I' to 3.

Visited vertices = A, B, C, F, G, D, E, H, I

Add neighbors of I to queue having status 1 and change their status to 2 = none.

Queue = empty

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |

∴ BFS traversal = A B C F G D E H I

**Program 6.3:** Program for BFS traversal.

```
#include <stdio.h>
#define MAX 10
// Recursive BFS is not possible
void bfs(int adj[][MAX],int n, int v)
{
 int i, front, rear,visited[MAX];
 int que[20];
 front = rear = -1;
 for (i = 0;i < n;i++)
 visited[i] = 0;
 printf("%d ", v);
 visited[v] = 1;
 rear++;
 front++;
 que[rear] = v;
 while (front <= rear)
 {
 v = que[front]; /* delete from queue */
 front++;

 for (i = 0;i < n;i++)
 {
 /* Check for adjacent unvisited nodes */

 if (adj[v][i] == 1 && visited[i] == 0)
 {
 printf("%d ", i);
 visited[i] = 1;
 rear++;
 que[rear] = i;
 }
 }
 } /*End of while*/
} /*End of bfs()*/
int main()
{
 int adj[MAX][MAX],n,i,j,v;
 printf("Enter the number of nodes in graph maximum = %d\n",MAX);
 scanf("%d",&n);
 printf("\nEnter the adjacency matrix :\n");
}
```



```
for(i=0;i<n;i++)
{
 for(j=0;j<n;j++)
 scanf("%d",&adj[i][j]);
}
printf("\nAdjancy matrix is :\n");
for(i=0;i<n;i++)
{
 for(j=0;j<n;j++)
 printf("%d\t",adj[i][j]);
 printf("\n");
}
printf("\nEnter the starting node :");
scanf("%d",&v);
printf("\n BFS traversal from vertex %d is \n",v);
bfs(adj,n,v);
return(0);
}
```

**Output:**

Enter the number of nodes in graph maximum = 10

9

Enter the adjancy matrix

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Adjancy matrix is :

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Enter the starting node :0

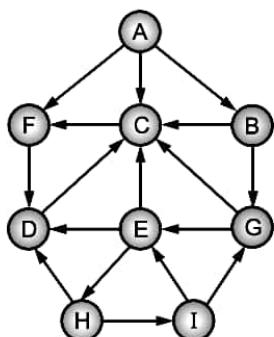
BFS traversal from vertex 0 is

0 1 3 4 2 6 5 7 8

### **6.3.2 Depth First Search**

[April 17, Oct. 15, 16, 17]

- The general idea behind a DFS beginning at a starting node A is as follows:
    - (a) First we examine the starting node A.
    - (b) Then we examine each node N along a path P which begins at A, that is we process a neighbor of A, then a neighbor of a neighbor of A and so on.
    - (c) After coming to a dead end that is to the end of the path P, we backtrack on P until we can continue along another path P' and so on.
  - The algorithm is very similar to the BFS except now we use a stack instead of the queue. Again a field STATUS is used to tell us the current status of a node.
  - The algorithm follows: This algorithm executes a DFS on a graph G beginning at a starting node A.
    1. Initialize all nodes to the ready state (STATUS=1).
    2. Push the starting node A onto STACK and change its status to the waiting state (STATUS=2).
    3. Repeat steps 4 and 5 until STACK is empty.
    4. Pop the top node N of STACK. Process N and change its status to the processed state (STATUS=3).
    5. Push onto STACK all the neighbors of N that are still in the ready state (STATUS=1) and change their status to the waiting state (STATUS=2).  
[end of step 3 loop]
    6. Stop.



**Adjacency List**

```
A = B, C, F
B = C, G
C = F
D = C
E = C, D, H
F = D
G = C, E
H = D, I
I = E, G
```

Consider the graph  $G$ . Suppose we want to find and print all the nodes reachable from the node  $A$  (including  $A$  itself).

(a) Push vertex 'A' into the stack.

Stack: A; change status of A to 2.



(b) Pop vertex from stack and change status of it to 3.

Popped vertex = A

Visited vertices = A

Push neighbors of A into stack having status 1 and change their status to 2 = B, C, F

top

Stack = B C F

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 2 | 2 | 1 | 1 | 2 | 1 | 1 | 1 |

(c) Pop vertex from stack and change status of it to 3.

Popped vertex = F

Visited vertices = A, F

Push neighbors to F into stack having status 1 and change their status to 2 = D.

top

Stack = B C D

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 2 | 2 | 2 | 1 | 3 | 1 | 1 | 1 |

(d) Pop vertex from stack and change status of it to 3.

Popped vertex = D

Visited vertices = A, F, D

Push neighbors of D into stack having status 1 and change their status to 2 = none.

top

Stack = B C

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 2 | 2 | 3 | 1 | 3 | 1 | 1 | 1 |

(e) Pop vertex from stack and change status of it to 3.

Popped vertex = C

Visited vertices = A, F, D, C

Push neighbors of C into stack having status 1 and change their status to 2 = none.

top

Stack = B

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I |
| 3 | 2 | 3 | 3 | 1 | 3 | 1 | 1 | 1 |



(f) Pop vertex from stack and change status of it to 3.

Popped vertex = B

Visited vertices = A, F, D, C, B

Push neighbors of B into stack having status 1 and change their status to 2 = G.

top

Stack = G

| A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|
| 3 | 3 | 3 | 3 | 1 | 3 | 2 | 1 | 1 |

(g) Pop vertex from stack and change status of it to 3.

Popped vertex = G

Visited vertices = A, F, D, C, B, G

Push neighbors of G into stack having status 1 and change their status to 2 = E.

Stack = E

| A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|
| 3 | 3 | 3 | 3 | 2 | 3 | 3 | 1 | 1 |

(h) Pop vertex from stack and change status of it to 3.

Popped vertex = E

Visited vertices = A, F, D, C, B, G, E

Push neighbors of E into stack having status 1 and change their status to 2 = H.

Stack = H

| A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 1 |

(i) Pop vertex from stack and change status of it to 3.

Popped vertex = H

Visited vertices = A, F, D, C, B, G, E, H

Push neighbors of H into stack having status 1 and change their status to 2 = I.

Stack = I

| A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|
| 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 2 |

(j) Pop vertex from stack and change status of it to 3.

Popped vertex = I

Visited vertices = A, F, D, C, B, G, E, H, I

Push neighbors of I into stack having status 1 and change their status to 2 = none.

Stack = empty

∴ DFS traversal = A F D C B G E H I

**Program 6.4:** Recursive program for DFS traversal.

```
#include <stdio.h>
#define MAX 10
void rec_dfs(int adj[][][MAX], int n,int v)
{
 static int visited[MAX]={0};
 int i;
 visited[v]=1;
 printf("%d\t",v);
 for (i = 0; i <= n-1;i++)
 {
 if (adj[v][i] == 1 && visited[i] == 0)
 {
 rec_dfs(adj,n,i);
 }
 }
}
int main()
{
 int adj[MAX][MAX],n,i,j,v;
 printf("Enter the number of nodes in graph maximum = %d\n",MAX);
 scanf("%d",&n);
 printf("\nEnter the adjancy matrix :\n");
 for(i=0;i<n;i++)
 {
 for(j=0;j<n;j++)
 scanf("%d",&adj[i][j]);
 }
 printf("\nAdjancy matrix is :\n");
 for(i=0;i<n;i++)
 {
 for(j=0;j<n;j++)
 printf("%d\t",adj[i][j]);
 printf("\n");
 }
 printf("\nEnter the starting node :");
 scanf("%d",&v);
 printf("\n DFS traversal from vertex %d is \n",v);
 rec_dfs(adj,n,v);
 return(0);
}
```

**Output:**

```
Enter the number of nodes in graph maximum = 10
9
Enter the adjancy matrix
0 1 0 1 1 0 0 0 0
0 0 1 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 1 0 0
0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1 0
0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0

Adjacency matrix is :
0 1 0 1 1 0 0 0 0
0 0 1 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 1 0 0
0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 1 0
0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0 0

Enter the starting node :0
DFS traversal from vertex 0 is
0 1 2 5 4 7 8 3 6
```

**Program 6.5:** Non-recursive program for DFS traversal.

```
#include <stdio.h>
#define MAX 10
void nonrec_dfs(int adj[][MAX],int n,int v)
{
 int i,stack[MAX], top = -1, pop_v;
 int visited[MAX];
 for (i = 0;i < n;i++)
 visited[i] = 0;
 top++;
 stack[top] = v;
 while (top >= 0)
 {
 pop_v = stack[top];
 top--; /*pop from stack*/
```



```
if (visited[pop_v] == 0)
{
 printf("%d ", pop_v);
 visited[pop_v] = 1;
}
else
continue;
for (i = n-1;i >= 0;i--)
{
 if (adj[pop_v][i] == 1 && visited[i] == 0)
 {
 top++;
 /* push all unvisited neighbours of pop_v */
 stack[top] = i;
 } /*End of if*/
} /*End of for*/
} /*End of while*/
} /*End of dfs()*/
int main()
{
 int adj[MAX][MAX],n,i,j,v;
 printf("Enter the number of nodes in graph maximum = %d\n",MAX);
 scanf("%d",&n);
 printf("\nEnter the adjancy matrix :\n");
 for(i=0;i<n;i++)
 {
 for(j=0;j<n;j++)
 scanf("%d",&adj[i][j]);
 }
 printf("\nAdjancy matrix is :\n");
 for(i=0;i<n;i++)
 {
 for(j=0;j<n;j++)
 printf("%d\t",adj[i][j]);
 printf("\n");
 }
 printf("\nEnter the starting node :");
 scanf("%d",&v);
 printf("\n DFS traversal from vertex %d is \n",v);
 nonrec_dfs(adj,n,v);
 return(0);
}
```

**Output:**

```
Enter the adjancy matrix
0 1 0 1 1 0 0 0 0
0 0 1 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 1 0 0
0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1 0
0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0
```

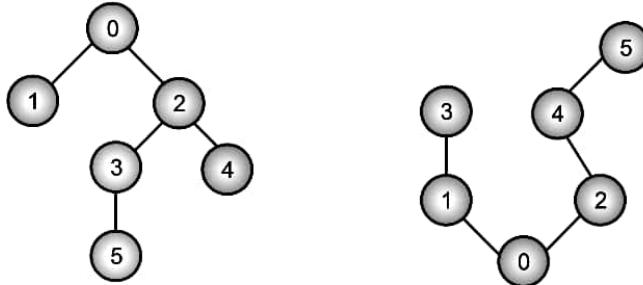
```
Adjacency matrix is :
0 1 0 1 1 0 0 0 0
0 0 1 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0
0 0 0 0 1 0 1 0 0
0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1 0
0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 0 0 0
```

```
Enter the starting node :0
DFS traversal from vertex 0 is
0 1 2 5 4 7 8 3 6
```

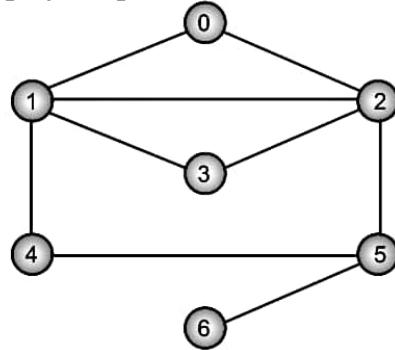
**6.4 SPANNING TREES**

[Oct. 17]

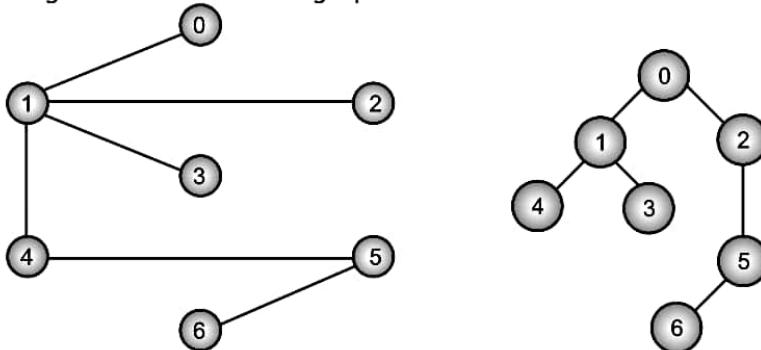
- A subgraph T of a connected graph G, which contains all the vertices of G and that tree is called a spanning tree of G because it spans overall vertices of graph G.
- Spanning tree does not contain cycles.
- It is not unique, there can be more than one spanning trees of a graph.
- If a graph is connected then it will always have a spanning tree.
- If a graph G is not connected then there will be no spanning tree of G.
- Trees can be defined as special cases of graphs. A tree may be defined as a connected graph without any cycle. Some examples of graphs as trees are:



- The only difference between general trees and trees defined here is that a tree defined as a special case of graph does not have special vertex called root. In fact any vertex can be chosen as root.
- A sub graph of a graph  $G = (V, E)$  which is tree containing all vertices of  $V$  is called a spanning tree of  $G$ . If  $G$  is not connected then there is no spanning tree of  $G$ . Now consider the graph  $G$  given below:



- Two spanning trees of the above graph are shown in below.



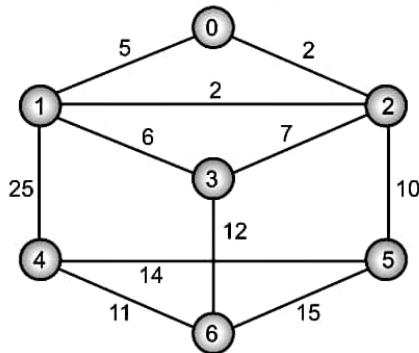
- If ' $G$ ' is a weighted graph and  $T_1, T_2$  are two spanning trees of ' $G$ ' then the sum of weights of all edges in  $T_1$  may be different from that of  $T_2$ .
- A spanning tree ' $T$ ' of ' $G$ ' where the sum of weights of all edges in ' $T$ ' is minimum is called the **minimal cost spanning tree** or **minimal spanning tree** of  $G$ .
- Algorithms for computing Minimal Spanning Tree:** There are two popular algorithms to calculate minimal cost spanning tree of a weighted undirected graph.  
**Algorithm Kruskal's and Prim's**



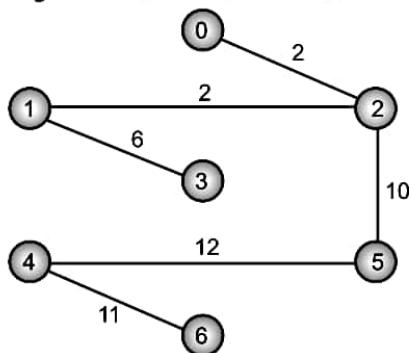
### 6.4.1 Kruskal's Algorithm

[April 17]

- Kruskal's algorithm functions on a list of edges of a graph where the list is arranged in order of weight of the edges.
- It begins with a forest of 'm' trees if the graph has 'm' vertices. The trees in the forest contain a different vertex of the graph and no edges.
- One edge from the sorted list is taken for connecting two forests in each step of the algorithm. The edge is added if the incorporation does not form a cycle. Once, an edge is selected, it is deleted from the list of edges.
- The algorithm continues until  $(n-1)$  edges are added to the list of edges exhausted. When the algorithm ends after adding  $(n-1)$  edges, a minimum spanning tree is produced. Consider the graph given below.



- There are eleven edges. The list of edges of the graph sorted in non-descending order may be given by  $\{ (0,2), (1,2), (0,1), (1,3), (2,3), (2,5), (4,6), (4,5), (3,6), (5,6), (1,4) \}$
- The minimal cost spanning tree is as shown below.



- First edge  $(0,2)$  is taken into account.
- $(1,2)$  is also be considered because edge does not result in a cycle.
- $(0,1)$  is not added to the partially formed trees as it forms a cycle.
- $(1,3)$  is now taken into account since inclusion of this edge does not form a cycle, this edge can be included.
- $(2,3)$  is added then a cycle  $2-1-3-2$  is formed, therefore this edge is not added.



- (2,5) is selected because edge does not form a cycle.
- (4,6) is also added.
- (4,5) is also considered.

**Program 6.6:** Program for Kruskal's Algorithm.

```
#include<stdio.h>
int i,j,k,a,b,u,v,n,ne=1;
int min,mincost=0,cost[9][9],parent[9];
int find(int);
int uni(int,int);
void main()
{
 printf("\n\n\tImplementation of Kruskal's algorithm\n\n");
 printf("\nEnter the no. of vertices\n");
 scanf("%d",&n);
 printf("\nEnter the cost adjacency matrix\n");
 for(i=1;i<=n;i++)
 {
 for(j=1;j<=n;j++)
 {
 scanf("%d",&cost[i][j]);
 if(cost[i][j]==0)
 cost[i][j]=999;
 }
 }
 printf("\nThe edges of Minimum Cost Spanning Tree are\n\n");
 while(ne<n)
 {
 for(i=1,min=999;i<=n;i++)
 {
 for(j=1;j<=n;j++)
 {
 if(cost[i][j]<min)
 {
 min=cost[i][j];
 a=u=i;
 b=v=j;
 }
 }
 }
 u=find(u);
 v=find(v);
 if(uni(u,v))
 {
 printf("\n%d edge (%d,%d) =%d\n",ne++,a,b,min);
 parent[u]=v;
 ne++;
 }
 }
}
```



```
 mincost +=min;
 }
 cost[a][b]=cost[b][a]=999;
}
printf("\n\tMinimum cost = %d\n",mincost);
}
int find(int i)
{
 while(parent[i])
 i=parent[i];
 return i;
}
int uni(int i,int j)
{
 if(i!=j)
 {
 parent[j]=i;
 return 1;
 }
 return 0;
}
```

**Output:**

```
Implementation of Kruskal's algorithm
Enter the no. of vertices
9
```

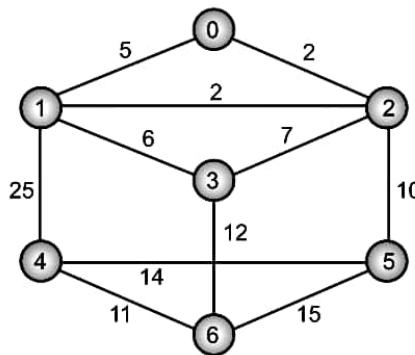
```
Enter the cost adjacency matrix
0 9 0 4 2 0 0 0 0
9 0 10 0 8 0 0 0 0
0 10 0 0 7 5 0 0 0
4 0 0 0 3 0 18 0 0
2 8 7 3 0 6 11 12 15
0 0 5 0 6 0 0 16 0
0 0 0 18 11 0 0 14 0
0 0 0 0 12 0 14 0 20
0 0 0 0 15 16 0 20 0
```

```
The edges of Minimum Cost Spanning Tree are
1 edge <1, 5> = 2
2 edge <4, 5> = 3
3 edge <3, 6> = 5
4 edge <5, 6> = 6
5 edge <2, 5> = 8
6 edge <5, 7> = 11
7 edge <5, 8> = 12
8 edge <5, 9> = 15
Minimum cost = 62
```



### 6.4.2 Prim's Algorithm

- Prim's algorithm starts with any arbitrary vertex as the partial minimal spanning tree 'T'. In each iteration of algorithm, one edge  $(U,V)$  is added to the partial tree 'T' so that exactly one end of this edge belongs to the set of vertices in 'T'.
- Of all such possible edges, the edge having the least cost is selected. The algorithm continues to add  $(n-1)$  edges. Consider the graph given below.



| <b>Set of vertices<br/>In the<br/>minimal<br/>Spanning<br/>tree</b> | <b>Edges, which<br/>have<br/>Exactly one end<br/>Belonging to the<br/>Partial minimal<br/>Spanning tree</b> | <b>The Edge<br/>chosen</b> | <b>The new partial<br/>Minimal spanning<br/>Tree</b> |
|---------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------|----------------------------|------------------------------------------------------|
| 0                                                                   | (0,2) (0,1)                                                                                                 | (0,2)                      |                                                      |
| (0,2)                                                               | (0,1) (1,2)<br>(2,3) (2,5)                                                                                  | (1,2)                      |                                                      |
| (0,1,2)                                                             | (2,3) (2,5)<br>(1,3) (1,4)                                                                                  | (1,3)                      |                                                      |
| (0,1,2,3)                                                           | (2,5) (1,4)<br>(3,6)                                                                                        | (2,5)                      |                                                      |

*contd. ...*



|               |                            |       |  |
|---------------|----------------------------|-------|--|
| (0,1,2,3,5)   | (1,4) (3,6)<br>(4,5) (5,6) | (4,5) |  |
| (0,1,2,3,4,5) | (3,6) (5,6)<br>(4,6)       | (4,6) |  |

**Program 6.7:** Program for Prim's Algorithm.

```
#define infinity 9999
#define MAX 20
#include<stdio.h>
#include<stdlib.h>
int G[MAX][MAX],n;
void prims();
int main()
{
 int i,j,op;
 do {
 printf("\n\n1>Create\n2>Prim's\n3>Quit");
 printf("\nEnter Your Choice : ");
 scanf("%d",&op);
 switch(op)
 { case 1:
 printf("\nEnter No. of vertices : ");
 scanf("%d",&n);
 printf("\nEnter the adjacency matrix :");
 for(i=0;i<n;i++)
 for(j=0;j<n;j++)
 scanf("%d",&G[i][j]);
 break;
 case 2: prims();break;
 }
 }while(op!=3);
}
```



```
void prims()
{
 int cost[MAX][MAX];
 int u,v,min_distance,distance[MAX],from[MAX];
 int visited[MAX],no_of_edges,i,min_cost,j;
 // create cost[][] matrix ,spanning[][]
 for(i=0;i<n;i++)
 for(j=0;j<n;j++)
 {
 if(G[i][j]==0)
 cost[i][j]=infinity;
 else
 cost[i][j]=G[i][j];
 }
 // initialise visited[],distance[] and from[]
 distance[0]=0;visited[0]=1;
 for(i=1;i<n;i++)
 {
 distance[i]=cost[0][i];
 from[i]=0;
 visited[i]=0;
 }
 min_cost=0; //cost of spanning tree
 no_of_edges=n-1; //no.of edges to be added
 while(no_of_edges>0)
 {
 //find the vertex at minimum distance from the tree
 min_distance=infinity;
 for(i=1;i<n;i++)
 if(visited[i]==0 && distance[i] < min_distance)
 {
 v=i;
 min_distance=distance[i];
 }
 u=from[v];
 printf("\n Edge in spanning tree is (v1,v2,cost) :
 (%d,%d,%d)",u,v,distance[v]);
 no_of_edges--;
 visited[v]=1;
 }
}
```



```
// update the distance[] array
for(i=1;i<n;i++)
 if(visited[i]==0 && cost[i][v] < distance[i])
 {
 distance[i]=cost[i][v];
 from[i]=v;
 }
 min_cost=min_cost+cost[u][v];
}
printf("\nTotal cost of spanning tree=%d",min_cost);
}
```

**Output:**

```
1. Create
2. Prim's
3. Quit
Enter Your choice : 1
Enter No. of vertices : 9

Enter the adjacency matrix :
0 9 0 4 2 0 0 0 0
9 0 10 0 8 0 0 0 0
0 10 0 0 7 5 0 0 0
4 0 0 0 3 0 18 0 0
2 8 7 3 0 6 11 12 15
0 0 5 0 6 0 0 16 0
0 0 0 18 11 0 0 14 0
0 0 0 12 0 14 0 20
0 0 0 15 16 0 20 0

1. Create
2. Prim's
3. Quit
Enter Your choice : 2
Edge in spanning tree is (v1, v2, cost) : (0, 4, 2)
Edge in spanning tree is (v1, v2, cost) : (4, 3, 3)
Edge in spanning tree is (v1, v2, cost) : (4, 5, 6)
Edge in spanning tree is (v1, v2, cost) : (5, 2, 5)
Edge in spanning tree is (v1, v2, cost) : (4, 1, 8)
Edge in spanning tree is (v1, v2, cost) : (4, 6, 11)
Edge in spanning tree is (v1, v2, cost) : (4, 7, 12)
Edge in spanning tree is (v1, v2, cost) : (4, 8, 15)
Total cost of spanning tree = 62
1. Create
2. Prim's
3. Quit
Enter Your choice : 2
```

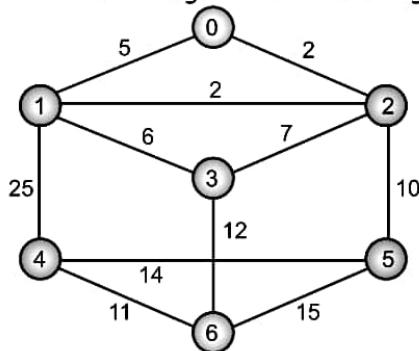


## 6.5 APPLICATIONS OF GRAPH

- Graphs can be used in various applications like:
  1. Representation of electric circuits - current flows.
  2. To represent maps indicating connectivity between different places.
  3. Telephone and computer networking.
  4. Routing from one location to another.
  5. For project scheduling.

### 6.5.1 Shortest Path

- As a final application of graphs one requiring somewhat more sophisticated reasoning, we consider the following problem. We are given a directed graph G in which every edge has a weight attached, and our problem is to find a path from one vertex V to another W such that the sum of the weights on the path is as small as possible. We call such a path as **shortest path**.
- Length of a path in a weighted graph is defined to be the sum of costs or weights of all edges in that path.
- In general there could be more than one path between a pair of specified vertices say "Vi" and "Vj" and a path with the minimum cost or weight is called the shortest path from "Vi" to "Vj". Note that the shortest path between two vertices may not be unique. Consider the weighted undirected graph

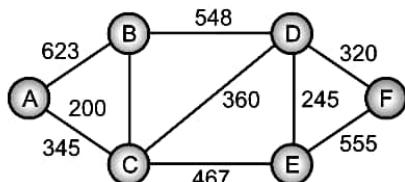


- It can be easily seen that there are more than three paths 0-1, 0-2-1, 0-2-3-1, from the vertex '0' to the vertex '1'. The path with the shortest path length is 0-2-1. The length of the shortest path is 4.
- There are many different variations of the shortest path problem. They vary with respect to the specification of the start vertex and end vertex. Some of the commonly known variants are listed below.
  - The shortest path from a specified source vertex to a specified destination vertex.
  - The shortest path from one specified vertex to all other vertices. This problem is also known as single source shortest path problem.
  - The shortest path between all possible source and destination vertices. This problem is also known as all pairs shortest path problem.
  - The shortest path can be determined using Kruskal's or Prim's algorithm.



### Solved Examples

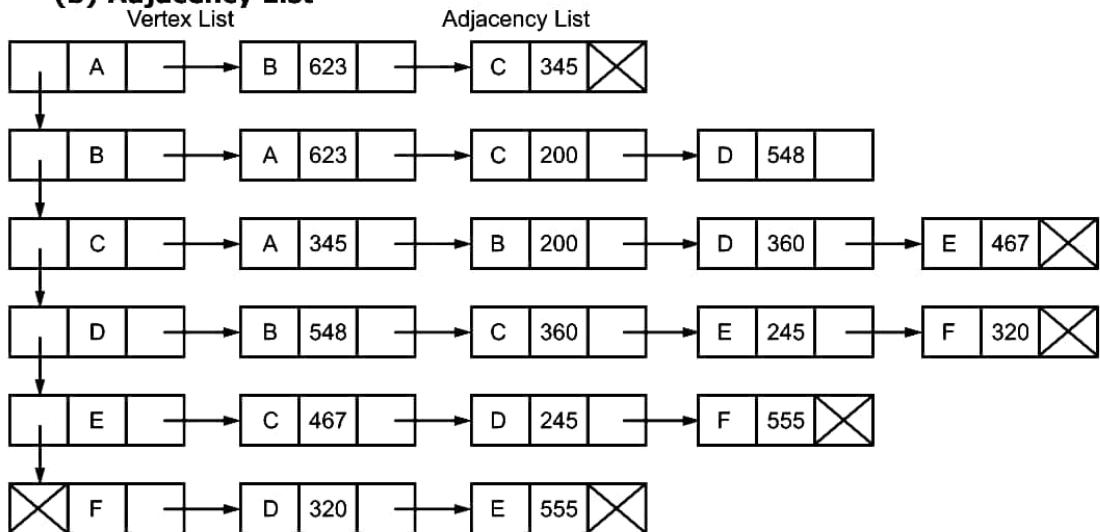
**Example 6.1:** Create an adjacency matrix and adjacency list for the weighted graph shown.



**Sol.: (a) Adjacency Matrix**

|   | A   | B   | C   | D   | E   | F   |
|---|-----|-----|-----|-----|-----|-----|
| A | 0   | 623 | 345 | 0   | 0   | 0   |
| B | 623 | 0   | 200 | 548 | 0   | 0   |
| C | 345 | 200 | 0   | 360 | 467 | 0   |
| D | 0   | 548 | 360 | 0   | 245 | 320 |
| E | 0   | 0   | 467 | 245 | 0   | 555 |
| F | 0   | 0   | 0   | 320 | 555 | 0   |

**(b) Adjacency List**

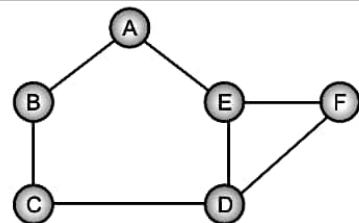


### Practice Questions

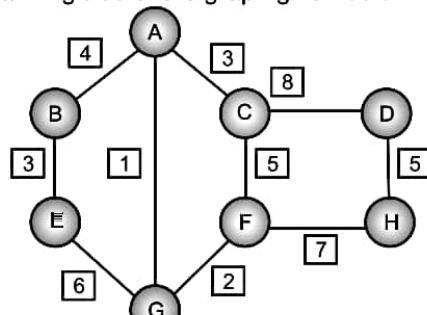
- Define graph.
- What is a graph?
- What do you mean by edges and vertices of a graph?
- Define the following terms: Degree, Cycle, Indegree, Sling.
- Differentiate between directed graph and undirected graph.
- What do you mean by complete graph?
- Differentiate between Breadth first and Depth search traversals in a graph.
- Describe spanning trees.
- How the graphs are implemented in computer memory?



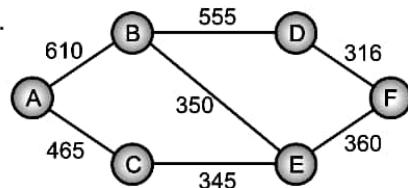
10. Write an algorithm for depth first traversal of a graph.  
Traverse the following graph using breadth first and depth first approach from node A.



11. Find the minimum spanning tree of the graph given below:



12. For a graph given below, draw the adjacency matrix.



13. Draw graph structure for the following matrix.

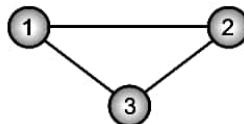
```

1 1 0 0 1 0
0 1 0 1 0 1
1 1 0 1 1 0
0 1 1 0 0 1
0 0 1 0 1 1
1 1 1 0 0 1

```

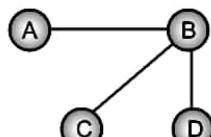
14. Consider the graph given below:

- Give adjacency matrix representation.
- Give adjacency list representation.



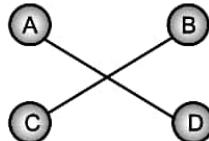
15. Consider the graph given below:

- Give adjacency matrix representation.
- Give adjacency list representation.



16. Consider the graph given below:

- Give adjacency matrix representation.
- Give adjacency list representation.





## University Question Papers

April 2015

**Time: 3 Hours**

**Max. Marks: 80**

**Q.1 Assume suitable data, if necessary:**

**[8 × 2 = 16]**

- (a) What is linked list structure?  
Ans. Refer Section 4.2.1.
- (b) Compare the efficiency of Bubble Sort with Insertion Sort?  
Ans. Refer Sections 2.4 and 2.5.
- (c) How to calculate count of Best, Worst and Average case?  
Ans. Refer Section 2.4.2.1.
- (d) What is Ancestor of Node?  
Ans. Refer Section 5.2.
- (e) What are the different types of data structures?  
Ans. Refer Section 1.8.
- (f) What are the applications of queue?  
Ans. Refer Section 4.10.
- (g) What is use of tree? How it is differ from linked list?  
Ans. Refer Sections 5.1.
- (h) What is difference between structure and polynomial?  
Ans. Refer Sections 1.1, 4.10.
- (i) State the types of graph.  
Ans. Refer Section 6.1.
- (j) What is use of (&) address operator and dereferencing (\*) operator?  
Ans. Refer Section 1.3.5.

**Q.2 Attempt any four of the following:**

**[4 × 4 = 16]**

- (a) Write an algorithm to convert given infix expression to prefix expression.  
Ans. Refer Section 4.5.
- (b) What is height-balanced tree? Explain LL and RR rotations.  
Ans. Refer Section 7.11.
- (c) Explain BFS with an example.  
Ans. Refer Section 6.3.1.
- (d) Write a function to remove last node of singly linked list and add it at the beginning.  
Ans. Refer Section 6.3.1.

```
#include<stdio.h>
#include<conio.h>
struct node
{
 int data;
 struct node *link;
}*start;
void create(int);
void disp();
void addlf();
```

(P.1)



```
void main()
{
int ch,n,i,m,a,pos;
clrscr();
start=NULL;
do
{
printf("\n\nMENU\n\n");
printf("\n1.CREATE\n");
printf("\n2.DISPLAY\n");
printf("\n3.ADDLF\n");
printf("\n4.EXIT\n");
printf("\nEnter UR CHOICE\n");
scanf("%d",&ch);
switch(ch)
{
case 1:
printf("\n\nHOW MANY NODES U WANT TO CREATE\n");
scanf("%d",&n);
for(i=0;i<n;i++)
{
printf("\nEnter THE DATA");
scanf("%d",&m);
create(m);
}
break;
case 3 :
addlf();
break;
case 2:
disp();
break;
case 4:
exit(0);
}
}
while(ch!=7);
getch();
}
```



```
void addlf()
{
 struct node *q, *tmp;
 q=start;
 while(q->link->link!=NULL)
 {
 q=q->link;
 }
 tmp=q->link;
 q->link=NULL;
 tmp->link=start;
 start=tmp;
}
void create(int data)
{
 struct node *q, *tmp;
 tmp=(struct node *)malloc(sizeof(struct node));
 tmp->data=data;
 tmp->link=NULL;
 if(start==NULL)
 {
 start=tmp;
 }
 else
 {
 q=start;
 while(q->link!=NULL)
 {
 q=q->link;
 q->link=tmp;
 }
 }
}
void disp()
{
 struct node *q;
 if(start==NULL)
 {
 printf("\n\nLIST IS EMPTY");
 }
 else
 {
 q=start;
```



```
while(q!=NULL)
{
 printf("%d->", q->data);
 q=q->link;
}
printf("NULL");
}
```

- (e) Write a function to display mirror image of given tree.

Ans. \*/

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

struct btnode
{
 int value;
 struct btnode *left, *right;
};

typedef struct btnode node;

/* function prototypes */
void insert(node *, node *);
void mirror(node *);

/* global variables */
node *root = NULL;
int val, front = 0, rear = -1, i;
int queue[20];
void main()
{
 node *new = NULL ;
 int num = 1;
 printf("Enter the elements of the tree(enter 0 to exit)\n");
 while (1)
 {
 scanf("%d", &num);
 if (num == 0)
 break;
 new = malloc(sizeof(node));
 new->left = new->right = NULL;
 new->value = num;
```



```
if (root == NULL)
 root = new;
else
{
 insert(new, root);
}
printf("mirror image of tree is\n");
queue[++rear] = root->value;
mirror(root);
for (i = 0;i <= rear;i++)
 printf("%d -> ", queue[i]);
printf("%d\n", root->right->right->right->value);
}
/* inserting nodes into the tree */
void insert(node * new , node *root)
{
 if (new->value > root->value)
 {
 if (root->right == NULL)
 root->right = new;
 else
 insert (new, root->right);
 }
 if (new->value < root->value)
 {
 if (root->left == NULL)
 root->left = new;
 else
 insert (new, root->left);
 }
}
/* mirror image of nodes */
void mirror(node *root)
{
 val = root->value;
 if ((front <= rear) && (root->value == queue[front]))
 {
 if (root->right != NULL || root->right == NULL)
 queue[++rear] = root->right->value;
 if (root->left != NULL)
 queue[++rear] = root->left->value;
 front++;
 }
}
```



```
if (root->right != NULL)
{
 mirror(root->right);
}
if (root->left != NULL)
{
 mirror(root->left);
}
```

**Q.3 Attempt any four of the following:****[4 × 4 = 16]**

- (a) Explain Quick sort technique with an example.

Ans. Refer Section 3.12.

- (b) Write a function which compares the contents of two queues and display message accordingly.

Ans. #include <functional>

```
#include <queue>
#include <vector>
#include <iostream>
template<typename T> void print_queue(T& q) {
 while(!q.empty()) {
 std::cout << q.top() << " ";
 q.pop();
 }
 std::cout << '\n';
}
int main() {
 std::priority_queue<int> q;
 for(int n : {1,8,5,6,3,4,0,9,7,2})
 q.push(n);
 print_queue(q);
 std::priority_queue<int, std::vector<int>,
 std::greater<int> > q2;
 for(int n : {1,8,5,6,3,4,0,9,7,2})
 q2.push(n);
 print_queue(q2);
 // Using lambda to compare elements.
 auto cmp = [] (int left, int right) { return (left ^ 1)
 < (right ^ 1); };
 std::priority_queue<int, std::vector<int>,
 decltype(cmp) > q3(cmp);
 for(int n : {1,8,5,6,3,4,0,9,7,2})
 q3.push(n);
 print_queue(q3);
}
```



- (c) What is doubly circular linked list? Explain its node structure.

Ans. Refer Section 3.3.4.

- (d) Write a function to merge given two singly linked Lists.

Ans. Refer Programs from Section 3.3.1.

- (e) Explain different types of asymptotic notations in details.

Ans. Refer Section 2.5.

**Q.4 Attempt any four of the following:**

[4 × 4 = 16]

- (a) Explain different types of dynamic memory allocation functions.

Ans. Refer Section 1.1.2.

- (b) Sort following data by using Insertion sort techniques: 12,5,122,9,7,54,4,23,88,60.

Ans. Insertion sort

[4, 5, 7, 9, 12, 23, 54, 60, 88, 122]

- (c) Write a function to display circular linked list in reverse order.

Ans. void circRev(struct node \*head1)  
{  
 if (head1->next==head)  
 {  
 cout << head1->data << "\n";  
 return;  
 }  
 circRev(head1->next);  
 cout << head1->data << "\n";  
}

- (d) Write a function to remove given node from singly linked list and add it at the end of list.

Ans. Refer Programs in Section 3.3.1.

- (e) What is graph? Explain its representation techniques in details.

Ans. Refer Sections 6.2.

**Q.5 Attempt any four of the following:**

[4 × 4 = 16]

- (a) Write a "C" program for addition of two polynomials.

Ans. #include<stdio.h>  
#include<conio.h>  
main()  
int a[10], b[10], c[10], m, n, k, k1, i, j, x  
clrscr()  
printf("\n\tPolynomial Addition\n")  
printf("\t=====\n")  
printf("\n\tEnter the no. of terms of the polynomial:")  
scanf("%d", &m)



```
printf("\n\tEnter the degrees and coefficients:")
for (i=0;i<2*m;i++
scanf("%d", &a[i]
printf("\n\tFirst polynomial is:")
k1=0
if(a[k1+1]==1
printf("x^%d", a[k1])
else
printf("%dx^%d", a[k1+1],a[k1])
k1+=2
while (k1<i
printf("+%dx^%d", a[k1+1],a[k1])
k1+=2
printf("\n\n\n\tEnter the no. of terms of 2nd polynomial:")
scanf("%d", &n)
printf("\n\tEnter the degrees and co-efficients:")
for(j=0;j<2*n;j++
scanf("%d", &b[j])
printf("\n\tSecond polynomial is:")
k1=0
if(b[k1+1]==1
printf("x^%d", b[k1])
else
printf("%dx^%d",b[k1+1],b[k1])
k1+=2
while (k1<2*n
printf("+%dx^%d", b[k1+1],b[k1])
k1+=2
i=0
j=0
k=0
while (m>0 && n>0
if (a[i]==b[j]
c[k+1]=a[i+1]+b[j+1]
c[k]=a[i]
m--
n--
i+=2
j+=2
else if (a[i]>b[j]
c[k+1]=a[i+1]
c[k]=a[i]
```



```
m--
i+=2
else
c[k+1]=b[j+1]
c[k]=b[j]
n--
j+=2
k+=2
while (m>0
c[k+1]=a[i+1]
c[k]=a[i]
k+=2
i+=2
m--
while (n>0
c[k+1]=b[j+1]
c[k]=b[j]
k+=2
j+=2
n--
printf("\n\n\n\n\tSum of the two polynomials is:");
k1=0;
if (c[k1+1]==1)
printf("x^%d", c[k1]);
else
printf("%dx^%d", c[k1+1],c[k1]);
k1+=2;
while (k1<k)
{
if (c[k1+1]==1)
printf("+x^%d", c[k1]);
else
printf("+%dx^%d", c[k1+1], c[k1]);
k1+=2;
}
getch();
return 0;
```

(b) What is an algorithm? How to measure its performance?

Ans. Refer Section 1.2.



(c) Write a function to count the number of leaf nodes in a tree.

Ans. /\*

```
* C Program to Count Number of Leaf Nodes in a Tree
 50
 / \
 20 30
 / \
 70 80
 / \ \
10 40 60
(50,20,30,70,80,10,40,60)
*/
#include <stdio.h>
#include <stdlib.h>
struct btnode {
 int value;
 struct btnode * l;
 struct btnode * r;
};
typedef struct btnode bt;
bt *root;
bt *new, *list;
int count = 0;
bt * create_node();
void display(bt *);
void construct_tree();
void count_leaf(bt *);
void main()
{
 construct_tree();
 display(root);
 count_leaf(root);
 printf("\n leaf nodes are : %d",count);
}
/* To create a empty node */
bt * create_node()
{
 new = (bt *)malloc(sizeof(bt));
 new->l = NULL;
 new->r = NULL;
}
```



```
/* To construct a tree */
void construct_tree()
{
 root = create_node();
 root->value = 50;
 root->l = create_node();
 root->l->value = 20;
 root->r = create_node();
 root->r->value = 30;
 root->l->l = create_node();
 root->l->l->value = 70;
 root->l->r = create_node();
 root->l->r->value = 80;
 root->l->r->r = create_node();
 root->l->r->r->value = 60;
 root->l->l->l = create_node();
 root->l->l->l->value = 10;
 root->l->l->r = create_node();
 root->l->l->r->value = 40;
}

/* To display the elements in a tree using inorder */
void display(bt * list)
{
 if (list == NULL)
 {
 return;
 }
 display(list->l);
 printf("->%d", list->value);
 display(list->r);
}

/* To count the number of leaf nodes */
void count_leaf(bt * list)
{
 if (list == NULL)
 {
 return;
 }
```



```
if (list->l == NULL && list->r == NULL)
{
 count++;
}
count_leaf(list->l);
count_leaf(list->r);
}
```

- (d) Write a recursive function for erasing linked list.

Ans. Refer Programs in Section 3.3.1.

- (e) What are the drawbacks of sequential storage?

Ans. Refer Section 3.7.3.

■ ■ ■

### October 2015

Time: 3 Hours

Max. Marks: 80

#### 1. Attempt any eight of the following:

[8 × 2 = 16]

- (a) What is use of "typedef" keyword?

Ans. • It allows us to introduce synonyms for data types which could have been declared some other way.  
• It is used to give New name to the Structure.  
• New name is used for Creating instances, Passing values to function, declaration etc.

- (b) What is self referential structure?

Ans. A self referential structure is used to create data structures like linked lists, stacks, etc. It is one of the data structures which refer to the pointer to (points) to another structure of the same type.

- (c) What is Priority queue?

Ans. A priority queue is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.

- (d) What are the advantages of array over linked list?

Ans. Advantages of array over linked list:

1. Complex to use and access - relatively complex as compared to arrays
2. No constant time access to the elements - simply because it doesn't involve the simple arithmetic used by arrays to compute the memory address, so relatively inefficient as compared to arrays



(e) What is pointer? What are the operations can be performed on the pointer?

Ans. A pointer is a memory location that holds a memory address.

- Pointers are more efficient in handling complex data structures and data tables.
- Pointers increase the execution speed.
- There are two special pointers operators \* and &.

(f) What is space complexity? How is it calculated?

Ans. The space complexity of the program is the amount of memory it needs to run to completion.

**Calculate Space Complexity:** Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

(g) What is difference between sorting and searching?

Ans. Searching allows you to find data that meets specific criteria. Sorting allows you to organize that data, based upon the rules you choose (most common is alphabetical).

(i) What is graph? State its types.

Ans. A graph is a set of two tuples such that  $G = (V, E)$   
where,  $G$  is graph

$V$  is a set of vertices or nodes

$E$  is the set of edges of graph  $G$

There are two types of graphs:

1. Undirected graph and
2. Directed graph

(j) What is use of (&) address operator and Dereferencing (\*) operator?

Ans. **Address Operator:** It returns address of the given variable.

For example, `int *ptr, var = 10;`  
`ptr = &var;`

**Dereferencing operator:** Is an unary operator which returns the value of the pointee by dereferencing the value which is addressed in pointer variable.

For example, `int var1 = 10, var;`  
`int *ptr;`  
`ptr = &var;`  
`var2 = ptr;`

## 2. Attempt any four of the following:

[**4 × 4 = 16**]

(a) Explain different types of Asymptotic notations in detail.

Ans. Refer Section 1.3.3.

(b) Explain BFS with an example.

Ans. Refer Section 6.3.1.

(c) Explain linear search method with an example.

Ans. Refer Section 2.1.

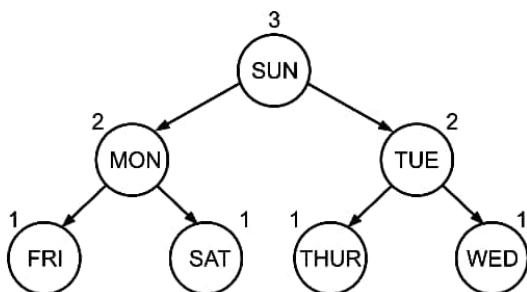


(d) Write a function to search given node in singly linked list and display its position.

Ans. Refer Section 3.3.1.

(e) Build AVL tree for the following data: FRI, MON, SAT, WED, SUN, TUE, THUR.

Ans.



**3. Attempt any four of the following:**

[4 × 4 = 16]

(a) What is queue? Explain its operations in detail.

Ans. Refer Section 4.7.1 and 4.8.

(b) What are the different ways we can represent graph? Explain any one with an example.

Ans. Refer Section 6.2.

(c) What is doubly linked list? Explain its node structure.

Ans. Refer Section 3.3.2.

(d) Convert given infix expression to postfix expression:  $(A - B)/C^D * E + (F - G)$

Ans.  $AB - CD ^ / E^* FG - +$

(e) Write a function to reverse singly linked list.

Ans. Refer Programs in Section 3.3.1

**4. Attempt any four of the following:**

[4 × 4 = 16]

(a) Write a function for adding and displaying elements from circular queue.

Ans. Refer Section 4.11.2.

(b) Write a function to check whether given string is palindrom€ or not (use stack).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 50
int top = -1, front = 0;
int stack[MAX];
void push(char);
void pop();
int main()
{
 int i, choice;
 char s[MAX], b;
```

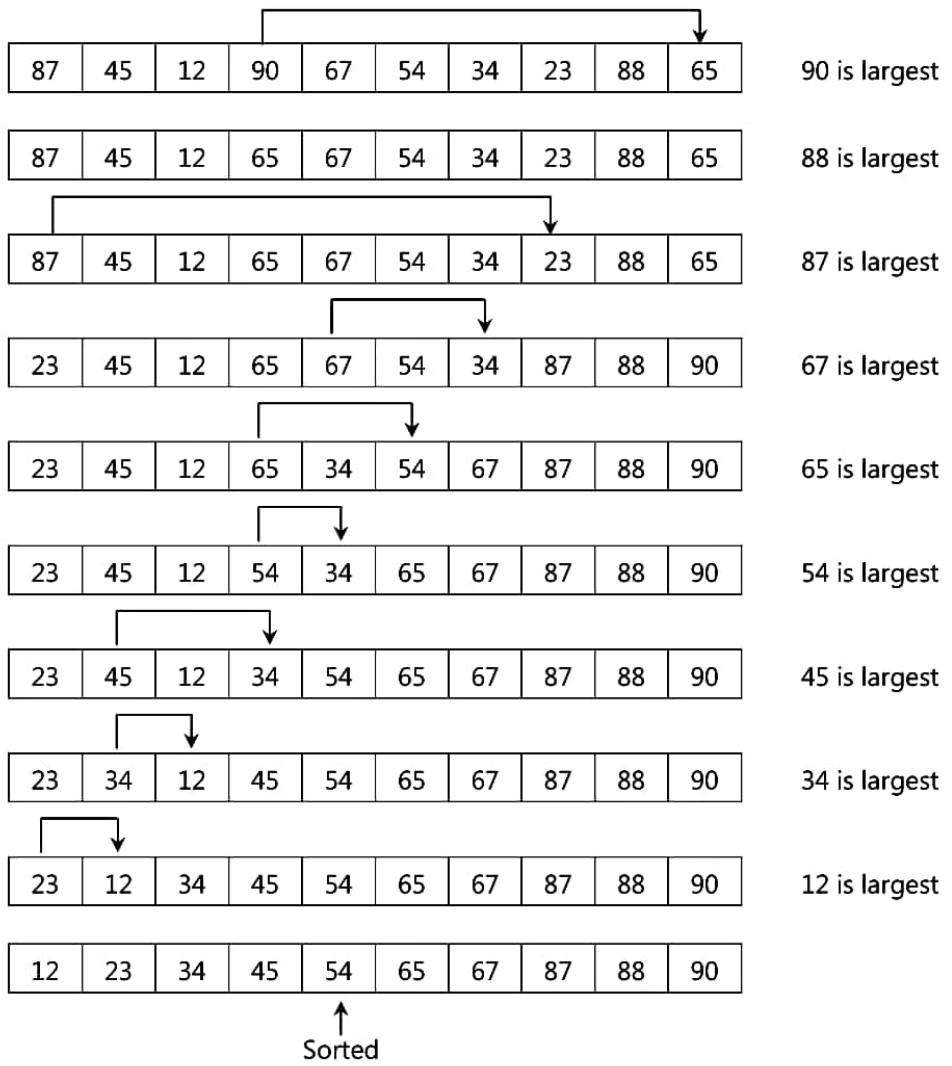


```
while (1)
{
 printf("1-enter string\n2-exit\n");
 printf("enter your choice\n");
 scanf("%d", &choice);
 switch (choice)
 {
 case 1:
 printf("Enter the String\n");
 scanf("%s", s);
 for (i = 0;s[i] != '\0';i++)
 {
 b = s[i];
 push(b);
 }
 for (i = 0;i < (strlen(s) / 2);i++)
 {
 if (stack[top] == stack[front])
 {
 pop();
 front++;
 }
 else
 {
 printf("%s is not a palindrome\n", s);
 break;
 }
 }
 if ((strlen(s) / 2) == front)
 printf("%s is palindrome\n", s);
 front = 0;
 top = -1;
 break;
 case 2:
 exit(0);
 default:
 printf("enter correct choice\n");
 }
}
/* to push a character into stack */
void push(char a)
{
 top++;
 stack[top] = a;
}
/* to delete an element in stack */
void pop()
{
 top--;
}
```



- (c) Sort the following data by using selection sort techniques 87, 45, 12, 90, 67, 54, 34, 23, 88, 65.

Ans.



- (d) Write a function to calculate the sum of elements of singly linked list.

Ans. Iterative Solution to calculate sum of elements of singly linked list

```
public int length(){
 int count=0;
 Node current = this.head;
 while(current != null){
 count++;
 current=current.next();
 }
 return count;
}
```



Recursive Solution to calculate sum of elements of singly linked list

```
public int length(Node current) {
 if(current == null) //base case
 return 0;
 return 1+length(current.next());
}
```

- (e) Write a function in "C" to traverse a graph using Depth first Search.

Ans. Refer Programs in Section 6.3.2.

**5. Attempt any four of the following:**

**[4 × 4 = 16]**

- (a) Write the recursive functions to traverse a tree by using inorder( ), preorder( ) and postorder() traversing techniques.

Ans. Node of a Binary tree is defined as below:

```
struct Node{
 Node * lptr; // Pointer to Left subtree
 int data;
 Node * rptr; // Pointer to Right subtree
};
```

**Pre-order traversal:** Visit the root node first (pre).

**Algorithm:**

1. Visit the root (we will print it when we visit to show the order of visiting)
2. Traverse the left subtree in pre-order
3. Traverse the right subtree in pre-order

**Code:**

```
/* Print Pre-order traversal of the tree */
void preOrder(node* r){
 if(r) {
 cout << r->data << " ";
 preOrder(r->left);
 preOrder(r->right);
 }
}
```

**Output:** 10 5 4 1 8 30 40

The root node comes before the left and right subtree (for every node of the tree)

Root Node



|                          |   |   |   |   |
|--------------------------|---|---|---|---|
| 10                       | 5 | 4 | 1 | 8 |
| <hr/>                    |   |   |   |   |
| Nodes of Left<br>subtree |   |   |   |   |

|                           |    |
|---------------------------|----|
| 3                         | 40 |
| <hr/>                     |    |
| Nodes of Right<br>subtree |    |



**In-order traversal:** Visit the root node in between the left and right node (in)

**Algorithm:**

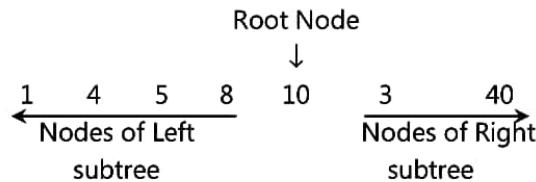
1. Traverse the left subtree in in-order
2. Visit the root (we will print it when we visit to show the order of visiting)
3. Traverse the right subtree in in-order

**Code:**

```
/* Print In-order traversal of the tree */
void inOrder(node* r){
 if(r){
 inOrder(r->left);
 cout << r->data << " ";
 inOrder(r->right);
 }
}
```

**Output:** 1 4 5 8 10 30 40

You may want to observe in the output that root lies in between the left and right traversals.



**Post-order traversal:** Visit the root node after (post) visiting the left and right subtree.

**Algorithm:**

1. Traverse the left subtree in in-order
2. Traverse the right subtree in in-order
3. Visit the root (we will print it when we visit to show the order of visiting)

**Code:**

```
/* Print Post-order traversal of the tree */
void postOrder(node* r){
 if(r){
 postOrder(r->left);
 postOrder(r->right);
 cout << r->data << " ";
 }
}
```

**Output:** 1 4 8 5 40 30 10



(b) Write an algorithm for evaluation of postfix expression.

Ans. Let 'operandstk' be an empty stack then the algorithm to evaluate postfix expression is:

1. Start
2. Input a postfix expression in the 'postfix' string.
3. for each character 'ch' of 'postfix' string

```
 if ('ch' is an operand)
 push 'ch' in 'operandstk'
 else
 {
 opnd2= pop an operand from 'operandstk'
 opnd1= pop an operand from 'operandstk'
 value= result of applying 'ch' in 'opnd1' and 'opnd2'
 push 'value' to 'operandstk'
 }
```

4. pop the top element of the stack 'operandstk' which is the required result.
5. Exit.

(c) Write a function to delete given position's node from singly linked list and display remaining list.

```
Ans. #include<stdio.h>
#include<assert.h>
#include<stdlib.h>
/* Link list node */
struct node
{
 int data;
 struct node* next;
};

/* Given a reference (pointer to pointer) to the head
 of a list and an int, push a new node on the front
 of the list. */
void push(struct node** head_ref, int new_data)
{
 /* allocate node */
 struct node* new_node =
 (struct node*) malloc(sizeof(struct node));
 /* put in the data */
 new_node->data = new_data;
 /* link the old list off the new node */
 new_node->next = (*head_ref);
```



```
/* move the head to point to the new node */
(*head_ref) = new_node;
}
void printList(struct node *head)
{
 struct node *temp = head;
 while(temp != NULL)
 {
 printf("%d ", temp->data);
 temp = temp->next;
 }
}
void deleteNode(struct node *node_ptr)
{
 struct node *temp = node_ptr->next;
 node_ptr->data = temp->data;
 node_ptr->next = temp->next;
 free(temp);
}
/* Sample program to test above function*/
int main()
{
 /* Start with the empty list */
 struct node* head = NULL;
 /* Use push() to construct below list
 1->12->1->4->1 */
 push(&head, 1);
 push(&head, 4);
 push(&head, 1);
 push(&head, 12);
 push(&head, 1);
 printf("\n Before deleting \n");
 printList(head);
 Deleting the head itself.
 deleteNode(head);
 printf("\n After deleting \n");
 printList(head);
 getchar();
 return 0;
}
```



(d) What is algorithm? Explain its characteristics in detail.

Ans. Refer Sections 1.2 and 1.2.4.

(e) Write a function to create and display circular linked list.

Ans. /\*

```
* C Program to Demonstrate Circular Single Linked List
*/
#include <stdio.h>
#include <stdlib.h>
struct node
{
 int data;
 struct node *link;
};
struct node *head = NULL, *x, *y, *z;
void create();
void ins_at_beg();
void ins_at_pos();
void del_at_beg();
void del_at_pos();
void traverse();
void search();
void sort();
void update();
void rev_traverse(struct node *p);
void main()
{
 int ch;
 printf("\n 1.Creation \n 2.Insertion at beginning
 \n 3.Insertion at remaining");
 printf("\n 4.Deletion at beginning
 \n 5.Deletion at remaining \n 6.traverse");
 printf("\n 7.Search\n 8.sort\n 9.update\n 10.Exit\n");
 while (1)
 {
 printf("\n Enter your choice:");
 scanf("%d", &ch);
```



```
switch(ch)
{
 case 1:
 create();
 break;
 case 2:
 ins_at_beg();
 break;
 case 3:
 ins_at_pos();
 break;
 case 4:
 del_at_beg();
 break;
 case 5:
 del_at_pos();
 break;
 case 6:
 traverse();
 break;
 case 7:
 search();
 break;
 case 8:
 sort();
 break;
 case 9:
 update();
 break;
 case 10:
 rev_traverse(head);
 break;
 default:
 exit(0);
}
```

}



```
/*Function to create a new circular linked list*/
void create()
{
 int c;
 x = (struct node*)malloc(sizeof(struct node));
 printf("\n Enter the data:");
 scanf("%d", &x->data);
 x->link = x;
 head = x;
 printf("\n If you wish to continue press 1 otherwise 0:");
 scanf("%d", &c);
 while (c != 0)
 {
 y = (struct node*)malloc(sizeof(struct node));
 printf("\n Enter the data:");
 scanf("%d", &y->data);
 x->link = y;
 y->link = head;
 x = y;
 printf("\n If you wish to continue press 1 otherwise 0:");
 scanf("%d", &c);
 }
}
/*Function to insert an element at the begining of the list*/
void ins_at_beg()
{
 x = head;
 y = (struct node*)malloc(sizeof(struct node));
 printf("\n Enter the data:");
 scanf("%d", &y->data);
 while (x->link != head)
 {
 x = x->link;
 }
 x->link = y;
 y->link = head;
 head = y;
}
```



```
/*Function to insert an element at any position the list*/
void ins_at_pos()
{
 struct node *ptr;
 int c = 1, pos, count = 1;
 y = (struct node*)malloc(sizeof(struct node));
 if (head == NULL)
 {
 printf("cannot enter an element at this place");
 }
 printf("\n Enter the data:");
 scanf("%d", &y->data);
 printf("\n Enter the position to be inserted:");
 scanf("%d", &pos);
 x = head;
 ptr = head;
 while (ptr->link != head)
 {
 count++;
 ptr = ptr->link;
 }
 count++;
 if (pos > count)
 {
 printf("OUT OF BOUND");
 return;
 }
 while (c < pos)
 {
 z = x;
 x = x->link;
 c++;
 }
 y->link = x;
 z->link = y;
}
/*Function to display the elements in the list*/
void traverse()
{
 if (head == NULL)
 printf("\n List is empty");
 else
 {
 x = head;
 while (x->link != head)
 {
 printf("%d->", x->data);
 x = x->link;
 }
 printf("%d", x->data);
 }
}
```

■■■

**April 2016****Time: 3 Hours****Max. Marks: 80****1. Attempt any eight of the following:** [8 × 2 = 16]

- (a) What is Data Structure?

Ans. Refer Section 1.4.

- (b) What is Ancestor of Node?

Ans. Refer Section 5.7.1.

- (c) What are the operations we can perform on to the queue?

Ans. Refer Section 4.10.

- (d) Differentiate between array and structure.

Ans. Refer Section 1.7.

- (e) What is Big-O notation?

Ans. Refer Section 1.3.3.

- (f) When multiplication of two polynomials is possible?

Ans. Refer Section 1.9.

- (g) What is strongly connected graph?

Ans. Refer Section 6.1.

- (h) Give the formulae for address calculation for row and column major representation?

Ans. Refer Section 1.8.

- (i) How to measure performance of an algorithm?

Ans. Refer Section 1.3.

- (j) What is indegree and outdegree of node in graph.

Ans. Refer Section 6.1.1.

**2. Attempt any four of the following:** [4 × 4 = 16]

- (a) Explain BFS traversing technique with an example.

Ans. Refer Section 6.3.1.

- (b) Explain Heap Sort technique with an example.

Ans. Refer Section 2.8.

- (c) Write a function to sort given singly linked list.

Ans. Refer Section 3.3.1.

- (d) Write a function to reverse a given string by using stack.

Ans. Refer Program in Chapter 4.

- (e) Write a program for addition of two polynomials.

Ans. Refer Section 1.9.3.

**3. Attempt any four of the following:** [4 × 4 = 16]

- (a) Write a function to traverse a graph by using DFS.

Ans. Refer Section 6.3.1.

- (b) Explain Height balance tree with an example.

Ans. Refer Section 5.7.

- (c) Explain graph representation techniques with an example.

Ans. Refer Section 6.2.



- (d) Sort the following data by using selection sort technique:  
56, 23, 2, 78, 122, 89, 43, 1

Ans. Refer Section 2.6.

- (e) Write a function to reverse singly linked list.

Ans. Refer Section 3.3.1.

**4. Attempt any four of the following:**

[ $4 \times 4 = 16$ ]

- (a) Write a function to calculate average of elements of nodes in singly linked list.  
(e.g. (value of first Node + Value of second node + ...)/Number of nodes)

Ans. Refer Section 3.3.1.

- (b) Write a function to create and display circular singly linked list.

Ans. Refer Section 3.3.3.

- (c) Explain different types of recursive tree traversing technique with example.

Ans. Refer Section 5.6.

- (d) Explain different types of recursive its characteristics in detail.

Ans. Refer Section 1.2.

**5. Attempt any four of the following:**

[ $4 \times 4 = 16$ ]

- (a) Write function to remove last node of singly linked list and add it at the beginning of list.

Ans. Refer Section 3.3.1.

- (b) Write a function to create doubly linked list and display it.

Ans. Refer Section 3.3.2.

- (c) Explain Quick Sort with an example.

Ans. Refer Section 2.7.

- (d) Write an algorithm for evaluation of postfix expression.

Ans. Refer Section 4.5.

- (e) What is circular queue? Explain it with an example.

Ans. Refer Section 4.11.

■■■

**October 2016**

**Time: 3 Hours**

**Max. Marks: 80**

**1. Attempt any eight of the following:**

[ $8 \times 2 = 16$ ]

- (a) What is linked list structure?

Ans. Refer Section 3.1.1.

- (b) What is the use of tree ? How is it differ from linked list ?

Ans. Tree data structure is used to store information having hierarchical relationship among the elements of that information.

For e.g. the directory structure maintain operating system.

The main difference between the tree and linked list is that the tree is non-linear data structure whereas linked list is linear data structure.

- (c) What is adjacency list ?

Ans. Refer Section 6.2.2.



- (d) How to calculate count of Best, Worst and Average case ?

Ans. Refer Section 1.3.2.

- (e) What is balance factor ? How is it calculated ?

Ans. In a binary tree the balance factor of a node N is defined to be the height difference between its left subtree and right subtree.

$$BF(N) = \text{Height of left subtree (N)} - \text{Height of right subtree (N)}$$

For an AVL tree, balance factor of every node of a tree can be either 0, 1 or -1.

- (f) Compare the efficiency of Bubble sort with Selection Sort.

Ans. Selection sort is faster than the bubble sort even though both have same complexities for worst, average and best cases.

Bubble sort uses more swap times, while selection sort avoids swaps. When using selection sort, it swaps n times at most, but when using bubble sort, it swaps almost  $n * (n - 1)$  times.

- (g) What is Polynomial ? How is it differ from Structure ?

Ans. Polynomial is an expression consisting of variables (or indeterminants) and coefficients, that involves only the operations of addition, subtraction, multiplication and non-negative integer exponents.

For example: Single variable polynomial can be  $3x^3 + x^2 - 4x + 7$ .

Structure is a collection of different data elements of different data types. Whereas polynomial is an array or linked list of structure of same type. That is each term of polynomial is one structure.

- (h) What is Priority Queue ?

Ans. Refer Section 4.11.1.

- (i) What is ADT ?

Ans. Refer Section 1.6.

- (j) List out the areas where data structures are applied extensively.

Ans. Areas where data structures are used:

(a) Compiler design

(b) Operating system

(c) Data management system

(d) Statistical analysis package

(e) Numerical analysis

(f) Graphics

(g) Artificial Intelligence

(h) Simulation

**2. Attempt any four of the following:** [4 × 4 = 16]

- (a) Write a function in "C" to transverse a graph using Breadth First Search technique.

Ans. #define MAX 10

```
//Graph is given using adjacency matrix as input
 //'n' is number of vertices
 //'v' is starting vertex
 void bfs (int adj[] [MAX], int n, int v)
{
 int i, front, rear, visited[MAX];
 int que[20]; //queue of integers
 front = rear = -1; //queue is initialized
 for(i = 0; i < n; i++)
 visited[i] = 0;
 printf("%d", v);
 visited[v] = 1;
 rear++;
 front++;
 que[rear] = v; //insert a vertex in queue
 while(front <= rear)
 {
 v = que[front] //delete a vertex from queue
 front++;
 for(i = 0; i < n; i++)
 {
 //check for adjacent unvisited nodes
 if(adj[v] [i] == 1 && visited [i] ==0)
 {
 printf("%d", i);
 visited[i] = 1;
 rear++;
 que[rear]=i;
 }
 }
 }
}
```

- (b) Explain different types of dynamic memory allocation functions.

Ans. Refer Section 1.1.2.



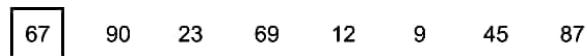
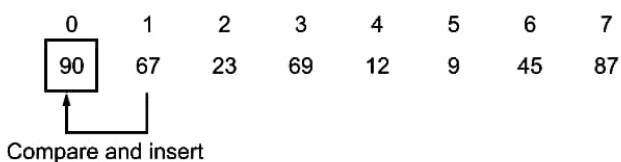
(c) Explain different types of Asymptotic notation in detail.

Ans. Refer Section 1.3.3.

(d) Sort the following data by using Insertion Sort technique : 90, 67, 23, 69, 12, 9, 45, 87

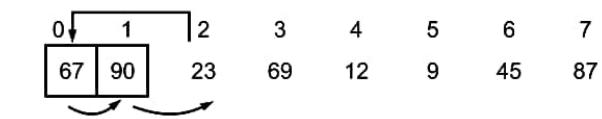
Ans. Iteration 1:

Iteration 1 :

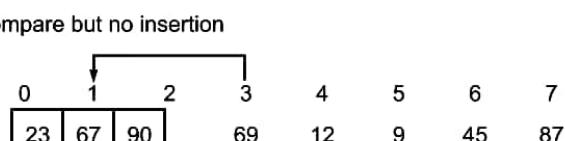
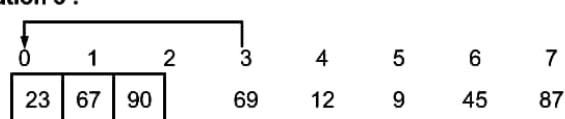


{Box indicate sorted list}

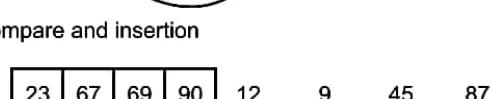
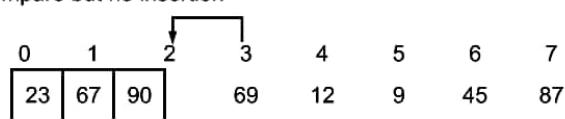
Iteration 2 :

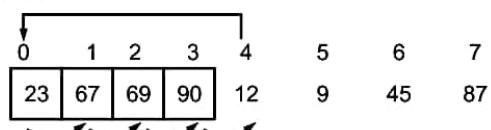


Iteration 3 :

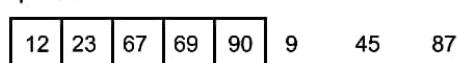
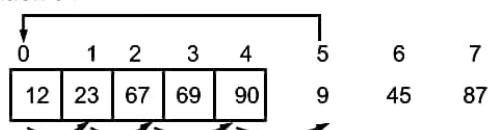


Compare but no insertion

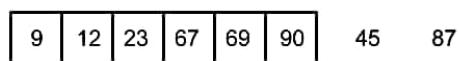
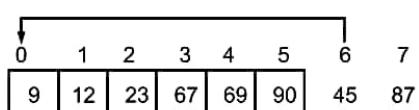


**Iteration 4 :**

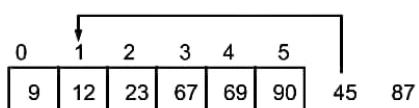
Compare and insert

**Iteration 5 :**

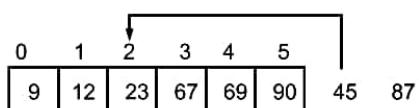
Compare and insert

**Iteration 6 :**

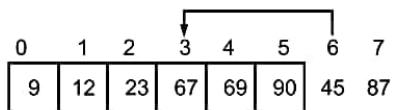
Compare but no insertion



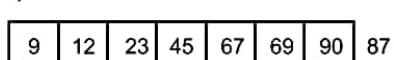
Compare but no insertion

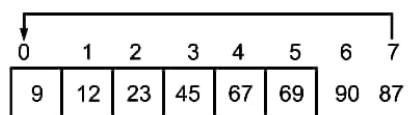


Compare but no insertion

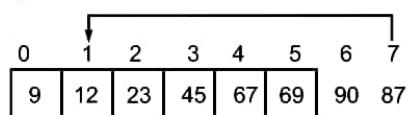


Compare and insert

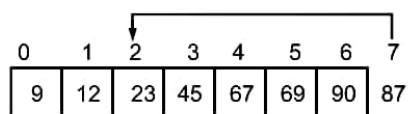


**Iteration 7 :**

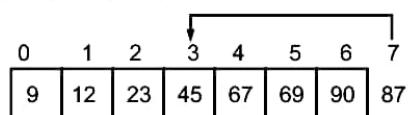
Compare but not insertion



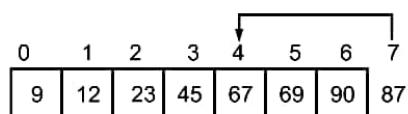
Compare but not insertion



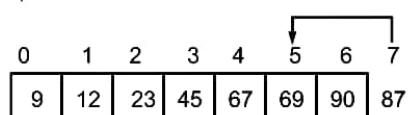
Compare but not insertion



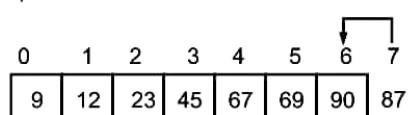
Compare but not insertion



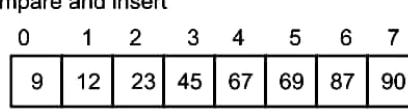
Compare but not insertion



Compare but not insertion



Compare and insert



Sorted array



- (e) Write a function to delete a node which is at given position in singly linked list and display remaining list.

Ans. `typedef struct node`

```
{
 int data;
 struct node * next;
} SNODE;
SNODE * delete_pos(SNODE * head, int pos)
{
 SNODE * temp, *ptr;
 int i;
 if(head == NULL)
 printf("\n list is empty.");
 else
 {
 temp=head; ptr=head;
 for(i=1; temp!=NULL && i < pos; i++)
 {
 ptr = temp;
 temp = temp → next;
 }
 if(temp==NULL)
 print("\n Wrong position.");
 else
 {
 if(temp == head)
 head=head→next;
 ptr→next=temp→next;
 temp→next=NULL;
 free(temp);
 }
 temp=head; //to print remaining list
 while(temp!=NULL)
 {
 print("%d \t", temp→data);
 temp=temp→next;
 }
 }
 return(head);
}
```

**3. Attempt any four of the following:****[4 × 4 = 16]**

- (a) Explain DFS with an example.

Ans. Refer Section 6.3.2.

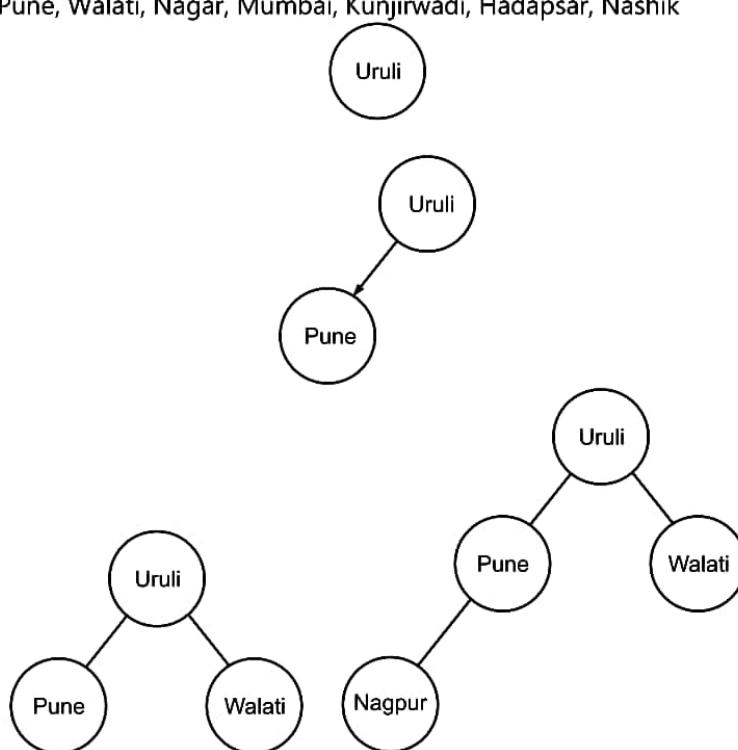
- (b) Write a recursive function for erasing linked list.

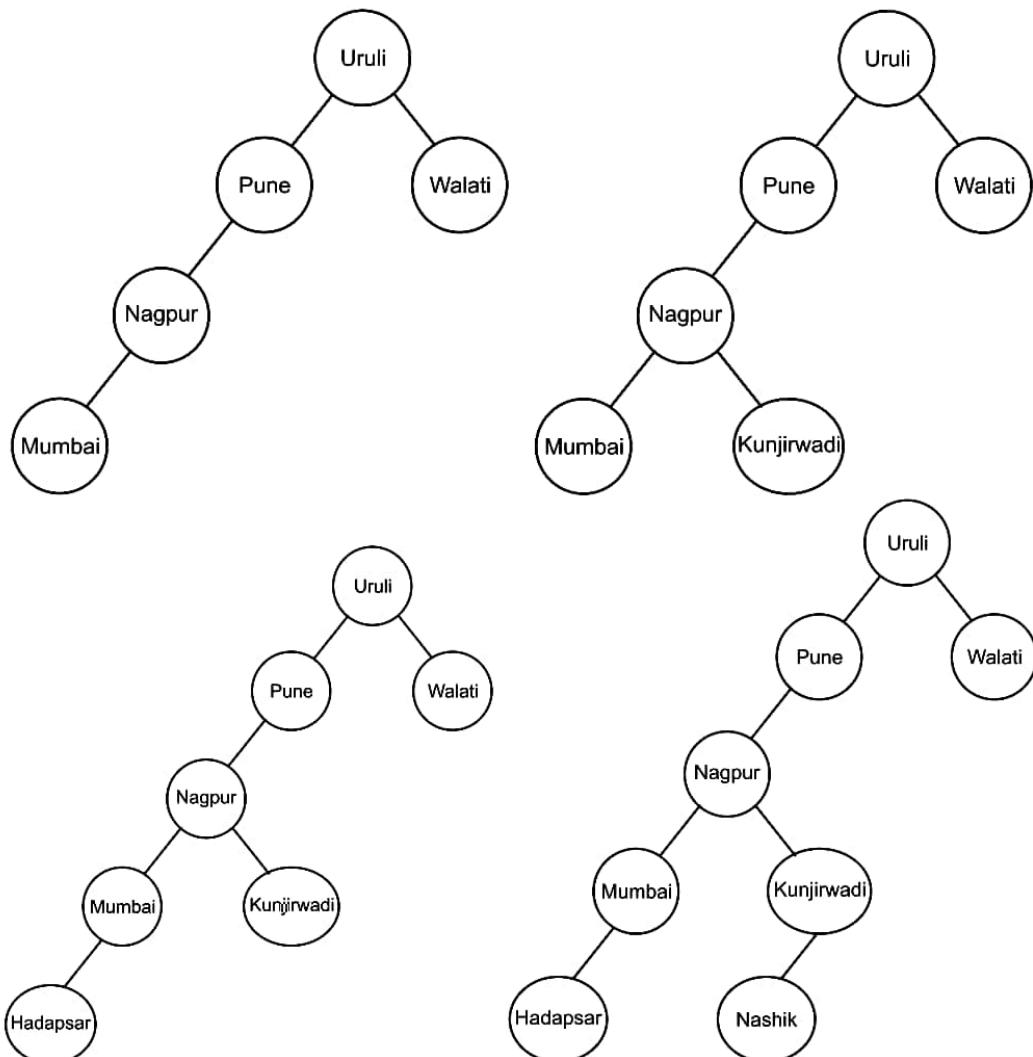
Ans. 

```
typedef struct node //node definition of SLL
{
 int data;
 struct node *next;
} SNODE;
void erase (SNODE *head)
{
 SNODE *temp = head;
 if(head!=NULL)
 {
 temp=head;
 head=head->next;
 temp->next=NULL;
 free(temp);
 erase(head);
 }
}
```

- (c) Construct Binary Search tree for the following data : Uruli, Pune, Walati, Nagar, Mumbai, Kunjirwadi, Hadapsar, Nashik.

Ans. Uruli, Pune, Walati, Nagar, Mumbai, Kunjirwadi, Hadapsar, Nashik





(d) Write a function to display mirror image of given tree.

Ans. `typedef struct node`  
{  
 int data;  
 struct node \*left;  
 struct node \*right;  
} BSTNODE;  
`BSTNODE *mirror(BSTNODE *root)`  
{  
 BSTNODE \*temp, \*temp1;  
 temp = root;



```
if(temp != NULL)
{
 if(temp->left!=NULL)
 temp->left=mirror(temp->left);
 if(temp->right!=NULL)
 temp->right=mirror(temp->right);
 temp1=temp->left;
 temp->left=temp->right;
 temp->right=>temp1;
 return(temp);
}
else
 return(NULL);
}
```

- (e) Explain Merge Sort Technique with an example.

Ans. Refer Section 2.9.

**4. Attempt any four of the following:**

**[4 x 4 = 16]**

- (a) Write a function to add node at given position in singly linked list and display complete list.

Ans. 

```
typedef struct node
```

```
{
 int data;
 struct node *next;
} SNODE;
SNODE *insert_pos(SNODE *head, int item, int pos)
{
 SNODE *newnode, *temp;
 int k;
 for (k=1; temp=head; temp!=NULL && k<pos; k++)
 {
 temp=temp->next;
 }
 if(temp == NULL)
 {
 printf("/n Wrong position.\n");
 return(head);
 }
 else
 {
 newnode=(SNODE *) malloc (size of (SNODE));
 newnode->data=item;
 newnode->next=NULL;
 if(temp!=head)
 {
 newnode->next=temp->next;
 temp->next=newnode;
 }
 }
}
```



```
 else
 {
 newnode->next=head;
 head=newnode;
 }
 temp=head;
 while(temp!=NULL)
 {
 printf("%d \t", temp->data);
 temp=temp->next;
 }
 return(head);
}
```

- (b) Differentiate between array and structure.

Ans. Refer Section 1.7.7.

- (c) Convert the following infix expression to postfix and prefix :  $(A - B/C) * (DAE + F)$

Ans. **Post expression form:**

First fully parenthesised the given expression according to operator precedence.

$((A - (B / C)) * ((D \wedge E)) + F))$

∴ Postfix expression is:

$A/-DE\wedge F+*$

**Prefix expression form:**

$((A - (B / C)) * ((D \wedge E)) + F))$

∴ Prefix expression is

$*-A/BC + F \wedge DE$

- (d) Write a 'C' program for evaluation of polynomial.

Ans. Refer Section 1.9.3, Program 1.3.

- (e) Write a function to count the number of leaf nodes in a tree.

Ans. `typedef struct node`

```
{
 int data;
 struct node *left;
 struct node *right;
} BSTNODE;
int count_total_leafnodes(BSTNODE *root)
{
 if(root==NULL)
 return 0;
 if(root->left==NULL && root->right==NULL)
 return 1;
 else
 return(count_total_leafnode(root->left)
 +count_total_leafnode(root->right));
}
```

**5. Attempt any four of the following :****[4 × 4 = 16]**

- (a) Write a function to create and display doubly circular linked list.

Ans. 

```
typedef struct dnode
```

```
{
 int data;
 struct dnode *prev;
 struct dnode *next;
} DNODE;
DNODE *head=NULL; //in main()
DNODE *Create(DNODE *head)
{
 int no;
 char ch;
 DNODE * newnode, *last;
 do
 {
 printf("\n Enter the value to insert:");
 scanf("%d", &no);
 newnode=(DNODE *) malloc(sizeof (DNODE));
 if(newnode==NULL)
 {
 printf("\n Insufficient memory.");
 exit(0);
 }
 else
 {
 newnode->data=no;
 newnode->prev=newnode->next=NULL;
 if(head==NULL) //first node
 {
 head=newnode;
 head->prev=head;
 head->next=head;
 last=head;
 }
 else
 { //append node
 last->next=newnode;
 newnode->prev=last;
 newnode->next=head;
 head->prev=newnode;
 last=newnode; //newnode will be now last node
 }
 }
 }
```



```
 printf("\n Do u want to enter another value (Y/N):");
 scanf("%c",&ch);
 }
 while(ch=='Y'|| ch=='y');
 return(head);
}
void display_foward(DNODE *head)
{
 DNODE *temp;
 if(head==NULL)
 printf("\n List is empty.");
 else
 {
 printf("\n linked list:");
 temp=head;
 do
 {
 printf("%d\t", temp->data);
 temp=temp->next;
 }while(temp!=head);
 }
}
void display_backward(DNODE *head)
{
 DNODE *temp, *last;
 if(head==NULL)
 printf("\n List is empty.");
 else
 {
 printf("\n linked list is:");
 temp=head->prev;
 last=head->prev;
 do{
 printf("%d \t", temp->data);
 temp=temp->prev;
 }while(temp!=last);
 }
}
```

- (b) Explain Linear Search Method with an example.

Ans. Refer Section 2.1.

- (c) Differentiate between Stack and Queue.

Ans. Refer Section 4.12.



- (d) Write a function to merge given two singly linked lists.

Ans. **Assumption:** Two singly linked list are sorted

```
typedef struct node
{
 int data;
 struct node *next;
} SNODE;
```

**Recursive function:**

```
SNODE * sortedmerge(SNODE * head1, SNODE * head2)
{
 SNODE & result=NULL;
 if(head1==NULL)
 return(head2);
 if(head2==NULL)
 return(head1);
 if(head1→data<=head2→data)
 {
 result=head1;
 result→next=sortedmerge(head1→next,head2);
 }
 else
 {
 result=head2;
 result→next=sortedmerge(head1, head2→next);
 }
 return(result);
}
```

- (e) Write a function which compares the contents of two queues and display message accordingly.

Ans. **typedef struct queue**

```
{
 int data[20];
 int front;
 int rear;
} SQUEUE;
void compare(SQUEUE*q1, SQUEUE*q2)
{
 int i, j, flag=1;
 i=q1→front;
 j=q2→front;
```



```
while(i<=q1->rear && j<=q2->rear)
{
 if(q1->data[i] !=q2->data[j])
 {
 printf("\n Queue contents are different.");
 flag=0;
 break;
 } else {i++; j++; }
 if(flag!=0)
 {
 if(i<=q1->rear)
 {
 printf("\n First queue contains more elements.");
 flag=0;
 }
 else if(j<=q2->rear)
 {
 printf("\n Second queue contains more elements.");
 flag=0;
 }
 }
 if(flag==1)
 {
 printf("\n Both queues contain same data.");
 }
}
```

■■■

**April 2017**

**Time: 3 Hours**

**Max. Marks: 80**

**1. Attempt any eight of the following:**

**[8 × 2 = 16]**

(a) What is Self Referential structure ?

Ans. Refer Section 1.10.

(b) What is efficiency of linear search method ?

Ans. • Efficiency of linear search method depends on the position of the target data stored in the list.  
• If the target data is stored at the first position, it requires only one comparison. Therefore, the best case complexity is 1.  
• If the target data is stored at the last position, it requires 'n' comparisons. Therefore, the worst case complexity is O(n).

Average number of comparisons =  $(1 + 2 + \dots + n)/n = (n + 1)/2$

Thus the average case complexity is O(n).



(c) What is difference between Binary Tree and Binary Search Tree ?

- Ans. • A Binary Tree is a special type of tree in every node or vertex has either no children or one child or maximum two children. The elements are inserted in the tree as per their insertion order.
- A binary search tree is a special type of Binary Tree in which every node or vertex also has maximum two children. But the element are inserted depending on its value i.e. if it is less than root then inserted as left child and if it is greater than root then inserted as right child.

(d) What are the different types of data structures ?

Ans. Refer Section 1.5.

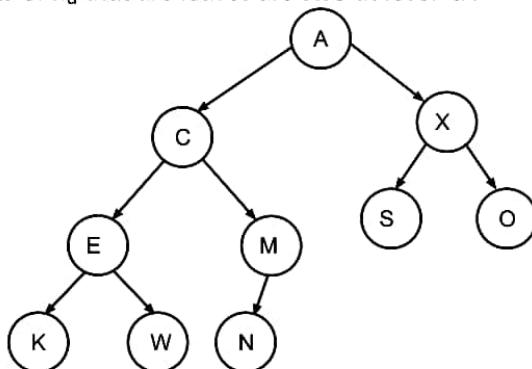
(e) What are the applications of queue ?

Ans. Refer Section 4.10.

(f) What is almost complete binary tree ?

Ans. A binary tree of depth 'd' is an almost complete binary tree if:

- (a) Each leaf in the tree is either at level 'd' or at level "d-1".
- (b) For any node  $n_d$  in the tree with a right descendant at level 'd', all the left descendants of  $n_d$  that are leaves are also at level 'd'.



(g) What is Double Ended Queue ?

Ans. Refer Section 4.11.3.

(h) State the types of graph.

Ans. Refer Section 6.1.

(i) What is the use of (&) address operator and Dereferencing (\*) operator ?

Ans. Refer Section 1.1.1.

(j) What is Pointer ? What are the operations we can perform on the pointer ?

Ans. A pointer is a variable whose value is the address of another variable i.e. direct address of the memory location.

#### Operations performed on the pointer:

**1. Assignment operator:** A pointer can be assigned the address stored by a pointer of the same type.

**2. Arithmetic operations:**

- (a) Adding or subtraction an integer to a pointer or from a pointer.
- (b) Subtracting one pointer from another.

**3. Comparison:** Pointers can be compared using ==, !=, <, >, etc.

**2. Attempt any four of the following:****[4 × 4 = 16]**

- (a) Write an algorithm to convert given infix expression to postfix expression.

Ans. Refer Section 4.6.2.

- (b) What is height-balanced tree ? Explain LL and RR rotations with an example.

Ans. Refer Section 5.7.

- (c) Differentiate between BFS and DFS.

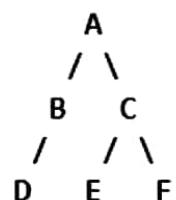
Ans.

| <b>BFS</b>                                                                                                                                                                                                      | <b>DFS</b>                                                                                                                                                                                                                                              |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1. BFS Stands for "Breadth First Search".                                                                                                                                                                       | 1. DFS stands for "Depth First Search".                                                                                                                                                                                                                 |
| 2. BFS starts traversal from the root node and then explore the search in the level by level manner i.e. as close as possible from the root node.                                                               | 2. DFS starts the traversal from the root node and explore the search as far as possible from the root node i.e. depth wise.                                                                                                                            |
| 3. Breadth First Search can be done with the help of queue i.e. FIFO implementation.                                                                                                                            | 3. Depth First Search can be done with the help of Stack i.e. LIFO implementations.                                                                                                                                                                     |
| 4. This algorithm works in single stage. The visited vertices are removed from the queue and then displayed at once.                                                                                            | 4. This algorithm works in two stages – in the first stage the visited vertices are pushed onto the stack and later on when there is no vertex further to visit those are popped-off.                                                                   |
| 5. BFS is slower than DFS.                                                                                                                                                                                      | 5. DFS is more faster than BFS.                                                                                                                                                                                                                         |
| 6. BFS requires more memory compare to DFS.                                                                                                                                                                     | 6. DFS require less memory compare to BFS.                                                                                                                                                                                                              |
| 7. Applications of BFS <ul style="list-style-type: none"> <li>• To find Shortest path</li> <li>• Single Source &amp; All pairs shortest paths</li> <li>• In Spanning tree</li> <li>• In Connectivity</li> </ul> | 7. Applications of DFS <ul style="list-style-type: none"> <li>• Useful in Cycle detection</li> <li>• In Connectivity testing</li> <li>• Finding a path between V and W in the graph.</li> <li>• useful in finding spanning trees and forest.</li> </ul> |
| 8. BFS is useful in finding shortest path. BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph                                                        | 8. DFS is not so useful in finding shortest path. It is used to perform a traversal of a general graph and the idea of DFS is to make a path as long as possible, and then go back (backtrack) to add branches also as long as possible.                |

***contd. ...***

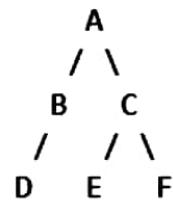


## 9. Example:



A, B, C, D, E, F

## 9. Example:



A, B, D, C, E, F

- (d) Write a function to display doubly linked list in reverse order.

```

Ans. //Definition of DLL node
typedef struct node
{
 int data;
 struct node*next;
 struct node*prev;
}DLLNODE;
Void display_reverse(DLLNODED *head)
{
 DLLNODE *temp=head;
 if(head==NULL)
 printf("\n list is empty.");
 else
 {
 printf("DLL in reverse order is:");
 while(temp->next!=NULL)
 temp=temp->next;
 while(temp!=NULL)
 {
 printf("%d \t", temp->data);
 temp=temp->next;
 }
 }
}

```

- (e) Explain different types of recursive tree traversing techniques with an example.

Ans. Refer Section 5.6.

**3. Attempt any four of the following:**

[4 × 4 = 16]

- (a) Explain Quick sort technique with an example.

Ans. Refer Section 2.7.

- (b) What is doubly circular linked list ? Explain its node structure.

Ans. Refer Section 3.3.4.

- (c) What are the drawbacks of sequential storage ?

Ans. Drawbacks of sequential storage:

1. In case of sequential storage, no direct access to a particular or in between element is possible.
2. Data has to be accessed in order i.e. first, second, third and so on.
3. In case of array sequential storage, the fixed amount of memory is allotted.



4. Linked list data structure has dynamic memory allocation, but it requires additional memory to store the pointers. Lot of pointer manipulations have to be performed while inserting or deleting data from linked list.
5. In case of linked list, loss of any pointer will cause in loss of data.

(d) Write a function to sort given singly linked list.

Ans. 

```
typedef struct node
```

```
{
 int data;
 struct node *next;
} SLLNODE;
void sort(SLLNODE * head)
{
 SLLNODE *temp, *temp1
 int data;
 temp=head;
 while(temp!=NULL)
 {
 temp1 = temp->next;
 while(temp1 !=NULL)
 {
 if(temp->data>temp1->data)
 {
 data=temp->data;
 temp->data=temp1->data;
 temp1->data=data;
 }
 temp1=temp1->next;
 }
 temp=temp->next;
 }
}
```

(e) Write a function to check whether given expression is parenthesis or not.

Ans. 

```
#define MAX 50
```

```
typedef struct
{
 char data[MAX];
 int top;
} STATIC_STACK;
void initstack(STATIC_STACK *s)
{
 int i;
 for(i=0; i<max; i++)
 s->data[i]=-1;
 s->top=-1;
}
```



```
int isfull(STATIC_STACK *s)
{
 if(s->top==MAX-1)
 return1;
 return 0;
}
int isempty(STATIC_STACK *s)
{
 if(s->top===-1)
 return1;
 return 0;
}
void push(STATIC_STACK * s, char ch)
{
 (s->top)++;
 s->data[s->top]=ch;
}
char pop(STATIC_STACK * s)
{
 char ch;
 ch = s->data[s->top];
 (s->top);
 return(ch);
}
int check_para_balance(char *str)
{
 STATIC_STACK st;
 int i=0;
 initstack(&st);
 while(str[i]!='\0');
 {
 if(str[i]=='(' || str[i] == '[' || str[i] == '{')
 {
 push(&st, str[i]);
 i++;
 }
 else
 { if(str[i]==')')
 {
 if*(isempty(&st) || pop (&s) != '(') return(0);
 else
 i++;
 }
 else if(str[i] == ']')
 {
 if(isempty(&st) || pop(&st)! = '[') return(0);
 else
 i++;
 }
 }
}
```



```

 else if(str[i] == '}')
 {
 if(esempty(&st) || pop(&st) != '(')
 return(0);
 else
 i++;
 }
 else
 i++;
 } //end of else
}// end of while
if(! isempty(&st))
 return(0);
else
 return(1);
}

```

**4. Attempt any four of the following:****[4 × 4 = 16]**

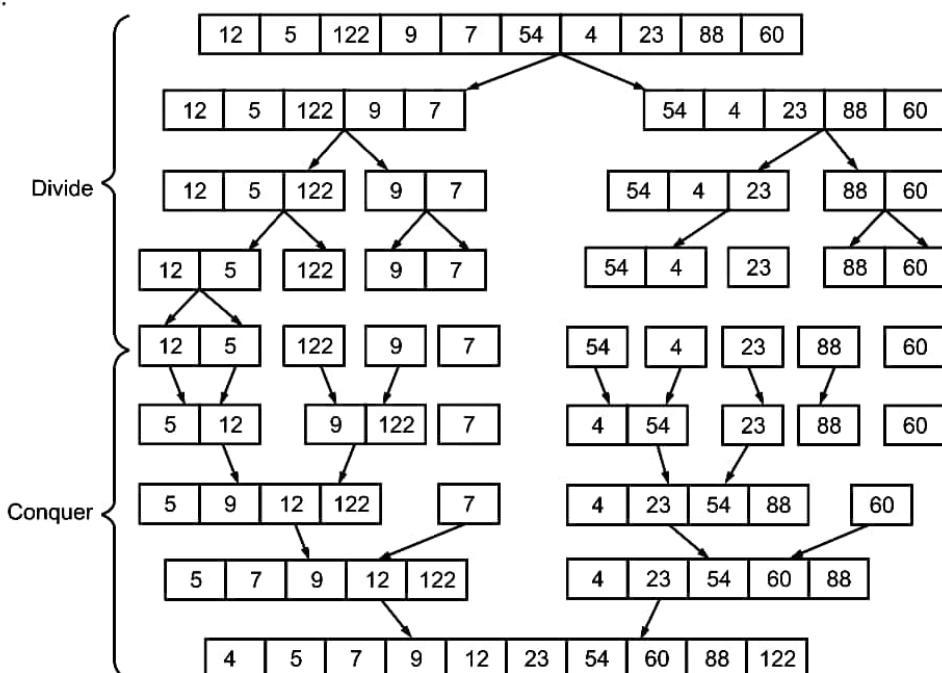
- (a) Explain different types of dynamic memory allocation functions.

Ans. Refer Section 1.1.2.

- (b) Sort following data by using Merge sort techniques :

12, 5, 122, 9, 7, 54, 4, 23, 88, 60

Ans.





- (c) Write a function to remove given node from singly linked list and add it at the given position in singly linked list.

Ans.

```
typedef struct node
{
 int data;
 struct node *next;
}SLLNODE;
//To a function we pass address of head node, address of a node to delete and
//position to insert.
SLLNODE*insert_delete(SLLNODE*head, SLLNODE*delnode, int (pos)
{
 SLLNODE *temp=head, *prev=head;
 int i;
 if(delnode==head) //headnode to be deleted
 {
 head=head->next;
 }
 else " temp=temp->next;
 while(temp!=NULL)
 {
 if(temp==delnode)
 break;
 else
 {
 prev=prev;
 temp=temp->next;
 }
 }
 if(temp==NULL)
 {
 printf("\n The node to be deleted is not found in
list.");
 return(head);
 }
 else
 {
 prev->next=temp->next;
 }
}// end of else
if(pos==1)
{
 delnode->next=head;
 head=delnode;
 return(head);
}
else
{
 temp=head;
 for(i=0; i<pos; i++)
 {
 temp=temp->next;
 if(temp==NULL)
 {
```



```
 printf("\n Wrong position");
 return(head);
 }
}
delnode->next=temp->next;
temp->next=delnode;
return(head);
} //end of else
}

(d) Write a function to create and display circular singly linked list.

Ans. typedef struct node
{
 int data;
 struct node *next;
} (CSLLNODE);
(CSLLNODE create(CLNNODE * head)
{
 int no;
 char ch;
 CSLL NODE *newnode, *last;
 do
 {
 printf("Enter the value to insert:");
 scanf("%d", &no);
 newnode=((CSLLNODE*)malloc(sizeof(CSLLNODE)));
 newnode->data=no;
 newnode->next=NULL;
 if(head==NULL)
 {
 head=newnode;
 newnode->next=head;
 last=head;
 }
 else
 {
 last->next=newnode;
 newnode->next=head;
 last=newnode;
 }
 printf("Do u want to enter another value(Y/N):");
 scanf("y.c",&ch);
 }while(ch=='y' || ch=='Y');
 printf("\n linked list is:");
 last=head;
 do
 {
 printf("%d \t", last->data);
 last=last->next;
 }
 while(last!=head);
}
```



(e) Evaluate the following Postfix expression : 4, 5, 4, 2, ^, +, \*, 2, 2, A, 9, 3, 1, \*, +, +  
 Ans.

| Token        | Stack      | Operand1 | Operand2 | Result            | Stack Operation                   |
|--------------|------------|----------|----------|-------------------|-----------------------------------|
| 4            | 4          |          |          |                   | Push                              |
| 5            | 4 5        |          |          |                   | Push                              |
| 4            | 4 5 4      |          |          |                   | Push                              |
| 2            | 4 5 4 2    |          |          |                   | Push                              |
| ^            | 4 5        | 4        | 2        | $4 \wedge 2 = 16$ | op2=pop()<br>op1=pop()<br>Push 16 |
| +            | 4          | 5        | 16       | $5 + 16 = 21$     | op2=pop()<br>op1=pop()<br>Push 21 |
| *            |            | 4        | 21       | $4 * 21 = 84$     | op2=pop()<br>op1=pop()<br>Push 84 |
| 2            | 84 2       |          |          |                   | Push                              |
| 2            | 84 2 2     |          |          |                   | Push                              |
| ^            | 84         | 2        | 2        | $2 \wedge 2 = 4$  | op2=pop()<br>op1=pop()<br>Push 4  |
| 9            | 84 4 9     |          |          |                   | Push                              |
| 3            | 84 4 9 3   |          |          |                   | Push                              |
| 1            | 84 4 9 3 1 |          |          |                   | Push                              |
| *            | 84 4 9     | 3        | 1        | $3 * 1 = 3$       | op2=pop()<br>op1=pop()<br>Push 3  |
| -            | 84 4       | 9        | 3        | $9 - 3 = 6$       | op2=pop()<br>op1=pop()<br>Push 6  |
| +            | 84         | 4        | 6        | $4 + 6 = 10$      | op2=pop()<br>op1=pop()<br>Push 10 |
| +            |            | 84       | 10       | $84 + 10 = 94$    | op2=pop()<br>op1=pop()<br>Push 94 |
| No<br>Symbol |            |          |          |                   | result=pop()<br>result=94         |

**5. Attempt any four of the following:****[4 × 4 = 16]**

- (a) Differentiate between doubly linked list and tree.

Ans.

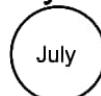
| Doubly Linked List                                                                                               | Tree                                                                                              |
|------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| 1. Doubly linked list is linear data structure.                                                                  | 1. Tree is non-linear data structure.                                                             |
| 2. DLL is used to store data having linked between them.                                                         | 2. Tree is used to store data having hierarchical relationship.                                   |
| 3. Node structure of DLL has 2 pointers-one pointing to predecessor node and another pointing to successor node. | 3. Tree node structure also has 2 pointers one pointing to left child and another to right child. |
| 4. DLL is a sequential data structure.                                                                           | 4. Tree is not a sequential data structure.                                                       |
| 5. All DLLs can be considered as tree.                                                                           | 5. All trees are not treated as doubly linked list.                                               |

- (b) Explain Kruskal's algorithm for minimum spanning tree with an example.

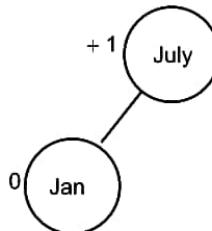
Ans. Refer Section 6.4.1.

- (c) Construct Binary Search Tree for the following Data : July, Jan, Feb, Dec, Mar, Oct, Nov, Apr, Jun, Aug.

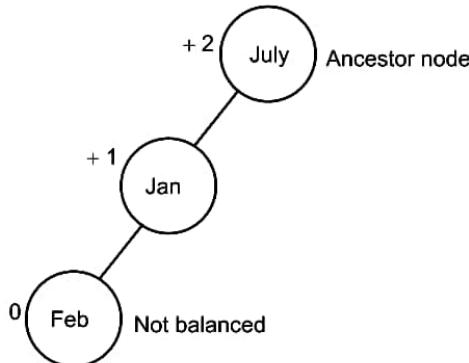
Ans. 1. July:

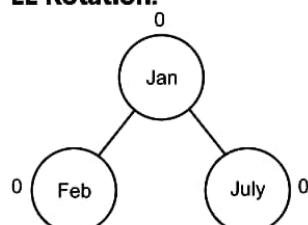
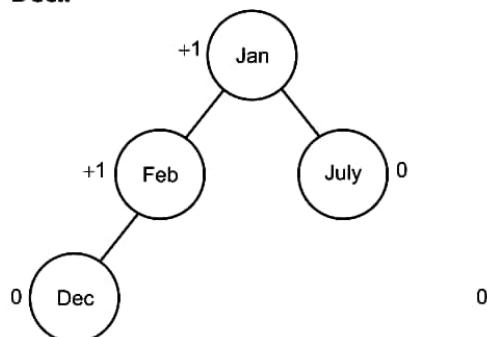
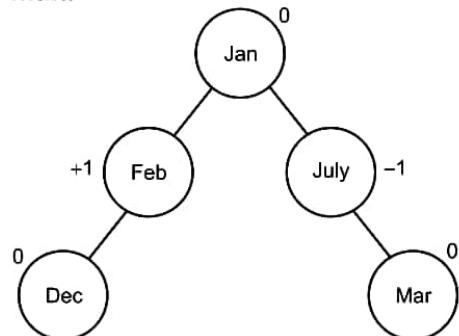
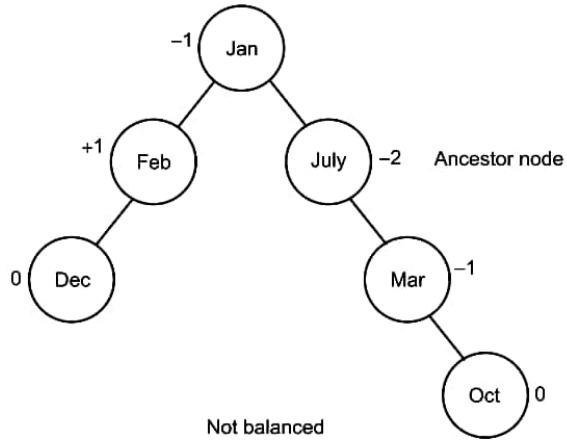


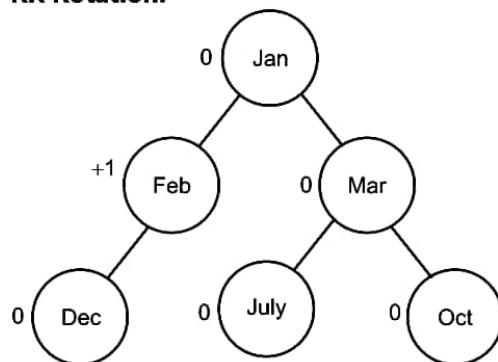
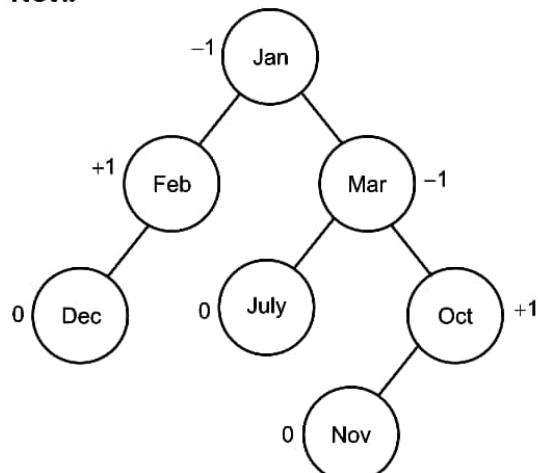
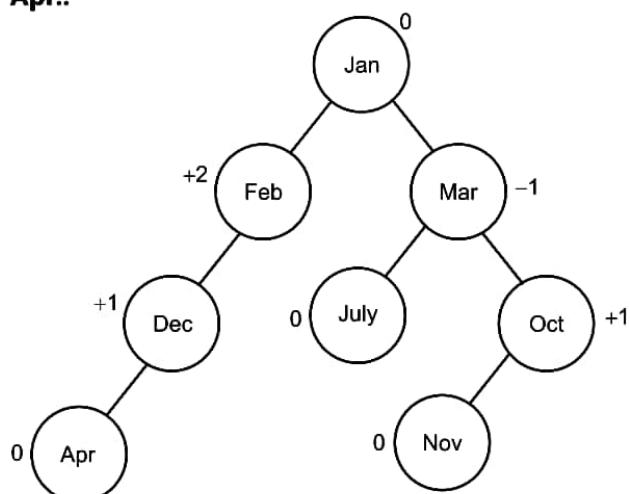
2. Jan.:

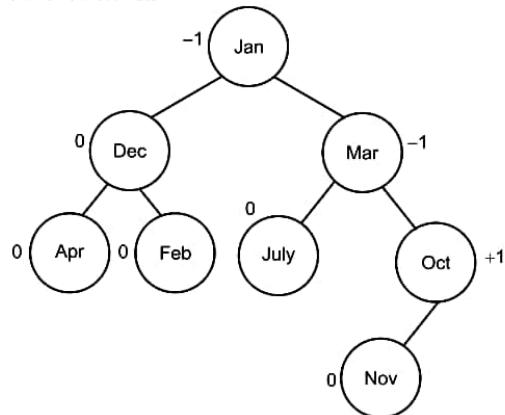
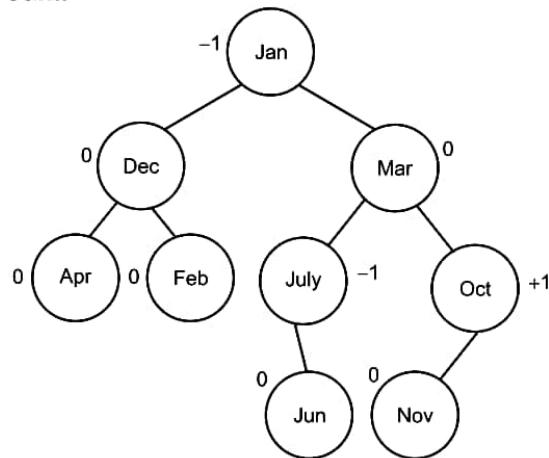
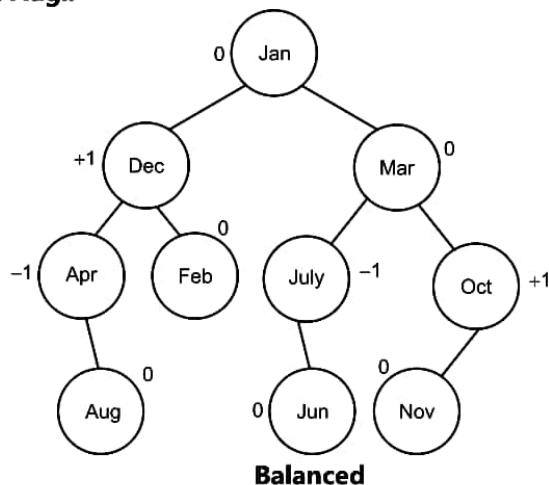


3. Feb.:



**LL Rotation:****4. Dec.:****5. Mar.:****6. Oct.:**

**RR Rotation:****7. Nov.:****8. Apr.:****Not Balanced**

**LL Rotation:****9. Jun.:****10. Aug.:**



- (d) Explain Binary Search Method with an example.  
Ans. Refer Section 2.2.
- (e) Write a function to remove first node from singly linked list and display remaining list.

Ans.

```
typedef struct node
{
 int data;
 struct node *next;
} SLLNODE;
SLLNODE *remove(SLLNODE *head)
{
 SLLNODE * temp=head;
 temp->next=NULL; head=head->next;
 free(temp);
 temp=head;
 while(temp!=NULL)
 {
 printf("%d\t", temp->data);
 temp=temp->next;
 }
 return(head);
}
```

■■■  
October 2017

Time : 3 Hours

Max. Marks: 80

**N.B.:**

1. All questions are compulsory.
2. All questions carry equal marks.
3. Assume suitable data, if necessary.

**1. Attempt any eight of the following:**

[8 × 2 = 16]

- (a) What is use of (&) address operator and Dereferencing (\*) operator?

Ans. Please Refer to Section 1.1.1.

- (b) What is efficiency of linear search method?

Ans. Please Refer to Q.1 (b) of April 2017.

- (c) What is Double Ended Queue?

Ans. Please Refer to 4.11.3.

- (d) Compare the efficiency of Bubble sort with Selection Sort.

Ans. Please Refer to Sections 2.4 and 2.6 and Q.1 (f) of October 2016.

- (e) What is self referential structure.

Ans. Please Refer to Q.1 (b) of October 2015 or Section 1.10.

- (f) What is polynomial? How is it represented?

Ans. Please Refer to Section 1.9.

- (g) Differentiate between malloc() and calloc() function.

Ans. Please Refer to Section 1.1.2.



(h) How to measure performance of an algorithm?

Ans. Please Refer to Section 1.3.

(i) Explain in brief, node structure of doubly circular linked list.

Ans. Please Refer to Section 3.3.

(j) What is strongly connected graph?

Ans. Please Refer to Section 6.1.

**2. Attempt any four of the following:**

[**4 × 4 = 16**]

(a) Write a 'C' program to accept and display polynomial.

Ans. #include<stdio.h>

```
#define SIZE 10
typedef struct
{
 float coeff;
 int power;
} POLY;
main()
{
 POLY P1[SIZE];
 int n, i;
 printf("\n Enter the no. of terms for the polynomial:");
 scanf("%d", &n);
 printf("\n Enter the terms in descending order of powers:");
 for(i = 0; i < n; i++)
 {
 printf("\n Enter coefficient for term %d", i+1);
 scanf("%f", &p[i].coeff);
 printf("\n Enter power for term %d", i+1);
 scanf("%d", &p[i].power);
 }
 printf("\n Entered polynomial is: \n");
 for(i= 0; i < n - 1; i++)
 {
 if(p[i].coeff !=0)
 printf("%2.2f[X^%d] + ", p[i].coeff, p[i].power);
 }
 printf("%d2.2f[X^%d]", p[i].coeff, p[i].power);
}
```

(b) What is graph? How is it represented? Explain any one technique in detail.

Ans. Refer Section 6.2.



(c) Explain different types of tree traversing techniques with an example.

Ans. Refer Section 5.6.

(d) Write a 'C' program to create and display circular singly linked list.

Ans. Refer Program 3.3.

(e) Write a function to insert a node at given position in doubly linked list.

Ans. 

```
typedef struct node
```

```
{
 int data;
 struct node *next;
 struct node *prev;
} DLLNODE;
DLLNODE * insert (DLLNODE *head, DLLNODE *newnode, int pos)
{
 int cnt = 1;
 DLLNODE *temp;
 temp = head;
 if(head == NULL)
 {
 printf("List is empty");
 return(head);
 }
 if(pos == 1)
 {
 newnode → next = head;
 head → prev = newnode;
 head = newnode;
 return(head);
 }
 else
 {
 while(temp! = NULL && cnt < pos)
 {
 temp = temp → next;
 cnt++;
 }
 if(temp == NULL)
 {
 printf("\n Wrong position");
 return(head);
 }
 else
 {
 newnode → next = temp → next;
 newnode → prev = temp;
 if(temp → next != NULL)
 (temp → next) → prev = newnode;
 temp → next = newnode;
 return(head);
 }
 }
}
```

**3. Attempt any four of the following:****[4 × 4 = 16]**

- (a) Write a function for the intersection of two singly linked list.

Ans.

```
typedef struct node
{
 int data;
 struct node *next;
} SLLNODE;

SLLNODE *intersection (SLLNODE *head1, SLLNODE *head2)
{
 SLLNODE *temp1, *temp2;
 SLLNODE *head3 = NULL, *last, *newnode;
 temp1 = head1;
 while (temp1 != NULL)
 {
 temp2 = head2;
 while(temp2 != NULL)
 {
 if (temp1 → data == temp2 → data)
 {
 newnode = (SLLNODE *) malloc(sizeof (SLLNODE));
 newnode → data = temp1 → data;
 newnode → next = NULL;
 if(head3 == NULL)
 {
 head3 = newnode;
 last = newnode;
 }
 else
 {
 last → next = newnode;
 last = newnode;
 }
 break;
 }
 else
 temp2 = temp2 → next;
 } // end of inner while
 temp1 = temp1 → next;
 } //end of outer while
 return (head 3);
}
```

- (b) Write a 'C' program for the implementation of dynamic queue.

Ans. Refer to Program 4.8.

- (c) Explain minimal spanning tree with an example.

Ans. Refer to Section 6.4.

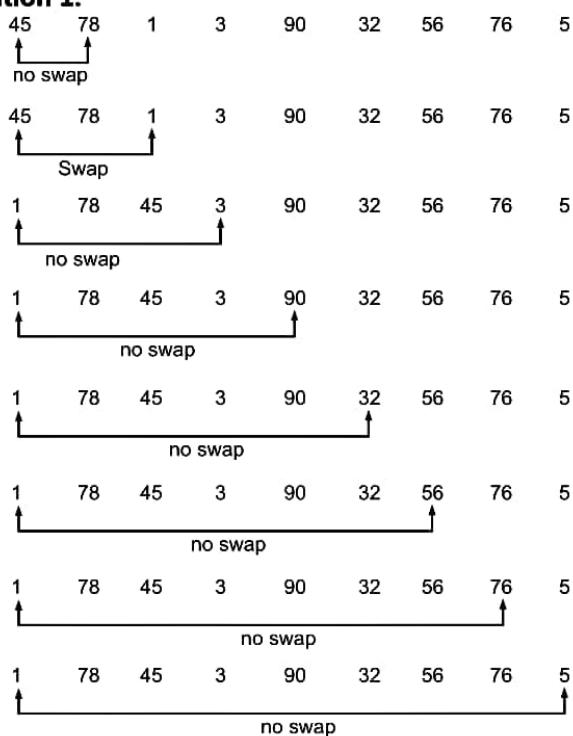
- (d) Write a function to count the number of leaf nodes in a tree.

Ans. Refer to Q.4 (e) of October 2016.

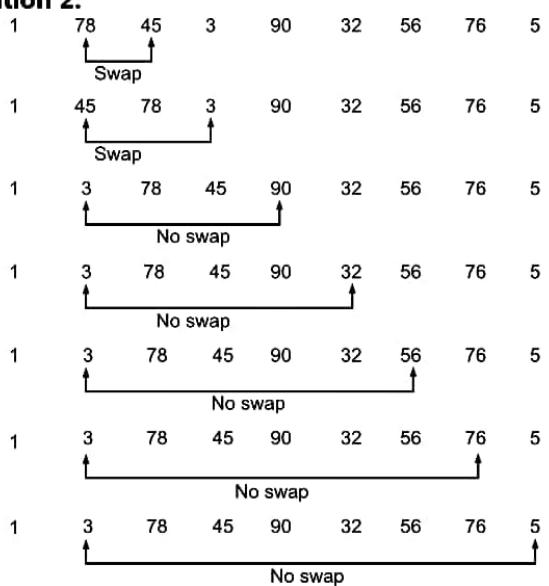


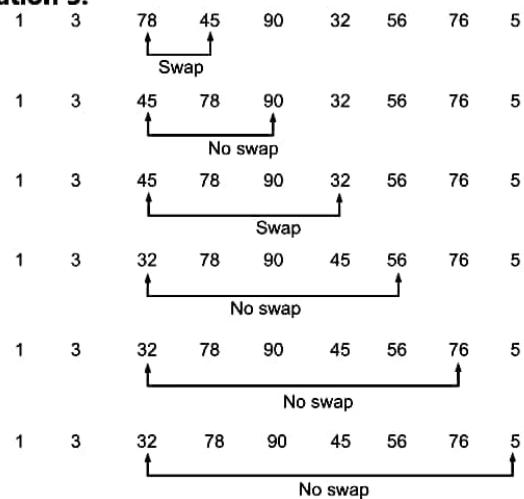
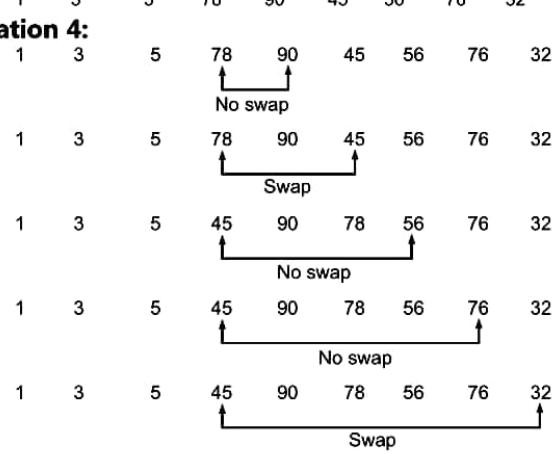
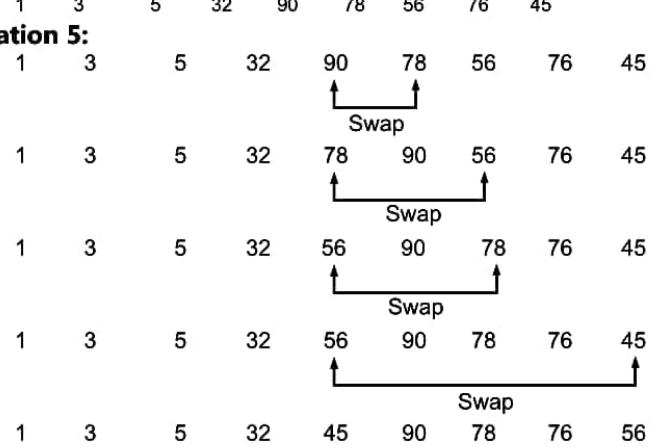
- (e) Sort the following data by using Selection Sort Technique:  
45, 78, 1, 3, 90, 32, 56, 76, 5

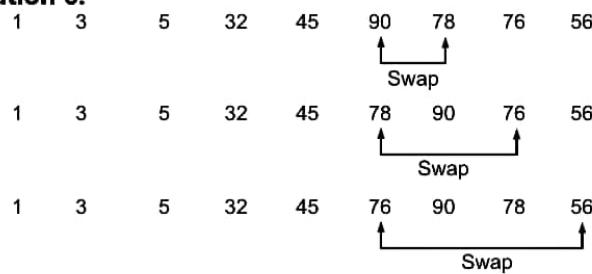
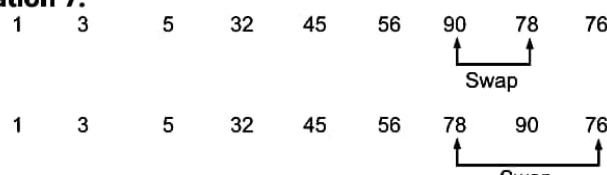
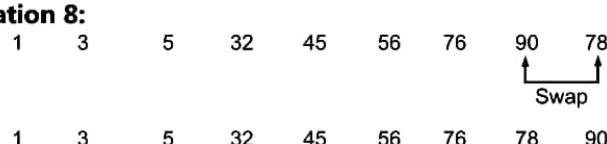
Ans. **Iteration 1:**



**Iteration 2:**



**Iteration 3:****Iteration 4:****Iteration 5:**

**Iteration 6:****Iteration 7:****Iteration 8:**

1    3    5    32    45    56    76    78    90

**4. Attempt any four of the following:**

[4 × 4 = 16]

- (a) Explain different types of asymptotic notations in detail.

Ans. Please Refer to Section 1.3.3.

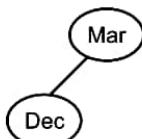
- (b) Construct binary search tree for the following data:

Mar, Dec, Jan, Sept, Jul, Oct, Feb, Aug

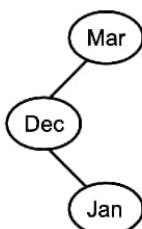
Ans. Insert Mar

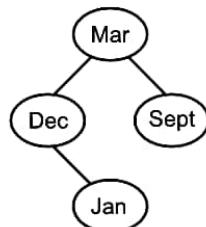
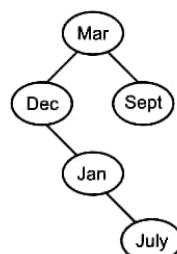
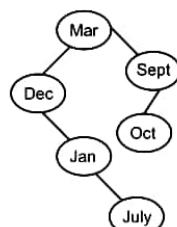
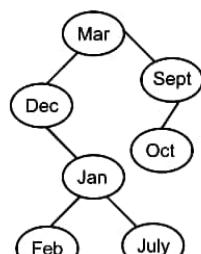
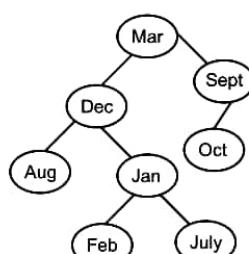


Insert Dec



Insert Jan



**Insert Sept****Insert Jul****Insert Oct****Insert Feb****Insert Aug**

- (c) Explain Heap sort technique with an example.  
Ans. Please Refer to Section 2.8.



- (d) Convert given infix expression into postfix expression:

(1)  $(A + B) \& C/D \% E$   
(2)  $(2+3)/1^*9-3$

Ans. (i)  $(A + B) * C/D \% E$

$$\underbrace{((A + B)}_{AB} \underbrace{*}_{+} \underbrace{C}_{\cdot} \underbrace{/}_{\cdot} \underbrace{D}_{\cdot} \underbrace{\%}_{\cdot} \underbrace{E}_{\cdot}$$

$$AB + C * D / E \%$$

(ii)  $(2 + 3)/1 * 9 - 3$

$$\underbrace{((2 + 3)}_{23} \underbrace{/}_{\cdot} \underbrace{1}_{\cdot} \underbrace{*}_{\cdot} \underbrace{9}_{\cdot} \underbrace{-}_{\cdot} \underbrace{3}_{\cdot}$$

$$23 + 1/9 * 3 -$$

- (e) Differentiate between Stack and Queue.

Ans. Refer Section 4.12.

**5. Attempt any four of the following:**

[**4 × 4 = 16**]

- (a) Explain DFS with an example.

Ans. Please Refer to Section 6.3.2.

- (b) What is searching? Explain binary search method with an example.

Ans. Please Refer to Sections 2.1 and 2.2.

- (c) Write a function which compares the contents of two stacks and display message accordingly.

Ans. `typedef struct`

```
{
 int data[50];
 int top;
} STATICSTACK;
void compare_stacks(STATICSTACK *S1, STATICSTACK *S2)
{
 int flag = 0;
 while((! is_empty(S1)) && (!is_empty(S2)) && flag==0)
 {
 if(pop(S1)!=pop(S2))
 {
 flag=1;
 break;
 }
 }
 if(flag == 1)
 printf("\n contents of stacks are not same");
 else if(isempty(S1))
 printf("\n second stack contains more elements");
 else
 printf("\n first stack contains more elements");
}
```

- (d) Explain LL and RR rotations with an example.

Ans. Please Refer to Section 5.7.1.

- (e) What is an algorithm? Explain its characteristics.

Ans. Please Refer to Sections 1.2.1 and 1.2.4.

