Course Code

AIT 121

Course Title

Data Structure Through C

Credit

3 (2+1)

Material Prepared

-by Prof. M. P. Raj

College of A. I. T., A.A.U., Anand

UNIT I

1. Need and importance of data structure, Types of Data structure
2. Operations on data structure, complexity analysis of algorithms
3. Recursion.

UNIT II

4. Arrays– row and columnar representation of Array
5. Sparse Array, Dynamic memory allocation
6. Stack its applications, PUSH POP, display
7. PEEP & CHANGE operations
8. Queues and its applications– Types of Queue
9. Queue Creation, insertion & display
10. Deletion and search operations in queue.

UNIT III

11. Introduction to linked lists - Singly,
12. Algorithms for creation, insertion, deletion & display, Search
13. Doubly and circularly linked lists
14. Algorithms for creation, insertion, deletion & display
15. Sorted linked list, Algorithms for creation, insertion, display

UNIT IV

16. Concepts, programming and operations of simple search & Binary search
17. Concepts, programming and applications of hashing technique.
18. Analysis of simple sorting techniques such as linear sort
19. Bubble sort, insertion sort ,Selection sort
20. Quick sort, Heap sort & merge sort.

UNIT V

21. Introduction to graphs representation Traversal-Depth first search, Breadth first search Adjacency matrix and list representation
22. Tree & Tree-Shortest path, minimum spanning tree – all pairs Shortest Path,
23. Transitive Closer, Splay Trees
24. Binary Trees Representation & operations: insert, delete Traversal –preorder, inorder, postorder.
25. N-ary trees: Definitions, balanced tree, definitions of B-tree.

UNIT - I

1) Why data structure is needed? What is the importance of data structure?

The possible ways in which the data items are logically related defines data structure.
Different logical relationship gives rise to different data structure.
Reasons are following.

1. To understand relationship between data elements, data structure is needed.
   There are two types of look out of data structure.
   - Physical lookout means how data store in memory.
   - Logical lookout means data are loaded & retrieve by the user.
2. Processes which are to be defined on the data can be confirmed.
3. Representative of data elements in the memory should be decided so that processing jobs can be coined out efficiently.
4. To decide the programming language used to solve the particular problem.

Fundamental Notations

- Primitive & composite data types.
- Time & space complexity of algorithm.

DATA
Data is a set of elementary items.

DATA STRUCTURE:

Data structure is a way of organizing data that considers not only the items stored but also relationship to each other.

OR

Data structure is a set of data elements arranged in logical order in which a only that it takes less time to access the data & requires less memory to store the data

OR

The possible way in which the data items (atoms) are logically related defined as DATA STRUCTURE.
e.g. Array, Pointer, file.

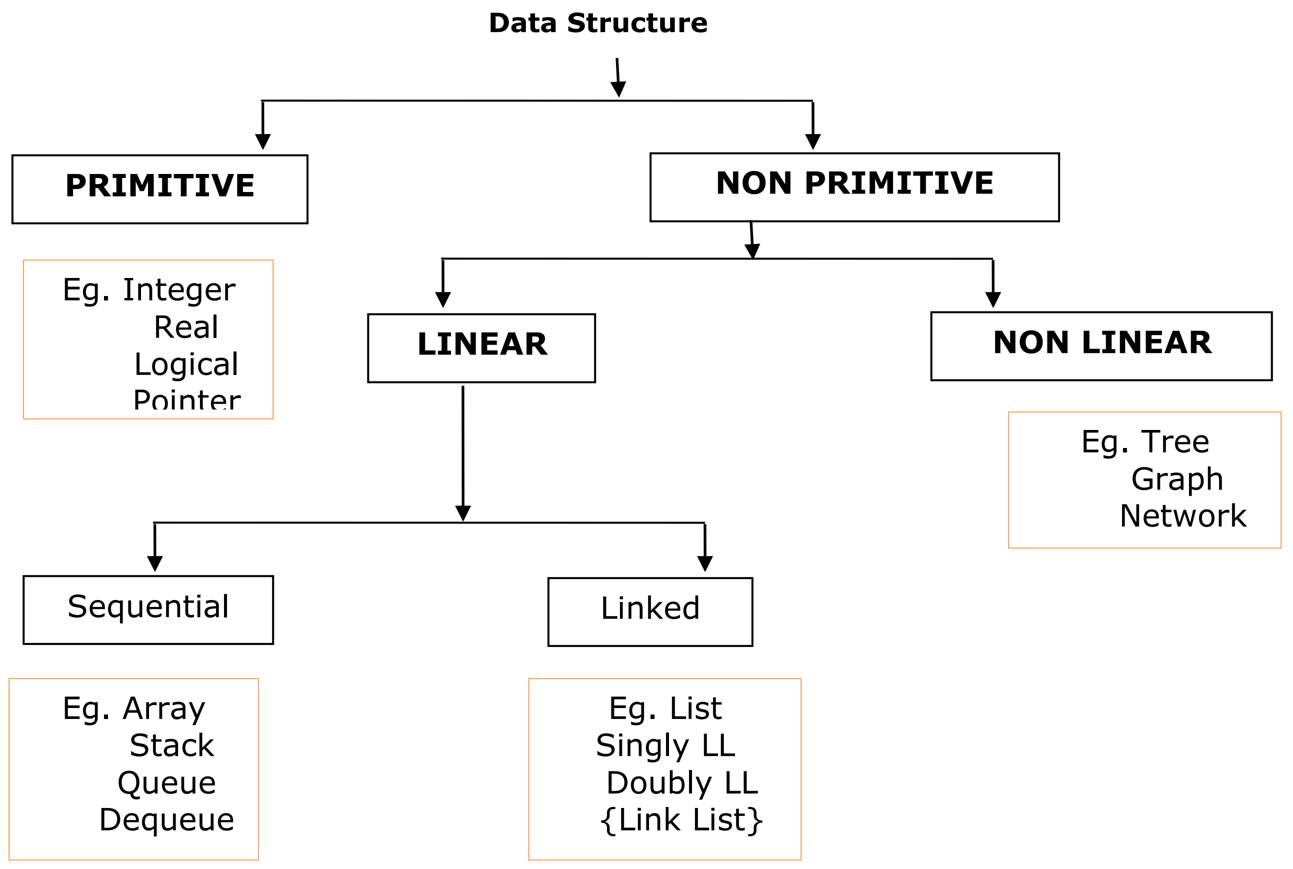
EXAMPLE

- The one dimensional array declared in C language.
```
int A[100];
```
- The one dimensional array declared in FORTRAN language.
```
Integer I, m(2c)
Read (*,10) (m(I),I=1,20)
10 FORMAT (2014)
```
- The RECORD used in language like COBOL, PASCAL etc.
```
01 STUD_REC
02 STUD_NO  PIC 9(3)
02 STUD_NAME  PIC ∗(20)
```
  Other examples are array, trees, graphs, linked list, stacks

A general understanding of data structure is essential for developing efficient algorithms.
There are some main points to take decision for designing any system.

1) The efficiency of a program with respect to its run time.
2) The efficiency of a program with respect to its utilization of main memory & secondary storage devices.
3) The development cost of a program.

Thus to take the decision about the efficiency of the algorithm, the thorough knowledge of programming language is not sufficient. But the knowledge of data structure is necessary.
One should examine the relative advantages & disadvantages of various data structure to select the appropriate data structure.



ALGORITHM

It is a logical module, designed to handle a specify problem relative to a particular data structure.

OR

Finite set of instructions that, if followed, accomplish a particular task.

OR

Algorithm is a sequence of instructions that act on some input data to produce some output in a finite number of steps

TYPES OF DATA STRUCTURE:--

Two types of data structure

1. Primitive data structure
2. Non primitive data structure or complex data structure

1) PRIMITIVE DATA STRUCTURE:
The data structures that typically are directly operated upon by machine level instructions are known as primitive data structure.

e.g. Integer, Real, Logical, Character, Pointer

2) NON PRIMITIVE DATA STRUCTURE:
It consists of structured set of primitive data structure.

e.g. Array {may consist of ordered set of integer}, stack, Queue, Linked List, Graph

Define following: (←)

1) STORAGE STRUCTURE:-
Representation of data structure in memory is known as STORAGE STRUCTURE.

2) FILE STRUCTURE:-
A storage structure representation in auxiliary memory is known as FILE STRUCTURE.

OPERATION ON PRIMITIVE DATA STRUCTURE:

There are mainly four operation performed on data structure.

a) CREATION
b) DESTROY
c) SELECTION
d) UPDATE

A. **CREATION**:-An operation frequently used in conjunction with data structure is one which create data structure. This operation is called CREATION operation.

e.g. Declaration of variables

- In C
  int n;
- In FORTRAN
  Integer I

B. **DESTROY**:-Another operation providing the complementary effect of a creation operation is one which destroys the data structure the operation is called DESTROY peration. Certain languages, such as FORTRAN, do not allow a programmer to destroy data structure once they have been created since all creations are performed at compile time & not at execution time In PASCAL, ALGOL, PL/I, data structure within a block is destroyed when the black is exited during execution. Destroy operation is not a necessary operation, it aids in the efficient use of memory.

C. **SELECTION**:-The most frequently used operation with data structure is one that the programmer uses to access data within a data structure. This type of operation is known as SELECTION.

D. **UPDATE**:-Another operation used conjunction with data structure is one which changes data in the structure. This operation will be called an UPDATE.

TIME AND SPACE

Time and Space requirements:-

Time & space analyses are also important for comparison of algorithm to determine the best one this selection will provide an introduction to the basic concepts in analyzing the time and space requirement of an algorithm. The analysis will initially emphasize timing analysis & then to a lesser extent space analysis.

- **Often more important than space complexity**
  - space available (for computer programs!) tends to be larger and larger
  - time is still a problem for all of us
- **3-4GHz processors on the market**
  - still …
  - researchers estimate that the computation of various transformations for 1 single DNA chain for one single protein on 1 TerraHZ computer would take about 1 year to run to completion

Basic Time Analysis of an Algorithm

ALGORITHM 1 Sum – VALUES

```
Step-1 [Sum the values in vector v]
          Sum ← 0
          Repeat for i=1,2,…..,N
          Sum ← Sum + v[i]
Step-2 [Finished]
          Exit
```

The above algorithm to sum the values in vector V that contains N value usually this is most easily done by isolating a particular operation, sometimes called on **active operation** that is central to the algorithm and that is executed eventually as often as any other. In the above example a good operation to isolate is the addition that occurs when another vector value is added to the partial sum. The other operation in the algorithm the assignments, the manipulations of the index 1, and the accessing of a value in the vector, occur no more than the addition of vector values these other operations are collectively called **book keeping operations** and are not generally counted. After, the active operation is isolated; the number of times that it is executed is counted.

ALGORITHM-2

MATRIX-MULTIPLICATION

```
Step 1  [multiply matrixes A & B. and store the result in matrix C]
            Repeat for i=1, 2, 3,…….., N
            Repeat for j=1, 2, 3,…….., N
            Sum ← 0
            Repeat for k=1, 2, 3,……., N
            Sum ← sum+ A [i, k] ∗ B [k, j]
            C [1 , j] ← sum
Step 2  [finished]
            Exit
```

In above algorithm given two dimensional square matrix A & B, each containing N rows & columns. This algorithm computes the matrix product and place the result in matrix C. i, j, k are integer variables.

ORDER NOTATIION

A notation has been developing to facilitate the handing of order of magnitude functions. A function f(n) is defined to be O(g(n)), that is, f(n)=O(g(n)) and is said to of order g(n), if there exists positive constant no (C) & C such where ln(n) is the natural logarithm n. Using the notation we have T_s (n) =O (n) & T_{mm} (n) =O (n^3) where T_s is the time for the vector sum algorithm & T_{mm} is the time for matrix multiplication algorithm.

MORE TIMING ANALYSIS

SEARCH ALGORITHM

```
Step 1  [search for the location of value × in vector V]
             Found ← False
             I ← 1
             Repeat while I<=N & not found if V [I] =X
             Then Found ← true
                       Location ← I
                       Exit
                 Else
                       I ← I+1
        Write ('Value of ', X,'NOT FOUND')
Step 2  [finished]
             Exit
```

In above algorithm given a vector V containing N element. This algorithm searches V for the location of given X. (found is a Boolean variable I & location are integer variable)

A reasonable active operation is the comparison of V & X. However a problem rises in counting the number of active operation executed, the answer depends on the index of the location containing X.

The best case is when X is equal to V [1] .Since only one Comparison is used the worst case is when X is equal to V[n], N Comparisons are used. Thus we obtain, T^{w}_{LS} (N) =O (N), T^{B}_{LS} (N) =O (1) where T^{B}_{LS} is the best case time for the linear search and T^{w}_{LS} is the worst case time for the linear search.

Question: - What time can be expected on the average?

We need to know the probability distribution for the value X in the vector (i.e. the probability of X occurring in each location). If we assumes the vector is not sorted, it is reasonable to assume that X is equally likely to be in each of the location But X is might not be the probability that X is in the list then using the above assumption, we have

- Probability X is in location I is q/N
- Probability X is not in the vector is 1-q

The average time is given by

T^A_{LS} (N) $\Sigma_{\sin s}$ (probability & situation S) * (time for situation S)
Where S is the set of all possible situations.
Thus for the above algorithm we have,
T^A_{LS} (N) = (Probability of x in location 1)* 1 + (Probability of x in location 2)* 2 ……
………. ……………
(Probability of in x in location N)* N + (Probability of x not in V)*N

$$= \sum_{<=1}^{N} \frac{q}{N} * S + (1-q) * N$$

$$= \frac{q}{N} \sum_{<=1}^{N} S + (1-q) * N$$

$$= \frac{q}{N} * \frac{N*(N+1)}{2} + (1-q) * N$$

$$= q * \frac{(N+1)}{2} + (1-q) * N$$

Thus, if q=1 then

$$TLS^A(N) = 1 * \frac{(N+1)}{2} + (1-1) * N$$

$$= \frac{N+1}{2} + 0 * N$$

$$= \frac{N+1}{2}$$

& if q=1 **/** 2 then,

$$TLS^A(N) = \frac{1}{2} * \frac{(N+1)}{2} + (1-\frac{1}{2}) * N$$

$$= \frac{N+1}{4} + \frac{1}{2} * N$$

$$= \frac{N+1}{4} + \frac{N}{2}$$

$$\sim \frac{3N}{4}$$

In either case T^A_{LS} (N) =O (N) unfortunately, as the above example indicates, the average Case timing analysis is generally more difficult than best case and the worst case. The difficulties begin with the need to obtain a reasonable probability distribution of the possible situation. For many problems this is difficult to do. As a result, only the worst Case timing analysis is done for many algorithms.

<u>ALGORITHMS 3</u>
ALTERNATE-SEARCH

```
Step 1 [Linear search when the value is initially placed at the End of the vector]
          V [N +1]← x
           I←1
          Repeat  while V[i] != x
          I←I+1
          If I = N + 1 Then
              Write ('value of ', x, 'not found')
          Else
            Location ← I
Step 2    [Finished]
          Exit
```

The number of comparisons between V & X is the same as for the previous algorithms [Algorithms-1], except that one more comparison is required for the present algorithms when x isn't in the vector. Thus using a count of exactions of the active operation, we would conclude that the first algorithm is marginally better but for algorithms with the same order. The constants associated with the largest term of the timing function should be estimated. In this example, it is easy to see that these are significantly more book-keeping operations for the algorithm than for the second. This result in a larger constant associated with the term for N in the equation for the time of the first algorithm. Thus the second algorithms will be more efficient.

SPACE ANALYSIS OF AN ALGORITHM:-

The analyses of the space requirements for an algorithm are generally easier than the timing analysis, but where necessary the same techniques are used. Usually, the space analysis is only done for the space to store the data values & hence does not include the space to store the algorithm itself. Also as for timing analysis the space function is usually expressed in Order Notation.

- **Space complexity = The amount of memory required by an algorithm to run to completion**
  - [Core dumps = the most often encountered cause is "memory leaks" – the amount of memory required larger than the memory available on a given system]
- **Some algorithms may be more efficient if data is completely loaded into memory**
  - Need to look also at system limitations.
  - E.g. classify 2GB of text in various categories [politics, tourism, sport, natural disasters, etc.] – can I afford to load the entire collection?
  -

1. **Fixed part: The size required to store certain data/variables, that is independent of the size of the problem:**
   - E.g. name of the data collection
   - Same size for classifying 2GB or 1MB of texts
2. **Variable part: Space needed by variables, whose size is dependent on the size of the problem:**
   - E.g. actual text
   - Load 2GB of text VS. Load 1MB of text

- **S(P) = c + S(instance characteristics)**
  - c = constant

Example:

```
float sum (float* a, int n)
{
      float s = 0;
   for(int i = 0; i<n; i++) {
    s+ = a[i];
   }
   return s;
}
```

Space? one word for n, one for a [passed by reference!], one for i → constant space!

Cases to Consider During Analysis

- **Best Case Input**
  The input set that allows an algorithm to perform most quickly. With this input the algorithm takes shortest time to execute, as it causes the algorithm to do the least amount of work.
  - Since the best case for an algorithm would usually be very small and frequently constant value, a best case analysis is often not done.
- **Worst Case Input**
  The input set that allows an algorithm to perform most slowly. Worst case is an important analysis because it gives us an idea of the most time an algorithm will ever take.
- **Average Case Input**

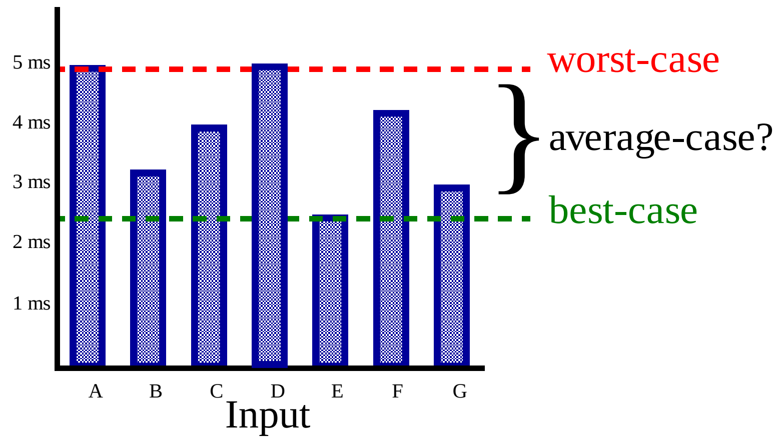
This represents the input set that allows an algorithm to deliver an average performance. Four step processes is required for this case.

- Determine the number of different groups into which all possible input set can be divided.
- Determine the probability that the input will come from each of these groups.

- Determine how long the algorithm will run for each of these groups. All inputs in each group should take same amount of time, and if they do not, the group must be split into two separate groups.
    - Formula : A(n) = ∑ pi * ti

  Where,

  n = Size of input.
  m = Number of groups.
  pi = Probability that the input will be from group i.
  ti = Time that the algorithm takes for input from group i.



Recursion

A method of defining functions in which the function being defined is applied/called within its own definition; specifically it is defining an infinite statement using finite components. The term is also used more generally to describe a process of repeating objects in a self-similar way.

The factorial function, whose domain is the natural numbers can be recursively defined as:

$$\text{FACTORIAL (n)} = \begin{cases} 1 \text{ if } n=0 \\ N^* \text{ Factorial (N-1),otherwise} \end{cases}$$

The general algorithm model for any recursive procedure contains the following steps.

1. [Prologue] saves the parameters, local variables & return address.
2. [Body] if the base criterion has been reached , then perform the final computation & go to step 3; otherwise perform the practical computation & go to step 1.
3. [Epilogue] Restore the most recently saved parameters. Local variables & return address , go to this return address

Iterative process:-

An iterative process has mainly 4 parts.

1. Initialization
2. Decision
3. Computation
4. Update

1. It is possible to transform mechanically any primitive recursive function into an equivalent iterative process but it is not possible in case for non- primitive recursive function.
   ex.:- For Ackermann’s function an iterative solution either does not exist or not easily found but it is easily found by recursive process.
2. In short the recursive solution may be much simpler (through sometime more time coming) then its iterative counter part.

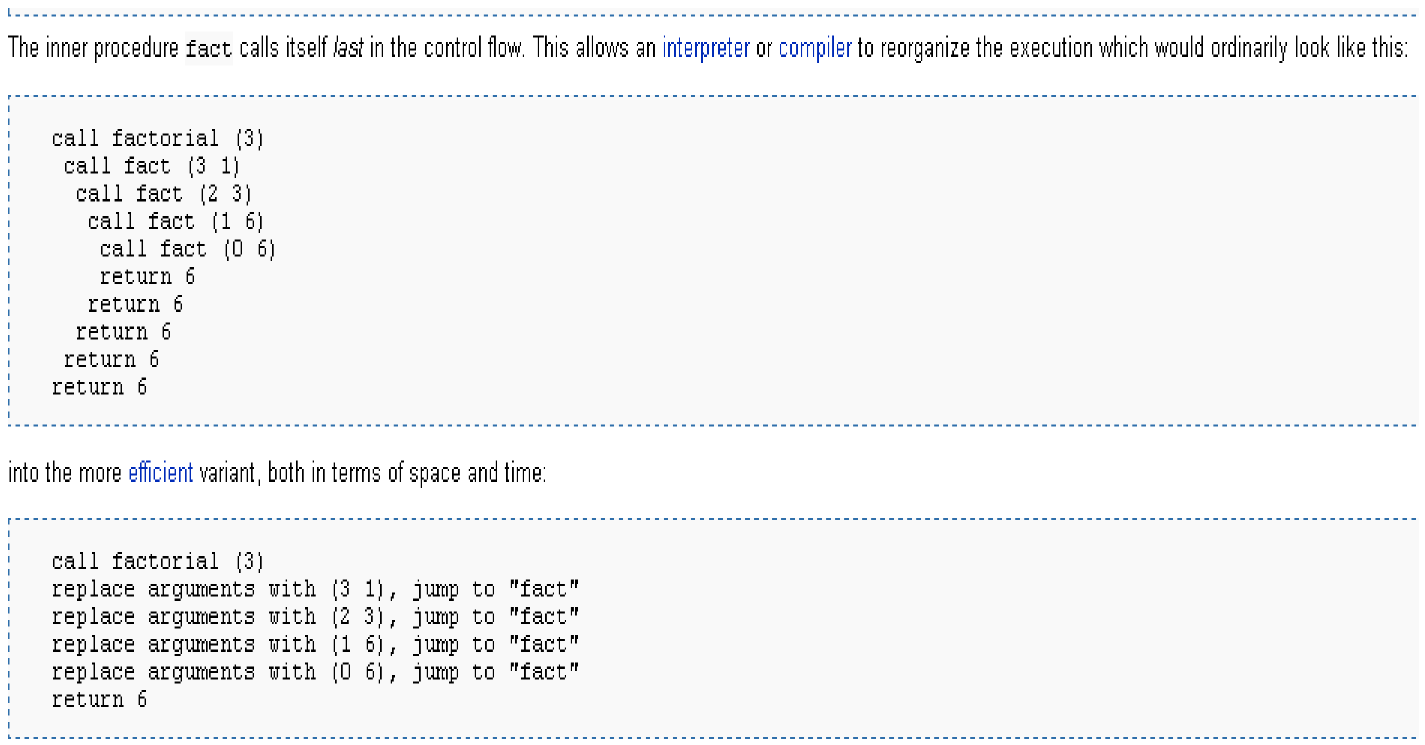
Algorithm to find factorial by recursive process:--

Give an integer N, this algorithm computer N! The stack A is used to used to store an activation record associated with each recursive call. Each activation record contains the current value of N & the current return address RET-ADPR TEMP_REC is also a record which contains two variables (PARM & ADDRESS). This temporary record is required to stimulate the proper transfer of control from one activation record of algorithm FACTORIAL to another whenever a TEMP_REC is placed on stack A, copies of PARM & ADDRESS are pushed onto A and assigned to N & RET-ADPR respectively. Top points to the top element of A and its value is set to the main calling address PRAM is set to the initial value of N.

```
1. [SAVE N & RETURN ADDRESS]
   Call PUSH (A , Top ,TEMP_REC)
2. [IS THE BASE CRITERIA FOUND?]
        If N=0 then factorial ← 1
              Go to step 4
        Else
              PARM ← N-1
              Address ← step 3
              Go to step 1
3. [CALCULATION N!]
              Factorial ← Factorial
              (The factorial of N)
4. [RESTORE PREVIOUS N & RETURN ADDRESS]
              TEMP_REC ← Pop (A, TOP)
                     (i.e. PARM ← N, ADDRESS ← RET_APDR POP stack)
                     Go to ADDRESS.
```

Program to find factorial

```
#include <stdio.h>
int fact(int);

void main()
{
  int n,fac;

  printf("\nEnter value of n = ");
  scanf("%d",&n);

  fac= fact(n);
  printf("\nFactorial of %d is %d",n,fac);
}

int fact(int n)
{
       if (n<=1)
         return 1;
       else
         return n * fact(n-1);
}
```



The inner procedure `fact` calls itself *last* in the control flow. This allows an interpreter or compiler to reorganize the execution which would ordinarily look like this:

```
call factorial (3)
 call fact (3 1)
  call fact (2 3)
   call fact (1 6)
    call fact (0 6)
    return 6
   return 6
  return 6
 return 6
return 6
```

into the more efficient variant, both in terms of space and time:

```
call factorial (3)
replace arguments with (3 1), jump to "fact"
replace arguments with (2 3), jump to "fact"
replace arguments with (1 6), jump to "fact"
replace arguments with (0 6), jump to "fact"
return 6
```

<u>Example 2: Print number 1 to 10</u>

With Starting and ending limits.

```
#include <stdio.h>
 void main()
 {
       clrscr();
       printnum(1,10);
 }

 printnum(int n, int end)
 {
  if(n>end)
     return;
  else
  {
     printf("\n%d",n);
     getch();
     printnum(n+1,end);
  }
 }
```

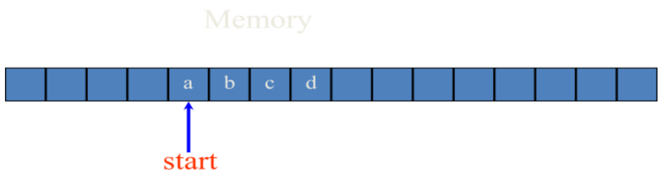
UNIT – II

Array

An array in C Programming Language can be defined as number of memory locations, each of which can store the same data type and which can be accessed via the same variable name.
An array is a collective name given to a group of similar quantities. These similar quantities could be percentage marks of 100 students, number of chairs in home, or salaries of 300 employees or ages of 25 students. Thus an array is a collection of similar elements. These similar elements could be all integers or all floats or all characters etc. Usually, the array of characters is called a "string", where as an array of integers or floats is called simply an array.

Arrays and pointers have a special relationship as arrays use pointers to reference memory locations.

Declaration of an Array: `data_type array_name[sizeofarray];`



- 1-dimensional array x = [a, b, c, d]
- space overhead = 4 bytes for start
- map into contiguous memory locations
- location(x[i]) = start + i

Multidimensional Arrays: In C Language one can have arrays of any dimensions.
`data_type array_name[size1][size2]...[sizeN];`

Data is often available in tabular form. Tabular data is often represented in arrays. Matrix is an example of tabular data and is often represented as a 2-dimensional array.

- Matrices are normally indexed beginning at 1 rather than 0
- Matrices also support operations such as **add**, **multiply**, and **transpose**, which are NOT supported by C's 2D array.

It is possible to **reduce time and space** using a **customized representation** of multidimensional arrays

e.g. int a[3][4]

may be shown as a table

a[0][0]	a[0][1]	a[0][2]	a[0][3]	**row 1**
a[1][0]	a[1][1]	a[1][2]	a[1][3]	**row 2**
a[2][0]	a[2][1]	a[2][2]	a[2][3]	**row 3**
Col 1	**Col 2**	**Col 3**	**Col 4**	

2D Array Representation in C

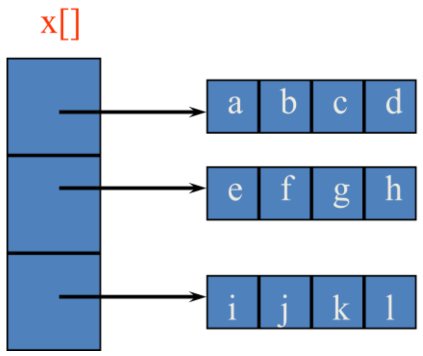
2-dimensional array x

a, b, c, d
e, f, g, h
i, j, k, l

view 2D array as a 1D array of rows

x = [row0, row1, row 2]
row 0 = [a, b, c, d]
row 1 = [e, f, g, h]
row 2 = [i, j, k, l]

and store as 4 1D arrays

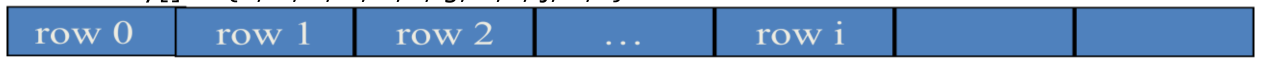


4 separate
1-dimensional arrays

- space overhead = overhead for 4 1D arrays
  = 4 * 4 bytes
  = 16 bytes
  = (number of rows + 1) x 4 bytes
- This representation is called the array-of-arrays representation.
- Requires contiguous memory of size 3, 4, 4, and 4 for the 4 1D arrays.
- 1 memory block of size number of rows and number of rows blocks of size number of columns

Row-Major Mapping

- Example 3 x 4 array:
  a b c d
  e f g h
  i j k l
- Convert into 1D array y by collecting elements by rows.
- Within a row elements are collected from left to right.
- Rows are collected from top to bottom.
- So y[] = {a, b, c, d, e, f, g, h, i, j, k, l}



Locating Element x[i][j]

- assume x has r rows and c columns
- each row has c elements
- i rows to the left of row i
- so ic elements to the left of x[i][0]
- x[i][j] is mapped to position ic + j of the 1D array

Space Overhead

4 bytes for start of 1D array + 4 bytes for c (number of columns)
= 8 bytes
It needs contiguous memory of size **rc**.

Column-Major Mapping

a b c d
e f g h
i j k l

- Convert into 1D array y by collecting elements by columns.
- Within a column elements are collected from top to bottom.
- Columns are collected from left to right.
- So y = {a, e, i, b, f, j, c, g, k, d, h, l}

Row- and Column-Major Mappings

2D Array:- int a[3][6];

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

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17

(a) Row-major mapping

0	3	6	9	12	15
1	4	7	10	13	16
2	5	8	11	14	17

(b) Column-major mapping

Row- and Column-Major Mappings

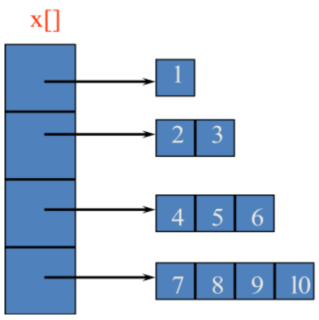
- Row-major order mapping functions

  map(i_1,i_2) = $i_1u_2+i_2$ for 2D arrays

  map(i_1,i_2,i_3) = $i_1u_2u_3+i_2u_3+i_3$ for 3D arrays

Sparse Array or Irregular 2D Arrays

Irregular 2D Arrays



Irregular 2-D array: the length of rows is not required to be the same.

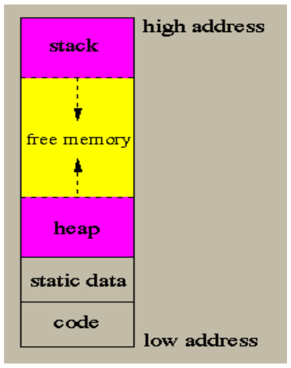
Sparse Array

A **sparse array** is an array in which most of the elements have the same value. The occurrence of zero elements in a large array is inefficient for both computation and storage. An array in which there is a large number of zero elements is referred to as being sparse.

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

int count;

 void main()
{
        int *data, *tuple, row, col, tot, i, j, k =0, x = 0, y = 0;

        printf("Enter the no of rows & columns:");
        scanf("%d%d", &row, &col);

        tot = row * col;
        data = (int *) malloc(sizeof (int) * tot);
        for (i = 0; i < tot; i++)
            {
                if (y % col == 0 && y != 0)
      {
                        y = 0;
                        x++;
                }
                printf("data[%d][%d]:", x, y);
                scanf("%d", &data[i]);
                if (data[i])
                        count++;
                y++;
            }
        tuple = (int *) malloc(sizeof (int) * (count + 1) * 3);

        /* store row, column & count in 3-tuple array - in 1st row */
        tuple[k++] = row;
        tuple[k++] = col;
        tuple[k++] = count;
        x = y = 0;
        for (i = 0; i < tot; i++)
        {
                if (y % col == 0 && y != 0)
      {
                        y = 0;
                        x++;
                }

                /* store row, column and non-zero val in 3 tuple */
                if (data[i] != 0) {
                        tuple[k++] = x;
                        tuple[k++] = y;
                        tuple[k++] = data[i];
                }
                y++;
          }

        printf("Given Sparse Matrix has %d non-zero elements\n", count);
        x = y = 0;

        /* printing given sparse matrix */
        for (i = 0; i < tot; i++)
        {
                if (y % col == 0 && y != 0)
      {
                        y = 0;
                        x++;
                        printf("\n");
                }
                printf("%d ", data[i]);
                y++;
        }
        printf("\n\n");

        /* 3-tuple represenation of sparse matrix */
        printf("3-Tuple representation of Sparse Matrix:\n");
        for (i = 0; i < (count + 1) * 3; i++)
        {
                if (i % 3 == 0)
                        printf("\n");
                printf("%d ", tuple[i]);
        }
   getch();
  }
```

Application to storage allocation, de-allocation and garbage

Consider a typical storage organization of a program:



All dynamically allocated data are stored in the heap. These are the data created by malloc in C. Heap is a vector of bytes (characters) and end_of_heap a pointer to the first available byte in the heap.

The C programming language manages memory statically, automatically, or dynamically. Static-duration variables are allocated in main (fixed) memory and persist for the lifetime of the program; automatic-duration variables are allocated on the stack and come and go as functions are called and return. For static-duration and automatic-duration variables, the size of the allocation is required to be compile-time constant. If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by allocating it from the heap, an area of memory structured for this purpose. In C, the library function malloc is used to allocate a block of memory on the heap. The program accesses this block of memory via a pointer that malloc returns. When the memory is no longer needed, the pointer is passed to ***free*** which deallocates the memory so that it can be used for other purposes.

The malloc function is one of the functions in standard C to allocate memory. Its function prototype is

```
void *malloc(size_t size);
```

which allocates *size* bytes of memory. If the allocation succeeds, a pointer to the block of memory is returned, otherwise a NULL pointer is returned.

Memory allocated via malloc is persistent: it will continue to exist until the program terminates or the memory is explicitly deallocated by the programmer (that is, the block is said to be "freed"). This is achieved by use of the free function. Its prototype is

```
void free(void *pointer);
```

which releases the block of memory pointed to by pointer. pointer must have been previously returned by malloc, calloc, or realloc and must only be passed to free once.

The standard method of creating an array of 10 int objects:

```
int array[10];
```

However, if one wishes to allocate a similar array dynamically, the following code could be used:

```
/* Allocate space for an array with ten elements of type int. */
int *ptr = malloc(10 * sizeof (int));
if (ptr == NULL) {
  /* Memory could not be allocated, the program should handle the error here as appropriate. */
} else {
  /* Allocation succeeded. Do something. */
  free(ptr); /* We are done with the int objects, and free the associated pointer. */
  ptr = NULL; /* The pointer must not be used again, unless re-assigned to using malloc again. */
}
```

malloc returns a null pointer to indicate that no memory is available, or that some other error occurred which prevented memory being allocated.

realloc function is used to resize already allocated memory block without changing or disturbing contents.

calloc function is used to allocate memory from C memory heap.

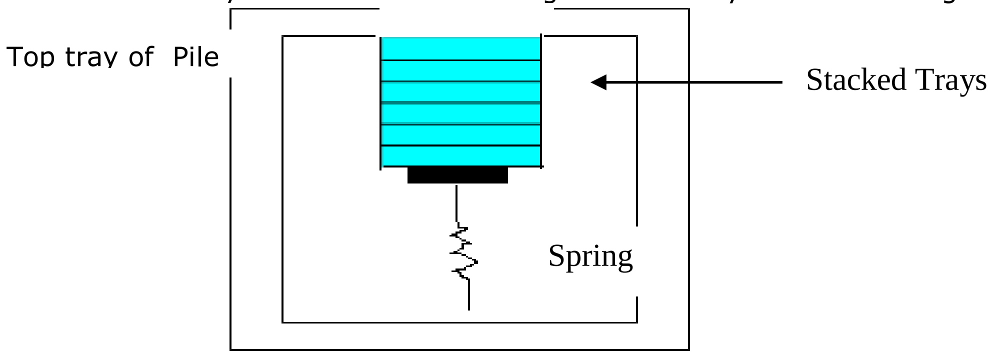
Automatic Garbage Collection

Heap-allocated records that are not reachable by any chain of pointers from program variables are considered garbage. All other data are 'live'. This is a very conservative approach but is always safe since you can never access unreachable data. On the other hand, it may keep objects that, even though reachable, they will never be accessed by the program in the future. With automatic storage management, user programs do not reclaim memory manually. When the heap is full, the run-time system suspends the program and starts garbage collection. When the garbage collection is done, the user program resumes execution.

STACK :-

Stack is the one of the most important linear data structures of variable size.
An important subclass of linear list permits the insertion and deletion of an element to occurs only at one end. A linear list belonging to this subclass is called a stack. The insertion operation is referred to as “PUSH” and the deletion operation is referred to as “POP”. The most and least accessible elements in a stack are known as the top and bottom of the stack. Since insertion and deletion operations are performed at one end of a stack, the element can only be removed in the opposite order from that in which they are added to the stack, such linear list is frequently referred to as a LIFO list.

A common example of a stack phenomenon, which permits the selection of only its end element, is a pile of trays in a cafeteria. These are supported by some kind of spring action in such a manner that a person deciding a tray finds that only one is available to him or her at the surface of the tray counter. The removed of the top tray causes the load on the spring to be lighter and the tray to appear at the surface of the counters. A tray at which is placed on the pile causes the entire pile to be pushed down and that tray to appear above the tray counter. Such a arrangement of trays is shown in fig.



A cafeteria - tray holder

Another familiar example of a stack is a railway system, for shunting cars, as shown in fig-2. In this system, the last railway car to be placed on the stack is the first to leave. [Using repeatedly the insertion and deletion operations permits the cars to be assigned on the output railway line in various orders]

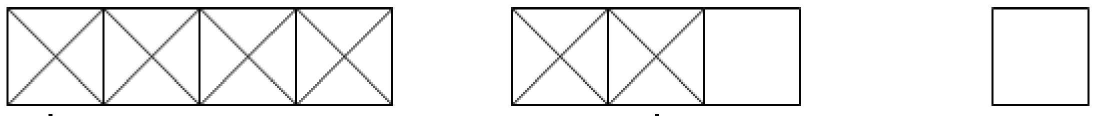
One more example of stack may be seen in big godowns, where the units which come last are taken first.

❖ **<u>Operations on Stacks</u> :-**

The operations on a stack are stimulated by using a vector consisting of some large no. of elements which should be sufficient in no to handle all possible instructions likely to be made to the stack.
A pointer top keeps track of the top element in the stack. Initially, when the stack is empty TOP has a value of zero and when the stack contains a single element, TOP has a value of ‘one’ & so on. Each, time a new element is inserted in the stack, the pointer is incremented by ‘one’ before the element is placed on the stack. The pointer is decremented by ‘one’ each time a deletion is made from the stack.

A more suitable representation of a stack is given below.

Deletion 🡐====
Insertion ===🡒



Alternative representation of a stack.

The rightmost occupied element of the stack represents its top element. The leftmost element of the stack represents its bottom element.
The algorithm for inserting an element in a stack follows.

Procedure PUSH (s, TOP, x)

This procedure inserts an element x to the top of a stack which is represented by a vector S containing N element with a pointer TOP denoting the top element in the stack.

```
1.  [ CHECK FOR STACK OVERFLOW ]
            if TOP ≥ N then
            write ( ‘STACK OVERFLOW’ )
        Return.
2.  [ INCREMENT TOP ]
        TOP ← TOP + 1
3.  [ INSERT ELEMENT ]
        S[ TOP ] ← x
4.  [ FINISHED ]
        Return.
```

The first step of this algorithm checks for an overflow condition. If such a condition exits , then the insertion cannot be performed and an appropriate error message results.

Programming structure of PUSH

```
struct node
{
    int info;
    struct node *link;
} *top=NULL;

void push()
{
    struct node *tmp;
    int pushed_item;
    tmp = (struct node *)malloc(sizeof(struct node));
    printf("Input the new value to be pushed on the stack : ");
    scanf("%d",&pushed_item);
    tmp->info=pushed_item;
    tmp->link=top;
    top=tmp;
}/*End of push()*/
```

The algorithm for deleting on element from a stack is given as follows.

Procedure POP (s, TOP)

This function removes the top element from a stack which is represented by a vector s & return this element. TOP is a pointer to the top element of the stack.

```
1. [CHECK FOR STACK UNDERFLOW ON POP]
    if TOP = 0 then
     Write (‘STACK UNDERFLOW ON POP’)
```

```
        Take action in response to underflow.
        Exit.
2. [DECREMENT POINTER]
        TOP ← TOP – 1
3. [RETURN FORMER TOP ELEMENT OF STACK]
        Return (S [TOP + 1])
```

An underflow condition is checked for in the first step of the algorithm. If there is an underflow, then some appropriate action should take place.

Programming structure of POP:-

```
void pop()
{
        struct node *tmp;
        if(top == NULL)
                printf("Stack is empty\n");
        else
    {
                tmp=top;
                printf("Popped item is %d\n",tmp->info);
                top=top->link;
                free(tmp);
        }
}
```

Function PEEP (S, TOP, I)

Given a vector S representing a sequentially allocated stack and a pointer TOP denoting to top element of stack, this function returns the value of the Ith element from the top of the stack.
The element is not deleted by this function

```
1. [CHECK FOR STACK UNDERFLOW]
            If TOP – I + 1 ≤ 0  then
                Write ('STACK UNDERFLOW ON PEEP');
                Take action in response to underflow
                Exit
2. [RETURN ITH ELEMENT FROM TOP OF STACK]
            Return (S [TOP – I + 1])
```

FUNCTION CHANGE(S, TOP, X, I)

A vector S represents a sequentially allocated stack and a pointer TOP denotes the top element of the stack.

This procedure changes the value of the Ith element from the top of the stack to the value contained in X.

```
1 [CHECK FOR STACK UNDERFLOW]
        IF TOP-I+1 <= 0 THEN
                WRITE('STACK UNDERFLOW ON CHANGE')
                Take action in response to underflow
          RETURN
2 [CHANGE Ith ELEMENT FROM TOP OF STACK]
        S[TOP-I+1] ← X
3 [FINISHED]
        RETURN
```

Applications of STACK

Main three application of STACK are as follows:

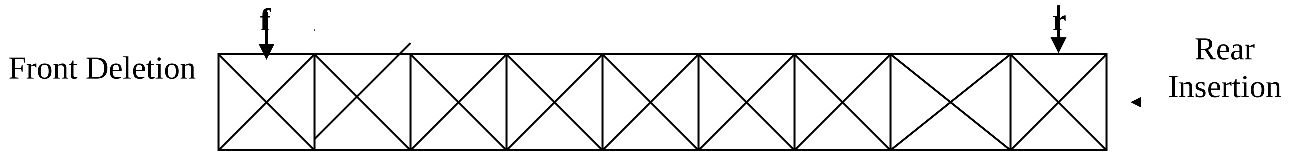
The first application deals with recursion. Recursion is an important facility in many programming languages. There are many problems whose algorithm description is best described in recursive manner.

The second application of a STACK is classical. It deals with the compilation of infix expression into object code.

The section ends with a brief discussion of stack machines, certain computer perform stack operations on hardware or machine level and these operations enable insertion to and deletion form a stack to be made very rapidly.

QUEUES

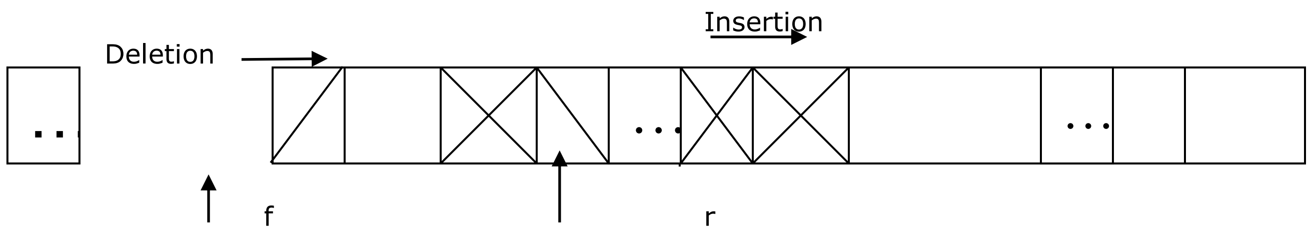
Another important subclass of lists permits deletions to be performed at one end of a list and insertions at the other. The information in such a list is processed in the same order as it was received, that is, on a first in first out (FIFO) or a first-come first served (FCFS) basis. This type of list is frequently referred to as a queue.
Following fig. is a representation of a queue illustrating how an insertion is made to the right of the right most elements in the queue and how a deletion consists of deleting the leftmost element in the queue.
In case of a queue, the updating operation may be restricted to the examination at the last or end element. If no such restriction is made, any element in the list can be selected.
e.g.: A check out line at a supermarket cash counter. The first person in the line is the first to check out.



The vector is assumed to consist of a larger no of element the vector representation of a queue requires pointers f and r which denote the pointers of its front and rear elements, respectively, an illustration of such an allocation scheme are given in fig.
An algorithm for insertion an element in a queue is given as follows.

PROCEDURE QINSERT (Q, F, R, N, Y)

F and R are two pointers to the front and rear element of a queue. A queue 'Q' consisting of N elements. This procedure insert Y at the rear of the queue, prior to the first invocation of the procedure, F and R have been set to zero.



```
1. [OVERFLOW?]
        If R>=N then
            Write (overflow);
            Return
2. [INCREMENT REAR POINTER]
        R ← R+1
3. [INSERT ELEMENT]
        Q[R] ← Y
4. [IS FRONT POINTER PROPERTY SET?]
        If f=0 then
            F ← 1
            Return
```

Programming structure of Qinsert:-

```
struct node
{
        int info;
        struct node* next;
} *front=NULL,*rear=NULL;

void QInsert()
{
        int no;
        struct node *tmp;
         tmp=(struct node*)malloc(sizeof(struct node));
         printf("\n Enter a value = ");
         scanf("%d",&no);
        tmp->info=no;
        tmp->next=NULL;
        if(rear==NULL||front==NULL)
         front=tmp;
        else
         rear->next=tmp;

        rear=tmp;
}
```

The following algorithm deletes an element from a queue.

Function QDelete (Q, F, R)

F and R the two pointers to the front and rear elements of the queue "Q" to which they correspond. Y is a temporary variable. This function deletes and returns the last element of the queue.

```
1. [UNDERFLOW]
        If F=0 then
                Write ('UNDERFLOW')
        Return (0); (0 elements an empty queue)
2. [DELETE ELEMENT]
        Y ←Q[F]
3. [QUEUE EMPTY]
     If F = R then
             F ←  R ←  0
     Else
             F  ← F+1 (increment front pointer)
4. [RETURN ELEMENT]
        Return (Y);
```

Programming structure of the QDelete:-

```
int QDelete ()
{
 struct node *p;
 int elt;
 if(front==NULL||rear==NULL)
 {
      printf("\nUnder Flow");
      getch();
      exit(0);
 }
 else
  {
      p=front;
      elt=p->info;
      front=front->next;
      free(p);
  }
 return(elt);
}
```

This pair of algorithms can be very wasteful of storage if the front pointer F never manages to the rear pointer. Actually, an arbitrarily large amount of memory would be required to accommodate the elements. This method of performing operations on a queue should only be used when the queue is empty at certain intervals. An overflow occurs in a queue when we try to insert N+1 element. Even though all queue locations are not being used.

CIRCULAR QUEUE

A more suitable method of representing a queue , which presents an excessive use of memory is to arrange the elements Q[1],Q[2]........., Q[N] in a circular fashion with Q[1] Following Q[N].

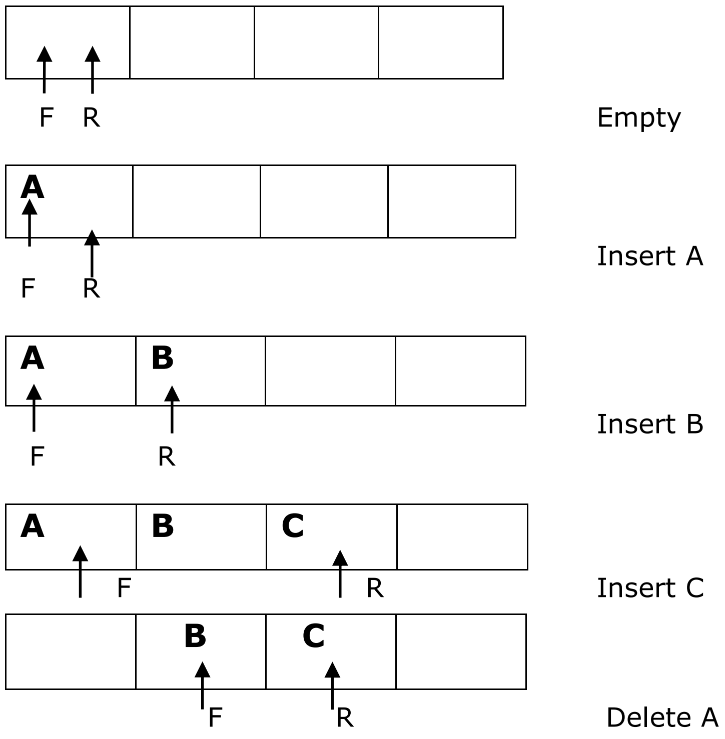
This can be represented in fig.3 . The insertion and deletion algorithm for a circular queue can raw be formulated.

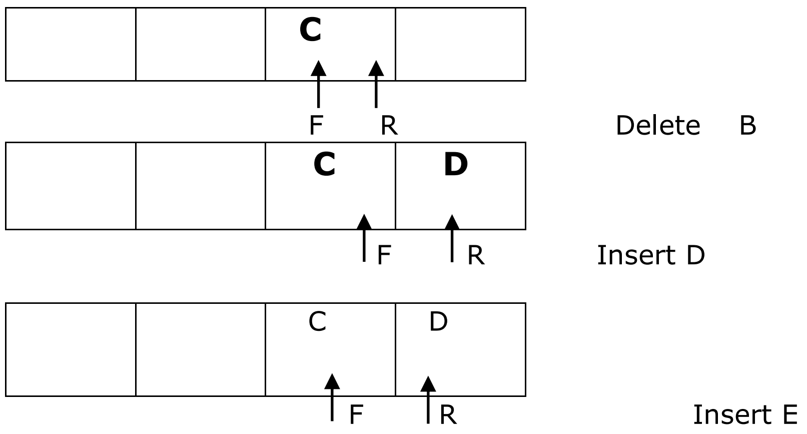
Procedure QInsert(F,R,Q,N,Y)

- Given pointers to the front and rear of a circular queue F&R.
- A vector Q consisting of N element and an element Y

  This procedure inserts Y at the rear of the queue F & R are set to zero.

```
1. [RESET REAR POINTER?]
      IF R=N THEN
           R ← 1
      ELSE
           R ← R+1
2. [OVERFLOW?]
      IF F=R THEN
           Write ('overflow')
           RETURN
3. [INSERT ELEMENT]
      Q[R] ← Y
4. [IS FRONT POINTER PROPERLY SET?]
      If F=0 THEN
           F ← 1
           RETURN
```





Trace of operation on a simple queue.

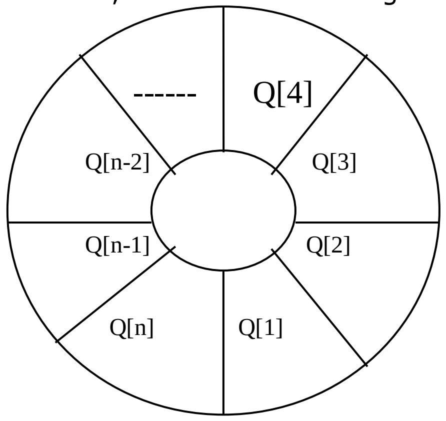
Function QDelete(F, R, Q, N)

Given F and R pointers to the Front and rear of a circular queue. A vector Q consisting of N elements. Y is temporary variable.

This function deletes & returns the last element of the queue

```
1. [UNDERFLOW?]
       If   F = 0 THEN
            Write ('UNDERFLOW')
            Return (0)
2. [DELETE ELEMENT]
       Y ← Q [F]
3. [QUEUE EMPTY?]
       If   F = R THEN
            F ← R ← 0
            Return(Y)
4. [INCREMENT FRONT POINTER]
         If F = N   THEN
              F ← 1
          Else
              F ← F+1
         Return(Y)
```

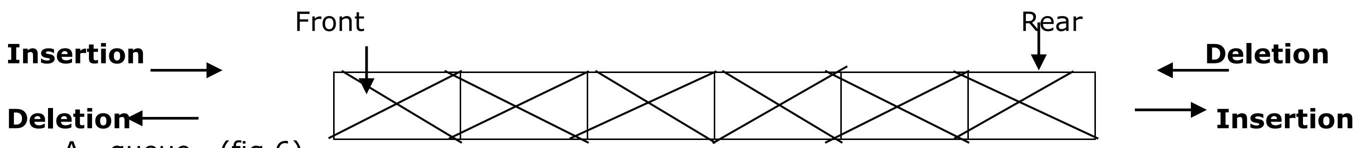
Consider an example of a circular queue that contains a maximum of four elements. It is required to perform a no. of insertion & deletion on operation on an initially empty queue. A trace of the queue contents, which is not shown as circular, for convenience is given below



A single queue has been describe as behaving in a first out manner in the sense that each deletion removes the oldest remaining item in the structure.

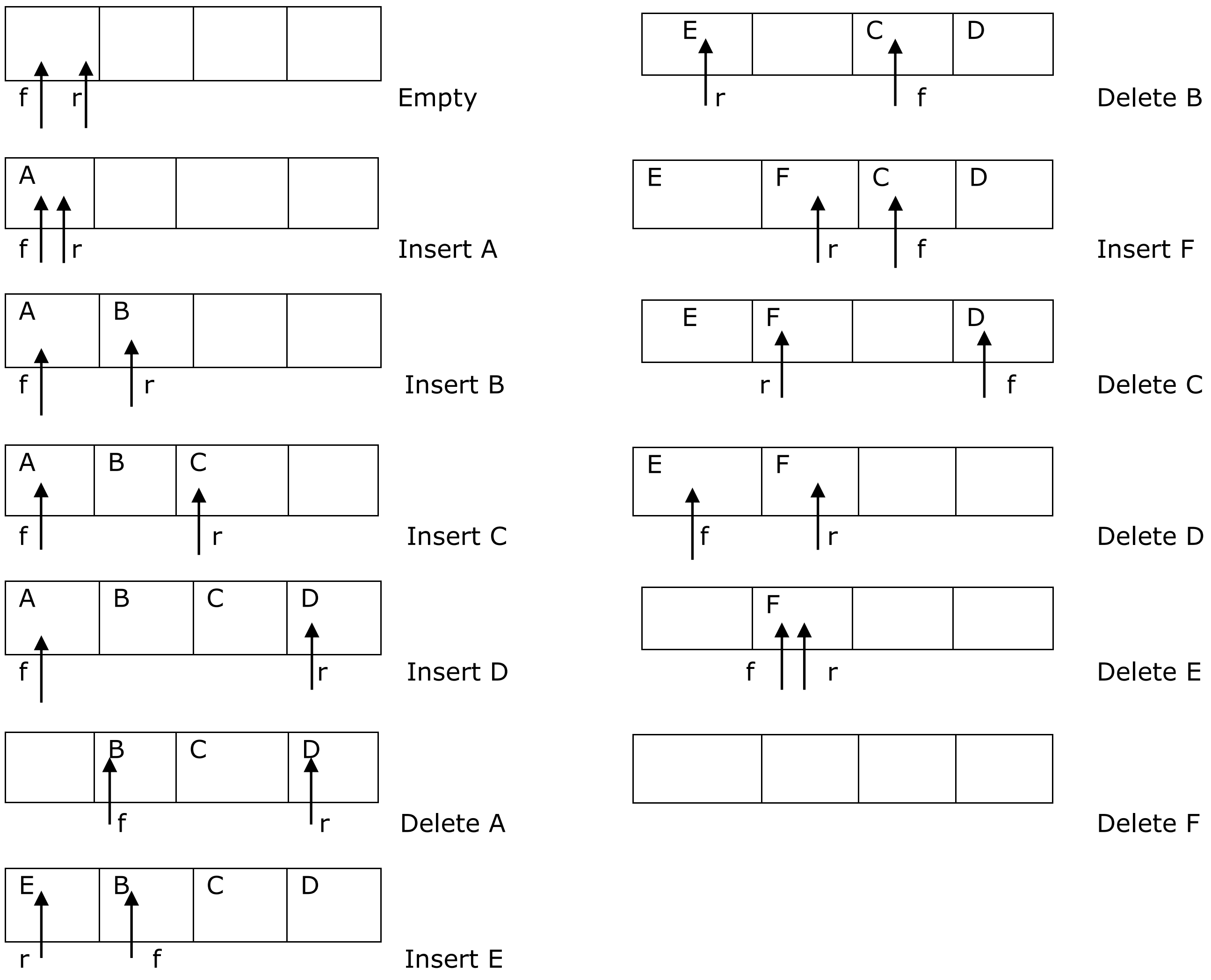
A deque (double – ended-queue) is a linear list In which insertion & deletion are made to or from either end to the structure.

Such a structure can be represented by below figure. It is clear that a deque is more general than a stack or queue . There are two variations of a deque viz.



A queue (fig.6)

(1) The input-restricted
(2) The output-restricted
The input-restricted deque allows insertion at only one end output restricted deque permits deletion at only one end



Trace of operation on a circular queue.

Application of Queue

Simulation - One of the classical areas to which queues can be applied is that of Simulation.
Simulation is the process of forming a subtract model from a real simulation in order to understand the impact of modification and the effect of introducing various strategies on the situation. The main objective of the simulation program is to aid the user, often an operations research specialist or system

analyst, in projecting what will happen to a given physical situation under certain simplifying assumption. It allows the user to experiment with real proposed situation.

Priority Queue A queue in which we are able to insert items or remove items from any position based on some priority is often referred as priority queue.

UNIT – III

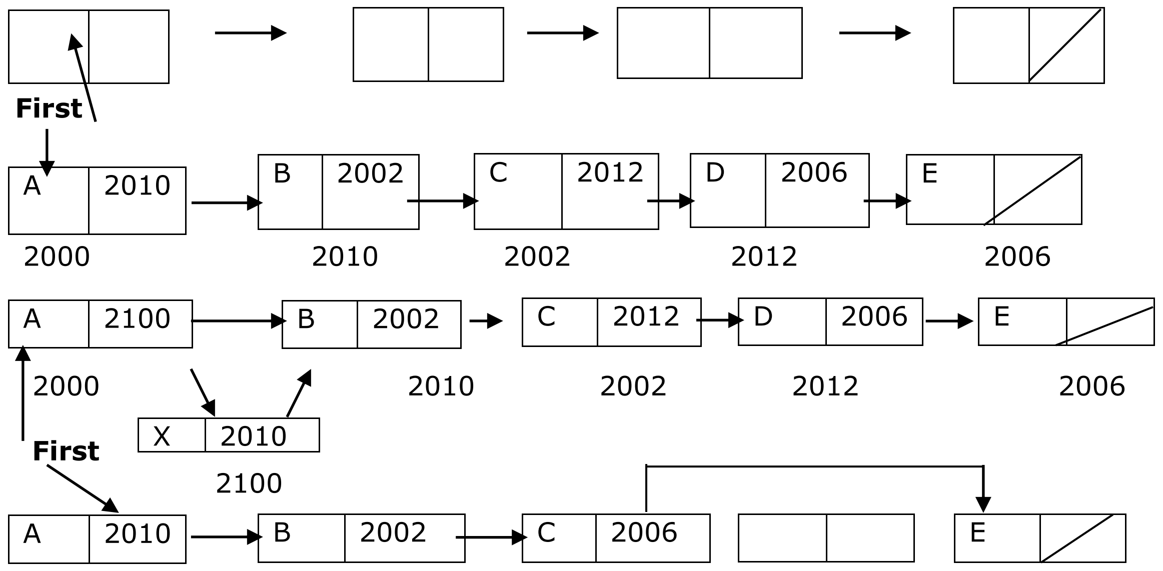
Consider a list consisting of element which vary individually in size the task of directly computing the address of a particular element becomes much more difficult , an obvious method of obtaining the address of a node (element) is to store this address in the computer memory .If the list in question has **n** nodes , we can store the address of each node in a vector consisting of **n** elements. The first element of the vector contains the address of the first node of the list , the second element contains the address of the second node & so on.

Pointers are always of the same length & this property enables the manipulation of pointers to be performed in a uniform manner using simple allocation technique , regardless of the configuration of the structures to which they may point.

Pointers are capable of representing a much more complex relationship between elements of a structure than linear order.

The use of pointers or links to refer to elements of a data structure implies that elements which are logically adjacent need not be physically adjacent in memory. This type of allocation is called linked allocation.

A list has been defined to consist of an ordered set of elements which may vary in number. A simple way to represent a linear list is to expand each node to contain a link or pointer to the next node . This representation is called a one-way chain or single linked linear list.



The variable first contains an address or pointer which gives the location of the first node of the list. Each node is divided in to two parts. The first part represents the information of the element Q the 2nd part contains the address of the next node . The last node of the list does not have a successor node & consequently , no actual address is stored in the pointer field . In such a case a NULL value is stored as the address.

Advantages over sequential allocation list (queue, stack):

1. Insertion & deletion can be done easily in link allocation.
2. To join or to split two linked list this can be done by changing pointers & does not requires movement of nodes.
3. The pointers or links consume additional memory , but if only a part of memory word is being used, then a pointer can be stored in the remaining part. It is possible to group nodes so as to require only one link per several nodes.
4. Pointers can be used to specify more complex relation between nodes such as that of a tree or a directed graph .

This is difficult & indeed for certain graphs, impossible to specify by using sequential allocation. In short , for certain operations linked allocated is more efficient than sequential operation and for other operation the opposite is true . In many applications both types of allocation are used.

Operation on linear lists using linked storage structure

We assume that a typical element or node consist of two fields , namely an information field called **INFO** and a pointer field denoted by **LINK**. The name of a typical element is denoted by NODE**.** The nodes structure is given as follows.

NODE

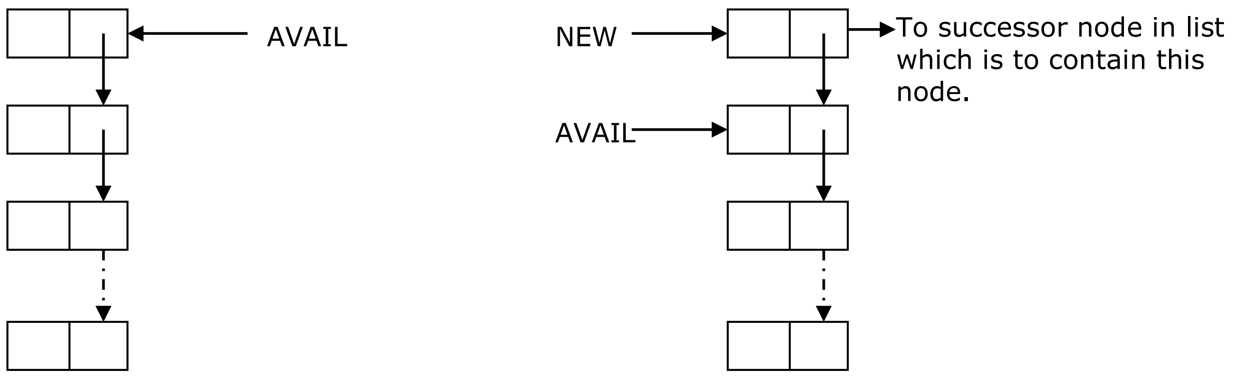
INFO	LINK

It is further assumed that an available area of storage for this node structure consist of a linked stack of available nodes , as shown in fig.1(a) where the pointer variable AVAIL contains the address of the top node in the stack.

A similar procedure can be formulated for the return of a discarded node to the availability stack. If the address of the discarded node is given by the variable FREE , then the linked field of this node is set to the value of FREE becomes the new of value of AVAIL. This process is shown in fig.

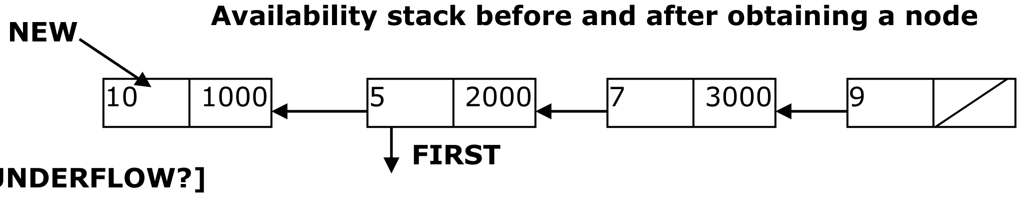
FUNCTION INSERTINFRONT(X , first)

- This function inserts X.
- Given **X** , a **NEW** element and **FIRST** a pointer to the element of a linked linear list whose typical node contains **INFO & LINK**.
- **AVAIL** is a pointer to the top element of the availability stack.
- **NEW** is a temporary pointer variable.
- It is required that X proceed the node whose address is given by FIRST.



(A) Before (B) After

Availability stack before and after obtaining a node



```
1. [UNDERFLOW?]
       IF AVAIL = NULL THEN
              Write ('AVAILABILITY STACK UNDERFLOW')
       Return (FIRST)
2. [OBTAIN ADDRESS OF NEXT FREE NODE]
       NEW ← AVAIL
3. [REMOVE FREE NODE FROM AVAILABILITY STACK]
       AVAIL ← LINK(AVAIL)
4. [INITIALIZE FIELDS OF NEW NODE AND ITS LINK TO THE LIST]
       INFO (NEW) ← X
       LINK (NEW) ← FIRST
5. [RETURN ADDRESS OF THE NEW NODE]
       Return (NEW)
```

Insert returns a pointer value to the variable FIRST.

FUNCTION INSERTATEND (X, FIRST)

- This function insert **X**.

- Given **X** , a **NEW** element and **FIRST** a pointer to the element of a linked linear list whose typical node contains **INFO & LINK**.
- **AVAIL** is a pointer to the top element of the availability stack.
- **NEW** and **SAVE** are temporary pointer variables.
- It is required that X be inserted at the end of the list.

```
1. [UNDERFLOW?]
        IF AVAIL = NULL THEN
            Write ('AVAILABILITY STACK UNDERFLOW')
        Return (FIRST)
2. [OBTAIN ADDRESS OF NEXT FREE NODE]
        NEW ← AVAIL
3. [REMOVE FREE NODE FROM AVAILABILITY STACK]
        AVAIL ← LINK(AVAIL)
4. [INITIALIZE FIELDS OF NEW NODE AND ITS LINK TO THE LIST]
        INFO (NEW) ← X
        LINK (NEW) ← NULL
5. [IS THE LIST EMPTY?]
        IF FIRST = NULL THEN
            Return (NEW)
6. [INITIALIZE SEARCH FOR THE LAST NODE]
        SAVE ← FIRST
7. [SEARCH FOR THE END OF THE LIST]
        REPEAT WHILE LINK (SAVE) ! = NULL
            SAVE ← LINK (SAVE)
8. [SET LINK FIELD OF LAST NODE OF NEW]
        LINK (SAVE) ← NEW
9. [RETURN FIRST NODE POINTER]
        Return (FIRST)
```

The following detailed algorithm performs on insertion in a list according to the ordering that all terms are kept in increasing order of their INFO field.

FUNCTION INSERTSORT(X, FIRST)

- This function insert **X**.
- Given **X** , a **NEW** element and **FIRST** a pointer to the element of a linked linear list whose typical node contains **INFO & LINK**.
- **AVAIL** is a pointer to the top element of the availability stack.
- **NEW** and **SAVE** are temporary pointer variables.
- It is required that X be inserted so that it preserves the ordering of the terms in increasing order of their INFO fields.

```
1. [UNDERFLOW?]
        IF AVAIL = NULL THEN
            Write ('AVAILABILITY STACK UNDERFLOW')
        Return (FIRST)
2. [OBTAIN ADDRESS OF NEXT FREE NODE]
        NEW ← AVAIL
3. [REMOVE FREE NODE FROM AVAILABILITY STACK]
        AVAIL ← LINK (AVAIL)
4. [COPY INFORMATION CONTENT TO THE NEW NODE INFO FIELD]
        INFO (NEW) ← X
5.  [IS THE LIST EMPTY?]
        IF FIRST = NULL THEN
        LINK (NEW) ← NULL
            Return (NEW)
6. [DOES THE NEW NODE PROCEDE ALL OTHERS IN THE LIST?]
        IF INFO (NEW) <= INFO (FIRST) THEN
            LINK (NEW) ← FIRST
            FIRST ← NEW
```

```
                Return (FIRST)
   7. [INITIALIZE TEMPORARY POINTER]
            SAVE ← FIRST
   8. [SEARCH FOR THE END OF THE LIST]
            REPEAT WHILE LINK (SAVE) ! = NULL AND INFO (LINK (SAVE)) <= INFO (NEW)
                SAVE ← LINK (SAVE)
   9. [SET LINK FIELD OF NEW NODE AND ITS PREDECESSOR]
            LINK (NEW) ← LINK (SAVE)
            LINK (SAVE) ← NEW
   10.[RETURN FIRST NODE POINTER]
            Return (FIRST)
```

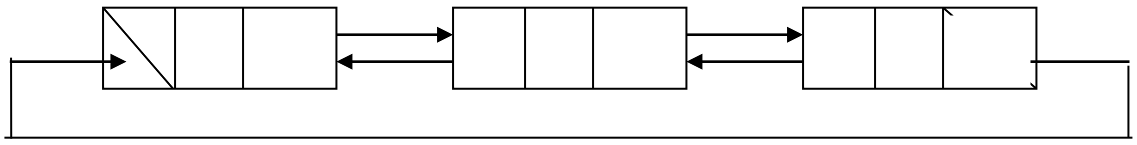
A general algorithm for deleting a node from a linked list is as follows:

FUNCTION DELETE (X, FIRST)
Given **Y** and **FIRST.**
This function deletes the node whose address is given by **X**.
TEMP is used to find the desired node.
PRED keeps track of the predecessor of **TEMP.**
FIRST is changed only when **X** is the **FIRST** element of the list.

```
   1. [EMPTY LIST?]
            IF FIRST = NULL THEN
                Write ('UNDERFLOW')
            Return
   2. [INITIALIZE SERACH FOR X]
            TEMP ← FIRST
   3. [FIND X]
            REPEAT THROUGH STEP 5 WHILE INFO (TEMP) != x AND LINK (TEMP) != NULL
   4. [UPDATE PREDECESSOR]
            PRED← TEMP
   5. [MOVE TO NEXT NODE]
            TEMP ← LINK (TEMP)
   6. [END OF THE LIST?]
            IF (TEMP) != X THEN
                Write('NODE NOT FOUND')
            Return
   7. [DELETE X]
            IF X = FIRST THEN
                FIRST ← LINK (FIRST)
            ELSE
                LINK (PRED) ← LINK (X)
   8. [RETURN NODE TO AVALIABLITY AREA]
            LINK (X) ← AVAIL
            AVAIL ← X
            Return
```

CIRCULAR LINKED LIST

If the Null pointer in the last node of a linked list is replace by the address of its first node then such a list is called circular linked list



Circular list have certain advantages over single link list.

Advantages:

The first advantage is concerned with the accessibility of a node. In the circular list every node is accessible from a given node.

A second advantage concerns the deletion operation. To delete an arbitrary node X, from a singly linked list the address of the first node of the list is necessary because to delete node X, the predecessor of this node has to be found. Where as in case of circular list such a requirement does not exist. Since the search for the predecessor of node X can be initialized from X itself.

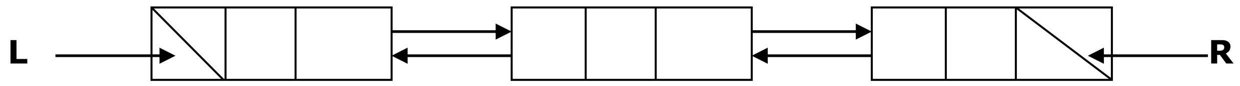
Disadvantage:

In using circular lists without some care in processing it is possible to get into an infinite loop.

Doubly Linked Linear List

In certain applications it is very desirable, sometimes indispensable that a list be traversed in either a forward or reversed manner. This property of a linked linear list implies that each node must contain two link fields instead of the usual one.

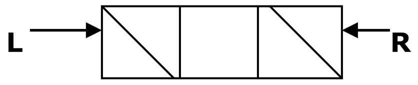
The links are used to denote the predecessor and successor of a node the link denoting the predecessor of a node is called the left link and that denoting successor is right link. A list containing this type of node is called a doubly linked linear list or a two-way chain. Such a linear list can be represented as show in below figure,



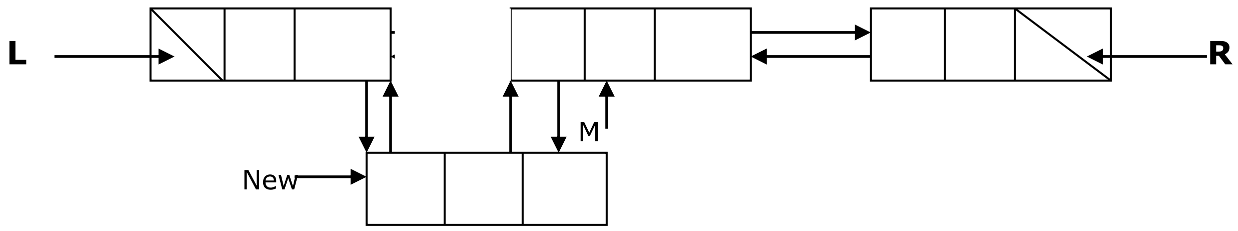
Where L and R are pointer variables denoting the left-most and right-most nodes in the list. The left link of the left-most node and right link of right-most node are set to NULL indicating the end of the list for each direction. The left and right links of a node are denoted by variables LPTR and RPTR respectively.

Consider the problem of inserting a node into a doubly linked linear list to the left of a specified node whose address is given by variable M.

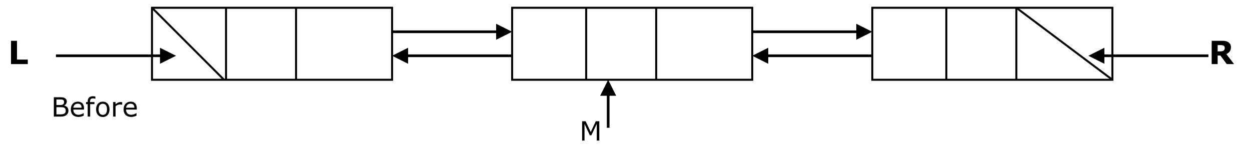
First of all the list would be empty. This is denoted by setting both L and R pointers to the address of the new node and by assigning a NULL value of the left and right links of the node being entered.



A second possibility is an insertion in the middle of the list. The list before and after insertion can be represent as shown above fig. where new is the address of the new node being inserted.



Finally the insertion can be made to the left of the left most node in the list. There by requiring the pointer L to be changed. This is shown below figure.



Function DOUBLYINSERT (L, R, M, X)

Given a doubly linked linear list whose left most and right most node address are given by the pointer variables **L** & **R**.

It is required to insert a node whose address is given by the pointer variable **NEW**.

The left and right links of a node are denoted by **LPTR** and **RPTR** respectively.

The information field is denoted by the variable **INFO.**

The name of an element of the list is **NODE**.

The insertion is to be preformed to the left of a specified node with its address given by the pointer variable **M.** The information to be entered in the node is contained in **X.**

```
1. [OBTAIN NEW NODE FROM AVAIBILITY STACK]
       NEW ← NODE
2. [COPY INFORMATION FIELD]
       INFO (NEW) ← X
3. [INSERTION INTO AN EMPTY LIST]
       IF R = NULL THEN
           LPTR (NEW) ← RPTR (NEW) ← NULL
           L ← R ← NEW
       Return
4. [LEFT MOST INSERTION]
       IF M = L THEN
           LPTR (NEW) ← NULL
           RPTR (NEW) ← M
           LPTR (M) ← NEW
           L ← NEW
       Return
5. [INSERT IN MIDDLE]
       LPTR (NEW) ← LPTR (M)
       RPTR (NEW) ← M
       LPTR (M) ← NEW
       RPTR (LPTR (M)) ← NEW
       Return
```

FUNCTION DOBLYDELETE (L, R, OLD)

Given a doubly linked list with the address of the left most and right most nodes given by the pointer variables **L** & **R**.

It is required to delete the node whose address is contained in the variables **OLD.**

Nodes contain left and right links with names **LPTR** & **RPTR.**

```
1. [UNDERFLOW?]
       IF R = NULL THEN
           Write ('UNDERFLOW')
       Return
2. [DELETE NODE]
       IF L = R THEN  (Single node in the list)
           L ← R ← NULL
       ELSE
           IF OLD = L THEN (Left most node being deleted)
               L ← RPTR (L)
               LPTR (L) ← NULL
           ELSE
               IF OLD = R THEN (Right most node being deleted)
                   R ← LPTR (R)
                   RPTR (R) ← NULL
               ELSE
                   RPTR (LPTR (OLD)) ← RPTR (OLD)
                   LPTR (RPTR (OLD)) ← LPTR (OLD)
3. [RETURN DELETED NODE]
       RESTORE (OLD)
       Return
```

Advantages of doubly link list:
- ➢ Access in both direction
- ➢ Binary trees, directed graph etc. are created using doubly link list.
- ➢ Searching is fast compared to singly link list.

UNIT - V

Sorting

What is sorting?

Sorting is a technique through which n elements or n records are arranged in either ascending or descending order which are stored in Array or file. This operation is most often performed in business data processing application as well as scientific applications. Actually, sorting is the operation of arranging the records or elements of a file or array into some sequential order according to a hierarchy.
There are two types of sorting technique

- External sorting
- Internal sorting

External sorting: The sorting in which data is sorted on secondary memory (i.e. floppy disk or magnetic tape) of the computer than the sorting is called external sorting.
Internal sorting: The sorting in which data is sorted in the main memory of the computer is called internal sorting.

Difference between Internal sort and External sort
Internal sort:

1. It is used for small file.
2. List of item being sorted was stored in high speed main memory; whole file is sorted internally at memory itself.
3. No extra processing except internal sorting is involved.
4. There are many popular techniques like simple sort, bubble sort, shuttle sort, etc.

External sort:

1. It is used for large file.
2. List of item being sorted may not fit into main memory.
3. There are generally few processed involved – RUN LIST are brought in Internal memory – Internal sort is performed on them – Sorted in a target file. – Later RUN LIST is retrieved from target file and merged together
4. There are three popular methods viz. Polyphase sort, Oscillating sort and Merge sort.

Q: How many sorting techniques are available?

Following are the sorting techniques which are internal sort:

1. Linear sort
2. Bubble sort
3. Selection sort
4. Insertion sort
5. Merge sort
6. Quick sort
7. Heap Sort
8. Radix sort
9. Shuttel sort
10. Shell sort

Linear sort (simple sort)

Suppose A(n) is the unsorted array consisting of n-elements or n-records & B(n) be the array which is to be output as a sorted array. Now the first step in this method is to find out the smallest element from a given array and store it in the output array B(n) as a first element. Now remove this element from the original array [by assuming dummy value]. Now again find out the smallest element from the array A(n) and place on store it in a array B(n) as a 2nd element and so on this process is repeated n times.

Hence, we have to find out smallest element and stored in another array n times.

Linear sort, is a simple one but it is inefficient because

1) If requires the twice the amount of storage.
2) If searches in the original table equal to the no. of elements in the table.

Bubble sort

In this method two records are interchanged immediately, when they are found out of order, when this approach is used, there are at most **(n-1) pass are required**. During the first pass R_1 and R_2

are compared and if they are out of order then the records R_1 and R_2 (are compared and if they are out of order then the records R_1 and R_2) are interchanged. This process is repeated for records R_2 and R_3 and after R_3 and R_4 and so on. This method will cause records with small value to move or "Bubble up" [After the first pass the record with the largest key will be in the last position. On each successive pass, the records with the next largest key will be placed in portion n-1, n-2. There by resulting in a sorted table.

After each pass a check can be made to determine whether any interchanges were done during that pass. If there is no interchange then the table is sorted and no further pass are required. In the best case one pass requires n-1 comparison.

The **worst case** performance of the bubble sort is **n(n-1)/2 comparisons and (n-1)/2 exchanges**. The **average case analysis is O(n^2)**.

FUNCTION BUBBLE-SORT (K, N)

Given a vector K of N elements, this procedure sorts the elements into ascending (increasing) order (using the method just described). The variables PASS and LAST denote the pass counter ad position of the last unsorted element, resp. The variable I is used to index the vector elements. The variable EXCHS is used to count the number of exchanges made on any pass. All variables are integer.

```
1. [INITIALIZE]
        LAST ← n (entire list assumed unsorted at this point)
2. [LOOP ON PASS INDEX]
        REPEAT THRU STEP 5 FOR PASS = 1, 2, 3, …….. , N – 1
3. [INITIALIZE EXCHANGE COUNTER FOR THIS PASS]
        EXCHS ← 0
4. [PERFORM PAIR WISE COMPARISIONS ON UNSORTED ELEMENTS]
        REPEAT FOR I = 1, 2, 3, …….. , LAST – 1
        IF K [I] > K [I+1]
        THEN  K [I] ←→ K [I+1]
              EXCHS ← EXCHS + 1
5. [WERE ANY EXCHANGES MADE ON THIS PASS]
        IF EXCHS = 0
        THEN  RETURN (Mission Accomplished, Return Early)
        ELSE LAST ← LAST – 1 (Reduce size of unsorted list)
6. [FINISHED]
        RETURN PASS (Maximum number of passes required)
```

<u>Selection sort</u>

Beginning with the first record in the table, a search is performed to locate the element which has smallest key.

Then it's is interchanged by the first record in the table. A search for the second smallest key is then carried out by examining the record from the second element onwards. The element with second smallest key is interchanged with the element located in the second portion of table.

In this way, the process of searching the record with next smallest key and placing it in its proper portion is carried out until all records have been sorted in ascending order.

The search for the records with the next smallest key is called a pass. There are (n-1) such pass required in order to place records in their proper location.

During the first pass (n-1) records are compared for the ith path n-i comparisons are required. The <u>total no of comparison</u> is therefore the sum

$$\sum_{i=1}^{(n-1)} (n-i) = \underline{\mathbf{\tfrac{1}{2}\, n\, (n-1)}}$$

The no. of comparison is proportional to n^2 i.e. **O(n^2)** the no. of records interchange depends upon how unsorted table is, <u>Since during each pass no more than one interchange is required</u>. <u>The more no. of interchange for sort is (n-1)</u>.

FUNCITON SELECTION_SORT (K, N);

Given a vector K of N elements, this procedure rearranges the vector I the order $K[1] \le K[2] \le \ldots.. \le K[N]$ (The sorting process is based on the techniques just described). The variable PASS denotes the pass index and the position of the element in the vector which is to be examined during a particular pass. The variable Min-index denotes the position of the smallest element encountered in a particular pass. The variable I is used to index elements K[pass] to K[n] in a given pass. All variables are of type integer.

```
1) [LOOP ON PASS INDEX]
       Repeat UPTO step 4 for pass=1, 2,…, N-1
2) [INITIALIZE MINIMUM INDEX]
       Min-index ← PASS
3) [MAKE A PASS AND OBTAIN ELEMENT WITH SMALLEST VALUE]
       Repeat for I=PASS+1, PASS+2,……,N
       IF K[I]<K[Min-index] then Min-index ← I
4) [EXCHANGE ELEMENTS]
       IF Min-index ≠ PASS then K[PASS] ‹--› K[Min-index]
5) [FINISHED]
       Return
```

Insertion Sort

Insertion sort is a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quick sort, heap sort, or merge sort. However, insertion sort provides several advantages:

- simple implementation
- efficient for (quite) small data sets
- adaptive, i.e. efficient for data sets that are already substantially sorted: the time complexity is O(n + d), where d is the number of inversions
- more efficient in practice than most other simple quadratic (i.e. O(n^2)) algorithms such as selection sort or bubble sort: the average running time is $n^2/4$, and the running time is linear in the best case.
- stable, i.e. does not change the relative order of elements with equal keys
- in-place, i.e. only requires a constant amount O(1) of additional memory space

Worst case performance	O(n^2)
Best case performance	O(n)
Average case performance	O(n^2)

Pseudo code

```
InsertionSort (A)
for j = 2 to length [A]
 do
  key = A [j]
 // Insert A [j] is a sorted in the sequence A [1 .. j - 1].
  i = j - 1
  while i> 0 and A [i]> key
   do A [i + 1] = A [i]
   i = i - 1
   A [i + 1] = key
```

Function in C

```
void insertionSort(int numbers[], int array_size)
{
  int i, j, value;
  for (i = 1; i < array_size; ++i)
  {
    value = numbers[i];
    for (j = i; j > 0 && numbers[j - 1] > value; --j)
    {
      numbers[j] = numbers[j - 1];
     }
      numbers[j] = value;
  }
}
```

Shell sort

Shell sort is a sorting algorithm that is a generalization of insertion sort, with two observations:

- Insertion sort is efficient if the input is "almost sorted".

- Insertion sort is typically inefficient because it moves values just one position at a time.

In shell sort entire data is divided into two parts. Suppose, the array has n elements and m=n. Now in the first step we have to find out the half of n means m=m/2 if m=0 than terminate the process. Else compare A(1) with A(m+1) shell of array and if necessary make exchange, than compare A(2) and A(m+2) similarly A(3) and A(m+3).

In general compare A(i) and A(m+i) until i+m becomes n. During the comparison if there are found out of order than interchange them and at that time i-m > 0 than perform shuttle sort otherwise compare next two elements.

If (i+m) becomes n than again find out m/2 and check if m=0 if it is, than terminate the process and sorted array is available.

Shell sort improves insertion sort by comparing elements separated by a gap of several positions. This lets an element take "bigger steps" toward its expected position. Multiple passes over the data are taken with smaller and smaller gap sizes. The last step of Shell sort is a plain insertion sort, but by then, the array of data is guaranteed to be almost sorted.

Gap can be decided by arranging list of numbers in a tabular format for e.g. with gap of 5 means 5 columns. Thus in shell sort entire data is divided into n1 columns. Suppose, the array of n elements and let as take m=n. find out the n1 of n means m=m/n1; if m=0 than terminate the process. Sort data in each column. Once all columns are sorted concatenate all columns. And repeat same process for a bigger gap (which is less than current value of n1 i.e. 3 or 2)

Worst case performance	depends on gap sequence. Best known: $O(n\log^2 n)$
Best case performance	$O(n)$
Average case performance	depends on gap sequence

Pseudo-code

```
input: an array a of length n with array elements numbered 0 to n − 1

inc ← round(n/2)
while inc > 0 do:
    for i = inc .. n − 1 do:
        temp ← a[i]
        j ← i
        while j ≥ inc and a[j − inc] > temp do:
            a[j] ← a[j − inc]
            j ← j − inc
        a[j] ← temp
  inc ← round(inc / 2.2)
```

<u>Quick Sort</u>

Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.
The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Worst case performance	$\Theta(n^2)$
Best case performance	$\Theta(n\log n)$
Average case performance	$\Theta(n\log n)$
Worst case space complexity	$\Theta(n)$

FUNCITON QUICK_SORT (K, LB, UB)

Given a table K of N records, this recursive procedure sorts the table in ascending order. A dummy record with key K [N+1] is assumed where K [I] <= K [N + 1] for all 1 <= I <= N. The integer parameters LB and UB denote the lower and upper bound of the current sub-table being processed. The indices I and J are used to select certain keys during the processing of each sub-table. KEY contains the key value which is being placed in its final position within the sorted sub-table. FLAG is a logical variable which indicates the end of the process that places a record in its final position. When flag becomes false, the input sub-table has been partitioned into logical disjointed parts.

```
1. [INITIALIZE]
         FLAG ← TRUE
2. [PERFORM SORT]
         IF LB < UB
         THEN  I ← LB
               J ← UB + 1
               KEY ← K[LB]
         REPEAT WHILE FLAG
               I ← I + 1
               REPEAT WHILE K [I] < KEY (Scan the key from left to right)
                     I ← I + 1
               J ← J - 1
               REPEAT WHILE K [J] > KEY (Scan the keys from right to left)
                     J ← J - 1
               IF I < J
               THEN  K [I] ←→ K [J] (Interchange records)
               ELSE FLAG ← FLASE
               K [LB] ←→ K [J] (Interchange records)
               CALL QUICK_SORT (K, LB, J-1) (Sort first sub-table)
               CALL QUICK_SORT (K, J + 1, UB) (Sort second sub-table)
3. [FINISHED]
         RETURN
```

Merge sort

It is an external sort. The operation of sorting is closely related to the process of merging. Merge sort is an *O*(*n* log *n*) comparison-based sorting algorithm. In most implementations it is stable, meaning that it preserves the input order of equal elements in the sorted output. It is an example of the divide and conquer algorithmic paradigm.

The process of merging of two ordered tables into single sorted table can be accomplished by successively selecting record with the smallest key. Occurring in the either of the table and placing this record in a new table, there by creating and order by list.

Worst case performance	Θ(*n*log*n*)
Best case performance	Θ(*n*log*n*) typical, Θ(*n*) natural variant
Average case performance	Θ(*n*log*n*)
Worst case space complexity	Θ(*n*) auxiliary

A	B	C
2	1	1
5	3	2
7	6	3
		5
		6
		7

FUNCITON MERGE_SORT (K, FIRST, SECOND, THIRD)

Given two ordered sub-tables stored in a vector K with FIRST, SECOND and THIRD, this procedure performs a simple merge. TEMP is the temporary vector used in the merging process. The variables I and J denote the index associated with the first and the second sub-tables respectively. L is an index variable associated with the vector TEMP.

```
1. [INITALIZE]
               I ← FIRST
               J ← SECOND
               L ← 0
2. [COMPARE CORRESPONDING ELEMENTS AND OUTPUT THE SMALLEST]
               REPEAT WHILE I < SECOND AND J < THIRD
```

```
                    IF K [I] <= K [J]
                    THEN L ← L + 1
                         TEMP [L] ← K [I]
                         I← I + 1
                    ELSE
                         L ← L + 1
                         TEMP [L] ← K [J]
                         J ← J + 1
3. [COPY THE REMAINING UNPORCESSED ELEMENTS IN OUTPUT AREA]
               IF I >= SECOND
               THEN REPEAT WHILE J <= THIRD
                    L ← L + 1
                    TEMP [L] ← K [J]
                    J ← J + 1
               ELSE
                    REPEAT WHILE I < SECOND
                    L ← L + 1
                    TEMP [L] ← K [I]
                    I ← I + 1
4. [COPY ELEMENTS INTO ORIGINAL AREA FORM TEMPORARY VECTOR]
          REPEAT FOR I = 1, 2, ...... , L
               K [FIRST – 1 + I] ← TEMP [I]
5. [FINISHED]
          RETURN
```

<u>Heap Sort</u>

Heap sort is a comparison-based sorting algorithm, and is part of the selection sort family. Although somewhat slower in practice on most machines than a good implementation of quick sort, it has the advantage of a worst-case $\Theta(n \log n)$ runtime. Heap sort is an in-place algorithm, but is not a stable sort.

The heap sort works as its name suggests. It begins by building a heap out of the data set, and then removing the largest item and placing it at the end of the sorted array. After removing the largest item, it reconstructs the heap, removes the largest remaining item, and places it in the next open position from the end of the sorted array. This is repeated until there are no items left in the heap and the sorted array is full. Elementary implementations require two arrays - one to hold the heap and the other to hold the sorted elements. Its worst case performance $\Theta(n\log n)$, best case performance $\Theta(n\log n)$, average case performance $\Theta(n\log n)$ and worst case space complexity $\Theta(n)$ total, $\Theta(1)$ auxiliary

Pseudocode

```
Heap-Sort (A)
   n <-- length[A]
   for j <-- n/2 downto 1 do
       Heapfy(A,j,n)
   for i <-- n downto 2 do
       t <-- A[i], A[i] <-- A[1], A[1] <-- t
       Heapfy(A,1,i-1)

Heapfy (A,j,n)
    k <-- j
    if 2j+1 <= n and A[2j+1] > A[k] then k <-- 2j+1
    if 2j <= n and A[2j] > A[k] then k <-- 2j
    if k <= j then
        t <-- A[j], A[j] <-- A[k], A[k] <-- t
        Heapfy(A,k,n)
```

<u>Comparison of Heap sort with other sorting technique</u>

Heap sort primarily competes with quick sort, another very efficient general purpose nearly-in-place comparison-based sort algorithm.

Quick sort is typically somewhat faster, due to better cache behavior and other factors, but the worst case running time for quick sort is O(n^2), which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk.
Thus, because of the O(n log n) upper bound on heap sort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heap sort.
Heap sort also competes with merge sort, which has the same time bounds, but requires $\Omega(n)$ auxiliary space, whereas heap sort requires only a constant amount. Heap sort also typically runs more quickly in practice on machines with small or slow data caches. On the other hand, merge sort has several advantages over heap sort:
Like quick sort, merge sort on arrays has considerably better data cache performance, often outperforming heap sort on a modern desktop PC, because it accesses the elements in order. Merge sort is a stable sort.
Merge sort parallelizes better; the most trivial way of parallelizing merge sort achieves close to linear speedup, while there is no obvious way to parallelize heap sort at all.
Merge sort can be easily adapted to operate on linked lists (with $\Theta(1)$ extra space[8]) and very large lists stored on slow-to-access media such as disk storage or network attached storage. Heap sort relies strongly on random access, and its poor locality of reference makes it very slow on media with long access times.
Intro sort is an interesting alternative to heap sort that combines quick sort and heap sort to retain advantages of both: worst case speed of heap sort and average speed of quick sort.

Desire Criteria of sorting Technique

It is difficult to assert that a particular sorting technique is always superior to other for every key set. Certain properties of a given key set play an important role in the determination of which sorting technique should be used.

Properties such as the no., size, distribution and orderness of keys often dictate as to which method should be used. Also, the amount of memory available in performing the sort may also be an important factor:

1) Selection sort or bubble sort can be used if the no. of records in the table is less.
2) If n is large and keys are short the radix sort can perform well.
3) With large n and long keys quick sort or merge sort or heap sort can be used.
4) If table is initially almost sorted the quick sort is avoided but bubble sort is used.

According to above four matters one can decide the particular sorting technique for best performance.

UNIT - VI

Searching

The technique through which a particular data element or a record can be find out from the n-element or n-records which may be sorted or not, in array or file is called searching.

There are two types of searching technique

1. Unsorted array linear search.
2. Sorted array search.

1. **Unsorted array linear search:** The searching in which data are already in unsorted form is known as unsorted array linear search. E.g. Linear search
2. **Sorted array search:** The searching in which data is already in sorted order is known as sorted array search. E.g. Binary search

Linear search: The simplest technique for searching on unordered table for a particular record is to scan each entry in the table in a sequential manner until the desired record is found.

This searching technique is the least efficient and easiest searching method. In this method the required element which has to be searched is compared with each and every element of an array. If the comparison is matched then the information about the key is output.

This method is also known as **sequential search**. An algorithm for such a search procedure is as follows:

FUNCTION LINEAR_SEARCH (K, N, X)

Given an unordered vector/array K consisting of N+1 (N >= 1) elements, this algorithm searches the vector for a particular element having value of X prior to the search. The function returns the index of the vector elements if the search is successful otherwise it returns 0 (zero).

```
1 [INITALISE SERACH]
      I ← 1
      K[N+1] ← X
2 [SERACH THE VECTOR]
      Repeat while k [I] != X
      I ← I + 1
3 [SUCCESSFUL SEARCH?]
      If I = N + 1
      Then write ('Unsuccessful search')
            Return (0)
      Else write ('Successful search')
            Return (I)
```

LINEAR SEARCH PROGRAM

```
# include<stdio.h>
int list[200];

void linear_search(int * , int , int );
void display(int *, int);

/* This function implements the linear search algorithm */

void  linear_search(int list[], int n, int key)
{
      int k;
      int flag = 0;
      for(k = 0; k< n; k++)
      {
            if(list[k] == key)
            {
                  printf("\n Search is successful \n");
                  printf("\n Element: %i Found at Location: %i", key,k+1);
                  flag = 1 ;
            }
```

```
	}
	if (flag==0)
		printf("\n Search is unsuccessful");
}

/* Output function display the data list */
void display(int list[], int n)
{
	int i;
	for(i = 0 ; i < n ; i++)
	{
		printf(" %d", list[i]);
	}
}

void main()
{
	int number, key, list[200];
	int i;
	printf("Input the number of elements in the list:");
	scanf("%d", &number);
	for(i = 0 ; i < number; i++)
	{
		list[i] = rand() %100; /* Generates data list */
	}
	printf("\n Input Element to be searched");
	scanf("%d", &key);

	printf("\n Entered list is as follows:\n");
	display(list, number);
	/* invoking function to search the key */
	linear_search(list, number, key);
	printf("\n In the following list\n");
	display(list, number);
}
```

Binary search

This searching method is also called sorted array searching. This method required the keys to be sorted in the array or list from which a key has to be searched.

Binary search can be handled only with sorted file. In this method the key which to be searched is compared with the entries at the extreme position of the array, in order to ensure that the key to be searched lies within the array. The array is divided into two parts and it is checked whether the key to be searched lies in which part. Then the part is again sub-divided into two parts. This process is to be continued until (a) the key match is obtained (b) the size of subpart is reached to element is other key is matched (c) search interval becomes empty.

In other word this is the simple method of accessing a table. The entries in the table are sorted in alphabetically or numerically increasing order. An appropriate method discussed in the previous section can be used to achieve this ordering. A search or a particular item with a certain key value resembles the search for a name in a telephone directory. The appropriate middle entry of the table is located, and its key value is examined. If its value is too high, then the key value of the middle entry of the first key of the table is examined and the procedure is repeated on the first half until the required item is found. If the value is too low, then the key of the middle entry of the second half of the table is tried and the procedure is repeated on the second half. This process continues until the desired key is found or the search interval becomes empty.

FUCNITON BINARY_SERACH (K, N, X)

Given a Vector/array K, consisting of N elements in ascending order, this algorithm searches the array for a given element whose value is given by X. The variables LOW, MIDDLE and HIGH denote the lower, middle and upper limits of the search intervals respectively. The function returns the index of the array element if the search is successful, otherwise returns 0 (zero).

```
	1 [INITALIZE SERACH]
```

```
            LOW ← 1
            HIGH ← N
      2 [PERFORM SERACH]
            Repeat THROUGH STEP 4 WHILE LOW <= HIGH
      3 [OBTAIN INDEX OF MID POINT OR INTERVAL]
            MIDDLE ← [ (LOW + HIGH) / 2]
      4 [COMPARE]
            If X < K [ MIDDLE ]
            Then HIGH ← MIDDLE – 1
            Else If X > K [ MIDDLE ]
                   Then LOW ← MIDDLE + 1
                   Else Wirte (‘Successful Search’)
                    Return (MIDDLE)
      5 [UNSUCCESSFUL SERACH]
            Wirte (‘Unsuccessful Search’)
            Return (0)
```

A binary search can also be performed recursively. Such an algorithm can be analyzed with respect to time and space. A recursive algorithm to perform this task is as follows.

FUCNTION BINARY_SEARCH (P, Q, K, X)

Given the bound of a array P, Q and a vector/array K, this algorithm recursively searches for the given elements whose value is equal to X. MIDDLE and LOC are integer variables.

```
1 [SEARCH VECTOR K BETWEEN P AND Q FOR VALUE X]
      If P > Q
      Then LOC ← 0
      Else MIDDLE ← [ (P+Q) / 2 ]
            If X < K [ MIDDLE ]
            Then LOC ← BINARY_SEARCH (P, MIDDLE -1 , K, X)
            Else  If X > K [ MIDDLE ]
                  THEN LOC ← BINARY_SEARCH (MIDDLE +1, Q, K, X)
                  Else LOC ← MIDDLE
2 [FINISHED]
      Return (LOC)
```

<u>BINARY SEARCH PROGRAM</u>

```
#include <stdio.h>
/* thisfunction implements the binary search algorithm */
int binary_search(int list[], int key, int n)
{
      int flag = 0;
      int high = n-1, low = 0, mid;

      mid = (high + low) / 2;

      printf("\n\n Looking for %d \n", key);

      while ((! flag) && (high >= low))
      {
            printf(" Low %d Mid %d High %d \n", low, mid, high);

            if (key == list[mid])
                  flag = 1;
            else if (key < list[mid])
                  high = mid - 1;
            else
                  low = mid + 1;

            mid = (high + low) / 2;
      }
      return((flag) ? mid: -1);
}
```

```
void main(void)
{
      int list[100], i;

      for (i = 0; i < 100; i++)
            list[i] = i+20;

      printf("Result of search %d\n", binary_search(list, 33, 100));
      printf("Result of search %d\n", binary_search(list, 75, 100));
      printf("Result of search %d\n", binary_search(list, 1001, 100));
}
-------------------------------------------------------------------------------
#include <stdio.h>
#define MAX_LEN 10
/* Non-Recursive function*/
void b_search_nonrecursive(int l[],int num,int ele)
{
      int l1,i,j, flag = 0;
      l1 = 0;
      i = num-1;
     while(l1 <= i)
    {
      j = (l1+i)/2;
      if( l[j] == ele)
      {
            printf("\nThe element %d is present at position %d in list\n",ele,j);
            flag =1;
            break;
      }
      else
      if(l[j] < ele)
            l1 = j+1;
      else
            i = j-1;
    }
   if( flag == 0)
      printf("\nThe element %d is not present in the list\n",ele);
}

/* Recursive function*/
int b_search_recursive(int l[],int arrayStart,int arrayEnd,int a)
{
      int m,pos;
      if (arrayStart<=arrayEnd)
      {
            m=(arrayStart+arrayEnd)/2;
            if (l[m]==a)
                  return m;
            else if (a<l[m])
                  return b_search_recursive(l,arrayStart,m-1,a);
                   else
                  return b_search_recursive(l,m+1,arrayEnd,a);
      }
   return -1;
}

void read_list(int l[],int n)
{
      int i;
      printf("\nEnter the elements:\n");
      for(i=0;i<n;i++)
            scanf("%d",&l[i]);
}

void print_list(int l[],int n)
{
      int i;
      for(i=0;i<n;i++)
            printf("%d\t",l[i]);
}
/*main function*/
void main()
{
      int l[MAX_LEN], num, ele,f,l1,a;
      int ch,pos;

      clrscr();

      printf("=======================================================");
      printf("\n\t\t\tMENU");
      printf("\n======================================================");
      printf("\n[1] Binary Search using Recursion method");
      printf("\n[2] Binary Search using Non-Recursion method");
      printf("\n\nEnter your Choice:");
      scanf("%d",&ch);

      if(ch<=2 & ch>0)
      {
            printf("\nEnter the number of elements : ");
            scanf("%d",&num);
            read_list(l,num);
            printf("\nElements present in the list are:\n\n");
            print_list(l,num);
            printf("\n\nEnter the element you want to search:\n\n");
            scanf("%d",&ele);

            switch(ch)
            {
                  case 1:printf("\nRecursive method:\n");
                        pos= b_search_recursive(l,0,num,ele);
                        if(pos==-1)
                        {
                              printf("Element is not found");
                        }
                        else
                        {
                              printf("Element is found at %d position",pos);
                        }
                        getch();
                        break;

                  case 2:printf("\nNon-Recursive method:\n");
                        b_search_nonrecursive(l,num,ele);
                        getch();
                        break;
            }
      }
   getch();
}
```

HASHING TECHNIQUE

The **linear search (sometimes called the *sequential search*) *is the simplest algorithm to*** search for a specific target key in a data collection, for example to search for a specific integer value in an array of integers. It is also the least efficient. It simply examines each element in turn, starting with the first element, until it finds the target element or it reaches the end of the array. A linear search does not require the array to be sorted.

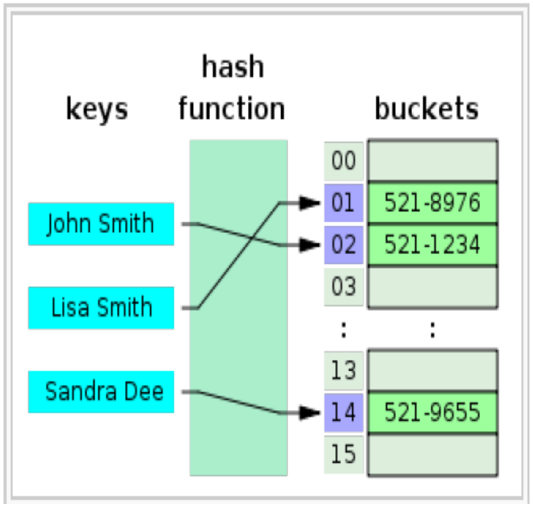
The **binary search is the standard method for searching through a sorted array. It is** much more efficient than the linear search, but it does require that the elements be in order.

The time taken for a search using each of these methods **depends on the size of the collection. The time for a linear search is proportional to the size of the collection –** it takes 10 times as long on average to find an element in an array of 100 elements as it does in an array of 10 elements. The binary search time depends on the logarithm of the collection size – it takes twice as long on average to find an element in an array of 100 elements as it does in an array of 10 elements.

A further search method is **hashing. Hash data structures allow the storage and retrieval** of data in an average time which **does not depend at all on the collection size.**

The simplest data structure is the **hash table. The base is of a hash table is a hash function. A hash function is a function that takes an element of whatever data type you** are storing (integer, string, etc) and outputs an integer in a certain range. This integer is used to determine the location in the table for that element. The hash table itself consists of an array whose indices are the range of the hash function.

Hashing provides a means for accessing data without the use of an index structure. Data is addressed on disk by computing a function on a search key instead.
For example, a hash table for a car database might make use of the registration plate as a key. Let's assume the table has a size of 20. A possible hash function would be **Value Mod 20,**
or example *(sum of numeric values of the characters) % 20*
where % is the modulus operator, giving the remainder after dividing.
This would give translations like this:
SD52DFG 28+13+5+2+13+15+16 = 92: ***Hashcode = 12***
FG02GTH 15+16+0+2+16+29+17 = 95: ***Hashcode = 15***
DR52GHY 13+27+5+2+16+17+34 = 114: ***Hashcode = 14***
FT52RET 15+29+5+2+27+14+29 = 121: ***Hashcode = 1***
NN52FRT 23+23+5+2+15+27+29 = 124: ***Hashcode = 4***
A hash table with size 20 which has records with these keys added could look like this:
Hash table contents
0: empty
1: occupied Make:Honda Reg:FT52RET
2: empty
3: empty
4: occupied Make:Jaguar Reg:NN52FRT
5: empty
6: empty
7: empty
8: empty
9: empty
10: empty
11: empty
12: occupied Make:Mazda Reg:SD52DFG
13: empty
14: occupied Make:Toyota Reg:DR52GHY
15: occupied Make:Rover Reg:FG02GTH
16: empty
17: empty
18: empty
19: empty



A good hash function is one that gets a fairly even distribution of the numbers in the output range, even if the input values are very poorly distributed (for e.g., English words are poorly distributed since they only use 26 different letters and some letters and some combinations of letter occur far more frequently than others.

Different Kind of hashing techniques are

- ***Static Hashing***
- **Dynamic Hashing**

Closed hashing

Linear probing

Linear probing is one method for dealing with collisions. If a data element hashes to a location in the table that is already occupied, the table is searched consecutively from that location until an open location is found. The new key would then be stored in **location 16, as location 15 is also already occupied.**

Rehashing

In rehashing, a second function is used to generate an alternative address. The function should hash to a location far from the original one.

Open hashing

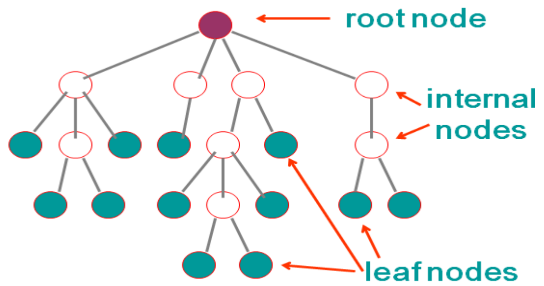
In open hashing further additions which have hashed to an occupied location are stored out with the main table. These locations may be stored as part of an overflow table or they may be allocated dynamically.

UNIT V

Trees

- A *tree* is a non-linear data structure that consists of a *root node* and potentially many levels of additional nodes that form a hierarchy
- Nodes that have no children are called *leaf nodes*
- Non-root and non-leaf nodes are called *internal nodes*

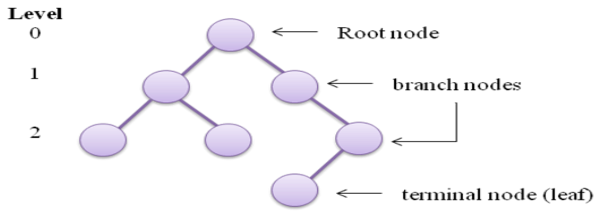
imagine an upside down tree



A tree data structure

Tree can represent inheritance relationship between classes.

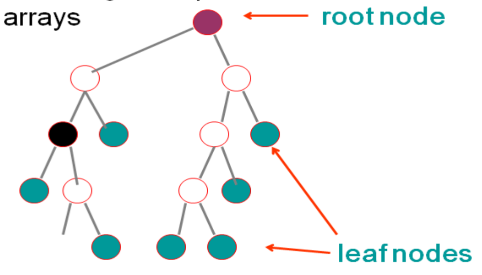
- A **directed tree** is an acyclic diagraph which has only one node called its **root**, with indegree 0, while all other nodes have indegree 1.
- Any node having the outdegree 0 is called **terminal node** or **leaf**.
- All other nodes are called **branch nodes**.
- The **level** of the root of a directed tree is 0, while the level of any node is equal to its distance from the root.



- The number of the sub-trees of a node is called the **degree of the node**.
- The set of disjoint trees is a **forest.**
- Tree:
  - A tree contains one or more nodes such that one of the nodes is called the root while all other nodes are partitioned into a finite number of trees called **subtrees**.
  - If in a directed tree, the outdegree of every node is less or equal to m, then the tree is called an **m-ary** tree.
- For m=2, the trees are called **binary** and **complete binary trees**.

Binary Trees

- A *binary tree* is defined recursively. Either it is empty (the base case) or it consists of a *root* and two *subtrees*, each of which is a binary tree
- Binary trees and trees typically are represented using references as dynamic links, though it is possible to use fixed representations like arrays

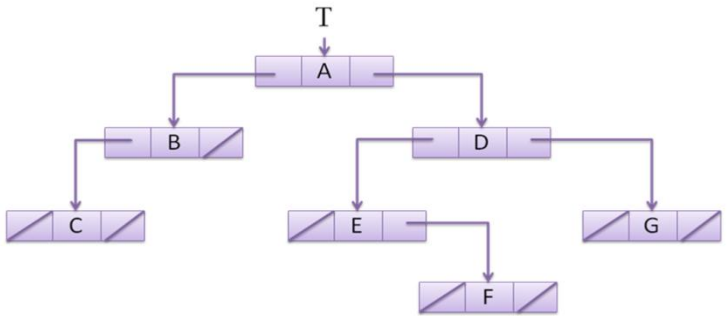


Linked representation of binary tree

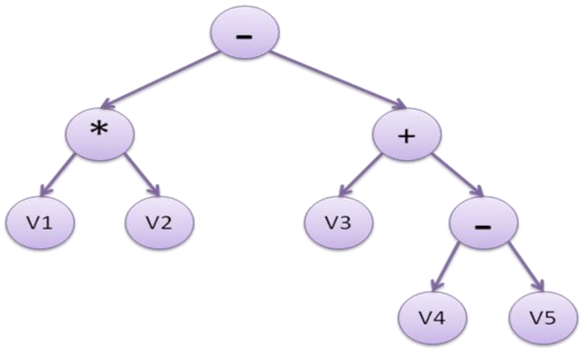
- Each node of the tree has the following representation:

LPTR	DATA	RPTR

- The following is the linked representation for the sample tree.



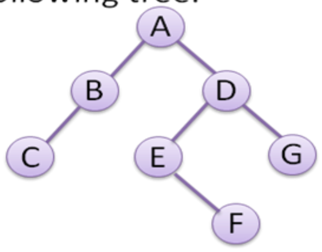
- The following is the tree representation for the expression
  v1 * v2 – (v3 + v4 - v5)

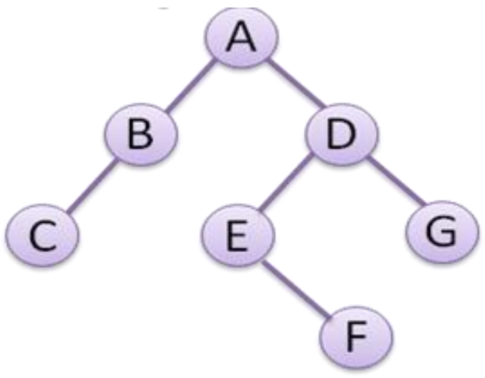


Traversal of binary tree

- Traversal is a procedure by which each node in the tree is processed exactly once in a systematic manner.
- There are 3 types of traversal:
  1. Preorder
  2. Inorder
  3. Postorder

- E.g Consider the following tree.

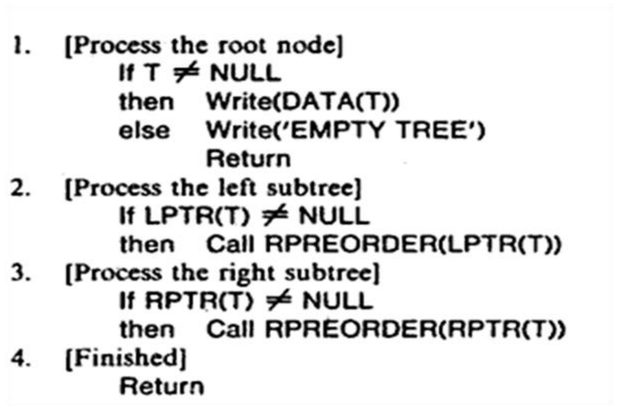




- **Preorder traversal**:
    - Process the root node
    - Traverse the left subtree in preorder
    - Traverse the right subtree in preorder
  - Processing order: A B C D E F G
- **Inorder traversal**:
    - Traverse the left subtree in inorder
    - Process the root node
    - Traverse the right subtree in inorder
  - Processing order: C B A E F D G
- **Postorder traversal**:
    - Traverse the left subtree in postorder
    - Traverse the right subtree in postorder
    - Process the root node
  - Processing order: C B F E G D A

Algorithm for preorder traversal

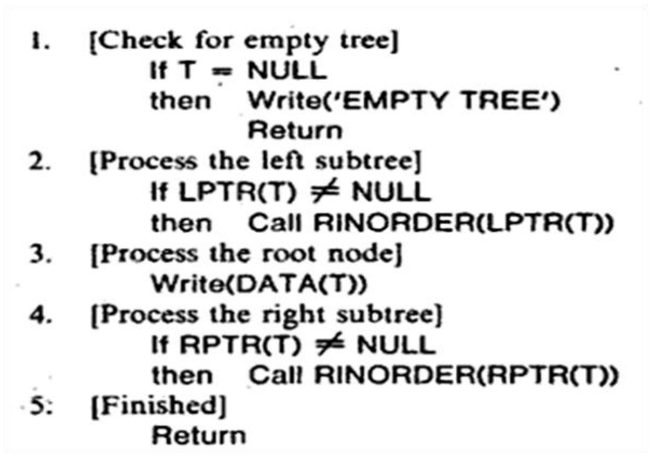
- RPREORDER(T).



```
1.  [Process the root node]
        If T ≠ NULL
        then   Write(DATA(T))
        else   Write('EMPTY TREE')
               Return
2.  [Process the left subtree]
        If LPTR(T) ≠ NULL
        then   Call RPREORDER(LPTR(T))
3.  [Process the right subtree]
        If RPTR(T) ≠ NULL
        then   Call RPREORDER(RPTR(T))
4.  [Finished]
        Return
```

Algorithm for inorder traversal

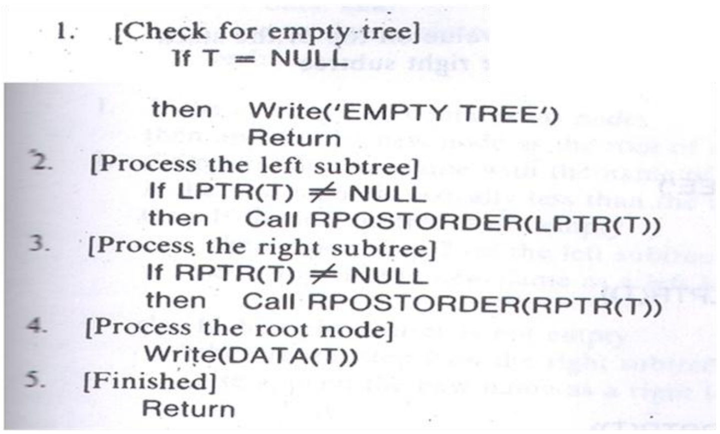
- RINORDER(T)



```
1.  [Check for empty tree]
        If T = NULL
        then   Write('EMPTY TREE')
               Return
2.  [Process the left subtree]
        If LPTR(T) ≠ NULL
        then   Call RINORDER(LPTR(T))
3.  [Process the root node]
        Write(DATA(T))
4.  [Process the right subtree]
        If RPTR(T) ≠ NULL
        then   Call RINORDER(RPTR(T))
5:  [Finished]
        Return
```

Algorithm for postorder traversal

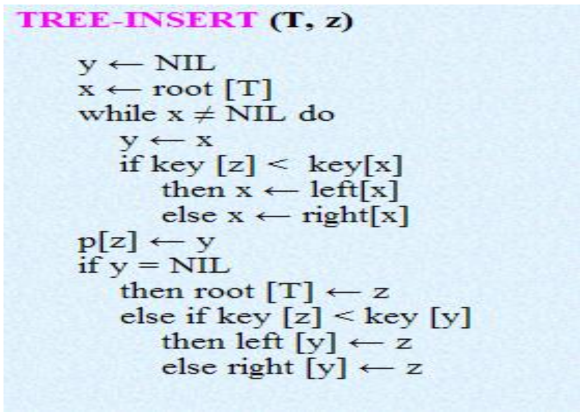
- RPOSTORDER(T)



```
1. [Check for empty tree]
       If T = NULL
       then   Write('EMPTY TREE')
              Return
2. [Process the left subtree]
       If LPTR(T) ≠ NULL
       then   Call RPOSTORDER(LPTR(T))
3. [Process the right subtree]
       If RPTR(T) ≠ NULL
       then   Call RPOSTORDER(RPTR(T))
4. [Process the root node]
       Write(DATA(T))
5. [Finished]
       Return
```

Binary Search Tree

- A Binary tree in which all the elements in the left sub-tree of a node n are less than the contents of n, and all the elements in the right sub-tree of n are greater than or equal to the contents of n is called **Binary Search Tree (BST).**
  - **Insertion of a node in a BST** : If value of new node is found to be greater than or equal to the data of the root node than the new node is inserted in the right sub-tree of the root node, otherwise, the new node is inserted in the left sub-tree of the root node.



```
TREE-INSERT (T, z)

    y ← NIL
    x ← root [T]
    while x ≠ NIL do
        y ← x
        if key [z] < key[x]
            then x ← left[x]
            else x ← right[x]
    p[z] ← y
    if y = NIL
        then root [T] ← z
        else if key [z] < key [y]
            then left [y] ← z
            else right [y] ← z
```

Like other primitive operations on search trees, this algorithm begins at the root of the tree and traces a path downward. Clearly, it runs in O(*h*) time on a tree of height *h*.

- **Deletion in BST :** There are four possible cases
  - No node in the tree contains the specified data.
  - The node containing the data has no children.
  - The node containing the data has exactly one child.
  - The node containing the data has two children.

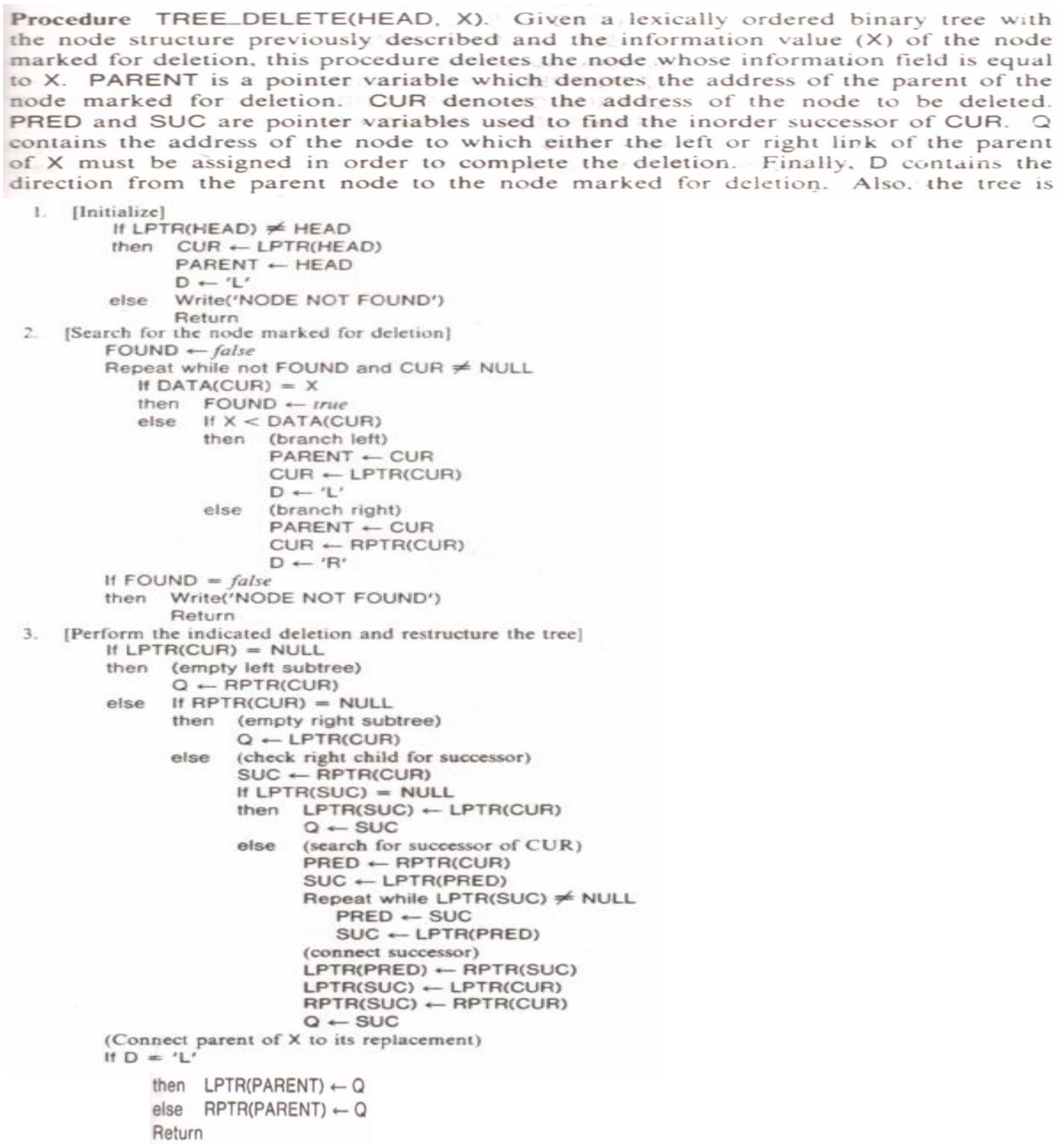
FUNCTION TREE-DELETE (T, z)

```
Removing a node from a BST is a bit more complex, since we do not want to
create any "holes" in the tree. If the node has one child then the child
is spliced to the parent of the node. If the node has two children then
its successor has no left child; copy the successor into the node and
delete the successor instead TREE-DELETE (T, z) removes the node pointed
to by z from the tree T. It returns a pointer to the node removed so that
the node can be put on a free-node list, etc.
```

```
1. if left [z] = NIL    .OR.     right[z] = NIL
2.     then y ← z
```

```
3.     else y ← TREE-SUCCESSOR (z)
4. if left [y] ≠ NIL
5.     then x ← left[y]
6.     else x ← right [y]
7. if x ≠ NIL
8.     then p[x] ← p[y]
9. if p[y] = NIL
10.          then root [T] ← x
11.          else if y = left [p[y]]
12.              then left [p[y]] ← x
13.              else right [p[y]] ← x
14.      if y ≠ z
15.          then key [z] ← key [y]
16.              if y has other field, copy them, too
17.      return y
```

The procedure runs in O(*h*) time on a tree of height *h*



Procedure TREE_DELETE(HEAD, X). Given a lexically ordered binary tree with the node structure previously described and the information value (X) of the node marked for deletion, this procedure deletes the node whose information field is equal to X. PARENT is a pointer variable which denotes the address of the parent of the node marked for deletion. CUR denotes the address of the node to be deleted. PRED and SUC are pointer variables used to find the inorder successor of CUR. Q contains the address of the node to which either the left or right link of the parent of X must be assigned in order to complete the deletion. Finally, D contains the direction from the parent node to the node marked for deletion. Also, the tree is

```
1.  [Initialize]
        If LPTR(HEAD) ≠ HEAD
        then   CUR ← LPTR(HEAD)
               PARENT ← HEAD
               D ← 'L'
        else   Write('NODE NOT FOUND')
               Return
2.  [Search for the node marked for deletion]
        FOUND ← false
        Repeat while not FOUND and CUR ≠ NULL
            If DATA(CUR) = X
            then   FOUND ← true
            else   If X < DATA(CUR)
                   then   (branch left)
                          PARENT ← CUR
                          CUR ← LPTR(CUR)
                          D ← 'L'
                   else   (branch right)
                          PARENT ← CUR
                          CUR ← RPTR(CUR)
                          D ← 'R'
        If FOUND = false
        then   Write('NODE NOT FOUND')
               Return
3.  [Perform the indicated deletion and restructure the tree]
        If LPTR(CUR) = NULL
        then   (empty left subtree)
               Q ← RPTR(CUR)
        else   If RPTR(CUR) = NULL
               then   (empty right subtree)
                      Q ← LPTR(CUR)
               else   (check right child for successor)
                      SUC ← RPTR(CUR)
                      If LPTR(SUC) = NULL
                      then   LPTR(SUC) ← LPTR(CUR)
                             Q ← SUC
                      else   (search for successor of CUR)
                             PRED ← RPTR(CUR)
                             SUC ← LPTR(PRED)
                             Repeat while LPTR(SUC) ≠ NULL
                                 PRED ← SUC
                                 SUC ← LPTR(PRED)
                             (connect successor)
                             LPTR(PRED) ← RPTR(SUC)
                             LPTR(SUC) ← LPTR(CUR)
                             RPTR(SUC) ← RPTR(CUR)
                             Q ← SUC
        (Connect parent of X to its replacement)
        If D = 'L'
            then   LPTR(PARENT) ← Q
            else   RPTR(PARENT) ← Q
            Return
```

- **Extended Binary Tree:** A binary tree can be converted to an extended binary tree by adding new nodes to its leaf nodes and to the nodes that have only one child. These nodes are added in such a way that all the nodes in the resultant tree have either zero or two children. The extended tree is also known as **2-tree.** The nodes of the original tree are called internal nodes and the new nodes that are added to binary tree, to make it an extended binary tree are called external nodes

Balanced Tree or AVL Tree

Searching in a binary tree is efficient if the heights of both left and right sub-trees of any node are equal. However frequent insertions and deletions in a BST is likely to make it unbalanced. The efficiency of searching is ideal if the difference between the heights of left and right sub-trees of all the nodes in a binary search tree is at the most one. Such a binary search tree is called as a **BALANCED BINARY TREE** it is also called AVL tree.

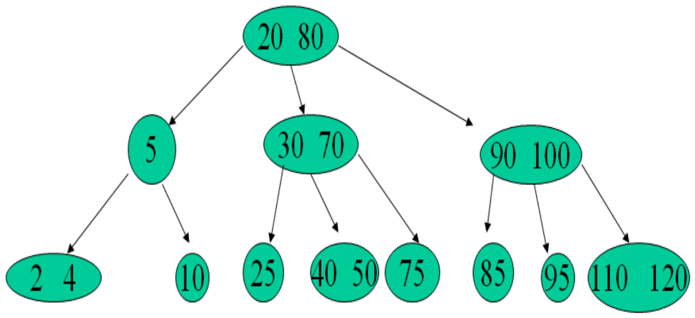
Balanced factor

- To balance a tree we need to find and set balance factor for each node of the tree
- How to find balance factor for each node?
  - The value of balance factor of any node is -1, 0 , 1. If it is other than these three values then the tree is not balanced or it is not an AVL tree.
  - If value is -1, then the height of the right sub-tree of that node is one more than the height of its left sub-tree.
  - If value is 0, then the height of its left and right sub-tree is exactly same.
  - If value is 1, then the height of the left sub-tree of that node is one more than the height of its right sub-tree.
- Rotation operation is performed to adjust balance factor of a node.

Multi-Way Search Trees

- To reduce the efficiency of operation performed on a tree we need to reduce the height of the tree.
- The time required to access the data from a secondary storage medium is very high.

Suggestion: If a node contains more number of values then at a time more values can be accessed from the secondary medium.



B-Trees

B-Tree is a multi-way search tree of order n that satisfies the following conditions:

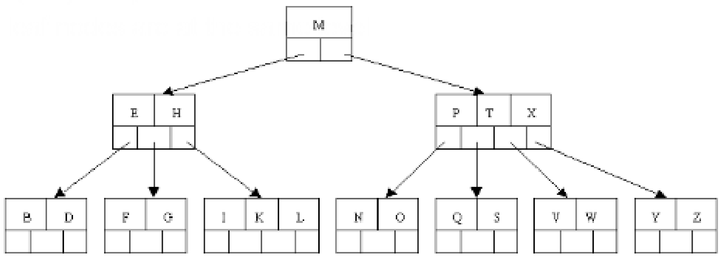
- All the non-leaf nodes have at least n/2 children and at the most n children.
- The non-leaf root node may have at the most n non-empty child and at least two child nodes.
- A B-tree can exists with only one node i.e. the root node containing no child.
- If a node has n children then it must have n – 1 values. All the values of a particular node are in increasing order.
- All the values that appear on the left most child of a node are smaller than then first value of that node. All the values that appear on the right most child of a node are greater than the last value of that value.
- If x and y are any two i^{th} and $(i+1)^{th}$ values of a node, where $x < y$, then all values appearing on the $(i+1)^{th}$ sub-tree of that node are greater than x and less than y.
- All the leaf nodes should appear on the same level.

A B-Tree is a height balanced search tree. A B-Tree of order m satisfies the following properties:

- The root node has at least 2 children (if it is not empty or is not a leaf

  node)
- All nodes other than root have at least ceil(m/2) children (i.e. links) and

  c
- A



A B-Tree of order 4 (3-4 search tree)

The following is an example of a B-tree of order 5 (A 4-5 B-Tree).

A 4-5 B-Tree means 4 key values and 5 links/children. To be precise,

- Other that the root node, all internal nodes have at least, ceil(5 / 2) = ceil(2.5) = 3 children. Maximum is 5.
- At least ceil(5/2)-1 = ceil(2.5)-1 = 3 -1 = 2 keys. Maximum is 4.
- In practice B-trees usually have orders a lot bigger than 5.

Insertion into a B-Tree (insert X):

1. Use search procedure to find leaf node where X should be added.
2. Add X to this node at the appropriate place among the values.
3. If there are <= M-1 values, we are done! Otherwise, the node has *overflowed*. To repair, split the node into 3 parts:

   Left: first (M-1)/2 values

   Middle: value at position 1+ (M-1)/2

   Right: last (M-1)/2 values

   Left and Right have just enough values: make them into nodes. They become the left and right children of Middle, which we add in the appropriate place in this node's parent.

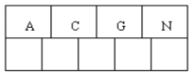
   If the parent overflows, we repeat the procedure.

   If the root overflows: we create a new root with Middle as its only value and Left and Right as its children.

B-TREE

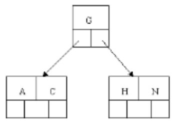
B-tree of order 5: (Creation & Addition)

C N G A **H E K Q M F W L T Z D P R X Y S**

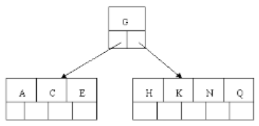


B-TREE

C N G A *H* E K Q M F W L T Z D P R X Y S



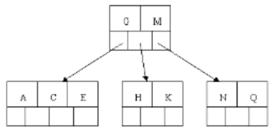
C N G A H *E K Q* M F W L T Z D P R X Y S



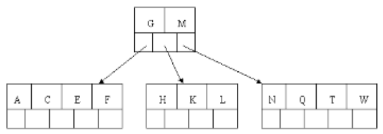
4

B-TREE

C N G A H E K Q *M* F W L T Z D P R X Y S



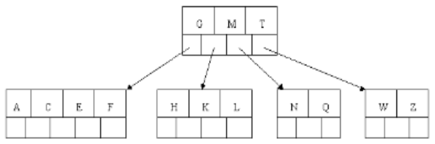
C N G A H E K Q M *F W L T* Z D P R X Y S



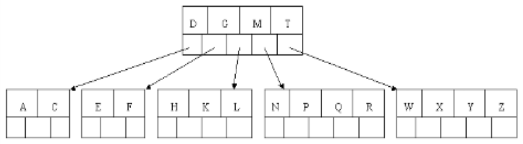
5

B-TREE

C N G A H E K Q M F W L T *Z* D P R X Y S

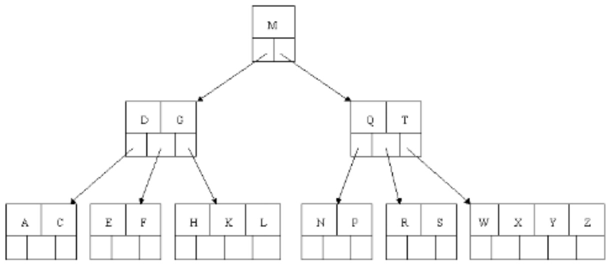


C N G A H E K Q M F W L T Z *D P R X Y* S



B-TREE

C N G A H E K Q M F W L T Z D P R X Y *S*

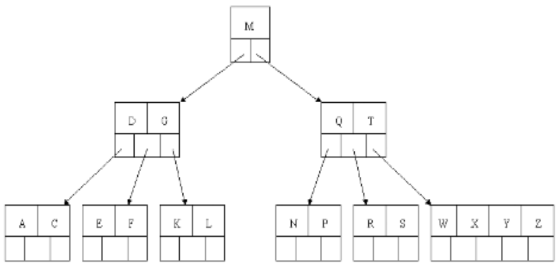


Deletion value from B-Tree

B-TREE

Deletion:

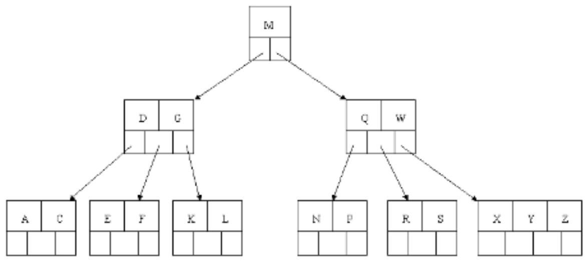
Delete H: Since H is in a leaf and the leaf has more than the minimum number of keys, this is easy. We move the K over where the H had been and the L over where the K had been. (Next T)



8

B-TREE

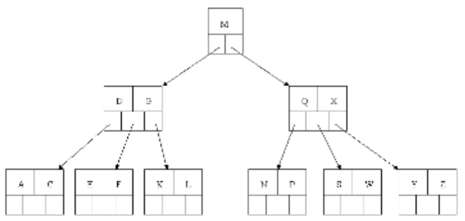
Delete T : Since T is not in a leaf, we find its successor (the next item in ascending order), i.e. W, and move W up to replace the T. That way, what we really have to do is to delete W from the leaf, which we already know how to do, since this leaf has extra keys. In ALL cases we reduce deletion to a deletion in a leaf, by using this method. (Next R)



9

B-TREE

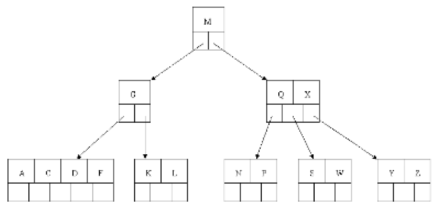
Delete R: Although R is in a leaf, this leaf does not have an extra key; the deletion results in a node with only one key, which is not acceptable for a B-tree of order 5. If the sibling node to the immediate left or right has an extra key, we can then borrow a key from the parent and move a key up from this sibling. In our specific case, the sibling to the right has an extra key. So, the successor W of S (the last key in the node where the deletion occurred), is moved down from the parent, and the X is moved up. (S and W are inserted in their proper place.) (Next E)



10

B-TREE

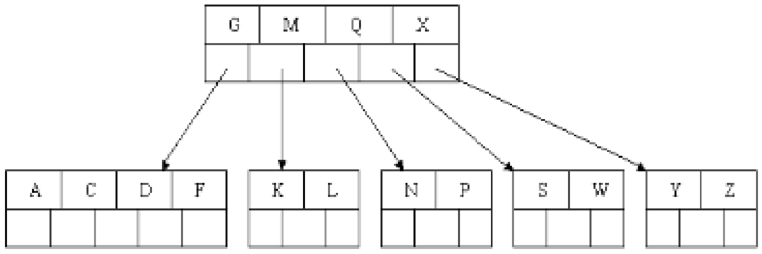
Delete E: This one causes lots of problems. Although E is in a leaf, the leaf has no extra keys, nor do the siblings to the immediate right or left. In such a case the leaf has to be combined with one of these two siblings. This includes moving down the parent's key that was between those of these two leaves. In our example, let's combine the leaf containing F with the leaf containing A C. We also move down the D.



11

B-TREE

Delete E: The parent node now contains only one key, G. This is not acceptable. If this problem node had a sibling to its immediate left or right that had a spare key, then we would again "borrow" a key. Since we have no way to borrow a key from a sibling, we must again combine with the sibling, and move down the M from the parent. In this case, the tree shrinks in height by one.

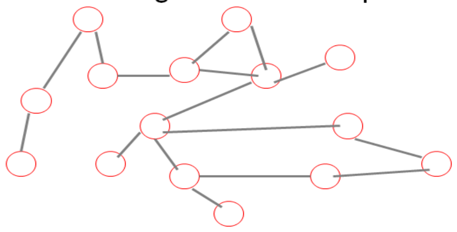


Graph

- A *graph* is a non-linear structure (also called a network)
- Unlike a tree or binary tree, a graph does not have a root – no primary entry point.
- Any node can be connected to any other node by an *edge*
- Can have any number of edges and nodes

- Analogy: the highway system connecting cities on a map

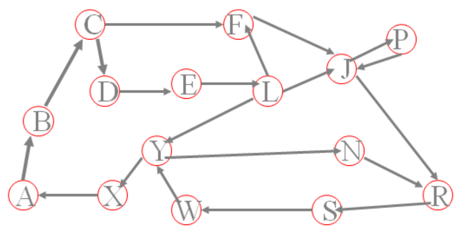
a graph data structure



DiGraphs

- Each edge of *directed graph* or *digraph* has a specific direction denoted by arrows.
- Edges with direction sometimes are called *arcs*
- Analogy #1: airline flights between airports (see below)
- Analogy #2: Solution to a problem (on board – miles on arcs)

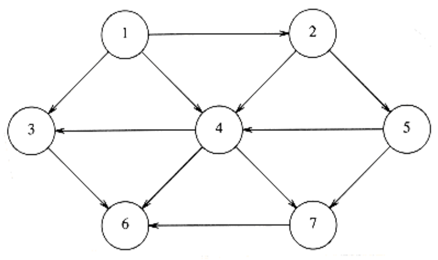
a directed graph



airline routes represented via digraph
What else could be provided?

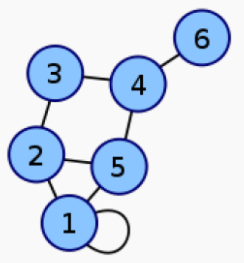
Adjacent Vertices

- In Undirected Graph → if one of the edge is (v1,v2) then v1 and v2 are adjacent to each other and the edge (v1,v2) is incident on vertices on v1 & v2.
- If <v1,v2> is a directed edge, then vertex v1 is said to be <u>adjacent to</u> v2 while v2 is <u>adjacent from</u> v1.
  Edge <v1,v2> is incident to v1 and v2
  Here vertex 3 is adjacent to 6 and edges incident on vertex 3 are <1,3>,<4,3>, <3,6>



Adjacency matrix

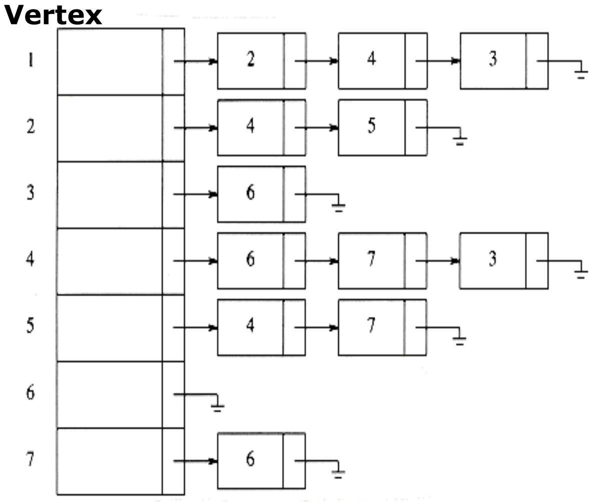
An ADJACENCY MATRIX is a means of representing which vertices of a graph are adjacent to which other vertices. Another matrix representation for a graph is the incidence matrix.
The adjacency matrix of a finite graph G with *n* vertices is the *n* × *n* matrix where the nondiagonal entry *aij* is the number of edges from vertex *i* to vertex *j*, and the diagonal entry *aii*, depending on the convention, is either once or twice the number of edges (loops) from vertex *i* to itself.
There exists a unique adjacency matrix for each graph (up to permuting rows and columns), and it is not the adjacency matrix of any other graph.
In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected, the adjacency matrix is symmetric and for directed graph it need not be symmetric.

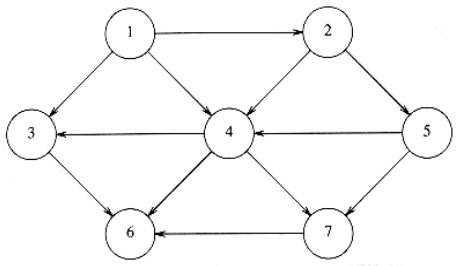
Labeled graph	Adjacency matrix
	$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$

- The adjacency matrix of a complete graph is all 1's except for 0's on the diagonal.
- The adjacency matrix of an empty graph is a zero matrix.

Adjacent List of directed graph

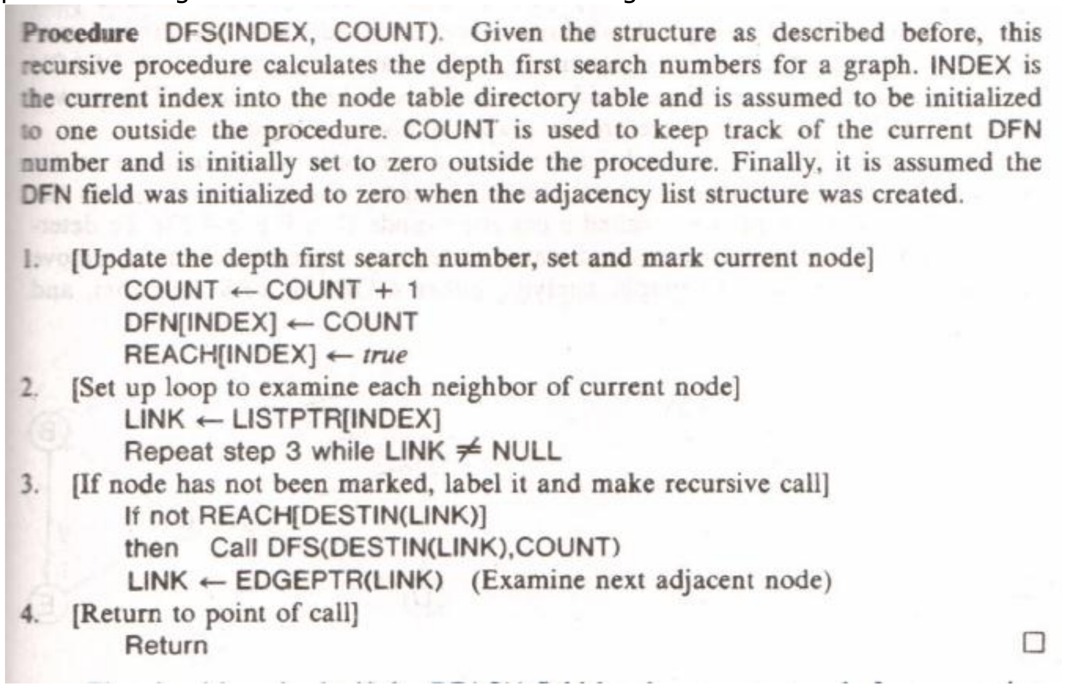
Vertex





Depth-first search (**DFS**)

Depth-first search (**DFS**) is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking.



Procedure DFS(INDEX, COUNT). Given the structure as described before, this recursive procedure calculates the depth first search numbers for a graph. INDEX is the current index into the node table directory table and is assumed to be initialized to one outside the procedure. COUNT is used to keep track of the current DFN number and is initially set to zero outside the procedure. Finally, it is assumed the DFN field was initialized to zero when the adjacency list structure was created.

```
1. [Update the depth first search number, set and mark current node]
       COUNT ← COUNT + 1
       DFN[INDEX] ← COUNT
       REACH[INDEX] ← true
2. [Set up loop to examine each neighbor of current node]
       LINK ← LISTPTR[INDEX]
       Repeat step 3 while LINK ≠ NULL
3. [If node has not been marked, label it and make recursive call]
       If not REACH[DESTIN(LINK)]
       then   Call DFS(DESTIN(LINK),COUNT)
       LINK ← EDGEPTR(LINK)   (Examine next adjacent node)
4. [Return to point of call]
       Return                                              □
```

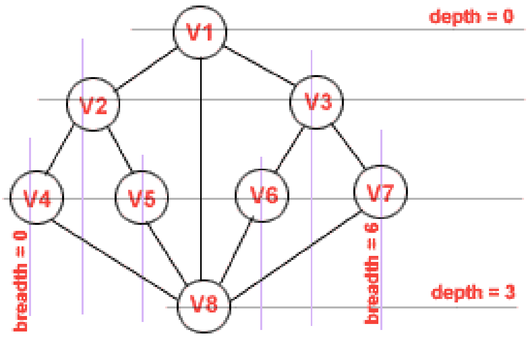
OR

ALGORITHM

```
Step-1: Set the starting point of traversal and push it inside the stack
Step-2: Pop the stack and add the popped vertex to the list of visited vertices
Step-3: Find the adjacent vertices for the most recently visited vertex( from the
        Adjacency Matrix)
Step-4: Push these adjacent vertices inside the stack (in the increasing order of their
        depth) if they are not visited and not there in the stack already
Step-5: Go to step-2 if the stack is not empty
```

The starting point can be any vertex and the next vertex to be visited is decided by the traversal you choose. If you choose the DFS, the next vertex to be visited will be the adjacent vertex (to the starting point), which has the highest depth value(??????...look at the graph below) and then the next adjacent vertex with the next higher depth value and so on till all the adjacent nodes for that vertex are not visited. We repeat this same procedure for every visited vertex of the graph.

If V1 is the starting point of my traversal then, as you can see - V2, V3 and V8 are its adjacent vertices and so in the Adjacency List of V1. Now, I am suppose to visit these vertices, but as this is DFS, I should visit V8 first and then V2 and V3. So, to keep track of which vertex to be visited next, we make use of a STACK, which holds these vertex values in the DFS order.

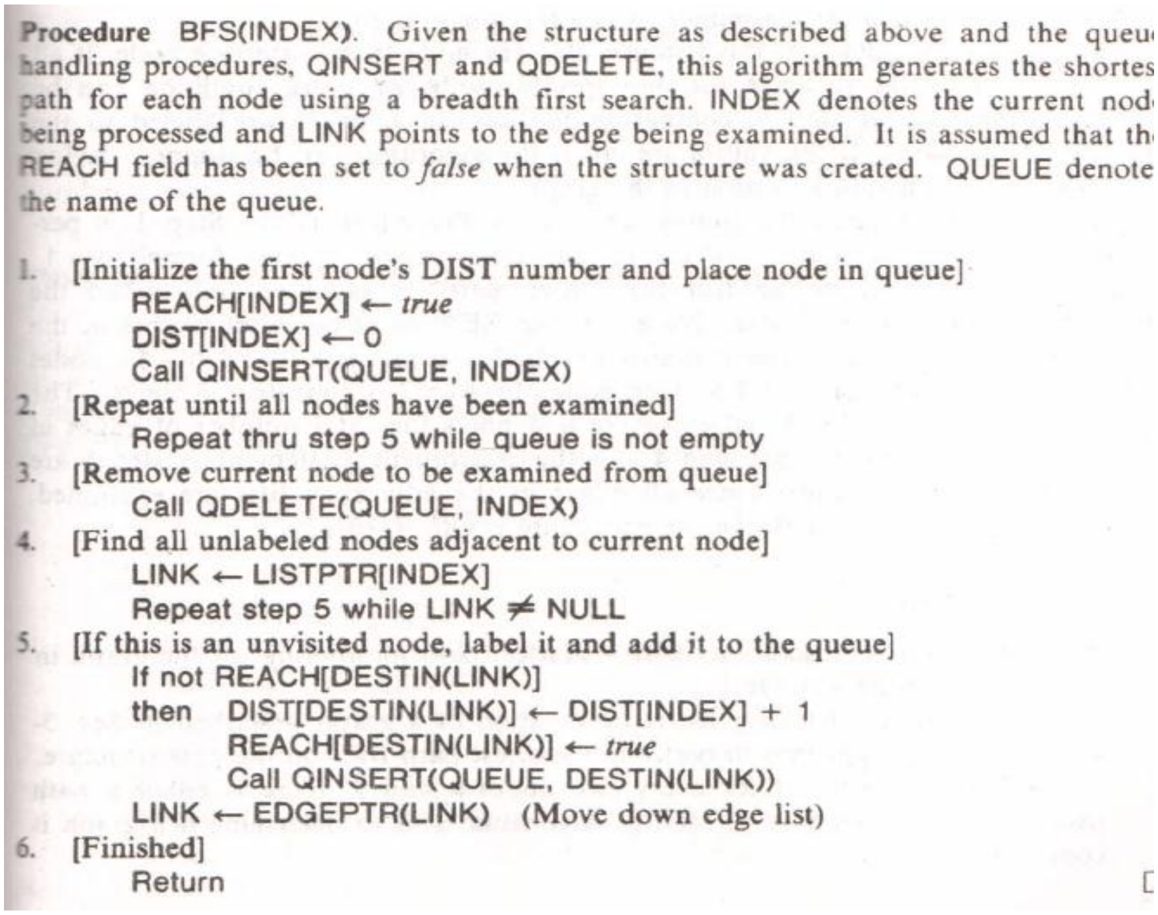


```
#include<stdio.h>
#include<conio.h>
#include<alloc.h>

void visit(int);
int visited(int);
int instack(int);
void push(int);
int count();
int pop();

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

struct node *bottom,*top;

int visited_list[10],k,nodes;

void main()
{
/*declare an array;*/
static int edges[10][10];
int i,j,vertex=0,adjnode;

clrscr();
bottom=(struct node*)malloc(sizeof(node));
top=bottom;
bottom->next=NULL;

/*get the no. of nodes from the user;*/
printf("Enter the number of nodes: ");
scanf("%d",&nodes);

/*loop: get the edges for these nodes;*/
for(i=0;i<nodes;i++)
 for(j=0;j<nodes;j++)
 {
 printf("Is there any edge between nodes %d and %d: ",i,j);
 scanf("%d",&edges[i][j]);
 }

/*---------------------------------------------------*/

/*while all the nodes are not visited do the following steps*/
do
{
  visit(vertex);/*visit the current node;*/
  for(j=0;j<nodes;j++)/*find all its adjacent nodes;*/
/*push them in the stack if they are not visited and not there in the stack;*/
```

```
            if(edges[vertex][j]==1)
                    {
                    adjnode=j;
                    if(!visited(adjnode) && !instack(adjnode))push(adjnode);
                    };
          vertex=pop();/*pop the unvisted node from the stack;*/
        }while(vertex!=NULL);/*end of while*/
        for(k=0;k<nodes;k++)printf("%d->",visited_list[k]);
        getch();
        }/*end of main*/

        /*------------------------------------------------------------*/
        void push(int value)
        {
        struct node *curr=(struct node*)malloc(sizeof(node));
        if(bottom->value==NULL)
          {
          bottom->value=value;
          top=bottom;
          }
        else
          {
          top->next=curr;
          curr->value=value;
          top=curr;
          curr->next=NULL;
          }
        }

        int pop()
        {
        struct node *curr;
        int value,no_nodes;
        curr=bottom;
        if(bottom==top && bottom->value!=NULL)
          {
          value=bottom->value;
          bottom->value=NULL;
          }
        else
          {
          no_nodes=count()-1;
          for(int i=1;i<no_nodes;i++)
                  curr=curr->next;
          top=curr;
          curr=curr->next;
          value=curr->value;
          top->next=NULL;
          free(curr);
          }
        return(value);
        }

        int count()
        {
        struct node *curr;
        int count=1;
        curr=bottom;
        while(curr->next!=NULL)
          {
          curr=curr->next;
          count++;
          }
        return(count);
        }

        void visit(int node)
        {
        /*add the node to the list of visited nodes;*/
        visited_list[k]=node;
```

```
k++;
}

int visited(int node)
{
int found=0,result;
/*check for this node in the list of visited nodes;*/
for(int i=0;i<nodes;i++)
  if(visited_list[i]==node)found=1;
if(found==1) result=1; else result=0;
return(result);
}

int instack(int node)
{
int found=0,result;
struct node *curr;
/*check the stack elements for the node;*/
curr=bottom;
if(bottom==top)
  {
  if(node==curr->value)found=1;
  }
else
  {
  while(curr->value!=NULL)
          {
          if(node==curr->value)found=1;curr=curr->next;
          }
  }
if(found==1)result=1; else result=0;
return(result);
}
```

Breadth First Search (BFS) → **Breadth-first search** (**BFS**) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal. In case above example for DFS make use of a QUEUE here instead of a stack and the adjacent vertices are inserted inside the queue in the increasing order of their BREADTH, so that the first adjacent vertex to be visited will be the one with the least breadth value.

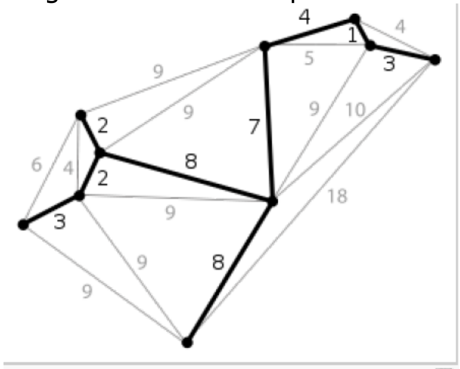


Procedure BFS(INDEX). Given the structure as described above and the queu handling procedures, QINSERT and QDELETE, this algorithm generates the shortes path for each node using a breadth first search. INDEX denotes the current nod being processed and LINK points to the edge being examined. It is assumed that th REACH field has been set to *false* when the structure was created. QUEUE denote the name of the queue.

```
1. [Initialize the first node's DIST number and place node in queue]
       REACH[INDEX] ← true
       DIST[INDEX] ← 0
       Call QINSERT(QUEUE, INDEX)
2. [Repeat until all nodes have been examined]
       Repeat thru step 5 while queue is not empty
3. [Remove current node to be examined from queue]
       Call QDELETE(QUEUE, INDEX)
4. [Find all unlabeled nodes adjacent to current node]
       LINK ← LISTPTR[INDEX]
       Repeat step 5 while LINK ≠ NULL
5. [If this is an unvisited node, label it and add it to the queue]
       If not REACH[DESTIN(LINK)]
       then   DIST[DESTIN(LINK)] ← DIST[INDEX] + 1
              REACH[DESTIN(LINK)] ← true
              Call QINSERT(QUEUE, DESTIN(LINK))
       LINK ← EDGEPTR(LINK)   (Move down edge list)
6. [Finished]
       Return
```

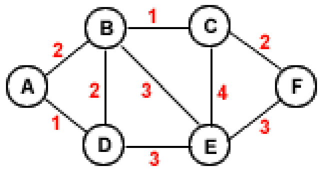
SPANNING TREE

A connected, undirected graph, a spanning tree of that graph is a subgraph which is a tree and connects all the vertices together. A single graph can have many different spanning trees. We can also assign a *weight* to each edge, which is a number representing how unfavorable it is, and use this to assign a weight to a spanning tree by computing the sum of the weights of the edges in that spanning tree. A minimum spanning tree (MST) or minimum weight spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree



The minimum spanning tree of a planar graph. Each edge is labeled with its weight, which here is roughly proportional to its length.

This is a little different algorithm and a minor diversion of the shortest path algorithm. In shortest path algorithm, we find the shortest path from all the vertices to a particular starting vertex. Here, its the shortest path between each vertex to every other vertex in the graph. For example



The least weighing edge is either AD or BC. Well, you can choose any of them. I choose AD. Add the edge AD to your Spanning Tree.

Now, find the adjacent edges to A and D.

AB - 2	**Least weighing**
AD - 1	**Already in the tree**
DB - 2	**Least weighing**
DE - 3	**> value**
DA - 1	**Already in the tree**

I add AB to the spanning tree.

So, you have edges (AD) and (AB) in the tree after this step.

Now, find the adjacent edges to B.

BA - 2	**Already in the tree**
BD - 2	**> value**
BC - 1	**Least weighing**
BE - 3	**> value**
DB - 2	**> value**
DE - 3	**> value**

Edges which exist in the tree and form loops are discarded and ones with > value are considered every time you find the least.

Now, BC is added to the tree.

Now, find the adjacent edges to C.

CB - 1	**Already in the tree**
CE - 4	**> value**
CF - 2	**Least weighing**
BD - 2	**Forming a loop**
BE - 3	**> value**
DB - 2	**Forming a loop**
DE - 3	**> value**

So, CF is added to the tree.

Find the adjacent edges to F.

FC - 2	**Already in the tree**
FE - 3	**least weighing**
CE - 4	**> value**
BE - 3	**least weighing**
DE - 3	**least weighing**

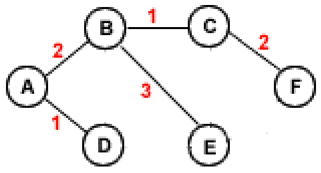
I add BE to the tree.

Find the adjacent edges to E.

EB - 3	**Already in the tree**
EC - 4	**Forming a loop**
ED - 3	**Forming a loop**
EF - 3	**Forming a loop**
FE - 3	**Forming a loop**
CE - 4	**Forming a loop**
DE - 3	**Forming a loop**

It means there no more edges left and that the tree is complete.

The resultant Spanning Tree with no loops giving you the shortest distance between any two vertices.



ALGORITHM

```
Step-1: Take the least weighing edge of the graph and add it to the minimum Spanning
        Tree
Step-2: Find the adjacent edges for the vertices in the spanning tree
Step-3: Find the least among all those edges
Step-4: Add that edge to the spanning tree if its not forming a loop in the tree and
        that its not already there in the tree.
Step-5: Stop if all vertices in the Graph have been covered by the Tree else
        go to step-2
```

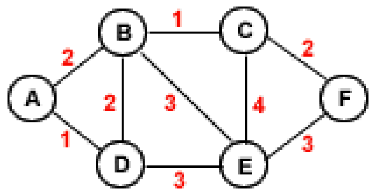
SHORTEST PATH

A **shortest path tree**, in graph theory, is a sub-graph of a given (possibly weighted) graph constructed so that the distance between a selected root node and all other nodes is minimal. It is a tree because if there are two paths between the root node and some vertex *v* (i.e. a cycle), we can delete the last edge of the longer path without increasing the distance from the root node to any node in the sub-graph.

If every pair of nodes in the graph has a unique shortest path between them, then the shortest path tree is unique. This is because if a particular path from the root to some vertex is minimal, then any part of that path (from node *u* to node *v*) is a minimal path between these two nodes.

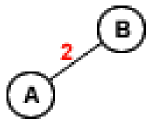
ALGORITHM

```
Step-1: Set the starting vertex
Step-2: Find the BFS (or DFS) for it
Step-3: For every vertex in the BFS
           A. Find the shortest path from it to the starting vertex
           B. Add that shortest path to the resultant TREE if it's already not there and
              not forming a loop
```

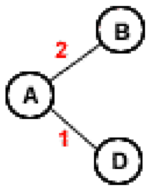


Set A as the starting vertex. So, the BFS becomes - A->B->D->C->E->F (Don't ask me why?...find out).

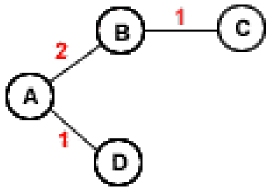
Shortest path from B to A is (BA) = 2



Shortest path from D to A is (DA) = 1

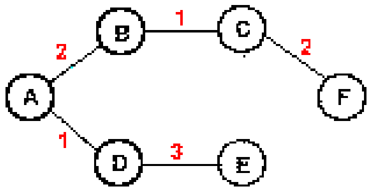


Shortest path from C to A is (CBA) = 3



Shortest path from E to A is (EDA) = 4

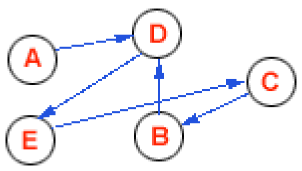
Shortest path from F to A is (FCBA) = 5



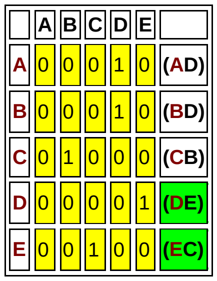
You can see there are no alternative paths and no loops in the resultant Shortest Path tree. The tree gives you the shortest path from A to rest of the vertices.

Transitive Closure

We have seen Adjacency Matrix, which tells you whose directly connected to whom. There can be many other paths which may exist and might not be visible from the Adjacency matrix, as these paths may have intermediate vertices. To find all such round about or via-paths there is a procedure to be followed, known as the transitive closure, which results in a matrix showing all those via-paths, if ever exist.
The input to this algorithm is an adjacency matrix, which has 1s for any row-column pair if there exists an edge between them. Check for every vertex, the incoming and outgoing links. If there exist both incoming and outgoing links for a particular vertex, put a 1 for the row-column pair formed by them in the adjacency matrix. For example



Directed Graph



	A	B	C	D	E	
A	0	0	0	1	0	(AD)
B	0	0	0	1	0	(BD)
C	0	1	0	0	0	(CB)
D	0	0	0	0	1	(DE)
E	0	0	1	0	0	(EC)

Adjacency Matrix

Take the vertex E into consideration. You have an edge coming from **D** and you have an edge going to **C**; quite visible from the matrix. So, as I said, we'll put a 1 for the element **DC** in the matrix. Similarly, for the vertex C, we'll put 1 for the element **EB**. We'll do this for all the vertex in the graph and what you get at the end is the Transitive Closure of this graph. This way of finding the transitive closure was derived by Warshall and thus the name, Warshall's algorithm.

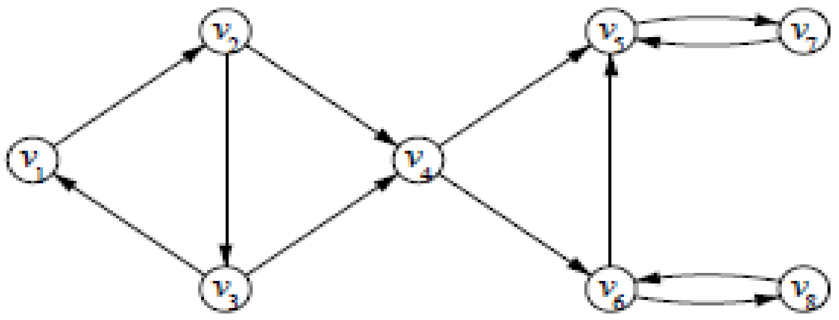
ALGORITHM

Step-1: Copy the Adjacency matrix into another matrix called the Path matrix
Step-2: Find in the Path matrix for every element in the Graph, the incoming and outgoing edges
Step-3: For every such pair of incoming and outgoing edges put a 1 in the Path matrix

C IMPLEMENTATION

```
transclosure( int adjmat[max][max], int path[max][max])
{
 for(i = 0; i < max; i++)
  for(j = 0; j < max; j++)
      path[i][j] = adjmat[i][j];

 for(i = 0;i <max; i++)
  for(j = 0;j < max; j++)
   if(path[i][j] == 1)
    for(k = 0; k < max; k++)
      if(path[j][k] == 1)
        path[i][k] = 1;
}
```

In Figure 2.1, we present an example graph $G = (V, E)$. The vertices are shown as circles and the edges as arrows going from the tail of the edge to the head of the edge. Below G we present its adjacency matrix and adjacency list representations. □



$$
\begin{array}{c}
 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8
\end{array}
\begin{array}{c}
\begin{array}{cccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{array} \\
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\end{array}
$$

$$
\begin{aligned}
AdjFrom(v_1) &= \{v_2\} \\
AdjFrom(v_2) &= \{v_3, v_4\} \\
AdjFrom(v_3) &= \{v_1, v_4\} \\
AdjFrom(v_4) &= \{v_5, v_6\} \\
AdjFrom(v_5) &= \{v_7\} \\
AdjFrom(v_6) &= \{v_5, v_8\} \\
AdjFrom(v_7) &= \{v_5\} \\
AdjFrom(v_8) &= \{v_6\}
\end{aligned}
$$

FIGURE 2.1: An example graph $G = (V, E)$ and its representations.

Definition. The *transitive closure of graph* $G = (V, E)$ is a graph $G^+ = (V, E^+)$ such that E^+ contains an edge (v, w) iff G contains a non-null path $v \overset{+}{\to} w$. The size of the transitive closure is denoted by e^+. The *successor set* of a vertex v is the set $Succ(v) = \{w \mid (v, w) \in E^+\}$, i.e., the set of all vertices that can be reached from vertex v via non-null paths. The *predecessor set* of a vertex v is the set $Pred(v) = \{u \mid (u, v) \in E^+\}$, i.e., the set of all vertices that v is reachable from via non-null paths. The vertices adjacent from vertex v are the *immediate successors of* v and the vertices adjacent to v are the *immediate predecessors of* v.

The transitive closure of graph G of Figure 2.1 is presented in Figure 2.2. As we see, the successor set of vertices v_1, v_2, and v_3 is V; the successor set of vertices v_4, v_6, and v_8 is $\{v_5, v_6, v_7, v_8\}$; and the successor set of v_5 and v_7 is $\{v_5, v_7\}$. Similarly, several vertices have a common predecessor set. □

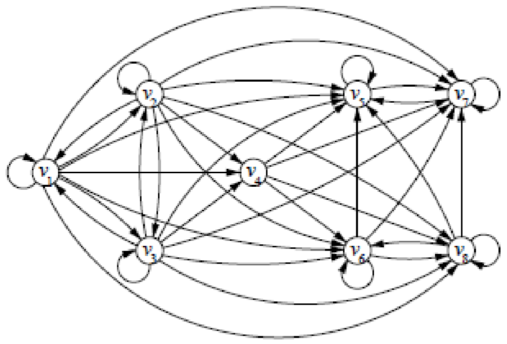


FIGURE 2.2: The transitive closure of graph G of Figure 2.1.

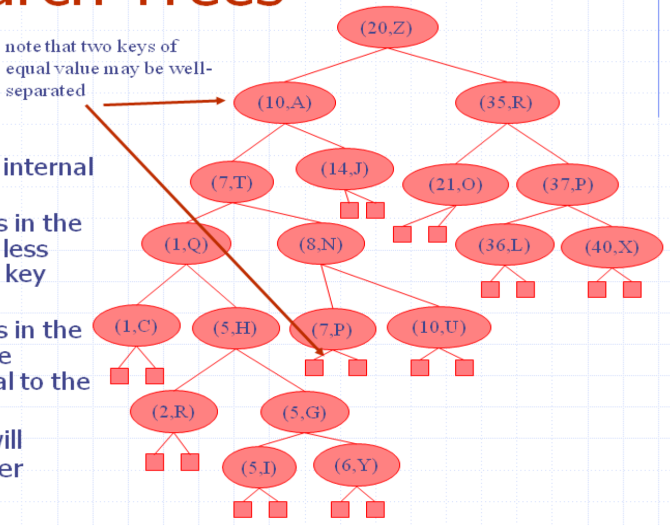
SPLAY TREE

A splay tree is a self-balancing binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in O(log(n)) amortized time. For many non-uniform sequences of operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown.

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top.

Splay Trees are Binary Search Trees

- BST Rules:
  - items stored only at internal nodes
  - keys stored at nodes in the left subtree of *v* are less than or equal to the key stored at *v*
  - keys stored at nodes in the right subtree of *v* are greater than or equal to the key stored at *v*
- An inorder traversal will return the keys in order

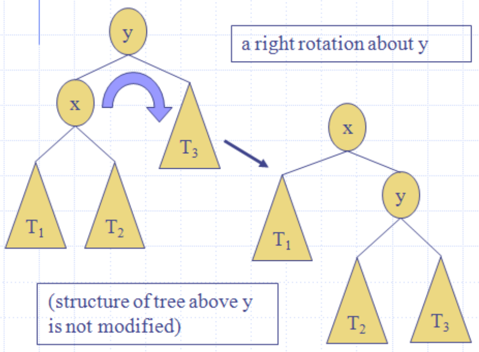


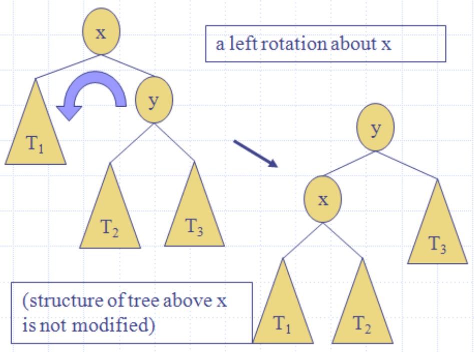
Splay Trees 2

Splay Trees do Rotations after Every Operation (Even Search)

- new operation: ***splay***
  - splaying moves a node to the root using rotations

- right rotation
  - makes the left child *x* of a node *y* into *y*'s parent; *y* becomes the right child of *x*
- left rotation
  - makes the right child *y* of a node *x* into *x*'s parent; *x* becomes the left child of *y*





Splay Trees 5