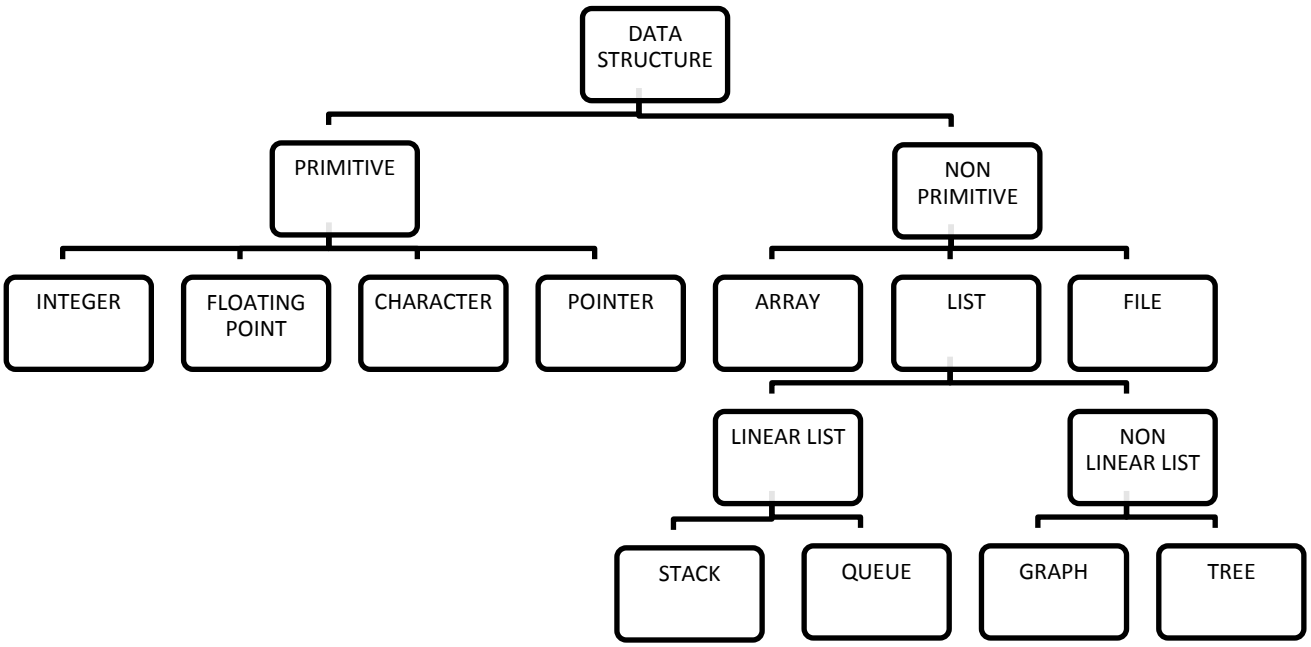
Introduction to Data Structure

- Computer is an electronic machine which is used for data processing and manipulation.
- When programmer collects such type of data for processing, he would require to store all of them in computer's main memory.
- In order to make computer work we need to know
  - Representation of data in computer.
  - Accessing of data.
  - How to solve problem step by step.
- For doing this task we use data structure.

What is Data Structure?

- **Data structure** is a representation of the logical relationship existing between individual elements of data.
- Data Structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.
- We can also define data structure as a mathematical or logical model of a particular organization of data items.
- The representation of particular data structure in the main memory of a computer is called as **storage structure.**
- The storage structure representation in auxiliary memory is called as **file structure**.
- It is defined as the way of storing and manipulating data in organized form so that it can be used efficiently.
- Data Structure mainly specifies the following four things
  - Organization of Data
  - Accessing methods
  - Degree of associativity
  - Processing alternatives for information
- Algorithm + Data Structure = Program
- Data structure study covers the following points
  - Amount of memory require to store.
  - Amount of time require to process.
  - Representation of data in memory.
  - Operations performed on that data.

Classification of Data Structure



Data Structures are normally classified into two broad categories

1. Primitive Data Structure
2. Non-primitive data Structure

Data types

A particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

Primitive Data Structure

- Primitive data structures are basic structures and are directly operated upon by machine instructions.
- Primitive data structures have different representations on different computers.
- Integers, floats, character and pointers are examples of primitive data structures.
- These data types are available in most programming languages as built in type.
  - Integer: It is a data type which allows all values without fraction part. We can use it for whole numbers.
  - Float: It is a data type which use for storing fractional numbers.
  - Character: It is a data type which is used for character values.

Pointer: A variable that holds memory address of another variable are called pointer.

Non primitive Data Type

- These are more sophisticated data structures.
- These are derived from primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous or heterogeneous data items.
- Examples of Non-primitive data type are Array, List, and File etc.
- A Non-primitive data type is further divided into Linear and Non-Linear data structure
  - **Array:** An array is a fixed-size sequenced collection of elements of the same data type.
  - **List:** An ordered set containing variable number of elements is called as Lists.
  - **File:** A file is a collection of logically related information. It can be viewed as a large list of records consisting of various fields.

Linear data structures

- A data structure is said to be Linear, if its elements are connected in linear fashion by means of logically or in sequence memory locations.
- There are two ways to represent a linear data structure in memory,
  - Static memory allocation
  - Dynamic memory allocation
- The possible operations on the linear data structure are: Traversal, Insertion, Deletion, Searching, Sorting and Merging.
- Examples of Linear Data Structure are Stack and Queue.
- Stack: Stack is a data structure in which insertion and deletion operations are performed at one end only.
  - The insertion operation is referred to as 'PUSH' and deletion operation is referred to as 'POP' operation.
  - Stack is also called as Last in First out (LIFO) data structure.
- Queue: The data structure which permits the insertion at one end and Deletion at another end, known as Queue.
  - End at which deletion is occurs is known as FRONT end and another end at which insertion occurs is known as REAR end.
  - Queue is also called as First in First out (FIFO) data structure.

Nonlinear data structures

- Nonlinear data structures are those data structure in which data items are not arranged in a sequence.
- Examples of Non-linear Data Structure are Tree and Graph.
- **Tree:** A tree can be defined as finite set of data items (nodes) in which data items are arranged in branches and sub branches according to requirement.
  - Trees represent the hierarchical relationship between various elements.
  - Tree consist of nodes connected by edge, the node represented by circle and edge lives connecting to circle.

- **Graph:** Graph is a collection of nodes (Information) and connecting edges (Logical relation) between nodes.
  - A tree can be viewed as restricted graph.
  - Graphs have many types:
    - Un-directed Graph
    - Directed Graph
    - Mixed Graph
    - Multi Graph
    - Simple Graph
    - Null Graph
    - Weighted Graph

Difference between Linear and Non Linear Data Structure

Linear Data Structure	Non-Linear Data Structure
Every item is related to its previous and next time.	Every item is attached with many other items.
Data is arranged in linear sequence.	Data is not arranged in sequence.
Data items can be traversed in a single run.	Data cannot be traversed in a single run.
Eg. Array, Stacks, linked list, queue.	Eg. tree, graph.
Implementation is easy.	Implementation is difficult.

Operation on Data Structures

Design of efficient data structure must take operations to be performed on the data structures into account. The most commonly used operations on data structure are broadly categorized into following types

1. **Create**
   The create operation results in reserving memory for program elements. This can be done by declaration statement. Creation of data structure may take place either during compile-time or run-time. malloc() function of C language is used for creation.

2. **Destroy**
   Destroy operation destroys memory space allocated for specified data structure. free() function of C language is used to destroy data structure.

3. **Selection**
   Selection operation deals with accessing a particular data within a data structure.

4. **Updation**
   It updates or modifies the data in the data structure.

5. **Searching**
   It finds the presence of desired data item in the list of data items, it may also find the locations of all elements that satisfy certain conditions.

6. **Sorting**
   Sorting is a process of arranging all data items in a data structure in a particular order, say for example, either in ascending order or in descending order.

7. **Merging**
   Merging is a process of combining the data items of two different sorted list into a single sorted list.

8. **Splitting**
   Splitting is a process of partitioning single list to multiple list.

9. **Traversal**
   Traversal is a process of visiting each and every node of a list in systematic manner.

Time and space analysis of algorithms

Algorithm

- An essential aspect to data structures is algorithms.
- Data structures are implemented using algorithms.
- An algorithm is a procedure that you can write as a C function or program, or any other language.
- An algorithm states explicitly how the data will be manipulated.

Algorithm Efficiency

- Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm, so it would be nice to have metrics for comparing algorithm efficiency.
- The complexity of an algorithm is a function describing the efficiency of the algorithm in terms of the amount of data the algorithm must process.
- Usually there are natural units for the domain and range of this function. There are two main complexity measures of the efficiency of an algorithm
- **Time complexity**
  - **Time Complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.

- "Time" can mean the number of memory accesses performed, the number of comparisons between integers, the number of times some inner loop is executed, or some other natural unit related to the amount of real time the algorithm will take.

- **Space complexity**
  - **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.
  - We often speak of "extra" memory needed, not counting the memory needed to store the input itself. Again, we use natural (but fixed-length) units to measure this.
  - We can use bytes, but it's easier to use, say, number of integers used, number of fixed-sized structures, etc. In the end, the function we come up with will be independent of the actual number of bytes needed to represent the unit.
  - Space complexity is sometimes ignored because the space used is minimal and/or obvious, but sometimes it becomes as important an issue as time.

Worst Case Analysis

In the worst case analysis, we calculate upper bound on running time of an algorithm. We must know the case that causes maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched is not present in the array. When x is not present, the search () functions compares it with all the elements of array [] one by one. Therefore, the worst case time complexity of linear search would be.

Average Case Analysis

In average case analysis, we take all possible inputs and calculate computing time for all of the inputs. Sum all the calculated values and divide the sum by total number of inputs. We must know (or predict) distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed. So we sum all the cases and divide the sum by (n+1).

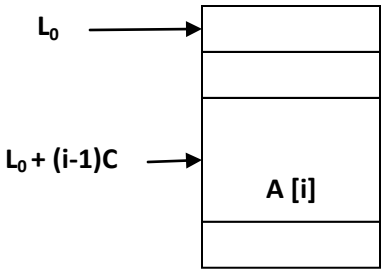
Best Case Analysis

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in worst case is constant (not dependent on n). So time complexity in the best case would be.

Explain Array in detail

One Dimensional Array

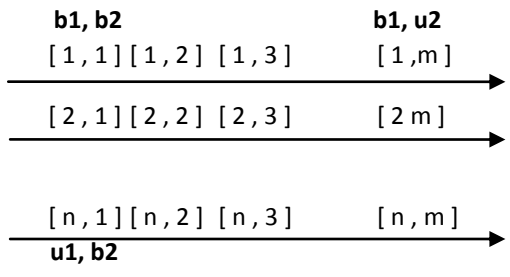
- Simplest data structure that makes use of computed address to locate its elements is the one-dimensional array or vector; number of memory locations is sequentially allocated to the vector.
- A vector size is fixed and therefore requires a fixed number of memory locations.
- Vector A with subscript lower bound of "one" is represented as below….



- L_0 is the address of the first word allocated to the first element of vector A.
- C words are allocated for each element or node
- The address of A_i is given equation **Loc (A_i) = L_0 + C (i-1)**
- Let's consider the more general case of representing a vector A whose lower bound for it's subscript is given by some variable b. The location of Ai is then given by **Loc (A_i) = L_0 + C (i-b)**

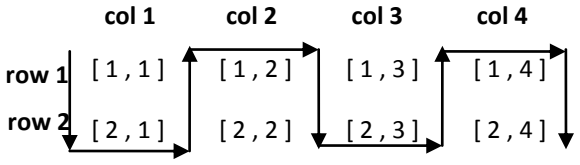
Two Dimensional Array

- Two dimensional arrays are also called table or matrix, two dimensional arrays have two subscripts
- Two dimensional array in which elements are stored column by column is called as column major matrix
- Two dimensional array in which elements are stored row by row is called as row major matrix
- First subscript denotes number of rows and second subscript denotes the number of columns
- Two dimensional array consisting of two rows and four columns as above Fig is stored sequentially by columns : A [1, 1], A [2 , 1], A [1 , 2], A [2 , 2], A [1 , 3], A [2 , 3], A [1, 4], A [2, 4]
- The address of element A [i , j] can be obtained by expression **Loc (A [i , j]) = L_0 + (j-1)*2 + i-1**
- In general for two dimensional array consisting of n rows and m columns the address element A [i , j] is given by **Loc (A [i , j]) = L_0 + (j-1)*n + (i – 1)**
- In row major matrix, array can be generalized to arbitrary lower and upper bound in its subscripts, assume that b1 ≤ I ≤ u1 and b2 ≤ j ≤u2



Row major matrix

No of Columns = m = u2 – b2 + 1



Column major matrix

- For row major matrix : **Loc (A [i , j]) = L_0 + (i – b1) *(u2-b2+1) + (j-b2)**

Applications of Array

1. Symbol Manipulation (matrix representation of polynomial equation)
2. Sparse Matrix

Symbol Manipulation using Array

- We can use array for different kind of operations in polynomial equation such as addition, subtraction, division, differentiation etc…
- We are interested in finding suitable representation for polynomial so that different operations like addition, subtraction etc… can be performed in efficient manner
- Array can be used to represent Polynomial equation

- **Matrix Representation of Polynomial equation**

	Y	**Y^2**	**Y^3**	**Y^4**
X	X Y	$X Y^2$	$X Y^3$	$X Y^4$
X^2	X^2 Y	$X^2 Y^2$	$X^2 Y^3$	$X^2 Y^4$
X^3	X^3 Y	$X^3 Y^2$	$X^3 Y^3$	$X^3 Y^4$
X^4	X^4 Y	$X^4 Y^2$	$X^4 Y^3$	$X^4 Y^4$

e.g. **$2x^2+5xy+Y^2$**
is represented in matrix form as below

		Y	**Y^2**	**Y^3**	**Y^4**
	0	0	**1**	0	0
X	0	**5**	0	0	0
X^2	**2**	0	0	0	0
X^3	0	0	0	0	0
X^4	0	0	0	0	0

e.g. **$x^2+3xy+Y^2+Y-X$**
is represented in matrix form as below

		Y	**Y^2**	**Y^3**	**Y^4**
	0	0	**1**	0	0
X	**-1**	**3**	0	0	0
X^2	**1**	0	0	0	0
X^3	0	0	0	0	0
X^4	0	0	0	0	0

- Once we have algorithm for converting the polynomial equation to an array representation and another algorithm for converting array to polynomial equation, then different operations in array (matrix) will be corresponding operations of polynomial equation

What is sparse matrix? Explain

- An mXn matrix is said to be sparse if “many” of its elements are zero.
- A matrix that is not sparse is called a dense matrix.
- We can device a simple representation scheme whose space requirement equals the size of the non-zero elements.

- **Example:-**
  - The non-zero entries of a sparse matrix may be mapped into a linear list in row-major order.
  - For example the non-zero entries of 4X8 matrix of below fig.(a) in row major order are 2, 1, 6, 7, 3, 9, 8, 4, 5

0	0	0	2	0	0	1	0
0	6	0	0	7	0	0	3
0	0	0	9	0	8	0	0
0	4	5	0	0	0	0	0

Fig (a) 4 x 8 matrix

Terms	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Row	1	1	2	2	2	3	3	4	4
Column	4	7	2	5	8	4	6	2	3
Value	2	1	6	7	3	9	8	4	5

Fig (b) Linear Representation of above matrix

- To construct matrix structure we need to record
  (a) Original row and columns of each non zero entries
  (b) No of rows and columns in the matrix
- So each element of the array into which the sparse matrix is mapped need to have three fields: row, column and value
- A corresponding amount of time is saved creating the linear list representation over initialization of two dimension array.

A =

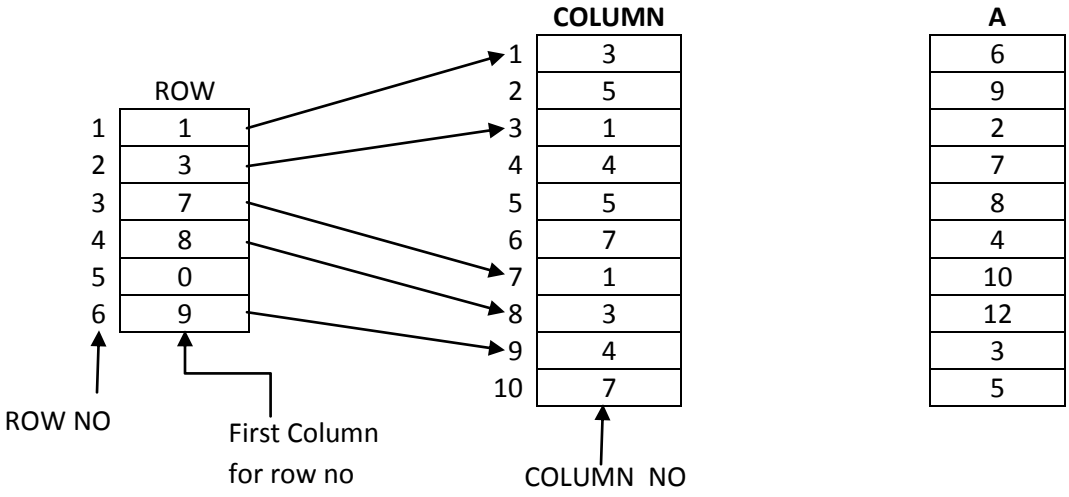
0	0	6	0	9	0	0
2	0	0	7	8	0	4
10	0	0	0	0	0	0
0	0	12	0	0	0	0
0	0	0	0	0	0	0
0	0	0	3	0	0	5

- Here from 6X7=42 elements, only 10 are non zero. A[1,3]=6, A[1,5]=9, A[2,1]=2, A[2,4]=7, A[2,5]=8, A[2,7]=4, A[3,1]=10, A[4,3]=12, A[6,4]=3, A[6,7]=5.
- One basic method for storing such a sparse matrix is to store non-zero elements in one dimensional array and to identify each array elements with row and column indices fig (c).

	ROW	**COLUMN**	**A**
1	1	3	6
2	1	5	9

3	2	1	2
4	2	4	7
5	2	5	8
6	2	7	4
7	3	1	10
8	4	3	12
9	6	4	3
10	6	7	5

Fig (c)

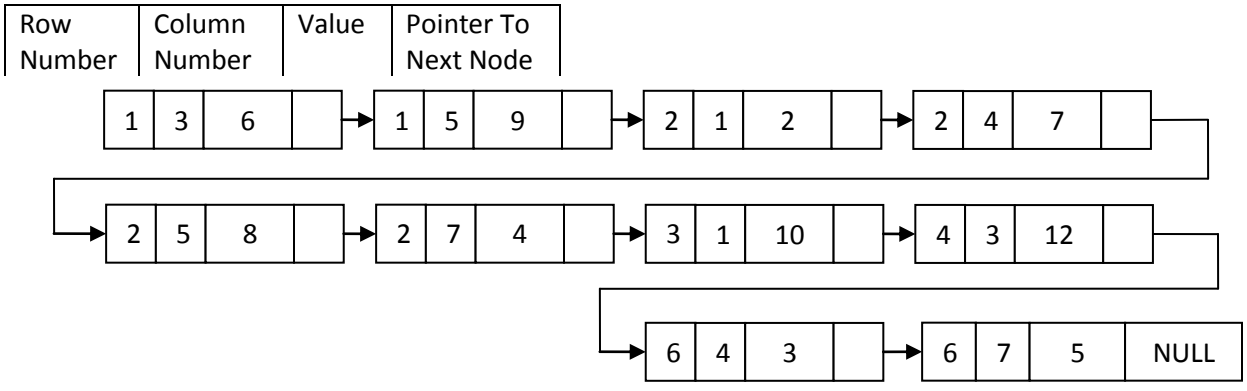


Fig(d)

- A more efficient representation in terms of storage requirement and access time to the row of the matrix is shown in fid (d). The row vector changed so that its i^{th} element is the index to the first of the column indices for the element in row I of the matrix.

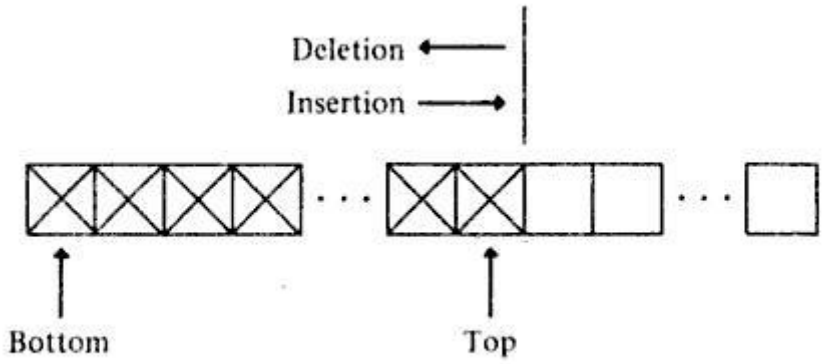
Linked Representation of Sparse matrix

Typical node to represent non-zero element is



Write algorithms for Stack Operations – PUSH, POP, PEEP

- A linear list which allows insertion and deletion of an element at one end only is called stack.
- The insertion operation is called as **PUSH** and deletion operation as **POP**.
- The most and least accessible elements in stack are known as top and bottom of the stack respectively.
- Since insertion and deletion operations are performed at one end of a stack, the elements can only be removed in the opposite orders from that in which they were added to the stack; such a linear list is referred to as a LIFO (last in first out) list.



Alternative representation of a stack

- A pointer TOP keeps track of the top element in the stack. Initially, when the stack is empty, TOP has a value of "one" and 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.

Applications of Stack

- Recursion
- Keeping track of function calls
- Evaluation of expressions
- Reversing characters
- Servicing hardware interrupts
- Solving combinatorial problems using backtracking.

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 elements 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

Function : POP (S, TOP)

- This function removes the top element from a stack which is represented by a vector S and returns this element. TOP is a pointer to the top element of the stack.

1. [Check for underflow of stack]
   If TOP = 0
   Then Write ('STACK UNDERFLOW ON POP')
   Take action in response to underflow
   Return
2. [Decrement Pointer]
   TOP ← TOP – 1
3. [Return former top element of stack]
   Return (S[TOP + 1])

Function : PEEP (S, TOP, I)

- This function returns the value of the i^{th} element from the TOP of the stack which is represented by a vector S containing N elements. 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 I^{th} element from top of the stack
   Return (S[TOP – I + 1])

Write an algorithm to change the i^{th} value of stack to value X

PROCEDURE : CHANGE (S, TOP, X, I)

- This procedure changes the value of the I^{th} element from the top of the stack to the value containing in X. Stack is represented by a vector S containing N elements.

1. [Check for stack Underflow]
If TOP – I + 1 ≤ 0
Then Write ('STACK UNDERFLOW ON CHANGE')
Return

2. [Change I^{th} element from top of the stack]
S[TOP – I + 1] ← X

3. [Finished]
Return

Write an algorithm which will check that the given string belongs to following grammar or not. L={wcw^R | w Є {a,b}*}(Where w^R is the reverse of w)

Algorithm : RECOGNIZE

- Given an input string named STRING on the alphabet {a, b, c} which contains a blank in its rightmost character position and function NEXTCHAR which returns the next symbol in STRING, this algorithm determines whether the contents of STRING belong to the above language. The vector S represents the stack, and TOP is a pointer to the top element of the stack.

1. [Initialize stack by placing a letter 'c' on the top]
TOP ← 1
S [TOP] ← 'c'

2. [Get and stack symbols either 'c' or blank is encountered]
NEXT ← NEXTCHAR (STRING)
Repeat while NEXT ≠ 'c'
    If NEXT = ' '
    Then Write ('Invalid String')
        Exit
    Else Call PUSH (S, TOP, NEXT)
        NEXT ← NEXTCHAR (STRING)

3. [Scan characters following 'c'; Compare them to the characters on stack]
Repeat While S [TOP] ≠ 'c'
    NEXT ← NEXTCHAR (STRING)
    X ← POP (S, TOP)
    If NEXT ≠ X
    Then Write ('INVALID STRING')
        Exit

4. [Next symbol must be blank]
If NEXT ≠ ' '
Then Write ('VALID STRING')
Else Write ('INVALID STRING')

5. [Finished]
Exit

Write an algorithm for push, pop and empty operations on stack. Using above functions write an algorithm to determine if an input character string is of the form a^ib^i where i>=1 i.e. no of a should be equal to no of b

Algorithm RECOGNIZE

- Given an input string named STRING on alphabet 'a' and 'b' which contain blank (' ') on right most character function NEXTCHAR which returns the next symbol from STRING. This algorithm determines if an input string is of form a^ib^i where i>1 i.e no of 'a' should be equal to no of 'b'. the vector S represent the stack and TOP is the pointer to the top element of stack. Counter is a counter B for 'b' occurrence.

1. [Initialize stack and counter]
   TOP ← 0
   COUNTER_B ← 0
2. [Get and stack character 'a' from whole string and count the occurrence of 'b']
   NEXT ← NEXTCHAR(STRING)
   Repeat while NEXT != ' '
   IF NEXT = 'a'
   Then PUSH (S,TOP,NEXT)
   Else COUNTER_B ← COUNTER_B+1
   NEXT ←NEXTCHAR(STRING)
3. [Pop the stack until empty and decrement the COUNTER_B]
   Repeat while TOP != 0
   POP (S,TOP)
   COUNTER_B← COUNTER_B-1
4. [Check for grammar]
   If COUNTER_B != 0
   Then write ('INVALID STRING')
   Else write ('VALID STRING')

What is recursion? Write a C program for GCD using recursion.

- A procedure that contains a procedure call to itself or a procedure call to second procedure which eventually causes the first procedure to be called is known as recursive procedure.
- There are two important conditions that must be satisfied by any recursive procedure
  a. Each time a procedure calls itself it must be nearer in some sense to a solution
  b. There must be a decision criterion for stopping the process or computation
- There are two types of recursion
  - Primitive Recursion: this is recursive defined function. E.g. Factorial function
  - Non-Primitive Recursion: this is recursive use of procedure. E.g. Find GCD of given two nunbers

C program for GCD using recursion

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

void main()
{
      int n1, n2, gcd;
      scanf("%d %d",&n1, &n2);
      gcd = Find_GCD(n1, &n2);
      printf("GCD of %d and %d is %d", n1, n2, gcd);
}

int Find_GCD(int m, int n)
{
      int gcdVal;
      if(n>m)
      {
            gcdVal = Find_GCD(n,m);
      }
      else if(n==0)
      {
            gcdVal = m;
      }
      else
      {
            gcdVal = Find_GCD(n, m%n);
      }
      return(gcdVal);
}
```

Write an algorithm to find factorial of given no using recursion

Algorithm: FACTORIAL

Given integer N, this algorithm computes factorial of N. Stack A is used to store an activation record associated with each recursive call. Each activation record contains the current value of N and the current return address RET_ADDE. TEMP_REC is also a record which contains two variables PARAM & ADDRESS.TOP is a pointer to the top element of stack A. Initially return address is set to the main calling address. PARAM is set to initial value N.

1. [Save N and return Address]
   CALL PUSH (A, TOP, TEMP_REC)
2. [Is the base criterion found?]
   If N=0
   then FACTORIAL← 1
   GO TO Step 4
   Else PARAM← N-1
   ADDRESS← Step 3
   GO TO Step 1
3. [Calculate N!]
   FACTORIAL← N * FACTORIAL
4. [Restore previous N and return address]
   TEMP_REC←POP(A,TOP)
   (i.e. PARAM←N, ADDRESS←RET_ADDR)
   GO TO ADDRESS

Give difference between recursion and iteration

Iteration	Recursion
In iteration, a problem is converted into a train of steps that are finished one at a time, one after another	Recursion is like piling all of those steps on top of each other and then quashing them all into the solution.
With iteration, each step clearly leads onto the next, like stepping stones across a river	In recursion, each step replicates itself at a smaller scale, so that all of them combined together eventually solve the problem.
Any iterative problem is solved recursively	Not all recursive problem can solved by iteration
It does not use Stack	It uses Stack

Write an algorithm to convert infix expression to postfix expression.

Symbol	Input precedence function F	Stack precedence function G	Rank function R
+, -	1	2	-1
*, /	3	4	-1
^	6	5	-1
Variables	7	8	1
(	9	0	-
)	0	-	-

Algorithm : REVPOL

- Given an input string INFIX containing an infix expression which has been padded on the right with ')' and whose symbol have precedence value given by above table, a vector S used as a stack and a NEXTCHAR which when invoked returns the next character of its argument. This algorithm converts INFIX into reverse polish and places the result in the string POLISH. The integer variable TOP denotes the top of the stack. Algorithm PUSH and POP are used for stack manipulation. The integer variable RANK accumulates the rank of expression. Finally the string variable TEMP is used for temporary storage purpose.

1. [Initialize stack]
   TOP ← 1
   S[TOP] ← '('
2. [Initialize output string and rank count]
   POLISH ← ' '
   RANK ← 0
3. [Get first input symbol]
   NEXT←NEXTCHAR (INFIX)
4. [Translate the infix expression]
   Repeat thru step 7 while NEXT != ' '
5. [Remove symbols with greater precedence from stack]
   IF TOP < 1
   Then write ('INVALID')
   EXIT
   Repeat while G (S[TOP]) > F(NEXT)
   TEMP ← POP (S, TOP)
   POLISH ← POLISH O TEMP
   RANK ← RANK + R(TEMP)
   IF RANK <1
   Then write 'INVALID')
   EXIT
6. [Are there matching parentheses]
   IF G(S[TOP]) != F(NEXT)
   Then call PUSH (S,TOP, NEXT)
   Else POP (S,TOP)
7. [Get next symbol]
   NEXT ← NEXTCHAR(INFIX)
8. [Is the expression valid]
   IF TOP != 0 OR RANK != 1
   Then write ('INVALID ')
   Else write ('VALID ')

Trace the conversion of infix to postfix form in tabular form.

(i) (A + B * C / D - E + F / G / (H + I))

Input Symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
A	((		0
+	((+	A	1
B	((+ B	A	1
*	((+ *	A B	2
C	((+ * C	A B	2
/	((+ /	A B C *	2
D	((+ / D	A B C *	2
-	((-	A B C * D / +	1
E	((- E	A B C * D / +	1
+	((+	A B C * D / + E -	1
F	((+ F	A B C * D / + E -	1
/	((+ /	A B C * D / + E – F	2
G	((+ / G	A B C * D / + E – F	2
/	((+ /	A B C * D / + E – F G /	2
(	((+ / (	A B C * D / + E – F G /	2
H	((+ / (H	A B C * D / + E – F G /	2
+	((+ / (+	A B C * D / + E – F G / H	3
I	((+ / (+ I	A B C * D / + E – F G / H	3
)	((+ /	A B C * D / + E – F G / H I +	3
)	(	A B C * D / + E – F G / H I + / +	1
)		A B C * D / + E – F G / H I + / +	1

Postfix expression is: **A B C * D / + E – F G / H I + / +**

(ii) (A + B) * C + D / (B + A * C) + D

Input Symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
A	((A		0
+	((+	A	1
B	((+ B	A	1
)	(	A B +	1
*	(*	A B +	1
C	(* C	A B +	1
+	(+	A B + C *	1
D	(+ D	A B + C *	1
/	(+ /	A B + C * D	2
(	(+ / (	A B + C * D	2
B	(+ / (B	A B + C * D	2
+	(+ / (+	A B + C * D B	3
A	(+ / (+ A	A B + C * D B	3
*	(+ / (+ *	A B + C * D B A	4
C	(+ / (+ * C	A B + C * D B A	4
)	(+ /	A B + C * D B A C * +	3
+	(+	A B + C * D B A C * + / +	1
D	(+ D	A B + C * D B A C * + / +	1
)		A B + C * D B A C * + / + D +	1

Postfix expression is: **A B + C * D B A C * + / + D +**

Convert the following string into prefix: A-B/(C*D^E)

Step-1 : reverse infix expression

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

Step-2 : convert '(' to ')' and ')' to '(' and append extra ')' at last

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

Step-3 : Now convert this string to postfix

Input Symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
E	((E		0
^	((^	E	1
(	((^ (	E	1
D	((^ (D	E	1
*	((^ (*	E D	2
C	((^ (* C	E D	2
)	((^	E D C *	2
)	(	E D C * ^	1
/	(/	E D C * ^	1
B	(/ B	E D C * ^	1
-	(-	E D C * ^ B /	1
A	(- A	E D C * ^ B /	1
)		E D C * ^ B / A -	1

Step 4 : Reverse this postfix expression

- A / B ^ * C D E

Translate the following string into Polish notation and trace the content of stack: (a + b ^ c ^ d) * (e + f / d)

Step-1 : reverse infix expression

) d / f + e (*) d ^ c ^ b + a (

Step-2 : convert '(' to ')' and ')' to '(' and append extra ')' at last

(d / f + e) * (d ^ c ^ b + a))

Step-3 : Now convert this string to postfix

Input symbol	Content of stack	Reverse polish	Rank
	(		0
(	((		0
d	((d		0
/	((/	d	1
f	((/ f	d	1
+	((+	d f /	1
e	((+ e	d f /	1
)	(	d f / e +	1
*	(+	d f / e +	1
(	(* (	d f / e +	1
d	(* (d	d f / e +	1
^	(* (^	d f / e + d	2
c	(* (^ c	d f / e + d	2
^	(* (^ ^	d f / e + d c	3
b	(* (^ ^ b	d f / e + d c	3
+	(* (+	d f / e + d c b ^ ^	2
a	(* (+ a	d f / e + d c b ^ ^	2
)	(*	d f / e + d c b ^ ^ a +	2
)		d f / e + d c b ^ ^ a + *	1

Step 4 : Reverse this postfix expression

*** + a ^ ^ b c d + e / f d**

Write an algorithm for evaluation of postfix expression and evaluation the following expression showing every status of stack in tabular form.

(i) 5 4 6 + * 4 9 3 / + * (ii) 7 5 2 + * 4 1 1 + / -

Algorithm: EVALUAE_POSTFIX

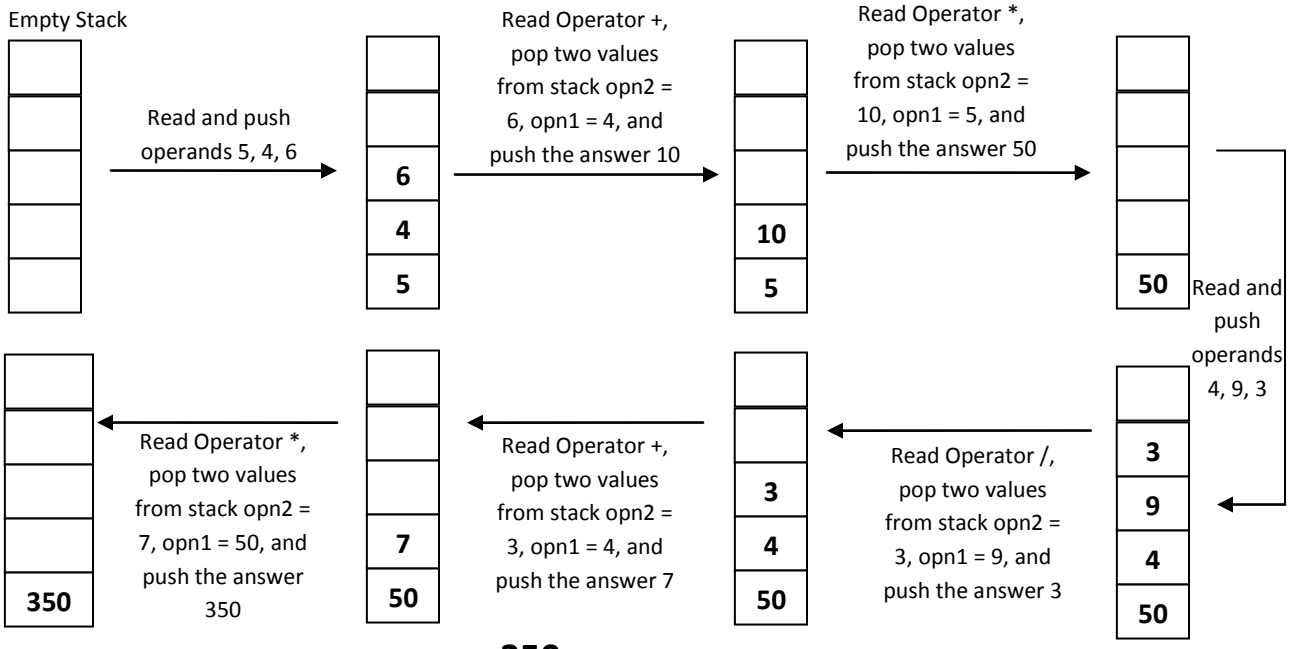
- Given an input string POSTFIX representing postfix expression. This algorithm is going to evaluate postfix expression and put the result into variable VALUE. A vector S is used as a stack PUSH and POP are the function used for manipulation of stack. Operand2 and operand1 are temporary variable TEMP is used for temporary variable NEXTCHAR is a function which when invoked returns the next character. PERFORM_OPERATION is a function which performs required operation on OPERAND1 AND OPERAND2.

1. [Initialize stack and value]
TOP ← 1
VALUE ← 0

2. [Evaluate the prefix expression]
Repeat until last character
TEMP ← NEXTCHAR (POSTFIX)
If TEMP is DIGIT
Then PUSH (S, TOP, TEMP)
Else OPERAND2 ← POP (S, TOP)
OPERAND1 ← POP (S, TOP)
VALUE ← PERFORM_OPERATION(OPERAND1, OPERAND2, TEMP)
PUSH (S, POP, VALUE)

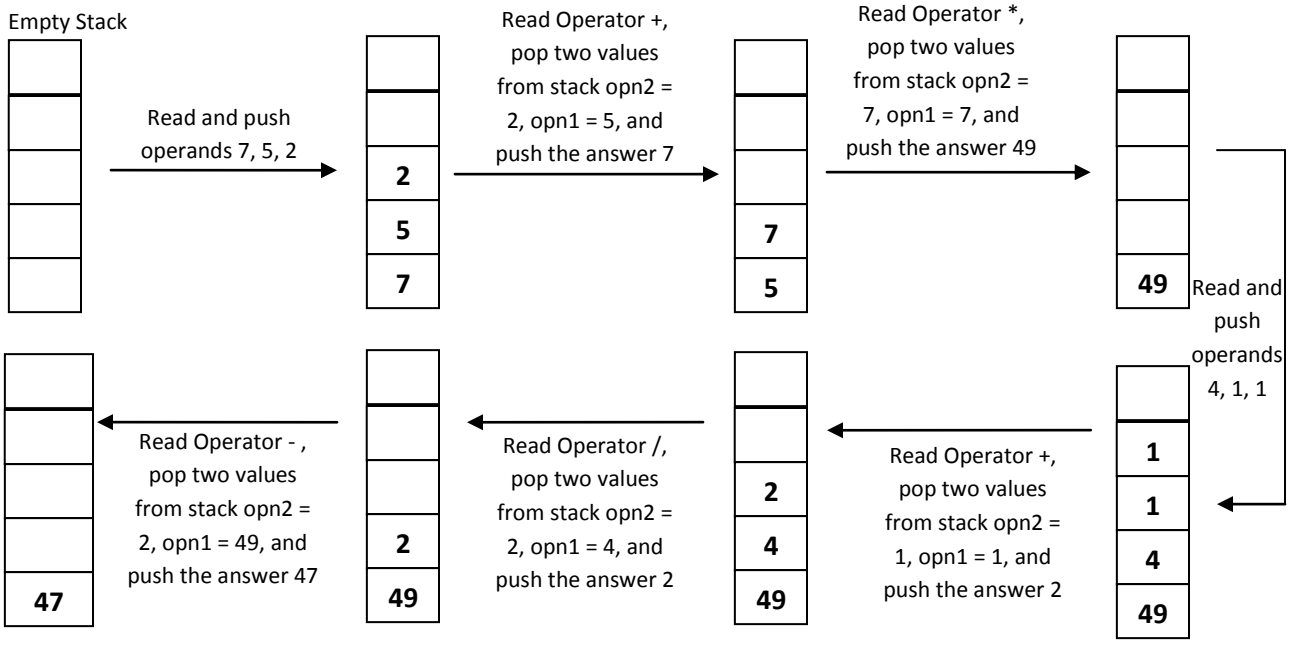
3. [Return answer from stack]
Return (POP (S, TOP))

Evaluate (i): 5 4 6 + * 4 9 3 / + *



Poped value **350** is the answer

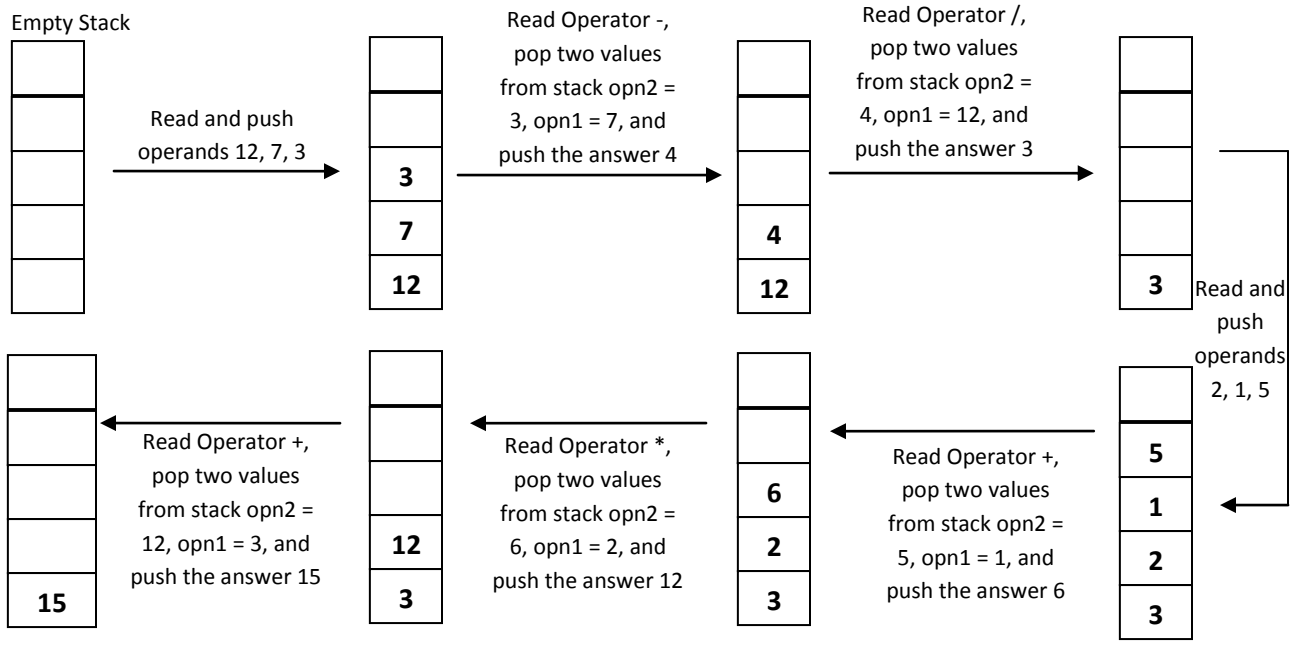
Evaluate (ii) : * 7 5 2 + * 4 1 1 + / -



Poped value **47** is the answer

Consider the following arithmetic expression P, written in postfix notation. Translate it in infix notation and evaluate. P: 12, 7, 3, -, /, 2, 1, 5, +, *, +

Same Expression in infix notation is : **(12 / (7 – 3)) + ((5 + 1) * 2)**



Poped value **15** is the answer

Explain Difference between Stack and Queue.

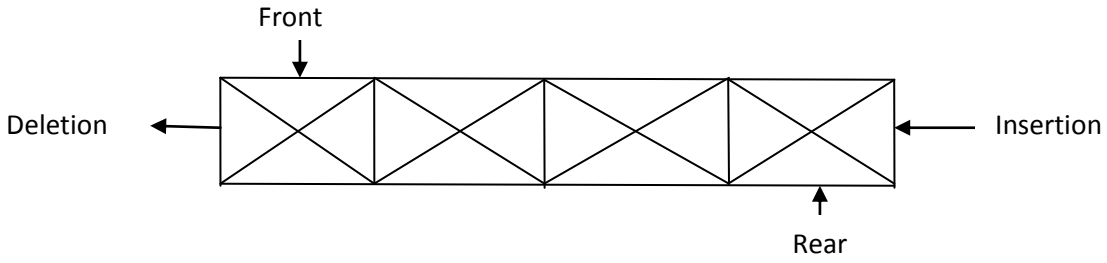
Stack	**Queue**
A Linear List Which allows insertion or deletion of an element at one end only is called as Stack	A Linear List Which allows insertion at one end and deletion at another end is called as Queue
Since insertion and deletion of an element are performed at one end of the stack, the elements can only be removed in the opposite order of insertion.	Since insertion and deletion of an element are performed at opposite end of the queue, the elements can only be removed in the same order of insertion.
Stack is called as Last In First Out (LIFO) List.	Queue is called as First In First Out (FIFO) List.
The most and least accessible elements are called as TOP and BOTTOM of the stack	Insertion of element is performed at FRONT end and deletion is performed from REAR end
Example of stack is arranging plates in one above one.	Example is ordinary queue in provisional store.
Insertion operation is referred as PUSH and deletion operation is referred as POP	Insertion operation is referred as ENQUEUE and deletion operation is referred as DQUEUE
Function calling in any languages uses Stack	Task Scheduling by Operating System uses queue

Explain following:

(i) Queue (ii) Circular Queue (iii) DQUEUE (iv) Priority Queue

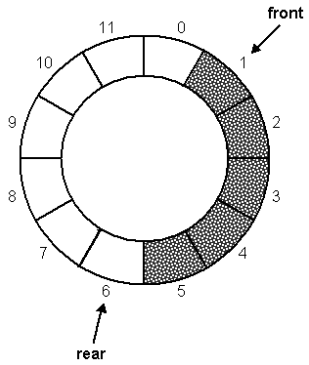
(i) Queue

- A linear list which permits deletion to be performed at one end of the list and insertion at the other end is called queue.
- The information in such a list is processed FIFO (first in first out) of FCFS (first come first served) pattern.
- Front is the end of queue from that deletion is to be performed.
- Rear is the end of queue at which new element is to be inserted.
- The process to add an element into queue is called **Enqueue**
- The process of removal of an element from queue is called **Dequeue**.
- The familiar and traditional example of a queue is Checkout line at Supermarket Cash Register where the first person in line is usually the first to be checkedout.



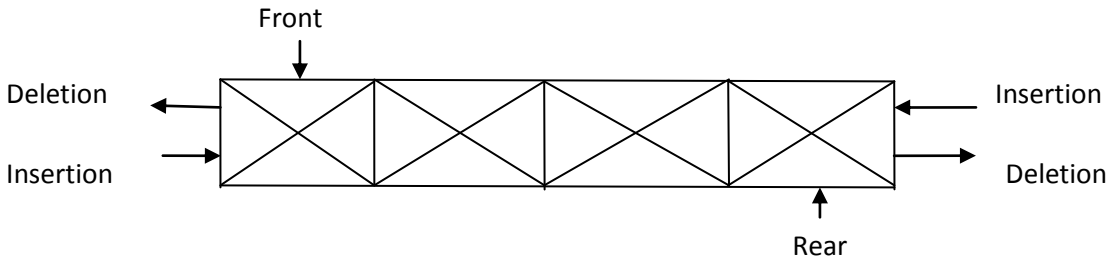
(ii) Circular Queue

- A more suitable method of representing simple queue which prevents 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 is called circular queue
- In a standard queue data structure re-buffering problem occurs for each **dequeue** operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue
- Circular queue is a linear data structure. It follows FIFO principle.
- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as “Ring buffer”.



(iii) Dequeue

- A dequeue (double ended queue) is a linear list in which insertion and deletion are performed from the either end of the structure.
- There are two variations of Dqueue
  - Input restricted dqueue- allows insertion at only one end
  - Output restricted dqueue- allows deletion from only one end
- Such a structure can be represented by following fig.



(iv) Priority Queue

- A queue in which we are able to insert remove items from any position based on some property (such as priority of the task to be processed) is often referred as priority queue.
- Below fig. represent a priority queue of jobs waiting to use a computer.
- Priorities of 1, 2, 3 have been attached with jobs of real time, online and batch respectively. Therefore if a job is initiated with priority i,it is inserted immediately at the end of list of other jobs with priorities i. Here jobs are always removed from the front of queue

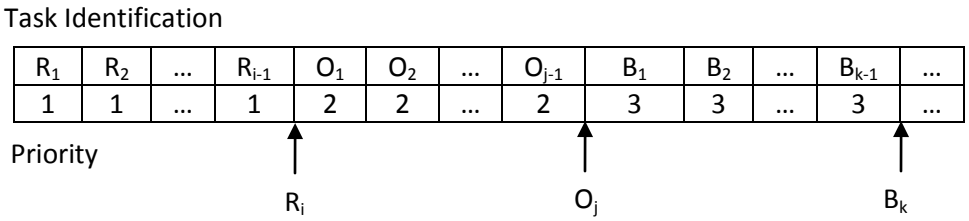


Fig (a) : Priority Queue viewed as a single queue with insertion allowed at any position.

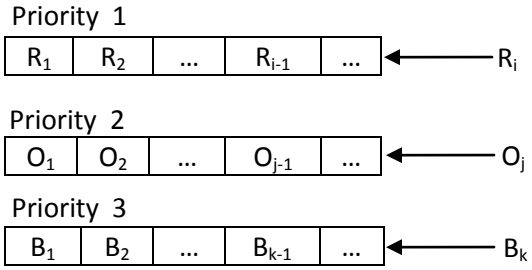


Fig (b) : Priority Queue viewed as a Viewed as a set of queue

Write algorithms of basic primitive operations for Queue

Procedure: QINSERT_REAR (Q, F, R, N,Y)

- Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This procedure inserts Y at rear end of Queue.

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 properly set]
   IF F=0
   Then F ← 1
   Return

Function: QDELETE_FRONT (Q, F, R)

- Given F and R pointers to the front and rear elements of a queue respectively. Queue Q consisting of N elements. This function deleted and element from front end of the Queue.

1. [Underflow]
   IF F= 0
   Then write ('UNDERFLOW')
   Return(0) (0 denotes an empty Queue)
2. [Decrement 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)

Write algorithms of basic primitive operations for Circular Queue

Procedure: CQINSERT (F, R, Q, N, Y)

- Given F and R pointers to the front and rear elements of a circular queue respectively. Circular queue Q consisting of N elements. This procedure inserts Y at rear end of Circular queue.

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

Function CQDELETE (F, R, Q, N)

- Given F and R pointers to the front and rear elements of a Circular queue respectively. Circular Queue Q consisting of N elements. This function deleted and element from front end of the Circular Queue. Y is temporary pointer variable.

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)

Write algorithms of basic primitive operations for DQueue

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

- Given F and R pointers to the front and rear elements of a queue, a queue consisting of N elements and an element Y, this procedure inserts Y at the front of the queue.

1. [Overflow]
   IF F = 0
   Then write ('EMPTY')
   Return
   IF F=1
   Then write ('OVERFLOW')
   Return
2. [Decrement front pointer]
   F ← F-1
3. [Insert element]
   Q[F] ← Y
   Return

Procedure DQDELETE_REAR (Q, F, R)

- Given F and R pointers to the front and rear elements of a queue. And a queue Q to which they correspond, this function deletes and returns the last element from the front end of a queue. And Y is temporary variable.

1. [Underflow]
IF R= 0
Then write ('UNDERFLOW')
Return(0)

2. [Delete element]
Y ← Q[R]

3. [Queue empty?]
IF R=F
Then R← F← 0
Else R← R-1 (decrement front pointer)

4. [Return element]
Return (Y)

PROCEDURE DQUEUE_DISPLAY (F,R,Q)

- Given F and Rare pointers to the front and rear elements of a queue, a queue consist of N elements. This procedure display Queue contents

```
1. [Check for empty]
   IF      F >= R
    Then   write ('QUEUE IS EMPTY')
           Return
2. [Display content]
   FOR     (I=FRONT; I<=REAER; I++)
           Write (Q[I])
3. [Return Statement]
   Return
```

Consider the following queue, where queue is a circular queue having 6 memory cells. Front=2, Rear=4

Queue: _, A, C, D, _, _

Describe queue as following operation take place:

F is added to the queue

Two letters are deleted

R is added to the queue

S is added to the queue

One letter is deleted

Positions	**1**	**2**	**3**	**4**	**5**	**6**
Initial Position of Queue, **Front=2, Rear=4**		A	C	D		
F is added to queue, **Front=2, Rear=5**		A	C	D	F	
Two letters are deleted, **Front=4, Rear=5**				D	F	
R is added to the queue, **Front=4, Rear=6**				D	F	R
S is added to the queue, **Front=4, Rear=1**	S			D	F	R
One letter is deleted, **Front=5, Rear=1**	S				F	R

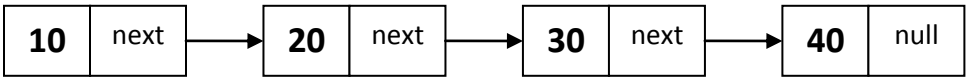
1. Linear Data Structure and their linked storage representation.

There are many applications where sequential allocation method is unacceptable because of following characteristics

- Unpredictable storage requirement
- Extensive manipulation of stored data

The linked allocation method of storage can result in both efficient use of computer storage and computer time.

- A linked list is a non-sequential collection of data items.
- The concept of a linked list is very simple, for every data item in the linked list, there is an associated pointer that would give the memory allocation of the next data item in the linked list.
- The data items in the linked list are not in a consecutive memory locations but they may be anywhere in memory.
- Accessing of these data items is easier as each data item contains within itself the address of the next data item.

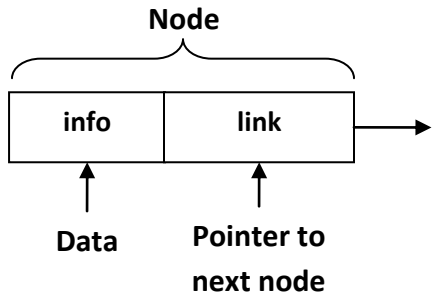


A Linked List

---

2. What is linked list? What are different types of linked list? OR Write a short note on singly, circular and doubly linked list. OR Advantages and disadvantages of singly, circular and doubly linked list.

- A linked list is a collection of objects stored in a list form.
- A linked list is a sequence of items (objects) where every item is linked to the next.
- A linked list is a non-primitive type of data structure in which each element is dynamically allocated and in which elements point to each other to define a linear relationship.
- Elements of linked list are called nodes where each node contains two things, data and pointer to next node.
- Linked list require more memory compared to array because along with value it stores pointer to next node.
- Linked lists are among the simplest and most common data structures. They can be used to implement other data structures like stacks, queues, and symbolic expressions, etc…




```
// C Structure to represent a node
struct node
{
      int info
      struct node *link
};
```

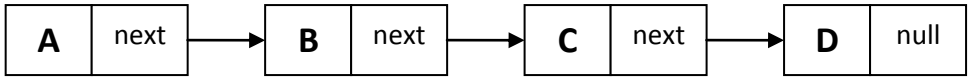
Operations on linked list

- Insert
  - Insert at first position
  - Insert at last position
  - Insert into ordered list
- Delete
- Traverse list (Print list)
- Copy linked list

Types of linked list

Singly Linked List

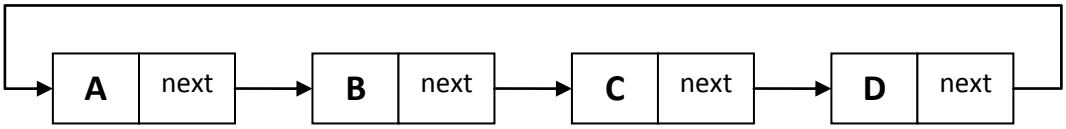
- It is basic type of linked list.
- Each node contains data and pointer to next node.
- Last node's pointer is null.
- Limitation of singly linked list is we can traverse only in one direction, forward direction.



Singly Linked List

Circular Linked List

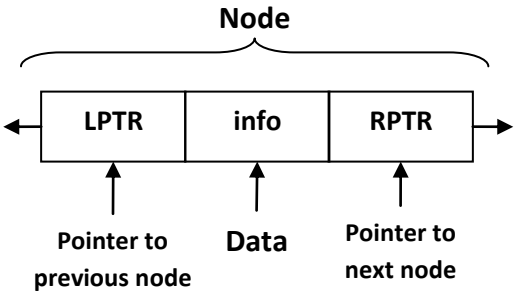
- Circular linked list is a singly linked list where last node points to first node in the list.
- It does not contain null pointers like singly linked list.
- We can traverse only in one direction that is forward direction.
- It has the biggest advantage of time saving when we want to go from last node to first node, it directly points to first node.
- A good example of an application where circular linked list should be used is a timesharing problem solved by the operating system.



Circular Linked List

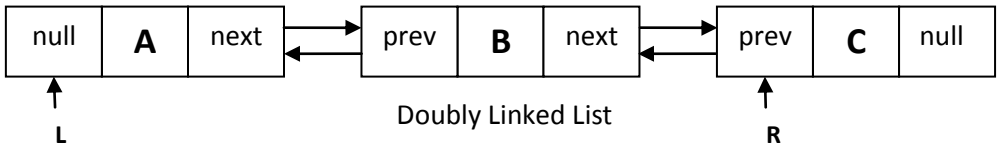
Doubly Linked list

- Each node of doubly linked list contains data and two pointers to point previous (LPTR) and next (RPTR) node.




```
// C Structure to represent a node
struct node
{
      int info
      struct node *lptr;
      struct node *rptr;
};
```

- Main advantage of doubly linked list is we can traverse in any direction, forward or reverse.
- Other advantage of doubly linked list is we can delete a node with little trouble, since we have pointers to the previous and next nodes. A node on a singly linked list cannot be removed unless we have the pointer to its predecessor.
- Drawback of doubly linked list is it requires more memory compared to singly linked list because we need an extra pointer to point previous node.
- L and R in image denote left most and right most nodes in the list.
- Left link of L node and right link of R node is NULL, indicating the end of list for each direction.



Doubly Linked List

3. Discuss advantages and disadvantages of linked list over array.

Advantages of an array

1. We can access any element of an array directly means random access is easy
2. It can be used to create other useful data structures (queues, stacks)

3. It is light on memory usage compared to other structures

Disadvantages of an array

1. Its size is fixed
2. It cannot be dynamically resized in most languages
3. It is hard to add/remove elements
4. Size of all elements must be same.
5. Rigid structure (Rigid = Inflexible or not changeable)

Advantages of Linked List

1. **Linked lists are dynamic data structures:** That is, they can grow or shrink during execution of a program.
2. **Efficient memory utilization:** Here memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (free) when it is no longer needed.
3. **Insertion and deletions are easier and efficient:** Linked list provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. **Elements of linked list are flexible:** It can be primary data type or user defined data types

Disadvantages of Linked List

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2. It cannot be easily sorted
3. We must traverse 1/2 the list on average to access any element
4. More complex to create than an array
5. Extra memory space for a pointer is required with each element of the list

---

3. What are the advantages and disadvantages of stack and queue implemented using linked list over array?

Advantages and disadvantages of stack & queue implemented using linked list over array is described below,

Insertion & Deletion Operation

- Insertion and deletion operations are known as push and pop operation in stack and as insert and delete operation in queue.
- In the case of an array, if we have n-elements list and it is required to insert a new element between the first and second element then n-1 elements of the list must be moved so as to make room for the new element.
- In case of linked-list, this can be accomplished by only interchanging pointers.
- Thus, insertion and deletions are more efficient when performed in linked list then array.

Searching a node

- If a particular node in a linked list is required, it is necessary to follow links from the first node onwards until the desired node is found.
- Where as in the case of an array, directly we can access any node

Join & Split

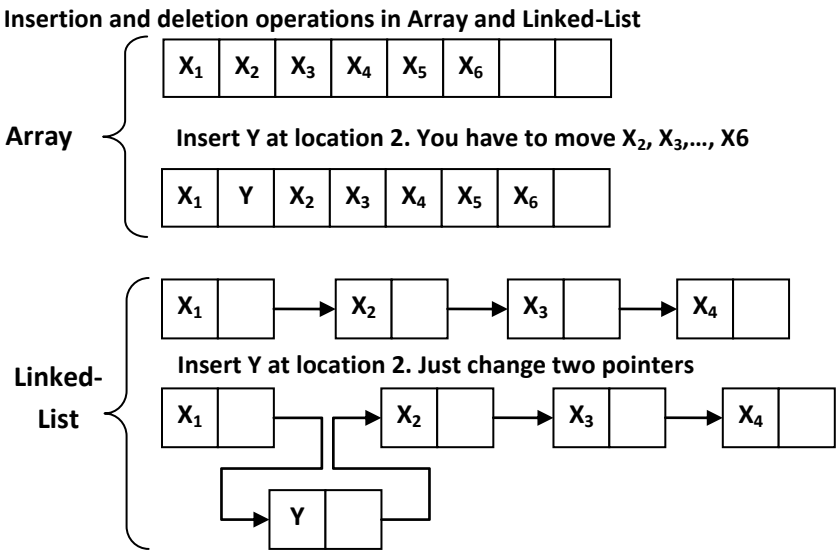
- We can join two linked list by assigning pointer of second linked list in the last node of first linked list.
- Just assign null address in the node from where we want to split one linked list in two parts.
- Joining and splitting of two arrays is much more difficult compared to linked list.

Memory

- The pointers in linked list consume additional memory compared to an array

Size

- Array is fixed sized so number of elements will be limited in stack and queue.
- Size of linked list is dynamic and can be changed easily so it is flexible in number of elements

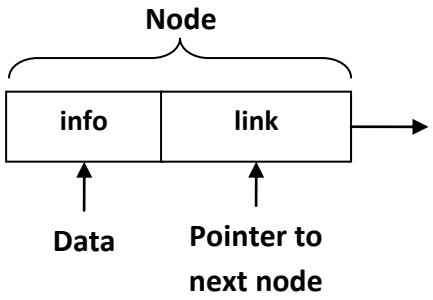


4. Write following algorithms for singly linked list.

1) Insert at first position
2) Insert at last position
3) Insert in Ordered Linked list
4) Delete Element
5) Copy Linked List

Few assumptions,

- We assume that a typical element or node consists of two fields namely; an information field called INFO and pointer field denoted by LINK. The name of a typical element is denoted by NODE.




```
// C Structure to represent a node
struct node
{
      int info
      struct node *link
};
```

Function: INSERT(X, First)

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. This function inserts a new node at the first position of linked list. This function returns address of FIRST 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 new node]
Return (NEW)

When INSERT is invoked it returns a pointer value to the variable FIRST

FIRST ← INSERT (X, FIRST)

Function: INSEND(X, First) (Insert at end)

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW is a temporary pointer variable. This function inserts a new node at the last position of linked list. This function returns address of FIRST 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 field of NEW node]
INFO (NEW) ← X
LINK (NEW) ← NULL

5 [Is the list empty?]
If FIRST = NULL
then Return (NEW)

6 [Initialize search for a last node]
SAVE ← FIRST

7 [Search for end of list]
Repeat while LINK (SAVE) ≠ NULL
SAVE ← LINK (SAVE)

8 [Set link field of last node to NEW)
LINK (SAVE) ← NEW

9 [Return first node pointer]
Return (FIRST)

When INSERTEND is invoked it returns a pointer value to the variable FIRST

FIRST ← INSERTEND (X, FIRST)

Insert a node into Ordered Linked List

- There are many applications where it is desirable to maintain an ordered linear list. The ordering is in increasing or decreasing order on INFO field. Such ordering results in more efficient processing.
- The general algorithm for inserting a node into an ordered linear list is as below.
  1. Remove a node from availability stack.
  2. Set the field of new node.
  3. If the linked list is empty then return the address of new node.
  4. If node precedes all other nodes in the list then inserts a node at the front of the list and returns its address.
  5. Repeat step 6 while information contain of the node in the list is less than the information content of the new node.
  6. Obtain the next node in the linked list.
  7. Insert the new node in the list and return address of its first node.

Function: INSORD(X, FIRST)

Given X, a new element and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. AVAIL is a pointer to the top element of the availability stack; NEW & SAVE are temporary pointer variables. This function inserts a new node such that linked list preserves the ordering of the terms in increasing order of their INFO field. This function returns address of FIRST 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. [Is the list is empty]
If FIRST = NULL
then LINK (NEW) ← NULL
Return (NEW)

5. [Does the new node precede all other node in the list?]
If INFO(NEW) ≤ INFO (FIRST)
then LINK (NEW) ← FIRST
Return (NEW)

6. [Initialize temporary pointer]
SAVE ← FIRST

7. [Search for predecessor of new node]
Repeat while LINK (SAVE) ≠ NULL and INFO (NEW) ≥ INFO (LINK (SAVE))
SAVE ← LINK (SAVE)

8. [Set link field of NEW node and its predecessor]
LINK (NEW) ← LINK (SAVE)
LINK (SAVE) ← NEW

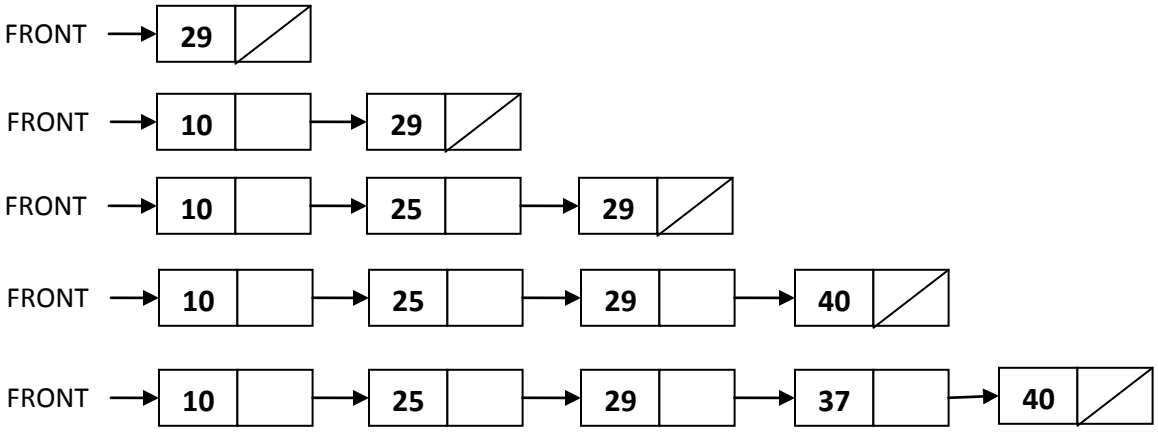
9. [Return first node pointer]
Return (FIRST)

When INSERTORD is invoked it returns a pointer value to the variable FIRST

FIRST ← INSERTORD (X, FIRST)

By repeatedly involving function INSORD; we can easily obtains an ordered liner list for example the sequence of statements.

FRONT ← NULL
FRONT ← INSORD (29, FRONT)
FRONT ← INSORD (10, FRONT)
FRONT ← INSORD (25, FRONT)
FRONT ← INSORD (40, FRONT)
FRONT ← INSORD (37, FRONT)



Trace of construction of an ordered linked linear list using function INSORD

Algorithm to delete a node from Linked List

- Algorithm that deletes node from a linked linear list:-
  1. If a linked list is empty, then write under flow and return.
  2. Repeat step 3 while end of the list has not been reached and the node has not been found.
  3. Obtain the next node in list and record its predecessor node.
  4. If the end of the list has been reached then write node not found and return.
  5. Delete the node from list.
  6. Return the node into availability area.

Procedure: DELETE(X, FIRST)

Given X, an address of node which we want to delete and FIRST is a pointer to the first element of a linked linear list. Typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables.

1. **[Is Empty list?]**
   If FIRST = NULL
   then write ('Underflow')
   return
2. **[Initialize search for X]**
   SAVE ← FIRST
3. **[Find X]**
   Repeat thru step-5 while SAVE ≠ X and LINK (SAVE) ≠ NULL
4. **[Update predecessor marker]**
   PRED ← SAVE
5. **[Move to next node]**
   SAVE ← LINK (SAVE)
6. **[End of the list]**
   If SAVE ≠ X
   then write ('Node not found')
   return
7. **[Delete X]**
   If X = FIRST (if X is first node?)
   then FIRST ← LINK (FIRST)
   else LINK (PRED) ← LINK (X)
8. **[Free Deleted Node]**
   Free (X)

Function: COPY (FIRST)

- FIRST is a pointer to the first node in the linked list, this function makes a copy of the list.
- The new list is to contain nodes whose information and pointer fields are denoted by FIELD and PTR, respectively. The address of the first node in the newly created list is to be placed in BEGIN. NEW, SAVE and PRED are points variables.
- A general algorithm to copy a linked list
  1. If the list is empty then return null

2. If the availability stack is empty then write availability stack underflow and return else copy the first node.
3. Report thru step 5 while the old list has not been reached.
4. Obtain next node in old list and record its predecessor node.
5. If availability stack is empty then write availability stack underflow and return else copy the node and add it to the rear of new list.
6. Set link of the last node in the new list to null and return.

1. **[Is Empty List?]**
   If FIRST = NULL
   then return (NULL)

2. **[Copy first node]**
   NEW ⇦ NODE
   New ← AVAIL
   AVAIL ← LINK (AVAIL)
   FIELD (NEW) ← INFO (FIRST)
   BEGIN ← NEW

3. **[Initialize traversal]**
   SAVE ← FIRST

4. **[Move the next node if not at the end if list]**
   Repeat thru step 6 while (SAVE) ≠ NULL

5. **[Update predecessor and save pointer]**
   PRED ← NEW
   SAVE ← LINK (SAVE)

6. **[Copy node]**
   If AVAIL = NULL
   then write ('Availability stack underflow')
   Return (0)
   else NEW ← AVAIL
   AVAIL ← LINK (AVAIL)
   FIELD (NEW) ← INFO (SAVE)
   PTR (PRED) ← NEW

7. **[Set link of last node and return]**
   PTR (NEW) ← NULL
   Return (BEGIN)

5. Write following algorithms for circular link list

1) Insert at First Position
2) Insert at Last Position
3) Insert in Ordered Linked List
4) Delete Element

PROCEDURE: CIRCULAR_LINK_INSERT_FIRST (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. This procedure inserts value X at the first position of Circular linked linear list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   If FIRST = NULL
   then LINK (NEW) ← NEW
   FIRST ← LAST ← NEW
   else LINK (NEW) ← FIRST
   LINK (LAST) ← NEW
   FIRST ← NEW
   Return

PROCEDURE: CIR_LINK_INSERT_END (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. This procedure inserts value X at the last position of Circular linked linear list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   If FIRST = NULL
   then LINK (NEW) ← NEW
   FIRST ← LAST ← NEW
   else LINK(NEW) ← FIRST
   LINK(LAST) ← NEW
   LAST ← NEW
   Return

PROCEDURE: CIR_LINK_INSERT_ORDER (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. This procedure inserts value X such that linked list preserves the ordering of the terms in increasing order of their INFO field.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Copy information content into new node]**
   INFO (NEW) ← X
3. **[Is Linked List is empty?]**
   If FIRST = NULL
   then LINK (NEW) ← NEW
   FIRST ← LAST ← NEW
   Return
4. **[Does new node precedes all other nodes in List?]**
   If INFO (NEW) ≤ INFO (FIRST)
   then LINK (NEW) ← FIRST
   LINK (LAST) ← NEW
   FIRST ← NEW
   Return
5. **[Initialize Temporary Pointer]**
   SAVE ← FIRST
6. **[Search for Predecessor of new node]**
   Repeat while SAVE ≠ LAST and INFO(NEW) ≥ INFO(LINK(SAVE))
   SAVE ← LINK(SAVE)
7. **[Set link field of NEW node and its Predecessor]**
   LINK(NEW) ← LINK(SAVE)
   LINK(SAVE) ← NEW
   If SAVE = LAST
   then LAST ← NEW
8. **[Finish]**
   Return

PROCEDURE: CIR_LINK_DELETE (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables. This procedure deletes a node whose address is given by pointer variable X.

1. **[Is Empty List?]**
   If FIRST = NULL
   then write ('Linked List is Empty')
   Return
2. **[Initialize Search for X]**
   SAVE ← FIRST
3. **[Find X]**
   Repeat thru step 5 while SAVE ≠ X and SAVE ≠ LAST
4. **[Update predecessor marker]**
   PRED ← SAVE
5. **[Move to next node]**
   SAVE ← LINK (SAVE)
6. **[End of Linked List]**
   If SAVE ≠ X
   then write('Node not found')
   return
7. **[Delete X]**
   If X = FIRST
   then FIRST ← LINK (FIRST)
   LINK (LAST) ← FIRST
   else LINK (PRED) ← LINK(X)
   If X = LAST
   then LAST ← PRED
8. **[Free Deleted Node]**
   Free (X)

6. Write an algorithm to perform each of the following operations on Circular singly linked list using header node

1) add node at beginning
2) add node at the end
3) insert a node containing x after node having address P
4) delete a node which contain element x

FUNCTION: CIR_LINK_HEAD_INSERT_FIRST (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. HEAD is the address of HEAD node. This procedure inserts value X at the first position of Circular linked linear list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   LINK (NEW) ← LINK (HEAD)
   LINK (HEAD) ← NEW

FUNCTION: CIR_LINK_HEAD_INSERT_LAST (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary points variable. HEAD is the address of HEAD node. This procedure inserts value X at the last position of Circular linked linear list.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   LINK (NEW) ← HEAD
   LINK (LAST) ← NEW
   LAST ← NEW

FUNCTION: CIR_LINK_HEAD_INSERT_AFTER_Node-P (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. NEW is a temporary pointer variable. HEAD is the address of HEAD node. This procedure insert a node after a node having address P.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Initialize fields of new node and its link to the list]**
   INFO (NEW) ← X
   LINK (NEW) ← LINK (P)
   LINK (P) ← NEW
   If P = LAST
   then LAST ← NEW

PROCEDURE: CIR_LINK_HEAD_DELETE (X, FIRST, LAST)

FIRST and LAST are pointers to the first and last element of a circular linked linear list respectively whose typical node contains INFO and LINK fields. SAVE & PRED are temporary pointer variables. HEAD is the address of HEAD node. This procedure deletes a node whose value is X.

1. **[Is Empty List?]**
   If FIRST = NULL
   then write ('Underflow)
   return
2. **[Initialize Search for X]**
   SAVE ← FIRST
3. **[Find X]**
   Repeat thru step 5 while INFO(SAVE) ≠ X and SAVE ≠ LAST
4. **[Update Predecessor]**
   PRED ← SAVE
5. **[Move to next node]**
   SAVE ← LINK(SAVE)
6. **[End of the List]**
   If INFO (SAVE) ≠ X
   then write('Node not Found')
   return
7. **[Delete node X]**
   If INFO (FIRST) = X
   then LINK (HEAD) ← LINK(FIRST)
   else LINK (PRED) ← LINK(SAVE)
   If SAVE = LAST
   then LAST ← PRED
8. **[Free Deleted Node]**
   Free (X)

7. Write following algorithms for doubly link list

1) Insert
2) Insert in Ordered Linked List
3) Delete Element

PRDCEDURE: DOUBINS (L, R, M, X)

Given a doubly link list whose left most and right most nodes addressed are given by the pointer variables L and R respectively. It is required to insert a node whose address is given by the pointer variable NEW. The left and right links of nodes are denoted by LPTR and RPTR respectively. The information field of a node is denoted by variable INFO. The name of an element of the list is NODE. The insertion is to be performed to the left of a specific node with its address given by the pointer variable M. The information to be entered in the node is contained in X.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Copy information field]**
   INFO (NEW) ← X
3. **[Insert into an empty list]**
   If R = NULL
   then LPTR (NEW) ← RPTR (NULL) ← NULL
   L ← R ← NEW
   Return
4. **[Is 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 (NEW)) ← NEW
   Return

PROCEDURE: DOUBINS_ORD (L, R, M, X)

Given a doubly link list whose left most and right most nodes addressed are given by the pointer variables L and R respectively. It is required to insert a node whose address is given by the pointer variable NEW. The left and right links of nodes are denoted by LPTR and RPTR respectively. The information field of a node is denoted by variable INFO. The name of an element of the list is NODE. The insertion is to be performed in ascending order of info part. The information to be entered in the node is contained in X.

1. **[Create New Empty Node]**
   NEW ⇦ NODE
2. **[Copy information field]**
   INFO (NEW) ← X
3. **[Insert into an empty list**]
   If R = NULL
   then LPTR (NEW) ← RPTR (NULL) ← NULL
   L ← R ← NEW
   return
4. **[Does the new node precedes all other nodes in List?]**
   If INFO(NEW) ≤ INFO(L)
   then RPTR (NEW) ← L
   LPTR(NEW)← NULL
   LPTR (L) ← NEW
   L ← NEW
   Return
5. **[Initialize temporary Pointer]**
   SAVE ← L
6. **[Search for predecessor of New node]**
   Repeat while RPTR(SAVE) ≠ NULL and INFO(NEW) ≥ INFO(RPTR(SAVE))
   SAVE ← RPTR (SAVE)
7. **[Set link field of new node and its predecessor]**
   RPTR (NEW) ← RPTR(SAVE)
   LPTR (RPTR(SAVE)) ← NEW
   RPTR (SAVE) ← NEW
   LPTR (NEW) ← SAVE

   If SAVE = R
   then RPTR(SAVE) ← NEW

PROCEDURE: DOUBDEL (L, R, OLD)

Given a doubly linked list with the addresses of left most and right most nodes are given by the pointer variables L and R respectively. It is required to delete the node whose address is contained in the variable OLD. Node contains left and right links with names LPTR and RPTR respectively.

```
1. [ Is underflow ?]
   If      R=NULL
   then    write (' UNDERFLOW')
           return

2. [Delete node]
   If      L = R (single node in list)
   then    L ← R ← NULL
   else    If      OLD = L (left most node)
           then    L ← RPTR(L)
                   LPTR (L) ← NULL
           else    if      OLD = R (right most)
                   then    R ← LPTR (R)
                           RPTR (R) ← NULL
                   else    RPTR  (LPTR (OLD)) ← RPTR (OLD)
                           LPTR  (RPTR (OLD)) ← LPTR (OLD)

3. [ FREE deleted node]
   FREE (OLD)
```

8. Write the implementation procedure of basic primitive operations of the stack using: (i) Linear array (ii) linked list.

Implement PUSH and POP using Linear array

```
#define MAXSIZE 100
int stack[MAXSIZE];
int top=-1;

void push(int val)
{
      if(top >= MAXSIZE)
          printf("Stack is Overflow");
      else
          stack[++top] = val;
}

int pop()
{
     int a;
     if(top>=0)
     {
          a=stack[top];
          top--;
          return a;
     }
     else
     {
          printf("Stack is Underflow, Stack is empty, nothing to POP!");
          return -1;
     }
}
```

Implement PUSH and POP using Linked List

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

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

void push(int val)
{
      struct node *p;
      p = (struct node*)malloc(sizeof(struct node));
      p → info = val;
      p → link = top;
      top = p;
      return;
}

int pop()
{
      int val;
      if(top!=NULL)
      {
            val = top → info;
            top=top →link;
            return val;
      }
      else
      {
            printf("Stack Underflow");
            return -1;
      }
}
```

---

9. Write the implementation procedure of basic primitive operations of the Queue using: (i) Linear array (ii) linked list

Implement Enqueue(Insert)and Dequeue(Delete)using Linear Array

```
# include <stdio.h>
# define MAXSIZE 100
int queue[MAXSIZE], front = -1, rear = -1;
void enqueue(int val)
{
      if(rear >= MAXSIZE)
      {
            printf("Queue is overflow") ;
            return ;
      }
      rear++;
      queue [rear] = val;
      if(front == -1)
      {
            front++;
      }
}
int dequeue()
{
      int data;
      if(front == -1)
      {
            printf("Queue is underflow") ;
            return -1;
      }
      data = queue [front];
      if(front == rear)
      {
            front = rear = -1;
      }
      else
      {
            front++;
      }
      return data;
}
```

Implement Enqueue(Insert)and Dequeue(Delete)using Linked List

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

struct node
{
    int info;
    struct node *link;
} *front, *rear;


void enqueue(int val)
{
     struct node *p;
     p = (struct node*)malloc(sizeof(struct node));
     p → info = val;
     p → link = NULL;
     if (rear == NULL || front == NULL)
     {
          front = p;
     }
     else
     {
          rear → link = p;
          rear = p;
     }
}

int dequeue()
{
    struct node *p;
    int val;
    if (front == NULL || rear == NULL)
    {
        printf("Under Flow");
        exit(0);
    }
    else
    {
        p = front;
        val = p → info;
        front = front → link;
        free(p);
    }
    return (val);
}
```

10. Write an algorithm to implement ascending priority queue using singular linear linked list which has insert() function such that queue remains ordered list. Also implement remove() function

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


remove()
{
      struct node *tmp;
      if(front == NULL)
            printf("Queue Underflow\n");
      else
      {
            tmp = front;
            printf("Deleted item is %d\n",tmp->info);
            front = front->link;
            free(tmp);
      }
}/*End of remove()*/
```

```
insert()
{
      struct node *tmp,*q;
      int added_item,item_priority;
      tmp = (struct node *)malloc(sizeof(struct node));
      printf("Input the item value to be added in the queue : ");
      scanf("%d",&added_item);
      printf("Enter its priority : ");
      scanf("%d",&item_priority);
      tmp->info = added_item;
      tmp->priority = item_priority;
      /*Queue is empty or item to be added has priority more than
first item*/
      if( front == NULL || item_priority < front->priority )
      {
            tmp->link = front;
            front = tmp;
      }
      else
      {
            q = front;
            while( q->link != NULL &&
                        q->link->priority <= item_priority )
            {
                  q=q->link;
            }
            tmp->link = q->link;
            q->link = tmp;
      }/*End of else*/
}/*End of insert()*/
```

```
display()
{
      struct node *ptr;
      ptr = front;
      if(front == NULL)
            printf("Queue is empty\n");
      else
      {
            printf("Queue is :\n");
            printf("Priority Item\n");
            while(ptr != NULL)
            {
                  printf("%5d %5d\n",ptr->priority,ptr->info);
                  ptr = ptr->link;
            }
      }/*End of else */
}/*End of display() */
```

1. Discuss following

1. Graph

- A graph G consist of a non-empty set V called the set of nodes (points, vertices) of the graph, a set E which is the set of edges and a mapping from the set of edges E to a set of pairs of elements of V.
- It is also convenient to write a graph as G=(V,E).
- Notice that definition of graph implies that to every edge of a graph G, we can associate a pair of nodes of the graph. If an edge X Є E is thus associated with a pair of nodes (u,v) where u, v Є V then we says that edge x connect u and v.

2. Adjacent Nodes

- Any two nodes which are connected by an edge in a graph are called adjacent node.

3. Directed & Undirected Edge

- In a graph G=(V,E) an edge which is directed from one end to another end is called a directed edge, while the edge which has no specific direction is called undirected edge.

4. Directed graph (Digraph)

- A graph in which every edge is directed is called directed graph or digraph.

5. Undirected graph

- A graph in which every edge is undirected is called undirected graph.

6. Mixed Graph

- If some of the edges are directed and some are undirected in graph then the graph is called mixed graph.

7. Loop (Sling)

- An edge of a graph which joins a node to itself is called a loop (sling).

8. Parallel Edges

- In some directed as well as undirected graphs, we may have certain pairs of nodes joined by more than one edges, such edges are called Parallel edges.

9. Multigraph

- Any graph which contains some parallel edges is called multigraph.

10. Weighted Graph

- A graph in which weights are assigned to every edge is called weighted graph.

11. Isolated Node

- In a graph a node which is not adjacent to any other node is called isolated node.

12. Null Graph

- A graph containing only isolated nodes are called null graph. In other words set of edges in null graph is empty.

13. Path of Graph

- Let G=(V, E) be a simple digraph such that the terminal node of any edge in the sequence is the initial node of the edge, if any appearing next in the sequence defined as path of the graph.

14. Length of Path

- The number of edges appearing in the sequence of the path is called length of path.

15. Degree of vertex

- The no of edges which have V as their terminal node is call as indegree of node V
- The no of edges which have V as their initial node is call as outdegree of node V
- Sum of indegree and outdegree of node V is called its Total Degree or Degree of vertex.

16. Simple Path (Edge Simple)

- A path in a diagraph in which the edges are distinct is called simple path or edge simple.

17. Elementary Path (Node Simple)

- A path in which all the nodes through which it traverses are distinct is called elementary path.

18. Cycle (Circuit)

- A path which originates and ends in the same node is called cycle (circuit).

19. Directed Tree

- A directed tree is an acyclic digraph which has one node called its root with in degree 0, while all other nodes have in degree 1.
- Every directed tree must have at least one node.
- An isolated node is also a directed tree.

20. Terminal Node (Leaf Node)

- In a directed tree, any node which has out degree 0 is called terminal node or leaf node.

21. Level of Node

- The level of any node is the length of its path from the root.

22. Ordered Tree

- In a directed tree an ordering of the nodes at each level is prescribed then such a tree is called ordered tree.

23. Forest

- If we delete the root and its edges connecting the nodes at level 1, we obtain a set of disjoint tree. A set of disjoint tree is a forest.

24. M-ary Tree

- If in a directed tree the out degree of every node is less than or equal to m then tree is called an m-ary tree.

25. Full or Complete M-ary Tree

- If the out degree of each and every node is exactly equal to m or 0 and their number of nodes at level **i** is $\mathbf{m^{(i-1)}}$ then the tree is called a full or complete m-ary tree.

26. Positional M-ary Tree

- If we consider m-ary trees in which the m children of any node are assumed to have m distinct positions, if such positions are taken into account, then tree is called positional m-ary tree.

27. Height of the tree

- The height of a tree is the length of the path from the root to the deepest node in the tree.

28. Binary tree

- If in a directed tree the out degree of every node is less than or equal to 2 then tree is called binary tree.

29. Strictly binary tree

- A strictly binary tree (sometimes proper binary tree or 2-tree or full binary tree) is a tree in which every node other than the leaves has two children.

30. Complete binary tree

- If the out degree of each and every node is exactly equal to 2 or 0 and their number of nodes at level **i** is $\mathbf{2^{(i-1)}}$ then the tree is called a full or complete binary tree.

31. Sibling

- Siblings are nodes that share the same parent node.

32. Binary search tree

- A binary search tree is a binary tree in which each node possessed a key that satisfy the following conditions

1. All key (if any) in the left sub tree of the root precedes the key in the root.
2. The key in the root precedes all key (if any) in the right sub tree.
3. The left and right sub tree sub trees of the root are again search trees.

33. Height Balanced Binary tree (AVL Tree)

- A tree is called AVL (height balance binary tree), if each node possesses one of the following properties
    1. A node is called left heavy if the longest path in its left sub tree is one longer then the longest path of its right sub tree.
    2. A node is called right heavy if the longest path in the right sub tree is one longer than path in its left sub tree.
    3. A node is called balanced, if the longest path in both the right and left sub tree are equal.

---

2. Explain the Preorder, Inorder and Postorder traversal techniques of the binary tree with suitable example.

- The most common operations performed on tree structure is that of traversal. This is a procedure by which each node in the tree is processed exactly once in a systematic manner.
- There are three ways of traversing a binary tree.
    1. Preorder Traversal
    2. Inorder Traversal
    3. Postorder Traversal

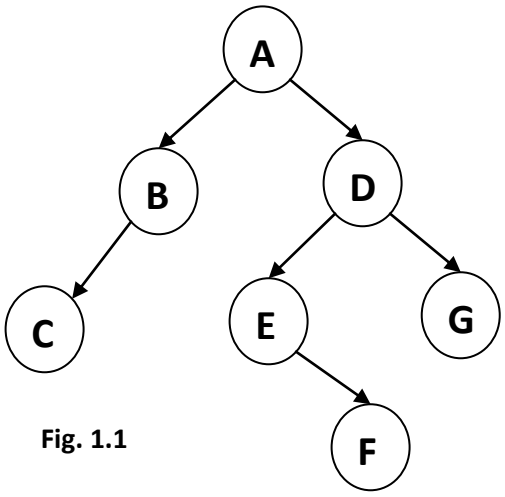


Fig. 1.1

Preorder traversal : A B C D E F G

Inorder traversal : C B A E F D G

Postorder traversal : C B F E G D A

Converse Preorder traversal : A D G E F B C

Converse Inorder traversal : G D F E A B C

Converse Postorder traversal : G F E D C B A

Preorder

- Preorder traversal of a binary tree is defined as follow
  - Process the root node
  - Traverse the left subtree in preorder
  - Traverse the right subtree in preorder
- If particular subtree is empty (i.e., node has no left or right descendant) the traversal is performed by doing nothing, In other words, a null subtree is considered to be fully traversed when it is encountered.
- The preorder traversal of a tree (Fig. 1.1) is given by A B C D E F G

Inorder

- The Inorder traversal of a binary tree is given by following steps,
  - Traverse the left subtree in Inorder
  - Process the root node
  - Traverse the right subtree in Inorder
- The Inorder traversal of a tree (Fig. 1.1) is given by C B A E F D G

Postorder

- The postorder traversal is given by
  - Traverse the left subtree in postorder
  - Traverse the right subtree in postorder
  - Process the root node
- The Postorder traversal of a tree (Fig. 1.1) is given by C B F E G D A

Converse …

- If we interchange left and right words in the preceding definitions, we obtain three new traversal orders which are called
  - Converse Preorder (A D G E F B C)
  - Converse Inorder (G D F E A B C)
  - Converse Postorder (G F E D C B A)

---

3. Write the algorithm of Preorder, Inorder and Postorder traversal techniques of the binary tree.

Procedure : RPREORDER(T)

- Given a binary tree whose root node address is given by pointer variable T and whose node structure is same as described below. This procedure traverses the tree in preorder, in a recursive manner.

1. **[Check for empty Tree]**
   If T = NULL
   then write ('Empty Tree')
   return
   else write (DATA(T))

2. **[Process the Left Subtree]**
   If LPTR (T) ≠ NULL
   then RPREORDER (LPTR (T))

3. **[Process the Right Subtree]**
   If RPTR (T) ≠ NULL
   then RPREORDER (RPTR (T))

4. **[Finished]**
   return

Procedure : RINORDER(T)

- Given a binary tree whose root node address is given by pointer variable T and whose node structure is same as described below. This procedure traverses the tree in inorder, in a recursive manner.

1. **[Check for empty Tree]**
   If T = NULL
   then write ('Empty Tree')
   return

2. **[Process the Left Subtree]**
   If LPTR (T) ≠ NULL
   then RINORDER (LPTR (T))

3. **[Process the root node]**
   write (DATA(T))

4. **[Process the Right Subtree]**
   If RPTR (T) ≠ NULL
   then RINORDER (RPTR (T))

5. **[Finished]**
   return

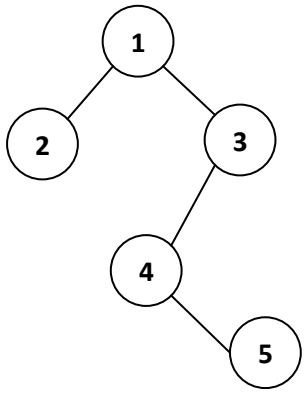
Procedure : RPOSTORDER(T)

- Given a binary tree whose root node address is given by pointer variable T and whose node structure is same as described below. This procedure traverses the tree in postorder, in a recursive manner.

1. **[Check for empty Tree]**
   If T = NULL
   then write ('Empty Tree')
   return
2. **[Process the Left Subtree]**
   If LPTR (T) ≠ NULL
   then RPOSTORDER (LPTR (T))
3. **[Process the Right Subtree]**
   If RPTR (T) ≠ NULL
   then RPOSTORDER (RPTR (T))
4. **[Process the root node]**
   write (DATA(T))
5. **[Finished]**
   return

---

4. Give traversal order of following tree into Inorder, Preorder and Postorder.



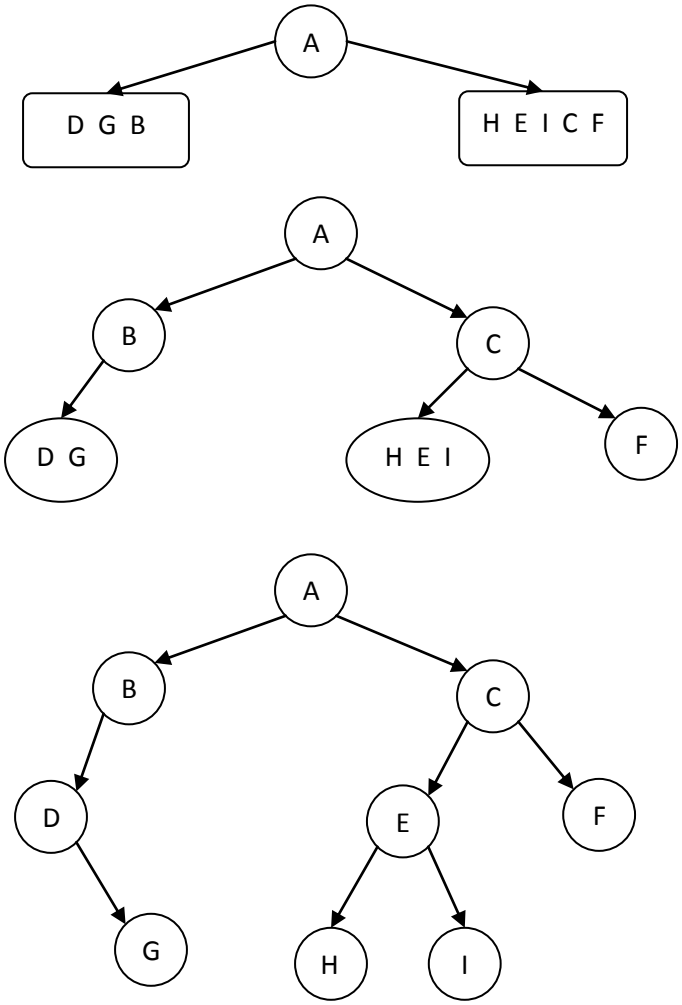
Inorder: 2 1 4 5 3
Preorder: 1 2 3 4 5
Post order: 2 5 4 3 1

---

5. Construct a tree for the given Inorder and Postorder traversals

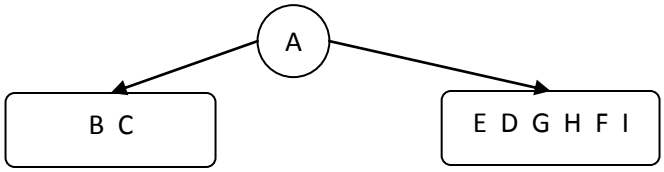
Inorder : D G B A H E I C F

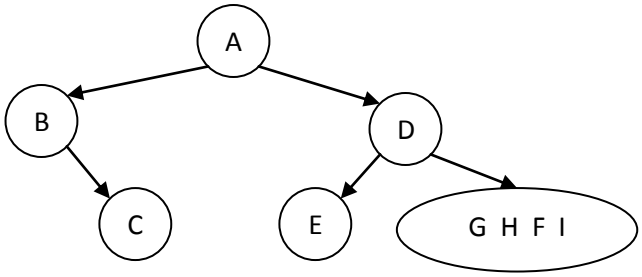
Postorder : G D B H I E F C A

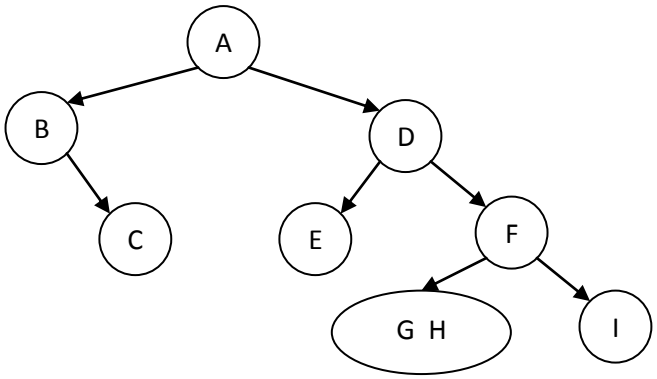


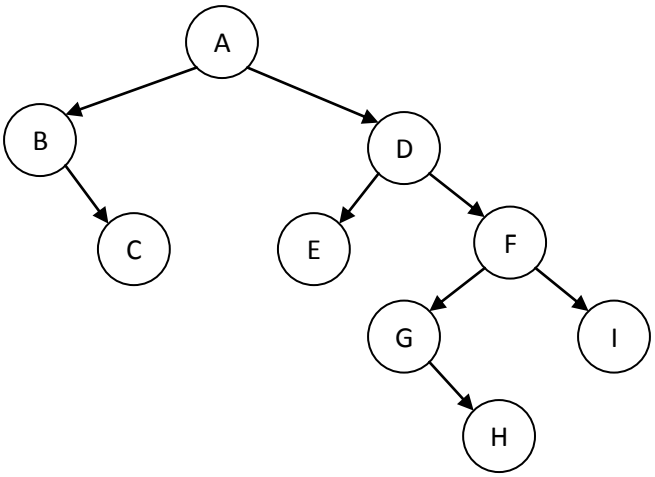
Postorder : C B E H G I F D A

Inorder : B C A E D G H F I



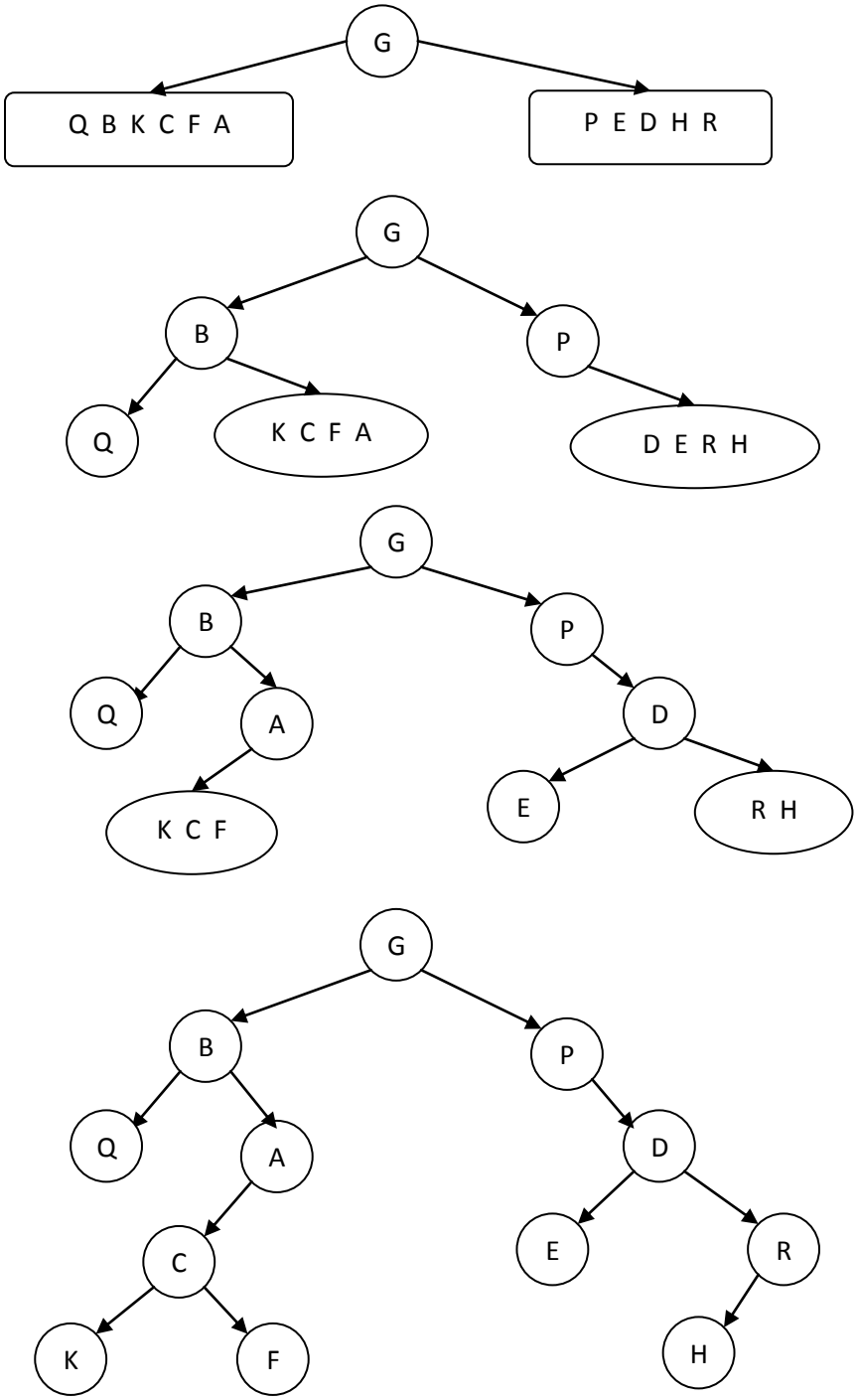




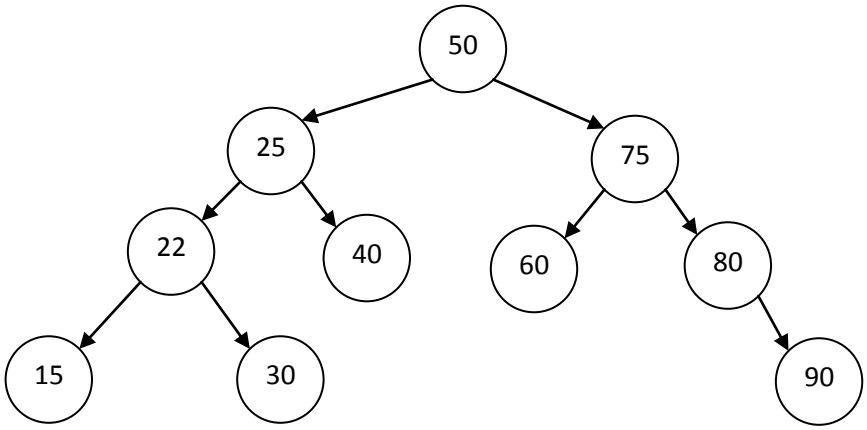


6. Construct a tree for the given Inorder and Preorder traversals

Preorder : G B Q A C K F P D E R H
Inorder : Q B K C F A G P E D H R

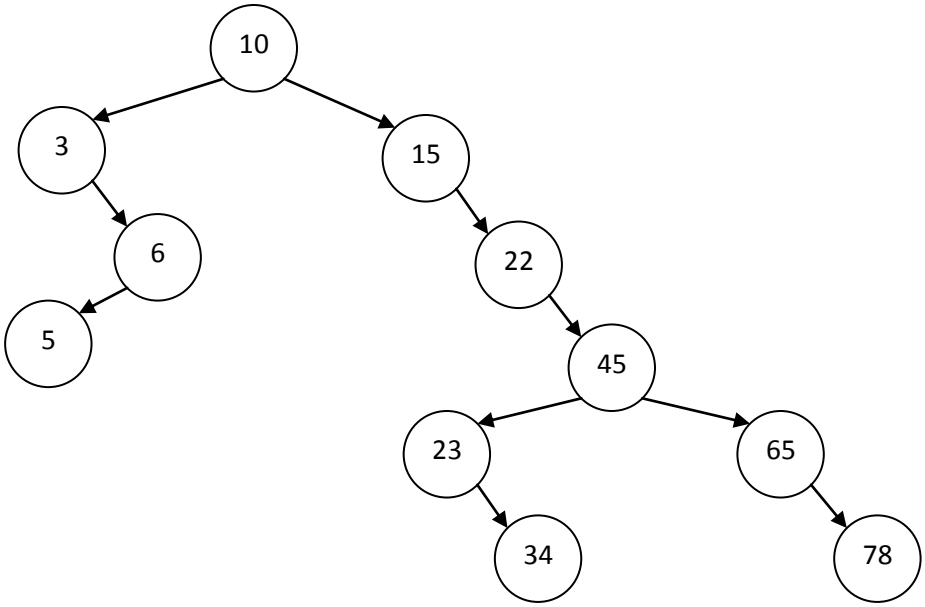


7. Create a binary search tree for the following data :
50 ,25 ,75, 22,40,60,80,90,15,30



---

8. Construct binary search tree for the following data and find its Inorder, Preorder and Postorder traversal
10,3,15,22,6,45,65,23,78,34,5



Preorder : 10, 3, 6, 5, 15, 22, 45, 23, 34, 65, 78
Inorder : 3, 5, 6, 10, 15, 22, 23, 34, 45, 65, 78
Postorder : 5, 6, 3, 34, 23, 78, 65, 45, 22, 15, 10

9. Write a short note on threaded binary tree

- The wasted NULL links in the binary tree storage representation can be replaced by threads.
- A binary tree is threaded according to particular traversal order. e.g.: Threads for the inorder traversals of tree are pointers to its higher nodes, for this traversal order.
    - If left link of node P is null, then this link is replaced by the address of its predecessor.
    - If right link of node P is null, then it is replaced by the address of its successor
- Because the left or right link of a node can denote either structural link or a thread, we must somehow be able to distinguish them.
- Method 1:- Represent thread a –ve address.
- Method 2:- To have a separate Boolean flag for each of left and right pointers, node structure for this is given below,

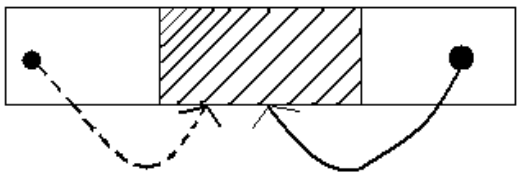
LPTR	LTHREAD	Data	RTHREAD	RPTR

Alternate node for threaded binary tree.

- LTHREAD = true = Denotes leaf thread link
- LTHREAD = false = Denotes leaf structural link
- RTHREAD = true = Denotes right threaded link
- RTHREAD = false = Denotes right structural link

- Head node is simply another node which serves as the predecessor and successor of first and last tree nodes. Tree is attached to the left branch of the head node

Head



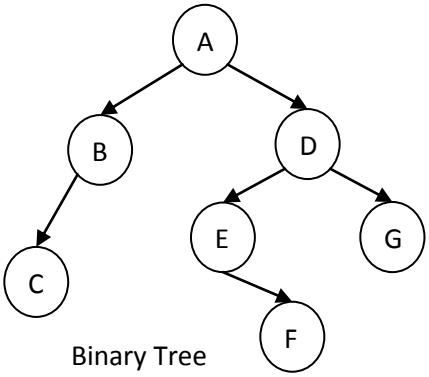
Advantages

- Inorder traversal is faster than unthreaded version as stack is not required.
- Effectively determines the predecessor and successor for inorder traversal, for unthreaded tree this task is more difficult.
- A stack is required to provide upward pointing information in tree which threading provides.
- It is possible to generate successor or predecessor of any node without having over head of stack with the help of threading.

Disadvantages

- Threaded trees are unable to share common subtrees

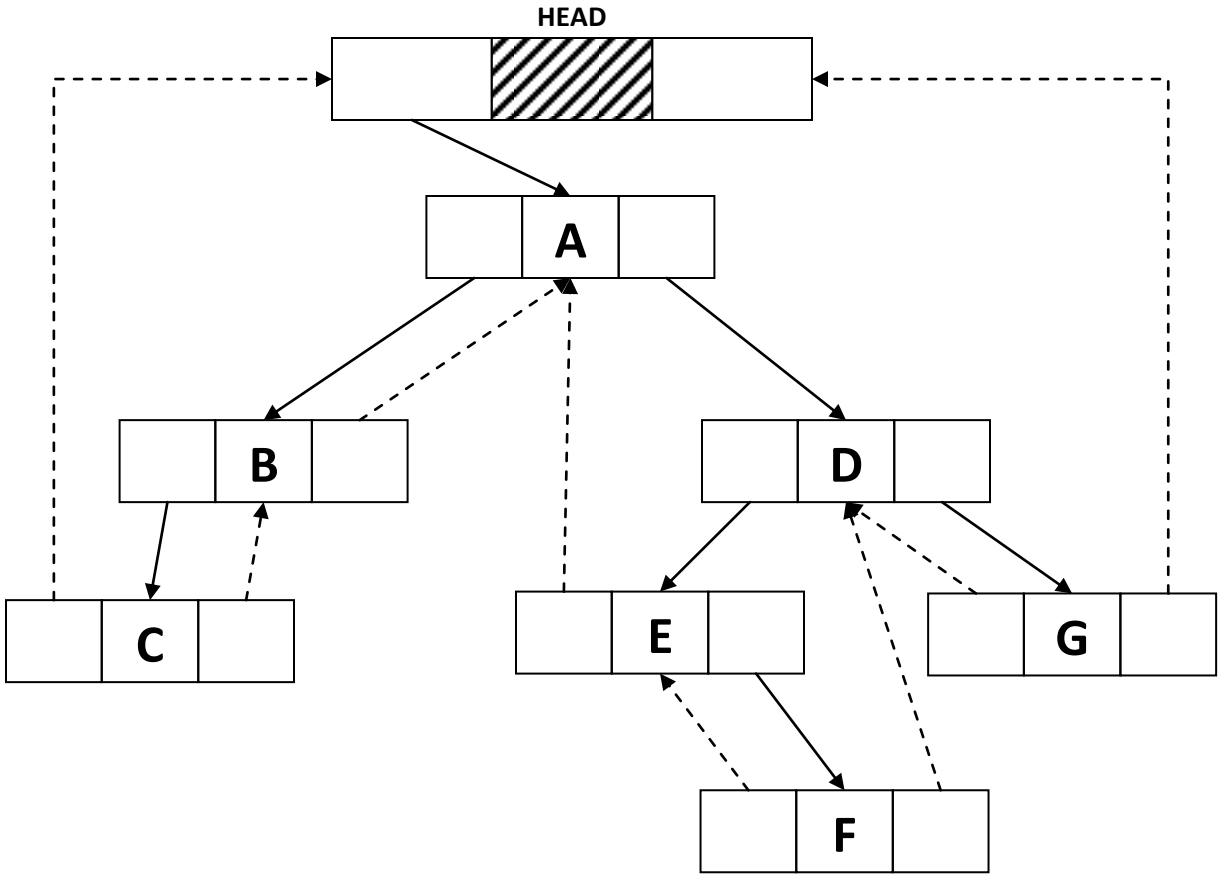
- If –ve addressing is not permitted in programming language, two additional fields are required.
- Insertion into and deletion from threaded binary tree are more time consuming because both thread and structural link must be maintained.



Binary Tree

Inorder Traversal C B A E F D G

Fully In-threaded binary tree of given binary tree



10. Draw a right in threaded binary tree for the given tree

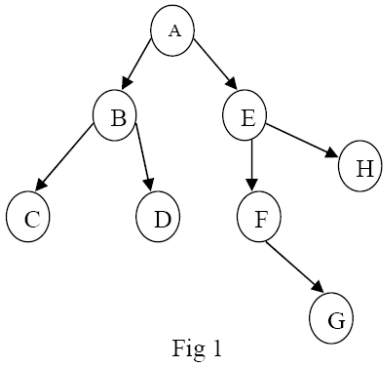
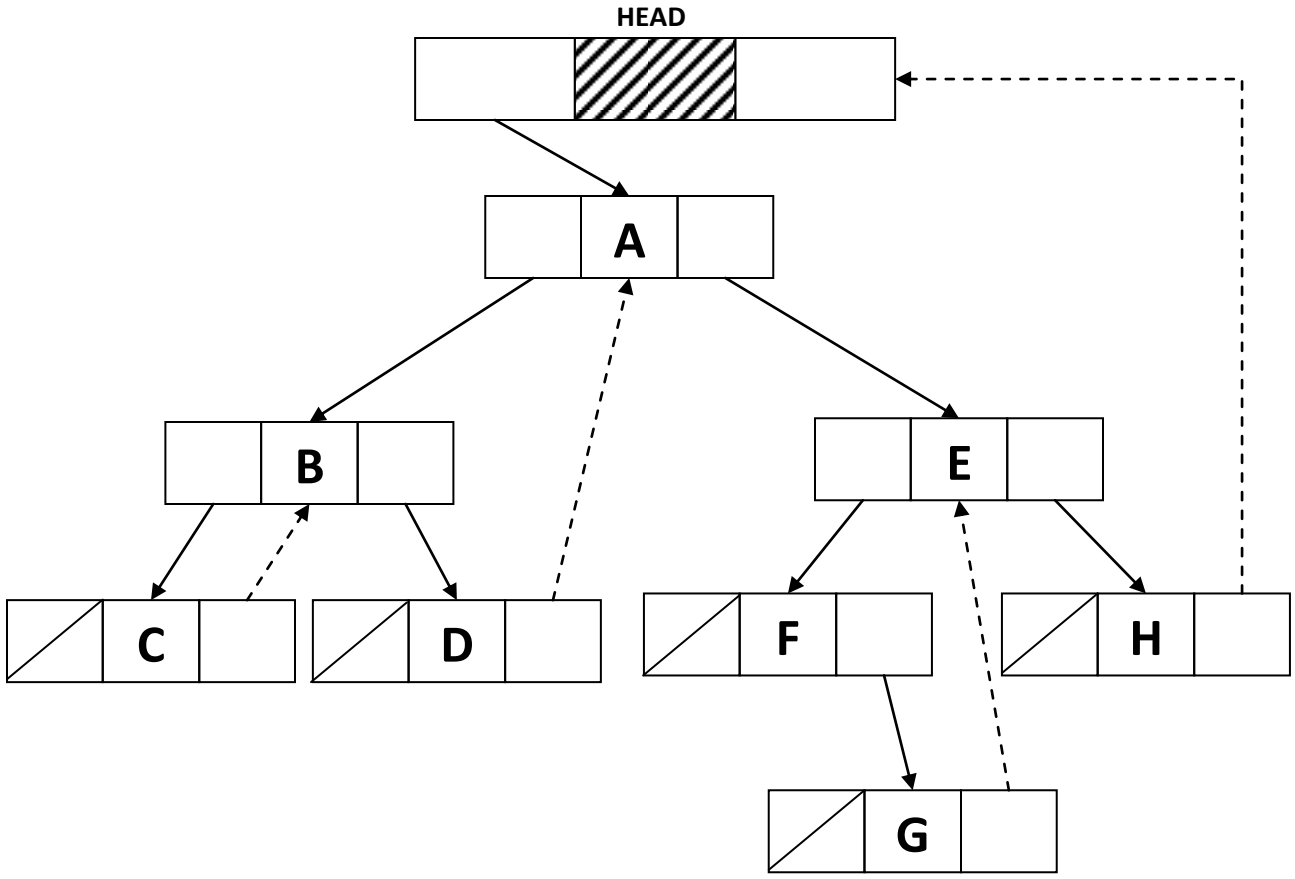


Fig 1

Right In-threaded binary tree of given binary tree



11. What is the meaning of height balanced tree? How rebalancing is done in height balanced tree.

A tree is called AVL (height balance binary tree), if each node possesses one of the following properties

1. A node is called left heavy if the longest path in its left sub tree is one longer then the longest path of its right sub tree.
2. A node is called right heavy if the longest path in the right sub tree is one longer than path in its left sub tree.
3. A node is called balanced, if the longest path in both the right and left sub tree are equal.

If tree becomes unbalanced by inserting any node, then based on position of insertion, we need to rotate the unbalanced node. Rotation is the process to make tree balanced

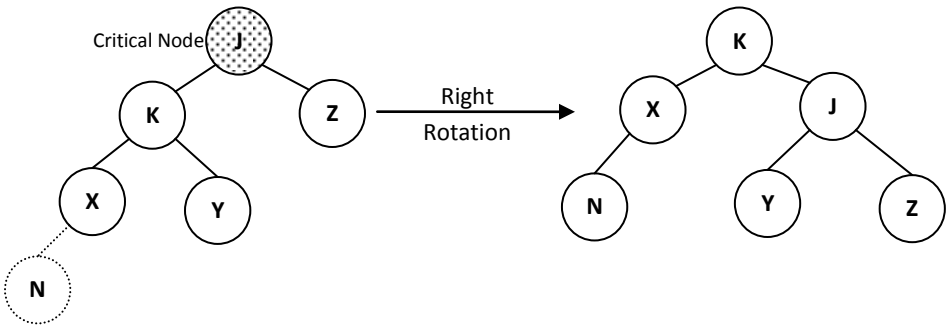
1) Insertion into Left sub-tree of nodes Left child – Single Right Rotation
2) Insertion into Right sub-tree of node's Left child – Left Right Rotation
3) Insertion into Left sub-tree of node's Right child – Right Left Rotation
4) Insertion into Right sub-tree of node's Right child – Single Left Rotation

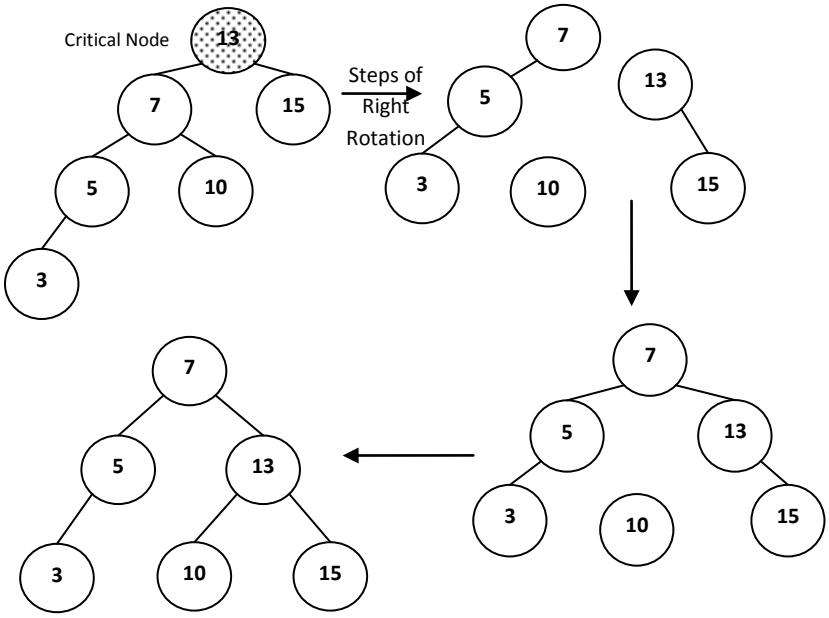
1) Insertion into Left sub-tree of nodes Left child – Single Right Rotation

If node becomes unbalanced after insertion of new node at Left sub-tree of nodes Left child, then we need to perform Single Right Rotation for unbalanced node.

Right Rotation

a. Detach leaf child's right sub-tree
b. Consider leaf child to be the new parent
c. Attach old parent onto right of new parent
d. Attach old leaf child's old right sub-tree as leaf sub-tree of new right child

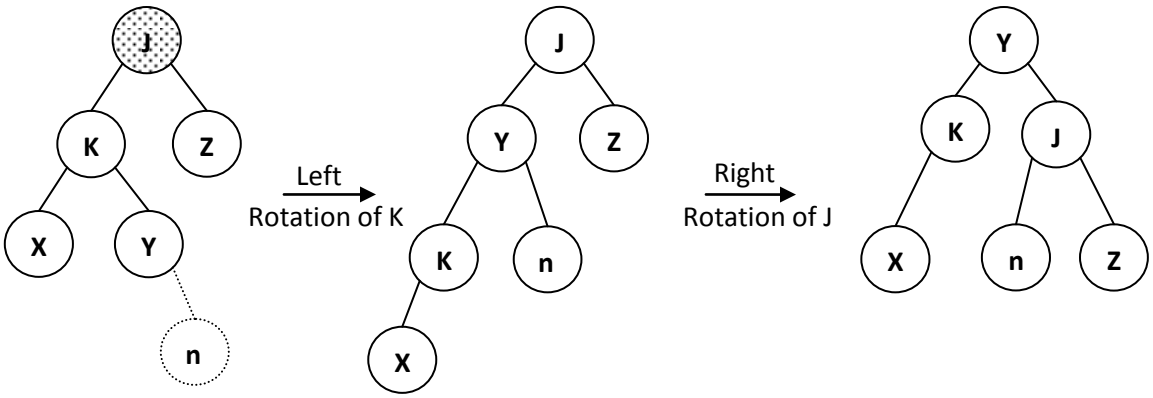




2) Insertion into Right sub-tree of node's Left child – Left Right Rotation

If node becomes unbalanced after insertion of new node at Right sub-tree of node's Left child, then we need to perform Left Right Rotation for unbalanced node.

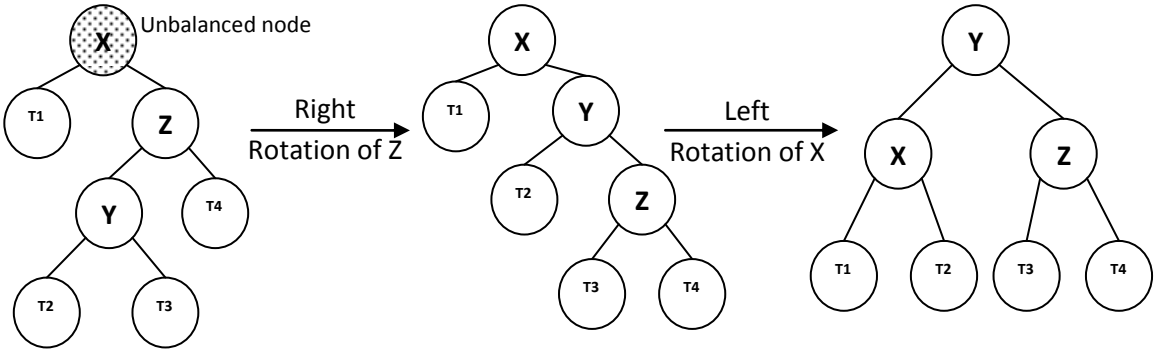
Leaf rotation of leaf child followed by right rotation of parent



3) Insertion into Left sub-tree of node's Right child – Right Left Rotation

If node becomes unbalanced after insertion of new node at Left sub-tree of node's Right child, then we need to perform Right Left Rotation for unbalanced node.

Single right rotation of right child followed by left rotation of parent

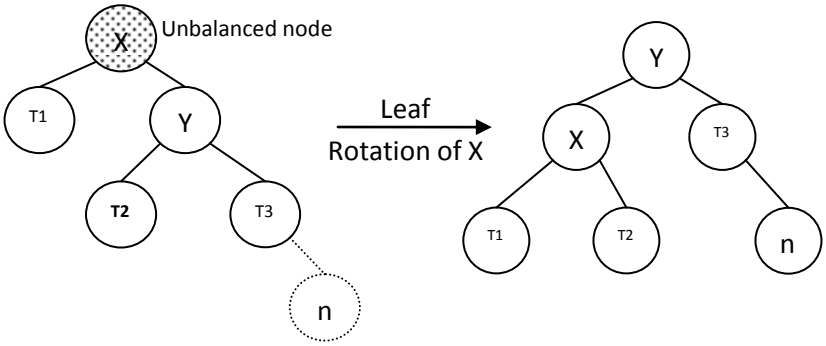


4) Insertion into Right sub-tree of node's Right child – Single Left Rotation

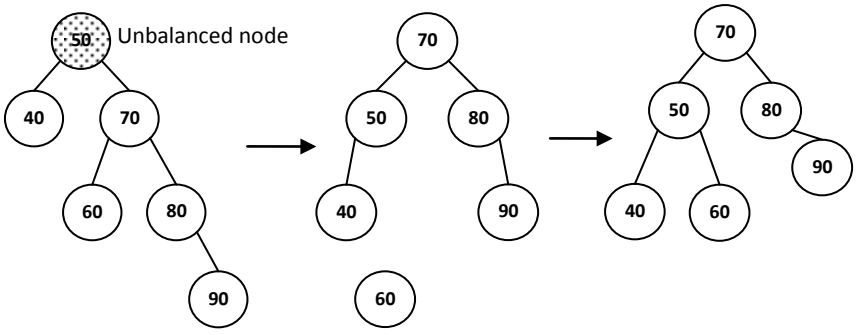
If node becomes unbalanced after insertion of new node at Right sub-tree of nodes Right child, then we need to perform Single Left Rotation for unbalanced node.

Left Rotation

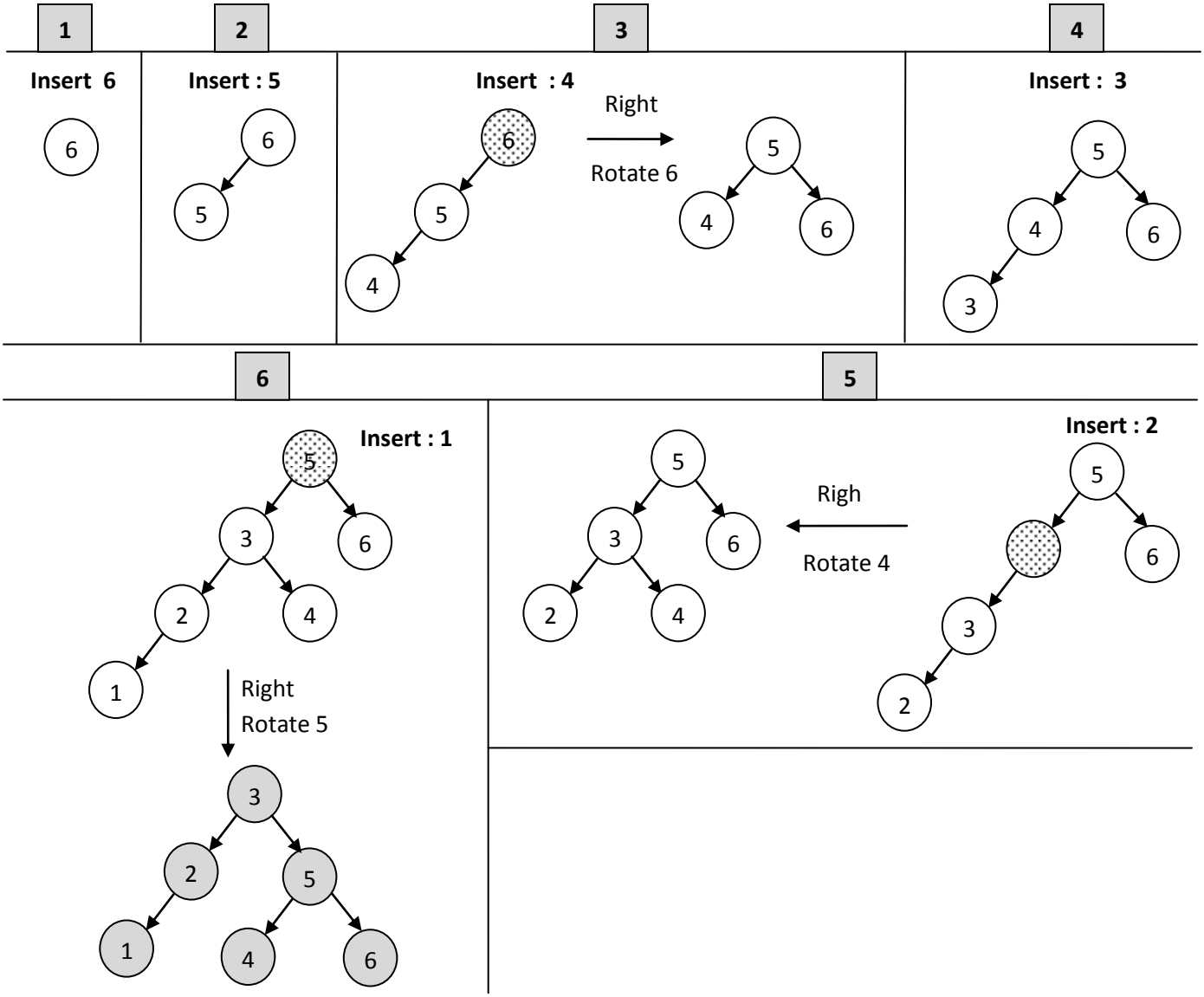
- **a.** Detach right child's leaf sub-tree
- **b.** Consider right child to be new parent
- **c.** Attach old parent onto left of new parent
- **d.** Attach old right child's old left sub-tree as right sub-tree of new left child



Example



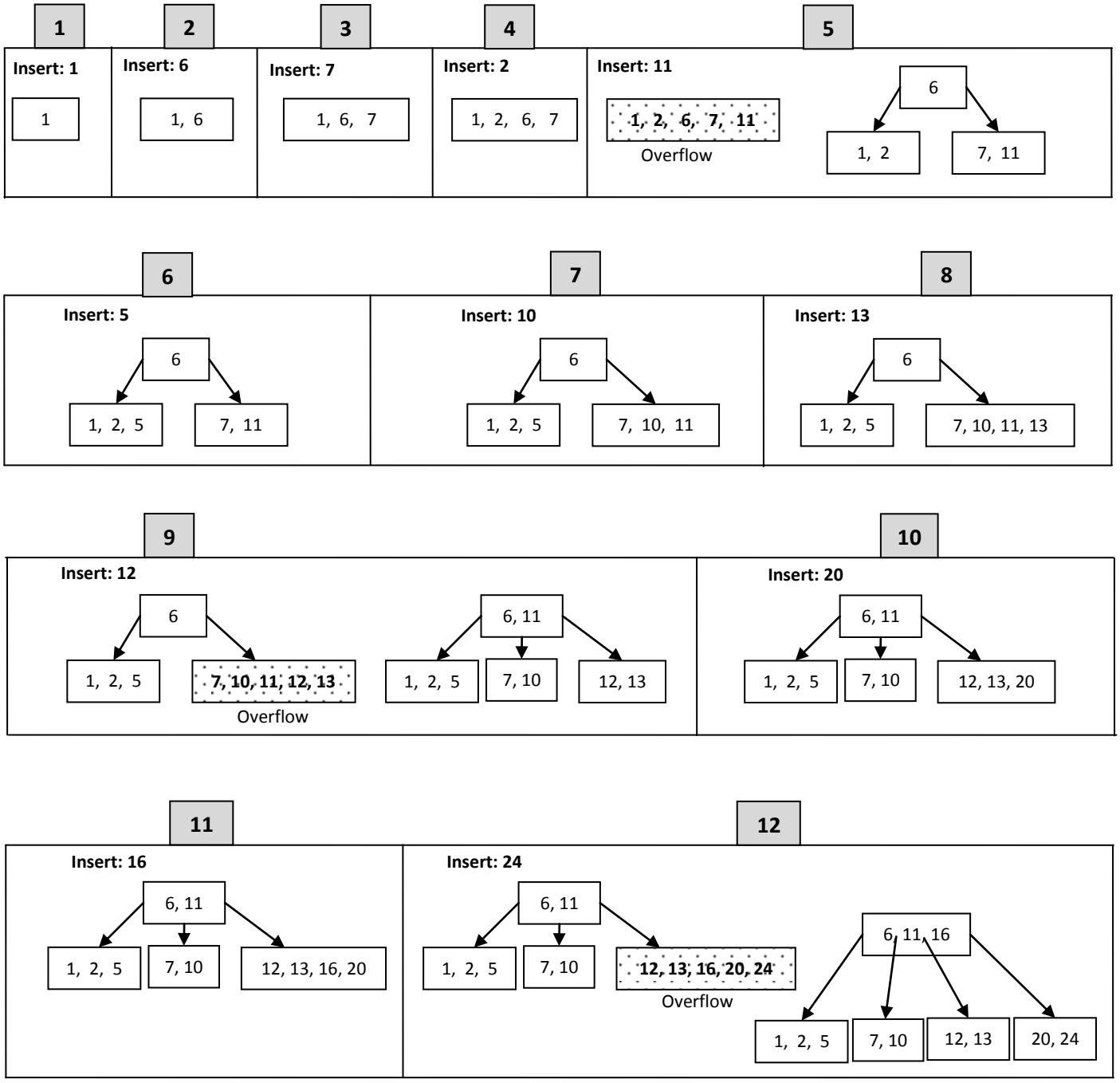
12. Construct AVL Search tree by inserting following elements in order of their occurrence 6, 5, 4, 3, 2, 1

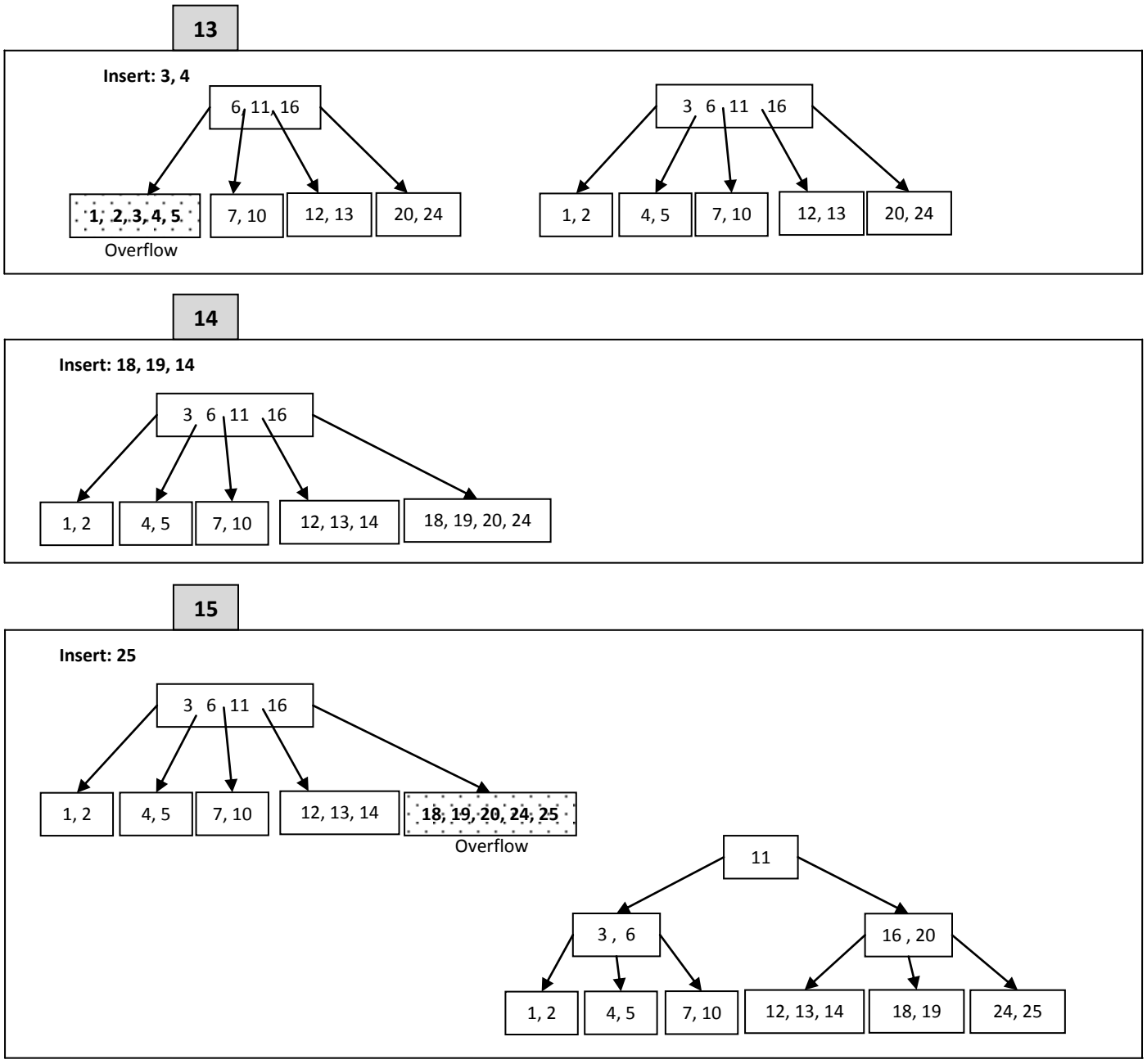


Assignment:

- Define height of the binary tree. Define height balanced tree with its advantages. Construct a height balanced binary tree (AVL tree) for the following data 42,06,54,62,88,50,22,32,12,33
- Construct the AVL search tree by inserting the following elements in the order of their occurrence. 64, 1, 44, 26, 13, 110, 98, 85

13. What are the advantages of Multiway search tree in disc access?
Construct B tree of order 5 for the following data
1,6,7,2,11,5,10,13,12,20,16,24,3,4,18,19,14,25





Assignment:

- Construct mutiway search tree for the following data of order for 100, 150, 50, 55, 250, 200, 170, 65, 75, 20, 30, 52, 10, 25, 180, 190, 300, 5

14. What is 2-3 tree?

A 2-3 tree is a type of data structure with following properties.

- All data appears at the leaves.
- Data elements are ordered from left (minimum) to right (maximum).
- Every path through the tree is the same length.
- Interior nodes have 2 or 3 subtrees.

---

15. What is graph? How it can be represented using adjacency matrix, what is path matrix? How path matrix can be found out using adjacency matrix .

Graph

- A graph G consist of a non empty set V called the set of nodes (points, vertices) of the graph, a set E which is the set of edges and a mapping from the set of edges E to a set of pairs of elements of V.
- It is also convenient to write a graph as G=(V,E).
- Notice that definition of graph implies that to every edge of a graph G, we can associate a pair of nodes of the graph. If an edge X Є E is thus associated with a pair of nodes (u,v) where u, v Є V then we says that edge x connect U and V.

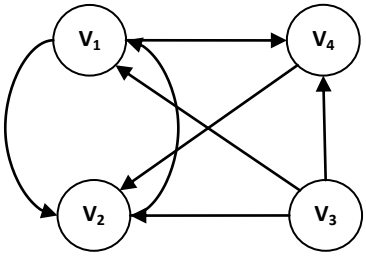
Adjacency matrix

Let G = (V, E) be a simple diagraph in which V = {v_1, v_2,…., v_n} and the nodes are assumed to be ordered from v_1 to v_n. An n x n matrix A whose elements are a_{ij} are given by

$$\mathbf{a_{ij}} = \begin{cases} 1 & if (Vi, Vj) \in E \\ 0 & otherwise \end{cases}$$

is called adjacency matrix of the graph G.

- Any element of the adjacency matrix is either 0 or 1.
- For a given graph G =m (V, E), an adjacency matrix depends upon the ordering of the elements of V.
- For different ordering of the elements of V we get different adjacency matrices.



	V_1	V_2	V_3	V_4
V_1	0	1	0	1
V_2	1	0	0	0
V_3	1	1	0	1
V_4	0	1	0	0

A digraph and its adjacency matrix

- We can extend the idea of matrix representation to multigraph and weighted graphs. In the case of multigraph or weighted graph we write a_{ji} = w, where a_{ij} denotes either the multiplicity or the weight of the edge.

Path matrix

- An entry of 1 in the ith row and jth column of A shows the existence of an edge (v_i, v_j), that is a path of length 1 from v_i to v_j.
- Let denote the elements of A^2 by $a_{ij}^{(2)}$. Then $a_{ij}^{(2)}\{\sum_{k=1}^{n} a_{ik}a_{kj}$
- Therefore $a_{ij}^{(2)}$ is equal to the number of different paths of exactly length 2 from v_i to v_j.
- Similarly element in i^{th} row and j^{th} column of A^3 gives number of paths of exactly length 3 from v_i to v_j.

$$A^2 = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 2 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix} \quad A^3 = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 2 & 2 & 0 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \quad A^4 = \begin{bmatrix} 1 & 2 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 2 & 3 & 0 & 2 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Different path matrices

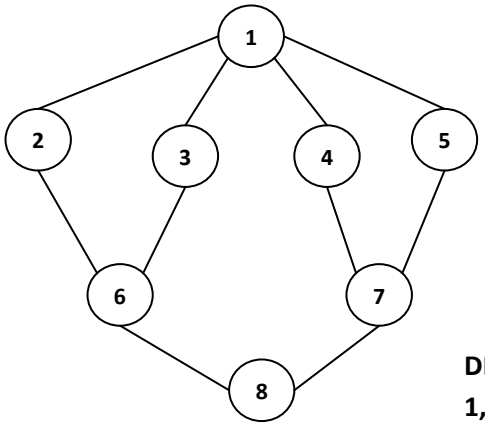
---

16. Which are the basic traversing techniques of the Graph? Write the algorithm of them.

- Most graph problems involve traversal of a graph. Traversal of a graph means visit each node exactly once.
- Two commonly used graphs traversal techniques are
  1. Depth First Search (DFS)
  2. Breadth First Search (BFS)

Depth First Search (DFS)

- It is like preorder traversal of tree.
- Traversal can start from any vertex vi
- Vi is visited and then all vertices adjacent to vi are traversed recursively using DFS



Graph G

DFS (G, 1) is given by

a) Visit (1)
b) DFS (G, 2)
DFS (G, 3)
DFS (G, 4)
DFS (G, 5)

DFS traversal of given graph is:
1, 2, 6, 3, 8, 7, 4, 5

- Since graph can have cycles, we must avoid re-visiting a node. To do this when we visit a vertex V, we marks it visited as visited should not be selected for traversal.

Procedure : DFS (vertecx V)

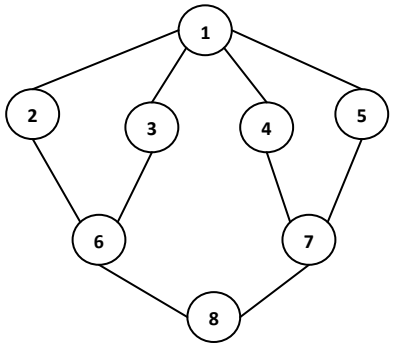
This procedure traverse the graph G in DFS manner. V is a starting vertex to be explored. S is a Stack, visited[] is an array which tells you whether particular vertex is visited or not. W is a adjacent node of vertex V. PUSH and POP are functions to insert and remove from stack respectively.

1. [Initialize TOP and Visited]
   visited[] ← 0
   TOP ← 0
2. [Push vertex into stack]
   PUSH (V)
3. [Repeat while stack is not empty]
   Repeat step 3 while stack is not empty

```
v ← POP()
if      visited[v] is 0
then    visited [v] ← 1
        for all W adjacent to v
                if      visited [w] is 0
                then    PUSH (W)
        end for
end if
```

Breadth First Search (BFS)

- This methods starts from vertex v_0
- V_0 is marked as visited. All vertices adjacent to v_0 are visited next
- Let vertices adjacent to v_0 are v_1, v_2, v_3, v_4
- v_1, v_2, v_3 and v_4 are marked visited.
- All unvisited vertices adjacent to v_1, v_2, v_3, v_4 are visited next.
- The method continuous until all vertices are visited
- The algorithm for BFS has to maintain a list of vertices which have been visited but not explored for adjacent vertices. The vertices which have been visited but not explored for adjacent vertices can be stored in queue.
- Initially the queue contains the starting vertex.
- In every iteration, a vertex is removed from the queue and its adjacent vertices which are not visited as yet are added to the queue.
- The algorithm terminates when the queue becomes empty.



Graph G

BFS traversal of given graph is:
1 | 2, 3, 4, 5 | 6, 7 | 8

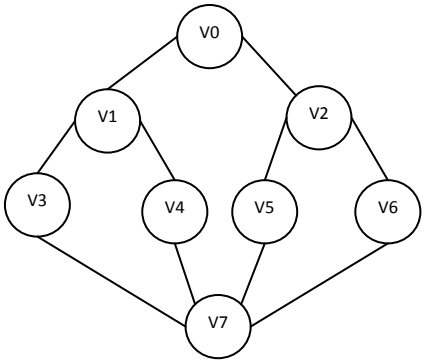
Procedure : BFS (Vertex V)

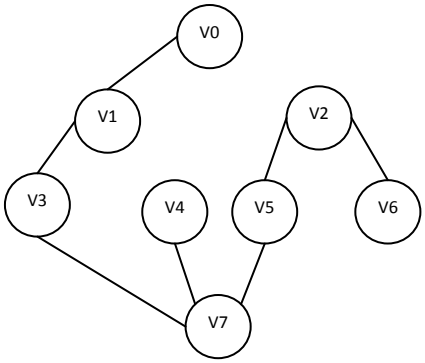
This procedure traverse the graph G in BFS manner. V is a starting vertex to be explored. Q is a queue, visited[] is an array which tells you whether particular vertex is visited or not. W is a adjacent node of vertex V.

1. Initialize Q
2. [Marks visited of V as 1]
   visited [v]← 1
3. [Add vertex v to Q]
   InsertQueue(V)
4. [Repeat while Q is not empty]
   Repeat while Q is not empty
   v ← RemoveFromQueue()
   For all vertices W adjacent to v
   if visited[w] is 0
   then visited[w] ←1
   InsertQueue(w)

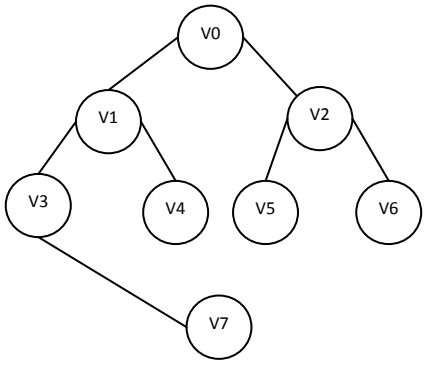
17. What is spanning tree?

- A Spanning tree of a graph is an undirected tree consisting of only those edges necessary to connect all the nodes in the original graph
- A spanning tree has the properties that
  - For any pair of nodes there exists only one path between them
  - Insertion of any edge to a spanning tree forms a unique cycle
- The particular Spanning for a graph depends on the criteria used to generate it.
- If DFS search is use, those edges traversed by the algorithm forms the edges of tree, referred to as Depth First Spanning Tree.
- If BFS Search is used, the spanning tree is formed from those edges traversed during the search, producing Breadth First Search Spanning tree.





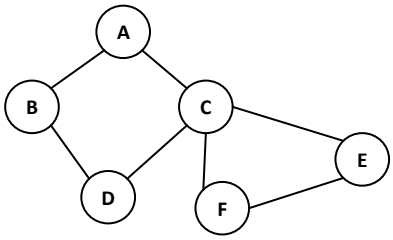
DFS Spanning Tree

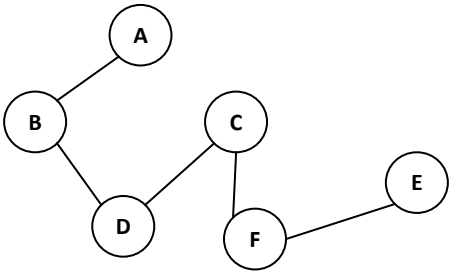


BFS Spanning Tree

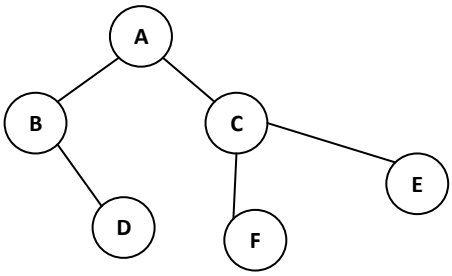
---

18. Consider the graph shown in Fig Find depth-first and breadth first traversals of this graph starting at A



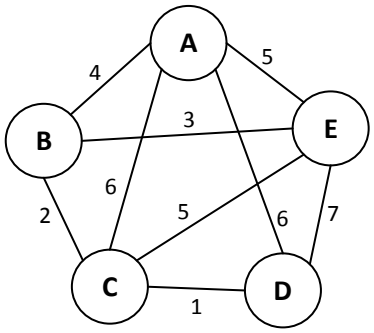


DFS : A B D C F E



BFS : A B C D F E

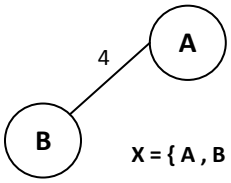
19. Define spanning tree and minimum spanning tree. Find the minimum spanning tree of the graph shown in Fig.



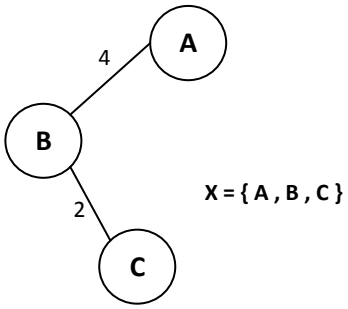
Using Prim's Algorithm:

Let X be the set of nodes explored, initially X = { A }

Step 1: Taking minimum weight edge of all Adjacent edges of **X = { A }**

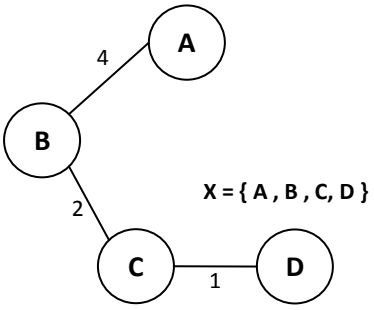


Step 2: Taking minimum weight edge of all Adjacent edges of **X = { A , B }**

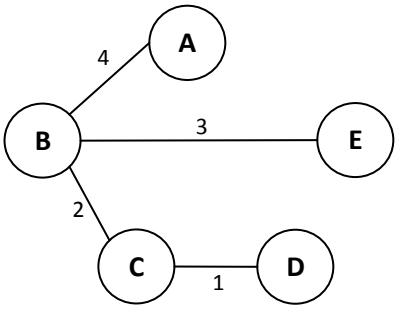


~~A – B | 4~~
A – E | 5
A – C | 6
A – D | 6
~~B – E | 3~~
~~B – C | 2~~
C – E | 6
~~C – D | 1~~
D – E | 7

Step 3: Taking minimum weight edge of all Adjacent edges of **X = { A , B , C }**

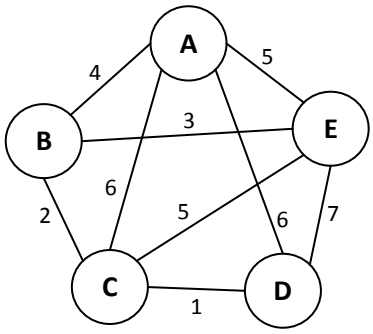


Step 4: Taking minimum weight edge of all Adjacent edges of **X = { A , B , C , D }**



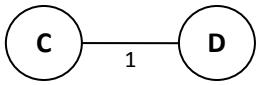
X = { A , B , C, D, E }

All nodes of graph are there with set X, so we obtained minimum spanning tree of cost**: 4 + 2 + 1 + 3 = 10**

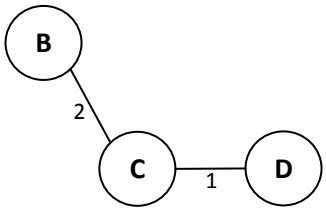


Using Kruskal's Algorithm

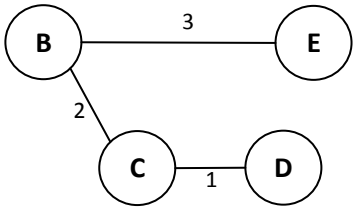
Step 1: Taking min edge (C,D)



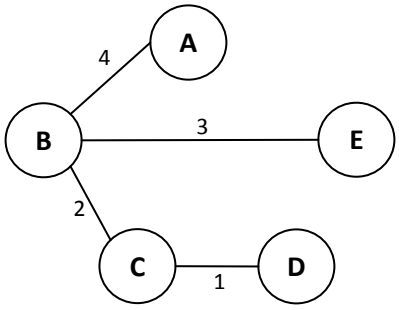
Step 2: Taking next min edge (B,C)



Step 3: Taking next min edge (B,E)



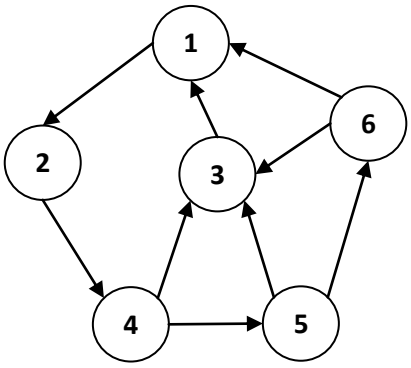
Step 4: Taking next min edge (A,B)



Step 5: Taking next min edge (A,E) it forms cycle so do not consider
Step 6: Taking next min edge (C,E) it forms cycle so do not consider
Step 7: Taking next min edge (A,D) it forms cycle so do not consider
Step 8: Taking next min edge (A,C) it forms cycle so do not consider
Step 9: Taking next min edge (E,D) it forms cycle so do not consider

All edges of graph has been visited,
so we obtained minimum spanning tree of cost:
4 + 2 + 1 + 3 = 10

20. Give example and applications of directed and undirected graphs. Find the adjacency matrix for the graph shown in Fig.



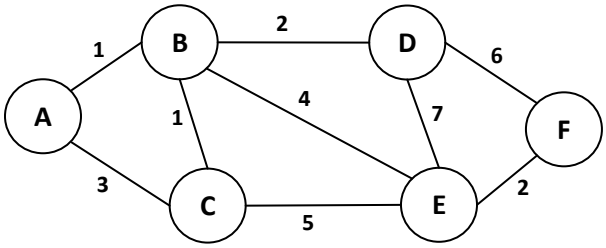
Adjacency matrix for the given graph

	1	2	3	4	5	6
1	0	**1**	0	0	0	0
2	0	0	0	1	0	0
3	**1**	0	0	0	0	0
4	0	0	**1**	0	**1**	0
5	0	0	**1**	0	0	**1**
6	**1**	0	**1**	0	0	0

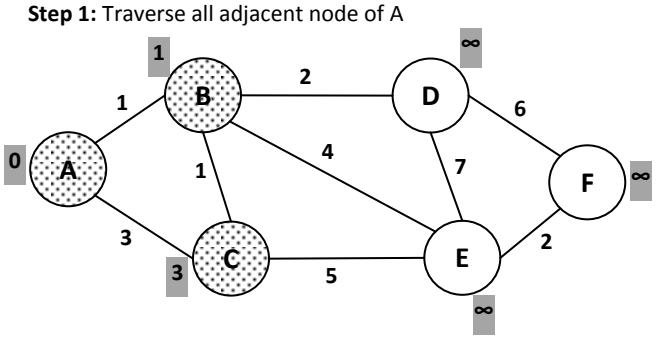
Applications of graph:

- Electronic Circuits
  - Printed Circuit Board
  - Integrated Circuit
- Transportation networks
  - Highway networks
    Modeling a road network with vertexes as towns and edge costs as distances.
  - Water Supply networks
    Modeling a water supply network. A cost might relate to current or a function of capacity and length. As water flows in only 1 direction, from higher to lower pressure connections or downhill, such a network is inherently an acyclic directed graph.
  - Flight network
    Minimizing the cost and time taken for air travel when direct flights don't exist between starting and ending airports.
- Computer networks
  - Local Area Network
  - Internet
    Dynamically modeling the status of a set of routes by which traffic might be directed over the Internet.
  - Web
    Using a directed graph to map the links between pages within a website and to analyze ease of navigation between different parts of the site.
- Databases
  - Entity Relationship Diagram

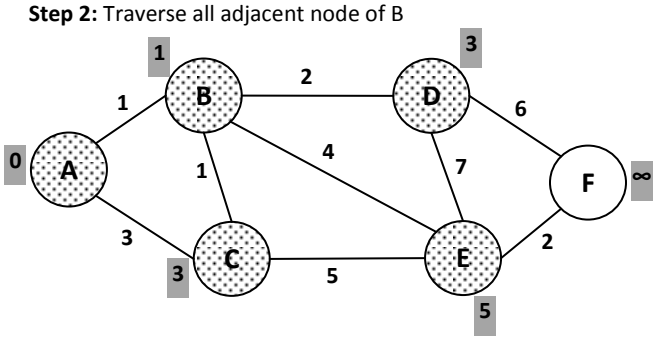
21. Apply Dijkstra's algorithm to find shortest path between vertex A and vertex F5 for the graph shown in Fig.



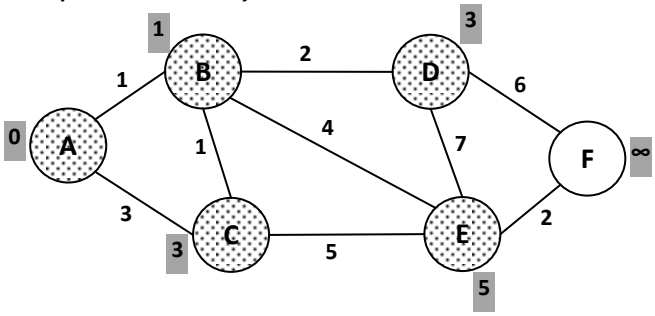
Step 1: Traverse all adjacent node of A



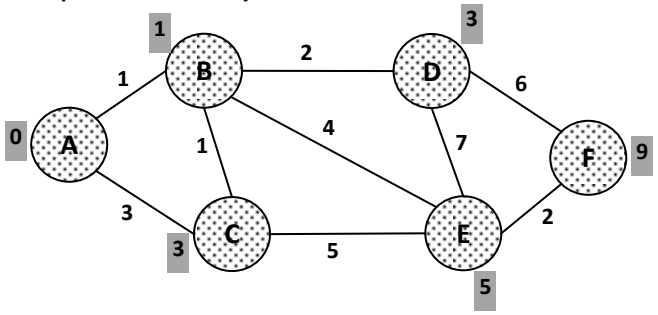
Step 2: Traverse all adjacent node of B



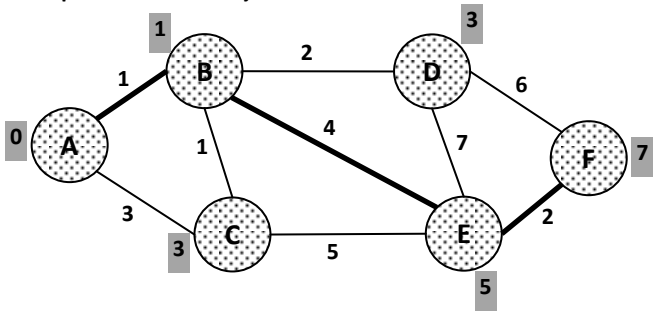
Step 3: Traverse all adjacent node of C



Step 4: Traverse all adjacent node of D



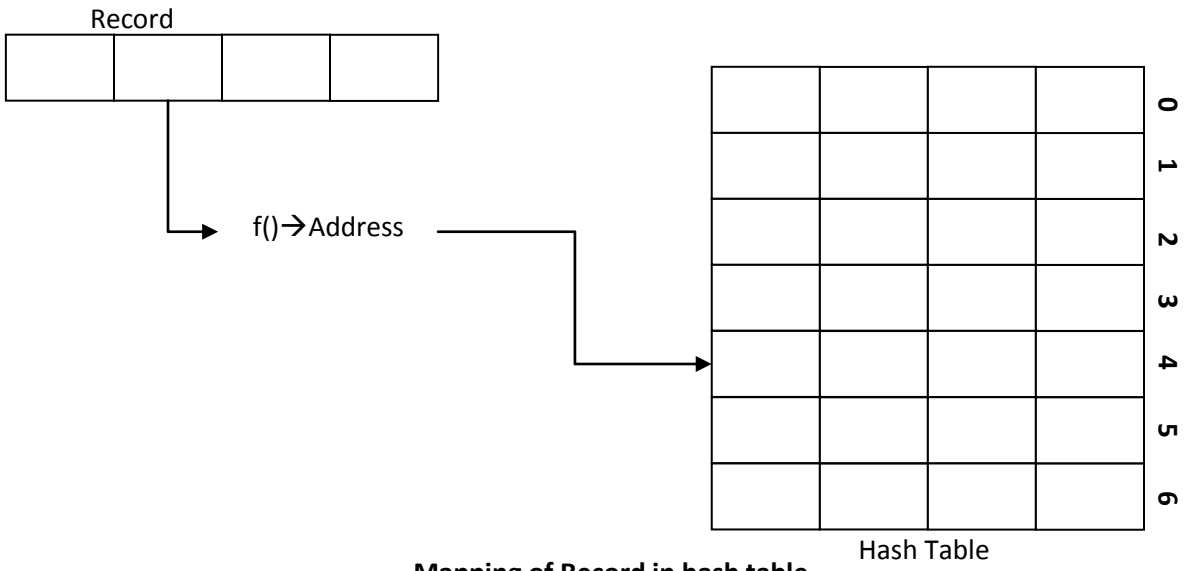
Step 5: Traverse all adjacent node of E



- Shortest path from node **A to F** is : **A – B – E – F** as shown in step 5
- Length of path is **7**

What is Hashing?

- Sequential search requires, on the average O(n) comparisons to locate an element. So many comparisons are not desirable for a large database of elements.
- Binary search requires much fewer comparisons on the average O (log n) but there is an additional requirement that the data should be sorted. Even with best sorting algorithm, sorting of elements require 0(n log n) comparisons.
- There is another widely used technique for storing of data called hashing. It does away with the requirement of keeping data sorted (as in binary search) and its best case timing complexity is of constant order (0(1)). In its worst case, hashing algorithm starts behaving like linear search.
- Best case timing behavior of searching using hashing = O(1)
- Worst case timing Behavior of searching using hashing = O(n)
- In hashing, the record for a key value "key", is directly referred by calculating the address from the key value. Address or location of an element or record, x, is obtained by computing some arithmetic function f. f(key) gives the address of x in the table.



Mapping of Record in hash table

Hash Table Data Structure:

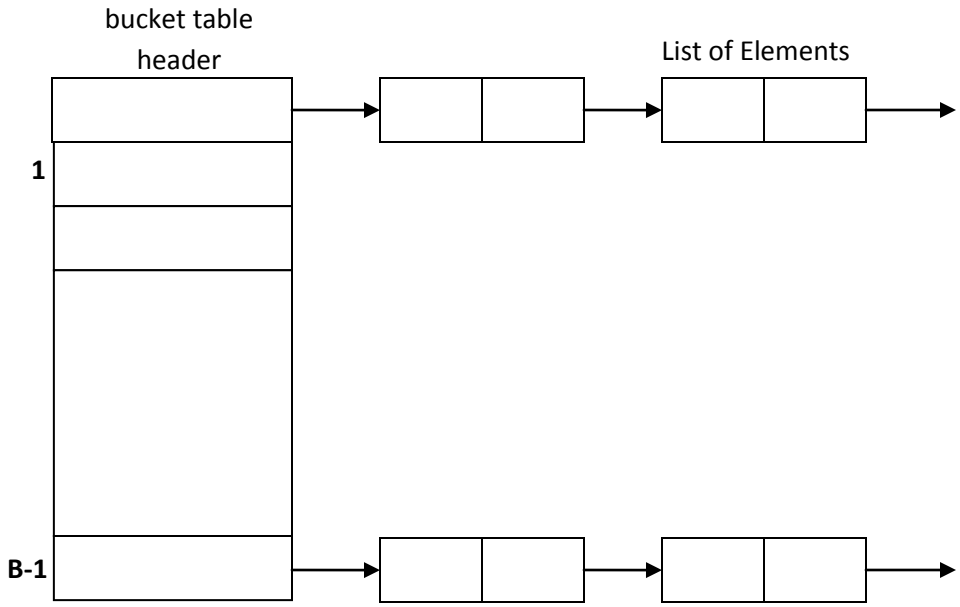
There are two different forms of hashing.

1. **Open hashing or external hashing**
   Open or external hashing, allows records to be stored in unlimited space (could be a hard disk). It places no limitation on the size of the tables.

2. Close hashing or internal hashing

Closed or internal hashing, uses a fixed space for storage and thus limits the size of hash table.

1. Open Hashing Data Structure



The open hashing data organization

- The basic idea is that the records [elements] are partitioned into B classes, numbered 0,1,2 … B-l
- A Hashing function f(x) maps a record with key n to an integer value between 0 and B-l.
- Each bucket in the bucket table is the head of the linked list of records mapped to that bucket.

2. Close Hashing Data Structure

0	b
1	
2	
3	
4	c
5	d

- A closed hash table keeps the elements in the bucket itself.
- Only one element can be put in the bucket
- If we try to place an element in the bucket f(n) and find it already holds an element, then we say that a collision has occurred.
- In case of collision, the element should be rehashed to alternate empty location f1(x), f2(x), ... within the bucket table
- In closed hashing, collision handling is a very important issue.

Hashing Functions

Characteristics of a Good Hash Function

- A good hash function avoids collisions.
- A good hash function tends to spread keys evenly in the array.
- A good hash function is easy to compute.

Different hashing functions

1. Division-Method
2. Midsquare Methods
3. Folding Method
4. Digit Analysis
5. Length Dependent Method
6. Algebraic Coding
7. Multiplicative Hashing

1. Division-Method

- In this method we use modular arithmetic system to divide the key value by some integer divisor m (may be table size).
- It gives us the location value, where the element can be placed.
- We can write,
  L = (K mod m) + 1
  where L => location in table/file
  K => key value
  m => table size/number of slots in file
- Suppose, k = 23, m = 10 then
  L = (23 mod 10) + 1= 3 + 1=4, The key whose value is 23 is placed in 4th location.

2. Midsquare Methods

- In this case, we square the value of a key and take the number of digits required to form an address, from the middle position of squared value.
- Suppose a key value is 16, then its square is 256. Now if we want address of two digits, then you select the address as 56 (i.e. two digits starting from middle of 256).

3. Folding Method

- Most machines have a small number of primitive data types for which there are arithmetic instructions.
- Frequently key to be used will not fit easily in to one of these data types
- It is not possible to discard the portion of the key that does not fit into such an arithmetic data type

- The solution is to combine the various parts of the key in such a way that all parts of the key affect for final result such an operation is termed folding of the key.
- That is the key is actually partitioned into number of parts, each part having the same length as that of the required address.
- Add the value of each parts, ignoring the final carry to get the required address.
- This is done in two ways :
    - **Fold-shifting:** Here actual values of each parts of key are added.
        - Suppose, the key is : 12345678, and the required address is of two digits,
        - Then break the key into: 12, 34, 56, 78.
        - Add these, we get 12 + 34 + 56 + 78 : 180, ignore first 1 we get 80 as location
    - **Fold-boundary:** Here the reversed values of outer parts of key are added.
        - Suppose, the key is : 12345678, and the required address is of two digits,
        - Then break the key into: 21, 34, 56, 87.
        - Add these, we get 21 + 34 + 56 + 87 : 198, ignore first 1 we get 98 as location

4. Digit Analysis

- This hashing function is a distribution-dependent.
- Here we make a statistical analysis of digits of the key, and select those digits (of fixed position) which occur quite frequently.
- Then reverse or shifts the digits to get the address.
- For example, if the key is : 9861234. If the statistical analysis has revealed the fact that the third and fifth position digits occur quite frequently, then we choose the digits in these positions from the key. So we get, 62. Reversing it we get 26 as the address.

5. Length Dependent Method

- In this type of hashing function we use the length of the key along with some portion of the key j to produce the address, directly.
- In the indirect method, the length of the key along with some portion of the key is used to obtain intermediate value.

6. Algebraic Coding

- Here a n bit key value is represented as a polynomial.
- The divisor polynomial is then constructed based on the address range required.
- The modular division of key-polynomial by divisor polynomial, to get the address-polynomial.
- Let f(x) = polynomial of n bit key = $a_1 + a_2x + \ldots\ldots + a_nx^{n-1}$
- d(x) = divisor polynomial = $x^1 + d_1 + d_2x + \ldots + d_1x^{1-1}$
- then the required address polynomial will be f(x) mod d(x)

7. Multiplicative Hashing

- This method is based on obtaining an address of a key, based on the multiplication value.

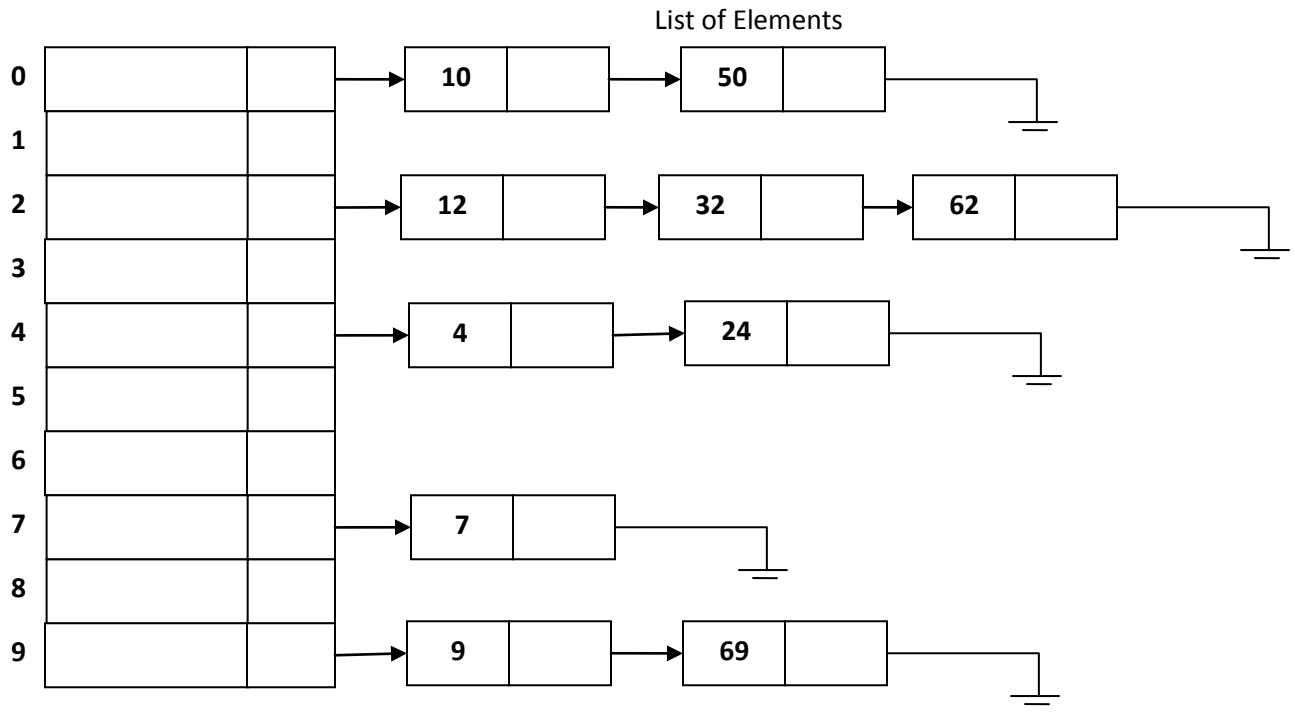
- If k is the non-negative key, and a constant c, (0 < c < 1), compute kc mod 1, which is a fractional part of kc.
- Multiply this fractional part by m and take a floor value to get the address
- $\lfloor m\ (kc\ mod\ 1) \rfloor$
- 0 < h (k) < m

Collision Resolution Strategies (Synonym Resolution)

- Collision resolution is the main problem in hashing.
- If the element to be inserted is mapped to the same location, where an element is already inserted then we have a collision and it must be resolved.
- There are several strategies for collision resolution. The most commonly used are :
    1. **Separate chaining** - used with open hashing
    2. **Open addressing** - used with closed hashing

1. Separate chaining

- In this strategy, a separate list of all elements mapped to the same value is maintained.
- Separate chaining is based on collision avoidance.
- If memory space is tight, separate chaining should be avoided.
- Additional memory space for links is wasted in storing address of linked elements.
- Hashing function should ensure even distribution of elements among buckets; otherwise the timing behavior of most operations on hash table will deteriorate.

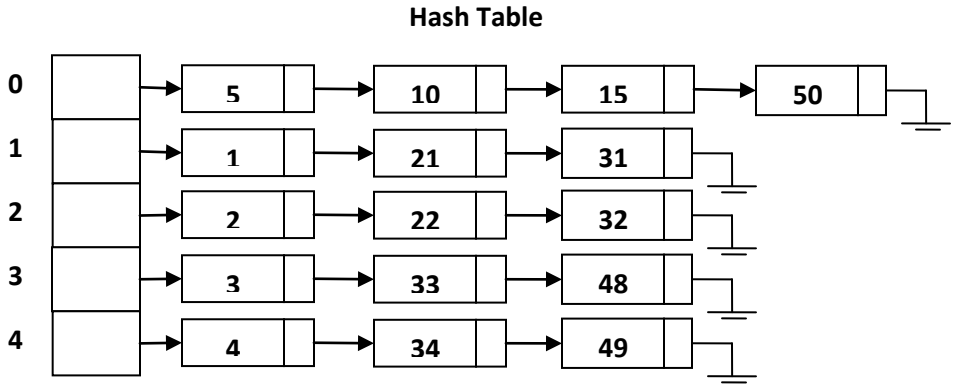


A Separate Chaining Hash Table

Example : The integers given below are to be inserted in a hash table with 5 locations using chaining to resolve collisions. Construct hash table and use simplest hash function. 1, 2, 3, 4, 5, 10, 21, 22, 33, 34, 15, 32, 31, 48, 49, 50

An element can be mapped to a location in the hash table using the mapping function **key % 10**.

Hash Table Location	Mapped element
0	5, 10, 15, 50
1	1, 21, 31
2	2, 22, 32
3	3, 33, 48
4	4, 34, 49



2. Open Addressing

- Separate chaining requires additional memory space for pointers. Open addressing hashing is an alternate method of handling collision.
- In open addressing, if a collision occurs, alternate cells are tried until an empty cell is found.
    a. Linear probing
    b. Quadratic probing
    c. Double hashing.

a) Linear Probing

- In linear probing, whenever there is a collision, cells are searched sequentially (with wraparound) for an empty cell.
- Fig. shows the result of inserting keys {5,18,55,78,35,15} using the hash function **(f(key)= key%10)** and linear probing strategy.

	Empty Table	After 5	After 18	After 55	After 78	After 35	After 15
0							15
1							
2							
3							
4							
5		5	5	5	5	5	5
6				55	55	55	55
7						35	35
8			18	18	18	18	18
9					78	78	78

- Linear probing is easy to implement but it suffers from "**primary clustering**"

- When many keys are mapped to the same location (clustering), linear probing will not distribute these keys evenly in the hash table. These keys will be stored in neighborhood of the location where they are mapped. This will lead to clustering of keys around the point of collision

b) Quadratic probing

- One way of reducing "primary clustering" is to use quadratic probing to resolve collision.
- Suppose the "key" is mapped to the location j and the cell j is already occupied. In quadratic probing, the location j, (j+1), (j+4), (j+9), ... are examined to find the first empty cell where the key is to be inserted.
- This table reduces primary clustering.
- It does not ensure that all cells in the table will be examined to find an empty cell. Thus, it may be possible that key will not be inserted even if there is an empty cell in the table.

c) Double Hashing

- This method requires two hashing functions f1 (key) and f2 (key).
- Problem of clustering can easily be handled through double hashing.
- Function f1 (key) is known as primary hash function.
- In case the address obtained by f1 (key) is already occupied by a key, the function f2 (key) is evaluated.
- The second function f2 (key) is used to compute the increment to be added to the address obtained by the first hash function f1 (key) in case of collision.
- The search for an empty location is made successively at the addresses f1 (key) + f2(key), f1 (key) + 2f2 (key), f1 (key) + 3f2(key),...

What is File?

- A file is a collection of records where a record consists of one or more fields. Each contains the same sequence of fields.
- Each field is normally of fixed length.
- A sample file with four records is shown below:

Name	Roll No.	Year	Marks
AMIT	1000	1	82
KALPESH	1005	2	54
JITENDRA	1009	1	75
RAVI	1010	1	79

- There are four records
- There are four fields (Name, Roll No., Year, Marks)
- Records can be uniquely identified on the field 'Roll No.' Therefore, Roll No. is the key field.
- A database is a collection of files.
- Commonly, used file organizations are :
  1. Sequential files
  2. Relative files
  3. Direct files
  4. Indexed Sequential files
  5. Index files
- Primitive Operations on a File :
  1. Creation
  2. Reading
  3. Insertion
  4. Deletion
  5. Updation
  6. Searching

Sequential Files

It is the most common type of file. In this type of file:

- A fixed format is used for record.
- All records are of the same length.
- Position of each field in record and length of field is fixed.
- Records are physically ordered on the value of one of the fields - called the ordering field.

Block 1

Name	Roll No.	Year	Marks
AMIT	1000	1	82
KALPESH	1005	2	54
JITENDRA	1009	1	75
RAVI	1010	1	79

Block 2

NILESH	1011	2	89

Some blocks of an ordered (sequential) file of students records with Roll no. as the ordering field

Advantages of sequential file over unordered files :

- Reading of records in order of the ordering key is extremely efficient.
- Finding the next record in order of the ordering key usually, does not require additional block access. Next record may be found in the same block.
- Searching operation on ordering key is must faster. Binary search can be utilized. A binary search will require $\mathbf{\log_2 b}$ block accesses where b is the total number of blocks in the file.

Disadvantages of sequential file :

- Sequential file does not give any advantage when the search operation is to be carried out on non-ordering field.
- Inserting a record is an expensive operation. Insertion of a new record requires finding of place of insertion and then all records ahead of it must be moved to create space for the record to be inserted. This could be very expensive for large files.
- Deleting a record is an expensive operation. Deletion too requires movement of records.
- Modification of field value of ordering key could be time consuming. Modifying the ordering field means the record can change its position. This requires deletion of the old record followed by insertion of the modified record.

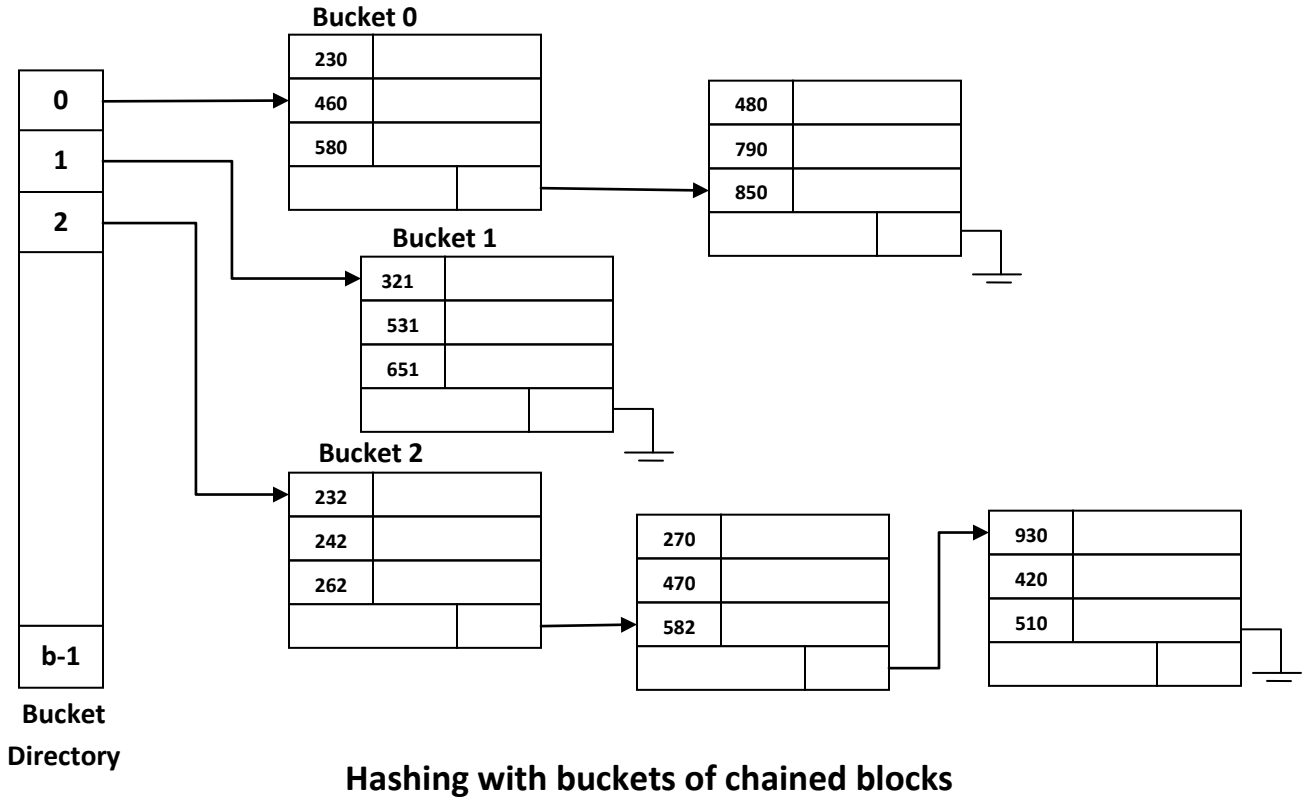
Hashing (Direct file organization):

- It is a common technique used for fast accessing of records on secondary storage.
- Records of a file are divided among buckets.
- A bucket is either one disk block or cluster of contiguous blocks.
- A hashing function maps a key into a bucket number. The buckets are numbered 0, 1,2...b-1.
- A hash function f maps each key value into one of the integers 0 through b - 1.
- If x is a key, f(x) is the number of bucket that contains the record with key x.

- The blocks making up each bucket could either be contiguous blocks or they can be chained together in a linked list.
- Translation of bucket number to disk block address is done with the help of bucket directory. It gives the address of the first block of the chained blocks in a linked list.
- Hashing is quite efficient in retrieving a record on hashed key. The average number of block accesses for retrieving a record.

$$= 1 \text{ (bucket directory)} + \frac{\boldsymbol{No\ of\ records}}{\boldsymbol{No\ of\ buckets\ x\ No\ of\ records\ per\ block}}$$

- Thus the operation is b times faster (b = number of buckets) than unordered file.
- To insert a record with key value x, the new record can added to the last block in the chain for bucket f(x). If the record does not fit into the existing block, record is stored in a new block and this new block is added at the end of the chain for bucket f(x).
- A well designed hashed structure requires two block accesses for most operations

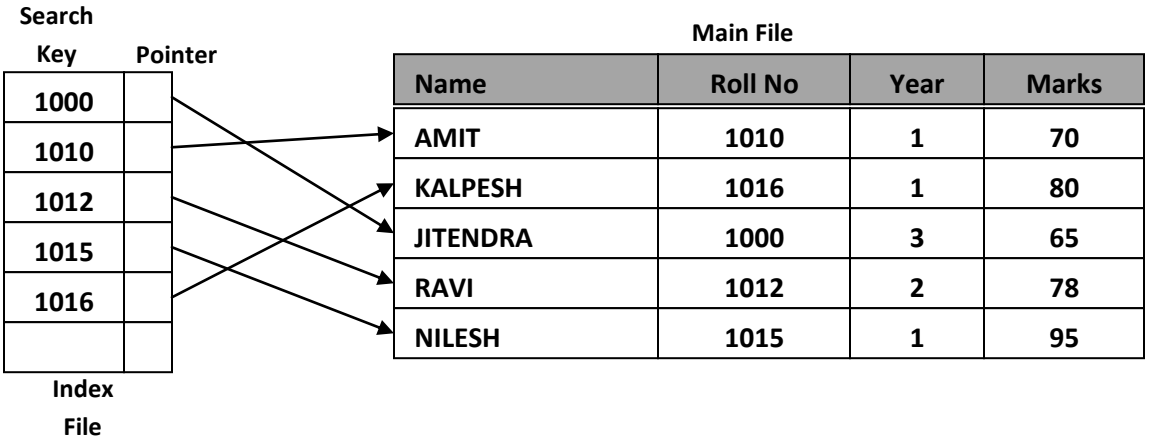


Hashing with buckets of chained blocks

Indexing

- Indexing is used to speed up retrieval of records.

- It is done with the help of a separate sequential file. Each record of in the index file consists of two fields, a key field and a pointer into the main file.
- To find a specific record for the given key value, index is searched for the given key value.
- Binary search can used to search in index file. After getting the address of record from index file, the record in main file can easily be retrieved.



- Index file is ordered on the ordering key Roll No. each record of index file points to the corresponding record. Main file is not sorted.

Advantages of indexing over sequential file:

- Sequential file can be searched effectively on ordering key. When it is necessary to search for a record on the basis of some other attribute than the ordering key field, sequential file representation is inadequate.
- Multiple indexes can be maintained for each type of field used for searching. Thus, indexing provides much better flexibility.
- An index file usually requires less storage space than the main file. A binary search on sequential file will require accessing of more blocks. This can be explained with the help of the following example. Consider the example of a sequential file with r = 1024 records of fixed length with record size R = 128 bytes stored on disk with block size B = 2048 bytes.
- Number of blocks required to store the file = $\frac{1024\ x\ 128}{2048} = 64$
- Number of block accesses for searching a record = $\log_2 64 = 6$
- Suppose, we want to construct an index on a key field that is V = 4 bytes long and the block pointer is P = 4 bytes long.
- A record of an index file is of the form <V;, Pj> and it will need 8 bytes per entry.
- Total Number of index entries = 1024
- Number of blocks b' required to store the file = $\frac{1024\ x\ 8}{2048} = 4$
- Number of block accesses for searching a record = $\log_2 4 = 2$

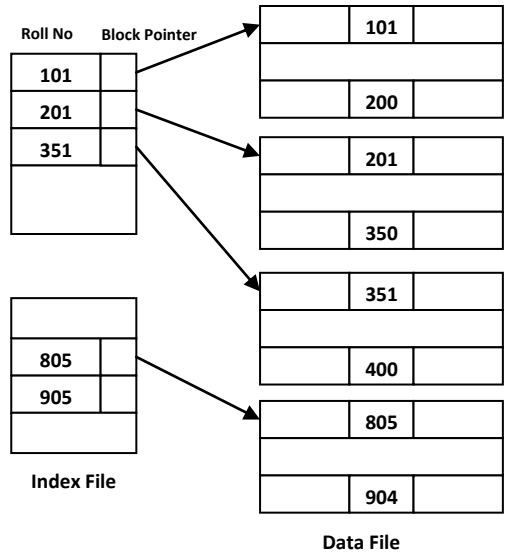
- With indexing, new records can be added at the end of the main file. It will not require movement of records as in the case of sequential file. Updation of index file requires fewer block accesses compare to sequential file

Types of Indexes:

1. Primary indexes
2. Clustering indexes
3. Secondary indexes

Primary Indexes (Indexed Sequential File):

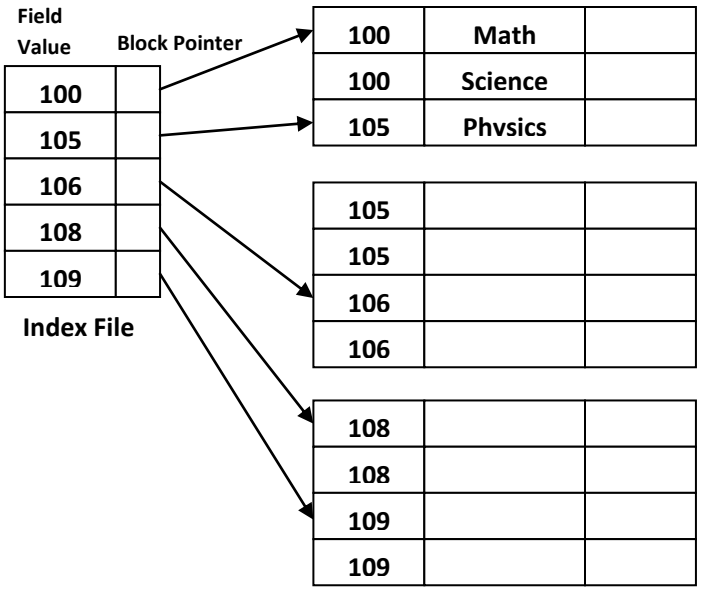
- An indexed sequential file is characterized by
    - Sequential organization (ordered on primary key)
    - Indexed on primary key
- An indexed sequential file is both ordered and indexed.
- Records are organized in sequence based on a key field, known as primary key.
- An index to the file is added to support random access. Each record in the index file consists of two fields: a key field, which is the same as the key field in the main file.
- Number of records in the index file is equal to the number of blocks in the main file (data file) and not equal to the number of records in the main file (data file).
- To create a primary index on the ordered file shown in the Fig. we use the rollno field as primary key. Each entry in the index file has rollno value and a block pointer. The first three index entries are as follows.
    - <101, address of block 1>
    - <201, address of block 2>
    - <351, address of block 3>
- Total number of entries in index is same as the number of disk blocks in the ordered data file.
- A binary search on the index file requires very few block accesses



Primary Index on ordering key field roll number

Clustering Indexes

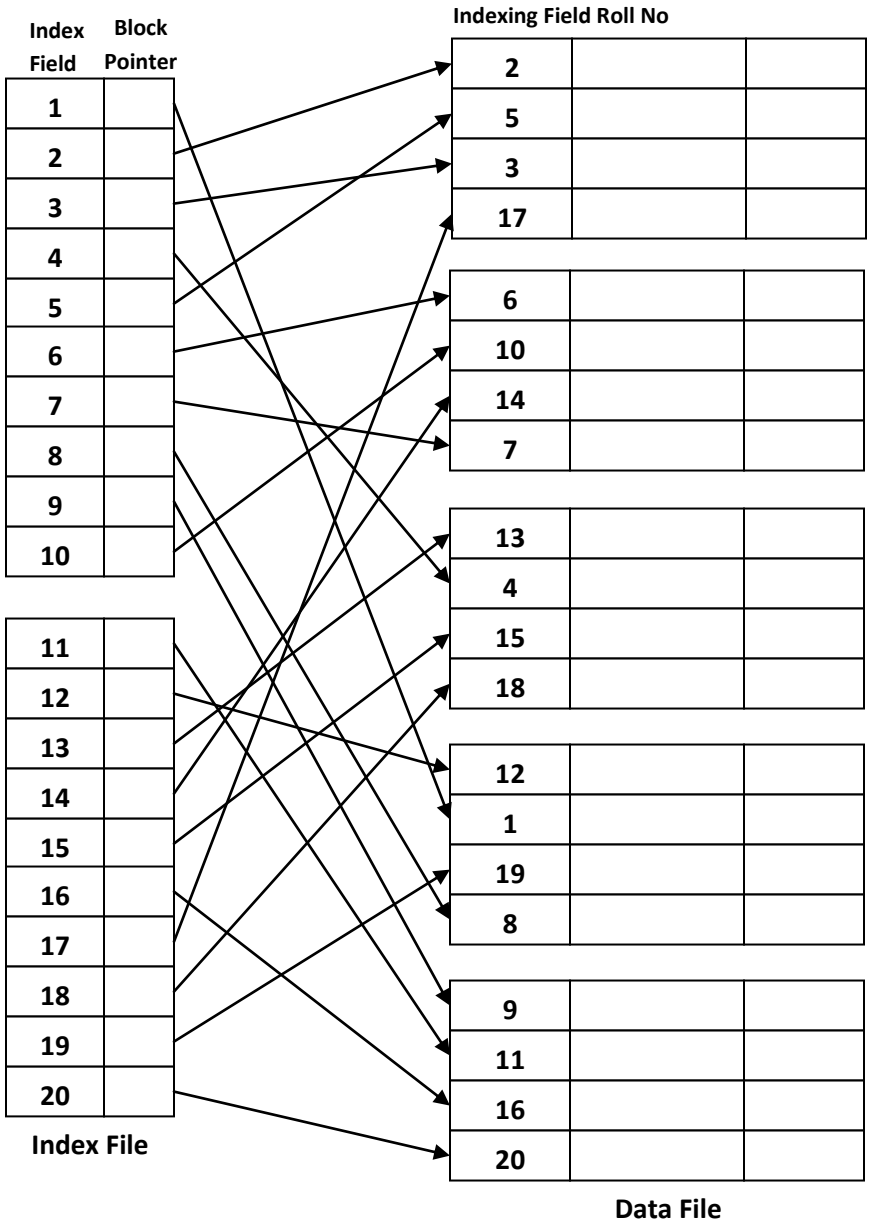
- If records of a file are ordered on a non-key field, we can create a different type of index known as clustering index.
- A non-key field does not have distinct value for each record.
- A Clustering index is also an ordered file with two fields.



Example of clustering index on roll no

Secondary indexes (Simple Index File)

- While the hashed, sequential and indexed sequential files are suitable for operations based on ordering key or the hashed key. Above file organizations are not suitable for operations involving a search on a field other than ordering or hashed key.
- If searching is required on various keys, secondary indexes on these fields must be maintained. A secondary index is an ordered file with two fields.
    - Some non-ordering field of the data file.
    - A block pointer
- There could be several secondary indexes for the same file.
- One could use binary search on index file as entries of the index file are ordered on secondary key field. Records of the data files are not ordered on secondary key field.
- A secondary index requires more storage space and longer search time than does a primary index.
- A secondary index file has an entry for every record whereas primary index file has an entry for every block in data file.
- There is a single primary index file but the number of secondary indexes could be quite a few.



A secondary index on a non-ordering key field

Bubble sort

- **Bubble sort**, sometimes referred as **sinking sort.**
- It is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- The algorithm gets its name from the way smaller elements "bubble" to the top of the list.
- As it only uses comparisons to operate on elements, it is a comparison sort.
- Although the algorithm is simple, it is too slow for practical use, even compared to insertion sort.

Algorithm

```
for i ← 1 to n do
        for j ← 1 to n-i do
                If Array[j] > Array[j+1] then       /* For decreasing order use < */
                        temp ← Array[j]
                        Array[j] ← A [j+1]
                        Array[j+1] ← temp
```

Program

```
#include <stdio.h>
void main()
{
        int array[100], n, i, j, temp;
        printf("Enter number of elements\n");
        scanf("%d", &n);
        printf("Enter %d integers\n", n);
        for (i = 0; I < n; i++)
        {
                scanf("%d", &array[i]);
        }
        for (i = 0 ;i< ( n - 1 );i++)
        {
                for (j = 0 ; j< n - c - 1; j++)
                {
                        if (array[j] > array[j+1])              /* For decreasing order use < */
                        {
```

```
                        temp      = array[j];
                        array[j]  = array[j+1];
                        array[j+1] = temp;
                    }
                }
        }
        printf("Sorted list in ascending order:\n");
        for (i = 0 ;i< n ;i++ )
        {
                printf("%d\n", array[i]);
        }
        getch();
}
```

Example

Consider an array A of 5 element

A[0]	45
A[1]	34
A[2]	56
A[3]	23
A[4]	12

Pass-1: The comparisons for pass-1 are as follows.

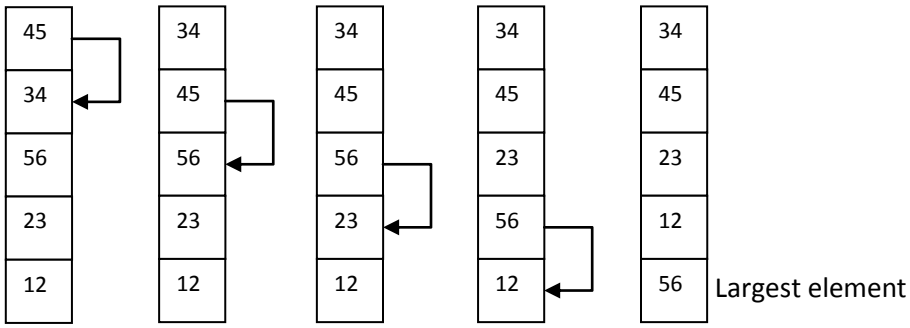
Compare A[0] and A[1]. Since 45>34, interchange them.

Compare A[1] and A[2]. Since 45<56, no interchange.

Compare A[2] and A[3]. Since 56>23, interchange them.

Compare A[3] and A[4]. Since 56>12 interchange them.

At the end of first pass the largest element of the array, 56, is bubbled up to the last position in the array as shown.

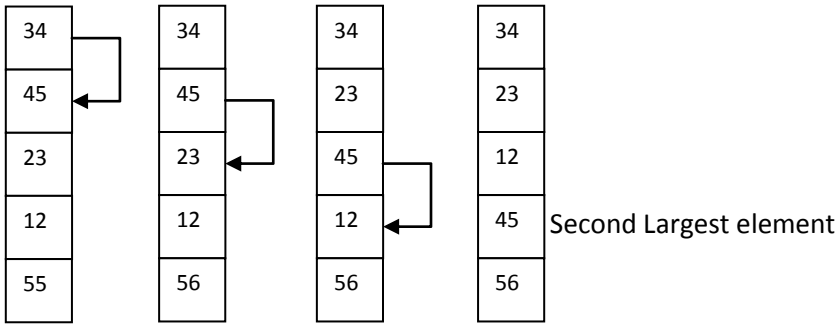


Pass-2: The comparisons for pass-2 are as follows.

Compare A[0] and A[1]. Since 34<45, no interchange.

Compare A[1] and A[2]. Since 45>23, interchange them.

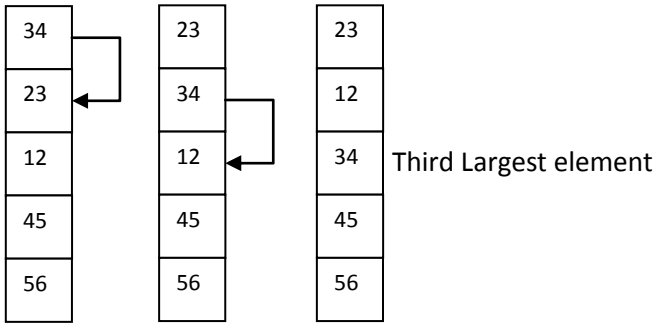
Compare A[2] and A[3]. Since 45>12, interchange them.



Pass-3: The comparisons for pass-3 are as follows.

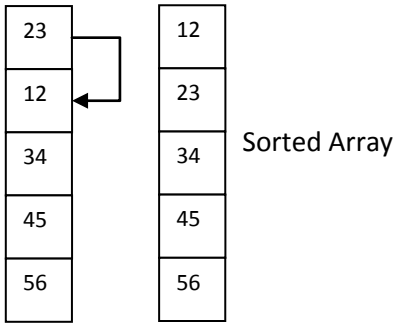
Compare A[0] and A[1]. Since 34>23, interchange them.

Compare A[1] and A[2]. Since 34>12, interchange them.



Pass-4: The comparisons for pass-4 are as follows.

Compare A[0] and A[1]. Since 23>12, interchange them.



Selection Sort

- The idea of algorithm is quite simple.
- Array is imaginary divided into two parts - sorted one and unsorted one.
- At the beginning, sorted part is empty, while unsorted one contains whole array.
- At every step, algorithm finds minimal element in the unsorted part and adds it to the end of the sorted one.
- When unsorted part becomes empty, algorithm stops.

Algorithm

SELECTION_SORT (A)

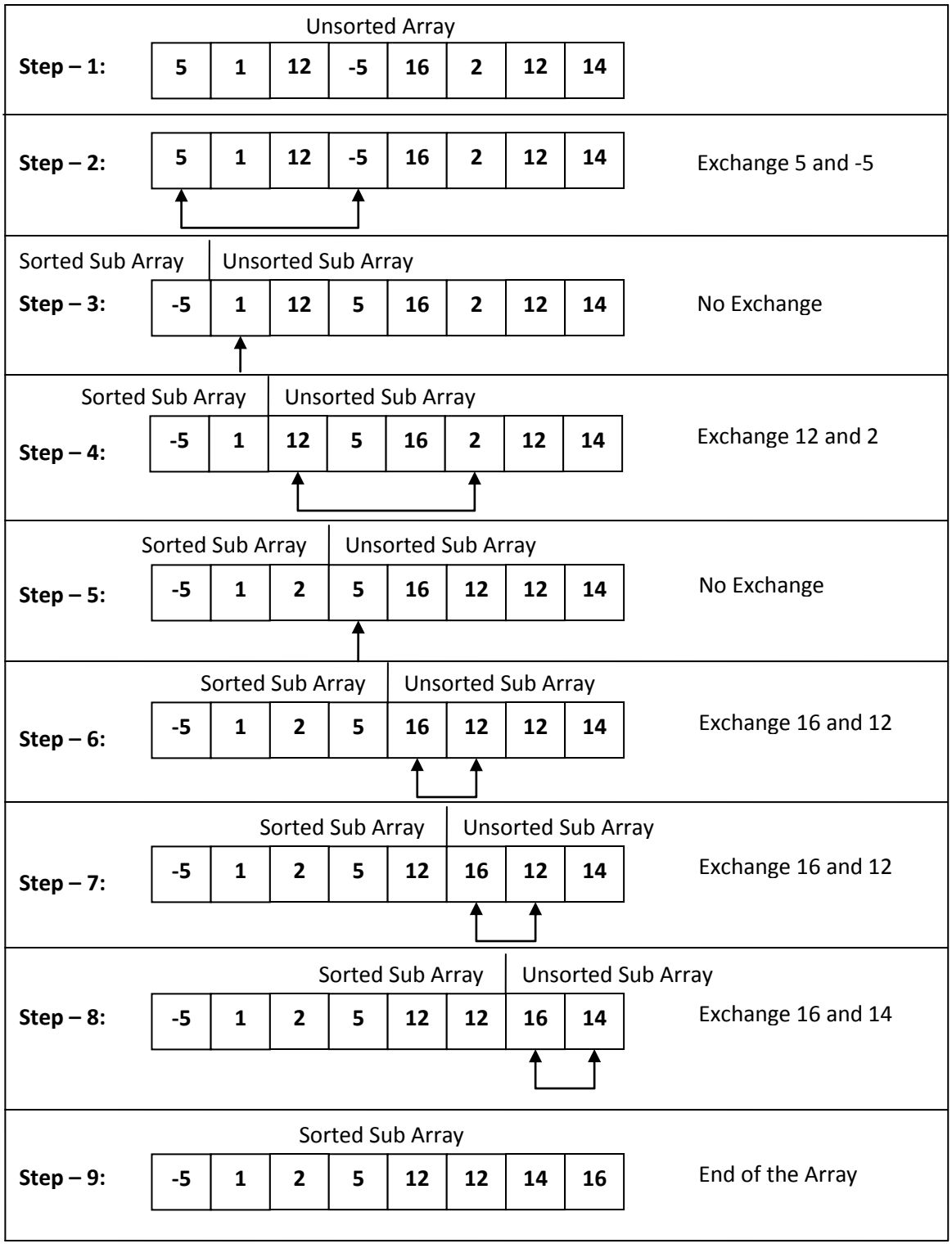
```
for i ← 1 to n-1 do
        min ← i;
        for j ← i + 1 to n do
                If A[j] < A[i] then
                        min ← j
        If min!=i then
                temp ← A[i]
                A[i] ← A [min]
                 A[min] ← temp
```

Program

```
#include <stdio.h>
void main()
{
        int array[100], n, i, j, min, temp;
        printf("Enter number of elements\n");
        scanf("%d", &n);
        printf("Enter %d integers\n", n);
        for ( i = 0 ; i < n ; i++ )
        {
                scanf("%d", &array[i]);
        }
        for ( i = 0 ; i < ( n - 1 ) ; i++ )
        {
                min = i;
                for ( j = i + 1 ; j < n ; j++ )
                {
                        if ( array[min] > array[j] )
                        min = j;
                }
```

```
            if ( min != i )
            {
                  temp = array[i];
                  array[i] = array[min];
                  array[min] = temp;
            }
      }
      printf("Sorted list in ascending order:\n");
      for ( i = 0 ; i < n ; i++ )
      {
            printf("%d\n", array[i]);
      }
      getch();
}
```

Example


Unsorted Array
Step – 1:
5 1 12 -5 16 2 12 14
Step – 2:
5 1 12 -5 16 2 12 14
Exchange 5 and -5
Sorted Sub Array
Unsorted Sub Array
Step – 3:
-5 1 12 5 16 2 12 14
No Exchange
Sorted Sub Array
Unsorted Sub Array
Step – 4:
-5 1 12 5 16 2 12 14
Exchange 12 and 2
Sorted Sub Array
Unsorted Sub Array
Step – 5:
-5 1 2 5 16 12 12 14
No Exchange
Sorted Sub Array
Unsorted Sub Array
Step – 6:
-5 1 2 5 16 12 12 14
Exchange 16 and 12
Sorted Sub Array
Unsorted Sub Array
Step – 7:
-5 1 2 5 12 16 12 14
Exchange 16 and 12
Sorted Sub Array
Unsorted Sub Array
Step – 8:
-5 1 2 5 12 12 16 14
Exchange 16 and 14
Sorted Sub Array
Step – 9:
-5 1 2 5 12 12 14 16
End of the Array

Quick Sort

- Quicksort is the currently fastest known sorting algorithm and is often the best practical choice for sorting, as its average expected running time is O(n log(n)).
- Pick an element, called a pivot, from the array.
- Reorder the array 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.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.
- Quicksort, like merge sort, is a divide-and-conquer recursive algorithm.
- The basic divide-and-conquer process for sorting a sub array A[$i..j$] is summarized in the following three easy steps:
  - **Divide:** Partition T[$i..j$] Into two sub arrays T[$i..l$-1] and T[l+1… j] such that each element of T[$i..l$-1] is less than or equal to T[l], which is, in turn, less than or equal to each element of T[l+1… j]. Compute the index l as part of this partitioning procedure
  - **Conquer:** Sort the two sub arrays T[$i..l$-1] and T[l+1… j] by recursive calls to quicksort.
  - **Combine:** Since the sub arrays are sorted in place, no work is needed to combing them: the entire array T[$i..j$] is now sorted.

Algorithm

Procedure *pivot* (T [$i\ldots j$]; ***var*** l)
{Permutes the elements in array T [$i\ldots j$] and returns a value l such that, at the end, $i<=l<=j$, T[k] <=P for all $i \le k < l$, T[l] =P, and T[k] > P for all $l < k \le j$, where P is the initial value T[i]}
P ← T[i]
$K \leftarrow i$; $l \leftarrow j$+1
Repeat $k \leftarrow k$+1 **until** T[k] > P
Repeat $l \leftarrow l$-1 **until** T[l] ≤ P
While $k < l$ **do**
    Swap T[k] and T[l]
    Repeat $k \leftarrow k$+1 **until** T[k] > P
    Repeat $l \leftarrow l$-1 **until** T[l] ≤ P
Swap T[i] and T[l]

Procedure *quicksort* (T [$i\ldots j$])
{Sorts sub array T [$i\ldots j$] into non decreasing order}
if $j - i$ is sufficiently small **then** insert (T[$i,\ldots,j$])
else
 pivot (T[$i,\ldots,j$],l)
 quicksort (T[$i,\ldots, l$ - 1])
 quicksort (T[l+1,…,j]

Program

```
#include<stdio.h>
void quicksort(int [10],int,int);
int partition(int [10],int, int);
void main()
{
        int x[20],size,i;

        printf("Enter size of the array: ");
        scanf("%d",&size);

        printf("Enter %d elements: ",size);
        for(i=0;i<size;i++)
        {
                scanf("%d",&x[i]);
        }

        quicksort(x,0,size-1);

        printf("Sorted elements: ");
        for(i=0;i<size;i++)
        {
                printf(" %d",x[i]);
        }
        getch();
}

void quicksort(int x[10],int first,int last)
{
        Int mid;
        if(first<last)
        {
                mid= partition(int x,int first,int last)
                quicksort(x,first,mid-1);
                quicksort(x,mid+1,last);
        }
}
int partition(int x[10],int p,int r)
{
        int value, i, j, temp;
```

```
        value=x[r];
        i=p-1;
        for(j=p;j<=r-1;j++)
        {
                If(x[j] ≤value)
                {
                        i=i+1;
                        temp=x[i];
                        x[i]=x[j];
                        x[j]=temp;
                }
        }
        temp=x[i+1];
        x[i]=x[r];
        x[r]=temp;

        Return (i+1);
}
```

Example

Sort given array using Quick Sort:

2	8	7	1	3	5	6	4

(a)

i	P , j						r
2	8	7	1	3	5	6	4

(b)

P , i	j						r
2	8	7	1	3	5	6	4

(c)

P , i		j					r
2	8	7	1	3	5	6	4

(d)

P , i			j				r
2	8	7	1	3	5	6	4

Exchange 8 and 1

(e)

P	i			j			r
2	1	7	8	3	5	6	4

Exchange 7 and 3

(f)

P		i			j		r
2	1	3	8	7	5	6	4

(g)

P		i				j	r
2	1	3	8	7	5	6	4

(h)

P		i					r
2	1	3	8	7	5	6	4

Exchange 8 and 4

(i)

P		i					r
2	1	3	4	7	5	6	8

2	1	3

Left Sub Array

4

7	5	6	8

Right Sub Array

Apply same method for left and right sub array finally we will get sorted

Merge Sort

- The merge sort algorithm is based on the classical divide-and-conquer paradigm. It operates as follows:
  - **DIVIDE:** Partition the n-element sequence to be sorted into two subsequences of n/2 elements each.
  - **CONQUER:** Sort the two subsequences recursively using the merge sort.
  - **COMBINE:** Merge the two sorted subsequences of size n/2 each to produce the sorted sequence consisting of n elements.
- Note that recursion "**bottoms out**" when the sequence to be sorted is of unit length.
- Since every sequence of length 1 is in sorted order, no further recursive call is necessary.
- The key operation of the merge sort algorithm is the merging of the two sorted sub sequences in the "combine step".
- To perform the merging, we use an auxiliary procedure Merge (A,p,q,r), where A is an array and p,q and r are indices numbering elements of the array such that procedure assumes that the sub arrays A[p..q] and A[q+1...r] are in sorted order.
- It merges them to form a single sorted sub array that replaces the current sub array A[p..r]. Thus finally, we obtain the sorted array A[1..n], which is the solution.

Algorithm

MERGE (*A,p,q,r*)

```
n1 = q  -p + 1
n2 = r – q
let L[1…n1+1] and R[1…n2+1] be new arrays
for i = 1 to n1
        L[i] = A[p+i-1]
for j = 1 to n2
        R[j] = A[q+j]
L[n1+1] = infinite
R[n2+1]= infinite
i=1
j=1
for k = p to r
        if L[i] ≤ R[j]
                A[k]=L[i]
                i = i +1
        else A[k] = R[j]
                j = j + 1
```

MERGE SORT (*A,p,r*)

```
if p < r
        then q<-- [ (p + r) / 2 ]
        MERGE SORT(A,p,q)
        MERGER SORT(A,q + 1,r)
        MERGE(A,p,q,r)
```

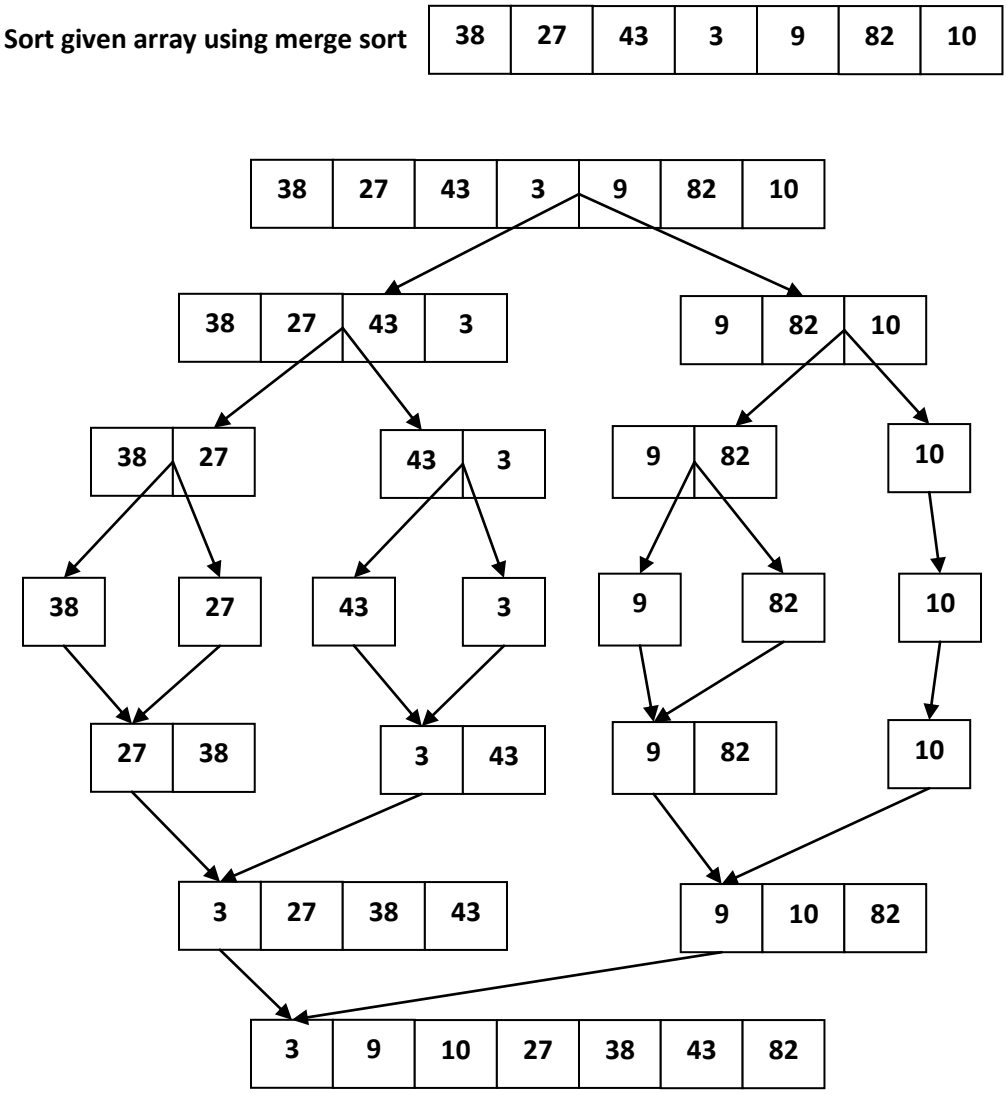
Program

```
#include<stdio.h>
void mergesort(int [20],int,int);
int merge(int [20],int, int, int);
void main()
{
        int x[20],size,i;

        printf("Enter size of the array: ");
        scanf("%d",&size);

        printf("Enter %d elements: ",size);
        for(i=0;i<size;i++)
        {
                scanf("%d",&x[i]);
        }

        mergesort(x,0,size-1);

        printf("Sorted elements: ");
        for(i=0;i<size;i++)
        {
                printf(" %d",x[i]);
        }
        getch();
}

void mergesort(int x[20],int p,int r)
{
        Int q;
        if(p<r)
```

```
        {
                q=(p+r)/2;
                mergesort(x,p,q);
                mergesort(x,q+1,r);
                merge(x,p,q,r)

        }
}
int merge(int x[20],int p,int q,int r)
{
        Int n1,n2,L[20],R[20],i,j,k;
        n1 = q  -p + 1;
        n2 = r – q;
        for(i=1; i<=n1;i++)
        {
                L[i]=x[p+i-1];
        }
        for(j=1; i<=n2;j++)
        {
                R[j]=x[q+j];
        }
        L[n1+1]=NULL;
        L[n2+1]=NULL;
        I=1;
        J=1;
        For(k=p;k<=r;k++)
        {
                If(L[i]<=R[j])
                {
                        X[k]=L[i];
                        I++;
                }
                Else
                {
                        x[k] = R[j];
                        j++;
                }
        }
}
```

Example

Sort given array using merge sort



Linear/Sequential Search

- In computer science, linear search or sequential search is a method for finding a particular value in a list that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.
- Linear search is the simplest search algorithm.
- It is a special case of brute-force search. Its worst case cost is proportional to the number of elements in the list.

Algorithm

```
# Input: Array A, integer key
# Output: first index of key in A,
# or -1 if not found

Algorith: Linear_Search
for i = 0 to last index of A:
        if A[i] equals key:
                return i
return -1
```

Program

```
#include <stdio.h>
void main()
{
        int array[100], key, i, n;

        printf("Enter the number of elements in array\n");
        scanf("%d",&n);

        printf("Enter %d integer(s)\n", n);

        for (i = 0; i < n; i++)
        {
                printf("Array[%d]=", i);
                scanf("%d", &array[i]);
        }

        printf("Enter the number to search\n");
        scanf("%d", &key);
```

```
        for (i = 0; i < n; i++)
        {
                if (array[i] == key)    /* if required element found */
                {
                        printf("%d is present at location %d.\n", key, i+1);
                        break;
                }
         }
        if (i == n)
        {
                printf("%d is not present in array.\n", search);
        }
        getch();
}
```

Example

Search for 1 in given array:

2	9	3	1	8

Comparing value of i^{th} index with element to be search one by one until we get seache element or end of the array

(a)

2	9	3	1	8

i

(b)

2	9	3	1	8

i

(c)

2	9	3	1	8

i

(d)

2	9	3	1	8

i

Element found at i^{th} index

Binary Search

- If we have an array that is sorted, we can use a much more efficient algorithm called a Binary Search.
- In binary search each time we divide array into two equal half and compare middle element with search element.
- If middle element is equal to search element then we got that element and return that index otherwise if middle element is less than search element we look right part of array and if middle element is greater than search element we look left part of array.

Algorithm

```
# Input: Sorted Array A, integer key
# Output: first index of key in A, or -1 if not found

Algorith: Binary_Search (A, left, right)
while left <= right
        middle = index halfway between left, right
        if D[middle] matches key
                return middle
        else if key less than A[middle]
                right = middle -1
        else
                left = middle + 1
return -1
```

Program

```
#include <stdio.h>
void main()
{
        int i, first, last, middle, n, key, array[100];

        printf("Enter number of elements\n");
        scanf("%d",&n);
        printf("Enter %d integers in sorted order\n", n);
        for ( i = 0 ; i < n ; i++ )
        {
                scanf("%d",&array[i]);
        }
        printf("Enter value to find\n");
        scanf("%d",&key);
```

```
        first = 0;
        last = n - 1;
        middle = (first+last)/2;

         while( first <= last )
        {
                if (array[middle] == key)
                {
                        printf("%d found at location %d.\n", key, middle+1);
                        break;
                }
                else if ( array[middle]>key )
                 {
                        Last=middle - 1;
                }
                else
                        first = middle + 1;

                middle = (first + last)/2;
         }
        if ( first > last )
        {
                printf("Not found! %d is not present in the list.\n", key);
        }
        getch();
}
```

Example

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 --> (middle element is 19 > 6): Search in left part

-1 5 6 18 **19** 25 46 78 102 114

Step 2 --> (middle element is 5 < 6): Search in Right part

-1 **5** 6 18

Step 3 --> (middle element is 6 == 6): Element Found

6 18