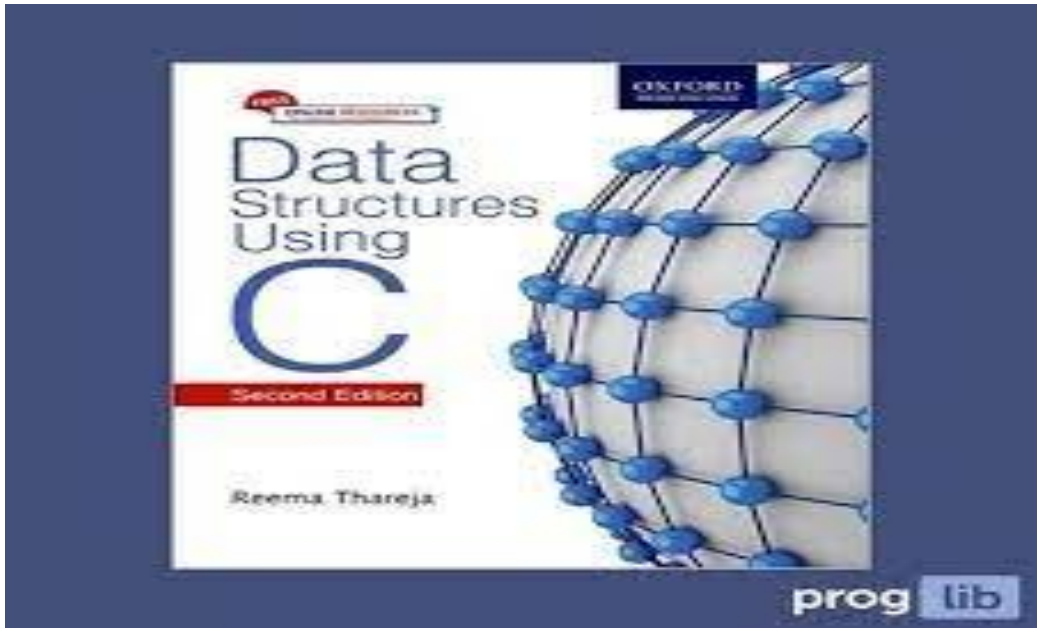
Text Books to Prescribed



UNIT-1

UNIT-1:

Data Structures: Definition, Classification of Data Structures, Operations on Data Structures, Abstract Data Type (ADT), Preliminaries of algorithms. Time and Space complexity. Searching- Linear search, Binary search, Fibonacci search. Sorting- Insertion sort, Selection sort, Exchange(Bubble sort, quick sort), distribution (radix sort),merging (Merge sort) algorithms**.**

DATA STRUCTURES :

- DEFINITION
- CLASSIFICATIONS
- OPERATIONS
- ABSTRACT DATA TYPES(ADT)
- PRELIMINARIES OF ALGORITHMS
- TIME and SPACE COMPLEXITY

SEARCHINGS:

- LINEAR SEACH
- BINARY SEARCH
- FIBONACCI SEARCH

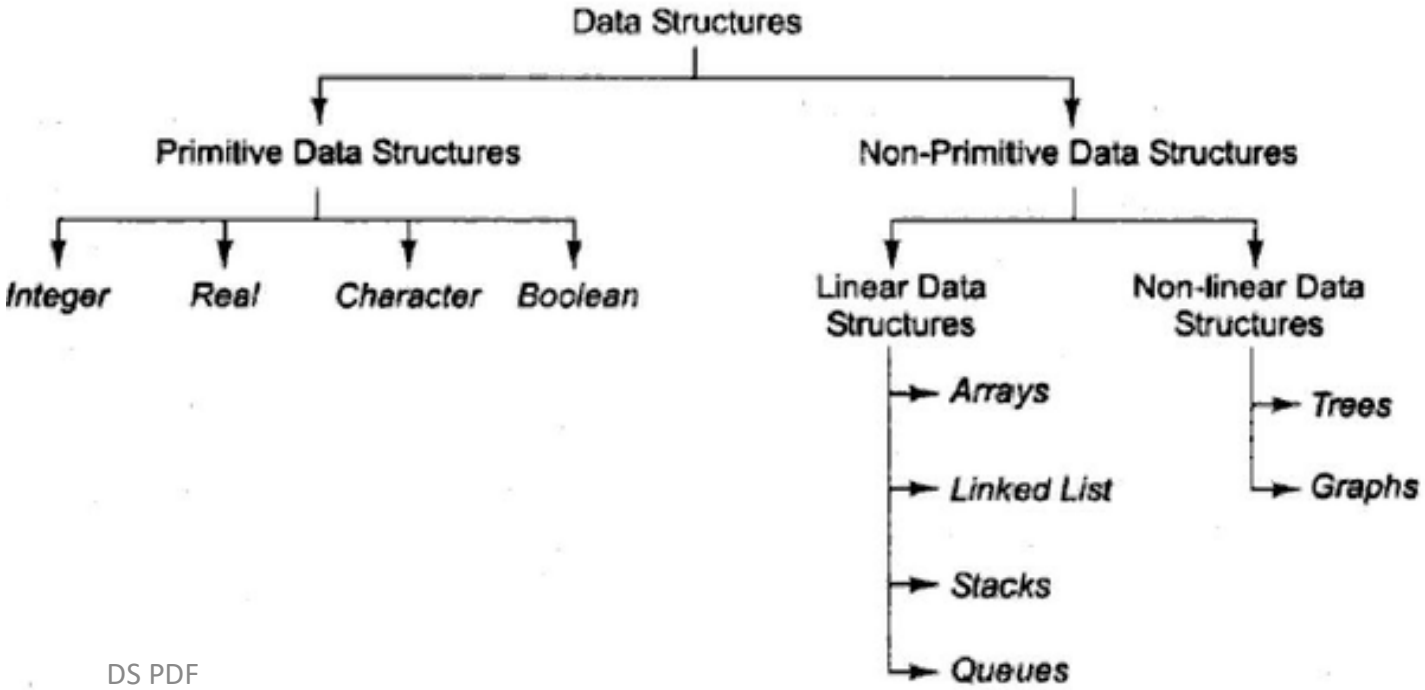
SORTINGS:

- INSERTION SORT, SELECTION SORT
- EXCHANGE LIKE BUBBLE SORT,QUICK SORT
- DISTRIBUTION LIKE RADIX SORT
- MERGING LIKE MERGE SORT ALGORITHMS.

UNIT-1

Data structure: Data may be organized in many ways, the logical or mathematical model of a particular organization of data is called data structure.

- Data structure deals with the study of
  1. How the data is organized in computer memory
  2. How effectively the data can be retrieved and manipulated.
- Data structures are classified into
  1. Primitive data structure
  2. Non primitive data structure



UNIT-1

- **Primitive data structures** are the fundamental data types which are supported by a programming language.
- Ex: integer, real, character, and Boolean.
- **Non-primitive data structures** are those data structures which are created using primitive data structures.
- EX: stacks, queues, linked lists, trees, and graphs.
- Non-primitive data structures can further be classified into two categories:
  1. linear data structures
  2. non-linear data structures.

1.linear data structures : If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure.

Ex: arrays, stacks, queues and linked lists.

UNIT-1

2.Non-linear data structures:

- If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.

  Ex: trees and graphs.

OPERATIONS ON DATA STRUCTURES

The different operations that can be performed on the various data structures are:

1. **Inserting** -It is used to add new data items to the given list of data items.

2. **Deleting-** It means to remove (delete) a particular data item from the given collection of data Items.

3. **Traversing -**It means to access each data item exactly once so that it can be processed.

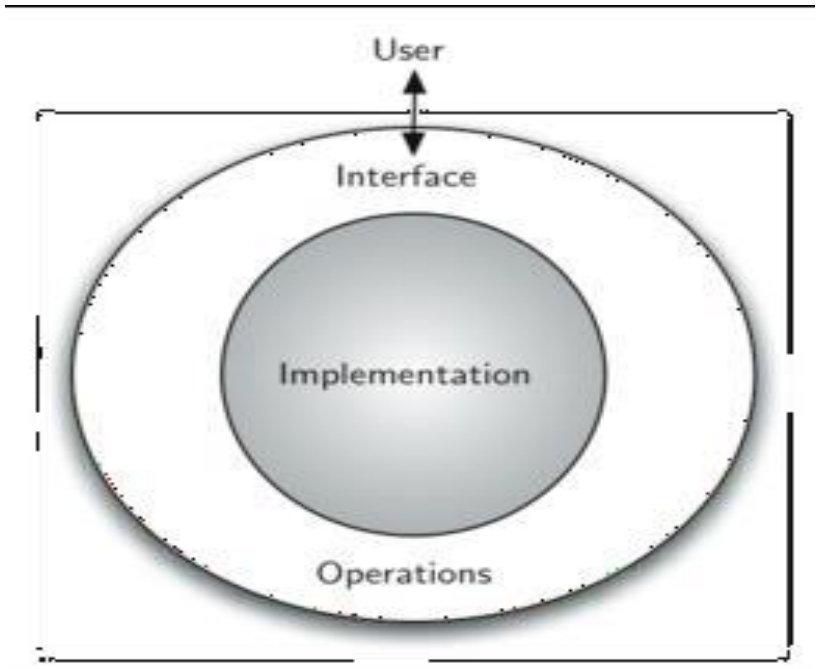
4. **Searching-** It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items.

5. **Sorting -**Data items can be arranged in some order like ascending order or descending order depending on the type of application.

6. **Merging-** Lists of two sorted data items can be combined to form a single list of sorted data items.

* Abstract Data Type:

An abstract data type, sometimes abbreviated ADT, is a logical description of how we view the data and the operations that are allowed without regard to how they will be implemented. The type is defined in terms of its data items and associative operations, not its implementation. The program language that supports ADT is c, c++, java and python. The implementation of an abstract data type often referred to as a data structure, will require that we provide a physical view of the data using some collection of programming constructs and primitive data types.



An algorithm is endowed with the following properties:

1. **Finiteness:** An algorithm must terminate after a finite number of steps.
2. **Definiteness:** The steps of the algorithm must be precisely defined or unambiguously specified.
3. **Generality:** An algorithm must be generic enough to solve all problems of a particular class.
4. **Effectiveness:** the operations of the algorithm must be basic enough to be put down on pencil and paper. They should not be too complex to warrant writing another algorithm for the operation.
5. **Input-Output:** The algorithm must have certain initial and precise inputs, and outputs that may be generated both at its intermediate and final steps.

Different Approaches to Design an (complexity of)Algorithm:

An algorithm does not enforce a language or mode for its expression but only demands adherence to its properties.

Practical Algorithm Design Issues:

1. **To save time (Time Complexity):** A program that runs faster is a better program.
2. **To save space (Space Complexity):** A program that saves space over a competing program is Considerable,

Efficiency of Algorithms:

The performances of algorithms can be measured on the scales of **time** and **space**. The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical and the other is experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity: The time complexity of an algorithm or a program is a function of the running time of the algorithm or a program. In other words, it is the amount of computer time it needs to run to completion.

Space Complexity: The space complexity of an algorithm or program is a function of the space needed by the algorithm or program to run to completion.

The time complexity of an algorithm can be computed either by an **empirical** or **theoretical** approach. The **empirical** or **posteriori testing** approach calls for implementing the complete algorithms and executing them on a computer for various instances of the problem. The time taken by the execution of the programs for various instances of the problem are noted and compared. The algorithm whose implementation yields the least time is considered as the best among the candidate algorithmic solutions.

STACKS

- A stack is a linear data structure.
- The elements in a stack are added and removed only from one end, which is called the TOP.
- Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

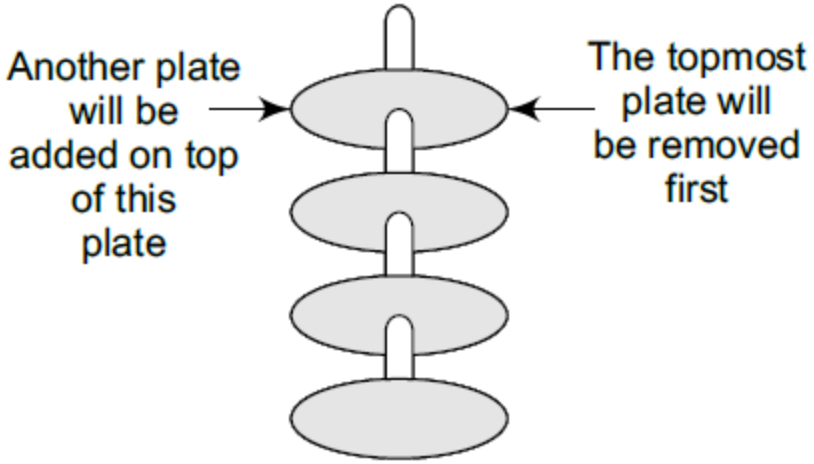


Figure 7.1 Stack of plates

STACKS

STACK ABSTRACT DATA TYPE:

A stack supports three basic operations

1. **Push Operation:**The push operation is used to insert an element into the stack.

2. **Pop Operation:**The pop operation is used to delete the topmost element from the stack.

3 .**Peek Operation :**Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.

STACKS

IMPLEMENTATION\REPRESENTATION OF STACKS:

Stacks can be represented using

1.Array

2.Linked list

1.Array representation of stacks:

- In the computer's memory, stacks can be represented as a linear array.
- Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from.
- There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.
- If TOP = NULL, then it indicates that the stack is empty.
- if TOP = MAX–1, then the stack is full.(You must be wondering why we have written MAX–1. It is because array indices start from 0.)

STACKS

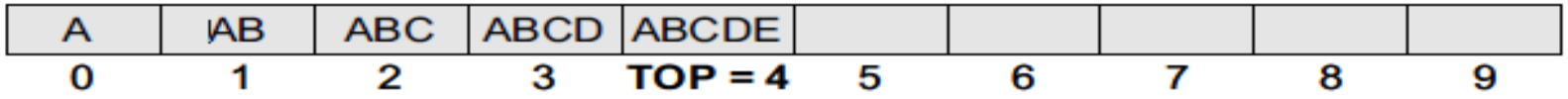


Figure 7.4 Stack

The stack in Fig. 7.4 shows that TOP = 4, so insertions and deletions will be done at this position. In the above stack, five more elements can still be stored.

STACKS

OPERATIONS ON A STACK:

1 Push Operation:

- The push operation is used to insert an element into the stack.
- The new element is added at the topmost position of the stack.
- Before inserting the value, we must first check if TOP=MAX–1, because if that is the case, then the stack is full and no more insertions can be done.
- If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.
- Consider the stack given in Fig.

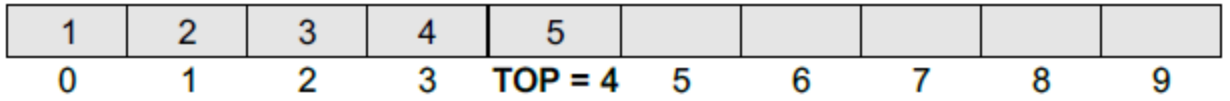


Figure 7.5 Stack

STACKS

- To insert an element with value 6, we first check if TOP=MAX–1. If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP].
- Thus, the updated stack becomes as shown in Fig. 7.6.

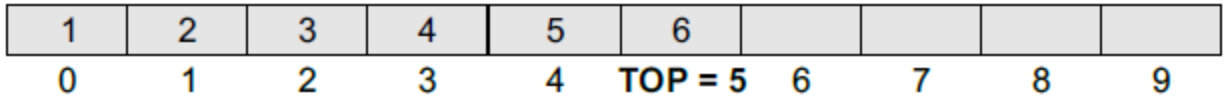


Figure 7.6 Stack after insertion

Algorithm to insert

Step 1: IF TOP = MAX-1
PRINT OVERFLOW
[END OF IF]
Goto step4
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END

STACKS

2 Pop Operation:

- The pop operation is used to delete the topmost element from the stack.
- However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done.
- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack given in Fig. 7.8.

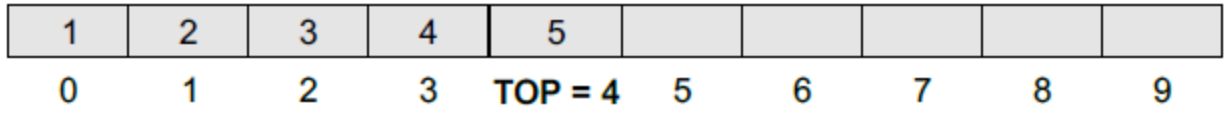


Figure 7.8 Stack

- To delete the topmost element, we first check if TOP=NULL. If the condition is false, then we decrement the value pointed by TOP. Thus, the updated stack becomes as shown in Fig. 7.9.

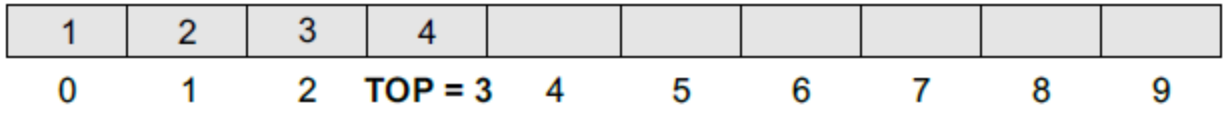


Figure 7.9 Stack after deletion

STACKS

Algorithm to delete an element from a stack:

Step 1: IF TOP = NULL

PRINT UNDERFLOW

[END OF IF]

Step 2: SET VAL = STACK[TOP]

Step 3: SET TOP = TOP - 1

Step 4: END

STACKS

3 Peek Operation:

- Peek is an operation that returns the value of the topmost element of the stack without deleting it from the stack.
- However, the Peek operation first checks if the stack is empty, i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.
- Consider the stack given in Fig. 7.12

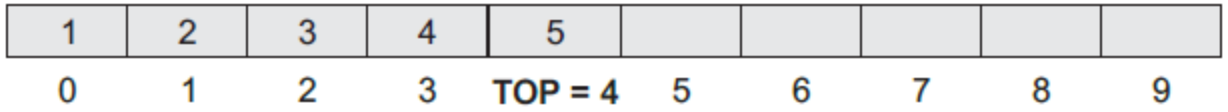


Figure 7.12 Stack

Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

Algorithm for Peek operation:

Step 1: IF TOP = NULL
PRINT STACK IS EMPTY
Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END

APPLICATIONS OF STACK

Evaluation of Arithmetic Expressions:

- Three different but equivalent notations of writing algebraic expressions.(polish notations)
- **1.** Infix
- 2. postfix
- 3. prefix notations

1. Infix notation: In an expression if the operator is placed in between the operands then such expressions are called infix expressions.

General form:

Operand1 operator operand2

Ex: a+b

2. Postfix notation: In an expression if the operator is placed after the operands then such expressions are called postfix expressions.

APPLICATIONS OF STACK

General form: Operand1 operand2 operator

Ex: ab+

2.Prefix notation: In an expression if the operator is placed before the operands then such expressions are called prefix expressions.

General form: operator Operand1 operand2

Ex: +ab

APPLICATIONS OF STACK

Conversion of infix to postfix expression:

RULES:

1.Priority of operators

^ exponent having highest priority

*,/,% next priority

+,- lowest priority

2.No two operators of same priority can stay together in the stack.If two operators have same priority before placing the operator we have to pop the existing operator and add the new operator to the stack.

3.Lowest priority cannot be placed before highest priority.

4.If we encounter (+) then pop the elements between the parenthesis and add those operators to prefix expression.

APPLICATIONS OF STACK

Algorithm:

Step1: scan the infix expression from left to right

Step2 :a:if the scanned symbol is left parenthesis,push it onto the stack.

b:if the sacnned symbol is an operand,then place directly in the postfix expression(output)

c:if the symbol scanned is right parenthesis,then go on popping the items from the stack and place them in the postfix expression till we get the matching left parenthesis.

Step3:if the symbol scanned is an operator,then go on removing all the operators from the stack and place them in the postfix expression,if and only if precedence of the operator which is on the top of the stack is greater than the precedence of the scanned operator and push the scanned operator onto the stack otherwise,push the scanned operator.

APPLICATIONS OF STACK

Ex: convert the infix expression **a+b** into postfix expression

Ex: convert the infix expression **x+(y*z)** into postfix expression

Ex: convert the infix expression **a-(b/c+(d%e*f)/g)*h** into postfix expression

Ex: convert the infix expression **a-(b/c+(d%e*f)/g*h)** into postfix expression

APPLICATIONS OF STACK

Evaluation of a postfix expression:

Step1:

start

Step2:

Every character of the postfix expression is scanned from left to right

Step3:

If the character encountered is an operand, it is pushed onto the stack.

Step4:

If the character encountered is operator, then the top 2 operands are popped from the stack and the operator is applied on these values and the result is pushed onto the stack.

Step5:

Repeat steps 3 and 4 until all the characters are scanned from the postfix expression

Step6:

Pop the final result from the stack as output

Step7:

Stop the process

APPLICATIONS OF STACK

Evaluation of a postfix expression:

Ex: evaluate the postfix expression **2 3* 5 1 / +**

Ex: evaluate the postfix expression **9 3 4 * 8 + 4 /-**

APPLICATIONS OF STACK

Conversion of infix to prefix expression:

- The precedence rules for converting infix to prefix is same as converting infix to postfix.
- The only change is that traverse the expression from right to left and the operator is placed before the operands rather than after them.

Ex: convert the infix expression **(a+b)*(c-d)** into prefix expression

APPLICATIONS OF STACK

Evaluation of a prefix expression:

- Ex: evaluate the prefix expression **+ - 2 7 * 8 / 4 12**

QUEUES

- queue is a data structure.
- A queue is a FIFO (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the REAR and removed from the other end called the FRONT.

QUEUES

Representation /implementation of queue:

Queues can be implemented by using either

1. arrays
2. linked lists

QUEUES

ARRAY REPRESENTATION OF QUEUEs:

- Queues can be easily represented using linear arrays.
- Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

Operations on Queues:

1.Inserting element into queue:

- However, before inserting an element in a queue, we mustvcheck for overflow conditions. An overflow will occur when we try to insert an element into a queue that is already full. When REAR = MAX – 1, where MAX is the size of the queue, we have an overflow condition.

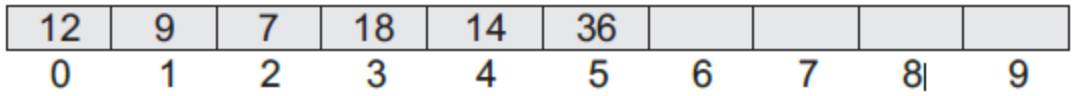


Figure 8.1 Queue

- In Fig. 8.1, FRONT = 0 and REAR = 5. Suppose we want to add another element with value 45, then REAR would be incremented by 1 and the value would be stored at the position pointed by REAR.
- The queue after addition would be as shown in

QUEUES

The queue after addition would be as shown in

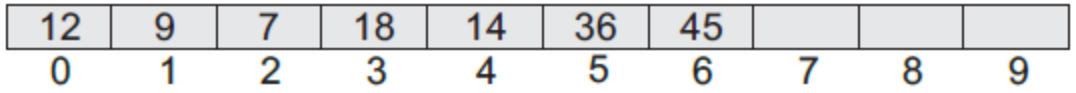


Figure 8.2 Queue after insertion of a new element

Here, FRONT = 0 and REAR = 6. Every time a new element has to be added, we repeat the same procedure.

QUEUES

Algorithm:

Step 1: IF REAR = MAX-1

Write OVERFLOW

Goto step 4

[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR

ELSE

SET REAR = REAR + 1

[END OF IF]

Step 3: SET QUEUE[REAR] = NUM

Step 4: EXIT

QUEUES

2.deleting element from queue:

- before deleting an element from a queue, we must check for underflow conditions. An underflow condition occurs when we try to delete an element from a queue that is already empty. If FRONT = –1 and REAR = –1, it means there is no element in the queue.
- If we want to delete an element from the queue, then the value of FRONT will be incremented.
- Deletions are done from only this end of the queue. The queue after deletion will be as shown in Fig

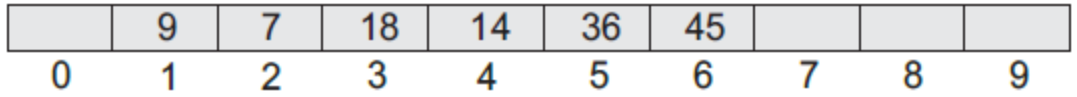


Figure 8.3 Queue after deletion of an element

- Here, FRONT = 1 and REAR = 6.

QUEUES

Algorithm:

Step 1: IF FRONT = -1 OR FRONT > REAR

Write UNDERFLOW

ELSE

SET FRONT = FRONT + 1

[END OF IF]

Step 2: EXIT

TYPES OF QUEUES: CIRCULAR QUEUES

- In linear queues, the insertions can be done only at one end called the REAR and deletions are always done from the other end called the FRONT. Look at the queue shown in Fig.

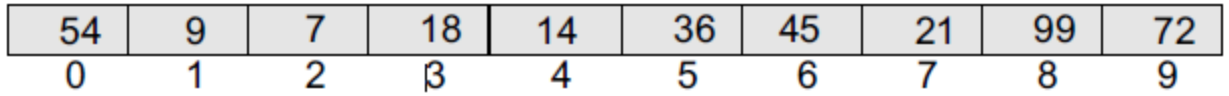


Figure 8.13 Linear queue

Here, FRONT = 0 and REAR = 9.

- Now, if you want to insert another value, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted. Consider a scenario in which two successive deletions are made. The queue will then be given as shown in Fig

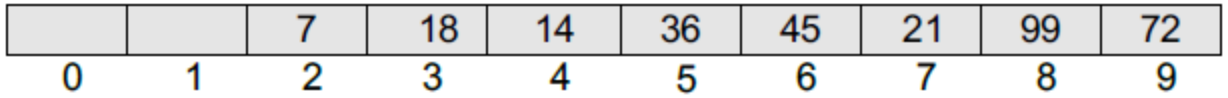


Figure 8.14 Queue after two successive deletions

Here, FRONT = 2 and REAR = 9.

TYPES OF QUEUES: CIRCULAR QUEUES

- Suppose we want to insert a new element in the queue shown in Fig. 8.14. Even though there is space available, the overflow condition still exists because the condition rear = MAX – 1 still holds true. This is a major drawback of a linear queue.
- To resolve this problem, we have two solutions.

1. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently. But

this can be very time-consuming, especially when the queue is quite large.

2. The second option is to use a circular queue.

TYPES OF QUEUES: CIRCULAR QUEUES

circular queue:

- In the circular queue, the first index comes right after the last index. Conceptually, you can think of a circular queue as shown in Fig. 8.15.

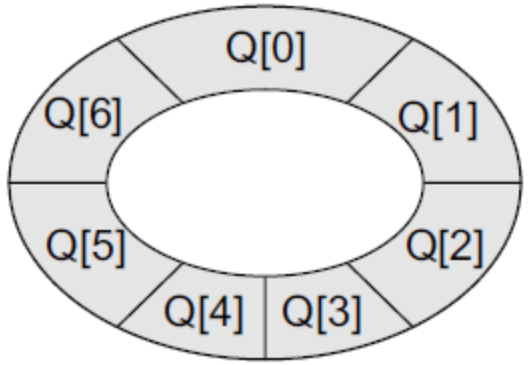


Figure 8.15 Circular queue

TYPES OF QUEUES: CIRCULAR QUEUES

- The circular queue will be full only when front = 0 and rear = Max – 1. A circular queue is implemented in the same manner as a linear queue is implemented.
- The only difference will be in the code that performs insertion and deletion operations.

Inserting an element in a circular queue:

- For insertion, we now have to check for the following three conditions:

1. If front = 0 and rear = MAX – 1, then the circular queue is full. Look at the queue given in Fig. 8.16 which illustrates this point.

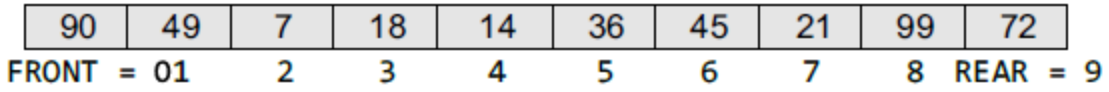


Figure 8.16 Full queue

TYPES OF QUEUES: CIRCULAR QUEUES

2. If rear != MAX – 1, then rear will be incremented and the value will be inserted as illustrated in Fig. 8.17.

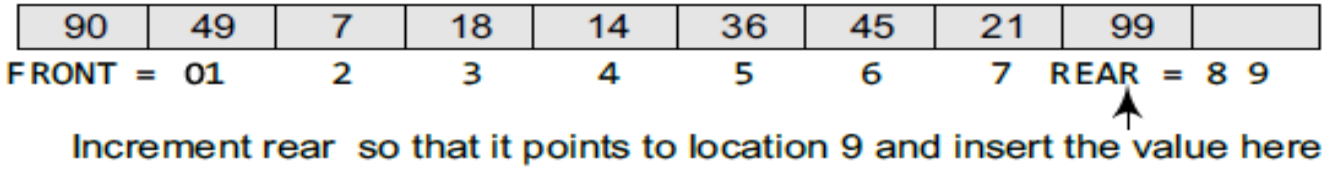


Figure 8.17 Queue with vacant locations

3. If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there, as shown in Fig. 8.18

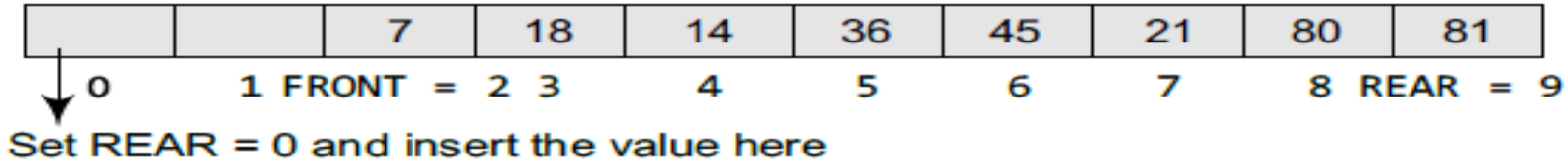


Figure 8.18 Inserting an element in a circular queue

TYPES OF QUEUES: CIRCULAR QUEUES

The algorithm to insert an element in a circular queue:

Step 1: IF FRONT = and Rear = MAX - 1

Write OVERFLOW

Step 2 : IF FRONT = -1 and REAR = -1

SET FRONT = REAR =0

ELSE IF REAR = MAX - 1 and FRONT !=0

SET REAR =0

ELSE

SET REAR = REAR + 1

[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

TYPES OF QUEUES: CIRCULAR QUEUES

Deleting an element from a circular queue:

- To delete an element, again we check for three conditions.

1. Look at Fig. 8.20. If front = –1, then there are no elements in the queue. So, an underflow condition will be reported.

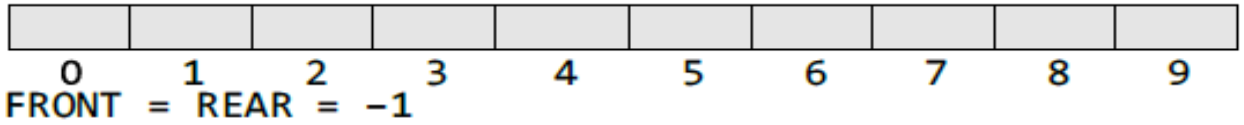


Figure 8.20 Empty queue

2. If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1. This is illustrated in Fig. 8.21.

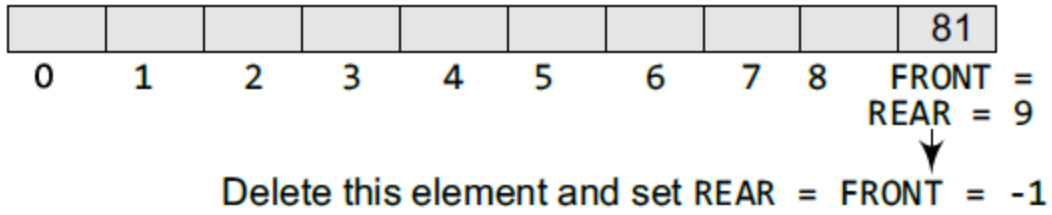


Figure 8.21 Queue with a single element

TYPES OF QUEUES: CIRCULAR QUEUES

3. If the queue is not empty and front = MAX–1,then after deleting the element at the front, front is set to 0. This is shown in Fig. 8.22.

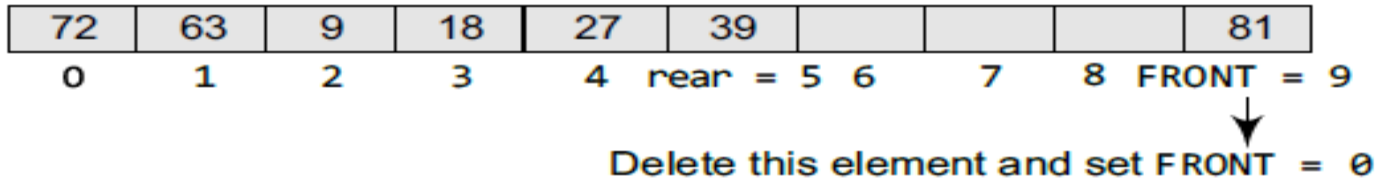


Figure 8.22 Queue where FRONT = MAX–1 before deletion

TYPES OF QUEUES: CIRCULAR QUEUES

Algorithm to delete an element from a circular queue:
Step 1: IF FRONT = -1
Write UNDERFLOW
Goto Step 4
[END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT =0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]
Step 4: EXIT

Tower of Hanoi

- Recursion is a technique that breaks a problem into one or more sub-problems that are similar to the original problem.
- The tower of Hanoi is one of the main applications of recursion. It says, 'if you can solve n–1 cases, then you can easily solve the nth case'.

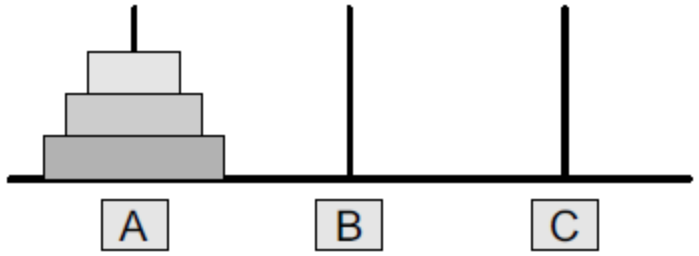


Figure 7.33 Tower of Hanoi

- Look at Fig. 7.33 which shows three rings mounted on pole A. The problem is to move all these rings from pole A to pole C while maintaining the same order.
- The main issue is that the smaller disk must always come above the larger disk.

Tower of Hanoi

- We will be doing this using a spare pole. In our case, A is the source pole, C is the destination pole, and B is the spare pole. To transfer all the three rings from A to C, we will first shift the upper two rings (n–1 rings) from the source pole to the spare pole. We move the first two rings from pole A to B as shown in Fig. 7.34.

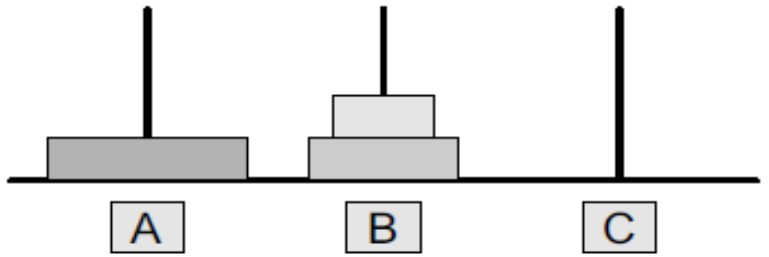


Figure 7.34 Move rings from A to B

Tower of Hanoi

- Now that n–1 rings have been removed from pole A, the nth ring can be easily moved from the source pole (A) to the destination pole (C). Figure 7.35 shows this step.

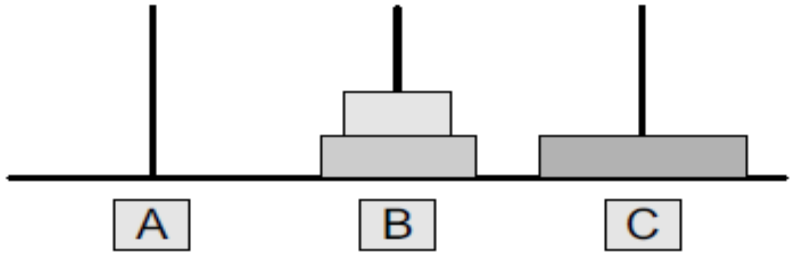


Figure 7.35 Move ring from A to C

- The final step is to move the n–1 rings from the spare pole (B) to the destination pole (C). This is shown in Fig. 7.36.

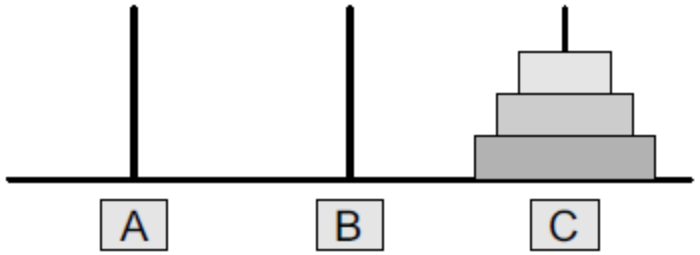


Figure 7.36 Move ring from B to C

Tower of Hanoi

- To summarize, the solution to our problem of moving n rings from A to C using B as spare can be given as:

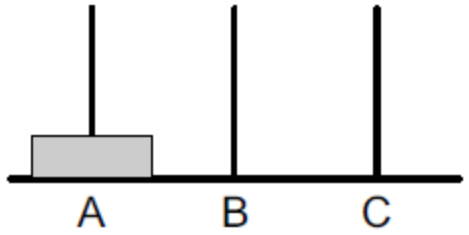
Base case: if n=1

- Move the ring from A to C using B as spare

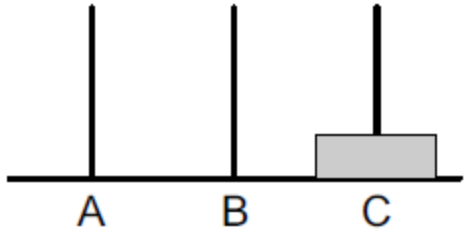
Recursive case:

- Move n – 1 rings from A to B using C as spare
- Move the one ring left on A to C using B as spare
- Move n – 1 rings from B to C using A as spare

Tower of Hanoi

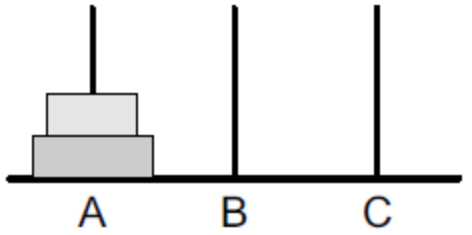


(Step 1)

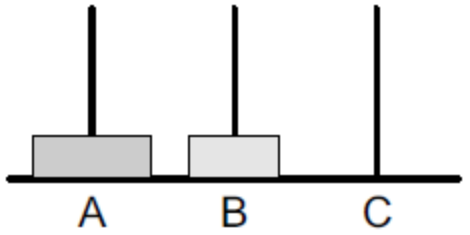


(Step 2)

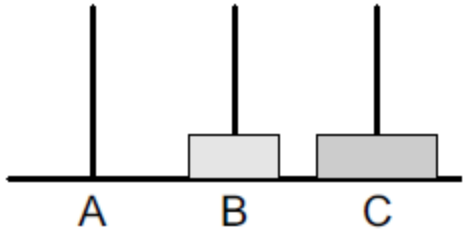
(If there is only one ring, then simply move the ring from source to the destination.)



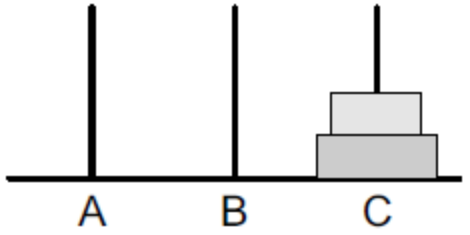
(Step 1)



(Step 2)



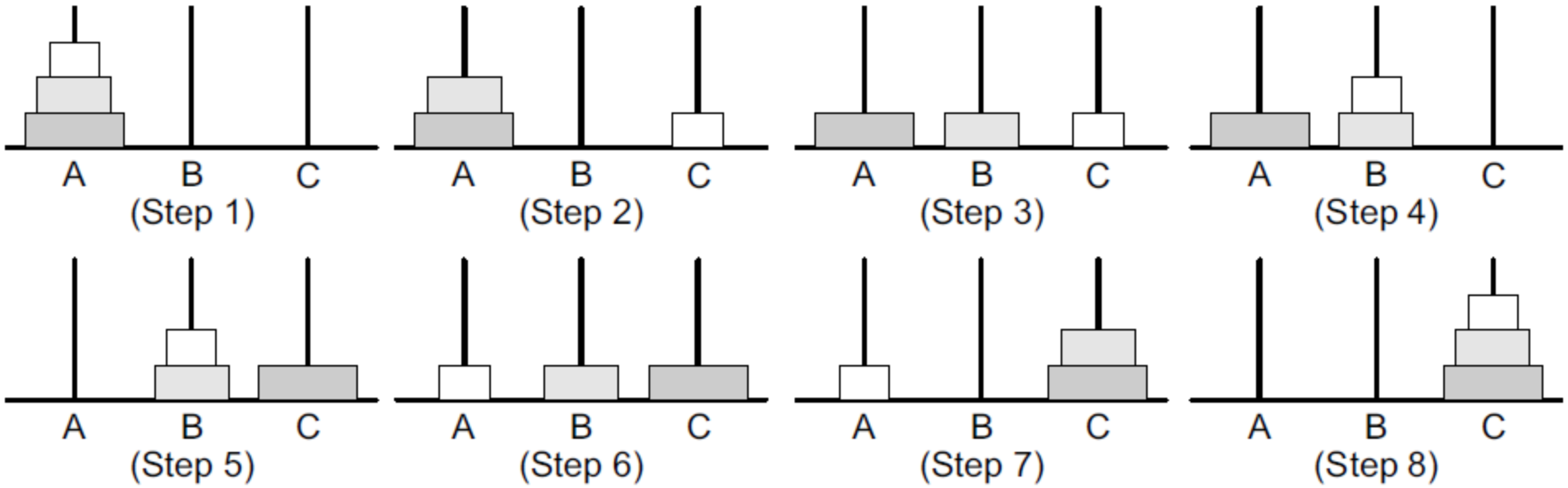
(Step 3)



(Step 4)

(If there are two rings, then first move ring 1 to the spare pole and then move ring 2 from source to the destination. Finally move ring 1 from spare to the destination.)

Tower of Hanoi



(Consider the working with three rings.)

Types of queue:dequeue

Dequeue(double ended queue):

- Dequeue is a type of queue in which insertion and deletion can be performed from either front or rear.
- It doesn't follow FIFO

Operations on dequeue:

1.Insertion at front

2.Insertion at rear

3.Deletion at front

4.Deletion at rear

There are 2 types of dequeus:

1.Input restricted dequeue

2.Output restricted dequeue

Types of queue:dequeue

1.Input restricted dequeue:

In this type of dequeue,

- Insertion can be performed only at rear end of queue
- Deletions can be performed from both front end and rear end of dequeue

2.Output restricted dequeue

In this type of dequeue,

- Insertion can be performed from both front end and rear end of dequeue
- Deletions can be performed from only front end of dequeue.

Types of queue:dequeue

Algorithm for inserting value at front of dequeue:

Step1:

If (front=0 && rear=MAX-1)|| front=rear+1

Over flow

Step2:

if front=-1 set front=rear=0

Else if front=0 set front=MAX-1

Else front=front-1

Step3:

Dequeue[front]=value

Types of queue:dequeue

Algorithm for inserting value at rear of dequeue:

Step1:

If (front=0 && rear=MAX-1)|| front=rear+1

Over flow

Step2:

if front=-1 set front=rear=0

Else if rear=Max-1 set rear=0

Else rear=rear+1

Step3:

Dequeue[rear]=value

Types of queue:dequeue

Algorithm for deleting value at front of dequeue:

Step1:

If front==-1

underflow

Step2:

if front=rear set front=-1 rear=-1

Else if front=MAX-1 set front=0

Else front=front+1

Types of queue:dequeue

Algorithm for deleting value at rear of dequeue:

Step1:

If front==-1

underflow

Step2:

if front=rear set front=-1 rear=-1

Else if rear=0 set rear=MAX-1

Else Rear=rear-1

Complexity of algorithms:Time and space

Complexity is a function which gives the running time or space in terms of input size.

Two types of complexities are

1. Time complexity
2. Space complexity

1.Time complexity:

Time complexity of a program is the amount of computation time it needs to completion.

2.Space complexity:

Space complexity of a program is the amount of computer space it occupies to store that program.

Complexity of algorithms:Time and space

There are 3 types of analysis:

1.best case: the minimum number of steps taken for an instance of size ‘n’

2.Average case: the average number of steps taken for an instance of size ‘n’

3.Worst case: the maximum number of steps taken for an instance of size ‘n’

Complexity of algorithms:Time complexity

1.Single statement:

C=a+b

Tc=O(1)

2.loops:

for i=1 to n ------n

b=b+a[i] -----1*n

Tc=n+n=2n=O(n)

3.Nested loops:

For i=1 to n -----n

for j=1 to n ------n*n

b=b+a[i] ------1*n*n

Tc= n+n^2+n^2

=n+2n^2

=O(n^2)

Complexity of algorithms:Time complexity

4.Consecutive statements:

for i=1 to n ------n

b=b+a[i] -----1*n

Tc=n+n=2n=O(n)

For i=1 to n -----n

for j=1 to n ------n*n

b=b+a[i] ------1*n*n

Tc= n+n^2+n^2

=n+2n^2

=O(n^2)

Tc=O(n^2)

Complexity of algorithms:Time complexity

5.if else statements:

tc=O(n)

6.While loop:

(I)While n>0

{

i=i+1

n=n/2

}

tc=O(log n)

(II) While n>0

{

i=i+1

N=n-1

}

Tc =O(n)

Complexity of algorithms:Time complexity

7.recursion:

Fact(n)

If n<=1 -----1

Return 1

Else

Return n*fact(n-1) ------n

Tc=n+1=O(n)

Complexity of algorithms:Time complexity

Time complexity in order of high efficiency to low efficiency (or) faster to lower execution speed

1. 1
2. Log n
3. N
4. N^2
5. N^3
6. 2^n
7. N!

UNIT-2

UNIT-2: LINKED LISTS

- POINTERS-POINTER ARRAYS
- LINKED LIST-NODE REPRESENTATION
- SINGLE LINKED LIST-TRAVERSING AND SEARCHING A SINGLE LINKED LIST-INSERTION INTO AND DELETION FROM A SINGLE LINKED LIST
- HEADER LINKED LIST
- CIRCULAR LINKED LISTS
- DOUBLY LINKED LIST
- LINKED STACKS AND QUEUES
- POLYNOMIALS-POLYNOMIAL REPRESENTATION
- SPARSE MATRICES

UNIT-2

Basic Terminologies

- A linked list, in simple terms, is a linear collection of data elements.
- These data elements are called nodes.
- Linked list is a data structure which in turn can be used to implement other datastructures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.
- A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node.

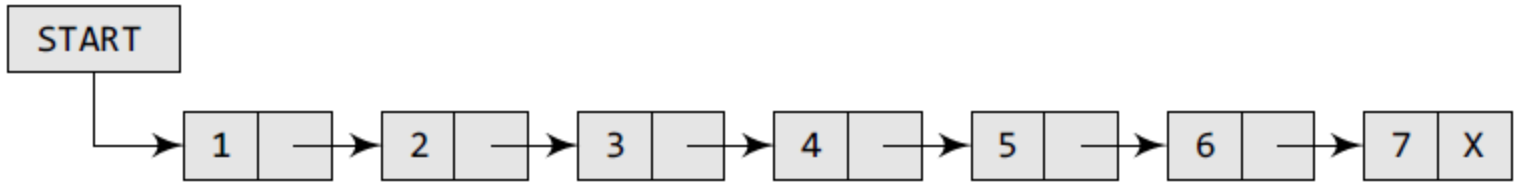


Figure 6.1 Simple linked list

UNIT-2

- In Fig. 6.1, we can see a linked list in which every node contains two parts, an integer and a pointer to the next node.
- The left part of the node which contains data may include a simple data type, an array, or a structure.
- The right part of the node contains a pointer to the next node (or address of the next node in sequence).
- The last node will have no next node connected to it, so it will store a special value called NULL. In Fig. 6.1, the NULL pointer is represented by X.
- While programming, we usually define NULL as –1. Hence, a NULL pointer denotes the end of the list.
- Since in a linked list, every node contains a pointer to another node which is of the same type, it is also called a self-referential data type.

UNIT-2

- Linked lists contain a pointer variable START that stores the address of the first node in the list. We can traverse the entire list using START which contains the address of the first node; the next part of the first node in turn stores the address of its succeeding node. Using this technique, the individual nodes of the list will form a chain of nodes. If START = NULL, then the linked list is empty and contains no nodes.

In C, we can implement a linked list using the following code:

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

UNIT-2

- Let us see how a linked list is maintained in the memory. In order to form a linked list, we need a structure called

node which has two fields, DATA and NEXT. DATA will store the information part and NEXT will store the address of the next node in sequence. Consider Fig. 6.2.

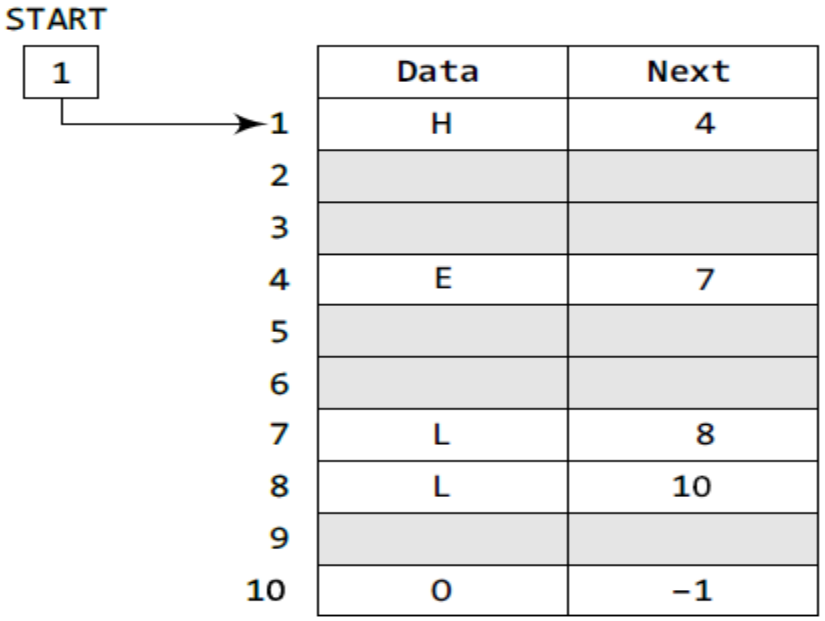


Figure 6.2 START pointing to the first element of the linked list in the memory

UNIT-2

- In the figure, we can see that the variable START is used to store the address of the first node. Here, in this example, START = 1, so the first data is stored at address 1, which is H.
- The corresponding NEXT stores the address of the next node, which is 4. So, we will look at address 4 to fetch the next data item.
- The second data element obtained from address 4 is E. Again, we see the corresponding NEXT to go to the next node. From the entry in the NEXT, we get the next address, that is 7, and fetch L as the data.
- We repeat this procedure until we reach a position where the NEXT entry contains –1 or NULL, as this would denote the end of the linked list.
- When we traverse DATA and NEXT in this manner, we finally see that the linked list in the above example stores characters that when put together form the word HELLO.

UNIT-2

There are different types of linked lists

1. **SINGLY LINKED Lists:**

- A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.
- By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence. A singly linked list allows traversal of data only in one way. Figure 6.7 shows a singly linked list.

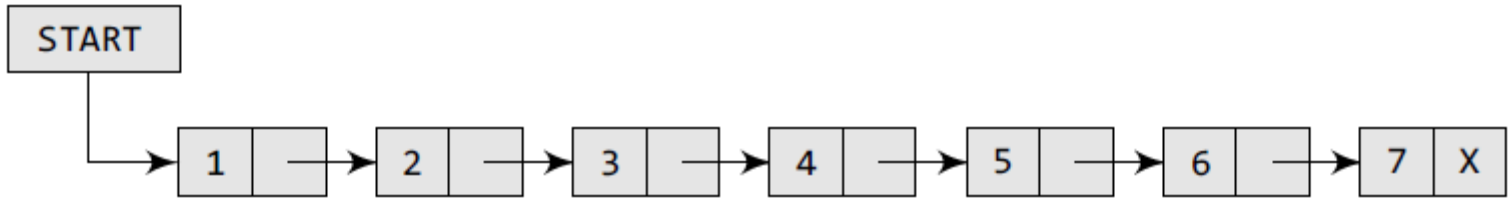


Figure 6.7 Singly linked list

UNIT-2

Algorithm for traversing a linked list:

Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: Apply Process to PTR →DATA
Step 4: SET PTR = PTR→ NEXT
[END OF LOOP]
Step 5: EXIT

UNIT-2

Algorithm to print the number of nodes in a linked list:

Step 1: [INITIALIZE] SET COUNT =0

Step 2: [INITIALIZE] SET PTR = START

Step 3: Repeat Steps 4 and 5 while PTR != NULL

Step 4: SET = COUNT=COUNT+1

Step 5: SET PTR = PTR →NEXT

[END OF LOOP]

Step 6: Write COUNT

Step 7: EXIT

UNIT-2

Searching for a Value in a Linked List:

- Searching a linked list means to find a particular element in the linked list.
- searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value.

Algorithm to search a linked list:

Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3: IF VAL = PTR→ DATA
SET POS = PTR
Go To Step 5
ELSE
SET PTR = PTR →NEXT
[END OF IF]
[END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT

UNIT-2

- Consider the linked list shown in Fig. 6.11. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.

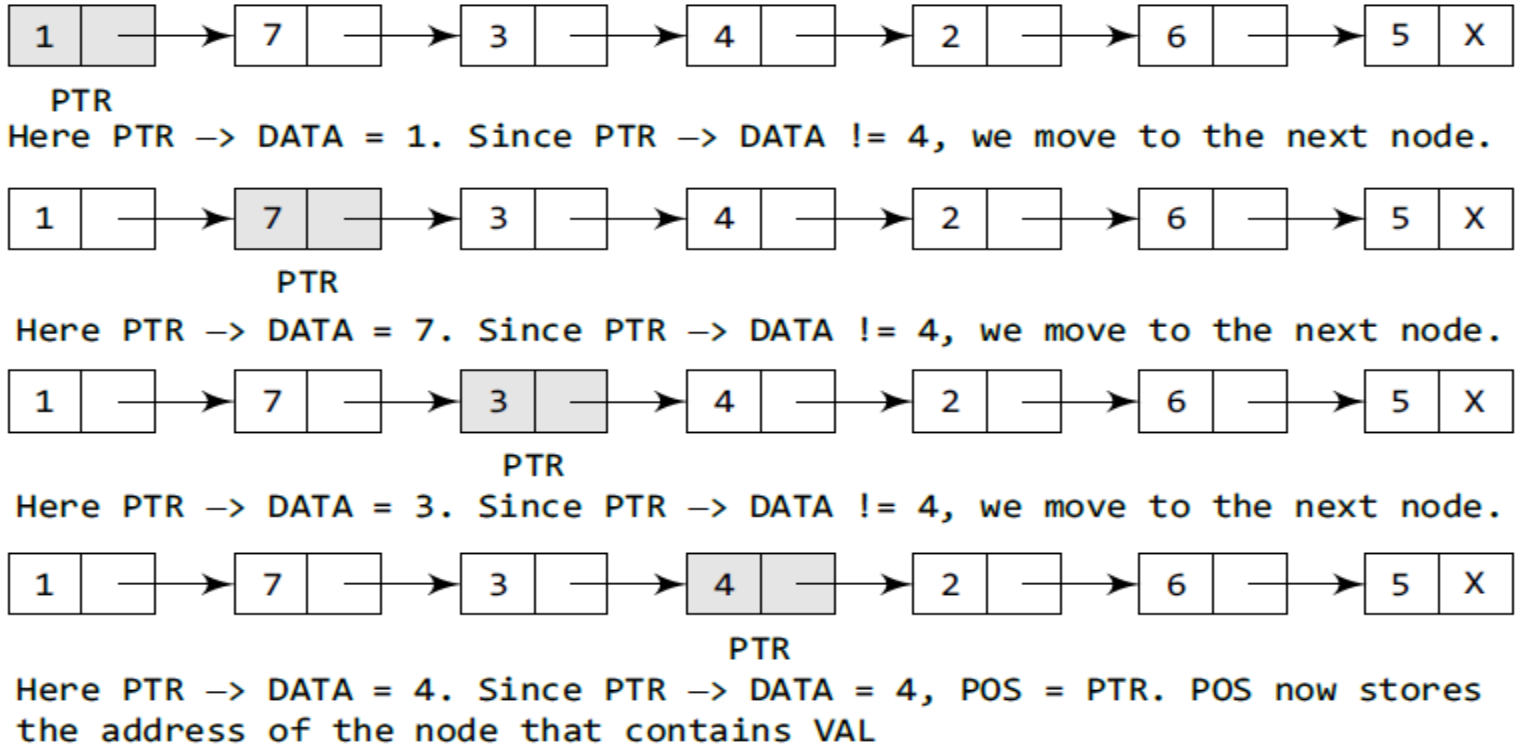


Figure 6.11 Searching a linked list

INSERTING ELEMENT IN SINGLE LINKED LIST

3 CASES TO INSERT VALUE IN SINGLE LINKED LIST:

1.Insert at beginning

2.Insert at end

3.Insert at middle of the list

INSERTING ELEMENT IN SINGLE LINKED LIST

1.Insert at beginning :

STEPS:

1.Create a new node and allocate memory to node.

2.Store value in the data part of new node

3.Change next part of new node to start

4.Change start to point to the newly created node.

INSERTING ELEMENT IN SINGLE LINKED LIST

```
void insertAtFront()
{
    int value;
    struct node* temp;
    temp = malloc(sizeof(struct node));
    printf("\nEnter number to be inserted : ");
    scanf("%d", &value);
    temp->data = value;

    // Pointer of temp will be
    // assigned to start
    temp->next = start;
    start = temp;
}
```

INSERTING ELEMENT IN SINGLE LINKED LIST

```
void insertAtEnd()
{
    int value;
    struct node *temp, *ptr;
    temp = malloc(sizeof(struct node));

    // Enter the number
    printf("\nEnter number to"
           " be inserted : ");
    scanf("%d", &value);

    // Changes links
    temp->next =NULL;
    temp->data= value;
    ptr = start;
    while (ptr->next != NULL) {
        ptr = ptr>next;
    }
    ptr->next= temp;
}
```

INSERTING ELEMENT IN SINGLE LINKED LIST

```
void insertAtPosition()
{
    struct node *ptr, *temp;
    int pos, value, i = 0;
    newnode = malloc(sizeof(struct node));

    // Enter the position and data
    printf("\nEnter position and data :");
    scanf("%d %d", &pos, &data);

    // Change Links
    ptr = start;
    temp->data = value;
    temp->next = NULL;
    while (i < pos - 1) {
        ptr = ptr->next;
        i++;
    }
    temp->next = ptr->next;
    ptr-->next = temp;
}
```

DELETING ELEMENT IN SINGLE LINKED LIST

```
void deleteFirst()
{
    struct node* temp;
    if (start == NULL)
        printf("\nList is empty\n");
    else {
        temp = start;
        start = start->link;
        free(temp);
    }
}
```

DELETING ELEMENT IN SINGLE LINKED LIST

```
void deleteEnd()
{
    struct node *temp, *prevnode;
    if (start == NULL)
        printf("\nList is Empty\n");
    else {
        temp = start;
        while (temp->link != 0) {
            prevnode = temp;
            temp = temp->link;
        }
        free(temp);
        prevnode->link = 0;
    }
}
```

CIRCULAR LINKED LIST

- In a circular linked list, the last node contains a pointer to the first node of the list.
- While traversing a circular linked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending. Figure 6.26 shows a circular linked list.

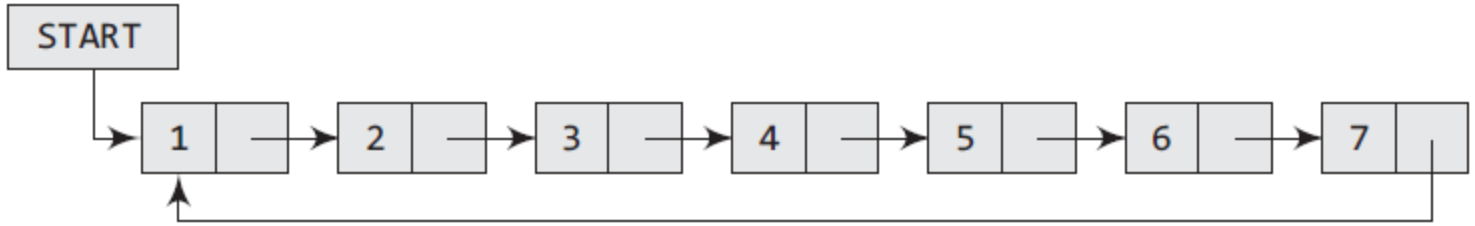


Figure 6.26 Circular linked list

- Note that there are no NULL values in the NEXT part of any of the nodes of list.

CIRCULAR LINKED LIST

- Consider Fig. 6.27.

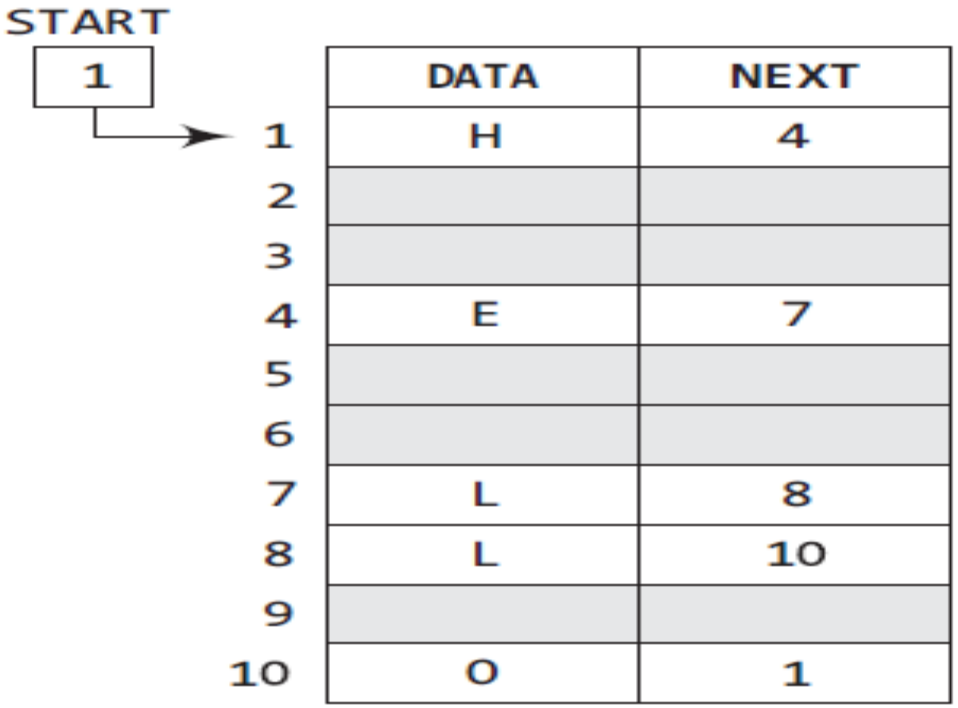


Figure 6.27 Memory representation of a circular linked list

CIRCULAR LINKED LIST

The node that contains the address of the first node is actually the last node of the list.

CIRCULAR LINKED LIST

Operations on circular linked list:

1.Insertion

In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

- Inserting At Beginning of the list
- Inserting At End of the list
- Inserting At Specific location in the list

2. **Deletion**

In a circular linked list, the deletion operation can be performed in three ways those are as follows...

- Deleting from Beginning of the list
- Deleting from End of the list
- Deleting a Specific Node

3.Traversing/display

CIRCULAR LINKED LIST

Inserting At Beginning of the list:

We can use the following steps to insert a new node at beginning of the circular linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**start** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **start**= **newNode** and **newNode→next** = **start**.
- **Step 4 -** If it is **Not Empty** then, define a Node pointer '**ptr**' and initialize with '**start'**.
- **Step 5 -** Keep moving the '**ptr**' to its next node until it reaches to the last node (until '**ptr→ next** == **start**').
- **Step 6 -** Set '**newNode → next** =**start**', '**start**= **newNode**' and '**ptr→ next** = **start**.

CIRCULAR LINKED LIST

- **Inserting At End of the list**
- We can use the following steps to insert a new node at end of the circular linked list...
- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** == **head**).
- **Step 6 -** Set **temp → next** = **newNode** and **newNode → next** = **head**.

CIRCULAR LINKED LIST

- **Inserting At Specific location in the list (After a Node)**
- We can use the following steps to insert a new node after a node in the circular linked list...
- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.
- **Step 8 -** If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**.

CIRCULAR LINKED LIST

- **Deleting from Beginning of the list**
- We can use the following steps to delete a node from beginning of the circular linked list...
- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- **Step 4 -** Check whether list is having only one node (**temp1 → next** == **head**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head**)
- **Step 7 -** Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

CIRCULAR LINKED LIST

- **Deleting from End of the list**
- We can use the following steps to delete a node from end of the circular linked list...
- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1 → next** == **head**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)
- **Step 7 -** Set **temp2 → next** = **head** and delete **temp1**.

CIRCULAR LINKED LIST

- **Deleting a Specific Node from the list**
- We can use the following steps to delete a specific node from the circular linked list...
- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next** == **head**)
- **Step 7 -** If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- **Step 1 1-** If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

CIRCULAR LINKED LIST

- **Displaying a circular Linked List**
- We can use the following steps to display the elements of a circular linked list...
- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
- **Step 5 -** Finally display **temp → data** with arrow pointing to **head → data**.

CIRCULAR LINKED LIST

```
void beginsert()
{
   struct node *ptr,*temp;
   int item;
   temp = (struct node *)malloc(sizeof(struct node));
   if(temp == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter the node data?");
      scanf("%d",&item);
      temp -> data = item;
      if(start == NULL)
      {
         start = temp;
         temp -> next = start;
      }

}
```

CIRCULAR LINKED LIST

```
else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
        printf("\nnode inserted\n");
    }

}
```

CIRCULAR LINKED LIST

```
void insert_end()
{
   struct node *ptr,*temp;
   int value;
   temp = (struct node *)malloc(sizeof(struct node));
   if(temp == NULL)
   {
      printf("memory not allocated\n");
   }
   else
   {
      printf("\nEnter value\n");
      scanf("%d",&value);
      temp->data =value;
      if(start== NULL)
      {
         start= temp;
         temp-> next = start;
      }
      else
      {
         ptr = start;
         while(ptr -> next != start)
         {
            ptr = ptr -> next;
         }
         ptr -> next = temp;
         temp -> next = start;
      }

      printf("node inserted\n");
   }

}
```

DOUBLY LINKED LISTS

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node as shown in Fig. 6.37.

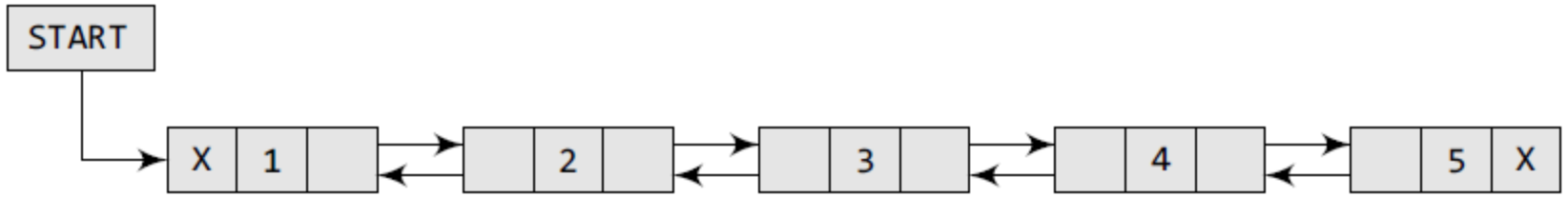


Figure 6.37 Doubly linked list

In C, the structure of a doubly linked list can be given as,

```
struct node
{
struct node *prev;
int data;
struct node *next;
};
```

DOUBLY LINKED LISTS

- The PREV field of the first node and the NEXT field of the last node will contain NULL. The PREV field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.
- However, a doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward). The main advantage of using a doubly linked list is that it makes searching twice as efficient. Let us view how a doubly linked list is maintained in the memory. Consider Fig. 6.38.

DOUBLY LINKED LISTS

- In the figure, we see that a variable START is used to store the address of the first node. In this example, START = 1, so the first data is stored at address 1, which is H. Since this is the first node, it has no previous node and hence stores NULL or –1 in the PREV field. We will traverse the list until we reach a position where the NEXT entry contains –1 or NULL. This denotes the end of the linked list.

START = 1

	DATA	PREV	NEXT
1	H	–1	3
2			
3	E	1	6
4			
5			
6	L	3	7
7	L	6	9
8			
9	O	7	–1

Figure 6.38 Memory representation of a doubly linked list

DOUBLY LINKED LISTS

- Operations in doubly linked list:

1.Traversing

2.Inserting

3.deleting

DOUBLY LINKED LISTS

Algorithm for traversing:

Step1:

Set ptr=start

Step2:

Repeat step 3 and 4 while ptr=NULL

Step3:

Process ptr→data

Step 4:

Set ptr=ptr→next

Step5:

exit

DOUBLY LINKED LISTS

Inserting at beginning of doubly linked list algorithm:

Step1:

Create a new node “temp”

Allocate memory to temp

If temp==null “memory not allocated”

Exit

Step2:

Set temp→data=value

Step3:

Set temp→prev=NULL

Step4:

Set temp→next==start

Step 5:

Set start→prev=temp

Step6:

Set start=temp

Step 7:

exit

DOUBLY LINKED LISTS

Inserting at end of doubly linked list algorithm:

Step1:

Create a new node “temp”

Allocate memory to temp

If temp==null “memory not allocated”

Exit

Step2:

Temp→data=value

Temp→next=NULL

Step3:

Set ptr=start

Step 4:

Repeat step 5 and 6 while ptr !=null

Step 5:

Set prevnode=ptr

DOUBLY LINKED LISTS

Inserting at end of doubly linked list algorithm:

Step6:

Set ptr=ptr→next

Step 7:

Set prevnode→next=temp

Step8:

Set temp→prev=prevnode

Step 9:

exit

DOUBLY LINKED LISTS

Inserting at specified position of doubly linked list algorithm:

Step1:

Create a new node “temp”

Allocate memory to temp

If temp==null “memory not allocated”

Exit

Step2:

Temp→data=value

Step3:

Set ptr=start

Step 4:

Read position pos

Step 5:

Set i=0

Step 6:

repeat step 7 to 10 while i<pos && ptr!=NULL

DOUBLY LINKED LISTS

Inserting at specified position of doubly linked list algorithm:

Step8:

Set prevnode=ptr

Step 9:

Set ptr=ptr→next

Step 10:

Set i=i+1

Step 11:

If ptr==null “position not found”

Step 12;

Set prevnode→next=temp

Step 13:

Temp→prev=prevnode

Step 14:

Set temp→next=ptr

Step 15:

Ptr→prev=temp

Step 16:

DOUBLY LINKED LISTS

Deletion at beginning of linked list:

step1:

Temp=start

Step2:

If start==null

“list empty”

Step 3:

Set Start=start→next

Step 4:

Set start→prev=null

Step5:

free(temp)

Step 6:

exit

DOUBLY LINKED LISTS

Deletion at end of linked list:

step1:

Temp=start

Step2:

If start==null

“list empty”

Step 3:

Repeat step 4 and 5 while temp→next !=null

Step 4:

Set prevnode=temp

Step5:

temp=temp→next

Step 6:

Prev→next=null

Step 7:

Free(temp)

Step 8:

exit

DOUBLY LINKED LISTS

Deletion at specified location of linked list:

Step1:

Set temp=start

Step2:

If temp==null

“underflow”

Exit

Step3:

Set i=0

Step4:

Read position pos

Step5:

If pos==0 and temp==null exit

Step 6:

Repeat step 7 and 8while temp!=null && i< pos

Step7:

Set prevnode=temp

Step 8:

Set temp=temp→next

Header linked LISTS

- A header linked list is a special type of linked list which contains a header node at the beginning of the list. So, i header linked list, START will not point to the first node of the list but START will contain the address of the hea node.
- The following are the two variants of a header linked list:
- **Grounded header linked list** *which stores NULL in the next field of the last node.*
- **Circular header linked list** *which stores the address of the header node in the next field of* the last node.
- Here, the header node will denote the end of the list. Look at Fig. 6.65 which shows both the types of header lin lists.

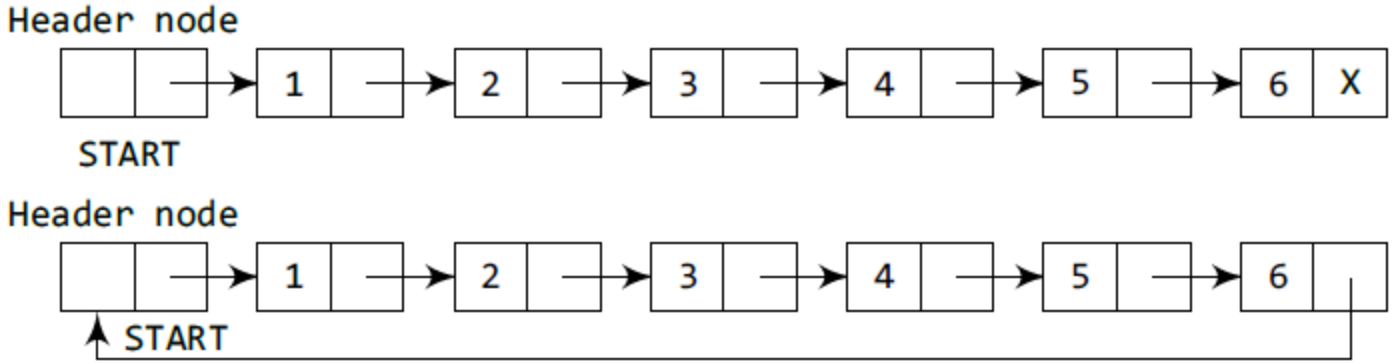


Figure 6.65 Header linked list

Header linked LISTS

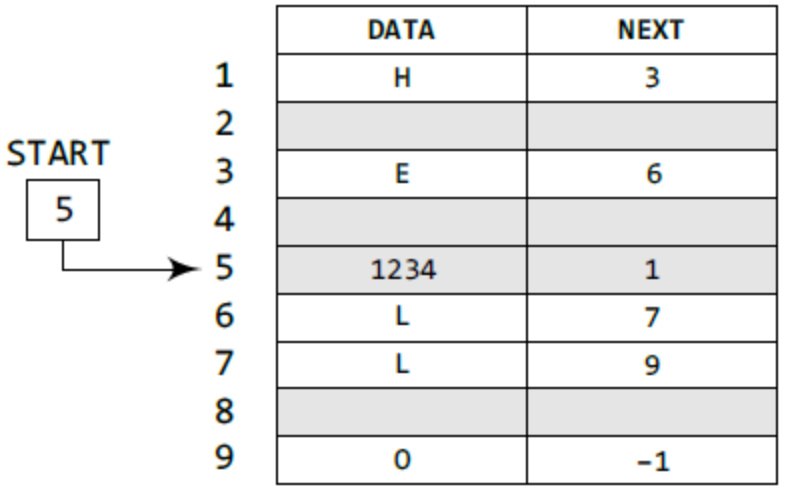


Figure 6.66 Memory representation of a header linked list

Header linked LISTS

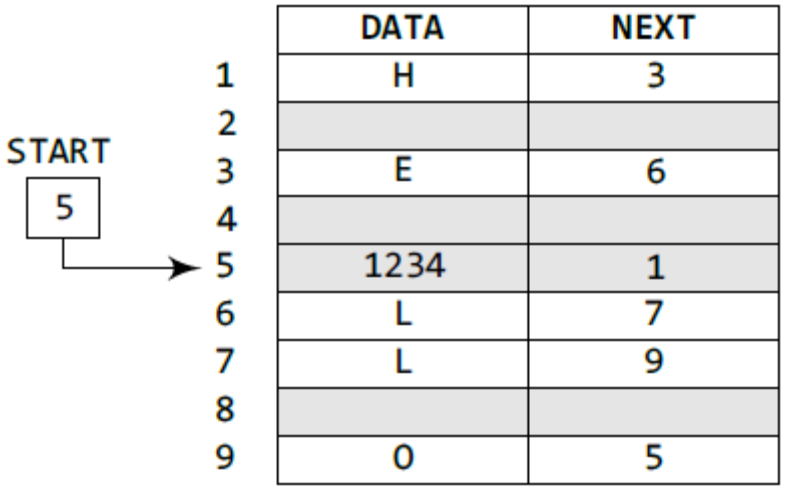


Figure 6.67 Memory representation of a circular header linked list

LINKED REPRESENTATION OF STACKS

- We have seen how a stack is created using an array.
- This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size.
- In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation.
- But if the array size cannot be determined in advance,then the other alternative, i.e., linked representation, is used.

LINKED REPRESENTATION OF STACKS

- In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.
- The linked representation of a stack is shown in Fig.

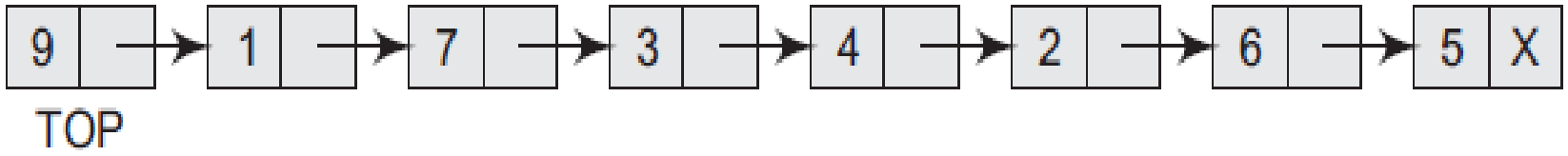


Figure 7.13 Linked stack

LINKED REPRESENTATION OF STACKS

OPERATIONS ON A LINKED STACK

- A linked stack supports all the three stack operations, that is, push, pop, and peek.

1 Push Operation:

- The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in Fig. 7.14.

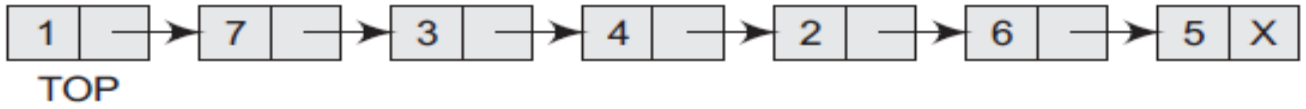


Figure 7.14 Linked stack

LINKED REPRESENTATION OF STACKS

- To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP. Thus, the updated stack becomes as shown in Fig. 7.15.

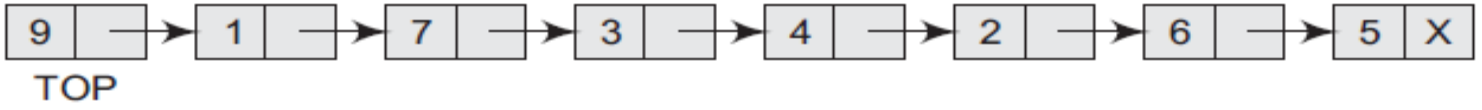


Figure 7.15 Linked stack after inserting a new node

LINKED REPRESENTATION OF STACKS

Algorithm to insert an element in a linked stack:

Step 1: Allocate memory for the new node and name it as NEW_NODE

Step 2: SET NEW_NODE→ DATA = VAL

Step 3: IF TOP = NULL

SET NEW_NODE →NEXT = NULL

SET TOP=NEW_NODE

ELSE

SET NEW_NODE →NEXT = TOP

SET TOP=NEW_NODE

Step 4: END

LINKED REPRESENTATION OF STACKS

2 Pop Operation:

- The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done.
- If an attempt is made to delete a value from a stack that is alreadyempty, an UNDERFLOW message is printed. Consider the stack shown in Fig. 7.17.

Figure 7.17 Linked stack

LINKED REPRESENTATION OF STACKS

- In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown in Fig. 7.18.

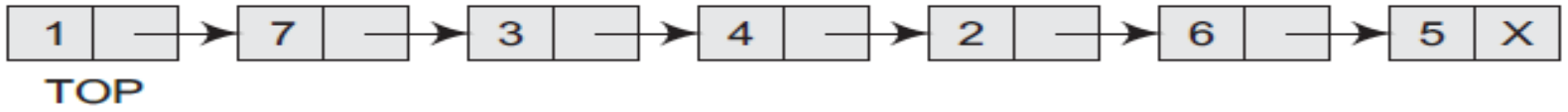


Figure 7.18 Linked stack after deletion of the topmost element

LINKED REPRESENTATION OF STACKS

Algorithm to delete an element from a linked stack:

Step 1: IF TOP = NULL

PRINT UNDERFLOW

GOTO STEP5

[END OF IF]

Step 2: SET PTR = TOP

Step 3: SET TOP = TOP -> NEXT

Step 4: FREE PTR

Step 5: END

LINKED REPRESENTATION OF QUEUES

- We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size. If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted. And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.
- In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.
- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty.

LINKED REPRESENTATION OF QUEUES

- The linked representation of a queue is shown in Fig. 8.6.

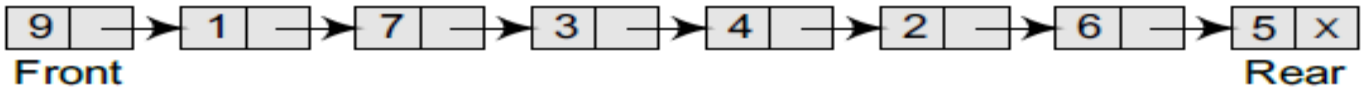


Figure 8.6 Linked queue

LINKED REPRESENTATION OF QUEUES

- **Operations on Linked Queues**

A queue has two basic operations:

- insert and
- delete.
- **The insert operation** adds an element to the **end of the queue**, and the **delete operation** removes an element from the front or the **start of the queue.** Apart from this, there is another **operation peek** which returns the value of the **first element of the queue**.

LINKED REPRESENTATION OF QUEUES

Insert Operation:

- The insert operation is used to insert an element into a queue. The new element is added as the last element of the queue. Consider the linked queue shown in Fig. 8.7.

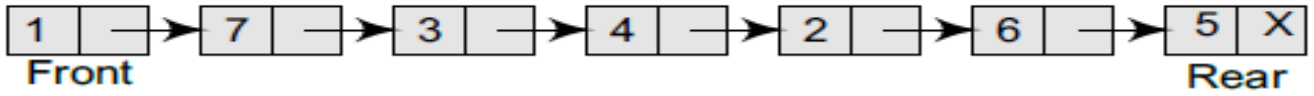


Figure 8.7 Linked queue

To insert an element with value 9, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its data part and NULL in its next part. The new node will then be called both FRONT and rear. However, if FRONT != NULL, then we will insert the new node at the rear end of the linked queue and name this new node as rear. Thus, the updated queue becomes as shown in Fig. 8.8.

LINKED REPRESENTATION OF QUEUES

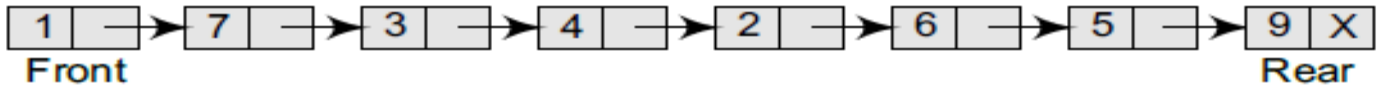


Figure 8.8 Linked queue after inserting a new node

LINKED REPRESENTATION OF QUEUES

Algorithm to insert an element in a linked queue:

Step 1: Allocate memory for the new node and name it as TEMP

Step 2: SET TEMP→ DATA = VALUE

Step 3: IF FRONT = NULL

SET FRONT = REAR = TEMP

SET FRONT→ NEXT = REAR→ NEXT = NULL

ELSE

SET REAR→ NEXT = TEMP

SET REAR = TEMP

SET REAR→ NEXT = NULL

Step 4: END

LINKED REPRESENTATION OF QUEUES

Delete Operation:

- The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT. However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed. Consider the queue shown in Fig. 8.10.

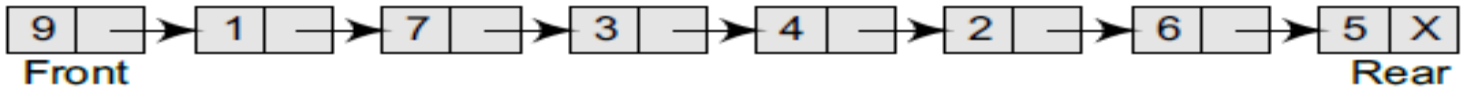


Figure 8.10 Linked queue

LINKED REPRESENTATION OF QUEUES

Delete Operation:

- To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the linked queue. Thus, the updated queue becomes as shown in Fig. 8.11.

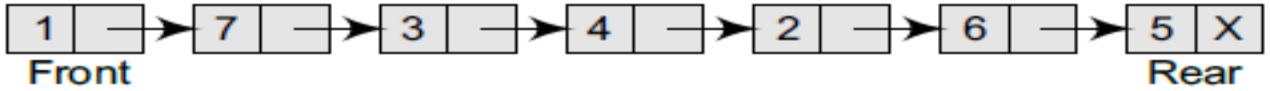


Figure 8.11 Linked queue after deletion of an element

LINKED REPRESENTATION OF QUEUES

Algorithm to delete an element from a linked queue:

Step 1: IF FRONT = NULL

Write Underflow

Go to Step 5

Step 2: SET PTR = FRONT

Step 3: SET FRONT = FRONT -> NEXT

Step 4: FREE PTR

Step 5: END

APPLICATIONS OF LINKED LISTS
Polynomial Representation

- Let us see how a polynomial is represented in the memory using a linked list. Consider a polynomial 6x3 + 9x2 + 7x + 1. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.
- Every term of a polynomial can be represented as a node of the linked list. Figure 6.74 shows the linked representation of the terms of the above polynomial.

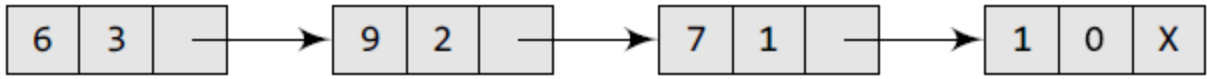


Figure 6.74 Linked representation of a polynomial

APPLICATIONS OF LINKED LISTS
Polynomial Representation

ALGORITHM FOR ADDITION OF 2 POLYNOMIALS:

STEP1:

Loop around all values of linked list

Step2:

If value of node exponent is greater copy this node to the result and increment the pointer

Step3:

If value of both exponents is same then add the coefficients and add to the result.

APPLICATIONS OF LINKED LISTS
Polynomial Representation

Structure of node:

```
Struct node
{
Int coefficient;
Int exponent;
Struct node *next;
};
```

APPLICATIONS OF LINKED LISTS
SPARSE MATRICES

- A sparse matrix is a matrix that contains more number of zero elements than non zero elements .
- A sparse matrix can be represented by using
- 1.Arrays
- 2.Linked list
- In linked list representation of sparse matrix a node contains 4 parts

1.row:

- Index of row where non zero element is located

2.column:

- Index of column where non zero element is located

3.value:

Value of the non zero element located at index row-column

4.next:

Address of next node

APPLICATIONS OF LINKED LISTS
SPARSE MATRICES

Node structure:

```
Struct node
{
Int row;
Int column;
Int value;
Struct node *next;
};
```

UNIT-3
TREES

1. **TERMINOLOGY**
2. **REPRESENTATION OF TREES**
3. **BINARY TREE**
   - ❖PROPERTIES OF BINARY TREES
   - ❖BINARY TREE REPRESENTATION
   - ❖BINARY TREE TRAVERSAL
     - ❖PREORDER,INORDER AND POSTORDER TRAVERSAL
4. **THREADS**
   - ❖THREAD BINARY TREES
5. **BALANCED BINARY TREES**
6. **HEAPS**
   - ❖MAX HEAP
     - ❖INSERTION INTO MAX HEAP
     - ❖DELETION FROM A MAX HEAP

UNIT-3
TREES

7.BINARY SEARCH TREES

- ❖SEARCHING
- ❖INSERTION
- ❖DELETION FROM BINARY SEARCH TREE
- ❖HEIGHT OF BINARY SEARCH TREE

8.M-WAY SEARCH TREES

9.B-TREES

UNIT-3
TERMINOLOGY

TREES:

•A tree is a non linear data structure

•A tree is a finite set of nodes with finite set of edges that define parent-child relationship and there are no circuits.

•EX:

•

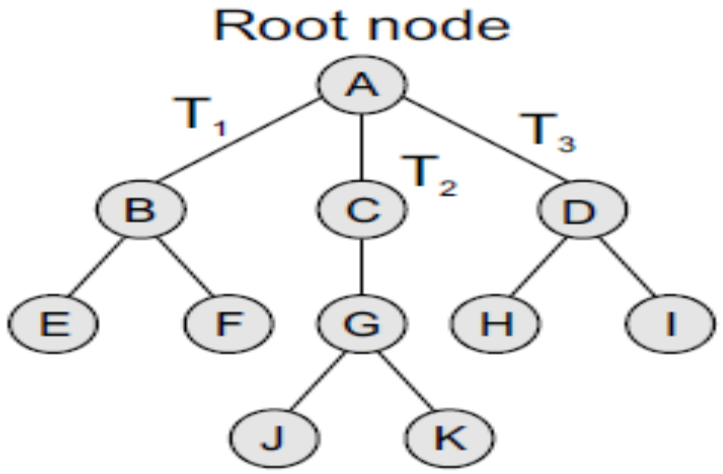


Figure 9.1 Tree

UNIT-3
TERMINOLOGY

Properties of tree:

1.there is only one root node having no parent.

2.Except root node,each node have exactly one parent.

3.A node may have zero or more children

4.There is unique path from root to each node

UNIT-3
TERMINOLOGY

1.Root node:
•Is the topmost node in the tree.
•Every tree contains only one root node.
•If root node is null it means tree is empty
2.Parent node:
•In a tree if a node contains atleast one child then such a node is called parent node.
3.Child node:/descendant node:
•In a tree if a node is having a parent node then such a node is called as child node.
4.siblings:
•In a tree the nodes which are having same parent is called as siblings.
5.Leaf node:
•In a tree if a node doesn't have any child then such a node is called as leaf nodes.
•Leaf nodes can also be called as external nodes or terminal nodes.
6.Edge:
•In a tree a connecting link between any two nodes is known as an edge.
•If a tree contains 'n' number of nodes then the tree contains 'n-1' edges.

UNIT-3
TERMINOLOGY

7.Sub tree:

•In a tree,a subtree can be formed by using child or children of a particular node.

8.Internal nodes:

•In a tree except the leaf nodes the remaining nodes can be called as internal nodes.

•The internal nodes can also be called as non terminal nodes.

9.degree of node:

•Degree of a node is equal to the number of children that a node has.

•The degree of leaf node is zero.

10.degree of tree:

•Highest degree of node in a given tree

11.level of tree:

•Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

12. Height:

•In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**

UNIT-3
TERMINOLOGY

13. Depth:

•In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**

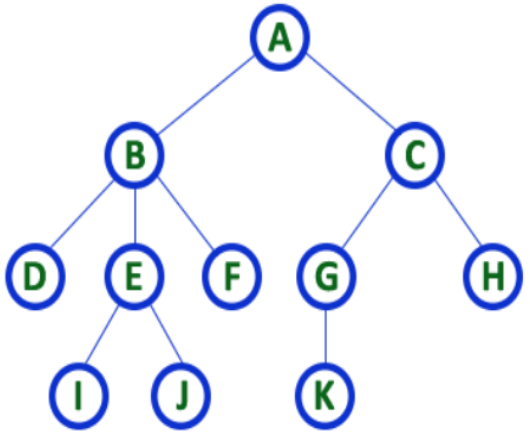
UNIT-3
Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

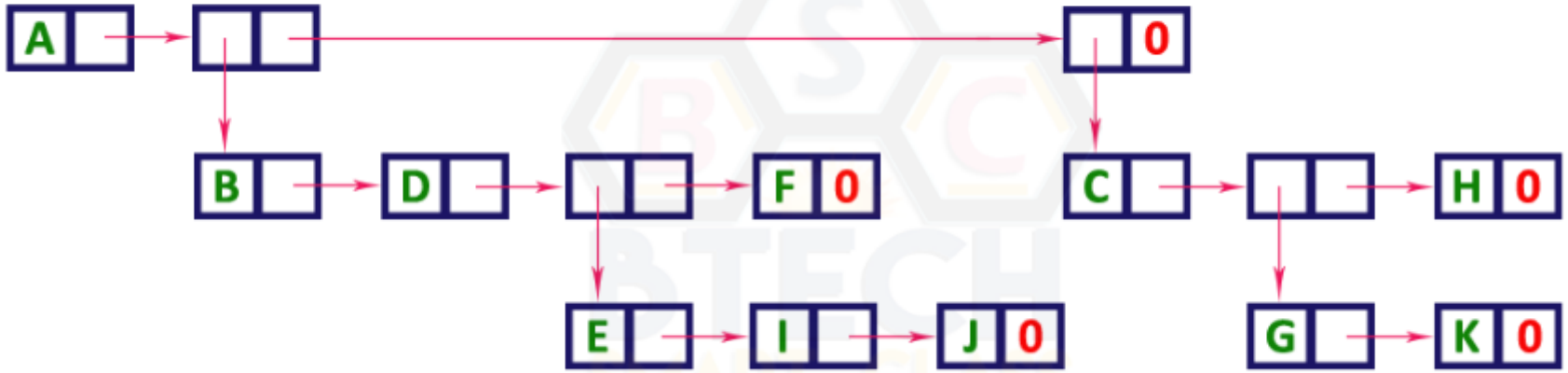
- **List Representation**
- **Left Child - Right Sibling Representation**

UNIT-3
Tree Representations

1. List Representation:

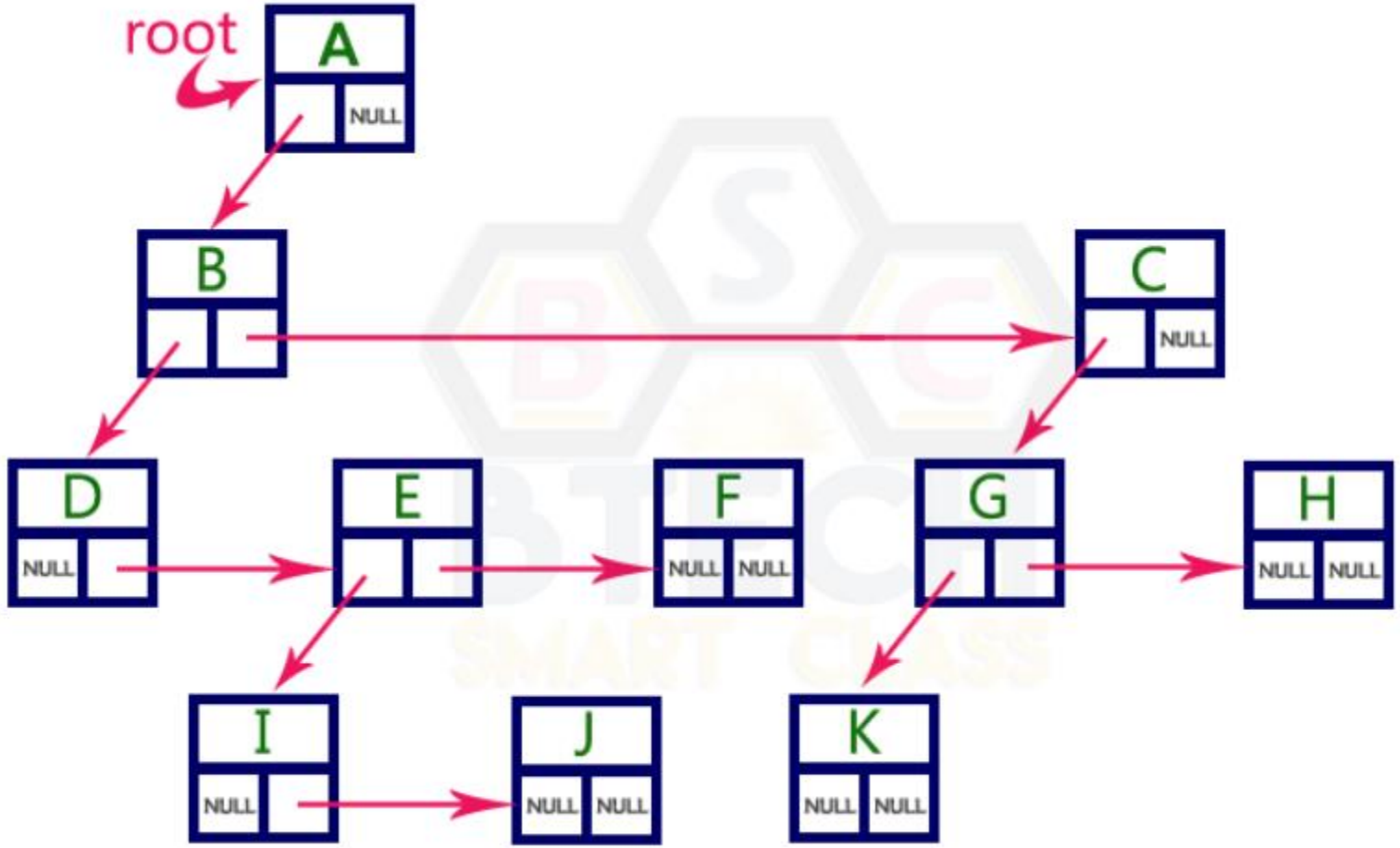


The above example tree can be represented using List representation as follows...



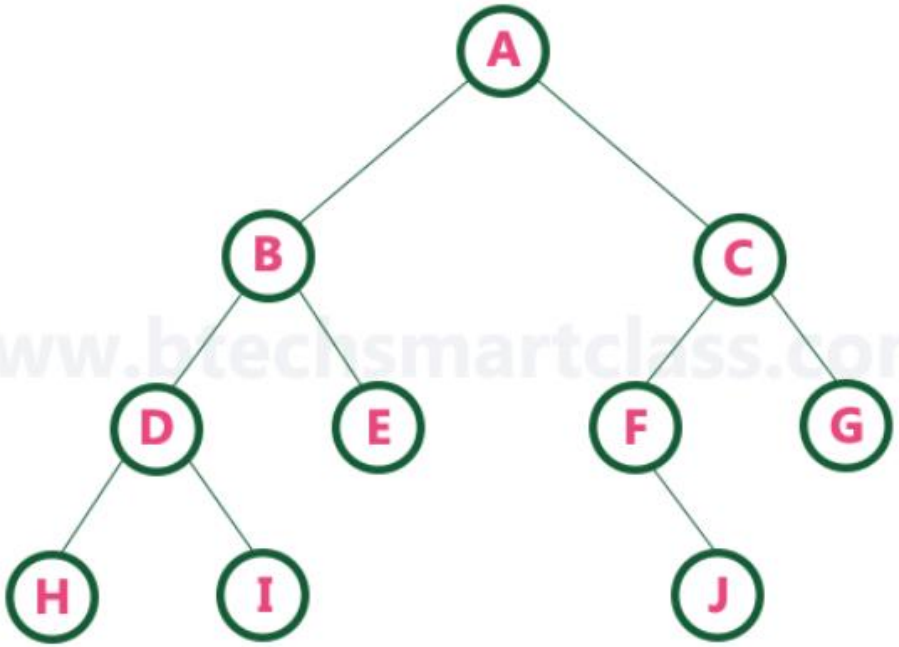
UNIT-3
Tree Representations

2.Left Child - Right Sibling Representation:



UNIT-3
BINARY TREE

- A binary tree is a data structure that is defined as a collection of elements called nodes. In a binarytree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children.
- A node that has zero children is called a leaf node or a terminal node.
- Every node contains a data element, a left pointer which points to the left child, and a right pointer which points to the right child.

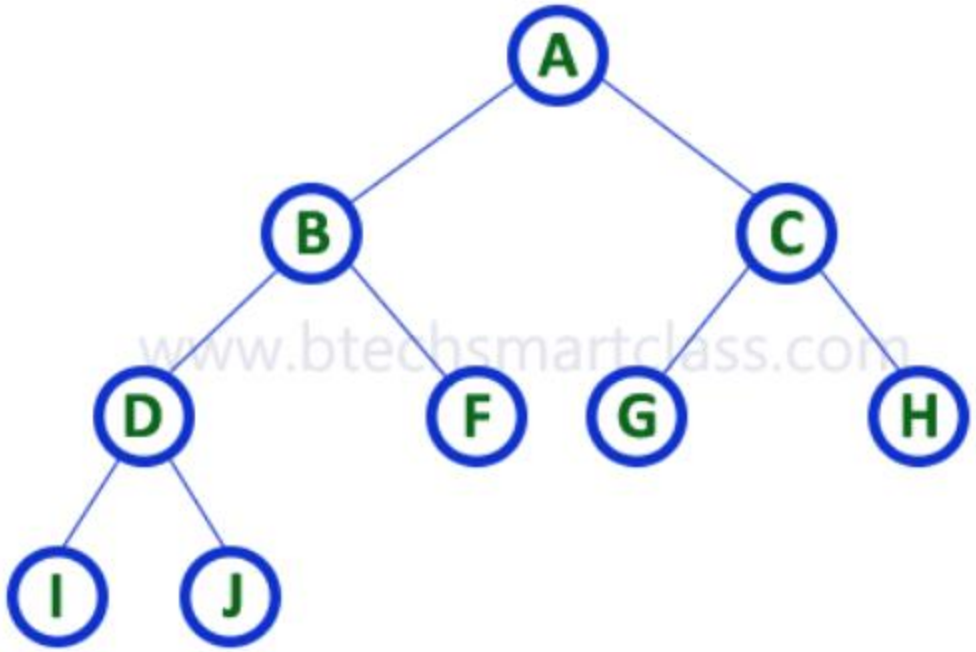


UNIT-3
BINARY TREE

- There are different types of binary trees and they are

1. Strictly Binary Tree:

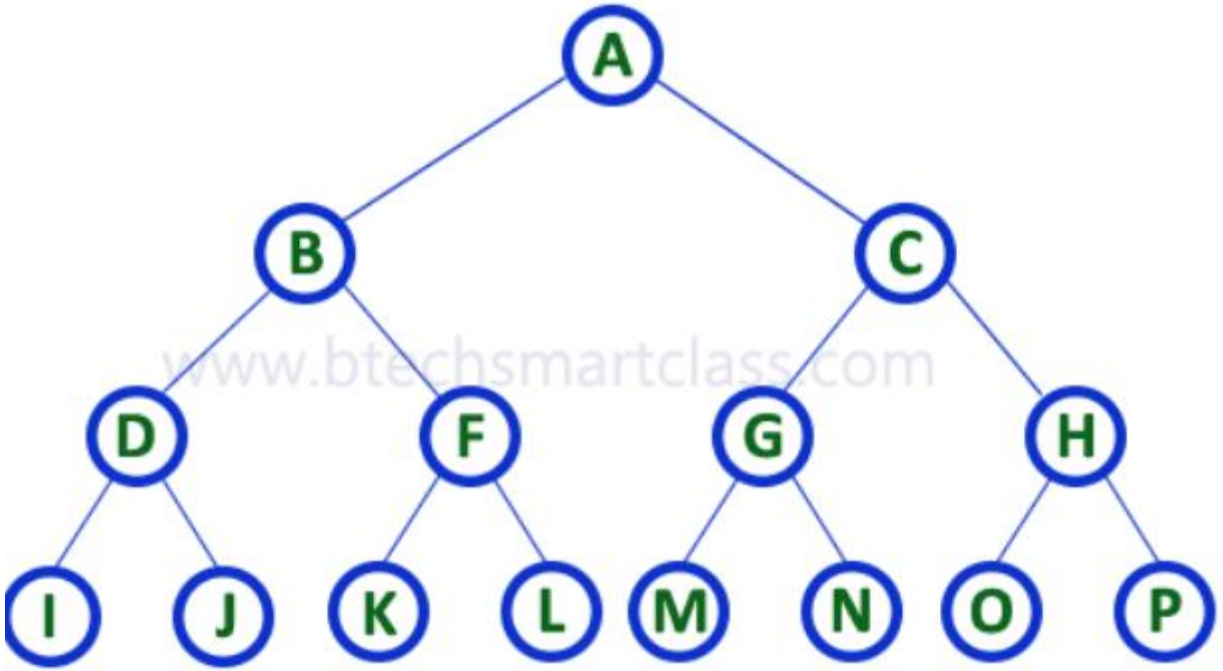
- A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree
- Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree**



UNIT-3
BINARY TREE

2.Complete Binary Tree:

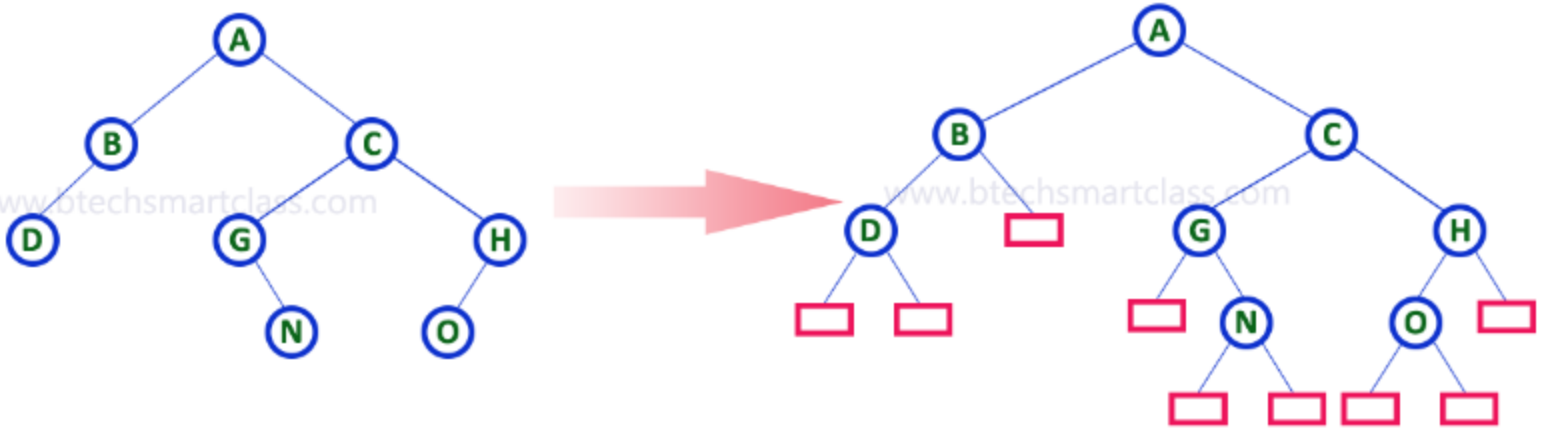
- A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.



UNIT-3
BINARY TREE

3. Extended Binary Tree:

- A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.
- **The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.**



UNIT-3
PROPERTIES OF BINARY TREE

1.The Maximum number of nodes in a binary tree of height ‘h’ is $2^{h+1} – 1$.
2.Minimum number of nodes in a binary tree of height ‘h’ is h+1
3.Total number of leafnodes in a binary tree is equal to the total number of nodes that has 2 children+1
4.The maximum number of nodes in a binary tree at level ‘l’ is 2l
5.If a binary tree ‘T’ has n nodes,then it has n-1 edges.

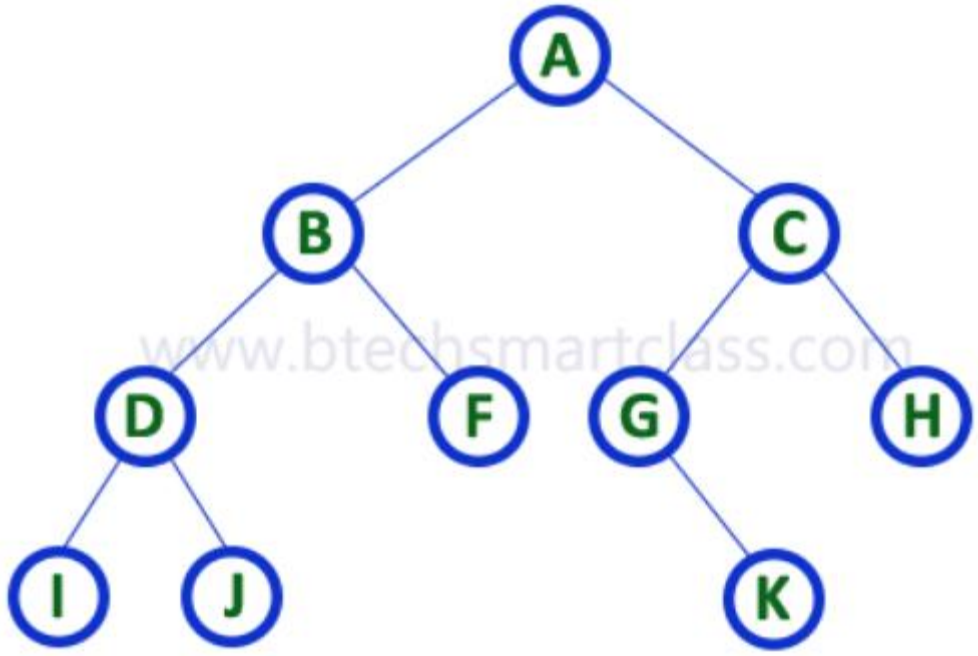
UNIT-3
BINARY TREE REPRESENTATION

- A binary tree data structure is represented using two methods. Those methods are as follows...

1.Array Representation

2.Linked List Representation

Consider the following binary tree



UNIT-3
BINARY TREE REPRESENTATION

1.Array Representation

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

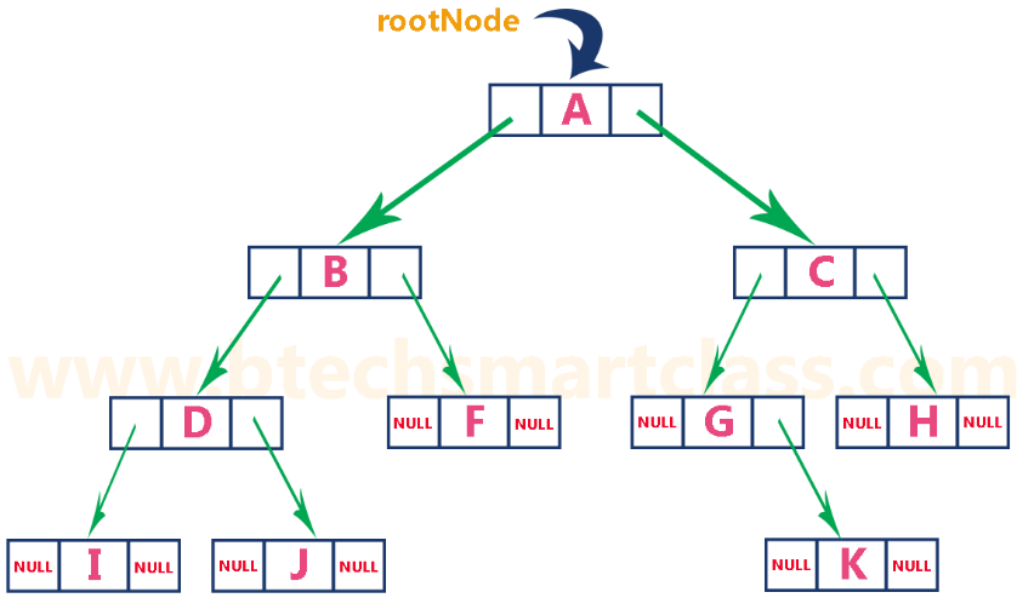
Consider the above example of a binary tree and it is represented as follows...

BINARY TREE REPRESENTATION

2. Linked List Representation of Binary Tree

- We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.
  In this linked list representation, a node has the following structure

-



UNIT-3
Binary Tree Traversals

Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal.

- There are three types of binary tree traversals.

1.In - Order Traversal

2.Pre - Order Traversal

3.Post - Order Traversal

UNIT-3
Binary Tree Traversals

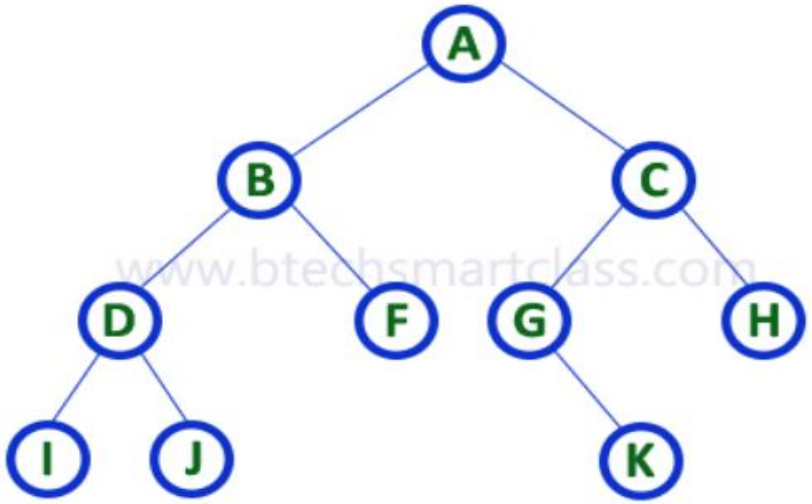
1.In - Order Traversal:

- In In-Order traversal, the root node is visited between the left child and right child. In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.

STEPS FOR IN-ORDER TRAVERSAL:(left-root-right)

1.Visit the leftsubtree using inorder
2.Visit the root
3.Visit the right subtree using inorede

Consider the following binary tree



In-Order Traversal for above example of binary tree is
I - D - J - B - F - A - G - K - C - H

UNIT-3
Binary Tree Traversals

2. Pre - Order Traversal (root - left - right)

- In Pre-Order traversal, the root node is visited before the left child and right child nodes. In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.
- **STEPS FOR PRE-ORDER TRAVERSAL:**

1.Visit the root
2.Visit the leftsubtree using PRE order
3.Visit the right subtree using PREORDER

Consider the following binary tree

Pre-Order Traversal for above example binary tree is
A - B - D - I - J - F - C - G - K - H

UNIT-3
Binary Tree Traversals

3. Post - Order Traversal (left - right - root)

- In Post-Order traversal, the root node is visited after left child and right child. In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.
- **STEPS FOR POST-ORDER TRAVERSAL:**

1.Visit the leftsubtree using Post order
2.Visit the right subtree using Post ORDER
3.Visit the root

Consider the following binary tree

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C - A

Binary Tree Traversals

Constructing a Binary Tree from Traversal Results

- We can construct a binary tree if we are given at least two traversal results.
- The first traversal must be the in-order traversal and the second can be either pre-order or post-order traversal.
- The in-order traversal result will be used to determine the left and the right child nodes, and the pre-order/post-order can be used to determine the root node.
- For example, consider the traversal results given below:
- **In–order Traversal: D B E A F C G Pre–order Traversal: A B D E C F G**
- Here, we have the in-order traversal sequence and pre-order traversal sequence. Follow the steps given below to construct the tree:

Step 1 Use the pre-order sequence to determine the root node of the tree. The first element would be the root node.

Step 2 Elements on the left side of the root node in the in-order traversal sequence form the left sub-tree of the root node. Similarly, elements on the right side of the root node in the in-order traversal sequence form the right sub-tree of the root node.

Step 3 Recursively select each element from pre-order traversal sequence and create its left and right sub-trees from the in-order traversal sequence.

Binary Tree Traversals

Constructing a Binary Tree from Traversal Results

- **In–order Traversal: D B E A F C G Pre–order Traversal: A B D E C F G**

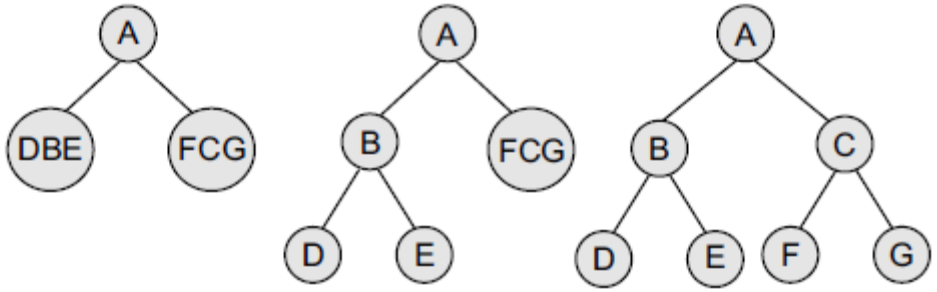


Figure 9.20

Binary Tree Traversals

Constructing a Binary Tree from Traversal Results

- Now consider the in-order traversal and post-order traversal sequences of a given binary tree. Before constructing the binary tree, remember that in post-order traversal the root node is the last node. Rest of the steps will be the same as mentioned above
- **In–order Traversal: D B H E I A F J C G Post order Traversal: D H I E B J F G C A**

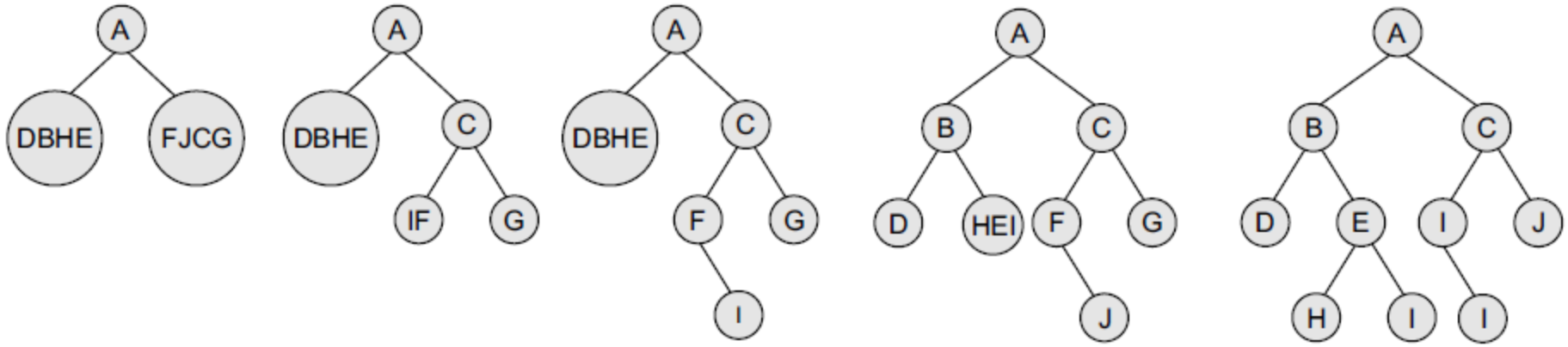


Figure 9.21 Steps to show binary tree

UNIT-3

THREADED BINARY TREE

- A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers.
- In the linked representation, a number of nodes contain a NULL pointer, either in their left or right fields or in both. This space that is wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information.
- For example, the NULL entries can be replaced to store a pointer to the **in-order predecessor** or the **in-order successor** of the node. These special pointers are called *threads and binary trees containing* threads are called *threaded trees. In the* linked representation of a threaded binary tree, threads will be denoted using arrows.

UNIT-3

balanced BINARY TREE

- A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.
- Following are the conditions for a height-balanced binary tree:

1.difference between the left and the right subtree for any node is not more than one

2.the left subtree is balanced

3.the right subtree is balanced

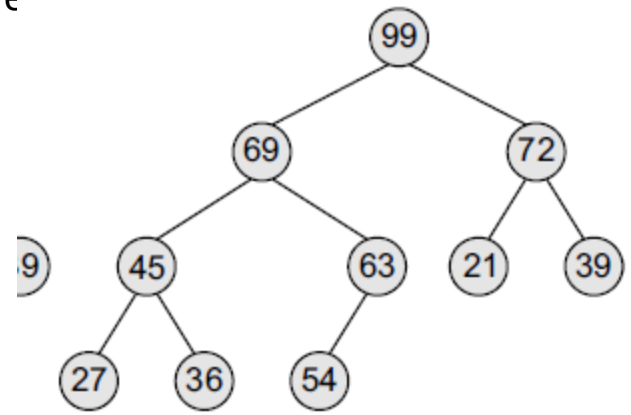
•

UNIT-3
priority queues

- A priority queue is a collection of elements where each element is assigned a priority and the order in which elements are processed is determined from the following rules:
- 1.an element is processed according to its priority
- 2.two elements with same priority are processed according to the order in which they are added to the queue.
- The operations that can be performed on priority queues are
- 1. find an element
- 2.insert an element
- 3.delete an element
- Priority queues can be 2 types:
- **1.max priority queue:**
  in this the find operation finds the element with maximum priority and the delete operation deletes this element
- **2.min priority queue:**
- in this the find operation finds the element with minimum priority and the delete operation deletes this element

UNIT-3
BINARY HEAPS

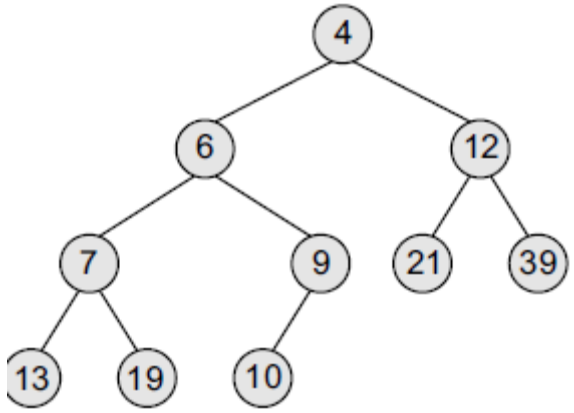
- A binary heap is a simple data structure that efficiently supports the priority queue operations
- A binary heap is a complete binary tree.
- A binary heap are of 2 types:
- **1.max-heap:**
- the elements at every node will be either greater than or equal to the element at its left and right child. Thus, the root node has the highest key value in the heap. Such a heap is commonly known as a *max-heap.*
- If B is a child of A, then key(A) > key(B)



Max heap

BINARY HEAPS

- **2.min-heap:**
- the elements at every node will be either less than or equal to the element at its left and right child. Thus, the root node has the lowest key value in the heap. Such a heap is commonly known as a *min-heap.*
- If B is a child of A, then key(A) <= key(B)



Min heap

BINARY HEAPS

- **Construct max heap 15,23,3,7,17,20,9,10,4,28,5**

Insert value 25 for the above max heap

Delete element from max heap

UNIT-3
BINARY SEARCH TREES

- A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order.
- In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node.
- Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node.
- The same rule is applicable to every sub-tree in the tree.
- **Create a binary search tree using the following data elements:**
- **45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81**

OPERATIONS ON BINARY SEARCH TREES

Searching for a Node in a Binary Search Tree:

- The search function is used to find whether a given value is present in the tree or not. The searching process begins at the
- root node. The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree. So, the search algorithm terminates by displaying an appropriate message. However, if there are nodes
- in the tree, then the search function checks to see if the key value of the current node is equal to the value to be searched. If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node. In case the value is greater than the value of the current node, it should be recursively called on the right child node.

OPERATIONS ON BINARY SEARCH TREES

Searching for a Node in a Binary Search Tree:

- **SearchElement (TREE, VAL)**
- Step 1: IF TREE →DATA = VAL OR TREE = NULL
- Return TREE
- ELSE
- IF VAL < TREE →DATA
- Return searchElement(TREE→ LEFT, VAL)
- ELSE
- Return searchElement(TREE→ RIGHT, VAL)
- [END OF IF]
- [END OF IF]
- Step 2: END

UNIT-3
OPERATIONS ON BINARY SEARCH TREES

Inserting a New Node in a Binary Search Tree:

OPERATIONS ON BINARY SEARCH TREES

Determining the Height of a Binary Search Tree:

- In order to determine the height of a binary search tree, we calculate the height of the left sub-tree and the right sub-tree.
- Whichever height is greater, 1 is added to it. For example, if the height of the left sub-tree is greater than that of the right sub-tree, then 1 is added to the left sub-tree, else 1 is added to the right sub-tree.
- Look at Fig. 10.16. Since the height of the right sub-tree is greater than the height of the left sub-tree, the height of the tree = height (right sub-tree) + 1= 2 + 1 = 3.

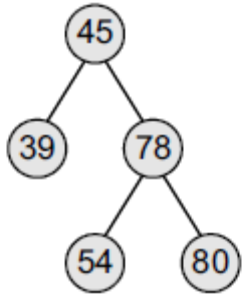


Figure 10.16 Binary search tree with height = 3

UNIT-3
Multi-way Search Trees

- In a binary search tree contains one value and two pointers, left and right, which point to the node's left and right sub-trees, respectively. The structure of a binary search tree node is shown in Fig

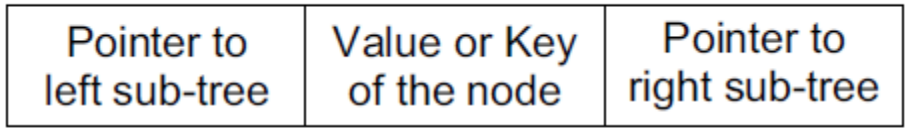


Figure 11.1 Structure of a binary search tree node

UNIT-3
Multi-way Search Trees

- The same concept is used in an M-way search tree which has M – 1 values per node and M subtrees.
- In such a tree, M is called the degree of the tree. Note that in a binary search tree M = 2, so it has one value and two sub-trees. In other words, every internal node of an M-way search tree consists of pointers to M sub-trees and contains M – 1 keys, where M > 2.

The structure of an M-way search tree node is shown in Fig. 11.2.

Figure 11.2 Structure of an M-way search tree node

- In the structure shown, P0, P1, P2, ..., Pn are pointers to the node's sub-trees and K0, K1, K2, ..., Kn–1 are the key values of the node. All the key values are stored in ascending order.

UNIT-3
Multi-way Search Trees

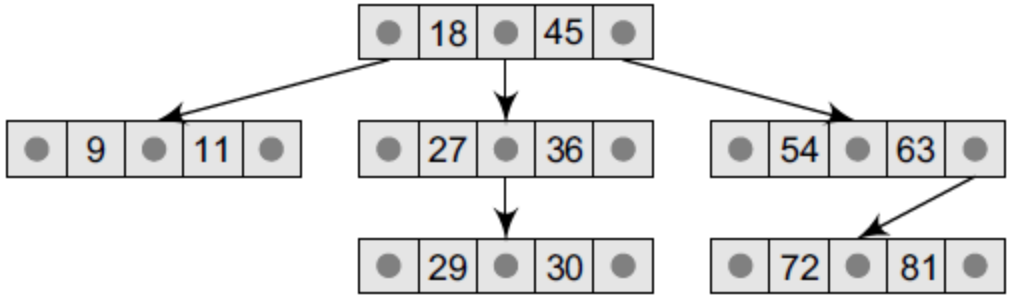


Figure 11.3 M-way search tree of order 3

UNIT-3
Multi-way Search Trees

basic properties of an M-way search tree:

- Note that the key values in the sub-tree pointed by P0 are less than the key value K0. Similarly, all the key values in the sub-tree pointed by P1 are less than K1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by Pi are less than Ki, where 0 £ i £ n–1.
- Σ Note that the key values in the sub-tree pointed by P1 are greater than the key value K0. Similarly, all the key values in the sub-tree pointed by P2 are greater than K1, so on and so forth. Thus, the generalized rule is that all the key values in the sub-tree pointed by Pi are greater thanKi–1, where 0 £ i £ n–1.

UNIT-3
B Trees

- A B tree is a specialized M-way tree
- B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.
- **B-Tree of Order m** has the following properties…

1. Every node in the B tree has at most (maximum) m children.
2. Each node has maximum m-1 values.
3. Minimum number of values

- Root has 1 value
- Leaf node has 1 value
- Intermediate nodes has m\2 value

4. All leaf nodes are at the same level.
5. Values are arranged in ascending order.
6. The tree is split into 2 nodes if and only if a new key value is to inserted into full node.

UNIT-3
B Trees

- For example, B-Tree of Order 4 contains a maximum of 3 key values in a node and maximum of 4 children for a node.

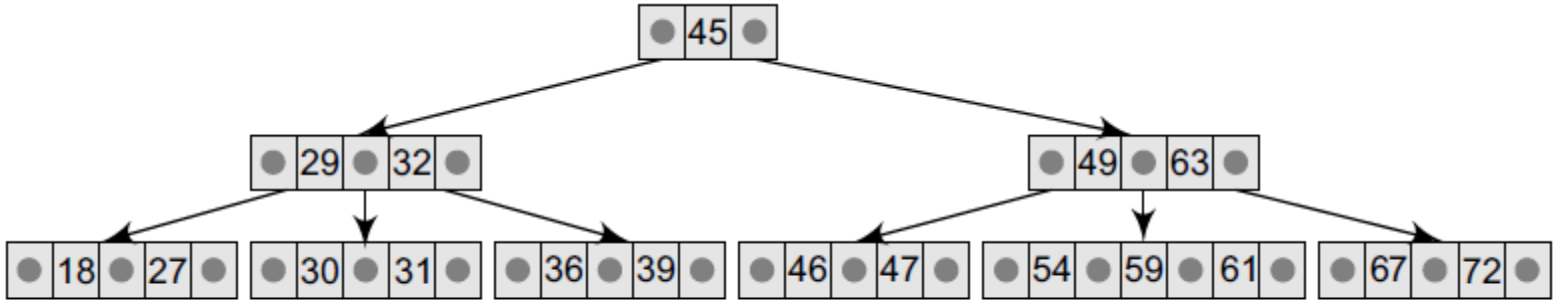


Figure 11.4 B tree of order 4

UNIT-3
B Trees

- **Operations on a B-Tree**
- The following operations are performed on a B-Tree...

1.Search

2.Insertion

3.Deletion

UNIT-3
B Trees

- Construct a B tree of order 4 where the elements are

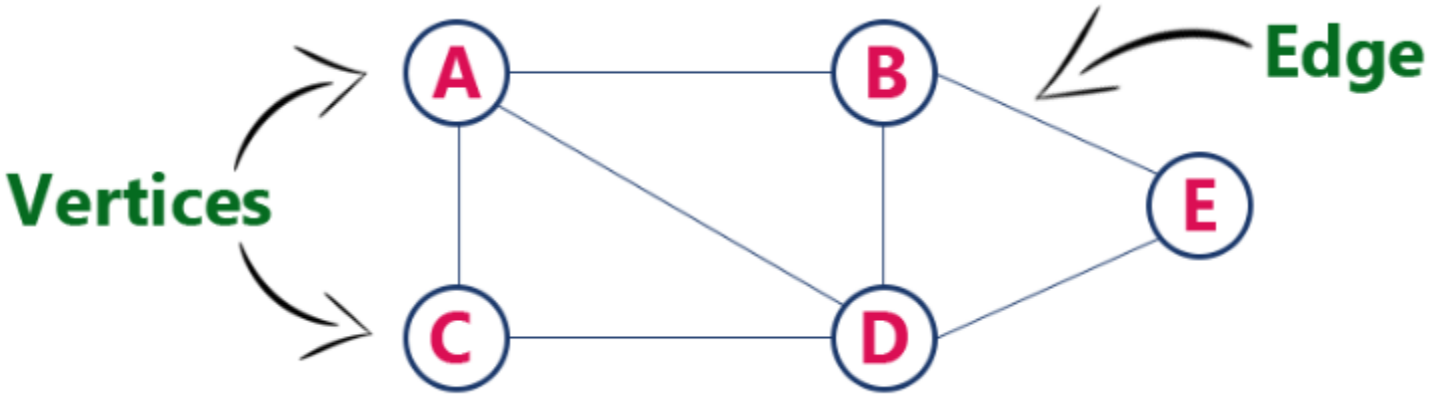
1,5,6,2,8,12,14,15,20,7,10

Unit-4
Graphs

- Graph Theory Terminology-Graph Representation-Graph Operations-Depth First Search-Breadth First Search-Connected Components-Spanning Trees-Biconnected Components-Minimum Cost Spanning Trees-Kruskal's Algorithm-Prism's Algorithm-Shortest Paths-Transitive Closure-All-Pairs Shortest Path-Warshall's Algorithm.

Unit-4
Graphs

- Graph is a non-linear data structure. It contains a set of points known as nodes (or vertices) and a set of links known as edges (or Arcs). Here edges are used to connect the vertices. A graph is defined as follows...
- **Graph is a collection of vertices and arcs in which vertices are connected with arcs**
- **Or**
- **Graph is a collection of nodes and edges in which nodes are connected with edges**
- Generally, a graph **G** is represented as **G = (V , E)**, where **V is set of vertices** and **E is set of edges**.



Unit-4
Graph Terminology

Vertex:

- Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**.

Edge:

- An edge is a connecting link between two vertices. **Edge** is also known as **Arc**.
- Edges are three types.
- **Undirected Edge -** An undirected egde is a bidirectional edge.
- **Directed Edge -** A directed egde is a unidirectional edge.
- **Weighted Edge -** A weighted egde is a edge with value (cost) on it.

Undirected Graph:

- A graph with only undirected edges is said to be undirected graph.

Directed Graph:

- A graph with only directed edges is said to be directed graph.

Origin and destination vertices:

- In a directed graph if there Is a edge between two vertices then starting vertex is origin vertex and the last vertex is called destination vertex.

Mixed Graph:

- A graph with both undirected and directed edges is said to be mixed graph.

Unit-4
Graph Terminology

Adjacent:

- If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.

Degree

- Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

- Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

- Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

- If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.

Self-loop

- Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.

Unit-4
Graph Terminology

Simple Graph

- A graph is said to be simple if there are no parallel and self-loop edges.

Multi graph:

- In a graph if there is either self loop or parallel edges or both then such a graph can be called as multi graph.

Path:

- A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

Cycle:

In a graph every closed path is called as cycle

Closed path:

In a graph if a path contains same end points then such a path is called as closed path.

Unit-4
Graph Terminology

Types of Graphs:

1.Null graph:

A graph with no edges.

2.Trivial graph:

A graph having only one vertex.

3.Regular graph:

In a graph every vertex has same degree then such a graph is called as regular graph.

4.Sub graph:

In a graph a part of the graph can be called as a sub graph.

5.Complete graph:

In a graph if every vertex is connected with every other vertex in the graph,then such a graph Is called as complete graph.

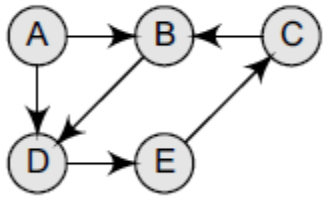
Unit-4
Graph Representations

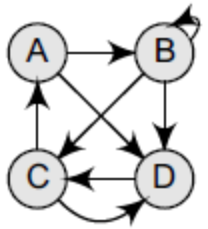
- Graph data structure is represented using following representations...
- **Adjacency Matrix**
- **Adjacency List**

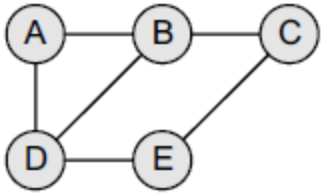
Unit-4
Graph Representations

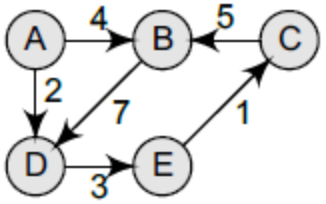
Adjacency Matrix:

- An adjacency matrix is used to represent which nodes are adjacent to one another. By definition, two nodes are said to be adjacent if there is an edge connecting them.





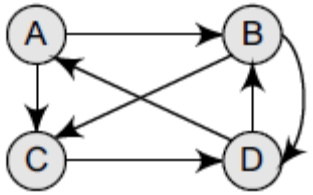


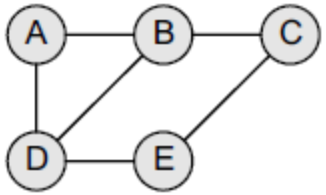


Unit-4
Graph Representations

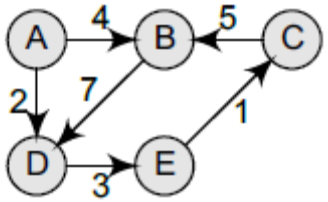
Adjacency List Representation

- An adjacency list is another way in which graphs can be represented in the computer's memory.





(Undirected graph)



(Weighted graph)

Unit-4
Graph Traversal

- Graph traversal is a technique used for a searching vertex in a graph. The graph traversal is also used to decide the order of vertices is visited in the search process. A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path..
- There are two graph traversal techniques and they are

1.DFS (Depth First Search)

2.BFS (Breadth First Search)

Unit-4

Graph Traversal DFS (Depth First Search/traversal)

DFS (Depth First Search)

- DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

Algorithm for depth first search traversal:

Step 1 - Define a Stack of size total number of vertices in the graph.

Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.

Step 3 - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.

Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.

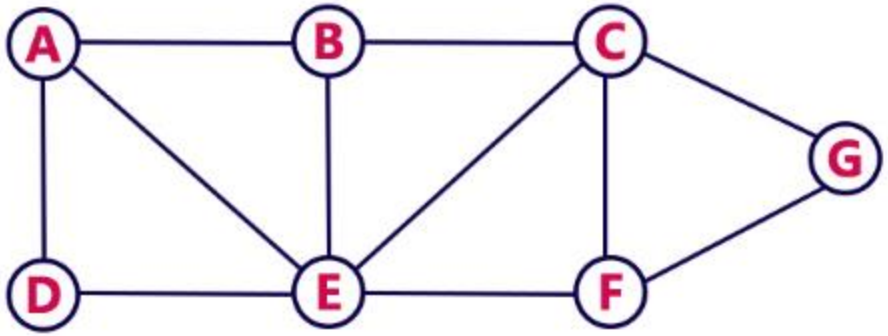
Step 5 - When there is no new vertex to visit then use **back tracking** and pop one vertex from the stack.

Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Graph Traversal DFS (Depth First Search/traversal)

Example:

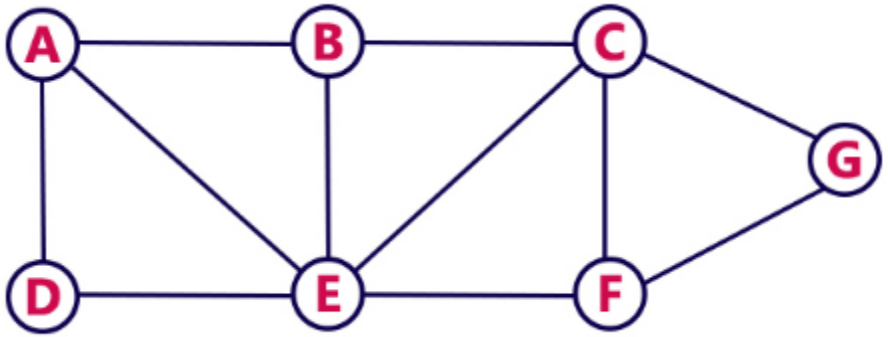


Graph Traversal BFS (Breadth First Search)

- BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.
- We use the following steps to implement BFS traversal...
- **Step 1 -** Define a Queue of size total number of vertices in the graph.
- **Step 2 -** Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3 -** Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4 -** When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5 -** Repeat steps 3 and 4 until queue becomes empty.
- **Step 6 -** When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Unit-4

Graph Traversal BFS (Breadth First Search)

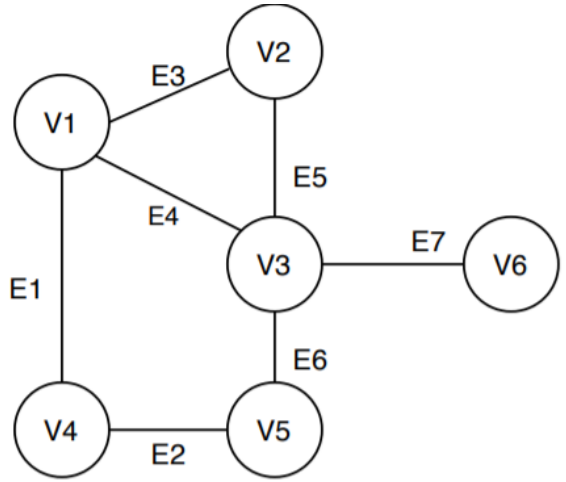


- A connected component or simply component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.

Or

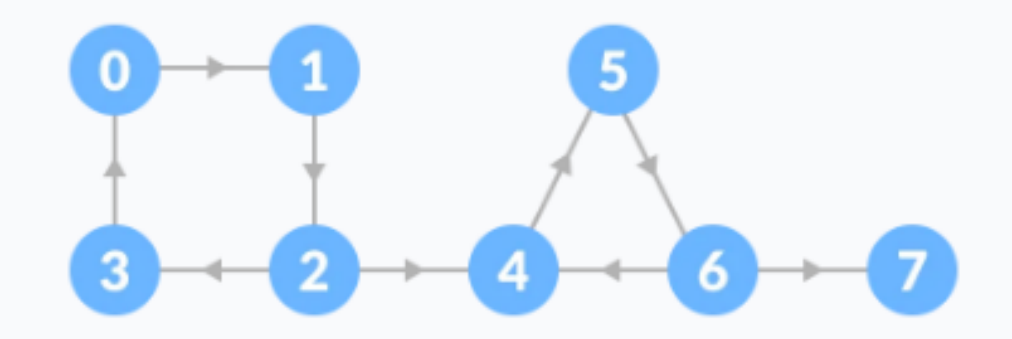
- A set of nodes forms a connected component in an undirected graph if any node from the set of nodes can reach any other node by traversing edges. **The main point here is reachability.**

- Ex:**One Connected Component**



Strongly connected graph

- A directed graph is strongly connected if there is a path between all pairs of vertices.



BI-CONNECTED components

- **BI-CONNECTED components**
- A vertex v of G is called an articulation point, if removing v along with the edges incident on v, results in a graph that has at least two connected components.
- A bi-connected graph (shown in Fig. 13.10) is defined as a connected graph that has no articulation vertices. That is, a bi-connected graph is connected and non-separable in the sense that even if we remove any vertex from the graph, the resultant graph is still connected.

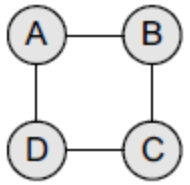


Figure 13.10 Bi-connected graph

BI-CONNECTED components

By definition,

- A bi-connected undirected graph is a connected graph that cannot be broken into disconnected pieces by deleting any single vertex.
- In a bi-connected directed graph, for any two vertices v and w, *there are two directed paths from v to w which have no* vertices in common other than v and w.
- Note that the graph shown in Fig. 13.9(a) is not a bi-connected graph, as deleting vertex C from the graph results in two disconnected components of the original graph (Fig. 13.9(b)).

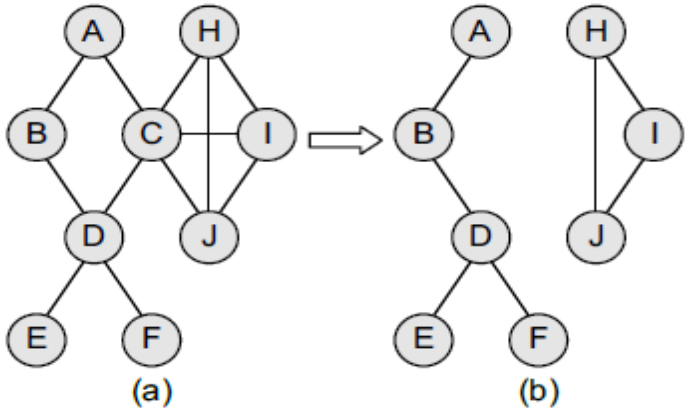
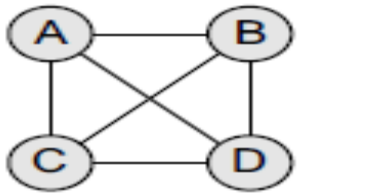


Figure 13.9 Non bi-connected graph

Spanning Trees

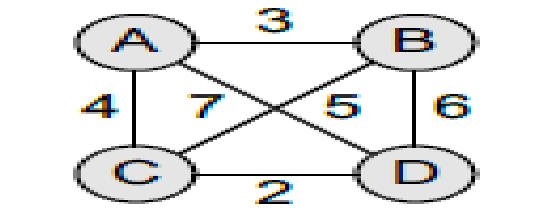
- A spanning tree of a connected, undirected graph G is a sub-graph of G which is a tree that connects all the vertices together. A graph G can have many different spanning trees.
- Consider an unweighted graph G given below



(Unweighted graph)

Spanning Trees

- A *minimum spanning tree (MST) is defined as a spanning tree with weight less than or equal* to the weight of every other spanning tree. In other words, a minimum spanning tree is a spanning tree that has weights associated with its edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.



(Weighted graph)

Kruskal's Algorithm

- Kruskal's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Kruskal's algorithm, the given graph must be weighted, connected and undirected.
- The Kruskal's algorithm is given as follows.

Algorithm:

Step- 1 : Remove all loops and Parallel Edges

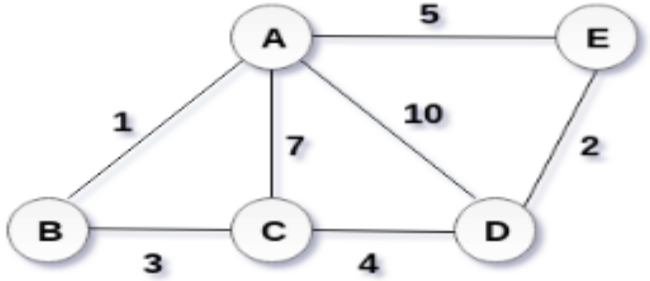
Step- 2 : Arrange all edges in their increasing order of weight

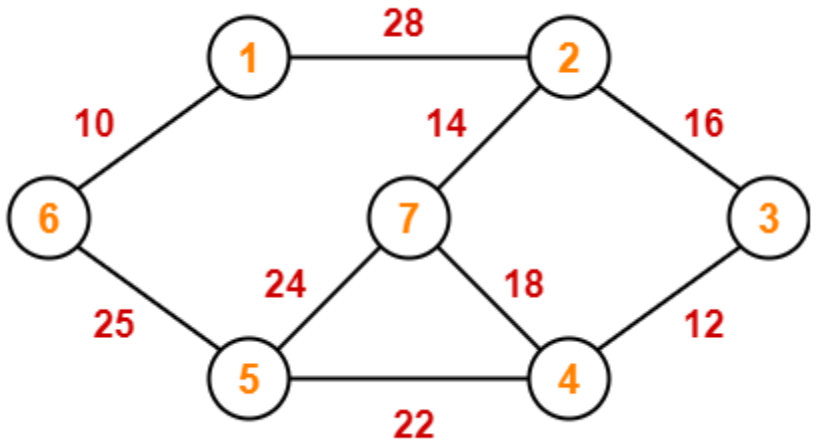
Step- 3 (a) Add the edge which has the least weightage

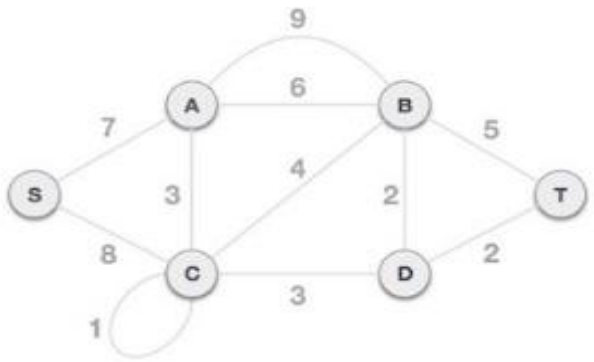
(b) If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

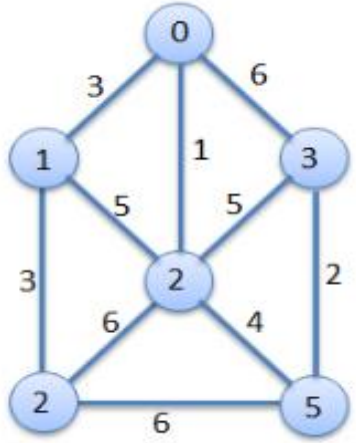
Step-4: Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

Kruskal's Algorithm









Prim's Spanning Tree Algorithm

- Prim's algorithm to find minimum cost spanning tree (as Kruskal's algorithm) uses the greedy approach
- Prim's algorithm, in contrast with Kruskal's algorithm, treats the nodes as a single tree and keeps on adding new nodes to the spanning tree from the given graph.

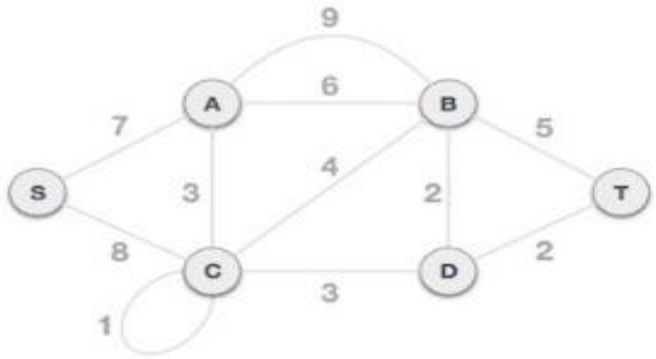
Algorithm:

Step 1 - Remove all loops and parallel edges

Step 2 - Choose any arbitrary node as root node

Step 3 - Check outgoing edges and select the one with less cost

Step 4 – repeat step3 until all the vertices in the graph are visited.



Warshall's Algorithm

- If a graph G is given as G=(V, E), where V is the set of vertices and E is the set of edges, the path matrix of G can be found as, P = A + A2 + A3 + … + An.
- This is a lengthy process, so Warshall has given a very efficient algorithm to calculate the path matrix.
- The path matrix Pn can be calculated with the formula given as:

$$P_k[i][j] = P_{k-1}[i][j] \vee (P_{k-1}[i][k] \wedge P_{k-1}[k][j])$$

Shortest paths –Dijkstras algorithm

- Given a graph G and a source node A, the algorithm is used to find the shortest path (one having the lowest cost) between A (source node) and every other node.
- Dijkstra's algorithm is used to find the length of an optimal path between two nodes in a graph. The term optimal can mean anything, shortest, cheapest, or fastest.

Dijkstra's algorithm:

1. Select the source node also called the initial node
2. Define an empty set N that will be used to hold nodes to which a shortest path has been found.
3. Label the initial node with 0 , and insert it into N.
4. Repeat Steps 5 to 7 until the destination node is in N or there are no more labelled nodes in N.
5. Consider each node that is not in N and is connected by an edge from the newly inserted node.
6. (a) If the node that is not in N has no label then SET the label of the node = the label of the newly inserted node + the length of the edge.

(b) Else if the node that is not in N was already labelled, then SET its new label = minimum (label of newly inserted vertex + length of edge, old label)

7. Pick a node not in N that has the smallest label assigned to it and add it to N.

Shortest paths –Dijkstras algorithm

Example 13.10 Consider the graph G given in Fig. 13.36. Taking D as the initial node, execute the Dijkstra's algorithm on it.

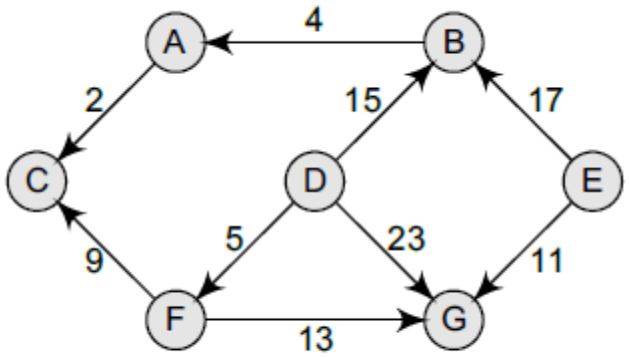


Figure 13.36 Graph G

All pairs shortest path -floyd warshall algorithm

- The problem: find the shortest path between every pair of vertices of a graph
- The graph: may contain negative edges but no negative cycles
- A representation: a weight matrix where
  W(i,j)=0 if i=j.
  W(i,j)=∞ if there is no edge between i and j.
  W(i,j)="weight of edge"
- Let $D^{(k)}[i,j]$=weight of a shortest path from v_i to v_j using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices in the path
  - $D^{(0)}=W$
  - $D^{(n)}=D$ which is the goal matrix
- How do we compute $D^{(k)}$ from $D^{(k-1)}$?

all pairs shortest path -floyd warshall algorithm

Case 1: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices does not use v_k. Then $D^{(k)}[i,j] = D^{(k-1)}[i,j]$.

Case 2: A shortest path from v_i to v_j restricted to using only vertices from $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices does use v_k. Then $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$.

Since

$D^{(k)}[i,j] = D^{(k-1)}[i,j]$ or

$D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$.

We conclude:

$$\boldsymbol{D^{(k)}[i,j] = \min\{ D^{(k-1)}[i,j],\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \}.}$$

A

Searching and Sorting

- Searching means to find whether a particular value is present in an array or not.
- If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.
- However, if the value is not present in the array, the searching process displays an appropriate message and in this case searching is said to be unsuccessful.
- There are two popular methods for searching the array elements:

  1.linear search and

  2.binary search.
- The algorithm that should be used depends entirely on how the values are organized in the array.
- For example, if the elements of the array are arranged in ascending order, then binary search should be used, as it is more efficient for sorted lists in terms of complexity.

Searching and Sorting-Linear Search

Step 1: Start
Step 2: Read n, a[i], key values as integers
Step 3: Search the list
While (i<n)
If(a[i] == key)
Break;
i = i + 1
Repeat step 3
Step 4: Successful search
if(key is available) then
Printf(“Element found in the list”)
else
printf (“Element not found in the list”)
Step 6: Stop

```
void main()
{
int i, a[20], n, key, flag = 0;
printf(“Enter the size of an array \n”);
scanf(“%d”, &n);
printf(“Enter the array elements”);
for(i = 0; i < n; i++)
scanf(“%d”, &a[i]);
printf(“Enter the key elements”);
scanf(“%d”, &key);
for(i = 0; i < n; i++)
{if(a[i] == key)
{flag = 1;
break; } }
if(flag == 1)
printf(“The key elements is found at location %d”, i + 1);
else
printf(“The key element is not found in the array”);
}
```