**(Time: 2½ hours)**

**Total Marks: 75**

N. B.: (1) **All** questions are **compulsory**.

(2) Make **suitable assumptions** wherever necessary and **state the assumptions** made.

(3) Answers to the **same question** must be **written together**.

(4) Numbers to the **right** indicate **marks**.

(5) Draw **neat labeled diagrams** wherever **necessary**.

(6) Use of **Non-programmable** calculators is **allowed**.

**1. Attempt *any three* of the following:** **15**

a. **List and Explain the different Asymptotic notations used in data structures.**

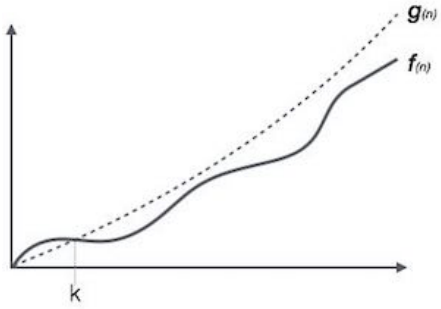
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

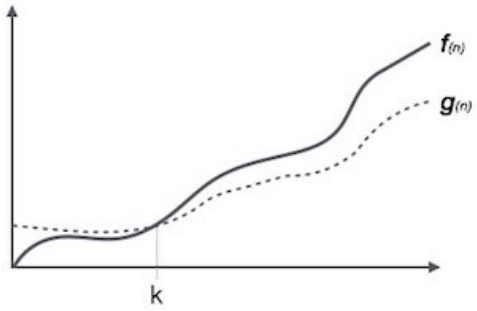



For example, for a function *f*(n)

O(*f*(n)) = { *g*(n) : there exists c > 0 and $n_0$ such that $f(n) \leq c.g(n)$ for all $n > n_0$. }

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

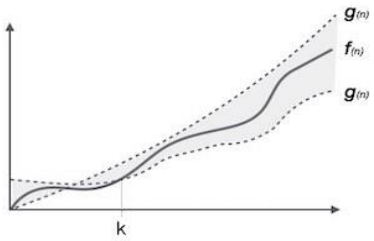



For example, for a function *f*(n)

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and $n_0$ such that *g*(n) ≤ c.*f*(n) for all n > $n_0$. }

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −




θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > $n_0$. }

b. **What are the different ways in which data structures are classified? Explain in detail.**

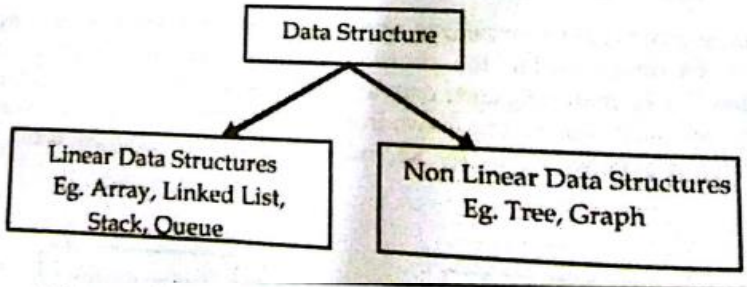
Data structures can be classified in several ways. These classifications are:

- Linear and Non-Linear Data Structures
- Static and Dynamic Data Structures
- Homogeneous and Non-Homogeneous Data Structures

### Linear and Non-Linear Data Structures

**Linear Data Structure:** The elements in a linear data structure form a linear sequence. Example of the linear data structures are: Array, Linked list, Queue, Stack etc.

**Non-Linear Data Structure:** The elements in a non-linear data structure do not form any linear sequence. For example, Tree and Graph.




### Static and Dynamic Data Structures

**Static Data Structure:** Static data structures are those whose memory occupation is fixed. The memory taken by these data structures cannot be increased or decreased at run time. Example of the static data structure is an **Array**. The size of an array is declared at the compile time and this size cannot be changed during the run time.

**Dynamic Data Structure:** Dynamic data structures are those whose memory occupation is not fixed. The memory taken by these data structures can be increased or decreased at run time. Example of the dynamic data structure is **Linked List**. The size of linked list can be changed during the run time.

Other data structures like stack, queue, tree, and graph can be static or dynamic depending on, whether these are implemented using an array or a linked list.

### Homogeneous and Non-Homogeneous Data Structures

**Homogeneous Data Structure:** Homogeneous data structures are those in which data of same type can be stored. Example of the homogeneous data structure is an **Array**.

**Non-Homogeneous Data Structure:** Non-Homogeneous data structures are those in which data of different types can be stored. Example of the non-homogeneous data structure is **linked list**.

c. **What do you mean by complexity of an algorithm.? Explain its types.**

Complexity is the time and space requirement of the algorithm. If time and space requirement of the algorithm is more, then complexity of the algorithm is more and if time and space requirement of the algorithm in less, complexity of that algorithm is less.

Out of the two factors, time and space, the space requirement of the algorithm is not a very important factor because it is available at very low cost. Only the time requirement of the algorithm is considered an important factor to find the complexity. Because of the importance of time in finding the complexity, it is sometimes termed as **time complexity**.

As the time requirement of the algorithm is dependent upon the input size irrespective of the other factors like machine/processor, time complexity is measured in terms of input size $n$. If the input size to the algorithm is more, the complexity will be more and if the input size to the algorithm is less, the complexity will be less.

For example, consider an algorithm which sorts an array of size 2000, will definitely take more time than to sort an array of size 20. Thus, we express the time complexity in terms of input size $n$.

Hence, to calculate the time complexity of an algorithm, the basic approach is to **count the number of times, a key operation is executed**. Here, the key operation is the major operation that is executed maximum number of times in the algorithm. For example, in a searching algorithm the key operation is comparison between the elements. We only count the key operation because most of the time taken by the algorithm is consumed by key operation. The time complexity is expressed as a function of key operation performed in that algorithm.

As the complexity of an algorithm is dependent upon the input size, still complexity can be divided into three types:

- Worst case complexity
- Best case complexity
- Average case complexity

For a particular input, the result can be obtained in minimum time or maximum time or average time. For instance, consider the linear search (detail of linear search is in the chapter of Array) in which we find the desired element by comparing it with all the elements of the list (say $n$ number of elements in the list) starting from the first element of the list.

If we get the desired element at first position then number of comparisons will be 1. So, complexity is 1. If we get the desired element at the last position then number of comparisons will be $n$ so complexity is $n$.

If we get the desired element at any other position then the complexity will be between 1 and $n$. So, the complexity can be different (maximum, minimum, or average) for a particular problem. Let us define the types of complexities.

**Worst Case Complexity**: If the running time of the algorithm is longest for all the inputs then the complexity is called worst case complexity. In this type of complexity, the key operation is executed maximum number of times. Worst case is the upper bound of complexity and in certain application domains e.g. air traffic control, medical surgery, the worst case complexity is of crucial/high importance.

**Best Case Complexity:** If the running time of the algorithm is shortest for all the inputs then the complexity is called best case complexity. In this type of complexity, the key operation is executed minimum number of times.

**Average Case Complexity:** If the running time of the algorithm falls between the worst case and the best case then the complexity is called average case complexity. Average case complexity of an algorithm is difficult to find. To calculate the average case complexity of an algorithm, we have to take some assumptions.

d. **Write an algorithm to for binary search in an array.**

**Algorithm: Search the position of an element '$Data$' in an array '$S$' with lower bound '$lb$' and upper bound '$ub$'.**

Step 1: Set $Start = lb$, $End = ub$

Step 2: Repeat Steps 3 to 5 While $Start < End$

Step 3: Set $Middle = Integer\left(\frac{Start+End}{2}\right)$ *// Round off the value*

Step 4: If $S[Middle] = Data$ Then

Print: "Element is found at position": $Middle$

Exit

[End If]

```
Step 5:     If S[Middle] < Data Then
                Set Start = Middle + 1
            Else
                Set End = Middle − 1
            [End If]
        [End Loop]
Step 6:     Print: "Element does not exist in the array"
Step 7:     Exit
```

e. **What are sparse matrix? Explain different types of sparse matrix.**

A matrix M is said to be sparse matrix if majority of its elements are meaningless. Such kind of matrices contains high density of meaningless elements or we can say that the sparse matrix has a very few elements which are significant. If we consider zero as meaningless elements then the sparse matrix is the matrix which has majority of zero elements. The below shown array of order $6 \times 5$ is a sparse matrix because maximum of its elements are zero.

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 7 & 0 & 0 & 5 \end{bmatrix}$$

**A Sparse Matrix**

**Diagonal Matrix**

A matrix $M$ is said to be diagonal matrix if and only if $M[i,j] = 0$ for $i \neq j$. That is, all the non-diagonal elements are zero.

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**A Diagonal Matrix**

**Upper Triangular Matrix**

A matrix $M$ is said to be upper triangular matrix if and only if $M[i,j] = 0$ for $i > j$. That is, all the elements of the matrix below the diagonal elements are zero. Matrix is said to be strictly upper triangular if all the diagonal elements are also zero along with the elements below the diagonal.

$$\begin{bmatrix} 5 & 8 & 7 & 9 \\ 0 & 3 & 2 & 16 \\ 0 & 0 & 5 & 7 \\ 0 & 0 & 0 & 8 \end{bmatrix}$$

**An Upper Triangular Matrix**

$$\begin{bmatrix} 0 & 8 & 7 & 9 \\ 0 & 0 & 2 & 16 \\ 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

**A Strictly Upper Triangular Matrix**

**Lower Triangular Matrix**

A matrix $M$ is said to be lower triangular matrix if and only if $M[i, j] = 0$ for $i < j$. That is, all the elements of the matrix above the diagonal elements are zero. A special case is strictly lower triangular matrix in which all the diagonal elements are also zero along with the elements above the diagonal.

$$\begin{bmatrix} 5 & 0 & 0 & 0 \\ 1 & 3 & 0 & 0 \\ 8 & 9 & 5 & 0 \\ 3 & 2 & 7 & 8 \end{bmatrix}$$

**A Lower Triangular Matrix**

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 8 & 9 & 0 & 0 \\ 3 & 2 & 7 & 0 \end{bmatrix}$$

**A Strictly Lower Triangular Matrix**

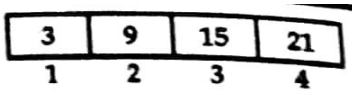
All the above discussed types of matrices fit into the category of sparse matrices. These kinds of matrices are generated from scientific applications as these applications generate a huge data that may have hundreds of rows and columns with a very few significant entries.

f. **Explain with the help of an example how to merge two sorted arrays .**

In the second approach, the third array will be sorted while merging the given sorted arrays. In this approach, the elements of the given arrays are compared and based on the comparison, it is decided that which element will go into the third array. Now, consider two arrays **A1** and **A2** as shown below in figure **c** and **d** respectively. These arrays are to be merged into a third array **A3** which is required to be in the sorted order.

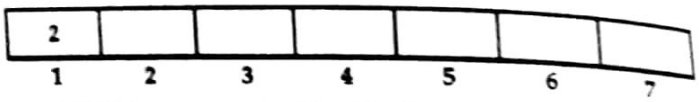
| 2 | 7 | 8 |
|---|---|---|
| 1 | 2 | 3 |

**c: A Sorted Array 'A1' with 3 Elements**

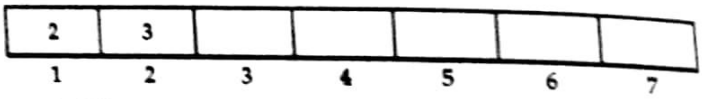



**d: A Sorted Array 'A2' with 4 Elements**

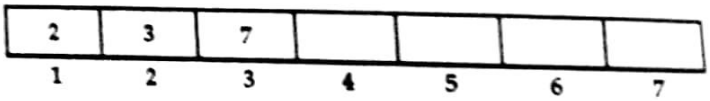
While merging the arrays $A1$ and $A2$, initially, element $A1[1]$ is compared with $A2[1]$. As $A1[1] < A2[1]$, so $A1[1]$ will go at the first location of the third array $A3$ and will occupy the $A3[1]$ position as shown below:



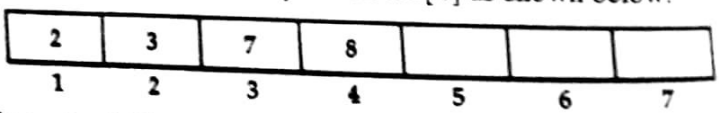

Then, the element $A1[2]$ is compared with $A2[1]$. As $A2[1] < A1[2]$, so $A2[1]$ will be placed at second position in the third array $A3$ i.e. $A3[2]$ as shown below:



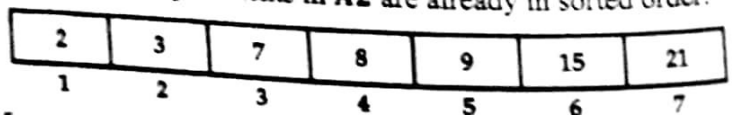

Now, the element $A1[2]$ is compared with $A2[2]$. As $A1[2] < A2[2]$, so $A1[2]$ will b placed in the third array $A3$ at the third position $A3[3]$ as shown below:




Now, the element $A1[3]$ is compared with $A2[2]$. As $A1[3] < A2[2]$, so $A1[3]$ wi placed in the third array $A3$ at the forth position $A3[4]$ as shown below:




Now, all the elements of **A1** are transferred to array **A3** and a few elements are left second array **A2** . The remaining elements of the array **A2** will be copied as such end of the array **A3** as shown below. The reason for copying the remaining elem **A2** to **A3** is that the remaining elements in **A2** are already in sorted order:



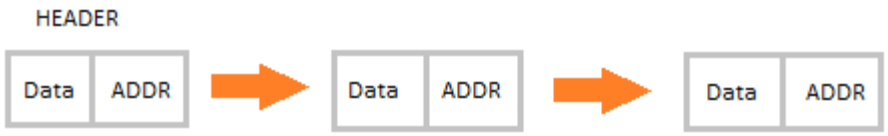

**2. Attempt *any three* of the following:** **15**

a. **Explain the structure advantages /disadvantages and types of linked list.**

Linked List is a very commonly used linear data structure which consists of group of nodes in a sequence.

Each node holds its own data and the address of the next node hence forming a chain like structure.

Linked Lists are used to create trees and graphs.




---

Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

---

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Types of Linked Lists

There are 3 different implementations of Linked List available, they are:

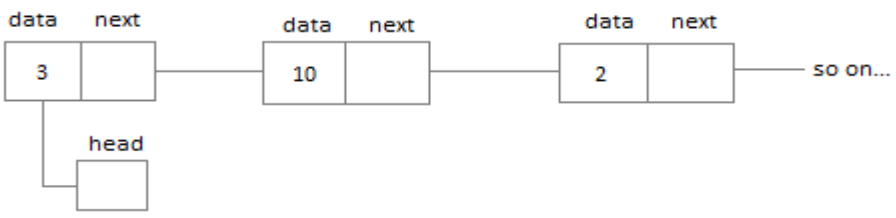
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Let's know more about them and how they are different from each other.

Singly Linked List

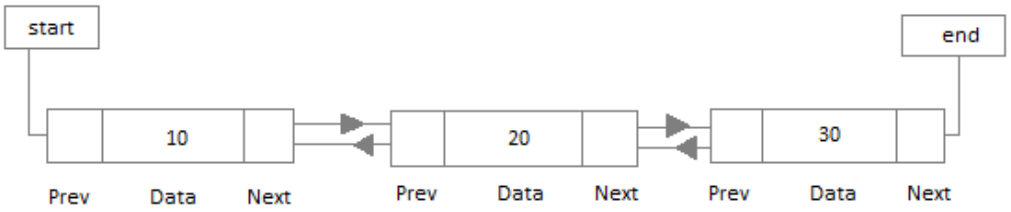
Singly linked lists contain nodes which have a data part as well as an address part i.e. next, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are insertion, deletion and traversal.

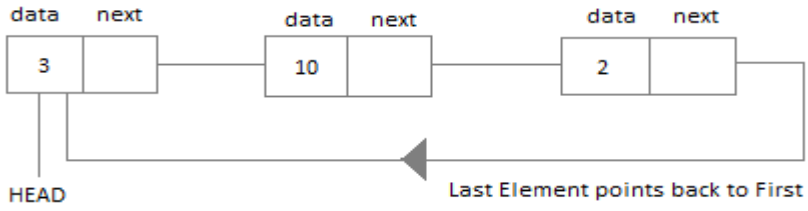



Doubly Linked List

In a doubly linked list, each node contains a data part and two addresses, one for the previous node and one for the next node.




Circular Linked List
In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.




b. **Write the algorithm for insertion of a node at the given position and deletion at the end in linked list.**

**Algorithm: Insertion of a given element *'Item'* after a specific element *'Data'* in the linked list.**

Step 1: If ***Free = Null*** Then
Print: "Overflow: No free space available for insertion"
Exit
[End If]

Step 2: Set ***Pointer = Begin***

Step 3: Repeat While ***Pointer ≠ Null AND Pointer → Info ≠ Data***
Set ***Pointer = Pointer → Next***
[End Loop]

Step 4: If ***Pointer = Null*** Then
Print: "The node containing element ***Data*** is not present, so insertion is

not possible."

Else

Allocate space to node ***New***

Set ***New*** = ***Free***, ***Free*** = ***Free*** → ***Next***, ***New*** → ***Info*** = ***Item***

Set ***New*** → ***Next*** = ***Pointer*** → ***Next***

Set ***Pointer*** → ***Next*** = ***New***

[End If]

Step 5: Exit

---

**Algorithm: Deleting the last node of the linked list.**

Step 1: If ***Begin*** = ***Null*** Then

Print: "Linked List is Empty"

Exit

[End If]

Step 2: If ***Begin*** → ***Next*** = ***Null*** Then *//Linked List is having only one n*

Set ***Data*** = ***Begin*** → ***Info***

Deallocate memory held by ***Begin***

(***Begin*** → ***Next*** = ***Free*** And ***Free*** = ***Begin***)

Set **Begin = Null** *//Indicating Empty Linked L*

---

Exit

[End If]

Step 3: Set ***Pointer1*** = ***Begin*** And ***Pointer2*** = ***Begin*** → ***Next***

Step 4: Repeat While ***Pointer2*** → ***Next*** ≠ ***Null***

Set ***Pointer1*** = ***Pointer2*** And ***Pointer2*** = ***Pointer2*** → ***Next***

[End Loop]

Step 5: Set ***Pointer1*** → ***Next*** = ***Pointer2*** → ***Next***

Step 6: Set ***Data*** = ***Pointer2*** → ***Info***

Step 7: Deallocate memory held by ***Pointer2***

(***Pointer2*** → ***Next*** = ***Free*** And ***Free*** = ***Pointer2***)

Step 8: Exit

c. **Give an algorithm to copy one link list into another link list.**

**Algorithm: Copying the elements of a linked list to another linked list.**

Step1: If $Begin1 = Null$ Then
Print: "Source List is Empty "
Exit
[End If]

Step 2: Set $Begin2 = Null$

Step 3: If $Free = Null$ Then
Print: "Free Space not Available"
Exit
Else
Allocate memory to the node $New$
Set $New = Free$ And $Free = Free \rightarrow Next$
[End If]

Step 4: Set $New \rightarrow Info = Begin1 \rightarrow Info$ And $New \rightarrow Next = N$

Step 5: Set $Begin2 = New$

Step 6: Set $Pointer1 = Begin1 \rightarrow Next$ And $Pointer2 = Begin2$

Step 7: Repeat While $Pointer1 \neq Null$ AND $Free \neq Null$
a. Allocate memory to the node $New$
($New = Free$ And $Free = Free \rightarrow Next$)
b. Set $New \rightarrow Info = Pointer1 \rightarrow Info$ And $New \rightarrow Next = Null$
c. Set $Pointer2 \rightarrow Next = New$
d. Set $Pointer1 = Pointer1 \rightarrow Next$ And $Pointer2 = New$
[End Loop]

Step 8: If $Pointer1 = Null$ Then
Print: "List Copied Successfully"
Else
Print: "Not Enough space Available to perform Copy operation"
[End If]

Step 9: Exit

d. **Give the algorithm to insert an element at the beginning and end of circular linked list.**

**Algorithm: Insertion of an element *'Data'* at the beginning of the circular linked list.**

Step 1: If $Free = Null$ Then
    Print: "No free space available"
    Exit
    [End If]

Step 2: Set $New = Free$ And $Free = Free \rightarrow Next$ *//Allocate memory to node* $New$

Step 3: Set $New \rightarrow Info = Data$

Step 4: If $Begin = Null$ Then
    Set $Begin = New$
    Set $New \rightarrow Next = Begin$
    Exit
    [End If]

Step 5: Set $Pointer = Begin$

Step 6: Repeat While $Pointer \rightarrow Next \neq Begin$
    Set $Pointer = Pointer \rightarrow Next$
    [End Loop]

Step 7: Set $New \rightarrow Next = Begin$

Step 8: Set $Begin = New$

Step 9: Set $Pointer \rightarrow Next = Begin$

Step 10: Exit

**Algorithm: Insertion of an element '*Item*' at the end of the circular linked list.**

Step 1: If $Free = Null$ Then
Print: "No Free Space Available for Insertion"
Exit
[End If]

Step 2: Allocate memory to node $New$
Set $New = Free$ And $Free = Free \rightarrow Next$

Step 3: Set $New \rightarrow Info = Item$

Step 4: If $Begin = Null$ Then
Set $Begin = New$ And $New \rightarrow Next = Begin$
Exit
[End If]

Step 5: Set $Pointer = Begin$

Step 6: Repeat while $Pointer \rightarrow Next \neq Begin$
Set $Pointer = Pointer \rightarrow Next$
[End Loop]

Step 7: Set $Pointer \rightarrow Next = New$ And $New \rightarrow Next = Begin$

Step 8: Exit

e. **Write and explain the algorithm for inserting at the beginning in two way linked list.**

A Simplified Appr...

**Algorithm: To insert a new node at the beginning of a two-way linked list.**

Step 1: If $Free = Null$ Then
Print: "Free space not available"
Exit
[End If]

Step 2: Allocate memory to node $New$
(Set $New = Free$ And $Free = Free \rightarrow Next$)

Step 3: Set $New \rightarrow Pre = Null$ And $New \rightarrow Info = Data$

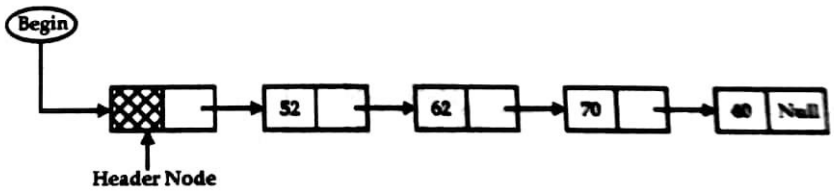
Step 4: If $Begin = Null$ Then
Set $New \rightarrow Next = Null$ And $End = New$
Else
Set $New \rightarrow Next = Begin$ And $Begin \rightarrow Pre = New$
[End If]

Step 5: Set $Begin = New$

Step 6: Exit

f. **Explain the different categories of header linked list.**

A header linked list is a special kind of linked list which **contains a special node at the** beginning of the list. This special node is known as head node. This head node contains important information regarding the linked list. This information may be **total number of nodes** in the linked list, some description for the user like **creation date**, **modification date** or information about, whether the data in the list is **sorted or unsorted**. The header linked list is shown below:



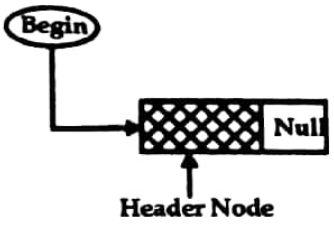

**A Header Linked List**

Depending upon the application, there are some categories of **header linked list**:

- Grounded Header Linked List
- Circular Header Linked List
- Two-way Header Linked List
- Circular Two-way Header Linked List

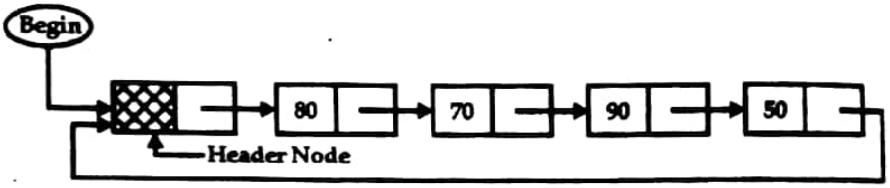
A **grounded header linked list** is a list, in which last **node** of the list contains the ***Null*** in its ***Next*** pointer field. The **header linked list** shown above is a **grounded header linked** list.

If grounded header linked list is empty then the ***Null*** value will be stored in the ***Next*** pointer field of the head node as shown in the figure below:



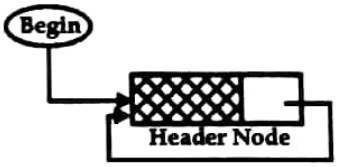

**An Empty Grounded Header Linked List**

A **circular header linked list** is a list in which last node of the list points back to the header node i.e. ***Next*** pointer field of the last node contains the address of the header node. Circular header linked list can be shown diagrammatically as:



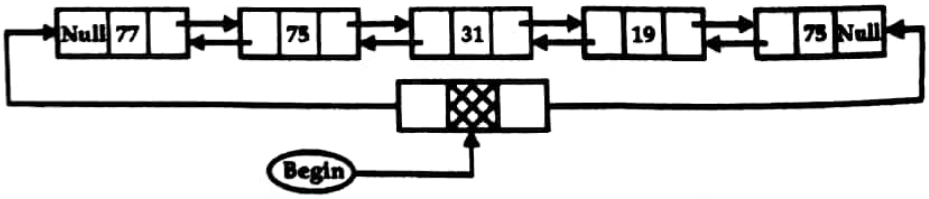

**A Circular Header Linked List**

If the circular header linked list is empty then this situation is handled by storing the address of the head node in the ***Next*** pointer field of head node itself. The case of empty circular header linked list can be shown diagrammatically as:



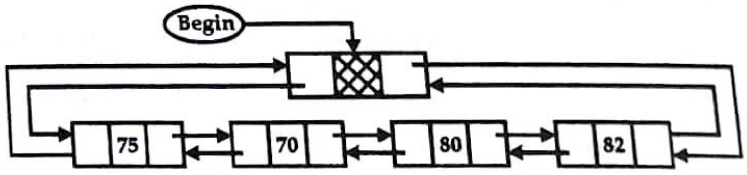

**An Empty Circular Header Linked List**

In general, a header node can be inserted in any type of linked list either one-way or two-way linked list. A two-way header linked list can be shown as:




**A Two-way Header Linked List**

A two-way **circular Header linked list** can be shown as,




**Circular Two-way Header Linked List**

**3. Attempt *any three* of the following:** **15**

**a. Write the algorithm for push and pop operation of the stack.**

**Algorithm: Push Operation - Insert a new element $'Data'$ at the top of the stack represented by an array $'S'$ of size $'Max'$ with a stack index variable $'Top'$ pointing to the topmost element of the stack.**

Step 1: If $Top = Max$ Then
Print: "Stack is already full, Overflow Condition"
Exit
[End If]

Step 2: Set $Top = Top + 1$

Step 3: Set $S[Top] = Data$

Step 4: Exit

---

**Algorithm: Pop Operation - Delete an element from the stack represented by an array $'S'$ and returns the element $'Data'$ which is at the top of the stack.**

Step 1: If $Top = NULL$ Then
Print: "Stack is already empty, Underflow Condition"
Exit
[End If]

Step 2: Set $Data = S[Top]$

Step 3: Set $Top = Top - 1$

Step 4: Exit

b. **Write the algorithm for converting infix to postfix and Convert the following expression to postfix notation using stack**
**I=(6+2)*5-8/4**

**Algorithm: Convert an arithmetic expression *'I'* written in infix notation into its equivalent postfix expression *'P'*.**

Step 1: Push a left parenthesis ( onto the stack.

Step 2: Append a right parenthesis ) at the end of the given expression ***I***.

Step 3: Repeat Steps from 4 to 8 by scanning ***I*** character by character from left to right until the stack is empty.

Step 4: If the current character in ***I*** is a white space, simply ignore it.

Sep 5: If the current character in ***I*** is an operand, write it as the next element of the postfix expression ***P***

Step 6: If the current character in ***I*** is a left parenthesis ( , push it onto the stack.

Step 7: If the current character in ***I*** is an operator Then

a. Pop operators (if there is any) at the top of stack while they have **equal or higher precedence** than the current operator and put the popped operators in the postfix expression ***P***

b. Push the currently scanned operator on the stack.

Step 8: If the current character in ***I*** is a right parenthesis Then

a. Pop operators from the top of the stack and insert them in the postfix expression ***P*** until a left parenthesis is encountered at the top of the stack.

b. Pop and discard left parenthesis ( from the stack.

[End Loop]

Step 9: Exit

$I = ( 6 + 2 ) * 5 - 8 / 4 )$

| Character Scanned | Status of Stack | Postfix Expression 'P' |
|---|---|---|
| | ( | |
| ( | (( | |
| 6 | (( | 6 |
| + | (( + | 6 |
| 2 | (( + | 6 2 |
| ) | ( | 6 2 + |
| * | ( * | 6 2 + |
| 5 | ( * | 6 2 + 5 |
| - | ( - | 6 2 + 5 * |
| 8 | ( - | 6 2 + 5 * 8 |
| / | ( - / | 6 2 + 5 * 8 |
| 4 | ( - / | 6 2 + 5 * 8 4 |
| ) | Null | 6 2 + 5 * 8 4 / - |

c. **Write the Algorithm for evaluating a postfix expression using stack and give an example.**

### 4.4.1.4 Evaluation of Postfix Notation

**Algorithm: Evaluate an arithmetic expression *'P'* written in postfix notation and calculates the result of the expression in variable *'Value'*.**

**Step 1:** Scan ***P*** from left to right and Repeat Steps 2 and 3 for each scanned character until end of the expression.

**Step 2:** If scanned character is an **operand**, push it onto the stack.

**Step 3:** If the scanned character is an **operator** Then

a. Pop the two top elements ***a*** and ***b*** from the stack where ***a*** is the top element and ***b*** is the next to top element.

b. Apply the operator on the operands ***b*** and ***a*** and push the result onto the stack.

[End Loop]

**Step 4:** Set ***Value = Stack[Top]***

**Step 5:** Print: "The value of the expression is": ***Value***

**Step 6:** Exit

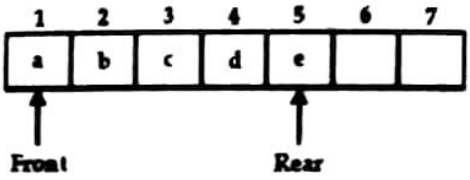
Consider the previously converted postfix expression $P = 6\ 2\ +\ 5\ \times\ 8\ 4\ /\ -$ which will be evaluated using algorithm as shown below:

| Character Scanned | Status of Stack |
|---|---|
| 6 | 6 |
| 2 | 6 2 |
| + | 8 |
| 5 | 8 5 |
| * | 40 |
| 8 | 40 8 |
| 4 | 40 8 4 |
| / | 40 2 |
| - | 38 |

**d. How insertion and deletion operation takes place in a queue.Explain in detail.**

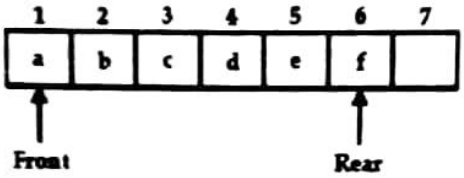
Consider an example of a list of 5 elements that is maintained by using an array $Q$ of size 7 and two variables ***Rear*** and ***Front***. The variable ***Front*** contains the index of the element at front end of the queue and the variable ***Rear*** contains the index of the

element at the rear end of the queue. Initially, the elements of the list are stored in an array as shown below:



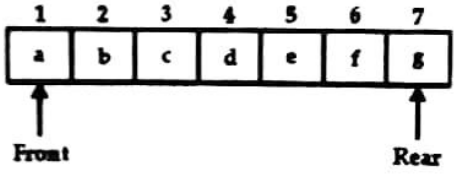

**Array Representation of Queue**

Whenever a new element is inserted into the queue, the value stored in the variable ***Rear*** will be incremented by one and element will be stored at the rear end of the queue. Thus, in the above example, a new element *f* will be inserted at index 6 as shown below:



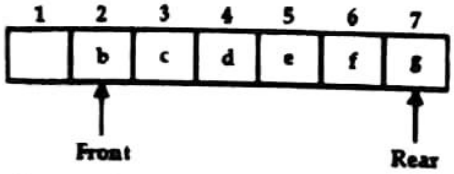

**Queue after Insertion of Element *'f'***

If we want to insert another element say ***g***, it will be inserted at index number 7 as shown below:



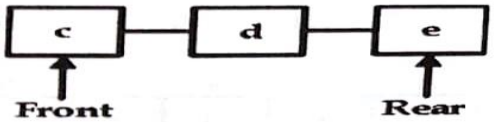

**Queue after Insertion of Element *'g'***

Now, let us delete the element from the queue, the only element which is at the front of the queue can be removed and variable ***Front*** of the queue will be incremented by one. If we delete an element from the queue ***Q*** shown in above figure, then after deletion, the queue will look like as shown below:




**Queue after Removal of an Element**

**If we want to delete one more element, then after deletion, the queue will be as shown below:**

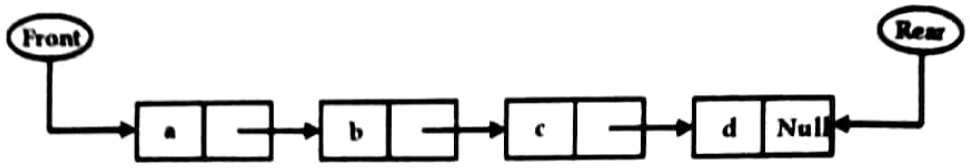



**Queue after the Removal of another Element**

e. **Explain how queue can be represented using linked list and give the algorithm**

**for insertion in it.**

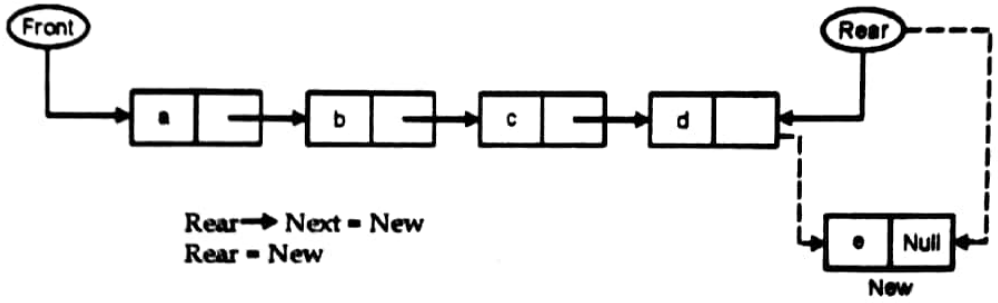
An alternative and efficient way of representing queues is by using linked list. The advantages of such a representation over array representation are similar to those of linked list over the array. Linked representation of queue can be represented diagrammatically as shown in figure below:




**A Queue Maintained using a Linked List**

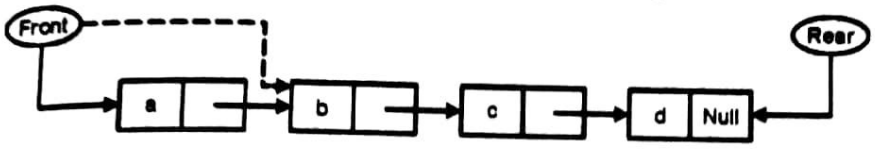
Here, two pointer variables ***Front*** and ***Rear*** contain the addresses of the elements at the front and rear end of the queue respectively. Initially, when there is no element in the queue, both its pointers ***Front*** and ***Rear*** will have value ***Null*** indicating an empty queue.

The insertion of a new element ***e*** in the above shown queue can be shown as in figure below:




**This insertion of an element *'e'* in the queue**

The deletion of an element from the queue can be shown as in figure below:




Front = Front → Next

**Deletion of an element from the Queue**

Here, the front element ***a*** has been deleted. So, the variable ***Front*** will point to the address of element ***b*** next to the deleted element. The algorithms for the insertion and deletion of the elements in the queue are as follows:

**Algorithm: Insert a given element *'Data'* in a queue which is implemented using a linked list *'Q'* having variable *'Front'* which contains the address of 1st element of the queue and variable *'Rear'* which contains the address of last element of the queue.**

Step 1: If ***Free = Null*** Then
Print: "No Free Space Available for Insertion"
Exit
[End If]

Step 2: Allocate memory to node ***New***
Set ***New = Free*** *And* ***Free = Free → Next***

Step 3: Set ***New → Info = Data*** *And* ***New → Next = Null***

Step 4: If ***Rear = Null*** Then
Set ***Front = New*** *And* ***Rear = New***
Else
Set ***Rear → Next = New*** *And* ***Rear = New***
[End If]

Step 5: Exit

f. **How priority queues are represented in memory. Explain them.**

Priority Queue is the special kind of queue data structure in which insertion and deletion operations are performed according to some special rule rather than just First In First Out (FIFO) rule. In case of priority queue, a priority number is associated with each element. The elements are inserted and deleted according to this priority number. The following two rules are applied to process the elements in the priority queue:

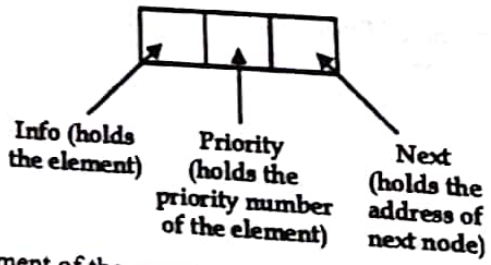
- The elements with higher priority are processed before the elements with lower priority.
- In case of elements with same priority, elements are processed according to First In First Out (FIFO) rule. That is, the element with same priority will be processed in the same order in which these are inserted into the queue.

Priority queue can be represented into the memory in various ways. The three main ways to represent a priority queue are:

- Priority queue using linked list
- Priority queue using multiple queues
- Priority queue using heap structure

### 5.5.2.1 Priority Queue Using Linked List

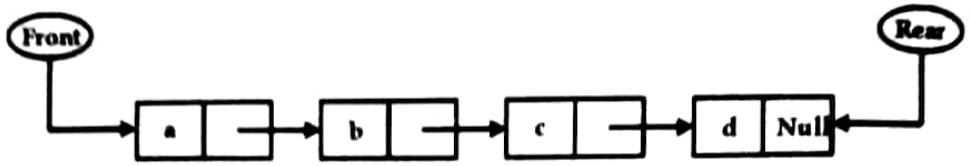
In the linked list representation of priority queue, each node of the linked list is divided into three parts as shown below:




***Info*** Part holds the element of the queue.
***Priority*** Part holds the priority number of the element.
***Next*** Part holds the address of next node of the linked list.

In linked list representation of priority queue, insertion of an element takes place according to the priority number of the element where as the deletion of an element takes place from the front end of the linked list (as the first most element will be of highest priority). Consider the following priority queue with 4 elements.

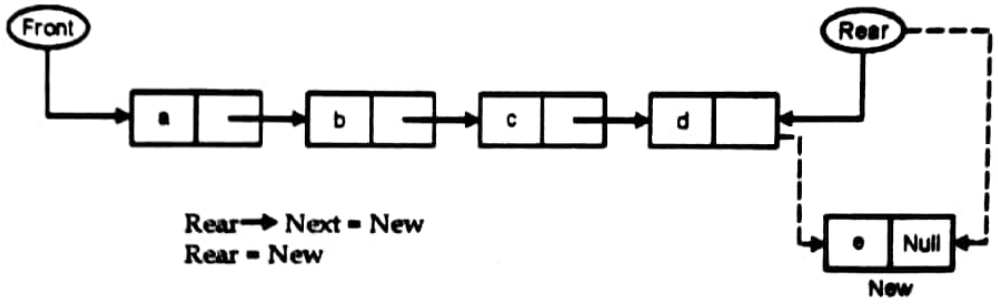
An alternative and efficient way of representing queues is by using linked list. The advantages of such a representation over array representation are similar to those of linked list over the array. Linked representation of queue can be represented diagrammatically as shown in figure below:




**A Queue Maintained using a Linked List**

Here, two pointer variables ***Front*** and ***Rear*** contain the addresses of the elements at the front and rear end of the queue respectively. Initially, when there is no element in the queue, both its pointers ***Front*** and ***Rear*** will have value ***Null*** indicating an empty queue.

The insertion of a new element ***e*** in the above shown queue can be shown as in figure below:




**This insertion of an element *'e'* in the queue**

Priority queue can be implemented using multiple arrays in the form of multiple queues. In this representation of priority queue, a separate queue is maintained for all the elements with same priority level which follows the general **FIFO** order. These separate queues may be circular queues for its efficient use and will also have separate variables ***Front*** and ***Rear***. While using multiple queues representation of priority queues, following information must be known in advance,

- The maximum number of priority levels (so that maximum number of separate queues may be known in advance).
- The maximum number of elements with same priority (so that maximum size of each queue may be known in advance).

Let us suppose that we have maximum of 5 priority levels (i.e. priority number $1, 2, 3, 4, and\ 5$) and maximum of 4 elements with same priority. So, the priority queue can accommodate maximum of $5 \times 4 = 20$ elements. Elements with priority $p$ are inserted in the respective queue of priority $p$ according to FIFO rule whereas the elements can be deleted according to the priority of the element (starting from the $1^{st}$ queue onwards).

Following is the representation of priority queue using multiple queues:

| Priority | Front | Rear | 1 2 3 4 |
|---|---|---|---|
| 1 | 0 | 0 | |
| 2 | 0 | 0 | |
| 3 | 0 | 0 | |
| 4 | 0 | 0 | |
| 5 | 0 | 0 | |

**Priority queue representation using multiple queues**

After insertion and deletion of the following elements in sequence, the resulting representation will be:

Insert ***p*** with priority 3
Insert ***n*** with priority 5
Insert ***g*** with priority 3
Insert ***s*** with priority 4
Insert ***m*** with priority 3
Delete an element
Insert ***k*** with priority 4
Insert ***d*** with priority 3
Delete an element
Insert ***b***with priority 2
Insert ***l*** with priority 2
Delete an element
Insert ***r*** with priority 1
Insert ***v*** with priority 2

| Priority | Front | Rear | 1 2 3 4 |
|---|---|---|---|
| 1 | 1 | 1 | r |
| 2 | 2 | 3 | b l v |
| 3 | 3 | 4 | p g m d |
| 4 | 1 | 2 | s k |
| 5 | 1 | 1 | n |

**4. Attempt *any three* of the following:** **15**

a. **Write an algorithm to find the minimum and maximum element in binary search tree.**

**Algorithm: To find the largest element in a Binary Search Tree.**

**Step 1:** **If** $Root = Null$ **Then**
**Print: "Tree is Empty"**
**Exit**
**Else**

---

Set $Pointer = Root$
[End If]
Step 2: Repeat while $Pointer \rightarrow Right \neq Null$
Set $Pointer = Pointer \rightarrow Right$
[End Loop]
Step 3: Set $Max = Pointer \rightarrow Info$
Step 4: Print: $Max$
Step 5: Exit

---

**Algorithm: To find the smallest element in a Binary Search Tree.**

Step 1: If $Root = Null$ Then
    Print: "Tree is Empty"
    Exit
  Else
    Set $Pointer = Root$
  [End If]

Step 2: Repeat while $Pointer \rightarrow Left \neq Null$
    Set $Pointer = Pointer \rightarrow Left$
  [End Loop]

Step 3: Set $Min = Pointer \rightarrow Info$

Step4: Print: $Min$

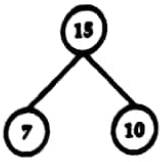
Step5: Exit

b. **Create a heap for the given elements 15 7 10 2 20 15 18.**

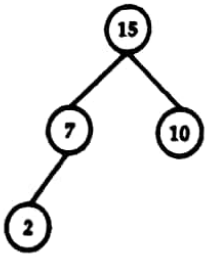
Step 1: Insert 15


Step 2: Insert 7

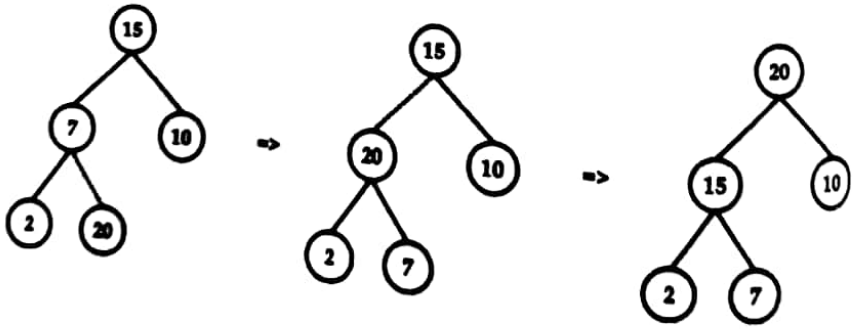

Step 3: Insert 10



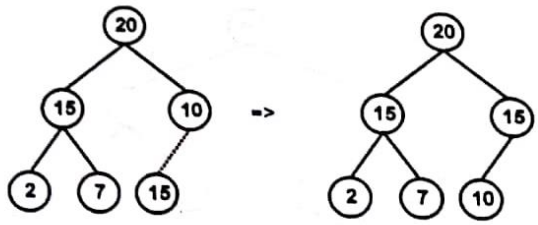

Step 3: Insert 2



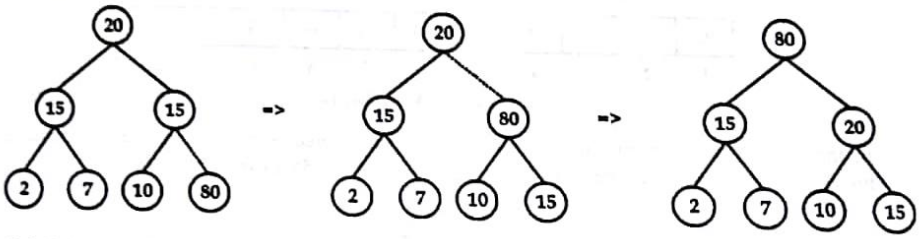

Step 5: Insert 20



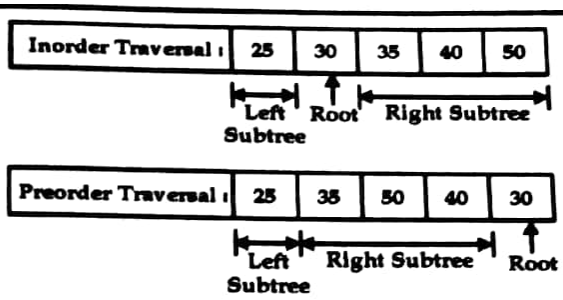
**Step 6: Insert 15**



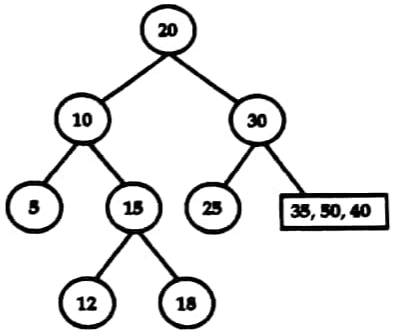

**Step 7: Insert 80**




c. **Construct a binary tree from its inorder and postorder traversals.**
**In-order: 5 10 12 15 18 20 25 30 35 40 50**
**Post-order 5 12 18 15 10 25 35 50 40 30 20**




The partial binary tree at this stage can be constructed as shown below:

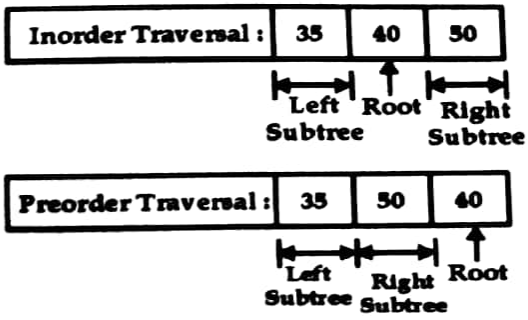



Now the right subtree of the node 30 is remaining whose in-order and post-order traversals are:

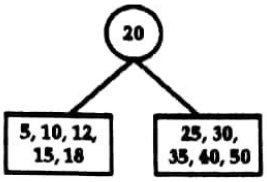
**In-order Traversal** : **35 40 50**

**Post-order Traversal** : **35 50 40**

From the post-order traversal, 40 is the root of this subtree and looking for this element in in-order traversal, we find that element 35 forms the left subtree and element 50 forms the right subtree as shown below:



At this stage, the partial tree can be constructed as shown in figure below:

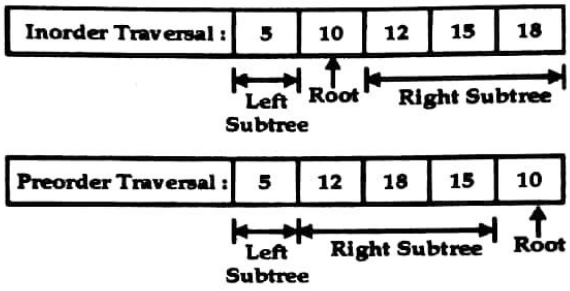



In the next stage, we will construct the left subtree with the elements of the left subtree, whose in-order and post-order traversals are:

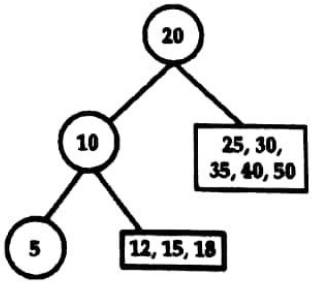
**In-order Traversal** : 5 10 12 15 18

**Post-order Traversal** : 5 12 18 15 10

By analyzing the post-order traversal, 10 is the root of the left subtree and looking for the element 10 in in-order traversal, we find that the element 5 form the left subtree and elements 12, 15, and 18 form the right subtree of the tree rooted at 10 as shown below:




At this stage, the partial binary tree can constructed as shown below:

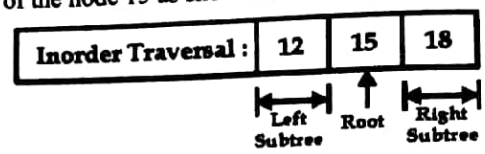


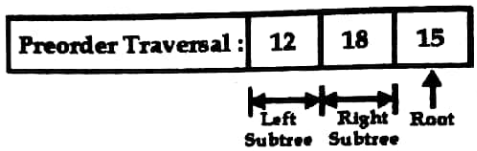

Now, following the same procedure for the right subtree of the node 10 whose in-order and post-order traversals are:

**In-order traversal :** 12 15 18

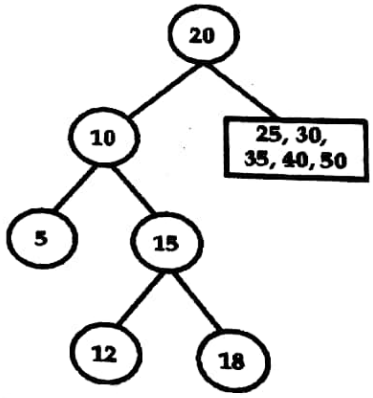
**Post-order traversal:** 12 18 15

From the post-order traversal, 15 is the root of the subtree and looking for this element 15 in in-order traversal, we find that the element 12 form the left subtree and element 18 form the right subtree of the node 15 as shown below:




The partial binary tree at this stage can be constructed as shown below:

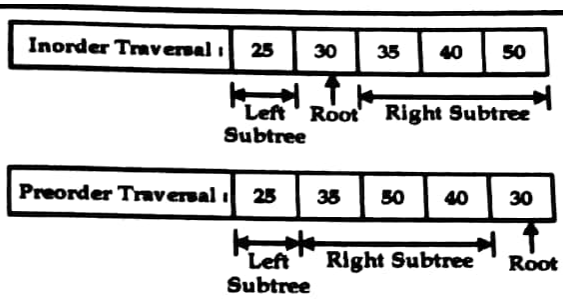



Now, the right subtree of the root node 20 is remaining whose in-order and post-order traversals are:

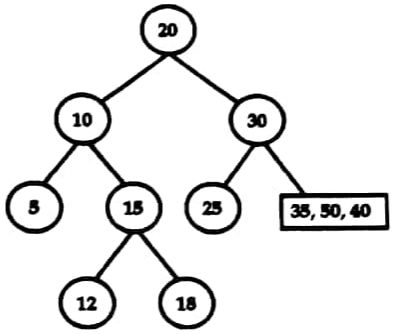
**In-order Traversal** : 25 30 35 40 50

**Post-order Traversal** : 25 35 50 40 30

From the post-order traversal, 30 is the root of this subtree and looking for this element in in-order traversal, we find that element 25 form the left subtree and elements 35, 40, 50 form the right subtree as shown below:




The partial binary tree at this stage can be constructed as shown below:

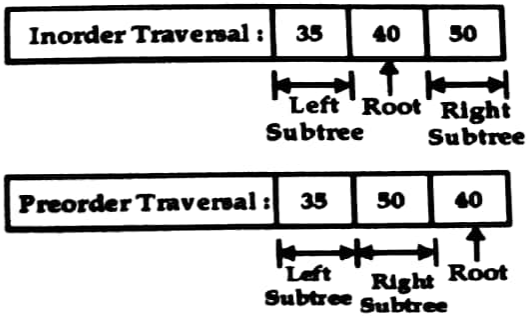



Now the right subtree of the node 30 is remaining whose in-order and post-order traversals are:

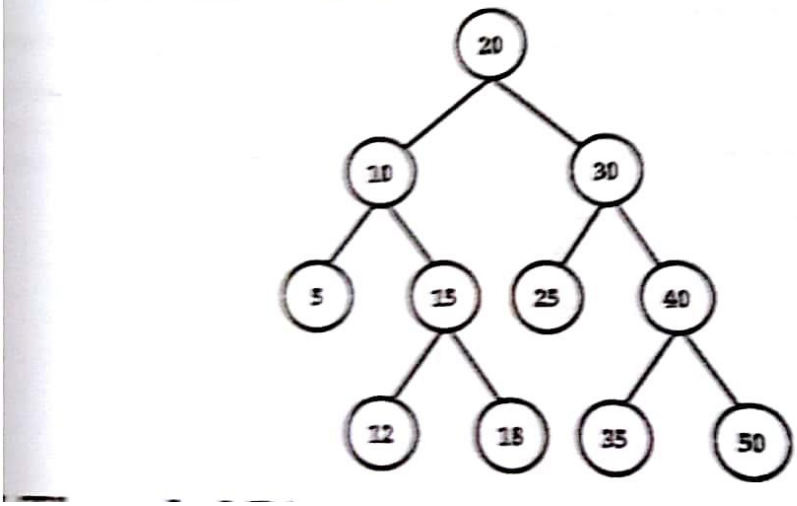
**In-order Traversal** : **35 40 50**

**Post-order Traversal** : **35 50 40**

From the post-order traversal, 40 is the root of this subtree and looking for this element in in-order traversal, we find that element 35 forms the left subtree and element 50 forms the right subtree as shown below:



this stage the binary tree will look like,




d. **Sort the following elements using selection sort.**
**22 35 17 8 13 44 5 28**

Selection sort is the sorting algorithm which is **in-place comparison** sort. Here, the term in-place means that this algorithm does not use any extra space for sorting the elements of the array. That is, the same array is used for sorting the elements of the array. Selection sort is **stable** as it does not exchange the relative position of elements with equal value.

The idea behind the selection sort is to find the smallest element in the array and replace it with the element at the first position in the array. Then, find the next smallest element of the array by starting the search from the $2^{nd}$ position to the last position of the array and replace it with the element at the $2^{nd}$ position in the array. This process is continued till the entire array is sorted.

Consider the unsorted array ***A*** of size **8** shown below:

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |
|---|---|---|---|---|---|---|---|
| 22 | 35 | 17 | 8 | 13 | 44 | 5 | 28 |

**1st Pass**

Search for the smallest element from $A[1]$ to $A[8]$. Here, the smallest element is 5 which is present at $7^{th}$ position of the array ***A***. Exchange the values at $A[1]$ and $A[7]$ which results in the array shown below:

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |
|---|---|---|---|---|---|---|---|
| 5 | 35 | 17 | 8 | 13 | 44 | 22 | 28 |

The resulting array after completion of the 1st pass.

**2nd Pass**

Now, search for the smallest element from $A[2]$ to $A[8]$. Here, the smallest element is 8 which is present at $4^{th}$ position of the array ***A***. Exchange the values at $A[2]$ and $A[4]$ which results in the array shown below:

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |
|---|---|---|---|---|---|---|---|
| 5 | 8 | 17 | 35 | 13 | 44 | 22 | 28 |

The resulting array after completion of the 2nd pass.

**3rd Pass**

Now, search for the smallest element from $A[3]$ to $A[8]$. Here, the smallest element 13 present at 5th position in the array $A$. Exchange the values at $A[3]$ and $A[5]$ which results in the array shown below:

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |
|---|---|---|---|---|---|---|---|
| 5 | 8 | 13 | 35 | 17 | 44 | 22 | 28 |

The resulting array after completion of the 3rd pass.

**4th Pass**

Now, search for the smallest element from $A[4]$ to $A[8]$. Here, the smallest element is 17 present at 5th position in the array $A$. Exchange the values at $A[4]$ and $A[5]$ which results in the array shown below:

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |
|---|---|---|---|---|---|---|---|
| 5 | 8 | 13 | 17 | 35 | 44 | 22 | 28 |

The resulting array after completion of the 4th pass.

**5th Pass**

Now, search for the smallest element from $A[5]$ to $A[8]$. Here, the smallest element is 22 present at 7th position in the array$A$. Exchange the values at $A[5]$ and $A[7]$ which results in the array shown below:

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |
|---|---|---|---|---|---|---|---|
| 5 | 8 | 13 | 17 | 22 | 44 | 35 | 28 |

The resulting array after completion of the 5th pass.

**6th Pass**

Now, search for the smallest element from $A[6]$ to $A[8]$. Here, the smallest element is 28 present at 8th position in the array$A$. Exchange the values at $A[6]$ and $A[8]$ which results in the array shown below:

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |
|---|---|---|---|---|---|---|---|
| 5 | 8 | 13 | 17 | 22 | 28 | 35 | 44 |

The resulting array after completion of the 6th pass.

**7th Pass**

Now, search for the smallest element from $A[7]$ to $A[8]$. Here, the smallest element is 35 present at 7th position in the array $A$, which is already at its proper position. So, in this case no exchange will take place. The array after 7th pass will be as shown below:

| A[1] | A[2] | A[3] | A[4] | A[5] | A[6] | A[7] | A[8] |
|---|---|---|---|---|---|---|---|
| 5 | 8 | 13 | 17 | 22 | 28 | 35 | 44 |

The resulting array after completion of the 7th pass.

Thus, the above array is sorted.

e. **Write and Explain the algorithm for finding a position of a given element and its parent in a binary search tree.**

To search a particular element in a binary search tree, we start at the root by comparing the desired element with the value stored at the root. If both are same then the search procedure is stopped otherwise we follow the left or right branch depending on whether the given element is less than or larger than the element stored at the root node. This procedure is repeated recursively until we find the desired element or conclude that element is not present in the binary search tree.

**Algorithm: To find the position of a given element 'Item' and its parent in a Binary Search Tree.**

**BSTSearch(Root, Item, Position, Parent)**

Step 1: If **Root = Null** Then
Set **Position = Null**
Set **Parent = Null**
Return
[End If]

Step 2: **Pointer = Root** And **PointerP = Null**

Step 3: Repeat Step 4 While **Pointer ≠ Null**

Step 4: If **Item = Pointer → Info** Then
Set **Position = Pointer**
Set **Parent = PointerP**
Return
Else If **Item < Pointer → Info** Then
Set $PointerP = Pointer$
Set $Pointer = Pointer \rightarrow Left$
Else
Set $PointerP = Pointer$
Set $Pointer = Pointer \rightarrow Right$
[End If]
[End Loop]

Step 5: Set $Position = Null$ And $Parent = Null$

Step 6: Return

f. **Write the algorithm for inserting in a node in Red-Black tree.**

**Algorithm: Insert a node in Red-Black tree.**

$RBTreeInsertion\ (T, x)$

Step 1: Insert the node $x$ in Red-Black tree using $BSTInsertion()$ algorithm and color the node $x$ as Red.

Step 2: Repeat While $(x \rightarrow Parent \rightarrow Color = Red)$

Step 3: If $(x \rightarrow Parent = x \rightarrow Parent \rightarrow Parent \rightarrow Left)$ Then

Step 4: If $(x \rightarrow Parent \rightarrow Parent \rightarrow Right \rightarrow Color = Red)$ Then

```
                    // Case
                    x → Parent → Color = Black                    red
                    x → Parent → Parent → Right → Color = Black
                    x → Parent → Parent → Color = Red
                    x = x → Parent → Parent
                Else
                    // Case 3 when x's parent is left child and x's uncle is black
                    If (x = x → Parent → Right) Then
                        // If x is Right Child
                        x = x → Parent
                        LeftRotate (T, x)
                    [End If]
                    // If x is Left Chid
                    x → Parent → Color = Black
                    x → Parent → Parent → Color = Red
                    RightRotate (x → Parent → Parent)
                [End If]
            Else
Step 6:         If (x → Parent → Parent → Left → Color = Red) Then
                    // Case 2, x's parent is right child and its uncle is red.
                    x → Parent → Color = Black
                    x → Parent → Parent → Left → Color = Black
                    x → Parent → Parent → Color = Red
                    x = x → Parent → Parent
                Else
                    // Case 4: x's parent is right child and x's uncle is black
                    If (x = x → Parent → Left) Then
                        x = x → Parent
                        RightRotate (x)
                    [End If]
                    x → Parent → Color = Black
                    x → Parent → Parent → Color = Red
```

$LeftRotate\ (x \rightarrow Parent \rightarrow Parent)$

[End If]

[End If]

[End Loop]

Step 6: $Root \rightarrow Color = Black$

Step 7: Exit

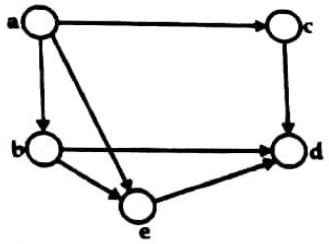
**5. Attempt *any three* of the following:** **15**

a. **What are the different ways to represent graphs in memory.Explain.**

Suppose $G = (V_g, E_g)$ is a directed graph having $n$ nodes. Suppose, vertices are ordered by using $v_1, v_2, v_3, v_4, \ldots\ldots\ldots v_n$. Then adjacency matrix $A$ for the graph $G$ will be a square matrix of order $n$ such that:

$$a_{ij} = \begin{cases} 1 & \text{if an edge lies between the vertices } v_i \text{ and } v_j \\ 0 & \text{if there is no edge between the vertices } v_i \text{ and } v_j \end{cases}$$

The adjacency matrix of a graph depends upon the ordering of its vertices. If we change the order of vertices in a graph then it will result in a different adjacency matrix. However, adjacency matrix formed after changing the order of vertices will be closely related to the preceding one. Consider a directed graph $G$ shown in figure below:



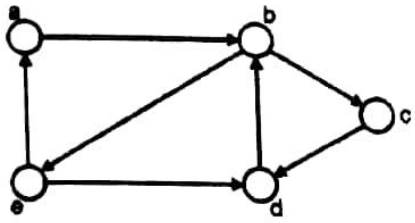

**A Directed Graph with 5 Edges**

$V_G = \{a, b, c, d, e\}$

The adjacency matrix corresponding to this ordering sequence will be:

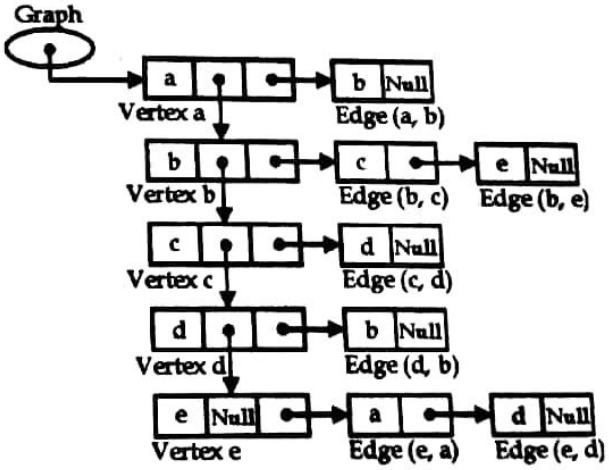
$$A = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

Another method for representing graph is using linked list structure. In this representation, each vertex in the graph is a node in a master linked list. Another linked list starts from each vertex node and denotes the vertices which are directly adjacent to a given source vertex. This method, often called an adjacency list, is more space efficient than the adjacency matrix representation. Consider a graph $G$ shown below in figure below:




Adjacency list of Nodes
a → b
b → c, e
c → d
d → b
e → a, d

**A Directed Graph with 5 Vertices**




**Adjacency List Representation of Graph shown in previous figure**

b. **Write and Explain the algorithm for Best First Search in a graph.**

In Breadth-first search, we begin with the start vertex say $v$ and mark it as discovered. The vertex $v$ at this time is said to be discovered but not processed. A vertex is said to be processed by an algorithm when the algorithm has visited all vertices adjacent to it. All unvisited vertices adjacent to $v$ will be visited next. Vertex $v$ has now been processed. The newly visited vertices have not been processed yet, so these will be put at the end of the list of unprocessed vertices. The first vertex on this list is the next to be explored. Exploration continues until single unexplored vertex is left. The list of unexplored vertices acts as a queue and can be represented using the standard queue representation.

**Algorithm: Traversal of a graph using Breadth-First Search. Traversal starts at vertex *'s'***

Step 1: Initially all the vertices of the graph $G$ are set with the status **Undiscovered.**

Step 2: Change the state of the starting vertex s of the graph to **Discovered**, and put it into the queue $Q$.

Step 3: Repeat Steps 4 to 5 While the **queue is not empty**

Step 4: Remove a vertex say $v$ which is at the front of the queue and change state to **Processed**

Step 5: Repeat For all **Undiscovered** adjacent vertices $u_i$ of vertex $v$

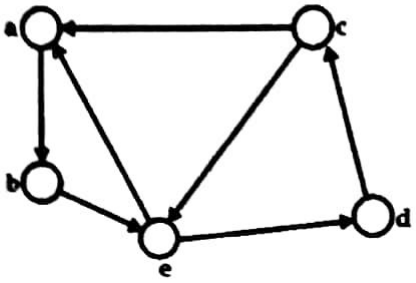
$u_i$ is set to the status **Discovered** and added to the queue

[End Loop]

[End Loop]

Step 6: Exit

To understand the working of breadth search algorithm, consider a graph $G$ shown below in the figure and its adjacency list. Suppose, we want to start the traversal from the vertex **d**, then it will act as the starting vertex for the breadth-first traversal.




Adjacency list of Nodes

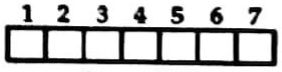
a → b

b → e

c → a, e

d → c

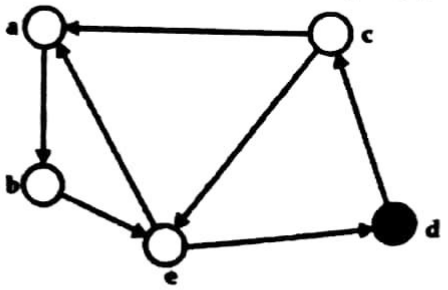
e → a, d

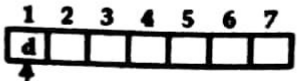



Front = Null

Rear = Null

Change the state of starting vertex **d** from undiscovered to the discovered by painting it with grey color and put it into the queue which is initially empty.

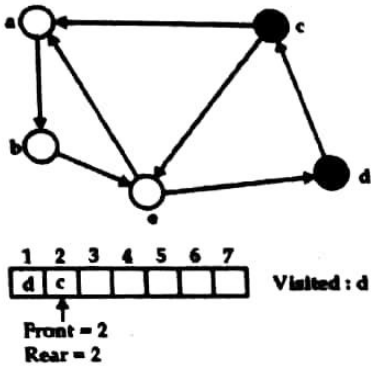


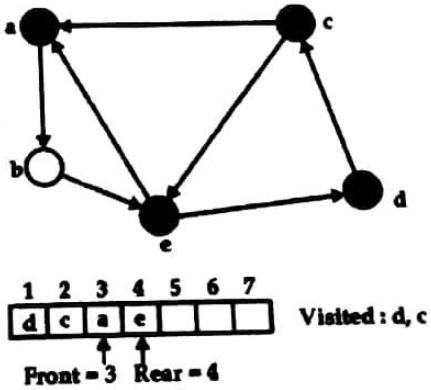

Front = 1

Rear = 1

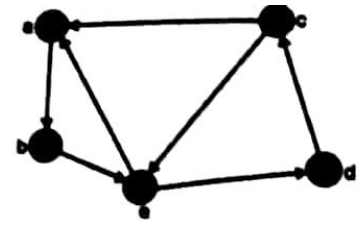
Now, remove the vertex ***d*** from the queue and change its state from discovered to the processed state by painting it with **Black** color and add all its adjacent vertices into the queue which are in undiscovered state. Here, the undiscovered adjacent vertex ***c*** is added into the queue.

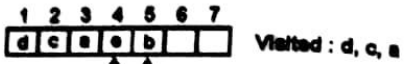



Now, remove a vertex from the front of the queue and change its state to the processed state and add all its adjacent vertices, which are in undiscovered state to the queue by changing their state from undiscovered to the discovered. Here, vertex ***c*** is processed and its undiscovered adjacent nodes ***a*** and ***e*** are added into the queue. These nodes can be added in the queue in any order but we have added these in alphabetic order.

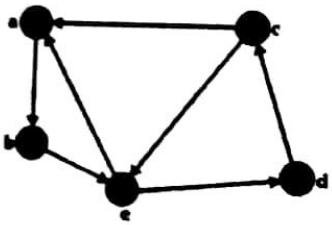


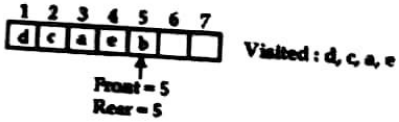

Remove the vertex from the front of the queue and change its state from the discovered to the processed state. Add all its adjacent vertices which are still in undiscovered state into the queue and change their state from undiscovered to the discovered. Here, vertex ***a*** is processed and its undiscovered adjacent vertex ***b*** is added into the queue.



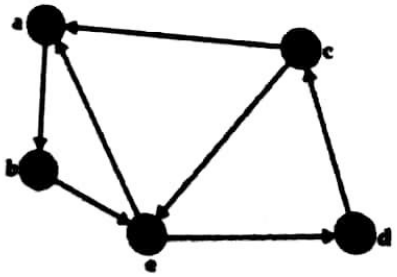




Now, remove the vertex ***e*** which is at the front of the queue and change its state to the processed state. The vertices which are adjacent to vertex ***e*** are vertex ***a*** and ***d***. But, both of these are already in processed state. So, no vertex will be added to the queue.



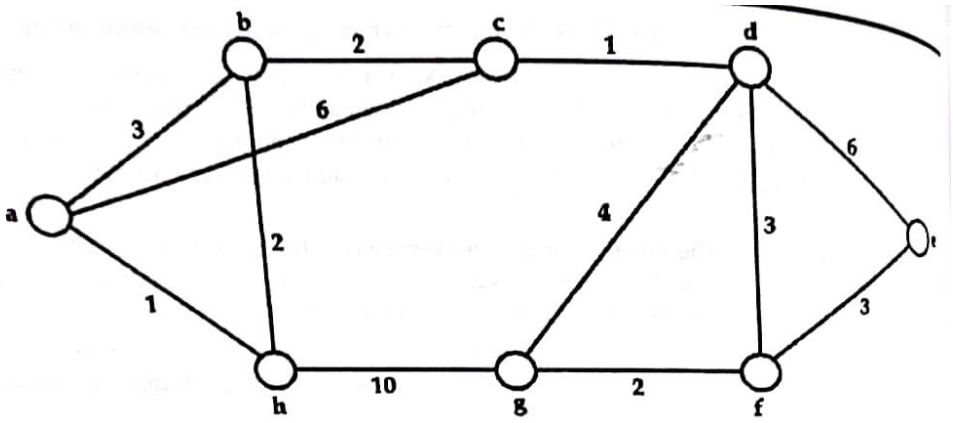



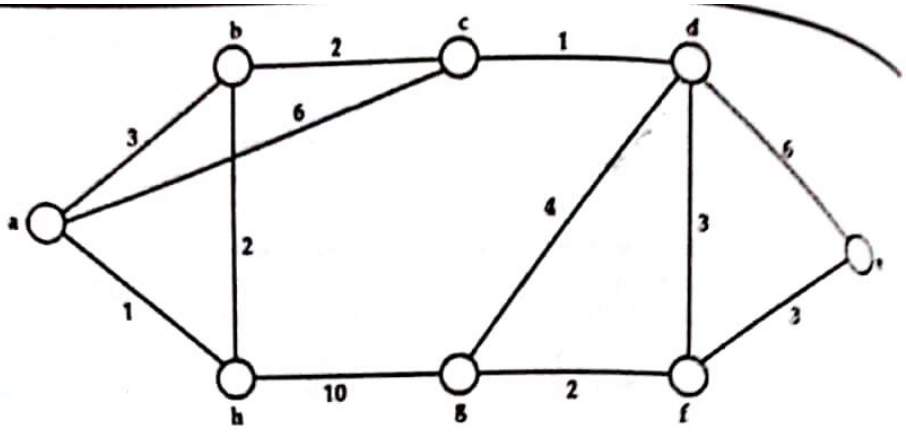

Remove the vertex ***b*** from the queue and change its state from discovered to the processed. Add all its undiscovered adjacent vertices into the queue. The only vertex which is adjacent to vertex ***b*** is vertex ***e***, but it is in processed state. So, no vertex will be added in the queue. At this stage, queue is empty and no vertex is left unprocessed. So, we will stop the process here.




Visited : d, c, a, e, b

c. **Using Prim's Algorithm find the minimum spanning tree.**


b
c
d
2
1
3
6
6
a
4
3
2
1
3
10
2
h
g
f




Initially, Set of Selected nodes $N = \{\phi\}$

Set of Selected Edges $M = \{\phi\}$

Set of Graph Edges $S = \{ab, bc, cd, de, ef, fg, gh, ha, ac, bh, gd, df\}$

$3 \quad 2 \quad 1 \quad 6 \quad 3 \quad 2 \quad 10 \quad 1 \quad 6 \quad 2 \quad 4 \quad 3$

Starting with the node $a$, we choose a least cost edge which start from node $a$ and does not form the cycle. Here, we use edge $ah$.

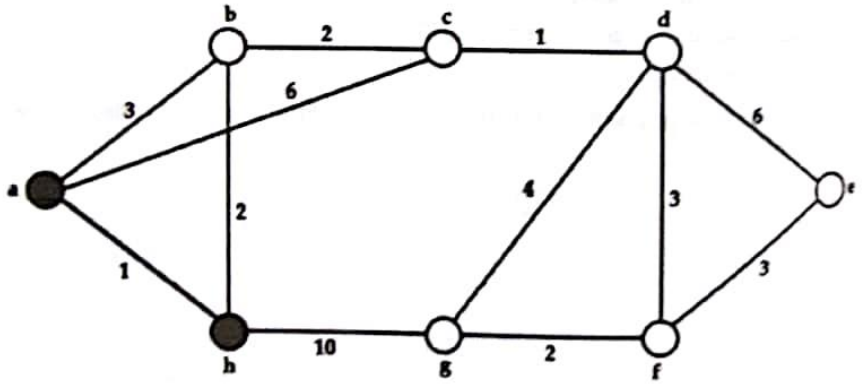
Put edge $ah$ to list $M$ and node $h$ to list $N$ ($a$ was initially in $N$) and remove the edge $ah$ from $S$

$N = \{a, h\}$

$M = \{ah\}$

$S = \{ab, bc, cd, de, ef, fg, gh, ac, bh, gd, df\}$

$3 \quad 2 \quad 1 \quad 6 \quad 3 \quad 2 \quad 10 \quad 6 \quad 2 \quad 4 \quad 3$



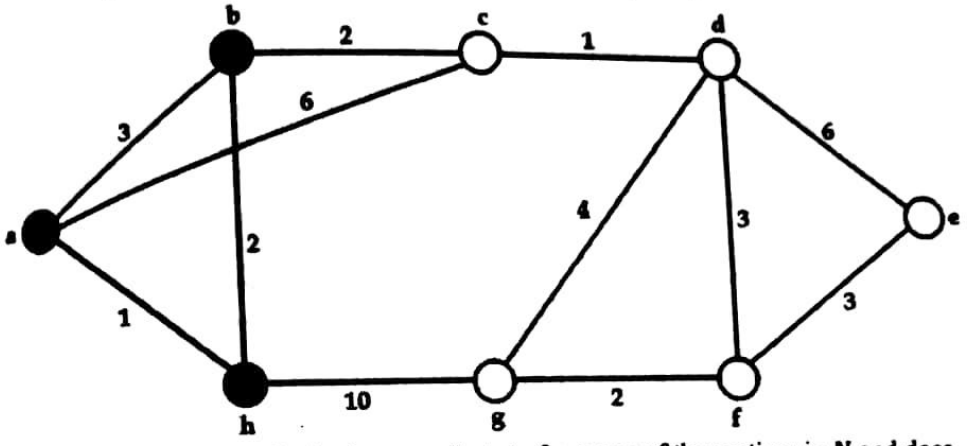
Now, the least cost edge **bh** is chosen as it starts from any of the vertices in **N** and does not form any cycle. Here, note that the edge **cd** is not choosen even it is least cost edge because this edge do not start from any of the vertices in **N**.

$N = \{a, h, b\}$

$M = \{ah, bh\}$

$S = \{ab, bc, cd, de, ef, fg, gh, ac, gd, df\}$

$\quad\quad 3 \quad 2 \quad 1 \quad 6 \quad 3 \quad 2 \quad 10 \quad 6 \quad 4 \quad 3$



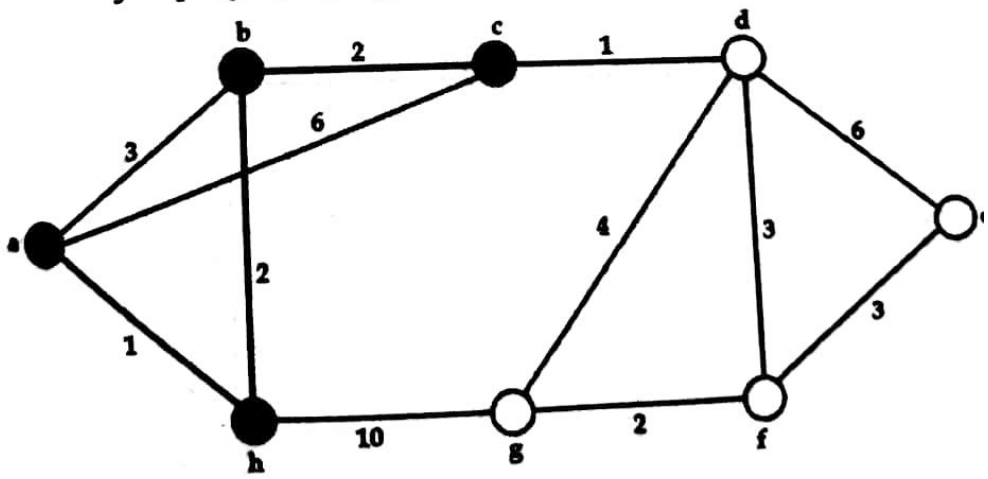

Now, the least cost edge **bc** is chosen as it starts from any of the vertices in **N** and does not form any cycle.

$N = \{a, h, b, c\}$

$M = \{ah, bh, bc\}$

$S = \{ab, cd, de, ef, fg, gh, ac, gd, df\}$

$\quad\quad 3 \quad 1 \quad 6 \quad 3 \quad 2 \quad 10 \quad 6 \quad 4 \quad 3$

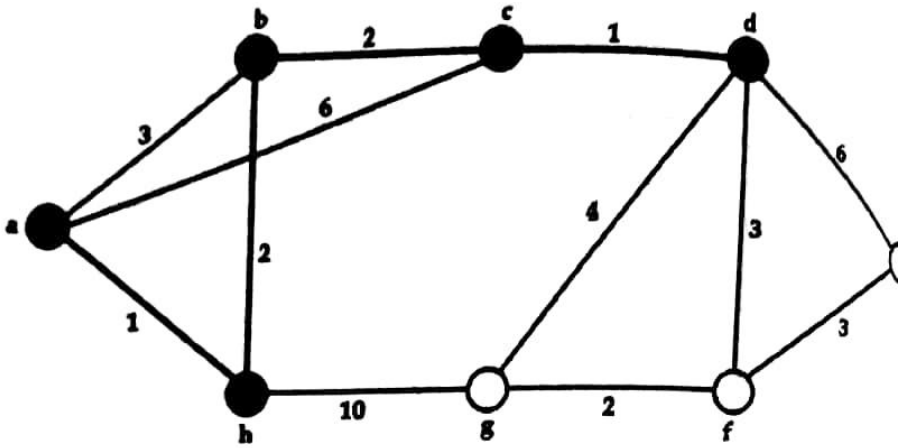


Now, the least cost edge $cd$ is chosen from $S$ as it starts from a vertex in $N$ and d
form any cycle.

$N = \{a, h, b, c, d\}$

$M = \{ah, bh, bc, cd\}$

$S = \{ab, de, ef, fg, gh, ac, gd, df\}$
$\quad\quad 3 \quad 6 \quad 3 \quad 2 \quad 10 \quad 6 \quad 4 \quad 3$

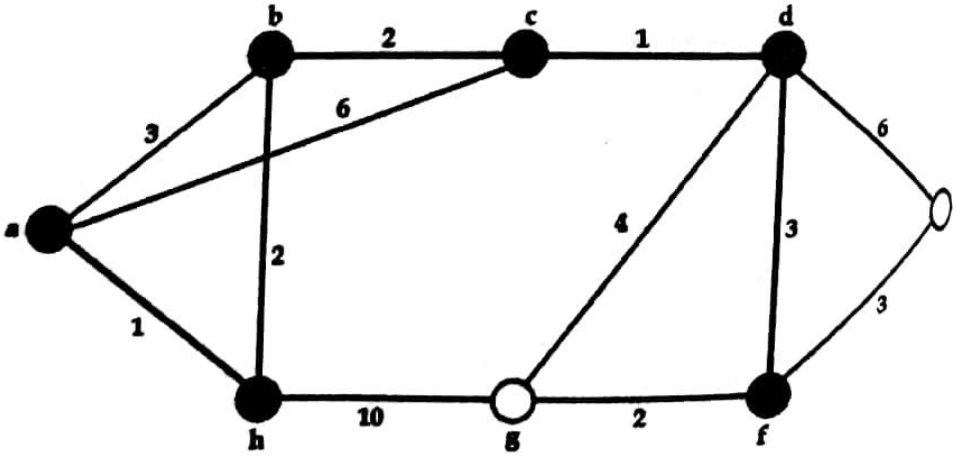



Now, we choose the least cost edge $df$ from $S$ as it starts from any of the vertices $N$
does not form any cycle.

$N = \{a, h, b, c, d, f\}$

$M = \{ah, bh, bc, cd, df\}$

$S = \{ab, de, ef, fg, gh, ac, gd\}$
$\quad\quad 3 \quad 6 \quad 3 \quad 2 \quad 10 \quad 6 \quad 4$

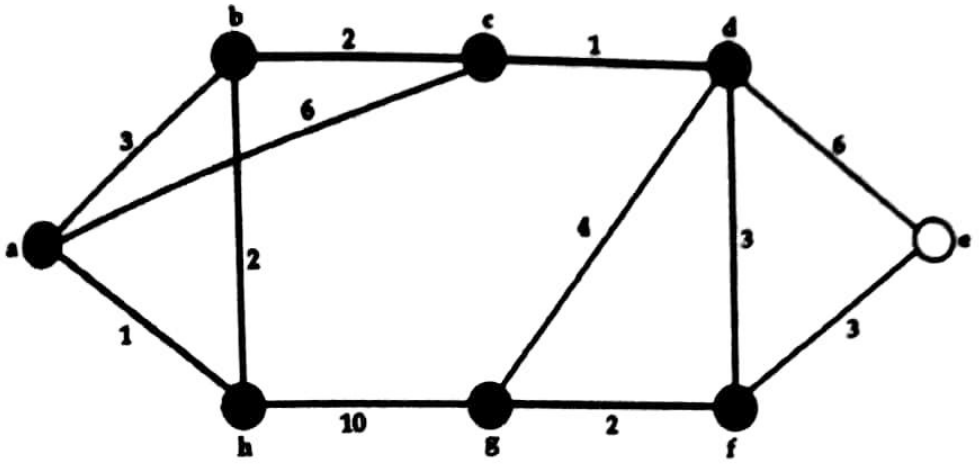


Now, we choose the least cost edge $gf$ from $S$ as it starts from any of the vertices in $N$ and does not form any cycle.

$N = \{a, h, b, c, d, f, g\}$

$M = \{ah, bh, bc, cd, df, fg\}$

$S = \{ab, de, ef, gh, ac, gd\}$
$\quad\quad 3 \quad 6 \quad 3 \quad 10 \quad 6 \quad 4$

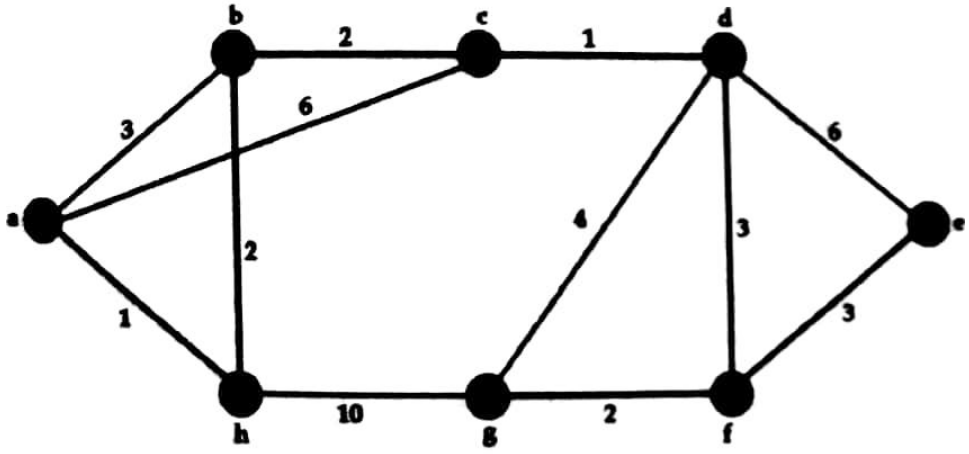



Now, we choose the least cost edge $ef$ from $S$ as it does not form cycle and it starts fro any of the vertices in $N$

$N = \{a, h, b, c, d, f, g, e\}$

$M = \{ah, bh, bc, cd, df, fg, ef\}$

$S = \{ab, de, gh, ac, gd\}$
$\quad\quad 3 \quad 6 \quad 10 \quad 6 \quad 4$

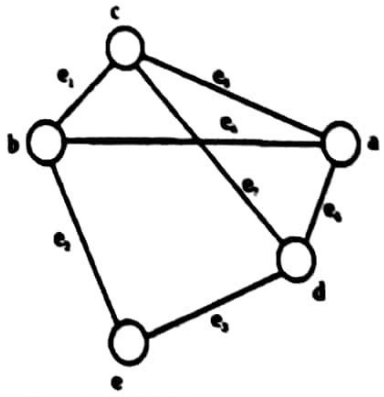



**The Resulting MST**

d. **Define the following terms.**

1. **Graph.**
2. **Weighted Graph.**
3. **MultiGraph.**
4. **Directed Graph.**
5. **Hamiltonian Path.**

A graph G consists of finite set of vertices $V_G$ and finite set of edges $E_G$ which can be denoted by a tuple $G = (V_G, E_G)$. Here, the set of vertices $V_G$ represent the entities which has names and some other attributes. An edge connects a pair of vertices and represents a relationship between the two entities. A graph may be pictorially represented as shown in figure below:




**A Graph with 5 Vertices and 7 Edges**

In this graph, vertices are labeled using letters $a, b, c, d, and\ e$. Therefore,

$V_G = \{a, b, c, d, e\}$

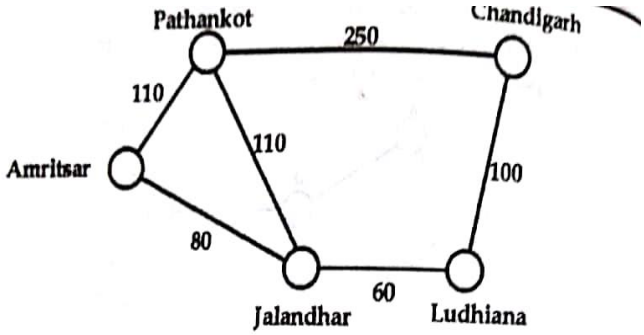
$E_G = \{ab, be, ed, dc, ca, bc, ad\}$ or

$E_G = \{e1, e2, e3, e4, e5, e6, e7\}$

In this graph, edge between two vertices can be written in any order. For example edge between the vertices $a$ and $d$ can be written as either $ad$ or $da$.

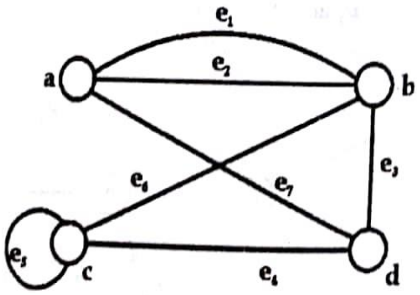
A graph $G$ is said to be weighted if edges in it are assigned weights. This is often desi to represent certain physical attributes/properties by means of graph. For example, edges of the graph shown in next figure are assigned weights. Here in the graph, weights assigned to the edges represent the distance between the cities, where the vert of the graph represent different cities. In case of weighted graph, an edge is represe as:

$e = (v_1, v_2, w)$ where $v_1$, $v_2$ are the vertices making an edge and $w$ is the weight o edge.




**A Weighted Graph**

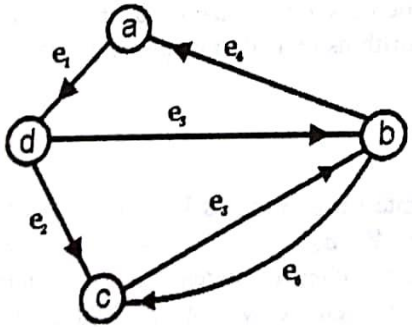
A graph $G$ is said to be **multigraph** if it contains multiple edges or loop in it. example, the graph shown in figure below is a multigraph as it contains a loop at vert and multiple edges between the vertices $a$ and $b$.




A simple graph does not allow any loop or multiple edges, or cycle in it.

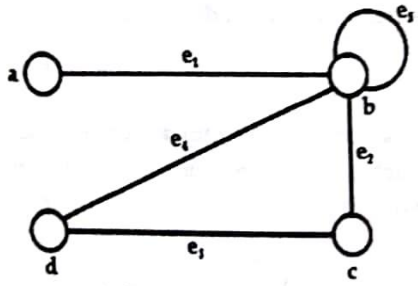
## Directed Graph

In the previous figure, no directions are associated with the edges of the graph. graph is an undirected graph. The edges of an undirected graph are represe unordered pair of vertices. In case of directed graph shown in below figure, each assigned a direction or we can say that each edge is identified by an ordered vertices in the graph rather than an unordered pair.




**A Directed Graph**

A path is said to be **Hamiltonian path** if it contains all the vertices in the graph e.g. in figure shown below:




**A Graph with 4 Nodes**

$a \rightarrow b \rightarrow c \rightarrow d$ is a Hamiltonian path

e. **Explain any two collision resolution techniques.**

One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as **open addressing** in that it tries to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called **linear probing**.

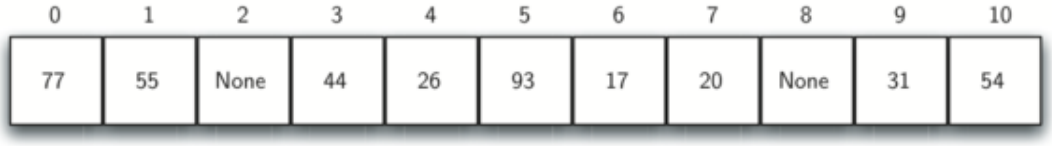
.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 77 | 44 | 55 | 20 | 26 | 93 | 17 | None | None | 31 | 54 |

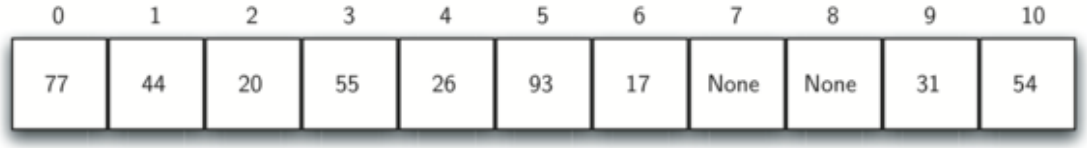
A disadvantage to linear probing is the tendency for **clustering**; items become clustered in the table. This means that if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution. This will have an impact on other items that are being inserted, as we saw when we tried to add the item 20 above. A cluster of values hashing to 0 had to be skipped to finally find an open position. This cluster is shown in Figure 9.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 77 | 44 | 55 | 20 | 26 | 93 | 17 | None | None | 31 | 54 |

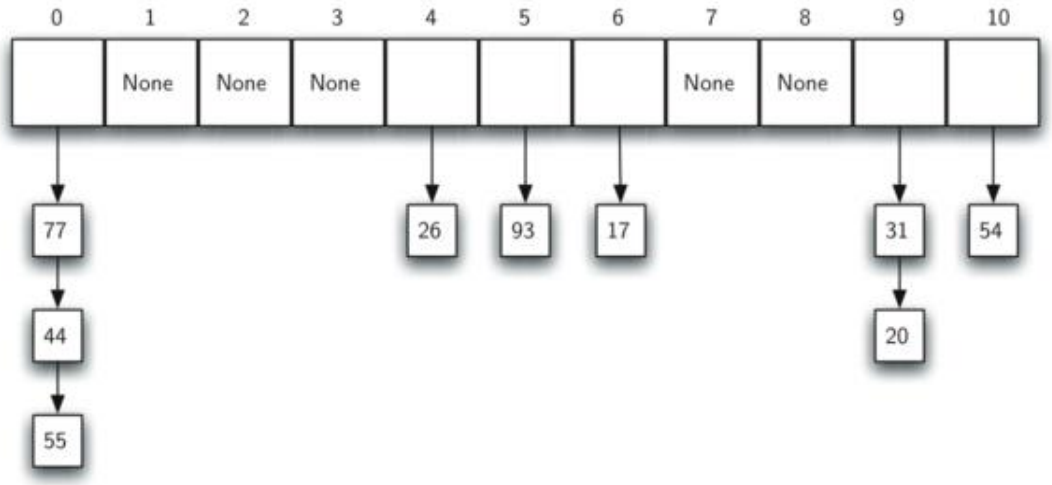
One way to deal with clustering is to extend the linear probing technique so that instead of looking sequentially for the next open slot, we skip slots, thereby more evenly distributing the items that have caused collisions. This will potentially reduce the clustering that occurs. Figure 10 shows the items when collision resolution is done with a “plus 3” probe. This means that once a collision occurs, we will look at every third slot until we find one that is empty.




A variation of the linear probing idea is called **quadratic probing**. Instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on. This means that if the first hash value is h, the successive values are $h+1$h+1, $h+4$h+4, $h+9$h+9, $h+16$h+16, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares. Figure 11 shows our example values after they are placed using this technique.




An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. **Chaining** allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases. Figure 12shows the items as they are added to a hash table that uses chaining to resolve collisions.




f. **What are Hash table and Hash Functions. Explain Folding Method and mid square method for constructing hash functions.**

A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0
The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and

return an integer in the range of slot names, between 0 and m-1

The Folding Method The key K is partitioned into a number of parts ,each of which has the same length as the required address with the possible exception of the last part . The parts are then added together , ignoring the final carry, to form an address. Example: If key=356942781 is to be transformed into a three digit address. P1=356, P2=942, P3=781 are added to yield 079.

The Mid- Square Method The key K is multiplied by itself and the address is obtained by selecting an appropriate number of digits from the middle of the square. The number of digits selected depends on the size of the table. Example: If key=123456 is to be transformed. (123456)2=15241383936 If a three-digit address is required, positions 5 to 7 could be chosen giving address 138.