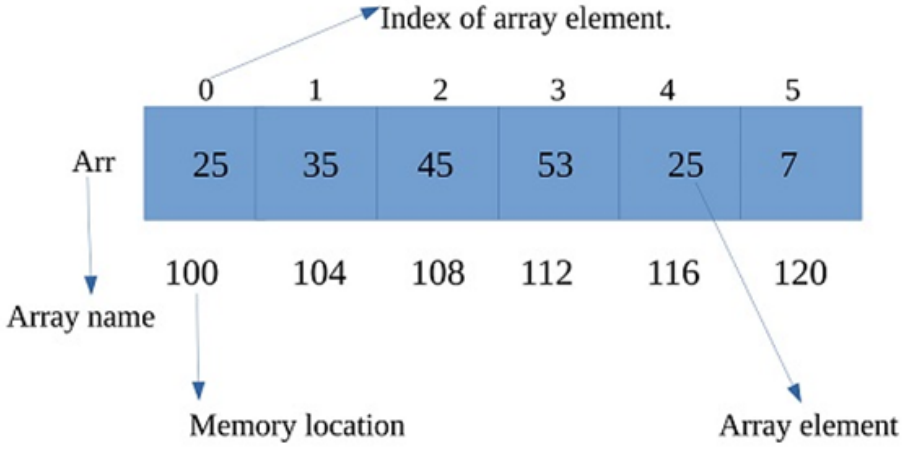
DATA STRUCTURE NOTES

- By Dipesh Adelkar

Explain One Dimensional Array

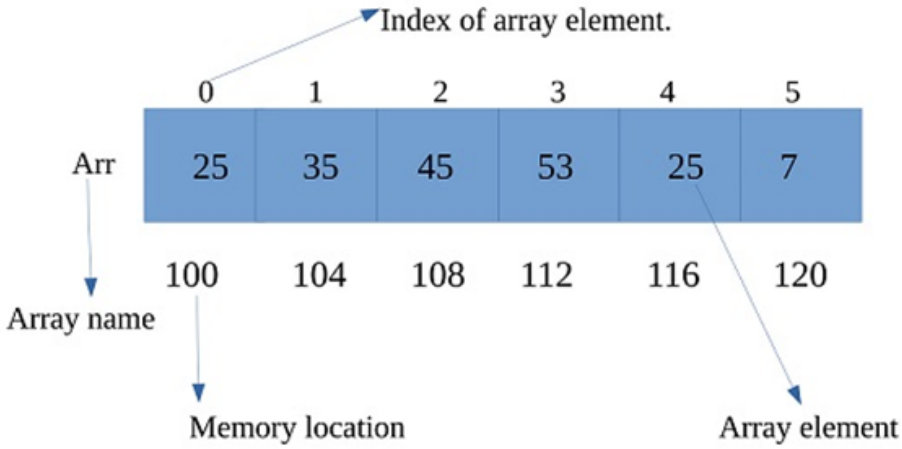
- An array is a linear collection of finite number of homogeneous elements
- Elements in an array are in a sequence
- The elements are homogeneous which means the elements are of similar datatype
- All the elements in an array are stored in consecutive memory location
- Example of array



- Here we have an array named ‘Arr’ with index starting with 0 till 5
- The number of elements in an array are called its size
- Size of array can be calculated as
- Size of array = ub – lb + 1
- In the above example ub(upper bound, highest index) is 5 and lb(lower bond, lowest index) is 0
- So, size of array = (5 – 0) + 1 = 6

Explain Memory Representation of array

- The elements in the array are stored in consecutive memory location
- Let’s consider an array with Base(S)



- Address of any other element of an array S can be calculated as
- Loc(Sk) = Base(S) + w(k – lb)
- Sk is Kth element of array and Loc(Sk) is location of Sk
- Base(S) is the base address of the array
- w is the size of data type of the array element
- (k - lb) is the distance of k from lower index(first index)

Example
Here Base(S) = 100 , w = 4 , lb = 0
Loc(Sk) = Base(S) + w(k – lb)
Loc(S3) = 100 + 4(3 – 0) **….finding memory location of elements at index 3**
Loc(S3) = 100 + 4 x 3
Loc(S3) = 100 + 12
Loc(S3) = 112
Hence element at index 3 is stored at location 112 in memory

Perform Traversing in Array

- It refers to visiting every element in an array
- Traversing an Array

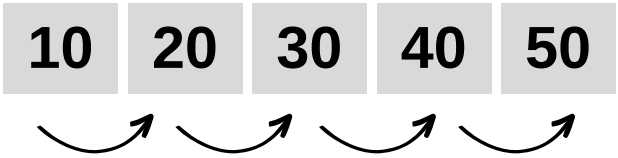
1. **set i = lb**
2. **while i <= ub**

   print s[i]

   set i = i + 1

   [end loop]
3. **Exit**

- We have assigned the value of lower bound to i
- We have used a loop until the value of i reaches upper bound
- In step 3 we have printed the ith element
- In step 4 we have incremented the value of i by 1

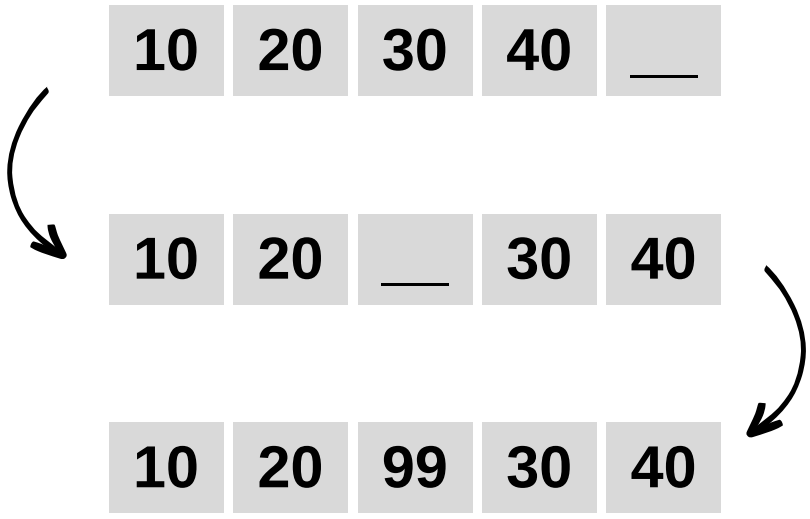


Perform Insertion In Array

- Insertion is to add new element in the array
- The new element can be added anywhere in the array
- Only possible if there is space for element in array
- Algorithm : inserting ‘New’ element at k where size of array ‘S’ is n

1. **while i = n to k**
   Set S[i+1] = S[i]
   Set i = i - 1
   [end loop]
2. **Set S[k] = New**
3. **Set n = n + 1**
4. **Exit**

- In this algorithm we are moving all the elements from k to n to one position at the right.
- When all the elements are moved one position to the right, we insert new element at kth position
- And then increase the size of array by 1
- Suppose we have to place 10 at the place of 40

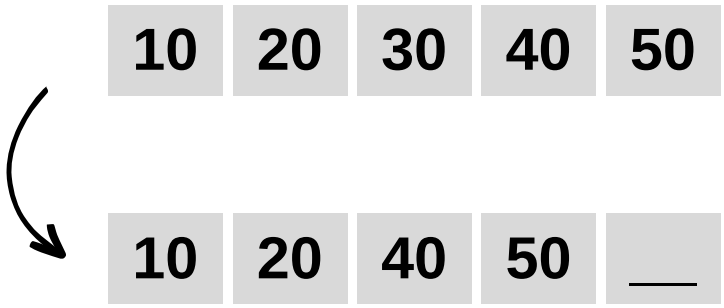


Perform Deletion In Array

- Deletion is to delete element in the array
- Algorithm : Deleting element at k where size of array ‘S’ is n

1. **While 2 and 3 for i = k to n**
   Set S[i] = S[i + 1]
   Set i = i + 1
   [end loop]
2. **Set n = n - 1**
3. **Exit**

- In this algorithm we are moving all the elements from k to n to one position at the left
- And then decrease the size of array by 1
- Suppose we have to remove 40



Perform Binary Search In Array

- Binary search can be applied when the array is sorted
- Binary search works like dictionary
- To use binary search, we need to find the index of middle element
- Middle index = (ub + lb)/2

10	20	30	40	50
1	2	3	4	5

- Suppose we have this array (S) and we have to find 40
- lb = 1 , ub = 5
- middle index = (5 + 1)/2 = 3
- S[middle index] = 30 < 40
- So, we will set lb as 4 and ub as 5, Middle index = (5 + 4)/2 = 4
- S[middle index] = 40 = Desired element, therefore we will stop search

1. **Set start = lb, end = ub**
2. **Repeat steps 3 and 5 while start <= end**
3. **Set middle = (start + end)/2**
4. **If S[middle] = data then**
   Print “element found”
   Exit
5. **If S[middle] > data then set**
   Start = middle - 1
6. **Else**
   Start = middle + 1

   End loop
7. **Exit**

Difference Between Linear and Binary Search

Linear search	Binary search
Linear search starts with lower bound	Binary search starts with middle index
sequential search, starts at one end and goes to another through each element of array	Binary search divides the array in half to search a certain part only
Also called as sequential search	Also called as half interval search
Less efficient	More efficient
Sorting is not required	The array should be sort to perform binary search
Complexity is 0(n)	Complexity is 0(log2n)
Good if the element is present at the first index	Good if the element is present at the middle index

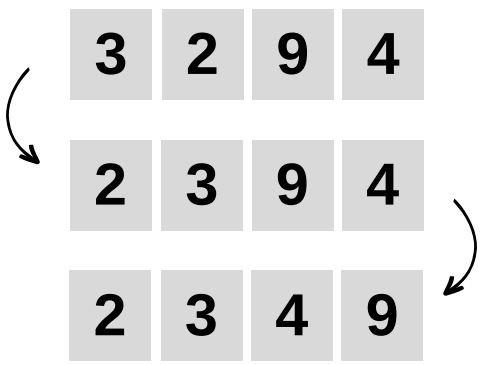
Explain Bubble Sorting of Array

- Sorting of array refers to arrange the elements of the array in some logical order
- this order can be in increasing or decreasing

S	10	20	30	40	50
	1	2	3	4	5

- Here S[1] < S[2] < S[3] < S[4] < S[5]

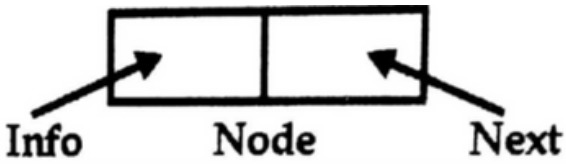
```
1   Repeat For p = 1 to n - 1
2       For i = 1 to n - p
3           If S[i] > S[i + 1] Then
                Exchange S[i] with S[i + 1]
            [End If]
        [End Loop]
    [End Loop]
4   Exit
```

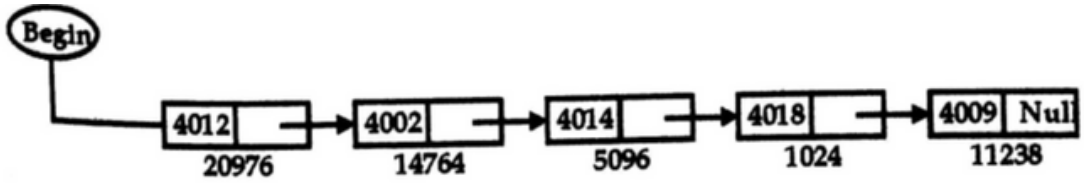


- Here a loop is executed for n - 1 times
- we have used a nested loop which is executed for certain steps
- later we have compared the two elements and exchanged if required

Explain Linked List

- Linked list is the linear collection of elements at nonconsecutive memory location
- In linked list the element/data is stored in a node
- Node has two partitions one to store data and another one which stores address/location of the next node
- Last node of the linked list has Null value as the next which defines end of linked list





- The 1st node is held by special pointer known as begin/Start
- 1st node points to the address of 2nd node i.e., 14764
- 2nd node points to the address of 3rd node i.e., 5096
- This happens till the last node which points to Null value

Perform Traversing in Linked List

- It refers to visiting every element in linked list
- Traversing an linked list

1. **If Begin = Null Then**
   Print “Linked list is empty”
   Exit
   [End If]
2. **Set Pointer = Begin**
3. **Repeat while Pointer != Null**
   Print : Pointer -> Info
   assign Pointer = Pointer -> Next
   [End Loop]
4. **Exit**

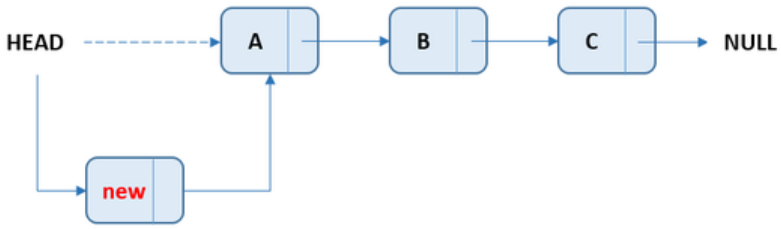
- Here we check if the linked list is empty or not
- if the list is not empty we execute step 2 , Pointer is assigned to the first node which carries the address of the node
- Info part of the current element gets printed and then the pointer is assigned to the address of next node
- once pointer encounters Null value the loop ends

Searching in Linked List

1. **If Begin = Null Then**
   Print “Linked list is empty”
   Exit
   [End If]
2. **Set Pointer = Begin**
3. **Repeat while Pointer != Null**
   If Pointer -> Info = Data Then
   print “Element found”
   else
   assign Pointer = Pointer -> Next
   [End Loop]
4. **Exit**

- Here we check if the linked list is empty or not
- if the list is not empty we execute step 2 , Pointer is assigned to the first node which carries the address of the node
- we check if the info of the element pointed by the Pointer is equal to the desired data or not
- once pointer encounters Null value the loop ends

Insertion at Beginning of Linked List



1. **If Free = Null Then**

   Print “No Free space available”

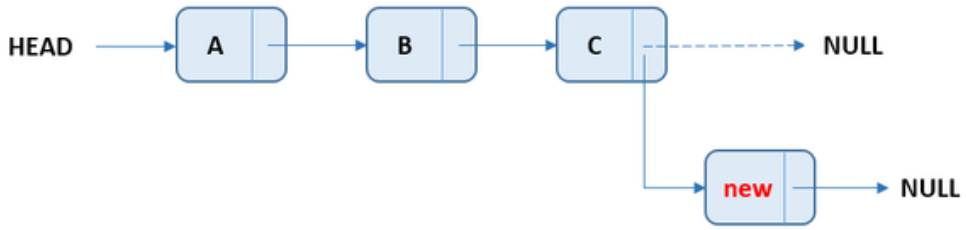
   Exit

   [End If]
2. **Allocate space to New**

   (Set New = Free and Free = Free -> Next)
3. **Set New -> Info = Data**
4. **Set New -> Next = Begin and Begin = New**
5. **Exit**

- Check if there is memory storage free for node or not
- we allocate the new node to the linked list
- Data is stored in the Info part of the New node
- The New node is inserted at the beginning of the linked list

Insertion at End of Linked List



Step 1 If Free = Null Then
Print “No Free space available”
Exit
[End If]

Step 2 Allocate space to New
(Set New = Free and Free = Free -> Next)

Step 3 Set New -> Info = Data, New -> Next = Null

Step 4 If Begin = Null then
Begin = New
Exit
[End If]

Step 5 Set Pointer = Begin

Step 6 Repeat While Pointer -> Next != Null
Set Pointer = Pointer -> Next
[End Loop]

Step 7 Set Pointer -> Next = New

Step 8 Exit

Deleting node at Beginning of Linked List

1. **If Begin = Null Then**
   Print “Linked List is Empty”
   Exit
   [End If]
2. **Set Item = Begin -> Info And Pos = Begin**
3. **Set Begin = Begin -> Next**
4. **Set Pos -> Next = Free and Free = Pos**
5. **Exit**

- Check if the linked list is empty or not
- Address of first Node is stored in a variable Pos
- The address of 2nd node Is assigned by the Begin
- the deleted node is added to the free storage list

Explain Two Way Linked List

- Pre - contains the address of preceding node
- Info - contains the data
- Next - contains the address of the next node

- Here in two way linked list two pointer variable are used - Begin and End
- Begin and End contains the address of the first node and last node respectively
- Pre part of the first node contains Null value
- Next part of the last node contains Null Value

Traversing Two Way Linked List

NOTE : start with explanation of what is two way linked list

1. **If End = Null Then**
   Print “Linked List is Empty”
   Exit
   [End If]
2. **Set Pointer = End**
3. **Repeat while Pointer != Null**
   Print Pointer -> Info
   Set Pointer = Pointer -> Pre
   [End Loop]
4. **Exit**

Searching in Two Way Linked List

1. If End = Null Then
   Print “Linked List is Empty”
   Exit
   [End If]
2. Set Pointer = End
3. Repeat while Pointer != Null
   If Pointer -> Info = Data
   Print “Desired element Found”
   Exit
   Else
   Pointer = Pointer -> Pre
   [End Loop]
4. Exit

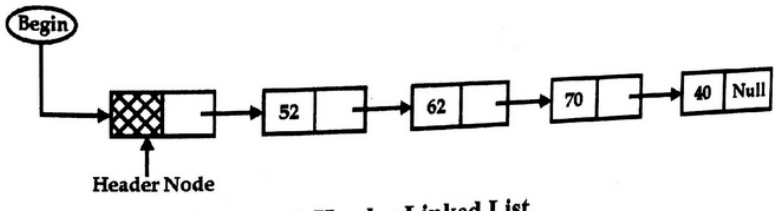
Insertion at Beginning of Double Linked List

1. If Free = Null Then
   Print “No Free space available”
   Exit
   [End If]
2. Allocate space to New
   (Set New = Free and Free = Free -> Next)
3. Set New -> Info = Data and New -> Pre = Null
4. If Begin = Null Then
   New -> Next = Null and End = New
   Else
   Set New -> Next = Begin And Begin -> Pre = New
5. Set Begin = New
6. Exit

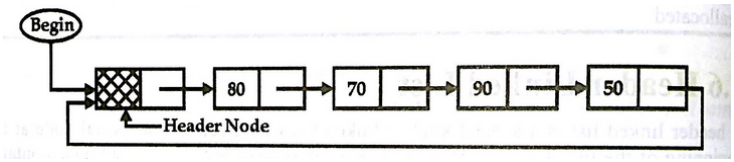
Explain Header Linked List

- Header Linked list is a special kind of linked list which contains a special node at the beginning of the lit
- This Special node is called as a head node
- This Head node contains some information about regarding the Linked List
- e.g. Number of nodes
- Header Linked List can be categorized into 4 parts
    - Grounded Header Linked List
    - Circular Header Linked List
    - Two way Header Linked List
    - Circular Two way Header Linked List

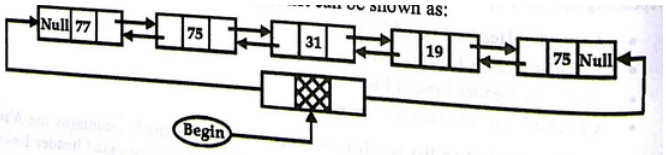
- Grounded Linked List - the last node contains Null in its Next Pointer



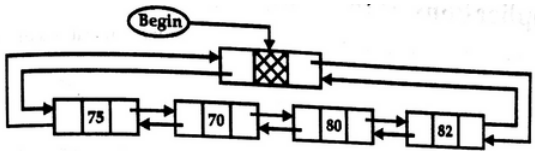
- Circular Linked List - The last node points back to the Header node



- Two Way Header Linked List - A Header node can also be Inserted in a Two way Linked List



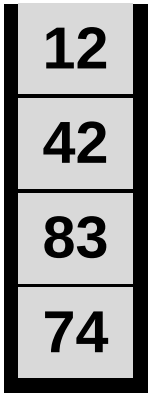
- Two Way Circular Header Linked List - The Last node of the Two way Linked list points back to the Header node



What is Stack?

- Stack is also called as ‘LIFO’ (Last In First Out) It means that the last item added to the stack will be the first item to be removed from the stack
- In Stack Insertion and Deletion takes place from only one end
- In Stack insertion operation is called as “PUSH”
- and deletion operation is called as “POP”

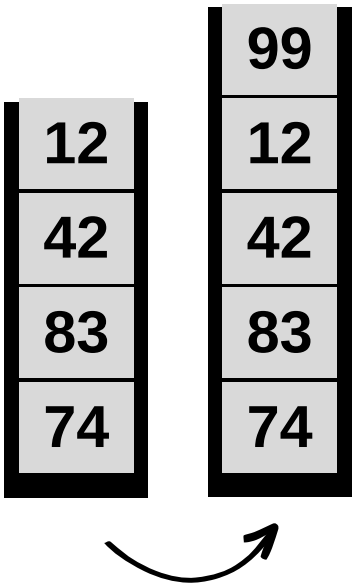
- PUSH - Push operation refers to insertion of a new element into the stack
- we can perform PUSH operation only if the stack is not full i.e. the stack should have sufficient space for new element
- If the stack is already full and when we try to insert new element it is called as “Stack Overflow” condition



- POP - Pop operation refers to removal of an element from the top of the stack
- We can perform POP operation only if the stack is not empty
- We need to make sure that the Stack is not empty before applying the POP operation
- When we try to perform POP operation while the stack is empty it is called as “Stack Underflow” condition

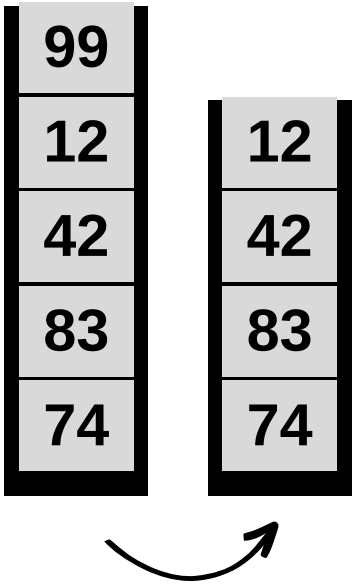
Array Representation of Stack
PUSH operation

1. If Top = Max then
   print “Stack is already full”
   Exit
   [End if]
2. Set Top = Top + 1
3. Set S[Top] = Data
4. Exit



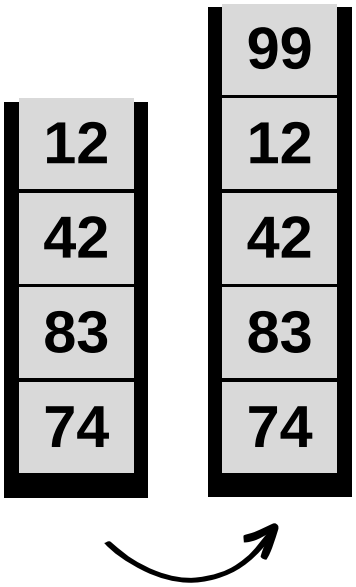
POP operation

1. If Top = Null then
   print “Stack is Empty”
   Exit
   [End if]
2. Set Data = S[Top]
3. Set Top = Top - 1
4. Exit



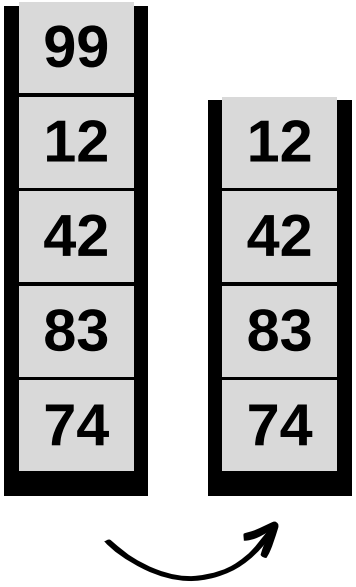
Linked List Representation of Stack
PUSH operation

1. If Free = Null then
   print “Stack is already full”
   Exit
   [End if]
2. Set New = Free and Free = Free -> Next
3. Set New -> Info = Data
   and New -> Next = Top
4. Set Top = New
5. Exit



POP operation

1. If Top = Null then
   print “Stack is Empty”
   Exit
   [End if]
2. Set Data = Top -> Info and Temp = Top
3. Set Top = Top -> Next
4. Deallocate the deleted node Temp to
   free storage list
5. Exit



Explain Infix Notation in Evaluation of Arithmetic Expression or Application of Stack

- Infix Notation : in this the operator is placed between its operands
- i.e. to multiply m and n we write m x n
- While solving the infix notation the main consideration is the preceding order of the operators and their associativity
- for example

e = q x r + s

- In This expression, the following is the preceding rule
- q and r will be multiplied and then will be added to s
- which means that x (multiplication) has preceding more than + (Addition)

Priority	Operator	Associativity
1st	Brackets () []	Inner to out left to right
2nd	Exponent ^	left to right
3rd	*/	Left to right
4th	+-	Left to right
5th	=	Left to right

Explain Prefix and Postfix Notation in Evaluation of Arithmetic Expression

- In Prefix notation, Operator is place before its operands
- for example, when we multiply m and n we write it as xmn

(a - b)/c

(-ab)/c

/ -abc

- Postfix notation is also know as ‘reverse polish notation’
- in this notation the operator is place after the operands

(a - b)/c

(ab-)/c

ab- c/

Evaluation of Postfix Notation

1. **Scan P from left to right and repeat step 2 and 3 till the end of expression**
2. **If scanned character is a Operand push it into the stack**
3. **If the scanned character is a operator then pop the two top elements (a and b) which are operands, apply the operator to these operands and push the result into the stack**
4. **[End Loop]**
5. **Set Value = Stack[Top]**
6. **Print “the value of the expression is ” : Value**
7. **Exit**

Explain Recursion

- Recursion is very important and powerful tool for developing algorithms for various problems, Recursion has the ability to call itself or to call itself or calling some other procedure which may result in calling the original procedure
- There are two very important conditions or we can say requirements which must be satisfied by any procedure to be defined recursively :
    - There must be a certain decision or conditional statement which can stop the further call of procedure
    - Each time the procedure calls itself it must be nearest to the solution
- Recursive functions can be implemented in various programming languages but some compilers are not able to handle recursive procedures, because they do not contain a stack mechanism
- Programming languages like C, C++ can be used to implement recursive procedures

Factorial Function in data structure

- The factorial of a positive number ‘n’ is the product of positive numbers from 1 to n
- factorial of a number is represented by place a ‘!’ next to the number
- e.g. 5!
- The factorial of a positive number ‘n’ will be defined as

$$n! = 1 \times 2 \times 3 \times 4 \ldots\ldots \times (n - 1) \times n$$

- The factorial of zero is taken as 1
- here are factorial of some positive numbers

$$3! = 3 \times 2 \times 1 = 6$$

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$$

Write a algorithm to find Fibonacci series to the nth term recursively

```
fibonacci(n):

        if n == 0:
                return 0

        elif n == 1:
                return 1

        else:
                return fibonacci(n - 1) + fibonacci(n - 2)

fibo = fibonacci(n)
```

Explain QUEUE

- Queue is a Linear collection of elements in which insertion takes place at one end known as “rear” and deletion takes place at another end known as the “front” of the queue
- The elements of the queue are processed in the same order as they were added into the queue, that’s why queue is also known as FIFO (First in First out)
- Two types of operations are performed in a queue
    - Insertion (rear)
    - Deletion (front)

- If the queue is already full and when we try to insert new element in the queue this condition can be called as “Overflow” condition
- If the queue is empty and if we try to delete an element from the queue this can be called as “Underflow” condition

Insertion in Queue (Array)

1. If Front = 1 and Rear = n Then
   Print “Queue is full, overflow condition”
   Exit

2. If Front = Rear + 1
   Print “Queue is full, overflow condition”
   Exit

3. If Rear = Null then
   Set front = 1 and Rear = 1
   Else if Rear = n then
   set Rear = 1
   Else
   Rear = Rear + 1

4. Set Q[Rear] = Data
5. Exit

Deletion in Queue (array)

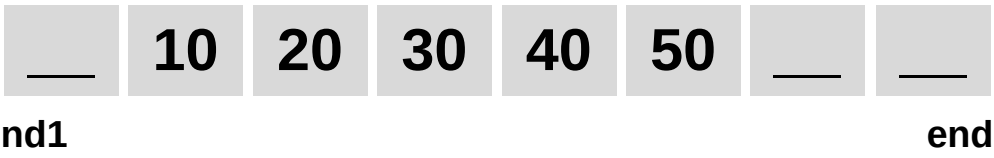
1. If Front = Null Then
   Print “Queue is Empty, underflow condition”
   Exit

2. Set Data = Q[Front]

3. If Front = Rear then
   // Queue has only one element
   Set Front = Null and Rear = Null
   Else if Front = n then
   Set Front = 1
   Else
   Set Front = Front + 1

4. Exit

Explain Deque

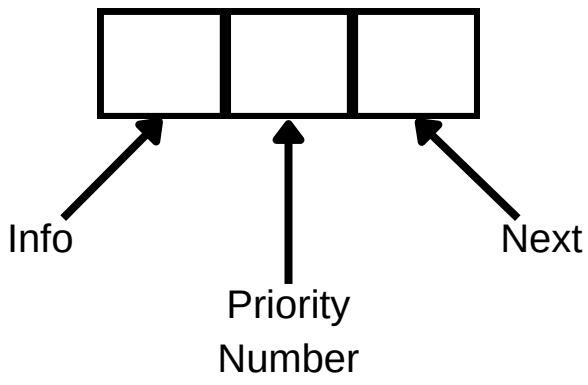


- Instead of using notation of Front and rear we use end1 and end2 to represent the index of the two ends of the queue
- An element can be inserted either at the end1 or at end2
- similarly the element can be deleted from the end1 or end2
- The Dequeue can be categorized into Two types
    - Input Restriction Deque : In this case the input is restricted from one end but the deletion can take place at both ends
    - Input is taken from one end and deletion is done from both ends
    - Output Restriction Deque : In this case the deletion operation is restricted to one end but the insertion can take place from both ends
    - Here, input is taken from both ends and the deletion is done from only one end

Explain Priority Queue

- Priority Queue is the kind of queue data structure in which insertion and deletion operation are performed according to some special rule rather than FIFO
- there are three ways to represent priority queue
    - Priority queue using linked list
    - priority queue using multiple queues
    - priority queue using heap structure

- Priority Queue using linked list :
    - in this priority queue the node of the linked list is divided into three parts
    - Info : this part contains the data element
    - Priority : this part hold the priority number of the element
    - Next : this part holds the address of the next node



- Priority Queue using Multiple Queues :
    - A Priority Queue using Multiple Queues is a data structure that manages elements with different priorities by maintaining multiple queues, each representing a distinct priority level.
    - Elements with higher priorities are removed from the queue before those with lower priorities, allowing for efficient access and removal of items based on their importance.

- Priority Queue using heap structure :
    - A Priority Queue using a heap structure is a data structure that maintains a binary heap, typically implemented as a min-heap or max-heap, to efficiently manage elements based on their priority.
    - In a min-heap, the smallest element is always at the root, while in a max-heap, the largest element holds this position, allowing for fast retrieval of the highest or lowest priority item.

Explain the types of Hashing Methods

- Mid Square Method
- Division Remainder Method
- Folding Method
- Shifting Method

- Mid Square Method
    - in this method the key value is squared and the digits are taken from the middle of the squared value
    - This middle 3 digits values of the square of the key is the Relative address

Key value	Square of Key vale	Relative address
5010	25100100	100
5016	25160256	160
5301	28100601	100
5400	29160000	160

- Division Remainder Method
    - in this method the the addresses are n=mapped by dividing the key values by the number of available addresses and the remainder is taken as the relative address

Key value	Remainder after dividing the key by 997
1098	101
1120	123
1185	118
1230	233

- Folding Method : in this method the key value is divided into equal number of parts except the last part, the first 3 digits and last 3 digits are reversed and added with the 3 digits in the middle and adjusted after ignoring the final carry
- Example 8 digit key value 51421673

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

- Shifting Method : The Key value is divided into 2 parts and added to each other ignoring the carry to make the address equal to the size of the divided parts
- Key 35367312

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

What is Open Addressing In Hashing

- Open addressing in data structures is a collision resolution technique used in hash tables.
- When a collision occurs (two elements hash to the same location), open addressing finds the next available slot in the hash table to place the item,
- there are 3 types of open addressing techniques
    i. linear Probing
    ii. Quadratic Probing
    iii. Double probing
- ‘K’ : value of keys
- ‘p’ : probing (can be 0,1,2,.., m-1)
- ‘m’ : size of array
- ‘c’ : constant
- H(K) = K mod m
- Linear Probing
    - Linear probing is a collision resolution technique in hash tables where, upon a collision, it searches for the next available slot by incrementing linearly until an empty slot is found.
    - **Hash(K, p) = [H(K) + p] mod m**

- Quadratic Probing
    - Quadratic probing is a collision resolution technique in hash tables where, upon a collision, it searches for the next available slot by incrementing quadratically until an empty slot is found
    - **Hash(K, p) = [H(K) + C1p + C2p^2] mod m**

- Double Probing
    - Double probing is a collision resolution technique in hash tables where, upon a collision, it uses a secondary hash function to calculate the step size for probing to find the next available slot.
    - **Hash(K, p) = [H1(K) + pH2(K)] mod m**

What is Linear Probing? with example

- Linear probing is a collision resolution technique in hash tables where, upon a collision, it searches for the next available slot by incrementing linearly until an empty slot is found.

Hash(K, p) = [H(K) + p] mod m

- p is probe number which can be 0,1,2,..., m-1
- n is the size of the hash table
- Keys : 33, 101, 99

K = 33

Hash(K, p) = [H(K) + p] mod m

Hash(K, p) = [33 mod 10 + 0] mod 10

Hash(K, p) = 3 mod 10

Hash(K, p) = 3

K = 101

Hash(K, p) = [H(K) + p] mod m

Hash(K, p) = [101 mod 10 + 0] mod 10

Hash(K, p) = 1 mod 10

Hash(K, p) = 1

K = 99

Hash(K, p) = [H(K) + p] mod m

Hash(K, p) = [99 mod 10 + 0] mod 10

Hash(K, p) = 9 mod 10

Hash(K, p) = 9

0	___
1	101
2	___
3	33
4	___
5	___
6	___
7	___
8	___
9	99

Explain Quadratic Probing, With Example

- Quadratic probing is a collision resolution technique in hash tables where, upon a collision, it searches for the next available slot by incrementing quadratically until an empty slot is found

Hash(K, p) = [H(K) + C1p + C2p^2] mod m

- p is probe number which can be 0,1,2,3,..., m-1
- m is size of hash table
- c1 and c2 are constant vales
- Let key values be 33,101, and 93

K = 33

Hash(K, p) = [H(K) + C1p + C2p^2] mod m

Hash(K, p) = [33mod10 + 0 + 0] mod 10

Hash(K, p) = 3 mod 10

Hash(K, p) = 3

K = 101

Hash(K, p) = [H(K) + C1p + C2p^2] mod m

Hash(K, p) = [101mod10 + 0 + 0] mod 10

Hash(K, p) = 1 mod 10

Hash(K, p) = 1

K = 93

Hash(K, p) = [H(K) + C1p + C2p^2] mod m

Hash(K, p) = [93mod10 + 0 + 0] mod 10

Hash(K, p) = 3 mod 10

Hash(K, p) = 3

0	___
1	101
2	___
3	33
4	___
5	___
6	___
7	93
8	___
9	___

- The record 93 should be place at position 3 in hash table
- but this position is occupied by other record
- so we will increase the probing by 1

K = 33

Hash(K, p) = [H(K) + C1p + C2p^2] mod m

Hash(K, p) = [33mod10 + 3 x 1 + 1 x 1] mod 10

Hash(K, p) = [3 + 3 + 1]mod 10

Hash(K, p) = 7

0	___
1	**101**
2	___
3	**33**
4	___
5	___
6	___
7	**93**
8	___
9	___

Explain Double Hashing With Example

- Double probing is a collision resolution technique in hash tables where, upon a collision, it uses a secondary hash function to calculate the step size for probing to find the next available slot.

Hash(K, p) = [H1(K) + pH2(K)] mod m

- p is probe number which can be 0,1,2,3,..., m-1
- m is size of hash table
- H1(K) = K mod m
- H2(k) = K mod m'
- m' < m
- Let key values be 33,103, and 93

K = 33

Hash(K, p) = [H1(K) + pH2(K)] mod m

Hash(K, p) = [33 mod 10 + 0 x 33 mod 8] mod 10

Hash(K, p) = 3 mod 10

Hash(K, p) = 3

K = 93

Hash(K, p) = [H1(K) + pH2(K)] mod m

Hash(K, p) = [93 mod 10 + 0 x 93 mod 8] mod 10

Hash(K, p) = 3 mod 10

Hash(K, p) = 3

- 93 should be stored at position 3 in hash table which is already occupied by other element so we increase the probing by 1

0	103
1	___
2	___
3	33
4	___
5	___
6	___
7	___
8	93
9	___

K = 93

Hash(K, p) = [H1(K) + pH2(K)] mod m

Hash(K, p) = [93 mod 10 + 1 x 93 mod 8] mod 10

Hash(K, p) = [3 + 1 x 5] mod 10

Hash(K, p) = 8

- So 93 will be stored at position 8 in hash table

K = 103

Hash(K, p) = [H1(K) + pH2(K)] mod m

Hash(K, p) = [103 mod 10 + 0 x 103 mod 8] mod 10

Hash(K, p) = [3] mod 10

Hash(K, p) = 3

- 103 should be stored at position 3 in hash table which is already occupied by other element so we increase the probing by 1

K = 103

Hash(K, p) = [H1(K) + pH2(K)] mod m

Hash(K, p) = [103 mod 10 + 1 x 103 mod 8] mod 10

Hash(K, p) = [3 + 7] mod 10

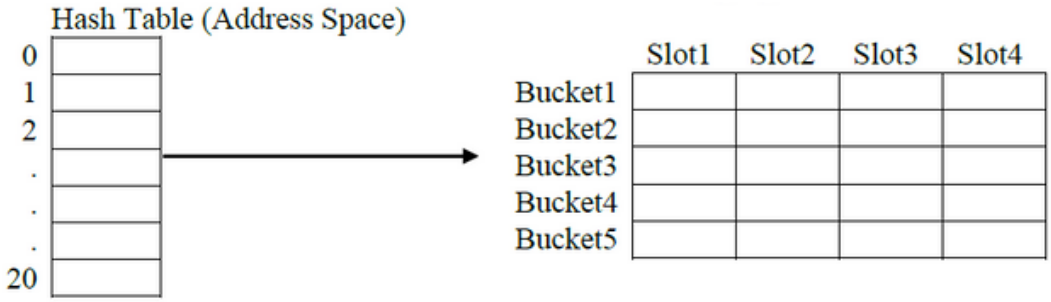
Hash(K, p) = 0

- so 103 will be stored at location 0

0	103
1	___
2	___
3	33
4	___
5	___
6	___
7	___
8	93
9	___

Explain Bucket Hashing With Example

- Bucket hashing is a type of collision resolution technique which is used to avoid collision of values/keys ate the same positions
- The bucket is simply a large space used to hold multiple records, the address space is divided into multiple buckets
- ‘M’ Addresses of the address space are divided into ‘B’ buckets
- these buckets are further divided into slots
- the number of slots in each bucket = M/B
- For example the Hash table with 20 addresses which are divided into 5 buckets will have 20/5 = 4 slots in each bucket



For 1st key, K=22, the bucket address generated
will be:
H(K) = K mod m
H(22) = 22 mod 5
= 2 mod 5
= 2

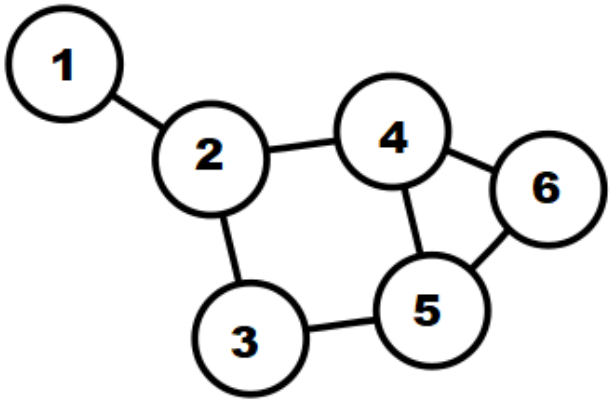
22		

For 2nd key, K=101, the bucket address generated
will be:
H(K) = K mod m
H(101) = 101 mod 5
= 1 mod 5
= 1

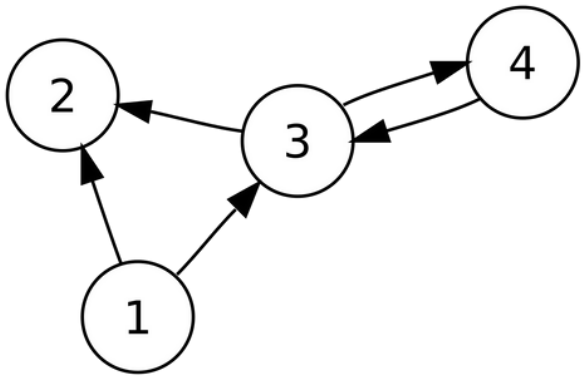
101		
22		

What is Graph?

- A graph G consist of finite number of vertices/nodes (V) and finite number of Edges/Path (E) which can be denoted as G = (V E)
- Here set of vertices V represent the entities which has some name, an Edge is a line that connect a pair of vertices



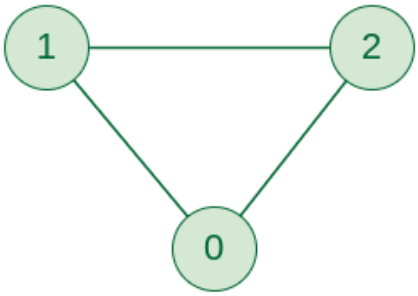
- In directed graph each edge is assigned a direction



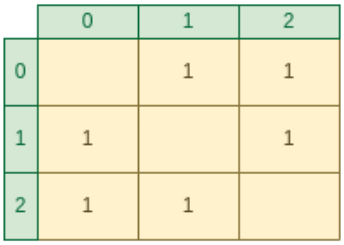
- The outdegree of the vertex V in directed graph is the number of Edges going out from the Vertex, while Indegree are the number of Edges Coming to the Vertex

Adjacency Matrix Representation of Graph

Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 to both cases

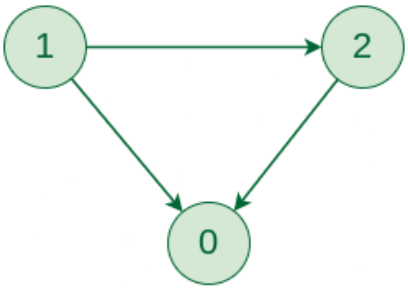


Undirected Graph

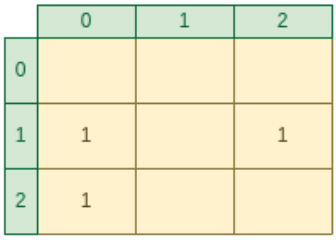


Adjacency Matrix

Initially, the entire Matrix is initialized to 0. If there is an edge from source to destination, we insert 1 for that particular destination



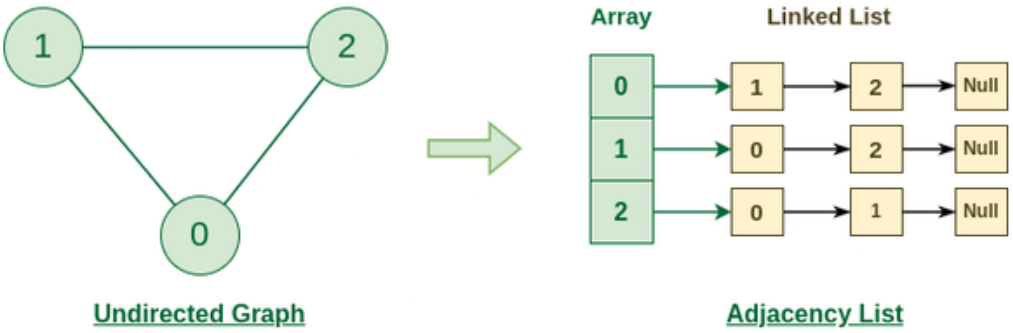
Directed Graph



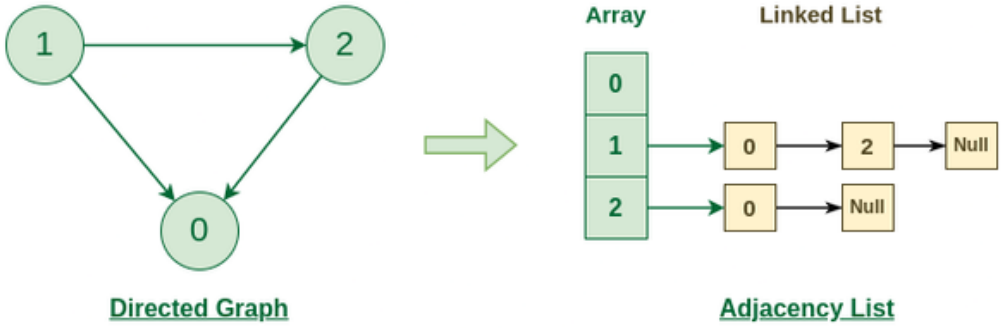
Adjacency Matrix

Linked List Representation of Graph

The below undirected graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has two neighbours (i.e, 1 and 2). So, insert vertex 1 and 2 at indices 0 of array. Similarly, For vertex 1, it has two neighbour (i.e, 2 and 1) So, insert vertices 2 and 1 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.

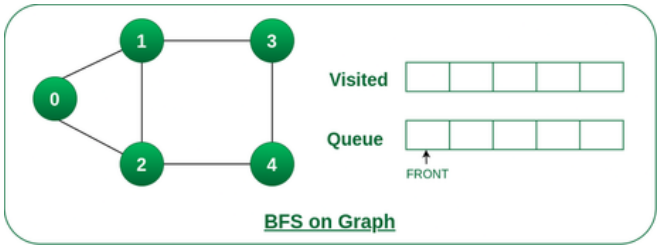


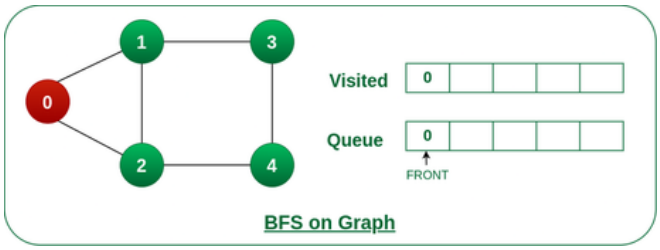
The below directed graph has 3 vertices. So, an array of list will be created of size 3, where each indices represent the vertices. Now, vertex 0 has no neighbours. For vertex 1, it has two neighbour (i.e, 0 and 2) So, insert vertices 0 and 2 at indices 1 of array. Similarly, for vertex 2, insert its neighbours in array of list.

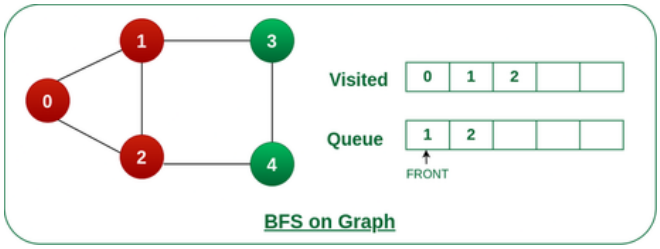


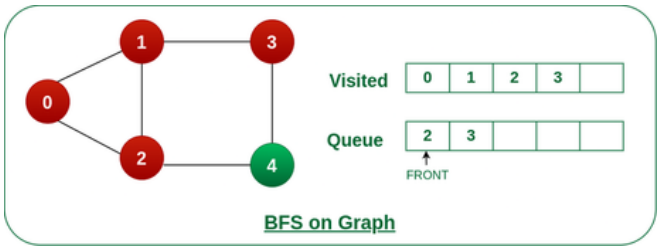
Explain BFS (Breath First Search)

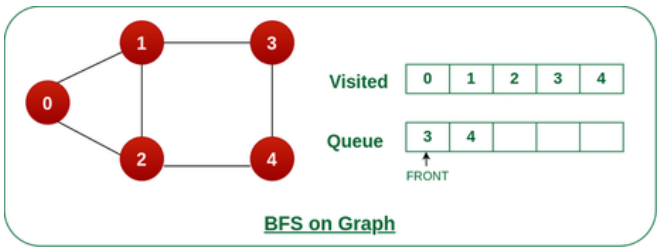
- Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.
- To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.







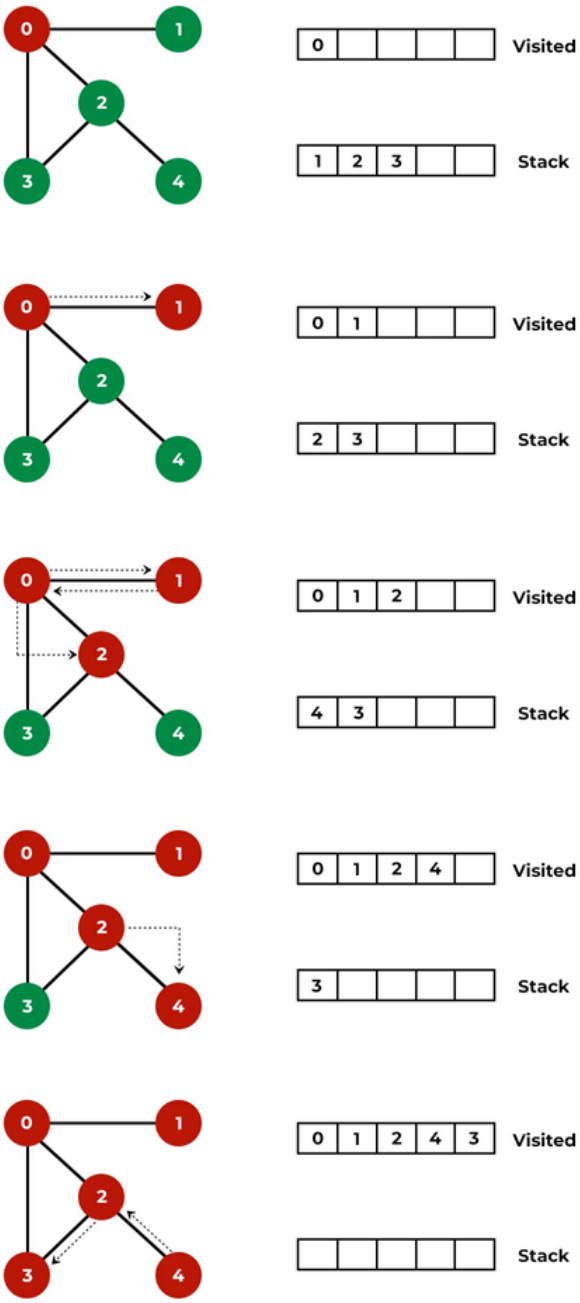




- Add 0 to the queue
- We Visit the 0 first and we will mark it as visited
- Remove 0 from queue
- We will then visit the neighbors of 0 (i.e. 1 and 2)
- Add 1 and 2 in the queue
- we visit 1 and 2 respectively and mark them as visited
- Remove both 1 and 2
- Add next of 1 to the queue (3)
- Visit the unvisited neighbor of 1 (i.e., 3)
- Remove 3 from the queue
- mark 3 as visited
- Add next of 2 to the queue
- Visit the neighbor of 2 (i.e.,4)
- remove 4 from the queue
- mark 4 as visited

Explain DFS (Deep First Search)

- Depth-first search is an algorithm for traversing or searching graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

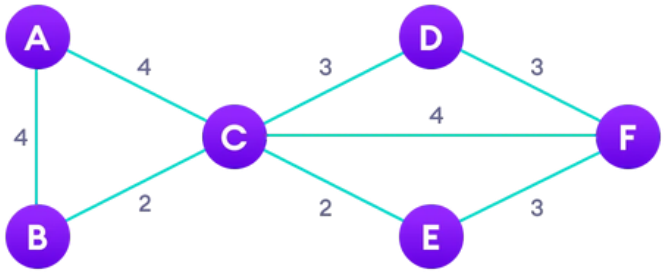


- Add 0 to the queue
- We Visit the 0 first and we will mark it as visited
- Remove 0 from queue
- We will then visit the neighbors of 0 (i.e. 1, 2 and 3)
- Add 1, 2 and 3 in the queue
- we visit 1 and mark it as visited
- Remove 1 from queue
- as we cannot go further than 1 we will go to 2
- mark 2 as visited and remove 2 from queue
- We can go to 4 from 2 so we will add 4 to the first at the queue
- visit 4, mark 4 as visited and remove 4 from the queue
- visit 3, mark 3 as visited and remove 3 from queue

Explain Prims Algorithm

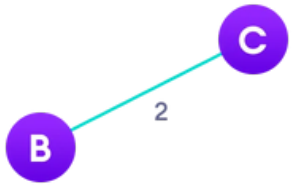
- Prim's algorithm, also known as Prim's minimum spanning tree algorithm, is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected graph.
- A minimum spanning tree is a subgraph of the original graph that includes all the vertices and a subset of the edges in such a way that it forms a tree (i.e., there are no cycles) and has the minimum possible total edge weight.

- Start with a weighted graph
- Choose a vertex
- Choose the shortest edge from this vertex and add it
- Choose the nearest vertex not yet visited
- Choose the nearest edge not yet visited, if there are multiple choices, choose one at random
- Make sure there is no loop while connecting the vertexes
- Repeat until you have a spanning tree (Edges = Vertex - 1)

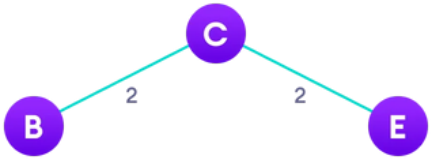


Step: 1

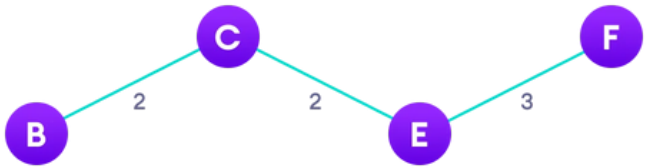
Step: 2



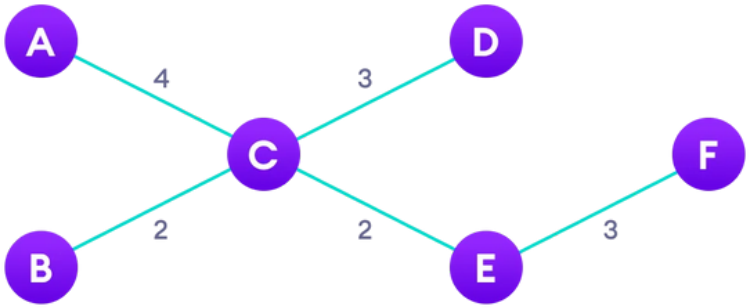
Step: 3



Step: 4



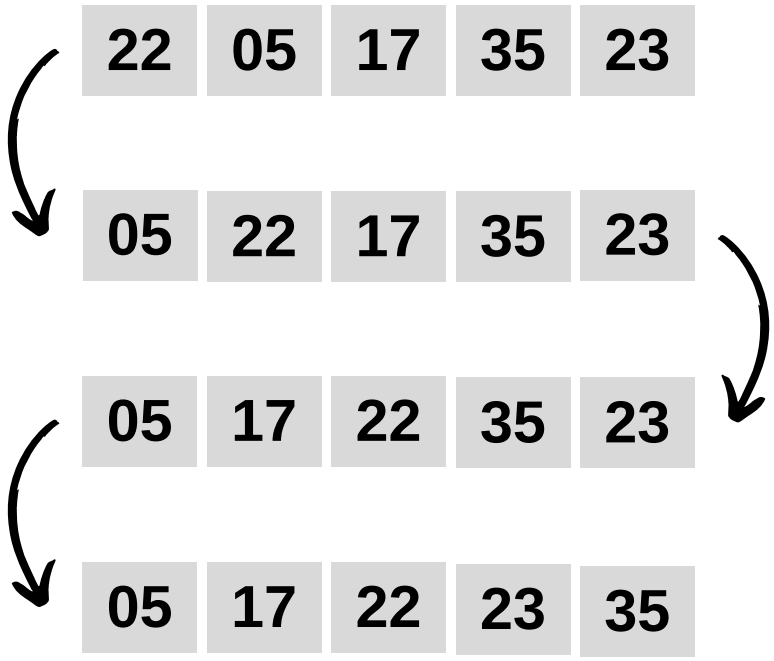
Step: 5



Step: 6

Explain Selection Sorting with Example

- Selection sorting algorithm is also called as in-place comparison sort
- in-place means that this algorithm does not take extra space for sorting the elements of the array
- in this algorithm the smallest element of the array is replace with the element at the first position
- Then the second smallest element is swapped with the element at 2nd position
- this process continues until the entire array is sorted



- 1st step : 5 is the smallest element so we will exchange its position with the element at position 1
- 2nd Step : 17 is the second smallest element so we will exchange its position with element at position 2
- We will continue this until the entire array is sorted

Selection Sorting algorithm

```
for (i = 0 ; i < n-1 ; i++)
{
    int min = 1
    for (j = 1 ; j <n ; j++)
    {
        if ( a[j] < min)
        {
            min = j;
        }
        if ( min! = 1 )
        {
            swap (a[i] , min)
        }
    }
}
```