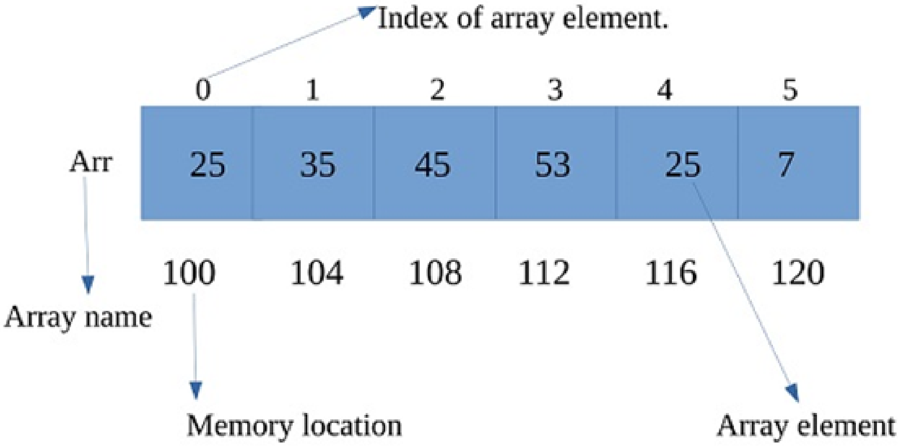
DATA STRUCTURE NOTES

- By Dipesh Adelkar

Explain One Dimensional Array

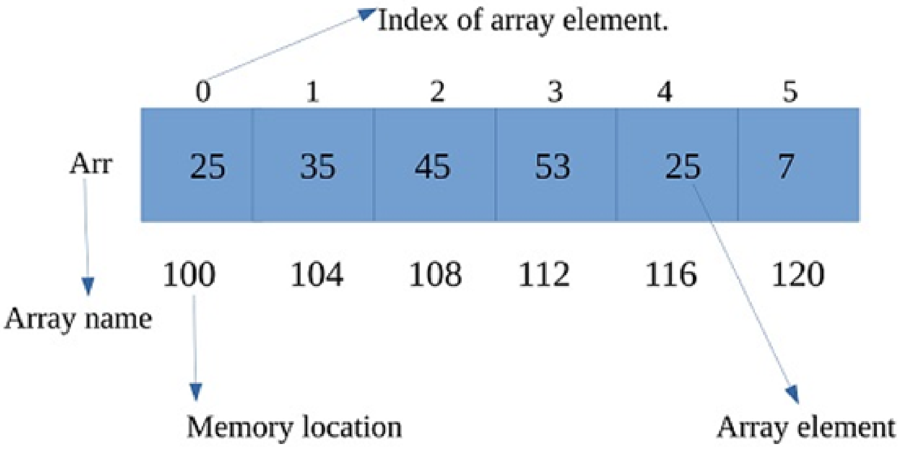
- An array is a linear collection of finite number of homogeneous elements
- Elements in an array are in a sequence
- The elements are homogeneous which means the elements are of similar datatype
- All the elements in an array are stored in consecutive memory location
- Example of array



- 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

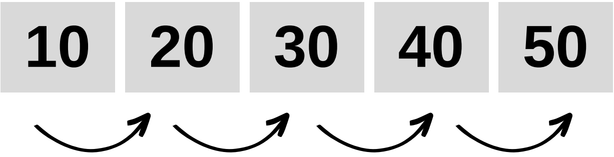
Hence element at index 3 is stored at location 112 in memory

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



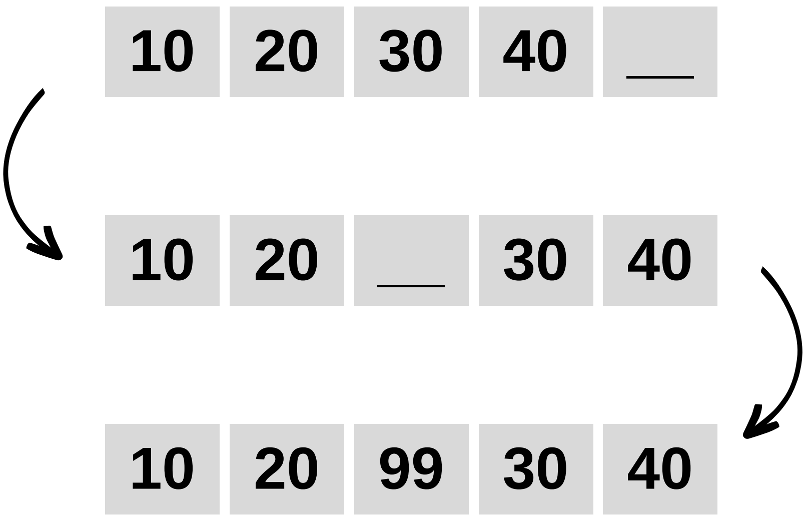
10	20	30	40	50

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



10 20 30 40 ___

10 20 ___ 30 40

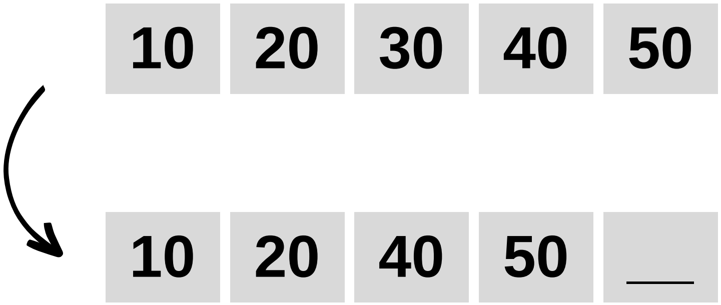
10 20 99 30 40

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



10	20	30	40	50
10	20	40	50	__

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

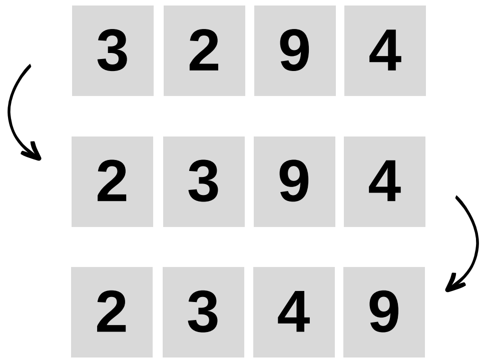
Explain Bubble Sorting of Array

- 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
```



3 2 9 4

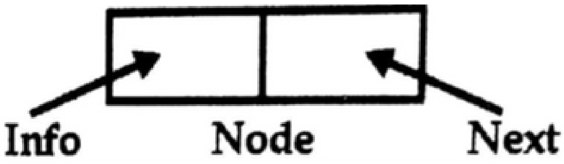
2 3 9 4

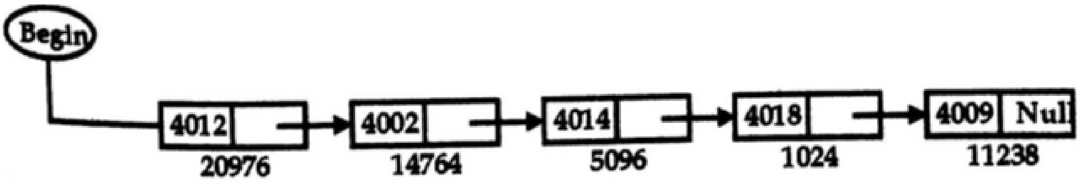
2 3 4 9

- 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
```

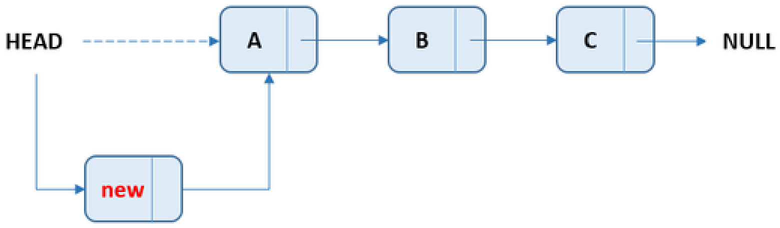
- Here we check if the linked list is empty or not
- if the list is not empty we execute step 2 , Pointer is assigned to the first node which carries the address of the node
- Info part of the current element gets printed and then the pointer is assigned to the address of next node
- once pointer encounters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

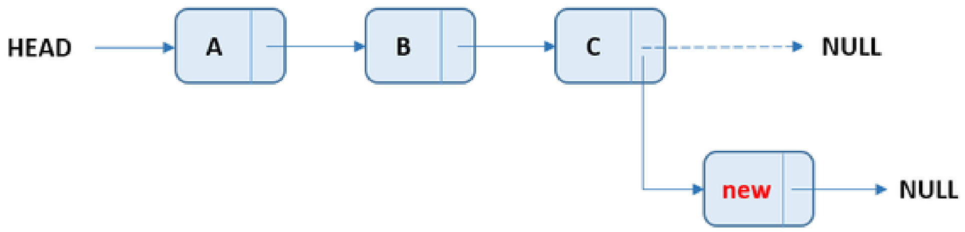
Insertion at Beginning of Linked List



```
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



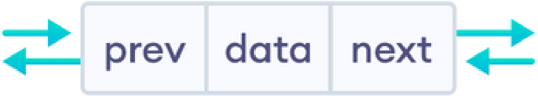
```
Step 1    If Free = Null Then
              Print "No Free space available"
              Exit
          [End If]
Step 2    Allocate space to New
          (Set New = Free and Free = Free -> Next)
Step 3    Set New -> Info = Data, New -> Next = Null
Step 4    If Begin = Null then
              Begin  = 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

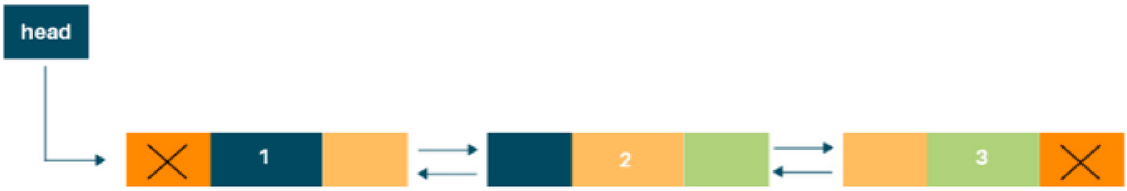
```
1   If Begin = Null Then
        Print “Linked List is Empty”
        Exit
    [End If]
2   Set Item = Begin -> Info And Pos = Begin
3   Set Begin = Begin -> Next
4   Set Pos -> Next = Free and Free = Pos
5   Exit
```

- 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

```
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
```

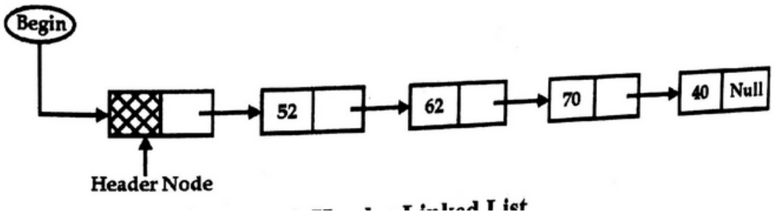
Insertion at Beginning of Double Linked List

```
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
```

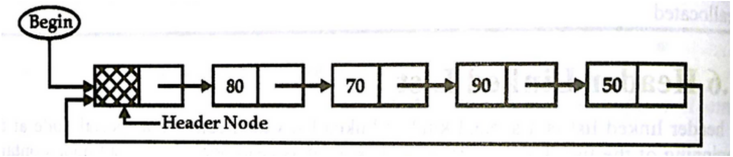
Explain Header Linked List

- 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

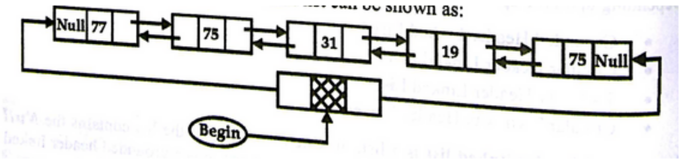
- Grounded Linked List - the last node contains Null in its Next Pointer



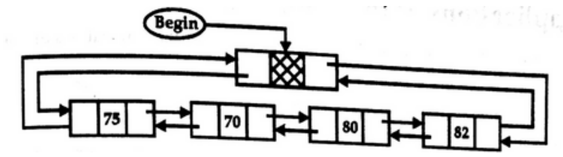
- Circular Linked List - The last node points back to the Header node



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

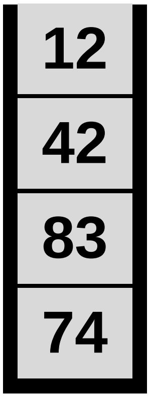


- Two Way Circular Header Linked List - The Last node of the Two way Linked list points back to the Header node



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”



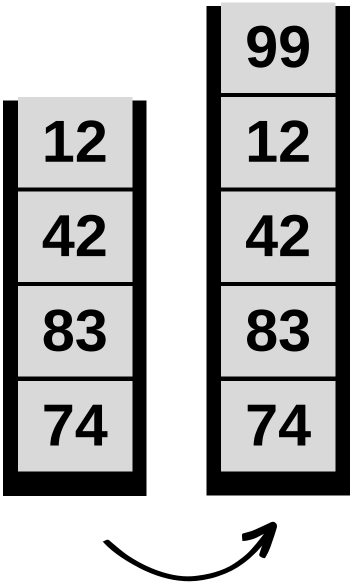
- PUSH - Push operation refers to insertion of a new element into the stack
- we can perform PUSH operation only if the stack is not full i.e. the stack should have sufficient space for new element
- If the stack is already full and when we try to insert new element it is called as “Stack Overflow” condition

- 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

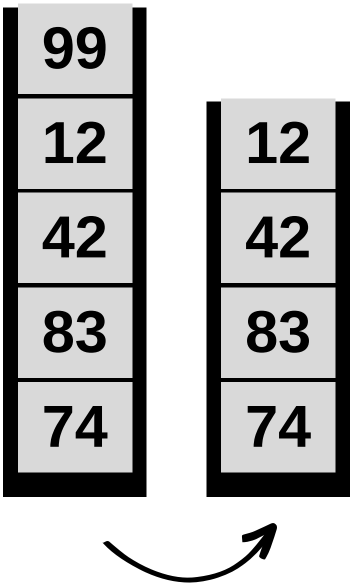
PUSH operation

```
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
```



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/

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

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

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

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

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

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

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
    1. linear Probing
    2. Quadratic Probing
    3. 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.

$$\text{Hash}(K, p) = [\ H(K) + p\ ] \bmod m$$

- 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.

$$\text{Hash}(K, p) = [\ H1(K) + pH2(K)\ ] \bmod m$$

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

K = 33

$$\text{Hash}(K, p) = [\ H1(K) + pH2(K)\ ] \bmod m$$

$$\text{Hash}(K, p) = [\ 33 \bmod 10 + 0 \times 33 \bmod 8\ ] \bmod 10$$

$$\text{Hash}(K, p) = 3 \bmod 10$$

$$\text{Hash}(K, p) = 3$$

K = 93

$$\text{Hash}(K, p) = [\ H1(K) + pH2(K)\ ] \bmod m$$

$$\text{Hash}(K, p) = [\ 93 \bmod 10 + 0 \times 93 \bmod 8\ ] \bmod 10$$

$$\text{Hash}(K, p) = 3 \bmod 10$$

$$\text{Hash}(K, p) = 3$$

- 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

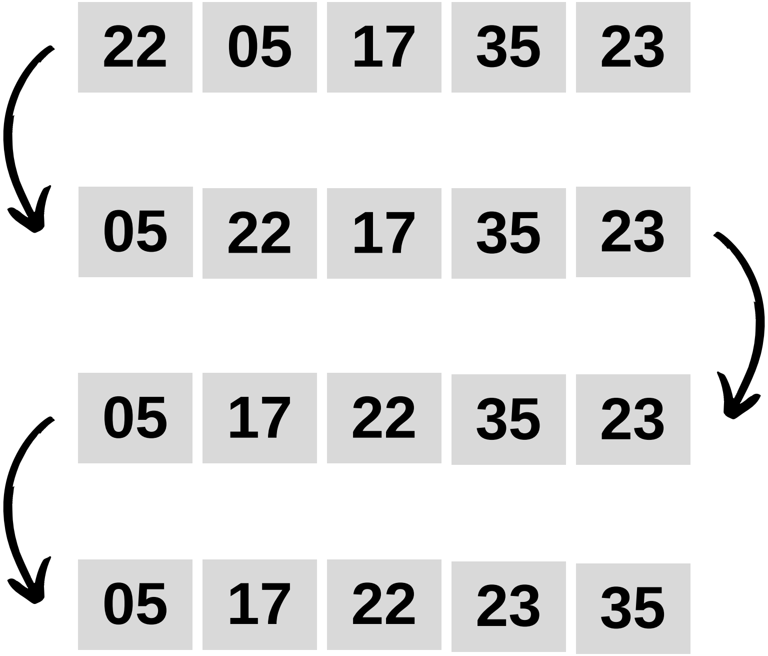
Hash(K, p) = 0

- so 103 will be stored at location 0

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

Explain 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



22 05 17 35 23

05 22 17 35 23

05 17 22 35 23

05 17 22 23 35

- 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)
        }
    }
}
```

NOTE

These notes are not yet finished

Sorry for the inconvenience, will be providing

the whole syllabus notes soon...