Course Name : PG DAC
Module Name : Algorithms & Data Structures Using Java.

Introduction:
Q. Why there is a need of data structures?
Q. What is data structures?

=> To store marks of 100 students:
int m1, m2, m3, m4, ..., m100;//sizeof(int): 4 bytes => 400 bytes

int marks[100];//400 bytes

=> Array : it is a basic/linear data structure, which is a collection/list of logically related similar type of elements gets stored into the memory at contiguos locations.

Q. Why array indexing starts from 0?
- to convert array notation into its equivalent notation is done by the compiler, (to maintained link between array elements is the job of compiler).
arr[i] ~= *(arr + i)

```
struct student
{
     int rollno;
     char name[32];
     float marks;
};
```

<data type> <var_name>;
- data type may be any primitive/non-primitive data type
- var_name is an identifier
e.g.
int n1;

struct student s1;//abstraction => abstract data type
struct student s2;

+ class => it is a linear/basic data structure which is a collection/list of logically related similar and disimmilar type of data elements referred as data members as well as functions which can be used to perform operations on data members referred as member funcction/methods.

e.g.

```
class student
{
	//data members
	private int rollno;
	private String name;
	private float salary;

	//methods:
	//ctor
	//mutators
	//getter functions
	//setter functions
	//facilitators
}


student s1;//ADT
```

** to learn data structures is not to learn any specific programming langauge, it is nothing but to learn an algorithms, and algorithms can be implemented by using any programming langauge (using Java).

Q. What is an algorithm?

+ traversal of an array => to visit each array element sequentially from first element max till last element.

- Algorithm to do sum of array elements: => Any User

step-1: initially take sum as 0
step-2: traverse an array and add each array element sequentially into the sum.
step-3: return final value of sum.

- Pseudocode to do sum of array elements: => Programmer User

```
Algorithm ArraySum(A, size){
	sum = 0;
	for( index = 1 ; index <= size ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```

- Program to do sum of array elements: => Machine

```
int ArraySum(int [] arr, int size){
	int sum = 0;
	for( index = 0 ; index < size ; index++ ){
		sum += arr[ index ];
	}
	return sum;
}
```

Bank Project => Bank Manager => Algorithm => Project Manager => Software Architect => Design Pseudocode => Developers => Programs => Machine

Q. What is a recursion ?
- it is a process in which function can be called within itself, such a function is referred as recursive function.

- function call for which calling function and called function are same, is referred as recursive function call.

- any thing can be defined in terms itself

```
main( ){
     print("sum = "+recArraySum(arr, 0) );//first time function calling to the rec function
     //calling function => main()
     //called function => recArraySum()
}

int recArraySum(int [] arr, int index ){
     if( index == arr.length )
          return 0;

     return ( arr[ index ] + recArraysum(arr, index+1) );
     //calling function => recArraySum()
     //called function => recArraySum()

}
```

- to delete function activation record / stack frame from stack called as stack cleanup is done either by calling function or called function and it depends on function calling convention.

```
main(){
     print("sum"+sum(10, 20);
     //10 & 20 => actual params
}

int sum(int n1, int n2){
     int sum;
     sum = n1 + n2;
     return sum;
}

//n1 & n2 => formal params
//sum => local var
```

+ function calling conventions:

__std__

__c__

__pascal

function calling conventions decides 2 things:

1. in which oreder params should gets passed to the function i.e. either from L -> R Or R -> L
2. stack cleanup

- when any function gets called an OS creates one entry onto the stack for that function call, called as function activation record/stack frame and it gets popped or removed from the stack when an execution of that function is completed and process to remove stack frame from stack is called as stack cleanup.

City-1:

City-2:

- there may exists multiple paths between 2 cities, in this case we need to decide an optimum/efficient path
- there are some factors/measures on which efficient/optimum path can be decides:
    - time
    - distance
    - cost
    - traffic condition
    - status of path
    - etc...

Space Complexity:

Space Complexity = Code Space + Data Space + Stack Space

Code Space => space required for an instructions

Data Space => space required for simple vars, constants and instance vars

Stack Space (applicable only in recursive algorithm) => space reqquired for FAR's.

- there are components of space complexity:

1. fixed component : code space + data space (space required for simple vars and constants).

2. variable component : data space (space required for instance vars) & stack space (applicable only in recursive algorithms).

Example:

```
Algorithm ArraySum(A, n){
     sum = 0;
     for( index = 1 ; index <= n ; index++ ){
          sum += A[ index ];
     }
     return sum;
}
```

S = C (Code Space) + Sp (Data Space)

Code Space =>
if size of an array = 5 => no. of instructions will be same
if size of an array = 10 => no. of instructions will be same
if size of an array = 100 => no. of instructions will be same
.
.
if size of an array = n => no. of instructions will be same

for any input size array no. of instructions in an algo will going to remains same => it will take constant amount of space for any input size array.

Sp = space required for simple vars + space required for constants + space required for instance vars

simple vars: A, sum, index => 3 units
1 unit of memory => A
1 unit of memory => sum
1 unit of memory => index

instaance var => n =>
for size of an array is 5 i.e. n = 5 => 5 units
for size of an array is 10 i.e. n = 10 => 10 units
for size of an array is 100 i.e. n = 100 => 100 units
.
.
for size of an array is n => n units => instance var

index++ => index = index + 1;

- quick sort
- merge sort
- implementation of advanced data structures algo's like - traversal methods in tree.

```
Algorithm ArraySum(A, n){
      sum = 0;
      for( index = 1 ; index <= n ; index++ ){
            sum += A[ index ];
      }
      return sum;
}
```

- for any input size array, no. of instructions will be remain same, hence compilation time also remains same for any input size array.

Asymptotic Analysis:

Searching => to search/find key element in a given collection/list of elements.
- there are basic two searching algorithms:
1. Linear Search:
2. Binary Search:

1. Linear Search/ Sequential Search:

```
Algorithm LinearSearch(A, n, key){//A is an array of size n
      for( index = 1 ; index <= n ; index++ ){
            if( key == A[ index ] )
                  return true;
      }

      return false;
}
```

\# Best Case : if key is found in an array at first position
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array no. of comparisons in best case = 1 and hence in this case linear search algo takes O(1) time.

Worst Case : if either key is found in an array at last position or key does not exists.

if size of an array = 10 => no. of comparisons = 10
if size of an array = 20 => no. of comparisons = 20
.
.
if size of an array = n => no. of comparisons = n

in worst case no. of comparisons depends on an input size of an array, hence running time of linear search algo in worst case is O(n).

Rule:
- if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected/ignored.

e.g.
O(n + 4) => O(n)
O(n - 2) => O(n)
O(n / 2) => O(n)
O(3 * n) => O(n)

Home Work : to implement Linear Search => by using recursion as well as non-recursive method.

DAY-02:
Searching Algorithms
Sorting Algorithms
Limitation of an array data structure

Course Name : PG DAC
Module Name : Algorithms & Data Structures Using Java.

DAY-01:

Introduction:
Q. Why there is a need of data structures?
Q. What is data structures?

=> To store marks of 100 students:

```
int m1, m2, m3, m4, ......, m100;//sizeof(int): 4 bytes => 400
bytes

int marks[ 100 ];//400 bytes
```

=> Array : it is a basic/linear data structure, which is a collection/list of logically related similar type of elements gets stored into the memory at contiguos locations.

Q. Why array indexing starts from 0?
- to convert array notation into its equivalent notation is done by the compiler, (to maintained link between array elements is the job of compiler).

```
arr[ i ] ~= *(arr + i)

struct student
{
     int rollno;
     char name[32];
     float marks;
};


<data type> <var_name>;
```

- data type may be any primitive/non-primitive data type
- var_name is an identifier

e.g.

```
int n1;

struct student s1;//abstraction => abstract data type
struct student s2;
```

+ class => it is a linear/basic data structure which is a collection/list of logically related similar and disimmilar type of data elements referred as data members as well as functions which can be used to perform operations on data members referred as member funcction/methods.

e.g.
```
class student
{
	//data members
	private int rollno;
	private String name;
	private float salary;

	//methods:
	//ctor
	//mutators
	//getter functions
	//setter functions
	//facilitators
}


student s1;//ADT
```

** to learn data structures is not to learn any specific programming langauge, it is nothing but to learn an algorithms, and algorithms can be implemented by using any programming langauge (using Java).

Q. What is an algorithm?

+ traversal of an array => to visit each array element sequentially from first element max till last element.

- Algorithm to do sum of array elements: => Any User
step-1: initially take sum as 0
step-2: traverse an array and add each array element sequentially into the sum.
step-3: return final value of sum.

- Pseudocode to do sum of array elements: => Programmer User
```
Algorithm ArraySum(A, size){
	sum = 0;
	for( index = 1 ; index <= size ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```


- Program to do sum of array elements: => Machine
```
int ArraySum(int [] arr, int size){
	int sum = 0;
	for( index = 0 ; index < size ; index++ ){
		sum += arr[ index ];
	}
	return sum;
}
```

Bank Project => Bank Manager => Algorithm => Project Manager => Software Architect => Design Pseudocode => Developers => Programs => Machine

Q. What is a recursion ?
- it is a process in which function can be called within itself, such a function is referred as recursive function.

- function call for which calling function and called function are same, is referred as recursive function call.

- any thing can be defined in terms itself

```
main( ){
     print("sum = "+recArraySum(arr, 0) );//first time function
calling to the rec function
     //calling function => main()
     //called function => recArraySum()
}

int recArraySum(int [] arr, int index ){
     if( index == arr.length )
          return 0;

     return ( arr[ index ] + recArraysum(arr, index+1) );
     //calling function => recArraySum()
     //called function => recArraySum()

}
```

- to delete function activation record / stack frame from stack called as stack cleanup is done either by calling function or called function and it depends on function calling convention.

```
main(){
     print("sum"+sum(10, 20);
     //10 & 20 => actual params
}

int sum(int n1, int n2){
     int sum;
     sum = n1 + n2;
     return sum;
}

//n1 & n2 => formal params
//sum => local var
```

+ function calling conventions:
__std__
__c__
__pascal

function calling conventions decides 2 things:
1. in which oreder params should gets passed to the function
i.e. either from L -> R Or R -> L
2. stack cleanup

- when any function gets called an OS creates one entry onto the stack for that function call, called as function activation record/stack frame and it gets popped or removed from the stack when an execution of that function is completed and process to remove stack frame from stack is called as stack cleanup.

City-1:
City-2:
- there may exists multiple paths between 2 cities, in this case we need to decide an optimum/efficient path
- there are some factors/measures on which efficient/optimum path can be decides:
    time
    distance
    cost
    traffic condition
    status of path
    etc...

Space Complexity:

Space Complexity = Code Space + Data Space + Stack Space

Code Space => space required for an instructions
Data Space => space required for simple vars, constants and instance vars
Stack Space (applicable only in recursive algorithm) => space reqquired for FAR's.

- there are components of space complexity:
1. fixed component : code space + data space (space required for simple vars and constants).

2. variable component : data space (space required for instance vars) & stack space (applicable only in recursive algorithms).

Example:

```
Algorithm ArraySum(A, n){
     sum = 0;
     for( index = 1 ; index <= n ; index++ ){
          sum += A[ index ];
     }
     return sum;
}
```

S = C (Code Space) + Sp (Data Space)

Code Space =>
if size of an array = 5 => no. of instructions will be same
if size of an array = 10 => no. of instructions will be same
if size of an array = 100 => no. of instructions will be same
.
.
if size of an array = n => no. of instructions will be same

for any input size array no. of instructions in an algo will going to remains same => it will take constant amount of space for any input size array.

Sp = space required for simple vars + space required for constants + space required for instance vars

simple vars: A, sum, index => 3 units
1 unit of memory => A
1 unit of memory => sum
1 unit of memory => index

instaance var => n =>
for size of an array is 5 i.e. n = 5 => 5 units
for size of an array is 10 i.e. n = 10 => 10 units
for size of an array is 100 i.e. n = 100 => 100 units
.
.
for size of an array is n => n units => instance var

```
index++ => index = index + 1;
```

- quick sort
- merge sort
- implementation of advanced data structures algo's like - traversal methods in tree.

```
Algorithm ArraySum(A, n){
      sum = 0;
      for( index = 1 ; index <= n ; index++ ){
            sum += A[ index ];
      }
      return sum;
}
```

- for any input size array, no. of instructions will be remain same, hence compilation time also remains same for any input size array.

Asymptotic Analysis:

Searching => to search/find key element in a given collection/list of elements.
- there are basic two searching algorithms:
1. Linear Search:
2. Binary Search:

1. Linear Search/ Sequential Search:

```
Algorithm LinearSearch(A, n, key){//A is an array of size n
      for( index = 1 ; index <= n ; index++ ){
            if( key == A[ index ] )
                  return true;
      }

      return false;
}
```

Best Case : if key is found in an array at first position
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array no. of comparisons in best case = 1 and hence in this case linear search algo takes O(1) time.

Worst Case : if either key is found in an array at last position or key does not exists.

if size of an array = 10 => no. of comparisons = 10
if size of an array = 20 => no. of comparisons = 20
.
.
if size of an array = n => no. of comparisons = n

in worst case no. of comparisons depends on an input size of an array, hence running time of linear search algo in worst case is O(n).

Rule:
- if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected/ignored.
e.g.
O(n + 4) => O(n)
O(n - 2) => O(n)
O(n / 2) => O(n)
O(3 * n) => O(n)

Home Work : to implement Linear Search => by using recursion as well as non-recursive method.

DAY-01:
- introduction to ds:
    why there is a need of ds ?
    what is a ds and its types
    what is an algorithm, program, pseudocode
    types of algorithms: iterative/non-recursive and recursive
    what is a recursion? recursive function, rec function call
    types of rec functions: tail & non-tail recursive
    analysis of an algo: space complexity & time complexity

    asymptotic analysis => it is a mathematical way to calculate time complexity and space complexity of an algo without implementing it in any porgramming language.

DAY-02:
2. Binary Search:

by means of calculating mid position, big size array gets divided
logically into two subarray’s:
left subarray and right subarray

for left subarray => value of left remains same, right = mid-1
for right subarray => value of right remains same, left = mid+1

best case : if key is found in an array in very first iteration
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
if size of an array = 100 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array, in best case no. Of comparisons = 1,
hence running time of an algo in best case = $O(1)$
time complexity of binary search in best case => $\Omega(1)$.

if(left <= right) => subarray is valid
OR
if(left > right) => subarray is invalid

n = 1000

iteration-1:
search space = 1000 = n

mid=500
[0.... 499] 500 [501 .... 1000] => no. Of comparisons=1

iteration-2:
search space = 500 = n/2

[0...249] 250 [251 ... 499] => no. Of comparisons=1

iteration-3:
search space = 250 => n / 4
[0 ...124] 125 [126 ... 250] => no. Of comparisons=1

.
.
.

if size of an array = 1 => trivial case => T(1) = O(1)

if size of an array > 1 i.e. size of an array = n

$T(n) = T(n) + 1$

after iteration-1:
$T(n) = T(n / 2) + 1 \Rightarrow T(n / 2^1) + 1$

after iteration-2:
$T(n) = T(n / 4) + 2 \Rightarrow T(n / 2^2) + 2$

after iteration-3:
$T(n) = T(n / 8) + 3 \Rightarrow T(n / 2^3) + 3$

.
.

lets assume, k no. of iterations takes place
after k iterations:
$T(n) = T(n / 2^k) + k$

lets assume kth iteration is the last iteration
assume => $n \Rightarrow 2^k$

=> $n \Rightarrow 2^k$
=> $\log n = \log 2^k$ ...... by taking log on both sides
=> $\log n = k \log 2$
=> $\log n = k$ ..... [$\log 2 \sim= 1$]
=> $\underline{k = \log n}$

$T(n) = T(n / 2^k) + k$
substitute values of $n = 2^k$ & $k=\log n$ in above equation, we get
$T(n) = T(2^k / 2^k) + \log n$
$T(n) = T(1) + \log n$
$T(n) = \log n$
$T(n) = O(\log n)$

+ Sorting Algorithms:

Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.

- basic sorting algorithms:
1. selection sort
2. bubble sort
3. insertion sort

- advanced sorting algorithm:
4. quick sort
5. merge sort

1. selection sort:

for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = n(n-1) / 2 => $(n^2 - n) / 2$
$T(n) = O((n^2 - n) / 2)$
$T(n) = O(n^2 - n)$ .... divisive can be neglected
$\underline{T(n) = O(n^2)}$

Rule:
if running time of an algo is having a polynomial, then only leading term in it will be considered in its time complexity.
e.g.
$O(n^3 + n^2 + 4) => O(n^3)$
$O(n^2 + 5) => O(n^2)$

Course Name : PG DAC
Module Name : Algorithms & Data Structures Using Java.

DAY-01:

Introduction:
Q. Why there is a need of data structures?
Q. What is data structures?

=> To store marks of 100 students:

```
int m1, m2, m3, m4, ......, m100;//sizeof(int): 4 bytes => 400
bytes

int marks[ 100 ];//400 bytes
```

=> Array : it is a basic/linear data structure, which is a collection/list of logically related similar type of elements gets stored into the memory at contiguos locations.

Q. Why array indexing starts from 0?
- to convert array notation into its equivalent notation is done by the compiler, (to maintained link between array elements is the job of compiler).

```
arr[ i ] ~= *(arr + i)

struct student
{
     int rollno;
     char name[32];
     float marks;
};
```

```
<data type> <var_name>;
```

- data type may be any primitive/non-primitive data type
- var_name is an identifier

e.g.

```
int n1;

struct student s1;//abstraction => abstract data type
struct student s2;
```

+ class => it is a linear/basic data structure which is a collection/list of logically related similar and disimmilar type of data elements referred as data members as well as functions which can be used to perform operations on data members referred as member funcction/methods.

```
e.g.
class student
{
	//data members
	private int rollno;
	private String name;
	private float salary;

	//methods:
	//ctor
	//mutators
	//getter functions
	//setter functions
	//facilitators
}


student s1;//ADT
```

** to learn data structures is not to learn any specific programming langauge, it is nothing but to learn an algorithms, and algorithms can be implemented by using any programming langauge (using Java).

Q. What is an algorithm?

+ traversal of an array => to visit each array element sequentially from first element max till last element.

- Algorithm to do sum of array elements: => Any User
step-1: initially take sum as 0
step-2: traverse an array and add each array element sequentially into the sum.
step-3: return final value of sum.

- Pseudocode to do sum of array elements: => Programmer User

```
Algorithm ArraySum(A, size){
	sum = 0;
	for( index = 1 ; index <= size ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```


- Program to do sum of array elements: => Machine

```
int ArraySum(int [] arr, int size){
	int sum = 0;
	for( index = 0 ; index < size ; index++ ){
		sum += arr[ index ];
	}
	return sum;
}
```

Bank Project => Bank Manager => Algorithm => Project Manager => Software Architect => Design Pseudocode => Developers => Programs => Machine

Q. What is a recursion ?
- it is a process in which function can be called within itself, such a function is referred as recursive function.

- function call for which calling function and called function are same, is referred as recursive function call.

- any thing can be defined in terms itself
example:

```
main( ){
     print("sum = "+recArraySum(arr, 0) );//first time function
calling to the rec function
     //calling function => main()
     //called function => recArraySum()
}

int recArraySum(int [] arr, int index ){
     if( index == arr.length )
          return 0;

     return  ( arr[ index ] + recArraysum(arr, index+1) );
     //calling function => recArraySum()
     //called function => recArraySum()

}
```

- to delete function activation record / stack frame from stack called as stack cleanup is done either by calling function or called function and it depends on function calling convention.

```
main(){
     print("sum"+sum(10, 20);
     //10 & 20 => actual params
}

int sum(int n1, int n2){
     int sum;
     sum = n1 + n2;
     return sum;
}

//n1 & n2 => formal params
//sum => local var
```

+ function calling conventions:
__std__
__c__
__pascal

function calling conventions decides 2 things:
1. in which oreder params should gets passed to the function i.e. either from L -> R Or R -> L
2. stack cleanup

- when any function gets called an OS creates one entry onto the stack for that function call, called as function activation record/stack frame and it gets popped or removed from the stack when an execution of that function is completed and process to remove stack frame from stack is called as stack cleanup.

City-1:
City-2:
- there may exists multiple paths between 2 cities, in this case we need to decide an optimum/efficient path
- there are some factors/measures on which efficient/optimum path can be decides:
    time
    distance
    cost
    traffic condition
    status of path
    etc...

Space Complexity:

Space Complexity = Code Space + Data Space + Stack Space

Code Space => space required for an instructions
Data Space => space required for simple vars, constants and instance vars
Stack Space (applicable only in recursive algorithm) => space reqquired for FAR's.

- there are components of space complexity:
1. fixed component : code space + data space (space required for simple vars and constants).

2. variable component : data space (space required for instance vars) & stack space (applicable only in recursive algorithms).

Example:

```
Algorithm ArraySum(A, n){
      sum = 0;
      for( index = 1 ; index <= n ; index++ ){
            sum += A[ index ];
      }
      return sum;
}
```

S = C (Code Space) + Sp (Data Space)

Code Space =>
if size of an array = 5 => no. of instructions will be same
if size of an array = 10 => no. of instructions will be same
if size of an array = 100 => no. of instructions will be same
.
.
if size of an array = n => no. of instructions will be same

for any input size array no. of instructions in an algo will going to remains same => it will take constant amount of space for any input size array.

Sp = space required for simple vars + space required for constants + space required for instance vars

simple vars: A, sum, index => 3 units
1 unit of memory => A
1 unit of memory => sum
1 unit of memory => index

instaance var => n =>
for size of an array is 5 i.e. n = 5 => 5 units
for size of an array is 10 i.e. n = 10 => 10 units
for size of an array is 100 i.e. n = 100 => 100 units
.
.
for size of an array is n => n units => instance var

index++ => index = index + 1;

- quick sort
- merge sort
- implementation of advanced data structures algo’s like - traversal methods in tree.

```
Algorithm ArraySum(A, n){
	sum = 0;
	for( index = 1 ; index <= n ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```

- for any input size array, no. of instructions will be remain same, hence compilation time also remains same for any input size array.

Asymptotic Analysis:

Searching => to search/find key element in a given collection/list of elements.
- there are basic two searching algorithms:
1. Linear Search:
2. Binary Search:

1. Linear Search/ Sequential Search:

```
Algorithm LinearSearch(A, n, key){//A is an array of size n
	for( index = 1 ; index <= n ; index++ ){
		if( key == A[ index ] )
			return true;
	}

	return false;
}
```

Best Case : if key is found in an array at first position
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array no. of comparisons in best case = 1 and hence in this case linear search algo takes O(1) time.

Worst Case : if either key is found in an array at last position or key does not exists.

if size of an array = 10 => no. of comparisons = 10
if size of an array = 20 => no. of comparisons = 20
.
.
if size of an array = n => no. of comparisons = n

in worst case no. of comparisons depends on an input size of an array, hence running time of linear search algo in worst case is O(n).

Rule:
- if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected/ignored.

e.g.
O(n + 4) => O(n)
O(n - 2) => O(n)
O(n / 2) => O(n)
O(3 * n) => O(n)

Home Work : to implement Linear Search => by using recursion as well as non-recursive method.

DAY-02:
2. Binary Search:

by means of calculating mid position, big size array gets divided
logically into two subarray's:
left subarray and right subarray

for left subarray => value of left remains same, right = mid-1
for right subarray => value of right remains same, left = mid+1

best case : if key is found in an array in very first iteration
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
if size of an array = 100 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array, in best case no. Of comparisons = 1,
hence running time of an algo in best case = O(1)
time complexity of binary search in best case => $\Omega(1)$.

if(left <= right) => subarray is valid
OR
if(left > right) => subarray is invalid

n = 1000

iteration-1:
search space = 1000 = n

mid=500
[0.... 499] 500 [501 .... 1000] => no. Of comparisons=1

iteration-2:
search space = 500 = n/2

[0...249] 250 [251 ... 499] => no. Of comparisons=1

iteration-3:
search space = 250 => n / 4
[0 ...124] 125 [126 ... 250] => no. Of comparisons=1

.
.
.

if size of an array = 1 => trivial case => T(1) = O(1)

if size of an array > 1 i.e. size of an array = n

T(n) = T(n) + 1

after iteration-1:
T(n) = T(n / 2) + 1 => T(n / 2^1) + 1

after iteration-2:
T(n) = T(n / 4) + 2 => T(n / 2^2) + 2

after iteration-3:
T(n) = T(n / 8) + 3 => T(n / 2^3) + 3

.
.

lets assume, k no. of iterations takes place
after k iterations:
T(n) = T(n / 2^k) + k

lets assume kth iteration is the last iteration
assume => n => 2^k

=> n => 2^k
=> log n = log 2^k ...... by taking log on both sides
=> log n = k log 2
=> log n = k ..... [log 2 ~= 1]
=> <u>k = log n</u>

T(n) = T(n / 2^k) + k
substitute values of n = 2^k & k=log n in above equation, we get
T(n) = T(2^k / 2^k) + log n
T(n) = T(1) + log n
T(n) = log n
T(n) = O(log n)

+ Sorting Algorithms:

Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.

- basic sorting algorithms:
1. selection sort
2. bubble sort
3. insertion sort

- advanced sorting algorithm:
4. quick sort
5. merge sort

1. selection sort:

for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = n(n-1) / 2 => (n^2 - n) / 2
T(n) = O((n^2 - n) / 2)
T(n) = O(n^2 - n) .... divisive can be neglected
<u>T(n) = O(n^2)</u>

Rule:
if running time of an algo is having a polynomial, then only leading term in it will be considered in its time complexity.
e.g.
$O(n^3 + n^2 + 4) \Rightarrow O(n^3)$
$O(n^2 + 5) \Rightarrow O(n^2)$

DAY-03:
searching algorithms:
linear search : algo, analysis and implementation(non-rec & rec)
binary search : algo, analysis and implementation(non-rec & rec)
sorting algorithms:
selection sort : algo, analysis and implementation

+ features of sorting algorithms:
1. inplace - if sorting algo do not takes extra space

2. adaptive - if a sorting algo works efficiently for already sorted input sequence

3. stable - if relative order of two elements having same key value remains same even after sorting.
e.g.

input array => 30 40 10 40' 20 50

after sorting:
output => 10 20 30 40 40' 50 => stable

output => 10 20 30 40' 40 50 => not stable

H.W. => to check stability of selection sort algo on paper with diifferent examples.

2. Bubble Sort:
for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = $n(n-1) / 2 \Rightarrow (n^2 - n) / 2$
$T(n) = O((n^2 - n) / 2)$
$T(n) = O(n^2 - n)$ .... divisive can be neglected
$\underline{T(n) = O(n^2)}$

```
n = size of an array / arr.lenght = 6

for itr=0 => pos=0,1,2,3,4
for itr=1 => pos=0,1,2,3
for itr=2 => pos=0,1,2

for( pos = 0 ; pos < n-1-itr ; pos++ )


input array => 10 20 30 40 50 60

flag = false

iteration-1:10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60


in first iteration,
if there is no need of swapping even single time if all pairs are
already in order => array is already sorted


best case: if array elements are already sorted
total no. of comparisons = n-1
T( n ) = O( n - 1)
T( n ) = O( n ) ... [ 1 is a substractive constant, it can be
neglected ].


3. Insertion Sort:


for( i = 1 ; i  < size ; i++ ){
     key = arr[ i ];
     j = i-1;

     //if pos is valid then only compare value of key with an ele
at that at that pos
     while( j >= 0 && key < arr[ j ] ){
          arr[ j+1 ] =  arr[ j ];//shift ele towards its right by 1
          j--;//goto prev ele
     }
     //insert key at its appropriate pos
     arr[ j+1 ] = key;
}
```

best case : if array is already sorted
input array => 10 20 30 40 50 60

iteration-1:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-2:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-3:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-4:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

- in inserttion, under best case, in every iteration only 1 comparison takes place and insertion sort requires max (n-1) no. Of iterations to sort array.
total no. Of comparisons = 1 * (n-1) = (n - 1)
T(n) = O(n - 1) => O(n) => Ω(n).

+ limitations of array data structures:
1. in an array we can combine/collect logically related only similar type of data elements => structure data structure

2. array is static i.e. size of an array cannot either grow or shrink during runtime, its size is fixed.
e.g.
int arr[100];

3. addition & deletion operations are not efficient on an array as it takes O(n) time.
- while adding ele into an array, we need to shift ele's towards right hand side by one one position, whereas while deleting ele from an array, we need to shift ele's towards left hand side by one one position => which takes O(n) time -> it depends on size of an array.

Q. Why Linked List ?
- to overcome last 2 limitations of an array data structure, linked list data structure has been designed.
- Linked List must be :
1. dynamic
2. addition & deletion operations must be performed on it efficiently i.e. expected in O(1) time.

Q. What is a Linked List ?
- Linked List is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which an addr/reference of first element in that list can be kept always into the head, and each element contains actual data as and link/reference/an addr of its next element (as well as its prev element).

- Elements in this data structure need to linked with each other explicitly by the programmer.
- element in a linked list is also called as a node.
- basically there are two types of linked list:
1. singly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node.
(no. Of links with each node = 1).

2. doubly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node as well as its prev node.
(no. of links with each node = 2).

- there are total 4 types of linked list:
1. singly linear linked list
2. singly circular linked list
3. doubly linear linked list
4. doubly circular linked list

1. singly linear linked list:

```
class Node{
     private int data;//4 bytes
     private Node next;//reference - 4 bytes
     .
     .

}
```

size of Node class object = 8 bytes.

- we can apply basic two operations on linked list:
1. addition : to add node into the linked list
- we can add node into the linked list by 3 ways:
i. add node into the linked list at last position
ii. add node into the linked list at first position
iii. add node into the linked list at specific position (inbetween position)

2. deletion : to delete node from the linked list
- we can delete node from linked list by 3 ways
i. delete node from the linked list which is at first position
ii. delete node from the linked list which is at last position
iii. delete node from the linked list which is at speecific position (in between position).

i. add node into the linked list at last position:
- we can add as many as we want no. of nodes into the slll at last position in O(n) time.

Best Case : O(1) : $\Omega(1)$- if list is empty
Worst Case : O(n) : O(n)
Average Case : O(n) : $\Theta(n)$

- to traverse a linked list => to visit each node in a linked list sequentially from first node max till last node.
(an addr of first node we will get always from head).

ii. add node into the linked list at first position:
- we can add as many as we want no. of nodes into the slll at first position in O(1) time.

Best Case : O(1) : $\Omega(1)$- if list is empty
Worst Case : O(1) : O(1)
Average Case : O(1) : $\Theta(1)$

iii. add node into the linked list at specific position:
- we can add as many as we want no. of nodes into the slll at specific position in O(n) time.

Best Case : O(1) : $\Omega(1)$- if pos = 1
Worst Case : O(n) : O(n) - if pos = last pos
Average Case : O(n) : $\Theta(n)$

- in a linked list programming - remember one rule => make before break
- always creates new links (links which are associated with newNode) first and then only break old links.

H.W. Convert program as a menu driven program.

Doubt Solving Session => 3 PM TO 4 PM.
Not compulsory, you can join only if having doubts.

Course Name : PG DAC
Module Name : Algorithms & Data Structures Using Java.

DAY-01:

Introduction:
Q. Why there is a need of data structures?
Q. What is data structures?

=> To store marks of 100 students:

```
int m1, m2, m3, m4, ......, m100;//sizeof(int): 4 bytes => 400
bytes

int marks[ 100 ];//400 bytes
```

=> Array : it is a basic/linear data structure, which is a collection/list of logically related similar type of elements gets stored into the memory at contiguos locations.

Q. Why array indexing starts from 0?
- to convert array notation into its equivalent notation is done by the compiler, (to maintained link between array elements is the job of compiler).

```
arr[ i ] ~= *(arr + i)

struct student
{
     int rollno;
     char name[32];
     float marks;
};
```

```
<data type> <var_name>;
```

- data type may be any primitive/non-primitive data type
- var_name is an identifier

e.g.

```
int n1;

struct student s1;//abstraction => abstract data type
struct student s2;
```

+ class => it is a linear/basic data structure which is a collection/list of logically related similar and disimmilar type of data elements referred as data members as well as functions which can be used to perform operations on data members referred as member funcction/methods.

```
e.g.
class student
{
	//data members
	private int rollno;
	private String name;
	private float salary;

	//methods:
	//ctor
	//mutators
	//getter functions
	//setter functions
	//facilitators
}


student s1;//ADT
```

** to learn data structures is not to learn any specific programming langauge, it is nothing but to learn an algorithms, and algorithms can be implemented by using any programming langauge (using Java).

Q. What is an algorithm?

+ traversal of an array => to visit each array element sequentially from first element max till last element.

- Algorithm to do sum of array elements: => Any User
step-1: initially take sum as 0
step-2: traverse an array and add each array element sequentially into the sum.
step-3: return final value of sum.

- Pseudocode to do sum of array elements: => Programmer User

```
Algorithm ArraySum(A, size){
	sum = 0;
	for( index = 1 ; index <= size ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```


- Program to do sum of array elements: => Machine

```
int ArraySum(int [] arr, int size){
	int sum = 0;
	for( index = 0 ; index < size ; index++ ){
		sum += arr[ index ];
	}
	return sum;
}
```

Bank Project => Bank Manager => Algorithm => Project Manager => Software Architect => Design Pseudocode => Developers => Programs => Machine

Q. What is a recursion ?
- it is a process in which function can be called within itself, such a function is referred as recursive function.

- function call for which calling function and called function are same, is referred as recursive function call.

- any thing can be defined in terms itself
example:

```
main( ){
     print(“sum = ”+recArraySum(arr, 0) );//first time function calling to the rec function
     //calling function => main()
     //called function => recArraySum()
}

int recArraySum(int [] arr, int index ){
     if( index == arr.length )
          return 0;

     return ( arr[ index ] + recArraysum(arr, index+1) );
     //calling function => recArraySum()
     //called function => recArraySum()

}
```

- to delete function activation record / stack frame from stack called as stack cleanup is done either by calling function or called function and it depends on function calling convention.

```
main(){
     print(“sum”+sum(10, 20);
     //10 & 20 => actual params
}

int sum(int n1, int n2){
     int sum;
     sum = n1 + n2;
     return sum;
}

//n1 & n2 => formal params
//sum => local var
```

+ function calling conventions:
__std__
__c__
__pascal

function calling conventions decides 2 things:
1. in which oreder params should gets passed to the function i.e. either from L -> R Or R -> L
2. stack cleanup

- when any function gets called an OS creates one entry onto the stack for that function call, called as function activation record/stack frame and it gets popped or removed from the stack when an execution of that function is completed and process to remove stack frame from stack is called as stack cleanup.

City-1:
City-2:
- there may exists multiple paths between 2 cities, in this case we need to decide an optimum/efficient path
- there are some factors/measures on which efficient/optimum path can be decides:
    time
    distance
    cost
    traffic condition
    status of path
    etc...

Space Complexity:

Space Complexity = Code Space + Data Space + Stack Space

Code Space => space required for an instructions
Data Space => space required for simple vars, constants and instance vars
Stack Space (applicable only in recursive algorithm) => space reqquired for FAR's.

- there are components of space complexity:
1. fixed component : code space + data space (space required for simple vars and constants).

2. variable component : data space (space required for instance vars) & stack space (applicable only in recursive algorithms).

Example:

```
Algorithm ArraySum(A, n){
     sum = 0;
     for( index = 1 ; index <= n ; index++ ){
          sum += A[ index ];
     }
     return sum;
}
```

S = C (Code Space) + Sp (Data Space)

Code Space =>
if size of an array = 5 => no. of instructions will be same
if size of an array = 10 => no. of instructions will be same
if size of an array = 100 => no. of instructions will be same
.
.
if size of an array = n => no. of instructions will be same

for any input size array no. of instructions in an algo will going to remains same => it will take constant amount of space for any input size array.

Sp = space required for simple vars + space required for constants + space required for instance vars

simple vars: A, sum, index => 3 units
1 unit of memory => A
1 unit of memory => sum
1 unit of memory => index

instaance var => n =>
for size of an array is 5 i.e. n = 5 => 5 units
for size of an array is 10 i.e. n = 10 => 10 units
for size of an array is 100 i.e. n = 100 => 100 units
.
.
for size of an array is n => n units => instance var

index++ => index = index + 1;

- quick sort
- merge sort
- implementation of advanced data structures algo's like - traversal methods in tree.

```
Algorithm ArraySum(A, n){
	sum = 0;
	for( index = 1 ; index <= n ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```

- for any input size array, no. of instructions will be remain same, hence compilation time also remains same for any input size array.

Asymptotic Analysis:

Searching => to search/find key element in a given collection/list of elements.
- there are basic two searching algorithms:
1. Linear Search:
2. Binary Search:

1. Linear Search/ Sequential Search:

```
Algorithm LinearSearch(A, n, key){//A is an array of size n
	for( index = 1 ; index <= n ; index++ ){
		if( key == A[ index ] )
			return true;
	}

	return false;
}
```

Best Case : if key is found in an array at first position
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array no. of comparisons in best case = 1 and hence in this case linear search algo takes O(1) time.

Worst Case : if either key is found in an array at last position or key does not exists.

if size of an array = 10 => no. of comparisons = 10
if size of an array = 20 => no. of comparisons = 20
.
.
if size of an array = n => no. of comparisons = n

in worst case no. of comparisons depends on an input size of an array, hence running time of linear search algo in worst case is O(n).

Rule:
- if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected/ignored.
e.g.
O(n + 4) => O(n)
O(n - 2) => O(n)
O(n / 2) => O(n)
O(3 * n) => O(n)

Home Work : to implement Linear Search => by using recursion as well as non-recursive method.

DAY-02:
2. Binary Search:

by means of calculating mid position, big size array gets divided
logically into two subarray's:
left subarray and right subarray

for left subarray => value of left remains same, right = mid-1
for right subarray => value of right remains same, left = mid+1

best case : if key is found in an array in very first iteration
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
if size of an array = 100 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array, in best case no. Of comparisons = 1,
hence running time of an algo in best case = $O(1)$
time complexity of binary search in best case => $\Omega(1)$.

if(left <= right) => subarray is valid
OR
if(left > right) => subarray is invalid

n = 1000

iteration-1:
search space = 1000 = n

mid=500
[0.... 499] 500 [501 .... 1000] => no. Of comparisons=1

iteration-2:
search space = 500 = n/2

[0...249] 250 [251 ... 499] => no. Of comparisons=1

iteration-3:
search space = 250 => n / 4
[0 ...124] 125 [126 ... 250] => no. Of comparisons=1

.
.
.

if size of an array = 1 => trivial case => $T(1) = O(1)$

if size of an array > 1 i.e. size of an array = n

T(n) = T(n) + 1

after iteration-1:
T(n) = T(n / 2) + 1 => T(n / 2^1) + 1

after iteration-2:
T(n) = T(n / 4) + 2 => T(n / 2^2) + 2

after iteration-3:
T(n) = T(n / 8) + 3 => T(n / 2^3) + 3

.
.

lets assume, k no. of iterations takes place
after k iterations:
T(n) = T(n / 2^k) + k

lets assume kth iteration is the last iteration
assume => n => 2^k

=> n => 2^k
=> log n = log 2^k ...... by taking log on both sides
=> log n = k log 2
=> log n = k ..... [log 2 ~= 1]
=> <u>k = log n</u>

T(n) = T(n / 2^k) + k
substitute values of n = 2^k & k=log n in above equation, we get
T(n) = T(2^k / 2^k) + log n
T(n) = T(1) + log n
T(n) = log n
T(n) = O(log n)

+ Sorting Algorithms:

Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.

- basic sorting algorithms:
1. selection sort
2. bubble sort
3. insertion sort

- advanced sorting algorithm:
4. quick sort
5. merge sort

1. selection sort:

for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = n(n-1) / 2 => (n^2 - n) / 2
T(n) = O((n^2 - n) / 2)
T(n) = O(n^2 - n) .... divisive can be neglected
<u>T(n) = O(n^2)</u>

Rule:
if running time of an algo is having a polynomial, then only leading term in it will be considered in its time complexity.
e.g.
$O(n^3 + n^2 + 4) \Rightarrow O(n^3)$
$O(n^2 + 5) \Rightarrow O(n^2)$

DAY-03:
searching algorithms:
linear search : algo, analysis and implementation(non-rec & rec)
binary search : algo, analysis and implementation(non-rec & rec)
sorting algorithms:
selection sort : algo, analysis and implementation

+ features of sorting algorithms:
1. inplace - if sorting algo do not takes extra space

2. adaptive - if a sorting algo works efficiently for already sorted input sequence

3. stable - if relative order of two elements having same key value remains same even after sorting.
e.g.

input array => 30 40 10 40' 20 50

after sorting:
output => 10 20 30 40 40' 50 => stable

output => 10 20 30 40' 40 50 => not stable

H.W. => to check stability of selection sort algo on paper with diifferent examples.

2. Bubble Sort:
for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = $n(n-1) / 2 \Rightarrow (n^2 - n) / 2$
$T(n) = O((n^2 - n) / 2)$
$T(n) = O(n^2 - n)$ .... divisive can be neglected
$\underline{T(n) = O(n^2)}$

```
n = size of an array / arr.lenght = 6

for itr=0 => pos=0,1,2,3,4
for itr=1 => pos=0,1,2,3
for itr=2 => pos=0,1,2

for( pos = 0 ; pos < n-1-itr ; pos++ )


input array => 10 20 30 40 50 60

flag = false

iteration-1:10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60


in first iteration,
if there is no need of swapping even single time if all pairs are
already in order => array is already sorted


best case: if array elements are already sorted
total no. of comparisons = n-1
T( n ) = O( n - 1)
T( n ) = O( n ) ... [ 1 is a substractive constant, it can be
neglected ].


3. Insertion Sort:


for( i = 1 ; i  < size ; i++ ){
     key = arr[ i ];
     j = i-1;

     //if pos is valid then only compare value of key with an ele
at that at that pos
     while( j >= 0 && key < arr[ j ] ){
          arr[ j+1 ] =  arr[ j ];//shift ele towards its right by 1
          j--;//goto prev ele
     }
     //insert key at its appropriate pos
     arr[ j+1 ] = key;
}
```

best case : if array is already sorted
input array => 10 20 30 40 50 60

iteration-1:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-2:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-3:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-4:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

- in inserttion, under best case, in every iteration only 1 comparison takes place and insertion sort requires max (n-1) no. Of iterations to sort array.
total no. Of comparisons = 1 * (n-1) = (n - 1)
T(n) = O(n - 1) => O(n) => Ω(n).

+ limitations of array data structures:
1. in an array we can combine/collect logically related only similar type of data elements => structure data structure

2. array is static i.e. size of an array cannot either grow or shrink during runtime, its size is fixed.
e.g.
int arr[100];

3. addition & deletion operations are not efficient on an array as it takes O(n) time.
- while adding ele into an array, we need to shift ele's towards right hand side by one one position, whereas while deleting ele from an array, we need to shift ele's towards left hand side by one one position => which takes O(n) time -> it depends on size of an array.

Q. Why Linked List ?
- to overcome last 2 limitations of an array data structure, linked list data structure has been designed.
- Linked List must be :
1. dynamic
2. addition & deletion operations must be performed on it efficiently i.e. expected in O(1) time.

Q. What is a Linked List ?
- Linked List is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which an addr/reference of first element in that list can be kept always into the head, and each element contains actual data as and link/reference/an addr of its next element (as well as its prev element).

- Elements in this data structure need to linked with each other explicitly by the programmer.
- element in a linked list is also called as a node.
- basically there are two types of linked list:
1. singly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node.
(no. Of links with each node = 1).

2. doubly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node as well as its prev node.
(no. of links with each node = 2).

- there are total 4 types of linked list:
1. singly linear linked list
2. singly circular linked list
3. doubly linear linked list
4. doubly circular linked list

1. singly linear linked list:

```
class Node{
     private int data;//4 bytes
     private Node next;//reference - 4 bytes
     .
     .

}
```

size of Node class object = 8 bytes.

- we can apply basic two operations on linked list:
1. addition : to add node into the linked list
- we can add node into the linked list by 3 ways:
i. add node into the linked list at last position
ii. add node into the linked list at first position
iii. add node into the linked list at specific position (inbetween position)

2. deletion : to delete node from the linked list
- we can delete node from linked list by 3 ways
i. delete node from the linked list which is at first position
ii. delete node from the linked list which is at last position
iii. delete node from the linked list which is at speecific position (in between position).

i. add node into the linked list at last position:
- we can add as many as we want no. of nodes into the slll at last position in O(n) time.

Best Case : O(1) : Ω(1)- if list is empty
Worst Case : O(n) : O(n)
Average Case : O(n) : Θ(n)

- to traverse a linked list => to visit each node in a linked list sequentially from first node max till last node.
(an addr of first node we will get always from head).

ii. add node into the linked list at first position:
- we can add as many as we want no. of nodes into the slll at first position in O(1) time.

Best Case : O(1) : Ω(1)- if list is empty
Worst Case : O(1) : O(1)
Average Case : O(1) : Θ(1)

iii. add node into the linked list at specific position:
- we can add as many as we want no. of nodes into the slll at specific position in O(n) time.

Best Case : O(1) : Ω(1)- if pos = 1
Worst Case : O(n) : O(n) - if pos = last pos
Average Case : O(n) : Θ(n)

- in a linked list programming - remember one rule => make before break
- always creates new links (links which are associated with newNode) first and then only break old links.

H.W. Convert program as a menu driven program.

Doubt Solving Session => 3 PM TO 4 PM.
Not compulsory, you can join only if having doubts.

DAY-04:
sorting algorithms:
- bubble sort: algorithm, analysis and implementation
- insertion sort : algorithm, analysis and implementation
- comparisons of sorting algorithms
- limitations of an array data structure
- why linked and what is linked list
- types of linked list
- operations on slll: addlast, addfirst, addatpos

- deletion of a node from linked list:
i. delete node from the list at first position
ii. delete node from the list at last position
iii. delete node from the list at specific position

i. delete node from the list at first position:
- we can delete node from slll at first position in O(1) time.
Best Case : O(1) : Ω(1)
Worst Case : O(1) : O(1)
Average Case : O(1) : Θ(1)

ii. delete node from the list at last position:
- we can delete node from slll at last position in O(n) time.
Best Case : O(1) : Ω(1) - if list contains only one node
Worst Case : O(n) : O(n)
Average Case : O(n) : Θ(n)

iii. delete node from the list at specific position:
- we can delete node from slll at specific position in O(n) time.
Best Case : O(1) : Ω(1) - if pos=1
Worst Case : O(n) : O(n) - if pos=last pos
Average Case : O(n) : Θ(n)

example:
system calls:

system call API => write( ) => _write( )sys call is used to write data into the file
write() sys call API is like a wrapper function which internally makes call to actual _write() sys call

in parameters => parameters which we pass to the function as an input
out parameters => parameters which we pass to the function as an output i.e. to get output.

=> priority queue => can be implemenetd by using linked list with search_and_delete() function.

```
Class Node{
     private <type> data;
     private int priority_value;
     Node next;

}
```

=> priority queue is a type of queue in which elements can be added into it randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.

=> delete() function in a binary search tree

+ limitations of slll:
- add node at last position & delete node at last position operations are not efficient as it takes O(n) time.
- we can traverse slll only in a forward dir
- we cannot access prev node of any node from it
- we can always start traversal from first node only
- we cannot revisit any node in a slll => to overcome this limitation scll has been designed.

SCLL:
- We can perform all the operations that we applied on SLLL as it is on SCLL, except we need to take care about/ maintained next part of last node always.

- add at position in slll & scll remains exactly same.

- deletion of a node at first position:

H.W. Implement SCLL functionalities by using menu driven program. addition & deletion

3 TO 4 => doubts solving session - not compulsory
Zoom Channel/9764658120

Course Name : PG DAC
Module Name : Algorithms & Data Structures Using Java.

DAY-01:

Introduction:
Q. Why there is a need of data structures?
Q. What is data structures?

=> To store marks of 100 students:

```
int m1, m2, m3, m4, ......, m100;//sizeof(int): 4 bytes => 400
bytes

int marks[ 100 ];//400 bytes
```

=> Array : it is a basic/linear data structure, which is a collection/list of logically related similar type of elements gets stored into the memory at contiguos locations.

Q. Why array indexing starts from 0?
- to convert array notation into its equivalent notation is done by the compiler, (to maintained link between array elements is the job of compiler).

```
arr[ i ] ~= *(arr + i)

struct student
{
     int rollno;
     char name[32];
     float marks;
};
```

<data type> <var_name>;
- data type may be any primitive/non-primitive data type
- var_name is an identifier

e.g.

```
int n1;

struct student s1;//abstraction => abstract data type
struct student s2;
```

+ class => it is a linear/basic data structure which is a collection/list of logically related similar and disimmilar type of data elements referred as data members as well as functions which can be used to perform operations on data members referred as member funcction/methods.

e.g.

```
class student
{
	//data members
	private int rollno;
	private String name;
	private float salary;

	//methods:
	//ctor
	//mutators
	//getter functions
	//setter functions
	//facilitators
}


student s1;//ADT
```

** to learn data structures is not to learn any specific programming langauge, it is nothing but to learn an algorithms, and algorithms can be implemented by using any programming langauge (using Java).

Q. What is an algorithm?

+ traversal of an array => to visit each array element sequentially from first element max till last element.

- Algorithm to do sum of array elements: => Any User
step-1: initially take sum as 0
step-2: traverse an array and add each array element sequentially into the sum.
step-3: return final value of sum.

- Pseudocode to do sum of array elements: => Programmer User

```
Algorithm ArraySum(A, size){
	sum = 0;
	for( index = 1 ; index <= size ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```


- Program to do sum of array elements: => Machine

```
int ArraySum(int [] arr, int size){
	int sum = 0;
	for( index = 0 ; index < size ; index++ ){
		sum += arr[ index ];
	}
	return sum;
}
```

Bank Project => Bank Manager => Algorithm => Project Manager => Software Architect => Design Pseudocode => Developers => Programs => Machine

Q. What is a recursion ?
- it is a process in which function can be called within itself, such a function is referred as recursive function.

- function call for which calling function and called function are same, is referred as recursive function call.

- any thing can be defined in terms itself
example:

```
main( ){
     print("sum = "+recArraySum(arr, 0) );//first time function calling to the rec function
     //calling function => main()
     //called function => recArraySum()
}

int recArraySum(int [] arr, int index ){
     if( index == arr.length )
          return 0;

     return ( arr[ index ] + recArraysum(arr, index+1) );
     //calling function => recArraySum()
     //called function => recArraySum()

}
```

- to delete function activation record / stack frame from stack called as stack cleanup is done either by calling function or called function and it depends on function calling convention.

```
main(){
     print("sum"+sum(10, 20);
     //10 & 20 => actual params
}

int sum(int n1, int n2){
     int sum;
     sum = n1 + n2;
     return sum;
}

//n1 & n2 => formal params
//sum => local var
```

+ function calling conventions:
__std__
__c__
__pascal

function calling conventions decides 2 things:
1. in which oreder params should gets passed to the function i.e. either from L -> R Or R -> L
2. stack cleanup

- when any function gets called an OS creates one entry onto the stack for that function call, called as function activation record/stack frame and it gets popped or removed from the stack when an execution of that function is completed and process to remove stack frame from stack is called as stack cleanup.

City-1:
City-2:
- there may exists multiple paths between 2 cities, in this case we need to decide an optimum/efficient path
- there are some factors/measures on which efficient/optimum path can be decides:
    - time
    - distance
    - cost
    - traffic condition
    - status of path
    - etc...

Space Complexity:

Space Complexity = Code Space + Data Space + Stack Space

Code Space => space required for an instructions
Data Space => space required for simple vars, constants and instance vars
Stack Space (applicable only in recursive algorithm) => space reqquired for FAR's.

- there are components of space complexity:
1. fixed component : code space + data space (space required for simple vars and constants).

2. variable component : data space (space required for instance vars) & stack space (applicable only in recursive algorithms).

Example:

```
Algorithm ArraySum(A, n){
	sum = 0;
	for( index = 1 ; index <= n ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```

S = C (Code Space) + Sp (Data Space)

Code Space =>
if size of an array = 5 => no. of instructions will be same
if size of an array = 10 => no. of instructions will be same
if size of an array = 100 => no. of instructions will be same
.
.
if size of an array = n => no. of instructions will be same

for any input size array no. of instructions in an algo will going to remains same => it will take constant amount of space for any input size array.

Sp = space required for simple vars + space required for constants + space required for instance vars

simple vars: A, sum, index => 3 units
1 unit of memory => A
1 unit of memory => sum
1 unit of memory => index

instaance var => n =>
for size of an array is 5 i.e. n = 5 => 5 units
for size of an array is 10 i.e. n = 10 => 10 units
for size of an array is 100 i.e. n = 100 => 100 units
.
.
for size of an array is n => n units => instance var

index++ => index = index + 1;

- quick sort
- merge sort
- implementation of advanced data structures algo's like - traversal methods in tree.

```
Algorithm ArraySum(A, n){
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
        sum += A[ index ];
    }
    return sum;
}
```

- for any input size array, no. of instructions will be remain same, hence compilation time also remains same for any input size array.

Asymptotic Analysis:

Searching => to search/find key element in a given collection/list of elements.
- there are basic two searching algorithms:
1. Linear Search:
2. Binary Search:

1. Linear Search/ Sequential Search:

```
Algorithm LinearSearch(A, n, key){//A is an array of size n
    for( index = 1 ; index <= n ; index++ ){
        if( key == A[ index ] )
            return true;
    }

    return false;
}
```

Best Case : if key is found in an array at first position
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array no. of comparisons in best case = 1 and hence in this case linear search algo takes O(1) time.

Worst Case : if either key is found in an array at last position or key does not exists.

if size of an array = 10 => no. of comparisons = 10
if size of an array = 20 => no. of comparisons = 20
.
.
if size of an array = n => no. of comparisons = n

in worst case no. of comparisons depends on an input size of an array, hence running time of linear search algo in worst case is O(n).

Rule:
- if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected/ignored.
e.g.
O(n + 4) => O(n)
O(n - 2) => O(n)
O(n / 2) => O(n)
O(3 * n) => O(n)

Home Work : to implement Linear Search => by using recursion as well as non-recursive method.

DAY-02:
2. Binary Search:

by means of calculating mid position, big size array gets divided logically into two subarray's:
left subarray and right subarray

for left subarray => value of left remains same, right = mid-1
for right subarray => value of right remains same, left = mid+1

best case : if key is found in an array in very first iteration
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
if size of an array = 100 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array, in best case no. Of comparisons = 1,
hence running time of an algo in best case = O(1)
time complexity of binary search in best case => Ω(1).

if(left <= right) => subarray is valid
OR
if(left > right) => subarray is invalid

n = 1000

iteration-1:
search space = 1000 = n

mid=500
[0.... 499] 500 [501 .... 1000] => no. Of comparisons=1

iteration-2:
search space = 500 = n/2

[0...249] 250 [251 ... 499] => no. Of comparisons=1

iteration-3:
search space = 250 => n / 4
[0 ...124] 125 [126 ... 250] => no. Of comparisons=1

.
.
.

if size of an array = 1 => trivial case => T(1) = O(1)

if size of an array > 1 i.e. size of an array = n

T(n) = T(n) + 1

after iteration-1:
T(n) = T(n / 2) + 1 => T(n / 2^1) + 1

after iteration-2:
T(n) = T(n / 4) + 2 => T(n / 2^2) + 2

after iteration-3:
T(n) = T(n / 8) + 3 => T(n / 2^3) + 3

.
.

lets assume, k no. of iterations takes place
after k iterations:
T(n) = T(n / 2^k) + k

lets assume kth iteration is the last iteration
assume => n => 2^k

=> n => 2^k
=> log n = log 2^k ...... by taking log on both sides
=> log n = k log 2
=> log n = k ..... [log 2 ~= 1]
=> <u>k = log n</u>

T(n) = T(n / 2^k) + k
substitute values of n = 2^k & k=log n in above equation, we get
T(n) = T(2^k / 2^k) + log n
T(n) = T(1) + log n
T(n) = log n
T(n) = O(log n)

+ Sorting Algorithms:

Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.

- basic sorting algorithms:
1. selection sort
2. bubble sort
3. insertion sort

- advanced sorting algorithm:
4. quick sort
5. merge sort

1. selection sort:

for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = n(n-1) / 2 => (n^2 - n) / 2
T(n) = O((n^2 - n) / 2)
T(n) = O(n^2 - n) .... divisive can be neglected
<u>T(n) = O(n^2)</u>

Rule:
if running time of an algo is having a polynomial, then only leading term in it will be considered in its time complexity.
e.g.
$O(n^3 + n^2 + 4) \Rightarrow O(n^3)$
$O(n^2 + 5) \Rightarrow O(n^2)$

DAY-03:
searching algorithms:
linear search : algo, analysis and implementation(non-rec & rec)
binary search : algo, analysis and implementation(non-rec & rec)
sorting algorithms:
selection sort : algo, analysis and implementation

+ features of sorting algorithms:
1. inplace - if sorting algo do not takes extra space

2. adaptive - if a sorting algo works efficiently for already sorted input sequence

3. stable - if relative order of two elements having same key value remains same even after sorting.
e.g.

input array => 30 40 10 40' 20 50

after sorting:
output => 10 20 30 40 40' 50 => stable

output => 10 20 30 40' 40 50 => not stable

H.W. => to check stability of selection sort algo on paper with diifferent examples.

2. Bubble Sort:
for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = n(n-1) / 2 => $(n^2 - n) / 2$
$T(n) = O((n^2 - n) / 2)$
$T(n) = O(n^2 - n)$ .... divisive can be neglected
$\underline{T(n) = O(n^2)}$

```
n = size of an array / arr.lenght = 6

for itr=0 => pos=0,1,2,3,4
for itr=1 => pos=0,1,2,3
for itr=2 => pos=0,1,2

for( pos = 0 ; pos < n-1-itr ; pos++ )


input array => 10 20 30 40 50 60

flag = false

iteration-1:10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60
```

in first iteration,
if there is no need of swapping even single time if all pairs are already in order => array is already sorted

best case: if array elements are already sorted
total no. of comparisons = n-1
T(n) = O(n - 1)
T(n) = O(n) ... [1 is a substractive constant, it can be neglected].

3. Insertion Sort:

```
for( i = 1 ; i  < size ; i++ ){
     key = arr[ i ];
     j = i-1;

     //if pos is valid then only compare value of key with an ele
at that at that pos
     while( j >= 0 && key < arr[ j ] ){
          arr[ j+1 ] =  arr[ j ];//shift ele towards its right by 1
          j--;//goto prev ele
     }
     //insert key at its appropriate pos
     arr[ j+1 ] = key;
}
```

best case : if array is already sorted
input array => 10 20 30 40 50 60

iteration-1:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-2:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-3:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-4:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

- in inserttion, under best case, in every iteration only 1 comparison takes place and insertion sort requires max (n-1) no. Of iterations to sort array.
total no. Of comparisons = 1 * (n-1) = (n - 1)
T(n) = O(n - 1) => O(n) => Ω(n).

+ limitations of array data structures:
1. in an array we can combine/collect logically related only similar type of data elements => structure data structure

2. array is static i.e. size of an array cannot either grow or shrink during runtime, its size is fixed.
e.g.
int arr[100];

3. addition & deletion operations are not efficient on an array as it takes O(n) time.
- while adding ele into an array, we need to shift ele's towards right hand side by one one position, whereas while deleting ele from an array, we need to shift ele's towards left hand side by one one position => which takes O(n) time -> it depends on size of an array.

Q. Why Linked List ?
- to overcome last 2 limitations of an array data structure, linked list data structure has been designed.
- Linked List must be :
1. dynamic
2. addition & deletion operations must be performed on it efficiently i.e. expected in O(1) time.

Q. What is a Linked List ?
- Linked List is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which an addr/reference of first element in that list can be kept always into the head, and each element contains actual data as and link/reference/an addr of its next element (as well as its prev element).

- Elements in this data structure need to linked with each other explicitly by the programmer.
- element in a linked list is also called as a node.
- basically there are two types of linked list:
1. singly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node.
(no. Of links with each node = 1).

2. doubly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node as well as its prev node.
(no. of links with each node = 2).

- there are total 4 types of linked list:
1. singly linear linked list
2. singly circular linked list
3. doubly linear linked list
4. doubly circular linked list

1. singly linear linked list:

```
class Node{
     private int data;//4 bytes
     private Node next;//reference - 4 bytes
     .
     .

}
```

size of Node class object = 8 bytes.

- we can apply basic two operations on linked list:
1. addition : to add node into the linked list
- we can add node into the linked list by 3 ways:
i. add node into the linked list at last position
ii. add node into the linked list at first position
iii. add node into the linked list at specific position (inbetween position)

2. deletion : to delete node from the linked list
- we can delete node from linked list by 3 ways
i. delete node from the linked list which is at first position
ii. delete node from the linked list which is at last position
iii. delete node from the linked list which is at speecific position (in between position).

i. add node into the linked list at last position:
- we can add as many as we want no. of nodes into the slll at last position in O(n) time.

Best Case : O(1) : $\Omega(1)$- if list is empty
Worst Case : O(n) : O(n)
Average Case : O(n) : $\Theta(n)$

- to traverse a linked list => to visit each node in a linked list sequentially from first node max till last node.
(an addr of first node we will get always from head).

ii. add node into the linked list at first position:
- we can add as many as we want no. of nodes into the slll at first position in O(1) time.

Best Case : O(1) : $\Omega(1)$- if list is empty
Worst Case : O(1) : O(1)
Average Case : O(1) : $\Theta(1)$

iii. add node into the linked list at specific position:
- we can add as many as we want no. of nodes into the slll at specific position in O(n) time.

Best Case : O(1) : $\Omega(1)$- if pos = 1
Worst Case : O(n) : O(n) - if pos = last pos
Average Case : O(n) : $\Theta(n)$

- in a linked list programming - remember one rule => make before break
- always creates new links (links which are associated with newNode) first and then only break old links.

H.W. Convert program as a menu driven program.

Doubt Solving Session => 3 PM TO 4 PM.
Not compulsory, you can join only if having doubts.

DAY-04:
sorting algorithms:
- bubble sort: algorithm, analysis and implementation
- insertion sort : algorithm, analysis and implementation
- comparisons of sorting algorithms
- limitations of an array data structure
- why linked and what is linked list
- types of linked list
- operations on slll: addlast, addfirst, addatpos

- deletion of a node from linked list:
i. delete node from the list at first position
ii. delete node from the list at last position
iii. delete node from the list at specific position

i. delete node from the list at first position:
- we can delete node from slll at first position in O(1) time.
Best Case : O(1) : Ω(1)
Worst Case : O(1) : O(1)
Average Case : O(1) : Θ(1)

ii. delete node from the list at last position:
- we can delete node from slll at last position in O(n) time.
Best Case : O(1) : Ω(1) - if list contains only one node
Worst Case : O(n) : O(n)
Average Case : O(n) : Θ(n)

iii. delete node from the list at specific position:
- we can delete node from slll at specific position in O(n) time.
Best Case : O(1) : Ω(1) - if pos=1
Worst Case : O(n) : O(n) - if pos=last pos
Average Case : O(n) : Θ(n)

example:
system calls:

system call API => write( ) => _write( )sys call is used to write data into the file
write() sys call API is like a wrapper function which internally makes call to actual _write() sys call

in parameters => parameters which we pass to the function as an input
out parameters => parameters which we pass to the function as an output i.e. to get output.

=> priority queue => can be implemenetd by using linked list with search_and_delete() function.

```
Class Node{
     private <type> data;
     private int priority_value;
     Node next;

}
```

=> priority queue is a type of queue in which elements can be added into it randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.

=> delete() function in a binary search tree

+ limitations of slll:
- add node at last position & delete node at last position operations are not efficient as it takes O(n) time.
- we can traverse slll only in a forward dir
- we cannot access prev node of any node from it
- we can always start traversal from first node only
- we cannot revisit any node in a slll => to overcome this limitation scll has been designed.

SCLL:
- We can perform all the operations that we applied on SLLL as it is on SCLL, except we need to take care about/ maintained next part of last node always.

- add at position in slll & scll remains exactly same.

- deletion of a node at first position:

H.W. Implement SCLL functionalities by using menu driven program. addition & deletion

DAY-04:

operations on sinlgy linear linked list:
- deletion: deleteFirst, deleteLast & deleteAtPosition
- display list in reverse order by recursive method
- reverse linked list
- search and delete
- limitations of slll => scll
- operations on scll => adddition & deletion

DAY-05:
+ limitations of slll:
- addLast, addFirst, deleteFirst & deleteFirst operations are not efficient as it takes O(n) time.
- we can traverse scll only in a forward dir
- we cannot access prev node of any node from it
- we can always start traversal from first node only

- to overcome limitations of singly linked list (slll & scll), doubly linked list has been designed.
- there are two types of doubly linked list
i. doubly linear linked list
ii. doubly circular linked list
- we can perform all operations on dlll which we applied on slll exactly as it is, except in dlll we need to maintained forward link as well as backward link of each node.

H.W.
Menu driven: SCLL, DLLL, DCLL
Implement deleteFirst => DCLL

Linked List => DCLL

Java Collection => Linked List

Applications of Linked List:
- Linked List data structure is used to store logically related similar type of data elements, if we dont know in advance size of the list/collection.
- Linked List is used to implement basic data structures like stack & queue.
- Linked List is used to implement kernel data structures like e.g. ready queue (list of PCB's), job queue (list of PCB's), iNode list (list of iNodes) => linked list implementation of queue
- Linked List is used to implement advanced data structures like tree, graph, hash table etc... and its algorithms can also be implemented by uing Linked List.

+ Stack: It is a basic/linear data structure, which is collection/list of logically related similar type of data elements, in which elements can be added as well as deleted from only one end referred as top end.

- in this list elements which was added last can only be deleted first, so this list works in last in first out manner, hence stack is also called as LIFO List/FILO List.

- we can apply basic 3 operations on stack data structure in O(1) time.
1. Push : to insert/add an element onto the stack from top end

2. Pop : to delete/remove an element from the stack which is at top end

3. Peek : to get the value of an element which is at top end without Push/Pop.

- Stack can be implemented by 2 ways:
1. static implementation of stack (by using an array)
2. dynamic implementation of stack (by using linked list)

- data structures can also be catagoriesed into two catagories:
1. static data structures: array, structure, class

2. dynamic data structure: linked list, tree, graph, hash table etc....

stack & queue
- Adaptive Data Structure => stack & queue data structure adapts feature of data structure by using which we implement it.
- If we implement stack by using an array it becomes static, whereas if we implement stack by using linked list it becomes dynamic.

1. static implementation of a stack (by using an array).

```
Class Stack{
     private int [] arr;
     private int top;
     .
     .
     .

}
```

1. Push   : to insert/add an element onto the stack from top end
step-1: check stack is not full (if stack is not full then only we can push element onto it).
step-2: increment the value of top by 1.
step-3: push/add element onto stack at top position

2. Pop   : to delete/remove an element from the stack which is at top end
step-1: check stack is not empty (if stack is not empty then only we can pop element from it).
step-2: decrement the value of top by 1.
(by means of decrementing the value of top by 1 we are achieving deletion of an element from the stack).

3. Peek   : to get the value of an element which is at top end without Push/Pop.
step-1: check stack is not empty (if stack is not empty then only we can peek element from it).
step-2: get the value of an element which is is at top end
[without incrementing/decrementing top].

Course Name   : PG DAC
Module Name   : Algorithms & Data Structures Using Java.

DAY-01:

Introduction:
Q. Why there is a need of data structures?
Q. What is data structures?

=> To store marks of 100 students:
```
int m1, m2, m3, m4, ......, m100;//sizeof(int): 4 bytes => 400
bytes

int marks[ 100 ];//400 bytes
```

=> Array : it is a basic/linear data structure, which is a collection/list of logically related similar type of elements gets stored into the memory at contiguos locations.

Q. Why array indexing starts from 0?
- to convert array notation into its equivalent notation is done by the compiler, (to maintained link between array elements is the job of compiler).
```
arr[ i ] ~= *(arr + i)

struct student
{
     int rollno;
     char name[32];
     float marks;
};
```

```
<data type> <var_name>;
```
- data type may be any primitive/non-primitive data type
- var_name is an identifier
e.g.
```
int n1;

struct student s1;//abstraction => abstract data type
struct student s2;
```

+ class => it is a linear/basic data structure which is a collection/list of logically related similar and disimmilar type of data elements referred as data members as well as functions which can be used to perform operations on data members referred as member funcction/methods.

e.g.

```
class student
{
	//data members
	private int rollno;
	private String name;
	private float salary;

	//methods:
	//ctor
	//mutators
	//getter functions
	//setter functions
	//facilitators
}


student s1;//ADT
```

** to learn data structures is not to learn any specific programming langauge, it is nothing but to learn an algorithms, and algorithms can be implemented by using any programming langauge (using Java).

Q. What is an algorithm?

+ traversal of an array => to visit each array element sequentially from first element max till last element.

- Algorithm to do sum of array elements: => Any User
step-1: initially take sum as 0
step-2: traverse an array and add each array element sequentially into the sum.
step-3: return final value of sum.

- Pseudocode to do sum of array elements: => Programmer User

```
Algorithm ArraySum(A, size){
	sum = 0;
	for( index = 1 ; index <= size ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```


- Program to do sum of array elements: => Machine

```
int ArraySum(int [] arr, int size){
	int sum = 0;
	for( index = 0 ; index < size ; index++ ){
		sum += arr[ index ];
	}
	return sum;
}
```

Bank Project => Bank Manager => Algorithm => Project Manager => Software Architect => Design Pseudocode => Developers => Programs => Machine

Q. What is a recursion ?
- it is a process in which function can be called within itself, such a function is referred as recursive function.

- function call for which calling function and called function are same, is referred as recursive function call.

- any thing can be defined in terms itself
example:

```
main( ){
     print(“sum = ”+recArraySum(arr, 0) );//first time function
calling to the rec function
     //calling function => main()
     //called function => recArraySum()
}

int recArraySum(int [] arr, int index ){
     if( index == arr.length )
          return 0;

     return ( arr[ index ] + recArraysum(arr, index+1) );
     //calling function => recArraySum()
     //called function => recArraySum()

}
```

- to delete function activation record / stack frame from stack called as stack cleanup is done either by calling function or called function and it depends on function calling convention.

```
main(){
     print(“sum”+sum(10, 20);
     //10 & 20 => actual params
}

int sum(int n1, int n2){
     int sum;
     sum = n1 + n2;
     return sum;
}

//n1 & n2 => formal params
//sum => local var
```

+ function calling conventions:
__std__
__c__
__pascal

function calling conventions decides 2 things:
1. in which oreder params should gets passed to the function i.e. either from L -> R Or R -> L
2. stack cleanup

- when any function gets called an OS creates one entry onto the stack for that function call, called as function activation record/stack frame and it gets popped or removed from the stack when an execution of that function is completed and process to remove stack frame from stack is called as stack cleanup.

City-1:
City-2:
- there may exists multiple paths between 2 cities, in this case we need to decide an optimum/efficient path
- there are some factors/measures on which efficient/optimum path can be decides:
  time
  distance
  cost
  traffic condition
  status of path
  etc...

Space Complexity:

Space Complexity = Code Space + Data Space + Stack Space

Code Space => space required for an instructions
Data Space => space required for simple vars, constants and instance vars
Stack Space (applicable only in recursive algorithm) => space reqquired for FAR's.

- there are components of space complexity:
1. fixed component : code space + data space (space required for simple vars and constants).

2. variable component : data space (space required for instance vars) & stack space (applicable only in recursive algorithms).

Example:

```
Algorithm ArraySum(A, n){
     sum = 0;
     for( index = 1 ; index <= n ; index++ ){
          sum += A[ index ];
     }
     return sum;
}
```

S = C (Code Space) + Sp (Data Space)

Code Space =>
if size of an array = 5 => no. of instructions will be same
if size of an array = 10 => no. of instructions will be same
if size of an array = 100 => no. of instructions will be same
.
.
if size of an array = n => no. of instructions will be same

for any input size array no. of instructions in an algo will going to remains same => it will take constant amount of space for any input size array.

```
Sp = space required for simple vars + space required for constants
+ space required for instance vars

simple vars: A, sum, index => 3 units
1 unit of memory => A
1 unit of memory => sum
1 unit of memory => index

instaance var => n =>
for size of an array is 5 i.e. n = 5 => 5 units
for size of an array is 10 i.e. n = 10 => 10 units
for size of an array is 100 i.e. n = 100 => 100 units
.
.
for size of an array is n  => n units => instance var


index++ => index = index + 1;

- quick sort
- merge sort
- implementation of advanced data structures algo's like -
traversal methods in tree.


Algorithm ArraySum(A, n){
     sum = 0;
     for( index = 1 ; index <= n ; index++ ){
          sum += A[ index ];
     }
     return sum;
}


- for any input size array, no. of instructions will be remain
same, hence compilation time also remains same for any input size
array.


Asymptotic Analysis:

Searching => to search/find key element in a given collection/list
of elements.
- there are basic two searching algorithms:
1. Linear Search:
2. Binary Search:

1. Linear Search/ Sequential Search:

Algorithm LinearSearch(A, n, key){//A is an array of size n
     for( index = 1 ; index <= n ; index++ ){
          if( key == A[ index ] )
               return true;
     }

     return false;
}
```

Best Case : if key is found in an array at first position
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array no. of comparisons in best case = 1 and hence in this case linear search algo takes O(1) time.

Worst Case : if either key is found in an array at last position or key does not exists.

if size of an array = 10 => no. of comparisons = 10
if size of an array = 20 => no. of comparisons = 20
.
.
if size of an array = n => no. of comparisons = n

in worst case no. of comparisons depends on an input size of an array, hence running time of linear search algo in worst case is O(n).

Rule:
- if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected/ignored.

e.g.
O(n + 4) => O(n)
O(n - 2) => O(n)
O(n / 2) => O(n)
O(3 * n) => O(n)

Home Work : to implement Linear Search => by using recursion as well as non-recursive method.

DAY-02:
2. Binary Search:

by means of calculating mid position, big size array gets divided logically into two subarray's:
left subarray and right subarray

for left subarray => value of left remains same, right = mid-1
for right subarray => value of right remains same, left = mid+1

best case : if key is found in an array in very first iteration
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
if size of an array = 100 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array, in best case no. Of comparisons = 1,
hence running time of an algo in best case = $O(1)$
time complexity of binary search in best case => $\Omega(1)$.

if(left <= right) => subarray is valid
OR
if(left > right) => subarray is invalid

n = 1000

iteration-1:
search space = 1000 = n

mid=500
[0.... 499] 500 [501 .... 1000] => no. Of comparisons=1

iteration-2:
search space = 500 = n/2

[0...249] 250 [251 ... 499] => no. Of comparisons=1

iteration-3:
search space = 250 => n / 4
[0 ...124] 125 [126 ... 250] => no. Of comparisons=1

.
.
.

if size of an array = 1 => trivial case => $T(1) = O(1)$

if size of an array > 1 i.e. size of an array = n

$T(n) = T(n) + 1$

after iteration-1:

$T(n) = T(n / 2) + 1 \Rightarrow T(n / 2^1) + 1$

after iteration-2:

$T(n) = T(n / 4) + 2 \Rightarrow T(n / 2^2) + 2$

after iteration-3:

$T(n) = T(n / 8) + 3 \Rightarrow T(n / 2^3) + 3$

.
.

lets assume, k no. of iterations takes place
after k iterations:
$T(n) = T(n / 2^k) + k$

lets assume kth iteration is the last iteration
assume => $n \Rightarrow 2^k$

=> $n \Rightarrow 2^k$
=> $\log n = \log 2^k$ ...... by taking log on both sides
=> $\log n = k \log 2$
=> $\log n = k$ ..... [log 2 ~= 1]
=> <u>$k = \log n$</u>

$T(n) = T(n / 2^k) + k$
substitute values of $n = 2^k$ & $k=\log n$ in above equation, we get
$T(n) = T(2^k / 2^k) + \log n$
$T(n) = T(1) + \log n$
$T(n) = \log n$
$T(n) = O(\log n)$

+ Sorting Algorithms:

Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.

- basic sorting algorithms:
1. selection sort
2. bubble sort
3. insertion sort

- advanced sorting algorithm:
4. quick sort
5. merge sort

1. selection sort:

for size of an array = n
total no. of comparisons = $(n-1) + (n-2) + (n-3) + (n-4) + \ldots + 1$
total no. of comparisons = $n(n-1) / 2 \Rightarrow (n^2 - n) / 2$
$T(n) = O((n^2 - n) / 2)$
$T(n) = O(n^2 - n)$ .... divisive can be neglected
<u>$T(n) = O(n^2)$</u>

Rule:
if running time of an algo is having a polynomial, then only leading term in it will be considered in its time complexity.
e.g.
$O(n^3 + n^2 + 4) \Rightarrow O(n^3)$
$O(n^2 + 5) \Rightarrow O(n^2)$

DAY-03:
searching algorithms:
linear search : algo, analysis and implementation(non-rec & rec)
binary search : algo, analysis and implementation(non-rec & rec)
sorting algorithms:
selection sort : algo, analysis and implementation

+ features of sorting algorithms:
1. inplace - if sorting algo do not takes extra space

2. adaptive - if a sorting algo works efficiently for already sorted input sequence

3. stable - if relative order of two elements having same key value remains same even after sorting.
e.g.

input array => 30 40 10 40' 20 50

after sorting:
output => 10 20 30 40 40' 50 => stable

output => 10 20 30 40' 40 50 => not stable

H.W. => to check stability of selection sort algo on paper with diifferent examples.

2. Bubble Sort:
for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = $n(n-1) / 2 \Rightarrow (n^2 - n) / 2$
$T(n) = O((n^2 - n) / 2)$
$T(n) = O(n^2 - n)$ .... divisive can be neglected
$\underline{T(n) = O(n^2)}$

n = size of an array / arr.lenght = 6

for itr=0 => pos=0,1,2,3,4
for itr=1 => pos=0,1,2,3
for itr=2 => pos=0,1,2

```
for( pos = 0 ; pos < n-1-itr ; pos++ )
```

input array => 10 20 30 40 50 60

flag = false

iteration-1:10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

in first iteration,
if there is no need of swapping even single time if all pairs are already in order => array is already sorted

best case: if array elements are already sorted
total no. of comparisons = n-1
T(n) = O(n - 1)
T(n) = O(n) ... [1 is a substractive constant, it can be neglected].

3. Insertion Sort:

```
for( i = 1 ; i  < size ; i++ ){
     key = arr[ i ];
     j = i-1;

     //if pos is valid then only compare value of key with an ele
at that at that pos
     while( j >= 0 && key < arr[ j ] ){
          arr[ j+1 ] =  arr[ j ];//shift ele towards its right by 1
          j--;//goto prev ele
     }
     //insert key at its appropriate pos
     arr[ j+1 ] = key;
}
```

best case : if array is already sorted
input array => 10 20 30 40 50 60

iteration-1:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-2:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-3:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-4:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

- in inserttion, under best case, in every iteration only 1 comparison takes place and insertion sort requires max (n-1) no. Of iterations to sort array.
total no. Of comparisons = 1 * (n-1) = (n - 1)
T(n) = O(n - 1) => O(n) => Ω(n).

+ limitations of array data structures:
1. in an array we can combine/collect logically related only similar type of data elements => structure data structure

2. array is static i.e. size of an array cannot either grow or shrink during runtime, its size is fixed.
e.g.
int arr[100];

3. addition & deletion operations are not efficient on an array as it takes O(n) time.
- while adding ele into an array, we need to shift ele's towards right hand side by one one position, whereas while deleting ele from an array, we need to shift ele's towards left hand side by one one position => which takes O(n) time -> it depends on size of an array.

Q. Why Linked List ?
- to overcome last 2 limitations of an array data structure, linked list data structure has been designed.
- Linked List must be :
1. dynamic
2. addition & deletion operations must be performed on it efficiently i.e. expected in O(1) time.

Q. What is a Linked List ?
- Linked List is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which an addr/reference of first element in that list can be kept always into the head, and each element contains actual data as and link/reference/an addr of its next element (as well as its prev element).

- Elements in this data structure need to linked with each other explicitly by the programmer.
- element in a linked list is also called as a node.
- basically there are two types of linked list:
1. singly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node.
(no. Of links with each node = 1).

2. doubly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node as well as its prev node.
(no. of links with each node = 2).

- there are total 4 types of linked list:
1. singly linear linked list
2. singly circular linked list
3. doubly linear linked list
4. doubly circular linked list

1. singly linear linked list:

```
class Node{
     private int data;//4 bytes
     private Node next;//reference - 4 bytes
     .
     .

}
```

size of Node class object = 8 bytes.

- we can apply basic two operations on linked list:
1. addition : to add node into the linked list
- we can add node into the linked list by 3 ways:
i. add node into the linked list at last position
ii. add node into the linked list at first position
iii. add node into the linked list at specific position (inbetween position)

2. deletion : to delete node from the linked list
- we can delete node from linked list by 3 ways
i. delete node from the linked list which is at first position
ii. delete node from the linked list which is at last position
iii. delete node from the linked list which is at speecific position (in between position).

i. add node into the linked list at last position:
- we can add as many as we want no. of nodes into the slll at last position in O(n) time.

```
Best Case       : O( 1 ) : Ω(1)- if list is empty
Worst Case      : O( n ) : O(n)
Average Case    : O( n ) : Θ(n)
```

- to traverse a linked list => to visit each node in a linked list sequentially from first node max till last node.
(an addr of first node we will get always from head).

ii. add node into the linked list at first position:
- we can add as many as we want no. of nodes into the slll at first position in O(1) time.

```
Best Case       : O( 1 ) : Ω(1)- if list is empty
Worst Case      : O( 1 ) : O(1)
Average Case    : O( 1 ) : Θ(1)
```

iii. add node into the linked list at specific position:
- we can add as many as we want no. of nodes into the slll at specific position in O(n) time.

```
Best Case       : O( 1 ) : Ω(1)- if pos = 1
Worst Case      : O( n ) : O(n) - if pos = last pos
Average Case    : O( n ) : Θ(n)
```

- in a linked list programming - remember one rule => make before break
- always creates new links (links which are associated with newNode) first and then only break old links.

H.W. Convert program as a menu driven program.

Doubt Solving Session => 3 PM TO 4 PM.
Not compulsory, you can join only if having doubts.

DAY-04:
sorting algorithms:
- bubble sort: algorithm, analysis and implementation
- insertion sort : algorithm, analysis and implementation
- comparisons of sorting algorithms
- limitations of an array data structure
- why linked and what is linked list
- types of linked list
- operations on slll: addlast, addfirst, addatpos

- deletion of a node from linked list:

i. delete node from the list at first position
ii. delete node from the list at last position
iii. delete node from the list at specific position

i. delete node from the list at first position:
- we can delete node from slll at first position in $O(1)$ time.

Best Case : $O(1)$: $\Omega(1)$
Worst Case : $O(1)$: $O(1)$
Average Case : $O(1)$: $\Theta(1)$

ii. delete node from the list at last position:
- we can delete node from slll at last position in $O(n)$ time.

Best Case : $O(1)$: $\Omega(1)$ - if list contains only one node
Worst Case : $O(n)$: $O(n)$
Average Case : $O(n)$: $\Theta(n)$

iii. delete node from the list at specific position:
- we can delete node from slll at specific position in $O(n)$ time.

Best Case : $O(1)$: $\Omega(1)$ - if pos=1
Worst Case : $O(n)$: $O(n)$ - if pos=last pos
Average Case : $O(n)$: $\Theta(n)$

example:
system calls:

system call API => write( ) => _write( )sys call is used to write data into the file
write() sys call API is like a wrapper function which internally makes call to actual _write() sys call

in parameters => parameters which we pass to the function as an input
out parameters => parameters which we pass to the function as an output i.e. to get output.

=> priority queue => can be implemenetd by using linked list with search_and_delete() function.

```
Class Node{
     private <type> data;
     private int priority_value;
     Node next;

}
```

=> priority queue is a type of queue in which elements can be added into it randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.

=> delete() function in a binary search tree

+ limitations of slll:
- add node at last position & delete node at last position operations are not efficient as it takes O(n) time.
- we can traverse slll only in a forward dir
- we cannot access prev node of any node from it
- we can always start traversal from first node only
- we cannot revisit any node in a slll => to overcome this limitation scll has been designed.

SCLL:
- We can perform all the operations that we applied on SLLL as it is on SCLL, except we need to take care about/ maintained next part of last node always.

- add at position in slll & scll remains exactly same.

- deletion of a node at first position:

H.W. Implement SCLL functionalities by using menu driven program. addition & deletion

DAY-04:

operations on sinlgy linear linked list:
- deletion: deleteFirst, deleteLast & deleteAtPosition
- display list in reverse order by recursive method
- reverse linked list
- search and delete
- limitations of slll => scll
- operations on scll => adddition & deletion

DAY-05:
+ limitations of slll:
- addLast, addFirst, deleteFirst & deleteFirst operations are not efficient as it takes O(n) time.
- we can traverse scll only in a forward dir
- we cannot access prev node of any node from it
- we can always start traversal from first node only

- to overcome limitations of singly linked list (slll & scll), doubly linked list has been designed.
- there are two types of doubly linked list
i. doubly linear linked list
ii. doubly circular linked list
- we can perform all operations on dlll which we applied on slll exactly as it is, except in dlll we need to maintained forward link as well as backward link of each node.

H.W.
Menu driven: SCLL, DLLL, DCLL
Implement deleteFirst => DCLL

Linked List => DCLL

Java Collection => Linked List

Applications of Linked List:
- Linked List data structure is used to store logically related similar type of data elements, if we dont know in advance size of the list/collection.
- Linked List is used to implement basic data structures like stack & queue.
- Linked List is used to implement kernel data structures like e.g. ready queue (list of PCB's), job queue (list of PCB's), iNode list (list of iNodes) => linked list implementation of queue
- Linked List is used to implement advanced data structures like tree, graph, hash table etc... and its algorithms can also be implemented by uing Linked List.

+ Stack: It is a basic/linear data structure, which is collection/list of logically related similar type of data elements, in which elements can be added as well as deleted from only one end referred as top end.

- in this list elements which was added last can only be deleted first, so this list works in last in first out manner, hence stack is also called as LIFO List/FILO List.

- we can apply basic 3 operations on stack data structure in O(1) time.
1. Push : to insert/add an element onto the stack from top end

2. Pop : to delete/remove an element from the stack which is at top end

3. Peek : to get the value of an element which is at top end without Push/Pop.

- Stack can be implemented by 2 ways:
1. static implementation of stack (by using an array)
2. dynamic implementation of stack (by using linked list)

- data structures can also be catagoriesed into two catagories:
1. static data structures: array, structure, class

2. dynamic data structure: linked list, tree, graph, hash table etc....

stack & queue
- Adaptive Data Structure => stack & queue data structure adapts feature of data structure by using which we implement it.
- If we implement stack by using an array it becomes static, whereas if we implement stack by using linked list it becomes dynamic.

1. static implementation of a stack (by using an array).

```
Class Stack{
     private int [] arr;
     private int top;
     .
     .
     .

}
```

1. Push   : to insert/add an element onto the stack from top end
step-1: check stack is not full (if stack is not full then only we can push element onto it).
step-2: increment the value of top by 1.
step-3: push/add element onto stack at top position

2. Pop   : to delete/remove an element from the stack which is at top end
step-1: check stack is not empty (if stack is not empty then only we can pop element from it).
step-2: decrement the value of top by 1.
(by means of decrementing the value of top by 1 we are achieving deletion of an element from the stack).

3. Peek   : to get the value of an element which is at top end without Push/Pop.
step-1: check stack is not empty (if stack is not empty then only we can peek element from it).
step-2: get the value of an element which is is at top end
[without incrementing/decrementing top].

DAY-06:
- Linked List
+ Stack :
  - concept and definition
- implementation of the stack by using an array => static stack

+ implementation of the stack by using linked list (dcll) => dynamic stack:

- stack works in last in first out manner

push : addLast( )
pop : deleteLast( )

peek

dynamic stack:
head => 22 11

OR
push : addFirst( )
pop : deleteFirst( )

H.W. Implement dynamic stack by using linked list.

+ Applications of Stack:
- undo & redo functionalities in an OS as well as in applications like editor, ms excel etc.... stack is used.
- stack is used to implement advanced data structure algorithms like dfs traversal in tree & grap, inorder, preorder and postorder traversal techniques in tree.
- stack is used by an OS to control flow of an execution of programs by maintaining FAR's onto the stack.
- to reverse a string
- stack is used in compilers for parenthesis balancing
- stack is also used in an algorithm to convert given infix expression into its prefix or postfix expression and in an algo to evaluate expressions.

+ expression conversion and evalution algorithms and their implementation.

Q. What is an expression ?
- An expression is a combination of an operands and an operators.

- there are 3 types of expressions:
1. infix expression : a+b
2. prefix expression : +ab
3. postfix expression : ab+

1. algorithm to convert given infix expression into its equivalent postfix:

Lab Work: Implement Postfix evaluation algorithm for operands having multiple digits.

+ Queue: It is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which elements can be added into it from one end referred as rear end, whereas elements can be deleted from the queue from another end referred as front end.

- in this list element which was inserted first can be deleted first, so this list works in first in first out manner, hence it is also called as FIFO list/LILO list.

- we can perform basic 2 operations on queue in O(1) time:
1. enqueue : to insert an element into the queue from rear end
2. dequeue : to delete an element from the queue which is at front end.

- there are diff types of queue
1. linear queue (fifo)

2. circular queue (fifo)

3. priority queue - it is a type of queue in which elements can be added into it from rear end randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.
- priority queue can be implemented by using linked list (searchAndDelete( ) function).

4. double ended queue (deque) - it is a type of queue in which elements can be added as well as deleted from both the ends.
- there are two types of deque:
i. input restricted deque : it is a type of deque in which elements can be inserted into it only from one end, whereas elements can be deleted from both the ends.

ii. output restricted deque : it is a type of deque in which elements can be inserted from both the ends, whereas elements can be deleted only from one end.

- we can perform basic 4 operations on deque in O(1) time:
```
1. push front  : addFirst( )
2. push back   : addLast( )
3. pop front   : deleteFirst( )
4. pop back    : deleteLast( )
```

- deque can be implemented by using dcll

HW: Priority queue & Dequeue

+ Implementation of Linear Queue & Circular Queue:
- FIFO Queue can be implented by two ways:
1. static implementation of fifo queue by using array => static queue
2. dynamic implementation of fifo queue by using linked list => dynamic queue.

1. static implementation of fifo queue by using array => static queue:

```
class Queue{
     private int [] arr;
     private int rear;
     private int front;

     Queue( ){
          arr = new int[ 5 ];
          rear = -1;
          front = -1;

     }
}
```

- we can perform basic 2 operations on queue in O(1) time:
1. enqueue : to insert an element into the queue from rear end
step-1: check queue is not full (if queue is not full then only we can insert an element into it from rear end).
step-2: increment the value of rear by 1.
step-3: insert an element into the queue at rear end
step-4: if(front == -1)
          front = 0;

2. dequeue : to delete an element from the queue which is at front end.
step-1: check queue is not empty
(if queue is not empty then only we can delete element from it from front end).
step-2: increment the value of from by 1.
[by means of incrementing the value of front by 1 we are achieving deletion of an element from the queue].

Circular Queue

```
rear = 4, front = 0
rear = 0, front = 1
rear = 1, front = 2
rear = 2, front = 3
rear = 3, front = 4

if front is at next pos of rear => cir queue is full

if( front == (rear + 1) % SIZE )
     cir queue is full

for => rear = 0, front = 1 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 1 == (0+1)%5
=> 1 == 1%5
=> 1 == 1 => LHS == RHS => cir q is full

for => rear = 1, front = 2 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 2 == (1+1)%5
=> 2 == 2%5
=> 2 == 2 => LHS == RHS => cir q is full

for => rear = 2, front = 3 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 3 == (2+1)%5
=> 3 == 3%5
=> 3 == 3 => LHS == RHS => cir q is full

for => rear = 3, front = 4 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 4 == (3+1)%5
=> 4 == 4%5
=> 4 == 4 => LHS == RHS => cir q is full

for => rear = 4, front = 0 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 0 == (4+1)%5
=> 0 == 5%5
=> 0 == 0 => LHS == RHS => cir q is full
```

```
- increment the value of front by 1.
=> front++;
=> front = front + 1;

to increment value of front by 1
front = (front+1)%SIZE
for front=0
=> front = (front+1)%SIZE
=> front = (0+1)%5
=> front = 1%5
=> front = 1

for front=1
=> front = (front+1)%SIZE
=> front = (1+1)%5
=> front = 2%5
=> front = 2

for front=2
=> front = (front+1)%SIZE
=> front = (2+1)%5
=> front = 3%5
=> front = 3

for front=3
=> front = (front+1)%SIZE
=> front = (3+1)%5
=> front = 4%5
=> front = 4

for front=4
=> front = (front+1)%SIZE
=> front = (4+1)%5
=> front = 5%5
=> front = 0
```

Course Name : PG DAC
Module Name : Algorithms & Data Structures Using Java.

DAY-01:

Introduction:

Q. Why there is a need of data structures?
Q. What is data structures?

=> To store marks of 100 students:

```
int m1, m2, m3, m4, ......, m100;//sizeof(int): 4 bytes => 400
bytes

int marks[ 100 ];//400 bytes
```

=> Array : it is a basic/linear data structure, which is a collection/list of logically related similar type of elements gets stored into the memory at contiguos locations.

Q. Why array indexing starts from 0?
- to convert array notation into its equivalent notation is done by the compiler, (to maintained link between array elements is the job of compiler).

```
arr[ i ] ~= *(arr + i)

struct student
{
     int rollno;
     char name[32];
     float marks;
};


<data type> <var_name>;
```

- data type may be any primitive/non-primitive data type
- var_name is an identifier

e.g.

```
int n1;

struct student s1;//abstraction => abstract data type
struct student s2;
```

+ class => it is a linear/basic data structure which is a collection/list of logically related similar and disimmilar type of data elements referred as data members as well as functions which can be used to perform operations on data members referred as member funcction/methods.

```
e.g.
class student
{
	//data members
	private int rollno;
	private String name;
	private float salary;

	//methods:
	//ctor
	//mutators
	//getter functions
	//setter functions
	//facilitators
}


student s1;//ADT
```

** to learn data structures is not to learn any specific programming langauge, it is nothing but to learn an algorithms, and algorithms can be implemented by using any programming langauge (using Java).

Q. What is an algorithm?

+ traversal of an array => to visit each array element sequentially from first element max till last element.

- Algorithm to do sum of array elements: => Any User
step-1: initially take sum as 0
step-2: traverse an array and add each array element sequentially into the sum.
step-3: return final value of sum.

- Pseudocode to do sum of array elements: => Programmer User

```
Algorithm ArraySum(A, size){
	sum = 0;
	for( index = 1 ; index <= size ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```


- Program to do sum of array elements: => Machine

```
int ArraySum(int [] arr, int size){
	int sum = 0;
	for( index = 0 ; index < size ; index++ ){
		sum += arr[ index ];
	}
	return sum;
}
```

Bank Project => Bank Manager => Algorithm => Project Manager => Software Architect => Design Pseudocode => Developers => Programs => Machine

Q. What is a recursion ?
- it is a process in which function can be called within itself, such a function is referred as recursive function.

- function call for which calling function and called function are same, is referred as recursive function call.

- any thing can be defined in terms itself
example:

```
main( ){
     print("sum = "+recArraySum(arr, 0) );//first time function
calling to the rec function
     //calling function => main()
     //called function => recArraySum()
}

int recArraySum(int [] arr, int index ){
     if( index == arr.length )
          return 0;

     return ( arr[ index ] + recArraysum(arr, index+1) );
     //calling function => recArraySum()
     //called function => recArraySum()

}
```

- to delete function activation record / stack frame from stack called as stack cleanup is done either by calling function or called function and it depends on function calling convention.

```
main(){
     print("sum"+sum(10, 20);
     //10 & 20 => actual params
}

int sum(int n1, int n2){
     int sum;
     sum = n1 + n2;
     return sum;
}

//n1 & n2 => formal params
//sum => local var
```

+ function calling conventions:
__std__
__c__
__pascal

function calling conventions decides 2 things:
1. in which oreder params should gets passed to the function i.e. either from L -> R Or R -> L
2. stack cleanup

- when any function gets called an OS creates one entry onto the stack for that function call, called as function activation record/stack frame and it gets popped or removed from the stack when an execution of that function is completed and process to remove stack frame from stack is called as stack cleanup.

City-1:
City-2:
- there may exists multiple paths between 2 cities, in this case we need to decide an optimum/efficient path
- there are some factors/measures on which efficient/optimum path can be decides:
    time
    distance
    cost
    traffic condition
    status of path
    etc...

Space Complexity:

Space Complexity = Code Space + Data Space + Stack Space

Code Space => space required for an instructions
Data Space => space required for simple vars, constants and instance vars
Stack Space (applicable only in recursive algorithm) => space reqquired for FAR's.

- there are components of space complexity:
1. fixed component : code space + data space (space required for simple vars and constants).

2. variable component : data space (space required for instance vars) & stack space (applicable only in recursive algorithms).

Example:

```
Algorithm ArraySum(A, n){
	sum = 0;
	for( index = 1 ; index <= n ; index++ ){
		sum += A[ index ];
	}
	return sum;
}
```

S = C (Code Space) + Sp (Data Space)

Code Space =>
if size of an array = 5 => no. of instructions will be same
if size of an array = 10 => no. of instructions will be same
if size of an array = 100 => no. of instructions will be same
.
.
if size of an array = n => no. of instructions will be same

for any input size array no. of instructions in an algo will going to remains same => it will take constant amount of space for any input size array.

```
Sp = space required for simple vars + space required for constants
+ space required for instance vars

simple vars: A, sum, index => 3 units
1 unit of memory => A
1 unit of memory => sum
1 unit of memory => index

instaance var => n =>
for size of an array is 5 i.e. n = 5 => 5 units
for size of an array is 10 i.e. n = 10 => 10 units
for size of an array is 100 i.e. n = 100 => 100 units
.
.
for size of an array is n  => n units => instance var


index++ => index = index + 1;

- quick sort
- merge sort
- implementation of advanced data structures algo's like -
traversal methods in tree.


Algorithm ArraySum(A, n){
     sum = 0;
     for( index = 1 ; index <= n ; index++ ){
          sum += A[ index ];
     }
     return sum;
}


- for any input size array, no. of instructions will be remain
same, hence compilation time also remains same for any input size
array.


Asymptotic Analysis:

Searching => to search/find key element in a given collection/list
of elements.
- there are basic two searching algorithms:
1. Linear Search:
2. Binary Search:

1. Linear Search/ Sequential Search:

Algorithm LinearSearch(A, n, key){//A is an array of size n
     for( index = 1 ; index <= n ; index++ ){
          if( key == A[ index ] )
               return true;
     }

     return false;
}
```

Best Case : if key is found in an array at first position
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array no. of comparisons in best case = 1 and hence in this case linear search algo takes O(1) time.

Worst Case : if either key is found in an array at last position or key does not exists.

if size of an array = 10 => no. of comparisons = 10
if size of an array = 20 => no. of comparisons = 20
.
.
if size of an array = n => no. of comparisons = n

in worst case no. of comparisons depends on an input size of an array, hence running time of linear search algo in worst case is O(n).

Rule:
- if running time of an algo is having any additive / substractive / divisive / multiplicative constant then it can be neglected/ignored.

e.g.
O(n + 4) => O(n)
O(n - 2) => O(n)
O(n / 2) => O(n)
O(3 * n) => O(n)

Home Work : to implement Linear Search => by using recursion as well as non-recursive method.

DAY-02:
2. Binary Search:

by means of calculating mid position, big size array gets divided
logically into two subarray's:
left subarray and right subarray

for left subarray => value of left remains same, right = mid-1
for right subarray => value of right remains same, left = mid+1

best case : if key is found in an array in very first iteration
if size of an array = 10 => no. of comparisons = 1
if size of an array = 20 => no. of comparisons = 1
if size of an array = 100 => no. of comparisons = 1
.
.
if size of an array = n => no. of comparisons = 1

for any input size array, in best case no. Of comparisons = 1,
hence running time of an algo in best case = O(1)
time complexity of binary search in best case => $\Omega(1)$.

if(left <= right) => subarray is valid
OR
if(left > right) => subarray is invalid

n = 1000

iteration-1:
search space = 1000 = n

mid=500
[0.... 499] 500 [501 .... 1000] => no. Of comparisons=1

iteration-2:
search space = 500 = n/2

[0...249] 250 [251 ... 499] => no. Of comparisons=1

iteration-3:
search space = 250 => n / 4
[0 ...124] 125 [126 ... 250] => no. Of comparisons=1

.
.
.

if size of an array = 1 => trivial case => T(1) = O(1)

if size of an array > 1 i.e. size of an array = n

$T(n) = T(n) + 1$

after iteration-1:
$T(n) = T(n/2) + 1 \Rightarrow T(n/2^1) + 1$

after iteration-2:
$T(n) = T(n/4) + 2 \Rightarrow T(n/2^2) + 2$

after iteration-3:
$T(n) = T(n/8) + 3 \Rightarrow T(n/2^3) + 3$

.
.

lets assume, k no. of iterations takes place
after k iterations:
$T(n) = T(n/2^k) + k$

lets assume kth iteration is the last iteration
assume => $n \Rightarrow 2^k$

=> $n \Rightarrow 2^k$
=> $\log n = \log 2^k$ ...... by taking log on both sides
=> $\log n = k \log 2$
=> $\log n = k$ ..... [$\log 2 \sim= 1$]
=> $\underline{k = \log n}$

$T(n) = T(n/2^k) + k$
substitute values of $n = 2^k$ & $k=\log n$ in above equation, we get
$T(n) = T(2^k / 2^k) + \log n$
$T(n) = T(1) + \log n$
$T(n) = \log n$
$T(n) = O(\log n)$

+ Sorting Algorithms:

Sorting => to arrange data elements in a collection/list of elements either in an ascending order or in a descending order.

- basic sorting algorithms:
1. selection sort
2. bubble sort
3. insertion sort

- advanced sorting algorithm:
4. quick sort
5. merge sort

1. selection sort:

for size of an array = n
total no. of comparisons = $(n-1) + (n-2) + (n-3) + (n-4) + \dots + 1$
total no. of comparisons = $n(n-1) / 2 \Rightarrow (n^2 - n) / 2$
$T(n) = O((n^2 - n) / 2)$
$T(n) = O(n^2 - n)$ .... divisive can be neglected
$\underline{T(n) = O(n^2)}$

Rule:
if running time of an algo is having a polynomial, then only leading term in it will be considered in its time complexity.
e.g.
$O(\ n^3 + n^2 + 4\ ) \Rightarrow O(\ n^3\ )$
$O(\ n^2 + 5\ ) \Rightarrow O(\ n^2\ )$

DAY-03:
searching algorithms:
linear search : algo, analysis and implementation(non-rec & rec)
binary search : algo, analysis and implementation(non-rec & rec)
sorting algorithms:
selection sort : algo, analysis and implementation

+ features of sorting algorithms:
1. inplace - if sorting algo do not takes extra space

2. adaptive - if a sorting algo works efficiently for already sorted input sequence

3. stable - if relative order of two elements having same key value remains same even after sorting.
e.g.

input array => 30 40 10 40' 20 50

after sorting:
output => 10 20 30 40 40' 50 => stable

output => 10 20 30 40' 40 50 => not stable

H.W. => to check stability of selection sort algo on paper with diifferent examples.

2. Bubble Sort:
for size of an array = n
total no. of comparisons = (n-1) + (n-2) + (n-3) + (n-4) +... + 1
total no. of comparisons = n(n-1) / 2 => $(n^2 - n) / 2$
$T(\ n\ ) = O(\ (n^2 - n) / 2\ )$
$T(\ n\ ) = O(\ n^2 - n\ )$ .... divisive can be neglected
<u>$T(\ n\ ) = O(\ n^2\ )$</u>

n = size of an array / arr.lenght = 6

for itr=0 => pos=0,1,2,3,4
for itr=1 => pos=0,1,2,3
for itr=2 => pos=0,1,2

```
for( pos = 0 ; pos < n-1-itr ; pos++ )
```

input array => 10 20 30 40 50 60

flag = false

iteration-1:10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

10 20 30 40 50 60

in first iteration,
if there is no need of swapping even single time if all pairs are already in order => array is already sorted

best case: if array elements are already sorted
total no. of comparisons = n-1
T(n) = O(n - 1)
T(n) = O(n) ... [1 is a substractive constant, it can be neglected].

3. Insertion Sort:

```
for( i = 1 ; i  < size ; i++ ){
     key = arr[ i ];
     j = i-1;

     //if pos is valid then only compare value of key with an ele
at that at that pos
     while( j >= 0 && key < arr[ j ] ){
          arr[ j+1 ] =  arr[ j ];//shift ele towards its right by 1
          j--;//goto prev ele
     }
     //insert key at its appropriate pos
     arr[ j+1 ] = key;
}
```

best case : if array is already sorted
input array => 10 20 30 40 50 60

iteration-1:
10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-2:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-3:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

iteration-4:

10 20 30 40 50 60

10 20 30 40 50 60

no. of comparisons = 1

- in inserttion, under best case, in every iteration only 1 comparison takes place and insertion sort requires max (n-1) no. Of iterations to sort array.
total no. Of comparisons = 1 * (n-1) = (n - 1)
T(n) = O(n - 1) => O(n) => Ω(n).

+ limitations of array data structures:
1. in an array we can combine/collect logically related only similar type of data elements => structure data structure

2. array is static i.e. size of an array cannot either grow or shrink during runtime, its size is fixed.
e.g.
int arr[100];

3. addition & deletion operations are not efficient on an array as it takes O(n) time.
- while adding ele into an array, we need to shift ele's towards right hand side by one one position, whereas while deleting ele from an array, we need to shift ele's towards left hand side by one one position => which takes O(n) time -> it depends on size of an array.

Q. Why Linked List ?
- to overcome last 2 limitations of an array data structure, linked list data structure has been designed.
- Linked List must be :
1. dynamic
2. addition & deletion operations must be performed on it efficiently i.e. expected in O(1) time.

Q. What is a Linked List ?
- Linked List is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which an addr/reference of first element in that list can be kept always into the head, and each element contains actual data as and link/reference/an addr of its next element (as well as its prev element).

- Elements in this data structure need to linked with each other explicitly by the programmer.
- element in a linked list is also called as a node.
- basically there are two types of linked list:
1. singly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node.
(no. Of links with each node = 1).

2. doubly linked list : it is a type of linked list in which each node contains link/reference/an addr of its next node as well as its prev node.
(no. of links with each node = 2).

- there are total 4 types of linked list:
1. singly linear linked list
2. singly circular linked list
3. doubly linear linked list
4. doubly circular linked list

1. singly linear linked list:

```
class Node{
     private int data;//4 bytes
     private Node next;//reference - 4 bytes
     .
     .

}
```

size of Node class object = 8 bytes.

- we can apply basic two operations on linked list:
1. addition : to add node into the linked list
- we can add node into the linked list by 3 ways:
i. add node into the linked list at last position
ii. add node into the linked list at first position
iii. add node into the linked list at specific position (inbetween position)

2. deletion : to delete node from the linked list
- we can delete node from linked list by 3 ways
i. delete node from the linked list which is at first position
ii. delete node from the linked list which is at last position
iii. delete node from the linked list which is at speecific position (in between position).

i. add node into the linked list at last position:
- we can add as many as we want no. of nodes into the slll at last position in O(n) time.

Best Case : O(1) : $\Omega(1)$- if list is empty
Worst Case : O(n) : O(n)
Average Case : O(n) : $\Theta(n)$

- to traverse a linked list => to visit each node in a linked list sequentially from first node max till last node.
(an addr of first node we will get always from head).

ii. add node into the linked list at first position:
- we can add as many as we want no. of nodes into the slll at first position in O(1) time.

Best Case : O(1) : $\Omega(1)$- if list is empty
Worst Case : O(1) : O(1)
Average Case : O(1) : $\Theta(1)$

iii. add node into the linked list at specific position:
- we can add as many as we want no. of nodes into the slll at specific position in O(n) time.

Best Case : O(1) : $\Omega(1)$- if pos = 1
Worst Case : O(n) : O(n) - if pos = last pos
Average Case : O(n) : $\Theta(n)$

- in a linked list programming - remember one rule => make before break
- always creates new links (links which are associated with newNode) first and then only break old links.

H.W. Convert program as a menu driven program.

Doubt Solving Session => 3 PM TO 4 PM.
Not compulsory, you can join only if having doubts.

DAY-04:
sorting algorithms:
- bubble sort: algorithm, analysis and implementation
- insertion sort : algorithm, analysis and implementation
- comparisons of sorting algorithms
- limitations of an array data structure
- why linked and what is linked list
- types of linked list
- operations on slll: addlast, addfirst, addatpos

- deletion of a node from linked list:
i. delete node from the list at first position
ii. delete node from the list at last position
iii. delete node from the list at specific position

i. delete node from the list at first position:
- we can delete node from slll at first position in O(1) time.
Best Case : O(1) : Ω(1)
Worst Case : O(1) : O(1)
Average Case : O(1) : Θ(1)

ii. delete node from the list at last position:
- we can delete node from slll at last position in O(n) time.
Best Case : O(1) : Ω(1) - if list contains only one node
Worst Case : O(n) : O(n)
Average Case : O(n) : Θ(n)

iii. delete node from the list at specific position:
- we can delete node from slll at specific position in O(n) time.
Best Case : O(1) : Ω(1) - if pos=1
Worst Case : O(n) : O(n) - if pos=last pos
Average Case : O(n) : Θ(n)

example:
system calls:

system call API => write( ) => _write( )sys call is used to write data into the file
write() sys call API is like a wrapper function which internally makes call to actual _write() sys call

in parameters => parameters which we pass to the function as an input
out parameters => parameters which we pass to the function as an output i.e. to get output.

=> priority queue => can be implemenetd by using linked list with search_and_delete() function.

```
Class Node{
     private <type> data;
     private int priority_value;
     Node next;

}
```

=> priority queue is a type of queue in which elements can be added into it randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.

=> delete() function in a binary search tree

+ limitations of slll:
- add node at last position & delete node at last position operations are not efficient as it takes O(n) time.
- we can traverse slll only in a forward dir
- we cannot access prev node of any node from it
- we can always start traversal from first node only
- we cannot revisit any node in a slll => to overcome this limitation scll has been designed.

SCLL:
- We can perform all the operations that we applied on SLLL as it is on SCLL, except we need to take care about/ maintained next part of last node always.

- add at position in slll & scll remains exactly same.

- deletion of a node at first position:

H.W. Implement SCLL functionalities by using menu driven program. addition & deletion

DAY-04:

operations on sinlgy linear linked list:
- deletion: deleteFirst, deleteLast & deleteAtPosition
- display list in reverse order by recursive method
- reverse linked list
- search and delete
- limitations of slll => scll
- operations on scll => adddition & deletion

DAY-05:
+ limitations of slll:
- addLast, addFirst, deleteFirst & deleteFirst operations are not efficient as it takes O(n) time.
- we can traverse scll only in a forward dir
- we cannot access prev node of any node from it
- we can always start traversal from first node only

- to overcome limitations of singly linked list (slll & scll), doubly linked list has been designed.
- there are two types of doubly linked list
i. doubly linear linked list
ii. doubly circular linked list
- we can perform all operations on dlll which we applied on slll exactly as it is, except in dlll we need to maintained forward link as well as backward link of each node.

H.W.
Menu driven: SCLL, DLLL, DCLL
Implement deleteFirst => DCLL

Linked List => DCLL

Java Collection => Linked List

Applications of Linked List:
- Linked List data structure is used to store logically related similar type of data elements, if we dont know in advance size of the list/collection.
- Linked List is used to implement basic data structures like stack & queue.
- Linked List is used to implement kernel data structures like e.g. ready queue (list of PCB's), job queue (list of PCB's), iNode list (list of iNodes) => linked list implementation of queue
- Linked List is used to implement advanced data structures like tree, graph, hash table etc... and its algorithms can also be implemented by uing Linked List.

+ Stack: It is a basic/linear data structure, which is collection/list of logically related similar type of data elements, in which elements can be added as well as deleted from only one end referred as top end.

- in this list elements which was added last can only be deleted first, so this list works in last in first out manner, hence stack is also called as LIFO List/FILO List.

- we can apply basic 3 operations on stack data structure in O(1) time.
1. Push   : to insert/add an element onto the stack from top end

2. Pop   : to delete/remove an element from the stack which is at top end

3. Peek   : to get the value of an element which is at top end without Push/Pop.

- Stack can be implemented by 2 ways:
1. static implementation of stack (by using an array)
2. dynamic implementation of stack (by using linked list)

- data structures can also be catagoriesed into two catagories:
1. static data structures: array, structure, class

2. dynamic data structure: linked list, tree, graph, hash table etc....

stack & queue
- Adaptive Data Structure => stack & queue data structure adapts feature of data structure by using which we implement it.
- If we implement stack by using an array it becomes static, whereas if we implement stack by using linked list it becomes dynamic.

1. static implementation of a stack (by using an array).

```
Class Stack{
     private int [] arr;
     private int top;
     .
     .
     .

}
```

1. Push   : to insert/add an element onto the stack from top end
step-1: check stack is not full (if stack is not full then only we can push element onto it).
step-2: increment the value of top by 1.
step-3: push/add element onto stack at top position

2. Pop   : to delete/remove an element from the stack which is at top end
step-1: check stack is not empty (if stack is not empty then only we can pop element from it).
step-2: decrement the value of top by 1.
(by means of decrementing the value of top by 1 we are achieving deletion of an element from the stack).

3. Peek   : to get the value of an element which is at top end without Push/Pop.
step-1: check stack is not empty (if stack is not empty then only we can peek element from it).
step-2: get the value of an element which is is at top end
[without incrementing/decrementing top].

DAY-06:
- Linked List
+ Stack :
 - concept and definition
- implementation of the stack by using an array => static stack

+ implementation of the stack by using linked list (dcll) => dynamic stack:

- stack works in last in first out manner
push : addLast( )
pop : deleteLast( )

peek

dynamic stack:
head => 22 11

OR
push : addFirst( )
pop : deleteFirst( )

H.W. Implement dynamic stack by using linked list.

+ Applications of Stack:
- undo & redo functionalities in an OS as well as in applications like editor, ms excel etc.... stack is used.
- stack is used to implement advanced data structure algorithms like dfs traversal in tree & grap, inorder, preorder and postorder traversal techniques in tree.
- stack is used by an OS to control flow of an execution of programs by maintaining FAR's onto the stack.
- to reverse a string
- stack is used in compilers for parenthesis balancing
- stack is also used in an algorithm to convert given infix expression into its prefix or postfix expression and in an algo to evaluate expressions.

+ expression conversion and evalution algorithms and their implementation.

Q. What is an expression ?
- An expression is a combination of an operands and an operators.

- there are 3 types of expressions:
1. infix expression   : a+b
2. prefix expression   : +ab
3. postfix expression   : ab+

1. algorithm to convert given infix expression into its equivalent postfix:

Lab Work: Implement Postfix evaluation algorithm for operands having multiple digits.

+ Queue: It is a basic/linear data structure, which is a collection/list of logically related similar type of data elements in which elements can be added into it from one end referred as rear end, whereas elements can be deleted from the queue from another end referred as front end.

- in this list element which was inserted first can be deleted first, so this list works in first in first out manner, hence it is also called as FIFO list/LILO list.

- we can perform basic 2 operations on queue in O(1) time:
1. enqueue : to insert an element into the queue from rear end
2. dequeue : to delete an element from the queue which is at front end.

- there are diff types of queue
1. linear queue (fifo)

2. circular queue (fifo)

3. priority queue - it is a type of queue in which elements can be added into it from rear end randomly (i.e. without checking priority), whereas element which is having highest priority can only be deleted first.
- priority queue can be implemented by using linked list (searchAndDelete( ) function).

4. double ended queue (deque) - it is a type of queue in which elements can be added as well as deleted from both the ends.
- there are two types of deque:
i. input restricted deque : it is a type of deque in which elements can be inserted into it only from one end, whereas elements can be deleted from both the ends.

ii. output restricted deque : it is a type of deque in which elements can be inserted from both the ends, whereas elements can be deleted only from one end.

- we can perform basic 4 operations on deque in O(1) time:
1. push front : addFirst( )
2. push back   : addLast( )
3. pop front   : deleteFirst( )
4. pop back   : deleteLast( )

- deque can be implemented by using dcll

HW: Priority queue & Dequeue

+ Implementation of Linear Queue & Circular Queue:
- FIFO Queue can be implented by two ways:
1. static implementation of fifo queue by using array => static queue
2. dynamic implementation of fifo queue by using linked list => dynamic queue.

1. static implementation of fifo queue by using array => static queue:

```
class Queue{
     private int [] arr;
     private int rear;
     private int front;

     Queue( ){
          arr = new int[ 5 ];
          rear = -1;
          front = -1;

     }
}
```

- we can perform basic 2 operations on queue in O(1) time:
1. enqueue : to insert an element into the queue from rear end
step-1: check queue is not full (if queue is not full then only we can insert an element into it from rear end).
step-2: increment the value of rear by 1.
step-3: insert an element into the queue at rear end
step-4: if(front == -1)
          front = 0;

2. dequeue : to delete an element from the queue which is at front end.
step-1: check queue is not empty
(if queue is not empty then only we can delete element from it from front end).
step-2: increment the value of from by 1.
[by means of incrementing the value of front by 1 we are achieving deletion of an element from the queue].

Circular Queue

```
rear = 4, front = 0
rear = 0, front = 1
rear = 1, front = 2
rear = 2, front = 3
rear = 3, front = 4

if front is at next pos of rear => cir queue is full

if( front == (rear + 1) % SIZE )
     cir queue is full

for => rear = 0, front = 1 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 1 == (0+1)%5
=> 1 == 1%5
=> 1 == 1 => LHS == RHS => cir q is full

for => rear = 1, front = 2 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 2 == (1+1)%5
=> 2 == 2%5
=> 2 == 2 => LHS == RHS => cir q is full

for => rear = 2, front = 3 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 3 == (2+1)%5
=> 3 == 3%5
=> 3 == 3 => LHS == RHS => cir q is full

for => rear = 3, front = 4 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 4 == (3+1)%5
=> 4 == 4%5
=> 4 == 4 => LHS == RHS => cir q is full

for => rear = 4, front = 0 => front is at next pos of rear => cir
q is full
=> front == (rear + 1) % SIZE
=> 0 == (4+1)%5
=> 0 == 5%5
=> 0 == 0 => LHS == RHS => cir q is full
```

- increment the value of front by 1.
=> front++;
=> front = front + 1;

to increment value of front by 1
front = (front+1)%SIZE
for front=0
=> front = (front+1)%SIZE
=> front = (0+1)%5
=> front = 1%5
=> front = 1

for front=1
=> front = (front+1)%SIZE
=> front = (1+1)%5
=> front = 2%5
=> front = 2

for front=2
=> front = (front+1)%SIZE
=> front = (2+1)%5
=> front = 3%5
=> front = 3

for front=3
=> front = (front+1)%SIZE
=> front = (3+1)%5
=> front = 4%5
=> front = 4

for front=4
=> front = (front+1)%SIZE
=> front = (4+1)%5
=> front = 5%5
=> front = 0

DAY-06:
- applications of stack data structure
- stack application algorithms: expression conversion and evaluation algorithms and their implementation.
- queue: concept, definition and types of queue
implementation of linear queue and circular queue by using an array.

Queue can be implemented by using an array as well as by using linked list.

- dynamic queue by using linked list(dcll):

enqueue => addLast( )
dequeue => deleteFirst( )

head => 40 50
OR
enqueue => addFirst( )
dequeue => deleteLast( )
Lab Work => Implement dynamic queue by using linked list.

+ **Array:**
arr[0] = 10 &arr[0] = 1000
arr[1] = 20 &arr[1] = 1004
arr[2] = 30 &arr[2] = 1008
.
.
.

10, 20, 30, 40., ... => int type data elements gets stored into the memory at contiguos locations => contiguos as well as a linear
- array ele's can be accessed linearly i.e. sequentially and as each array ele has index so we can access by random access method as well.

Linked List:
head [1000] => | 10 : 2000 | => | 20 : 3000 | => | 30 : 4000 | => | 40 : null |

linked list elements do not gets stored into the memory in a contiguos manner, but as we link them in a linear manner hence it can be accessed linearly/sequencially

1^{st} node is at 1000
2^{nd} node is at 1008
3^{rd} node is at 1016

DAY-07:
- Advanced Data Structures:

+ tree: it is a non-linear/advanced data structure, which is a collection/list of logically related similar type of finite no. of data elements in which there is a first specially designated element referred as root element and remaining all elements are connected to the root element in a hierarchical follows parent-child relatioship.

- root node
- parent node/father
- child node/son
- grand parent node / grand father
- grand child node / grand son
- ancestors: all the nodes which are in the path from root node to that node.
- root node is an anscestor of all the nodes
- descendents: all the nodes which can be accessible from that node.
As all the nodes can be accessed from root node, hence all the nodes are descedents of root node.
- degree of a node = no. of child node/s having that node.
- degree of a tree = max degree of any node in a given tree
- leaf node is also called as terminal node/external node
- non-leaf node is also called as non-terminal node/internal node

- by concept tree is a dynamic data structure
- as tree is a dynamic data structure, so any node can have any no. of child nodes and it can grow upto any level, but due to this feature, operations on it like specially addition, deletion & searching becomes inefficient, so restrictions can be applied on tree, hence there are different types of tree:

- binary tree : it is a type of tree in which each node can have max 2 no. of child nodes.
OR each node can have degree either 0 OR 1 OR 2
- it is a tree in which max degree of any node is 2.

empty set / null set = 0 no. of elements
singleton set = contains only 1 element
set = more than 1 elements

- further restrictions can be applied on binary tree to achieve operations like addition, deletion and searching efficiently i.e. expected in O(log n) time, hence binary search tree has been designed.

- binary search tree => it is a binary tree in which left child is always smaller than its parent and right child is always greater or equal to its patent.

- we can add as many as we want number of nodes into the BST in O(log n) time, as to find an appropriate position for a newly created node takes O(log n) time.

- to traverse a tree/BST => to visit each node in a tree at once.
- in a tree, we can start traversal always from root node only.
- there are basic two tree traversal methods:
1. bfs (breadth first search) traversal
2. dfs (depth first search) traversal

1. bfs (breadth first search) traversal:
- bfs traversal is also called as level-wise traversal
- in this traversal method, traversal starts from root node and nodes gets visited level wise from left to right.

2. dfs (depth first search) traversal:
- in this traversal method, traversal starts from root node.
- under dfs, further bst can be traversed by 3 ways:

i. Preorder (VLR)
ii. Inorder (LVR)
iii. Postorder(LRV)

L => Left Subtree
R => Right Subtree
V => visit

i. Preorder traversal (VLR):
- in this traversal method, traversal starts from root node.
- very first we can visit cur node, then we can visit its left subtree, and then only we can visit right subtree of cur node i.e. we can visit right subtree of any node only after visiting its whole left subtree or left subtree is empty.
- in this type of traversal, root node always gets visited at first, this property remains same recursively for each subtree.

ii. Inorder traversal (LVR):
- in this traversal method, traversal starts from root node.
- very first we can visit left subtree of cur node then we can visit that node and then only we can visit its right subtree.
- we can visit any node only after visiting its whole left subtree or its left subtree is empty and then only we can visit right subtree of that node.

- in this type of traversal, all the nodes always gets visited in ascending order.

iii. Postorder traversal (LRV):
- in this traversal method, traversal starts from root node.
- first we need to visit whole left subtree of cur node, then we can visit its right subtree and then only we can visit cur node i.e. we can visit any node either only after visiting its left subtree as well as a right subtree or left subtree and right right subtree are empty.

- in this type of traversal, root node always gets visited at last, this property remains same recursively for each subtree.

ALGO_DS_DAY-09:

Quick Sort:

Partitioning:
step-1: select left most element as a pivot element
step-2: shift ele's which are smaller than pivot towards left (as possible), and shift ele's which are greater than pivot towards right (as possible).
In first pass, pivot ele gets settled/fixed at its appropriate position and big size array gets divided logically into two partitions => left partition & right partition
left partition => left to j-1
right partition => j+1 to right

for left partition => value of left remains same, right = j-1
for right partition => value of right remains same, left = j+1

- we can apply partitioning on left partition as well as right partition recursively till the size of partition is >1.

partition is valid till left < right
if(left >= right) => partition is invalid

```
i=left;
j=right;
pivot = arr[ left ];

while( i < j ){

     while( i <= right && arr[ i ] <= pivot )
          i++;


     if( i < j )//if i & j have not crossed swap them
     {
          swap(arr[ i ], arr[ j ]);
     }
}

//swap pivot ele with j^th element
swap(arr[ j ], arr[ left ]);
```

=> for partitioning quick sort algo takes log n, and no. Passes is depends on size of an array, as size of an increases no. of passes i.e. no. Of times paritionining required also increases and in avg as well as best case time required for this algo is n*log n
T(n) = O(n log n)

=> worst case occurs in quick sort if either array is already sorted or array ele's are exactly in a reverse order, which rarely occurs.

[10 20 30 40 50 60]

pass-1: partitioning => [10 20 30 40 50 60] => pivot = 10

[LP] 10 [20 30 40 50 60]

pass-2: partitioning => [20 30 40 50 60] => pivot = 20
[LP] 20 [30 40 50 60]

pass-3: partitioning => [30 40 50 60] => pivot = 30
[LP] 30 [40 50 60]

pass-4: partitioning => [40 50 60] => pivot = 40
[LP] 40 [50 60]

pass-5: partitioning => [50 60] => pivot = 50
[LP] 50 [60]

- in quick sort, under worst case partitioning takes O(n), array is not gets divided equally into two partitions, and no. of passes = n-1
total no. of comparisons = n * (n-1)
T(n) = O(n * (n-1)
T(n) = O(n^2 - n)
T(n) = O(n^2)

Merge Sort:
works on twi princeipals:
1. as the size of an array is min sorting is efficient.
2. it is always efficient to merge two already sorted arrays into a single array in a sorted manner.

algorithm:
- step-1: divide big size array logically into smallest size (i.e. having size 1) subarray's.
- we can divide array into subarray's logically by means of calculating mid pos
mid = (left+right)/2,
by means of calculating mid pos, big size array gets divided logically into two subarray's
left subarray & right subarray
left subarray => left to mid
right subarray => mid+1 to right

for left subarray => value of left remains same, right = mid
for right subarray => value of right remains same, left = mid+1

for divisioning it takes O(log n) time, as size of an array increases time required for dividing also gets increases
T(n) = O(n * log n)

- merge two already sorted lists into a single list in a sorted manner

list1 => head => null
list2 => head => null

list3 => head => 5 -> 10 -> 15 -> 20 -> 25 -> 30 -> 35 -> 40 -> 45 -> 50 -> 60 -> null

+ "Graph" : it is a non-linear/advanced data structure, which is a collection of logically related similar and disimmilar type of data elements, which contains
- finite set of elements called as vertices, also referred as a "nodes", and finite set of ordered/unordered pairs of vertices called as an "edges", also referred as an "arcs", which may contains weight/cost/value and it may be -ve.

example:
google map

to store info about cities and info about paths between cities graph data structure is used.
- to store info about city => vertices => City class object
- to store info about paths between cities => edges => Path class objects

```
class City{
     String cityCode;
     String cityName;
     String stateName;
     String countryName;
     .......
     .......
     .......
}

class Path{
     String pathCode;
     String pathName;
     String srcCityName;
     String destName;
     float distInKm;
     ......
     ......

}
```

- graph is a collection of 100's of city class objects and 1000's of path class objects between those cities.
- we can apply algo's like to find shortest distance of all vertices from given source vertex like dijskra's algo, floyd-warshall algo or to find MST like prim's & kruskal's.

- if we want to store information in a network we can go for graph data structure.

ALGO_DS_DAY-09:

Quick Sort:

Partitioning:
step-1: select left most element as a pivot element
step-2: shift ele's which are smaller than pivot towards left (as possible), and shift ele's which are greater than pivot towards right (as possible).
In first pass, pivot ele gets settled/fixed at its appropriate position and big size array gets divided logically into two partitions => left partition & right partition
left partition => left to j-1
right partition => j+1 to right

for left partition => value of left remains same, right = j-1
for right partition => value of right remains same, left = j+1

- we can apply partitioning on left partition as well as right partition recursively till the size of partition is >1.

partition is valid till left < right
if(left >= right) => partition is invalid

```
i=left;
j=right;
pivot = arr[ left ];

while( i < j ){

     while( i <= right && arr[ i ] <= pivot )
          i++;


     if( i < j )//if i & j have not crossed swap them
     {
          swap(arr[ i ], arr[ j ]);
     }
}

//swap pivot ele with jth element
swap(arr[ j ], arr[ left ]);
```

=> for partitioning quick sort algo takes log n, and no. Passes is depends on size of an array, as size of an increases no. of passes i.e. no. Of times paritionining required also increases and in avg as well as best case time required for this algo is n*log n
T(n) = O(n log n)

=> worst case occurs in quick sort if either array is already sorted or array ele's are exactly in a reverse order, which rarely occurs.

[10 20 30 40 50 60]

pass-1: partitioning => [10 20 30 40 50 60] => pivot = 10

```
[ LP ] 10 [ 20 30 40 50 60 ]

pass-2: partitioning => [ 20 30 40 50 60 ] => pivot = 20
[ LP ] 20 [ 30 40 50 60 ]

pass-3: partitioning => [ 30 40 50 60 ] => pivot = 30
[ LP ] 30 [ 40 50 60 ]

pass-4: partitioning => [ 40 50 60 ] => pivot = 40
[ LP ] 40 [ 50 60 ]


pass-5: partitioning => [ 50 60 ] => pivot = 50
[ LP ] 50 [ 60 ]

- in quick sort, under worst case partitioning takes O(n), array
is not gets divided equally into two partitions, and no. of passes
= n-1
total no. of comparisons = n * (n-1)
T( n ) = O( n * (n-1 )
T( n ) = O( n^2 - n )
T( n ) = O( n^2 )


Merge Sort:
works on twi princeipals:
1. as the size of an array is min sorting is efficient.
2. it is always efficient to merge two already sorted arrays into
a single array in a sorted manner.

algorithm:
- step-1: divide big size array logically into smallest size (i.e.
having size 1 ) subarray's.
- we can divide array into subarray's logically by means of
calculating mid pos
mid = (left+right)/2,
by means of calculating mid pos, big size array gets divided
logically into two subarray's
left subarray & right subarray
left subarray => left to mid
right subarray => mid+1 to right

for left subarray => value of left remains same, right = mid
for right subarray => value of right remains same, left = mid+1


for divisioning it takes O(log n) time, as size of an array
increases time required for dividing also gets increases
T( n ) = O( n * log n )


- merge two already sorted lists into a single list in a sorted
manner

list1 => head =>  null
list2 => head => null
```

list3 => head => 5 -> 10 -> 15 -> 20 -> 25 -> 30 -> 35 -> 40 -> 45 -> 50 -> 60 -> null

+ "Graph" : it is a non-linear/advanced data structure, which is a collection of logically related similar and disimmilar type of data elements, which contains
- finite set of elements called as vertices, also referred as a "nodes", and finite set of ordered/unordered pairs of vertices called as an "edges", also referred as an "arcs", which may contains weight/cost/value and it may be -ve.

example:
google map

to store info about cities and info about paths between cities graph data structure is used.
- to store info about city => vertices => City class object
- to store info about paths between cities => edges => Path class objects

```
class City{
	String cityCode;
	String cityName;
	String stateName;
	String countryName;
	.......
	.......
	.......
}

class Path{
	String pathCode;
	String pathName;
	String srcCityName;
	String destName;
	float distInKm;
	......
	......

}
```

- graph is a collection of 100's of city class objects and 1000's of path class objects between those cities.
- we can apply algo's like to find shortest distance of all vertices from given source vertex like dijskra's algo, floyd-warshall algo or to find MST like prim's & kruskal's.

- if we want to store information in a network we can go for graph data structure.

- quick sort
- merge sort
- graph: concept & definition
- graph terminologies
- graph representation methods:
1. adjancency matrix (2-d array)
2. adjacenncy list (array of of linked lists - arraylist).

ALGO_DS_DAY-10:

LabWork => to implement weighted graph by using adjList

divide-and-conquer : binary search, quick sort, merge sort

greedy approach : dijsktra's, prim's, kruskal's etc...
- if any problem has solution by using greedy approach it is most efficient way

ALGO_DS_DAY-09:

Quick Sort:

Partitioning:
step-1: select left most element as a pivot element
step-2: shift ele's which are smaller than pivot towards left (as possible), and shift ele's which are greater than pivot towards right (as possible).
In first pass, pivot ele gets settled/fixed at its appropriate position and big size array gets divided logically into two partitions => left partition & right partition
left partition => left to j-1
right partition => j+1 to right

for left partition => value of left remains same, right = j-1
for right partition => value of right remains same, left = j+1

- we can apply partitioning on left partition as well as right partition recursively till the size of partition is >1.

partition is valid till left < right
if(left >= right) => partition is invalid

```
i=left;
j=right;
pivot = arr[ left ];

while( i < j ){

     while( i <= right && arr[ i ] <= pivot )
          i++;


     if( i < j )//if i & j have not crossed swap them
     {
          swap(arr[ i ], arr[ j ]);
     }
}

//swap pivot ele with jth element
swap(arr[ j ], arr[ left ]);
```

=> for partitioning quick sort algo takes log n, and no. Passes is depends on size of an array, as size of an increases no. of passes i.e. no. Of times paritionining required also increases and in avg as well as best case time required for this algo is n*log n
T(n) = O(n log n)

=> worst case occurs in quick sort if either array is already sorted or array ele's are exactly in a reverse order, which rarely occurs.

[10 20 30 40 50 60]

pass-1: partitioning => [10 20 30 40 50 60] => pivot = 10

```
[ LP ] 10 [ 20 30 40 50 60 ]

pass-2: partitioning => [ 20 30 40 50 60 ] => pivot = 20
[ LP ] 20 [ 30 40 50 60 ]

pass-3: partitioning => [ 30 40 50 60 ] => pivot = 30
[ LP ] 30 [ 40 50 60 ]

pass-4: partitioning => [ 40 50 60 ] => pivot = 40
[ LP ] 40 [ 50 60 ]


pass-5: partitioning => [ 50 60 ] => pivot = 50
[ LP ] 50 [ 60 ]

- in quick sort, under worst case partitioning takes O(n), array
is not gets divided equally into two partitions, and no. of passes
= n-1
total no. of comparisons = n * (n-1)
T( n ) = O( n * (n-1 )
T( n ) = O( n^2 - n )
T( n ) = O( n^2 )


Merge Sort:
works on twi princeipals:
1. as the size of an array is min sorting is efficient.
2. it is always efficient to merge two already sorted arrays into
a single array in a sorted manner.

algorithm:
- step-1: divide big size array logically into smallest size (i.e.
having size 1 ) subarray's.
- we can divide array into subarray's logically by means of
calculating mid pos
mid = (left+right)/2,
by means of calculating mid pos, big size array gets divided
logically into two subarray's
left subarray & right subarray
left subarray => left to mid
right subarray => mid+1 to right

for left subarray => value of left remains same, right = mid
for right subarray => value of right remains same, left = mid+1


for divisioning it takes O(log n) time, as size of an array
increases time required for dividing also gets increases
T( n ) = O( n * log n )


- merge two already sorted lists into a single list in a sorted
manner

list1 => head =>  null
list2 => head => null
```

list3 => head => 5 -> 10 -> 15 -> 20 -> 25 -> 30 -> 35 -> 40 -> 45 -> 50 -> 60 -> null

+ "Graph" : it is a non-linear/advanced data structure, which is a collection of logically related similar and disimmilar type of data elements, which contains
- finite set of elements called as vertices, also referred as a "nodes", and finite set of ordered/unordered pairs of vertices called as an "edges", also referred as an "arcs", which may contains weight/cost/value and it may be -ve.

example:
google map

to store info about cities and info about paths between cities graph data structure is used.
- to store info about city => vertices => City class object
- to store info about paths between cities => edges => Path class objects

```
class City{
     String cityCode;
     String cityName;
     String stateName;
     String countryName;
     .......
     .......
     .......
}

class Path{
     String pathCode;
     String pathName;
     String srcCityName;
     String destName;
     float distInKm;
     ......
     ......

}
```

- graph is a collection of 100's of city class objects and 1000's of path class objects between those cities.
- we can apply algo's like to find shortest distance of all vertices from given source vertex like dijskra's algo, floyd-warshall algo or to find MST like prim's & kruskal's.

- if we want to store information in a network we can go for graph data structure.

- quick sort
- merge sort
- graph: concept & definition
- graph terminologies
- graph representation methods:
1. adjancency matrix (2-d array)
2. adjacenncy list (array of of linked lists - arraylist).

ALGO_DS_DAY-10:

LabWork => to implement weighted graph by using adjList

divide-and-conquer : binary search, quick sort, merge sort

greedy approach : dijsktra's, prim's, kruskal's etc...
- if any problem has solution by using greedy approach it is most efficient way

ALGO_DS_DAY-11:

graph algorithms:
- adjacency list graph
- dfs traversal
- bfs traversal
- dfs spanning tree
- bfs spanning tree
- to check graph is connected or not
- dijsktra's algo => to find shortest distance of all vertices from given source vertex
- prim's algo => to find MST of given graph.
- problem solving technique/approach => greedy approach.

- Kruskal's algo: to find MST of a given graph

- Lab Work : Optional => implenent disjktra's, prims & kruskal algo's by using adjList graph.

PCB => Printed Circuit Board - MatLab

Hash Table:

we want to store 1000 customer records, mobile no. Of cust as a key and personal info.

Array => YES => size of 1000
Linked List=> searching is not efficient
Balanced BST => addition ,deletion and searching => O(log n)

PG-DAC SEPT-2021

ALGORITHMS & DATA STRUCTURES

SACHIN G. PAWAR

SUNBEAM INSTITUTE OF INFORMATION & TECHNOLOGIES

PUNE & KARAD

Data Structures: Introduction

Name of the Module : Algorithms & Data Structures Using Java.

Prerequisites: Knowledge of programming in C/C++/Java with object oriented concepts.

Weightage : 100 Marks (Theory Exam : 40% + Lab Exam : 40% + Mini Project : 20%).

 # Importance of the Module:

1. CDAC - Syllabus
2. To improve programming skills
3. Campus Placements
4. Applications in Industry work

Data Structures: Introduction

Q. Why there is a need of data structure?

- There is a need of data structure to achieve 3 things in programming:

1. **efficiency**
2. **abstraction**
3. **reusability**

Q. What is a Data Structure?

Data Structure is **a way to store data elements into the memory** (i.e. into the main memory) in **an organized manner** so that operations like **addition, deletion, traversal, searching, sorting** etc... can be performed on it efficiently.

Data Structures: Introduction

Two types of **Data Structures** are there:

1. Linear / Basic data structures : data elements gets stored / arranged into the memory in a **linear manner** (e.g. sequentially) and hence can be accessed linearly / sequentially.

- **Array**
- **Structure & Union**
- **Class**
- **Linked List**
- **Stack**
- **Queue**

2. Non-Linear / Advanced data structures : data elements gets stored / arranged into the memory in a **non-linear manner** (e.g. hierarchical manner) and hence can be accessed non-linearly.

- **Tree (Hierarchical manner)**
- **Graph**
- **Hash Table(Associative manner)**
- **Binary Heap**

Data Structures: Introduction

+ Array: It is a **basic/linear data structure** which is **a collection/list of logically related similar type of data elements** gets stored/arranged into the memory at **contiguos locations**.

+ Structure: It is a **basic/linear data structure** which is **a collection/list of logically related similar and disimmilar type of data elements** gets stored/arranged into the memory **collectively i.e. as a single entity/record.**

sizeof of the structure = sum of size of all its members.

+ Union: Union is same like structure, except, memory allocation i.e. size of union is the size of max size member defined in it and that memory gets shared among all its members for effective memory utilization (can be used in a special case only).

Data Structures: Introduction

Q. What is a Program?

- A Program is **a finite set of instructions written in any programming language** (either in a high level programming language like C, C++, Java, Python or in a low level programming language like assembly, machine etc...) given to the machine to do specific task.

Q. What is an Algorithm?

- An algorithm is **a finite set of instructions written in any human understandable language (like english)**, if followed, acomplishesh a given task.

- **Pseudocode :** It is a **special form of an algorithm**, which is a finite set of instructions written in any human understandable language (like english) **with some programming constraints**, if followed, acomplishesh a given task.

- An algorithm is a template whereas a program is an implementation of an algorithm.

Data Structures: Introduction

 # Algorithm : to do sum of all array elements

Step-1: initially take value of sum is 0.

Step-2: traverse an array sequentially from first element till last element and add each array element into the sum.

Step-3: return final sum.

 # Pseudocode : to do sum of all array elements

```
Algorithm ArraySum(A, n){//whereas A is an array of size n
    sum=0;//initially sum is 0
    for( index = 1 ; index <= size ; index++ ) {
    sum += A[ index ];//add each array element into the sum
    }
    return sum;
}
```

Data Structures: Introduction

- There are two types of Algorithms OR there are two approaches to write an algorithm:

1. iterative (non-recursive) approach :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
            sum += A[ index ];
    }
        return sum;
}
```

```
for( exp1 ; exp2 ; exp3 ){
    statement/s
}
```

exp1 => initialization
exp2 => termination condition
exp3 => modification

Data Structures: Introduction

2. recursive approach:
While writing recursive algo: we need to take care about 3 things
1. initialization: at the time first time calling to recursive function
2. base condition/termination condition : at the begining of recursive function
3. modification: while recursive function call

Example:

```
Algorithm RecArraySum( A, n, index )
{
   if( index == n )//base condition
   return 0;

   return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

Data Structures: Introduction

Recursion : it is a process in which we can give call to the function within itself.

function for which recursion is used => recursive function

- there are two types of recursive functions:

1. tail recursive function : recursive function in which recursive function call is the last executable statement.

```
void fun( int n )
{
    if( n == 0 )
        return;

    printf("%4d", n);
    fun(n--);//rec function call
}
```

2. non-tail recursive function : recursive function in which recursive function call is not the last executable statement

```
void fun( int n )
{
   if( n == 0 )
      return;

   fun(n--);//rec function call
   printf("%4d", n);
}
```

Data Structures: Introduction

- An Algorithm is a solution of a given problem.
- Algorithm = Solution
- One problem may has many solutions.

For example: Problem => **Sorting :** to arrange data elements in a collection/list of elements either in an ascending order or in descending order.

A1 : Selection Sort
A2 : Bubble Sort
A3 : Insertion Sort
A4 : Quick Sort
A5 : Merge Sort
etc...

- When one problem has many solutions/algorithms, in that case we need to select an efficient solution/algo, and to decide efficiency of an algo's we need to do their analysis.

Data Structures: Introduction

- **Analysis of an algorithm** is a work of determining how much **time** i.e. computer time and **space** i.e. computer memory it needs to run to completion.

- There are two **measures** of an **analysis of an algorithms:**

1. Time Complexity of an algorithm is the amount of **time i.e. computer time** it needs to run to completion.

2. Space Complexity of an algorithm is the amount of **space i.e. computer memory** it needs to run to completion.

Data Structures: Introduction

 # Space Complexity of an algorithm is the amount of space i.e. computer memory it needs to run to completion.

Space complexity = code space + data space + stack space (applicable only for recursive algo)
code space = space required for an instructions
data space = space required for **simple variables, constants & instance variables.**
stack space = space required for **function activation records.**

- Space complexity has **two components:**
1. fixed component: data space (space required for simple vars & constants) and code space.
2. variable component : instance characteristics (i.e. space required for instance vars) and stack space (which is applicable only in recursive algorithms).

Data Structures: Introduction

 # Calculation of space complexity of non-recursive algo:

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
        sum += A[ index ];
    }
    return sum;
}
```

Sp = data space + instance charactristics

simple vars => formal param: A & local vars: sum, index

constants : 0 & 1

instance variable = n, input size of an array = **n units**

data space = 3 units (for simple vars => A, sum & index) + 2 units (for constants => 0 & 1)

=> data space = **5 units**

Sp = (n + 5) units.

Data Structures: Introduction

S = C (code space) + Sp
S = C + (n+5)
S >= (n + 5) ... (as C is constant, it can be neglected)
S >= O(n) => O(n)
Space required for an algo = O(n) => whereas n = input size array.

 # <u>Calculation of space complexity of recursive algorithm:</u>

```
Algorithm RecArraySum( A, n, index ){
    if( index == n )//base condition
    return 0;
    return ( A[ index ] + RecArraySum(A, n, index+1) );
}
```

space complexity = code space + data space + stack space (applicable only in recursive algo)
code space = space required for instructions
data space = space required for variables, constants & instance characteristics
stack space = space required for FAR's.

Data Structures: Introduction

- When any function gets called one entry gets created onto the stack for that function call, referred as **function activation record / stack frame,** it contains **formal params, local vars, return addr, old frame pointer etc...**

In our example of recursive algorithm:

3 units (for A, index & n) + 2 units (for constants 0 & 1) = total 5 **units** of memory is required per function call.

- for size of an array = **n,** algo gets called **(n+1) no. of times.**

Hence, total space required = **5 * (n+1)**

S = 5n + 5

S >= 5n.

S >= 5n

S ~= 5n => O(n), wheras n = size of an array

Data Structures: Introduction

 # Time Complexity:

time complexity = compilation time + execution time

Time complexity has two components :

1. fixed component : compilation time

2. variable component : execution time => it depends on instance char of an algorithm.

Example :

```
Algorithm ArraySum( A, n){//whereas A is an array of size n
    sum = 0;
    for( index = 1 ; index <= n ; index++ ){
    sum += A[ index ];
    }

    return sum;
}
```

Data Structures: Introduction

- for size of an array = 5 => instruction/s inside for loop will execute 5 no. of times
- for size of an array = 10 => instruction/s inside for loop will execute 10 no. of times
- for size of an array = 20 => instruction/s inside for loop will execute 20 no. of times
- **for size of an array = n => instruction/s inside for loop will execute n no. of times**

 # Scenario-1

Machine-1 : Pentium-4 : Algorithm : input size = 10
Machine-2 : Core i5 : Algorithm : input size = 10

 # Scenario-2

Machine-1 : Core i5 : Algorithm : input size = 10 : system fully laoded with other processes
Machine-2 : Core i5 : Algorithm : input size = 10 : system not fully laoded with other processes.

- it is observed that, execution time is not only depends on instance chars, it also depends on some external factors like hardware on which algorithm is running as well as other conditions, and hence it is not a good practice to decide efficiency of an algo i.e. calculation of time complexity on the basis of an execution time and compilation time, and hence to do analysis of an algorithms **asymptotic analysis** is preferred.

Data Structures: Introduction

 # Asymptotic Analysis : It is a **mathematical way** to calculate time complexity and space complexity of an algorithm **without implementing it in any programming language.**

- In this type of analysis, analysis can be done on the basis of **basic operation** in that algorithm.

e.g. in searching & sorting algorithms **comparison** is the basic operation and hence analysis can be done on the basis of no. of comparisons, in addition of matrices algorithm **addition** is the basic operation and hence on the basis of addition operation analysis can be done.

"Best case time complexity": if an algo takes **min** amount of time to run to completion then it is referred as best case time complexity.

"Worst case time complexity": if an algo takes **max** amount of time to to run to completion then it is referred as worst case time complexity.

"Average case time complexity": if an algo takes **neither min nor max** amount of time to run to completion then it is referred as an average case time complexity.

Asympotic Notations:

1. Big Omega (Ω) : this notation is used to denote **best case time complexity** – also called as **asymptotic lower bound,** running time of an algorithm cannot be less than its asymptotic lower bound.

2. Big Oh (O) : this notation is used to denote **worst case time complexity** - also called as **asymptotic upper bound,** running time of an algorithm cannot be more than its asymptotic upper bound.

3. Big Theta (θ) : this notation is used to denote an **average case time complexity** - also called as **asymptotic tight bound,** running time of an algorithm cannot be less than its asymptotic lower bound and cannot be more than its asymptotic upper bound i.e. it is **tightly bounded.**

Data Structures: Searching Algorithms

1. Linear Search / Sequential Search:

 # Algorithm :

Step-1 : accept key from the user

Step-2 : start traversal of an array and compare value of the key with each array element sequentially from first element either till match is not found or max till last element, if key is matches with any of array element then return true otherwise return false if key do not matches with any of array element.

 # Pseudocode:

```
Algorithm LinearSearch(A, size, key){
    for( int index = 1 ; index <= size ; index++ ){
    if( arr[ index ] ==  key )
    return true;
    }
    return false;
}
```

Data Structures: Searching Algorithms

Best Case: If key is found at very first position in only 1 no. of comparison then it is considered as a best case and running time of an algorithm in this case is **O(1)** => and hence time complexity = **Ω(1)**

Worst Case: If either key is found at last position or key does not exists, in this case maximum **n** no. of comparisons takes place, it is considered as a worst case and running time of an algorithm in this case is **O(n)** => and hence time complexity = **O(n)**

Average Case: If key is found at any in between position it is considered as an average case and running time of an algorithm in this case is **O(n/2) => O(n)** => and hence time complexity = **θ(n)**

2. Binary Search/Logarithmic Search:

- This algorithm follows <u>**divide-and-conquer**</u> approach.

- To apply binary search on an array **prerequisite is that array elements must be in a sorted manner.**

Step-1: accept key from the user

Step-2: in first iteration, find/calculate **mid position** by the formula **mid=(left+right)/2**, (by means of finding mid position big size array gets divided logically into two subarrays, left subarray and right subarray. **Left subarray = left to mid-1** & **right subarray = mid+1 to right**).

Step-3 : compare value of key with an element which is at mid position, if key matches in very first iteration in only one comparison then it is considered as a **best case**, if key matches with mid pos element then return true otherwise if key do not matches then we have to go to next iteration, and in next iteration we go to search key either into the left subarray or into the right subarray.

Step-4 : repeat step-2 & step-3 till either key is not found or max till subarray is valid, if subarray is not valid then key is not found in this case return false.

Data Structures: Searching Algorithms

- as in each iteration 1 comparison takes place and search space is getting reduced by half.

n => n/2 => n/4 => n/8 ......

after iteration-1 => n/2 + 1 => $T(n) = (n/2^1) + 1$

after iteration-2 => n/4 + 2 => $T(n) = (n/2^2) + 2$

after iteration-3 => n/8 + 3 => $T(n) = (n/2^3) + 3$

Lets assume, after k iterations => **$T(n) = (n/2^k) + k$ ...... (equation-I)**

let us assume,

=> **$n = 2^k$**

=> $\log n = \log 2^k$ (by taking log on both sides)

=> $\log n = k \log 2$

=> $\log n = k$ (as log 2 ~= 1)

=> $k = \log n$

By substituting value of n & k in equation-I, we get

=> $T(n) = (n / 2^k) + k$

=> $T(n) = (2^k / 2^k) + \log n$

=> $T(n) = 1 + \log n$ => $T(n) = O(1 + \log n$ **) => $T(n) = O(\log n)$.**

Data Structures: Searching Algorithms

```
Algorithm BinarySearch(A, n, key)//A is an array of size "n", and key to be search
{
  left = 1;
  right = n;

  while( left <= right )
  {
    //calculate mid position
    mid = (left+right)/2;
    //compare key with an ele which is at mid position
    if( key == A[ mid ] )//if found return true
      return true;

    //if key is less than mid position element
    if( key < A[ mid ] )
    {
      right = mid-1;//search key only in a left subarray
    }
    else//if key is greater than mid position element
    {
      left = mid+1;//search key only in a right subarray
    }
  }//repeat the above steps either key is not found or max any subarray is valid
  return false;
}
```

Data Structures: Searching Algorithms

Best Case: if the key is found in very first iteration at mid position in only 1 no. of comparison / if key is found at root position it is considered as a best case and running time of an algorithm in this case is O(1) = **Ω(1).**

Worst Case: if either key is not found or key is found at leaf position it is considered as a worst case and running time of an algorithm in this case is O(log n) = **O(log n).**

Average Case: if key is found at non-leaf position it is considered as an average case and running time of an algorithm in this case is O(log n) = **θ(log n).**

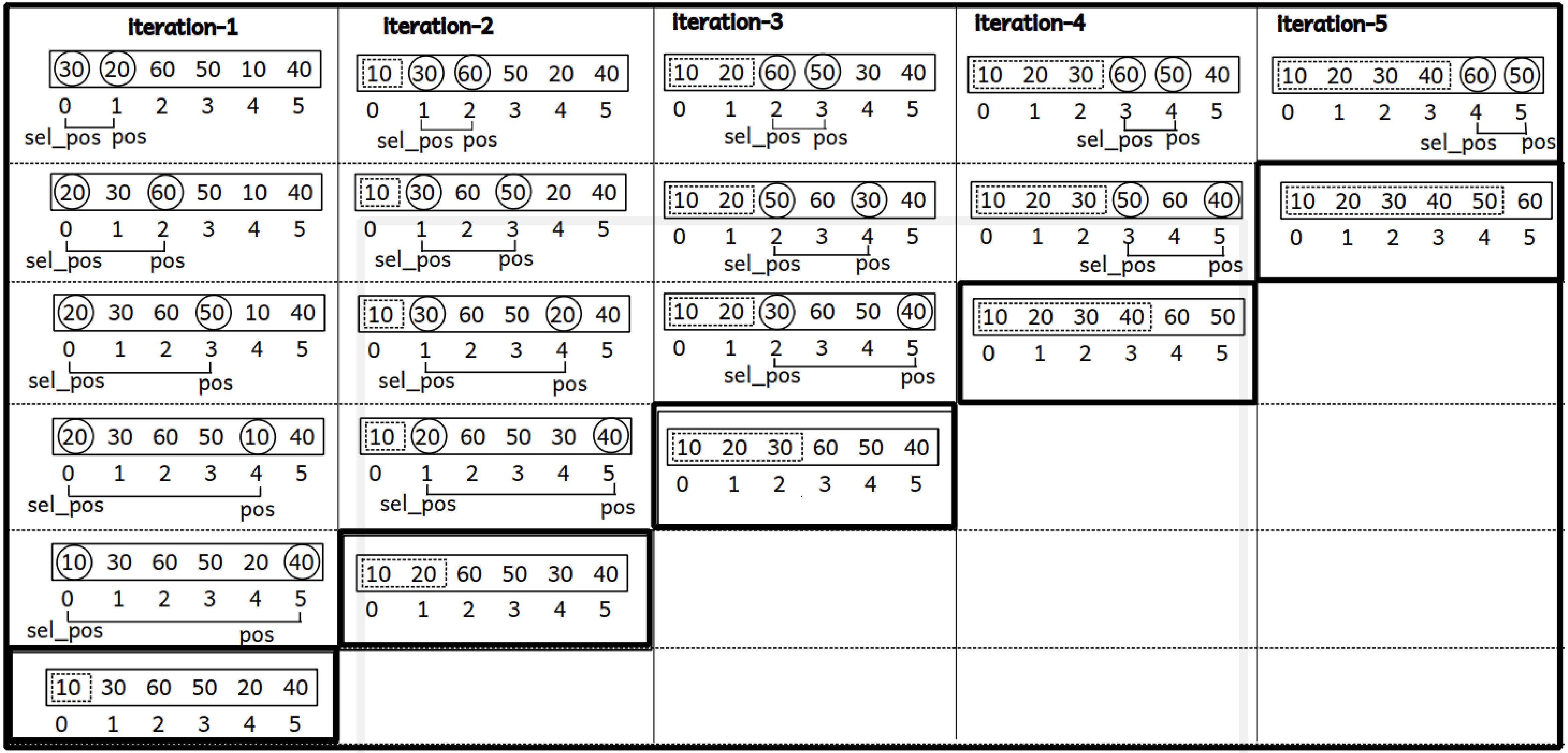
Data Structures: Sorting Algorithms

1. Selection Sort:

- In this algorithm, in first iteration, **first position gets selected** and **element which is at selected position gets compared with all its next position elements**, **<u>if selected position element found greater than any other position element then swapping takes place</u>** and in first iteration **smallest element** gets setteled at first position.

- In the second iteration, **second position gets selected and element which is at selected position gets compared with all its next position elements, if selected position element found greater than any other position element then swapping takes place** and in second iteration **second smallest element** gets setteled at second position, and so on **<u>in maximum (n-1) no. of iterations all array elements gets arranged in a sorted manner.</u>**

Data Structures: Sorting Algorithms



Best Case : $\Omega(n^2)$

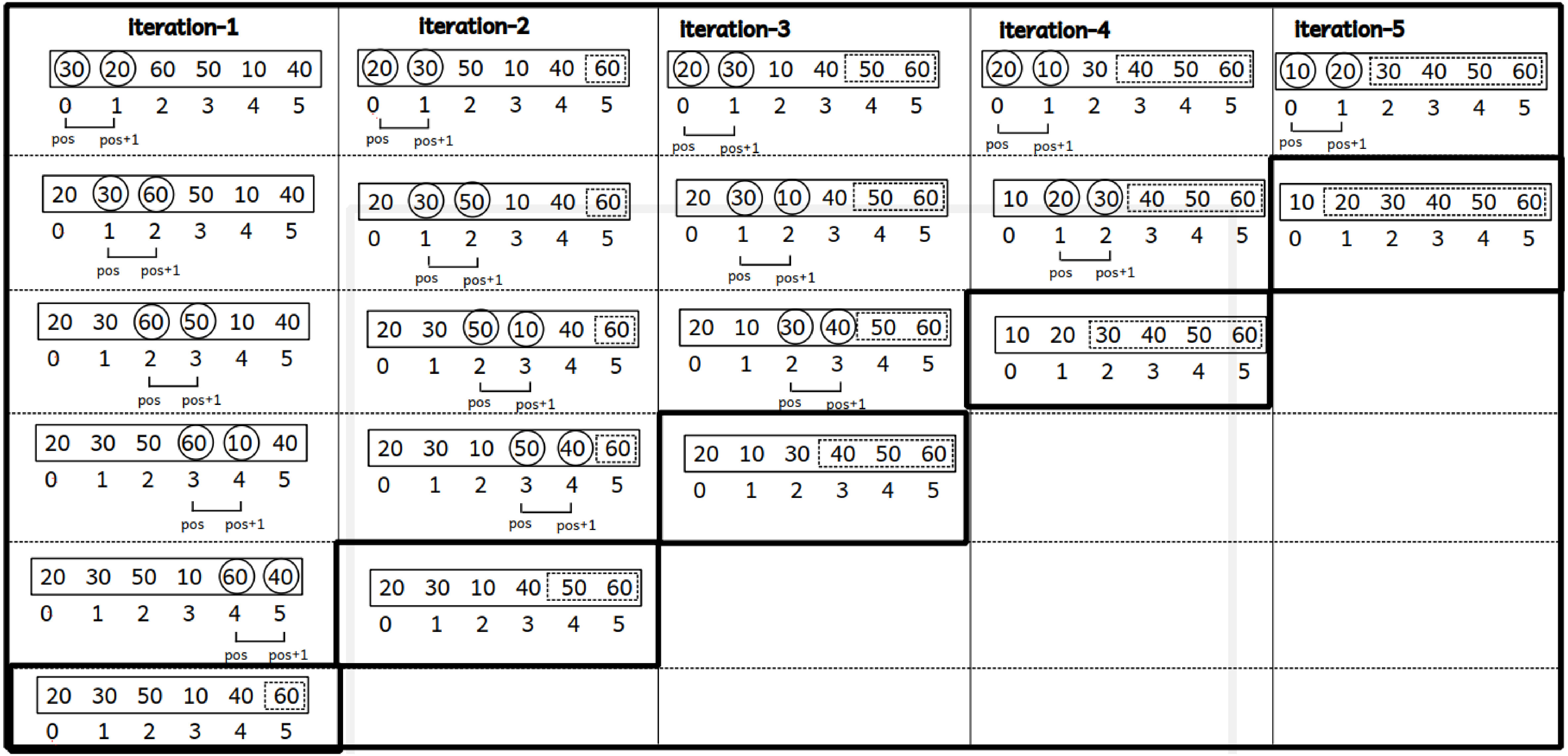
Worst Case : $O(n^2)$

Average Case : $\theta(n^2)$

2. Bubble Sort:

- In this algorithm, **in every iteration elements which are at two consecutive positions gets compared, if they are already in order then no need of swapping between them, but if they are not in order i.e. if prev position element is greater than its next position element then swapping takes place**, and by this logic in first iteration largest element gets setteled at last position, in second iteration second largest element gets setteled at second last position and so on, **in max (n-1) no. of iterations all elements gets arranged in a sorted manner.**

Data Structures: Sorting Algorithms



Best Case **: Ω(n) –** if array elements are already arranged in a sorted manner.

Worst Case **: $O(n^2)$**

Average Case **: $\theta(n^2)$**

3. Insertion Sort:

- In this algorithm, in every iteration one element gets selected as a **key element** and key element gets inserted into an array at its appropriate position towards its left hand side elements in a such a way that elements which are at left side are arranged in a sorted manner, and so on, in max **(n-1)** no. of iterations all array elements gets arranged in a sorted manner.
- **This algorithm works efficiently for already sorted input sequence by design** and hence running time of an algorithm is O(n) and it is considered as a best case.

Best Case : $\Omega(n)$ – if array elements are already arranged in a sorted manner.

Worst Case : $O(n^2)$

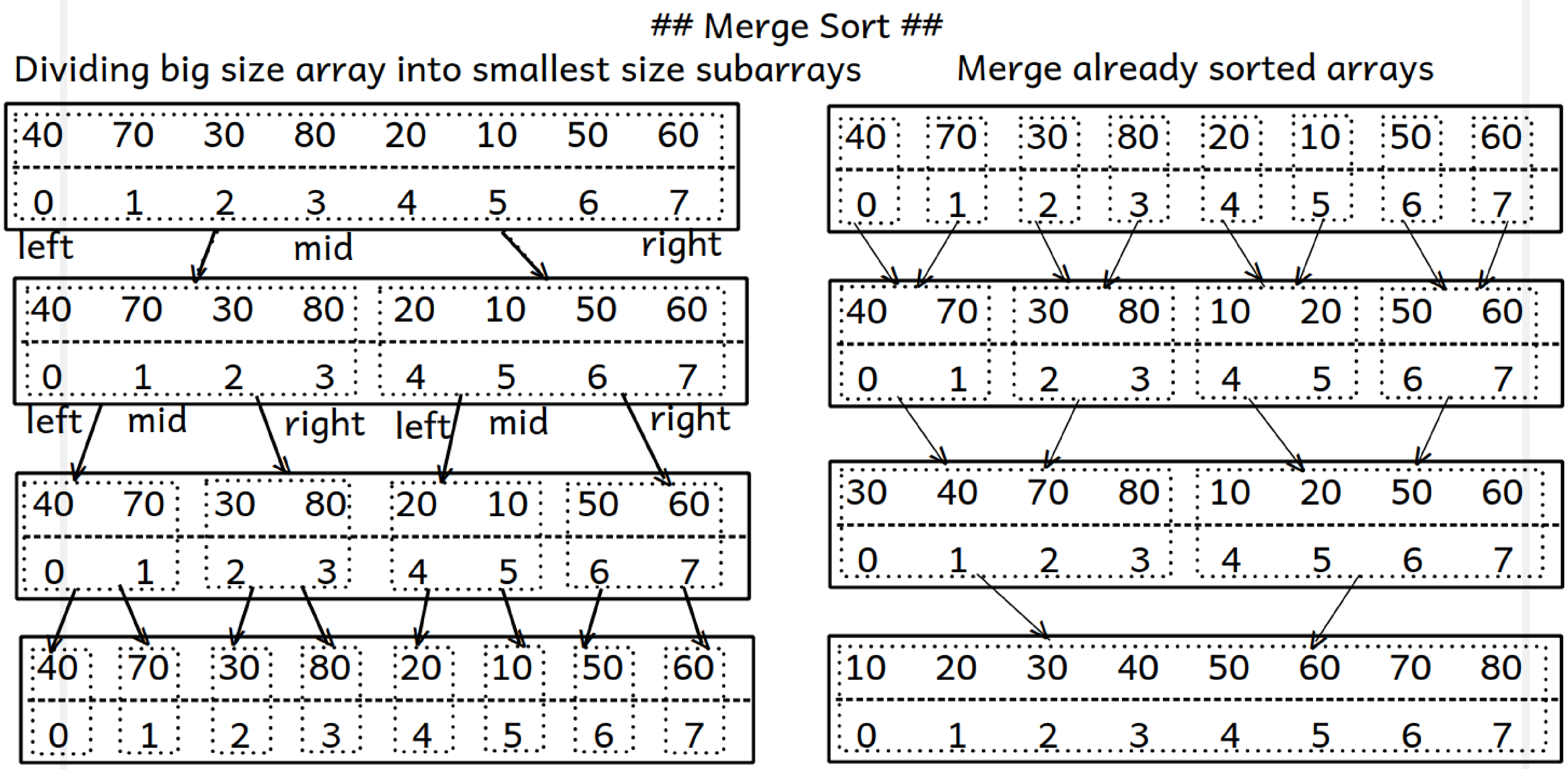
Average Case: $\theta(n^2)$

- Insertion sort algortihm is an efficient algorithm for smaller input size array.

4. Merge Sort:

- This algorithm follows **divide-and-conquer** approach.
- In this algorithm, big size array is divided logically into smallest size (i.e. having size 1) subarrays, as if size of subarray is 1 it is sorted, after dividing array into sorted smallest size subarray’s, subarrays gets merged into one array step by step in a sorted manner and finally all array elements gets arranged in a sorted manner.
- This algorithm works fine for **even** as well **odd** input size array.
- This algorithm takes extra space to sort array elements, and hence its space complexity is more.

Data Structures: Sorting Algorithms



Best Case : Ω(n log n)
Worst Case : O(n log n)
Average Case : θ(n log n)

5. Quick Sort:
- This algorithm follows **divide-and-conquer** approach.
- In this algorithm the basic logic is a **partitioning.**
- Partitioning: in parititioning, pivot element gets selected first (it may be either leftmost or rightmost or middle most element in an array), after selection of pivot element all the elements which are smaller than pivot gets arranged towards as its left as possible and elements which are greater than pivot gets arranged as its right as possible, and big size array is divided into two subarray’s, so after first pass pivot element gets settled at its appropriate position, elements which are at left of pivot is referred as **left partition** and elements which are at its right referred as a **right partition.**

Best Case : $\Omega(n \log n)$

Worst Case : $O(n^2)$ – worst case rarely occures

Average Case : $\theta(n \log n)$

- Quick sort algortihm is an efficient sorting algorithm for larger input size array.

Data Structures: Linked List

- Limitations of an array data structure:

1. Array is static, i.e. size of an array is fixed, its size cannot be either grow or shrink during runtime.

2. Addition and deletion operations on an array are not efficient as it takes O(n) time, and hence to overcome these two limitations of an Array data structure **Linked List** data structure has been designed.

Linked List: It is a basic/linear data structure, which is a collection/list of logically related similar type of elements in which, an address of first element in a collection/list is stored into a pointer variable referred as a head pointer and each element contains actual data and link to its next element i.e. an address of its next element (as well as an addr of its previous element).

- An element in a Linked List is also called as a **Node.**

- Four types of linked lists are there: **Singly Linear Linked List, Singly Circular Linked List, Doubly Linear Linked List and Doubly Circular Linked List.**

Data Structures: Linked List

- Basically we can perform **addition, deletion, traversal** etc... operations onto the linked list data structure.

- We can add and delete node into and from linked list by three ways:

add node into the linked list **at last position**, **at first position** and **at any specific position**, simillarly we can delete node from linked list which is **at first position**, **at last position** and **at any specific position.**

1. Singly Linear Linked List: It is a type of linked list in which

- head always contains an address of first element, if list is not empty.

- each node has two parts:

i. data part : it contains actual data of any primitive/non-primitive type.

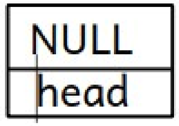
ii. pointer part (next) : it contains an address of its next element/node.

- last node points to NULL, i.e. next part of last node contains NULL.

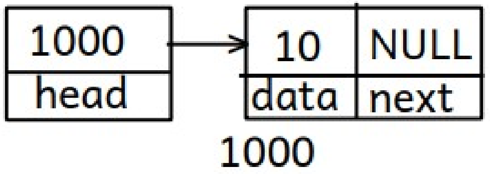
Data Structures: Linked List

SINGLY LINEAR LINKED LIST

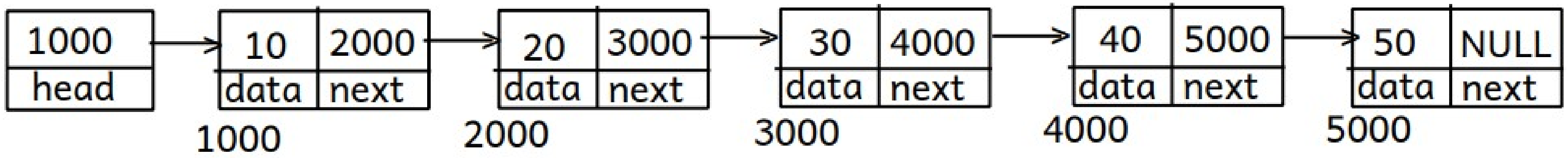
1) singly linear linked list --> list is empty



2) singly linear linked list --> list contains only one node



3) singly linear linked list --> list contains more than one nodes



Data Structures: Linked List

Limitations of Singly Linear Linked List:

- Add node at last position & delete node at last position operations are not efficient as it takes O(n) time.

- We can start traversal only from first node and can traverse the list only in a forward direction.

- Previous node of any node cannot be accesed from it.

- Any node cannot be revisited – to overcome this limitation Singly Circular Linked List has been designed.

2. Singly Circular Linked List: It is a type of linked list in which

- head always contains an address of first node, if list is not empty.

- each node has two parts:

i. data part : contains data of any primitive/non-primitive type.

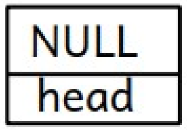
ii. pointer part(next) : contains an address of its next node.

- last node points to first node, i.e. next part of last node contains an address of first node.

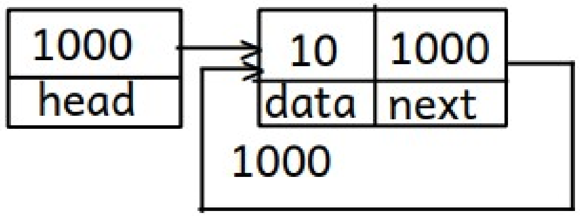
Data Structures: Linked List

SINGLY CIRCULAR LINKED LIST

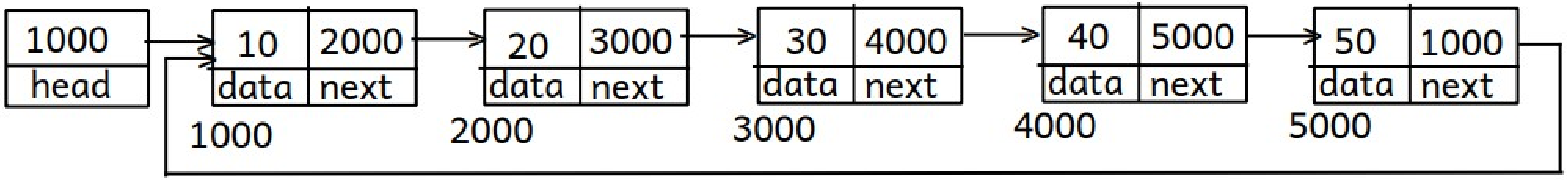
1) singly circular linked list --> list is empty



2) singly circular linked list --> list contains only one node



3) singly circular linked list --> list contains more than one nodes



Data Structures: Linked List

Limitations of Singly Circular Linked List:

- Add last, delete last & add first, delete first operations are not efficient as it takes O(n) time.

- We can starts traversal only from first node and can traverse the SCLL only in a forward direction.

- **Previous node of any node cannot be accesed from it** – to overcome this limitation Doubly Linear Linked List has been designed.

3. Doubly Linear Linked List: It is a linked list in which

- head always contains an address of first element, if list is not empty.

- each node has three parts:

i. data part: contains data of any primitive/non-primitive type.

ii. pointer part(next): contains an address of its next element/node.

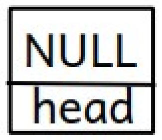
iii .pointer part(prev): contains an address of its previous element/node.

- next part of last node & prev part of first node points to NULL.

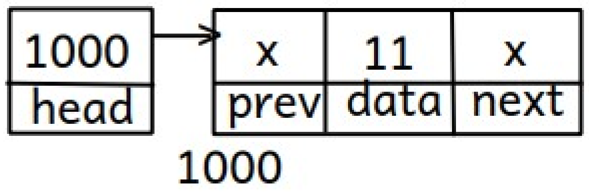
Data Structures: Linked List

DOUBLY LINEAR LINKED LIST

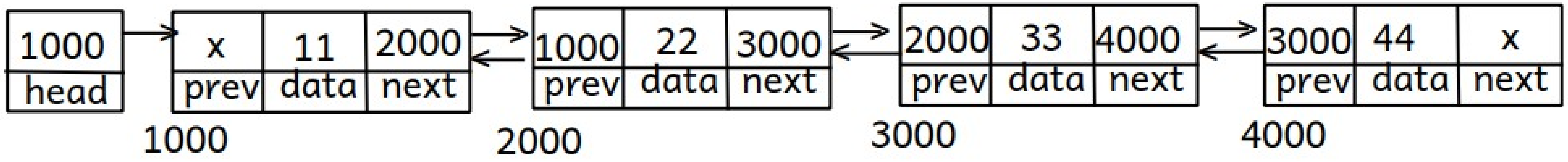
1. doubly linear linked list --> list is empty



2. doubly linear linked list --> list is contains only one node



3. doubly linear linked list --> list is contains more than one nodes



Data Structures: Linked List

Limitations of Doubly Linear Linked List:

- **Add last and delete last** operations are not efficient as it takes **O(n)** time.

- We can starts traversal only from first node, and hence to overcome these limitations **Doubly Circular Linked List** has been designed.

4. Doubly Circular Linked List: It is a linked list in which

- head always contains an address of first node, if list is not empty.

- each node has three parts:

i. data part: contains data of any primitive/non-primitive type.

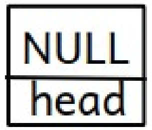
ii. pointer part(next): contains an address of its next element/node.

iii .pointer part(prev): contains an address of its previous element/node.

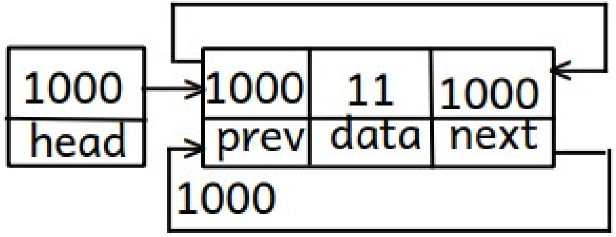
- **next part of last node contains an address of first node** & **prev part of first node contains an address of last node.**

DOUBLY CIRCULAR LINKED LIST

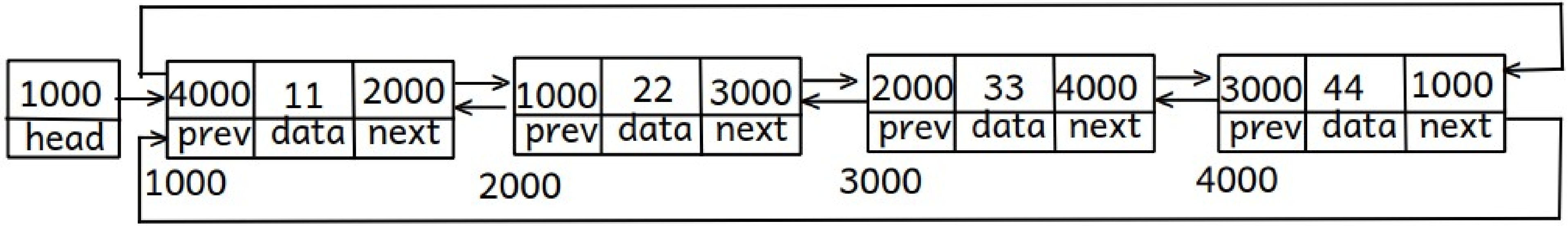
1. doubly circular linked list --> list is empty



2. doubly circular linked list -> list is contains only one node



3. doubly circular linked list --> list is contains more than one nodes



Data Structures: Linked List

Advantages of Doubly Circular Linked List:

- DCLL can be traverse in forward as well as in a backward direction.

- **Add last, add first, delete last & delete first** operations are efficient as it takes **O(1)** time and are convenient as well.

- Traversal can be start either from first node (i.e. from head) or from last node (from head.prev) in O(1) mtime.

- Any node can be revisited.

- Previous node of any node can be accessed from it

 # Array v/s Linked List => Data Structure:

- Array is **static** data structure whereas linked list is **dynamic** data structure.

- Array elements can be accessed by using **random access method** which is **efficient** than **sequential access method** used to access linked list elements.

- **Addition & Deletion operations are efficient** on linked list than on an array.

- Array elements gets stored into the **stack section**, whereas linked list elements gets stored into **heap section.**

- In a linked list **extra space is required to maintain link between elements**, whereas in an array to maintained link between elements is the job of the **compiler**.
- searching operation is faster on an array than on linked list as on linked list we cannot apply binary search.

Data Structures: Stack

Stack: It is a collection/list of logically related similar type elements into which data elements can be added as well as deleted from only one end referred **top** end.

- In this collection/list, element which was inserted last only can be deleted first, so this list works in **last in first out/first in last out** manner, and hence it is also called as **LIFO list**/FILO list.

- We can perform basic three operations on stack in **O(1)** time: **Push, Pop & Peek.**

1. Push : **to insert/add an element onto the stack at top position**

step1: check stack is not full

step2: increment the value of top by 1

step3: insert an element onto the stack at top position.

2. Pop : to delete/remove an element from the stack which is at top position

step1: check stack is not empty

step2: decrement the value of top by 1.

Data Structures: Stack

3. Peek : to get the value of an element which is at top position without push & pop.

step1: check stack is not empty

step2: return the value of an element which is at top position

Stack Empty : top == -1

Stack Full : top == SIZE-1

 # Applications of Stack:

- Stack is used by an OS to control of flow of an execution of program.
- In recursion internally an OS uses a stack.
- undo & redo functionalities of an OS are implemented by using stack.
- Stack is used to implement advanced data structure algorithms like **DFS: Depth First Search** traversal in tree & graph.
- Stack is used in an algorithms to covert given infix expression into its equivalent postfix and prefix, and for postfix expression evaluation.

Data Structures: Stack

- Algorithm to convert given infix expression into its equivalent postfix expression:

Initially we have, an Infix expression, an empty Postfix expression & empty Stack.

```
# algorithm to convert given infix expression into its equivalent postfix expression
step1: start scanning infix expression from left to right
step2:
    if( cur ele is an operand )
        append it into the postfix expression
    else//if( cur ele is an operator )
    {
        while( !is_stack_empty(&s) && priority(topmost ele) >= priority(cur ele) )
        {
            pop an ele from the stack and append it into the postfix expression
        }

        push cur ele onto the stack
    }
step3: repeat step1 & step2 till the end of infix expression
step4: pop all remaining ele's one by one from the stack and append them into the
postfix expression.
```

- Algorithm to convert given infix expression into its equivalent prefix expression:

Initially we have, an Infix expression, an empty Prefix expression & empty Stack.

```
# algorithm to convert given infix expression into its equivalent prefix:
step1: start scanning infix expression from right to left
step2:
    if( cur ele is an operand )
        append it into the prefix expression
    else//if( cur ele is an operator )
    {
        while( !is_stack_empty(&s) && priority(topmost ele) > priority(cur ele) )
        {
            pop an ele from the stack and append it into the prefix expression
        }

        push cur ele onto the stack
    }
step3: repeat step1 & step2 till the end of infix expression
step4: pop all remaining ele's one by one from the stack and append them into the
prefix expression.
step5: reverse prefix expression - equivalent prefix expression.
```

Data Structures: Queue

Queue: It is a collection/list of logically related similar type of elements into which elements can be added from one end referred as **rear** end, whereas elements can be deleted from another end referred as a **front** end.

- In this list, element which was inserted first can be deleted first, so this list works in **first in first out** manner, hence this list is also called as **FIFO list/LILO list.**

- Two basic operations can be performed on queue in O(1) time.

1. Enqueue: to insert/push/add an element into the queue from rear end.

2. Dequeue: to delete/remove/pop an element from the queue which is at front end.

- There are different types of queue:

1. Linear Queue (works in a fifo manner)

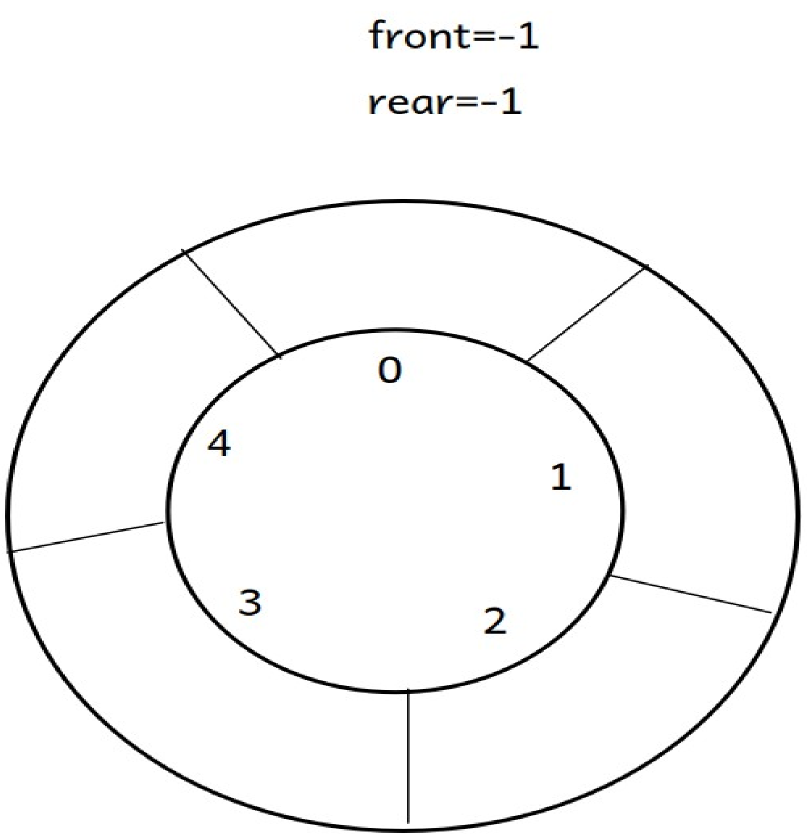
2. Circular Queue (works in a fifo manner)

3. Priority Queue: it is a type of queue in which elements can be inserted from rear end randomly (i.e. without checking priority), whereas an element which is having highest priority can only be deleted first.

- Priority queue can be implemented by using linked list, whereas it can be implemented efficiently by using **binary heap.**

4. Double Ended Queue (deque) : it is a type of queue in which elements can added as well as deleted from both the ends.

Data Structures: Queue



Circular Queue
is_queue_full : front == (rear+1)%SIZE
is_queue_empty : rear == -1 && front == rear

1. "enqueue": to insert/add/push an element into the queue from rear end:
step1: check queue is not full
step2: increment the value of rear by 1 [rear = (rear+1)%SIZE]
step3: push/add/insert an ele into the queue at rear position
step4: if(front == -1)
front = 0

2. "dequeue": to remove/delete/pop an element from the queue which is at front position.
step1: check queue is not empty
step2:
if(front == rear)//if we are deleting last ele
front = rear = -1;
else
increment the value of front by 1 [i.e. we are deleting an ele from the queue]. [front = (front+1)%SIZE]

Data Structures: Queue

Applications of Queue:

- Queue is used to implement OS data structures like **job queue, ready queue, message queue, waiting queue** etc...

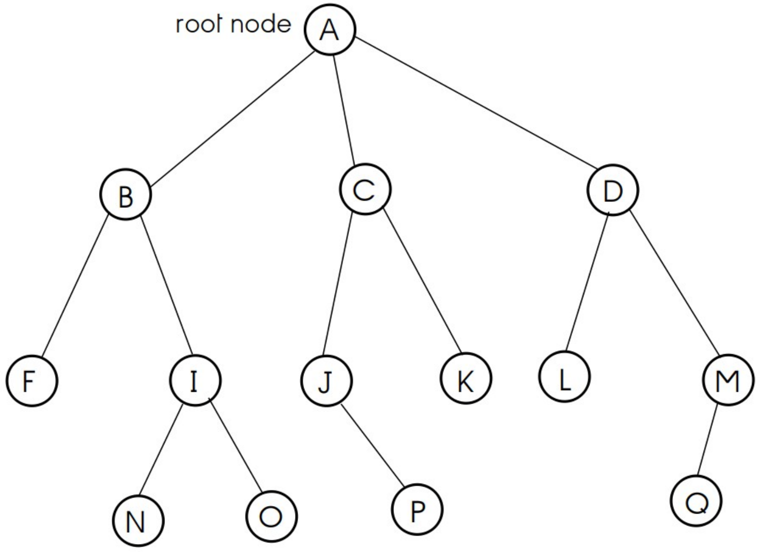
- Queue is used to implement OS algorithms like **FCFS CPU Scheduling, Priority CPU Scheduling, FIFO Page Replacement** etc...

- Queue is used to implenent an advanced data structure algorithms like **BFS: Breadth First Search** Traversal in tree and graph.

- Queue is used in any application/program in which list/collection of elements should works in a **first in first out manner or whereaver it should works according to priority.**

Data Structures: Tree

Tree: It is a **non-linear / advanced data structure** which is a **collection of finite no. of logically related similar type of data elements** in which, there is a first specially designated element referred as a **root element**, and remaining all elements are connected to it in a **hierarchical manner**, follows **parent-child relationship**.



Tree: Data Structure

Data Structures: Tree

- **siblings/brothers:** child nodes of same parent are called as siblings.
- **ancestors:** all the nodes which are in the path from root node to that node.
- **descedents:** all the nodes which can be accessible from that node.
- **degree of a node** = no. of child nodes having that node
- **degree of a tree** = max degree of any node in a given tree
- **leaf node/external node/terminal node:** node which is not having any child node OR node having degree 0.
- **non-leaf node/internal node/non-terminal node:** node which is having any no. of child node/s OR node having non-zero degree.
- **level of a node** = level of its parent node + 1
- **level of a tree** = max level of any node in a given tree (by assuming level of root node is at level 0).
- **depth of a tree** = max level of any node in a given tree.
- as tree data structure can grow upto any level and any node can have any number of child nodes, operations on it becomes unefficient, so restrictions can be applied on it to achieve efficiency and hence there are diefferent types of tree.

Data Structures: Tree

- Binary tree: it is a tree in which each node can have max 2 number of child nodes, i.e. each node can have either 0 OR 1 OR 2 number of child nodes.

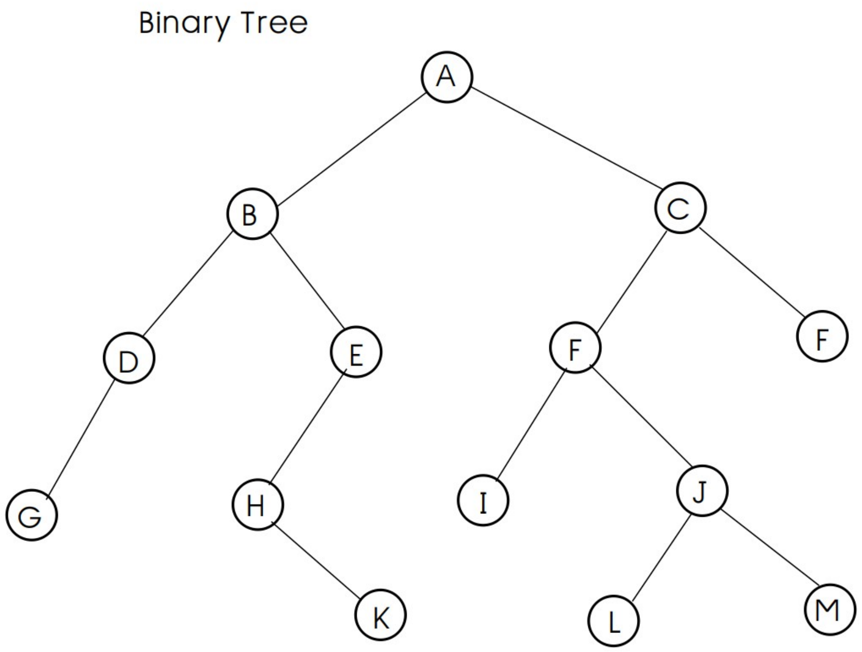
OR

Binary tree: it is a set of finite number of elements having three subsets:

1. root element

2. left subtree (may be empty)

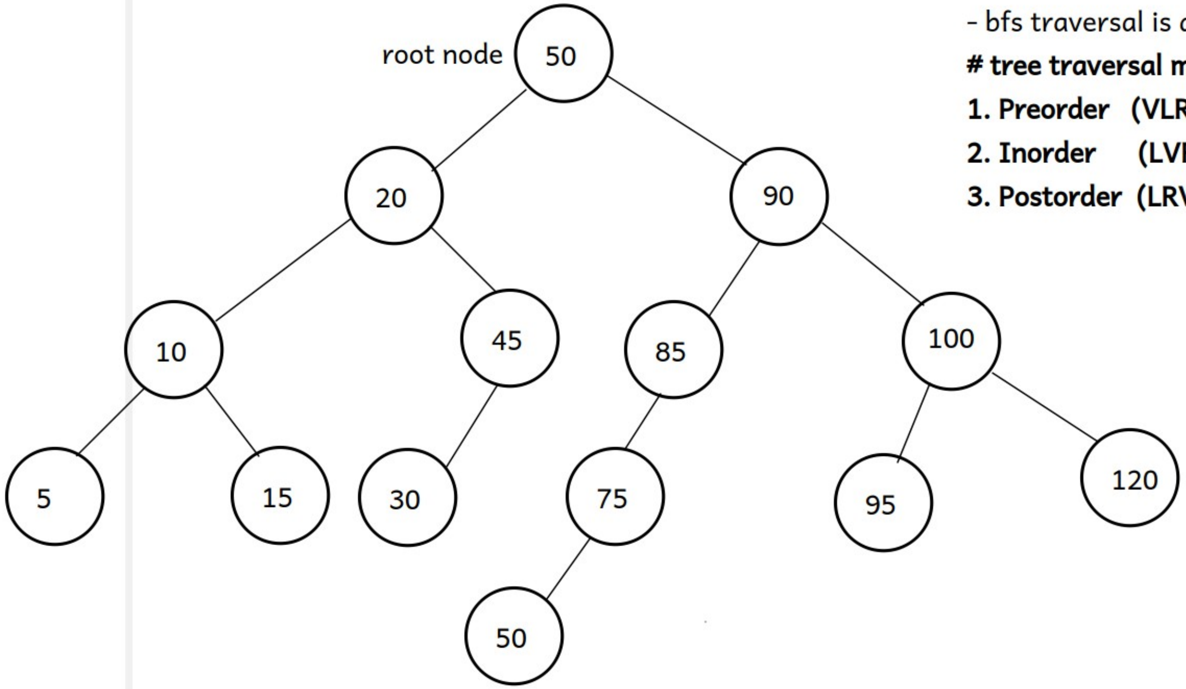
3. right subtree (may be empty)



Data Structures: Tree

- **Binary Search Tree(BST):** it is a **binary tree** in which left child is always smaller than its parent and right child is always greater than or equal to its parent.

input order of an ele's for BST: 50 20 90 85 10 45 30 100 15 75 95 120 5 50



1. dfs traversal: 50 20 10 5 15 45 30 90 85 75 50 100 95 120

2. bfs traversal: 50 20 90 10 45 85 100 5 15 30 75 95 120 50

- bfs traversal is also called as "levelwise traversal".

 # tree traversal methods on BST:

1. Preorder (VLR) : 50 20 10 5 15 45 30 90 85 75 50 100 95 120

2. Inorder (LVR): 5 10 15 20 30 45 50 50 75 85 90 95 100 120

3. Postorder (LRV): 5 15 10 30 45 20 50 75 85 95 120 100 90 **50**

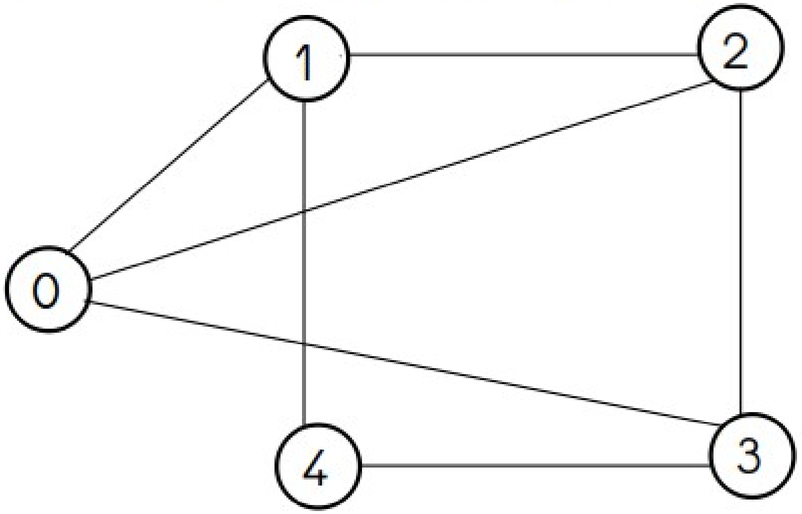
Data Structures: Graph

Graph: It is **non-linear**, **advanced** data structure, which is a collection of logically related similar and disimmilar type of elements which contains:

- set of finite no. of elements referred as a **vertices**, also called as **nodes**, and

- set of finite no. of ordered/unordered pairs of vertices referred as an **edges**, also called as an **arcs**, whereas it may carries weight/cost/value (cost/weight/value may be -ve).

G(V,E): V={0,1,2,3,4}; E={ (0,1),(0,2),(0,3),(1,2),(1,4),(2,3),(3,4) }



Data Structures: Graph

- If there exists a direct edge between two vertices then those vertices are referred as **adjacent vertices** otherwiese **non-adjacent**.

- if we can represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices i.e. undirected edge.**

(u,v) == (v,u) --> unordered pair of vertices -> undirected edge -> undirected graph

- if we cannot represent any edge either (u,v) OR (v,u) then it is referred as **unordered pair of vertices** i.e. directed edge.

(u,v) != (v,u) --> ordered pair of vertices -> directed edge -> directed graph (di-graph).

- complete graph: if all the vertices are adjacent to remaining all vertices in a given graph.

- connected vertices: if path exists between two vertices then those two vertices are referred as a connected vertices otherwise not-connected.

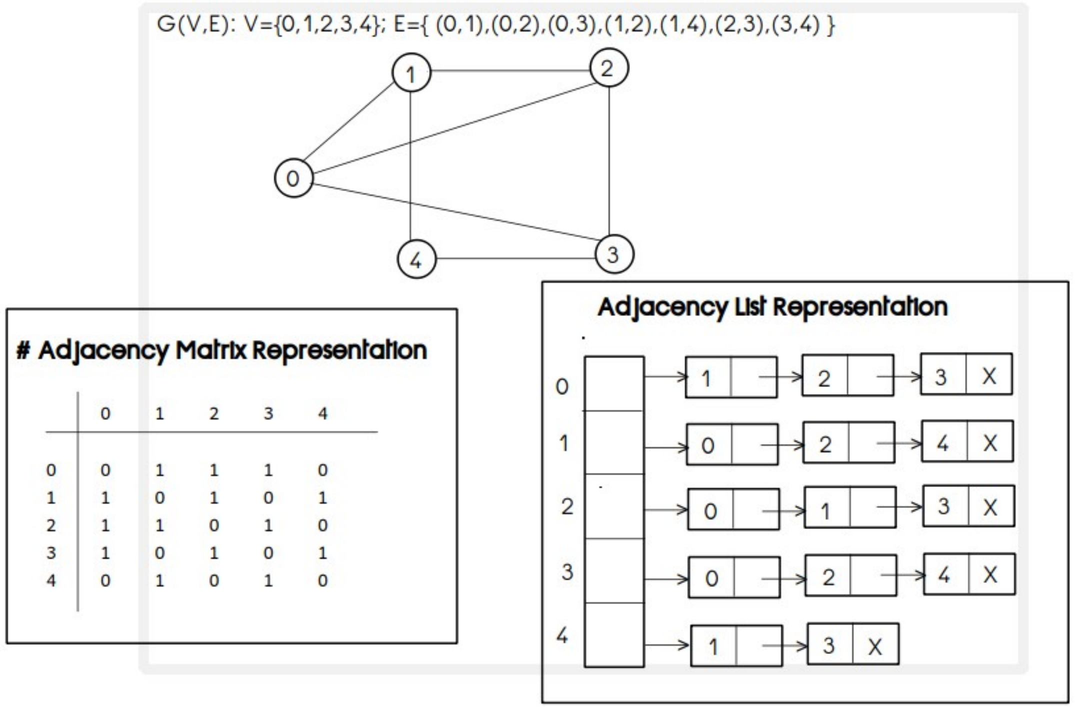
- connected graph: if any vertex is connected to remaining all vertices in a given graph

Data Structures: Graph

- There are two graph representation methods:

1. Adjacency Matrix Representation (2-D Array)

2. Adjacency List Representation (Array of Linked Lists)



Data Structures: Hash Table

Hash Table: it is a **non-linear/advanced data structure** which is a **collection of finite number of logically related similar type of data elements/records** gets stored into the memory in an **associative manner i.e. in a key-value pairs** (for faster searching).

Hashing: It is an improvement over **"Direct Access Table"** in which hash function can be used and the table is reffered as "Hash Table".

Hash Function: it is a function that **converts a given big key value/number into a small practical integer value/key** which is reffered as **hash key/hash code** which is a mapped value can be used as an index in a hash table.

Collision: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some **collision handling technique.**

- There are two **collission handling techniques:**

1. Chaining/Seperate Chaining

2. Open Addressing

Data Structures: Hash Table

1. Chaining:

- The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

- Advantages:

1. Simple to implement.
2. Hash table never fills up, we can always add more elements to the chain.
3. Less sensitive to the hash function or load factors.
4. It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

- Disadvantages:

1. Cache performance of chaining is not good as keys are stored using a linked list. Open addressing provides better cache performance as everything is stored in the same table.
2. Wastage of Space (Some Parts of hash table are never used).
3. If the chain becomes long, then search time can become O(n) in the worst case.
4. Uses extra space for links.

Data Structures: Hash Table

2. Open Addressing:
- In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.
- Open Addressing is done following ways:

A. Linear Probing:
- In linear probing, we linearly probe/search for next slot.
For example, typical gap between two probes is 1 as taken in below example also.
- let hash(x) be the slot index computed using hash function and S be the table size

  If slot **hash(x) % S** is full, then we try **(hash(x) + 1) % S**
  If **(hash(x) + 1) % S** is also full, then we try **(hash(x) + 2) % S**
  If **(hash(x) + 2) % S** is also full, then we try **(hash(x) + 3) % S**
  ....
  ....

Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

Data Structures: Hash Table

B. Quadratic Probing:

- We look for i^2th slot in ith iteration.
- let **hash(x)** be the slot index computed using hash function.

If slot **hash(x) % S** is full, then we try **(hash(x) + 1*1) % S**

If **(hash(x) + 1*1) % S** is also full, then we try **(hash(x) + 2*2) % S**

If **(hash(x) + 2*2) % S** is also full, then we try **(hash(x) + 3*3) % S**

....

....

C. Double Hashing:

- We use another hash function **hash2(x)** and look for **i*hash2(x)** slot in i'th rotation.
- let hash(x) be the slot index computed using hash function.

If slot **hash(x) % S** is full, then we try **(hash(x) + 1*hash2(x)) % S**

If **(hash(x) + 1*hash2(x)) % S** is also full, then we try **(hash(x) + 2*hash2(x))%S**

If **(hash(x) + 2*hash2(x)) % S** is also full, then we try **(hash(x) + 3*hash2(x)) % S**

....

....