Data Structure:
- way of organizing the data in the memory

- **Primitive**
    * int
    * char
    * float
- **Non-Primitive**
    * *Linear*
        List
        Stack
        Queue
        Array
    * *Non-Linear*
        Tree
        Graph

Collections:
- It is a framework that helps to store group of objects
- performs operations like searching, sorting, insertion, deletion, etc
- all elements must be object
- java.util package

Ex: List of Student Names, Set of Roll Numbers, Group of Phone Numbers

Arrays and Collection

Arrays	**Collections**
Arrays are fixed in size	dynamically grows
store both primitive data and objects	only objects
coding is difficult	coding is easy
store similar type of data	can store dissimilar data

```
10 20 30 40 50
10 20 30 40    50
10 20 30    40 50

 int arr[ ]=new int[5];
 Employee emp[ ]=new Employee[5];
```

Java Collections API:
- List Interface
- Set Interface
- Map interface

Hierarchy

```
                              Iterable
                                 |
                             Collection
                                 |
             List                Set                  Queue
              |                   |
 ArrayList  LinkedList  Vector  HashSet     SortedSet
                                  |             |
                            LinkedHashSet    TreeSet
```

Methods:
- size()
- isEmpty()
- contains()
- add()
- remove()

List: store duplicate elements
- Can have multiple null values
- Follows order of insertion

Ex: Student Names, Employee Name, Employee Salary

* **ArrayList**
- allows to store duplicate elements
- follows the order of insertion
- implementation using array
- allows null

* **LinkedList**
- allows to store duplicate elements
- follows the order of insertion
- implementation using linked list

* **Vector**
- allows to store duplicate elements
- follows the order of insertion

Ex: ArrayList

```
ArrayList al=new ArrayList();
al.add("ABC");
al.add(34);
al.add(25.5);
s.o.p(al);
```

```
        ArrayList<String> al=new ArrayList<String>();
        al.add("ABC");
        al.add("XYZ");
        al.add(23);   //not possible

          ArrayList<int> al=new ArrayList<int>();  // not possible to create collection for primitive
          type
         ArrayList<Integer> al=new ArrayList<Integer>();
```

Ex: LinkedList

```
        LinkedList<String> list=new LinkedList<>();
        list.add("ABC");
        list.add("MN");
        list.add(1,"PQR");
        list.addFirst("DEF");
        list.addLast("SSS");
        list.remove(2);
        list.remove("SSS");
        list.removeFirst();
        list.removeLast();
        list.add("MN");
        list.add("MN");
        list.removeFirstOccurrence("MN");
        list.removeLastOccurrence("MN");

        LinkedList<String> list1=new LinkedList<>();
        list1.add("BBB");
        list1.add("CCC");

        list.addAll(list1);
        list.removeAll(list1);

        s.o.p(list);
```

Iterating Through Collection:

- for loop
- Advanced for loop
- iterator
- forEach() - introduced in Java8

Advanced for loop:

```
  for(String s:list){
        s="welcome "+s;
        s.o.p(s);
  }
```

Iterator:

```
  Iterator<String> itr=list.iterator();
  while(itr.hasNext()){
      s.o.p(itr.next());
  }
```

forEach():

```
    list.forEach((s)-> {
                        s="Welcome "+s;
                        s.o.p(s);
                    }
                );
```

Set:

- Set is a collection that stores unique elements (do not contain duplicates)
- Allow only one null value
  Ex: EmployeeId, RegNo, emailId, MobileNo

Types:

* **HashSet**
  - stores unique values
  - allows only one null
  - do not follow the order of insertion
* **LinkedHashSet**
  - stores unique values
  - allows only one null
  - follows the order of insertion
* **TreeSet**
  - stores unique values
  - do not allow null value
  - follows the sorted order (Ascending order)

Ex:

```
    HashSet hs=new HashSet();
    hs.add("ABC");
    hs.add("PQR");
    hs.add("ABC");   // duplicate can not be added

    LinkedHashSet<String> lhset=new LinkedHashSet<>();
    lhset.add("ABC");
    lhset.add("XYZ");

    TreeSet<String> treeSet=new TreeSet<>();
    treeSet.add("XYZ");
    treeSet.add("PQR");
    treeSet.add("ABC");
```

Ex:

```
   class Student{
          private int stuId;
          private String stuName;
          ......
   }
   class Tester{
       public static void main(String[] args){
          Student s1=new Student(1,"ABC");
          Student s2=new Student(2,"PQR");
          Student s3=new Student(3,"XYZ");

          List<Student> list=new ArrayList<>();
          list.add(s1);
          list.add(s2);
          list.add(s3);

          for(Student s:list){
              s.o.p(s.getStuId()+"    "+s.getStuName());
          }
```

Map:

- key : value pairs
- both key and value must be object
- keys can not be duplicated
- values can be duplicated

Hierarchy:

```
                                         Map
                                           |
    SortedMap      HashMap        LinkedHashMap      HashTable
           |
     TreeMap
```

Methods:

- get()
- put()
- remove()
- size()
- isEmpty()

Types:

*** HashMap**
- stores unique key
- do not follow any order
- null values are allowed

*** LinkedHashMap**
- stores unique key
- follows the order of insertion
- null values are allowed

*** TreeMap**
- follows sorted order
- null values are not allowed

*** HashTable**
- null values are not allowed

Ex:

```
    Map<Integer, String> map=new HashMap<>();
    map.put(101,"ABC");
    map.put(102,"XYZ");
    map.put(103,"PQR");
    map.put(101,"MN");

    Set<Integer> keys=map.keySet();

    Iterator<Integer> itr=keys.iterator();

    while(itr.hasNext()){
        Integer k=itr.next();
        String s=map.get(k);
        s.o.p(k+"   "+s);
    }
```

Comparable and Comparator:

- helps to sort user defined objects
- Collections.sort() - automatically perform sorting operation

Comparable:
- has abstract method compareTo()
- sorting will be performed based on only one field
  empId or empName or salary
- java.lang

Comparator:
- has abstract method compare()
- sorting based on multiple fields
- can have more than one comparator class

 - java.util

Ex: Comparable

```
class Student implements Comparable<Student>{
    private int stuId;
    private String stuName;
    .....
    public int compareTo(Student s){
      if(this.stuId>s.stuId){
          return 1;
      }else if(this.stuId<s.stuId){
          return -1;
      }else{
          return 0;
      }
    }
 }
```

Ex: Comparator

```
 class Student{
    private int stuId;
    private String stuName;
    .....
    //getter and setter method
 }

 class StudentIdComparator implements Comparator<Student>{
    public int compare(Student s1, Student s2){
      if(s1.stuId>s2.stuId){
          return 1;
      }else if(s1.stuId<s2.stuId){
          return -1;
      }else{
          return 0;
      }
    }
  }
```

Queue:
- Linear Data Structure
- FIFO : First In First Out
- Insertion at one end called rear and deletion will be in another end called front
- Enqueue - inserting an element to the queue
- Dequeue - deleting an element from the queue

- LinkedList
- PriorityQueue
- ArrayQueue

Ex:
```
  Queue<String> queue=new LinkedList<>();
  queue.add("one");
  queue.add("two");
  queue.add("three");
  queue.add("four");

  queue.isEmpty();
  queue.size();
  queue.remove();

  Queue<Integer> queue=new PriorityQueue<>();
  queue.add(40);
  queue.add(20);
  queue.add(30);
  queue.add(5);
  s.o.p(queue);    //  5  40  20  30
  queue.remove();
  s.o.p(queue);    // 20  40  30
```

Deque: Double Ended Queue
- insertion and deletion takes places at both the ends

  * addFirst();
  * addLast();
  * removeFirst();
  * removeLast();
  * peekFirst();
  * peekLast();

Stack:
- LIFO: Last in First Out principle
  Ex: undo
- both insertion and deletion performed at one end is called top of the stack
- push -inserting element to stack
- pop - deleting an element from stack

- when try to push an element in full stack is called overflow condition
- when try to pop an element from empty stack is called unerflow condition
- peek - returns top element

- array
- Linkedlist

Sorting:

- arrangement of elements in the ascending order

```
* Bubble Sort
* Insertion Sort
* Quick Sort
```

Bubble Sort:

- repeatedly swap the adjacent elements

```
  Ex:  5   3  4  7  2

 First Pass:  5  3  4  7  2   - compare first two elements 1st and 2nd , if 2nd element is smaller
then swap
              3  5  4  7  2  - compare 2nd and 3rd elements
              3  4  5  7  2  - compare 3rd and 4th elements
              3  4  5  7  2  - compare 4th and 5th elements
              3  4  5  2  7
 Second Pass: 3  4  5  2  7   - compare 1 and 2
              3  4  5  2  7  - compare 2 and 3
              3  4  5  2  7  - compare 3 and 4
              3  4  2  5  7  - compare 4 and 5
              3  4  2  5  7
 Third Pass:  3  4  2  5  7   - compare 1 and 2
              3  4  2  5  7  - compare 2 and 3
              3  2  4  5  7  -
 Fourth Pass: 3  2  4  5  7   - compare 1 and 2
              2  3  4  5  7  -
 Fifth Pass:  2  3  4  5  7

  time complexity :
     best :  O(n)
     worst:  O(n^2)
     avg : O(n^2)
```

Insertion Sort:

```
 Ex:  12  31  25   8  32  17

     12                  - consider first element is already sorted
     12  31
     12  25  31
      8  12  25  31
      8  12  25  31  32
      8  12  17  25  31  32
```

Time complexity:

best : O(n)
worst: O(n^2)
avg : O(n^2)

Quick Sort:

- highly efficient sorting algorithm
- larger arrays of elements splitted into smaller array of elements

Ex: 24 9 29 14 19 27

soln:

```
 P
24   9   29   14   19   27
 L                        R

P
24   9   29   14   19   27
L                   R

                    P
19   9   29   14   24   27
     L              R

                    P
19   9   29   14   24   27
          L         R

          P
19   9   24   14   29   27
          L    R

               P
19   9   14   24   29   27
              L R
```

best : worst : avg : O(nlogn)

Searching:

- Linear Search
- Binary Search

Linear Search:

* Simple Searching algorithm
* Sequential Search
* Traverse the entire list to check whether the search element is present or not
* if the element is present then it will return the position

```
Ex:   34  23  56  12  78  39  46  51  82  20

    Key : 46

           34==46
           23==46
           56==46
            ....
           46==46

      time complexity:
          best case : O(1)
          worst case: O(n)
          avg case  : O(n)

      space complexity:   O(1)
```

Binary Search:

* binary search can be applied for only sorted elements

```
          13  24   28  31  40   43   52  59  62  73
         s=0              mid=4                  e=9

           mid=(s+e)/2
              =(0+9)/2=4

         key=62

         int binarySearch(int a[],int s, int e, int key){
               mid=(s+e)/2;
               if(a[mid]==key){
                   return mid+1;
               }else if(a[mid]<key){
                   s=mid+1;
                   binarySearch(a[],s,e,key);
               } else{
                   e=mid-1;
                   binarySearch(a[],s,e,key);
               }
          }

          time complexity:
             best  : O(1)
             worst : O(logn)
```

avg   : O(logn)
space : O(1)

Divide and Conquer:

- **divide :** break the given problem into small sub problems
- **conquer :** finding the solution for subproblem and integrate it

Ex: Merge Sort, Binary Search, Quick Sort

```
          23 45 27 33 21  59 62 41
     23  45  27  33         21  59  52  41
 23  45      27  33      21  59       52  41
23   45    27     33    21     59     52      41
  23, 45     27,33       21,59        41, 52
       23,27,33,45           21,41,52,59
             21,23,27,33,41,45,52,59
```

Greedy Approach:

- select the possible option at a given time
- decision is taken based on current available information without considering the future

Applications:

- minimum spanning tree
- scheduling algorithm (Deadline First Scheduling)