The linked list class is almost
the ArrayList.

```
import java.util.LinkedList;
public class Main {
    public static void main(String[] args) {
    LinkedList<String> cars = new LinkedList
    cars.add("volvo");
    cars.add("BMW");
    cars.add("Ford");
    Cars.add("Mazda");
    System.out.println(cars);
    }
}
```

→ Linked List class is a collection which can contain many objects of the same type, just like the Array List.
→ The Linked List class has all of the same metho as the ArrayList class because they both implement the List interface. This means that you can add items, change items, remove items and clear the list in the same way.
→ The ArrayList class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new larger array is created to replace the old one and old one gets removed.

How LinkedList Works:-

The LinkedList stores its items in "containers" The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list the element is placed into a new container

data, LinkedList to manipulate data.

LinkedList Methods:-

For many cases, the ArrayList is more efficient as it is common to need access to random items in the list, but the LinkedList provides several methods to do certain operations more efficiently.

addFirst():-

```
import java.util.LinkedList;
    public class Main{
        public static void main(String[] args){
            LinkedList<String> cars = new
            LinkedList<String>();
            cars.add("volvo");
            cars.add("BMW");
            Cars.add("Ford");
        Cars.addFirst("Mazda");
        System.out.println(cars);
        }
    }
```

addLast:-

```
import java.util.LinkedList;
    public class main{
public static void main(String[] args){
LinkedList<String> cars = new LinkedList<String>();
    cars.add("volvo");
    cars.add("BMW");
    cars.add("Ford");
    Cars.addLast("Mazda");
    System.out.println(cars);
    }
    }
```

→ removeLast():-
→ getFirst():-
→ getLast():-

HashMap:-

HashMap however, store items in key/value pairs and you can access them by an index of another type (e.g a string). one object is used as a key (index) to another object (value). It can store different types:-

String keys and Integer values or the same type like: string keys and string values.

```
import java.util.HashMap;
HashMap<String, String> capitalCities = new HashMap<
        String, String>();
```
↑ creating hashmap

Add items:-

Use put() method

```
Capitalcities.put("England", "London");
```

Access an item:-

```
Capitalcities.get("England)
```

Remove:-

```
CapitalCities.remove("England");
```

clear:-

```
CapitalCities.clear();
```

Size:-

```
CapitalCities.size();
```

Looping:-

Use keyset() method if you only want the keys, and use the values() method if you

A hashset is a collection of items where every item is unique, and it is found in the java.util package.

```
import java.util.HashSet(); → import the hashset class
HashSet<String> cars = new
                    HashSet<String>();
```

Add items:-

```
cars.add("volvo");
```

→ Here if we add same item twice, then we get only once, because it has to set unique.

Item exists:-

```
cars.contains
```

will not main order of insertion

```
.add(13);
.add(45);
.add(76)
```

o/p:-76, 13, 45

remove:-

```
cars.remove("volvo");
```

clear:-

```
cars.clear();
```

size:-

```
cars.size();
```

hasnext()

Java Iterator:- An iterator is an object that can be used to loop through collections, like ArrayList and Hashset. It is called iterator because it is technical term for looping

→ To use an iterator, you must import it from the java.util package.

Iterator method can be used to get an iterator for any collection.

```
import java.util.ArrayList;
import java.util.Iterator;
public class Main {
  public static void main(String[] args) {
    ArrayList<String> cars = new ArrayList<String>();
    cars.add("volvo");
    cars.add("BMW");
    cars.add("Ford");
    Iterator it = al.iterator();
    while (it.hasNext())
    {
      S.O.P(it.next()); }
```

```
    }
}
```

Looping through Collection:-

```
import java.util.ArrayList;
import java.util.Iterator;
public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList
                <String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        Iterator<String> it = cars.iterator();
        while(it.hasNext()) {
        System.out.println(it.next());
    }
}
```

Removing:- Same way removing

Java Wrapper classes:-

Wrapper classes provide a way to use primitive data types (int, boolean, etc) as objects.

primitive Data Type	Wrapper class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

objects, such as ArrayList, where primitive data types cannot be used (the list only store objects)

```
ArrayList<int> myNumbers = new ArrayList<int>
    ();// Invalid

ArrayList<Integer> myNumbers = new ArrayList<Integer
    >(); valid
```

Creating Wrapping objects:

```
public class main{
  public static void main(String[] args)
  Integer myInt = 5;
  Double myDouble = 9.99;
  Character myChar = 'A';
  System.out.println(myInt);
  system.out.println(myDouble);
  system.out.println(myChar);
  }
```

→ you can use certain methods to get information about the specific object.

→ for example the folbwing methods are used to get the value associated with the corresponding wrapper object.

intValue(), byteValue(),
shortValue(), longValue(),
floatValue(), doubleValue(),
charValue(), booleanValue()

```
public class main{
  public static void main(String[] args){
    Integer myInt = 5;
    Double myDouble = 5.99;
    Character myChar = 'A';
  System.out.println(myInt.intValue());
  System.out.println(myDouble.doubleValue());
```

contains:- It is used to check element present in Linked List or not.

peek():- It is used to return the first element

poll():- It is used to return and remove the element at the front end of the container.

```
import java.util.LinkedList;
Public class Collectionsdemo {
    public static void main(String[] args) {
        LinkedList ll = new LinkedList();
        ll.add(11);
        ll.add("messi");
        ll.add(3.14);
        ll.add(99);
        ll.add(true);                    => 11, messi, 3.14, 99, true.
        S.o.p(ll);
        S.o.p(ll.contain(3.14));
        S.o.p(ll.contains("ronaldo"));
        S.o.p("peek:" + ll.peek());
        S.o.p(ll);  → 11, messi, 3.14, 99, true
        S.o.p("pool:" + ll.pool());
        S.o.p(ll);   messi, 3.14, 99, true.
    }
}
```

→ It internally implements a double-ended queue, means it will be ready to add elements at either end.

→ We can efficiently add the elements at the front and back end by using ArrayDeque.

```
ArrayDeque ad = new ArrayDeque();
ad.add(10);
ad.add("virat");
ad.add(false);
ad.add(5.55);
ad.addFirst("first");
ad.addLast("Last");
```

```
import java.util.ArrayDeque;
import java.util.Iterator;
ArrayDeque<Integer> arrayDeque = new ArrayDeque<>();
arrayDeque.addFirst(10);
arrayDeque.addLast(20);
arrayDeque.offerFirst(5);
arrayDeque.offerLast(25);
S.o.p("First element: " + arrayDeque.getFirst());
S.o.p(arrayDeque.getLast());
S.o.p(arrayDeque.removeFirst());
S.o.p(arrayDeque.removeLast());
S.o.p(arrayDeque.pop());
S.o.p(arrayDeque.size());
S.o.p(arrayDeque.isEmpty());
arrayDeque.add(50);
arrayDeque.add(60);
S.o.p("Array Elements:");
for(Integer num: arrayDeque){
    S.o.p(num); }
S.o.p("Array in reverse order:");
Iterator<Integer> desc Iterator = arrayDeque.
        descendingIterator();
while(desc Iterator.hasNext()) {
```

It will returns an iterator to traverse from back to front.

priority queue is used whenever we want the priority or the smallest element at the top in such cases priority queue is used.

→ whenever number passed to the priorityqueue it will bring smallest numbers at top.

→ whenever strings are passed to the priorityqueue it will choose element based on dictionary format

```
import java.util.*;
public class CollectionsDemo{
    public static void main(String[] args){
        priorityqueue pq = new priorityqueue();
        pq.add(50);
        pq.add(40);
        pq.add(20);
        pq.add(10);
        pq.add(30);
        S.o.p(pq);
    }
}
```

→ priorityqueue class implements the queue interface. It elements or objects which are to be processed by their priorities.

→ priority queue doesn't allow allow null values to be stored in the queue.

```
import java.util.*;
public class TestJavaCollections{
public static void main(String[] arg){
PriorityQueue<String> queue = new priorityQueue<String>();
 queue.add("vijay sharma");
queue.add("Amit sharma");
queue.add("Jaishankar");
queue.add("Raj");
System.out.println("head"+queue.element());
S.o.p("head:"+queue.peek());
S.o.p("iterating the queue elements");
Iterator itr = queue.iterator();
while(itr.hasNext()){
S.o.p(itr.next());
}
queue.remove();
queue.poll();
S.o.p("after removing two elements");
Iterator<String> itr2 = queue.iterator();
while(itr2.hasNext()){
S.o.p(itr2.next());
}
}
}
```

will arrange all the elements of collection based class in the ascending order or according to the dictionary format.

```
import java.util.*
public class collectionsDemo{
    public static void main (string[] args){
    Treeset ngt=new Treeset();
    ngt.add(44)) => 11, 22, 33, 44, 55
    ngt.add(22);
    ngt.add(11);      .add ("virat");
    ngt.add(55);      .add ("dhoni");
    ngt.add(33);      .add ("Pant");
    s.o.p(ngt);       .add ("bumrah")
                      => bumarh, dhoni, hardik,
                         Pant, virat}
}
```

→ It uses tree for storage. Like hashset, treeset also contains unique elements. However, the access and retrieval time of Treeset is quite fast.

→ The elements in Treeset stored in ascending order.

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String[] args){
Treeset<String> set=new Treeset<String>();
Set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
```

→ Ajay, Ravi, Vijay.

```
import java.util.*
public class LHSDemo{
    public static void main(String[] args){
        LinkedHashSet lhs=new LinkedHashSet();
        lhs.add(13);
        lhs.add(45);
        lhs.add(76);
        S.O.P(lhs); → [13,45,76]
    }
}
```

→ The concrete classes ArrayList, Vector and LinkedList work in similar ways but have different performance characteristics.

→ using list interface instead of concrete class reference allows you to later switch a different concrete class to get better performance.

```
import java.util.*;
public class Test{
    public static void main(String[]args){
        List<String> nameList=new ArrayList<String>();
        String[] names={"Ann","Bob","Carol"};
        for(int k=0;k<names.length;k++)
            nameList.add(names[k]); → add to arraylist
        for(int k=0;k<namesList.size();k++){
            S.O.P(nameList.get(k)); → displaying
        }
    }
}
```

LinkedList:-

```
import java.util.*;
public class Test{
    public static void main(String[] args){
        List<String>nameList=new LinkedList<String>();
        String[] names={"Ann","Bob","Carol"};
        for(int k=0;k<names.length;k++)
            namesList.add(names[k]);
        for(int k=0;k<namesList.size();k++)
```

addAll() will add all the elements in one collection to another collection class.

```
import java.util.*;
public class HSDemo{
    public static void main(String[] args){
        HashSet cricketers = new HashSet();
        cricketers.add("sachin");
        Cricketers.add("doni")
        S.O.P("cricketers" + Cricketers);
        Hashset footballers = new hashset();
        football.add("messi");
        S.O.P("football players:" + players);
        players.removeAll(footballers)
        S.O.P("players:" + players);
    }
}
```

ArrayList:- It preserve the order of insertion.
It allow duplicate element.

ArrayDeque:- It preserve the order of insertion
It allow duplicate element

LinkedList:- It allows duplicate element
It prese:e the order of insertion.

Priorityqueue:- It does not preserve the order of insertion. It allow duplicate element

Treeset:- It does not preserve the order by insertion
It does not allow duplicate element.

Hashset:- It does not allow duplicate element
It does not preserve the order of insertion.

Linkedhashset:-
It preserve the order of insertion
It does not allow duplicate element.

maps → Abstract map → Hashmap, Treema
↳ Sorted map → Navigate Map
↳ ~~Hash map~~

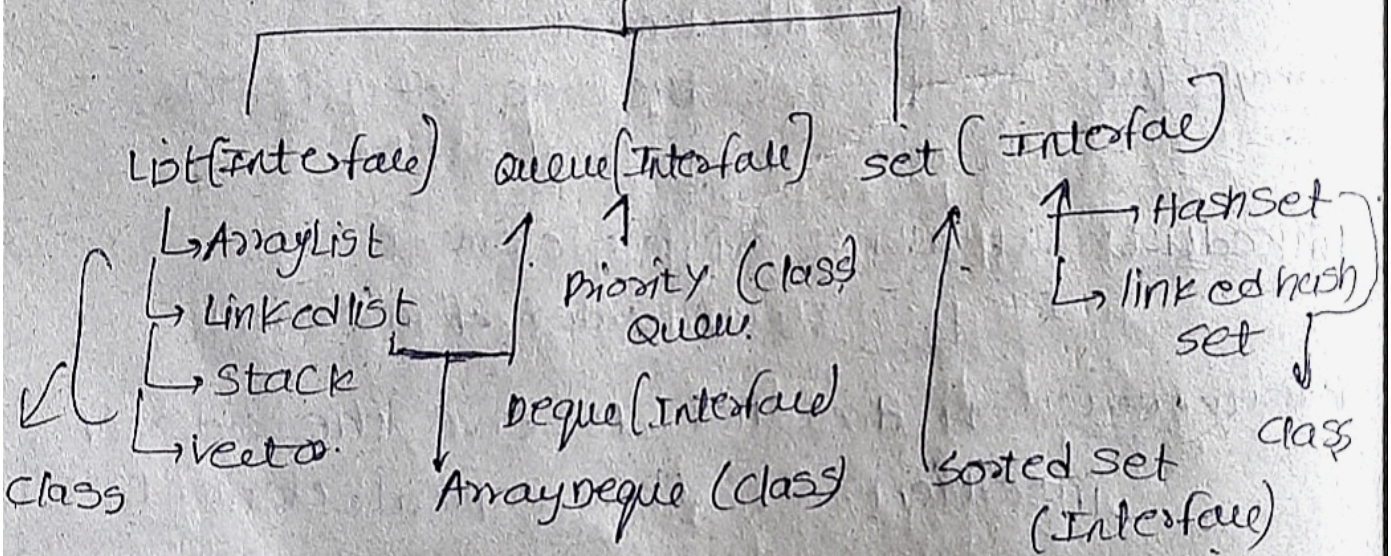
Treemap:-
Stores the values as key value-pair.
sorts the values based upon the keys.

Hashmap:-
Internally uses the hashing algorithm.
The order to insertion is not preserved

Linked Hashmap:-
The order of insertion is preserved.

Tree Map:-

→ Java util contains classes and interfaces



ArrayList:-
→ It uses dynamic array to store the duplicate element of different data types.
→ Contains or maintains the insertion order and is non synchronized.
→ elements can be randomly accessed.

```
import java.util.*;
class TestJavaCollection2
```

Creating Arraylist.

```
list.add("Ajay");
Iterator it = list.iterator();
while(it.hasNext()){
  S.O.P(it.next());
}
}
```

Vijay
Ravi
Ajay

Linked List:-

→ Implements collection interface.
→ It uses doubly linked list internally to store the elements.
→ It can store duplicate elements.
→ It maintains the insertion order and not synchronized.
→ In linked list, the manipulation is fast because no shifting required.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String[] args){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next()); } }}
```

→ Ravi, vijay, Ravi, Ajay.

Vector:- Vector uses a dynamic array to store the data elements. It is similar to ArrayList. It is synchronized and contains many methods that are not part of collection framework.

```
import java.util.*;
public class TestJavaCollection3{
public static void main(String[] args){
Vector<String> v=new Vector<String>();
v.add("Ayush");
v.add("Ashish");
v.add("Gaurav");
Iterator<String> itr=v.iterator();
while(itr.hasNext()){
```

⇒ Ayush, Amit, Ashish, Gaurav.

```
public class MapsDemo {
public static void main(String[] args) {
 TreeMap map = new TreeMap();
map.put(771, "deep");
map.put(171, "kushal");
map.put(501, "saurav");
map.put(873, "kushal");
S.O.P(map);
}
}
```

=> 111 = ayush
171 = kushal
501 = saurav
771 = deep
873 = kushal

LinkedHashmap:-

```
public class MapsDemo {
 public static void main(String[] args) {
  LinkedHashMap map = new LinkedHashMap();
map.put(771, "deep");
map.put(171, "kushal");
map.put(501, "saurav");
map.put(873, "ayush");
S.O.P(map);
```

Variable hiding:-

A variable which have same name in parent class child class. when we call variable parent class variable is hidden

```
class A {
public int x = 10;
}
```

```
main() {
 B b1 = new B();
 S.O.println(b1.x);
```

overriding:-

```
public class mainclass {
 public int y = 20;
 public static int z = 30;
 public static void main(String[] args) {
  A a1 = new A();
  S.O.println(a1.x);
  S.O.println(mainclass.z);
mainclass m = new mainclass();
```

→ static member accessing non static member in a static method. so we create object

→ these

Using iterator:

→ Call the iterator to retrieve an iterator object.
→ Use the hasnext() (boolean) if there is still element to be returned and the next() method to return the next available element.

```
public static void main(String[] args) {
    List<String> nameList = new ArrayList<String>();
    String[] names = {"Ann", "Bob", "Carol"};
    ListIterator<String> it = nameList.listIterator();
    for(int k=0; k<names.length; k++) {
        it.add(names[k]);  → Add to arraylist
    it = nameList.listIterator();
    while(it.hasNext())
        S.o.P(it.next()); }
```

→ The enhanced for loop can be used with any Collection, the compiler converts the enhanced for loop into a traditional loop that uses the collection's iterator.

Hashset:

→ It stores elements according to a hash code.
→ The procedure used to compute the hash code of an element is called the hashing function or the hashing algorithm.

of the object (or its absolute value).
For character objects you can use the Unicode value for the character.
for string objects, you can use a function that takes into account the unicode values of characters that makes up string,
→ Avery simple (but not very good) hashing function for strings might assign to each string the unicode value of its first charader.
→ Note all the strings with same hashcode are assigned the same hash code.
→ when two distinct objects have the same hash code we say that we have a collision.

Implementation of hashset:
hashset is a collection of buckets, and each bucket is a collection of elements.
the collection of buckets is actually list of buckets or ArrayList.
→ Hashset regarded as collection of buckets. Each bucket corresponds to a hash code and stores all objects in the set that have that particular hash code.
→ some buckets will have just one element, whereas other buckets may have many.
→ A good hashing scheme should distribute elements among the buckets so that all buckets have approximately the same number of elements.

How a hashset work:
→ to add an element x, the hash code for x is used (as an index) to locate the appropriate bucket. x is then added to the list for that bucket. If x is already in the bucket (The test is done using the equals method) then it is not added.

Hashset performance considerations:
→ have enough buckets, fewer bucket means more collisions.
→ The load factor of a hashset is the fraction of buckets that must be occupied before the number of buckets is increased.
→ The number of buckets in a hashset is called its

apacity

→ HashSet() Creates an empty hashset object with a default initial capacity of 16 and load factor of 0.75.

→ Creates an empty hashset object with the specified initial capacity and load factor.

→ Creates an empty hashset object with the specified initial capacity and a load factor of 0.75.

```
import java.util.*;
public class HashSetDemo{
    public static void main (String[] args){
        Set<String> fruitSet = new HashSet<String>();
        fruitSet.add("apple");
        fruitSet.add("Banana");
        fruitSet.add("Pear");
        fruitSet.add("Strawberry");
        System.out.println("Here are the elements");
        for (String element : fruitSet)
            System.out.println(element);
        if (!fruitSet.add("Banana"))
            System.out.println("Trying to add Banana Again, the set
                                already");
```

the last-in-first-out data structure, i.e stack
→ The stack contains all of the methods of vector class and also provides its methods like boolean push(), boolean peek(), boolean push(Object o), which defines its properties.

```
import java.util.*;
Public class TestJava Collections {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        Stack.push("Ayush");
        stack.push("Gavit");
        stack.push("Amit");
        Stack.push("Ashish");
        Stack.pop();
        Iterator<String> itr = stack.iterator();
        While(itr.hasNext()) {
            S.O.P(itr.next()); } } }
```

⇒ Ayush, Gavit, Amit, Ashish

Queue Interface:

→ First-in-first-out.
→ It can be defined as an ordered list that is used to hold the elements which are about to be processed.
→ There are various classes like priorityQueue, peque, and ArrayDeque which implements the queue interface.

```
Queue<String> q1 = new priorityQueue();
Queue<String> q2 = new ArrayDeque();
```

operations → Adding → offer(e e) → Adds an element to the queue, Returns true if succesful, otherwise false.
→ add(E e) → Adds an element to the queue, throws an exception if the operation fails

Removing → poll() → Retrieves and removes the head of queue or returns null if the queue is empty.
→ remove()

Inspecting → peek() → retrives but does not remove the head of queue, or returns null if the
→ element() → retrives and removes the head of the queue. throws an exception if the queue is empty

element → Retrieves but does not remove the head throws an exception if the queue is empty.

Difference b/w offer and add?

offer() -

→ Returns true if the element was succesfully added to the queue. [Both will do like this only].

→ Returns false [.offer] if element could not be added.

→ whereas add throws an illegal state exception if the element cannot be added due to capacity restrictions.

```
import java.util.ArrayList;
import java.util.Collections;
public class ArrayListExample {
    public static void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();
    list.add("Apple");
    list.add("Banana");
    list.add("After adding elements: " + list);
    list.add(1, "Grapes");  → Here it will add this element at specified function, and size also increase.
    S.O.P("After adding 'Grapes' at index 1: " + list);
    String fruit = list.get(2);
    S.O.P("Element at index 2: " + fruit);
    list.set(2, "Mango");  → Here we are modifying, size not changed.
    S.O.P("After removing element at index 1: " + list);
    list.remove("Banana");
    S.O.P("After removing 'Banana': " + list);
    boolean hasApple = list.contains("Apple");
    S.O.P("List contains 'Apple': " + hasApple);
    int size = list.size();
    S.O.P("Size of the list: " + size);
    S.O.P("Iterating over the list:");
    for(String item: list) {
        S.O.P(item); }
    Collection.sort(list);
    list.clear();
    S.O.P(list);
    boolean isEmpty = list.isEmpty();
    S.O.P(isEmpty);
}
}
```

List methods:-

```
import java.util.ArrayList;
import java.util.List;
public class ListMethodsExample {
    public static void main(String[] args) {
    List<String> list1 = new ArrayList<>();
    list1.add("Apple");
```

```
list2.add("Banana");
list2.add("Grapes");
list2.add("Mango");
list1.addAll(list2);
S.O.P(list1);
list1.removeAll(list2);
S.O.P(list1);
list1.addAll(list2);
list1.retainAll(list2);
S.O.P(list1);
```

→ only the elements in list1 that are contained in list2

```
List<String> sublist = list1.subList(1,3);
```

Returns a list from specified position

```
int firstIndex = list.indexOf("banana");
S.O.P(firstIndex);
int lastIndex = list1.lastIndexOf("banana");
S.O.P(lastIndex);
boolean containsAll = list1.containsAll(list2);
S.O.P(containsAll);
boolean containsApple = list1.contains("Apple");
S.O.P(containsApple);
```

LinkedList:-

```
import java.util.LinkedList;
public class LinkedListExample {
    public static void main(String[] args) {
        LinkedList<String> list = new LinkedList();
        list.add("Apple");
        list.add("Banana");
        list.add("cherry");
        S.O.P(list);
        list.add(1, "orange");
        S.O.P(list);
```

```
S.o.p( list.getFirst());
S.o.p( list.getLast());
S.o.p (list.get(2));
list.remove();
list.remove(2);
list.removeFirst();
list.removeLast();
list.add ("Apple");
list.add ("Banana");
list.add ("cherry");
S.o.p ("List: " + list);
S.o.p ( list.indexOf ("cherry"));
S.o.p ( list.lastIndexOf ("cherry"));
S.o.p ( list.peek());
S.o.p ( list.poll());
S.o.p ( list);
list.offer ("Grapes");
S.o.p(list);
list.clear();
S.o.p(list);
}
```

→ retrieves the first element in list. If the list is empty it returns null. without removing.

It retrieves and removes the first element of list. returns null if it is empty.

Adds an element to the end of the list. Returns true upon success.

Vector list:-

```
import java.util.Vector;
public class VectorExample {
  public static void main(String[] args) {
    Vector<String> vector = new Vector<>();
    vector.add("Apple");
    vector.add("Banana");
    vector.add("Cherry");
    vector.add(1, "orange");
    Vector<String> moreFruits = new Vector<>();
    moreFruits.add("Grapes");
    moreFruits.add("mango");
    vector.addAll(moreFruits);
    S.o.p(vector);
    S.o.p(vector.get(2));
```

```
vector.set(3, "Blueberry");
S.O.P(After set S.O.P(vector);
vector.remove(3);
S.O.P(vector);
vector.remove("orange");
S.O.P(vector);
S.O.P(vector.size());
S.O.P(vector.capacity());
vector.clear();
```

Stack:- It extends vector with method allow a vector to be treated as stack.

```
import java.util.Stack;
Public class StackExample {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("Apple");
        stack.push("Banana");
        stack.push("cherry");
        S.O.P("stack after pushing elements:"
              +stack);
        S.O.P("stack.peek()); → it will retrieve top element without removing it
        S.O.P(stack.pop()); → it will retrives top element and removes it.
        S.O.P(stack.isEmpty());
        S.O.P(stack.search("Apple"));
```

top element ←

Set:-

```
import java.util.HashSet;
import java.util.Set;
Public class SetExample {
    public static void main(String[] args) {
        Set<String> myset = new HashSet<>();
        myset.add("Apple");
        myset.add("Banana");
        myset.add("Orange");
        S.O.P(myset);
        S.O.P(myset.remove("Banana"));
        boolean hasApple = myset.contains("Apple");
```

```
myset.clear();
S.o.P(myset);
boolean isEmpty = myset.isEmpty();
S.o.P(isempty);
```

→ Addall
→ removeall
→ retainAll

Hashset:
→ add
→ contains
→ remove
→ size
→ clear
→ isEmpty.

sortedset:
→ add(element);
→ remove("orange")
→ first()
→ last();
→ contains
→ size
→ clear()
→ isempty()

LinkedHashset:

Navigable set: It is a subinterface of sorted set that provides methods for navigating through the elements in a set. It extends the functionality of sorted set by adding methods that allow for easier retrieval of the closed matches for given search targets as well as methods to return subsets of the set.

→ The primary implementation of navigable set is Treeset.

```
import java.util.Navigableset
import java.util.Treeset;
public class NavigableSetExample {
    public static void main(String[] args) {
        Navigableset<Integer> numbers =
            new Treeset<>();
        numbers.add(10);
        numbers.add(20);
        numbers.add(30);
        numbers.add(40);
        numbers.add(50);
        S.o.P(numbers);
        S.o.P(numbers.lower(30));
        S.o.P(numbers.floor(30));
```

→ This creates a new treeset to store integer elements while maintaing their sorted order.

```
S.O.P(numbers.firstC));
S.O.P(numbers.lastC));
NavigableSet<Integer> subset = numbers.
    subset(20, true, 40, true);
numbers.remove(20);
S.O.P(numbers)
```

→ lower(element) → it will returns the greatest element less than e.

[10, 20, 30, 40, 50]

lower(30) → 20 → Returns the greatest element less than or equal to e.

floor(30) → 30

Ceil(30) → 30 → Returns the least element greater than or equal to e.

higher(30) → 40 → Returns the least element greater than e.

Queue:-

→ poll and remove

→ Both retrives and removes the head (first element of the queue. → poll

→ Returns null if the queue is empty, throws a NoSuch Element Exception if the queue is empty.

↓

remove

→ Both → peek and element will retrieves head of the queue. Returns null[peek] if the queue is empty.

→ ~~Ref~~ throws a NoSuchElementException, if the queue is empty

priorityqueue:-

```
import java.util.priorityQueue;
import java.util.Iterator;
priorityQueue<Integer> pq=new priorityQueue<>();
  pq.add(30);
  pq.add(10);
  pq.offer(20);
  S.O.P("Head using peek(): "+pq.peek());
  S.O.P(pq.element());
  S.O.P(pq.poll());
  S.O.P(pq.remove());
```

```
S.o.p(pq.isEmpty());
S.o.p(pq.contains(40));
pq.add(50);
pq.add(60);
Iterator<Integer> iterator = pq.iterator();
while(iterator.hasNext()) {
    S.o.p(iterator.next());
}
}
}
```

Deque: The deque interface is implemented by classes like ArrayDeque and LinkedList.

```
import java.util.Deque;
import java.util.ArrayDeque;
import java.util.Iterator;
    Deque<Integer> deque = new ArrayDeque<>();
    deque.addFirst(10);
    deque.addLast(20);
    deque.offerFirst(5);
    deque.offerLast(25);
    S.o.p(deque.getFirst());
    S.o.p(deque.getLast());
    S.o.p(deque.removeFirst());
    S.o.p(deque.removeLast());
    S.o.p(deque.pop());
    S.o.p(deque.size());
    S.o.p(deque.isEmpty());
    Iterator<Integer> desCIterator = deque.desending
        Iterator();
    while(desCIterator.hasNext())
        S.o.p(desCIterator.next())
}
}
}
```

→ Each key maps to exactly one value, and keys are not allowed.

→ The map interface is implemented by:-

- Hashmap
- Linkedhashmap
- Treemap
- Hashtable.

Hashmap uses hashtable for storage. It allows null keys and values, doesn't gurantee order and generally fast for insertion and retrieval.

→ Linkedhashmap extends hashmap and maintains insetion order or access order. Useful when you need to maintain order.

→ Treemap doesn't allow null keys.

→ Hashtable similar to hashmap, but is synchronized It does not allow null keys or values and is generally slower.

```
import java.util.HashMap;
import java.util.Map;
public class MapExample{
    public static void main(String[] args) {
    Map<String, Integer> map = new HashMap<>();
    map.put("APPLE", 10);
    map.put("Banana", 20);
    Map.putIfAbsent("APPLE", 15);
    S.o.P(map.get("APPLE"));
    S.o.P(map.getOrDefault("cherry", 0));
    map.remove("Banana");
    S.o.P(map.containsKey("APPLE"));
    S.o.P(map.containsValue(20));
    S.o.P(map.size());
    S.o.P(map.isEmpty());
    S.o.P("mapentries:");
```

S.o.P(Integer>entry: map.entrySet)

```
});
map.replace("Apple", 12);
map.replace("Apple", 10, 15);
```
only replaces if Apple is mapped to 10.

→ Hashmap, fast allows null keys and values, unordered.
→ LinkedHashMap, maintains insertion/access order, allows null keys and values.
→ Treemap → sorted by natural ordering or comparator, does not allow null keys
→ hashtable → synchronized (thread safe) does not allow null keys or values, slower due to synchronization
→ ~~Returns a~~ entrySet(): → Returns a set view of the key-value pairs (Map.Entry objects) contained in this map.

```
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Set;
import java.util.Collection;
Map<String, Integer> linkedHashMap = new LinkedHash
        Map<>();
linkedMap.put("Apple", 10);
linkedMap.put("Banana", 20);
linkedMap.put("cherry", 30);
Integer appleCount = linkedMap.get("Apple");
~~Syste~~ S.o.P(linkedMap.containsKey("Banana");
S.o.P(linkedMap.containsValue(30);
linkedMap.remove("Banana");
int size = linkedMap.size();
S.o.P(size);
boolean isEmpty = linkedMap.isEmpty();
linkedMap.clear();
linkedMap.put("Apple", 10);
linkedMap.put("Banana", 20);
linkedMap.put("cherry", 30);
Set<String> keys = linkedMap.keySet();
Collection<Integer> values = linkedMap.values();
```

for(Map.Entry<String, entries) {
S.O.P("Key:" + entry.getKey() + "
Value:" + entry.getValue());
}
}
)

→ Allows one null key and multiple null values.
→ maintains insertion order of elements, slower than hashmap for insertion, but provides ordered iteration.
→ Allows one null key and multiple null values.
→ Not synchronized.

TreeMap - Maintains its entries sorted based on the natural ordering of the keys or according to a custom comparator provided at map creation time.
→ Implements the navigable map interface, which provides methods to navigate through the map.
→ Does not allow null keys but allows multiple null values.

```
import java.util.Map;
import java.util.TreeMap;
public class TreeMapExample {
    public static void main(String[] args) {
        TreeMap<String, Integer> treemap = new
            TreeMap<>();
        treemap.put("Apple", 10);
        treemap.put("Banana", 20);
        treemap.put("Cherry", 30);
        S.O.P(treemap.get("Apple"));
        S.O.P(boolean hasBanana = treeMap.
            containsKey("Banana"));
        S.O.P(hasBanana);
        boolean hasValue30 = treeMap.containsValue(30);
        S.O.P(hasValue30);
        treemap.remove("Banana");
```

treeMap.put("Date", 40);
treeMap.put("Elderberry", 50);
Map<String, Integer> headMap = treeMap.headMap
("cherry");
Map<String, Integer> tailMap = treeMap.tailMap
("cherry");
Map<String, Integer> subMap = treeMap.subMap
("Apple", "Date");
S.o.p(treeMap.keySet());
S.o.p(treeMap.values());

}
}

Hashtable:-
It is a legacy class that implements the Map interface and is part of the java.util package.
→ It stores key-value pairs in hash-table, and each key is unique.
→ Hashtable is similar to HashMap, but with some notable differences in terms of synchronization, handling of null values, and performance.
→ Thread-safe[synchronized]
→ No null keys or value, if we try to add it will get error like 'nullpointerexception.
→ legacy Although still in use
→ like hashmap it uses hashing to store data.

```
Hashtable<String, Integer> hashtable = new
                    hashtable<>();
hashtable.put("Apple", 10);
hashtable.put("Banana", 20);
hashtable.put("cherry", 30);
Integer appleCount = hashtable.get("Apple")
```

→ containsKey → keySet();
→ containsValue → values();
→ remove(element)
→ size()
→ isEmpty()