**Collections in java**

**ArrayList**

ArrayList<Integer> l = new ArrayList<Integer>();

* Java ArrayList class can contain duplicate elements.
* Java ArrayList class maintains insertion order.
* Java ArrayList class is non synchronized.
* Java ArrayList allows random access because the array works on an index basis.
* In ArrayList, manipulation is a little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

methods.

1.add(int data) : appends the elements.

2.add(int data,int index) : add the elements at specified index.

3.indexOf(int index) : to get the first occurence of the element in the list.

If not found then return -1.

4.remove(int data) : to remove the first occurence of the element in the list.

5.get(int index) : to get the element from the arraylist at index i.

6.clear() : it clears the whole arraylist.

7.contains(int data) : checks whether the element is present in the list or not.

8.size() : returns the current size of the arraylist.

9.isEmpty() : checks whether the list is empty or not.

10.set(int index,int element) : overrides the data at specified index

with new element.

11.sort() : sorts the arraylist in ascending order.

12.addAll(int index,Collection c) : inserts the collection c into the array list

at specified index.

**LinkedList**

LinkedList<Integer> ll = new LinkedList<Integer>();

* Java LinkedList class can contain duplicate elements.
* Java LinkedList class maintains insertion order.
* Java LinkedList class is non synchronized.
* In Java LinkedList class, manipulation is fast because no shifting needs to occur.
* Java LinkedList class can be used as a list, stack or queue.

1.add(int data) : appends the elements.

2.add(int data,int index) : add the elements at specified index.

3.addAll(Collection c) : It is used to append all of the elements in the specified

collection to the end of this list.

4.addFirst(int ele) : adds the elements at the beginning of the linked list.

5.addLast(int ele) : adds the elements at the end of the linked list.

6.clear() : removes all the elements in the linked list.

7.contains(Object obj) : checks whether the object is present in linked list or not.

8.get(int index) : gets the element at the specified index poistion.

9.getFirst() : it returns the first element in the linked list.

10.getLast() : it returns the last element in the linked list.

11.indexOf(int index) : it returns the element present at specified index.

12.poll() : it returns and removes the first element in the linked list.

13.pollFirst(): it returns and removes the first element of the linked list.

14.pollLast() : it returns and removes the last element of the linked list.

15.remove(Object obj) : it removes the first occurence of the object in linked list.

16.set(int index,Object obj) : it overrides the element at the specified index position

with the specified new element.

17.size() : it returns the size of the linked list.

**HashSet**

* HashSet stores the elements by using a mechanism called hashing.
* HashSet contains unique elements only.
* HashSet allows null value.
* HashSet class is non synchronized.
* HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
* HashSet is the best approach for search operations.

HashSet<Integer> h = new HashSet();

1.add(int data) : adds the element into the hashset if it is not present in the set.

2.clear() : it clears all the elements in the set.

3.contains(int ele) : it checks whether the element is present in the hashset or not.

4.remove(int ele) : it removes the specified element from hashset.

5.size() : returns the size of Hashset.

**Java List**

List in Java provides the facility to maintain the ordered collection. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

**Syntax** : List<String> list=new ArrayList<String>();

Methods

1. Add(int index,int ele) : adds the elements into the list at specified index position.
2. Add(int ele) : it appends the elements at the end of the list.
3. Clear() : it removes all the elements in the list.
4. Equals(int ele) : it is used to compare the element with another element.
5. Get(int index) : it is used to fetch the element at specified index position.
6. Contains(int ele) : It checks whether the list contains the given element or not.
7. Remove(int ele) : it is used to remove the first occurrence of the specified element in the list.
8. Sort() : sorts the array in ascending order.
9. Size() : it returns the length or number of elements present in the list.

**Linked HashSet**

Java LinkedHashSet class is a Hashtable and Linked list implementation of the Set interface. It inherits the HashSet class and implements the Set interface.

The important points about the Java LinkedHashSet class are:

1. Java LinkedHashSet class contains unique elements only like HashSet.
2. Java LinkedHashSet class provides all optional set operations and permits null elements.
3. Java LinkedHashSet class is non-synchronized.
4. Java LinkedHashSet class maintains insertion order.

Linked Hashset supports same methods of Hashset.

**Tree Set**

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements the NavigableSet interface. The objects of the TreeSet class are stored in ascending order.

The important points about the Java TreeSet class are:

1. Java TreeSet class contains unique elements only like HashSet.
2. Java TreeSet class access and retrieval times are quiet fast.
3. Java TreeSet class doesn't allow null element.
4. Java TreeSet class is non synchronized.
5. Java TreeSet class maintains ascending order.

Methods in Treeset

1. Add(ele) : it is used to add the specified element into the set if not present.
2. Ceiling(ele) : it returns the closest or greatest number of the specified element in treeset.if not found returns null.
3. DescendingIterator() : it is used to iterate the elements in reverse order.
4. descendingSet() : it returns the set in reverse order.
5. Floor(ele) : it returns the closest or less number of treeset if found otherwise return null.
6. SortedSet headset(E) : it returns the group of elements which are less than specified element.
7. isEmpty() : checks whether the tree set is empty or not.
8. Clear() : removes all the elements of the treeset
9. Remove(ele) : it removes the specified element from tree set.
10. SortedSet tailSet(E) : returns the group of elements which are greater than the specified element.

**Queue**

The interface Queue is available in the java.util package and does extend the Collection interface. It is used to keep the elements that are processed in the First In First Out (FIFO) manner. It is an ordered list of objects, where insertion of elements occurs at the end of the list, and removal of elements occur at the beginning of the list.

Methods

1. Add(ele) : it adds the element into the queue and returns true upon success.
2. Offer(ele) : it is also used to add the element into the queue.
3. Remove() : it returns and removes the head of the queue.
4. Poll() : it returns and removes the head of the queue or return null if the queue is empty.
5. Element() : it retrieves but does not remove the element form the queue.
6. Peek() : it is used to retrieve the first element from the queue but does not remove it.

* FIFO concept is used for insertion and deletion of elements from a queue.
* The Java Queue provides support for all of the methods of the Collection interface including deletion, insertion, etc.
* PriorityQueue, ArrayBlockingQueue and LinkedList are the implementations that are used most frequently.

**Priority Queue**

The PriorityQueue class in Java is a built-in implementation of a priority queue. It is part of the Java Collections framework and provides a convenient way to store elements in a queue based on their priority. Here's an overview of how to use the PriorityQueue.

In Priority queue the elements are stored in sorted order.

The methods of priority queue are similar to that of queue.

Map does not insertion order.

**Deque**

The interface called Deque is present in java.util package. It is the subtype of the interface queue. The Deque supports the addition as well as the removal of elements from both ends of the data structure. Therefore, a deque can be used as a stack or a queue. We know that the stack supports the Last In First Out (LIFO) operation, and the operation First In First Out is supported by a queue. As a deque supports both, either of the mentioned operations can be performed on it. Deque is an acronym for "double ended queue".

Methods

1. Add(ele) : it is used to insert the specified element into deque.
2. Offer(ele) : it is used to insert the specified element into deque
3. Remove() : it retrieves and removes the head of deque.
4. Poll() : it also retrieves and removes the head of the queue,and null if deque is empty.
5. Peek() : it is used to retrieve but not removing the first element from deque.
6. peekFirst() : returns the head of the deque.
7. peekLast() : returns the last element of the deque.
8. offerFirst(e) : insert the element at the front of the deque.
9. offerLast(e) : insert the element at the end of the deque.

**Java Map Interface**

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

A Map is useful if you have to search, update or delete elements on the basis of a key.

A Map doesn't allow duplicate keys, but you can have duplicate values.

Methods of Map

1. put(key,value) : it is used to insert a key,value pair into a map.
2. Put(map) : it is used to insert a map into the current map.
3. putIfAbsent(key,value) : it inserts the entry into the map only if the current entry is not present in the map.
4. Remove(key) : it is used to delete and entry of specified key from the map.
5. keyset() : it returns set view of all the keys from map.
6. Clear() : it removes all the entries from the map.
7. containsValue(value) : returns true if the specified value is present in the map.
8. containsKey(key) : returns true if the specified key is present in the map.
9. Get(key) : it returns the value associated with the key.
10. isEmpty() : checks if the map is empty or not.
11. Values() : it returns a collection of values from the map.
12. Size() : it returns the size of the map.

Map.Entry Interface

It provides methods to get key and value.

1. getKey() : it is used to obtain a key.
2. getValue() : it is used to obtain a value.
3. setValue(value) : it is used to override the value.

**Linked HashMap**

* Java LinkedHashMap contains values based on the key.
* Java LinkedHashMap contains unique elements.
* Java LinkedHashMap may have one null key and multiple null values.
* Java LinkedHashMap is non synchronized.
* Java LinkedHashMap maintains insertion order.

Methods

1. Get(key) : it returns the value associated with key in linked hashmap.
2. Clear() : it removes all the entries in the hashmap.
3. containsValue(int value) : it returns true if the value is mapped to a key in hashmap.

**TreeMap**

It provides an efficient means of storing key-value pairs in sorted order.

* Java TreeMap contains values based on the key. It implements the NavigableMap interface and extends AbstractMap class.
* Java TreeMap contains only unique elements.
* Java TreeMap cannot have a null key but can have multiple null values.
* Java TreeMap is non synchronized.
* Java TreeMap maintains ascending order.

Methods

1. Put(k,v) : it inserts key and value into the tree map.
2. Replace(k,oldvalue,newvalue) : it replaces old value with new value.
3. containsKey(k) : checks whether the given key is present in the hashmap.
4. containsValue(value) : checks whether the givrn value is present or not.
5. firstKey() : returns the lowest key from the sorted map.
6. laskKey() : returns the greated key from the sorted map.
7. tailMap(key) : returns key value pair whose keys are greater than the specified key.
8. subMap(start key,end key) : it returns the key value pairs whose keys are in range of start key inclusive and end keys exclusive.

**Stack**

The stack is a linear data structure that is used to store the collection of objects. It is based on Last-In-First-Out (LIFO). Java collection framework provides many interfaces and classes to store the collection of objects. One of them is the Stack class that provides different operations such as push, pop, search, etc.

Methods

1. empty() : checks whether the stack is empty or not.
2. Push(e) : pushes or appends the element into the stack.
3. Pop() : returns and removes the top element of the stack.
4. Peek() : it returns but not removes the top element of the stack.
5. Search(ele) : the method searches for the specified element in the stack and returns the position of the element.

**Java Collections Class**

Java collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

The important points about Java Collections class are:

* Java Collection class supports the polymorphic algorithms that operate on collections.
* Java Collection class throws a NullPointerException if the collections or class objects provided to them are null.

Methods

1. addAll(collection) : it appends the specified collection to the current collection.
2. binarySearch(ele) : it searched for the element based on binary search algorithm and returns the position of the element in the sorted collection.
3. Copy(dest,src) : It is used to copy a src collection into src collection.Make sure that the two lists have equal number of elements.
4. Fill(collection,element) : it is used to fill the specified collection with the specified element.
5. Frequency(collection,ele) : it is used to find the number of occurences of the specified element in the specified collection.
6. Max(collection) : it is used to find out the maximum element from the collection
7. Min(collection) : it is used to find the minimum element from the collection.
8. Reverse(collection) : it is used to reverse the elements of the specified collection.
9. Rotate(collection,distance) : it is used to rotate the collection of elements by specified distance.
10. Sort(collection) : it is used to sort the elements in a collection in a sorted order.
11. replaceAll(collection,oldval,newval) : it is used to replace all the occurences of the specified element in the collection with a new value.