DONE BY:

Sneha U

Harinder Kumar

Pavan Chitapur

Sujit Pandey

Collection Framework

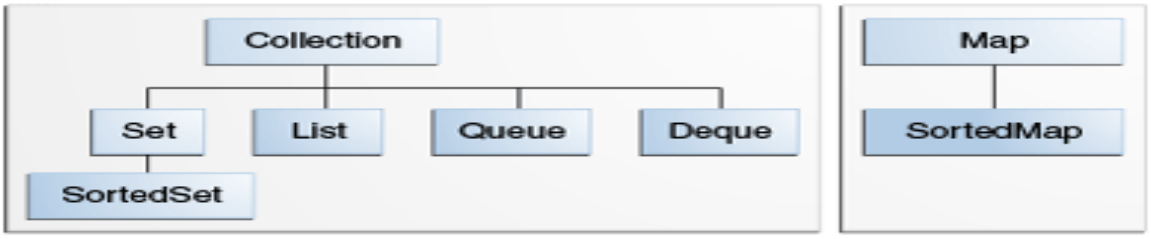


Fig.1 . Collection framework

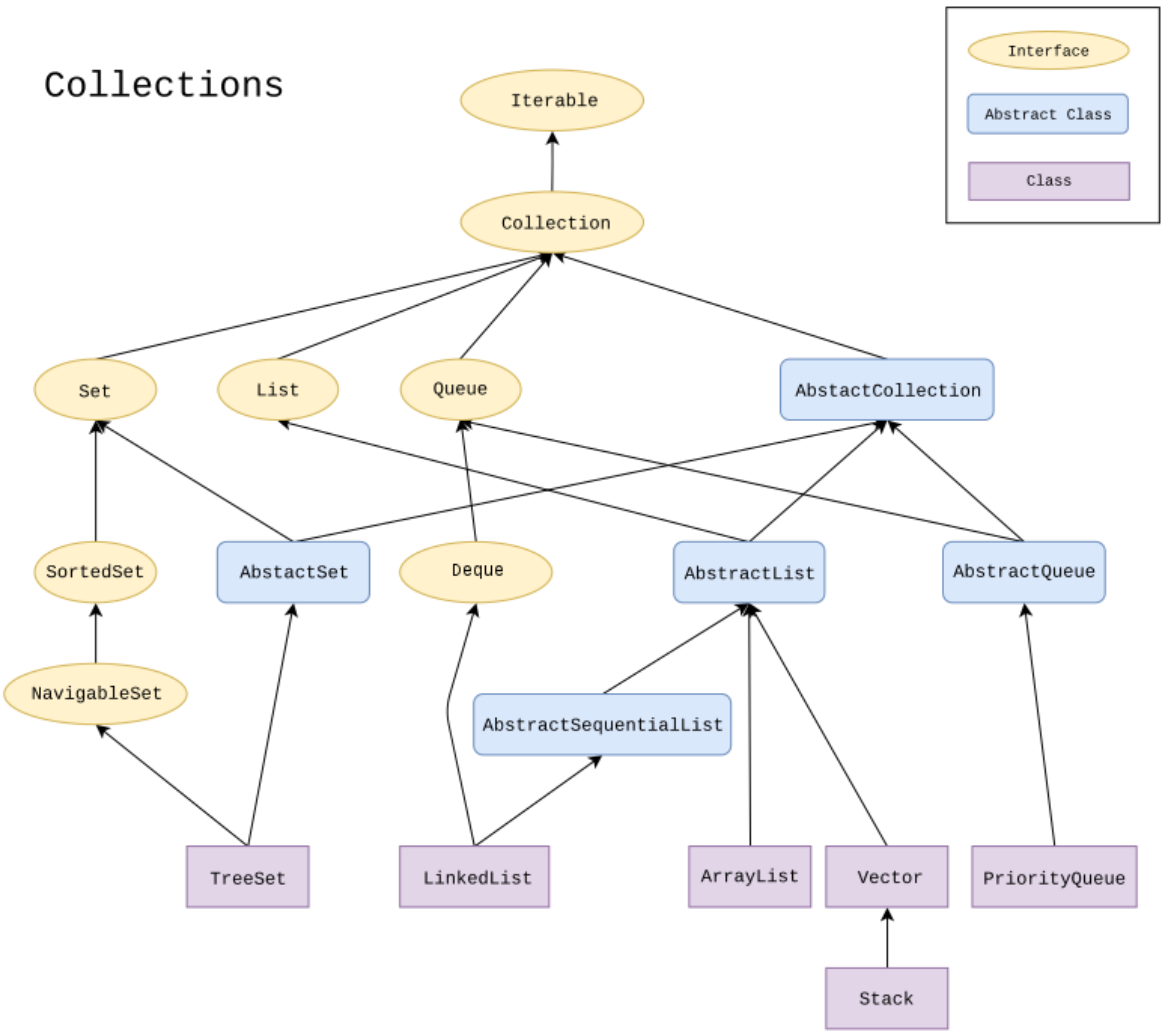


Fig.2 The full collection Framework

The Set Interface:

A Set is a Collection that cannot contain duplicate elements. It models the mathematical set abstraction. The Set interface contains only methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behaviour of the equals and Hash Code operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

Deque

The **java.util.Deque** interface is a subtype of the **java.util.Queue** interface. The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure, it can be used as a queue (first-in-first-out/FIFO) or as a stack (last-in-first-out/LIFO). These are faster than Stack and LinkedList.

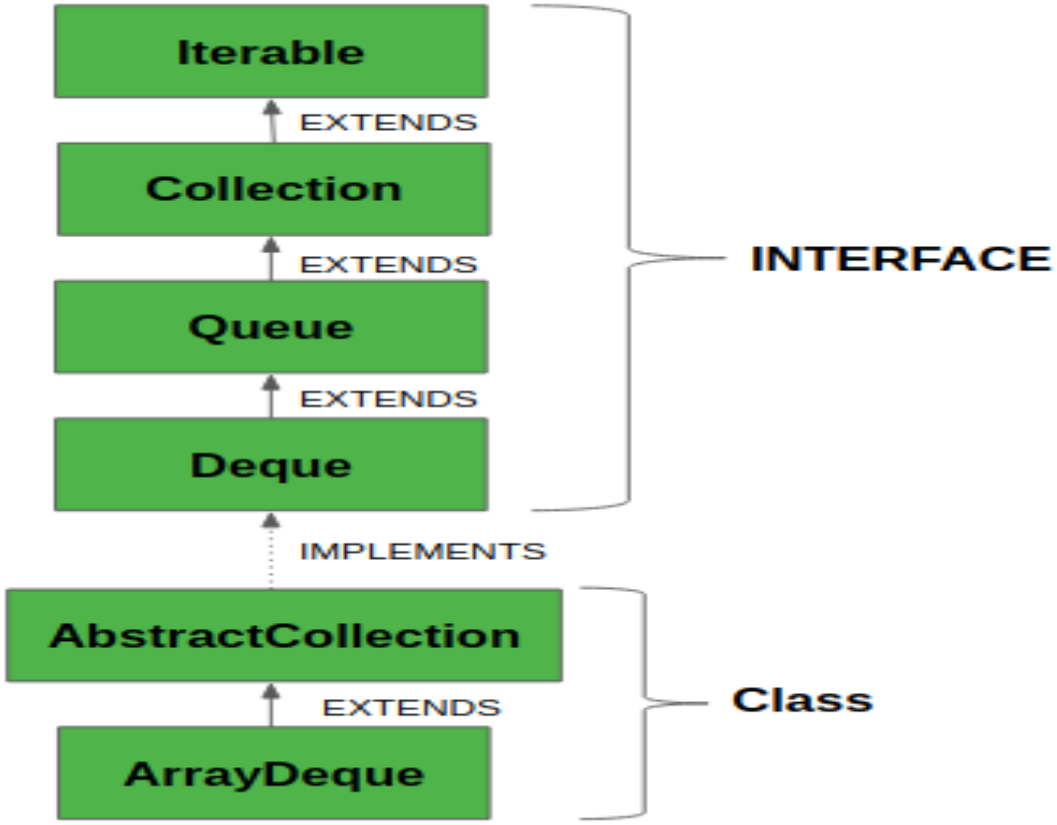


Fig.3 This is the hierarchy of Deque interface in Java.

Important points of Deque

- It provides the support of resizable array and helps in restriction-free capacity, so to grow the array according to the usage.
- Array deques prohibit the use of Null elements and do not accept any such elements.
- Any concurrent access by multiple threads is not supported.
- In the absence of external synchronization, Deque is not **thread safe.**

PROGRAM

```
// Java program to demonstrate working of
// Deque in Java
import java.util.*;

public class DequeExample
{
   public static void main(String[] args)
   {
      Deque<String> deque = new LinkedList<String>();

      // We can add elements to the queue in various ways
      deque.add("Element 1 (Tail)"); // add to tail
      deque.addFirst("Element 2 (Head)");
      deque.addLast("Element 3 (Tail)");
      deque.push("Element 4 (Head)"); //add to head
      deque.offer("Element 5 (Tail)");
      deque.offerFirst("Element 6 (Head)");
      deque.offerLast("Element 7 (Tail)");

      System.out.println(deque + "\n");

      // Iterate through the queue elements.
      System.out.println("Standard Iterator");
      Iterator iterator = deque.iterator();
      while (iterator.hasNext())
         System.out.println("\t" + iterator.next());


      // Reverse order iterator
      Iterator reverse = deque.descendingIterator();
      System.out.println("Reverse Iterator");
      while (reverse.hasNext())
         System.out.println("\t" + reverse.next());

      // Peek returns the head, without deleting
      // it from the deque
      System.out.println("Peek " + deque.peek());
      System.out.println("After peek: " + deque);

      // Pop returns the head, and removes it from
      // the deque
      System.out.println("Pop " + deque.pop());
      System.out.println("After pop: " + deque);

      // We can check if a specific element exists
      // in the deque
      System.out.println("Contains element 3: " +
                  deque.contains("Element 3 (Tail)"));
```

```
        // We can remove the first / last element.
        deque.removeFirst();
        deque.removeLast();
        System.out.println("Deque after removing " +
                        "first and last: " + deque);

    }
}
```

Various Methods of Deque

1. **add(element):** Adds an element to the tail.
2. **addFirst(element):** Adds an element to the head.
3. **addLast(element):** Adds an element to the tail.
4. **offer(element):** Adds an element to the tail and returns a boolean to explain if the insertion was successful.
5. **offerFirst(element):** Adds an element to the head and returns a boolean to explain if the insertion was successful.
6. **offerLast(element):** Adds an element to the tail and returns a boolean to explain if the insertion was successful.
7. **iterator():** Return an iterator for this deque.
8. **descendingIterator():** Returns an iterator that has the reverse order for this deque.
9. **push(element):** Adds an element to the head.
10. **pop(element):** Removes an element from the head and returns it.
11. **removeFirst():** Removes the element at the head.
12. **removeLast():** Removes the element at the tail.
13. **poll():** Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.
14. **pollFirst():** Retrieves and removes the first element of this deque, or returns null if this deque is empty.
15. **pollLast():** Retrieves and removes the last element of this deque, or returns null if this deque is empty.
16. **peek():** Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.
17. **peekFirst():** Retrieves, but does not remove, the first element of this deque, or returns null if this deque is empty.
18. **peekLast():** Retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty.

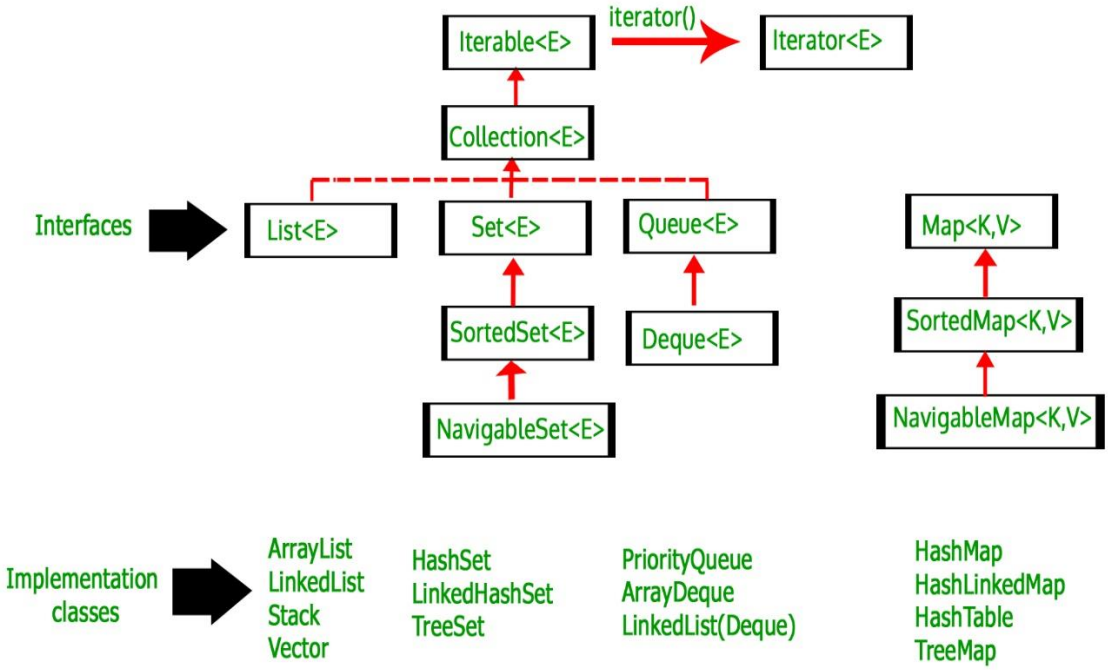
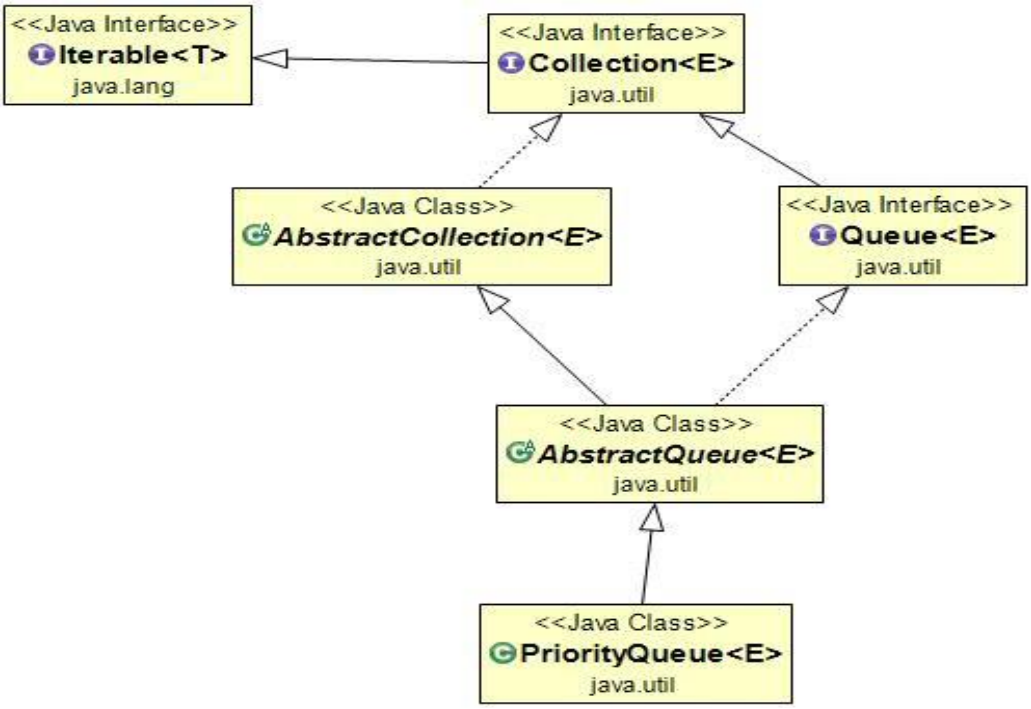
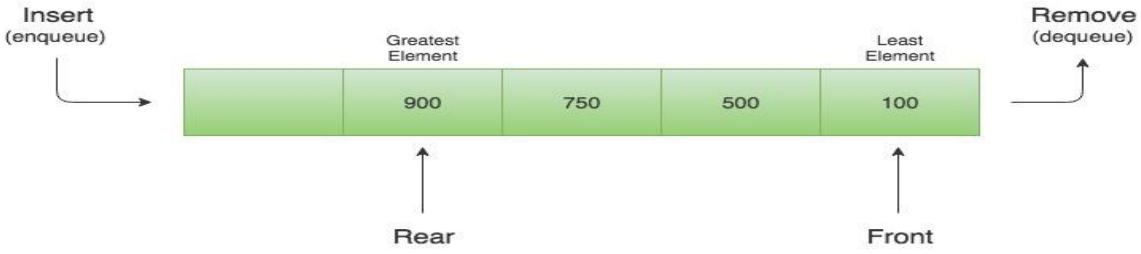


Fig.4Deque in Collection Hierarchy

Priority Queue in Java



Priority Queue Data Structure



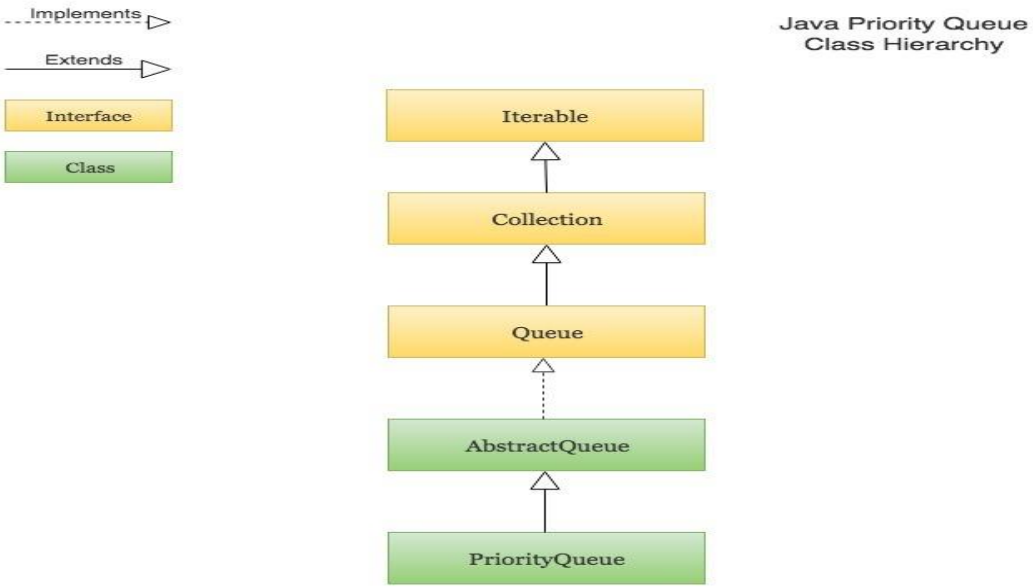


Fig . Priority Queue in Java Collection

- A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority, that's when the PriorityQueue comes into play. The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
- A priority queue in Java is a special type of queue wherein all the elements are **ordered** as per their natural ordering or based on a custom `Comparator` supplied at the time of creation.
- The *front* of the priority queue contains the least element according to the specified ordering, and the *rear* of the priority queue contains the greatest element.

Few important points on Priority Queue are as follows:

- PriorityQueue doesn't permit NULL pointers.
- We can't create PriorityQueue of Objects that are non-comparable
- PriorityQueue are unbound queues.
- The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements — ties are broken arbitrarily.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
- It inherits methods from AbstractQueue, AbstractCollection, Collection and Object class.

PRIORITY QUEUE PROGRAM

```
// Java program to demonstrate working of priority queue in Java
import java.util.*;

class PriorityQueueExample
{
   public static void main(String args[])
   {
      // Creating empty priority queue
      PriorityQueue<String> pQueue =
                  new PriorityQueue<String>();

      // Adding items to the pQueue using add()
      pQueue.add("C");
      pQueue.add("C++");
      pQueue.add("Java");
      pQueue.add("Python");

      // Printing the most priority element
      System.out.println("Head value using peek function:"
```

```
                                + pQueue.peek());

        // Printing all elements
        System.out.println("The queue elements:");
        Iterator itr = pQueue.iterator();
        while (itr.hasNext())
           System.out.println(itr.next());

        // Removing the top priority element (or head) and
        // printing the modified pQueue using poll()
        pQueue.poll();
        System.out.println("After removing an element" +
                        "with poll function:");
        Iterator<String> itr2 = pQueue.iterator();
        while (itr2.hasNext())
           System.out.println(itr2.next());

        // Removing Java using remove()
        pQueue.remove("Java");
        System.out.println("after removing Java with" +
                        " remove function:");
        Iterator<String> itr3 = pQueue.iterator();
        while (itr3.hasNext())
           System.out.println(itr3.next());

        // Check if an element is present using contains()
        boolean b = pQueue.contains("C");
        System.out.println ( "Priority queue contains C " +
                         "or not?: " + b);

        // Getting objects from the queue using toArray()
        // in an array and print the array
        Object[] arr = pQueue.toArray();
        System.out.println ( "Value in array: ");
        for (int i = 0; i<arr.length; i++)
          System.out.println ( "Value: " + arr[i].toString()) ;
    }
}
```

Methods in PriorityQueue class:

1. **boolean add(E element)**: This method inserts the specified element into this priority queue.
2. **public remove():** This method removes a single instance of the specified element from this queue, if it is present
3. **public poll():** This method retrieves and removes the head of this queue, or returns null if this queue is empty.
4. **public peek():** This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
5. **Iterator iterator():** Returns an iterator over the elements in this queue.
6. **boolean contains(Object o):** This method returns true if this queue contains the specified element
7. **void clear():** This method is used to remove all of the contents of the priority queue.
8. **boolean offer(E e):** This method is used to insert a specific element into the priority queue.
9. **int size():** The method is used to return the number of elements present in the set.
10. **toArray():** This method is used to return an array containing all of the elements in this queue.
11. **Comparator comparator():** The method is used to return the comparator that can be used to order the elements of the queue.

NAVIGABLE SET

NavigableSet represents a navigable set in **Java Collection Framework**. The NavigableSet interface inherits from the **Sorted interface**. It behaves like a SortedSet with the exception that we have navigation methods available in addition to the sorting mechanisms of the SortedSet. For example, NavigableSet interface can navigate the set in reverse order compared to the order defined in SortedSet.
The classes that implement this interface are, and ConcurrentSkipListSet

Methods of NavigableSet (Not in SortedSet):

1. **Lower(E e)** : Returns the greatest element in this set which is less than the given element or NULL if there is no such element.
2. **Floor(E e)** : Returns the greatest element in this set which is less than or equal to given element or NULL if there is no such element.
3. **Ceiling(E e)** : Returns the least element in this set which is greater than or equal to given element or NULL if there is no such element.
4. **Higher(E e)** : Returns the least element in this set which is greater than the given element or NULL if there is no such element.
5. **pollFirst()** : Retrieve and remove the first least element. Or return null if there is no such element.
6. **pollLast()** : Retrieve and remove the last highest element. Or return null if there is no such element.

SAMPLE CODE

```
import java.util.NavigableSet;
import java.util.TreeSet;

public class hashset
{
   public static void main(String[] args)
   {
      NavigableSet<Integer> ns = new TreeSet<>();
      ns.add(0);
      ns.add(1);
      ns.add(2);
      ns.add(3);
      ns.add(4);
      ns.add(5);
      ns.add(6);

      NavigableSet<Integer> reverseNs = ns.descendingSet();

      System.out.println("Normal order: " + ns);
      System.out.println("Reverse order: " + reverseNs);

      NavigableSet<Integer> threeOrMore = ns.tailSet(3, true);
      System.out.println("3 or  more:  " + threeOrMore);
      System.out.println("lower(3): " + ns.lower(3));
      System.out.println("floor(3): " + ns.floor(3));
      System.out.println("higher(3): " + ns.higher(3));
      System.out.println("ceiling(3): " + ns.ceiling(3));

      System.out.println("pollFirst(): " + ns.pollFirst());
      System.out.println("Navigable Set:  " + ns);

      System.out.println("pollLast(): " + ns.pollLast());
      System.out.println("Navigable Set:  " + ns);

      System.out.println("pollFirst(): " + ns.pollFirst());
      System.out.println("Navigable Set:  " + ns);

      System.out.println("pollFirst(): " + ns.pollFirst());
      System.out.println("Navigable Set:  " + ns);

      System.out.println("pollFirst(): " + ns.pollFirst());
      System.out.println("Navigable Set:  " + ns);

      System.out.println("pollFirst(): " + ns.pollFirst());
      System.out.println("pollLast(): " + ns.pollLast());
```

```
    }
}
```

OUTPUT

```
Normal order: [0, 1, 2, 3, 4, 5, 6]
Reverse order: [6, 5, 4, 3, 2, 1, 0]
3 or  more:  [3, 4, 5, 6]
lower(3): 2
floor(3): 3
higher(3): 4
ceiling(3): 3
pollFirst(): 0
Navigable Set:  [1, 2, 3, 4, 5, 6]
pollLast(): 6
Navigable Set:  [1, 2, 3, 4, 5]
pollFirst(): 1
Navigable Set:  [2, 3, 4, 5]
pollFirst(): 2
Navigable Set:  [3, 4, 5]
pollFirst(): 3
Navigable Set:  [4, 5]
pollFirst(): 4
pollLast(): 5
```

CONCURRENT SKIP LIST SET

public class **ConcurrentSkipListSet<E>**

extends AbstractSet<E>

implements NavigableSet<E>, Cloneable, Serializable

- A scalable concurrent NavigableSet implementation based on a ConcurrentSkipListMap.
- ***The elements of the set are kept sorted according to their natural ordering,*** or by a Comparator provided at set creation time, depending on which constructor is used.
- This implementation provides expected average *log(n)* time cost for the contains, add, and remove operations and their variants.
- Insertion, removal, and access operations safely execute concurrently by multiple threads. Iterators are *weakly consistent*, returning elements reflecting the state of the set at some point at or since the creation of the iterator. They do *not* throw ConcurrentModificationException, and may proceed concurrently with other operations. Ascending ordered views and their iterators are faster than descending ones.
- Beware that, unlike in most collections, the size method is *not* a constant-time operation. Because of the asynchronous nature of these sets, determining the current number of elements requires a traversal of the elements, and so may report inaccurate results if this collection is modified during traversal.
- Additionally, the bulk operations addAll, removeAll, retainAll, containsAll, equals, and toArray are *not* guaranteed to be performed atomically. For example, an iterator operating concurrently with an addAll operation might view only some of the added elements.
- This class and its iterators implement all of the *optional* methods of the Set and Iterator interfaces. Like most other concurrent collection implementations, this class does not permit the use of null elements, because nullarguments and return values cannot be reliably distinguished from the absence of elements.
- This class is a member of the Java Collections Framework.

ConcurrentModificationException

public class **ConcurrentModificationException**
extends RuntimeException

This exception may be thrown by methods that have detected concurrent modification of an object when such modification is not permissible.

For example, it is not generally permissible for one thread to modify a Collection while another thread is iterating over it. In general, the results of the iteration are undefined under these circumstances. Some Iterator implementations (including those of all the general purpose collection implementations provided by the JRE) may choose to throw this exception if this behaviour is detected. Iterators that do this are known as *fail-fast* iterators, as they fail quickly and cleanly, rather that risking arbitrary, non-deterministic behaviour at an undetermined time in the future.

Note that this exception does not always indicate that an object has been concurrently modified by a *different* thread. If a single thread issues a sequence of method invocations that violates the contract of an object, the object may throw this exception. For example, if a thread modifies a collection directly while it is iterating over the collection with a fail-fast iterator, the iterator will throw this exception.

Note that fail-fast behaviour cannot be guaranteed as it is, generally speaking,impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast operations throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: *ConcurrentModificationException* should be used only to detect bugs.

JAVA VECTOR CLASS

- **Java Vector Class**

  Java Vector class comes under the java.util package. The vector class implements a growable array of objects. Like an array, it contains the component that can be accessed using an integer index.

  Vector is very useful if we don't know the size of an array in advance or we need one that can change the size over the lifetime of a program.

  Vector implements a dynamic array that means it can grow or shrink as required. It is similar to the ArrayList, but with two differences-

- ❖ Vector is synchronized.
- ❖ The vector contains many legacy methods that are not the part of a collections framework

- **Java Vector Class Declaration**

```
public class Vector<E>
extends Object<E>
implements List<E>, Cloneable, Serializable
```

- **Example 1:**

```
import java.util.*;
public class VectorExample1 {
 public static void main(String args[]) {
//Create an empty vector with initial capacity 4
    Vector<String> vec = new Vector<String>(4);
//Adding elements to a vector
     vec.add("Tiger");
     vec.add("Lion");
     vec.add("Dog");
     vec.add("Elephant");
  //Check size and capacity
     System.out.println("Size is: "+vec.size());
     System.out.println("Default capacity is: "+vec.capacity());
   //Display Vector elements
     System.out.println("Vector element is: "+vec);
         vec.addElement("Rat");
         vec.addElement("Cat");
         vec.addElement("Deer");
         //Again check size and capacity after two insertions
         System.out.println("Size after addition: "+vec.size());
         System.out.println("Capacity after addition is: "+vec.capacity());
         //Display Vector elements again
         System.out.println("Elements are: "+vec);
         //Checking if Tiger is present or not in this vector
          if(vec.contains("Tiger"))
          {
            System.out.println("Tiger is present at the index " +vec.indexOf("Tiger"));
         }
          else
```

```
        {
          System.out.println("Tiger is not present in the list.");
        }
        //Get the first element
       System.out.println("The first animal of the vector is = "+vec.firstElement());
      //Get the last element
      System.out.println("The last animal of the vector is = "+vec.lastElement());
    }
}
```