Using Iterator to modify a Collection which is not possible with foreach loop:

```
private static List<String> getDaysOfWeek() {
    List<String> returnVal = new ArrayList<>();
    returnVal.add("Monday");     returnVal.add("Tuesday");
    returnVal.add("Wedsday");    returnVal.add("Thursday");
    returnVal.add("January");    returnVal.add("Friday");
    returnVal.add("Saturday");   returnVal.add("Sunday");
    return returnVal;
}

public static void main(String[] args) {
  List<String> daysOfWeek = getDaysOfWeek();

  System.out.println(daysOfWeek);

  Iterator<String> iterator = daysOfWeek.iterator();

/**
 * Below code we are explicitly maintaining an index to iterate over
 * the collection and then remove an element
 */
```

```
 for(int i = 0; i < daysOfWeek.size(); i++) {
    if(daysOfWeek.get(i).equals("January")) {
     daysOfWeek.remove("January");
   }
  }
  System.out.println(daysOfWeek);
```

```
/**
 * Below code uses foreach loop, so that we don't have to explicitly maintain
 * an index to traverse the collection.  But any attempt to modify the Collection
 * in foreach loop will result in ConcurrentModificationException.
 *
 * In case of foreach loop the Collection being processed is tightly
 * coupled with the loop, so concurrent modification is not possible.
 */
```

```
for (String string : daysOfWeek) {
  if(string.equals("January"))
     daysOfWeek.remove("January");  [ or daysOfWeek.add("February"); ]
}
System.out.println(daysOfWeek);
```

```
/**
 * Below code uses Iterator to traverse the collection.
 * An Iterator and the Collection are tightly coupled.
 * We can modify a Collection while traversing it with
 * the help of an iterator.
```

```
 *
 * NOTE: As of Java-11, Iterator has support for remove(Object o)
 * method of collection.  In later versions support for add(Object o)
 * has also been provided.
 */
```

```
  while(iterator.hasNext()) {
    if(iterator.next().equals("January")) {
      iterator.remove();
    }
  }
  System.out.println(daysOfWeek);
```

} *//end of main*

Stack

```
public class StackDemo {

      public static void main(String[] args) {
            Stack<String> topicStack = new Stack<>();

            topicStack.push("Collections");
            topicStack.push("PostgreSQL");
            topicStack.push("JDBC");
            topicStack.push("Multithreading & Concurrency");
            topicStack.push("IO Streams with NIO");

            System.out.println(topicStack.size());
            System.out.println(topicStack);
            System.out.println("~".repeat(20));

            String currentTopic = topicStack.peek();
            System.out.println("After calling peek method: " + topicStack.size());
            System.out.println(currentTopic);
            System.out.println(topicStack);
            System.out.println("~".repeat(20));

            currentTopic = topicStack.pop();
            System.out.println("After calling pop 1st attempt: "
                        + topicStack.size());
            System.out.println(currentTopic);
            System.out.println(topicStack);
            System.out.println("~".repeat(20));

            currentTopic = topicStack.pop();
            System.out.println("After calling pop 2nd attempt: "
                        + topicStack.size());
            System.out.println(currentTopic);
            System.out.println(topicStack);
            System.out.println("~".repeat(20));
```

```
        System.out.println("Checking if stack is empty: " + topicStack.empty());
        System.out.println("Searching for JDBC in topic stack: "
                           + topicStack.search("JDBC"));

        System.out.println("Searching for Multithreading & Concurrency in topic stack: " +
                           topicStack.search("Multithreading & Concurrency"));
        System.out.println("~".repeat(20));

        System.out.println("Before traversing on topicStack with iterator: "
                           + topicStack.size());
        Iterator<String> iterator = topicStack.iterator();
        while(iterator.hasNext()) {
              System.out.println("Next topic: " + iterator.next());
        }
        System.out.println("After traversing on topicStack with iterator: "
                           + topicStack.size());

        for (String string : topicStack) {
              System.out.println("Next topic: " +  string);
        }
        System.out.println("After traversing on topicStack with foreach loop: "
                           + topicStack.size());

        for(int i = 0; i < topicStack.size(); i++) {
              System.out.println("Next topic: " + topicStack.get(i));
        }
        System.out.println("After traversing on topicStack with for loop and using get(idx) method: "
                           + topicStack.size());

        // Below code is a logicl error.  As we using pop() inside for loop for each iteration
        //  the size of the stack will reduce by 1.  Then for each iteration the for-loop condition
        //  will be different.
        // --> Instead we should use while(!topicStack.empty()){ }
        /*
        for(int i = 0; i < topicStack.size(); i++) {
           System.out.println(topicStack.size());
           System.out.println("Next topic: " + topicStack.pop());
        }
        System.out.println("After traversing on topicStack with "
                           + "for loop and using pop() method: "
                           + topicStack.size());
        */

        while(!topicStack.empty() ) {
          System.out.println(topicStack.size());
          System.out.println("Next topic: " + topicStack.pop());
        }
        System.out.println("After traversing on topicStack with while loop "
                     + "with !empty() and using pop() method: "
                     + topicStack.size());
    }

}
```

NOTE: search() method works properly only when equals() is overridden in POJO.

Linked List:

Linked List is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node.

Advantages of Linked List:
As elements (nodes) are connected to other nodes by the address informatoin of other nodes and not by being present contiguous memory locations, there will not be a need to reorganize the element positions every time we have to insert or remove an element unlike arrays where elements are connected by their contiguous memory locations.

With arrays adding or removing an element will result in reorganizing the memory locations of later elements, and the effort required is directly proportional to the number of elements that are present after the location or index at which we have to insert or remove. With linked list no such effort of reorganizing elements is required.

While working with arrays if the entire length of array has been consumed to add a new element another new array of bigger size has to be created. The data from old array has to be copied into new array, the new element has to be added to the array and the old array has to be discarded. This effort is required every time the length of the array has be consumed and we have to add another element. All this effort is not required with linked list to add a new element.

Disadvantages:
The nodes (elements) cannot be accessed directly instead we need to start from the head and follow through the link to reach a node we wish to access.

In case of a double linked list we can start from either head node or tail node, but with single linked list it is always the head only.

```
public class LinkedListDemo {

	public static void main(String[] args) {
		Queue<String> topicQueueLL = new LinkedList<>();

		topicQueueLL.offer("Collections");
		topicQueueLL.offer("PostgreSQL");
		topicQueueLL.offer("JDBC");
		topicQueueLL.offer("Multithreading & Concurrency");
		topicQueueLL.offer("IO Streams with NIO");

		System.out.println(topicQueueLL.size());
		System.out.println(topicQueueLL);
		System.out.println("~".repeat(20));

		String currentTopic = topicQueueLL.peek();
		System.out.println("After calling peek method: "
                                        + topicQueueLL.size());
		System.out.println(currentTopic);
		System.out.println(topicQueueLL);
		System.out.println("~".repeat(20));
```

```java
currentTopic = topicQueueLL.poll();
System.out.println("After calling poll method: "
                                + topicQueueLL.size());
System.out.println(currentTopic);
System.out.println(topicQueueLL);
System.out.println("~".repeat(20));

// Alternatively we can use remove() instead of poll()
// But when the Queue is empty poll() will return null, while
//    remove will raise NoSuchElementException.
// It is safe to use poll()
currentTopic = topicQueueLL.remove();
System.out.println("After calling remove method: "
                          + topicQueueLL.size());
System.out.println(currentTopic);
System.out.println(topicQueueLL);
System.out.println("~".repeat(20));


//Clear the queue and test poll() and then remove()
// Similarly peek() and element()
/*
topicQueueLL.clear();
System.out.println(topicQueueLL.poll());
System.out.println(topicQueueLL.remove());

topicQueueLL.clear();
System.out.println(topicQueueLL.peek());
System.out.println(topicQueueLL.element());
*/


// Traversing with while loop and terminate the loop
//   with !isEmpty()
// Retrieving elemenents with poll() method
while(!topicQueueLL.isEmpty()) {
      System.out.println(topicQueueLL.poll());
}
System.out.println("Size after traversing with while loop: "
                          + topicQueueLL.size());


/**
 * LinkedList class implements below interfaces:
 * See
 * 1) java.io.Serializable - marker interface to support serialization
 *        For this data-structure to be serialized properly the classes of
 *        the objects stored in it also should be Serializable
 *  2) java.lang.Clonable - marker interface to support cloning
 *        Support is for shallow cloning only.
 *  3) java.util.List
 *  4) java.util.Deque and Deque extends java.util.Queue
```

```
                *
                * See Java documentation to know the methods of List, Queue & Deque
                *
                * https://docs.oracle.com/javase/8/docs/api/index.html
                *
                * https://docs.oracle.com/en/java/javase/11/docs/api/index.html
                */
               List<String> topicListLL = (List) topicQueueLL;
               Deque<String> topicDequeLL = (Deque) topicQueueLL;
               LinkedList<String> topicLL = (LinkedList) topicQueueLL;
//             topicLL.
//             topicDequeLL.
//             topicQueueLL.
//             topicListLL.

               /**
                * Methods like peekFirst(), peekLast(), pollFirst(), pollLast()
                * are from Deque interface
                *
                * Methods like add()/offer(), remove()/poll(), element()/peek()
                * are from Queue interface
                *
                * Methods like add remove with index based support are from
                * List interface
                * add(int index, E element);
                * set(int index, E element);
                * remove(int index);
                * etc..
                */
        }

}
```

Priority Queue

Priority Queue is a data structure in which elements are ordered by priority, with the highest-priority elements appearing at the front of the queue.

The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering when default constructor is used, or by a Comparator provided at queue construction time.

Natural ordering implies comparison as provided by implementing Comparable interface. In this case by default min-heap algorithm is used. To make use of max-heap algorithm with natural ordering, pass Comparator.reverseOrder() to the PriorityQueue constructor.

PriorityQueue with String data:

```
public class PriorityQueueDemo01 {

        public static void main(String[] args) {
               ////Default constructor of PriorityQueue provides
               //// Min-Heap functionality
```

```java
        Queue<String> topicPriQueue = new PriorityQueue<>();

        ////For Max-Heap functionality pass Comparator.reverseOrder()
        ////  to the PriorityQueue constructor
        //Queue<String> topicPriQueue =
        //          new PriorityQueue<>(Comparator.reverseOrder());

        topicPriQueue.offer("Collections");
        topicPriQueue.offer("PostgreSQL");
        topicPriQueue.offer("JDBC");
        topicPriQueue.offer("Multithreading & Concurrency");
        topicPriQueue.offer("IO Streams with NIO");

        //Stage-1
        System.out.println(topicPriQueue.size());
        System.out.println(topicPriQueue);
        System.out.println("~".repeat(20));

        //Stage-2
        String currentTopic = topicPriQueue.peek();
        System.out.println("After calling peek method: "
                                + topicPriQueue.size());
        System.out.println(currentTopic);
        System.out.println(topicPriQueue);
        System.out.println("~".repeat(20));

        //Stage-3
        currentTopic = topicPriQueue.poll();
        System.out.println("After calling poll method: "
                                + topicPriQueue.size());
        System.out.println(currentTopic);
        System.out.println(topicPriQueue);
        System.out.println("~".repeat(20));


        //Stage-4
        currentTopic = topicPriQueue.poll();
        System.out.println("After calling poll method: "
                                + topicPriQueue.size());
        System.out.println(currentTopic);
        System.out.println(topicPriQueue);
        System.out.println("~".repeat(20));
    }
}
```

PriorityQueue with Student POJO:

In the POJO generate parameterized constructor, toString(), getters & setters.
Implement Comparable to compare by stuId

Student.java

```java
public class Student implements Comparable<Student>{
```

```java
	private int stuId;
	private String stuName;
	private int groupId;
	private int rank;

	public Student(int stuId, String stuName, int groupId, int rank) {
		this.stuId = stuId;
		this.stuName = stuName;
		this.groupId = groupId;
		this.rank = rank;
	}

	@Override
	public String toString() {
		return "Student [stuId=" + stuId + ", stuName=" + stuName
			+ ", groupId=" + groupId + ", rank=" + rank + "]";
	}

	@Override
	public int compareTo(Student other) {
		return this.stuId - other.stuId;
	}

	// Generate all getters & setters
}
```

PriorityQueueDemo02.java

```java
public class PriorityQueueDemo02 {

	public static void main(String[] args) {
		Queue<Student> stuQ = new PriorityQueue<>();
		//Queue<Student> stuQ = new PriorityQueue<>(Comparator.reverseOrder());


		/*
		Queue<Student> stuQ = new PriorityQueue<>(new Comparator<Student>() {
			public int compare(Student s1, Student s2) {
				return s1.getRank() - s2.getRank();
			}
		});
		*/

		/*
		Queue<Student> stuQ = new PriorityQueue<>(new Comparator<Student>() {
			public int compare(Student s1, Student s2) {
				int groupIdDiff = s1.getGroupId() - s2.getGroupId();
				if(groupIdDiff == 0) {
					return s1.getStuName().compareTo(s2.getStuName());
				}
				return groupIdDiff;
			}
		});
		*/

		stuQ.offer(new Student(276, "Vishal", 20, 23));
```

```java
            stuQ.offer(new Student(263, "Beena", 10, 8));
            stuQ.offer(new Student(280, "Abigail", 20, 12));
            stuQ.offer(new Student(271, "Vishal", 20, 32));
            stuQ.offer(new Student(206, "Ajay", 10, 2));

            System.out.println(stuQ);

            Student stu = stuQ.poll();
            System.out.println("After 1st poll()");
            System.out.println("Poll result: " + stu);
            System.out.println(stuQ);

            stu = stuQ.poll();
            System.out.println("After 2nd poll()");
            System.out.println("Poll result: " + stu);
            System.out.println(stuQ);
        }
}
```

ArrayDeque : The ArrayDeque class provides constant-time performance for inserting and removing elements from both ends of the queue

ArrayDequeDemo.java

```java
public class ArrayDequeDemo {

        public static void main(String[] args) {
            Deque<Integer> deque = new ArrayDeque<>();

            deque.offer(35);
            deque.offer(24);
            System.out.println(deque);

            /* offerFirst will add an element at the begining of the queue */
            deque.offerFirst(67);

            /* offer & offerLast will add an element at the end of the queue */
            deque.offerLast(6);
            System.out.println(deque);

            List<Integer> list = new LinkedList<>();
            list.add(-14);
            list.add(-8);
            list.add(-59);
            deque.addAll(list);
            System.out.println(deque);
            System.out.println("~".repeat(20));

            Integer res = deque.peek();
            System.out.println("peek(): " + res);

            res = deque.peekFirst();
            System.out.println("peekFirst(): " + res);

            res = deque.peekLast();
```

```java
            System.out.println("peekLast(): " + res);
            System.out.println("~".repeat(20));

            System.out.println(deque + "\n");
            res = deque.poll();
            System.out.println("poll(): " + res);
            System.out.println(deque + "\n");

            res = deque.pollFirst();
            System.out.println("pollFirst(): " + res);
            System.out.println(deque + "\n");

            res = deque.pollLast();
            System.out.println("pollLast(): " + res);
            System.out.println(deque + "\n");
        }

}
```