

Zunächst definieren wir die Basisklasse für die Listenelemente, die gleichzeitig einen Verweis auf weitere Listenelemente beinhaltet:

```
class ListElement {
    int value;
    ListElement next;

    ListElement() {
        next=null;
        value=0;
    }
}
```

Die folgende Klasse implementiert die Liste. Wir verwenden `ListElement head` als Anker für den Anfang der Liste (sein Inhalt wird nicht verwendet), sowie `ListElement point` um auf die Position vor dem aktuellen Listenelement zu verweisen und die "Bewegung" innerhalb der Liste zu ermöglichen. Das Listenende wird durch Setzen der Klassenvariable `next` auf `null` gekennzeichnet. Dazu beachten Sie auch folgende schematische Darstellung:

```
/**
 * This class implements a linked list. The head is not part of the
 * list elements. The point points to the element BEFORE the active
 * element.
 *
 * +-----+
 * |head|->null           is the empty list.
 * +-----+           list.empty()==true, list.end()==true.
 *     \__point
 *
 * +-----+ +-----+ +-----+
 * |head|->| 2  |->| 3  |->null   is a list containing a 2 and a 3.
 * +-----+ +-----+ +-----+
 *     \__point               in this case, getValue returns the 2.
 *                             list.empty()==false, list.end()==false
 *
 * +-----+ +-----+ +-----+
 * |head|->| 2  |->| 3  |->null   is a list containing a 2 and a 3.
 * +-----+ +-----+ +-----+
 *                 \__point     in this case, getValue returns the 3.
 *                             list.empty()==false, list.end()==false
 *
 * +-----+ +-----+ +-----+
 * |head|->| 2  |->| 3  |->null   is a list containing a 2 and a 3.
 * +-----+ +-----+ +-----+
 *                 \__point     in this case, getValue fails.
 *                             list.empty()==false, list.end()==true
 */
```

Bei der Implementierung der Methoden der Klasse `List` ist der Sonderfall, dass das Ende der Liste erreicht ist, immer zu beachten. Eine Variante der Implementierung ist dann die folgende:

```
class List {
    ListElement head, point;

    List() {           // initialize the list
        head=new ListElement();
        head.value=-1; // this value is not used
        point=head;
    }
}
```

```

public boolean empty() {
    if(head.next==null) {
        return true;
    } else {
        return false;
    }
}

public boolean end() {
    if(point.next==null) {
        return true;
    } else {
        return false;
    }
}

public void next() { // go the next list element
    if(!end()) {
        point=point.next;
    }
}

public void rewind() { // go back to the beginning of the list
    point=head;
}

public int getValue() { // get the value of the active element
    if(!end()) {
        return point.next.value;
    } else {
        System.err.println("ERROR");
        return -1;
    }
}

public void setValue(int val) { // set the value of the active element
    if(!end()) {
        point.next.value=val;
    } else {
        System.err.println("ERROR");
    }
}

public void insert(int val) {
    // insertion BEFORE the active element
    // (which is point.next, not point!)
    ListElement n=new ListElement(); // reserve memory for the new element
    n.value=val;

    n.next=point.next; // adjust the pointers
    point.next=n;
}

public void remove() {
    if(!end()) {
        point.next=point.next.next; // automatic garbage collection
    } else {
        System.err.println("ERROR");
    }
}

```

```

public void print() {
    ListElement akt=point; //save current point

    rewind();
    do {
        System.out.println(getValue());
        next();
    } while(!end());

    point=akt; // go back to the original point
}
}

```

Für die Implementierung der doppelt verketteten Liste ist die folgende Basisklasse mit zwei Referenzierungsvariablen notwendig:

```

class DoubleListElement {
    int value;
    DoubleListElement next;
    DoubleListElement prev;

    DoubleListElement() {
        next=null;
        prev=null;
        value=-1;
    }
}

```

In der unten vorgeschlagenen Implementierung der Klasse `DoubleList` verweist jetzt `point` auf das aktuelle Listenelement selbst. Diese Konvention erleichtert das Einfügen neuer Elemente. Schematisch ist die doppelt verkettete Liste so darstellbar:

```

/**
 * This class implements a linked list. The point points TO the active element.
 *
 *      +-----+
 * null<-|head|->null           is the empty list.
 *      +-----+               list.empty()==true, list.end()==true,
 *      \__point                 list.start()==true.
 *
 *
 *      +-----+ +-----+ +-----+
 * null<-|head|<->|   |<->|   |->null  is a list containing a 2, a 3, and a 4.
 *      | 2 |   | 3 |   | 4 |
 *      +-----+ +-----+ +-----+
 *      \__point                 in this case, getValue returns the 2.
 *                               list.empty()==false, list.end()==false,
 *                               list.start()==true.
 *
 *      +-----+ +-----+ +-----+
 * null<-| 2 |<->| 3 |<->| 4 |->null  is a list containing a 2, a 3, and a 4.
 *      +-----+ +-----+ +-----+
 *                               \__point
 *                               in this case, getValue returns the 3.
 *                               list.empty()==false, list.end()==false,
 *                               list.start()==false.
 */

```

Die Implementierung der Methoden ist im wesentlichen analog zu der einfach verketteten Liste. Die Methode `start()` prüft, ob der "Zeiger" `prev` auf `null` verweist oder nicht. Bei der Implementierung der in

der Aufgabenstellung geforderten Methode `insertBefore()` ist zu beachten, dass der Verweis `next` des vorherigen Elementes (das durch `point.prev`) erreichbar ist, entsprechend geändert werden muss, aber nur falls `point` nicht auf das erste Element der Liste verweist. Im Sonderfall des Listenbeginns muss `head` das gerade eingefügten neuen Element referenzieren, da das Einfügen **vor** dem Listenanfang stattfinden soll.

```
class DoubleList {
    DoubleListElement head, point;

    DoubleList() {
        head=new DoubleListElement();
        head.value=-1;
        point=head;
    }

    public boolean empty() {
        if((head.next==null)&&(head.prev==null)) {
            return true;
        } else {
            return false;
        }
    }

    public boolean end() {
        if(point.next==null) {
            return true;
        } else {
            return false;
        }
    }

    public boolean start() {
        if(point.prev==null) {
            return true;
        } else {
            return false;
        }
    }

    public void next() {
        if(!end()) {
            point=point.next;
        }
    }

    public void back() {
        if(!start()) {
            point = point.prev;
        }
    }

    public void rewind() {
        point=head;
    }
}
```

```

public int getValue() {
    if(!end()) {
        return point.value; // note: the point value itself is printed,
    } // not the point.next.value!
    else {
        System.err.println("ERROR");
        return -1;
    }
}

public void setValue(int val) {
    if(!end()) {
        point.value=val; // note: the point value itself is set,
    } // not the point.next.value!
    else {
        System.err.println("ERROR");
    }
}

public void insertBefore(int val) {

    DoubleListElement n=new DoubleListElement();
    n.value=val;
    n.prev = point.prev;
    n.next = point; // the active element is referenced
                  // by the inserted element as the next element
    if(!start()) {
        point.prev.next = n; // the previous element references
    } // the inserted element as the next one
    else {
        head = n; // the inserted element is the new list beginning
    }
    point.prev = n; // the inserted element is
                  // the new previous element for the active element
}

public void print() {

    DoubleListElement akt=point; //save point
    rewind();
    do {
        System.out.println(getValue());
        next();
    } while(!end());

    point=akt; //restore point
}
}

```

Die hier vorgeschlagene Implementierungen von Stack und Queue greifen auf die Methoden der Klasse List zu, d. h. die Klassenvererbung von Java wird ausgenutzt. Alternativ könnte man auch die entsprechende Klassen vollständig neu implementieren, also im wesentlichen die Methoden insert() und remove() modifizieren.

Beim Stack wird immer am Anfang der Liste ein Element eingefügt und auch entfernt:

```

class Stack extends List {
    public void push(int val) {
        rewind();
        insert(val);
    }
}

```

```

    public int pop() {
        rewind();
        int x = getValue();
        remove();
        return x;
    }
}

```

Bei der Queue, in der unten vorgeschlagenen Implementierung, wird ein neues Element immer **am Ende der Liste** eingefügt (Aufruf von `insert`-Methode gefolgt von `next()`). Um dann vom Anfang der Queue die Elemente rauszunehmen, und dann wieder am Ende weiter zu schreiben, benötigt man einen zusätzlichen Verweis, der als verborgene Variable `endPoint` gespeichert wird.

```

class Queue extends List {

    private ListElement endPoint;

    public void put(int val) {
        insert(val);
        next();
    }
    public int get() {
        endPoint = point; // save the end position
        rewind();
        int x = getValue();
        remove();
        point = endPoint; // return to the end position
        return x;
    }
}

```

Der Testdurchlauf wird in der `main()`-Methode der Klasse `ListTest` implementiert. Bei dem Aufgabenteil a) ist folgendes zu beachten:

- In den Codezeilen gekennzeichnet mit `!!!` wird durch Aufruf von `insert()` neuer Speicher reserviert. Obwohl eine dritte Liste nicht verwendet wird, kommt man mit dem bisher vorhandenen Speicher nicht aus. Das liegt daran, dass in Java die Klassenvariablen (hier: die Verweise `head` und `point`) in Objekten eingekapselt sind. Sie können nicht als ("call-by-reference") Parameter an die Methoden der gleichen Klasse übergeben werden, die ein anderes Objekt der Klasse aufruft. Aus diesem Grund ist Java für die Implementierung solcher Listenoperationen nicht besonders geeinigt.

Das Testen der doppelt verketteten Liste, sowie vom Stack und von der Queue, besteht aus Einfügungen von Elementen und Ausgabe eines bestimmten Elementes.

```

class ListTest {
    public static void main(String args[]) {
        int k;
        List liste1=new List();
        List liste2=new List();

        System.out.println("Trying to remove from empty list...");
        liste1.remove(); // Output: ERROR

        for(int i=1;i<=10;++i) { // fill the first list with elements
            liste1.insert(i);
        }
        for(int i=1;i<=15;++i) { // fill the second list with elements
            liste2.insert(-i);
        }
    }
}

```

```

System.out.println("Merging two lists into List 1..."); // Aufgabe 4 Teil a)
liste1.rewind();
liste2.rewind();
while(!liste1.end() && !liste2.end()) {
    liste1.insert(liste2.getValue()); // !!!
    liste1.next(); // skip the just inserted element
    liste1.next(); // skip the next (previously present) element
    liste2.next(); // go to the next element in the second list
}
while(!liste2.end()) {
    liste1.insert(liste2.getValue()); // !!!
    liste1.next(); // skip the just inserted element
    liste2.next(); // go to the next element in the second list
}
System.out.println("List 1 after merging:");
liste1.print();
// should be printed:
// -15 10 -14 9 -13 8 -12 7 -11 6 -10 5 -9 4 -8 3 -7 2 -6 1 -5 -4 -3 -2 -1
/*****
System.out.println("Testing the double linked list (going back 4 elements)...");
DoubleList d = new DoubleList();
for(int i=1; i<=10; ++i) {
    d.insertBefore(i);
}
d.back();
d.back();
d.back();
d.back(); // go back 4 Elements
System.out.println(d.getValue()); // 7 should be printed
/*****
System.out.println("Testing the stack...");
Stack st = new Stack();
for(int i=1; i<=10; ++i) {
    st.push(i);
}
k = st.pop();
k = st.pop();
System.out.print("The value pushed before last: ");
System.out.println(k); // 9 should be printed
/*****
System.out.println("Testing the queue...");
Queue q = new Queue();
for(int i=1; i<=10; ++i) {
    q.put(i);
}
k = q.get();
k = q.get();
System.out.print("The value put as second: ");
System.out.println(k); // 2 should be printed
}
}

```