Für die Länge der Warteschlange soll der Endknoten nicht zählen, d.h. es wird nur die Anzahl der Datenknoten benötigt. Die Frage nach der Länge wird also bis zum Endknoten durchgereicht, dieser antwortet mit 0, alle anderen zählen zum Ergebnis ihres Vorgängers eins dazu und geben weiter zurück.
Beispiel:Der letzte Datenknoten erhält vom Endknoten die 0, addiert 1 und gibt an den vorletzten 1 zurück. Dieser addiert wiederum 1 und gibt 2 weiter zurück, usw.

```
//class MyListComp
public int length() {
    return first.length();
}

//class Node
public abstract int length();

//class DataNode
@Override
public int length() {
    return next.length() + 1;
}

//class EndNode
@Override
public int length() {
    return 0;
}
```

Um ein Datenelement an einer bestimnmten Position zu finden müssen wir wieder auf das Konzept des Zählers, der durchgereicht wird, zurückgreifen. Die Abfrage, ob die Liste kürzer als die gesuchte Position ist können wir umgehen, indem wir den Endknoten null zurückgeben lassen, sollte er erreicht werden. Wie in früheren Beispielen soll in dieser Implementierung außerdem das Zählen bei 1 beginnen, d.h. der Erste ist wirklich der Erste und nicht der „Nullte“ in der Reihe.

```
//class MyListComp
public DataElement itemAtPosition(int position){
    int counter = 1;
    DataElement found = first.itemAtPosition(position, counter);
    return found;
}

//class Node
public abstract itemAtPosition(int position, int counter);

//class DataNode
@Override
public DataElement itemAtPosition(int position, int counter) {
    if(counter == position){
        return this.getData();
    } else {
        return next.itemAtPosition(position, counter + 1);
    }
}

//class EndNode
@Override
public DataElement itemAtPosition(int position, int counter) {
```

```
        return null;
    }
```

Die Methode, die in der Schlange sucht folgt der exakt selben Logik, nur gibt hier der Endknoten die (willkürlich) festgelegte -1 statt null zurück:

```
    //class MyListComp
    public int searchItemInQueue(DataElement data) {
        int counter = 1;
        int position = first.searchItemInQueue(data,counter);
        return position;
    }

    //class Node
    public abstract int searchItemInQueue(DataElement data, int counter);

    //class DataNode
    @Override
    public int searchItemInQueue(DataElement data, int counter) {
        if(this.getData().equals(data)){
            return counter;
        } else {
            return next.searchItemInQueue(data, counter +1);
        }
    }

    //class EndNode
    @Override
    public int searchItemInQueue(DataElement data, int counter) {
        return -1;
    }
```

Wie immer kann für die enthält()-Methode ein eigener rekursiver Durchlauf verwendet werden, oder es wird auf die Methode aus der vorherigen Aufgabe zurückgegriffen, diesmal in der kürzeren Variante:

```
    //class MyListComp
    public boolean contains(DataElement data) {
        if(this.searchItemInQueue(data) != -1) {
            return true;
        } else {
            return false;
        }
    }
```

Die Methode, die an einer beliebigen Stelle entfernt, kann auf die vorneEntfernen()-Methode zurückgreifen, falls die spezifizierte Position der ersten entspricht. Andernfalls wird wieder ein Zähler benötigt, die Struktur sieht völlig analog zu früheren Varianten aus (der Endknoten wird wieder null zurückgeben!):

```
    //class MyListComp
    public DataElement removeAtPosition(int position) {
        if(position <= 0) {
            return null;
        }
        if(position == 1) {
            return this.pop();
        } else {
            int counter = 1;
            return first.removeAtPosition(position, counter);
```

```
        }
    }

    //class Node
    public abstract DataElement removeAtPosition(int position, int counter);

    //class DataNode
    @Override
    public DataElement removeAtPosition(int position, int counter){
        if(counter == position - 1){
            DataElement toReturn = next.getData();
            next = next.getNext();
            return toReturn;
        } else {
            return next.removeAtPosition(position, counter + 1);
        }
    }

    //class EndNode
    @Override
    public DataElement removeAt(int position, int counter){
        return null;
    }
```

In der Liste wird nur auf dem Ersten das Entfernen gestartet. Da der Erste in jedem Fall mindestens ein Endknoten ist, entfällt wiederum eine Prüfung, ob die Liste leer ist. Ein Endknoten gibt nur sich selbst zurück und es entsteht kein Schaden. Durch die Implementierung auf diese Weise ist allerdings keine Bestätigung der Ausführung möglich, da für das „elegante“ Durchreichen eine Rückgabe des Knotens notwendig ist.
Im Datenknoten wird überprüft, ob das übergebene Element identisch mit dem Inhalt dieses Knotens ist (dazu wird wieder die vergleichsMethode des Datenelements genutzt!). Falls dies der Fall ist, wird der Nachfolger des Knotens zurückgegeben. Dadurch erhält sein Vorgänger eine Referenz auf den Nachfolger und der Knoten mit den gesuchten Daten ist entfernt.

```
    //class MyListComp
    public void removeElement(DataElement toRemove) {
        first = first.removeElement(toRemove);
    }

    //class Node
    public abstract Node removeElement(DataElement toRemove);

    //class DataNode
    public Node removeElement(DataElement toRemove) {
        if(toRemove.getData().equals(data.getData())) {
            return next;
        } else {
            next = next.removeElement(toRemove);
        }
        return this;
    }

    //class EndNode
    public Node removeElement(DataElement toRemove) {
        return this;
    }
```