Warteschlangen und ihre Implementierungen

Die Warteschlange (Queue) als ADT

Eine Warteschlange (Queue) verwaltet eine Folge von Objekten, wobei man immer Objekte (an das Ende) anfügen kann und zu jedem Zeitpunkt das Objekt löschen kann, welches (von den vorhandenen) als erstes eingefügt wurde.
Stack und Queue unterscheiden sich also nach dem Auswahlprinzip für das zu löschende Element:

Stapel (Stack)	Warteschlange (Queue)
last in first out	first in first out
LIFO	**FIFO**

Typische Anwendungen für Warteschlangen in der Informatik sind:

- Druckerschlange
- Tastatureingabe
- Multitask-Systeme
- Breitensuche

Zur Beschreibung einer Queue als ADT verwenden wir wieder eine Typvariable `E` für Datentyp der Elemente, die in der Queue gehalten werden sollen. Ein Interface `Queue<E>` sollte Hauptmethoden folgenden Typs enthalten:

```
void enqueue(E o); // Einfuegen des Objekts o von Typ E an letzte Position
E    dequeue();    // Entfernen des vordertsten Objekts plus Rueckgabe
                   // Fehlermeldung, wenn Queue schon leer ist
```

sowie als Ergänzung die Methoden:

```
int     size();     // Anzahl der Objekte in der Queue
boolean isEmpty();  // Queue leer?
E       front();    // vorderstes Objekt ausgeben, ohne Loeschen
```

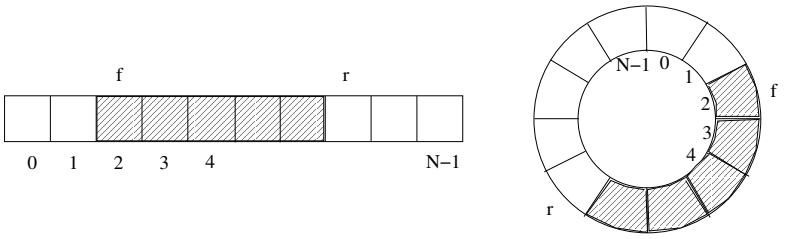
Das in `java.util` definierte Interface `Queue<E>` verwendet zum Teil andere Methodennamen. In den alten Versionen (ohne Generics) setzt man an Stelle von `E` den Typ `Object`. Ähnlich wie im Fall von Stacks kann man sich bei Queues prinzipiell für Implementierungen mit Arrays und mit verketteten Listen entscheiden.

Implementierungen von Queues mit Arrays

Variante a) In der einfachsten Version wird ein Array Q der Default-Größe N oder einer als Parameter übergebenen Größe n angelegt. Zwei Zahlen f und r repräsentieren die erste belegte Zelle und die erste freie Zelle hinter dem belegten Teil. Der Spezialfall einer leeren Queue wird durch $f = r$ repräsentiert, d.h. die Konstruktoren setzen f und r auf 0. Zum Einfügen wird in $Q[r]$ die Referenz auf das einzufügende Objekt eingetragen und r um 1 erhöht. Zum Entfernen wird $Q[f]$ (das ist das zuerst eingefügte Element) zurückgegeben und f um 1 erhöht. Die aktuelle Größe ergibt sich einfach durch $r - f$. Bei dieser Queue-Implementierung können (im Gegensatz zur Stack-Implementierung mit

Feldern fester Größe) auch dann Überlauf-Probleme auftreten, wenn nur sehr wenige Objekte in der Warteschlange stehen. Der Grund für diesen Unterschied erklärt sich daraus, dass beim Stack freigegebenener Speicherplatz für neue Einträge genutzt werden kann (die Zahl top kann steigen und fallen, während für die Queue die Zahl r mit jedem neuen Eintrag steigt). Die Implementierung mit dynamischen Arrays (Vektoren) könnte in diesem Fall den Überlauf verhindern, aber keine dynamische Anpassung des Speicherbedarfs sicher stellen.

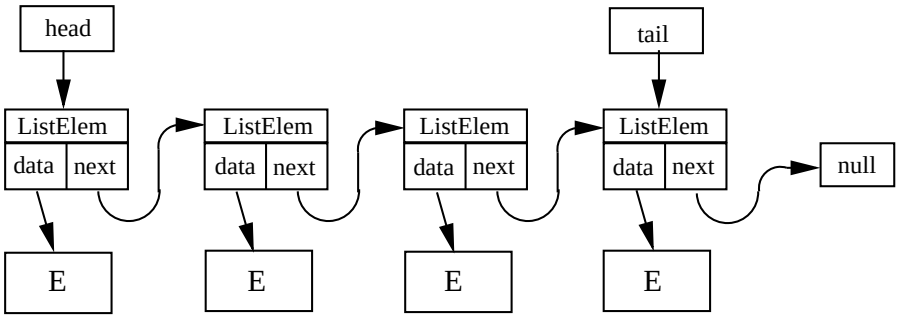
Variante b) Die Wiederverwendung von freigewordenen Speicherplatz wird durch sogenannte **zyklische Arrays** möglich. Man kann sich das als ein Feld von Daten vorstellen, das zu einem Ring geschlossen wurde. Die Idee ist in der folgenden Abbildung illustriert.



Natürlich gibt es im Rechner keinen Speicher dieser Art, man muss den Ringschluss durch Indexarithmetik modulo N simulieren. Die Zahlen f und r repräsentieren nun in Uhrzeigerrichtung die erste belegte Zelle und die erste freie Zelle hinter dem belegten Teil. Beim Einfügen muss r und beim Entfernen muss f um 1 modulo N erhöht werden. Ein Überlauffehler tritt immer dann auf, wenn bei einer Einfügeoperation die Situation $f == r$ eintritt. Wenn die Situation $f == r$ durch eine Löschoperation entsteht, ist das zulässig, die Anwendung einer Löschoperation bei $f == r$ führt aber zu einem Fehler. Man kann beobachten, dass bei dieser Implementierung immer mindestens eine Zelle unbelegt bleiben muss, da man nur durch die Werte von f und r nicht zwischen einem voll belegtem und einem leeren zyklischen Array unterscheiden kann. Durch Kombination von zyklischen Arrays mit dynamischer Größenanpassung werden Überlauffehler verhindert.

Implementierung mit verkettenen Listen

Wie schon bei Stacks wird hier mit der Klasse `ListElem<E>` gearbeitet. Der wesentliche Unterschied zur Stack-Implementierung besteht darin, dass eine Klasse `ListQueue<E>` zwei Referenzen auf Listenelemente benötigt, das erste und das letzte in der verketten Liste. Wir nennen sie `head` und `tail` (zur Erinnerung, bei Stacks brauchte man nur das erste Element).



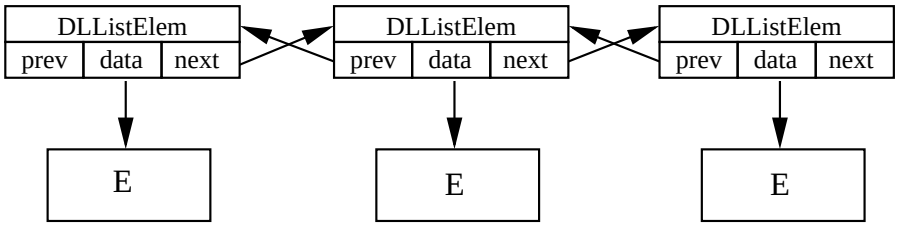
Bleibt die Frage, wie man einfügen und löschen kann. Es ist leicht zu sehen, dass man auf beiden Seiten einfügen könnte. Zuerst wird ein neues Listenelement `le` erzeugt und die Referenz `data` auf das einzufügende Objekt gesetzt. Will man `le` auf der rechten Seite (tail) einfügen, so muss `tail.next` auf `le` und `le.next` auf `null` gesetzt werden und zum Abschluss setzt man `tail=le;`. Will man `le` auf der linken Seite (head) einfügen, so muss `le.next` auf `head` und `head=le;` gesetzt werden. Dagegen ist das Löschen nur auf der Seite von `head` möglich. Deahlb wird bei einer Queue-Implementierung auf der `tail`-Seite eingefügt und auf der `head`-Seite gelöscht.
Aus dieser Beschreibung folgt, dass alle Queue-Operationen auf verketteten Listen in konstanter Zeit ausgeführt werden.

Doppelt verkettete Listen (Double-Ended Queues)

Eine Double-Ended Queue (Deque, gesprochen Deck) ist eine geordnete Datenstruktur, bei der man am Anfang und am Ende der Ordnung jeweils einfügen und löschen kann. Ein solcher ADT ist bei Verwendung einer Typvariablen `E` durch die folgenden Methoden gekennzeichnet:

```
void insertFirst(E o);
void insertLast(E o);
E    removeFirst();
E    removeLast();
E    first();   \\erstes Objekt zurueckgeben ohne zu loeschen
E    last();    \\letztes Objekt zurueckgeben ohne zu loeschen
```

Damit ist klar, dass man mit einer Deque-Implementierung sowohl einen Stack als auch eine Queue simulieren kann. Die Implementierung kann wieder mit zyklischen, dynamischen Arrays erfolgen oder mit doppelt verketteten Listen. Die Idee zu dieser Struktur besteht darin, zu jeder `next`-Referenz von einem Listenelement auf das nächste auch die Rückreferenz `prev` zu speichern. Die folgende Abbildung illustriert den Aufbau eines Listenelements in einer doppelt verketteten Liste:

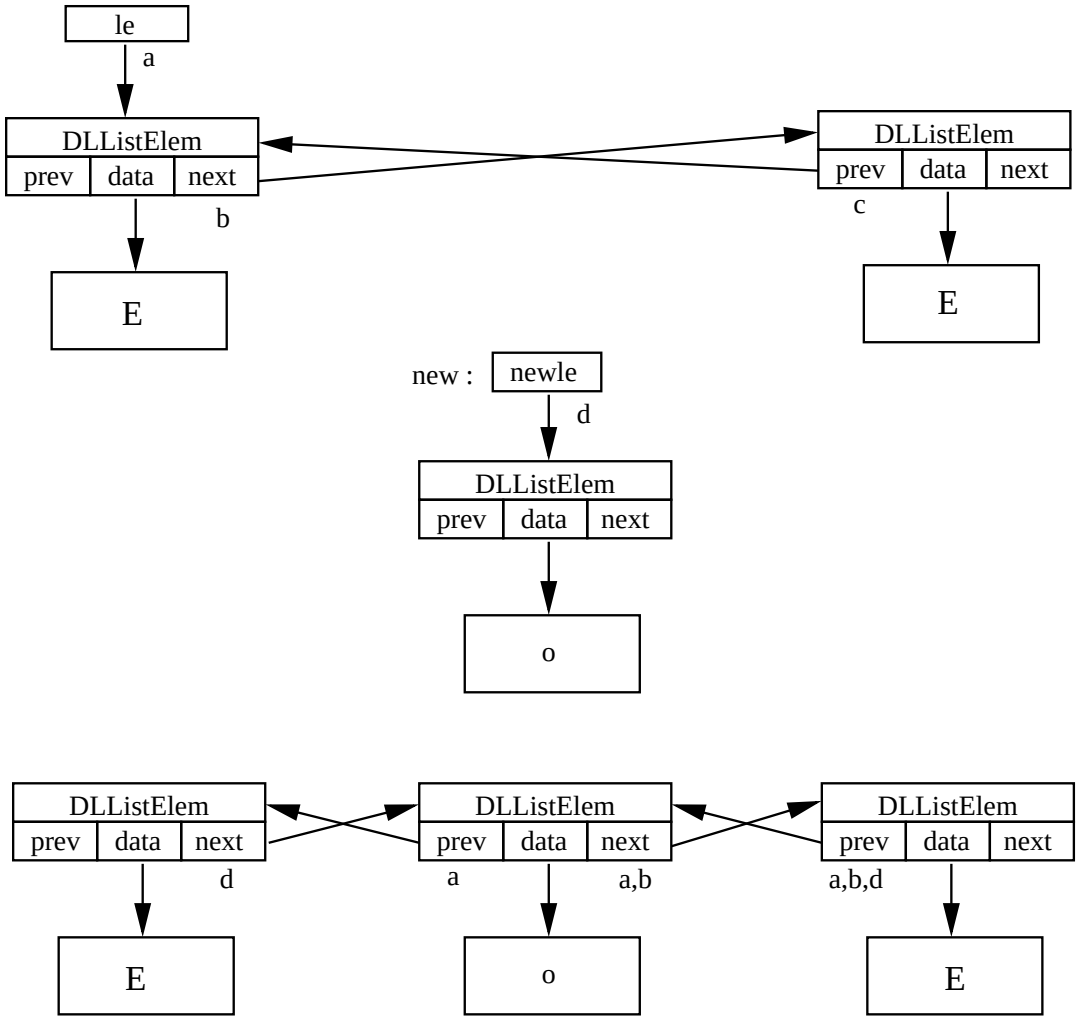


Ein Objekt von `DLListElem<E>` hat also die Attribute

```
E          data;
DLListElem next;
DLListElem prev;
```

Die Einfüge- und Löschoperationen funktioniern, wie schon bei den Queues beschrieben, in konstanter Zeit. Darüber hinaus kann man bei doppelt verketteten Listen auch an beliebigen Stellen in konstanter Zeit Objekte einfügen und wieder löschen. Man braucht zum Einfügen nur eine Referenz auf das Listenelement, hinter dem eingefügt werden soll, und zum Löschen eine Referenz auf das zu streichende Listenelement.
Die folgende Abbildung beschreibt die Schritte zur Ausführung einer Einfügeoperation eines Objekts

`o` hinter ein gegebenes `DLListElem le`. Zuerst muss ein neues `DLListElem newle` angelegt werden, dessen `data`-Referenz auf `o` gesetzt wird. Danach müssen die `prev`- und `next`-Referenzen von `newle`, die `prev`-Referenz des alten Nachfolgers von `le` und zuletzt die `next`-Referenz von `le` neu gesetzt werden. Die Buchstaben unter den entsprechenden Feldern zeigen an, welche von den alten Referenzen gebraucht werden, um die neue Referenz festzulegen.



Datenstrukturen, deren Einträge wie in verketteten Listen oder in doppelt verketteten Listen intern durch eine Zeigerstruktur vernetzt sind, werden auch als *Geflechte* bezeichnet. In einer der nächsten Vorlesungen werden wir binäre Bäume als einen weiteren Geflechtstyp kennenlernen.