11 Abstrakte Datentypen

Erinnerung:

- Abstrakter Datentyp spezifiziert nur die Operationen
- Implementierung und andere Details sind verborgen

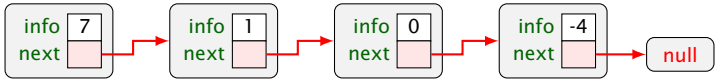
Dieses ist ein sehr puristischer Ansatz. Im folgenden werden wir häufig nicht ganz so streng sein, und manchmal Zugriff auf die Datenstruktur auch über direkte Manipulation von Attributen gestatten.

11.1 Listen

Nachteil von Feldern:

- feste Größe
- Einfügen neuer Elemente nicht möglich
- Streichen ebenfalls nicht

Idee: Listen



Listen – Version A

`info` : Datenelement der Liste;
`next` : Verweis auf nächstes Element;
`null` : leeres Objekt.

Eigentlich ist die Liste, die wir hier implementieren, kein abstrakter Datentyp. Die Operationen `insert` und `delete` benötigen eine Referenz auf ein Listenelement hinter dem eingefügt bzw. gelöscht wird. Dies erlaubt keine gute Datenkapselung.

Operationen:

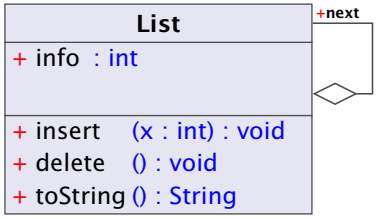
`void insert(int x)` : fügt neues `x` hinter dem aktuellen (ersten) Element ein;

`void delete()` : entfernt Knoten hinter dem aktuellen (ersten) Element;

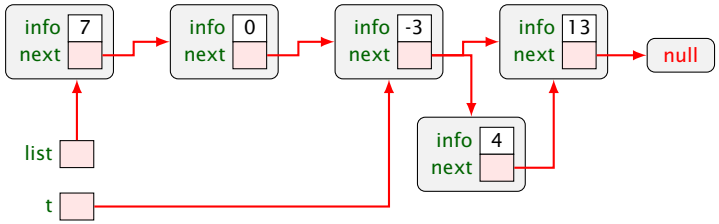
`String toString()` : liefert eine `String`-Darstellung.

Modellierung

Modellierung als UML-Diagramm:

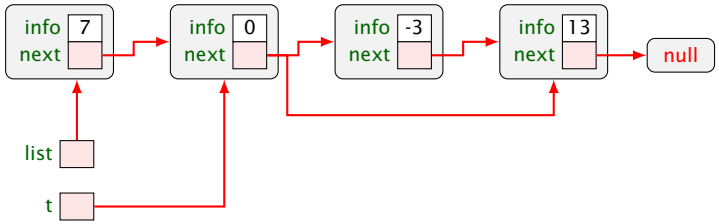


Listen – Insert



t.insert(4)

Listen – Delete



t.delete()

11.1 Listen

Weitere Operationen:

- Liste auf Leerheit testen
- Neue Listen erzeugen ($\Longrightarrow$ Konstruktoren)
  - z.B. eine einelementige Liste
  - eine bestehende Liste verlängern
- Umwandlung zwischen Listen und Feldern...

Das `null`-Objekt versteht keinerlei Objektmethoden; da wir `null` als leere Liste interpretieren, müssen wir uns etwas einfallen lassen...

Listen – Implementierung

```
1  public class List {
2      public int info;
3      public List next;
4
5      // Konstruktoren:
6      public List (int x, List l) {
7          info = x;
8          next = l;
9      }
10     public List (int x) {
11         info = x;
12         next = null;
13     }
14 // continued...
```

Listen – Implementierung

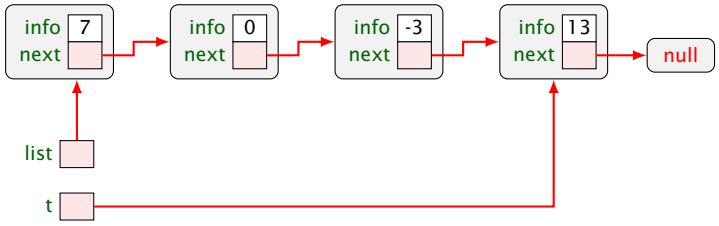
```
15      // Objekt-Methoden:
16      public void insert(int x) {
17          next = new List(x,next);
18      }
19      public void delete() {
20          if (next != null)
21              next = next.next;
22      }
23      public String toString() {
24          String result = "[" + info;
25          for(List t = next; t != null; t = t.next)
26              result = result + ", " + t.info;
27          return result + "]";
28      }
29  // continued...
```


Erläuterungen

- Die Attribute sind `public` und daher beliebig einsehbar und modifizierbar; sehr fehleranfällig.
- `insert()` legt einen neuen Listenknoten an, und fügt ihn hinter dem aktuellen Knoten ein.
- `delete()` setzt den aktuellen `next`-Verweis auf das übernächste Element um.

Achtung:
Wenn `delete()` mit dem letzten Listenelement aufgerufen wird, zeigt `next` auf `null`; wir tun dann nichts...

Listen – Delete



`t.delete()`

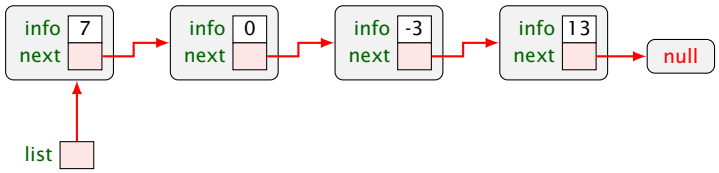
Erläuterungen

Weil Objektmethoden nur für von `null` verschiedene Objekte aufgerufen werden können, kann die leere Liste nicht mittels `toString()` als `String` dargestellt werden.

Der Konkatenations-Operator `+` ist so schlau, vor Aufruf von `toString()` zu überprüfen, ob ein `null`-Objekt vorliegt. Ist das der Fall, wird "null" ausgegeben.

Für eine andere Darstellung benötigen wir eine Klassenmethode `toString(List l)`;

Listen – toString()




```
write("" + list);
```

liefert: „[7, 0, -3, 13]“

Listen – toString()


```
write("" + list);
```

liefert: „null“

Listen – Implementierung

```
30      // Klassen-Methoden:
31      public static boolean isEmpty(List l) {
32          return (l == null);
33      }
34      public static String toString(List l) {
35          if (l == null)
36              return "[]";
37          else
38              return l.toString();
39      }
40  // continued...
```

Der Aufruf erfolgt dann über `List.isEmpty(a)` bzw. `List.toString(a)` für eine Liste `a`. Leider funktioniert letzteres nicht zusammen mit dem Konkatenationsoperator. Über diesen wird weiterhin „null“ ausgegeben.

Listen – Implementierung

```
41      public static List arrayToList(int[] a) {
42          List result = null;
43          for(int i = a.length-1; i >= 0; --i)
44              result = new List(a[i], result);
45          return result;
46      }
47      public int[] listToArray() {
48          List t  = this;
49          int n   = length();
50          int[] a = new int[n];
51          for(int i = 0; i < n; ++i) {
52              a[i] = t.info;
53              t = t.next;
54          }
55          return a;
56      }
57  // continued...
```

Listen – Implementierung

- Damit das erste Element der Ergebnisliste `a[0]` enthält, beginnt die Iteration in `arrayToList()` beim letzten Element.
- `listToArray()` ist als Objekt-Methode realisiert und funktioniert darum nur für nicht-leere Listen.
- Wir benötigen die Länge einer Liste:

```
58     private int length() {
59         int result = 1;
60         for(List t = next; t != null; t = t.next)
61             result++;
62         return result;
63     }
64 } // end of class List
```


Listen – Implementierung

- Weil `length()` als `private` deklariert ist, kann es nur von den Methoden der Klasse `List` benutzt werden.
- Damit `length()` auch für `null` funktioniert, hätten wir analog zu `toString()` auch noch eine Klassen-Methode `int length(List l)` definieren können.
- Diese Klassen-Methode würde uns ermöglichen, auch eine Klassen-Methode `static int[] listToArray (List l)` zu definieren, die auch für leere Listen definiert ist.

> Es ist eine generelle Stilfrage ob man eine leere Liste als `null` implementieren sollte. Die meisten Java-Bibliotheken nutzen ein spezielles Objekt, dass eine leere Liste/Collection etc. repräsentiert. Dann kann man z.B. immer `a.toString()` aufrufen anstatt `List.toString(a)` etc.

Mergesort – Sortieren durch Mischen

Mergesort ist ein schneller Sortieralgorithmus der auf auf der Mischoperation beruht.

John von Neumann (1945)

Mergesort – Sortieren durch Mischen

Die Mischoperation

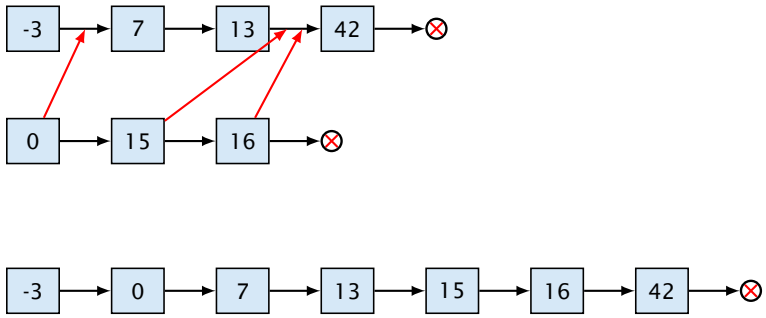
Input: zwei sortierte Listen

Output: eine gemeinsame sortierte Liste

Später bauen wir damit einen Sortieralgorithmus...

Beispiel – Mischen

Hier benutzen wir das Symbol ⊗ für das `null`-Objekt.



Mergesort – Sortieren durch Mischen

Idee:

- Konstruiere sukzessive die Ausgabeliste aus den Argumentlisten.
- Um das nächste Element für die Ausgabe zu finden, vergleichen wir die beiden kleinsten Elemente der noch verbliebenen Inputlisten.
- Falls m und n die Längen der Argumentlisten sind, sind offenbar maximal nur $m + n - 1$ Vergleiche nötig.

Beispiel – Mischen

Animation ist nur in der Vorlesungsversion der Folien vorhanden.

Mergesort – Implementierung

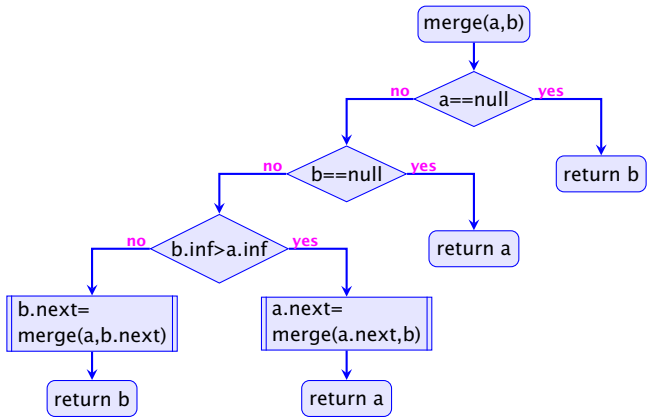
Rekursive Implementierung

- Falls eine der beiden Listen a und b leer ist, geben wir die andere aus.
- Andernfalls gibt es in jeder der beiden Listen ein erstes (kleinstes) Element.
- Von diesen beiden Elementen nehmen wir ein kleinstes.
- Dahinter hängen wir die Liste, die wir durch Mischen der verbleibenden Elemente erhalten. . .

Mergesort – Implementierung

```
1  public static List merge(List a, List b) {
2      if (b == null)
3          return a;
4      if (a == null)
5          return b;
6      if (b.info > a.info) {
7          a.next = merge(a.next, b);
8          return a;
9      } else {
10         b.next = merge(a, b.next);
11         return b;
12     }
13 }
```


Kontrollfluss



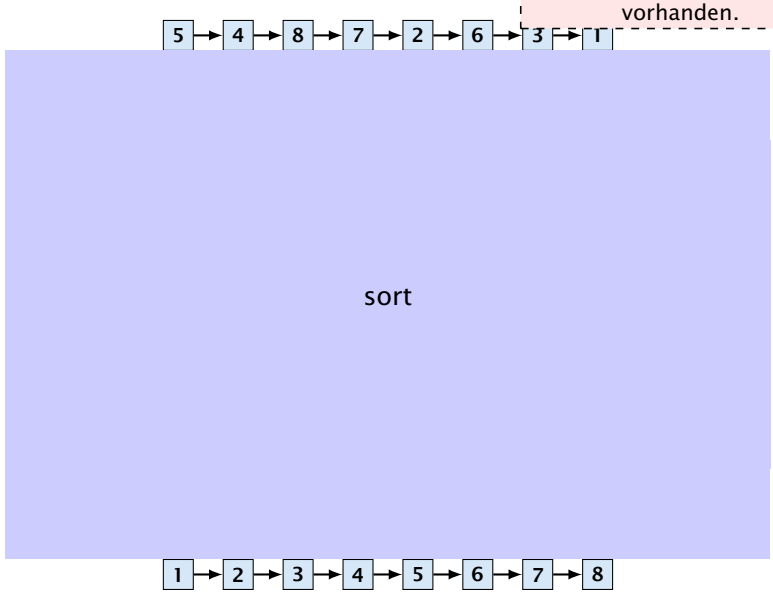
Mergesort

Sortieren durch Mischen:

1. Teile zu sortierende Liste in zwei Teillisten;
2. sortiere jede Hälfte für sich;
3. mische die Ergebnisse!

Mergesort

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



Mergesort – Implementierung

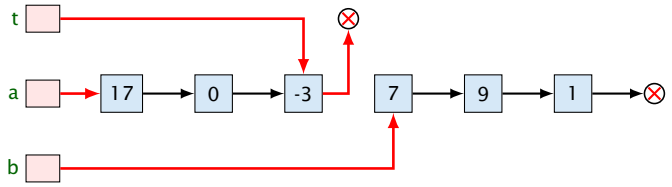
```
1 public static List sort(List a) {
2     if (a == null || a.next == null)
3         return a;
4     List b = a.half(); // Halbiere!
5     a = sort(a);
6     b = sort(b);
7     return merge(a,b);
8 }
```


Mergesort – Implementierung

```
1 public List half() {
2     int n = length();
3     List t = this;
4     for (int i = 0; i < n/2-1; i++)
5         t = t.next;
6     List result = t.next;
7     t.next = null;
8     return result;
9 }
```


Halbieren

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



a.half()

Mergesort – Analyse

- Sei $V(n)$ die Anzahl der Vergleiche, die Mergesort maximal zum Sortieren einer Liste der Länge n benötigt.
  Dann gilt:

$$\begin{aligned} V(1) &= 0 \\ V(2n) &\leq 2 \cdot V(n) + 2 \cdot n \end{aligned}$$

- Für $n = 2^k$, sind das dann nur $k \cdot n = n \log_2 n$ Vergleiche!!!

Dies ist wesentlich effizienter als die Methode „Sortieren durch Einfügen“, die wir vorher kennengelernt haben.

Mergesort – Bemerkungen

Achtung:

- Unsere Funktion `sort()` zerstört ihr Argument!
- Alle Listenknoten der Eingabe werden weiterverwendet.
- Die Idee des Sortierens durch Mischen könnte auch mithilfe von Feldern realisiert werden (wie?)
- Sowohl das Mischen wie das Sortieren könnte man statt rekursiv auch iterativ implementieren (wie?)

11.2 Keller (Stacks)

Operationen:

`boolean isEmpty()`	:	testet auf Leerheit;
`int pop()`	:	liefert oberstes Element;
`void push(int x)`	:	legt `x` oben auf dem Keller ab;
`String toString()`	:	liefert eine String-Darstellung

Weiterhin müssen wir einen leeren Keller anlegen können.

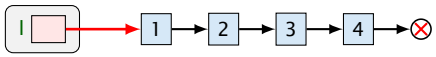
Modellierung Stack

Stack
+ Stack ()
+ isEmpty () : boolean
+ push (x : int) : void
+ pop () : int
+ toString () : String

Stack via List

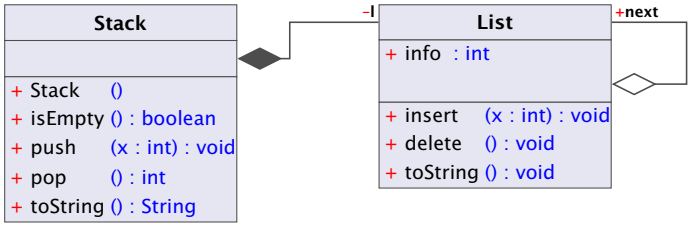
Idee

- Realisiere `Stack` mithilfe einer Liste:



- Das Attribut `l` zeigt auf das oberste Kellerelement.

Modellierung Stack via List



Die gefüllte Raute bezeichnet eine Komposition. Die Liste existiert nur solange wie der Stack (d.h. wird üblicherweise durch diesen erzeugt und zerstört). Außerdem kann die Liste nur Teil eines Stacks sein.

Stack – Implementierung

```
1  public class Stack {
2      private List l;
3      // Konstruktor :
4      public Stack() {
5          l = null;
6      }
7      // Objektmethoden :
8      public boolean isEmpty() {
9          return l == null;
10     }
11     // continued...
```


Stack – Implementierung

```
12     public int pop() {
13         int result = l.info;
14         l = l.next;
15         return result;
16     }
17     public void push(int a) {
18         l = new List(a,l);
19     }
20     public String toString() {
21         return List.toString(l);
22     }
23 } // end of class Stack
```


Bemerkungen

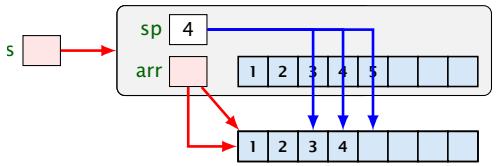
- Implementierung ist sehr einfach;
- nutzt gar nicht alle Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Keller mithilfe eines Feldes und eines Stackpointers, der auf das oberste Element zeigt.
- Läuft das Feld über, ersetzen wir es durch ein größeres.

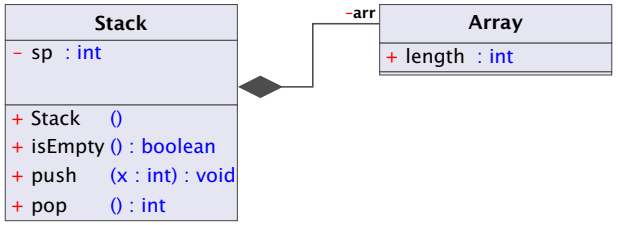
Stack via Array

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



s.push(5)

Modellierung Stack



Implementierung

```
1  public class Stack {
2      private int sp;
3      private int[] arr;
4      // Konstruktoren:
5      public Stack() {
6          sp  = -1;
7          arr = new int[4];
8      }
9      // Objekt-Methoden:
10     public boolean isEmpty() {
11         return sp < 0;
12     }
13     // continued...
```


Implementierung

```
14     public int pop() {
15         return arr[sp--];
16     }
17     public void push(int x) {
18         ++sp;
19         if (sp == arr.length) {
20             int[] b = new int[2*sp];
21             for (int i = 0; i < sp; ++i) b[i] = arr[i];
22             arr = b;
23         }
24         arr[sp] = x;
25     }
26     public String toString() {...}
27 } // end of class Stack
```

11.2 Keller (Stacks)

Nachteil:

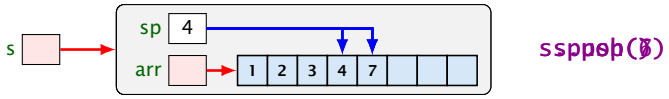
- Es wird zwar neuer Platz allokiert, aber nie welcher freigegeben.

Erste Idee:

- Sinkt der Pegel wieder auf die Hälfte, geben wir diese frei...

Stack via Array

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



sspppop()

11.2 Keller (Stacks)

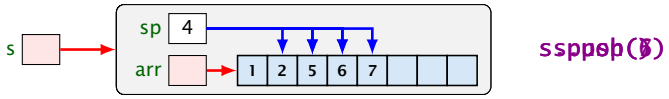
- Im schlimmsten Fall müssen bei jeder Operation sämtliche Elemente kopiert werden.

Zweite Idee:

- Wir geben erst frei, wenn der Pegel auf ein Viertel fällt – und dann auch nur die Hälfte!

Stack via Array

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



sspppop()

11.2 Keller (Stacks)

Beobachtung:

- Vor jedem Kopieren werden mindestens halb so viele Operationen ausgeführt wie Elemente kopiert werden.
- Gemittelt über die gesamte Folge der Operationen werden pro Operation maximal zwei Zahlen kopiert (↑amortisierte Aufwandsanalyse)

Implementierung

```
1  public int pop() {
2      int result = arr[sp];
3      if (sp == arr.length/4 && sp >= 2) {
4          int[] b = new int[2*sp];
5          for(int i = 0; i < sp; ++i)
6              b[i] = arr[i];
7          arr = b;
8      }
9      sp--;
10     return result;
11 }
```


11.3 Schlangen (Queues)

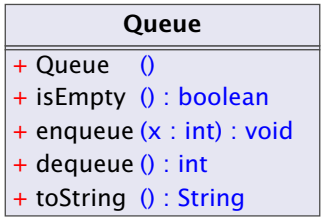
(Warte-) Schlangen verwalten ihre Elemente nach dem FIFO-Prinzip (First-In-First-Out).

Operationen:

`boolean isEmpty()`	:	testet auf Leerheit;
`int dequeue()`	:	liefert erstes Element;
`void enqueue(int x)`	:	reiht x in die Schlange ein;
`String toString()`	:	liefert eine String-Darstellung.

Weiterhin müssen wir eine leere Schlange anlegen können.

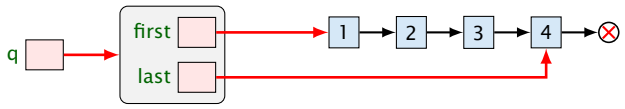
Modellierung Queue



Queue via List

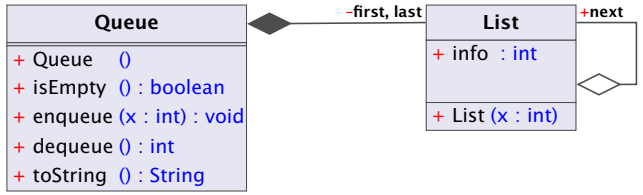
Erste Idee:

- Realisiere Schlange mithilfe einer Liste:



- `first` zeigt auf das nächste zu entnehmende Element;
- `last` zeigt auf das Element hinter dem eingefügt wird;

Modellierung: Queue via List



Objekte der Klasse `Queue` enthalten zwei Verweise auf Objekte der Klasse `List`.

Queue – Implementierung

```
1  public class Queue {
2      private List first, last;
3      // Konstruktor:
4      public Queue() {
5          first = last = null;
6      }
7      // Objekt-Methoden:
8      public boolean isEmpty() {
9          return List.isEmpty(first);
10     }
11     // continued...
```


Queue – Implementierung

```
12     public int dequeue() {
13         int result = first.info;
14         if (last == first) last = null;
15         first = first.next;
16         return result;
17     }
18     public void enqueue(int x) {
19         if (first == null)
20             first = last = new List(x);
21         else {
22             last.insert(x);
23             last = last.next;
24         }
25     }
26     public String toString() {
27         return List.toString(first);
28     }
29 } // end of class Queue
```

Bemerkungen

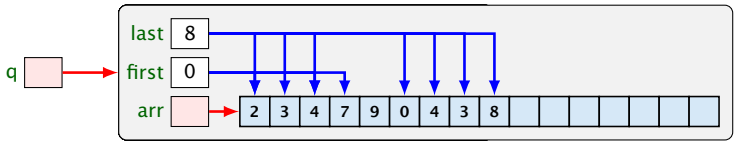
- Implementierung ist wieder sehr einfach;
- nutzt mehr Features von `List`;
- **Nachteil:** Die Listenelemente sind evt. über den gesamten Speicher verstreut:
  $\Longrightarrow$ schlechtes Cache-Verhalten!

Zweite Idee:

- Realisiere Schlange mithilfe eines Feldes und zweier Pointer, die auf erstes bzw. letztes Element zeigen.
- Läuft das Feld über, ersetzen wir es durch ein größeres.

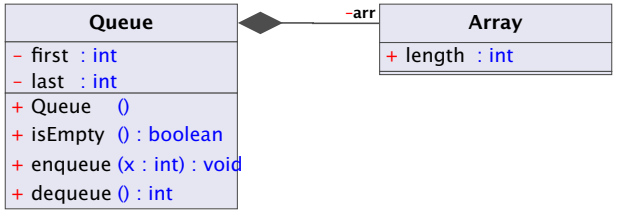
Queue via Array

Animation ist nur in der Vorlesungsversion der Folien vorhanden.



`q.dequeue()`

Modellierung: Queue via Array



Implementierung

```
1  public class Queue {
2      private int first, last;
3      private int[] arr;
4      // Konstruktor:
5      public Queue() {
6          first = last = -1;
7          arr = new int[4];
8      }
9      // Objekt-Methoden:
10     public boolean isEmpty() { return first == -1; }
11     public String toString() {...}
12     //continued...
```

Implementierung von enqueue()

- Falls die Schlange leer war, muss `first` und `last` auf `0` gesetzt werden.
- Andernfalls ist das Feld `a` genau dann voll, wenn das Element `x` an der Stelle `first` eingetragen werden sollte.
- In diesem Fall legen wir ein Feld doppelter Größe an.

  Die Elemente `a[first],..., a[a.length-1]`, `a[0], a[1],..., a[first-1]` kopieren wir nach `b[0],..., b[a.length-1]`.
- Dann setzen wir `first = 0; last = a.length; a = b;`
- Nun kann `x` an der Stelle `a[last]` abgelegt werden.

Implementierung

```
13      public void enqueue(int x) {
14          if (first == -1) {
15              first = last = 0;
16          } else {
17              int n = arr.length;
18              last = (last + 1) % n;
19              if (last == first) {
20                  int[] b = new int[2*n];
21                  for (int i = 0; i < n; ++i)
22                      b[i] = arr[(first + i) % n];
23                  first = 0;
24                  last  = n;
25                  arr   = b;
26              }
27          } // end if and else
28          arr[last] = x;
29      }
```


Implementierung von dequeue()

- Falls nach Entfernen von `a[first]` die Schlange leer ist, werden `first` und `last` auf `-1` gesetzt.
- Andernfalls wird first um `1` (modulo der Länge von `arr`) inkrementiert.

Falls danach höchstens noch $n/4$ Elemente da sind, werden diese an die Positionen `b[0],...,b[num-1]` in einem neuen Array kopiert.

Implementierung

```
30  public int dequeue() {
31      int result = arr[first];
32      if (last == first) {
33          first = last = -1;
34          return result;
35      }
36      int n = arr.length;
37      first = (first+1) % n;
38      // Anzahl der Elemente ist (last-first) mod n + 1
39      // aber % in Java ist keine modulo-Operation
40      int num = (last-first+n) % n + 1;
41      // continued...
```

Implementierung

```
42      if (num > 1 && num <= n/4) {
43          int[] b = new int[n/2];
44          for (int i = 0; i < num; ++i)
45              b[i] = arr[(first + i) % n];
46          first = 0;
47          last = num - 1;
48          arr = b;
49      }
50      return result;
51  }
```


11 Abstrakte Datentypen

Zusammenfassung

- Der Datentyp `List` ist nicht sehr abstrakt, dafür extrem flexibel (gut für rapid prototyping)
- Für die nützlichen (eher) abstrakten Datentypen `Stack` und `Queue` lieferten wir zwei Implementierungen. Eine sehr einfache, und eine cache-effiziente.
- **Achtung:** oft werden bei diesen Datentypen noch weitere Operationen zur Verfügung gestellt.