IIF
INSTITUT FÜR
INFORMATIK
FREIBURG

Vorlesung Informatik 2
Algorithmen und Datenstrukturen

(05 – Elementare Datenstrukturen)

Prof. Dr. Susanne Albers

Lineare Listen (1)

- Lineare Anordnung von Elementen eines Grundtyps
  (elementarer Datentyp oder Grundklasse, etwa Object)
- Grundtyp ist oft weiter strukturiert, etwa

```
class Grundklasse {
        int key; // eindeutiger Schluessel
        Infoklasse info; // weitere Informationen
}
```

- Bsp: Liste von Zahlen, Zeichenkette, …
- Operationen (Methoden)
  - Initialisieren (leere Liste, . . .)
  - Länge bestimmen
  - Suchen (inhaltsbasiert), kann aktuelles Element festlegen
  - Positionieren (nach laufender Nummer), . . .
  - Einfügen, an gegebener oder aktueller Position
  - Entfernen, . . .
  - Verketten, Ausgeben, Teilliste bilden, Umkehren, . . .

Lineare Listen (2)

Formalere Definition für Lineare Listen:

- $X \neq \emptyset$; Grundmenge
- $L =< a_1,...,a_n >$ mit $a_i \in X$

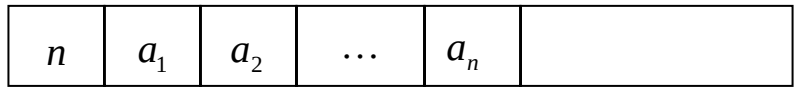
  ist linear angeordnete Liste mit Elementen aus X
- $<>$ ist die leere Liste
- Vorgänger von a_i ist a_{i-1}, falls $i \neq 1$
- Nachfolger von a_i ist a_{i+1}, falls $i \neq n$
- Länge von $L =< a_1,...,a_n >$ ist $|L| = n;\quad |<>| = 0$
- L.entferne(p) bewirkt $L' =< a_1,...,a_{p-1},a_{p+1},...,a_n >$ mit $|L'| = n-1$
- L.einfüge(a, p) bewirkt $L' =< a_1,...,a_{p-1},a,a_p,a_{p+1},...,a_n >$ mit $|L'| = n+1$
- L.suche(a) liefert true, falls a in L enthalten ist und false sonst

  (Alternative: liefert die erste Position p, an der a vorkommt, oder -1)

  das p-te Element wird aktuelles Element
- L.zugriff(p) liefert a_p, falls $1 \leq p \leq n$, und ist undefiniert sonst (Exception)

Implementierungen für Lineare Listen (1)

Mit Hilfe von Arrays:

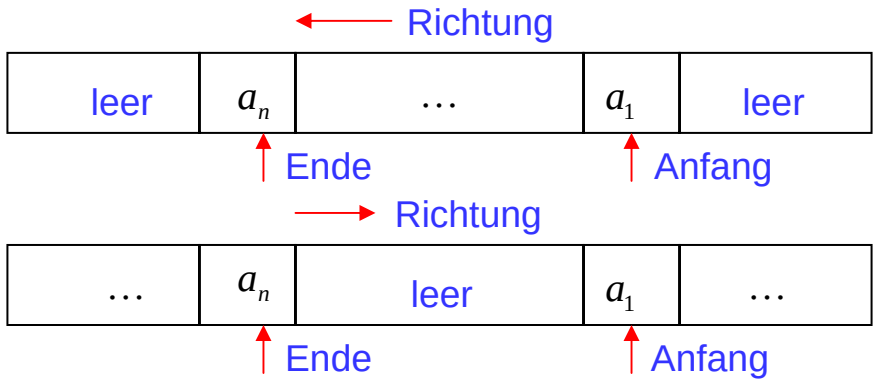


Eigenschaften dieser Lösung:

- Maximale Anzahl von Elementen vorgegeben (Länge des Arrays)
  Abhilfe: Dynamisierung
- Schlechte Speicherplatzausnutzung für kurze Listen
- Teure Verschiebeoperationen beim Einfügen/Entfernen von Elementen
+ Direkter Zugriff auf feste Positionen

Implementierungen für Lineare Listen (2)

Zyklische Speicherung mit Hilfe von Arrays:



Besondere Eigenschaften dieser Lösung:

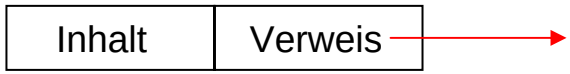
+ Einfügen/Entfernen an den Listenenden in konstanter Zeit

+ Umkehren der Liste in konstanter Zeit

Implementierungen für Lineare Listen (3)

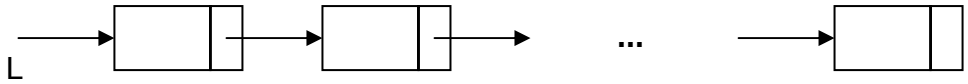
Mit Hilfe einfach verketteter (verlinkter) Elemente:

• speichere mit jedem Element einen Verweis auf das nächste oder null, falls das Element nicht existiert:

Knoten: 


```
class node {
    Grundklasse content;
    node next;
}
```

• Liste L: Verweis auf das erste Element der Kette



Besondere Eigenschaften dieser Lösung:

\+ Länge der Liste kann sich beliebig ändern

\- Kein direkter Zugriff auf Element an der Position p mehr möglich; nur Durchlaufen der Liste in O(p)

Einfach verkettete Lineare Listen (1)

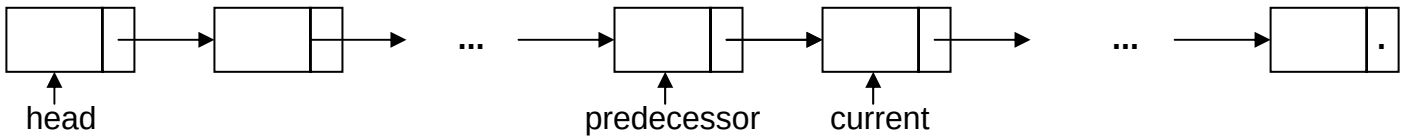
Einfaches Beispiel für Listen aus ganzen Zahlen:

```
public interface IntList {
        public boolean empty ();          // leer?
        public void first ();             // erstes Element wird aktuell
        public void last ();              // letztes Element wird aktuell
        public boolean hasCurrent ();     // aktuelles Element bekannt?
        public int get ();                // liefert aktuelles Element
        public void insert (int i);       // nach aktuellem El. einfuegen
        public boolean search (int i);    // Suchen nach i
        public boolean setPos (int p);    // setze aktuelle Position auf p
        public boolean insert (int i, int p); // als p-tes El. einfuegen
        public boolean delete (int p);    // p-tes Element loeschen
        public void remove ();            // aktuelles Element loeschen
        public void print ();             // Ausgabe aller Elemente
}
```

Einfach verkettete Lineare Listen (2)

```
class IntNode {                                  // die Knotenklasse
        private int content;                     // Inhalt
        private IntNode next;                    // Nachfolger
        public IntNode (int i, IntNode n){ // Konstruktor
                content = i;                     // setzt Inhalt
                next = n;                        // und Nachfolger
        }
        public int getContent (){                // gibt Inhalt raus
                return content;
        }
        public void setContent (int i){          // setzt Inhalt
                content = i;
        }
        public IntNode getNext (){               // gibt Nachfolger raus
                return next;
        }
        public void setNext (IntNode n){         // setzt Nachfolger
                next = n;
        }
}
```

Einfach verkettete Lineare Listen (3)



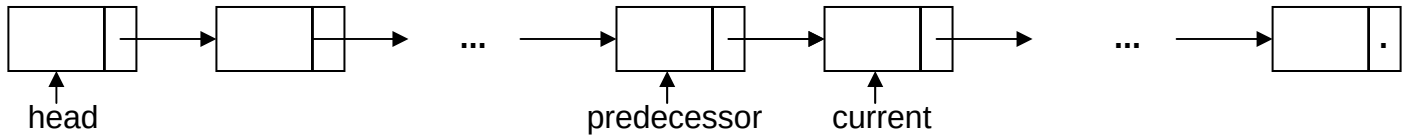

```
class LinkedIntList implements IntList {
        private IntNode Head, current, predecessor;
        public LinkedIntList (){      // Konstruktor
                Head = predecessor = current = null;
        }
        public boolean empty (){      // leer?
                return Head == null;
        }
        public void first (){         // erstes Element
                current = Head;       // wird aktuell
                predecessor = null;
        }
// ...
```

Einfach verkettete Lineare Listen (4)

```
public void last (){                    // letztes Element
        IntNode h; // wird aktuell
        if (current == null) h = Head;// h auf Kopf oder
        else h = current.getNext ();  // Nachfolger von current
        while (h != null){
                predecessor = current;
                current = h;
                h = current.getNext ();
        }
}
public boolean hasCurrent () {
        return current != null;
}
public int get () {                     // liefert aktuelles El.
        return current.getContent ();
}
// ...
```

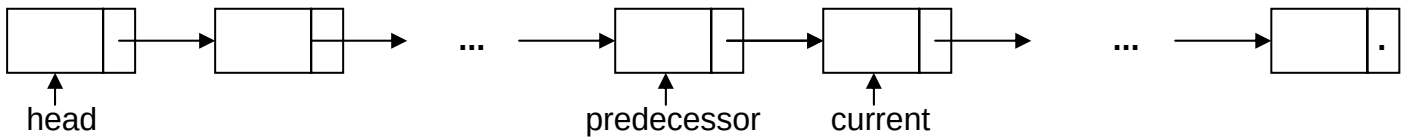
Einfügen

```
public void insert (int i){                    // nach dem aktuellen
        predecessor = current;                 // Element einfuegen!
        if (current == null)
                current = Head = new IntNode (i, Head);
        else { current = new IntNode (i, predecessor.getNext ());
                predecessor.setNext (current);
        }
}
```



Suchen

```
public boolean search (int i){              // Suchen nach i
        current = Head;
        predecessor = null;
        while (current != null) {
                if (current.getContent() == i) break;
                predecessor = current;
                current = current.getNext ();
        }
        if (current == null) predecessor = null;
        return current != null;
} ...
```



Einfach verkettete Lineare Listen (6)

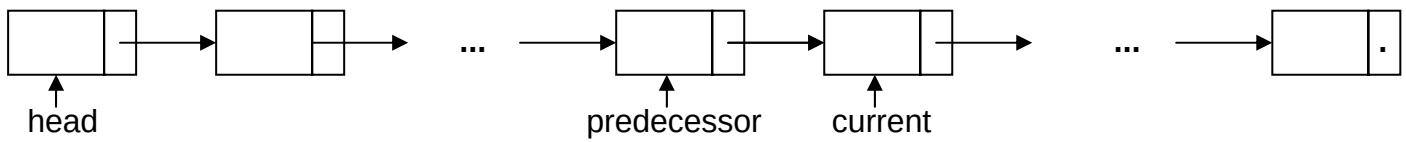
```
public boolean setPos (int p){              // setze aktuelle Position auf p
        if (p <= 0) {
                predecessor = current = null;
                return p == 0;
        }
        first ();
        while (current != null) {
                if (--p == 0) break;
                predecessor = current;
                current = current.getNext ();
        }
        if (current == null) predecessor = null;
        return current != null;
}
public boolean insert (int i, int p){ // als p-tes Element einfuegen
        boolean ok = setPos(p-1);
        if (ok) insert (i);
        return ok;
}
// ...
```

Einfach verkettete Lineare Listen (7)

```
public boolean delete (int p){ // p-tes Element loeschen
        boolean ok = setPos(p);
        if (ok) remove ();
        return ok;
}
```

Entfernen

```
public void remove (){
       IntNode h;
       if (current != null){
              h = current.getNext ();
              if (h == null)
                     if (predecessor == null) Head = null;
                     else predecessor.setNext (null);
              else { current.setContent (h.getContent ());
                     current.setNext (h.getNext ());
              }
              current = predecessor = null;
       }
}
```



Einfach verkettete Lineare Listen (8)

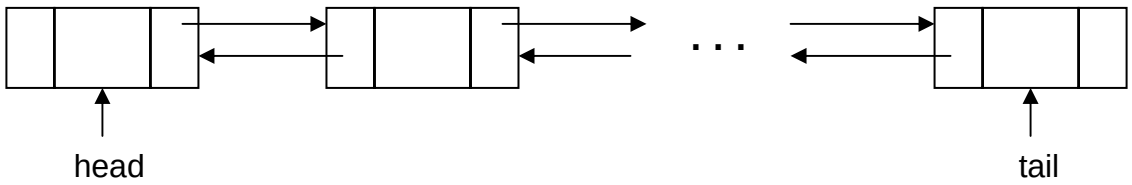
```
public String toString (){                    // Ausgabe aller Elemente
        String res= " ";
        IntNode h = Head;
        while (h != null) {
                res+=h.getContent() + " ";
                h = h.getNext ();
        }
        return res;
}
```

Implementierungen für Lineare Listen (4)

Mit Hilfe doppelt verketteter (verlinkter) Elemente:

• speichere mit jedem Element zwei Verweise auf das vorherige und nächste Element oder null, falls ein Element nicht existiert.

• Liste:



Besondere Eigenschaften dieser Lösung:

+ Umkehren der Liste in $\Theta(1)$
+ Entfernen von Elementen leichter machbar
+ Einfügen/Entfernen/Zugriff am Anfang/Ende in $\Theta(1)$
+ Hintereinanderhängen von Listen in $\Theta(1)$

Doppelt verkettete Lineare Listen (1)

Einfaches Beispiel für Liste aus beliebigen Objekten:

```
class Dlist {
        private Dnode Head, Tail; // Anfang, Ende der Liste
        public Dlist (){ // Konstruktor,
                Head = Tail = null; // erzeugt leere Liste
        }
        public boolean empty (){ // falls Head == null,
                return Head == null; // dann auch Tail == null
        }
        public Dlist pushHead (Object o){
                Dnode h = Head; // das alte Head
                Head = new Dnode (o); // das neue Head mit Inhalt
                if (Tail == null) Tail = Head; // falls Dlist vorher leer
                else { Head.connect (h); // sonst: verbinde Head mit h
                        h.connect (Head); // und umgekehrt
                }
                return this; // erlaubt kaskadieren
        }
```

Doppelt verkettete Lineare Listen (2)

```
public Dlist pushTail (Object o){ // wird mithilfe von reverse ()
        reverse (); // auf pushHead() reduziert
        pushHead (o); // funktioniert & ist
        reverse (); // effizient
        return this; // erlaubt kaskadieren
}
public Object popHead (){
        Dnode h = Head; // aufzugebendes El.
        Head = Head.getSingle (); // Head auf naechtes El.
        if (Head == null) Tail = null; // Dlist ist jetzt leer
        else Head.disconnect (h); // Verbindung loesen
        return h.getContent (); // Rueckgabewert
}
public Object popTail (){ // wird mithilfe von reverse ()
        reverse (); // auf popHead() reduziert
        Object o = popHead ();
        reverse ();
        return o;
}
```

Doppelt verkettete Lineare Listen (3)

```
        public Dlist concat (Dlist dl){ // Head von dl kommt an DIESEN Schwanz
                if (dl.Head != null){ // anders waeren wir schon fertig
                        if (Head == null) Head = dl.Head;// falls DIES leer
                        else { Tail.connect (dl.Head); // sonst: verbinden
                               dl.Head.connect (Tail);
                        }
                        Tail = dl.Tail; // neuer Schwanz
                }
                return this; // erlaubt kaskadieren
        }
        public void reverse (){ // Vertauschen von
                Dnode h = Head; // Head und Tail
                Head = Tail;
                Tail = h;
        }
}//class Dlist
```

Doppelt verkettete Lineare Listen (4)

```
/*******************************************************
* Ein Knoten (Dnode) hat ein Object als Inhalt und
* (zwei Verweise auf) weitere Dnode Knoten.
* Damit kann man doppelt verkettete Listen erzeugen.
*******************************************************/
class Dnode{
        private Object content; // sollen aussen nicht
        private Dnode linkA, linkB; // bekannt sein
        public Dnode (Object o){ // Konstruktor, Inhalt wird uebergeben
                content = o; // lege Inhalt ab
                linkA = linkB = null; // trage leere Verweise ein
        }
        public Object getContent (){ // Schnittstelle fuer Herausgabe
                return content; // des Inhalts
        }
        public Dnode getSingle (){ // gibt einzige Verbindung
                return getNext (null); // heraus
        }
```

Doppelt verkettete Lineare Listen (5)

```
        public Dnode getNext (Dnode dn){ // gibt die Verbindung raus
                if (linkA == dn) return linkB; // die *nicht* gleich dn ist
                else return linkA;
        }
        public void connect (Dnode dn){ // Dnode muss mindestens einen
                if (linkA == null && dn.linkA != this) linkA = dn;
                                        // link == null haben!
                else linkB = dn;
        }
        public void disconnect (Dnode dn){ // Dnode muss einen link
                if (linkA == dn) linkA = null; // auf Dnode haben
                else linkB = null;
        }
}//class Dnode
```

Stacks (Stapel) und Queues (Schlangen) (1)

$L =< a_1,...,a_n >$

Operation	Wirkung	Stack	Queue
Einfügungen L.pushHead(x) L.pushTail(x)	$< x, a_1,...,a_n >$ $< a_1,...,a_n, x >$	⊗	⊗
Enfernungen L.popHead() L.popTail()	$< a_2,...,a_n > \rightarrow a_1$ $< a_2,...,a_{n-1}, x > \rightarrow a_n$	⊗	⊗
Lesen L.top() L.bottom()	$\rightarrow a_1$ $\rightarrow a_n$	⊗	⊗
Diverses L.init() L.empty()	<> $(L ==<>)$	⊗ ⊗	⊗ ⊗

Stacks (Stapel) und Queues (Schlangen) (2)

Anmerkungen:

- Sonderfälle bei einigen Methoden, falls $L == <>$
- L.pushHead(x).top() = x; Kellerspeicher
- Stapel: Last In First Out, LIFO
  Schlange: First In First Out, FIFO

Implementierungen:

- Arrays
- Einfach oder doppelt verkettete Listen
- Subklassen bestehender Klassen, . . .