Kapitel 4: Lineare Datenstrukturen

Einfluss der Datenstruktur auf Komplexität

Die Organisation der Daten hat erheblichen Einfluss auf die Effizienz der durchzuführenden Operationen.

Problem von Datenstruktur

Aspekte:

- **Effizienzmaß**: Speicherplatz, Laufzeitverhalten, Programmieraufwand
- **Priorität der Operationen**: Üblicherweise Häufigkeit des Aufrufs.

Fundamentale Abstrakte Datentypen (ADT = Abstrakter Datentyp)

ADT = Abstrakter Datentyp: Ist eine abstrakte Darstellung von Methoden.

ADT: Lineare Liste

Definition: **Lineare Liste** - Eine lineare Liste ist eine endliche Folge von Elementen eines Grundtyps.

Operationen des ADT: Lineare Liste

Formal:

$L = \langle a_1, \ldots, a_n \rangle \quad n > 0$

$L = \langle\,\rangle \quad n = 0$

`init(L)`: initialisiert L, d.h. L=<>

`insert(x,p,L)`: fügt x an Position p in Liste L ein, alle Elemente ab Position p rücken um eine Position nach hinten.

- *formale Fallunterscheidung:*
  a) $L=\langle a_1,\ldots a_n\rangle$ und $1 \leq p \leq n$ → $L' = \langle a_1,\ldots,a_{p-1},x,a_p,\ldots,a_n\rangle$
  b) $L=\langle a_1,\ldots a_n\rangle$ und $p=n+1$ → $L' = \langle a_1,\ldots,a_n,x\rangle$
  c) sonst (d.h für $p \leq 0$ und $p>n+1$): Operation nicht definiert.

`delete(p,L)`: Löscht Element an Position p und verkürzt Liste.

`locate(x,L)`: Gibt erste Position von L in der x vorkommt zurück

`retrieve(p,L)`: Liefert das Element an Position p der Liste L.

`concat(L1,L2)`: Hintereinanderfügen der Listen L1 und L2.

Generelle Alternativen zur Implementierung

- **Sequentielle Speicherung:** Listenelemente in zusammenhängendem Speicherbereich hintereinander abgelegt.
- **Verkettete Speicherung:** Listenelemente beliebig über den Speicher verteilt, Zugriff durch Verweise (Zeiger), die den Listenzusammenhang herstellen.

Seq. Speicherung linearer Listen

sequentielle Suche: Durchlaufen aller Elemente des Arrays von vorn nach hinten und Vergleich jedes Elements mit Schlüssel

- Abbruch wenn Schlüssel gefunden

Anzahl der notwendigen Schlüsselvergleiche

- worst case: $C_{max} = n$
- average case: $C_{avg} = \frac{1}{n} * \sum_{i=0}^{n} i = \frac{1}{n} * \frac{n+1}{2} = \frac{n+1}{2}$

Bewegungen: Anzahl der Wertänderungen. Da der Index i immer um eins erhöht wird gilt:

- $M_{max} = n$
- $M_{avg} = \frac{1}{n} * \sum_{i=0}^{n} i = \frac{1}{n} * \frac{n+1}{2} = \frac{n+1}{2}$

Sequentielle Speicherung:

- Vorteil ist der wahlfreie Zugriff.
- Nachteil: Einfügen bzw. Löschen von Elementen
  - Block-Verschiebung

Suche in sortierter linearer Liste: Binäre Suche

- Annahme Suchverfahren:
  - **Sortierte** lineare Listen nach aufsteigenden Schlüsselwerten sortiert.
  - sequentielle Speicherung (**wahlfreier Zugriff**).
- **Binäre Suche** nach Schlüssel k
  1.) Falls Liste leer, endet Suche erfolglos.
  sonst betrachte A[m] an mittlerer Position der Liste A
  2.) Falls k=A[m] → fertig (d.h. return m)
  3.) Falls k<A[m], durchsuche linke Teilliste nach demselben Verfahren.
  4.) Falls k>A[m], durchsuche rechte Teilliste nach demselben Verfahren.

```
Algorithm BinSearchIterative(A,n,k)
  left←1;right←n;
  repeat
     mid←(left+right) div 2;
     if(k<A[mid]) then right←mid-1;
                  else left←mid+1;
  until(k=A[mid] OR left>right)

  if(k=A[mid]) then return mid;
               else return -1;
```

Bemerkung: Auch hier macht die formale Beschreibung der Lösungsstrategie noch keine Vorgabe über die Art der Implementierung z.B. rekursive Implementierung Implementierung möglich.

Analyse: Binäre Suche

Worst case: **(im worst case optimal)**
- Für erfolglose sowie erfolgreiche Suche wird geteilt, bis Folge ein-elementig ist

- Annahme: Länge der Liste ist n=2m-1:
  - Ergebnis: m Unterteilungen notwendig, also: **extrem effizienter Algorithmus**..

$$2^m - 1 = n \quad \Leftrightarrow \quad 2^m = n+1$$
$$\Leftrightarrow \quad m = \log_2(n+1)$$

Suche in sortierter linearer Liste: Interpolationssuche

- Grundidee: Mache die Position des inspizierten Elements abhängig von den Schlüsselwerten k, A[left],A[right]
  - Teilungspunkt so zu setzten in Relation zu dem Wertebereich!!
  - anstelle von $mid = \left\lfloor \frac{left+right}{2} \right\rfloor = \left\lfloor left + \frac{1}{2}(right-left) \right\rfloor$
  - verwende $mid = \left\lfloor left + \frac{k - A[left]}{A[right]-A[left]}(right-left) \right\rfloor$

left = 1
right = 26
k = W

$mid = \lfloor 1 + \frac{23-1}{26-1} \cdot (26-1) \rfloor$
$= \lfloor 1 + \frac{22}{25} \cdot 25 \rfloor = \lfloor 1 + 22 \rfloor$
$= 23$

Strategie:
- Anstatt wie bei der binären Suche immer in der Mitte zu teilen, versuche einen **günstigeren Teilungspunkt** zu erraten.
- Hat k einen großen Wert, befindet sich das gesuchte Element vermutlich im hinteren Teil der Daten → Teilung weiter hinten
- Bei kleinem Schlüssel wird das Feld weiter vorne geteilt.

Bewertung:
- Bei gleichverteilten Schlüsseln gilt: $C_{avg} = \log_2 \log_2 n + 1$
- Allerdings $C_{max} \in \Theta(n)$

Beispiel mit guter Eingabe:
- Suche nach k=5: starte mit l=1, r=9:

1	3	5	7	10	13	17	20	23
1	2	3	4	5	6	7	8	9

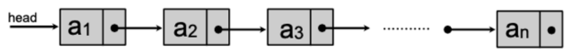
$$mid = \left\lfloor 1 + \frac{5 - A[1]}{A[9]-A[1]} \cdot (9-1) \right\rfloor = \left\lfloor 1 + \frac{5-1}{22-1} \cdot (9-1) \right\rfloor = \lfloor 2.52 \rfloor = 2$$

Beispiel mit ungünstiger Eingabe:
- Suche nach k=10: starte mit l=1, r=9:
- Berechnung läuft sukzessive alle Indices 1...9 ab.

1	3	4	5	6	7	8	9	1000
1	2	3	4	5	6	7	8	9

$$mid = \left\lfloor 1 + \frac{9}{999} \cdot 8 \right\rfloor = \lfloor 1.07 \rfloor = 1$$

➔ **Binäre Suche im WorstCase am Besten**
➔ **Interpolationssuche im Mittel besser, worst case Verhalten schlechter**.



Neues Konzept: Verkettete Speicherung lin. Listen

Idee: Speichere zusammen mit jedem Listenelement einen Verweis auf das nächste Element

Vorteil: Elemente können über Speicher verteilt sein

Nachteil: nur seq. Verfahren für Suche (selbst bei sortierter Eingabe)

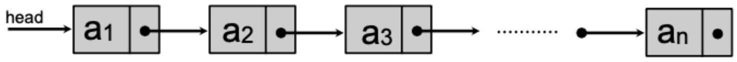
- Grund: kein wahlfreier Zugriff (keine binäre suche möglich)

Implementierungsalternative 1a:

- Liste wird realisiert durch Zeiger (head) auf Listenanfang.
- Listenende wird realisiert durch NULL-Zeiger.

Nachteile:

- Zugriff auf a_n erfordert lineare Zeit.
- Es muss immer auf NOT NULL geprüft werden




```
public class Node {
    public String a;
    public Node next;

    public Node(String s, Node n) {
         a=s;
         next=n;
    }
}
public class LinkedList{
    public Node head;
    public LinkedList(){head=null;}
}
```

Umsetzung von Implementierungsalternative 1a:

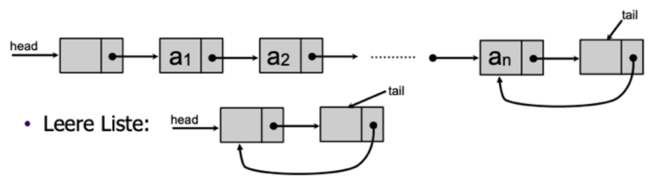
Leider ist diese Implementierung nicht so gut. Attribute sollten private sein und beim Erstellen des ersten Elements muss darauf geachtet werden, dass der head gesetzt wird

Implementierungsalternative 1b:

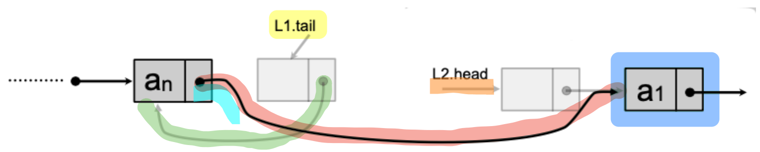
2 dummy-Elemente an 0-ter und n+1-ter Position(head, tail)

Vorteil:

- Listenende ist explizit bekannt, muss nicht gesucht werden
- Bein Aneinanderfügen zweier Listen ist der Aufwand O(1) statt O(n)!



`concat(L1,L2)`: Hintereinanderfügen der Listen L1 und L2. Siehe unten




```
head=L1.head;
L1.tail.next.next=L2.head.next;
tail=L2.tail;
if(L2.tail.next=L2.head) //If(L2 leer)
   tail.next=L1.tail.next;
```

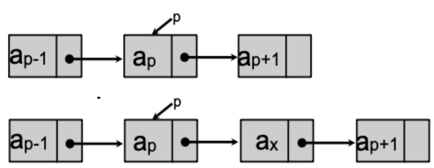
```
Algorithm concat(L1,L2)
  If L1.tail.next = L1.head then
    Return L2
  If L2.tail.next = L2.head then
    Return L1

  L1.tail.next.next=L2.head.next;
  Return L1
```

`insert(x,p,L)`:

fügt x an Position p in Liste L ein, alle Elemente ab Position p rücken um eine Position nach hinten.

- Der Zeiger von x wird auf den Zeiger von p gesetzt
- der Zeiger von p wird auf x gesetzt

```
x.next=p.next;
p.next=x;
if(x.next=tail)
  tail.next=x;
```



vor tail:

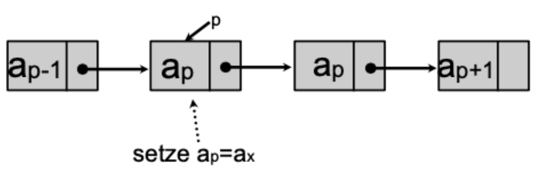
- tail hat Referenz auf das letzte reguläre Element, Einfügen nach diesem Element wie oben gezeigt.

vor einem Element:

- Schwierig weil next-Referenz des Vorgängers neu gesetzt werden muss, Vorgänger ist aber nicht in konstantem Aufwand zu finden.

Trick:

- 1. Füge eine Kopie von Element p nach p ein.
- 2. Kopiere die Daten von x in das erste der beiden Elemente p.

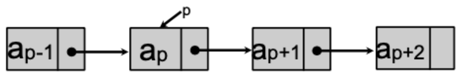



```
ap.next=apx.next;
apx.next=ap.next;
ap.setval(apx.getVal());
apx.setVal(x);
```

`delete(p,L)`: Löscht Element an Position p und verkürzt Liste.

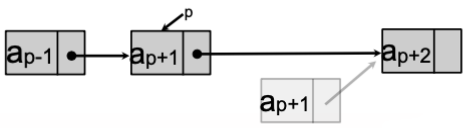
- wieder muss die next-Referenz des Vorgängers neu gesetzt werden
- wieder Trick um das mit konstantem Aufwand zu erledigen.

vorher:

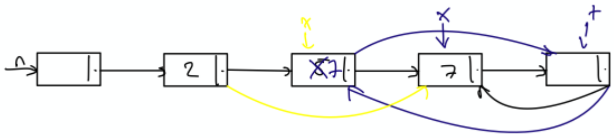


Trick:

1. Kopiere Daten von Knoten p+1 an die Position p.
2. Setze den next-Zeiger von p auf p+2.




```
Algorithm delete(x,L)
  If L.tail.next ≠ x then
    x.data ← x.next.data
    if L.tail.next = x.next then
      L.tail.next ←x
    x.next ← x.next.next
  else
    v ← L.head
    while v.next ≠ x do
      v  ← v.next
    v.next ← x.next
    L.tail.next ← v
```

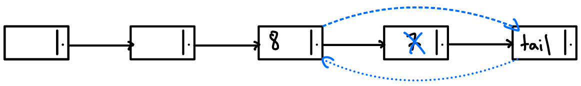


- Problematisch ist es das letzte reguläre Element zu löschen, da der Tailzeiger nicht neu in konstanter Zeit gesetzt werden kann.
- **Problem**: next-Zeiger des dummy-Elements am Ende kann nicht in konstanzer Zeit neu gesetzt werden, da kein direkter Zugriff auf Position k ist gegeben durch Knoten p-1.

Implementierungsvariante 1c:

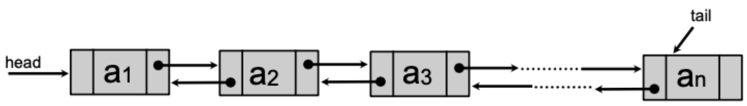
- Position k ist geg. durch Zeiger p auf k-1-te Listenelement

  salopp: aus Einfügen vor wird Einfügen nach dem Vorgänger etc...



Implementierungsvariante 2: Doppelt verkettete Listen

- Liste sei z.B. ohne dummy-Elemente, aber mit head + tail Zeigern realisiert.

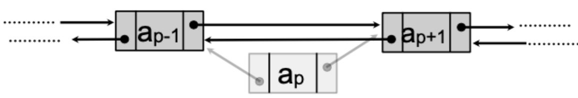



```
if(p==head)
  head=p.next;
if(p==tail)
  tail=p.prev
p.prev.next=p.next;
p.next.prev=p.prev;
```

Bemerkungen:

- höherer Speicherbedarf als bei einfacher Verkettung
- Aktualisierungsoperation etwas aufwändiger.
- insgesamt größere Flexibilität.

• Nachher:



`locate(x,L)`: Gibt erste Position von L in der x vorkommt zurück

- Für doppelt verkette Listen nicht besser, da man immer durch die ganze Liste laufen muss, egal ob vorwärts oder rückwärts.
- Auch hier kein wahlfreier Zugriff, keine effiziente Suche

Verkettete Listen:

- Einfügen von neuen Elementen effizient.
- Bei doppelt verketteten Listen auch Löschen effizient.
- Suchen ist ineffizient.

	1a	1b	1c	2
Insert(x, p, L)	O(1)	O(1)	O(1)	O(1)
Delete(p, L)	O(n)	O(n)	O(1)	O(1)
Locate(x, L)	O(n)	O(n)	O(n)	O(n)
ConCat(L1,L2)	O(n)	O(1)	O(1)	O(1)

Selbstorganisierende Listen

Wenn die Zugriffshäufigkeiten vorab nicht bekannt sind werden selbstorganisierende Liste benötigt

Ansatz 1: FC-Regel (Frequency Count)

- Häufigkeitszähler pro Element
- Bei Zugriff erhöht sich der Zähler des Elements
- Element wird mit den linken Nachbarn verglichen und soweit nach links verschoben, dass die Häufigkeitszähler eine absteigend sortierte Folge bilden.

Nachteil: hoher Wartungsaufwand und Speicherplatzbedarf, **trägheit**

Ansatz 2: T-Regel (Transpose)

- das Zielelement eines Suchvorgangs wird dabei mit dem unmittelbar vorangehenden Element vertauscht
- häufig referenzierte Elemente wandern langsam an den Listenanfang

Vorteil: weniger Aufwand bei Neuorganisation pro Zugriff

Problem: in der absoluten Zugriffswahrscheinlichkeit wird eine solche lokale und temporäre Häufung nicht beachtet.

Ansatz 3: MF-Regel (Move-to-Front)

- Zielelement eines Suchvorgangs wird nach jedem Zugriff an die erste Position der Liste gesetzt.
- relative Reihenfolge der übrigen Elemente bleibt gleich
- auch bei einfach verketten Listen effizient
- kürzlich referenzierte Elemente sind am Anfang der Liste

Vorteile:

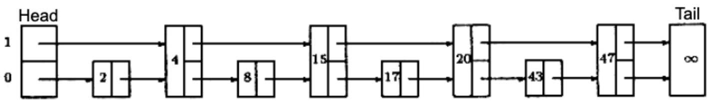
- Lokalität wird gut genutzt
- Lokalität wird schnell hergestellt
- Einfügen am Anfang effizient.

```
Algorithm insert(root,new)
  level ← maxLevel
  prev ← root
  while level >= 0 do
    curr ← prev.link[level]
    prevArr[level] ← prev
    if curr.val = new.value then        //Element gefunden
      return                             muss nicht eingefuegt warden
    level ← level -1
    if curr.val < new.value then
      prev ← curr
      index ← index + 2^level
    new.link[0] ← prev.link[0]
    prev.link[0] ← new
    curr ← new
  while curr ≠ NULL do
    if curr.val = ∞ then
      index ← 2^maxLevel
    else
      index ← index + 1
    level ← 1
    while index mod 2^level = 0 do
      preArr[level].link[level] ← curr
      prevArr[level] ← curr
      level ← level + 1
    curr ← curr.link[0]
```

Skip Listen

Ziel: eine verkettete Liste mit logarithmischem Aufwand für Suche, Einfügen und Löschen von Schlüsseln.

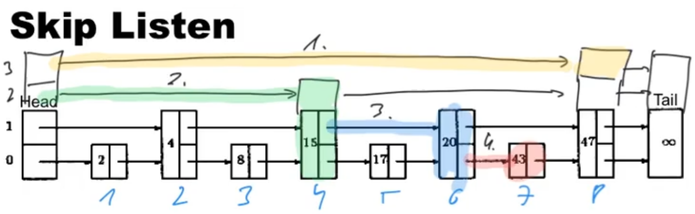
- Verkettung auf Ebene 0 verbindet alle Elemente
- Verkettung auf Ebene 1 verbindet jedes zweite Element
- Verkettung auf Ebene i verbindet jedes 2i-te Element.



  - Listenkopfelement enthält keinen Schlüssel.
  - Endelement enthält Schlüssel (∞), der größer ist als alle anderen Schlüssel.

Such ein Skip Listen:

- beginnt auf oberster Ebene m bis Element E gefunden wird, dessen Schlüssel den Suchschlüssel übersteigt
  - dabei werden viele Elemente übersprungen
- Fortsetzung der Suche auf darunter liegender Ebene m-1
  - Beginne die Suche mit dem Vorgänger von E in Ebene m.
  - Entspricht in array-Sichtweise dem Finden der Mittel- Indexposition von E und dessen Vorgänger in Ebene m.



Zeiger auf Ebenen

$E0: n = 2^m$

$E1: \frac{n}{2} = 2^{m-1}$

$E2: \frac{n}{4} = 2^{m-2}$

$Em: \frac{n}{2^m} = 2^{m-m} = 1$

Annahme $n = 2^m$

$\Leftrightarrow m = \log_2 n$

Summe über die Ebene: $2^0 + 2^1 + \ldots + 2^m = 2^{m+1} - 1$

$= 2^{(\log_2 n)+1} - 1$

$= 2 \cdot 2^{\log n} - 1$

$= 2 \cdot n - 1 \in O(n)$

- $\boldsymbol{Anzahl\ der\ Ebenen = \lfloor log_2 n \rfloor + 1}$
- **Es gilt: n= Anzahl Knoten + 2**
  +2 wegen Head und Tail

Perfekte Skip Listen

Such ein perfekter Skip Liste:

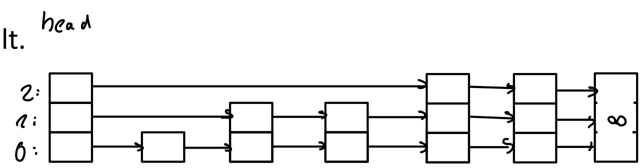
- Maximale # Schlüsselvergleiche pro Ebene: 2
- Bei Annahme n= 2k: # =2(k+1) = 2 log n +2 O(log n)

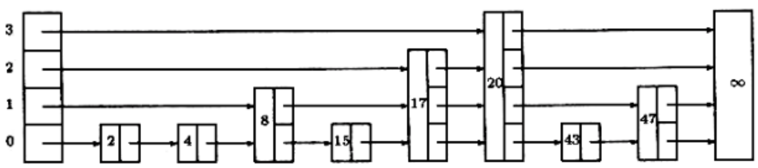
Nachteil von perfekten Skip Listen

- <u>Einfügen/ Entfernen</u> erfordert im worst-case
  eine vollständige Reorganisation der Liste, d.h. **<u>Θ(n).</u>**

Randomisierte Skip-Listen

- Die Höhe eines Elements wird nach dem Zufallsprinzip ermittelt.
- Suche in O(log n), Preis ist höherer Speicherverbrauch
- **Gesamt also Suchen, Einfügen, Löschen in O(log n)**
  - schlechter als verkettete Liste: O(1)






```
class Element(object):
        def __init__(self,value):
                self.value=value
                self.level=0
                self.link=[]
                lev=0
                while (lev<=maxLevelCnt):
                        self.link.append(None)
                        lev=lev+1
                while (random.randint(0,1)==1 and self.level<maxLevelCnt):
                        self.level=self.level+1
```

```
def findValue(root, value):
        levelNr=maxLevelCnt
        prev=root
        while (levelNr>=0):
                curr=prev.link[levelNr]
                while ((curr!=None) and (curr.value<value)):
                        prev=curr
                        curr=curr.link[levelNr]
                if (curr!=None) and (curr.value==value):
                        return curr
                levelNr=levelNr-1
        return None
```

Spezielle Listen: Stacks

- Ein Stack kann als spezielle Liste aufgefasst werden, bei der alle Einfügungen und Löschungen nur an einem Ende, TOP genannt vorgenommen werden.

Stack Operationen:

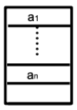
`create():`	erzeugt leeren Stack	O(1)
`init(S):`	initialisiert S als leeren Stack (alle Elemente löschen).	O(1)
`push(S,x):`	Fügt das Element x als oberstes Element von S ein.	O(1)
`pop(S):`	Löschen des Elements, da s als letztes in den Stack eingefügt wurde.	O(1)
`top(S):`	Abfragen des Elements, das als letztes in dem Stack eingefügt wurde.	O(1)
`empty(S):`	Abfragen, ob der Stack S leer ist.	O(1)

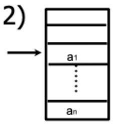
Bemerkungen:

- Diese formale Definition legt noch nicht fest ob sequentiell oder verkettet gespeichert wird.
- Einfach verkettet Liste: alle Operation in O(1), Stack kann beliebig groß werden, Speicherverbrauch entspricht der momentanen Belegung.

Stacks und seq. Speicherung

- Selbst programmierte Stacks sollten aus zwei Gründen nicht sequentiell gespeichert werden:

1) 
Oberstes Element des Stacks am Anfang des Arrays bedeutet Verschiebeaufwand beim Einfügen.

• 2) 
Speicher am Ende des Arrays von hinten nach vorne belegen erspart zwar das Verschieben, aber

- Zusätzlicher Zeiger auf Adresse von a_1 notwendig
- Stack hat vorgegebene Maximalgröße.
- in der Praxis oft so implementiert damit keine Objekte als Einträge nötig sind (next-Zeiger, Effizienzfrage)
- In diesem array-Kontext möglich: „Stack overflow“

Verwendung von Stacks

Beispiel 1: Iterative Auswertung von rekursiv definierten Funktionen

- Der Binomialkoeffizient ist die Anzahl der k elementigen Teilmengen einer n-elementigen Menge (k=6,n=49: Lotto)

$$\binom{n}{k} = \begin{cases} 1 & k=0, k=n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & 0<k<n \end{cases}$$

```
Algorithm binomial(n,k)
  if (k==0)||(k==n)
    return 1
  else
    return binomial(n-1,k-1) + binomial(n-1,k);
```

Bemerkungen:

- Die rekursive Implementierung wird dann ineffizient, wenn Stack bei Berechnung sehr voll wird.
- Das passiert insbesondere dann wenn bei der rekursiven Berechnung Zwischenergebnisse mehrfach berechnet werden.
- Die **theoretischen Laufzeitanalyse ist schlecht**, und der **Platzverbrauch** auf dem Stack.

Beispiel 2: Überprüfung von Klammerausdrücken – nicht so schön

Überprüfungsmethode basierend auf Verwendung eines Stacks

- Beim Einlesen der Zeichenkette
- Speichere öffnende Klammern in den Stack
- entferne oberstes Stackelement bei jeder schließenden Klammer (wenn Stack vorher schon leer dann Fehler->kein wgk)

Nach Durchlauf der Zeichenkette: wgk genau dann wenn Stack leer .

```
Algorithm wgk(string s)
  j←1; count←0;
  while (j<s.length)&&(count>=0) do
    if (s[i]==“(“) count++;
    if (s[i]==“)“) count--;
  if(count==0) return true;
  else return false;
```

Spezielle Listen: Queues

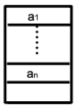
Synonyme: Schlangen, FIFO-Liste (first-in-first-out), Warteschlangen

- Eine spezielle Liste, bei der die Elemente an einem Ende („hinten“) eingefügt werden und am anderen Ende („vorne“) entfernt werden.

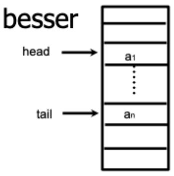
Operationen:

`create():`	erzeugt eine leere Queue	
`init(Q):`	initialisiert Q als leere Schlange.	
`enqueue(Q,x):`	fügt x am Ende von Q an.	O(1)
`dequeue(Q):`	löscht das erste Element von Q.	O(1)
`front(Q):`	Ausgabe des ersten Elements.	
`empty(Q):`	Abfrage ob Q leer ist.	

Bemerkungen zur Implementierung (Speicherbereich 0..m-1)

1) Naiv: 

Einfügen gut, Löschen schlecht, da restliche Elemente hochgeschoben werden müssen

2) besser 

z.B. Einfügen: Platz kann „modulo“ genutzt werden

```
Algorithm enqueue(Q,x,m)
  if ((tail+1)%m != head)
tail←(tail+1)%m; Q[tail]← x; return
true;
  else return false;
```

Erweiterung: Priority Queues

- Jedes Element erhält Priorität.
- Entfernt wird Element mit höchster Priorität.
- **Anwendung**: Liste von **Druckjobs** verwalten, ...

Implementierung:

- Naiv: unsortiert, Einfügen in O(1), Entfernen O(n)
- Naiv: sortiert, Einfügen in O(n), Entfernen O(1)

Tatsächliche Implementierung:

- Nutzung einer Baum Datenstruktur mit spez. Bedingungen (heaps)
- Einfügen in sortierte Liste O(log n), Entfernen: O(1)
- Queue zu Priority-Queue machen O(n log n)

```
Algorithm rotate(queue)
  curr ← queue.front()
  while !queue.empty() do
    dummyS.push(curr)
    queue.dequeue()
    curr ← queue.front()

  curr ← dummyS.pop()
  while ! dummyS.empty() do
    queue.enque(curr)
    curr ← dummyS.top();
    curr ← dummyS.pop()
  return queue
```

Spezielle Listen:

- Stacks und Queues für spezielle Anwendungen gut geeignet
- Priority-Queue stösst an die Grenzen

Wörterbuchproblem

- Mengen von Elementen eines Grundtyps mit den Operationen. (Init, Einfügen, Suchen, Entfernen)
- Bisher kann keine Datenstruktur alle Operationen für dieses Anwendungs-Szenario effizient umsetzen