Untere Schranke für den worst-case

Beobachtung:

- Der längste Pfad von der Wurzel zu einem Blatt entspricht der worst-case-Anzahl an Vergleichen.

Daher:

- Untere Schranke für die Höhe des Entscheidungsbaums ist daher untere Schranke der Laufzeit.

Nächster Satz liefert diese Schranke.

Untere Schranke für den worst-case

Satz Jeder Entscheidungsbaum, der n Elemente sortiert, hat die Höhe $\Omega(n \lg n)$.

Bew.: Ex. $n!$ Permutationen. Binärer Baum der Höhe h hat höchstens 2^h Blätter. Daher:

$$\begin{aligned} n! &\le 2^h \\ \lg n! &\le h \end{aligned}$$

Mit Sterlings Formel

$$n! = \sqrt{2\pi n}\frac{n^n}{e}\left(1 + \Theta(\frac{1}{n})\right)$$

ergibt sich

$$\begin{aligned} h &\ge \lg\left(\frac{n^n}{e}\right) \\ &\ge n \lg n - n \lg e \\ &= \Omega(n \lg n) \end{aligned}$$

Korollar: HEAP-SORT und MERGE-SORT sind asymptotisch optimale vergleichsbasierte Sortieralgorithmen.

Bew.: Die $O(n \lg n)$ worst-case Schranke beider Algorithmen fällt mit der unteren Schranke für den worst-case zusammen.

Count Sort

Annahme:

- Eingabelemente im Bereich $[1, k]$

Dann:

- $k = O(n)$ impliziert Laufzeit $O(n)$

Idee:

- Für jedes x bestimme die Anzahl der Elemente $\leq x$.

Falls 17 Elemente $\leq x$, dann muß x an die 18te Stelle.
(falls keine Duplikate)

Count Sort

Annahmen für Algorithmus:

- $A[1 \ldots n]$ für Eingabe
- $B[1 \ldots n]$ für sortierte Ausgabe
- $C[1 \ldots n]$ für Zwischenergebnisse

```
COUNTING-SORT (A, B, k)
for i ← 1 to k
    do C[i] ← 0
for j ← 1 to length[A]
    do C[A[j]] ← C[A[j]] + 1
▷C[i] now contains the number of elements = i.
for i ← 2 to k
    do C[i] ← C[i] + C[i − 1]
▷C[i] now contains the number of elements ≤ i.
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] − 1
```

	1	2	3	4	5	6	7	8
A	3	6	4	1	3	4	1	4

	1	2	3	4	5	6
C	2	0	2	3	0	1

(a)

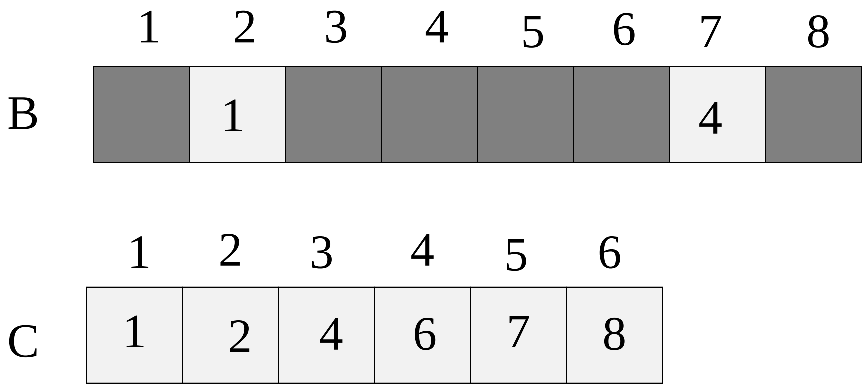
	1	2	3	4	5	6
C	2	2	4	7	7	8

(b)

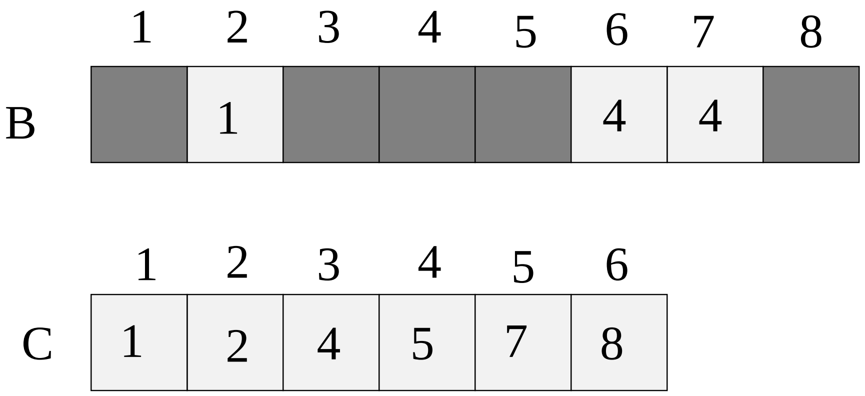
	1	2	3	4	5	6	7	8
B							4	

	1	2	3	4	5	6
C	2	2	4	6	7	8

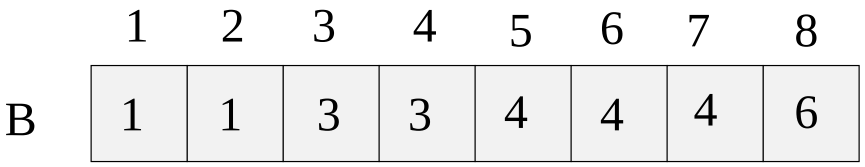
(c)


1 2 3 4 5 6 7 8
B
1
4
1 2 3 4 5 6
C
1 2 4 6 7 8

(d)


1 2 3 4 5 6 7 8
B
1
4 4
1 2 3 4 5 6
C
1 2 4 5 7 8

(e)


1 2 3 4 5 6 7 8
B
1 1 3 3 4 4 4 6

(f)

COUNTING-SORT Analyse

Zeile		Aufwand
1-2	:	$O(k)$
3-4	:	$O(n)$
6-7	:	$O(k)$
9-11	:	$O(n)$
Σ	:	$O(n+k)$

COUNTING-SORT wird benutzt, falls $k = O(n)$.

Dann ist der Aufwand

$$O(n)$$

Besser als $\Omega(n \lg n)$, da nicht vergleichsbasiert. Mehr noch: es kommt nicht ein einziger Vergleich vor.

COUNTING-SORT Eigenschaft

COUNTING-SORT ist ein

- **stabiles** Sortierverfahren.

heißt:

- gleiche Zahlen kommen in der Ausgangsfolge in der gleichen Reihenfolge wie in der Eingangsfolge vor.

Zieht natürlich nur, wenn Records sortiert werden.

Radix Sort

Ursprung:

- Kartenstapel sortieren

Idee:

- erst nach signifikantester Stelle sortieren
- dann naechst signifikante Stelle (rekursiv) usw.

Problem:

- Bei 10-System: 9 von 10 Stapeln müssen beim Sortieren zur Seite gelegt werden.

Daher (TRICK):

- Sortiere (stabil) nach am wenigsten signifikanter Stelle
- dann naechst signifikantere Stelle usw.

RADIX-SORT

Sortiere d-stellige Zahl.

RADIX-SORT(A, d)

for $i \leftarrow 1$ **to** d

  do use a stable sort to sort array A on digit i

RADIX-SORT

329		720		720		329
457		355		329		355
657		436		436		436
839	$\Rightarrow$	457	$\Rightarrow$	839	$\Rightarrow$	457
436		657		355		657
720		329		457		720
355		839		657		839
		$\uparrow$		$\uparrow$		$\uparrow$

RADIX-SORT Analyse

Analyse abhängig vom benutzten Sortierverfahren. Falls eine Stelle von $1 \ldots k$ läuft und k nicht zu groß ist:

- COUNTING-SORT gute Wahl

Dann ist der Aufwand für RADIX-SORT:

- $\Theta(dn + dk)$

Falls d konstant ist und $k = O(n)$ dann hat RADIX-SORT

- linearen Aufwand

Bucket Sort

Annahme für Bucket Sort:

- Zahlen zufällig in $[0, 1[$

Dann:

- avg-case: linear

Idee:

- Verteilen der Zahlen auf m Körbe (buckets)
- z.B. 10: $[0, 0.1[$, $[0.1, 0.2[$, ...
- Dann buckets sortieren und fertig.

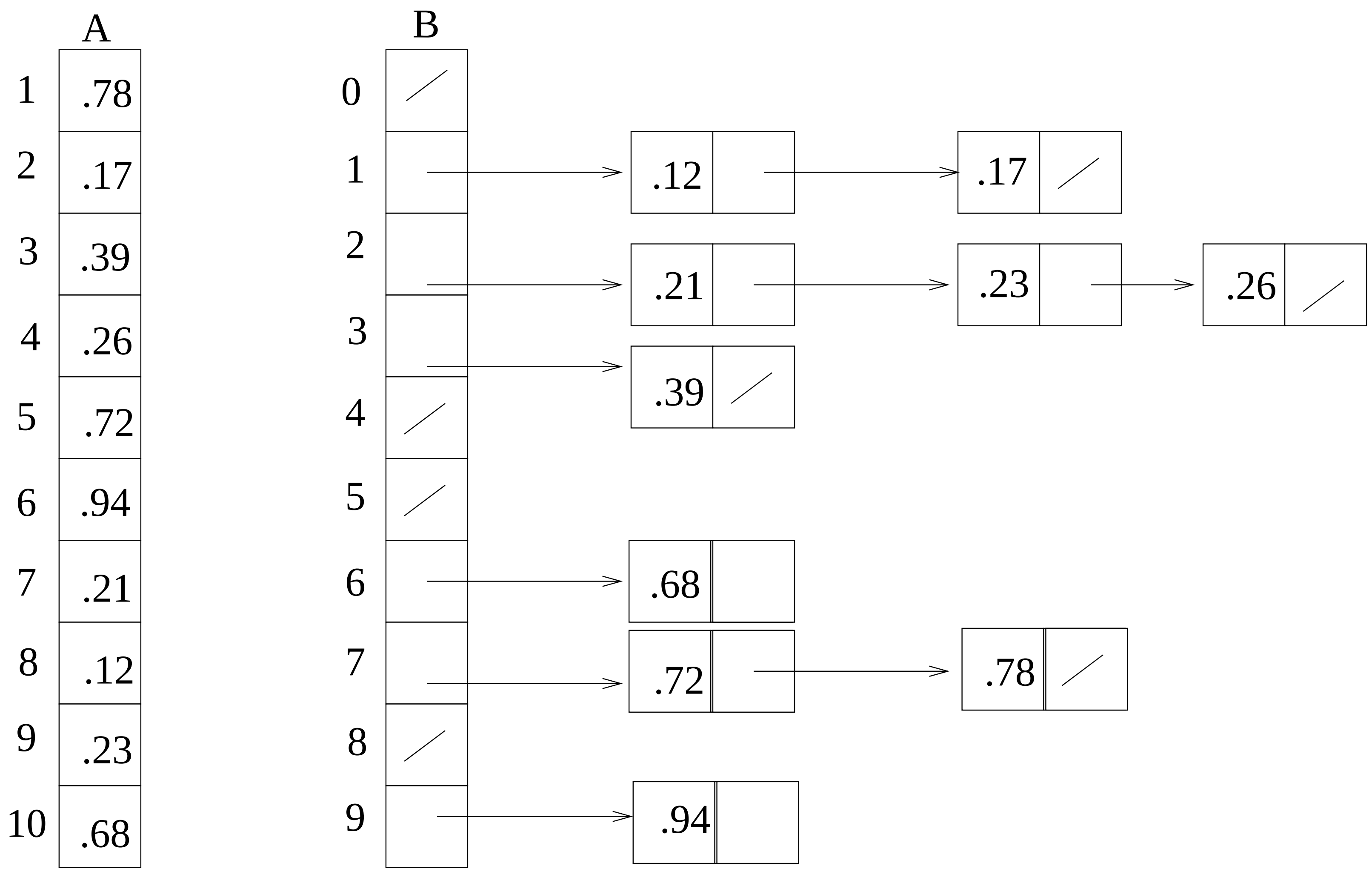
Bucket Sort

Annahmen im Code:

- n-elementiges Array A
- $0 \leq A[i] < 1$ f.a. i
- $B[0 \ldots n-1]$ Hilfsarray mit Listen von Zahlen (kleiner Vorgriff)

BUCKET-SORT(A)

$n \leftarrow$ **length**$[A]$
for $i \leftarrow 1$ **to** n
    do insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
for $i \leftarrow 0$ **to** $n - 1$
    do sort list $B[i]$ with insertion sort
concatenate $B[0], B[1], \ldots, B[n-1]$


A
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
B
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94

(a) (b)

BUCKET-SORT Analyse

- Jede Zeile außer Zeile 5 benötigt $O(n)$.
- Zeile 5: n_i^2, falls Länge von bucket i gleich n_i.
- Falls die Zahlen gut verteilt sind und es viele buckets gibt (ungefähr n), so ist der Aufwand trotzdem $\Theta(1)$.

Insgesamt:

- Aufwand: $O(n)$ (unter obigen Annahmen)

Mengen

Operationen:

- insert
- delete
- member

Variiert aber mit Anwendung.

“Parameter”:

- Worauf basiert Mengenzugehörigkeit?
  Schlüssel (key), Sekundärdaten (satellite data)
- Ordnungen?
- Mehrfachmengen?

Mengen Operationen

- SEARCH(S,k) gibt $\uparrow x$ zurück, falls $x \in S \wedge key(x) = k$
- INSERT(S,x) fügt x in S ein
- DELETE(S,x) entfernt x aus S
- MINIMUM(S) falls S total geordnet, gibt minimales Element zurück
- MAXIMUM(S) analog
- SUCCESSOR(S,x) falls S total geordnet, gibt Nachfolgeelement von x zurück
- PREDECESSOR(S,x) analog

Bei Mehrfachmengen: SUCC/PRED durchlaufen erst Elemente mit gleichem Schlüssel.

Komplexität der Operationen:

- Gemessen in $|S|$

Übersicht

1. sehr einfache sehr wichtige Datenstrukturen (Brot&Butter)
2. Hash-Tabellen
3. Binäre Suchbäume
4. Rot-Schwarz-Bäume

Stacks and Queues

Stapel und Schlangen:

- Ordnung: Zeitpunkt des Einfügens
  - Stack: last-in-first-out (LIFO)
  - Queue: first-in-first-out (FIFO)
- basierend auf Array

Stack

STACK-EMPTY(S)
if top$[S] = 0$
  then return TRUE
  else return FALSE

PUSH(S, x)
top$[S] \leftarrow$ **top**$[S] + 1$
$S[$**top**$[S]] \leftarrow x$

POP(S)
if STACK-EMPTY(S)
  then error "underflow"
  else top$[S] \leftarrow$ **top**$[S] - 1$
    return $S[$**top**$[S] + 1]$

	1	2	3	4	5	6	7
S	15	6	2	9			

top[S] = 4

	1	2	3	4	5	6	7
S	15	6	2	9	17	3	

top[S] = 6

	1	2	3	4	5	6	7
S	15	6	2	9	17	3	

top[S] = 5

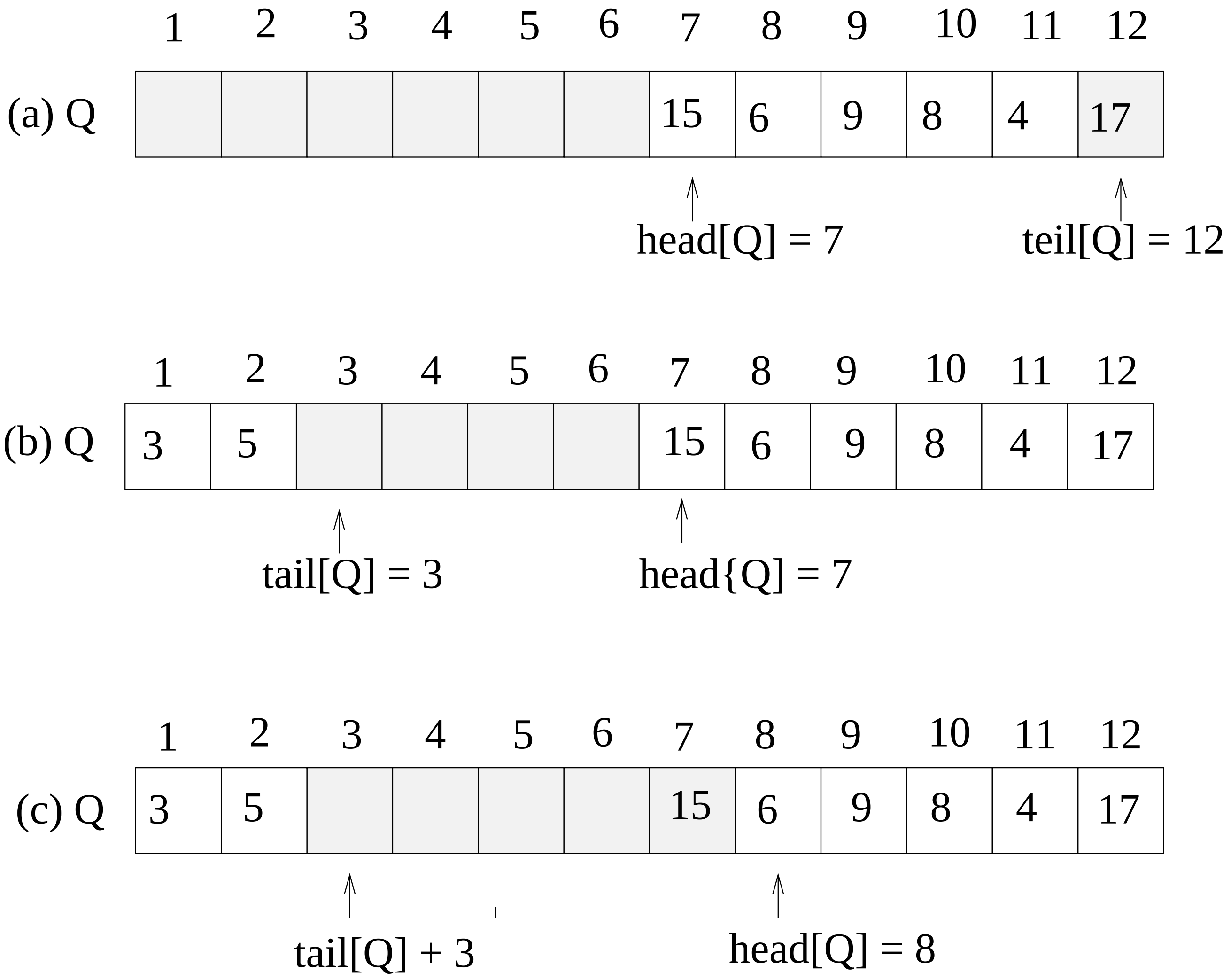
Stack

- Jede Operation: $O(1)$
- Nachteil: Array begrenzt Tiefe
- Häufig noch: TOP(S)

Queue

```
ENQUEUE(Q, x)
    Q[tail[Q]] ← x
    if tail[Q] = length[Q]
        then tail[Q] ← 1
        else tail[Q] ← tail[Q] + 1

DEQUEUE(Q)
    x ← Q[head[Q]]
    if head[Q] = length[Q]
        then head[Q] ← 1
        else head[Q] ← head[Q] + 1
    return x
```



1 2 3 4 5 6 7 8 9 10 11 12
(a) Q
15 6 9 8 4 17
head[Q] = 7
teil[Q] = 12
1 2 3 4 5 6 7 8 9 10 11 12
(b) Q
3 5 15 6 9 8 4 17
tail[Q] = 3
head{Q] = 7
1 2 3 4 5 6 7 8 9 10 11 12
(c) Q
3 5 15 6 9 8 4 17
tail[Q] + 3
head[Q] = 8

Queues

- Jede Operation: $O(1)$
- Nachteil: Array begrenzt Länge
- Häufig noch: FIRST(S)

Verkettete Listen

Reihenfolge durch:

- zeigerbasierte Verkettung

Ausprägungen:

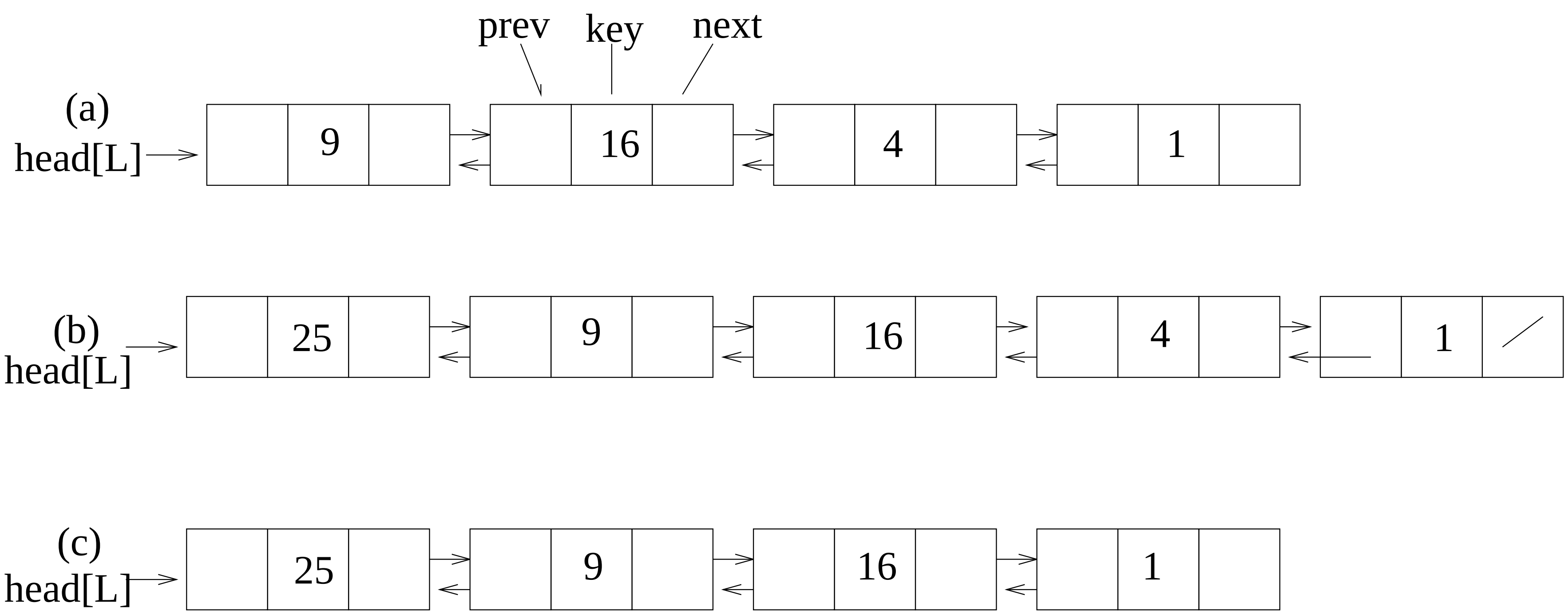
- einfach verkettete Liste: (succ,key)
- doppelt verkettete Liste: (prev,succ,key)

Optional:

- zyklisch: letztes Element verweist auf erstes (und umgekehrt)

Operationen:

- im Prinzip alle oben genannten, aber einige ineffizienter


prev
key
next
(a)
head[L]
9
16
4
1
(b)
head[L]
25
9
16
4
1
(c)
head[L]
25
9
16
1

doppelt verkettete Listen

Attribute:

- head[L] für Liste L
- tail[L] für Liste L
- next[x] für Listenelement x
- prev[x] für Listenelement x

Operationen:

- LIST-SEARCH(L,k)
- LIST-INSERT(L,x)
- LIST-DELETE(L,x)

für Liste L, Schlüssel k und Listenelement x.

nicht betrachtet:

- woher kommt x

LIST-SEARCH(L,k)

$x \leftarrow$ **head**[L]
while $x \neq$ NIL and **key**[x] $\neq$ k
  do $x \leftarrow$ **next**[x]
return x

Laufzeit (worst-case):

- $\Theta(n)$

LIST-INSERT(L, x)

next$[x] \leftarrow$ **head**$[L]$

if head$[L] \neq$ NIL

  then prev$[$ **head**$[L]] \leftarrow x$

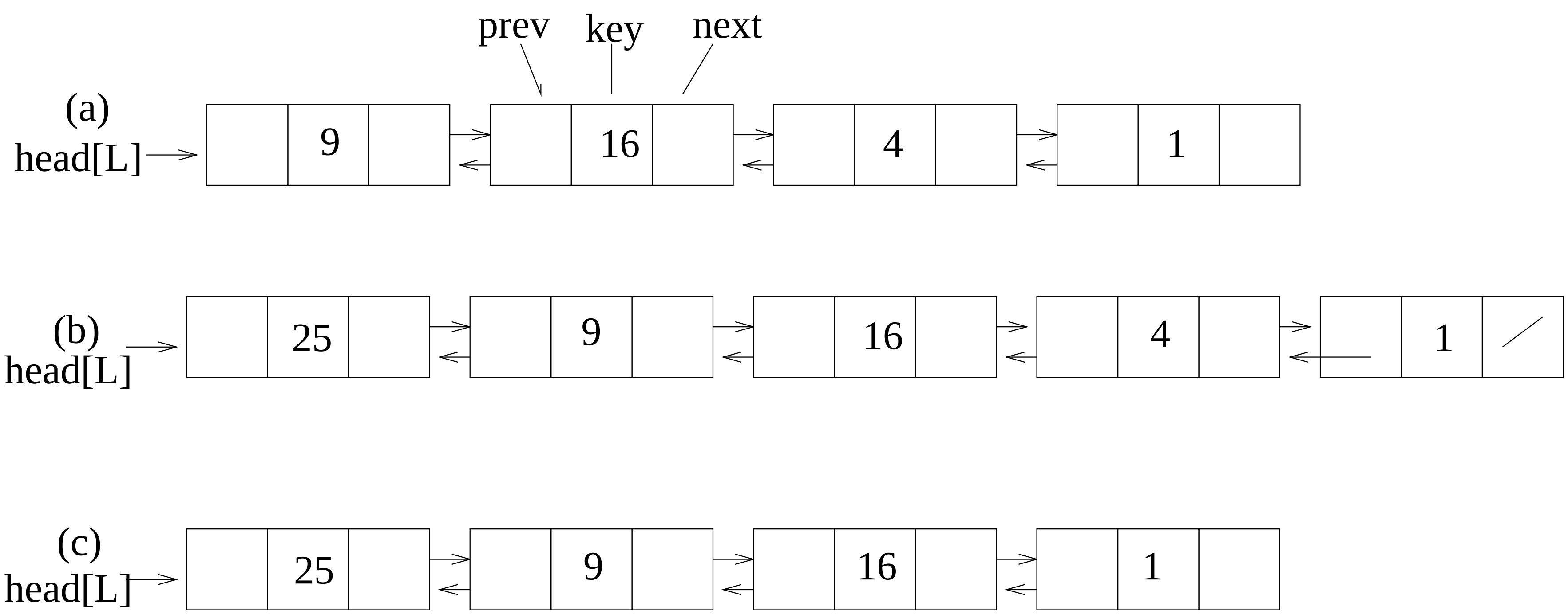
head$[L] \leftarrow x$

prev$[x] \leftarrow$ NIL

Laufzeit:

- $O(1)$

```
LIST-DELETE(L, x)
    if prev[x] ≠ NIL
        then next[ prev[x]] ← next[x]
        else head[L] ← next[x]
    if next[x] ≠ NIL
        then prev[ next[x]] ← prev[x]
```

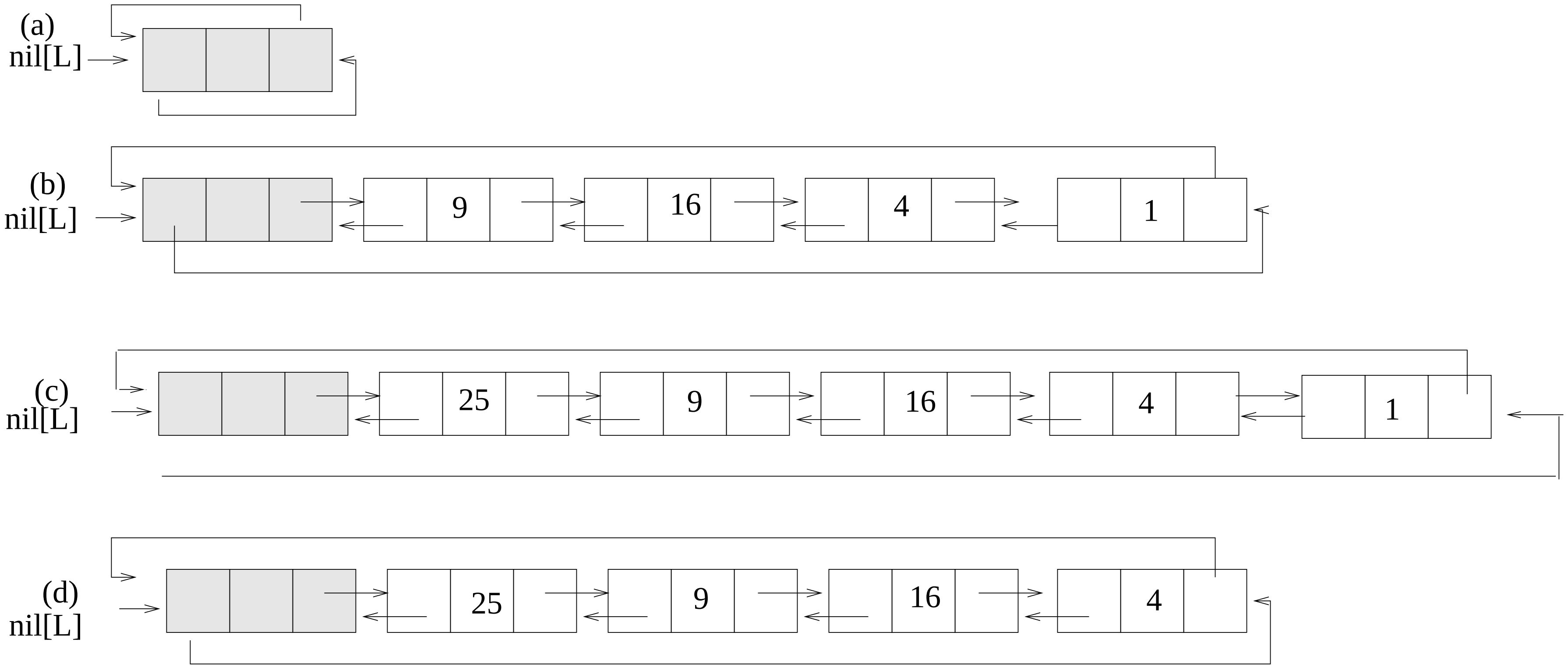

prev
key
next
(a)
head[L]
9
16
4
1
(b)
head[L]
25
9
16
4
1
(c)
head[L]
25
9
16
1

Dummy-Elemente

Falls kein Grenzfall zu berücksichtigen ist, ist der Code von LIST-DELETE viel einfacher:

LIST-DELETE'(L, x)
1 $\quad$ **next**[**prev**$[x]] \leftarrow$ **next**$[x]$
2 $\quad$ **prev**[**next**$[x]] \leftarrow$ **prev**$[x]$

Mit Dummy-Element (**nil[L]**) kann man dies erreichen:


(a)
nil[L]
(b)
nil[L]
9
16
4
1
(c)
nil[L]
25
9
16
4
1
(d)
nil[L]
25
9
16
4

LIST-SEARCH'(*L*,*k*)

```
x ← next[nil[L]]
while x ≠ nil[L] and key[x] ≠ k
    do x ← next[x]
return x
```

LIST-INSERT'(*L*,*k*)

```
next[x] ← next[nil[L]]
prev[next[nil[L]]] ← x
next[nil[L]] ← x
prev[x] ← nil[L]
```

Dummy-Elemente Bewertung

Beachte:

- Einführung kostet Speicherplatz

Nur Einführen, falls erhebliche Vereinfachung im Code, oder sogar erheblicher Laufzeitvorteil. Ansonsten, besonders bei kurzen Listen nicht angebracht.

Implementierung Zeiger und Objekte

Wann?

- falls die Sprache keine zur Verfügung stellt
- falls eigene Speicherverwaltung sinnvoll ist

Idee:

- Benutze Arrays und Arrayindizes

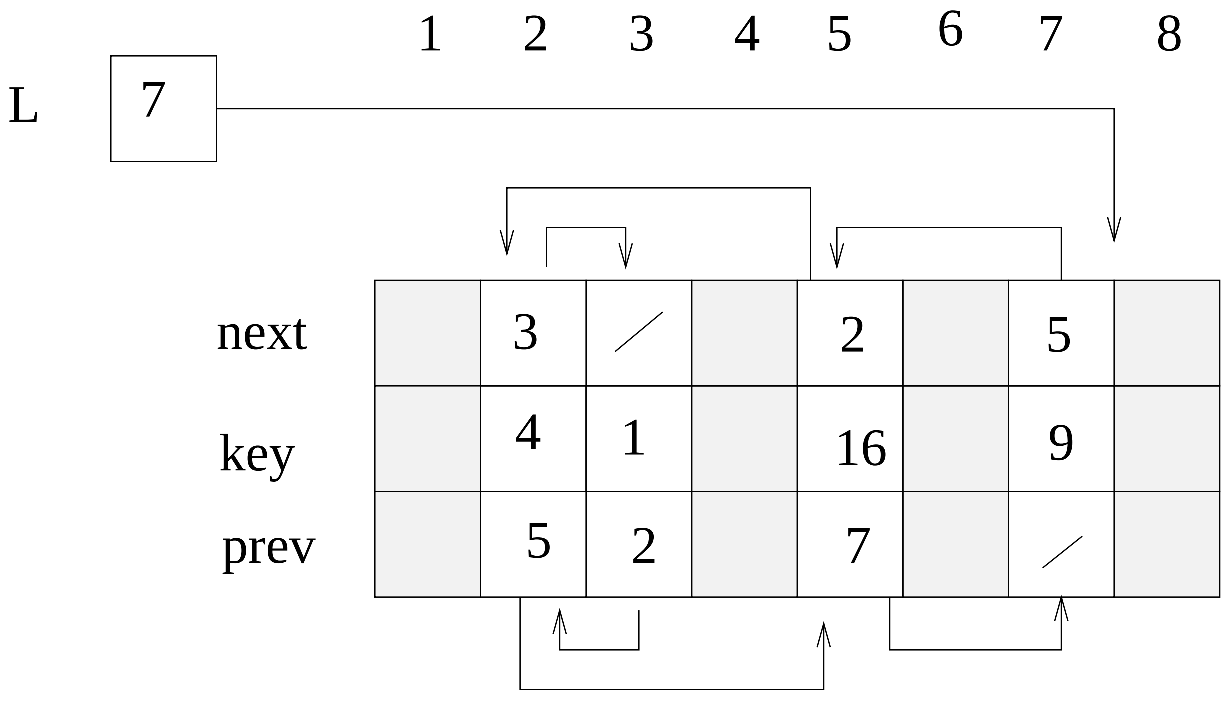
Unterscheide:

- Mehr-Array-Implementierung
- Ein-Array-Implementierung

Mehr-Array-Implementierung

Beispiel:

- Liste:
  - 3 Arrays: next, key, prev



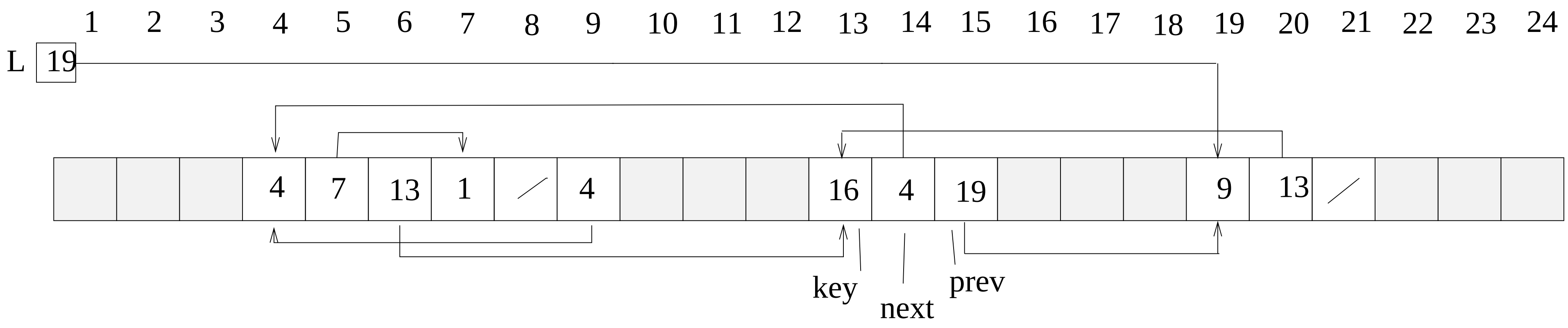
Linked List

Ein-Array-Implementierung

Beispiel:

- Liste:
  - Array von Tupeln (records) mit 3 Feldern: next, key, prev

(kann auch durch offset realisiert sein)


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
L
19
4
7
13
1
4
16
4
19
9
13
key
next
prev

Speicher Anfordern und Freigeben

Alternativen für Freigabe von Speicher:

- automatisch
  - garbage collector (Freispeichersammler)
- manuell
  - storage manager (Freispeicherverwaltung)

  Hier muß Programmierer Speicher explizit freigeben

Diskussion:

- Freispeicherverwaltung
- nur gleich große Stücke
- basiert auf Mehr-Array-Implementierung

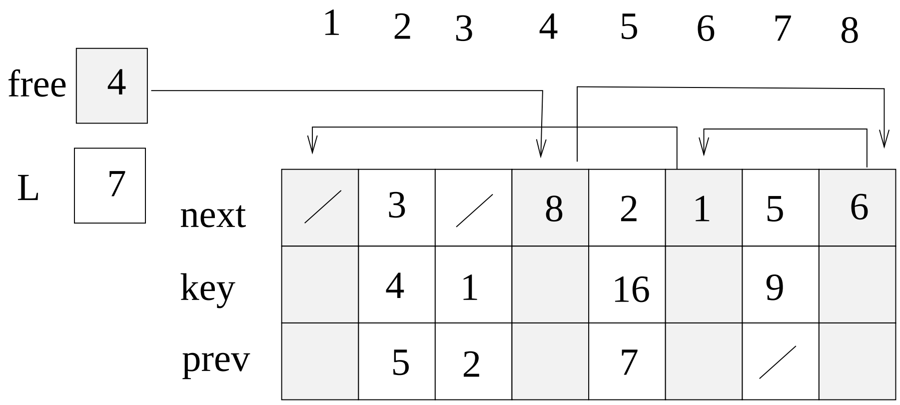
Freispeicherverwaltung

Idee:

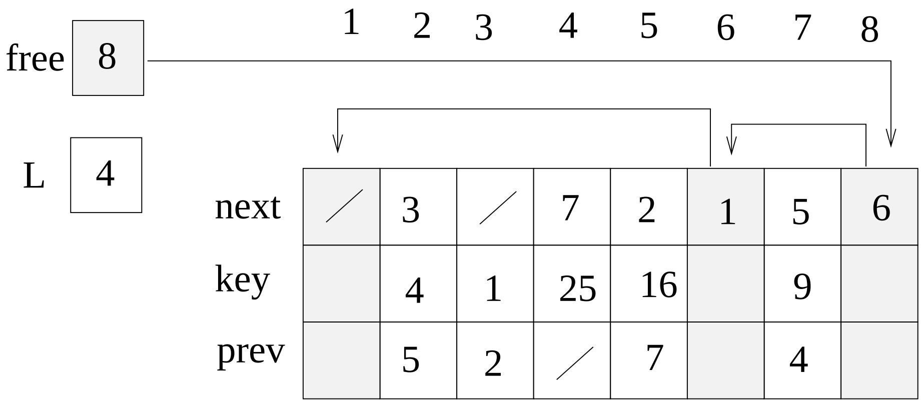
- freien Speicher in Liste organisieren
  free list (Freispeicherliste)
- Anfordern: von Freispeicherliste ausfügen
- Freigabe: in Freispeicherliste einfügen
- free list als Stack organisiert, nur **next** benutzt

```
ALLOCATE-OBJECT()
if free = NIL
    then error "out of space"
    else x ← free
        free ← next[x]
        return x
```

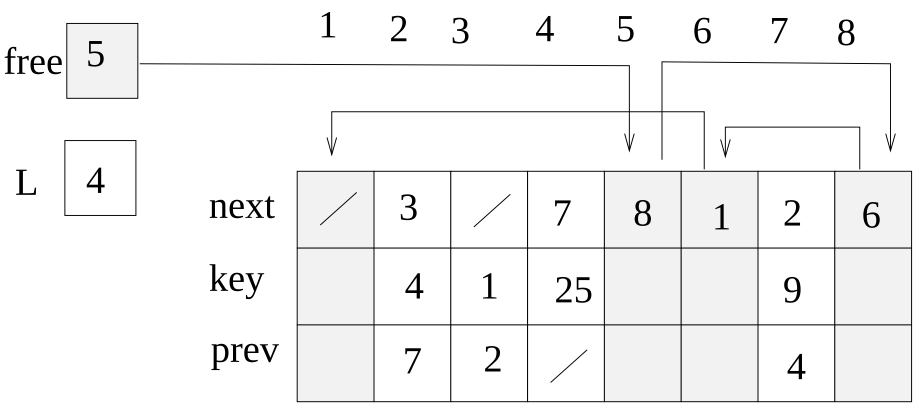
```
FREE-OBJECT(x)
next[x] ← free
free ← x
```


1 2 3 4 5 6 7 8
free 4
L 7
next 3 8 2 1 5 6
key 4 1 16 9
prev 5 2 7

(a)


1 2 3 4 5 6 7 8
free 8
L 4
next 3 7 2 1 5 6
key 4 1 25 16 9
prev 5 2 7 4

(b)


1 2 3 4 5 6 7 8
free 5
L 4
next 3 7 8 1 2 6
key 4 1 25 9
prev 7 2 4

(c)

free	10
L_2	9
L_1	3

	1	2	3	4	5	6	7	8	9	10
next	5		6	8		2	1		7	4
key	k_1	k_2	k_3		k_5	k_6	k_7		k_9	
prev	7	6			1	3	9			

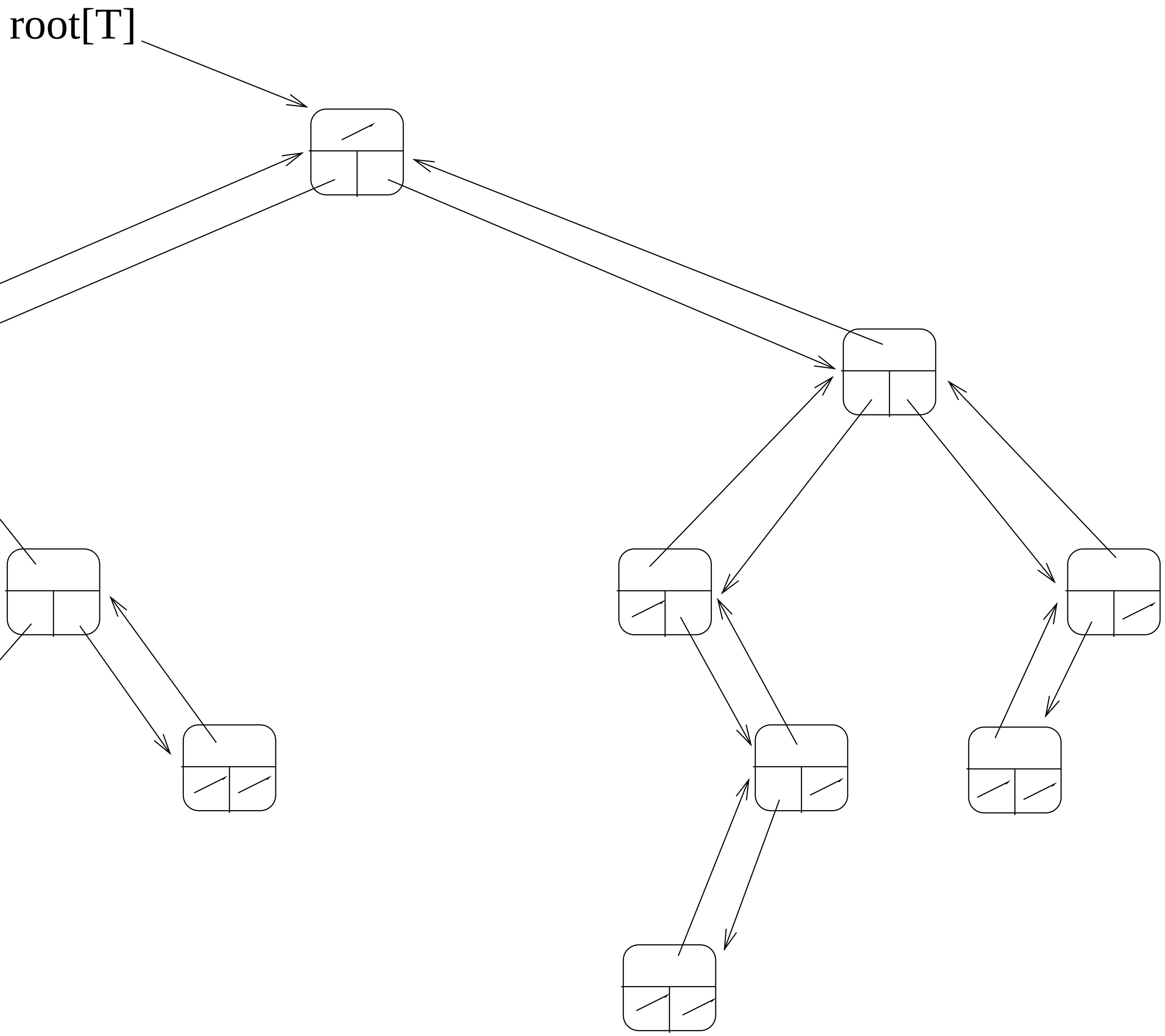
Zwei Listen L1 und L2

Repräsentation von Wurzelbäume

Binärbäume:

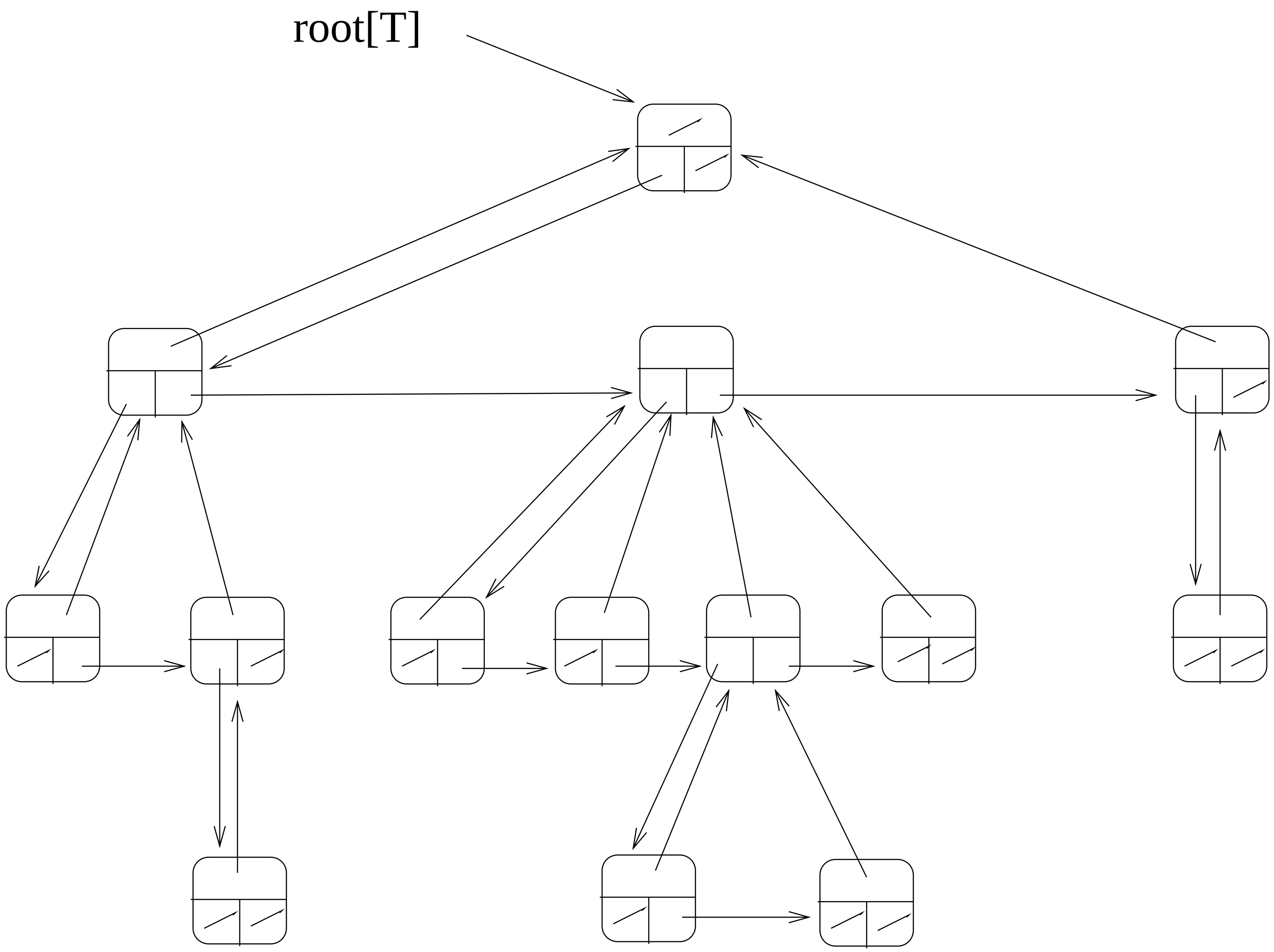
- Felder:
  - p: parent
  - left: left child
  - right: right child
- p[x]==NIL $\Longrightarrow$ x ist Wurzel
- root[T]: ist Wurzel
- root[T]==NIL: Baum ist leer.

Schema kann verwendet werden, solange der Knotengrad beschränkt ist.


root[T]

Unbegrenzter Grad

- Binärbaume können hierfür benutzt werden:
- left-child, right-sibling Repräsentation:
- Felder:
  - p: parent
  - left-child[x]: leftmost child of x
  - right-child[x]: sibling to the right of x
- left-child[x]==NIL: no children
- right-child[x]==NIL: x is rightmost child


root[T]

Andere Repräsentationen

- Heap
- einige Zeiger können wegfallen
- vieles möglich

Hash-Tabellen

Oft benötigt:

- dynamische Menge von Elementen mit Operationen
  - INSERT
  - SEARCH
  - DELETE
- Realisiert Abbildung Key$\mapsto$Element
- Ergibt Dictionary (Map).

Hash-Tabellen

Idee:

- Benutze Schlüssel als Arrayindex

Bei Strings, großen Zahlen etc. schwierig.

Daher:

- Berechne Arrayindex aus Schlüssel

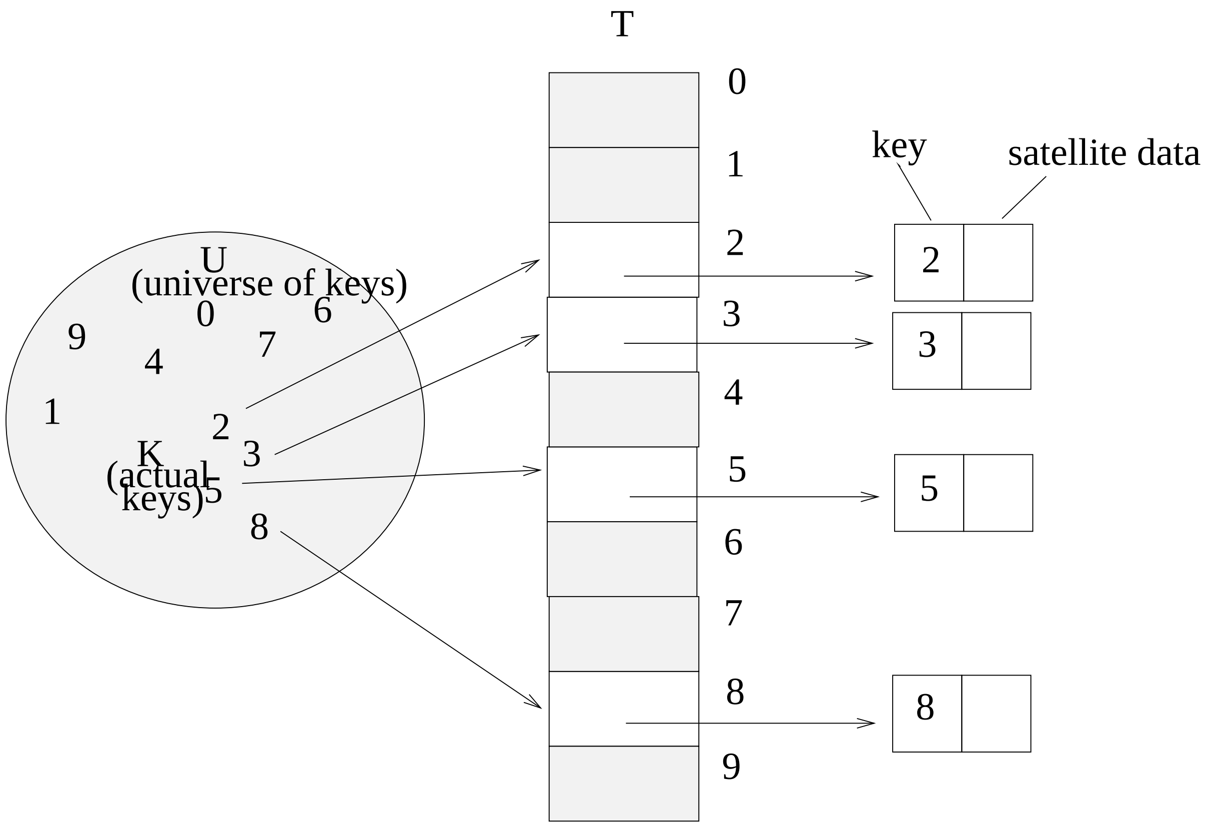
Eigenschaften:

- Alle Operationen in $O(1)$

Direkte Adressierung

Annahme:

- Alle Schlüssel aus $U = \{0, 1, \ldots, m - 1\}$
- Array T ist unsere Tabelle (direct address table)
- ein Feld in T heißt **slot**


T
0
1
2
3
4
5
6
7
8
9
key
satellite data
2
3
5
8
U
(universe of keys)
0
6
9
7
4
1
2
3
K
(actual
keys)
5
8

Operationen

DIRECT-ADDRESS-SEARCH($\mathbf{T}$,$\mathbf{k}$)
  return $\mathbf{T}[\mathbf{k}]$

DIRECT-ADDRESS-INSERT($\mathbf{T}$,$\mathbf{x}$)
  $\mathbf{T}[\mathbf{key}[\mathbf{x}]] \leftarrow \mathbf{x}$

DIRECT-ADDRESS-DELETE($\mathbf{T}$,$\mathbf{x}$)
  $\mathbf{T}[\mathbf{key}[\mathbf{x}]] \leftarrow \mathrm{NIL}$

Hash-Tabellen

Probleme mit direkter Adressierung offensichtlich:

- großes U
- nicht dichtes U
- Speicheraufwand $\Theta(U)$, selbst wenn nur Schlüssel aus K mit $|K| << |U|$ tatsächlich gespeichert werden.

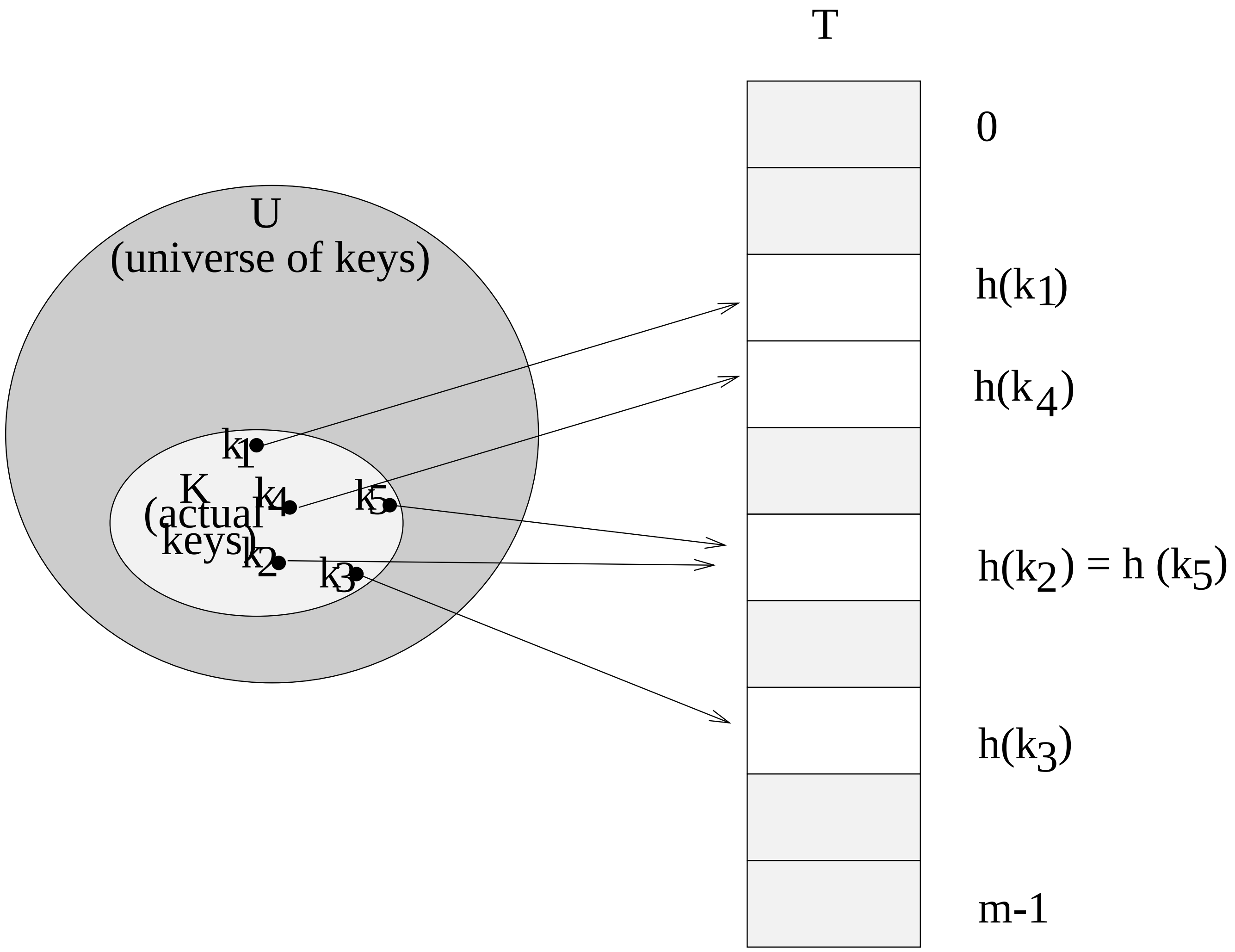
Daher Hashtabellen mit

- O(1) Zeit (jetzt allerdings avg-case)
- $\Theta(|K|)$ Speicher

Hash-Funktion

Bei direkter Adressierung wurde ein Element k an Adresse k gespeichert. Jetzt:

- speichere k an Adresse $h(k)$
- h heißt Hash-Funktion
- $h : U \longrightarrow \{0, \ldots, m-1\}$
- $h(k)$ heißt hash value von k.
- $h(k_1) = h(k_2)$: Kollision
- h sollte möglichst zufällig aussehen, um Kollisionen zu vermeiden
- Treten trotzdem auf, da normalerweise $|U| > m$
- Daher wichtig: Kollisionsbehandlung


T
U
(universe of keys)
K
(actual
keys)
k1
k4
k5
k2
k3
0
h(k1)
h(k4)
h(k2) = h (k5)
h(k3)
m-1

Kollisionsbehandlung durch Listen

Idee:

- an jedem Slot hängt eine Liste, die alle kollidierenden Elemente enthält

CHAINED-HASH-INSERT(**T,x**)
insert x at the head of list $T[h(key[x])]$

CHAINED-HASH-SEARCH(**T,k**)
search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(**T,x**)
delete x from the list $T[h(key[x])]$

Analyse

- INSERT: O(1) worst-case
- SEARCHING: proportional zur Länge der Liste
- DELETE: wie suchen

SEARCHING untersuchen wir noch genauer.

Analyse

Problem:

- Gegeben: Tabelle T mit m Plätzen und n Einträgen
- Gesucht: durchschnittliche Länge der Überlauflisten

Wir definieren noch:

- Füllgrad $\alpha := n/m$

Analyse

- worst-case: Liste hat Länge n:
  $\rightsquigarrow \Theta(n)$ Aufwand für SEARCHING
- avg-case: Verteilung durch h kritisch
- Annahmen für avg-case:
  - h verteilt gleichmäßig auf alle Plätze von T
  - h berechenbar in $\Theta(1)$
- Aufwand steckt im Ablaufen der Kollisionsliste
- 2 Fälle: Suche nicht erfolgreich/erfolgreich

Nicht erfolgreiche Suche

Theorem 12.1
In einer Hash-Tabelle mit Kollisionslisten hat eine nicht erfolgreiche Suche den durchschnittlichen Aufwand von $\Theta(1+\alpha)$

Beweis:

- Unter der Gleichverteilungsannahme: Jedes k wird auf einen der m Slots mit gleicher Wahrscheinlichkeit gehasht.
- durchschnittlicher Aufwand entspricht also dem Durchschnittsaufwand zum vollständigen Durchlaufen einer Kollisionsliste
- Durchschnittslänge der Liste $= \alpha = n/m$
- n/m Elemente durchschnittlich durchlaufen
- avg-case (mit Berechnung von $h(k)$):
  $\Theta(1+\alpha)$

Erfolgreiche Suche

Theorem 12.2
In einer Hash-Tabelle mir Kollisionslisten hat eine erfolgreiche Suche den durchschnittlichen Aufwand von $\Theta(1+\alpha)$.

Beweis
Annahmen:

- Jedes der n in der Tabelle gespeicherten Elemente ist mit gleicher Wahrscheinlichkeit das gesuchte Element.
- CHAINED-HASH-INSERT fügt am Ende an (Übung: beeinflußt nicht avg-case Suchzeit im Erfolgsfall)

Beobachtung:

- Erfolgreiche Suche benötigt 1 plus die Zeit der nicht erfolgreichen Suche VOR dem Einfügen des gesuchten Elements.

Um die durchschnittliche Anzahl der besuchten Elemente zu berechnen, nehmen wir den Durchschnitt

- über allen n gespeicherten Elementen (i)
- von 1+ der erwarteten Länge der Liste zu welcher i hinzugefügt wurde.

Diese erwartete Länge ist $(i-1)/m$, da

- bei Einfügen von i bereits $i-1$ Elemente in der Tabelle sind

Also ist die durchschnittliche Anzahl von besuchten Elementen:

$$
\begin{aligned}
1/n \sum_{i=1}^{n} \left(1 + \frac{i-1}{m}\right) &= 1 + \frac{1}{nm} \sum_{i=1}^{n} (i-1) \\
&= 1 + \left(\frac{1}{nm}\right) \left(\frac{(n-1)n}{2}\right) \\
&= 1 + \frac{\alpha}{2} - \frac{1}{2m}
\end{aligned}
$$

Also Gesamtaufwand (mit $h(k)$ ausrechnen):

$$
\Theta(2 + \frac{\alpha}{2} - \frac{1}{2m}) = \Theta(1 + \alpha)
$$

Analyse

Falls

- $\alpha = n/m = O(m)/m = O(1)$

dann

- SEARCH braucht $O(1)$.

Auch:

- INSERTION: $O(1)$
- DELETION: WIE SEARCH!!!

Hash-Funktionen

Frage:

- Wie bastel ich mir eine Hash-Funktion?

Hash-Funktionen

Idee:

- Interpretiere Schlüssel als natürliche Zahlen

Zum Beispiel:

- integer: z.B. Vorzeichen weg machen (als unsigned interpretieren)
- ASCII string: Zahlen von 0-127, einzelnen Buchstaben multipliziert mit ihrer Wertigkeit ($*128^i$) aufaddieren

Annahme:

- Schlüssel ist natürliche Zahl

Divisionsmethode

Definition:

$$h(k) := k \bmod m$$

Wahl von m:

- keine Zweierpotenz: nur letzten i bit berücksichtigt
- keine Zehnerpotenzen, falls wir im Dezimalsystem sind.
- Falls
  - $m = 2^p - 1$ für Primzahl p

  so bildet h zwei Strings in Basis-128-darstellung bei denen nur 2 Zeichen vertauscht wurden, auf den gleichen Wert ab.
- GUT: Primzahl die nicht in der Nähe einer Zweierpotenz ist.

Beispiel: $n = 2000$: Wähle $m = 701$.

Multiplikationsmethode

Definition:

$$h(k) := \lfloor m((kA) - \lfloor kA \rfloor) \rfloor$$

mit $0 < A < 1$.

Vorteil:

- Güte nicht abhängig von m
- Daher Wahl von m: kann 2-Potenz sein.

Güte von A abhängig. Gute Wahl:

- $A = (\sqrt{5} - 1)/2 = 0.6180339887\ldots$

(nach Knuth)

Offene Adressierung

Bisher:

- Elemente in Liste gespeichert

Jetzt:

- Elemente in Hashtabelle speichern

Idee:

- Falls frei: nutze $h(k)$ zum speichern
- sonst: nutze andere freie Felder

Damit $\alpha \leq 1$.

Offene Adressierung

Wir suchen freies Feld abhängig von einer erweiterten Hashfunktion

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

wobei für jeden Schlüssel k die Probensequenz

$$\langle h(k,0), h(k,1), \ldots, h(k, m-1) \rangle$$

eine Permutation von $\{0, 1, \ldots, m-1\}$ sein muß.

Probensequenz wird dann genutzt um Speicherplatz für ein Element zu finden und auch um es zu suchen.

Offene Adressierung

HASH-INSERT(**T,k**)

```
i ← 0
    repeat j ← h(k, i)
        if T[j] = NIL
            then T[j] ← k
                return j
            else i ← i + 1
    until i = m
error "hash table overflow"
```

Offene Adressierung

HASH-SEARCH(**T,k**)

```
i ← 0
    repeat j ← h(k, i)
            if T[j] = k
                then return j
            i ← i + 1
        until T[j] = NIL or i = m
return NIL
```

Offene Adressierung

Lineares Hashing:

$$h(k,i) := (h'(k) + i) \bmod m$$

Quadratisches Hashing:

$$h(k,i) := (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Doppeltes Hashing:

$$h(k,i) := (h_1(k) + i h_2(k)) \bmod m$$

mit $h_2(k)$ prim zu m.

Offene Adressierung Analyse

- Analyse in termini von $\alpha = n/m$
  (n Elemente in m Slots)
- Annahme: gleichmäßiges Hashen:
  - Sequenz $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$
  - ist mit gleicher Wahrscheinlichkeit jede der Permutationen von $\langle 0, \ldots, m-1\rangle$.
- Wir berechnen nun die erwartete Anzahl von Vergleichen für Hashing mit offener Adressierung.
- Zuerst nicht-erfolgreiche, dann erfolgreiche Suche.

Offene Adressierung Analyse

Satz
Gegeben sei eine Hashtabelle mit offener Adressierung und einem Füllgrad $\alpha = n/m < 1$. Dann ist die erwartete Anzahl von Vergleichen im erfolglosen Fall höchstens $1/(1-\alpha)$, falls wir gleichmäßiges Hashen annehmen.

Beweis Bei der erfolglosen Suche vergleicht man zunächst mit einer Reihe von besetzten Slots, bis man zum Schluß auf einen leeren Slot trifft. Wir definieren

$$p_i = Pr\{\text{genau } i \text{ Vergl. greifen auf besetzte Slots zu}\}$$

für $i = 0, 1, \ldots$. Für $i > n$ ist $p_i = 0$, da nur n Slots besetzt sind.

Der Erwartungswert für die Anzahl der Vergleiche ist also

$$1 + \sum_{i=0}^{\infty} i p_i$$

(Definition Erwartungswert)

Wir definieren

$$q_i = Pr\{\text{mind. } i \text{ Vergl. greifen auf besetzte Slots zu}\}$$

Es gilt

$$\sum_{i=0}^{\infty} ip_i = \sum_{i=0}^{\infty} q_i$$

da

$$\begin{aligned} E[X] &= \sum_{i=0}^{\infty} iPr\{X = i\} \\ &= \sum_{i=0}^{\infty} i(Pr\{X \geq i\} - Pr\{X \geq i+1\}) \\ &= \sum_{i=0}^{\infty} Pr\{X \geq i\} \end{aligned}$$

Für $i = 1$ ist q_i:

$$q_1 = \left(\frac{n}{m}\right)$$

für $i = 2$ ist q_i:

$$q_2 = \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)$$

Also

$$\begin{aligned} q_i &= \left(\frac{n}{m}\right)\left(\frac{n-1}{m-1}\right)\cdots\left(\frac{n-i+1}{m-i+1}\right) \\ &\leq \left(\frac{n}{m}\right)^i \\ &= \alpha^i \end{aligned}$$

und

$$\begin{aligned} 1 + \sum_{i=0}^{\infty} i p_i &= 1 + \sum_{i=0}^{\infty} q_i \\ &\leq 1 + \alpha + \alpha^2 + \ldots \\ &= \frac{1}{1-\alpha} \end{aligned}$$

Falls die Hashtabelle halb voll ist, so brauchen wir im Schnitt höchstens

$$1/(1-0.5)=2$$

Vergleiche im erfolglosen Fall.

Ist die Hashtabelle zu 90% voll, so brauchen wir im Schnitt höchstens

$$1/(1-0.9)=10$$

Vergleiche im erfolglosen Fall.

Korollar Einfügen eines Elementes in eine Hashtabelle mit offener Adressierung kostet im Durchschnitt nicht mehr als $1/(1-\alpha)$ Vergleiche, falls $\alpha = n/m$ der Füllgrad ist. Wir nehmen wieder gleichmäßiges Hashing an.

Beweis Wir können nur Einfügen, wenn $\alpha < 1$. Dann Suchen wir erst einen freien Platz. Das ist genau das gleiche wie erfolglose Suche.

Satz Gegeben sei eine Hashtabelle mit Füllgrad $\alpha < 1$. Weiter sei das Hashverfahren gleichmäßig. Der Erwartungswert für die durchschnittliche Anzahl der Vergleiche im Erfolgsfall ist höchstens

$$\frac{1}{\alpha} \ln \frac{1}{1-\alpha} + \frac{1}{\alpha}$$

falls jeder Schlüssel mit der gleichen Wahrscheinlichkeit gesucht wird.

Beweis Die Suche nach k bewirkt die gleiche Folge an Vergleichen, die auch benutzt wurde, um k in die Hashtabelle einzufügen.

Nach Korollar: Falls k der $i+1$-te eingeügte Schlüssel ist, so ist die maximale Anzahl von Vergleichen bei der Suche nach k:

$$1/(1-i/m) = m/(m-i)$$

Der Durchschnitt über alle n enthaltenen Elemente ist

$$\begin{aligned}\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\ &= \frac{1}{\alpha}(H_m - H_{m-n})\end{aligned}$$

mit H_n ist die harmonische Zahl

$$\begin{aligned}H_n &= 1+1/2+1/3+1/4+\ldots+1/n \\ &= \ln n + O(1)\end{aligned}$$

Mit
$$\ln i \leq H_i \leq \ln i + 1$$
Gilt:
$$\begin{aligned}\frac{1}{\alpha}(H_m - H_{m-n}) &\leq \frac{1}{\alpha}(\ln m + 1 - \ln(m-n))\\ &= \frac{1}{\alpha}\ln\frac{m}{m-n} + \frac{1}{\alpha}\\ &= \frac{1}{\alpha}\ln\frac{1}{1-\alpha} + \frac{1}{\alpha}\end{aligned}$$

Falls die Hashtabelle halb voll ist, so brauchen wir im Schnitt höchstens

$$3.387$$

Vergleiche im erfolgreichen Fall.

Ist die Hashtabelle zu 90% voll, so brauchen wir im Schnitt höchstens

$$3.670$$

Vergleiche im erfolgreichen Fall.

Binäre Suchbäume

Benutzt für Mengen mit Operationen wie

- SEARCH
- MINIMUM,MAXIMUM
- PREDECESSOR,SUCCESSOR
- INSERT, DELETE

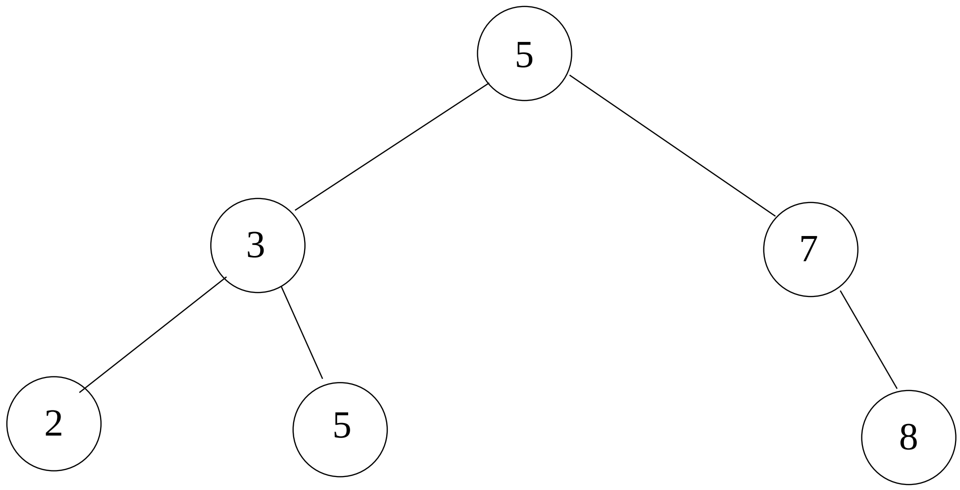
Operationen in $\Theta(h)$ mit h Höhe des Baumes. Variiert:

- $h = O(n)$ (Baum degradiert zu Liste o.ä.)
- $h = \Theta(n)$ (Baum ist balanciert.)

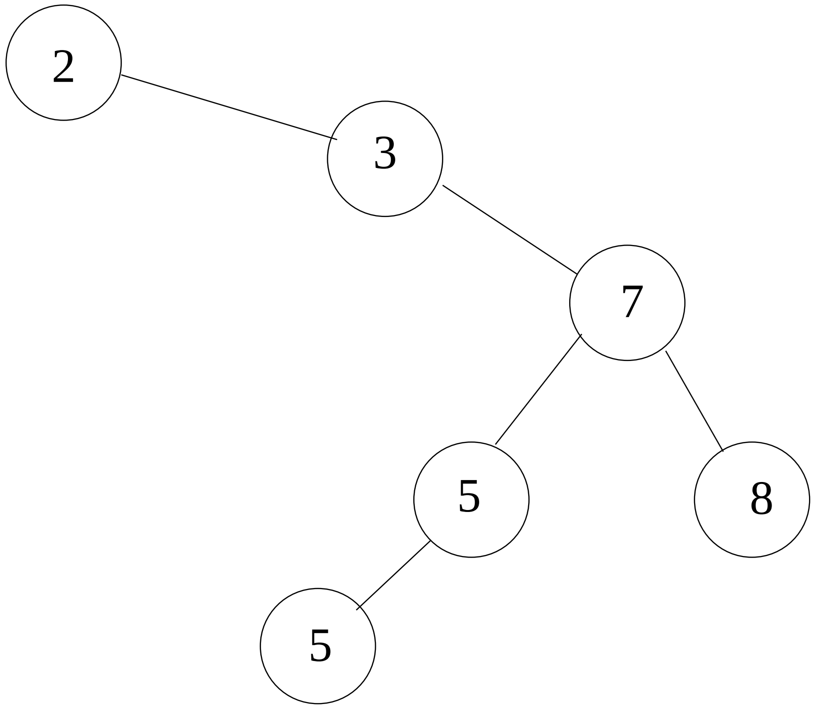
Binäre Suchbäume (BSB)

sind

1. binäre Bäume mit Attributen
   left, right, p und **key**
2. erfüllen binäre Suchbaumeigenschaft:
   - Sei x ein Knoten. Dann gilt
     (a) $key[left[x]] \leq key[x]$
     (b) $key[right[x]] \geq key[x]$



(a)



(b)

Durchläufe

Elemente in Suchbaum können leicht in aufsteigender Reihenfolge durchlaufen werden:

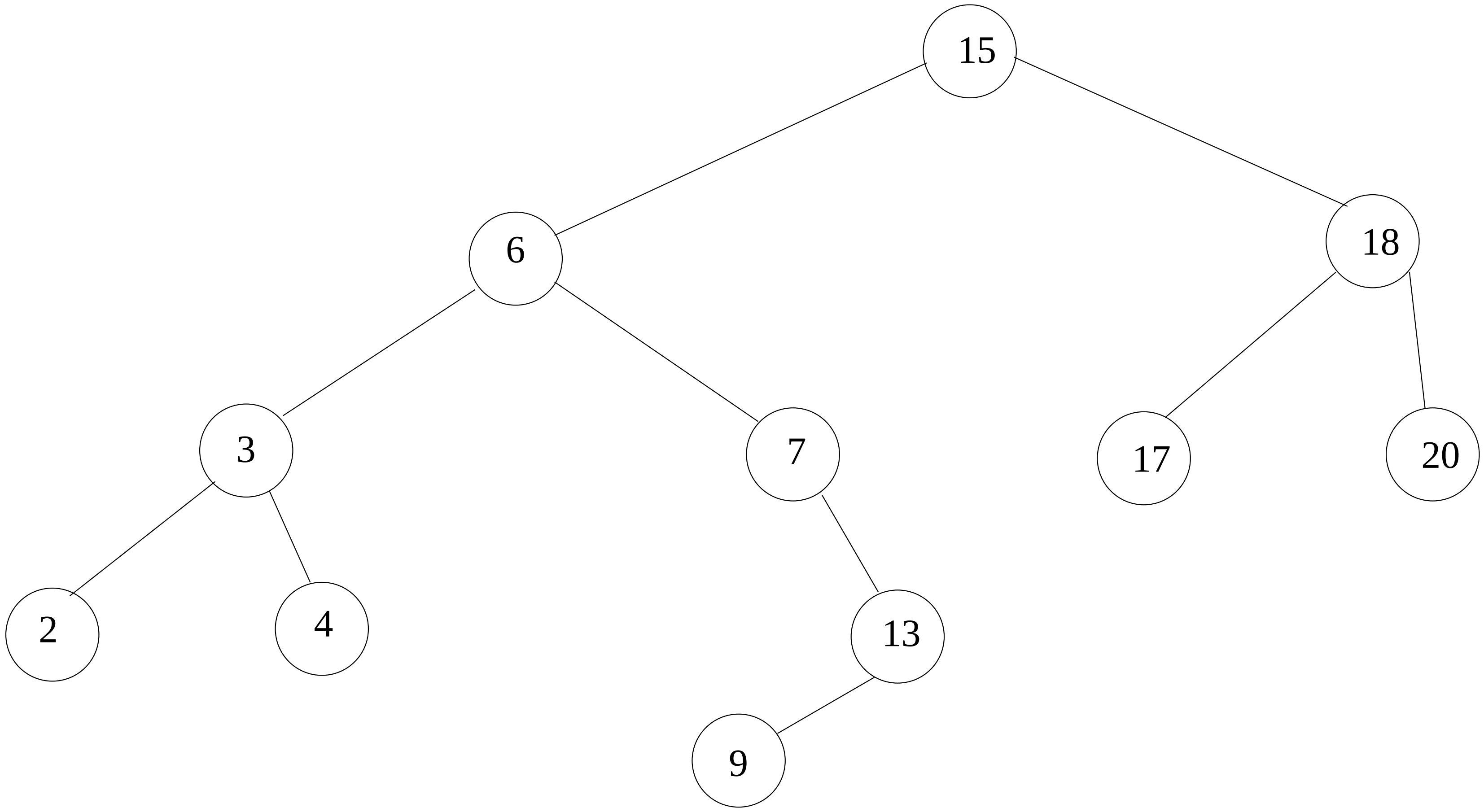
```
INORDER-TREE-WALK(x)
if x ≠ NIL
    then INORDER-TREE-WALK(left[x])
         print key[x]
         INORDER-TREE-WALK(right[x])
```

Analog: PREORDER-TREE-WALK und POSTORDER-TREE-WALK

Suchen

TREE-SEARCH(**x,k**)
if x $=$ NIL or $k =$ **key**[**x**]
    then return x
if $k <$ **key**[**x**]
    then return TREE-SEARCH(**left**[**x**],k)
    else return TREE-SEARCH(**right**[**x**],k)

Laufzeit: $O(h)$


15
6
18
3
7
17
20
2
4
13
9

Suchen (iterativ)

ITERATIVE-TREE-SEARCH(**x,k**)

```
while x ≠ NIL and k ≠ key[x]
    do if k < key[x]
        then x ← left[x]
        else x ← right[x]
return x
```