7 Datenstrukturen II

Stapel (Stacks)

Kennzeichen

Datenstruktur, bei der die modifizierenden Operationen auf vordefinierte Positionen wirken:

- DELETE(S)
  Wirkt auf das oberste Element im Stapel. Dieses Element wird aus dem Stapel *S* gelöscht und dem aufrufenden Programm übergeben.
- INSERT(S, X)
  Dem Stapel *S* wird ein neues Element *x* hinzugefügt. *x* wird an die oberste Position im Stapel geschrieben.
- LIFO-Prinzip = last-in, first-out

Wir werden die Operationen noch anders (treffender) benennen.

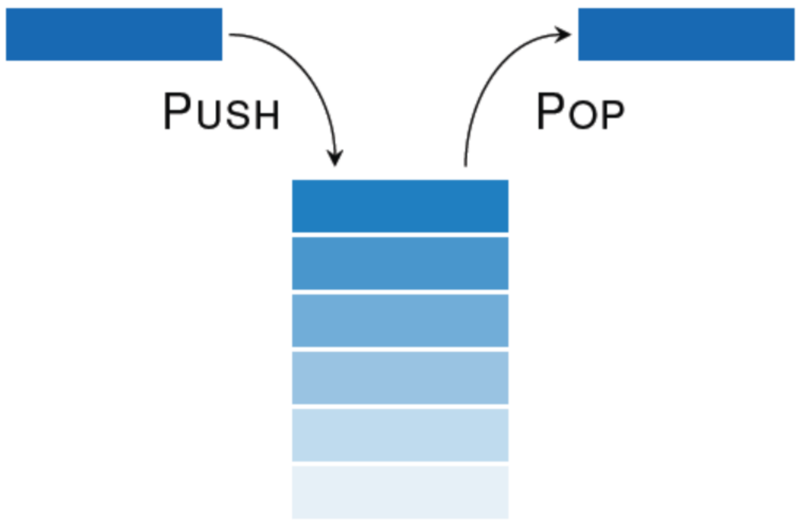
Stapel

Bemerkungen

- Wir werden Stapel mit Hilfe von Arrays implementieren.
- Umbenennung

  ... INSERT → PUSH

  ... DELETE → POP



- Damit verbunden ist [die] bildhafte Beschreibung des Vorgangs des Stapelns von z.B. Tabletts in der Cafeteria

Stapel

Notation

- S Der Stapel selbst wird mit S bezeichnet
- $S.top$ Die Position des obersten Elementes wird mit $S.top$ angesprochen
- $S[i]$ Das Element an Position i wird mit $S[i]$ angesprochen
- $S[1, ..., S.top]$ Alle Elemente des Stapels von $S[1]$ bis $S[S.top]$.

Stapel

- Leerer Stapel:

1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0

S.top = 0

- Schnappschuss:

1	2	3	4	5	6	7	8
15	6	2	9	0	0	0	0

S.top = 4

Hochschule Aalen

Stapel: Operationen auf Stapeln

```
function STACK-EMPTY(S)
    if S.top == 0 then
        return true
    else
        return false
    end if
end function
```

Stapel: Operationen auf Stapeln

1. **function** PUSH(S, x)
2. $\quad S.top \leftarrow S.top + 1$
3. $\quad S[S.top] \leftarrow x$
4. **end function**

Stapel: Operationen auf Stapeln

```
function POP(S)
    if  STACK-EMPTY(S) then
        Fehlerrückgabe „Underflow“
    else
        S.top ← S.top − 1
        Return S[S.top+1]
    end if
end function
```

Stapel

- Aktueller Zustand:

1	2	3	4	5	6	7	8
15	6	2	9	0	0	0	0

S.top = 4

- Push(S,17):

1	2	3	4	5	6	7	8
15	6	2	9	17	0	0	0

S.top = 5

Stapel

- Aktueller Zustand:

1	2	3	4	5	6	7	8
15	6	2	9	17	0	0	0

S.top = 5

- Push(S, 3):

1	2	3	4	5	6	7	8
15	6	2	9	17	3	0	0

S.top = 6

Stapel

- Aktueller Zustand:

1	2	3	4	5	6	7	8
15	6	2	9	17	3	0	0

S.top = 6

- Pop(S):

1	2	3	4	5	6	7	8
15	6	2	9	17	3	0	0

S.top = 5

Stapel

Beobachtungen

Unsere Implementation

- beachtet den Überlauf nicht,
- zerstört das gelöschte Element nicht.

Laufzeiten

- Alle Laufzeiten sind von der max. Stapelgröße sowie von der Anzahl der im Stapel enthaltenen Elemente unabhängig.
- Daher gilt für alle Algorithmen: $T(n) = \Theta(1)$

Stapel: Anwendungsbeispiel

Umgekehrte Polnische Notation (Postfixnotation)

- Operanden werden zuerst eingegeben
- Operator folgt danach

- Beispiel

$(1 + 2) \cdot (3 + 4)$ 1 2+3 4+*

Schlangen (Queues)

Kennzeichen

- Datenstruktur, bei der die modifizierenden Operationen (wie beim Stapel) auf vordefinierte Positionen wirken:

DELETE(Q)

- Wirkt auf das Element, welches sich am längsten in der Schlange befindet. Dieses Element wird aus der Schlange *Q* gelöscht und dem aufrufenden Programm übergeben.

INSERT(Q, X)

- Der Schlange *Q* wird ein neues Element *x* hinzugefügt. *x* wird an die hintere Position in der Schlange geschrieben.

- FIFO-Prinzip = first-in, first-out

- Wir werden die Operationen noch anders (treffender) benennen.

Schlangen

Bemerkungen

- Wir werden auch Schlangen mit Hilfe von Arrays implementieren.
- Umbenennung

  ... INSERT → ENQUEUE

  ... DELETE → DEQUEUE
- Damit verbunden ist die bildhafte Beschreibung der Warteschlange, an die man sich hinten anstellt und bedient wird, wenn man die erste Position erreicht hat.

Schlangen

Notation

- Q Die Schlange selbst wird mit Q bezeichnet.
- *Q.head* Die Position des nächsten zu zerstörenden (bedienenden) Elementes.
- *Q.tail* Position, an die das nächste Element zu liegen kommt.
- $Q[i]$ Das Element an Position i wird mit $Q[i]$ angesprochen.
- *Q.length* Max. Anzahl an Elementen in Q.
  Die Kapazität ist nur *Q.length* − 1!

Schlangen

- Leerer Stapel:

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	0	0	0	0	0	0

Q.head = Q.tail = 0

- Schnappschuss:

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	15	6	9	8	4	0

Q.head = 7

Q.tail = 12

Schlangen

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

Schlangen

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

Schlangen: Beispiele

- Aktueller Zustand:

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	15	6	9	8	4	0

Q.head = 7

Q.tail = 12

- ENQUEUE(Q,17):

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	15	6	9	8	4	17

Q.tail = 1

Q.head = 7

Schlangen: Beispiele

- Aktueller Zustand:

1	2	3	4	5	6	7	8	9	10	11	12
0	0	0	0	0	0	15	6	9	8	4	17

Q.tail = 1

Q.head = 7

- ENQUEUE(Q,3):

1	2	3	4	5	6	7	8	9	10	11	12
3	0	0	0	0	0	15	6	9	8	4	17

Q.tail = 2

Q.head = 7

Schlangen: Beispiele

- Aktueller Zustand:

1	2	3	4	5	6	7	8	9	10	11	12
3	0	0	0	0	0	15	6	9	8	4	17

Q.tail = 2

Q.head = 7

- ENQUEUE(Q,5):

1	2	3	4	5	6	7	8	9	10	11	12
3	5	0	0	0	0	15	6	9	8	4	17

Q.tail = 3

Q.head = 7

Schlangen: Beispiele

- Aktueller Zustand:

1	2	3	4	5	6	7	8	9	10	11	12
3	5	0	0	0	0	15	6	9	8	4	17

Q.tail = 3

Q.head = 7

- DEQUEUE(Q):

1	2	3	4	5	6	7	8	9	10	11	12
3	5	0	0	0	0	0	6	9	8	4	17

Q.tail = 3

Q.head = 8

Schlangen

Beobachtungen

- Überlauf bzw. Unterlauf werden nicht berücksichtigt.
- Das gelöschte Element wird nicht zerstört.
- Q ist leer ⇔ Q.head == Q.tail .
- Q ist voll ⇔ Q.head == Q.tail + 1 mod Q.length.

Laufzeiten

- Alle Laufzeiten sind von der max. Schlangengröße sowie von der Anzahl der in der Schlange enthaltenen Elemente unabhängig.
  Daher gilt für alle Algorithmen: $T(n) = \Theta(1)$

Wörterbücher

Definition (Wörterbuch, engl. dictionary)

Eine Datenstruktur heißt Wörterbuch, wenn sie die Operationen

- Suchen
- Einfügen
- Löschen

unterstützt.

Wörterbücher

Elemente

Jedes Element einer solchen Datenstruktur ist ein Objekt, das verschiedene Felder enthält:

- Schlüssel (key)

  Identifizierendes Element. Oft wird gefordert, dass alle Schlüssel verschieden sind.
- Daten, Satellitendaten

  Die eigentlichen Nutzdaten. Sie spielen für die grundlegenden Algorithmen keine Rolle.
- Zeiger, Verwaltungsdaten

  Zusätzliche Felder zur Verwaltung der Datenstruktur.

Wörterbücher

Notation

- D Die Datenstruktur selbst
- x Ein Element in der Datenstruktur
- $x.key$ Schlüssel des Elements x

Wörterbücher: Operationen auf dynamischen Mengen

Arten von Operationen

Operationen werden in zwei Gruppen unterschieden:

- Anfrage gibt eine Referenz auf ein Element zurück.
- Modifikation verändert die Datenstruktur oder Elemente.

Wörterbücher: Operationen auf dynamischen Mengen

Modifikationen

- INSERT(S, X) Fügt Element *x* in *D* ein
- DELETE(S, X) Löscht Element *x* aus *D*
- DELETE(S, K) Löscht Element mit Schlüsssel *k* aus *D*

Wörterbücher: Operationen auf dynamischen Mengen

Anfragen

• SEARCH(D, K)	Liefert Element x mit $x.key = k$.
• MINIMUM(D)*	Liefert Element mit kleinstem Schlüssel.
• MAXIMUM(D)*	Liefert Element mit größtem Schlüssel.
• SUCCESSOR(D, X)*	Liefert Zeiger auf das nächst größere Element oder NIL, falls $x.key$ maximaler Schlüssel ist.
• PREDECESSOR(D, X)*	Liefert Zeiger auf das nächst kleinere Element oder NIL, falls $x.key$ minimaler Schlüssel ist.

*Für diese Operationen muss eine Ordnung unter den Schlüsseln existieren.

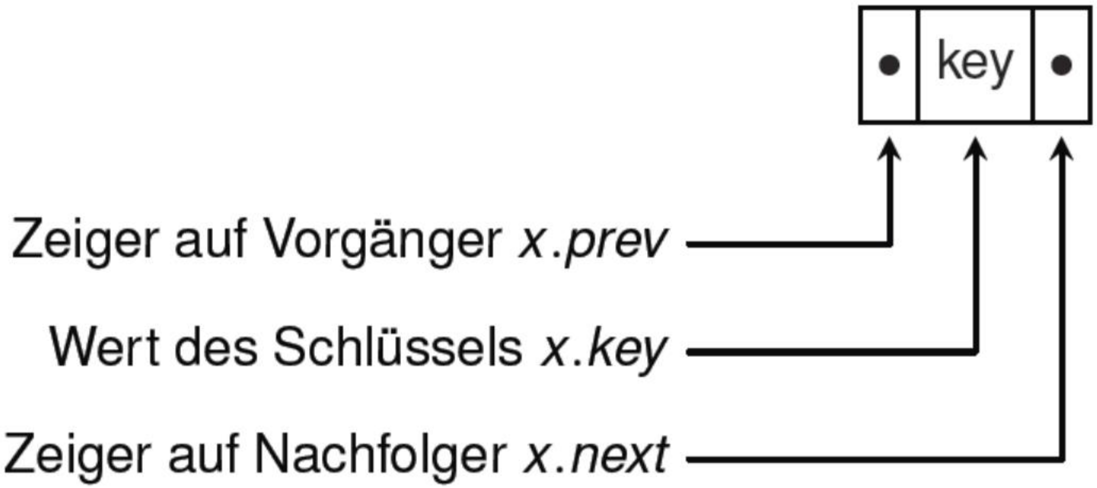
Verkettete Listen (linked lists)

Kennzeichen

- Elemente sind linear angeordnet.
- Die Anordnung der Elemente ergibt sich durch Verkettung (Zeiger)
  $\Rightarrow$ Position eines Elementes kann nicht errechnet werden
- Verkettete Listen unterstützen alle vorher definierten Operationen (Wörterbuchfunktionen).
- Allerdings: Diese Operationen sind u.U. nicht sehr effizient realisierbar.

Verkettete Listen

- Auch hier können Listenelemente wieder Nutzdaten enthalten. Diese sind für die Implementation der grundlegenden Algorithmen unerheblich.



Verkettete Listen

Notation

- *L* Die Liste selbst wird mit *L* bezeichnet.
- *L.head* Zeiger auf das erste Element der Liste (Listenkopf).
- *L.tail* Zeiger auf das letzte Element der Liste (Listenende).
- *x* Ein Listenelement.
- *x.key* Der Schlüssel des Listenelementes *x* .
- *x.next* Zeiger auf den Nachfolger in der Liste.
- *x.prev* Zeiger auf den Vorgänger in der Liste.

Verkettete Listen

Notation

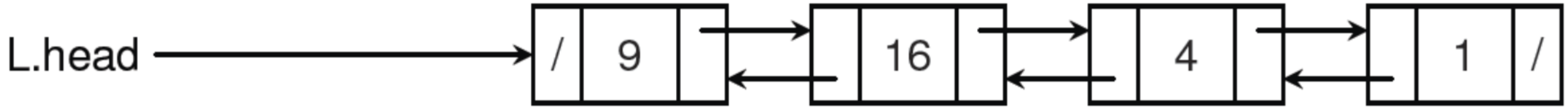
- *L* Die Liste selbst wird mit *L* bezeichnet.
- *L.head* Zeiger auf das erste Element der Liste (Listenkopf).
- *L.tail* Zeiger auf das letzte Element der Liste (Listenende).
- *x* Ein Listenelement.
- *x .key* Der Schlüssel des Listenelementes *x* . *x .next* Zeiger auf den Nachfolger in der Liste. *x .prev* Zeiger auf den Vorgänger in der Liste.

Verkettete Listen

- Leere Liste:

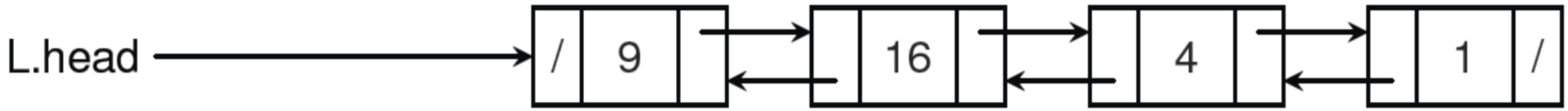
  L.head /

- Schnappschuss:

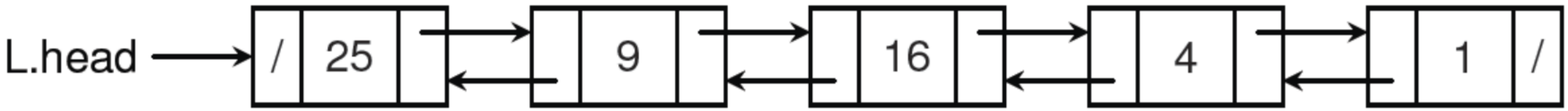


Verkettete Listen: Beispiele

- Aktueller Zustand:

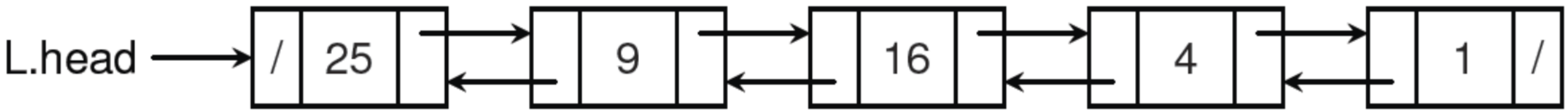


- INSERT(L,x) mit x.key = 25:

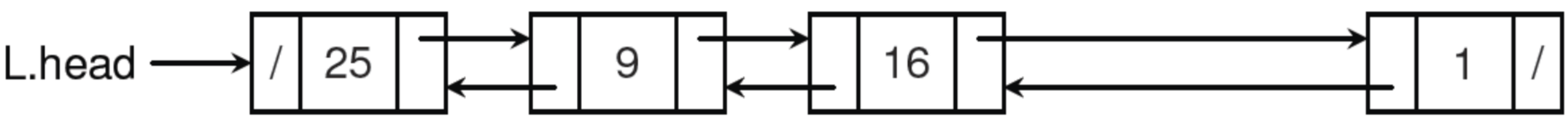


Verkettete Listen: Beispiele

- Aktueller Zustand:



- DELETE(L,x) mit x.key = 4:



Verkettete Listen

Bemerkungen

- $x\,.prev = NIL$ $\Rightarrow$ Element x hat keinen Vorgänger
  $\Rightarrow$ Element ist der Listenkopf!
- $x\,.next = NIL$ $\Rightarrow$ Element x hat keinen Nachfolger
  $\Rightarrow$ Element ist das Listenende!
- $L.head = NIL$ $\Rightarrow$ Liste L ist leer
- Einfach verkettete Listen sind Spezialfall von doppelt verketteten Listen. Feld $x\,.prev$ wird nicht benutzt.
- Listen können sortiert sein.
- Zyklische Listen $\Leftrightarrow L.head\,.prev = L.tail \;\wedge\; L.tail\,.next = L.head$

Hochschule Aalen

Verkettete Listen: Operationen

Suche eines Elements

```
function LIST-SEARCH(L, k)
    x ← L.head
    while x /= NIL and x .key /= k do
        x ← x .next
    end while
    return x
end function
```

Verkettete Listen: Operationen

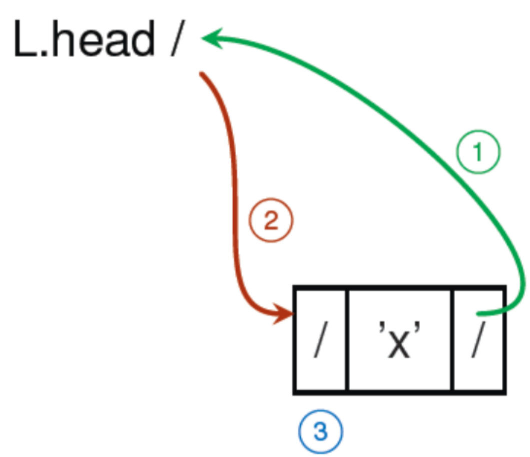
Einfügen eines Elements

```
function LIST-INSERT(L, x)
    x .next ← L.head
    if L.head /= NIL then
        L.head .prev ← x
    end if
    L.head ← x
    x .prev ← NIL
end function
```

Verkettete Listen: Operationen

Einfügen eines Elements in leere Liste

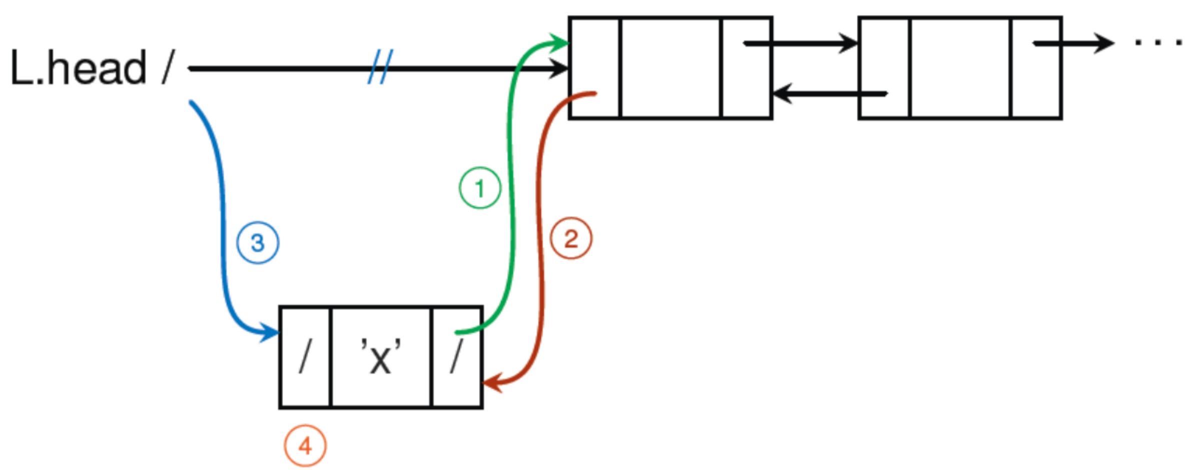
```
function LIST-INSERT(L, x)
    x .next ← L.head
    if L.head /= NIL then
        L.head .prev ← x
    end if
    L.head ← x
    x .prev ← NIL
end function
```



Verkettete Listen: Operationen

Einfügen eines Elements in nicht-leere Liste

```
function LIST-INSERT(L, x)
    x .next ← L.head
    if L.head /= NIL then
        L.head .prev ← x
    end if
    L.head ← x
    x .prev ← NIL
end function
```



Verkettete Listen: Operationen

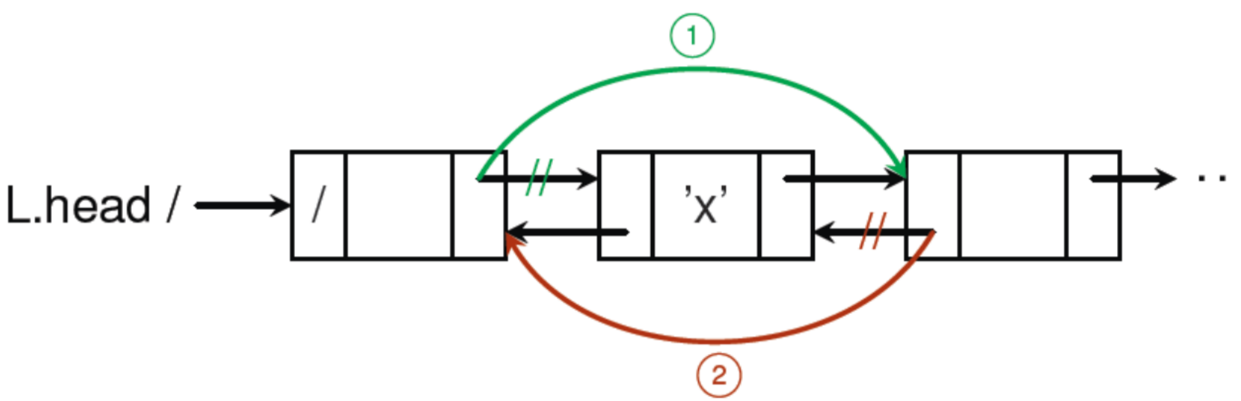
Löschen eines Elements

```
function LIST-DELETE(L, x)
    if x .prev /= NIL then
        x .prev .next ← x .next
    else
        L.head ← x .next
    end if
    if x .next /= NIL then
        x .next .prev ← x .prev
    end if
end function
```

Verkettete Listen: Operationen

Löschen eines Elements aus der Mitte

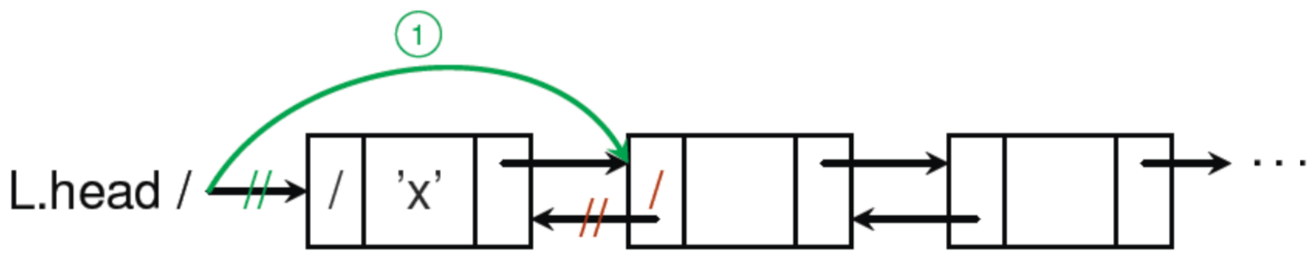
```
function LIST-DELETE(L, x)
    if x .prev /= NIL then
        x .prev .next ← x .next
    else
        L.head ← x .next
    end if
    if x .next /= NIL then
        x .next .prev ← x .prev
    end if
end function
```



Verkettete Listen: Operationen

Löschen eines Elements am Anfang

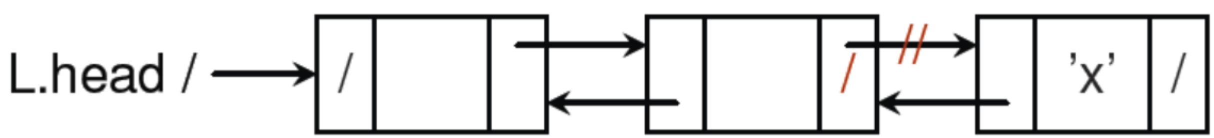
```
function LIST-DELETE(L, x)
    if x .prev /= NIL then
        x .prev .next ← x .next
    else
        L.head ← x .next
    end if
    if x .next /= NIL then
        x .next .prev ← x .prev
    end if
end function
```



Verkettete Listen: Operationen

Löschen eines Elements am Ende

```
function LIST-DELETE(L, x)
    if x.prev /= NIL then
        x.prev.next ← x.next
    else
        L.head ← x.next
    end if
    if x.next /= NIL then
        x.next.prev ← x.prev
    end if
end function
```



Verkettete Listen

Bemerkungen

- Die Algorithmen sind etwas „unschön“ durch die vielen Abfragen
- Die Abfragen beziehen sich immer auf Randelemente
- Verbesserung: Einführen eines speziellen Wächterelementes (sentinel)
- Alle Verweise auf *NIL* werden durch Verweise auf den Wächter ersetzt
- Damit befinden sich „normale“ Elemente immer in der Mitte

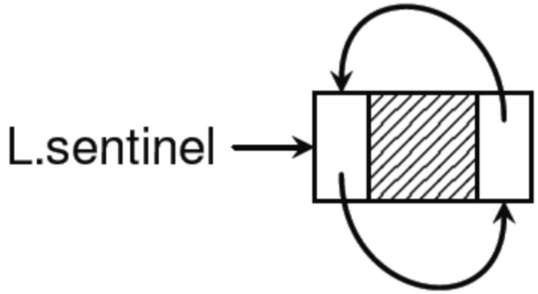
Verkettete Listen mit Wächter (sentinel)

Codebeispiel

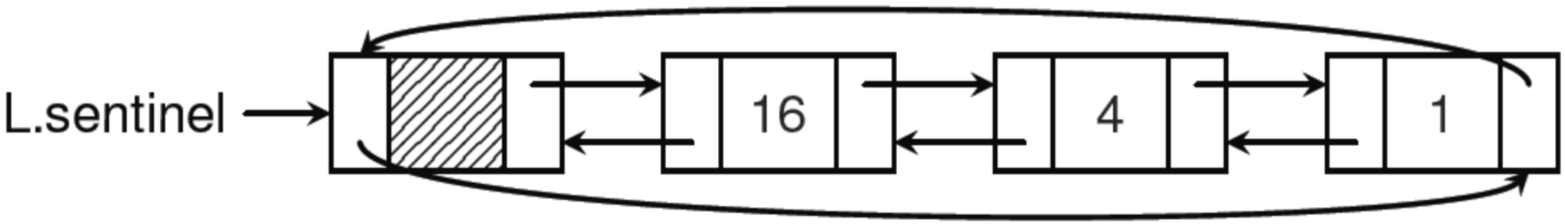
```java
public class ListElement{
    public int key;
    public ListElement next;
    public ListElement prev;
    public ListElement(int k)
    {
        key = k;
        prev = null;
        next = null;
    }
}

public class DLListS {
    private ListElement sentinel;
    public DLListS()
    {
        sentinel = new ListElement(0);
        sentinel.next = sentinel;
        sentinel.prev = sentinel;
    }
}
```

Verkettete Listen

- Leere Liste:



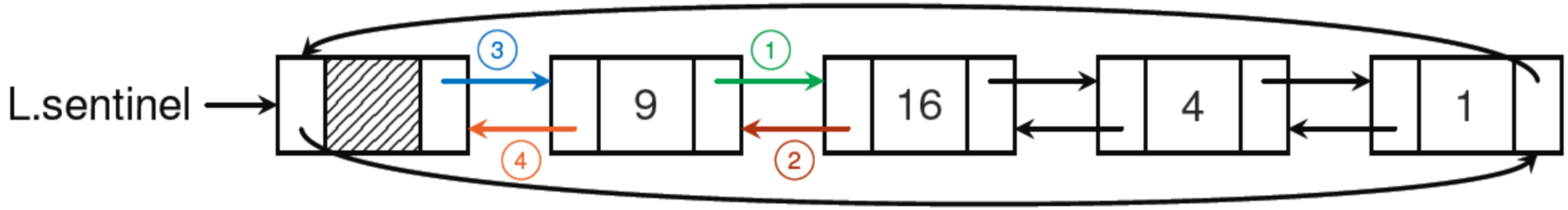
- Schnappschuss:



Verkettete Listen

Einfügen eines Elements mit Schlüssel „9“

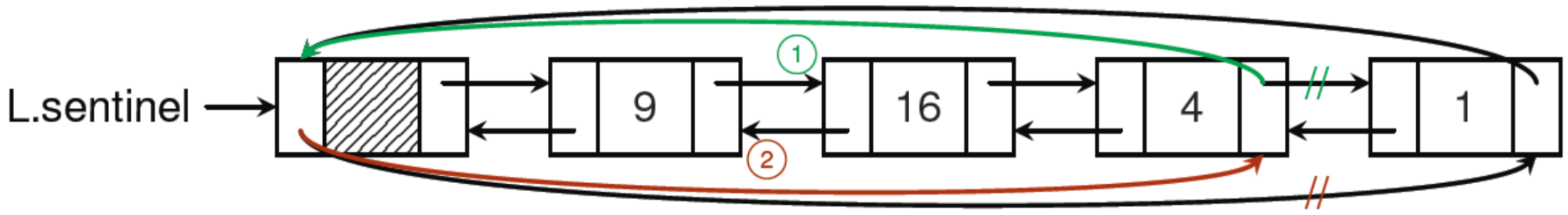
1. **function** LIST-INSERT’(L, x)
2. *x .next* ← *L.sentinel .next*
3. *L.sentinel .next .prev* ← *x*
4. *L.sentinel .next* ← *x*
5. *x .prev* ← *L.sentinel*
6. **end function**



Verkettete Listen

Löschen des Elements mit Schlüssel „1“

1. **function** LIST-DELETE’(L, x)
2. $x.prev.next \leftarrow x.next$
3. $x.next.prev \leftarrow x.prev$
4. **end function**



Verkettete Listen

Suchen eines Elements in der Liste

```
function LIST-SEARCH'(L, k)
    x ← L.sentinel .next
    while x /= L.sentinel and x .key /= k do
        x ← x .next
    end while
    return x
end function
```

Verkettete Listen

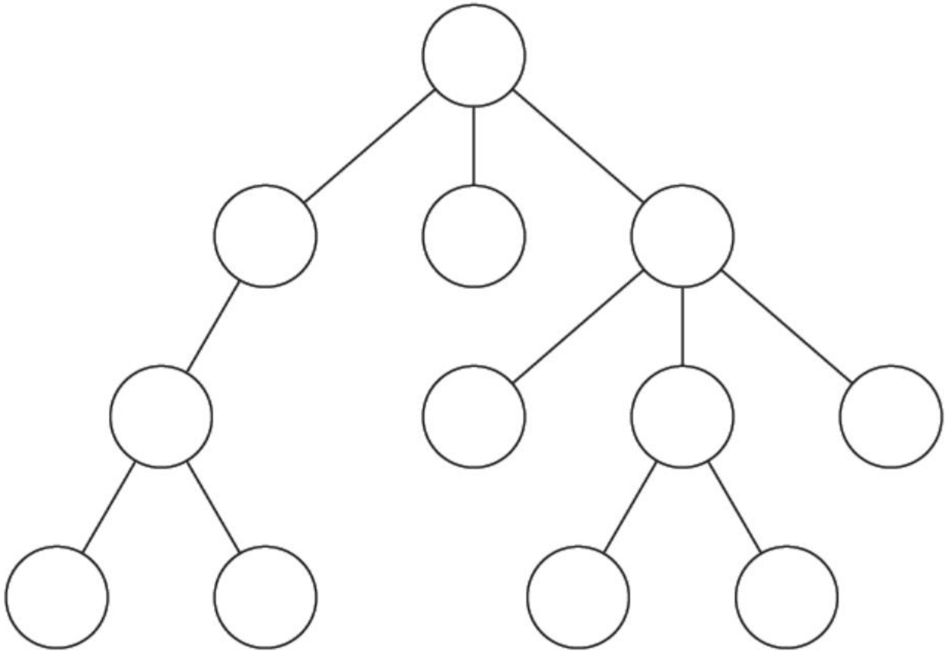
Beobachtungen

- Aus der Liste wird eine zirkuläre Liste (Ring).
- Die leere Liste besteht nur aus dem Wächter.
- Liste ist leer $\Leftrightarrow$ L.sentinel.next == L.sentinel.prev

Baumstrukturen

Definition (Baum)

- Ein Baum T ist eine nichtleere Menge N , E von Knoten N und Kanten E .



Baumstrukturen: Terminologie

- Knoten sind einfache Objekte, die typischerweise einen Namen haben und Information tragen können.
- Wir repräsentieren Knoten meist durch ihre Schlüsselwerte.
- Sie haben eine endliche Zahl von Nachfolgern (Kinder, children).
- Kindknoten einer Ebene heißen Geschwister (siblings).
- Der direkte Vorgänger eines Knotens heißt Elternknoten (Vater, parent).
- Knoten ohne Kinder heißen Blattknoten.
- Ein Knoten ist als Wurzel (root) ausgezeichnet. Er ist der einzige Knoten ohne Vorgänger

Baumstrukturen: Terminologie

- Jeder Knoten ist die Wurzel eines Unterbaumes (der evtl. leer ist).
- Die Anzahl von Kindern eines Knotens heißt Ordnung d des Knotens.
- Die Ordnung d eines Baumes ist die max. Ordnung unter seinen Knoten.
- $d = 2$ → Binärbäume
- $d > 2$ → Vielwegbäume
- Ein Baum heißt vollständig, wenn er auf jedem Niveau (auf jeder Ebene) - bis auf evtl. die letzte - die max. Anzahl Knoten hat.
- Wenn unter den Knoten eines Baums eine Ordnung existiert, so heißt der Baum sortiert.

Baumstrukturen: Terminologie

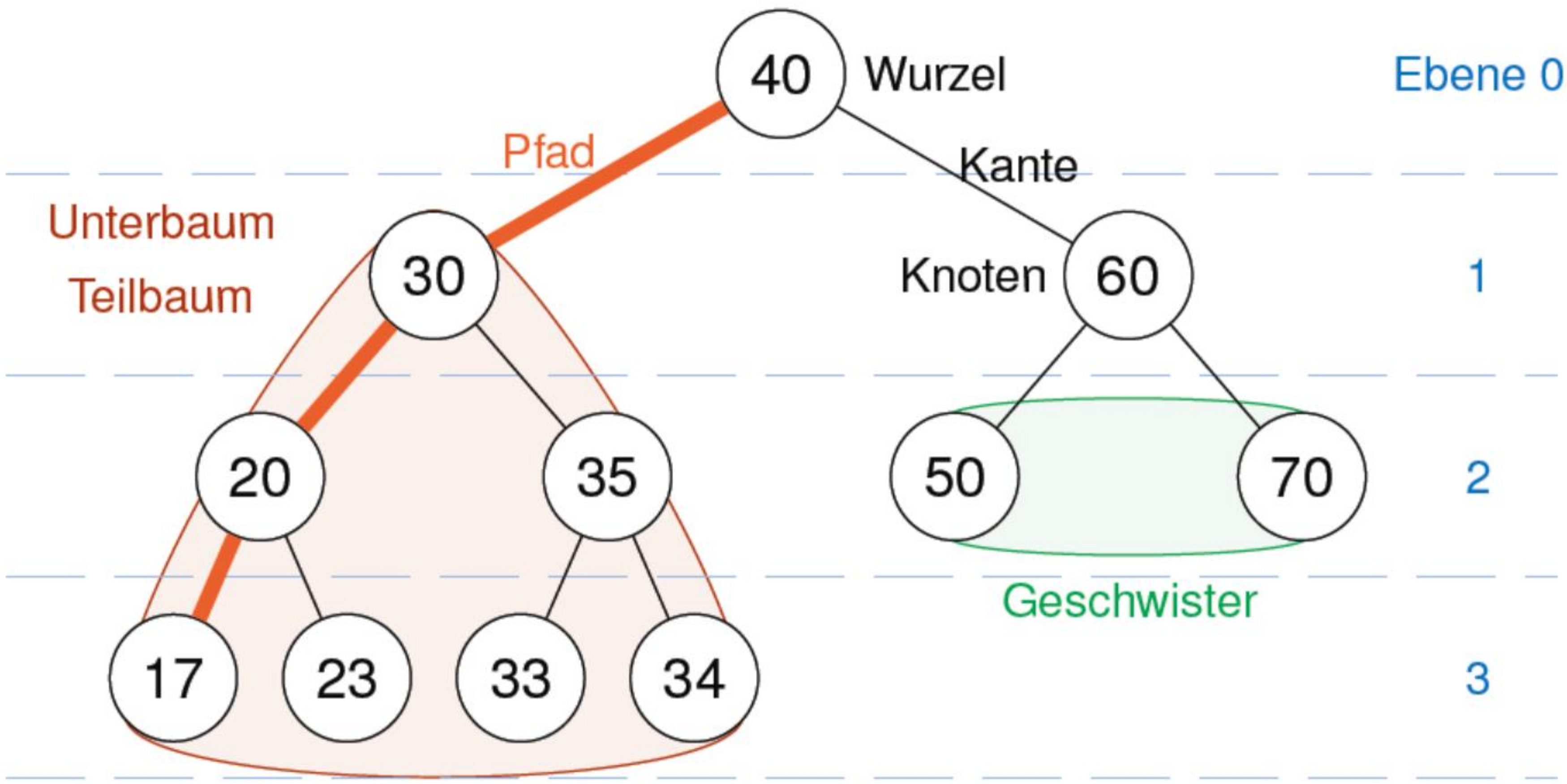
- Ein Pfad (Weg, path) ist eine Folge von Knoten

  $$p0, p1, \ldots pk$$

  mit pi ist Kind von $pi - 1$.

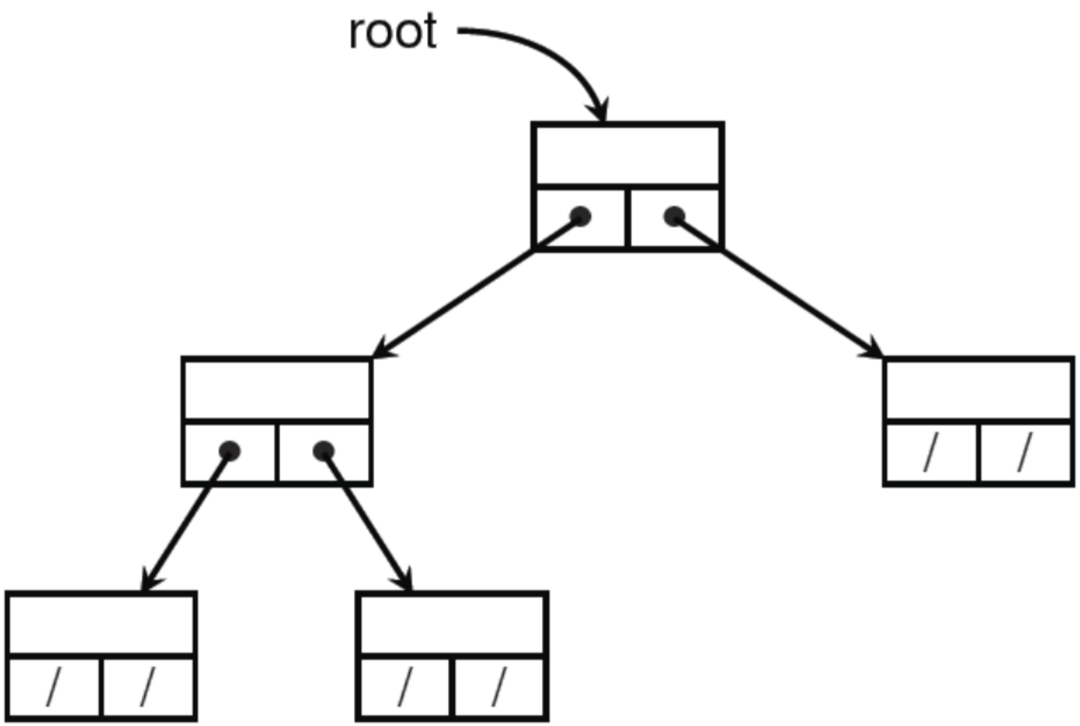
- Jeder Nichtwurzelknoten ist über genau einen Pfad mit der Wurzel verbunden.

Baumstrukturen: Darstellung



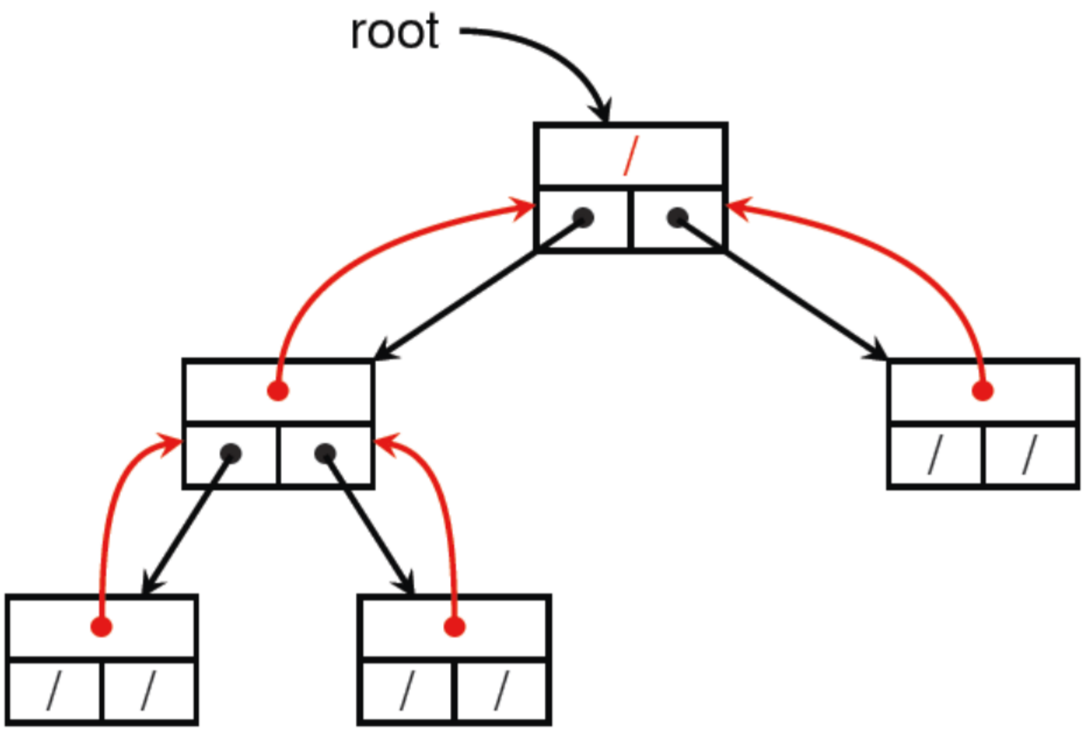
Baumstrukturen

- Knoten mit Zeigern ohne Elternzeiger



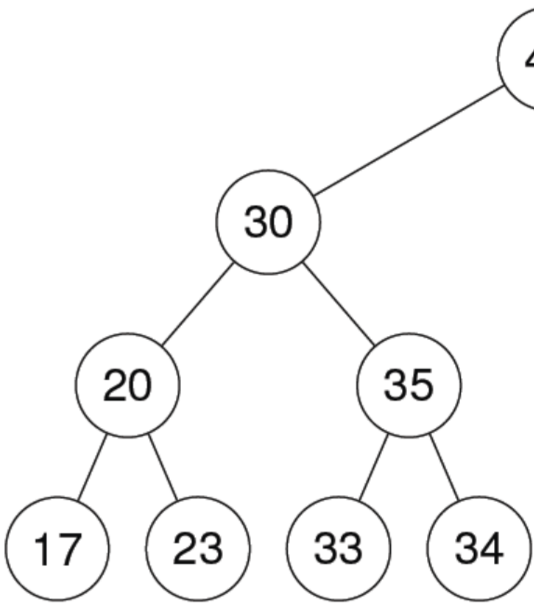
Baumstrukturen

- Knoten mit Zeigern mit Elternzeiger



Baumstrukturen

- Einbettung in ein Array



40	30	60	20	35	50	70	17	23	33	34

Baumstrukturen

- Adressrechnung bei Einbettung in ein Array

- Parent(i): $i/2$
- Left(i): $2i$
- Right(i): $2i + 1$

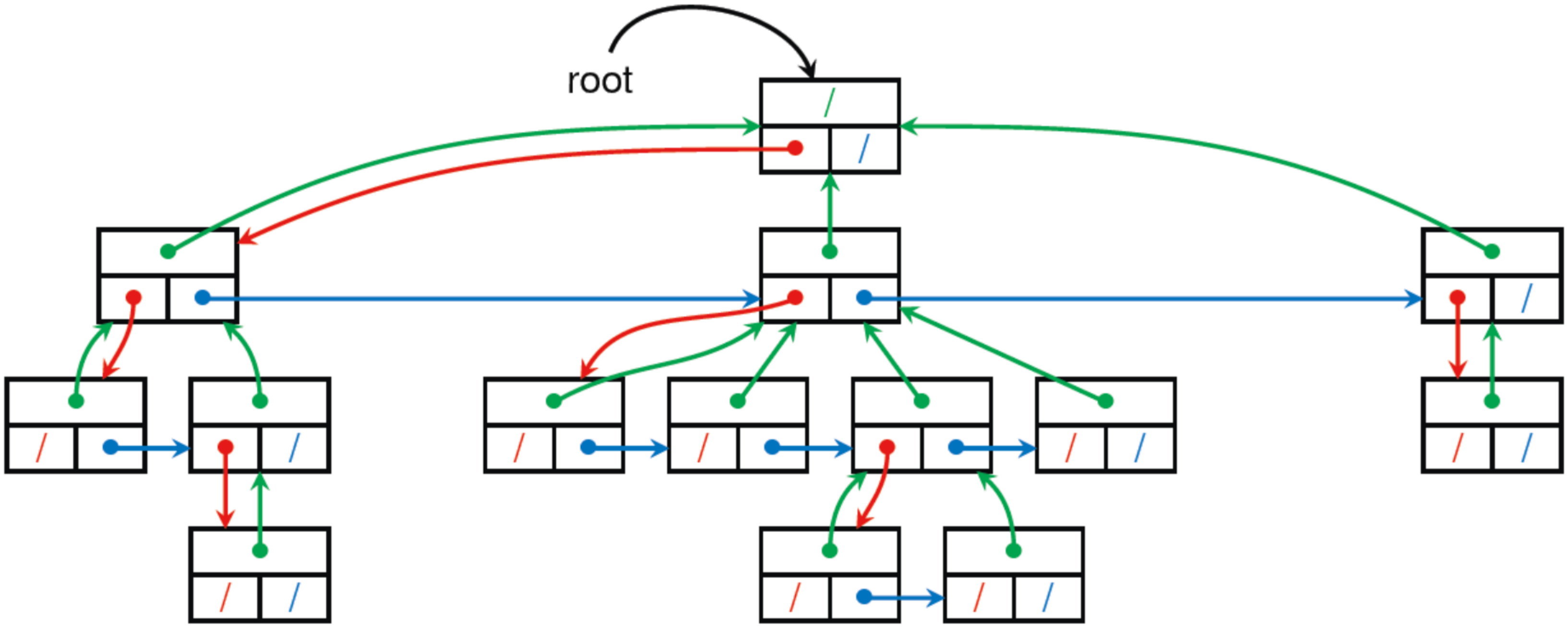
Baumstrukturen

Verzeigerung

- Für Vielwegbäume mit bel. aber fester Ordnung *d* ist dieses Schema leicht anwendbar:

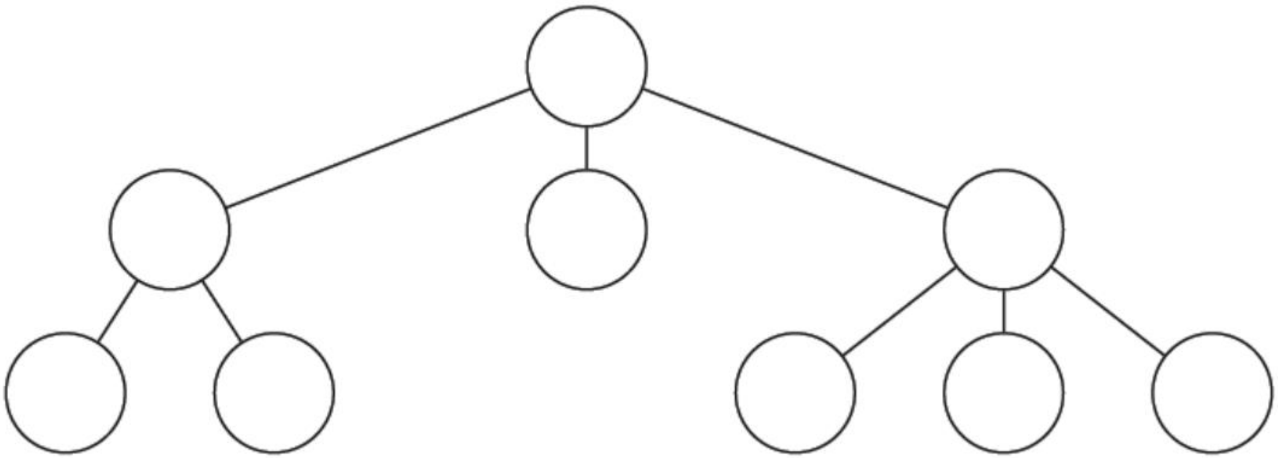
Baumstrukturen

- Alternative: „left child, right sibling“



Baumstrukturen

- Einbettung in ein Array möglich



1	2	3	4	5	6	7	8	9	Arrayindex
									Schlüsselwerte
/	1	1	1	2	2	4	4	4	Elternknoten
2	5	/	7	/	/	/	/	/	Links (von
4	6	/	9	/	/	/	/	/	Rechts (bis)

Zusammenfassung

- Vorstellung einiger grundlegender und wichtiger Datenstrukturen
- Erweiterung des Bekannten: doppelt verkettete Listen, Wächter, . . .
- Bäume und Implementierungsmöglichkeiten
- Anwendung: Verwendung dieses Wissens zur Implementation einer effizienteren Wörterbuchstruktur (nächstes Kapitel).