technische universität
dortmund

LS 2 /
Informatik

Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Vorgehensweise bei dynamischer Programmierung

1. Bestimme rekursive Struktur einer optimalen Lösung.

2. Entwerfe rekursive Methode zur Bestimmung des Wertes einer optimalen Lösung.

3. Transformiere rekursiv Methode in eine iterative (bottom-up) Methode zur Bestimmung des Wertes einer optimalen Lösung.

4. Bestimmen aus dem Wert einer optimalen Lösung und in 3. ebenfalls berechneten Zusatzinformationen eine optimale Lösung.

Dynamische Programmierung

Das Rucksackproblem

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Dynamische Programmierung

Das Rucksackproblem

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Beispiel

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

Dynamische Programmierung

Das Rucksackproblem

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Beispiel

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekt 1 und 3 passen in den Rucksack und haben Gesamtwert 13

Dynamische Programmierung

Das Rucksackproblem

- Rucksack mit begrenzter Kapazität
- Objekte mit unterschiedlichem Wert und unterschiedlicher Größe
- Wir wollen Objekte von möglichst großem Gesamtwert mitnehmen

Beispiel

- Rucksackgröße 6

Größe	5	2	1	3	7	4
Wert	11	5	2	8	14	9

- Objekt 1 und 3 passen in den Rucksack und haben Gesamtwert 13
- Objekt 2,3 und 4 passen und haben Gesamtwert 15

Dynamische Programmierung

Das Rucksackproblem

- Eingabe: Anzahl der Objekte n
  Für jedes Objekt i seine ganzzahlige Größe g[i] und seinen ganzzahligen Wert v[i]
  Rucksackgröße W
- Ausgabe: $S \subseteq \{1,\ldots,n\}$, so dass $\sum_{i \in S} g[i] \leq W$ und $\sum_{i \in S} v[i]$ maximal ist

Dynamische Programmierung

Lösungsansatz

- Bestimme zunächst den Wert einer optimalen Lösung
- Leite dann die Lösung selbst aus der Tabelle des dynamischen Programms her

Dynamische Programmierung

Herleiten der Rekursion

- Sei $O \subseteq \{1,..,i\}$ eine optimale Lösung für das Rucksackproblem mit Objekten 1,..,i und Rucksackgröße j
- Sei Opt(i,j) der Wert einer solchen optimalen Lösung
- Gesucht: Opt(n,W)

Dynamische Programmierung

Lemma 25 (Struktur einer optimalen Lösung des Rucksackproblems)

- Sei $O \subseteq \{1,..,i\}$ eine optimale Lösung für das Rucksackproblem mit Objekten 1,..,i und Rucksackgröße j. Es bezeichne Opt(i,j) den Wert dieser optimalen Lösung. Dann gilt:
- (a) Ist Objekt i in O enthalten, so ist $O \setminus \{i\}$ eine optimale Lösung für das Rucksackproblem mit Objekten 1,…,i-1 und Rucksackgröße j-g[i]. Insbesondere gilt Opt(i,j)=v[i] + Opt(i-1, j-g[i]).
- (b) Ist Objekt i nicht in O enthalten, so ist O eine optimale Lösung für das Rucksackproblem mit Objekten 1,…,i-1 und Rucksackgröße j. Insbesondere gilt Opt(i,j) = Opt(i-1, j).

Dynamische Programmierung

Beweis

- (a) z.z.: Ist Objekt i in O enthalten, so ist O \ {i} eine optimale Lösung für das Rucksackproblem mit Objekten 1,…,i-1 und Rucksackgröße j-g[i]. Insbesondere gilt Opt(i,j)=v[i] + Opt(i-1, j-g[i]).

Dynamische Programmierung

Beweis

- (a) z.z.: Ist Objekt i in O enthalten, so ist O \ {i} eine optimale Lösung für das Rucksackproblem mit Objekten 1,…,i-1 und Rucksackgröße j-g[i]. Insbesondere gilt Opt(i,j)=v[i] + Opt(i-1, j-g[i]).
- Für i=1 ist die Aussage offensichtlich korrekt. Sei also i>1.

Dynamische Programmierung

Beweis

- (a) z.z.: Ist Objekt i in O enthalten, so ist O \ {i} eine optimale Lösung für das Rucksackproblem mit Objekten 1,…,i-1 und Rucksackgröße j-g[i]. Insbesondere gilt Opt(i,j)=v[i] + Opt(i-1, j-g[i]).
- Für i=1 ist die Aussage offensichtlich korrekt. Sei also i>1.
- Sei O eine optimale Lösung mit Kosten Opt(i,j), die Objekt i enthält. Da Objekt i Größe g[i] hat, gilt sicher, dass O \ {i} eine Gesamtgröße von höchstens j-g[i] hat. Damit ist O \ {i} eine gültige Lösung für das Rucksackproblem mit Objekten 1,…,i-1 und Rucksackgröße j-g[i].

Dynamische Programmierung

Beweis

- (a) z.z.: Ist Objekt i in O enthalten, so ist O \ {i} eine optimale Lösung für das Rucksackproblem mit Objekten 1,…,i-1 und Rucksackgröße j-g[i]. Insbesondere gilt Opt(i,j)=v[i] + Opt(i-1, j-g[i]).
- Für i=1 ist die Aussage offensichtlich korrekt. Sei also i>1.
- Sei O eine optimale Lösung mit Kosten Opt(i,j), die Objekt i enthält. Da Objekt i Größe g[i] hat, gilt sicher, dass O \ {i} eine Gesamtgröße von höchstens j-g[i] hat. Damit ist O \ {i} eine gültige Lösung für das Rucksackproblem mit Objekten 1,…,i-1 und Rucksackgröße j-g[i].

Dynamische Programmierung

Beweis

- Annahme: O \ {i} hat Kosten R = Opt(i,j) – v[i] und ist keine optimale Lösung für das Rucksackproblem mit Objekten 1,..,i-1 und Rucksackgröße j-g[i].

Dynamische Programmierung

Beweis

- Annahme: O \ {i} hat Kosten R = Opt(i,j) – v[i] und ist keine optimale Lösung für das Rucksackproblem mit Objekten 1,..,i-1 und Rucksackgröße j-g[i].
- Dann gibt es eine bessere Lösung O* für dieses Problem mit Kosten R*>R. Weiterhin ist O*∪ {i} eine gültige Lösung für das Rucksackproblem mit Objekten 1,…,i und Rucksackgröße j. Die Kosten dieser Lösung sind R*+v[i] > R + v[i] = Opt(i,j). Widerspruch zur Optimalität von O.

Dynamische Programmierung

Beweis

- Annahme: O \ {i} hat Kosten R = Opt(i,j) – v[i] und ist keine optimale Lösung für das Rucksackproblem mit Objekten 1,..,i-1 und Rucksackgröße j-g[i].
- Dann gibt es eine bessere Lösung O* für dieses Problem mit Kosten R*>R. Weiterhin ist O*∪ {i} eine gültige Lösung für das Rucksackproblem mit Objekten 1,…,i und Rucksackgröße j. Die Kosten dieser Lösung sind R*+v[i] > R + v[i] = Opt(i,j). Widerspruch zur Optimalität von O.
- Damit ergibt sich sofort Opt(i,j)= v[i] + Opt(i-1,j-g[i]).

Dynamische Programmierung

Beweis

- Annahme: O \ {i} hat Kosten R = Opt(i,j) – v[i] und ist keine optimale Lösung für das Rucksackproblem mit Objekten 1,..,i-1 und Rucksackgröße j-g[i].
- Dann gibt es eine bessere Lösung O* für dieses Problem mit Kosten R*>R. Weiterhin ist O*∪ {i} eine gültige Lösung für das Rucksackproblem mit Objekten 1,…,i und Rucksackgröße j. Die Kosten dieser Lösung sind R*+v[i] > R + v[i] = Opt(i,j). Widerspruch zur Optimalität von O.
- Damit ergibt sich sofort Opt(i,j)= v[i] + Opt(i-1,j-g[i]).

Dynamische Programmierung

Beweis

- (b) analog zu (a).

Dynamische Programmierung

Korollar 26 (Rekursion zur Berechnung der Kosten einer opt. Lösung)

- Es gilt
- $Opt(0,j) = 0$ für $0 \leq j \leq W$,
- $Opt(i,j) = \max\{Opt(i-1,j), v[i] + Opt(i-1,j-g[i])\}$, falls $i>0$ und $g[i] \leq j$, und
- $Opt(i,j) = Opt(i-1,j)$, sonst.

Dynamische Programmierung

Korollar 26 (Rekursion zur Berechnung der Kosten einer opt. Lösung)

- Es gilt
- Opt(0,j)= 0 für $0 \leq j \leq W$,
- Opt(i,j) = max{Opt(i-1,j), v[i] + Opt(i-1,j-g[i])}, falls i>0 und $g[i] \leq j$, und
- Opt(i,j) = Opt(i-1,j), sonst.

Beweis

- Aufgrund von Lemma 25 wissen wir, dass die Kosten einer optimalen Lösung entweder durch Opt(i-1,j) oder durch v[i] + Opt(i-1, j-g[i]) gegeben sind. Letzterer Fall kann nur auftreten, wenn $g[i] \leq j$ ist. Beide Werte entsprechen außerdem den Kosten einer zulässigen Lösung. Dies zeigt die Korrektheit der Rekursion.

Dynamische Programmierung

Wenn Objekt i nicht in den Rucksack, sind in der optimalen Lösung nur Objekte aus {1,..,i-1}

Rekursion

- Wenn j<g[i] dann Opt(i,j) = Opt(i-1,j)
- Sonst,
  Opt(i,j) = max{Opt(i-1,j), v[i] + Opt(i-1,j-g[i])}

Rekursionsabbruch

- Opt(0,j)= 0 für $0 \leq j \leq W$

Dynamische Programmierung

Rekursion

- Wenn j<g[i] dann Opt(i,j) = Opt(i-1,j)
- Sonst,
  Opt(i,j) = max{Opt(i-1,j), v[i] + Opt(i-1,j-g[i])}

Sonst ist entweder i in der optimalen Lösung oder die beste Lösung besteht aus Objekten aus {1,..,i-1}

Rekursionsabbruch

- Opt(0,j)= 0 für $0 \leq j \leq W$

Dynamische Programmierung

Rekursion

- Wenn j<g[i] dann Opt(i,j) = Opt(i-1,j)
- Sonst,
  Opt(i,j) = max{Opt(i-1,j), v[i]

Gibt es keine Objekte, so kann auch nichts in den Rucksack gepackt werden

Rekursionsabbruch

- Opt(0,j)= 0 für $0 \leq j \leq W$

Dynamische Programmierung

Rucksack(n,g,v,W)

1. **new array** Opt[0,..,n][0,..,W]
2. **for** j ← 0 **to** W **do**
3. Opt[0,j] ← 0
4. **for** i ← 0 **to** n **do**
5. **for** j ← 0 **to** W **do**
6. Berechne Opt[i,j] nach Rekursion
7. **return** Opt[n,W]

Dynamische Programmierung

Rucksack(n,g,v,W)

1. **new array** Opt[0,..,n][0,..,W]
2. **for** j ← 0 **to** W **do**
3. Opt[0,j] ← 0
4. **for** i ← 0 **to** n **do**
5. **for** j ← 0 **to** W **do**
6. Berechne Opt[i,j] nach Rekursion
7. **return** Opt[n,W]

Laufzeit

- O(nW)

Dynamische Programmierung

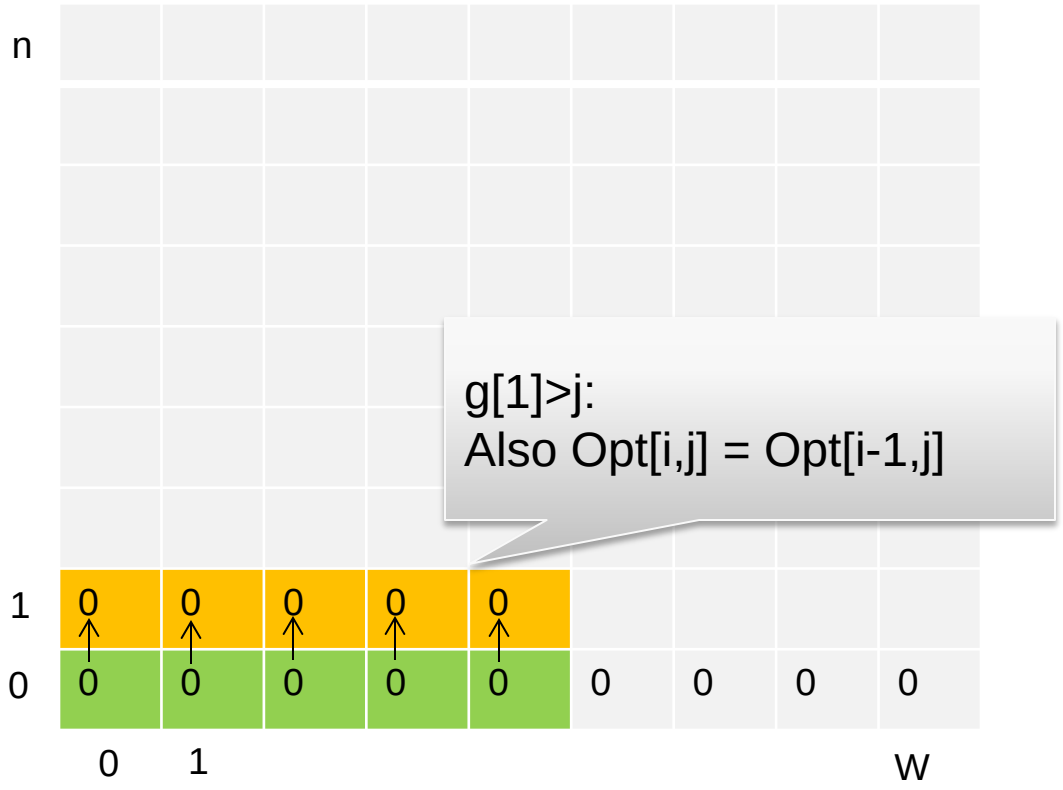
Beispiel

n									
1									
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

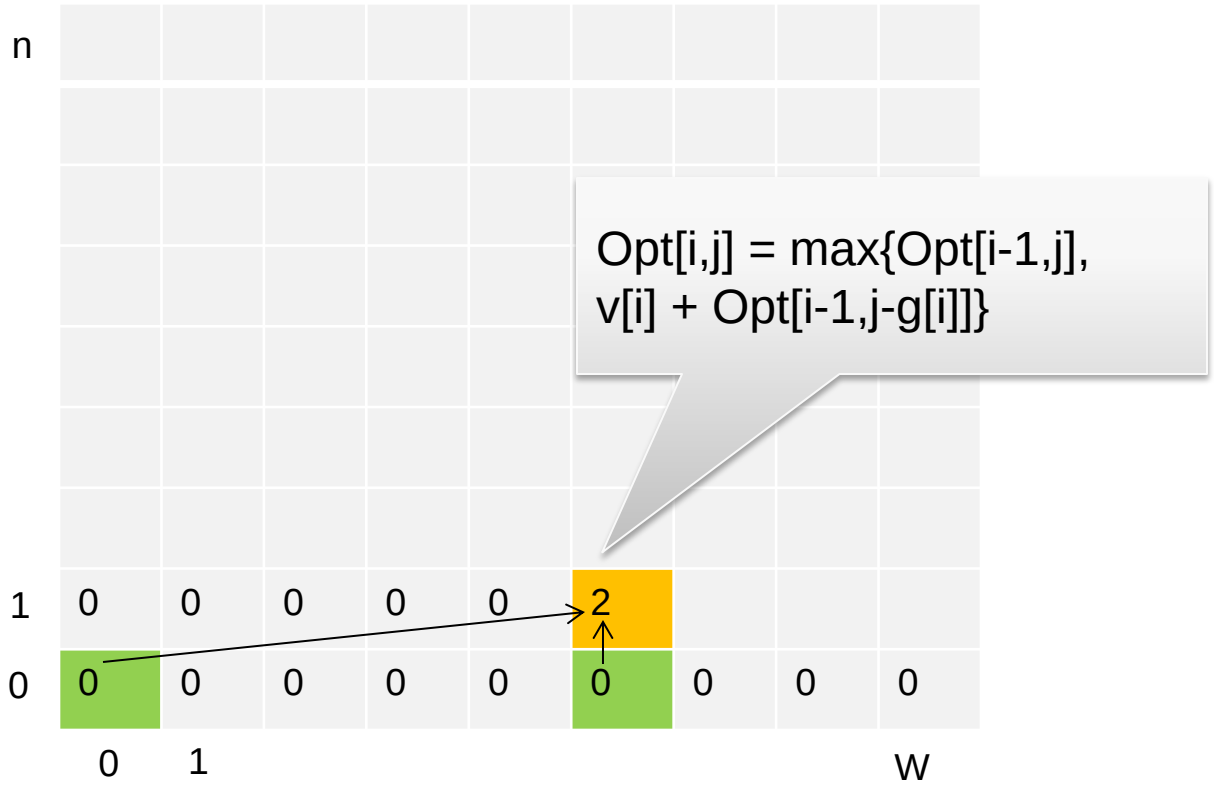
Beispiel



	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

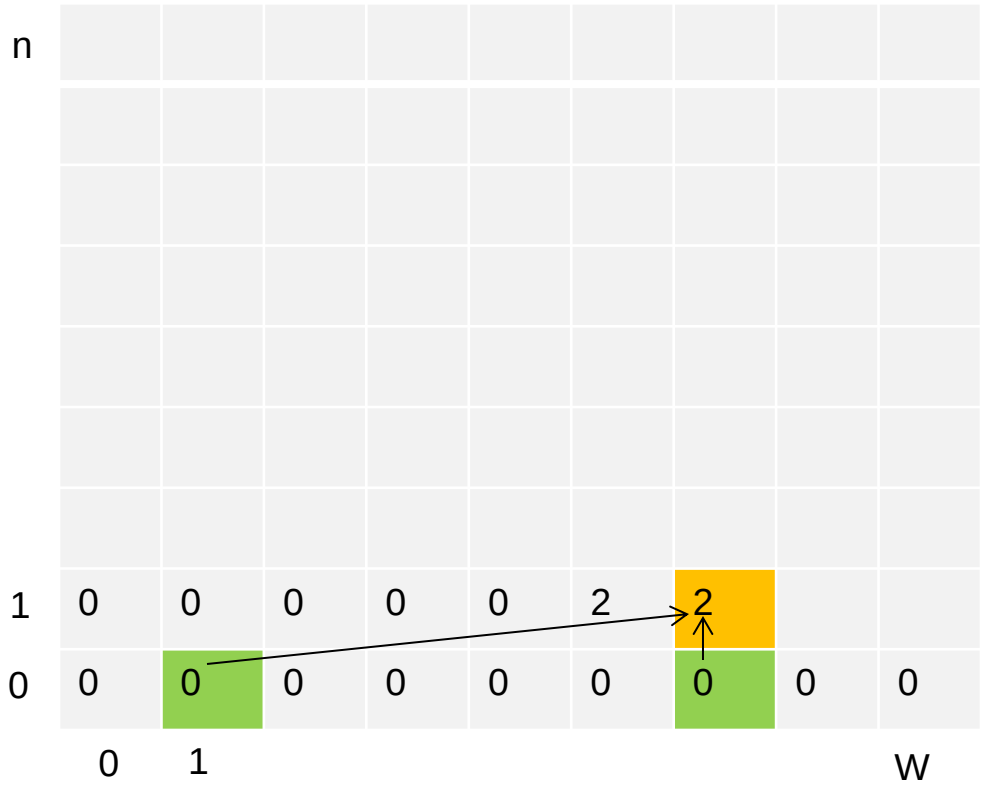


Größe Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

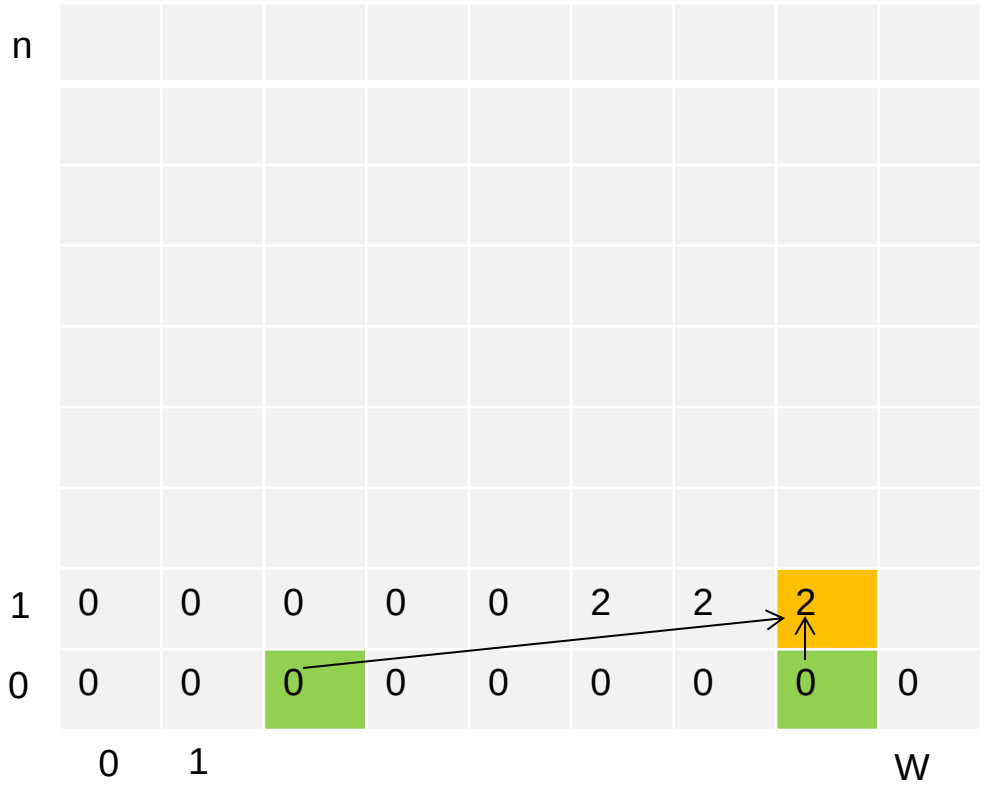


Größe Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

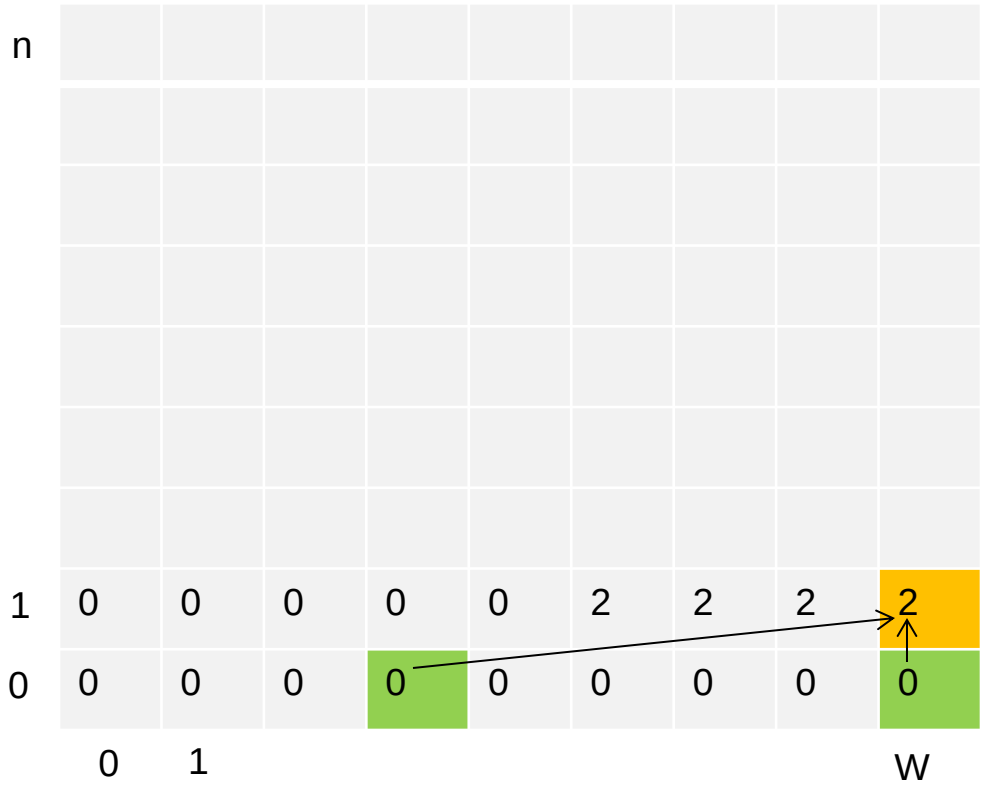
Beispiel



	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

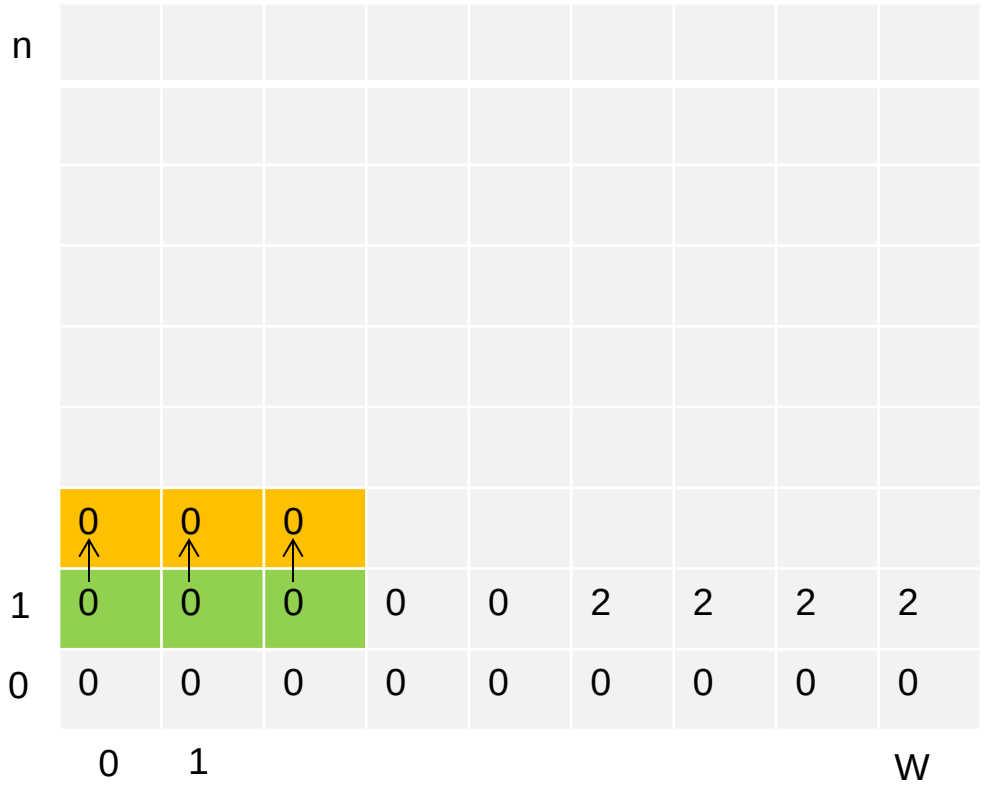


Größe Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

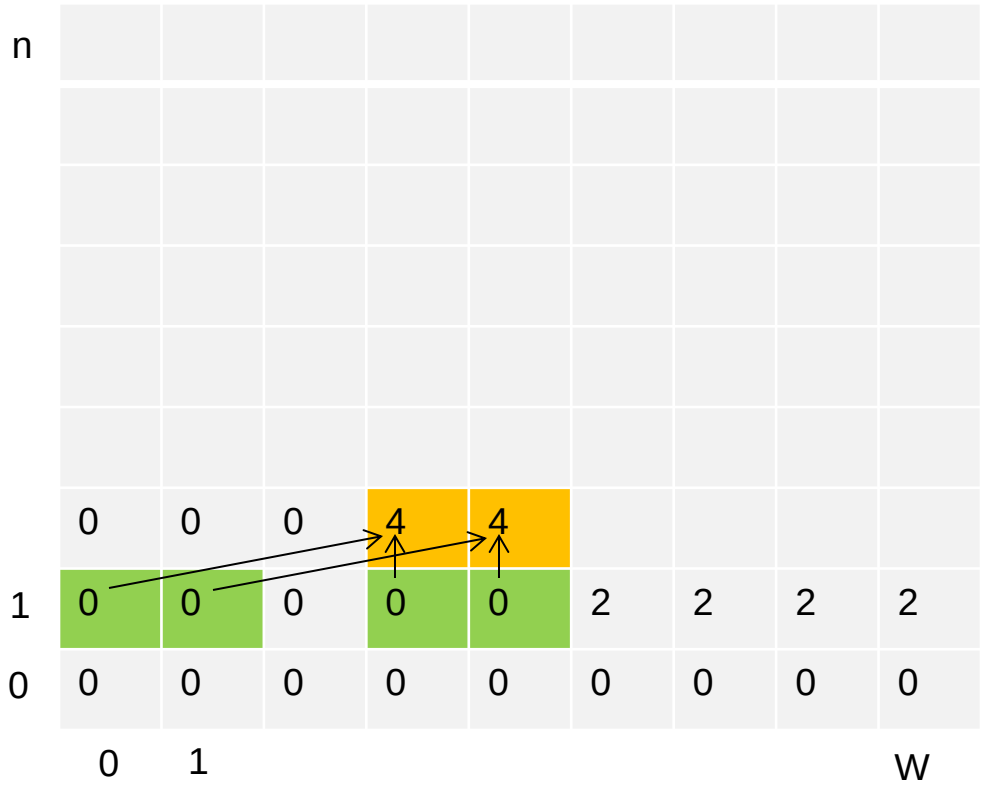
Beispiel



	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

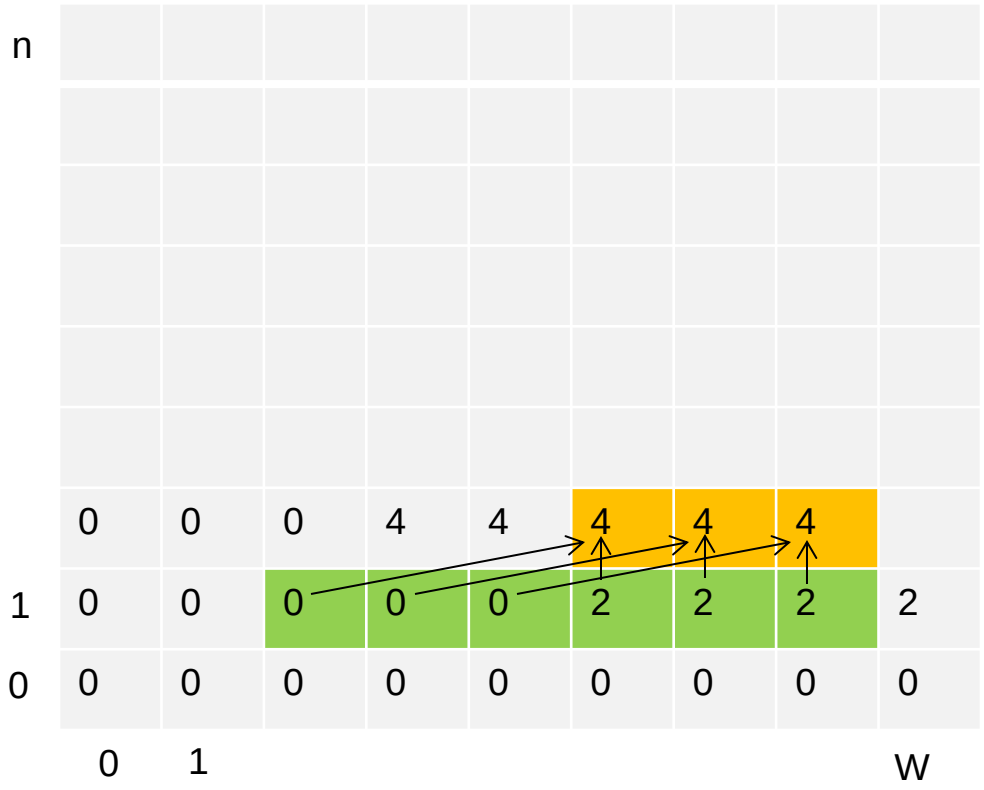
Beispiel



	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

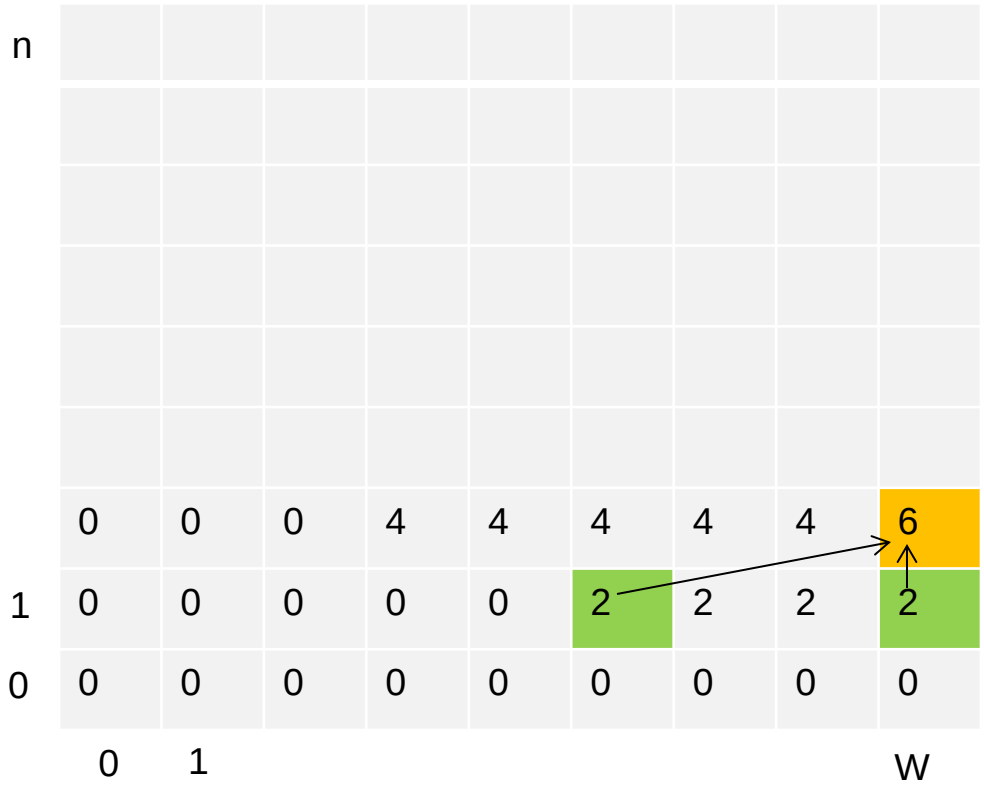
Beispiel



	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel



	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

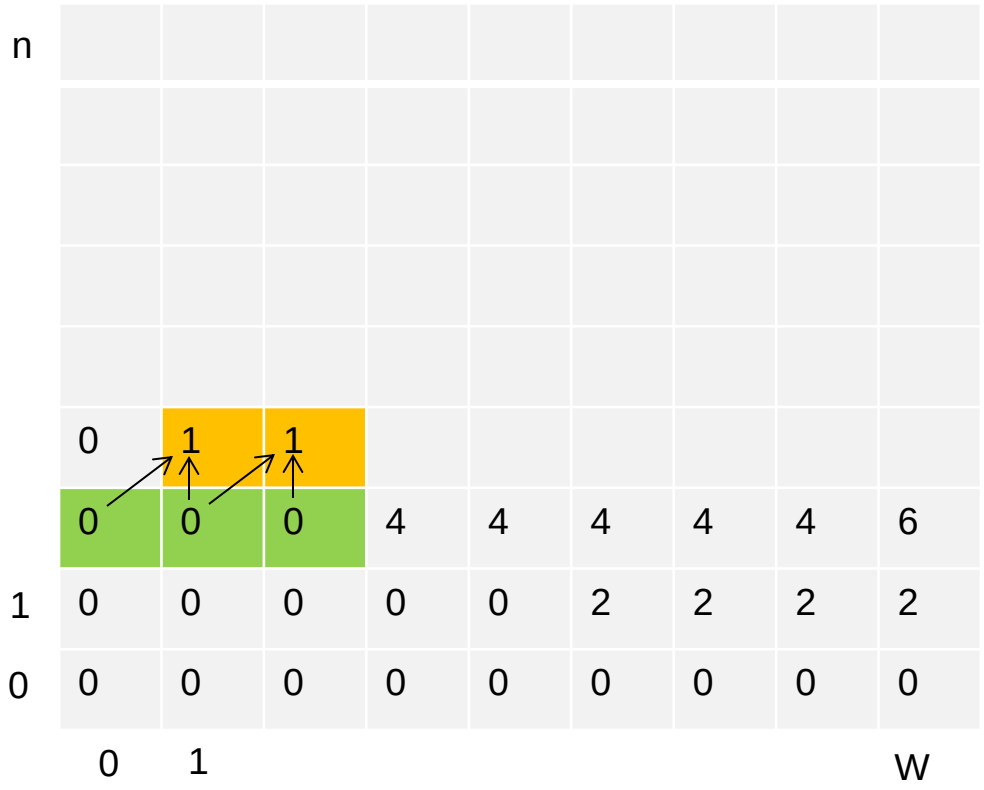
Beispiel

n									
	0								
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

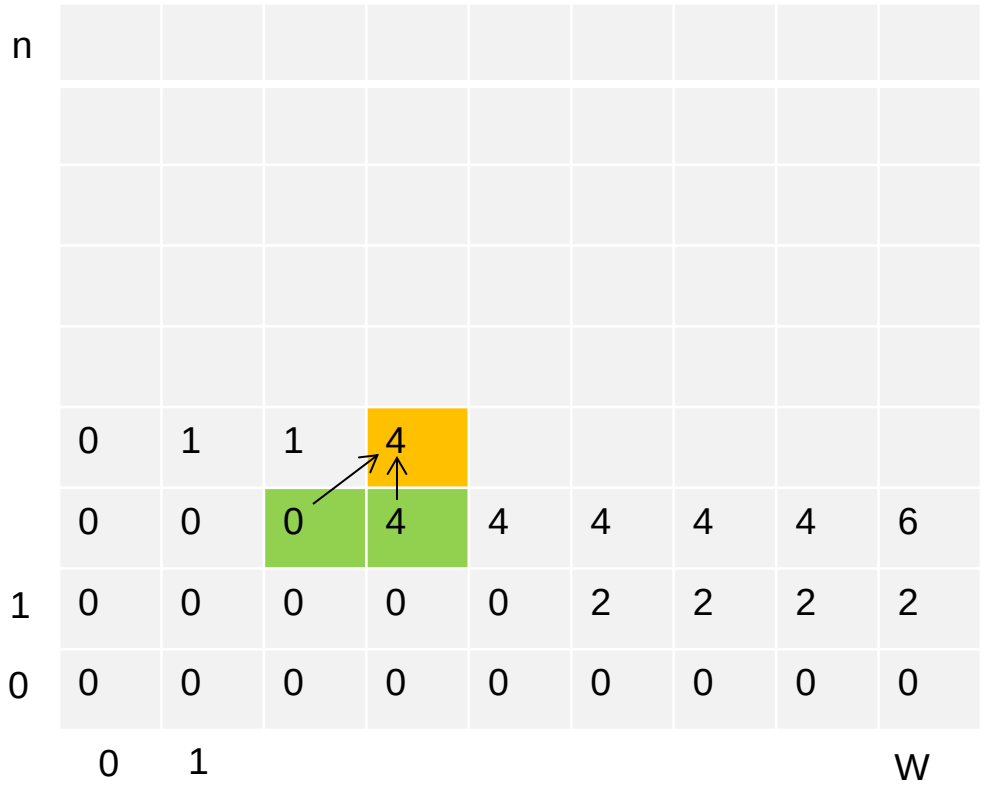


Größe Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel



Größe Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

n									
	0	1	1	4	5	5	5	5	
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe: g	Wert: v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

n									
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

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

	Größe (g)	Wert (v)
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

n									
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Größe Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

n									
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

n									
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Größe Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Größe Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beispiel

Optimaler Lösungswert für W=8

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	Größe g	Wert v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Beobachtung:

- Sei R der Wert einer optimalen Lösung für die Elemente 1,..,i
- Falls g[i]≤j und Opt[i-1,j-g[i]] +v[i]= R, so ist Objekt i in mindestens einer optimalen Lösung enthalten

Dynamische Programmierung

Wie kann man eine optimale Lösung berechnen?

- Idee: Verwende Tabelle der dynamischen Programmierung
- Fallunterscheidung + Rekursion:
  - Falls das i-te Objekt in einer optimalen Lösung für Objekte 1 bis i und Rucksackgröße j ist, so gib es aus und fahre rekursiv mit Objekt i-1 und Rucksackgröße j-g[i] fort
  - Ansonsten fahre mit Objekt i-1 und Rucksackgröße j fort

Dynamische Programmierung

RucksackLösung(Opt,g,v,i,j)

1. **if** i=0 **return** ∅
2. **else if** g[i]>j **then return** RucksackLösung(Opt,g,v,i-1,j)
3. **else if** Opt[i,j]=v[i] + Opt[i-1,j-g[i]] **then**
   return {i} ∪ RucksackLösung(Opt,g,v,i-1,j-g[i])
4. **else return** RucksackLösung(Opt,g,v,i-1,j)

Dynamische Programmierung

RucksackLösung(Opt,g,v,i,j)

1. **if** i=0 **return** ∅
2. **else if** g[i]>j **then return** RucksackLösung(Opt,g,v,i-1,j)
3. **else if** Opt[i,j]=v[i] + Opt[i-1,j-g[i]] **then**
   return {i} ∪ RucksackLösung(Opt,g,v,i-1,j-g[i])
4. **else return** RucksackLösung(Opt,g,v,i-1,j)

Aufruf

- Nach der Berechnung der Tabelle Opt von Rucksack wird RucksackLösung mit Opt, g,v, i=n und j=W aufgerufen.
- Nach dem Lemma wird dann die optimale Lösung konstruiert

Dynamische Programmierung

Opt[i,j]=13, j=8, i=8:
Es gilt Opt[i,j] > v[i]+Opt[i-1,j-g[i]]

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Opt[i,j]=13, j=8, i=7:
Es gilt Opt[i,j] = v[i]+Opt[i-1,j-g[i]]

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Opt[i,j]=6, j=4, i=6:
Es gilt g[i]>j

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Opt[i,j]=6, j=4, i=5:
Es gilt Opt[i,j] = v[i]+Opt[i-1,j-g[i]]

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Opt[i,j]=6, j=4, i=5:
Es gilt Opt[i,j] = v[i]+Opt[i-1,j-g[i]]

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

rt

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Opt[i,j]=4, j=3, i=4:
Es gilt Opt[i,j = v[i]+Opt[i-1,j-g[i]]

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Opt[i,j]=1, j=1, i=3:
Es gilt Opt[i,j] = v[i]+Opt[i-1,j-g[i]]

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

ert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Opt[i,j]=0, j=0, i=1:
Es gilt g[i]>j

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Opt[i,j]=0, j=0, i=1:
Es gilt g[i]>j

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Opt[i,j]=0, j=0, i=0:
Es gilt i=0

Beispiel

n	0	2	3	5	7	9	10	12	13
	0	2	3	5	7	9	10	12	13
	0	2	3	5	6	7	9	10	10
	0	2	3	5	6	7	9	10	10
	0	1	3	4	5	7	8	8	8
	0	1	1	4	5	5	5	5	6
	0	0	0	4	4	4	4	4	6
1	0	0	0	0	0	2	2	2	2
0	0	0	0	0	0	0	0	0	0
	0	1							W

Wert

	g	v
1	5	2
2	3	4
	1	1
	2	3
	1	2
	7	3
	4	7
n	3	3

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- Aufgrund von Korollar 26 enthält Opt[i,j] jeweils den Wert Opt(i,j) einer optimalen Lösung für Objekte {1,..,i} und Rucksackgröße j. Wir zeigen das Lemma per Induktion.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- Aufgrund von Korollar 26 enthält Opt[i,j] jeweils den Wert Opt(i,j) einer optimalen Lösung für Objekte {1,..,i} und Rucksackgröße j. Wir zeigen das Lemma per Induktion.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- Aufgrund von Korollar 26 enthält Opt[i,j] jeweils den Wert Opt(i,j) einer optimalen Lösung für Objekte {1,..,i} und Rucksackgröße j. Wir zeigen das Lemma per Induktion.
- Beweis per Induktion über i.
- (I.A.) Ist i=0, so gibt der Algorithmus die leere Menge zurück. Dies ist korrekt, da kein Objekt in den Rucksack gepackt werden kann.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- Aufgrund von Korollar 26 enthält Opt[i,j] jeweils den Wert Opt(i,j) einer optimalen Lösung für Objekte {1,..,i} und Rucksackgröße j. Wir zeigen das Lemma per Induktion.
- Beweis per Induktion über i.
- (I.A.) Ist i=0, so gibt der Algorithmus die leere Menge zurück. Dies ist korrekt, da kein Objekt in den Rucksack gepackt werden kann.
- (I.V.) Die Aussage stimmt für i-1.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- Aufgrund von Korollar 26 enthält Opt[i,j] jeweils den Wert Opt(i,j) einer optimalen Lösung für Objekte {1,..,i} und Rucksackgröße j. Wir zeigen das Lemma per Induktion.
- Beweis per Induktion über i.
- (I.A.) Ist i=0, so gibt der Algorithmus die leere Menge zurück. Dies ist korrekt, da kein Objekt in den Rucksack gepackt werden kann.
- (I.V.) Die Aussage stimmt für i-1.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- (I.S.) Ist g[i] > j, so kann Objekt i Teil keiner Lösung sein. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,v,i-1,j) zurück. Dies ist nach (I.V.) und Lemma 25 korrekt.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- (I.S.) Ist g[i] > j, so kann Objekt i Teil keiner Lösung sein. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,v,i-1,j) zurück. Dies ist nach (I.V.) und Lemma 25 korrekt.
- Ist g[i] ≤ j und Opt[i,j]=v[i] + Opt[i-1,j-g[i]], so gibt es eine optimale Lösung, die Objekt i enthält.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- (I.S.) Ist g[i] > j, so kann Objekt i Teil keiner Lösung sein. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,v,i-1,j) zurück. Dies ist nach (I.V.) und Lemma 25 korrekt.
- Ist g[i] ≤ j und Opt[i,j]=v[i] + Opt[i-1,j-g[i]], so gibt es eine optimale Lösung, die Objekt i enthält. In diesem Fall gibt der Algorithmus {i} ∪ RucksackLösung(Opt,g,v,i-1,j-g[i]) zurück. Dies ist nach (I.V.) korrekt.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- (I.S.) Ist g[i] > j, so kann Objekt i Teil keiner Lösung sein. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,v,i-1,j) zurück. Dies ist nach (I.V.) und Lemma 25 korrekt.
- Ist g[i] ≤ j und Opt[i,j]=v[i] + Opt[i-1,j-g[i]], so gibt es eine optimale Lösung, die Objekt i enthält. In diesem Fall gibt der Algorithmus {i} ∪ RucksackLösung(Opt,g,v,i-1,j-g[i]) zurück. Dies ist nach (I.V.) korrekt.
- Ist g[i] ≤ j und Opt(i,j)>v[i] + Opt[i-1,j-g[i]], so kann Objekt i nicht zu einer optimalen Lösung gehören.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- (I.S.) Ist g[i] > j, so kann Objekt i Teil keiner Lösung sein. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,v,i-1,j) zurück. Dies ist nach (I.V.) und Lemma 25 korrekt.
- Ist g[i] ≤ j und Opt[i,j]=v[i] + Opt[i-1,j-g[i]], so gibt es eine optimale Lösung, die Objekt i enthält. In diesem Fall gibt der Algorithmus {i} ∪ RucksackLösung(Opt,g,v,i-1,j-g[i]) zurück. Dies ist nach (I.V.) korrekt.
- Ist g[i] ≤ j und Opt(i,j)>v[i] + Opt[i-1,j-g[i]], so kann Objekt i nicht zu einer optimalen Lösung gehören. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,v,i-1,j) zurück. Dies ist nach (I.V.) korrekt.

Dynamische Programmierung

Lemma 27

- Hat die optimale Lösung für Objekte 1,..,i und Rucksackgröße j den Wert Opt(i,j), so berechnet Algorithmus Rucksacklösung eine Teilmenge S von {1,..,i}, so dass $\sum_{i \in S} g[i] \leq j$ und $\sum_{i \in S} v[i] = Opt(i,j)$ ist.

Beweis:

- (I.S.) Ist g[i] > j, so kann Objekt i Teil keiner Lösung sein. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,v,i-1,j) zurück. Dies ist nach (I.V.) und Lemma 25 korrekt.
- Ist g[i] ≤ j und Opt[i,j]=v[i] + Opt[i-1,j-g[i]], so gibt es eine optimale Lösung, die Objekt i enthält. In diesem Fall gibt der Algorithmus {i} ∪ RucksackLösung(Opt,g,v,i-1,j-g[i]) zurück. Dies ist nach (I.V.) korrekt.
- Ist g[i] ≤ j und Opt(i,j)>v[i] + Opt[i-1,j-g[i]], so kann Objekt i nicht zu einer optimalen Lösung gehören. Der Algorithmus gibt in diesem Fall RucksackLösung(Opt,g,v,i-1,j) zurück. Dies ist nach (I.V.) korrekt.

Dynamische Programmierung

RucksackKomplett(n,g,v,W)

1. Rucksack(n,g,v,W)
2. **return** RucksackLösung(Opt,g,v,n,W)

Dynamische Programmierung

Satz 28

Algorithmus RucksackKomplett berechnet in $\Theta(nW)$ Zeit den Wert einer optimalen Lösung, wobei n die Anzahl der Objekte ist und W die Größe des Rucksacks.

Beweis:

- Die Laufzeit wird durch Algorithmus Rucksack dominiert und ist somit $\Theta(nW)$. Die Korrektheit folgt aus den beiden Lemmas.

Datenstrukturen

Was ist eine Datenstruktur?

- Eine Datenstruktur ist eine Anordnung von Daten, die effizienten Zugriff auf die Daten ermöglicht
- Datenstrukturen für viele unterschiedliche Anfragen vorstellbar

Datenstrukturen

Ein grundlegendes Datenbank-Problem

- Speicherung von Datensätzen

Beispiel

- Kundendaten (Name, Adresse, Wohnort, Kundennummer, offene Rechnungen, offene Bestellungen,…)

Anforderungen

- Schneller Zugriff
- Einfügen neuer Datensätze
- Löschen bestehender Datensätze

Datenstrukturen

Zugriff auf Daten

- Jedes Datum (Objekt) hat einen Schlüssel
- Eingabe des Schlüssels liefert Datensatz
- Schlüssel sind vergleichbar (es gibt totale Ordnung der Schlüssel)

Beispiel

- Kundendaten (Name, Adresse, Kundennummer)
- Schlüssel: Name
- Totale Ordnung: Lexikographische Ordnung

Datenstrukturen

Zugriff auf Daten

- Jedes Datum (Objekt) hat einen Schlüssel
- Eingabe des Schlüssels liefert Datensatz
- Schlüssel sind vergleichbar (es gibt totale Ordnung der Schlüssel)

Beispiel:

- Kundendaten (Name, Adresse, Kundennummer)
- Schlüssel: Kundennummer
- Totale Ordnung: ‚≤‘

Datenstrukturen

Problem:

- Gegeben sind n Objekte O_1,.., O_n mit zugehörigen Schlüsseln $s(O_i)$

Operationen:

- Suche(x); Ausgabe O mit Schlüssel s(O) =x;
  nil, falls kein Objekt mit Schlüssel x in Datenbank
- Einfügen(O); Einfügen von Objekt O in Datenbank
- Löschen(O); Löschen von Objekt O mit aus der Datenbank

Datenstrukturen

Vereinfachung:

- Schlüssel sind natürliche Zahlen
- Eingabe nur aus Schlüsseln

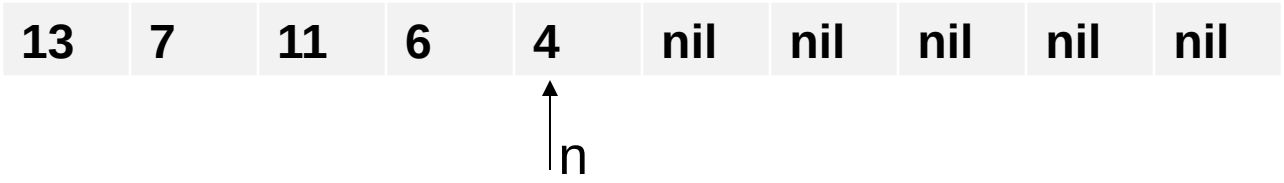
Analyse von Datenstrukturen

- Platzbedarf in Θ- bzw. O-Notation
- Laufzeit der Operationen in Θ- bzw. O-Notation

Datenstrukturen

Einfaches Feld

- Feld A[1,…,max]
- Integer n, $1 \leq n \leq max$
- n bezeichnet Anzahl Elemente in Datenstruktur

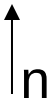


Datenstrukturen

Einfügen(s)

1. **if** n=max **then** Ausgabe „Fehler: Kein Platz in Datenstruktur“
2. **else**
3. n ← n+1
4. A[n] ← s

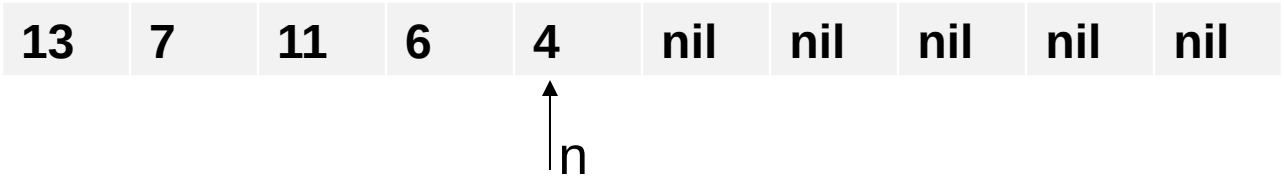
13	7	11	6	4	nil	nil	nil	nil	nil



Datenstrukturen

Einfügen(s)

1. **if** n=max **then** Ausgabe „Fehler: Kein Platz in Datenstruktur“
2. **else**
3. n ← n+1
4. A[n] ← s

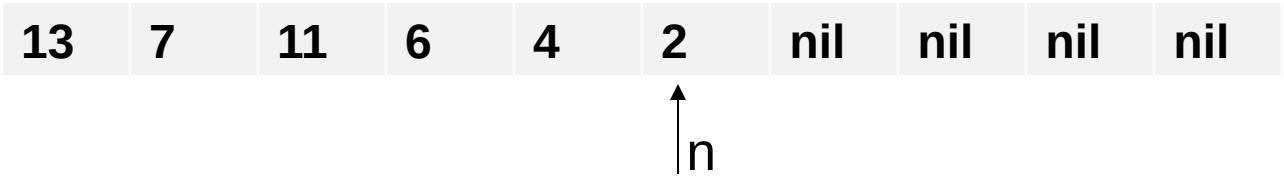


Einfügen(2)

Datenstrukturen

Einfügen(s)

1. **if** n=max **then** Ausgabe „Fehler: Kein Platz in Datenstruktur“
2. **else**
3. n ← n+1
4. A[n] ← s

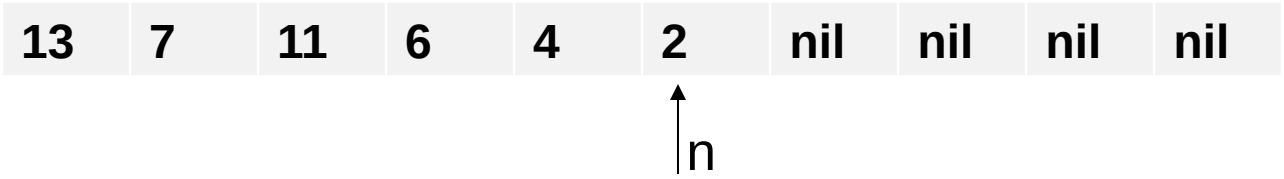


Einfügen(2)

Datenstrukturen

Suche(x)

1. **for** i ← 1 **to** n **do**
2. **if** A[i] = x **then return** i
3. **return nil**



Datenstrukturen

Löschen(i)

1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1

Annahme:
Wir bekommen Index i des zu löschenden Objekts

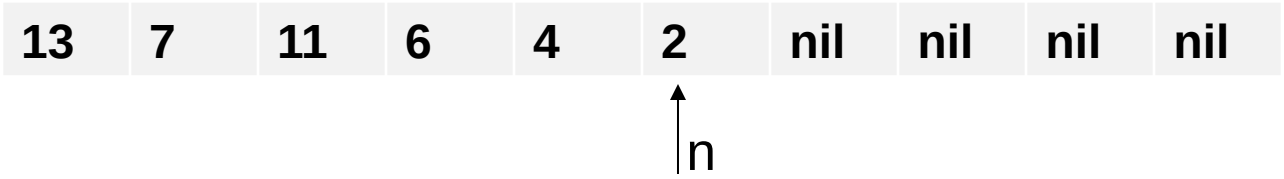
13	**7**	**11**	**6**	**4**	**2**	**nil**	**nil**	**nil**	**nil**

n

Datenstrukturen

Löschen(i)

1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1

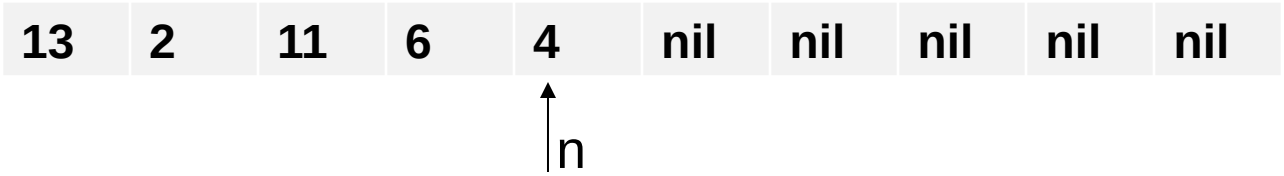


Löschen(2)

Datenstrukturen

Löschen(i)

1. A[i] ← A[n]
2. A[n] ← **nil**
3. n ← n-1



Löschen(2)

Datenstrukturen

Datenstruktur Feld

- Platzbedarf $\Theta(max)$
- Laufzeit Suche: $\Theta(n)$
- Laufzeit Einfügen/Löschen: $\Theta(1)$

Vorteile

- Schnelles Einfügen und Löschen

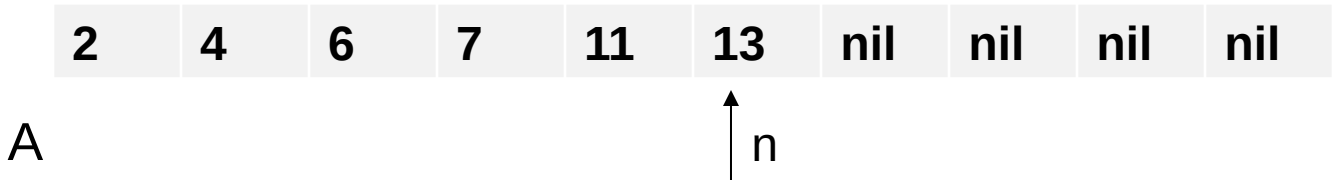
Nachteile

- Speicherbedarf abhängig von max (nicht vorhersagbar)
- Hohe Laufzeit für Suche

Datenstrukturen

Datenstruktur „sortiertes Feld“

- Sortiertes Feld A[1,…,max]
- Integer n, $1 \leq n \leq max$
- n bezeichnet Anzahl Elemente in Datenstruktur



Datenstrukturen

Einfügen(s)

1. n ← n+1
2. i ← n
3. **while** s < A[i-1] **do**
4. A[i] ← A[i-1]
5. i ← i -1
6. A[i] ← s

2	**4**	**6**	**7**	**11**	**13**	**nil**	**nil**	**nil**	**nil**

n

Einfügen(10)

Datenstrukturen

Einfügen(s)

1. n ← n+1
2. i ← n
3. **while** s < A[i-1] **do**
4. A[i] ← A[i-1]
5. i ← i -1
6. A[i] ← s

2	**4**	**6**	**7**	**11**	**13**	**nil**	**nil**	**nil**	**nil**

n

Einfügen(10)

Datenstrukturen

Einfügen(s)

1. n ← n+1
2. i ← n
3. **while** s < A[i-1] **do**
4. A[i] ← A[i-1]
5. i ← i -1
6. A[i] ← s

2	4	6	7	11	13	nil	nil	nil	nil

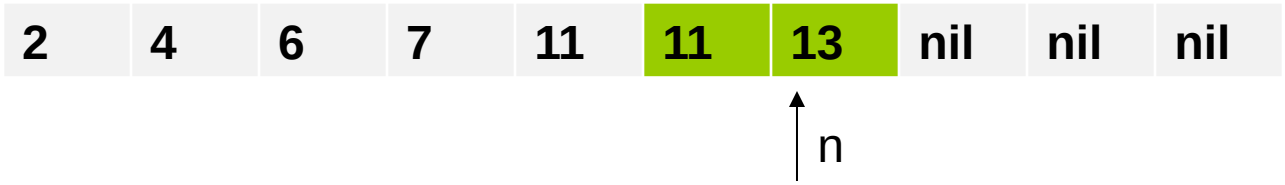
n

Einfügen(10)

Datenstrukturen

Einfügen(s)

1. n ← n+1
2. i ← n
3. **while** s < A[i-1] **do**
4. A[i] ← A[i-1]
5. i ← i -1
6. A[i] ← s



Einfügen(10)

Datenstrukturen

Einfügen(s)

1. n ← n+1
2. i ← n
3. **while** s < A[i-1] **do**
4. A[i] ← A[i-1]
5. i ← i -1
6. A[i] ← s

Laufzeit O(n)

2	4	6	7	10	11	13	nil	nil	nil

n

Einfügen(10)

Datenstrukturen

Löschen(i)

1. **for** j ← i **to** n-1 **do**
2. A[j] ← A[j+1]
3. A[n] ← nil
4. n ← n-1

Parameter ist der Index des zu löschenden Objekts

2	4	6	7	11	13	nil	nil	nil	nil

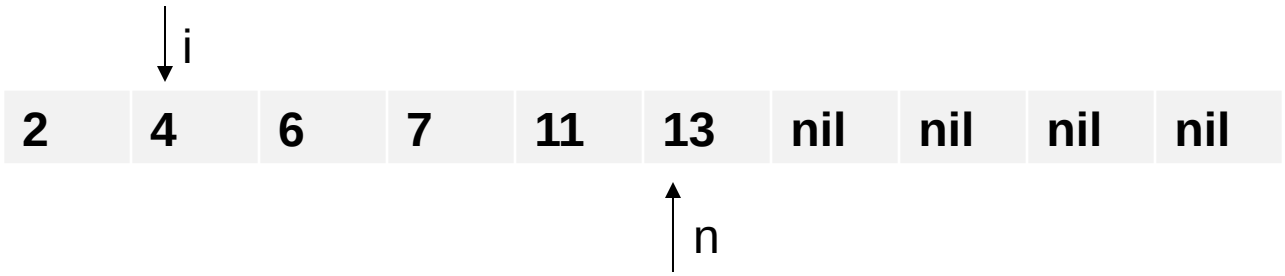
n

Datenstrukturen

Löschen(i)

1. **for** j ← i **to** n-1 **do**
2. A[j] ← A[j+1]
3. A[n] ← nil
4. n ← n-1

Parameter ist der Index des zu löschenden Objekts

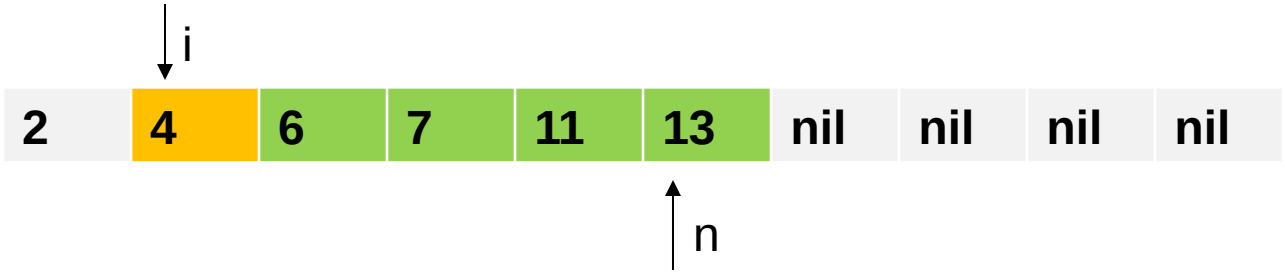


Löschen(2)

Datenstrukturen

Löschen(i)

1. **for** j ← i **to** n-1 **do**
2. A[j] ← A[j+1]
3. A[n] ← nil
4. n ← n-1

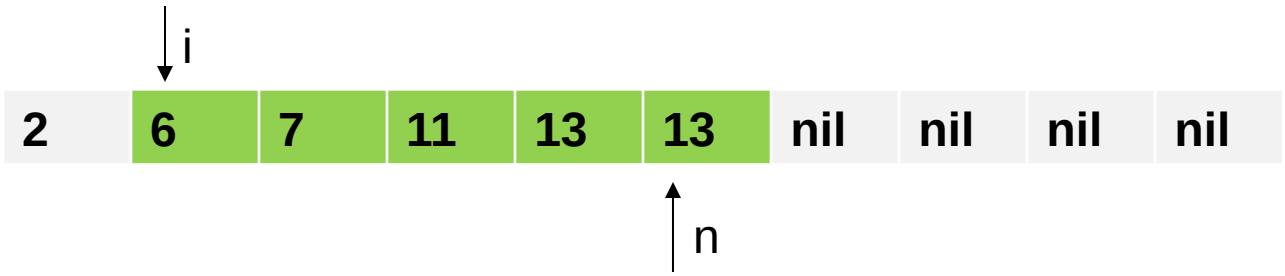


Löschen(2)

Datenstrukturen

Löschen(i)

1. **for** j ← i **to** n-1 **do**
2. A[j] ← A[j+1]
3. A[n] ← nil
4. n ← n-1

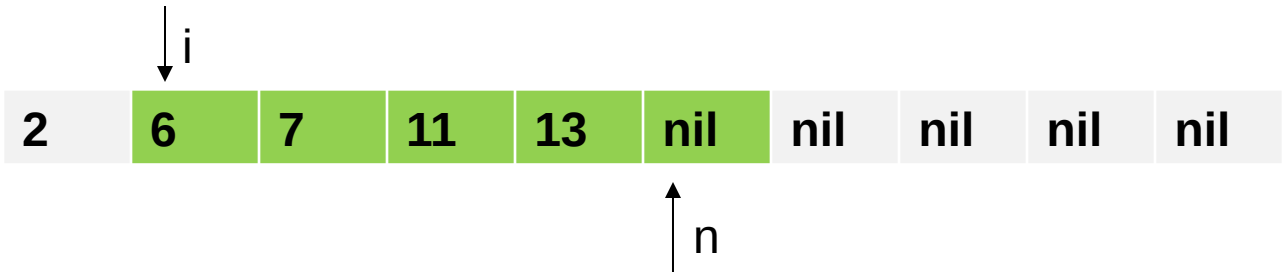


Löschen(2)

Datenstrukturen

Löschen(i)

1. **for** j ← i **to** n-1 **do**
2. A[j] ← A[j+1]
3. A[n] ← nil
4. n ← n-1

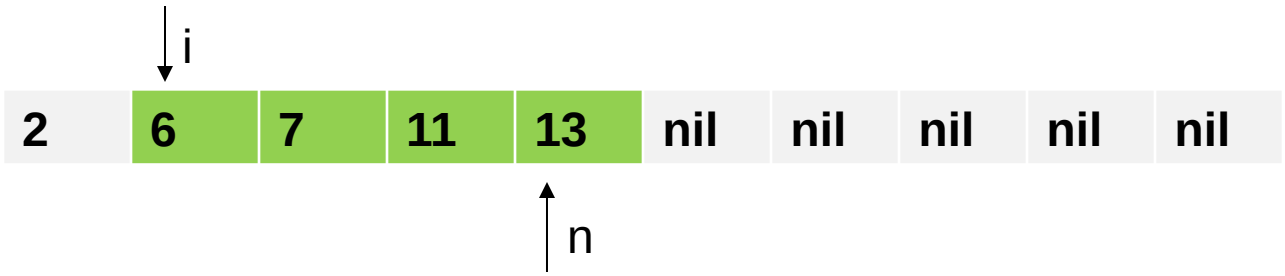


Löschen(2)

Datenstrukturen

Löschen(i)

1. **for** j ← i **to** n-1 **do**
2. A[j] ← A[j+1]
3. A[n] ← nil
4. n ← n-1

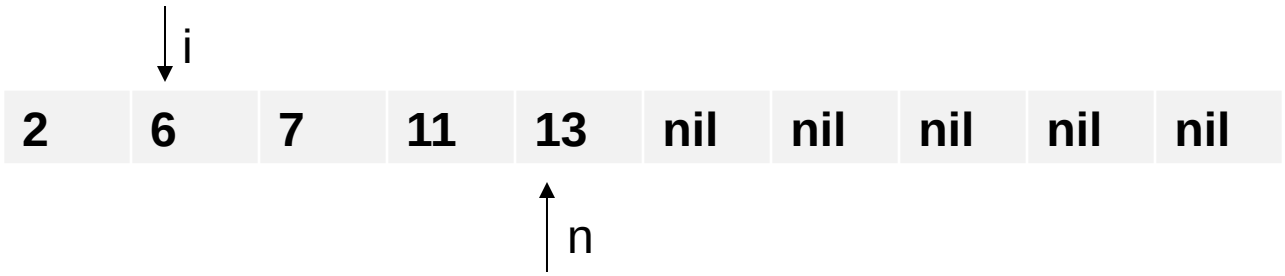


Löschen(2)

Datenstrukturen

Suchen(x)

- Binäre Suche
- Laufzeit O(log n)



Löschen(2)

Datenstrukturen

Datenstruktur sortiertes Feld

- Platzbedarf $\Theta(max)$
- Laufzeit Suche: $\Theta(\log n)$
- Laufzeit Einfügen/Löschen: $\Theta(n)$

Vorteile

- Schnelles Suchen

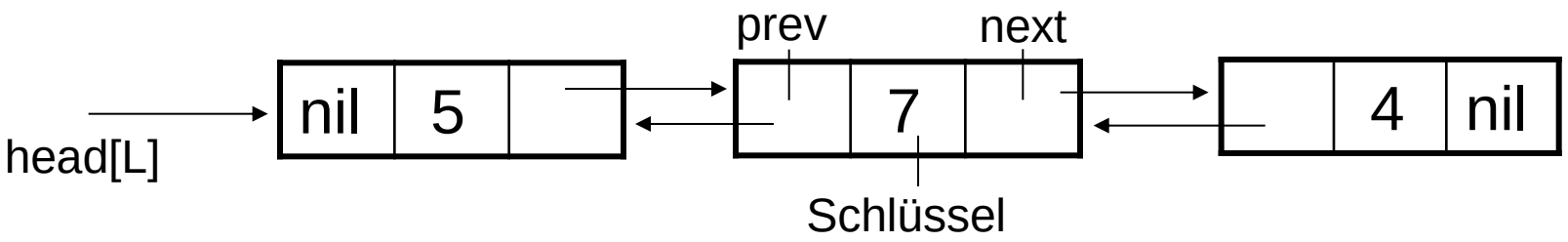
Nachteile

- Speicherbedarf abhängig von max (nicht vorhersagbar)
- Hohe Laufzeit für Einfügen/Löschen

Datenstrukturen

Doppelt verkettete Listen

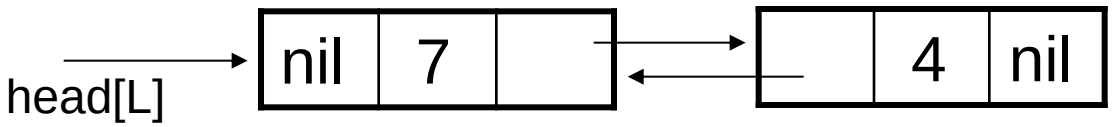
- Listenelement x ist Objekt bestehend aus Schlüssel und zwei Zeigern prev und next
- next verweist auf Nachfolger von x
- prev verweist auf Vorgänger von x
- prev/next sind **nil**, wenn Vorgänger/Nachfolger nicht existiert
- head[L] zeigt auf das erste Element



Datenstrukturen

Einfügen(L,x)

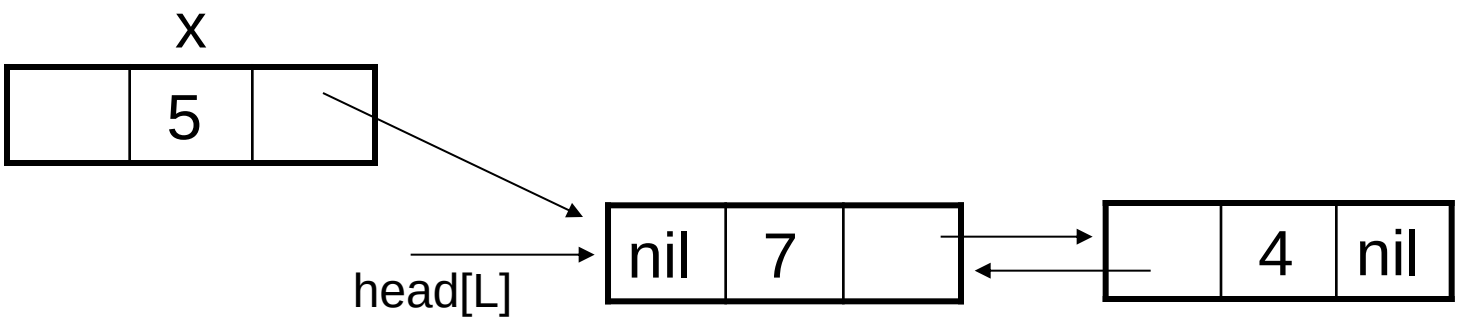
1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**



Datenstrukturen

Einfügen(L,x)

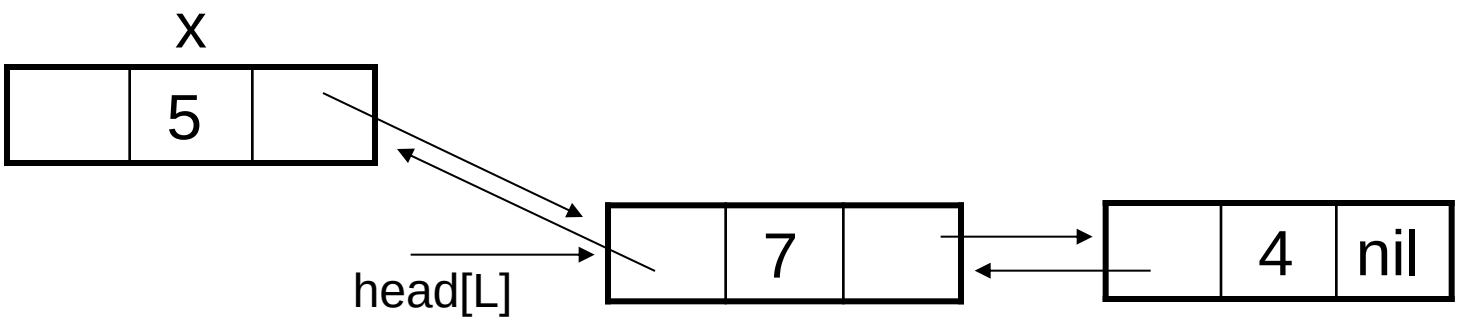
1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**



Datenstrukturen

Einfügen(L,x)

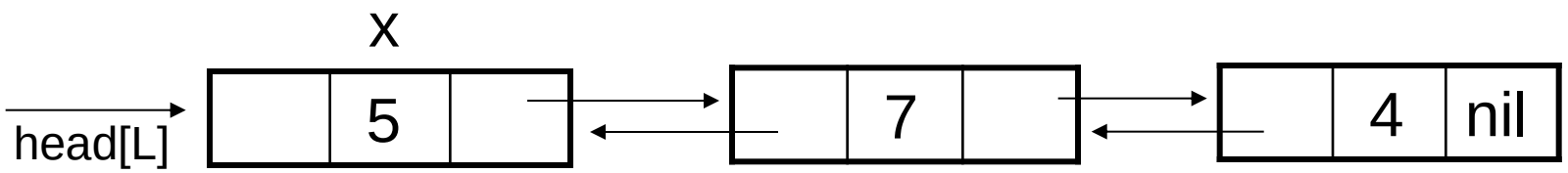
1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**



Datenstrukturen

Einfügen(L,x)

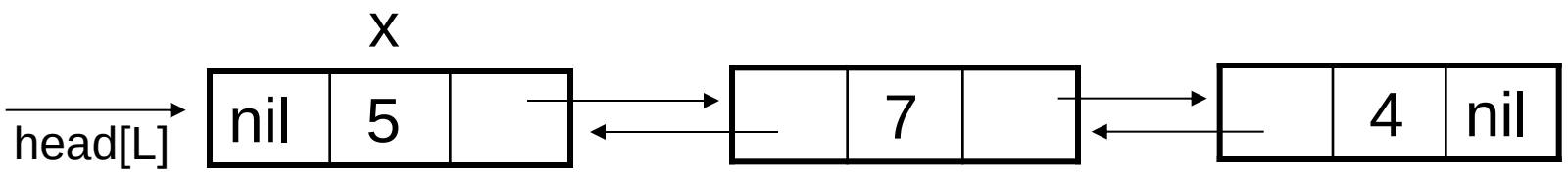
1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**



Datenstrukturen

Einfügen(L,x)

1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**



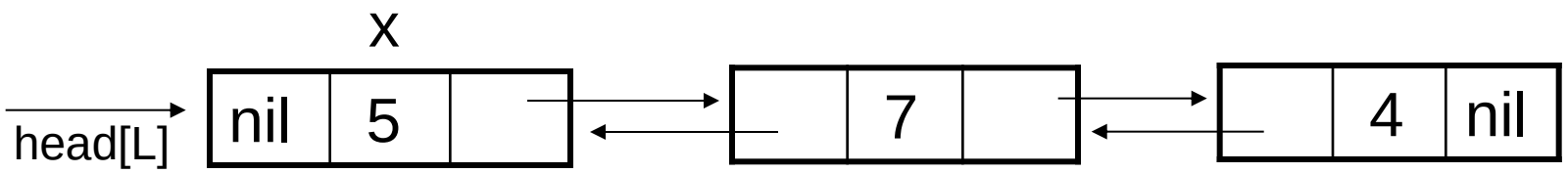
Datenstrukturen

Einfügen(L,x)

1. next[x] ← head[L]
2. **if** head[L] ≠ **nil then** prev[head[L]] ← x
3. head[L] ← x
4. prev[x] ← **nil**

Laufzeit

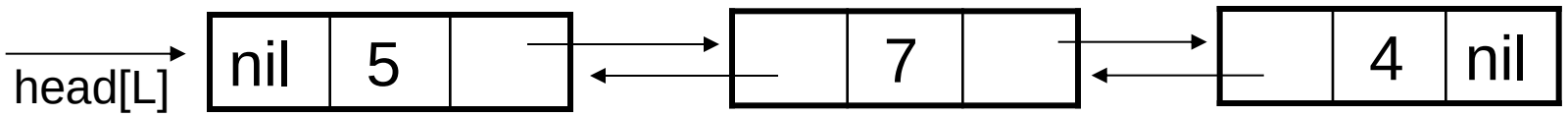
- O(1)



Datenstrukturen

Löschen(L,x)

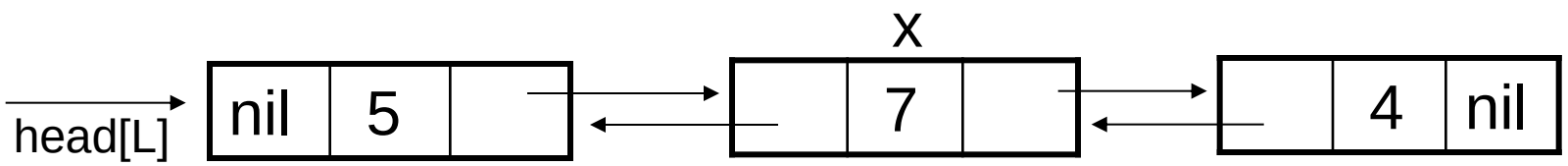
1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]
4. **delete** x



Datenstrukturen

Löschen(L,x)

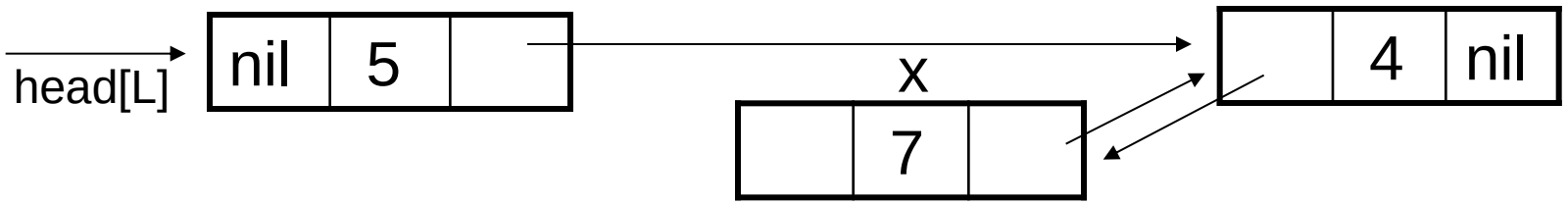
1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]
4. **delete** x



Datenstrukturen

Löschen(L,x)

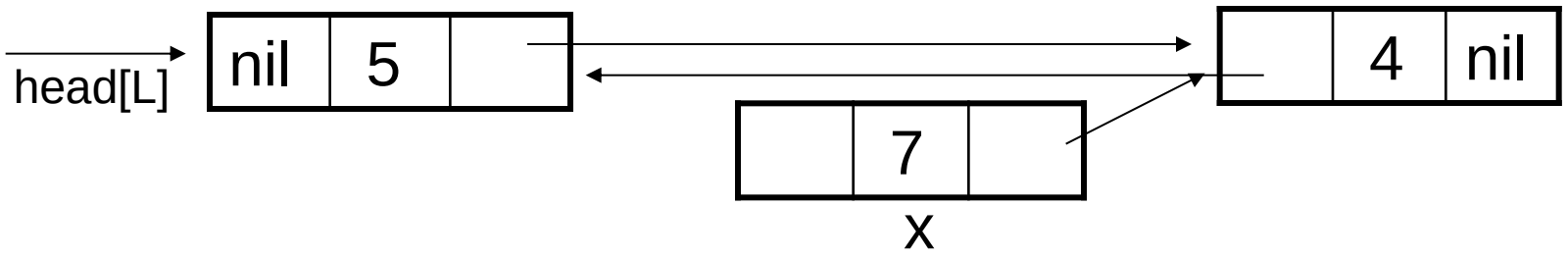
1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]
4. **delete** x



Datenstrukturen

Löschen(L,x)

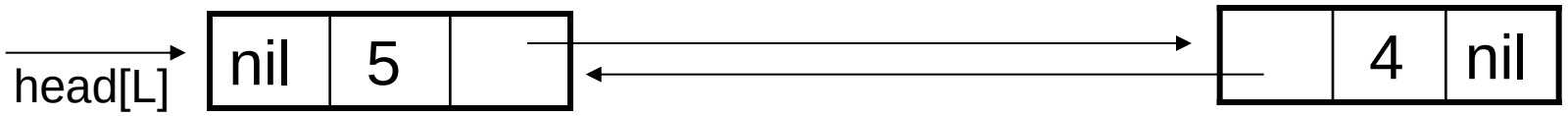
1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]
4. **delete** x



Datenstrukturen

Löschen(L,x)

1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]
4. **delete** x



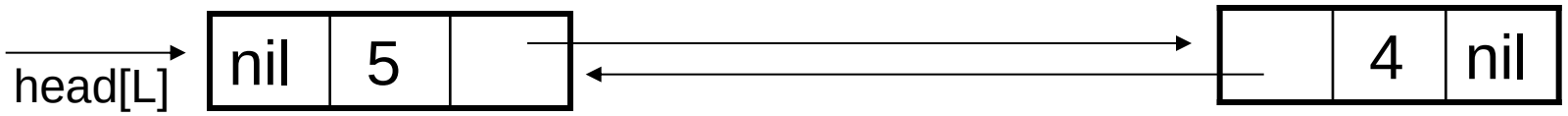
Datenstrukturen

Löschen(L,x)

1. **if** prev[x] ≠ **nil then** next[prev[x]] ← next[x]
2. **else** head[L] ← next[x]
3. **if** next[x] ≠ **nil then** prev[next[x]] ← prev[x]
4. **delete** x

Laufzeit

- O(1)



Datenstrukturen

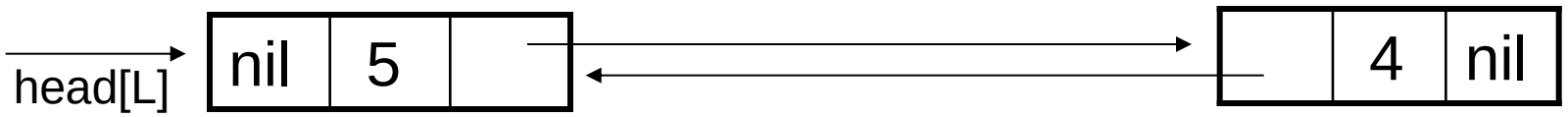
Suche(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

Datenstrukturen

Suche(L,k)

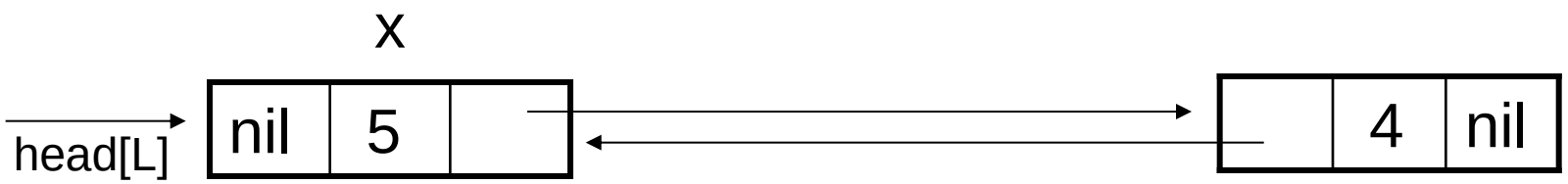
1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**



Datenstrukturen

Suche(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

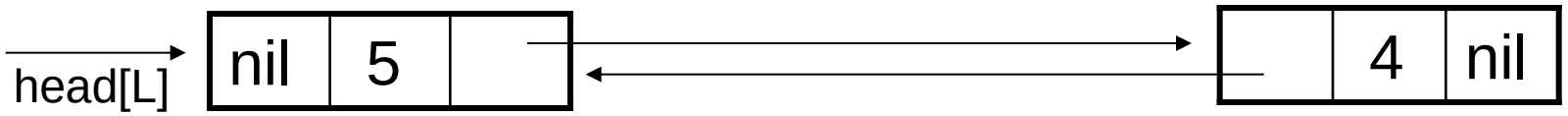


Suche(L,4)

Datenstrukturen

Suche(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

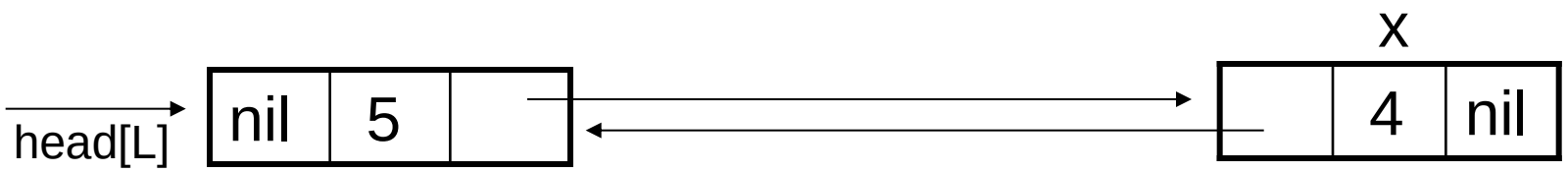


Suche(L,4)

Datenstrukturen

Suche(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

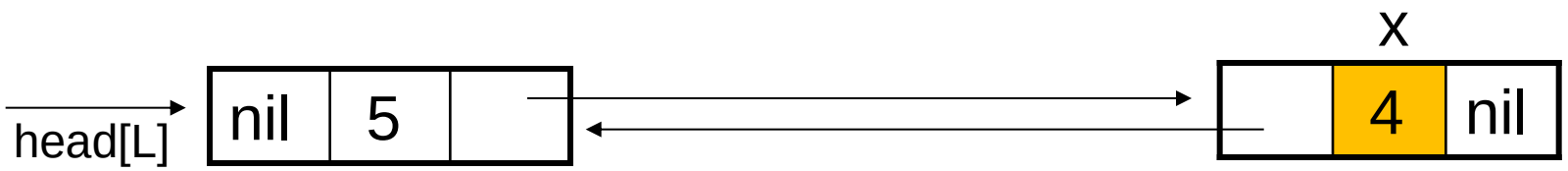


Suche(L,4)

Datenstrukturen

Suche(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**



Suche(L,4)

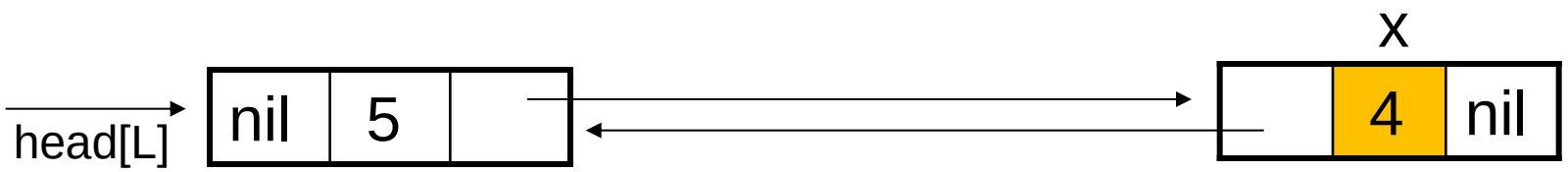
Datenstrukturen

Suche(L,k)

1. x ← head[L]
2. **while** x≠nil and key[x]≠k **do**
3. x ← next[x]
4. **return x**

Laufzeit

- O(n)



Suche(L,4)

Datenstrukturen

Datenstruktur Liste:

- Platzbedarf $\Theta(n)$
- Laufzeit Suche: $\Theta(n)$
- Laufzeit Einfügen/Löschen: $\Theta(1)$

Vorteile

- Schnelles Einfügen/Löschen
- O(n) Speicherbedarf

Nachteile

- Hohe Laufzeit für Suche