Dynamic programming

Rekursion ist oft die Ursache für manche der langsamsten Kategorien von Big-O, wie z.B. $O(2^N)$.

-> Geschwindigkeitsfallen in rekursivem Code identifizieren

-> Techniken finden, um sie zu fixen

Geschwindigkeitsfalle: Unnötige rekursive Aufrufe

Diese rekursive Funktion findet die größte Zahl in einem Array (Teilproblem, Verarbeitung beginnt von rechts):

```
int max (int array[], int size)
{
        // Abbruchkriterium: wenn das Array nur 1 Element hat,
        // ist das die größte Zahl:
        if  (1 == size)  return array[0];

       //   Das aktuelle Element mit dem größten des Rests des Arrays vergleichen.
       //    Ist dieses Element größer, geben wir es als größte Zahl zurück.
        if (array[0] > max (array+1, size-1)) return array[0];

        // Sonst geben wir die größte Zahl des restlichen Arrays zurück.
        else  return max (array+1, size - 1);
}
```

Geschwindigkeitsfalle: Unnötige rekursive Aufrufe

```
int max (int array[], int size)
{
    // Abbruchkriterium: wenn das Array nur 1 Element hat,
    // ist das die größte Zahl:
    if  (1 == size)  return array[0];

    // Das aktuelle Element mit dem größten des Rests des Arrays vergleichen.
    // Ist dieses Element größer, geben wir es als größte Zahl zurück.
    if (array[0] > max (array+1, size-1)) return array[0];

    // Sonst geben wir die größte Zahl des restlichen Arrays zurück.
    else  return max (array+1, size - 1);
}
```

Wir erreichen den Vergleich mit einem bedingten Befehl.

Die Prüfung der Bedingung enthält diesen Test:

```
if (array[0] > max (array+1, size-1)) return array[0];
```

Die zweite Hälfte des bedingten Befehls ist:

```
else  return max (array+1, size - 1);
```

Dieser Code funktioniert, enthält aber eine verborgene Ineffizienz:

Er enthält die Phrase max (array+1, size - 1) u.U. zweimal: bei jedem Test, und in jedem else-Zweig.

Jedesmal, wenn wir max (array+1, size - 1) aufrufen, triggern wir eine ganze Lawine rekursiver Aufrufe...

Geschwindigkeitsfalle: Unnötige rekursive Aufrufe

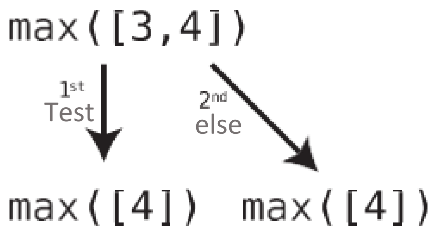
Beispielarray: [1, 2, 3, 4] (Zahl ist immer kleiner als die größte Zahl des Rests)

Start: Vergleich der 1 mit der größten Zahl des restlichen Array [2, 3, 4].

Vergleich der 2 mit dem Rest [3, 4].

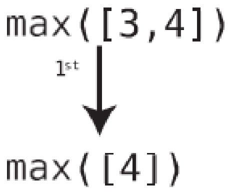
Vergleich der 3 mit der [4]. Das triggert einen weiteren rekursiven Aufruf der [4] selbst, die Abbruchkriterium ist.

Der Weg hinunter in den Call stack:

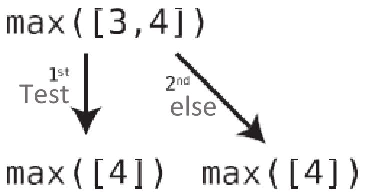


Geschwindigkeitsfalle: Unnötige rekursive Aufrufe

Wenn max() im Abbruchkriterium für [4] aufgerufen wird, ist es ziemlich einfach—ein einzelner Funktionsaufruf..
Wenn wir max() für das Array [3, 4] aufrufen, vergleichen wir in der 1. Hälfte die 3 mit dem Rest von [4]:



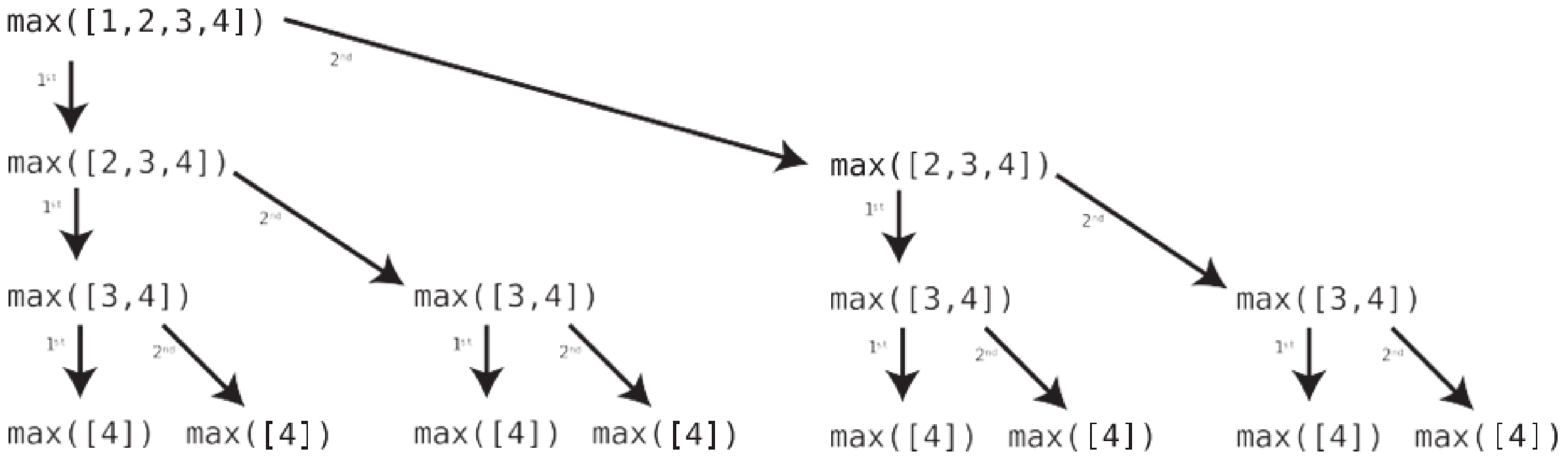
Da die 3 nicht größer als 4 ist, trigger wir die 2. Hälfte des Befehls, die max([4]) zurückgibt —
aber max([4]) triggert den rekursiven Aufruf nochmal:



-> Wir rufen max([4]) zweimal auf.
Wir haben das Ergebnis von max([4]) schon einmal berechnet,
warum sollten wir es für das gleiche Ergebnis nochmals aufrufen??

Geschwindigkeitsfalle: Unnötige rekursive Aufrufe

Wenn wir den Callstack weiter nach oben gehen, verschlimmert sich das Problem noch:



Komplexität: $O(2^n-1)$ für n=4: 15 Aufrufe, für n=10: 1023 Aufrufe!

Der kleine Fix für Big-O

Wir sollten max() in jedem Durchlauf nur 1x aufrufen, und das Ergebnis in einer Variable speichern:

```
int max1 (int array[], int size)
{
        // Abbruchkriterium: wenn das Array nur 1 Element hat,
        // ist das die größte Zahl:
        if  (1 == size)  return array[0];

        // Berechne das Maximum des Rests des Arrays und speichere ihn
        // in eine Variable:
        int max_of_remainder = max1 (array+1, size-1);

        // Vergleich des ersten Elements mit der Variable:
        if (array[0] > max_of_remainder)   return array[0];
        else  return max_of_remainder;
}
```

Wir rufen max1() in Summe nur noch 4x auf!
Wichtiger Unterschied: In jedem Durchlauf wird die Funktion nur noch einmal aufgerufen!

Der kleine Fix für Big-O

Die verbesserte Variante max1() ruft sich nur noch so oft auf, wie das Array Elemente hat.

Sie hat dadurch eine Effizienz von O(N).

-> Sehr kleine Codeänderung führt zur Geschwindigkeitserhöhung von $O(2^N)$ zu O(N).

-> Zusätzliche, unnötige rekursive Calls müssen unbedingt vermieden warden!

Rekursionskategorie: Überlappende Teilprobleme

Eine Fibonacci–Folge ist eine unendliche mathematische Zahlensequenz:

jede Zahl ist die Summe der vorangegangenen beiden Zahlen der Sequenz: fib(n): fib(n-1) + fib(n-2).

n: 0, 1, 2, 3, 4, 5, 6, ...

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...

Die Funktion fib() gibt die n.te Zahl eine Fibonacci-Folge zurück.

Fibonacci (10): 55 (die 10. Zahl in der Folge, die 0 gilt als 0-te Zahl der Folge).

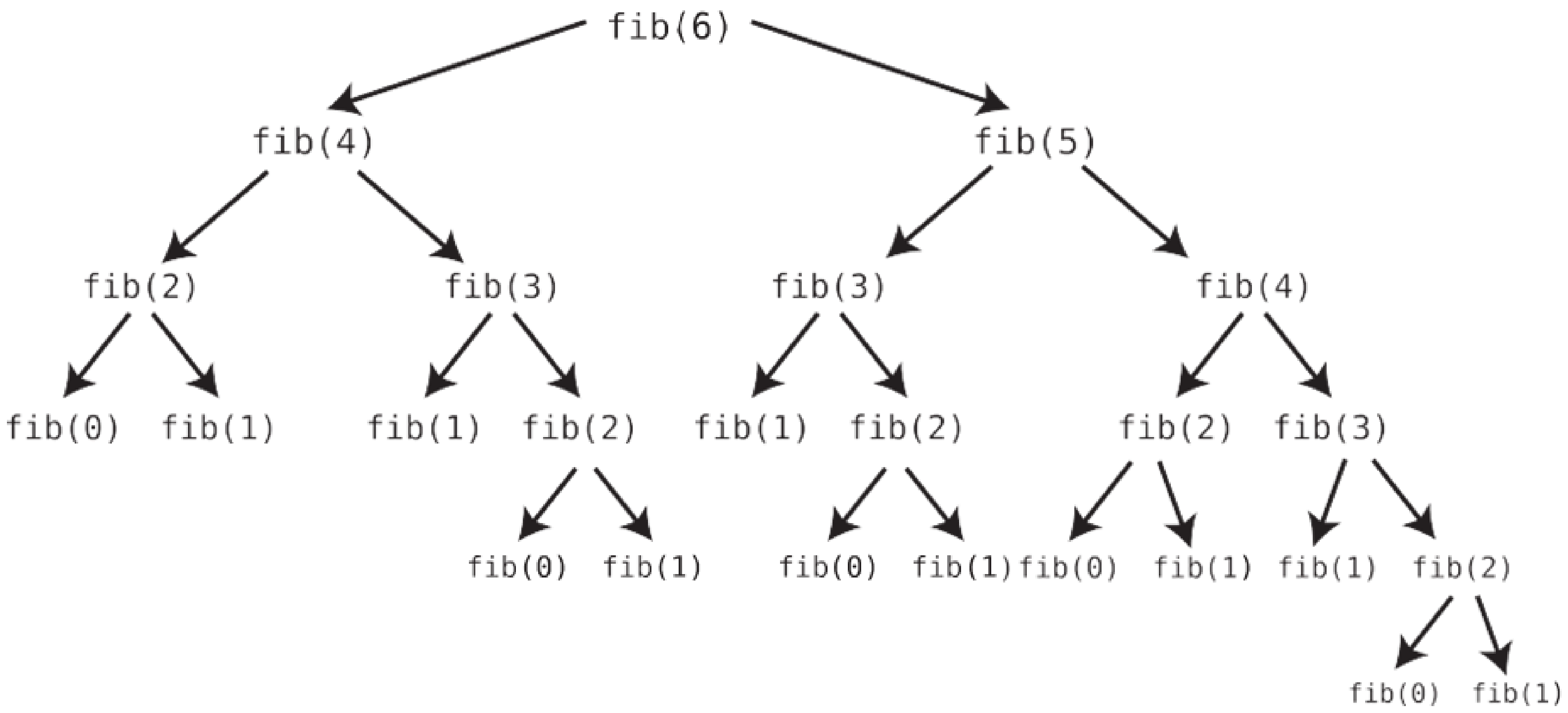
```
int fib (int n)
{
    // Abbruchkriterium sind die ersten beiden Zahlen in der Serie:
    if ((0 == n) ||  (1 == n))     return n;

    // Gibt die Summe  der vorigen beiden Fibonacci-Zahlen zurück:
    return (fib(n - 2) + fib(n - 1));
}
```

Diese Funktion ruft sich selbst jedoch immer zweimal auf!*!!*

Überlappende Teilprobleme

Wir geben die “6” in die Funktion hinein: O (2^n)



Überlappende Teilprobleme

Wir müssen sowohl fib(n - 2) und fib(n - 1) aufrufen!

Die obigen Teilprobleme überlappen sich, da fib(n-2) und fib(n-1) alle Funktionen inklusive und unterhalb von fib(n-2) doppelt aufrufen.

Lösung: **Dynamische Programmierung**

Dynamische Programmierung nennen wir die Optimierung rekursiver Probleme, die überlappende Teilprobleme haben.

Dynamische Programmierung durch Memoisation

a) **Dynamische Programmierung durch Memoisation (top-down)**

Memoisation (Abspeicherung) ist eine Technik, um Computerprogramme zu beschleunigen, indem Rückgabewerte von Funktionen zwischengespeichert anstatt mehrfach neu berechnet werden

Fibonacci-Beispiel: erster Aufruf von fib(3): führt Berechnung durch und gibt die Zahl “2” zurück.
Sie könnten vor der Weiterarbeit dieses Ergebnis in eine Hashtabelle speichern.

Hashtabelle: {3: 2} – das Ergebnis von fib(3) ist die Zahl 2.

Ergebnisse aller neuen, erstmaligen Berechnungen werden abgespeichert.

Hashtabelle nach der Durchführung von fib(4), fib(5) und fib(6):

```
{
    3: 2,
    4: 3,
    5: 5,
    6: 8
}
```

Dynamische Programmierung durch Memoisation

fib(4) ruft nicht mehr fib(3) und fib(2) auf, sondern schlägt zunächst den Key 4 in der Hashtabelle nach

- enthalten?: Wert wird sofort zurückgegeben (bereits berechnet)
- noch nicht in der Hashtabelle? Berechnung von fib(4) starten!

Überlappende Teilprobleme führen die gleichen rekursiven Berechnungen wieder und wieder aus.

Durch Memoisation machen wir keine Berechnung mehr, die vorher schon durchgeführt wurde.

Wie bekommt eine rekursive Funktion Zugriff auf diese Hashtabelle?

-> Deren Adresse wird als zusätzlicher Parameter in die Funktion übergeben.

Dynamische Programmierung durch Memoisation

Beim Funktionsaufruf übergeben wir jetzt die Zahl und eine “leere” Hashtabelle:

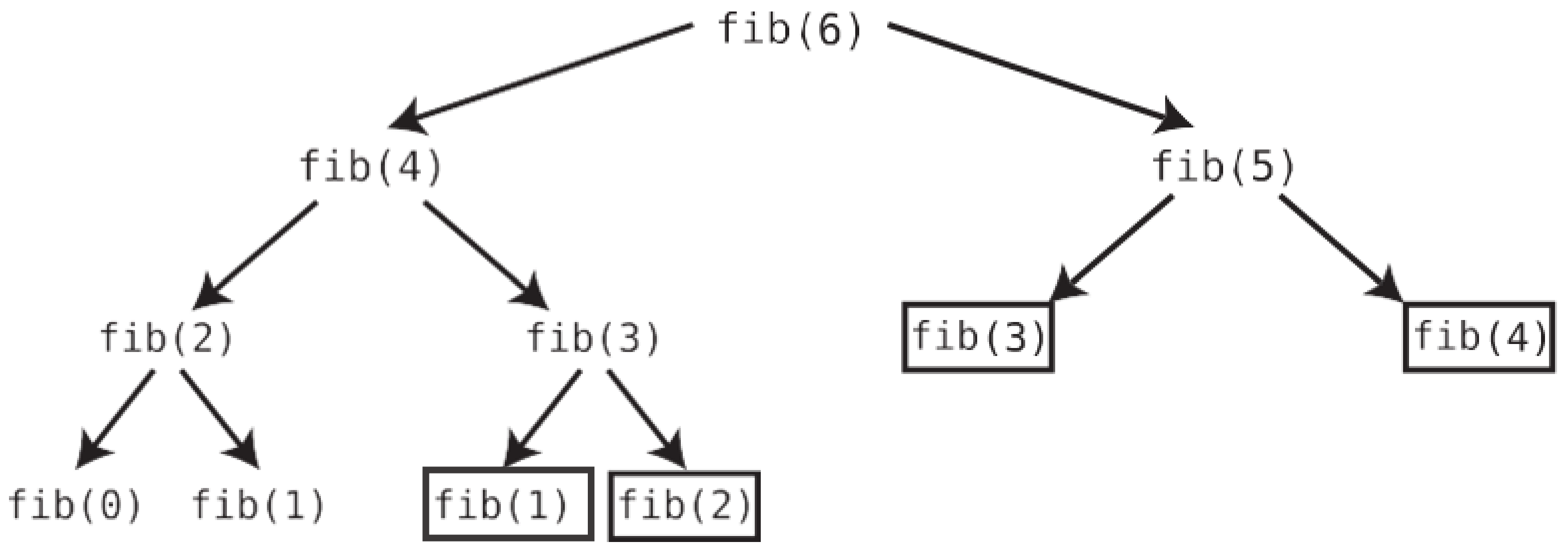
```
int fib1 (int n, struct hashElement *memo)
{
    if ((0 == n) || (1 == n))   return n;

    // Prüfen, ob die (memo genannte) Hashtabelle bereits einen Wert für den
    // Key n enthält (ob (fib(n) zuvor bereits berechnet wurde oder nicht):
    // Ist n nicht in memo enthalten, berechnen wir fib(n) mit Rekursion und
    // speichern das Ergebnis in die Hashtabelle:
    if (0 == memo[n].key)
    {
        memo[n].key=n;
        memo[n].value = fib1(n - 2, memo) + fib1(n - 1, memo);
    }


    // Jetzt ist der Wert von  fib(n) sicher in memo (entweder weil er vorher
    // schon drinstand,oder weil wir ihn gerade reingeschieben haben.
    // Wir können ihn zurückgeben:
    return memo[n].value;
}
```

Dynamische Programmierung durch Memoisation

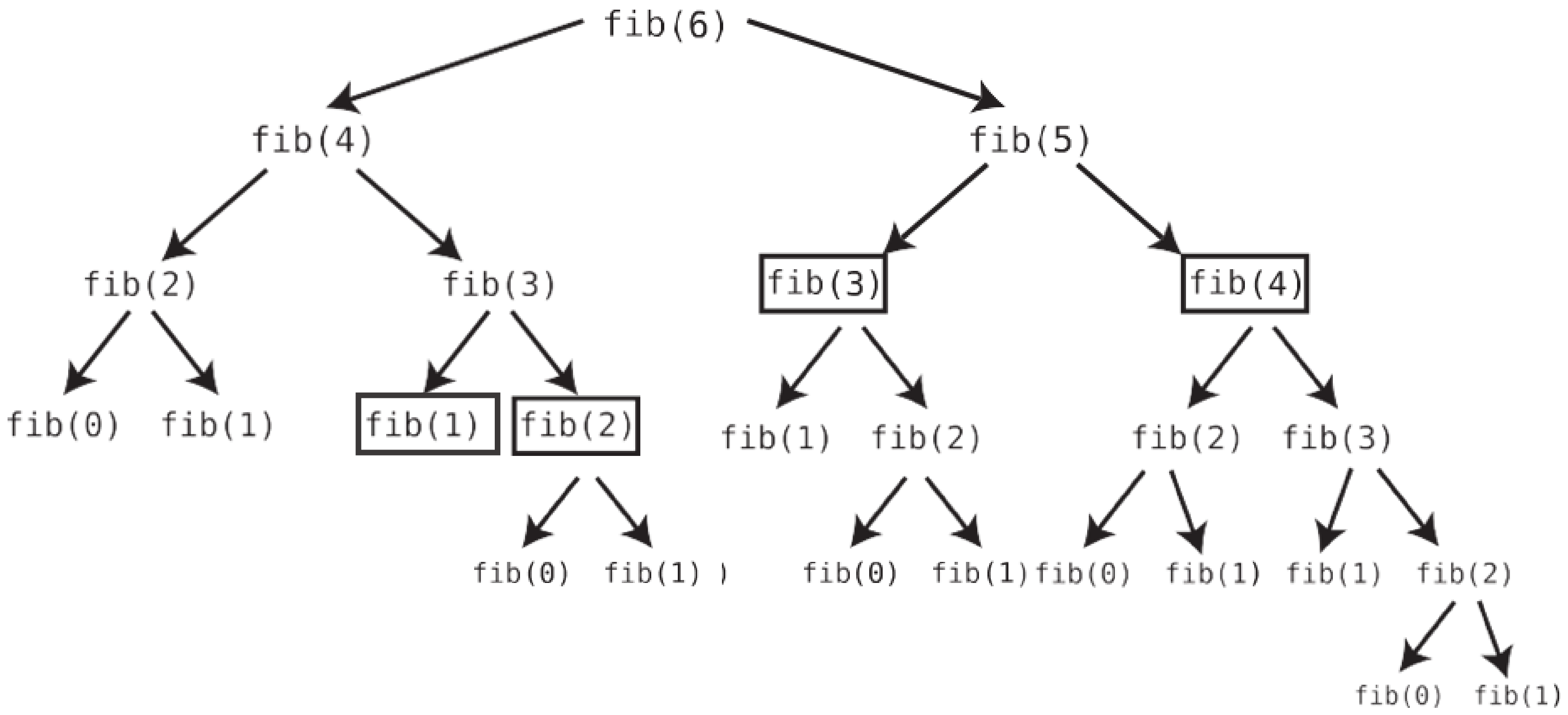
Die Übersicht der rekursiven Aufrufe sieht für unsere Version mit Memoisation jetzt so aus:



Jeder eingerahmte Aufruf findet das Ergebnis in der Hashtabelle!

Dynamische Programmierung ohne Memoisation

Zum Vergleich:



Dynamische Programmierung durch Memoisation

Was ist jetzt das Big O unserer Funktion?

N	**Recursive calls**
1	1
2	3
3	5
4	7
5	9
6	11

Wir machen für N nur (2N) - 1 Aufrufe: ein O(N)-Algorithmus! Enorme Beschleunigung gegenüber O(2^N)!

Dynamische Programmierung durch Bottom-up

Rekursiver Ansatz mit Memoisation:

- Berechnung von fib(n) als Summe der beiden vorherigen Fibonacci-Zahlen.
- startete mit der höchsten Zahl, und berechnete diese auf Basis der niedrigeren Zahlen (top-down).

b) Dynamische Programmierung durch Bottom-up (Iteration)

Statt dem rekursiven Ansatz:

- bottom-up-Ansatz verwendbar -> Rekursion völlig vermeidbar
- überlappende Teilprobleme werden damit überwunden

Dynamische Programmierung durch Bottom-up

Bottom-up-Ansatz mit einer normalen Schleife:

```
int fibLoop (int n)
{
    if (0 == n) return 0;
    // a und b starten mit den ersten beiden Zahlen der Serie, also:
    int a = 0, b = 1;

    // Schleife von 1 bis n:
    for (int i=0;  i<=n-2;  i++)
    // a und b werden zu den nächsten Zahlen der Serie.
    // b wird die Summe aus b + a, und a wird das bisherige b.
    {
        b=b+a;
        a=b-a;  // b(alt)=b(neu) -a
    }
    return b;
}
```

Memoisation (Top-down) vs. Bottom-up

Ist ein Verfahren besser als das andere?   Hängt vom Problem ab und davon, warum Sie Rekursion benutzen möchten:

Bietet die Rekursion eine elegante und verständliche Lösung für das Problem?

-> Memoisation nutzen, um die überlappenden Teilprobleme zu eliminieren
-> Allerdings: Memoisation erfordert die Nutzung einer Speicherplatz konsumierenden Hashtabelle.

Bottom-up-Ansatz existiert, der funktioniert und für Andere verständlich ist?

-> diesen verwenden

Zusammenfassung

Jetzt können Sie effizienten rekursiven Code schreiben.

Übung

Die folgende Funktion berechnet rekursiv the n-te Zahl einer mathematischen Sequenz, die “Golomb-Folge”[1] genannt wird. Sie ist furchtbar ineffizient! Benutzen Sie Memoisation, um sie zu optimieren:

```
int golomb (int n)
{
        if (1 == n) return 1;
        return (1 + golomb (n - golomb (golomb (n - 1))));
}
```

[1] Die *Golomb-Folge* ist eine sich selbst erzeugende Folge ganzer Zahlen, bei der die an n-ter Stelle aufsteigend stehende natürliche Zahl a_n angibt, wie oft n in der aufsteigend erzeugten Folge vorkommt.

Index n: 1 2 3 4 5 6 7 8 9 …
Zahl: 1 2 2 3 3 4 4 4 5 5 5 6 6 6 6 7 7 7 7 8 8 8 8 9 9 9 9 9

Lösung

Übung: 15 min.

Die folgende Funktion erhält ein Array von Zahlen und gibt deren Summe zurück, solange keine der Zahlen die Summe auf über 100 erhöht.
Sobald eine der Zahlen die Summe auf über 100 erhöhen würde, wird sie ignoriert.
Die Funktion macht jedoch unnötige rekursive Aufrufe. Fixen sie den Code, um unnötige Rekursionen zu vermeiden:

```
int add_until_100 (int array[], int size)
{
    if (0 == size) return 0;
    if (100 <= (array[0] + add_until_100 (&array[1], size- 1)))
            return add_until_100 (&array[1], size - 1);
    else    return (array[0] + add_until_100(&array[1], size- 1));
}
```

Lösung

Wir haben hier 2 rekursive Aufrufe der Funktion selbst:

```
int add_until_100 (int array[], int size)
{
    if (0 == size) return 0;
    if (100 <= (array[0] + add_until_100 (&array[1], size- 1)))
            return add_until_100 (&array[1], size - 1);
    else    return (array[0] + add_until_100(&array[1], size- 1));
}
```

Wir können diese auf einen Aufruf reduzieren:

Rekursive Algorithmen für mehr Geschwindigkeit

Wir wollen im Folgenden verstehen, warum Rekursion auch der Schlüssel zu Algorithmen sein kann, die viel, viel schneller laufen:

Bereits behandelte Sortieralgorithmen: **Bubble Sort, Selection Sort**, und **Insertion Sort**.

Im realen Leben werden diese (für die Lehre gut geeigneten) Algorithmen nicht wirklich für die Arraysortierung verwendet.

Die meisten Programmiersprachen haben on-board Sortierfunktionen für Arrays (Eigenimplementierung unnötig)

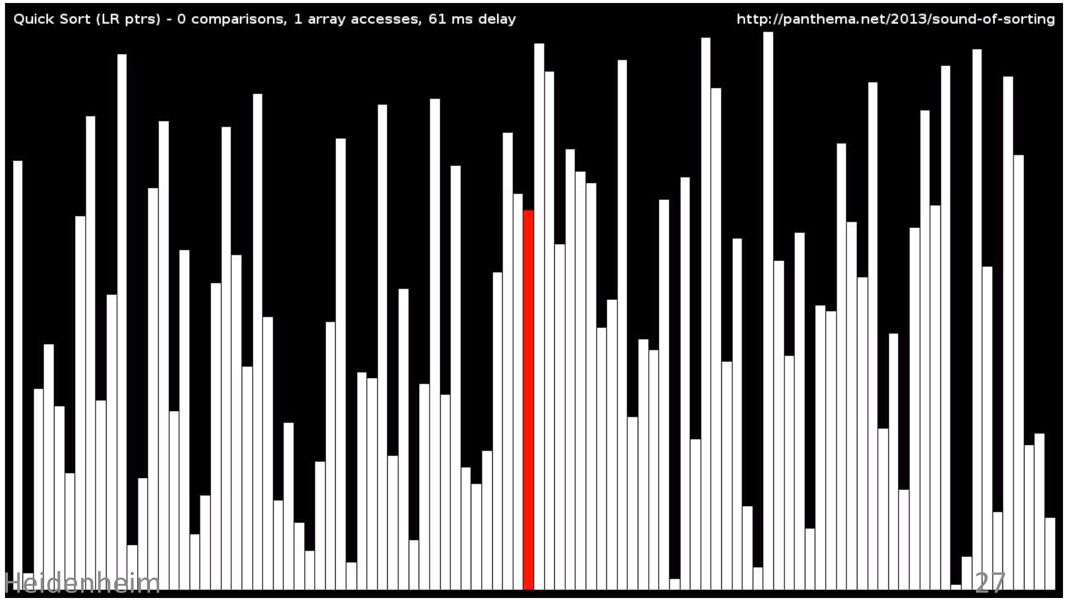
Der in vielen dieser Sprachen unter der Haube laufende Algorithmus: ***Quicksort***

QuickSort: Sortieren durch Zusammensetzen

Idee: Nimm die (erste)Karte aus dem Stapel. Durchlaufe die restlichen Karten und teile sie auf in alle mit einem Wert kleiner oder gleich dem der ersten Karte (Stapel 1) und mit Wert größer als dem der ersten Karte (Stapel 2).

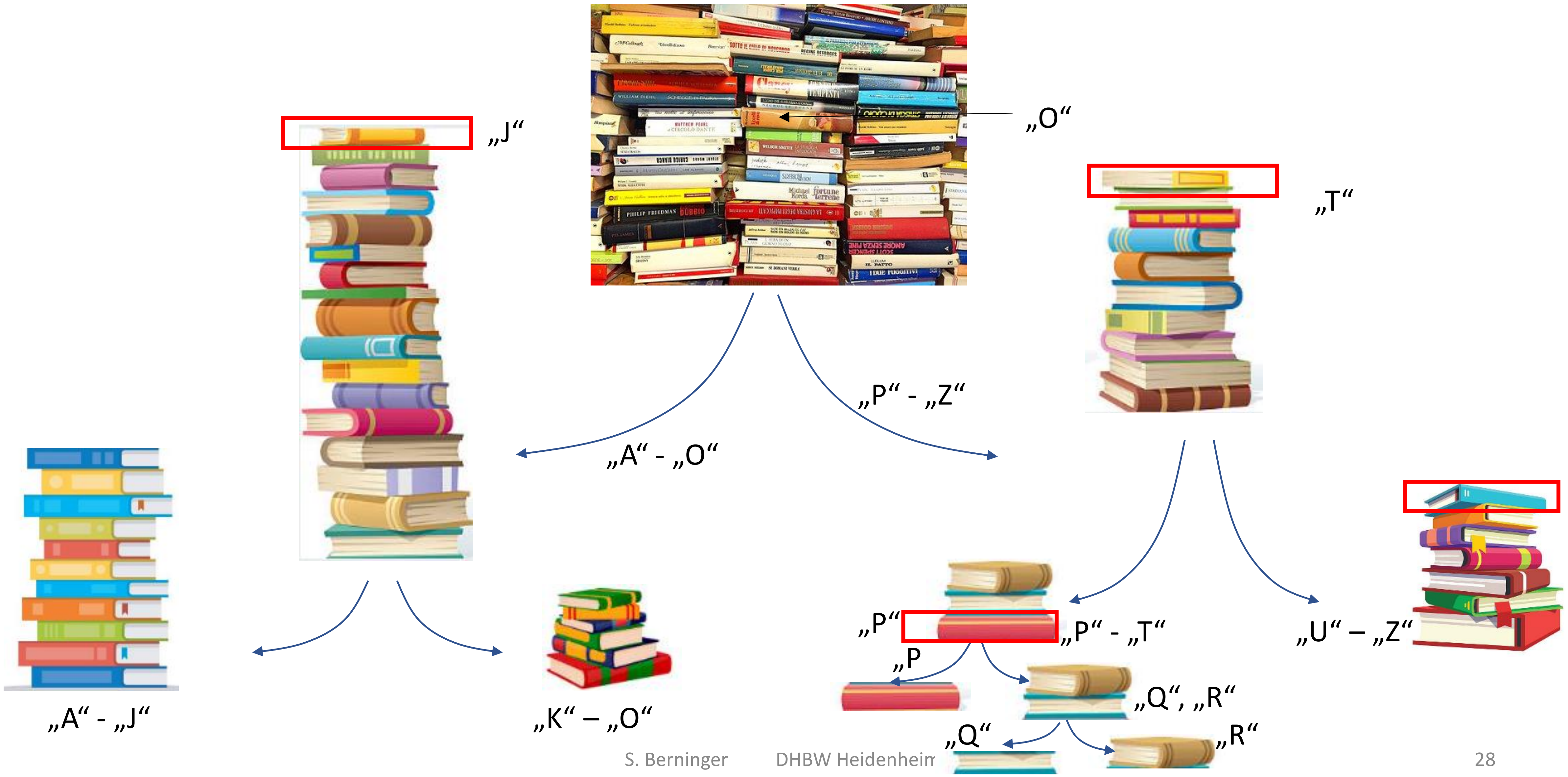
Gib die beiden so entstandenen Teilstapel, wenn sie überhaupt Karten enthalten, an je einen Helfer mit der Bitte, auch nach dem hier beschriebenen Verfahren vorzugehen.

Warte, bis Dir beide sortierte Teile zurückgegeben wurden, dann lege zuunterst den sortierten Stapel 1, darauf die anfangs gezogene Karte, darauf den sortierten Stapel 2 und gib das Ganze als sortiert zurück.



http://panthema.net/2013/sound-of-sorting

QuickSort: Sortieren durch Zusammensetzen



Quicksort

Quicksort:

- ist ein extrem schneller Sortieralgorithmus, besonders effizient für Average-Szenarien
- performed in worst-case Szenarien (invers sortierten Arrays) gleich zu Insertion und Selection sort
- ist sehr viel schneller für Average-Szenarien – die am häufigsten auftreten!
- stützt sich auf ein ***Partitionierung*** genanntes Konzept ab: Einführung siehe nächste Folien

Quicksort: Partitionierung im Array

Partitionierung eines Arrays:

- einen zufälligen Wert des Arrays nehmen – wir nennen ihn Pivot(-element)
- sicherstellen, dass jede Zahl des Arrays, die kleiner als das Pivotelement ist, links davon einsortiert wird, und jede Zahl, die größer ist als das Pivotelement, rechts davon einsortiert wird

Beispiel:

0	5	2	1	6	3

Der Konsistenz halber wählen wir hier stets das ganz rechte Element als Pivot (könnten technisch jedes nehmen)

0	5	2	1	6	3

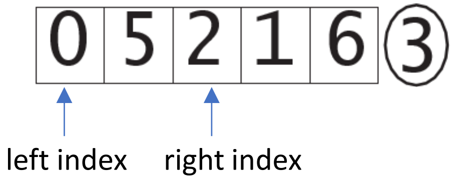
Wir wählen dann Indices —initial einen auf das Element des Arrays vor dem 0., und den Index 0 des Arrays.

0	5	2	1	6	3

left index
right index

Quicksort: Partitionierung im Array

Partitionierung:



1. Der rechte Index wird solange immer wieder auf die nächste Zelle erhöht,
   bis er einen Wert erreicht, der kleiner als das Pivotelement ist (oder das Ende des Arrays), und stoppt dann.

2. Erreicht der rechte Index einen Wert < Pivot, erhöht er den linken Pointer und tauscht den Wert
   mit dessen Inhalt.

3. Final vertauschen wir das Pivotelement mit dem Inhalt des um 1 erhöhten linken Index.

Quicksort: Partitionierung

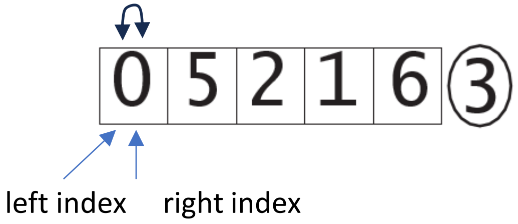
Nach der Partitionierung gilt:

- alle Werte links vom linken Index sind kleiner/ gleich dem Pivot
- alle Werte rechts vom Pivot sind größer als das Pivot
- das Pivot selbst ist jetzt am korrekten Platz im Array
- die anderen Werte sind untereinander noch nicht sortiert…

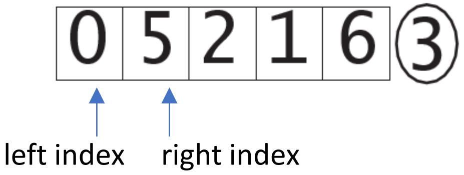
Quicksort: Partitioning

Anwendung auf das Beispiel:

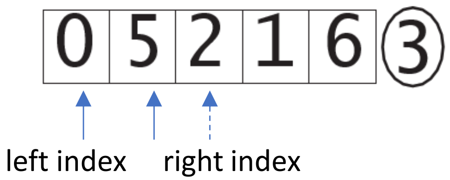
Schritt #1 (Wert 0):
Ist der Inhalt des rechten Index kleiner als das Pivot (Wert 3)?
Ja: linker Index wird erhöht, swap.



Schritt #2 (Wert 5): Der rechte Index wird erhöht:
Ist das Pivot erreicht?
Nein: Schritt 1.



Schritt #1 (Wert 5): Ist der Inhalt des rechten Index kleiner als das Pivot (Wert 3)?
Nein.

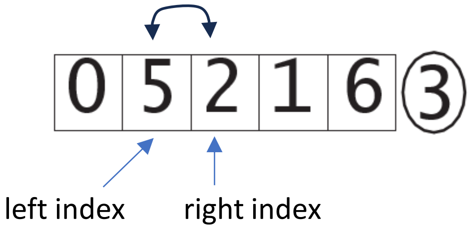


Schritt #2 (Wert 2): Der rechte Index wird erhöht:
Ist das Pivot erreicht?
Nein: Schritt 1.

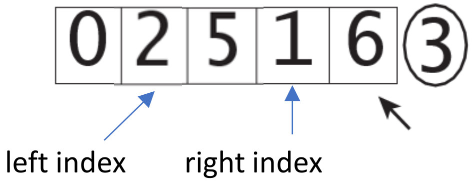
Quicksort: Partitioning

Anwendung auf das Beispiel:

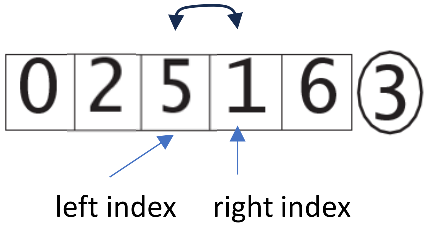
Schritt #1 (Wert 2): Ist der Inhalt des rechten Index kleiner als das Pivot (Wert 3)?
Ja: linker Index wird erhöht, swap.



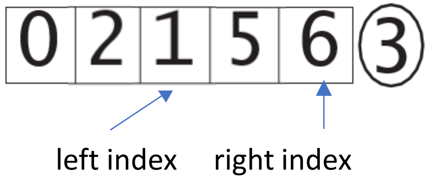
Schritt #2 (Wert 1): Der rechte Index wird erhöht:
Ist das Pivot erreicht?
Nein: Schritt 1.



Schritt #1 (Wert 1): Ist der Inhalt des rechten Index kleiner als das Pivot (Wert 3)?
Ja: linker Index wird erhöht, swap.



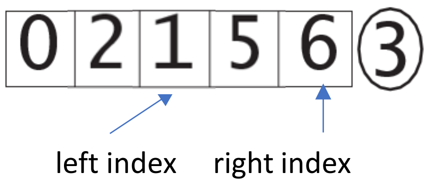
Schritt #2 (Wert 6): Der rechte Index wird erhöht:
Ist das Pivot erreicht?
Nein: Schritt 1.



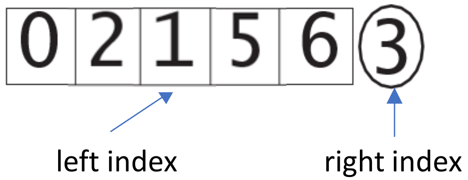
Quicksort: Partitioning

Anwendung auf das Beispiel:

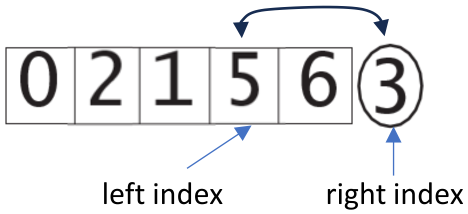
Schritt #1 (Wert 6): Ist der Inhalt des rechten Index kleiner als das Pivot (Wert 3)?
Nein.



Schritt #2 (Pivot): Der rechte Index wird erhöht:
Ist das Pivot erreicht?
Ja.



Schritt #3: Finaler Schritt der Partitionierung:
Pivot tauscht mit Inhalt des um 1 erhöhten linken Index. Stop.



Partitionierung ist erfolgreich abgeschlossen (Array ist <u>nicht</u> komplett sortiert).

Das Pivot (die 3) ist *jetzt an ihrem korrekten Platz im Array.*

Quicksort: Partitioning

```
public class Algorithms
{
  public int[]? NumArray { get; set; } // Array

  private int Partition  (int left, int right)  // between left and right border
  {
    int pivot = NumArray[right]; // select pivotPointer from right border
    int i = left - 1;

    for (int j = left; j < right; j++)
    {
      if (NumArray[j] <= pivot)  // number < pivot, swapped with latest smaller than pivot
      {
        i++;
        int temp = NumArray[i];
        NumArray[i] = NumArray[j];
        NumArray[j] = temp;
      }
    }

    int temp1 = NumArray[i + 1];  // Pivot swapped with the number next to the latest smaller one
    NumArray[i + 1] = NumArray[right];
    NumArray[right] = temp1;

    return i + 1; // new pivot position
  }
```

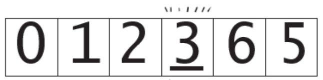
Quicksort: Partitioning

Der Quicksort-Algorithmus basiert stark auf Partitionierung:

1. Wir partitionieren das Array. Das Pivot ist jetzt am richtigen Platz.

2. Wir behandeln das linke und rechte Teilarray als eigenständige Arrays, und wiederholen rekursiv die Schritte #1 und #2. Wir partitionen dabei jedes Teilarray und erhalten kleinere Teil-Teil-Arrays links und rechts von jedem Teilarray-Pivot. Wir partitionieren dann diese Teil-Teil-Arrays, usw. usf. …

3. Haben wir nur noch Teilarrays mit 0 oder 1 Element, haben wir das Abbruchkriterium erreicht und wir stoppen.

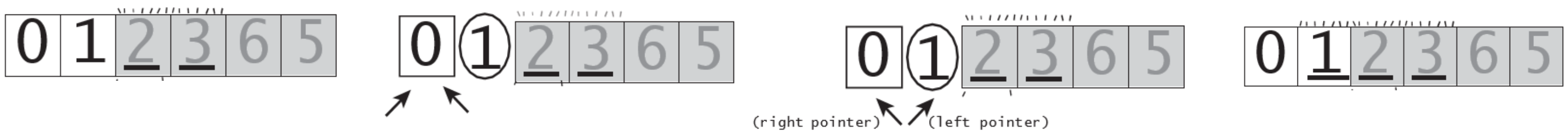
Initiale Partitionierung des Arrays [0, 5, 2, 1, 6, 3] für Quicksort: 

Dann:

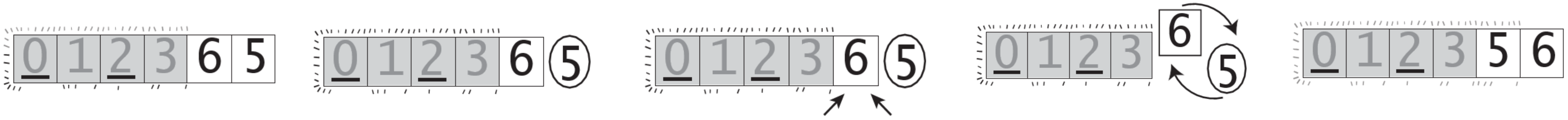
Alle Zahlen links vom Pivot als eigenes Array behandeln und partitionieren.

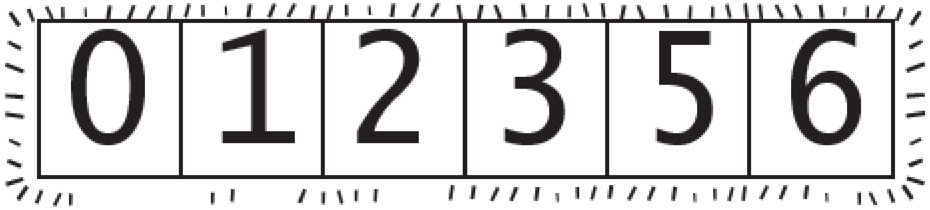
Quicksort: Partitioning

Wir haben nun ein Teilarray [0, 1] links vom Pivot (der 2) und kein Teilarray rechts. Nächstes Pivot ist damit die 1:



Nun fokussieren wir uns auf [6, 5]:





Quicksort: Implementierung

```
public class Algorithms
{
   public int[]? NumArray { get; set; } // Array

   private int Partition  (int left, int right)  // between left and right border
   {
      // ....
   }

   public void QuickSort(int left, int right)
   {
        if   (left < right)
        {
              int pivot = Partition(left, right);

              QuickSort( left, pivot - 1);
              QuickSort( pivot + 1, right);
        }
    }
}
```

Kein doppelter Aufruf!

Die Effizienz von Quicksort

Effizienz einer *einzelnen* Partitionierung aller Elemente:

Partitionierung hat 2 Arten von Schritten:

Vergleiche: Wir vergleichen jeden Wert mit dem Pivot *Vertauschungen*: Wenn nötig, vertauschen wir die Inhalte des linken und rechten Pointers
(abhängig vom Sortierungsgrad der Daten maximal N / 2 Vertauschungen).

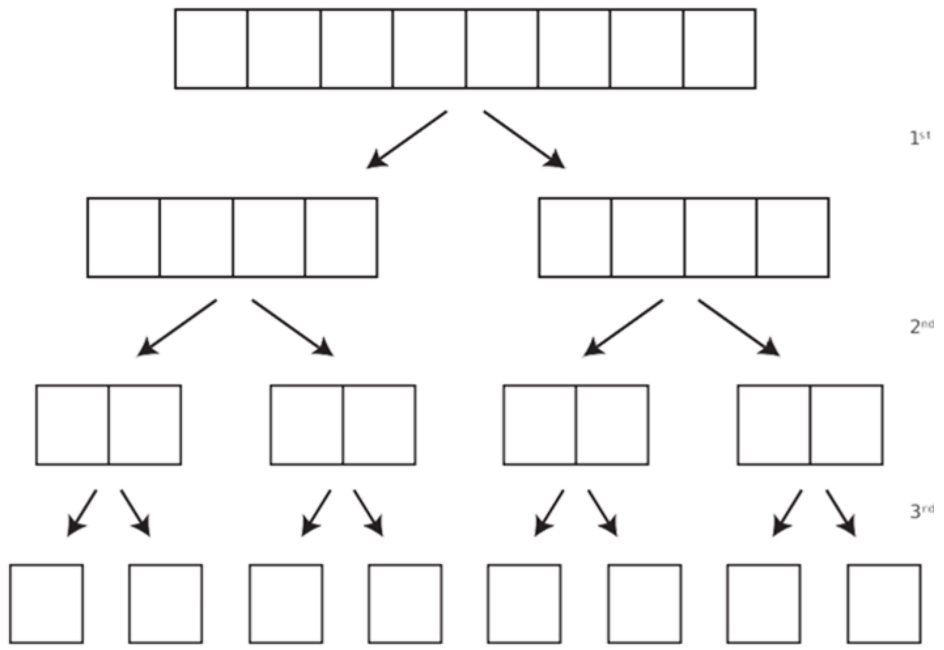
Bei *zufällig* sortierten Daten vertauschen wir ca. die Hälfte der Werte. Im Schnitt machen wir also N/4 Vertauschungen.

Das ergibt ca. 1.25N Schritte für N Datenelemente. In Big O-Notation: **O(N)**

Das ist die **Effizienz einer *einzelnen* Partitionierung.**

Die Effizienz von Quicksort

Wie häufig partitionieren wir die Elementmenge?



8 Elemente partitionieren wir log(n) mal!

Quicksort besteht aus einer Serie von Partitionierungen, und **jede Partitionierung** braucht N Schritte für **N Elemente eines Teilarrays.**
Gesamtzahl der Quicksort-Schritte:

Produkt aller nötigen Partitionierungen (log(n)) mit den zu partitionierenden Elementen (n)

Die Effizienz von Quicksort

N	Quicksort steps (approx.)
4	8
8	21
16	64
32	160

Die Anzahl der Quicksort-Schritte ist ca. *N * log N*:

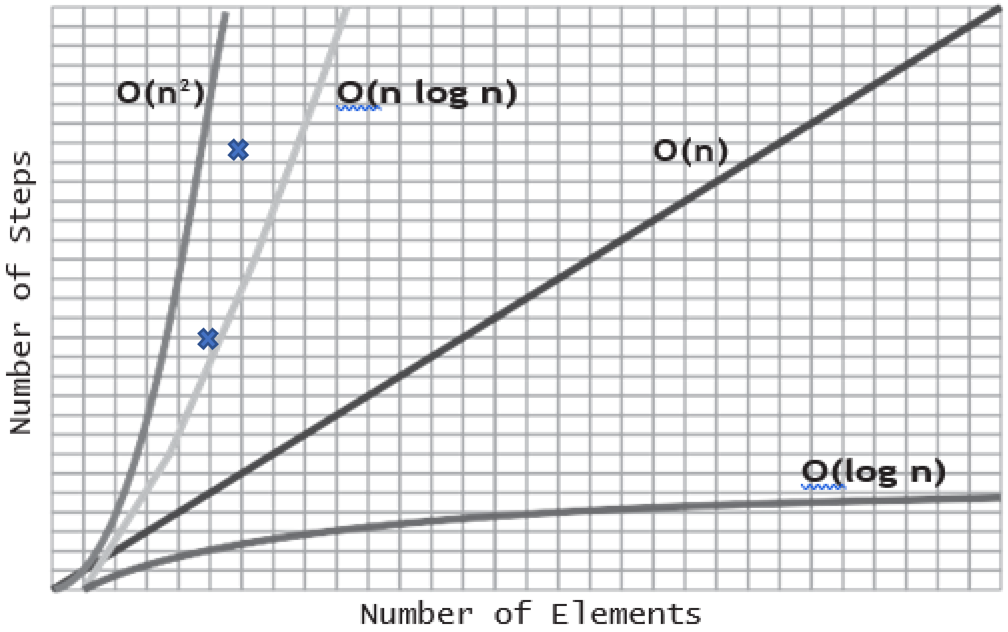
N	log N	N * log N
4	2	8
8	3	24
16	4	64
32	5	160

Quicksort ist ein ***O(N log N)*** *– Algorithmus im Average case*.

Eine für uns neue Big O - Kategorie!

Die Effizienz von Quicksort

Einordnung von O(N log N) in die anderen Kategorien von Big O:



Die Effizienz von Quicksort

Für ein Array der Größe 8 benötigen wir 3 “Halbierungen”, und wir schauen deshalb 3 x 8 Elemente an.

Quicksort benötigt N * log N Schritte:

Wir brauchen log N Halbierungen,
und für jede Halbierung führen wir eine Partitionierung aller Teilarrays durch,
deren Elementanzahl insgesamt N beträgt.

Übung

Die worst-case Effizienz von Quicksort ist:

A: O(1)
B: O(logN)
C: O(N)
D: O(N*logN)
E: $O(N^2)$

Übung

Die average-case Effizienz von Quicksort ist:

A: O(1)
B: O(logN)
C: O(N)
D: O(N*logN)
E: $O(N^2)$

Übung

Die best-case Effizienz von Quicksort ist:

A: O(1)
B: O(logN)
C: O(N)
D: O(N*logN)
E: $O(N^2)$

Abschliessender Vergleich

	Bubble sort	Insertion sort	Selection sort	Quick sort
Worst case:	N^2	$N^2 + N$	$N^2/2$	N^2
Best case:	N	N	$N^2/2$	N*logN
Average case:	$(N^2-N)/4$	$N^2/2$	$N^2/2$	N*logN

Knotenbasierte Datenstrukturen

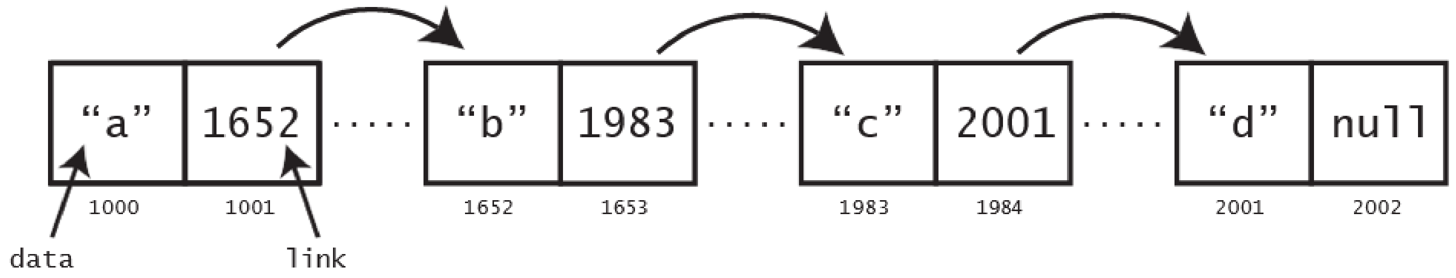
Knoten sind Teile verbundener Daten, die im Speicher gestreut liegen können

- Verkettete Listen: einfachste knotenbasierte Datenstruktur

- Statt einem fortlaufenden Speicherblock können die Knoten verketteter Listen verstreut sein.

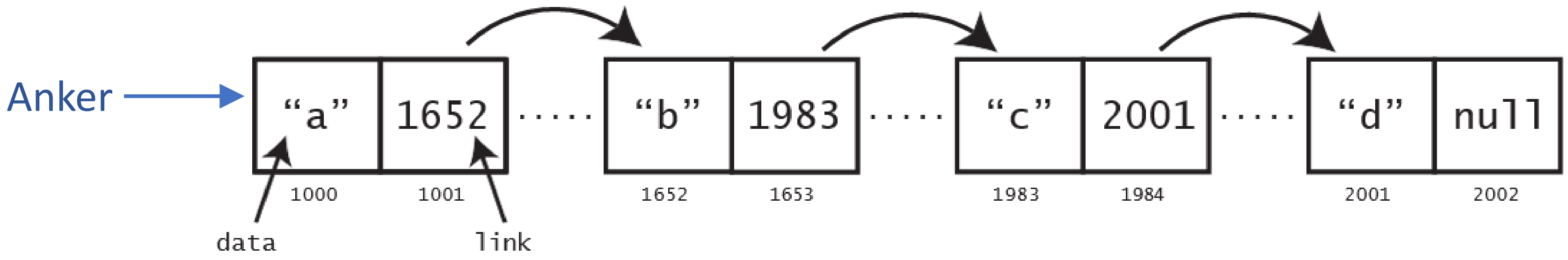
Woher weiss ein Algorithmus, welche Knoten in welcher Reihenfolge zur gleichen verketteten Liste gehören?

- jeder Knoten trägt eine Extrainformation: die Speicheradresse des nächsten Knoten in der Liste
- Typ: Pointer auf die Adresse des nächsten Elements



- Daten: vier Datenelemente: "a", "b", "c", und "d"

Wiederholung: Verkettete Listen



Die Beispielliste benötigt 8 Speicherzellen, da jeder Knoten aus 2 Speichereinträgen besteht:

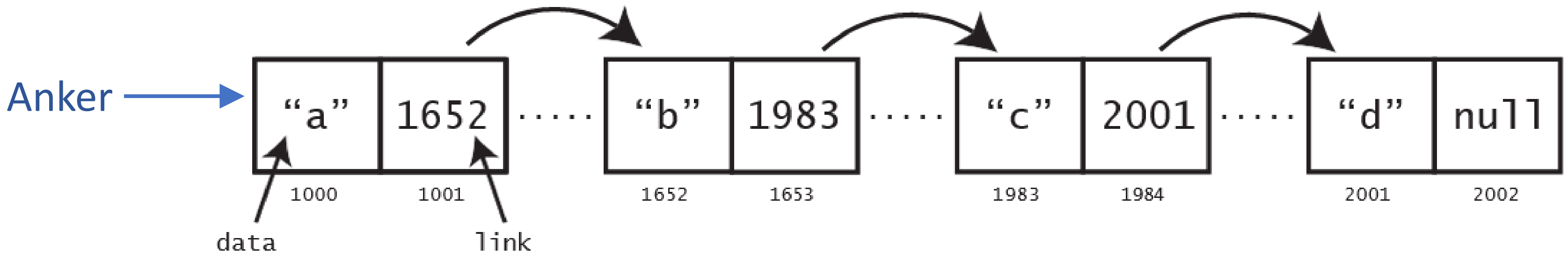
- die 1. Zelle jeden Knotens enthält die Daten,
- die 2. Zelle enthält den Pointer auf den nächsten Knoten

- Der Pointer des letzten Knoten enthält NULL (=0)
- In C verwenden wir die Datenstruktur "struct" für den Knotentyp

- Knoten können (müssen) dynamisch angefordert und hinzugefügt bzw. entfernt und freigegeben werden

- Unabhängig von der Datenstruktur selbst braucht man stets den Anker (oder "root") als Zeiger auf das erste Element/ den ersten Knoten

Verkettete Listen: Implementierung



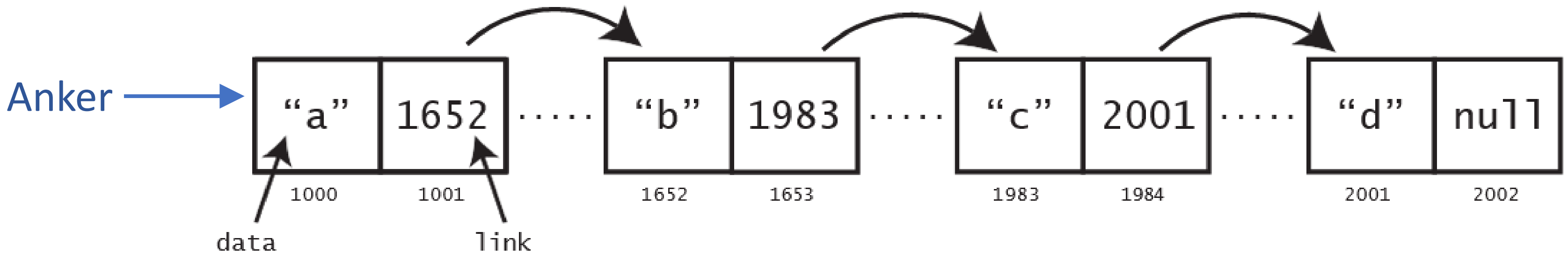
- Manche Sprachen (z.B. C#, Java) unterstützen Listen als eigene Datenstruktur, aber viele Sprachen auch nicht
- Struktur für den Knoten (Bsp: Daten als int-Wert):

```
struct node { int data; struct node *pNext; };
```

- Liste mit 2 Elementen:

```
struct node *pAnker=malloc (sizeof (struct node));  // Ankerpointer, root
struct node *pAnyNode= malloc (sizeof(struct node));// zweites Listenelement
pAnker->pNext=pAnyNode;
pAnyNode->pNext=NULL;
```

Zugriff, Suche, Einfügen und Löschen



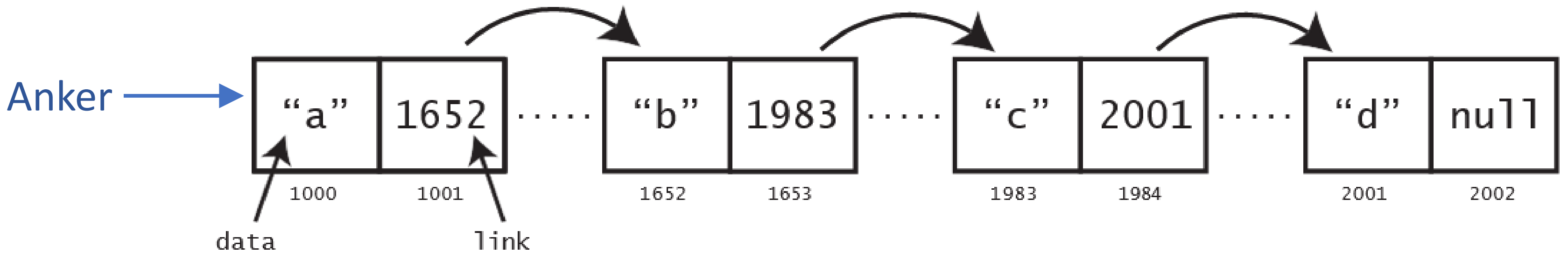
Zugriff:

Auf den Wert des 3. Listenelements kann nicht direkt zugegriffen werden!

Zuerst muss der erste Knoten adressiert werden. Dann folgt man dem Pointer des 1. Knotens zum 2., und dann dem Pointer des 2. zum 3. Knoten.

-> Verkettete Listen haben einen **worst-case-Zugriff** von **O(N)** (im Vergleich Arrays: O(1)).

Implementierung: Zugriff



Zugriffsfunktion:

```
int read (struct node *i_pAnker, int i_index, int *o_pData)
{               // We begin at the first node of the list:
                struct node *pCurrentNode = i_pAnker;
                int currentIndex=0;
                int retValue=EXIT_FAILURE;
                while ((currentIndex < i_index) && (NULL != pCurrentNode))
                {     // We keep following the links of each node until we
                      // get to the index we're looking for:
                      pCurrentNode = pCurrentNode->pNext;
                      currentIndex ++; □              }
                if (currentIndex == i_index)
                {
                      retValue=EXIT_SUCCESS;
                     *o_pData=pCurrentNode->data;
                }
                return retValue;
}
```

Implementierung: Suche nach Datenwert

- Die Suche in Arrays kostet O(N), weil der Algorithmus jeden Wert einzeln ansehen musss
- Nicht überraschend: verkettete Listen haben ebenfalls eine Suchgeschwindigkeit von O(N)!
- Gleicher Algorithmus wie beim Zugriff – aber nicht der Index wird verglichen, sondern der Datenwert…

: Aufruf: struct node *pElement= searchElement (pAnker, 42);

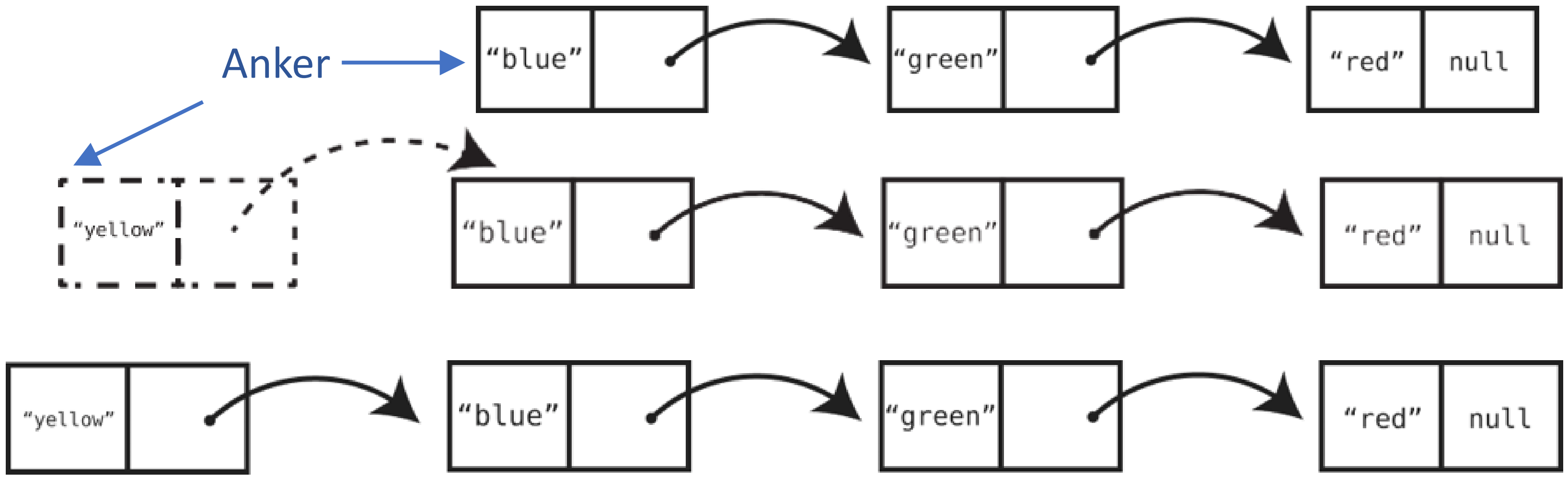
```c
struct node * SearchValue (struct node *i_pAnker, int i_value)
{
            // We begin at the first node of the list:
            struct node *pCurrentNode = i_pAnker;

            while ((NULL != pCurrentNode) && (pCurrentNode->data !=  i_value))
            {    // We keep following the links of each node until we get to
                 // the index we're looking for:
                 pCurrentNode = pCurrentNode->pNext;
            }
            return pCurrentNode;    // NULL if not found;
}
```

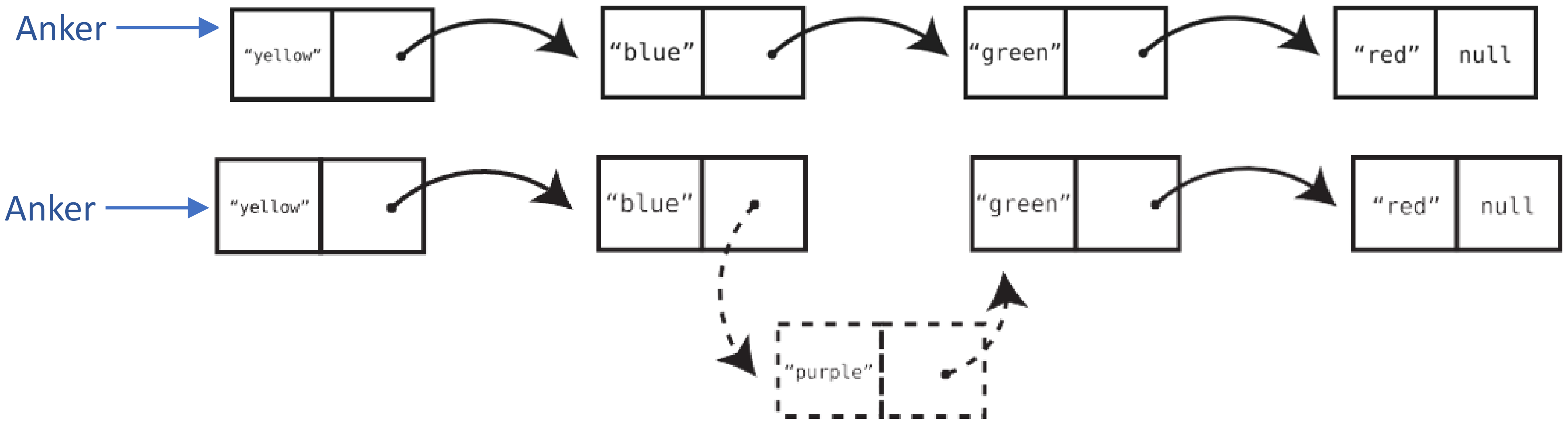
Implementierung: Einfügen

U.U. entscheidende Vorteile verketteter Listen beim Einfügen gegenüber Arrays:

- Das worst-case-Szenario für das Einfügen in Arrays ist das Einfügen am Index 0, weil alle Elemente des Arrays verschoben werden müssen (O(N))
- Das Einfügen am Beginn einer Liste kostet nur O(1).
- Das Einfügen in eine verkettete Liste kostet *überall* nur einen Schritt – wenn man bereits den Pointer auf das vorige Element hat, nach dem eingefügt werden soll



Implementierung: Einfügen



Schritt 1: Zugriff auf das Element mit dem Wert „blue": worst case O(n)
Schritt2: Einfügen

Szenario	Array	Verkettete Liste
Einfügen am Anfang	Worst case	Best case
Einfügen in der Mitte	Average case	Average case
Einfügen am Ende	Best case	Worst case

Implementierung: Einfügen nach einem Element

```
void InsertAfter (struct node *i_pBefore, int i_value)
{
    struct node * newPtr=malloc (sizeof (struct node));
    if (newPtr)
    {
      newPtr-> data = i_value;
      newPtr->pNext=i_pBefore->pNext;
      i_pBefore->pNext=newPtr;
    }
}
```

Löschen (nach einem Element)

Gleich zum Einfügen:

Szenario	Array	Verkettete Liste
Einfügen am Anfang	Worst case	Best case
Einfügen in der Mitte	Average case	Average case
Einfügen am Ende	Best case	Worst case

```
void DeleteAfter (struct node *i_pBefore)
{
    if (NULL != i_pBefore->pNext)
    {
        struct node * pToBeDeleted=i_pBefore->pNext;
        i_pBefore->pNext= i_pBefore->pNext->pNext;
        free (pToBeDeleted);
    }
}
```

Effizienz verketteter Listen

Operation	**Array**	**Verkettete Liste**
Zugriff	O(1)	O(N)
Suche	O(N)	O(N)
Einfügen	O(N) (O(1) am Ende)	O(1)
Löschen	O(N) (O(1) am Ende)	O(1)

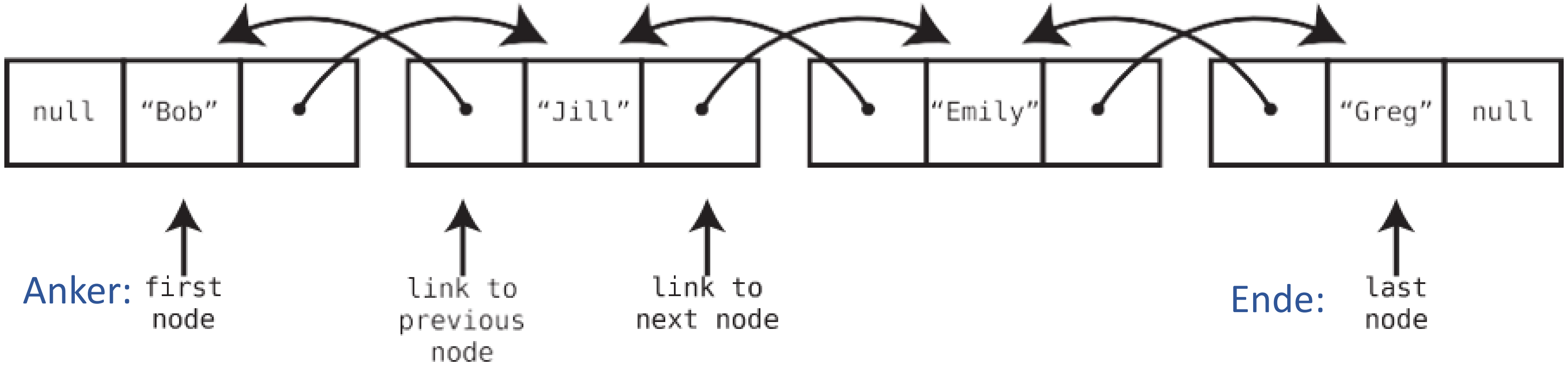
Die *reinen Schritte für's Einfügen und Löschen in verketteten Listen* sind nur O(1).

Meist haben wir das vorige Element durch vorangegangene Suchvorgänge schon in der Hand!

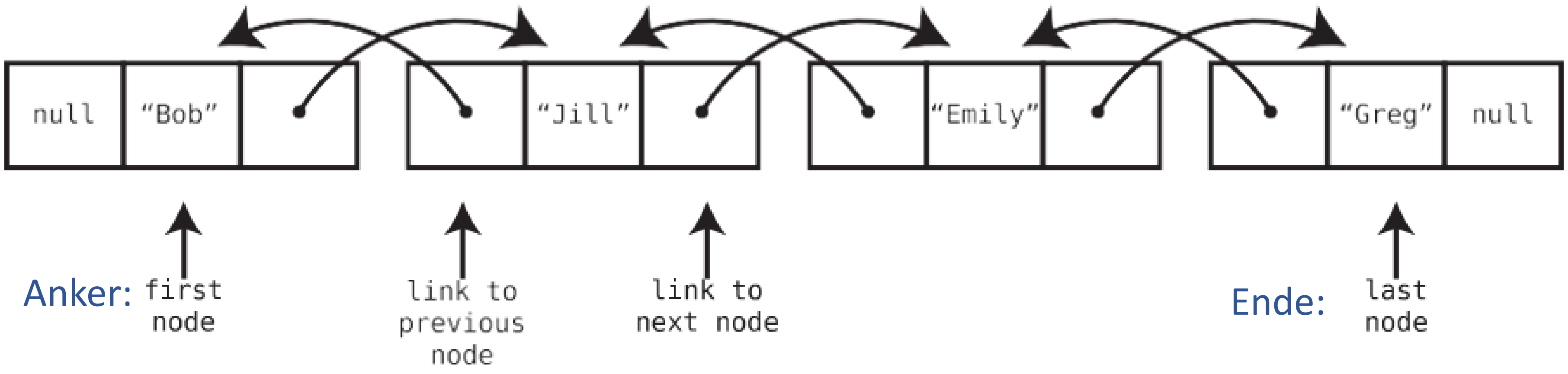
Doppelt verkettete Listen

Jeder Knoten hat *zwei Verbindungen*—eine zeigt auf den *nächsten*, die andere auf den *vorigen* Knoten.

Zusätzlich bewahrt man stets den **Anker auf den ersten Knoten** und den **Endepointer auf den letzten Knoten** auf!



Doppelt verkettete Listen: Implementierung

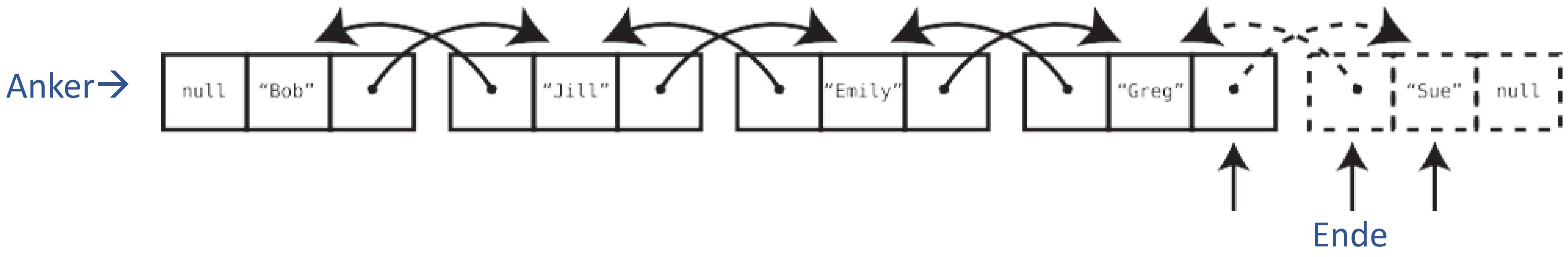


Doppelt verkettete Liste mit 2 Pointern: `struct nodeD {struct node *pPrev; int data; struct node *pNext; };`

```
struct nodeD *pDAnker=malloc(sizeof(struct nodeD));
struct nodeD *pDLast = malloc(sizeof(struct nodeD));

pDAnker->pNext=pDLast;
pDAnker->pPrev=NULL;
pDLast->pNext=NULL;
pDLast->pPrev=pDAnker;
```

Doppelt verkettete Listen: Einfügen



Einfügen am Ende einer doppelt verketteten Liste

Queues als doppelt verkettete Listen

Doppelt verkettete Listen haben:

- einen Direktzugriff zum Anfang der Liste
- einen Direktzugriff zum Ende der Liste

-> sie können Daten mit O(1) an jedem der beiden Enden einfügen und auch löschen.

Aufgrund dieser Eigenschaft sind **doppelt verkettete Listen** die perfekte **Basis-Datenstruktur für Queues**.

(Arrays sind O(1) für Einfügen am Ende, jedoch O(N) für Löschen am Beginn)
(Einfach verkettete Listen: O(N) für Einfügen/ Löschen am Ende, jedoch O(1) für Éinfügen/Löschen am Beginn)

Zusammenfassung

Feine Unterschiede zwischen Arrays und Listen schaffen neue Wege, unseren Code (z.B. für Queues) schneller zu machen.

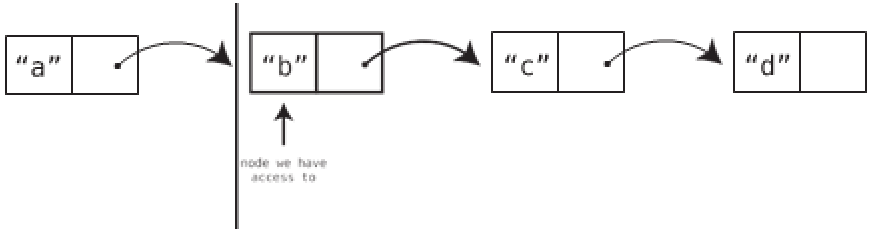
Verkettete Listen sind die Einfachste der knotenbasierten Strukturen.

Übung: 10 min. für Lösungsideen

Ein brilliantes kleines Puzzle verketteter Listen:

Sie haben Zugriff auf einen bestimmten Knoten irgendwo in der Mitte einer **einfach verketteten Liste**, nicht aber auf den Anker selbst.
Sie haben also nur einen Knoten, aber nicht die Instanz der Liste.
In dieser Situation haben Sie durch Verfolgen der Links Zugriff auf alle Knoten von dem bestimmten bis zum letzten, aber Sie haben keine Chance, Zugriff auf die Knoten davor zu bekommen.

Wie können Sie trotzdem diesen bestimmten Knoten effizient aus der Liste löschen (heißt: die Liste effizient um diesen Knoten verkürzen)?



Lösung