Theoretische Informatik II...

Besonderer Dank für bereitgestellte Inhalte gilt:

Prof. Dr. Andreas Mahr, Studiengang Informatik, DHBW Heidenheim, für die Bereitstellung seines Scripts

Jay Wengrow für die Betaversionen seines Buches „Data structures and algorithms“

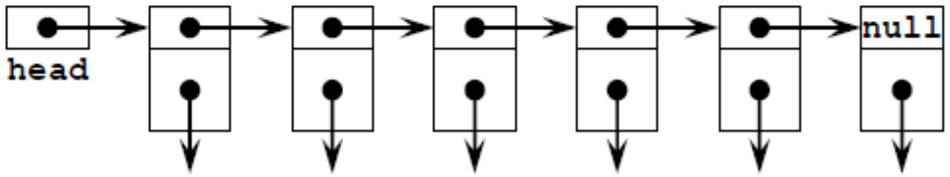
Alle Codebeispiele wurden bewusst nicht in Pseudocode angegeben, sondern in der Sprache C, die Sie schon beherrschen.

Lassen Sie die Beispiele laufen, spielen und experimentieren Sie damit – und sehen Sie mir Fehler nach!

Algorithmen und Datenstrukturen – wozu?

Inhalte der Vorlesung Algorithmen und Datenstrukturen

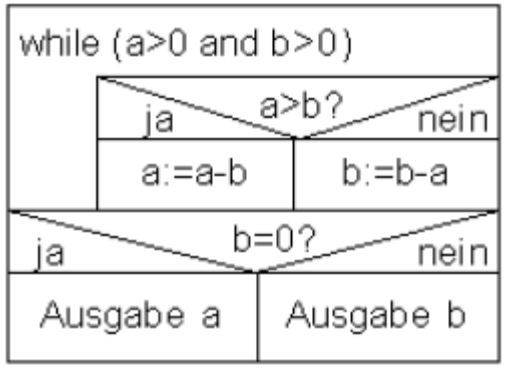
Daten speichern



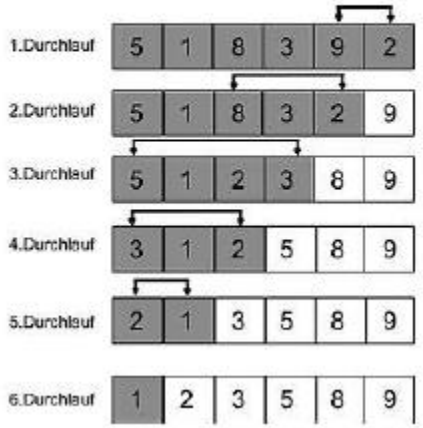
Verweis auf Nachfolger

Referenzen auf die in der Liste abgelegten Objekte

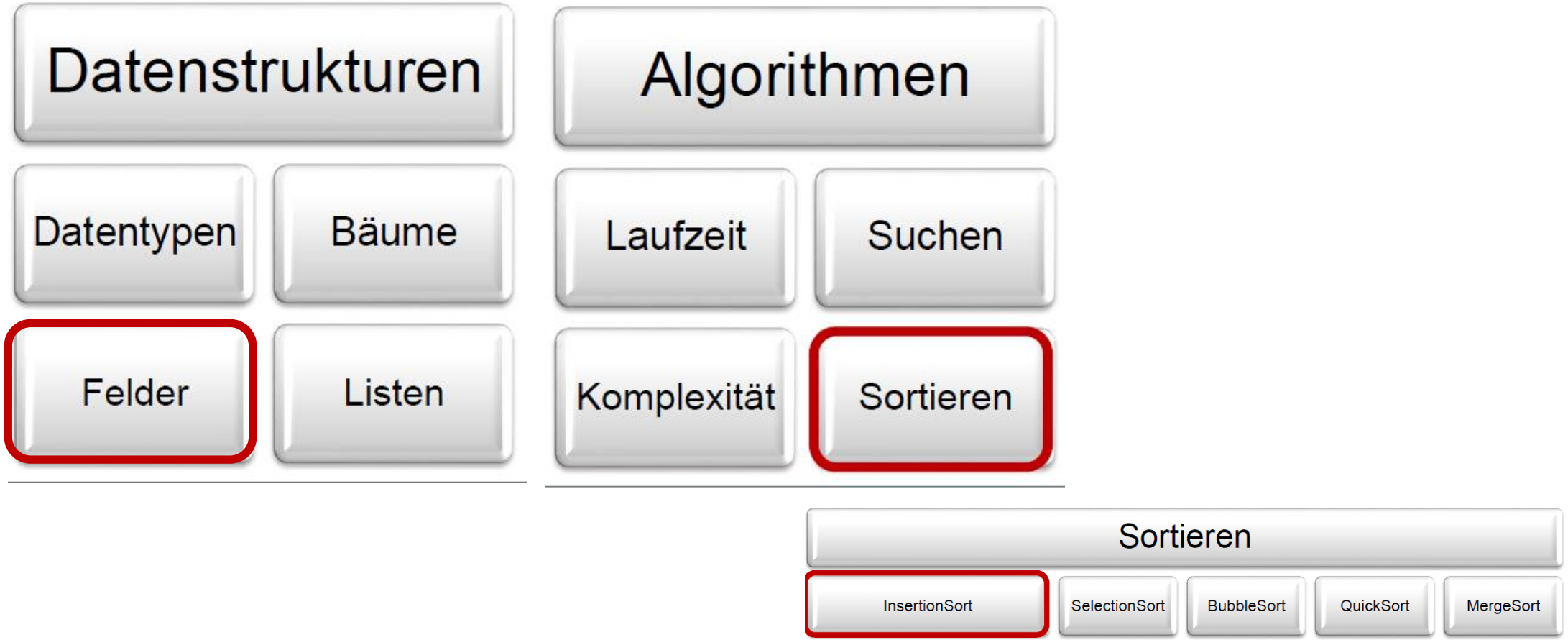
Daten mit Hilfe von Algorithmen bearbeiten



Daten sortieren

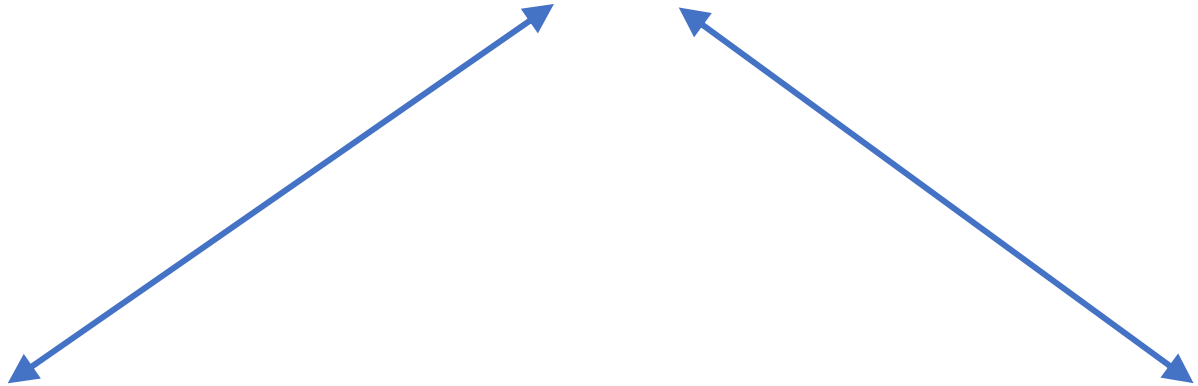


Algorithmen und Datenstrukturen – wozu?



Algorithmen und Datenstrukturen

Ziel: **Optimale algorithmische Problemlösung**



Optimale Datenstruktur zur Verwaltung/
Organisation der Eingangsdaten

Listen, Stacks, Queues, Bäume, Graphen

Optimaler Algorithmus auf
Basis der Datenstruktur

Einfügen, Löschen, Sortieren, Suchen, …
mit Divide&Conquer, Greedy-Algor., Rekursionen…

By the way: Was ist „optimal“? -> Einflußgrößen: Laufzeit, Komplexität, Speicherbedarf…

Algorithmen und Datenstrukturen

Diese beiden Funktionen geben jede alle geraden Zahlen zwischen 2 und 100 aus:

```c
void PrintNumbersVersionOne()
{
    int number = 2;
    while (100 >= number)
    {
        // If number is even, print it:
        if (0 == number % 2)
        {
            printf ("%d ", number);
        }
        number += 1;
    }
}
```

```c
void PrintNumbersVersionTwo ()
{
    int number=2;
    while (100 >= number)
    {
        printf ("%d ", number);

        // Increase number by 2, which, by definition,
        // is the next even number!
        number+=2;
    }
}
```

Läuft eine der Funktionen schneller, und warum?

A: Version 1 und 2 brauchen je 100 Schritte
B: Version 2 braucht halb so viele Schritte wie Version 1
C: Version 2 braucht doppelt so viele Schritte wie Version 1

Lösung

Diese beiden Funktionen geben jede alle geraden Zahlen zwischen 2 und 100 aus:

```c
void PrintNumbersVersionOne()
{
    int number = 2;
    while (100 >= number)
    {
        // If number is even, print it:
        if (0 == number % 2)
        {
            printf ("%d ", number);
        }
        number += 1;
    }
}
```

```c
void PrintNumbersVersionTwo ()
{
    int number=2;
    while (100 >= number)
    {
        printf ("%d ", number);

        // Increase number by 2, which, by definition,
        // is the next even number!
        number+=2;
    }
}
```

Läuft eine der Funktionen schneller, und warum?

A: Version 1 und 2 brauchen je 100 Schritte
✓ B: Version 2 braucht halb so viele Schritte wie Version 1
C: Version 2 braucht doppelt so viele Schritte wie Version 1

Stoffgliederung

Woche 1	Rolle der Datenstrukturen
	Rolle der Algorithmen
Woche 2	Vorträge
	Komplexität von Algorithmen: Big-O
Woche 3	Schnelleren Code mit Big-O: Bubble sort
	Codeoptimierung mit Big-O: Selection sort
Woche 4	Optimierungen für optimistische Szenarios: Insertion sort
	Schneller Lookup: Hash tables
	Hashes in der Codierung
Woche 5	Stacks und Queues für temporäre Daten
	Rekursionen
	Rekursiven Code schreiben lernen
Woche 6	Dynamisches Programmieren
	Rekursion für mehr Geschwindigkeit: Quicksort
Woche 7	Knotenbasierte Datenstrukturen
	Bäume
	Binäre Suchbäume
Woche 8	Prioritätenmanagement mit Heaps
Woche 9	AutoComplete mit Präfixbäumen
	Graphen für soziale Netzwerke
Woche 10	Gewichtete Graphen für Kürzeste Wege (Dijkstra)
	Gewichtete Graphen für Komprimierung (Huffman)
	Minimale Spannbäume (Kruskal, Prim)
Woche 11	Umgang mit Speicherlimitationen: MergeSort CountingSort
	Verwaltung großer, externer Datenmengen
	B-Bäume, B*-Bäume

Literatur

Sedgewick, Robert: *Algorithmen*, Pearson Verlag 2014 (diverse Auflagen in Pseudocode, C, C++, Java)
– wenn Sie nur 1 Buch kaufen, dann das – für Ihre Lieblings-Programmiersprache!

Th. H. Cormen, Ch. E. Leiserson, R. Rivest, C. Stein: *Algorithmen – Eine Einführung*, 2. Auflage, Oldenbourg Verlag, 2007.

Uwe Schöning, *Algorithmik*, Spektrum Akademischer Verlag, 2001.

J. Hromkovič, *Algorithmics for Hard Problems*, 2nd Edition, Springer, 2003.

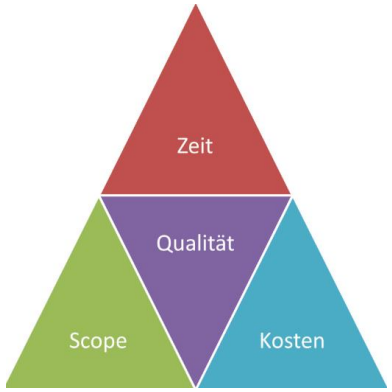
S. Dasgupta, C. Papadimitriou, U. Vazirani, „*Algorithms*“, McGraw-Hill, 2007.

V. Heun, „*Grundlegende Algorithmen*“, 2. Auflage, Vieweg, 2003.

J. Kleinberg, E. Tardos, „*Algorithm Design*“, Pearson Education, 2005.

Die Rolle der Datenstrukturen…

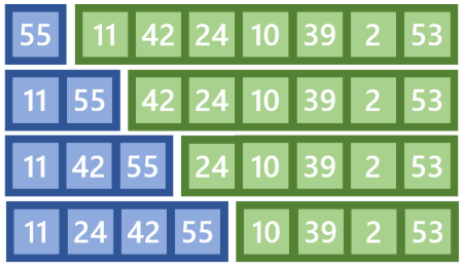
- Effizienterer, schnellerer Code:

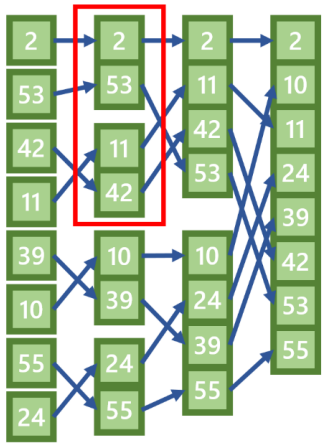


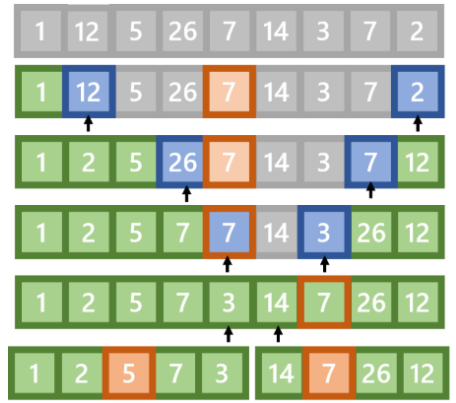
- Eleganter Code:

```
// I don't know what I did, but it works...
// Please don't modify it!
```

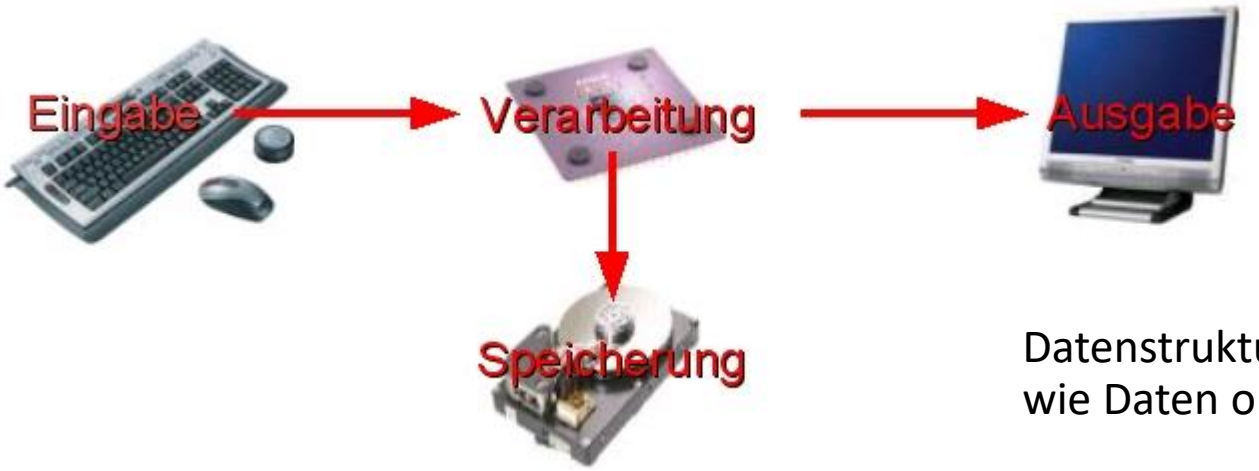
- Algorithmen über verschiedenen Datenstrukturen vergleichen:







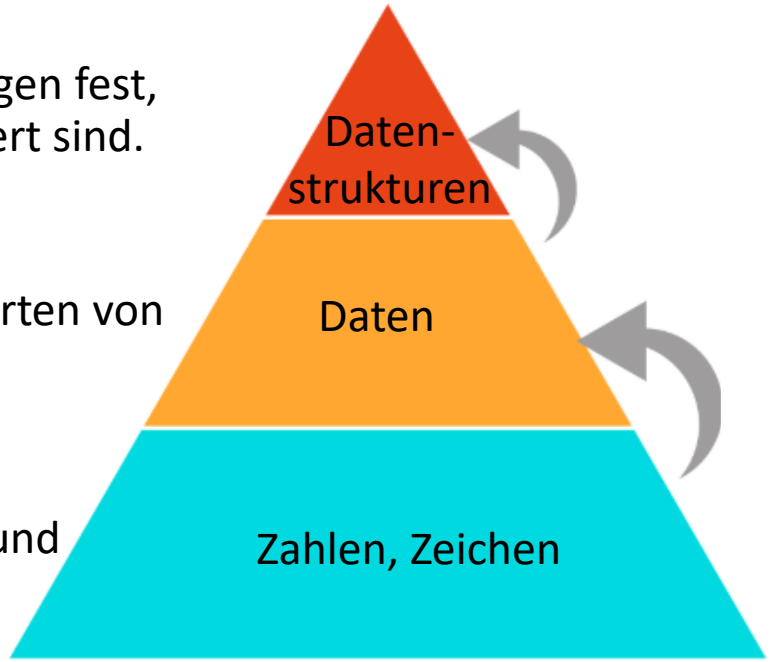
Die Rolle der Datenstrukturen …



Datenstrukturen legen fest, wie Daten organisiert sind.

Daten meinen alle Arten von Information,

angefangen von den elementaren Zahlen und Zeichen



Die Organisation von Daten ist kein Selbstzweck, sondern bestimmt signifikant, wie schnell Ihr Code läuft.

Die Rolle der Datenstrukturen … Das Array…

Das Array ist eine der ganz elementaren Datenstrukturen

```c
int main(int argc, char** argv)
{
    char *x = "Hello! ";
    char *y = "How are you ";
    char *z = "today?";

    printf ("%s%s%s", x,y,z);

    return (EXIT_SUCCESS);
}
```

```c
int main(int argc, char** argv)
{
    char *array []= {"Hello! ", "How are you ", "today?"};

    printf ("%s%s%s", array[0], array[1],array[2]);

    return (EXIT_SUCCESS);
}
```

Basis-Datenstrukturen: Array

- Das Array ist die grundlegende Datenstruktur in der Informatik:
  n gleiche Datenelemente, dicht aneinander im Speicher angeordnet

- Der Index eines Arrays ist die Zahl, die identifiziert, wo innerhalb des Arrays ein Datenelement liegt

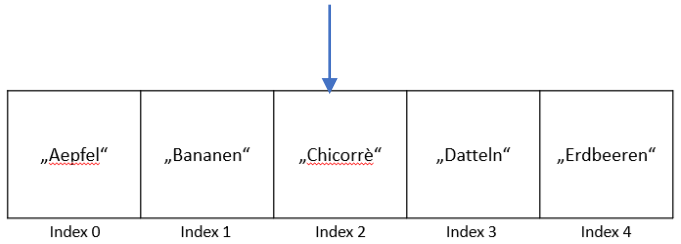

```
char *array []= {“Aepfel", “Bananen", “Chicorrè", “Datteln", “Erdbeeren“};
```

„Aepfel“	„Bananen“	„Chicorrè“	„Datteln“	„Erdbeeren“
Index 0	Index 1	Index 2	Index 3	Index 4

Basis-Datenstrukturen: Array

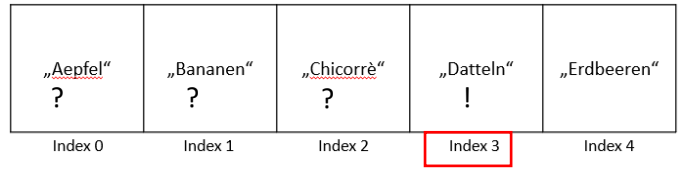
Nutzung von Datenstrukturen durch **4 Basis-Zugriffsarten/ -Operationen**:

- **Direktzugriff Lesen/ Ändern***:* **Zugriff** auf Daten an einer **bestimmten Stelle** innerhalb der Datenstruktur.

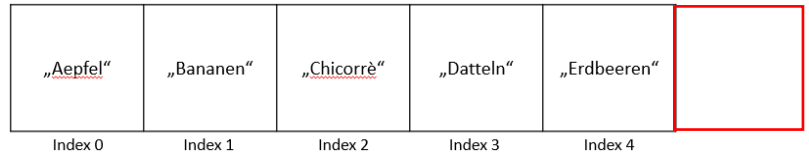


- **Suchen:** Suche bedeutet das **Auffinden eines bestimmten Wertes** in einer Datenstruktur.

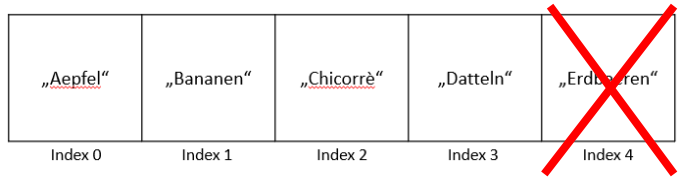
Datteln?



- **Einfügen:** Einfügen bedeutet, einen **weiteren Wert** zu unserer Datenstruktur **hinzuzufügen**.



- **Löschen:** Löschen meint das **Entfernen eines Wertes** aus unserer Datenstruktur.



Geschwindigkeit dieser Operation auf einem Array

Absolute Zeiten sind abhängig vom Rechnertyp, Umgebung, Ressorcen,…

-> Vergleichswert für Performance: Notwendige Schritte!

Wenn Operation A **5 Schritte** benötigt, und Operation B **500 Schritte**, können wir davon ausgehen, dass Operation A auf der gleichen Hardware immer schneller laufen wird als Operation B – unabhängig von der Basishardware!

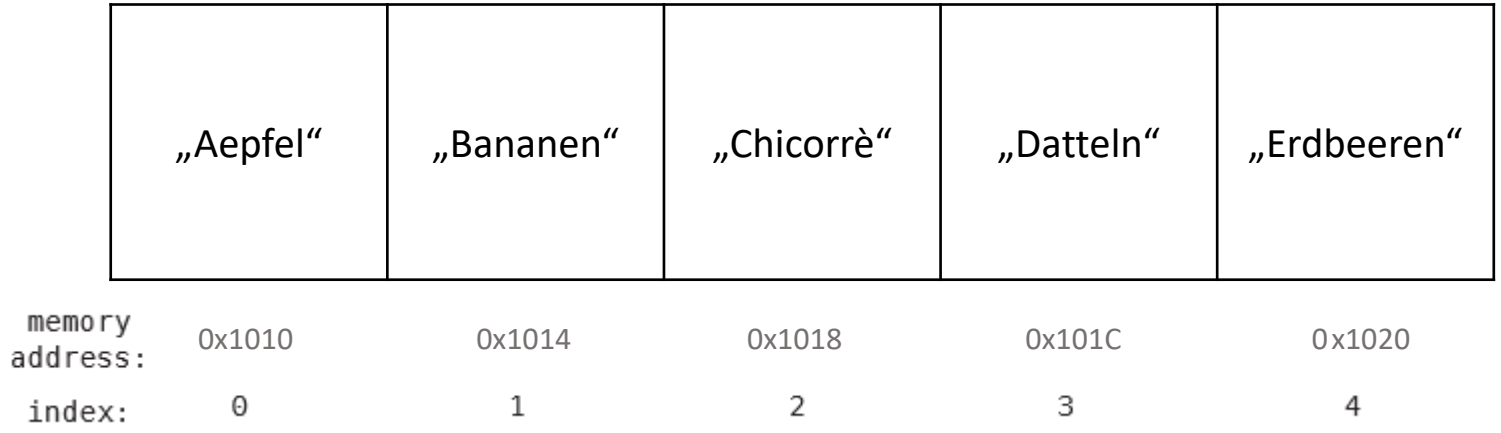
Synonyme Begriffe: ***Geschwindigkeit***, ***(Zeit-) Komplexität***, ***Effizienz***, und ***Performance*** – gemessen in Schritten.

Geschwindigkeit von Operationen auf einem Array

1) **Direktzugriff**:

Genau 1 Schritt (max. die Addition des Index auf die Adresse), da Direktzugriff über die Adresse/ Index (feste Relation zwischen erster Speicheradresse und erstem Index während der gesamten Laufzeit!).

&array[0] == array !

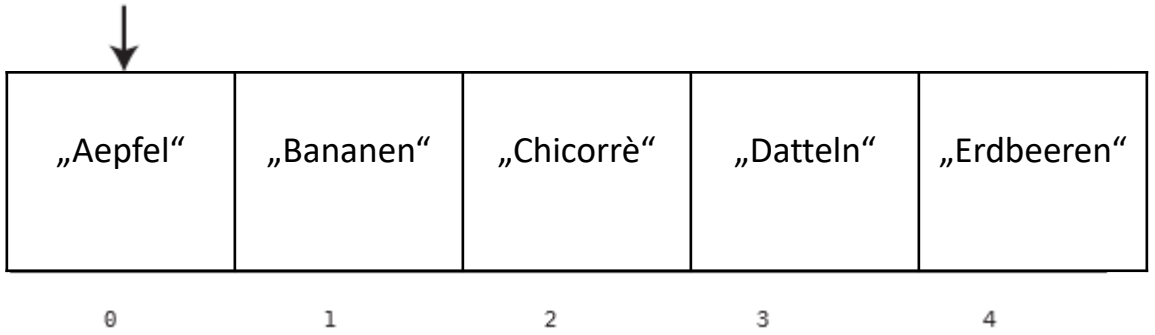


Geschwindigkeit von Operationen auf einem Array

2) Suche nach Datenwert:

Invers zum Direktzugriff:

Nicht Direktzugriff auf Wert eines Index/Adresse, sondern sequentielle (lineare) Suche nach dem Wert und Ermittlung dessen Index/ Adresse!



Suche nach Datteln:

4 Vergleiche, Index: 4 - 1

Arraylänge: n -> **maximale Schritte: n** (n Vergleiche im worst case, wenn gesuchter Wert nicht vorhanden ist)

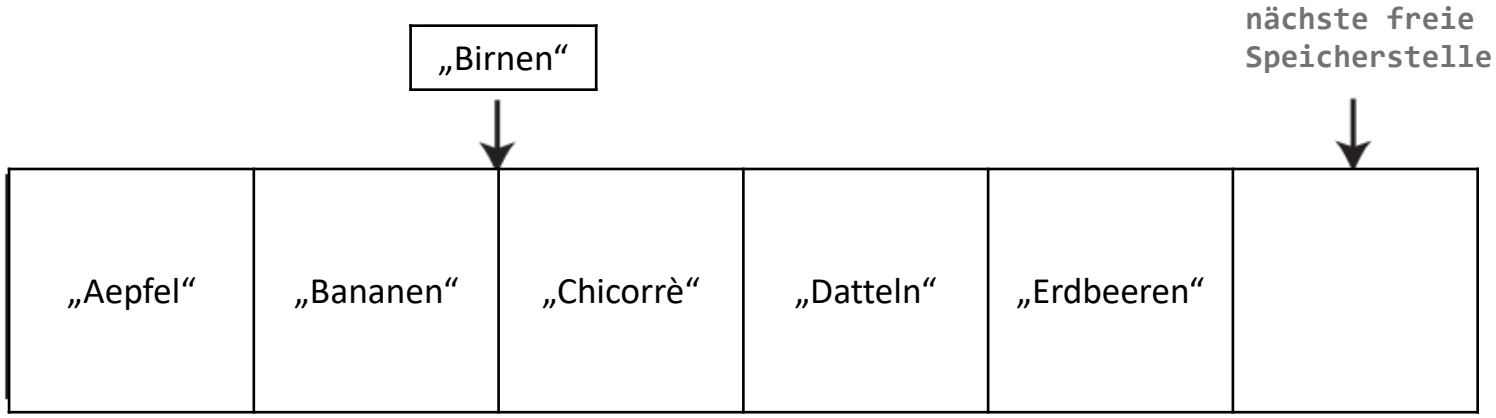
Geschwindigkeit von Operationen auf einem Array

3) Einfügen eines Datenwerts in ein Array:

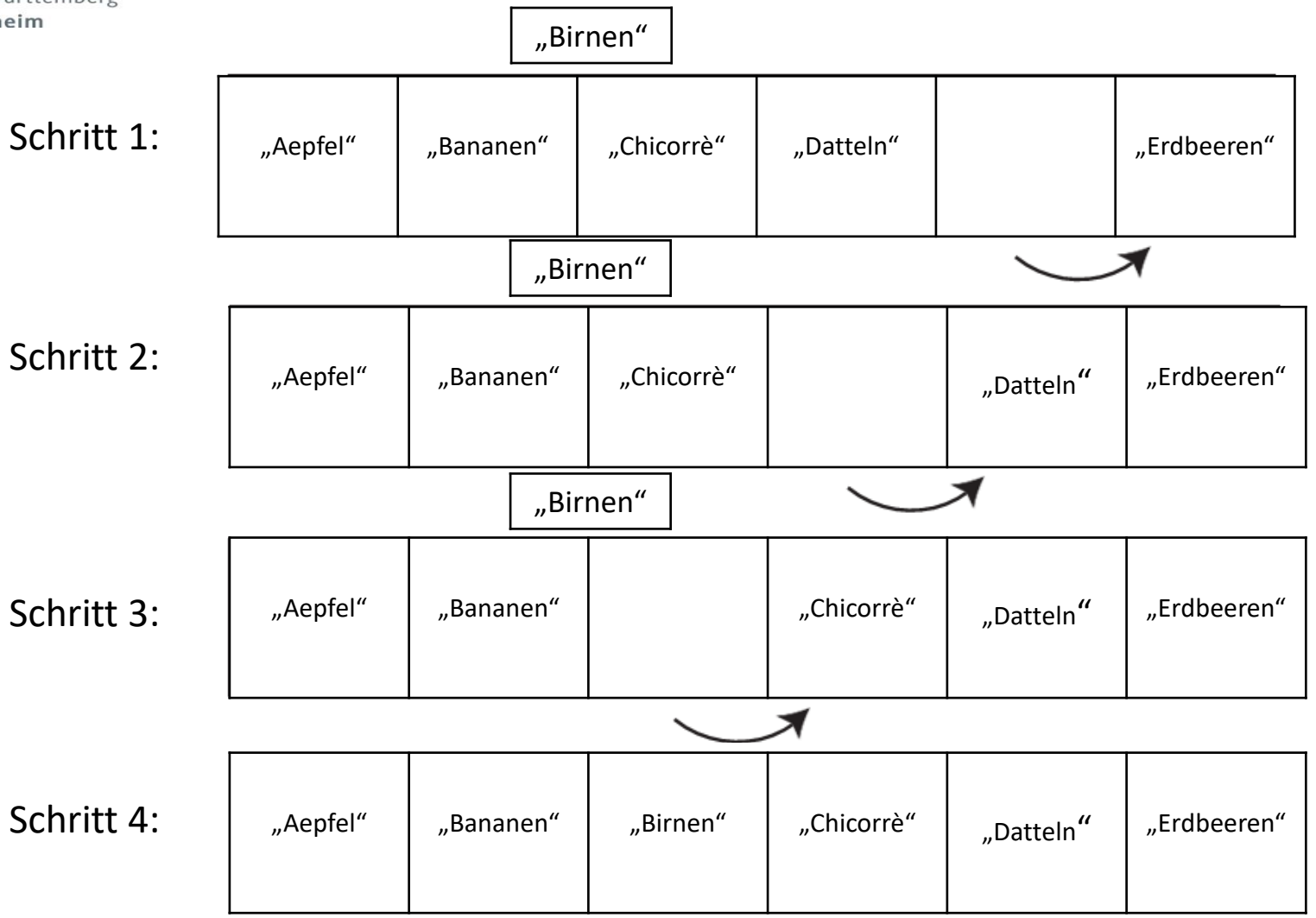
Am Ende einfügen (nach letztem Element): 1 Schritt (Programmiersprachen reallokieren Speicher unterschiedlich)

Am Anfang oder in der Mitte einfügen: Daten müssen verschoben/ kopiert werden

Feigen sollen an den Index 2 eingefügt werden:



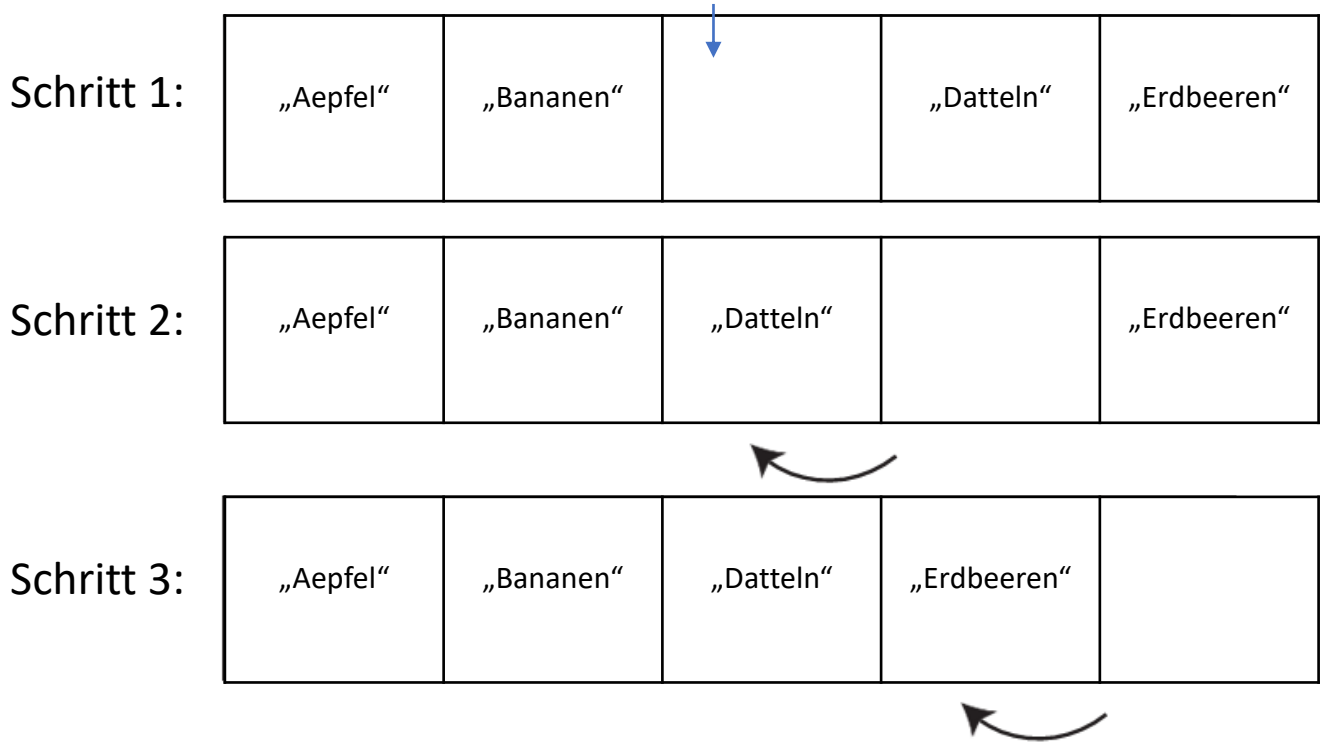
Geschwindigkeit von Operationen auf einem Array



Worst case: Einfügen am Beginn, Arraylänge: n -> maximale Schritte: **n + 1** (n x Schieben + Schreiben)

Löschen eines Elements

4) Invers zum Einfügen: **Löschen** des Chicorrès



Worst case: Löschen am Beginn, Arraylänge: n -> maximale Schritte: n (1 x Löschen + n-1 x Schieben)

Komplexität von Operationen

Wir bestimmen die Effizienz/ Komplexität von Operationen immer in Abhängigkeit von der Größe der Datenmenge!

Besondere Arrays: Sets

Sets sind Arrays, die keine doppelten Datenwerte erlauben (Bsp: Lookup-Tabellen wie Telefonbücher…)

Sie haben nur diese einzige zusätzliche Einschränkung zu den Arrays!

Zugriff/ Lesen:	Komplexität (Schritte) wie bei Arrays
Suchen:	Komplexität (Schritte) wie bei Arrays
Löschen:	Komplexität (Schritte) wie bei Arrays
Einfügen/ Schreiben:	?

Sets: Einfügen und Ändern

Zur **Vermeidung von doppelten Einträgen**: vor jedem Schreib- und Einfügevorgang zunächst Suche, ob der Wert bereits vorhanden ist!

Für Einfügen am Anfang (Array: n+1) zusätzlich: worst case: n Schritte (Vergleiche)

Für Schreiben an Index (Array: 1) zusätzlich: worst case: n Schritte (Vergleiche)

Summe bei Set:

Einfügen: worst case: 2n + 1 Schritte

Schreiben: worst case: n + 1 Schritte

Zusammenfassung Datenstrukturen

Die **Komplexität von Operationen** auf Datenstrukturen bestimmen heißt:

Bestimmen, wie viele Schritte auf dieser Datenstruktur diese Operation in Abhängigkeit von der Anzahl der Elemente der Datenstruktur benötigt.

-> die richtige Datenstruktur für notwendige Operationen wählen!
(Es gibt wesentlich mehr Datenstrukturen als Arrays und Sets…)

-> nächstes Kapitel: Performance verschiedener **Algorithmen** auf einer Datenstruktur vergleichen

Die Rolle der Algorithmen…

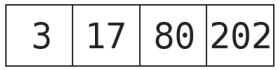
Die richtig gewählte Datenstruktur kann die Performance Ihres Codes signifikant beeinflussen, aber:

- Selbst wenn wir uns für die richtige Datenstruktur entschieden haben, beeinflusst ein anderer wichtiger Faktor die Effizienz des Codes: die adäquate Auswahl des *Algorithmus.*
- Ein Algorithmus ist *"… eine wiederholbare Handlungsanweisung für einen Computer auf einer Menge von Eingangsdaten"*.
- Ein Algorithmus löst ein definiertes Problem, das Ergebnis muss spezifierten Kriterien genügen
- Meist kann das Problem durch mehrere verschiedene Algorithmen gelöst werden
- Auf einer Datenstruktur kann ein Algorithmus schneller/ effizienter sein als ein anderer Algorithmus

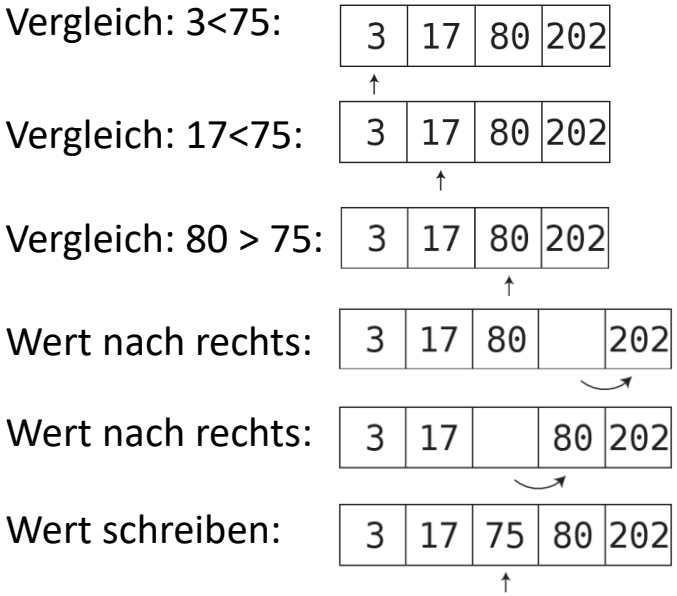
Neue Datenstruktur: Das sortierte Array

Einziger Unterschied des sortierten Arrays zum Array:
Werte müssen stets (nach jeder Operation wieder) einer vorgegebenen Sortierung genügen

Beispiel: Array [3, 17, 80, 202]



Das sortierte Array: Einfügen



Sortiertes Array, Einfügen (Worst case): n Elemente, m= Index des <u>größeren</u> Elements

Vergleiche (m+1) + Verschieben (n-m) + Schreiben (1) = n + 2 (unsortiertes Array: n+1)

Das sortierte Array: Lineare Suche

Beispiel: Array [3, 17, 75, 80, 202], suchen nach “22” – kann im sortierten Array bei “75” abgebrochen werden!

```
int LinearSearch (int i_array[], int i_size, int i_searchValue)
{
    bool continueLoop=true;
    int result=-1;
    for (int i=0; ((i<i_size) && continueLoop); i++)
    {
        if (i_array[i] == i_searchValue)
        {
            continueLoop = false;    // found!
            result=i;
        }
        else if (i_array[i] > i_searchValue) continueLoop = false;          // found larger element – not contained ☹
            // hier bricht die Suche im sortierten Array u.U. früher ab!
    }
    return result;
}

int main(int argc, char** argv)
{
    int array[]={3, 17, 75, 80, 202};
    int result=LinearSearch (array, sizeof(array)/sizeof(int), 22);

    return (EXIT_SUCCESS);
}
```

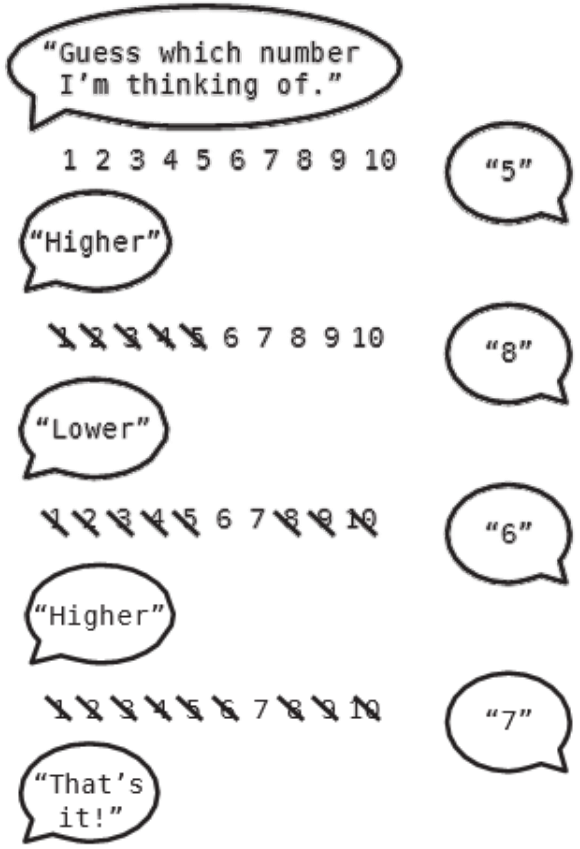
Aber: Kein Unterschied zur Suche im unsortierten Array beim Worst case (alle Elemente < gesuchtes) ☹!

Das sortierte Array: Binäre Suche

Es gibt andere Suchalgorithmen über sortierten Arrays!

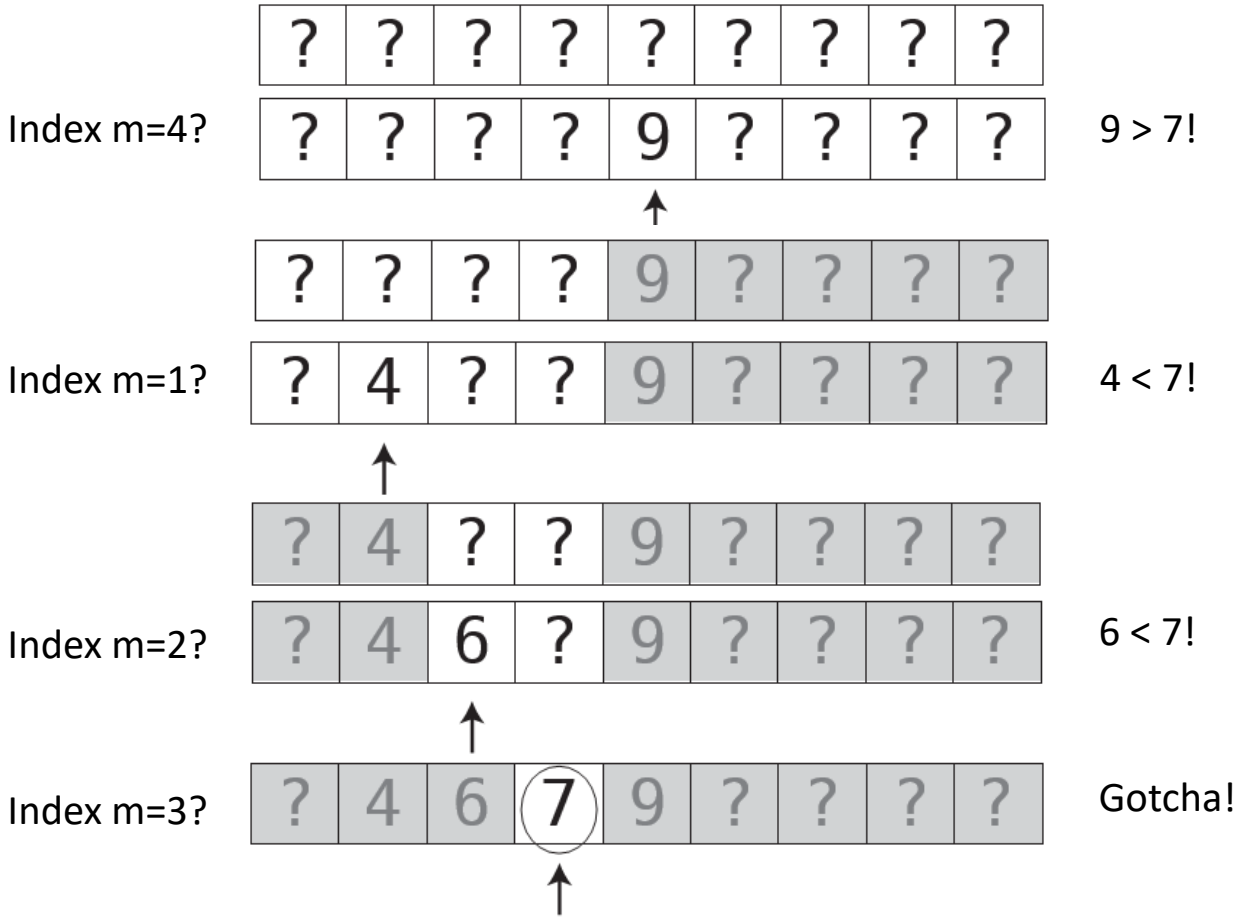
Beispiel: Binäre Suche

(„Ich denke mir eine Zahl zwischen 1…100,
Du rätst, und ich sage Dir, ob höher oder niedriger…“ –
Du wirst in der Mitte starten zu raten…)



Das sortierte Array: Binäre Suche

Binäre Suche nach „7“ in einem sortierten Array der Länge n=9:



Das sortierte Array: Binäre Suche

Implementierung:

```
int BinarySearch (int i_array[], int i_size, int i_searchValue)
{   //  First, we establish the lower and upper bounds of where the value we're searching for can be.
    //  To start, the lower bound is the first value in the array, while the upper bound is the last value:
    int lowerBound = 0;  // unterer Index
    int upperBound = i_size - 1; // initialer oberer Index
    int midPoint=0;
    int result= -1;
    bool notFound = true;
    //  We begin a loop in which we keep inspecting the middlemost value between the upper and lower bounds:
    while ((lowerBound <= upperBound) && (notFound))  // noch nicht gefunden und Indizes nicht gekreuzt
    {
        // We find the midpoint between the upper and lower bounds:
        midPoint = (upperBound + lowerBound) / 2; // Index des neuen Kandidats
        // We inspect the value at the midpoint:
        if (i_array[midPoint] == i_searchValue)  // getroffen?
        {
            result=midPoint;
            notFound=false;
        }
        else
        {
            if (i_searchValue < i_array [midPoint]) upperBound=midPoint-1; // oberen Index absenken
            else if (i_searchValue > i_array [midPoint]) lowerBound= midPoint+1;  // unteren Index anheben
        }
    }
    return result;
}
```

```
int main(int argc, char** argv)
{
    int array[]={3, 17, 75, 80, 202};
    int result=BinarySearch (array, sizeof(array)/sizeof(int), 22);

    return (EXIT_SUCCESS);
}
```

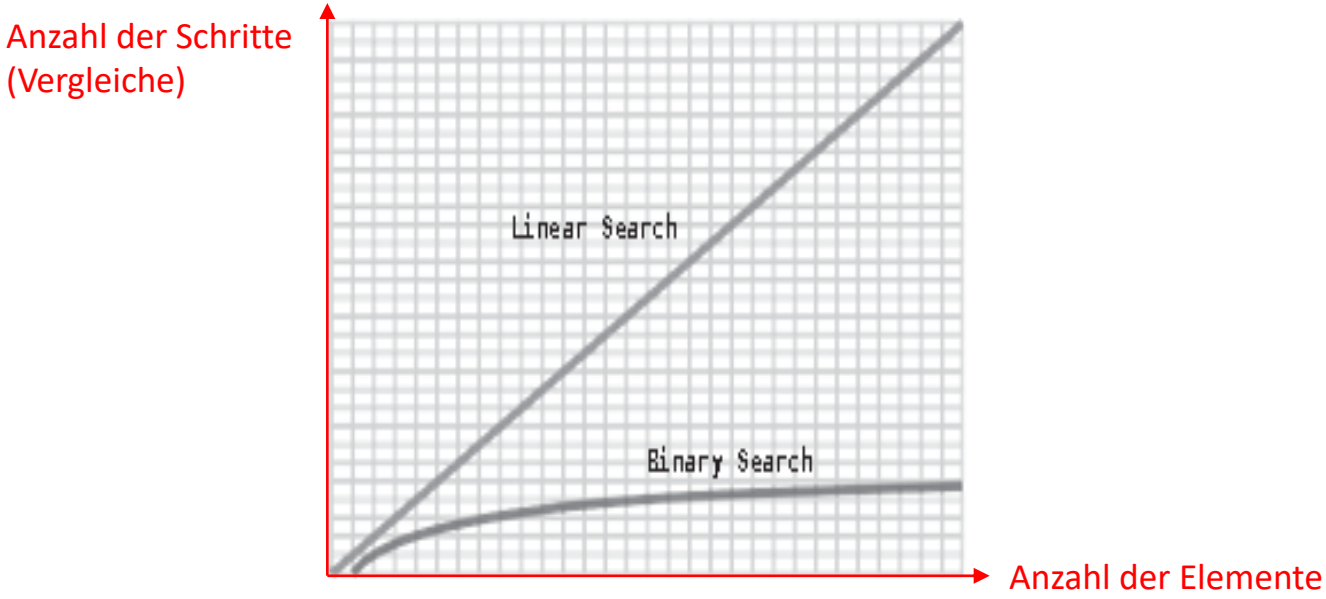
Das sortierte Array: Binäre vs. Lineare Suche

Gegeben: Sortiertes Array der Länge n=100

Lineare Suche (Worst case): 100 Vergleiche (=n)

Binäre Suche (Worst case): 7 Vergleiche (=$\log_2$ n: Die Verdopplung der Länge erhöht die Zahl der Schritte um 1)

(„Errate eine Zahl…“) Ein Schritt mehr bei jeder Verdopplung der Länge (x*2 -> y+1)



Das sortierte Array: Binäre vs. Lineare Suche

Sortierte Arrays:

- Einfügen ist langsamer als bei unsortierten Arrays
- Lineare Suche ist genauso schnell wie bei unsortierten Arrays
- Aber: Binäre Suche ist wesentlich schneller als bei unsortierten Arrays!

Daten-struktur / Algo-rithmus	**Unsort. Array**	**Sortiert. Array**
Lineare Suche	n Schritte	n Schritte
Binäre Suche	_	log n Schritte
Einfügen	n+1 Schritte	n+2 Schritte

Vor der Auswahl der Datenstruktur (unsortiertes/ sortiertes Array) und des Algorithmus (Lineare/ Binäre Suche) ist also zu klären:

Muss die Applikation häufig Elemente einfügen? Ist die Suche ein häufig benutztes Feature?

Zusammenfassung Algorithmen

- Es gibt nie nur die eine perfekte Datenstruktur und den einen perfekten Algorithmus für ein Problem.
- Die Analyse konkurrierender Algorithmen beruht auf dem Zählen ihrer Schritte.
- Gesucht ist:
  Eine allgemeine Sprache für einen formalen Weg, die Zeitkomplexität konkurrierender Datenstrukturen und Algorithmen auszudrücken und zu vergleichen

Übung

1) Wir sagten, dass die Binäre Suche für ein sortiertes Array mit 100 Elementen 7 Schritte braucht. Wieviele Schritte braucht die Binäre Suche für ein sortiertes Array mit *200* Elementen?

A: 7

B: 8

C: 14

D: 107

E: 200

Übung

2) Auf einem Teich wächst eine Seerose. Mit jedem Tag verdoppelt sich die Anzahl der Seerosen. Am 100. Tag ist der Teich völlig damit bedeckt.
Frage: „Am wievielten Tag ist der Teich zur Hälfte mit Rosen bedeckt“?

A: am 2. Tag

B: am 32. Tag

C: am 50. Tag

D: am 64. Tag

E: am 99. Tag

Übung 1.1. – Frage 1

1. Berechnen Sie die Anzahl von Schritten, die die folgenden Operationen benötigen, für ein **unsortiertes Array** von 100 Elementen:

   - Lesezugriff

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. – Frage 2

2. Berechnen Sie die Anzahl von Schritten, die die folgenden Operationen benötigen, für ein **unsortiertes Array** von 100 Elementen:

   - Lineares Suchen nach einem Wert, der nicht Bestandteil des Arrays ist

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. – Frage 3

3. Berechnen Sie die Anzahl von Schritten, die die folgenden Operationen benötigen, für ein **unsortiertes Array** von 100 Elementen:

   - Einfügen am Beginn des Arrays

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. – Frage 4

4. Berechnen Sie die Anzahl von Schritten, die die folgenden Operationen benötigen, für ein **unsortiertes Array** von 100 Elementen:

   - Einfügen am Ende des Arrays

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. – Frage 5

5. Berechnen Sie die Anzahl von Schritten, die die folgenden Operationen benötigen, für ein **unsortiertes Array** von 100 Elementen:

   - Löschen am Beginn des Arrays

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. – Frage 6

6. Berechnen Sie die Anzahl von Schritten, die die folgenden Operationen benötigen, für ein **unsortiertes Array** von 100 Elementen:

   - Löschen am Ende des Arrays

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. –Frage 2.1

1. Berechnen Sie die Anzahl von Schritten, die für die folgenden Operationen für ein Array-basiertes Set von 100 Elementen nötig sind:

   - Lesezugriff

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. –Frage 2.2

2. Berechnen Sie die Anzahl von Schritten, die für die folgenden Operationen für ein Array-basiertes Set von 100 Elementen nötig sind:

   Suchen nach einem Wert, der nicht Bestandteil des Sets ist

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. – Frage 2.3

3. Berechnen Sie die Anzahl von Schritten, die für die folgenden Operationen für ein Array-basiertes Set von 100 Elementen nötig sind:

   - Einfügen am Beginn des Sets (der Wert ist noch nicht Bestandteil des Sets).

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. – Frage 2.4

4. Berechnen Sie die Anzahl von Schritten, die für die folgenden Operationen für ein Array-basiertes Set von 100 Elementen nötig sind:

   - Einfügen am Ende des Sets (der Wert ist noch nicht Bestandteil des Sets).

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. – Frage 2.5

5. Berechnen Sie die Anzahl von Schritten, die für die folgenden Operationen für ein Array-basiertes Set von 100 Elementen nötig sind:

   - Löschen am Beginn des Sets

A: 1
B: 100
C: 101
D: 200
E: 201

Übung 1.1. – Frage 2.6

6. Berechnen Sie die Anzahl von Schritten, die für die folgenden Operationen für ein Array-basiertes Set von 100 Elementen nötig sind:

   - Löschen am Ende des Sets

A: 1
B: 100
C: 101
D: 200
E: 201