Suchalgorithmen

Einführung in die Programmierung 1

Sommersemester 21

TU Wien

Überblick

- Lineare Suche
- Binäre Suche

Lineare Suche

Allgemeine Annahmen

- Allgemein
  - Daten werden in einem Array gespeichert
- Daten
  - Alle Beispiele mit ganzen Zahlen
  - Die präsentierten Algorithmen funktionieren grundsätzlich auch auf anderen Datentypen
    - Z. B. mit kleinen Änderungen bei Vergleichen
- Implementierung in Methoden
  - Es werden immer existierende Arrays übergeben
  - Es wird daher nicht explizit auf `null` überprüft

Lineare Suche

- Suche nach einem Wert in einem unsortierten Array der Länge *n*
- Rückgabe
  - Position im Array (Index), an der der Wert gefunden wurde
  - Ungültiger Index, wenn nicht gefunden
    - Z. B. -1
    - Kann als Konstante festgelegt werden

Implementierung

```
private static int search(int[] data, int key) {
    for (int i = 0; i < data.length; i++) {
        if (data[i] == key) {
            return i;
        }
    }
    return NOT_FOUND;
}
```

Konstante, z. B.

```
private static final int NOT_FOUND = -1;
```

Unterschiedliche Fälle (Wiederholung)

Bester Fall
(Best Case)

- Gesuchter Wert befindet sich an erster Position im Array, d.h. nur ein Element muss untersucht werden.

Durchschnittlicher Fall
(Average Case)

- Bei Gleichverteilung ist der gesuchte Wert mit jeweils gleicher Wahrscheinlichkeit an jeder möglichen Stelle im Array.
- Im Durchschnitt werden ca. $n/2$ Elemente untersucht.

Schlechtester Fall
(Worst Case)

- Gesuchter Wert befindet sich an letzter Position bzw. nicht im Array, d.h. alle n Elemente müssen untersucht werden.

Analyse

- Praxis
  - Es wird vom Worst Case ausgegangen
    - Da dieser bei der Suche sehr oft auftreten kann (Suche nach nicht vorhandenen Werten)
  - Laufzeit liegt in **O(n)**

```
private static int search(int[] data, int key) {
    for (int i = 0; i < data.length; i++) {
        if (data[i] == key) {
            return i;
        }
    }
    return NOT_FOUND;
}
```

Eine Schleifen über alle n Elemente des Arrays – $O(n)$
Wichtig:
- Die Laufzeiten aller Anweisungen (Vergleich, return-Anweisung) in der Schleife liegen jeweils in $O(1)$.
- Die return-Anweisung am Ende ist auch in $O(1)$.

Rekursive Variante

```
private static int searchRecursive(int[] data, int key) {
    return searchRecursive(data, 0, key);
}

private static int searchRecursive(int[] data, int n, int key) {
    if (n == data.length) {
        return NOT_FOUND;
    } else if (data[n] == key) {
        return n;
    } else {
        return searchRecursive(data, n + 1, key);
    }
}
```

- Eigenschaften
  - Laufzeit in **O(n)**
  - StackOverflow bei großen Arrays

Binäre Suche

Binäre Suche

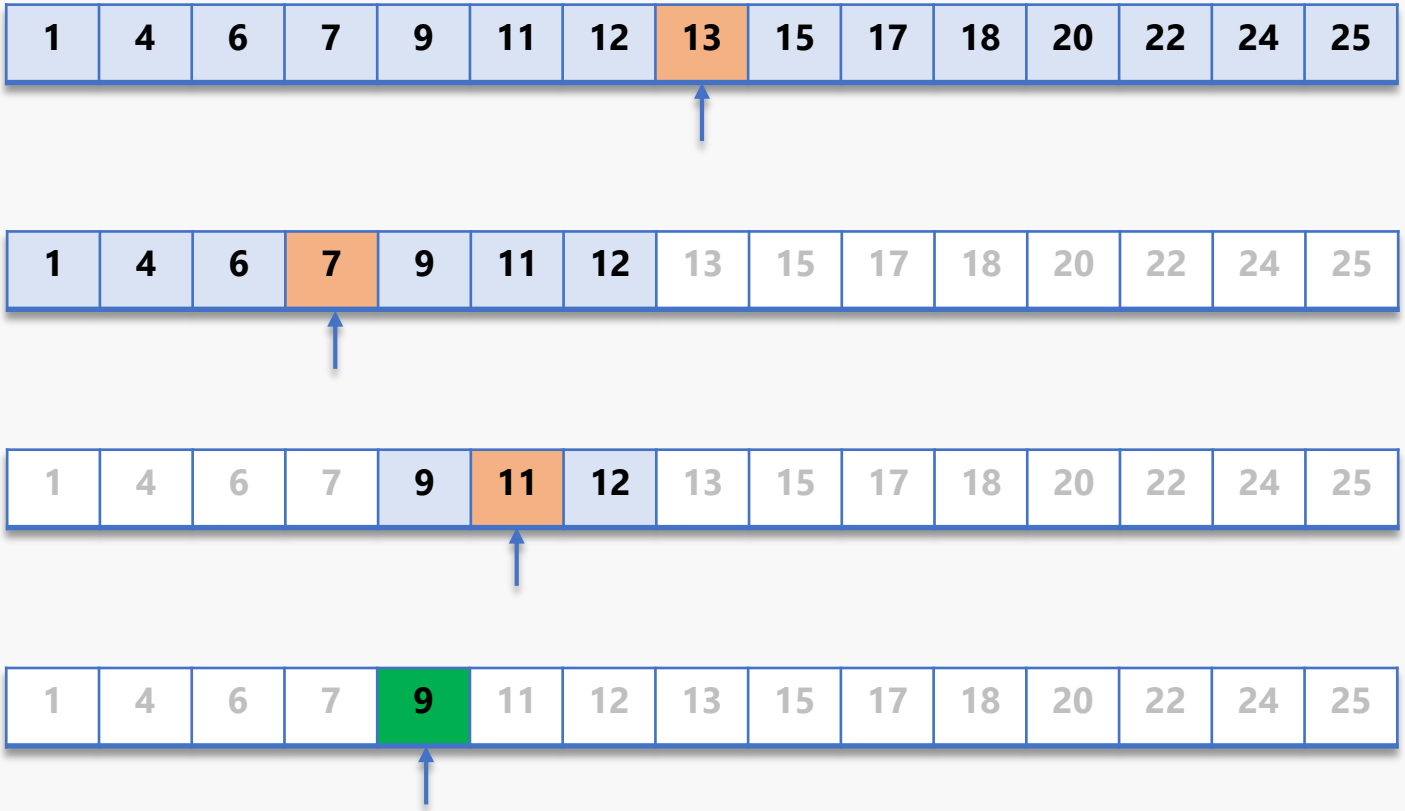
- Suche nach einem Wert in einem sortierten Array der Länge *n*
  - Lineare Suche möglich, aber ineffizient
  - Sortierung ausnutzen
- Rückgabe
  - Position im Array (Index), an der der Wert gefunden wurde
  - Ungültiger Index, wenn nicht gefunden
    - Z. B. -1
    - Kann als Konstante festgelegt werden

Idee

- Überprüfe das Element in der Mitte des sortierten Arrays
  - Ist es gleich dem gesuchten Element, ist die Suche beendet
  - Ist es kleiner als das gesuchte Element
    - Suche in der rechten Hälfte weiter und ignoriere die linke Hälfte
  - Ist es größer als das gesuchte Element
    - Suche in der linken Hälfte weiter und ignoriere die rechte Hälfte
- In der zu untersuchenden Hälfte (und erneut in den folgenden Hälften usw.) wird genauso verfahren
  - Die Länge des Suchbereiches wird von Schritt zu Schritt halbiert
  - Die Suche endet, wenn der Suchbereich nur mehr aus einem Element besteht

Beispiel

- 9 im Array?



Implementierung

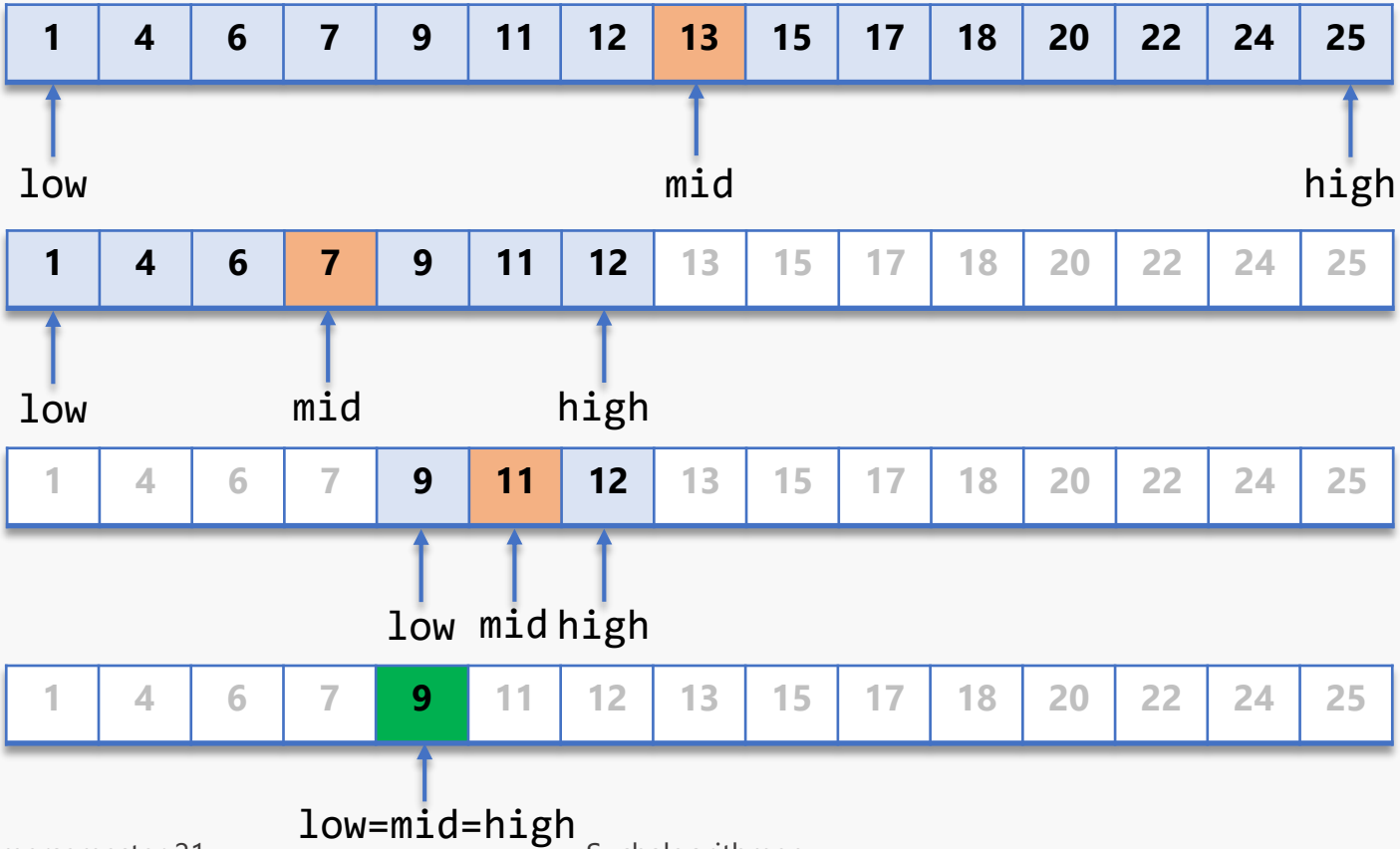
```
private static int binarySearch(int[] data, int key) {
    int low = 0, high = data.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        int value = data[mid];
        if (value < key) {
            low = mid + 1;
        } else if (value > key) {
            high = mid - 1;
        } else {
            return mid;
        }
    }
    return NOT_FOUND;
}
```

Vermeidet Überlauf

Konstante, z. B. -1

Beispiel mit Variablen

- 9 im Array?



Analyse

- Bester Fall
  - Das Element wird im ersten Durchlauf (in der Mitte des Arrays) gefunden
- Schlechtester Fall
  - Bis zu einem Bereich der Größe 1 suchen
  - Die Schleife wird dann bei *n* Elementen $\lfloor log_2(n) + 1 \rfloor$ - mal durchlaufen
    - Alle Anweisungen in der Schleife liegen in O(1)
  - Im schlechtesten Fall daher **O(log n)**
- Durchschnittlicher Fall
  - Komplizierter zu analysieren aber auch **O(log n)**

Anzahl der Schleifendurchläufe

- Beispiele für schlechtesten Fall

n	Anzahl der Durchläufe
1	1
10	4
100	7
1000	10
10000	14
100000	17
1000000	20
10000000	24
100000000	27
1000000000	30

Typische Implementierungsfehler

- Offensichtliche Variante für die Berechnung der Mitte
  - `int mid = (low + high)/2;`
  - Kann Überlauf produzieren
  - Bekannter Fehler
    - Einige Jahre auch in der Java-Bibliothek
- Grenzen des Suchbereichs
  - `mid` statt `mid + 1` bzw. `mid – 1` wählen
  - Kann zu Endlosschleifen führen
- Schleifenbedingung
  - Statt `low <= high` nur `low < high`
  - Es werden dann nicht alle Einträge gefunden

Rekursive Variante

```
private static int binarySearchRecursive(int[] data, int key) {
    return binarySearchRecursive(data, key, 0, data.length - 1);
}

private static int binarySearchRecursive(int[] data, int key, int low, int high) {
    if (low > high) {
        return NOT_FOUND;
    }
    int mid = low + (high - low) / 2;
    int value = data[mid];
    if (value < key) {
        return binarySearchRecursive(data, key, mid + 1, high);
    } else if (value > key) {
        return binarySearchRecursive(data, key, low, mid - 1);
    } else {
        return mid;
    }
}
```

- Laufzeit im schlechtesten Fall wiederum in **O(log n)**

Iterative Version für Strings

```
private static int binarySearch(String[] data, String key) {
    int low = 0, high = data.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        int comp = data[mid].compareTo(key);
        if (comp < 0) {
            low = mid + 1;
        } else if (comp > 0) {
            high = mid - 1;
        } else {
            return mid;
        }
    }
    return NOT_FOUND;
}
```

Lexikographischer Vergleich der Form
`String1.compareTo(String2)`
Rückgabe:

- 0 - beide Strings (Inhalt) gleich
- < 0 - String1 kleiner als String2
- > 0 - String1 größer als String2

Java-API (int-Version in Klasse Arrays)

```java
public static int binarySearch(int[] a, int fromIndex, int toIndex, int key) {
    rangeCheck(a.length, fromIndex, toIndex);
    return binarySearch0(a, fromIndex, toIndex, key);
}

// Like public version, but without range checks.
private static int binarySearch0(int[] a, int fromIndex, int toIndex, int key) {
    int low = fromIndex;
    int high = toIndex - 1;

    while (low <= high) {
        int mid = (low + high) >>> 1;
        int midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else
            return mid; // key found
    }
    return -(low + 1);  // key not found.
}
```

Öffentliche Methode für den Aufruf (mit Indexangabe)

Auch wenn die Addition einen Überlauf erzeugt, ist das Ergebnis nach dem logischen Shift positiv!

Java-API (float-Version in Klasse Arrays)

```
// Like public version, but without range checks.
private static int binarySearch0(float[] a, int fromIndex, int toIndex, float key) {
    int low = fromIndex;
    int high = toIndex - 1;

    while (low <= high) {
        int mid = (low + high) >>> 1;
        float midVal = a[mid];

        if (midVal < key)
            low = mid + 1;
        else if (midVal > key)
            high = mid - 1;
        else {
            int midBits = Float.floatToIntBits(midVal);
            int keyBits = Float.floatToIntBits(key);
            if (midBits == keyBits)     // Values are equal
                return mid;             // Key found
            else if (midBits < keyBits) // (-0.0, 0.0) or (!NaN, NaN)
                low = mid + 1;
            else                        // (0.0, -0.0) or (NaN, !NaN)
                high = mid - 1;
        }
    }
    return -(low + 1);  // key not found.
}
```

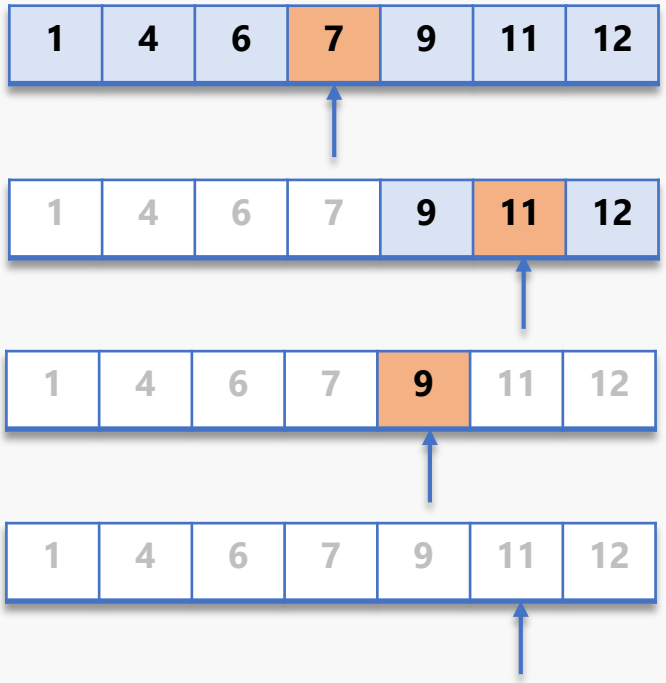
Besonderheiten bei Gleitkommazahlen berücksichtigen - siehe auch:
https://docs.oracle.com/javase/specs/jls/se14/html/jls-4.html#jls-4.2.3

Rückgabewert in der Java-API (1)

- Berechnung des Rückgabewerts, wenn nicht gefunden
  - `-(low + 1)`
- Idee
  - Negativer Wert zeigt an, dass der gesuchte Wert nicht gefunden wurde
  - Aus dem Rückgabewert kann die Stelle rekonstruiert werden, an der der Wert in der sortierten Reihenfolge stehen würde

Rückgabewert in der Java-API (2)

- Beispiel für erfolglose Suche nach Wert 10



low = 0, high = 6

low = 4, high = 6

low = 4, high = 4

low = 5, high = 4

Rückgabewert = -(low + 1) = -6.
Absolutbetrag des Rückgabewerts - 1 (in Beispiel 5) entspricht der Position, an der 10 im Array stehen würde.

Vergleich (lineare Suche – binäre Suche)

- Sortierung
  - Lineare Suche funktioniert auf unsortierte und sortierte Arrays
  - Binäre Suche funktioniert nur auf sortierte Arrays
- Laufzeit
  - Lineare Suche in O(n)
  - Binäre Suche in O(log n)
    - Erfordert ein sortiertes Array
- Gesuchter Wert kommt mehrfach im Array vor
  - Lineare Suche findet den Array-Eintrag mit dem kleinsten Index
  - Binäre Suche findet einen Array-Eintrag (von mehreren) mit dem gesuchten Wert

Wann binäre Suche der linearen Suche vorziehen?

- Wenn die Daten schon in sortierter Reihenfolge vorliegen
- Wenn die Daten nur einmal sortiert werden müssen aber die Suche sehr oft durchgeführt wird
  - Die Kosten für die Sortierung (SORT) treten nur einmal auf
  - Die Kosten für das Suchen k mal (k eine entsprechend große Zahl)
  - D. h. binäre Suche verwenden, wenn
    - SORT + k * O(log n) < k * O(n) (binär, linear)
    - Für große k wird der Aufwand für die Sortierung (SORT) irrelevant (setzt schnelle Sortierung voraus)