Einführung in die Informatik

Organizing Objects

Indizierungen, Suchen, Aufwandsanalyse, Sortieren

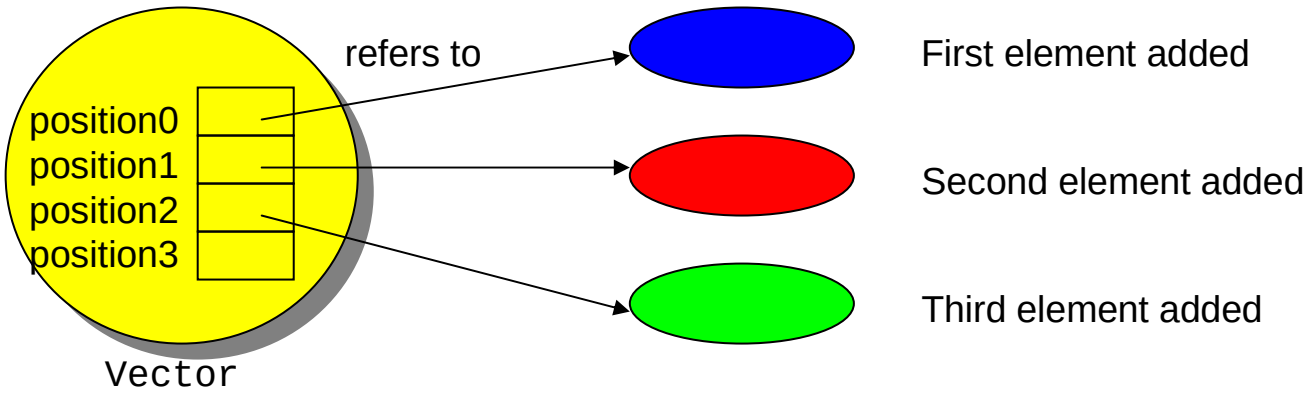
Wolfram Burgard
Cyrill Stachniss

Motivation

- Bisher haben wir Objekte in **Kollektionen** zusammengefasst und **Enumerations** verwendet, um Durchläufe zu realisieren.
- **Enumerations** lassen jedoch offen, wie die Objekte in der Kollektion angeordnet sind und in welcher Reihenfolge sie ausgegeben werden.
- In diesem Kapitel werden wir Möglichkeiten kennenlernen, **Ordnungen auf den Objekten** auszunutzen.
- Darüber werden wir **Aufwandsanalysen** kennenlernen, die es uns z.B. erlauben, die von Programmen benötigte Rechenzeit zu charakterisieren.
- Schließlich werden wir auch diskutieren, wie man **Objekte sortieren** kann.

Indizierung

- Ein `Vector`-Objekt stellt nicht nur eine Kollektion von Objekten dar.
- Die in einem `Vector`-Objekt abgelegten Objekte sind entsprechend ihrer Einfügung angeordnet: Das erste Objekt befindet sich an Position 0. Die weiteren folgen an den Positionen 1, 2, . . .
- Die Nummerierung der Positionen von Objekten heißt **Indizierung** oder **Indexing**.
- Ein **Index** ist die **Position** eines Objektes in einer **Kollektion**.



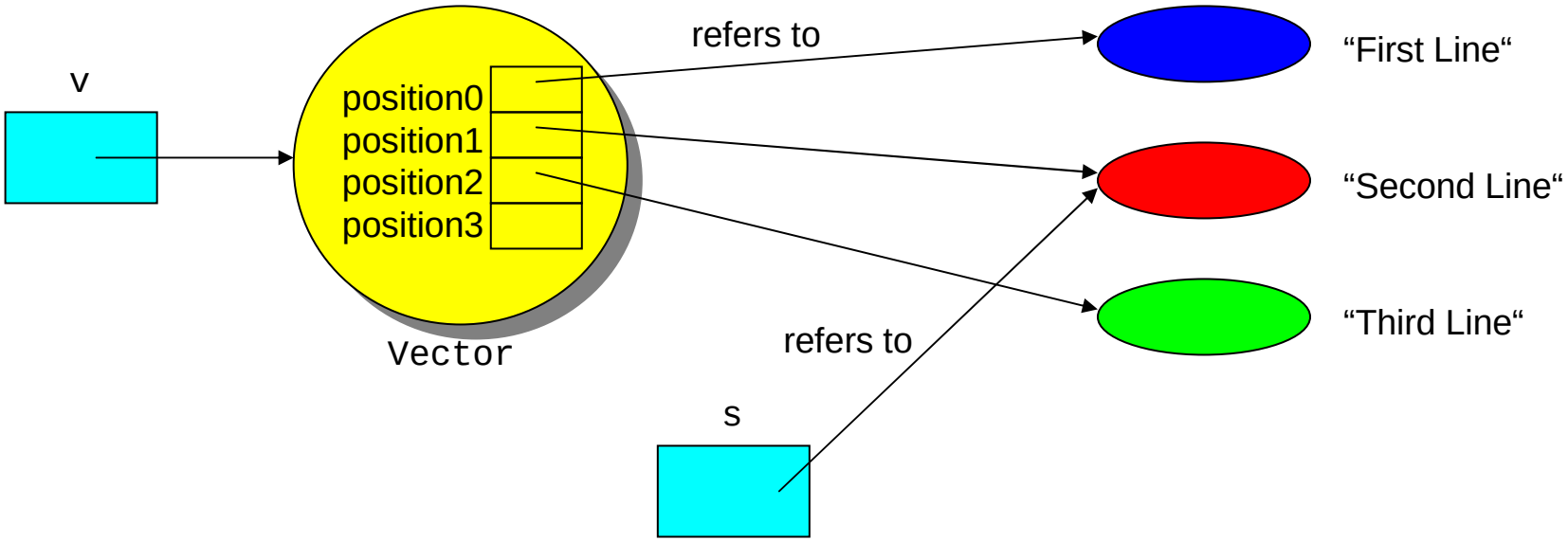
Die Methode `elementAt` für den Zugriff auf ein Objekt an einer Position

- Um auf die an einer bestimmten Position gespeicherte Referenz eines `Vector`-Objektes zu erhalten, verwenden wir die Methode **`elementAt`**:

```
Vector v = new Vector();
v.addElement("First line");
v.addElement("Second line");
v.addElement("Third line");
String s = (String) v.elementAt(1);
System.out.println(s);
```

Dieses Programm druckt den Text `Second line`.

Situation nach der vierten Anweisung



- Die Position des ersten Objektes ist 0.
- Entsprechend ist die Position des letzten Objektes `v.size()-1`.

Grenzen von Enumerations

- **Enumerations** stellen eine komfortable Möglichkeit dar, Durchläufe durch Kollektionen zu realisieren.
- Allerdings haben wir keinen Einfluss darauf, in welcher Reihenfolge die Kollektionen durchlaufen werden.
- Ein typisches Beispiel ist das Invertieren eines Textes, bei dem die eingelesenen Zeilen in umgekehrter Reihenfolge ausgegeben werden sollen.
- In diesem Fall ist die Reihenfolge, in der die Objekte durchlaufen werden, relevant.
- Ein weiteres Beispiel ist das Sortieren von Objekten.

Anwendung: Eingelesene Zeilen in umgekehrter Reihenfolge ausgeben

Prinzip:

1. Einlesen der Zeilen in ein `Vector`-Objekt.
2. Ausgeben der Zeilen „von hinten nach vorne“.

Einlesen der Zeilen in ein `Vector`-Objekt:

```
BufferedReader br = new BufferedReader(
                        new InputStreamReader(System.in));
Vector         v = new Vector();
String         line;
line = br.readLine();
while (line != null) {
    v.addElement(line);
    line = br.readLine();
}
```

Ausgeben in umgekehrter Reihenfolge

1. Wir starten mit dem letzten Objekt, welches sich an Position `k == v.size()-1` befindet.
2. Wir geben das Objekt an Position `k` aus und gehen zu dem Objekt an der Position davor (`k = k-1`).
3. Schritt 2) wiederholen wir solange, bis wir das erste Objekt ausgegeben haben. In diesem Fall muss gelten: `k ==-1`.

Dies ergibt folgenden Code:

```
int k = v.size()-1;
while (k != -1) {
    System.out.println(v.elementAt(k));
    --k;
}
```

Das komplette Programm

```
import java.io.*;
import java.util.*;

class ReverseLines {
    public static void main(String[] a) throws Exception {
        BufferedReader br = new BufferedReader(
                                    new InputStreamReader(System.in));
        Vector<String> v = new Vector<String> ();
        String line;
        line = br.readLine();
        while (line != null) {
            v.addElement(line);
            line = br.readLine();
        }
        int k = v.size()-1;
        while (k != -1) {
            System.out.println(v.elementAt(k));
            --k;
        }
    }
}
```

Enumerations versus Indizierung

- Ähnlich wie mit einem `Enumeration`-Objekt können wir auch mit der Methode `elementAt` eine Aufzählung realisieren.
- Dabei haben wir sogar noch die Flexibilität, die Reihenfolge, in der die Elemente prozessiert werden, festzulegen.
- Was sind die Vorteile einer `Enumeration`?
  - In vielen Fällen ist die Reihenfolge, in der die Objekte prozessiert werden, irrelevant.
  - Die **Verwendung einer Enumeration ist deutlich weniger fehleranfällig als die Verwendung von `elementAt`.**
  - Ein typischer Fehler bei der Verwendung von Indizes ist der **Zugriff auf nicht existierende Objekte**. Weiter passiert es auch häufig, dass **nicht auf alle Objekte zugegriffen** wird. Ursache sind in der Regel falsche Abbruchbedingungen.

Suche in `Vector`-Objekten

- Das **Suchen in Kollektionen nach Objekten** mit bestimmten Eigenschaften ist eine der typischen Aufgaben von Programmen.
- In diesem Kapitel befassen wir uns mit der Suche von Objekten, die mit einem gegebenen Objekt übereinstimmen.
- Dabei soll die **Suchprozedur den Index** des gefundenen Objektes in einem `Vector`-Objektes **zurückgeben**, sofern es in dem `Vector` enthalten ist und `-1` sonst.
- Wir unterscheiden **zwei Fälle**:
    1. Die **Objekte** sind im `Vector`-Objekt **ungeordnet** angeordnet.
    2. Die **Objekte** sind entsprechend ihrer Ordnung **angeordnet**.

Suche in ungeordneten `Vector`-Objekten

- Wenn das `Vector`-Objekt ungeordnet ist, müssen wir (ebenso wie bei der Methode `contains` der Klasse `Set`) den kompletten `Vector` durchlaufen.
- Wir starten mit dem Objekt an Position `0` und brechen ab, sobald wir das Ende des `Vectors` erreicht haben oder das Objekt mit dem gesuchten übereinstimmt.
- Die Bedingung der `while`-Schleife ist somit

```
while (!(k==v.size() || o.equals(v.elementAt(k))))
```

- Hierbei nutzen wir die bedingte Auswertung logischer Ausdrücke aus.

Die resultierende `while`-Schleife

```
public int linearSearch(Vector v, Object o) {
    int k =0;
    while (!(k==v.size() || o.equals(v.elementAt(k))))
        k++;
    // k==v.size || o.equals(v.elementAt(k))
    if (k==v.size())
        return -1;
    else
        return k;
}
```

Aufwandsanalysen

In der Informatik interessiert man sich nicht nur für die Korrektheit, sondern auch für die **Kosten von Verfahren**.

Hierbei gibt es prinzipiell verschiedene Kriterien:

- Wie hoch ist der **Programmieraufwand**?
- Wie hoch ist der **erforderliche Kenntnisstand eines Programmierers**?
- **Wie lange rechnet das Programm**?
- **Wieviel Speicherplatz** benötigt das Verfahren?

Wir werden uns im folgenden auf die **Rechenzeit** und den **Speicherplatzbedarf** beschränken.

Asymptotische Komplexität

Wir beschränken uns auf die so genannte **asymptotische Analyse des Aufwands** (oder der **Komplexität**).

Wir bestimmen die Rechenzeit und den Platzbedarf als eine **Funktion der Größe der Eingabe** (Anzahl der eingelesenen/verarbeiteten Elemente).

Dabei werden wir untersuchen, ob die Rechenzeitkurve logarithmisch, linear, quadratisch etc. ist, und dabei **konstante Faktoren außer Acht lassen**.

Die konstanten Faktoren hängen in der Regel von dem gegebenen Rechner und der verwendeten Programmiersprache ab, d.h. bei Verwendung einer anderen Sprache oder eines anderen Rechners bleibt die Kurvenform erhalten.

Zeitkomplexität

Um die Zeitkomplexität eines Verfahrens zu ermitteln, betrachten wir stets die Anzahl der durchzuführenden Operationen

1. **im schlechtesten Fall (Worst Case)**,
2. **im besten Fall (Best Case)** und
3. **im Durchschnitt (Average Case)**.

Komplexität der Suche in nicht geordneten `Vector`-Objekten (1)

Wir nehmen an, dass die einzelnen Operationen in unserer Suchprozedur jeweils konstante Zeit benötigen.

Dies schließt ein

- k_1 viele Operationen für die Initialisierung und das Beenden der Prozedur,
- k_2 viele Operationen für den Test der Bedingung und
- k_3 viele Operationen in jedem Schleifendurchlauf.

Der Worst Case und der Best Case

Worst Case: Die Schleife wird höchstens `n == v.size()` mal durchlaufen.

Somit benötigt die Ausführung von `linearSearch` höchstens $k_1 + n*(k_2 + k_3) + k_2$ Operationen (ein zusätzlicher Test am Ende, wenn das gesuchte Objekt nicht enthalten war).

Best Case: Ist das erste Element bereits das gesuchte oder der Vektor leer, so benötigen wir $k_1 + k_2$ Operationen, d.h. die Ausführungszeit ist unabhängig von n .

Der Average Case

Nehmen wir an, dass nach jedem Objekt gleich häufig gesucht wird (der Fall, dass das Objekt nicht enthalten ist bleibt hier unberücksichtigt).

Bei *n* Aufrufen wird somit nach jedem der *n* Objekte im `Vector`-Objekt genau einmal gesucht.

Dies ergibt die folgenden, durchschnittlichen Aufwand (n>=1):

$$k_1 + \frac{1+2+...+(n-1)}{n} * (k_2 + k_3) + k_2$$

$$= k_1 + k_2 + \frac{(n-1)*n}{n*2} * (k_2 + k_3)$$

$$= k_1 + k_2 + (k_2 + k_3) * \frac{n-1}{2}$$

Da diese Funktion linear in *n* ist, sagen wir, dass die durchschnittliche, **asymptotische Komplexität** von `linearSearch` **linear in der Anzahl der Elemente ist** .

Die O-Notation für asymptotische Komplexität

Anstatt die Komplexität eines Verfahrens exakt zu berechnen, wollen wir im Folgenden eher die Ordnung der Komplexität bestimmen.

Im Prinzip bestimmen wir das Wachstum der Funktion, welches die Komplexität des Verfahrens beschreibt, und abstrahieren von möglichen Konstanten.

Definition: *Sei* $f : N \to R_0^+$. *Die* ***Ordnung*** *von* f *ist die Menge*

$$O(f) = \left\{ g : N \to R_0^+ \,\middle|\, \exists c \in R^+ \exists n_0 \in N \forall n \geq n_0 : g(n) \leq c \cdot f(n) \right\}$$

Informell: Ist $g \in O(f)$,so wächst *g* (bis auf konstante Faktoren) höchstens so wie f.

Beispiele

1. $g(n) = 10 * n + 3$ ist in O(n) :

$$10 * n + 3 \leq 13 * n$$
$$\Leftrightarrow 3 \leq 3 * n$$
$$\Leftrightarrow 1 \leq n$$

Wir wählen $n_0 = 1$ und $c = 13$.

2. $g(n) = 2n^2 + 3n + 10$ ist in $O(n^2)$, denn für $n_0 = 1$ und $n \geq n_0$ gilt:

$$2n^2 + 3n + 10 \leq 2n^2 + 3n^2 + 10n^2 = 15n^2$$

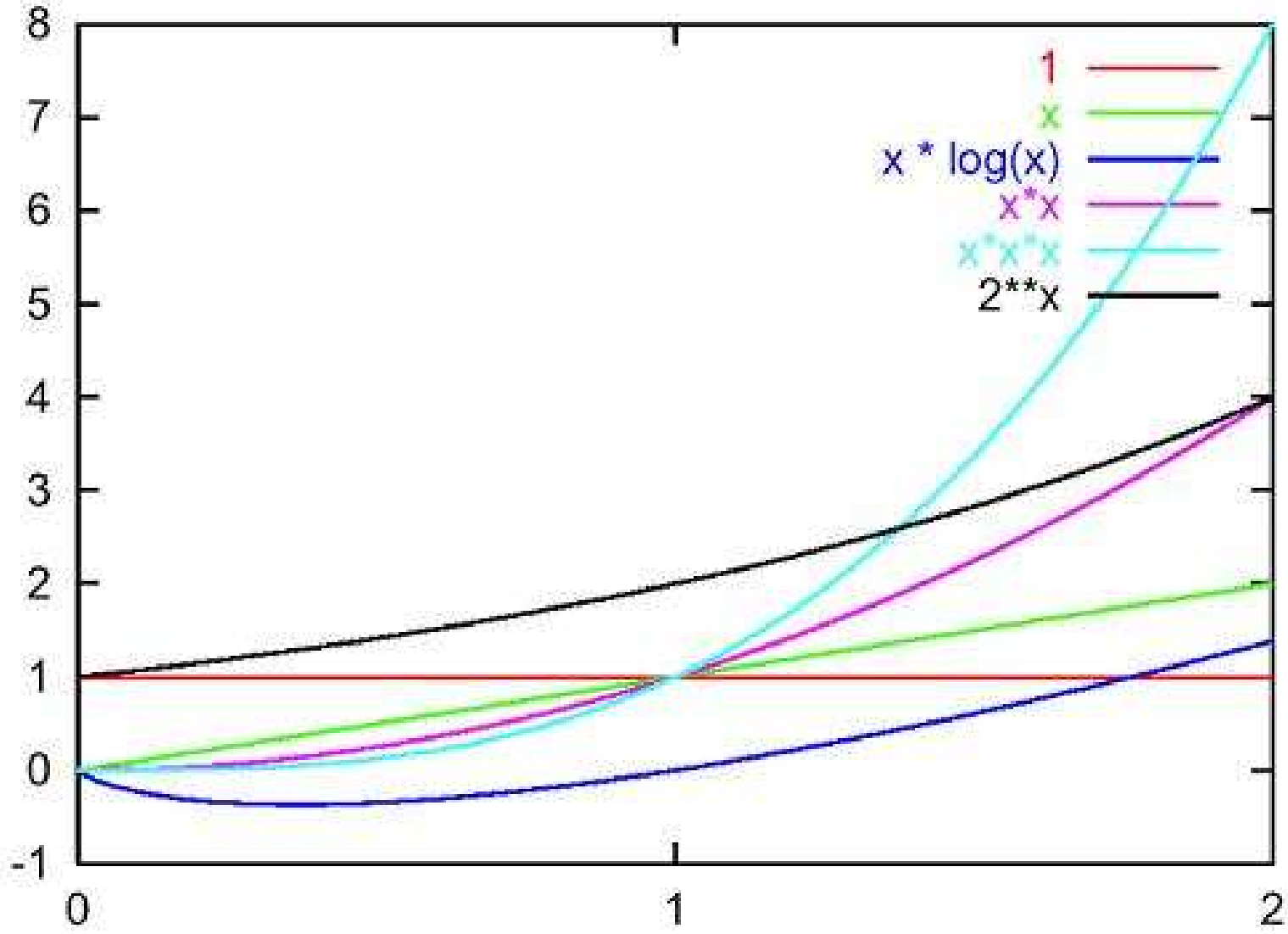
Somit wählen wir $c = 15$;

3. Entsprechend gilt, dass ein Polynom vom Grad k in $O(n^k)$ liegt.

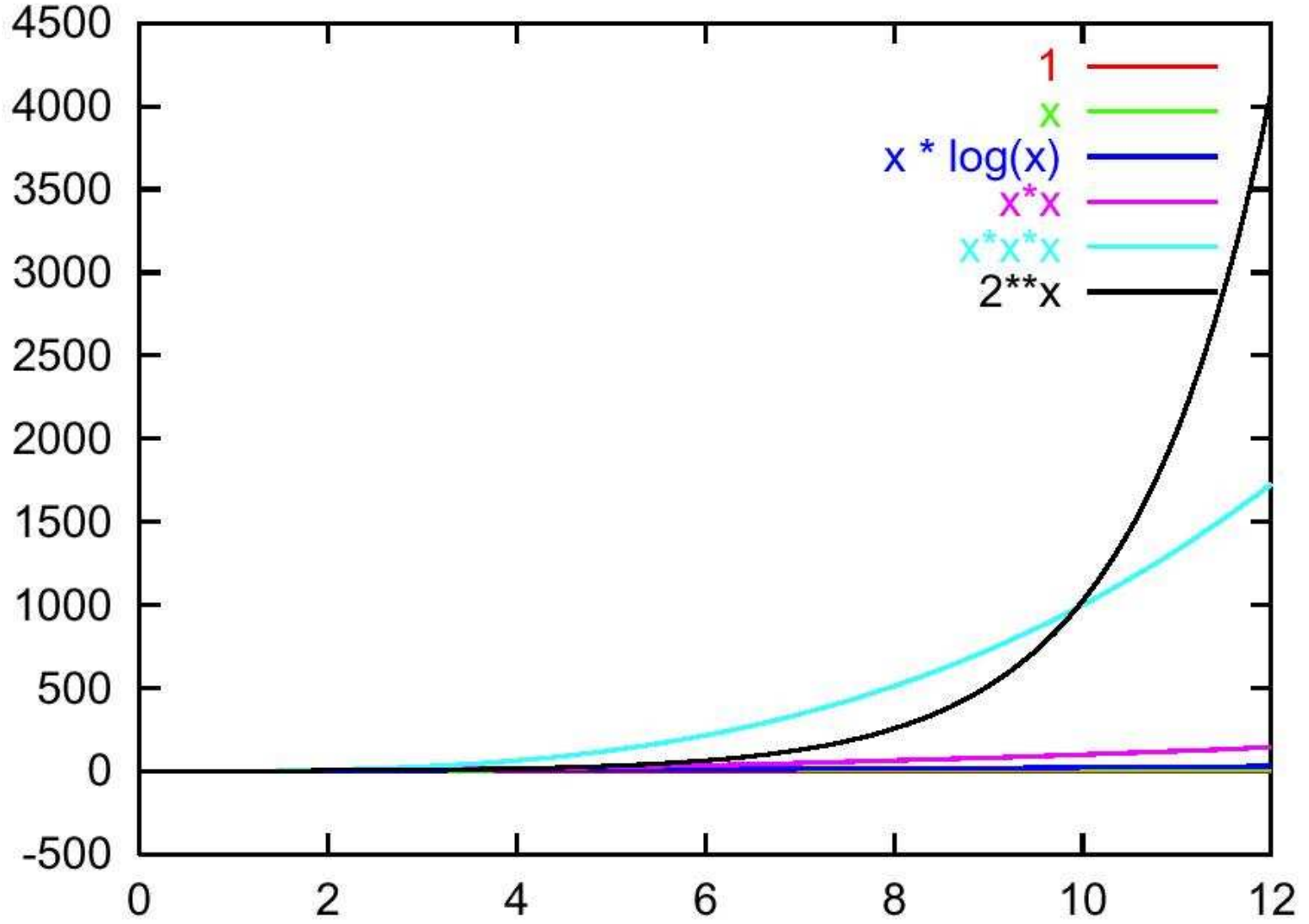
Typische Wachstumskurven

Klasse	Bezeichnug
$\mathcal{O}(1)$	konstant
$\mathcal{O}(\log n)$	logarithmisch
$\mathcal{O}(n)$	linear
$\mathcal{O}(n \log n)$	$n \log n$
$\mathcal{O}(n^2)$	quadratisch
$\mathcal{O}(n^3)$	kubisch
$\mathcal{O}(n^k)$	polynomiell
$\mathcal{O}(2^n)$	exponentiell

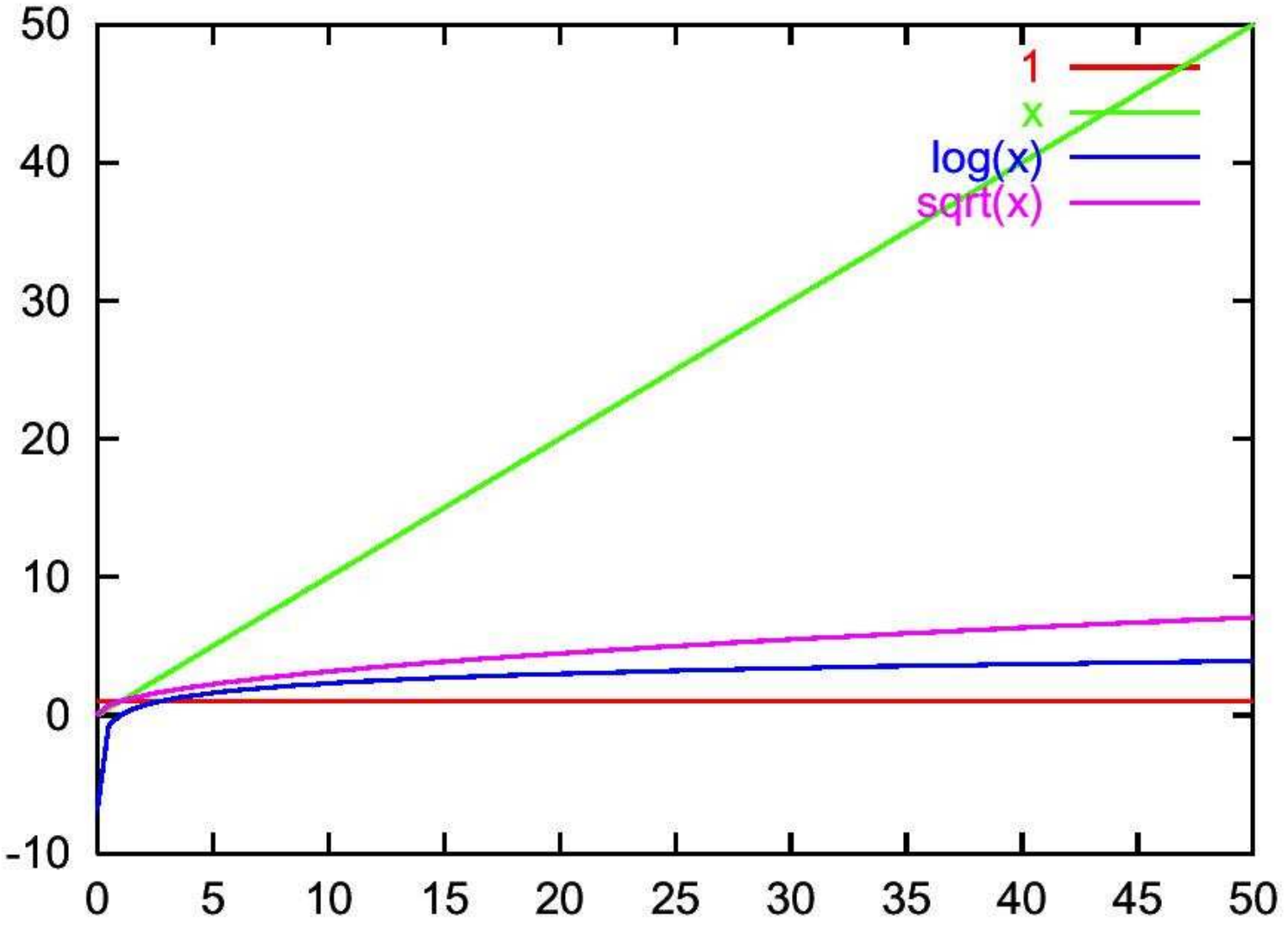
Verläufe von Kurven (1)



Verläufe von Kurven (2)



Typische Wachstumskurven (3)



Methoden der Klasse `Vector` und ihre Effizienz

Bisher haben wir verschiedene Methoden für Vektoren kennengelernt:

`size()`	Liefert Anzahl der Elemente in einem `Vector`-Objekt.
`elementAt(int n)`	Liefert eine Referenz auf das an Position n gespeicherte Objekt.
`insertElementAt(Object o, int n)`	Fügt die Referenz auf Objekt o an Position n ein. Vorher wird die Position der Objekte an Position jeweils um eins erhöht.
`removeElementAt(int n)`	Löscht das Objekt an Position n.

Aufwand von `size`

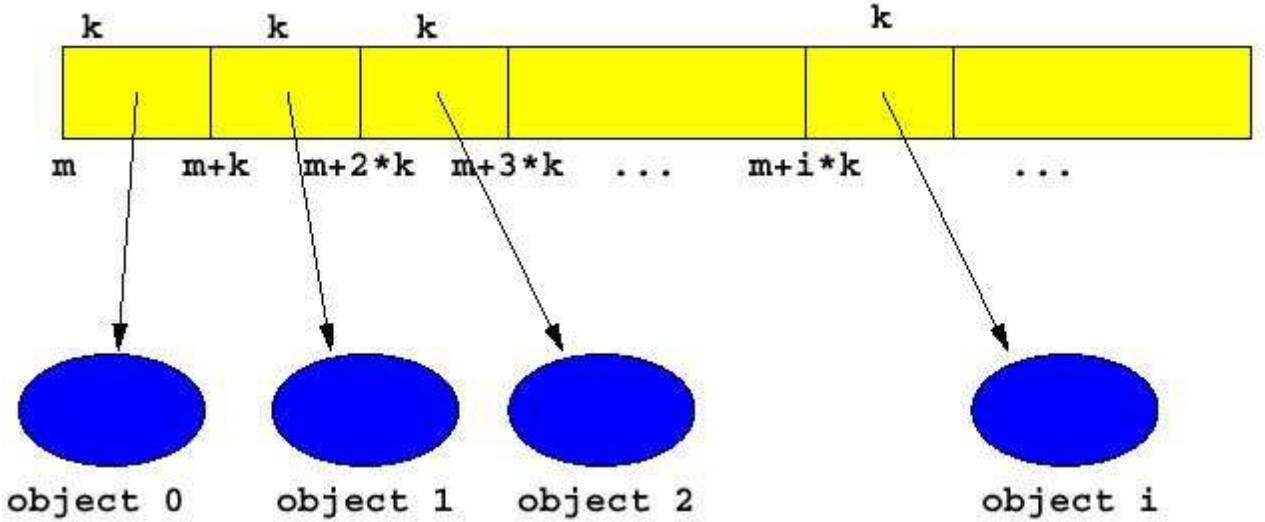
- Die Methode `size` kann sehr einfach umgesetzt werden, wenn man man die Anzahl der Elemente in einem `Vector` in einer Instanzvariablen ablegt.
- Dieser Zähler muss dann natürlich bei jeder Einfüge- oder Löschoperation inkrementiert oder dekrementiert werden.
- Wenn man so vorgeht und die Anzahl der Elemente beispielsweise in einer Variablen `numberOfElements` ablegt, so kann die Methode `size` folgendermaßen realisiert werden:

```
public int size(){
    return this.numberOfElements;
}
```

- Da diese Funktion konstante Zeit benötigt, ist die Komplexität von `size` in $O(1)$.

Komplexität von `elementAt`

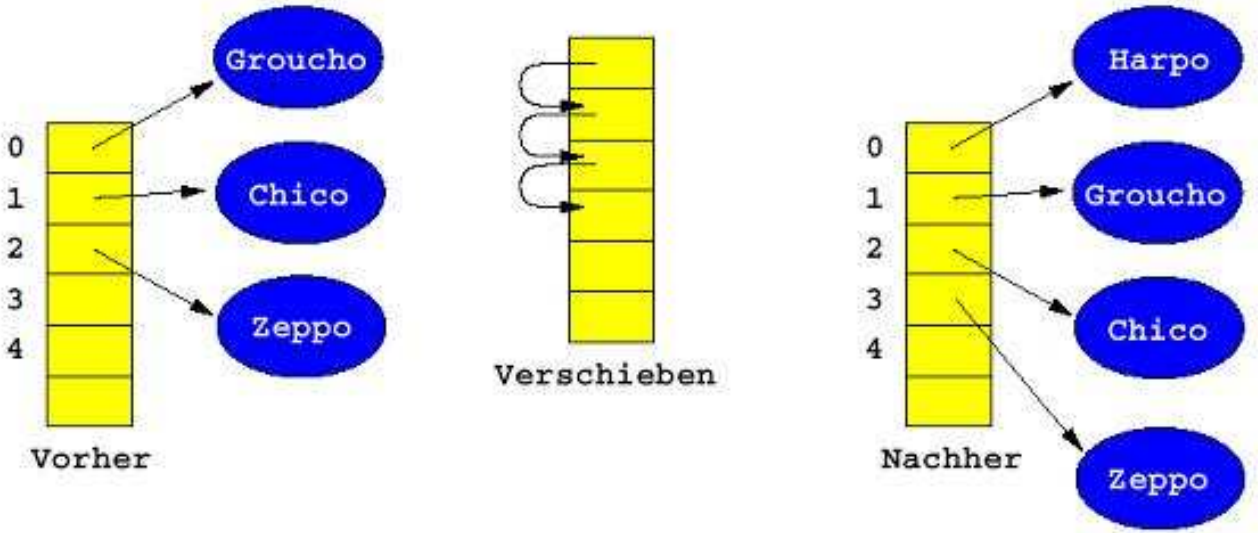
- Üblicherweise werden Vektoren durch so genannte **Felder** realisiert.
- Ein Feld wiederum lässt sich vergleichen mit einer Tabelle oder einer Reihe von durchnummerierten Plätzen, in der die Objekte hintereinander angeordnet sind.
- Benötigt man nun für eine eine Referenzvariable *k* Bytes und kennt man die Stelle *m* im Programmspeicher, an der sich das erste Objektes der Tabelle befindet, so kann man die Position des *i*-ten Objektes mit der Formel *m + k * i* in *O*(*1*) (d.h. in konstanter Zeit) berechnen.



Die Kosten für das Einfügen

Wenn wir ein Objekt an einer Position `0` einfügen wollen, dann müssen wir alle Objekte in dem `Vector` um jeweils eine Position nach hinten verschieben.

Die Einfügeoperation `insertElementAt("Harpo",0)` geht demnach folgendermaßen vor:



Da im Durchschnitt größenordnungsmäßig *n/2* Objekte verschoben werden müssen, ist die durchschnittliche Laufzeit *O*(*n*).

Das Entfernen eines Objektes

- Beim Entfernen eines Objektes aus einem `Vector`-Objekt geht Java umgekehrt vor.
- Wenn wir ein Objekt mit `removeElementAt(i)` an der Position `i` löschen, rücken alle Objekte an den Positionen `i+1` bis `v.size()-1` jeweils um einen Platz nach vorne.
- D.h., wir müssen `v.size()-i-1` Verschiebungen durchführen.
- Ist `n == v.size()` so ist die durchschnittliche Laufzeit von `removeElementAt` ebenfalls in $O(n)$.

Folgerung: Werden in einem `Vector`-Objekt viele Vertauschungen von Objekten durchgeführt, sollte man ggf. alternative Methoden mit geringerem Aufwand verwenden.

Beispiel: Invertieren der Reihenfolge in einem Vektor

- Um die Reihenfolge aller Objekte in einem `Vector`-Objekt umzudrehen, müssen wir den `Vector` von `0` bis `v.size()/2` durchlaufen und jeweils die Objekte an den Position `i` und `v.size()-i-1` vertauschen.
- Verwenden wir für die Vertauschung die Methoden `removeElementAt` und `insertElementAt`, so benötigen wir folgenden Code:

```
Integer j1, j2;
j1 = (Integer) v.elementAt(i);
j2 = (Integer) v.elementAt(v.size()-i-1);
v.removeElementAt(i);
v.insertElementAt(j2, i);
v.insertElementAt(j1,v.size()-i-1);
v.removeElementAt(v.size()-i-1);
```

Invertieren mit `removeElementAt` und `insertElementAt`

```
import java.util.*;

class ReverseInsert {
    public static void main(String [] args) {

        Vector v = new Vector();
        int i;
        for (i = 0; i < 40000; i++)
            v.addElement(new Integer(i));

        for (i = 0; i < v.size() / 2; i++){
            Integer j1, j2;
            j1 = (Integer) v.elementAt(i);
            j2 = (Integer) v.elementAt(v.size()-i-1);
            v.removeElementAt(i);
            v.insertElementAt(j2, i);
            v.insertElementAt(j1,v.size()-i-1);
            v.removeElementAt(v.size()-i-1);
        }
    }
}
```

Aufwandsabschätzung für `ReverseInsert`

Sei `n == v.size()`. Falls $n > 0$ und n gerade, führen wir für $i=0,...,n/2$ jeweils vier Operationen aus. Dabei gilt:

- `removeElementAt(i)` benötigt $n-i-1$ Verschiebungen (danach enthält `v` noch $n-1$ Objekte).
- `insertElementAt(j2, i)` erfordert $n-i-1$ Verschiebungen.
- `insertElementAt(j1, v.size()-i-1)` erfordert $i+1$ Verschiebungen.
- `removeElementAt(v.size()-i-1)` benötigt i Verschiebungen.

Damit erhalten wir für jedes $i=0,...n/2$

$$(n - i - 1) + (n - i - 1) + (i + 1) + i = 2n - 1$$

notwendige Verschiebungen.

Bei $n/2$ Wiederholungen ergibt das $(n^2 - \frac{n}{2}) \in O(n^2)$ Verschiebungen.

Herleitung

`removeElementAt(i)`: $n - i - 1$ Verschiebungen.

`insertElementAt(j2, i)`: $n - i - 1$ Verschiebungen.

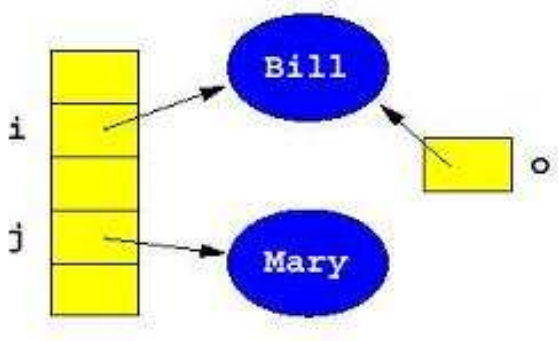
Eine effizientere Variante auf der Basis von `setElementAt`

Anstatt die Objekte aus dem Vektor zu entfernen und wieder neu einzufügen, können wir auch einfach deren Inhalte vertauschen.

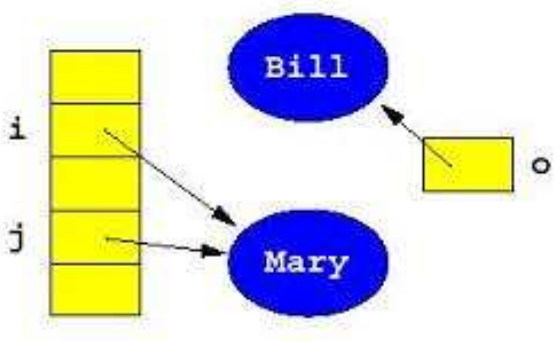
Hierzu verwenden wir die folgende Methode `swap`, welche in einem `Vector`-Objekt die Inhalte der Objekte an den Positionen `i` und `j` vertauscht.

```
static void swap(Vector v, int i, int j){
    Object o = v.elementAt(i);
    v.setElementAt(v.elementAt(j), i);
    v.setElementAt(o, j);
}
```

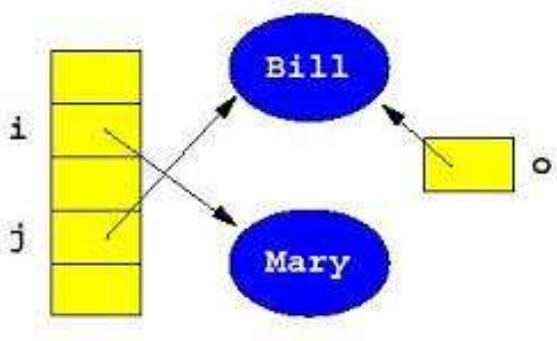
Wirkung der Methode swap



o = v.elementAt(i)



v.setElementAt(v.elementAt(j), i)



v.setElementAt(o, j)

Das komplette Beispielprogramm

```
import java.util.*;

class ReverseSwap {
    static void swap(Vector v, int i, int j){
        Object o = v.elementAt(i);
        v.setElementAt(v.elementAt(j), i);
        v.setElementAt(o, j);
    }

    public static void main(String [] args) {
        Vector<Integer> v = new Vector<Integer> ();
        int i;
        for (i = 0; i < 40000; i++)
            v.addElement(new Integer(i));

        for (i = 0; i < v.size() / 2; i++)
            swap(v, i, v.size()-i-1);
    }
}
```

Aufwandsabschätzung für `ReverseSwap`

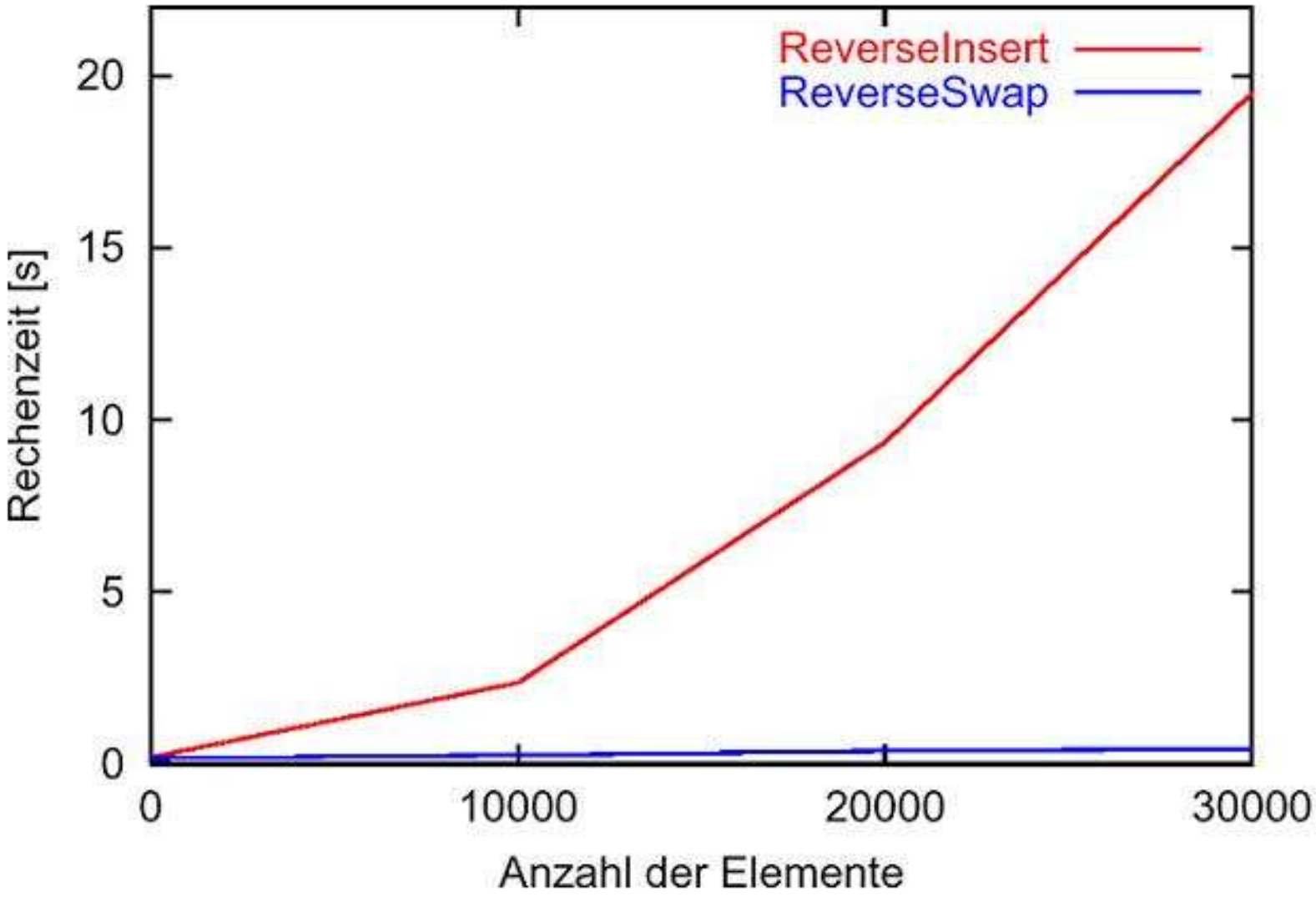
Nehmen wir erneut an, dass `n==v.size()` mit $n > 0$ und n gerade.

Pro Runde führen wir jetzt eine Vertauschung der Werte aus. Dies erfordert drei Wertzuweisungen.

Damit erhalten wir einen **Aufwand** von $\frac{3}{2} n \in O(n)$.

Schlussfolgerung: Die Variante auf der Basis von `setElementAt` ist deutlich effizienter als die Variante, die `removeElementAt` und `insertElementAt` verwendet.

Rechenzeit in Abhängigkeit von der Anzahl der Elemente



Effizientere Suche für sortierte `Vector`-Objekte

Wenn Vektoren sortiert sind, kann man schneller als mit `linearSearch` nach einem Objekt suchen.

Beispiel: Sie sollen eine ganze Zahl n zwischen 0 und 999 erraten, die Ihr Gegenüber sich ausgedacht hat. Sie dürfen Ihrem Gegenüber nur Fragen stellen, die er mit `true` oder `false` beantworten kann.

Eine schnelle Methode, die Zahl zu erraten, ist das wiederholte Halbieren des Suchintervalls.

Anwendungsbeispiel

Wir suchen eine Zahl in $[0 : 1000[$. Der Gegenüber hat 533 gewählt.

- Ist $n = 499$? Antwort: `false`
- Ist $n > 499$? Antwort: `true` $\rightarrow$ Zahl ist in $[500 : 1000[$.
- Ist $n = 749$? Antwort: `false`
- Ist $n > 749$? Antwort: `false` $\rightarrow$ Zahl ist in $[500 : 749[$.
- ...

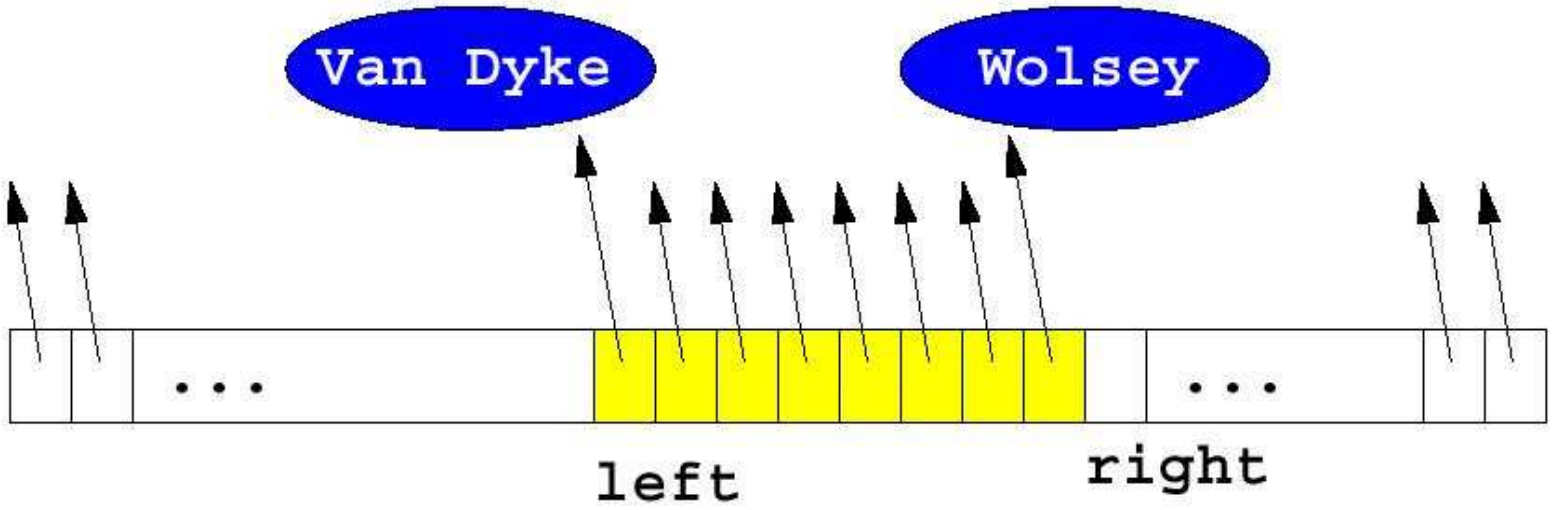
Informelle Prozedur

Zur Suche eines `String`-Objekts `s` in sortierten `Vector`-Objekten gehen wir analog vor.

- Wir verwenden zwei Positionen `left` und `right`, die das Suchintervall charakterisieren.
- In jedem Schritt betrachten wir das Element in der Mitte von `left` und `right`.
- Liegt das Element in der Mitte alphabetisch vor `s`, wird die Mitte des Intervalls plus 1 die neue linke Grenze `left`. Ist es größer als `s`, wird `right` auf die Mitte des Intervalls gesetzt.
- Ist das Element in der Mitte identisch mit `s`, ist die Mitte das Ergebnis. Ist irgendwann das Suchintervall leer, so ist `s` nicht in dem Vektor enthalten.

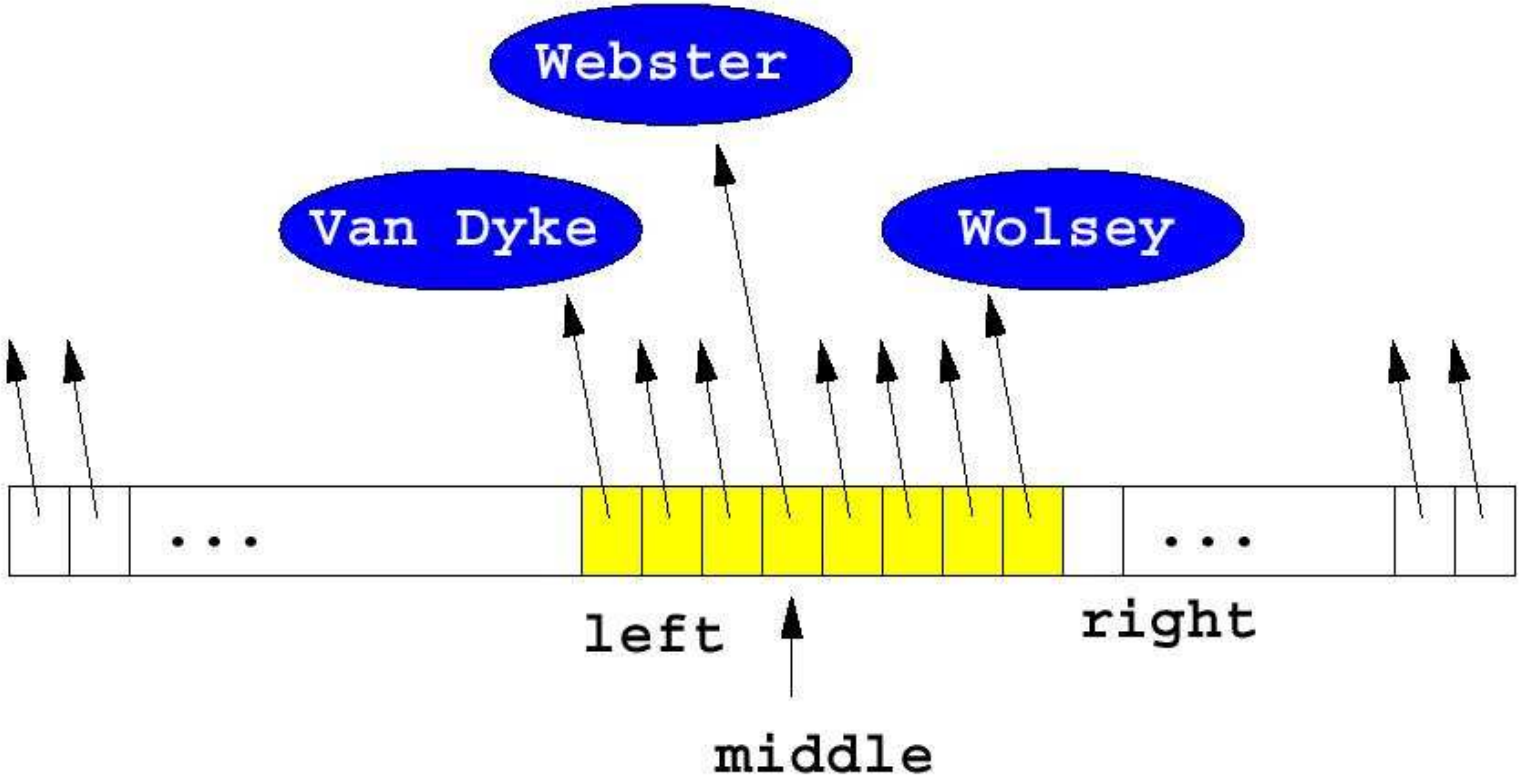
Prinzip der Binärsuche (1)

- Angenommen unser Suchstring `s` ist `"Wolfram"`.
- Weiter nehmen wir an, `left` und `right` hätten die folgenden Werte.



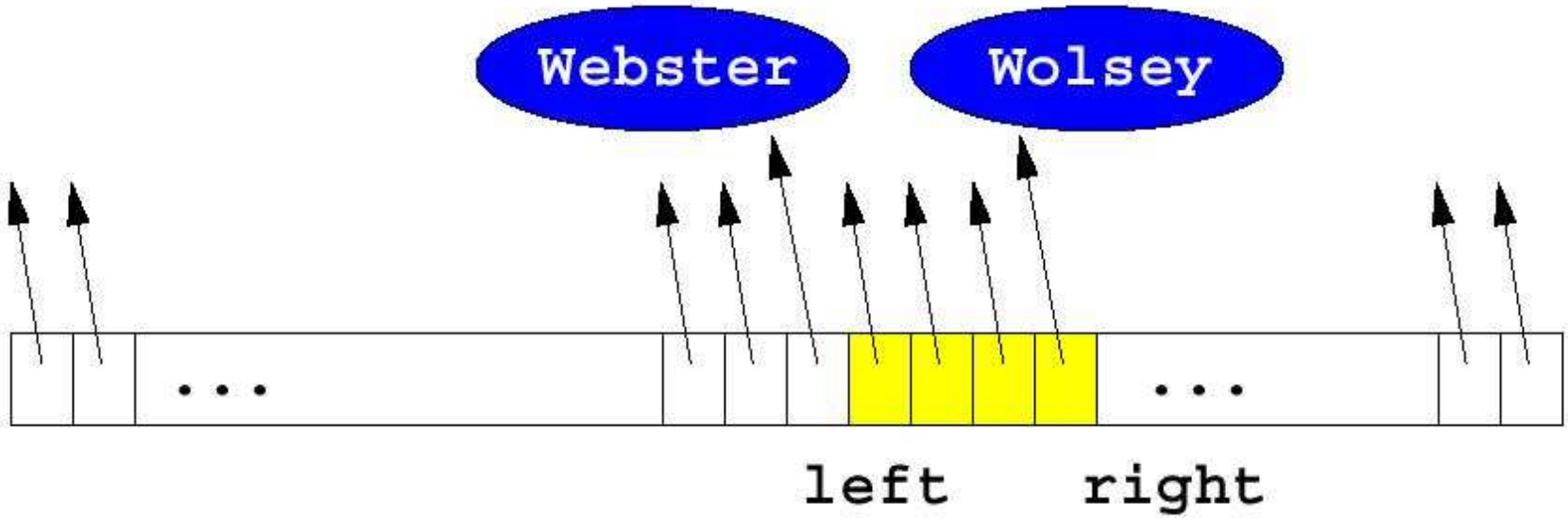
Prinzip der Binärsuche (2)

- Wir vergleichen `"Wolfram"` mit dem Element in der Mitte zwischen `left` und `right`.
- Dort finden wir das Element `"Webster"`.



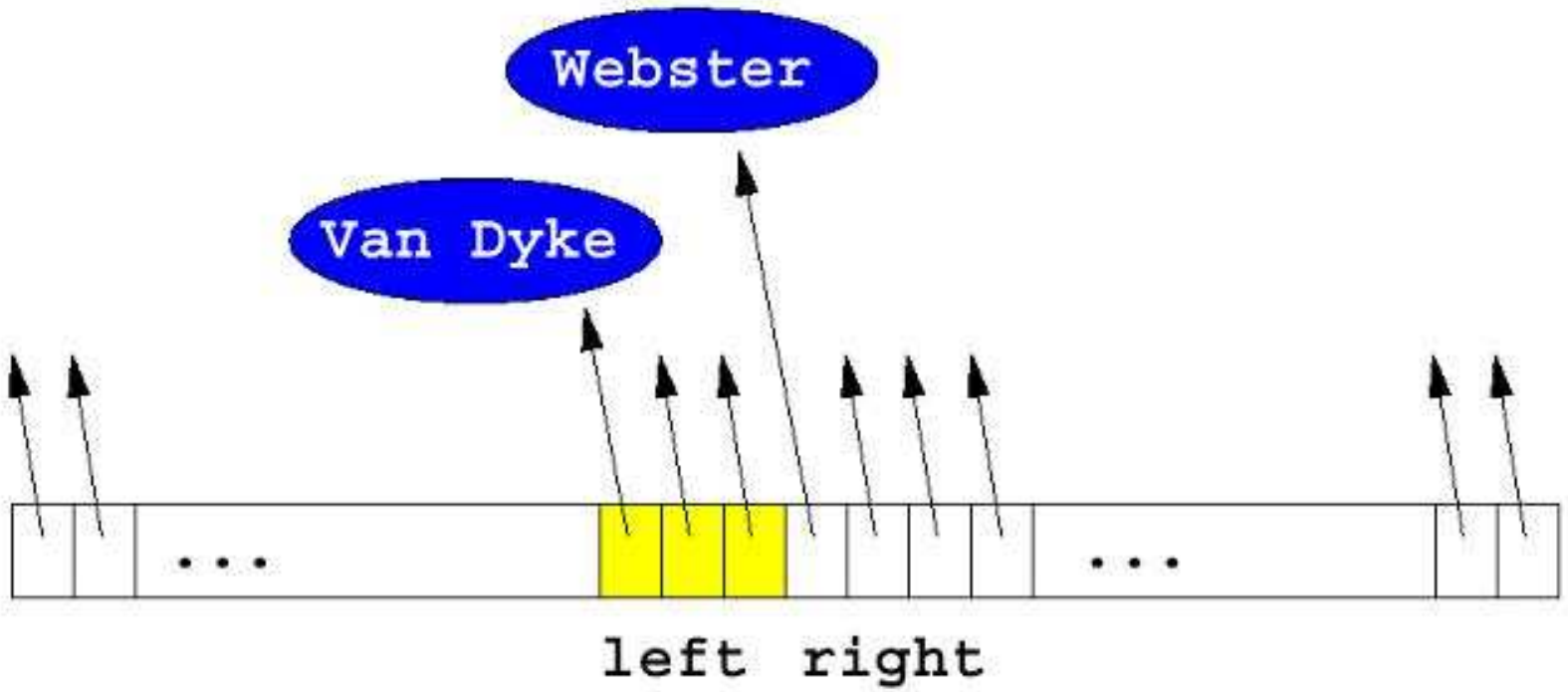
Prinzip der Binärsuche (3)

- Entsprechend der lexikographischen Ordnung ist `"Webster"` kleiner als `"Wolfram"`.
- Daher wird die Position rechts von `"Webster"` zur neuen linken Grenze `left`.



Prinzip der Binärsuche (4)

- Angenommen unser Suchstring wäre `"Wallace"`.
- Da `"Webster"` größer als `"Wallace"` ist, erhalten wir folgendes, neues Suchintervall.



Benötigte Variablen

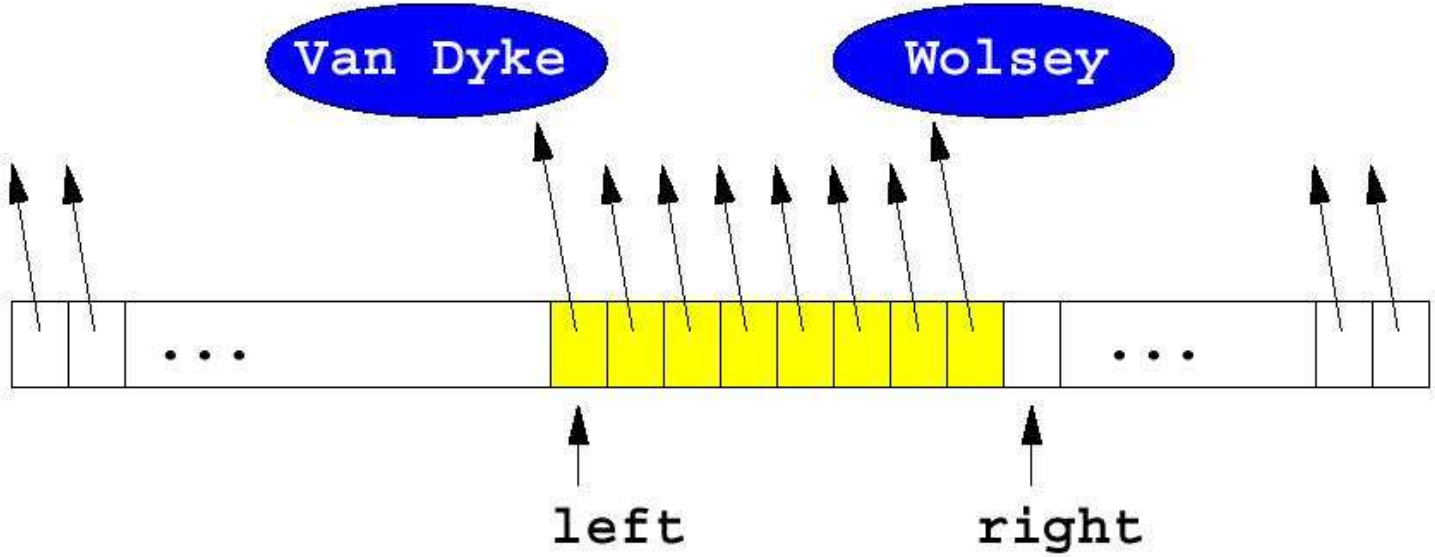
- Um das aktuelle Suchintervall zu beschreiben, benötigen wir zwei Integer-Variablen:

```
int left, right;
```

- Zusätzlich zu den Variablen selbst müssen wir die Bedeutung dieser Variablen genau festlegen.
- Im Folgenden gehen wir davon aus, dass **`left` die Position des ersten Elementes im Suchintervall** enthält.
- Im Gegensatz dazu ist **`right` die Position des ersten Elementes rechts vom Suchintervall**.

Das Suchintervall

- `left` ist der Index des kleinsten im Suchintervall.
- `right` ist der Index des ersten Elementes rechts vom Suchintervall.
- Die Indizes im Suchintervall gehen daher von `left` bis `right-1`.
- Wir müssen solange suchen, wie noch Elemente im Suchintervall sind, d.h. solange `left < right`.



Die Bedingung der `while`-Schleife

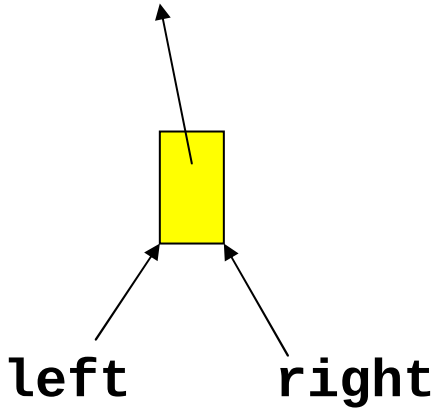
- Wir suchen solange, wie das Intervall noch wenigstens ein Element enthält. Das Suchintervall ist leer, falls

```
left >= right
```

- Die Bedingung der `while`-Schleife hat somit die folgende Form:

```
while (left < right)
```

- Nach Beendigung der `while`-Schleife ist das Suchintervall leer:



Initialisierung

- Unsere Suchmethode soll ein gegebenes Objekt `s` einem `Vector`-Objekt finden.
- Demnach muss das Suchintervall anfangs den gesamten Vektor umfassen.
- Anfangs muss `left` daher den Wert `0` und `right` den Wert `v.size()`haben.
- Die Initialisierungsanweisung für unsere Bedingung in der `while`-Schleife ist demnach:

```
left = 0;
right = v.size();
```

Der Rumpf der Schleife (1)

Im Rumpf der Schleife müssen wir drei Fälle unterscheiden:

1. Das Objekt in der Mitte ist **kleiner** als das gesuchte Objekt,
2. das Objekt in der Mitte ist **größer** als das gesuchte Objekt oder
3. das Objekt in der Mitte **stimmt** mit dem gesuchten Objekt **überein**.

Die `String`-Methode `compareTo`

- Um `String`-Objekte miteinander zu vergleichen stellt Java die Methode `compareTo` zur Verfügung, die als Ergebnis einen Wert vom Typ `int` liefert.
- Dabei liefert `s1.compareTo(s)` den Wert `0`, wenn `s1` und `s` übereinstimmen, einen Wert kleiner als `0` wenn das Empfängerobjekt `s1` **lexikographisch kleiner** ist als das Argument `s`, und einen Wert größer als `0` sonst.
- Die **lexikographische Ordnung entspricht bei `String`-Objekten der alphabetischen Reihenfolge** der Zeichenketten.

```
"hugo".compareTo("hugo") --> 0
"paul".compareTo("paula") --> <0
"paul".compareTo("hugo") --> >0
```

Der Rumpf der Schleife (2)

Sei `s1` das `String`-Objekt in der Mitte `mid` des Suchintervalls. Entsprechend unserer Festlegung von `left` und `right` muss gelten:

1. Ist `s1.compareTo(s) < 0`,so ist `left = mid+1`.
2. Ist `s1.compareTo(s) > 0`,so ist `right = mid`.
3. Ist `s1.compareTo(s) == 0`, so ist `mid` die gesuchte Position.

Dies entspricht:

```
if (s1.compareTo(s) < 0)
    left = mid+1;
else if (s1.compareTo(s) > 0)
    right = mid;
else
    return mid;
```

Der Rumpf der Schleife (3)

Es fehlen noch die Anweisungen zur Initialisierung der Variablen `mid` und `s1` im Schleifenrumpf.

1. Die Variable `mid` muss auf die Mitte des Intervalls gesetzt werden.
2. Hierfür wählen wir die Position `(left+(right-1))/2`.
3. Das Objekt in der Mitte erhalten wir dann mit der Methode `elementAt`.

```
int mid = (left + (right-1)) / 2;
String s1 = (String) v.elementAt(mid);
```

Wenn das gesuchte Objekt nicht gefunden wird . . .

- Ist das gesuchte Objekt nicht in dem Vektor enthalten, so müssen wir `-1` als Return-Wert zurückgeben.
- In unserem Programm ist das Element nicht enthalten, wenn die Schleife beendet ist, d.h. wenn `left >= right`.
- Wir führen daher nach Beendigung der Schleife die Anweisung

  ```
  return -1;
  ```

  aus.

Das komplette Programm

```
import java.util.*;

class BinarySearch {
    static int binarySearch(Vector<String> v, String s){
        int left, right;

        left = 0;
        right = v.size();

        while (left < right){
            int mid = (left + (right-1)) / 2;
            String s1 = v.elementAt(mid);
            if (s1.compareTo(s) < 0)
                left = mid+1;
            else if (s1.compareTo(s) > 0)
                right = mid;
            else
                return mid;
        }
        return -1;
    }
}
```

Ein Anwendungsbeispiel

```
import java.io.*;
import java.util.*;

class UseBinarySearch {
    public static void main(String [] args) {
        Vector<String> v = new Vector<String> ();

        v.addElement("hugo");
        v.addElement("paula");
        v.addElement("peter");

        for (int i = 0; i < v.size(); i++)
            System.out.println(v.elementAt(i) + " is at position „
               + BinarySearch.binarySearch(v, v.elementAt(i)));
        System.out.println("wolfram is at position „
               + BinarySearch.binarySearch(v,"wolfram"));
    }
}
```

Ausgabe des Anwendungsprogramms

Die Ausgabe dieses Programms ist:

```
java UseBinarySearch

hugo is at position 0
paula is at position 1
peter is at position 2
wolfram is at position –1

Process UseBinarySearch finished
```

Aufwandsabschätzung für `BinarySearch`

- Im schlimmsten Fall bricht die Suche ab, nachdem das Suchintervall die Größe 1 hat.
- In jeder Runde halbieren wir das Suchintervall.
- Starten wir mit 4000 Elementen, so ergeben sich ungefähr folgende 12 Intervallgrößen:

  $4000 \rightarrow 2000 \rightarrow 1000 \rightarrow 500 \rightarrow 250 \rightarrow 125 \rightarrow 62 \rightarrow 31 \rightarrow 15 \rightarrow 7 \rightarrow 3 \rightarrow 1$

- Da wir ein Intervall mit n Elementen größenordnungsmäßig $\log_2(n)$mal halbieren können, erhalten wir eine Komplexität von $O(\log(n))$.
- `BinarySearch` **benötigt demnach im schlechtesten Fall nur logarithmisch viele Schritte um das gesuchte Objekt zu finden**.

Sortieren

- Unsere Suchprozedur `BinarySearch` für **binäre Suche** erforderte, dass die Elemente im Vektor sortiert sind.
- Im Folgenden wollen wir nun betrachten, wie man die Elemente eines Vektors **sortieren** kann.
- Hierbei bezeichnen wir einen Vektor als gemäß einer Ordnung Ω sortiert, falls er folgende Eigenschaft hat:
- Für je zwei Zahlen i und j mit $0 \leq i \leq j \leq$ v.size() gilt `v.elementAt(i)` ist unter Ω kleiner oder gleich `v.elementAt(j)`.
- Wir betrachten hier demnach aufsteigend sortierte Vektoren.
- Für das Sortieren gibt es zahlreiche Methoden. Wir werden hier das sehr einfache **Sortieren durch Auswählen** betrachten.

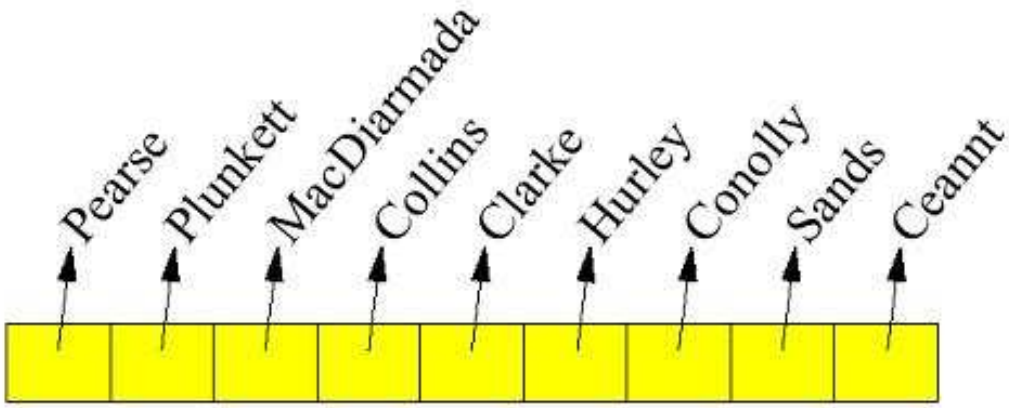
Informelles Verfahren

Für einen Vektor mit `n==v.size()` Elementen gehen wir folgendermaßen vor.

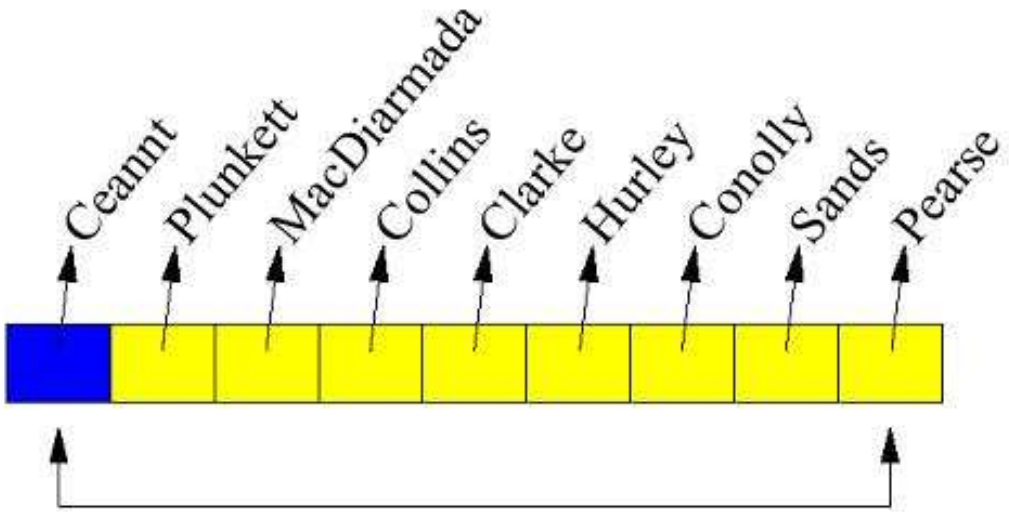
1. Wir suchen das kleinste Element an den Positionen `0` bis `n-1`.
2. Wir vertauschen das kleinste Element mit dem Element an Position `0`. Danach ist das kleinste Element offensichtlich bereits an der richtigen Position.
3. Wir wiederholen die Schritte 1) und 2) für die die Positionen `k==1,..., n-2`. Dabei suchen wir stets das kleinste Element an den Positionen `k` bis `n-1` und vertauschen es mit dem Element an Position `k`.

Vorgehen dieser Prozedur (1)

Ausgangssituation:

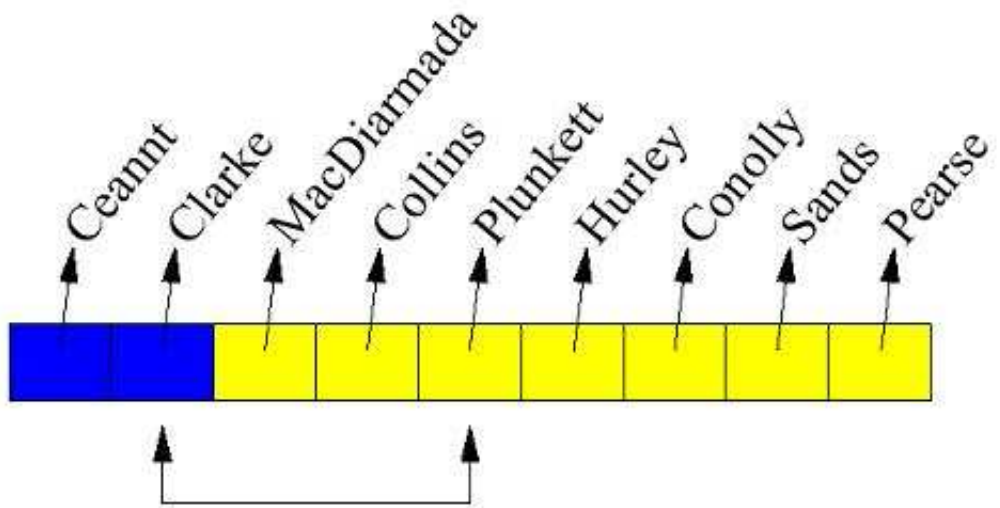


`k=0:`

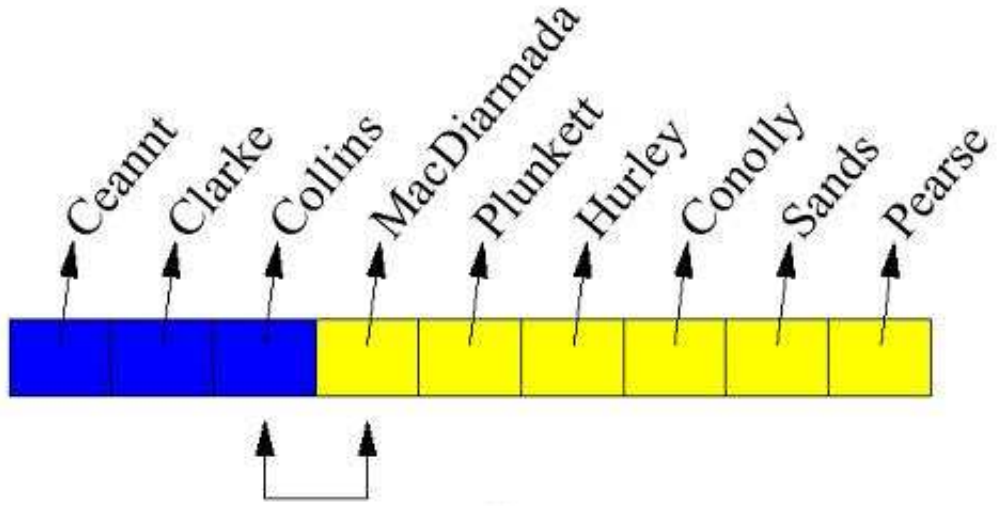


Vorgehen dieser Prozedur (2)

k=1:



k=2:



Die benötigten Variablen

- Während der Sortierung wird der Vektor in zwei Teile aufgeteilt.
- Während die eine Hälfte des Vektors ist bereits sortiert ist, enthält die zweite Hälfte alle noch nicht sortierten Elemente.
- Um die Position zu markieren, ab der wir noch sortieren müssen, verwenden wir die Variable k. Diese wird anfangs mit `0` initialisiert.
- Darüber hinaus verwenden wir die Variable n für die Anzahl der Element im Vektor.

```
int k = 0;
int n = v.size();
```

Skelett der Prozedur

```
static void sort(Vector v){
    int k =0;
    int n = v.size();

    while (condition)
        body
    ...
}
```

Die Bedingung der `while`-Schleife und Terminierung

- Offensichtlich sind wir fertig, wenn der zu sortierende Bereich des `Vector`-Objekts nur noch ein Objekt enthält.
- Vektoren der Länge `1` und `0` sind immer sortiert.
- Dementsprechend können wir abbrechen, sobald `k >= n-1`
- Die Bedingung der `while`-Schleife ist somit:

```
while (k < n-1)
```

- Um die Terminierung zu garantieren, müssen wir in jeder Runde `k` um `1` erhöhen:

```
k++;
```

Suchen des kleinsten Elementes

- Um das kleinste Element im noch nicht sortierten Bereich zu suchen, müssen wir unsere Methode `linearSearch` modifizieren.
- Anstatt die Suche immer bei `0` zu starten, erlauben wir jetzt einen beliebigen Startpunkt, den wir als Argument übergeben:

```
static int getSmallest(Vector<String> v, int k)
```

Die modifizierte `getSmallest`-Methode

```
static int getSmallest(Vector<String> v, int k) {
    if  (v==null || v.size()<=k)
        return -1;
    int     i = k+1;
    int     small = k;
    String  smallest = v.elementAt(small);
    while (i != v.size()) {
        String current = v.elementAt(i);
        if  (current.compareTo(smallest)<0){
            small = i;
            smallest = v.elementAt(i);
        }
        i++;
    }
    return small;
}
```

Tauschen zweier Elemente

Das Vertauschen von zwei Elementen haben wir bereits bei der Invertierung der Reihenfolge im Vektor kennengelernt:

```
static void swap(Vector<String> v, int i, int j){
    String  o = v.elementAt(i);
    v.setElementAt(v.elementAt(j), i);
    v.setElementAt(o, j);
}
```

Die komplette Sortiermethode

- Um den Vektor v mit n==v.size() Elementen zu sortieren, gehen wir wie geplant vor.
- Für k = 0, ..., n-2 vertauschen wir das Element an Position k mit dem kleinsten Element im Bereich i = k, ..., n-1:

```
static void sort(Vector<String> v) {
    int k = 0;
    int n = v.size();
    while (k < n-1) {
        swap(v, k, getSmallest(v, k));
        k++;
    }
}
```

Eine äquivalente Version auf der Basis der `for`-Anweisung

```
static void sort(Vector<String> v) {
    for(int k =0;k<v.size()-1; k++)
        swap(v, k, getSmallest(v, k));
}
```

Das Programm zum Sortieren eines Vektors

```
import java.io.*;
import java.util.*;

class Sort {

    static void swap(Vector<String> v, int i, int j){
        String o = v.elementAt(i);
        v.setElementAt(v.elementAt(j), i);
        v.setElementAt(o, j);
    }

    static int getSmallest(Vector<String> v, int k) {
        if (v==null || v.size()<=k)
            return -1;
        int i = k+1;
        int small = k;
        String smallest = v.elementAt(small);
        while (i != v.size()) {
            String current = v.elementAt(i);
            if (current.compareTo(smallest)<0){
                small = i;
                smallest = v.elementAt(i);
            }
            i++;
        }
        return small;
    }
    static void sort(Vector<String> v) {
        for(int k =0;k <v.size()-1; k++)
            swap(v, k, getSmallest(v, k));
    }

    public static void main(String arg[]) {
        Vector<String> v =
                    new Vector<String> ();
        Vector<String> v1 =
                    new Vector<String> ();
        Vector<String> v2 =
                    new Vector<String> ();

        v.addElement("Albert");
        v.addElement("Ludwig");
        v.addElement("Jaeger");
        v1.addElement("Albert");
        sort(v);
        sort(v1);
        sort(v2);
        System.out.println(v.toString());
        System.out.println(v1.toString());
        System.out.println(v2.toString());
    }
}
```

Eine Aufwandsanalyse für Sortieren durch Auswählen

In der Methode `sort` wird der Rumpf der `for`-Schleife genau `n = v.size()-1` mal ausgeführt.

In jeder Runde werden die Methoden `getSmallest` und `swap` ausgeführt.

Während `swap` jeweils drei Anweisungen ausführt, benötigt `getSmallest` stets `n-k-1` Schritte.

Insgesamt sind die Kosten daher

$$(n-1) * c_1 + ((n-1) + (n-2) + \ldots + 1) * c_2 = (n-1) * c_1 + \frac{(n-1) * n}{2} * c_2$$

wobei c_1 und c_2 die Kosten für die einzelnen Operationen sind.

Der **Aufwand für Sortieren durch Auswählen liegt daher in $O(n^2)$**.

Dabei gibt es **keinen Unterschied zwischen Best, Worst und Average Case**.

Alternative Sortiermethoden

- Für das Sortieren von Kollektionen gibt es eine Vielzahl von (in der Regel komplizierteren) Alternativen.
- Die meisten dieser Verfahren sind zumindest in bestimmten Fällen (Worst Case, Best Case oder Average Case) besser als Sortieren durch Auswählen.
- **Die besten Methoden benötigen $O(n \log n)$**.
- Dies entspricht auch dem Aufwand, den man für das Sortieren von Vektoren mindestens benötigt.

Zusammenfassung

- Der **Index der Elemente in Vektoren beginnt bei** 0.
- **Aufwandsanalysen** dienen dazu, den **Zeit- und Platzbedarf von Programmen/Verfahren** zu **ermitteln**.
- Für Aufwandsanalysen verwenden wir die **O-Notation**.
- Es gibt **verschiedene Klassen für die Komplexität**. Beispiele sind $O(n)$, $O(n^2)$ oder O(log n).
- Die **Suche in unsortierten Vektoren gelingt in $O(n)$**.
- Sind die Elemente im **Vektor sortiert**, können wir mit **Binärsuche** in $O(\log n)$ **suchen**.
- Eine Methode für das Sortieren ist **Sortieren durch Auswählen**.
- **Dies benötigt** $O(n^2)$. Die effizientesten Verfahren können Vektoren jedoch in $O(n \log n)$ sortieren.