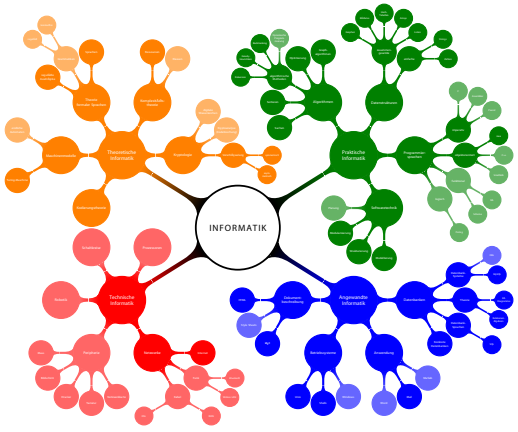
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Kapitel 16
Suchalgorithmen
Welche Codons kodieren Arginin?

Vorlesung Einführung in die Informatik 1 vom 17. Dezember 2013 von Till Tantau



Lernziele von Kapitel 16

1. Lineare Suche verstehen und implementieren können
2. Binäre Suche verstehen und implementieren können
3. Laufzeit der Verfahren vergleichen können
4. Abschätzen können, wann sich Vorsortierung lohnt

Gliederung von Kapitel 16

Es gibt viele Varianten des Suchproblems.

Suchen eines Wertes

Eingaben:

- Ein Array von Werten und
- *ein Wert*, der gesucht wird.

Ausgabe:

- *Eine* Position, an der der Wert im Array vorkommt.

Es gibt viele Varianten des Suchproblems.

Suchen aller Werte

Eingaben:

- Ein Array von Werten und
- *ein Wert*, der gesucht wird.

Ausgabe:

- *Alle* Positionen, an der der Wert im Array vorkommt.

Es gibt viele Varianten des Suchproblems.

Suchen eines Wertes mit einer Eigenschaft

Eingaben:

- Ein Array von Werten und
- *eine Eigenschaft*, die Werte haben können oder auch nicht.

Ausgabe:

- *Eine* Position eines Wertes, der die Eigenschaft hat.

Es gibt viele Varianten des Suchproblems.

Suchen aller Werte mit einer Eigenschaft

Eingaben:

- Ein Array von Werten und
- *eine Eigenschaft*, die Werte haben können oder auch nicht.

Ausgabe:

- *Alle* Positionen von Werten, die die Eigenschaft haben.

Die lineare Suche ist das einfachste Suchverfahren.

- Bei der *linearen Suche* werden einfach alle Werte des Arrays überprüft.
- Eine oder alle Positionen, an denen Werte mit der gewünschten Eigenschaft stehen, werden zurückgegeben.

Beispiel einer linearen Suche.

Finden einer Telefonnummer, die auf 6 endet.


```
String[] telephoneNumbers = {"7974311",
                             "2147856",
                             "2161555",
                             "5553466"};

// Suche linear nach einer Telefonnummer, die auf 6 endet.

String nummer = "";

for (int i=0; i < telephoneNumbers.length; i++) {
  if (telephoneNumbers[i].charAt
        (telephoneNumbers[i].length()-1) == '6') {
    nummer = telephoneNumbers[i];
  }
}
// nummer ist jetzt "5553466"
```

Beispiel einer linearen Suche.

Finden der Position einer Telefonnummer im einem Array.


```
String[] telephoneNumbers = {"7974311",
                             "2147856",
                             "2161555",
                             "5553466"};

// Suche linear nach "2161555":

int position_of_value = -1; // noch nicht gefunden

for (int i=0; i < telephoneNumbers.length; i++) {
  if (telephoneNumbers[i].equals("2161555")) {
    positions_of_value = i;
  }
}
// position_of_value ist jetzt 2.
```

Eine allgemeine lineare Suche.


```
class SearchAlgorithms
{
  static int linearSearch(String[] strings,
                          String   value)
  {
    // Findet erstes Vorkommen von value in strings.
    // Kommt es nicht vor wird -1 zurückgegeben

    for (int i = 0; i < strings.length; i++) {
      if (strings[i].equals(value)) {
        return i;
      }
    }

    return -1;
  }
}
```

5-Minuten-Aufgabe

Zur Übung

Wie viele Vergleiche führt die Methode `linearSearch` bei einem Array der Länge *n*

1. mindestens,
2. höchstens und
3. im Durchschnitt

aus?

Beispielanwendung der linearen Suche.

- Zu einem Codon soll die zugehörige Aminosäure ermitteln werden und umgekehrt.
- Dazu werden zwei Arrays erstellt:
- Das erste speichert die Codons, das zweite an der entsprechenden Position die Säure.

Codon zu Aminosäure Umrechnung.


```
static String codonToAcid(String codon)
{
  String[] codons = {
    "UUU","UUC","UUA","UUG","UCU","UCC","UCA","UCG",
    "UAU","UAC","UAA","UAG","UGU","UGC","UGA","UGG",
    "CUU","CUC","CUA","CUG","CCU","CCC","CCA","CCG",
    "CAU","CAC","CAA","CAG","CGU","CGC","CGA","CGG",
    "AUU","AUC","AUA","AUG","ACU","ACC","ACA","ACG",
    "AAU","AAC","AAA","AAG","AGU","AGC","AGA","AGG",
    "GUU","GUC","GUA","GUG","GCU","GCC","GCA","GCG",
    "GAU","GAC","GAA","GAG","GGU","GGC","GGA","GGG"};
  String[] acids = {
    "Phe","Phe","Leu","Leu","Ser","Ser","Ser","Ser",
    "Tyr","Tyr","Stp","Stp","Cys","Cys","Stp","Trp",
    "Leu","Leu","Leu","Leu","Pro","Pro","Pro","Pro",
    "His","His","Gln","Gln","Arg","Arg","Arg","Arg",
    "Ile","Ile","Ile","Met","Thr","Thr","Thr","Thr",
    "Asn","Asn","Lys","Lys","Ser","Ser","Arg","Arg",
    "Val","Val","Val","Val","Ala","Ala","Ala","Ala",
    "Asp","Asp","Glu","Glu","Gly","Gly","Gly","Gly"};
  return
    acids[linearSearch(codons, codon)];
}
```

5-Minuten-Aufgabe

Zur Übung

Schreiben Sie eine Methode, die eine Aminosäure als Eingabe erhält und *alle* Codons zu dieser Aminosäure auf dem Bildschirm ausgibt. (Für Fortgeschrittene: Wie können Sie alle Codons in einem Array zurückgeben?)

Die Grundidee der binären Suche.

Beobachtung

- Suchen wir einen Namen im Telefonbuch, so suchen wir diesen natürlich nicht linear.
- Vielmehr fangen wir grob in der Mitte an und gehen dann sprungweise nach vorne oder nach hinten.

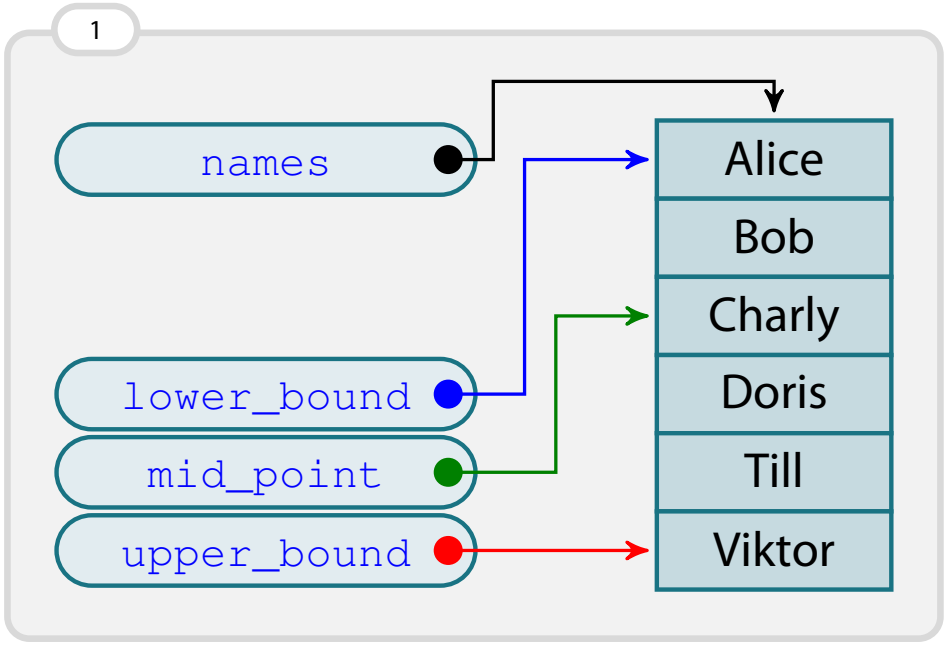
Binäre Suche

- Binäre Suche arbeitet auf *sortierten* Arrays.
- Man *halbiert* zu Anfang den Suchraum in der Mitte.
- Dann behandelt man nur noch eine der beiden Seiten.

Beispiel einer binären Suche.

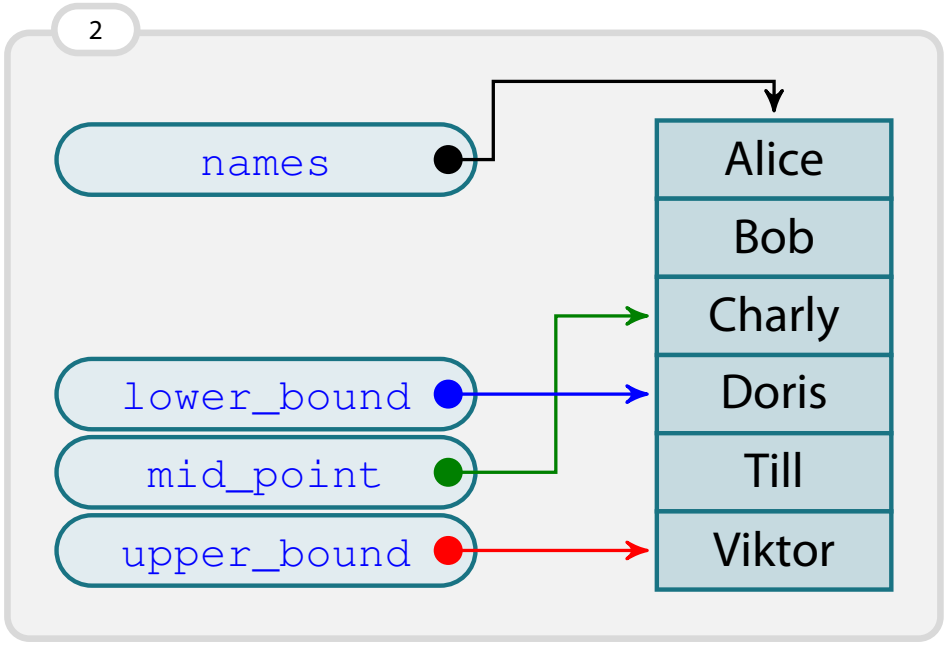

```
String[] names =
 {"Alice", "Bob", "Charly", "Doris", "Till", "Viktor"};

// Binäre Suche nach "Till":

int lower_bound = 0;
int upper_bound = names.length - 1;

while (lower_bound != upper_bound)
{
  int mid_point = (lower_bound + upper_bound) / 2;
  if (names[mid_point] < "Till") {// geschummelt
    lower_bound = mid_point + 1;
  }
  else {
    upper_bound = mid_point;
  }
}

return lower_bound;
```

Beispiel einer binären Suche.



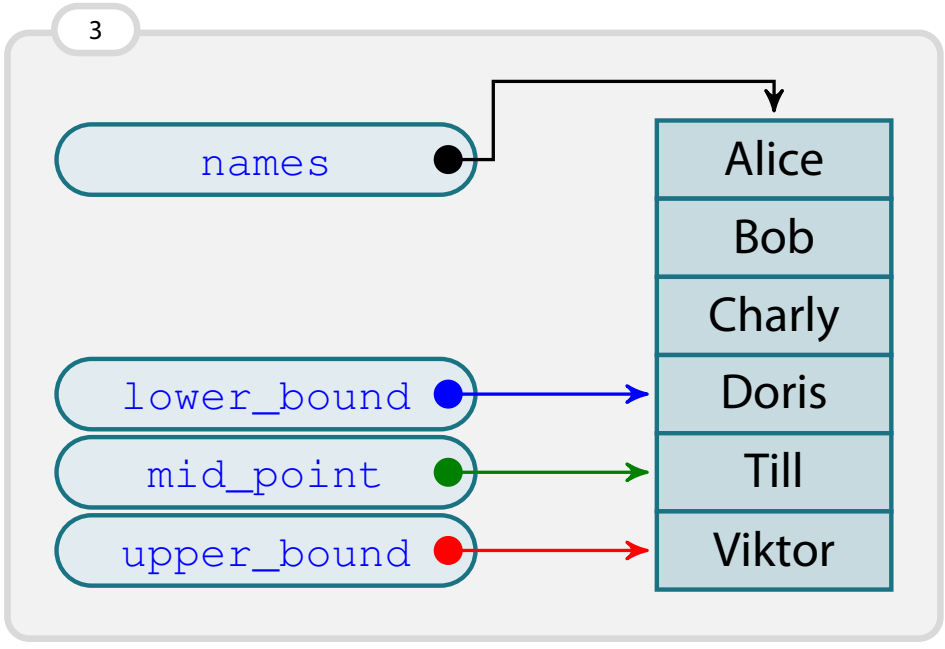

```
while (lower_bound != upper_bound)
{
  int mid_point = (lower_bound + upper_bound) / 2;
  if (names[mid_point] < "Till") { // geschummelt
    lower_bound = mid_point + 1;
  }
  else {
    upper_bound = mid_point;
  }
}
```

Beispiel einer binären Suche.



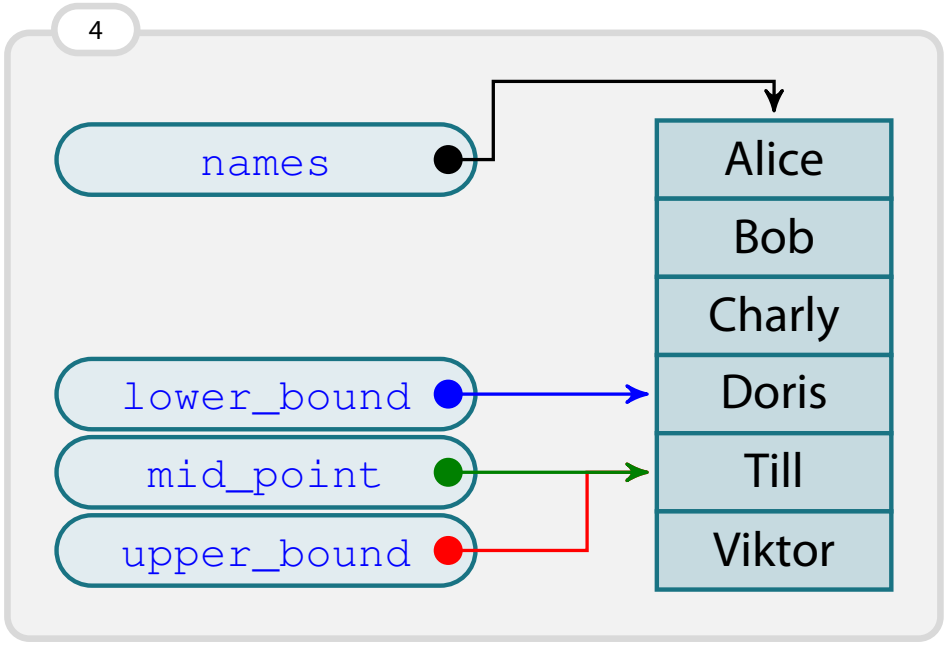

```
while (lower_bound != upper_bound)
{
  int mid_point = (lower_bound + upper_bound) / 2;
  if (names[mid_point] < "Till") { // geschummelt
    lower_bound = mid_point + 1;
  }
  else {
    upper_bound = mid_point;
  }
}
```

Beispiel einer binären Suche.



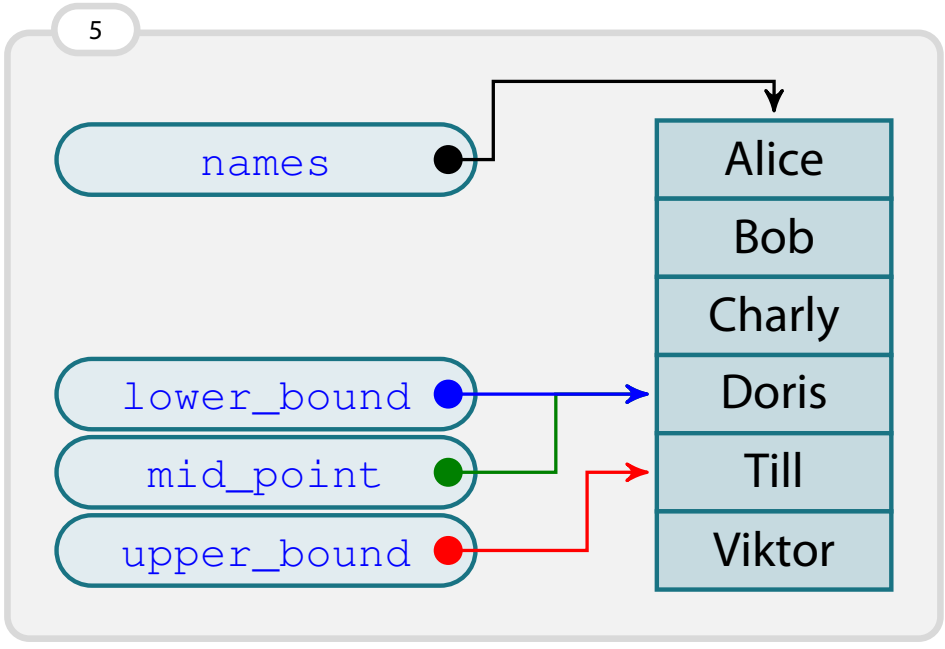

```
while (lower_bound != upper_bound)
{
  int mid_point = (lower_bound + upper_bound) / 2;
  if (names[mid_point] < "Till") { // geschummelt
    lower_bound = mid_point + 1;
  }
  else {
    upper_bound = mid_point;
  }
}
```

Beispiel einer binären Suche.



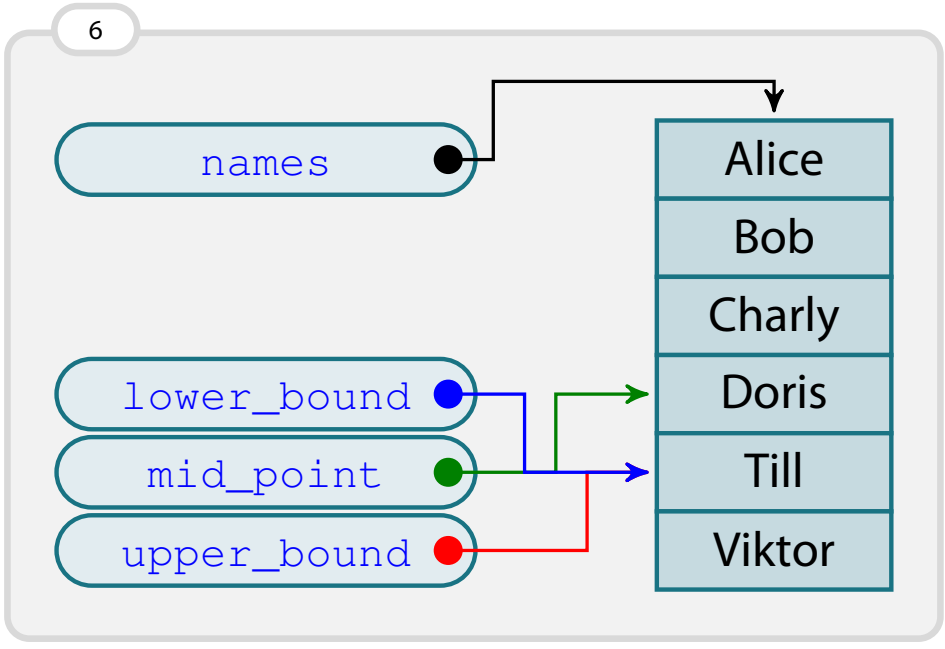

```
while (lower_bound != upper_bound)
{
  int mid_point = (lower_bound + upper_bound) / 2;
  if (names[mid_point] < "Till") { // geschummelt
    lower_bound = mid_point + 1;
  }
  else {
    upper_bound = mid_point;
  }
}
```

Beispiel einer binären Suche.




```
while (lower_bound != upper_bound)
{
  int mid_point = (lower_bound + upper_bound) / 2;
  if (names[mid_point] < "Till") { // geschummelt
    lower_bound = mid_point + 1;
  }
  else {
    upper_bound = mid_point;
  }
}
```

Beispiel einer binären Suche.




```
while (lower_bound != upper_bound)
{
  int mid_point = (lower_bound + upper_bound) / 2;
  if (names[mid_point] < "Till") { // geschummelt
    lower_bound = mid_point + 1;
  }
  else {
    upper_bound = mid_point;
  }
}
```

Eine allgemeine binäre Suche.


```
class SearchAlgorithms
{
  static int binarySearch(String[] strings,
                          String   value)
  {
    // Finde Vorkommen von value in strings (sortiert).
    int lower_bound = 0;
    int upper_bound = strings.length - 1;

    while (lower_bound != upper_bound)
    {
      // Berechne Mitte des Intervals
      int mid_point = (lower_bound + upper_bound) / 2;

      if (strings[mid_point].compare(value) < 0) {
        // Im oberen Interval, erhöhe untere Schranke
        lower_bound = mid_point + 1;
      }
      else { // Unteres Interval, senke obere Schranke
        upper_bound = mid_point;
      }
    }
    return lower_bound;
  }
}
```

5-Minuten-Aufgabe

Zur Übung

Wie viele Vergleiche führt die Methode `binarySearch` bei einem Array der Länge *n*

1. mindestens,
2. höchstens und
3. im Durchschnitt

aus?

Vorsortierung ist nützlich, wenn öfters gesucht werden soll.

- Binäre Suche *funktioniert nur* auf sortierten Daten.
- Will man unbedingt binäre Suche verwenden, so kann man die Daten *vorsortieren*.
- Das Sortieren von Daten dauert aber (wesentlich) länger als *eine* lineare Suche.
- Sind die Daten aber einmal sortiert, gehen *nachfolgende* binäre Suchen schnell.

Folgerung

Vorsortieren lohnt sich nur, wenn in den Daten *mehrmals* gesucht werden soll.

(Genauer: Vorsortieren lohnt sich ab etwa $\log_2 n$ Suchen.)

Lineare Suche

Die *lineare Suche* durchläuft einfach alle Elemente. Das Wesentliche ist folgende die Schleife:

```
for (int i = 0; i < values.length; i++) {
  if (values[i] == what_we_search_for) {
    return i;
  }
}
```

Die lineare Suche benötigt im Schnitt $n/2$ Vergleiche bei Arrays der Länge n.

Zusammenfassung

Binäre Suche

Die *binäre Suche* springt in einem *sortierten* Array herum. Das Wesentliche ist folgende die Schleife:

```
while (lower_bound != upper_bound) {
  int mid_point = (lower_bound + upper_bound) / 2;
  if (values[mid_point] < what_we_search_for) {
    lower_bound = mid_point + 1;
  }
  else {
    upper_bound = mid_point;
  }
}
```

Die binäre Suche benötigt $\log_2 n$ Vergleiche, was *sehr schnell ist*.