10 Suchalgorithmen

10.1 Sequenzielle Suche

10.2 Binäre Suche

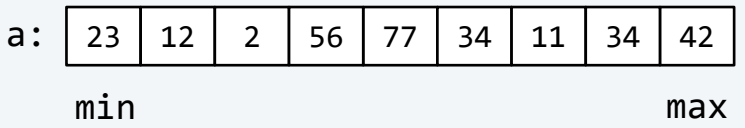
Einleitung

- Sinn und Zweck der Speicherung von Datenbeständen ist es, später auf sie als Ganzes oder auf Teile davon zugreifen zu können
- Dabei ist es oft erforderlich, nach jenen Datenobjekten zu suchen, die ein bestimmtes Kriterium erfüllen
- Die Suche nach Datenobjekten, zum Zweck der Informationsbeschaffung, ist ein zentraler Bestandteil nahezu jedes Softwaresystems. Es besteht daher ein besonderes Interesse an möglichst effizienten Suchverfahren
- Im Allgemeinen haben Datenobjekte mehrere Merkmale (Attribute), von denen im Kontext einer bestimmten Suchaufgabe einem die (Schlüssel-) Rolle zukommt, d. h. nach dessen Ausprägung (Wert) soll gesucht werden. Man nennt dieses Merkmal deshalb in der Regel auch Suchschlüssel (*key*)

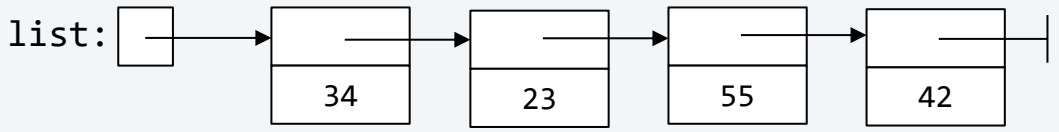
10.1 Sequenzielle Suche

- Die sequenzielle Suche (*sequential search*) ist das einfachste aber auch das ineffizienteste Suchverfahren
- Es hat jedoch den Vorteil, dass es bei jeder Art von Behältern, unabhängig von der verwendeten Datenstruktur und der Anordnung der Elemente (also insbesondere auch bei unsortierten Behältern) anwendbar ist

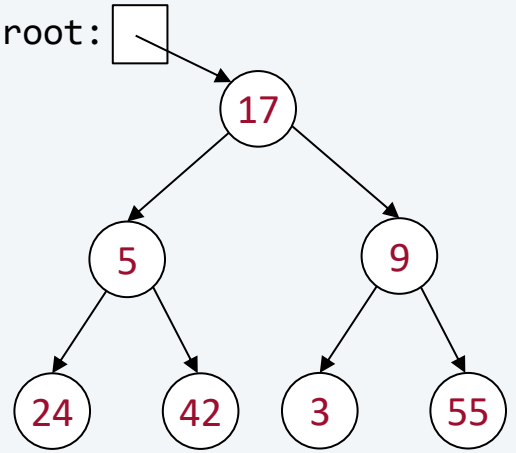
Feld:



Verkettete Liste:



Binärbaum:



(1) Sequenzielle Suche in Feldern

Gegeben seien folgende Datentypen

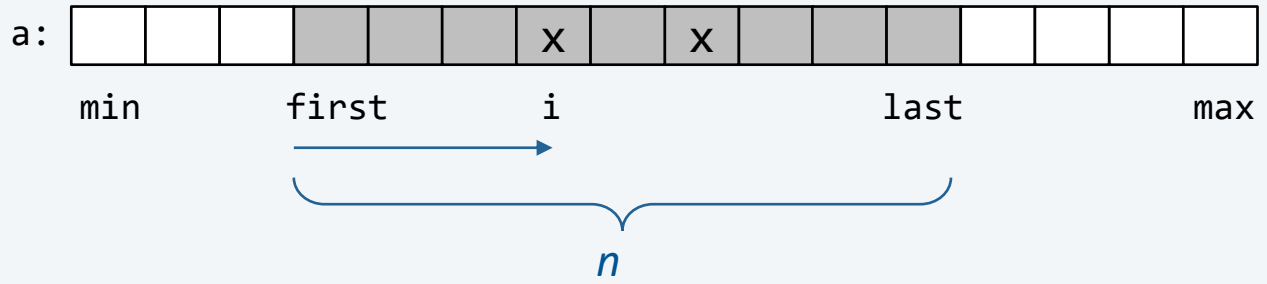
```
type
  KeyType = …  -- type which allows operation =
               -- and operation < for “sorted” containers
ObjectType = compound
    key: KeyType
    data1: DataType1   -- any type
    data2: DataType2   -- any type
    …
    dataN: DataTypeN   -- any type
  end -- ObjectType

const
  min = …
  max = …  -- min ≤ max

type
  ElementType = ObjectType
  ArrayType = array [min:max] of ElementType
```

Sequenzielle Suche in Feldern

Aufgabe: Suche nach dem Index des ersten Elements mit dem Schlüsselwert x in einem Teilbereich eines Felds



Gesucht ist ein Algorithmus `FirstIndexOf`, der in dem durch die beiden Grenzindizes `first` und `last` (mit $\texttt{min} \leq \texttt{first} \leq \texttt{last} \leq \texttt{max}$) festgelegten Teilbereich des Felds a „von vorne weg“ sequentiell nach dem Index des ersten Elements sucht, das in seiner Komponente key den gesuchten Schlüsselwert x enthält.

Der Algorithmus soll als Ergebnis den Wert `min - 1` liefern, falls kein solches Element gefunden werden konnte.

Sequenzielle Suche in Feldern

Algorithmus:

```
FirstIndexOf(↓a: ArrayType ↓first: int ↓last: int ↓x: KeyType): int
  var
    i: int
begin
  i := first
  while (i ≤ last) and (a[i].key ≠ x) do
    i := i + 1
  end -- while
  if i ≤ last then
    return i
  else
    return min – 1
  end -- if
end FirstIndexOf
```

(2) Sequenzielle Suche in verketteten Listen

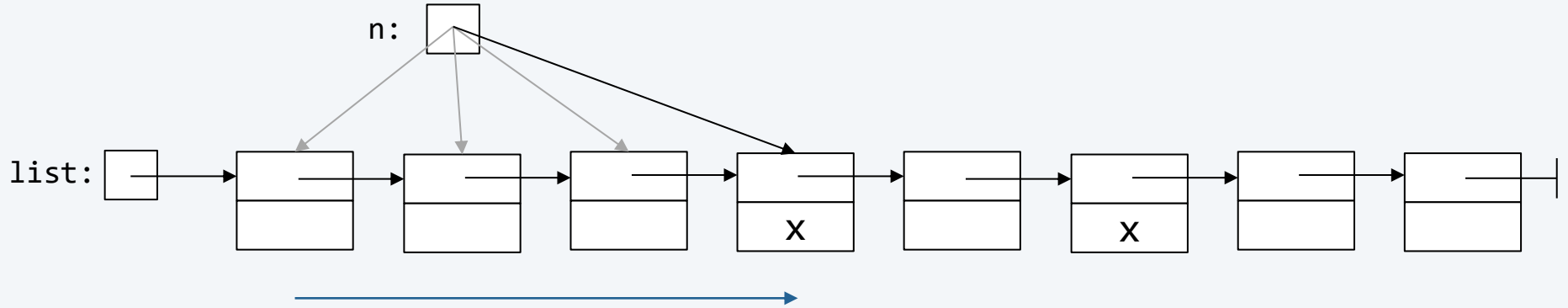
Gegeben seien folgende Datentypen

```
type
  ListNodePtr = →ListNode
  ListNode = compound
    next, prev: ListNodePtr
    key: KeyType
    dataI: DataTypeI
  end -- ListNode
  ListPtr = ListNodePtr
```

Gesucht ist ein Algorithmus `FirstNodeWith` der in einer einfach- oder doppelt-verketteten Liste `list` sequentiell nach dem ersten Knoten sucht, der in der Komponente `key` den gesuchten Wert `x` enthält.

Der Algorithmus soll als Ergebnis den Wert `null` liefern, falls kein solcher Knoten gefunden werden konnte.

Sequenzielle Suche in verketteten Listen



Algorithmus:

```
FirstNodeWith(↓list: ListPtr ↓x: KeyType): ListNodePtr
  var
    n: ListNodePtr
begin
  n := list
  while (n ≠ null) and (n→key ≠ x) do
    n := n→next
  end -- while
  return n
end FirstNodeWith
```

(3) Sequenzielle Suche in Binärbäumen

Gegeben seien folgende Datentypen

```
type
  TreeNodePtr = →TreeNode
  TreeNode = compound
    left, right: TreeNodePtr
    key: KeyType
    dataI: DataTypeI
  end -- TreeNode
  TreePtr = TreeNodePtr
```

Gesucht ist ein (rekursiver) Algorithmus `FirstInOrderNodeWith` der in einem Binärbaum `tree` in der durch den In-Order-Durchlauf gegebenen Reihenfolge sequenziell nach dem ersten Knoten sucht, der in der Komponente `key` den gesuchten Wert `x` enthält.

Der Algorithmus soll als Ergebnis den Wert `null` liefern, falls kein solcher Knoten gefunden werden konnte.

Sequentielle Suche in Binärbäumen

Algorithmus

```
FirstInOrderNodeWith(↓tree: TreePtr ↓x: KeyType): TreeNodePtr
  var
    n: TreeNodePtr
begin
  if tree ≠ null then
    n := FirstInOrderNodeWith(↓tree→left ↓x)
    if n ≠ null then
      return n
    end -- if
    if tree→key = x then
      return tree
    end -- if
    n := FirstInOrderNodeWith(↓tree→right ↓x)
    if n ≠ null then
      return n
    end -- if
  end -- if
  return null
end FirstInOrderNodeWith
```

1. look in left subtree

2. look in root node

3. look in right subtree

Laufzeitkomplexität sequenzieller Suchalgorithmen

Im günstigsten Fall: Die minimale Anzahl von Suchschritten ist $S_{min}(n) = 1$.

- Dieser Fall tritt ein, wenn der zu durchsuchende Bereich des Behälters nur ein Element umfasst oder der gesuchte Schlüsselwert gleich im ersten besuchten Element gefunden wird.

Im ungünstigsten Fall: Die max. Anzahl von Suchschritten ist $S_{max}(n) = n$.

- Dieser Fall tritt ein, wenn der gesuchte Schlüsselwert in keinem der Elemente vorkommt oder erst im letzten besuchten Element gefunden wird.

Im typischen oder durchschnittlichen Fall:

- Wenn wir annehmen, dass alle n zu durchsuchenden Elemente des Behälters unterschiedliche Schlüsselwerte enthalten und nach jedem Schlüsselwert genau einmal gesucht wird, ergibt sich folgende Gesamtanzahl von Suchschritten $S_{total}(n) = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$.

Laufzeitkomplexität sequenzieller Suchalgorithmen

Im typischen oder durchschnittlichen Fall

- Wenn wir weiter annehmen, dass nach jedem Schlüsselwert mit der gleichen Wahrscheinlichkeit gesucht wird, so beträgt die durchschnittliche Anzahl von Suchschritten

$$S_{avg}(n) = \frac{S_{total}}{n} = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2}$$

- Für große n gilt also

$$S_{avg}(n) \approx \frac{n}{2}$$

- Die asymptotische Laufzeitkomplexität der sequenziellen Suchalgorithmen ist demnach $O(n)$, sie ist also linear.
- Das bedeutet z. B. dass die Suche in einem Behälter mit doppelt so vielen Elementen, doppelt so viele Suchschritte erfordert.

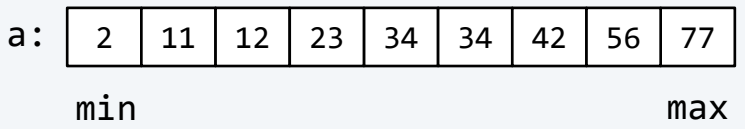
10.2 Binäre Suche

Eine deutliche Verbesserung des Laufzeitverhaltens gegenüber jenem der sequentiellen Suche, erreichen wir durch ein Suchverfahren, das nach dem Prinzip Teile und Herrsche (*divide and conquer*) arbeitet:

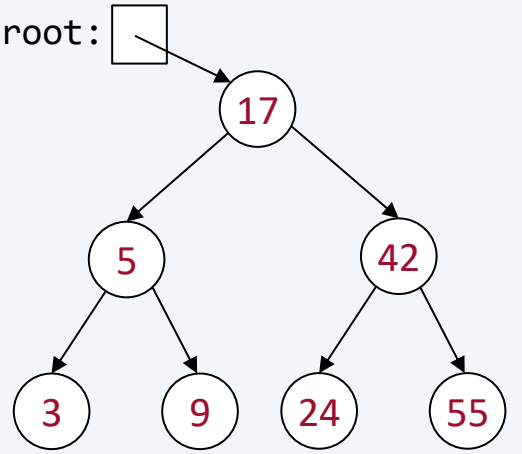
das binäre Suchen (*binary search*)

Voraussetzung für die Anwendung dieses Suchverfahrens ist allerdings, dass die Elemente im Behälter sortiert angeordnet sind, und ein direkter Zugriff auf das „mittlere“ Element möglich ist.

Sortiertes Feld:



Binärer Suchbaum:



Binäres Suchen

Lösungsidee

- Es wird geprüft, ob der gesuchte Schlüsselwert mit dem des mittleren Elementes übereinstimmt
- Wenn das nicht der Fall ist, kann wegen der Sortierung der Behälterelemente durch den Vergleich des gesuchten Schlüsselwerts mit dem Wert der Schlüsselkomponente des gerade betrachteten Elements feststellt werden, ob in der „linken Hälfte“ oder in der „rechten Hälfte“ des Behälters analog (ev. rekursiv) weitergesucht werden muss
- Dadurch vermindert sich nach jedem nicht erfolgreichen Suchschritt die Anzahl der noch zu durchsuchenden Elemente um die Hälfte, was zu einer signifikanten Reduktion der Suchschritte im Vergleich zur sequentiellen Suche führt

(1) Binäre Suche in einem Feld

Die Transformation der Lösungsidee des binären Suchens in einem als Feld organisierten (sortierten) Datenbestandes führt zu folgendem Algorithmus:

```
IndexOf(↓a: ArrayType ↓first: int ↓last: int ↓x: KeyType): int
  var
    mid: int
begin
  while first ≤ last do
    mid := (first + last) div 2
    if x = a[mid].key then
      return mid
    else
      if x < a[mid].key then
        last := mid - 1
      else -- x > a[mid].key
        first := mid + 1
      end -- if
    end -- if
  end -- while
  return min - 1
end IndexOf
```

(2) Binäre Suche in einem binären Suchbaum

Gegeben seien folgende Datentypen

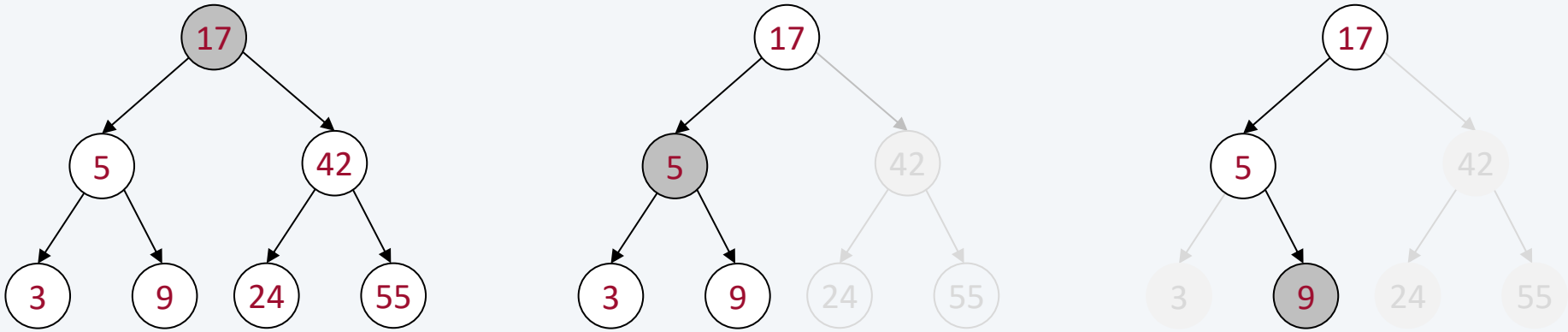
```
type
  TreeNodePtr = →TreeNode
  TreeNode = compound
    left, right: TreeNodePtr
    key: KeyType
    dataI: DataTypeI
  end -- TreeNode
  TreePtr = TreeNodePtr
```

Auf Basis dieser Deklarationen kann, wenn ein nach den Werten der Schlüsselkomponente (`key`) sortierter binärer Suchbaum `tree` vorliegt, ein einfacher rekursiver Algorithmus `NodeWith` formuliert werden, mit dem ein Knoten, dessen Komponente `key` den gesuchten Wert `x` enthält, gefunden werden kann

Binäre Suche in einem binären Suchbaum

Grundprinzip der Lösungsidee für den Algorithmus `NodeWith` ist, dass im jeweils betrachteten Baum `tree`, sofern dieser nicht leer ist, zunächst der Wurzelknoten untersucht wird, und wenn der gesuchte Wert darin nicht enthalten ist, die Suche durch einen rekursiven Aufruf entweder im linken (`tree→left`) oder im rechten Teilbaum (`tree→right`) fortgesetzt wird, je nachdem, ob der gesuchte Wert x kleiner oder größer als der Wert der Komponente key im Wurzelknoten ist

Beispiel: Suche nach Knoten mit dem Schlüsselwert x = 9



Binäre Suche in einem binären Suchbaum

Der Algorithmus NodeWith der diese Lösungsidee umsetzt lautet:

```
NodeWith(↓tree: TreePtr ↓x: KeyType): TreeNodePtr
begin
  if tree = null then
    return null
  elsif tree→key = x then
    return tree
  elsif x < tree→key then
    return NodeWith(↓tree→left ↓x)
  else -- x > tree→key
    return NodeWith(↓tree→right ↓x)
  end -- if
end NodeWith
```

Laufzeitkomplexität binärer Suchalgorithmen

Im günstigsten Fall: Die minimale Anzahl von Suchschritten ist ist $S_{min}(n) = 1$.

- Dieser Fall tritt ein, wenn der zu durchsuchende Bereich des Behälters nur ein Element umfasst oder der gesuchte Wert auf Anhieb im mittleren Element des Felds bzw. im Wurzelknoten des Suchbaums gefunden wird.

Im ungünstigsten Fall: Die maximale Anzahl von Suchschritten ist $S_{max}(n) = \lfloor \mathrm{ld}(n) \rfloor + 1$.

- Dieser Fall tritt ein, wenn der gesuchte Wert nicht im zu durchsuchenden Behälter enthalten ist oder erst im letzten besuchten Element/Knoten gefunden wird.

Im typischen oder durchschnittlichen Fall:

- Der Einfachheit halber betrachten wir den Spezialfall eines sortierten Felds oder eines balancierten binären Suchbaums mit $n = 2^k - 1$ (für $k > 1$) Elementen/Knoten und nehmen an, dass die n Elemente/Knoten des Behälters in ihren Schlüsselkomponenten unterschiedliche Werte enthalten.

Laufzeitkomplexität binärer Suchalgorithmen

- Die Gesamtzahl der Suchschritte S_{total}, die notwendig sind, um jedes Element bzw. jeden Knoten der beiden Behälter zu lokalisieren, beträgt:

$$S_{total}(2^4 - 1) = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 4 + 4 \cdot 8$$
$$= 1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 = 49$$

=> Grobanalyse, Kapitel 8!

- Verallgemeinern wir das Berechnungsergebnis (mit $n = 2^k - 1$ und n unterschiedlichen Schlüsselwerten) so erhalten wir:

$$S_{total}\left(2^k - 1\right) = \sum_{i=1}^{k} i \cdot 2^{i-1} = \ \ldots = (k-1) \cdot 2^k + 1$$

- Unter der Annahme, dass nach jedem Element des Behälters mit der gleichen Wahrscheinlichkeit gesucht wird, beträgt die durchschnittliche Anzahl von Suchschritten:

$$\mathrm{S}_{avg}\left(2^k - 1\right) = \frac{S_{total}\left(2^k - 1\right)}{2^k - 1} = \frac{(k-1) \cdot 2^k + 1}{2^k - 1}$$

- Für große k gilt $S_{avg}(n) \approx (k-1)$ $\quad (+1\ , -1$ vernachlässigen)

Laufzeitkomplexität binärer Suchalgorithmen

- Dieses Ergebnis bedeutet, dass sich das Laufzeitverhalten binärer Suchverfahren im durchschnittlichen Fall nur unwesentlich vom ungünstigsten Fall unterscheidet: nur etwa ein Suchschritt weniger wird benötigt.
- Die asymptotische Laufzeitkomplexität binärer Suchalgorithmen ist demnach $O(\mathrm{ld}\, n) = O(\log n)$, sie ist also logarithmisch.