

2. Einfache Suchverfahren

Lineare Listen

Sequentielle Suche

Binäre Suche

Weitere Suchverfahren auf sortierten Feldern

- Fibonacci-Suche
- Sprungsuche
- Exponentielle Suche
- Interpolationssuche

Auswahlproblem



Lineare Listen

Lineare Liste

- endliche Folge von Elementen eines Grundtyps (Elementtyps)
 $\langle a_1, a_2, \dots, a_n \rangle$ $n \geq 0$
 $n=0$: leere Liste $\langle \rangle$
- Position von Elementen in der Liste ist wesentlich

Typische Operationen

- INIT (L): Initialisiert L, d.h. L wird eine leere Liste
- INSERT (L, x, p): Fügt x an Position p in Liste L ein und verschiebt die Elemente an p und den nachfolgenden Positionen auf die jeweils nächsthöhere Position
- DELETE (L, p): Löscht das Element an Position p der Liste L
- ISEMPY (L): Ermittelt, ob Liste L leer ist
- SEARCH (L, x) bzw. LOCATE (L, x): Prüft, ob x in L vorkommt bzw. gibt die erste Position von L, in der x vorkommt, zurück
- RETRIEVE (L, p): Liefert das Element an Position p der Liste L zurück
- FIRST (L): Liefert die erste Position der Liste zurück
- NEXT (p, L) und PREVIOUS (p, L): Liefert Element der Liste, das Position p nachfolgt bzw. vorausgeht, zurück
- PRINTLIST (L): Schreibt die Elemente der Liste L in der Reihenfolge ihres Auftretens aus
- CONCAT (L1, L2): Hintereinanderfügen von L1 und L2



Lineare Listen (2)

Komplexität der Operationen abhängig von

- gewählter Implementierung sowie
- ob Sortierung der Elemente vorliegt

Wesentliche Implementierungsalternativen

- Sequentielle Speicherung (Reihung, Array)
- Verkettete Speicherung

Sequentielle Speicherung (Reihung, Array)

- statische Datenstruktur (hoher Speicheraufwand)
- wahlfreie Zugriffsmöglichkeit über (berechenbaren) Index
- 1-dimensionale vs. mehrdimensionale Arrays

Verkettete Speicherung

- dynamische Datenstruktur
- sequentielle Navigation
- Varianten: einfache vs. doppelte Verkettung etc.



Beispiel-Spezifikation in Java

Nutzung des Interface-Konzepts (ADT-Umsetzung)

- Interface: Sammlung abstrakter Methoden
- Implementierung erfolgt durch Klassen (instanzierbar)
- Klassen können mehrere Interfaces implementieren (Simulation der Mehrfachvererbung)
- mehrere Implementierungen pro Interface möglich

```
public interface Liste { // Annahme Schlüssel vom Typ int
    public void insert (int x, int p) throws ListException;
    public void delete (int p) throws ListException;
    public boolean isempty ();
    public boolean search (int x);
    ...
    public Liste concat (Liste L1, Liste L2);
}

public class ListException extends RuntimeException {
    public ListException (String msg) {super (msg); }
    public ListException () { }
}
```



Beispiel (2)

Einsatzbeispiel für Interface Liste

- Implementierung ArrayListe wird vorausgesetzt

```
public class ListExample {
    public static void main (String args[]) {
        Liste list = new ArrayListe ();

        try {
            for ( int i = 1; i <= 20; i++ ) {
                list.insert (i*i,i);
                System.out.println (i*i);
            }
            // ...
            if (! list.search (1000)) list.insert (1000,1);
        }
        catch (ListException exc) {
            System.out.println (exc);
        }
    }
}
```



Array-Realisierung linearer Listen

```
public class ArrayListe implements Liste {

    int[] L = null; // Spezialfall: Elemente vom Typ int
    int laenge=0;
    int maxLaenge;

    /* Konstruktoren */

    public ArrayListe(int max) {
        L = new int [max+1]; // Feldindex 0 bleibt reserviert
        maxLaenge = max;
    }
    public ArrayListe() { // Default-Größe 100
        L = new int [101]; // Feldindex 0 bleibt reserviert
        maxLaenge = 100;
    }
}
```



Array-Realisierung (2)

```
public void insert (int x, int pos) throws ListException {
    // Einfügen an Positon pos (Löschen analog)
    if (laenge == maxLaenge) throw new ListException("Liste voll!");
    else if ((pos < 1) || (pos > laenge + 1))
        throw new ListException("Falsche Position!");
    else {
        for (int i=laenge; i >= pos; i--) L[i+1] = L[i];
        L [pos] = x;
        laenge++;
    }
}
// weitere Methoden von ArrayListe
}
```

Einfüge-Komplexität für Liste mit n Elementen:



Sequentielle Suche

Suche nach Element mit Schlüsselwert x

Falls unbekannt, ob die Elemente der Liste nach ihren Schlüsselwerten sortiert sind, ist die Liste sequentiell zu durchlaufen und elementweise zu überprüfen (sequentielle Suche)

```
public boolean search (int x) {
    // Annahme: Methode eingebettet in Klasse ArrayListe
    int pos = 1;
    while ((pos <= laenge) && (L [pos] != x)) pos++;
    return (pos <= laenge);
}
```

Kosten

- erfolglose Suche erfordert n Schleifendurchläufe
- erfolgreiche Suche verlangt im ungünstigsten Fall n Schlüsselvergleiche (und n-1 Schleifendurchläufe)
- mittlere Anzahl von Schlüsselvergleichen bei erfolgreicher Suche:

$$C_{\text{avg}}(n) = \frac{1}{n} \cdot \sum_{i=1}^n i = \frac{n+1}{2}$$



Sequentielle Suche (2)

leichte Verbesserung: vereinfachte Schleifenbedingung durch Hinzufügen eines Stoppers bzw. Wächters (englisch "Sentinel") in Position 0 der Liste

```
public boolean searchSeqStopper(int x) {
    int pos = laenge;
    L[0] = x;
    while (L[pos] != x) pos--;
    return (pos > 0);
}
```



Binäre Suche

auf sortierten Listen können Suchvorgänge effizienter durchgeführt werden

- sequentielle Suche auf sortierten Listen bringt nur geringe Verbesserungen (für erfolgreiche Suche)
- Binärsuche wesentlich effizienter durch Einsatz der *Divide-and-Conquer-Strategie*

Suche nach Schlüssel x in Liste mit aufsteigend sortierten Schlüsseln:

- Falls Liste leer ist, endet die Suche erfolglos.
Sonst: Betrachte Element L [m] an mittlerer Position m
- Falls $x = L [m]$ dann ist das gesuchte Element gefunden.
- Falls $x < L [m]$, dann durchsuche die linke Teilliste von Position 1 bis m-1 nach demselben Verfahren
- Sonst ($x > L [m]$) durchsuche die rechte Teilliste von Position m+1 bis Laenge nach demselben Verfahren

Beispiel

Liste: 3 8 17 22 30 32 36 42 43 49 53 55 61 66 75



Binäre Suche (2)

Iterative Lösung

```
public boolean binarySearch(int x) {
    int pos;     /* aktuelle Suchposition
                 (enthält bei erfolgreicher Suche Ergebnis) */
    int ug = 1; // Untergrenze des aktuellen Suchbereichs
    int og = laenge; // Obergrenze des aktuellen Suchbereichs
    boolean gefunden = false;
    while ((ug <= og) && (! gefunden)) {
        pos = (ug + og) / 2;
        if (L[pos] > x)
            og = pos - 1; // linken Bereich durchsuchen
        else if (L[pos] < x)
            ug = pos + 1; // rechten Bereich durchsuchen
        else
            gefunden = true;
    }
    return gefunden;
}
```

Kosten

$$C_{\min}(n) = 1 \quad C_{\max}(n) = \lceil \log_2(n+1) \rceil \quad C_{\text{avg}}(n) \approx \log_2(n+1) - 1, \text{ für große } n$$



Suche auf Objekten

bisher Suche nach int-Schlüsseln

Verallgemeinerung auf Schlüssel (Objekte), auf denen Ordnung vorliegt und zwischen denen Vergleiche möglich sind

```
public interface Comparable {
    public boolean equals (Comparable o);
    public boolean less (Comparable o);
    public boolean greater (Comparable o);
    public boolean lessEqual (Comparable o);
    public boolean greaterEqual (Comparable o);
    public Comparable minKey ();
}

public class ComparableFloat implements Comparable {
    float key; // Schlüssel vom Typ float
    String wert; // weitere Inhalte
    ComparableFloat (float f) {this.key=f;} // Konstruktor
    public boolean equals (Comparable o) {
        return this.key==((ComparableFloat)o).key;}
    public boolean less (Comparable o) {
        return this.key < ((ComparableFloat)o).key;}
    ...
    public Comparable minKey () {
        return new ComparableFloat(Float.MIN_VALUE);}
}
```



```

public class ArrayListe2 implements Liste2 {
// Liste2 entspricht Liste, jedoch mit Orderable- statt int-Elementen
    Orderable[] L = null;
    int laenge=0;
    int maxLaenge;

    public ArrayListe2(int max) {
        L = new Orderable [max+1];
        maxLaenge = max;
    }

    public boolean search (Orderable x) {
        int pos = 1;
        while ((pos <= laenge) && (! L [pos].equals (x))) pos++;
        return (pos <= laenge);
    }

// weitere Operationen: binarySearch etc. ...
}

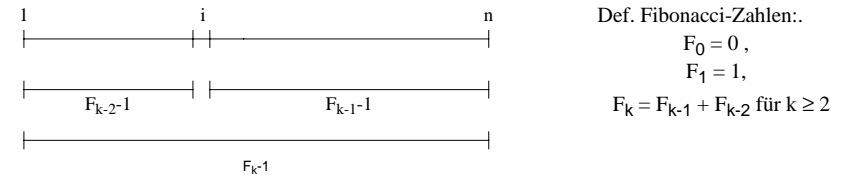
```



Fibonacci-Suche

ähnlich der Binärsuche, jedoch wird Suchbereich entsprechend der Folge der Fibonacci-Zahlen geteilt

Teilung einer Liste mit $n = F_k - 1$ sortierten Elementen:



- Element an der Position $i = F_{k-2}$ wird mit dem Suchschlüssel x verglichen
- Wird Gleichheit festgestellt, endet die Suche erfolgreich.
- Ist x größer, wird der rechte Bereich mit $F_{k-1} - 1$ Elementen, ansonsten der linke Bereich mit $F_{k-2} - 1$ Elementen auf dieselbe Weise durchsucht.

Kosten

- für $n = F_k - 1$ sind im schlechtesten Fall $k - 2$ Suchschritte notwendig, d.h. $O(k)$ Schlüsselvergleiche



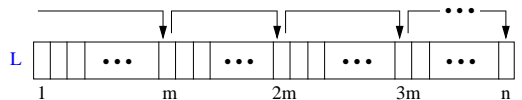
- Da gilt $F_k \approx c \cdot 1.618^k$, folgt $C_{\max}(n) = O(\log_{1.618}(n+1)) = O(\log n)$.



Sprungsuche

Prinzip

- der sortierte Datenbestand wird zunächst in Sprüngen überquert, um Abschnitt zu lokalisieren, der ggf. den gesuchten Schlüssel enthält
- danach wird der Schlüssel im gefundenen Abschnitt nach irgendeinem Verfahren gesucht



Einfache Sprungsuche

- konstante Sprünge zu Positionen $m, 2m, 3m \dots$
- Sobald $x \leq L[i]$ mit $i = j \cdot m$ ($j = 1, 2, \dots$) wird im Abschnitt $L[(j-1)m+1]$ bis $L[j \cdot m]$ sequentiell nach dem Suchschlüssel x gesucht.

mittlere Suchkosten

ein Sprung koste a ; ein sequentieller Vergleich b Einheiten $C_{\text{avg}}(n) = \frac{1}{2}a \cdot \frac{n}{m} + \frac{1}{2}b(m-1)$

optimale Sprungweite: $m = \sqrt{(a/b)n}$ bzw. $m = \sqrt{n}$

- falls $a=b \Rightarrow C_{\text{avg}}(n) = a\sqrt{n} - a/2$

Komplexität $O(\sqrt{n})$



Sprungsuche (2)

Zwei-Ebenen-Sprungsuche

- statt sequentieller Suche im lokalisierten Abschnitt wird wiederum eine Quadratwurzel-Sprungsuche angewendet, bevor dann sequentiell gesucht wird $\frac{1}{4} + \frac{1}{2} + \frac{1}{4}$
- Mittlere Kosten: $C_{\text{avg}}(n) \leq \frac{1}{2} \cdot a \cdot \sqrt{n} + \frac{1}{2} \cdot b \cdot n^{\frac{1}{4}} + \frac{1}{2} \cdot c \cdot n^{\frac{1}{4}}$
 - a Kosten eines Sprungs auf der ersten Ebene;
 - b Kosten eines Sprungs auf der zweiten Ebene;
 - c Kosten für einen sequentiellen Vergleich

Verbesserung durch optimale Abstimmung der Sprungweiten m_1 und m_2 der beiden Ebenen

- Mit $a = b = c$ ergeben sich als optimale Sprungweiten $m_1 = n^{\frac{2}{3}}$ und $m_2 = n^{\frac{1}{3}}$
- mittlere Suchkosten: $C_{\text{avg}}(n) = \frac{3}{2} \cdot a \cdot n^{\frac{1}{3}}$

Verallgemeinerung zu n-Ebenen-Verfahren ergibt ähnlich günstige Kosten wie Binärsuche (Übereinstimmung bei $\log_2 n$ Ebenen)

Sprungsuche vorteilhaft, wenn Binärsuche nicht anwendbar ist (z.B. bei blockweisem Einlesen der sortierten Sätze vom Externspeicher)

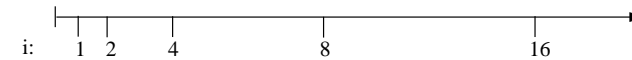


Exponentielle Suche

Anwendung, wenn Länge des Suchbereichs n zunächst unbekannt bzw. sehr groß

Vorgehensweise

- für Suchschlüssel x wird zunächst obere Grenze für den zu durchsuchenden Abschnitt bestimmt
`int i = 1; while (x > L[i]) i=2*i;`
- Für $i > 1$ gilt für den auf diese Weise bestimmten Suchabschnitt: $L[i/2] < x \leq L[i]$
- Suche innerhalb des Abschnitts mit irgendeinem Verfahren



enthält die sortierte Liste nur positive, ganzzahlige Schlüssel ohne Duplikate, wachsen Schlüsselwerte mindestens so stark wie die Indizes der Elemente

=> i wird höchstens $\log_2 x$ mal verdoppelt

- Bestimmung des gesuchten Intervalls erfordert maximal $\log_2 x$ Schlüsselvergleiche
- Suche innerhalb des Abschnitts (z.B. mit Binärsuche) erfordert auch höchstens $\log_2 x$ Schlüsselvergleiche

Gesamtaufwand $O(\log_2 x)$



Interpolationssuche

Schnellere Lokalisierung des Suchbereichs in dem Schlüsselwerte selbst betrachtet werden, um "Abstand" zum Suchschlüssel x abzuschätzen

nächste Suchposition pos wird aus den Werten ug und og der Unter- und Obergrenze des aktuellen Suchbereichs wie folgt berechnet:

$$pos = ug + \frac{x - L[ug]}{L[og] - L[ug]} \cdot (og - ug)$$

sinnvoll, wenn Schlüsselwerte im betreffenden Bereich einigermaßen gleichverteilt

- erfordert dann im Mittel lediglich $\log_2 \log_2 n + 1$ Schlüsselvergleiche

im schlechtesten Fall (stark ungleichmäßige Werteverteilung) entsteht jedoch linearer Suchaufwand ($O(n)$)



Auswahlproblem

Finde das i -kleinste Element in einer Liste L mit n Elementen

- Spezialfälle: kleinster Wert, größter Wert, mittlerer Wert (Median)

trivial bei sortierter Liste

unsortierter Liste: Minimum/Maximum-Bestimmung erfordert lineare Kosten $O(n)$

einfache Lösung für i -kleinstes Element:

- $j = 0$
- solange $j < i$: Bestimme kleinstes Element in L und entferne es aus L ; erhöhe j um 1
- gebe Minimum der Restliste als Ergebnis zurück

Komplexität:

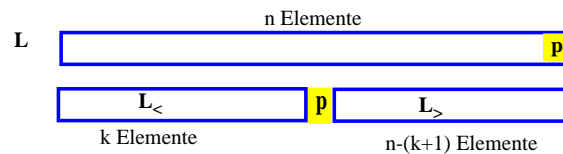
schneller: Sortieren + Auswahl



Auswahlproblem (2)

Divide-and-Conquer-Strategie (i-kleinstes Element von n paarweise unterschiedlichen Elementen)

- bestimme Pivot-Element p
- Teile die n Elemente bezüglich v in 2 Gruppen: Gruppe 1 enthält die k Elemente die kleiner sind als v ; Gruppe 2 die $n-k-1$ Elemente, die größer als p sind
- falls $i=k+1$, dann ist p das Ergebnis.
falls $i \leq k$ wende das Verfahren rekursiv auf Gruppe 1 an;
falls $i > k+1$: verwende Verfahren rekursiv zur Bestimmung des $i - (k+1)$ -te Element in Gruppe 2



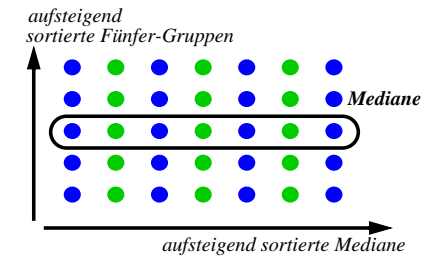
Laufzeit:



Auswahlproblem (3)

Lösung des Auswahlproblems mit linearen Kosten: Median-of-Median-Strategie:

- falls $n < \text{Konstante}$, berechne i -kleinstes Element direkt und beende Algorithmus
Sonst:
- teile die n Elemente in $n/5$ Gruppen zu je 5 Elementen und höchstens eine Gruppe mit höchstens vier Elementen auf -> *lineare Kosten*
- sortiere jede dieser Gruppen (in konstanter Zeit) und bestimme in jeder Gruppe das mittlere Element (Median) -> *lineare Kosten*
- wähle Verfahren rekursiv auf die Mediane an und bestimme das mittlere Element p (Median der Mediane) -> Pivot-Element
- Teile die n Elemente bezüglich p in 2 Gruppen: Gruppe 1 enthält die k Elemente die kleiner sind als p ; Gruppe 2 die $n-k-1$ Elemente, die größer als p sind
- falls $i=k+1$, dann ist p das Ergebnis.
falls $i \leq k$ wende das Verfahren rekursiv auf Gruppe 1 an;
falls $i > k+1$: verwende Verfahren rekursiv zur Bestimmung des $i - (k+1)$ -te Element in Gruppe 2



Zusammenfassung

Sequentielle Suche

- Default-Ansatz zur Suche
- lineare Kosten $O(n)$

Binärsuche

- setzt Sortierung voraus
- Divide-and-Conquer-Strategie
- wesentlich schneller als sequentielle Suche
- Komplexität: $O(\log n)$

Weitere Suchverfahren auf sortierten Arrays für Sonderfälle

- Fibonacci-Suche
- Sprungsuche
- exponentielle Suche
- Interpolationssuche

Auswahlproblem auf unsortierter Eingabe mit linearen Kosten lösbar

