Datenstrukturen

Adrian Schüpbach
adrian_laurent.schuepbach@alumni.ethz.ch

Worum geht es?

- Datenstrukturen verstehen
  - Warum ist es relevant, die richtige Datenstruktur zu wählen?
  - Wie funktionieren dann eigentlich Datenstrukturen?
  - → Eigene Datenstrukturen programmieren können
- Datenstrukturen in Java kennen
  - Java hat bereits Datenstrukturen
  - → Diese nützen können
  - Richtige Datenstruktur wählen mit gelerntem Hintergrundwissen

Worum geht es?

- Datenstrukturen verstehen
  - Warum ist es relevant, die richtige Datenstruktur zu wählen?
  - Wie funktionieren dann eigentlich Datenstrukturen?
  - → Eigene Datenstrukturen programmieren können
- Datenstrukturen in Java kennen
  - Java hat bereits Datenstrukturen
  - → Diese nützen können
  - Richtige Datenstruktur wählen mit gelerntem Hintergrundwissen

- Ziele:
  - Eigenschaften verschiedener Datenstrukturen kennen → Bessere Wahl
  - Deshalb anschauen, wie sie funktionieren
  - Später Java-Datenstrukturen verwenden, nicht jedesmal selbst implementieren

Richtige Datenstruktur wählen: Motivation

- Gegeben: Grosse Textdatei
- Gesucht: Wie oft kommt gegebenes Wort vor?
  - Annahme: Programm weiss nicht, welches Wort Benutzer will
  - Ansonsten könnte Programm gezielt nur dieses Wort zählen

Richtige Datenstruktur wählen: Motivation

- Gegeben: Grosse Textdatei
- Gesucht: Wie oft kommt gegebenes Wort vor?
  - Annahme: Programm weiss nicht, welches Wort Benutzer will
  - Ansonsten könnte Programm gezielt nur dieses Wort zählen
- → Demo

Richtige Datenstruktur wählen: Motivation

Fünf Varianten von Datenstrukturen

Datenstruktur	Ladezeit	Abfragezeit pro Element
String-Array	2201ms	33.002ns
LinkedList	4758ms	84.343ns
Hashtable	497ms	0.234ns
HashMap	183ms	0.12ns
TreeMap	141ms	0.456ns

Dynamische Datenstrukturen vs. Array fixer Grösse

Element in Array “einfügen”

Dynamische Datenstrukturen vs. Array fixer Grösse

Element in Array “einfügen”

1. Einfüge-Index bestimmen

Dynamische Datenstrukturen vs. Array fixer Grösse

Element in Array “einfügen”

1. Einfüge-Index bestimmen
2. Dieses Element und alle folgenden Elemente um eine Position verschieben

Dynamische Datenstrukturen vs. Array fixer Grösse

Element in Array “einfügen”

1. Einfüge-Index bestimmen
2. Dieses Element und alle folgenden Elemente um eine Position verschieben
   - Schleife von hinten beginnend
   - Letztes Element um eine Position verschieben
   - Zweitletztes Element wird zu letztem Element
   - ...

Dynamische Datenstrukturen vs. Array fixer Grösse

Element in Array "einfügen"

1. Einfüge-Index bestimmen
2. Dieses Element und alle folgenden Elemente um eine Position verschieben
   - Schleife von hinten beginnend
   - Letztes Element um eine Position verschieben
   - Zweitletztes Element wird zu letztem Element
   - ...
3. Neues Element in entstandene Lücke einfügen

Dynamische Datenstrukturen vs. Array fixer Grösse

Element in Array “einfügen”

1. Einfüge-Index bestimmen
2. Dieses Element und alle folgenden Elemente um eine Position verschieben
   - Schleife von hinten beginnend
   - Letztes Element um eine Position verschieben
   - Zweitletztes Element wird zu letztem Element
   - ...
3. Neues Element in entstandene Lücke einfügen

- Voraussetzung: Array hat genügend freie Plätze
- → Fixe Grösse, kann nicht wachsen

Dynamische Datenstrukturen vs. Array fixer Grösse

- Vorteile Array
  - Schneller Zugriff an gewünschter Stelle
  - Einfach zu programmieren
  - Kaum Metadaten
- Nachteile Array
  - Fixe Grösse
  - Element “einfügen” ist langsam

- Vorteile dynamischer Datenstrukturen
  - Variable Grösse
  - Optimierbar für Zugriff/einfügen/löschen/suchen...
- Nachteile dynamischer Datenstrukturen
  - Aufwendigere Programmierung
  - Metadaten nötig

Verkettete Liste

- Verkettete Liste (linked list) ähnlich wie Array
- Elemente nacheinander
- Wichtige Unterschiede zu Array
  - Kein Index
  - Kann nicht direkt an bestimmte Position springen

Verkettete Liste

Konstruktion

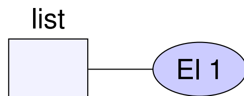
- Verkettete Liste aus Objekten aufgebaut
  - Variable zeigt auf erstes Objekt
  - Objekt zeigt auf nächstes Objekt
  - Letztes Objekt zeigt nirgendwo hin (`null`)
- Eigene Klasse `class ListenObjekt` erstellen

Verkettete Liste

Elemente einfügen

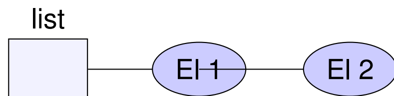
Verkettete Liste

Elemente einfügen



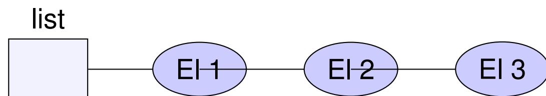
Verkettete Liste

Elemente einfügen



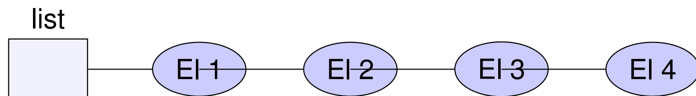
Verkettete Liste

Elemente einfügen



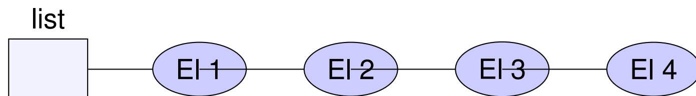
Verkettete Liste

Elemente einfügen



Verkettete Liste

Elemente einfügen



- Einfügen langsam
- Einfügezeit steigt linear mit der Anzahl Elemente

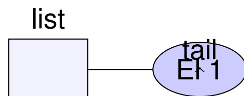
Verkettete Liste

Elemente einfügen: Optimierung: Tail-Zeiger

- Mit Tail-Zeiger Einfügezeit am Ende der Liste schnell

Verkettete Liste

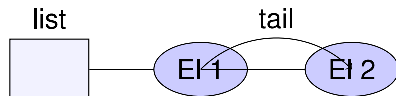
Elemente einfügen: Optimierung: Tail-Zeiger



- Mit Tail-Zeiger Einfügezeit am Ende der Liste schnell

Verkettete Liste

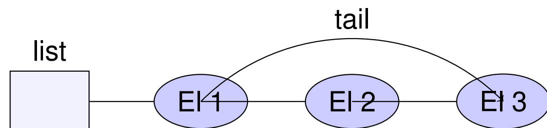
Elemente einfügen: Optimierung: Tail-Zeiger



- Mit Tail-Zeiger Einfügezeit am Ende der Liste schnell

Verkettete Liste

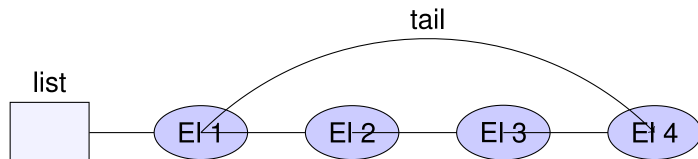
Elemente einfügen: Optimierung: Tail-Zeiger



- Mit Tail-Zeiger Einfügezeit am Ende der Liste schnell

Verkettete Liste

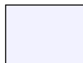
Elemente einfügen: Optimierung: Tail-Zeiger



- Mit Tail-Zeiger Einfügezeit am Ende der Liste schnell

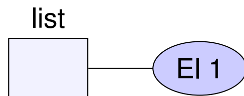
Verkettete Liste

Elemente einfügen: Alternative: Vorne einfügen



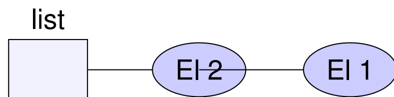
Verkettete Liste

Elemente einfügen: Alternative: Vorne einfügen



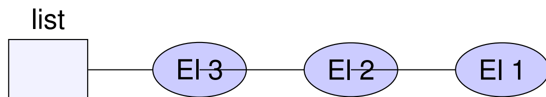
Verkettete Liste

Elemente einfügen: Alternative: Vorne einfügen



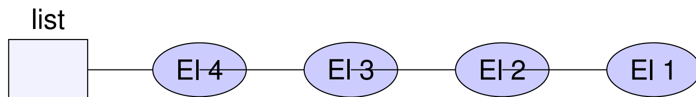
Verkettete Liste

Elemente einfügen: Alternative: Vorne einfügen



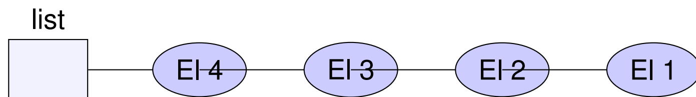
Verkettete Liste

Elemente einfügen: Alternative: Vorne einfügen



Verkettete Liste

Elemente einfügen: Alternative: Vorne einfügen



- Vorne einfügen schnell

Verkettete Liste

Iterative Liste

```
class ListeIt {
    Element head; Element tail;
    class Element {
        Element next;   int wert;
        Element (int wert) {
            this.wert = wert;
        }
    }
    public void add(int wert) {
        Element e = new Element(wert);
        if (head == null) {
            head = e;   tail = e;
        } else {
            tail.next = e;  tail = e;
        }
    }
```

Verkettete Liste

Iterative Liste: Elemente anzeigen

```
class ListeIt {
    // ...

  public void show() {
    for (Element e = head; e != null; e = e.next) {
        System.out.println("Wert = " + e.wert);
    }
  }
}
```

Verkettete Liste

Rekursive Liste

```
class ListeRek {
    ListeRek next;  int wert;
    ListeRek(int wert) {
        this.wert = wert;
    }
    public void add(int wert) {
        if (next == null) {
            next = new ListeRek(wert);
        } else {
            next.add(wert);
        }
    }
    // ...
}
```

Verkettete Liste

Rekursive Liste: Elemente anzeigen

```
class ListeRek {
    // ...
    public void show() {
        System.out.println("Wert = " + wert);
        if (next != null) {
            next.show();
        }
    }
}
```

Verkettete Liste

Eigenschaften

- Dynamisch, keine fixe Grösse
- Vorne und hinten einfügen und wegnehmen schnell
- Einfügereihenfolge bleibt erhalten
  - Schleife durch alle Elemente → richtige Reihenfolge
- Mitte einfügen und löschen möglich, langsam

Stack

Stack

- Stack (Stapel) kann Elemente stapeln

Stack

- Stack (Stapel) kann Elemente stapeln
- Funktionen
  - Kann neues Element auf den Stapel legen (push)
  - Kann oberstes Element anschauen (peek)
  - Kann oberstes Element wegnehmen (pop)

Stack

- Anwendung: Klammerung überprüfen
  - Öffnende runde Klammer passt zu schliessender runder Klammer
  - Genauso: Eckige Klammern, geschweifte Klammern, spitze Klammern

Stack

- Anwendung: Klammerung überprüfen
  - Öffnende runde Klammer passt zu schliessender runder Klammer
  - Genauso: Eckige Klammern, geschweifte Klammern, spitze Klammern
- Algorithmus

Stack

- Anwendung: Klammerung überprüfen
  - Öffnende runde Klammer passt zu schliessender runder Klammer
  - Genauso: Eckige Klammern, geschweifte Klammern, spitze Klammern
- Algorithmus
  1. Leeren Stack erzeugen
  2. Öffnende Klammer: Klammer auf den Stack setzen
  3. Schliessende Klammer: Klammer vom Stack holen & prüfen, ob es gleicher Typ ist

Stack

Klammerung testen

```
public class Klammerung {
  public static void main(String[] args) throws Exceptio
    StackImpl s = new StackImpl();
    String text = "17 * ((3 + 3) * {37 - (2* 99)})";
    for (int i = 0; i < text.length(); i++) {
      switch(text.charAt(i)) {
        case '(': s.push(KLAMMER_RUND); break;
        case '{': s.push(KLAMMER_GESCHWEIFT); break;
        case ')':
          if (s.pop() != KLAMMER_RUND) {
            throw new KlammerException(i);
          } break;
        case '}':
          if (s.pop() != KLAMMER_GESCHWEIFT) {
            throw new KlammerException(i);
          } break;
```

Stack

Klammerung testen

```
class KlammerException extends Exception {
    int pos;
    public KlammerException(int wert) {
      pos = wert;
    }
    public String getMessage() {
      return ("Klammerfehler an Position "+pos);
    }
}
```

Stack

- Anwendung: Entscheidung revidieren

Stack

- Anwendung: Entscheidung revidieren
- Treffe Entscheidung, weiss nicht, ob sie richtig ist

Stack

- Anwendung: Entscheidung revidieren
- Treffe Entscheidung, weiss nicht, ob sie richtig ist
- Lösung:
  1. Treffe Entscheidung
  2. Merke Kontext
  3. Speichere Kontext auf Stack
  4. Falls unzufrieden, nehme letzte Entscheidung vom Stack
  5. Entscheide neu
  6. Falls unzufrieden, kann 2, 3, ... Entscheidungen vom Stack nehmen

Stack

- Anwendung: Entscheidung revidieren
- Treffe Entscheidung, weiss nicht, ob sie richtig ist
- Lösung:
  1. Treffe Entscheidung
  2. Merke Kontext
  3. Speichere Kontext auf Stack
  4. Falls unzufrieden, nehme letzte Entscheidung vom Stack
  5. Entscheide neu
  6. Falls unzufrieden, kann 2, 3, ... Entscheidungen vom Stack nehmen
- Beispiel: Weg finden
  - Schlage Weg ein
  - Falls kein Ausgang → Gehe zurück
  - Schlage neuen Weg ein
  - Im Extremfall zurück zur ersten Kreuzung

Stack

Implementation

- Stack im wesentlichen verkettete Liste
  - Elemente vorne einfügen
  - Elemente vorne wegnehmen
- Gleiche Effizienz-Eigenschaften wie verkettete Listen

Stack

Stack-Implementation

```
public class StackImpl {
    Element head;
    class Element {
        int wert;   Element next;
    }
    public void push(int wert) {
        Element e = new Element();
        e.wert = wert;
        e.next = head;
        head = e;
    }
    int pop() {
        int w = head.wert;
        head = head.next;
        return (w);
    }
```

Stack

Implementation

- Java: `LinkedList`
  - `push`
  - `peek`
  - `pop`
- Java: `Stack`: Spezialisiert, lässt wirklich nur Element-Operationen an der “Spitze” zu
  - `push`
  - `peek`
  - `pop`
- Variationen → Java-API

Warteschlange (Queue)

Warteschlange (Queue)

- Elemente stehen hinten an/hinten einfügen
- Vorderstes Element wird weggenommen und bearbeitet

Warteschlange (Queue)

- Elemente stehen hinten an/hinten einfügen
- Vorderstes Element wird weggenommen und bearbeitet
- Eigenschaften
  - Bearbeitungsreihenfolge == Einfügereihenfolge
  - Hinten einfügen schnell
  - Vorne wegnehmen schnell

Warteschlange (Queue)

Implementation

- Queue kann als verkettete Liste implementiert werden
- Neue Elemente hinten einfügen
- Elemente vorne wegnehmen

- → Java: `LinkedList`
  - `addLast()`
  - `getFirst()`
- Variationen → Java-API

Warteschlange (Queue)

Anwendungen

- Puffer:
  - Prozess 1 generiert Daten
    - Datei einlesen
    - Vom Netzwerk empfangen
  - Prozess 2 bearbeitet Daten
  - Prozess 1 fügt Datn in Queue ein
  - Prozess nimmt Daten aus der Queue

Warteschlange (Queue)

Anwendungen

- Simulation
  - Elemente stehen für etwas in echter Welt
  - Ein Element kann durch Programm teilweise bearbeitet werden
  - Danach wieder in Queue einfügen
  - → Gerecht, Elemente kommen immer wieder nacheinander an an die Reihe

Warteschlange (Queue)

Anwendungen

- Nachrichten/Ereignisse
  - Nachrichten oder Ereignisse treten zeitlich nacheinander auf
  - “Leser” soll sie in gleicher Reihenfolge lesen
  - → Queue sammelt Nachrichten in richtiger Reihenfolge

Zusammenfassung listenbasierte Datenstrukturen

Zusammenfassung listenbasierte Datenstrukturen

- Einfach zu implementieren
- Vorne und hinten einfügen und löschen effizient
- Schleife über alle Elemente einfach
- Suchen, sortieren, in Mitte einfügen/löschen langsam

Binärer Baum

Binärer Baum

- Ziel: Elemente schnell suchen können

Binärer Baum

- Ziel: Elemente schnell suchen können
- Wie findet man etwas schnell?

Binärer Baum

- Ziel: Elemente schnell suchen können
- Wie findet man etwas schnell?
- → Wenn es sortiert ist

Binärer Baum

- Daten als Baum organisiert (statt verketteter Liste)
- Element zeigt auf *zwei* nächste Elemente
  - Zwei Kinder

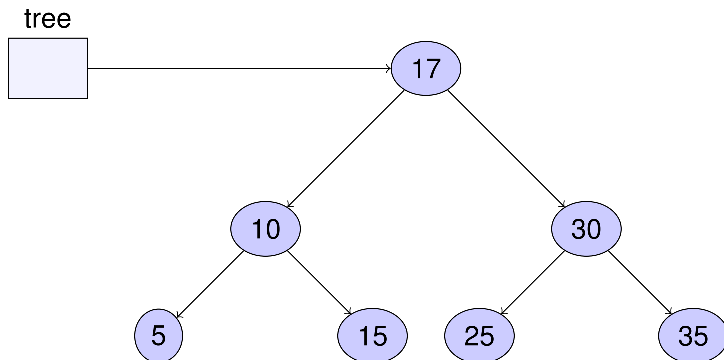
Binärer Baum

- Daten als Baum organisiert (statt verketteter Liste)
- Element zeigt auf *zwei* nächste Elemente
  - Zwei Kinder
- Linkes Kind kleiner
- Rechtes Kind grösser

Binärer Baum

- Daten als Baum organisiert (statt verketteter Liste)
- Element zeigt auf *zwei* nächste Elemente
  - Zwei Kinder
- Linkes Kind kleiner
- Rechtes Kind grösser
- → Elemente müssen vergleichbar sein

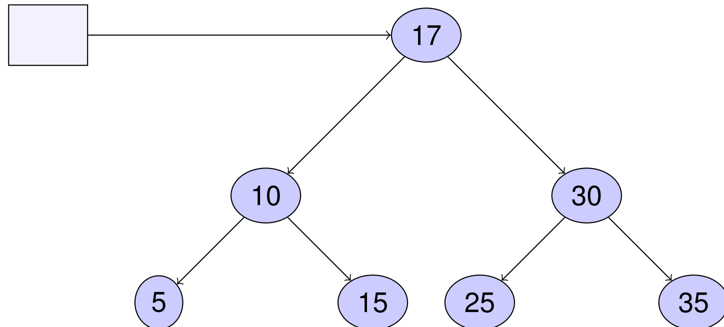
Binärer Baum



- Linkes Kind kleiner ⇒ Linke Baumhälfte kleiner

Binärer Baum

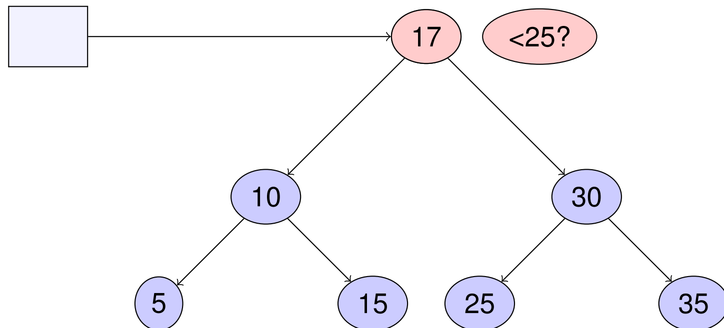
Element suchen



- Element **25** suchen: 3 Vergleiche:

Binärer Baum

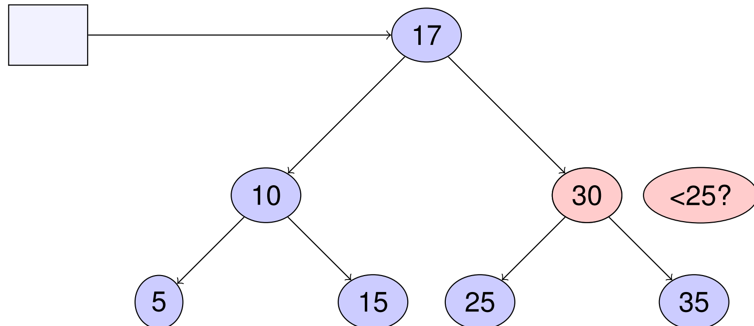
Element suchen



- Element **25** suchen: 3 Vergleiche: 17

Binärer Baum

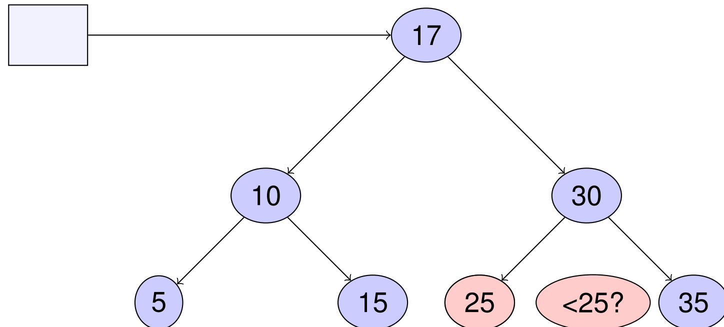
Element suchen



- Element **25** suchen: 3 Vergleiche: 17 , 30

Binärer Baum

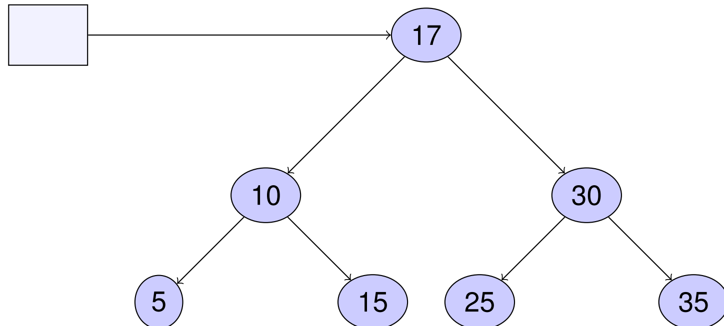
Element suchen



- Element **25** suchen: 3 Vergleiche: 17 , 30 , 25

Binärer Baum

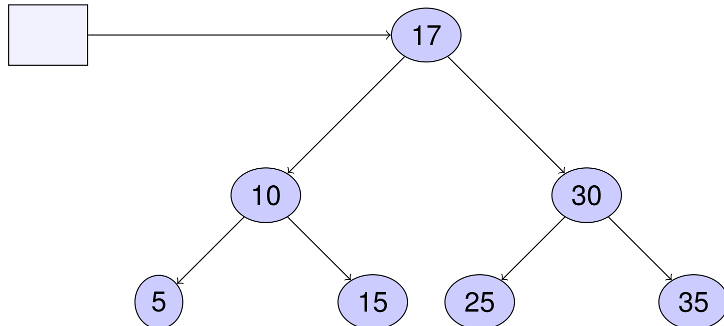
Element suchen



- Element **25** suchen: 3 Vergleiche: 17 , 30 , 25
- 8 Elemente

Binärer Baum

Element suchen



- Element **25** suchen: 3 Vergleiche: 17 , 30 , 25
- 8 Elemente
- $2^3 = 8 \Rightarrow log_2(n)$ Vergleiche

Binärer Baum

Eigenschaften

- Dynamisch, keine fixe Grösse
- Einfügen, suchen, löschen schnell ($log_2(n)$)
- Höhere Anzahl Elemente $\rightarrow$ nur logarithmischer Anstieg der Zeit

Binärer Baum

Eigenschaften

- Dynamisch, keine fixe Grösse
- Einfügen, suchen, löschen schnell ($log_2(n)$)
- Höhere Anzahl Elemente → nur logarithmischer Anstieg der Zeit

- → Java-API

Hashtable

Hashtable

- Ziel: Sehr schnell einfügen und suchen

Hashtable

- Ziel: Sehr schnell einfügen und suchen
- Prinzip:
  - Tabelle fixer Grösse
  - Datenmenge ungefähr bekannt
  - Hashwert als Index in Tabelle berechnen
  - Direkter Zugriff auf Tabelle via Index

Hashtable

Hashwert

- Hashwert = Kurzdarstellung/Fingerabdruck der Daten
  - Verschiedene Techniken möglich
  - Resultat: Zahl mit fixer Anzahl Stellen
- Hashwert nicht eindeutig, weil nur Kurzdarstellung

Hashtable

Hashwert

- Beispiel: Ganzzahldivision mit Rest
  - Zahl wird dividiert
  - Keine Nachkommastellen
  - Rest, der nicht dividiert werden kann = Hashwert
  - → Nennt man “Modulo-Division” (mod)
  - Java: mod → %

Hashtable

Hashwert

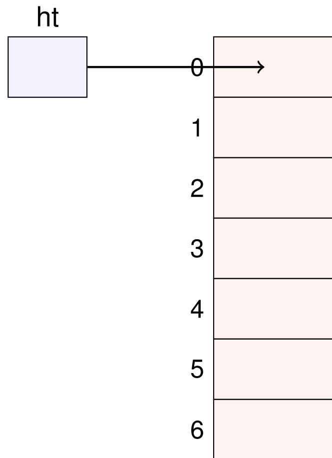
- Beispiel: Ganzzahldivision mit Rest
  - Zahl wird dividiert
  - Keine Nachkommastellen
  - Rest, der nicht dividiert werden kann = Hashwert
  - → Nennt man “Modulo-Division” (mod)
  - Java: mod → %
- Beispiele: Division durch 19
  - 33 % 19 = 14
  - 238 % 19 = 10

Hashtable

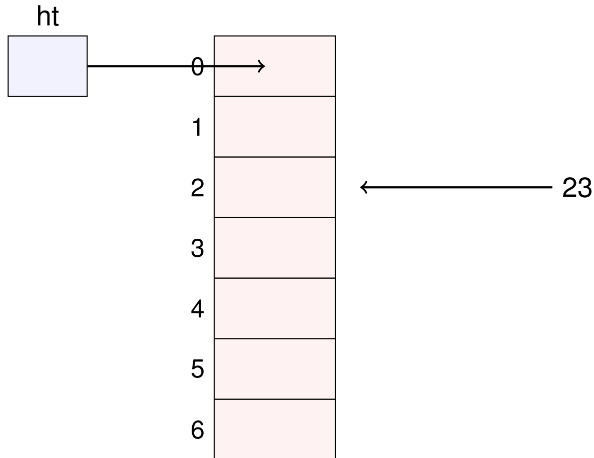
Hashwert

- Hashwert von allen Datentypen berechenbar
  - Zahlen
  - Text
  - Objekte
  - ...
- → Kann beliebige Objekte in Hashtable einfügen

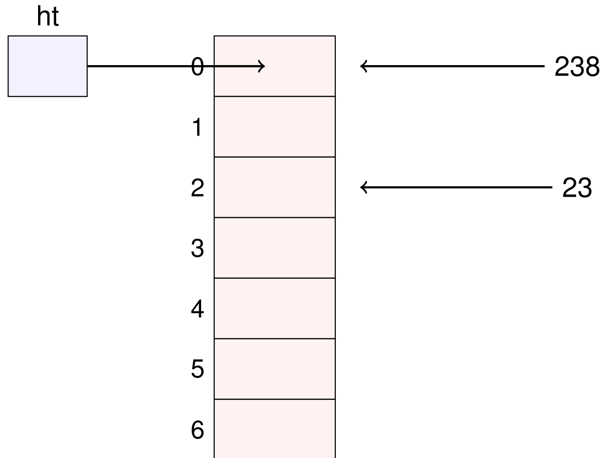
Hashtable



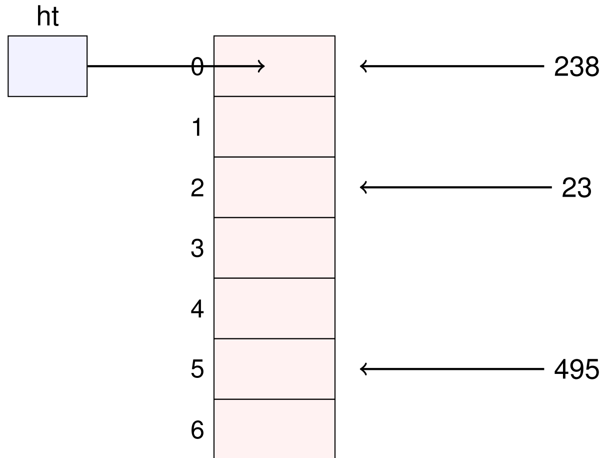
Hashtable



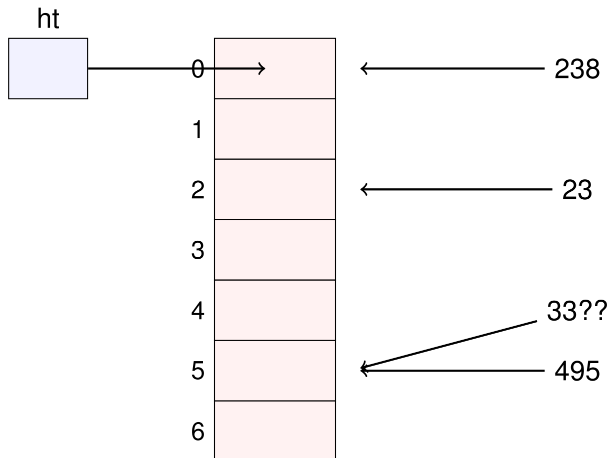
Hashtable



Hashtable



Hashtable



Hashtable

Eigenschaften

- Einfügen, suchen und löschen schnell
  - Direktzugriff an richtiger Position (wie Array)
  - Keine grossen Lücken, wenn Daten Lücken haben (!= Array)
  - Keine Sortierung
  - Optimal: Hashtable $\frac{1}{4}$ bis $\frac{3}{4}$ gefüllt

Hashtable

Eigenschaften

- Einfügen, suchen und löschen schnell
  - Direktzugriff an richtiger Position (wie Array)
  - Keine grossen Lücken, wenn Daten Lücken haben (!= Array)
  - Keine Sortierung
  - Optimal: Hashtable $\frac{1}{4}$ bis $\frac{3}{4}$ gefüllt
- Kollisionen
  - Daten mit gleichem Hashwert an gleiche Stelle in Hashtable
  - Braucht Strategie (Liste, zweiter Hashwert, ..)
  - Hashwert nicht genug, muss Daten vergleichen

Hashtable

Anwendungen

- Schnelle Datensuche
- Sich merken, ob man Elemente schonmal gesehen hat
- Zwei Datensätze:
  - Testen, ob daten des einen Datensatzes im anderen Datensatz enthalten sind
  - $\rightarrow$ Datenbank
- Benutzen, wenn Sortierung egal ist

Hashtable

Java

- `Hashtable`
- `HashMap`
- `HashSet`
- $\rightarrow$ Java-API

Min/Max-Heap

Min/Max-Heap

- Will wissen, welches Element aktuell kleinsten (grössten) Wert hat
- Abfragen soll sehr schnell sein
- Einfügen, löschen soll schnell sein

Min/Max-Heap

- Min-Heap (Max-Heap) ist Baum
- Wurzel (oberstes Element) hat immer kleinsten (grössten) Wert
- Beim Einfügen und beim Löschen wird Baum aktualisiert
  - Wurzel ist immer kleinstes (grösstes) Element

Min/Max-Heap

Eigenschaften

- Dynamisch, keine fixe Grösse
- Kleinstes (grösstes) Element abfragen: Direkter Zugriff
  - Da direkter Zugriff auf Wurzel
  - Muss nicht suchen
- Einfügen, löschen $log_2 n$
  - $\rightarrow$ Baum
  - Sortiert

Min/Max-Heap

Anwendungen

- “Rangliste” von Elementen
- Element mit kleinstem (grösstem) Wert muss zuerst bearbeitet werden
- Zugriff auf kleinstes (grösstes) Element muss sehr schnell sein
- Muss aktuell kleinstes (grösstes) Element, während Baum modifiziert wird, abfragen können

Allgemeinster Element-Typ

Anwendungen

Allgemeinster Element-Typ

Anwendungen

- Eigene Datenstrukturen für `int` implementiert (ListenObjekt, Stack)
- Was aber, wenn ich `String` einfügen will?

Allgemeinster Element-Typ

Anwendungen

- Eigene Datenstrukturen für `int` implementiert (ListenObjekt, Stack)
- Was aber, wenn ich `String` einfügen will?
- $\rightarrow$ Java nimmt Elemente des allgemeinsten Typs `Object` entgegen
- Beim Auslesen bekommt man `Object`
  - Muss es zurückcasten in originalen Typ

Cast

Cast: String s = (String)o;

```
public class Programm56 {
  public static void main(String[] args) {
    LinkedList lll = new LinkedList();
    lll.add(new String("bla"));

    // Cast Object -> String
    String s = (String)lll.getFirst();
  }
}
```

Zusammenfassung Datenstrukturen

Datenstrukturen in Java

Datenstruktur in Java	Typ/Zweck
`LinkedList`	Verkettete Liste
`Hashtable`	Hashtable
`HashMap`	Hashtable, nicht synchronisiert
`HashSet`	Hashtable, speichert nur Schlüssel
`TreeMap`	Baum
`TreeSet`	Baum, speichert nur Schlüssel
`Stack`	Stack/Stapel
`PriorityQueue`	Min-Heap → Schnell Minimum finden

Auch wenn fast alle add() haben und somit austauschbar sind, ist Geschwindigkeitsunterschied je nach Anwendung erheblich!