TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa / Arnd Poetzsch-Heffter

AG Softech
FB Informatik
TU Kaiserslautern

Überblick

- Weitere Sortierverfahren
  - Merge Sort
  - Heap Sort
- Praktische Auswirkungen der Laufzeitabschätzungen
- Abstrakte Datentypen
  - Listen
  - Stapel
  - Warteschlangen

Merge Sort

Sortieren durch Mischen (*merge sort*)

Algorithmische Idee (Divide-and-Conquer Strategie):

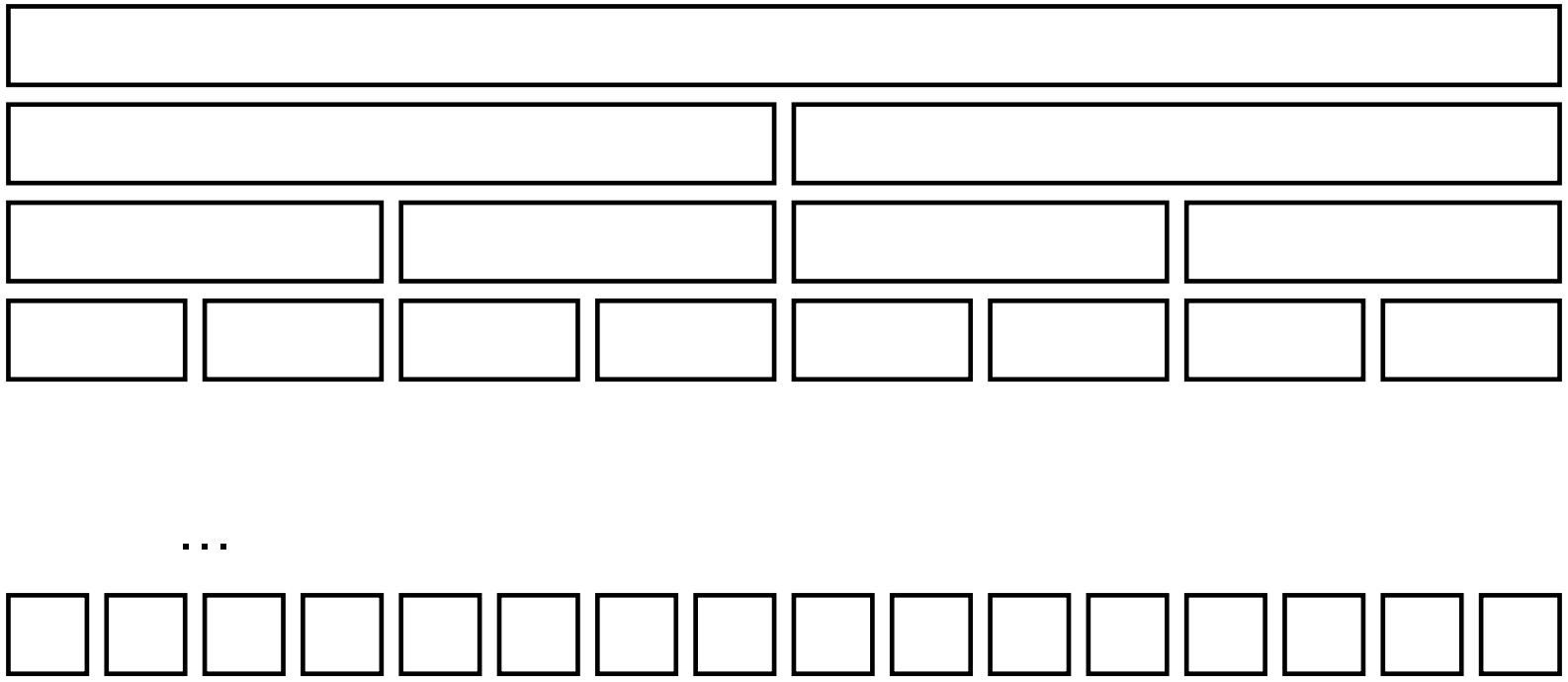
1. Hat die zu sortierende Folge mehr als ein Element, teile diese in zwei gleich große Teile auf (bzw. mit einem Größenunterschied von 1).
2. Sortiere die zwei Teilfolgen.
3. Füge die zwei sortierten Teilfolgen nun wieder zu einer sortierten Folge zusammen.

Implementierung auf Arrays:

- Kopiere die sortierten Teilarrays in ein *Hilfsarray*.
- Verwende dieses, um die sortierten Teilfolgen wieder zusammenzufügen.

Laufzeitabschätzung I

Struktur des Aufrufbaums:
(Annahme: $N = 2^n$ für $n \in \mathbb{N}$)



Wie oft wird MergeSort rekursiv verschachtelt aufgerufen?
$\Rightarrow \log N$, da bei jedem rekursiven Aufruf das jeweilige Teilarray halbiert wird

Laufzeitabschätzung II

In jedem Merge werden die Schlüssel des Teilarrays genau zweimal kopiert (in das Hilfsarray und zurück). Die Anzahl der Schlüsselvergleiche beträgt insgesamt pro Schlüssel max. $\log N$.

Schlüsselvergleiche: $C(N) \in O(N \log N)$

Datensatzzuweisungen: $M(N) \in O(N \log N)$

Bemerkungen

- Es wird ein Hilfsarray benötigt, d.h. bei der Implementierung auf Arrays wird zusätzlich Speicherplatz für ein Array von gleicher Größe wie die Eingabe reserviert.
- Gut geeignet für die Sortierung von verketteten Listen ($\Rightarrow$ siehe Übung)

Heap Sort

Weitere Anwendungen von Sortiertechniken

Fragestellung:

Gegeben N Datensätze, wie finde ich die Einträge mit den M größten Schlüsseln?

Aus der Praxis:

- Top 1000 der besten Spielerergebnisse eines Online-Games
- Aufgaben mit Prioritäten in einer Todo-List
- Messergebnisse bei Experimenten

→ Ergebnisse evtl. nicht alle direkt verfügbar (Größe N zunächst unbekannt)

→ Datensätze sollten nacheinander hinzugefügt werden können

Lösungsansatz

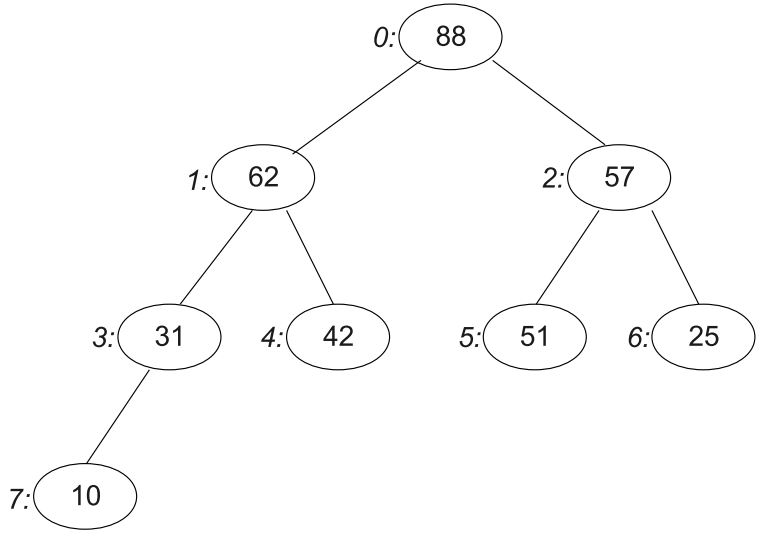
Entferne aus den Datensätzen M-mal den jeweils größten Datensatz!

Datenstrukturen zur Datenrepräsentierung

- Ungeordnetes Array
  - Annahme: Arraygröße ist $\geq N$
  - Hinzufügen von neuen Elementen $O(1)$
  - Entfernen des Maximums $O(N)$
- Geordnetes Array
  - Annahme: Arraygröße ist $\geq N$
  - Entfernen des Maximums $O(1)$
  - Hinzufügen von neuen Elementen $O(N)$ (Insertion Sort)
- Sortierte Verlinkte Liste
  - ähnlich wie bei geordnetem Array

Datenstruktur Heap

Idee: Verwende einen Binärbaum, bei dem der Schlüssel in jedem Knoten größer oder gleich der Schlüssel in den direkten Kindknoten ist (**Heap-Eigenschaft**)



Die Heap-Eigenschaft garantiert, dass der Schlüssel eines Knotens größer gleich *aller* Schlüssel in den Unterbäumen ist.

Insbesondere steht in der Wurzel ein Element mit einem maximalen Schlüssel.

Bemerkung

- Achtung: Der Begriff *Heap* ist in der Informatik überladen!
- Auch der Speicher für zur Laufzeit angelegte Variablen wird im Englischen *heap* genannt.

Adressierung der Knoten

- Speichern der Datensätze in einem Array statt aufwendiger Verlinkung der Knoten
- Arithmetik auf den Arrayindizes zur Navigation zwischen Knoten
- Zu einem Knoten k befindet sich
  - der linke Kindknoten an Position $2(k+1)-1$
  - der rechte Kindknoten an Position $2(k+1)$
- Das Array darf keine "Lücken" haben.

Begriffsklärung: Vollständiger Baum

Die **Tiefe** eines Knotens ist die Länge des Pfades von der Wurzel bis zu dem Knoten.

Die **Höhe** eines Baumes ist um eins größer als die maximalen Tiefe der Knoten.

Ein Binärbaum der Höhe h heißt **vollständig**, wenn jeder Knoten mit maximaler Tiefe $(h-1)$ ein Blatt ist und jeder Knoten mit geringerer Tiefe zwei (nicht-leere) Unterbäume hat.

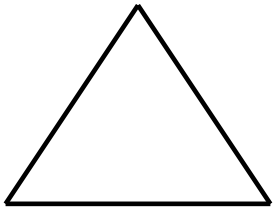
Begriffsklärung: Fast vollständiger Baum

Ein Binärbaum der Höhe h heißt **fast vollständig**, wenn

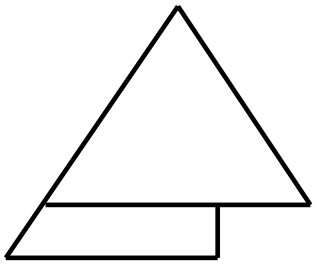
- jedes Blatt die Tiefe $h-1$ oder $h-2$ hat,
- jeder Knoten mit einer Tiefe kleiner $h-2$ zwei (nicht-leere) Unterbäume hat, und
- für die Knoten k des Niveaus $h-2$ gilt:
  1. Hat k zwei nicht-leere Unterbäume, dann auch alle linken Nachbarn (Knoten gleicher Tiefe) von k.
  2. Ist k ein Blatt, dann sind auch alle rechten Nachbarn von k Blätter.
  3. Es gibt maximal ein k mit genau einem nicht-leeren linken Unterbaum.

Strukturskizze:

Vollständiger Baum



Fast vollständiger Baum



Begriffsklärung: Heap

Ein markierter, fast vollständiger Binärbaum mit n Knoten, die mittels Indizes direkt referenziert werden können, heißt **Heap** der Größe n, wenn die Heap-Eigenschaft erfüllt ist:

Ist i ein Knoten und j ein Kind von i, so gilt für die Markierungen der Knoten:

$$m(i) \geq m(j)$$

Herstellen der Heap-Eigenschaft

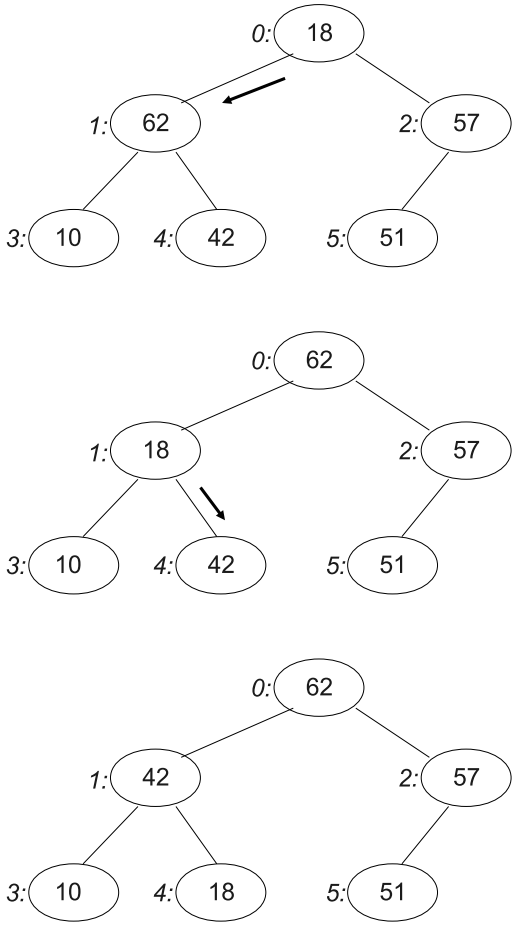
Gegeben sei ein markierter, fast vollständiger Binärbaum der Größe n, der die Heap-Eigenschaft nur an einem Knoten k verletzt.

Die Heap-Eigenschaft kann hergestellt werden, indem man den Knoten k *versickern* lässt.

Idee: Vertausche den Knoten mit dem größeren der beiden Kind-Knoten.

- Gibt es kein Kind, gibt es nichts zu tun (Heap-Eigenschaft gilt!)
- Gibt es genau ein Kind l, dann ist dies links und kinderlos:
  Falls $m(k) < m(l)$, vertausche die Markierungen.
  Danach gilt die Heap-Eigenschaft wieder.
- Gibt es zwei Kinder l und r und ist j das Kind mit dem größeren Schlüssel:
  Ist $m(k) < m(j)$, vertausche die Markierungen und fahre rekursiv mit dem Versickern fort.

Beispiel: Versickern lassen



Indexberechnung

```
// Index des linken Kindknotens im Array
static int leftChild(int k) {
  return 2 * (k +1) - 1;
}

// Index des rechten Kindknotens im Array
static int rightChild(int k) {
  return 2 * (k +1);
}
```

Implementierung

```
static void sink (DataSet[] f, int n, int k) {
  if (leftChild(k) < n) {     // Gibt es mind. ein Kind?
    int l = leftChild(k) ;    // Index des linken Kinds
    int r = rightChild(k);    // Index des rechten Kinds

    int j = l;
    if (r < n && f[l].key < f[r].key) {
    // Falls es zwei Kinder gibt und
    // r den groesseren Schluessel hat
      j = r;
    }

    if (f[k].key < f[j].key) {
      swap(f, k, j);
      sink(f, n, j);
    }
  }
}
```

Heapsort

Sortieren mit Hilfe der Heap-Datenstruktur

- 1. Schritt: Erstelle einen Heap auf dem Eingabearray
- 2. Schritt:
  - Tausche Maximumelement (Wurzelelement) mit dem aktuell letzten Element des Heaps
  - Versickere das neue Wurzelelement
  - Fahre mit Schritt 2 fort, bis der Heap leer ist

Optimierung für Schritt 1

Die initiale Heap-Erzeugung erfolgt von rechts nach links durch die Erzeugung von Heaps auf den kleinen Teilbäumen.

Blätter (d.h. Teilbäume der Größe 1) erfüllen die Heap-Eigenschaft direkt, daher können diese bei der Konstruktion übersprungen werden.

Lemma:
In einem fast vollständigen Binärbaum der Größe n haben die Blätter die Indizes $(n \text{ div } 2)$ bis $n - 1$.

HeapSort

```
static void sort(DataSet[] f) {
  int size = f.length;

  //  Schritt 1: Herstellen der Heap-Bedingung
  for (int i = size/2 - 1; i >= 0; i--) {
    sink(f, size, i);
  }

  //  Schritt 2: Sortieren
  while (size > 0) {
    size--;
    swap(f, 0, size);
    sink(f, size, 0);
  }
}
```

Grobe Laufzeitabschätzung von Heapsort I

Sei C die Anzahl der Vergleichsoperationen,
M die Anzahl der Vertauschungen und
N die Größe der Eingabe.

Wir betrachten hier nur den ungünstigsten Fall.

Grobe Laufzeitabschätzung von Heapsort II

Herstellen der Heap-Eigenschaft

Sei j die Anzahl der Niveaus im Heap, also $2^{j-1} \leq N \leq 2^j - 1$.

Dann gibt es auf Niveau k höchstens 2^k Schlüssel und C_{max} und M_{max} sind proportional zu $j - k$.

Insgesamt gilt dann für die Anzahl der Operationen zur Herstellung der Heap-Eigenschaft:

$$\sum_{k=1}^{j-1} 2^k(j-k) = \sum_{i=1}^{j-1} 2^{j-i} i = 2^j \sum_{i=1}^{j-1} \frac{i}{2^i} \leq 2N * 2 \in O(N)$$

Grobe Laufzeitabschätzung von Heapsort III

Auswahl des Wurzelelements und Versickern

Da die Höhe eines fast vollständigen Binärbaums von Ordnung $O(\log N)$ ist, führt `sink` $O(\log N)$ Operationen aus.

Damit ergibt sich für das Versickern aller Elemente die Komplexität $O(N \log N)$.

Komplexität des gesamten Algorithmus':

$$O(N \log N)$$

Übersicht: Laufzeitverhalten der Sortieralgorithmen

Die Laufzeiten im "worst case" entsprechen folgenden Komplexitätsklassen:

Selection Sort	$O(N^2)$
Insertion Sort	$O(N^2)$
Quicksort	$O(N^2)$
Merge Sort	$O(N \log N)$
Heap Sort	$O(N \log N)$

- In der Praxis ist Quicksort (mit verfeinerten Pivotwahl) ein sehr effizienter und schneller allgemeiner Sortieralgorithmus.

Abstrakte Datentypen

Abstrakte Datentypen

- Ein **abstrakter Datentyp (ADT)** ist i.A. eine Menge von Elementen (bzw. Objekten) zusammen mit den Operationen, die für die Elemente der Menge charakteristisch sind.
- ADTs können unabhängig von ihrer konkreten Implementierung in verschiedenen Kontexten eingesetzt werden, einzig basierend auf einer wohldefinierten Schnittstelle.
- Java unterstützt abstrakte Datentypen durch das Interface-Konzept.
- Die Spezifikation der Operationen und der dabei verwendeten Datentypen ist durch die Signaturen der Methoden gegeben.
- Die Semantik kann entweder in einer formale Spezifikationssprache definiert oder zumindest informell durch Kommentare beschrieben werden.

Abstrakter Datentyp: Liste

```
public interface IntList {

  // Haengt ein Element an das Ende der Liste an
  void add(int element);

  // Liefert das Element an Position index
  int get(int index);

  // Entfernt das Element an Position index
  // Liefert true im Erfolgsfall, sonst false
  boolean remove(int index);

  // Anzahl der Elemente
  int size();
}
```

Konkrete Implementierungen des ADTs sind z.B. einfachverkettete Listen, doppeltverkette Listen, Array-Listen.

Abstrakter Datentyp: Stapel (engl. *stack*)

- Basiert auf dem LIFO-Prinzip (*Last-in-First out*)
- Das Element, welches zuletzt eingefügt wurde, wird als erstes wieder entfernt.
- Anwendungsbeispiel: Verwaltung von Hyperlinks im Browser (Back-Button), verschachtelter Methodenaufruf

Spezifikation: Stapel

```
public interface StackOfInt {

  // Testet, ob der Stapel leer ist.
  boolean empty();

  // Legt Element oben auf den Stapel.
  void push(int item);

  // Gibt das oberste Element vom Stapel zurueck.
  // Wirft eine EmptyStackException, falls der Stapel leer ist.
  int peek() throws EmptyStackException;

  // Gibt das oberste Element vom Stapel zurueck
  // und entfernt es vom Stapel.
  // Wirft eine EmptyStackException, falls der Stapel leer ist.
  int pop() throws EmptyStackException;
}
```

Stack-Implementierung mit Arrays

```
public class ArrayStackOfInt implements StackOfInt {
  private int[] elems;
  private int n;

  public ArrayStackOfInt(int max) {
    this.elems = new int[max];
  }
  public boolean empty() {
    return n == 0;
  }
  public void push(int item) {
      elems[n] = item;
      n++;
  }
  public int peek() throws EmptyStackException {
    if (empty()) throw new EmptyStackException();
    return elems[n-1];
  }
  public  int pop() {
    int result = this.peek();
    n--;
    return result;
  }
}
```

Diskussion

- Laufzeitabschätzung: Wie viele Operationen benötigt man, um ein Element einzufügen bzw. zu löschen?
- Was sind die Nachteile dieser Implementierung?

Stack-Implementierung mit einfach verketteten Listen

```
class Node {
  int item;
  Node next;
  Node(int item, Node next) { ... }
}

public class ListStackOfInt implements StackOfInt {
  private Node first;

  public ListStackOfInt() { }

  public boolean empty() {
    return (first == null);
  }
  public void push(int item) {
    Node n = new Node(item, first);
    first = n;
  }
  public int peek() throws EmptyStackException {
    // TODO
  }
  public  int pop() throws EmptyStackException {
    // TODO
  }
}
```

Warteschlange (engl. *queue*)

- Basiert auf dem FIFO-Prinzip (*First-in-First-out*)
- Anwendungsbeispiele: Musik-Playlists, Warteschlangen beim Einkaufen, Beantworten von Server-Anfragen, Druckaufträge
- Aspekt der Fairness!

Spezifikation: Queue

```
public interface QueueOfInt {
  // Fuegt ein Element hinten in die Warteschlange an.
  void add(int e);

  // Gibt das erste Element in der Warteschlange zurueck.
  int element();

  // Gibt das erste Element in der Warteschlange zurueck
  // und entfernt es aus der Warteschlange.
  int remove();

  // Testet, ob die Warteschlange leer ist.
  boolean empty();
}
```

Queue-Implementierung mit einfachverketteten Listen I

```
public class ListQueueOfInt implements QueueOfInt {
  private Node first;   // Element mit laengster Verweildauer
  private Node last;    // Neuestes Element

  // Testet, ob die Warteschlange leer ist.
  public boolean empty() {
    return (first == null);
  }

  // Fuegt ein Element hinten in die Warteschlange an.
  public void add(int e) {
    Node n = new Node(e,null);
    if (empty()) {
      first = n;
    } else {
      last.next = n;
    }
    last = n;
  }
```

Queue-Implementierung mit einfachverketteten Listen II

```
  // Fortsetzung...

  // Gibt das erste Element in der Warteschlange zurueck.
  public int element() {
    return first.elem;
  }

  // Gibt das erste Element in der Warteschlange zurueck
  // und entfernt es aus der Warteschlange.
  public int remove() {
    int result = first.elem;
    first = first.next;
    if (empty()) {
      last = null;
    }
    return result;
  }
}
```

Ausblick

- Und wenn ich eine Liste/Stack/Queue mit Strings, Students, beliebigen Objekten haben will?
  $\Rightarrow$ Parametrisierbare Datentypen in Java (Java Generics)

- Weitere Datentypen zum Verwalten von Objekt-Sammlungen: Java Collections