

Die Höhe eines Baumes ist die Anzahl der Ebenen. Das Prinzip der binären Suche entspricht einem Baum, denn es halbiert sich immer weiter auf dem Weg nach unten.

Drei Ebenen der Betrachtung der Datenstruktur:

- Spezifikationsebene: Ein Datentyp besteht aus einer speziellen Sorte, der Signatur, der Zuordnung von Wertebereichen zu den Sorten und der Spezifikation der Operationen
- Algorithmische Ebene: Die zu dem Datentyp zugehörige Datenstruktur legt die Repräsentation der Objekte fest. Weiterhin wird für jede Operation des Datentyps ein Algorithmus angegeben.
- Programmierungsebene: Die Datenstruktur wird nun in einer konkreten Programmiersprache (bei uns Java) implementiert. Die Datenstruktur wird in eine Klasse umgesetzt.

1. Spezifikationsebene:

- a. Welche Werte sollen repräsentiert werden?
- b. Welche Operationen sollen unterstützt werden, Was sollen die Operatoren leisten?
- c. Bei der Spezifikation des Datentyps sind alle Operationen Funktionen

Datentyp Intset

Sorten Intset, int, boolean;

Operationen empty => Intset

...

Wertebereiche Intset: Menge der ganzen Zahlen

2. Algorithmische Ebene:

- a. Wie sollen die Objekte der Sorte Intset repräsentiert werden?
- b. Wie können die in der Datenstruktur aufgeführten Operationen realisiert werden (Angabe eines Algorithmus)?

3. Programmierungsebene: Implementierung in Java:

- a. Eine Datenstruktur wird in Java durch eine Klasse implementiert
- b. Zusätzlich besteht in Java die Möglichkeit durch eine sogenannte Schnittstellenklasse (Interface) einen Datentyp zu spezifizieren.

Interface:

- i. Alle Methoden einer Schnittstellenklasse besitzen keine Implementierung. In Java spricht man dann auch von abstrakten Methoden.
- ii. Schnittstellenklassen können als Obergrenze von anderen Klassen benutzt werden. Die von der Schnittstellenklasse abgeleiteten Klassen müssen jede Methode der Schnittstellenklasse implementieren.
- iii. Schnittstellenklassen können mehrere Unterklassen besitzen, die jeweils eine andere Implementierung der gewünschten Funktionalität liefern.

- Eine Schnittstellenklasse entspricht einer „Klasse“, die nur aus abstrakten Methoden (ohne Rumpf) und Konstanten besteht.

Vorteil von Schnittstellen:

- Schnittstellen können von mehreren Klassen implementiert werden
- Je nach Anwendung kann eine Objektvariable der Schnittstellenklasse ein Objekt jener Klassen zugewiesen bekommen, welche die Schnittstelle implementiert
- Im folgenden Beispiel wird eine Objektvariable der Schnittstellenklasse Intset verwendet:

```
public static void main (String[]args) throws IOException
{
    Intset S;
    Random r = new Random();
    If (args.length == 0)
        S= new ArrayIntset();
    else {
        if (args[0].equals("list"))
            S = new ListIntset();
        else
            S = new ArrayIntset();
    }
    for (int I = 0; I < 20; I++)
        S.insert(r.nextInt() % 1000);
    System.in.read();
}
```

eine als Public deklarierte Methode kann auch außerhalb der Klasse genutzt werden.

Implements Intset zeigt an, dass diese Klasse alle Methoden der Schnittstellenklasse Intset implementiert.

QS:

- Testen, ob die Prozeduren tatsächlich die Spezifikation der Algorithmen erfüllen.
 - ⇒ Testprozeduren
 - ⇒ Von verschiedenen Personen
- Validierung der Analyse von den Algorithmen

Sequentielle Speicherung:

- Abgeleitet von der Darstellung für Mengen (Intset)
- Listenelemente werden in aufeinanderfolgenden Zellen eines Arrays gespeichert
- Menge der Referenzen entspricht dem Index des Arrays ({1,2,...} und 0)

Nachteil der sequentiellen Speicherung:

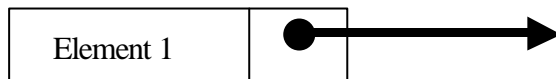
- hoher Aufwand beim Einfügen, Löschen und Suchen
- die Positionszeiger der Folgeelemente sind nicht stabil

Vorteil der sequentiellen Speicherung

- einfache Implementierung

Verkettete Speicherung:

- Aufbau einer rekursiven Datenstruktur
 - ⇒ Ein Knoten der Datenstruktur enthält dabei neben dem Listenelement eine Referenz auf den nächsten Knoten der Liste



- ⇒ Datenrepräsentation in Java

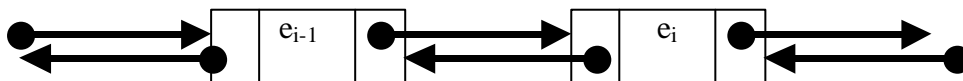
```
class Knoten {  
    ListElement elem;  
    Knoten next;  
    ...  
}
```

- Die Datenrepräsentation einer Liste kann bezüglich Anfang und Ende der Liste und der Darstellung variieren

Vorteil im Vergleich zur Implementierung mit Arrays:

- Positionszeiger der Elemente sind stabil

Doppelte Verkettung:



Vorteil:

- konstanter Aufwand für das Löschen

Geordnete lineare Listen

- zusätzliche Anforderung: Daten sollen in der Liste aufsteigend sortiert sein
=> neuer Datentyp

Sequentielle Speicherung:

- Liste wird wiederum durch ein Array implementiert
- Aufwand bei der Suche ist $O(\log n)$
- Aufwand für das Einfügen und Löschen ist immer noch $O(n)$

Interpolationssuche:

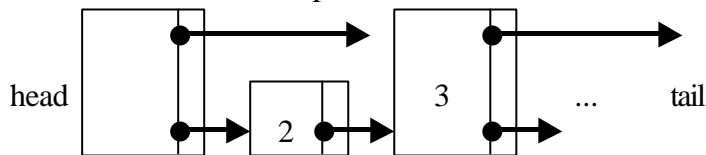
- Ersetzen des Faktors $\frac{1}{2}$ durch die erwartete Position, wo der Eintrag steht
- Beispiel: die Suche im Telefonnummernbuch ist am Anfang

Laufzeit der Interpolationssuche:

- im Durchschnitt ist Interpolationssuche effizienter als binäre Suche $O(\log \log n)$ im schlimmsten Fall benötigt die Interpolationssuche aber $O(n)$

Verkettete Speicherung: Skip-Listen

- Beschleunigung der Suche in einer verketteten Liste, indem wir jedes zweite Element in einer zweiten Liste abspeichern



- ⇒ Aufwand bei der Suche verbessert sich um den Faktor 2
- ⇒ Jedes zweite Element wird mit einem zusätzlichen Zeiger bestückt

Nachteil perfekter Skip-Listen:

- Der Aufwand beim Einfügen eines neuen Elementes beträgt im schlimmsten Fall (n)
- Fügen wir z.B. das Element 1 in die obere Skip-Liste ein, würde sich die Höhe bei sämtlichen Knoten ändern

Randomisierte Skip-Listen:

- Aufgabe der Forderung, dass zwei benachbarte Elemente der Höhe h genau ein Element der Höhe $h-1$ liegen muss
- es wird das Ziel verfolgt, dass das Verhältnis zwischen der Anzahl der Elemente, die zu verschiedenen Höhen der Skip-Liste gehören etwa dem einer perfekten Skipliste entsprechen
- Beim Einfügen eines neuen Elementes x in einen Knoten p wird die Höhe von p per Zufall bestimmt

Eigenschaften:

- Bei Eingabe der gleichen Daten in der gleichen Einfügereihenfolge in zwei leere Skip-Listen werden mit hoher Wahrscheinlichkeit verschiedene Strukturen entstehen

Performance von Skip-Listen:

- im schlimmsten Fall: Leistung entspricht der von einer linearen verketteten Liste (unwahrscheinlich)
- im Durchschnitt (abhängig von den Eingabewerten):
 - ⇒ logarithmische Suchkosten
 - ⇒ logarithmischer Aufwand beim Einfügen und Löschen
 - ⇒ linearer Speicheraufwand
 - ⇒ Beim Vergleich zwischen Skip-Listen und verketteten Listen sind Skip wesentlich (20*) schneller

Adaptive Listen:

- häufig gesuchte Element werden an den Anfang der Liste gespeichert. Somit kann dadurch die erfolgreiche Suche verbessert werden
- Strategien:
 - ⇒ Move-To-Front (MF): Beim Zugriff auf eine Element (erfolgreiche Suche) wird dieses an den Anfang der Liste bewegt.
 - ⇒ Transpose (T): Vertausche ein Element (nach erfolgreicher Suche) mit seinem Vorgänger
 - ⇒ Frequency Count (FC): Ordne jedem Element ein Häufigkeitszähler zu, der anfangs 0 ist und die Anzahl der Elementzugriffe (durch erfolgreiche Suche) zählt. Liste wird absteigend bzgl. den Häufigkeitszählern geordnet ist.

Algorithmus benutzt zwei Stacks:

- wird ein Operator gelesen, so wird dieser auf den Operatorstack geschrieben
- Wird eine Zahl gelesen, so wird diese auf den Operandenstack geschrieben

Hashverfahren:

- Hashverfahren dienen zur effizienten Implementierung von Mengen
- Die wichtigsten Operationen eines allgemeinen Datentyps Menge:
 - ⇒ Insert
 - ⇒ Delete
 - ⇒ contains
- **Ziele bei Hashverfahren: Operationen effizienter als durch Skip-Listen zu unterstützen**

Jeder Datensatz $R = \langle K, \text{info} \rangle$ besteht aus einem Schlüssel K und einem Informationsteil info .

- der Schlüssel bestimmt, wo der Datensatz abgespeichert wird. Wir werden deshalb auch häufig nur von Schlüsseln statt von Datensätzen sprechen
- Den Wertebereich eines Schlüssels bezeichnet D

Adressen werden auch als Hashadressen und die Menge der Behälter als Hashtabelle bezeichnet.

Terminologie:

- Ein Datensatz $\langle K, \text{info} \rangle$ nennt man Synonym, falls beim Einfügen der Behälter $B_{h(K)}$ bereits ein Element enthält. Man spricht dann von einer Kollision.
- Sei M eine Menge von n Datensätzen mit $n \leq m$ fest gegeben. Dann nennen wir eine Hashfunktion perfekt für M , falls es keine Kollision gibt.
- Belegungsfaktor $\alpha = n/m$
 N = aktuelle Anzahl der Datensätze in der Hashtabelle
 M = Größe der Hashtabelle

Anforderungen an das Hashverfahren:

- Anforderungen an Hashverfahren
 - ⇒ Hoher Belegungsfaktor
 - ⇒ Bei beliebiger Dateneingabe soll die Hashfunktion
 - Perfekt
 - Und effizient berechenbar sein

Offene Hashverfahren:

- jeder Hashadresse können beliebig viele Datensätze zugeordnet werden
⇒ Überläufer sind Datensätze, die nicht in ihrem Behälter abgespeichert werden können
- Überläufer werden bei Hashverfahren mit Verkettung in einem separaten Speicherbereich verwaltet.

Analyse:

- im schlimmsten Fall werden alle Datensätze auf einer Hashtabelle abgebildet. Somit hat man im Vergleich zu einer Liste nichts gewonnen.
- Bei einer erfolglosen Suche wird komplett eine Liste durchlaufen, die zu einer Hashadresse gehört. Da jede Hashadresse mit gleicher Wahrscheinlichkeit aufgesucht wird, ist die erwartete Anzahl von Zugriffen auf Schlüssel bei der erfolglosen Suche gleich $n/m = \acute{a}$

Beurteilung:

- niedriger Aufwand beim Suchen, Einfügen und Löschen bei großem \acute{a}

Nachteile:

- zusätzlicher Speicherplatz für Synonyme (auch wenn in der eigentlichen Hashtabelle noch vorhanden ist)
- zusätzlicher Speicherplatz für Zeiger
- Worst-case für alle Operationen (O)n

Separate Verkettung (seperate chaining):

- ist sehr ähnlich zur direkten Verkettung. Statt des Kopfs einer Liste wird nun der erste Datensatz einer Hashadresse in der Hashtabelle abgespeichert und die Überläufer in einer sogenannten Überlaufliste gehalten.

Vergleich zur direkten Verkettung:

- Wenn in einer Zelle nur ein Element abgelegt ist, benötigt separate Verkettung nur einen Verweis auf ein Listenelement (Überlaufliste)
Die direkte Verkettung braucht dagegen zwei Verweise
- Wenn auf eine Hashadresse h kein Datensatz abgebildet wurde, ist bei separater Verkettung trotzdem Speicherplatz für einen Datensatz allokiert.

Geschlossene Hashverfahren:

- Überläufer werden nicht in einem separaten Speicherbereich, sondern in der eigentlichen Hashtabelle gespeichert
- Ist beim Einfügen eines Datensatzes $\langle K, \text{info} \rangle$ der Speicherplatz $h(K)$ in der Hashtabelle schon belegt, so muss eine freie Stelle gefunden werden.

Prinzipielle Idee:

- Für jeden Schlüssel K gibt es eine Reihenfolge, in der die Speicherplätze der Hashtabelle auf Belegung geprüft werden. Diese Reihenfolge wird auch die Sondierungsreihenfolge genannt.
- Verfahren unterscheiden sich durch die Sondierungsfolge:
 - ⇒ Lineares Sondieren
 - ⇒ Quadratisches Sondieren
 - ⇒ Zufälliges Sondieren (Double-Hashing)

Primäre Clusterung:

Unter einem Cluster verstehen wir eine zusammenhängende Menge von belegten Behältern in der Hashtabelle.

Primäre Clusterung:

- Ein neuer Datensatz, dessen Hashadresse im Cluster liegt, muss nun linear die Sondierungsreihenfolge bis zum Ende des Clusters verfolgen.
- Danach wird er in den Behältern direkt vor (hinter) dem Cluster eingefügt; was zu Vergrößerung des Clusters führt.

Vermeidung von primärer Clusterung:

- quadratisches Sondieren ist effizienter als lineares Sondieren

Sekundäre Clusterung:

- Datensätze mit gleicher Hashadresse besitzen auch bei quadratischen Sondieren die gleiche Sondierungsreihenfolge.
Dieser Effekt wird Sekundäre Clusterung bezeichnet.
- Der negative Effekt von sekundärer Clusterung ist bei weitem geringer als der von primärer Clusterung.
Es stellt sich hier die Frage, ob es sich überhaupt lohnt, diesen theoretischen Makel zu beseitigen.

Double Hashing:

Vermeidung von primärer und sekundärer Clusterung

- Idealerweise (uniformes Sondieren):
Insgesamt gibt es $m!$ (ideale) Sondierungsreihenfolgen. Ein Schlüssel wählt mit Wahrscheinlichkeit $1/m!$ eine davon aus. Die Auswahl geschieht unabhängig von der anderer Schlüssel (insbesondere unabhängig von der Auswahl der Synonyme).

Hybrides Hashing:

- Beurteilung bisheriger Verfahren:
 - ⇒ Verfahren mit Verkettung sind wesentlich schneller als Verfahren mit Sondieren
 - ⇒ Nachteil der Verfahren mit Verkettung ist aber, dass zusätzlich Speicherplatz allokiert wird, obwohl Teile der Hashtabelle noch nicht belegt sind.
- Hybrides Hashing bietet nun die Vorteile beider Methoden
 - ⇒ Verkettung der Überläufer innerhalb der Hashtabelle ($n \leq m$)
 - ⇒ Zusätzlich wird nun pro Hashadresse ein Zeiger (Hashadresse) benötigt.
- Unterschied zum Verfahren der separaten Verkettung
 - ⇒ Eine komplette Kette mit Überläufern enthält i.a. nicht nur Datensätze, die zu einer gemeinsamen Hashadresse gehören.
 - ⇒ Suche nach freiem Platz muss zusätzlich implementiert werden

Dynamisches Hashverfahren:

Probleme bei bisherigen Verfahren:

- geeignete Wahl von der Größe der Hashtabelle ist wichtig
 - ⇒ á klein: schlechte Ausnutzung der Hashtabelle
 - ⇒ á gross: hohe Suchkosten
- keine Unterstützung stark anwachsender Hashtabellen
- Lösungsansätze für die Probleme
 - ⇒ Globale Reorganisation der Hashverfahren:
Wahl einer neuen Hashfunktion und Umspeicherung der Datensätze.
=> Hashtabelle steht während der Reorganisation nicht zur Verfügung
 - ⇒ Dynamische Hashverfahren:
Ständige Anpassung der Größe der Hashtabelle an die Anzahl der abgespeicherten Datensätze durch „kleine“ lokale Reorganisationen

Lineares Hashing:

Dynamisches Hashverfahren zur Verwaltung von Daten auf dem Externspeicher

- LH kann aber auch als interne Datenstruktur genutzt werden
- Als Kollisionsbehandlung werden wir uns im folgenden auf separate Verkettung beschränken. Andere Strategien sind ebenfalls auf LH übertragbar.

Das Verfahren:

1. Initialisierung
2. Mit zunehmender Datensatzanzahl steigt die Wahrscheinlichkeit für Überläufer
 - ⇒ Idee: Falls nach einem Einfügen eines Datensatzes á (Belegungsfaktor) zu groß geworden ist, wird die Hashtabelle um jeweils einen Behälter vergrößert (Expansionsschritt)
 - ⇒ Ein Expansionszeiger p zeigt auf den als nächsten aufzuspaltenden Behälter.

Analyse:

- im schlimmsten Fall betragen die Kosten für das Suchen, Einfügen und Löschen $O(n)$
- im Durchschnitt betragen die Kosten für das Suchen, Einfügen und Löschen konstant ($O(1)$)

Bäume:

Bäume erlauben es, hierarchische Beziehungen zwischen Objekten darzustellen. Derartige Hierarchien treten vielfach in der realen Welt auf.

- Der Grad eines Knotens K ist gleich der Anzahl der nicht-leeren Teilbäume von K
- $K = 0$, dann Blatt (keine Abzweigungen mehr)
- Jeder Knoten K ungleich $w(T)$ hat einen eindeutigen Vorgänger $v(K)$, auch Vater von K genannt.
- K wird dann auch als Nachfolger oder Sohn von $v(K)$ bezeichnet.
- Die Knoten, die denselben Vater $v(K)$ haben, werden als Brüder bezeichnet.
- Die maximale Höhe eines Baumes vom Grad m , $m > 0$, mit n Knoten ist n .

Bsp.: Abbildung in Java:

```
Class ArrTree {
    Elem [] key;
    ...
    Tree (Tree l; Elem x; Tree r){
        ...
    }
}
```

Datenstruktur Heap:

- Die Datenstruktur Heap (Heide) ist ein Spezialfall eines binären Baumes
- Heaps werden i.a. dazu benutzt den Datentyp Vorrangwarteschlange zu implementieren.

Binäre Suchbäume:

- in diesem Abschnitt betrachten wir Binärbäume zur Implementierung des Datentyps Menge
- Datensätze der Form $\langle K, \text{info} \rangle$
- Ein binärer Baum heißt Binärer Suchbaum, wenn für jeden seiner Knoten sie Suchbaumeigenschaft gilt, d.h. alle Schlüssel im linken Teilbaum sind kleiner, alle Schlüssel im rechten Teilbaum sind größer als der Schlüssel im Knoten

AVL-Bäume:

Ein binärer Suchbaum ist ein AVL-Baum, wenn für jeden Knoten p des Baumes gilt, dass die Höhe des linken Teilbaumes von p und die Höhe des rechten Teilbaumes von p sich höchstens um 1 unterscheidet.

Die Höhe eines beliebigen AVL-Baumes mit n Datensätzen ist $O(\log n)$

2-3-Bäume:

- Neben binären Knoten gibt es in 2-3-Bäumen ternäre Knoten, d.h. Knoten mit drei Söhnen
- Jeder Knoten hat entweder 2 (binärer Knoten) oder 3 (ternärer Knoten) Söhne
- Ein Knoten mit 2 Söhnen enthält einen Schlüssel, ein Knoten mit 3 Söhnen enthält 2 Schlüssel.
- Alle Blätter befinden sich auf derselben Ebene.
- Jeder Knoten erfüllt die Eigenschaft eines Suchbaumes, d.h.:

Vergleich von 2-3-Bäumen zu AVL-Bäumen:

- beide Verfahren benötigen im schlimmsten Fall logarithmischen Aufwand für die Operationen member, insert und delete
- alle Blätter liegen auf der gleichen Ebene
- einfache Implementation
-

Verallgemeinerung der 2-3-Bäume zu a-b-Bäumen, wobei:

- a der minimale Grad des Knoten ist
- b der maximale Grad des Knoten ist

Diese Bäume werden dann auch für große a und b als externe Datenstrukturen in DBS benutzt. Man spricht auch von B-Bäumen.

Graph:

Ein Graph ist eine Menge von Objekten und Beziehungen zwischen den Objekten

Ein Graph ist ein Paar $G = (V,E)$ mit:

- V: endliche, nichtleere Menge von Knoten
- E: Menge von Kanten: E Teilmenge von $V \times V$

Sei $k = (v,w)$ eine Kante. Man sagt, dass k inzident zu v und w ist, Knoten v und w benachbart (adjazent) sind.

Gilt für alle Kanten $(v,w) \in E$, dass auch $(w,v) \in E$, dann spricht man von einem ungerichteten Graph.

Datenstrukturen zur Repräsentation von Graphen unterscheiden sich durch:

- den benötigten Speicheraufwand
- den Aufwand für die Zugriffe auf Kanten und Knoten
- $G = (V,E)$ ein Graph
 - $n = |V|$ (Anzahl der Knoten)
 - $m = |E|$ (Anzahl der Kanten)

Adjazenzmatrix:

Markierte Adjazenzmatrix:

Adjenzliste

Vorteile :

- Platzbedarf beträgt $O(n+m)$
- Alle Nachbarn eines Knoten werden in linearer Zeit (in der Anzahl der Kanten, die von diesem Knoten ausgehen) gefunden.

Nachteil:

- Zugriffskosten für Suche nach einer Kante sind nicht konstant.

Durchlaufen von Graphen:

Probleme:

- Finde einen Ausweg aus einem planaren Labyrinth
- Prüfe, ob in einem Netzwerk ein Computer mit allen anderen Computern eine Verbindung aufbauen kann

Idee der Verfahren:

- Markieren von bereits benutzten Kanten
- Speichern von bereits besuchten Knoten, deren Nachfolger bereits alle besucht wurden, in einer Menge GRÜN
- Speichern der bereits besuchten Knoten, die möglicherweise noch nicht besuchte Nachfolger besitzen, in einer Menge GELB

Jeder Graph lässt sich auf einen Baum abbilden:

Minimal spannende Bäume:

Ein minimaler Spannbaum eines ungerichteten zusammenhängenden Graphen G ist ein Spannbaum mit minimaler Summe der Kosten über alle Kanten unter allen Spannbäumen von G .

Algorithmus von Kruskal (Schnitt basiert darauf):

- Am Anfang ist $E_T =$ leere Menge und jeder Knoten von T repräsentiert eine Komponente eines minimal spannenden Baumes (mit jeweils nur einem Knoten)
- Die Kantenmenge E wird in eine Priority Queue eingefügt
- Die Kanten werden nun nacheinander überprüft, ob die Endpunkte in zwei verschiedenen Komponenten liegen. Falls ja, wird die Kante gewählt und die beiden Komponenten miteinander verschmolzen. Andernfalls wird die Kante verworfen.

Sortieren:

Anordnen einer gegebenen Menge von Objekten in einer bestimmten Ordnung.

Klassifizierung von Sortierverfahren:

- interne / externe Sortierverfahren
- methodische Unterscheidung
- Effizienz
- Im Array sortieren oder nicht

- Allgemeine Sortierverfahren/ „eingeschränkte“ Sortierverfahren
- Laufzeitanalyse:
 - ⇒ Divide-Schritt:
 - Im Zerlegungsprozess wird das Pivotelement mit allen anderen Elementen verglichen
 - Zusätzlich ist noch ein weiterer Vergleich notwendig um die Schleife abubrechen.
 - ⇒ Conquer-Schritt:
 - Laufzeit hängt von der rekursiven Partitionierung des Arrays ab.
 - Darstellung der Partitionierung in einem binären Baum
 - Höhe des Baumes entspricht der Rekursionstiefe des Verfahrens.

Heap-Struktur:

Ausgangspunkt: vollständiger Baum bzw. unsortiertes Array:
 Elemente in den Blättern erfüllen die Heap-Eigenschaften

Absteigend über alle Ebenen werden die inneren Schlüssel „abgesenkt“, welche die Heap-Eigenschaft mit Ihren Söhnen verletzen.

Jedes allgemeine Sortierverfahren benötigt im schlimmsten Fall $\tilde{O}(n \log n)$ Vergleiche.

Implementierungsaspekte:

- Problem: Wie sieht eine geeignete Datenstruktur für die Behälter aus?
- Anforderungen an die Behälter:
 - ⇒ Einfügen in einen Behälter immer am Ende
 - ⇒ Entnehmen aus dem Behälter immer vom Anfang
 - ⇒ Ein Behälter enthält im schlimmsten Fall alle Elemente.
- Lösung:
 - ⇒ Ähnlich wie bei einer Hashtabelle mit direkter Verkettung implementieren wir die Behälter als ein Array von Listen und nutzen dabei die bereits
 - ⇒
 - ⇒ implementierte Funktionalität einer Liste aus.
 - ⇒ Einbettung der Behälter in ein Array mit n Schlüsseln

Sortierverfahren:

- Bottom-up Heapsort ist ein worst-case-optimales allgemeines Sortierverfahren. Im Durchschnitt ist es sogar besser als Quicksort. Darüber hinaus ist es ein echtes in-situ Verfahren und vermeidet Rekursion.
- Distribution Sort sortiert sogar mit linearem Aufwand (unter gewissen Annahmen, die durchaus nicht unrealistische sind)

NP Probleme:

- Die Klasse der NP-Probleme (np = nichtdeterministisch polynomial) lassen sich auf einem speziellen „Computer“ in polynomialer Zeit lösen.
- Dieser Computer ist eine nichtdeterministische Maschine, die bei der Auswahl des nächsten Schritts immer in Richtung der Lösung geht. Solch ein Computer kann leider niemals gebaut werden.

NP-vollständige Probleme:

- NP-vollständige Probleme sind spezielle (die schwierigsten) Probleme der NP.
- Jedes Problem aus NP kann auf ein beliebiges NP-vollständiges Problem in polynomialer Zeit zurückgeführt werden.

Divide-and-Conquer Algorithmen:

Vorgehensweise:

1. Divide: Teile das Problem in zwei etwa gleichgroße Teilprobleme
2. Conquer: Löse die Teilprobleme auf dieselbe Art (rekursiv)
3. Merge: Füge die Teillösungen zu einer Gesamtlösung zusammen.

Randomisierte Algorithmen:

- Sind Algorithmen, deren Ausführungen durch Zufallszahlen gesteuert werden (und nicht nur über die Eingabeparameter).
- Werden eingesetzt, um NP-vollständige Probleme effizient „zu lösen“

Backtracking Algorithmen:

- Algorithmen zur Lösung von i.A. NP-vollständigen Problemen, die eine erschöpfende Suche geschickt implementieren.

Toleranzschwellenverfahren:

Fragen:

- Wie soll die Toleranzschwelle T gewählt werden?
- Was heißt längere Zeit?
- Wie soll T im Schritt 4 (Absenkung auf 0) abgesenkt werden?

Sintflut-Algorithmus

Simuliertes Ausglühen:

Randomisiertes Verfahren:

- zufällige Auswahl einer Nachbarlösung
- Falls die Nachbarlösung schlechter ist als die bisher berechnete Lösung wird mit einer gewissen Wahrscheinlichkeit die Lösung akzeptiert
- Die Wahrscheinlichkeit der Akzeptanz einer Lösung nimmt während des Verfahrens kontinuierlich ab.