Algorithmus
- Anleitung zum Lösen einer Aufgabenstellung. Bsp: Programm -> Kein Prog. ohne Algo.
- Tests sind da um Fehler zu erkennen, nicht um die Korrektheit des Prog. zu zeigen
- Determinierheit: Identische Eingabe führen stets zur identischen Ergebnissen
- Determinismus: Ablauf des Verfahrens ist an jedem Punkt fest vorgeschrieben
- Terminierung: Für jede Eingabe liegt Ergebnis nach endlich vielen Schritten vor
- Effizienz: „Wirtschaftlichkeit“ des Aufwands relativ zum vorgegebenen Massstab

Abstrakter Datentyp
- Unterscheidung zwischen Schnittstelle(sichtbar) und Implementation(unsichtbar)
-> Information Hiding
- Beschreibt was, aber nicht wie
- Java Implementation mittels Interface

Stack
- Last in first out (LIFO) //Bspw. Klammerungtests
- minimale Methoden: push(Obj x) (hinzufügen), pop() (entfernen), isEmpty()
- zus. Methoden: peek() (Abfrage oberster Wert), removeAll(), isFull()
- Vorbedingungen darf man abschwächen, bspw. Wertebereich erweitern short -> int
- Nachbedingung darf man verstärken, bspw. Wertebereich verkleinern int -> short
- Stack lässt sich gut mit einer Liste umsetzen (Array hat feste Grösse)

Queues
- First in first out(FIFO)
- Warteschlang bspw. Drucker; häufig auch für Buffers verwendet bspw. Ringbuffer
- gibt Interface Queue in Java
- ArraImplementation: zwei int Variablen (outIdx[älteste Element; inIdx[nächste freie Stelle]) bestimmen momenten gültigen Inhalt der Queue
- nofItems = (inIdx - outIdx)
- Priority Queues: Elemente erhalten Prio und wandern so entsprechend nach vorne

O-Notation
$3n^3 + n^2 + 1000n + 500 = \mathbf{O}(n^3)$

Ignoriert Proportionalitätskonstante

Ignoriert Teile der Funktion mit kleinerer Ordnung

- Welche Zeit braucht ein Algorithmus -> Zeitkomplexität
- Wie viel Speicher benötigt ein Algorithmus -> Speicherkomplexität
- O(1) konstanter Aufwand (Add., Subtr. Multipl.); O(log n) -> logarithmischer Aufwand (binäre Suche); O(n) -> linearer Aufwand; O(n*log n) (gute Suchalgorithmen); O(n^2) quadratischer Aufwand (Primfaktorzerlegung); O(n^k) -> konstantes k > 1 polynomialer Aufw.; O(2^n) -> exponentieller Aufwand
- grösser Exponent ist relevant -> wachsen schneller; der am schnellsten wachsende Term dominiert; Bei Log-Funktionen ist Basis und Mulitpl.Faktor irrelevant, langsamer als Polynome; Exponential-Funkt. Multipl. Fakt. irrelevant, wachsen schneller als Polynome;
- Für doppelt so grosses n, brauche ich einen Durchlauf mehr

Listen
- Verwendung: Stack, Queue, Speicherverwaltung OS
- Vorteil: muss beim Erstellen nicht wissen wie gross
- Methoden: add(Obj x), add(int pos, Obj x), get(int pos), remove(int pos), size, isEmpty()
- Rekursive Def: Liste ist entweder leer, besteht aus einem Element oder aus einem Element welches auf Teilliste

Doppelt verkettete Listen
- arbeiten mit .next und .prev

Methoden: add() (Aufwand O(1)); remove(); -> Jeweils Umhängen von von next & prev

Liste vs. Array
- i-te Element ist in Arrays günstig und List relativ teuer

Iterator
- mit O(n) durch Liste iterieren
- Iterator muss von Liste erzeugt werden, Methoden: hasNext(), next()
- Bei foreach wird der Iterator implizit erzeugt

Rekursion
- heisst rekursiv definiert, wenn sich selbst als Teil enthält oder mit Hilfe von sich selber definiert ist
- direkte Rekursion: Ruft eine Methode sich selber wieder auf
- indirekte Rekursion: 2 oder mehrere Methoden rufen sich gegenseitig auf (Fehlerquelle!)
- Endrekursion: Programme bei denen der rekursive Aufruf die letzte Aktion im Else-Zweig ist werden endrekursiv bezeichnet
- Zu jedem rekursiv formulierten Algo. gibt es einen äquivalenten iterativen Algo. (Umgekehrt gilt das nicht immer!)
- Vorteile: kurz, leichtverständlich, Einsparung, teilweise sehr effiziente Problemlösung
- Nachteile: wenig effizientes Laufzeitverhalten (overhead), Konstruktion gewöhnungsbedürf.

```
//normale Ausgabe (Rekursiv)
Void traverse(ListNode p) {
    if(p==null)  //Abbruch
    else {
        System.out.println(p.value);
        traverse(p.next);
    }
}
```

```
//umgekehrte Ausgabe (Rekursiv)
void traverse(ListNode p) {
    if(p==null)  //Abbruch
    else {
        traverse(p.next);
        System.out.println(p.value);
    }
}
```

```
//umgekehrte Ausgabe (Iterative Lösung)
p = node
while(p == null) {
    push(p);
    p = p.next;
}
while(p==pop()){
    System.out.println(p);
}
```

Bäume *(Menge Knoten&Menge gerichtet Kanten, root nur Ausgangsknoten, Kanten genau eine Eingangsknoten)*
- Wenn eine Liste nicht mehr ausreicht, verwendet man Bäume
- Ein Baum ist entweder leer oder er besteht aus einem Knoten mit keinem, einem oder mehrere disjunkten (zwei Teilbäume dürfen sich nicht einen Knoten teilen) Teilbäume. Diese Teilbäume entsprechend dann wiederrum der Definition eines Baumes
- Alle Knoten ausser Wurzel (**root**) sind nachfolger genau eines Vorgänger-Knotens
- Knoten mit Nachfolger werden innerer **Knoten** (auch vertex/Node) genannt
- Knoten ohne Nachfolger werden **Blatt** genannt, Knoten mit Nachfolger **Innerer Knoten**
- Knoten mit dem gleichen Vorgänger-Knoten sind **Geschwisterknoten** (Siblings)
- Es gibt genau einen Pfad vom Wurzel-Knoten zu jedem anderen Knoten
- Die Anzahl **Kanten** (Edge), denen wir folgen müssen, bestimmen die Weglänge (path length)
- Die **Tiefe** (oder Höhe) eines Baumes gibt an, wie weit die tiefsten Blättern von der Wurzel entfern sind: Anzahl Kanten + 1
- Das **Gewicht** ist die Anzahl Knoten des (Teil-)Baumes

root-Node = A

Tiefe = 4

Gewicht = 11

Nachfolger von B = F/G

Nachfolger von A = restlichen

Vorgänger von K = J/E/A

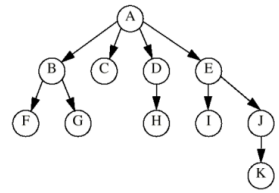
Blattknoten = F/G/C/H/I/K

Geschwister von C = B/D/E

Geschwister von H = F/G/I/J

Ein Knoten hat genau einen ..... direkte Vorgänger-Knoten.

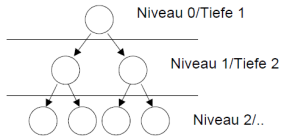
Ein Knoten kann bel. viele ..... direkte Nachfolger haben.

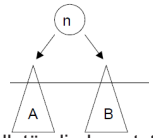


Tiefe/Höhe: k

auf jedem Niveau 2^n Knoten oder $2^{(k-1)}$

Maximal Anzahl $2^k - 1$ Knoten



Binärbaum


- Ein Knoten hat maximal zwei Nachfolger
- Ein Knoten besteht grundsätzlich immer drei Referenzen (pro Ref. 8 Byte bei 64-Bit)
- maxHeight: n minHeight: abgerundet($\log_2(n)$) + 1
- Ein Binärbaum heisst **voll** (oder vollständig) wenn ausser der letzten alle seine Ebenen vollständig besetzt sind.

Traversierung
- Preorder: n, A, B (Verarbeitung am Anfang - von oben nach unten)
  1. Besuche die Wurzel, 2. Traverse linker Teilbaum, 3. traverse rechter Teilbaum
- Inorder: A, n, B (Verarbeitung in der Mitte - von links nach rechts)
  1. Traverse linker Teilbaum, 2. Besuche Wurzel, 3. traverse rechter Teilbaum
- Postorder: A, B, n (Verarbeitung am Schluss - von unten nach oben)
  1. Traverse linker Teilbaum, 2. traverse rechter Teilbaum, 3. Besuche Wurzel
- Levelorder: n, a0, b0, a1, b1,... (Verarbeitung schichtenweise - Bspw. Queue LIFO)
  1. zuerst Wurzel, 2. die Wurzel des linken Teilbaum, 3. die Wurzel des rechten Teilbaum...

  wichtig! kann man nicht rekursiv programmieren, muss iterativ sein!

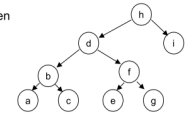
```
visitor.visit(node.element);
preorder(node.left,visitor);
preorder(node.right,visitor);
```

```
inorder(node.left,visitor);
visitor.visit(node.element);
inorder(node.right,visitor);
```

```
postorder(node.left,visitor);
postorder(node.right,visitor)
visitor.visit(node.element);
```

```
void levelorder(TreeNode<T> node, Visitor<T> visitor) {
    Queue q = new Queue();
    if (node != null) q.enqueue(node);
    while (!q.isEmpty()){
       node = q.dequeue();
       visitor.visit(node.element);
       if (node.left !=null) q.enqueue(node.left);
       if (node.right != null) q.enqueue(node.right);
    }
}
```

Zeigen Sie die Reihenfolge bei den verschiedenen Traversierungsarten auf



Preorder: (n, L, R) h, d, b, a, c, f, e, g, i

Inorder: (L, n, R) a, b, c, d, e, f, g, h, i

Postorder: (L, R , n) a, c, b, e, g, f, d, i, h

Levelorder: h, d, i, b, f, a, c, e, g

Mutation von sortierten Bäume
- Bäume werden durch das anhängen von neuen Blättern erweitert-> jeweils am Schluss!
- Unterschied zum Baum: Der Baum kann zwei Nachfolge Knoten haben und nicht nur einen -> Entweder links oder rechts anhängen. Falls man jetzt immer nur rechts oder links ein neues Element hinzufügen würde, dann hätte man quasi eine Liste.

  -> Folgerung: Eine Liste ist ein Baum
- Möglichkeit ausgeglichen Baum: könnte man mit Zufallszahlen oder abwechselnd links/rechts arbeiten.

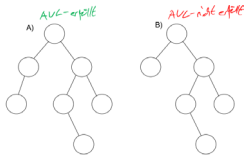
Sortierte Binärbäume
- Werden anhand ihres Schlüsselwertes geordnet eingefügt
- jeden Knoten gilt: linken Unterbaum sind jeweils kleinere und rechts grössere Elemente

Suchen im sortierten Binärbäume
- Wenn x == Wurzelelement -> x gefunden; Wenn x > Wurzelelemetn wird die die Suche im rechten Teilbaum fortgesetzt, sonst im linken Teilbaum **Aufwand**: $Log_2(n)$

Zugriffszeiten und Tiefe
- Die Zugriffszeit ist proportional zur Tiefe des Baumes -> Das Ziel sollte sein ein Baum zu kreieren, der möglichst flach ist

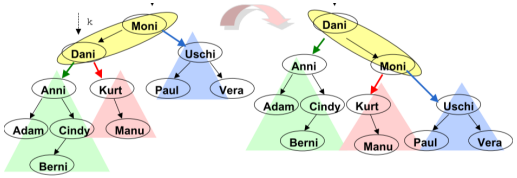
Balancierte Bäume
- Ein voller Baum ist immer ausgeglichen
- Ein Binärbaum mit n Elementen hat im Minimum eine Tiefe von Log2(n+1) (auf nächste Zahl aufgerundet)
- **vollständig ausgeglichen**: Die Gewichte der beiden Teilbäume unterscheiden sich +/- 1
  eine solche „super Ausgeglichenheit“ ist schwierig und mühsam

AVL-Bedingung (Einfügen / Suchen O(log(n)))

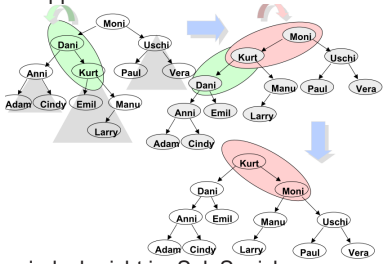


- Die Tiefen der beiden unterscheiden sich max. um 1
- Beim Einfügen und Löschen sorgt man dafür, dass die AVL-Bedingung erhalten bleibt
- Vorteil: einfacher zu realisieren, Degenerierung Liste nicht möglich, Aufwand O(log(n))
- Nachteil: Zus. Aufwand bei Programmierung, Einfügen und Löschen sind komplizierter
- Operationen: Suchen und traverse bleiben identisch (ist Binärbaum)
- Zum Wiederherstellen der AVL-Bedingung verwendet man Rotationen

Einzelrotation



Doppelrotation



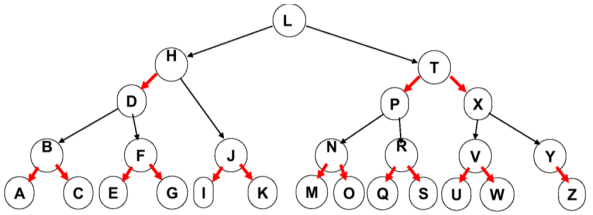
B-Bäume
- Binärbäume eigenen sich gut für Struktur im Hauptspeicher, jedoch nicht im Sek.Speicher
-> Baum so aufbauen, dass wahlfreie Zugriffe auf Disk-Blöcke minimiert wird:
Möglichst viele Daten pro Block; B-Bäume = möglichst viele Verweise pro Knoten;
Knoten eines B-Baumes entsprechend einem Abschnitt auf der Harddisk; Alle Knoten sind gleich gross; Baum ist immer ausgeglichen - Werden bei der Konstruktion (Einfügen / Löschen) automatisch balanciert; In einem B-Baum der Ordnung n entählt jeder Knoten ausser der Wurzel mindestens n/2 und höchstens n Schlüssel; Jeder Knoten ist entweder ein Blatt oder hat m+1 Nachfolger, wobei m die Anzahl Schlüssel des Knotens ist
Anwendung: Windows NTFS-Filesystem
- Unterschied Hauptsiecherzugriff (ns) zu Disk(ms) ist 1:1‘000‘000
- B-Bäume werden von unten nach oben aufgebaut

Suchen: 1) Den Wurzelblock lesen 2) Gegebenen Schlüssel S auf dem gelesenen Block suchen 3) Wenn gefunden, Datenblock lesen fertig 4) Ansonsten i finden, so das Si < S < Si+1Block Nr i einlesen, Schritte 2 bis 5 wiederholen (rekursiver Aufruf)
Aufwand Suche: O(log(n))

Rot-Schwarz-Bäume
- Man erstellt einen B-Baum im Hauptsiecher und ergänzt diesen durch Binär-Bäume
- grosse Knoten werden durch Binärbäume mit „roten“ Kanten/Knoten implementiert
- Ausgleichsverfahren so dass Rot-Schwarz Bedingung erhalten bleiben
- Die roten Kanten zählen nicht zum Gewicht
- Bedingung:

1) Jeder Knoten im Baum ist entweder rot oder schwarz
2) Die Wurzel des Baumes ist schwarz
3) Jedes Blatt (NIL-Knoten) ist schwarz
4) Kein roter Knoten hat ein rotes Kind (auf rote Kanten folgt immer eine schwarze)
5) Jeder Pfad, von einem gegebenen Knoten zu seinem Blattknoten, enthält die gleiche Anzahl schwarze Knoten(Schwarzhöhe/Schwarztiefe)

--> Für die Tiefe zählt man nur die schwarzen Knoten -> Max 2* Knoten



Binärbaum
- allgemeine Grundform, einfach in der Implementierung

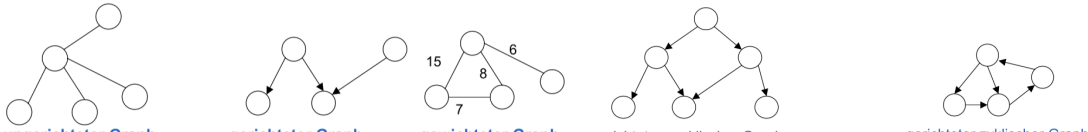
B-Baum:
- Optimiert für Massenspeicherzugriff. Es gibt mindestens n/2 Unterbäume. Dadurch wird der Baum weniger hoch
  ⇒ weniger Plattenzugriffe notwendig

2-3-4 Baum
- B-Baum mit max. 4 Nachfolgern

Rot-Schwarz Baum:
- Durch "Färben" der Kanten/Kanten kann 2-3-4 Baum als Binärbaum implementiert werden: java.util.TreeMap

Splay-Tree (nicht behandelt):
- Knoten, auf welche oft ein Zugriff erfolgt, werden zum root-Knoten rotiert.
- Zugriff wird schneller (self-adjusting strategy)

Graphen
- Ein Graph G=(V,E) besteht aus einer endlichen Menge von Knoten V und einer Menge von Kanten E
- Zwei Knoten x und y sind benachbart (**adjacent**) falls es eine Kante e_{xy}=(x,y) gibt
- **Verbundener Graph** (connected graph) ist jeder Knoten mit jedem anderen Knoten verbunden = es exisiert ein Pfad
- **Verbunderener Teilgraph** Gesamter Graph kann aus Teilgraphen bestehen, die untereinander nicht verbunden sein müssen
- Kann *können* gerichtet und/oder mit Gewicht sein
- **Kompletter Graph**: Jeder noten ist mit jedem anderen Knoten direkt verbunden
- **Dichter Graph** (dense) Nur wenige Kanten (bez. auf kompletter Graph)im Graph fehlen
- **Dünner Graph** (sparse) Nur wenige Kanten im Graph (bez. kompletter Graph) vorhanden
- **Gewichtete Graphen** werden auch Netzwerk genannt, Pfadlänge = Summe aller Kanten
- **ungerichteter Graph** ist die relation in jedem Fall symmetrisch
- Ein Baum ist auch ein Graph



ungerichteter Graph | gerichteter Graph | gewichteter Graph | gerichteter azyklischer Graph | gerichteter zyklischer Graph

Adjazenz-Liste
- jeder Knoten führt (Adjazenz-)Liste, welch alle Kanten zu den benachbarten Knoten enthält

Adjazenz-Matrix
- optimal bei wenig Knoten mit vielen Verbindungen (dichter Graph)
- Vorteil: effizient, einfache Implementation, speichereffizient
- Nachteil: ziemlicher (Speicher-) Overhead

	A	B	C
A	false	true	true
B	true	false	false
C	false	true	true

	Vorteile	Nachteile
Adjazenzmatrix	Berechnung der Adjazenz sehr effizient	hoher Platzbedarf und teure Initialisierung: wachsen quadratisch mit $O(n^2)$
Adjazenzliste	Platzbedarf ist proportional zu n+m	Effizienz der Kantensuche abhängig von der mittleren Anzahl ausgehender Kanten pro Knoten

Graph-Traversierung

Tiefensuche: Ausgehend von einem Startknoten geht man vorwärts (tiefer) zu einem neuen unbesuchten Knoten, solange einer vorhanden (d.h. erreichbar) ist. Hat es keine weiteren (unbesuchten) Knoten mehr, geht man rückwärts und betrachtet die noch unbesuchten Knoten. Entspricht der Preorder Traversierung bei Bäumen (man geht vorwärts (tiefer))

Breitensuche: Ausgehend von einem Startknoten betrachtet man zuerst alle benachbarten Knoten (d.h. auf dem gleichen Level), bevor man einen Schritt weitergeht. Entspricht der Levelorder Traversierung bei Bäumen Man geht nach links & rechts (breiter)

Dijkstras Algorithmus
- schnellster Algorithmus um den kürzesten Pfad zu finden
- besuchte Knoten werden rot markiert. Suche unter den benachbarten Knoten denjenigen, dessen Pfad zum Startknoten das kleinste Gewicht hat. Besuche diesen und bestimme dessen benachbarten Knoten

Maximaler Fluss
- Die Kanten geben den maximal Fluss zw. den Knoten an.

Spannbaum
- Ein Spannbaum eines Graphen ist ein Baum, der alle Knoten des Graphen enthält. Ein minimaler Spannbaum (minimum spanning tree) ist ein Spannbaum eines gewichteten Graphen, sodass die Summe aller Kanten minimal ist

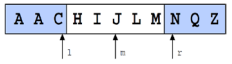
Trial and Error
- Im Labyrinth den richtigen Weg finden, ist ein Entscheidungsbaum
- Bei Sackgasse geht man einen oder mehrere Schritte zurück (-> Backtracking)

Vorgehen: Solange die Lösung nicht gefunden -> Erweitung der bestehenden (Teil-)Lösung möglich,
Ja ->
füge Erweiterung hinzu, überprüfe, ob erweiterte Lösung zum Ziel führt
ja -> Lösung gefunden = Abbruch, nein -> nehme Erweiterung zurück
Nein ->
keine Lösung möglich, Abbruch

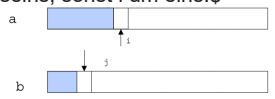
Suchen
- Vorbedingung: Aussage, die **vor** dem Ausführen der Programmsequenz gilt
- Nachbedingung: Aussage, die **nach** dem Ausführen der Programmsequenz gilt
- Invariante: Ist eine Bedingung die vor und nach dem Programmdurchlauf und demnach nicht veränderlich. Diese können für den formalen Beweis einer Korrektheit verwendet werden. -> Am Schluss gilt jeweils die Invariante und die Abbruchbedingung der Schleife

Binäres Suchen in einem sortierten Array
- Zuerst in die Mitte (m), dann schauen ob grösser oder kleiner als gesuchtes und dementsprechend linker (l) oder rechter Pfeil (r) setzen; Dieser Vorgang wiederholen mit der restlichen Menge (innerhalb der Pfeile)
- Falls die Pfeile sich überschneiden, dann ist das Element nicht in der Menge

Aufwand O(log(n)) -> Man halbiert bei jedem Durchgang den Bereich wo gesucht wird

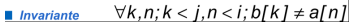


Schnelle Suche in zwei sortierten Arrays
- Man vergleicht jeweils b[j] mit a[i], wenn j kleiner ist als i, dann erhöht man j um seins, sonst i um eins.$
- Der Aufwand beträgt O(n)



Aufwand für Suchen und Einfügen
- sortierter Array
  - Einfügen: O(n/2)
  - binäres Suchen: O(Log_2(n))
- lineare sortierte Liste
  - Einfügen: O(n/2)
  - Suchen: O(n/2)
- sortierter Binärbaum
  - Einfügen: O(Log_2(n))
  - Suchen: O(Log_2(n))

Sentinel: Ist ein Element das angibt,



Hashing
- Es gibt jeweils einen Schlüssel und einen Inhalt, wobei der Schlüssel Teil des Inhalts sein kann
- Es gibt eine **Hash-Funktion** die den Schlüssel und somit den „Platz im Array“ definiert. Eine einfache Möglichkeit wäre hier: X % tablesize; weitere Möglichkeiten: k%n oder (k*n) % m | n,m € Primzahl
- Dadurch, dass der ursprüngliche Wert nicht wieder hergestellt werden kann, wird der Originalwert gespeichert.
- Wenn an einem Speicherort ein Element eingefügt werden soll, es bereits aber belegt ist, nennt man Kollision
- hashCode-Methode: equals == true, müssen Objekte denselben Hashwert liefern und compareTo == 0 liefern, false = unterschiedliche und compareTo != 0 liefern

Folgerung! Immer equals, compareTo und hashCode zusammen überschreiben

Kollisionen
- Die Anzahl Kollisionen hängt von der Güte der Hash-Funktion und der Belegung der Zellen ab
- Der Load-Faktor (ƛ) sagt wie stark der Hash-Bereich belegt ist und bewegt sich zwischen 0 und 1. Die Anzahl Kollision ist abhängig von ƛ und h: f(h, ƛ)

-> Ab dem Wert 0.7/0.8 gibt es einen sehr starken Anstieg der Wsk. an Kollisionen, daher füllt man sein Array jeweils nur bis 70 / 80 %

linear Probing

1) Anwendung Hash-Funktion, 2) Wenn es zu einer Kollision kommt, dann geht man einfach Feld weiter, bis ein leerer Platz gefunden wurde

Phänomen des Primary Clustering: musste einmal eine freie Zelle nebem Hash-Wert belegt werde, steigt die Wsk, dass weiter gesucht werden muss -> Verlängerung durchsch. Zeitaufwandes zum Sondieren

quadratic Probing
- Es gibt eine quadratic-Probing Formel: bspw. (h(x) + i^2) % tablesize i = beginnend 1

1) Anwendung Hash-Funktion 2) Wenn es zu einer Kollision kommt, Anwendung der quadr.Prob.Formel, wenn Platz frei -> fertig 3) i um 1 inkrementieren und wiederholen -> primary Clustering tritt viel weniger auf

löschen

Werte können nicht einfach so gelöscht werden, entweder man führ ein rehashing durch oder man löscht das Element nicht, sondern markiert es nur als gelöscht.

- Vorteil: Suche ist sehr effizient, Implementationsaufwand ist kleiner als für ausgeglichene binär Bäume
- Nachteil: kleinste oder grösste Elem. nicht einfach zu finden, geordnete Ausgabe nicht möglich, Suche nach Bereich nicht möglich

Folgerung: Hash-Tabellen sind geeignet wenn: die Reihenfolge nicht von Bedeutung sit und nicht nach Bereichen gesucht werden muss und die ungefähre (max) Anzahl bekannt ist.

Extendible Hashing (kein Platz mehr im Array)
- Falls noch im Hauptspeicher Platz: Rehashing -> Arraysize verdoppeln und „kopieren“ -> relativ teuer
- Es wird in der Hashtable eine Referenz auf ein sogenanntes „Bucket“ gespeichert. In diesem Bucket (eigenes Array) werden dann die einzelnen Elemente eingefügt.

Sortierverfahren
- bei kleinen Datensätzen kann man im Arbeitsspeicher des Computers sortieren -> **internes Sortieren**
- Der **Sortierschlüssel** muss nicht zwingend eindeutig sein! Der Sortierschlüssel soll einfach eine Ordnung schaffen (bspw. Jahrgang) und kann auch aus mehreren Teilfeldern bestehen. Sortierschlüssel ist Teil des Inhaltes
- bei einer alphabetischer Sortierung kann das comparable-Interface nicht verwendet werden, denn je nach Land und Sprache gelten unterschiedliche „Richtlinien“ für die Sortierung. Aus diesem Grund gibt es Colllator-Klassen in Java, welche das Comparable-Interface implementiert

Sortieralgorithmen
- Häufig wird eine swap-Methode verwendet, diese könnte man direkt generisch auf Stufe Methode erzeugen.

```
private static <K> void swap(K[] k, int i, int j){
        K h = k[i];k[i] = k[j];k[j] = h;
        }
```

Deutschland

```
DIN 5007 Variante 1
ä und a sind gleich
ö und o sind gleich
ü und u sind gleich
ß und ss sind gleich

DIN 5007 Variante 2
ä und ae sind gleich
ö und oe sind gleich
ü und ue sind gleich
ß und ss sind gleich
```

Schweiz/Österreich

```
Ä folgt auf a
Ö folgt auf o
Ü folgt auf u
ß folgt auf ss
St. folgt auf Sankt
```

Finnisch

```
Å kommt nach z
Ä kommt nach Å
Ö kommt nach Ä
Ü und y sind gleich
```

BubbleSort
- Pro Durchgang ist das verbleibende grösste Element rechts
- Im Zweifelsfall kann man immer BubbleSort verwenden
- Der Algorithmus erkennt, wenn ein Datensatz bereits sortiert ist
- Ist für kleine Datensätze genügend schnell
- Aufwand: BestCase O(N); AverageCase & WorstCase O(N^2)

```
static <T extends Comparable> void BubbleSortG(T[] a){
  for (int k = a.length-1; k > 0; k--){
    // bubbleUp
    for (int i = 0; i < k; i++)
      if (a[i].compareTo(a[i+1]) > 0) swap (a, i, i+1);
    }
}
```

```
static void BubbleSort2(char[] a){
  for (int k = a.length-1; k > 0; k--){
      boolean noSwap = true;
      for (int i = 0; i < k; i++){
        if ( a[i] > a[i+1]) {
            swap (a, i, i+1);
            noSwap = false;
        }
      }
      if (noSwap) break;
  }
}
```

SelectionSort
- Es gibt grundsätzlich keinen Grund Selection Sort zu verwenden
- für relativ lang und völlig unsortierte Datensätze geeignet
- Wenn die Invariante giltet, so ist dieser Bereich sortiert
- erkennt nicht ob bereits „fertig sortiert“ oder nicht
- Man teilt den Bereich in einen sortieren & unsortierten Bereich wobei am Anfang der sortierte Bereich 0 Elemente hat
- Man geht durch die ganze Liste und sucht das Min-Element und fügt es in den sortierten Bereich
- Aufwand: O(N^2)

```
static void SelectionSort(char[] a){
    for (int k = 0; k < a.length; k++){
          int min = k;
          for (int i = k+1; i < a.length; i ++) {
              if (a[i] < a[min]) min = i;
          }
          if (min != k) swap (a, min, k);
    }
}
```

finde kleinstes Element

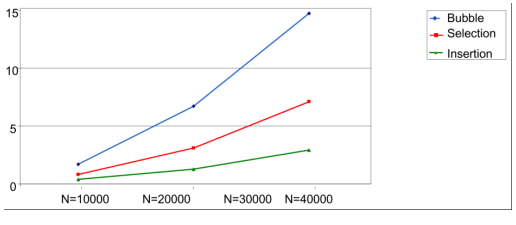
InsertionSort
- Für relativ kurze und gut vorsortierte Datensätze geeignet
- Man hat einen sortierten & unsortierten Bereich
- Man nimmt das erste Element aus dem unsortierten Bereich
- Fügt es an der „sortierten Stelle“ im sortierten Bereich ein
- Aufwand: O(N)
- Ist der Schnellste unter den Langsamen

```
static void InsertionSort(char[] a){
  for (int k = 1; k < a.length; k++)
    if (a[k] < a[k-1]){
        char x = a[k];
        int i;
        for (i = k; ((i > 0) && (a[i-1] > x));i--)
            a[i] = a[i-1];
        a[i] = x;
    }
}
```

richtiger Platz?
Element das eingeordnet werden soll
verschiebe Lücke

Laufzeit vs. Ordnung
- Die Ordnung besagt, wie stark sich de Aufwand bei einer Veränderung der Eingangsdaten verändern: O(N^2) -> Verdopplung von n, 4-mal grösser
- Die Laufzeit besagt, wie lange das Programm benötigt. Ist von vielen ,auch Hardware, Faktoren abhängig

Stabilität
- Fügt man ein zweites Kriterium hinzu, bleibt die Sortierreihenfolge die gleiche



Algorithmus	Effizienz	Stabilität
Bubble Sort	O(n^2)	stabil
Selection Sort	O(n^2)	instabil
Insertion Sort	O(n^2)	stabil
Quick Sort	O(n * log n)	instabil
Merge Sort	O(n * log n)	stabil*

Prinzip: Teile und Herrsche / Divide and conquer
Zerlege das Problem in kleinere, einfach zu lösende Teile -> löse die erhaltenen Teilprobleme -> füge die Teillösungen wieder zusammen. Falls das Problem klein genug ist, muss nichts gemacht werden. Ansonsten wird das Problem in zwei Teilmengen geteilt und dann wieder geordnet zusammengefügt -> Rekursion

Quicksort
Verfolgt das Grundprinzip von „Teile und Herrsche“
Teilt die zu sortierende Menge in zwei gleich grosse Teilmenge A und B. Des Weiteren wird ein mittleres Element W bestimmt. Sämtliche Elemente der Teilmenge A sind kleiner als W, und der Teilmenge B grösser als W. Innerhalb der Teilmenge sind die Elemente jedoch noch nicht sortiert. Dieses Vorgehen wiederholt man nun für alle Unterteilmengen bis die Menge sortiert ist. Das Zusammenfügen der Teilmengen passiert automatisch.
Pivot (W) wird dabei auf einen Median geschätzt, denn bei einer genauen Bestimmung des Medians ginge der Laufzeitenvorteil von Quicksort wieder verloren. Nimm das linke, rechte und ungefähr mittlere Element. Danach gibt es drei Strategien. 1. nehme eines der drei Elemente; 2. nehme das wertmässige Mittlere; 3. nehme das arithmetische Mittlere
Mittelwert: durchschnittlicher Wert; Median: teilt in zwei gleich grosse Teilmenge (50 / 50)
Aufwand: BestCase: O(n*log(n)); AverageCase: O(n*log(n)); WorstCase: O(n^2) -> erkennt nicht Vorsortierung
--> Quicksort ist immer die erste Wahl bei einer grösseren Menge welche unsortiert ist.

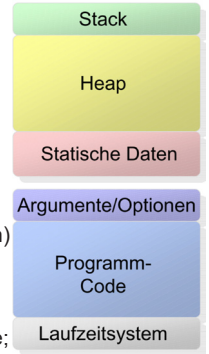
Distributionsort
Einzige Variante welche keinen Vergleich von zwei Elementen durchführt und ist die schnellste Art wie man sortieren kann. Die zu sortierenden Elemente werden entsprechend dem Sortierschlüssel in Fächer verteilt O(n) und zusammengetragen O(n). Früher wurde beispielsweise so die Briefe nach PLZ bei der Post sortiert.
Aufwand zwischen O(n) und O(n*log(n)) -> Am Anfang ist ebenfalls ein Aufwand notwendig
Vorteile: schneller geht nicht, da linearer Aufwand
Nachteile: Verfahren muss an jeweiligen Sortierschlüssel angepasst werden; geht nur für Schlüssel mit kleinem Wertebereich;
-> Wird heute praktisch nicht mehr verwendet, da der zusätzliche (Vorbereitungs-)Aufwand sich nicht lohnt
Unter Umständen ist bei vorsortierten Mengen die InsertionSort die bessere Variante

Externes Sortieren (-> Mischen)
Beim externen Sortieren liegen die Daten in einer Datei auf der Festplatte. Zwei Arten des Zugriffs sind möglich:
1. sequentieller Zugriff; 2. der bel. auf die Elemente wäre zwar möglich, ergäbe jedoch einen grossen Effizienzverlust -> Annahme: Datenstrom der sequentiell gelesen wird, jeweils nur ein Tei lder Daten passen in Hauptspeicher
1. Phase: Lade jeweils einen Teil der Datei in den Speicher -> sortiere diesen Teil mit schnellem internem Verfahren -> sortiere Abschnitte in mehrere (mind. 2) Ausgabedateien -> Es entsteht Folgen von Sortierten Abschnitten
2. Phase: nimm zwei geordnete Abschnitte -> lese von beiden Dateien das erste Element und schreibe das kleinere in einen neuen Abschnitt und lese das nächste von der gleichen Datei. Dies hat zur Folge, dass die Länge der geordneten Abschnitte sich verdoppelt haben. Der Output wird zum neuen Input. Dieser Vorgang wiederholt man bis es vollständig sortiert ist.

Wahl des Sortierverfahrens
1. internes (im Hauptspeicher) oder externes (Hauptspeicher und Sekundärspeicher) Verfahren
2. Methode des Algorithmus (Vertauschen, auswählen, einfügen, rekursion, verteilen, mehrphasen)
3. nach Effizienz (Laufzeit oder Speicherbedarf)
4. Voraussetzungen (Schlüsselvergleiche, Reihenfolge ädern, Stabilität, Struktur)

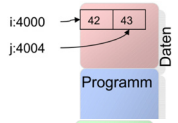
Optimierung
Den Quicksort kann man auf mehreren Threads laufen lassen und durch das eine höhere Performance erzielen. Die naive Art erzeugt jedes mal einen neuen Thread. Die bessere Art ist die Implementierung mittels Threadpools, wobei ein Thread immer eine Aufgabe erfüllt und anschliessend dem Pool wieder zur Verfügung steht. Dasselbe Prinzip Fork / Join „Teile und Herrsche“ Prinzip auf Parallelität angewandt.

Dynamische Speicherverwaltung
- Früher hatte man bis zu ca. 40% des Aufwandes für den dynamischen Speicher gebraucht
- Dynamischer Speicher ist oft ausschlaggebend für die Effizienz
- Fatale Programmfehler (Abstürze) sind meist auf den Speicher zurückzuführen
- Fehler im dynamischen Speicher schwierig zu finden, weil Sie erst zur Laufzeit erfolgen
- in OOS schwierig festzustellen welche Objekte noch verwendet werden
- Java: automatische Speicherfreigabe -> Garbage Collector

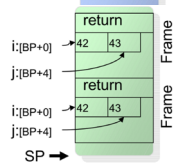


- Daten liegen im Stack (Rücksprung-Adresse und lokale Variablen), im Heap (dynam. Daten) und statische Daten.
- Argumente und Optionen bspw. Stringargument beim Java Programm-Startpunkt

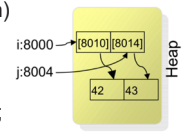
Statische Datenzuteilung
Einfachste Zuteilungsstrategie; Compiler legt Zuordnung Variablenname zu Speicheradresse; Speicher wird beim Start angefordert und beim Verlassen freigegeben; Java mittels static
Vorteil: einfach und bereits beim Start kann gesagt werden ob Speicher ausreicht
Nachteil: Grössen müssen zur Übersetzungszeit bekannt sein; so viel Speicher anfordern wie max. notwendig; keine Rekursion; keine dynamische Datenstruktur (Listen und Bäume) möglich



Stack Datenzuteilung
nach dem LIFO-Prinzip; Rücksprungadressen oder lokale Var; Java: double, int etc.
Vorteil: Rekursion möglich; effiziente Zuteilung von Speicher
Nachteil: Grösse muss bekannt sein; Aufrufer kann nicht auf Werte des Aufgerufenene nach dessen Rückkehr zugreifen; nach Verlassen sind Werte verloren; Stack-Overflow möglich



Heap Datenzuteilung
Grössert Speicher; Jedesmal mit „new“ beanträgt man Referenz (Speicheradresse); Java: alle Variablen von nicht eingebauten Typen (Klassen), Strings, Arrays
Vorteil: Rekursion möglich; Grösse zur Laufzeit festlegen; dynam. Datenstrukturen (bspw. Listen)
Nachteil: Zuteilung/Freigabe rechenintensiv/kompliziert; Speicher muss explizit angefordert werden; Heap-Overflow möglich



Einfache Speicherverwaltung
- Schnittstelle um Speicher anzufordern/freizugeben; gibt Adresse des Speicherbereichs zurück; bei OOS unpraktisch, da Grösseangabe redundant & zusätzlich Konstruktor-Aufruf

Speicherverwalter mit freier Zuteilung
- mit zwei Listen (belegt- & frei-Liste) kann eine Speicherverwaltung realisiert werden
- Speicher anfordern: 1. Block mit angeforderter Grösse aus Frei-Liste als belegt markiert; 2. Wird in die belegt-Liste eingetragen; 3. RestBereich wird in Frei-Liste eingetragen; 4. Referenz auf den Bereich wird zurückgegeben
- Speicher freigeben: Speicher wird aus belegt-Liste entfernt und in Frei-Liste eingetragen
-> Problematik: Fragmentierung „Löcher im Hauptspeicher“, freier Speicher nicht mehr an einem Block
-> Lösung: Defragmentierungsprogramme regelmässig laufenlassen

Fehler bei Anforderung von Heap-Speicher
- *Vergessen Speicher anzufordern*: Konsequenz 1. Referenz-Var. enthält zufälligen Wert (auch Dangling Pointer genannt) 2. wird auf bel. SPeicherstelle zugegriffen / verändert 3. komische Werte evtl. Programmabbruch; Abhilfe: Alle Referenz-Variablen automatisch mit 0-Wert initialisieren(Java)
- *Zuwenig Speicher angefordert (bspw. Array):* Konsequenz 1. Es wird benachbarten Speicher überschrieben; Abhilfe: 1. Überprüfung innerhalb gültigen Bereich(Java); 2. Speichergrösse von Objekten automatisch bestimmt (Java); 3. Nur sichere Casts werden erlaubt (Java)

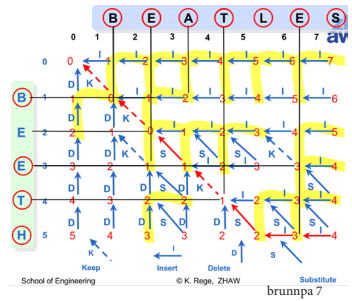
Fehler bei Freigabe von Heap-Speicher
-*Vergessen Speicher freizugeben (Memory-Leak)*: Konsequenz 1. benötigt immer mehr Speicher; 2. wird langsamer Abhilfe: 1 Storage-Manager gibt Speicher automatisch frei
-*Speicher wird freigegeben, obwohl noch verwendet (Dangling-Pointer)*: Konsequenz 1. Zuerst passiert nichts, aber wenn neu vergegeben, zeigt alte Var. immernoch in den alten Bereich -> zwei Pointer in unterschiedlichen Methoden, greifen auf selbe Adresse zu Abhilfe: 1 Storage-Manager gibt Speicher automatisch frei

Automatische Speicherverwaltung
- Hauptaufgabe der autom. Speicherverwaltung ist die Freigabe nicht mehr benötigten Speichers, dadurch verliert man ca. 5 - 10% der Performance
- Java macht dies // C/C++ unmöglich (Pointerarithmetik, Unsichere Casts, Mehrfachbelegung von Speichern)
- Speicherverwalter muss für diesen Zweck die Referenzketten traversieren 1. Alle Wurzelobjekte (statische Var. und Var. beim Aufruf im Stack) 2. Weiterverfolgten Ketten (Innerhalb Objekt) -> Java: java.lang.reflect

Referenzzähler
- Es wird gezählt wie viele Referenzen (Pointer) auf ein Objekt (bspw. Speicher) verweisen, falls keine Referenzen mehr vorhanden (Referenzzähler = 0) kann Speicher freigegeben werden.
Vorteil: einfach, geringer Verwaltungsaufwand; Speicher wird zum frühest möglichen Zeitpunkt freigegeben
Nachteil: Programmierer muss durchführen (Fehlerquelle); zusätzliche Operationen, keine zyklischen Datenstrukturen (bspw. doppelt verkettete Liste)

Smart-Pointer
- merken selber wenn neuer Wert zugewiesen oder nicht mehr zugreifbar sind
Vorteil: kein Fehler beim de/inkrementieren // Nachteil: keine zyklischen Datenstrukturen

Mark-Sweep Garbage Collector
- Speicher wird erst freigegeben wenn Bedarf- Suche nach Blöcken in zwei Phasen: 1. Markieren (von root, alle erreichbaren Blöcken) 2. Sweep (sequentiell durch Heap gehend, alle nicht markierten Blöcke löschen, Markierung aufheben)Vorteil: keine zusätzlichen Operationen; zyklische Datenstrukturen möglichNachteil: Aufwand, Programm muss gestoppt werden;



Levenshtein Distanz
- Aufwand: O(n*m), falls n=m -> O(n^2)