ADT, Stack, Queue

Abstrakte Datentypen (ADT)

Information Hiding:
- Nur so viel, wie für die Verwendung einer Klasse nötig ist, wird für andere sichtbar gemacht.
- Besteht aus verwendbaren Schnittstellen und aus einer ausserhalb des Moduls unsichtbaren Implementation.

Schnittstellen:
- Sicherstellung der **inneren Logik** der Daten (soll erhalten bleiben).
- Liefert **Zugriffsmethoden**, welche die Daten lesen oder verändern können.

Implementation:
- Implementation kann **verändert werden**, ohne dass dies das verwendende Programm merkt.

Java Realisierung:
- Mit Interfaces und Klassen: Beschreibung der Schnittstelle (interface) Implementation (class)
- Variable kann dann vom Typ des Interfaces sein.

Stack

⇨ **LIFO (last in, first out)**

Funktionen:
- push
- pop
- isEmpty

Queue

⇨ **FIFO (first in, first out)**

Funktionen:
- enqueue
- dequeue
- isEmpty

Implementation:
- Hat 2 Zeiger: OutIdx, InIdx (freie Pos.)

Array Implementation:
- Braucht Zyklus:

```
inIdx = (inIdx+1) % size;
```

Index	0	1	2	3	4	5	6	
Inhalt	?	?	34	27	11	77	52	
			outIdx					inIdx

Index	0	1	2	3	4	5	6
Inhalt	45	?	34	27	11	77	52
	inIdx		outIdx				

Priority Queue

Bekommen im enqueue eine Priorität mitgegeben, so dass das jeweilige Objekt in der Warteschlange nach vorne bis < nächste Prioobj. ist.

Listen

Aufwand (O-Notation)

- Laufzeitkomplexität in Bezug auf die Grösse der Eingabe darstellbar.
- Die O-Notation repräsentiert dabei eine Menge von Funktionen (bzw. ein Algorithmus) der ab n0 nicht schneller wächst als die Funktion, welche in den Klammern angegeben wird.

Beispiele von Algorithmen:

$s_1;\ s_2;\ s_3;\ s_4;\dots;s_k$
- O(1)

```
for(int i = 0; i< n; i++) s;
```
- wenn: s ist O(1)
- Zeitbedarf ist O(n)

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) s;
}
```
- Zeitbedarf ist O(n) • O(n) also O(n²)

```
h = 1;
while (h <= n) {
  h = 2 * h;
}
```
Komplexität ist also O(log n)

Beispiele von Funktionen:

$3n^3 + n^2 + 1000n + 500 = O(n^3)$

$f = 2^{n*2} + 3n + 5 = O(2^n)$

$f = n^{1.00001} + 1000 \log(n) = O(n^{1.00001})$

$f = 1.00001^n + x * 1000n = O(1.00001^n)$

$f = n^{-1} + n = O(n)$

$f = n(n-1)/2 = O(n^2)$

$$O(1) \subset O(\log\log n) \subset O(\log n) \subset O(\sqrt[c]{\log n}) \subset O(\sqrt[c]{n}) \subset O(n) \subset O(n \log n) = O(\log n!) \subset O(n^c) \subset O(c^n) \subset O(n!)$$

Iterator (ADT)

Problem:
```
for (int i = 0; i < list.size(); i++) {
    String element = (String)list.get(i);
    System.out.println(i + ": " + element);
}
```

Besser: (So braucht es nur O(n))
```
Iterator itr = list.iterator();
for (int i = 0; itr.hasNext(); i++) {
    Object element = itr.next();
    System.out.println(i + ": " + element);
}
```

LinkedList

Einfach verkettete Liste:
- Private Attribute: head (LinkedList), current (ListIterator)
- Einfügen am Anfang: O(1) am Ende: O(n)

Doppel verkettete Liste:
- Einfacheres Einfügen und Löschen (da next und prev).

⇨ Zyklisch doppelt verkettete Listen mit einem Dummy Anfangsknoten am elegantesten.

Sortierte Listen

- Elemente immer sortiert durch «insert()»
- Zb. bei PriorityQueues

Comparable:
- Die Listenobjekte müssen dabei Comparable implementieren («compareTo()»)
- Collections.sort(liste) sortiert eine Liste, welche Comparables hat

Comparator:
- Wenn nach anderem Kriterium verglichen werden soll.
- «compare(obj1, obj2)»

Arrays und Listen

Arrays:
- Indizierter Zugriff effizient (a[i])
- Einfügen und Löschen: ineffizient

Listen:
- Indizierter Zugriff ineffizient (a.get(i))
- Einfügen und Löschen (ohne kopieren)
- ArrayList: Schneller Zugriff, langsamere Mutationen

Erweitertes Java

Lischkovsches Substitutionsprinzip:
- Programm, welches ein Objekt von Klasse T verwendet, muss auch mit einer abgeleiteten Klasse S funktionieren.
- Dabei dürfen Definitionsbereiche (Aufrufparameter) nur erweitert werden und Wertebereiche (Rückgabe) nur verkleinert werden.

Generics:
- Erhöhen Sicherheit => Compilerprüfung
- extends: Class oder Interface muss erweitert sein
- super: Klasse muss diese Basisklasse/Subklasse haben
- <?>: Variable die alle Generics haben kann
- Performance besser, Weniger Laufzeittests, Unterschiedliche Typen
- Typenradierer: Macht alles bei Laufzeit zu Object wegen Java Versionkomp.

Rekursion

Ablauf

1. Basisfall (Verankerung)
   ⇨ Termination des Programms
2. Allgemeiner Fall (Induktionsschritt)
3. Progress
   ⇨ Falls Progress nicht Richtung Basisfall, dann könnte es schneller zu Stack-overflow kommen

Endrekursion (=Tailrekursion)

⇨ Letzte Aktion ist der rekursive Aufruf (im allgemeinen Fall / Else-Zweig)

⇨ Compiler Optimizations

⇨ Stackpointer unnötig

Lassen sich einfach in iterative Form überführen.

⇨ Generell: Jede Iteration kann in eine Rekursion überführt werden vice versa

```
void p(int i) {
  while ( <Bedingung>; i++ )
    <Anweisung>
}
```

```
void pR(int i) {
  if ( <Bedingung> ) {
    <Anweisung>
    pR(i+1);
  }
}
```

Beispiel Aufwandschätzung

```
void recursiveFun4(int n, int m, int o)
{
    if (n <= 0)
    {
        printf("%d, %d\n",m, o);
    }
    else
    {
        recursiveFun4(n-1, m+1, o);
        recursiveFun4(n-1, m, o+1);
    }
}
```
$O(2^n)$

```
int recursiveFun3(int n)
{
    if (n <= 0)
        return 1;
    else
        return 1 + recursiveFun3(n/5);
}
```
$O(\log(n))$

Bäume

Definition

Entweder leer oder bestehend aus einem Knoten mit keinem, einem oder mehreren disjunkten Teilbäumen.

$T = (V, E)$

V: *Knoten*

E: *gerichtete Kanten*

$\forall e \in E\ (\exists v_1, v_2 \in V\ (e = (v_1, v_2) \land v_1 \neq v_2))$

- Alle Knoten (ausser Wurzel) sind Nachfolger genau eines Knotens (parent)
- Knoten mit Nachfolger: Innere Knoten
- Knoten ohne Nachfolger: Blattknoten
- Knoten mit gleichem parent = sibling
- Pfade sind eindeutig
- Anzahl Kanten = Weglänge
- Tiefe: Anzahl Kanten + 1
- Gewicht: Anzahl Knoten Teilbaums

Binärbaum (=> DAG)

⇨ Maximal 2 Nachfolger Knoten

Höhe/Tiefe: k

Auf jedem Niveau Anzahl Knoten: 2^n

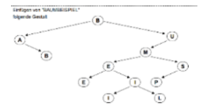
Maximal Anzahl Knoten: 2^k-1

Sortiert: (= Suchbaum)

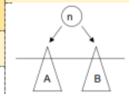
Für jeden Knoten im Baum gilt die Invariante

im linken Unterbaum sind alle kleineren Elemente $K_L <=^* k$

im rechten Unterbaum sind alle grösseren Elemente: $K_R >^* k$



Traversierung



Preorder Knoten zuerst: n, A, B

Inorder Knoten in der Mitte: A, n, B

Postorder Knoten am Schluss: A, B, n

Levelorder: n, a_0, b_0, a_1, a_2, b_1, b_2, ..

Preorder: (Verarbeitung am Anfang)
- Besuche die Wurzel
- Traversiere den linken Teilbaum (in preorder)
- Traversiere den rechten Teilbaum (in preorder)

```
private void preorder(TreeNode<T> node, Visitor<T> visitor) {
  if (node != null) {
    visitor.visit(node.getValue());
    preorder(node.left,visitor);
    preorder(node.right,visitor);
  }
}
```
call-back (Hollywood) Prinz

Postorder: (Verarbeitung am Schluss)
- Traversiere den linken Teilbaum (in postorder)
- Traversiere den rechten Teilbaum (in postorder)
- Besuche die Wurzel

```
private void postorder(TreeNode<T> node, Visitor<T> visitor) {
  if (node != null) {
    postorder(node.left,visitor);
    postorder(node.right,visitor);
    visitor.visit(node.getValue());
  }
}
```

Inorder: (Verarbeitung in der Mitte)
- Traversiere den linken Teilbaum (in inorder)
- Besuche die Wurzel
- Traversiere den rechten Teilbaum (in inorder)

```
private void inorder(TreeNode<T> node, Visitor<T> visitor) {
  if (node != null) {
    inorder(node.left,visitor);
    visitor.visit(node.getValue());
    inorder(node.right,visitor);
  }
}
```

Levelorder: (Verarbeitung Schichtenweise)
- Besuche die Wurzel
- Dann die Wurzel des linken und rechten Teilbaumes
- Dann die nächsten Schichten usw.

```
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.getValue());
        if (node.left !=null) q.enqueue(node.left);
        if (node.right != null) q.enqueue(node.right);
    }
}
```

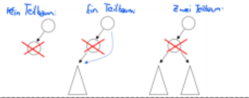
Mutationen von sortierten Bäumen

Einfügen unsortiert:
```
class BinaryTree<T> implements Tree<T>{
  private TreeNode<T> root;

  private TreeNode insertAt(TreeNode node, T x) {
    if (node == null) {
      return new TreeNode(x);
    }
    else {
      node.right = insertAt(node.right, x);
      // or
      // node.left = insertAt(node.left, x);
      return node;
    }
  }

  public void insert (T x) {
    root = insertAt(root, x);
  }
}
```
links oder rechts einfügen

Einfügen sortiert:
```
private TreeNode<T> insertAt(TreeNode<T> node, T x) {
  if (node == null) {
    return new TreeNode(x);
  }
  else {
    if (x.compareTo(element) <= 0)
       node.left = insertAt(node.left, x);
    else
       node.right = insertAt(node.right, x);
    return node;
  }
}
```

Löschen Fälle:

Kein Teilbaum: Ein Teilbaum: Zwei Teilbaum:



Löschen mit zwei Teilbäume:

```
// find node to replace
private TreeNode<T> findRepAt(TreeNode<T> node,
                      TreeNode<T> rep) {
  if (node.right != null) {
    node.right = findRepAt(node.right,rep);
  } else { //found
    rep.elements = node.elements;
    node = node.left;
  }
  return node;
}
```
Linker Teilbaum

copy ele.

Knoten aus Baum entfernt

Suchen im Binärbaum:
- Falls x == Wurzelelement: x gefunden
- Falls x > Wurzelelement: Suche im rechten Teilbaum fortsetzen, sonst im linken.

```
T search(TreeNode<T> node, T  x) {
  if (node == null) return node;
  else if (x.compareTo(node.element) == 0)
    return node.element;
  else if (x.compareTo(node.element) <= 0)
    return search(node.left,x);
  else
    return search(node.right,x);
}
```

Aufwand Binärbaum

Suchen:
- Bei einem vollen Binärbaum müssen Log2 Schritte durchgeführt werden. Man traversiert Baum von oben nach unten: Jede Ebene einen Knoten anfahren.
- Gleich wie Binarysearch Aufwand
- Sehr Effizient: 1000 Elemente -> 10 Schritte

Balancierte Binär- & B-Bäume

Zugriffszeiten und Tiefe

- Zugriffszeit (Such- & Einfügezeit) von Elementen ist proportional zur Tiefe

⇨ Ziel: Baum mit möglichst geringer Tiefe

⇨ Duplikate werden gezählt

Balancieren

Voller Baum: Alles bis auf letzte Stufe gefüllt.

Schlimmster Fall: Liste

Vollständig ausgeglichen: Die Gewichte der beiden Teilbäume unterscheiden sich max um 1.

AVL-Ausgeglichenheits-Bedingung:
- Tiefen unterscheiden sich max. um 1
- Beim Einfügen und Löschen muss diese Bedingung eingehalten bleiben

⇨ Einfacher als Gewichtsbedingung

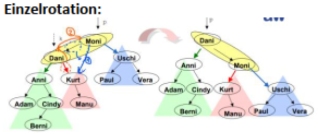
⇨ Suchoperationen: O(log(n))

Wichtig: AVL-Bäume sind sortiert.

Rotationen

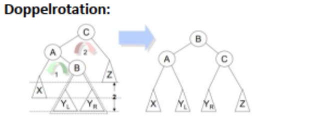
- Pro Knoten eine Zahl, die speichert, wie tief die nachfolgenden Teilbäume sind.

Einzelrotation:



⇨ Problem: Wenn ein Teilbaum Tiefe > 1 (=2) hat kann nicht mit Einzelrotation balanciert werden.

Doppelrotation:



Ziel beim Balancing: Entweder linker Teilbaum im linken Teilbaum oder rechter Teilbaum im rechten Teilbaum am grössten, so dass man diesen dann «nach oben» rotieren kann:

Oder

⇨ und dann Rotation nach rechts bzw. links.

B-Bäume

Entwickelt, um mit Bäume effizient auf Festplatten oder anderen sekundären Speichermedien zu arbeiten (mittels «Schlüssel»/«Index»)

Idee: Zugriffe auf Blöcke minimieren.

- Baum immer ausgeglichen
- Möglichst viele Infos in einem Block
- Möglichst breiter Baum
- Alle Knoten gleich gross

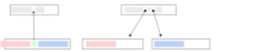
Bedingungen:

- B-Baum mit Ordnung n enthält jeder Knoten ausser der Wurzel min. n/2max.n-1 Schlüssel
- B-Bäume werden automatisch balanciert
- Jeder Knoten ist entweden ein Blatt oder hat m+1 Nachfolger (m = Anzahl Schlüssel des Knotens)
- Alle Schlüssel sortiert (innerhalb Knoten)
- Alle Blätter auf selber Stufe
- Mind. n/2 Unterbäume Baum=weniger hoch

Einfügen:

⇨ Immer in den Blättern.

- Falls Platz: Einfügen
- Falls Überlauf:



⇨ Extrem Fall: Vaterknoten überlauft (geht bis zur Wurzel => Tiefe + 1)

Löschen:

- Falls Blatt: Element Löschen
- Falls innerer Knoten: Gleich wie Binärbaum (Ersatzwert suchen: Rechtestes Element im linken Teilbaum)
- Falls Blatt und Unterlauf:

"Ausleihen" bei einem Nachbarknoten

oder

Zwei benachbarte Knoten können zu einem zusammengefasst

Suchen:

⇨ Ähnlich wie Binärbaum:

1) den Wurzelblock lesen
2) gegebenen Schlüssel S auf dem gelesenen Block suchen
3) wenn gefunden, Datenblock lesen fertig
4) ansonsten i finden, sodass $S_i < S < S_{i+1}$
5) Block Nr i einlesen, Schritte 2 bis 5 wiederholen

Tiefe des Baumes $\lceil \log_{\text{Anzahl Verweise}} \text{Anzahl Elemente} \rceil$

Anzahl Zugriffe: proportional zu Tiefe des Baumes

2-3-4 Bäume

- B-Baum mit max. 4 Nachfolgern (Ordnung 4 => 3 Schlüssel)

Rot-Schwarz Bäume

1. Jeder Knoten im Baum ist entweder rot oder schwarz.
2. Die Wurzel des Baums ist schwarz.
3. Jedes Blatt (NIL-Knoten) ist schwarz.
4. Kein roter Knoten hat ein rotes Kind.
5. Jeder Pfad, von einem gegebenen Knoten zu seinen Blattknoten, enthält die gleiche Anzahl schwarzer Knoten (Schwarzhöhe/Schwarztiefe).

Auf eine rote Kante muss immer eine schwarze Kante folgen

Vorteil: Einfachheit von Binärbäumen und Ausgeglichenheit von B-Bäumen

Zb. 2-3-4 Bäume als Binärbaum

Graphen

Definition

Ein Graph $G=(V,E)$ besteht aus einer endlichen Menge von *Knoten V* und einer Menge von *Kanten* $E \subseteq V \times V$.

Ein Graph $G = (V, E)$ kann zu einem **gewichteten Graphen** $G = (V, E, g_w(E))$ erweitert werden, wenn man eine Gewichtsfunktion $g_w: E \to int$ (oder $g_w: E \to double$) hinzu nimmt, die jeder Kante $e \in E$ ein **Gewicht** $g_w(e)$ zuordnet.

Gewichtete gerichtete Graphen: «Netzwerk»
Gewichtete Pfadlänge: Summe der Pfadkosten
Ungerichtete Graphen: Relationen symmetrisch
Einfacher Pfad: Falls kein Knoten doppelt

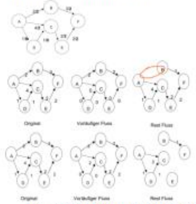
Topologisches Sortieren

Ziel:
Reihenfolge eines DAGs.
Implementationsvariante:

- Hole jeden Nachbarknoten
- Wenn dort einkommende Pfade - counter = 1: Mache etwas. Wenn nicht, addiere einen counter in diesem Knoten um 1.

Maximaler Fluss

Was in die Knoten hinein fliesst, muss auch wieder heraus (Kirchhof)
Kleinstes Gewicht entscheidet.
Bspw.: (Resultat = 4)



⇨ *Kein Pfad mehr von A-F im Rest = Fertig*

Traveling Salesman Problem

Kürzeste Reiseroute durch den ganzen Graphen, welche jeden Knoten einmal anfahrt.

Backtracking Zeitkomplexität: ~O(m^d)
wobei m durchschnittliche Anzahl Entscheidungen und d max. Tiefe

- Wenn mehr als ein Knoten nur über einen Weg erreichbar oftmals keine Lösung
- Bestimmung aller kurzen Wege: O(n!)
- Viele TSP-Algorithmen: O(a^n)
- Wenn eine einzige Kante hinzukommt, verdoppelt sich der TSP Aufwand
- (NP Problem: Exponentiell)

Trial & Error, Backtracking

Labyrinth

Idee:

- Gehe bis Kreuzung
- Wähle einen Weg und Folge diesen
- Falls Kreuzung: Folge einem weiteren Weg
- Falls Ziel: Fertig
- Falls Sackgasse: Gehe zurück zu einer Kreuzung und wähle einen anderen Weg

⇨ Aus Sackgasse zurück gehen=Backtracking
⇨ Es entsteht ein Entscheidungsbaum

Springer-,8 Damen-&Rucksackproblem

Springerproblem:

- Irgendwo Springer setzen, dieser soll dann jedes Feld genau einmal besuchen. Beendet wenn nr = n*n.

8 Damenproblem:

- Position für 8 Damen, so dass keine eine andere schlägt.

Rucksackproblem:

- Maximaler Betrag der Gegenstände bei einer Begrenzung (Rucksack)
- O(2^n) (aus n entweder true oder false)

Erschöpfende Suche

Durchsuche aller Möglichkeiten von Versuch & Irrtum.

Zielfunktion & «Branch&Bound»

- Berechnet für jeden Knoten im Entscheidungsbaum den Zielwert aus.
- Fahre nun die Kanten ab mit dem höchsten Zielwert (O(log(n))

⇨ Problem muss allerdings meistens schon gelöst sein (Baum aufgebaut)

Idee: Kostenfunktion die eine obere Schranke schätzt (immer besser als die exakte Zielfunktion)
Branch: Erstellt Zweige (Teilprobleme)

- Gehe den Pfad mit dem höchsten Bound zuerst entlang (oben 10).
- Korrigiere den b(v) Wert vom momentanen Knoten mit dem max. bound Wert der darunter liegenden Knoten (hier 4).
- Somit versucht man den anderen Weg mit b(v) = 7
- Der Knoten b(v) = 3 kann abgeschnitten werden, da b(v) = 10(4) besser ist. = «**Pruning**» (4 ist bereits eine Lösung)

Implementationen

V1: Adjazenzlisten:

- Jeder Knoten führt eine Liste von ausgehenden Kanten.

V2: Adjazenzmatrix:

- Jede Mögliche Kanten Kombination in Matrix. Darin werden bei gewichteten Graphen gewichte eingetragen, bei ungewichteten Graphen true/false.

	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

- **n** ist dabei die Knotenzahl und **m** die Kantenzahl eines Graphen G.

Traversierung

Tiefensuche: (= Preorder) => Stack
Ausgehend von Startknoten geht man vorwärts zu einem unbesuchten Knoten. Hat es keine weiteren unbesuchten Knoten geht man Rückwärts und betrachtet die noch unbesuchten Knoten.

```
void depthFirstSearch()
  s = new Stack();
  mark startNode;
  s.push(startNode)
  while (!s.empty()) {
     currentNode = s.pop()
     print currentNode
     for all nodes n adjacent to currentNode {
       if (!(marked(n)) {
         mark n
         s.push(n)
       }
     }
  }
}
```

Breitensuche: (= Levelorder) => Queue
Ausgehend von Startknoten betrachtet man alle benachbarten Knoten, bevor man einen Schritt weiter geht.

```
void breadthFirstSearch()
  q = new Queue()
  mark startNode
  q.enqueue(startNode)
  while (!q.empty()) {
     currentNode = q.dequeue()
     print currentNode
     for all nodes n adjacent to currentNode {
       if (!(marked(n)) {
         mark n
         q.enqueue(n)
       }
     }
  }
}
```

Kürzester Pfad

Ungewichtete Graphen: (Alle gleiches Gewicht)

1. Alle Knoten mit Distanz markiert und wie er erreicht wurde:

2. Vom Endpunkt aus Rückwärts

Gewichtete Graphen:

1. Gleich wir oben, aber korrigiere Einträge für Distanzen:

2. C Wird neuen kleineren Wert gesetzt. Gehe nun Weiter bis der neue Weg länger als angetroffene Weg

Dijkstra: (mit Priority Queue)
3 Gruppen:

- Besuchte Knoten
- Benachbarte Knoten
- Unbesehene Knoten (n.prev == null)

O(n^2) (Im schlimmsten Fall)
Setze Startknoten in Queue.
Solange Queue nicht leer ist:

- Nimm erstes Node aus Queue
- Markiere dieses
- Wenn = goal: Fertig
- Für alle Kanten aus Node:
  - Wenn nicht markiert &(kleinere Distanz als bereits besucht oder unbesehen):
    - Berechne Distanz
    - Setze Distanz und prev
    - Lege in Queue mit Distanz als Priority

Spannbaum

Ein Baum, der alle Knoten eines Graphens verwendet, so dass die Summe der Gewichte minimal ist (ungleich kürzester Pfad).

Greedy (Gierige) Algorithms

Eine Art Algorithmus, welcher bei der Wahl eines Folgezustands zum Zeitpunkt der Wahl den Zustand, welcher am meisten Gewinn bringt, wählt (Bspw. Dijkstra, Gradientenverf.).
Problem: Stecken bleiben in lokalen Maxima.

Horizont Effekt

Bspw. beim Schach kann ein Programm nicht alle möglichen Kombinationen durchrechnen sondern muss nach n Zügen abbrechen.
Problem: Gleich hinter dem Horizont kann sich die gefundene Lösung als schlecht erweisen.

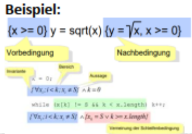
⇨ Die ausgewählte Lösung muss weiter ausgewertet werden.

Schnelles Suchen & Hashing

Invariante

Identifizieren Vor- und Nachbedingungen.

Beispiel:



Binäres Suchen

Suchen in einem sortierten Array:

1. Zwei Indizes l und r wählen (l = -1, r = a.length)
2. Index m wählen der in der Mitter zwischen l und r liegt
3. Falls a[m] = S: fertig
4. Falls a[m] < S: l = m
5. Falls a[m] > S: r = m
6. Falls l+1 >= r: S nicht in a

Aufwand: O(log2(n)) => in jedem Durchgang wird r-l halbiert.

Einfaches Suchen in 2 Listen a,b:

```
for (int i = 0; i < a.length; i++) {
  for (int j = 0; j < b.length; j++) {
    if(a[i].equals(b[j])) return i;
  }
}
```

Aufwand: O(n*m)

Sortiertes Suchen in 2 Listen a,b:

Aufwand: O(n)

Übersicht:

- **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 $O(\log_2(n))$

Schlüssel und Hashing

Menge der Schlüsselwerte klein:

- Einfügen & Sortieren: O(1)
- Werte in Array an ihre Indexpositionen
- Bspw. char[] h = new char[256]

Hashfunktion:

- Um Wertebereich auf einen kleineren abzubilden (->Einfach: X modulo tablesize)
- Gut: Gleichmässig über Wertebereich verteilt, Berechnung effizient

Contract von hashCode/compareTo/equals:

1) Ein Objekt muss während seiner Lebensdauer immer denselben Hashwert zurückliefern, solange der Zustand (Wert) des Objekts nicht verändert wurde
- Wenn aber die JVM neu gestartet oder eine andere JVM verwendet wird darf der Hashwert ändern!

2) wenn equals == true, dann müssen die Objekte denselben Hashwert liefern
3) wenn equals == false, dann sollten die Objekte unterschiedliche Hashwerte liefern (d.h. Haswerte müssen nicht eindeutig sein).

Erlaubt: Unterschiedlicher Wert -> gleicher Hash

Kollisionen:

- Abhängig von Güte der Hash Funktion und Belegung der Zellen.

⇨ Separate Chaining: Überlauflisten bei gleichen Hash Wert
⇨ Open Addressing:
  - Linear Probing: Suche sequenziell nach nächster freier Zelle.
  - Quadratic Probing: In wachsenden Schritten F+1, F+4, F+9 ... F+i^2

Aufwand: i.d.R. O(1)
Bei hohem Load Faktor/ungünstigen Daten bricht Performance ein
Löschen: Rehashing oder Label «del»

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

⇨ HashTables können anders wie BinaryTrees nicht degenerieren
⇨ Impl. einfacher als AVL trees
⇨ Suche in einem bestimmten Bereich nicht möglich

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

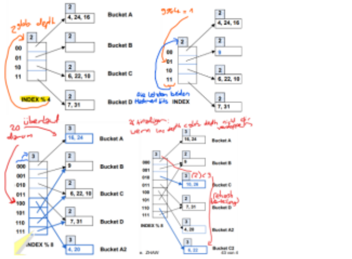
Extendible Hashing

Hashtabelle überlauft, was tun?:

- Passt noch in Hauptspeicher: Überlaufketten, Kopieren in neue Tabelle (vergrössern und rehashing)
- Passt nicht mehr in Hauptspeicher: Extendible Hashing

Idee:

- Verwende Verzeichnis von Verweisen zu Buckets (analog zu B-Bäume)



Suchen in Texten

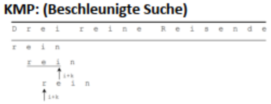
String Suche

Bruteforce: (Stringteil in String)

```
static int indexOf(String str, String pattern) {
  for (int i = 0; i < str.length() - pattern.length() + 1; i++) {
    int k;
    for (k = 0;
         k < pattern.length() && str.charAt(i+k) == pattern.charAt(k);
         k++);
    if (k == pattern.length()) return i;
    }
  return -1;
}
```

Aufwand: O(n * m)

KMP: (Beschleunigte Suche)



Problem: der String-Zeiger (i+k) wird immer wieder zurück gesetzt

Grundidee von Knuth, Morris & Pratt: wenn ich bis zum "i" im Suchmuster gekommen bin, dann kann ich direkt weitergehen, wenn im Suchmuster dies das erste Vorkommen ist:

im Muster "r e i n" kommt "i" nur einmal vor.

Es wird ein Automat so konstruiert, dass im Text selber nur vorwärts gelesen wird.

Aufwand: O(n + m)

⇨ Next Table enthält wie viel Mal eine Zeichenkette vorkommt:

a	a	b	a	a	a	b
0	1	0	1	2	2	3

⇨ In modernen Prozessoren ist jedoch die Bruteforce Variante trotzdem doppelt so schnell (linearer Speicherzugriff).

Index und Indexierung

Suche in grossen Textsammlungen und Suchmaschinen.

Index: Datenstruktur für die unstrukturierten Daten

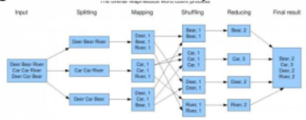
Indexierung: Prozess der Erstellung eines Indexes.

⇨ Für einen schnellen Zugriff auf Textdokumente über erhaltene Stichwörter, müssen Texte zu einem invertierten Dateisystem aufbereitet werden.

Invertiertes Dateisystem:

- Direkte Dateien: Textdokumente
- Index: Alphabetische Liste mit Stichwörter; Verweist auf eine invertierte Datei.
- Invertierte Datei: Verbindungen zwischen dem Index und den direkten Dateien.

Beispiel MapReduce Google: (Funktionsweise gleich wie Java Parallelstreams)



⇨ (Parallel Stream: Viele Daten, Viele Ops)

Unscharfe/Fuzzy Suche

String-Matching-Algorithmen; Auch ähnliche Zeichenketten («Did you mean: …»)

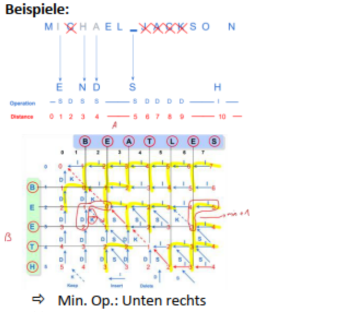
Levenshtein Distanz(Analogie:Hemming)

Wie viele Editieroperationen sind minimal nötig, um den einen String in den andern überzuführen.

⇨ Ersetzen, Einfügen, Löschen

⇨ O(n+m) (O(n*m) mit Tabelle)

Beispiele:



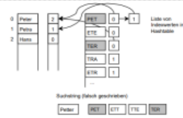
⇨ Min. Op.: Unten rechts

$m = |u|$
$n = |v|$
$D_{0,0} = 0$
$D_{i,0} = i, 1 \le i \le m$
$D_{0,j} = j, 1 \le j \le n$

$$D_{i,j} = \min \begin{cases} D_{i-1,j-1} & +0 \text{ falls } u_i = v_j \\ D_{i-1,j-1} & +1 \text{ (Ersetzung)} \\ D_{i,j-1} & +1 \text{ (Einfügung)} \\ D_{i-1,j} & +1 \text{ (Löschung)} \end{cases}$$
$1 \le i \le m, 1 \le j \le n$

Trigram Suche

Wörter Match (Wort mit meisten Index Übereinstimmungen).



- Fehlertolerante Suche
- Effizient für grosse Datenbestände
- Index: Wort in 3-Buchstaben Gruppen
- Eignet sich für verdrehte Worte

Phonetische Suche

E

Ziffer	Repräsentierte Buchstaben
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M, N
6	R

D

Ziffer	Repräsentierte Buchstaben
0	a, e, i, o, u, ä, ö, ü, y, j, h
1	b, p, f, v, w
2	c, g, k, q, x, s, z, ß
3	d, t
4	l
5	m, n
6	r
7	ch

Britney → BRTN → **B635**	bewährten → BRTN → **B635**
Spears → SPRS → **S162**	Superzicke → SPRZCK → S16222 → **S162**

⇨ Wort besteht aus ersten Buchstaben gefolgt von 3 Ziffern (kurze Worte = 0)

⇨ Vokale und Konsonanten (H, W, Y) ignorierbar

Sortierverfahren

Internes / Externes Sortieren

Intern:

⇨ Sortieren im Arbeitsspeicher.

⇨ Wenn Datensätze und Umfang gering.

Extern:

⇨ Es können nicht gleichzeitig alle Datensätze im Arbeitsspeicher gehalten werden.

⇨ Daten können vor Sortierprozess geteilt werden. Danach intern sortiert und wieder zusammengefügt werden.

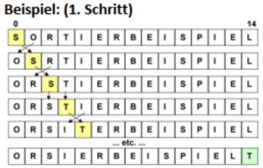
Sortierschlüssel:

- Kriterien, nach denen Datensätze sortiert oder gesucht werden.
- Eindeutig, wenn höchstens 1 Datensatz (bspw. MartikelNr).
- Für das Sortieren nicht zwingend notwendig aber sinnvoll.
- Bspw. Collator Klasse für Locale Comparator.

Bubble Sort

Sortieren durch Vertauschen von Nachbarfeldern.

Beispiel: (1. Schritt)



Ablauf:

- Der Array wird von links nach rechts mehrmals durchwandert bis sortiert.
- Dabei wird verglichen: a[i] < a[i+1] ? mache nichts : tausche; i++
- Analogie zu Luftblasen, welche nach oben steigen.

⇨ Nach spätestens n – 1 Durchläufe sortiert.

⇨ Optimiert: Testen ob überhaupt swap. Wenn nicht dann break.

Aufwand:

Best: O(n) (sortedcheck), Average & Worst: O(n^2)

Schlimm wenn Array verkehrt sortiert

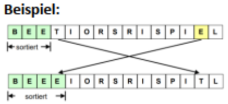
Selection Sort

Sortieren durch Auswählen des jeweils kleinsten der verbleibenden Elemente und Einfügen in den Bereich der sortierten Elemente.

Idee:

- Array in 2 Bereich aufteilen: sortiert & unsortiert.

Beispiel:



(Jetzt k = 3)

Ablauf:

- Suche kleinstes Element im unsortierten (rechten) Teil und Tausche es mit dem Anfangssymbol (Index k) des unsortierten Bereichs.
- Inkrementiere den Zähler (k).
- Wenn k == n dann fertig.

Aufwand:

Immer O(n^2) (Vorsortiertheit kann nicht ausgenutzt werden.

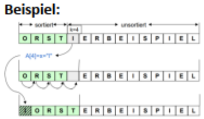
Vorteil:

⇨ Weniger Vertauschungen als Bubblesort (vgl. n Mals swapen vs. 1 Mal swapen)

Insertion Sort

Analogie Kartenspiel: Eine Karte wird genommen und an der richtigen Stelle eingeordnet.

Beispiel:



Ablauf:

- Aus unsortiertem Teil entnehmen wir ein Element.
- An der richtigen Stelle im sortierten Teil einordnen.
- Dabei wird das Element «herausgenommen» (Lücke entsteht). Der sortierte Teil wird nach rechts geschoben bis richtiger Platz gefunden:

Aufwand:

Best: O(n), Sonst O(n^2)

⇨ Durchschnittlich nur die Hälfte aller maximal notwendigen Vergleiche (bei Selection Sort müssen immer alle gemacht werden bis kleinstes gefunden); Es müssen jedoch mehr swaps ausgeführt werden

⇨ Gut wenn Datensätze kurz (Java = Pointer), Gut wenn gut vorsortiert (beste überhaupt)

Ordnung vs. Laufzeit

Ansatz nach Aufwand:

- *Bsp: $O(n^2)$ ->Polynom 2-ten Grades: $k_1 * n^2 + k_2 * n + k_3$*

Messen von verschiedenen Laufzeiten

- *$0.8 = k_1 * 10'000^2 + k_2 * 10'000 + k_3$*
- *$3.1 = k_1 * 20'000^2 + k_2 * 20'000 + k_3$*
- *$7.1 = k_1 * 30'000^2 + k_2 * 30'000 + k_3$*

nach k_1, k_2 und k_3 auflösen

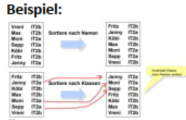
⇨ Man kann nun k1, k2, k3 bestimmen und dann für weitere grössere n extrapolieren. Man kann auch schätzen, in dem man den grössten Term nimmt.

Stabilität

Wie werden Elemente mit gleichen Sortierschlüsseln behandelt.

Sei S = ((k_0, e_0), … , (k_{n-1}, e_{n-1})) eine Sequenz von Elementen: Ein Sortieralgorithmus heisst *stabil (stable)*, wenn für zwei beliebige Elemente (k_i, e_i) und (k_j, e_j) mit gleichem Schlüssel k_i = k_j und i < j (d.h. Element i kommt vor Element j), i < j auch noch nach dem Sortieren gilt (Element i kommt immer noch vor Element j).

Beispiel:



Übersicht:

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

Teile und Herrsche Prinzip

Zerlege das Problem in kleinere, einfacher zu lösende Teile. Oftmals rekursiv.

```
Sort (Menge a)
      if (Menge der Datenobjekte klein genug)
            Ordne sie direkt;
      else {
            Zerlege in zwei Teilmengen
            Sort(Teilmenge1); Sort(Teilmenge2)
            Füge Teilmengen geordnet zusammen
      }
}
```

Bei der Zerlegung sollten die Teile möglichst gleich gross sein.

Quick Sort

Zerlege Problem in Teilaufgaben.

Ablauf:

```
Methode Sortiere (A){
     Konstruiere die Partition A = A1 W A2;
     Sortiere A1;
     Sortiere A2;
 }
```

Für alle A1 <= W <= A2

Partitionierung:

- Wähle ein Element W
- Suche Elemente die links und rechts auf falscher Seite sind und tausche sie.
- Wiederhole bis l >= r (sich kreuzen)

Wahl des Pivots (W):

A[l] das (der Position nach) linke Element von A;

A[r] das (der Position nach) rechte Element von A;

A[mid] das (der Position nach) mittlere Element von A mit mid = (l+r)/2

Strategie 1: nehme eines der drei Elemente

Strategie 2: nehme das (wertmässig) mittlere der drei Elemente

Strategie 3: nehme das arithmetisch Mittel der drei Elemente

Median wäre aufwendig (Laufzeitvorteil ginge verloren)

Beispiel ungünstige Partitionierung:

Optimal: So dass A in 2 Teile geteilt wird.

Beispiel gute Partitionierung: (mit Median)

Code:

```
public static int partition (int[] a, int left, int right) {
       int pivot = arr[(left + right) / 2];
       while (left <= right) {
              while (arr[left] < pivot) {
                     left++;
              }
              while (arr[right] > pivot) {
                     right--;
              }
              if (left <= right) {
                     swap(a,left,right);
                     left++;
                     right--;
              }
       }
       return left;
   }
void quickSort(int[] a, int left, int right) {
     if (left < right) {
          int l = partition (a,left,right);
          quickSort(arr, left, l-1);
          quickSort(arr, l , right);
     }
}

public void quickSort(int[] a){
   quickSort(a, 0, a.length-1);
}
```

⇨ Nach Partitionierung wird wieder rekursiv partitioniert bis fertig.

Aufwand:

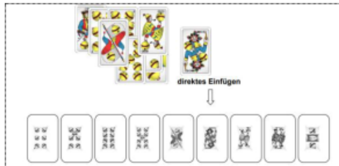
Log2(n) Mal muss ein Bereich geteilt werden und dies n Mal => O(nlogn). Ungünstiger Fall: Jeder Partitionierungsschritt nur jeweils das Vergleichselement (d.h A1 oder A2 = {}) => O(n^2). Tritt jedoch sehr selten auf! QuickSort bei grösseren ungeordneten Daten immer erste Wahl.

Distribution Sort

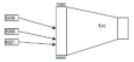
Ohne Vergleiche (swap).

Idee:

- Entsprechend dem Sortierschlüssel in Fächer verteil (O(n))
- Zusammengetragen (O(n))



Wie bei Hashtable direkte Adressierung:



- ⇨ Schneller geht's nicht (O(n))
- ⇨ Verfahren muss an den Sortierschlüssel angepasst werden
- ⇨ Nur bei Schlüsseln mit kleinem Wertebereich
- ⇨ Allgemeines Hashing problematisch da es sein kann: ungleichmässige Verteilung
- ⇨ Sehr spezifisch je nach Problemstellung
- ⇨ Bei vorsortierten Daten Laufzeittechnisch Quick schneller (unabhängig O(...))

Übung: Extrapolation

Ein O(n*log n) Sortieralgorithmus brauche 1 Sekunde für 10'000 Elemente; wie lange braucht er für 100'000 Elemente?

- es soll nur der erste Koeffizient betrachtet werden.

1 = k1 * n * log(n) + k2 * n + k3
Nur erster Koeffizient:
k1 = 1 / (n*log(n)) = 1 / (10000 * log(10000))= 1 / 40000 = 0.000025
x = k1 * 100000 * log(100000) = 2.5 * 5 = 12.5s

Optimalitätssatz

Satz: Ein Sortieralgorithmus, der darauf beruht, dass Elemente **untereinander verglichen** werden, kann bestenfalls eine Komplexität von O(n⋅log(n)) im Worst Case haben.

- ⇨ Es kann keinen besseren Vergleichssortieralgorithmus als O(nxlogn) im Worst Case geben.

Externes Sortieren

Phase 1: Sortieren-Verteilen

- Lade jeweils einen Teil der zu sortierenden Datei in den Speicher.
  - Sortiere diese
  - Schreibe die sortierten einzelnen Daten in mind. 2 Ausgabedateien
- ⇨ Es entstehen Folgen von sortierten Abschnitten

Phase 2: Mischen

- Lese von den Ausgabedateien das erste Element
- Schreibe das kleinere und lese das nächste Symbol der gleichen Datei. Länge der geordneten Abschnitte verdoppelt sich.

Aufwand: (n Sequenzen, m Eingabedateien)

- Anzahl Mischphasen: O(logm(n))
- Mischen: O(n * logm(n))

Wahl des Sortierverfahrens

- **Internes oder externes Verfahren**
  - Sortieren im Hauptspeicher
  - Sortieren im Hauptspeicher und Sekundärspeicher (Platte)
- **Methode des Algorithmus:**
  - Vertauschen
  - Auswählen
  - Einfügen
  - Rekursion
  - Verteilen
  - Mehrphasen: Sortieren-Mischen
- Nach Effizienz:
  - Laufzeit: $O(n^2)$ oder $O(n \times \log n)$ oder sogar $O(n)$
  - Speicherbedarf: Wieviel Speicher wird zusätzlich zu dem für die zu sortierenden Daten benötigt?
- Allgemein / mit speziellen Voraussetzungen:
  - allgemein: Benötigt nur Schlüsselvergleiche
  - allgemein: Ändern der Reihenfolge oder Angabe der Reihenfolge
  - allgemein: Wird die Reihenfolge von Datensätzen mit gleichem Sortierkriterium durch den Algorithmus geändert? →Stabilität
  - speziell: Der Algorithmus setzt eine bestimmte Struktur der Schlüssel voraus.
- Bibliotheksfunktion -> Collections.sort oder Arrays.sort
- wenige Datensätze (weniger als 1000).
  - Laufzeit unerheblich.
  - möglichst einfachen Sortieralgorithmus wählen (also Insertion-Sort, Selection-Sort oder Bubble-Sort).
- vorsortierte Datenbestände
  - dann Insertion- oder Bubble-Sort .
- viele ungeordnete Daten
  - dann Quick-Sort bevorzugen.
- viele Daten, ungeordnet, sehr oft zu sortieren
  - Distribution-Sort an das spezielle Problem anzupassen
- sehr viele Daten
  - externes Sortierverfahren in Kombination mit schnellem internem

Optimierung

Optimierung durch Kombination von Algorithmen (bspw. Insertion und Quick). Oder auch durch **Parallelisierung**.

Naiv:

- Neuer Thread pro Partitionierung
- Kann ein if gesetzt werden (wenn n klein dann ohne Thread lösen)

Automatische Speicherverwaltung

Ziel: Freigabe des nicht mehr benötigten Speichers.

Referenzzählung:

- Es wird gezählt wie viele Referenzen auf ein Objekt verweisen.
- Wenn Counter = 0, dann kann Objekt gelöscht werden.
- Zuweisung counter++; Wegnahme counter—
- ⇨ Zyklische Datenstrukturen können nicht freigegeben werden (Memory Leak)
- ⇨ Bspw. doppelt verkettete Listen

als Methoden:

```
int rc;

void addRef() {
    rc ++;
}
```

```
int release() {
  if (--rc == 0) {
    delete(this);
  }
  return rc;
}
```

Smart Pointers:

- Merken selber, wenn ihnen ein neuer Wert zugewiesen wird oder nicht mehr zugreifbar sind.
- ⇨ Keine Fehler beim Erhöhen und Verringern des Referenzzählers, da automatisch
- ⇨ Immer noch keine zyklischen Strukturen

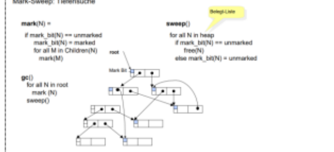
```
class MySmartRef {
    MyObject ref;
    void assign (MyClass obj)  {
        if (ref != null) ref.release();
        ref = obj;
        if (ref != null) ref.addRef();
    }
    MyObject get() {
      return ref;
    }
```

```
class Test {
   MySmartRef left = new MySmartRef();

void foo() {
    MyClass a = new MyClass();
    left.assign(a); // left = a;
    left.assign(null); // left = null;
}
```

Mark-Sweep GC:

- Nach Blöcken suchen, die freigegeben werden können
- Mark:
  - Alle erreichbaren Blöcke von der Wurzel werden markiert
- Sweep:
  - Sequenziell durch Heap
  - Alle nicht markierten Blöcke werden freigegeben
  - Markierungen von markierten Blöcken werden gelöscht
- ⇨ Keine zusätzlichen Pointer Zuweisungen (wie Counter)
- ⇨ Zyklische Strukturen auflösbar
- ⇨ Das Programm muss allerdings während GC gestoppt werden
- ⇨ Löcher entstehen

Mark-Sweep: Tiefensuche

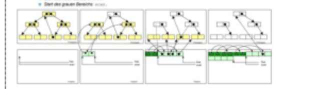


Kompaktierende Speicherverw.

Fragmentierung des Heap: (Problem)

Copying GC:

- Speicher in zwei gleiche Teile aufgeteilt (Aktuelle Daten, Obsolete Daten)
- Neue Daten -> aktuellen Semi Space
- Wenn kein Platz mehr: Alle Daten in anderen Semi Space kopiert und Rollen vertauscht
- ⇨ Es entstehen keine Löcher
- ⇨ Suche nach freien Blöcken entfällt
- ⇨ Doppelt so viel Speicher benötigt



- ⇨ Langer Programmunterbruch
- ⇨ Schlechte Performance (Lokalität des RAM Zugriffs geht verloren)

Threadpools:

- Ineffektiv Threads zu erstellen die nicht benötigt sind (> core Anzahl). Darum Threadpools.
- Threads wieder verwenden.
- Im Fall von hoher Parallelität Thread Safe Collections verwenden.

Fork/Join:

- Teile Herrsche Prinzip auf Parallelität angewandt

Threads	Fork/Join
Do subclass **Thread**	Do subclass **RecursiveTask<V>**
Do override **run**	Do override **compute**
Do call **start**	Do call **invoke, invokeAll, fork**
Do call **join**	Do call **join** which returns answer
	or
	Do call **invokeAll** on multiple tasks

Speicherverwaltung

Datenzuteilungen

- Fehler mit dynamischem Speicher schwer zu finden, da erst bei Laufzeit.
- *Java => Garbage Collector*

Statische Datenzuteilung:

- Garantiert kein Speicherfehler
- Einfach
- Zu Beginn eines Programms alles vordefiniert
- Java: static
- Keine Rekursion, Keine dyn. Datenstr.
- Schon zu Beginn klar, ob Speicher ausreicht

Stack Datenzuteilung:

- Aufruf einer Prozedur wird Speicherbereich reserviert
- Via Stackcounter
- Rekursion möglich
- Daten nach Prozedur gehen verloren
- Stack-Overflow möglich

Heap Datenzuteilung:

- Mit new und delete (C++)
- Rekursion möglich
- Grösse zur Laufzeit
- Zuteilung/Freigabe rechenintensiv
- Heap-Overflow möglich

Einfache Speicherverwaltung

- new: malloc() mit Objektgrösse (redundante Info) + Konstruktor Aufruf

Speicherverwalter:

- Belegt-Liste, Frei-Liste
- Problem: Fragmentierung (Löcher entstehen)
- Lösung: kompaktiert (Löcher werden zusammengeschoben)

Fehler:

1. Vergessen den Speicher anzufordern
=> Referenzvariable enthält zufälligen Wert;
Java gelöst mit null

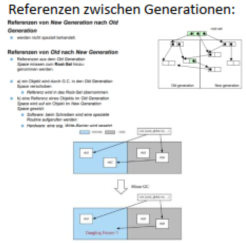
2. Zu wenig Speicher angefordert
=> Benachbarter Speicher wird überschrieben;
Überprüfen ob gültiger Bereich z.B. Array-Index (Java)

3. Vergessen Speicher freizugeben (Memory Leak)
=> Programm benötigt immer mehr Speicher;
Muss auf Disk ausgelagert werden
Java gelöst mit GC

4. Speicher freigegeben obwohl er noch verwendet wird (Dangling Pointer)
=> Zuerst nichts, aber wenn Speicherverwalter Platz wieder vergibt, passieren kuriose Dinge
Java gelöst mit GC

Generational GC:

- Nachteile bisherige GC Verfahren: Ganzer Speicher muss durchlaufen werden, Programm muss dabei angehalten werden
- ⇨ Lebensdauer von Objekt unterschiedlich
- ⇨ Darum Objekt in Generationen unterteilen und neue Generationen mehr durchsuchen.
- ⇨ Neue Objekte werden im New Generation Space angelegt.
- Referenzen zwischen Generationen:



- ⇨ Referenzen zwischen Old und New muss richtig aktualisiert werden (wenn bspw. Objekt in Old kopiert) sonst Fehler.
- ⇨ Root-Set: Dient für GC zum Überprüfen, welche Objekte noch erreichbar sind.
- ⇨ Bei Old Generation Space müssen alle Referenzen betrachtet werden (Major Collection), New Generation Space unabhängig von Old (Minor Collection)

Anmerkung: ...^n => NP

Weak References

Referenzen auf Objekte, die Java, wenn sie nicht anderweitig referenziert werden, freigeben kann.

- ⇨ Datenstrukturen, die Sammlungen von anderen Objekten beinhalten, benötigen interne Referenzen auf diese Objekte.
- ⇨ Werden von GC gefunden und traversiert
- ⇨ Wrapper Klasse WeakReference
- ⇨ Falls Objekt gelöscht, dann wird zu null gesetzt

Es gilt: Sind starke Referenzen auf ein Objekt weg und nur noch schwache Referenzen vorhanden, wird es vom GC trotzdem als nicht erreichbar bezeichnet und entfernt.

Finalizer:

- Besser AutoClosable
- Wird nicht zwingend aufgerufen
- «finalize» Methode (muss super.finalize aufrufen)
- Problem: finalize Methode wird bevor ein Objekt endgültig freigegeben wird, aufgerufen. Wenn dabei eine Referenz auf das Objekt wiederhergestellt wird könnte dies zu unerwarteten Verhalten führen.

Übung Ordnung

```
a) int n = K; // was passiert wenn n eine Fliesskommazahl ist?
   while (n > 0) {
       n = n / 3;
   }
```

↳ wird iteriert bis n/=? < kleinst darstellbare Zahl => Rundung

$1 + \sqrt[3]{n} = O(\sqrt[3]{n})$

```
b) for (int i = 0; i < n; i ++) {
       for (int j = i; j < n; j++) {
           foo(i,j);
       }
   }
```

(n+1)

$\frac{n^2}{2} = O(n^2)$

```
c) for (int i = 0; i < n; i ++) {
       for (int j = n; j > 0; j = j/2) {
           foo(j);
       }
   }
```

(n+1)

$n \cdot \sqrt{n} = O(n\sqrt{n}) = O(n^{\frac{3}{2}})$

Letztes: O(n * log(n))
Sqrt wenn Math.pow(i, 0.5)