# Objektorientierte Programmierung 2 | OOP2

Zusammenfassung

## FILE I/O, Serialisierung

**Serialisierung:** Objekt codieren  
**Deserialisierung:** Objekt wiederherstellen  
  
**Byte Streams:** Byteweises Lesen von Dateien  
FileInputStream, FileOutputStream

**Character Streams:** Zeichenweises Lesen von Dateien (Unicode) FileReader, FileWriter

**Ein Bild, das Text, Screenshot, Schrift, Handschrift enthält.

Automatisch generierte Beschreibung**Ein Bild, das Text, Schrift, Reihe, Screenshot enthält.

Automatisch generierte Beschreibung

**Ein Bild, das Text, Screenshot, Schrift, Reihe enthält.

Automatisch generierte Beschreibung**

Datei binär lesen  
try (var in = new **FileInputStream**(“”)) {  
 int b = in.read();  
 for (int i = 0; b >= 0; ++i){   
 // b == -1 bedeutet Dateiende  
 if (i % 16 == 0) { // Zeilennummer  
 System.*out*.printf(“%n%04X;”, i);  
 }  
 System.*out*.printf(“ %02X”, b);  
 //b == gelesenes Byte  
 b = in.read();  
 }  
} // in.close() in implizitem finally

File-Outputtry(var out = new **FileOutputStream**(“”)){  
 while (…) {byte b = …;  
 out.write(b);}  
}  
//Datei neu anlegen bzw. überschreiben  
new FileOutputStream(“test.data”, true)  
//Anhängen, falls Datei existiertFile-Readertry (var reader = new **FileReader**(“”)) {  
 int value = reader.read();  
 while (value >= 0) { //-1 end of file  
 char c = (char) value;  
 value = reader.read(); }}

Zeilenweises lesen

try (var reader = new **BufferedReader** (new FileReader(“quotes.txt”)) {  
 String line = reader.readLine();  
 while (line != null) {  
 System.out.println(line);  
 line = reader.readLIne();  
 }  
}

File-Writer

try (var writer = new **Filewriter**(“test.txt”, true)) {  
 writer.write(“Hello!”); //String  
 writer.write(“\n”); //Einzelner char  
}

Die Serializable-Interface implementation zeigt an, dass die Klasse serialisiert werden kann:

class Person implements Serializable {…}  
//...  
try(var stream = new **ObjectOutputStream**( new **FileOutputStream**(“serial.bin”))) {  
 stream.writeObject(person);  
}

Objekt aus Bytestrom deserialisieren

try(var stream = new ObjectInputStream( new FileInputStream(“serial.bin”))){  
 Person p=(Person) stream.readObject();  
}

Sollte man nicht verwenden: Nicht interoperabel, Sicherheitsprobleme, Änderungen der Implementierung einer Klasse sind schwierig

Deshalb: JSON zur Datenübertragung

Serialisierung mit Jackson

class Person {

public String firstName;

...

Person(){}

}

var contacts= new ArrayList<Personen>();

// Personen hinzufügen

String jsonString = new ObjectMapper(). writeValueAsString(contacts);

Deserialisierung mit Jackson

ArrayList<Person> contacts = new ObjectMapper(). readerFor(new TypeReference<ArrayList<Person>>(){}) .readvalue(jsonString);

## Generische Programmierung

Die generische Programmierung ermöglicht es, eine Implementierung mit unterschiedlichen Datentypen zu verwenden (z.B. String statt Integer). So wird ermöglicht, dass eine Stack-Implementierung sicher unterschiedliche Datentypen aufnehmen kann.   
Verwendung von Typen (Klassen und Schnittstellen) als Parameter bei der Definition von Klassen, Schnittstellen und Methoden.  
**Vorteile:**

* Strengere Typüberprüfungen zur Kompilierzeit: Ein Java-Compiler wendet eine strenge Typüberprüfung auf generischen Code an und gibt Fehler aus, wenn der Code die Typsicherheit verletzt. Die Behebung von Kompilierfehlern ist einfacher als die Behebung von Laufzeitfehlern, die schwer zu finden sein können.
* Keine Casts notwendig

Generische Klasse: public class ListStack<T> implements Stack<T> { /\*...\*/ }   
Generisches Interface: interface Iterator<E> { /\*...\*/ }

Generische Methode: public <E> Stack<E> multiPush(E value, int times) { /\*...\*/ }

Typ-Parameter in spezifischer Methode

public class StackTest {  
 public**<E>** Stack **<E>** multiPush(**E** value, int times) {  
 var result = new Stack<**E**>();  
 for (int i = 0; i<times; i++) {  
 result.push(value);  
 }  
 return result;  
 }   
}

Bei gemischten Argumenttypen wir der nächste gemeinsame Basistyp verwendet. Z.B. Integer und Double -> Number

**<T> T** majority(**T** x, **T** y, **T** z);  
Number n = majority(1, 3.141. 1);

Generische Interfaces

Iterable

interface **Iterable**<T> {  
 Iterator<T> iterator();  
}

Iterator

interface **Iterator**<E> {  
 boolean hasNext();  
 E next();  
}

Iterator für Stack-Klasse

**class Stack<T> implements Iterable<T>** {  
 private Entry<T> top;  
 // ...  
 public Iterator<T> iterator() {  
 return new StackIterator<T>(top);  
 }  
}

**class StackIterator<T> implements Iterator<T>** {  
 private Entry<T> current;  
 StackIterator(Entry<T> top) {  
 current = top;  
 }  
 public boolean hasNext() {  
 return current != null;  
 }  
 public T next() {  
 var value = current.getValue();  
 current = current.getPrevious();  
 return value;  
 }  
}

Einsatz: Enhanced For-Loop

for (String s : stringStack) {...}

Generics vs. Arrays

Kompiliert:

String[] arrayOfStrings = new String[5]  
Object[] arrayOfObjects = arrayOfStrings  
arrayOfObjects[0] = Integer.valueOf(2);

Funktioniert nicht:

ArrayList<String> arrayOfStrings = new ArrayList<>();  
ArrayList<Object> arrayOfObjects = arrayOfStrings;  
ArrayList<Object> arrayOfObjects = new ArrayList<>();  
ArrayList<String> arrayOfStrings = arrayOfObjects;  
Object[] arrayOfObjects = new Object[10]  
String[] arrayOfStrings = arrayOfObjects

Type Bounds

Beschränken des Parameter-Typ auf Subtypen einer bestimmten Klasse.

class GraphicStack<**<T extends Graphic>** extends Stack**<T>** {  
 public void drawAll() { ... }  
}

* Typargumente einschränken
* Stellt Kompabilität sicher
* Erhöht Flexibilität

Type Erasure

Typbeschränkungen zur Komplilierzeit, Löschen der Typ-Informationen zur Laufzeit. Macht es wegen backward compatibility.

Generische Klasse

class Stack**<T>** {  
 void push(**T** value) { ... }  
}

Konsequenzen von Type Erasure für MyClass<T>

* Keine instanceof T
* Type-Casts zu T sind ungeprüft
* Kein new T()
* Keine primitiven Typen als Typ-Argumente
* Methoden müssen auch zur Laufzeit eindeutig identifizierbar sein

Wildcards

Funktion ohne Wildcards

Public static **<T>** List**<T>** mergeToList( Collection**<T>** coll1, Collection**<T>** coll2) {  
 var result = new ArrayList<T>();  
 merge(coll1, coll2, result);  
 return result;  
}  
List**<Number>** result1 = CollectionFunctions.mergeToList  
(new HashSet**<Double>**(); new ArrayList**<Integer>**());

Funktion mit Wildcards

Public static **<T>** List**<T>** mergeToList( Collection  
**<? extends T>** coll1, Collection**<? extends T>** coll2) {  
 var result = new ArrayList**<T>**();  
 merge(coll1, coll2, result);  
 return result;  
}

Typparameter-Varianten

**Invarianz C<T>:** Keine Änderungen erlaubt. Ein generischer Typparameter bleibt unverändert und kann nicht durch einen anderen Typparameter ersetzt werden, selbst wenn eine Subtyp- oder Supertyp-Beziehung besteht.

Grenzen generischer Invarianz

private static <T> void move(Stack<T> from, Stack<T> to) {  
 while(!from.isEmpty()){  
 to.push(from.pop());  
 }  
}  
//Main-Funktion  
var RStack = new Stack<Rectangle>();  
var GStack = new Stack<Graphic>();  
  
GStack.push(new Rectangle()); //Funktioniert  
move(RStack, GStack); //Funktioniert nicht

**Kovarianz C<? extends T>:** Nur abgeleitete (Unter-)Typen sind erlaubt. Ein generischer Typparameter kann durch einen abgeleiteten Subtyp ersetzt werden, aber nicht durch einen Supertyp.  
**Kontravarianz C<? super T>:** Nur Supertypen sind erlaubt. Ein generischer Typparameter kann durch einen Supertyp ersetzt werden, aber nicht durch einen Subtyp.  
**Bivarianz C<?>:** Eine Kombination aus Kovarianz und Kontravarianz. Ein generischer Typparameter kann sowohl durch einen abgeleiteten Subtyp als auch durch einen Supertyp ersetzt werden.

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte Beschreibung

## **Annotations & Reflections**

Annotations sind Metadaten über das Programm und kein Teil des Programmcodes.

Anwendungen: Informationen für den Compiler: Fehler erkennen oder Warnungen unterdrücken.Verarbeitung zur Kompilier- und Bereitstellungszeit: Annotations verarbeiten, um Code oder XML-Dateien zu generieren.

Vordefinierte Annotationen:

@Override  
@Deprecated  
@SuppressWarnings(value=»unchecked»)  
@FunctionalInterface  
...

Eigene Annotation definieren:

public @interface MyAnnotation { ... }

Auswahl annotierter Methoden

... for (var m : methods) {  
 if(m.isAnnotationPresent(Profile .class)) {  
 Analyzer.profileMethod( ... );  
 }  
}

Mit Reflections kann das Programm seine eigene Struktur, sein Verhalten und seine Metadaten zur Laufzeit untersuchen und manipulieren.

Funktion von Reflections:

* Informationen über Klassen, Methoden und Felder
* Dynamische Methodenaufrufe
* Abschalten von Zugriffsbeschränkungen
* Auslesen von Annotations

Anwendungen:

* Programmier-Tools (z.B. Debugger, Object Inspector, Class Browser)
* Dynamisches Laden von Code

## Ein Bild, das Diagramm, Reihe, Screenshot, Design enthält. Automatisch generierte BeschreibungArray & Listen

Abstrakter Datentyp Liste

|  |  |  |
| --- | --- | --- |
| **public interface List<E> extends Collection<E>** | | |
| **boolean** | add(E element) | Appends the specified element to the   end of this list |
| **boolean** | add(int index,   E element) | Inserts the speficied element at the   specified position in this list |
| **boolean** | contains(Object o) | Returns true if this list contains the   specified element |
| **E** | get(int index) | Returns the element at the specified   position in the list. |

Definition Array

* Speichern von gleichartigen Objekten
* Wahlfreier Zugriff mit Index bis
* Array speichert Referenzen auf Objekte – ändert sich referenziertes Objekt, so ändert sich Inhalt des Arrays
* int[] anArray = new int[10]

public void add(GameEntry entry) {  
 int newScore = entry.getScore();  
 if(isHighScore(newScore)) {  
 if(numEntries < board.length) {  
 numEntries++;  
 }  
 int j = numEntries - 1;  
 for (; j>0 && board[j-1].getScore() < newScore; j--) {  
 board[j] = board[j - 1];  
 j--;   
 }  
 board[j] = entry;  
 }  
}

Nachteil von Arrays: Müssen bei nicht mehr vorhandenem Platz umgespeichert werden. Verkettete Liste ist für das besser geeignet.

Verkettete Liste

* Sequenz von Knoten
* Jeder Knoten besitzt ein Element und einen Zeiger auf den nächsten Knoten

Ein Bild, das Screenshot, Reihe, Diagramm, Schrift enthält.

Automatisch generierte BeschreibungEinfach verkettete Liste

A picture containing text, clock, gauge

Description automatically generatedDoppelt Verkettete Liste

public class Node<E> {  
 private E element;  
 private Node<E> next;  
 public Node(E e, Node<E> n) {  
 element = e;  
 next = n;  
 }  
 public E getElement() {  
 return element;  
 }  
}

Anwendungen: Undo Funktionalität, Hash Tabellen, Graphen  
Doubly-Linked-List: Jeder Knoten speichert Verbindung zum Vorgänger und Nachfolger

|  |  |  |
| --- | --- | --- |
|  | **Worst Case** | **Best Case** |
| **Suchen** |  |  |
| **Einfügen** |  |  |
| **Löschen** |  |  |

Element hinzufügen: Element erstellen next/prev richtig setzen und next/prev von dem Element vor und nach dem neuen Element umhängen. Size-Value   
Element löschen: next/prev des zu löschenden Elements auf null. Next des vorherigen Elements und prev des nächsten Elements vom zu löschenden Element umhängen. Size-Value

Array vs List

Laufzeitvergleich

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Lesen** | | **Element einfügen** | |
| **Array** |  | |  | |
| **Liste** |  | |  | |
| Array | | List | |
| Kann multidimensional sein | | Kann nur eindimensional sein | |
| Länge statisch und fixe Länge | | Dynamische Länge | |
| Schneller wegen fixer Länge | | Langsamer wegen dynamischer Länge | |
| Type-unsafe da keine Generics | | Type-safe da Generics möglich | |
| Lesen: | | Lesen: | |
| Element einfügen: | | Element am Anfang/Ende einfügen: | |

## Sortieralgorithmen

Insertionsort

Elementarer Sortieralgorithmus, eignet sich für kleinere Datenmengen oder für Einfügen in eine bereits sortierte Liste.   
Nimmt jeden Wert aus dem Array und fügt ihn in einen neuen Array an der richtigen Stelle ein.   
**Bsp. 1 4 3 2: [1]; [1,4]; [1,3,4]; [1,2,3,4]**  
public static void insertionSort(int array[]) {

for (int step = 1; step < array.length; step++) {

int key = array[step];

int j = step - 1;

// For descending order, change < to >

while (j >= 0 && key < array[j]) {

array[j + 1] = array[j];

--j;

}

array[j + 1] = key;}}

**Worst Case:**  Liste ist verkehrt herum sortiert

**Average Case:**

**Best Case:** vorsortierte Liste

Selection Sort

Von der unsortierten Liste das grösste oder kleinste Element in eine neue Liste einfügen. Danach das nächstgrössere oder kleinere Element wählen, bis die unsortierte Liste leer ist. Laufzeit unabhängig von Eingabe.  
**Bsp. 1 4 3 2: [1]; [1,2]; [1,2,3]; [1,2,3,4]**

public static void selectionSort(int[] arr) {  
 int n = arr.length;  
 for (int i = 0; i < n - 1; i++) {  
 int minIndex = i;  
 for (int j = i + 1; j < n; j++) {  
 if (arr[j] < arr[minIndex]) {minIndex = j;}  
 }  
 int temp = arr[minIndex];  
 arr[minIndex] = arr[i]; arr[i] = temp; }}

**Worst Case:**

**Average Case:**

**Best Case:**

Shell Sort

Weit auseinander liegende Einträge (Start normalerweise bei ) austauschen, um teilweise sortierte Arrays zu erzeugen. Diese dann mit Insertion Sort sortieren.

4-sortieren: Jedes 4te Element vergleichen und vertauschen, sodass diese 4 sortiert sind. Schritt wiederholen mit jedem 2. Element, am Schluss Array mit Insertion Sort sortieren.

public static void shellSort(int[] a) {  
 int n = a.length;  
 //3x+1 increment sequence: 1,4,13,40,121,…  
 int h=1;  
 while (h < n/3) { h = 3 \* h + 1; }  
 while (h >= 1) { //h-sort the array  
 for (int i = h; i < n; i++) {  
 for (int j = i; j >= h && a[j] < a [j-h]; j = j-h) {  
 swap (a, j, j-h);  
 }} h /= 3; }}

**Worst Case:**

**Average Case:**

**Best Case:** vorsortierte Liste

Bubble Sort

Durch Liste iterieren und in zweier-Paaren vergleichen und Positionen vertauschen falls unsortiert. Wiederholen, bis Liste sortiert ist. Wenn der Algorithmus einmal durch die ganze Liste geht, ohne etwas zu verschieben, heisst das, dass die Liste sortiert ist.  
**Bsp. 1 4 3 2: [1,4,3,2]; [1,3,4,2]; [1,3,2,4]; [1,3,2,4]; [1,2,3,4]**

public static void bubbleSort(int[] array) {

int size = array.length;

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) {

// change > to < to sort in descending order

if (array[j] > array[j + 1]) {

// swapping occurs if elements are in wrong order

int temp = array[j];

array[j] = array[j + 1];

array[j + 1] = temp;

}}}}

**Worst Case:**

**Average Case:**

**Best Case:** vorsortierte Liste

Counting Sort

Für jedes Eingabeelement wird die Anzahl der kleineren Elemente und die Anzahl der gleichen Elemente ermittelt. So kann jedes Element an der richtigen Position im Output Array platziert werden.  
**Bsp. 1 4 3 2: [ ,2, , ]; [ ,2,3, ]; [ ,2,3,4]; [1,2,3,4];**

public static void countingSort(int[] arr) {  
 int n = arr.length;  
   
 // Step 1: Find the range of input elements  
 int min = Integer.MAX\_VALUE;  
 int max = Integer.MIN\_VALUE;  
 for (int num : arr) {  
 min = Math.min(min, num);  
 max = Math.max(max, num);  
 }  
  
 // Step 2: Create the count array  
 int range = max - min + 1;  
 int[] count = new int[range];   
  
 // Step 3: Count the occurrences  
 for (int num : arr) { count[num - min]++; }  
  
 // Step 4: Calculate cumulative counts  
 for (int i = 1; i < range; i++) {  
 count[i] += count[i - 1];  
 }  
  
 // Step 5: Build the sorted array  
 int[] sorted = new int[n];  
 for (int i = n - 1; i >= 0; i--) {  
 int num = arr[i];  
 int index = count[num - min] - 1;  
 sorted[index] = num;  
 count[num - min]--;}  
  
 // Step 6: Copy the sorted array back to the input array  
 System.arraycopy(sorted, 0, arr, 0, n);  
}

**Worst Case:**  bzw.

**Average Case:** bzw.

**Best Case:** bzw.

Ein Bild, das Text, Screenshot, Diagramm, Reihe enthält.

Automatisch generierte BeschreibungHeapsort

Binörer Baum mit folgenden Eigenschaften: Baum ist vollständig, Schlüssel jedes Knotens kleiner oder gleich als Schlüssel seiner Kinder.  
Wurzel enthält immer kleinestes Element.

* Wurzelelement herausnehmen
* Letztes Element an Stelle von Wurzel verschieben
* Falls in nächster Stufe kleineres Element vorhanden, kleinstes Element mit Wurzel vertauschen, Element so lange nach unten tauschen bis der Baum wieder die korrekten Eigenschaften hat
* Nächste Wurzel aus dem Array nehmen

public static void **percolate** (Comparable[] arrayToSort, int startIndex, int last) {  
 int i = startlndex;  
 while(hasLeftChi1d(i, last)) {  
 int leftChild = getLeftChild(i);  
 int rightChild = getRightChild(i);  
 int exchangeWith = 0;  
 if(arrayToSort[i].compareTo(arrayToSort[leftChild])>0)  
 { exchangeWith = leftChild; }  
 if(rightChild <= last && arrayToSort[leftChild].   
 compareTo(arrayToSort[rightChild])>0) {  
 exchangeWith = rightChild;  
 }  
 if(exchangeWith == 0 || arrayToSort[i].   
 compareTo(arrayToSort[exchangeWith]) <= 0 ) {  
 break;  
 }  
 swap(arrayToSort, i, exchangeWith);  
 i = exchangeWith;  
 }  
}

public static void **heapSort**(Comparable[] arrayToSort) {  
 int i;  
 heapifyMe(arrayToSort);  
 for (i = arrayToSort.length – 1; i > 0; i--) {  
 swap(arrayToSort, 0, i) //1. Element mit n. tauschen  
 percolate(arrayToSort, 0, i-1);//Heap wiederherstellen  
 }  
}

//Alternative

public void **sort**(int arr[]) {

int n = arr.length;

// Build max heap

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

// Heap sort

for (int i = n - 1; i >= 0; i--) {

int temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

// Heapify root element

heapify(arr, i, 0);

}

}

void **heapify**(int arr[], int n, int i) {

// Find largest among root, left child and right child

int largest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

if (l < n && arr[l] > arr[largest]) {largest = l;}

if (r < n && arr[r] > arr[largest]) {larges = r;}

// Swap and continue heapifying if root is not largest

if (largest != i) {

int swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

heapify(arr, n, largest);

}

}

**Worst Case:**

**Average Case:**

**Best Case:**

## Algorithmenparadigmen

Greedy

In jedem Teilschritt die möglichst optimale Lösung wählen. Optimiert lokal, um global zu optimieren.

Beispiel: Fülle einen Rucksack, der 35kg Platz hat, mit folgenden Werten: 30kg, 20kg, 15kg. Der Greedy Algorithmus wird zuerst das Element mit 30kg wählen, weil es in diesem Teilschritt die optimale Lösung ist. Ist jedoch nicht die globale optimale Lösung. -> Liefert eine schnelle, aber nicht unbedingt optimale Lösung.

public static void calcSolution(HashSet<String> statesNeeded, HashMap<String, HashSet<String>> stations) {

var finalStations = new HashSet<String>();

while (!statesNeeded.isEmpty()) {

String bestStation = "";

var statesCovered = new HashSet<String>();

for (String station : stations.keySet()) {

var covered = new HashSet<String>(statesNeeded);

covered.retainAll(stations.get(station));

if (covered.size() > statesCovered.size()) {

bestStation = station;

statesCovered = covered;

}

}

statesNeeded.removeAll(statesCovered);

finalStations.add(bestStation);

}

System.out.println(finalStations);

}

Divide and Conquer

Problem rekursiv in kleinere Subprobleme aufteilen. Subprobleme lösen und Lösung zusammenfügen.

Beispiel Binärsuche, funktioniert nur, wenn Elemente nach einer logischen Funktion sortiert sind.

Ein Bild, das Text, Reihe, Schrift, Screenshot enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Reihe, Schrift, Screenshot enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Schrift, Reihe, Zahl enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Schrift, Reihe, Zahl enthält.

Automatisch generierte Beschreibung

Public static <T extends Comparable<T>> boolean searchBinary(List<T> data, T target, int low, int high) {  
 if (low > high) {  
 return false;  
 } else {  
 int pivot = low + ((high – low) / 2);  
 if (target.equals(data.get(pivot))) {  
 return true;  
 } else if (target.compareTo(data.get(pivot)) < 0){  
 return searchBinary(data, target,low, pivot -1 );  
 } else {  
 return searchBinary(data, target, pivot+1, high);  
 }  
 }  
}

Backtracking

Trial & Error. Falls aktueller Zweig nicht zur Lösung führt, letzte Entscheidung zurücksetzen und anderen Pfad probieren.

Diagram

Description automatically generatedDiagram

Description automatically generated with medium confidence

Vorgehen:

* Position auf Feld markieren
* Rekursionsabbruch (Alle Felder besucht, Ausgang gefunden)
* Alle Operationen probieren:  
  Neues Feld / Koordinate festlegen  
  Überprüfen, ob Feld gültig ist und noch nicht besucht wurde  
  Wenn Ja: Überprüfen, ob rekursiver Aufruf «true» zurückgibt
* Sonst Backtracking: Markierung vom Feld entfernen und «false» zurückgeben

procedure BACKTRACK (K: Konfiguration) {  
 if[K ist Lösung] then [gib K aus]  
 else for each [direkte Erweiterung K’  
 von K] do BACKTRACK(K’)  
}

**Backtracking example (near light)**

static final int *N* = 5;  
static final int[] *row* = {2, 1, -1, -2, -2, -1, 1, 2, 2};  
static final int[] *col* = {1, 2, 2, 1, -1, -2, -2, -1, 1};

private static boolean isValid(int x, int y) {  
 return x >= 0 && y >= 0 && x < N && y < N;  
}  
public static boolean knightTour(int[][] visited, int x, int y, int pos) {  
 visited[x][y] = pos;//Feld markieren  
   
 //Abbruchsbedingungif (pos >= *N* \* *N*) { return true; }  
  
 //Alle Züge probierenfor (int k = 0; k < 8; k++) {  
 int newX = x + *row*[k];  
 int newY = y + *col*[k];  
  
 // Ist neue Feld im Spielbrett und unbesuchtif (*isValid*(newX,newY) && visited[newX][newY] == 0) {  
 // Wird Abbruchsbedingung rekursiv erreicht?if (*knightTour*(visited, newX, newY, pos + 1)) {  
 return true;  
 }  
 }  
 }  
   
 //Falls 4, 5 nicht erfüllt sind, backtracking  
visited[x][y] = 0;  
 return false;  
}

**Backtracking example Labyrinth**

public void walk(int x, int y) {  
 if (step(x, y)) {  
 maze.setField(x, y, State.*WALKED*);  
 }}  
//Backtracking methodpublic boolean step(int x, int y) {  
 amountOfSteps++;  
 System.*out*.println(maze);  
 //Return true in case the goal was foundif (maze.checkField(x, y, State.*GOAL*)) {return true;}  
 //Return false falls wand oder verwendeter pfad erreichtif (maze.checkField(x, y, State.*WALL*) ||   
 maze.checkField(x, y, State.*WALKED*)) {return false;}  
 //Mark current location as walkedmaze.setField(x, y, State.*WALKED*);  
  
 //Try to go Rightif (step(x, y + 1)) {return true;}  
 //Try to go Upif (step(x - 1, y)) {return true;}  
 //Try to go Leftif (step(x, y - 1)) {return true;}  
 // Try to go Downif (step(x + 1, y)) {return true;}//Mark current location as backtrackedmaze.setField(x, y, State.*BACKTRACKED*);  
 return false;//Go back  
}

**Backtracking example Sudoku**

public boolean checkRow (int row, int num) {  
 for (int col = 0; col < 9; col++) {  
 if (sudokuArray[row][col] == num {return false;}  
 }  
 return true;  
}

public boolean checkBox(int row, int col, int num) {  
 row = (row / 3) \* 3;  
 col = (col / 3) \* 3;  
 for (int r = 0; r < 3; r++) {  
 for (int c = 0; c < 3; c++) {  
 if (sudokuArray[row+r][col+c] == num) {return false}  
 }  
 }  
 return true;  
}  
  
public boolean solve(int row, int col) {  
 if (row == 8 && col == 9) return true;  
 if (col == 9) {  
 row++;  
 col = 0;  
 }  
 if (sudokuArray[row][col] != 0) {  
 return solve(row, col+1);  
 } else {  
 for (int num = 1; num < 10; num++) {  
 if (checkRow(row,num) && checkCol(col,num) &&  
 checkBox(row, col, num)) {  
 sudokuArray[row][col] = num;  
 }  
 if (solve(row, col + 1)) {return true;}  
 }  
 }  
 sudokuArray[row][col] = 0;  
 return false;  
}

Dynamische Programmierung

Lösung mit bereits berechneten Ergebnissen finden.

public static long dynamisch(int n) {  
 long[] f = new long[n+2];  
 f[0] = 0;  
 f[1] = 1;  
 for (int i = 2; i <= n; i++) {  
 f[i] = f[i-1] + f[i-2]  
 }  
 return f[n];  
}

Rekursion

Rekursionsabbruch:Werte der Parameter, für die kein rekursiver Aufruf ausgeführt wird. In jeder Rekursion muss es einen Base Case geben, welcher die Rekursion nicht weiterführt. Rekursive Aufrufe: Rufen sich selbst wieder auf und bewegen sich Richtung Base CaseLineare Rekursion: Ein rekursiver Aufruf startet höchstens einen weiteren rekursiven AufrufBinäre Rekursion: Rekursiver Aufruf macht höchstens zwei rekursive AufrufeMehrfache Rekursion: Rekursiver Aufruf macht mehr als 2 weitere rekursive AufrufeEndrekursion: Funktion, bei der rekursiver Aufruf letzter Schritt ist. Weniger Speicherbedarf auf Call Stack und kann in iterative Funktion umgewandelt werdenTeile und Herrsche:Problem aufteilen, in kleinere Probleme aufteilen und Funktion rekursiv mit kleinerer Eingabemenge aufrufen

Ohne Endrekursion:

private static int recsum(int x) {  
 if (x == 0) {   
 return 0;   
 } else { //Addition ist letzte Anweisung  
 return x + recsum(x - 1);  
 }  
}

Mit Endrekursion:

private static int tailrecsum(int x, int total) {  
 if (x == 0) {  
 return total;  
 } else {  
 return tailrecsum(x - 1, total + x);  
 }  
}

Iterative Version von Endrekursion:

private static int tailsum(int x) {  
 int total = 0;  
 for (; x > 0; x--) { total += x; }  
 return total;  
}

Lineare Rekursion:

static int[] reverseArray(int[] a, int I, int j) {  
 if (i < j) {  
 int temp; temp = a[j];  
 a[j] = a[i]; a[i] = temp;  
 reverseArray(a, i + 1, j - 1);  
 }  
 return a;  
}

Palindrome Checker

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

Automatisch generierte BeschreibungPublic boolean palindrom (String word) {  
 int length = word.length();  
 if (length < 2) {return true; } else {  
 if (word.charAt(0) != word.charAt(length-1)) {  
 return false  
 } else {  
 return palindrom(word.substring(1, length-1));  
 }  
 }  
}

**Dezimal zu Binär**

public static int decimalToBinary(int decimal) {

if (decimal == 0) {

return 0;

} else {

return (decimal%2)+(10\*(decimalToBinary(decimal/2)));

}

}

## Diagram Description automatically generatedAbstrakter datentyp (adt)

ADT: Abstraktion einer konkreten Datenstruktur, beschreibt das «was», nicht das «wie». In Java als Interface realisiert. Er beschreibt Attribute, Operationen auf den Attributen und Ausnahmen und Fehler.  
Datenstruktur: Speichert / Organisiert Daten, konkrete Implementierung der Schnittstelle  
Ziel: Kapselung (Nutzung ausschliesslich über Schnittstelle) / Geheimnisprinzip (Interne Realisierung ist verborgen)  
Beispiel: Der abstrakte Datentyp Stack beschreibt eine Reihe von Funktionen (z.B. pop(), push()). Diese können auf unterschiedliche Art und Weise (z.B. mit Array oder einer Liste) implementiert werden.

## Analyse von Algorithmen

Empirische Laufzeitmessung

Algorithmus implementieren und mit unterschiedlichen Eingaben ausführen. Ergebnisse aufzeichnen und vergleichen.

Herausforderungen: Ergebnisse können durch Störungen in Hardware, Software und Systembelastung verfälscht werden. Algorithmus muss implementiert werden, Eingaben beeinflussen Ergebnis.

Vorteilhaft: Wenn Laufzeit abhängig ist von externen Faktoren, tatsächliche Performance wird gemessen. Oder wenn Algorithmus bereits auf einem konkreten System implementiert ist und die Laufzeit dort gemessen werden kann.

Big-O Notation

Mit Big-O Notation lässt sich Laufzeit und Speicherverbrauch eines Algorithmus mittels algebraischer Terme beschreiben. Konzentration auf dem Worst Case.

ist falls reelle, positive Konstante ,   
Ganzzahl-Konstante , sodass

Ein Bild, das Text, Schrift, Reihe, Screenshot enthält.

Automatisch generierte Beschreibung

Vorteile bei Betrachtung vom Worst-Case  
Einfache Analyse, vermeidet Unsicherheit, Gut für Anwendungen, die garantierte Antwortzeiten benötigen

**Limitation**

* Konstanten werden ignoriert (Wenn diese sehr gross ist, ist dies ein Problem)
* Kann Bereich geben wo Funktion mit schlechterer Komplexität schneller ist

Regeln

* Falls ein Polynom vom Grad ist, dann ist
* Jeweils optimalste Funktion verwenden (tiefst mögliche Potenz)
* So stark wie möglich vereinfachen  
   ist und nicht

Ein Bild, das Reihe, Diagramm, Schrift, Steigung enthält.

Automatisch generierte Beschreibung

|  |  |  |
| --- | --- | --- |
| Wachstumsrate | Name | Beschreibung |
|  | Konstant | Statement (1 Linie Code) |
|  | Linear | In Hälfte teilen (Binäre Suche) |
|  | Logarithmisch | Loop |
|  | Linearithmische | Sortieralgorithmen (Merge Sort) |
|  | Quadratisch | Doppelter Loop |
|  | Kubisch | Dreifacher Loop |
|  | Exponential | Brute Force |

Ablauf

* Pseudocode verfassen
* Neben den Code schreibem wie viele Operationen, bzw. wie Laufzeit
* Text

  Description automatically generatedZusammenzählen, höchster Polynomwert ist Laufzeit

Beweisen Sie: ist

**Gesucht**: und , sodass für

**Lösung**:

## Data Structures

**Ein Bild, das Reihe, Diagramm enthält.

Automatisch generierte Beschreibung**Stack

Grundprinzip: Last in – First Out (LIFO). Zeitkomplexität: Pop und Push Operationen sind beide

* **void push(<E> e):** Element auf Stapel legen.
* **<E> pop():** Oberstes Element entfernen und zurückgeben
* **<E> top():** Liefert zuletzt eingefügtes Element, ohne dieses zu entfernen
* **int size():** Zahl gespeicherter Elemente
* **boolean isEmpty():** Zeigt, ob Stack leer ist

class Stack {

private int arr[];

private int top;

private int capacity;

Stack(int size) {

arr = new int[size];

capacity = size;

top = -1;

}

public void push(int x) {

if (isFull()) {

System.out.println("OverFlow Program Terminated");

System.exit(1);

}

System.out.println("Inserting " + x);

arr[++top] = x;

}

public int pop() {

if (isEmpty()) {

System.out.println("STACK EMPTY");

System.exit(1);

}

return arr[top--];

}

public int size() {return top + 1;}

public Boolean isEmpty() { return top == -1; }

public Boolean isFull() {return top == capacity - 1;}

}

Linked List

Sequenz von Knoten (Element + Zeiger zum nächsten Knoten).



**Ein Bild, das Entwurf, Diagramm, technische Zeichnung, Rechteck enthält.

Automatisch generierte Beschreibung**Queue

Grundprinzip: First in – First Out (FIFO). Einfügen am Ende, Entfernen am Anfang. Zeitkomplexität: Pop und Push Operationen sind beide

* **enqueue(e):** Element am Ende der Queue einfügen
* **E dequeue():** Element vom Anfang der Queue entfernen und zurückgeben
* **E first():** Liefert erstes Element, ohne es zu entfernen
* **integer size():** Anzahl gespeicherter Elemente
* **boolean isEmpty():** Queue leer?

public class Queue {

int SIZE = 5;

int items[] = new int[SIZE];

int front, rear;

Queue() { front = -1; rear = -1;}

boolean isFull() {

return front == 0 && rear == SIZE - 1;

}

boolean isEmpty() {return front == -1;}

void enQueue(int element) {  
 if (isFull()) {

System.out.println("Queue is full");

} else {

if (front == -1) {

front = 0;

rear++;

items[rear] = element;

System.out.println("Inserted " + element);

}}}

int deQueue() {

int element;

if (isEmpty()) {

System.out.println("Queue is empty");

return -1;

} else {

element = items[front];

if (front >= rear) {

front = -1;

rear = -1;

} else {front++; }

System.out.println("Deleted -> " + element);

return (element);

}}}

Ringbuffer

public void enqueue(E element){

if(this.storedElements == this.capacity){

throw new IllegealStateException();

} else{

int r = (this.frontElement + this.storedElements) %   
 this.capacity;

this.data[r] = element;

this.storedElements++;

}

}  
public void dequeue(){

if(this.isEmpty()){

return null;

} else{

E elem = this.data[this.frontElement];

this.frontElement = (this.frontElement + 1) %   
 this.capactiy;

this.storedElements--;

return elem;

}}

Graphical user interface

Description automatically generated with medium confidencePriority Queue

Grundprinzip: Einfügen von Werten mit Priority k, Liste ist nach k sortiert.   
Anwendung: Dijkstra-Algorithmus, Datenkompression in Huffman Code

* E insert(K k,V v): Fügt Eintrag mit Schlüssel k und Wert v ein
* E removeMin(): Entfernt Eintrag mit kleinstem Schlüssel und gibt ihn zurück
* E min(): Liefert Eintrag mit kleinstem Schlüssel ohne ihn zu entfernen
* int size(): Anzahl Elemente in Queue
* boolean isEmpty(): Sind Elemente in der Queue?

|  |  |  |
| --- | --- | --- |
| Methode | Unsorted List | Sorted List |
| Size |  |  |
| isEmpty |  |  |
| Insert |  |  |
| Min |  |  |
| removeMin |  |  |

**Adaptable Priority Queue ADT**:  
Gleiches Interface wie Priority Queue, aber mehr Methoden

* E remove(E e): Entfernt Eintrag
* boolean replaceKey(E e, K k): Schlüssel des Eintrags ersetzen
* boolean replaceValue(E e, V v): Wert des Eintrags ersetzen

//Insert **unsortierte** Liste:  
public Entry<K,V> insert (K key, V value) {  
 Entry<K,V> newest= new PriorityQueueEntry<>(key, value);  
 list.add(newest);  
 return newest;  
}

//RemoveMin **unsortierte** Liste:  
public Entry<K,V> removeMin() {  
 if (list.isEmpty()) {return null;}  
 var entry = findMin();  
 list.remove(entry);  
 return entry;  
}

//Min **unsortierte** Liste:  
public Entry<K,V> min() {  
 if (list.isEmpty()) {return null;}  
 return findMin();  
}  
private Entry<K,V> findMin() {  
 Entry<K,V> small = list.get(0);  
 for (Entry<K,V> walk : list) {  
 if (compare(walk, small) < 0) {small = walk;}  
 }return small;  
}

//Insert **sortierte** Liste:  
public Entry<K, V> insert(K key, V value) {  
 var newest = new PriorityQueueEntry<>(key, value);  
 if (list.size() == 0) {  
 list.add(newest);  
 return newest;  
 }  
 Entry<K,V> walk = list.get(list.size() - 1);  
 int index = 0;  
 for (index = list.size() - 1; index >= 0 &&   
 compare(newest, walk) > 0; index--) {   
 walk = list.get(index);  
 }  
 if (index == -1) {list.add(newest);}   
 else {list.add(index, newest);}  
 return newest;  
}

//RemoveMin **sortierte** Liste:  
public Entry<K, V> removeMin() {return list.remove(0);}

//Min **sortierte** Liste:  
public Entry<K, V> min() {return list.get(0);}  
public int size() {return list.size();}  
public boolean isEmpty() {return list.isEmpty();}

Trees / Bäume

Ein Bild, das Diagramm, Text, Reihe, Screenshot enthält.

Automatisch generierte BeschreibungZweidimensionale Datenstruktur, welche eine hierarchische Beziehung darstellt.

* Wurzel: Knoten ohne Elternknoten
* Innerer Knoten: Knoten mit min. einem Kind
* Blatt: Knoten ohne Kinder
* Vorgängerknoten: Eltern, Grosseltern, …
* Geschwister: Knoten mit selben Eltern
* Tiefe eines Knotens: Anzahl Vorgänger (Start bei 0)
* Höhe des Baums: Maixmale Tiefe der Knoten
* Subtree (Unterbaum): Baum aus einem Knoten und seinen Nachfolgern

import java.util.Iterator;  
public interface Tree<E> extends Iterable<E> {  
 Node<E> root();  
 Node<E> parent(Node<E> p);  
 Iterable<Node<E>> children(Node<E> p);£  
 Int numChildren(Node<E> p);  
 boolean isInternal(Node<E> p);  
 boolean isExternal(Node<E> p);  
 boolean isRoot(Node<E> p);  
}

Ein Bild, das Text, Screenshot, Reihe, Schrift enthält.

Automatisch generierte BeschreibungTiefe berechnen

* ist Wurzel des Baums: Tiefe von ist 0
* nicht Wurzel des Baums: 1 + Tiefe des Eltern-Knotens

public int depth(Node<E> p) {  
 if(isRoot(p)) {return 0;}   
 else {return 1 + depth(parent(p));}  
}

Höhe berechnen

public int height(Node<E> p) {  
 int h = 0;  
 for (Node<E> c : children(p)) {  
 h = Math.max(h, 1 + height(c));  
 } return h;  
}

Ein Bild, das Text, Diagramm, Screenshot, Reihe enthält.

Automatisch generierte BeschreibungBinärer Baum

Spezialform vom Baum. Jeder Knoten hat höchstens zwei Kinder. Kinder eines Knotens sind ein geordnetes Paar (links, rechts). Jeder Knoten ist ein Objekt mit Element, Elternknoten und Folge von Kindknoten.

A picture containing clock, watch

Description automatically generatedpublic interface BinaryTree<E> extends Tree<E> {  
 Node<E> left(Node<E> p);  
 Node<E> right(Node<E> p);  
 Node<E> sibling(Node<E> p);  
 Node<E> addRoot(E e);  
 Node<E> addLeft(Node<E> p, E e);  
 Node<E> addRight(Node<E> p, E e);  
}

Traversierung

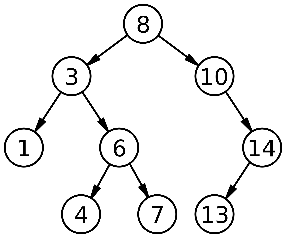
|  |  |  |
| --- | --- | --- |
| Methode | Beschreibung | Reihenfolge |
| Pre-Order (W-L-R) | Knoten wird **vor** seinen Kindern besucht | D,B,A,C,E |
| Post-Order (L-R-W) | Knoten wird **nach** seinen Kindern besucht | A,C,B,E,D |
| In-Order (L-W-R) | Knoten nach linkem Subtree und vor rechtem Subtree besuchen | A,B,C,D,E |
| Breadth-First | Alle **Knoten einer Stufe** besuchen, bevor Nachfolgeknoten besucht werden | D,B,E,A,C |

|  |  |
| --- | --- |
| algorithm **preOrder**(v)  visit(v)  for each child w of v  preOrder(w) | algorithm **postOrder**(v)  for each child w of v  postOrder(w)  visit(v) |

algorithm **breadthFirst**()  
//Initialize queue + containing root  
while Q not empty  
 v = Q.dequeue()

visit(v)  
 for each child w in children(v)  
 Q.enqueue(w)

algorithm **inOrder**(v)  
 if hasLeft(v)  
 inOrder(left(v))  
 visit(v)  
 if hasRight(v)  
 inOrder(right(v))

Binärsuche /   
Binary Search Tree (BST)

Binärer Baum muss vollständig gefüllt sein (ausser evtl. letztes Element). Jeder Knoten muss grösser sein als alle Knoten im linken Teilbaum und kleiner als die Knoten im rechten Teilbaum.

Anzahl Vergleiche bei Knoten:

int binarySearch(int array[], int x, int low, int high) {  
 if (high >= low) {  
 int mid = low + (high - low) / 2;  
 if (array[mid] == x) *// Lösung gefunden* return mid;  
 if (array[mid] > x) *//linke Hälfte suchen* return binarySearch(array, x, low,   
 mid - 1);  
 return binarySearch(array, x, mid + 1,  
 high);   
 return -1;  
}

Hash Map

Grundprinzip: Menge von Schlüssel-Wert Paaren von denen jeder Schlüssel unique ist.   
Hashing: Hash-Funktion h bildet Schlüssel auf Indexwerte im Intervall [0,n-1] ab und bestimmt für Elemente e die Position h(e) im Feld. Ziel ist, Werte mit lesen zu können.  
Anomalien: Leerstellen sind Platzverschwendung. Ein Hash-Code für mehrere Einträge ergeben Kollisionen  
Doppeltes Hashing: Zweite Hashfunktion d(x), falls Kollision entstanden ist. , Anzahl Kollisionen, Tabellengrösse (oft Primzahl)  
Erweiterbares Hashing: Vereinfacht vergrössern der Hashtabelle ohne Reallokation aller Werte. Verwendet erste Bit, binäres Ergebnis Hashfunktion . Überlauf + 1 Bit  
  
Eigenschaften guter Hashfunktion: Konsistenz (Trotz wiederholtem Aufruf gleichen Hashcode), Effiziente Berechnung, Gleichmässige Verteilung der Schlüssel (geringe Anomalien)  
Wichtig: Wenn equals() überschrieben wird muss im Normalfalls hashcode() auch überschrieben werden, sonst ergeben sich beim Einsatz von Maps Fehler  
Erweiterung: 2 Teilfunktionen für Hashing verwenden. Hash-Funktion soll Schlüssel möglichst zufällig verteilen. Kompressionsfunktion soll Schlüssel in fixes Intervall transformieren  
Beispiel Kompressionsfunktion:   
Gleichheit: Referenzvergleich equals() standardmässig, Inhaltsvergleich equals() überschrieben (nur bei Strings implementiert)  
Kollision: Mehrere Schlüssel werden auf einen Behälter abgebildet  
Überlauf: Datensatz mit Schlüssel heisst Überläufer, wernn durch Hashfunktion zugewiesener Behälter schon voll ist  
Offene Adressierung: Wenn Behälter voll ist, wird bei einer Kollision Platz im nächsten Behälter der Hashtabelle gesucht.  
Geschlossene Adressierung: Behälter sind (verkettete) Listen, Platz nicht begrenzt, prinzipiell keine Überläufer. Wie lange dauert die Suche nach einem Eintrag? Einträge pro Bucket im Schnitt (Lastfaktor): , Lineare Suche im Kollisionsbereich:   
Separate Chaining: Jede Zelle der Hashtabelle zeigt auf Liste, einfache Umsetzung dafür zusätzliche Datenstruktur und Speicherbedarf  
Offene Adressierung: Für Überläufer in anderen Behältern Platz suchen, Sondierungsfolge bestimmt Weg zum Speichern und Wiederauffinden der Überläufer  
Lineare Sondierung: Überläufer in nächste verfügbare Zelle einfügen, durchsuchen bis leere Zelle gefunden wurde

Mögliche Methoden zum Hashen:

* Divisionsrestverfahren:
* Integer Cast: Schlüssel als Integer interpretieren  
  byte[] b = key.getBytes(StandardCharsets.UTF\_16);  
  ByteBuffer wrapped = ByteBuffer.wrap(b);  
  return wrapped.getInt();
* Komponentensumme: Schlüssel in Komponenten fester Länge unterteilen, Komponenten summieren, Overflow ignorieren  
  int hash = 0;  
  for (int i = 0; i < s.length(); i++) {  
   hash += s.charAt(i);  
  }
* Polynom-Akkumulation: Komponenten bei Komponentensumme unterschiedlich gewichten (Gut für Strings)  
   int hash = 0;  
  for (int i = 0; i < s.length(); i++) {  
   hash += s.charAt(i) \* Math.pow(31,(s.length()-i+1);  
  }  
  return hash;

private V findSlot(int indexInHashTable, K key) {  
 int j = indexInHashTable;  
 int slot = -1;  
 do {  
 if (isAvailable(j)) {  
 slot = j; return slot;  
 } else if (table[j].getKey().equals(key)) {  
 slot = j; return slot;  
 }  
 j = (j+1) % capacity;  
 } while (j != indexInHashTable);  
 return slot;  
}

**Sondierung offene Adressierung**

//Lineare Sondierung  
private int findAvailableSlot(int indexInHashTable,K key){

int probedIndex = indexInHashTable;

do {

if(isAvailable(probedIndex)) {

return probedIndex;

}

else if(table[probedIndex].getKey().equals(key)) {

return probedIndex;

}

probedIndex = this.probingFunction.apply(probedIndex)  
 % capacity;

} while (probedIndex != indexHashTable); return -1;

}

//Lineare Sondierung

@Override

public V get(K key) {

int hashIndex = Math.abs(key.hashCode() % capacity);

int availableSlot = findAvailableSlot(hashIndex, key);

if(availableSlot == -1) {

return null;

}

return table[availableSlot].getValue();

}

@Override

public V remove(K key) {

int hashIndex = Math.abs(key.hashCode() % capacity);

int indexInHashMap = probeForDeletion(hashIndex, key);

if(indexInHashMap == -1){return null;}

V answer = table[indexInHashMap].getValue();

table[indexInHashMap] = DELETED;

return answer;

}

**Dynamisches Hashing**

private int **getPosition**(Object o, int d) {

BitSet bits = hashValueToBitSet(o);

int pos = 0;

for (int i = 0; i < d; i++) {

pos \*= 2;

if(bits.get(i)) {

pos++;

}

}

return pos;

}

private void **extenIndex**() {

int newSize = 1 << globalDepth;

Block newIndex[] = new Block[newSize \* 2);

for(int i = 0; i newSize; i++) {

newIndex[2 \* i] = hashIndex[i];

newIndex[2 \* i + 1] = hashIndex[i]

}

hashIndex = newIndex;

globalDepth++;

}

public void **add**(Object o) {

int index = getPosition(o,a globalDepth);

Block block = hashIndex[index];

if(block.elements().contains(o)) {

return;

}

while(block.elements().size() == MAXSIZE){

if(block.getDepth() == globalDepth) {

extendIndex();

index = getPosition(o, globalDepth);

}

splitBlock(index);

block = hashIndex[index];

}

block.elements().add(o);

}

Set

Sammlung gleichartiger Objekte, erlaubt keine Duplikate. Elemente haben keine Reihenfolge.

* add(e): Fügt e hinzu, falls noch nicht vorhanden
* remove(e): Entfernt e, falls vorhanden
* contains(e): Prüft, ob das Set e enthält
* size(): Gibt die Anzahl an Elementen zurück
* isEmpty(): Gibt zurück, ob das Set leer ist

|  |  |
| --- | --- |
| @Override public boolean add(E e) {  if (list.contains(e)) {  return false;  } else {  list.add(e);  return true;  }} | @Override public boolean contains(Object o) {  return list.contains(o); } |

Multiset

Set mit erlaubten Duplikaten.

* get(e): Fügt ein Exemplar von e hinzu
* remove(e): Ein Exemplar von e entfernen
* remove(e,n): n Exemplare von e entfernen
* contains(e): Prüft, ob das Set mindestens ein e enthält
* count(e): Gibt die Anzahl an Exemplaren von e zurück

Map

Ein Bild, das Screenshot, Reihe, Diagramm, Rechteck enthält.

Automatisch generierte BeschreibungSpeichert Schlüssel-Wert Paare. Schlüssel sind einzigartig, je Schlüssel genau ein Wert.

* get(k): Wert mit Schlüssel k zurückgeben, sonst null
* put(k,v): Entry(k,v) hinzufügen und null zurückgeben; Sonst vorhandenen Wert durch v ersetzen und alten Wert zurückgeben
* remove(k): Eintrag zum Schlüssel k entfernen und Wert zurückgeben; wenn kein Eintrag vorhanden, null zurückgeben
* size(): Gibt die Anzahl an Elementen zurück
* isEmpty(): Gibt zurück, ob das Set leer ist
* keySet(): Iterierbare Collection mit allen Schlüsseln
* values(): Iterierbare Collection mit allen Werten
* entrySet(): Iterierbare Collection mit allen Einträgen

Implementierung mit unsortierter Liste

public v **get**(K key, V value) {  
 var iter = this.list.iterator();  
 while (iter.hasNext()) {  
 var node = iter.next();  
 if (node.getKey().equals(key)) {  
 V t = node.getValue();  
 node.setValue(value);  
 return t;}  
 }  
 this.list.addLast(new ListNode<K,V>(key, value));  
 return null;  
}

public V **put**(K key, V value) {

var iter = this.list.iterator();

while(iter.hasNext()) {

var node iter.next();

if(node.getKey().equals(key)) {

V t = node.getValue();

node.setValue(value);

return t;}

}

this.list.addLast(new ListNode<K,V>(key, value));

return null;

}

public V **remove**(K key) {

var iter = this.list.iterator();

while(iter.hasNext()) {

var node = iter.next();

if(node.getKey().equals(key)) {

V val = node.getValue();

list.remove(node);

return val;}

}

return null;

}

Laufzeit

|  |  |  |
| --- | --- | --- |
|  | Schlechtester Fall | Bester Fall |
| put() |  |  |
| get() |  |  |
| remove() |  |  |

Ein Bild, das Screenshot, Diagramm, Reihe, Design enthält.

Automatisch generierte BeschreibungMultimap

Entspricht einer Map, wobei es zu einem Key mehrere Values geben kann.

* get(k): Gibt Collection aller Werte zurück, die mit e verknüpft sind.
* put(k,v): Fügt neuen Eintrag hinzu, zusätzliches Mapping von k auf v
* remove(k,v): Mapping zwischen k und v entfernen, falls eines existiert
* removeAll(k): Alle Einträge mit dem Key k entfernen

## Design Patterns

Motivation: Wiederverwendbare Lösungen für wiederkehrende Probleme verwenden. Lösung ist abstrakt und in neuen Kontexten anwendbar.

Elemente

* Mustername: Benennung des Patterns
* Problem: Beschreibung, wann das Pattern anzuwenden ist
* Lösung: Elemente aus denen Pattern besteht
* Konsequenzen: Vor- und Nachteile bei Anwendung des Patterns

Erzeugungsmuster: Abstrahieren Instanziierung (Factory, Singleton, …)

Strukturmuster: Zusammensetzung von Klassen & Objekten zu grösseren Strukturen (Adapter, Fassade, …)

Verhaltensmuster: Algorithmen und Verteilung von Verwantwortung zwischen Objekten (Iterator, Visitor, …)

Iterator

Auf Elemente eines Aggregatsobjekt sequenziell zugreifen, ohne zugrunde liegende Repräsentation offenzulegen.

Public interface Iterator<E> {  
 boolean hasNext();  
 E next() throws NoSuchElementException;  
 void remove() throws UnsupportedOperationException,  
 IllegalStateException;  
}  
while (iter.hasNext()) {  
 string value = iter.next();  
 system.out.println(value);  
} for (Element v : collection) {}

**For mit Iterator**

for (Iterator<String> i = stringList.iterator(); i.hasNext();) {String s = i.next();System.*out*.println(s);}

Lazy Iterator: Iteration auf originaler Datenstruktur. Niedrigere Speicherkosten . Änderungen der Datenstruktur verunmöglichen jedoch Iteration. Fehl-Verhalten bei unerwarteten Modifikationen der Ausgangs-Datenstruktur.

Table

Description automatically generated

Snapshot-Iterator: Original-Datenstruktur beibehalten. Kopie der Ausgangs-Datenstruktur erzeugen, Änderungen auf Ausgangs-Datenstruktur beinflussen Iteration nicht. Speicher und Laufzeit.

public SArrLiIter(T[] elem,int size,Comparator<T> comp) {  
 this.elements = copy(elements, size);  
 Arrays.sort(this.elements, comp);  
 this.comparator = comp;  
}

Visitor

Trennung von Algorithmen und Datenstrukturen, auf denen sie operieren. Ziel ist es, Algorithmen nicht über Datenstrukturen zu verteilen.   
  
Anwendung:

* Beschreibt Operationen auf Elementen einer anderen Objektstruktur
* Neue Operation definieren, ohne Klassen auf denen operiert wird zu ändern
* Klassen nicht mit zusätzlichen Operationen verkomplizieren
* Zusammenhängende Operationen in einer Klasse zusammenfassen

Adapter (auch Wrapper)

Strukturmuster: Fungiert als Verbindungsstück zwischen zwei inkompatiblen Schnittstellen, die sonst nicht direkt verbunden werden können. Das Hauptziel dieses Musters besteht darin, eine vorhandene Schnittstelle in eine andere umzuwandeln, die der Client erwartet.

|  |  |
| --- | --- |
| Objekt-Adapter:  class Adapter extends X {  private Y yObj;  public void a() {  yObj.b();  }  } | Klassen-Adapter:  class Adapter extends Y implements X {  public void a() {  b();  }  } |

Template-Method

Backbone eines Algorithmus definieren, Teilschritte später in Subklassen spezifizieren. Lässt Subklassen Teile des Algorithmus verfeinern, ohne Struktur des Algorithmus zu verändern. Gemeinsame, unveränderliche Teile werden in abstrakter Klasse implementiert (Template). Variable Schritte werden in Methoden ausgelagert. Anwendung: Frameworks