Input/Output streams:

File-Input bytes

```
var in = new FileInputStream("myFile.data");
int value = in.read();
while (value >= 0) {
  byte b = (byte)value;
  // work with b
  value = in.read();
}
in.close();
```

Bestehende Datei zum Lesen öffnen

-1: end of file

Gelesenes Byte (wenn positiv)

```
try (var in = new FileInputStream("...")) {
```

with try the in.close() is implicit.

File-Output

```
var out = new FileOutputStream("test.data");
while (…) {
  byte b = …;
  out.write(b);
}
out.close();
```

Datei neu anlegen bzw. überschreiben

Schreiben («Flush») des Rests beim Schliessen

```
int read(byte[] b, int offset, int length)
void write(byte[] b, int offset, int length)
void flush()
```

An Datei anhängen, falls existiert

```
Files.write(Path.of("out.bin"), data);
```

text read/write

```
try (var reader = new FileReader("quotes.txt")) {
  try (var writer = new FileWriter("test.txt", true))
    String line = reader.readLine();
Stream<String> stream = Files.lines(
  Path.of("in.txt"), StandardCharsets.UTF_8);
Files.write(Path.of("out.txt"), lines, StandardCharsets.UTF_8);
List<String> lines = Files.readAllLines(
    Path.of("in.txt"), StandardCharsets.UTF_8);
  private transient Image thumbnailImage;
    byte[] data = Files.readAllBytes(Path.of("in.bin"));
```

Append

read 1 line

read as stream

write entire file

read all lines

prevents Serializing of variable.

Serialize (turn to bytestream):

```
var person = new Person(…);

try (var stream = new ObjectOutputStream(
                  new FileOutputStream("serial.bin"))) {
  stream.writeObject(person);
}
```

Deserialize:

```
try (var stream = new ObjectInputStream(
                  new FileInputStream("serial.bin"))) {
Person p = (Person)stream.readObject();
…
}
```

Generics

left without type specification, right with type

```
List names = new ArrayList();
names.add("John");
names.add("Mary");
names.add(Boolean.FALSE); // Kein Fehler
```

```
List<String> names =
    new ArrayList<String>();
names.add("John");
names.add("Mary");
names.add(Boolean.FALSE); // Fehler
```

```
class Stack<T> {
  // …
}
```

Typ-Parameter

```
Stack<String>  stack1;
Stack<Integer> stack2;
```

Typ-Argument

```
public class StackTest {
  public <E> Stack<E> multiPush(E value, int times) {
```

Typ-Parameter

Typ-Argument

Iterators:

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

E für Element (Name beliebig)

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

Typ-Parameter

Typ-Argument

```
class Stack<T> implements Iterable<T> {
  private Entry<T> top;
  // …

  public Iterator<T> iterator() {
    return new StackIterator<T>(top);
  }
}
  for (String value : stringStack) {
    …
  }
```

```
class GraphicStack<T> extends Stack<T> {
  public void drawAll() {
    for (T item : this) {
      item.draw();
    }
  }
}
```

problem? every type that is entered needs the draw function

solution, extend the type to graphic

```
class GraphicStack<T extends Graphic> extends Stack<T> {
```

now the type has the default draw from graphic, as long as the type extends the graphic class

multi type extensions

```
class ClassName<T extends Type1 & Type2 & … >
```

only the first type can be a class, the rest are interfaces!!!! because java....

Covariance: check if type extends another

```
Stack<? extends Graphic> stack = new Stack<Rectangle>();
```

stack would have type graphic!

Contravariance: is it at least a cat ?

```
void bar(Comparator<? super Cat> catComparator) {
```

once again type would be cat.

a generic can't be static!! it is unclear what type it would have

```
public class MobileDevice<T> {
    private static T os;
    // ...
}
```

	Typ	read	write	compatible:
invariance	C<T>	✓	✓	T
Covariance	C<? extends T>	✓	✗	T and subtypes
Contravariance	C<? super T>	✗	✓	T and supertypes
Bivariance	C<?>	✗	✗	all

Java type erasure:
Essentially the types we define get removed on compiliation, reason is that not only does the machine not need this information, it would also make older java versions imcompatible, this is counterproductive as java works with vms. In order to preserve compatibility types are removed on compilation.

Algorithm Paradigms

Greedy-Algorithms

get as much as possible with every single step.

-- approximation of solution
-- faster than optimal solutions
-- usually finds at least 1 solution

the optimal solution here would be to steal a guitar and a laptop, but the greedy algorithm as the name says goes for the big item, aka the stereo.
It therefore loses 5lbs worth of items.

set covering problems:

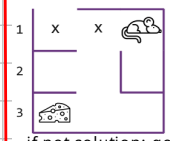
Instead we simply take the best one until no states are left, the overlap doesn't matter that much here.
We get a solution that is acceptable but not perfect, however we didn't wait until we are long dead.
-> 51 states, x amount of radio stations with different coverage what is the best arrangement of the radio stations?
not possible: -> O(2^n)!!

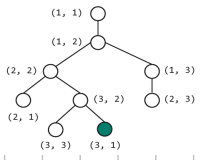
Recursion Backtracking

simple recursion: example factorial

```
static int recursiveFactorial(int n) {
  if (n == 0)
    return  1;
  else
    return n * recursiveFactorial(n-1);
}
```

recurse until solution found:





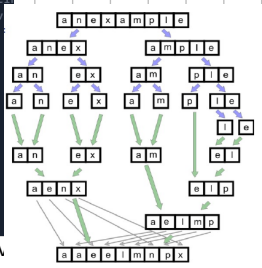
if not solution: go down,
if down not possible, go left,
if left not possible go right,

Recursion Divide and Conquer

mergesort:

```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge [] [] = []
merge (x:xs) (y:ys)
  | x <= y = [x] ++ merge xs (y:ys)
  | otherwise = [y] ++ merge ys (x:xs)


-- Exercise 6.8 (**)
-- Using merge, define a function msort
r the two lists that result from sorti
-- Hint: first define a function halv


halve :: [a] -> ([a], [a])
halve xs = (take len xs, drop len xs
  where len = div (length xs) 2

msort :: Ord a => [a] -> [a]
msort [] = []
msort (x:[]) = [x]
msort xs = merge (msort x) (msort y)
  where x = fst $ halve xs
        y = snd $ halve xs
```

halve halve halve halv



binary search also falls into this category!!!!

Dymanic Programming

0, 1, 1, 2, 3, 5, 8, 13, ...

Fibonacci:

```
static void fibonacciIterativ(int N) {
    int num1 = 0
    int num2 = 1;
    int counter = 0;

    while (counter < N) {
        System.out.print(num1 + " ");

        int num3 = num2 + num1;
        num1 = num2;
        num2 = num3;
        counter++;
    }
}
```

```
static int fibonacciDynamisch(int n) {
  int f[] = new int[n + 2];

  int i;
  f[0] = 0;
  f[1] = 1;

  for (i = 2; i <= n; i++) {
      f[i] = f[i - 1] + f[i - 2];
  }

  return f[n];
}
```

n	Iterativ	Dynamisch
5000	32.134.250	188.041
10000	42.381.917	313.125
20000	65.227.792	606.958

Big O Notation, comparison of Complexities:

$O(n)$ O = complexity, n = amount of operations.

big-O
$f(n)$ ist $O(g(n))$ falls $f(n)$ asymptotisch **kleiner oder gleich** $g(n)$ ist:
$f(n) \leq cg(n)$ für $n \geq n_0$

big-Omega
$f(n)$ ist $\Omega(g(n))$ falls $f(n)$ asymptotisch **grösser oder gleich** $g(n)$ ist:
$f(n) \geq c * g(n)$ für $n \geq n_0$

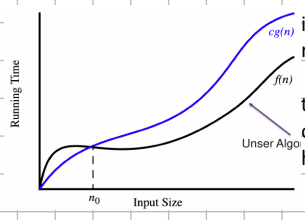
big-Theta
$f(n)$ ist $\Theta(g(n))$ falls $f(n)$ asymptotisch **gleich** $g(n)$ ist:
$c'g(n) \leq f(n) \leq c''g(n)$ für $n \geq n_0$

empirical analysis: compare algorithms:

show that algorithm is in linear complexity

$f(n) \leq cg(n)$ für $n \geq n_0$

c and n0 are variables.
it is only important that there is an n0 and c

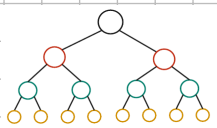


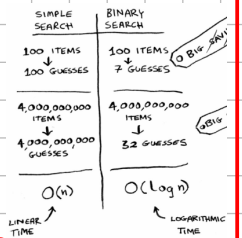
is n^2 also in linear time?
no, there is no c that would fulfill the requirements!!
check top right on how to fulfill them

- $2n + 10 \leq cn$
- $10 \leq cn - 2n$
- $10 \leq (c - 2)n$
- $\frac{10}{c-2} \leq n$
- $c = 3$ und $n_0 = 10$ ⟵

1. implement
2. test
3.compare results

binary





The linear search has a worst case of n.
aka last element.
the binary search divides the array, and thefore has a lower worst case!. in both algorithms, the best case is 1. aka first element

linear

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	4	5	7	8	9	12	14	17	19	22	25	27	28	33	37

Counting of Primitive Operations:

```
Algorithm arrayMax(A, n)                    # Operationen
  currentMax ← A[0]                         1 Indexierung + 1 Zuweisung:          2
  for i ← 1 to n − 1 do                     1 Zuweisung + n (Subtraktion + Test): 1 + 2n
    if A[i] > currentMax then               (Indexierungen + Test) (n − 1)        2(n − 1)
      currentMax ← A[i]                     (Indexierungen + Zuweisung) (n − 1)   0 | 2(n − 1)
    increment i                             (Inkrement + Zuweisung) (n − 1)       2(n − 1)
  return currentMax                         1 Verlassen der Methode               1
```

Worst Case: $2 + (1 + 2n) + 2(n - 1) + 2(n - 1) + 2(n - 1) + 1 = 8n - 2$

Best Case: $2 + (1 + 2n) + 2(n - 1) + 0 + 2(n - 1) + 1 = 6n$

Average Case?

Pseudocode

Methodendeklaration

```
Algorithm method (arg [, arg…])
  Input …    Output …
```

Methodenaufruf

```
var.method (arg [, arg…])
```

Rückgabewert

```
return expression
```

Kommentar

```
{ Ein Kommentar }
```

Kontrollfluss

- **if … then … [else …]**
- **while … do …**
- **repeat … until …**
- **for … do …**
- Einrücken, statt Klammern

```
Algorithm prefixAverages2(X, n)
  Input array X of n integers
  Output array A of prefix averages of X
  A ← new array of n integers                    ?
  s ← 0                                          ?
  for i ← 0 to n − 1 do                          ?
    s ← s + X[i]          { Summe der Elemente }  ?
    A[i] ← s / (i + 1)    { Durchschnittsbildung } ?
  return A                                       ?
```

Algorithmus *prefixAverages2* läuft in $O(n)$

	$f(n)$ ist $O(g(n))$	$g(n)$ ist $O(f(n))$
$g(n)$ wächst schneller	✓	✗
$f(n)$ wächst schneller	✗	✓
Selbes Wachstum	✓	✓

Note: in case of O(8n), we often ommit the 8 to describe that f(n) is in linear complexity, however this also restricts us from comparing linear complexities

usually we only care about the worst case scenario, as it shows the worst performance. However, depending on what we need/want we might consider median or best case as well

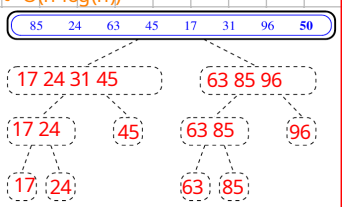
Sorting Algorithms:

merge sort java fast -> O(n*log(n))

```
int[] merge(int[] leftArray, int[] rightArray) {
        // …
        int targetPos = 0; int leftPos = 0; int rightPos = 0;
        while (leftPos < leftLen && rightPos < rightLen) {
                int leftValue = leftArray[leftPos];
                int rightValue = rightArray[rightPos];
                if (leftValue <= rightValue) {
                        target[targetPos++] = leftValue;
                        leftPos++;
                } else {
                        target[targetPos++] = rightValue;
                        rightPos++;
                }
        }
        while (leftPos < leftLen) {
                target[targetPos++] = leftArray[leftPos++];
        }
        while (rightPos < rightLen) {
                target[targetPos++] = rightArray[rightPos++];
        }
        return target;
}
private int[] mergeSort(int[] elements, int left, int right) {
        if (left == right)
                return new int[]{elements[left]};

        int middle = left + (right - left) / 2;
        int[] leftArray = mergeSort(elements, left, middle);
        int[] rightArray = mergeSort(elements, middle + 1, right);
        return merge(leftArray, rightArray);
}
```

Mergesort	Quicksort
Teilen in Listen gleicher Grösse ($^n/_2$)	Grösse der Listen abhängig vom Pivotelement
Immer $O(\log n)$	Im Worstcase $O(n^2)$
Zusätzlicher Speicherbedarf beim Merge	Kein zusätzlicher Speicherbedarf

quicksort fast O(n log(n))




```
static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

the partition is the pivot! You need a special function for that!

Stringbuilder vs string copy

```
String repeat1(char c, int n) {
  String answer = "";
  for (int j = 0; j < n; j++)
    answer += c;
  return answer;
}
```

```
String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder();
  for (int j = 0; j < n; j++)
    sb.append(c);
  return sb.toString();
}
```

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135

The difference is simply the speed.
All the int operations, alongside printline every single iteration is costly!!!
the second the calculation first then prints it!
It therefore keeps the results and bases the next result on that!

Recursion

recursive

```
public static char arrayMaximum(char[] arr, int low, int high) {
  if(low == high)
    return arr[low];
  int mid = (low + high) / 2;
  char c1 = arrayMaximum(arr, low, mid);
  char c2 = arrayMaximum(arr, mid + 1, high);
  if(c1 < c2)
    return c2;
  else
    return c1;
}
```

iterative

```
public static void arrayMax(char[] charArray) {
    private static int n = charArray.length;
    char max = a[0];
    for (int i = 1; i < n; i++) {
        if (max < a[i])
            max = a[i];
    }
    System.out.println("Max: " + max);
}
```

end-recursion

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

```
tailrecsum(5, 0)
tailrecsum(4, 5)
tailrecsum(3, 9)
tailrecsum(2, 12)
tailrecsum(1, 14)
tailrecsum(0, 15)
15
```

here the recursion call is the only statement in the return

```
private static int recsum(int x) {
    if (x == 0) {
        return 0;
    }
    else {
        return x + recsum(x - 1);
    }
}
```

```
recsum(5)
5 + recsum(4)
5 + (4 + recsum(3))
5 + (4 + (3 + recsum(2)))
5 + (4 + (3 + (2 + recsum(1))))
5 + (4 + (3 + (2 + (1 + recsum(0)))))
5 + (4 + (3 + (2 + (1 + 0))))
5 + (4 + (3 + (2 + 1)))
5 + (4 + (3 + 3))
5 + (4 + 6)
5 + 10
15
```

here the return statement is only part of the return.

- recursions with end-recursion can easily be made into an iterative form.
- recursive implemtations usually need more resources!

```
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;
}
```

```
static int[] iterativReverseArray(
  int[] a, int i, int j) {
  while (i < j) {
    int temp;
    temp = a[j];
    a[j] = a[i];
    a[i] = temp;
    i = i + 1;
    j = j - 1;
  }
  return a;
}
```

recursive to iterative : reverse array

Bogosort: meme

sort randomly until we get a solution……
as you might expect, this algorithm might run forever

```
void bogo(int[] arr) {
  int shuffle = 1;
  for(; !isSorted(arr); shuffle++)
    shuffle(arr);
}

void shuffle(int[] arr) {
  int i = arr.length-1;
  while(i > 0)
    swap(arr, i--, (int)(Math.random() * i));
}
```

Selection Sort slow

Band	Play Count
Linkin Park	20
Afrob	141
Meshuggah	156
Sido	**50**
Papa Roach	97
In Flames	48
Wurzel 5	111

Band	Play Count
Linkin Park	20
Sido	**50**
Afrob	141
Meshuggah	156

sort based on attribute, here play count, ascending

-- easy to implement
-- complexity independent from input
-- not much changes in the array
problems? yeah 1:

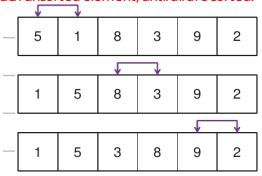
F
O(n^2

```
static void selectionSort(int[] array) {
  int marker = array.length - 1;
  while (marker >= 0) {
    int max = 0;
    for (int i = 1; i <= marker; i++) {
      if (array[i] > array[max]) {
        max = i;
      }
    }
    swap(array, marker, max);
    marker--;
  }
}
```

BubbleSort slow

iterate through the entire array multiple times and swap each unsorted element, until all are sorted.

```
bubbleSort(Array a)
    for (n = a.size; n > 1; n--) {
        for (i = 0; i < n - 1; i++) {
            if (a[i] > a[i + 1]) {
                a.swap(i, i + 1)
            }
        }
    }
```



$$(n-1)+(n-2)+(n-3)+(n-4)+\cdots+1=\sum_{i=1}^{n-1} i=\frac{n(n-1)}{2}$$

Insertion Sort slow

Band	Play Count
Linkin Park	20
Afrob	141
Meshuggah	156
Sido	50
Papa Roach	97
In Flames	48
Wurzel 5	111

```
public static void sort(Comparable[] a) {
  int n = a.length;
  for (int i = 1; i < n; i++) {
    for (int j = i; j > 0 && less(a[j], a[j - 1]); j--) {
      exchange(a, j, j - 1);
    }
  }
}
```

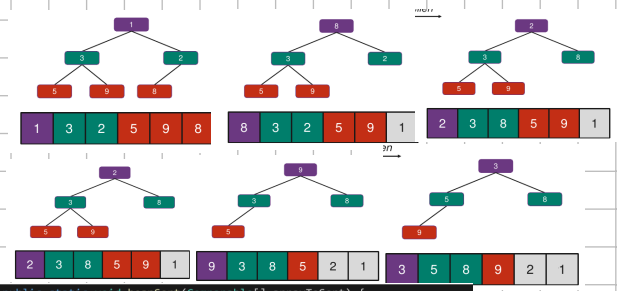
Band	Play Count
Meshuggah	156
Afrob	141
Wurzel 5	111
Papa Roach	97
Sido	50
In Flames	48
Linkin Park	20

$$O(n+(n-1)+\cdots+2+1)=O\left(\sum_{i=1}^{n} i\right)=\frac{n(n+1)}{2}=\frac{n^2+n}{2}\sim O(n^2)$$

Complexity O(n^2) just like selection sort

Heapsort:

prerequisite: child node keys are smaller than parent keys



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

```
public static void percolate(Comparable[] arrayToSort, int startIndex, int last) {
    int i = startIndex;
    while(hasLeftChild(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;
    }
}
```

Hash Codes:

Hash-Codes: Integer Cast

- Schlüssel als Integer interpretieren.
  - Gute Wahl, solange Anzahl Bits Interpretation als Integer erlaubt.
- Floating Point: $[0 \ldots 1] * m - 1$

Hash-Codes: Komponentensumme

- Bits des Schlüssels in Komponenten fixer Länge (16 oder 32 Bits) unterteilen
- Komponenten summieren und Overflows ignorieren
- Gut für Schlüssel fixer Länge, grösser/gleich Anzahl Bits von Integer

```
int hash = 0;
for (int i = 0; i < s.length(); i++) {
    hash = (R * hash + s.charAt(i)) % m;
}
```

Polynom-Akkumulation (1)

- Hashing von Werten der Form $(x_0, x_1, \ldots, x_{n-1})$ schwierig durch Addition

```
int hash = (((day * R + month) % m) * R + year) % m;
```

- Polynom
  - $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$
  - für fixes x

Gut für Strings

- Mit $z = 33$ maximal 6 Kollisionen bei 50.000 englischen Wörtern

```
String.hashCode()
```

Gibt Hash-Code für Zeichenfolge zurück.

```
s[0] * 31 ^ (n - 1) + s[1] * 31 ^ (n - 2) + ... + s[n - 1]
```

unter Verwendung von int-Arithmetik, wobei `s[i]` das i-te Zeichen der Zeichenkette, n die Länge der Zeichenkette und ^ die Potenzierung ist.

Eigenschaften guter Hashfunktionen

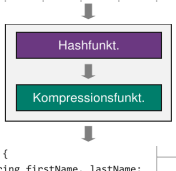
- Konsistenz
- Effiziente Berechnung
- Gleichmässige Verteilung der Schlüssel

low collisions!

Zuerst **Hashfunktion** anwenden, dann **Kompressionsfunktion** auf Ergebnis der **Hashfunktion**:

$$h(x) = h_2(h_1(x))$$

- **Hashfunktion:** Schlüssel möglichst zufällig zu verteilen
- **Kompressionsfunktion:** Schlüssel in fixes Intervall transformieren



Referenzvergleich: `equals()` standardmässig Referenzvergleich (*referential equality*)
- `a.equals(b)` vs. `a == b`

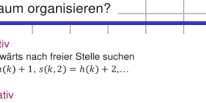
Inhaltsvergleich: `equals()` überschreiben (*structural equality*)
- Bei String implementiert (bei anderen Datentypen nicht!)

```
class Person {
  private String firstName, lastName;

  @Override
  public boolean equals(Object obj) {
    // Implementation
  }
```

Geschlossene Adressierung

- Behälter sind (verkettete) Listen
- Platz nicht begrenzt, Prinzipiell keine Überläufer

Separate Chaining: Jede Zelle der Tabelle zeigt auf Liste



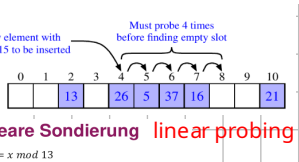
Gut: Einfach

Schlecht: Zusätzliche Datenstruktur und Speicherbedarf

- Einträge pro Bucket im Schnitt: $\frac{n}{N}$
- Lineare Suche im Bucket: $O(\frac{n}{N})$
- Überlaufbereich als Baum organisieren?

Offene Adressierung

- Für Überläufer in anderen Behältern Platz suchen
- Sondierungsfolge bestimmt Weg zum Speichern und Wiederauffinden der Überläufer.
  - Platz in anderem Behälter suchen
  - Linear probing: Überläufer in nächste verfügbare Zelle der Tabelle einfügen
  - Inspizierte Zellen werden als *probe* bezeichnet.

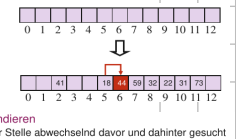


Lineare Sondierung linear probing

$h(x) = x \bmod 13$

Einfügen von 18, 41, 22, 44, 59, 32, 31, 73:

- $18 \bmod 13 = 5$
- $41 \bmod 13 = 2$
- $22 \bmod 13 = 9$
- $44 \bmod 13 = 5$
- $59 \bmod 13 = 7$



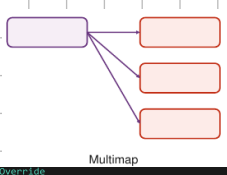
Alternierendes Sondieren
- Es wird nach freier Stelle abwechselnd davor und dahinter gesucht

Alternierendes quadratisches Sondieren
- Mit quadratischer Funktion nach freier Stelle weitersuchen
- $s(k,1) = h(k) - 1, s(k,2) = h(k) + 2^2, s(k,3) = h(k) - 3^2, \ldots$

```
private int findSlot(int startIndex, K key) {
  int available = -1;
  int j = startIndex;
  do {
    if (isAvailable(j)) {
      if (available == -1) {
        available = j;
      }
      if (table[j] == null) {
        break;
      }
    } else if (table[j].getKey().equals(key)) {
      return j;
    }
    j = (j + 1) % capacity;
  } while (j != startIndex); // Map nur einmal durchsuchen
  return -(available + 1);
}
```

```
protected V bucketGet(int h, K k) {
    int j = findSlot(h, k);
    if (j < 0)
        return null;
    return table[j].getValue();
}
```

- **Linear positiv**
  - Linear vorwärts nach freier Stelle suchen
  - $s(k,1) = h(k) + 1, s(k,2) = h(k) + 2, \ldots$
- **Linear negativ**
  - Linear rückwärts nach freier Stelle suchen
  - $s(k,1) = h(k) - 1, s(k,2) = h(k) - 2, \ldots$
- **Quadratisch**
  - Mit quadratischer Funktion nach freier Stelle suchen
  - $s(k,1) = h(k) + 1, s(k,2) = h(k) + 2^2, s(k,3) = h(k) + 3^2, \ldots$




```
package ch.ost.oop2.hashmultimap;

import java.util.Map;

public interface Multimap<K, V> {
    public Iterable<V> get (K key);
    public void put(K key, V value);
    public boolean remove(K key, V value);
    public Iterable<V> removeAll(K key);
    public Iterable<Map.Entry<K, V>> entries();
    public int size();
    public boolean isEmpty();
}
```

```
public class HashMultimap<K, V> implements Multimap<K, V> {
    Map <K, List<V>> map = new HashMap<>();

    int numberOfEntries = 0;
    public HashMultimap() {};

    @Override
    public int size() {
        return numberOfEntries;
    }

    @Override
    public boolean isEmpty() {
        return (numberOfEntries == 0);
    }
    // ...
}
```

```
@Override
public void put(K key, V value) {
    List<V> secondary = map.get(key);
    if (secondary == null) {
        secondary = new ArrayList<>();
        map.put(key, secondary);
    }
    secondary.add(value);
    numberOfEntries++;
}
```

```
@Override
public void put(K key, V value) {
    List<V> secondary = map.get(key);
    if (secondary == null) {
        secondary = new ArrayList<>();
        map.put(key, secondary);
    }
    secondary.add(value);
    numberOfEntries++;
}
```

```
@Override
public Iterable<V> get (K key) {
    List<V> secondary = map.get(key);
    if (secondary != null) {
        return secondary;
    }
    return new ArrayList<>();
}
```

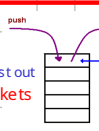
```
@Override
public boolean remove(K key, V value) {
    boolean wasRemoved = false;
    List<V> secondary = map.get(key);
    if (secondary != null) {
        wasRemoved = secondary.remove(value);
        if (wasRemoved) {
            numberOfEntries--;
            if(secondary.isEmpty()) {
                map.remove(key);
            }
        }
    }
    return wasRemoved;
}
```

- `get(k)`: Alle Werte zu Schlüssel *k* zurückgeben
- `put(k, v)`: Eintrag <k, v> hinzufügen, weitere Mappings werden beibehalten
- `remove(k, v)`: Entfernt Mapping von *k* auf *v*, falls eines existiert
- `removeAll(k)`: Entfernt alle Einträge mit Schlüssel *k* aus der Multimap
- `size()`: Anzahl an Einträgen zurückgeben
- `entries()`: Alle Einträge aus Multimap zurückgeben
- `keys()`: Alle Schlüssel zurückgeben (Duplikate für Keys mit mehreren Values)
- `keySet()`: Alle Schlüssel zurückgeben (ohne Duplikate)
- `values()`: Alle Werte für alle Einträge der Multimap zurückgeben

Stack

LIFO last in first out

Check brackets with stack:




```
public interface Stack<E> {
    int size();
    boolean isEmpty();
    void push(E element);
    E top();
    E pop();
}
```

```
private static boolean isMatched(String expression) {
    final String opening = "({[";
    final String closing = ")}]";
    Stack<Character> buffer = new Stack<>();

    for(char c : expression.toCharArray()) {
        if(opening.indexOf(c) != -1) {
            buffer.push(c);
        } else if(closing.indexOf(c) != -1) {
            if (buffer.isEmpty()) {
                return false;
            }
            if(closing.indexOf(c) != opening.indexOf(buffer.pop())) {
                return false;
            }
        }
    }
    return buffer.isEmpty();
}
```

Performance for stack:
O(n)
every operation needs O(1)

Array stack


```
public void push(Item item) {
    if(n == a.length) resize(2 * a.length);
    a[n++] = item;
}

private void resize(int max) {
    Item[] temp = (Item[]) new Object();
    for (int i = 0; i < n; i++) {
        temp[i] = a[i];
    }
    a = temp;
}
```

```
public class ArrayStack<E> implements Stack<E> {

    private E[] data;
    private int t = -1;

    public ArrayStack(int capacity){
        data = (E[]) new Object[capacity];
    }

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

    public boolean isEmpty(){
        return (t == -1);
    }

public void push(E element) throws IllegalStateException {
  if (size() == data.length)
    throw new IllegalStateException ("Stack is full!");
  data[++t] = element;
}

public E top() {
  if (isEmpty()) return null;
  return data[t];
}

public E pop() {
  if (isEmpty()) return null;
  E element = data[t];
  data[t--] = null;
  return element;
}
```

examples:
Stack, Lists, linked lists, arrays, Queues, etc.

Linked list is just a stack based on lists

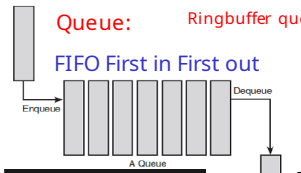
Stack	Singly-Linked-List
size()	list.size()
isEmpty()	list.isEmpty()
push(element)	list.addFirst(element)
pop()	list.removeFirst()
top()	list.first()

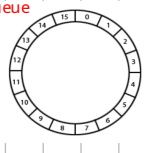
Abstract data types:
An abstract datatype (ADT) is a user defined type or class with functions etc. that can be applied to other types.
example array -> int array
List -> String List... etc

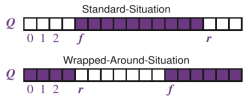
Queue:

FIFO First in First out

Ringbuffer queue







point to return element
size
point to write element

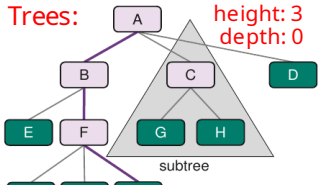
```
public interface Queue<E> {
    int size();
    boolean isEmpty();
    void enqueue(E element);
    E first();
    E dequeue();
}
```

enqueue(E) = add element at end of queue
<Object> dequeue() = remove element at start of queue and return it
<E> first() = returns first element without removing it
<integer> size() = returns how many elements there are in the queue
<boolean> isEmpty() = true if the queue has no elements

```
public void enqueue(E element) {
    if (this.storedElements == this.capacity) {
        throw new IllegalStateException();
    } else {
        int r = (this.frontElement + this.storedElements) % this.capacity;
        this.data[r] = element;
        this.storedElements++;
    }
}
```

```
public E dequeue() {
    if (this.isEmpty()) {
        return null;
    } else {
        E elem = this.data[this.frontElement];
        this.frontElement = (this.frontElement + 1) % this.capacity;
        this.storedElements--;
        return elem;
    }
}
```

Trees:



height: 3
depth: 0

depth: 3
heigth: 0

A,B,C,F = Nodes (at least one child node)
A = root Node (Node without parent nodes)
Leaf = Node without child nodes
siblings = Nodes at same level
parents, grandparents,etc = higher nodes.
depth of node = amount of predecessors
depth of tree = highest depth of node in tree
height = opposite of depth

Methoden
- Position root()
- Position parent(p)
- PositionList children(p)
- Integer numCildren(p)

Abfragemethoden
- Boolean isInternal(p)
- Boolean isExternal(p)
- Boolean isRoot(p)

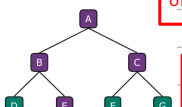
```
public interface Position<E> {
    /** Return the element stored at this position */
    E getElement();
}
```

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

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

Binary-Tree:

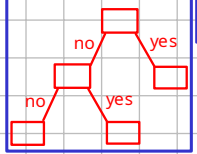
a Tree that only allows a maximum of 2 children for each node.



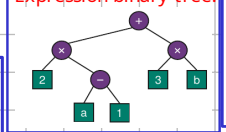
node

each of these can be empty.

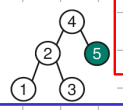
Decision Trees:



Expression binary tree:



Binary Search Tree (BST):



-- Each node has a key, aka value
-- keys of nodes to the left are smaller
-- keys of nodes to the right are bigger

```
public interface BinaryTree<E> extends Tree<E> {
    Position<E> left(Position<E> p);
    Position<E> right(Position<E> p);
    Position<E> sibling(Position<E> p);
    Position<E> addRoot(E e);
    Position<E> addLeft(Position<E> p, E e);
    Position<E> addRight(Position<E> p, E e);
```

in binary tree

```
public Position<E> sibling(Position<E> p)
        throws IllegalArgumentException {
    Position<E> parent = parent(p);
    if (parent == null) {
        return null;
    }
    if (p == left(parent)) {
        return right(parent);
    } else {
        return left(parent);
    }
}
```

in abstract bin tree

```
public int numChildren(Position<E> p)
        throws IllegalArgumentException {
    int count = 0;
    if (left(p) != null) {
        count++;
    }
    if (right(p) != null) {
        count++;
    }
    return count;
}
```

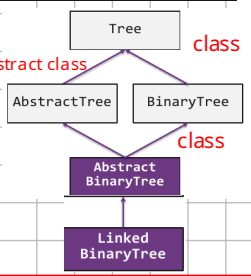
in abstract bin tree

```
public Iterable<Position<E>> children(Position<E> p)
        throws IllegalArgumentException {
    List<Position<E>> snapshot = new ArrayList<>(2);
    if (left(p) != null) {
        snapshot.add(left(p));
    }
    if (right(p) != null) {
        snapshot.add(right(p));
    }
    return snapshot;
}
```

in abstract bin tree

```
public Position<E> addLeft(Position<E> p, E e)
        throws IllegalArgumentException {
    Node<E> parent = null;
    try {
        parent = validate(p);
    } catch (IllegalAccessException e1) {
        e1.printStackTrace();
    }
    if (parent.getLeft() != null) {
        throw new IllegalArgumentException("p already has a left child");
    }
    Node<E> child = createNode(e, parent, null, null);
    parent.setLeft(child);
    size++;
    return child;
}
```

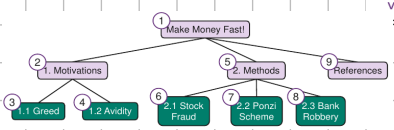
in linked binary tree



Traversal:

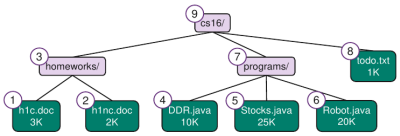
Preorder W-L-R : root,left,right

```
algorithm preOrder(v)
    visit(v)
    for each child w of v
        preOrder(w)
```



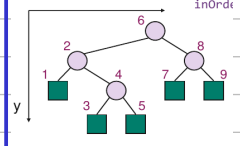
Postorder L-R-W: left,right,root

```
algorithm postOrder(v)
    for each child w of v
        Order(w)
    v)
```

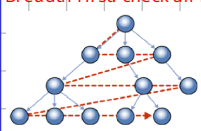


Inorder L-W-R left, root, right

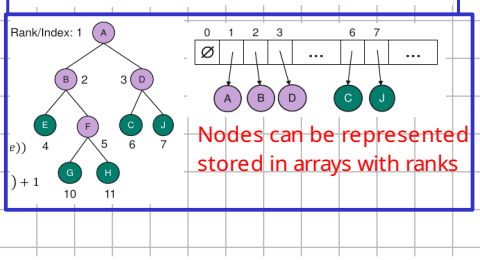
```
algorithm inOrder(v)
    if hasLeft(v)
        inOrder(left(v))
    visit(v)
    if hasRight(v)
        inOrder(right(v))
```



Breadth-First: check all nodes of same level first.



```
algorithm breadthFirst()
    Initialize queue Q containing root
    while Q not empty
        v = Q.dequeue()
        visit(v)
        for each child w in children(v)
            Q.enqueue(w)
```



Nodes can be represented stored in arrays with ranks

Iterator pattern:

```
while (iter.hasNext()) {
    string value = iter.next();
    System.out.println(value);
}
```

```
public interface Iterator<E> {
    boolean hasNext();
    E next() throws NoSuchElementException;
    void remove() throws UnsupportedOperationException, IllegalStateException;
}
```

Reverse Iterator

```
public ReverseArrayListIterator(ArrayList<T> arraylist) {
    this.arraylist = arraylist;
    currentIndex = arraylist.size() - 1;
}
```

Sorted Iterator

```
public SortedArrayListIterator(ArrayList<T> arraylist) {
    this.arraylist = arraylist;
    arraylist.sort();
}
```

```
public class IntArrayList {
  private class MyIterator implements Iterator<Integer> {/*...*/}
  private class MySortedIterator implements Iterator<Integer> {/*...*/}
  private class MyReverseIterator implements Iterator<Integer> {/*...*/}

  public Iterator<Integer> iterator() {
    return new MyIterator();
  }

  public Iterator<Integer> sortedIterator() {
    return new MySortedIterator();
  }

  public Iterator<Integer> reverseIterator() {
    return new MyReverseIterator();
  }
}
```

user defined iterator

Lazy Iterator: default iterator.

This iterator runs over the original datastructure. This is a problem however as changes on the original datastructure could lead to exceptions! ITERATOR INVALIDATION!!

Complexity = O(1)

Solution: Snapshot-Iterator

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

private T[] copy(T[] elements, int size) {
    T[] tmp = (T[]) new Object[size];
    System.arraycopy(elements, 0, tmp, 0, size);
    return tmp;
}
```

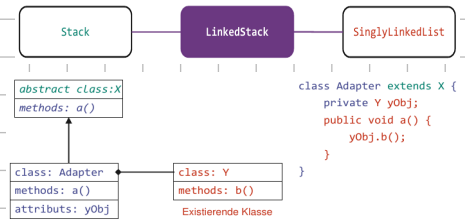
work with a copy of the original datastructure. This means we can now change the original datastructure without changing the one we iterate over. Iterator invalidation solved!!

Complexity = O(n)

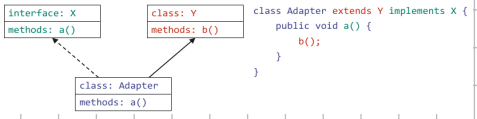
```
public class SortedSnapshotArrayListIterator<T> implements Iterator<T> {
    private int currentIndex;
    private final T[] elements;

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

    private T[] copy(T[] elements, int size) {
        T[] tmp = (T[]) new Object[size];
        System.arraycopy(elements, 0, tmp, 0, size);
        return tmp;
    }

    @Override
    public boolean hasNext() {
        return currentIndex < elements.length;
    }

    @Override
    public T next() {
        return elements[currentIndex++];
    }
}
```

Adapter Pattern:

change an existing class so that it can be used with a different class / interface



Modifier and Type	Method and Description
void	**add(E e)** Inserts the specified element into the list (optional operation).
boolean	**hasNext()** Returns true if this list iterator has more elements when traversing the list in the forward direction.
boolean	**hasPrevious()** Returns true if this list iterator has more elements when traversing the list in the reverse direction.
E	**next()** Returns the next element in the list and advances the cursor position.
int	**nextIndex()** Returns the index of the element that would be returned by a subsequent call to **next()**.
E	**previous()** Returns the previous element in the list and moves the cursor position backwards.
int	**previousIndex()** Returns the index of the element that would be returned by a subsequent call to **previous()**.
void	**remove()** Removes from the list the last element that was returned by **next()** or **previous()** (optional operation).
void	**set(E e)** Replaces the last element returned by **next()** or **previous()** with the specified element (optional operation).



here you use the LinkedStack as a linked list from a stack

```
public class LinkedStack<E> implements Stack<E>{
    private LinkedList<E> list = new LinkedList<>();
    public int size() { return list.size(); }
    public boolean isEmpty() { return list.isEmpty(); }
    public void push(E item) { list.addFirst(item); }
    public E top() { return list.getFirst(); }
    public E pop() { return list.remove(); }
}
```

```
public class LinkedStackInherit<E> extends LinkedList<E> implements Stack<E> {
    public int size() {
        return this.size();
    }
    public boolean isEmpty() {
        return this.isEmpty();
    }
    public void push(E item) {
        this.addFirst(item);
    }
    public E top() {
        return this.getFirst();
    }
    public E pop() {
        return this.remove();
    }
}
```

Here you use the linkedstackinherit for everything.

Template Pattern:

Write code so it is reusable, aka seperate the specific and the generic code.

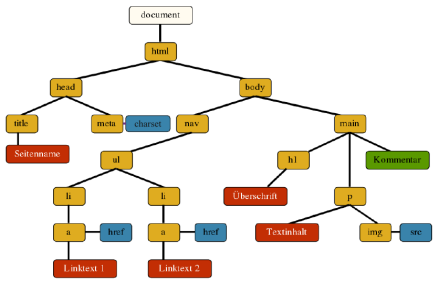
```
public abstract class EulerTour<E> {
    protected abstract void visitLeaf(Node<E> node);
    protected abstract void visitLeft(Node<E> node);
    protected abstract void visitBelow(Node<E> node);
    protected abstract void visitRight(Node<E> node);

    public void eulerPath(Node<E> node, Node<E> parent) {
        if (node == null) {
            return;
        }

        if (node.isLeaf()) {
            visitLeaf(node);
        } else {
            visitLeft(node);
            eulerPath(node.getLeft(), node);
            visitBelow(node);
            eulerPath(node.getRight(), node);
            visitRight(node);
        }
    }
}
```

```
public class ConcreteEulerTour<E> extends EulerTour<E> {
    @Override
    protected void visitLeft(Node<E> n) {
        System.out.println("Left: " + n.getValue());
    }

    @Override
    protected void visitBelow(Node<E> n) {
        System.out.println("Below: " + n.getValue());
    }
    @Override
    protected void visitRight(Node<E> n) {
        System.out.println("Right: " + n.getValue());
    }

    @Override
    protected void visitLeaf(Node<E> n) {
        System.out.println("Leaf: " + n.getValue());
    }
}
```

Example of templates with the EulerTour a generic form of traversing a binary tree.

Visitor Pattern:

similar functionality as the templates. Seperate the algorithm from a datastructure so when you add a new one, you don't have to make too many changes.




```
public interface TagVisitor {
    void visit(HtmlTag html);
    void visit(HeadTag head);
    void visit(BodyTag body);
    void visit(PTag p);
    void leave(HtmlTag html);
    void leave(HeadTag head);
    void leave(BodyTag body);
    void leave(PTag p);
}
```

```
public class TagTreeVisitor implements TagVisitor{
    @Override
    public void visit(HtmlTag html) {
        System.out.println("<html>");
    }

    @Override
    public void visit(HeadTag head) {
        System.out.println("<head>\n\t<title>\n\t\t" + head.getTitle());
    }

    @Override
    public void visit(BodyTag body) {
        System.out.println("<body>");
    }

    @Override
    public void visit(PTag p) {
        System.out.println("\t<p>\n\t\t" + p.getText());
    }
    // ...
}
```

```
public class HtmlTag implements Tag {
    private final HeadTag headTag;
    private final BodyTag bodyTag;

    public HtmlTag(HeadTag headTag, BodyTag bodyTag) {
        this.headTag = headTag;
        this.bodyTag = bodyTag;
    }

    public HeadTag getHeadTag() { return headTag; }
    public BodyTag getBodyTag() { return bodyTag; }

    @Override
    public void accept(TagVisitor visitor) {
        visitor.visit(this);
        headTag.accept(visitor);
        bodyTag.accept(visitor);
        visitor.leave(this);
    }
}
```

Priority Queue:

ADT: Methoden

k: Priorität

- insert(k, v) — Fügt Eintrag mit Schlüssel k und Wert v ein
- removeMin() — Entfernt Eintrag mit kleinstem Schlüssel und gibt ihn zurück
- min() — Liefert Eintrag mit kleinstem Schlüssel, ohne diesen zu entfernen
- size() — Anzahl Elemente in Queue
- isEmpty() — Sind Elemente in der Queue?

```
public interface PriorityQueue<K, V> {
    int size();
    boolean isEmpty();
    Entry<K, V> insert(K key, V value);
    Entry<K, V> min();
    Entry<K, V> removeMin();
}

public interface Entry<K,V> {
    public K key();
    public V value();
}
```

unsorted Queue:

```
@Override
public Entry<K,V> insert(K key, V value) {
    checkKey(key);
    Entry<K,V> newest = new PriorityQueueEntry<>(key, value);
    list.add(newest);
    return newest;
}
```

```
@Override
public Entry<K,V> removeMin() {
    if (list.isEmpty()) {
        return null;
    }
    var entry = findMin();
    list.remove(entry);
    return entry;
}

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;
}
```

Sorted:

```
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;
}
```

Methode	Unsorted List	Sorted List
size	$O(1)$	$O(1)$
isEmpty	$O(1)$	$O(1)$
insert	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
removeMin	$O(n)$	$O(1)$

```
@Override
public Entry<K, V> removeMin() {
    return list.remove(0);
}
```

Maps:

```
public V get(K key) {
    var iter = this.list.iterator();
    while (iter.hasNext()) {
        var node = iter.next();
        if (node.getKey().equals(key)) {
            return node.getValue();
        }
    }
    return null;
}
```

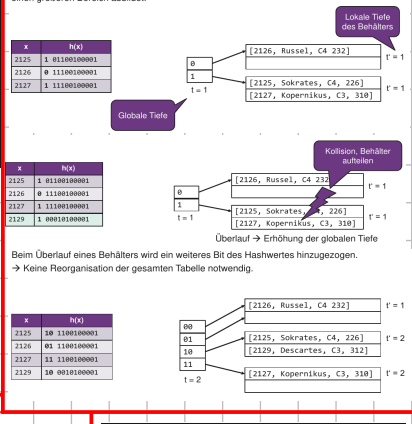
O(n)

```
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;
}
```

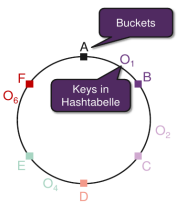
O(n)

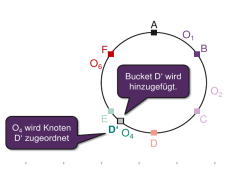
Erweiterbares Hashing (1)

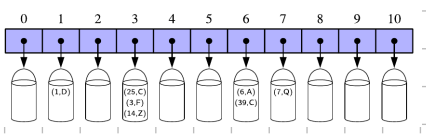
Beim *erweiterbaren Hashing* wird binäres Ergebnis der Hashfunktion $h(x)$ verwendet, die auf einen größeren Bereich abbildet.



Hashmaps:








```
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;
}
```

O(n)