Universität Trier

APW

Algorithmen für Programmierwettbewerbe

Vorlesung 2: Datenstrukturen in der Standardbibliothek

Philipp Kindermann

Sommersemester 2025

Überblick

IO

Lineare Datenstrukturen

Nicht-lineare Datenstrukturen

Tipps und Tricks

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

[xkcd.com/1185]

IO

C++	Java

`int64_t i,j;`
`cin >> i >> j;`
	`Scanner s = new Scanner(System.in);`
	`int i = s.nextInt();`
	`int j = FastIO.nextInt();`

`cout << i << endl;` `cout << j << "\n";`	`System.out.println(i);` `FastIO.out.println(j);`

- `Scanner` ist langsam
  - ➡ bei großen Eingaben `FastIO` verwenden
- `endl` ist langsamer als `"\n"`, weil es noch einen Flush verursacht, den man eigentlich nicht braucht (ist aber fast immer völlig egal). Äquivalent zu:
  `cout << i << "\n" << flush;`

Java FastIO

```
public class FastIO {
  static BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
  static StringTokenizer tokens = new StringTokenizer("");
  static PrintStream out = new PrintStream(new BufferedOutputStream(System.out,false));

  static boolean hasNext() throws IOException {
    while (!tokens.hasMoreTokens()) {
      String line = r.readLine();
      if (line == null) return false;
      tokens = new StringTokenizer(line);
    }
    return true;
  }

  static String next() throws IOException {
    hasNext();
    return tokens.nextToken();
  }

  static int nextInt() throws IOException {
    return Integer.parseInt(next());
  }
}
```

- `FastIO` muss am Ende geschlossen werden:

```
FastIO.out.close();
```

Überblick

IO

Lineare Datenstrukturen

Nicht-lineare Datenstrukturen

Tipps und Tricks

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

[xkcd.com/1185]

Statisches Array

- Liste einer festen Anzahl von Elementen
- Anzahl der Elemente kann während der Ausführung des Programms **nicht** geändert werden.
- Arrays sind: einfach, schnell, oft ausreichend
- Operationen: Indexzugriff

```
int64_t a[100];
a[2] = 10;
```

Achtung in C++

- Elemente uninitialisiert, es sei denn das Array ist global deklariert
- In Funktionen keine großen Arrays anlegen, sonst Stackoverflow
  (globale Arrays landen im statischen Speicherbereich, lokale Arrays auf dem Stack)
- Arraygröße muss eine (Übersetzungszeit-)Konstante sein!

Dynamisches Array

- Liste mit dynamisch änderbarer Anzahl von Elementen

C++	Java	Laufzeit
`std::vector<T>`	`java.util.ArrayList<T>`	
`operator[]`	`get()`	$\mathcal{O}(1)$
`push_back()`	`add()`	$\mathcal{O}(1)^\star$
`pop_back()`	-	$\mathcal{O}(1)$
`erase()`	`remove()`	$\mathcal{O}(n)$
`clear()`	`removeAllElements()`	$\mathcal{O}(1)$
`size()`	`size()`	$\mathcal{O}(1)$
`empty()`	`isEmpty()`	$\mathcal{O}(1)$
`back()`	-	$\mathcal{O}(1)$
`resize()`	`setSize()`	$\mathcal{O}(n)$
`reserve()`	`ensureCapacity()`	$\mathcal{O}(n)$

*amortisiert

Dynamisches Array (Verwendung)

```cpp
#include <vector>
using namespace std;

int main() {
        vector<int64_t> numbers { 1, 2, 3 };
        numbers.push_back(4);
        numbers.push_back(5);
        cout << numbers[3] << endl; // Ausgabe: 4

        // gibt alle Zahlen aus: 1 2 3 4 5
        for(int64_t i = 0; i < numbers.size(); ++i) {
                cout << numbers[i] << "␣";
        }
}
```

Dynamisches Array (Verwendung II)

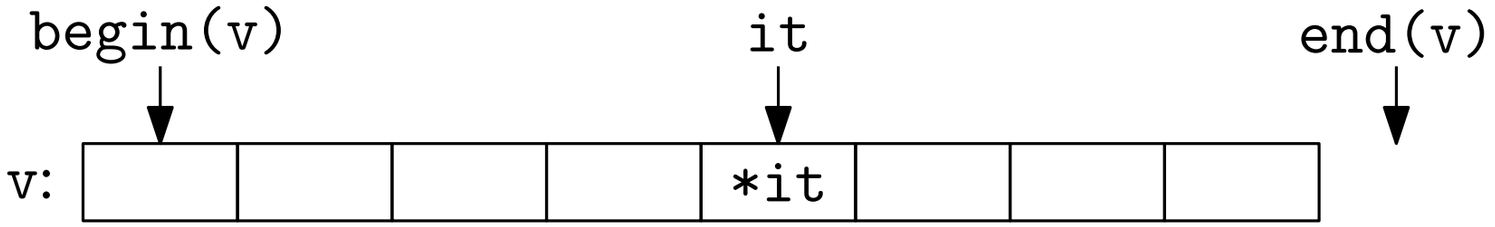
```
struct TodoItem {
        string name;
};
// ...
vector<TodoItem> stack;
stack.push_back({"Einkaufen"});
stack.push_back({"Spülen"});
// ...
while (!stack.empty()) {
        cout << stack.back().name << endl;
        stack.pop_back();
}
```

```
void pop_back();
```

Iteratoren

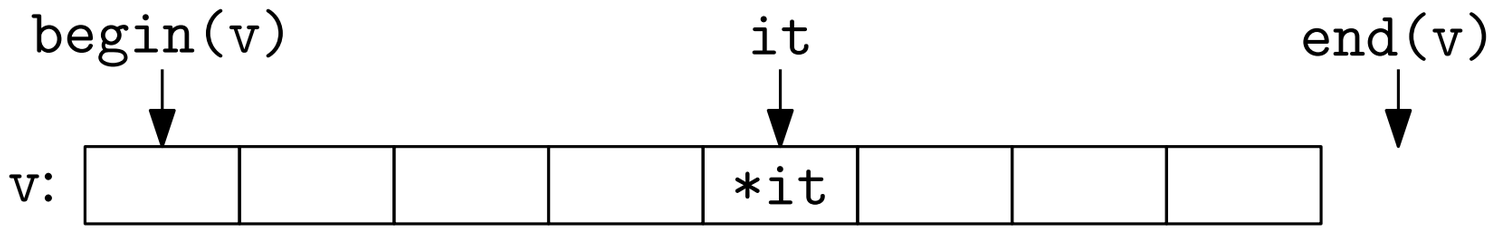
- *Iterator* ist ein Zeiger auf ein Objekt in einem Container
- Symbolischer End-Iterator zeigt auf Ende des Containers (**nicht** auf letztes Element)

C++	Java
`begin()` `end()`	`iterator()`
`++iterator` `*iterator`	`next()`



Verwendung von Iteratoren

```
vector<int64_t> numbers { 1, 2, 3, 4, 5 };
for (auto it = begin(numbers);
     it != end(numbers);
     ++it) {
  cout << *it << "␣";
}
//äquivalent:
for (int64_t x : numbers) {
  cout << x << "␣";
}
```



Vorsicht beim Modifizieren von Containern

```
vector<int64_t> numbers { 1, 2, 3, 4, 5 };
for (auto it = begin(numbers); it != end(numbers); ++it) {
    if (*it % 2 != 0) {
        numbers.erase(it);
    }
}
```

Fehler! Modifikation des Containers invalidiert die Iteratoren (auch beim Einfügen)

Korrekt:

```
vector<int64_t> numbers { 1, 2, 3, 4, 5 };

numbers.erase(remove_if(
                begin(numbers),
                end(numbers),
                [](const int64_t & x) { return x%2 != 0; }),
              end(numbers));
```

Sortieren und Suchen

- Arrays kann man sortieren und in ihnen suchen.

C++	Java

`#include <algorithm>`
 	
`std::sort(begin,end)`
	`import java.util.Array`
	`import java.util.Collections`
	`sort()`

- Operationen für sortierte Arrays/Vektoren:

C++	Java
`std::lower_bound` `std::upper_bound`	`binarySearch()`

- Beispiel:

```
vector<int64_t> v {3, 1, 2, 2, 5};
sort(begin(v), end(v));                 /* v = {1,    2, 2,    3, 5} */
lower_bound(begin(v),end(v),2); /* v = {1, ->2, 2,    3, 5} */
upper_bound(begin(v),end(v),2); /* v = {1,    2, 2, ->3, 5} */
distance(begin(v), lower_bound(begin(v),end(v),2)); /* Index: 1 */
distance(begin(v), upper_bound(begin(v),end(v),2)); /* Index: 3 */
```

C++: `std::next_permutation`

`std::next_permutation`:

- Ordnet die Elemente in die (lexikographisch) nächstgrößere Permutation um
- Vorher sortieren um mit kleinster Permutation zu starten
- `std::next_permutation` gibt `false` zurück,
  wenn es keine größere Permutation gibt.

Beispiel: 123 → 132 → 213 → 231 → 312 → 321 → fertig

std::next_permutation (Verwendung)

```
string sequenz = "321";

// Ausgabe 321
do {
        cout << sequenz << "␣";
} while (next_permutation(begin(sequenz),end(sequenz));
cout << endl;

// Sortiert sequenz => sequenz = "123"
sort(begin(sequenz), end(sequenz));

// Ausgabe: 123 132 213 231 312 321
do {
        cout << sequenz << "␣";
} while (next_permutation(begin(sequenz),end(sequenz));
cout << endl;
```

Queue

Elemente werden hinten in Warteschlange eingefügt und vorne entfernt.
Funktioniert nach dem FIFO-Prinzip (first in, first out).

C++	Java	Laufzeit
`std::queue<T>`	`java.util.Queue<T>`	
`push()`	`add()`	$\mathcal{O}(1)^\star$
`pop()`	`remove()`	$\mathcal{O}(1)$
`front()`	`element()`	$\mathcal{O}(1)$
`empty()`	`isEmpty()`	$\mathcal{O}(1)$

*amortisiert

Wann benutzen?

- Breitensuche
- Warteschlangen

Queue (Verwendung)

```
vector<int64_t> v {3, 1, 7, 5};
queue<int64_t> q;

for (uint32_t i = 0; i < v.size(); ++i)
        q.push(v[i]);

// Ausgabe: 3 1 7 5
while (!q.empty()) {
        cout << q.front() << "␣";
        q.pop();
}
```

Deque

Double-ended queue, implementiert Queue- und Stackinterface.
Queue ist eine Deque mit eingeschränktem Interface.

C++	Java	Laufzeit
`std::deque<T>`	`java.util.ArrayDeque<T>`	
`push_back()`	`addLast()`	$\mathcal{O}(1)^\star$
`pop_back()`	`pollLast()`	$\mathcal{O}(1)$
`back()`	`getLast()`	$\mathcal{O}(1)$
`push_front()`	`addFirst()`	$\mathcal{O}(1)^\star$
`pop_front()`	`pollFirst()`	$\mathcal{O}(1)$
`front()`	`getFirst()`	$\mathcal{O}(1)$
`operator[]`	`get()`	$\mathcal{O}(1)$
`empty()`	`isEmpty()`	$\mathcal{O}(1)$

*amortisiert

Wo ist der Haken? ➡ `operator[]` etwas langsamer als bei `vector`

Überblick

IO

Lineare Datenstrukturen

Nicht-lineare Datenstrukturen

Tipps und Tricks

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

[xkcd.com/1185]

Balancierte Suchbäume

- Suchbäume stellen Operationen wie Einfügen, Suchen und Löschen zur Verfügung und halten Sortierung aufrecht.
- Balanciert: Operationen garantiert in $\mathcal{O}(\log n)$

C++	Java	Laufzeit
`std::set<T>`	`java.util.TreeSet<T>`	
`std::map<K,V>`	`java.util.TreeMap<K,V>`	
`operator[]`	`get()`	$\mathcal{O}(\log n)$
`insert()`	`add() / put()`	$\mathcal{O}(\log n)$
`count()`	`contains() / containsKey()`	$\mathcal{O}(\log n)$
`erase()`	`remove()`	$\mathcal{O}(\log n)$
`size()`	`size()`	$\mathcal{O}(1)$
`empty()`	`isEmpty()`	$\mathcal{O}(1)$

Map

Einträge einer `map<Key,Value>` bestehen aus Tupeln.
Die `map` ordnet Schlüsseln Werte zu.

`Key`	muss eindeutig sein. Elemente der `map` nach Schlüsseln sortiert.
`Value`	ist der Typ der abgespeicherten Werte.

C++	Java
`Key` muss den <-Operator überladen (`operator<`)	`Key` muss `Comparable` implementieren

std::map (Verwendung)

```cpp
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main() {
  map<string, int64_t> m;
  string s;
  while(cin >> s) {
    m[s]++;
  }

  for (auto& val : m) {
    cout << val.first << "␣" << val.second << endl;
  }
}
```

Eingabe:

```
icpc
contest
unitrier
icpc
icpc
```

Ausgabe:

```
contest 1
icpc 3
unitrier 1
```

std::map – Überladen von Operatoren

```
struct Point {
  int64_t x,y;
};

bool operator<(const Point& p1, const Point& p2) {
  return (p1.x < p2.x) || (p1.x == p2.x && p1.y < p2.y);
}

bool operator==(const Point& p1, const Point& p2) {
  return p1.x == p2.x && p1.y == p2.y;
}

// ...
map<Point, int64_t> m;
m[Point {2, 3}] = 10;
// ...
```

std::set

Ein Set repräsentiert eine sortierte Schlüsselmenge (ohne Duplikate)

Beispiel mit `lower_bound`:

```
set<int64_t> S;
S.insert(10);
S.insert(20);
S.insert(30);
auto it = S.lower_bound(15);
cout << *it << endl;
```

Ausgabe

```
20
```

Varianten

C++	Java
(unordered_)?(multi)?(map—set)	(Tree—Hash)(Map—Set)

- `multi...` erlaubt mehrere Einträge mit dem gleichen Schlüssel.
- `unordered_...` bzw. `Hash...` ist durch Hashtabelle implementiert. muss `std::hash<Key>` überladen bzw. `Key` muss `hashcode` überladen

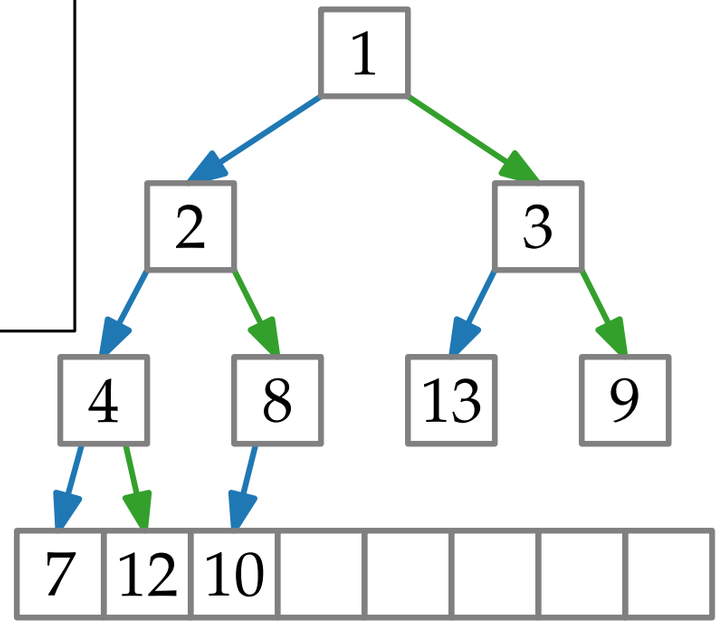
Für mehr Informationen

```
http://en.cppreference.com/w/cpp/container/map
http://en.cppreference.com/w/cpp/container/set
https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/TreeMap.html
https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/TreeSet.html
```

Heaps

- Heaps ermöglichen Finden des größten Elements einer Menge
- Baum erfüllt die Heap-Eigenschaft: Elternknoten hat größeren (kleineren) Wert als alle seine Kinder
  - ➡ größtes (kleinstes) Element an der Wurzel

C++	Java	Laufzeit
`std::priority_queue<T>`	`java.util.PriorityQueue<T>`	
`push()`	`put()`	$\mathcal{O}(\log n)$
`pop()`	`get()`	$\mathcal{O}(\log n)$
`top()`	`peak()`	$\mathcal{O}(1)$
größtes Element an Wurzel	kleinstes	



Verwendung:

- ereignisbasierte Simulation
- Suchalgorithmen in Graphen
- …

std::priority_queue (Anwendung)

```
priority_queue<int64_t> elems;
int64_t x;

// Input: 4 2 76 3 21 34 45 42
while(cin >> x) {
  elems.push(x);

  if (x == 42) {
    // Ausgabe: 76 45 42 34 21 4 3 2
    while (!elems.empty()) {
      cout << elems.top() << "␣";
      elems.pop();
    }
    cout << endl;
  }
}
```

Aufgabe:

Gegeben sei eine leere Schachtel. Nacheinander werden unterschiedlich nummerierte Kugeln eingeworfen.

Wird eine Kugel mit Nummer 42 in die Schachtel gelegt, sollen alle Kugeln der Größe nach (die Größte zuerst) aus der Schachtel entfernt werden.

Überblick

IO

Lineare Datenstrukturen

Nicht-lineare Datenstrukturen

Tipps und Tricks

INEFFECTIVE SORTS

```
DEFINE HALFHEARTEDMERGESORT(LIST):
    IF LENGTH(LIST) < 2:
        RETURN LIST
    PIVOT = INT(LENGTH(LIST) / 2)
    A = HALFHEARTEDMERGESORT(LIST[:PIVOT])
    B = HALFHEARTEDMERGESORT(LIST[PIVOT:])
    // UMMMMM
    RETURN [A, B] // HERE. SORRY.
```

```
DEFINE FASTBOGOSORT(LIST):
    // AN OPTIMIZED BOGOSORT
    // RUNS IN O(N LOG N)
    FOR N FROM 1 TO LOG(LENGTH(LIST)):
        SHUFFLE(LIST):
        IF ISSORTED(LIST):
            RETURN LIST
    RETURN "KERNEL PAGE FAULT (ERROR CODE: 2)"
```

```
DEFINE JOBINTERVIEWQUICKSORT(LIST):
    OK SO YOU CHOOSE A PIVOT
    THEN DIVIDE THE LIST IN HALF
    FOR EACH HALF:
        CHECK TO SEE IF IT'S SORTED
            NO, WAIT, IT DOESN'T MATTER
        COMPARE EACH ELEMENT TO THE PIVOT
            THE BIGGER ONES GO IN A NEW LIST
            THE EQUAL ONES GO INTO, UH
            THE SECOND LIST FROM BEFORE
        HANG ON, LET ME NAME THE LISTS
            THIS IS LIST A
            THE NEW ONE IS LIST B
        PUT THE BIG ONES INTO LIST B
        NOW TAKE THE SECOND LIST
            CALL IT LIST, UH, A2
        WHICH ONE WAS THE PIVOT IN?
        SCRATCH ALL THAT
        IT JUST RECURSIVELY CALLS ITSELF
        UNTIL BOTH LISTS ARE EMPTY
            RIGHT?
        NOT EMPTY, BUT YOU KNOW WHAT I MEAN
    AM I ALLOWED TO USE THE STANDARD LIBRARIES?
```

```
DEFINE PANICSORT(LIST):
    IF ISSORTED(LIST):
        RETURN LIST
    FOR N FROM 1 TO 10000:
        PIVOT = RANDOM(0, LENGTH(LIST))
        LIST = LIST[PIVOT:] + LIST[:PIVOT]
        IF ISSORTED(LIST):
            RETURN LIST
    IF ISSORTED(LIST):
        RETURN LIST:
    IF ISSORTED(LIST):  //THIS CAN'T BE HAPPENING
        RETURN LIST
    IF ISSORTED(LIST): // COME ON COME ON
        RETURN LIST
    // OH JEEZ
    // I'M GONNA BE IN SO MUCH TROUBLE
    LIST = [ ]
    SYSTEM("SHUTDOWN -H +5")
    SYSTEM("RM -RF ./")
    SYSTEM("RM -RF ~/*")
    SYSTEM("RM -RF /")
    SYSTEM("RD /S /Q C:\*")  //PORTABILITY
    RETURN [1, 2, 3, 4, 5]
```

[xkcd.com/1185]

Tipps und Tricks für C++

- Oft kann man `std::pair` oder `std::tuple` anstelle eigener Datentypen verwenden
- Unter Linux: `#include <bits/stdc++.h>` macht die komplette Standardbibliothek verfügbar.
- Funktionen in `<algorithm>` sind mächtig und lassen sich oft gut kombinieren.
  Beispiel: doppelte Elemente aus einem Array entfernen

```
vector<i64> array { 1, 4, 3, 2, 3, 1 };          // 1 4 3 2 3 1
sort(begin(array),end(array));                   // 1 1 2 3 3 4
auto last = unique(begin(array),end(array));     // 1 2 3 4 ->3 4
array.erase(last,end(array));                    // 1 2 3 4
```

`https://en.cppreference.com/w/`