



Algorithms and Data Structures

Data Types

Ulf Leser

Content of this Lecture

- Example
- Abstract Data Types
- Realization in Java

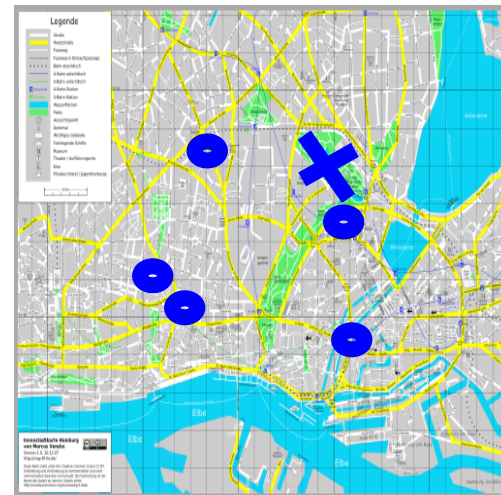
Problem

- Suppose you are in the centre of Hamburg and are **looking for the next (i.e., closest)** laptop repair shop
- Fortunately, your mobile knows your position and has a list of laptop repair shops in Hamburg
- How does your mobile find the **closest shop**?

Classical Post-Box Problem

- Suppose a city with n boxes located at arbitrary positions
- You wake up in the middle of the city with a letter in your hand; the letter should be thrown in the closest post-box
- How do you find the closest post-box?
 - You have a list with locations of all post boxes
- Looking at a map is not the answer!
- Devise an algorithm

```
S: set_of_coordinates;  
c: coordinate (x,y)  
...
```



Simple Solution

- How much work?

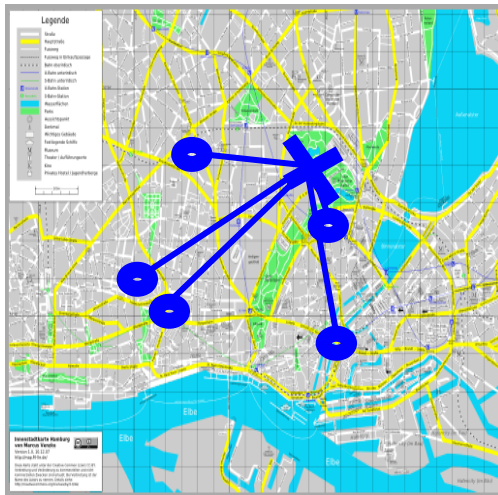
```
Input
  S: set_of_coordinates;
  c: coordinate (x,y);    # your loc
  t: coordinate;          # closest
  m: real := MAXREAL;     # smal. dist
  for each c' ∈ S do
    if m > distance(c,c') then
      m := distance(c,c');
      t := c';
    end if;
  end for;
  return t;
```

Simple Solution

```
Input
  S: set_of_coordinates;
  c: coordinate (x,y);    # your loc.
  t: coordinate;          # closest box
  m: real := MAXREAL;     # smal. dist
  for each c' ∈ S do
    if m > distance(c,c') then
      m := distance(c,c');
      t := c';
    end if;
  end for;
  return t;
```

- How much work?
- Clearly, we can save the second call to “distance”
- Thus, we need to compute $|S|$ distances, make $|S|$ comparisons, and perform at most $2 * |S|$ assignments

Simple Solution



- How much work?
- Clearly, we can save the second call to “distance”
- Thus, we need to compute $|S|$ distances, make $|S|$ comparisons, and perform at most $2 * |S|$ assignments
- Euclidian distance
 - 6 arithmetic ops per distance

$$\text{dist}((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Not the only Option



- How much work?
- Clearly, we can save the second call to “distance”
- Thus, we need to compute $|S|$ distances, make $|S|$ comparisons, and perform at most $2 * |S|$ assignments
- **Manhattan distance**
 - 5 operations, and different ones

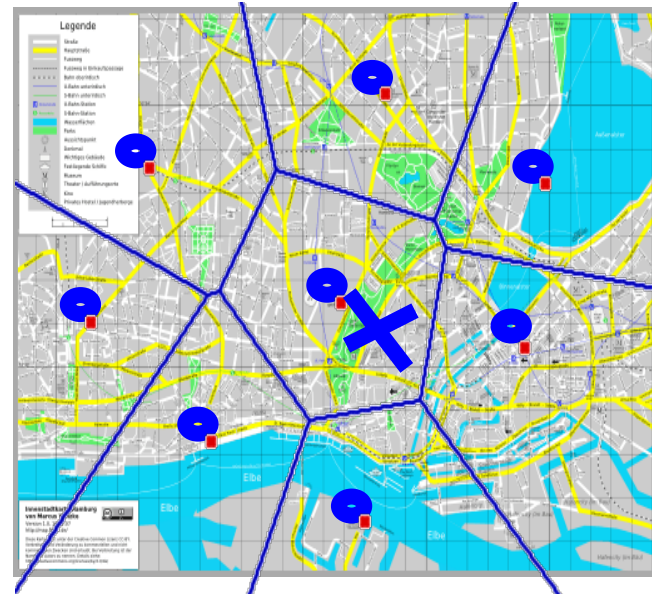
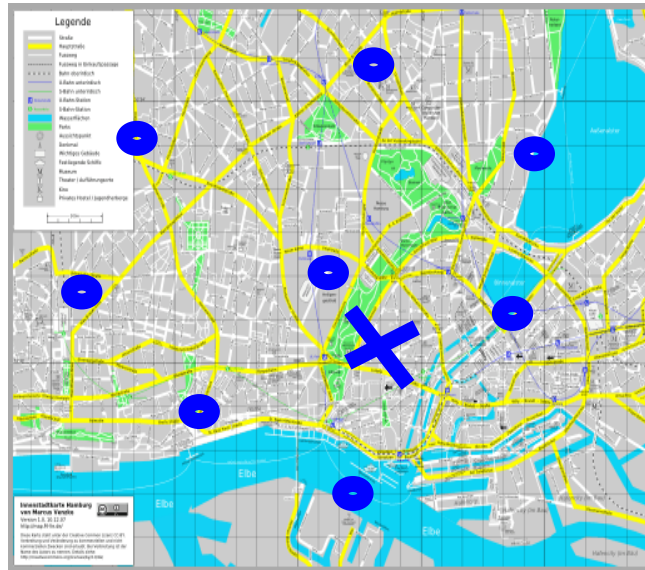
$$\text{dist}((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$$

Data Structure Point of View

```
input
  S: set_of_coordinates;
  c: coordinate (x,y);
t: coordinate;
m: real := MAXREAL;
For each c' ∈ S do
  if m > dist(c,c') then
    m := dist(c,c');
    t := c';
  end if;
end for;
return t;
```

- Data structures
 - We need a list of coordinates
 - The algorithm must iterate over the elements of this list
 - A **linked list** would suffice
- Now assume we need to perform **searches very often**
 - Can we represent S in **another way (S')**, such that searching requires less work?
 - Note: Time for computing S' from S may be ignored
 - Performed before searching starts
 - Assuming that S **does not change**

Voronoi Diagrams



- Compute for every point $s \in S$ its **Voronoi area**, i.e., the area in which all points have s as nearest point from S
- This is not easy, but can be achieved in **$O(|S| \cdot \log(|S|))$ time**
- Nearest-neighbor search using Voronoi diagrams is $O(\log(|S|))$
- Finding a **proper data structure** does pay off

More Abstract

- We want a **piece of software T** that
 - can store a list of coordinates
 - can compute the nearest point to a given point c
- Thus, T must support (at least) two **operations**
 - T.init (S: list_of_coordinate)
 - T.nearestNeighbor (c: coordinate)
 - T apparently uses another data type: “coordinate”
- Such **combinations of object sets and operations** on these sets are called a **data type**
- If we abstract from the implementation and only look at the sets and operations, we call this an **abstract data type**

Content of this Lecture

- Example
- Abstract Data Types
- Realization in Java

Abstract Data Types (ADT)

- It ADT defines a set of operations over a set of objects of a certain (more basic) type
 - Or over **multiple sets** of objects of different or same types
- The set of operations (and types) is an ADT's **signature**
- An ADT is **independent of a implementation**
 - Different data structures to represent the objects
 - Different algorithms to implement the operations
 - An ADT is independent of any programming language
- **Encapsulation**: Objects are accessed only through the ops
- An implementation of a ADT is called a **concrete (or physical) data type**

Example

```
type points
import
  coordinate;
operators
  add:      points x coordinate → points;
  neighbor: points x coordinate → coordinate;
```

- ADT that we could use for our app for searching shops
- We only need **two operations**
 - A way to insert shops (with their coordinates)
 - A way to get the nearest shop with respect to a given coordinate
- Not the only way ...

Modeling More Details

```
type shop
import
  coordinate;
  string;
operators
  getName: shop → string;
  getCoor: shop → coordinate;
```

```
type shops
import
  shop;
operators
  add: shops x shop → shops;
  neighborC: shops x coordinate → coordinate;
  neighborN: shops x coordinate → string;
  neighborS: shops x coordinate → shop;
```

- An ADT defines **what is necessary**
 - We assume primitive data types to be given (string)
- Design of ADT is a **modeling decision**
 - Shop owner? Laptop models being repaired? Opening hours?
 - Depends on necessity, design choices, understandability, extensibility, personal preferences, existing ADTs, ...

Reusing Existing ADTs

- For implementing points (or shops), it would be helpful to import something that can hold a set of coordinates
- We **need a list** – an ADT in itself
 - A **parameterized ADT** – a list of elements of an arbitrary ADT T
 - For our ADT points, T will be **coordinate**

```
type list( T)
import
    integer, bool;
operators
    isEmpty:  list → bool;
    add:      list x T → list;
    delete:  list x T → list;
    contains: list x T → bool;
    length:   list → integer;
```


Axioms: What we Know about an ADT

- We expect operations on lists to have a **certain semantic**
 - Adding an element increases length by one
 - If we assume **bag semantics**
 - Deleting an element that doesn't exist creates an error
 - If a list is empty, its length is 0
 - ...

```
type list( T)
import
  integer, bool;
operators
  isEmpty:  list → bool;
  add:      list x T → list;
  contains: list x T → bool;
  delete:   list x T → list;
  length:   list → integer;
axioms:  $\forall l: \text{list}, \forall t: T$ 
  length( add(l, t)) = length( l) + 1;
  length( l)=0  $\Leftrightarrow$  isEmpty(l);
  ...
```

List versus Points

```
type points
import
  coordinate, bool, list( coordinates);
Operators
  contains: points x coordinate → bool;
           # Implement as list.contains
  add:      points x coordinate → points;
           # Implement as list.add
  neighbor: points x coordinate → coordinate;
           # Not implemented in list!
axioms
  neighbor(p,c) = {x | contains(p,x) ∧ ∀x':contains(p, x')=>
                    distance(x,c) ≤ distance(x',c);
```

- What's wrong?
 - What happens if multiple x have the same distance to c ?

List versus Points

```
type points
import
  coordinate, bool, 2Dspace;
Operators
  contains: points x coordinate → bool;
  add:      points x coordinate → points;
  neighbor: points x coordinate → points;
axioms
  neighbor(p,c) = {x | contains(p,x) ∧ ∀x': contains(p,x'):
                                distance(x,c) ≤ distance(x',c);
```

Lists, Stacks, Queues

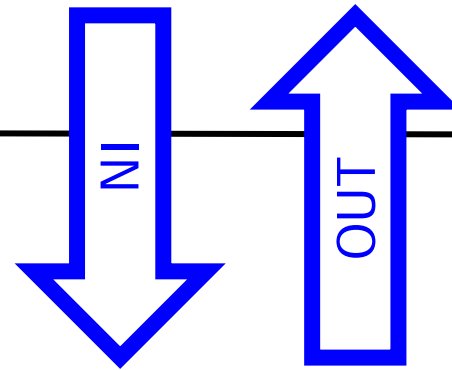
- We looked at a data type (points, shops) which essentially is a list with one **special operation**: nearestNeighbor
 - Canonical list operations: insert, search, delete, update, length
- There are many ways to implement the **general ADT list**
 - Array, linked lists, double-linked lists, trees, ...
- Two types of lists are of exceptional importance in computer science: **Stacks and Queues**
 - Both support mostly two operations
 - These suffice for surprisingly many problems and applications
 - Can be **implemented very efficiently**

Queues



- Operations: enqueue, dequeue
- Special semantic: First in, first out (FIFO)
- Breadth-first traversal, shortest paths, BucketSort, ...

Stacks



- Operations: push, pop
- Special semantic: Last in, first out (LIFO)
- Call stacks, backtracking, "Kellerautomaten", ...

As Abstract Data Types

```
type stack( T)
import
  bool;
operators
  isEmpty: stack → bool;
  push:    stack x T → stack;
  pop:     stack → stack;
  top:     stack → T;
```

```
type queue( T)
import
  bool;
operators
  isEmpty: queue → bool;
  enqueue: queue x T → queue;
  dequeue: queue → queue;
  head:    queue → T;
```

- Where's the **difference**?

Signature does not Suffice


```
type a( T)
import
  bool;
operators
  isEmpty: a → bool;
  add:      a x T → a;
  remove:   a → a;
  give:     a → T;
```

```
type a( T)
import
  bool;
operators
  isEmpty: a → bool;
  add:      a x T → a;
  remove:   a → a;
  give:     a → T;
```

- Where's the difference?
- From the **signature alone**, there is no difference
- Yet – we expect a **different behavior**

Defining the Difference

```
type stack( T)
import
  bool;
operators
  isEmpty: stack → bool;
  push:    stack x T → stack;
  pop:     stack → stack;
  top:     stack → T;
axioms ∀ q:stack, ∀ t:T
  top( push( s, t)) = t;
  pop( push( s, t)) = s;
```



Long version:

$\text{push}(s, t) \circ \text{top}(s) = t' \Rightarrow t = t'$

$\text{push}(s, t) \circ \text{pop}(s) = s' \Rightarrow s = s'$

```
type queue( T)
import
  bool;
operators
  isEmpty: queue → bool;
  enqueue: queue x T → queue;
  dequeue: queue → queue;
  head:    queue → T;
axioms ∀ q:queue, ∀ t:T
  head( enqueue( q, t)) =
    if isEmpty( q): t
    else head( q);
  dequeue( enqueue( q, t)) =
    if isEmpty( q): q
    else enqueue( dequeue( q), t);
```

Detour

```
type queue( T )
...
dequeue( enqueue( q, t )) =
  if isEmpty( q ): q
  else enqueue( dequeue( q ), t );
```

```
d( e( <3,2>, 5 )) = e( d( <3,2> ), 5 ) =
  e( d( e( <3>, 2 ) ), 5 ) =
  e( e( d( <3> ), 2 ), 5 ) =
  e( e( d( e( <> ), 3 ), 2 ), 5 ) =
  e( e( <>, 2 ), 5 ) =
  <2,5>
```

We Stop Here

- There are various ways to **formally specify the behavior of operations of** an ADT
- In this lecture, we only look at **signature**
 - No semantics (except parameters of operations)
 - Supported by **most programming languages** (e.g. Java)
- **Algebraic specification**
 - Define an algebra over the object sets of the ADT
 - Includes axioms defining the **semantics of operations**
 - Axioms are essential to **proof aspects** of a system's behavior
 - Supported by **few programming languages**
 - Ideally, one only specifies and never programs
- See lecture on “Modellierung und Spezifikation”

Content of this Lecture

- Data Structures Again
- Abstract Data Types
- Realization in Java

ADTs in Java

- Recall
 - An ADT summarizes the **essential operations** on a **set of objects**
 - An ADT is **independent of a realization**/implementation
 - Any implementation of a ADT is called a **concrete data type**
- Realization in Java?
- **Interfaces**
 - Only exhibit the essential operations on a class of objects
 - Can have different implementations
 - Can be implemented by a concrete class

Remarks

- Java **does not support axioms** on interfaces
 - Some other languages do, e.g. contracts in Eiffel
- Java adds functionality we mostly ignore, such as
 - **Inheritance** (syntactic sugar), different levels of visibility (public, protected, private, ...), overloading, ...
- Historically, **ADTs are a predecessor** of classes in programming languages
- ADTs can be realized at least in all OO languages
 - Critical: **encapsulation** – you must not see anything of an object / do anything with an object that is not represented in its (public) interface

Summary

- ADT's specify the possible **operations** on a data structure
- ADT's are **free of implementation** details
- We often discuss pros/contras of **different ways to implement** a given ADT
- (Formal) ADTs can be used for much more
 - Proving properties of a data type
 - Proving that a concrete data type implements a ADT
 - Proving that an implementation does not hurt axioms
 - **Program verification**