Algorithms and Data Structures

(Abstract) Data Types

Ulf Leser

Content of this Lecture

- Example
- Abstract Data Types
- Two important Examples: Stacks and Queues

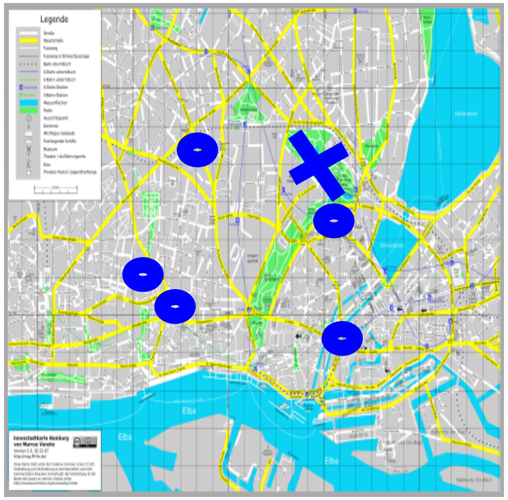
Problem

- 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

```
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;
```

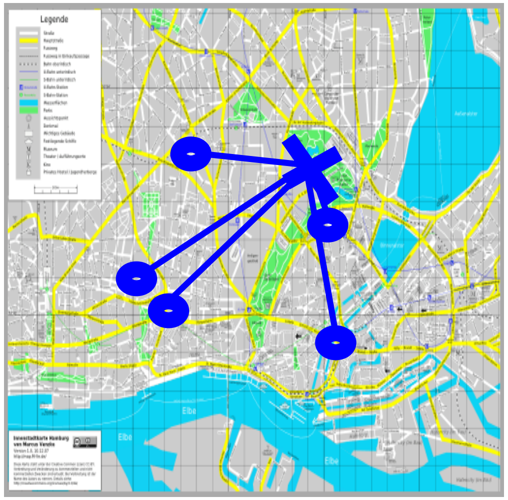
- How much work?

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;
```

- 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
- Together: We perform O(|S|) operations, which are either in O(1) or distance computations

Simple Solution



- We compute |S| distances …
- Euclidian distance
  - In 2D: 6 arithmetic ops

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

Not the only Option

- We compute |S| distances

  …

- Manhattan distance
  - 5 basic operations

$$dist((x_1, y_1),(x_2, y_2)) = | x_1 - x_2 | + | y_1 - y_2 |$$

Complexity

- We compute |S| distances

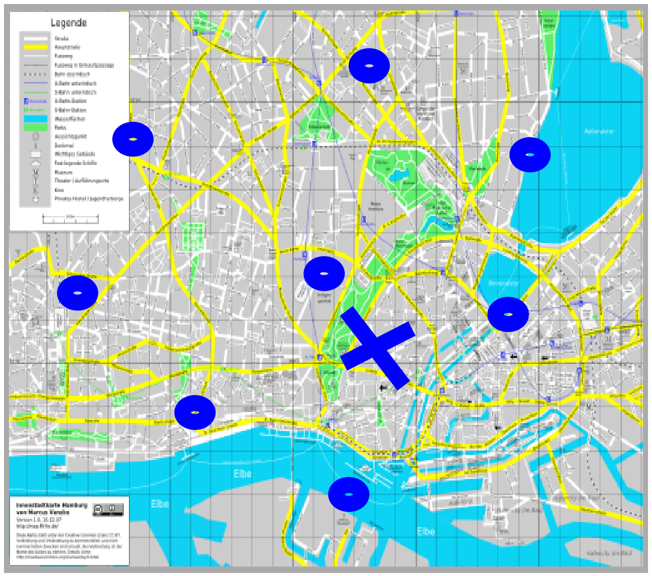
  …
- Both cases: O(|S|*dim(S))
  - dim(S): Number of dimensions of points in S
  - If dim(S)=k and considered a constant: O(|S|)

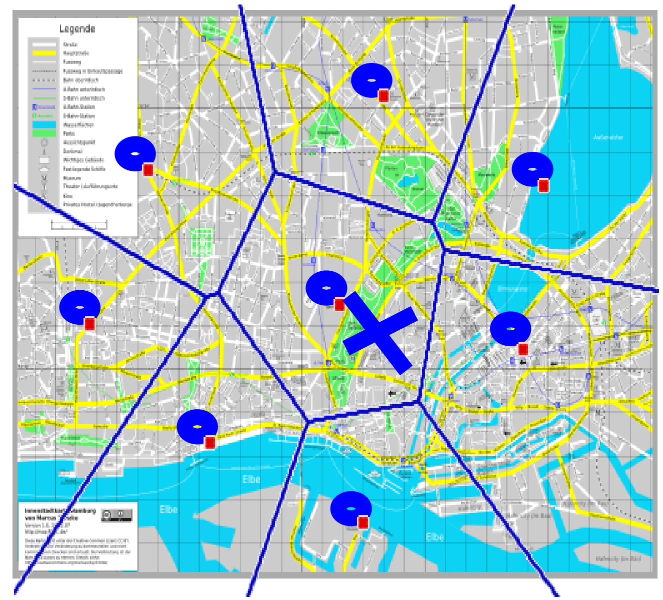
Data Structure Point of View

```
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 set S of 2D-coordinates
  - For NN-search, the algorithm must iterate over the elements of this set in any order
- Now assume we need to perform such searches very often
  - Can we represent S in another way (S'), such that searching requires less work?
  - Note: Time for computing S' from S will be ignored
    - Perform before searching starts
    - Assuming that S does not change

Voronoi Diagrams





- Pre-processing: 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
  - Can be achieved in O(|S|*log(|S|)) time (no details here)
- Nearest-neighbor search using Voronoi diagrams is O(log(|S|))
- Conclusion: Finding a proper data structure does pay off

Data Structures and Data Types

- A data structure is a computational representation of elementary objects
  - An array, a linked list, a matrix, a tree, a graph, ….
- A combination of data structure and operations on this structure is called a data type
  - "Operations": Application programming interface (API)
  - If we ignore implementation: Abstract data type
    - Also called signature
    - No complexity analysis, but correctness proofs
  - With concrete implementation: Physical data type
    - Software libraries

- ADT: Like a class in Java, i.e. variables and interface

Searching Shops

- We want a piece of software T that …
- T must store data
  - Set of coordinates (data structure)
- T must support (at least) two operations
  - T.init (S: set_of_coordinates)
  - T.nearestNeighbor(c: coordinate): coordinate
  - T apparently uses another data structure: “coordinate”
- T could have many more operations
  - T.insert(c: coordinate)
  - T.delete(c: coordinate)
  - T.print()
  - …

Content of this Lecture

- Example
- Abstract Data Types
- Two important Examples: Stacks and Queues

Abstract Data Types (ADT)

- An 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
- An ADT is independent of an implementation
  - Different physical means to represent the objects
  - Different algorithms to implement the operations
- Typical requirement: Encapsulation
  - Objects are accessed only through the operations

Example ADT

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

- ADT that we could use for our app for searching shops
- Defines two operations
  - A way to insert shops (with their coordinates)
  - A way to get the nearest shop with respect to a given coordinate
- Assumes a data type "coordinate" to be given
  - We always assume basic data types to be given: Int, real, string,…
- Not the only way

Modeling More Details

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

```
type shops
import
  shop;
operators
  add:         shops x shop → shops;
  n_neighborC: shops x coordinate → coordinate;
  n_neighborN: shops x coordinate → string;
  n_neighborS: shops x coordinate → shop;
```

- An ADT defines what is necessary and convenient
- Specifying an ADT is a design process
  - Shop owner? Laptop models being repaired? Opening hours?
  - Depends on requirements, ease-of-use, extensibility, personal preferences, existing ADTs, …
  - See lectures on Software Engineering

Reusing Existing ADTs

- For implementing shops, it would be helpful to reuse something that can manage a set of objects
- We need a set – an ADT in itself
  - A parameterized ADT– a set of elements of arbitrary type T
  - For our ADT **`points`**, T will manage objects of type **`coordinate`**

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

A data type – not a variable

Reusing Existing ADTs

- For implementing shops, it would be helpful to reuse something that can manage a set of objects
- We need a set – an ADT in itself
  - A parameterized ADT– a set of elements of arbitrary type T
  - For our ADT **`points`**, T will manage objects of type **`coordinate`**

```
type set( T)
import
  integer, bool;
operators
  isEmpty:  set → bool;
  add:      set x T → set;
  delete:   set x T → set;
  contains: set x T → bool;
  size:     set → integer;
  …
```

Java interface SET has ~20 operations

Axioms: What we know about an ADT

- We expect operations on sets to have a certain semantic
  - Adding an element increases size by one
  - If a set is empty, its length is 0
  - …
- These can be encoded as axioms: Conditions that must always hold
  - Defined as logical formulas
  - Also called invariants

```
type set( T)
import
operators
  isEmpty:   set → bool;
  add:       set x T → set;
  contains:  set x T → bool;
  delete:    set x T → set;
  length:    set → integer;
axioms: ∀ f: set, ∀ t: T
  size(add(f,t)) = size(f) + 1;
  size(f)=0 ⇔ isEmpty(f);
  …
```

Axioms: What we know about an ADT

- We expect operations on sets to have a certain semantic
  - Adding an element increases size by one
  - If a set is empty, its length is 0
  - …
- These can be encoded as axioms: Conditions that must always hold
  - Defined as logical formulas
  - Also called invariants
- But stop! Where is the error!

```
type set( T)
import
operators
  isEmpty:   set → bool;
  add:       set x T → set;
  contains:  set x T → bool;
  delete:    set x T → set;
  length:    set → integer;
axioms: ∀ f: set, ∀ t: T
  size(add(f,t)) = size(f) + 1;
  size(f)=0 ⇔ isEmpty(f);
  …
```

Axioms: What we know about an ADT

- We expect operations on sets to have a certain semantic
  - Adding an element increases size by one if not a duplicate
  - If a set is empty, its length is 0
  - …
- These can be encoded as axioms: Conditions that must always hold
  - Defined as logical formulas
  - Also called invariants

```
type set( T)
import
operators
  isEmpty:   set → bool;
  add:       set x T → set;
  contains:  set x T → bool;
  delete:    set x T → set;
  length:    set → integer;
axioms: ∀ f: set, ∀ t: T
  if contains(f,t) then
    ERROR;
  else
    size(add(f,t)) = size(f) + 1;
  size(f)=0 ⇔ isEmpty(f);
  …
```

Set versus Points

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

- `points` can build on a set, but must add further operations
- But there is a problem … which one?
  - What happens if multiple x have the same distance to c?

Set versus Points

```
type points
import
  coordinate, set(coordinate);
Operators
  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)};
```

Content of this Lecture

- Data Structures Again
- Abstract Data Types
- Two important examples: Stacks and Queues

Sets and Lists

- We looked at data types (points, shops) which essentially are sets
  - Canonical operations: add, contains, delete, size, …
  - And special operation: nearestNeighbor
- A related ADT is `list`
  - In a list, elements are ordered (arbitrarily yet fixed)
  - Canonical operations: addAt, contains, deleteAt, length, …
  - Different behavior (axioms)
    - Duplicates are no problem (same object at different positions)
    - No insertion after list end
    - …

One Take Home Message

- This lecture will be obsessed with lists and sets
- Why?
  - There are things
  - … and there a lists of things

- In CS, we need lists everywhere
  - Basis of every non-trivial algorithm
  - Investing effort in getting them efficient pays of in many many applications

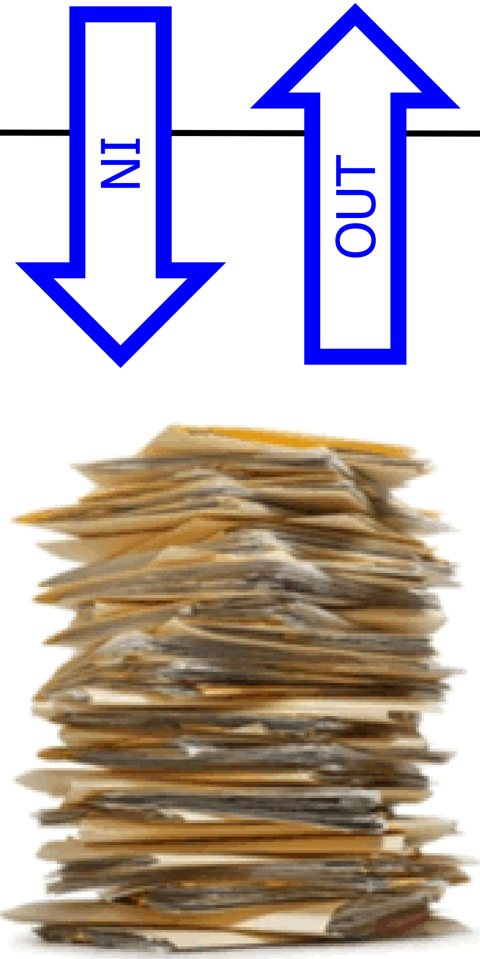
Stacks and Queues

- Two related ADTs are of exceptional importance in computer science: Stacks and Queues
  - Both support mostly two operations
    - No contains, length, addAt, deleteAt, …
  - These suffice for surprisingly many problems and applications
  - Both ADTs can be implemented very efficiently
    - More efficiently than sets or lists

Queues

- Two operations: Enqueue, dequeue
  - No access to objects of the list except the "head"
- Special semantic: First in, first out (FIFO)
- Apps: Breadth-first traversal, shortest paths, BucketSort, …

Stacks



- Operations: push, pop
  - No access to objects of the list except the “top”
- Special semantic: Last in, first out (LIFO)
- Apps: Call stacks, backtracking, “Kellerautomaten”, …

As Abstract Data Types

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

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

- Where is the difference?

Signature does not Suffice

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

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

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

Defining the Difference

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

```
type queue( T)
import
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);
```

Sets, Lists, Stacks, Queues

- Compared to sets
  - No contains
  - No duplicate checks before insertion
    - Much faster!
  - Typically no size
  - Additional behavior with push/pop
- Compared to lists
  - No contains, no order, no positions
    - Much faster!
  - Typically no size
  - Additional behavior with push/pop

Summary

- We very briefly sneaked into (abstract) data types
  - Formal syntax for specification, semantics of axioms in physical data types, concrete language for axioms, specialization hierarchies, formal correctness proofs, …
  - See module on "Methoden und Modelle des Systementwurfs"
- An old dream: Provide only precise specification and let all code be generated automatically
  - Provide so many axioms that all relevant behavior is covered
  - Enables formal proofs of correctness
  - Relevant especially for security-relevant domains
    - E.g. embedded systems in cars, airplanes, …
- Practically: Very time consuming, error prone, and hard to maintain

For this Lecture

- Algorithms take an input; input has a type; this type may offer special operations
  - Whose complexity depends on the physical implementation
- We rarely talk about the "data structure" aspect but about implementation of operations
  - Whose complexity also depend on complexity of operations on basic types
- As basic types, we assume Int, real, string
  - With operations add, multiply, compare, …
  - We assume O(1) for all basic operations

Exemplary Questions

- What is an abstract data type, what is a physical data type?
- What are typical operations of a list? Of a stack?
- Imagine a class storing rectangles in a plane. We want to add and remove rectangles, test if there are any rectangles, and find all rectangles intersection of given one. Define the ADT. What could be possible axioms?