Algorithmics

Sebastian Iwanowski
FH Wedel

1. Introduction into formal algorithmics

Algorithmics 1

1.1 Comparing basic sorting techniques

- Decription and functionality of algorithms:
  Permutationsort, Selectionsort, Mergesort, Quicksort

  Description in words, graphic visualization using arrays

- Estimating the run time for the worst case

  Setting up recursive equations, computing an explicite solution

  Run time estimation using the Big-O notation

- Results:

Permutationsort:	O(exp(n))
Selectionsort:	$O(n^2)$
Mergesort:	O(n log n)
Quicksort:	$O(n^2)$

References
Alt S. 4 – 7 (in German), Cormen ch. 2, Levitin ch. 3.1, ch. 4

visual demonstration: https://www.youtube.com/watch?v=yn0EgXHb5jc

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Selectionsort: „brute force“ strategy

- Pass all positions of data array in order.
- Search the minimum element upward from current position.
- Swap this element with element of current position.
- Output the new array after all positions have been passed.

```
procedure selectionsort (data): array
begin
  pos := 1;
  while pos < length(data) do
  begin
    newPos := minPos (data, pos, length(data));
    aux := data[pos];
    data[pos] := data[newPos];
    data[newPos] := aux;
    pos := pos + 1;
  end; {while}
  return data;
end {selectionsort}
```

```
procedure sort (data): array
begin
  newData := copy (data);
  return selectionsort (newData);
end {sort}
```

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of auxiliary procedure *minPos*:

```
procedure minPos (data, first, last): integer
begin
  resultPos := first;
  resultValue := data[resultPos];
  pos := first;
  while pos < last do
  begin
    pos := pos + 1;
    if data[pos] < resultValue
      then
        begin
          resultPos := pos;
          resultValue := data[resultPos];
        end;
  end; {while}
  return resultPos;
end {minPos}
```

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Mergesort: „divide and conquer“ strategy

- Divide data array into 2 halves.
- Sort the halves separately.
- Merge the sorted halves into a second array.

```
procedure mergesort
          (fromData, toData, left, right)
begin
  if left < right
  then
    begin
      mid := (left + right) div 2;
      mergesort (toData, fromData,
                               left, mid);
      mergesort (toData, fromData,
                           mid+1, right);
      merge (fromData, toData,
               left, mid, mid+1, right);
    end {if}
  end {mergesort}
```

Recursive version

```
procedure sort (data): array
begin
  data1 := copy (data);
  data2 := copy (data);
  mergesort (data1,
     data2, 1, length(data));
  return data2
end {sort}
```

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of auxiliary procedure *merge*:

```
procedure merge (fromData, toData, left1,
                     right1, left2, right2)
begin
  pos1 := left1; pos2 := left2; pos := left1;
  while (pos ≤ right2) do
  begin
    if pos1 > right1 /** first array has been used up already **/
      then
        begin toData[pos] := fromData[pos2]; pos2++ end
      else if pos2 > right2 /** second array has been used up already **/
        then
          begin toData[pos] := fromData[pos1]; pos1++ end
      else if fromData[pos1] ≤ fromData[pos2] /** regular case **/
        then
          begin toData[pos] := fromData[pos1]; pos1++ end
        else
          begin toData[pos] := fromData[pos2]; pos2++ end;
    pos++;
  end {while}
end {merge}
```

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Mergesort: „divide and conquer“ strategy

- Divide data array into 2 halves.
- Sort the halves separately.
- Merge the sorted halves into a second array.

Iterative version

```
procedure mergesortIter (data): array
begin
  data2 := copy (data); n := length(data);
  sortedLength := 1;
  while sortedLength < n do
  begin
    left1 := 1;
    while (left1+sortedLength) < n do
    begin
      right1 := left1+sortedLength; left2 := right1+1; right2 := left2+sortedLength;
      merge (data, data2, left1, right1, left2, right2);
      left1 := right2 + 1
    end;
    sortedLength := sortedLength + sortedLength;
    aux := data; data := data2; data2 := aux
  end;
  return data
end {sort2}
```

```
procedure sort (data): array
begin
  newData := copy (data);
  return mergesortIter(newData)
end {sort}
```

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Quicksort „divide and conquer“ strategy

- Quicksort (A, i, j):

  A is an array of n elements (a[1], …, a[n]).
  i,j are indices between 1 and n.
  At the end, the elements between a[i] and a[j] are sorted in an increasing order.

- Partition (A,i,k,j) → **`order number`**:

  At the end, A is rearranged between a[i] and a[j] such that
  first, only elements ≤ x := a[k] are placed, then x, then only elements > x.
  The return value **`order number`** is the new position of x.

- Implementation of Quicksort (A, i, j): Start with Quicksort (A,1,n)

```
if i < j
   then k := random number between i and j; /** k is the Pivot index *//
        dividingIndex := Partition (A,i,k,j);
            /** dividingIndex is the order number of the Pivot element *//
        Quicksort (A, i, dividingIndex-1);
        Quicksort (A, dividingIndex+1,j);
```

References:
Cormen ch. 7.1 (algorithm there without random number)
Levitin ch. 4.2

Algorithmics 1

1.1 Comparing basic sorting techniques

Details of Quicksort „divide and conquer“ strategy

- Partition (A,i,k,j) → **`order number`**:

  At the end, A is rearranged between a[i] and a[j] such that
  first, only elements ≤ x := a[k] are are placed, then x, then only elements > x.
  The return value **`order number`** is the new position of x.

- Implementation of Partition:

```
x := a[k];
count := number of elements ≤ x between a[i] and a[j];
order := i+count-1;
Swap x with a[order]; // now x is placed on correct new position
right := j;
for left := i to count-2 do
   if a[left] > x
      then while a[right] > x do right := right - 1;
           Swap a[left] with a[right];
return order;
```

References:
Cormen ch. 7.1 (algorithm there without random number)
Levitin ch. 4.2

Algorithmics 1

1.1 Comparing basic sorting techniques

Exact run time estimate: $\Theta(n^2)$

- lower run time estimate $\Omega(n^2)$:

  For each n there is an input of size n with run time in $\Omega(n^2)$

- upper run time estimate $O(n^2)$:

  using the recursive equation of script and explicite solution of the following: $T(n) \leq c \cdot n^2$
  (proof by mathematical induction using n)

  Remark to German script:
  The proposition that k=1 or k= n are the worst cases (which is true) is not proven in the script, but this is not necessary to show in order to show the above run time limits.

References:
Alt S. 7 (in German)
Cormen ch. 7.2

Algorithmics 1

1.2 Complexity measures for the analysis of algorithms

Calculating with Landau symbols (“asymptotic size”)

- Definition of O, Ω and Θ

  $T(n) \in O(f(n)) \Leftrightarrow \exists c \in \mathbb{R}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0: T(n) \leq c \cdot f(n)$
  $T(n) \in \Omega(f(n)) \Leftrightarrow \exists c \in \mathbb{R}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0: T(n) \geq c \cdot f(n)$
  $T(n) \in \Theta(f(n)) \Leftrightarrow \exists c_1, c_2 \in \mathbb{R}\ \exists n_0 \in \mathbb{N}\ \forall n \geq n_0: c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)$

- Computational rules for Landau symbols

  1) $x < y \Rightarrow O(n^x) \subsetneq O(n^y)$

  2) $x > 0 \Rightarrow O(\log n) \subsetneq O(n^x)$

  3) $O(f(n)+g(n)) \in O(f(n)) \cup O(g(n))$ (“maximum”)

References:
Cormen ch. 3

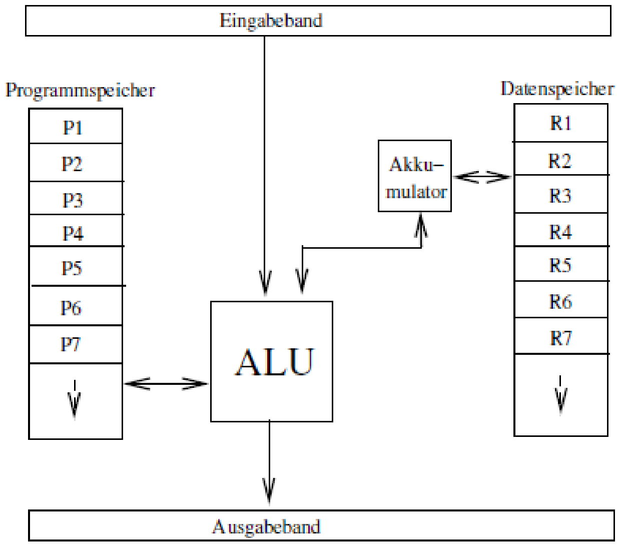
Algorithmics 1

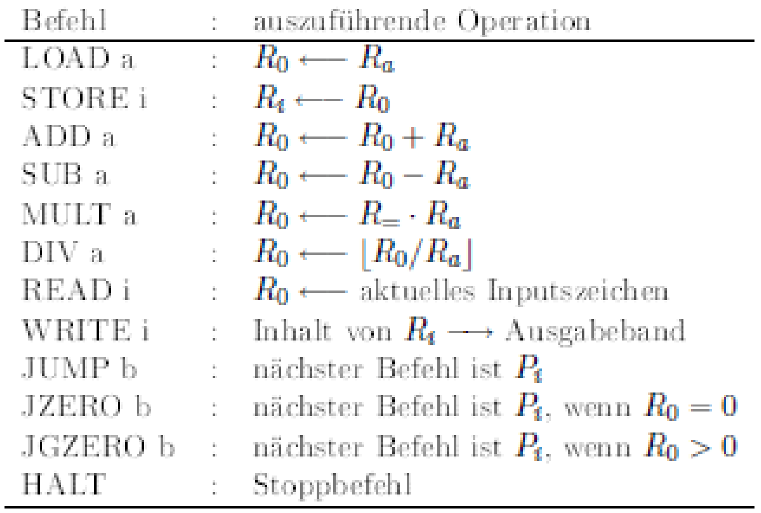
1.2 Complexity measures for the analysis of algorithms

Computational model: RAM (Random Access Machine)

- Definition of a RAM

  small assembler-like command pool,
  control unit with constant time access to program storage and data storage





Befehl	:	auszuführende Operation
LOAD a	:	$R_0 \longleftarrow R_a$
STORE i	:	$R_i \longleftarrow R_0$
ADD a	:	$R_0 \longleftarrow R_0 + R_a$
SUB a	:	$R_0 \longleftarrow R_0 - R_a$
MULT a	:	$R_0 \longleftarrow R_= \cdot R_a$
DIV a	:	$R_0 \longleftarrow \lfloor R_0 / R_a \rfloor$
READ i	:	$R_0 \longleftarrow$ aktuelles Inputszeichen
WRITE i	:	Inhalt von $R_i \longrightarrow$ Ausgabeband
JUMP b	:	nächster Befehl ist P_i
JZERO b	:	nächster Befehl ist P_i, wenn $R_0 = 0$
JGZERO b	:	nächster Befehl ist P_i, wenn $R_0 > 0$
HALT	:	Stoppbefehl

from Lang, ch. 4.5

References:

Alt S. 11-13 (in German)
Skript Lang, Kap. 4.5 (in German)

Mehlhorn ch. 2.2, 2.3 (outline, with a different perspective)

Algorithmics 1

1.2 Complexity measures for the analysis of algorithms

Computational model: RAM (Random Access Machine)

- Cost measures

  UCM: All operations cost the same independent of operands' size.

  LCM: The cost of an operation depends on size of operand.

- Run time equivalence

  Algorithm requires on a RAM time in Θ(f(n)) (UCM oder LCM)
  ⇔ Algorithm requires the same time class Θ(f(n)) on a „normal" computer.

- Polynomial relation

  Algorithm requires on a RAM time in Θ(f(n)) using LCM
  ⇔ Algorithm requires on a Turing machine time in Θ(P(f(n))) for a polynomial P.

References:
Alt S. 11-13 (in German)
Skript Lang, Kap. 4.5 (in German)
Mehlhorn ch. 2.2, 2.3 (outline, with a different perspective)

Algorithmics 1

1.2 Complexity measures for the analysis of algorithms

Master-Theorem
for the asymptotic run time estimation of divide & conquer algorithms

Let $T(n)$ be the recursive equation for a divide & conquer algorithm given by:

$$T(n) = a\,T(n/b) + f(n)$$

Then for $b > 1$ and $f(n) \in \Theta(n^k)$ the following holds:

1) $a < b^k \Rightarrow T(n) \in \Theta(n^k)$

2) $a = b^k \Rightarrow T(n) \in \Theta(n^k \log n)$

3) $a > b^k \Rightarrow T(n) \in \Theta(n^{\log_b a})$

The same results hold for O and Ω

References:
Cormen ch. 4

Algorithmics 1

1.2 Complexity measures for the analysis of algorithms

Denoting the complexity of algorithms by Landau symbols

Let I(A) be an admissible input for algorithm A and size(I(A)) be the input size.
Let T_A(I(A)) be the run time of A (counting the number of operations), when I(A) is the input.

- Upper run time limit in worst case:

  A is an O(f(n)) algorithm $\Leftrightarrow \forall n \in \mathbb{N}\ \forall$ I(A),size(I(A))=n: $T_A(I(A)) \in O\,(f(n))$
  "All inputs are bounded by this asymptotic run time."

- Lower run time limit in worst case:

  A is an Ω(f(n)) algorithm $\Leftrightarrow \forall n \in \mathbb{N}\ \exists$ I(A),size(I(A))=n: $T_A(I(A)) \in \Omega\,(f(n))$
  "For each n there is an input with this asymptotic run time bound."

- Exact asymptotic run time in worst case:

  A is a Θ(f(n)) algorithm in a weak sense $\Leftrightarrow$
  A is an O(f(n)) algorithm and A is an Ω(f(n)) algorithm

  A is a Θ(f(n)) algorithm in a strong sense $\Leftrightarrow \forall n \in \mathbb{N}\ \forall$ I(A),size(I(A))=n: $T_A(I(A)) \in \Theta\,(f(n))$
  "All inputs have this asymptotic run time."

References: ? (thanks for giving me hints)

Algorithmics 1

1.3 Lower bounds for algorithms using comparisons only

- Lower bound for the search of a maximal element

  Given n elements (input size).
  Compare graph must be connected ➔ at least n-1 comparisons ($\Omega(n)$)
  There is an O(n) algorithm for this problem ➔ This algorithm is optimal.

- Lower bound for the search of the k-th element of a given set

  Given n elements (input size).
  Compare graph must be connected ➔ at least n-1 comparisons ($\Omega(n)$)
  Optimal algorithm for this problem? ➔ Chapter 2

- Lower bound for sorting

  Correlate depth of a compare tree with the number of comparisons
  Correlate depth of a binary search tree with the number of leaves
  Estimate n! and make a conclusion for log (n!) ➔ at least $\Omega(n \log n)$ comparisons
  Mergesort needs only O(n log n) comparisons ➔ Mergesort is optimal.

References:
Alt S. 17 – 21 (in German)   Cormen ch. 8.1   Levitin ch. 11.1 (outline)

Algorithmics

Sebastian Iwanowski
FH Wedel

2. Advanced searching and sorting

Algorithmics 2

2.1 Order statistics

`SELECT (k,A): Element`

Search for the k-th element of an unsorted array A with n elements, i.e. search for the Element x ∈ A such that k Elements of A are less or equal.

Straightforward solutions

1. Sort the array and then determine the k-th element. — run time $\Theta(n \log n)$ w.c. and a.c.

2. k = 1 or k = n:
   Traverse the array in a single pass
   and update the current maximum or minimum during traversal. — run time $\Theta(n)$ w.c. and a.c.

References:

Alt, S. 23 (in German)

Cormen, ch. 9.1

Levitin, ch. 5.6 (first problem, very superficial)

Algorithmics 2

2.1 Order statistics

`SELECT (k,A): Element`

Search for the k-th element of an unsorted array A with n elements, i.e. search for the Element $x \in A$ such that k Elements of A are less or equal.

Randomised algorithm run time $\Theta(n^2)$ w.c., $\Theta(n)$ a.c.

If A contains less than c elements (for an abitrary constant c), determine the k-th Element directly with a straightforward method.
Else:

1. Choose an arbitrary index j from {1,...,n} and consider element a = A[j].
2. Swap the elements of A with quicksort partition such that a is placed into its correct position, all smaller elements left of a and all greater elements right of a.
   Partition A into three subsequences $A_<$, $A_=$ and $A_>$.
3. $|A_<| < k \leq |A_<| + |A_=|$ => return a
   $k \leq |A_<|$ => return SELECT (k, $A_<$)
   $k > |A_<| + |A_=|$ => return SELECT (k - ($|A_<|+|A_=|$), $A_>$)

References:
Alt, S. 23 – 26 (in German) Cormen, ch. 9.2

Algorithmics 2

2.1 Order statistics

`SELECT (k,A): Element`

Search for the k-th element of an unsorted array A with n elements, i.e. search for the Element $x \in A$ such that k Elements of A are less or equal.

Deterministic algorithm run time $\Theta(n)$ w.c. and a.c.

If A contains less than c elements (for an abitrary constant c), determine the k-th Element directly with a straightforward method.
Else:

1. Partition A into $\lceil n/5 \rceil$ subsequences of length 5.
2. Sort each subsequence i and determine its median a_i.
3. Determine the median a of the medians a_i invoking SELECT($\lceil n/10 \rceil$,$\{a_1,\ldots,a_{\lceil n/5 \rceil}\}$)
4. Swap the elements of A with quicksort partition such that a is placed into its correct position, all smaller elements left of a and all greater elements right of a.
   Partition array A into three subsequences $A_<$, $A_=$ and $A_>$.
5. $|A_<| < k \le |A_<| + |A_=|$ => return a
   $k \le |A_<|$ => return SELECT (k, $A_<$)
   $k > |A_<| + |A_=|$ => return SELECT (k - ($|A_<|+|A_=|$), $A_>$)

References: Alt, S. 27 – 29 (in German) Cormen, ch. 9.3

Algorithmics 2

2.2 Searching in sorted arrays

`SELECT (a,A,left,right): Element`

Search for index of element a in an n-element sorted array A[1..n] between indices left and right. If a $\notin$ A, return 0.

Advantage to dynamic data structures (cf. ch. 3): less storage space

1. Binary search	run time $\Theta(\log n)$ w.c. and a.c.
2. Interpolation search	run time $\Theta(n)$ w.c. and $\Theta(\log(\log n))$ a.c.
3. Quadratic binary search	run time $\Theta(n^{0,5})$ w.c. und $\Theta(\log(\log n))$ a.c.

References:

Alt, S. 30 – 35 (in German)	Levitin, ch. 4.3; ch. 5.6 (second problem)
Mehlhorn 1988 (in German)	Sedgewick ch. 14 (superficial)
Cormen ?	Knuth ch. 6.2.1

Algorithmics 2

2.3 Sorting in finite domains

- Countingsort (for n data with k different values)

  Simple variant (Alt), complex data variant (Cormen),
  run time $\Theta(n+k)$, stable, i.e. preserves the input order for equal data
  Utilisation: for all data with compare key domain that is discrete and linearily ordered

- Bucketsort (for n data within bounded limits)

  Originally only for real numbers from $[0,1)$ (Cormen), but can be used for any bounded domains having a function $f: U \rightarrow \{1,\dots,k\}$ which preserves the order ($f(x) \leq f(y) \Leftrightarrow x \leq y$) and which can be computed in constant time.

  The crucial run time advantage is sorting of buckets, which is $O(n \log n)$ w.c.,
  but for $k \in \Theta(n)$ run time is $O(n)$ a.c., even if bucket sorting is executed by $O(n^2)$ w.c. algorithm, proof in Cormen.
  If data consists of k different values only, sorting of k Buckets needs run time $\Theta(n+k)$ a.c., algorithm is stable.

References:
Alt, S. 21 – 22 (in German),
Cormen, ch. 8

Algorithmics 2

2.3 Sorting in finite domains

- Radixsort

  Sorting of n words of finite length s over alphabet with k characters.
  Run time $\Theta(s \cdot (n+k))$, uses countingsort (Cormen) or bucketsort (Alt) as subroutine.
  May be generalised to any domain which is lexicographically ordered, i.e. numbers in logarithmic representation.

References:
Alt, S. 21 – 22 (in German),
Cormen, ch. 8

Algorithmics

Sebastian Iwanowski
FH Wedel

3. Solutions for the dictionary problem
3.1 Hashing and other methods for optimizing the avarage case behaviour

Implementation of dictionaries

A dictionary is a data structure for elements comparable by a key implementing the functions search (key), insert (key, newdata) and delete (key)

Using a sorted array for a dictionary:

search (key)	run time Θ(log n) w.c. and Θ(log log n) a.c. achievable	☺
	not by the same algorithm	
insert (key, newdata)	run time Θ(n) w.c. and a.c.	☹
delete (key)	run time Θ(n) w.c. and a.c.	☹

Better method for insert / delete with indexed arrays: Hashing (cf. following slides)

References:

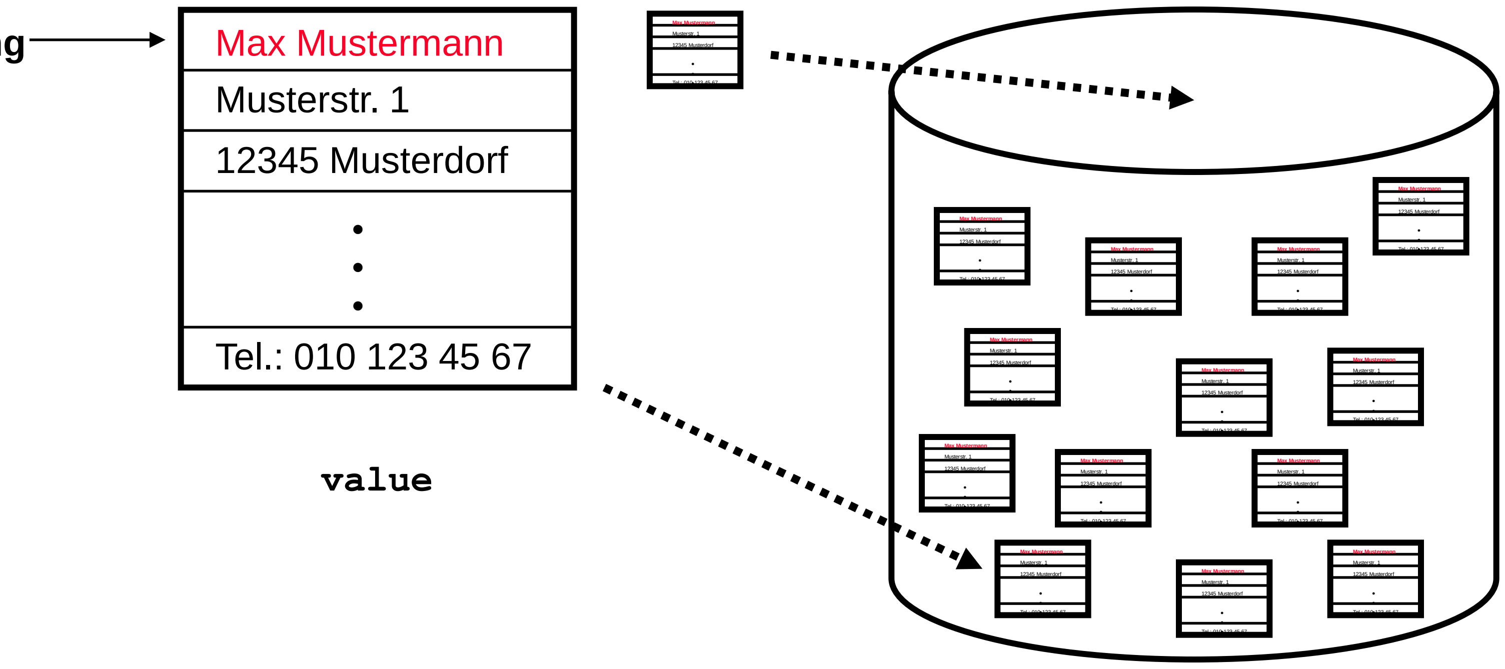
Skript Alt, S. 30 – 35 (for search) in German
more information: cf. previous chapter

Which problem does hashing solve?

data record: **data base:**

key for searching

identifies
data record
uniquely



`key` `value`

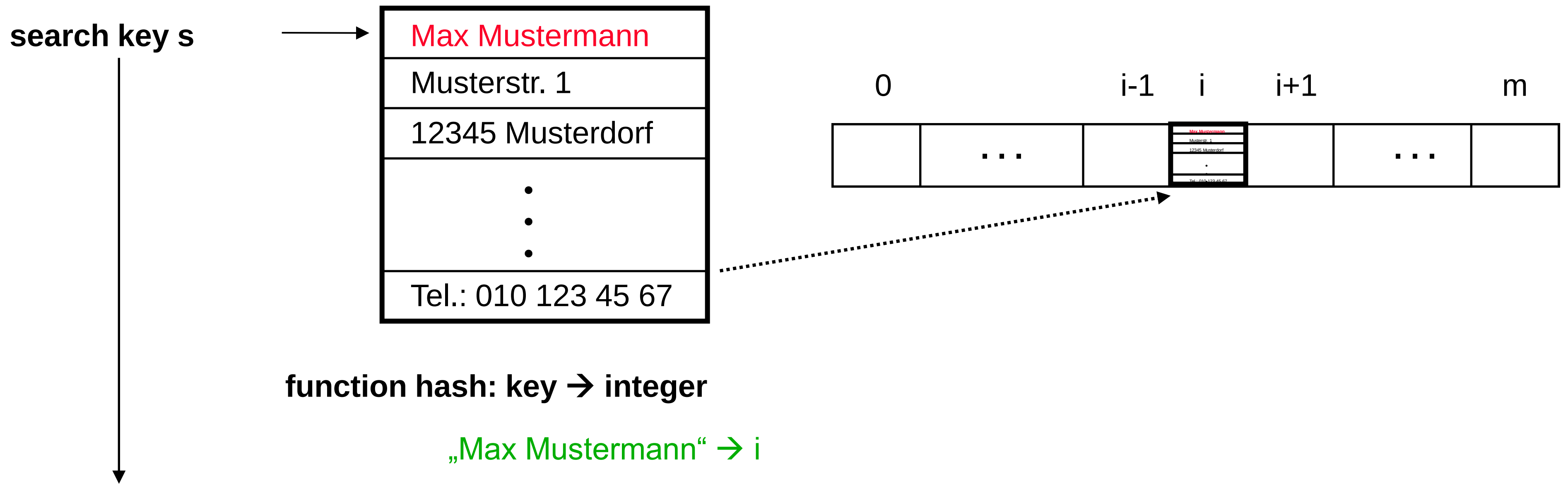
data administration operations: `map operations`

- **search** `get (key)`
- **insert** `put (key, value)`
- **delete** `remove (key)`

***Hashing* is a method implementing these operations efficiently.**

Outline of method

data record: **hash table T:**



function hash: key → integer

„Max Mustermann“ → i

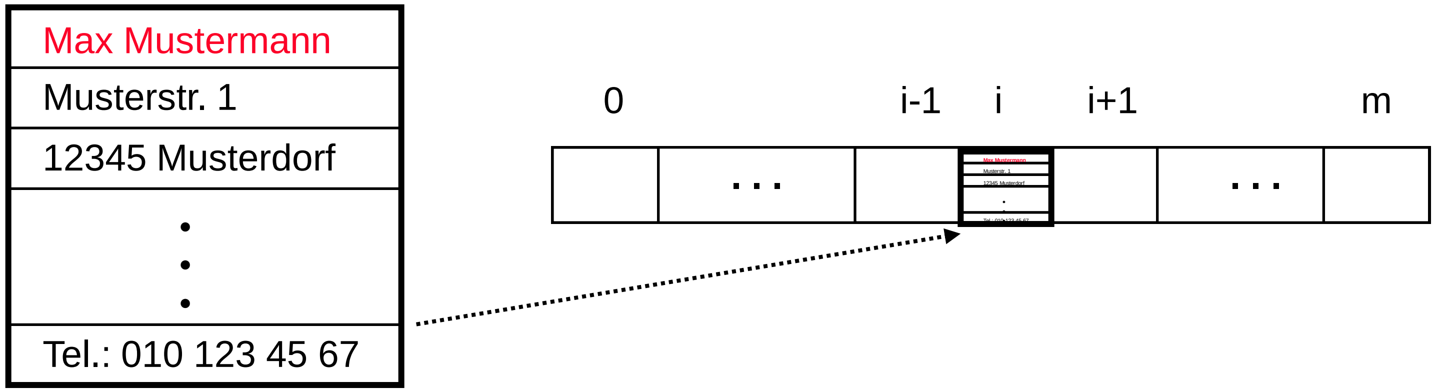
hash number hash(s) hash(„Max Mustermann“) = i

- **search** Determine i=hash(s) → Data record searched is in T[i].
- **insert** Determine i=hash(s) → Store new data record in T[i].
- **delete** Determine i=hash(s) → Delete data record from T[i]. `(T[i] = null)`

Discussing details

data record:

hash table T:



1) **How to define a good hash function?**

2) **Where to store the data record in the hash table?**

1) How to define a good hash function ?

Case 1: **Hash table contains at least as many records as different keys are possible.**

Goal: **Each key is mapped to a *different* hash number.**

perfect hashing

Solution: **Sort the keys by order (e.g. lexikographically) !**

Map each key to its order number!

Example: „Max Mustermann“ → (13 1 24 0 13 21 19 20 5 18 13 1 14 14)

(for strings as keys)

hash („Max Mustermann“) = $13*27^{13} + 1*27^{12} + 24*27^{11} + 0*27^{10} + 13*27^{9} + 21*27^{8} + 19*27^{7}$
$+ 20*27^{6} + 5*27^{5} + 18*27^{4} + 13*27^{3} + 1*27^{2} + 14*27^{1} + 14*27^{0}$
≈ 52966834350000000000 (20-digit number)

In general a lot of *different* keys are possible!

Conclusion: **Case 1 is not realistic !**

1) How to define a good hash function ?

Case 2: **Hash table contains fewer records than different keys that are possible.**

m k

Case 2: $m < k$

Conclusion: **Different keys have to be mapped to the same hash number**

collision

Goal:

Each hash number 0 thru m-1 is the function value of aproximately equally many keys (i.e. approximately k/m).

Solution: **Sort the keys by order (e.g. lexikographically) !**

Map each key to its order number modulo m !

1) How to define a good hash function ?

Example:

(for strings as keys)

m = 1000 „Antje“ $\rightarrow$ (1 14 20 10 5)

hash („Antje“) = $(1 * 27^4 + 14 * 27^3 + 20 * 27^2 + 10 * 27^1 + 5) \bmod 1000$

= 821858 mod 1000

= 858

Algorithm for a good hash function (according to *Horner*‘s method) :

hash(„Antje“) = ((((1 * 27 + 14) * 27 + 20) * 27 + 10) * 27 + 5) mod 1000

= (((((((1 mod 1000 * 27 + 14) mod 1000) * 27 + 20) mod 1000) * 27 + 10) mod 1000) * 27 + 5) mod 1000

Java code:

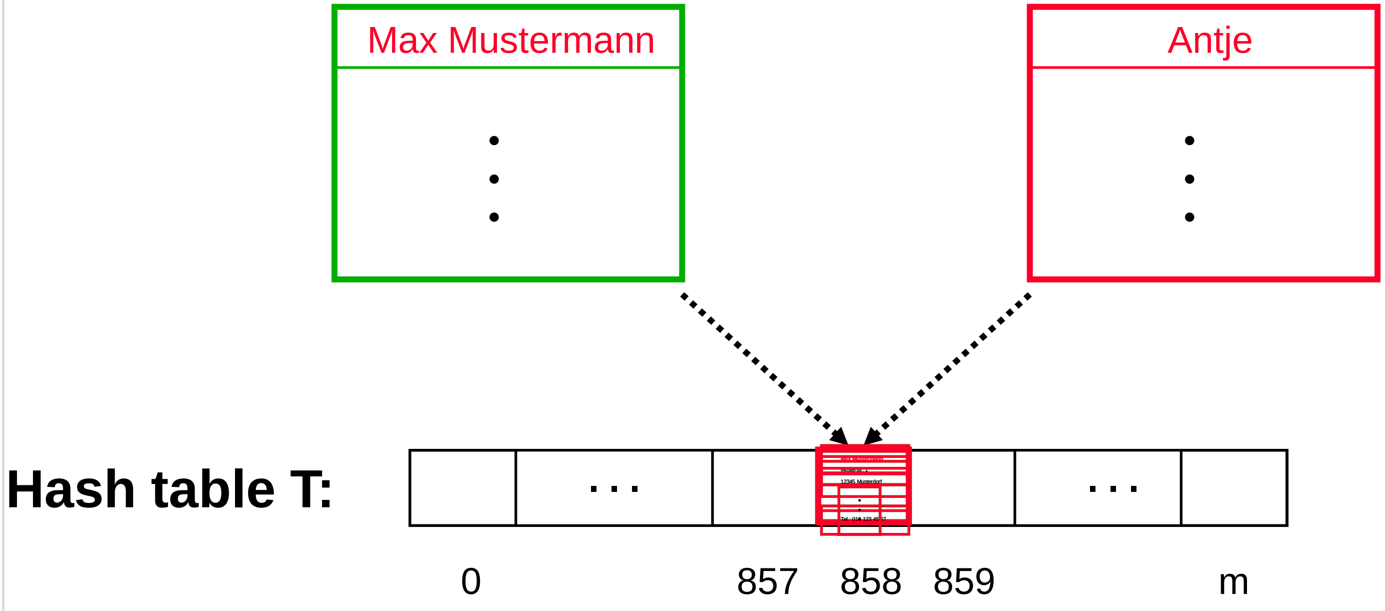
```
static int hash (String key, int m)
  {
    int result = 0, numberSymbols = 27;
    for (int i = 0; i < key.length(); i++)
      result = (result*numberSymbols + order (key.charAt(i))) % m;
    return result;
  }
```

2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Example: hash („Max Mustermann“) = 858

hash („Antje“) = 858

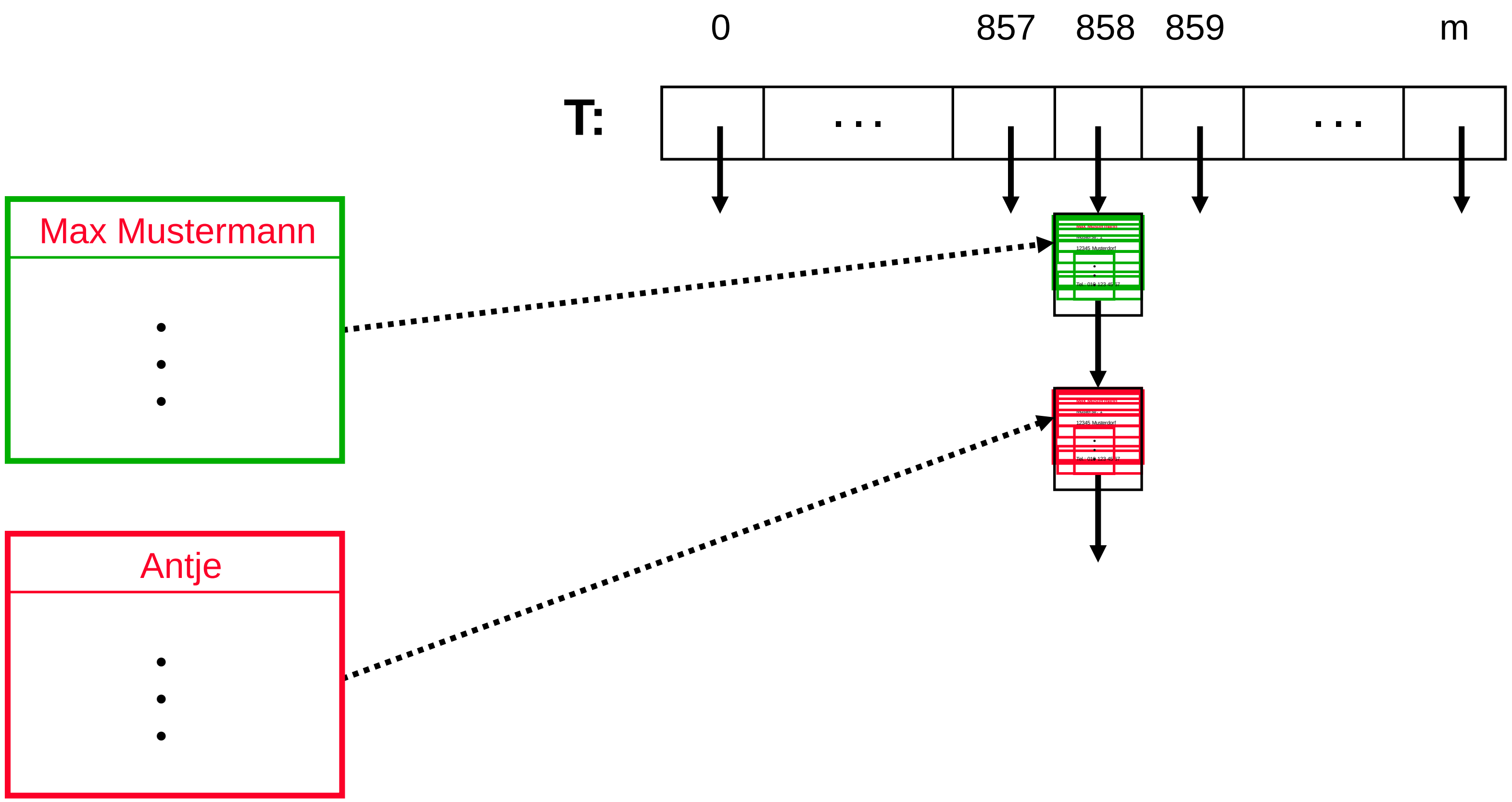


Does anybody have a better idea?

2) Where to store the data record in the hash table?

Problem: How to handle collisions?

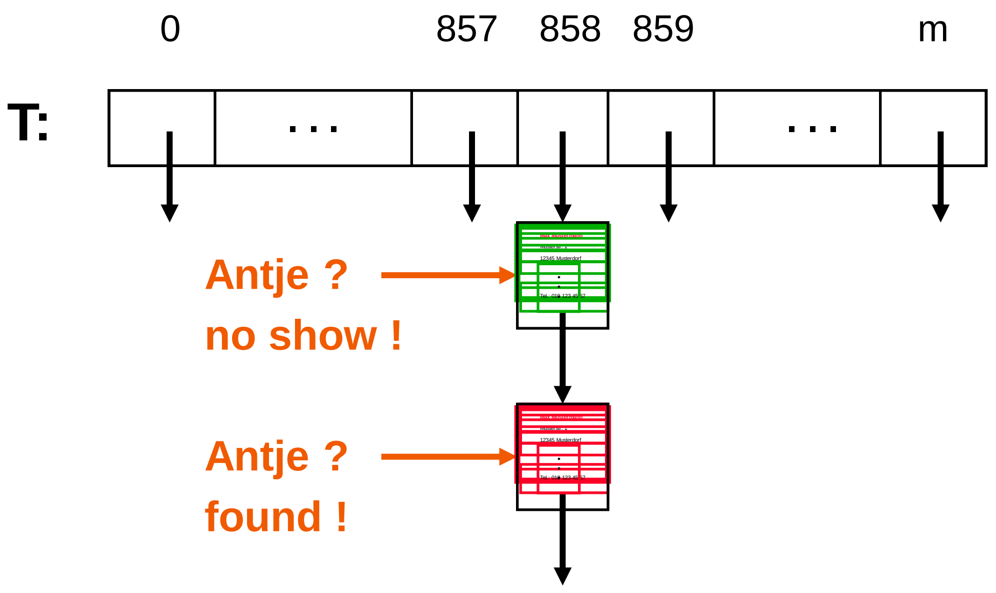
Solution: **T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.**



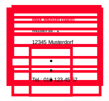
2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Solution: T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.



Search: **Antje ?**



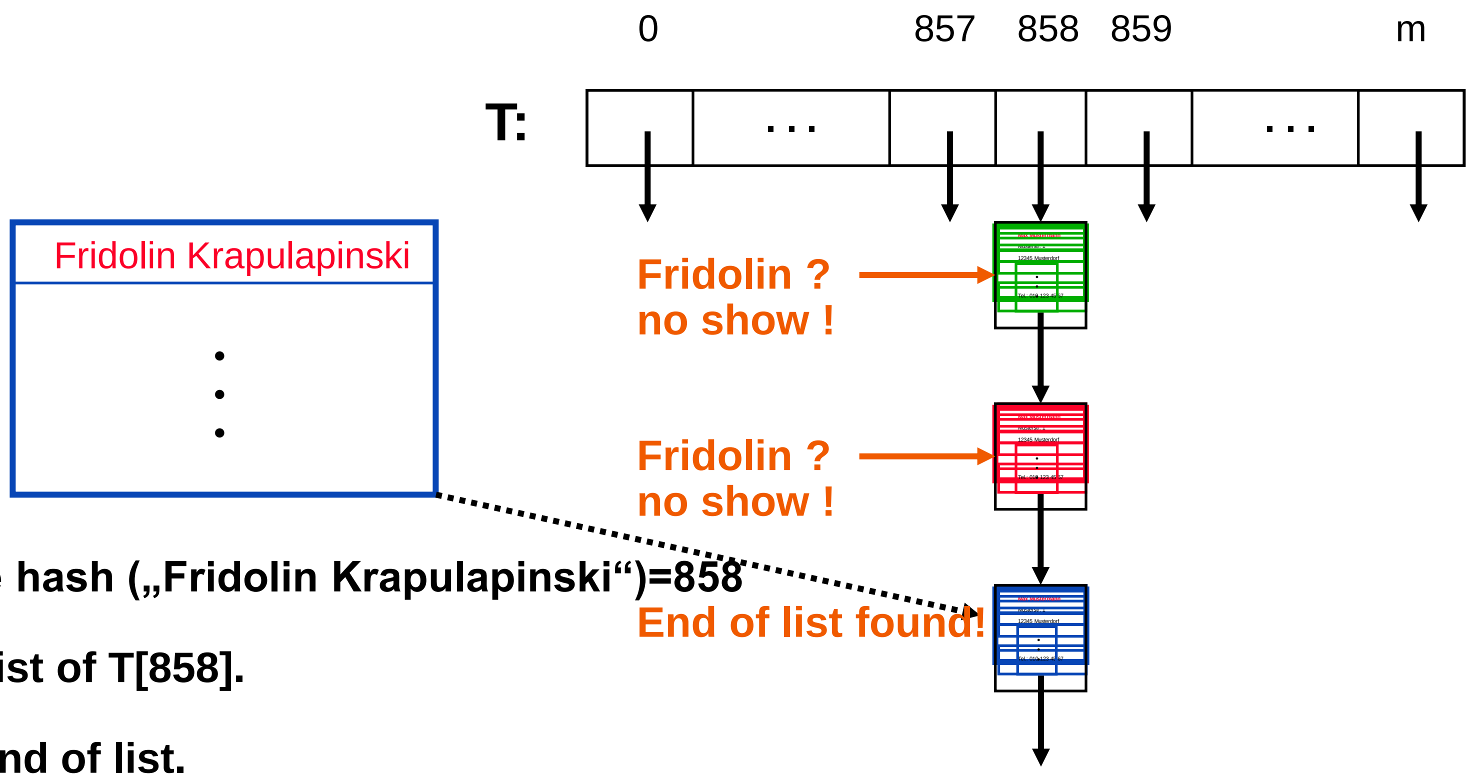
1) Determine hash („Antje“) = 858

2) Traverse list of T[858]

2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Solution: T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.



1) Determine hash („Fridolin Krapulapinski“)=858

2) Traverse list of T[858].

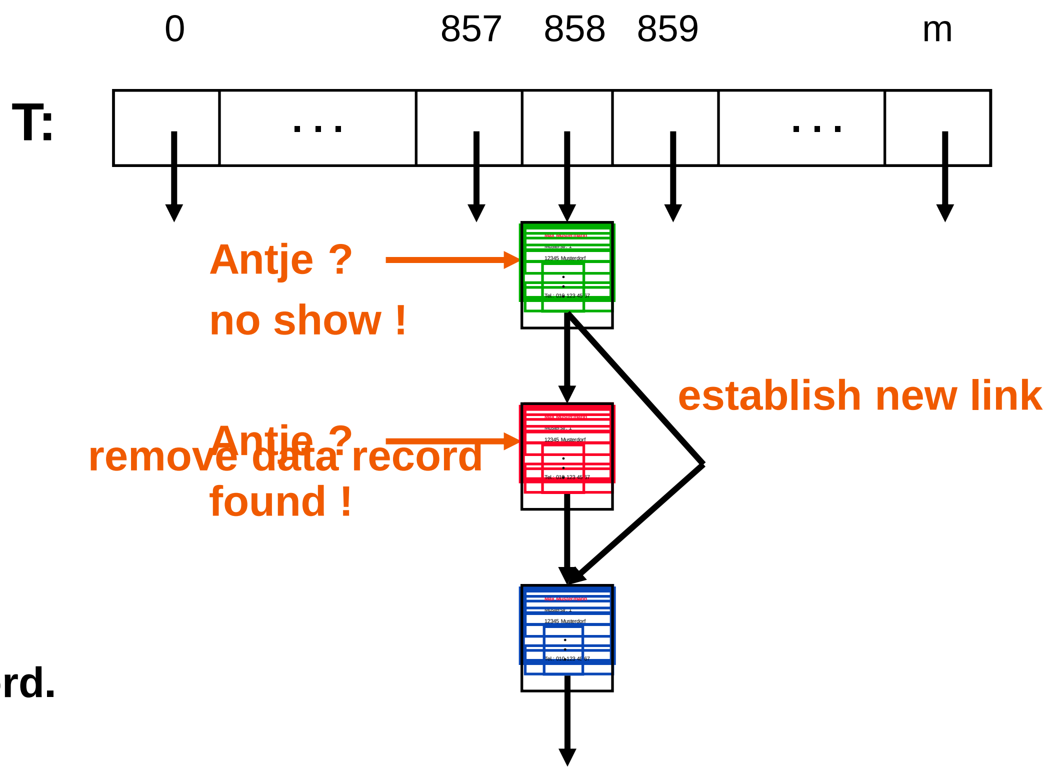
3) Insert at end of list.

2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Solution: T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.

Delete: **Antje**



1) Determine hash („Antje“)=858

2) Traverse list of T[858].

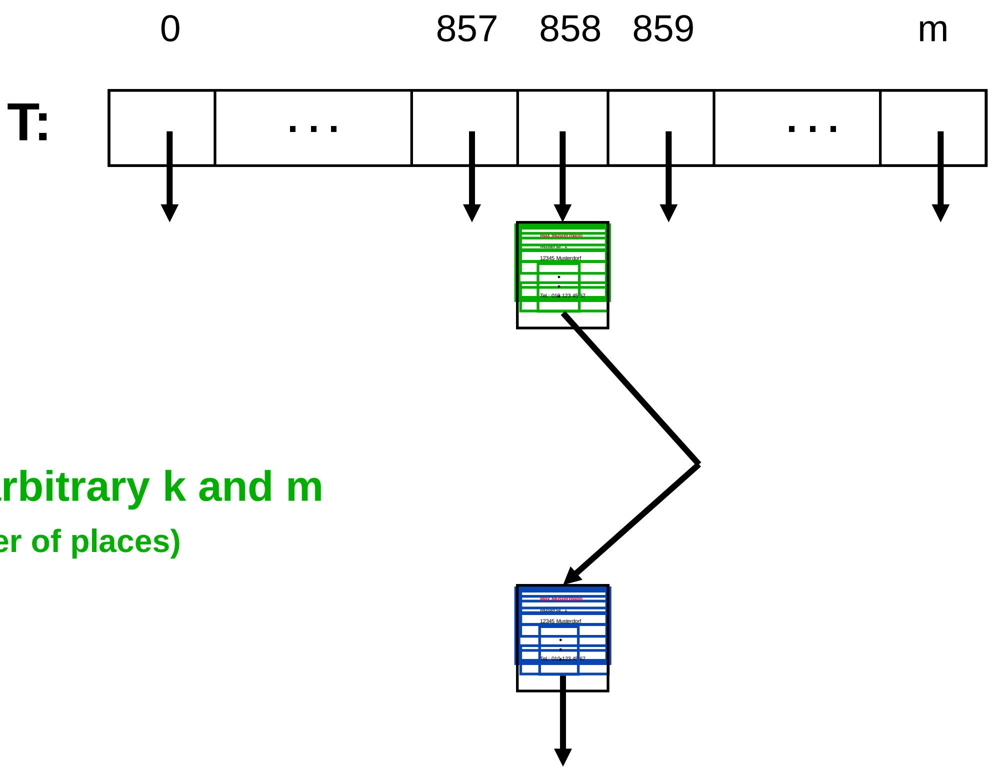
3) Remove the respective data record.

2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Solution: **T[i] contains pointers to linked lists of those data records whose keys have the same hash number i.**

(closed hashing)



Evaluating the presented method:

- **easy to implement**
- **implements hashing for arbitrary k and m**
  (k: number of keys, m: number of places)

objection:

- **waste of storage space !**

2) Where to store the data record in the hash table?

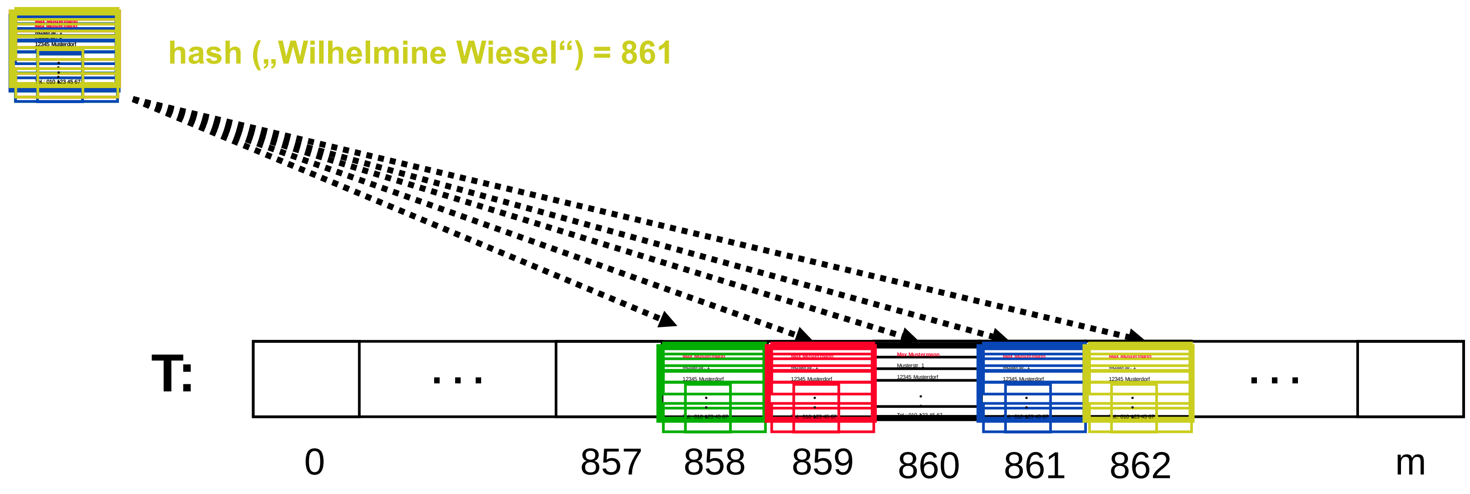
Problem: How to handle collisions?

Alternative solution: *(open hashing)*

Search for other free space in hash table: Proceed from T[i] according to a certain rule until free space is found

$\downarrow$ *probing rule*

Example for probing rule: **move right by one**



2) Where to store the data record in the hash table?

Problem: How to handle collisions?

Alternative solution: ***(open hashing)***

Search for other free space in hash table: Proceed from T[i] according to a certain rule until free space is found

$\downarrow$

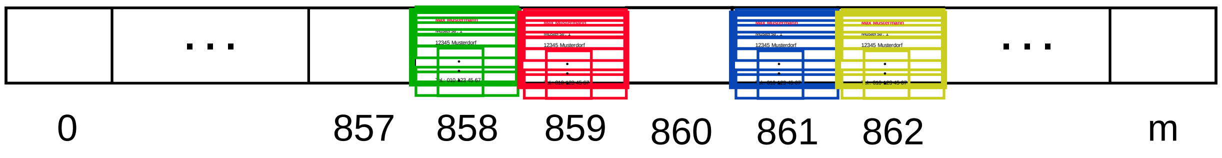
probing rule

Other methods for probing rules:

1. **move by quadratically increasing distances**
2. **move according to a second hash function** ***(double hashing)***
3. **lots more of rules in literature and practice**

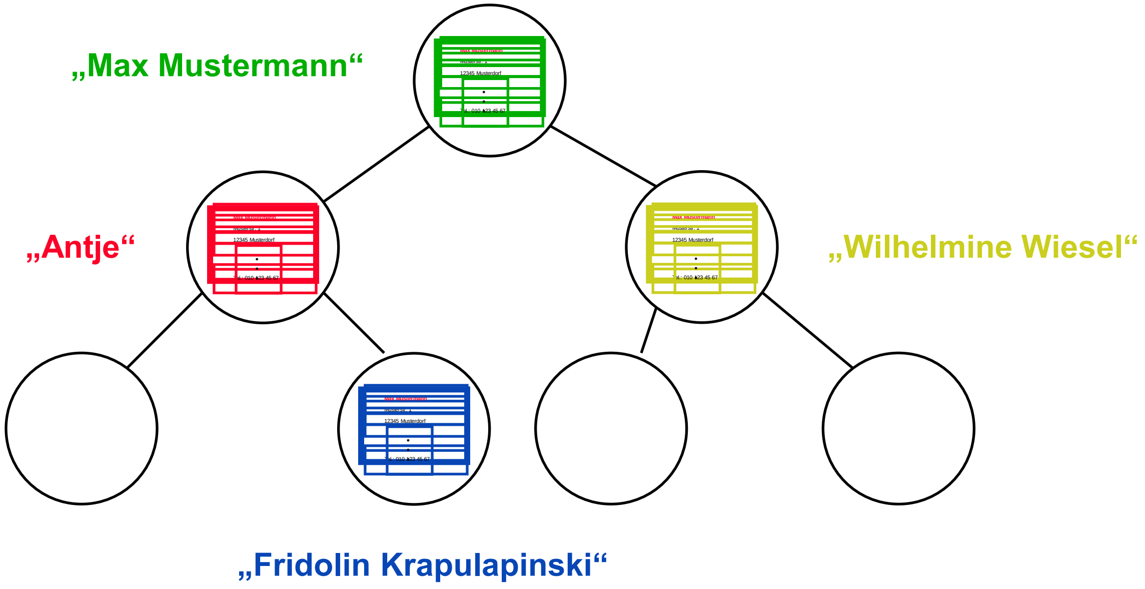
Compare with other techniques

Hash tables



What is better?

Search trees



Compare with other techniques

m = 1000
n = 500
n = 1000
n = 2000
n = 1 000 000

	Search trees (containing n entries)		**Hash tables** (containing n entries and m hash places)		
Storage	**O(n)**	≈ 500 ≈ 1000 ≈ 2000	**O(m+n)**	≈ 1500 ≈ 2000 ≈ 3000	**improvement by open hashing**
avarage run time of one operation (search / insert / delete)	**O(log n)**	≈ 9 ≈ 10 ≈ 11 ≈ 20	**O(n/m)**	≈ 1,2 ≈ 1,3 ≈ 2,1 ≈ 1000	
Applicability	**for arbitrarily many data**		**only for constant number of data (n ≈ m)**		**improvement by dynamic hashing**
Recommendation of use	**for frequent insert and delete**		**for frequent search**		

Algorithmics 3

Implementation of dictionaries

A dictionary is a data structure for elements comparable by a key implementing the functions search (key), insert (key, newdata) and delete (key)

Summarizing hashing

data type: Indexed array with m positions

Principle of operation:

- There is a hash function h: Keys $\rightarrow \{0,\dots,m-1\}$
- Each element is stored at h(k),
  as long as this position is still free (where k is the element's key)
- If position h(k) is occupied,
  a collision handling must be performed (different strategies available)

All 3 dictionary functions:	run time $\Theta(n)$ w.c. and $\Theta(n/m)$ a.c.
	$\Rightarrow$ for $n \in O(m)$: run time $\Theta(1)$ a.c.

References:

Cormen, ch. 11

Algorithmics 3

Implementation of dictionaries

A dictionary is a data structure for elements comparable by a key implementing the functions search (key), insert (key, newdata) and delete (key)

Summarizing hashing: Strategies for collision handling

data type: Indexed array with m positions

Hash lists

- At position h(k), there is a pointer to a linked lists instead of the data record .
- All data to be mapped to h(k) will be inserted sequentially into the linked list.

Open hashing

- If position h(k) is occupied,
  a special probing rule determine a different position.
- There are different strategies for probing rules.
- If all positions are occupied, the array must be enhanced
  and the hash function must be adapted (**rehashing**)

References:

Cormen, ch. 11

Algorithmics 3

Implementation of dictionaries

A dictionary is a data structure for elements comparable by a key implementing the functions search (key), insert (key, newdata) and delete (key)

Summarizing search trees:

data type: pair (data, list of children trees) $\longleftarrow$ nodes

Principle of operation:

- Each operation inspects the data of the node where it is invoked.
- If the operation may not be executed directly at node, it will be passed to one of the children. The choice to which child will be decided locally in the node.
- The search tree bares invariants that must be maintained (e.g. property that each node has got exactly 2 children)
- The maintenance of the invariants may require additional operations for insert and delete.

Each operation must be performed in constant time per node.

All 3 dictionary functions run time $\Theta(h)$ $\longleftarrow$ h is height of search tree

h is between $\Omega(\log n)$ and $O(n)$ w.c., $\Theta(\log n)$ a.c.

$\uparrow$

different ways of considering a.c.

References:

Cormen, ch. 12, Skript Alt, S. 40-41 (in German)

Algorithmics

Sebastian Iwanowski
FH Wedel

3. Solutions for the dictionary problem
3.2 (2,3)-trees as example for an optimal worst case behaviour tree

Algorithmics 3

Implementation of dictionaries

A dictionary is a data structure for elements comparable by a key implementing the functions search (key), insert (key, newdata) and delete (key)

3.2 (2,3)-trees

Data structure: Search tree with the following properties:

i. All data is stored in the leaves which are all on equal level.
ii. All inner nodes have 2 or 3 children (thus, a (2,3)-tree is not binary!)
iii. For directions, the inner nodes contain for each child the greatest key of the subtree rooted in the respective child.

A (2,3)-tree is height balanced (due to i.)

=> for all 3 dictionary operations: same run time for w.c. and a.c.

due to ii) (+ update of a node in constant time): run time Θ(log n) w.c. and a.c.

References:

Skript Alt S. 44 – 51 (Kap. 3.1.5) in German
Levitin, ch. 6.3 AHU (in balanced tree section)

Algorithmics 3

Implementation of the core functions of a (2,3)-tree

search23 (returns reference to node containing key data)

1. If node is a leaf:
   If node data contains key, return root, else null.
2. Else: Determine child where search has to continue:
   This is the child whose tree contains the smallest key greater or equal to searched key.
3. If such a child exists: Invoke search23 for this child recursively.
4. Else: Return null (key is evidently not found in the tree).

Algorithmics 3

Implementation of the core functions of a (2,3)-tree

insert23 (returning root of a (2,3)-tree containing newdata)

1. If root is an inner node (otherwise special handling necessary):
   Determine child root in which newdata has to be inserted:
   This is the child whose tree contains the smallest key greater or equal to searched key (if existing)
   or the child with greatest key (otherwise).
2. Invoke insert23rec for this child.
3. If only one root is returned, adopt this as new child (instead of the former one) and return own root.
4. If two roots are returned and only one more child exists, adopt both roots as new children and return own root.
5. If two roots are returned and two more other children exist (resulting in four new children),
   split root node into 2 nodes, generate a new root node and attach the nodes of the split as children.
   Return the new root. (*Remark:This increases the depth by 1*)

insert23rec (returns 1 or 2 roots of (2,3)-trees one of them containing newdata)

1. If root is a leaf: Generate another leaf for newdata and return the reference to this and the old root (resulting in 2 roots).
2. Else: Determine child root in which newdata has to be inserted.
3. Invoke insert23rec for this child root recursively.
4. If only one root is returned, adopt this as new child (instead of the former one) and return own root.
5. If two roots are returned and only one more child exists, adopt both roots as new children and return own root.
6. If two roots are returned and two more other children exist (resulting in four new children),
   split root node into 2 nodes, distribute the children such that each node has 2 of them and return the references to the nodes of the split.

Implementation of the core functions of a (2,3)-tree

delete23 **(returns root of a (2,3)-tree not containing any data with searched key).**

1. If root is an inner node, invoke delete23rec for root (otherwise special handling necessary).
2. If only one child is returned, return this child as new root.
   (***Remark: This decreases the depth by 1***)
3. Else: Generate a new node with the returned nodes as children and return reference to this node.

delete23rec **(returns 1 – 3 children of invoked node not containing any data with searched key).**

1. If the children of root are leaves, return those children not containing data with the searched key (1, 2, or 3).
2. Else: Determine child root from which data has to be removed:
   This is the child whose tree contains the smallest key greater of equal to searched key (if existing).
3. If such a child does not exist, stop recursion and return the former children as new children (2 or 3).
4. Else: Invoke delete23rec for the determined child recursively.
5. If this invocation gets at least 2 children in return, adopt the old child with these children and change nothing.
6. Else (only 1 child returned):
   If there are only 2 other grandchildren, generate one child with the 3 remaining grandchildren as children and return just this child (only 1).
   Otherwise distribute the 4 to 8 remaining grandchildren to 2 or 3 new children and return these children:
   The new grandchildren distribution will be (2,2), (2,3), (3,3), (2,2,3), (2,3,3)
   depending on how many grandchildren are left.

Algorithmics

Sebastian Iwanowski
FH Wedel

3. Solutions for the dictionary problem
3.3 Other optimal worst case methods for search trees

Algorithmics 3

3.3 Other optimal worst-case methods for search trees

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

(a,b)-tree: Generalisation of (2,3)-trees ($a \geq 2$, $b \geq 2a-1$)

Data structure: search tree with the following properties:

i. All data is stored in the leaves which are all on equal level.
ii. All inner nodes (except for the root) have at least a and at most b children.
    The root has at least 2 and at most b children.
iii. For directions, the inner nodes contain for each child the greatest key of the subtree rooted in the respective child.

An (a,b)-tree is height balanced (due to i.)

=> for all 3 dictionary operations: same run time for w.c. and a.c.

due to ii) (+ update of a node in constant time): run time $\Theta(\log n)$ w.c. and a.c.

References:

Skript Alt S. 44 – 51 (Kap. 3.1.5) in German
Mehlhorn ch. 7

Algorithmics 3

3.3 Other optimal worst-case methods for search trees

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

AVL tree

Data structure: binary search tree with the following property:

For each node, the height of the children subtrees differs by at most 1.

The maintenance of this invariant is guaranteed by rotations and double rotations of nodes which can be executed during insert and delete in constant time per node.

All 3 dictionary operations:	same run time Θ(log n) w.c. and a.c.
	Proof is complicated (via Fibonacci estimations).

References:

Skript Alt S. 41 – 44 (Kap. 3.1.4)	contain both exact functionality and run time proof,
Knuth ch. 6.2.3	not relevant for exam

Algorithmics 3

3.3 Other optimal worst-case methods for search trees

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

Red-black tree

Data structure: binary search tree, each node being red or black, with the following properties:

i. The tree is full (each inner vertex has got two children) $^{*)}$.
ii. Red nodes have black children.
iii. For each node n, each path from n to any leaf contains the same number b(n) of black nodes.

The maintenance of this invariant is guaranteed by rotations and double rotations of nodes which can be executed during insert and delete in constant time per node.

All 3 dictionary operations: same run time Θ(log n) w.c. and a.c.

Proof is shown in class.

$^{*)}$ In standard references this property is replaced by: Leaves must be black. NIL leaves are mandatory at each level.

References:

Skript Alt S. 54 (Kap. 3.1.7) in German
Cormen ch.13 (including algorithmic details and run time estimation)

Algorithmics 3

3.3 Other optimal worst-case methods for search trees

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

Glossary:

A binary tree is called full (or saturated), if all inner nodes have exactly 2 children.
A binary tree is called complete, if all leaves are located on the lowest levels (completely filled from left to right)
A binary tree is called perfect, if all leaves are located on the same level.

Note: A red-black tree must be full while an AVL tree need not.

Red-black tree

Prop.:: Red-black trees may be easily transformed to (2,4)-trees and vice versa.

➔: Each black node is combined with its red children.
The children of the red children and the possibly existing other black child of the original node become the children of the combined node.

🡐:
i. Any node with 2 children becomes a black node with 2 black children.
ii. Any node with 3 children a_1, a_2, a_3 becomes a black node with a black child a_1 and a new red child having a_2, a_3 as black children.
iii. Any node with 4 children becomes a black node with 2 new red children, each having 2 of the former children as own black children.

References: Skript Alt S. 54 (Kap. 3.1.7) in German

Algorithmics 3

3.3 Other optimal worst-case methods for search trees

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

B trees: Special (a,b)-trees

A B tree is equivalent to a (t,2t)-tree (t ≥ 2) "filled at least by half"

A B* tree (Knuth) is defined as ((2m-1)/3,m)-tree "filled at least by 2/3"
where the root may have between 2 and 2·(2m-1)/3 + 1 children.

Practical application: rapid disc access

All 3 dictionary operations: same run time Θ(log n) for w.c. and a.c.

References:

Cormen ch. 18 (B trees)
Knuth ch. 6.2.4

Algorithmics 3

3.3 Other optimal worst-case methods for search trees

A dictionary is a data structure for elements comparable by a key implementing the functions member (key), insert (key, newdata) and delete (key)

Trie tree(from re**trie**val) for strings over a k element alphabet

Data structure: k-ary tree having the following properties:

i. The root is empty.
ii. Each other node contains a letter being used in some string stored.
For each string using this node there is a child containing the next letter of the string.
iii. Each node has a flag indicating if this is the end of a string or not.

All 3 dictionary operations Run time: Θ(length of string) w.c. and a.c. (clear)

Run time for a search in a tree storing n strings:
$\log_k(n)$ a.c. (Knuth, summary)

Space for a tree storing n strings: n/ln k a.c. (Knuth, summary)

References:

Skript Alt S. 54 – 56 (Kap. 3.1.8) in German
Knuth ch. 6.3 (Digital Searching) with several improvements, not relevant for exam
Seminar talk Nr. 8 in https://intern.fh-wedel.de/mitarbeiter/iw/lv/ws-2019/seminar/ (in German)

Algorithmics

Sebastian Iwanowski
FH Wedel

3. Solutions for the dictionary problem
3.4 Optimal binary search trees

Algorithmics 3

3.4 Optimal binary search trees

Problem:

i. Let $S = \{a_1, a_2, \ldots, a_n\}$ be a linearily ordered set with predetermined probablities p_i for the occurrence of a_i und q_i für the occurrence of an element a in between: $a_i < a < a_{i+1}$.

ii. Construct a binary search tree which minimizes the expected response time (i.e. number of comparisons with elements a_i).

Required tree properties:

The tree should not only find the position of elements contained in the given dictionary, but also locate the position where new elements would be placed:
Inner nodes correspond to elements contained, leaves correspond to elements in between

Solution by the algorithm of Bellman (1957)

Time for the construction of the search tree: $O(n^3)$ (easy to prove)

Improvement: $O(n^2)$

References:

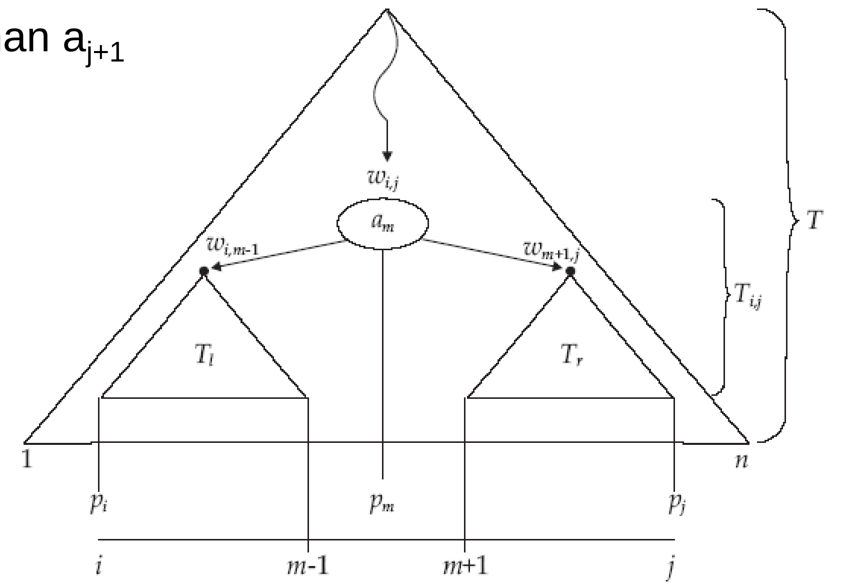
Skript Alt S. 65 – 70 (ch. 3.3) in German: Other references are less clear

Cormen 15.5 (ch. Dynamic Programming)

Knuth 6.2.2 (Binary Tree Searching)

Bellman's Algorithm for optimal binary search trees:

$T_{i,j}$: subtree for search items greater than a_{i-1} and less than a_{j+1}



Special cases:

$T_{i,i}$: subtree for search items greater than a_{i-1} and less than a_{i+1}.
This tree consist of one node comparing with a_i

$T_{i,i-1}$: subtree for search items greater than a_{i-1} and less than a_i.
This tree is empty and corresponds to a leaf.

$T_{i,n}$: subtree for search items greater than a_{i-1}

$T_{1,j}$: subtree for search items less than a_{j+1}

$T_{1,n}$: tree for all search items

Notation: Skript Alt

Bellman's Algorithm for optimal binary search trees:

$T_{i,j}$: subtree for search items greater than a_{i-1} and less than a_{j+1}

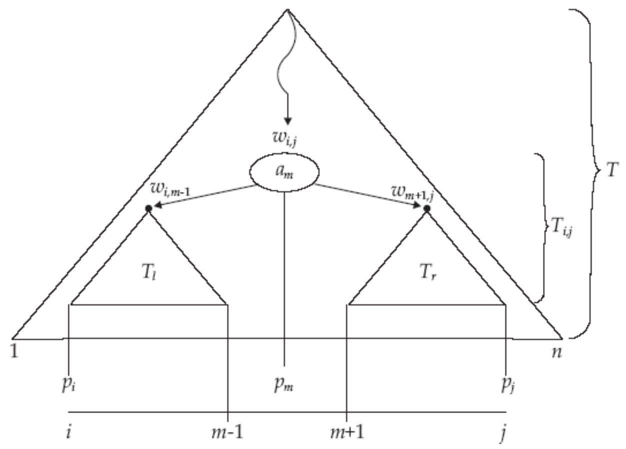
$r_{i,j}$: index m of the root of $T_{i,j}$: The item to be compared with is a_m

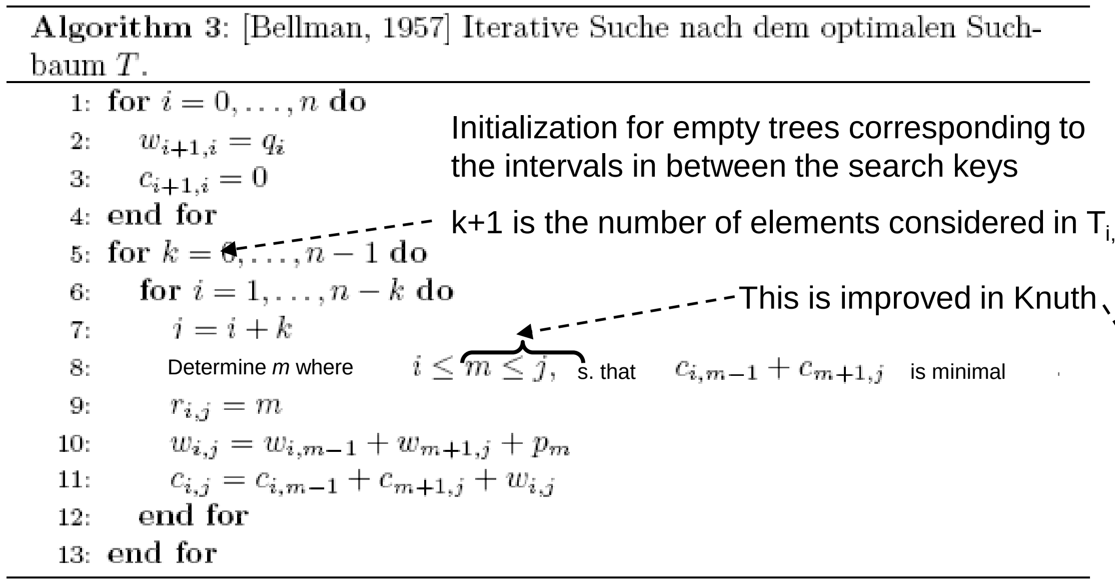
$P(T_{i,j})$: expected costs for $T_{i,j}$ if $T_{i,j}$ is chosen

$w_{i,j}$: probability that $T_{i,j}$ is chosen

$c_{i,j}$: expected costs for $T_{i,j}$ if no precondition is known

Lemma 3.3.5: If $T_{i,j}$ is optimal, then each subtree is also optimal.




Algorithm 3: [Bellman, 1957] Iterative Suche nach dem optimalen Suchbaum T.

```
1:  for $i = 0, \ldots, n$ do
2:      $w_{i+1,i} = q_i$
3:      $c_{i+1,i} = 0$
4:  end for
5:  for $k = 0, \ldots, n-1$ do
6:      for $i = 1, \ldots, n-k$ do
7:          $j = i + k$
8:          Determine $m$ where $i \le \overbrace{m \le j}$, s. that $c_{i,m-1} + c_{m+1,j}$ is minimal
9:          $r_{i,j} = m$
10:         $w_{i,j} = w_{i,m-1} + w_{m+1,j} + p_m$
11:         $c_{i,j} = c_{i,m-1} + c_{m+1,j} + w_{i,j}$
12:     end for
13: end for
```

Initialization for empty trees corresponding to the intervals in between the search keys

k+1 is the number of elements considered in $T_{i,j}$

This is improved in Knuth

Assertion 3.3.6: does not depend on m

$w_{i,j} = w_{i,m-1} + p_m + w_{m+1,,j}$

$$\begin{aligned} c_{i,j} &= w_{i,j} \cdot P(T_{i,j}) \\ &= w_{i,j} \cdot (1 + P(T_{i,m-1}) + P(T_{m+1,j})) \\ &= w_{i,j} + c_{i,m-1} + c_{m+1,,j} \end{aligned}$$

depends on m

Lemma 3.3.7:

$r_{i,j-1} \le r_{i,j} \le r_{i+1,j}$

Notation: Skript Alt

Example from Skript Alt:

Resulting construction of search tree:

p_1=0　　p_2=0,1　　p_3=0,2　　p_4=0,2

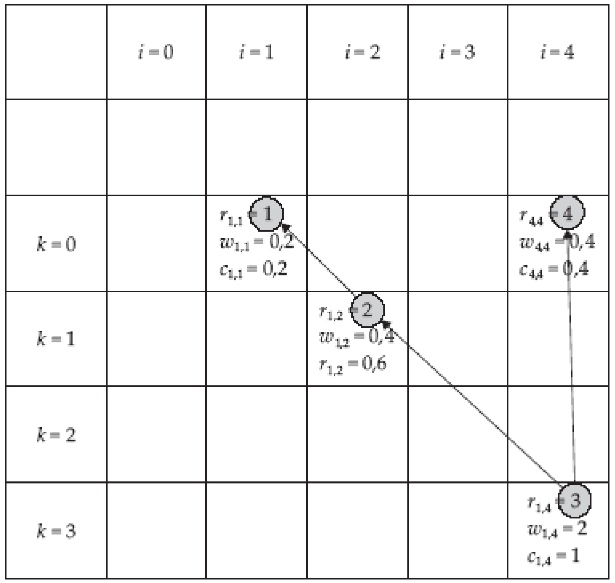
q_0=0,1　　q_1=0,1　　q_2=0,1　　q_3=0,1　　q_4=0,1

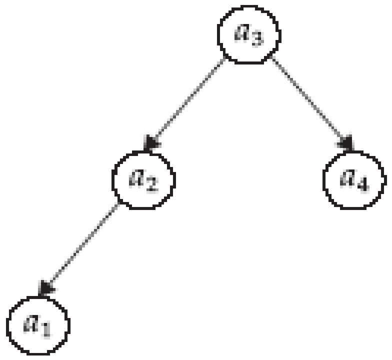
Short notation:

(0,1) – **1**(0) – (0,1) – **2**(0,1) – (0,1) – **3**(0,2) – (0,1) – **4**(0,2) – (0,1)

i	0	1	2	3	4
$Init$	$w_{1,0} = 0,1$ $c_{1,0} = 0$	$w_{2,1} = 0,1$ $c_{2,1} = 0$	$w_{3,2} = 0,1$ $c_{3,2} = 0$	$w_{4,3} = 0,1$ $c_{4,3} = 0$	$w_{5,4} = 0,1$ $c_{5,4} = 0$
$k = 0$		$r_{1,1} = 1$ $w_{1,1} = 0,2$ $c_{1,1} = 0,2$	$r_{2,2} = 2$ $w_{2,2} = 0,3$ $c_{2,2} = 0,3$	$r_{3,3} = 3$ $w_{3,3} = 0,4$ $c_{3,3} = 0,4$	$r_{4,4} = 4$ $w_{4,4} = 0,4$ $c_{4,4} = 0,4$
$k = 1$		$r_{1,2} = 2$ $w_{1,2} = 0,4$ $c_{1,2} = 0,6$	$r_{2,3} = 3$ $w_{2,3} = 0,6$ $c_{2,3} = 0,9$	$r_{3,4} = 3$ $w_{3,4} = 0,7$ $c_{3,4} = 1,1$	
$k = 2$		$r_{1,3} = 2$ $w_{1,3} = 0,7$ $c_{1,3} = 1,3$	$r_{2,4} = 3$ $w_{2,4} = 0,9$ $c_{2,4} = 1,6$		
$k = 3$		$r_{1,4} = 3$ $w_{1,4} = 1$ $c_{1,4} = 2$			

Tabelle 3.1: Tabelle zur Speicherung der Berechnungen des Algorithmus





Notation: Skript Alt

Algorithmics

Sebastian Iwanowski
FH Wedel

4. Graph algorithms
4.1 Minimum spanning trees as motivation for basic algorithms

Algorithmics 4

4.1 Minimum spanning trees

Kruskal‘s Algorithm (simple variant):

Construction of a minimum spanning tree for an arbitrary graph G:

- Start with an empty forest F consisting of no edge
- Repeat for all edges e_1, e_2, ..., e_m of G (edges are in sorted order):

```
        Check if $e_i$ may be inserted into F
                such that F is still without circles;
        If so, insert $e_i$ into F;
until F consists of n-1 edges (let n be the number of vertices of G).
```

Theorem: Thus constructed forest F is a minimum spanning tree of G

Proof: see next slide

How to do this smarter?

Time complexity: O(m log m + nm) (nm due to determination of connectivity component)

References for catching up and delving into:

Skript Diskrete Mathematik 6, Folien 2,3,4,8,11,12,13 (graph theoretic basics)
Turau Kap. 2.4 (Grundlagen), 3.6.1 (Kruskal)
Cormen ch. 23 (Minimum spanning trees)

Algorithmics 4

4.1 Minimum spanning trees

Proposition (implies correctness of Kruskal‘s algorithm, why?):

For each edge set $\{e_1, e_2, ..., e_j\}$ which is successively constructed by Kruskal‘s algorithm there is a minimum spanning tree T_j of G containing this edge set.

Proof by mathematical induction over j

Inductive step:

The assumption may hold for an edge set E_j consisting of j edges, i.e. there is a minimum spanning tree T_j where $E_j \subseteq T_j$.
Let e_{j+1} be the next edge chosen by Kruskal. If $e_{j+1} \in T_j$, choose $T_{j+1} = T_j$.
Otherwise there must be a circle in $T_j \cup \{e_{j+1}\}$ containing e_{j+1}. At least one of the other edges e_0 of this circle should not be contained in E_j (otherwise, Kruskal would not have chosen e_{j+1} because E_j would not have been free of circles). Replace this edge e_0 by edge e_{j+1} => spanning tree T_{j+1} containing $E_j \cup \{e_{j+1}\}$.
$c(e_0) \geq c(e_{j+1})$, because otherwise Kruskal would have chosen e_0 before e_{j+1}.
Thus, T_{j+1} must be minimum as well as T_j.

Deutschsprachige Referenzen zum Nacharbeiten und Vertiefen:

Skript Alt, Lemma 4.3.2 (S. 76): Beweisskizze eines verwandten Satzes
Turau, Kapitel 3.6.1: genauer Beweis des Satzes wie oben (inkl. Induktionsverankerung)
Lang: Skript Berechenbarkeit und Komplexität, Kap. 4.2.3 (Greedy-Algorithmen für Matroide)

Algorithmics 4

4.1 Basic algorithms for graph theory

Union-Find-Structure

In general: works on sets of disjoint sets, administrates the change of set partitions:
implements the efficient identification of the set containing a given element and the efficient union of sets

Graph theoretic application: efficient location and union of connectivity components

Other applications: segmentation of images, unification of expressions in compiling

O(log n)	`Find (v)`	returns a unique reference node of the connectivity component of v.
O(1)	`Union (v,w)`	unifies the connectivity components of v and w after reference node has been determined

Data representation:

Array of nodes: The contents are pairs of the form (index of parent, height of subtree)
The subtree height is only important for root elements, for other elements the height is not considered anymore.

With path compression:

Expected time complexity of Find is in O(log*n)

References:

Skript Alt, Kap. 3.2 (p. 56 ff.), Cormen ch. 21 (Data structures for disjoint sets)

Algorithmics 4

4.1 Basic algorithms for graph theory

Heap

Efficient management of a priority queue

Invariants:
1) A heap is a complete binary tree (elements may be missing only in the last depth level).
2) The keys of the children of each node are not less than the key of each node.

O(log n)	**`DeleteMin()`**	deletes the minimum element of the heap.
O(log n)	**`Insert (v)`**	inserts an arbitrary new element into the heap.
O(1)	**`SearchMin()`**	finds the minimum element of the heap.

Data representation:

Array of the heap nodes:

The contents are the contents of the heap nodes.
The children of the node with index i are the nodes with indices 2i und 2i+1
(assuming that the array starts with index 1)

References:

Cormen, ch. 6 (Heapsort)

Algorithmics 4

4.1 Minimum spanning trees

Kruskal‘s Algorithm (efficient variant):

Construction of a minimum spanning tree for an arbitrary graph G:

- Start with an empty forest F consisting of no edge
- Start with a **union-find-structure** in which each vertex has its own connectivity component
- Insert all edges into a **heap**

- While F consists of less than n-1 edges:
  Search and delete the minimum element e_{min} from the heap;
  Check if the vertices v and w incident with e_{min} are in the same connectvity component
  If not: Insert e_{min} into F and unify the connectivity components of v and w.

Time complexity: O(m log m) (m is the number of edges in G)

References:

Cormen, ch. 23.2 (Algorithms of Kruskal and Prim)

Algorithmics

Sebastian Iwanowski
FH Wedel

4. Graph algorithms
4.2 Shortest paths

Algorithmics 4

SSSP: Single Source Shortest Path

Find the shortest paths from a source s to all other nodes

Remark: For the problem to find the shortest path between two given nodes there is no better algorithm known than those for SSSP, and those have not been proved being optimal even for SSSP.

Algorithm of Dijkstra for SSSP: (for graphs G with nonnegative edge costs only)

```
•   Initialize the node set Done by s;
    Initialize the node set Undone by all other nodes of graph G;
    For all nodes v of the graph G:
        Let label (v) := length of edge from s to v (∞ if no edge is existing, 0 if v = s);

•   While Undone is not empty:
      Search and delete the node v from Undone with minimal label;
      Insert v into Done;
      Update all neighbors n of v that are in Undone:
        If label (n) > label (v) + length of edge between v and n:
          Replace label (n) by that number;
          Let v be the predecessor of n.
```

Theorem: The labels of nodes v in **Done** are always the shortest path length from s to v and the shortest path is the shortest path from s to the predecessor of v followed by the edge from the predecessor to v.

Proof: Mathematical induction by number of iterations.

Algorithmics 4

SSSP: Single Source Shortest Path

Algorithm of Dijkstra for SSSP: (for graphs G with nonnegative edge costs only)

Theorem of correctness (to be proven):

For each node v ∈ **`Done`** holds:

label (v) = d_s(v).

where label (v) is the length of a path found from s to v
and d_s(v) is the length of a shortest path from s to v

Lemma Subpath:

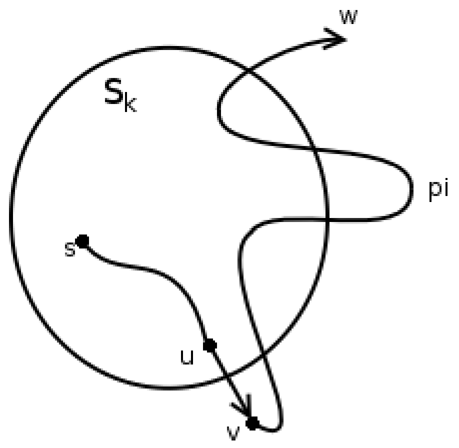
For each node u on the shortest path from s to v holds:
The subpath from s to u is the shortest path from s to u.

Proof by contraposition:
Let u be on any path from s to v and the subpath from s to u be not the shortest path from s to u.
Then the path from s to v is not the shortest path from s to v.
(Details: Exercise)

Algorithmics 4

SSSP: Single Source Shortest Path

Proof of the theorem of correctness by mathematical induction over iteration cycle i, which shifts v to `Done`:



Base case for i = 0 clear (Exercise!)

Inductive step ≤ i -> i+1:

(*) Let w be the node, which is shifted to **`Done`** in the (i+1). cycle

Assume, label (w) ≠ d_s(w) (will be used to contradict (*)).

Since label (w) = ∞ or label (w) = length of some path to w, the following holds: d_s(w) < label (w).
Let (u,v) be the first edge on the shortest path to w leaving **`Done`**, i.e. u ∈ **`Done`** and v ∉ **`Done`**.

shortest path some path

Then: $d_s(v) = \underset{\substack{\uparrow \\ \text{cf. Lemma Subpath}}}{d_s(u)} + c(u,v) = \underset{\substack{\uparrow \\ \text{Ind.ass.}}}{\text{label}(u)} + c(u,v) \geq \text{label}(v) \Rightarrow \overset{\text{shortest path}\searrow}{d_s(v)} = \overset{\swarrow\text{some path}}{\text{label}(v)}$

has been set to minimum of path via u (since u ∈ **`Done`**)
and previous value

ass. above

This holds: $\text{label}(v) = d_s(v) \leq d_s(w) < \text{label}(w)$.
Since v,w ∉ **`Done`**, the following holds: v is shifted to **`Done`** before w => contradiction to (*)

Thus: label (w) = d_s(w) q.e.d.

Algorithmics 4

SSSP: Single Source Shortest Path

Find the shortest paths from a source s to all other nodes

Remark: For the problem to find the shortest path between two given nodes there is no better algorithm known than those for SSSP, and those have not been proved being optimal even for SSSP.

Algorithm of Dijkstra for SSSP: (for graphs G with nonnegative edge costs only)

Organize the edge costs in a heap.

Time complexity: O((m+n)log n) (by direct inspection of the nodes)

for arbitrary graphs: $O(n^2 \log n)$
for graphs with a constant number of neighbors per node: O(n log n)

Remark: Using the special structure `Fibonacci Heap` with corresponding methods for decreasing keys, a very careful analysis using amortised time arguments over all loops shows that the worst case is O(m + n log n) only. This is an improvement for dense graphs only.

References:

Skript Alt 4.4.1 (p. 79-81),
Cormen, ch. 24 (much more detailed: SSSP)

APSP: All Pairs Shortest Path

Find the shortest paths between all pairs of nodes

Trivial solution: Apply Dijkstra iteratively for all nodes as sources

Time complexity: O(n(m+n)log n) (or only O(n(m + n log n))

for arbitrary graphs: O(n^3log n) (or only O(n^3))
for graphs with a constant number of neighbors per node: O(n^2 log n)

Algorithm of Floyd-Warshall:

Let V = {1,...n}.
$d_{ij}^{(k)}$ is the length of the shortest path between i and j using in between at most nodes from {1,...k}.

Time complexity: O(n^3) (with simple implementation)

<u>Other advantage of FW to Dijkstra</u>:
FW works also for **negative** weights
(but no negative cycles).

```
1:  for $i = 1, \ldots, n$ do
2:    for $j = 1, \ldots, n$ do
3:      $d_{ij}^{(0)} = \begin{cases} c(i,j): & \text{falls } (i,j) \in E \\ \infty: & \text{sonst} \end{cases}$
4:    end for
5:  end for
6:  for $k = 1, \ldots, n$ do
7:    for $i = 1, \ldots, n$ do
8:      for $j = 1, \ldots, n$ do
9:        $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
10:     end for
11:   end for
12: end for
```

References:

Skript Alt 4.4.2, 4.4.3 (p. 81-83),
Cormen, ch. 25.2 (Floyd-Warshall)

Algorithmics 4

APSP: All Pairs Shortest Path

Find the shortest paths between all pairs of nodes

Relation to matrix multiplication:

Let V = {1,...n}.
$d_{ij}^{(k)}$ is the length of the shortest path between i und j using at most k edges.
Note: This definition is different from Floyd-Warshall's!

Theorem: Let A be the adjacency matrix.
Define the operation min instead of addition and the operation + instead of multiplication.

Then A^k stores in position (i,j) the length $d_{ij}^{(k)}$.
In particular, A^{n-1} stores in position (i,j) the length of the shortest path from i to j.

Quadratic potentiation: A^{n-1} may be computed with O(log n) matrix multiplications.

Standard matrix multiplication: 2 matrices may be multiplied with $O(n^3)$ number operations.

Conclusion for APSP: $O(n^3 \log n)$ number operations (worse than Floyd-Warshall!)

References for a deeper insight:

Cormen, ch. 25.1 (relation to matrix multiplication)

Algorithmics 4

APSP: All Pairs Shortest Path

Find the shortest paths between all pairs of nodes

Strassens‘s algorithm for matrix multiplication: Two nxn-matrices may be multiplied with O($n^{\log 7}$) operations.

Note that log 7 ≈ 2,81

Conclusion for APSP? Time complexity O($n^{\log 7}$ log n) **?**

Unfortunately, no!

Strassen‘s algorithm needs inverse functions to the additive operation.
This does not hold for the minimum operation needed in the transform from APSP to matrix multiplication.

References for a deeper insight:

Cormen, ch. 25.1 (relation to matrix multiplication), ch. 28.2 (Strassen’s algorithm)

Algorithmics

Sebastian Iwanowski
FH Wedel

4. Graph algorithms
4.3 Computation of maximum flows in s/t-networks

Algorithmics 4

4.3 Computation of maximum flows in s/t-networks

Notation

Def.: s/t-network (q/s-Netzwerk):

Complete directed graph (V,E) with nonnegative edge capacities c(e) for all edges e and a selected source vertex s (Quelle q) and a selected target vertex t (Senke s)

Def.: flow f: function $E \to \mathbb{R}$ where

- $f(e) \le c(e)$ for all edges e
- $f(u,v) = - f(v,u)$
- For all vertices $v \neq s,t$ the following holds:
  The sum of all flows from v to all neighbors is 0.

Def.: value |f| of a flow:

net flow out of s resp. net flow into t (both values must be equal)

References:

Cormen, ch. 26.1 (flow networks)
Alt, Kap. 4.5.1
Turau, Kap. 6.1 (siehe auch Ausarbeitung und Vortrag Seminararbeit Claudia Padberg)

Algorithmics 4

4.3 Computation of maximum flows in s/t-networks

Notation

Def.: Augmenting path (Erweiterungsweg) of a flow f:

Path from s to t where the following holds for each edge (u,v): $f(u,v) < c(u,v)$
The value $c(u,v) – f(u,v)$ is called the remainder capacity.

Note: $f(u,v)$ may be negative which means that $f(v,u) > 0$.
In this case, $f(v,u) = c(v,u)$ is permitted.

Def.: Residual network (Restegraph, Restnetz) G_f:

For each edge (u,v) with positive remainder capacity in G, insert an edge $(u,v) \in G_f$ where the capacity is equal to that remainder capacity.

For each edge (u,v) with positive flow f(u,v) in G, insert an edge $(v,u) \in G_f$ where $c(v,u)= f(u,v)$

Prop. 1: A path p is an augmenting path in G $\Leftrightarrow$ p is a directed path from s to t in G_f

Prop. 2: A flow f may be increased by the *residual flow* (Restfluss) whose value is the minimum capacity of a directed path from s to t in G_f.

References:

Cormen, ch. 26.2 (Ford-Fulkerson method)
Alt, Kap. 4.5.2
Turau, Kap. 6.1, 6.3 (Restegraph) (siehe auch Ausarbeitung und Vortrag Seminararbeit C. Padberg)

Algorithmics 4

4.3 Computation of maximum flows in s/t-networks

Notation

Def.: s/t-cut (X,Y) (q/s-Schnitt):

Partition of vertices in G such that $s \in X$ und $t \in Y$

Def.: capacity c(X,Y) of an s/t-cut:

Sum of all capacities c(u,v) where $u \in X$ and $v \in Y$

Def.: flow f(X,Y) of an s/t-cut:

Sum of all flows f(u,v) where $u \in X$ and $v \in Y$

Note: If there is a backward flow from v to u, this means that f(u,v) is negative.
In this case, the flow value f(v,u) must be subtracted.

Prop. 1: For each s/t-cut (X,Y) and any given flow f the following holds: $|f| = f(X,Y)$

Prop. 2: $|f| \leq \min \{c(X,Y);\ (X,Y) \text{ is s/t-cut}\}$

References:

Cormen, ch. 26.2 (Ford-Fulkerson method)
Turau, Kap. 6.1 (siehe auch Ausarbeitung und Vortrag Seminararbeit Claudia Padberg)

Algorithmics 4

4.3 Computation of maximum flows in s/t-networks

Max-flow min-cut theorem (Ford-Fulkerson theorem)

The following propositions are equivalent:

- f is a maximum flow in G
- There is no augmenting path for f in G
- There is an s/t-cut (X,Y) where |f| = c(X,Y)

Proof:

Circular argument:

1) => 2) trivial
2) => 3) will be shown in class (according to Cormen)
3) => 1) follows by Prop.2 of last slide

References:

Cormen, ch. 26.2 (Ford-Fulkerson method)
Turau, Kap. 6.2 (anderer Beweis)

Algorithmics 4

4.3 Computation of maximum flows in s/t-networks

Algorithm of Edmonds-Karp:

(using the notation of Skript Alt)

```
1) Initialize f by 0 for all edges.
Repeat
        2a) Compute residual graph $G_f$
        2b) Find augmenting path in $G_f$ **with breadth first search**
        3) Increase f by the residual flow of the augmenting path (Prop. 2, slide 3)
until no augmenting path exists
```

Correctness: follows by Ford-Fulkerson theorem

Time complexity: $O(nm^2)$

Outline of time complexity proof:

Each loop costs O(m) time, and there are O(nm) loop iterations because of the following:
Each augmenting path has got a **critical** edge, i.e. an edge of which the capacity is used up completely by the maximum possible augmentation.
Each edge can be critical at most O(n) times. There are m edges.

References:

Cormen, ch. 26.2 (Ford-Fulkerson method)
Alt, Kap. 4.5.4
Turau, Kap. 6.3 (mit Pseudocode) (siehe auch Seminararbeit Claudia Padberg)

Algorithmics 4

4.3 Computation of maximum flows in s/t-networks

Algorithm of Edmonds-Karp:

Details of time complexity proof:

Def.: Let $\delta_f(u,v)$ be the minimum number of edges between u and v in the residual network G_f

For a *breadth first search*, a source s and a target t, the following holds:

Lemma 1: Each path in a graph found by breadth first search starting at a source s has got the minimum number of edges.

Lemma 2: For each edge (u,v) of a path P_f in the residual network G_f found by breadth first search, the following holds: $\delta_f(s,v) = \delta_f(s,u) + 1$

Lemma 4.5.8 / 26.8: (Monotonicity) Let f_1, f_2 be two flows subsequently generated by Edmonds-Karp: Then for all vertices v holds: $\delta_{f_1}(s,v) \leq \delta_{f_2}(s,v)$

Lemma 4.5.9 / 26.9: (O(n) theorem) Each edge will be at most n/2 times a critical one.

References:

Cormen, ch. 26.2 (Ford-Fulkerson method)
Alt, Kap. 4.5.4
Turau, Kap. 6.3 (anderer Beweisaufbau und Notation)

4.3 Computation of maximum flows in s/t-networks

Algorithm of Edmonds-Karp:

Proof of Lemma 4.5.8:

Let for v the following hold: $\delta_{f_1}(s,v) > \delta_{f_2}(s,v)$. It is obvious that $v \neq s$. **(*)**

Let v be the node with property (*) having minimal distance from s in G_{f_2},
i.e. for all u with $\delta_{f_2}(s,u) < \delta_{f_2}(s,v)$, the following holds: $\delta_{f_1}(s,u) \le \delta_{f_2}(s,u)$ **(**)**

Let P_2 be the shortest path from s to v in G_{f_2}, and let u be the predecessor of v on that path.
Since $v \neq s$, such predecessor u exists. Thus, $(u,v) \in G_{f_2}$, and u satisfies (**).

Consider the following two cases:

a) $f_1(u,v) < c(u,v)$.

This implies: $(u,v) \in G_{f_1}$ => $\delta_{f_1}(s,v) \overset{\text{because v can be reached via (u,v) in } G_{f_1}}{\le} \delta_{f_1}(s,u) + 1 \overset{(**)}{\le} \delta_{f_2}(s,u) + 1 \overset{\text{because (u,v) is part of the shortest path in } G_{f_2}}{=} \delta_{f_2}(s,v)$ contradicting (*)

b) $f_1(u,v) = c(u,v)$.

This implies: $(u,v) \notin G_{f_1}$ Since $(u,v) \in G_{f_2}$, $f_2(v,u) > 0$ and (since $(u,v) \notin G_{f_1}$) $f_1(v,u) = 0$

Thus, (v,u) is part of an augmenting path which was used in order to increase f_1 to obtain f_2.

By Lemma 2, $\delta_{f_1}(s,v) = \delta_{f_1}(s,u) - 1 \underset{(**)}{\le} \delta_{f_2}(s,u) - 1 \underset{\text{Lemma 2}}{=} \delta_{f_2}(s,v) - 2 \underset{(*)}{<} \delta_{f_1}(s,v)$ contradiction !

In either case, we get a contradiction
which proves that for all v the following holds: $\delta_{f_1}(s,v) \le \delta_{f_2}(s,v)$

Algorithmics 4

4.3 Computation of maximum flows in s/t-networks

Algorithm of Edmonds-Karp:

Proof of Lemma 4.5.9:

Let (u,v) be a critical edge in an augmenting path for flow f_1.

By Lemma 2, $\delta_{f_1}(s,v) = \delta_{f_1}(s,u) + 1$ **(**)**

If (u,v) becomes a critical edge again implies:
(v,u) is in an augmenting path some time in between for a flow f_2.

Consider the following:

$$\delta_{f_2}(s,u) \underset{\text{Lemma 2}}{=} \delta_{f_2}(s,v) + 1 \underset{\text{Lemma 4.5.8}}{\geq} \delta_{f_1}(s,v) + 1 \underset{(**)}{=} \delta_{f_1}(s,u) + 2$$

Thus, the distance from u to the source s has increased by at least 2

This can happen at most n/2 times, because the distance is never greater than n. **q.e.d.**

Algorithmics 4

4.3 Computation of maximum flows in s/t-networks

Algorithm of Dinic

Notation:

Def.: Level graph L_f: **(Turau: Niveaugraph G'_f)**

Delete all edges (u,v) from G_f where $\delta_f(s,v) \leq \delta_f(s,u)$

Def.: blocking flow:

A flow where each path from s to t has got a critical edge,
i.e. an edge where the flow uses up all of the capacity.

Theorem: f is maximal $\Rightarrow$ f is blocking

Def. (Increase of a flow f by a flow r in L_f):

Let r be a flow in L_f. For each edge e, let $f'(e) = f(e) + r(e) - \overleftarrow{r}(e)$

Theorem: $|f'| = |f| + |r|$

References:

Cormen, ch. 26.4 (push relabel algorithms)
Turau, Kap. 6.4 (siehe auch Ausarbeitung und Vortrag Seminararbeit C. Padberg)
Alt, Kap. 4.7

Algorithmics 4

4.3 Computation of maximum flows in s/t-networks

Algorithm of Dinic

Difference to Edmonds-Karp:
Maximize each path in the flow, not just one.

```
1) Initialize f by 0 for all edges.
Repeat
        2a) Compute $L_f$
        2b) Search for a blocking flow r in $L_f$
        3) Increase f by the blocking flow r
until no blocking flow exists (t cannot be reached anymore in $L_f$ from s)
```

Time complexity: $O(n^2m)$ Improvement in Turau: $O(n^3)$

Outline of time complexity proof:

In each iteration, $\delta_f(s,t)$ is increased by at least 1 $\Rightarrow$ there are O(n) loop iterations

2a) and b) may be combined with a repeated depth first search: O(nm)
Improvement in Turau: $O(n^2)$

References for the details:

Cormen, ch. 26.4 (push relabel algorithms: with proof of correctness)
Turau, Kap. 6.4 (siehe auch Ausarbeitung und Vortrag Seminararbeit C. Padberg)
Alt, Kap. 4.7

Algorithmics

Sebastian Iwanowski
FH Wedel

4. Graph algorithms
4.4 Computation of graph matchings

Algorithmics 4

Matchings in graphs

Def.: A matching is a set of edges such that no edge is adjacent to another edge.

Def.: maximum matching:

i) maximum number of edges (only this is investigated in the references below)

ii) for valued edges: matching with maximum value

Def.: Set theoretic statement of graph matching (**2DM**):

Given a set $E \subseteq VxV$: Find a maximal subset $T \subseteq E$ where: All elements of T are pairwise disjoint.

Def.: Generalization of graph matching (**kDM**):

Given an set $E \subseteq Vx...xV$: Find a maximal subset $T \subseteq E$ where: All elements of T are pairwise disjoint.

Theorem: kDM is NP-complete for $k \geq 3$ and 2DM is in P.

References:

Alt, Definition 4.6.1

Laszlo Lovasz / Michael Plummer: *Matching Theory*, North Holland 1986, ISBN 9630541688, ch. 9.1
James McHugh: *Algorithmic Graph Theory*, Prentice Hall 1990, ISBN 0130236152, ch. 8.3
Christos Papadimitriou / Kenneth Steiglitz: *Combinatorial Optimization*, Dover 1998, ch. 10

Algorithmics 4

Matchings in graphs (maximum number of edges)

Special case considered in detail: Matchings in bipartite graphs

Def.: A flow f is integer-valued ⇔ f(u,v) is integer-valued *for each edge* (u,v)

Def.: For a given bipartite Graph G = ((V,U),E), construct an s/t-network G‘ as follows:
There is a source s ∈ G‘ with a directed edge *to* each vertex of V, each edge having capacity 1.
There is a target t ∈ G‘ with a directed edge *from* each vertex of U, each edge having capacity 1.
For each edge of G, there is a directed edge from a vertex in V to a vertex in U having capacity 1.

Lemma: (Alt 4.6.3) G has got a matching where |M|=k ⇔ G‘ has got an integer valued flow where |f| = k

Theorem: (Alt 4.6.4, Cormen 26.11) Given an s/t-network with integer capacities for all edges:
i) Then the value of a maximum flow is an integer as well.
ii) There exists always a maximum flow that is integer-valued.

Proof: i) follows from max flow / min cut theorem
ii) has to be proven separately

Corollary: The maximum matching in G is one-to-one related to the maximum flow in G‘.

References:

Alt, Kap. 4.6
Cormen, ch. 26.3 (maximum bipartite matching)

Algorithmics 4

Matchings in graphs (maximum number of edges)

Algorithms for bipartite matchings und integer valued flows

Prop.: A maximum bipartite matching can be found by the maximum flow algorithm of Edmonds-Karp in O(nm).
(Remark: For integer-valued networks, time complexity is better than for arbitrary networks).

Improvements:

Hopcroft-Karp: $O(n^{0.5}m)$
Alt et al.: $O(n^{1.5}(m/\log n)^{0.5})$ (this is an improvement for dense graphs)

Prop.: In unit networks (networks where each edge has got capacity 1), the algorithm of Dinic needs only $n^{0.5}$ iterations.
The inner operations do not sum up to O(nm) as in the general case, but only to O(m).
Thus, the algorithm of Dinic performed in unit networks requires run time $O(n^{0.5}m)$.

Corollary: The run time of Hopcroft-Karp for bipartite matching may be achieved also with the algorithm of Dinic.

References:

Alt, Kap. 4.7
Cormen, Problem 26-7
Turau Kap. 7 (vor allem Literaturhinweise 7.6)

Algorithmics 4

Matchings in graphs (maximum number of edges)

Techniques for matchings in general graphs

Def.: An augmenting path is a path from an unmatched vertex to an unmatched vertex using unmatched and matched edges alternately.

Def.: An outer vertex of an augmenting path is a vertex being an odd successor in the path, i.e., it is the 1., 3., 5., ... vertex of the augmenting path.
Except for the first, an outer vertex is always at the end of a matched edge.

Def.: A blossom is an odd cycle with a maximum matching:
A blossom consists of 2k+1 edges, k being matched.

(good examples in McHugh)

Remark: A blossom will be discovered in the course of the search for augmenting paths whenever the fact is discovered that two outer edges are adjacent.

References for details:

Laszlo Lovasz / Michael Plummer: *Matching Theory*, North Holland 1986, ISBN 9630541688, ch. 9.1
James McHugh: *Algorithmic Graph Theory*, Prentice Hall 1990, ISBN 0130236152, ch. 8.3
Christos Papadimitriou / Kenneth Steiglitz: *Combinatorial Optimization*, Dover 1998, ch. 10

Algorithmics 4

Matchings in graphs (maximum number of edges)

Algorithm of Edmonds for general graphs

Main loop:

```
1) Search for augmenting path AP:
        1a) Start with an unmatched vertex and an empty augmenting path AP.
        1b) Look at neighbors:
              If one is not matched -> augmenting path AP is found.
              Otherwise, augment AP by an unmatched edge to a neighbor and its matched vertex:
                 If this yields a blossom (outer vertex adjacent to outer vertex),
                    contract the blossom and continue with the contracted graph
                 If this yields an even cycle (outer vertex connected to inner vertex)
                                   or if an unmatched edge to a neighbor does not exist,
                    backtrack to a previous outer vertex and augment AP with other unmatched edge
               Continue with 1b)
2) If no augmenting path AP has been found -> Matching is maximum.
   If  augmenting path AP has been found:
        2a) Decontract graph by all previously found blossoms in reverse order of their findings.
        2b) Augment AP in original (decontracted) graph.
        2c) Increase matching by inverting the matching of AP and continue at 1)
```

References for details:

Laszlo Lovasz / Michael Plummer: *Matching Theory*, North Holland 1986, ISBN 9630541688, ch. 9.1
James McHugh: *Algorithmic Graph Theory*, Prentice Hall 1990, ISBN 0130236152, ch. 8.3
Christos Papadimitriou / Kenneth Steiglitz: *Combinatorial Optimization*, Dover 1998, ch. 10
Seminarvortrag Nr. 5 (in German), https://intern.fh-wedel.de/mitarbeiter/iw/lv/ss2013/seminar/
Demonstration and tutorial at https://www-m9.ma.tum.de/graph-algorithms/matchings-blossom-algorithm/index_en.html

Algorithmics 4

Matchings in graphs (maximum number of edges)

Algorithm of Edmonds for general graphs

Time complexity: 1) $O(n^2)$ (proof nontrivial)

2) $O(n)$ (clear)

The main loop is performed $O(n)$ times, because each time the matching is increased by one edge -> total time complexity: $O(n^3)$

Correctness:

Prop. 1: Matching is maximum ⇔ There is no augmenting path

„⇦ " is not trivial, will be shown in class (Papadimitriou, prop. 10.1)

Prop. 2: Let M be a matching in G. Let G have a blossom and let G‘ be the contracted graph: G has got an augmenting path for M ⇔ G‘ has got an augmenting path for M

(proof not difficult, but lengthy in detail, cf. Papadimitriou, prop. 10.4)

References for details:

Laszlo Lovasz / Michael Plummer: *Matching Theory*, North Holland 1986, ISBN 9630541688, ch. 9.1
James McHugh: *Algorithmic Graph Theory*, Prentice Hall 1990, ISBN 0130236152, ch. 8.3
Christos Papadimitriou / Kenneth Steiglitz: *Combinatorial Optimization*, Dover 1998, ch. 10
Seminarvortrag Nr. 5 (in German), https://intern.fh-wedel.de/mitarbeiter/iw/lv/ss2013/seminar/
Demonstration and tutorial at https://www-m9.ma.tum.de/graph-algorithms/matchings-blossom-algorithm/index_en.html

Algorithmics

Sebastian Iwanowski
FH Wedel

5. String Matching

Algorithmics 5

String Matching

Task: Given a text $T = \{t_1,\dots,t_n\}$ with n literals and a pattern $P = \{p_1,\dots,p_m\}$ with m literals: Find the starting positions where P occurs in T.

naive algorithm: needs O(nm) time

Algorithm of Knuth-Morris-Pratt: needs O(n) time

Def.: P_q denotes the prefix of P consisting of the first q literals. $(P_q = P[1],\dots,P[q])$

Def.: The prefix function $\pi: \mathbb{N}\setminus\{0\} \rightarrow \mathbb{N}$ for the pattern P is defined as:
$\pi(q) = k \Leftrightarrow$ k is the length of the longest strict prefix of P_q (*strict* means: $k < q$)
which is also a Suffix of P_q

General method of the KMP algorithm:

For each $q \leq m$, compute the value $\pi(q)$ of the prefix function and store it.
Then scan T in only one iteration and shift P at any mismatch in pattern position q+1
by $q - \pi(q)$. This does not omit any valid match.

In class: Why is this correct?

References:

Alt, Kap. 4.8
Cormen, ch. 32 (String matching), esp. 32.4 (KMP)

Algorithmics 5

String Matching

Algorithm of Knuth-Morris-Pratt: needs O(n) time

Implementation of main procedure:

```
i := 1; q := 0;
while i ≤ n do
{
    while (q>0) and (T[i] ≠ P[q+1])
        q := π (q);
    if T[i] = P[q+1] then q := q+1;
    if q = m
        then
        {
            print („Matching at position “, i-m);
            q := π (q);
        }
    i := i+1;
}
```

Invariant: *q=0: no prefix of P coincides at a suffix of T ending at i*
q>0 corresponds to the maximum index ≤ i s.t. (T[i-q+1],…,T[i]) coincides with (P[1],…,P[q])

In class: Why is this algorithm correct?

Home work:
Why does this algorithm need O(n) time?

References:

Alt, Kap. 4.8
Cormen, ch. 32 (String matching), esp. 32.4 (KMP)

Algorithmics 5

String Matching

Proof of the invariant:

Mathematical induction using i (where i is the loop counter which is increased at the end of the loop)**:**

i = 1:

Then q has an initial value of 0. => Inner loop is not executed.

q = 1 if comparison with P(1) is true and 0 otherwise which is the statement of the invariant.

Assume the invariant holds for i and consider i+1:

If T(i+1) = P(q+1) before the inner loop,

then the maximum prefix of P until i+1 has length maximum length of P until i plus 1 which is by assumption q + 1. So the assignment of the new q is correct.

If T(i+1) ≠ P(q+1) before the inner loop and q = 0,

then q remains 0 which is correct.

If T(i+1) ≠ P(q+1) before the inner loop and q > 0,

then (T[i-q+1],…,T[i]) coincides with (P[1],…,P[q]) by inductive assumption for i.

If there is no q‘ < q where T(i+1) = P(q‘+1), then no prefix coincides at a suffix of T ending at i+1.

The inner while loop will then eventually set q to 0 which is the correct invariant for i+1.

If for some q‘ < q, T(i+1) = P(q‘+1), and this is the end of a matched prefix,

such that T[i-q‘+1],…,T[i]) coincides with (P[1],…,P[q‘]), then by the above assumption

(P[1],…,P[q‘]) also coincides with (P[q-q‘+1],…,P[q]) which means that q‘ ≤ π(q).

Thus, it is ok if q is reduced accordingly in the inner while loop.

If q‘ = π(q), it is clearly the maximum and thus the q defined in the invariant.

Otherwise, q‘ < π(q) and will be found in a later loop iteration.

References:

Alt, Kap. 4.8

Cormen, ch. 32 (String matching), esp. 32.4 (KMP)

Algorithmics 5

String Matching

Algorithm of Knuth-Morris-Pratt: needs O(n) time

Implementation of prefix function (according to Cormen/Alt): needs O(m) time

```
π(1) := 0;
i := 2; q := 0;
while i ≤ m do
{
    while (q>0) and (P[i]≠P[q+1]) do
        q := π(q);
    if P[i]=P[q+1] then q := q+1;
    π(i) := q;
    i := i+1;
}
```

Invariant: *q=0: no strict prefix of P coincides at a suffix of P ending at i*
q>0 corresponds to the maximum index < i s.t.
(P[i-q+1],…,P[i]) coincides with (P[1],…,P[q])

Home work: Why is this algorithm correct?

In class:
Why does this algorithm need O(m) time?

References:

Alt, Kap. 4.8
Cormen, ch. 32 (String matching), esp. 32.4 (KMP)

Algorithmics

Sebastian Iwanowski
FH Wedel

6. Fundamentals of Computational Geometry
6.1 Basic Problems and the Use of Voronoi Diagrams for Solving them

Algorithmics 6

6.1 Basic Problems

Given n reference points in $\mathbb{R}^k$

1) Detection of the closest reference point for a new point $x \in \mathbb{R}^k$

Brute force algorithm: O(n)

i) Compute the distance between x and each reference point.
ii) Successively update the reference point with minimum distance to x.

2) Detection of the closest pair among the reference points

Brute force algorithm: $O(n^2)$

i) Compute the distance between each pair of reference points.
ii) Successively update the closest pair.

References:

deBerg et al., ch. 1, Levitin, ch. 3.3

Algorithmics 6

6.1 Basic Problems

Given n reference points in $\mathbb{R}^k$

3) Minimum spanning tree between all reference points

Brute force algorithm: $O(n^2 \log n)$

- i) Compute the distance for each pair of reference points and create an edge weighted with that distance.
- ii) In the resulting graph, compute the minimum spanning tree with the algorithm of Kruskal.

4) Detection of a set of reference points comprising the convex hull

Brute force algorithm: $O(n^{k+1})$

- i) For each subset of k points, compute the supporting hyperplane determined by them.
- ii) For each hyperplane, check if the other points are located on the same side:
  The supporting hyperplane is part of the convex hull
  $\Leftrightarrow$ All points are located on the same side.
- iii) Collect the points belonging to the supporting lines of the convex hull.

References:

deBerg et al., ch. 1, Levitin, ch. 3.3

Algorithmics 6

6.1 Voronoi Diagrams in the plane $\mathbb{R}^2$

Def.: Given a set S of n reference points in the plane.
The Voronoi diagram is a data structure belonging to the equivalence classes of the closest reference points *if that point is unique*:
The diagram partitions the plane into disjoint regions, each having exactly one reference point as the closest. The boundaries of these regions consist of the points whose closest reference point is not unique.

Structure: The Voronoi diagram V(S) consists of the following type of objects:

nodes:	points having at least three reference points as the closest
edges:	points having exactly two reference points as the closest
regions:	points having exactly one reference point as the closest

One region is uniquely associated with one reference point.
The incident edges and nodes are attributes of a region.
The attributes of an edge are the incident regions and nodes.
The attributes of a node are the incident regions and edges.

Theorem: There are only O(n) Voronoi objects.

References:

Klein, Kap. 5.1, 5.2 (in German), de Berg et al., ch. 7.1

Algorithmics 6

6.1 Voronoi Diagrams in the plane $\mathbb{R}^2$

Applications

1) Detection of the closest reference point for a new point $x \in \mathbb{R}^2$

Algorithm: $O(n^2 \log n)$ preprocessing + $O(\log n)$ runtime

O(n) preprocessing time is achievable by a more sophisticated method

Preprocessing: $O(n^2 \log n)$

i) Sort the nodes of V(S) by their y-coordinate and divide the plane into according horizontal strips.

ii) Intersect all edges of V(S) with the horizontal strips and sort them by x-coordinate within a particular strip.

Note: This may yield $O(n^2)$ segments

Runtime: $O(\log n)$

i) Determine the correct horizontal strip for x by binary search towards the y-coordinate.

ii) Determine the two edges closest to x by binary search towards the x-coordinate.

iii) The region belonging to both of the edges is the correct one.

References:

Klein, Kap. 5.3.1 (in German), Edelsbrunner ch. 13 (superficially)

Algorithmics 6

6.1 Voronoi Diagrams in the plane $\mathbb{R}^2$

Applications

2) Detection of the closest pair among the reference points

Algorithm: O(n)

i) Compute the distance only between points whose regions are adjacent.
ii) Successively update the closest pair.

of order O(n), because there are
only O(n) Voronoi objects

3) Minimum spanning tree in the plane

Algorithm: O(n log n)

i) Compute the distance from each reference point only to those reference points belonging to adjacent regions and create an edge weighted with that distance.
ii) In the resulting graph, compute the minimum spanning tree with the algorithm of Kruskal.

References:

Klein, Kap. 5.3.2, 5.3.3 (in German), Edelsbrunner ch. 13 (superficially)

Algorithmics 6

6.1 Voronoi Diagrams in the plane $\mathbb{R}^2$

Applications

4) Detection of a set of reference points comprising the convex hull

Consider the dual graph of the Voronoi diagram:
Connect two reference points by an edge iff the corresponding regions are adjacent.

If no four points are cocircular, this is a triangulation with the following properties:

$\left.\begin{array}{l}\text{1) The circumference of any triple of reference points belonging to a triangle of}\\ \quad\text{the triangulation does not contain any other reference point.}\\ \text{2) The minimum angle among all triangles is maximized among all possible}\\ \quad\text{triangulations.}\end{array}\right\}$ important for computer graphics

3) The convex hull of the reference points is obtained by the edges between points of infinite regions of the Voronoi diagram.

$\uparrow$ of order O(n), because there are only O(n) Voronoi objects

$\downarrow$

O(n)-Algorithm for Problem 4):

i. Start with an arbitrary voronoi region and proceed via adjacent regions to an infinite region: The correspondent reference point p_0 is the first point of the convex hull.
ii. Consecutively, proceed to adjacent regions that are infinite as well until p_0 is reached again.

References:

Klein, Kap. 5.4 (in German), Preparata ch. 3,4 (other algorithms), deBerg et al. ch. 9,11 (advanced)

Algorithmics

Sebastian Iwanowski
FH Wedel

6. Fundamentals of Computational Geometry
6.2 Sweep Techniques

Algorithmics 6

6.2 Sweep Techniques

Transformation static d-dimensional → dynamic (d-1)-dimensional

<u>d = 1:</u> Line Sweep

1) Maximum search among n numbers	**O(n)**
2) Closest Pair: Among n numbers, search the two that are closest together.	**O(n log n)**
Preprocessing: Sort all numbers	O(n log n)
Sweep: Scan from left to right and keep the closest pair respectively	O(n)

References:

Klein, Kap. 2.2 (in German)

Algorithmics 6

6.2 Sweep Techniques

Transformation static d-dimensional → dynamic (d-1)-dimensional

d = 2: Plane Sweep

3) Closest Pair: Among n points, search the two that are closest together. **O(n log n)**

Preprocessing: Sort all points by x-coordinate
Sweep: Scan from left to right with 2 vertical lines *left* and *right*:

Invariants:

Horizontal distance between *left* and *right* is the minimum distance of the closest pair left of *left*.
Line content maintains all points between *left* and *right* sorted by y-coordinate.

Events and actions:

left passes point p: p is deleted — O(log n)
right passes point p: p is inserted into *line content* and its distance is computed — O(log n)
to all other points of *line content* of which the y coordinate differs from p at most by the minimum distance between points found so far.

only constant number! →

References: Klein, Kap. 2.3.1 (in German)

6.2 Sweep Techniques

Transformation static d-dimensional → dynamic (d-1)-dimensional

Characteristic properties of sweep techniques:

- Scan over selected and sorted x-coordinates (events) from left to right:
  The events lie in an EPS (Event point schedule)
- Sweep status structure (SSS) with invariants (SLS: Sweep Line Status)
- Events are computed statically during preprocessing (i.e. original reference points) and dynamically during updating the SSS.
-

Sleeping objects:	right of SSS, will yet be considered
Active objects:	within SSS, are currently relevant for updating the SSS
Dead objects:	left of SSS, need never be considered again

References:

Klein, Kap. 2.3.1 (in German), Preparata (see subject index), de Berg et al., ch. 2 (for other application)

Algorithmics 6

Application: Computation of Voronoi diagrams by plane sweep

Objects of SSS:

All segments have got a horizontal axis, because L is vertical.
This makes the intersection points of adjacent parabolic segments monotonic in y coordinate.

- Right vertical line L (current x of EPS)
- Left beach line consisting of:
  - Parabolic segments P(p,q,r) belonging to Bisector (p, L)
    and adjacent to Bisector (q,L) above and Bisector (r,L) below.
  - Spikes: Bisectors B(p,q) for two adjacent parabolic segments
    belonging to Bisector (p, L) and Bisector (q, L).
    Each parabolic segment has got two adjacent spikes
    (except for the first and the last).

The parabolic segments are ordered in SSS by y coordinate of their intersection points
(Note: An intersection coordinate will only be computed explicitely when a parabola vanishes in a spike event,
see next slide)

Lemma: The overall size of the beach line and hence of SSS is of order O(n)

References:

Klein, Kap. 6.3 (in German), de Berg et al., ch. 7.2

Algorithmics 6

Application: Computation of Voronoi diagrams by plane sweep

Objects of EPS:

- Point events: x coordinate of a reference point p (p_x, p_y)

  How do we insert the corresponding parabolic segment at the correct position into the SSS?

  Perform a logarithmic search in the SSS and check with each parabolic segment $P(r,q_1,q_2)$ encountered in the SSS:
  Compute the y coordinates y_1 and y_2 of the real intersection between parabolas (r, q_1) and (r, q_2) respectively. Note that these real intersection coordinates must be computed now because the parabola equations change dynamically depending on p.
  If both, y_1 and y_2, is higher (lower) than p_y, search below (above).
  If p_y is in between y_1 and y_2, $P(r,q_1,q_2)$ is the correct parabolic segment hit by the new parabola. $P(r,q_1,q_2)$ has to be replaced by $P(r,q_1,p)$, $P(p,r,r)$, $P(r,p,q_2)$.

- Spike events: x coordinate of the sweep line at which a beach line segment vanishes.

  How do we compute the x coordinate of a spike event?

  Let (x_0,y_0) be the intersection point of two adjacent spikes.
  Let $p_i = (x_i,y_i)$ be one of the 3 reference points contributing to one of the two spikes.
  Then $x := x_0 + | (x_i,y_i) - (x_0,y_0) |$.

References:

Klein, Kap. 6.3 (in German), de Berg et al., ch. 7.2

Algorithmics 6

Application: Computation of Voronoi diagrams by plane sweep

Events and actions during sweep:

- Point event: New point is passed: Generation of new beach line segment.
  Analogously to the above, this requires the computation of new spikes
  (new adjacencies, update of SSS) and spike events (update of EPS).

- Spike event: Intersection of adjacent spikes: Associated beach line segment vanishes.
  This requires an update of the beach line in the SSS:
  The reference point between the two intersecting spikes is not relevant anymore.
  Its Voronoi cell is finally computed.
  This requires the computation of a new spike and the intersection with its
  upper and lower neighbor (if on the right hand) as new spike events.
  These spike events have to be inserted into the EPS.

  NOTE: A spike event may not be relevant anymore because the involved spikes have been terminated due to another spike event from above or below earlier in this sweep. Thus, the adjacency of the involved bisectors must be checked again, and if not adjacent anymore, the spike event is ignored. Acknowledged spike events will lead to the creation of Voronoi nodes (the common center of three involved reference points). The started and terminated spikes are the Voronoi edges.

References:

Klein, Kap. 6.3 (in German), de Berg et al., ch. 7.2

Algorithmics 6

Application: Computation of Voronoi diagrams by plane sweep

Run time analysis:

- The n point events are inserted at the initialisation of the algorithm into EPS. O(n log n)
  The SSS is initialised empty.

During the sweep:

- Point event: inserts at most 3 items into SSS, deletes at most 1 item from SSS,
  inserts at most 2 items into EPS (the spike events),
- Spike event: deletes 1 item from SSS (the vanishíng parabola), inserts 2 items into EPS

Run time: Update of each event in O(log n)

$O(n)$ events ➔ **Total time complexity**: $O(n \log n)$

↖ *crucial!* ↖ *This is optimal!*

References:

Klein, Kap. 6.3 (in German), de Berg et al., ch. 7.2