

WS 2018/19

# Efficient Algorithms and Data Structures

Harald Räcke

Fakultät für Informatik  
TU München

<http://www14.in.tum.de/lehre/2018WS/ea/>

Winter Term 2018/19

## Part I

### Organizational Matters

## Part I

### Organizational Matters

- ▶ Modul: IN2003
- ▶ Name: “Efficient Algorithms and Data Structures”  
“Effiziente Algorithmen und Datenstrukturen”
- ▶ ECTS: 8 Credit points
- ▶ Lectures:
  - ▶ 4 SWS  
Mon 10:00–12:00 (Room Interim2)  
Fri 10:00–12:00 (Room Interim2)
- ▶ Webpage: <http://www14.in.tum.de/lehre/2018WS/ea/>

- ▶ Required knowledge:
  - ▶ IN0001, IN0003  
“Introduction to Informatics 1/2”  
“Einführung in die Informatik 1/2”
  - ▶ IN0007  
“Fundamentals of Algorithms and Data Structures”  
“Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - ▶ IN0011  
“Basic Theoretic Informatics”  
“Einführung in die Theoretische Informatik” (THEO)
  - ▶ IN0015  
“Discrete Structures”  
“Diskrete Strukturen” (DS)
  - ▶ IN0018  
“Discrete Probability Theory”  
“Diskrete Wahrscheinlichkeitstheorie” (DWT)

## The Lecturer

- ▶ Harald Räcke
- ▶ Email: raecke@in.tum.de
- ▶ Room: 03.09.044
- ▶ Office hours: (by appointment)

## Tutorials

- A01** Monday, 12:00–14:00, 00.08.038 (Lederer)
- A02** Monday, 12:00–14:00, 00.09.038 (Stotz)
- A03** Monday, 14:00–16:00, 02.09.023 (Lederer)
- B04** Tuesday, 10:00–12:00, 00.08.053 (Czerner)
- D05** Thursday, 10:00–12:00, 03.11.018 (Stotz)
- E06** Friday, 12:00–14:00, 00.13.009 (Czerner)

## Assignment sheets

In order to pass the module you need to pass an exam.

## Assessment

### Assignment Sheets:

- ▶ An assignment sheet is usually made available on Monday on the module webpage.
- ▶ Solutions have to be handed in in the following week before the lecture on Monday.
- ▶ You can hand in your solutions by putting them in the mailbox "Efficient Algorithms" on the basement floor in the MI-building.
- ▶ Solutions have to be given in English.
- ▶ Solutions will be discussed in the tutorial of the week when the sheet has been handed in, **i.e., sheet may not be corrected by this time.**
- ▶ **You should submit solutions in groups of up to 2 people.**

## Assessment

### Assignment Sheets:

- ▶ Submissions must be handwritten by a member of the group. Please indicate who wrote the submission.
- ▶ Don't forget name and student id number for each group member.

## Assessment

Assignment can be used to improve you grade




### Requirements for Bonus

- ▶ 50% of the points are achieved on submissions 2–8,
- ▶ 50% of the points are achieved on submissions 9–14,
- ▶ each group member has written at least 4 solutions.





## 1 Contents

- ▶ Foundations
  - ▶ Machine models
  - ▶ Efficiency measures
  - ▶ Asymptotic notation
  - ▶ Recursion
- ▶ Higher Data Structures
  - ▶ Search trees
  - ▶ Hashing
  - ▶ Priority queues
  - ▶ Union/Find data structures
- ▶ Cuts/Flows
- ▶ Matchings





## 2 Literatur

-  Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman:  
*The design and analysis of computer algorithms*,  
Addison-Wesley Publishing Company: Reading (MA), 1974
-  Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest,  
Clifford Stein:  
*Introduction to algorithms*,  
McGraw-Hill, 1990
-  Michael T. Goodrich, Roberto Tamassia:  
*Algorithm design: Foundations, analysis, and internet  
examples*,  
John Wiley & Sons, 2002

## 2 Literatur

-  Ronald L. Graham, Donald E. Knuth, Oren Patashnik:  
*Concrete Mathematics*,  
2. Auflage, Addison-Wesley, 1994
-  Volker Heun:  
*Grundlegende Algorithmen: Einführung in den Entwurf und die Analyse effizienter Algorithmen*,  
2. Auflage, Vieweg, 2003
-  Jon Kleinberg, Eva Tardos:  
*Algorithm Design*,  
Addison-Wesley, 2005
-  Donald E. Knuth:  
*The art of computer programming. Vol. 1: Fundamental Algorithms*,  
3. Auflage, Addison-Wesley, 1997

## 2 Literatur

-  Donald E. Knuth:  
*The art of computer programming. Vol. 3: Sorting and Searching*,  
3. Auflage, Addison-Wesley, 1997
-  Christos H. Papadimitriou, Kenneth Steiglitz:  
*Combinatorial Optimization: Algorithms and Complexity*,  
Prentice Hall, 1982
-  Uwe Schöning:  
*Algorithmik*,  
Spektrum Akademischer Verlag, 2001
-  Steven S. Skiena:  
*The Algorithm Design Manual*,  
Springer, 1998

## Part II

### Foundations

## 3 Goals

- ▶ Gain knowledge about efficient algorithms for important problems, i.e., learn how to solve certain types of problems efficiently.
- ▶ Learn how to analyze and judge the efficiency of algorithms.
- ▶ Learn how to design efficient algorithms.

## 4 Modelling Issues

### What do you measure?

- ▶ Memory requirement
- ▶ Running time
- ▶ Number of comparisons
- ▶ Number of multiplications
- ▶ Number of hard-disc accesses
- ▶ Program size
- ▶ Power consumption
- ▶ ...

## 4 Modelling Issues

### How do you measure?

- ▶ Implementing and testing on representative inputs
  - ▶ How do you choose your inputs?
  - ▶ May be very time-consuming.
  - ▶ Very reliable results if done correctly.
  - ▶ Results only hold for a specific machine and for a specific set of inputs.
- ▶ Theoretical analysis in a specific **model of computation**.
  - ▶ Gives **asymptotic bounds** like “this algorithm always runs in time  $\mathcal{O}(n^2)$ ”.
  - ▶ Typically focuses on the **worst case**.
  - ▶ Can give lower bounds like “any comparison-based sorting algorithm needs at least  $\Omega(n \log n)$  comparisons in the worst case”.

## 4 Modelling Issues

### Input length

The theoretical bounds are usually given by a function  $f: \mathbb{N} \rightarrow \mathbb{N}$  that maps the **input length** to the running time (or storage space, comparisons, multiplications, program size etc.).

The **input length** may e.g. be

- ▶ the size of the input (number of bits)
- ▶ the number of arguments

### Example 1

Suppose  $n$  numbers from the interval  $\{1, \dots, N\}$  have to be sorted. In this case we usually say that the input length is  $n$  instead of e.g.  $n \log N$ , which would be the number of bits required to encode the input.

## Model of Computation

### How to measure performance

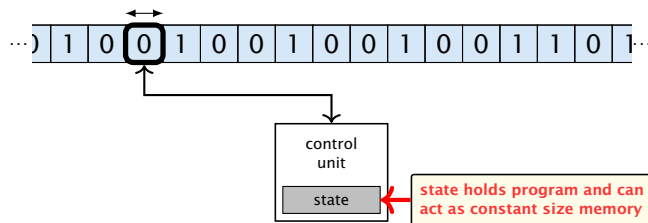
1. Calculate running time and storage space etc. on a simplified, idealized model of computation, e.g. Random Access Machine (RAM), Turing Machine (TM), ...
2. Calculate number of certain basic operations: comparisons, multiplications, harddisc accesses, ...

Version 2. is often easier, but focusing on one type of operation makes it more difficult to obtain meaningful results.

## Turing Machine

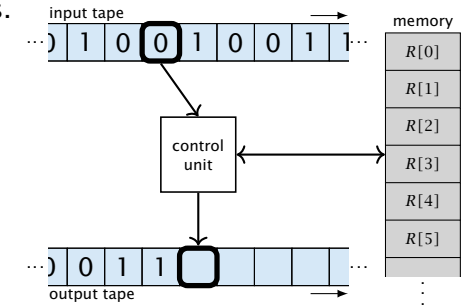
- ▶ Very simple model of computation.
- ▶ Only the “current” memory location can be altered.
- ▶ Very good model for discussing computability, or polynomial vs. exponential time.
- ▶ Some simple problems like recognizing whether input is of the form  $xx$ , where  $x$  is a string, have quadratic lower bound.

⇒ **Not a good model for developing efficient algorithms.**



## Random Access Machine (RAM)

- ▶ Input tape and output tape (sequences of zeros and ones; unbounded length).
- ▶ Memory unit: infinite but countable number of registers  $R[0], R[1], R[2], \dots$
- ▶ Registers hold integers.
- ▶ Indirect addressing.



Note that in the picture on the right the tapes are one-directional, and that a READ- or WRITE-operation always advances its tape.

## Random Access Machine (RAM)

### Operations

- ▶ input operations (input tape  $\rightarrow R[i]$ )
  - ▶ READ  $i$
- ▶ output operations ( $R[i] \rightarrow$  output tape)
  - ▶ WRITE  $i$
- ▶ register-register transfers
  - ▶  $R[j] := R[i]$
  - ▶  $R[j] := 4$
- ▶ **indirect** addressing
  - ▶  $R[j] := R[R[i]]$   
loads the content of the  $R[i]$ -th register into the  $j$ -th register
  - ▶  $R[R[i]] := R[j]$   
loads the content of the  $j$ -th into the  $R[i]$ -th register

## Random Access Machine (RAM)

### Operations

- ▶ branching (including loops) based on comparisons
  - ▶ jump  $x$   
jumps to position  $x$  in the program;  
sets instruction counter to  $x$ ;
  - ▶ jumpz  $x R[i]$   
jump to  $x$  if  $R[i] = 0$   
if not the instruction counter is increased by 1;
  - ▶ jumpi  $i$   
jump to  $R[i]$  (indirect jump);
- ▶ arithmetic instructions:  $+, -, \times, /$ 
  - ▶  $R[i] := R[j] + R[k];$   
 $R[i] := -R[k];$

The jump-directives are very close to the jump-instructions contained in the assembler language of real machines.

## Model of Computation

- ▶ **uniform** cost model  
Every operation takes time 1.
- ▶ **logarithmic** cost model  
The cost depends on the content of memory cells:
  - ▶ The time for a step is equal to the largest operand involved;
  - ▶ The storage space of a register is equal to the length (in bits) of the largest value ever stored in it.

**Bounded word RAM model:** cost is uniform but the largest value stored in a register may not exceed  $2^w$ , where usually  $w = \log_2 n$ .

The latter model is quite realistic as the word-size of a standard computer that handles a problem of size  $n$  must be at least  $\log_2 n$  as otherwise the computer could either not store the problem instance or not address all its memory.

## 4 Modelling Issues

### Example 2

#### Algorithm 1 RepeatedSquaring( $n$ )

```

1:  $r \leftarrow 2$ ;
2: for  $i = 1 \rightarrow n$  do
3:    $r \leftarrow r^2$ 
4: return  $r$ 
    
```

- ▶ running time:
  - ▶ uniform model:  $n$  steps
  - ▶ logarithmic model:  $1 + 2 + 4 + \dots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$
- ▶ space requirement:
  - ▶ uniform model:  $\mathcal{O}(1)$
  - ▶ logarithmic model:  $\mathcal{O}(2^n)$

There are **different types of complexity bounds**:

- ▶ **best-case** complexity:

$$C_{bc}(n) := \min\{C(x) \mid |x| = n\}$$

Usually easy to analyze, but not very meaningful.

- ▶ **worst-case** complexity:

$$C_{wc}(n) := \max\{C(x) \mid |x| = n\}$$

Usually moderately easy to analyze; sometimes too pessimistic.

- ▶ **average case** complexity:

$$C_{avg}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

more general: probability measure  $\mu$

$$C_{avg}(n) := \sum_{x \in I_n} \mu(x) \cdot C(x)$$

$C(x)$	cost of instance $x$
$ x $	input length of instance $x$
$I_n$	set of instances of length $n$

There are **different types of complexity bounds**:

- ▶ **amortized** complexity:

The average cost of data structure operations over a worst case sequence of operations.

- ▶ **randomized** complexity:

The algorithm may use random bits. Expected running time (over all possible choices of random bits) for a fixed input  $x$ . Then take the worst-case over all  $x$  with  $|x| = n$ .

$C(x)$	cost of instance $x$
$ x $	input length of instance $x$
$I_n$	set of instances of length $n$

## 4 Modelling Issues

### Bibliography

[MS08] Kurt Mehlhorn, Peter Sanders:  
*Algorithms and Data Structures — The Basic Toolbox*,  
Springer, 2008

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:  
*Introduction to algorithms (3rd ed.)*,  
McGraw-Hill, 2009

Chapter 2.1 and 2.2 of [MS08] and Chapter 2 of [CLRS90] are relevant for this section.



## 5 Asymptotic Notation

We are usually not interested in exact running times, but only in an asymptotic classification of the running time, that ignores constant factors and constant additive offsets.

- ▶ We are usually interested in the running times for large values of  $n$ . Then constant additive terms do not play an important role.
- ▶ An exact analysis (e.g. *exactly* counting the number of operations in a RAM) may be hard, but wouldn't lead to more precise results as the computational model is already quite a distance from reality.
- ▶ A linear speed-up (i.e., by a constant factor) is always possible by e.g. implementing the algorithm on a faster machine.
- ▶ Running time should be expressed by simple functions.



## Asymptotic Notation

### Formal Definition

Let  $f$  denote functions from  $\mathbb{N}$  to  $\mathbb{R}^+$ .

- ▶  $\mathcal{O}(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$   
(set of functions that asymptotically grow **not faster** than  $f$ )
- ▶  $\Omega(f) = \{g \mid \exists c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$   
(set of functions that asymptotically grow **not slower** than  $f$ )
- ▶  $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$   
(functions that asymptotically have **the same growth** as  $f$ )
- ▶  $o(f) = \{g \mid \forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$   
(set of functions that asymptotically grow **slower** than  $f$ )
- ▶  $\omega(f) = \{g \mid \forall c > 0 \exists n_0 \in \mathbb{N}_0 \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$   
(set of functions that asymptotically grow **faster** than  $f$ )



## Asymptotic Notation

There is an equivalent definition using limes notation (**assuming that the respective limes exists**).  $f$  and  $g$  are functions from  $\mathbb{N}_0$  to  $\mathbb{R}_0^+$ .

- ▶  $g \in \mathcal{O}(f): 0 \leq \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$
- ▶  $g \in \Omega(f): 0 < \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} \leq \infty$
- ▶  $g \in \Theta(f): 0 < \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} < \infty$
- ▶  $g \in o(f): \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = 0$
- ▶  $g \in \omega(f): \lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = \infty$

- Note that for the version of the Landau notation defined here, we assume that  $f$  and  $g$  are positive functions.
- There also exist versions for arbitrary functions, and for the case that the limes is not infinity.





## Asymptotic Notation

### Abuse of notation

1. People write  $f = \mathcal{O}(g)$ , when they mean  $f \in \mathcal{O}(g)$ . This is **not** an equality (how could a function be equal to a set of functions).
2. People write  $f(n) = \mathcal{O}(g(n))$ , when they mean  $f \in \mathcal{O}(g)$ , with  $f: \mathbb{N} \rightarrow \mathbb{R}^+, n \mapsto f(n)$ , and  $g: \mathbb{N} \rightarrow \mathbb{R}^+, n \mapsto g(n)$ .
3. People write e.g.  $h(n) = f(n) + o(g(n))$  when they mean that there exists a function  $z: \mathbb{N} \rightarrow \mathbb{R}^+, n \mapsto z(n), z \in o(g)$  such that  $h(n) = f(n) + z(n)$ .

2. In this context  $f(n)$  does **not** mean the function  $f$  evaluated at  $n$ , but instead it is a shorthand for the function itself (leaving out domain and codomain and only giving the rule of correspondence of the function).

3. This is particularly useful if you do not want to ignore constant factors. For example the median of  $n$  elements can be determined using  $\frac{3}{2}n + o(n)$  comparisons.

## Asymptotic Notation

### Abuse of notation

4. People write  $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$ , when they mean  $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$ . Again this is not an equality.

2. In this context  $f(n)$  does **not** mean the function  $f$  evaluated at  $n$ , but instead it is a shorthand for the function itself (leaving out domain and codomain and only giving the rule of correspondence of the function).

3. This is particularly useful if you do not want to ignore constant factors. For example the median of  $n$  elements can be determined using  $\frac{3}{2}n + o(n)$  comparisons.

## Asymptotic Notation in Equations

How do we interpret an expression like:

$$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$$

Here,  $\Theta(n)$  stands for an **anonymous function** in the set  $\Theta(n)$  that makes the expression true.

Note that  $\Theta(n)$  is on the right hand side, otw. this interpretation is wrong.

## Asymptotic Notation in Equations

How do we interpret an expression like:

$$2n^2 + \mathcal{O}(n) = \Theta(n^2)$$

Regardless of how we choose the anonymous function  $f(n) \in \mathcal{O}(n)$  there is an anonymous function  $g(n) \in \Theta(n^2)$  that makes the expression true.

## Asymptotic Notation in Equations

How do we interpret an expression like:

$$\sum_{i=1}^n \Theta(i) = \Theta(n^2)$$

### Careful!

“It is understood” that every occurrence of an  $\Theta$ -symbol (or  $\Theta, \Omega, o, \omega$ ) on the left represents **one anonymous function**.

Hence, the left side is **not** equal to

$$\Theta(1) + \Theta(2) + \dots + \Theta(n-1) + \Theta(n)$$

$\Theta(1) + \Theta(2) + \dots + \Theta(n-1) + \Theta(n)$  does not really have a reasonable interpretation.

The  $\Theta(i)$ -symbol on the left represents **one** anonymous function  $f: \mathbb{N} \rightarrow \mathbb{R}^+$ , and then  $\sum_i f(i)$  is computed.

## Asymptotic Notation in Equations

We can view an expression containing asymptotic notation as generating a set:

$$n^2 \cdot \Theta(n) + \Theta(\log n)$$

represents

$$\{f: \mathbb{N} \rightarrow \mathbb{R}^+ \mid f(n) = n^2 \cdot g(n) + h(n)\}$$

with  $g(n) \in \Theta(n)$  and  $h(n) \in \Theta(\log n)$

Recall that according to the previous slide e.g. the expressions  $\sum_{i=1}^n \Theta(i)$  and  $\sum_{i=1}^{n/2} \Theta(i) + \sum_{i=n/2+1}^n \Theta(i)$  generate different sets.

## Asymptotic Notation in Equations

Then an asymptotic equation can be interpreted as containment btw. two sets:

$$n^2 \cdot \Theta(n) + \Theta(\log n) = \Theta(n^2)$$

represents

$$n^2 \cdot \Theta(n) + \Theta(\log n) \subseteq \Theta(n^2)$$

Note that the equation does not hold.

## Asymptotic Notation

### Lemma 3

Let  $f, g$  be functions with the property

$\exists n_0 > 0 \forall n \geq n_0 : f(n) > 0$  (the same for  $g$ ). Then

- ▶  $c \cdot f(n) \in \Theta(f(n))$  for any constant  $c$
- ▶  $\Theta(f(n)) + \Theta(g(n)) = \Theta(f(n) + g(n))$
- ▶  $\Theta(f(n)) \cdot \Theta(g(n)) = \Theta(f(n) \cdot g(n))$
- ▶  $\Theta(f(n)) + \Theta(g(n)) = \Theta(\max\{f(n), g(n)\})$

The expressions also hold for  $\Omega$ . Note that this means that  $f(n) + g(n) \in \Theta(\max\{f(n), g(n)\})$ .

## Asymptotic Notation

### Comments

- ▶ Do not use asymptotic notation within induction proofs.
- ▶ For any constants  $a, b$  we have  $\log_a n = \Theta(\log_b n)$ . Therefore, we will usually ignore the base of a logarithm within asymptotic notation.
- ▶ In general  $\log n = \log_2 n$ , i.e., we use 2 as the default base for the logarithm.



## Asymptotic Notation

In general asymptotic classification of running times is a good measure for comparing algorithms:

- ▶ If the running time analysis is tight and actually occurs in practise (i.e., the asymptotic bound is not a purely theoretical worst-case bound), then the algorithm that has better asymptotic running time will always outperform a weaker algorithm for large enough values of  $n$ .
- ▶ However, suppose that I have two algorithms:
  - ▶ Algorithm A. Running time  $f(n) = 1000 \log n = \mathcal{O}(\log n)$ .
  - ▶ Algorithm B. Running time  $g(n) = \log^2 n$ .

Clearly  $f = o(g)$ . However, as long as  $\log n \leq 1000$  Algorithm B will be more efficient.



## 5 Asymptotic Notation

### Bibliography

- [MS08] Kurt Mehlhorn, Peter Sanders: *Algorithms and Data Structures — The Basic Toolbox*, Springer, 2008
- [CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein: *Introduction to algorithms (3rd ed.)*, McGraw-Hill, 2009

Mainly Chapter 3 of [CLRS90]. [MS08] covers this topic in chapter 2.1 but not very detailed.



## 6 Recurrences

### Algorithm 2 mergesort(list $L$ )

```
1:  $n \leftarrow \text{size}(L)$ 
2: if  $n \leq 1$  return  $L$ 
3:  $L_1 \leftarrow L[1 \cdots \lfloor \frac{n}{2} \rfloor]$ 
4:  $L_2 \leftarrow L[\lfloor \frac{n}{2} \rfloor + 1 \cdots n]$ 
5: mergesort( $L_1$ )
6: mergesort( $L_2$ )
7:  $L \leftarrow \text{merge}(L_1, L_2)$ 
8: return  $L$ 
```

This algorithm requires

$$T(n) = T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \mathcal{O}(n) \leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n)$$

comparisons when  $n > 1$  and 0 comparisons when  $n \leq 1$ .



## Recurrences

How do we bring the expression for the number of comparisons ( $\approx$  running time) into a **closed form**?

For this we need to **solve** the recurrence.



## Methods for Solving Recurrences

### 1. Guessing+Induction

Guess the right solution and prove that it is correct via induction. It needs experience to make the right guess.

### 2. Master Theorem

For a lot of recurrences that appear in the analysis of algorithms this theorem can be used to obtain tight asymptotic bounds. It does not provide exact solutions.

### 3. Characteristic Polynomial

Linear homogenous recurrences can be solved via this method.



## Methods for Solving Recurrences

### 4. Generating Functions

A more general technique that allows to solve certain types of linear inhomogenous relations and also sometimes non-linear recurrence relations.

### 5. Transformation of the Recurrence

Sometimes one can transform the given recurrence relations so that it e.g. becomes linear and can therefore be solved with one of the other techniques.



## 6.1 Guessing+Induction

First we need to get rid of the  $\mathcal{O}$ -notation in our recurrence:

$$T(n) \leq \begin{cases} 2T(\lceil \frac{n}{2} \rceil) + cn & n \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

### Informal way:

Assume that instead we have

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

One way of solving such a recurrence is to **guess** a solution, and check that it is correct by plugging it in.



## 6.1 Guessing+Induction

Suppose we guess  $T(n) \leq dn \log n$  for a constant  $d$ . Then

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(d\frac{n}{2} \log \frac{n}{2}\right) + cn \\ &= dn(\log n - 1) + cn \\ &= dn \log n + (c - d)n \\ &\leq dn \log n \end{aligned}$$

if we choose  $d \geq c$ .

Formally, this is not correct if  $n$  is not a power of 2. Also even in this case one would need to do an induction proof.

## 6.1 Guessing+Induction

$$T(n) \leq \begin{cases} 2T\left(\frac{n}{2}\right) + cn & n \geq 16 \\ b & \text{otw.} \end{cases}$$

**Guess:**  $T(n) \leq dn \log n$ .

**Proof.** (by induction)

► **base case** ( $2 \leq n < 16$ ): **true** if we choose  $d \geq b$ .

► **induction step**  $2 \dots n - 1 \rightarrow n$ :

Suppose statem. is true for  $n' \in \{2, \dots, n - 1\}$ , and  $n \geq 16$ .

We prove it for  $n$ :

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(d\frac{n}{2} \log \frac{n}{2}\right) + cn \\ &= dn(\log n - 1) + cn \\ &= dn \log n + (c - d)n \\ &\leq dn \log n \end{aligned}$$

- Note that this proves the statement for  $n \in \mathbb{N}_{\geq 2}$ , as the statement is wrong for  $n = 1$ .
- The base case is usually omitted, as it is the same for different recurrences.

Hence, statement is **true** if we choose  $d \geq c$ .

## 6.1 Guessing+Induction

How do we get a result for all values of  $n$ ?

We consider the following recurrence instead of the original one:

$$T(n) \leq \begin{cases} 2T\left(\lceil \frac{n}{2} \rceil\right) + cn & n \geq 16 \\ b & \text{otherwise} \end{cases}$$

Note that we can do this as for constant-sized inputs the running time is always some constant ( $b$  in the above case).

## 6.1 Guessing+Induction

We also make a guess of  $T(n) \leq dn \log n$  and get

$$\begin{aligned} T(n) &\leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \\ &\leq 2\left(d\left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil\right) + cn \\ \left\lceil \frac{n}{2} \right\rceil &\leq \frac{n}{2} + 1 && \leq 2\left(d\left(\frac{n}{2} + 1\right) \log\left(\frac{n}{2} + 1\right)\right) + cn \\ \frac{n}{2} + 1 &\leq \frac{9}{16}n && \leq dn \log\left(\frac{9}{16}n\right) + 2d \log n + cn \\ \log \frac{9}{16}n &= \log n + (\log 9 - 4) && = dn \log n + (\log 9 - 4)dn + 2d \log n + cn \\ \log n &\leq \frac{n}{4} && \leq dn \log n + (\log 9 - 3.5)dn + cn \\ &&& \leq dn \log n - 0.33dn + cn \\ &&& \leq dn \log n \end{aligned}$$

for a suitable choice of  $d$ .

## 6.2 Master Theorem

Note that the cases do not cover all possibilities.

### Lemma 4

Let  $a \geq 1, b \geq 1$  and  $\epsilon > 0$  denote constants. Consider the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

#### Case 1.

If  $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$  then  $T(n) = \Theta(n^{\log_b a})$ .

#### Case 2.

If  $f(n) = \Theta(n^{\log_b(a)} \log^k n)$  then  $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$ ,  $k \geq 0$ .

#### Case 3.

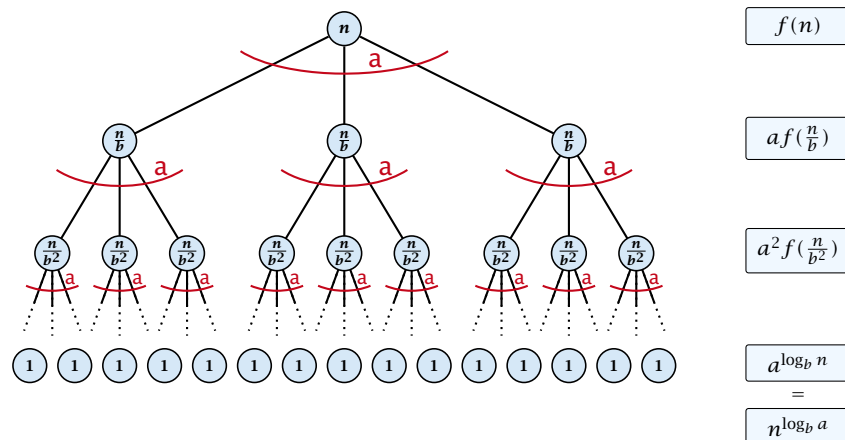
If  $f(n) = \Omega(n^{\log_b(a)+\epsilon})$  and for sufficiently large  $n$   $af\left(\frac{n}{b}\right) \leq cf(n)$  for some constant  $c < 1$  then  $T(n) = \Theta(f(n))$ .

## 6.2 Master Theorem

We prove the Master Theorem for the case that  $n$  is of the form  $b^\ell$ , and we assume that the non-recursive case occurs for problem size 1 and incurs cost 1.

## The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:



## 6.2 Master Theorem

This gives

$$T(n) = n^{\log_b a} + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right).$$

Case 1. Now suppose that  $f(n) \leq cn^{\log_b a - \epsilon}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon}$$

$$\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i} (b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} = cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} (b^\epsilon)^i$$

$$\begin{aligned} \boxed{\sum_{i=0}^k q^i = \frac{q^{k+1} - 1}{q - 1}} &= cn^{\log_b a - \epsilon} (b^{\epsilon \log_b n} - 1) / (b^\epsilon - 1) \\ &= cn^{\log_b a - \epsilon} (n^\epsilon - 1) / (b^\epsilon - 1) \\ &= \frac{c}{b^\epsilon - 1} n^{\log_b a} (n^\epsilon - 1) / (n^\epsilon) \end{aligned}$$

Hence,

$$T(n) \leq \left(\frac{c}{b^\epsilon - 1} + 1\right) n^{\log_b a} \Rightarrow T(n) = \mathcal{O}(n^{\log_b a}).$$

Case 2. Now suppose that  $f(n) \leq cn^{\log_b a}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} = cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 = cn^{\log_b a} \log_b n$$

Hence,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n) \Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log n).$$

Case 2. Now suppose that  $f(n) \geq cn^{\log_b a}$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} = cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 = cn^{\log_b a} \log_b n$$

Hence,

$$T(n) = \Omega(n^{\log_b a} \log_b n) \Rightarrow T(n) = \Omega(n^{\log_b a} \log n).$$

Case 2. Now suppose that  $f(n) \leq cn^{\log_b a} (\log_b(n))^k$ .

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b \left(\frac{n}{b^i}\right)\right)^k = cn^{\log_b a} \sum_{i=0}^{\ell - 1} \left(\log_b \left(\frac{b^\ell}{b^i}\right)\right)^k = cn^{\log_b a} \sum_{i=0}^{\ell - 1} (\ell - i)^k = cn^{\log_b a} \sum_{i=1}^{\ell} i^k \approx \frac{1}{k} \ell^{k+1} \approx \frac{c}{k} n^{\log_b a} \ell^{k+1} \Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log^{k+1} n).$$

Case 3. Now suppose that  $f(n) \geq dn^{\log_b a + \epsilon}$ , and that for sufficiently large  $n$ :  $af(n/b) \leq cf(n)$ , for  $c < 1$ .

From this we get  $a^i f(n/b^i) \leq c^i f(n)$ , where we assume that  $n/b^{i-1} \geq n_0$  is still sufficiently large.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \leq \sum_{i=0}^{\log_b n - 1} c^i f(n) + \mathcal{O}(n^{\log_b a})$$

$$q < 1 : \sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} \leq \frac{1}{1-q} \leq \frac{1}{1-c} \Rightarrow T(n) \leq \frac{1}{1-c} f(n) + \mathcal{O}(n^{\log_b a})$$

Hence,

$$T(n) \leq \mathcal{O}(f(n)) \Rightarrow T(n) = \Theta(f(n)).$$

Where did we use  $f(n) \geq \Omega(n^{\log_b a + \epsilon})$ ?

## Example: Multiplying Two Integers

Suppose we want to multiply two  $n$ -bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers  $A$  and  $B$ :

$$\begin{array}{r} 110110101 \quad A \\ 100010011 \quad B \\ \hline 10111001000 \end{array}$$

This gives that two  $n$ -bit integers can be added in time  $\mathcal{O}(n)$ .

## Example: Multiplying Two Integers

Suppose that we want to multiply an  $n$ -bit integer  $A$  and an  $m$ -bit integer  $B$  ( $m \leq n$ ).

$$\begin{array}{r} 10001 \times 1011 \\ \hline 10001 \\ 100010 \\ 0000000 \\ 10001000 \\ \hline 10111011 \end{array}$$

- This is also known as the "school method" for multiplying integers.
- Note that the intermediate numbers that are generated can have at most  $m + n \leq 2n$  bits.

Time requirement:

- Computing intermediate results:  $\mathcal{O}(nm)$ .
- Adding  $m$  numbers of length  $\leq 2n$ :  $\mathcal{O}((m+n)m) = \mathcal{O}(nm)$ .

## Example: Multiplying Two Integers

A recursive approach:

Suppose that integers  $A$  and  $B$  are of length  $n = 2^k$ , for some  $k$ .

$$\boxed{B_1} \boxed{B_0} \times \boxed{A_1} \boxed{A_0}$$

Then it holds that

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ and } B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

Hence,

$$A \cdot B = A_1 B_1 \cdot 2^n + (A_1 B_0 + A_0 B_1) \cdot 2^{\frac{n}{2}} + A_0 B_0$$



## Example: Multiplying Two Integers

### Algorithm 3 mult( $A, B$ )

1: <b>if</b> $ A  =  B  = 1$ <b>then</b>	$\mathcal{O}(1)$
2: <b>return</b> $a_0 \cdot b_0$	$\mathcal{O}(1)$
3: split $A$ into $A_0$ and $A_1$	$\mathcal{O}(n)$
4: split $B$ into $B_0$ and $B_1$	$\mathcal{O}(n)$
5: $Z_2 \leftarrow \text{mult}(A_1, B_1)$	$T(\frac{n}{2})$
6: $Z_1 \leftarrow \text{mult}(A_1, B_0) + \text{mult}(A_0, B_1)$	$2T(\frac{n}{2}) + \mathcal{O}(n)$
7: $Z_0 \leftarrow \text{mult}(A_0, B_0)$	$T(\frac{n}{2})$
8: <b>return</b> $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$	$\mathcal{O}(n)$

We get the following recurrence:

$$T(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n) .$$

## Example: Multiplying Two Integers

**Master Theorem:** Recurrence:  $T[n] = aT(\frac{n}{b}) + f(n)$ .

- ▶ Case 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$       $T(n) = \mathcal{O}(n^{\log_b a})$
- ▶ Case 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$     $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- ▶ Case 3:  $f(n) = \Omega(n^{\log_b a + \epsilon})$       $T(n) = \Theta(f(n))$

In our case  $a = 4$ ,  $b = 2$ , and  $f(n) = \mathcal{O}(n)$ . Hence, we are in Case 1, since  $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$ .

We get a running time of  $\mathcal{O}(n^2)$  for our algorithm.

⇒ Not better than the “school method”.

## Example: Multiplying Two Integers

We can use the following identity to compute  $Z_1$ :

$$\begin{aligned} Z_1 &= A_1 B_0 + A_0 B_1 && = Z_2 && = Z_0 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \underbrace{A_1 B_1}_{Z_2} - \underbrace{A_0 B_0}_{Z_0} \end{aligned}$$

Hence,

### Algorithm 4 mult( $A, B$ )

1: <b>if</b> $ A  =  B  = 1$ <b>then</b>	$\mathcal{O}(1)$
2: <b>return</b> $a_0 \cdot b_0$	$\mathcal{O}(1)$
3: split $A$ into $A_0$ and $A_1$	$\mathcal{O}(n)$
4: split $B$ into $B_0$ and $B_1$	$\mathcal{O}(n)$
5: $Z_2 \leftarrow \text{mult}(A_1, B_1)$	$T(\frac{n}{2})$
6: $Z_0 \leftarrow \text{mult}(A_0, B_0)$	$T(\frac{n}{2})$
7: $Z_1 \leftarrow \text{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$	$T(\frac{n}{2}) + \mathcal{O}(n)$
8: <b>return</b> $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$	$\mathcal{O}(n)$

A more precise (correct) analysis would say that computing  $Z_1$  needs time  $T(\frac{n}{2} + 1) + \mathcal{O}(n)$ .

## Example: Multiplying Two Integers

We get the following recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) .$$

**Master Theorem:** Recurrence:  $T[n] = aT(\frac{n}{b}) + f(n)$ .

- ▶ Case 1:  $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$       $T(n) = \mathcal{O}(n^{\log_b a})$
- ▶ Case 2:  $f(n) = \Theta(n^{\log_b a} \log^k n)$     $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- ▶ Case 3:  $f(n) = \Omega(n^{\log_b a + \epsilon})$       $T(n) = \Theta(f(n))$

Again we are in Case 1. We get a running time of  $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$ .

A huge improvement over the “school method”.

## 6.3 The Characteristic Polynomial

Consider the recurrence relation:

$$c_0T(n) + c_1T(n-1) + c_2T(n-2) + \dots + c_kT(n-k) = f(n)$$

This is the general form of a **linear** recurrence relation of **order  $k$**  with constant coefficients ( $c_0, c_k \neq 0$ ).

- ▶  $T(n)$  only depends on the  $k$  preceding values. This means the recurrence relation is of **order  $k$** .
- ▶ The recurrence is linear as there are no products of  $T[n]$ 's.
- ▶ If  $f(n) = 0$  then the recurrence relation becomes a linear, **homogenous** recurrence relation of order  $k$ .

Note that we ignore **boundary conditions** for the moment.



## 6.3 The Characteristic Polynomial

**Observations:**

- ▶ The solution  $T[1], T[2], T[3], \dots$  is completely determined by a set of **boundary conditions** that specify values for  $T[1], \dots, T[k]$ .
- ▶ In fact, any  $k$  consecutive values completely determine the solution.
- ▶  $k$  non-consecutive values might not be an appropriate set of boundary conditions (depends on the problem).

**Approach:**

- ▶ First determine all solutions that satisfy recurrence relation.
- ▶ Then pick the right one by analyzing boundary conditions.
- ▶ First consider the homogenous case.



## The Homogenous Case

The solution space

$$S = \{ \mathcal{T} = T[1], T[2], T[3], \dots \mid \mathcal{T} \text{ fulfills recurrence relation} \}$$

is a **vector space**. This means that if  $\mathcal{T}_1, \mathcal{T}_2 \in S$ , then also  $\alpha\mathcal{T}_1 + \beta\mathcal{T}_2 \in S$ , for arbitrary constants  $\alpha, \beta$ .

**How do we find a non-trivial solution?**

We guess that the solution is of the form  $\lambda^n$ ,  $\lambda \neq 0$ , and see what happens. In order for this guess to fulfill the recurrence we need

$$c_0\lambda^n + c_1\lambda^{n-1} + c_2 \cdot \lambda^{n-2} + \dots + c_k \cdot \lambda^{n-k} = 0$$

for all  $n \geq k$ .



## The Homogenous Case

Dividing by  $\lambda^{n-k}$  gives that all these constraints are identical to

$$\underbrace{c_0\lambda^k + c_1\lambda^{k-1} + c_2 \cdot \lambda^{k-2} + \dots + c_k}_{\text{characteristic polynomial } P[\lambda]} = 0$$

This means that if  $\lambda_i$  is a root (**Nullstelle**) of  $P[\lambda]$  then  $T[n] = \lambda_i^n$  is a solution to the recurrence relation.

Let  $\lambda_1, \dots, \lambda_k$  be the  $k$  (complex) roots of  $P[\lambda]$ . Then, because of the vector space property

$$\alpha_1\lambda_1^n + \alpha_2\lambda_2^n + \dots + \alpha_k\lambda_k^n$$

is a solution for arbitrary values  $\alpha_i$ .



## The Homogenous Case

### Lemma 5

Assume that the characteristic polynomial has  $k$  distinct roots  $\lambda_1, \dots, \lambda_k$ . Then *all* solutions to the recurrence relation are of the form

$$\alpha_1 \lambda_1^n + \alpha_2 \lambda_2^n + \dots + \alpha_k \lambda_k^n .$$

### Proof.

There is one solution for every possible choice of boundary conditions for  $T[1], \dots, T[k]$ .

We show that the above set of solutions contains one solution for every choice of boundary conditions.

## The Homogenous Case

### Proof (cont.).

Suppose I am given boundary conditions  $T[i]$  and I want to see whether I can choose the  $\alpha_i$ 's such that these conditions are met:

$$\alpha_1 \cdot \lambda_1 + \alpha_2 \cdot \lambda_2 + \dots + \alpha_k \cdot \lambda_k = T[1]$$

$$\alpha_1 \cdot \lambda_1^2 + \alpha_2 \cdot \lambda_2^2 + \dots + \alpha_k \cdot \lambda_k^2 = T[2]$$

$$\vdots$$

$$\alpha_1 \cdot \lambda_1^k + \alpha_2 \cdot \lambda_2^k + \dots + \alpha_k \cdot \lambda_k^k = T[k]$$

## The Homogenous Case

### Proof (cont.).

Suppose I am given boundary conditions  $T[i]$  and I want to see whether I can choose the  $\alpha_i$ 's such that these conditions are met:

$$\begin{pmatrix} \lambda_1 & \lambda_2 & \dots & \lambda_k \\ \lambda_1^2 & \lambda_2^2 & \dots & \lambda_k^2 \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_1^k & \lambda_2^k & \dots & \lambda_k^k \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_k \end{pmatrix} = \begin{pmatrix} T[1] \\ T[2] \\ \vdots \\ T[k] \end{pmatrix}$$

We show that the column vectors are linearly independent. Then the above equation has a solution.

## Computing the Determinant

$$\begin{vmatrix} \lambda_1 & \lambda_2 & \dots & \lambda_{k-1} & \lambda_k \\ \lambda_1^2 & \lambda_2^2 & \dots & \lambda_{k-1}^2 & \lambda_k^2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \lambda_1^k & \lambda_2^k & \dots & \lambda_{k-1}^k & \lambda_k^k \end{vmatrix} = \prod_{i=1}^k \lambda_i \cdot \begin{vmatrix} 1 & 1 & \dots & 1 & 1 \\ \lambda_1 & \lambda_2 & \dots & \lambda_{k-1} & \lambda_k \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ \lambda_1^{k-1} & \lambda_2^{k-1} & \dots & \lambda_{k-1}^{k-1} & \lambda_k^{k-1} \end{vmatrix}$$

$$= \prod_{i=1}^k \lambda_i \cdot \begin{vmatrix} 1 & \lambda_1 & \dots & \lambda_1^{k-2} & \lambda_1^{k-1} \\ 1 & \lambda_2 & \dots & \lambda_2^{k-2} & \lambda_2^{k-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & \lambda_k & \dots & \lambda_k^{k-2} & \lambda_k^{k-1} \end{vmatrix}$$

## Computing the Determinant

$$\begin{vmatrix} 1 & \lambda_1 & \cdots & \lambda_1^{k-2} & \lambda_1^{k-1} \\ 1 & \lambda_2 & \cdots & \lambda_2^{k-2} & \lambda_2^{k-1} \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & \lambda_k & \cdots & \lambda_k^{k-2} & \lambda_k^{k-1} \end{vmatrix} =$$

$$\begin{vmatrix} 1 & \lambda_1 - \lambda_1 \cdot 1 & \cdots & \lambda_1^{k-2} - \lambda_1 \cdot \lambda_1^{k-3} & \lambda_1^{k-1} - \lambda_1 \cdot \lambda_1^{k-2} \\ 1 & \lambda_2 - \lambda_1 \cdot 1 & \cdots & \lambda_2^{k-2} - \lambda_1 \cdot \lambda_2^{k-3} & \lambda_2^{k-1} - \lambda_1 \cdot \lambda_2^{k-2} \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & \lambda_k - \lambda_1 \cdot 1 & \cdots & \lambda_k^{k-2} - \lambda_1 \cdot \lambda_k^{k-3} & \lambda_k^{k-1} - \lambda_1 \cdot \lambda_k^{k-2} \end{vmatrix}$$

## Computing the Determinant

$$\begin{vmatrix} 1 & \lambda_1 - \lambda_1 \cdot 1 & \cdots & \lambda_1^{k-2} - \lambda_1 \cdot \lambda_1^{k-3} & \lambda_1^{k-1} - \lambda_1 \cdot \lambda_1^{k-2} \\ 1 & \lambda_2 - \lambda_1 \cdot 1 & \cdots & \lambda_2^{k-2} - \lambda_1 \cdot \lambda_2^{k-3} & \lambda_2^{k-1} - \lambda_1 \cdot \lambda_2^{k-2} \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & \lambda_k - \lambda_1 \cdot 1 & \cdots & \lambda_k^{k-2} - \lambda_1 \cdot \lambda_k^{k-3} & \lambda_k^{k-1} - \lambda_1 \cdot \lambda_k^{k-2} \end{vmatrix} =$$

$$\begin{vmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & (\lambda_2 - \lambda_1) \cdot 1 & \cdots & (\lambda_2 - \lambda_1) \cdot \lambda_2^{k-3} & (\lambda_2 - \lambda_1) \cdot \lambda_2^{k-2} \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & (\lambda_k - \lambda_1) \cdot 1 & \cdots & (\lambda_k - \lambda_1) \cdot \lambda_k^{k-3} & (\lambda_k - \lambda_1) \cdot \lambda_k^{k-2} \end{vmatrix}$$

## Computing the Determinant

$$\begin{vmatrix} 1 & 0 & \cdots & 0 & 0 \\ 1 & (\lambda_2 - \lambda_1) \cdot 1 & \cdots & (\lambda_2 - \lambda_1) \cdot \lambda_2^{k-3} & (\lambda_2 - \lambda_1) \cdot \lambda_2^{k-2} \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & (\lambda_k - \lambda_1) \cdot 1 & \cdots & (\lambda_k - \lambda_1) \cdot \lambda_k^{k-3} & (\lambda_k - \lambda_1) \cdot \lambda_k^{k-2} \end{vmatrix} =$$

$$\prod_{i=2}^k (\lambda_i - \lambda_1) \cdot \begin{vmatrix} 1 & \lambda_2 & \cdots & \lambda_2^{k-3} & \lambda_2^{k-2} \\ \vdots & \vdots & & \vdots & \vdots \\ 1 & \lambda_k & \cdots & \lambda_k^{k-3} & \lambda_k^{k-2} \end{vmatrix}$$

## Computing the Determinant

Repeating the above steps gives:

$$\begin{vmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_{k-1} & \lambda_k \\ \lambda_1^2 & \lambda_2^2 & \cdots & \lambda_{k-1}^2 & \lambda_k^2 \\ \vdots & \vdots & & \vdots & \vdots \\ \lambda_1^k & \lambda_2^k & \cdots & \lambda_{k-1}^k & \lambda_k^k \end{vmatrix} = \prod_{i=1}^k \lambda_i \cdot \prod_{i>\ell} (\lambda_i - \lambda_\ell)$$

Hence, if all  $\lambda_i$ 's are different, then the determinant is non-zero.

## The Homogeneous Case

### What happens if the roots are not all distinct?

Suppose we have a root  $\lambda_i$  with multiplicity (**Vielfachheit**) at least 2. Then not only is  $\lambda_i^n$  a solution to the recurrence but also  $n\lambda_i^n$ .

To see this consider the polynomial

$$P[\lambda] \cdot \lambda^{n-k} = c_0\lambda^n + c_1\lambda^{n-1} + c_2\lambda^{n-2} + \dots + c_k\lambda^{n-k}$$

Since  $\lambda_i$  is a root we can write this as  $Q[\lambda] \cdot (\lambda - \lambda_i)^2$ .

Calculating the derivative gives a polynomial that still has root  $\lambda_i$ .

This means

$$c_0n\lambda_i^{n-1} + c_1(n-1)\lambda_i^{n-2} + \dots + c_k(n-k)\lambda_i^{n-k-1} = 0$$

Hence,

$$\underbrace{c_0n\lambda_i^n}_{T[n]} + \underbrace{c_1(n-1)\lambda_i^{n-1}}_{T[n-1]} + \dots + \underbrace{c_k(n-k)\lambda_i^{n-k}}_{T[n-k]} = 0$$

## The Homogeneous Case

Suppose  $\lambda_i$  has multiplicity  $j$ . We know that

$$c_0n\lambda_i^n + c_1(n-1)\lambda_i^{n-1} + \dots + c_k(n-k)\lambda_i^{n-k} = 0$$

(after taking the derivative; multiplying with  $\lambda$ ; plugging in  $\lambda_i$ )

Doing this again gives

$$c_0n^2\lambda_i^n + c_1(n-1)^2\lambda_i^{n-1} + \dots + c_k(n-k)^2\lambda_i^{n-k} = 0$$

We can continue  $j-1$  times.

Hence,  $n^\ell\lambda_i^n$  is a solution for  $\ell \in 0, \dots, j-1$ .

## The Homogeneous Case

### Lemma 6

Let  $P[\lambda]$  denote the characteristic polynomial to the recurrence

$$c_0T[n] + c_1T[n-1] + \dots + c_kT[n-k] = 0$$

Let  $\lambda_i, i = 1, \dots, m$  be the (complex) roots of  $P[\lambda]$  with multiplicities  $\ell_i$ . Then the general solution to the recurrence is given by

$$T[n] = \sum_{i=1}^m \sum_{j=0}^{\ell_i-1} \alpha_{ij} \cdot (n^j \lambda_i^n) .$$

The full proof is omitted. We have only shown that any choice of  $\alpha_{ij}$ 's is a solution to the recurrence.

## Example: Fibonacci Sequence

$$T[0] = 0$$

$$T[1] = 1$$

$$T[n] = T[n-1] + T[n-2] \text{ for } n \geq 2$$

The characteristic polynomial is

$$\lambda^2 - \lambda - 1$$

Finding the roots, gives

$$\lambda_{1/2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} + 1} = \frac{1}{2} (1 \pm \sqrt{5})$$

## Example: Fibonacci Sequence

Hence, the solution is of the form

$$\alpha \left( \frac{1 + \sqrt{5}}{2} \right)^n + \beta \left( \frac{1 - \sqrt{5}}{2} \right)^n$$

$$T[0] = 0 \text{ gives } \alpha + \beta = 0.$$

$$T[1] = 1 \text{ gives}$$

$$\alpha \left( \frac{1 + \sqrt{5}}{2} \right) + \beta \left( \frac{1 - \sqrt{5}}{2} \right) = 1 \Rightarrow \alpha - \beta = \frac{2}{\sqrt{5}}$$

## Example: Fibonacci Sequence

Hence, the solution is

$$\frac{1}{\sqrt{5}} \left[ \left( \frac{1 + \sqrt{5}}{2} \right)^n - \left( \frac{1 - \sqrt{5}}{2} \right)^n \right]$$

## The Inhomogeneous Case

Consider the recurrence relation:

$$c_0 T(n) + c_1 T(n-1) + c_2 T(n-2) + \dots + c_k T(n-k) = f(n)$$

with  $f(n) \neq 0$ .

While we have a fairly general technique for solving **homogeneous**, linear recurrence relations the inhomogeneous case is different.

## The Inhomogeneous Case

The general solution of the recurrence relation is

$$T(n) = T_h(n) + T_p(n) ,$$

where  $T_h$  is **any** solution to the homogeneous equation, and  $T_p$  is **one** particular solution to the inhomogeneous equation.

There is no general method to find a particular solution.



## The Inhomogeneous Case

Example:

$$T[n] = T[n - 1] + 1 \quad T[0] = 1$$

Then,

$$T[n - 1] = T[n - 2] + 1 \quad (n \geq 2)$$

Subtracting the first from the second equation gives,

$$T[n] - T[n - 1] = T[n - 1] - T[n - 2] \quad (n \geq 2)$$

or

$$T[n] = 2T[n - 1] - T[n - 2] \quad (n \geq 2)$$

I get a completely determined recurrence if I add  $T[0] = 1$  and  $T[1] = 2$ .



## The Inhomogeneous Case

Example: Characteristic polynomial:

$$\underbrace{\lambda^2 - 2\lambda + 1}_{(\lambda-1)^2} = 0$$

Then the solution is of the form

$$T[n] = \alpha 1^n + \beta n 1^n = \alpha + \beta n$$

$T[0] = 1$  gives  $\alpha = 1$ .

$T[1] = 2$  gives  $1 + \beta = 2 \Rightarrow \beta = 1$ .



## The Inhomogeneous Case

If  $f(n)$  is a polynomial of degree  $r$  this method can be applied  $r + 1$  times to obtain a homogeneous equation:

$$T[n] = T[n - 1] + n^2$$

Shift:

$$T[n - 1] = T[n - 2] + (n - 1)^2 = T[n - 2] + n^2 - 2n + 1$$

Difference:

$$T[n] - T[n - 1] = T[n - 1] - T[n - 2] + 2n - 1$$

$$T[n] = 2T[n - 1] - T[n - 2] + 2n - 1$$

$$T[n] = 2T[n - 1] - T[n - 2] + 2n - 1$$

Shift:

$$\begin{aligned} T[n - 1] &= 2T[n - 2] - T[n - 3] + 2(n - 1) - 1 \\ &= 2T[n - 2] - T[n - 3] + 2n - 3 \end{aligned}$$

Difference:

$$\begin{aligned} T[n] - T[n - 1] &= 2T[n - 1] - T[n - 2] + 2n - 1 \\ &\quad - 2T[n - 2] + T[n - 3] - 2n + 3 \end{aligned}$$

$$T[n] = 3T[n - 1] - 3T[n - 2] + T[n - 3] + 2$$

and so on...

## 6.4 Generating Functions

### Definition 7 (Generating Function)

Let  $(a_n)_{n \geq 0}$  be a sequence. The corresponding

- ▶ **generating function** (Erzeugendenfunktion) is

$$F(z) := \sum_{n \geq 0} a_n z^n;$$

- ▶ **exponential generating function** (exponentielle Erzeugendenfunktion) is

$$F(z) := \sum_{n \geq 0} \frac{a_n}{n!} z^n.$$



## 6.4 Generating Functions

### Example 8

1. The generating function of the sequence  $(1, 0, 0, \dots)$  is

$$F(z) = 1.$$

2. The generating function of the sequence  $(1, 1, 1, \dots)$  is

$$F(z) = \frac{1}{1 - z}.$$



## 6.4 Generating Functions

There are two different views:

A generating function is a **formal power series** (formale Potenzreihe).

Then the generating function is an **algebraic object**.

Let  $f = \sum_{n \geq 0} a_n z^n$  and  $g = \sum_{n \geq 0} b_n z^n$ .

- ▶ **Equality:**  $f$  and  $g$  are equal if  $a_n = b_n$  for all  $n$ .
- ▶ **Addition:**  $f + g := \sum_{n \geq 0} (a_n + b_n) z^n$ .
- ▶ **Multiplication:**  $f \cdot g := \sum_{n \geq 0} c_n z^n$  with  $c_n = \sum_{p=0}^n a_p b_{n-p}$ .

There are no convergence issues here.





## 6.4 Generating Functions

The arithmetic view:

We view a power series as a function  $f: \mathbb{C} \rightarrow \mathbb{C}$ .

Then, it is important to think about convergence/convergence radius etc.

## 6.4 Generating Functions

What does  $\sum_{n \geq 0} z^n = \frac{1}{1-z}$  mean in the algebraic view?

It means that the power series  $1 - z$  and the power series  $\sum_{n \geq 0} z^n$  are inverses, i.e.,

$$(1 - z) \cdot \left( \sum_{n \geq 0} z^n \right) = 1.$$

This is well-defined.

## 6.4 Generating Functions

Suppose we are given the generating function

$$\sum_{n \geq 0} z^n = \frac{1}{1-z}.$$

We can compute the derivative:

$$\underbrace{\sum_{n \geq 1} n z^{n-1}}_{\sum_{n \geq 0} (n+1) z^n} = \frac{1}{(1-z)^2}$$

Hence, the generating function of the sequence  $a_n = n + 1$  is  $1/(1-z)^2$ .

Formally the derivative of a formal power series  $\sum_{n \geq 0} a_n z^n$  is defined as  $\sum_{n \geq 0} n a_n z^{n-1}$ .

The known rules for differentiation work for this definition. In particular, e.g. the derivative of  $\frac{1}{1-z}$  is  $\frac{1}{(1-z)^2}$ .

Note that this requires a proof if we consider power series as algebraic objects. However, we did not prove this in the lecture.

## 6.4 Generating Functions

We can repeat this

$$\sum_{n \geq 0} (n+1) z^n = \frac{1}{(1-z)^2}.$$

Derivative:

$$\underbrace{\sum_{n \geq 1} n(n+1) z^{n-1}}_{\sum_{n \geq 0} (n+1)(n+2) z^n} = \frac{2}{(1-z)^3}$$

Hence, the generating function of the sequence  $a_n = (n+1)(n+2)$  is  $\frac{2}{(1-z)^3}$ .

## 6.4 Generating Functions

Computing the  $k$ -th derivative of  $\sum z^n$ .

$$\begin{aligned}\sum_{n \geq k} n(n-1) \cdots (n-k+1)z^{n-k} &= \sum_{n \geq 0} (n+k) \cdots (n+1)z^n \\ &= \frac{k!}{(1-z)^{k+1}}.\end{aligned}$$

Hence:

$$\sum_{n \geq 0} \binom{n+k}{k} z^n = \frac{1}{(1-z)^{k+1}}.$$

The generating function of the sequence  $a_n = \binom{n+k}{k}$  is  $\frac{1}{(1-z)^{k+1}}$ .



## 6.4 Generating Functions

$$\begin{aligned}\sum_{n \geq 0} nz^n &= \sum_{n \geq 0} (n+1)z^n - \sum_{n \geq 0} z^n \\ &= \frac{1}{(1-z)^2} - \frac{1}{1-z} \\ &= \frac{z}{(1-z)^2}\end{aligned}$$

The generating function of the sequence  $a_n = n$  is  $\frac{z}{(1-z)^2}$ .



## 6.4 Generating Functions

We know

$$\sum_{n \geq 0} y^n = \frac{1}{1-y}$$

Hence,

$$\sum_{n \geq 0} a^n z^n = \frac{1}{1-az}$$

The generating function of the sequence  $f_n = a^n$  is  $\frac{1}{1-az}$ .



## Example: $a_n = a_{n-1} + 1, a_0 = 1$

Suppose we have the recurrence  $a_n = a_{n-1} + 1$  for  $n \geq 1$  and  $a_0 = 1$ .

$$\begin{aligned}A(z) &= \sum_{n \geq 0} a_n z^n \\ &= a_0 + \sum_{n \geq 1} (a_{n-1} + 1)z^n \\ &= 1 + z \sum_{n \geq 1} a_{n-1} z^{n-1} + \sum_{n \geq 1} z^n \\ &= z \sum_{n \geq 0} a_n z^n + \sum_{n \geq 0} z^n \\ &= zA(z) + \sum_{n \geq 0} z^n \\ &= zA(z) + \frac{1}{1-z}\end{aligned}$$



## Example: $a_n = a_{n-1} + 1, a_0 = 1$

Solving for  $A(z)$  gives

$$\sum_{n \geq 0} a_n z^n = A(z) = \frac{1}{(1-z)^2} = \sum_{n \geq 0} (n+1)z^n$$

Hence,  $a_n = n + 1$ .



## Some Generating Functions

<i>n</i> -th sequence element	generating function
1	$\frac{1}{1-z}$
$n + 1$	$\frac{1}{(1-z)^2}$
$\binom{n+k}{k}$	$\frac{1}{(1-z)^{k+1}}$
$n$	$\frac{z}{(1-z)^2}$
$a^n$	$\frac{1}{1-az}$
$n^2$	$\frac{z(1+z)}{(1-z)^3}$
$\frac{1}{n!}$	$e^z$



## Some Generating Functions

<i>n</i> -th sequence element	generating function
$cf_n$	$cF$
$f_n + g_n$	$F + G$
$\sum_{i=0}^n f_i g_{n-i}$	$F \cdot G$
$f_{n-k} \ (n \geq k); \ 0 \text{ otw.}$	$z^k F$
$\sum_{i=0}^n f_i$	$\frac{F(z)}{1-z}$
$nf_n$	$z \frac{dF(z)}{dz}$
$c^n f_n$	$F(cz)$



## Solving Recursions with Generating Functions

1. Set  $A(z) = \sum_{n \geq 0} a_n z^n$ .
2. Transform the right hand side so that boundary condition and recurrence relation can be plugged in.
3. Do further transformations so that the infinite sums on the right hand side can be replaced by  $A(z)$ .
4. Solving for  $A(z)$  gives an equation of the form  $A(z) = f(z)$ , where hopefully  $f(z)$  is a simple function.
5. Write  $f(z)$  as a formal power series.  
Techniques:
  - ▶ partial fraction decomposition (Partialbruchzerlegung)
  - ▶ lookup in tables
6. The coefficients of the resulting power series are the  $a_n$ .



**Example:  $a_n = 2a_{n-1}, a_0 = 1$**

1. Set up generating function:

$$A(z) = \sum_{n \geq 0} a_n z^n$$

2. Transform right hand side so that recurrence can be plugged in:

$$A(z) = a_0 + \sum_{n \geq 1} a_n z^n$$

2. Plug in:

$$A(z) = 1 + \sum_{n \geq 1} (2a_{n-1})z^n$$

**Example:  $a_n = 2a_{n-1}, a_0 = 1$**

3. Transform right hand side so that infinite sums can be replaced by  $A(z)$  or by simple function.

$$\begin{aligned} A(z) &= 1 + \sum_{n \geq 1} (2a_{n-1})z^n \\ &= 1 + 2z \sum_{n \geq 1} a_{n-1}z^{n-1} \\ &= 1 + 2z \sum_{n \geq 0} a_n z^n \\ &= 1 + 2z \cdot A(z) \end{aligned}$$

4. Solve for  $A(z)$ .

$$A(z) = \frac{1}{1 - 2z}$$

**Example:  $a_n = 2a_{n-1}, a_0 = 1$**

5. Rewrite  $f(z)$  as a power series:

$$\sum_{n \geq 0} a_n z^n = A(z) = \frac{1}{1 - 2z} = \sum_{n \geq 0} 2^n z^n$$

**Example:  $a_n = 3a_{n-1} + n, a_0 = 1$**

1. Set up generating function:

$$A(z) = \sum_{n \geq 0} a_n z^n$$

**Example:  $a_n = 3a_{n-1} + n, a_0 = 1$**

2./3. Transform right hand side:

$$\begin{aligned} A(z) &= \sum_{n \geq 0} a_n z^n \\ &= a_0 + \sum_{n \geq 1} a_n z^n \\ &= 1 + \sum_{n \geq 1} (3a_{n-1} + n)z^n \\ &= 1 + 3z \sum_{n \geq 1} a_{n-1} z^{n-1} + \sum_{n \geq 1} n z^n \\ &= 1 + 3z \sum_{n \geq 0} a_n z^n + \sum_{n \geq 0} n z^n \\ &= 1 + 3zA(z) + \frac{z}{(1-z)^2} \end{aligned}$$

**Example:  $a_n = 3a_{n-1} + n, a_0 = 1$**

4. Solve for  $A(z)$ :

$$A(z) = 1 + 3zA(z) + \frac{z}{(1-z)^2}$$

gives

$$A(z) = \frac{(1-z)^2 + z}{(1-3z)(1-z)^2} = \frac{z^2 - z + 1}{(1-3z)(1-z)^2}$$

**Example:  $a_n = 3a_{n-1} + n, a_0 = 1$**

5. Write  $f(z)$  as a formal power series:

We use partial fraction decomposition:

$$\frac{z^2 - z + 1}{(1-3z)(1-z)^2} \stackrel{!}{=} \frac{A}{1-3z} + \frac{B}{1-z} + \frac{C}{(1-z)^2}$$

This gives

$$\begin{aligned} z^2 - z + 1 &= A(1-z)^2 + B(1-3z)(1-z) + C(1-3z) \\ &= A(1-2z+z^2) + B(1-4z+3z^2) + C(1-3z) \\ &= (A+3B)z^2 + (-2A-4B-3C)z + (A+B+C) \end{aligned}$$

**Example:  $a_n = 3a_{n-1} + n, a_0 = 1$**

5. Write  $f(z)$  as a formal power series:

This leads to the following conditions:

$$\begin{aligned} A + B + C &= 1 \\ 2A + 4B + 3C &= 1 \\ A + 3B &= 1 \end{aligned}$$

which gives

$$A = \frac{7}{4} \quad B = -\frac{1}{4} \quad C = -\frac{1}{2}$$

### Example: $a_n = 3a_{n-1} + n, a_0 = 1$

5. Write  $f(z)$  as a formal power series:

$$\begin{aligned} A(z) &= \frac{7}{4} \cdot \frac{1}{1-3z} - \frac{1}{4} \cdot \frac{1}{1-z} - \frac{1}{2} \cdot \frac{1}{(1-z)^2} \\ &= \frac{7}{4} \cdot \sum_{n \geq 0} 3^n z^n - \frac{1}{4} \cdot \sum_{n \geq 0} z^n - \frac{1}{2} \cdot \sum_{n \geq 0} (n+1)z^n \\ &= \sum_{n \geq 0} \left( \frac{7}{4} \cdot 3^n - \frac{1}{4} - \frac{1}{2}(n+1) \right) z^n \\ &= \sum_{n \geq 0} \left( \frac{7}{4} \cdot 3^n - \frac{1}{2}n - \frac{3}{4} \right) z^n \end{aligned}$$

6. This means  $a_n = \frac{7}{4}3^n - \frac{1}{2}n - \frac{3}{4}$ .



## 6.5 Transformation of the Recurrence

### Example 9

$$f_0 = 1$$

$$f_1 = 2$$

$$f_n = f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2.$$

Define

$$g_n := \log f_n.$$

Then

$$g_n = g_{n-1} + g_{n-2} \text{ for } n \geq 2$$

$$g_1 = \log 2 = 1 (\text{for } \log = \log_2), g_0 = 0$$

$$g_n = F_n \text{ (} n\text{-th Fibonacci number)}$$

$$f_n = 2^{F_n}$$



## 6.5 Transformation of the Recurrence

### Example 10

$$f_1 = 1$$

$$f_n = 3f_{\frac{n}{2}} + n; \text{ for } n = 2^k, k \geq 1;$$

Define

$$g_k := f_{2^k}.$$

Then:

$$g_0 = 1$$

$$g_k = 3g_{k-1} + 2^k, k \geq 1$$



## 6 Recurrences

We get

$$\begin{aligned} g_k &= 3[g_{k-1}] + 2^k \\ &= 3[3g_{k-2} + 2^{k-1}] + 2^k \\ &= 3^2[g_{k-2}] + 3 \cdot 2^{k-1} + 2^k \\ &= 3^2[3g_{k-3} + 2^{k-2}] + 3 \cdot 2^{k-1} + 2^k \\ &= 3^3g_{k-3} + 3^2 \cdot 2^{k-2} + 3 \cdot 2^{k-1} + 2^k \\ &= 2^k \cdot \sum_{i=0}^k \left(\frac{3}{2}\right)^i \\ &= 2^k \cdot \frac{\left(\frac{3}{2}\right)^{k+1} - 1}{1/2} = 3^{k+1} - 2^{k+1} \end{aligned}$$



## 6 Recurrences

Let  $n = 2^k$ :

$$\begin{aligned}g_k &= 3^{k+1} - 2^{k+1}, \text{ hence} \\f_n &= 3 \cdot 3^k - 2 \cdot 2^k \\&= 3(2^{\log 3})^k - 2 \cdot 2^k \\&= 3(2^k)^{\log 3} - 2 \cdot 2^k \\&= 3n^{\log 3} - 2n .\end{aligned}$$

## 6 Recurrences

### Bibliography

- [MS08] Kurt Mehlhorn, Peter Sanders:  
*Algorithms and Data Structures — The Basic Toolbox*,  
Springer, 2008
- [CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:  
*Introduction to algorithms (3rd ed.)*,  
MIT Press and McGraw-Hill, 2009
- [Liu85] Chung Laung Liu:  
*Elements of Discrete Mathematics*  
McGraw-Hill, 1985

The Karatsuba method can be found in [MS08] Chapter 1. Chapter 4.3 of [CLRS90] covers the "Substitution method" which roughly corresponds to "Guessing+induction". Chapters 4.4, 4.5, 4.6 of this book cover the master theorem. Methods using the characteristic polynomial and generating functions can be found in [Liu85] Chapter 10.

## Part III

## Data Structures

## Abstract Data Type

An abstract data type (ADT) is defined by an interface of operations or methods that can be performed and that have a defined behavior.

The data types in this lecture all operate on objects that are represented by a [key, value] pair.

- ▶ The **key** comes from a totally ordered set, and we assume that there is an efficient comparison function.
- ▶ The **value** can be anything; it usually carries satellite information important for the application that uses the ADT.

## Dynamic Set Operations

- ▶  **$S$ .search( $k$ )**: Returns pointer to object  $x$  from  $S$  with  $\text{key}[x] = k$  or **null**.
- ▶  **$S$ .insert( $x$ )**: Inserts object  $x$  into set  $S$ .  $\text{key}[x]$  must not currently exist in the data-structure.
- ▶  **$S$ .delete( $x$ )**: Given pointer to object  $x$  from  $S$ , delete  $x$  from the set.
- ▶  **$S$ .minimum()**: Return pointer to object with smallest key-value in  $S$ .
- ▶  **$S$ .maximum()**: Return pointer to object with largest key-value in  $S$ .
- ▶  **$S$ .successor( $x$ )**: Return pointer to the next larger element in  $S$  or **null** if  $x$  is maximum.
- ▶  **$S$ .predecessor( $x$ )**: Return pointer to the next smaller element in  $S$  or **null** if  $x$  is minimum.

## Dynamic Set Operations

- ▶  **$S$ .union( $S'$ )**: Sets  $S := S \cup S'$ . The set  $S'$  is destroyed.
- ▶  **$S$ .merge( $S'$ )**: Sets  $S := S \cup S'$ . Requires  $S \cap S' = \emptyset$ .
- ▶  **$S$ .split( $k, S'$ )**:  
 $S := \{x \in S \mid \text{key}[x] \leq k\}$ ,  $S' := \{x \in S \mid \text{key}[x] > k\}$ .
- ▶  **$S$ .concatenate( $S'$ )**:  $S := S \cup S'$ .  
Requires  $\text{key}[S.\text{maximum}()] \leq \text{key}[S'.\text{minimum}()]$ .
  
- ▶  **$S$ .decrease-key( $x, k$ )**: Replace  $\text{key}[x]$  by  $k \leq \text{key}[x]$ .

## Examples of ADTs

### Stack:

- ▶  **$S$ .push( $x$ )**: Insert an element.
- ▶  **$S$ .pop()**: Return the element from  $S$  that was inserted most recently; delete it from  $S$ .
- ▶  **$S$ .empty()**: Tell if  $S$  contains any object.

### Queue:

- ▶  **$S$ .enqueue( $x$ )**: Insert an element.
- ▶  **$S$ .dequeue()**: Return the element that is longest in the structure; delete it from  $S$ .
- ▶  **$S$ .empty()**: Tell if  $S$  contains any object.

### Priority-Queue:

- ▶  **$S$ .insert( $x$ )**: Insert an element.
- ▶  **$S$ .delete-min()**: Return the element with lowest key-value; delete it from  $S$ .

## 7 Dictionary

### Dictionary:

- ▶  **$S$ .insert( $x$ )**: Insert an element  $x$ .
- ▶  **$S$ .delete( $x$ )**: Delete the element pointed to by  $x$ .
- ▶  **$S$ .search( $k$ )**: Return a pointer to an element  $e$  with  $\text{key}[e] = k$  in  $S$  if it exists; otherwise return **null**.

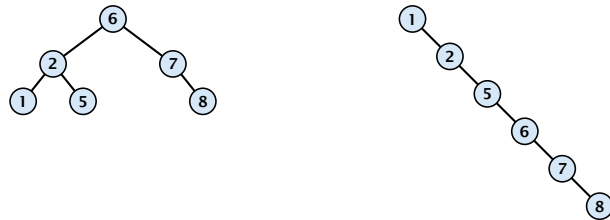


## 7.1 Binary Search Trees

An (internal) **binary search tree** stores the elements in a binary tree. Each tree-node corresponds to an element. All elements in the left sub-tree of a node  $v$  have a smaller key-value than  $\text{key}[v]$  and elements in the right sub-tree have a larger-key value. We assume that all key-values are different.

(External Search Trees store objects only at leaf-vertices)

Examples:



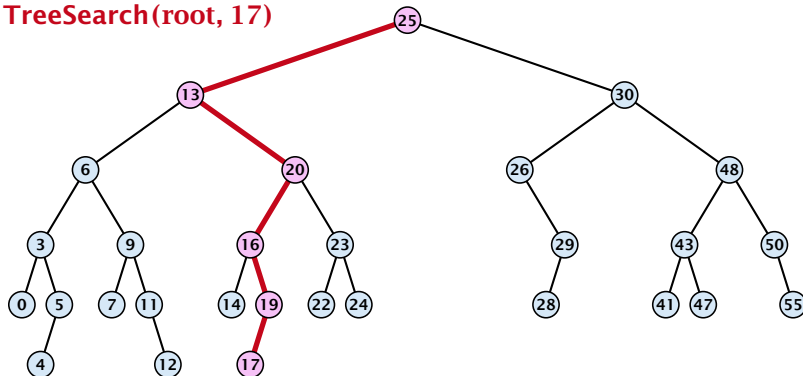
## 7.1 Binary Search Trees

We consider the following operations on binary search trees. Note that this is a super-set of the dictionary-operations.

- ▶  $T.\text{insert}(x)$
- ▶  $T.\text{delete}(x)$
- ▶  $T.\text{search}(k)$
- ▶  $T.\text{successor}(x)$
- ▶  $T.\text{predecessor}(x)$
- ▶  $T.\text{minimum}()$
- ▶  $T.\text{maximum}()$

## Binary Search Trees: Searching

**TreeSearch(root, 17)**

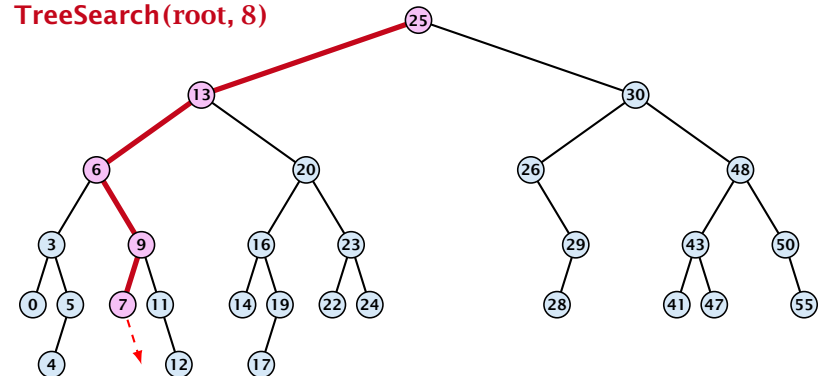


**Algorithm 1**  $\text{TreeSearch}(x, k)$

- 1: **if**  $x = \text{null}$  **or**  $k = \text{key}[x]$  **return**  $x$
- 2: **if**  $k < \text{key}[x]$  **return**  $\text{TreeSearch}(\text{left}[x], k)$
- 3: **else return**  $\text{TreeSearch}(\text{right}[x], k)$

## Binary Search Trees: Searching

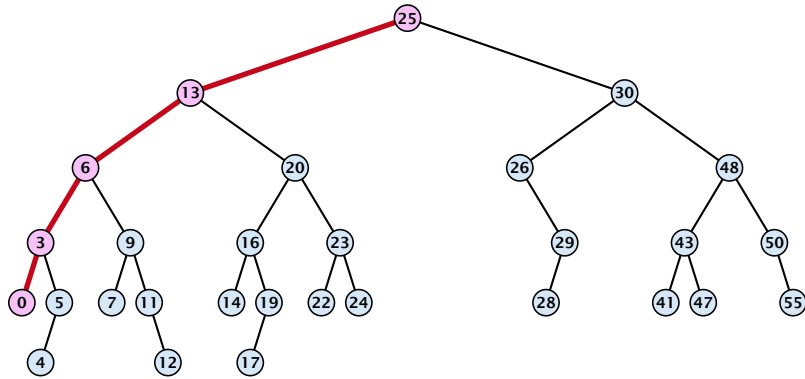
**TreeSearch(root, 8)**



**Algorithm 1**  $\text{TreeSearch}(x, k)$

- 1: **if**  $x = \text{null}$  **or**  $k = \text{key}[x]$  **return**  $x$
- 2: **if**  $k < \text{key}[x]$  **return**  $\text{TreeSearch}(\text{left}[x], k)$
- 3: **else return**  $\text{TreeSearch}(\text{right}[x], k)$

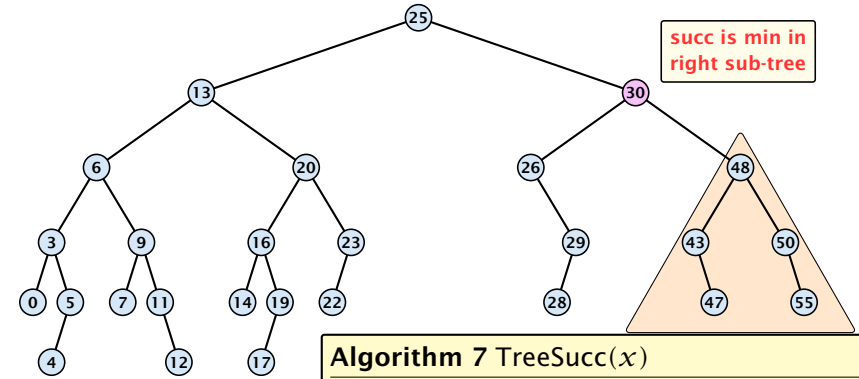
## Binary Search Trees: Minimum



### Algorithm 2 TreeMin(x)

- 1: if  $x = \text{null}$  or  $\text{left}[x] = \text{null}$  return  $x$
- 2: return  $\text{TreeMin}(\text{left}[x])$

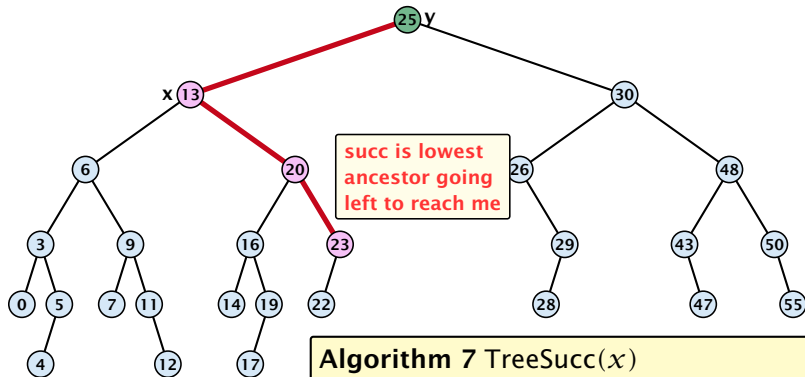
## Binary Search Trees: Successor



### Algorithm 7 TreeSucc(x)

- 1: if  $\text{right}[x] \neq \text{null}$  return  $\text{TreeMin}(\text{right}[x])$
- 2:  $y \leftarrow \text{parent}[x]$
- 3: while  $y \neq \text{null}$  and  $x = \text{right}[y]$  do
- 4:      $x \leftarrow y; y \leftarrow \text{parent}[x]$
- 5: return  $y$ ;

## Binary Search Trees: Successor



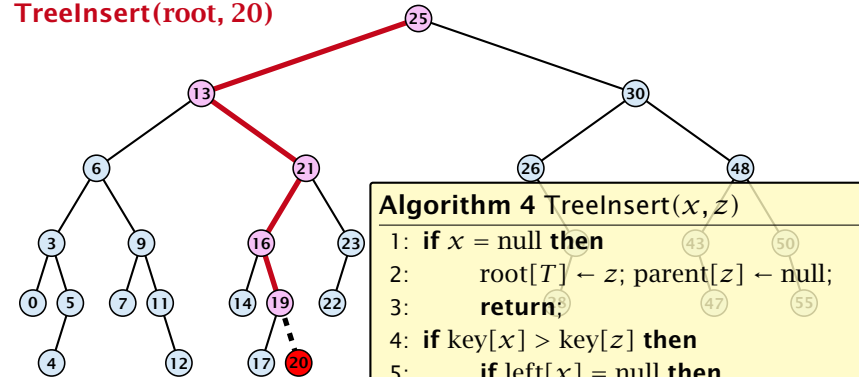
### Algorithm 7 TreeSucc(x)

- 1: if  $\text{right}[x] \neq \text{null}$  return  $\text{TreeMin}(\text{right}[x])$
- 2:  $y \leftarrow \text{parent}[x]$
- 3: while  $y \neq \text{null}$  and  $x = \text{right}[y]$  do
- 4:      $x \leftarrow y; y \leftarrow \text{parent}[x]$
- 5: return  $y$ ;

## Binary Search Trees: Insert

Insert element **not** in the tree.

**TreeInsert(root, 20)**

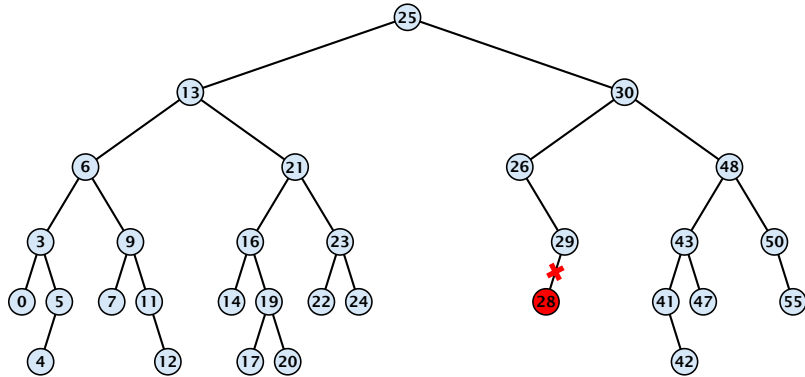


Search for  $z$ . At some point the search stops at a null-pointer. This is the place to insert  $z$ .

### Algorithm 4 TreeInsert(x, z)

- 1: if  $x = \text{null}$  then
- 2:      $\text{root}[T] \leftarrow z; \text{parent}[z] \leftarrow \text{null};$
- 3:     return;
- 4: if  $\text{key}[x] > \text{key}[z]$  then
- 5:     if  $\text{left}[x] = \text{null}$  then
- 6:          $\text{left}[x] \leftarrow z; \text{parent}[z] \leftarrow x;$
- 7:     else  $\text{TreeInsert}(\text{left}[x], z);$
- 8: else
- 9:     if  $\text{right}[x] = \text{null}$  then
- 10:          $\text{right}[x] \leftarrow z; \text{parent}[z] \leftarrow x;$
- 11:     else  $\text{TreeInsert}(\text{right}[x], z);$

## Binary Search Trees: Delete

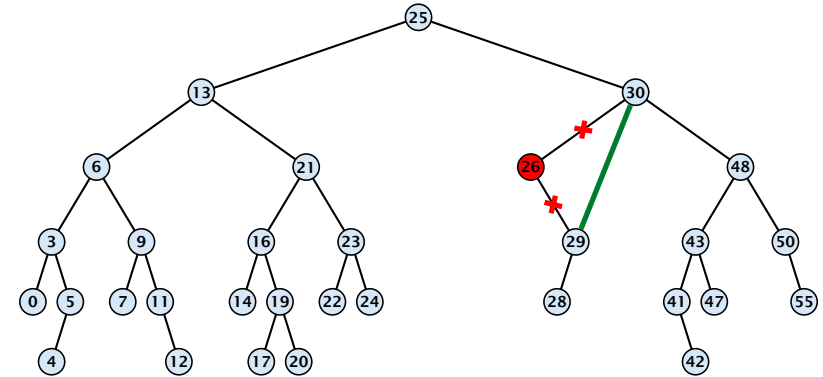


### Case 1:

Element does not have any children

- Simply go to the parent and set the corresponding pointer to **null**.

## Binary Search Trees: Delete

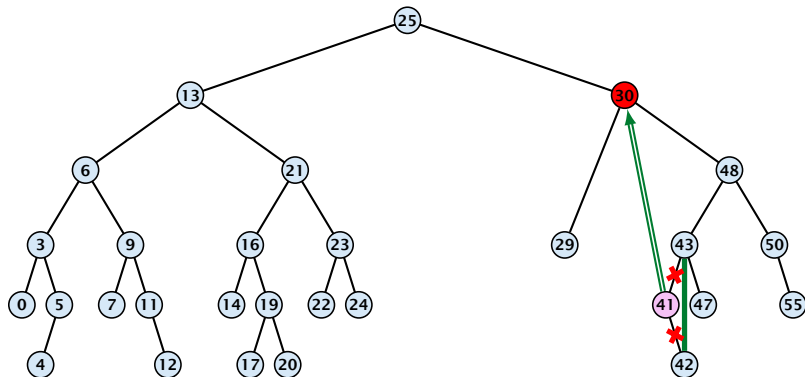


### Case 2:

Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.

## Binary Search Trees: Delete



### Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

## Binary Search Trees: Delete

### Algorithm 9 TreeDelete( $z$ )

```

1: if left[z] = null or right[z] = null
2:   then  $y \leftarrow z$  else  $y \leftarrow \text{TreeSucc}(z)$ ;   select  $y$  to splice out
3: if left[y]  $\neq$  null
4:   then  $x \leftarrow \text{left}[y]$  else  $x \leftarrow \text{right}[y]$ ;  $x$  is child of  $y$  (or null)
5: if  $x \neq \text{null}$  then parent[x]  $\leftarrow$  parent[y];   parent[x] is correct
6: if parent[y] = null then
7:   root[T]  $\leftarrow$  x
8: else
9:   if  $y = \text{left}[\text{parent}[y]]$  then
10:    left[parent[y]]  $\leftarrow$  x
11:   else
12:    right[parent[y]]  $\leftarrow$  x
13: if  $y \neq z$  then copy  $y$ -data to  $z$ 
    
```

## Balanced Binary Search Trees

All operations on a binary search tree can be performed in time  $\mathcal{O}(h)$ , where  $h$  denotes the height of the tree.

However the height of the tree may become as large as  $\Theta(n)$ .

### Balanced Binary Search Trees

With each insert- and delete-operation perform **local** adjustments to guarantee a height of  $\mathcal{O}(\log n)$ .

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

## Binary Search Trees (BSTs)

### Bibliography

[MS08] Kurt Mehlhorn, Peter Sanders:  
*Algorithms and Data Structures — The Basic Toolbox*,  
Springer, 2008

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:  
*Introduction to Algorithms (3rd ed.)*,  
MIT Press and McGraw-Hill, 2009

Binary search trees can be found in every standard text book. For example Chapter 7.1 in [MS08] and Chapter 12 in [CLRS90].

## 7.2 Red Black Trees

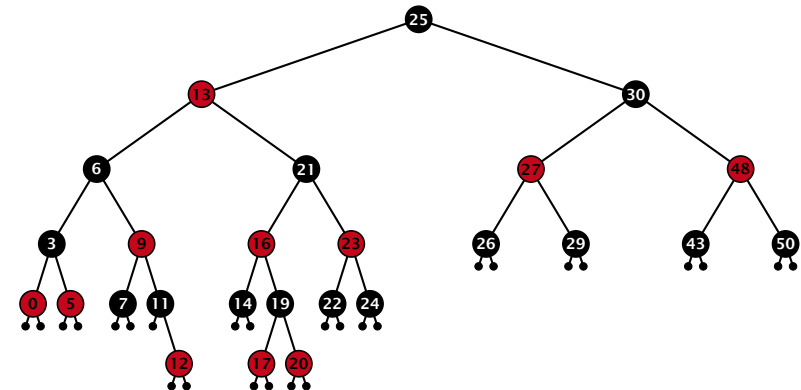
### Definition 11

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The **null**-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

## Red Black Trees: Example



## 7.2 Red Black Trees

### Lemma 12

A red-black tree with  $n$  internal nodes has height at most  $\mathcal{O}(\log n)$ .

### Definition 13

The **black height**  $\text{bh}(v)$  of a node  $v$  in a red black tree is the number of black nodes on a path from  $v$  to a leaf vertex (not counting  $v$ ).

We first show:

### Lemma 14

A sub-tree of black height  $\text{bh}(v)$  in a red black tree contains at least  $2^{\text{bh}(v)} - 1$  internal vertices.

## 7.2 Red Black Trees

### Proof of Lemma 14.

#### Induction on the height of $v$ .

#### base case ( $\text{height}(v) = 0$ )

- ▶ If  $\text{height}(v)$  (maximum distance btw.  $v$  and a node in the sub-tree rooted at  $v$ ) is 0 then  $v$  is a leaf.
- ▶ The black height of  $v$  is 0.
- ▶ The sub-tree rooted at  $v$  contains  $0 = 2^{\text{bh}(v)} - 1$  inner vertices.

## 7.2 Red Black Trees

### Proof (cont.)

#### induction step

- ▶ Suppose  $v$  is a node with  $\text{height}(v) > 0$ .
- ▶  $v$  has **two** children with strictly smaller height.
- ▶ These children ( $c_1, c_2$ ) either have  $\text{bh}(c_i) = \text{bh}(v)$  or  $\text{bh}(c_i) = \text{bh}(v) - 1$ .
- ▶ By induction hypothesis both sub-trees contain at least  $2^{\text{bh}(v)-1} - 1$  internal vertices.
- ▶ Then  $T_v$  contains at least  $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$  vertices.

□

## 7.2 Red Black Trees

### Proof of Lemma 12.

Let  $h$  denote the height of the red-black tree, and let  $P$  denote a path from the root to the furthest leaf.

At least half of the node on  $P$  must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least  $h/2$ .

The tree contains at least  $2^{h/2} - 1$  internal vertices. Hence,  $2^{h/2} - 1 \leq n$ .

Hence,  $h \leq 2 \log(n + 1) = \mathcal{O}(\log n)$ .

□

## 7.2 Red Black Trees

### Definition 1

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

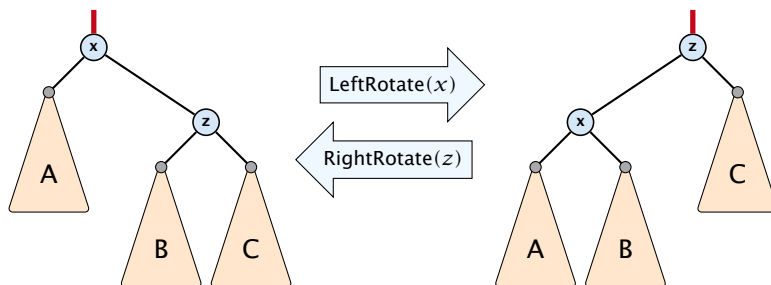
The **null**-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data.

## 7.2 Red Black Trees

We need to adapt the insert and delete operations so that the red black properties are maintained.

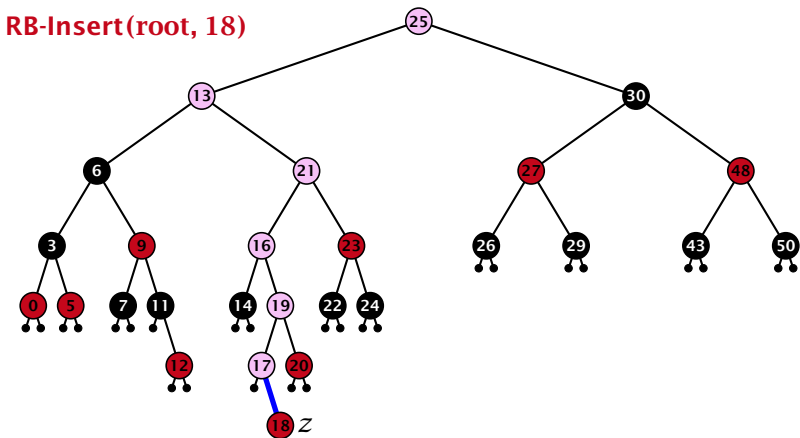
## Rotations

The properties will be maintained through rotations:



## Red Black Trees: Insert

RB-Insert(root, 18)



Insert:

- ▶ first make a normal insert into a binary search tree
- ▶ then fix red-black properties

## Red Black Trees: Insert

### Invariant of the fix-up algorithm:

- ▶  $z$  is a red node
- ▶ the black-height property is fulfilled at every node
- ▶ the only violation of red-black properties occurs at  $z$  and  $\text{parent}[z]$ 
  - ▶ either both of them are red (most important case)
  - ▶ or the parent does not exist (violation since root must be black)

If  $z$  has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

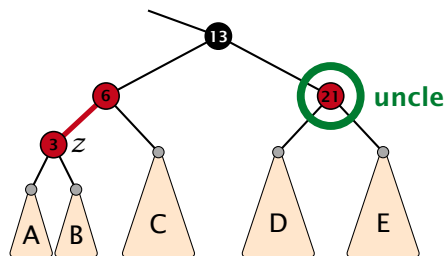
## Red Black Trees: Insert

### Algorithm 10 InsertFix( $z$ )

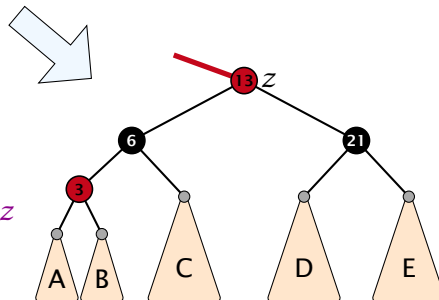
```

1: while parent[z] ≠ null and col[parent[z]] = red do
2:   if parent[z] = left[gp[z]] then z in left subtree of grandparent
3:     uncle ← right[grandparent[z]]
4:     if col[uncle] = red then Case 1: uncle red
5:       col[p[z]] ← black; col[u] ← black;
6:       col[gp[z]] ← red; z ← grandparent[z];
7:     else Case 2: uncle black
8:       if z = right[parent[z]] then 2a: z right child
9:         z ← p[z]; LeftRotate(z);
10:      col[p[z]] ← black; col[gp[z]] ← red; 2b: z left child
11:      RightRotate(gp[z]);
12:     else same as then-clause but right and left exchanged
13:   col[root[T]] ← black;
  
```

### Case 1: Red Uncle

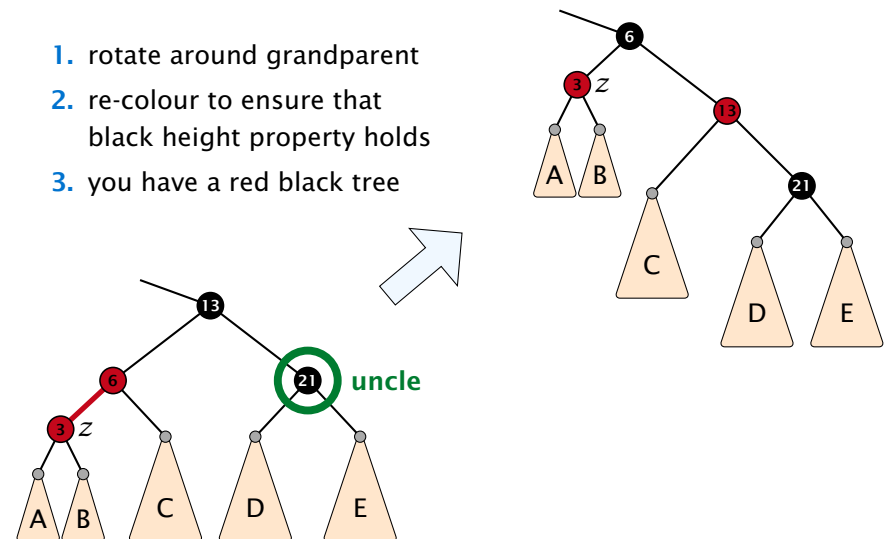


1. recolour
2. move  $z$  to grand-parent
3. invariant is fulfilled for new  $z$
4. you made progress



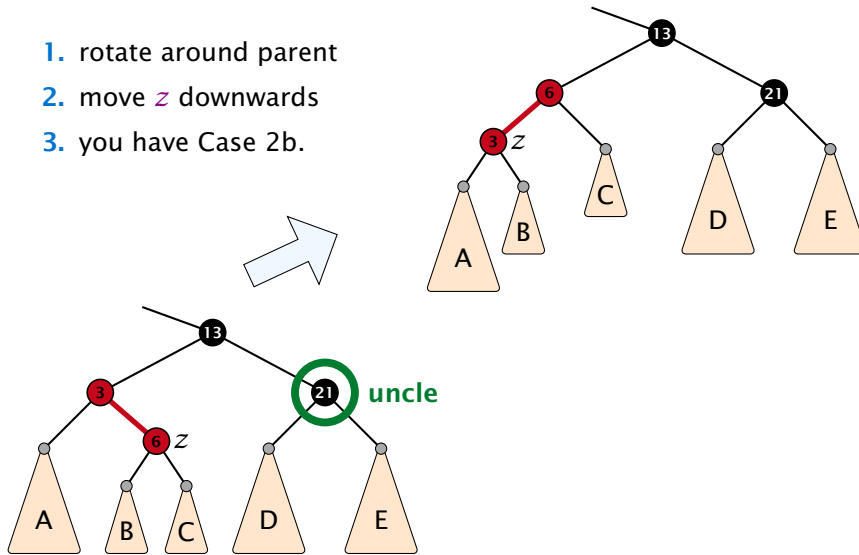
### Case 2b: Black uncle and $z$ is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree



## Case 2a: Black uncle and z is right child

1. rotate around parent
2. move  $z$  downwards
3. you have Case 2b.



## Red Black Trees: Insert

### Running time:

- ▶ Only Case 1 may repeat; but only  $h/2$  many steps, where  $h$  is the height of the tree.
- ▶ Case 2a  $\rightarrow$  Case 2b  $\rightarrow$  red-black tree
- ▶ Case 2b  $\rightarrow$  red-black tree

Performing Case 1 at most  $\mathcal{O}(\log n)$  times and every other case at most once, we get a red-black tree. Hence  $\mathcal{O}(\log n)$  re-colorings and at most 2 rotations.

## Red Black Trees: Delete

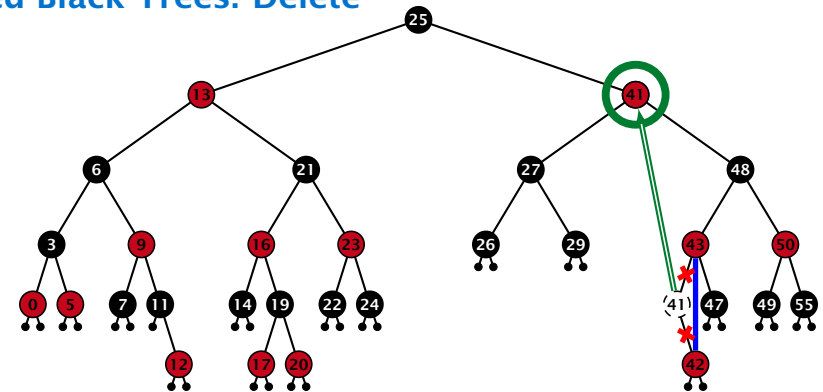
First do a standard delete.

If the spliced out node  $x$  was red everything is fine.

If it was black there may be the following problems.

- ▶ Parent and child of  $x$  were red; two adjacent red vertices.
- ▶ If you delete the root, the root may now be red.
- ▶ Every path from an ancestor of  $x$  to a descendant leaf of  $x$  changes the number of black nodes. Black height property might be violated.

## Red Black Trees: Delete



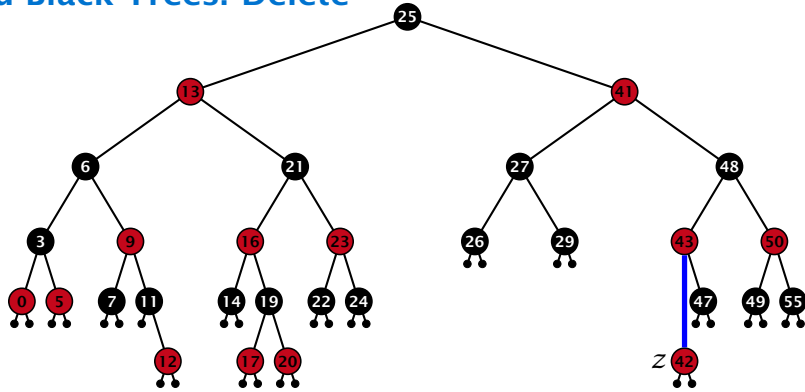
### Case 3:

Element has two children

- ▶ do normal delete
- ▶ when replacing content by content of successor, don't change color of node



## Red Black Trees: Delete



### Delete:

- ▶ deleting black node messes up black-height property
- ▶ if  $z$  is red, we can simply color it black and everything is fine
- ▶ the problem is if  $z$  is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

## Red Black Trees: Delete

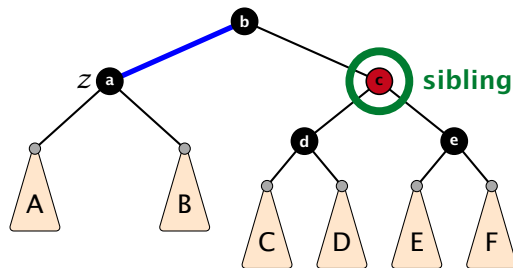
### Invariant of the fix-up algorithm

- ▶ the node  $z$  is black
- ▶ if we “assign” a fake black unit to the edge from  $z$  to its parent then the black-height property is fulfilled

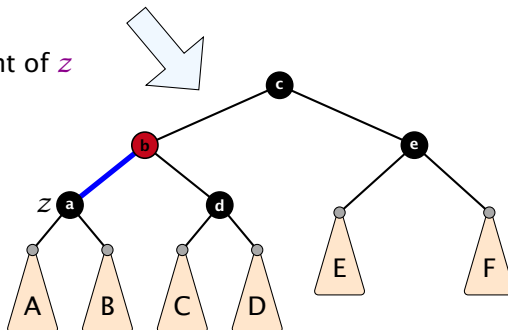
**Goal:** make rotations in such a way that you at some point can remove the fake black unit from the edge.



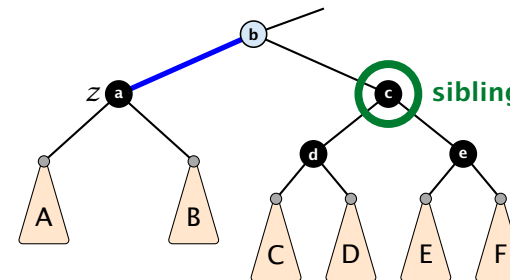
## Case 1: Sibling of $z$ is red



1. left-rotate around parent of  $z$
2. recolor nodes  $b$  and  $c$
3. the new sibling is black (and parent of  $z$  is red)
4. Case 2 (special), or Case 3, or Case 4

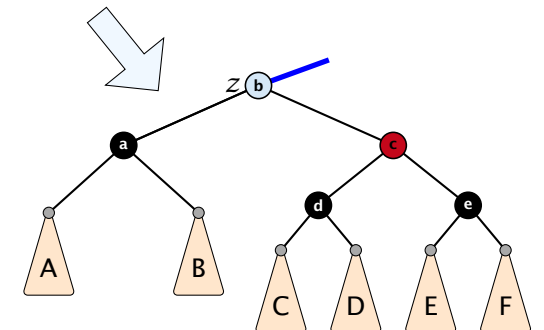


## Case 2: Sibling is black with two black children



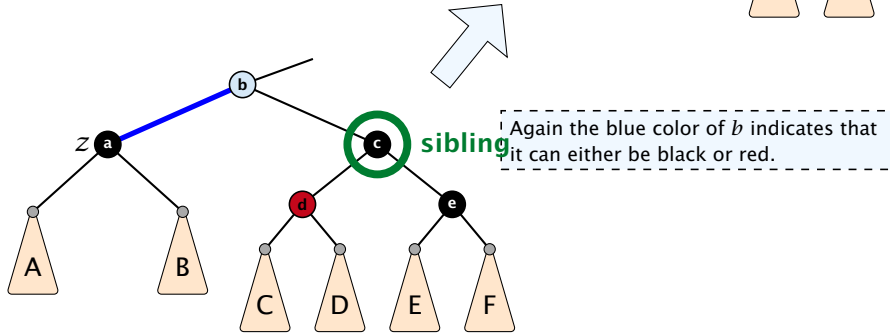
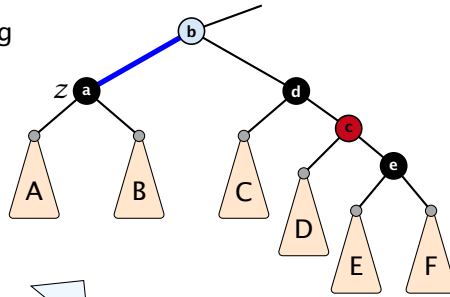
Here  $b$  is either black or red. If it is red we are in a special case that directly leads to a red-black tree.

1. re-color node  $c$
2. move fake black unit upwards
3. move  $z$  upwards
4. we made progress
5. if  $b$  is red we color it black and are done



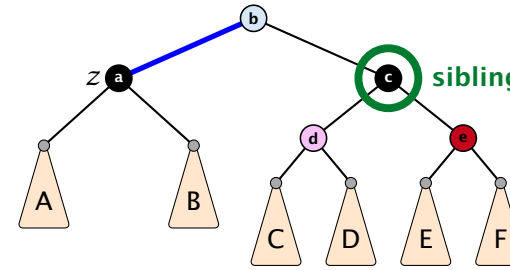
### Case 3: Sibling black with one black child to the right

1. do a right-rotation at sibling
2. recolor  $c$  and  $d$
3. new sibling is black with red right child (Case 4)

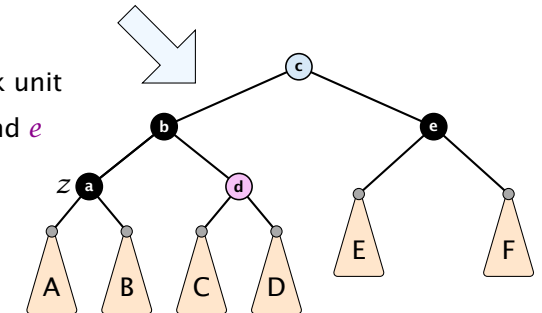


### Case 4: Sibling is black with red right child

- Here  $b$  and  $d$  are either red or black but have possibly different colors.
- We recolor  $c$  by giving it the color of  $b$ .



1. left-rotate around  $b$
2. remove the fake black unit
3. recolor nodes  $b$ ,  $c$ , and  $e$
4. you have a valid red black tree



#### Running time:

- ▶ only Case 2 can repeat; but only  $h$  many steps, where  $h$  is the height of the tree
- ▶ Case 1 → Case 2 (special) → red black tree
- ▶ Case 1 → Case 3 → Case 4 → red black tree
- ▶ Case 1 → Case 4 → red black tree
- ▶ Case 3 → Case 4 → red black tree
- ▶ Case 4 → red black tree

Performing Case 2 at most  $\mathcal{O}(\log n)$  times and every other step at most once, we get a red black tree. Hence,  $\mathcal{O}(\log n)$  re-colorings and at most 3 rotations.

### Red-Black Trees

#### Bibliography

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein: *Introduction to Algorithms (3rd ed.)*, MIT Press and McGraw-Hill, 2009

Red black trees are covered in detail in Chapter 13 of [CLRS90].

## Splay Trees

### Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

### Splay Trees:

- + after access, an element is moved to the root;  $\text{splay}(x)$  repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

## Splay Trees

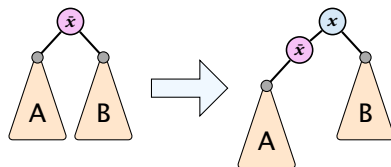
### find( $x$ )

- ▶ search for  $x$  according to a search tree
- ▶ let  $\bar{x}$  be last element on search-path
- ▶  $\text{splay}(\bar{x})$

## Splay Trees

### insert( $x$ )

- ▶ search for  $x$ ;  $\bar{x}$  is last visited element during search (successor or predecessor of  $x$ )
- ▶  $\text{splay}(\bar{x})$  moves  $\bar{x}$  to the root
- ▶ insert  $x$  as new root

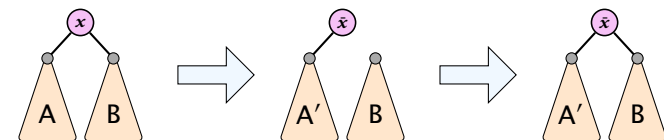


The illustration shows the case when  $\bar{x}$  is the predecessor of  $x$ .

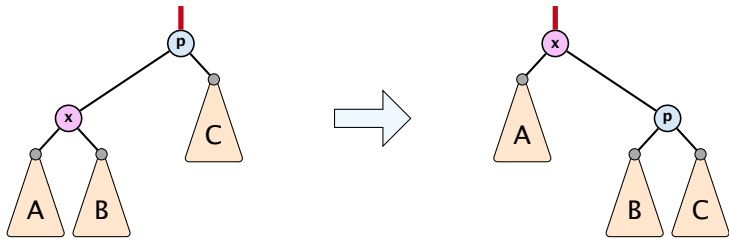
## Splay Trees

### delete( $x$ )

- ▶ search for  $x$ ;  $\text{splay}(x)$ ; remove  $x$
- ▶ search largest element  $\bar{x}$  in  $A$
- ▶  $\text{splay}(\bar{x})$  (on subtree  $A$ )
- ▶ connect root of  $B$  as right child of  $\bar{x}$



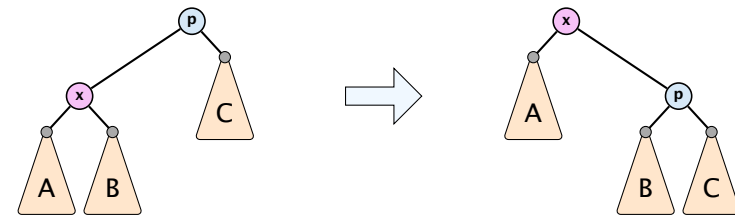
## Move to Root



### How to bring element to root?

- ▶ one (bad) option:  $\text{moveToRoot}(x)$
- ▶ iteratively do rotation around parent of  $x$  until  $x$  is root
- ▶ if  $x$  is left child do right rotation otw. left rotation

## Splay: Zig Case

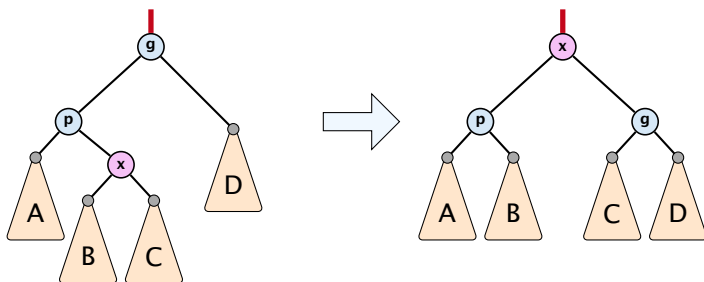


### better option $\text{splay}(x)$ :

- ▶ zig case: if  $x$  is child of root do left rotation or right rotation around parent

Note that  $\text{moveToRoot}(x)$  does the same.

## Splay: Zigzag Case

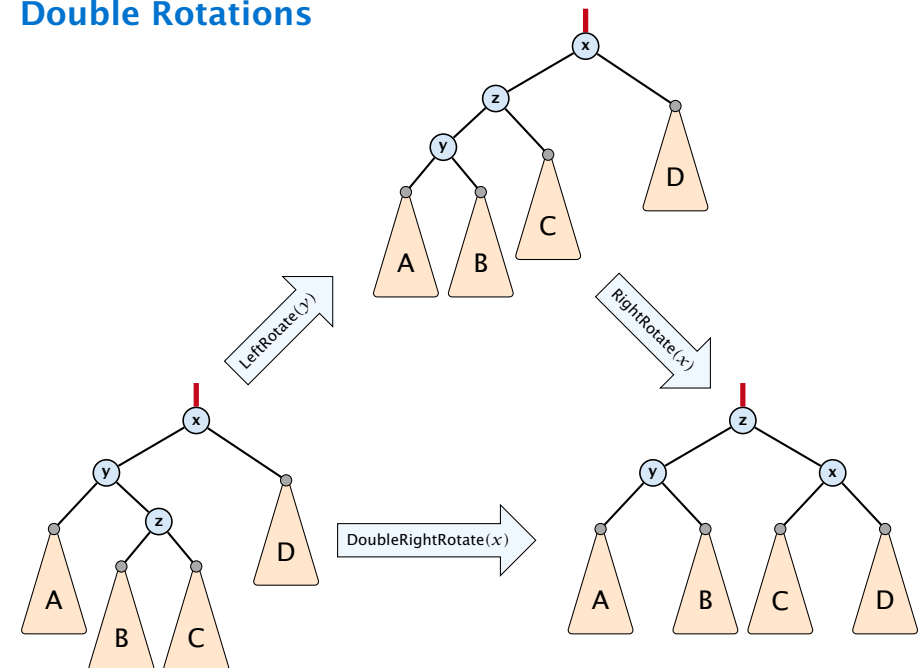


### better option $\text{splay}(x)$ :

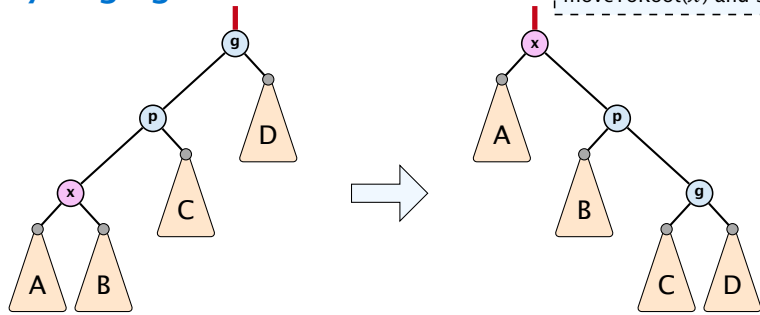
- ▶ zigzag case: if  $x$  is right child and parent of  $x$  is left child (or  $x$  left child parent of  $x$  right child)
- ▶ do double right rotation around grand-parent (resp. double left rotation)

Note that  $\text{moveToRoot}(x)$  does the same.

## Double Rotations



## Splay: Zigzig Case

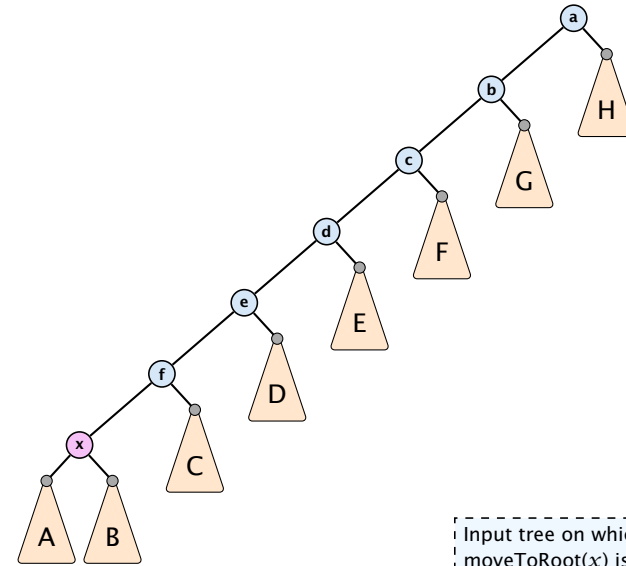


This case is different between  $\text{moveToRoot}(x)$  and  $\text{splay}(x)$ .

### better option $\text{splay}(x)$ :

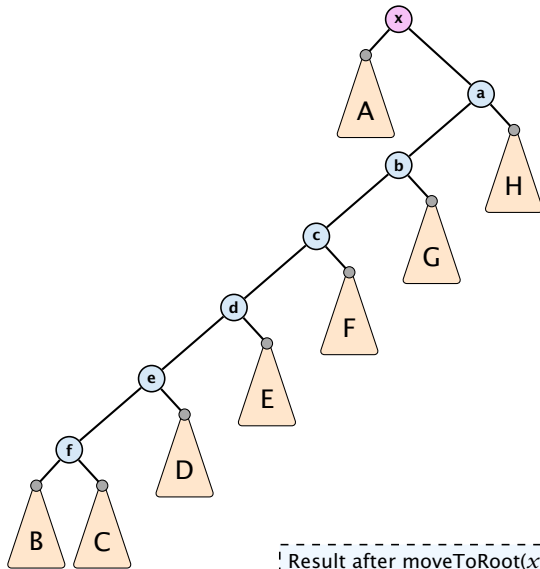
- ▶ zigzig case: if  $x$  is left child and parent of  $x$  is left child (or  $x$  right child, parent of  $x$  right child)
- ▶ do right rotation around grand-parent followed by right rotation around parent (resp. left rotations)

## Splay vs. Move to Root



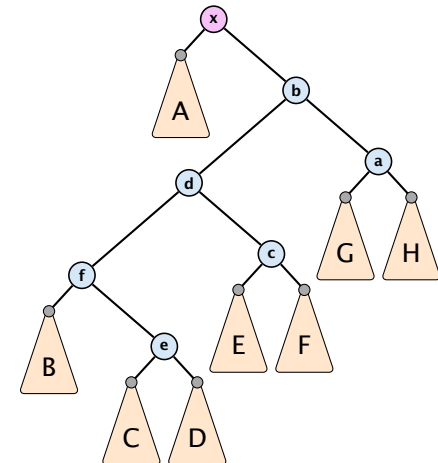
Input tree on which  $\text{splay}(x)$  and  $\text{moveToRoot}(x)$  is executed.

## Splay vs. Move to Root



Result after  $\text{moveToRoot}(x)$ .

## Splay vs. Move to Root



Result after  $\text{splay}(x)$ .

## Static Optimality

Suppose we have a sequence of  $m$  find-operations.  $\text{find}(x)$  appears  $h_x$  times in this sequence.

The cost of a **static** search tree  $T$  is:

$$\text{cost}(T) = m + \sum_x h_x \text{depth}_T(x)$$

The total cost for processing the sequence on a splay-tree is  $\mathcal{O}(\text{cost}(T_{\min}))$ , where  $T_{\min}$  is an **optimal static search tree**.

$\text{depth}_T(x)$  is the number of edges on a path from the root of  $T$  to  $x$ .  
Theorem given without proof.

## Dynamic Optimality

Let  $S$  be a sequence with  $m$  find-operations.

Let  $A$  be a data-structure based on a search tree:

- ▶ the cost for accessing element  $x$  is  $1 + \text{depth}(x)$ ;
- ▶ after accessing  $x$  the tree may be re-arranged through rotations;

### Conjecture:

A splay tree that only contains elements from  $S$  has cost  $\mathcal{O}(\text{cost}(A, S))$ , for processing  $S$ .

### Lemma 15

*Splay Trees have an **amortized** running time of  $\mathcal{O}(\log n)$  for all operations.*

## Amortized Analysis

### Definition 16

A data structure with operations  $\text{op}_1(), \dots, \text{op}_k()$  has amortized running times  $t_1, \dots, t_k$  for these operations if the following holds.

Suppose you are given a sequence of operations (**starting with an empty data-structure**) that operate on at most  $n$  elements, and let  $k_i$  denote the number of occurrences of  $\text{op}_i()$  within this sequence. Then the actual running time must be at most  $\sum_i k_i \cdot t_i(n)$ .

## Potential Method

Introduce a potential for the data structure.

- ▶  $\Phi(D_i)$  is the potential after the  $i$ -th operation.
- ▶ Amortized cost of the  $i$ -th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) .$$

- ▶ Show that  $\Phi(D_i) \geq \Phi(D_0)$ .

Then

$$\sum_{i=1}^k c_i \leq \sum_{i=1}^k c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^k \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.



## Example: Stack

Stack

- ▶ **S.push()**
- ▶ **S.pop()**
- ▶ **S.multipop(k)**: removes  $k$  items from the stack. If the stack currently contains less than  $k$  items it empties the stack.
- ▶ The user has to ensure that pop and multipop do not generate an underflow.

Actual cost:

- ▶ **S.push()**: cost 1.
- ▶ **S.pop()**: cost 1.
- ▶ **S.multipop(k)**: cost  $\min\{\text{size}, k\} = k$ .



## Example: Stack

Use potential function  $\Phi(S) = \text{number of elements on the stack}$ .

Amortized cost:

- ▶ **S.push()**: cost

$$\hat{C}_{\text{push}} = C_{\text{push}} + \Delta\Phi = 1 + 1 \leq 2 .$$

- ▶ **S.pop()**: cost

$$\hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta\Phi = 1 - 1 \leq 0 .$$

- ▶ **S.multipop(k)**: cost

$$\hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta\Phi = \min\{\text{size}, k\} - \min\{\text{size}, k\} \leq 0 .$$

Note that the analysis becomes wrong if pop() or multipop() are called on an empty stack.



## Example: Binary Counter

Incrementing a binary counter:

Consider a computational model where each bit-operation costs one time-unit.

Incrementing an  $n$ -bit binary counter may require to examine  $n$ -bits, and maybe change them.

Actual cost:

- ▶ Changing bit from 0 to 1: cost 1.
- ▶ Changing bit from 1 to 0: cost 1.
- ▶ Increment: cost is  $k + 1$ , where  $k$  is the number of consecutive ones in the least significant bit-positions (e.g., 001101 has  $k = 1$ ).



## Example: Binary Counter

Choose potential function  $\Phi(x) = k$ , where  $k$  denotes the number of ones in the binary representation of  $x$ .

**Amortized cost:**

- ▶ Changing bit from 0 to 1:

$$\hat{C}_{0 \rightarrow 1} = C_{0 \rightarrow 1} + \Delta\Phi = 1 + 1 \leq 2.$$

- ▶ Changing bit from 1 to 0:

$$\hat{C}_{1 \rightarrow 0} = C_{1 \rightarrow 0} + \Delta\Phi = 1 - 1 \leq 0.$$

- ▶ **Increment:** Let  $k$  denotes the number of consecutive ones in the least significant bit-positions. An increment involves  $k$   $(1 \rightarrow 0)$ -operations, and one  $(0 \rightarrow 1)$ -operation.

Hence, the amortized cost is  $k\hat{C}_{1 \rightarrow 0} + \hat{C}_{0 \rightarrow 1} \leq 2$ .

## Splay Trees

**potential function for splay trees:**

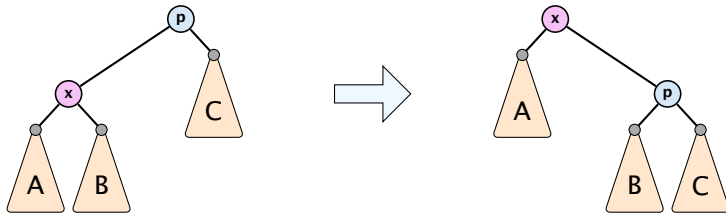
- ▶ size  $s(x) = |T_x|$
- ▶ rank  $r(x) = \log_2(s(x))$
- ▶  $\Phi(T) = \sum_{v \in T} r(v)$

amortized cost = real cost + potential change

The cost is essentially the cost of the splay-operation, which is 1 plus the number of rotations.



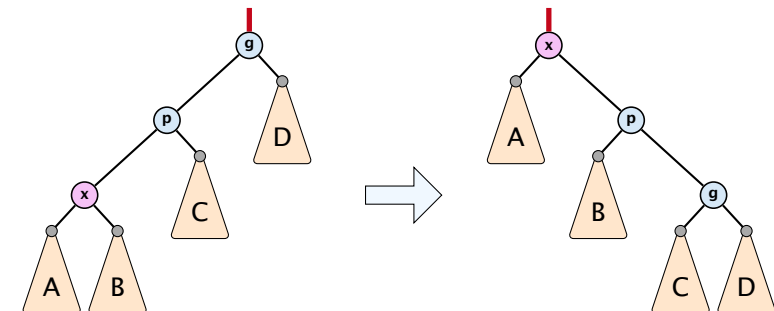
## Splay: Zig Case



$$\begin{aligned} \Delta\Phi &= r'(x) + r'(p) - r(x) - r(p) \\ &= r'(p) - r(x) \\ &\leq r'(x) - r(x) \end{aligned}$$

$$\text{cost}_{\text{zig}} \leq 1 + 3(r'(x) - r(x))$$

## Splay: Zigzig Case



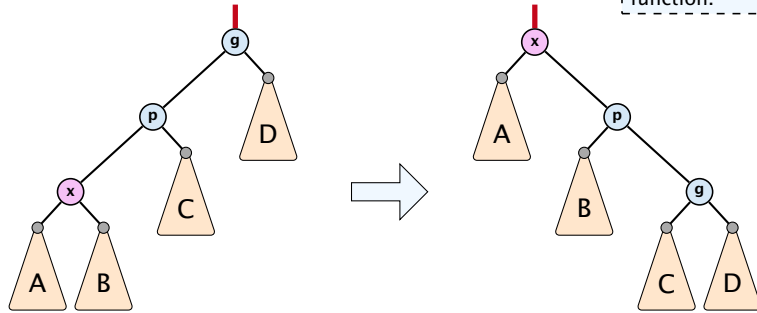
Last inequality follows from next slide.

$$\begin{aligned} \Delta\Phi &= r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g) \\ &= r'(p) + r'(g) - r(x) - r(p) \\ &\leq r'(x) + r'(g) - r(x) - r(x) \\ &= r'(x) + r'(g) + r(x) - 3r'(x) + 3r'(x) - r(x) - 2r(x) \\ &= -2r'(x) + r'(g) + r(x) + 3(r'(x) - r(x)) \\ &\leq -2 + 3(r'(x) - r(x)) \Rightarrow \text{cost}_{\text{zigzig}} \leq 3(r'(x) - r(x)) \end{aligned}$$





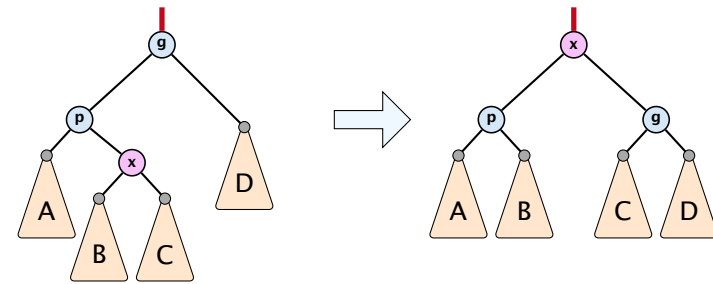
## Splay: Zigzig Case



The last inequality holds because log is a concave function.

$$\begin{aligned} & \frac{1}{2}(r(x) + r'(g) - 2r'(x)) \\ &= \frac{1}{2}(\log(s(x)) + \log(s'(g)) - 2\log(s'(x))) \\ &= \frac{1}{2}\log\left(\frac{s(x)}{s'(x)}\right) + \frac{1}{2}\log\left(\frac{s'(g)}{s'(x)}\right) \\ &\leq \log\left(\frac{1}{2}\frac{s(x)}{s'(x)} + \frac{1}{2}\frac{s'(g)}{s'(x)}\right) \leq \log\left(\frac{1}{2}\right) = -1 \end{aligned}$$

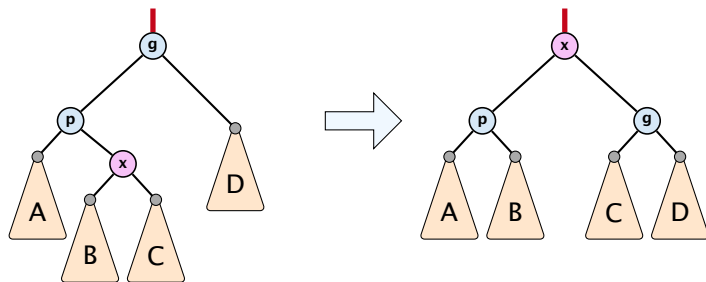
## Splay: Zigzag Case



$$\begin{aligned} \Delta\Phi &= r'(x) + r'(p) + r'(g) - r(x) - r(p) - r(g) \\ &= r'(p) + r'(g) - r(x) - r(p) \\ &\leq r'(p) + r'(g) - 2r'(x) + 2r'(x) - 2r(x) \\ &\leq -2 + 2(r'(x) - r(x)) \Rightarrow \text{cost}_{\text{zigzag}} \leq 3(r'(x) - r(x)) \end{aligned}$$



## Splay: Zigzag Case



$$\begin{aligned} & \frac{1}{2}(r'(p) + r'(g) - 2r'(x)) \\ &= \frac{1}{2}(\log(s'(p)) + \log(s'(g)) - 2\log(s'(x))) \\ &\leq \log\left(\frac{1}{2}\frac{s'(p)}{s'(x)} + \frac{1}{2}\frac{s'(g)}{s'(x)}\right) \leq \log\left(\frac{1}{2}\right) = -1 \end{aligned}$$

Amortized cost of the whole splay operation:

$$\begin{aligned} & \leq 1 + 1 + \sum_{\text{steps } t} 3(r_t(x) - r_{t-1}(x)) \\ &= 2 + 3(r(\text{root}) - r_0(x)) \\ &\leq \mathcal{O}(\log n) \end{aligned}$$

The first one is added due to the fact that so far for each step of a splay-operation we have only counted the number of rotations, but the cost is 1+#rotations.

The second one comes from the zig-operation. Note that we have at most one zig-operation during a splay.



# Splay Trees

## Bibliography

~~~~~

# 7.4 Augmenting Data Structures

Suppose you want to develop a data structure with:

- ▶ **Insert( $x$ )**: insert element  $x$ .
- ▶ **Search( $k$ )**: search for element with key  $k$ .
- ▶ **Delete( $x$ )**: delete element referenced by pointer  $x$ .
- ▶ **find-by-rank( $\ell$ )**: return the  $\ell$ -th element; return “error” if the data-structure contains less than  $\ell$  elements.

**Augment an existing data-structure instead of developing a new one.**

# 7.4 Augmenting Data Structures

## How to augment a data-structure

1. choose an underlying data-structure
  2. determine additional information to be stored in the underlying structure
  3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
  4. develop the new operations
- Of course, the above steps heavily depend on each other. For example it makes no sense to choose additional information to be stored (Step 2), and later realize that either the information cannot be maintained efficiently (Step 3) or is not sufficient to support the new operations (Step 4).
  - However, the above outline is a good way to describe/document a new data-structure.

# 7.4 Augmenting Data Structures

**Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .**

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node  $v$  the size of the sub-tree rooted at  $v$ .
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

## 7.4 Augmenting Data Structures

**Goal:** Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .

4. How does find-by-rank work?

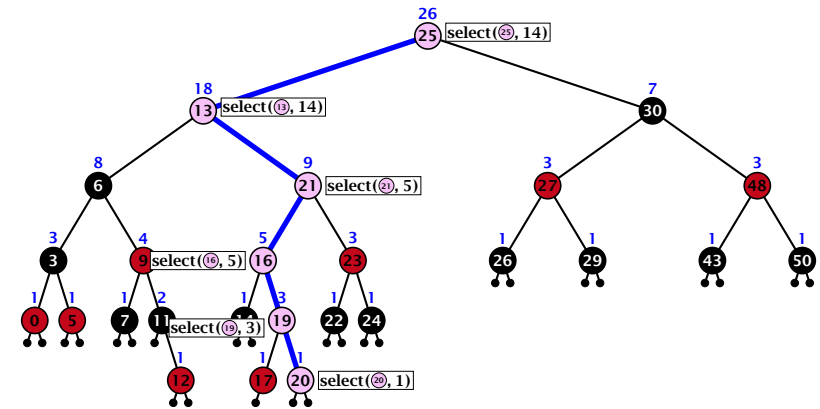
Find-by-rank( $k$ ) := Select(root,  $k$ ) with

**Algorithm 11** Select( $x, i$ )

```

1: if  $x = \text{null}$  then return error
2: if left[ $x$ ]  $\neq$  null then  $r \leftarrow$  left[ $x$ ].size + 1 else  $r \leftarrow 1$ 
3: if  $i = r$  then return  $x$ 
4: if  $i < r$  then
5:     return Select(left[ $x$ ],  $i$ )
6: else
7:     return Select(right[ $x$ ],  $i - r$ )
    
```

## Select( $x, i$ )



**Find-by-rank:**

- ▶ decide whether you have to proceed into the left or right sub-tree
- ▶ adjust the rank that you are searching for if you go right

## 7.4 Augmenting Data Structures

**Goal:** Design a data-structure that supports insert, delete, search, and find-by-rank in time  $\mathcal{O}(\log n)$ .

3. How do we maintain information?

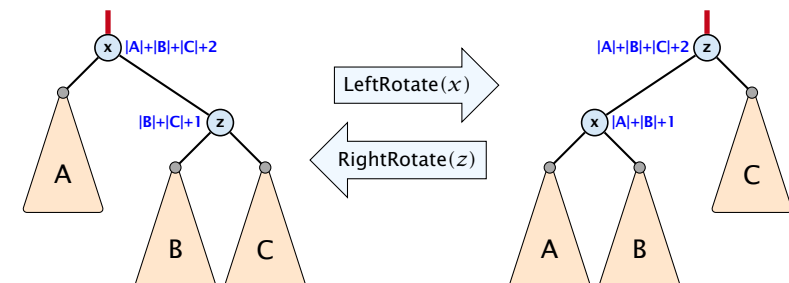
**Search( $k$ ):** Nothing to do.

**Insert( $x$ ):** When going down the search path increase the size field for each visited node. **Maintain the size field during rotations.**

**Delete( $x$ ):** Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. **Maintain the size field during rotations.**

## Rotations

The only operation during the fix-up procedure that alters the tree and requires an update of the size-field:



The nodes  $x$  and  $z$  are the only nodes changing their size-fields.

The new size-fields can be computed **locally** from the size-fields of the children.

## Augmenting Data Structures

### Bibliography

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:  
*Introduction to Algorithms (3rd ed.)*,  
MIT Press and McGraw-Hill, 2009

See Chapter 14 of [CLRS90].



## 7.5 ( $a, b$ )-trees

### Definition 17

For  $b \geq 2a - 1$  an ( $a, b$ )-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex  $v$  has at least  $a$  and at most  $b$  children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value  $\infty$



## 7.5 ( $a, b$ )-trees

Each internal node  $v$  with  $d(v)$  children stores  $d - 1$  keys  $k_1, \dots, k_{d-1}$ . The  $i$ -th subtree of  $v$  fulfills

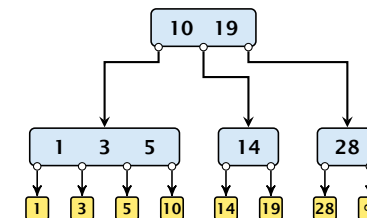
$$k_{i-1} < \text{key in } i\text{-th sub-tree} \leq k_i,$$

where we use  $k_0 = -\infty$  and  $k_d = \infty$ .



## 7.5 ( $a, b$ )-trees

### Example 18



## 7.5 (a, b)-trees

### Variants

- ▶ The dummy leaf element may not exist; it only makes implementation more convenient.
- ▶ Variants in which  $b = 2a$  are commonly referred to as  $B$ -trees.
- ▶ A  $B$ -tree usually refers to the variant in which keys and data are stored at internal nodes.
- ▶ A  $B^+$  tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- ▶ A  $B^*$  tree requires that a node is at least  $2/3$ -full as opposed to  $1/2$ -full (the requirement of a  $B$ -tree).

### Lemma 19

Let  $T$  be an  $(a, b)$ -tree for  $n > 0$  elements (i.e.,  $n + 1$  leaf nodes) and height  $h$  (number of edges from root to a leaf vertex). Then

1.  $2a^{h-1} \leq n + 1 \leq b^h$
2.  $\log_b(n + 1) \leq h \leq 1 + \log_a(\frac{n+1}{2})$

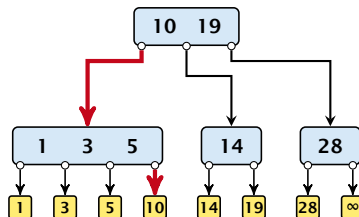
### Proof.

- ▶ If  $n > 0$  the root has degree at least 2 and all other nodes have degree at least  $a$ . This gives that the number of leaf nodes is at least  $2a^{h-1}$ .
- ▶ Analogously, the degree of any node is at most  $b$  and, hence, the number of leaf nodes at most  $b^h$ .

□

## Search

### Search(8)

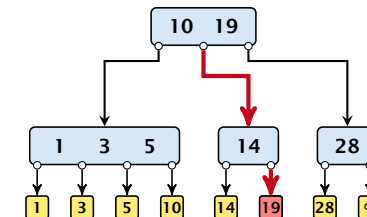


The search is straightforward. It is only important that you need to go all the way to the leaf.

Time:  $\mathcal{O}(b \cdot h) = \mathcal{O}(b \cdot \log n)$ , if the individual nodes are organized as linear lists.

## Search

### Search(19)



The search is straightforward. It is only important that you need to go all the way to the leaf.

Time:  $\mathcal{O}(b \cdot h) = \mathcal{O}(b \cdot \log n)$ , if the individual nodes are organized as linear lists.

## Insert

Insert element  $x$ :

- ▶ Follow the path as if searching for  $\text{key}[x]$ .
- ▶ If this search ends in leaf  $\ell$ , insert  $x$  **before** this leaf.
- ▶ For this add  $\text{key}[x]$  to the key-list of the last internal node  $v$  on the path.
- ▶ If after the insert  $v$  contains  $b$  nodes, do  $\text{Rebalance}(v)$ .

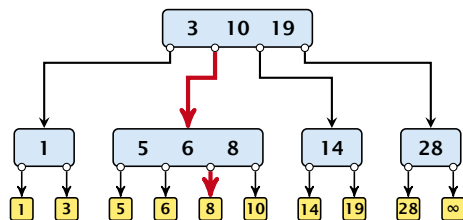
## Insert

$\text{Rebalance}(v)$ :

- ▶ Let  $k_i, i = 1, \dots, b$  denote the keys stored in  $v$ .
- ▶ Let  $j := \lfloor \frac{b+1}{2} \rfloor$  be the middle element.
- ▶ Create two nodes  $v_1$ , and  $v_2$ .  $v_1$  gets all keys  $k_1, \dots, k_{j-1}$  and  $v_2$  gets keys  $k_{j+1}, \dots, k_b$ .
- ▶ Both nodes get at least  $\lfloor \frac{b-1}{2} \rfloor$  keys, and have therefore degree at least  $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$  since  $b \geq 2a - 1$ .
- ▶ They get at most  $\lceil \frac{b-1}{2} \rceil$  keys, and have therefore degree at most  $\lceil \frac{b-1}{2} \rceil + 1 \leq b$  (since  $b \geq 2$ ).
- ▶ The key  $k_j$  is promoted to the parent of  $v$ . The current pointer to  $v$  is altered to point to  $v_1$ , and a new pointer (to the right of  $k_j$ ) in the parent is added to point to  $v_2$ .
- ▶ Then, re-balance the parent.

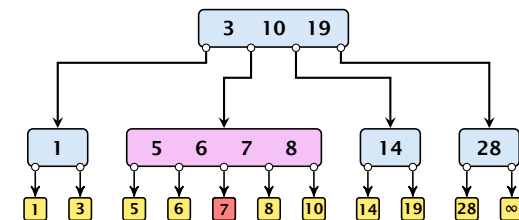
## Insert

Insert(7)



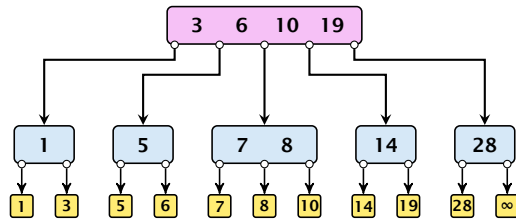
## Insert

Insert(7)



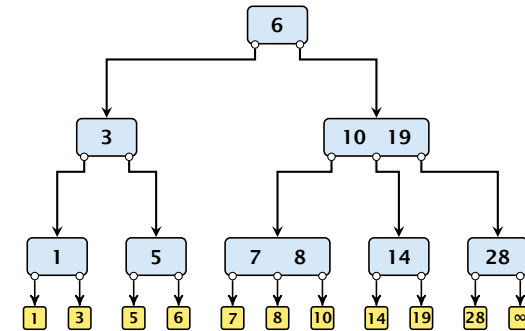
## Insert

### Insert(7)



## Insert

### Insert(7)



## Delete

Delete element  $x$  (pointer to leaf vertex):

- ▶ Let  $v$  denote the parent of  $x$ . If  $\text{key}[x]$  is contained in  $v$ , remove the key from  $v$ , and delete the leaf vertex.
- ▶ Otherwise delete the key of the predecessor of  $x$  from  $v$ ; delete the leaf vertex; and replace the occurrence of  $\text{key}[x]$  in internal nodes by the predecessor key. (Note that it appears in exactly one internal vertex).
- ▶ If now the number of keys in  $v$  is below  $a - 1$  perform  $\text{Rebalance}'(v)$ .

## Delete

$\text{Rebalance}'(v)$ :

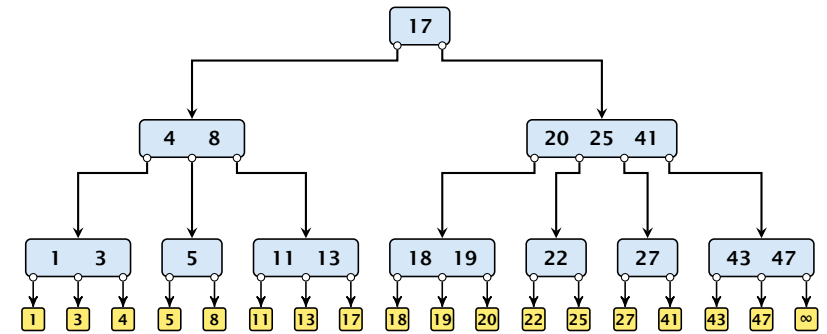
- ▶ If there is a neighbour of  $v$  that has at least  $a$  keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- ▶ If not: merge  $v$  with one of its neighbours.
- ▶ The merged node contains at most  $(a - 2) + (a - 1) + 1$  keys, and has therefore at most  $2a - 1 \leq b$  successors.
- ▶ Then rebalance the parent.
- ▶ During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

## Delete

Animation for deleting in an  $(a, b)$ -tree is only available in the lecture version of the slides.

## $(2, 4)$ -trees and red black trees

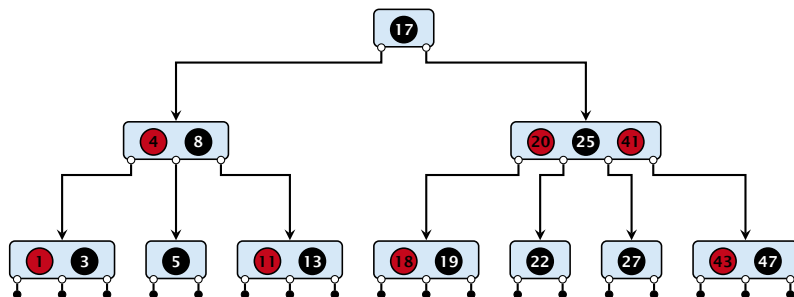
There is a close relation between red-black trees and  $(2, 4)$ -trees:



First make it into an internal search tree by moving the satellite-data from the leaves to internal nodes. Add dummy leaves.

## $(2, 4)$ -trees and red black trees

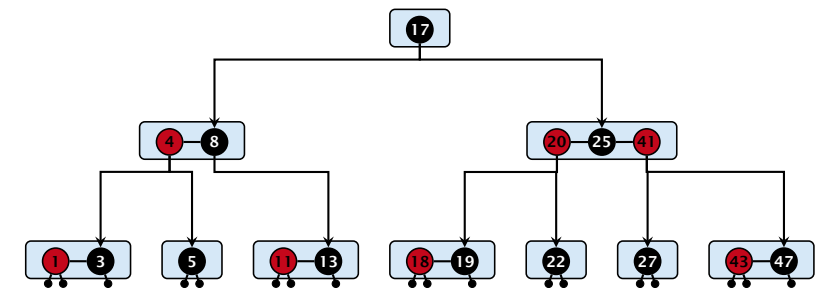
There is a close relation between red-black trees and  $(2, 4)$ -trees:



Then, color one key in each internal node  $v$  black. If  $v$  contains 3 keys you need to select the middle key otherwise choose a black key arbitrarily. The other keys are colored red.

## $(2, 4)$ -trees and red black trees

There is a close relation between red-black trees and  $(2, 4)$ -trees:

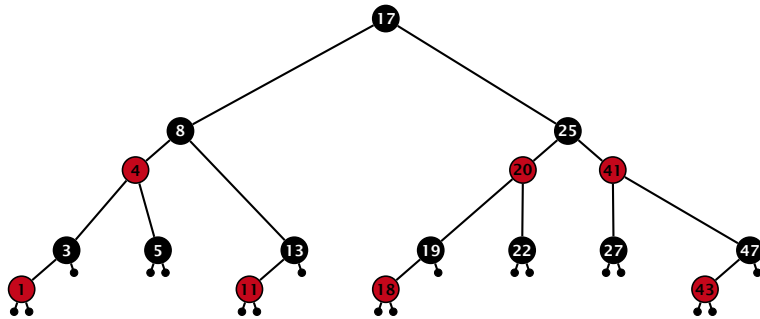


Re-attach the pointers to individual keys. A pointer that is between two keys is attached as a child of the red key. The incoming pointer, points to the black key.



## (2, 4)-trees and red black trees

There is a close relation between red-black trees and (2, 4)-trees:



Note that this correspondence is not unique. In particular, there are different red-black trees that correspond to the same (2, 4)-tree.

## Augmenting Data Structures

### Bibliography

[MS08] Kurt Mehlhorn, Peter Sanders:  
*Algorithms and Data Structures — The Basic Toolbox*,  
Springer, 2008

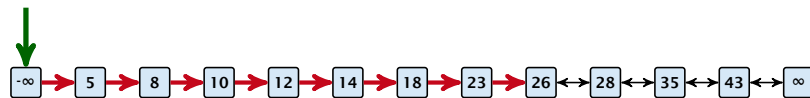
[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:  
*Introduction to algorithms (3rd ed.)*,  
MIT Press and McGraw-Hill, 2009

A description of B-trees (a specific variant of (a, b)-trees) can be found in Chapter 18 of [CLRS90]. Chapter 7.2 of [MS08] discusses (a, b)-trees as discussed in the lecture.

## 7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

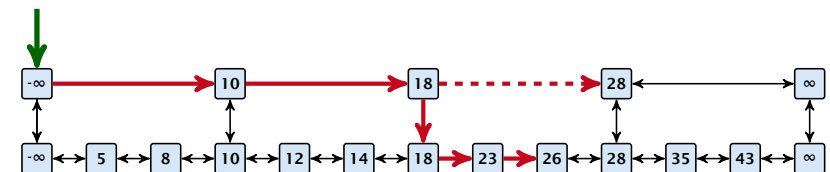
- ▶ time for search  $\Theta(n)$
- ▶ time for insert  $\Theta(n)$  (dominated by searching the item)
- ▶ time for delete  $\Theta(1)$  if we are given a handle to the object, otw.  $\Theta(n)$



## 7.6 Skip Lists

How can we improve the search-operation?

Add an express lane:



Let  $|L_1|$  denote the number of elements in the “express lane”, and  $|L_0| = n$  the number of all elements (ignoring dummy elements).

Worst case search time:  $|L_1| + \frac{|L_0|}{|L_1|}$  (ignoring additive constants)

Choose  $|L_1| = \sqrt{n}$ . Then search time  $\Theta(\sqrt{n})$ .

## 7.6 Skip Lists

Add more express lanes. Lane  $L_i$  contains roughly every  $\frac{L_{i-1}}{L_i}$ -th item from list  $L_{i-1}$ .

**Search(x) ( $k + 1$  lists  $L_0, \dots, L_k$ )**

- ▶ Find the largest item in list  $L_k$  that is smaller than  $x$ . At most  $|L_k| + 2$  steps.
- ▶ Find the largest item in list  $L_{k-1}$  that is smaller than  $x$ . At most  $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$  steps.
- ▶ Find the largest item in list  $L_{k-2}$  that is smaller than  $x$ . At most  $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$  steps.
- ▶ ...
- ▶ At most  $|L_k| + \sum_{i=1}^k \frac{L_{i-1}}{L_i} + 3(k + 1)$  steps.



## 7.6 Skip Lists

Choose ratios between list-lengths evenly, i.e.,  $\frac{L_{i-1}}{|L_i|} = r$ , and, hence,  $L_k \approx r^{-k}n$ .

Worst case running time is:  $\mathcal{O}(r^{-k}n + kr)$ .

Choose  $r = n^{\frac{1}{k+1}}$ . Then

$$\begin{aligned} r^{-k}n + kr &= \left(n^{\frac{1}{k+1}}\right)^{-k} n + kn^{\frac{1}{k+1}} \\ &= n^{1 - \frac{k}{k+1}} + kn^{\frac{1}{k+1}} \\ &= (k + 1)n^{\frac{1}{k+1}}. \end{aligned}$$

Choosing  $k = \Theta(\log n)$  gives a logarithmic running time.



## 7.6 Skip Lists

**How to do insert and delete?**

- ▶ If we want that in  $L_i$  we always skip over roughly the same number of elements in  $L_{i-1}$  an insert or delete may require a lot of re-organisation.

**Use randomization instead!**



## 7.6 Skip Lists

**Insert:**

- ▶ A search operation gives you the insert position for element  $x$  in every list.
- ▶ Flip a coin until it shows head, and record the number  $t \in \{1, 2, \dots\}$  of trials needed.
- ▶ Insert  $x$  into lists  $L_0, \dots, L_{t-1}$ .

**Delete:**

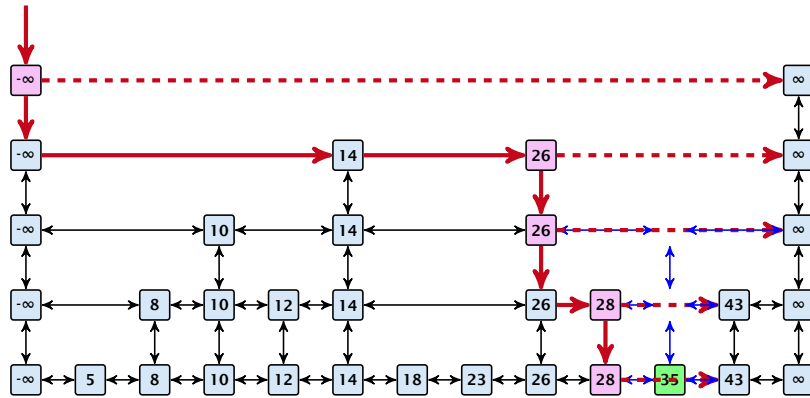
- ▶ You get all predecessors via backward pointers.
- ▶ Delete  $x$  in all lists it actually appears in.

**The time for both operations is dominated by the search time.**



## 7.6 Skip Lists

Insert (35):



## High Probability

### Definition 20 (High Probability)

We say a **randomized** algorithm has running time  $\mathcal{O}(\log n)$  with **high probability** if for any constant  $\alpha$  the running time is at most  $\mathcal{O}(\log n)$  with probability at least  $1 - \frac{1}{n^\alpha}$ .

Here the  $\mathcal{O}$ -notation hides a constant that may depend on  $\alpha$ .

## High Probability

Suppose there are **polynomially** many events  $E_1, E_2, \dots, E_\ell$ ,  $\ell = n^c$  each holding with high probability (e.g.  $E_i$  may be the event that the  $i$ -th search in a skip list takes time at most  $\mathcal{O}(\log n)$ ).

Then the probability that all  $E_i$  hold is at least

$$\begin{aligned} \Pr[E_1 \wedge \dots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \dots \vee \bar{E}_\ell] \\ &\geq 1 - n^c \cdot n^{-\alpha} \\ &= 1 - n^{c-\alpha}. \end{aligned}$$

This means  $\Pr[E_1 \wedge \dots \wedge E_\ell]$  holds with high probability.

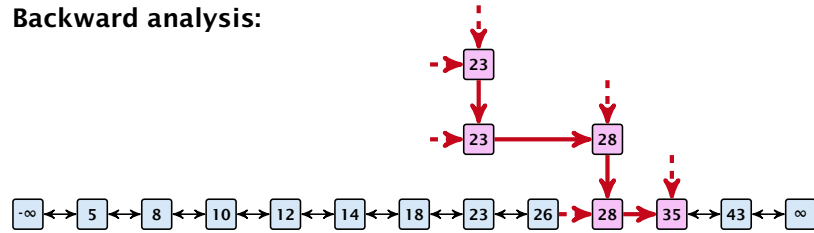
## 7.6 Skip Lists

### Lemma 21

A search (and, hence, also insert and delete) in a skip list with  $n$  elements takes time  $\mathcal{O}(\log n)$  with high probability (w. h. p.).

## 7.6 Skip Lists

Backward analysis:



At each point the path goes up with probability  $1/2$  and left with probability  $1/2$ .

We show that w.h.p:

- ▶ A “long” search path must also go very high.
- ▶ There are no elements in high lists.

From this it follows that w.h.p. there are no long paths.

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot \dots \cdot (n-k+1)}{k \cdot \dots \cdot 1} \geq \left(\frac{n}{k}\right)^k$$

$$\begin{aligned} \binom{n}{k} &= \frac{n \cdot \dots \cdot (n-k+1)}{k!} \leq \frac{n^k}{k!} = \frac{n^k \cdot k^k}{k^k \cdot k!} \\ &= \left(\frac{n}{k}\right)^k \cdot \frac{k^k}{k!} \leq \left(\frac{en}{k}\right)^k \end{aligned}$$

## 7.6 Skip Lists

Let  $E_{z,k}$  denote the event that a search path is of length  $z$  (number of edges) but does not visit a list above  $L_k$ .

In particular, this means that during the construction in the backward analysis we see at most  $k$  heads (i.e., coin flips that tell you to go up) in  $z$  trials.

## 7.6 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing  $k = \gamma \log n$  with  $\gamma \geq 1$  and  $z = (\beta + \alpha)\gamma \log n$

$$\leq \left(\frac{2ez}{k}\right)^k 2^{-\beta k} \cdot n^{-\gamma \alpha} \leq \left(\frac{2ez}{2\beta k}\right)^k \cdot n^{-\alpha}$$

$$\leq \left(\frac{2e(\beta + \alpha)}{2\beta}\right)^k n^{-\alpha}$$

now choosing  $\beta = 6\alpha$  gives

$$\leq \left(\frac{42\alpha}{64\alpha}\right)^k n^{-\alpha} \leq n^{-\alpha}$$

for  $\alpha \geq 1$ .

## 7.6 Skip Lists

So far we fixed  $k = \gamma \log n$ ,  $\gamma \geq 1$ , and  $z = 7\alpha\gamma \log n$ ,  $\alpha \geq 1$ .

This means that a search path of length  $\Omega(\log n)$  visits a list on a level  $\Omega(\log n)$ , w.h.p.

Let  $A_{k+1}$  denote the event that the list  $L_{k+1}$  is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} .$$

For the search to take at least  $z = 7\alpha\gamma \log n$  steps either the event  $E_{z,k}$  or the event  $A_{k+1}$  must hold.

Hence,

$$\begin{aligned} \Pr[\text{search requires } z \text{ steps}] &\leq \Pr[E_{z,k}] + \Pr[A_{k+1}] \\ &\leq n^{-\alpha} + n^{-(\gamma-1)} \end{aligned}$$

This means, the search requires at most  $z$  steps, w. h. p.

## Skip Lists

### Bibliography

[GT98] Michael T. Goodrich, Roberto Tamassia  
*Data Structures and Algorithms in JAVA*,  
John Wiley, 1998

Skip lists are covered in Chapter 7.5 of [GT98].



## 7.7 Hashing

### Dictionary:

- ▶ **S.insert(x)**: Insert an element  $x$ .
- ▶ **S.delete(x)**: Delete the element pointed to by  $x$ .
- ▶ **S.search(k)**: Return a pointer to an element  $e$  with  $\text{key}[e] = k$  in  $S$  if it exists; otherwise return **null**.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object  $x$  with key  $k$  is determined by successively comparing  $k$  to split-elements.

Hashing tries to **directly** compute the memory location from the given key. The goal is to have constant search time.

## 7.7 Hashing

### Definitions:

- ▶ Universe  $U$  of keys, e.g.,  $U \subseteq \mathbb{N}_0$ .  $U$  very large.
- ▶ Set  $S \subseteq U$  of keys,  $|S| = m \leq |U|$ .
- ▶ Array  $T[0, \dots, n-1]$  hash-table.
- ▶ Hash function  $h : U \rightarrow [0, \dots, n-1]$ .

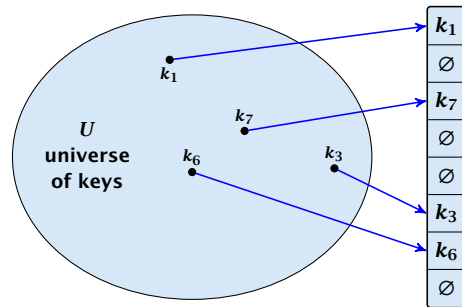
### The hash-function $h$ should fulfill:

- ▶ Fast to evaluate.
- ▶ Small storage requirement.
- ▶ Good distribution of elements over the whole table.



## Direct Addressing

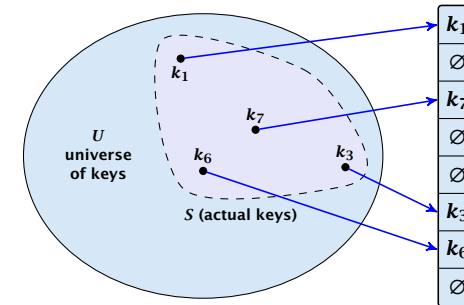
Ideally the hash function maps **all** keys to different memory locations.



This special case is known as **Direct Addressing**. It is usually very unrealistic as the universe of keys typically is quite large, and in particular larger than the available memory.

## Perfect Hashing

Suppose that we **know** the set  $S$  of actual keys (no insert/no delete). Then we may want to design a **simple** hash-function that maps all these keys to different memory locations.



Such a hash function  $h$  is called a **perfect hash function** for set  $S$ .

## Collisions

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

### Problem: Collisions

Usually the universe  $U$  is much larger than the table-size  $n$ .

Hence, there may be two elements  $k_1, k_2$  from the set  $S$  that map to the same memory location (i.e.,  $h(k_1) = h(k_2)$ ). This is called a **collision**.

## Collisions

Typically, collisions do not appear once the size of the set  $S$  of actual keys gets close to  $n$ , but already when  $|S| \geq \omega(\sqrt{n})$ .

### Lemma 22

The probability of having a collision when hashing  $m$  elements into a table of size  $n$  under **uniform hashing** is at least

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}}.$$

### Uniform hashing:

Choose a hash function uniformly at random from all functions  $f: U \rightarrow [0, \dots, n-1]$ .

## Collisions

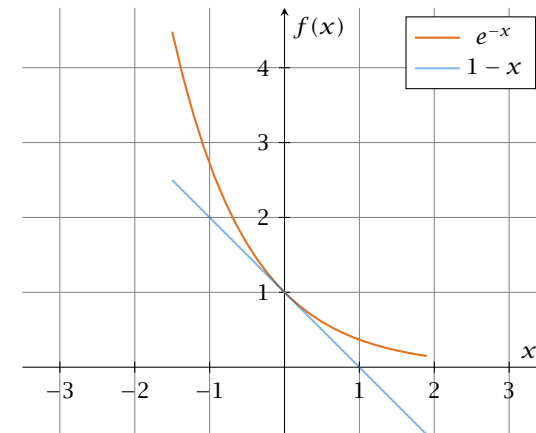
### Proof.

Let  $A_{m,n}$  denote the event that inserting  $m$  keys into a table of size  $n$  does **not** generate a collision. Then

$$\begin{aligned} \Pr[A_{m,n}] &= \prod_{\ell=1}^m \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \left(1 - \frac{j}{n}\right) \\ &\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}}. \end{aligned}$$

Here the first equality follows since the  $\ell$ -th element that is hashed has a probability of  $\frac{n-\ell+1}{n}$  to not generate a collision under the condition that the previous elements did not induce collisions.  $\square$

## Collisions



The inequality  $1 - x \leq e^{-x}$  is derived by stopping the Taylor-expansion of  $e^{-x}$  after the second term.

## Resolving Collisions

The methods for dealing with collisions can be classified into the two main types

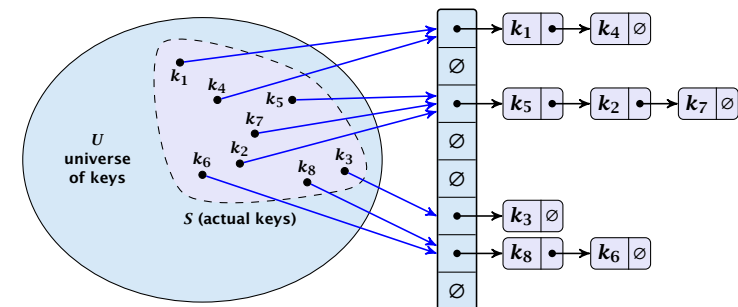
- ▶ **open addressing**, aka. closed hashing
- ▶ **hashing with chaining**, aka. closed addressing, open hashing.

There are applications e.g. computer chess where you do not resolve collisions at all.

## Hashing with Chaining

Arrange elements that map to the same position in a linear list.

- ▶ Access: compute  $h(x)$  and search list for  $\text{key}[x]$ .
- ▶ Insert: insert at the front of the list.



## Hashing with Chaining

Let  $A$  denote a strategy for resolving collisions. We use the following notation:

- ▶  $A^+$  denotes the average time for a **successful** search when using  $A$ ;
- ▶  $A^-$  denotes the average time for an **unsuccessful** search when using  $A$ ;
- ▶ We parameterize the complexity results in terms of  $\alpha := \frac{m}{n}$ , the so-called **fill factor** of the hash-table.

We assume **uniform hashing** for the following analysis.

## Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined. The average length of a list is  $\alpha = \frac{m}{n}$ . Hence, if  $A$  is the collision resolving strategy “Hashing with Chaining” we have

$$A^- = 1 + \alpha .$$

## Hashing with Chaining

For a successful search observe that we do **not** choose a list at random, but we consider a random key  $k$  in the hash-table and ask for the search-time for  $k$ .

This is 1 plus the number of elements that lie before  $k$  in  $k$ 's list.

Let  $k_\ell$  denote the  $\ell$ -th key inserted into the table.

Let for two keys  $k_i$  and  $k_j$ ,  $X_{ij}$  denote the indicator variable for the event that  $k_i$  and  $k_j$  hash to the same position. Clearly,  $\Pr[X_{ij} = 1] = 1/n$  for uniform hashing.

The expected successful search cost is

$$E \left[ \frac{1}{m} \sum_{i=1}^m \left( 1 + \sum_{j=i+1}^m X_{ij} \right) \right]$$

keys before  $k_i$ 
cost for key  $k_i$

## Hashing with Chaining

$$\begin{aligned} E \left[ \frac{1}{m} \sum_{i=1}^m \left( 1 + \sum_{j=i+1}^m X_{ij} \right) \right] &= \frac{1}{m} \sum_{i=1}^m \left( 1 + \sum_{j=i+1}^m E[X_{ij}] \right) \\ &= \frac{1}{m} \sum_{i=1}^m \left( 1 + \sum_{j=i+1}^m \frac{1}{n} \right) \\ &= 1 + \frac{1}{mn} \sum_{i=1}^m (m-i) \\ &= 1 + \frac{1}{mn} \left( m^2 - \frac{m(m+1)}{2} \right) \\ &= 1 + \frac{m-1}{2n} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2m} . \end{aligned}$$

Hence, the expected cost for a successful search is  $A^+ \leq 1 + \frac{\alpha}{2}$ .



## Hashing with Chaining

### Disadvantages:

- ▶ pointers increase memory requirements
- ▶ pointers may lead to bad cache efficiency

### Advantages:

- ▶ no à priori limit on the number of elements
- ▶ deletion can be implemented efficiently
- ▶ by using balanced trees instead of linked list one can also obtain worst-case guarantees.

## Open Addressing

All objects are stored in the table itself.

Define a function  $h(k, j)$  that determines the table-position to be examined in the  $j$ -th step. The values  $h(k, 0), \dots, h(k, n-1)$  must form a permutation of  $0, \dots, n-1$ .

**Search( $k$ ):** Try position  $h(k, 0)$ ; if it is empty your search fails; otw. continue with  $h(k, 1), h(k, 2), \dots$

**Insert( $x$ ):** Search until you find an empty slot; insert your element there. If your search reaches  $h(k, n-1)$ , and this slot is non-empty then your table is full.

## Open Addressing

Choices for  $h(k, j)$ :

- ▶ **Linear probing:**  
 $h(k, i) = h(k) + i \pmod n$   
(sometimes:  $h(k, i) = h(k) + ci \pmod n$ ).
- ▶ **Quadratic probing:**  
 $h(k, i) = h(k) + c_1i + c_2i^2 \pmod n$ .
- ▶ **Double hashing:**  
 $h(k, i) = h_1(k) + ih_2(k) \pmod n$ .

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing  $h_2(k)$  must be relatively prime to  $n$  (**teilerfremd**); for quadratic probing  $c_1$  and  $c_2$  have to be chosen carefully).

## Linear Probing

- ▶ Advantage: **Cache-efficiency**. The new probe position is very likely to be in the cache.
- ▶ Disadvantage: **Primary clustering**. Long sequences of occupied table-positions get longer as they have a larger probability to be hit. Furthermore, they can merge forming larger sequences.

### Lemma 23

Let  $L$  be the method of linear probing for resolving collisions:

$$L^+ \approx \frac{1}{2} \left( 1 + \frac{1}{1-\alpha} \right)$$

$$L^- \approx \frac{1}{2} \left( 1 + \frac{1}{(1-\alpha)^2} \right)$$

## Quadratic Probing

- ▶ Not as cache-efficient as Linear Probing.
- ▶ **Secondary clustering**: caused by the fact that all keys mapped to the same position have the same probe sequence.

### Lemma 24

Let  $Q$  be the method of quadratic probing for resolving collisions:

$$Q^+ \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

$$Q^- \approx \frac{1}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right) - \alpha$$

## Double Hashing

- ▶ Any probe into the hash-table usually creates a cache-miss.

### Lemma 25

Let  $A$  be the method of double hashing for resolving collisions:

$$D^+ \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

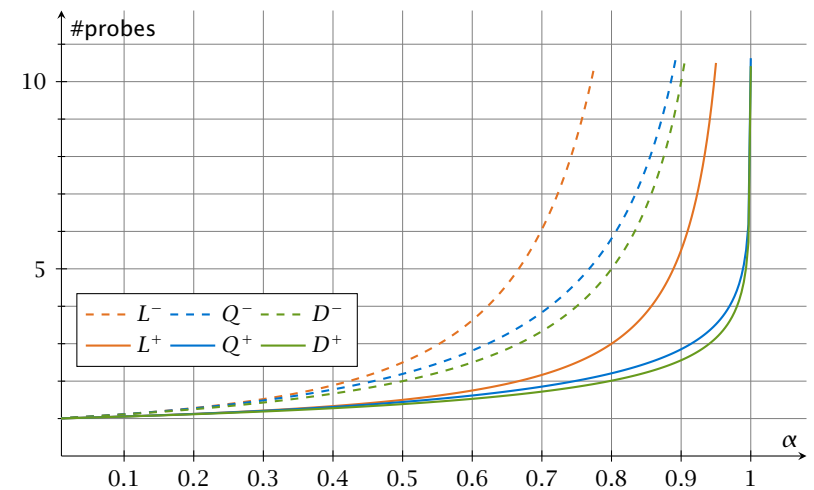
$$D^- \approx \frac{1}{1-\alpha}$$

## Open Addressing

Some values:

| $\alpha$ | Linear Probing |       | Quadratic Probing |       | Double Hashing |       |
|----------|----------------|-------|-------------------|-------|----------------|-------|
|          | $L^+$          | $L^-$ | $Q^+$             | $Q^-$ | $D^+$          | $D^-$ |
| 0.5      | 1.5            | 2.5   | 1.44              | 2.19  | 1.39           | 2     |
| 0.9      | 5.5            | 50.5  | 2.85              | 11.40 | 2.55           | 10    |
| 0.95     | 10.5           | 200.5 | 3.52              | 22.05 | 3.15           | 20    |

## Open Addressing



## Analysis of Idealized Open Address Hashing

We analyze the time for a search in a very idealized Open Addressing scheme.

- ▶ The probe sequence  $h(k, 0), h(k, 1), h(k, 2), \dots$  is equally likely to be any permutation of  $\{0, 1, \dots, n-1\}$ .

## Analysis of Idealized Open Address Hashing

Let  $X$  denote a random variable describing the number of probes in an **unsuccessful** search.

Let  $A_i$  denote the event that the  $i$ -th probe **occurs** and is to a non-empty slot.

$$\begin{aligned} \Pr[A_1 \cap A_2 \cap \dots \cap A_{i-1}] \\ = \Pr[A_1] \cdot \Pr[A_2 | A_1] \cdot \Pr[A_3 | A_1 \cap A_2] \cdot \\ \dots \cdot \Pr[A_{i-1} | A_1 \cap \dots \cap A_{i-2}] \end{aligned}$$

$$\begin{aligned} \Pr[X \geq i] &= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \dots \cdot \frac{m-i+2}{n-i+2} \\ &\leq \left(\frac{m}{n}\right)^{i-1} = \alpha^{i-1}. \end{aligned}$$

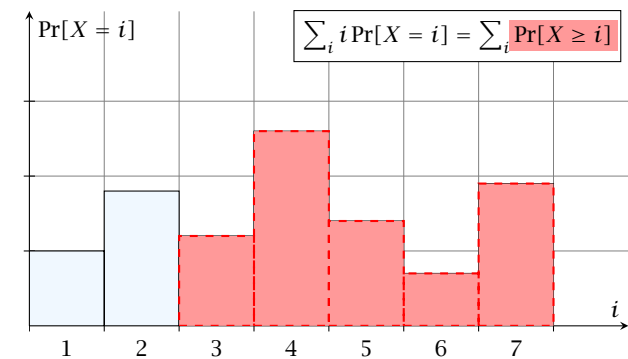
## Analysis of Idealized Open Address Hashing

$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}.$$

$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$$

## Analysis of Idealized Open Address Hashing

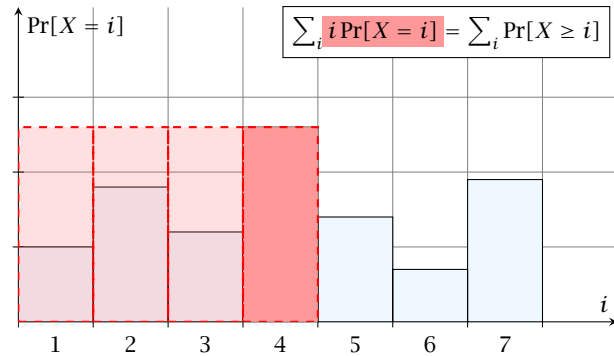
$i = 3$



The  $j$ -th rectangle appears in both sums  $j$  times. ( $j$  times in the first due to multiplication with  $j$ ; and  $j$  times in the second for summands  $i = 1, 2, \dots, j$ )

## Analysis of Idealized Open Address Hashing

$i = 4$



The  $j$ -th rectangle appears in both sums  $j$  times. ( $j$  times in the first due to multiplication with  $j$ ; and  $j$  times in the second for summands  $i = 1, 2, \dots, j$ )

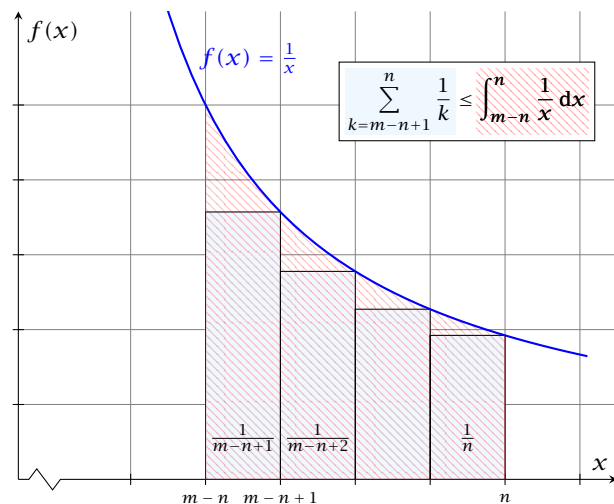
## Analysis of Idealized Open Address Hashing

The number of probes in a **successful** search for  $k$  is equal to the number of probes made in an unsuccessful search for  $k$  at the time that  $k$  is inserted.

Let  $k$  be the  $i + 1$ -st element. The expected time for a search for  $k$  is at most  $\frac{1}{1-i/n} = \frac{n}{n-i}$ .

$$\begin{aligned} \frac{1}{m} \sum_{i=0}^{m-1} \frac{n}{n-i} &= \frac{n}{m} \sum_{i=0}^{m-1} \frac{1}{n-i} = \frac{1}{\alpha} \sum_{k=n-m+1}^n \frac{1}{k} \\ &\leq \frac{1}{\alpha} \int_{n-m}^n \frac{1}{x} dx = \frac{1}{\alpha} \ln \frac{n}{n-m} = \frac{1}{\alpha} \ln \frac{1}{1-\alpha} . \end{aligned}$$

## Analysis of Idealized Open Address Hashing



## Deletions in Hashtables

**How do we delete in a hash-table?**

- ▶ For hashing with chaining this is not a problem. Simply search for the key, and delete the item in the corresponding list.
- ▶ For open addressing this is difficult.

## Deletions in Hashtables

- ▶ Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- ▶ One can delete an element by replacing it with a **deleted**-marker.
  - ▶ During an insertion if a **deleted**-marker is encountered an element can be inserted there.
  - ▶ During a search a **deleted**-marker must not be used to terminate the probe sequence.
- ▶ The table could fill up with **deleted**-markers leading to bad performance.
- ▶ If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.



## Deletions for Linear Probing

- ▶ For Linear Probing one can delete elements without using **deletion**-markers.
- ▶ Upon a deletion elements that are further down in the probe-sequence may be moved to guarantee that they are still found during a search.



## Deletions for Linear Probing

### Algorithm 12 delete( $p$ )

```
1:  $T[p] \leftarrow \text{null}$ 
2:  $p \leftarrow \text{succ}(p)$ 
3: while  $T[p] \neq \text{null}$  do
4:    $y \leftarrow T[p]$ 
5:    $T[p] \leftarrow \text{null}$ 
6:    $p \leftarrow \text{succ}(p)$ 
7:   insert( $y$ )
```

$p$  is the index into the table-cell that contains the object to be deleted.

**Pointers into the hash-table become invalid.**



## Universal Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that  $h$  is chosen randomly from all functions  $f: U \rightarrow [0, \dots, n-1]$  is clearly unrealistic as there are  $n^{|U|}$  such functions. Even writing down such a function would take  $|U| \log n$  bits.

Universal hashing tries to define a set  $\mathcal{H}$  of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from  $\mathcal{H}$ .



## Universal Hashing

### Definition 26

A class  $\mathcal{H}$  of hash-functions from the universe  $U$  into the set  $\{0, \dots, n-1\}$  is called **universal** if for all  $u_1, u_2 \in U$  with  $u_1 \neq u_2$

$$\Pr[h(u_1) = h(u_2)] \leq \frac{1}{n} ,$$

where the probability is w. r. t. the choice of a random hash-function from set  $\mathcal{H}$ .

Note that this means that the probability of a collision between two arbitrary elements is at most  $\frac{1}{n}$ .



## Universal Hashing

### Definition 27

A class  $\mathcal{H}$  of hash-functions from the universe  $U$  into the set  $\{0, \dots, n-1\}$  is called **2-independent** (pairwise independent) if the following two conditions hold

- ▶ For any key  $u \in U$ , and  $t \in \{0, \dots, n-1\}$   $\Pr[h(u) = t] = \frac{1}{n}$ , i.e., a key is distributed uniformly within the hash-table.
- ▶ For all  $u_1, u_2 \in U$  with  $u_1 \neq u_2$ , and for any two hash-positions  $t_1, t_2$ :

$$\Pr[h(u_1) = t_1 \wedge h(u_2) = t_2] \leq \frac{1}{n^2} .$$

This requirement clearly implies a universal hash-function.



## Universal Hashing

### Definition 28

A class  $\mathcal{H}$  of hash-functions from the universe  $U$  into the set  $\{0, \dots, n-1\}$  is called  **$k$ -independent** if for any choice of  $\ell \leq k$  distinct keys  $u_1, \dots, u_\ell \in U$ , and for any set of  $\ell$  not necessarily distinct hash-positions  $t_1, \dots, t_\ell$ :

$$\Pr[h(u_1) = t_1 \wedge \dots \wedge h(u_\ell) = t_\ell] \leq \frac{1}{n^\ell} ,$$

where the probability is w. r. t. the choice of a random hash-function from set  $\mathcal{H}$ .



## Universal Hashing

### Definition 29

A class  $\mathcal{H}$  of hash-functions from the universe  $U$  into the set  $\{0, \dots, n-1\}$  is called  **$(\mu, k)$ -independent** if for any choice of  $\ell \leq k$  distinct keys  $u_1, \dots, u_\ell \in U$ , and for any set of  $\ell$  not necessarily distinct hash-positions  $t_1, \dots, t_\ell$ :

$$\Pr[h(u_1) = t_1 \wedge \dots \wedge h(u_\ell) = t_\ell] \leq \frac{\mu}{n^\ell} ,$$

where the probability is w. r. t. the choice of a random hash-function from set  $\mathcal{H}$ .



## Universal Hashing

Let  $U := \{0, \dots, p-1\}$  for a prime  $p$ . Let  $\mathbb{Z}_p := \{0, \dots, p-1\}$ , and let  $\mathbb{Z}_p^* := \{1, \dots, p-1\}$  denote the set of invertible elements in  $\mathbb{Z}_p$ .

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

### Lemma 30

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from  $U$  to  $\{0, \dots, n-1\}$ .

## Universal Hashing

### Proof.

Let  $x, y \in U$  be two distinct keys. We have to show that the probability of a collision is only  $1/n$ .

$$\blacktriangleright ax + b \not\equiv ay + b \pmod{p}$$

$$\text{If } x \neq y \text{ then } (x - y) \not\equiv 0 \pmod{p}.$$

Multiplying with  $a \not\equiv 0 \pmod{p}$  gives

$$a(x - y) \not\equiv 0 \pmod{p}$$

where we use that  $\mathbb{Z}_p$  is a field (Körper) and, hence, has no zero divisors (nullteilerfrei).

## Universal Hashing

- ▶ The hash-function does not generate collisions before the  $(\bmod n)$ -operation. Furthermore, every choice  $(a, b)$  is mapped to a different pair  $(t_x, t_y)$  with  $t_x := ax + b$  and  $t_y := ay + b$ .

This holds because we can compute  $a$  and  $b$  when given  $t_x$  and  $t_y$ :

$$t_x \equiv ax + b \pmod{p}$$

$$t_y \equiv ay + b \pmod{p}$$

$$t_x - t_y \equiv a(x - y) \pmod{p}$$

$$t_y \equiv ay + b \pmod{p}$$

$$a \equiv (t_x - t_y)(x - y)^{-1} \pmod{p}$$

$$b \equiv t_y - ay \pmod{p}$$

## Universal Hashing

There is a one-to-one correspondence between hash-functions (pairs  $(a, b)$ ,  $a \neq 0$ ) and pairs  $(t_x, t_y)$ ,  $t_x \neq t_y$ .

Therefore, we can view the first step (before the  $\bmod n$ -operation) as choosing a pair  $(t_x, t_y)$ ,  $t_x \neq t_y$  uniformly at random.

What happens when we do the  $\bmod n$  operation?

Fix a value  $t_x$ . There are  $p-1$  possible values for choosing  $t_y$ .

From the range  $0, \dots, p-1$  the values  $t_x, t_x + n, t_x + 2n, \dots$  map to  $t_x$  after the modulo-operation. These are at most  $\lceil p/n \rceil$  values.

## Universal Hashing

As  $t_y \neq t_x$  there are

$$\left\lfloor \frac{p}{n} \right\rfloor - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing  $t_y$  such that the final hash-value creates a collision.

This happens with probability at most  $\frac{1}{n}$ .

## Universal Hashing

It is also possible to show that  $\mathcal{H}$  is an (almost) pairwise independent class of hash-functions.

$$\frac{\left\lfloor \frac{p}{n} \right\rfloor^2}{p(p-1)} \leq \Pr_{t_x \neq t_y \in \mathbb{Z}_p^2} \left[ \begin{array}{l} t_x \bmod n = h_1 \\ t_y \bmod n = h_2 \end{array} \right] \leq \frac{\left\lfloor \frac{p}{n} \right\rfloor^2}{p(p-1)}$$

Note that the middle is the probability that  $h(x) = h_1$  and  $h(y) = h_2$ . The total number of choices for  $(t_x, t_y)$  is  $p(p-1)$ . The number of choices for  $t_x$  ( $t_y$ ) such that  $t_x \bmod n = h_1$  ( $t_y \bmod n = h_2$ ) lies between  $\left\lfloor \frac{p}{n} \right\rfloor$  and  $\left\lceil \frac{p}{n} \right\rceil$ .

## Universal Hashing

### Definition 31

Let  $d \in \mathbb{N}$ ;  $q \geq (d+1)n$  be a prime; and let  $\bar{a} \in \{0, \dots, q-1\}^{d+1}$ . Define for  $x \in \{0, \dots, q-1\}$

$$h_{\bar{a}}(x) := \left( \sum_{i=0}^d a_i x^i \bmod q \right) \bmod n .$$

Let  $\mathcal{H}_n^d := \{h_{\bar{a}} \mid \bar{a} \in \{0, \dots, q-1\}^{d+1}\}$ . The class  $\mathcal{H}_n^d$  is  $(e, d+1)$ -independent.

Note that in the previous case we had  $d = 1$  and chose  $a_d \neq 0$ .

## Universal Hashing

For the coefficients  $\bar{a} \in \{0, \dots, q-1\}^{d+1}$  let  $f_{\bar{a}}$  denote the polynomial

$$f_{\bar{a}}(x) = \left( \sum_{i=0}^d a_i x^i \right) \bmod q$$

The polynomial is defined by  $d+1$  distinct points.



## Universal Hashing

Fix  $\ell \leq d + 1$ ; let  $x_1, \dots, x_\ell \in \{0, \dots, q - 1\}$  be keys, and let  $t_1, \dots, t_\ell$  denote the corresponding hash-function values.

Let  $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i \text{ for all } i \in \{1, \dots, \ell\}\}$

Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \underbrace{\{t_i + \alpha \cdot n \mid \alpha \in \{0, \dots, \lceil \frac{q}{n} \rceil - 1\}\}}_{=: B_i}$$

In order to obtain the cardinality of  $A^\ell$  we choose our polynomial by fixing  $d + 1$  points.

We first fix the values for inputs  $x_1, \dots, x_\ell$ .

We have

$$|B_1| \cdot \dots \cdot |B_\ell|$$

possibilities to do this (so that  $h_{\bar{a}}(x_i) = t_i$ ).

- $A^\ell$  denotes the set of hash-functions such that every  $x_i$  hits its pre-defined position  $t_i$ .
- $B_i$  is the set of positions that  $f_{\bar{a}}$  can hit so that  $h_{\bar{a}}$  still hits  $t_i$ .

## Universal Hashing

Now, we choose  $d - \ell + 1$  other inputs and choose their value arbitrarily. We have  $q^{d-\ell+1}$  possibilities to do this.

Therefore we have

$$|B_1| \cdot \dots \cdot |B_\ell| \cdot q^{d-\ell+1} \leq \lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}$$

possibilities to choose  $\bar{a}$  such that  $h_{\bar{a}} \in A^\ell$ .



## Universal Hashing

Therefore the probability of choosing  $h_{\bar{a}}$  from  $A^\ell$  is only

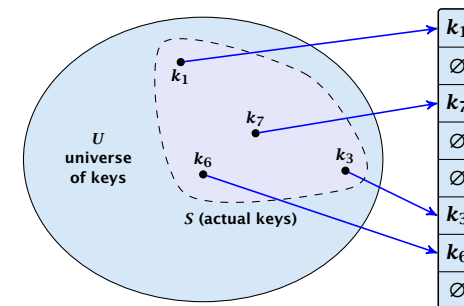
$$\begin{aligned} \frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} &\leq \frac{(\frac{q+n}{n})^\ell}{q^\ell} \leq \left(\frac{q+n}{q}\right)^\ell \cdot \frac{1}{n^\ell} \\ &\leq \left(1 + \frac{1}{q}\right)^\ell \cdot \frac{1}{n^\ell} \leq \frac{e}{n^\ell}. \end{aligned}$$

This shows that the  $\mathcal{H}$  is  $(e, d + 1)$ -universal.

The last step followed from  $q \geq (d + 1)n$ , and  $\ell \leq d + 1$ .

## Perfect Hashing

Suppose that we **know** the set  $S$  of actual keys (no insert/no delete). Then we may want to design a **simple** hash-function that maps all these keys to different memory locations.



## Perfect Hashing

Let  $m = |S|$ . We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$E[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n}.$$

If we choose  $n = m^2$  the **expected number** of collisions is strictly less than  $\frac{1}{2}$ .

Can we get an upper bound on the **probability of having collisions**?

The probability of having **1** or more collisions can be at most  $\frac{1}{2}$  as otherwise the expectation would be larger than  $\frac{1}{2}$ .

## Perfect Hashing

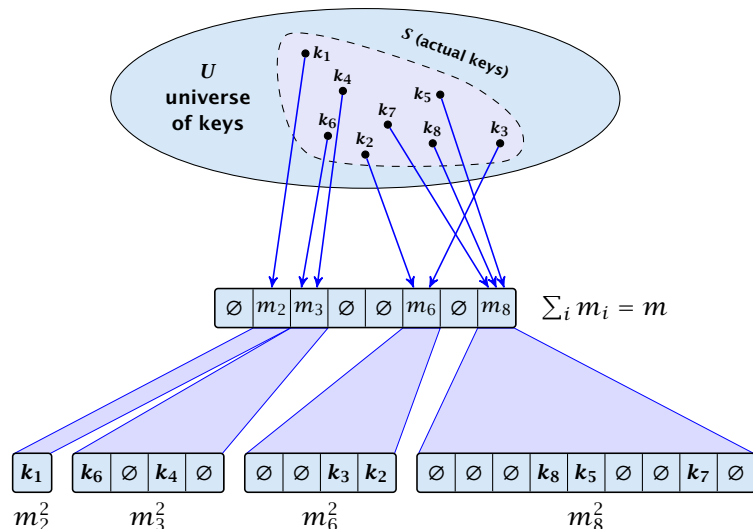
We can find such a hash-function by a few trials.

However, a hash-table size of  $n = m^2$  is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from  $S$  to  $m$  buckets.

Let  $m_j$  denote the number of items that are hashed to the  $j$ -th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size  $m_j^2$ . The second function can be chosen such that all elements are mapped to different locations.

## Perfect Hashing



## Perfect Hashing

The total memory that is required by all hash-tables is  $\mathcal{O}(\sum_j m_j^2)$ . Note that  $m_j$  is a random variable.

$$\begin{aligned} E\left[\sum_j m_j^2\right] &= E\left[2 \sum_j \binom{m_j}{2} + \sum_j m_j\right] \\ &= 2E\left[\sum_j \binom{m_j}{2}\right] + E\left[\sum_j m_j\right] \end{aligned}$$

The first expectation is simply the expected number of collisions, for the first level. Since we use universal hashing we have

$$= 2 \binom{m}{2} \frac{1}{m} + m = 2m - 1.$$

## Perfect Hashing

We need only  $\mathcal{O}(m)$  time to construct a hash-function  $h$  with  $\sum_j m_j^2 = \mathcal{O}(4m)$ , because with probability at least  $1/2$  a random function from a universal family will have this property.

Then we construct a hash-table  $h_j$  for every bucket. This takes expected time  $\mathcal{O}(m_j)$  for every bucket. A random function  $h_j$  is collision-free with probability at least  $1/2$ . We need  $\mathcal{O}(m_j)$  to test this.

We only need that the hash-functions are chosen from a universal family!!!

## Cuckoo Hashing

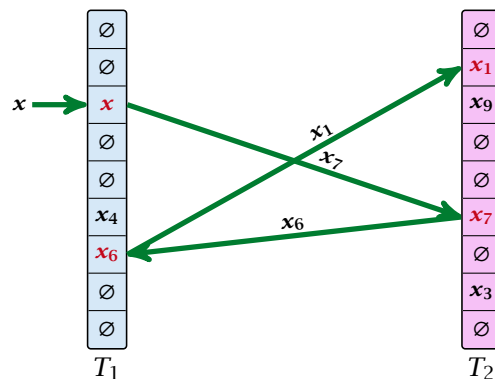
### Goal:

Try to generate a hash-table with constant worst-case search time in a dynamic scenario.

- ▶ Two hash-tables  $T_1[0, \dots, n-1]$  and  $T_2[0, \dots, n-1]$ , with hash-functions  $h_1$ , and  $h_2$ .
- ▶ An object  $x$  is either stored at location  $T_1[h_1(x)]$  or  $T_2[h_2(x)]$ .
- ▶ A search clearly takes constant time if the above constraint is met.

## Cuckoo Hashing

### Insert:



## Cuckoo Hashing

### Algorithm 13 Cuckoo-Insert( $x$ )

```
1: if  $T_1[h_1(x)] = x \vee T_2[h_2(x)] = x$  then return
2: steps  $\leftarrow 1$ 
3: while steps  $\leq$  maxsteps do
4:   exchange  $x$  and  $T_1[h_1(x)]$ 
5:   if  $x = \text{null}$  then return
6:   exchange  $x$  and  $T_2[h_2(x)]$ 
7:   if  $x = \text{null}$  then return
8:   steps  $\leftarrow$  steps + 1
9: rehash() // change hash-functions; rehash everything
10: Cuckoo-Insert( $x$ )
```

## Cuckoo Hashing

- ▶ We call one iteration through the while-loop a **step** of the algorithm.
- ▶ We call a sequence of iterations through the while-loop without the termination condition becoming true a **phase** of the algorithm.
- ▶ We say a phase is **successful** if it is not terminated by the **maxstep**-condition, but the while loop is left because  $x = \text{null}$ .

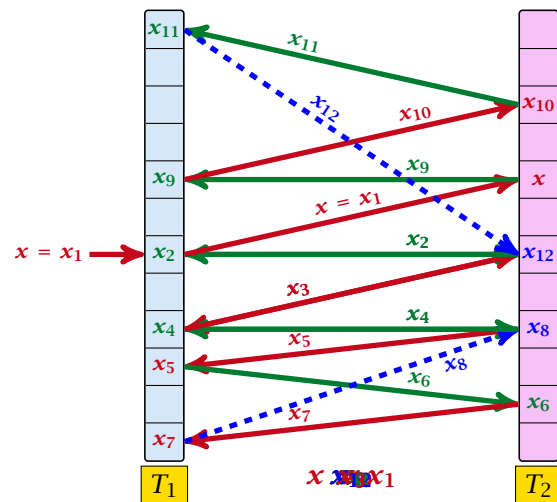
## Cuckoo Hashing

### What is the expected time for an insert-operation?

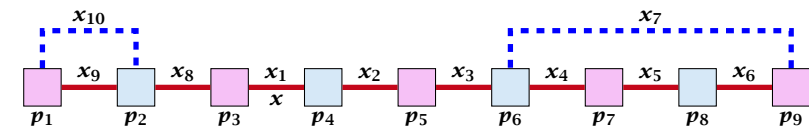
We first analyze the probability that we end-up in an infinite loop (that is then terminated after **maxsteps** steps).

Formally what is the probability to enter an infinite loop that touches  $s$  different keys?

## Cuckoo Hashing: Insert



## Cuckoo Hashing



A **cycle-structure of size  $s$**  is defined by

- ▶  $s - 1$  different cells (alternating btw. cells from  $T_1$  and  $T_2$ ).
- ▶  $s$  distinct keys  $x = x_1, x_2, \dots, x_s$ , linking the cells.
- ▶ The leftmost cell is “linked forward” to some cell on the right.
- ▶ The rightmost cell is “linked backward” to a cell on the left.
- ▶ One link represents key  $x$ ; this is where the counting starts.

## Cuckoo Hashing

A cycle-structure is **active** if for every key  $x_\ell$  (linking a cell  $p_i$  from  $T_1$  and a cell  $p_j$  from  $T_2$ ) we have

$$h_1(x_\ell) = p_i \quad \text{and} \quad h_2(x_\ell) = p_j$$

### Observation:

If during a phase the insert-procedure runs into a cycle there must exist an active cycle structure of size  $s \geq 3$ .



## Cuckoo Hashing

What is the probability that all keys in a cycle-structure of size  $s$  correctly map into their  $T_1$ -cell?

This probability is at most  $\frac{\mu}{n^s}$  since  $h_1$  is a  $(\mu, s)$ -independent hash-function.

What is the probability that all keys in the cycle-structure of size  $s$  correctly map into their  $T_2$ -cell?

This probability is at most  $\frac{\mu}{n^s}$  since  $h_2$  is a  $(\mu, s)$ -independent hash-function.

These events are independent.



## Cuckoo Hashing

The probability that a given cycle-structure of size  $s$  is active is at most  $\frac{\mu^2}{n^{2s}}$ .

What is the probability that **there exists** an active cycle structure of size  $s$ ?



## Cuckoo Hashing

The number of cycle-structures of size  $s$  is at most

$$s^3 \cdot n^{s-1} \cdot m^{s-1}.$$

- ▶ There are at most  $s^2$  possibilities where to attach the forward and backward links.
- ▶ There are at most  $s$  possibilities to choose where to place key  $x$ .
- ▶ There are  $m^{s-1}$  possibilities to choose the keys apart from  $x$ .
- ▶ There are  $n^{s-1}$  possibilities to choose the cells.



## Cuckoo Hashing

The probability that there exists an active cycle-structure is therefore at most

$$\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} = \frac{\mu^2}{nm} \sum_{s=3}^{\infty} s^3 \left(\frac{m}{n}\right)^s$$

$$\leq \frac{\mu^2}{m^2} \sum_{s=3}^{\infty} s^3 \left(\frac{1}{1+\epsilon}\right)^s \leq \mathcal{O}\left(\frac{1}{m^2}\right).$$

Here we used the fact that  $(1 + \epsilon)m \leq n$ .

Hence,

$$\Pr[\text{cycle}] = \mathcal{O}\left(\frac{1}{m^2}\right).$$

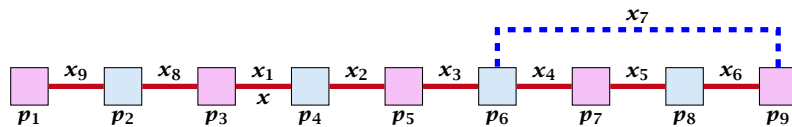


## Cuckoo Hashing

Now, we analyze the probability that a phase is not successful without running into a closed cycle.



## Cuckoo Hashing



Sequence of visited keys:

$$x = x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_3, x_2, x_1 = x, x_8, x_9, \dots$$



## Cuckoo Hashing

Consider the sequence of not necessarily distinct keys starting with  $x$  in the order that they are visited during the phase.

### Lemma 32

If the sequence is of length  $p$  then there exists a sub-sequence of at least  $\frac{p+2}{3}$  keys starting with  $x$  of *distinct* keys.



## Cuckoo Hashing

Taking  $x_1 \rightarrow \dots \rightarrow x_i$  twice, and  $x_1 \rightarrow x_{i+1} \rightarrow \dots \rightarrow x_j$  once gives  $2i + (j - i + 1) = i + j + 1 \geq p + 2$  keys. Hence, one of the sequences contains at least  $(p + 2)/3$  keys.

### Proof.

Let  $i$  be the number of keys (including  $x$ ) that we see before the first repeated key. Let  $j$  denote the total number of distinct keys.

The sequence is of the form:

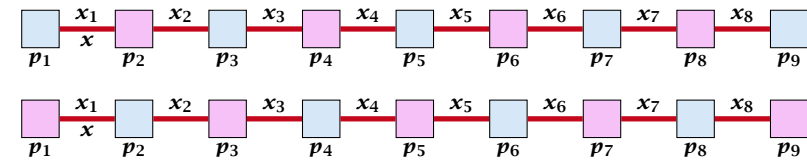
$x = x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_i \rightarrow x_r \rightarrow x_{r-1} \rightarrow \dots \rightarrow x_1 \rightarrow x_{i+1} \rightarrow \dots \rightarrow x_j$

As  $r \leq i - 1$  the length  $p$  of the sequence is

$$p = i + r + (j - i) \leq i + j - 1.$$

Either sub-sequence  $x_1 \rightarrow x_2 \rightarrow \dots \rightarrow x_i$  or sub-sequence  $x_1 \rightarrow x_{i+1} \rightarrow \dots \rightarrow x_j$  has at least  $\frac{p+2}{3}$  elements.  $\square$

## Cuckoo Hashing



A path-structure of size  $s$  is defined by

- ▶  $s + 1$  different cells (alternating btw. cells from  $T_1$  and  $T_2$ ).
- ▶  $s$  distinct keys  $x = x_1, x_2, \dots, x_s$ , linking the cells.
- ▶ The leftmost cell is either from  $T_1$  or  $T_2$ .

## Cuckoo Hashing

A path-structure is **active** if for every key  $x_\ell$  (linking a cell  $p_i$  from  $T_1$  and a cell  $p_j$  from  $T_2$ ) we have

$$h_1(x_\ell) = p_i \quad \text{and} \quad h_2(x_\ell) = p_j$$

### Observation:

If a phase takes at least  $t$  steps without running into a cycle there must exist an active path-structure of size  $(2t + 2)/3$ .

Note that we count **complete** steps. A search that touches  $2t$  or  $2t + 1$  keys takes  $t$  steps.

## Cuckoo Hashing

The probability that a given path-structure of size  $s$  is active is at most  $\frac{\mu^2}{n^{2s}}$ .

The probability that there exists an active path-structure of size  $s$  is at most

$$2 \cdot n^{s+1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} \leq 2\mu^2 \left(\frac{m}{n}\right)^{s-1} \leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{s-1}$$

Plugging in  $s = (2t + 2)/3$  gives

$$\leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{(2t+2)/3-1} = 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{(2t-1)/3}.$$

## Cuckoo Hashing

We choose  $\text{maxsteps} \geq 3\ell/2 + 1/2$ . Then the probability that a phase terminates unsuccessfully without running into a cycle is at most

$$\begin{aligned} & \Pr[\text{unsuccessful} \mid \text{no cycle}] \\ & \leq \Pr[\exists \text{ active path-structure of size at least } \frac{2\text{maxsteps}+2}{3}] \\ & \leq \Pr[\exists \text{ active path-structure of size at least } \ell + 1] \\ & \leq \Pr[\exists \text{ active path-structure of size exactly } \ell + 1] \\ & \leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^\ell \leq \frac{1}{m^2} \end{aligned}$$

by choosing  $\ell \geq \log\left(\frac{1}{2\mu^2 m^2}\right) / \log\left(\frac{1}{1+\epsilon}\right) = \log(2\mu^2 m^2) / \log(1+\epsilon)$

This gives  $\text{maxsteps} = \Theta(\log m)$ .

Note that the existence of a path structure of size larger than  $s$  implies the existence of a path structure of size exactly  $s$ .

## Cuckoo Hashing

So far we estimated

$$\Pr[\text{cycle}] \leq \mathcal{O}\left(\frac{1}{m^2}\right)$$

and

$$\Pr[\text{unsuccessful} \mid \text{no cycle}] \leq \mathcal{O}\left(\frac{1}{m^2}\right)$$

Observe that

$$\begin{aligned} \Pr[\text{successful}] &= \Pr[\text{no cycle}] - \Pr[\text{unsuccessful} \mid \text{no cycle}] \\ &\geq c \cdot \Pr[\text{no cycle}] \end{aligned}$$

for a suitable constant  $c > 0$ .

This is a very weak (and trivial) statement but still sufficient for our asymptotic analysis.

## Cuckoo Hashing

The expected number of complete steps in the **successful phase** of an insert operation is:

$$\begin{aligned} & \mathbb{E}[\text{number of steps} \mid \text{phase successful}] \\ &= \sum_{t \geq 1} \Pr[\text{search takes at least } t \text{ steps} \mid \text{phase successful}] \end{aligned}$$

We have

$$\begin{aligned} & \Pr[\text{search at least } t \text{ steps} \mid \text{successful}] \\ &= \Pr[\text{search at least } t \text{ steps} \wedge \text{successful}] / \Pr[\text{successful}] \\ &\leq \frac{1}{c} \Pr[\text{search at least } t \text{ steps} \wedge \text{successful}] / \Pr[\text{no cycle}] \\ &\leq \frac{1}{c} \Pr[\text{search at least } t \text{ steps} \wedge \text{no cycle}] / \Pr[\text{no cycle}] \\ &= \frac{1}{c} \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] . \end{aligned}$$

$$\Pr[A \mid B] = \frac{\Pr[A \wedge B]}{\Pr[B]}$$

## Cuckoo Hashing

Hence,

$$\begin{aligned} & \mathbb{E}[\text{number of steps} \mid \text{phase successful}] \\ & \leq \frac{1}{c} \sum_{t \geq 1} \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \\ & \leq \frac{1}{c} \sum_{t \geq 1} 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{(2t-1)/3} = \frac{1}{c} \sum_{t \geq 0} 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{(2(t+1)-1)/3} \\ & = \frac{2\mu^2}{c(1+\epsilon)^{1/3}} \sum_{t \geq 0} \left(\frac{1}{(1+\epsilon)^{2/3}}\right)^t = \mathcal{O}(1) . \end{aligned}$$

This means the expected cost for a successful phase is constant (even after accounting for the cost of the incomplete step that finishes the phase).



## Cuckoo Hashing

A phase that is not successful induces cost for doing a complete rehash (this dominates the cost for the steps in the phase).

The probability that a phase is not successful is  $q = \mathcal{O}(1/m^2)$  (probability  $\mathcal{O}(1/m^2)$  of running into a cycle and probability  $\mathcal{O}(1/m^2)$  of reaching `maxsteps` without running into a cycle).

A rehash try requires  $m$  insertions and takes expected constant time per insertion. It fails with probability  $p := \mathcal{O}(1/m)$ .

The expected number of unsuccessful rehashes is  $\sum_{i \geq 1} p^i = \frac{1}{1-p} - 1 = \frac{p}{1-p} = \mathcal{O}(p)$ .

Therefore the expected cost for re-hashes is  $\mathcal{O}(m) \cdot \mathcal{O}(p) = \mathcal{O}(1)$ .

## Formal Proof

Let  $Y_i$  denote the event that the  $i$ -th rehash does not lead to a valid configuration (assuming  $i$ -th rehash occurs) (i.e., one of the  $m + 1$  insertions fails):

$$\Pr[Y_i] \leq (m + 1) \cdot \mathcal{O}(1/m^2) \leq \mathcal{O}(1/m) =: p .$$

Let  $Z_i$  denote the event that the  $i$ -th rehash occurs:

The 0-th (re)hash is the initial configuration when doing the insert.

$$\Pr[Z_i] \leq \Pr[\wedge_{j=0}^{i-1} Y_j] \leq p^i$$

Let  $X_i^s$ ,  $s \in \{1, \dots, m + 1\}$  denote the cost for inserting the  $s$ -th element during the  $i$ -th rehash (assuming  $i$ -th rehash occurs):

$$\begin{aligned} E[X_i^s] &= E[\text{steps} \mid \text{phase successful}] \cdot \Pr[\text{phase successful}] \\ &\quad + \text{maxsteps} \cdot \Pr[\text{not successful}] = \mathcal{O}(1) . \end{aligned}$$

The expected cost for all rehashes is

$$E\left[\sum_i \sum_s Z_i X_i^s\right]$$

Note that  $Z_i$  is independent of  $X_j^s$ ,  $j \geq i$  (however, it is not independent of  $X_j^s$ ,  $j < i$ ). Hence,

$$\begin{aligned} E\left[\sum_i \sum_s Z_i X_i^s\right] &= \sum_i \sum_s E[Z_i] \cdot E[X_i^s] \\ &\leq \mathcal{O}(m) \cdot \sum_i p^i \\ &\leq \mathcal{O}(m) \cdot \frac{p}{1-p} \\ &= \mathcal{O}(1) . \end{aligned}$$

## Cuckoo Hashing

**What kind of hash-functions do we need?**

Since `maxsteps` is  $\Theta(\log m)$  the largest size of a path-structure or cycle-structure contains just  $\Theta(\log m)$  different keys.

Therefore, it is sufficient to have  $(\mu, \Theta(\log m))$ -independent hash-functions.

## Cuckoo Hashing

How do we make sure that  $n \geq (1 + \epsilon)m$ ?

- ▶ Let  $\alpha := 1/(1 + \epsilon)$ .
- ▶ Keep track of the number of elements in the table. When  $m \geq \alpha n$  we double  $n$  and do a complete re-hash (table-expand).
- ▶ Whenever  $m$  drops below  $\alpha n/4$  we divide  $n$  by 2 and do a rehash (table-shrink).
- ▶ Note that right after a change in table-size we have  $m = \alpha n/2$ . In order for a table-expand to occur at least  $\alpha n/2$  insertions are required. Similar, for a table-shrink at least  $\alpha n/4$  deletions must occur.
- ▶ Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

## Cuckoo Hashing

### Lemma 33

*Cuckoo Hashing has an expected constant insert-time and a worst-case constant search-time.*

Note that the above lemma only holds if the fill-factor (number of keys/total number of hash-table slots) is at most  $\frac{1}{2(1+\epsilon)}$ .

The  $1/(2(1+\epsilon))$  fill-factor comes from the fact that the total hash-table is of size  $2n$  (because we have two tables of size  $n$ ); moreover  $m \leq (1+\epsilon)n$ .

## Hashing

### Bibliography

- [MS08] Kurt Mehlhorn, Peter Sanders: *Algorithms and Data Structures — The Basic Toolbox*, Springer, 2008
- [CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein: *Introduction to algorithms (3rd ed.)*, MIT Press and McGraw-Hill, 2009

Chapter 4 of [MS08] contains a detailed description about Hashing with Linear Probing and Hashing with Chaining. Also the Perfect Hashing scheme can be found there.

The analysis of Hashing with Chaining under the assumption of uniform hashing can be found in Chapter 11.2 of [CLRS90]. Chapter 11.3.3 describes Universal Hashing. Collision resolution with Open Addressing is described in Chapter 11.4. Chapter 11.5 describes the Perfect Hashing scheme.

Reference for Cuckoo Hashing???

## 8 Priority Queues

A **Priority Queue**  $S$  is a dynamic set data structure that supports the following operations:

- ▶  **$S$ .build( $x_1, \dots, x_n$ )**: Creates a data-structure that contains just the elements  $x_1, \dots, x_n$ .
- ▶  **$S$ .insert( $x$ )**: Adds element  $x$  to the data-structure.
- ▶ **element  $S$ .minimum()**: Returns an element  $x \in S$  with minimum key-value  $\text{key}[x]$ .
- ▶ **element  $S$ .delete-min()**: Deletes the element with minimum key-value from  $S$  and returns it.
- ▶ **boolean  $S$ .is-empty()**: Returns **true** if the data-structure is empty and false otherwise.

Sometimes we also have

- ▶  **$S$ .merge( $S'$ )**:  $S := S \cup S'$ ;  $S' := \emptyset$ .

## 8 Priority Queues

An **addressable Priority Queue** also supports:

- ▶ **handle  $S$ .insert( $x$ ):** Adds element  $x$  to the data-structure, and returns a **handle** to the object for future reference.
- ▶  **$S$ .delete( $h$ ):** Deletes element specified through handle  $h$ .
- ▶  **$S$ .decrease-key( $h, k$ ):** Decreases the key of the element specified by handle  $h$  to  $k$ . Assumes that the key is at least  $k$  before the operation.



## Dijkstra's Shortest Path Algorithm

### Algorithm 14 Shortest-Path( $G = (V, E, d), s \in V$ )

```
1: Input: weighted graph  $G = (V, E, d)$ ; start vertex  $s$ ;  
2: Output: key-field of every node contains distance from  $s$ ;  
3:  $S$ .build(); // build empty priority queue  
4: for all  $v \in V \setminus \{s\}$  do  
5:    $v$ .key  $\leftarrow \infty$ ;  
6:    $h_v \leftarrow S$ .insert( $v$ );  
7:  $s$ .key  $\leftarrow 0$ ;  $S$ .insert( $s$ );  
8: while  $S$ .is-empty() = false do  
9:    $v \leftarrow S$ .delete-min();  
10:  for all  $x \in V$  s.t.  $(v, x) \in E$  do  
11:    if  $x$ .key  $> v$ .key +  $d(v, x)$  then  
12:       $S$ .decrease-key( $h_x, v$ .key +  $d(v, x)$ );  
13:       $x$ .key  $\leftarrow v$ .key +  $d(v, x)$ ;
```



## Prim's Minimum Spanning Tree Algorithm

### Algorithm 15 Prim-MST( $G = (V, E, d), s \in V$ )

```
1: Input: weighted graph  $G = (V, E, d)$ ; start vertex  $s$ ;  
2: Output: pred-fields encode MST;  
3:  $S$ .build(); // build empty priority queue  
4: for all  $v \in V \setminus \{s\}$  do  
5:    $v$ .key  $\leftarrow \infty$ ;  
6:    $h_v \leftarrow S$ .insert( $v$ );  
7:  $s$ .key  $\leftarrow 0$ ;  $S$ .insert( $s$ );  
8: while  $S$ .is-empty() = false do  
9:    $v \leftarrow S$ .delete-min();  
10:  for all  $x \in V$  s.t.  $\{v, x\} \in E$  do  
11:    if  $x$ .key  $> d(v, x)$  then  
12:       $S$ .decrease-key( $h_x, d(v, x)$ );  
13:       $x$ .key  $\leftarrow d(v, x)$ ;  
14:       $x$ .pred  $\leftarrow v$ ;
```



## Analysis of Dijkstra and Prim

Both algorithms require:

- ▶ 1 build() operation
- ▶  $|V|$  insert() operations
- ▶  $|V|$  delete-min() operations
- ▶  $|V|$  is-empty() operations
- ▶  $|E|$  decrease-key() operations

**How good a running time can we obtain?**



## 8 Priority Queues

| Operation    | Binary Heap   | BST        | Binomial Heap | Fibonacci Heap* |
|--------------|---------------|------------|---------------|-----------------|
| build        | $n$           | $n \log n$ | $n \log n$    | $n$             |
| minimum      | 1             | $\log n$   | $\log n$      | 1               |
| is-empty     | 1             | 1          | 1             | 1               |
| insert       | $\log n$      | $\log n$   | $\log n$      | 1               |
| delete       | $\log n^{**}$ | $\log n$   | $\log n$      | $\log n$        |
| delete-min   | $\log n$      | $\log n$   | $\log n$      | $\log n$        |
| decrease-key | $\log n$      | $\log n$   | $\log n$      | 1               |
| merge        | $n$           | $n \log n$ | $\log n$      | 1               |

Note that most applications use **build()** only to create an empty heap which then costs time 1.

\* Fibonacci heaps only give an amortized guarantee.

\*\* The standard version of binary heaps is not addressable. Hence, it does not support a delete.

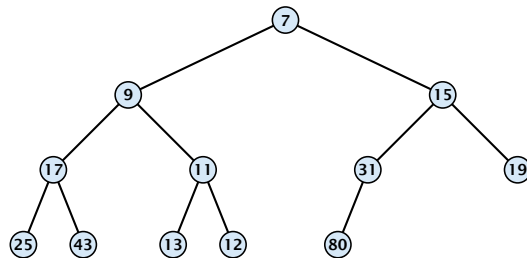
## 8 Priority Queues

Using Binary Heaps, Prim and Dijkstra run in time  $\mathcal{O}((|V| + |E|) \log |V|)$ .

Using Fibonacci Heaps, Prim and Dijkstra run in time  $\mathcal{O}(|V| \log |V| + |E|)$ .

### 8.1 Binary Heaps

- ▶ Nearly complete binary tree; only the last level is not full, and this one is filled from left to right.
- ▶ **Heap property:** A node's key is not larger than the key of one of its children.



### Binary Heaps

#### Operations:

- ▶ **minimum():** return the root-element. Time  $\mathcal{O}(1)$ .
- ▶ **is-empty():** check whether root-pointer is null. Time  $\mathcal{O}(1)$ .

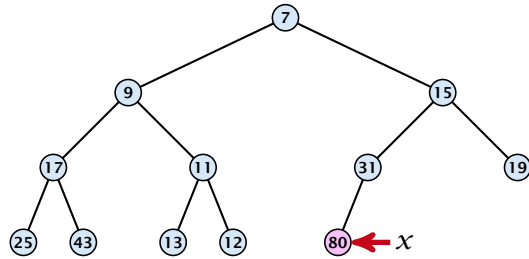
## 8.1 Binary Heaps

Maintain a pointer to the **last element**  $x$ .

- ▶ We can compute the predecessor of  $x$  (last element when  $x$  is deleted) in time  $\mathcal{O}(\log n)$ .

go up until the last edge used was a right edge.  
go left; go right until you reach a leaf

if you hit the root on the way up, go to the rightmost element



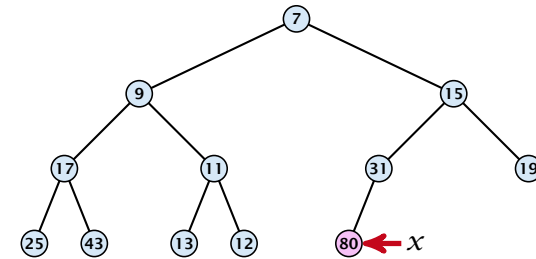
## 8.1 Binary Heaps

Maintain a pointer to the **last element**  $x$ .

- ▶ We can compute the successor of  $x$  (last element when an element is inserted) in time  $\mathcal{O}(\log n)$ .

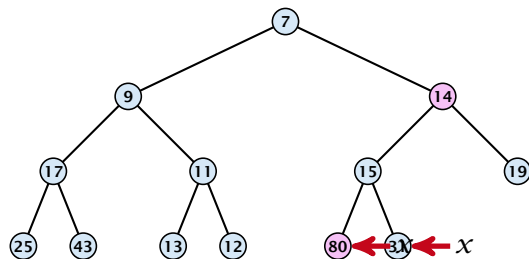
go up until the last edge used was a left edge.  
go right; go left until you reach a **null-pointer**.

if you hit the root on the way up, go to the leftmost element; insert a new element as a left child;



## Insert

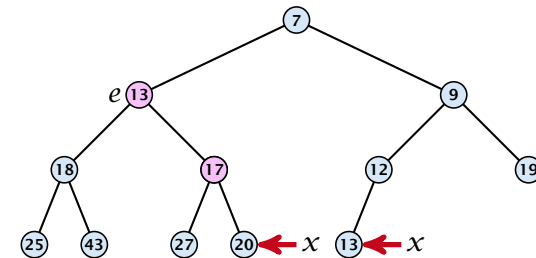
1. Insert element at successor of  $x$ .
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

## Delete

1. Exchange the element to be deleted with the element  $e$  pointed to by  $x$ .
2. Restore the heap-property for the element  $e$ .



At its new position  $e$  may either travel up or down in the tree (but not both directions).

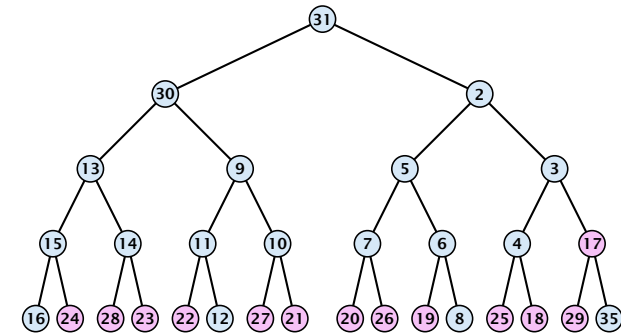
## Binary Heaps

### Operations:

- ▶ **minimum()**: return the root-element. Time  $\mathcal{O}(1)$ .
- ▶ **is-empty()**: check whether root-pointer is **null**. Time  $\mathcal{O}(1)$ .
- ▶ **insert( $k$ )**: insert at successor of  $x$  and bubble up. Time  $\mathcal{O}(\log n)$ .
- ▶ **delete( $h$ )**: swap with  $x$  and bubble up or sift-down. Time  $\mathcal{O}(\log n)$ .

## Build Heap

We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = \mathcal{O}(2^h) = \mathcal{O}(n)$$

## Binary Heaps

### Operations:

- ▶ **minimum()**: Return the root-element. Time  $\mathcal{O}(1)$ .
- ▶ **is-empty()**: Check whether root-pointer is **null**. Time  $\mathcal{O}(1)$ .
- ▶ **insert( $k$ )**: Insert at  $x$  and bubble up. Time  $\mathcal{O}(\log n)$ .
- ▶ **delete( $h$ )**: Swap with  $x$  and bubble up or sift-down. Time  $\mathcal{O}(\log n)$ .
- ▶ **build( $x_1, \dots, x_n$ )**: Insert elements arbitrarily; then do sift-down operations starting with the lowest layer in the tree. Time  $\mathcal{O}(n)$ .

## Binary Heaps

The standard implementation of binary heaps is via arrays. Let  $A[0, \dots, n-1]$  be an array

- ▶ The parent of  $i$ -th element is at position  $\lfloor \frac{i-1}{2} \rfloor$ .
- ▶ The left child of  $i$ -th element is at position  $2i+1$ .
- ▶ The right child of  $i$ -th element is at position  $2i+2$ .

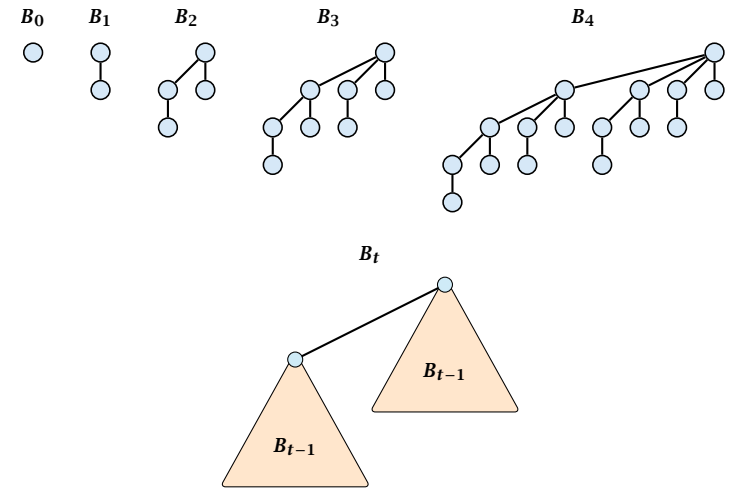
Finding the successor of  $x$  is much easier than in the description on the previous slide. Simply increase or decrease  $x$ .

The resulting binary heap is not addressable. The elements don't maintain their positions and therefore there are no stable handles.

## 8.2 Binomial Heaps

| Operation    | Binary Heap   | BST        | Binomial Heap              | Fibonacci Heap* |
|--------------|---------------|------------|----------------------------|-----------------|
| build        | $n$           | $n \log n$ | $n \log n$                 | $n$             |
| minimum      | 1             | $\log n$   | $\log n$                   | 1               |
| is-empty     | 1             | 1          | 1                          | 1               |
| insert       | $\log n$      | $\log n$   | $\log n$                   | 1               |
| delete       | $\log n^{**}$ | $\log n$   | $\log n$                   | $\log n$        |
| delete-min   | $\log n$      | $\log n$   | $\log n$                   | $\log n$        |
| decrease-key | $\log n$      | $\log n$   | $\log n$                   | 1               |
| merge        | $n$           | $n \log n$ | <b><math>\log n</math></b> | 1               |

## Binomial Trees

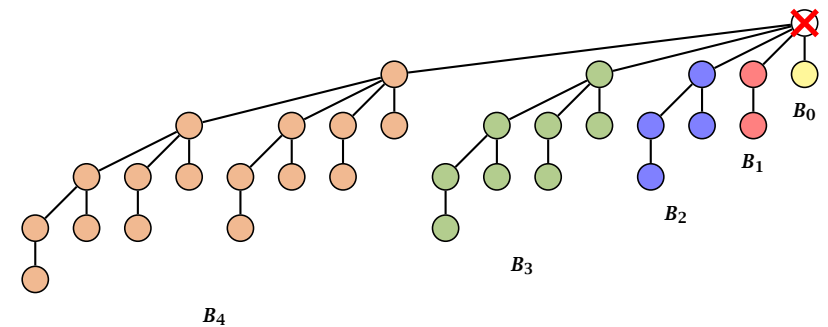


## Binomial Trees

### Properties of Binomial Trees

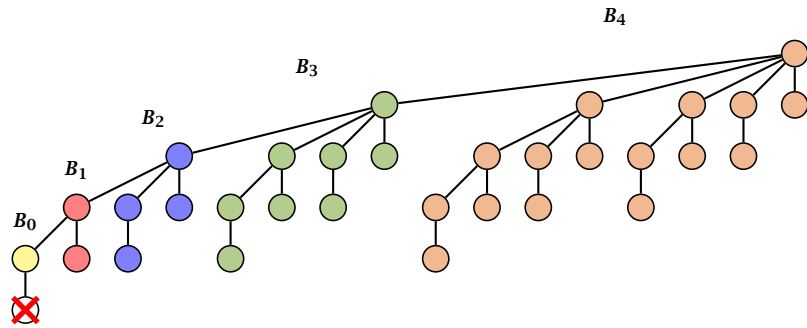
- ▶  $B_k$  has  $2^k$  nodes.
- ▶  $B_k$  has height  $k$ .
- ▶ The root of  $B_k$  has degree  $k$ .
- ▶  $B_k$  has  $\binom{k}{\ell}$  nodes on level  $\ell$ .
- ▶ Deleting the root of  $B_k$  gives trees  $B_0, B_1, \dots, B_{k-1}$ .

## Binomial Trees



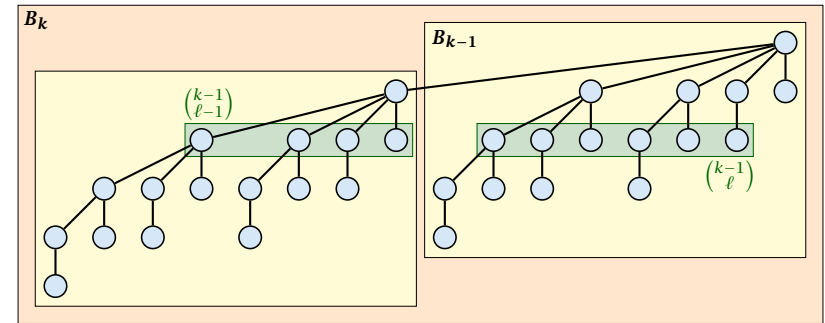
Deleting the root of  $B_5$  leaves sub-trees  $B_4, B_3, B_2, B_1$ , and  $B_0$ .

## Binomial Trees



Deleting the leaf furthest from the root (in  $B_5$ ) leaves a path that connects the roots of sub-trees  $B_4$ ,  $B_3$ ,  $B_2$ ,  $B_1$ , and  $B_0$ .

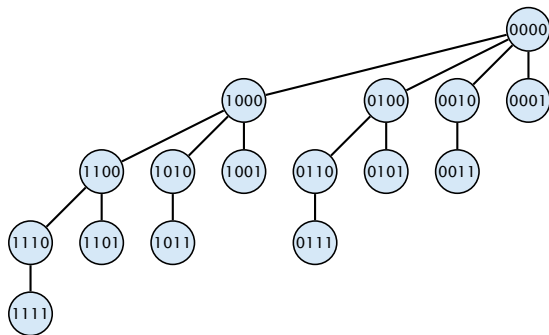
## Binomial Trees



The number of nodes on level  $\ell$  in tree  $B_k$  is therefore

$$\binom{k-1}{\ell-1} + \binom{k-1}{\ell} = \binom{k}{\ell}$$

## Binomial Trees



The binomial tree  $B_k$  is a sub-graph of the hypercube  $H_k$ .

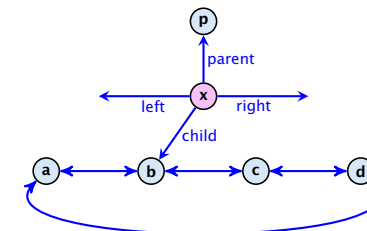
The parent of a node with label  $b_k, \dots, b_1$  is obtained by setting the least significant 1-bit to 0.

The  $\ell$ -th level contains nodes that have  $\ell$  1's in their label.

## 8.2 Binomial Heaps

How do we implement trees with non-constant degree?

- ▶ The children of a node are arranged in a **circular linked list**.
- ▶ A child-pointer points to an arbitrary node within the list.
- ▶ A parent-pointer points to the parent node.
- ▶ Pointers  $x$ .left and  $x$ .right point to the left and right sibling of  $x$  (if  $x$  does not have siblings then  $x$ .left =  $x$ .right =  $x$ ).

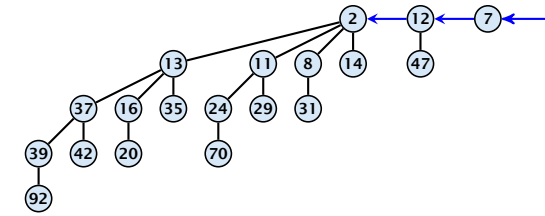




## 8.2 Binomial Heaps

- ▶ Given a pointer to a node  $x$  we can splice out the sub-tree rooted at  $x$  in constant time.
- ▶ We can add a child-tree  $T$  to a node  $x$  in constant time if we are given a pointer to  $x$  and a pointer to the root of  $T$ .

## Binomial Heap



In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees  $B_0$ ,  $B_1$ , and  $B_4$ .

## Binomial Heap: Merge

Given the number  $n$  of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

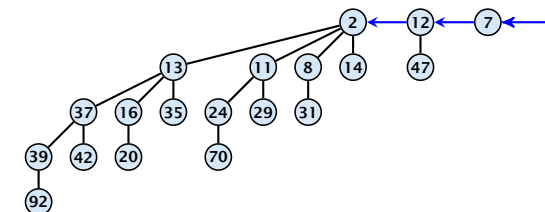
Let  $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$  denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then  $n = \sum_i 2^{k_i}$  must hold. But since the  $k_i$  are all distinct this means that the  $k_i$  define the non-zero bit-positions in the binary representation of  $n$ .

## Binomial Heap

### Properties of a heap with $n$ keys:

- ▶ Let  $n = b_a b_{a-1}, \dots, b_0$  denote binary representation of  $n$ .
- ▶ The heap contains tree  $B_i$  iff  $b_i = 1$ .
- ▶ Hence, at most  $\lfloor \log n \rfloor + 1$  trees.
- ▶ The minimum must be contained in one of the roots.
- ▶ The height of the largest tree is at most  $\lfloor \log n \rfloor$ .
- ▶ The trees are stored in a single-linked list; ordered by dimension/size.



## Binomial Heap: Merge

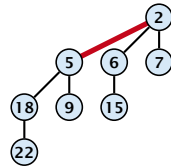
The merge-operation is instrumental for binomial heaps.

A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

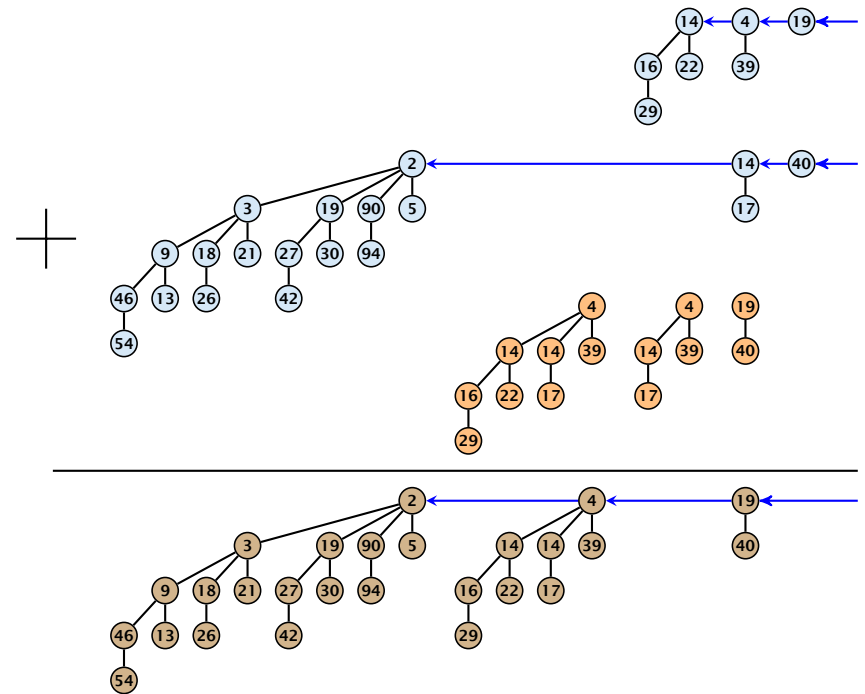
Note that we do not just do a concatenation as we want to keep the trees in the list sorted according to size.

Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.



For more trees the technique is analogous to binary addition.



## 8.2 Binomial Heaps

$S_1$ . merge( $S_2$ ):

- ▶ Analogous to binary addition.
- ▶ Time is proportional to the number of trees in both heaps.
- ▶ Time:  $\mathcal{O}(\log n)$ .

## 8.2 Binomial Heaps

**All other operations can be reduced to merge().**

$S$ . insert( $x$ ):

- ▶ Create a new heap  $S'$  that contains just the element  $x$ .
- ▶ Execute  $S$ .merge( $S'$ ).
- ▶ Time:  $\mathcal{O}(\log n)$ .

## 8.2 Binomial Heaps

### ***S*. minimum():**

- ▶ Find the minimum key-value among all roots.
- ▶ Time:  $\mathcal{O}(\log n)$ .

## 8.2 Binomial Heaps

### ***S*. delete-min():**

- ▶ Find the minimum key-value among all roots.
- ▶ Remove the corresponding tree  $T_{\min}$  from the heap.
- ▶ Create a new heap  $S'$  that contains the trees obtained from  $T_{\min}$  after deleting the root (note that these are just  $\mathcal{O}(\log n)$  trees).
- ▶ Compute  $S.\text{merge}(S')$ .
- ▶ Time:  $\mathcal{O}(\log n)$ .

## 8.2 Binomial Heaps

### ***S*. decrease-key(handle $h$ ):**

- ▶ Decrease the key of the element pointed to by  $h$ .
- ▶ Bubble the element up in the tree until the heap property is fulfilled.
- ▶ Time:  $\mathcal{O}(\log n)$  since the trees have height  $\mathcal{O}(\log n)$ .

## 8.2 Binomial Heaps

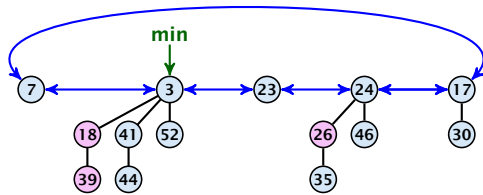
### ***S*. delete(handle $h$ ):**

- ▶ Execute  $S.\text{decrease-key}(h, -\infty)$ .
- ▶ Execute  $S.\text{delete-min}()$ .
- ▶ Time:  $\mathcal{O}(\log n)$ .

## 8.3 Fibonacci Heaps

Collection of trees that fulfill the heap property.

Structure is much more relaxed than binomial heaps.



## 8.3 Fibonacci Heaps

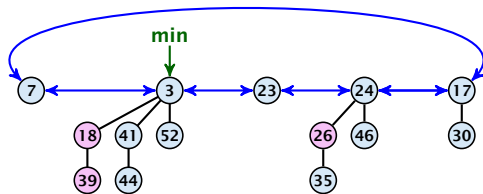
### Additional implementation details:

- ▶ Every node  $x$  stores its degree in a field  $x.degree$ . Note that this can be updated in constant time when adding a child to  $x$ .
- ▶ Every node stores a boolean value  $x.marked$  that specifies whether  $x$  is **marked** or not.

## 8.3 Fibonacci Heaps

### The potential function:

- ▶  $t(S)$  denotes the number of trees in the heap.
- ▶  $m(S)$  denotes the number of marked nodes.
- ▶ We use the potential function  $\Phi(S) = t(S) + 2m(S)$ .



The potential is  $\Phi(S) = 5 + 2 \cdot 3 = 11$ .

## 8.3 Fibonacci Heaps

We assume that one unit of potential can pay for a constant amount of work, where the constant is chosen “big enough” (to take care of the constants that occur).

To make this more explicit we use  $c$  to denote the amount of work that a unit of potential can pay for.

## 8.3 Fibonacci Heaps

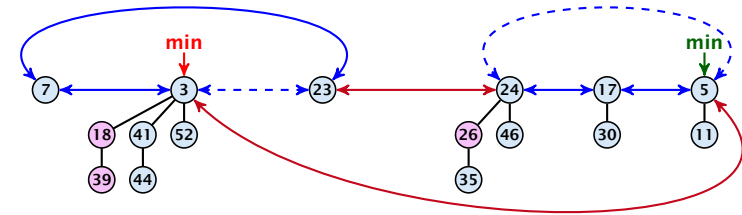
### S. minimum()

- ▶ Access through the min-pointer.
- ▶ Actual cost  $\mathcal{O}(1)$ .
- ▶ No change in potential.
- ▶ Amortized cost  $\mathcal{O}(1)$ .

## 8.3 Fibonacci Heaps

### S. merge( $S'$ )

- ▶ Merge the root lists.
- ▶ Adjust the min-pointer



- In the figure below the dashed edges are replaced by red edges.
- The minimum of the left heap becomes the new minimum of the merged heap.

### Running time:

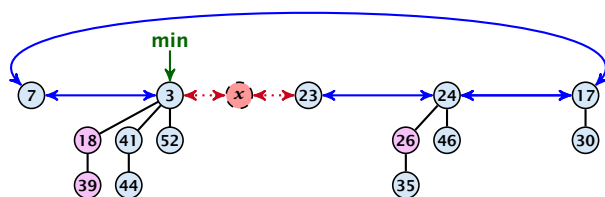
- ▶ Actual cost  $\mathcal{O}(1)$ .
- ▶ No change in potential.
- ▶ Hence, amortized cost is  $\mathcal{O}(1)$ .

## 8.3 Fibonacci Heaps

### S. insert( $x$ )

- ▶ Create a new tree containing  $x$ .
- ▶ Insert  $x$  into the root-list.
- ▶ Update min-pointer, if necessary.

$x$  is inserted next to the min-pointer as this is our entry point into the root-list.



### Running time:

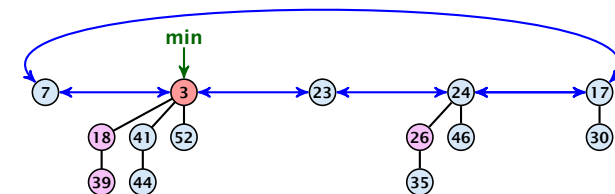
- ▶ Actual cost  $\mathcal{O}(1)$ .
- ▶ Change in potential is  $+1$ .
- ▶ Amortized cost is  $c + \mathcal{O}(1) = \mathcal{O}(1)$ .

## 8.3 Fibonacci Heaps

### S. delete-min( $x$ )

- ▶ Delete minimum; add child-trees to heap; time:  $D(\min) \cdot \mathcal{O}(1)$ .
- ▶ Update min-pointer; time:  $(t + D(\min)) \cdot \mathcal{O}(1)$ .

$D(\min)$  is the number of children of the node that stores the minimum.

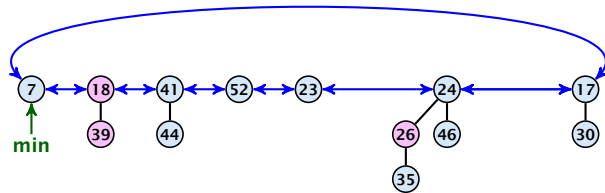


## 8.3 Fibonacci Heaps

$D(\min)$  is the number of children of the node that stores the minimum.

### S. delete-min( $x$ )

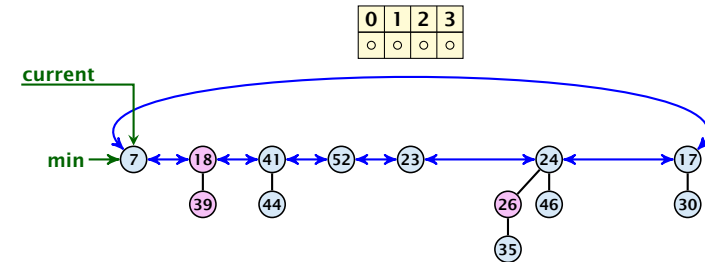
- ▶ Delete minimum; add child-trees to heap; time:  $D(\min) \cdot \mathcal{O}(1)$ .
- ▶ Update min-pointer; time:  $(t + D(\min)) \cdot \mathcal{O}(1)$ .



- ▶ Consolidate root-list so that no roots have the same degree. Time  $t \cdot \mathcal{O}(1)$  (see next slide).

## 8.3 Fibonacci Heaps

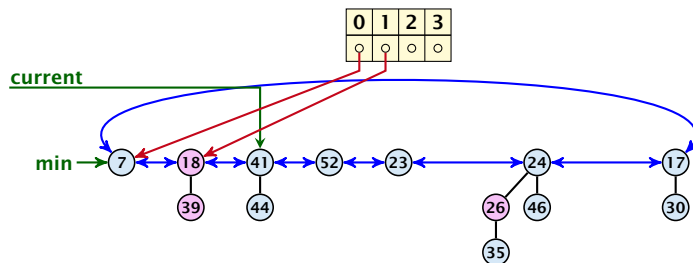
### Consolidate:



During the consolidation we traverse the root list. Whenever we discover two trees that have the same degree we merge these trees. In order to efficiently check whether two trees have the same degree, we use an array that contains for every degree value  $d$  a pointer to a tree left of the current pointer whose root has degree  $d$  (if such a tree exist).

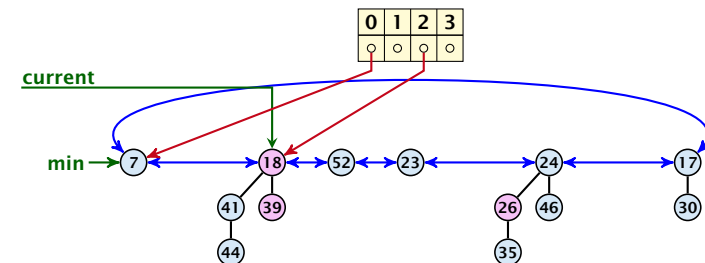
## 8.3 Fibonacci Heaps

### Consolidate:



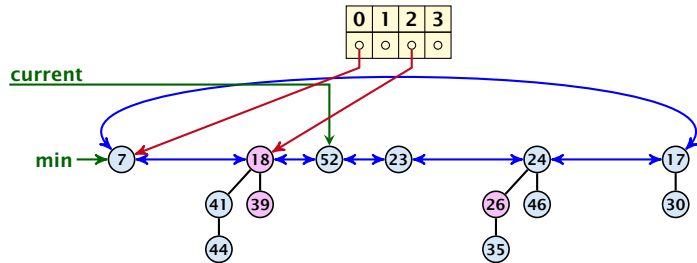
## 8.3 Fibonacci Heaps

### Consolidate:



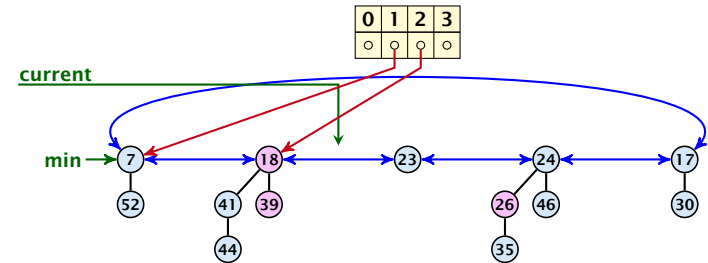
## 8.3 Fibonacci Heaps

Consolidate:



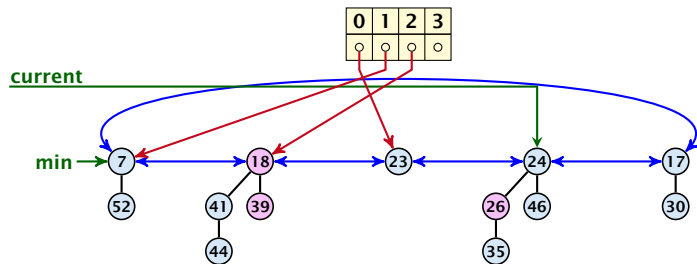
## 8.3 Fibonacci Heaps

Consolidate:



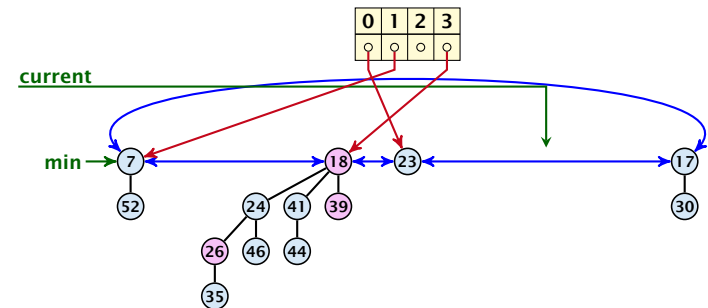
## 8.3 Fibonacci Heaps

Consolidate:



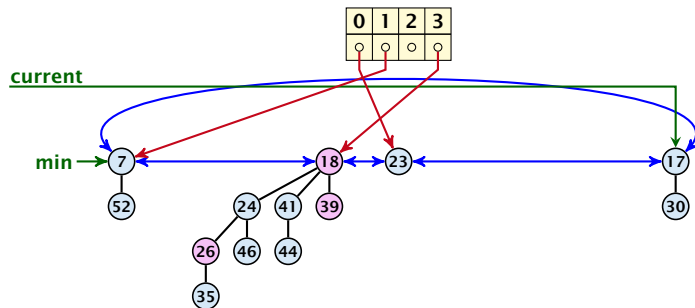
## 8.3 Fibonacci Heaps

Consolidate:



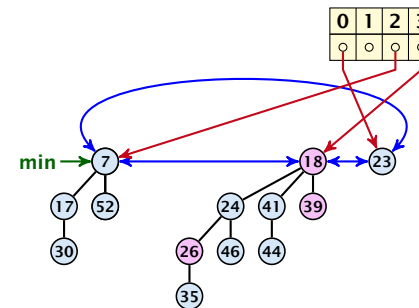
## 8.3 Fibonacci Heaps

Consolidate:



## 8.3 Fibonacci Heaps

Consolidate:



## 8.3 Fibonacci Heaps

Actual cost for delete-min()

- ▶ At most  $D_n + t$  elements in root-list before consolidate.
- ▶ Actual cost for a delete-min is at most  $\mathcal{O}(1) \cdot (D_n + t)$ .  
Hence, there exists  $c_1$  s.t. actual cost is at most  $c_1 \cdot (D_n + t)$ .

Amortized cost for delete-min()

- ▶  $t' \leq D_n + 1$  as degrees are different after consolidating.
- ▶ Therefore  $\Delta\Phi \leq D_n + 1 - t$ ;
- ▶ We can pay  $c \cdot (t - D_n - 1)$  from the potential decrease.
- ▶ The amortized cost is

$$c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1) \\ \leq (c_1 + c)D_n + (c_1 - c)t + c \leq 2c(D_n + 1) \leq \mathcal{O}(D_n)$$

for  $c \geq c_1$ .

$t$  and  $t'$  denote the number of trees before and after the delete-min() operation, respectively.  
 $D_n$  is an upper bound on the degree (i.e., number of children) of a tree node.

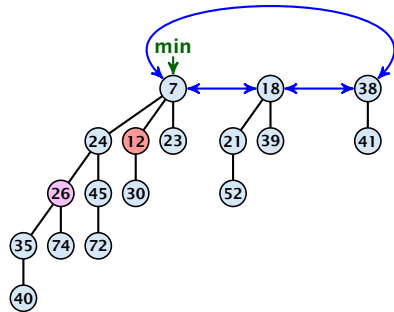
## 8.3 Fibonacci Heaps

If the input trees of the consolidation procedure are binomial trees (for example only singleton vertices) then the output will be a set of distinct binomial trees, and, hence, the Fibonacci heap will be (more or less) a Binomial heap right after the consolidation.

If we do not have delete or decrease-key operations then  $D_n \leq \log n$ .



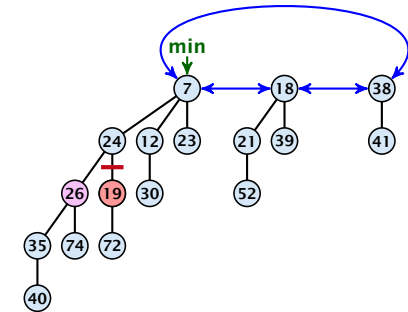
## Fibonacci Heaps: decrease-key(handle $h, v$ )



### Case 1: decrease-key does not violate heap-property

- ▶ Just decrease the key-value of element referenced by  $h$ . Nothing else to do.

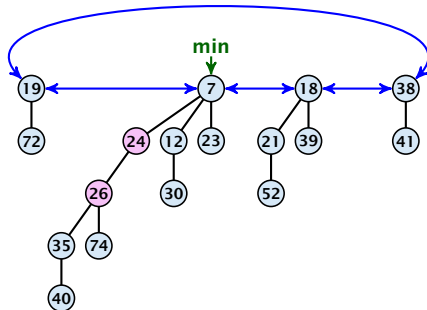
## Fibonacci Heaps: decrease-key(handle $h, v$ )



### Case 2: heap-property is violated, but parent is not marked

- ▶ Decrease key-value of element  $x$  reference by  $h$ .
- ▶ If the heap-property is violated, cut the parent edge of  $x$ , and make  $x$  into a root.
- ▶ Adjust min-pointers, if necessary.
- ▶ Mark the (previous) parent of  $x$  (unless it's a root).

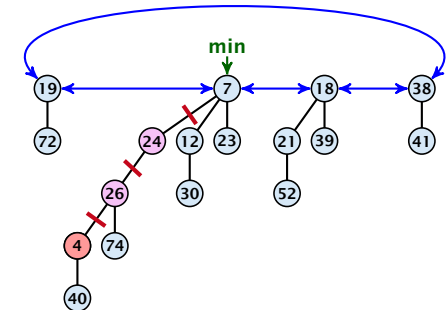
## Fibonacci Heaps: decrease-key(handle $h, v$ )



### Case 2: heap-property is violated, but parent is not marked

- ▶ Decrease key-value of element  $x$  reference by  $h$ .
- ▶ If the heap-property is violated, cut the parent edge of  $x$ , and make  $x$  into a root.
- ▶ Adjust min-pointers, if necessary.
- ▶ Mark the (previous) parent of  $x$  (unless it's a root).

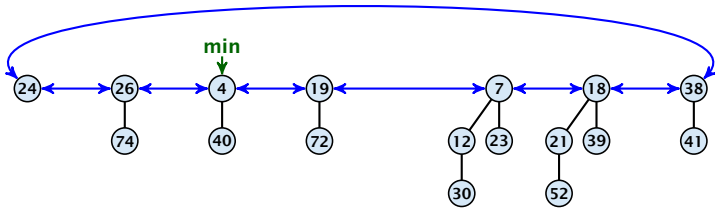
## Fibonacci Heaps: decrease-key(handle $h, v$ )



### Case 3: heap-property is violated, and parent is marked

- ▶ Decrease key-value of element  $x$  reference by  $h$ .
- ▶ Cut the parent edge of  $x$ , and make  $x$  into a root.
- ▶ Adjust min-pointers, if necessary.
- ▶ Continue cutting the parent until you arrive at an unmarked node.

## Fibonacci Heaps: decrease-key(handle $h, v$ )



### Case 3: heap-property is violated, and parent is marked

- ▶ Decrease key-value of element  $x$  reference by  $h$ .
- ▶ Cut the parent edge of  $x$ , and make  $x$  into a root.
- ▶ Adjust min-pointers, if necessary.
- ▶ Continue cutting the parent until you arrive at an unmarked node.

## Fibonacci Heaps: decrease-key(handle $h, v$ )

### Case 3: heap-property is violated, and parent is marked

- ▶ Decrease key-value of element  $x$  reference by  $h$ .
- ▶ Cut the parent edge of  $x$ , and make  $x$  into a root.
- ▶ Adjust min-pointers, if necessary.
- ▶ Execute the following:
 

```

 $p \leftarrow \text{parent}[x];$ 
while ( $p$  is marked)
     $pp \leftarrow \text{parent}[p];$ 
    cut of  $p$ ; make it into a root; unmark it;
     $p \leftarrow pp;$ 
if  $p$  is unmarked and not a root mark it;
            
```

Marking a node can be viewed as a first step towards becoming a root. The first time  $x$  loses a child it is marked; the second time it loses a child it is made into a root.

## Fibonacci Heaps: decrease-key(handle $h, v$ )

### Actual cost:

- ▶ Constant cost for decreasing the value.
- ▶ Constant cost for each of  $\ell$  cuts.
- ▶ Hence, cost is at most  $c_2 \cdot (\ell + 1)$ , for some constant  $c_2$ .

### Amortized cost:

- ▶  $t' = t + \ell$ , as every cut creates one new root.
- ▶  $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$ , since all but the first cut unmarks a node; the last cut may mark a node.

$$\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$$

- ▶ Amortized cost is at most

$$c_2(\ell + 1) + c(4 - \ell) \leq (c_2 - c)\ell + 4c + c_2 = \mathcal{O}(1),$$

if  $c \geq c_2$ .

$t$  and  $t'$ : number of trees before and after operation.  
 $m$  and  $m'$ : number of marked nodes before and after operation.

## Delete node

### $H$ . delete( $x$ ):

- ▶ decrease value of  $x$  to  $-\infty$ .
- ▶ delete-min.

### Amortized cost: $\mathcal{O}(D_n)$

- ▶  $\mathcal{O}(1)$  for decrease-key.
- ▶  $\mathcal{O}(D_n)$  for delete-min.

## 8.3 Fibonacci Heaps

### Lemma 34

Let  $x$  be a node with degree  $k$  and let  $y_1, \dots, y_k$  denote the children of  $x$  in the order that they were linked to  $x$ . Then

$$\text{degree}(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i - 2 & \text{if } i > 1 \end{cases}$$

The marking process is very important for the proof of this lemma. It ensures that a node can have lost at most one child since the last time it became a non-root node. When losing a first child the node gets marked; when losing the second child it is cut from the parent and made into a root.

## 8.3 Fibonacci Heaps

### Proof

- ▶ When  $y_i$  was linked to  $x$ , at least  $y_1, \dots, y_{i-1}$  were already linked to  $x$ .
- ▶ Hence, at this time  $\text{degree}(x) \geq i - 1$ , and therefore also  $\text{degree}(y_i) \geq i - 1$  as the algorithm links nodes of equal degree only.
- ▶ Since, then  $y_i$  has lost at most one child.
- ▶ Therefore,  $\text{degree}(y_i) \geq i - 2$ .

## 8.3 Fibonacci Heaps

- ▶ Let  $s_k$  be the minimum possible size of a sub-tree rooted at a node of degree  $k$  that can occur in a Fibonacci heap.
- ▶  $s_k$  monotonically increases with  $k$
- ▶  $s_0 = 1$  and  $s_1 = 2$ .

Let  $x$  be a degree  $k$  node of size  $s_k$  and let  $y_1, \dots, y_k$  be its children.

$$\begin{aligned} s_k &= 2 + \sum_{i=2}^k \text{size}(y_i) \\ &\geq 2 + \sum_{i=2}^k s_{i-2} \\ &= 2 + \sum_{i=0}^{k-2} s_i \end{aligned}$$

## 8.3 Fibonacci Heaps

$\phi = \frac{1}{2}(1 + \sqrt{5})$  denotes the *golden ratio*.  
Note that  $\phi^2 = 1 + \phi$ .

### Definition 35

Consider the following non-standard Fibonacci type sequence:

$$F_k = \begin{cases} 1 & \text{if } k = 0 \\ 2 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

### Facts:

1.  $F_k \geq \phi^k$ .
2. For  $k \geq 2$ :  $F_k = 2 + \sum_{i=0}^{k-2} F_i$ .

The above facts can be easily proved by induction. From this it follows that  $s_k \geq F_k \geq \phi^k$ , which gives that the maximum degree in a Fibonacci heap is logarithmic.

$$\begin{aligned}
k=0: & \quad 1 = F_0 \geq \Phi^0 = 1 \\
k=1: & \quad 2 = F_1 \geq \Phi^1 \approx 1.61 \\
k-2, k-1 \rightarrow k: & \quad F_k = F_{k-1} + F_{k-2} \geq \Phi^{k-1} + \Phi^{k-2} = \Phi^{k-2} \underbrace{(\Phi + 1)}_{\Phi^2} = \Phi^k
\end{aligned}$$

$$\begin{aligned}
k=2: & \quad 3 = F_2 = 2 + 1 = 2 + F_0 \\
k-1 \rightarrow k: & \quad F_k = F_{k-1} + F_{k-2} = 2 + \sum_{i=0}^{k-3} F_i + F_{k-2} = 2 + \sum_{i=0}^{k-2} F_i
\end{aligned}$$

## Priority Queues

### Bibliography

[CLRS90] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:  
*Introduction to algorithms (3rd ed.)*,  
MIT Press and McGraw-Hill, 2009

[MS08] Kurt Mehlhorn, Peter Sanders:  
*Algorithms and Data Structures — The Basic Toolbox*,  
Springer, 2008

Binary heaps are covered in [CLRS90] in combination with the heapsort algorithm in Chapter 6. Fibonacci heaps are covered in detail in Chapter 19. Problem 19-2 in this chapter introduces Binomial heaps.

Chapter 6 in [MS08] covers Priority Queues. Chapter 6.2.2 discusses Fibonacci heaps. Binomial heaps are dealt with in Exercise 6.11.

## 9 Union Find

**Union Find Data Structure  $\mathcal{P}$ :** Maintains a partition of **disjoint** sets over elements.

- ▶  **$\mathcal{P}$ . makeset( $x$ ):** Given an element  $x$ , adds  $x$  to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for  $x$  in the data-structure.
- ▶  **$\mathcal{P}$ . find( $x$ ):** Given a handle for an element  $x$ ; find the set that contains  $x$ . Returns a representative/identifier for this set.
- ▶  **$\mathcal{P}$ . union( $x, y$ ):** Given two elements  $x$ , and  $y$  that are currently in sets  $S_x$  and  $S_y$ , respectively, the function replaces  $S_x$  and  $S_y$  by  $S_x \cup S_y$  and returns an identifier for the new set.

## 9 Union Find

### Applications:

- ▶ Keep track of the connected components of a dynamic graph that changes due to insertion of nodes and edges.
- ▶ Kruskals Minimum Spanning Tree Algorithm

## 9 Union Find

### Algorithm 16 Kruskal-MST( $G = (V, E), w$ )

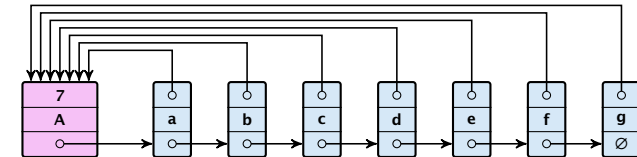
```

1:  $A \leftarrow \emptyset$ ;
2: for all  $v \in V$  do
3:    $v.set \leftarrow \mathcal{P}.makeset(v.label)$ 
4: sort edges in non-decreasing order of weight  $w$ 
5: for all  $(u, v) \in E$  in non-decreasing order do
6:   if  $\mathcal{P}.find(u.set) \neq \mathcal{P}.find(v.set)$  then
7:      $A \leftarrow A \cup \{(u, v)\}$ 
8:      $\mathcal{P}.union(u.set, v.set)$ 

```

## List Implementation

- ▶ The elements of a set are stored in a list; each node has a backward pointer to the head.
- ▶ The head of the list contains the identifier for the set and a field that stores the **size** of the set.

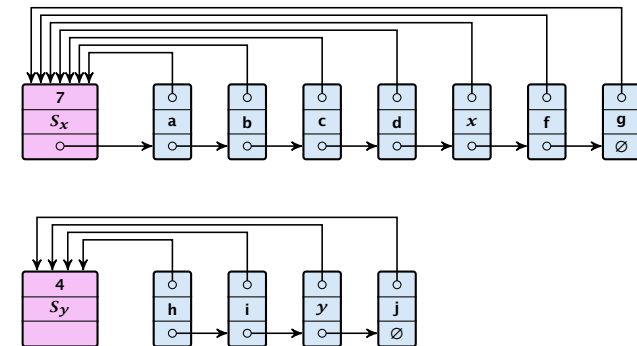


- ▶  $makeset(x)$  can be performed in constant time.
- ▶  $find(x)$  can be performed in constant time.

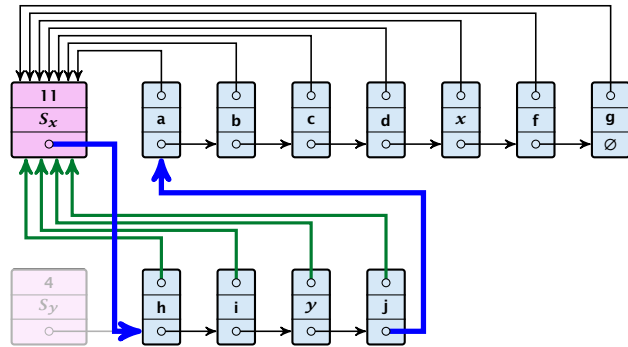
## List Implementation

### $union(x, y)$

- ▶ Determine sets  $S_x$  and  $S_y$ .
- ▶ Traverse the smaller list (say  $S_y$ ), and change all backward pointers to the head of list  $S_x$ .
- ▶ Insert list  $S_y$  at the head of  $S_x$ .
- ▶ Adjust the size-field of list  $S_x$ .
- ▶ Time:  $\min\{|S_x|, |S_y|\}$ .



## List Implementation



## List Implementation

### Running times:

- ▶  $\text{find}(x)$ : constant
- ▶  $\text{makeset}(x)$ : constant
- ▶  $\text{union}(x, y)$ :  $\mathcal{O}(n)$ , where  $n$  denotes the number of elements contained in the set system.

## List Implementation

### Lemma 36

The list implementation for the ADT union find fulfills the following amortized time bounds:

- ▶  $\text{find}(x)$ :  $\mathcal{O}(1)$ .
- ▶  $\text{makeset}(x)$ :  $\mathcal{O}(\log n)$ .
- ▶  $\text{union}(x, y)$ :  $\mathcal{O}(1)$ .

## The Accounting Method for Amortized Time Bounds

- ▶ There is a bank account for every element in the data structure.
- ▶ Initially the balance on all accounts is zero.
- ▶ Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- ▶ Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.
- ▶ If we can find a charging scheme that guarantees that balances always stay positive the amortized time bounds are proven.

## List Implementation

- ▶ For an operation whose actual cost exceeds the amortized cost we charge the **excess** to the elements involved.
- ▶ In total we will charge at most  $\mathcal{O}(\log n)$  to an element (regardless of the request sequence).
- ▶ For each element a makeset operation occurs as the first operation involving this element.
- ▶ We inflate the amortized cost of the makeset-operation to  $\Theta(\log n)$ , i.e., at this point we fill the bank account of the element to  $\Theta(\log n)$ .
- ▶ Later operations charge the account but the balance never drops below zero.

## List Implementation

**makeset(x):** The actual cost is  $\mathcal{O}(1)$ . Due to the cost inflation the amortized cost is  $\mathcal{O}(\log n)$ .

**find(x):** For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost:  $\mathcal{O}(1)$ .

**union(x, y):**

- ▶ If  $S_x = S_y$  the cost is constant; no bank accounts change.
- ▶ Otw. the actual cost is  $\mathcal{O}(\min\{|S_x|, |S_y|\})$ .
- ▶ Assume wlog. that  $S_x$  is the smaller set; let  $c$  denote the hidden constant, i.e., the actual cost is at most  $c \cdot |S_x|$ .
- ▶ Charge  $c$  to every element in set  $S_x$ .

## List Implementation

### Lemma 37

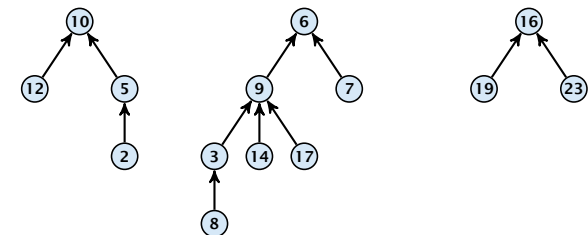
An element is charged at most  $\lceil \log_2 n \rceil$  times, where  $n$  is the total number of elements in the set system.

### Proof.

Whenever an element  $x$  is charged the number of elements in  $x$ 's set doubles. This can happen at most  $\lceil \log_2 n \rceil$  times.  $\square$

## Implementation via Trees

- ▶ Maintain nodes of a set in a tree.
- ▶ The root of the tree is the label of the set.
- ▶ Only pointer to parent exists; we cannot list all elements of a given set.
- ▶ Example:



Set system  $\{2, 5, 10, 12\}$ ,  $\{3, 6, 7, 8, 9, 14, 17\}$ ,  $\{16, 19, 23\}$ .

## Implementation via Trees

### **makeset( $x$ )**

- ▶ Create a singleton tree. Return pointer to the root.
- ▶ Time:  $\mathcal{O}(1)$ .

### **find( $x$ )**

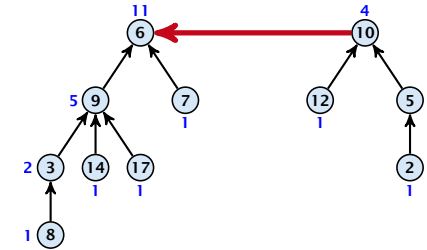
- ▶ Start at element  $x$  in the tree. Go upwards until you reach the root.
- ▶ Time:  $\mathcal{O}(\text{level}(x))$ , where  $\text{level}(x)$  is the distance of element  $x$  to the root in its tree. **Not constant.**

## Implementation via Trees

To support union we store the size of a tree in its root.

### **union( $x, y$ )**

- ▶ Perform  $a \leftarrow \text{find}(x)$ ;  $b \leftarrow \text{find}(y)$ . Then:  $\text{link}(a, b)$ .
- ▶  $\text{link}(a, b)$  attaches the **smaller** tree as the child of the larger.
- ▶ In addition it updates the size-field of the new root.



- ▶ Time: constant for  $\text{link}(a, b)$  plus two find-operations.

## Implementation via Trees

### **Lemma 38**

The running time (non-amortized!!!) for  $\text{find}(x)$  is  $\mathcal{O}(\log n)$ .

### **Proof.**

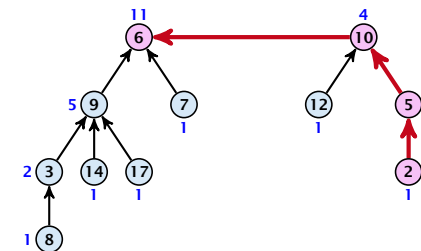
- ▶ When we attach a tree with root  $c$  to become a child of a tree with root  $p$ , then  $\text{size}(p) \geq 2 \text{size}(c)$ , where  $\text{size}$  denotes the value of the size-field right after the operation.
- ▶ After that the value of  $\text{size}(c)$  stays fixed, while the value of  $\text{size}(p)$  may still increase.
- ▶ Hence, at any point in time a tree fulfills  $\text{size}(p) \geq 2 \text{size}(c)$ , for any pair of nodes  $(p, c)$ , where  $p$  is a parent of  $c$ .

□

## Path Compression

### **find( $x$ ):**

- ▶ Go upward until you find the root.
- ▶ Re-attach all visited nodes as children of the root.
- ▶ Speeds up successive find-operations.



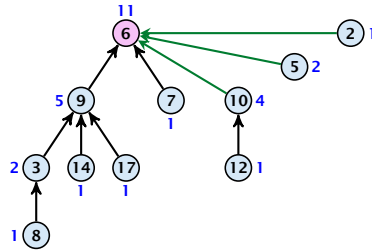
- ▶ Note that the size-fields now only give an upper bound on the size of a sub-tree.



## Path Compression

**find( $x$ ):**

- ▶ Go upward until you find the root.
- ▶ Re-attach all visited nodes as children of the root.
- ▶ Speeds up successive find-operations.



One could change the algorithm to update the size-fields. This could be done without asymptotically affecting the running time.

However, the only size-field that is actually required is the field at the root, which is always correct.

We will only use the other size-fields for the proof of Theorem 41.

- ▶ Note that the size-fields now only give an upper bound on the size of a sub-tree.

## Path Compression

Asymptotically the cost for a find-operation does not increase due to the path compression heuristic.

However, for a worst-case analysis there is no improvement on the running time. It can still happen that a find-operation takes time  $\mathcal{O}(\log n)$ .

## Amortized Analysis

**Definitions:**

- ▶  $\text{size}(v)$  := the number of nodes that were in the sub-tree rooted at  $v$  when  $v$  became the child of another node (or the number of nodes if  $v$  is the root).

Note that this is the same as the size of  $v$ 's subtree in the case that there are no find-operations.

- ▶  $\text{rank}(v) := \lfloor \log(\text{size}(v)) \rfloor$ .
- ▶  $\Rightarrow \text{size}(v) \geq 2^{\text{rank}(v)}$ .

**Lemma 39**

*The rank of a parent must be strictly larger than the rank of a child.*

## Amortized Analysis

**Lemma 40**

*There are at most  $n/2^s$  nodes of rank  $s$ .*

**Proof.**

- ▶ Let's say a node  $v$  sees node  $x$  if  $v$  is in  $x$ 's sub-tree at the time that  $x$  becomes a child.
- ▶ A node  $v$  sees at most one node of rank  $s$  during the running time of the algorithm.
- ▶ This holds because the rank-sequence of the roots of the different trees that contain  $v$  during the running time of the algorithm is a strictly increasing sequence.
- ▶ Hence, every node sees at most one rank  $s$  node, but every rank  $s$  node is seen by at least  $2^s$  different nodes.  $\square$

## Amortized Analysis

We define

$$\text{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\text{tow}(i-1)} & \text{otw.} \end{cases} \quad \text{tow}(i) = 2^{2^{2^{2^{\dots}}}} \text{ } i \text{ times}$$

and

$$\log^*(n) := \min\{i \mid \text{tow}(i) \geq n\} .$$

### Theorem 41

Union find with path compression fulfills the following amortized running times:

- ▶  $\text{makeset}(x) : \mathcal{O}(\log^*(n))$
- ▶  $\text{find}(x) : \mathcal{O}(\log^*(n))$
- ▶  $\text{union}(x, y) : \mathcal{O}(\log^*(n))$

## Amortized Analysis

In the following we assume  $n \geq 2$ .

### rank-group:

- ▶ A node with rank  $\text{rank}(v)$  is in rank group  $\log^*(\text{rank}(v))$ .
- ▶ The rank-group  $g = 0$  contains only nodes with rank 0 or rank 1.
- ▶ A rank group  $g \geq 1$  contains ranks  $\text{tow}(g-1) + 1, \dots, \text{tow}(g)$ .
- ▶ The maximum non-empty rank group is  $\log^*(\lfloor \log n \rfloor) \leq \log^*(n) - 1$  (which holds for  $n \geq 2$ ).
- ▶ Hence, the total number of rank-groups is at most  $\log^* n$ .

## Amortized Analysis

### Accounting Scheme:

- ▶ create an account for every find-operation
- ▶ create an account for every node  $v$

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from  $v$  to  $\text{parent}[v]$  as follows:

- ▶ If  $\text{parent}[v]$  is the root we charge the cost to the find-account.
- ▶ If the group-number of  $\text{rank}(v)$  is the same as that of  $\text{rank}(\text{parent}[v])$  (before starting path compression) we charge the cost to the node-account of  $v$ .
- ▶ Otherwise we charge the cost to the find-account.

## Amortized Analysis

### Observations:

- ▶ A find-account is charged at most  $\log^*(n)$  times (once for the root and at most  $\log^*(n) - 1$  times when increasing the rank-group).
- ▶ After a node  $v$  is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- ▶ After some charges to  $v$  the parent will be in a larger rank-group.  $\Rightarrow v$  will **never** be charged again.
- ▶ The total charge made to a node in rank-group  $g$  is at most  $\text{tow}(g) - \text{tow}(g-1) - 1 \leq \text{tow}(g)$ .

## Amortized Analysis

What is the total charge made to nodes?

- ▶ The total charge is at most

$$\sum_g n(g) \cdot \text{tow}(g) ,$$

where  $n(g)$  is the number of nodes in group  $g$ .

## Amortized Analysis

For  $g \geq 1$  we have

$$\begin{aligned} n(g) &\leq \sum_{s=\text{tow}(g-1)+1}^{\text{tow}(g)} \frac{n}{2^s} \leq \sum_{s=\text{tow}(g-1)+1}^{\infty} \frac{n}{2^s} \\ &= \frac{n}{2^{\text{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} = \frac{n}{2^{\text{tow}(g-1)+1}} \cdot 2 \\ &= \frac{n}{2^{\text{tow}(g-1)}} = \frac{n}{\text{tow}(g)} . \end{aligned}$$

Hence,

$$\sum_g n(g) \text{tow}(g) \leq n(0) \text{tow}(0) + \sum_{g \geq 1} n(g) \text{tow}(g) \leq n \log^*(n)$$

## Amortized Analysis

Without loss of generality we can assume that all **makeset**-operations occur at the start.

This means if we inflate the cost of **makeset** to  $\log^* n$  and add this to the node account of  $v$  then the balances of all node accounts will sum up to a positive value (this is sufficient to obtain an amortized bound).

## Amortized Analysis

The analysis is not tight. In fact it has been shown that the amortized time for the union-find data structure with path compression is  $\mathcal{O}(\alpha(m, n))$ , where  $\alpha(m, n)$  is the inverse Ackermann function which grows a lot lot slower than  $\log^* n$ . (Here, we consider the average running time of  $m$  operations on at most  $n$  elements).

There is also a lower bound of  $\Omega(\alpha(m, n))$ .

## Amortized Analysis

$$A(x, y) = \begin{cases} y + 1 & \text{if } x = 0 \\ A(x - 1, 1) & \text{if } y = 0 \\ A(x - 1, A(x, y - 1)) & \text{otw.} \end{cases}$$

$$\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log n\}$$

- ▶  $A(0, y) = y + 1$
- ▶  $A(1, y) = y + 2$
- ▶  $A(2, y) = 2y + 3$
- ▶  $A(3, y) = 2^{y+3} - 3$
- ▶  $A(4, y) = \underbrace{2^{2^{2^{\dots}}}}_{y+3 \text{ times}} - 3$

## Union Find

### Bibliography

- [CLRS90a] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest: *Introduction to Algorithms (1st ed.)*, MIT Press and McGraw-Hill, 1990
- [CLRS90b] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein: *Introduction to Algorithms (2nd ed.)*, MIT Press and McGraw-Hill, 2001
- [CLRS90c] Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein: *Introduction to Algorithms (3rd ed.)*, MIT Press and McGraw-Hill, 2009
- [AHU74] Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974

Union find data structures are discussed in Chapter 21 of [CLRS90b] and [CLRS90c] and in Chapter 22 of [CLRS90a]. The analysis of union by rank with path compression can be found in [CLRS90a] but neither in [CLRS90b] nor in [CLRS90c]. The latter books contains a more involved analysis that gives a better bound than  $\mathcal{O}(\log^* n)$ .

A description of the  $\mathcal{O}(\log^*)$ -bound can also be found in Chapter 4.8 of [AHU74].

## Part IV

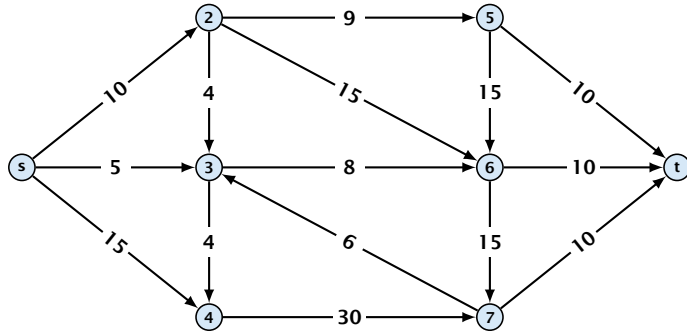
## Flows and Cuts

The following slides are partially based on slides by Kevin Wayne.

## 10 Introduction

### Flow Network

- ▶ directed graph  $G = (V, E)$ ; edge capacities  $c(e)$
- ▶ two special nodes: source  $s$ ; target  $t$ ;
- ▶ no edges entering  $s$  or leaving  $t$ ;
- ▶ at least for now: no parallel edges;



## Cuts

### Definition 42

An  $(s, t)$ -cut in the graph  $G$  is given by a set  $A \subset V$  with  $s \in A$  and  $t \in V \setminus A$ .

### Definition 43

The **capacity** of a cut  $A$  is defined as

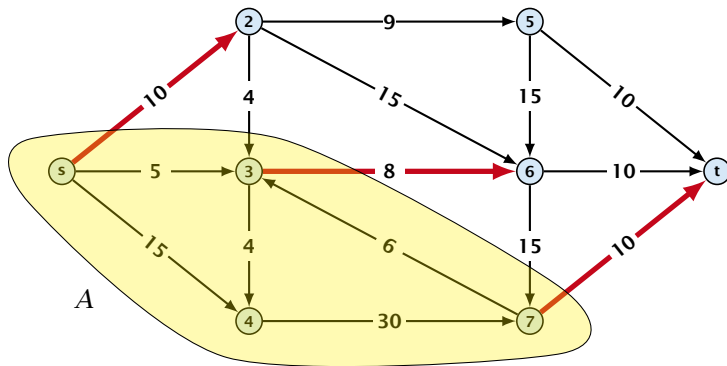
$$\text{cap}(A, V \setminus A) := \sum_{e \in \text{out}(A)} c(e),$$

where  $\text{out}(A)$  denotes the set of edges of the form  $A \times V \setminus A$  (i.e. edges leaving  $A$ ).

**Minimum Cut Problem:** Find an  $(s, t)$ -cut with minimum capacity.

## Cuts

### Example 44



The capacity of the cut is  $\text{cap}(A, V \setminus A) = 28$ .

## Flows

### Definition 45

An  $(s, t)$ -flow is a function  $f : E \rightarrow \mathbb{R}^+$  that satisfies

1. For each edge  $e$

$$0 \leq f(e) \leq c(e).$$

(capacity constraints)

2. For each  $v \in V \setminus \{s, t\}$

$$\sum_{e \in \text{out}(v)} f(e) = \sum_{e \in \text{into}(v)} f(e).$$

(flow conservation constraints)

## Flows

### Definition 46

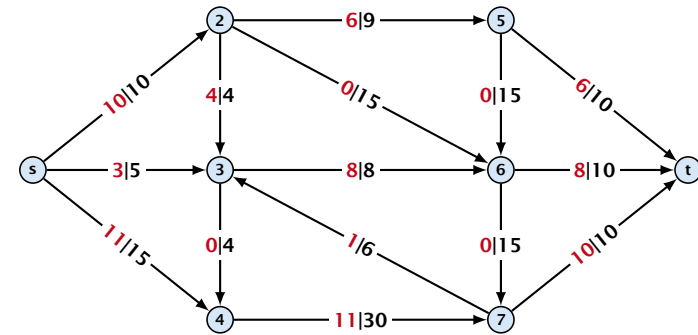
The value of an  $(s, t)$ -flow  $f$  is defined as

$$\text{val}(f) = \sum_{e \in \text{out}(s)} f(e).$$

**Maximum Flow Problem:** Find an  $(s, t)$ -flow with maximum value.

## Flows

### Example 47



The value of the flow is  $\text{val}(f) = 24$ .

## Flows

### Lemma 48 (Flow value lemma)

Let  $f$  be a flow, and let  $A \subseteq V$  be an  $(s, t)$ -cut. Then the net-flow across the cut is equal to the amount of flow leaving  $s$ , i.e.,

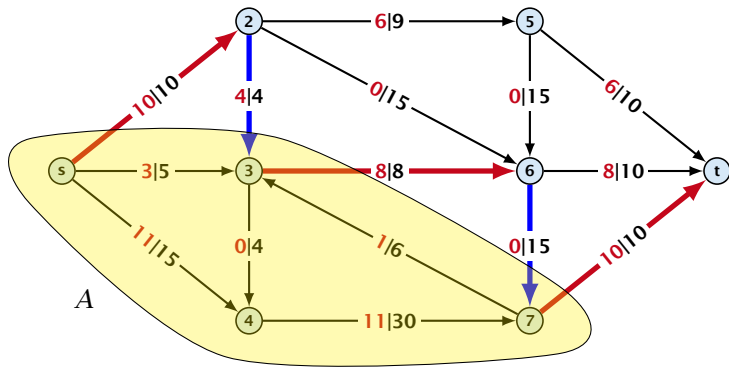
$$\text{val}(f) = \sum_{e \in \text{out}(A)} f(e) - \sum_{e \in \text{into}(A)} f(e).$$

### Proof.

$$\begin{aligned} \text{val}(f) &= \sum_{e \in \text{out}(s)} f(e) \\ &= \sum_{e \in \text{out}(s)} f(e) + \sum_{v \in A \setminus \{s\}} \left( \sum_{e \in \text{out}(v)} f(e) - \sum_{e \in \text{in}(v)} f(e) \right) \\ &= \sum_{e \in \text{out}(A)} f(e) - \sum_{e \in \text{into}(A)} f(e) \end{aligned}$$

The last equality holds since every edge with both end-points in  $A$  contributes negatively as well as positively to the sum in Line 2. The only edges whose contribution doesn't cancel out are edges leaving or entering  $A$ .  $\square$

### Example 49



### Corollary 50

Let  $f$  be an  $(s, t)$ -flow and let  $A$  be an  $(s, t)$ -cut, such that

$$\text{val}(f) = \text{cap}(A, V \setminus A).$$

Then  $f$  is a maximum flow.

#### Proof.

Suppose that there is a flow  $f'$  with larger value. Then

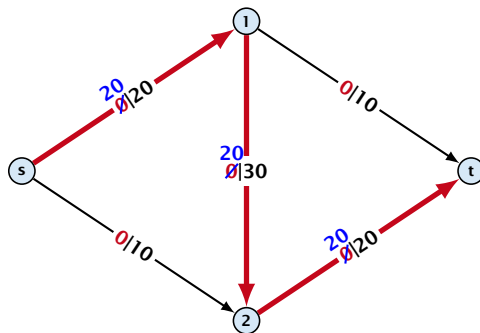
$$\begin{aligned} \text{cap}(A, V \setminus A) &< \text{val}(f') \\ &= \sum_{e \in \text{out}(A)} f'(e) - \sum_{e \in \text{into}(A)} f'(e) \\ &\leq \sum_{e \in \text{out}(A)} f'(e) \\ &\leq \text{cap}(A, V \setminus A) \end{aligned}$$

□

## 11 Augmenting Path Algorithms

### Greedy-algorithm:

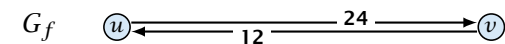
- ▶ start with  $f(e) = 0$  everywhere
- ▶ find an  $s$ - $t$  path with  $f(e) < c(e)$  on every edge
- ▶ augment flow along the path
- ▶ repeat as long as possible



## The Residual Graph

From the graph  $G = (V, E, c)$  and the current flow  $f$  we construct an auxiliary graph  $G_f = (V, E_f, c_f)$  (the residual graph):

- ▶ Suppose the original graph has edges  $e_1 = (u, v)$ , and  $e_2 = (v, u)$  between  $u$  and  $v$ .
- ▶  $G_f$  has edge  $e'_1$  with capacity  $\max\{0, c(e_1) - f(e_1) + f(e_2)\}$  and  $e'_2$  with capacity  $\max\{0, c(e_2) - f(e_2) + f(e_1)\}$ .



## Augmenting Path Algorithm

### Definition 51

An **augmenting path** with respect to flow  $f$ , is a path from  $s$  to  $t$  in the auxiliary graph  $G_f$  that contains only edges with non-zero capacity.

### Algorithm 1 FordFulkerson( $G = (V, E, c)$ )

- 1: Initialize  $f(e) \leftarrow 0$  for all edges.
- 2: **while**  $\exists$  augmenting path  $p$  in  $G_f$  **do**
- 3:     augment as much flow along  $p$  as possible.



## Augmenting Path Algorithm

Animation for augmenting path algorithms is only available in the lecture version of the slides.



## Augmenting Path Algorithm

### Theorem 52

A flow  $f$  is a maximum flow **iff** there are no augmenting paths.

### Theorem 53

The value of a maximum flow is equal to the value of a minimum cut.

### Proof.

Let  $f$  be a flow. The following are equivalent:

1. There exists a cut  $A$  such that  $\text{val}(f) = \text{cap}(A, V \setminus A)$ .
2. Flow  $f$  is a maximum flow.
3. There is no augmenting path w.r.t.  $f$ .

□



## Augmenting Path Algorithm

1.  $\Rightarrow$  2.

This we already showed.

2.  $\Rightarrow$  3.

If there were an augmenting path, we could improve the flow. Contradiction.

3.  $\Rightarrow$  1.

- ▶ Let  $f$  be a flow with no augmenting paths.
- ▶ Let  $A$  be the set of vertices reachable from  $s$  in the residual graph along non-zero capacity edges.
- ▶ Since there is no augmenting path we have  $s \in A$  and  $t \notin A$ .





## Augmenting Path Algorithm

$$\begin{aligned}\text{val}(f) &= \sum_{e \in \text{out}(A)} f(e) - \sum_{e \in \text{into}(A)} f(e) \\ &= \sum_{e \in \text{out}(A)} c(e) \\ &= \text{cap}(A, V \setminus A)\end{aligned}$$

This finishes the proof.

Here the first equality uses the flow value lemma, and the second exploits the fact that the flow along incoming edges must be 0 as the residual graph does not have edges leaving  $A$ .

## Analysis

Assumption:

All capacities are integers between 1 and  $C$ .

Invariant:

Every flow value  $f(e)$  and every residual capacity  $c_f(e)$  remains integral throughout the algorithm.

### Lemma 54

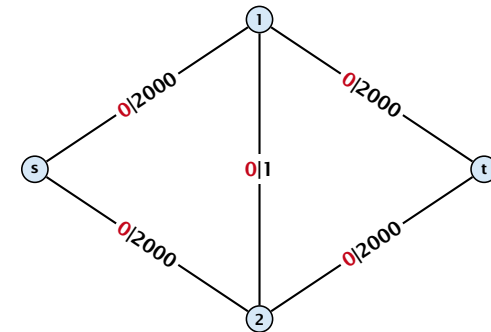
The algorithm terminates in at most  $\text{val}(f^*) \leq nC$  iterations, where  $f^*$  denotes the maximum flow. Each iteration can be implemented in time  $\mathcal{O}(m)$ . This gives a total running time of  $\mathcal{O}(nmC)$ .

### Theorem 55

If all capacities are integers, then there exists a maximum flow for which every flow value  $f(e)$  is integral.

## A Bad Input

Problem: The running time may not be polynomial.

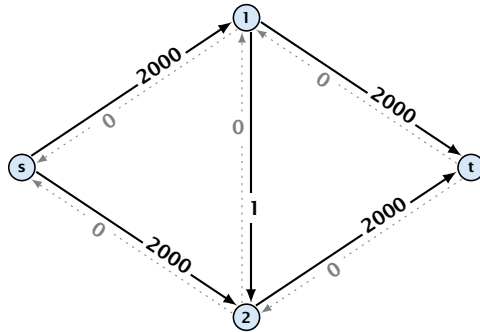


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

## A Bad Input

Problem: The running time may not be polynomial.



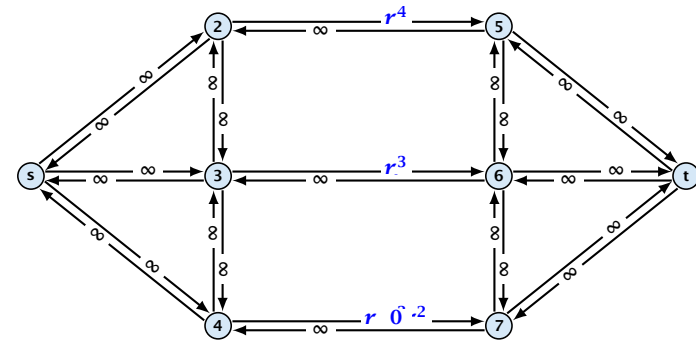
Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

See the lecture-version of the slides for the animation.

## A Pathological Input

Let  $r = \frac{1}{2}(\sqrt{5} - 1)$ . Then  $r^{n+2} = r^n - r^{n+1}$ .



Running time may be infinite!!!

See the lecture-version of the slides for the animation.

### How to choose augmenting paths?

- ▶ We need to find paths efficiently.
- ▶ We want to guarantee a small number of iterations.

### Several possibilities:

- ▶ Choose path with maximum bottleneck capacity.
- ▶ Choose path with sufficiently large bottleneck capacity.
- ▶ Choose the shortest augmenting path.

## Overview: Shortest Augmenting Paths

### Lemma 56

*The length of the shortest augmenting path never decreases.*

### Lemma 57

*After at most  $\mathcal{O}(m)$  augmentations, the length of the shortest augmenting path strictly increases.*

## Overview: Shortest Augmenting Paths

These two lemmas give the following theorem:

### Theorem 58

The shortest augmenting path algorithm performs at most  $\mathcal{O}(mn)$  augmentations. This gives a running time of  $\mathcal{O}(m^2n)$ .

### Proof.

- ▶ We can find the shortest augmenting paths in time  $\mathcal{O}(m)$  via BFS.
- ▶  $\mathcal{O}(m)$  augmentations for paths of exactly  $k < n$  edges.

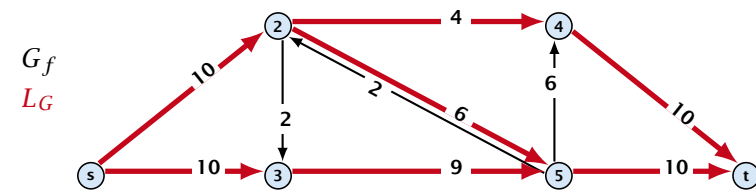
□

## Shortest Augmenting Paths

Define the level  $\ell(v)$  of a node as the length of the shortest  $s$ - $v$  path in  $G_f$ .

Let  $L_G$  denote the subgraph of the residual graph  $G_f$  that contains only those edges  $(u, v)$  with  $\ell(v) = \ell(u) + 1$ .

A path  $P$  is a shortest  $s$ - $t$  path in  $G_f$  if it is an  $s$ - $t$  path in  $L_G$ .



In the following we assume that the residual graph  $G_f$  does not contain zero capacity edges.

This means, we construct it in the usual sense and then delete edges of zero capacity.

## Shortest Augmenting Path

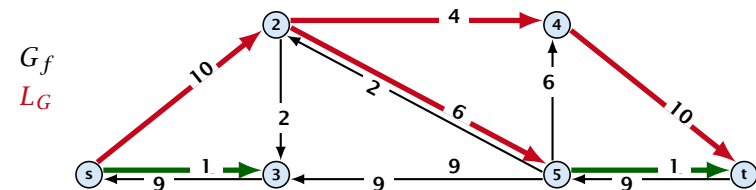
### First Lemma:

The length of the shortest augmenting path never decreases.

After an augmentation  $G_f$  changes as follows:

- ▶ Bottleneck edges on the chosen path are deleted.
- ▶ Back edges are added to all edges that don't have back edges so far.

These changes cannot decrease the distance between  $s$  and  $t$ .



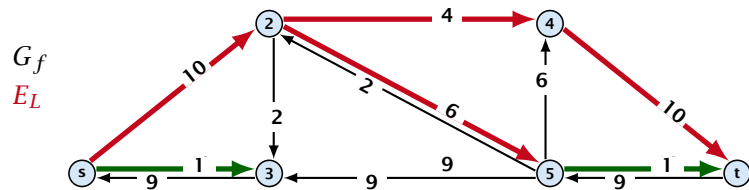
## Shortest Augmenting Path

**Second Lemma:** After at most  $m$  augmentations the length of the shortest augmenting path strictly increases.

Let  $E_L$  denote the set of edges in graph  $L_G$  at the beginning of a round when the distance between  $s$  and  $t$  is  $k$ .

An  $s$ - $t$  path in  $G_f$  that uses edges not in  $E_L$  has length larger than  $k$ , even when considering edges added to  $G_f$  during the round.

In each augmentation one edge is deleted from  $E_L$ .



## Shortest Augmenting Paths

### Theorem 59

The shortest augmenting path algorithm performs at most  $\mathcal{O}(mn)$  augmentations. Each augmentation can be performed in time  $\mathcal{O}(m)$ .

### Theorem 60 (without proof)

There exist networks with  $m = \Theta(n^2)$  that require  $\mathcal{O}(mn)$  augmentations, when we restrict ourselves to only augment along shortest augmenting paths.

### Note:

There always exists a set of  $m$  augmentations that gives a maximum flow (why?).



## Shortest Augmenting Paths

When sticking to shortest augmenting paths we cannot improve (asymptotically) on the number of augmentations.

However, we can improve the running time to  $\mathcal{O}(mn^2)$  by improving the running time for finding an augmenting path (currently we assume  $\mathcal{O}(m)$  per augmentation for this).

## Shortest Augmenting Paths

We maintain a subset  $E_L$  of the edges of  $G_f$  with the guarantee that a shortest  $s$ - $t$  path using only edges from  $E_L$  is a shortest augmenting path.

With each augmentation some edges are deleted from  $E_L$ .

When  $E_L$  does not contain an  $s$ - $t$  path anymore the distance between  $s$  and  $t$  strictly increases.

Note that  $E_L$  is not the set of edges of the level graph but a subset of level-graph edges.



Suppose that the initial distance between  $s$  and  $t$  in  $G_f$  is  $k$ .

$E_L$  is initialized as the level graph  $L_G$ .

Perform a **DFS search** to find a path from  $s$  to  $t$  using edges from  $E_L$ .

Either you find  $t$  after at most  $n$  steps, or you end at a node  $v$  that does not have any outgoing edges.

You can delete incoming edges of  $v$  from  $E_L$ .

Let a phase of the algorithm be defined by the time between two augmentations during which the distance between  $s$  and  $t$  strictly increases.

Initializing  $E_L$  for the phase takes time  $\mathcal{O}(m)$ .

The total cost for searching for augmenting paths during a phase is at most  $\mathcal{O}(mn)$ , since every search (successful (i.e., reaching  $t$ ) or unsuccessful) decreases the number of edges in  $E_L$  and takes time  $\mathcal{O}(n)$ .

The total cost for performing an augmentation **during** a phase is only  $\mathcal{O}(n)$ . For every edge in the augmenting path one has to update the residual graph  $G_f$  and has to check whether the edge is still in  $E_L$  for the next search.

There are at most  $n$  phases. Hence, total cost is  $\mathcal{O}(mn^2)$ .

### How to choose augmenting paths?

- ▶ We need to find paths efficiently.
- ▶ We want to guarantee a small number of iterations.

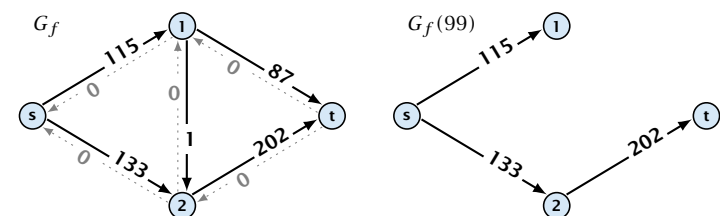
### Several possibilities:

- ▶ Choose path with maximum bottleneck capacity.
- ▶ **Choose path with sufficiently large bottleneck capacity.**
- ▶ Choose the shortest augmenting path.

## Capacity Scaling

### Intuition:

- ▶ Choosing a path with the highest bottleneck increases the flow as much as possible in a single step.
- ▶ Don't worry about finding the exact bottleneck.
- ▶ Maintain scaling parameter  $\Delta$ .
- ▶  $G_f(\Delta)$  is a sub-graph of the residual graph  $G_f$  that contains only edges with capacity at least  $\Delta$ .



## Capacity Scaling

### Algorithm 2 $\text{maxflow}(G, s, t, c)$

```
1: foreach  $e \in E$  do  $f_e \leftarrow 0$ ;  
2:  $\Delta \leftarrow 2^{\lceil \log_2 C \rceil}$   
3: while  $\Delta \geq 1$  do  
4:    $G_f(\Delta) \leftarrow \Delta$ -residual graph  
5:   while there is augmenting path  $P$  in  $G_f(\Delta)$  do  
6:      $f \leftarrow \text{augment}(f, c, P)$   
7:     update( $G_f(\Delta)$ )  
8:    $\Delta \leftarrow \Delta/2$   
9: return  $f$ 
```



## Capacity Scaling

### Assumption:

All capacities are integers between 1 and  $C$ .

### Invariant:

All flows and capacities are/remain integral throughout the algorithm.

### Correctness:

The algorithm computes a maxflow:

- ▶ because of integrality we have  $G_f(1) = G_f$
- ▶ therefore after the last phase there are no augmenting paths anymore
- ▶ this means we have a maximum flow.



## Capacity Scaling

### Lemma 61

There are  $\lceil \log C \rceil + 1$  iterations over  $\Delta$ .

**Proof:** obvious.

### Lemma 62

Let  $f$  be the flow at the end of a  $\Delta$ -phase. Then the maximum flow is smaller than  $\text{val}(f) + m\Delta$ .

**Proof:** less obvious, but simple:

- ▶ There must exist an  $s$ - $t$  cut in  $G_f(\Delta)$  of zero capacity.
- ▶ In  $G_f$  this cut can have capacity at most  $m\Delta$ .
- ▶ This gives me an upper bound on the flow that I can still add.



## Capacity Scaling

### Lemma 63

There are at most  $2m$  augmentations per scaling-phase.

**Proof:**

- ▶ Let  $f$  be the flow at the end of the previous phase.
- ▶  $\text{val}(f^*) \leq \text{val}(f) + 2m\Delta$
- ▶ Each augmentation increases flow by  $\Delta$ .

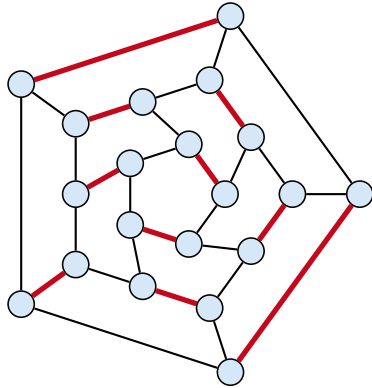
### Theorem 64

We need  $\mathcal{O}(m \log C)$  augmentations. The algorithm can be implemented in time  $\mathcal{O}(m^2 \log C)$ .



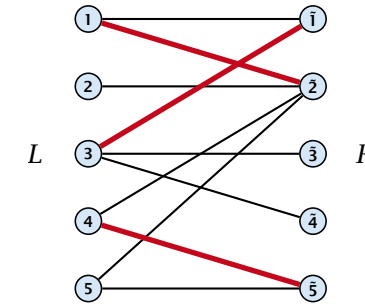
## Matching

- ▶ Input: undirected graph  $G = (V, E)$ .
- ▶  $M \subseteq E$  is a **matching** if each node appears in at most one edge in  $M$ .
- ▶ Maximum Matching: find a matching of maximum cardinality



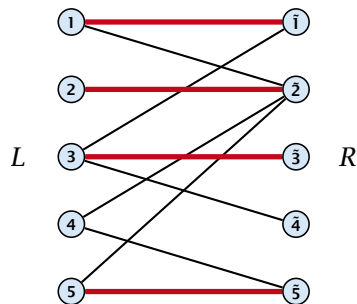
## Bipartite Matching

- ▶ Input: undirected, **bipartite** graph  $G = (L \uplus R, E)$ .
- ▶  $M \subseteq E$  is a **matching** if each node appears in at most one edge in  $M$ .
- ▶ Maximum Matching: find a matching of maximum cardinality



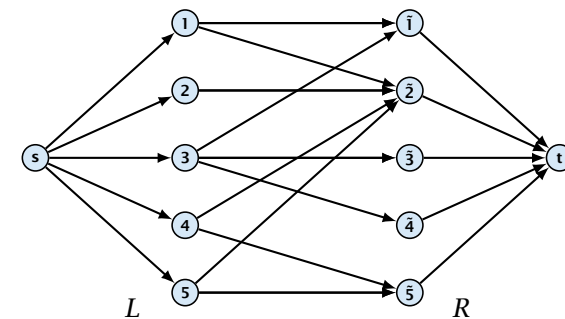
## Bipartite Matching

- ▶ Input: undirected, **bipartite** graph  $G = (L \uplus R, E)$ .
- ▶  $M \subseteq E$  is a **matching** if each node appears in at most one edge in  $M$ .
- ▶ Maximum Matching: find a matching of maximum cardinality



## Maxflow Formulation

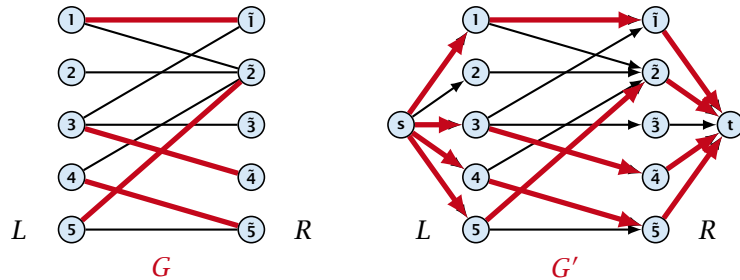
- ▶ Input: undirected, bipartite graph  $G = (L \uplus R \uplus \{s, t\}, E')$ .
- ▶ Direct all edges from  $L$  to  $R$ .
- ▶ Add source  $s$  and connect it to all nodes on the left.
- ▶ Add  $t$  and connect all nodes on the right to  $t$ .
- ▶ All edges have unit capacity.



## Proof

### Max cardinality matching in $G \leq$ value of maxflow in $G'$

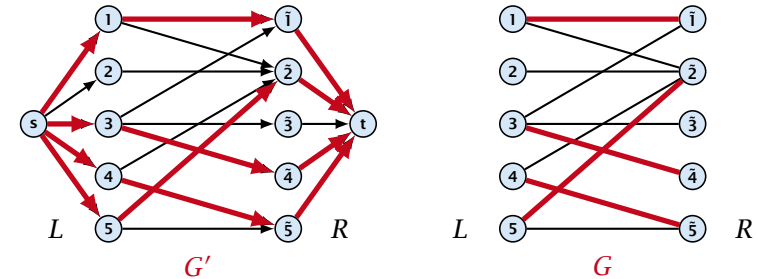
- ▶ Given a maximum matching  $M$  of cardinality  $k$ .
- ▶ Consider flow  $f$  that sends one unit along each of  $k$  paths.
- ▶  $f$  is a flow and has cardinality  $k$ .



## Proof

### Max cardinality matching in $G \geq$ value of maxflow in $G'$

- ▶ Let  $f$  be a maxflow in  $G'$  of value  $k$
- ▶ Integrality theorem  $\Rightarrow k$  integral; we can assume  $f$  is 0/1.
- ▶ Consider  $M =$  set of edges from  $L$  to  $R$  with  $f(e) = 1$ .
- ▶ Each node in  $L$  and  $R$  participates in at most one edge in  $M$ .
- ▶  $|M| = k$ , as the flow must use at least  $k$  middle edges.



## 12.1 Matching

### Which flow algorithm to use?

- ▶ Generic augmenting path:  $\mathcal{O}(m \cdot \text{val}(f^*)) = \mathcal{O}(mn)$ .
- ▶ Capacity scaling:  $\mathcal{O}(m^2 \log C) = \mathcal{O}(m^2)$ .
- ▶ Shortest augmenting path:  $\mathcal{O}(mn^2)$ .

For **unit capacity simple graphs** shortest augmenting path can be implemented in time  $\mathcal{O}(m\sqrt{n})$ .

A graph is a **unit capacity simple graph** if

- ▶ every edge has capacity 1
- ▶ a node has either at most one leaving edge **or** at most one entering edge

## Baseball Elimination

| team<br>$i$  | wins<br>$w_i$ | losses<br>$\ell_i$ | remaining games |     |    |     |
|--------------|---------------|--------------------|-----------------|-----|----|-----|
|              |               |                    | Atl             | Phi | NY | Mon |
| Atlanta      | 83            | 71                 | –               | 1   | 6  | 1   |
| Philadelphia | 80            | 79                 | 1               | –   | 0  | 2   |
| New York     | 78            | 78                 | 6               | 0   | –  | 0   |
| Montreal     | 77            | 82                 | 1               | 2   | 0  | –   |

### Which team can end the season with most wins?

- ▶ Montreal is eliminated, since even after winning all remaining games there are only 80 wins.
- ▶ But also Philadelphia is eliminated. Why?



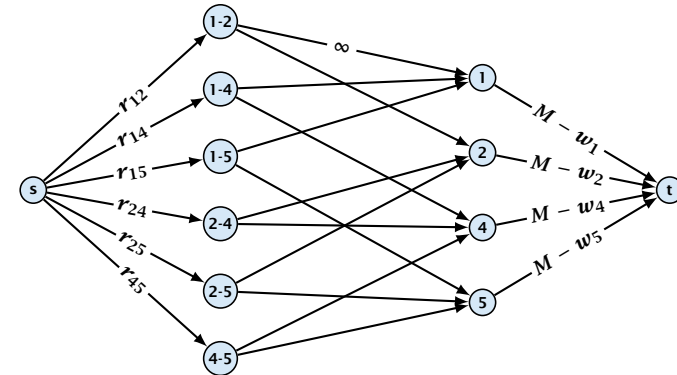
## Baseball Elimination

### Formal definition of the problem:

- ▶ Given a set  $S$  of teams, and one specific team  $z \in S$ .
- ▶ Team  $x$  has already won  $w_x$  games.
- ▶ Team  $x$  still has to play team  $y$ ,  $r_{xy}$  times.
- ▶ Does team  $z$  still have a chance to finish with the most number of wins.

## Baseball Elimination

Flow network for  $z = 3$ .  $M$  is number of wins Team 3 can still obtain.



**Idea.** Distribute the results of remaining games in such a way that no team gets too many wins.

## Certificate of Elimination

Let  $T \subseteq S$  be a subset of teams. Define

$$w(T) := \sum_{i \in T} w_i, \quad r(T) := \sum_{i, j \in T, i < j} r_{ij}$$

wins of teams in  $T$ 
remaining games among teams in  $T$

If  $\frac{w(T) + r(T)}{|T|} > M$  then one of the teams in  $T$  will have more than  $M$  wins in the end. A team that can win at most  $M$  games is therefore eliminated.

### Theorem 65

A team  $z$  is eliminated if and only if the flow network for  $z$  does not allow a flow of value  $\sum_{i, j \in S \setminus \{z\}, i < j} r_{ij}$ .

### Proof ( $\Leftarrow$ )

- ▶ Consider the mincut  $A$  in the flow network. Let  $T$  be the set of team-nodes in  $A$ .
- ▶ If for node  $x-y$  not both team-nodes  $x$  and  $y$  are in  $T$ , then  $x-y \notin A$  as otw. the cut would cut an infinite capacity edge.
- ▶ We don't find a flow that saturates all source edges:

$$\begin{aligned}
 r(S \setminus \{z\}) &> \text{cap}(A, V \setminus A) \\
 &\geq \sum_{i < j: i \notin T \vee j \notin T} r_{ij} + \sum_{i \in T} (M - w_i) \\
 &\geq r(S \setminus \{z\}) - r(T) + |T|M - w(T)
 \end{aligned}$$

- ▶ This gives  $M < (w(T) + r(T))/|T|$ , i.e.,  $z$  is eliminated.

## Baseball Elimination

### Proof ( $\Rightarrow$ )

- ▶ Suppose we have a flow that saturates all source edges.
- ▶ We can assume that this flow is **integral**.
- ▶ For every pairing  $x$ - $y$  it defines how many games team  $x$  and team  $y$  should win.
- ▶ The flow leaving the team-node  $x$  can be interpreted as the additional number of wins that team  $x$  will obtain.
- ▶ This is less than  $M - w_x$  because of capacity constraints.
- ▶ Hence, we found a set of results for the remaining games, such that no team obtains more than  $M$  wins in total.
- ▶ Hence, team  $z$  is not eliminated.

## Project Selection

### Project selection problem:

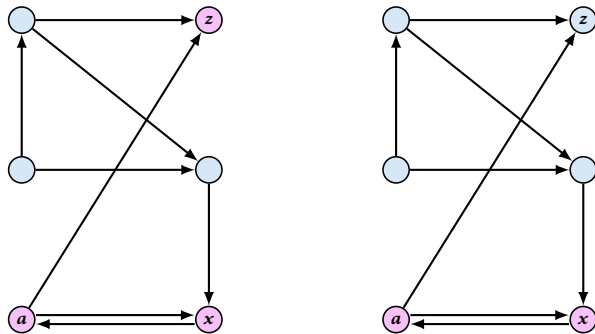
- ▶ Set  $P$  of possible projects. Project  $v$  has an associated profit  $p_v$  (can be positive or negative).
- ▶ Some projects have requirements (taking course EA2 requires course EA1).
- ▶ Dependencies are modelled in a graph. Edge  $(u, v)$  means “can’t do project  $u$  without also doing project  $v$ .”
- ▶ A subset  $A$  of projects is **feasible** if the prerequisites of every project in  $A$  also belong to  $A$ .

**Goal:** Find a feasible set of projects that maximizes the profit.

## Project Selection

### The prerequisite graph:

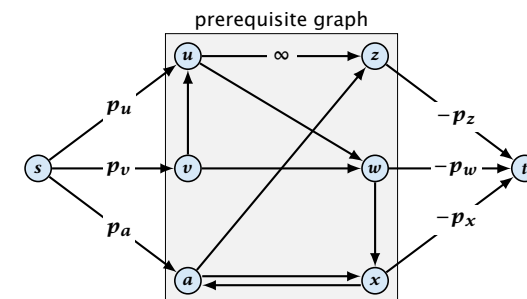
- ▶  $\{x, a, z\}$  is a feasible subset.
- ▶  $\{x, a\}$  is infeasible.



## Project Selection

### Mincut formulation:

- ▶ Edges in the prerequisite graph get infinite capacity.
- ▶ Add edge  $(s, v)$  with capacity  $p_v$  for nodes  $v$  with positive profit.
- ▶ Create edge  $(v, t)$  with capacity  $-p_v$  for nodes  $v$  with negative profit.



### Theorem 66

$A$  is a mincut if  $A \setminus \{s\}$  is the optimal set of projects.

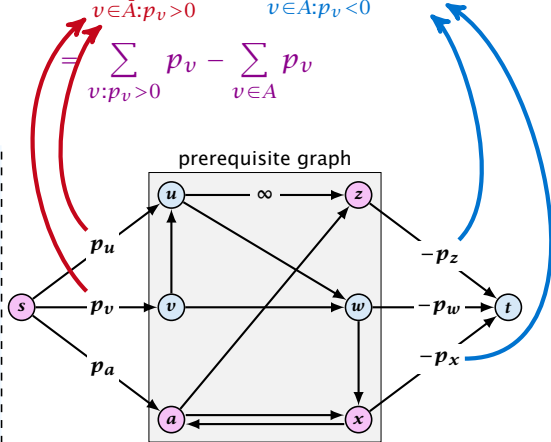
#### Proof.

▶  $A$  is feasible because of capacity infinity edges.

$$\text{cap}(A, V \setminus A) = \sum_{v \in A: p_v > 0} p_v + \sum_{v \in A: p_v < 0} (-p_v)$$

$$= \sum_{v: p_v > 0} p_v - \sum_{v \in A} p_v$$

For the formula we define  $p_s := 0$ .  
 The step follows by adding  $\sum_{v \in A: p_v > 0} p_v - \sum_{v \in A: p_v > 0} p_v = 0$ .  
 Note that minimizing the capacity of the cut  $(A, V \setminus A)$  corresponds to maximizing profits of projects in  $A$ .



### Preflows

#### Definition 67

An  $(s, t)$ -preflow is a function  $f: E \rightarrow \mathbb{R}^+$  that satisfies

1. For each edge  $e$

$$0 \leq f(e) \leq c(e).$$

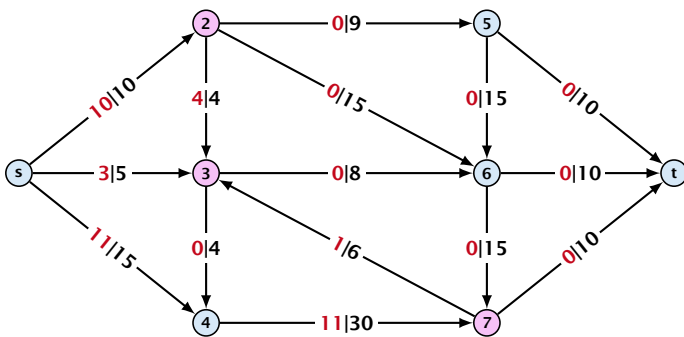
(capacity constraints)

2. For each  $v \in V \setminus \{s, t\}$

$$\sum_{e \in \text{out}(v)} f(e) \leq \sum_{e \in \text{into}(v)} f(e).$$

### Preflows

#### Example 68



A node that has  $\sum_{e \in \text{out}(v)} f(e) < \sum_{e \in \text{into}(v)} f(e)$  is called an **active node**.

### Preflows

#### Definition:

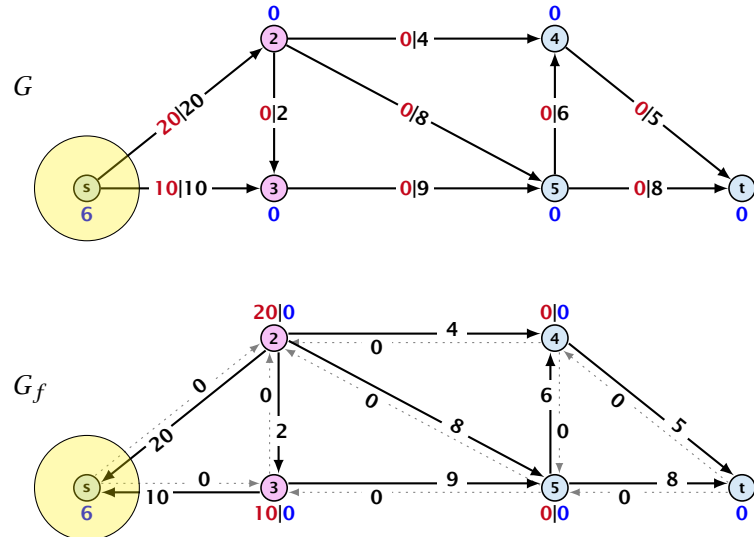
A **labelling** is a function  $\ell: V \rightarrow \mathbb{N}$ . It is **valid** for preflow  $f$  if

- ▶  $\ell(u) \leq \ell(v) + 1$  for all edges  $(u, v)$  in the residual graph  $G_f$  (only non-zero capacity edges!!!)
- ▶  $\ell(s) = n$
- ▶  $\ell(t) = 0$

#### Intuition:

The labelling can be viewed as a height function. Whenever the height from node  $u$  to node  $v$  decreases by more than 1 (i.e., it goes very steep downhill from  $u$  to  $v$ ), the corresponding edge must be saturated.

## Preflows



## Preflows

### Lemma 69

A *preflow* that has a valid labelling saturates a cut.

**Proof:**

- ▶ There are  $n$  nodes but  $n + 1$  different labels from  $0, \dots, n$ .
- ▶ There must exist a label  $d \in \{0, \dots, n\}$  such that none of the nodes carries this label.
- ▶ Let  $A = \{v \in V \mid \ell(v) > d\}$  and  $B = \{v \in V \mid \ell(v) < d\}$ .
- ▶ We have  $s \in A$  and  $t \in B$  and there is no edge from  $A$  to  $B$  in the residual graph  $G_f$ ; this means that  $(A, B)$  is a saturated cut.

### Lemma 70

A *flow* that has a valid labelling is a maximum flow.

## Push Relabel Algorithms

**Idea:**

- ▶ start with some preflow and some valid labelling
- ▶ successively change the preflow while maintaining a valid labelling
- ▶ stop when you have a flow (i.e., no more active nodes)

Note that this is somewhat dual to an augmenting path algorithm. The former maintains the property that it has a feasible flow. It successively changes this flow until it saturates some cut in which case we conclude that the flow is maximum. A preflow push algorithm maintains the property that it has a saturated cut. The preflow is changed iteratively until it fulfills conservation constraints in which case we can conclude that we have a maximum flow.

## Changing a Preflow

An arc  $(u, v)$  with  $c_f(u, v) > 0$  in the residual graph is **admissible** if  $\ell(u) = \ell(v) + 1$  (i.e., it goes downwards w.r.t. labelling  $\ell$ ).

### The push operation

Consider an active node  $u$  with **excess flow**

$f(u) = \sum_{e \in \text{into}(u)} f(e) - \sum_{e \in \text{out}(u)} f(e)$  and suppose  $e = (u, v)$  is an admissible arc with residual capacity  $c_f(e)$ .

We can send flow  $\min\{c_f(e), f(u)\}$  along  $e$  and obtain a new preflow. The old labelling is still valid (!!!).

- ▶ **saturating push:**  $\min\{f(u), c_f(e)\} = c_f(e)$   
the arc  $e$  is deleted from the residual graph
- ▶ **non-saturating push:**  $\min\{f(u), c_f(e)\} = f(u)$   
the node  $u$  becomes inactive

Note that a push-operation may be saturating **and** non-saturating at the same time.

## Push Relabel Algorithms

### The relabel operation

Consider an active node  $u$  that does not have an outgoing admissible arc.

Increasing the label of  $u$  by 1 results in a valid labelling.

- ▶ Edges  $(w, u)$  incoming to  $u$  still fulfill their constraint  $\ell(w) \leq \ell(u) + 1$ .
- ▶ An outgoing edge  $(u, w)$  had  $\ell(u) < \ell(w) + 1$  before since it was not admissible. Now:  $\ell(u) \leq \ell(w) + 1$ .

## Push Relabel Algorithms

### Intuition:

We want to send flow downwards, since the source has a height/label of  $n$  and the target a height/label of  $0$ . If we see an active node  $u$  with an admissible arc we push the flow at  $u$  towards the other end-point that has a lower height/label. If we do not have an admissible arc but excess flow into  $u$  it should roughly mean that the level/height/label of  $u$  should rise. (If we consider the flow to be water then this would be natural.)

Note that the above intuition is very incorrect as the labels are integral, i.e., they cannot really be seen as the height of a node.

## Reminder

- ▶ In a **preflow** nodes may not fulfill conservation constraints; a node may have more incoming flow than outgoing flow.
- ▶ Such a node is called **active**.
- ▶ A labelling is **valid** if for every edge  $(u, v)$  in the residual graph  $\ell(u) \leq \ell(v) + 1$ .
- ▶ An arc  $(u, v)$  in residual graph is **admissible** if  $\ell(u) = \ell(v) + 1$ .
- ▶ A **saturating push** along  $e$  pushes an amount of  $c(e)$  flow along the edge, thereby saturating the edge (and making it disappear from the residual graph).
- ▶ A **non-saturating push** along  $e = (u, v)$  pushes a flow of  $f(u)$ , where  $f(u)$  is the **excess flow** of  $u$ . This makes  $u$  inactive.

## Push Relabel Algorithms

```
Algorithm 3 maxflow( $G, s, t, c$ )
1: find initial preflow  $f$ 
2: while there is active node  $u$  do
3:   if there is admiss. arc  $e$  out of  $u$  then
4:     push( $G, e, f, c$ )
5:   else
6:     relabel( $u$ )
7: return  $f$ 
```

In the following example we always stick to the same active node  $u$  until it becomes inactive but this is not required.

## Preflow Push Algorithm

Animation for push relabel algorithms is only available in the lecture version of the slides.

## Analysis

Note that the lemma is almost trivial. A node  $v$  having excess flow means that the current preflow ships something to  $v$ . The residual graph allows to *undo* flow. Therefore, there must exist a path that can undo the shipment and move it back to  $s$ . However, a formal proof is required.

### Lemma 71

An active node has a path to  $s$  in the residual graph.

#### Proof.

- ▶ Let  $A$  denote the set of nodes that can reach  $s$ , and let  $B$  denote the remaining nodes. Note that  $s \in A$ .
- ▶ In the following we show that a node  $b \in B$  has excess flow  $f(b) = 0$  which gives the lemma.
- ▶ In the residual graph there are no edges into  $A$ , and, hence, no edges leaving  $A$ /entering  $B$  can carry any flow.
- ▶ Let  $f(B) = \sum_{v \in B} f(v)$  be the excess flow of all nodes in  $B$ .

Let  $f : E \rightarrow \mathbb{R}_0^+$  be a preflow. We introduce the notation

$$f(x, y) = \begin{cases} 0 & (x, y) \notin E \\ f((x, y)) & (x, y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b \in B} f(b) \\ &= \sum_{b \in B} \left( \sum_{v \in V} f(v, b) - \sum_{v \in V} f(b, v) \right) \\ &= \sum_{b \in B} \left( \sum_{v \in A} f(v, b) + \sum_{v \in B} f(v, b) - \sum_{v \in A} f(b, v) - \sum_{v \in B} f(b, v) \right) \\ &= - \sum_{b \in B} \sum_{v \in A} f(b, v) \\ &\leq 0 \end{aligned}$$

Hence, the excess flow  $f(b)$  must be 0 for every node  $b \in B$ .

## Analysis

### Lemma 72

The label of a node cannot become larger than  $2n - 1$ .

#### Proof.

- ▶ When increasing the label at a node  $u$  there exists a path from  $u$  to  $s$  of length at most  $n - 1$ . Along each edge of the path the height/label can at most drop by 1, and the label of the source is  $n$ .

### Lemma 73

There are only  $O(n^2)$  relabel operations.

## Analysis

### Lemma 74

The number of *saturating pushes* performed is at most  $\mathcal{O}(mn)$ .

#### Proof.

- ▶ Suppose that we just made a saturating push along  $(u, v)$ .
- ▶ Hence, the edge  $(u, v)$  is deleted from the residual graph.
- ▶ For the edge to appear again, a push from  $v$  to  $u$  is required.
- ▶ Currently,  $\ell(u) = \ell(v) + 1$ , as we only make pushes along admissible edges.
- ▶ For a push from  $v$  to  $u$  the edge  $(v, u)$  must become admissible. The label of  $v$  must increase by at least 2.
- ▶ Since the label of  $v$  is at most  $2n - 1$ , there are at most  $n$  pushes along  $(u, v)$ .

### Lemma 75

The number of *non-saturating pushes* performed is at most  $\mathcal{O}(n^2m)$ .

#### Proof.

- ▶ Define a potential function  $\Phi(f) = \sum_{\text{active nodes } v} \ell(v)$
- ▶ A saturating push increases  $\Phi$  by  $\leq 2n$  (when the target node becomes active it may contribute at most  $2n$  to the sum).
- ▶ A relabel increases  $\Phi$  by at most 1.
- ▶ A non-saturating push decreases  $\Phi$  by at least 1 as the node that is pushed from becomes inactive and has a label that is strictly larger than the target.
- ▶ Hence,  
$$\# \text{non-saturating\_pushes} \leq \# \text{relabels} + 2n \cdot \# \text{saturating\_pushes} \leq \mathcal{O}(n^2m)$$

## Analysis

### Theorem 76

There is an implementation of the generic push relabel algorithm with running time  $\mathcal{O}(n^2m)$ .

## Analysis

#### Proof:

For every node maintain a list of admissible edges starting at that node. Further maintain a list of active nodes.

A push along an edge  $(u, v)$  can be performed in constant time

- ▶ check whether edge  $(v, u)$  needs to be added to  $G_f$
- ▶ check whether  $(u, v)$  needs to be deleted (saturating push)
- ▶ check whether  $u$  becomes inactive and has to be deleted from the set of active nodes

A relabel at a node  $u$  can be performed in time  $\mathcal{O}(n)$

- ▶ check for all outgoing edges if they become admissible
- ▶ check for all incoming edges if they become non-admissible

## Analysis

For special variants of push relabel algorithms we organize the neighbours of a node into a linked list (possible neighbours in the residual graph  $G_f$ ). Then we use the discharge-operation:

### Algorithm 4 discharge( $u$ )

```
1: while  $u$  is active do
2:    $v \leftarrow u.current-neighbour$ 
3:   if  $v = \text{null}$  then
4:     relabel( $u$ )
5:      $u.current-neighbour \leftarrow u.neighbour-list-head$ 
6:   else
7:     if  $(u, v)$  admissible then push( $u, v$ )
8:     else  $u.current-neighbour \leftarrow v.next-in-list$ 
```

Note that  $u.current-neighbour$  is a global variable. It is only changed within the discharge routine, but keeps its value between consecutive calls to discharge.

### Lemma 77

If  $v = \text{null}$  in Line 3, then there is no outgoing admissible edge from  $u$ .

### Proof.

- ▶ While pushing from  $u$  the current-neighbour pointer is only advanced if the current edge is not admissible.
- ▶ The only thing that could make the edge admissible again would be a relabel at  $u$ .
- ▶ If we reach the end of the list ( $v = \text{null}$ ) all edges are not admissible.  $\square$

This shows that discharge( $u$ ) is correct, and that we can perform a relabel in Line 4.

In order for  $e$  to become admissible the other end-point say  $v$  has to push flow to  $u$  (so that the edge  $(u, v)$  re-appears in the residual graph). For this the label of  $v$  needs to be larger than the label of  $u$ . Then in order to make  $(u, v)$  admissible the label of  $u$  has to increase.

## 13.2 Relabel to Front

### Algorithm 21 relabel-to-front( $G, s, t$ )

```
1: initialize preflow
2: initialize node list  $L$  containing  $V \setminus \{s, t\}$  in any order
3: foreach  $u \in V \setminus \{s, t\}$  do
4:    $u.current-neighbour \leftarrow u.neighbour-list-head$ 
5:  $u \leftarrow L.head$ 
6: while  $u \neq \text{null}$  do
7:    $old-height \leftarrow \ell(u)$ 
8:   discharge( $u$ )
9:   if  $\ell(u) > old-height$  then // relabel happened
10:     move  $u$  to the front of  $L$ 
11:    $u \leftarrow u.next$ 
```

## 13.2 Relabel to Front

### Lemma 78 (Invariant)

In Line 6 of the relabel-to-front algorithm the following invariant holds.

1. The sequence  $L$  is topologically sorted w.r.t. the set of admissible edges; this means for an admissible edge  $(x, y)$  the node  $x$  appears before  $y$  in sequence  $L$ .
2. No node before  $u$  in the list  $L$  is active.



### Proof:

#### ► Initialization:

1. In the beginning  $s$  has label  $n \geq 2$ , and all other nodes have label 0. Hence, no edge is admissible, which means that any ordering  $L$  is permitted.
2. We start with  $u$  being the head of the list; hence no node before  $u$  can be active

#### ► Maintenance:

1.
  - Pushes do not create any new admissible edges. Therefore, if  $\text{discharge}()$  does not relabel  $u$ ,  $L$  is still topologically sorted.
  - After relabeling,  $u$  cannot have admissible incoming edges as such an edge  $(x, u)$  would have had a difference  $\ell(x) - \ell(u) \geq 2$  before the re-labeling (such edges do not exist in the residual graph). Hence, moving  $u$  to the front does not violate the sorting property for any edge; however it fixes this property for all admissible edges leaving  $u$  that were generated by the relabeling.

## 13.2 Relabel to Front

### Proof:

#### ► Maintenance:

2. If we do a relabel there is nothing to prove because the only node before  $u'$  ( $u$  in the next iteration) will be the current  $u$ ; the  $\text{discharge}(u)$  operation only terminates when  $u$  is not active anymore.

For the case that we do not relabel, observe that the only way a predecessor could be active is that we push flow to it via an admissible arc. However, all admissible arcs point to successors of  $u$ .

Note that the invariant means that for  $u = \text{null}$  we have a preflow with a valid labelling that does not have active nodes. This means we have a maximum flow.



## 13.2 Relabel to Front

### Lemma 79

There are at most  $\mathcal{O}(n^3)$  calls to  $\text{discharge}(u)$ .

Every discharge operation without a relabel advances  $u$  (the current node within list  $L$ ). Hence, if we have  $n$  discharge operations without a relabel we have  $u = \text{null}$  and the algorithm terminates.

Therefore, the number of calls to discharge is at most  $n(\# \text{relabels} + 1) = \mathcal{O}(n^3)$ .

## 13.2 Relabel to Front

### Lemma 80

The cost for all relabel-operations is only  $\mathcal{O}(n^2)$ .

A relabel-operation at a node is constant time (increasing the label and resetting  $u.\text{current-neighbour}$ ). In total we have  $\mathcal{O}(n^2)$  relabel-operations.



## 13.2 Relabel to Front

Note that by definition a saturating push operation ( $\min\{c_f(e), f(u)\} = c_f(e)$ ) can at the same time be a non-saturating push operation ( $\min\{c_f(e), f(u)\} = f(u)$ ).

### Lemma 81

The cost for all saturating push-operations that are **not** also non-saturating push-operations is only  $\mathcal{O}(mn)$ .

Note that such a push-operation leaves the node  $u$  active but makes the edge  $e$  disappear from the residual graph. Therefore the push-operation is immediately followed by an increase of the pointer  $u.current-neighbour$ .

This pointer can traverse the neighbour-list at most  $\mathcal{O}(n)$  times (upper bound on number of relabels) and the neighbour-list has only  $degree(u) + 1$  many entries (+1 for null-entry).

## 13.2 Relabel to Front

### Lemma 82

The cost for all non-saturating push-operations is only  $\mathcal{O}(n^3)$ .

A non-saturating push-operation takes constant time and ends the current call to `discharge()`. Hence, there are only  $\mathcal{O}(n^3)$  such operations.

### Theorem 83

The push-relabel algorithm with the rule *relabel-to-front* takes time  $\mathcal{O}(n^3)$ .

## 13.3 Highest Label

### Algorithm 6 highest-label( $G, s, t$ )

```
1: initialize preflow
2: foreach  $u \in V \setminus \{s, t\}$  do
3:    $u.current-neighbour \leftarrow u.neighbour-list-head$ 
4: while  $\exists$  active node  $u$  do
5:   select active node  $u$  with highest label
6:    $discharge(u)$ 
```

## 13.3 Highest Label

### Lemma 84

When using highest label the number of non-saturating pushes is only  $\mathcal{O}(n^3)$ .

A push from a node on level  $\ell$  can only “activate” nodes on levels strictly less than  $\ell$ .

This means, after a non-saturating push from  $u$  a relabel is required to make  $u$  active again.

Hence, after  $n$  non-saturating pushes without an intermediate relabel there are no active nodes left.

Therefore, the number of non-saturating pushes is at most  $n(\#relabels + 1) = \mathcal{O}(n^3)$ .

## 13.3 Highest Label

Since a discharge-operation is terminated by a non-saturating push this gives an upper bound of  $\mathcal{O}(n^3)$  on the number of discharge-operations.

The cost for relabels and saturating pushes can be estimated in exactly the same way as in the case of the generic push-relabel algorithm.

### Question:

How do we find the next node for a discharge operation?

## 13.3 Highest Label

Maintain lists  $L_i$ ,  $i \in \{0, \dots, 2n\}$ , where list  $L_i$  contains active nodes with label  $i$  (maintaining these lists induces only constant additional cost for every push-operation and for every relabel-operation).

After a discharge operation terminated for a node  $u$  with label  $k$ , traverse the lists  $L_k, L_{k-1}, \dots, L_0$ , (in that order) until you find a non-empty list.

Unless the last (non-saturating) push was to  $s$  or  $t$  the list  $k-1$  must be non-empty (i.e., the search takes constant time).

## 13.3 Highest Label

Hence, the total time required for searching for active nodes is at most

$$\mathcal{O}(n^3) + n(\#non-saturating-pushes-to-s-or-t)$$

### Lemma 85

The number of non-saturating pushes to  $s$  or  $t$  is at most  $\mathcal{O}(n^2)$ .

With this lemma we get

### Theorem 86

The push-relabel algorithm with the rule highest-label takes time  $\mathcal{O}(n^3)$ .

## 13.3 Highest Label

### Proof of the Lemma.

- ▶ We only show that the number of pushes to the source is at most  $\mathcal{O}(n^2)$ . A similar argument holds for the target.
- ▶ After a node  $v$  (which must have  $\ell(v) = n+1$ ) made a non-saturating push to the source there needs to be another node whose label is increased from  $\leq n+1$  to  $n+2$  before  $v$  can become active again.
- ▶ This happens for every push that  $v$  makes to the source. Since, every node can pass the threshold  $n+2$  at most once,  $v$  can make at most  $n$  pushes to the source.
- ▶ As this holds for every node the total number of pushes to the source is at most  $\mathcal{O}(n^2)$ .

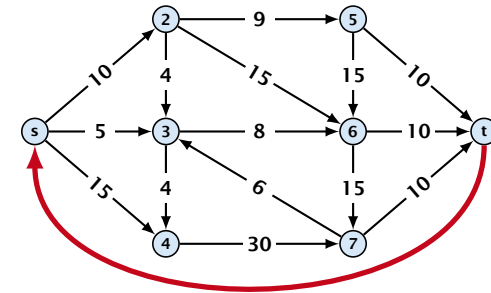
## Mincost Flow

### Problem Definition:

$$\begin{aligned} \min \quad & \sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: 0 \leq f(e) \leq u(e) \\ & \forall v \in V: f(v) = b(v) \end{aligned}$$

- ▶  $G = (V, E)$  is a **directed graph**.
- ▶  $u: E \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$  is the **capacity function**.
- ▶  $c: E \rightarrow \mathbb{R}$  is the **cost function** (note that  $c(e)$  may be negative).
- ▶  $b: V \rightarrow \mathbb{R}$ ,  $\sum_{v \in V} b(v) = 0$  is a **demand function**.

## Solve Maxflow Using Mincost Flow



- ▶ Given a flow network for a standard maxflow problem.
- ▶ Set  $b(v) = 0$  for every node. Keep the capacity function  $u$  for all edges. Set the cost  $c(e)$  for every edge to 0.
- ▶ Add an edge from  $t$  to  $s$  with infinite capacity and cost  $-1$ .
- ▶ Then,  $\text{val}(f^*) = -\text{cost}(f_{\min})$ , where  $f^*$  is a maxflow, and  $f_{\min}$  is a mincost-flow.

## Solve Maxflow Using Mincost Flow

### Solve decision version of maxflow:

- ▶ Given a flow network for a standard maxflow problem, and a value  $k$ .
- ▶ Set  $b(v) = 0$  for every node apart from  $s$  or  $t$ . Set  $b(s) = -k$  and  $b(t) = k$ .
- ▶ Set edge-costs to zero, and keep the capacities.
- ▶ There exists a maxflow of value at least  $k$  if and only if the mincost-flow problem is feasible.

## Generalization

### Our model:

$$\begin{aligned} \min \quad & \sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: 0 \leq f(e) \leq u(e) \\ & \forall v \in V: f(v) = b(v) \end{aligned}$$

where  $b: V \rightarrow \mathbb{R}$ ,  $\sum_v b(v) = 0$ ;  $u: E \rightarrow \mathbb{R}_0^+ \cup \{\infty\}$ ;  $c: E \rightarrow \mathbb{R}$ ;

### A more general model?

$$\begin{aligned} \min \quad & \sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \ell(e) \leq f(e) \leq u(e) \\ & \forall v \in V: a(v) \leq f(v) \leq b(v) \end{aligned}$$

where  $a: V \rightarrow \mathbb{R}$ ,  $b: V \rightarrow \mathbb{R}$ ;  $\ell: E \rightarrow \mathbb{R} \cup \{-\infty\}$ ,  $u: E \rightarrow \mathbb{R} \cup \{\infty\}$   
 $c: E \rightarrow \mathbb{R}$ ;

## Generalization

### Differences

- ▶ Flow along an edge  $e$  may have non-zero lower bound  $l(e)$ .
- ▶ Flow along  $e$  may have negative upper bound  $u(e)$ .
- ▶ The demand at a node  $v$  may have lower bound  $a(v)$  and upper bound  $b(v)$  instead of just lower bound = upper bound =  $b(v)$ .

## Reduction I

$$\begin{aligned} \min \quad & \sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: l(e) \leq f(e) \leq u(e) \\ & \forall v \in V: a(v) \leq f(v) \leq b(v) \end{aligned}$$

We can assume that  $a(v) = b(v)$ :

Add new node  $r$ .

Add edge  $(r, v)$  for all  $v \in V$ .

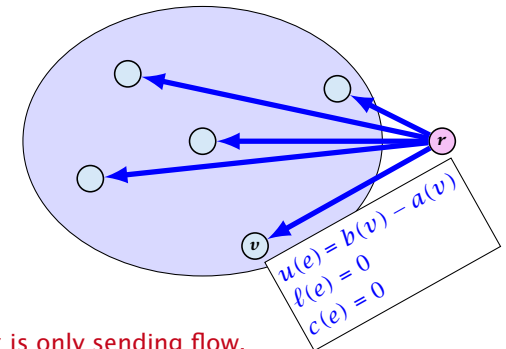
Set  $l(e) = c(e) = 0$  for these edges.

Set  $u(e) = b(v) - a(v)$  for edge  $(r, v)$ .

Set  $a(v) = b(v)$  for all  $v \in V$ .

Set  $b(r) = -\sum_{v \in V} b(v)$ .

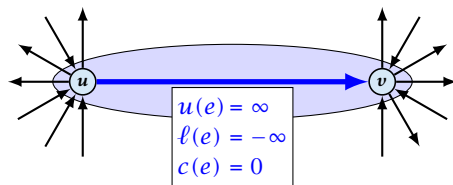
$-\sum_v b(v)$  is negative; hence  $r$  is only sending flow.



## Reduction II

$$\begin{aligned} \min \quad & \sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: l(e) \leq f(e) \leq u(e) \\ & \forall v \in V: f(v) = b(v) \end{aligned}$$

We can assume that either  $l(e) \neq -\infty$  or  $u(e) \neq \infty$ :

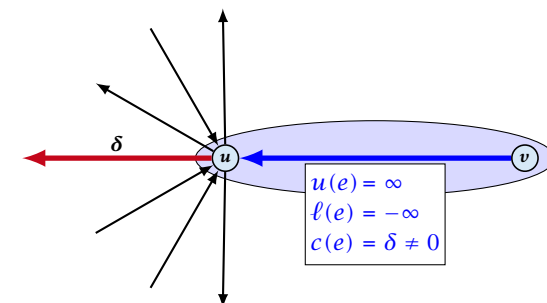


If  $c(e) = 0$  we can contract the edge/identify nodes  $u$  and  $v$ .

If  $c(e) \neq 0$  we can transform the graph so that  $c(e) = 0$ .

## Reduction II

We can transform any network so that a particular edge has cost  $c(e) = 0$ :

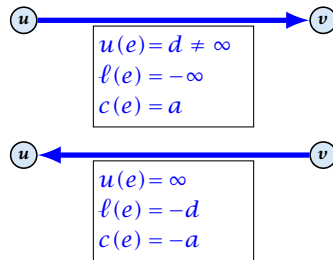


Additionally we set  $b(u) = 0$ .

## Reduction III

$$\begin{aligned} \min \quad & \sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \ell(e) \leq f(e) \leq u(e) \\ & \forall v \in V: f(v) = b(v) \end{aligned}$$

We can assume that  $\ell(e) \neq -\infty$ :

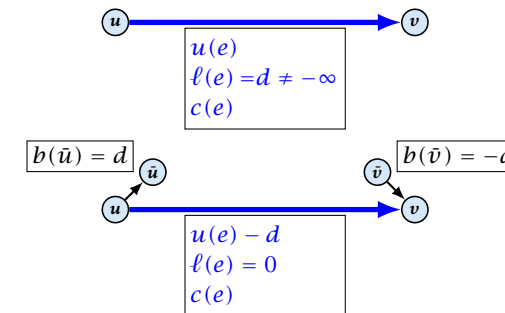


Replace the edge by an edge in opposite direction.

## Reduction IV

$$\begin{aligned} \min \quad & \sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \ell(e) \leq f(e) \leq u(e) \\ & \forall v \in V: f(v) = b(v) \end{aligned}$$

We can assume that  $\ell(e) = 0$ :

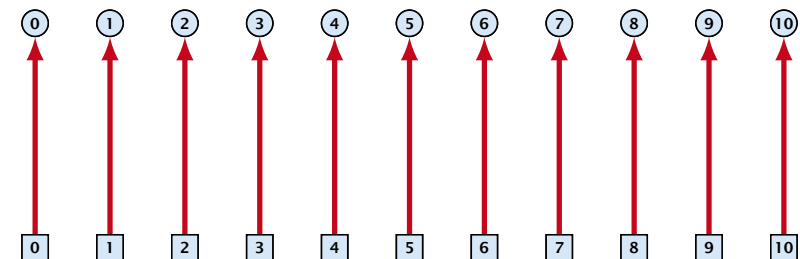


The added edges have infinite capacity and cost  $c(e)/2$ .

## Applications

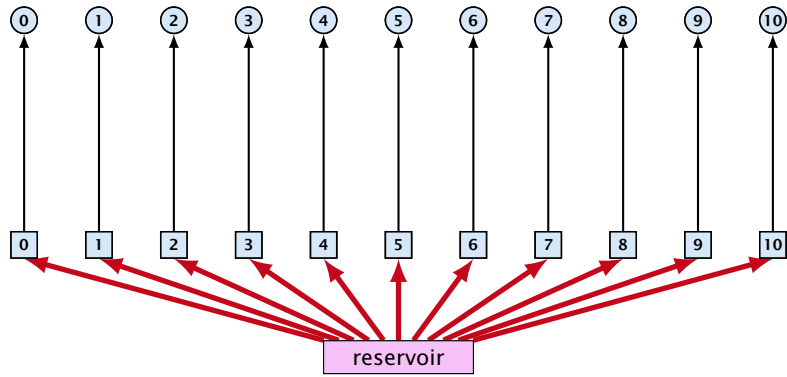
### Caterer Problem

- ▶ She needs to supply  $r_i$  napkins on  $N$  successive days.
- ▶ She can buy new napkins at  $p$  cents each.
- ▶ She can launder them at a fast laundry that takes  $m$  days and cost  $f$  cents a napkin.
- ▶ She can use a slow laundry that takes  $k > m$  days and costs  $s$  cents each.
- ▶ At the end of each day she should determine how many to send to each laundry and how many to buy in order to fulfill demand.
- ▶ Minimize cost.

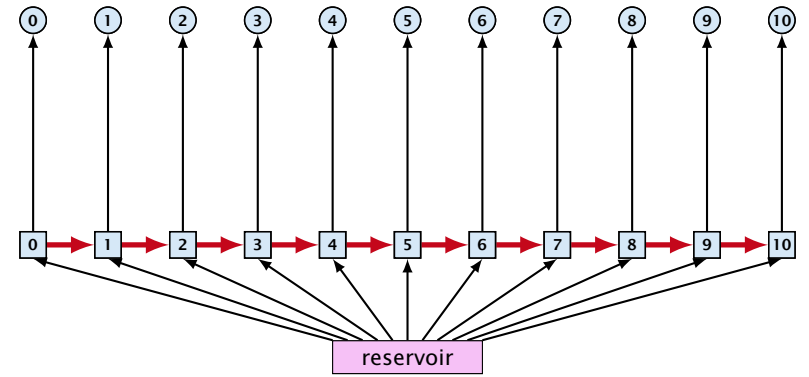


day edges:

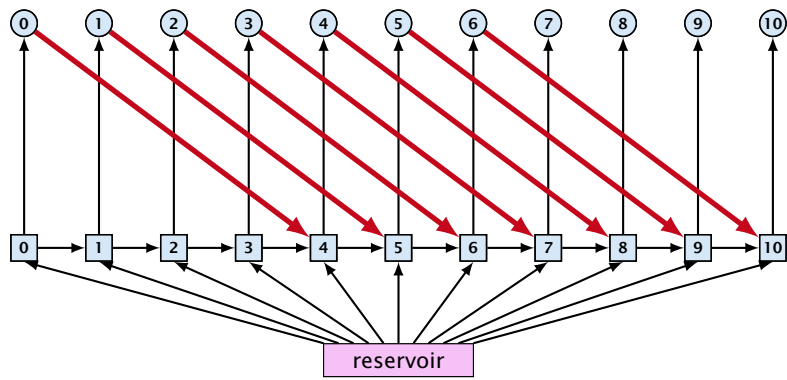
$$\begin{aligned} \text{upper bound: } & u(e_i) = \infty; \\ \text{lower bound: } & \ell(e_i) = r_i; \\ \text{cost: } & c(e) = 0 \end{aligned}$$



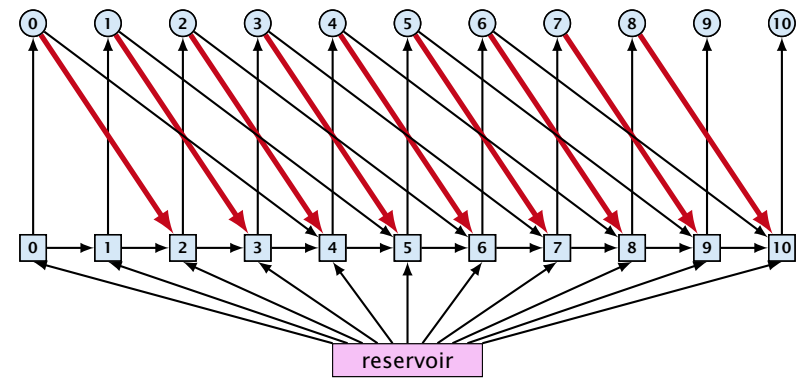
**buy edges:** upper bound:  $u(e_i) = \infty$ ;  
 lower bound:  $\ell(e_i) = 0$ ;  
 cost:  $c(e) = p$



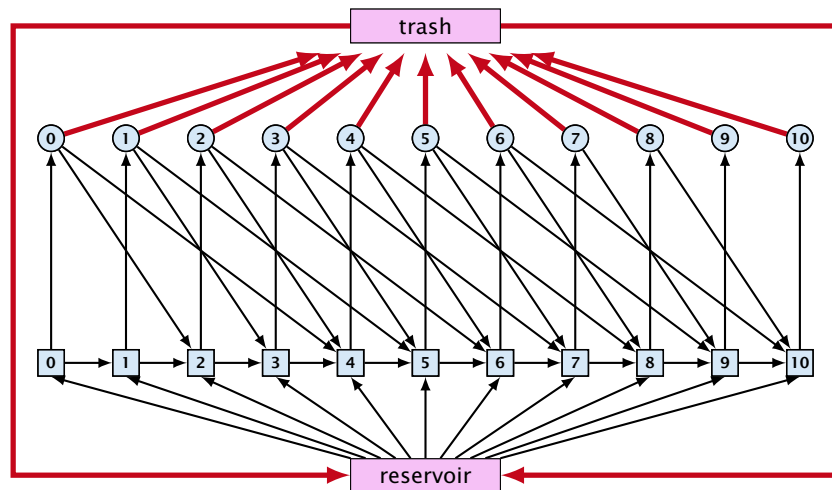
**forward edges:** upper bound:  $u(e_i) = \infty$ ;  
 lower bound:  $\ell(e_i) = 0$ ;  
 cost:  $c(e) = 0$



**slow edges:** upper bound:  $u(e_i) = \infty$ ;  
 lower bound:  $\ell(e_i) = 0$ ;  
 cost:  $c(e) = s$



**fast edges:** upper bound:  $u(e_i) = \infty$ ;  
 lower bound:  $\ell(e_i) = 0$ ;  
 cost:  $c(e) = f$



trash edges:

upper bound:  $u(e_i) = \infty$ ;  
 lower bound:  $\ell(e_i) = 0$ ;  
 cost:  $c(e) = 0$

## Residual Graph

### Version A:

The residual graph  $G'$  for a mincost flow is just a copy of the graph  $G$ .

If we send  $f(e)$  along an edge, the corresponding edge  $e'$  in the residual graph has its lower and upper bound changed to  $\ell(e') = \ell(e) - f(e)$  and  $u(e') = u(e) - f(e)$ .

### Version B:

The residual graph for a mincost flow is exactly defined as the residual graph for standard flows, with the only exception that one needs to define a cost for the residual edge.

For a flow of  $z$  from  $u$  to  $v$  the residual edge  $(v, u)$  has capacity  $z$  and a cost of  $-c((u, v))$ .

## 14 Mincost Flow

A **circulation** in a graph  $G = (V, E)$  is a function  $f: E \rightarrow \mathbb{R}^+$  that has an excess flow  $f(v) = 0$  for every node  $v \in V$ .

A circulation is **feasible** if it fulfills capacity constraints, i.e.,  $f(e) \leq u(e)$  for every edge of  $G$ .

### Lemma 87

A given flow is a mincost-flow if and only if the corresponding residual graph  $G_f$  does not have a feasible circulation of negative cost.

⇒ Suppose that  $g$  is a feasible circulation of negative cost in the residual graph.

Then  $f + g$  is a feasible flow with cost  $\text{cost}(f) + \text{cost}(g) < \text{cost}(f)$ . Hence,  $f$  is not minimum cost.

⇐ Let  $f$  be a non-mincost flow, and let  $f^*$  be a min-cost flow. We need to show that the residual graph has a feasible circulation with negative cost.

Clearly  $f^* - f$  is a circulation of negative cost. One can also easily see that it is feasible for the residual graph. (after sending  $-f$  in the residual graph (pushing all flow back) we arrive at the original graph; for this  $f^*$  is clearly feasible)



For previous slide:

$g = f^* - f$  is obtained by computing  $\Delta(e) = f^*(e) - f(e)$  for every edge  $e = (u, v)$ . If the result is positive set  $g((u, v)) = \Delta(e)$  and  $g((v, u)) = 0$ . Otherwise set  $g((u, v)) = 0$  and  $g((v, u)) = -\Delta(e)$ .

## 14 Mincost Flow

### Lemma 88

A graph (without zero-capacity edges) has a feasible circulation of negative cost if and only if it has a negative cycle w.r.t. edge-weights  $c : E \rightarrow \mathbb{R}$ .

Proof.

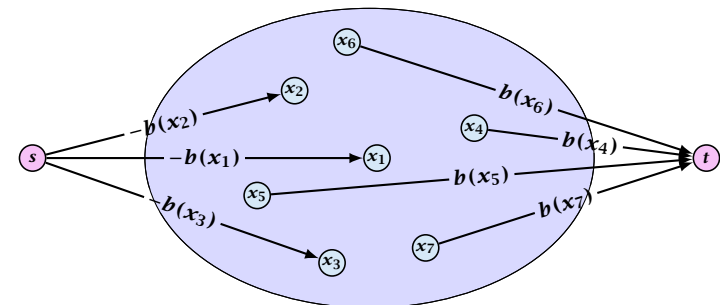
- ▶ Suppose that we have a negative cost circulation.
- ▶ Find directed cycle only using edges that have non-zero flow.
- ▶ If this cycle has negative cost you are done.
- ▶ Otherwise send flow in opposite direction along the cycle until the bottleneck edge(s) does not carry any flow.
- ▶ You still have a circulation with negative cost.
- ▶ Repeat.

## 14 Mincost Flow

### Algorithm 23 CycleCanceling( $G = (V, E), c, u, b$ )

- 1: establish a feasible flow  $f$  in  $G$
- 2: **while**  $G_f$  contains negative cycle **do**
- 3:   use Bellman-Ford to find a negative circuit  $Z$
- 4:    $\delta \leftarrow \min\{u_f(e) \mid e \in Z\}$
- 5:   augment  $\delta$  units along  $Z$  and update  $G_f$

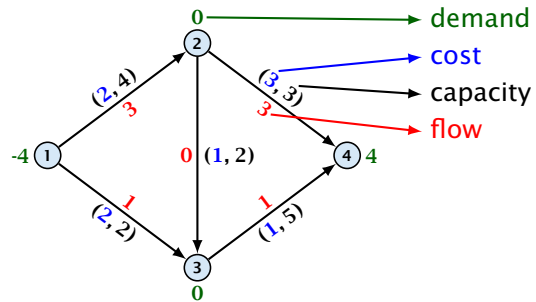
## How do we find the initial feasible flow?



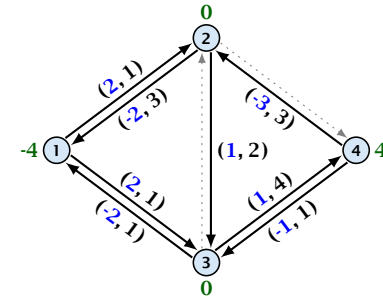
- ▶ Connect new node  $s$  to all nodes with negative  $b(v)$ -value.
- ▶ Connect nodes with positive  $b(v)$ -value to a new node  $t$ .
- ▶ There exist a feasible flow in the original graph iff in the resulting graph there exists an  $s$ - $t$  flow of value

$$\sum_{v:b(v)<0} (-b(v)) = \sum_{v:b(v)>0} b(v) .$$

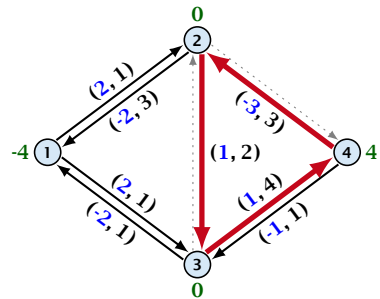
# 14 Mincost Flow



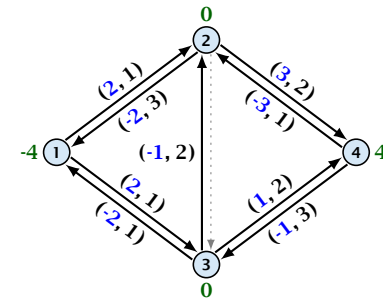
# 14 Mincost Flow



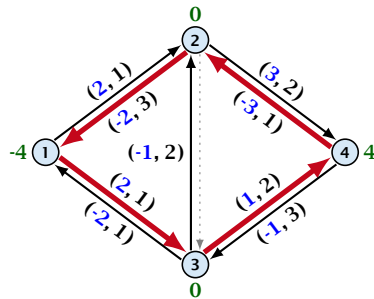
# 14 Mincost Flow



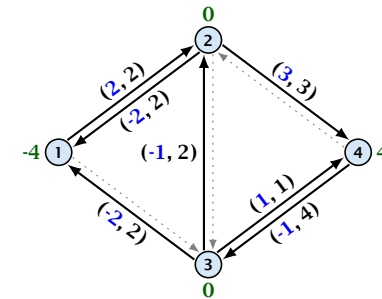
# 14 Mincost Flow



## 14 Mincost Flow



## 14 Mincost Flow



## 14 Mincost Flow

### Lemma 89

The improving cycle algorithm runs in time  $\mathcal{O}(nm^2CU)$ , for integer capacities and costs, when for all edges  $e$ ,  $|c(e)| \leq C$  and  $|u(e)| \leq U$ .

- ▶ Running time of Bellman-Ford is  $\mathcal{O}(mn)$ .
- ▶ Pushing flow along the cycle can be done in time  $\mathcal{O}(n)$ .
- ▶ Each iteration decreases the total cost by at least 1.
- ▶ The true optimum cost must lie in the interval  $[-mCU, \dots, +mCU]$ .

Note that this lemma is weak since it does not allow for edges with infinite capacity.

## 14 Mincost Flow

A general mincost flow problem is of the following form:

$$\begin{aligned} \min \quad & \sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \ell(e) \leq f(e) \leq u(e) \\ & \forall v \in V: a(v) \leq f(v) \leq b(v) \end{aligned}$$

where  $a: V \rightarrow \mathbb{R}$ ,  $b: V \rightarrow \mathbb{R}$ ;  $\ell: E \rightarrow \mathbb{R} \cup \{-\infty\}$ ,  $u: E \rightarrow \mathbb{R} \cup \{\infty\}$   
 $c: E \rightarrow \mathbb{R}$ ;

### Lemma 90 (without proof)

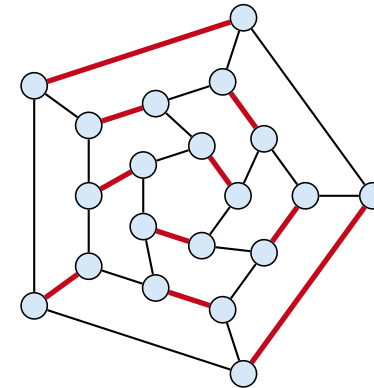
A general mincost flow problem can be solved in polynomial time.

# Part V

## Matchings

## Matching

- ▶ Input: undirected graph  $G = (V, E)$ .
- ▶  $M \subseteq E$  is a **matching** if each node appears in at most one edge in  $M$ .
- ▶ Maximum Matching: find a matching of maximum cardinality



## 16 Bipartite Matching via Flows

### Which flow algorithm to use?

- ▶ Generic augmenting path:  $\mathcal{O}(m \text{val}(f^*)) = \mathcal{O}(mn)$ .
- ▶ Capacity scaling:  $\mathcal{O}(m^2 \log C) = \mathcal{O}(m^2)$ .
- ▶ Shortest augmenting path:  $\mathcal{O}(mn^2)$ .

For **unit capacity simple graphs** shortest augmenting path can be implemented in time  $\mathcal{O}(m\sqrt{n})$ .

## 17 Augmenting Paths for Matchings

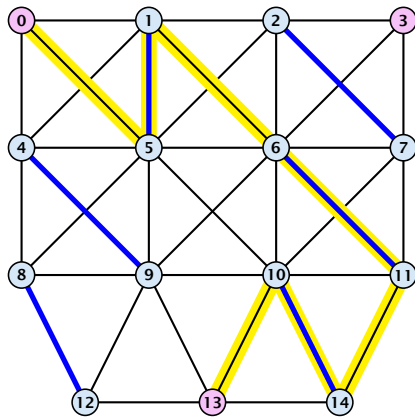
### Definitions.

- ▶ Given a matching  $M$  in a graph  $G$ , a vertex that is not incident to any edge of  $M$  is called a **free vertex** w. r. t.  $M$ .
- ▶ For a matching  $M$  a path  $P$  in  $G$  is called an **alternating path** if edges in  $M$  alternate with edges not in  $M$ .
- ▶ An alternating path is called an **augmenting path** for matching  $M$  if it ends at distinct free vertices.

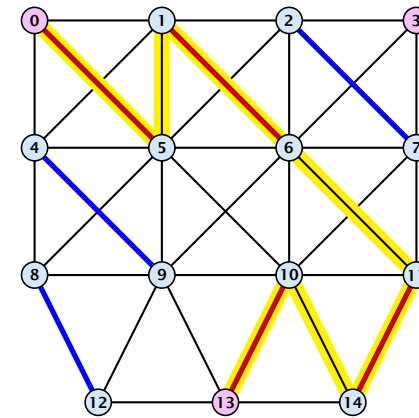
### Theorem 91

*A matching  $M$  is a maximum matching if and only if there is no augmenting path w. r. t.  $M$ .*

## Augmenting Paths in Action



## Augmenting Paths in Action



## 17 Augmenting Paths for Matchings

### Proof.

- ⇒ If  $M$  is maximum there is no augmenting path  $P$ , because we could switch matching and non-matching edges along  $P$ . This gives matching  $M' = M \oplus P$  with larger cardinality.
- ⇐ Suppose there is a matching  $M'$  with larger cardinality. Consider the graph  $H$  with edge-set  $M' \oplus M$  (i.e., only edges that are in either  $M$  or  $M'$  but not in both).

Each vertex can be incident to at most two edges (one from  $M$  and one from  $M'$ ). Hence, the connected components are alternating cycles or alternating path.

As  $|M'| > |M|$  there is one connected component that is a path  $P$  for which both endpoints are incident to edges from  $M'$ .  $P$  is an alternating path.

## 17 Augmenting Paths for Matchings

### Algorithmic idea:

As long as you find an augmenting path augment your matching using this path. When you arrive at a matching for which no augmenting path exists you have a maximum matching.

### Theorem 92

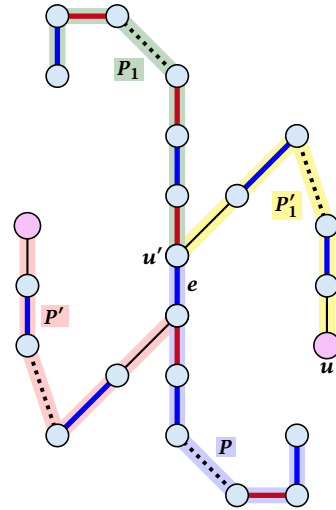
Let  $G$  be a graph,  $M$  a matching in  $G$ , and let  $u$  be a free vertex w.r.t.  $M$ . Further let  $P$  denote an augmenting path w.r.t.  $M$  and let  $M' = M \oplus P$  denote the matching resulting from augmenting  $M$  with  $P$ . If there was no augmenting path starting at  $u$  in  $M$  then there is no augmenting path starting at  $u$  in  $M'$ .

The above theorem allows for an easier implementation of an augmenting path algorithm. Once we checked for augmenting paths starting from  $u$  we don't have to check for such paths in future rounds.

# 17 Augmenting Paths for Matchings

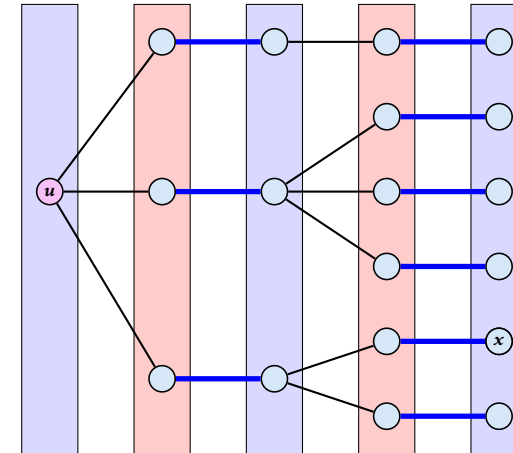
## Proof

- Assume there is an augmenting path  $P'$  w.r.t.  $M'$  starting at  $u$ .
- If  $P'$  and  $P$  are node-disjoint,  $P'$  is also augmenting path w.r.t.  $M$  ( $\neq$ ).
- Let  $u'$  be the first node on  $P'$  that is in  $P$ , and let  $e$  be the matching edge from  $M'$  incident to  $u'$ .
- $u'$  splits  $P$  into two parts one of which does not contain  $e$ . Call this part  $P_1$ . Denote the sub-path of  $P'$  from  $u$  to  $u'$  with  $P'_1$ .
- $P_1 \circ P'_1$  is augmenting path in  $M$  ( $\neq$ ).



# How to find an augmenting path?

## Construct an alternating tree.



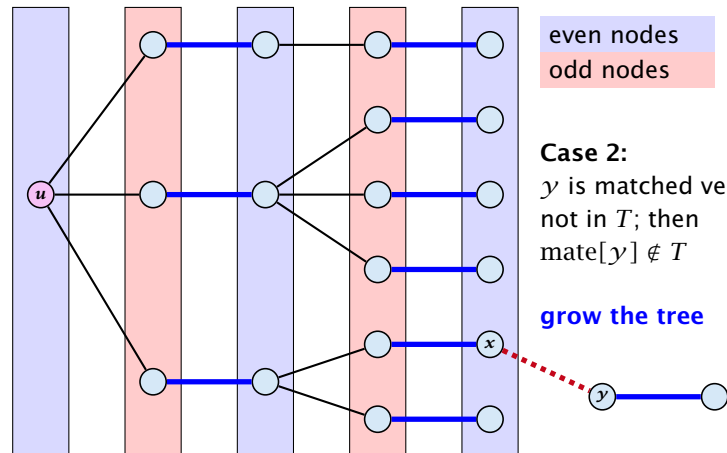
even nodes  
odd nodes

Case 1:  
 $y$  is free vertex not contained in  $T$

you found alternating path

# How to find an augmenting path?

## Construct an alternating tree.



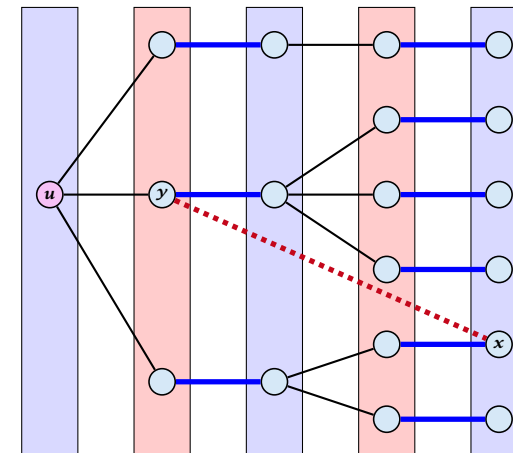
even nodes  
odd nodes

Case 2:  
 $y$  is matched vertex not in  $T$ ; then  $\text{mate}[y] \notin T$

grow the tree

# How to find an augmenting path?

## Construct an alternating tree.



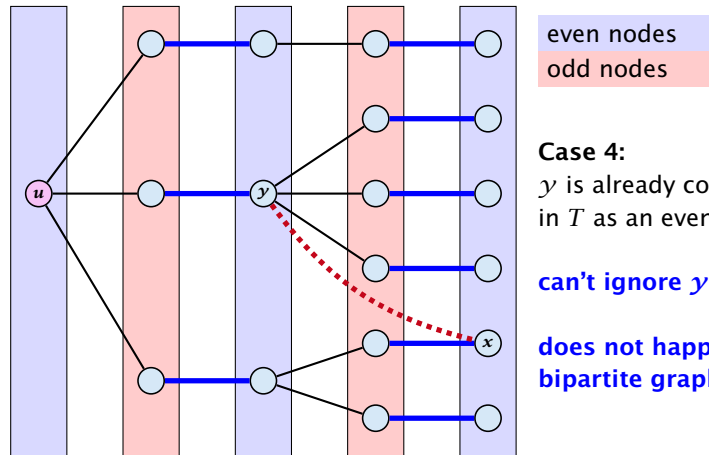
even nodes  
odd nodes

Case 3:  
 $y$  is already contained in  $T$  as an odd vertex

ignore successor  $y$

## How to find an augmenting path?

Construct an alternating tree.



**Case 4:**  
y is already contained  
in  $T$  as an even vertex

can't ignore  $y$

does not happen in  
bipartite graphs

## Algorithm 24 BiMatch( $G, match$ )

```

1: for  $x \in V$  do  $mate[x] \leftarrow 0$ ;
2:  $r \leftarrow 0$ ;  $free \leftarrow n$ ;
3: while  $free \geq 1$  and  $r < n$  do
4:    $r \leftarrow r + 1$ 
5:   if  $mate[r] = 0$  then
6:     for  $i = 1$  to  $n$  do  $parent[i] \leftarrow 0$ 
7:      $Q \leftarrow \emptyset$ ;  $Q.append(r)$ ;  $aug \leftarrow false$ ;
8:     while  $aug = false$  and  $Q \neq \emptyset$  do
9:        $x \leftarrow Q.dequeue()$ ;
10:      for  $y \in A_x$  do
11:        if  $mate[y] = 0$  then
12:           $augm(mate, parent, y)$ ;
13:           $aug \leftarrow true$ ;
14:           $free \leftarrow free - 1$ ;
15:        else
16:          if  $parent[y] = 0$  then
17:             $parent[y] \leftarrow x$ ;
18:             $Q.enqueue(mate[y])$ ;

```

The lecture  
slides contain a  
step by step

graph  $G = (S \cup S', E)$   
 $S = \{1, \dots, n\}$   
 $S' = \{1', \dots, n'\}$

## 18 Weighted Bipartite Matching

### Weighted Bipartite Matching/Assignment

- ▶ Input: undirected, bipartite graph  $G = L \cup R, E$ .
- ▶ an edge  $e = (\ell, r)$  has weight  $w_e \geq 0$
- ▶ find a matching of maximum weight, where the weight of a matching is the sum of the weights of its edges

### Simplifying Assumptions (wlog [why?]):

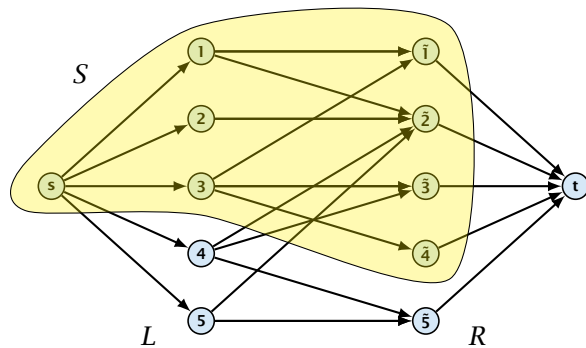
- ▶ assume that  $|L| = |R| = n$
- ▶ assume that there is an edge between every pair of nodes  $(\ell, r) \in V \times V$
- ▶ can assume goal is to construct maximum weight **perfect** matching

## Weighted Bipartite Matching

### Theorem 93 (Halls Theorem)

A bipartite graph  $G = (L \cup R, E)$  has a perfect matching if and only if for all sets  $S \subseteq L$ ,  $|\Gamma(S)| \geq |S|$ , where  $\Gamma(S)$  denotes the set of nodes in  $R$  that have a neighbour in  $S$ .

## 18 Weighted Bipartite Matching



## Halls Theorem

### Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in  $S$  could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph  $G'$  is at least  $|L|$ .
  - ▶ Let  $S$  denote a minimum cut and let  $L_S \triangleq L \cap S$  and  $R_S \triangleq R \cap S$  denote the portion of  $S$  inside  $L$  and  $R$ , respectively.
  - ▶ Clearly, all neighbours of nodes in  $L_S$  have to be in  $S$ , as otherwise we would cut an edge of infinite capacity.
  - ▶ This gives  $R_S \geq |\Gamma(L_S)|$ .
  - ▶ The size of the cut is  $|L| - |L_S| + |R_S|$ .
  - ▶ Using the fact that  $|\Gamma(L_S)| \geq L_S$  gives that this is at least  $|L|$ .



## Algorithm Outline

### Idea:

We introduce a node weighting  $\vec{x}$ . Let for a node  $v \in V$ ,  $x_v \in \mathbb{R}$  denote the weight of node  $v$ .

- ▶ Suppose that the node weights dominate the edge-weights in the following sense:

$$x_u + x_v \geq w_e \text{ for every edge } e = (u, v).$$

- ▶ Let  $H(\vec{x})$  denote the subgraph of  $G$  that only contains edges that are **tight** w.r.t. the node weighting  $\vec{x}$ , i.e. edges  $e = (u, v)$  for which  $w_e = x_u + x_v$ .
- ▶ Try to compute a perfect matching in the subgraph  $H(\vec{x})$ . If you are successful you found an optimal matching.



## Algorithm Outline

### Reason:

- ▶ The weight of your matching  $M^*$  is

$$\sum_{(u,v) \in M^*} w_{(u,v)} = \sum_{(u,v) \in M^*} (x_u + x_v) = \sum_v x_v.$$

- ▶ Any other perfect matching  $M$  (in  $G$ , not necessarily in  $H(\vec{x})$ ) has

$$\sum_{(u,v) \in M} w_{(u,v)} \leq \sum_{(u,v) \in M} (x_u + x_v) = \sum_v x_v.$$





## Algorithm Outline

### What if you don't find a perfect matching?

Then, Hall's theorem guarantees you that there is a set  $S \subseteq L$ , with  $|\Gamma(S)| < |S|$ , where  $\Gamma$  denotes the neighbourhood w.r.t. the subgraph  $H(\vec{x})$ .

**Idea:** reweight such that:

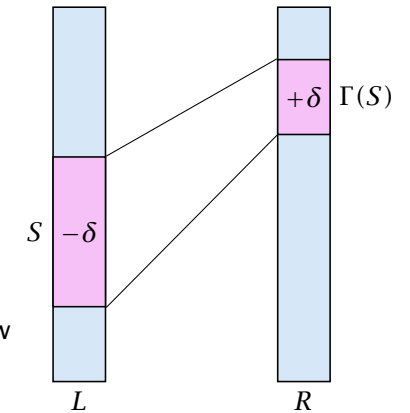
- ▶ the total weight assigned to nodes decreases
- ▶ the weight function still dominates the edge-weights

If we can do this we have an algorithm that terminates with an optimal solution (we analyze the running time later).

## Changing Node Weights

Increase node-weights in  $\Gamma(S)$  by  $+\delta$ , and decrease the node-weights in  $S$  by  $-\delta$ .

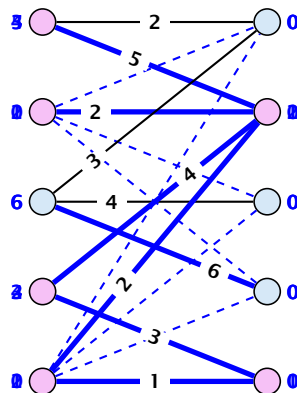
- ▶ Total node-weight decreases.
- ▶ Only edges from  $S$  to  $R - \Gamma(S)$  decrease in their weight.
- ▶ Since, none of these edges is tight (otw. the edge would be contained in  $H(\vec{x})$ , and hence would go between  $S$  and  $\Gamma(S)$ ) we can do this decrement for small enough  $\delta > 0$  until a new edge gets tight.



## Weighted Bipartite Matching

Edges not drawn have weight 0.

$$\delta = 1 \quad \delta = 1$$



## Analysis

### How many iterations do we need?

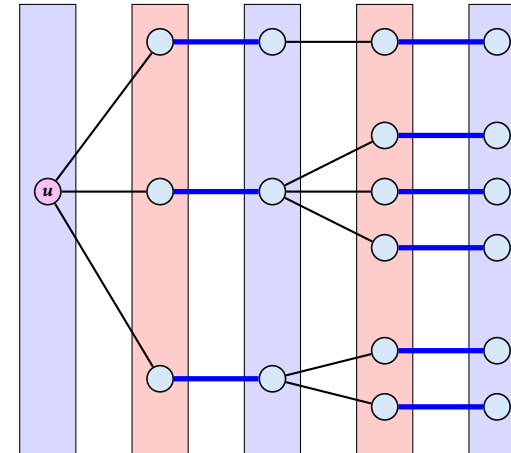
- ▶ One reweighting step increases the number of edges out of  $S$  by at least one.
- ▶ Assume that we have a maximum matching that saturates the set  $\Gamma(S)$ , in the sense that every node in  $\Gamma(S)$  is matched to a node in  $S$  (we will show that we can always find  $S$  and a matching such that this holds).
- ▶ This matching is still contained in the new graph, because all its edges either go between  $\Gamma(S)$  and  $S$  or between  $L - S$  and  $R - \Gamma(S)$ .
- ▶ Hence, reweighting does not decrease the size of a maximum matching in the tight sub-graph.

## Analysis

- ▶ We will show that after at most  $n$  reweighting steps the size of the maximum matching can be increased by finding an augmenting path.
- ▶ This gives a polynomial running time.

## How to find an augmenting path?

Construct an alternating tree.



## Analysis

### How do we find $S$ ?

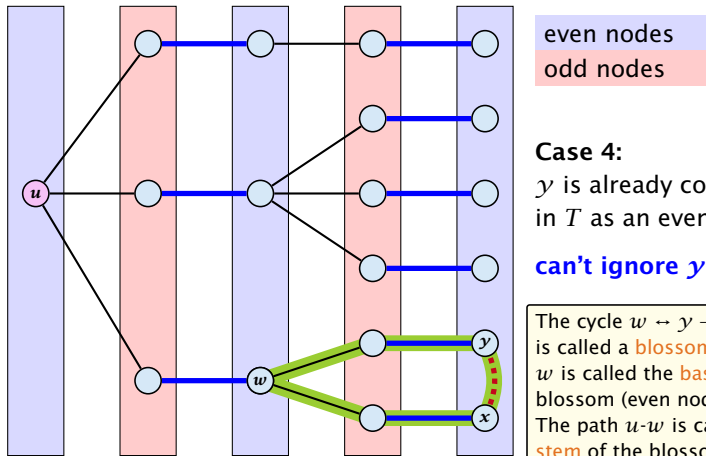
- ▶ Start on the left and compute an alternating tree, starting at any free node  $u$ .
- ▶ If this construction stops, there is no perfect matching in the tight subgraph (because for a perfect matching we need to find an augmenting path starting at  $u$ ).
- ▶ The set of even vertices is on the left and the set of odd vertices is on the right **and** contains all neighbours of even nodes.
- ▶ All odd vertices are matched to even vertices. Furthermore, the even vertices additionally contain the free vertex  $u$ . Hence,  $|V_{\text{odd}}| = |E(V_{\text{even}})| < |V_{\text{even}}|$ , and all odd vertices are saturated in the current matching.

## Analysis

- ▶ The current matching does not have any edges from  $V_{\text{odd}}$  to  $L \setminus V_{\text{even}}$  (edges that may possibly be deleted by changing weights).
- ▶ After changing weights, there is at least one more edge connecting  $V_{\text{even}}$  to a node outside of  $V_{\text{odd}}$ . After at most  $n$  reweightings we can do an augmentation.
- ▶ A reweighting can be trivially performed in time  $\mathcal{O}(n^2)$  (keeping track of the tight edges).
- ▶ An augmentation takes at most  $\mathcal{O}(n)$  time.
- ▶ In total we obtain a running time of  $\mathcal{O}(n^4)$ .
- ▶ A more careful implementation of the algorithm obtains a running time of  $\mathcal{O}(n^3)$ .

## How to find an augmenting path?

Construct an alternating tree.



**Case 4:**  
 $y$  is already contained  
in  $T$  as an even vertex

can't ignore  $y$

The cycle  $w \leftrightarrow y - x \leftrightarrow w$   
is called a **blossom**.  
 $w$  is called the **base** of  
the blossom (even node!!!).  
The path  $u-w$  is called the  
**stem** of the blossom.

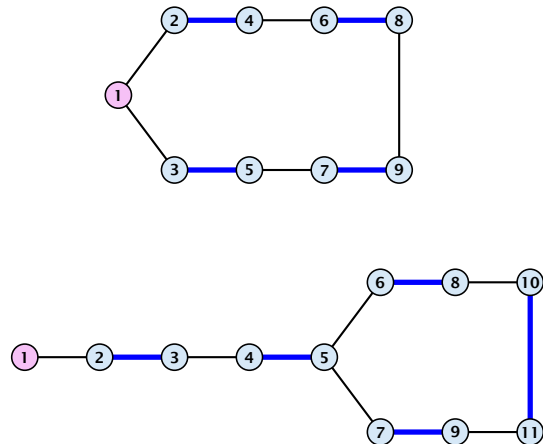
## Flowers and Blossoms

### Definition 94

A **flower** in a graph  $G = (V, E)$  w.r.t. a matching  $M$  and a (free) root node  $r$ , is a subgraph with two components:

- ▶ A **stem** is an even length alternating path that starts at the root node  $r$  and terminates at some node  $w$ . We permit the possibility that  $r = w$  (empty stem).
- ▶ A **blossom** is an odd length alternating cycle that starts and terminates at the terminal node  $w$  of a stem and has no other node in common with the stem.  $w$  is called the **base** of the blossom.

## Flowers and Blossoms



## Flowers and Blossoms

### Properties:

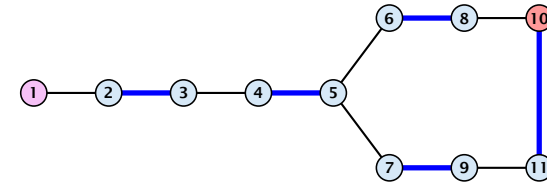
1. A stem spans  $2\ell + 1$  nodes and contains  $\ell$  matched edges for some integer  $\ell \geq 0$ .
2. A blossom spans  $2k + 1$  nodes and contains  $k$  matched edges for some integer  $k \geq 1$ . The matched edges match all nodes of the blossom except the base.
3. The base of a blossom is an even node (if the stem is part of an alternating tree starting at  $r$ ).

## Flowers and Blossoms

### Properties:

4. Every node  $x$  in the blossom (except its base) is reachable from the root (or from the base of the blossom) through two distinct alternating paths; one with even and one with odd length.
5. The even alternating path to  $x$  terminates with a matched edge and the odd path with an unmatched edge.

## Flowers and Blossoms



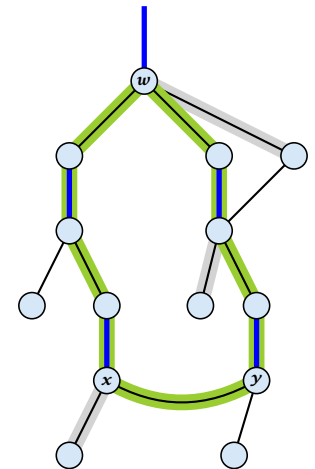
## Shrinking Blossoms

When during the alternating tree construction we discover a blossom  $B$  we replace the graph  $G$  by  $G' = G/B$ , which is obtained from  $G$  by contracting the blossom  $B$ .

- ▶ Delete all vertices in  $B$  (and its incident edges) from  $G$ .
- ▶ Add a new (pseudo-)vertex  $b$ . The new vertex  $b$  is connected to all vertices in  $V \setminus B$  that had at least one edge to a vertex from  $B$ .

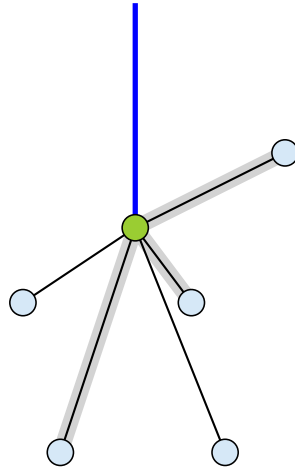
## Shrinking Blossoms

- ▶ Edges of  $T$  that connect a node  $u$  not in  $B$  to a node in  $B$  become tree edges in  $T'$  connecting  $u$  to  $b$ .
- ▶ Matching edges (there is at most one) that connect a node  $u$  not in  $B$  to a node in  $B$  become matching edges in  $M'$ .
- ▶ Nodes that are connected in  $G$  to at least one node in  $B$  become connected to  $b$  in  $G'$ .



## Shrinking Blossoms

- ▶ Edges of  $T$  that connect a node  $u$  not in  $B$  to a node in  $B$  become tree edges in  $T'$  connecting  $u$  to  $b$ .
- ▶ Matching edges (there is at most one) that connect a node  $u$  not in  $B$  to a node in  $B$  become matching edges in  $M'$ .
- ▶ Nodes that are connected in  $G$  to at least one node in  $B$  become connected to  $b$  in  $G'$ .



## Example: Blossom Algorithm

Animation of Blossom Shrinking algorithm is only available in the lecture version of the slides.

## Correctness

Assume that in  $G$  we have a flower w.r.t. matching  $M$ . Let  $r$  be the root,  $B$  the blossom, and  $w$  the base. Let graph  $G' = G/B$  with pseudonode  $b$ . Let  $M'$  be the matching in the contracted graph.

### Lemma 95

If  $G'$  contains an augmenting path  $P'$  starting at  $r$  (or the pseudo-node containing  $r$ ) w.r.t. the matching  $M'$  then  $G$  contains an augmenting path starting at  $r$  w.r.t. matching  $M$ .

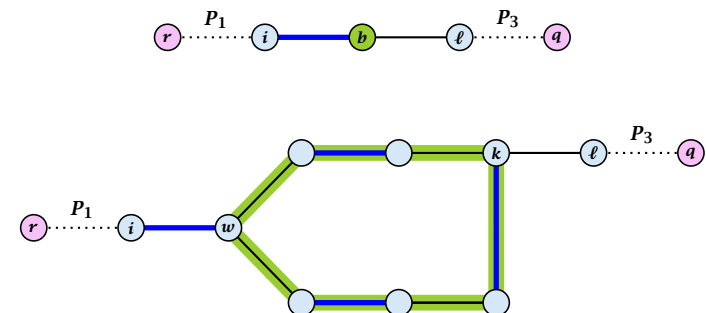
## Correctness

### Proof.

If  $P'$  does not contain  $b$  it is also an augmenting path in  $G$ .

### Case 1: non-empty stem

- ▶ Next suppose that the stem is non-empty.



## Correctness

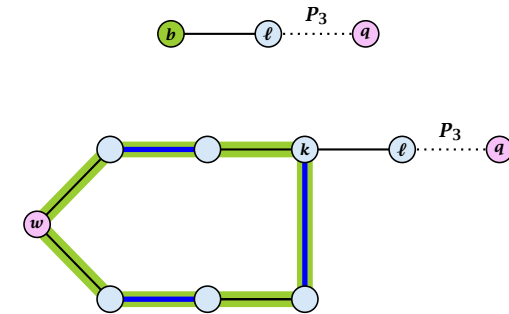
- ▶ After the expansion  $\ell$  must be incident to some node in the blossom. Let this node be  $k$ .
- ▶ If  $k \neq w$  there is an alternating path  $P_2$  from  $w$  to  $k$  that ends in a matching edge.
- ▶  $P_1 \circ (i, w) \circ P_2 \circ (k, \ell) \circ P_3$  is an alternating path.
- ▶ If  $k = w$  then  $P_1 \circ (i, w) \circ (w, \ell) \circ P_3$  is an alternating path.

## Correctness

### Proof.

#### Case 2: empty stem

- ▶ If the stem is empty then after expanding the blossom,  $w = r$ .



- ▶ The path  $r \circ P_2 \circ (k, \ell) \circ P_3$  is an alternating path.

## Correctness

### Lemma 96

If  $G$  contains an augmenting path  $P$  from  $r$  to  $q$  w.r.t. matching  $M$  then  $G'$  contains an augmenting path from  $r$  (or the pseudo-node containing  $r$ ) to  $q$  w.r.t.  $M'$ .

## Correctness

### Proof.

- ▶ If  $P$  does not contain a node from  $B$  there is nothing to prove.
- ▶ We can assume that  $r$  and  $q$  are the only free nodes in  $G$ .

#### Case 1: empty stem

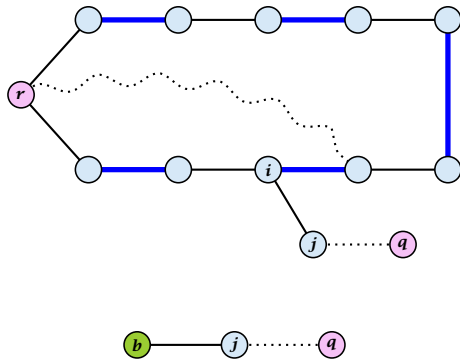
Let  $i$  be the last node on the path  $P$  that is part of the blossom.

$P$  is of the form  $P_1 \circ (i, j) \circ P_2$ , for some node  $j$  and  $(i, j)$  is unmatched.

$(b, j) \circ P_2$  is an augmenting path in the contracted network.

## Correctness

### Illustration for Case 1:



## Correctness

### Case 2: non-empty stem

Let  $P_3$  be alternating path from  $r$  to  $w$ ; this exists because  $r$  and  $w$  are root and base of a blossom. Define  $M_+ = M \oplus P_3$ .

In  $M_+$ ,  $r$  is matched and  $w$  is unmatched.

$G$  must contain an augmenting path w.r.t. matching  $M_+$ , since  $M$  and  $M_+$  have same cardinality.

This path must go between  $w$  and  $q$  as these are the only unmatched vertices w.r.t.  $M_+$ .

For  $M'_+$  the blossom has an empty stem. Case 1 applies.

$G'$  has an augmenting path w.r.t.  $M'_+$ . It must also have an augmenting path w.r.t.  $M'$ , as both matchings have the same cardinality.

This path must go between  $r$  and  $q$ .

The lecture slides contain a step by step explanation.

#### Algorithm 25 search( $r, found$ )

- 1: set  $\bar{A}(i) \leftarrow A(i)$  for all nodes  $i$
- 2:  $found \leftarrow false$
- 3: unlabeled all nodes;
- 4: give an even label to  $r$  and initialize  $list \leftarrow \{r\}$
- 5: **while**  $list \neq \emptyset$  **do**
- 6:     delete a node  $i$  from  $list$
- 7:     examine( $i, found$ )
- 8:     **if**  $found = true$  **then return**

Search for an augmenting path starting at  $r$ .

The lecture slides contain a step by step explanation.

#### Algorithm 26 examine( $i, found$ )

- 1: **for all**  $j \in \bar{A}(i)$  **do**
- 2:     **if**  $j$  is even **then** contract( $i, j$ ) and **return**
- 3:     **if**  $j$  is unmatched **then**
- 4:          $q \leftarrow j$ ;
- 5:         pred( $q$ )  $\leftarrow i$ ;
- 6:          $found \leftarrow true$ ;
- 7:         **return**
- 8:     **if**  $j$  is matched and unlabeled **then**
- 9:         pred( $j$ )  $\leftarrow i$ ;
- 10:         pred(mate( $j$ ))  $\leftarrow j$ ;
- 11:         add mate( $j$ ) to  $list$

Examine the neighbours of a node  $i$

**Algorithm 27**  $\text{contract}(i, j)$

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Contract blossom identified by nodes  $i$  and  $j$

**Algorithm 27**  $\text{contract}(i, j)$

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Get all nodes of the blossom.  
Time:  $\mathcal{O}(m)$

**Algorithm 27**  $\text{contract}(i, j)$

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Identify all neighbours of  $b$ .  
Time:  $\mathcal{O}(m)$  (how?)

**Algorithm 27**  $\text{contract}(i, j)$

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

$b$  will be an even node, and it has unexamined neighbours.



### Algorithm 27 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Every node that was adjacent to a node in  $B$  is now adjacent to  $b$

### Algorithm 27 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Only for making a blossom expansion easier.

### Algorithm 27 contract( $i, j$ )

- 1: trace pred-indices of  $i$  and  $j$  to identify a blossom  $B$
- 2: create new node  $b$  and set  $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(x)$
- 3: label  $b$  even and add to *list*
- 4: update  $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$  for each  $j \in \bar{A}(b)$
- 5: form a circular double linked list of nodes in  $B$
- 6: delete nodes in  $B$  from the graph

Only delete links from nodes not in  $B$  to  $B$ .  
When expanding the blossom again we can recreate these links in time  $\mathcal{O}(m)$ .

## Analysis

- ▶ A contraction operation can be performed in time  $\mathcal{O}(m)$ . Note, that any graph created will have at most  $m$  edges.
- ▶ The time between two contraction-operation is basically a BFS/DFS on a graph. Hence takes time  $\mathcal{O}(m)$ .
- ▶ There are at most  $n$  contractions as each contraction reduces the number of vertices.
- ▶ The expansion can trivially be done in the same time as needed for all contractions.
- ▶ An augmentation requires time  $\mathcal{O}(n)$ . There are at most  $n$  of them.
- ▶ In total the running time is at most

$$n \cdot (\mathcal{O}(mn) + \mathcal{O}(n)) = \mathcal{O}(mn^2) .$$

## Example: Blossom Algorithm

Animation of Blossom Shrinking algorithm is only available in the lecture version of the slides.

## A Fast Matching Algorithm

### Algorithm 28 Bimatch-Hopcroft-Karp( $G$ )

```
1:  $M \leftarrow \emptyset$ 
2: repeat
3:   let  $\mathcal{P} = \{P_1, \dots, P_k\}$  be maximal set of
4:   vertex-disjoint, shortest augmenting path w.r.t.  $M$ .
5:    $M \leftarrow M \oplus (P_1 \cup \dots \cup P_k)$ 
6: until  $\mathcal{P} = \emptyset$ 
7: return  $M$ 
```

We call one iteration of the repeat-loop a **phase** of the algorithm.

## Analysis Hopcroft-Karp

### Lemma 97

Given a matching  $M$  and a maximal matching  $M^*$  there exist  $|M^*| - |M|$  **vertex-disjoint** augmenting path w.r.t.  $M$ .

### Proof:

- ▶ Similar to the proof that a matching is optimal iff it does not contain an augmenting path.
- ▶ Consider the graph  $G = (V, M \oplus M^*)$ , and mark edges in this graph blue if they are in  $M$  and red if they are in  $M^*$ .
- ▶ The connected components of  $G$  are cycles and paths.
- ▶ The graph contains  $k \stackrel{\text{def}}{=} |M^*| - |M|$  more red edges than blue edges.
- ▶ Hence, there are at least  $k$  components that form a path starting and ending with a red edge. These are augmenting paths w.r.t.  $M$ .

## Analysis Hopcroft-Karp

- ▶ Let  $P_1, \dots, P_k$  be a maximal collection of vertex-disjoint, shortest augmenting paths w.r.t.  $M$  (let  $\ell = |P_i|$ ).
- ▶  $M' \stackrel{\text{def}}{=} M \oplus (P_1 \cup \dots \cup P_k) = M \oplus P_1 \oplus \dots \oplus P_k$ .
- ▶ Let  $P$  be an augmenting path in  $M'$ .

### Lemma 98

The set  $A \stackrel{\text{def}}{=} M \oplus (M' \oplus P) = (P_1 \cup \dots \cup P_k) \oplus P$  contains at least  $(k+1)\ell$  edges.

## Analysis Hopcroft-Karp

### Proof.

- ▶ The set describes exactly the symmetric difference between matchings  $M$  and  $M' \oplus P$ .
- ▶ Hence, the set contains at least  $k + 1$  vertex-disjoint augmenting paths w.r.t.  $M$  as  $|M'| = |M| + k + 1$ .
- ▶ Each of these paths is of length at least  $\ell$ .

## Analysis Hopcroft-Karp

### Lemma 99

$P$  is of length at least  $\ell + 1$ . This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.

### Proof.

- ▶ If  $P$  does not intersect any of the  $P_1, \dots, P_k$ , this follows from the maximality of the set  $\{P_1, \dots, P_k\}$ .
- ▶ Otherwise, at least one edge from  $P$  coincides with an edge from paths  $\{P_1, \dots, P_k\}$ .
- ▶ This edge is not contained in  $A$ .
- ▶ Hence,  $|A| \leq k\ell + |P| - 1$ .
- ▶ The lower bound on  $|A|$  gives  $(k + 1)\ell \leq |A| \leq k\ell + |P| - 1$ , and hence  $|P| \geq \ell + 1$ .

## Analysis Hopcroft-Karp

If the shortest augmenting path w.r.t. a matching  $M$  has  $\ell$  edges then the cardinality of the maximum matching is of size at most  $|M| + \frac{|V|}{\ell + 1}$ .

### Proof.

The symmetric difference between  $M$  and  $M^*$  contains  $|M^*| - |M|$  vertex-disjoint augmenting paths. Each of these paths contains at least  $\ell + 1$  vertices. Hence, there can be at most  $\frac{|V|}{\ell + 1}$  of them.

## Analysis Hopcroft-Karp

### Lemma 100

The Hopcroft-Karp algorithm requires at most  $2\sqrt{|V|}$  phases.

### Proof.

- ▶ After iteration  $\lfloor \sqrt{|V|} \rfloor$  the length of a shortest augmenting path must be at least  $\lfloor \sqrt{|V|} \rfloor + 1 \geq \sqrt{|V|}$ .
- ▶ Hence, there can be at most  $|V| / (\sqrt{|V|} + 1) \leq \sqrt{|V|}$  additional augmentations.

## Analysis Hopcroft-Karp

### Lemma 101

One phase of the Hopcroft-Karp algorithm can be implemented in time  $\mathcal{O}(m)$ .

construct a “level graph”  $G'$ :

- ▶ construct Level 0 that includes all free vertices on left side  $L$
  - ▶ construct Level 1 containing all neighbors of Level 0
  - ▶ construct Level 2 containing **matching** neighbors of Level 1
  - ▶ construct Level 3 containing all neighbors of Level 2
  - ▶ ...
  - ▶ stop when a level (apart from Level 0) contains a free vertex
- can be done in time  $\mathcal{O}(m)$  by a modified BFS

## Analysis Hopcroft-Karp

- ▶ a shortest augmenting path **must** go from Level 0 to the last layer constructed
- ▶ it can only use edges between layers
- ▶ construct a maximal set of vertex disjoint augmenting path connecting the layers
- ▶ for this, go forward until you either reach a free vertex or you reach a “dead end”  $v$
- ▶ if you reach a free vertex delete the augmenting path and all incident edges from the graph
- ▶ if you reach a dead end backtrack and delete  $v$  together with its incident edges

## Analysis Hopcroft-Karp

See lecture versions of the slides.

## Analysis: Shortest Augmenting Path for Flows

**cost for searches during a phase is  $\mathcal{O}(mn)$**

- ▶ a search (successful or unsuccessful) takes time  $\mathcal{O}(n)$
- ▶ a search deletes at least one edge from the level graph

**there are at most  $n$  phases**

Time:  $\mathcal{O}(mn^2)$ .

## Analysis for Unit-capacity Simple Networks

cost for searches during a phase is  $\mathcal{O}(m)$

- ▶ an edge/vertex is traversed at most twice

need at most  $\mathcal{O}(\sqrt{n})$  phases

- ▶ after  $\sqrt{n}$  phases there is a cut of size at most  $\sqrt{n}$  in the residual graph
- ▶ hence at most  $\sqrt{n}$  additional augmentations required

Time:  $\mathcal{O}(m\sqrt{n})$ .

