Algorithms and Data Structures (DA4006)

Marc Hellmuth

Department of Mathematics
Stockholm University

Contents

Abstract

Dear Students,

This manuscript serves as a summary of the main parts covered in the lecture. However, it is primarily an automatically generated script from LaTeX Beamer slides. This means that detailed additional information within the text is not provided. Nevertheless, I hope this summary assists you in better understanding the main details and provides a more substantial overview than just bare slides.

Let's get started, Marc

Chapter 0

Organisational Matters

All information / news / exercises etc.pp. can be found online: `https://kurser.math.su.se/`
Course: DA4006
Course examination is done in four parts:

- Home assignments ("LABO") worth 3 HP, graded A-F.

  Consists of ≥ 4 individual exercise sheets
- A written 4 hour exam ("THEO"), worth 4.5 HP, graded A-F.

You pass the course if LABO and exam have been passed.

To pass LABO, at least 50% of the exercises must be correct (in total).
To pass THEO, at least 50% of THEO must be correct.

All solutions must be provided in English!
Team work to discuss the LABO exercises is allowed and also recommended.
BUT:

- everyone has to hand in an individual and independent solution of the exercies
- you must be able to explain your solutions in the tutorial
- no copies of solutions
- always write your name on your solutions

There are deadlines when to hand-in specified at the DA4006 webpage and ex-sheets.
Hand in the respectice exercise *before* the end of the deadline as follows:

- LABO-exercises should be provided as single PDF-file (handwritten-scanned or latex-PDF),

  except for programming exercises (here provide src-files and a readme about how to compile or use the prg).
- upload the files at the course homepage under the respective assignment link.

TIME MANAGEMENT

- SU homepage: *"1.5 HP = 40 hours"*
- LABO (≥ 4 Exercises) $= 3$ HP $=$ *total* $80h$
- Hence each Exercise corresponds to $\sim 10 - 20h$.

The exercises are not super difficult but possibly time-intensive!

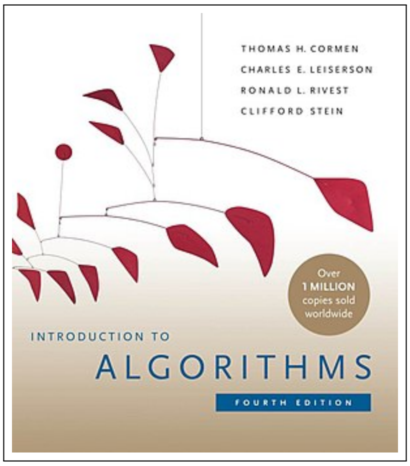
START EARLY! START EARLY! START EARLY!

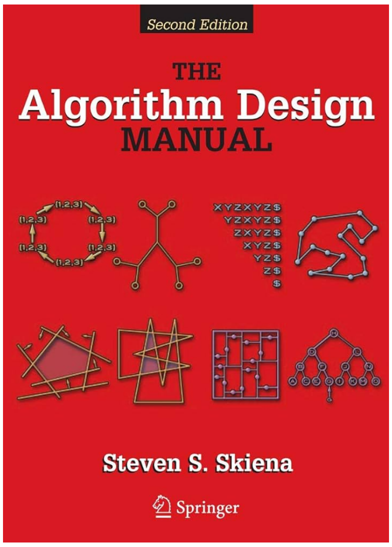
Students with medical diagnoses that may impair their concentration or reading ability, or anything else that hinders them from providing exercises on time, are requested to inform me ***before the first exercise hand-in!***

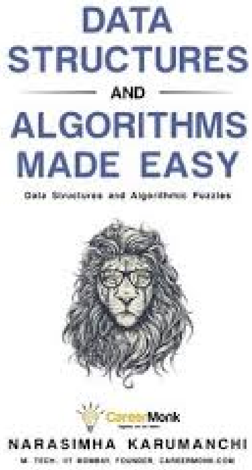
- all slides will be provided online.
- The handwritten notes I use to create content on the board are also available online.

  BUT: These notes are for me to create board content for you. That means the readability of the script is not guaranteed. So, it's better to come to the lectures to see what I have written within the script.

course books:







Primary course book: Introduction to Algorithms, Cormen at al

Chapter 1

Fundamentals

Part 1 focuses on the following topics.

1-1 What is an algorithm?

1-2 Correctness of algorithms

1-3 Runtime of algorithms & space complexity

The latter Parts **1-1**, **1-2**, **1-3** will, in particular, be examined on a sorting algorithm `Insertion_Sort`. Further examples will be provided. We then continue with

1-4 Elementary Data Structures

1.1 What is an algorithm?

Informally, an algorithm is a step-by-step unambiguous instructions to solve a given problem by taking some some value(s) as input and by producing some value(s) as output.

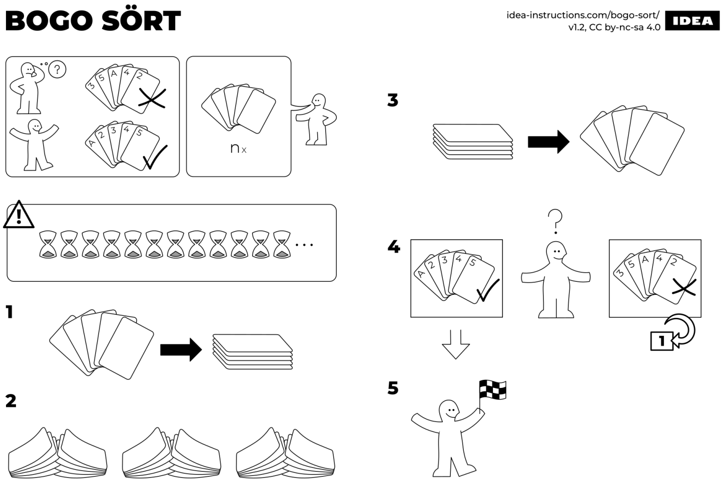
```
Cooking_Pasta(Water, Pasta, Salt)
    Add 1ℓ water to pot
    Add salt to pot
    Boil-up Water
    Add pasta to pot
    Cook until done
    Drain water
    RETURN Cooked delicious pasta
```

HOW TO COOK PASTA
1
pour water
2
add salt
3
212°F
100°C
bring to a boil
4
add pasta
5
al dente - 5-6 min
until the end - 8-10 min
cook until done
6
drain the water
7
bon appetit!

Question: unambiguous? executable by some machine/robot? . . .

A human may know how to "boil-up" water by using a cooking plate (. . . or open fire . . . ?) but does a robot know this?

```
Bogo_Sort*($n$ cards)
    1 Align cards to a pack-of-cards
    2 Shuffle cards 3 times
    3 Spread cards
    4 IF (cards are ordered) THEN
            goto step 5
      ELSE goto step 1
    5 RETURN Sorted Cart Deck
```



Question: unambiguous (is "order" well-defined)? does it terminate (runtime)?...

(780–850) Persian mathematician, astronomer, geographer, ...

The word 'algorithm' has its roots in the name of Persian mathematician Muhammad ibn Musa **al-Khwarizmi**.

He wrote a fundamental treatise on the "Hindu–Arabic numeral system" which was translated into Latin during the 12th century.

Here: al-Khwarizmi was translated into Algorizmi

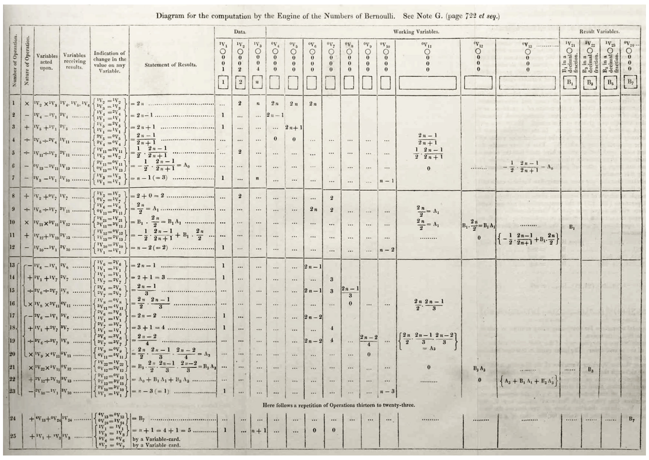


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 et seq.)

(1815-1852) English mathematician

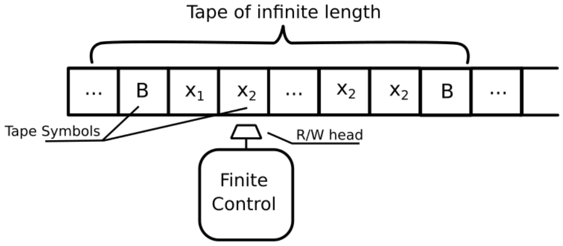
[*] AKA stupid sort

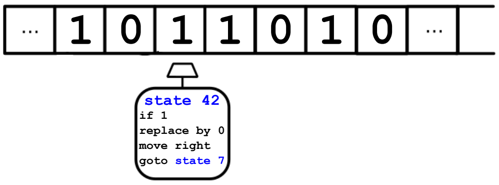
The first computer program was written by Ada Lovelace for the “Analytical Engine” [design for a simple mechanical computer] by Charles Babbage to compute Bernoulli numbers.

Definition (algorithm)**.** *A calculation rule for a problem is called* ***algorithm*** *if there is a Turing machine equivalent to this calculation rule which stops for every input that has a solution.*

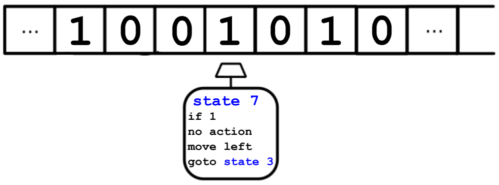
The formal definition of algorithm goes back to Alan Turing who designed Turing machines as a theoretical concept to simulate the operating principles of a computer (central processing unit = CPU)

A Turing machine is a mathematical model of computation describing an abstract machine that manipulates symbols on a strip of tape according to a table of rules.





```
state 42
if 1
replace by 0
move right
goto state 7
```



```
state 7
if 1
no action
move left
goto state 3
```

(1912-1954) English mathematician, computer scientist, logician, . . .

Whatever any computer can do, can be done by a Turing machine !!

To check whether a "calculation rule / program" is in fact an algorithm, we must design a Turing machine mimicking these rules.

This is beyond the scope of this course.

However, to establish algorithms and to analyze their costs, we need to have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs while still having a "correct" notion of algorithm (Turing machines).

There are other computer-models that are "equivalent" to Turing machines and that are closer to the notion of computers that we know. This "equivalence" allows us to use these models instead.

Excellent overview of Turing machines: https://plato.stanford.edu/entries/turing-machine/ https://www.youtube.com/watch?v=dNRDvLACg5Q

Whatever any computer can do, can be done by those models and thus, by a Turing machine !!
"Equivalent" to Turing machines are register machines. One of them are random-access machines (RAM): an abstract model of computers that is "closest" to the common notion of a computer and where instructions are executed one after another, with no concurrent operations.

(unlimited) memory $\mathrm{MEM}[0], \mathrm{MEM}[1], \mathrm{MEM}[2], \ldots$

fixed number of registers $R_1, \ldots, R_k$
[Registers are the memory locations that the CPU can access directly. The registers contain operands or the instructions that the processor is currently accessing.]

memory and registers store w-bit integers $n \in \{0, \ldots, 2^w - 1\}$

instructions:

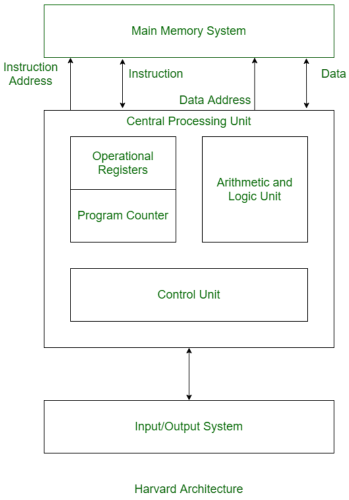
load/store $R_i = \mathrm{MEM}[j]$, $\mathrm{MEM}[j] = R_i$

basic operations on registers:
$R_k = R_i + R_j$ (arithmetic is *modulo* 2^w!)
also $R_k = R_i - R_j$, $R_i * R_j$, $R_i \mathrm{div} R_j$, $R_i \mathrm{mod} R_j$
[these basic operations are "easy" to be implement on hardware]

conditional / unconditional jumps
[algorithms are lines of instructions, jump back and forth to these lines]

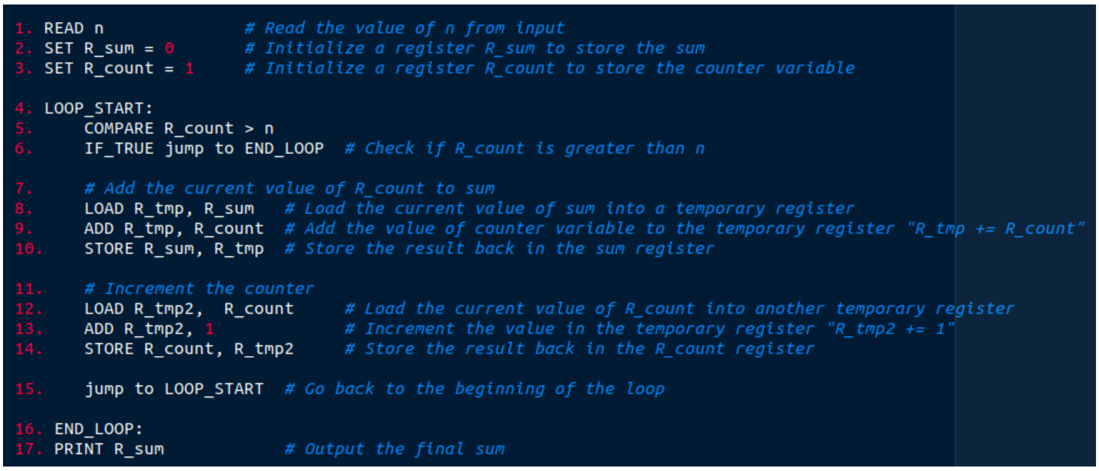
costs = number of executed step-by-step instructions (i.e., each instruction takes constant time)

Full Details about RAM in THE DESIGN AND ANALYSIS OF COMPUTER ALGORITHMS; Aho et al. 1974



Harvard Architecture

Already simplified, but typical RAM-code (here for computing $\sum_{i=1}^{n} i$)



```
READ n                  # Read the value of n from input
SET R_sum = 0           # Initialize a register R_sum to store the sum
SET R_count = 1         # Initialize a register R_count to store the counter variable

LOOP_START:
    COMPARE R_count > n
    IF_TRUE jump to END_LOOP  # Check if R_count is greater than n

    # Add the current value of R_count to sum
    LOAD R_tmp, R_sum   # Load the current value of sum into a temporary register
    ADD R_tmp, R_count  # Add the value of counter variable to the temporary register "R_tmp += R_count"
    STORE R_sum, R_tmp  # Store the result back in the sum register

    # Increment the counter
    LOAD R_tmp2,  R_count     # Load the current value of R_count into another temporary register
    ADD R_tmp2, 1             # Increment the value in the temporary register "R_tmp2 += 1"
    STORE R_count, R_tmp2     # Store the result back in the R_count register

    jump to LOOP_START  # Go back to the beginning of the loop

END_LOOP:
PRINT R_sum             # Output the final sum
```

Example: Board

RAM code for $\sum_{i=1}^{n} i$:

```
1.  n = 2
2.  R_sum = 0
3.  R_count = 1
5.   1 > 2
6.   not true
8.  R_tmp = 0        (Load necessary!)
9.  R_tmp = R_Tmp + Rcount = 0 + 1
10. Rsum = 1

12   R_tmp 2 = 1
13   R tmp 2 = 2
14   Rcount = 2

15   go to line 4

5   2 > 2
6   not true

8   R_tmp = 1
9   Rtmp = RTmp + Rcount = 1 + 2
10  Rsum = 3

12-14: Rcount = 3
15   goto line 4
5     3 > 2
6     true → go to Line 16
17    print 3
```

Strictly speaking, we should precisely define the instructions of the RAM model and their costs (*which is quite cumbersome!*). Nevertheless, the latter should give you a general idea of the term “algorithm” and we provide here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

instructions	Example
basic arithmetics	addition, subtraction, multiplication, division, floor, ceiling, modulo,. . .
data movement (save/load/copy)	init array A of size n and save number k at its i-entry ($A[i] := k$)
conditional execution	`IF` (condition) `THEN` Instruction `ELSE` Some-Other-Instruction
jump	`GOTO` instruction x (or line x)
iteration	`WHILE` (condition) `DO` Instruction `REPEAT` Instruction `UNTIL` (condition) `FOR` ($j = 1$ to $n-1$) `DO` Instruction
recursion	algorithm calls itself e.g. $n! = 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n = (n-1)! \cdot n$

Classical ways to represent algorithms briefly explained on the "Sum-up" problem:
Compute $total\_sum = \sum_{i=1}^{n} i$ for a given integer n.

Verbal "*We define total_sum to be* 0 *and then add to total_sum the integer* 1*, then we add* 2*, . . . , then we add* n*.*"

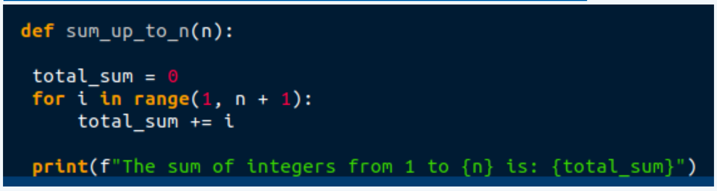
for communication of ideas often sufficient, usually indicates only implicitly sequence of instructions
[must be careful here when it comes to checking costs!]

pseudocode

```
Sum(n)

    $total\_sum := 0$
    FOR ($i := 1$ to $n$) DO
        $total\_sum := total\_sum + i$
    PRINT $total\_sum$
```

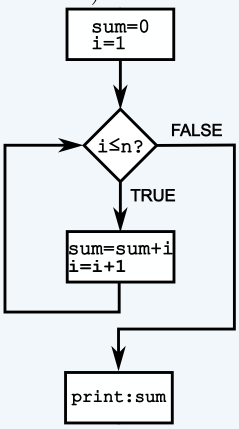
The pseudo-code description is in a "free form", mixing English words and "programming-like" statements as long as it serves the purpose of conveying –without ambiguity– how our algorithm runs.

Some "real" programming language (here python)



```
def sum_up_to_n(n):

 total_sum = 0
 for i in range(1, n + 1):
     total_sum += i

 print(f"The sum of integers from 1 to {n} is: {total_sum}")
```

diagram/flowchart (good to "see" the step-by-step instructions)



1.2 Correctness of algorithms

We are, in particular, interested in algorithms that solve problems:

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the previous "sum-up" problem are the n for a specific integer (e.g. $n = 3$)

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

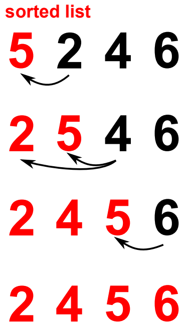
[$\Longleftrightarrow$ an incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer]

Let's have a look to a specific problem, design an algorithm and show its correctness, i.e., it solves this problem.

An Example: Sorting Problem

Given: A finite sequence of integers $(a_1, a_2 \dots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \dots, a'_n)$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

$A = (5, 2, 4, 6)$ should become $(2, 4, 5, 6)$



A simple sorting "algorithm" idea:

We assume to have an order list (*highlighted in red*)

Then, subsequently insert the next element x into this sorted list by comparing x with the elements in sorted list from right to left

We put this into an algorithm, known as `Insertion_Sort`.

```
//$A$ is array of size $n$ containing the integers, 1st entry A[0]
Insertion_Sort($A$)

FOR ($j := 1$ to $n-1$) DO
    $key := A[j]$
    //insert $A[j]$ into the sorted sequence $A[0 \dots j-1]$
    $i := j-1$
    WHILE ($i \geq 0$ and $A[i] > key$)
        $A[i+1] := A[i]$
        $i := i-1$
    $A[i+1] := key$
RETURN Sorted array $A$
```

Example: `Insertion_Sort`(A) for $A = [5, 2, 4, 6, 1]$:

```
[5, 5, 4, 6, 1] Line 5
[2, 5, 4, 6, 1] Line 7
[2, 5, 5, 6, 1] Line 5
[2, 4, 5, 6, 1] Line 7
[2, 4, 5, 6, 1] Line 7
[2, 4, 5, 6, 6] Line 5
[2, 4, 5, 5, 6] Line 5
[2, 4, 4, 5, 6] Line 5
[2, 2, 4, 5, 6] Line 5
[1, 2, 4, 5, 6] Line 7
```

This shows that `Insertion_Sort` correctly works precisely for the input sequence $(5, 2, 4, 6, 1)$.

An **instance** of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.

Example: Instances of the sorting-problem are all finite sequences of integers, e.g. $(1, 2, 3)$, $(1, 1, 1, \ldots, 1)$, $(5, 4, 5, 3)$, ...

An algorithm is **correct** or **solves** the problem if, for every input instance, it halts with the correct output w.r.t. the given problem.

So-far, we showed that `Insertion_Sort` correctly works only for the specific instance $(5, 2, 4, 6, 1)$.

Let's prove its correctness, i.e., we show that `Insertion_Sort` terminates and correctly returns a sorted list for any finite input sequence of integers.

Theorem. *`Insertion_Sort` correctly sorts a given finite sequence A of integers.*

Proof. *board* via loop-invariants

Theorem 1: Iteration_Sort correctly solves the sorting problem.

proof: Observe that it is infact an algorithm:

- "unambigously" described.
- finite input $A = A(0) \ldots A(n-1)$
- finite output —"—
- "memory" n numbers must be stored + 2 variables i, j = finite.

- **terminates**

  FOR runs till $j = n$ stops

  WHILE: $i \leq j$ (L3)

  i decrement (L6) $\Rightarrow$ $i < 0$ at some iteration ($< n$ iteration)

  $\Rightarrow$ terminates

  $\Rightarrow$ Alg. terminates.

proof now correctness:

We use "loop invariants" = statement that is true across multiple iterations of a loop.

$A = (x_0 \cdots x_{n-1})$

in the following we write $A(i..j)$ for the elements $A[i] .. A[j]$, $0 \leq i \leq j \leq n-1$

loop-invariant $P(j)$ = at each step, $A(0..j-1)$ "consist" of all original elements $x_0 .. x_{j-1}$ but in sorted order.

Show: (I) $P(j)$ holds at start of j-th iteration of FOR-loop

(II) $P(j)$ —"— at end of j-th iteration of FOR loop

(III) $P(j)$ holds when alg. terminates.

j = 1 (= first iteration) : at start $A(0,0) = A[0]$ sorted, since it contains only 1 element, i.e, $P(1)$ is true.

at this point we can assume that $P(j)$ is true at start of j-th iteration of FOR-loop.
=> (I) holds

Show now that (II) holds.

2 cases : WHILE loop starts or does not run in jth iterat. of FOR loop

L3: $i = j - 1$

By assumption, (I) $P(j)$ holds:

$\circledast$ $A(0..j-1)$ is sorted & containing $x_0 .. x_{j-1}$

$A(0) \leq A(1) \leq .. \leq A(j-1)$

if WHILE-loop doesnot run

=> Since $i \geq 0$ as $j \geq 1$, we have $A(i) \leq key$. in L4.

=> L7 : $A(i+1) = key$, $i+1 = j$ => $A(j) = key$

$\overset{\circledast}{=>}$ $A(0) \leq A(1) \leq .. A(j-1) \leq A(j)$

=> $A(0..j)$ correctly ordered & contains $x_0 .. x_j$

if WHILE-loop runs

$\Rightarrow$ since $i \geq 0$ as $j \geq 1$, we have $A(i) > key$.

L5-6 ensure that entry $A(i)$ is "shifted"
to entry $A(i+1)$ as long
as $A(i) > k$ & $i \geq 0$

that is in L7

we have for

$$\begin{tikzcd} A: & \underline{0 \cdots\cdots} & i+1 & \underline{\cdots\cdots\cdots\cdots j} & key \arrow[ll, bend left] \end{tikzcd}$$

st $A(0..i+1)$ & $A(i+1..j)$ is sorted:

$A(0) \leq A(1) \leq \dots \leq A(i+1) \leq \dots A(j)$

& contains $x_0 \dots x_j$. $\Rightarrow$ II holds.

Since the latter holds for each iteration of FOR-loop,
$A(0...n-1)$ is sorted & contains $x_0 -- x_{n-1}$ □
[(III) hold]

□

We have discussed so-far the topics

1-1 What is an algorithm?

We shortly explained that TMs and equivalent models (RAM) are used to define the term algorithm

Deeper details are beyond the scope of this course. Instead, we provided here a "rough, approximate" definition:

A procedure is an algorithm if . . .

. . . it is a finite linear sequence of instructions (Instruction 1, followed by Instruction 2 and so on . . .)

. . . its input is a finite set of values

. . . in every step only a finite amount of memory is used

. . . has some finite set of values as output (in case it terminates)

This finite linear sequence of instructions can be written using e.g. pseudo-code or flowcharts

1-2 Correctness of algorithms

An algorithm is said to be **correct** if, for every input instance, it halts (=terminates) with the correct output w.r.t. the given computational problem. In this case, we also say that the algorithm **solves** the problem.

Parts **1-1** and **1-2** have been examined on a sorting algorithm `Insertion_Sort`.

Let's now continue with Part **1-3** "Runtime"

1.3 Runtime of algorithms

Naive idea: measure the time from start to end in (milli)seconds

say we want to know for some input N how fast the algorithm is:

$N = 4000$ and runtime 6.3 seconds
$N = 8000$ and runtime 51.1 seconds
$N = 16000$ and runtime 410.8 seconds

Hypothesis: For arbitrary N runtime is $\sim 10^{-10}N^3$

not really comparable since this can differ on distinct computers.
we need a notation to classify "runtime" that is independent on the "performance" of computer.

Time complexity

NOT: measure runtime on a specific computer

BUT: determine effort for idealized computer model (e.g. RAM-model)

We need *abstract* measure for time complexity to estimate *asymptotic* costs that depends on the size of the input

Add two numbers

$$\begin{array}{cccccc} & 1 & 1 & 1 & 3 & 7 \\ + & & 2 & 8 & 3 & 4 \\ \hline = & 1 & 3 & 9 & 7 & 1 \end{array}$$

Takes 5 single additions.

Hence, addition needs $\max\{m,n\}$ operations (even slightly more if we consider "carryover") for two numbers having m, resp., n digits.

There two main types of cost models:

- the unit-cost model assigns a constant cost to every machine operation, regardless of the size of the numbers involved.
- the logarithmic-cost model, assigns a cost to every machine operation proportional to the number of bits involved [Integer $n \in \{0,\ldots,2^w-1\}$ needs w bits to be stored]

  not used in this course, however, important e.g. in cryptography

In this course ***unit-cost model***

The RAM-model contains instructions commonly found in real computers:

arithmetic (such as add, subtract, multiply, divide, remainder, floor, ceiling),

data movement (load, store, copy), and

control (conditional and unconditional branch, subroutine call and return).

Each such instruction is counted as one time-unit and thus, takes a constant amount of time.
Hence, we essentially count the number of execution of instructions (as the number of operations)

We denote with $T(|I|)$ the runtime of an algorithm with input I. Here, $|I|$ is the size of the input and $T(|I|)$ is the number of operations/instructions used in this algorithm with input I.

Input $I = A$ with n entries: $|I| = n$

```
Count_Zeros(array $A$)
    int $i, count$
    $count := 0$
    FOR($i := 0$ to $n-1$)
        IF($A[i] == 0$) DO $count++$
```

variable declaration (e.g. int i, *count*):	2
assignment statement (e.g. $i := 0$):	2
increment (i and *count*)	$n+n$
compare "$A[i] == 0$"	n
Σsingle instructions $= T(n) =$	$3n+4$

Still, this is unsatisfying, e.g. if you have $T(n) = 3n+4$ vs $T'(n) = 4n$ *(which is faster?)*
For $n = 1,2,3,4$ we have $T(n) \geq T'(n)$ and $T(n) < T'(n)$ for $n > 5$

$T(|I|))$ = runtime of an algorithm (number of operations/instructions) with input I of size $|I|$.

Input $I = A$ array of n integer: $|I| = n$

```
Insertion_Sort($A$)

1 FOR ($j = 1$ to $n-1$) DO
2     $key = A[j]$
3     $i := j-1$
4     WHILE ($i \geq 0$ and $A[i] > key$)
5         $A[i+1] := A[i]$
6         $i := i-1$
7     $A[i+1] := key$
8 RETURN Sorted array $A$
```

	T_{line_nr}
1	$n-1$
2	$n-1$
3	$n-1$
4	$2\sum_{j=1}^{n-1} t_j$ with $t_j \geq 1$ is number of times the while-loop "test" (2 comparisons) is executed for that value of j
5	$\sum_{j=1}^{n-1}(t_j-1)$ here: (t_j-1) since "last" test of while-loop-test
6	$\sum_{j=1}^{n-1}(t_j-1)$ stops this loop and we have no extra run
7	$n-1$
8	1

Σsingle instructions $= T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} t_j + 2\sum_{j=1}^{n-1}(t_j-1) + 1$

Best-case, Worst-case and average-case analysis

To understand how good or bad an algorithm is in general, we must know how it works over *all* instances.

The worst-case complexity of an algorithm is the function defined by the maximum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $2n^2 - 1$

The best-case complexity of an algorithm is the function defined by the minimum number of steps/instructions taken in any instance I of size $|I|$.

Example insertion sort: $6n - 5$

The average-case complexity of an algorithm is the function defined by the average number of steps over all instances I of size $|I|$.

Example `Insertion_Sort`:

worst-case: A in "sorted order reversed", in which case we must compare every $key = A[j]$ with each value in $A[1 \dots j-1]$.

Then, body of while-loop executed $j-1$ times + one extra test to terminate while-loop $\implies t_j = j$ for all j.

$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1}(j-1) + 1$$

$$= 4(n-1) + 2\sum_{j=1}^{n-1} j + 2\sum_{j=1}^{n-1} j - \sum_{j=1}^{n-1} 2 + 1$$

$$= 4(n-1) + 4((n^2-n)/2) - 2(n-1) + 1 = 2n^2 - 1 \ (\textit{quadratic runtime})$$

best-case: A already sorted, in which case $A[i] \leq key$ for each $i = j-1$ and thus, $t_j = 1$ for all j (Line 5,6 not executed).

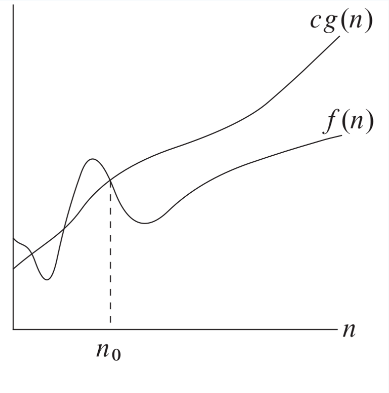
$$T(n) = 4(n-1) + 2\sum_{j=1}^{n-1} 1 + 0 + 1 = 4(n-1) + 2(n-1) + 1 = 6n - 5 \ (\textit{linear runtime})$$

average-case: Suppose that we randomly choose n numbers. How long does it take to determine where in subarray $A[1..j-1]$ to insert element $A[j]$? On average, half the elements in $A[1..j-1]$ are less than $A[j]$ and half the elements are are greater. On average, therefore, we check half of the subarray $A[1..j-1]$ and t_j is about $j/2$. The resulting average-case running time turns out to be a quadratic function of the input size (exercise).

In this course, we are mainly interested in the worst-case analysis, since it gives us an upper bound on the running time for any input. Knowing it provides a guarantee that the algorithm will never take any longer.
Note, in many cases, the worst-case occurs fairly often and the average-case is often roughly as bad as the worst-case.
As we have seen before, $T(n)$ is a function that depends on the input size n. However, we are, in general, not interested in specific values for n but the asymptotic behavior of $T(n)$ (that is for large n).

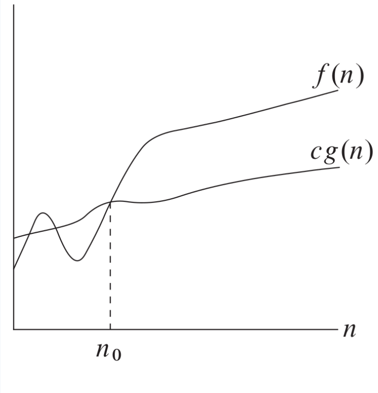
Notation Big-O, Big-Θ- and Big-Ω:

Definition.
$O(g(n)) \coloneqq \{f(n)\colon$ *there are positive constants* c *and* n_0 *such that* $0 \le f(n) \le cg(n)$ *for all* $n \ge n_0\}$



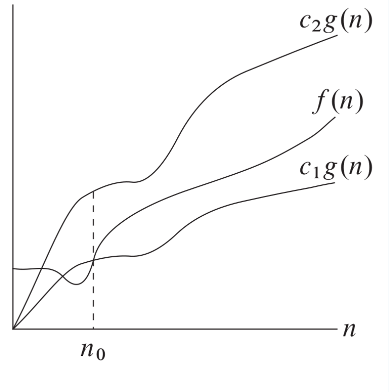
We use O*-notation to give an upper bound on a function, to within a constant factor (asymptotic upper bound)*

$\Omega(g(n)) \coloneqq \{f(n)\colon$ *there are positive constants* c *and* n_0 *such that* $0 \le cg(n) \le f(n)$ *for all* $n \ge n_0\}$



We use Ω*-notation to give a lower bound on a function, to within a constant factor (asymptotic lower bound)*

$\Theta(g(n)) \coloneqq \{f(n)\colon$ *there are positive constants* c_1, c_2 *and* n_0 *such that* $0 \le c_1g(n) \le f(n) \le c_2g(n)$ *for all* $n \ge n_0\}$



For all $n \ge n_0$ *, the function* $f(n)$ *is equal to* $g(n)$ *to within a constant factor (asymptotically tight bound for* $f(n)$*)*

Theorem. $f(n) \in O(g(n))$ *and* $f(n) \in \Omega(g(n))$ *if and only if* $f(n) \in \Theta(g(n))$.

[proof exercise]

Examples.

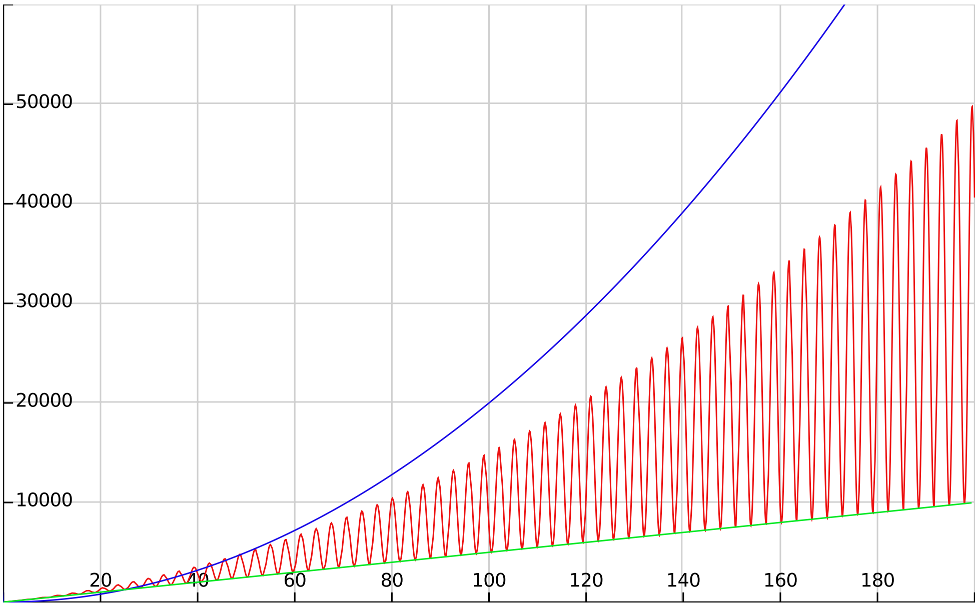
For $f(n) = 0.5n^2 + 3n$ show:

- $f(n) \in O(n^2)$; $f(n) \in \Omega(n^2)$ (and thus, $f(n) \in \Theta(n^2)$).
- $f(n) \notin O(n)$; $f(n) \in \Omega(n)$ (and thus, $f(n) \notin \Theta(n)$).
- $f(n) \in O(n^3)$; $f(n) \notin \Omega(n^3)$ (and thus, $f(n) \notin \Theta(n^3)$).

Show $f(n) = 2^{n+1} \in \Theta(2^n)$

For $f(n) = n^2(sin(n))^2 + 50n$ show:

- $f(n) \in O(n^2)$ and $f(n) \in \Omega(n)$.



$50n \leq n^2 sin(n) \leq x^2$

O, Ω, Θ-notation

▶ $f(n) = \frac{1}{2} n^2 + 3n$ in $\Omega(n^2)$, $O(n^2)$, $\Theta(n^2)$

$f(n) \in \Omega(n^2)$: must show exist pos. const c_1, n_0 st.

$$c_1 g(n) = c_1 \cdot n^2 \leq f(n) = \frac{1}{2} n^2 + 3n \quad \forall n \geq n_0$$

$$\Leftrightarrow \quad c_1 n^2 \leq \frac{1}{2} n^2 + 3n \quad \Big| - \frac{1}{2} n^2$$

$$(c_1 - \tfrac{1}{2}) n^2 \leq 3n \quad \Big| \cdot \frac{1}{n}$$

$$(c_1 - \tfrac{1}{2}) n \leq 3$$

$$\boxed{\text{holds e.g for } c_1 = \tfrac{1}{3} \text{ \& } n \geq n_0 = 1}$$

since $\underbrace{\left(\frac{1}{3} - \frac{1}{2}\right)}_{<0} \cdot n < 0 \leq 3$

$f(n) \in O(n^2)$: must show exist pos. const c_2, n_0 st.

$$f(n) = \frac{1}{2} n^2 + 3n \leq c_2 n^2 \quad \forall n \geq n_0$$

$$\Leftrightarrow \quad \left(\frac{1}{2} - c_2\right) n^2 \leq -3n \quad \Big| \cdot \frac{1}{n^2}$$

$$\Leftrightarrow \quad \left(\frac{1}{2} - c_2\right) \leq -\frac{3}{n}$$

$$\boxed{\text{holds eg for } c_2 = 5,\ n \geq n_0 = 1}$$

since then $\left(\frac{1}{2} - 5\right) = -4.5 \leq -3 \leq -\frac{3}{n}$

$f(n) \in \Theta(n^2)$: choose $c_1 = \frac{1}{3}$, $c_2 = 5$, $n_0 = 1$

► $f(n) = \frac{1}{2}n^2 + 3n \quad \notin O(n)$

$\in \Omega(n)$

$\notin O(n)$: $\frac{1}{2}n^2 + 3n \le c \cdot n$ for some const. $c, n_0 \ge 0$ & all $n \ge n_0$?

$\frac{1}{2}n^2 \le (c-3)\cdot n \quad \Big| \cdot \frac{2}{n}$

$n \le 2c - 6$

there is always an $n > 2c-6$ $\forall n \ge n_0 \ge 0$ since c is fixed ↯

$\in \Omega(n)$: $\frac{1}{2}n^2 + 3n \ge cn$

$\frac{1}{2}n^2 + 3n \ge \frac{1}{2}n^2 \ge cn \quad \Big| \cdot \frac{2}{n}$

$n > 2c$

choose $c = 1$, $n_0 = 2$

$\Rightarrow f(n) \ge cn \quad \forall\ n \ge n_0 = 2,\ c = 1.$

▶ $f(n) = \frac{1}{2}n^2 + 3n$ $\quad \in O(n^3)$

$\notin \Omega(n^3)$

$\in O(n^3)$: $\quad \frac{1}{2}n^2 + 3n \le cn^3$

$$0 \le cn^3 - \frac{1}{2}n^2 - 3n$$

$$= n(cn^2 - \frac{1}{2}n - 3) \quad n \ge n_0 \ge 0$$

$\Rightarrow$ when is $cn^2 - \frac{1}{2}n - 3 = 0$?

$$n_{1,2} = \frac{\frac{1}{2} \pm \sqrt{\frac{1}{4} + 4\cdot 3\cdot c}}{2c}$$ for any $c > 0$, say $c = 1$

$$\overset{=}{c=1} \frac{\frac{1}{2} \pm \sqrt{12,25}}{2} = \frac{0,5 \pm 3,5}{2}$$

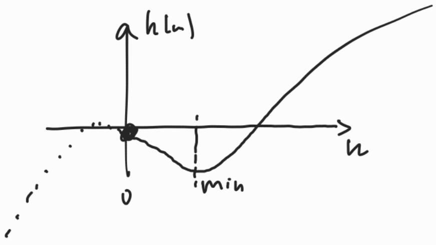
Since $n_0 \ge 0$ $\Rightarrow$ $n_0 = 2$ $\Rightarrow$ $f(n) \in O(n^3)$

$\notin \Omega(n^3)$: $\quad c\cdot n^3 \le \frac{1}{2}n^2 + 3n \quad | \cdot \frac{1}{n^3}$

$\Leftrightarrow$ $h(n) = cn^3 - \frac{1}{2}n^2 - 3n \le 0$

Sketch (no official proof, but idea)

$\forall c > 0$:



that is there is no n_0 st for all $n \ge n_0$ it holds $h(n) \le 0$ $\Leftrightarrow$ $cn^3 \le \frac{1}{2}n^2 + 3n$.

▶ $2^{n+1} \in \Theta(2^n)$?

$2^{n+1} = 2 \cdot 2^n \quad \Rightarrow \quad 2 \cdot 2^n \le 2^{n+1} \le 2 \cdot 2^n$

$\Rightarrow$ choose $c = 2$, $n_0 = 1$

▶ $f(n) = n^2 (\sin(n))^2 + 50n \in \begin{matrix} O(n^2) \\ \Omega(n) \end{matrix}$

$\sin(n) \le 1 \quad \Rightarrow \quad f(n) \le \underbrace{n^2 + 50n \le cn^2}_{}$

$\Leftrightarrow \quad 50 \le (c-1)n = c'n$

$50 \le c' n \quad$ for $c' = 1$, $n = 50$

$\Rightarrow \quad f(n) \in O(n^2)$

$n^2 (\sin(n))^2 \ge 0 \quad \Rightarrow \quad f(n) \ge 50n \quad \forall\, n \ge 1 \quad \Rightarrow \quad f(n) \in \Omega(n).$

Insertion-sort revisited:

best-case: $T(n) = 6n - 5 \xRightarrow{\text{Exerc.}} T(n) \in \Omega(n)$

worst-case: $T(n) = 2n^2 - 1 \xRightarrow{\text{Exerc.}} T(n) \in O(n^2)$

Thus, the running time of insertion-sort is in $O(n^2)$, that is, no matter what particular input of size n is chosen, the running time on that input is always bounded from above by some function cn^2 for some constants $c, n_0 > 0$ and all $n \geq n_0$.

At the same time, the running time of insertion-sort is in $\Omega(n)$, that is, no matter what particular input of size n is chosen, the running time on that input is at least cn, for some constants $c, n_0 > 0$ and all $n \geq n_0$.

Moreover, these bounds are asymptotically as tight as possible:
The running time of insertion-sort is not $\Omega(n^2)$, since there exists an input for which insertion sort runs in $\Theta(n)$ time (e.g., when the input is already sorted).

It is not contradictory, however, to say that the *worst-case* running time of insertion sort is $\Omega(n^2)$ since there exists an input that causes the algorithm to take $\Omega(n^2)$ time.

Let $f(n)$ and $g(n)$ be asymptotically positive functions.

Then, for $\Upsilon \in \{O, \Omega, \Theta\}$, it holds that

- $f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity] proof *board*
- $f(n) \in \Upsilon(f(n))$ [reflexivity] proof exercise

Moreover, it holds that

- $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry] proof exercise
- $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry] proof exercise

Transitivity : $f(n) \in O(g(n))$ & $g(n) \in O(h(n))$

$\Rightarrow f(n) \in O(h(n))$.

proof: $f(n) \in O(g(n)) \Rightarrow \exists\, c, n_0$ st $f(n) \leq c \cdot g(n) \;\; \forall n \geq n_0$ I

$g(n) \in O(h(n)) \Rightarrow \exists\, c', n_0'$ st $g(n) \leq c' \cdot h(n) \;\; \forall n \geq n_0'$ II

let $N_0 = \max\{n_0, n_0'\}$, $C = \max\{c, c'\}$

I: $f(n) \leq c \cdot g(n) \leq C\, g(n) \quad \forall \;\; n \geq N_0 \geq n_0$

II: $g(n) \leq c' \cdot h(n) \leq C \cdot h(n) \quad \forall \;\; n \geq N_0 \geq n_0'$

I & II: $f(n) \leq C\, g(n) \leq C\,(C\, h(n)) = C^2\, h(n) = \underbrace{K}_{\text{pos. const}}\, h(n)$

$\forall\, n \geq N_0$

$\Rightarrow f(n) \in O(h(n))$.

(similar arguments for Ω & Θ)

□

$O(\dots)$ (rt=runtime)	typical framework	typical examples
$O(1)$ constant rt	`a=b+c` // `if(a<b)`	assignments, in/output, 32/64bit-arithmetic, cases
$O(\log n)$ logarithmic rt	`while(N>1) N = N/2`	binary search
$O(n)$ linear rt	`for(i=0; i<n; i++){...}`	loop find the maximum

$O(n^2)$ quadratic rt	`for(i=0; i<n; i++)`
	` for(j=0; j<n; j++) {...}`
		double loop,
		check all pairs

$O(n^3)$ cubic rt	`for(i=0; i<n; i++)`
	` for(j=0; j<n; j++)`
	`  for(k=0; k<n; k++) {...}`
		triple loop,
		check all triples

$O(2^n)$ exponential rt	`see combinatorial lecture;)`	exhaustive search check all subsets

Instead of $f(n) \in O(g(n))$ one often writes $f(n) = O(g(n))$ (similar for Ω, Θ)

This is sometimes convenient when establishing certain estimations or calculations.

IMPORTANT NOT !!! $O(g(n)) = f(n)$

Example
$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$ means that there is some anonymous function $f(n) \in \Theta(n)$ that we do not care to name, such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.

This can help eliminate inessential detail and clutter in an equation and allows us also to write for $\Upsilon \in \{O, \Omega, \Theta\}$:

- $\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$ proof *next slides*
- $c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$ proof exercise
- $\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$ proof exercise

Proof of $O(f(n)) + O(g(n)) = O(f(n) + g(n)) = O(\max(f(n), g(n)))$:

We need to show that for any $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ it holds that

$$h(n) := \tilde{f}(n) + \tilde{g}(n) \in O(f(n) + g(n))$$

$\tilde{f}(n) \in O(f(n)) \implies \tilde{f}(n) \leq c'f(n)$ for some constants $c', n_0' > 0$ and all $n \geq n_0'$

$\tilde{g}(n) \in O(g(n)) \implies \tilde{g}(n) \leq c''g(n)$ for some constants $c'', n_0'' > 0$ and all $n \geq n_0''$

Thus, $h(n) := \tilde{f}(n) + \tilde{g}(n) \leq c'f(n) + c''g(n)$
$\leq c(f(n) + g(n))$ for all $n \geq n_0$ with $c = \max\{c', c''\}$ and $n_0 = \max\{n_0', n_0''\}$

Hence, $h(n) \in O(f(n) + g(n))$.

Since $\tilde{f}(n) \in O(f(n))$ and $\tilde{g}(n) \in O(g(n))$ have been arbitrarily chosen, we have

$$O(f(n)) + O(g(n)) = O(f(n) + g(n)).$$

Let $h(n) \in O(f(n) + g(n))$.

$\implies h(n) \leq c(f(n) + g(n))$ for some constants $c, n_0 > 0$ and all $n \geq n_0$

$\implies h(n) \leq c \cdot 2 \cdot \max\{f(n), g(n)\}$

$\implies h(n) \leq \tilde{c} \cdot \max\{f(n), g(n)\}$ with $\tilde{c} = 2c$

$\implies h(n) \in O(\max(f(n), g(n)))$

Since $h(n) \in O(f(n) + g(n))$ has been arbitrarily chosen we have

$$O(f(n) + g(n)) = O(\max(f(n), g(n)))$$

Exmpl: Applying $O(f(n)) + O(g(n)) = O(\max(f(n), g(n)))$ and $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$

```
Do_Smth(int $n$)
  1 PRINT "Hello World"
  2 FOR ($i = 0$ to $n - 1$) DO
  3     $i := i + 1$
  4     IF ($n$ is even) THEN RETURN 0
  5     ELSE
  6        FOR ($j = 0$ to $n - 1$) DO
  7           $j := j + 1$
```

All basic-instructions (eg. PRINT, $i = 0$, $j := j + 1$, RETURN 0, ...) in $O(1)$ time

Do_Smth consists of two main-parts:

$$A_1 = \texttt{PRINT "Hello World"} \text{ and } A_2 = \text{Line 2-7}$$

Hence, runtime of Do_Smth is in $O(1)+$ runtime A_2 $\implies$ *examine* A_2 *!*
runtime A_2 = Line 2-7

The most expensive task within the loop in Line 2 is in $O(n)$:

Line 3: $O(1)$
Line 4: $O(1) + O(1) = O(\max(1, 1)) = O(1)$
Line 5: $O(1)$
Line 6-7: $O(n) \cdot O(1) = O(n \cdot 1) = O(n)$

Line 3-7: $O(1) + O(1) + O(1) + O(n) = O(\max(1, 1, 1, n)) = O(n)$

FOR-loop in Line 2 runs n times. *runtime* A_2= $O(n) \cdot O(n) = O(n \cdot n) = O(n^2)$

Runtime Do_Smth is in $O(1)+$ *runtime* A_2 = $O(1) + O(n^2) = O(\max(1, n^2)) = O(n^2)$

Summary up to here

- $O(g(n)) := \{f(n)\colon \exists$ constants $c, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$
- $\Omega(g(n)) := \{f(n)\colon \exists$ constants $c, n_0 > 0$ such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$
- $\Theta(g(n)) := \{f(n)\colon \exists$ constants $c_1, c_2, n_0 > 0$ such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0\}$

Theorem: $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ if and only if $f(n) \in \Theta(g(n))$.

Let $f(n)$ and $g(n)$ be asymptotically positive functions and $\Upsilon \in \{O, \Omega, \Theta\}$. Then,

- $f(n) \in \Upsilon(g(n))$ and $g(n) \in \Upsilon(h(n))$ implies $f(n) \in \Upsilon(h(n))$ [transitivity]
- $f(n) \in \Upsilon(f(n))$ [reflexivity]

Moreover, it holds that

- $f(n) \in \Theta(g(n))$ if and only if $g(n) \in \Theta(f(n))$ [symmetry]
- $f(n) \in O(g(n))$ if and only if $g(n) \in \Omega(f(n))$ [transpose symmetry]

The following rules can be applied:

- $\Upsilon(f(n)) + \Upsilon(g(n)) = \Upsilon(f(n) + g(n)) = \Upsilon(\max(f(n), g(n)))$
- $c \cdot \Upsilon(f(n)) = \Upsilon(c \cdot f(n))$
- $\Upsilon(f(n)) \cdot \Upsilon(g(n)) = \Upsilon(f(n) \cdot g(n))$

It could be that some of these equations must be proven in exercises or the exam!

Further example

```
Halve(number $n$)
    WHILE ($n > 1$) DO
        $n := \frac{n}{2}$
```

$$\begin{aligned} T(n) &= \Theta(1) + T(\frac{n}{2}) \\ &= \Theta(1) + (\Theta(1) + T(\frac{n}{4})) = 2 \cdot \Theta(1) + T(\frac{n}{2^2}) \\ &= 2 \cdot \Theta(1) + (\Theta(1) + T(\frac{n}{8})) = 3 \cdot \Theta(1) + T(\frac{n}{2^3}) \\ &= \ldots \\ &= N \cdot \Theta(1) + T(\frac{n}{2^N})) \end{aligned}$$

How often can one repeat this, that is, what is N?

In other words: For which $k = \frac{n}{2^N}$ does Halve(k) terminate?
Answer: For any $k \leq 1 \iff \frac{n}{2^N} \leq 1 \iff n \leq 2^N \iff \log_2(n) \leq N$

Put $N = \log_2(n)$ and note $T(1) = \Theta(1)$:

$$\begin{aligned} T(n) &= N \cdot \Theta(1) + T(\frac{n}{2^N}) \\ &= \log_2(n) \cdot \Theta(1) + T(1) \\ &= \Theta(\log_2(n)) \cdot \Theta(1) + \Theta(1) \\ &= \Theta(\log_2(n) \cdot 1) + \Theta(1) \\ &= \Theta(\max\{\log_2(n), 1\}) = \Theta(\log_2(n)) \end{aligned}$$

Iterative vs. recursive algorithms

iterative	**recursive**

Sum(n)
$total\_sum := 0$
FOR $(i = 1$ to $n)$ DO
$total\_sum := total\_sum + i$
PRINT $total\_sum$
	Sum(int n)
	IF$(n = 1)$ THEN RETURN 1
	RETURN $n + \textsc{Sum}(n - 1)$

What are these algorithms doing? **Answer:** Sum computes the sum $\sum_{i=1}^{n} i$, where $n \geq 1$.

iterative ($n = 4$)	**recursive** ($n = 4$)
$total\_sum := 0$	`RETURN` $4 + \textsc{Sum}(3)$ (the return value of $\textsc{Sum}(4)$)
$total\_sum := 0 + 1 = 1$	`RETURN` $3 + \textsc{Sum}(2)$ (the return value of $\textsc{Sum}(3)$)
$total\_sum := 1 + 2 = 3$	`RETURN` $2 + \textsc{Sum}(1)$ (the return value of $\textsc{Sum}(2)$)
$total\_sum := 3 + 3 = 6$	`RETURN` 1 (the return value of $\textsc{Sum}(1)$) $[\textsc{Sum}(1) = 1]$
$total\_sum := 6 + 4 = 10$	$\implies 2 + \textsc{Sum}(1) = 2 + 1 = 3$ $[\textsc{Sum}(2) = 3]$
	$\implies 3 + \textsc{Sum}(2) = 3 + 3 = 6$ $[\textsc{Sum}(3) = 6]$
	$\implies 4 + \textsc{Sum}(3) = 4 + 6 = 10$ $[\textsc{Sum}(4) = 10]$

Runtime iterative $\textsc{Sum}$: $\Theta(n)$ [Exercise]

Runtime recursive $\textsc{Sum}$:

$$\begin{aligned} T(n) &= \Theta(1) + T(n-1) \\ &= \Theta(1) + (\Theta(1) + T(n-2) = 2 \cdot \Theta(1) + T(n-2) \\ &= \dots \\ &= (n-1)\Theta(1) + T(1) \in \Theta(n) \text{ since } T(1) \in \Theta(1) \end{aligned}$$

Often recurrences come in the form

$$T(n) = aT(n/b) + f(n)$$

with constants $a \geq 1$ and $b > 1$.

$n \in \mathbb{N}_{\geq 1}$ is the input size

a is the number of subproblems in the recursion

n/b is the size of a single subproblem and means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$ *

$f(n)$ denotes the cost incurred by dividing the problem and combining the partial solutions

Here, we assume that $f(n) = \Theta(n^d)$ for some $d \geq 0$.

Theorem. ***Master Theorem*** *[simplified version]*
Let $a \geq 1$, $b > 1$ and $d \geq 0$ be constants and $n \in \mathbb{N}_{\geq 1}$. If $T(n) = aT(n/b) + \Theta(n^d)$, then

$$T(n) = \begin{cases} \Theta(n^d) & \textit{if } a < b^d \\ \Theta(n^d \log_2 n) & \textit{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \textit{if } a > b^d \end{cases}$$

proof: For simplicity write $n^d = \Theta(n^d)$

$$\begin{aligned} T(n) &= aT(\tfrac{n}{b}) + n^d \quad \text{//use formular for input of size } n/b \\ &= a[aT(\tfrac{n}{b^2}) + (\tfrac{n}{b})^d] + n^d = a^2T(\tfrac{n}{b^2}) + a(\tfrac{n}{b})^d + n^d \quad \text{//use formular for input of size } n/b^2 \\ &= a^2[aT(\tfrac{n}{b^3}) + (\tfrac{n}{b^2})^d] + a(\tfrac{n}{b})^d + n^d = a^3T(\tfrac{n}{b^3}) + a^2(\tfrac{n}{b^2})^d + a(\tfrac{n}{b})^d + n^d \\ &= \dots \\ &= a^kT(\tfrac{n}{b^k}) + \textstyle\sum_{j=0}^{k-1} a^j \left(\tfrac{n}{b^{\,j}}\right)^d \quad \text{//by induction (exercise)} \\ &= a^kT(\tfrac{n}{b^k}) + n^d \textstyle\sum_{j=0}^{k-1} \left(\tfrac{a}{b^d}\right)^j \end{aligned}$$

We have $T(n) = a^kT(\frac{n}{b^k}) + n^d \sum_{j=0}^{k-1} \left(\frac{a}{b^d}\right)^j$ for all $k \geq 1$

[*] $\lceil x \rceil$ denotes the least integer greater than or equal to x
$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.

$T(1)$ terminates.

Hence, it terminates for k for which $T(1) = T(\frac{n}{b^k})$ holds

$\iff 1 = \frac{n}{b^k} \iff b^k = n \iff k = \log_b(n)$

Hence, we can write: $T(n) = a^{\log_b(n)} T(\frac{n}{b^{\log_b(n)}}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $T(\frac{n}{b^{\log_b(n)}}) = T(1)$ we have: $T(n) = \Theta(a^{\log_b(n)}) \quad + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

Since $a^{\log_b(n)} = n^{\log_b(a)}$ *(exercise!)*, we have: $T(n) = \Theta(n^{\log_b(a)}) \quad + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

We have $T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

We consider now the three cases: $a < b^d$, $a = b^d$ and $a > b^d$

Case $a < b^d$: $\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \geq \left(\frac{a}{b^d}\right)^0 = 1 \implies \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Omega(1)$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \leq \sum_{j=0}^{\infty} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. series}}{=} \frac{1}{1-\frac{a}{b^d}} = O(1)$ //since $\frac{a}{b^d} \in (0,1)$ and constant

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \in \Theta(1)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j$

$= \Theta(n^{\log_b(a)}) + n^d \Theta(1)$

$= \Theta(n^{\log_b(a)} + n^d)$

Since $a < b^d$ we have $\log_b(a) \leq \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d + n^d) = \Theta(n^d)$ as desired

Case $a = b^d$: $\iff \frac{a}{b^d} = 1$

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j = \sum_{j=0}^{\log_b(n)-1} 1 = \log_b(n)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \log_b(n)$

Since $a = b^d$ we have $\log_b(a) = \log_b(b^d) = d$ and therefore, $T(n) = \Theta(n^d) + n^d \log_b(n) = \Theta(n^d \log_b(n))$

Since $\log_2(n) = \frac{\log_b(n)}{\log_b(2)}$ and $\log_b(2)$ is constant, we have $\Theta(\log_b(n)) = \Theta(\log_2(n))$ and thus,

$T(n) = \Theta(n^d \log_2(n))$ as desired

Case $a > b^d$:

$\sum_{j=0}^{\log_b(n)-1} \left(\frac{a}{b^d}\right)^j \overset{\text{geom. sum}}{=} \frac{\left(\frac{a}{b^d}\right)^{\log_b(n)-1} - 1}{\frac{a}{b^d} - 1} \overset{\text{exerc.}}{=} \Theta\left(\left(\frac{a}{b^d}\right)^{\log_b(n)}\right) \overset{\text{exerc.(log-rules)}}{=} \Theta\left(\frac{n \log_b(a)}{n^d}\right)$

$T(n) = \Theta(n^{\log_b(a)}) + n^d \Theta\left(\frac{n \log_b(a)}{n^d}\right) = \Theta(n^{\log_b(a)})$ as desired □

Example:

```
Some_Rec($n$)
    IF ($n > 1$) THEN
      someTask
      Some_Rec($n/3$) + Some_Rec($n/3$)
```

Suppose someTask has runtime in $\Theta(n^5)$

$\implies T(n) = 2T(n/3) + \Theta(n^5)$, $a = 2, b = 3, d = 5$

$\implies a < b^d \implies T(n) = \Theta(n^5)$

Example:

```
Halve($n$)
    IF ($n > 1$) THEN Halve($n/2$)
```

Runtime without Master Theorem: $O(\log_2(n))$ (similar arguments as for Halve above with WHILE-loop)

With Master Theorem: $T(n) = T(n/2) + \Theta(1)$

$\implies a = 1,\ b = 2,\ d = 0$

In formula above: $1 = a = b^d = 2^0$ and thus, runtime is in $\Theta(n^d \log_2 n) = \Theta(n^0 \log_2 n) = \Theta(\log_2 n)$

Further examples: Assume that $d = 2$ and $b = 3$:

$a = 8$: $T(n) = 8T(\frac{n}{3}) + \Theta(n^2) \xRightarrow{8<3^2} T(n) = \Theta(n^2)$

$a = 9$: $T(n) = 9T(\frac{n}{3}) + \Theta(n^2) \xRightarrow{9=3^2} T(n) = \Theta(n^2 \log_2 n)$

$a = 10$: $T(n) = 10T(\frac{n}{3}) + \Theta(n^2) \xRightarrow{10>3^2} T(n) = \Theta(n^{log_3(10)})$

Space complexity is a measure of the amount of working storage an algorithm needs and is also often expressed asymptotically in big-O, Big-Ω, Big-Θ notation.

```
Some_Sum(int $x$, $y$, $z$)
    int $r = x + y + z$
    RETURN $r$
```

Requires 3 units of space for the parameters x, y, z and 1 for the local variable r.

Space complexity is in $O(1)$

```
Sum(array $a$ of length $n$)
    int $r = 0$
    FOR ($i = 1$ to $n$) DO $r := r + a[i]$
    RETURN $r$
```

Requires n units of space for array a and 2 for the local variables r and i.

Space complexity is in $O(n)$

```
Fact_iter(int $n$)
    int fac $= 1$
    FOR ($i = 1$ to $n$) DO
        fac := fac $\cdot\, i$
    RETURN fac
```

requires 3 space units for the variables n, fac and i

$\implies O(1)$ space.

```
Fact_rec(int $n$)
    IF($n == 0$ or $n == 1$) THEN
        RETURN 1
    ELSE
        RETURN $n \cdot$ Fact_rec($n - 1$)
```

requires 1 space units for the variable n

Now, examine the extra space that is taken by the algorithm temporarily to finish its work [auxiliary space]:

for n the return-value Fact_rec($n-1$) must temporarily be stored
for $n-1$ the return-value Fact_rec($n-2$) must temporarily be stored

$\vdots$

for 2 the return-value Fact_rec(1) must temporarily be stored
for 1 the return-value is 1

At this point the values can be used to compute Fact_rec(n) and we temporarily stored $n - 1 = O(n)$ variables.

$\implies O(n)$ space

But be careful here: If things are passed by pointer or reference, then space is shared [later].

We mainly focus here on time complexity

Side Note:

During each time step, you can only access one memory location. Therefore you can never access more memory locations than you have time

$\implies$ space complexity is bounded by time complexity

Does runtime matter?

	insertion-sort	merge-sort *[later]*
runtime	$O(n^2)$	$O(n\log_2(n))$

Should we care about factor n vs $\log_2(n)$?

For large enough n and constant c, we have

	insertion-sort	merge-sort
runtime	$c \cdot n^2$	$c \cdot n\log_2(n)$

Say $c = 100$ and $n = 10^7$ (e.g. list of population in Sweden). Suppose we have a computer that can perform $10^9 op/s$ where op/s = operations per seconds.

insertion-sort: $\frac{100\cdot(10^7)^2 op}{10^9 op/s} = \frac{10^{16} op}{10^9 op/s} = 10^7 s \qquad \approx 115$ days

merge-sort: $\frac{100\cdot 10^7 \cdot \log_2(10^7) op}{10^9 op/s} = \frac{10^9 \cdot \log_2(10^7) op}{10^9 op/s} = \log_2(10^7) s \qquad \approx 24s$

Runtime matters !

1.4 Elementary Data Structures

- The right organizational form and choice of data structure significantly impact the efficiency of data operations.

- Example:

  Consider a phone book. There it is easy to find a phone number for a given name based on the alphabetical order.

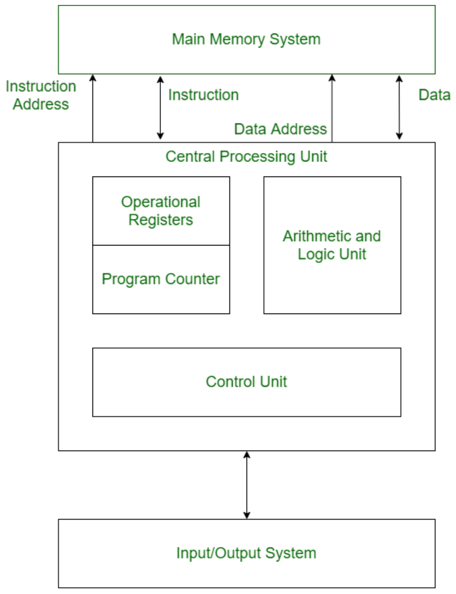
  What if we are interested in the reverse task (finding for a given number the name)? Ideas?

- The optimal choice of a data structure is not always obvious and one data structure might be very suitable for one task but not for some other

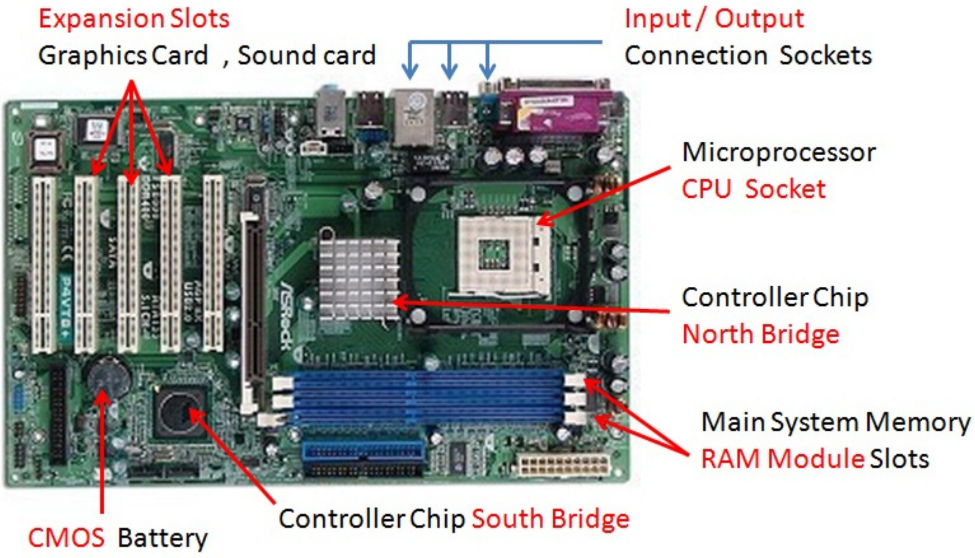
- Determining an efficient data structure is usually influenced by the operations needed later on the data (searching, replacing, re-sorting, . . .)

1.4.1 The idea of memory allocation

How does memory work and how is this related to Data Structures?

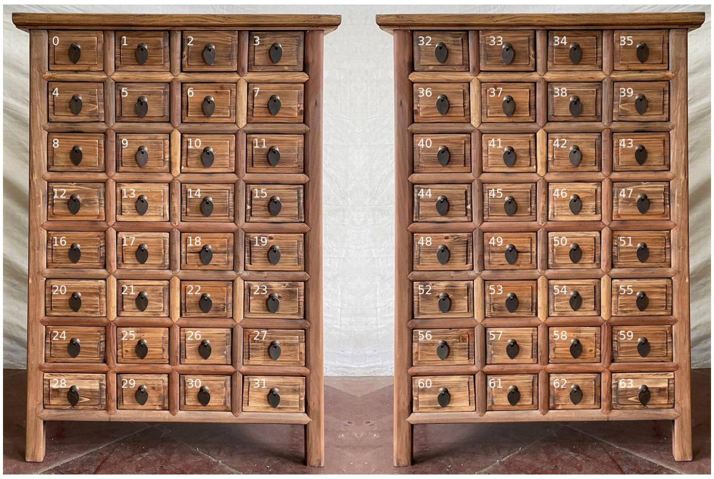


Harvard Architecture

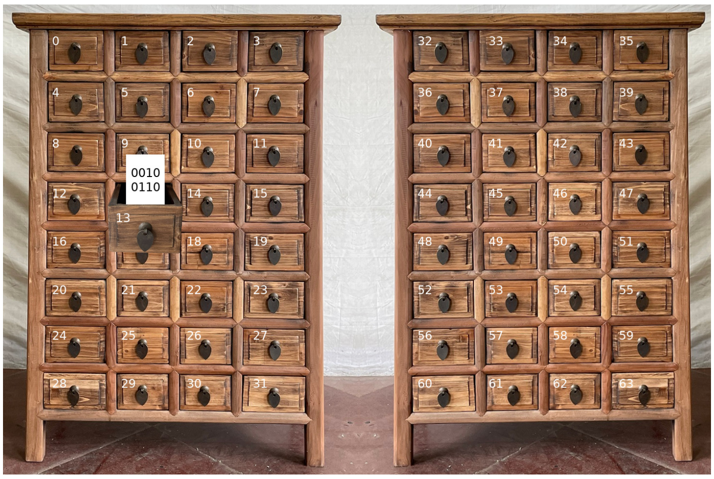


mainboard of a computer

main memory or also RAM = Random Access Memory

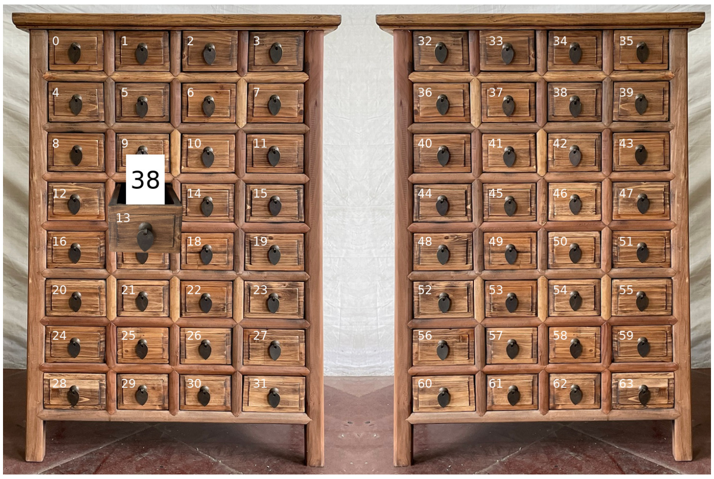


Main memory consists of a number of regularly arranged memory cells,
comparable to the compartments of a cabinet.
Since memory cells are regularly arranged, they can be numbered consecutively.
Each cell therefore has a unique number (=address).

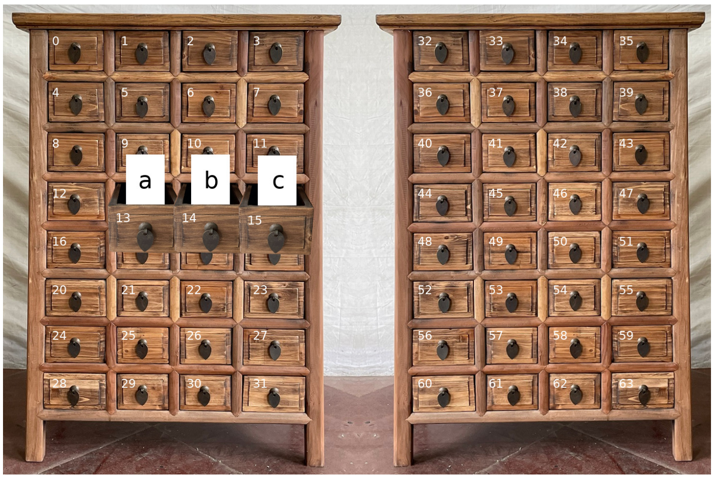


All memory cells are the same size and can store a value (number, character, . . .).
This value is a fixed-length sequence of 0s and 1s (e.g. 1byte = 8 bits)

[8 bit per cell is pure convention (a few exceptions exist)].



The value stored in a cell represents some information (e.g. a number or a character)



But also "longer" information can be stored using "chunks of cells"

(e.g., the first 8bits of a 32bit integer n [to store n we need then 4 cells each of size 1byte]

or the first 3 characters of the alphabet)

Difference between a 32-bit and a 64-bit architecture? n-bit architecure means that CPU can handle data in chunks of n-bit at a time. Thus, n-bit computer can can process data and perform calculations on numbers that are n-bits long.

32-bit system that can access 2^{32} (or 4,294,967,296) bytes of RAM. Meanwhile, a 64-bit processor can handle 2^{64} (or 18,446,744,073,709,551,616) bytes of RAM. In other words, a 64-bit processor can process more data than 4 billion 32-bit processors combined.

The value can also be the *address of another memory cell*. In this case, we refer to it as a **pointer**.

A variable in (compiled) source-code refers to one or more consecutive cells in memory that store the "value/information" we assigned to this variable.

Variables can thus contain values or be pointers to another variable.

Storage of information in different languages (here as example `C` / `Python`)

Memory `C`

Adr.	.	
	.	
	.	
6	–	
7	10	x
8	10	y
	.	
	.	
80	7	px
	.	

```
int x // init integer variable x
int y // init integer variable y
x = 5 // assign 5 to x
y = 10 // assign 5 to y
printf("%i", x) // prints "5"
printf("%i", y) // prints "10"
printf("%p", &x) // prints "7" (address of x)
x = y
printf("%i", x) // prints "10"
printf("%p", &x) // prints "7" (address of x)
int *px; // init pointer px that point to some integer variable
px = &x;
printf("%p", px); // prints "7" (content of px)
printf("%i", *px); // prints "10" (content of content of px [Dereference])
```

many famous games are based on game engines written in C/C++ (Fortnite, GTA, DOOM, Civilization,. . .)

Memory "somewhat similar to what `Python` does"

Adr.	.	
6	–	
7	22	x
8	21	y
	.	
	.	
21	42	42
22	10	10
	.	

```
x = 5 // init cell for x and 5 and x contains address of 5
y = 10 // init cell for y and 10 and y contains address of 10
print(x) // prints "5"
print(y) // prints "10"
x=y // let x "point to" address that y "points to"
As "5" is no longer used, memory cell 21 is freed up [garbage collector].
print(x) // prints "10"
y=42 // y contains address of 42
print(x) // prints "10"
print(y) // prints "42"
```

In `python` there are lot of secrets in the memory allocation that cannot directly be handled by user and a lot of vodoo (incl. garbage collection) takes control about the latter

Pointer = variable p that stores address of another memory cell containing information about "some object x".

in symbols "$p \rightarrow x$"

Data structures can be classified as either contiguous or linked, depending upon whether they are based on arrays or pointers:

- **Contiguously-allocated structures** are composed of single slabs of memory, and include arrays, matrices, heaps, and hash tables.
- **Linked data structures** are composed of distinct chunks of memory bound together by pointers, and include lists, trees, and graph adjacency lists.

1.4.2 Array

The **array** is the fundamental contiguously-allocated data structure. Arrays are structures of fixed-size data records such that each element can be efficiently located by its index (or address).

Analogy/Example:

Init new array L of length 3 [=allocate 3 consecutive cells (here the ones with address 13,14,15)]
and put $L[1] = a$, $L[2] = b$, $L[3] = c$

Advantages:

- Constant-time access given the index

  Because the index of each element maps directly to a particular memory address, we can access arbitrary data items instantly provided we know the index.

- Space efficiency

  Arrays consist purely of data, so no space is wasted with links or other formatting information. Further, end-of-record information is not needed because arrays are built from fixed-size records.

Disadvantages:

- Fixed size and content

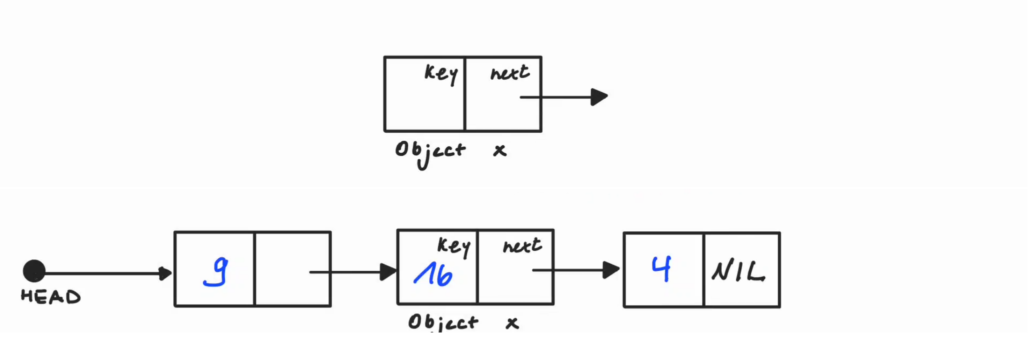
  An array can only save one type of data (e.g. only integer, or only bool, . . .)

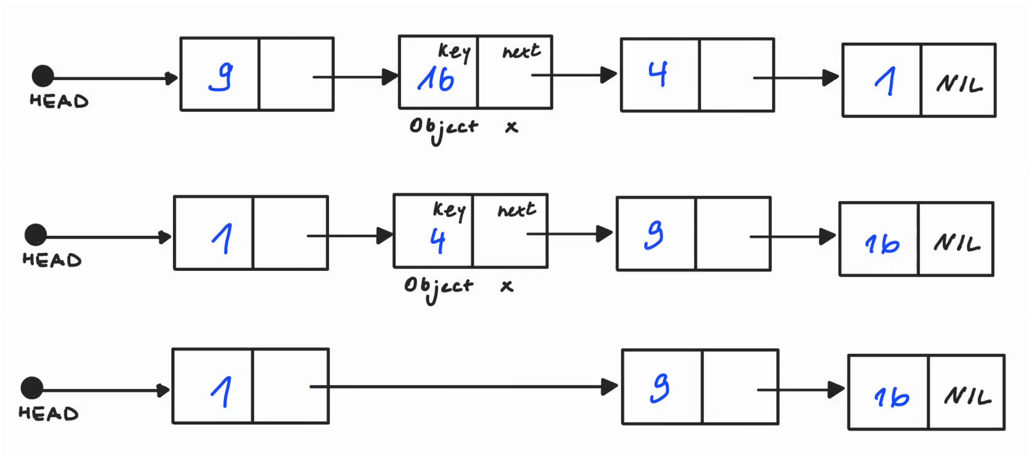
  One cannot adjust the size of an array in the middle of a program's execution

  Our program will fail soon as we try to add an $(n + 1)$-entry if only space for n records was allocated (= overflow). This can be compensated by allocating extremely large arrays, but this can waste space.

1.4.3 Linked Lists

A **(single) linked list** is a data structure in which the elements are arranged in a linear order.
Each list element is an object with an attribute key (data) and one pointer: next.
Last element points to NIL. Head points to first element.





x.next points to its successor in the linked list
x.key is the data stored in object x (here x.key = 16)

Unlike an array in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

The list elements can be scattered arbitrarily throughout the memory; in particular, it is no longer necessary to preallocate a region of sufficient size to accommodate all list elements.

Instead, the occupied memory space dynamically adjusts to the current size of the list. However, one needs memory space not only for the list elements themselves but also for the pointers.

Such linked lists can be used to realize "dynamic sets" (here the set $\{1, 4, 9, 16\}$)
Suppose a linked list L and an array A is sorted:

How easy is it in L, resp., A to remove an object such that L, resp., A stays sorted?

$A = [1, 4, 9, 16]$ and remove $A[1] = 4$:

$A[1] = A[2]$, $A[2] = A[3]$, $A[3]$ =fantasy number "42" stating "$A[3]$ is not in use"

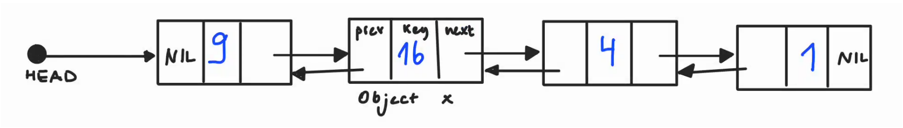
$\implies \Theta(|A|)$ time

$L = [1, 4, 9, 16]$ remove x (say the one with $x.key = 4$ and assume we know the predecessor x' of x)

$x'.next = x.next$

$\implies \Theta(|1|)$ time

Keeping track of predecessor can be done more efficiently with **doubly linked list**:
Each list element is an object with an attribute key (data) and two pointers: next, prev.



Advantages:

- New elements can be placed anywhere in memory and added in constant time before or after a given element by changing the pointers.

Disadvantages:

- When searching for an element, you have to go through the list from the first (or last) element to the respective position.

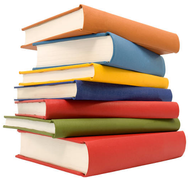
  Searching in (sorted) lists L only takes $O(|L|)$ time.

1.4.4 Queues and Stacks

Stacks and **queues** are dynamic sets in which the position of elements inserted / removed from the set is prespecified by a particular order [can be realized by using e.g. single-linked lists].

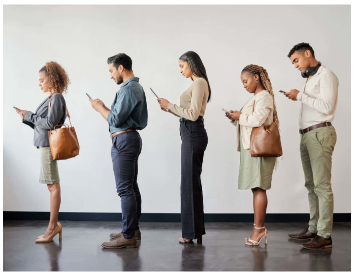
Stacks follow LIFO = last-in, first-out

- S.push(x): Inserts item x at the top (last item) of stack S.
- S.pop(): Removes the top item of stack S.
- S.top(): Returns the top item of stack S.
- S: $|2| \to |6| \to |7|$
S.push(3): $|2| \to |6| \to |7| \to |3|$
S.top(): returns 3
S.pop(): modifies S to $|2| \to |6| \to |7|$



Queues follow FIFO = first in, first out

- Q.enqueue(x): Inserts item x at the back (last item) of queue Q.
- Q.dequeue(): Removes the first item from queue Q.
- Q.front(): Returns the first item from queue Q.
- Q: $|2| \to |6| \to |7|$
Q.enqueue(3): $|2| \to |6| \to |7| \to |3|$
Q.front(): returns 2
Q.dequeue(): modifies Q to $|6| \to |7| \to |3|$



1.4.5 Trees

Trees form a more general framework than linked list and are defined as "special graphs".

Let us start with the formal definition first.

The next slide contains a lot of definitions that we also need later on (e.g. for heaps, binary search trees, AVL trees, ...). Most of these defs refer Sec B4 and B5 in the Cormen et al. course-book.

BUT don't be afraid, they are easy to grasp: STEP-BY-STEP and stay with me!
[for all we give examples - board!!!]

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

A path (of length k) is a sequence $P = (v_0, v_1, \dots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$, $0 \leq i < k$. $P = (v_0, v_1, \dots, v_k)$ is also called v_0v_k-path and said to connect v_0 and v_k.

A path P is simple if the vertices $v_0, v_1, \dots, v_k$ are pairwise distinct. Note that $P = (v_0)$ is a simple path of length 0.

A simple cycle $C = (v_0, v_1, \dots, v_k, v_0)$ of length $k+1$ is defined by:

$$P = (v_0, v_1, \dots, v_k) \text{ is a simple path of length } k \geq 2 \text{ and } \{v_k, v_0\} \in E.$$

A graph G is connected if for any two vertices $x, y \in V(G)$ there is an xy-path.

Graphs without simple cycles are called acyclic or forest.

A connected acyclic graph is a tree.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree.*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

If $x \preceq_T y$, then y is an ancestor of x and x a descendant of y

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

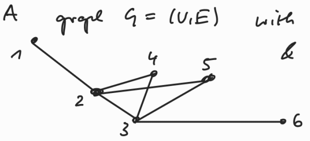
A vertex in T without any children is a leaf. A vertex that has a child is an internal or inner vertex.

2 vertices with the same parent are siblings.

A rooted tree is ordered if for every vertex v its children are ordered.

BASICS: Graphs & trees (part 1)

A graph $G = (V,E)$ with $V = \{1,2,3,4,5,6\}$ & $E = \{\{1,2\}, \{2,3\}, \{2,4\}, \{2,5\}, \{3,4\}, \{3,5\}, \{3,6\}\}$



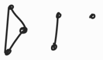
(connected)

path $P = (1,2,4,3,2,5)$ connecting 1 & 5 of length 5 [not simple]

simple path e.g $P = (1,2,5)$ length 2 (# edges)

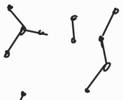
cycle $P = (2,3,4,2)$

G not connected

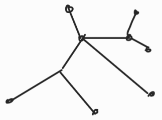


contain cycles

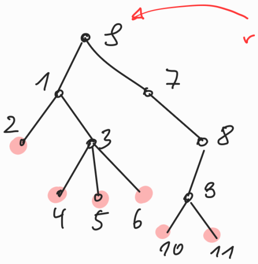
acyclic/forest



tree



rooted tree:



root usually drawn on top!

$5 \leq_T 1$

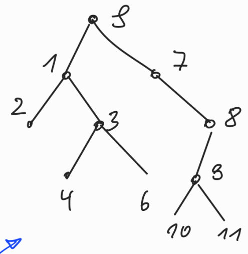
1 is parent of 3, 3 is child of 1

leaves highlighted ●

2 & 3 are siblings

binary tree

T



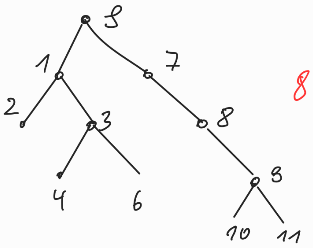
8 has a left child (9) but no right child.

drawing implies order on children, e.g,

„ 4 left of 6 "

" 11 right of 10 "

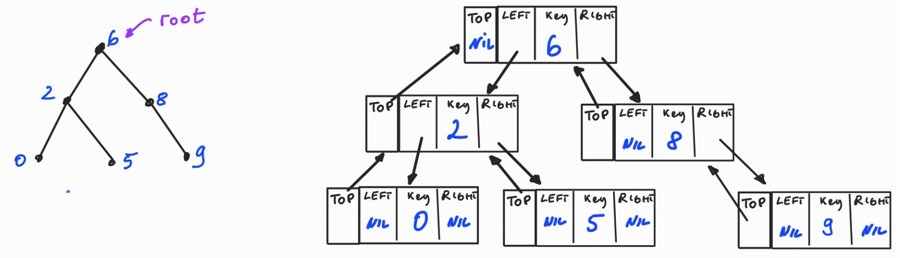
T'



8 has right child but no left child

⇒ T & T' are different trees!!

A rooted tree T is **binary** if each vertex as *at most* two children. If T is ordered and binary, then there is a clear distinction between right and left child (even if a vertex has only child).



Advantages:

- New elements can be placed anywhere in memory and added in constant time before or after a given element by changing the pointers.
- Searching in a sorted tree takes $O(h)$ time, with h = height of tree (=longest simple path from root to some leaf).

  In so-called "balanced trees" $h \in O(logn)$ where n = number of vertex (key/data) stored in T *[details in upcoming lectures]*

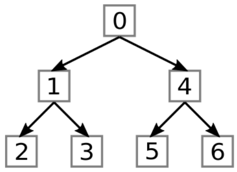
Disadvantages:

- Searching in "non-balanced" tree $O(|n|)$ time (as in linked-lists)
- Making a non-balanced tree to a balanced one gets tricky (in particular, insertion of elements is more complicated)

Traversal of trees (more details in upcoming lectures).

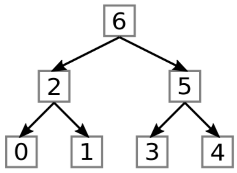
***Pre*order:**

1. *visit current vertex*
2. recursively traverse left subtree
3. recursively traverse right subtree



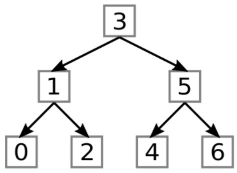
***Post*order:**

1. recursively traverse left subtree
2. recursively traverse right subtree
3. *visit current vertex*



***In*order:**

1. recursively traverse left subtree
2. *visit current vertex*
3. recursively traverse right subtree



numbers in squares =
order in which nodes are visited

Plenty of other data structures exist and we will examine some of them later in the course

Chapter 2

Sorting

- Investigations by computer manufacturers and users have shown for many years that more than a quarter of commercially consumed computing time is dedicated to sorting operations.
- It is therefore not surprising that significant efforts have been made to develop the most efficient procedures for sorting data using computers.
- The accumulated knowledge about sorting algorithms now fills volumes. Still, new insights into sorting continue to appear in scientific journals, and numerous theoretically and practically important problems related to the task of sorting a set of data remain unresolved.

Given: A sequence of integers $(a_1, a_2 \ldots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

Example: $A = (5, 2, 2, 4, 6)$ should become $(2, 2, 4, 5, 6)$

- In practice, the numbers to be sorted are rarely isolated values.
- We usually deal with a collection of data called a record.

  Each record contains a key, which is the value to be sorted.

  The remainder of the record consists of satellite data, which are usually carried around with the key.
- In practice, when a sorting algorithm permutes the keys, it must permute the satellite data as well. If each record includes a large amount of satellite data, we often permute an array of pointers to the records rather than the records themselves in order to minimize data movement.

Example:

key (birth year)	sat-data (name)	sat-data (living town)
2000	Max	Linköping
1980	Anna	Uppsala
1988	Paula	Stockholm

key (birth year)	sat-data (name)	sat-data (living town)
1980	Anna	Uppsala
1988	Paula	Stockholm
2000	Max	Linköping

key = birth year

Example:

sat-data (birth year)	*key (name)*	sat-data (living town)
2000	Max	Linköping
1980	Anna	Uppsala
1988	Paula	Stockholm

sat-data (birth year)	*key (name)*	sat-data (living town)
1980	Anna	Uppsala
2000	Max	Linköping
1988	Paula	Stockholm

key = name

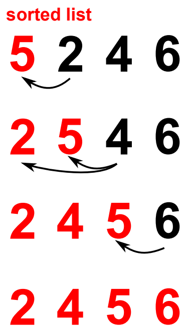
To understand the principles of the basic sorting algorithms we focus here mainly on sequences of integers only.

2.1 Insertion-Sort (Revisited)

Given: A sequence of integers $(a_1, a_2 \dots, a_n)$
Goal: A re-ordering $(a'_1, a'_2 \dots, a'_n)$ such that $a'_1 \leq a'_2 \leq \dots \leq a'_n$

Example: $A = (5, 2, 2, 4, 6)$ should become $(2, 2, 4, 5, 6)$

Recap (`Insertion_Sort`):



A simple sorting "algorithm" idea:

We assume to have an order list (*highlighted in red*)

Then, subsequently insert the next element x into this sorted list by comparing x with the elements in sorted list from right to left

We put this into an algorithm, known as `Insertion_Sort`.

`Insertion_Sort` sorts the input numbers **in place**: it rearranges the numbers within the array A, with at most a constant number of them stored outside the array at any time.

In the upcoming part, we introduce three more sorting algorithms (assuming n numbers need to be sorted):

- `Merge Sort`: not in place, but runtime $\Theta(n \log n)$
- `Heapsort`: in place, runtime $\Theta(n \log n)$
- `Quicksort`: in place, but worst-case runtime $\Theta(n^2)$. However, its expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort

The latter algorithms (incl. insertion sort) are all *comparison sorts*: they determine the sorted order of an input array by comparing elements.

We then continue with proving a lower bound of $\Omega(n \log n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

This lower bound can be improved if one adds additional requirements on the input data and thus, if one can gather information about the sorted order of the input by means other than comparing elements. As an example, we consider

- `Counting Sort`: not in place, runtime $\Theta(n + k)$ in case the numbers to be sorted
  are in $\{1, \dots, k\} \implies \Theta(n)$ if $k = O(n)$.

Some of these algorithm use a classical divide-and-conquer approach that is based on the following three steps:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

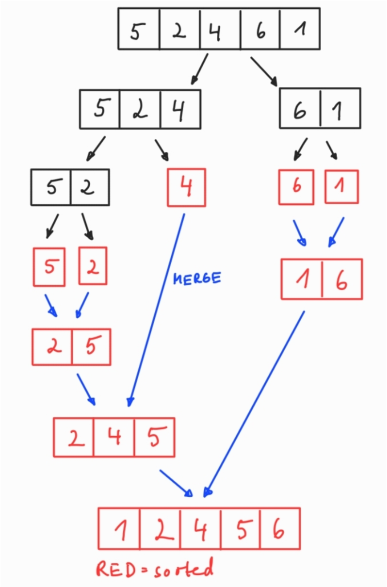
Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

Combine the solutions to the subproblems into the solution for the original problem.

2.2 Merge-Sort

Merge sort is a classical example of divide-and-conquer approaches. The divide-and-conquer paradigm involves three steps at each level of the recursion:

Main Idea:



Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

merge sort: Divide the n-element sequence to be sorted into two subsequences of approx. $n/2$ elements each

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

merge sort: Sort the two subsequences recursively using merge sort

Combine the solutions to the subproblems into the solution for the original problem.

merge sort: Merge the two sorted subsequences to produce the sorted answer

The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine" step.

The key idea of the "merging-operation" is the merging of two sorted sequences which is done by calling an auxiliary procedure

$$\texttt{MERGE}(A, p, q, r)$$

where A is an array and p, q, r integers such that $p \leq q < r$.

The procedure assumes that the subarrays $A[p..q]$ and $A[q+1..r]$ are sorted and merges them to form a single sorted subarray that replaces the current subarray $A[p..r]$

Exmpl: $A[3..4] = (\cancel{2}2,\ \cancel{4}4)$ and $A[5..7] = (\cancel{1}1,\ \cancel{2}2,\ \cancel{7}7)$
take smallest (first elements) of $A[3..4]$ and $A[5..8]$ and put the smaller one to list and repeat with next smallest elements:

$$1\ ,\ 2\ ,\ 2\ ,\ 4\ ,\ 7$$

Each comparison: $\Theta(1)$ time

Thus, merging takes $\Theta(r-p+1)$ time since we have in total $r-p+1$ comparisons.

```
MERGE($A, p, q, r$)                                   //1st entry of array $M$ is $M[1]$
 1 $n_1 := q - p + 1$                                          //length of $A[p..q]$
 2 $n_2 := r - q$                                            //length of $A[q+1..r]$
 3 Init new arrays $L[1..n_1+1]$ and $R[1..n_2+1]$
 4 FOR ($i = 1$ to $n_1$) DO $L[i] := A[p+i-1]$           //"copy" $A[p..q]$ to $L[1..n_1]$
 5 FOR ($j = 1$ to $n_2$) DO $R[j] := A[q+j]$         //"copy" $A[q+1..r]$ to $R[1..n_2]$
 6 $L[n_1+1] := \infty$ and $R[n_2+1] := \infty$ //to avoid: "array index out of bounds"
   in L9, 12 and to further increment $i$, resp, $j$ when $L$, resp., $R$ has been copied
 7 $i := 1$ and $j := 1$, $k := p$
 8 WHILE ($k \leq r$) DO                  //"merge" elements, while runs from $k = p..r$
 9    IF($L[i] \leq R[j]$) THEN
10       $A[k] := L[i]$
11       $i := i + 1$
12    ELSE
13       $A[k] := R[j]$
14       $j := j + 1$
15    $k := k + 1$
```

Lemma. MERGE(A, p, q, r) *correctly merges the sorted arrays* $A[p..q]$ *and* $A[q+1..r]$ *into the sorted array* $A[p..r]$ *in* $\Theta(r-p+1)$ *time.*

Show correctness: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Line 1+2: computes the length n_1 of $A[p..q]$ and n_2 of $A[q+1..r]$.

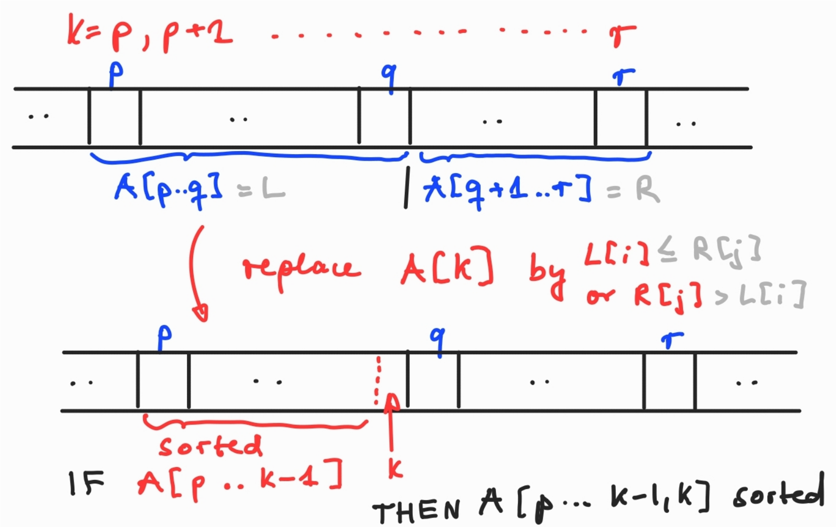
Line 3-5: Copy " $A[p..q]$ to $L[1..n_1]$ and $A[q+1..r]$ to $R[1..n_2]$

note that L and R are sorted

Loop-invariant: At the start of each iteration of the while-loop (L 8–14), it holds

[I] $A[p..k-1]$ contains the $k-p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order and

[II] $L[i], R[j]$ are the smallest elements of their arrays that have not been copied back into A.


k= p, p+1 . . . . r
p
q
r
. .
. .
. .
. .
A[p..q] = L
A[q+1..r] = R
replace A[k] by L[i] ≤ R[j]
or R[j] > L[i]
p
q
r
sorted
IF A[p .. k-1]
k
THEN A[p ... k-1, k] sorted

[I] & *[II]* hold when initializing first run of while-loop:

$k = p$ (just k initialized in L7, no run of while-loop so-far) $\Rightarrow A[p..p-1]$ "empty" and thus has $0 = k - p$ elements $\Rightarrow$ *[I]* holds

Since $i = 1$, $j = 1$ and $A[p..k-1]$ empty so-far and since L, R are sorted $\Rightarrow$ *[II]* holds

[I] & *[II]* are maintained when running the while-loop:

Two Cases: $L[i] \leq R[j]$ or $L[i] > R[j]$. Assume that $L[i] \leq R[j]$ in L9

"By induction": $L[i]$ is the smallest element not yet copied back into A (*[I]* holds) and in L10 $L[i]$ is copied back to A ($A[k] \coloneqq L[i]$)

This with $L[i] \leq R[j]$ and the fact that, by *[I]*, $A[p..k-1]$ contains the $k - p$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ implies that

$A[p..k]$ contains now the $k - p + 1$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$

Incrementing k (in L15) and i (in L11) restablishes the loop invariant for the next iteration.

If instead $L[i] > R[j]$ then L13-14 perform the appropriate action to maintain the loop invariant.

Since *[I]* & *[II]* hold in each step, we have at termination of while-loop:

After termination, we have $k = r + 1$

By the loop-invariant, $A[p..k-1] = A[p..r]$ contains the $k - p = r - p + 1$ smallest elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order

Since $L[1..n_1]$ and $R[1..n_2]$ have together $n_1 + n_2 = r - p + 1$ elements

$\implies A[p..k-1] = A[p..r]$ contains the $r - p + 1$ smallest elements and thus ALL elements of $L[1..n_1]$ and $R[1..n_2]$ in sorted order.

Since $L[1..n_1]$ and $R[1..n_2]$ contain all elements of $A[p..r] \implies A[p..r]$ is now sorted.

$\implies$ MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$

Show Runtime: Let $N = n_1 + n_2$. It suffices to use N and this is, in particular, helpful when analyzing merge-sort

L1,2,6,7,9-15: each $\Theta(1)$

L3-5: $\Theta(n_1) + \Theta(n_2) = \Theta(n_1 + n_2) = \Theta(N)$.

While-loop in L8 runs $r - p = (n_2 + q) - (-n_1 + q - 1) = n_1 + n_2 + 1 = \Theta(N)$ times (for latter equation see L1,2)

Since all tasks within this while-loop take constant time we have runtime $\Theta(N)$ for L8-14.

$\implies$ overall runtime $\Theta(N)$.

In summary, we have shown: MERGE(A, p, q, r) correctly merges the sorted arrays $A[p..q]$ and $A[q+1..r]$ into the sorted array $A[p..r]$ in $\Theta(N)$ time with N being the sum of the length of $A[q+1..r]$ and $A[p..r]$. □

We can now use the MERGE procedure as a subroutine in the merge sort algorithm MERGE_SORT(A, p, r).

2 Cases:

$p \geq r$: Then $A[p..r]$ has 0 or 1 elements and is thus sorted

$p < r$: Then $A[p..r]$ has $N \geq 2$ elements.

In this case, we compute an index q that subdivides $A[p..r]$ into two arrays:

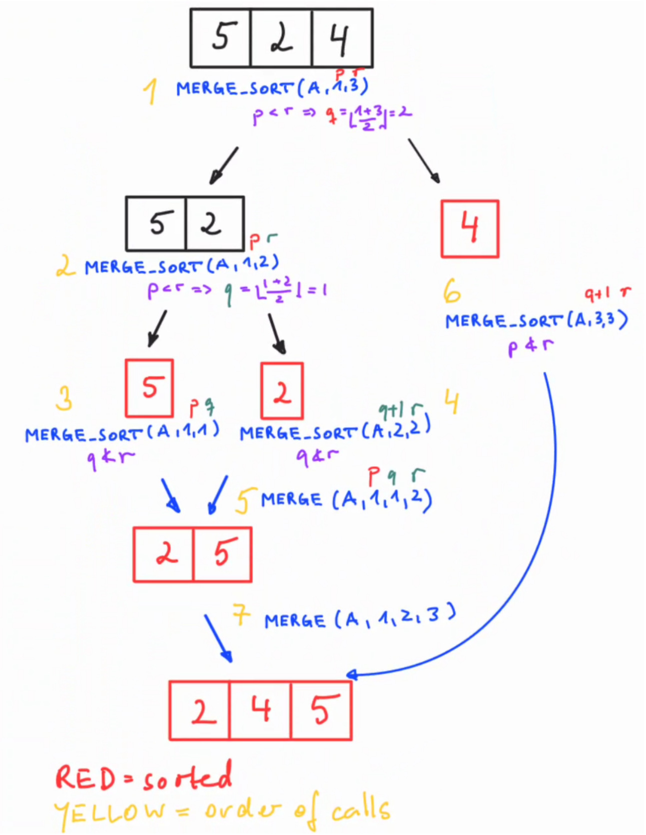
$A[p..q]$ of size $\lceil N/2 \rceil$ and $A[q+1..r]$ of size $\lfloor N/2 \rfloor$

```
MERGE_SORT$(A, p, r)$
    IF$(p < r)$ THEN
        $q = \lfloor (p+r)/2 \rfloor$
        MERGE_SORT$(A, p, q)$
        MERGE_SORT$(A, q+1, r)$
        MERGE$(A, p, q, r)$
```

Start algorithm by calling MERGE_SORT$(A, 1, A.length)$

$\lceil x \rceil$ denotes the least integer greater than or equal to x

$\lfloor x \rfloor$ denotes the greatest integer less than or equal to x.



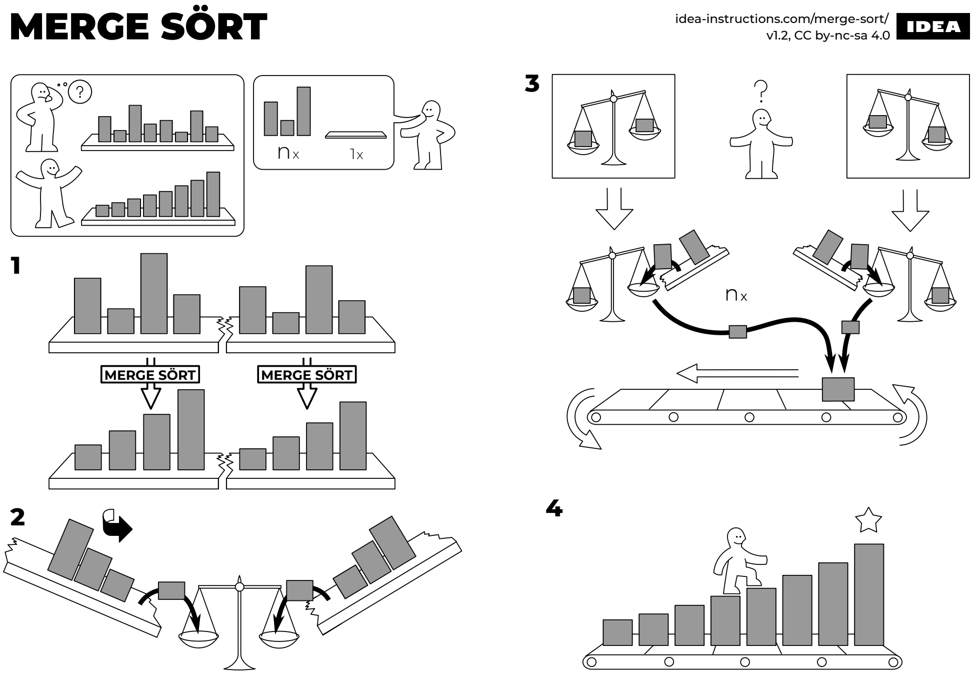
Theorem. MERGE_SORT$(A, 1, A.length)$ *correctly sorts the array* A *in* $\Theta(n \log_2(n))$ *time.*

Correctness: by previous arguments.

Runtime:
$T(n) = 2T(n/2) + \Theta(n^1)$, i.e., $a = b = 2$, $d = 1 \xRightarrow{2=2^1} \Theta(n \log_2 n)$
We can achieve runtime by using the Master theorem:
If $T(n) = aT(n/b) + \Theta(n^d)$ with constants $a \geq 1$ and $b > 1$, then

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log_2 n) & \text{if } a = b^d \\ \Theta(n^{\log_b(a)}) & \text{if } a > b^d \end{cases}$$

□



2.3 Heaps and Heap-Sort

We introduce another sorting algorithm: `Heapsort`.

Like merge sort, but unlike insertion sort, heapsort's running time is in $O(n \log n)$.

Like insertion sort, but unlike merge sort, heapsort sorts in place: only a constant number of array elements are stored outside the input array at any time.

Thus, heapsort combines the better attributes of the two sorting algorithms we have already discussed.

Before starting we need some further basic introduction into some new terms: Graphs, Tree & Co *[next slides]*

To recall:

A graph $G = (V, E)$ is a tupel consisting of a vertex set $V := V(G)$ and an edge set $E(G) := E$ that is a subset of the 2-elementary subsets of V.

Theorem. *The following statements are equivalent for every graph $G = (V, E)$ (exercise):*

1. *G is a tree (per def: G is connected and acyclic).*
2. *Any two vertices in G are connected by a unique simple path.*
3. *G is connected, and $|E| = |V| - 1$.*
4. *G is acyclic, and $|E| = |V| - 1$.*

[for all definitions examples are given on the next pages!]

A tree $T = (V, E)$ is rooted if there is a distinguished vertex $\rho \in V$, called the root of T

For a rooted tree we can define a partial order $\preceq_T$ on V such $x \preceq_T y$ if y lies on the unique path from the root ρ to x.

$T(x)$ denotes the subtree of T rooted at x, i.e., the subgraph consisting of all vertices $v \in V$ that satisfy $v \preceq_T x$ and all edges between them.

If $x \preceq_T y$ and $\{x, y\} \in E$, then x is child of y and y a parent of x.

A vertex in T without any children is a leaf. A vertex that has child is an internal or inner vertex.

Given a tree $T = (V, E)$ with root ρ we use the following notation:

- depth(x) is the length of the (unique) path from ρ to $x \in V$
- Vertices are at the same level if they have the same depth
- height $h(x)$ of $x \in V$ is the length of a longest simple path from x to a leaf ℓ with $\ell \preceq_T x$
- height of $T = h(T)$ is the height of the root ρ

A rooted tree is ordered if for every vertex v its children are ordered.

A binary tree is an ordered, rooted tree for which each vertex v has at most two children and, if v has only child, then there is a clear distinction as whether this child is right or left child.

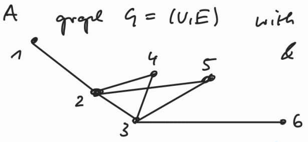
A binary tree is fully binary if each vertex is a leaf or has two children.

A binary tree is complete if all leaves have the same depth h and all inner (=non-leaf) vertices have two children.

A binary tree is nearly-complete if all vertices at depth $\leq h(T) - 2$ have to children and all leaves have depth $h(T)$ or $h(T) - 1$ and are "filled-up" from left to right, i.e., for all vertices w at depth $h(T) - 1$ it holds that if w has two children then all vertices at depth $h(T) - 1$ that are left of w have two children; if w is a leaf, then all vertices at depth $h(T) - 1$ that are right from w are leaves; there is at most one vertex w at depth $h(T) - 1$ that has only child (in which case, this child must be a left child of w).

BASICS: Graphs & trees (part2)

A graph $G = (V,E)$ with $V = \{1,2,3,4,5,6\}$
& $E = \{\{1,2\}, \{2,3\}, \{2,4\}, \{2,5\}, \{3,4\}, \{3,5\}, \{3,6\}\}$



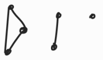
(connected)

path $P = (1,2,4,3,2,5)$ connecting 1 & 5 of length 5
[not simple]

simple path e.g $P = (1,2,5)$ length 2 (# edges)

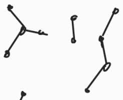
cycle $P = (2,3,4,2)$

6 not connected

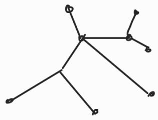


contain cycles

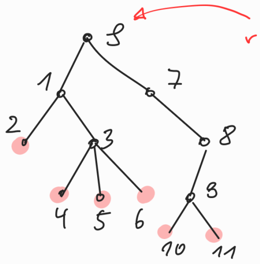
acyclic/forest



tree



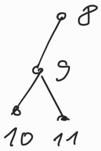
rooted tree:



root usually draw on top!

$5 \leq_T 1$

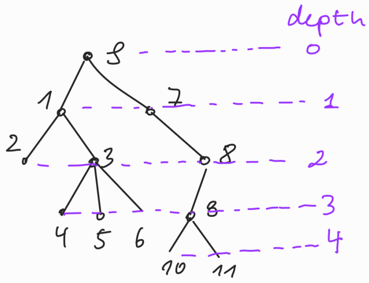
For $x = 8$: $T(x)$



1 is parent of 3, 3 is child of 1

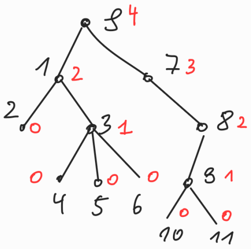
leaves highlighted ●

2 & 3 are siblings



1 & 7 on same level

height

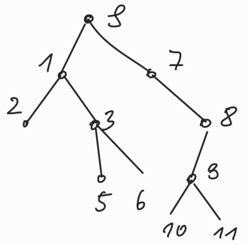


$h(T) = 4$

drawing implies order on children, e.g,

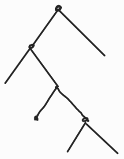
4 left of 5 left of 6
or 2 left of 3.

binary

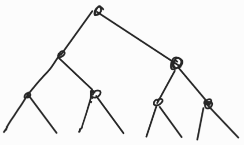


! in this drawing 8 has only one child namely the left child

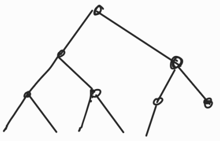
7 has a right child



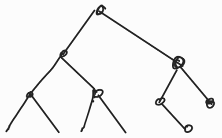
fully-binary
(not complete)



complete



nearly-complete.



not nearly-complete!

Lemma. *Let T be a binary tree with L leaves. The number of vertices in T having 2 children is $L-1$.*

Proof.
By induction along $|V(T)|$. Let B be the number of vertices in T having 2 children.

Base case: $n = 1 \implies T = (\{v\}, \emptyset) =$ "single_vertex_graph" and thus $L = 1$ and $B = 0 \implies B = L - 1 = 0$ is correct.

Assume the statement is true for all trees on $n \geq 1$ vertices.

Let T be a tree with $|V(T)| = n+1$ vertices and L leaves. [It holds that $L \geq 1$ *(exercise!)*]

Let x be a leaf and consider the tree $T - x$ from which x and the unique edge $\{w, x\}$ containing x has been removed.

Denote with L' the number of leaves in $T - x$ and with B' the number of vertices in $T - x$ having 2 children.

Since $T - x$ has n vertices we can apply the Ind-Hyp. on $T - x$.

There are two cases: In T, the parent w of x has either (a) two or (b) one children.

Case (a): In $T - x$, vertex w has still one child and is, therefore, not a leaf. Hence, $L' = L - 1$.

By Ind-Hyp: $B' = L' - 1 = L - 2$.

In T we have now exactly one vertex more than in $T - x$ that has two children, namely w.

Thus, $B = B' + 1 = L - 1$

Case (b): In $T - x$, vertex w is now a leaf. Hence, $L' = L$. Moreover, observe that $B = B'$

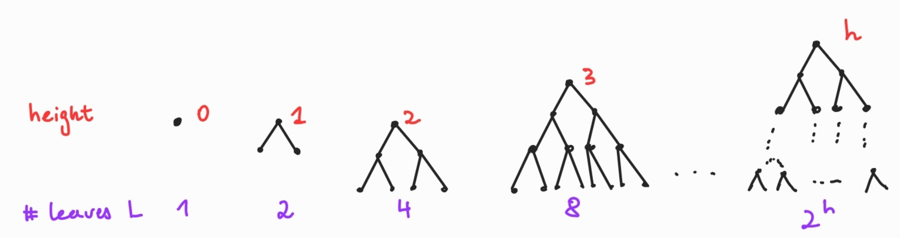
By Ind-Hyp: $B' = L' - 1 = L - 1$. Since $B = B'$, we have thus $B = L - 1$.

□

Lemma. *A complete binary tree $T = (V, E)$ has height $h(T) = \log_2(|V| + 1) - 1 \in O(\log_2(|V|))$.*

Proof. Let L be the number of leaves in T, $h := h(T)$ its height and $n = |V|$.

"Easy to verify by induction:"



$$L = 2^h.$$

$L = 2^h \iff h = \log_2(L)$.

By the previous lemma, we have $B = L - 1$ with B being the number of vertices with 2 children.

Since T is a complete binary tree its vertex set can be partitioned into leaves and vertices with two children, i.e.,

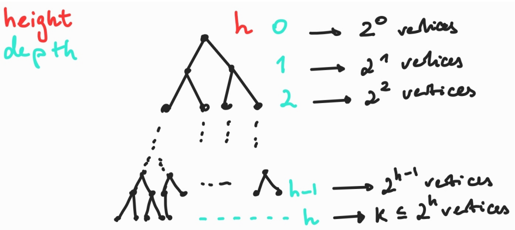
$$n = B + L = L - 1 + L = 2L - 1 \iff L = \frac{n+1}{2}$$

$\implies h = \log_2(\frac{n+1}{2}) = \log_2(n+1) - \log_2(2) = \log_2(n+1) - 1$

□

Lemma. *A nearly-complete binary tree $T = (V, E)$ has height $h(T) = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.*

Proof.



$\implies |V| \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
If only one leaf at depth h exists, then $|V| = 1 + \sum_{i=0}^{h-1} 2^i = 1 + (2^h - 1) = 2^h$.
Since at least one leaf at depths h must exist, we obtain

$$2^h \leq |V| \leq 2^{h+1} - 1 < 2^{h+1}$$

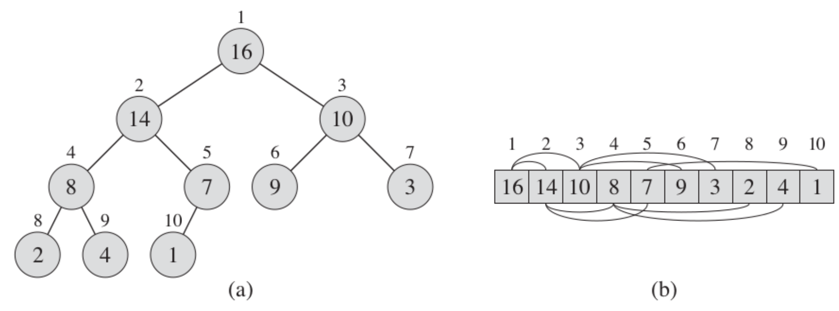
The latter is, if and only if,

$$h \leq \log_2(|V|) < h + 1$$

Since h is an integer, $h = \lfloor \log_2(|V|) \rfloor \in O(\log_2(|V|))$.

□

Heapsort uses a data structure called **(binary) heap** which is an array A with a particular structure that can be viewed as a nearly complete binary tree:



Number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children.

- 1st entry of A (i.e., $A[1]$) corresponds to value of root 1,
- Parent of i is $\lfloor i/2 \rfloor$ and children of i are $2i$ (left) and $2i + 1$ (right)
- $A.length$ = length of array
- $A.heap\_size$ = number of elements in heap that are stored in A

  That is, although $A[1..A.length]$ may contain numbers, only the elements in $A[1..A.heap\_size]$, where $0 \leq A.heap\_size \leq A.length$, are valid elements of the heap

Lemma. *If A is a heap with $n = A.heap\_size$, then the leaves in the tree representation have indices $i \in I = \{\lfloor n/2 \rfloor + 1, \ldots, n\}$.*

[proof board]

HEAPs & HEAP-SORT

Lemma: Leaves in a heap of size n have indices $\lfloor \frac{n}{2} \rfloor + 1, \dots, n$

proof: $I := \{ \lfloor \frac{n}{2} \rfloor + 1, \dots, n \}$

By contradiction, if $k \in I$ & k non-leaf
$\Rightarrow$ it has at least one child $k' \in \{2k, 2k+1\}$

since $k \geq \lfloor \frac{n}{2} \rfloor + 1$
we have $2k \geq 2\lfloor \frac{n}{2} \rfloor + 2 > n$, i.e. child is outside of heap-size ↯

$\Rightarrow \quad \forall\, i \in I : \; i$ is a leaf.

could there be leaves $i \notin I$?

let i be a leaf.

since i has no children $\Rightarrow$ $2i, 2i+1$ are not part of tree

& in particular of heap

$\Rightarrow \; 2i, 2i+1 > n$

$\Rightarrow \; i > \frac{n}{2} \; \Rightarrow \; i \in I.$

$\square$

There are two kinds of binary heaps: max-heaps and min-heaps; specified by a "heap property" that must be satisfied by the values stored at each vertex.

Heap property

- max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$
- min-heap: for every node i other than the root it holds that $A[parent(i)] \leq A[i]$

For heapsort we use max-heaps (see Example in figure).

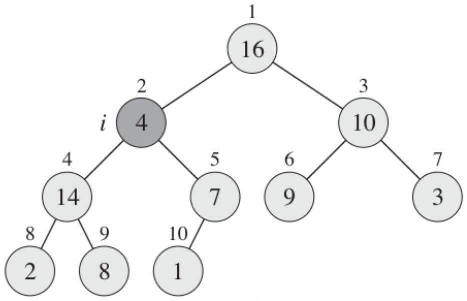
max-heap: for every node i other than the root it holds that $A[parent(i)] \geq A[i]$

Assume we have a binary heap (=nearly complete binary tree) and an index i such that the binary trees rooted at

$$left(i) = 2i \text{ and } right(i) = 2i + 1 \text{ are max-heaps, but that } A[i] < A[2i] \text{ or } A[i] < A[2i + 1]$$

$\Longrightarrow$ i violates the max-heap property.

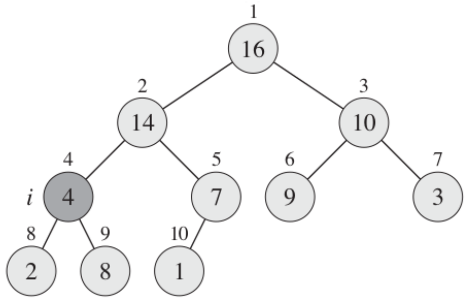
```
Max-Heapify(A, i)
$l := 2i$ and $r := 2i + 1$
$largest := i$
IF ($l \leq A.heap\_size$ and $A[l] > A[largest]$) THEN $largest := l$
IF ($r \leq A.heap\_size$ and $A[r] > A[largest]$) THEN $largest := r$
IF ($largest \neq i$) THEN
    exchange $A[i]$ with $A[largest]$
    Max-Heapify(A, largest)
```



Here, $A.heap\_size = 10$

call Max-Heapify$(A, 2)$

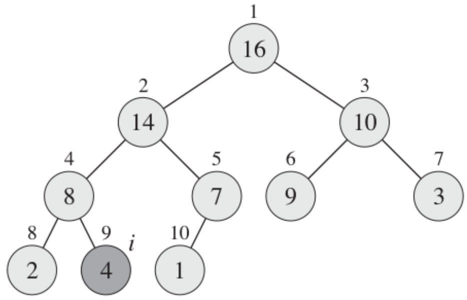
```
1   $l := 4$, $r := 5$,
2   $largest := 2$
3   $A[4] = 14 > A[2] = 4$
     $\implies$ $largest := 4$
4   $A[5] = 7 \not> A[4]$.
5-7 $largest \neq 2$
    exchange $A[2]$ with $A[4]$
    call Max-Heapify$(A, 4)$
```



Here, $A.heap\_size = 10$

call Max-Heapify$(A, 4)$

```
1   $l := 8$, $r := 9$,
2   $largest := 4$
3   $A[8] = 2 \not> A[4] = 4$
4   $A[9] = 8 > A[4] = 4$.
     $\implies$ $largest := 9$
5-7 $largest \neq 4$
    exchange $A[4]$ with $A[9]$
    call Max-Heapify$(A, 9)$
```



Here, $A.heap\_size = 10$

call Max-Heapify$(A, 9)$

```
1   $l := 18$, $r := 19$,
2   $largest := 9$
3   $l \not\leq A.heap\_size = 10$
4   $r \not\leq A.heap\_size = 10$
5-7 $largest = i = 9$
    Stop.
```

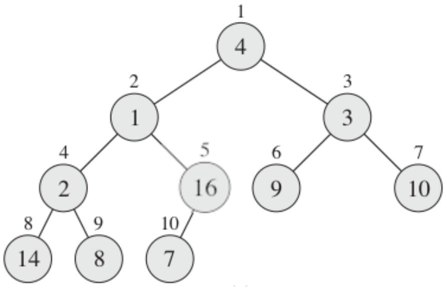
Max-Heapify lets the value at $A[i]$ "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property.

This is achieved by exchanging $A[i]$ recursively with the largest element of the children

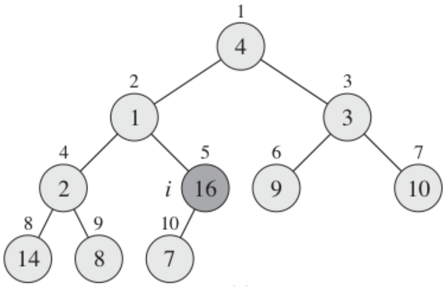
Since we always exchanged with largest child-value, the final tree $T(i)$ is a max-heap.

Runtime?
Comparing values and exchange per call: $\Theta(1)$
# of calls: $O(\log(n))$ as $h(T) \in O(\log(n))$ or via master theorem *[exercise - see Cormen Sec 6.2]*
$\implies$ Total runtime is $O(\log(n))$
Every array A can naturally be viewed as a heap (however, A might not be a max-heap)

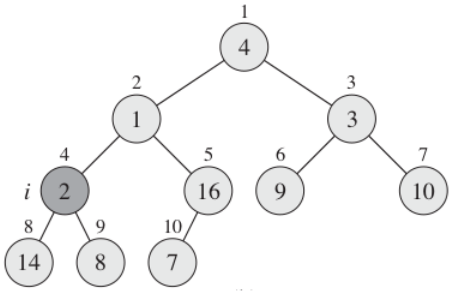
```
Build-Max-Heap($A$)

    $A.heap\_size = A.length$
    FOR ($i = \lfloor A.length/2 \rfloor$ TO 1 DO
       Max-Heapify($A$, $i$)
```



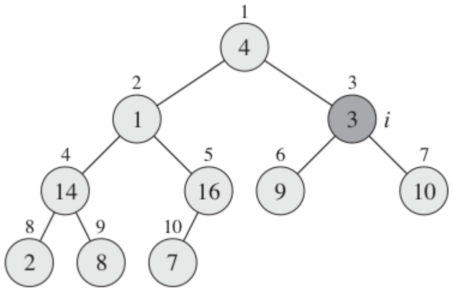
$A = [\quad 4_{1}|\ 1_{2}|\ 3_{3}|\ 2_{4}|\ 16|_{5} 9_{6}|\ 10|_{7} 14|_{8} 8_{9}|\ 7_{10}\,]$



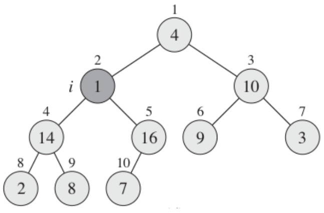
$A = [\quad 4_{1}|\ 1_{2}|\ 3_{3}|\ 2_{4}|\ 16|_{5} 9_{6}|\ 10|_{7} 14|_{8} 8_{9}|\ 7_{10}\,]$



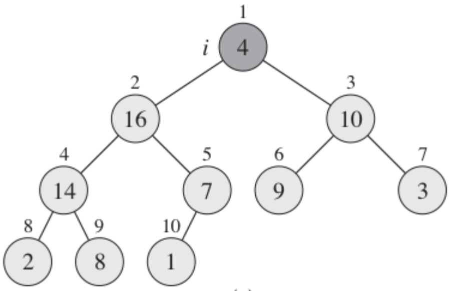
$A = [\quad 4_{1}|\ 1_{2}|\ 3_{3}|\ 2_{4}|\ 16|_{5} 9_{6}|\ 10|_{7} 14|_{8} 8_{9}|\ 7_{10}\,]$



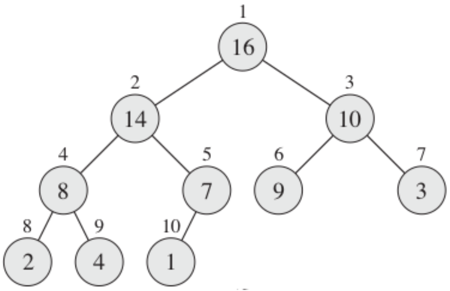
$A = [\quad 4_{1}|\ 1_{2}|\ 3_{3}|\ 14|_{4} 16|_{5} 9_{6}|\ 10|_{7} 2_{8}|\ 8_{9}|\ 7_{10}\,]$



$A:\quad 4_{1}|\ 1_{2}|\ 10|_{3} 14|_{4} 16|_{5} 9_{6}|\ 3_{7}|\ 2_{8}|\ 8_{9}|\ 7_{10}$



$A = [\ 4_1 |\ 16 |_2\ 10 |_3\ 14 |_4\ 7\ _5|\ 9\ _6|\ 3\ _7|\ 2\ _8|\ 8\ _9|\ 1\ _{10}\ ]$



$A = [\ 16 |_1\ 14 |_2\ 10 |_3\ 8\ _4|\ 7\ _5|\ 9\ _6|\ 3\ _7|\ 2\ _8|\ 4\ _9|\ 1\ _{10}\ ]$

To transform A into a max-heap, we may simply apply `Max-Heapify` bottom-up to each i violating the max-heap property.

Note, by the previous lemma, for each leaf i, $T(i)$ is a single vertex and thus, already a max-heap.

Theorem. `Build-Max-Heap`(A) *correctly transforms A into a max-heap in $O(A.heap\_size)$ time*

[proof correctness on board, proof runtime omitted - rather lengthy (ideas in Cormen Sec 6.3) - full proof 5 pages in my notes]

Theorem: BUILD-MAX-HEAP(A) correctly transforms A into a max-heap in $O(A.heapsize)$ time

PROOF

Correctedness

"loop-invariant":

At start of each iteration of the FOR loop (Line 2-3) with value i each node $i+1, i+2, \dots n$ is the root of a max-heap ($\Leftrightarrow$ $T(i+1), \dots, T(n)$ are max heaps)

prior to first run of FOR-loop ($i = \lfloor \frac{n}{2} \rfloor$) each vertex $i+1 \dots n$ is a leaf (↗ lemma above)

$\Rightarrow$ each $T(i+1) \dots T(n)$ is a single vertex tree & thus a max-heap

in FOR loop, with value i:

children of i have value $2i, 2i+1 \in \{i+1 \dots n\}$ (by definition)

$\Rightarrow$ $T(2i)$ & $T(2i+1)$ are max heaps.

This is precisely the condition that ensures that MAX-HEAPIFY(A,i) transform A to max-heap with root i

$\Rightarrow$ $T(i)$ is max-heap

Note MAX HEAPIFY(A,i) preserves the property of $T(i+1) \dots T(n)$ being max-heaps

As this holds for all i, loop-variant is maintained in each iteration of FOR-loop.

Finally the Alg. terminates after run of FOR-loop with $i = 1$ $\Rightarrow$ $T(1) \hat{=} A$ is a max-heap.

$\Rightarrow$ BUILD-MAX-HEAP(A) is correct.

RUNTIME [omitted in lecture]

each call of MAX-HEAPIFY : $O(\log(n))$ & we have $O(n)$ such calls.

$\rightsquigarrow$ $O(n \log(n))$.

this upper bound is correct but not asymptotically tight.

BETTER:

claim: Moreover in heap-size $= n$ heap, there are at most $\lceil \frac{n}{2^{h+1}} \rceil$ nodes with height h. (**)

proof of claim: N_h = # vertices at height h.

Note that height h of vertex v = longest distance from v to some leaf $x \preceq v$!

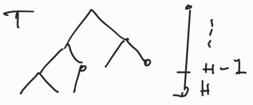
[NOT depth!]

Induction on h:

BASECASE: $h = 0$ $\Rightarrow$ N_h = # leaves in heap.

we show that $N_0 = \lceil \frac{n}{2^{0+1}} \rceil = \lceil \frac{n}{2} \rceil$

let $h_T = H$. Since T nearly complete $\Rightarrow$ all leaves are at depth H or $H-1$



let x = # leaves in depth H

Observe that T - those leaves =: T' is a complete "binary" tree & $|V(T')| = 2^{H-1} - 1$ (↗ "Graphs & Trees")

& $V(T') = n - x = 2^{H-1} - 1$ = odd number

$\Rightarrow$ if n odd then x even
n even then x odd.

We must distinguish between the cases that n is odd vs n is even.

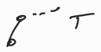
n odd $\Rightarrow$ x even $\Rightarrow$ each internal vertex has 2 children

$\Rightarrow$ # internal nodes = # leaves -1 (↗ lemma in "Graph, trees & co")

$\Rightarrow$ n = # internal nodes + # leaves
$= 2$ # leaves $- 1$

$\Rightarrow$ # leaves $= \frac{n}{2} + 1 = \lceil \frac{n}{2} \rceil$
↑ because n is odd.

n even: $\Rightarrow$ x odd $\Rightarrow$ not all internal nodes have 2 children

$\Rightarrow$ exists leaf that does not have a sibling  T

$\Rightarrow$ add add sibling $\rightsquigarrow$ T'

we have $n+1$ (odd) vertices, ie, case (a) applies.

$\Rightarrow$ $\underbrace{\text{\# leaves}}_{T'} + 1 \underset{(a)}{=} \lceil \frac{n+1}{2} \rceil = \lceil \frac{n}{2} \rceil$
↑ because n is even.

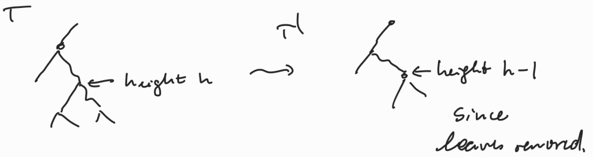
$\Rightarrow$ base case correct!

Assume claim true for $h-1 \geq 0$ & consider h

Let T' be the tree obtained from T by removing all leaves.

T has n vertices (= heap-size n)
T' has $n' = n - N_0$ vertices (N_0 = # leaves = # vertices with height 0)

$$n' = n - N_0 \underset{\text{base case}}{=} n - \left\lceil \frac{n}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$$



$\Rightarrow$ $N_h = N'_{h-1}$ = # vertices at height $h-1$ in T'

By ind. hyp.

$$N_h = N'_{h-1} \leq \left\lceil \frac{n'}{2^{(h-1)+1}} \right\rceil = \left\lceil \frac{n'}{2^h} \right\rceil = \left\lceil \frac{\left\lfloor \frac{n}{2} \right\rfloor}{2^h} \right\rceil$$

$$\leq \left\lceil \frac{\frac{n}{2}}{2^h} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$$ qed claim

To summarize:

In heap with heap-size = n, there are at most $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodes with height h.

shown in "graphs trees & co"

+ heap of heap-size = n has height $\lg(n)$ Since it is nearly complete binary tree

Time required by MAX-HEAPIFY when called with a node at height h is $O(h)$

$\Rightarrow$ runtime BUILD-MAX-HEAP can be bounded from above by

$$\sum_{h=0}^{\lg(n)} \left(\left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot \underbrace{c \cdot h}_{\substack{O(h) \\ (\text{\& constant } c)}} \right) \leq \sum_{h=0}^{\lg(n)} \left(\frac{n}{2^h} \cdot c \cdot h \right)$$

$$= c \cdot n \cdot \sum_{h=0}^{\lg(n)} \frac{h}{2^h}$$

[NOTE: (Analysis) $\sum_{k=0}^{\infty} k \cdot x^k = \frac{x}{(1-x)^2}$ for $|x| < 1$

$x = \frac{1}{2}$ $\Rightarrow$ $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$]

$$\leq c \cdot n \sum_{h=0}^{\infty} \frac{h}{2^h} = 2 \cdot c \cdot n = O(n)$$ □

proof RUNTIME.

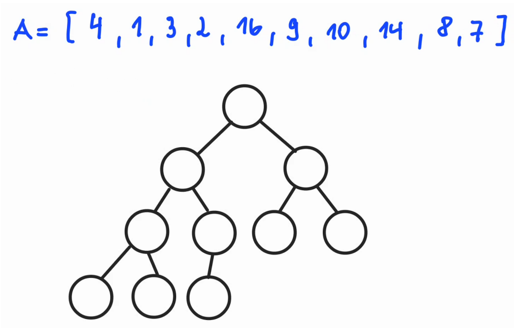
Aim: Order elements of a given array $A = A[1..n]$ such that $A[1] \leq A[2] \leq \cdots \leq A[n]$.

Heapsort Alg. Idea:

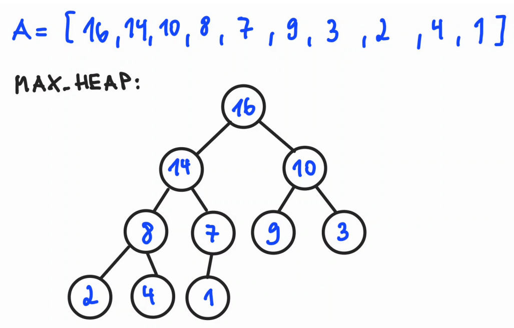
1 Transform A to a max-heap
2 By definition , if A is a max-heap, then $A[1]$ is the largest element in A (it is stored at the root)
3 Exchange $A[1]$ with $A[n]$, now the largest element of $A[1..n]$ is in the correct position n
4 Goto Step 1 with $A[1..n-1]$ “playing the role” of A and repeat until all elements in A have been processed

We can realize the latter idea as follows:

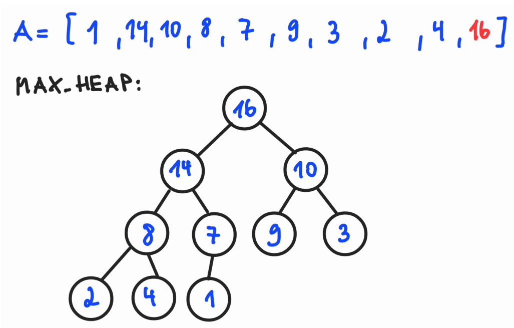
```
Heapsort($A$)

Build-Max-Heap($A$)
FOR $i = A.length$ TO 2 DO
    Exchange $A[1]$ with $A[i]$
    $A.heap\_size := A.heap\_size - 1$
    Max-Heapify($A$, 1)
```



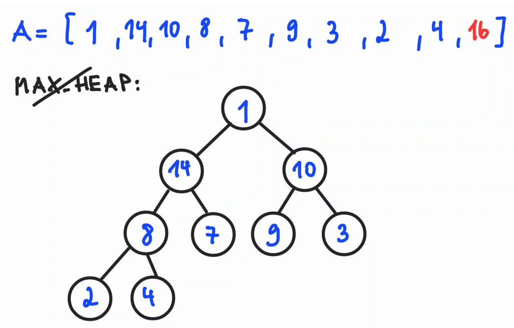
(1)



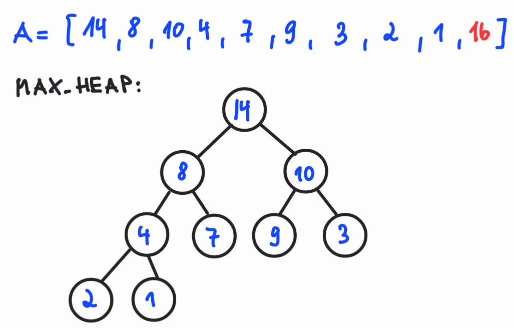
(2)



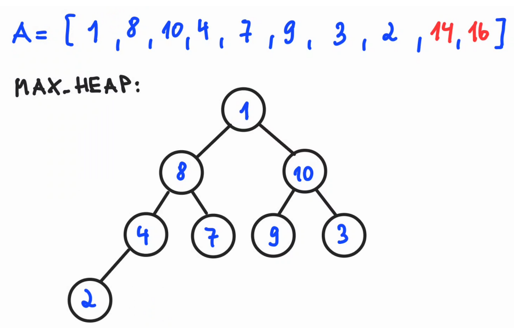
(3)



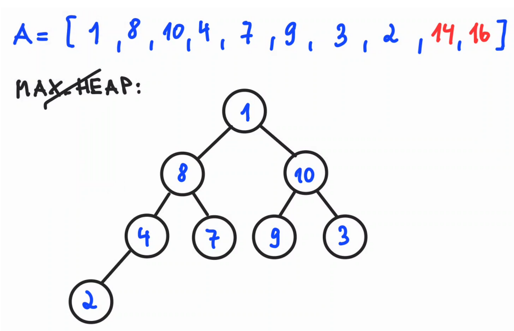
(4)



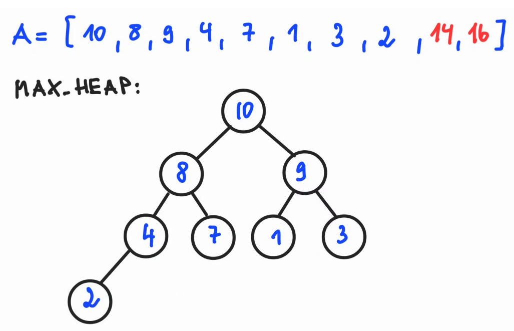
(5)



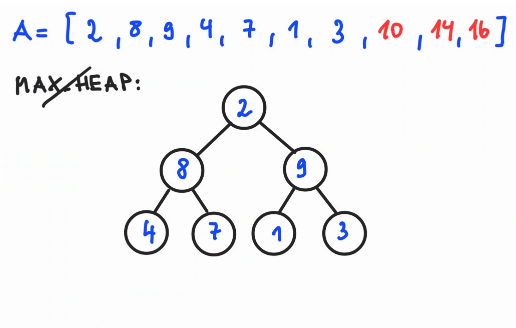
(6)



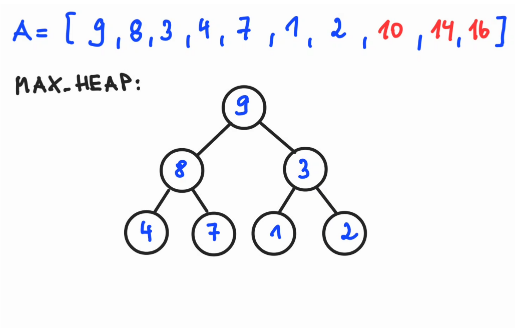
(7)



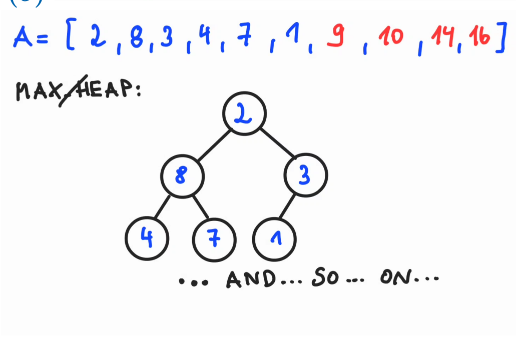
(8)



(9)



(10)



(11)

Theorem. Heapsort(A) *correctly orders* A *in* $O(n\log(n))$ *time* ($n = A.length$)

Proof: By the latter arguments, Heapsort is correct. Runtime: $(n-1)$ calls of Max-Heapify each takes $O(\log(n))$ time. □

2.4 Quick-Sort

Quicksort: in place sorting, but worst-case runtime $\Theta(n^2)$. However, its expected runtime is $\Theta(n\log n)$ and in practice it outperforms heapsort.

```
Quicksort($A$, $p$, $r$) //$A[1]$=first entry of $A$
IF ($p < r$) THEN
    $q$ = Partition($A, p, r$)
    Quicksort($A$, $p$, $q-1$)
    Quicksort($A$, $q+1$, $r$)
```

The key to Quicksort is the Partition procedure, which rearranges the subarray $A[p..r]$ in place.

To sort an entire array A, the initial call is

Quicksort(A, 1, $A.length$)

We call $A[p..q-1]$ the **low side** and $A[q+1..r]$ the **high side**

```
Partition($A$, $p$, $r$)
$x = A[r]$ //pivot (other ways to pick $x$ are possible!)
$i = p-1$ //$i$ is highest index of low side
FOR $j = p$ TO $r-1$ DO
//process each element other than pivot
  IF ($A[j] \leq x$) THEN
  //does this element belong on the low side?
    $i := i+1$ //index of a new slot in the low side
    exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
//pivot goes just to the right of the low side
RETURN $i+1$
```

Like merge sort, quicksort applies the divide-and-conquer paradigm. Here is the three-step divide-and-conquer process for sorting a typical subarray $A[p..r]$:

Divide the problem into a number of (non-overlapping) subproblems that are smaller instances of the same problem.

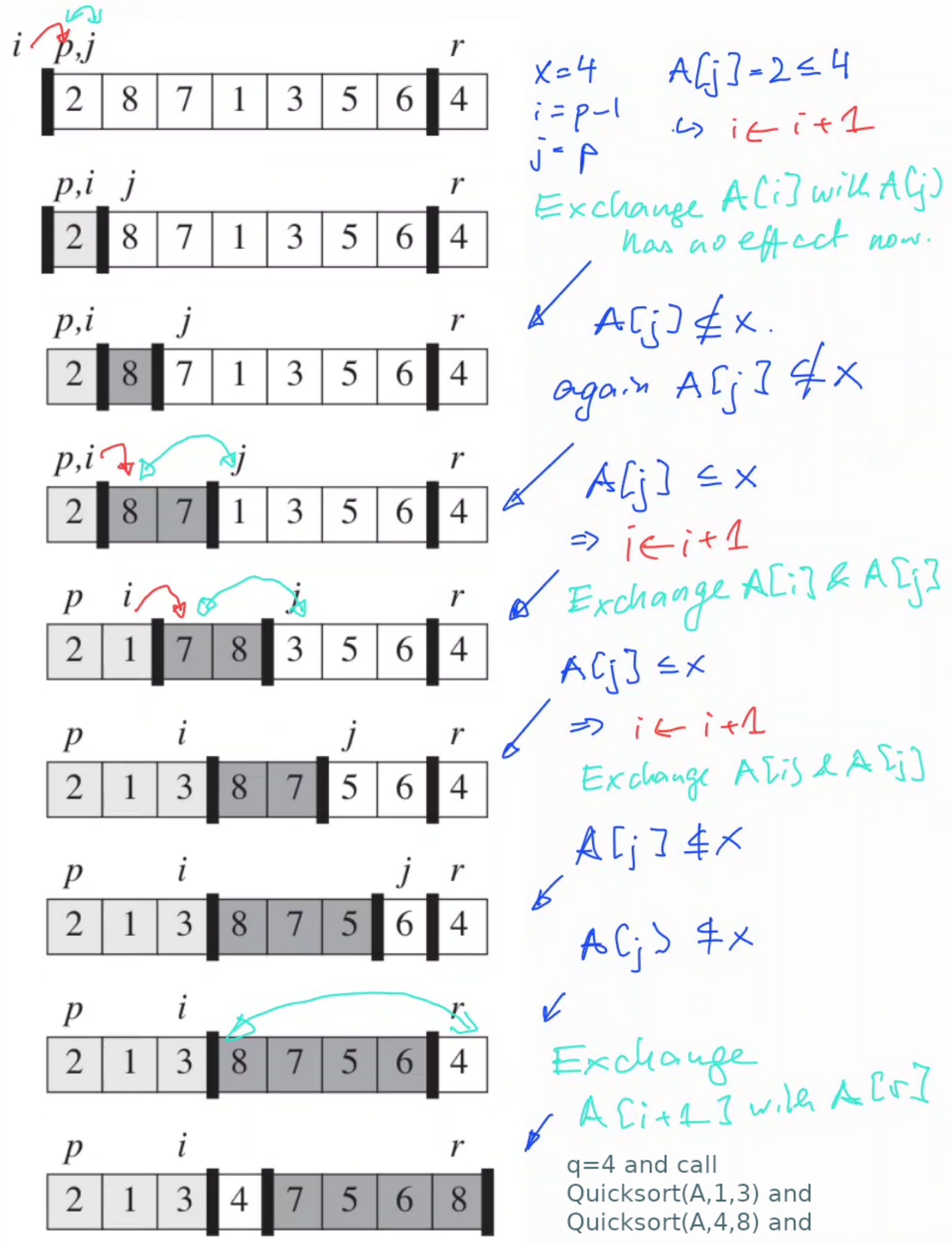
quicksort: partition/rearrange the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that all elements in $A[p..q-1]$ are less than or equal to the **pivot** $A[q]$, which is, in turn, less than or equal to each element $A[q+1..r]$. Compute the index q of the pivot as part of this partitioning procedure.

Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.

quicksort: Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quicksort.

Combine the solutions to the subproblems into the solution for the original problem.

quicksort: Because the subarrays are already sorted, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.



$x=4$ $A[j]=2\le 4$
$i=p-1$ ↳ $i\leftarrow i+1$
$j=p$

Exchange $A[i]$ with $A[j]$ has no effect now.

$A[j]\not\le x$.

again $A[j]\not\le x$

$A[j]\le x$
$\Rightarrow i\leftarrow i+1$

Exchange $A[i]$ & $A[j]$

$A[j]\le x$
$\Rightarrow i\leftarrow i+1$
Exchange $A[i]$ & $A[j]$

$A[j]\not\le x$

$A[j]\not\le x$

Exchange $A[i+1]$ with $A[r]$

q=4 and call
Quicksort(A,1,3) and
Quicksort(A,4,8) and

Theorem. *`Quicksort`$(A,\ 1,\ n)$ correctly sorts the array (in place) in $O(n^2)$ time.* $(n = A.length)$

Proof. correctness: *easy exercise - see Cormen Sec 7.1.*

runtime: Let $T(n)$ be worst-case runtime for size n input.

$q = \textsc{Partition}(A, 1, n)$ runs in $\Theta(n)$ time.

Then, we consider the subproblems $A[p..q-1]$ and $A[q+1..r]$ of total size $n-1$.

One of them is of size ℓ and the other of size $n-1-\ell$.

$$\implies T(n) = \max_{0\le\ell\le n-1}\{T(\ell)+T(n-1-\ell)\}+\Theta(n)$$

Show by induction $T(n)\in O(n^2)$

Base case $n=1$: $T(n)=T(0)+T(0)+\Theta(1)=\Theta(1)=\Theta(1^2)$, since the recursive call on array of size 0 just returns.

Assume, the statement is true for all instance of size $<n$. Consider an instance of size n

By Ind-hyp., $T(i)\in O(i^2)$ and thus, $T(i)\le c\cdot i^2$ for all $i<n$ and large enough constants c

$$\begin{aligned}\implies T(n) &\le \max_{0\le\ell\le n-1}\left\{c\cdot\ell^2+c\cdot(n-1-\ell)^2\right\}+\Theta(n)\\ &= c\cdot\max_{0\le\ell\le n-1}\left\{\ell^2+(n-1-\ell)^2\right\}+\Theta(n)\end{aligned}$$

For which ℓ is maximum in $f(\ell)=\ell^2+(n-1-\ell)^2$ achieved? Answer: For $\ell=0$ and $\ell=n-1$ (take first derivative $\frac{df(\ell)}{d\ell}=0$)

Either choice of ℓ implies that $T(n)\le c\cdot(n-1)^2+c\cdot 0+\Theta(n)=c(n^2-2n+1)+\Theta(n)\le\tilde{c}n^2\in O(n^2)$ □

In fact, there are example where worst-case is achieved, i.e., there are instances such that $T(n)\in\Omega(n^2)$ (e.g. if A is in reversed order *(exercise)*)

These worst-case examples are rare and we can obtained better expected runtime when entries in A are are pairwise distinct and randomly distributed.

Theorem. *If the entries in A are pairwise distinct and randomly distributed (i.e., each of the $n!$ possible permutations of the entries in A are equiprobable), then the average time of* `Quicksort`*$(A, 1, n)$ is in $O(n\log n)$.*

Proof. W.l.o.g. $n=A.length$ and $A[i]\in\{1,\dots,n\}$. Recall that $q=\textsc{Partition}(A,1,n)$ runs in $\Theta(n)$ time.

By assumption, each $k\in\{1,\dots,n\}$ is equally likely on some position of A.

$\implies$ with probability $\frac{1}{n}$ we have $A[n]=k$ and thus, k is a pivot in which case the problem is subdivided into two smaller problems of size $k-1$ and $n-k$

$$\begin{aligned}\implies T(n) &= \frac{1}{n}\sum_{k=1}^{n}(T(k-1)+T(n-k))+\Theta(n)\\ &= \frac{1}{n}\big((T(0)+T(n-1))+(T(1)+T(n-2))+\cdots+(T(n-2)+T(1))+(T(n-1)+T(0))\big)+\Theta(n)\\ &= \frac{1}{n}\sum_{k=0}^{n-1}2T(k)+\Theta(n)\le\frac{1}{n}\sum_{k=0}^{n-1}2T(k)+\tilde{c}n \quad \text{for large enough } \tilde{c}\end{aligned}$$

Note $T(0)=\hat{c}$ (constant time $\hat{c}$ on array of size 0: 1 comparison in L1 / constant effort in `Partition`)

$$\implies T(n)\le\hat{c}+\frac{1}{n}\sum_{k=1}^{n-1}2T(k)+\tilde{c}n\le\frac{1}{n}\sum_{k=1}^{n-1}2T(k)+cn$$

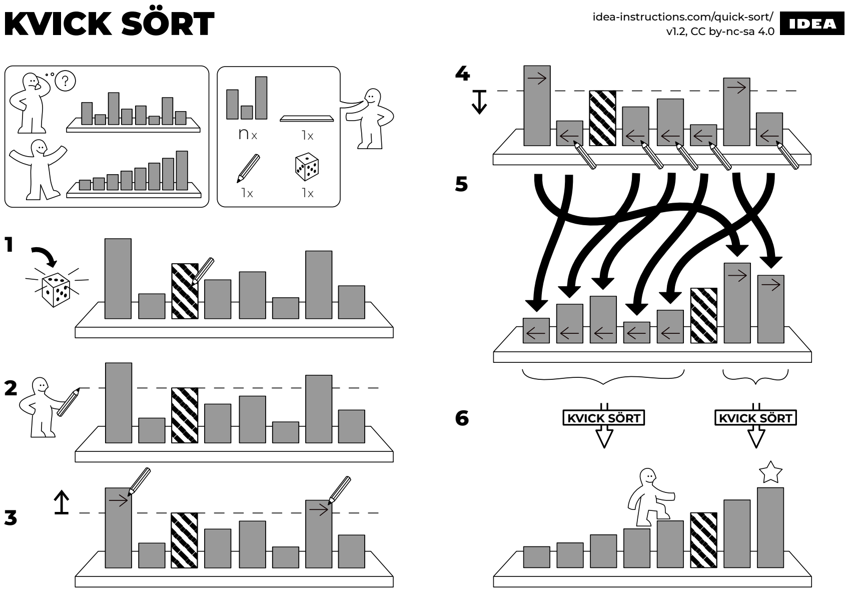
By induction, we can assume that $T(N)\in O(N\log N)$ for all $N<n$ (base case $n=1,2$ check by yourself)

$$\implies T(n)\le\frac{2}{n}\sum_{k=1}^{n-1}T(k)+cn\le\frac{2}{n}\sum_{k=1}^{n-1}(c'(k\log k))+cn \quad \text{for large enough } c'$$

Note that $\sum_{k=1}^{n-1}(k\log k)\le\int_1^n(x\log x)\,dx$ *(Why? Hint: $k\log k$ is monoton increasing).*

Moreover, as you have learned or will learn in Analysis: $\int_1^n(x\log x)\,dx=\frac{1}{2}n^2\log n-\frac{n^2}{4}+\frac{1}{4}\le\frac{1}{2}n^2\log n-\frac{n^2}{8}$

Hence, $T(n) \leq \frac{2c'}{n} \sum_{k=1}^{n-1} (k \log k) + cn \leq \frac{2c'}{n}(\frac{1}{2}n^2 \log n - \frac{n^2}{8}) + cn = c'n \log n + (c - \frac{c'}{4})n \in O(n \log n)$ □



Here, the pivot is randomly chosen!

2.5 Lower bound for "comparison sort"

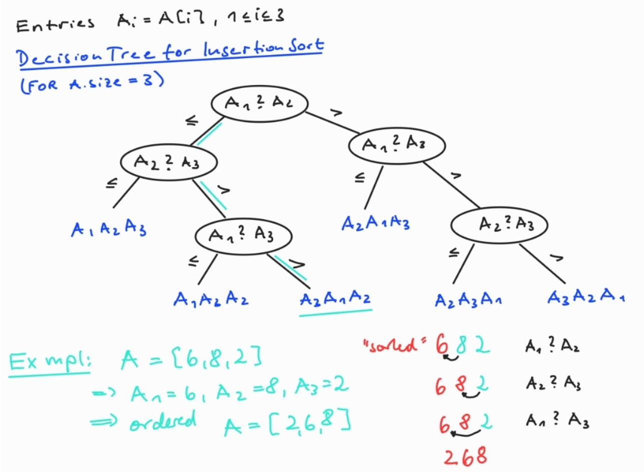
We have now seen a handful of algorithms that can sort n numbers in $O(n \log n)$ time.

Whereas merge sort and heapsort achieve this upper bound in the worst case, quicksort achieves it on average.

Moreover, for each of these algorithms, we can produce a sequence of n input numbers that causes the algorithm to run in $\Omega(n \log n)$ time *[exercise]*

These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements. We call such sorting algorithms **comparison sorts**. Thus, all the sorting algorithms introduced so far are comparison sorts.

In what follows, we show that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.



$\log(n!) = \log(1 \cdot 2 \cdot \ldots \cdot n)$ implies together with log-rules:

$$\log(n!) = \sum_{i=1}^{n} \log(i) \le \sum_{i=1}^{n} \log(n) = n\log(n) \in O(n\log(n))$$

$$\log(n!) = \sum_{i=1}^{n} \log(i) \ge \sum_{i=n/2}^{n} \log(i) =$$

$$= \log(n/2) + \log(n/2+1) + \cdots + \log(n-1) + \log(n)$$

$$\ge \log(n/2) + \ldots + \log(n/2) = n/2 \cdot \log(n/2) \in \Omega(n\log(n))$$

For the worst case, we can assume that all elements in $A[1..n]$ are distinct.

We can view comparison sorts abstractly in terms of decision trees. A **decision tree** is a full binary tree (each node is either a leaf or has both children) that represents the comparisons between elements that are performed by a particular sorting algorithm operating on an input of a given size.

Because any correct sorting algorithm must be able to produce each permutation of its input, each of the $n!$ permutations on n elements must appear as at least one of the leaves of the decision tree for a comparison sort to be correct.

Since a binary tree of height h has no more than 2^h leaves, we have $2^h \ge n! \iff h \ge \log(n!) = \Omega(n\log n)$

So, in worst-case at least $\Omega(n\log n)$ comparisons must be performed in any comparison sort to sort n elements.

Comparison sort algorithms must make $\Omega(n\log n)$ comparisons in the worst case to sort n elements. Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.

2.6 Counting-Sort

How to sort an array, if not not by comparing the elements ??

Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k and runs in $\Theta(n+k)$ time.

Hence, if $k = O(n)$, the counting sort runs in $\Theta(n)$ time.

Counting sort first determines, for each input element x, the number of elements less than or equal to x.

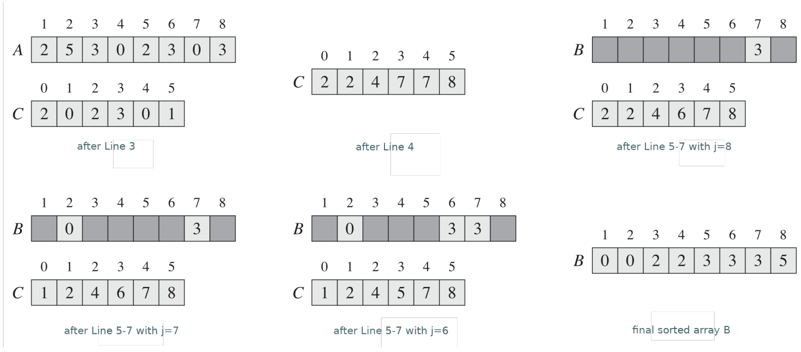
It then uses this information to place element x directly into its position in the sorted output array.

Example: $A = [2, 6, 5, 0, 1] \implies$ 4 elements are less than or equal to $x = 5$
$\implies$ in sorted array, $x = 5$ must be placed in position 4, i.e., $A[4] = 5$ must hold.

We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want them all to end up in the same position.

```
COUNTING-SORT($A$, $n$, $k$)
1 let $B[1..n]$ and $C[0..k]$ be new arrays
2 FOR ($i = 0$ to $k$) DO $C[i] = 0$
3 FOR ($j = 1$ to $n$) DO $C[A[j]] = C[A[j]] + 1$ //$C[i]$ now contains the nr of elements equal to $i$
4 FOR ($i = 1$ to $k$) DO $C[i] = C[i] + C[i-1]$ //$C[i]$ now contains the nr of elements $\leq i$
5 FOR ($j = n$ to 1) DO
6     $B[C[A[j]]] = A[j]$ //Copy $A$ to $B$, starting from the end of $A$.
7     $C[A[j]] = C[A[j]] - 1$ //to handle duplicate values
8 RETURN $B$
```

Example Board



Theorem. COUNTING-SORT(A, n, k) *correctly sorts the elements of* A *into* B *in* $\Theta(n+k)$ *time.*

Proof: correctness *[Exercise]*.

runtime:

Line 1 takes $\Theta(n+k)$ time

FOR-loops of line 2 and 4 both take $\Theta(k)$ time

FOR-loops of line 3 and 5-7 both takes $\Theta(n)$ time

$\implies$ overall time is $\Theta(k+n)$ □

In practice, we usually use counting sort when we have $k = O(n)$, in which case the running time is $\Theta(n)$. Thus, counting sort can beat the lower bound of $\Omega(n \log n)$ because it is not a comparison sort. In fact, no comparisons between input elements occur anywhere in the code.

There are many further algorithms that can beat this lower when additional information about the data to be sorted is available. E.g. bucket sorts assumes that the input is drawn from a uniform distribution and has an average-case running time of $O(n)$.

2.7 Summary

Given: A sequence of integers $(a_1, a_2 \ldots, a_n)$

Goal: A re-ordering $(a'_1, a'_2 \ldots, a'_n)$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

We have seen now several sorting algorithms (assuming n numbers need to be sorted):

- Insertion Sort: in place, runtime $O(n^2)$
- Merge Sort: not in place, runtime $\Theta(n \log n)$
- Heapsort: in place, runtime $\Theta(n \log n)$
- Quicksort: in place, worst-case runtime $\Theta(n^2)$. However, expected runtime is $\Theta(n \log n)$ and in practice it outperforms heapsort

The latter algorithms are all *comparison sorts*: they determine the sorted order of an input array by comparing elements.

We provided a lower bound of $\Omega(n \log n)$ on the worst-case running time of any comparison sort on n inputs, thus showing that heapsort and merge sort are asymptotically optimal comparison sorts.

This lower bound can be improved if one adds additional requirements on the input data and thus, if one can gather information about the sorted order of the input by means other than comparing elements. As an example, we considered

- `Counting Sort`: not in place, runtime $\Theta(n+k)$ in case the n numbers to be sorted are in $\{1, \ldots, k\}$

  $\implies$ runtime $\Theta(n)$ if $k = O(n)$.

Chapter 3

Searching and Search Trees

So-far we considered sorting. What about searching an element?

Searching in arrays means to determine if a given element (key) is in an array and, in the affirmative case, provide its position.

We focus on the following algorithms:

- Linear Search
- Binary Search
- Jump Search
- Exponential Search

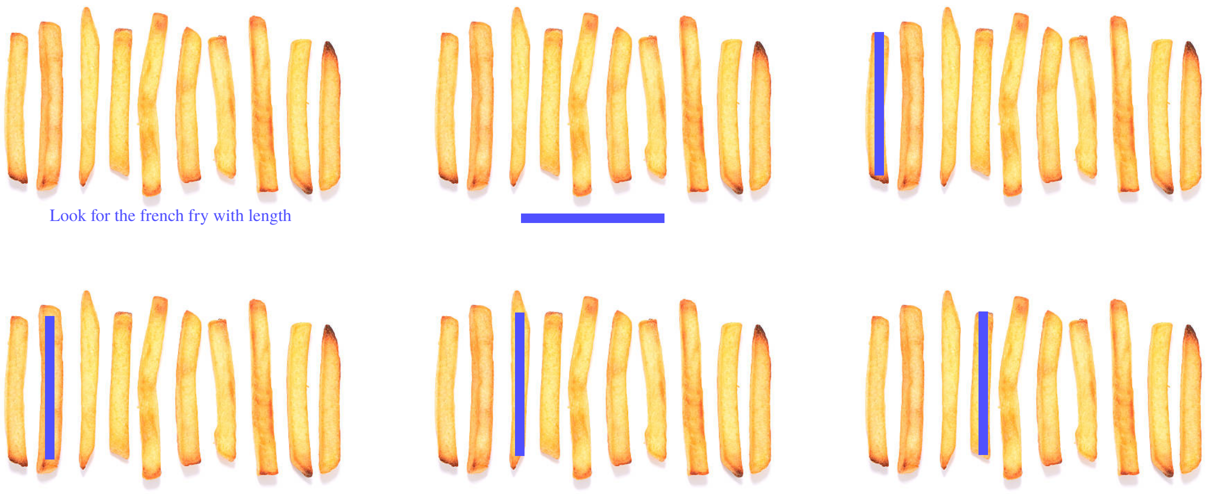
We then focus on a special data structure binary search trees (BST). In particular, we are dealing with two special types of BSTs:

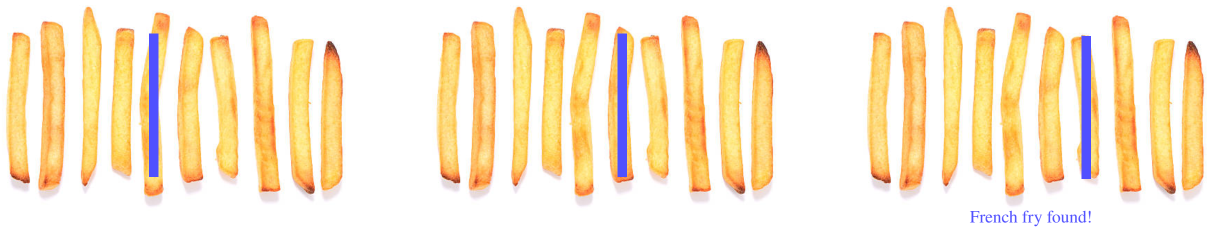
- AVL trees
- Red-Black trees

3.1 Searching in Arrays

3.1.1 Linear Search

Linear search (often also called sequential search) is a method for finding an element (key) within an array. It sequentially checks each element of the array until a match is found or the whole array has been searched.





given an array of length n:

- An unsuccessful search requires n key comparisons (all elements must be checked).
- A successful search requires at most n key comparisons in the worst case (the desired element is at the end of the list).

This implies

Time-complexity of Sequential Search: $O(n)$

Let's have a look to the average case.

Let A be an array with n pairwise distinct elements that are randomly distributed along A. Then, the average number C_{avg} of key comparisons in a successful sequential search in A is:

$$C_{avg} = \frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n}\frac{n(n+1)}{2} = \frac{n+1}{2}$$

WHY? since, with probability $\frac{1}{n}$ a key x is at position i and to find this key needs then i comparisons. The necessary key comparisons for all cases (with x at position i) amount to: $1+2+\cdots+n$.

Can we do better, e.g., if A is already sorted?

3.1.2 Binary Search

Linear search: checking if x is in $A[1..n]$ works in $O(n)$ time. *However, we can do better!*

Suppose that A is sorted, then we can apply binary search
[sorting costs $O(n \log n)$ time but needs to be done only once!]:

```
BinarySearch($A$, $x$, $first$, $last$) //find $x$ in sorted $A[first..last]$ initialized with $(A, x, 1, n)$
    IF ($first > last$) THEN RETURN "A does not contain x"
    $mid = \lfloor (first + last)/2 \rfloor$
    IF ($A[mid] = x$) THEN RETURN "A contains x on position mid"
    IF ($A[mid] > x$) THEN BinarySearch($A$, $x$, $first$, $mid - 1$)
    ELSE BinarySearch($A$, $x$, $mid + 1$, $last$)
```

Example. Find $x = 3$ in A

$$\begin{array}{lccccccc} \text{pos} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \text{A=} & [1 & 2 & 3 & 5 & 7 & 8 & 9\,] \end{array}$$

call BinarySearch(A, x, 1, 7) $\implies$ $mid = \lfloor (1+7)/2 \rfloor = 4$
Since $A[4] = 5 > 3$ and A is sorted, x must be contained in $A[1..3]$ (if x exists in A)

Example. Find $x = 3$ in A

$$\begin{array}{llllllll} \text{pos} & \mathit{1} & \mathit{2} & \mathit{3} & 4 & 5 & 6 & 7 \\ \text{A=} & [\,\mathit{1} & \mathit{2} & \mathit{3} & 5 & 7 & 8 & 9\,] \end{array}$$

call `BinarySearch`(A, x, 1, 3) $\implies$ $mid = \lfloor(1+3)/2\rfloor = 2$
Since $A[2] = 2 < 3$ and A is sorted, x must be contained in $A[3..3]$ (if x exists in A)

Example. Find $x = 3$ in A

$$\begin{array}{llllllll} \text{pos} & 1 & 2 & \mathit{3} & 4 & 5 & 6 & 7 \\ \text{A=} & [\,1 & 2 & \mathit{3} & 5 & 7 & 8 & 9\,] \end{array}$$

call `BinarySearch`(A, x, 3, 3) $\implies$ $mid = \lfloor(3+3)/2\rfloor = 3$
Since $A[3] = 3$ and we found x

Theorem. `BinarySearch`(A,x,1,n) *correctly determines if x exists in sorted A in* $\Theta(\log_2(n))$

Proof. correctness-sketch: If $x = A[\lfloor\frac{n}{2}\rfloor]$ we are done.
Otherwise, if $x < A[\lfloor\frac{n}{2}\rfloor]$ it must be contained (if at all) in $A[1..\lfloor\frac{n}{2}\rfloor - 1]$ (Since A is sorted)
Otherwise, if $x > A[\lfloor\frac{n}{2}\rfloor]$ it must be contained (if at all) in $A[\lfloor\frac{n}{2}\rfloor + 1, n]$ (Since A is sorted)
Now recurse (exercise)

runtime: *any idea?* $T(n) = T(\frac{n}{2}) + \Theta(1)$
Via Mastertheorem: $a = 1, b = 2, d = 0 \implies a = b^d \implies T(n) \in \Theta(n^0 \log_2(n)) = \Theta(\log_2(n))$

To give you a sense of just how fast binary search:

Twenty questions is a popular children's game:

- Player 1 selects a word (e.g. from a printed dictionary)
- Player 2 repeatedly asks true/false questions in an attempt to guess it.
- If the word remains unidentified after 20 questions, the player 1 wins; otherwise, the player 2 takes the honors.

Is there a winning strategy for player 2
Answer: YES!

- Player 2 opens dictionary in the middle, selects a word (say "move"), and asks whether the unknown word is before "move" in alphabetical order. Since standard dictionaries contain $50'000$ to $200'000$ words, we can be certain that the process will terminate within twenty questions since $log_2(200'000) = 17.61$.

  Player 2 always wins!

3.1.3 Jump Search

Pre-condition: (1) Array L sorted (increasing) and (2) keys in L are pairwise distinct
(first element is $L[1]$)

Principle: L is divided into sections of fixed length m. Jump over the sections to determine the section of the key.

Sections: $1 \ldots m$, $m+1 \ldots 2m$, $2m+1 \ldots 3m$, and so on.

Simple Jump Search:

- Jump to positions $i \cdot m + 1$ (for $i = 1, 2, \dots$) one after another.
- As soon as $x < L[i \cdot m + 1]$, x can only be in the i-th section $(i-1) \cdot m + 1 \dots i \cdot m$;
- In this i-th section, we apply linear search for finding x.

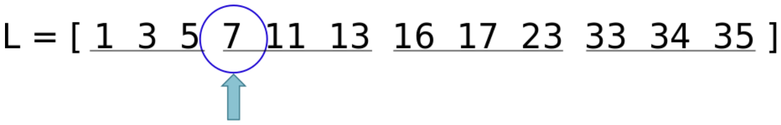
Example:

L = [1 3 5 7 11 13 16 17 23 33 34 35]

Find key $x = 17$ in L und chose here jump_width $m = 3$.

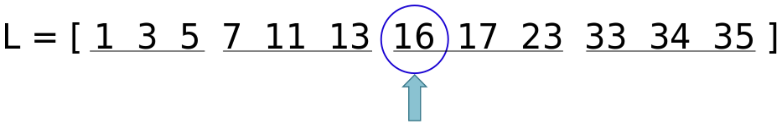
L = [1 3 5 7 11 13 16 17 23 33 34 35]

Subdivision of L into blocks of length $m = 3$.



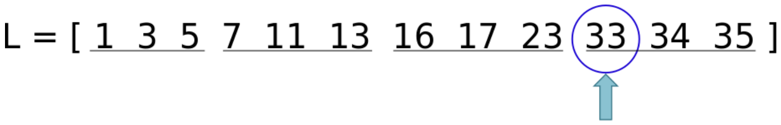
$i = 1$, $m = 3$ **and jump to** $i \cdot m + 1 = 4$

Since $L[4] = 7 < 17$, we take the next $i := 2$.



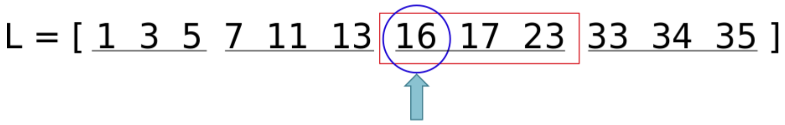
$i = 2$, $m = 3$ **and jump to** $i \cdot m + 1 = 7$

Since $L[7] = 16 < 17$, we take next $i := 3$.



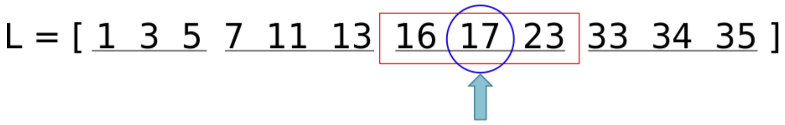
$i = 3$, $m = 3$ **and jump to** $i \cdot m + 1 = 10$

Since $L[10] = 33 > 17$, it follows that 17 must be in section $L[2m + 1..3m]$ (if 17 is in L at all).



Start linear search in section $[16, 17, 23]$.

Since $16 < 17$ (already compared), go to next element in this section



We find 17 at position 8 in L, return 8 and stop searching.

Assuming that all keys in L are randomly distributed and n is length of L and jumps and comparison can be done in constant time:

Average Search Costs[a]**:**

$$C_{avg}(n) \in O(\frac{n}{m} + m)$$

- In total, at most $\frac{n}{m}$ jumps are possible
- and we need to check one of the blocks of size m to check if x exists via linear search

Question: Is there an optimal jump-width?

Idea: One could attempt to optimize the average key comparisons, i.e., finding the m that minimizes $\frac{n}{m} + m$.

Sketch: Take the derivative of $C_{avg}(n)$, set it to 0, and solve the equation for m:

$$\frac{d}{dm} C_{avg}(n) = 1 - \frac{n}{m^2} = 0 \quad \Rightarrow \quad m^2 = n \quad \Rightarrow \quad m = \sqrt{n}$$

$\Rightarrow$ Optimal jump length $m = \lfloor \sqrt{n} \rfloor$; then complexity is in $O(\sqrt{n})$ [better than linear search!]

[a] For more information, see Shneiderman, B. (1978) "Jump searching: a fast sequential search technique" Communications of the ACM, 21(10), pp.831-834.

Jump seach is better than linear search but worse than binary search.

Why not use binary search instead?

If your data is too large for main memory, with jump search, only parts (blocks) need to be loaded into main memory!

3.1.4 Exponential Search

Question: How can we search when the length of a search range is initially unknown?

Binary search and or jump search with optimal jump-width assume that one knows the length of the range to be searched before starting the search. However, there may be cases where the search range is finite but “practically” unlimited. In such a case, it is reasonable to first determine an upper limit for the range to be searched, within which an element with key k must lie if such an element exists at all.

Idea: Determine, in exponentially growing steps, a range in which the search key must lie.

Principle of Exponential Search for x in increasingly sorted L (first entry $L[1]$):

1. Test $L[1]$, $L[2]$, $L[4]$, $L[8]$, ..., $L[2^j]$, ...
2. At the smallest j such that $x \leq L[2^j]$: Either x is in $L[(2^{j-1}+1)\dots 2^j]$ or x is not in L.
3. Search within $[L[2^{j-1}+1], ..., L[2^j]]$ using any search method.

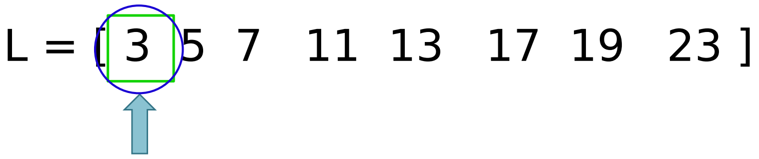
```
Exponential Search($L$ (sorted increasingly), $x$)
IF ($x = L[1]$) RETURN 1
$i := 2$
WHILE ($x > L[i]$) DO //Determine boundaries of search space
    $i := 2 \cdot i$
FOR ($j = i/2+1, i/2+2, \dots, i$) DO
    IF ($L[j] = x$) THEN RETURN $j$
RETURN $-1$//$x$ not in $L$
```

Missing detail: Must be careful here, to avoid that program crashes when i in while-loop is out of range of L. In the pseudo-code, we simply assume that it terminates when $L[i] = NIL$, i.e., i is out of range of L.

Example.

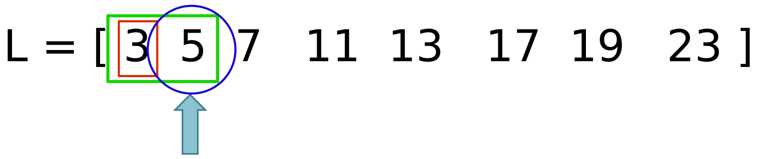
L = [3 5 7 11 13 17 19 23]

Find $x = 13$ in L.



$\mathbf{i} = \mathbf{1}$

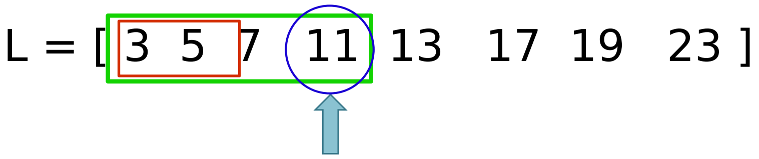
Since $L[1] = 3 < 13$, we have $i := 2 \cdot 1 = 2$. Check now $L[2]$.



$\mathbf{i} = \mathbf{2}$.

Note: In red part, we cannot find x since L is sorted.

Since $L[2] = 5 < 13$, we have $i := 2 \cdot 2 = 4$. Check now $L[4]$.

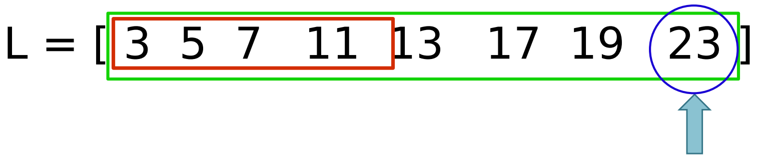


L = [3 5 7 11 13 17 19 23]

$\mathbf{i = 4}$.

Note: In red part, we cannot find x since L is sorted.

Since $L[4] = 11 < 13$, we have $i := 2 \cdot 4 = 8$. Check now $L[8]$.

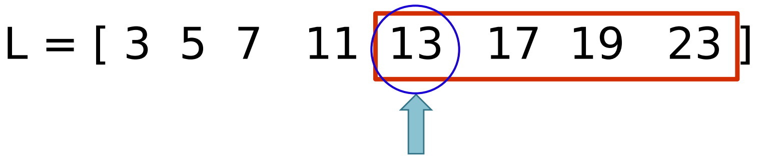


L = [3 5 7 11 13 17 19 23]

$\mathbf{i = 8}$.

Note: In red part, we cannot find x since L is sorted.

Since $L[8] = 23 > 13$, it follows that 13 is in $L[5..8]$ (if x in L at all).



L = [3 5 7 11 13 17 19 23]

Check existence of $x = 13$ via linear search in $L[5..8]$:
In the first step of the linear search we found 13. Return position of 13.

Costs.

Theorem. *If L contains only pairwise-disinct keys from $\mathbb{N}$, then exponential search runs in $O(\log_2 x)$ time*

- Note, $\lfloor \log_2 x \rfloor + 1 =$ number of bits in binary representation of the key x [x not size of array!]
  Hence, if $x \leq 2^k$ for some constant k, then $\log_2 x \leq \log_2 2^k = k$ and thus, exponential search runs in constant time

Proof-sketch (runtime in $O(\log_2 x)$).

Since L contains only pairwise-distinct keys from $\mathbb{N}$

$\Longrightarrow$ Key values x grow at least as fast as the element indices, i.e., $L[k] \geq k \; \forall k$.

$\Longrightarrow$ i is doubled at most $\log_2 x$ times because $i \geq \log_2(x)$ implies
$L[2^i] \geq 2^i \geq 2^{\log_2(x)} = x$. *(gives stop criterion!)*

$\Longrightarrow$ Determine the correct interval: in $\leq j \leq \log_2 x$ comparisons with $i = 2^j$.

Thus, $j \in O(\log_2 x)$

Length of this interval: $2^j - 2^{j-1} = 2^{j-1}(2-1) = 2^{j-1}$ where $j \in O(\log_2 x)$.
This means that the length of the interval being searched is in $O(2^{\log_2(x)-1})$.

Search within this intervall (e.g., binary search): $O(\log_2(2^{\log_2(x)-1})) = O(\log_2(x) - 1) = O(\log_2 x)$.

$\Longrightarrow$ Overall effort $O(\log_2 x)$ □

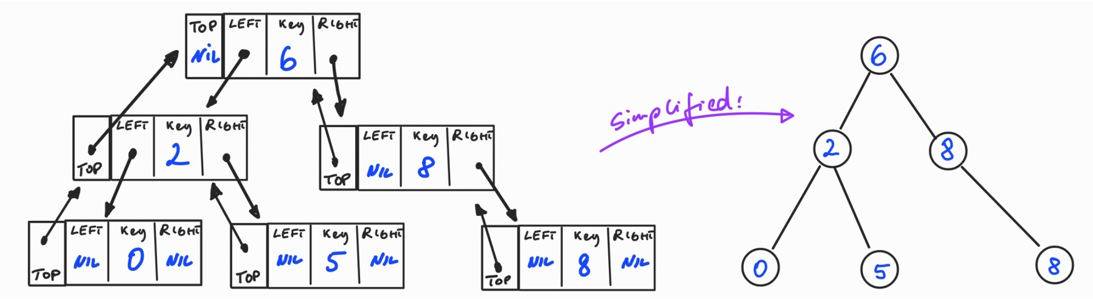
While searching in arrays is reasonable fast once they are sorted, modication of arrays (removing or adding elements) is a somewhat teadious task.

To support dynamic-set operations (including search, find_minimum, find_maximum, but also insert or delete) a tree data structure is more suitable.

3.2 Search Trees

3.2.1 Binary Search Trees

A **binary** tree is a rooted tree for which each vertex has *at most* two children.



In a tree data structure, each vertex x is an object with

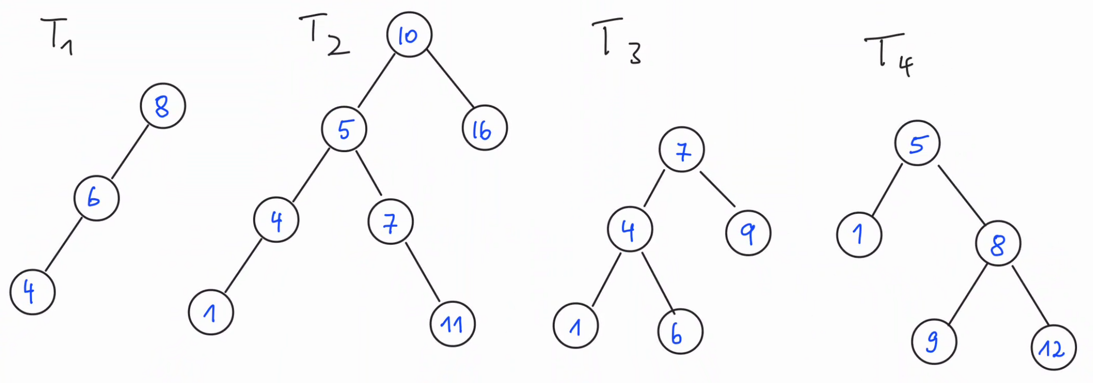
- $x.key$ some value to be stored (maybe also some extra satellite data)
- $x.left$, $x.right$, $x.top$ are pointers refering to the address of the left child, right child and parent of x, respectively

  If a child or the parent is missing, the appropriate attribute contains the value NIL .

A **Binary Search Trees (BST)** is a binary tree in which the keys are always stored in such a way that they satisfy the binary-search-tree property:

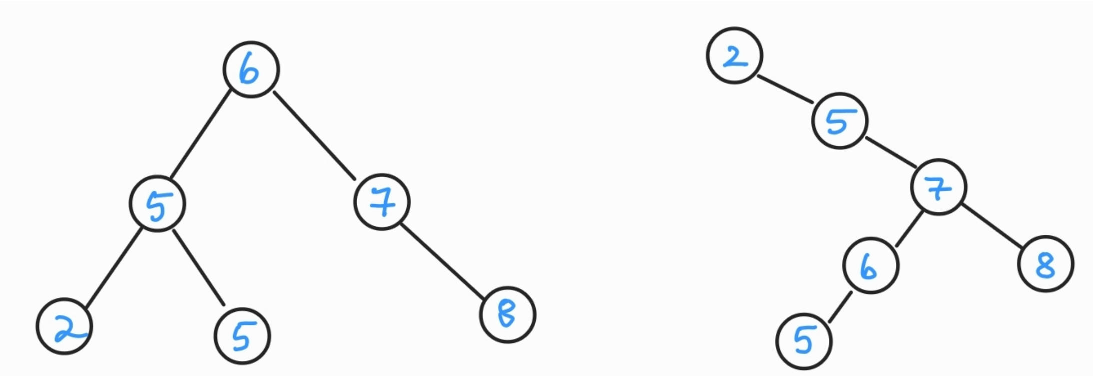
- Let x be a node in a binary search tree.
- If y is a node in the left subtree of x, then $y.key \leq x.key$.

  If y is a node in the right subtree of x, then $y.key \geq x.key$.



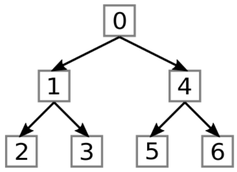
Which of them are search-trees?
Answer: Only T_1 and T_3 (T_2: 10/11, T_4: 8/9)

Binary Search Trees are not necessarily uniquely determined:



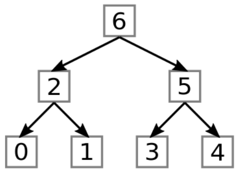
***Pre*order:**

1. *visit current vertex*
2. recursively traverse left subtree
3. recursively traverse right subtree



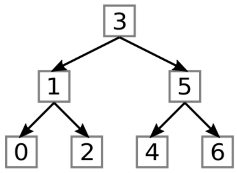
***Post*order:**

1. recursively traverse left subtree
2. recursively traverse right subtree
3. *visit current vertex*



***In*order:**

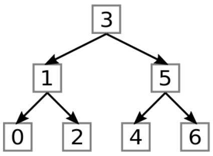
1. recursively traverse left subtree
2. *visit current vertex*
3. recursively traverse right subtree



numbers in squares =
order in which nodes are visited

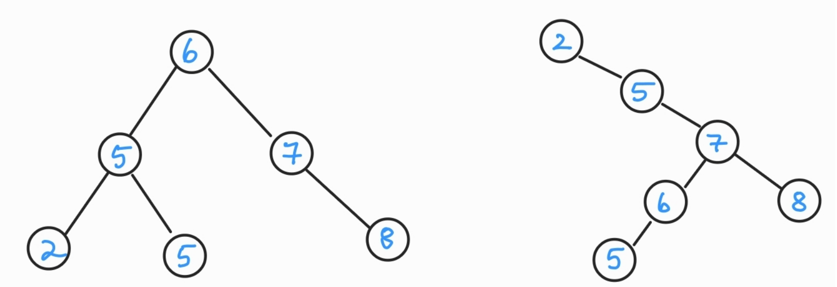
Inorder:

1. recursively traverse left subtree
2. visit current vertex
3. recursively traverse right subtree



```
INORDER-TREE-WALK($x$)
IF ($x \neq NIL$) THEN
    INORDER-TREE-WALK($x.left$)
    PRINT $x.key$
    INORDER-TREE-WALK($x.right$)
```

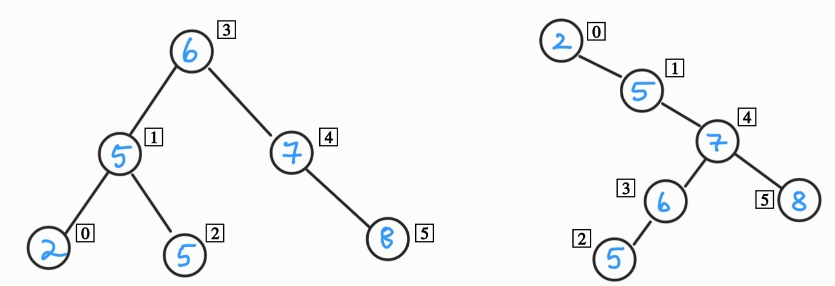
Execute INORDER-TREE-WALK(x) *in:*



PRINT $x.key$: 2,5,5,6,7,8

Inorder traversal allows us to print all elements in a search tree in sorted order.

Execute INORDER-TREE-WALK(x) *in:*



PRINT $x.key$: 2,5,5,6,7,8

Inorder traversal allows us to print all elements in a search tree in sorted order.

Theorem. *If x is a root of an n-vertex binary search tree, then* INORDER-TREE-WALK(x) *prints all elements in sorted order in $\Theta(n)$ time*

Proof. *correct:* due to binary-search-tree property:
$y.key \leq x.key$ if y in left subtree and $x.key \leq y.key$ if y in right subtree

runtime: INORDER-TREE-WALK(x) visits all n nodes of the subtree $\implies T(n) = \Omega(n)$

left subtree $k \geq 0$ nodes and right subtree has $n-k-1$ nodes
$\implies T(n) = T(k) + T(n-k-1) + d$ where $T(0) = c$ (c, d constants)

Show, by induction, $T(n) = (c+d)n + c$.

Base case $\ell = 0$: $T(0) = (c+d) \cdot 0 + c = c$ correct

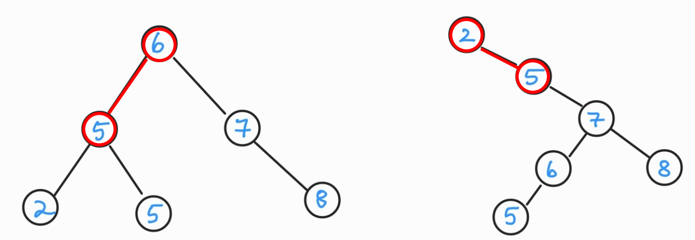
Assume $T(\ell) = (c+d)\ell + c$ true for all $\ell < n$.

$$T(n) = T(k) + T(n-k-1) + d =$$
$$= [(c+d)k + c] + [(c+d)(n-k-1) + c] + d = (c+d)n + c = O(n)$$ □

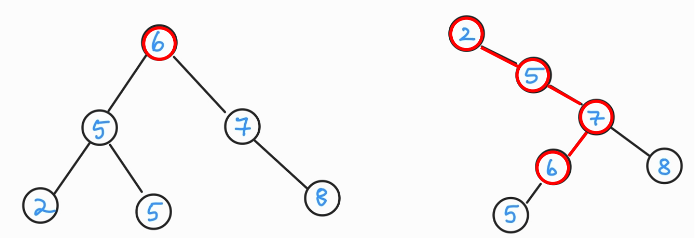
As discussed next, binary search trees support the queries search, find_minimum, find_maximum, ...

Each query can be done on $O(h)$ time on any binary search tree of height h.
Recall: height of tree T is $h(T)$ = #edges along longest simple path from ρ_T to a leaf.

```
//find value $k$ in subtree rooted at $x$
Tree-Search($x, k$)
    IF ($x = NIL$ or $k = x.key$) THEN
        RETURN $x$
    IF ($k < x.key$) THEN
        RETURN Tree-Search($x.left, k$)
    ELSE RETURN Tree-Search($x.right, k$)
```



Tree-Search$(T.root, 5)$



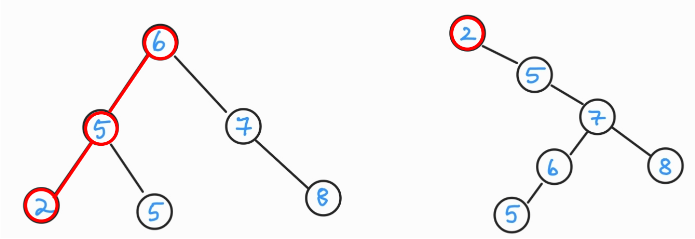
Tree-Search$(T.root, 6)$

The Tree-Search procedure begins its search at the root and traces a simple "downward" path

If $k = x.key$, search terminates (k found). If $x = NIL$, search terminates (k not found).
Hence, $k \neq x.key$ and $x \neq NIL$ implies either $k < x.key$ (continue left) or $k > x.key$ (continue right).
Due to binary-search-tree property, Tree-Search is correct.

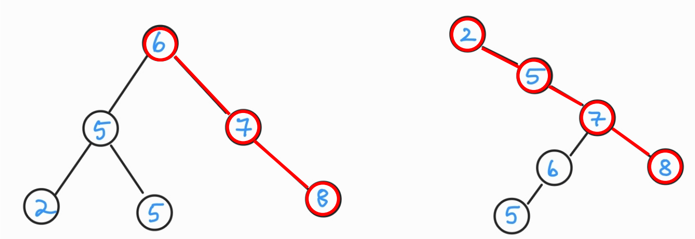
The nodes encountered during the recursion form a simple path downward from the root of the tree, and thus the running time of Tree-Search is $O(h)$ where h is the height of the tree.

```
//Find minimum key in $T(x)$ assuming $x \neq NIL$
Tree-Min($x$)
    WHILE ($x.left \neq NIL$) DO
        $x := x.left$
    RETURN $x$
```



Tree-Min($T.root$)

```
//Find maximum key in $T(x)$ assuming $x \neq NIL$
Tree-Max($x$)
    WHILE ($x.right \neq NIL$) DO
        $x := x.right$
    RETURN $x$
```

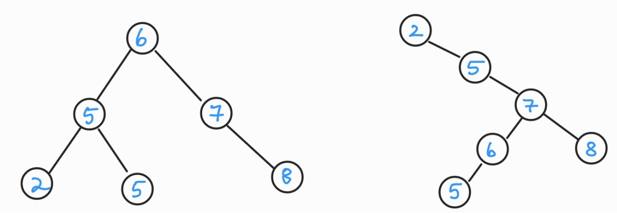


Tree-Max($T.root$)

Similar arguments as before show that Tree-Min(x), resp., Tree-Max(x) correctly determines the max, resp., min element in the subtree rooted at x in $O(h)$ time where h is the height of the tree.

```
Tree-Insert($T, z$)
    $x := T.root$ //node being compared with $z$
    $y := NIL$ //$y$ will be parent of $z$
    WHILE ($x \neq NIL$)
    //descend until reaching a leaf
        $y := x$
        IF ($z.key < x.key$) THEN $x := x.left$
        ELSE $x := x.right$
    $z.top := y$
    //found the location - insert $z$ with parent $y$
    IF ($y = NIL$) THEN $T.root := z$ //$T$ was empty
    ELSEIF ($z.key < y.key$) THEN $y.left := z$
    ELSE $y.right := z$
```

Example board



get left tree by insertion in order e.g. $6, 5, 5, 2, 7, 8$
get right tree by insertion in order e.g. $2, 5, 7, 6, 5, 8$

BINARY SEARCH TREE

Insert:

assume $T = \emptyset$ & we insert elements in order: $(5, 6)$

Tree-Insert (T, z) // $T = \emptyset$ so-far, i.e., T.root = NIL

```
L1   x = T.root = NIL
L2   y = NIL
L3   WHILE not applied as x = NIL
 {
L7   z.top = y = NIL
L8   IF ( y = NIL ) => T.root := z
     stop
```

z:

TOP	L	key	R
NIL	NIL	5	NIL

T.root = z =^? (5) = T.root!

Tree-Insert (T, z)

z:

TOP	L	key	R
NIL	NIL	6	NIL

```
L.1   x = T.root
L2    y = NIL
L3    WHILE( x ≠ NIL)
L4          y = x
      L5  %
      L6 yes:   x := x.right   (= NIL)
   while-loop stop as  x = NIL
```

x:

TOP	L	key	R
NIL	NIL	5	NIL

= T.root

‖ y

x.right

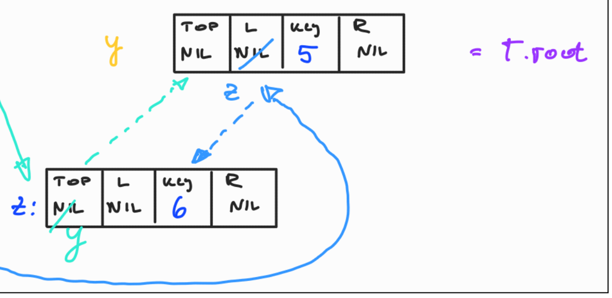
TOP	L	key	R
NIL	NIL	NIL	NIL

"NIL"

‖ x

up to here!

```
L.7   z.top = y
L.8   %
L.9   %
L.10  z.key ≥ y.key
        ‖
        6      => y.left = z
```



y

TOP	L	key	R
NIL	~~NIL~~	5	NIL

= T.root

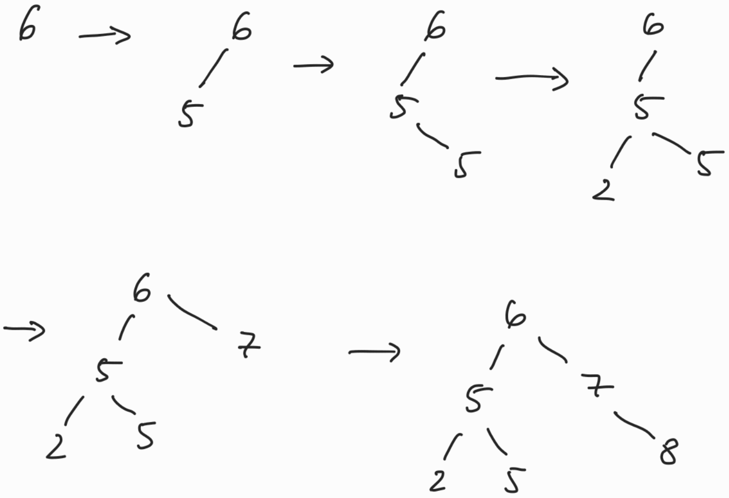
z

z:

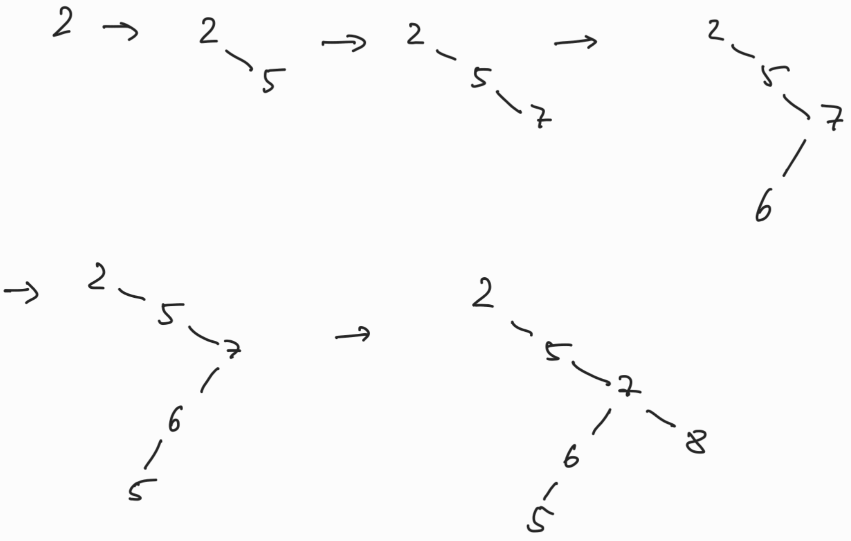
TOP	L	key	R
~~NIL~~ y	NIL	6	NIL

insert "short & simplified tree representation:

insert in order: 6, 5, 5, 2, 7, 8

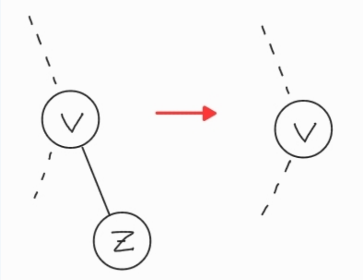


insert in order: 2, 5, 7, 6, 5, 8

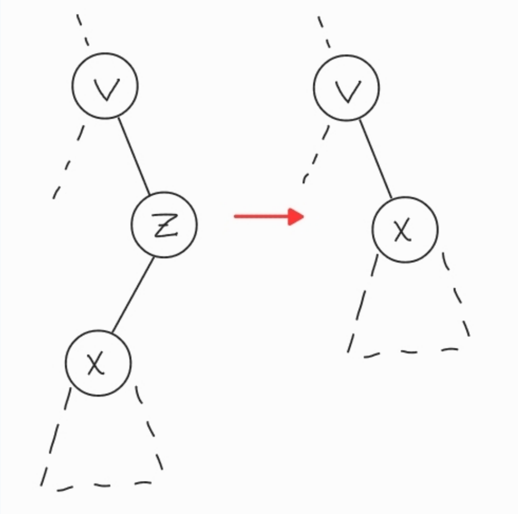


Lemma. *`Tree-Insert`(T, z) yields a BST and runs in $O(h)$ time where h = height of tree [exercise]*

`Tree-Delete`(T, z) *(pseudocode = exercise)*

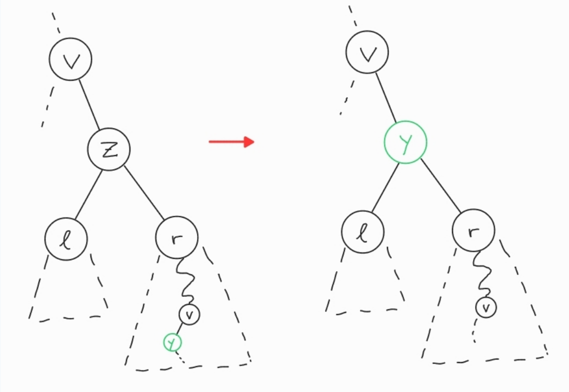


CASE z has no child:
just delete z
$\Rightarrow T$-z is BST



CASE z has one child:
Delete z and make child x of z ..
.. the right child of parent$(z) = v$ in case z is right child of v
.. the left child of parent$(z) = v$ in case z is left child of v
to get tree T'

Case: z is right child of v:
T is $BST \Rightarrow \forall\ w$ in $T(z)$: $v.key \leq w.key$
$\Rightarrow \forall\ w$ in $T(x)$: $v.key \leq w.key$
$\Rightarrow \forall\ w$ in $T'(x)$: $v.key \leq w.key$
and other *key* "posititions" unchanged in T'
$\Rightarrow T'$ is BST.
(for case z is left child of v replace $\leq$ by $\geq$)



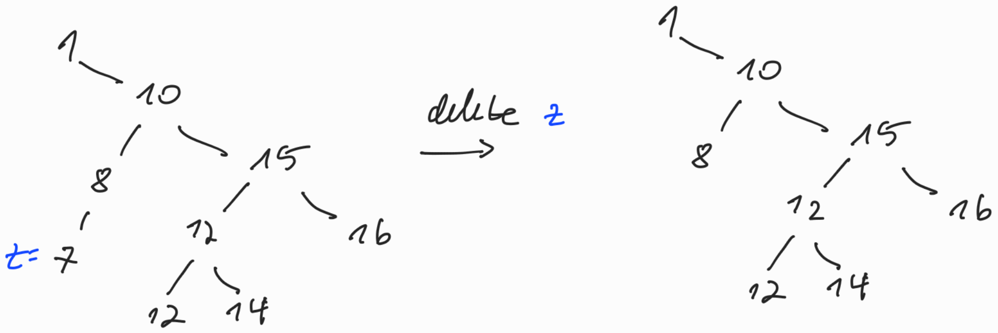
CASE z has two children:
Find y in $T(r)$ with min $y.key$ and
such that y has no left child $\Rightarrow$ y has 0 or 1 child
[latter needed if $y'.key = y.key$ for some y']
Delete y from T [see cases above]
Replace z by y [$y = r$ is possible] and we get tree T'

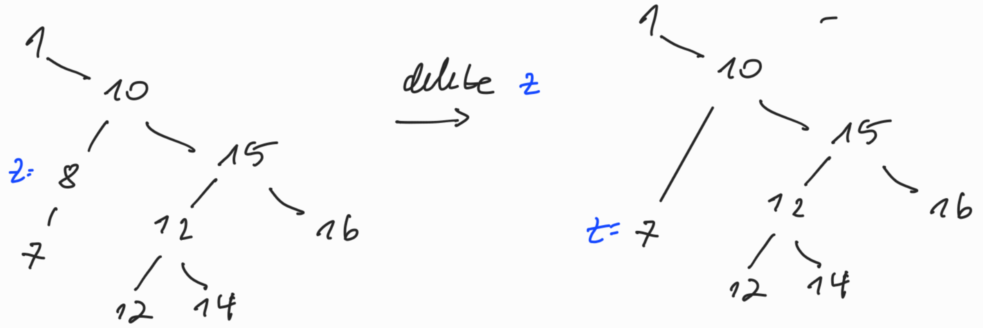
By choice of y: $\forall w \in T(r) : w.key \leq y.key$
Since T is BST:$\forall w' \in T(\ell) : w'.key \leq z.key \leq y.key$
$\Rightarrow \forall w \in T'(r)$:
$y.key \leq w.key$ and $\forall w' \in T(\ell) : w'.key \leq y.key$
and other *key* "posititions" unchanged in T'
$\Rightarrow T'$ is BST

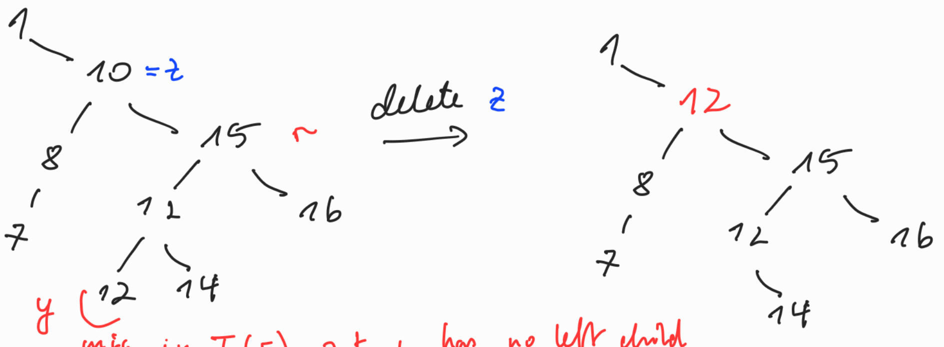
Delete vertex w means remove w from $V(T)$ and all edges from $E(T)$ that contain w !! *[Example Board]*

Lemma. *`Tree-Delete`(T, z) yields a BST and runs in $O(h)$ time where h = height of tree [exercise]*

delete





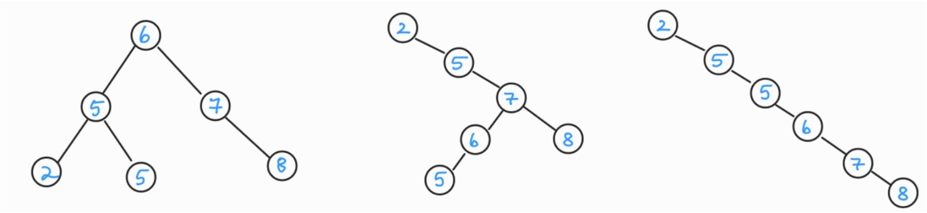


y (min in T(r) s.t y has no left child

Since y min ⇒ all keys in T(r) are ≥ y.key
while z.key ≤ y.key (as y ∈ T(r))
⇒ correct.!

To summarize at this point:

- queries as search, find_minimum, find_maximum as well as insertion, deletion can be done in $O(h)$ time in binary search trees where h = height of tree.
- Problem: $O(h) = O(n)$ where n =number of vertices [can we control the height?]



- We consider now AVL trees and Red-Black trees that are one of *many* search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

3.2.2 AVL Trees

Recall

- height of x in T: $h(x) :=$ #edges along longest simple path from x to a leaf $l \preceq_T x$
- height of T: $h(T) = h(\rho_T)$, i.e., height of root ρ_T of T
- We define the height of an empty tree as $h(\emptyset) = -1$.

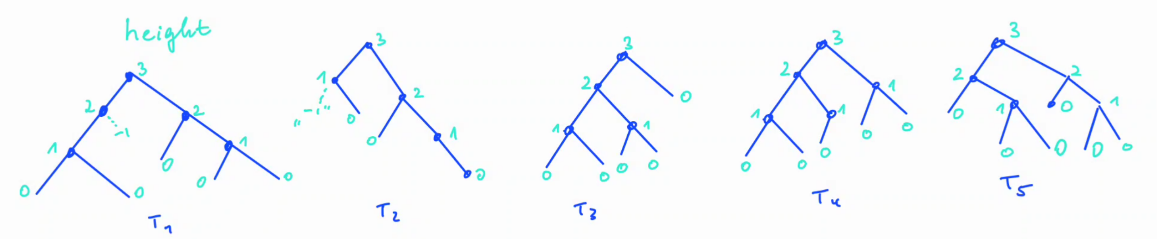
An **AVL tree** is a binary search tree (BST) in which the height of the two subtree rooted at children of any vertex differ by at most 1.

More formal:

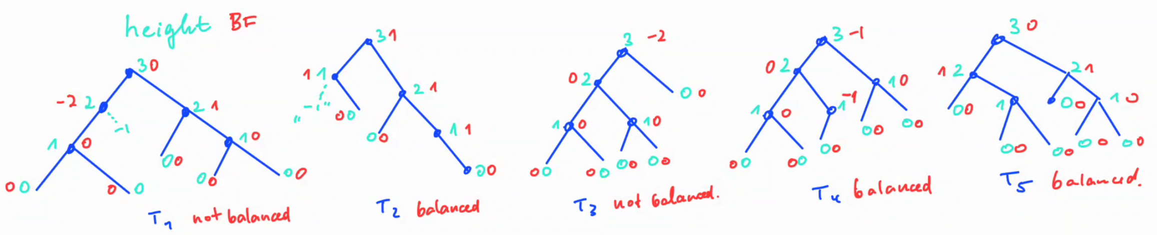
- Define the **balance factor of x** in a binary tree T as $BF(x) = h(T(x.right)) - h(T(x.left))$.
- A binary tree is **balanced** if $BF(x) \in \{1, 0, -1\}$ for all nodes x in T.
- A balanced BST T is called **AVL tree**.

[AVL named after inventors **A**delson-**V**elsky and **L**andis]

Which of the trees is balanced?



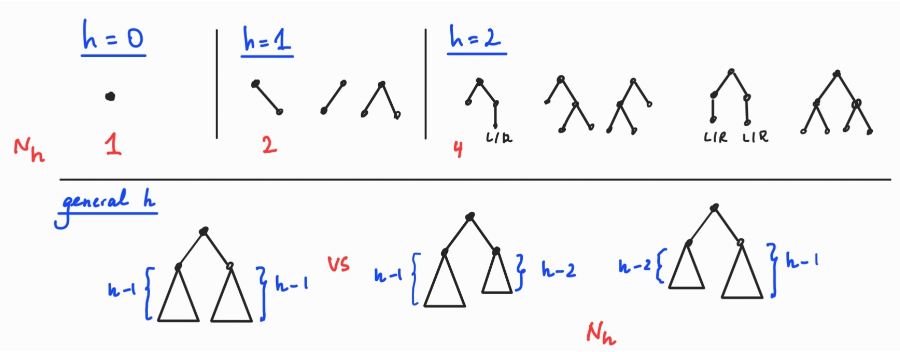
Which of the tree is balanced?



T is **AVL tree** = if T is a balanced BST, i.e., $BF(x) \in \{1, 0, -1\}$ for all nodes x in T with $BF(x) = h(T(x.right)) - h(T(x.left))$.

Lemma. *Any AVL tree with n nodes has height $O(\log n)$.*

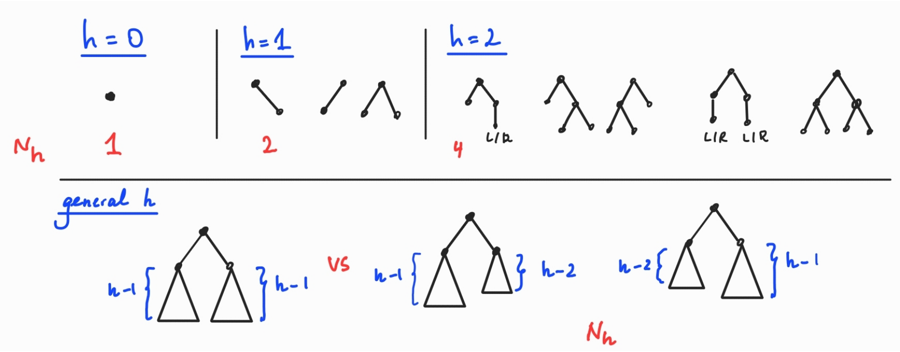
Proof: Let N_h = min number of vertices in AVL tree of height h $\implies$ $N_h = 1 + N_{h-1} + N_{h-2}$



$$N_h = 1 + N_{h-1} + N_{h-2}$$

We now show, by induction on h, that $N_h > 2^{\frac{h}{2}-1}$ for all $h \geq 0$

Base cases: $h = 0 \implies N_h = 1 > 2^{\frac{0}{2}-1} = 0.5$ and $h = 1 \implies N_h = 2 > 2^{\frac{1}{2}-1} \simeq 0.71$



Assume now that $N_k > 2^{\frac{k}{2}-1}$ for all $k \in \{0, 1, \ldots, h-1\}$.

Let T be an AVL tree of height h with N_h vertices.

Hence, $N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} > 2N_{h-2} \overset{Ind.hyp.}{>} 2(2^{\frac{h-2}{2}-1}) = 2^{\frac{h-2}{2}} = 2^{\frac{h}{2}-1}$

(which completes induct.-proof)

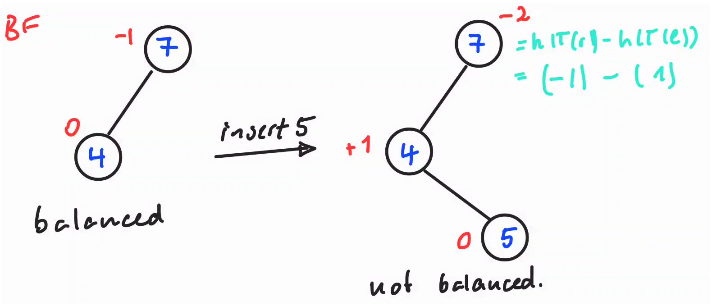
$\implies N_h > 2^{\frac{h}{2}-1} \iff \log_2(N_h) > \frac{h}{2} - 1 \iff 2\log_2(N_h) + 2 > h$

For general AVL tree T with height h it holds that $|V(T)| \geq N_h$

$\implies h < 2\log_2(N_h) + 2 \leq 2\log_2(|V(T)|) + 2$ and thus, $h \in O(\log_2(|V(T)|))$ □

Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $h \in O(\log n)$ time in AVL trees.

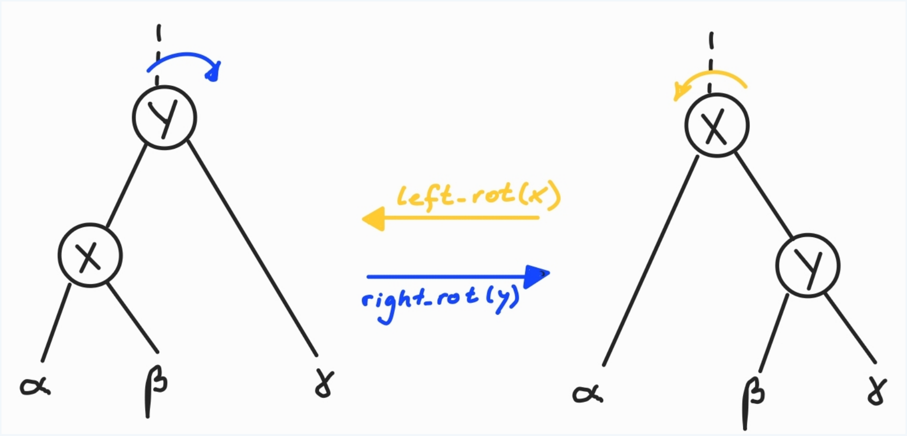
However, after a single use of the operations insert and delete it is not ensured that the resulting tree is still balanced, that is, the BST is possibly not an AVL tree.



Hence, it is not ensured that the latter operations still run in $O(\log n)$ time $\implies$ *must correct trees to obtain AVL trees*

These corrections should run in $O(\log n)$ time to ensure that the overall time complexity together with the operations as above remains in $O(\log n)$

An important role for obtaining a AVL tree after insertion/deletion are **rotations**:



[Some Examples on Board]

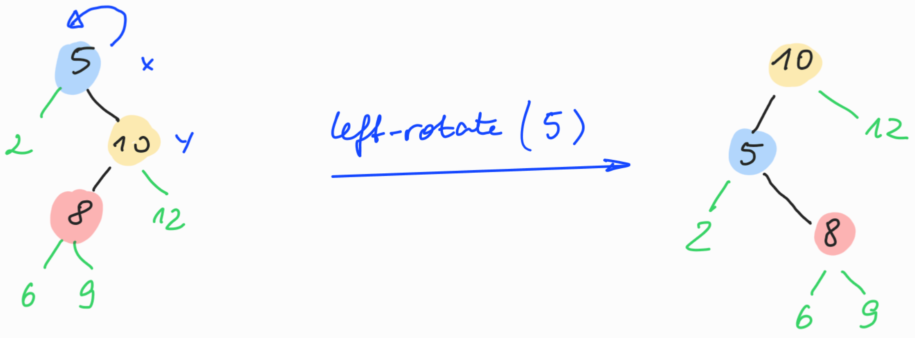
Rotations in BST preserve binary-search tree properies: "$\alpha.keys$" $\leq x.key \leq$ "$\beta.keys$" $\leq y.key \leq$ "$\gamma.keys$"

Note, a rotation is just a "rearrangement" of a constant nr. of pointers and thus runs in $\Theta(1)$ time

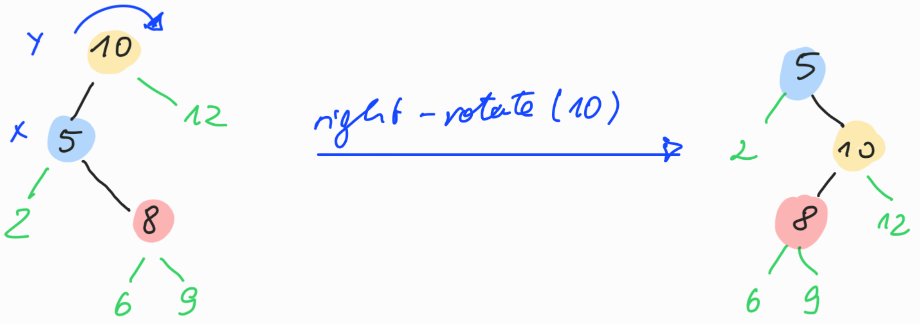
[pseudocode = exercise (don't forget cases as y is right/left of parents_y, x or y is root, $\dots$)]

Example Rotation:

Search tree:



green part "remains unchanged"



Inserting a vertex x to T is done as in case for BST via Tree-Insert(T, z) $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).

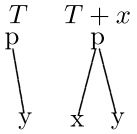
Corrections of tree $T + x$ are based on the following cases for parent p of x in $T + x$.
We denote with $BF_{T'}(v)$ the balance factor of v in tree T'
Since p is parent of x in $T + x$ it can have at most one child (since $T + x$ is binary search tree)

[1] p has right child y but no left child:

Since T is balanced, y must be a leaf and $BF_T(p) = 0 - (-1) = +1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$ (since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T + x$ is an AVL tree. *[nothing to correct]*

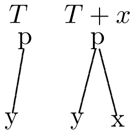


[2] p has left child y but no right child:

Since T is balanced, y must be a leaf and $BF_T(p) = -1 - 0 = -1$

After inserting x: $BF_{T+x}(p) = 0$ and $BF_T(v) = BF_{T+x}(v)$ for all $v \in V(T)$ (since height of $T(v)$ remains unchanged for all $v \in V(T)$)

$\implies$ $T + x$ is an AVL tree. *[nothing to correct]*

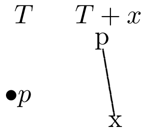


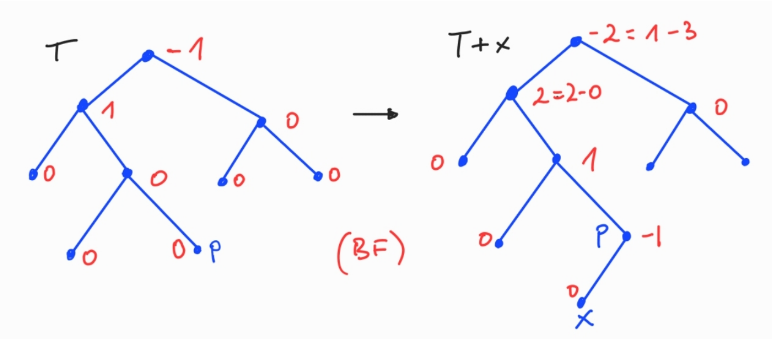
[3] p has no child:

Since p is a leaf, $BF_T(p) = -1 - (-1) = 0$

After inserting x: $BF_{T+x}(p) \in \{+1, -1\}$ and $BF_T(v) \neq BF_{T+x}(v)$ might be possible for $v \in V(T)$

Note, if $BF_T(v) \neq BF_{T+x}(v)$, then v is located on path from ρ to p, i.e., at most $h = O(\log(n))$ vertices could be affected *[possible correction needed]*

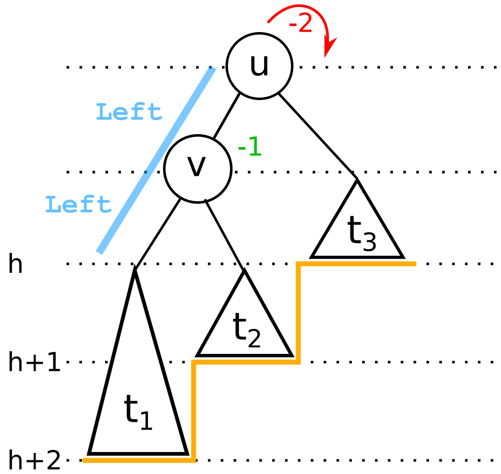




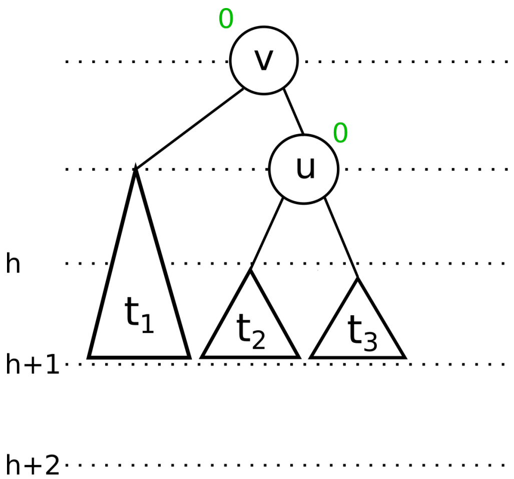
In $BF_{T+x}(v) \in \{-2, -1, 0, +1, +2\}$ since height of T is increased by at most 1 in $T + x$

Let u be a vertex with child v in T where v is located in the subtree with greater height.
4 cases that yield different "types of rotations":

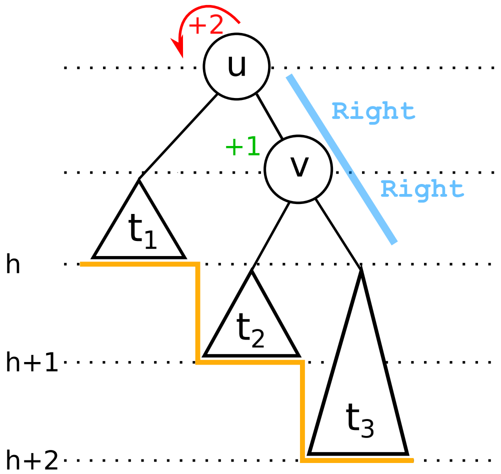
1. $BF_{T'}(u) = -2$, $BF_{T'}(v) \in \{0, -1\}$: **single rotation** $right\_rot(u)$ Left-Left-Case
2. $BF_{T'}(u) = +2$, $BF_{T'}(v) \in \{0, +1\}$: **single rotation** $left\_rot(u)$ Right-Right-Case
3. $BF_{T'}(u) = -2$, $BF_{T'}(v) = +1$: **double rotation** $left\_rot(v) + right\_rot(u)$ Left-Right-Case
4. $BF_{T'}(u) = +2$, $BF_{T'}(v) = -1$: **double rotation** $right\_rot(v) + left\_rot(u)$ Right-Left-Case



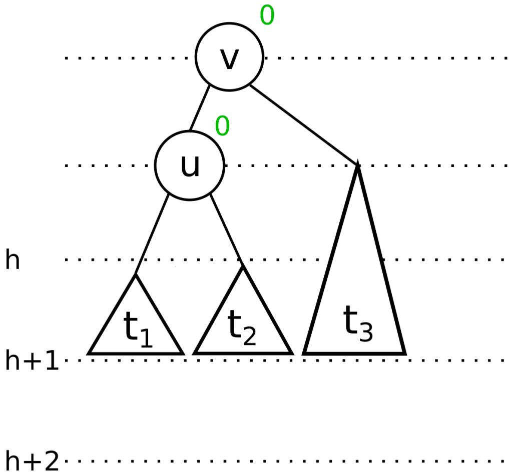
$u - \texttt{Left} - v - \texttt{Left}$



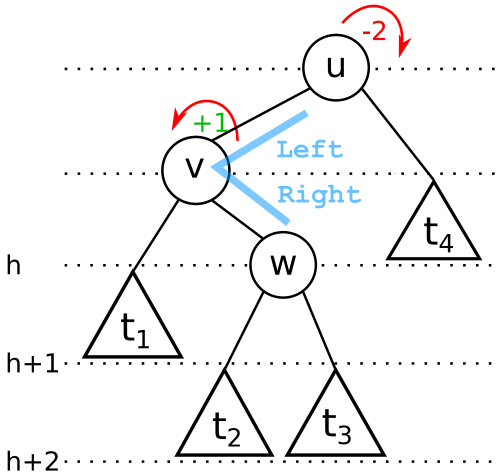
$right\_rot(u)$



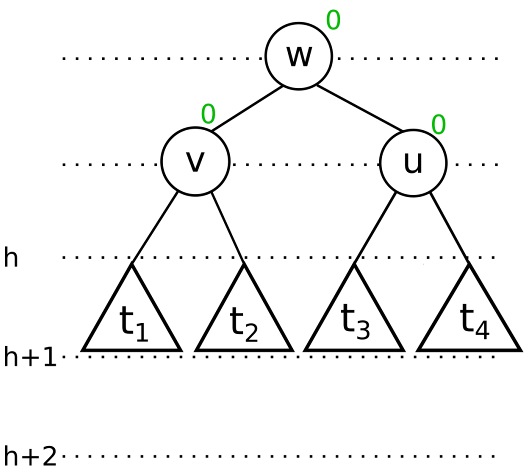
$u - \texttt{Right} - v - \texttt{Right}$



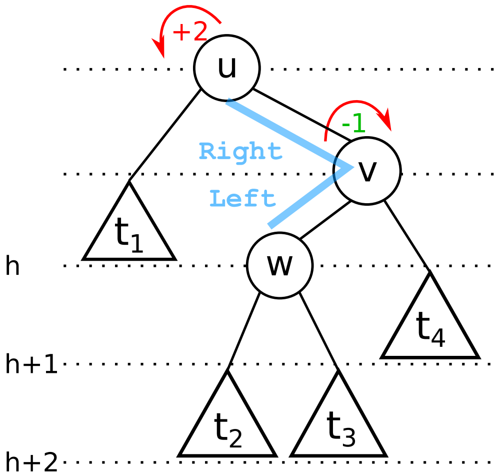
$left\_rot(u)$



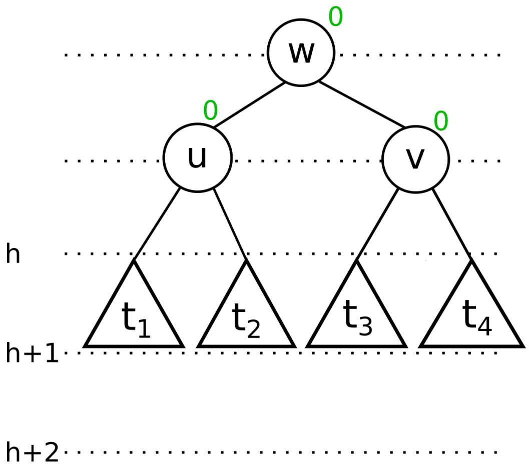
$u - \texttt{Left} - v - \texttt{Right}$



$left\_rot(v) + right\_rot(u)$



u – Right – v – Left



$right\_rot(v) + left\_rot(u)$

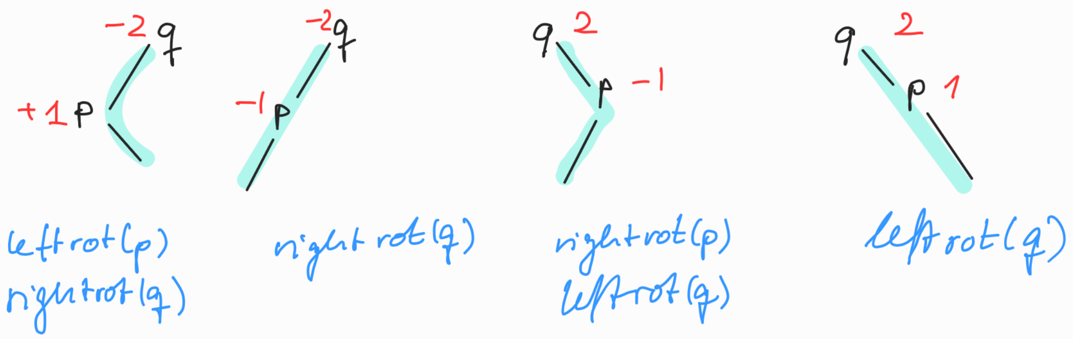
Let $T + x$ be BST obtained from AVL tree after inserting x.

Next "rebalancing" algorithm applied on $T' = T + x$ ensures that the resulting tree is an AVL tree.

```
pseudocode - sketch ReBalance:
  FOR all vertices $u$ on path from $x$ to root (in this order) DO
    • IF $BF_{T'}(u) = -2$ THEN consider left child $v$ of $u$. //Left
        IF $BF_{T'}(v) \leq 0$ THEN $right\_rot(u)$. //Left-Left
        ELSE $left\_rot(v)$ and $right\_rot(u)$. //Left-Right
    • IF $BF_{T'}(u) = +2$ THEN consider right child $v$ of $u$. //Right
        IF $BF_{T'}(v) \geq 0$ THEN $left\_rot(u)$. //Right-Right
        ELSE $right\_rot(v)$ and $left\_rot(u)$. //Right-Left
```

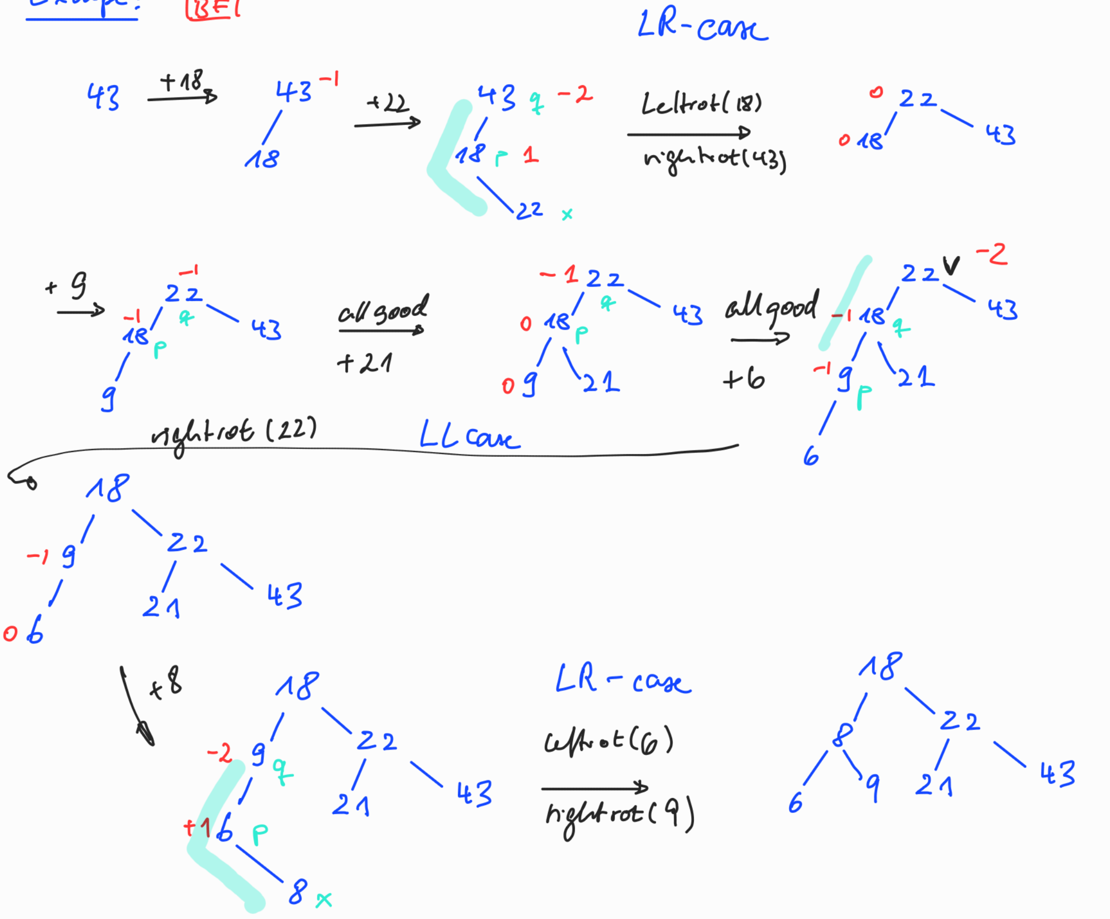
[Example Board]

BF in T+x



insert to AVL

Exmple: BF



RUNTIME-sketch (insert incl. rebalancing):

- Costs for inserting a new vertex: $O(\log(n))$ time.
- Costs for single/double rotation: $O(1)$ time.
- there are at most $h = \log(n)$ vertices along path from x to root.
- determining BFs in $T + x$ can be done in $O(1)$ time

  (since it is determined by the BFs in T *[advanced exercise]*)
- Total time for insert (incl. rebalancing): $O(\log(n))$

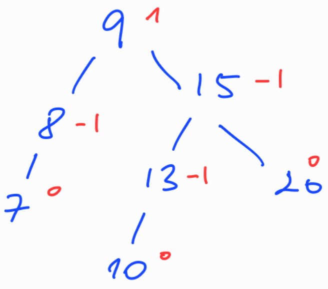
Deleting a vertex x to T is done as in case for BST via `Tree-Delete`(T, z) $\implies$ we get BST $T + x$ which might be imbalanced (corrections!).

Now apply pseudocode - sketch `ReBalance`:

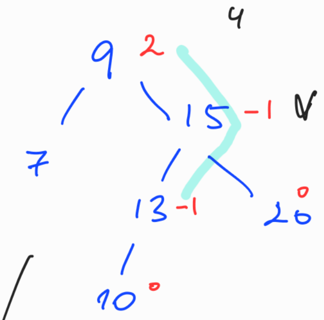
$\implies$ Total time for delete in AVL tree (incl. rebalancing): $O(\log(n))$

[Example Board]

delete from AVL

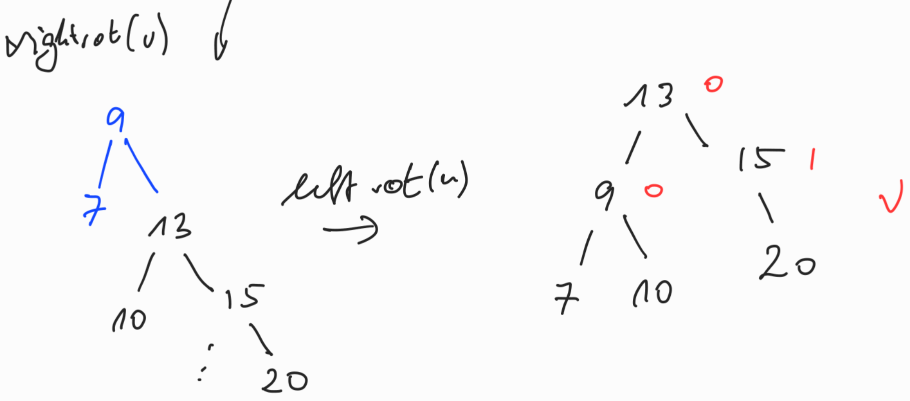


delete 8:



RL-case

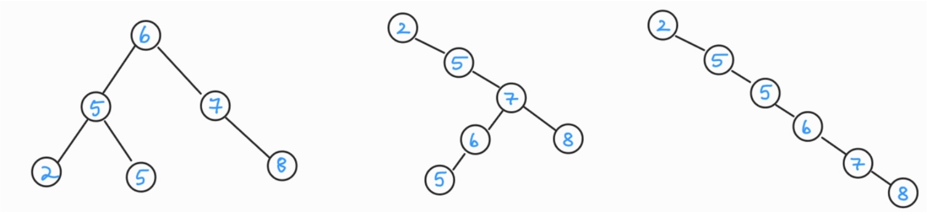
Rightrot (V)
+ left rot (u)



3.2.3 Red-Black Trees

To summarize at this point:

- queries as search, find_minimum, find_maximum as well as insertion, deletion can be done in $O(h)$ time in binary search trees where h = height of tree.
- Problem: $O(h) = O(n)$ where n =number of vertices [can we control the height?]



- Answer: In AVL tree we can control height $h \in O(\log n)$

  We continue now with Red-Black trees that are one of *many* search-tree schemes that are “balanced” in order to guarantee that basic dynamic-set operations take $O(\log n)$ time in the worst case.

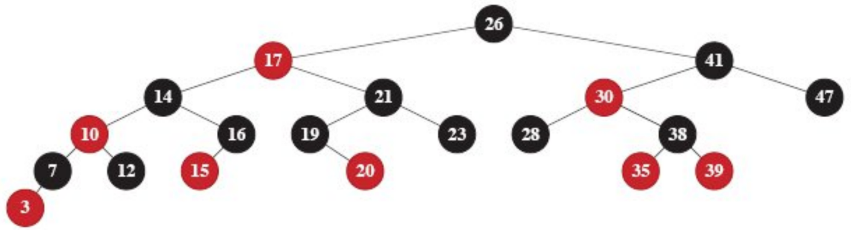
In Red-Black trees we have one extra bit of storage per node: its color, either Red or Black.

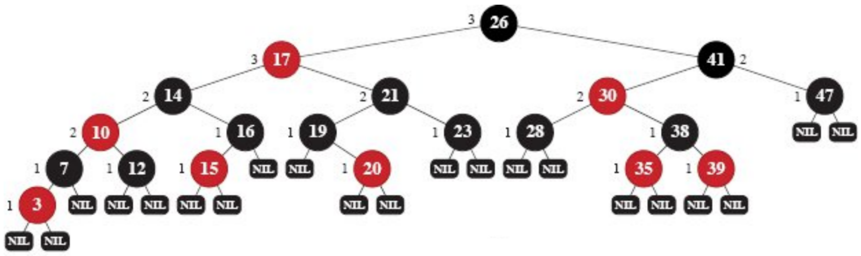
A Red-Black tree is a binary search tree that satisfies the following Red-Black properties:

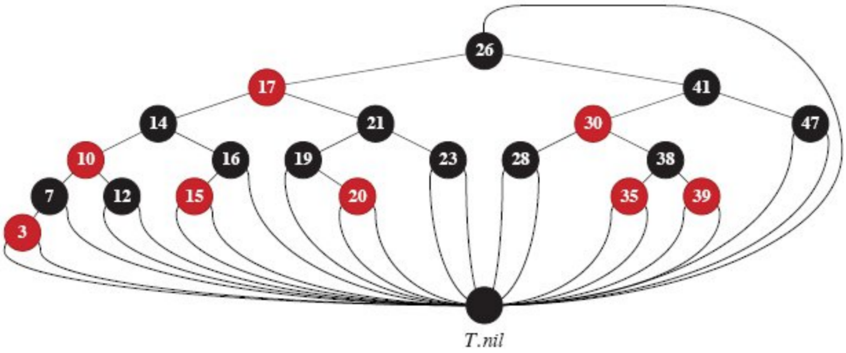
I. (a) Every node is either Red or Black. (b) The root is Black. *(c) Every leaf (T.nil) is* Black.

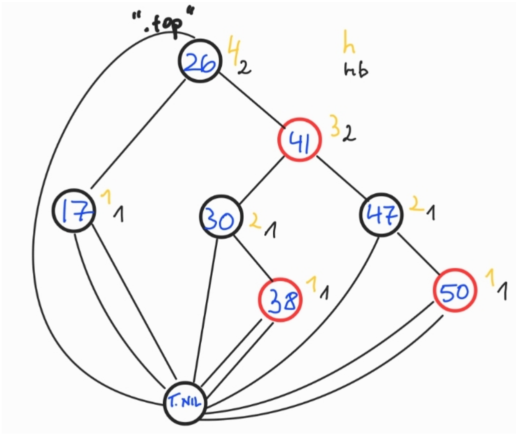
II. If a node is Red, then each of its children must be Black

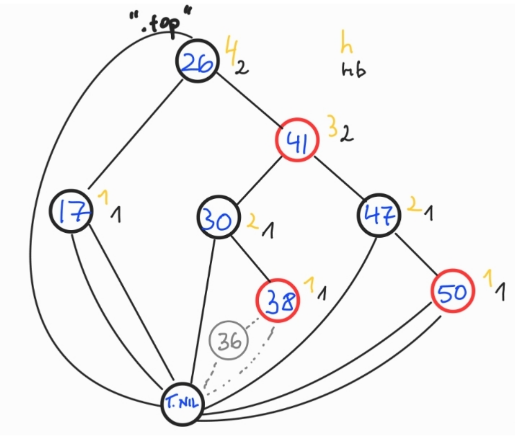
III. For each node x, all simple paths from x to descendant leaves contain the same number of Black nodes.











If a child or the parent of a node does not exist, the corresponding pointer attribute of the node contains the value NIL: Think of these NILs as pointers to leaves (external nodes) of the binary search tree and the normal, key-bearing nodes as internal nodes of the tree. Those NIL-leaves are always supposed to be BLACK.

As a matter of convenience in dealing with boundary conditions and for having simpler code, we use a single **sentinel** $T.nil$ (*sv: väktare*) to represent NIL and that is always of color BLACK and we don't care about $T.nil$s' other attributes $\Longrightarrow$ Red-Black tree is fully binary

black-height $bh(x)$ = number of BLACK nodes on any simple path from, but not including, node x down to a leaf ($T.nil$)

Lemma. *Any Red-Black tree with n internal nodes (=vertices x with $x.key \neq NIL$) has height $O(\log n)$*

[proof board]

RED-BLACK-tree

<u>Lmma</u>: red-black with n internal nodes has height $O(\log n)$

show first:

<u>proof</u>:

$$\boxed{\text{CLAIM. subtree } T(x) \text{ rooted at } x \text{ contains at least } 2^{bh(x)} - 1 \text{ internal nodes.}}$$

By induction on height of x

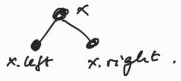
IF $h(x) = 0 \Rightarrow x$ leaf $(= NIL)$

$\Rightarrow bh(x) = 0$

$\Rightarrow 2^0 - 1 = 0$ intern. nodes ✓

Now let $h(x) > 0 \Rightarrow x$ not leaf $(\neq NIL)$

By construction, x must have 2 children.
[NIL possible]



if <u>child z of x</u> black, then z adds "+1" to $bh(x)$
red, then z adds "+0" to $bh(x)$.

$$\Rightarrow bh(z) = \begin{cases} bh(x) - 1, & \text{if } z \text{ black} \\ bh(x), & \text{if } z \text{ red} \end{cases}$$

Note $h(z) < h(x) \Rightarrow$ can apply ind. assumption:

$\Rightarrow T(z)$ has at least $2^{bh(z)} - 1$ nodes

as $bh(z) \geq bh(x) - 1$

$\Rightarrow T(z)$ has at least $2^{bh(x)-1} - 1$ nodes.

Since x has 2 children

$\Rightarrow T(x)$ has at least

$$\underbrace{2}_{\text{2 children}} \cdot \left(2^{bh(x)-1} - 1\right) + \underbrace{1}_{x} \text{ internal nodes}$$

$= 2^{bh(x)} - 1$ internal nodes.

End-proof-of CLAIM

To complete proof of Lemma, let h be height of the tree.

According to property 2 (each red vertex has black children)

$\Longrightarrow \geq \frac{1}{2}$ vertices on any simple path from root to a leaf (not incl. root) must be black

$\Rightarrow bh(root) \geq \frac{h}{2}$

$\xrightarrow{\text{CLAIM}} n \geq 2^{bh(root)} - 1 \geq 2^{\frac{h}{2}} - 1$

$\Rightarrow \log_2(n+1) \geq \frac{h}{2} \Rightarrow h \in O(\log(n))$ □

Thus, queries as search, find_minimum, find_maximum as well as insert and delete can be done in $O(h) = O(\log n)$ time in Red-Black trees. However, after a single of the operations insert and delete it is not ensured that the resulting tree is a Red-Black tree and so, $h > \log(n)$ might be possible.

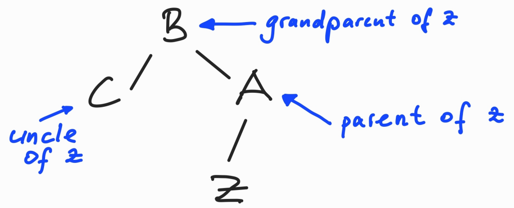
In this case, it is not ensured anymore that the latter operations still run in $O(\log n)$ time
insert key=36: where and which color? Due to Cond. II it must be Black BUT then Cond. III is violated
We show now how to insert into and delete from a Red-Black tree in $O(\log n)$ time while preserving the Red-Black properties.

```
RB-Insert($T, z$)
  1 Tree-Insert($T, z$)
  2 $z.right := T.NIL$ and $z.left := T.NIL$
    //both of $z$'s children are the sentinel
  3 $z.color :=$ RED
  4 RB-Insert-Fixup($T, z$)
```

Insert as in usual binary search tree + Line *2,3,4*

We use RB-Insert-Fixup(T, z) *which is based on re-coloring and rotations to fix issues that may occur in I.b and II.*

Coloring z BLACK would yield other issues that are harder to fix!



Notation in binary tree (left&right not important in this fig!)

Scenarios where z needs some fix up:

I.b z is the root (T was empty at start)

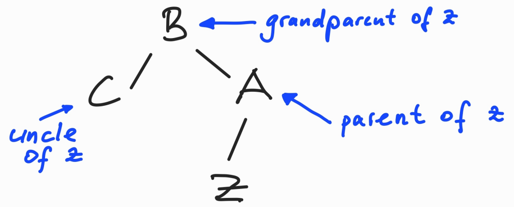
II. parent of z is RED

Then we distinguish:

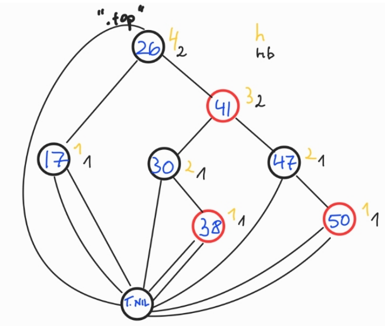
i. uncle of z is RED

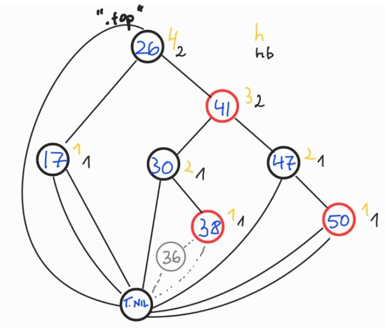
ii. uncle of z is BLACK (triangle)

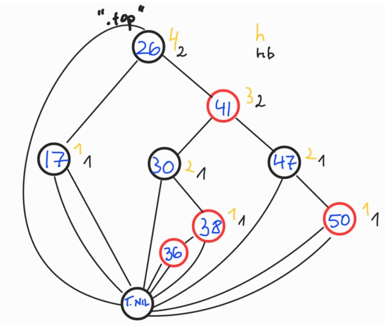
iii. uncle of z is BLACK (line)



Notation in binary tree (left&right not important in this fig!)







Which Red-Black properties could be violated?

I.a Every node is either RED or BLACK. **OK!**

I.b The root is BLACK *OK, unless z is now root.*

I.c Every leaf ($T.nil$) is BLACK. **OK!**

II If a node is RED, then each of its children must be BLACK
violated if $(z.top).color$ is RED

III For each node x, all simple paths from x to descendant leaves contain the same number of BLACK nodes.
OK! hence, $bh(x)$ remains unchanged for all x

Scenarios where z needs some fix up:

I.b z is the root (T was empty at start)

II parent of z is RED

Then we distinguish:

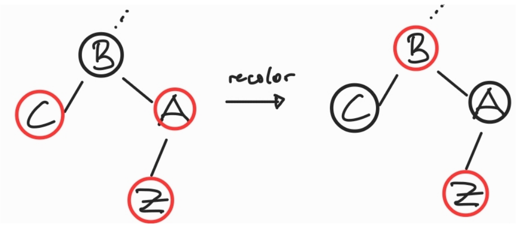
i. uncle of z is RED

ii. uncle of z is BLACK *(triangle)*

iii. uncle of z is BLACK *(line)*

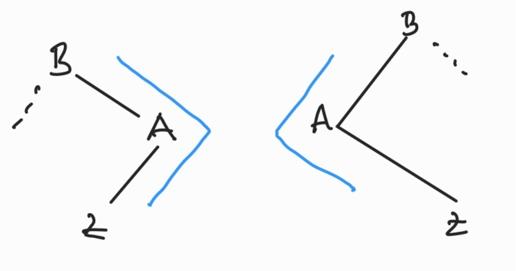
Case (I.b): recolor z to Black $\implies$ we get a valid Red-Black-tree with single root z

Case (II.i):

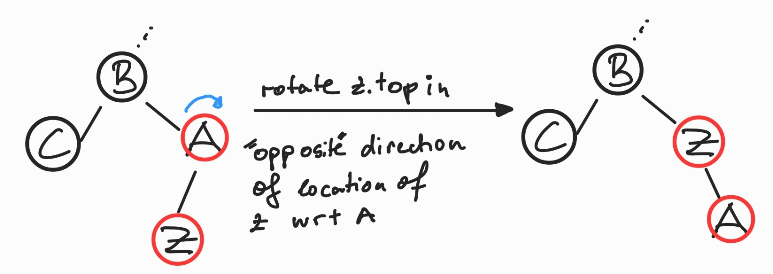


Of course this may cause further violations if parent of B is red, but this will be corrected afterwards.

Case (II.ii):
triangle: either A left of B and z right of A or A right of B and z left of A

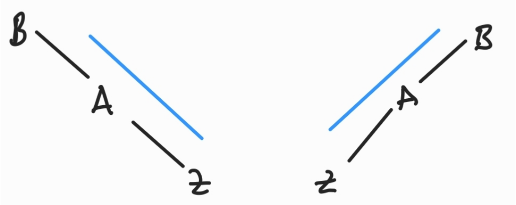


Case (II.ii):

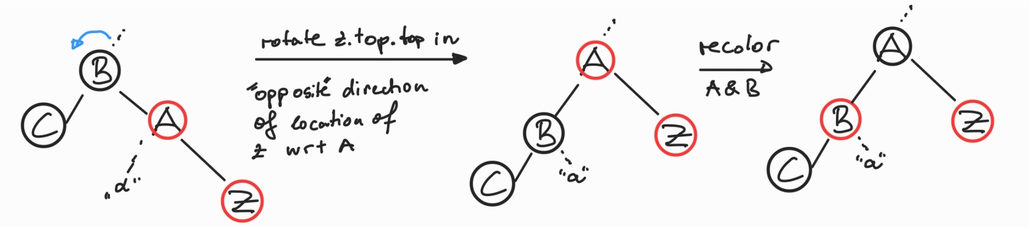


Still II (node is Red, children Black) is violated, but now we are in Case II.iii with A playing the role of z

Case (II.iii):
line: either A left of B and z left of A or A right of B and z left of A



Case (II.iii):
line: either A left of B and z left of A or A right of B and z left of A

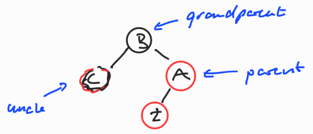


Working Example Board.

RED-BLACK-TREE EXAMPLE (insert)

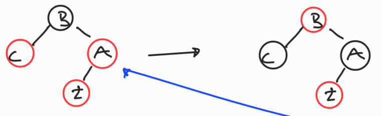
Case 1.b) z root of T ⇒ recolor z to red

2) parent of z is RED:



A, B, C, z are names of nodes, not their values, which are A.key, B.key, C.key, z.key.

i) uncle RED: Recolor A, B, C

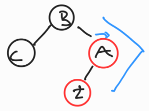


→ recheck properties for B

[Line 8 in cormen: z = z.p.p.]

"z = B"

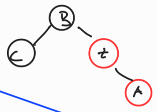
ii) uncle BLACK (triangle)



(or "<")

rotate A → opposite of z's pos. to A.

here: z left of A ⇒ right-rotate (T, A)

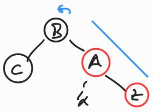


⇒ case (iii) with A playing role of z.

[Line 10 in cormen: z = z.p]

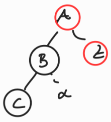
z = A

iii) uncle Black (Line)

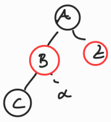


(or "/")

rotate B → in oppos. of z's pos. to A.

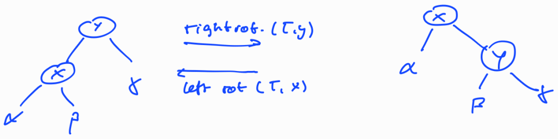


recolor A & B →

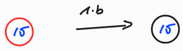


done.

Recap:

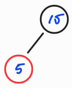


insert 15:



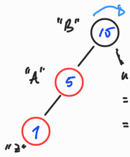
("+ T.NIL")

insert 5:



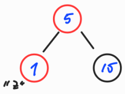
nothing violated!

insert 1:



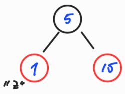
uncle of z "C" = NIL = Black

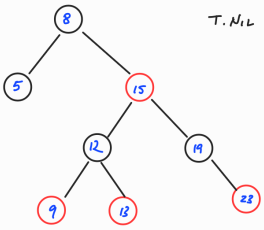
2.iii →



↓ recolor "A" "B"

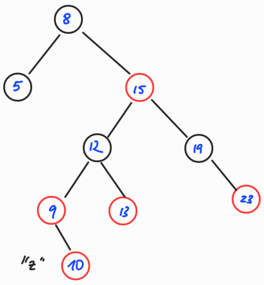
5 15



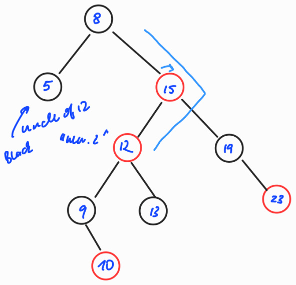


T.NIL omitted

insert 10:

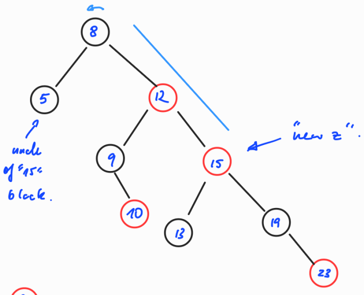


⇒ case 2.i (recolor)

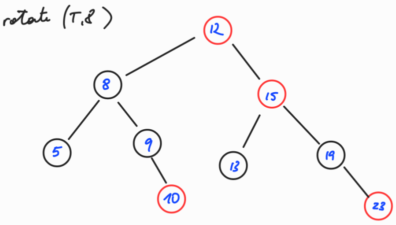


recheck for "new" z = "12"

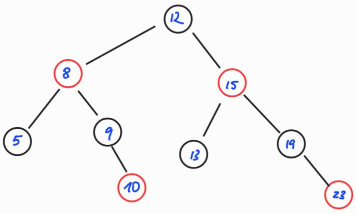
⇒ right rotate (T, 15):



⇒ left rotate (T, 8)



+ recolor:



done.

To summarize in a nutshell:

Insert z as in usual BST and then `RB-Insert-Fixup`(T, t) which is based on 4 scenarios:

I.b z is the root $\Longrightarrow$ Re-color z

II parent of z is Red

i. uncle of z is Red
$\Longrightarrow$ Recolor + Repeat with "new z" (L.8)

ii. uncle of z is Black (triangle)
$\Longrightarrow$ Rotate parent of z + Repeat with "new z" (L.10)

iii. uncle of z is Black (line)
$\Longrightarrow$ Rotate grandparent of z and recolor

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5 //case II.i      z.p.color = BLACK
 6 //case II.i      y.color = BLACK
 7 //case II.i      z.p.p.color = RED
 8 //case II.i      z = z.p.p
 9          else if z == z.p.right
10 //case II.ii         z = z.p
11 //case II.ii         LEFT-ROTATE(T, z)
12 //case II.iii      z.p.color = BLACK
13 //case II.iii      z.p.p.color = RED
14 //case II.iii      RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK //case I.b
```

$v.p$ means parent of v ($=v.top$)

Souce: Introduction to Algorithms (3rd edition), Cormen

Theorem. *Insertion of elements into Red-Black tree while maintaining Red-Black properties can be done* $O(\log(n))$ *time*

Proof. Correctness of `RB-Insert`(T, z) and `RB-Insert-Fixup`(T, z), see Sec 13.3 in Cormen-course-book for more details.

`RB-Insert`(T, z) runs in $O(h(T)) = O(\log(n))$ time.
`RB-Insert-Fixup`(T, z): for each z constant "reassignments" of pointers.
All "new.z" that might cause conflicts and need to be fixed up are ancestors of the "original z" $\Longrightarrow$ While-loop executions: $O(\log(n))$. □

Deletion is much more involved and omitted here, see Sec 13.4 in Cormen-course-book for more details.

3.2.4 Summary

We considered the class of binary search trees (BST).
In particular, we had a closer look to the subclasses:

- AVL trees
- Red-Black trees

Question: when using AVL tree, when Red-Black trees?

Since the invention of AVL trees in 1962 and Red-black trees in 1978, researchers were divided in two separated communities, AVL supporters and Red-Black ones.

Often, AVL trees are used for retrieval applications (Search Engines, Database queries) whereas Red Black trees are used in updates operation (insertion, replace information)

Worst case	**AVL**	**Red-Black**
Height	1.44 Log (n)	2 log (n+1)
Updates complexity	O(Log (n))	O(Log (n))
Retrieval Complexity	O(Log (n))	O(Log (n))
Rotations for insert	2	2
Rotations for delete	Log (n)	3

AVL and Red-Black tree as a single balanced tree, Bounif and Zegour, Proc. of the Fourth Intl. Conf. Advances in Computing, Communication and Information Technology, 2016

Chapter 4

Hashing

4.1 The Idea and Notation

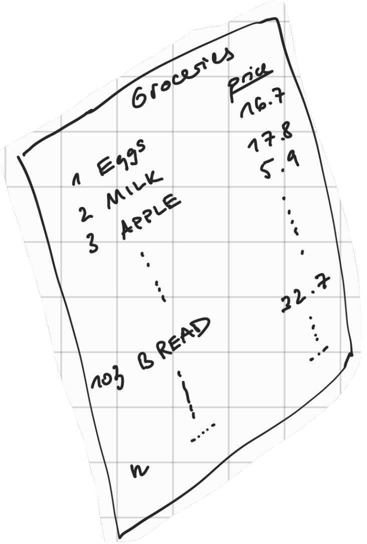
- Many applications require dictionary, i.e., a dynamic set that supports only the operations INSERT, SEARCH, DELETE.

- A hash table is an effective data structure for implementing dictionaries.

  Side note: the built-in dictionaries of Python are implemented with hash tables.

- Although searching for an element in a hash table can take as long as searching for an element in a linked list - $\Theta(n)$ time in the worst case - in practice, hashing performs extremely well.

  Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.



POV: You work at a grocery store and have this
"list" of items in unsorted order together with prices.

When a customer buys products, you have to look up the prices in this huge list of n elements.

How much is a bread? $O(n)$ time :(!

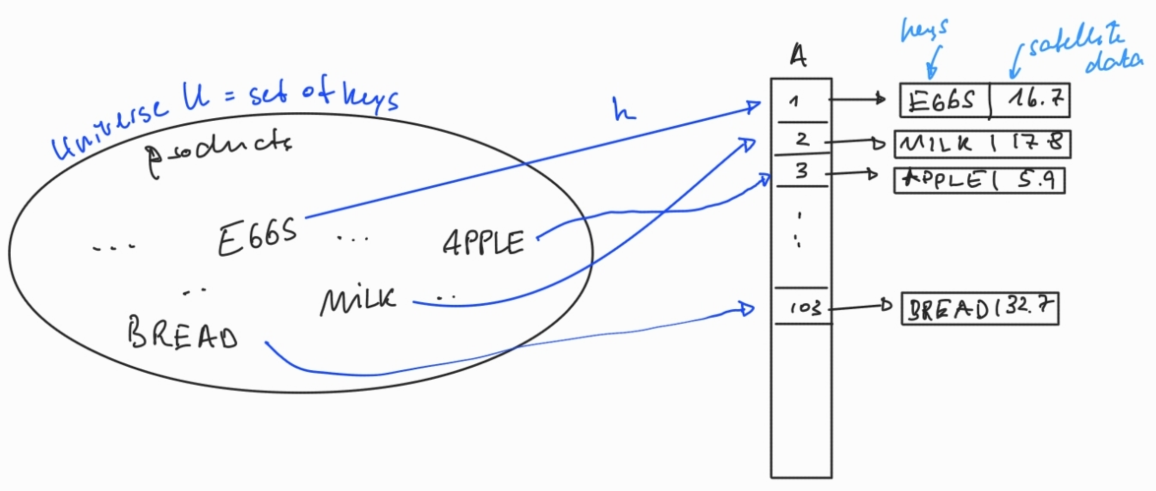
What if we have a function

$$h\colon \text{products} \to \text{listNr}$$

$$\begin{aligned}&\text{EGGS} \mapsto 1\\ &\text{MILK} \mapsto 2\\ &\vdots\\ &\text{BREAD} \mapsto 103\\ &\vdots\end{aligned}$$

Then, look up at takes $O(1)$ time

This is the essential idea of hashing, i.e, the function h maps keys (here: products) together with their satellite data (here: prices) to some entry in an array (hash table).



Idea: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \leq |U|$

Interpreting keys as numbers

- Most hash functions assume that the universe of keys is the set $\mathbb{N}_0 = \{0, 1, 2, \ldots\}$
- If that is not the case, we need to interpret (injective mapping) them as numbers

Example: A string $s = s_0 s_1 \ldots s_r$ of characters may be interpreted as a number in the following way:

- Look up ascii-code of each character (`www.ascii-code.com`)
- Each character s_i is associated with a number a_i in $\{0, \ldots, 127\}$
- Then s can be interpreted as a number k in a base 128 system:

$$k = \sum_{i=0}^{r} a_i \cdot 128^i$$

- Example: for string $s =$`Mia` we have `M` $= 77$, `i` $= 105$, `a` $= 97$ (*ASCII*)
  Thus, `Mia` corresponds to number $k = 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$

In what follows, we (mainly) assume that the keys are natural numbers.

Let $T[0..m-1]$ be an array of size m (hash table).

Let U be the "universe" of all potential keys. In practice, we are often interested in subsets $K \subseteq U$ only.

The size m of the hash table is typically much less than $|U|$.

A hash function

$$h\colon U \to \{0, \ldots, m-1\}$$

is a map that assigns to each key $k \in U$ a number $h(k) \in \{0, \ldots, m-1\}$ as a potential slot in the hash table T.

We say that an element with key k hashes to slot $h(k)$ and that $h(k)$ is the hash value of key k.

Since usually $m < |U|$, it is easy to see that h is not necessarily injective (collisions)!

4.2 Types and choice of hash functions

There are different ways of defining hash functions. The typical ways are:

- Direct Addressing (perfect hashing)
- “Non-Direct” Addressing
  - Division Method
  - Multiplication method
  - Random Method
  - Selecting hash function from a set of well-designed hash functions

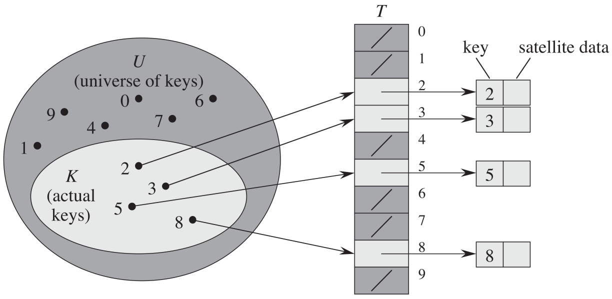
4.2.1 Direct Addressing

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

Let U be the universe and suppose $|U|$ that is not too large

Exmpl: $U =$ set of all possible membership IDs in sports club in a town of a small number $|U|$ of people. (no two members have the same ID)

Init hash table $T[0..|U|-1]$.



Choose $h\colon U \to \{0, \ldots, |U|-1\}$ as a bijective map (e.g. via total/lexicographic order order on elements in U).

We are interested in $K \subseteq U$ (e.g. all those people in the town that actually are member of sports club.)

Hence, $h\colon K \to \{0, \ldots, |U|-1\}$ is injective, and

$T[h(k)]$ points to the element with key k or is NULL, when this key is not in the set

This allows an $O(1)$ time search (called perfect hashing)

Problems:

- If U gets large or infinite, storing an array T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer. (e.g. $U = \{$all phone numbers$\}$)
- If actual sets $K \subset U$ of used keys are much smaller than U, then a direct-address table is a big waste of space

Solution: Use hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

4.2.2 "Non-Direct" Addressing

If direct addressing is not feasible, use a hash function $h\colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$

A good hash function should be as easy and fast to compute as possible and evenly distribute the data records to be stored across the hash table to avoid collisions.

In other words, a good hash function satisies (approximately) the assumption of simple uniform hashing:
Each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to

Unfortunately, you typically have no way to check this condition, unless you happen to know the probability distribution from which the keys are drawn. Moreover, the keys might not be drawn independently (e.g. $U = \{$all phone numbers$\}$ and two phone numbers $k_1 \neq k_2$ often share the same prefix, the area code).

Choose m in the order of the number of elements expected to be stored ["good" m depend on h]

Division Method

A natural way is the **division method** where a hash function is maps a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is

$$h(k) = k \bmod m$$

Example: If $m = 12$ and key is $k = 100$, then $h(k) = 4$.

Since it requires only a single division operation, hashing by division is quite fast.

Problems one needs to be aware of:

- If $m = 2^p$, then $h(k)$ depends only on the last p bits of k.

  Hence, all keys that agree in the last p bits hash to the same slot.

  Depending on distribution of keys, performance of such a hash table may be bad.

  Example: $m = 2^2 = 4$, then last 2 bits of a number in binary representation are always one of $00, 01, 10, 11$ and we have

  $$\begin{aligned} k \bmod 4 = 0 &\text{ iff last 2 bits are 00}\\ k \bmod 4 = 1 &\text{ iff last 2 bits are 01}\\ k \bmod 4 = 2 &\text{ iff last 2 bits are 10}\\ k \bmod 4 = 3 &\text{ iff last 2 bits are 11}\end{aligned}$$

- Suppose, $m = 2^p - 1$ and we are hashing strings with the division method and are interpreting them as base 2^p numbers

  Then a string $s = s_r \cdots s_1 s_0$ in which each character s_i is interpreted as a number in $\{0, \ldots, 2^p\}$ hashes to

  $$\begin{aligned} h(s) &= \left(\sum_{i=0}^{r} s_i \cdot (2^p)^i\right) \bmod (2^p - 1) \qquad //(2^p)^i \bmod (2^p-1) = 1 + \text{mod-rules}\\ &= \left(\sum_{i=0}^{r} s_i\right) \bmod (2^p - 1)\end{aligned}$$

  Thus, the hash value of a string is *invariant against permutations* of the string.

  Example: $s =$`Mia` via ASCII: `M` $= 77$, `i` $= 105$, `a` $= 97$

  $\Longrightarrow\ s \mathrel{\widehat{=}} 77 + 105 \cdot 128 + 97 \cdot 128^2 = 1602765$ and $s' =$`iMa` $\mathrel{\widehat{=}} 105 + 77 \cdot 128 + 97 \cdot 128^2 = 1599209$

  $\Longrightarrow\ h(s) = 1602765 \bmod 127 = 25 = h(s') = 1599209 \bmod 127$

Solutions to the latter problems are provided by carefully choosing m.
Usually m is a particular well-chosen prime number = number theory (not part of this course)

Further Methods

Further ways:

- **multiplication method** where a hash function is created that maps a key k into one of m slots by using

$$h(k) = \lfloor m(k\phi \bmod 1) \rfloor,$$

  where $\phi \in (0, 1)$ is an (irrational) number and "$k\phi \bmod 1$" means the fractional part of $k\phi$, that is, $k\phi - \lfloor k\phi \rfloor$

- **random method**, e.g., by using key k as the seed for a random number generator producing a number between $0, 1, \ldots, m-1$

- **Selecting $h \in \mathcal{H}$**: randomly pick h from a precomputed and carefully designed set $\mathcal{H}$ of hashfunction

Summary

The behavior/performance of a hash function depends on the chosen set of keys.

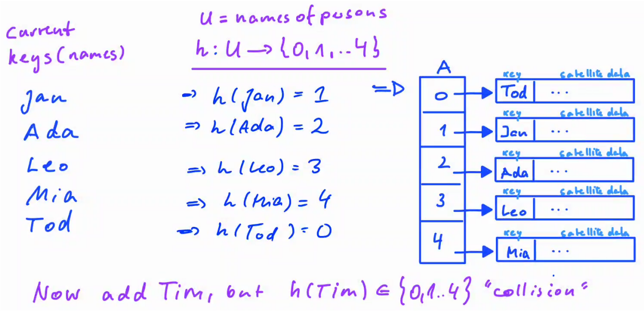
- Therefore, they can only be insufficiently studied theoretically or with the help of analytical models.
- Given a hash function, it is always possible to find a set of keys for which it generates many collisions.
- No hash function is always better than all others.

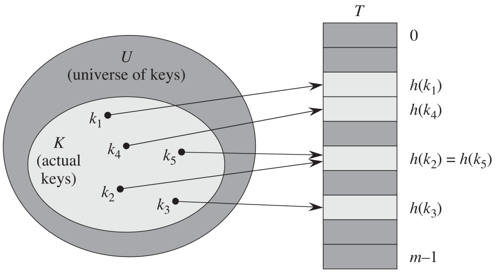
However, there are several empirical studies on the quality of different hash functions.

- The division method is generally the most efficient; however, for certain sets of keys, other techniques may perform better.
- If the key distribution is unknown, then the division method is the preferred hashing technique.
- Important: Use sufficiently large hash table and use a prime number as divisor.
- Moreover, hashing is, in practice not based on a fixed hash function, but on carefully designed sets $\mathcal{H}$ of hash functions from which we randomly pick one.

4.3 Collisions

If direct addressing is not feasible, use a hash function $h \colon U \to \{0, \ldots, m-1\}$ where $m \ll |U|$





Problem: different keys may receive the same h value (collision)

Question: How likely is it that two keys receive the same h value assuming that $h(k)$ is randomly chosen for all keys k?

A problem that is similar in fashion:

Birthday problem: what is the probability that, in a set of ℓ randomly chosen people, at least two will share a birthday.

Here $m = 365$ and, say $\ell = 23$ persons (keys) – *What is your gut feeling ?*

There are $\prod_{i=1}^{23} 365 = 365^{23}$ variations of all-birthday-person-cases (all cases equality likely).

Let us consider the nr c of cases that all persons have a birthday on different days

(1st person 365 possibilities, 2nd person remaining 364 days, . . .):

$$c = 365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343$$

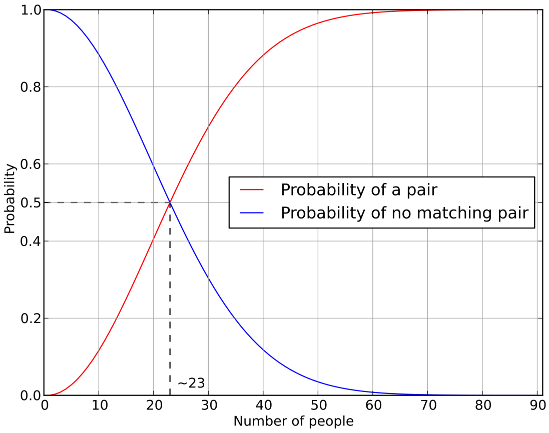
Probability p that all 23 persons have a birthday on different days

$$p = \frac{365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343}{365^{23}}$$

Probability p' that there are at least two persons having the same birthday

$$p' = 1 - \frac{365 \cdot 364 \cdot 363 \cdot \cdots \cdot 343}{365^{23}} \approx 0.51$$

With a 51% chance two elements collide when $\ell = 23 \ll m = 365$



source: wikipedia (red-line: at least 2 have same birthday, blue-line: all different birthdays)

Even for $\ell = 23$ keys and $m = 365$ slots the chances are 51% to have collisions
$\Longrightarrow$ *collisions MUST BE addressed!*

A collision occurs when two keys hash to the same slot.

Question:

- How to deal with and to resolve collisions?

There are two main approaches:

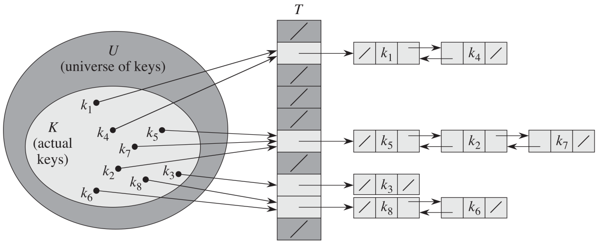
- Resolving collision via chaining
- Resolving collision via open addressing

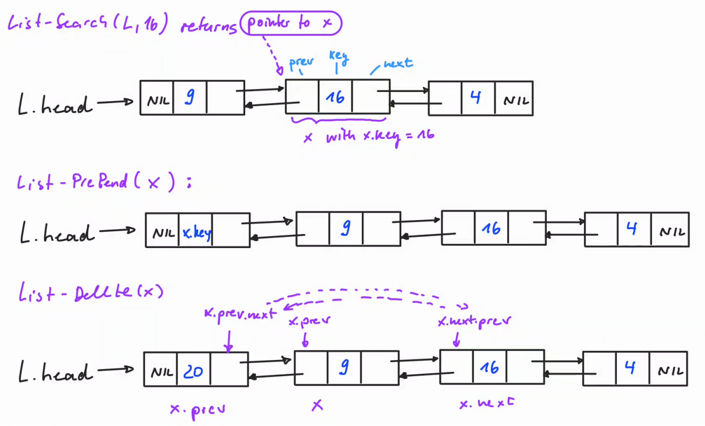
4.4 Resolving collision

4.4.1 Resolving collision via chaining

Chaining:

- $T[j] := NIL$ if no element in the set has a key k with Hashing $h(k) = j$
- Otherwise, store at $T[j]$ the pointer to the head of a doubly-linked list of all such elements





basic operations on doubly-linked lists L and hash-tables T:

```
List-Search(L,k) //finds the first element with key k in list L by a simple linear search
                                                       returning a pointer to this element
    $x := L.head$ //$x$ is pointer to first element in $L$
    WHILE $x \neq NIL$ and $x.key \neq k$ DO
        $x := x.next$
    return $x$

Chained-Hash-Search(T,k)
    return List-Search(T([h(k)]),k)      //Takes time proportional to number of elements in list T[h(k)].
```

```
List-PrePend($L$,$x$) //inserts object/element $x$ to the front of $L$
    $x.next := L.head$ //$x$ is pointer to first element in $L$
    $x.prev := NIL$
    IF $L.head \neq NIL$ THEN $L.head.prev := x$
    $L.head := x$

Chained-Hash-Insert($T$,$x$)
    List-PrePend($T[h(x.key)]$,$x$) //Takes constant time if we assume that $x$ is not already in the hash table
                                    // (if not known, do a search first).
```

```
List-Delete($L$,$x$)
//deletes object/element $x$ from $L$ (It must be given a pointer to $x$)
    IF $x.prev \neq NIL$ THEN $x.prev.next := x.next$
    ELSE $L.head := x.next$
    IF $x.next \neq NIL$ THEN $x.next.prev := x.prev$
```

To delete an element with a given key k, fist call List-Search(L,k) to retrieve a pointer to the element

```
Chained-Hash-Delete($T$,$x$)
    List-Delete($T[h(x.key)]$,$x$)          //Takes constant time if we have pointer to $x$
                                            // (if pointer not known, do a search first).
```

How well does hashing with chaining perform? In particular, how long does it take to search for an element with a given key?

Given: A hash table T with m slots that stores n elements

Worst case: h maps all n element to same slot (that is, the list to which $T[i]$ points to for some i)
$\implies$ seaching takes $O(n)$ time + time to compute hash function
$\implies$ no better than using one linked list for all the elements.

Average case is more interesting !

We define the load factor α for T as

$$\alpha = n/m,$$

that is, the average number of elements stored in a list $T[i]$, $i \in \{1, \dots, m\}$.

The average-case performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

A hash function h is **simple uniform**:

- Any given element is equally likely to hash into any of the m slots and
- where a given element hashes to is independent of where any other elements hash to.

Theorem. *In hashing with chaining a search takes average-case time $\Theta(1+\alpha)$ under the simple uniform hashing assumption.*

[proof omitted - see Cormen course book]

$\implies$ if $n \leq m$ then $\alpha \leq 1$ and thus, $\Theta(1+\alpha) = \Theta(1)$
$\implies$ Insert-, Delete-, Search-operations take constant *expected time* and
Insert- and Delete-operations take constant time in *the worst-case*

4.4.2 Resolving collision via open addressing

In contrast to chaining, in open addressing, all elements occupy the hash table itself. That is, each hash table entry contains either an element of the dynamic set or NIL.

The idea: when trying to enter the key k into the hash table at position $h(k)$ and it is discovered that $T[h(k)]$ is already occupied, then – according to a fixed rule – unoccupied space (an open one) is used to accommodate k.

Since you cannot know in advance which locations will be occupied and which will not, you define an order for each key in which all storage locations are viewed, one after the other. As soon as a space in question is free, the key is stored there.

In open addressing, the hash function h is a function

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

such that $(h(k,0), h(k,1), \ldots, h(k,m-1))$ is a permutation of $(0, 1, \ldots, m-1)$ for every $k \in U$, called the probe sequence.

Examples ($k \in U, i \in \{0, \ldots, m-1\}$ and $h' \colon U \to \{0, \ldots, m-1\}$ is "auxiliary" hash functions):

- linear probing uses hash function $h(k,i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant

Example (Linear Probing)
Insert in order: 79, 28, 49, 88, 59 into hash table of size $m = 10$.

Assume: $h'(k) = k$ and $a = 1$ (just for simplicity)
Recall: $h(k,i) = (h'(k) + i) \bmod 10,\ i = 0, 1, 2, \ldots$

0	1	2	3	4	5	6	7	8	9

We now add 79 and then 28

0	1	2	3	4	5	6	7	8	9
								28	**79**

$h(79,0) = (79+0) \bmod 10 = 9 \Rightarrow$ slot 9 unoccupied $\Rightarrow T[9] = 79$

Then, $h(28,0) = (28+0) \bmod 10 = 8 \Rightarrow$ slot 8 unoccupied $\Rightarrow T[8] = 28$

0	1	2	3	4	5	6	7	8	9
49								28	79

Since $h(49,0) = (49+0) \bmod 10 = 9$ and slot 9 is already occupied, we have a collision.

Since $h(49,1) = (49+1) \bmod 10 = 0$ and slot 0 is unoccupied $\Rightarrow T[0] = 49$

0	1	2	3	4	5	6	7	8	9
49	**88**							28	79

$h(88,0) = (88+0) \bmod 10 = 8$ occupied!
$h(88,1) = (88+1) \bmod 10 = 9$ occupied!
$h(88,2) = (88+2) \bmod 10 = 0$ occupied!
$h(88,3) = (88+3) \bmod 10 = 1$ unoccupied!

0	1	2	3	4	5	6	7	8	9
49	88	**59**						28	79

$h(59,0) = (59+0) \bmod 10 = 9$ occupied!
$h(59,1) = (59+1) \bmod 10 = 0$ occupied!

$h(59, 2) = (59 + 2) \bmod 10 = 1$ occupied!
$h(59, 3) = (59 + 3) \bmod 10 = 2$ unoccupied!

Examples ($k \in U, i \in \{0, \ldots, m-1\}$ and $h', h'' \colon U \to \{0, \ldots, m-1\}$ are "auxiliary" hash function):

- linear probing uses hash function $h(k, i) = (h'(k) + a \cdot i) \bmod m$, $a \neq 0$ constant
- quadratic probing uses hash function $h(k, i) = (h'(k) + a \cdot i + b \cdot i^2) \bmod m$ where a and $b \neq 0$ are constants.
  - *a, b must be chosen such that the probe sequence is indeed a permutation of* $(0, \ldots, m-1)$.
- double hashing uses the hash function $h(k, i) = (h'(k) + i \cdot h''(k)) \bmod m$
  - *(linear probing is just a special case where $h''(k) = a$ for all k)*
  - *$h''(k)$ must be relatively prime to m so that all slots are probed.*

We omit further theoretical aspects of runtime at this point.

4.4.3 Resolving collision: summary

Two different approaches for resolving collisions: chaining and open addressing (e.g. linear probing).

Which one is better? This question is beyond theoretical analysis, as the answer depends on the intended use and many technical parameters[*].

A disadvantage of open addressing is that search times become high when the number of elements approaches the table size. For chaining, the expected access time remains small. On the other hand, chaining wastes space on pointers that open addressing could use for a larger table.

However, experimental results show that both techniques performed almost equally well when they were given the same amount of memory.

4.5 Factors Affecting Hash Function Performance

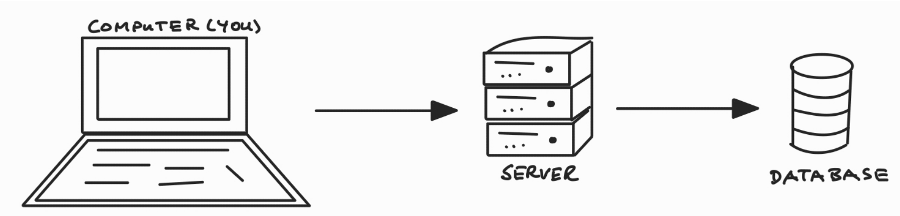
The efficiency of a hash function depends on many factors and parameters!

- Type of hash function
- Data type of the key space: Integer, String, ...
- Distribution of currently used keys
- Load factor α of the hash table
- Number of records that can be stored at an address without causing collisions (List capacity)
- Collision resolution technique
- Possibly, the order of storing the records (open addressing)

[*] Algorithms and Data Structures - The Basic Toolbox, Mehlhorn and Sanders, Springer, 2008

4.6 An application of Hashing: Bloom Filters

Bloom filters are probabilistic data structures based on hashing to check memberships in sets.
Aim: avoid time-consuming queries.
Example: Suppose you want to register to a web-service and are ask to provide a username.



Question: Does your username already exist?
Answer: check in database – can be very time-consuming.
Solution: Bloom-filter "If we can say for sure that username does not exist we can skip time-consuming query!"

A Bloom filter $(B, \mathcal{H})$ consists of

- an array B of m bits, initially all set to 0 and
- a set $\mathcal{H} = \{h_1, \ldots, h_k\}$ of k independent hash functions all in range $\{0, \ldots, m-1\}$.

A Bloom filter $(B, \mathcal{H})$ represents a set $S = \{x_1, x_2, ..., x_n\}$ if, for each element $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$.

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0,0,0,0,0,0,0,0]$, and $\mathcal{H} = \{h_1\}$

$$\begin{array}{lll}\text{Add Jan: } h_1(Jan) = 1 & \Longrightarrow & B = [0,1,0,0,0,0,0,0]. \\ \text{Add Ada: } h_1(Ada) = 2 & \Longrightarrow & B = [0,1,1,0,0,0,0,0]. \\ \text{Add Leo: } h_1(Leo) = 3 & \Longrightarrow & B = [0,1,1,1,0,0,0,0]. \\ \text{Add Tod: } h_1(Tod) = 0 & \Longrightarrow & B = [1,1,1,1,0,0,0,0].\end{array}$$

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$.
Since $B[4] = 0$, we can say for sure that "Mia" does not exist and we don't need to look up the database S!

Now we want to check if username "Tim" exists and say we have $h_1(Tim) = 1$.
Although $B[1] = 1$, we cannot say with certainty that "Tim" exists or not due to possible collisions and we must look up S!

Comic Example: $S = \{Jan, Ada, Leo, Tod\}$, $B[0..7] = [0,0,0,0,0,0,0,0]$, and $\mathcal{H} = \{h_1, h_2\}$

$$\left.\begin{array}{l} h_1(Jan) = 1, h_2(Jan) = 5 \\ h_1(Ada) = 2, h_2(Ada) = 7 \\ h_1(Leo) = 3, h_2(Leo) = 5 \\ h_1(Tod) = 0, h_2(Tod) = 3 \end{array}\right\} \Longrightarrow B = [1,1,1,1,0,1,0,1]$$

$(B, \mathcal{H})$ represents S.

Now we want to check if username "Mia" exists and say we have $h_1(Mia) = 4$ and $h_2(Mia) = 6$.
Again, since $B[4] = 0$, we can say for sure that "Mia" does not exist and we don't need to look up the database S!

Now we want to check if username "Tim" exists and say we have $h_1(Tim) = 1$ and $h_2(Tim) = 6$. Although $B[1] = 1$ we have $B[6] = 0$ and we can say for sure that "Tim" does not exist and we don't need to look up S!

In summary, if we want to check membership:

- If $B[h_i(x)] = 0$ for at least one i, then it is ensured that $x \notin S$.
- Otherwise (i.e., $B[h_i(x)] = 1$ for all i with $1 \leq i \leq k$), we must compare all elements in S with x to verify if $x \in S$ or $x \notin S$.

Question: What is the false-positive rate and how can we minimize it?

Important Observation:

- We can never have false-negatives (0 bit in B leads never to wrong conclusion that key is in S)!
- However, false-positives are possible (key is "possibly" in the S (bit in $B = 1$), but it's not.)
  To decrease false-positive rate use more hash-functions!

False-positive rate.

We assume that all hash functions $h \in \mathcal{H}$ are simple uniform and independent from each other.

$\frac{1}{m}$	probability that $h(x) = i$ and thus, $B[h(x)] = 1$ for $h \in \mathcal{H}$
$1 - \frac{1}{m}$	probability that $B[i]$ is not set to 1 by a given $h \in \mathcal{H}$
$(1 - \frac{1}{m})^k$	probability that $B[i]$ is not set to 1 by any of the k hash-functions $h \in \mathcal{H}$

From analysis: $$\frac{1}{e} = \lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m \implies \left(1 - \frac{1}{m}\right)^k = \left(1 - \frac{1}{m}\right)^{\frac{km}{m}} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{\frac{k}{m}} \overset{\text{for large } m}{\approx} \left(\frac{1}{e}\right)^{\frac{k}{m}} = e^{-\frac{k}{m}}$$

$(1 - \frac{1}{m})^{kn} \approx e^{-\frac{kn}{m}}$	probability that $B[i]$ is not set to 1 by any $h \in \mathcal{H}$ after "adding" the n elements from S
$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$	probability that $B[i] = 1$ after "adding" the n elements from S
$\epsilon = \left(1 - e^{-\frac{kn}{m}}\right)^k$	probabiliy that all k entries $B[h_i(x)] = 1$ for x (this could cause the algorithm to erroneously claim that the element is in the set)

Observation: ϵ decreases with increasing m (more slots) and increases with with increasing n (more elements)

Assume we have $|S| = n$ and some m.
What is an optimal number k of hash functions?: $\frac{d\epsilon}{dk} = 0 \implies k = \frac{m}{n} \ln(2)$

Assume we have $|S| = n$, a desired false-postive rate ϵ and an optimal k.
What is the required number m of bits?: replace k by $\frac{m}{n} \ln(2)$ in $\epsilon \implies m = \frac{-n \ln(\epsilon)}{\ln(2)^2}$
($\epsilon < 1$, so $-n \ln(\epsilon)$ is positive)

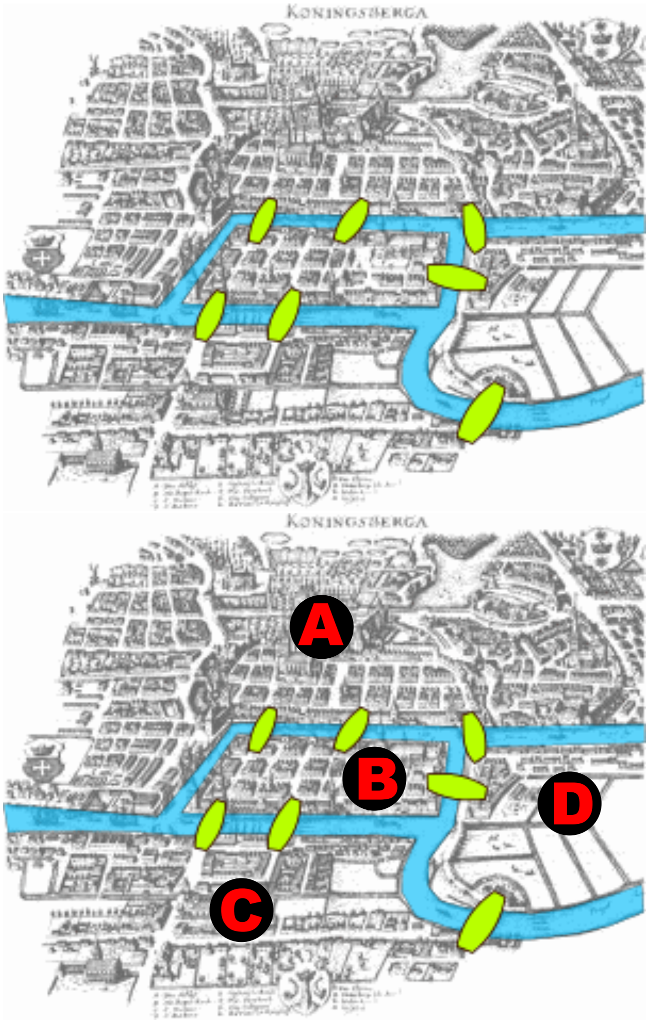
Example: $|S| = 1000$ and want $\epsilon = 0.1$ then we should choose $m \approx 4800$ and $k \approx 3$

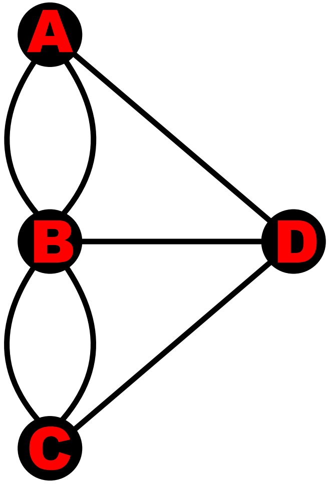
Chapter 5

Elementary Graph Algorithms

5.1 Intro and Basics

Seven Bridges of Königsberg (Euler, 1736)

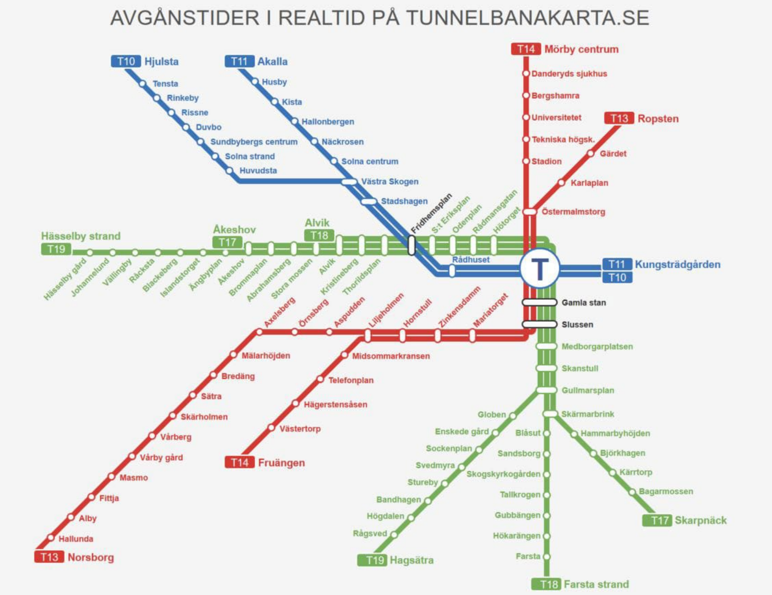




The "first problem" in graph theory:
Is there a walk through the city that would cross each of those bridges once and only once.

Original article: `https://scholarlycommons.pacific.edu/euler-works/`

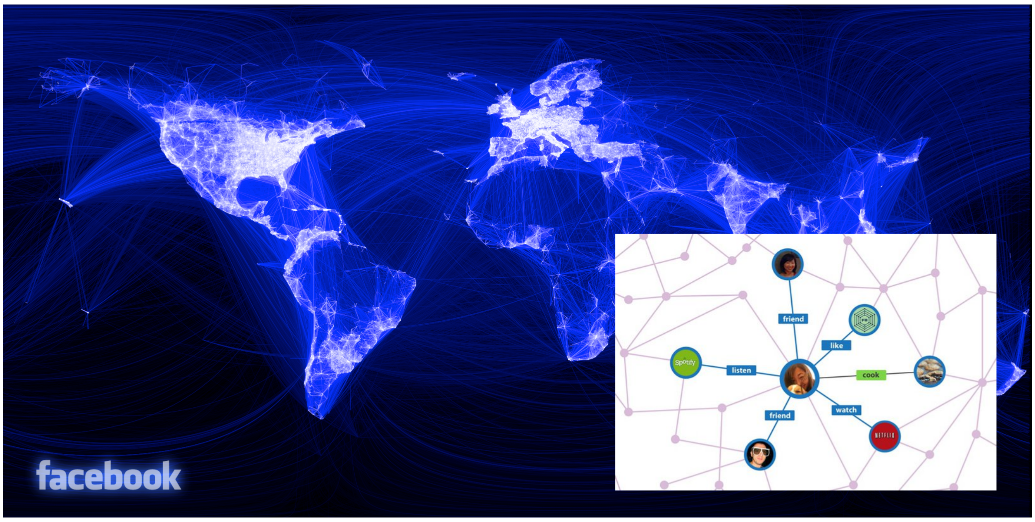
Metro Network



Vertices = metro stations
Edges = direct connections between stations

Source `https://tunnelbanakarta.se/`

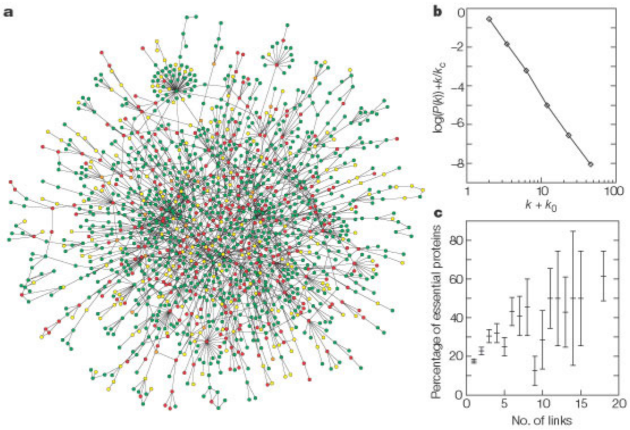
Social Networks



Vertices = user accounts
Edges = two accounts are "friends"

Source: `http://blog.revolutionanalytics.com`

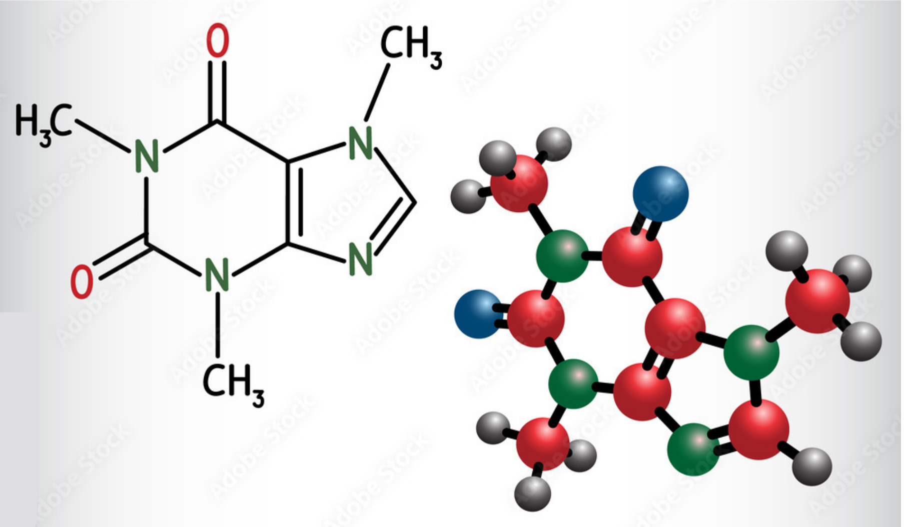
Graph of Protein-Interactions (yeast)



Vertices = proteins
Edges = two proteins interact

Source `https://www.nature.com/`

Chemistry and Molecules



$C_8H_{10}N_4O_2$ = caffeine

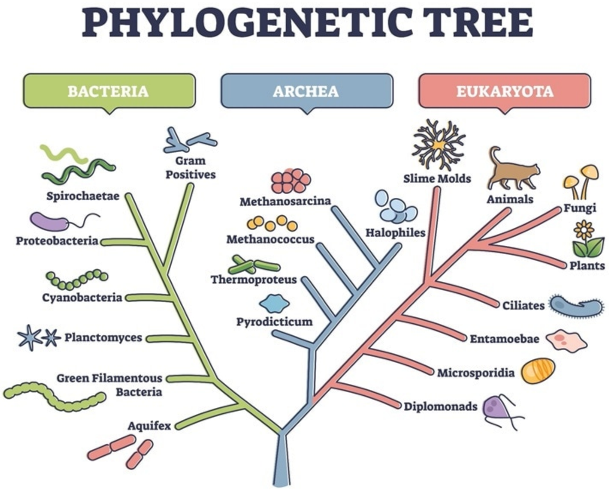
Vertices = atoms
Edges = chemical bonds

Source `https://stock.adobe.com/`

Any idea what this molecule is?

(Hint: makes you awake and can be transformed by mathematicians into theorems)

Phylogenetics Trees



Vertices = Taxa (e.g. genes or species)
Edges = ancestor relationship

Source: `https://www.azolifesciences.com/`

Graphs can be used to model various types of real-world scenarios.

Given such a graph, it is therefore of interest, to understand its structure (what does structure mean?).
$\Longrightarrow$ we need algorithms to analyze them.

Here, we focus on simple structural properties and basic algorithms to compute them.

To recall:

A tuple (V, E) is called an ***undirected graph*** if

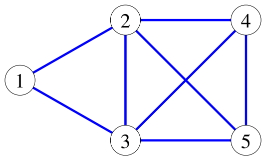
- V is a finite set, and
- E is a set of unordered pairs of elements in V.

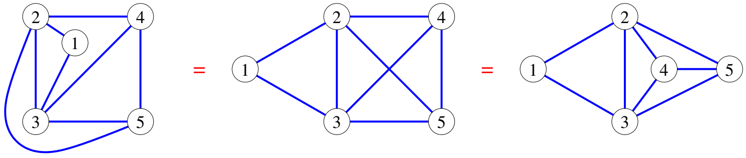
V is called the **vertex set**, and the elements of V are called **vertices** (often also **nodes**).
E is called the **edge set**, and the elements of E are called **edges**.

For now, we consider undirected graphs only and call them just ***graphs***

Example. $V = \{1, 2, 3, 4, 5\}$

$E = \{\{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{2,5\}, \{3,4\}, \{3,5\}, \{4,5\}\}$





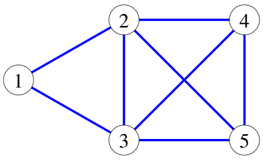
The neighborhood $N(v)$ of v in $G = (V, E)$ is the set $N(v) = \{w \in V \mid \{v, w\} \in E\}$ of all vertices w of G such that $\{v, w\}$ form an edge in G. If $u \in N(v)$ (and thus, $v \in N(u)$), then u and v are called neighbors.

Handshake-lemma. $\sum_{v\in V}|N(v)| = 2|E|$ *for every graph* $G=(V,E)$.

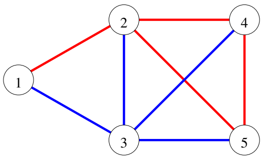
Proof: Each edge $\{u,w\}$ connects exactly the two vertices u and w and so it contributes 1 to $|N(u)|$ and 1 to $|N(w)|$. Thus, each edge contributes 2 to $\sum_{v\in V}|N(v)|$. Hence, $\sum_{v\in V}|N(v)|$ is equal to twice the number of edges. □

Let $G=(V,E)$ and $G'=(V',E')$ be graphs. Then,

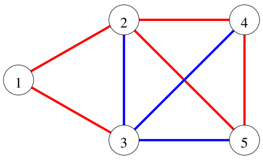
- G' is called a subgraph of G if $V'\subseteq V$ and $E'\subseteq E$.
- G' is called a spanning subgraph of G if G' is a subgraph of G with $V'=V$ (G' contains the same vertices as G).



$N(4)=\{2,3,5\}$ and $|N(4)|=3$



a subgraph



a spanning subgraph

A graph G is connected if for any two vertices $x,y\in V(G)$ there is an xy-path.
A connected and acyclic graph is a tree (see previous lectures for further defs).

Lemma. *If* $T=(V,E)$ *is a tree, then* $(V,E\setminus e)$ *is disconnected for all* $e\in E$.

Proof: Let $T=(V,E)$ be a tree. By definition, T is connected.

$\Rightarrow$ for all $u,v\in V$ there is a uv-path in T.

In particular, there cannot be two uv-paths in T since, otherwise, we can find simple cycles.

$\Rightarrow$ For all $u,v\in V$ there is a *unique* uv-path in T

$\Rightarrow$ For $e=\{u,v\}\in E$, the unique uv-path in T is precisely the edge $\{u,v\}$.

$\Rightarrow$ $(V,E\setminus e)$ does not contain a uv-path and is, therefore, not connected.

□

Tree-Theorem. *$T = (V, E)$ is a tree if and only if T is connected and $|E| = |V| - 1$.*

Proof: *only-if-direction:* Let T be a tree. By definition T is connected.

We show $|E| = |V| - 1$ by induction on $|V|$.

Base case: A tree with one vertex has no edges $\Rightarrow |V| - 1 = 0 = |E|$
A tree with two vertices contains exactly one edge edge $\Rightarrow |V| - 1 = 1 = |E|$

Ind.-Hyp.: Assume that statement is true for all trees with $< k$ vertices, $k \geq 2$

Let $T = (V, E)$ be a tree with $|V| = k$, $k \geq 2$. At least one edge $e \in E$ must exist, else T would be disconnected.

By the previous lemma, $T' = (V, E \setminus \{e\})$ is disconnected.

In particular, e "links" two subgraphs in T' and these two subgraphs must be connected, i.e., T' consists of exactly two connected subgraphs $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$.

Since we have not added edges, T_1 and T_2 are acyclic and, therefore, trees.

By Ind.-Hyp. $|E_1| = |V_1| - 1$ and $|E_2| = |V_2| - 1$.

Note that $|V| = |V_1| + |V_2|$ and $|E| = |E_1| + |E_2| + 1$.

Hence, $|E| = |E_1| + |E_2| + 1 = (|V_1| - 1) + (|V_2| - 1) + 1 = |V_1| + |V_2| - 1 = |V| - 1$.

if-direction: Let $T = (V, E)$ be connected and $|E| = |V| - 1$.

Assume, for contradiction, that T contains *simple* cycle $C = (v_0, v_1, \ldots, v_k, v_0)$, i.e.,
$P = (v_0, v_1, \ldots, v_k)$ is a simple path of length $k \geq 2$ and $e = \{v_k, v_0\} \in E$.

Removing e from T results in $T_1 = (V, E \setminus \{e\})$.

Note T_1 remains connected but it does not contain the cycle C anymore. If T_1 is a tree, we stop.

If T_1 is not a tree,, we continue with the latter process by taking the next simple cycle C' in T_1.
remove an edge from C' to get a connected graph T_2 that does neither contain C nor C'.

After k steps this process must terminate, i.e., we obtain a tree $T_k = (V, \tilde{E})$.

We already proved: $|\tilde{E}| = |V| - 1$ (*only-if-direction*).

By assumption, $|E| = |V| - 1$ and thus, $|\tilde{E}| = |E|$ must hold.

$\Rightarrow E = \tilde{E}$, i.e., there cannot be simple cycles in T and, since T is connected, it is a tree. □

Graphs can be used to model various types of real-world scenarios.

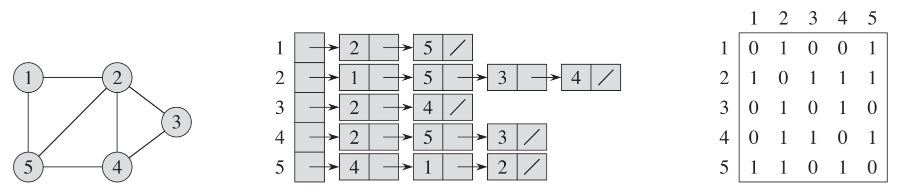
Given such a graph, it is therefore of interest, to understand its structure (what does structure mean?).
$\Longrightarrow$ we need algorithms to analyze them.

Simple structural properties:

- Is G connected?
- Does G contain simple cycles?
- Is G a tree?
- What is distance between two vertices u, v, i.e., length of shortest (simple) uv-path?

How to store graphs

Before we delve into the algorithms, let's take a closer look at how to store graphs.
Here, we assume that the vertics in $G = (V, E)$ are integers $1, \dots, |V|$.



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

G adjacency-list adjacency matrix

There are two common standart ways (among others):

- The adjacency-list of a graph $G = (V, E)$ ..

  .. consists of an array A of $|V|$ linked lists, one for each vertex j in V and each list $A[j]$ contains all vertices $i \in N(j)$, i.e., those i for which $\{i, j\} \in E$.

  Usually used when graph is sparse, i.e., $|E|$ is much less than $|V^2|$.

- The adjacency matrix of a graph $G = (V, E)$ is a Boolean $|V| \times |V|$-matrix $A = (a_{ij})$ with

$$a_{ij} = \begin{cases} 1 & \text{falls } \{i, j\} \in E \\ 0 & \text{sonst} \end{cases}$$

  Usually used when graph is dense, i.e., $|E| \simeq |V^2|$ or if we want to remove/insert edges in $O(1)$ time.

For the simple graph properties we want to test as listed above we can use graph traversal, that is:

- visit the vertices in a graph beginning with a start vertex s such that
  - Each vertex reachable from s is visited exactly once.
  - The next visited vertex always has at least one neighbor in the set of previously visited nodes.

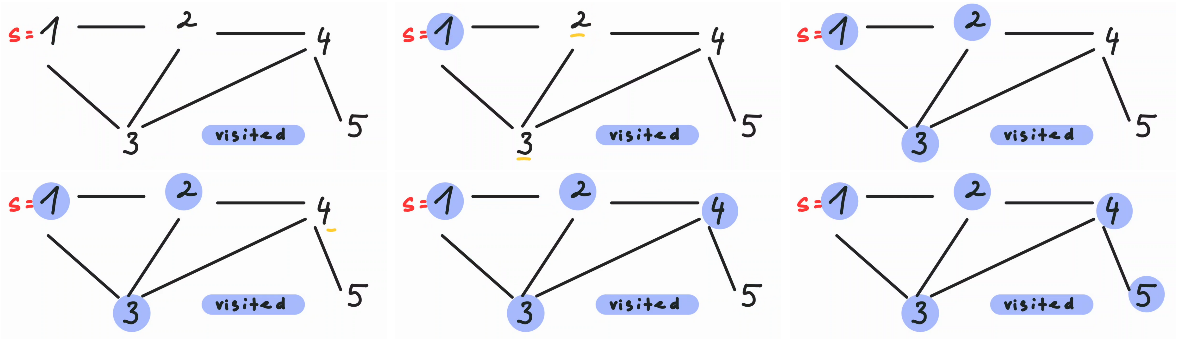
We consider here two classical graph traversal algorithms:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)

5.2 Breadth-First Search (BFS)

```
BFS($G = (V, E), s$)
visited($v$):=false for all $v \in V$ //to keep track of visited nodes
init empty queue $Q$ //FIFO
visited($s$):=true
$Q$.enqueue($s$)
WHILE ($Q \neq \emptyset$) DO
  $v := Q$.front() and $Q$.dequeue()
  FOR (all neighbors $w \in N(v)$ of $v$ for which visited($w$)=false) DO
    $Q$.enqueue($w$)
    visited($w$):=true
```

Example.



after L1-4: $Q = (1)$ visited(1):=true

L6 **(BFS-order)**	L8	L9
$v = 1$ and $Q = ()$	$Q = (2, 3)$	visited(2) = visited(3) =true
$v = 2$ and $Q = (3)$	$Q = (3, 4)$	visited(4) =true
$v = 3$ and $Q = (4)$	–	–
$v = 4$ and $Q = ()$	$Q = (5)$	visited(5) =true
$v = 5$ and $Q = ()$	–	–

Lemma. *BFS$(G = (V, E), s)$ can be implemented to run in $O(|V| + |E|)$ time for every graph $G = (V, E)$.*

Proof: Assume that G is stored as adjacency-list (for all v the set of neighbors $N(v)$ is available)
L1: $O(|V|)$ time // L2-4: $O(|1|)$ time
while-loop

- Only non-visited are added to queue and then marked as visited
  $\Rightarrow$ each $v \in V$ is enqueued at most once, and hence dequeued at most once.
- The operations of enqueuing and dequeuing take $O(1)$ time, and so the total time devoted to queue operations is $O(|V|)$.
- The neighbors in $N(v)$ of each vertex v are scanned only when the vertex is dequeued
  $\Rightarrow$ the neighbors in each $N(v)$ are scanned at most once
- The handshake-lemma implies $\sum_{v \in V} |N(v)| = 2|E| \in O(|E|)$.
- $\implies$ while-loop runs in $O(|V| + |E|)$ time.

BFS runtime: $O(|V| + |E|)$. □

5.2.1 Is $G = (V, E)$ connected?

Now, lets squeeze out all structural properties of a graph G we may get using BFS.

In what follows, we say that v is marked as visited, precisely if visited(v) =true

Structural Property: Is $G = (V, E)$ connected?

Lemma. All vertices in V are marked as visited after run of BFS(G, s) $\iff$ G is connected

Proof: "$\Leftarrow$" Suppose that G is connected. Thus, there is an sx-path P in G for all $x \in V$.

Assume, for contradiction, that $x \in V$ is *not* marked as visited after run of BFS(G, s).

Since s is marked as `visited`, but x is not, there are consecutive vertices v, w in $P = (s, \dots, v, w, \dots x)$ such that v is marked as `visited`, but w is not.

In particular, $\{v, w\} \in E$ and, therefore $w \in N(v)$.

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v was enqueued to Q (directly before/after v was marked as `visited`).

$\Rightarrow$ At some step of `BFS`(G, s), the vertex v is dequeued from Q and the for-loop is entered.

As w is a non-`visited` neighbor of v, it will be considered during the run of the for-loop for v and is thus, marked as `visited`; a contradiction ↯.

Since the latter holds for all $x \in V$, all $x \in V$ are marked as `visited`.

"$\Rightarrow$" Assume that all vertices in V are marked as `visited` after run of `BFS`(G, s).

If $V = \{s\}$, then G is connected. Assume that $|V| > 1$ and let $w \in V \setminus \{s\}$.

Since w is marked as `visited`, we have $w \in N(v_1)$ whereby $v_1 \in Q$ prior to point where w is considered in for-loop

Since v_1 was added to Q, it holds that either

$v_1 = s$ (in which case we found an sw-path $P = (s, w)$) or

$v_1 \in N(v_2)$ whereby $v_2 \in Q$ prior to point where v_1 is considered in for-loop

Repeating the latter, ends in a sequence of vertices $v_k, \dots, v_1, v_0$ with $s = v_k$ and $w = v_0$ and such that $v_i \in N(v_{i+1})$, $0 \leq i \leq k-1$ and thus, we obtain an sw-path $P = (v_k, \dots, v_0)$.

$\Rightarrow$ for all $w \in V$ there is an sw-path

$\Rightarrow$ For all $x, y \in V$ there is an sx-path and sy-path and combining these two paths yields an xy-path

$\Rightarrow$ G is connected. □

Thus, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is connected can be done $O(|V| + |E|)$ time.

5.2.2 Is $G = (V, E)$ a tree?

Structural Property: Is $G = (V, E)$ a tree?

Since we can test connectedness of $G = (V, E)$ in $O(|V| + |E|)$ time, we obtain

Lemma. Testing whether a graph $G = (V, E)$ is a tree can be done $O(|V| + |E|)$ time.

Proof. First test if G is connected in $O(|V| + |E|)$ time. Then, check if $|E| = |V| - 1$. In the affirmative case, the Tree-Theorem implies that G is a tree. □

5.2.3 Finding spanning trees

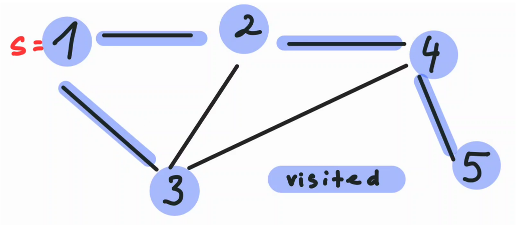
Find spanning tree of G (if there is one).

In many applications we are also interested in subgraphs of G that are trees, in particular, spanning trees.

To obtain spanning trees, let us slightly modify BFS by marking also edges as visited.

```
modi_BFS($G = (V, E), s$)
visited($v$):=false for all $v \in V$ and visited($e$):=false for all $e \in E$
init empty queue $Q$ //FIFO
visited($s$):=true
$Q$.enqueue($s$)
WHILE ($Q \neq \emptyset$) DO
    $v := Q$.front() and $Q$.dequeue()
    FOR (all neighbors $w \in N(v)$ of $v$ for which visited($w$)=false) DO
        $Q$.enqueue($w$)
        visited($w$):=true
        visited($\{v, w\}$):=true
```

Example.



Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$

visited edges: $\{1, 2\}$ $\{1, 3\}$,$\{2, 4\}$, $\{4, 5\}$.

In this example, we obtain a spanning tree (called BFS-tree).

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all visited edges after run of modi_BFS.

Lemma. *If $G = (V, E)$ is connected, then $T_{BFS} = (V, F)$ is a spanning tree of G*

Proof: We show that T_{BFS} is connected and has $|V| - 1$ edges.

Since G is connected, we can apply the same arguments as in the proof of the "⇒"-direction of the lemma "All vertices marked as visited iff G connected", to conclude that, for all $w \in V$, there is sw-path along visited vertices that consists of visited edges only. Hence, T_{BFS} is connected.

Since G is connected, the latter lemma also implies that all vertices are marked as visited. In particular, all $|V| - 1$ vertices distinct from s must have been marked during the execution of the for-loop and each vertex is marked visited precisely once. Hence, F contains precisely $|V| - 1$ edges.

Thus, T_{BFS} is connected and has $|V| - 1$ edges and the Tree-Theorem implies that T_{BFS} is a tree.

Since the vertex set of T_{BFS} and G is V, T_{BFS} is a spanning tree of G □

Corollary. *$G = (V, E)$ is a tree if and only if G is connected and $|F| = |E|$*

Proof: If $G = (V, E)$ is a tree, then G is connected and there is only one spanning tree and thus, $G = T_{BFS}$ (hence, $|E| = |F|$)

Suppose that G is connected and $|F| = |E|$. Since F is edge set of the spanning tree T_{BFS} of G it holds that $|F| = |V| - 1$ and thus, $|E| = |V| - 1$. By the Tree-Threorem, G is a tree. □

Let $T_{BFS} = (V, F)$ be the graph with vertex set V and F being the set of all `visited` edges after run of `modi_BFS`.

Summary so-far: For a given graph $G = (V, E)$, `BFS` and its modification allows us to determine in $O(|V| + |E|)$ time . . .

- . . . if G is connected (all vertices in V are marked as `visited`)
- . . . if G is a tree. (Check first connectedness and then count number f of `visited` edges and compare f and $|E|$)
- . . . a spanning tree $T_{BFS} = (V, F)$ of G in case G is connected.

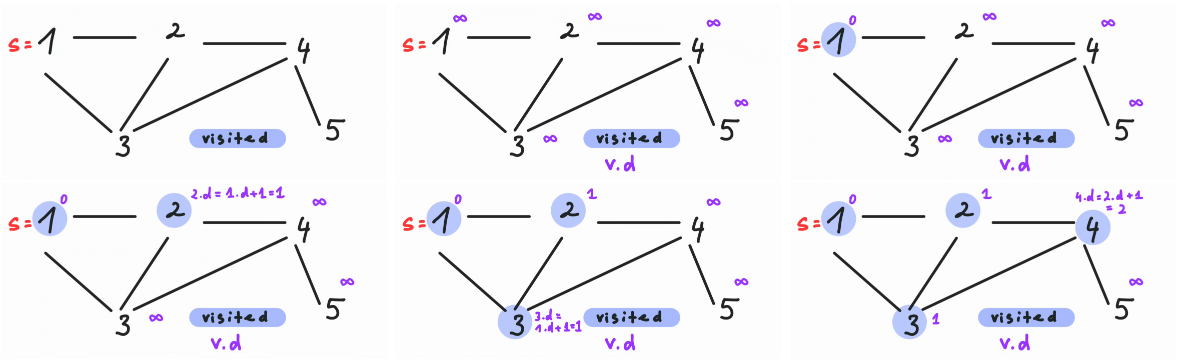
The latter results also imply

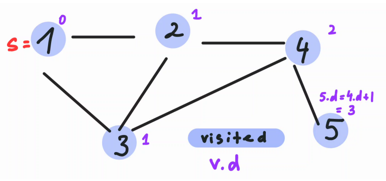
Theorem. *For each connected graph $G = (V, E)$ we have $|E| \geq |V| - 1$ and G has a spanning tree.*

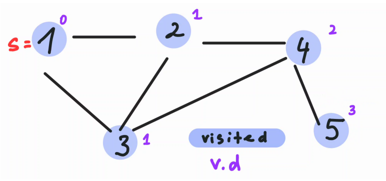
5.2.4 Determine distances in G

Structural Property: Distances in $G = (V, E)$?

```
dist_BFS($G = (V, E), s$)
visited($v$):=false and $v.d := \infty$ for all $v \in V$
init empty queue $Q$ //FIFO
visited($s$):=true and $s.d := 0$
$Q$.enqueue($s$)
WHILE ($Q \neq \emptyset$) DO
    $v :=$ $Q$.front() and $Q$.dequeue()
    FOR (all neighbors $w \in N(v)$ of $v$ for which visited($w$)=false) DO
        $w.d := v.d + 1$
        $Q$.enqueue($w$)
        visited($w$):=true
```







Example. Given the same order in which vertices are added to Q as in previous slides: $1, 2, 3, 4, 5$ What have we computed? Answer: $d(s, v)$ for all $v \in V$. Let's prove correctness!

In many application it is useful to know the distances $d(x, y)$ between vertices x and y in G, where

$$d(x,y) = \begin{cases} \min_{P \in \mathcal{P}} |P| & \text{if set } \mathcal{P} \text{ of all } xy\text{-paths is not empty} \\ \infty & \text{else} \end{cases}$$

Here $|P| \coloneqq k$ denotes the length of a path $P = (v_0, v_1, \ldots, v_k)$.

In other words $d(x, y)$ denotes the length of a shortest xy-path in G. If no such path exists, then $d(x, y) = \infty$.

To this end, consider the modification `dist_BFS`.

Theorem. *`dist_BFS`(G,s) correctly computes the distances $d(s, v)$ between s and all $v \in V$ in $O(|V| + |E|)$ time.*

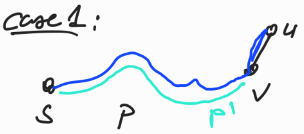
proof board

dist_BFS

L.1. let $G=(V,E)$ be a graph & $s\in V$.
Then $d(s,v) \leq d(s,u)+1 \quad \forall \{u,v\}\in E$.

proof: if v no sv-path $\Rightarrow$ no vu path $\Rightarrow$ $d(sv)=d(su) = \infty$.

else **case 1:**



shortest su path contains v $\Rightarrow$ $d(su) = d(sv)+1$

as p' must be shortest sv path

Case 2 shortest su path does not contain v

$\Rightarrow d(sv) \leq d(su)+1$ ✓□

L.2: Let $G=(V,E)$ be a graph.
Then, for all $u \in V$ the value $u.d \geq d(s,u)$
at all steps of dist-BFS(G,s)
[incl. termination].

proof:

proof by induction on # enqueue-operations.

Base case: after $\underline{\underline{1}}$ enque-op (L.4): $s.d = 0 = d(s,s)$
& $u.d = \infty \geq d(s,u)$ ✓

Assume statement true after k enqueue-operations.
Now, the $k+1$ enqueue-op takes place
in FOR-loop for some $w \in N(v)$: Q.enqueue(w)

Before Q.enqueue(w): $u.d \geq d(s,u) \forall u$ by ind. hyp.
Then in L8: $w.d = v.d + 1 \underset{\text{by ind. hyp.}}{\geq} d(s,v) + 1 \underset{\text{since } \{v,w\} \text{ edge \& by L.1}}{\geq} d(s,w)$

Since all other $u.d$ remain unaffected & $w.d \geq d(s,w)$
$\Rightarrow$ AFTER Q.enque(w) (= $k+1$ enqueue op)
$\forall u \in V$: $u.d \geq d(s,u)$

Since each u is when enqued once
never enqued again (since marked as visited)
$\Rightarrow$ value $u.d$ remains unchaged after
u enqued once $\Rightarrow$ □

L.3 Suppose during run of dist-BFS on $G=(V,E)$, the queue $Q = (\underset{\text{first}}{v_1}, \dots, \underset{\text{Last}}{v_r})$

Then, $\overset{i)}{v_r.d \le v_1.d+1}$ & $\overset{ii)}{v_i.d \le v_{i+1}.d}$, $1 \le i < r$

[intuition: $\overbrace{\underbrace{v_1 d \le v_2.d \le \dots \le v_r.d}_{}}^{} \le v_1 d + 1$ (ii) (i)

$$\overbrace{v_1 d \le v_2.d \le \dots \le v_r.d}^{(ii)} \le v_1 d+1$$

dist-values of all vertices are either all equal or form sequence $(k, .. k, k+1 .. k+1)$ for some integer $k \ge 0$

proof: Now, induction over # operation on Q. (enqueue + dequeue)

Base case: # op on Q = 1 => Q = (s) in L4 statement trivially true as Q contains only 1 element.

Assumption: statement true after # op on Q = k.

k+1 op. can be dequeue or enqueue.

Suppose first k+1 op on Q = $(v_1 \dots v_r)$ is dequeue

Ind hyp: (i) $v_r.d \le v_1.d+1$ & *
(ii) $v_1.d \le v_2.d$ **

-> after dequeue: $Q = (v_2 \dots v_r)$ or $\underbrace{Q = \emptyset \text{ if } r=1}_{\text{i, ii trivially true}}$

[must show i) ii) holds for this Q])

by * $v_r \le v_1.d+1 \overset{**}{\le} v_2 d+1$ => (i) $v_r \le v_2.d+1$

& (ii) holds for all $2 \le i \le r$

=> i) ii) satisfied for Q.

Now assume k+1 op. on $Q = (v_1 \cdots v_r)$ is enqueue (L. 9)

k+1 op: $Q.\text{enqueue}(w) \Rightarrow \tilde{Q} = (v_1 \dots v_r\ w)$

[must show i) ii) holds for this $\tilde{Q}$]

if $Q = (\ )$ before enqueue w, i & ii) trivially hold. in $\tilde{Q} = (w)$

Ann $r \geq 1$.

Before FOR-loop where w is is enqueue vertex v was dequed $[v \in N(w)]$ in L.6

In particular, since v was already in the queue & in L6 dequeued directly before FOR-loop where $w \in N(v)$ is taken.

Before v dequeued

$$\tilde{Q} = (v, x_1 \dots x_{r'})$$

after v dequeued:

$\hookrightarrow$ Ind. Hyp: $\overset{*}{\underbrace{v.d \leq x_1.d}_{(ii)}} \leq \dots \leq \underbrace{x_{r'}.d \leq v.d + 1}_{(i)}$ ⊛

$$\tilde{\tilde{Q}} = (x_1 \dots x_{r'})$$

after W endqueued.

$$Q = (\overset{v_1}{x_1} \dots \overset{v_{r'}}{x_{r'}}, \overset{v_{r'+1} \cdots v_{r-1}}{y \dots} \overset{v_r}{w})$$

by construction (L8)

$w.d = v.d + 1$

$\overset{⊛}{\leq} x_1.d + 1$ $\Rightarrow$ i) holds for Q.

For ii) observe:

$$Q = (\overset{v_1}{x_1} \dots \overset{v_{r'}}{x_{r'}} , \overbrace{\overset{v_{r'+1}}{y} \overset{\dots v_{r-1}}{\dots} \overset{v_r}{w}}^{\text{all those must be in } N(v)\,!}) \quad \overset{\text{Line 8}}{\Rightarrow} v_{r'+1}.d = \dots = v_{r-1}.d = \overset{w.d}{\overset{\shortparallel}{v_r}}.d = v.d+1$$

$\circledast$

$$v.d \le \underset{v_1 d}{x_1.d} \le \dots \le \underset{v_{r'}.d}{x_{r'}.d} \le v.d+1 = v_{r'+1}.d = \dots = v_{r-1}.d = \underset{\substack{\shortparallel \\ w.d}}{v_r.d}$$

$\Rightarrow$ ii) holds for Q.

$\square$

COR 1: if v is enqued before v' is enqued in distBFS. THEN $v.d \leq v'.d$ (direct consequence of L3.)

Thm: dist_BFS(G,s) computes all dist $d(s,v)$ correctly for all $v \in V(G)$: $v.d = d(s,v)$ after termination.

proof:

By contradiction, assume $v.d \neq d(s,v)$.

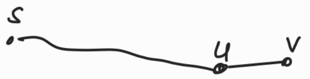
Choose v among all vertices v' with $v'.d \neq d(s v')$ as the one for which $d(sv)$ is minimum. $\Big\}$ ⊛

Lemma 2 + $v.d \neq d(sv)$ implies: $v.d > d(s,v)$

Note $v \neq s$, since $s.d = d(s,s) = 0$ correct (L3)

Moreover there must be an sv-path in G, otherwise $\infty = d(s.v) \geq v.d$ (contradicting $v.d > d(s,v)$)

Let P be some shortest sv-path $(s \ldots \underbrace{uv}_{\text{edge}})$ (since $s \neq v$, u exists where $u = s$ possible)



$\Rightarrow d(sv) = d(su) + 1$

By choice of v (min dist ⊛) $\Rightarrow u.d = \underset{<\infty}{d(s,u)}$

$\rightarrow v.d > d(sv) = d(su) + 1 = u.d + 1$ ⊛

Now consider time when distBFS choses

"$u := Q.\text{dequeue}$"

which must happen as $u.d < \infty$

v is either visited or not at this point.

IF v is non visited $\Rightarrow$ $v \in N(u)$ & FORLOOP (L7)

implies v will be considered

& $v.d = u.d + 1$ ↯

contradiction to $\circledast$

IF v visited $\Rightarrow$ then it was enqued

to Q at some point (L9+10)

▶ IF v was enqued before u was enqued

$\Rightarrow$ COR 1: $v.d \leq u.d$ ↯ contradiction to $\circledast$

▶ IF v was enqued after u was enqued

$\Rightarrow$ $v \in N(u')$, $u' \neq u$, u' was enqued before u.

& $v.d = u'.d + 1$

$\Rightarrow$ COR 1: $u'.d \leq u.d$ $\Rightarrow$ $v.d \leq u.d + 1$ ↯

contr. $\circledast$

$\Rightarrow \quad v.d = d(s,v) \quad \forall v \in V$

[this also implies additionally that all vertices reachable from s are visited otherwise $v.d = \infty > d(sv)$ would hold]

$\square$

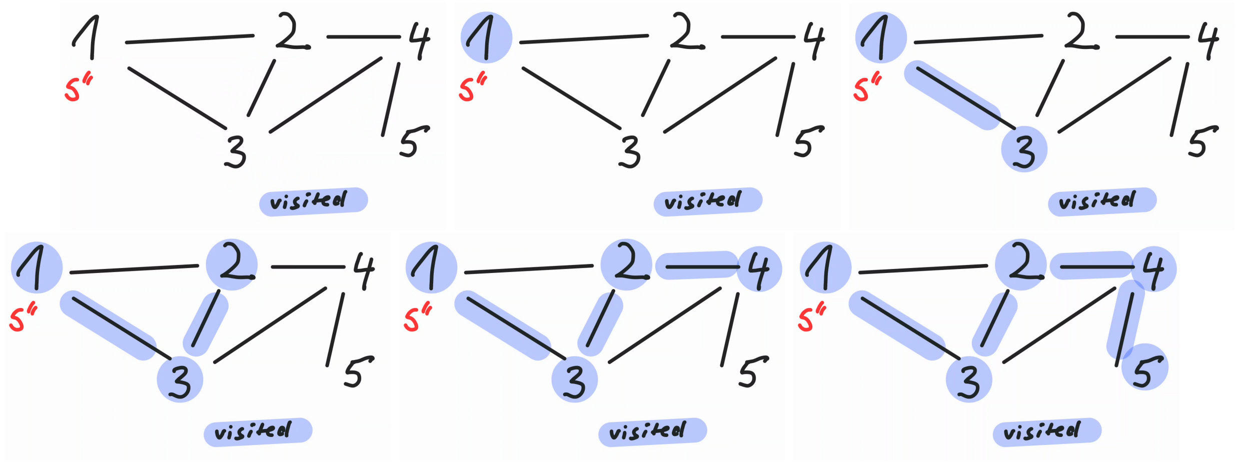
We could run `dist_BFS`(G,w) for each $w \in V$ as start vertex and thus, obtain the distances $d(w, v)$ for all $w, v \in V$, which implies

Lemma. *The distances between all pairs of vertices in $G = (V, E)$ can be computed $O(|V|(|V| + |E|)) = O(|V|^2 + |V||E|)$ time.*

There are many other algorithms that can be used to determine distances that even work in the case that we may have edge weights (here `dist_BFS` would fail in general).

5.3 Depth-First Search (DFS)

```
DFS($G = (V, E), s$)
1 visited($v$):=false for all $v \in V$
2 init empty stack $S$ //LIFO
3 $S$.push($s$)
4 WHILE ($S \neq \emptyset$) DO
5   $v := S$.top() and $S$.pop()
6   IF (visited($v$)$\neq$true) THEN
7     visited($v$):=true
8     FOR (all neighbors $w \in N(v)$ of $v$ for which visited($w$)=false) DO
9       $S$.push($w$)
```



after L1-4: $S = (1)$

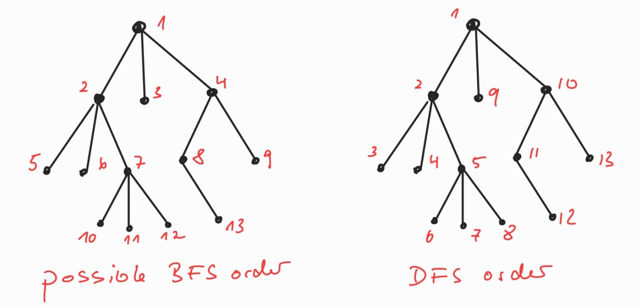
L5	L7 **(DFS-order)**	L9
$v = 1$ and $S = ()$	visited(1) =true	$S = (2, 3)$
$v = 3$ and $S = (2)$	visited(3) =true	$S = (2, 4, 2)$
$v = 2$ and $S = (2, 4)$	visited(2) =true	$S = (2, 4, 4)$
$v = 4$ and $S = (2, 4)$	visited(4) =true	$S = (2, 4, 5)$
$v = 5$ and $S = (2, 4)$	visited(5) =true	-
$v = 4$ and $S = (2)$	-	-
$v = 2$ and $S = ()$	-	-

DFS plays a particular role when considering directed graphs.

Both BFS and DFS graph traversal algorithms, that is:

- they visit the vertices in a graph beginning with a start vertex s such that
  - Each vertex reachable from s is visited exactly once.
  - The next visited vertex always has at least one neighbor in the set of previously visited nodes.

BFS goes first into "breadth" while DFS goes first in "depth".



Neither the BFS- nor DFS-order is unique (e.g., we could have visited first 4 and then 3 in both trees)

5.4 Kruskal Algorithm and Minimum Spanning Trees

In many cases, we are interested in weighted graphs and minimum spanning trees.

Let (V, E) be a graph and $w : E \to \mathbb{R}$.

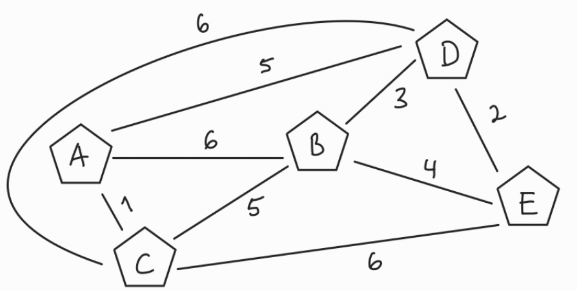
- The triple $G = (V, E, w)$ is called a weighted (or edge-weighted) graph.
- $w(e)$ is called the weight (or length) of edge $e \in E$.

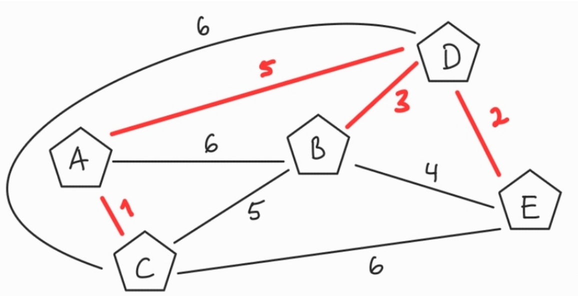
Given: Weighted connected graph $G = (V, E, w)$.
Find: A minimum spanning tree (MST), i.e., a spanning tree $T = (V, F)$ with minimum total edge weight. That is, choose the set of edges $F \subseteq E$ such that

$$\textstyle\sum_{e\in F} w(e)$$

is minimized.

Example
Consider a scenario where a company needs to connect several buildings on its campus using cables or fiber optics. Each building is represented as a vertex, and the distances between the buildings are represented as edges with weights (the cost of laying cables/fiber).





In this scenario, we want to connect all the buildings with the minimum cost possible. This is where an MST becomes useful which will give us the subset of edges that form a tree connecting all the nodes with the minimum total edge weight.

Question: Does BFS still work?
Answer: No, take graph on 3 vertices s, u, v and weights $w(\{s, v\}) = w(\{s, u\}) = 2$ and $w(\{u, v\}) = 1$.

$\Longrightarrow$ we need a different approach.

Many optimization problems are rather difficult to solve (keyword: NP-hard).

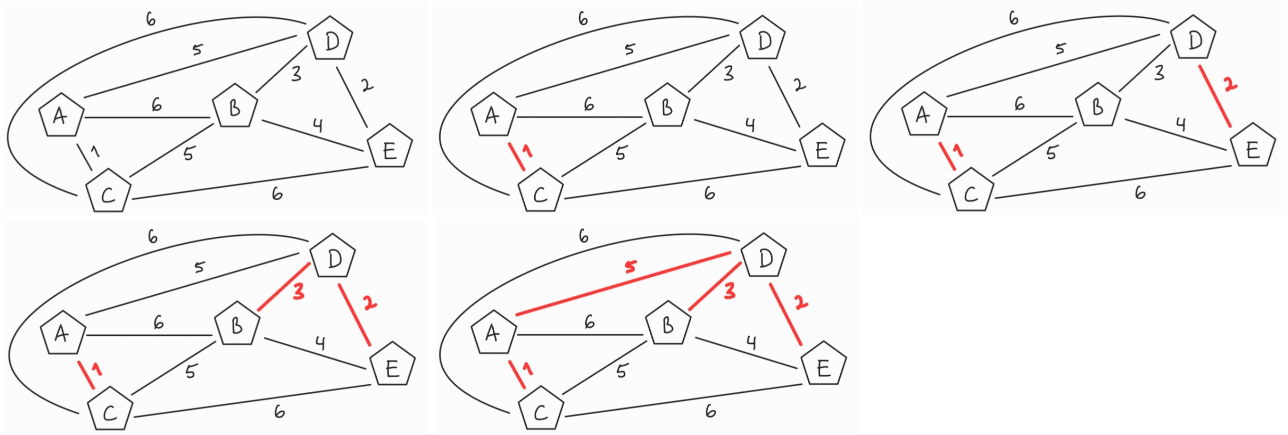
However, somewhat surprisingly the MST problem can be solved in polynomial time with a simple greedy algorithm.

Question: What is a greedy algorithm?
Answer: A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

How does a greedy algorithm for finding an MST look like?

```
Kruskal($G = (V, E, w), w\colon E \to \mathbb{R}$) //$m = |E|$
Sort edges such that $w(e_1) \leq w(w_2) \cdots \leq w(e_m)$
$F := \emptyset$, $T := (V, F)$
FOR $i = 1, \ldots, m$ DO
    IF $(V, F \cup \{e_i\})$ is acyclic
        $F := F \cup \{e_i\}$
return $T$
```

Kruskal's algorithm finds the minimum spanning tree as follows: It starts by sorting edges by weight, then adds them one by one from lightest to heaviest while ensuring that the "intermediate" graph T remains a forest, until all vertices have been checked and when possible added to T



$$F = \{\{A, C\}, \{D, E\}, \{B, D\}, \{A, D\}\}$$

Theorem. *Kruskal correctly computes an MST for a given undirected, connected graph $G = (V, E)$ in $O(|E||V|)$ time.*

Proof *Board.*

KRUSKAL

Theorem: Kruskal's algorithm correctly computes a MST for given undirected graph G. in $O(|V||E|)$ time

- RUNTIME: [sketch]

Sort edges : $O(|E| \log(|E|)) = O(|E| \log|V|)$ since $|E| \leq |V|^2$ $\Rightarrow \log(|E|) \leq \log(|V|^2) = 2\log(|V|)$

$F = \emptyset$: $O(1)$

$T = (V, F)$: $O(|V|)$

FOR loop: in the i-th step, when you check if $(V, F \cup \{uv\}) = T'$ is acyclic

Start $BFS(T', v)$ $\Rightarrow$ this goes along all vertices that are reachable from v

$\Rightarrow$ gives you incl.-max subgraph H of G that contains v & any new cycle is in A (if there is any)

H as at most $2i$ vertices & i edges.

$\Rightarrow BFS(T', v)$ in i-th step runs in $O(i + 2i) = O(i) \leq O(|V|)$ time

Total: $O(|E||V|)$

with efficient datastructure, this can be improved to $O(|E| \log(|V|))$

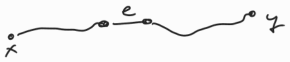
- Correctness?

proof: 1) Alg. terminates ✓

2) resulting graph T contains is acyclic & $V(T) = V(G)$ } T is spanning forest of G.

3) T is connected: IF not $\Rightarrow \exists\, x,y \in V(T)$ st no x-y-path exists in T

G is connected $\Rightarrow \exists$ x-y-path in G.

$\Rightarrow$ simple x-y-path & edge on e st $e \notin F$:  in G

at some i-th step of FOR-loop edge e is considered.

But $T' = (V, F \cup \{e\})$ (at step i) remains acyclic (otherwise there would have been a x-y-path already in T' & thus in T)

$\Rightarrow$ e is added to T at step i $\lightning$ $e \notin F$

$\Rightarrow$ T is spanning tree of G.

remains to show that T is <u>minimum</u> spanning tree of G.

Let F_i be the set of edges formed up to step i

<u>Claim:</u> $\exists$ MST T^* of G st $F_k \subseteq E(T^*)$. $\forall$ steps $k = 1..m$

$i=0$, $T = (V, \emptyset)$ ✓

<u>Ind hyp:</u> Assume claim is true for all steps $i \le k$

Let T^* be MST of G st $F_k \subseteq E(T^*)$

now: $i = k+1$. and let $e = (uv)$ be the current edge considered in step $i = k+1$, i.e.,

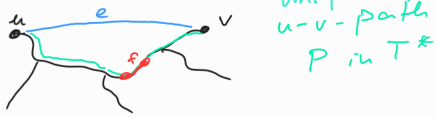
- IF e not added to F_k
  THEN $F_{k+1} = F_k \subseteq E(T^*)$ ✓
- IF e added to F_k
  THEN $F_{k+1} = F_k \cup \{e\}$

$\hookrightarrow$ IF $e \in E(T^*) \Rightarrow F_{k+1} \subseteq E(T^*)$ ✓

ELSE $e \notin E(T^*)$ & therefore, $(V, E^*(T) \cup \{e\})$ must contain a simple cycle

since $\forall\, uv\, \exists!$ simple uv-path in T^*, since $e = \{u,v\}$ not in $T^* \Rightarrow$ + e forms cycle

Situation: T^*



unique simple u-v-path P in T^*

$e=(uv)$

Since e is contained in T, at least one edge f of P cannot be contained in T

otherwise T contains cycle.

$\Rightarrow \exists f \notin F_k \subseteq E(T)$

Now, $\tilde{T}$ = "$T^* - f + e$" is again a tree. (exercise)

Moreover, $w(f) \geq w(e)$ [otherwise if $w(f) < w(e)$, then the greedy-choice would be f & not e in Step $i=k+1$]

$\Rightarrow w(\tilde{T}) \leq w(T^*)$

& since T^* is MST $\Rightarrow w(\tilde{T}) \geq w(T^*) \Rightarrow w(\tilde{T}) = w(T^*)$

$\Rightarrow \tilde{T}$ is MST that contains the edges in F_{k+1}.

Now repeat the latter arguments until all edges of T are contained in MST $\Rightarrow T$ is MST

◻

Chapter 6

Questions and Answers

6.1 Questions from students

What do I need to know and learn to pass the course? See slides 0-Orgastuff.pdf for the hard-coded details. Moreover, if you can follow the content of the course (i.e., you understand the slides as well as all the proofs) and if you are able to solve the exercises below then you are possibly well-prepared for the exam. Exam questions will be "similar in flavor" to those exercises [not identical of course!]. Moreover, for the exam I give from time to time some hints within in the lecture, to give you an idea of what is expected.

What means "understanding something"? Understanding something typically refers to the ability to comprehend, grasp, or make sense of a particular concept and idea,. It involves more than just knowing facts! However, it is hard to define. Take the term "understand" as something like "I am able to explain it to some other students". By way of example: Are you able to explain when an algorithm is said to solve a problem? Can you prove that $n^2 \in O(n^2.5)$ and explain it to some other student? Do you know what it means to change the base of a logarithm? Can you prove the correctness of insertion-sort and explain this proof to some other student? A good practice here is to explain the content and to discuss it with other students. Judging if you understood the content is completely up to you: The teacher can explain it to you but not understand it for you!

What if I dont understand certain parts of the content? If you cannot follow the content of the course, then be self-organized based on my hints on the course page. Hints which chapters you can read are given at the kurser homepage: explicit for each lecture. Discuss the content with other students. Go to the tutorials and ask the TAs to recap some parts of the lectures.

How can I deepen my knowledge? If you want to deepen your knowledge, be self-organized based on my hints on the course page. Hints which chapters you can read are given at the kurser homepage. Moreover, feel free to attack additional problems stated in the course book w.r.t. each of the sessions.

What is the difference between the courses DA4005 and DA4006? DA4006 is a foundational course concerned with fundamental content aimed at understanding how algorithms work, their applications, methods to prove their correctness, and analyzing their time and space complexity. While we primarily cover problems that can be solved by algorithms in polynomial time in DA4006, the course DA4005 also delves into the complexity of algorithms for problems that may not necessarily be solved in polynomial time. This includes an exploration of NP-completeness, as well as heuristics to solve such problems.

How are the "laboration"-sessions structured? The "laboration"-sessions are structured based on the current topics being discussed and the individual tutor's approach. They typically involve a combination of individual work by students during the sessions, supplemented with additional examples, explanations, and theory provided by the tutor. Practical aspects, like implementing algorithms and data structures in different programming languages, may also be covered. Furthermore, these sessions offer opportunities for students to ask questions and receive assistance when facing challenges with exercises.

What are "bonus exercises" in the Exercises (Home Assignments "LABO") Bonus exercises can be used to earn extra points. While all "non-bonus" exercises are mandatory, the "bonus" exercises are voluntary. The total points you can earn by completing the exercises are based on the non-bonus exercises. Therefore, completing the additional bonus exercises gives you the opportunity to receive more than 100% of the points. Note that there are no second submissions allowed, so you may use bonus exercises to also achieve the minimum passing grade of 50% for the LABO part.

6.2 The exam structure and exam problems

The following problems serve just as an idea for potential exam questions.

Examples are in many case in extrem simplified form to give you an idea and he exam problems can be expected to be more difficult

Additional exam question not provided here might be asked.

Problem (*Basic Questions*)

▶ A couple of basic questions will be asked.

Example (Examples might be provided in a very simplified form to give you an idea)

- What is an instance of a problem?
- What is a stack?
- What is a spanning tree of a graph?

Problem (*Algorithm Design*)

▶ Here pseudocode for a simple task should be provided and examined on a give example.

Example: (Examples might be provided in a very simplified form to give you an idea)

- Provide pseudocode for an recursive algorithm that takes as input an integer n and that computes the sum $\sum_{i=1}^{n}$. Use only the basic arithmetic operation substraction and addition.

  Exemplify how your algorithm works step-by-step by using as input the integer $n = 7$ and provide the value of each of your used variables in each step of your executed algorithm.

Problem (O, Ω, *and* Θ*-Notation*)

▶ Here you should be able to prove or indicate if for a given function f it holds that $f \in O(g)$, $f \in \Omega(g)$ or $f \in \Theta(g)$ (it might be an extra task to determine g).

Example: (Examples might be provided in a very simplified form to give you an idea)

- Prove in detail that $T(n) = (3n + 2)^4 \in O(n^4)$
- Indicate with *yes* or *no* if $\frac{1}{3}2^n \in O(3n^2)$
- Determine an asymptotically tight bound $g(n)$ such that $f(n) = \frac{1}{3}2^n \in \Theta(g(n))$.

Problem (*Time and Space Complexity and Master Theorem*)

▶ Here you should be able to determine for a given algorithm its space and time complexity. This might also include application of the Master Theorem.

Example: (Examples might be provided in a very simplified form to give you an idea)

- Provide the exact bounds for the time complexity $T(n) \in \Theta(..)$ and space complexity $S(n) \in \Theta(..)$ of the following program `Sum(n)` in Θ-notation, assuming a unit-cost model.

```
Sum(int $n$)
```

```
$total\_sum := 0$
FOR ($i = 1$ to $n$) DO
  $total\_sum := total\_sum + i$
PRINT $total\_sum$
```

- Given is the following pseudo-code of a divide-and-conquer approach.

  ```
  Some_Rec($n$)
  IF ($n > 1$) THEN
      print("hello")
      Some_Rec($n/2$) + Some_Rec($n/2$)
  ```

  Call Some_Rec(10) and provide the output of the print command in each of the single recursive calls in the order they appear.

  Use the Master Theorem to determine the runtime $T(n)$ of Some_Rec(n), assuming a unit-cost model. In particular, specify a, b, and d as well as the function f such that $T(n) \in \Theta(f)$.

- What complexity $T(n) \in \Theta(..)$ would result from the Master theorem for the recurrence equation $T(n) = 2T(n/4) + \Theta(n^{10})$?

Problem (*Sorting*)

▶ Here you should be able to apply any of the sorting algorithms as provided in the lecture.

Example: (Examples might be provided in a very simplified form to give you an idea)

- Given is the array $A = [2, 3, 1, 5, 4]$. Apply counting_sort as provided in the lecture to sort A. For each of the steps in counting_sort provide the auxilary array B used to sort B as well as the array C used to count the occurences of keys (here $k = 5$).

Problem (*Searching*)

▶ Here you should be able to apply any of the searching algorithms as provided in the lecture.

Example: (Examples might be provided in a very simplified form to give you an idea)

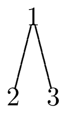
- Given is the array $A = [2, 3, 1, 5, 4]$. Use binary_search as provided in the lecture and provide the search steps for searching for element 5. Specify for each step, in which part of the array you are searching and shortly state how this decision has been derived in each step.

Problem (*Graphs and Trees*)

▶ Here you should be able to apply any of the results we established for graphs and tree, determine structural properties provide proofs of certain statements

Example: (Examples might be provided in a very simplified form to give you an idea)

- Given is the following rooted tree T.

  Specify the height of T and if T is nearly complete.

  Write the keys in the order they are visited in T using post-order traversal

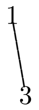
- Prove in detail the handshake-lemma.

Problem (*Binary Search Trees / AVL trees / Red-Black Trees*)

▶ Here you should be able to apply any of the results we established for binary search trees (including their subclasses)

Example: (Examples might be provided in a very simplified form to give you an idea)

- For the given sequence of integers $1, 3, 2$ build the binary search tree, that is, insert the integers one after another as keys into an initially empty binary tree and draw the tree after each insertion.
- Given is the AVL tree T



Insert 5 into the tree T and draw the resulting tree “$T + 5$” including the balance factors next to each vertex. Specify the rotation(s) for rebalancing and draw the resulting tree after rebalancing it.

Problem (*Hashing*)

▶ Here you should be able to apply any of the results we established in the section hashing (**e.g. chaining, probing, bloom filters**)

Example: (Examples might be provided in a very simplified form to give you an idea)

- Given is a hash table H of size $m = 17$.

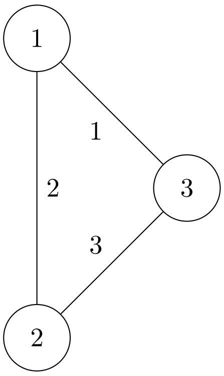
  Insert the keys `16,17` in this order into the initially empty hash table H using the hash function $h(k) = k \bmod m$.

Problem (*Elementary Graph Algorithms*)

▶ Here you should be able to apply any of the results we established in the section Elementary Graph Algorithms (**e.g. BFS, DFS, Kruskal**)

Example: (Examples might be provided in a very simplified form to give you an idea)

- Given is the following undirected weighted graph G.



Use `BFS`(1) and provide the order in which the vertices are visited (if several choices are possible prioritize the vertices from small to large). How many spanning tree does G have? How many of them are minimum spanning trees?