Algorithms & Complexity
Lecture 3: Sorting

October 1, 2019

CentraleSupélec / ESSEC Business School

Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

Thu		Topic
Thu, 12.09.2019	PM	Introduction, Combinatorics, O-notation, data structures
Tue, 24.09.2019	PM	Sorting algorithms I
➨ Tue, 1.10.2019	PM	Sorting algorithms II, recursive algorithms
Tue, 8.10.2019	PM	Greedy algorithms
Tue, 15.10.2019	PM	Dynamic programming
Thu, 31.10.2019	AM	Randomized Algorithms and Blackbox Optimization
Tue, 5.11.2019	PM	Complexity theory I
Tue, 26.11.2019	PM	Complexity theory II
Tue, 17.12.2019	AM	Exam (written)

discussion home exercises

Discussion Home Exercise

Exercise 1: Connected Components

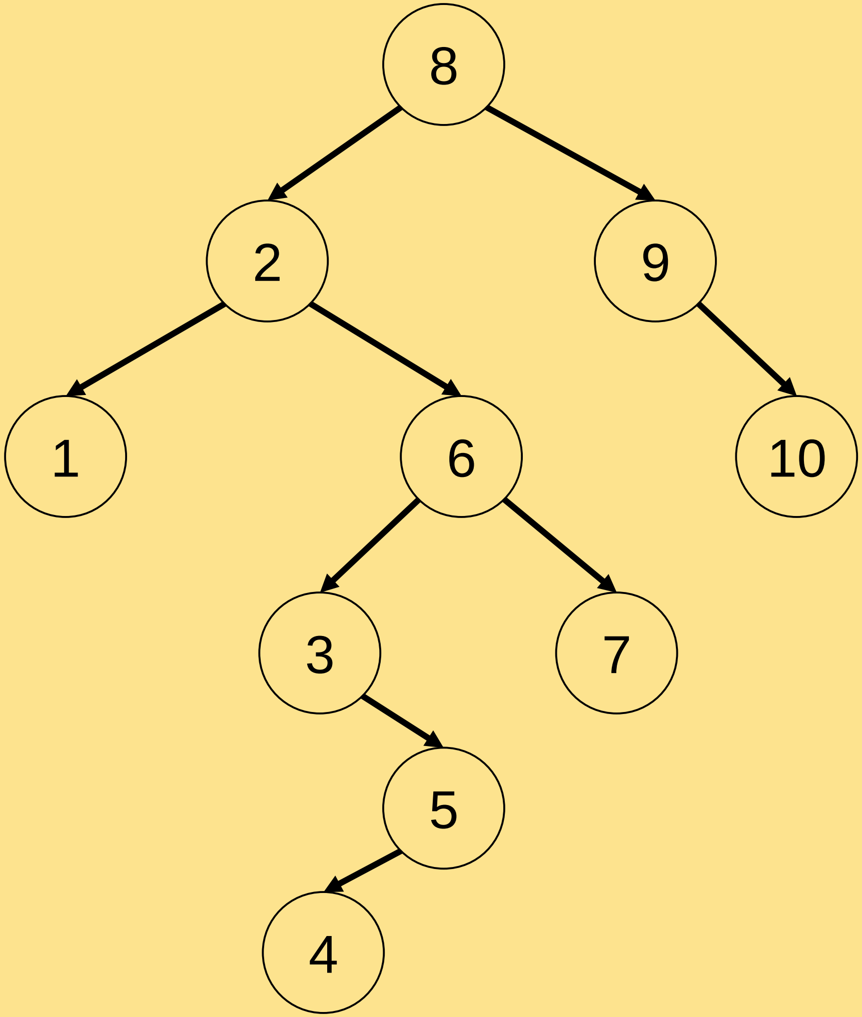
only two possibilities:

- new edge added *within* a connected component:
  # connected components ±0
- new edge added *"in between"* two connected components:
  # connected components +1

Discussion Home Exercise

Exercise 2: Binary Search Tree

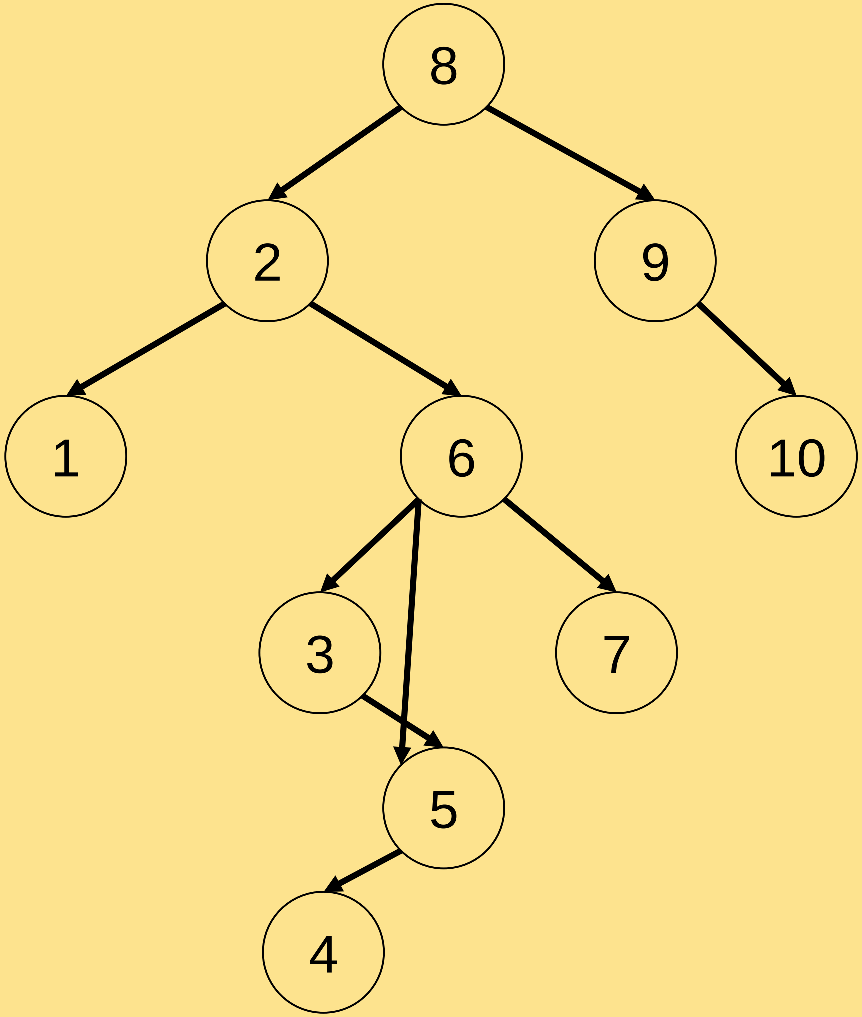
add 8, 9, 2, 10, 6, 1, 3, 7, 5, 4:



Discussion Home Exercise

Exercise 2: Binary Search Tree

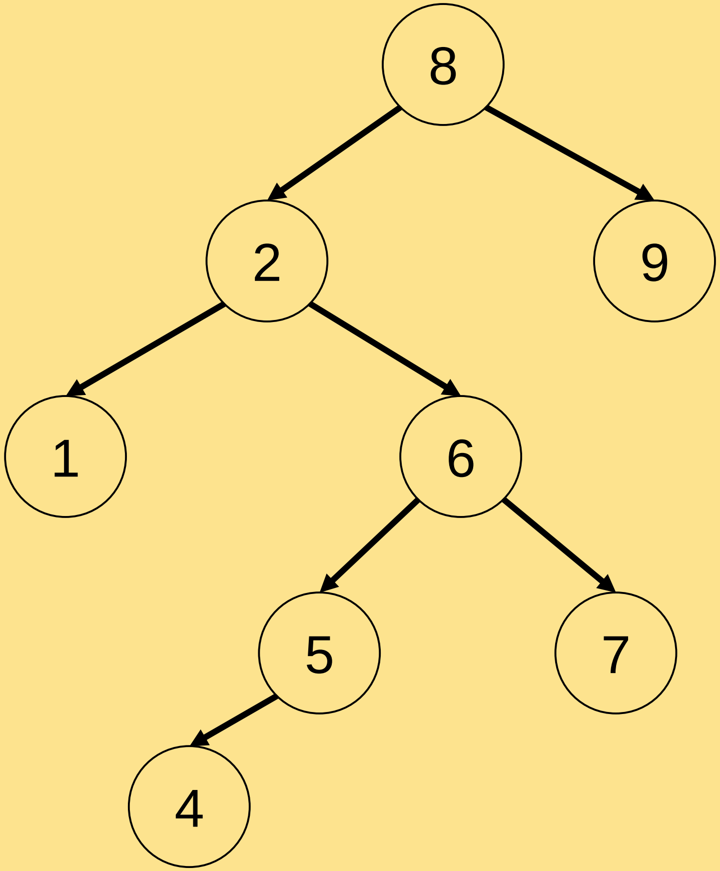
remove 10, 3, 8:



Discussion Home Exercise

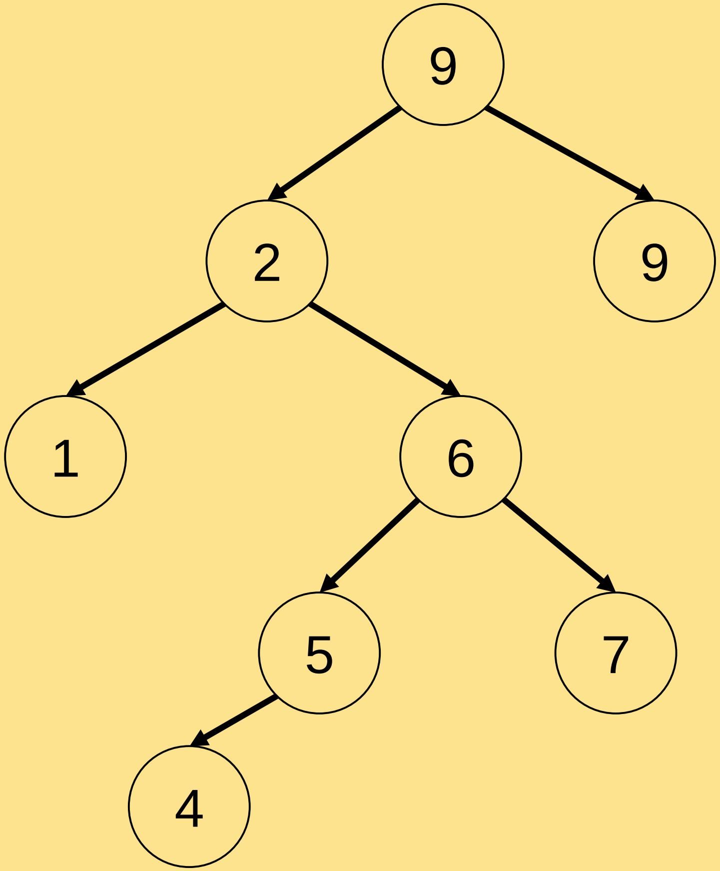
Exercise 2: Binary Search Tree

remove 10, 3, 8:



Exercise 2: Binary Search Tree

remove 10, 3, 8:



Discussion Home Exercise

Exercise 3: DFS/BFS

assumption (important): children stored from left to right

DFS order: 1, 2, 5, 6, 3, 7, 4, 8, 9, 10

BFS order: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Discussion Home Exercise

Exercise 4: Hashing with $h(x) = x \bmod 19$

Insert the (key, value) pairs

(key, value)
(63, “one”)
(388, “two”)
(296, “three”)
(68, “four”)
(160, “five”)
(264, “six”)
(10, “seven”)
(85, “eight”)

address	
0	
…	
6	(63, “one”)
7	
8	(388, “two”)
9	(160, “five”)
10	(10, “seven”)
11	(296, “three”)
12	(68, “four”)
13	(85, “eight”)
14	
…	
17	(264, “six”)
18	


now: sorting...
BY NC ND
jwhittenburg

Exercise: Sorting

Aim: Sort a set of numbers

Questions:

- What is the underlying algorithm you used?
- How long did it take to sort?
  - What is a good measure?
- Is there a better algorithm or did you find the optimal one?

Overview of Today's Lecture

Sorting

- Insertion sort
- Insertion sort with binary search
- Mergesort
- Timsort idea

Exercise

- Comparison of sorting algorithms

Essential vs. Non-Essential Operations

In sorting, we distinguish

- comparison- and non-comparison-based sorting
- in the former, we distinguish further:
  - comparisons as essential operations
    - they are comparable over computer architectures, operating systems, implementations, (historic) time
    - they can take more time than other operations, e.g. when we compare trees w.r.t. their lexicographic DFS sorting
  - other non-essential operations: additions, multiplications, shifts/swaps in arrays, …

Insertion Sort

Idea:

for k from 1 to n-1:

- assume array a[1]…a[k] is already sorted
- insert a[k+1] correctly into a[1]…a[k+1]

swapping a[k+1] with all other numbers larger than a[k+1]

6 5 3 1 8 7 2 4

see also https://en.wikipedia.org/wiki/Insertion_sort

Insertion Sort: Analysis

Worst case:

- reverse ordering: insert always to the beginning
- then $1 + 2 + 3 + \cdots + (n-1) \ = \Theta(n^2)$ comparisons needed

Average Case:

- even here: $\Theta(n^2)$ comparisons needed (without proof)

Insertion Sort with Binary Search

Idea for an improved version:

use binary search for the right position of new entry in sorted subarray

- to insert array element $a[i]$, we need $\lceil \log(i+1) \rceil$ comparisons in worst case (= depth of the binary tree search)
- overall, therefore

$$\sum_{1 \leq i \leq n-1} \lceil \log(i+1) \rceil = \sum_{2 \leq i \leq n} \lceil \log(i) \rceil < \log(n!) + n$$

  comparisons are needed

- from last time, we know that

$$\log(n!) \leq e n^{n+\frac{1}{2}} e^{-n} = n \log(n) - n \log(e) + O(\log(n))$$

in total, insertion sort with binary search needs

$$n \log(n) - 0.4426n + O(\log(n))$$

comparisons in the worst case.

Another Possible Sorting Idea:

- sort first and second half of the array independently
- then merge the pre-sorted halves:
  - take the smaller of the smallest two values each time

$\text{Mergesort}(a_1, \dots, a_n)$

if $n = 1$ then stop

if $n > 1$ then:

- $(b_1, \dots, b_{\lceil n/2 \rceil}) = \text{Mergesort}(a_1, \dots, a_{\lceil n/2 \rceil})$
- $(c_1, \dots, c_{\lfloor n/2 \rfloor}) = \text{Mergesort}(a_{\lceil n/2 \rceil + 1}, \dots, a_n)$
- return $(d_1, \dots, d_n) = \text{Merge}(b_1, \dots, b_{\lceil n/2 \rceil}, c_1, \dots, c_{\lfloor n/2 \rfloor})$

Mergesort

Another Possible Sorting Idea:

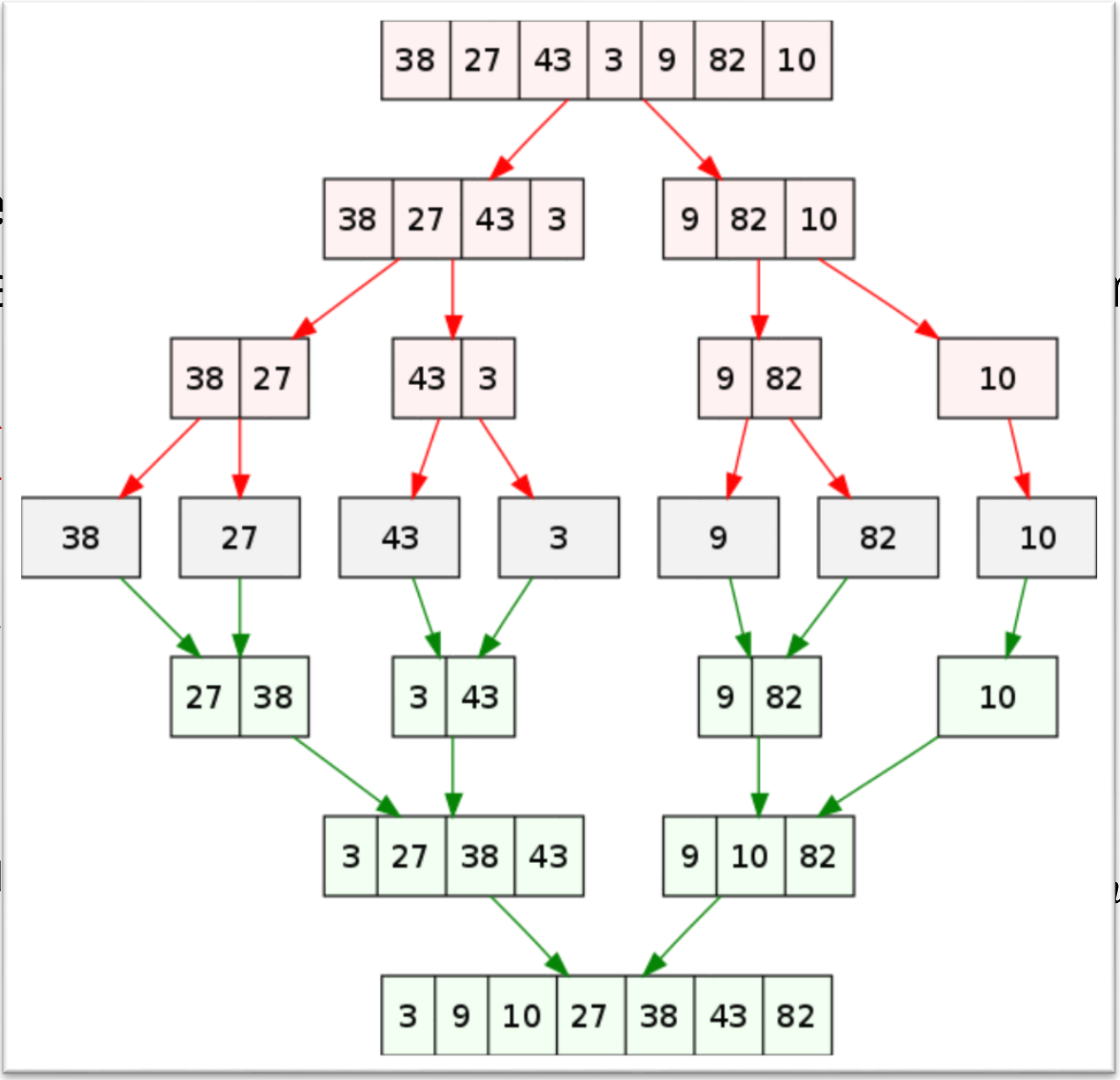
- sort first an
- then merge
  - take the ... ne

Mergesort(

if $n = 1$

if $n > 1$

- $(b_1,$
- $(c_1,$
- retu ... $/2\rfloor)$



Mergesort: Runtime

- the number of essential comparisons C(n) when sorting n items with Mergesort is

$$C(1) = 0, \qquad C(n) = C\left(\left\lceil \frac{n}{2} \right\rceil\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1$$

sorting left half | sorting right half | merging

- without proof, $C(n) = n\log(n) + n - 1$ if $n = 2^k$

Remarks:

Mergesort is practical for huge data sets, that don't fit into memory

Mergesort is a recursive algorithm (= calls itself)

…solves a problem by solving smaller sub-problems first

Python's Sorting: Timsort

- python uses a combination of Mergesort with insertion sort

  https://en.wikipedia.org/wiki/Timsort
- insertion sort for small arrays quicker than merging from n=1 (can be done in memory/cache)
- in addition, Timsort searches for subarrays which are already sorted (called "natural runs") and that are handled as blocks
- worst case runtime of $O(n\log(n))$, best case: $O(n)$

Lower Bound for Comparison-Based Sorting

- Insertion Sort, standard: $\Theta(n^2)$
- Insertion Sort with binary search: $n\log(n) - 0.4426n + O(\log(n))$
- Mergesort: $n\log(n) + n - 1$ if $n = 2^k$

Can we do better than $n\log(n)$?

- No! [at least for comparison-based sorting]
- Lower bound for comparison-based sorting of $\Omega(n\log(n))$

without proof here

Exercise in Python

Comparing sorting algorithms in python

Goals:

- learn about Mergesort (and how to implement it)
- observe the differences in runtime between your own Mergesort and python's internal Timsort
- learn how to do a scientific (numerical) experiment and how to report the results

Exercise in Python

TODOs:

❶ implement your own Mergesort e.g. based on lists
http://www.cmap.polytechnique.fr/~dimo.brockhoff/algorithmsandcomplexity/2019/schedule.php

❷ compare the differences in runtime between your own Mergesort and python's internal Timsort (`'sorted(…)'`) on randomly generated lists of integers

❸ plot the times to sort 1,000 lists of equal length n with both algorithms for different values of $n \in \{10, 100, 1\,000, 10\,000\}$

Tip:

```
>>> import timeit
>>> timeit.timeit('your code', number=1000)
```

Another (even more important) Tip:

use the "?" to get help on a module (and "??" to inspect the code)

Conclusions

I hope it became clear...

...what sorting is about and how fast we can do it