Algorithms & Complexity

September 12, 2019

CentraleSupélec / ESSEC Business School

Dimo Brockhoff

Inria Saclay – Ile-de-France

Why Algorithms & Complexity?

Algorithm

(noun.)

Word used by programmers when they do not want to explain what they did.

Why Algorithms & Complexity?

[…] an algorithm is a set of instructions, typically to solve a class of problems or perform a computation.
[from wikipedia]

Algorithms widespread in almost every aspect of the "real-world"

- (automatic) problem solving
- sorting
- accessing data in data structures
- …

Example: Sorting

Aim: Sort a set of cards/words/data

Re-formulation: minimize the “unsortedness”

E F C A D B

B A C F D E

A B C D E F

$\downarrow$ sortedness increases

Example: Sorting

Classical Questions:

- What is the underlying algorithm?
  (How do I solve a problem?)
- How long does it run to solve the problem?
  (How long does it take? Which guarantees can I give? What is its convergence rate?)
- Is there a better algorithm or did I find the optimal one?

Be Aware

Caution:

- This is not an “algorithms for data scientists” lecture
  - we do not cover algorithms for regression, regularization, dimensionality reduction, clustering, deep learning, …
  - …but cover much more basic things:
    - data structures
    - data sorting
    - fundamental algorithm design ideas
    - how to analyze an algorithm
    - how to prove lower runtime bounds for hard problems
    - …

What we plan to do in the A&C lecture

Learning Goals:

❶ know basic design principles behind good algorithms (*“building blocks to help solving “your own” problems”*)

❷ be able to analyze theoretically some algorithms
- give strong bounds on their “effectiveness”
- understand the ideas of (worst case) algo complexity (*"Am I too dumb to find a quick algorithm or can nobody do better?"*)

❸ be able to use and understand existing algorithms (*“practice, practice, practice!”*)

What we plan to do in the A&C lecture

How are we going to do that?

- look at a lot of examples of algorithms
- mixture of lectures and small exercises
- practice and theory
- additionally 1 home exercise per week

Please ask questions
if things are unclear throughout the course!

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)

Remarks on Exercises I

- included within the lecture (typically 1/3 of it)
- expected to be done on paper or in python
- hence, please make sure you have python installed on your laptop until the second lecture
- Anaconda is the recommended way to get there:

  `https://www.anaconda.com/distribution/`
- (basic) example solutions will be made available afterwards
- not graded but please see it as training for the exam

Remarks on Exercises II

In addition:

- 7 home exercises with 20 points each
- Counts 1/3 to overall grade (exam is the other 2/3)

Remarks on Exercises II

In addition:

- 7 home exerc
- Counts 1/3 to
- Graded as:

Achieved points	grade	Difference
$136 \leq p \leq 140$	20	4
$132 \leq p < 136$	19	4
$128 \leq p < 132$	18	4
$124 \leq p < 128$	17	4
$118 \leq p < 124$	16	6
$112 \leq p < 118$	15	6
$106 \leq p < 112$	14	8
$98 \leq p < 106$	13	8
$90 \leq p < 98$	12	8
$80 \leq p < 90$	11	10
$70 \leq p < 80$	10	10
$60 \leq p < 70$	9	10
$50 \leq p < 60$	8	10
$40 \leq p < 50$	7	10
$34 \leq p \leq 40$	6	6
…	1..5	6, 6, 6, 6, 6
$0 \leq p <4$	0	4

Remarks on Exercises II

In addition:

- 7 home exercises with 20 points each
- Counts 1/3 to overall grade (exam is the other 2/3)
- Graded as explained before
- Group submissions of 2 students allowed (and even encouraged!)
- But: maximally 4 submissions with the same student pair

The Exam

- Tuesday, 17th December 2019 in the morning (3 hours)
- open book: take as much material as you want
- but: no electronic devices allowed that connect to the internet
- (most likely) multiple-choice with 20-30 questions

All information also available at

`http://www.cmap.polytechnique.fr/~dimo.brockhoff/algorithmsandcomplexity/2019/`

(exercise sheets, lecture slides, additional information, links, ...)

any questions?

Overview of Today's Lecture

Basics

- Fundamental combinatorics
- notations such as the O-notation
- algorithms on basic data structures
  - arrays
  - lists
  - trees
  - …

Basics I: Combinatorics

For this and the next parts, a nice-to-read reference is
https://www.math.upenn.edu/~wilf/AlgoComp.pdf

Combinatorics = Counting

counting combinations and counting permutations

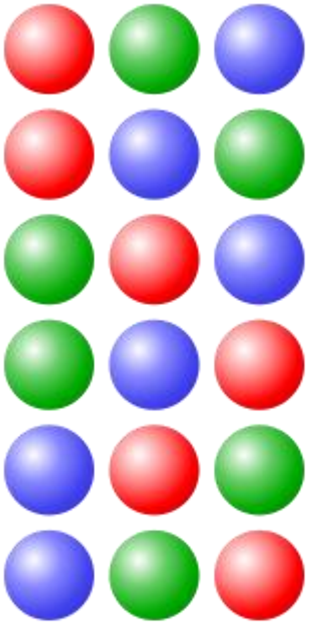
Why combinatorics?

- In order to compute probabilities

$$P(event) = \frac{\#\text{favorable outcomes}}{\#\text{possible outcomes}}$$

- Related to graph theory (later)
- Related to combinatorial optimization (later)

Permutation: a sequence/order of members of a set



Watchduck (a.k.a. Tilman Piesk)

How many different orders exist on $[n] := \ 1, \ldots, n$?

- First integer: choice among n
- Second integer: choice among n-1
- Last integer: no choice among 1

- In total: $n \cdot (n-1) \cdot \ldots \cdot 1 \ =: n!$

How to Generate a Random Permutation?

Idea: generate a random vector, sort it and use the generated sorting order as the permutation

```
import numpy as np
n = 4
random_array = np.random.rand(n)
random_perm = np.argsort(random_array)
```

More elegant way:

```
random_perm = np.random.permutation(n)
```
☺

Combinations Without Replacement (*k*-combination)

How many combinations of set members of a given size exist?

Example: number of different poker hands

- 52*51*50*49*48 = 311,875,200 ways to hand 5 cards out of 52
- but: order does not matter here!
- There are 5! = 120 orders of 5 cards
- Hence, there are 311,875,200/120 = 2,598,960 distinct pokers hands in total

In general, the number of k-combinations of n items (without replacements) is

$$\binom{n}{k} := \frac{n!}{k!\,(n-k)!}$$

Combinations with replacement

What if we want to allow duplicates?

- combinations with replacement
- also known as k-combination with repetitions or k-multicombination

Example:

Combinations with replacement

Wh

-
- ation

Exa

Combinations with replacement

What if we want to allow duplicates?

- combinations with replacement
- also known as k-combination with repetitions or k-multicombination

Example:

eat 3 donuts from a choice of 4 different ones

Combinations with replacement

What if we want to allow duplicates?

- combinations with replacement
- also known as k-combination with repetitions or k-multicombination

Example:

eat 3 donuts from a choice of 4 different ones

Number of k-combinations with replacement:

$$\binom{n+k-1}{k} \left[= \binom{n+k-1}{n-1} \right]$$

Here with $n = 4$, $k = 3$: $\binom{4+3-1}{3} = \binom{6}{3} = 20$ combinations

Why That? The Stars and Bars Method

Stars and Bars: A useful counting method popularized by W. Feller*

How many combinations to put k objects into n bins?

- objects: stars
- bins: separated by bars

- Example of n=5 bins and k=7 objects: ★ ★ |★|| ★ ★ ★ | ★
- Donut example: n=4 bins/donut types, k=3 objects

Number of combinations to put k objects into n bins
= number of combinations to place k objects on n+k-1 places ⇨ $\binom{n+k-1}{k}$
= number of combinations to place n-1 bars on n+k-1 places ⇨ $\binom{n+k-1}{n-1}$

How to Generate a Random k-Combination?

Naïve way:

```
from itertools import combinations
import numpy as np

n = 4
k = 2
# all k-combinations of [0, 1, …, n-1]:
comb = list(combinations(np.arange(n), k))

# pick one at random
random_k_combination =
    comb[np.random.randint(len(comb))]
```

Works only for small enough n and k:

`len(comb)` is 15,890,700 for n=50 and k=6

and 99,884,400 for n=50 and k=7

How to Generate a Random k-Combination?

More efficient way:

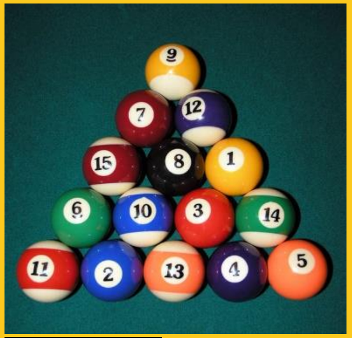
- iterate across each element of {1,…,n}
- pick each element with a dynamically changing probability of

$$\frac{k - \#samples\ chosen}{n - \#samples\ visited}$$

until k elements are picked.

Exercise

a) In how many different ways can the 15 balls of a pool billiard be placed (on a line)?



b) How many different combinations of five coins (Euros) can you have in your pocket?

c) How likely is it to get your bike stolen with the lock on the right?

Solutions

a) 15! (we look for the number of permutations of 15 distinct balls)

b) (8+5-1) choose 5 = 792 (8 different coins, choose 5 with repetition)

c) it's pretty safe: the probability to find the right number is $\frac{1}{10^5} = 10^{-5}$, assuming that a random number out of all $10 \cdot 10 \cdot 10 \cdot 10 \cdot 10 = 10^5$ lock numbers is tried. It takes >10min to try out 1% of all 10^5 numbers if you try 2 lock combinations per second.

Basics II: The O-Notation

Excursion: The O-Notation

Motivation:

- we often want to characterize how quickly a function $f(x)$ grows asymptotically
- e.g. we might want to say that an algorithm takes quadratically many steps (in n) to find the optimum of a problem with n (binary) variables, it is never exactly n^2, but maybe $n^2 + 1$ or $(n+1)^2$

Big-O Notation

should be known, here mainly restating the definition:

Definition 1 We write $f(x) = O(g(x))$ iff there exists a constant $c > 0$ and an $x_0 > 0$ such that $|f(x)| \leq c|g(x)|$ holds for all $x > x_0$

we also view O(g(x)) as the set of all functions growing at most as quickly as g(x) and write f(x)$\in$O(g(x))

Big-O: Examples

- $f(x) + c = O(f(x))$ [as long as f(x) does not converge to zero]
- $c \cdot f(x) = O(f(x))$
- $f(x) \cdot g(x) = O(f(x) \cdot g(x))$
- $3n^4 + n^2 - 7 = O(n^4)$

Intuition of the Big-O:

- if $f(x) = O(g(x))$ then g(x) gives an upper bound (asymptotically) for f
- constants don't play a role
- with Big-O, you should have '≤' in mind

Excursion: The O-Notation

Further definitions to generalize from '≤' to '≥' and '=':

- $f(x) = \Omega(g(x))$ if $g(x) = O(f(x))$
- $f(x) = \Theta(g(x))$ if $f(x) = O(g(x))$ and $g(x) = O(f(x))$

Note: Definitions equivalent to '<' and '>' exist as well, but are not needed in this course

Exercise O-Notation

Please order the following functions in terms of their asymptotic behavior (from smallest to largest):

- $\exp(n^2)$
- $\log n$
- $\ln n / \ln \ln n$
- n
- $n \log n$
- $\exp(n)$
- $\ln(n!)$

Give for two of the relations a formal proof.

Exercise O-Notation (Solution)

Correct ordering:

$\frac{\ln(n)}{\ln(\ln(n))} = O(\log n)$ log n = O(n) n = O(n log n)

n log n = Θ(ln(n!)) ln(n!)= O(e^n) e^n = O($e^{n\wedge 2}$)

but for example $e^{n\wedge 2} \neq O(e^n)$

One exemplary proof:

$\frac{\ln(n)}{\ln(\ln(n))}$ = O(log n):

$$\frac{\ln(n)}{\ln(\ln(n))} = \frac{\log(n)}{\log(e)\ln(\ln(n))} \underset{\text{for } n>1}{\leq} \frac{3\log(n)}{\ln(\ln(n))} \underset{\text{for } n>15}{\leq} 3|\log(n)|$$

Exercise O-Notation (Solution)

One more proof: ln n! = O(n log n)

- Stirling's approximation: $n! \sim \sqrt{2\pi n}\,(n/e)^n$ or even

$$\sqrt{2\pi}\, n^{n+1/2} e^{-n} \le n! \le e\, n^{n+1/2} e^{-n}$$

- $\ln n! \le \ln(e n^{n+\frac{1}{2}} e^{-n}) = 1 + \left(n + \frac{1}{2}\right) \ln n - n$

$$\le \left(n + \frac{1}{2}\right) \ln n \le 2n \ln n = 2n\,\frac{\log n}{\log e} = c \cdot n \log n$$

okay for $c = 2/\log e$ and all $n \in \mathbb{N}$

- n ln n = O(ln n!) proven in a similar vein

basic data structures

Why Data Structures? What are those?

A data structure is a data organization, management, and storage format that enables efficient access and modification.

More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.

from wikipedia

Why important to know?

- Only with knowledge of data structures can you program well
- Knowledge of them is important to design efficient algorithms

Data Structures and Algorithm Complexity

Depending on how data is stored, it is more or less efficient to

- Add data
- Remove data
- Search for data

Common Complexities

Complexity	Running Time	
constant	$O(1)$	independent of data size
logarithmic	$O(\log(n))$	often base 2, grows relatively slowly with data size
linear	$O(n)$	nearly same amount of steps than data points
	$O(n\log(n))$	Common, still efficient in practice if n not huge
quadratic	$O(n^2)$	Often not any more efficient with large data sets
…		
exponential	$O(2^n), O(n!)$, ...	Should be avoided ☺

see also: https://introprogramming.info/english-intro-csharp-book/read-online/chapter-19-data-structures-and-algorithm-complexity

Best, Worst and Average Cases

Algorithm complexity can be given as best, worst or average cases:

Worst case:

- Assumes the worst possible scenario
- Algorithm can never perform worse
- Corresponds to an upper bound (on runtime, space requirements, …)
- Most common

Best case:

- Best possible scenario
- Algorithm is never quicker/better/more efficient/…

Average case:

- Complexity averaged over all possible scenarios
- Often difficult to analyze

Arrays

Array: a fixed chunk of memory of constant size that can contain a given number of n elements of a given type

- think of a vector or a table
- in python:
  - **`import numpy as np`**
  - **`a = np.array([1, 2, 3])`**
  - **`a[1]`** returns **`2`** [python counts from 0!]

Common operations and their complexity:

  - Get(i) and Update(i) in constant time
  - but Remove(i), Move j in between positions i and i+1, … are not possible in constant time, because necessary memory alterations not local
  - To know whether a given item is in the array: linear time

Searching in Sorted Arrays

- Assume a sorted array $a[1] < a[2] < \ldots < a[n]$.
- How long will it take to find the smallest element $\geq k$?
  (Best case, worse case, average case)

Searching in Sorted Arrays

- Assume a sorted array $a[1] < a[2] < \ldots < a[n]$.
- How long will it take to find the smallest element $\geq k$?
  Or to decide whether a value a is in the array?
  (best case, worse case, average case)

Linear search

- go through array from $a[1]$ to $a[n]$ until entry found
- still $\Theta(n)$ in the worst case
- average case the same (if we assume that each item is queried with equal probability)

Searching in Sorted Arrays

Binary search

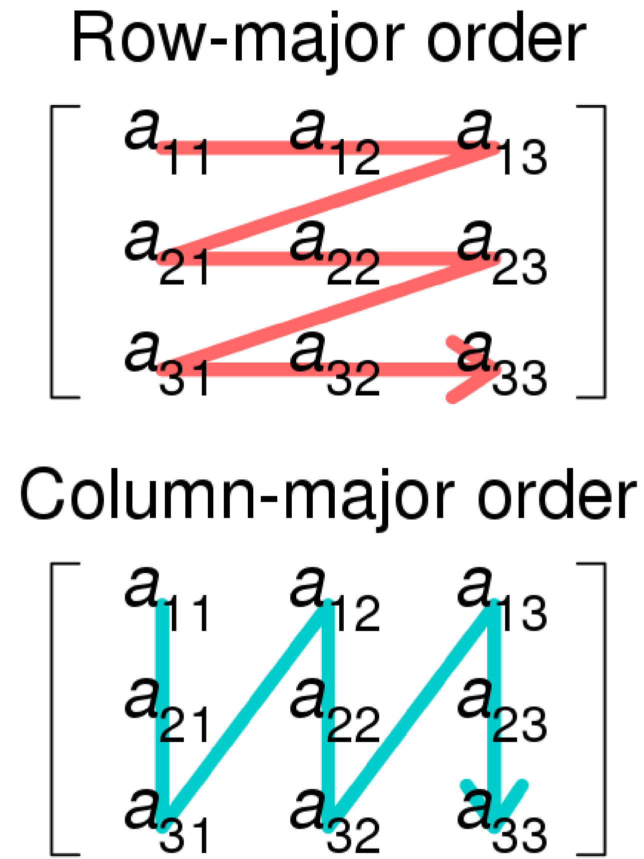
- Look at position $\lceil n/2 \rceil$ first
- Is it the sought after entry? If yes, stop
- If not: search recursively in left or right interval, depending on whether the middle entry is larger or smaller than the sought after entry

Runtimes

- Best case: 1
- Worst case:
  - sought after entry not in array
  - simple case: $n = 2^k - 1$ array elements
  - array-part where entry could be located is of length $2^{k-1} - 1$
  - by induction: maximally k comparisons needed
  - $k = \Theta(\log(n))$

Remarks: Arrays and Matrices

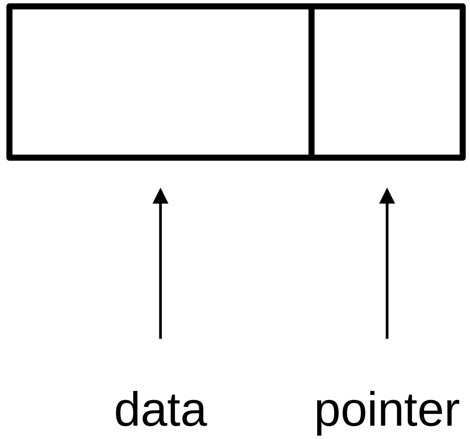
- Matrices can be stored in arrays, too
- Row first or column first?
- Storing sparse matrices efficiently: not covered here



Lists

- Dynamic data structure of varying length
- Allows to add and remove entries (remember: arrays don't)
- However, also not stored in contiguous memory anymore

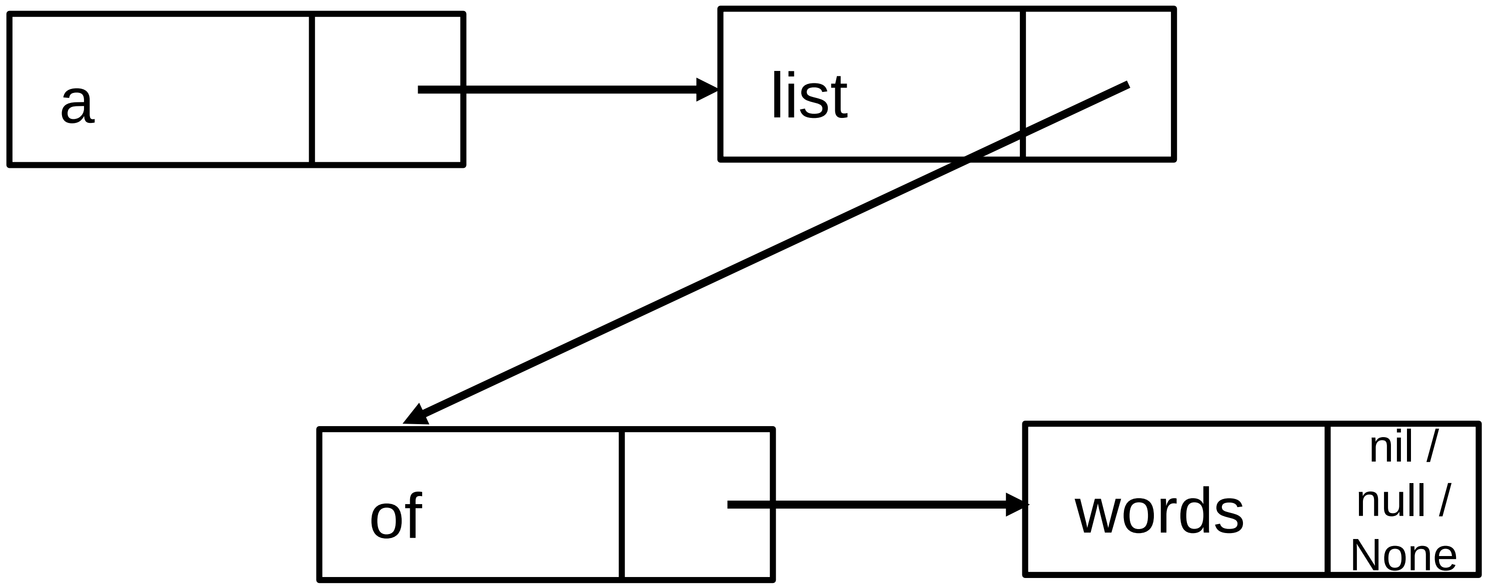
Idea of a Linear List



Lists

- Dynamic data structure of varying length
- Allows to add and remove entries (remember: arrays don't)
- However, also not stored in contiguous memory anymore

Idea of a Linear List



Lists

- Dynamic data structure of varying length
- Allows to add and remove entries (remember: arrays don't)
- However, also not stored in contiguous memory anymore

Idea of a Linear List

[4, 7, 1, …] in memory could be for example:

memory address	**…**	**87**	**88**	**89**	**90**	**91**	**92**	**93**	**…**
memory content	…	4	90	…	7	92	1	104	…

Lists

- Dynamic data structure of varying length
- Allows to add and remove entries (remember: arrays don't)
- However, also not stored in contiguous memory anymore

Idea of a Linear List

?

[4, ~~7,~~ 1, …] in memory could be for example:

memory address	…	87	88	89	90	91	92	93	…
memory content	…	4	90	…	7	92	1	104	…

Lists

- Dynamic data structure of varying length
- Allows to add and remove entries (remember: arrays don't)
- However, also not stored in contiguous memory anymore

Idea of a Linear List

[4, ~~7~~, 1, …] in memory could be for example:

memory address	…	87	88	89	90	91	92	93	…
memory content	…	4	~~90~~ 92	…	7	92	1	104	…

- go through list until 7 is found
- always keep track of last pointer (the one finally to 7)
- move this pointer to the former pointer of entry 7

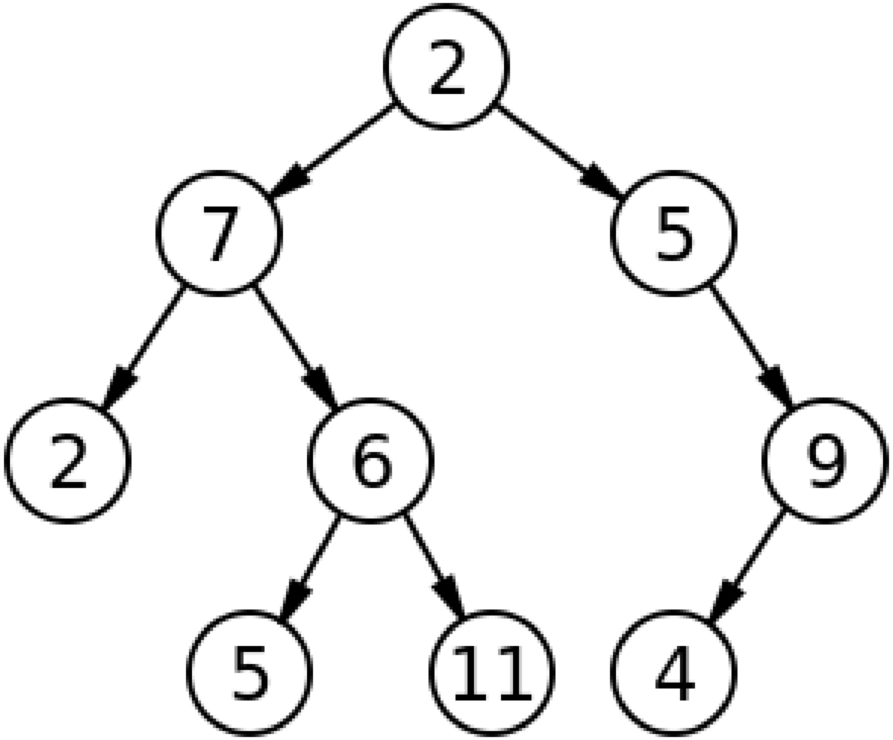
Lists

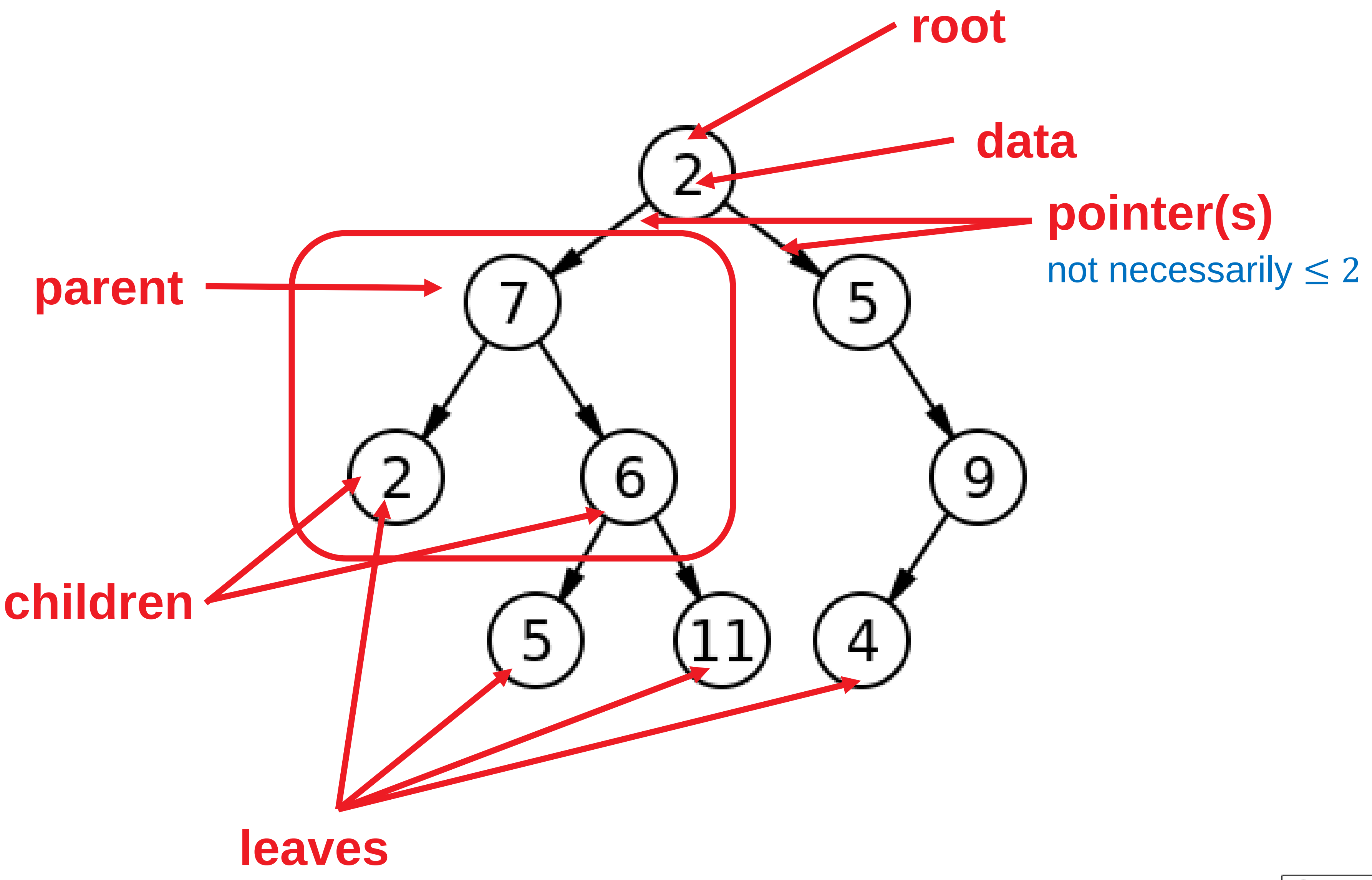
- removal of element in constant time $O(1)$
- very similar for adding: $O(1)$
- adding into a sorted list: $O(n)$
- but now searching is more difficult, even if sorted
  - reason: we don't have access to the "middle" element
  - search for element i: $\Theta(i)$ if list is sorted

we need a different data structure if we want to search, insert, and delete efficiently

Trees

Trees





root
data
pointer(s)
not necessarily ≤ 2
parent
children
leaves
2
7
5
2
6
9
5
11
4

Trees are Special Graphs

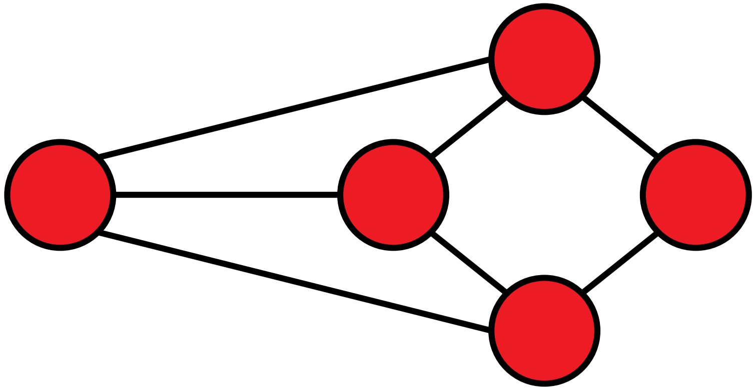
For a more formal definition, we need to introduce the concept of graphs…

Basic Concepts of Graph Theory

[following for example http://math.tut.fi/~ruohonen/GT_English.pdf]

Graphs

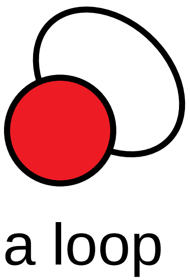
Definition 1 *An undirected graph* G *is a tupel* $G = (V, E)$ *of edges* $e = \{u, v\} \in E$ *over the vertex set* V *(i.e.,* $u, v \in V$*).*

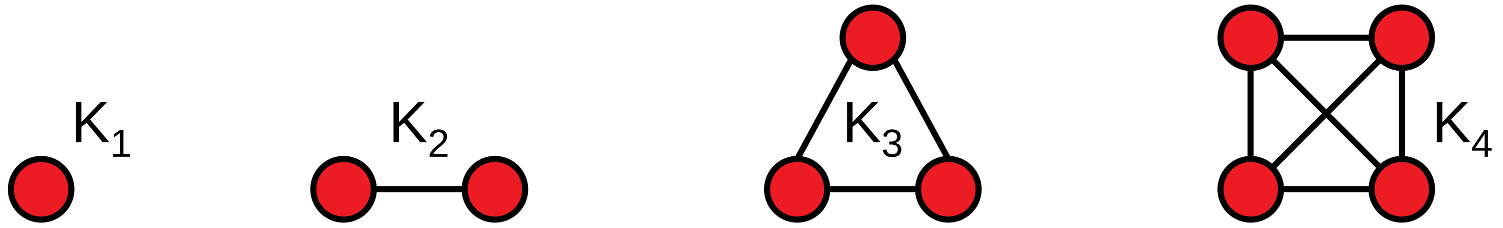


- vertices = nodes
- edges = lines
- Note: edges cover two *unordered* vertices (*undirected* graph)
  - if they are *ordered*, we call G a *directed* graph

Graphs: Basic Definitions

- G is called *empty* if E empty
- u and v are *end vertices* of an edge {u,v}
- Edges are *adjacent* if they share an end vertex
- Vertices u and v are *adjacent* if {u,v} is in E
- The *degree* of a vertex is the number of times it is an end vertex
- A complete graph contains all possible edges (once):



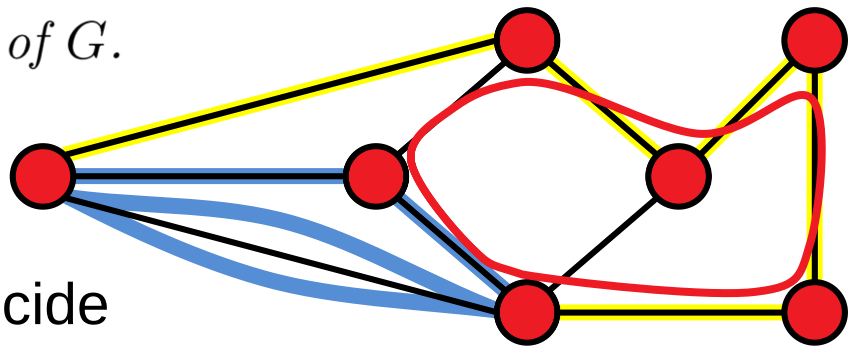


Walks, Paths, and Circuits

Definition 1 *A* walk *in a graph* $G = (V, E)$ *is a sequence*

$$v_{i_0}, e_{i_1} = (v_{i_0}, v_{i_1}), v_{i_1}, e_{i_2} = (v_{i_1}, v_{i_2}), \ldots, e_{i_k}, v_{i_k},$$

alternating vertices and adjacent edges of G.



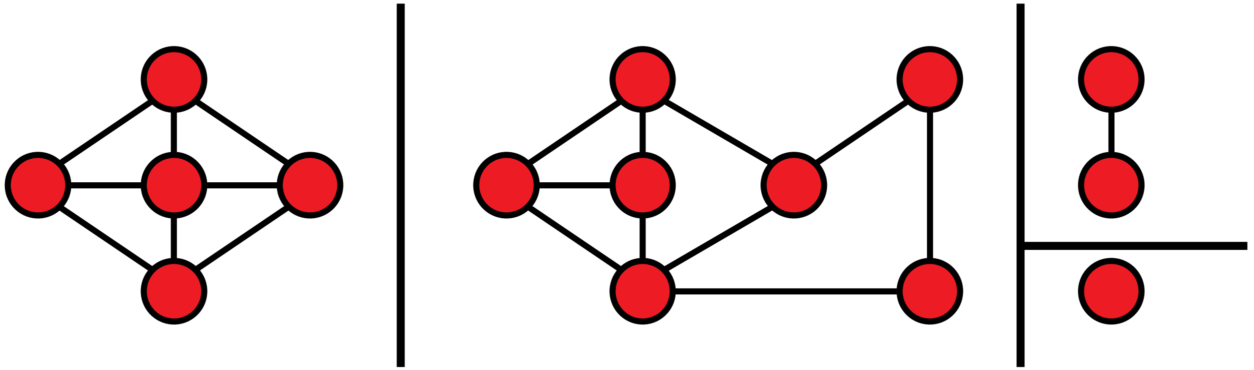
A walk is

- *closed* if first and last node coincide
- a *trail* if each edge traversed at most once
- a *path* if each vertex is visited at most once

- a closed path is a *circuit* or *cycle*
- a closed path involving all vertices of G is a *Hamiltonian cycle*

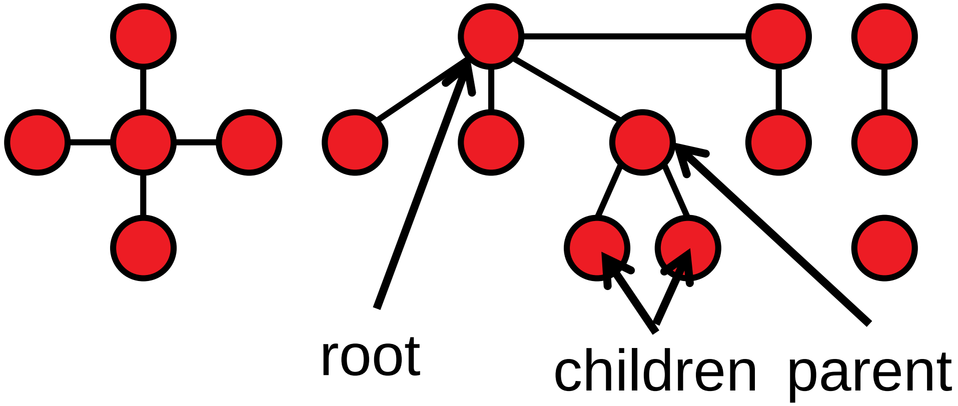
Graphs: Connectedness

- Two vertices are called *connected* if there is a walk between them in G
- If all vertex pairs in G are connected, G is called connected
- The *connected components* of G are the (maximal) subgraphs which are connected.

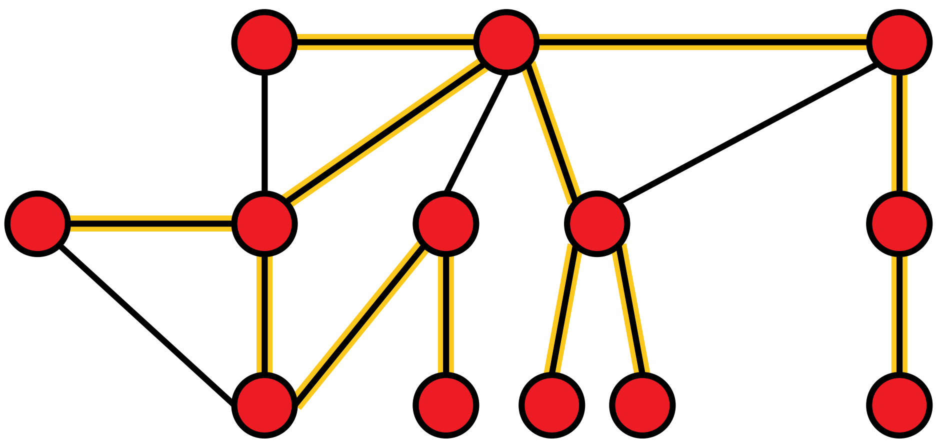


Trees and Forests

- A *forest* is a cycle-free graph
- A *tree* is a connected forest



A *spanning tree* of a connected graph G is a tree in G which contains all vertices of G



Depth-First Search (DFS)

Sometimes, we need to traverse a graph, e.g. to find certain vertices

Depth-first search and breadth-first search are two algorithms to do so

Depth-first Search (for undirected/acyclic and connected graphs)

❶ start at any node x; set i=0

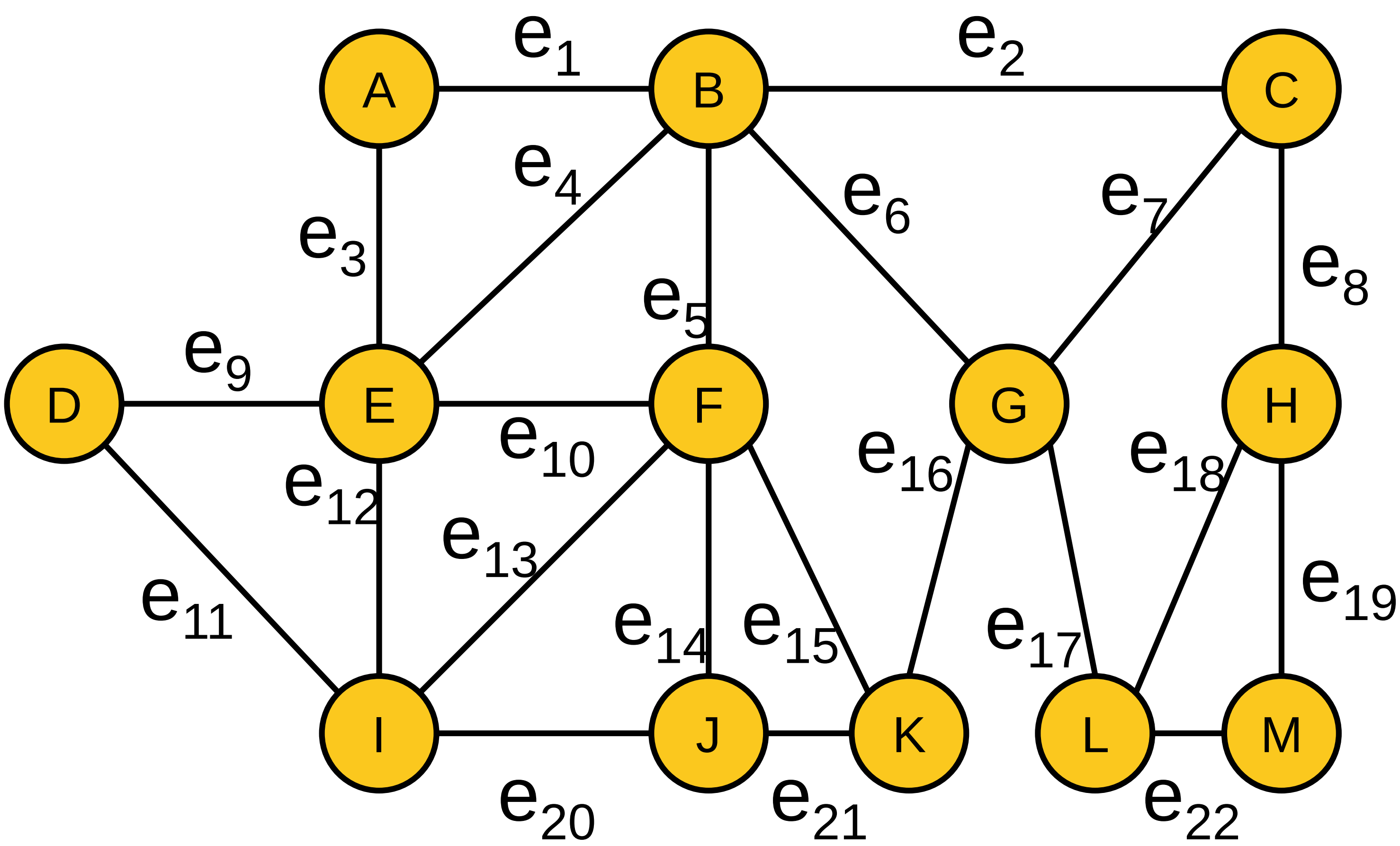
❷ as long as there are unvisited edges {x,y}:

- choose the next unvisited edge {x,y} to a vertex y and mark x as the parent of y
- if y has not been visited so far: i=i+1, give y the number i, and continue the search at x=y in step 2
- else continue with next unvisited edge of x

❸ if all edges {x,y} are visited, we continue with x=parent(x) at step 2 or stop if x==v0

DFS: Stage Exercise

Exercise the DFS algorithm on the following graph!



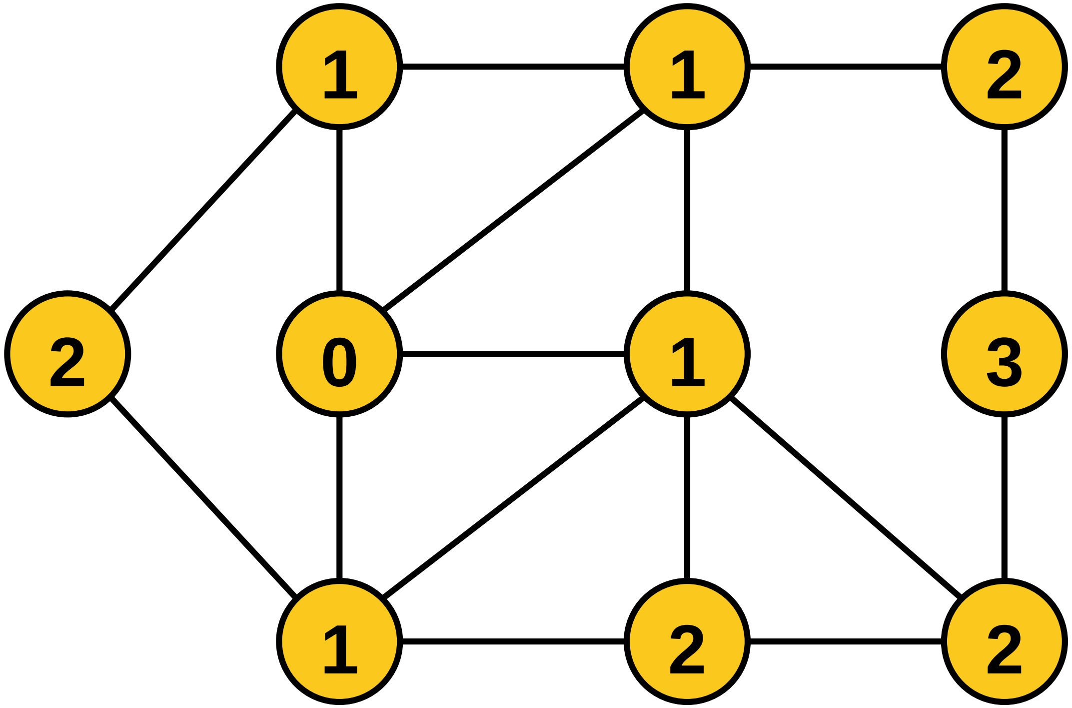
Breadth-first Search (for undirected/acyclic and connected graphs)

❶ start at any node x, set i=0, and label x with value i

❷ as long as there are unvisited edges {x,y} which are adjacent to a vertex x that is labeled with value i:

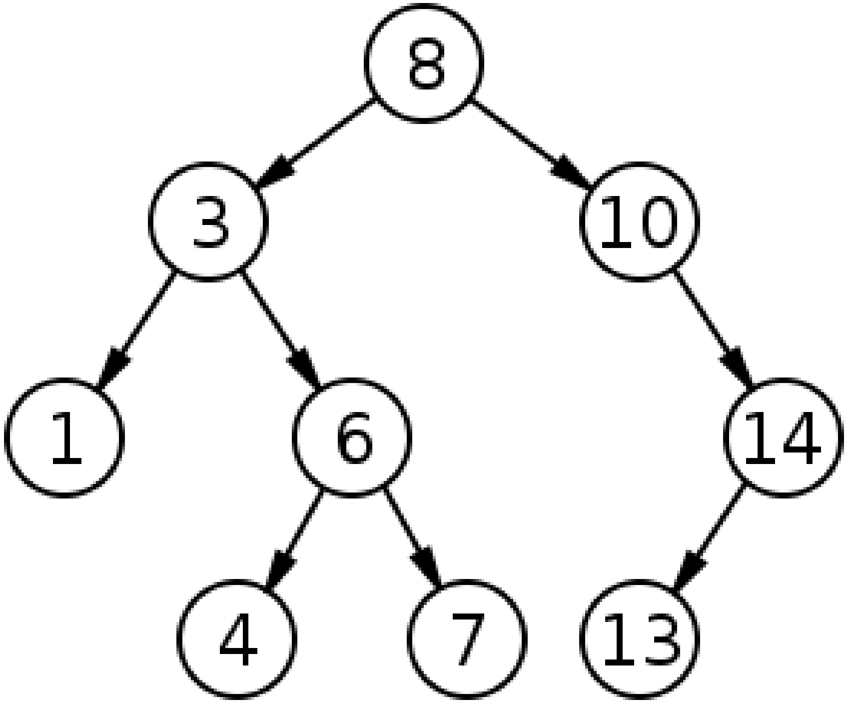
- label all vertices y with value i+1

❸ set i=i+1 and go to step 2



Binary Search Tree

- a tree with degree ≤ 2
- children sorted such that the left subtree always contains values smaller than the corresponding root and the right subtree only values larger



Class Exercise: Filling a Binary Search Tree

Round 1:

give an integer to be filled into our tree

Round 2:

tell where the next integer inserts

Binary Search Tree: Complexities

Search

- similar to binary search in array (go left or right until found)
- $O(\log(n))$ if tree is well balanced
- $\Theta(n)$ in worst case (linear list)

Insertion

- first like search to determine the parent of the new node
- then add in $O(1)$ [we are always at a leaf]

Remove (more tricky)

- if node has no child, remove it
- if node has a single child, replace node by its child
- if node has two children: find left-most tree entry L larger than the to-be-removed node, copy its value to the to-be-removed node, and remove L according to the two above rules
- cost: $O(\text{tree depth})$, in worst case: $\Theta(\text{n})$

Binary Trees: Can We Do Better?

Binary Search Tree

average case (random inserts) | worst case

search	insert	delete	search	insert	delete
$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Guarantee a balanced tree:

- AVL trees
- B trees
- Red-Black trees
- …

average case (random inserts) | worst case

search	insert	delete	search	insert	delete
$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

Can We Do Even Better on Average?

Balanced Trees

average case (random inserts)			worst case		
search	insert	delete	search	insert	delete
$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$

?

average case (random inserts)			worst case		
search	insert	delete	search	insert	delete
$\boldsymbol{\Theta(1)}$	$\boldsymbol{\Theta(1)}$	$\boldsymbol{\Theta(1)}$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Dictionaries

In python:

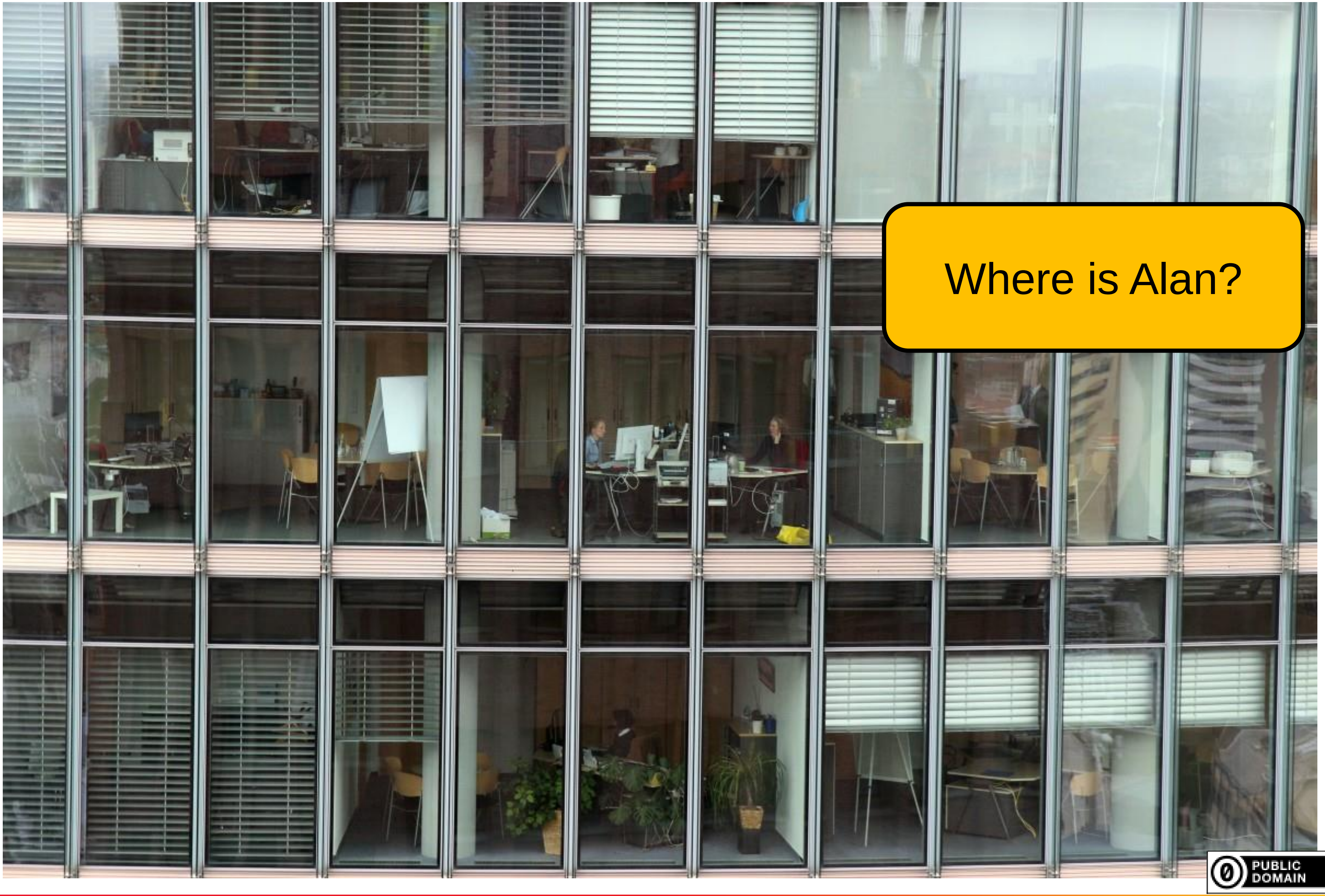
```
my_dict = {'Joe': 113, 'Pete': 7, 'Alan': '110'}
print("my_dict['Joe']: " + my_dict['Joe'])
```

gives `my_dict['Joe']: 113` as output

- the immutables **'Joe'**, **'Pete'**, and **'Alan'** are the keys
- **113**, **7**, and **110** are the values (or the stored data)

Next: Why dictionaries and how are they implemented?

Dictionaries



Where is Alan?

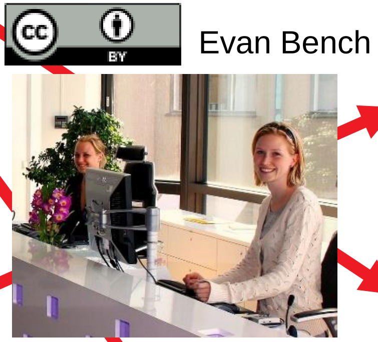
- Go through all offices one by one?

like in list and array

- No, you would ask the receptionist for the office number

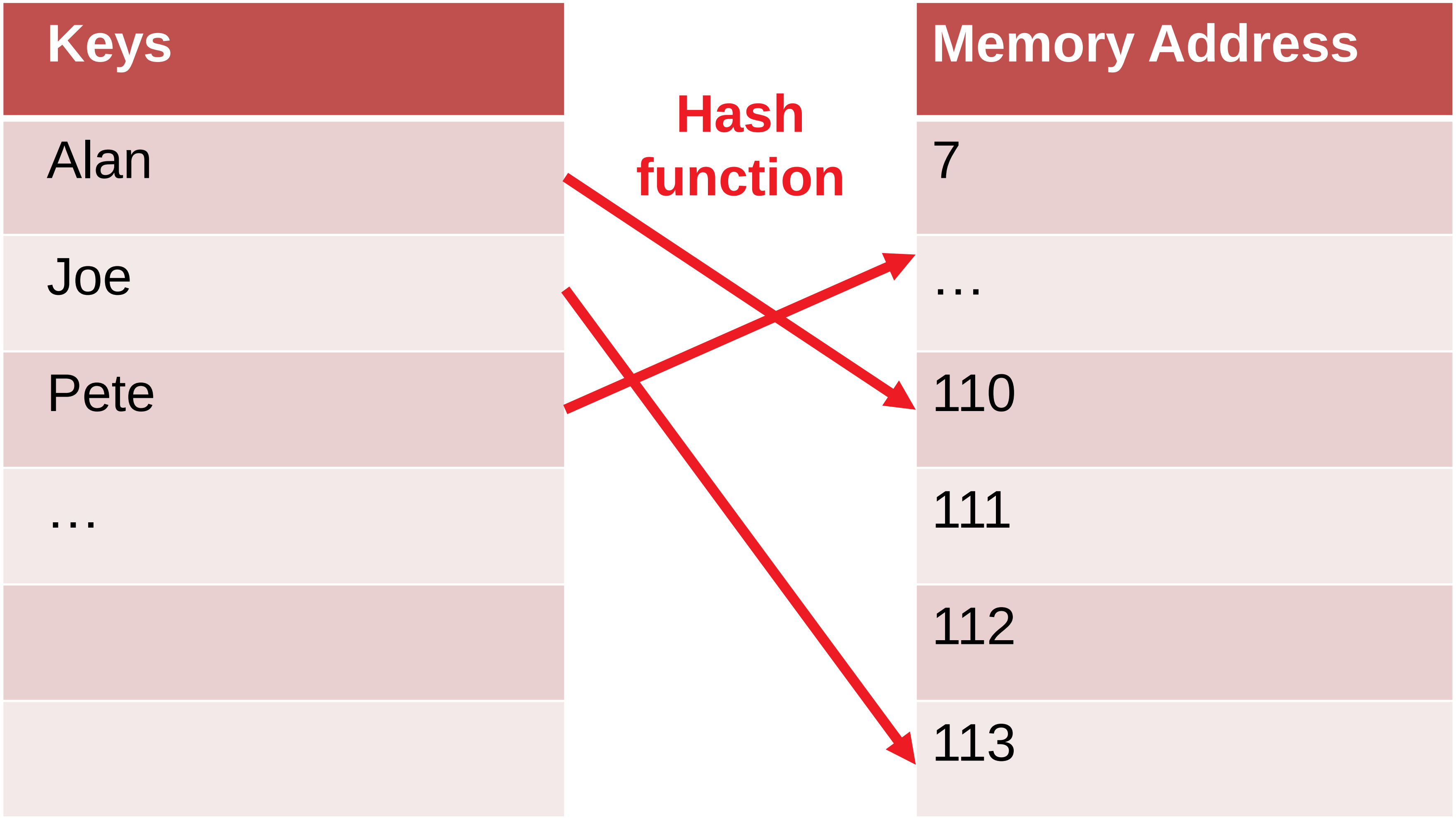
Dictionaries Implemented as Hashtables

Names
Alan
Joe
Pete
…



Offices
7
…
110
111
112
113

Dictionaries Implemented as Hashtables



Possible hash function: $h = z \textbf{ mod } n$

Hash Functions

…should be

- deterministic: find data again
- uniform: use allocated memory space well
  [more tricky with variable length keys such as strings]

Problems to address in practice:

- how to deal with collisions (e.g. via multiple hash functions)
- deleting needs to insert dummy keys when a collision appeared
- what if the hash table is full? → resizing

All this gives a constant average performance in practice

Not more details here, but if you are interested:

For more details on python's dictionary:
https://www.youtube.com/watch?v=C4Kc8xzcA68

What Have We Learned Today?

- Combinatorics: basic ways of counting things
- O-notation: how to formalize classes of asymptotic function growth
- Basic data structures and their operations
  - arrays
  - lists
  - (binary search) trees
  - dictionaries / hash tables

see also https://www.bigocheatsheet.com/

- And along the way: graph theory, DFS, and BFS