Algorithms & Complexity
Lecture 2: Data Structures

September 28, 2020

CentraleSupélec / ESSEC Business School

Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

Thu		Topic
Mon, 21.09.2020	PM	Introduction, Combinatorics, O-notation, data structures
Mon, 28.09.2020	PM	Data structures II
Mon, 5.10.2020	PM	Sorting algorithms, recursive algorithms
Mon, 12.10.2020	PM	Greedy algorithms
Mon, 19.10.2020	PM	Dynamic programming
Mon, 2.11.2020	PM	Randomized Algorithms and Blackbox Optimization
Mon, 16.11.2020	PM	Complexity theory I
Mon, 23.11.2020	PM	Complexity theory II
Mon, 14.12.2019	PM	Exam

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

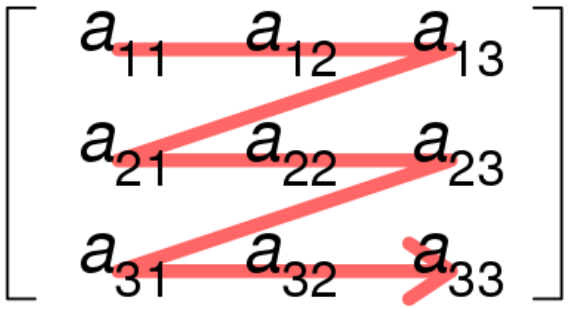
Runtimes

- 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
  - $k = \Theta(\log(n))$

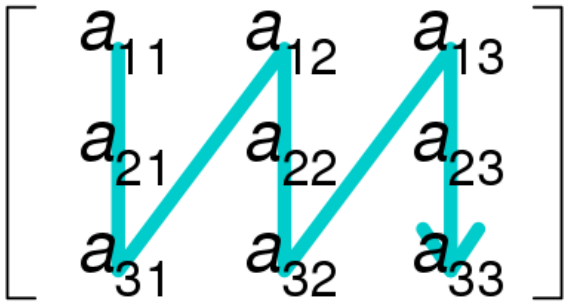
Remarks: Arrays and Matrices

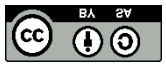
- Matrices can be stored in arrays, too
- Row first or column first?
- Storing sparse matrices efficiently: not covered here

Row-major order



Column-major order



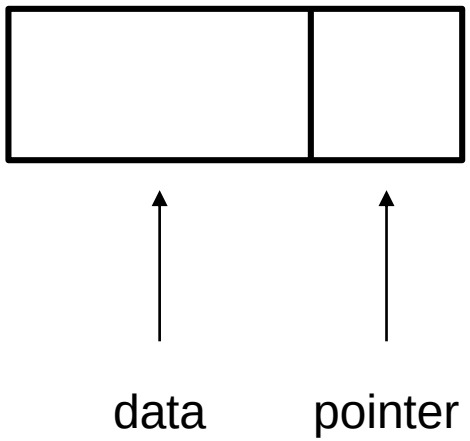


Cmglee

Linked 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

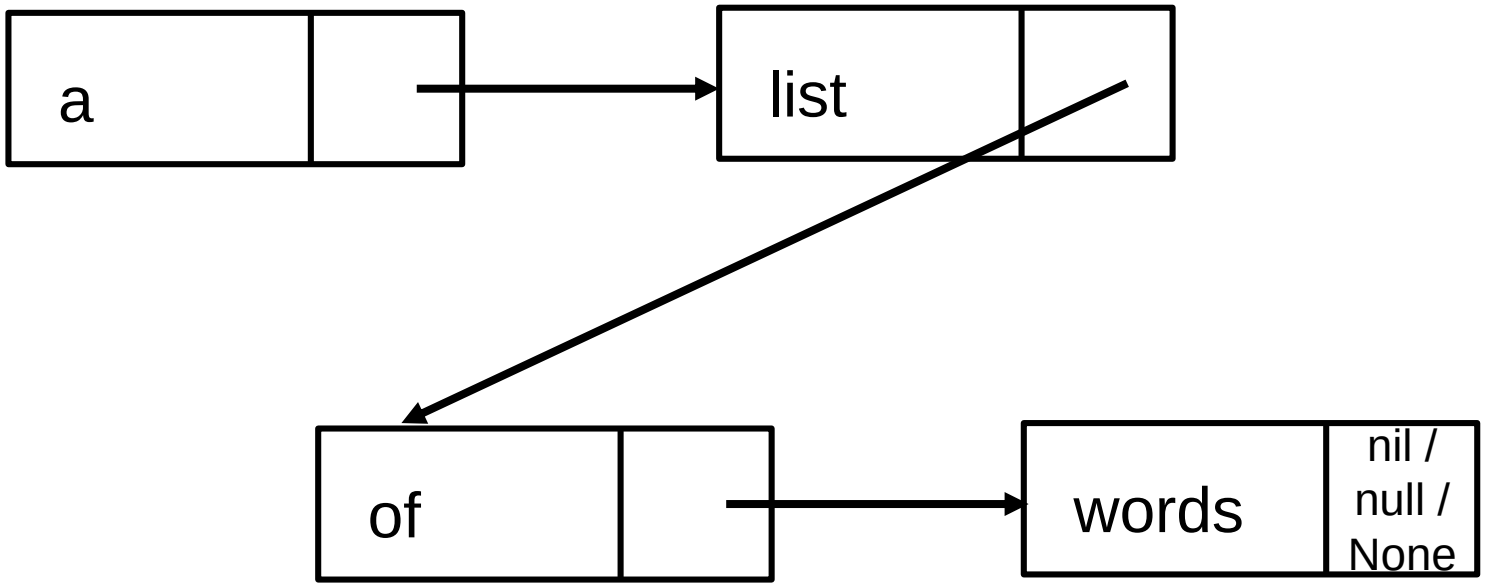
Idea of a Linear List



Linked 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



Linked 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	…

Linked 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	…

Linked 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

Linked Lists

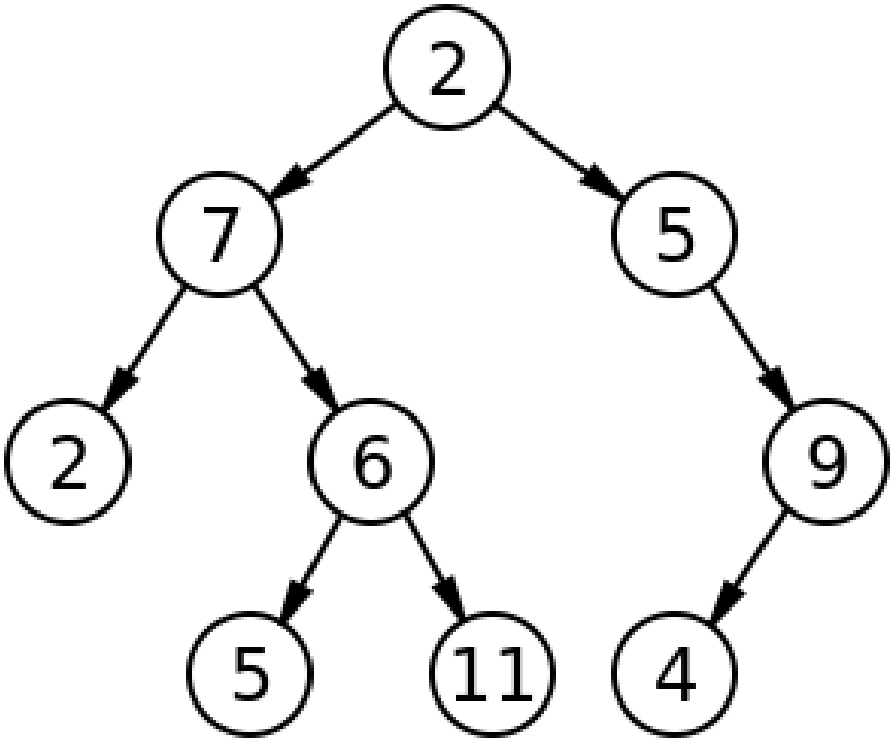
- removal of element in constant time $\mathcal{O}(1)$
- very similar for adding: $\mathcal{O}(1)$
- adding into a sorted list: $\mathcal{O}(n)$
- but now searching is more difficult, even if sorted
  - reason: we don't have access to the "middle" element
  - search for element i: $\Theta(i)$ if list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

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

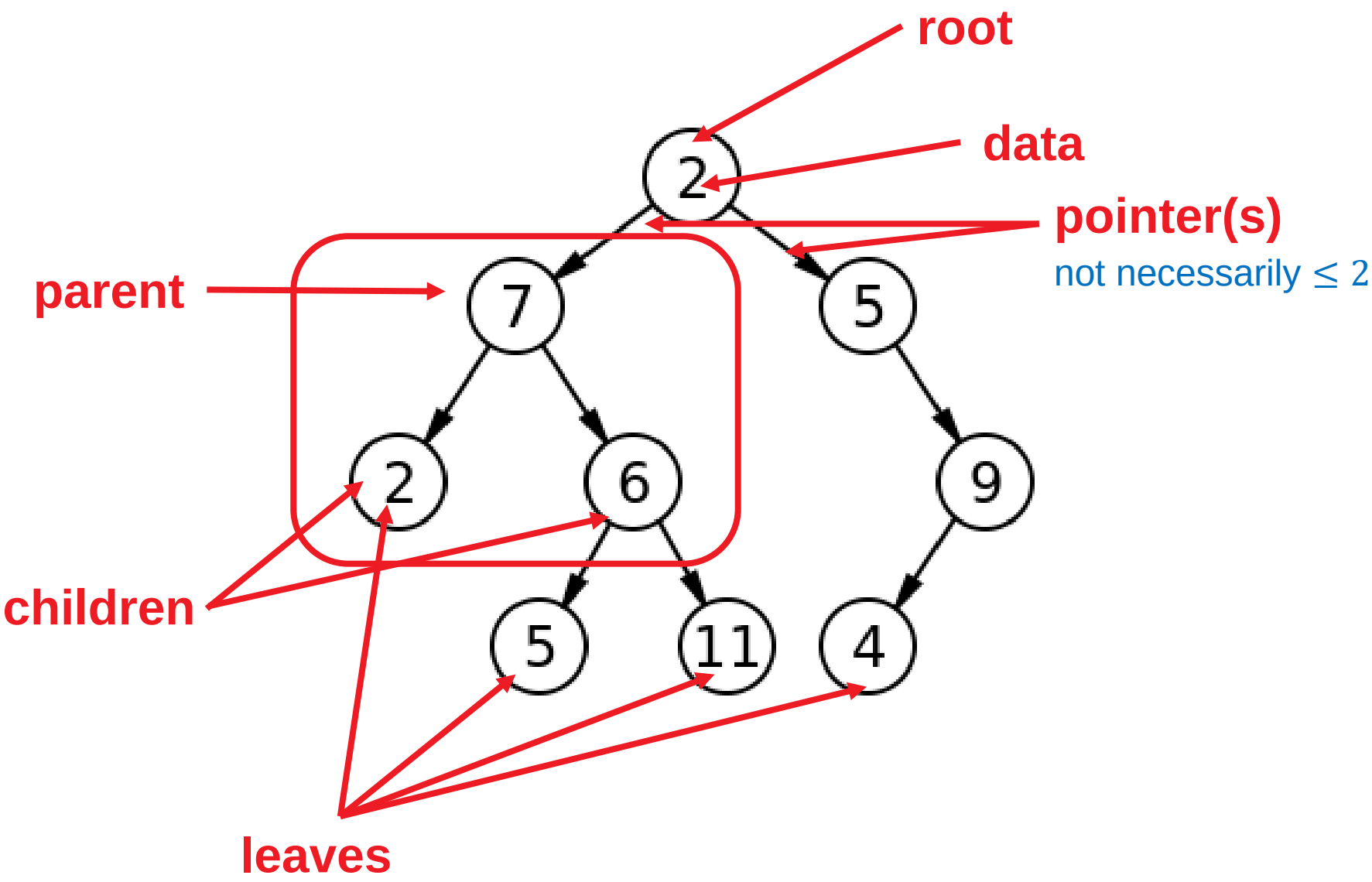
Trees

Brian Green

Trees



Trees



Trees are Special Graphs

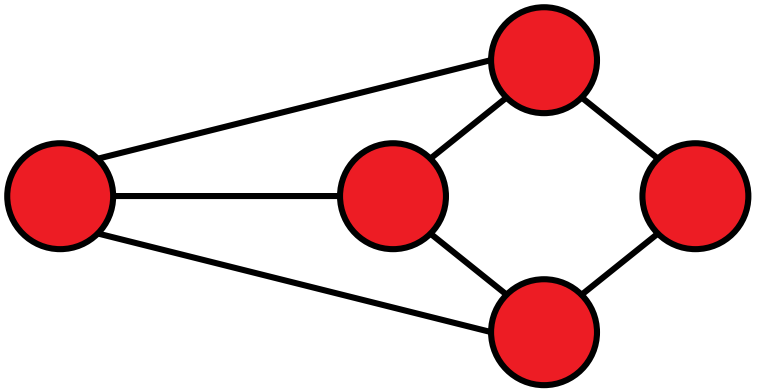
For a more formal definition, we need to introduce the concept of graphs…

Basic Concepts of Graph Theory

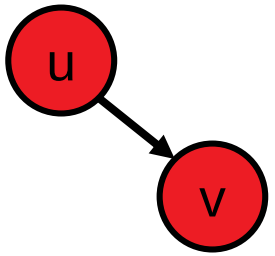
[following for example http://math.tut.fi/~ruohonen/GT_English.pdf]

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., $u, v \in V$).*

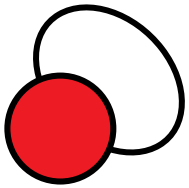


- vertices = nodes
- edges = lines
- Note: edges cover two *unordered* vertices (*undirected* graph)
  - if they are *ordered*, we call G a *directed* graph with edges $e = (u, v)$
  - to draw an ordered graph, we use arrows

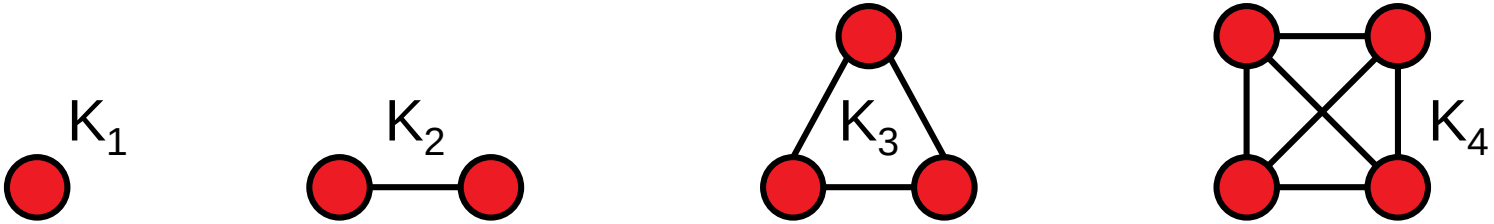


Graphs: Basic Definitions

- 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):



a loop

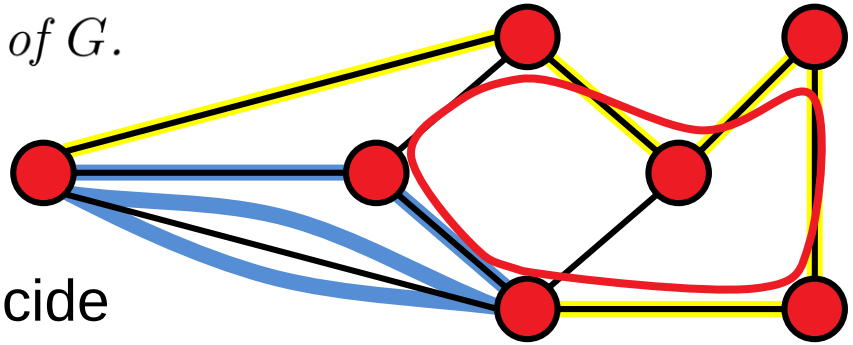


Walks, Paths, and Circuits

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

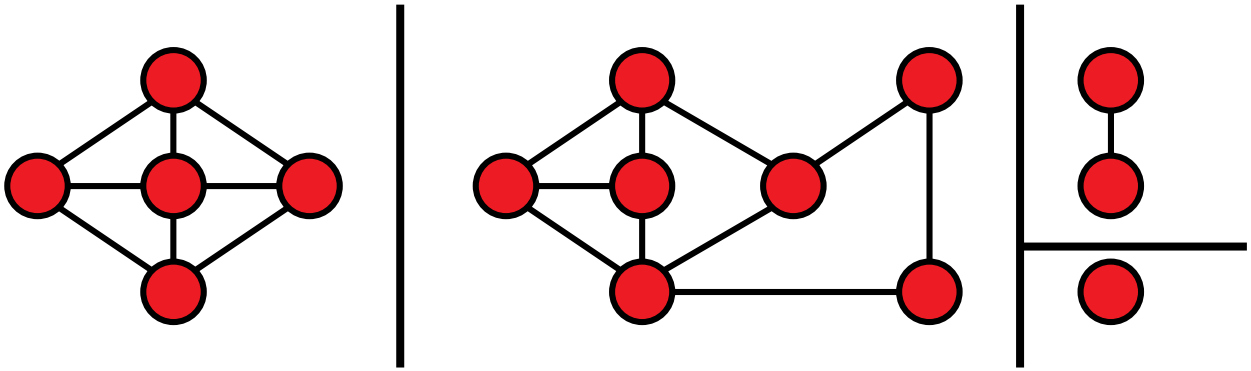
- *closed* if first and last node coincide
- a *path* if each vertex is visited at most once

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

Graphs: Connectedness

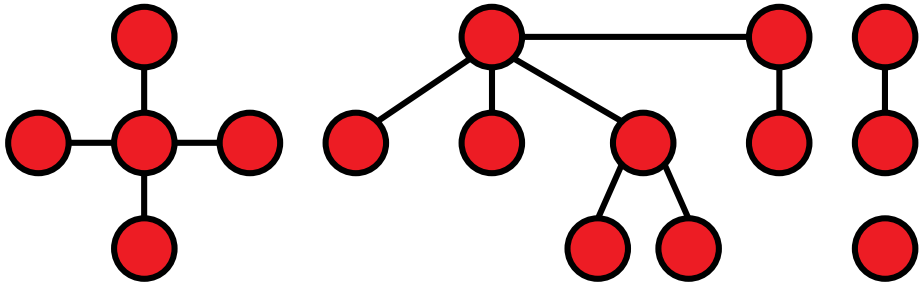
- Two vertices are called *connected* if there is a walk between them in G
- If all vertex pairs in G are connected, G is called connected

- The *connected components* of G are the (maximal) subgraphs which are connected.

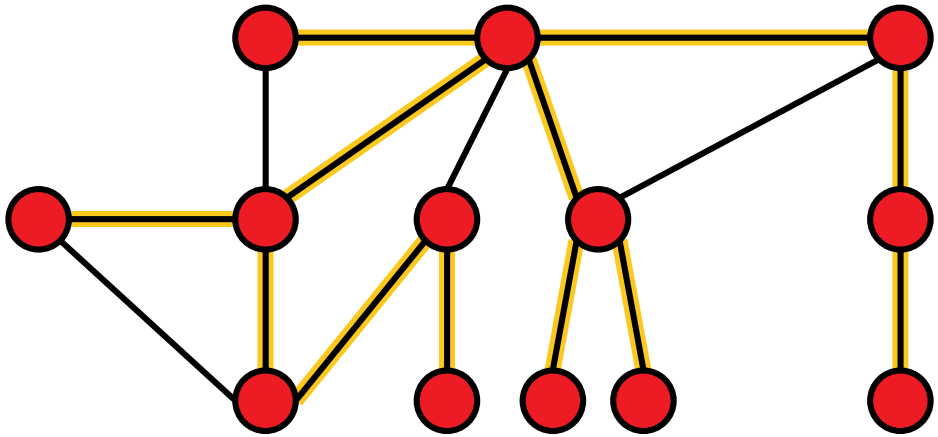


Trees and Forests

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

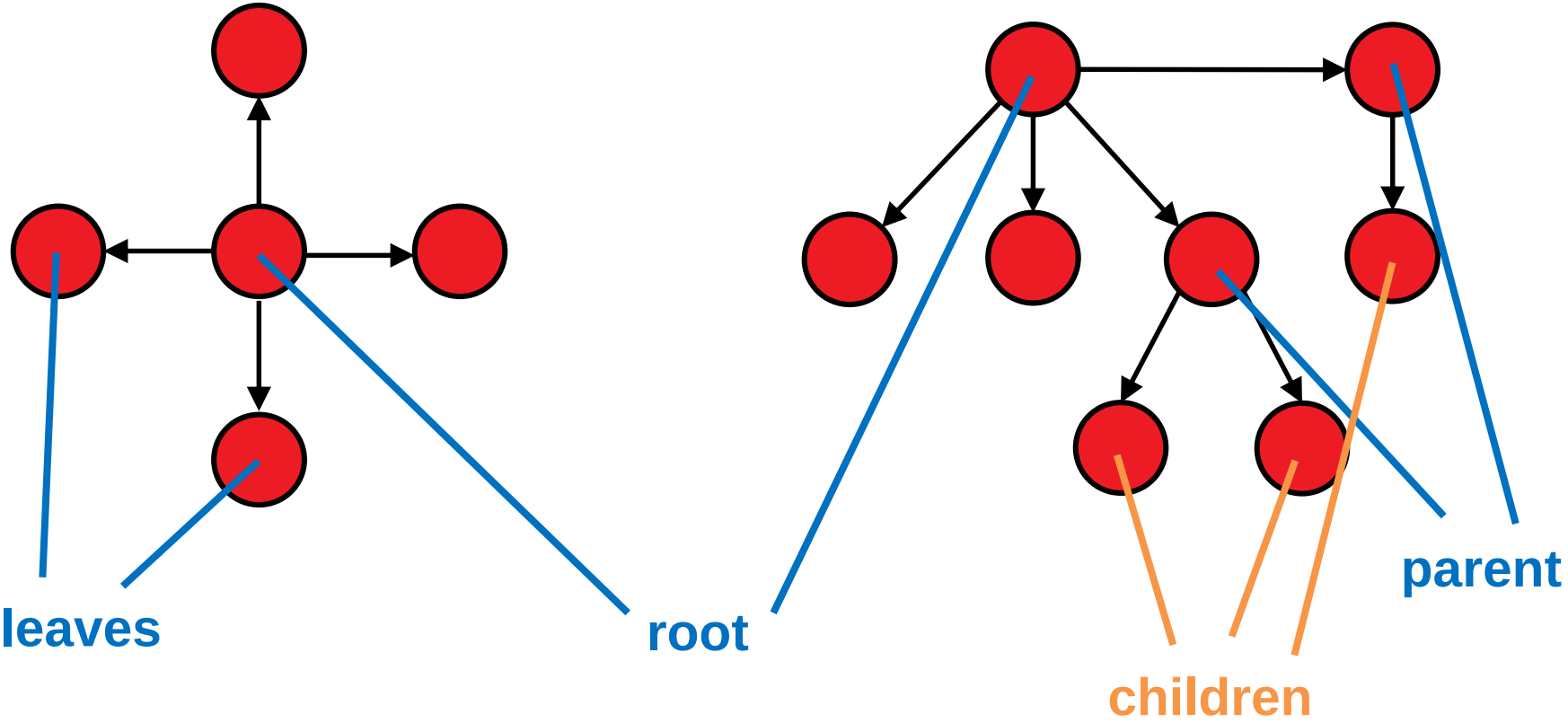


A *spanning tree* of a connected graph G is a tree in G which contains all vertices of G



Special Notations for Trees With Added Directions

Assume: Tree-like graph with directed edges s.t. each vertex is connected to a specific vertex, the root



Note:

choice of root/parent/children not always unique in undirected trees

Depth-First Search (DFS)

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

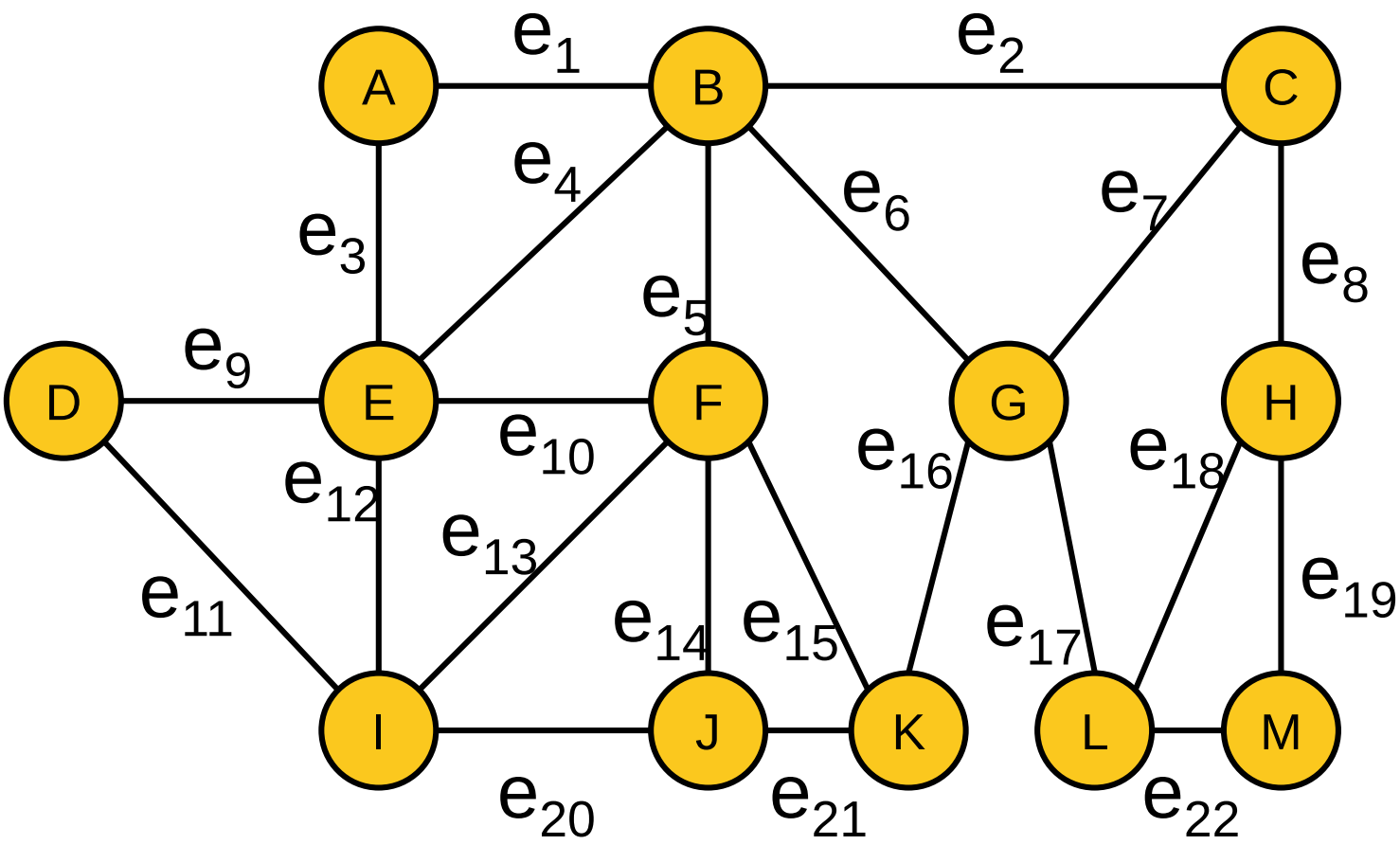
Depth-first search and breadth-first search are two algorithms to do so

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

1. start at any node x; set i=0
2. 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, label y as the node visited at iteration i, and continue the search at x=y in step 2
   - else continue with next unvisited edge of x
3. if all edges {x,y} are visited, we continue with x=parent(x) at step 2 or stop if x equals the starting node v0

DFS: Stage Exercise

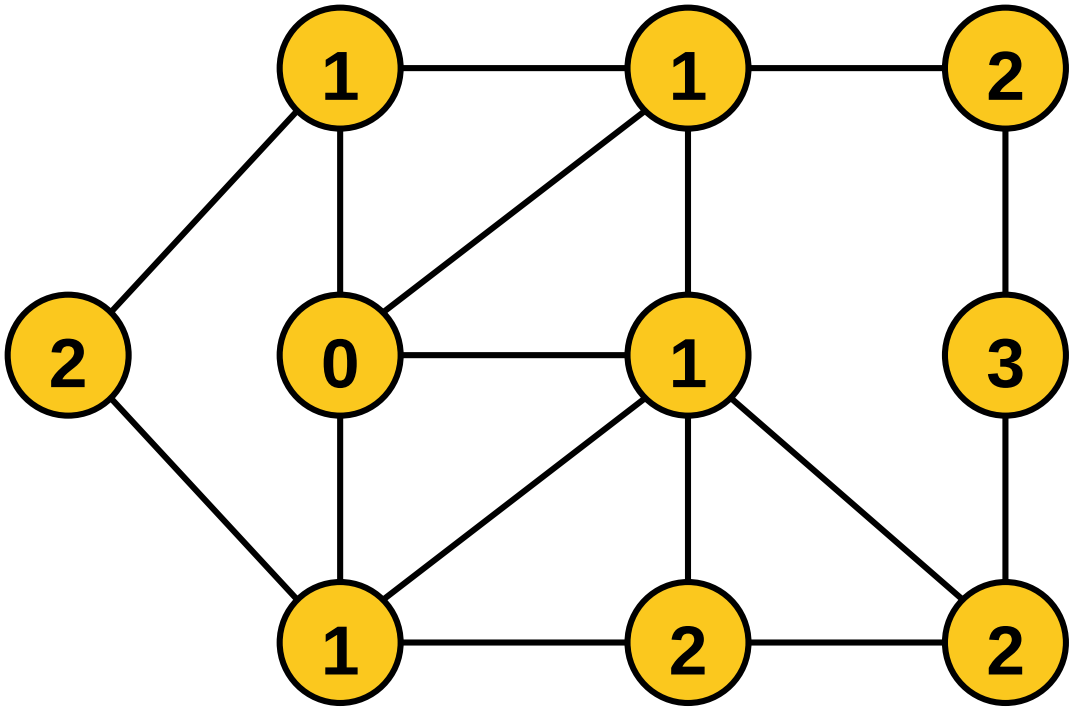
Exercise the DFS algorithm on the following graph!



Breadth-First Search (BFS)

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

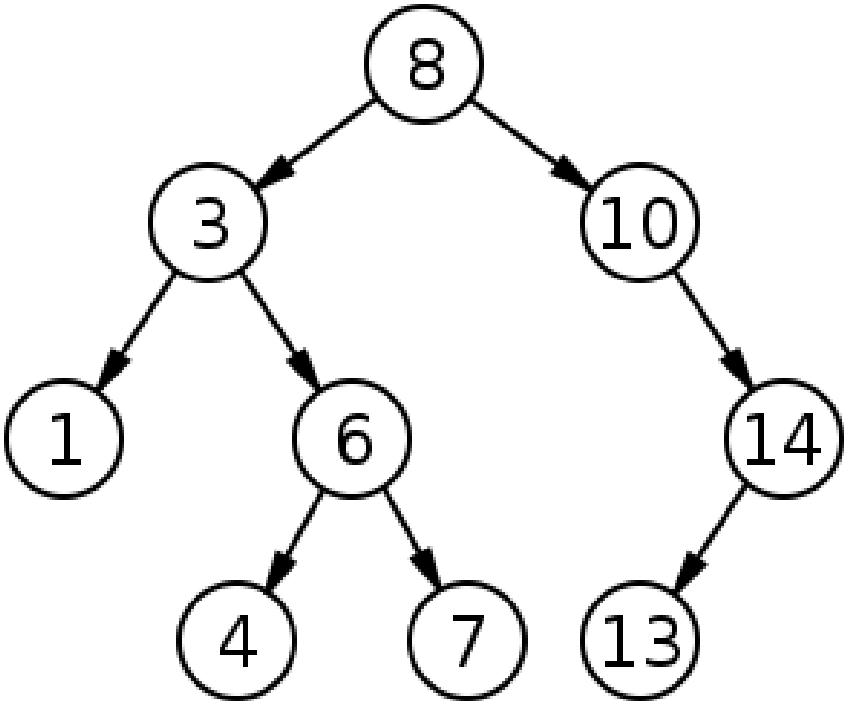
1. start at any node x, set i=0, and label x with value i
2. as long as there are unvisited edges {x,y} which are adjacent to a vertex x that is labeled with value i:
   - label all unlabeled vertices y with value i+1
3. set i=i+1 and go to step 2



Back to Trees as Data Structure

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:

Each online student: give an integer to be filled into tree

Round 2:

In class: tell where the next integer inserts

Binary Search Tree: Complexities

Search

- similar to binary search in array (go left or right until found)
- $\mathcal{O}(\log(n))$ 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 $\mathcal{O}(1)$ [we are always at a leaf or have an “empty child”]

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: $\mathcal{O}(\text{tree depth})$, in worst case: $\Theta(n)$

Binary Trees: Can We Do Better?

Binary Search Tree

average case (random inserts) | worst case

search	insert	delete	Search	Insert	Delete
$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Guarantee a balanced tree:

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

average case (random inserts) | worst case

search	insert	delete	search	insert	delete
$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$

Can We Do Even Better on Average?

Balanced Trees

average case (random inserts) | | | worst case | | |

search	insert	delete	search	insert	delete
$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$	$\mathcal{O}(\log(n))$

?

average case (random inserts) | | | worst case | | |

search	insert	delete	search	insert	delete
$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

Dictionaries

In python:

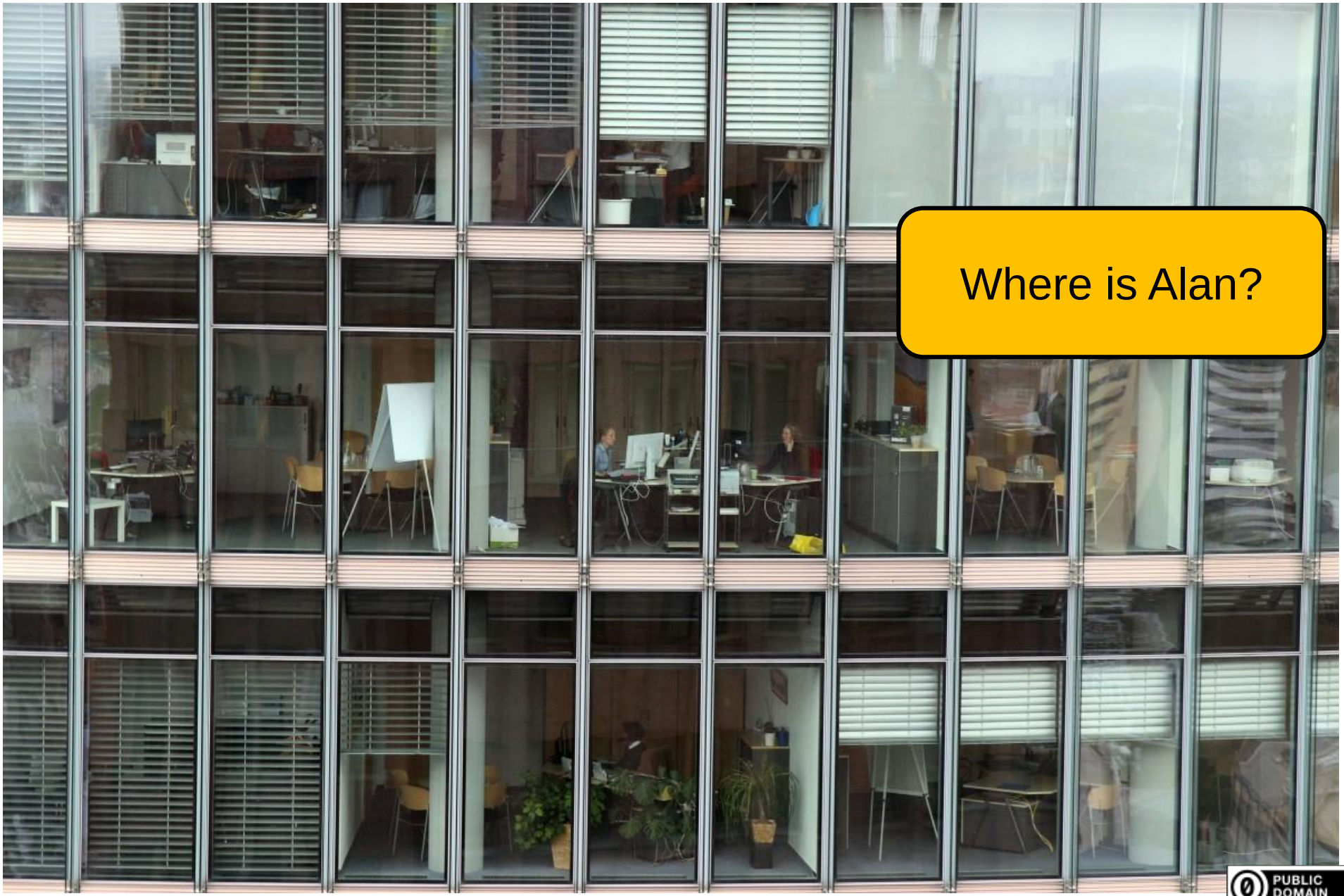
```
my_dict = {'Joe': 113, 'Pete': 7, 'Alan': '110'}
print("my_dict['Joe']: " + my_dict['Joe'])
```

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

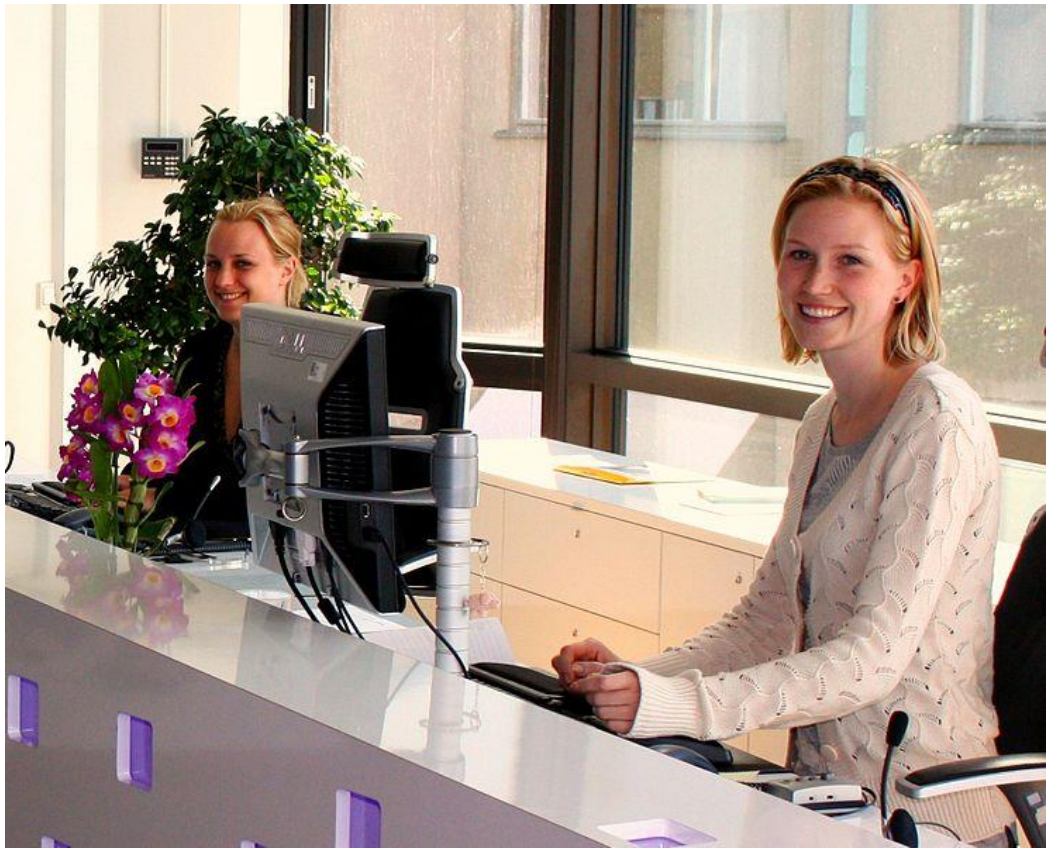


Where is Alan?

- Go through all offices one by one?

like in list and array

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



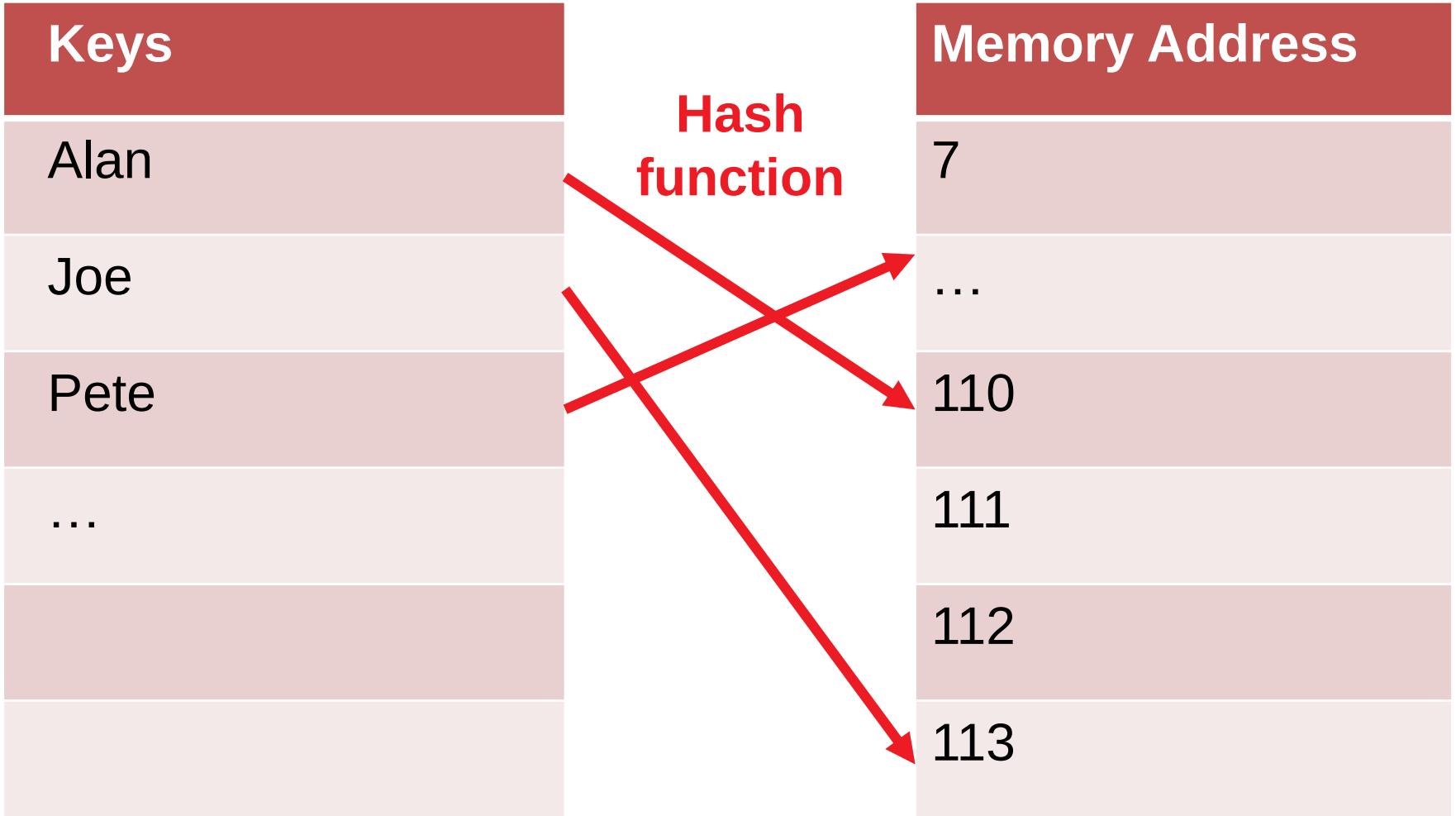
Evan Bench

Dictionaries Implemented as Hashtables

Names	Offices
Alan	7
Joe	…
Pete	110
…	111
	112
	113

Evan Bench

Dictionaries Implemented as Hashtables



Possible hash function: $h = z$ **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
and a worst case of $\Theta(n)$ for insert/remove/search

Not more details here, but if you are interested:

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

Quick Recap Data Structures

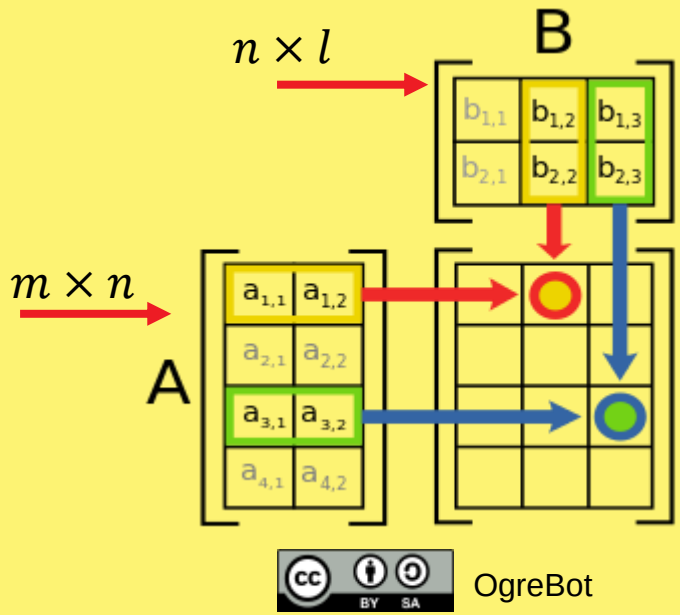
- Arrays: fast access, slow search, no insert
- (Linked) Lists: slow access, slow search, but insert/remove in constant time
  - Hence python lists are implemented as dynamic arrays (once array is full, a larger chunk of memory gets allocated) http://www.laurentluce.com/posts/python-list-implementation/
- Trees: log(n) access, log(n) add/remove
- Dictionaries: constant average performance in practice and a linear worst case for insert/remove/search

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

discussion home exercises

Discussion Home Exercise

Exercise 1: Matrix Multiplication



- $c_{ij} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$
- naïve implementation:

$$\textbf{for } i = 1 \textbf{ to } m \textbf{ do}:$$
$$\quad \textbf{for } j = 1 \textbf{ to } l \textbf{ do}:$$
$$\quad\quad c_{ij} = 0$$
$$\quad\quad \textbf{for } k = 1 \textbf{ to } n \textbf{ do}:$$
$$\quad\quad\quad c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$$

- computation per cell: n additions and n multiplications
- has to be done for all $m \times l$ cells
- in total: $m \cdot l \cdot n$ additions and $m \cdot l \cdot n$ multiplications
- $\Theta(n^3)$ if $k = l = n$
- interesting: we can do better:
  $\mathcal{O}(n^{\log 7}) = \mathcal{O}(n^{2.807\ldots})$ by Strassen (1968)
  even $\mathcal{O}(n^{2.3728639})$ by Le Gall (2014)

Discussion Home Exercise

Exercise 2: Finding Smallest Element

simple implementation:

assume array to be $\boldsymbol{A} = [\boldsymbol{A}[\mathbf{1}], \ldots, \boldsymbol{A}[\boldsymbol{n}]]$

$\mathbf{m} = \mathbf{A}[\mathbf{1}]$
for $i = 2$ **to** n **do**:
  if $\boldsymbol{A}[\boldsymbol{i}] < \boldsymbol{m}$ **do**:
    $\mathbf{m} = \mathbf{A}[\mathbf{i}]$

Worst case: smallest element is at $\boldsymbol{A}[\boldsymbol{n}]$

- Runtime $\mathcal{O}(n)$

Best case (?): smallest element is at $\boldsymbol{A}[\mathbf{1}]$

- But Algorithm does not know this before to see all elements!
- Also runtime of $\mathcal{O}(n)$

Discussion Home Exercise

Exercise 3: Finding Smallest Element II
(finding also 2nd and 3rd smallest)

Idea: always keep the two/three smallest *already seen* solutions

$\mathbf{s, m, l} = +\infty, +\infty, +\infty$ **# s: smallest, m: 2nd smallest, l: 3rd smallest**
for $i = 1$ **to** n **do**:
    if $\boldsymbol{A[i]} < \boldsymbol{s}$ **do**:
        $\mathbf{s, m, l} = \mathbf{A[i], s, m}$
    else if $\mathbf{A[i]} < \mathbf{m}$:
        $\mathbf{s, m, l} = \mathbf{s, A[i], m}$
    else if $\mathbf{A[i]} < \mathbf{l}$:
        $\mathbf{s, m, l} = \mathbf{s, m, A[i]}$

Worst & best case again (asymptotically) the same: $\mathcal{O}(n)$

Discussion Home Exercise

Exercise 4: $\mathcal{O}$-Notation

$$\mathcal{O}(f_1) \neq \mathcal{O}(\log(f_1))$$

Proof:

- choose for example $g_1(x) = 2^x$ and $f_1(x) = 2^x$. Then, by definition, $g_1 = \mathcal{O}(f_1)$
- But $g_1 \notin \mathcal{O}(\log(f_1))$:
  - $|g_1(x)| = 2^x > x = \log(2^x) = \log(f_1(x))$ for all $x > 1$
  - Thus, there exists no constant $c > 0$ such that the left-hand side can be smaller than c times the right-hand side for *all* x above a certain threshold $x_0 > 0$

Discussion Home Exercise

Exercise 4: $\mathcal{O}$-Notation

$$\mathcal{O}(f_1) \cdot \mathcal{O}(f_2) = \mathcal{O}(f_1 \cdot f_2)$$

Proof:

- choose $g_1 \in \mathcal{O}(f_1)$ and $g_2 \in \mathcal{O}(f_2)$ arbitrarily
- i.e. we have constants $n_1, n_2, c_1, c_2 > 0$ such that
  $g_1(n) \leq c_1 \cdot f_1(n)$ for all $n > n_1$ and
  $g_2(n) \leq c_2 \cdot f_2(n)$ for all $n > n_2$
- but with $c' = c_1 \cdot c_2$ then also

$$|g_1(n) \cdot g_2(n)| \leq |g_1(n)| \cdot |g_2(n)|$$
$$\leq c_1 \cdot f_1(n) \cdot c_2 \cdot f_2(n) = c' \cdot (f_1(n) \cdot f_2(n))$$

  for all $n > \max(n_1, n_2)$