Algorithms & Complexity
Lecture 2: Data Structures

September 20, 2021

CentraleSupélec / ESSEC Business School

Dimo Brockhoff

Inria Saclay – Ile-de-France

Course Overview

Thu		Topic
Mon, 13.09.2021	AM	Introduction, Combinatorics, O-notation, data structures
Mon, 20.09.2021	AM	Data structures II, Sorting algorithms I 1st exercise out
Mon, 27.09.2021	AM	Sorting algorithms II, recursive algorithms
Mon, 04.10.2021	-	Deadline 1st exercise
Mon, 11.10.2021	AM	Greedy algorithms 2nd exercise out
Mon, 18.10.2021	AM	Dynamic programming
Mon, 25.10.2021	AM	Randomized Algorithms and Blackbox Optimization 3rd exercise out Deadline 2nd exercise
Mon, 01.11.2021	-	-
Mon, 08.11.2021	AM	Complexity theory I Deadline 3rd exercise
Mon, 15.11.2021	AM	Complexity theory II
Mon, 13.12.2021	PM	Exam

Recap: 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!]

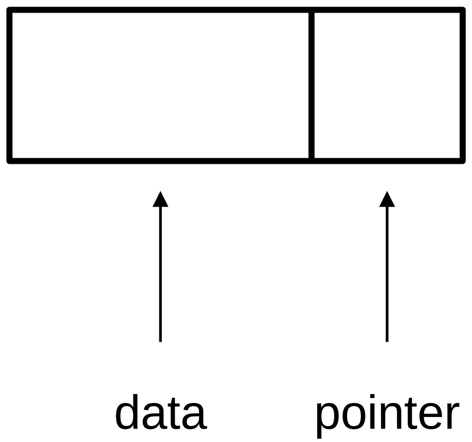
Common operations and their complexity:

- 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

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

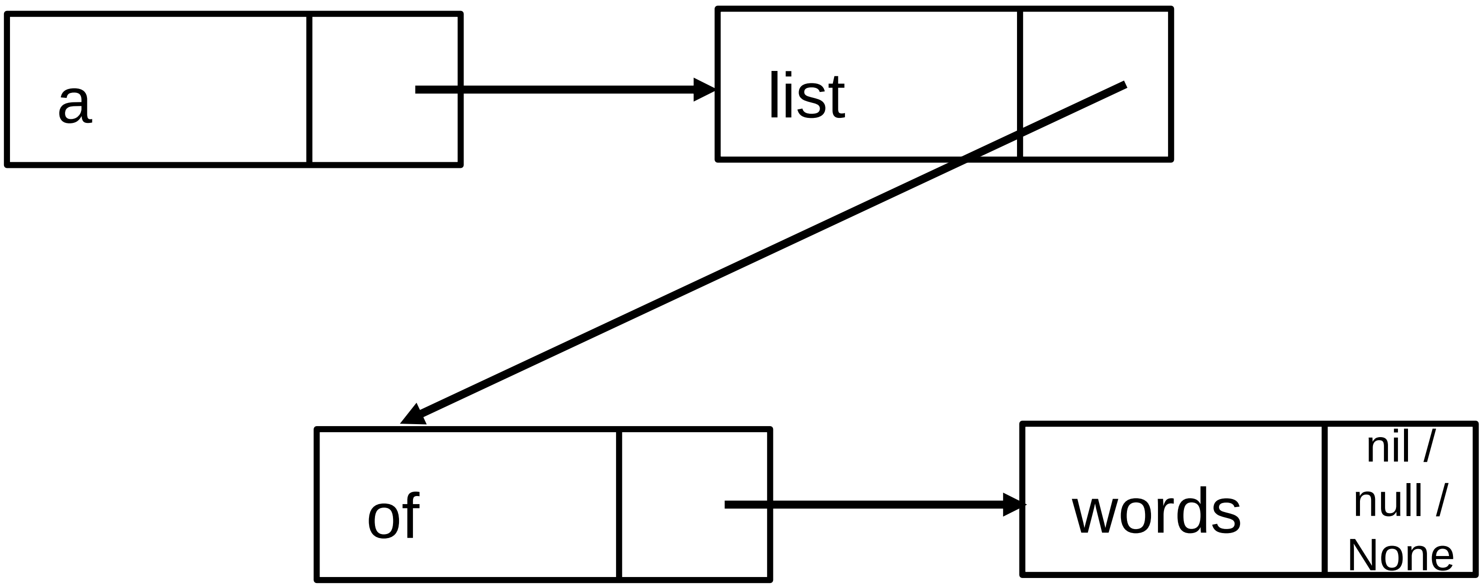
Idea of a Linear List



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

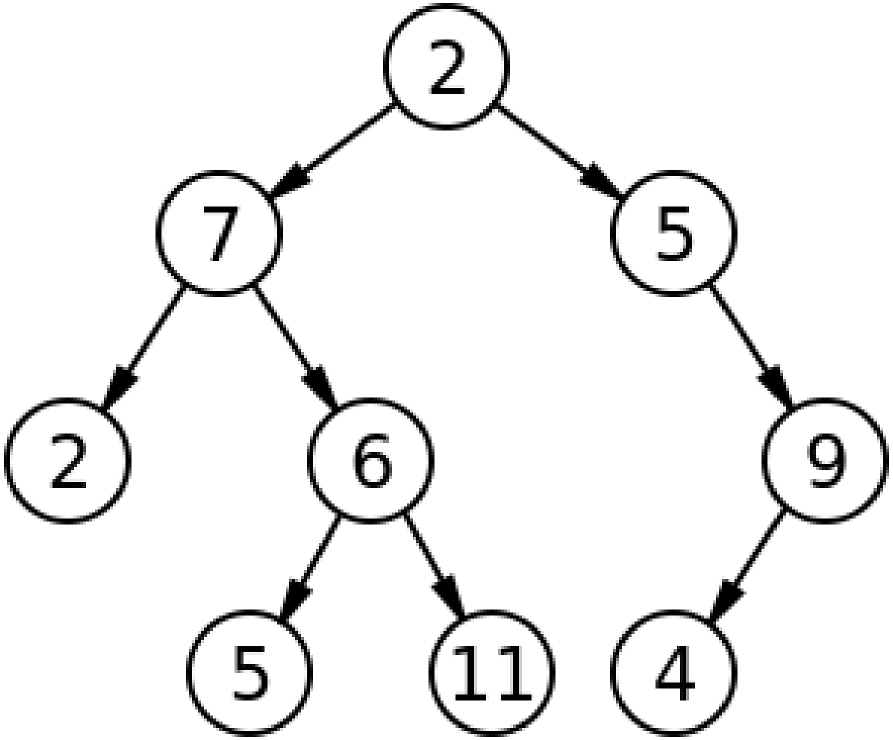
- removal of element in constant time $\mathcal{O}(1)$
- very similar for adding: $\mathcal{O}(1)$
- adding (in order) into a sorted list: $\mathcal{O}(n)$
- but now searching is more difficult, even if sorted
  - reason: we don’t have access to the “middle” element
  - search for element i: $\Theta(i)$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

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

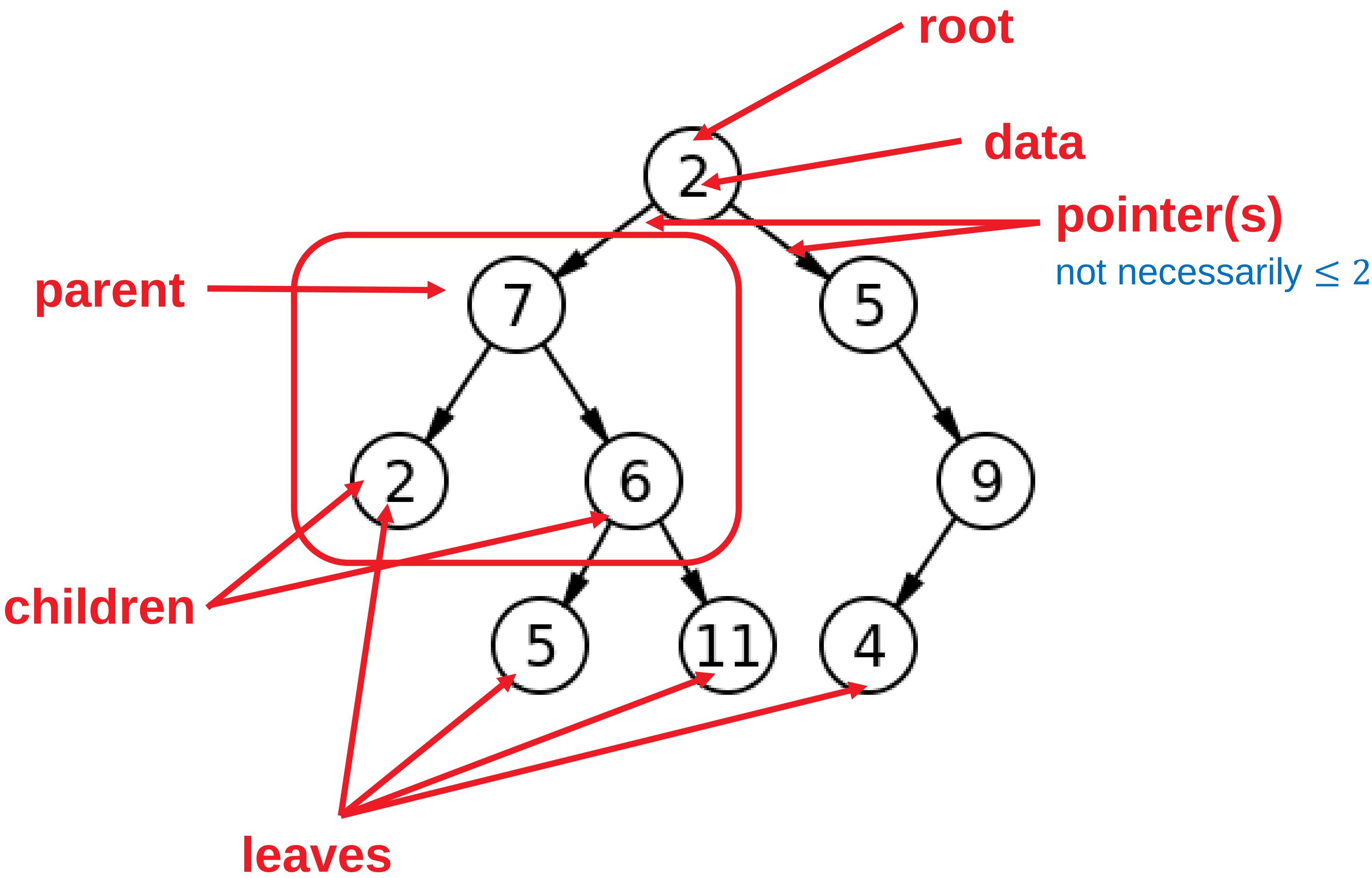
Trees

Brian Green

Trees



Trees


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

Trees are Special Graphs

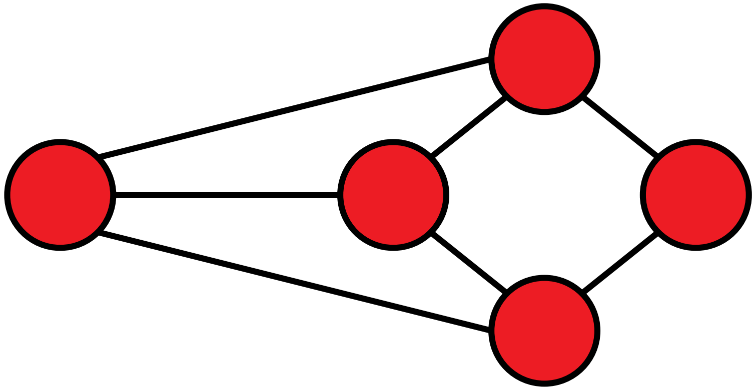
For a more formal definition, we need to introduce the concept of graphs…

Basic Concepts of Graph Theory

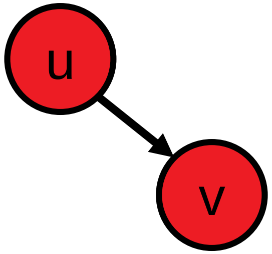
[following for example http://math.tut.fi/~ruohonen/GT_English.pdf]

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

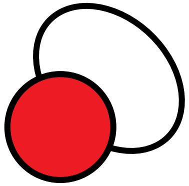


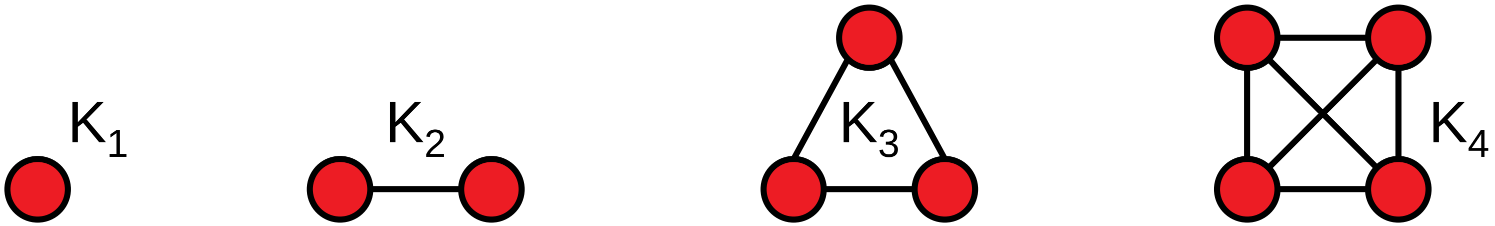
- vertices = nodes
- edges = lines
- Note: edges cover two *unordered* vertices (*undirected* graph)
  - if they are *ordered*, we call G a *directed* graph with edges $e = (u, v)$
  - to draw an ordered graph, we use arrows



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

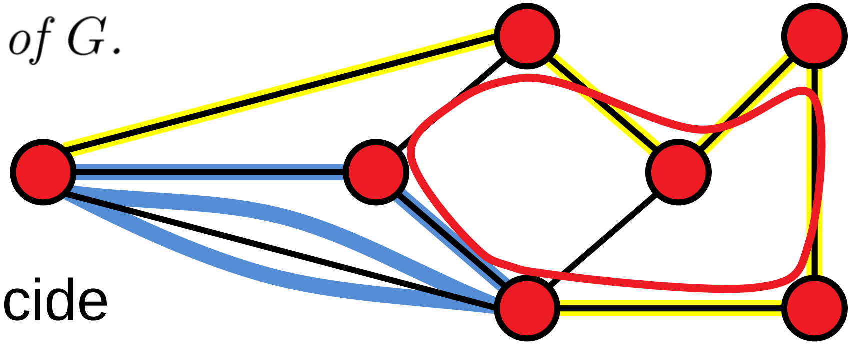


Walks, Paths, and Circuits

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

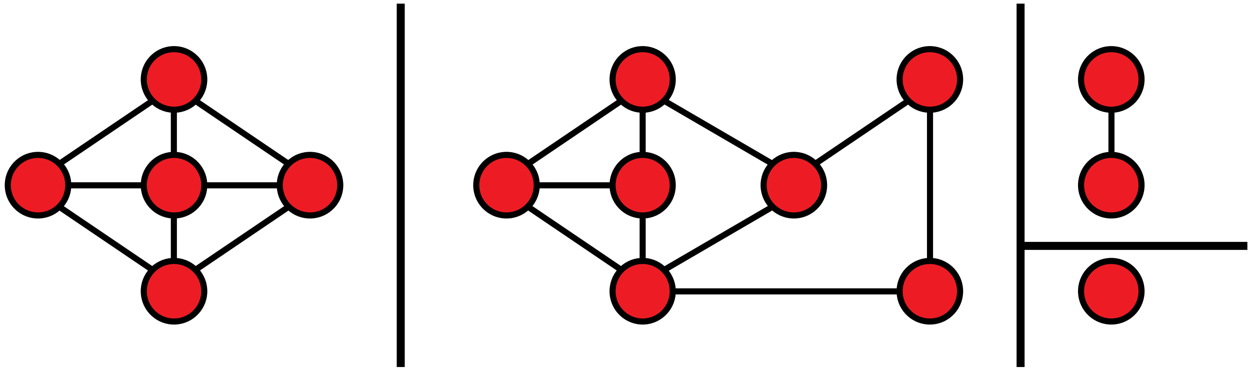
- *closed* if first and last node coincide
- a *path* if each vertex is visited at most once (except for first/last nodes)

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

Graphs: Connectedness

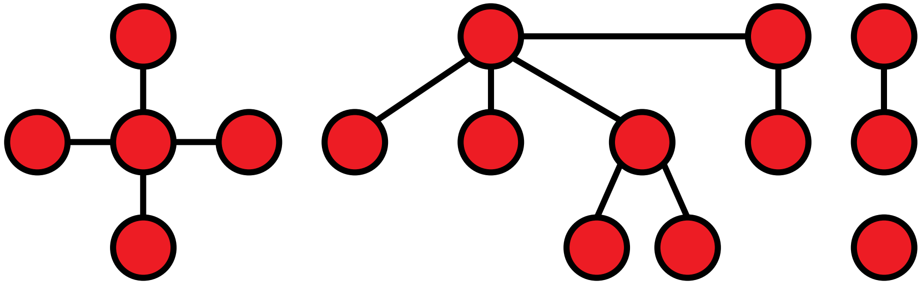
- Two vertices are called *connected* if there is a walk between them in G
- If all vertex pairs in G are connected, G is called connected

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



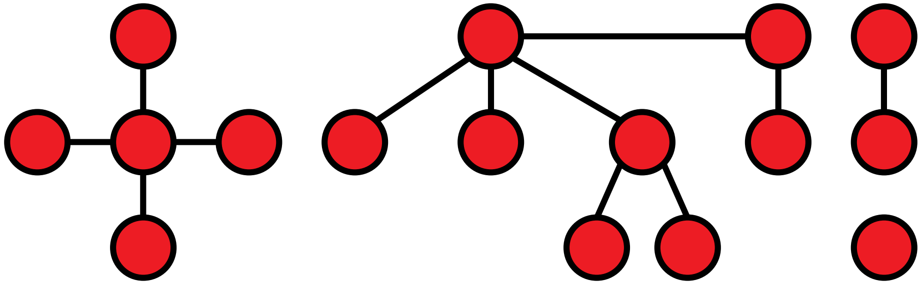
Trees and Forests

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

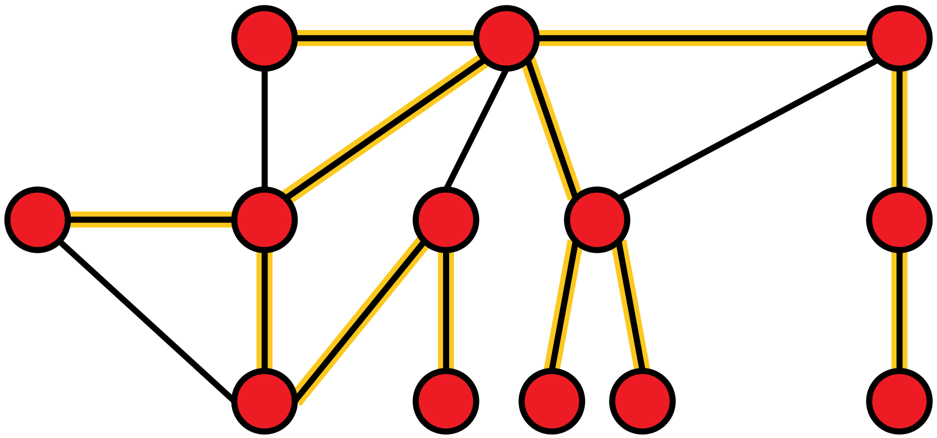


Trees and Forests

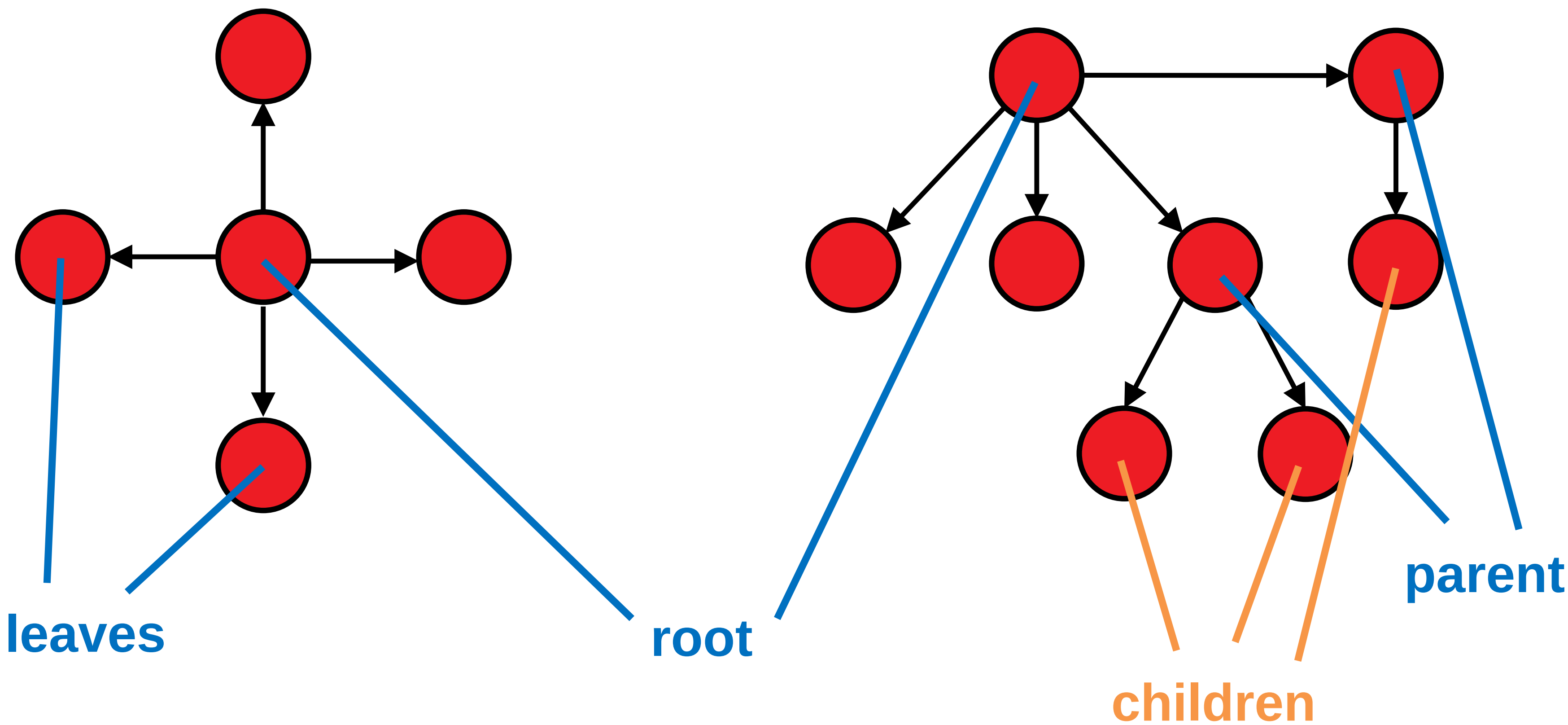
- A *forest* is a cycle-free graph
- A *tree* is a connected forest



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



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)

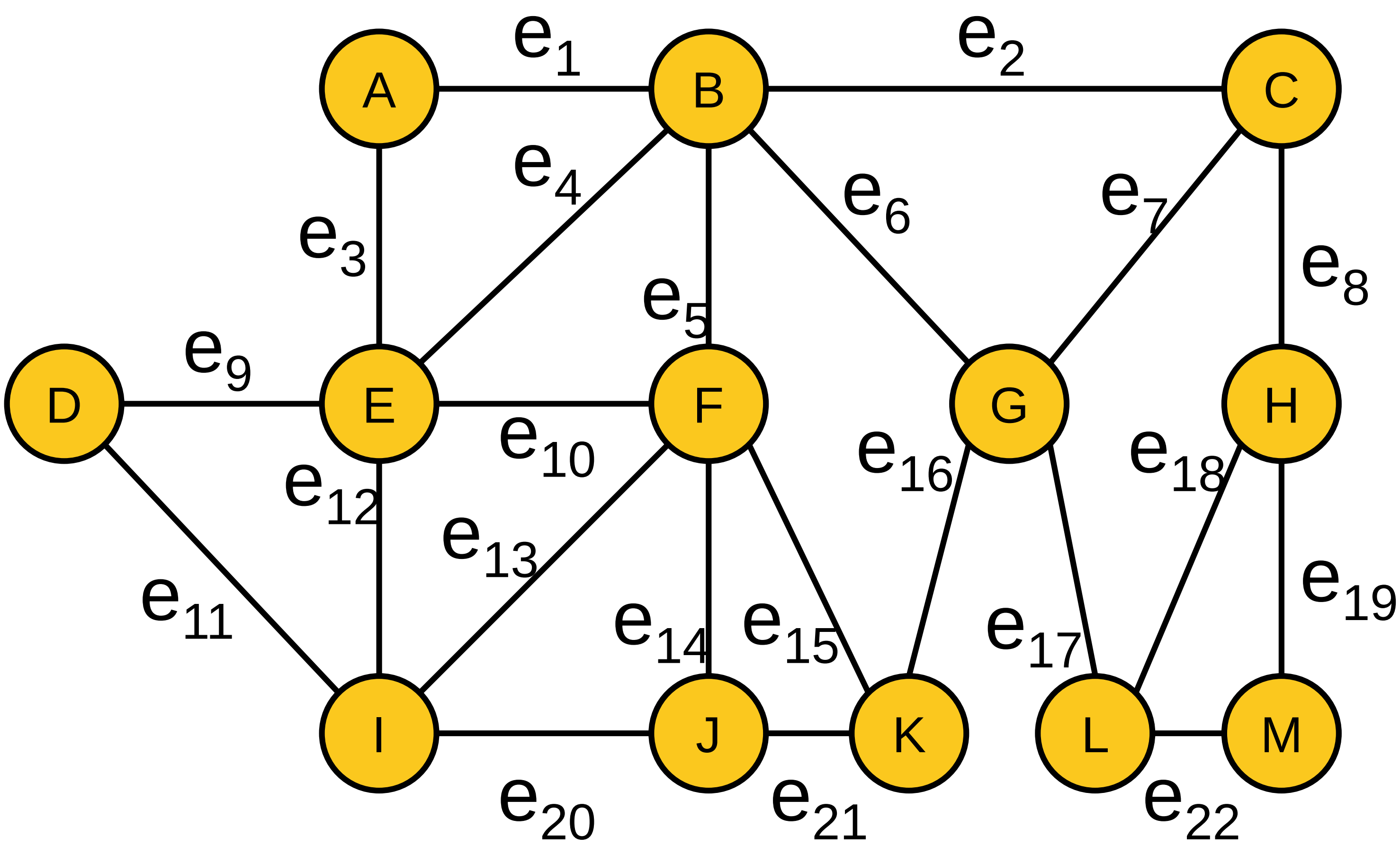
❶ start at any node x; set i=0

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, 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

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

Exercise the DFS algorithm on the following graph!



Breadth-First Search (BFS)

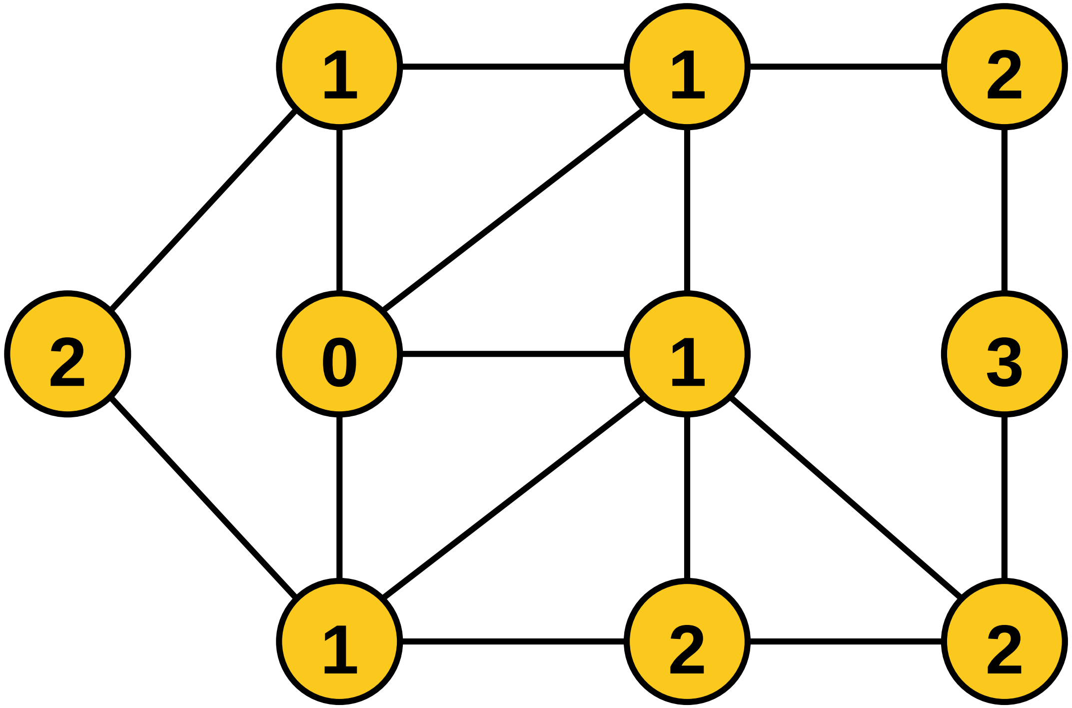
Breadth-first Search (for undirected/acyclic and connected graphs)

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:

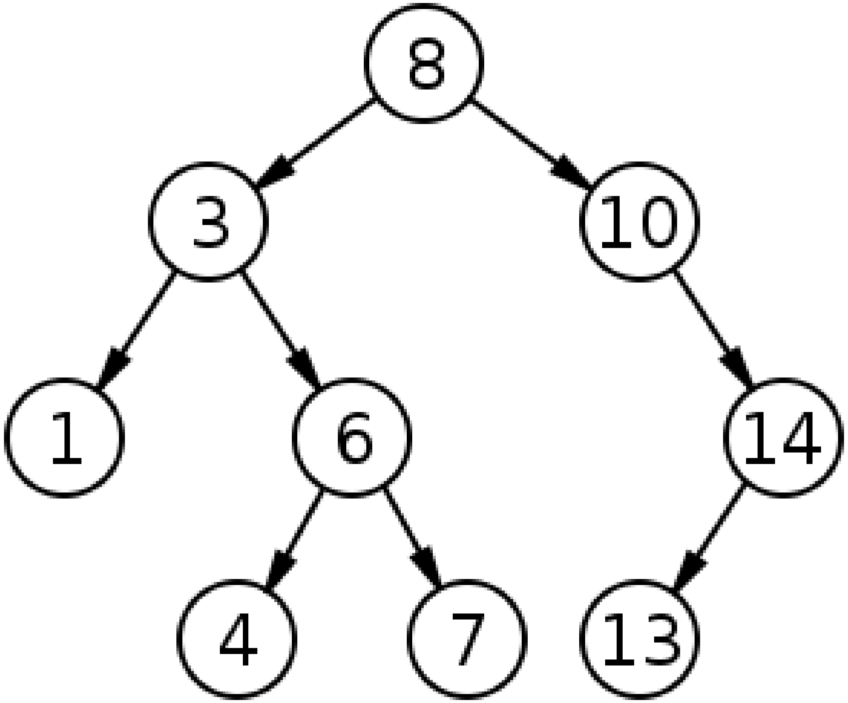
- label all unlabeled vertices y with value i+1

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:

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 (wc: $\Theta(n)$)
- 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 smallest 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: $\Theta(\mathrm{n})$

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
$\boldsymbol{\Theta(1)}$	$\boldsymbol{\Theta(1)}$	$\boldsymbol{\Theta(1)}$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

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

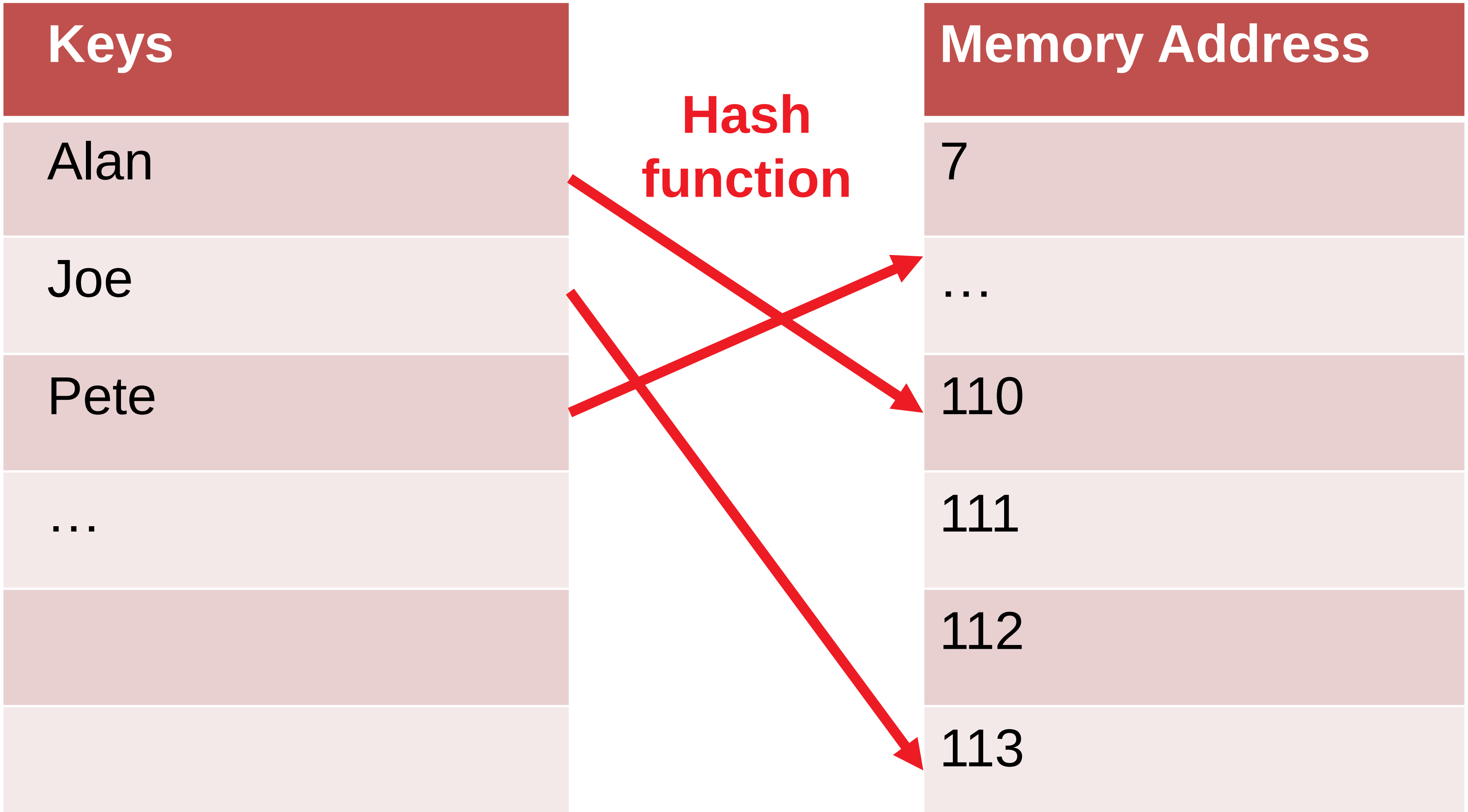
- No, you would ask the receptionist for the office number

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
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/


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, …