Friedrich-Alexander-Universität
School of Business,
Economics and Society

Week 8 – Graphs and Trees

Introduction to Computer Science | WS 22/23

Chair of Digital Industrial Service Systems | Prof. Dr. Martin Matzner & Prof. Dr. Andreas Harth

After this lecture, students understand the theory of graphs and trees

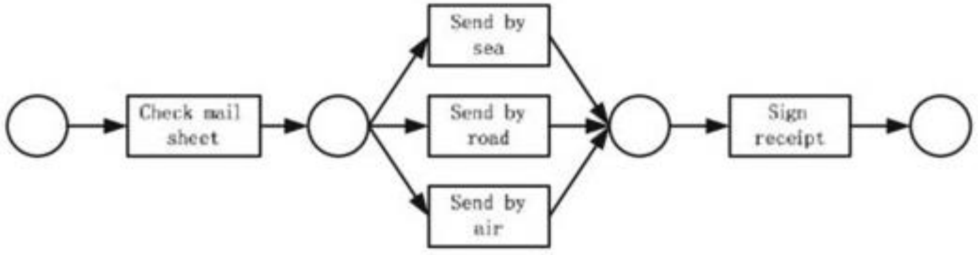
Learning Objectives

- **Formally define a graph or tree**
- **Can distinguish directed and graphs, trees, binary search trees**
- **Name basic properties of graphs and tress**
- **Know how to store graphs and trees computationally**
- **Know traversal algorithms (i.e. DFS & BFS, and Pre-, Post- and Inorder)**
- **Heard of self-balancing AVL-Trees**

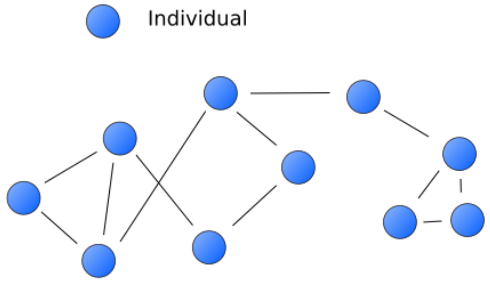
In Business Information System Engineering, graphs are at the core of many applications.

Graph-Examples

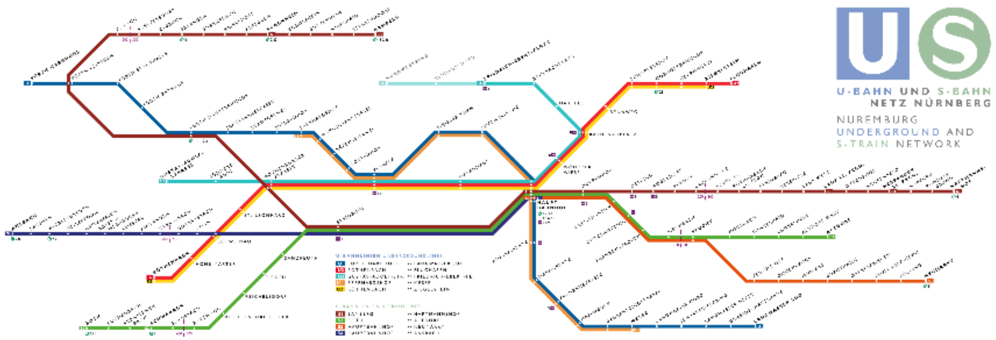
Process Models



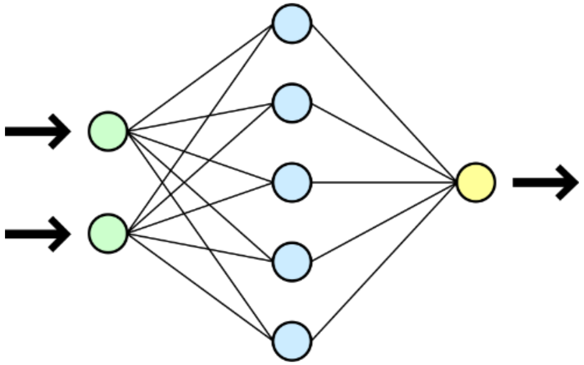
Social Networks



Maps



Neural Networks



Graph theory – formal representation

In 1736, Euler (supposedly) developed the first formally defined graph to solve a riddle.

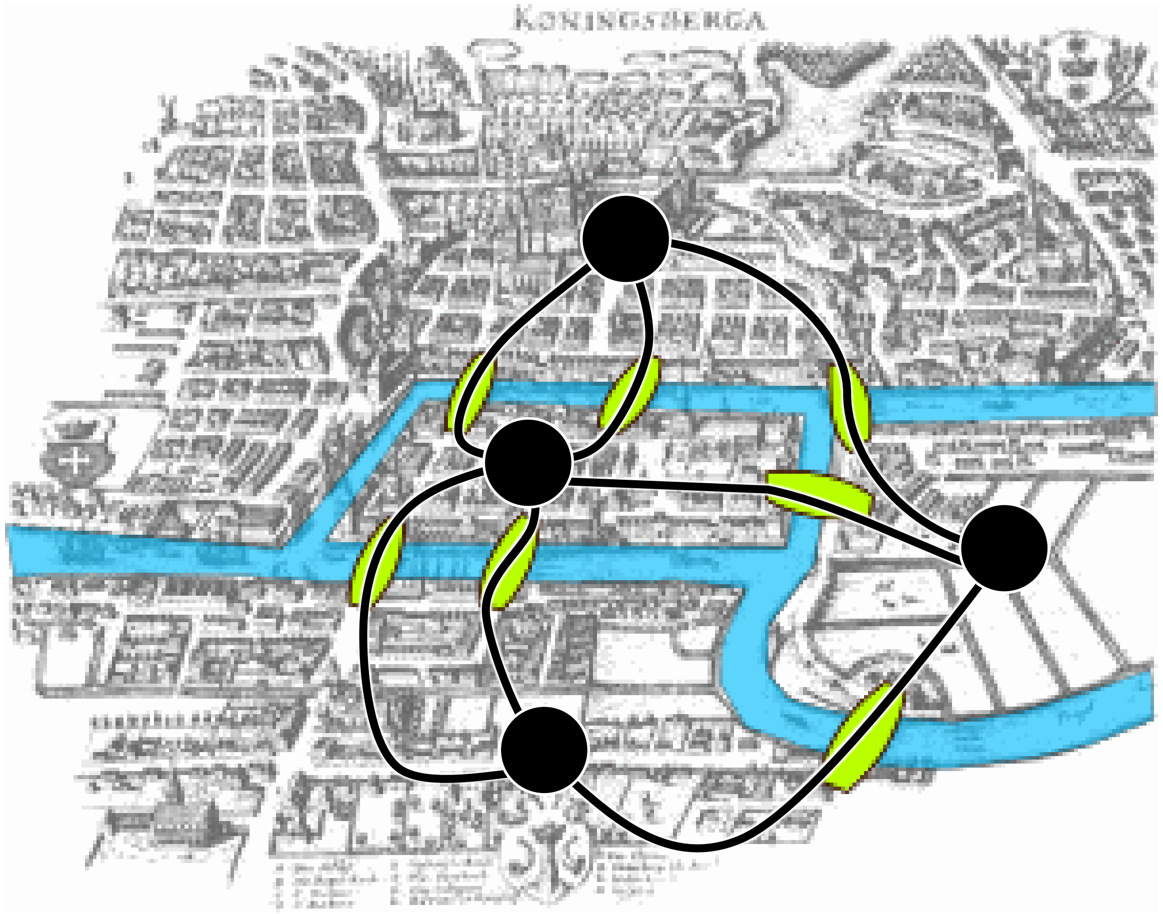
The Seven Bridges of Königsberg

Devise a walk through the city that would cross each of those bridges once and only once.

By specifying the logical task unambiguously, solutions involving either

- reaching an island or mainland bank other than via one of the bridges, or
- accessing any bridge without crossing to its other end

are prohibited.



The formal definition of a directed graph consists of all vertices and all edges in a graph.

Directed Graphs

Definition: $\boldsymbol{G = (V, E)}$ ***where***:

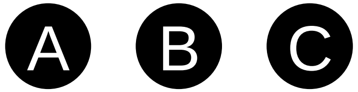
V is the set of all **vertices in a graph**

- $v \in V$ is one vertex of a graph
- for each vertex we draw one node

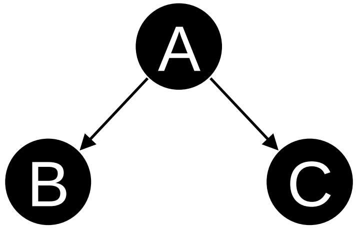
E is the set of all **edges in a graph**

- $\mathrm{e} \in \mathrm{E}$ is one edge of a graph
- $\mathrm{e} = (\mathrm{u}, \mathrm{v})$, e is a relation between two vertices
  - u is the start vertex
  - v is the end/destination vertex
- For each edge, we draw an arrow from the start to the end node

$G = (V, \emptyset), with$
$V = \{A, B, C\}$
$E = \emptyset$
$\emptyset$ is an empty set



$G = (V, E)\ with:$
$V = \{A, B, C\},$
$E = \{(A, B), (A, C)\}$

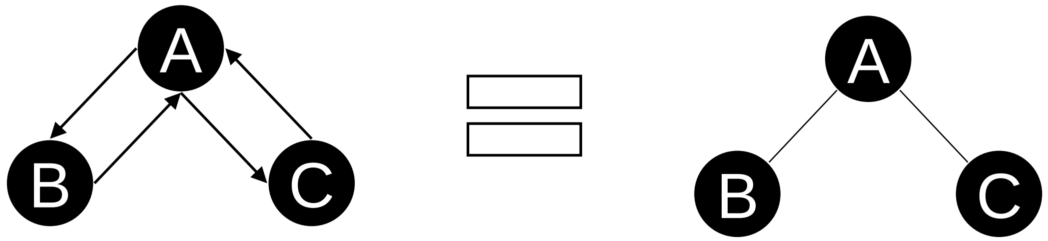


While defining undirected graphs, we do not need to repeat opposite edges.

Undirected Graphs

If a graph: $\boldsymbol{G = (V, E)}$ *has a symmetric set of edges (E) we speak of undirected graphs*:

$G = (V, E)$ *with:*
$V = \{A, B, C\}$,
$E = \{(\boldsymbol{A}, \boldsymbol{B}), (\boldsymbol{A}, \boldsymbol{C}), (\boldsymbol{B}, \boldsymbol{A}), (\boldsymbol{C}, \boldsymbol{A})\}$



Symmetry of E:

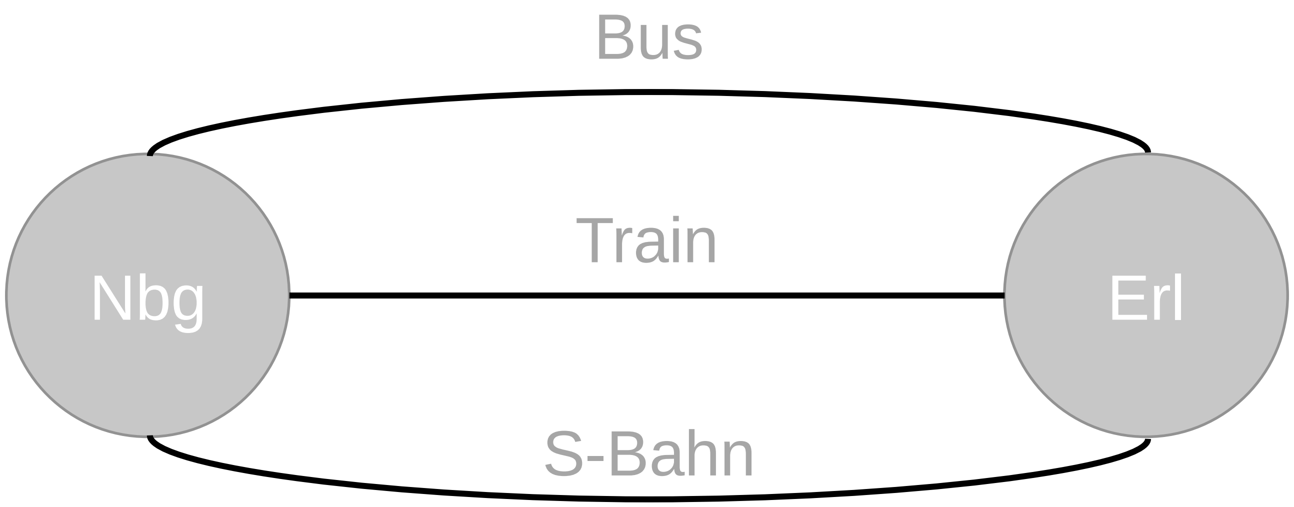
- $\forall\ (e(u, v) \in E\ \exists\ e(v, u) \in E)$
- For all edges from u to v in E, there is also an edge from v to u.

We leave out arrows and simply use lines in undirected graphs instead

Parallel edges are only allowed in Multigraphs.

Multigraphs

- Multigraphs allow **parallel edges**
- Edges are parallel, if they start at the same and end at the same vertices
- Example: Public transport between Nuremberg and Erlangen
- $G = (\{Nbg, Erl\}, E\{(Nbg, Erl), (Nbg, Erl), (Nbg, Erl)\})$

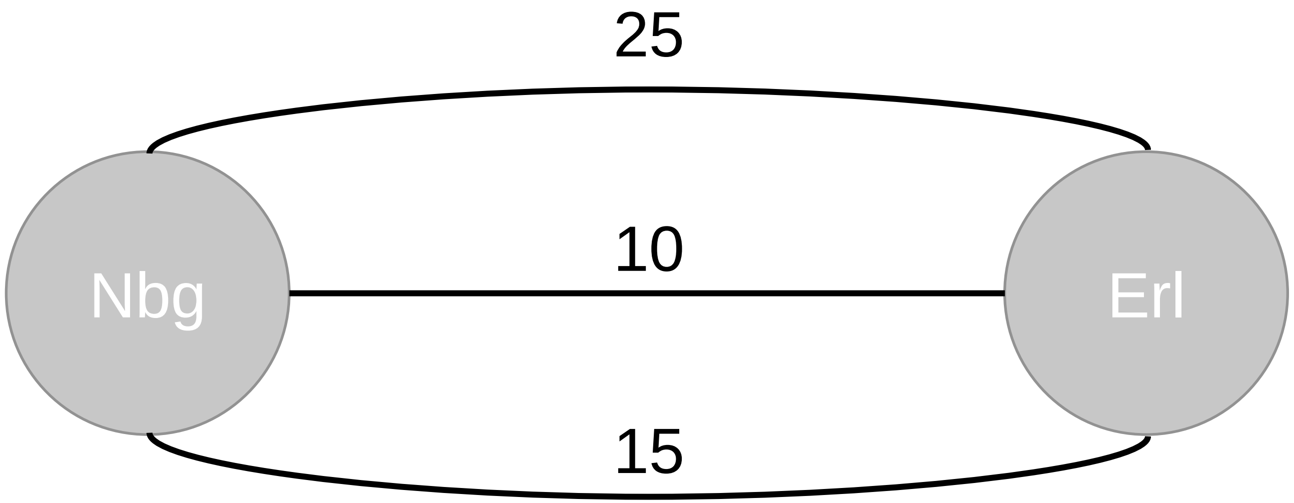


Weighted edges can add information to graphs (e.g. distance in minutes).

Weighted Edges

- Edges can be weighted with values like costs, time, or anything you find useful.
- In the previous example we could either travel by Bus, Train or S-Bahn. Each of these means of transportation takes a different time. Now, we see three ways to travel from Nürnberg to Erlangen, where a way either takes 25, 15, or 10 minutes.

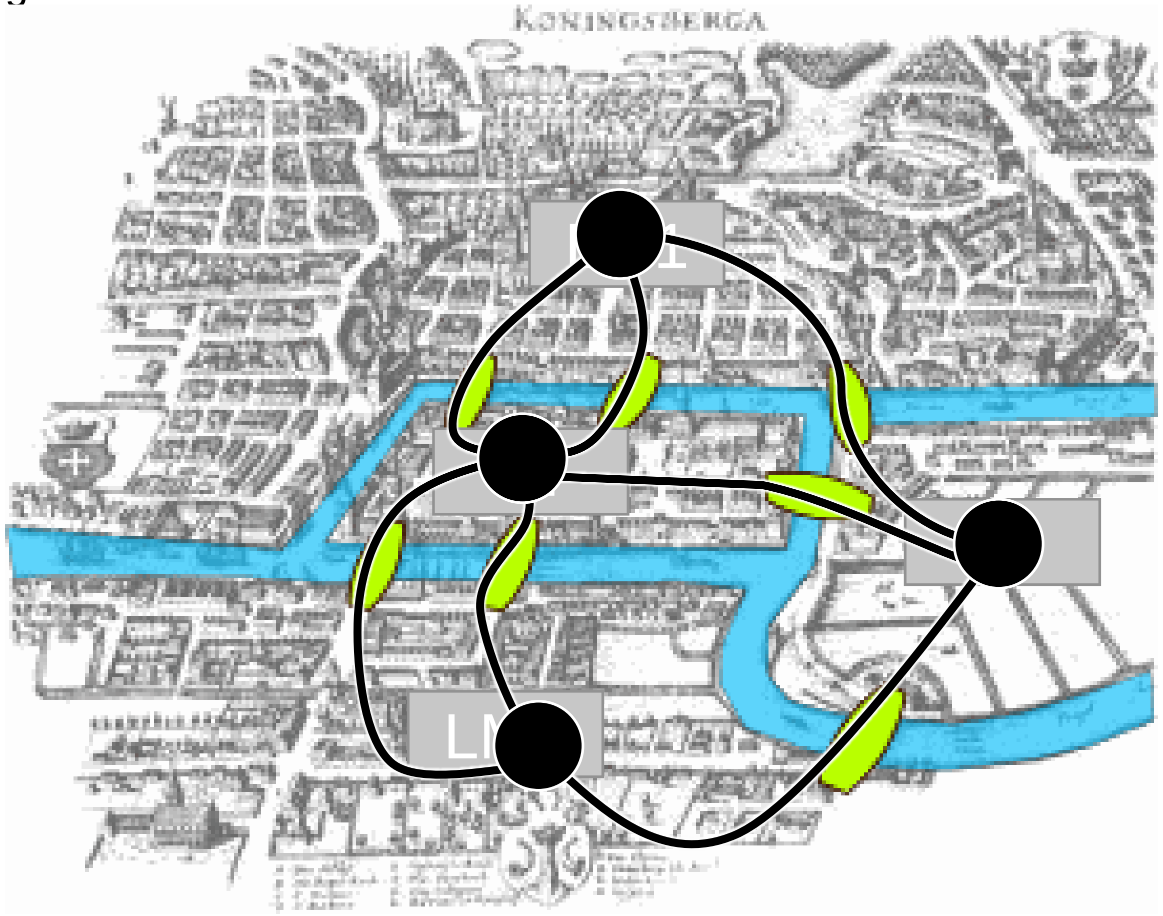
In formal notation:
G (V = {Nbg, Erl}, E = {(Nbg, Erl, 10), (Nbg, Erl, 15), (Nbg, Erl, 25)}



The first graph proved: It is not possible to devise a walk to every landmass and island which crosses every bridge exactly once.

Exemplary Definition of a Graph

- What's the definition of the graph Euler used in Königsberg?
  LM = Landmass
  I = Island
- Is the graph directed?
- Is it a multi or a simple graph?



Solution

Undirected Multigraph $G = (V, E)$

$V = \{LM1, LM2, I1, I2\}$

$E = \{(LM1, I1), (LM1, I1), (LM1, I2),$

$(I1, I2), (I1, LM2), (I1, LM2), (I2, LM2)\}$

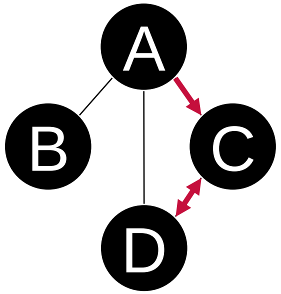
To travel through a graph, graph theorists use the terms: walk, trail, and paths.

Graph Travelling

A (finite) Walk:

- is sequence of vertices leading to a vertex sequence.

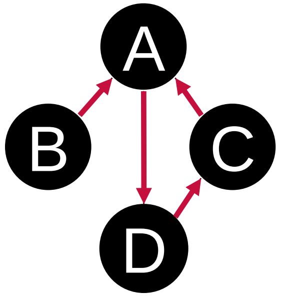
Walk:
A-C-D-C



A (finite) Trail:

- Is a walk where all edges are distinct

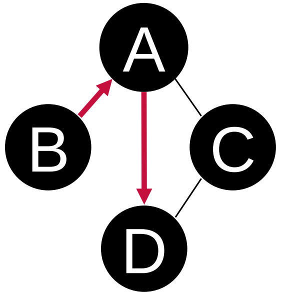
Trail:
B-A-D-C-A



A (finite) Path:

- Is a walk where all edges and vertices are distinct

Path:
B-A-D



Computational representations of graphs

Graphs can be stored computationally in adjacency matrices.

Adjacency matrix

Formal definition:

- Let $G(V, E)$ be a graph with $V = \{v_1, \dots, v_n\}$.
- Then the $n \times n$ Matrix:

$$A_G = (a_{i,j})_{1 \leq i,j \leq n} \;\; where \qquad a_{i,j} = 1 \;\; if \; (v_i, vj) \in E$$

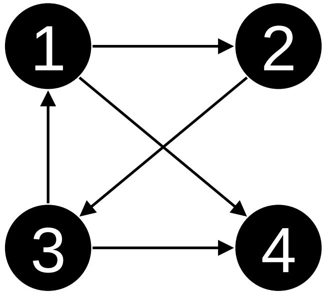
$$a_{i,j} = 0 \;\; otherwise$$

is called adjacency matrix of Graph G

Note:

- For weighted edges in a graph,
  we use the weight instead of the 1 to indicate
  the weight of an edge.

Example



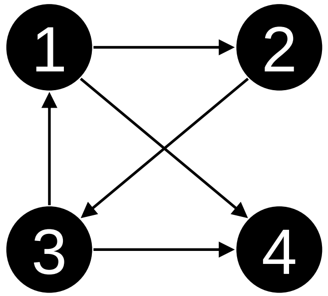
$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Adjacency matrices answer the question – is there an edge from this vertex to the other vertex – with either Yes (1) or No (0).

Adjacency matrix

(Simplified) Explanation:

- Imagine every node as a row and a column in the matrix:



Vertices	1	2	3	4	Can I come here from n?
1	No	Yes	No	Yes	
2	No	No	Yes	No	
3	Yes	No	No	Yes	
4	No	No	No	No	
Can I go to n from here?					

If you look at the column of a node, you find all the incoming edges

If you look at the row of a node, you find all the outgoing edges

Simply convert the previous answers into a two-dimensional array to convert the matrix into its computational representation of a graph

Adjacency matrix

(Simplified) Explanation:

- *Now, since we know vertex 1 is in row 0 and column 0, vertex 2 is in row 1 and column 1. We can omit the vertices numbers in the resulting matrix.*
- *After that we encode a "Yes" as an answer to the previous questions to 1 (or the respective weight) and a "No" to 0.*

Vertices	1	2	3	4
1	No	Yes	No	Yes
2	No	No	Yes	No
3	Yes	No	No	Yes
4	No	No	No	No

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Let's recreate a graph from an adjacency matrix

Adjacency Matrix to Graph

Draw the graph from the following matrix!

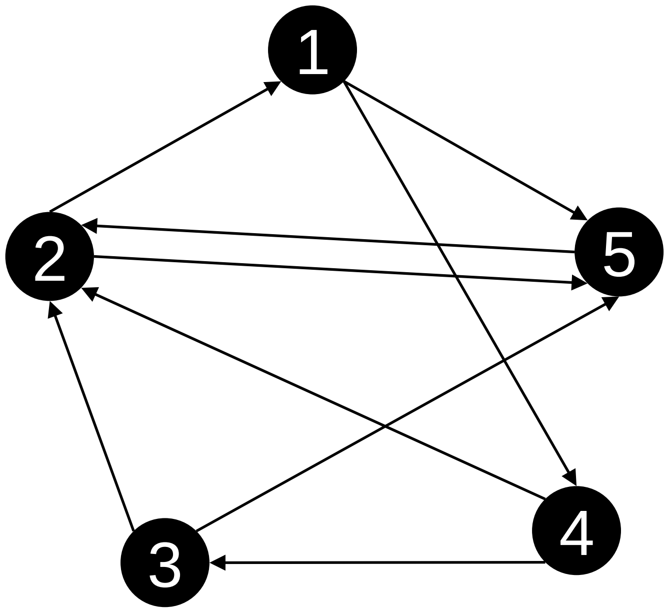
$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

If you draw a graph that you do not know, put all vertices in a circle.

Adjacency Matrix to Graph

Draw the graph from the following matrix!

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix} \equiv$$

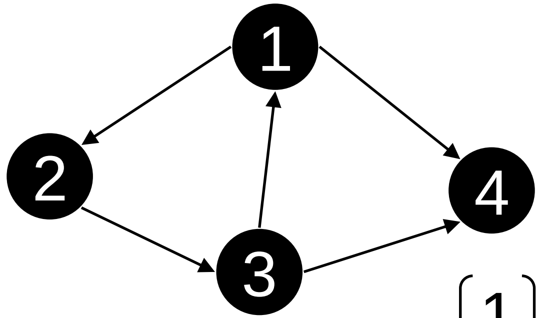


The placement of a node in a drawn image from a graph does not affect its semantics.

Graph Drawing

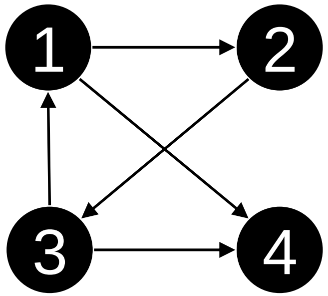
If we draw a graph from an adjacency matrix, the result might look different:

$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \equiv$$



(1)

$=$

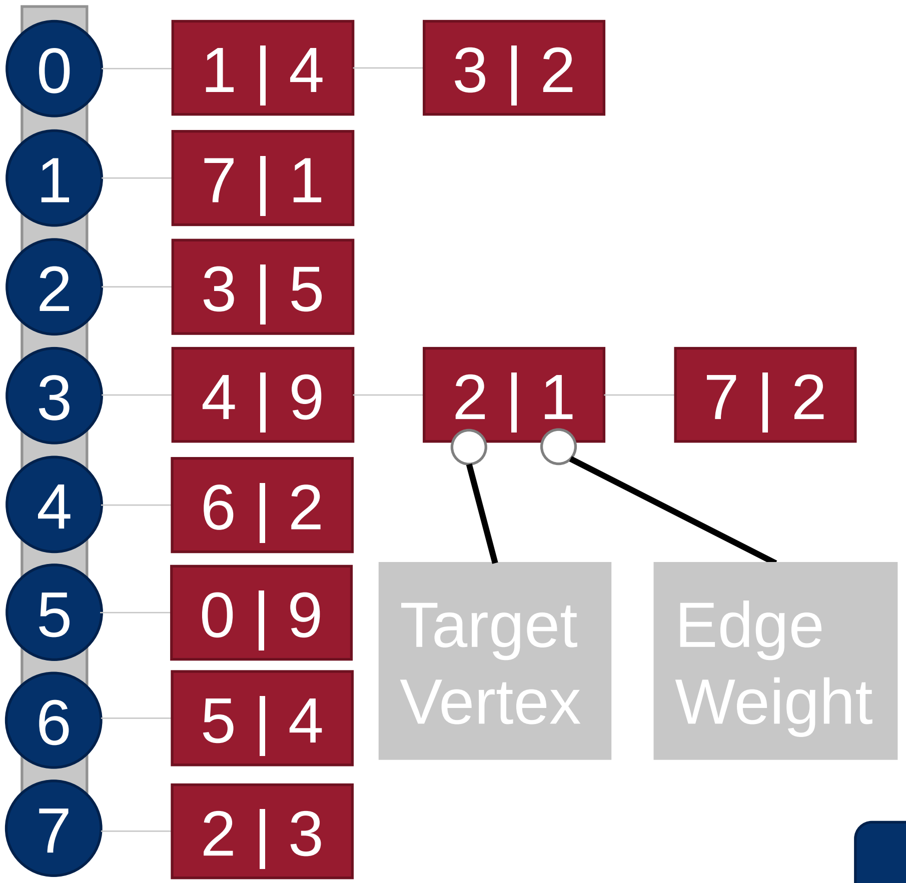


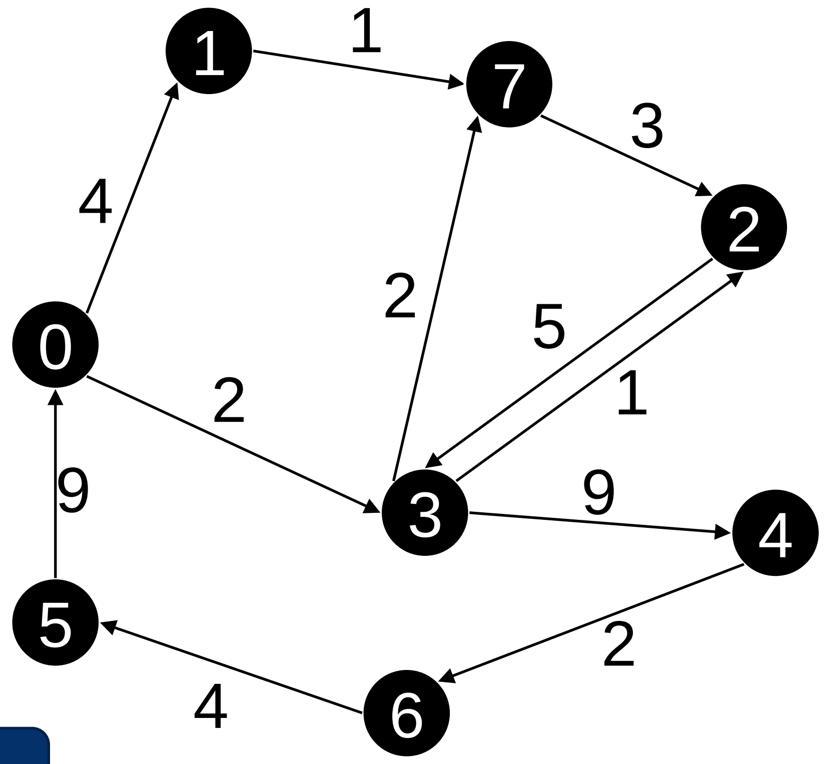
(2)

Even though the graphs (1) and (2) look differently, they are the same. The only difference is the placement of the vertices.

Another way to store graphs computationally is an adjacency list.

Adjacency List





Example

cf: https://visualgo.net/en/graphds

Let’s think about how we might implement an adjacency list in a computer.

Definition

How could we implement an adjacency list?

The implementation of adjacency lists relies on hash tables, arrays, or object orientation.

Definition

How could we implement an adjacency list?

Three popular ways to implement an adjacency list are:

Hash tables (Dictionaries)	Arrays with indices	Object-oriented

An implementation of a graph could, for instance, store all edges and nodes in a graph.

OOP graph

A graph node knows of its neighbors and stores its ID.

```
class GraphNode:
        def __init__(self):
                self.neighbors = []
                self.name = “”

        def add_neighbor(self, node):
                …
```

A graph comprises a list of nodes and all edges per source node in a dictionary.

```
class Graph:
        def __init__(self):
                self.nodes = []
                self.edges = {} # adj. list

        def add_node(self, node):
                …
```

Graph traversal – DFS and BFS

Depth-First Search (DFS) and Breadth-First Search (BFS) are the two strategies to traverse a graph.

BFS and DFS

DFS: Find nodes in a graph by walking down all paths from a node

- Pseudo Code:

```
DFS (node):
    Set up stack and visited list
    Add node to stack
    While stack not empty:
        Set node to stack pop
        Add node to visited
        For neighbor in node.neighbors:
            If neighbor not visited:
                push neighbor on stack
```

DFS: Find nodes in a graph by visiting all neighbors from a node

- Pseudo Code:

```
BFS (node):
    Set up queue and visited list
    Add node to queue
    While queue not empty:
        Set node to queue pop
        Add node to visited
        For neighbor in node.neighbors:
            If neighbor not visited:
                push neighbor into queue
```

In DFS, an algorithm follows every path as long as it can. If it reaches a dead end, it tries the next path.

Depth-first search (DFS)

DFS (node):

  Set up stack and visited list

  Add node to stack

  While stack not empty:

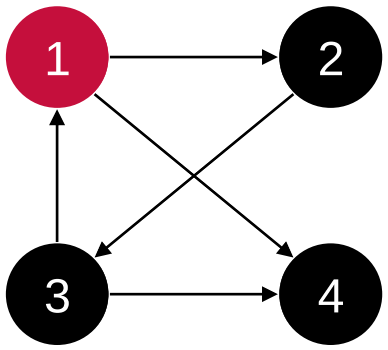
    Set node to stack pop

    Add node to visited

    For neighbor in node.targets:

      If neighbor not visited:

        push neighbor on stack



Node = 1

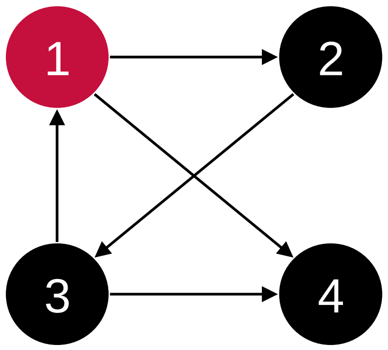
Visited = [ ]

Stack = [1]

In DFS, an algorithm follows every path as long as it can. If it reaches a dead end, it tries the next path.

Depth-first search (DFS) in detail

```
DFS (node):
    Set up stack and visited list
    Add node to stack
    While stack not empty:
        Set node to stack pop
        Add node to visited
        For neighbor in node.targets:
            If neighbor not visited:
                push neighbor on stack
```



Node = 1

Visited = [1]

Stack = [2, 4]

In DFS, an algorithm follows every path as long as it can. If it reaches a dead end, it tries the next path.

Depth-first search (DFS) in detail

DFS (node):

Set up stack and visited list

Add node to stack

While stack not empty:

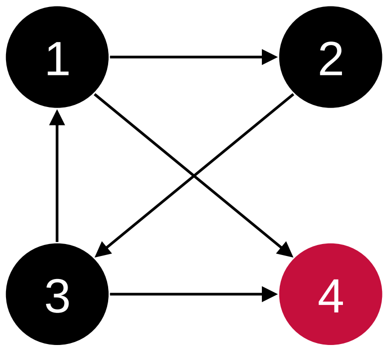
Set node to stack pop

Add node to visited

For neighbor in node.targets:

If neighbor not visited:

push neighbor on stack



Node = 4

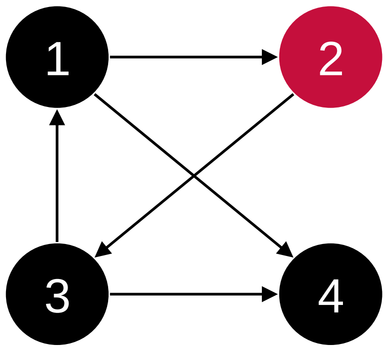
Visited = [1, 4]

Stack = [2]

In DFS, an algorithm follows every path as long as it can. If it reaches a dead end, it tries the next path.

Depth-first search (DFS) in detail

DFS (node):
  Set up stack and visited list
  Add node to stack
  While stack not empty:
    Set node to stack pop
    Add node to visited
    For neighbor in node.targets:
      If neighbor not visited:
        push neighbor on stack



Node = 2

Visited = [1, 4, 2]

Stack = [3]

In DFS, an algorithm follows every path as long as it can. If it reaches a dead end, it tries the next path.

Depth-first search (DFS) in detail

DFS (node):

Set up stack and visited list

Add node to stack

While stack not empty:

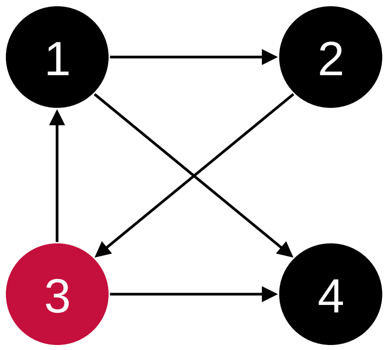
Set node to stack pop

Add node to visited

For neighbor in node.targets:

If neighbor not visited:

push neighbor on stack



Node = 3

Visited = [1, 4, 2, 3]

Stack = [ ]

Stack is empty, therefore we can end the search

In BFS, an algorithm first visits all neighbors of a vertex and does not follow a specific path.

Breadth-first search (BFS)

BFS (node):

  Set up queue and visited list

  Add node to queue

  While queue not empty:

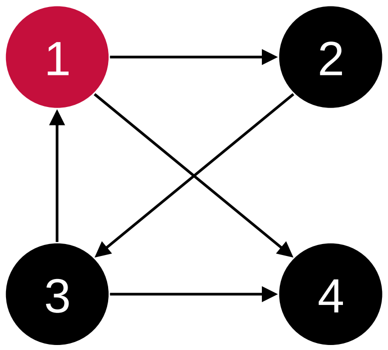
    Set node to queue pop

    Add node to visited

    For neighbor in node.targets:

      If neighbor not visited:

        push neighbor into queue



Node = 1

Visited = [ ]

Queue = [1]

In BFS, an algorithm first visits all neighbors of a vertex and does not follow a specific path.

Breadth-first search (BFS)

BFS (node):

Set up queue and visited list

Add node to queue

While queue not empty:

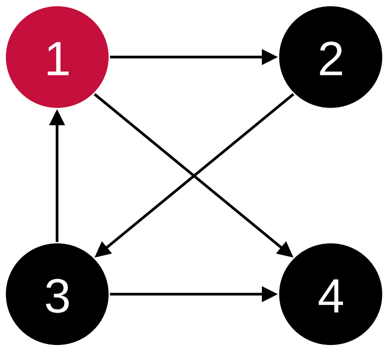
Set node to queue pop

Add node to visited

For neighbor in node.targets:

If neighbor not visited:

push neighbor into queue



Node = 1

Visited = [1]

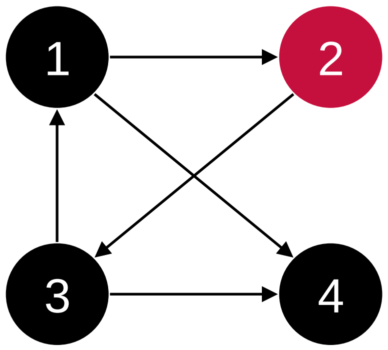
Queue = [2, 4]

In BFS, an algorithm first visits all neighbors of a vertex and does not follow a specific path.

Breadth-first search (BFS)

BFS (node):

- Set up queue and visited list
- Add node to queue
- While queue not empty:
  - Set node to queue pop
  - Add node to visited
  - For neighbor in node.targets:
    - If neighbor not visited:
      - push neighbor into queue



Node = 2

Visited = [1, 2]

Queue = [4, 3]

In BFS, an algorithm first visits all neighbors of a vertex and does not follow a specific path.

Breadth-first search (BFS)

BFS (node):

Set up queue and visited list

Add node to queue

While queue not empty:

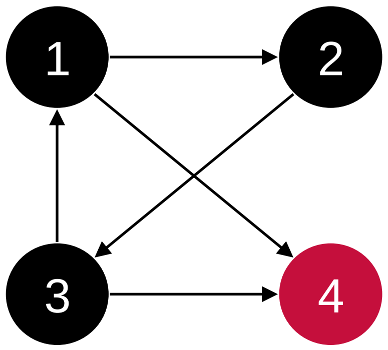
Set node to queue pop

Add node to visited

For neighbor in node.targets:

If neighbor not visited:

push neighbor into queue



Node = 4

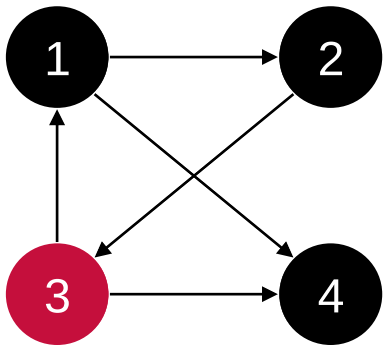
Visited = [1, 2, 4]

Queue = [3]

In BFS, an algorithm first visits all neighbors of a vertex and does not follow a specific path.

Breadth-first search (BFS)

BFS (node):
- Set up queue and visited list
- Add node to queue
- While queue not empty:
  - Set node to queue pop
  - Add node to visited
  - For neighbor in node.targets:
    - If neighbor not visited:
      - push neighbor into queue



Node = 3

Visited = [1, 2, 4, 3]

Queue = [ ]

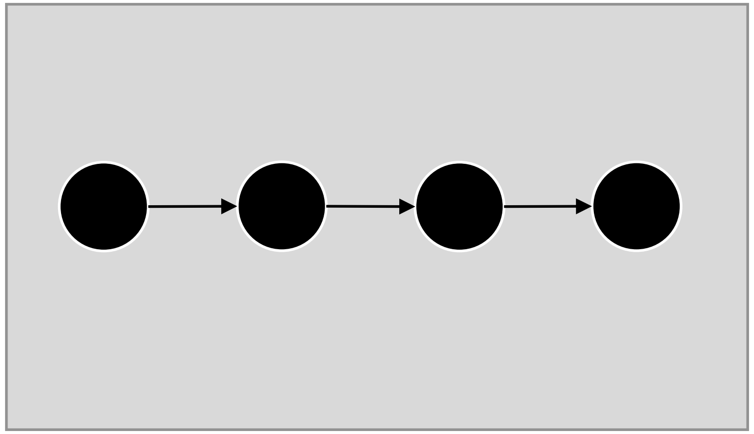
→ Queue is empty, therefore we can end the search

Tree data structures

Abstract data structures that rely on linking elements can be distinguished by their number of predecessors and successors

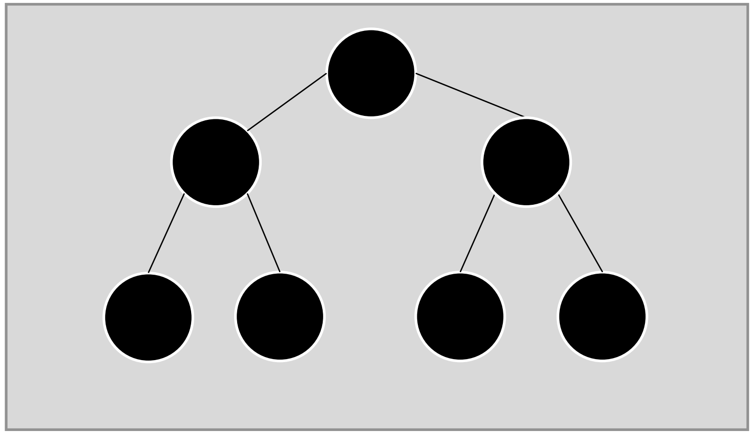
Linking Data Structures

List



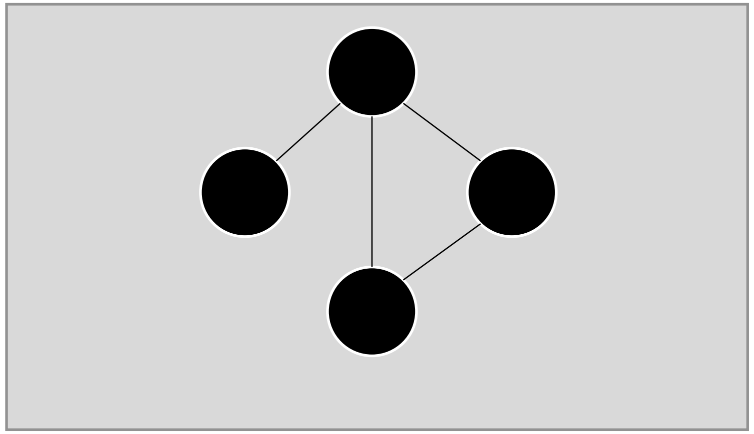
Each element has at most 1 predecessor and 1 successor

Tree



Each element has at most 1 predecessor and 0 to n successors

Graph

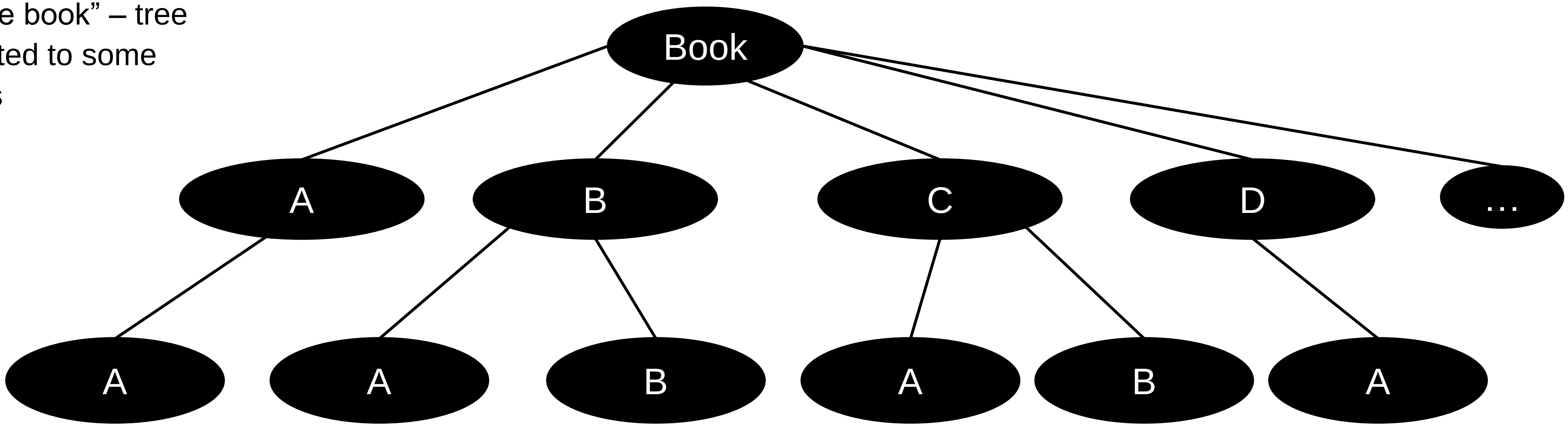


Each element has 0 to n predecessors and 0 to n successors

In a tree representation, we can infer hierarchies into data as in tries or binary trees.

Representation of hierarchical information

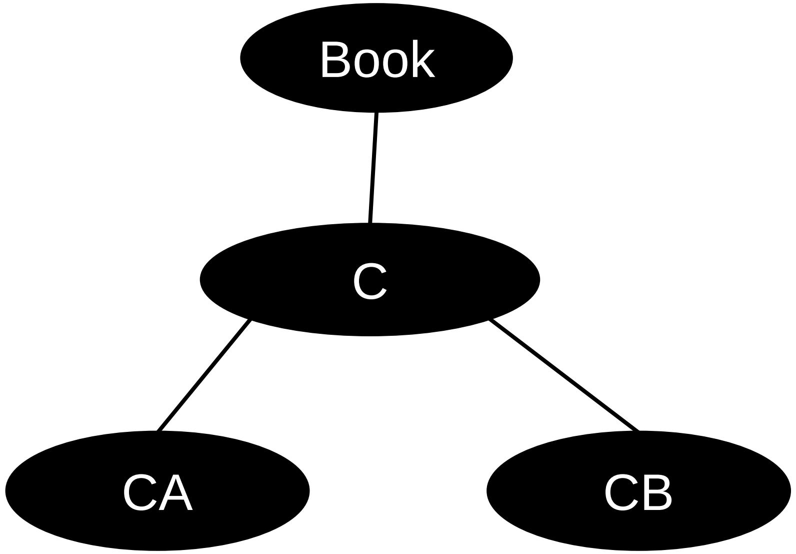
- For example, a phone book:
- Due to simplicity the "phone book" – tree is limited to some letters



Each name contains several letters. Each level of the tree represents one letter.

When we search a name in the previous trie-like tree, we only need two steps to locate the element.

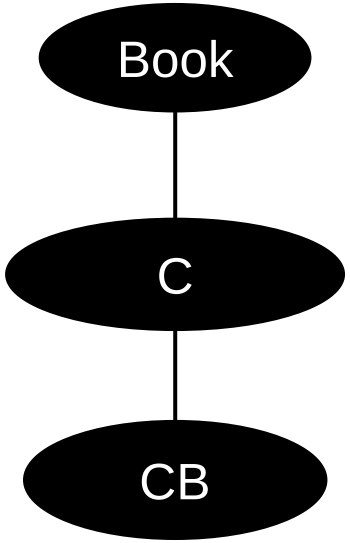
Searching a person with the name CB



With one search step, we limit the phone book search space from 26 (A - Z) nodes to 1 (C).

Tries speed up finding, inserting and deleting elements to *O(k)*, where *k* is the key length of the stored data.

Searching a person with the name CB

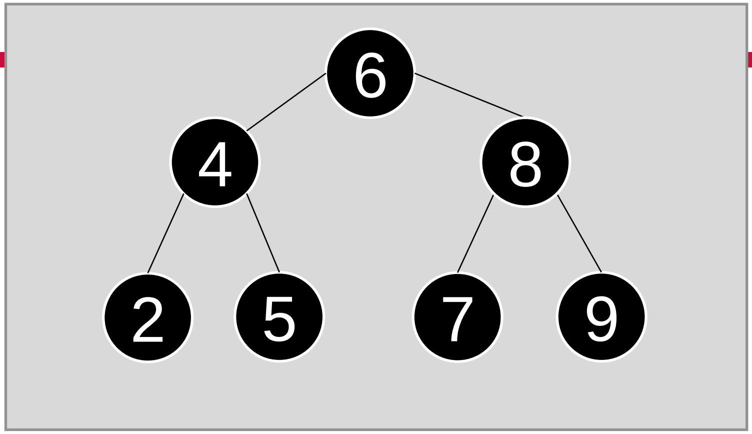


In the next step we found the name “CB”

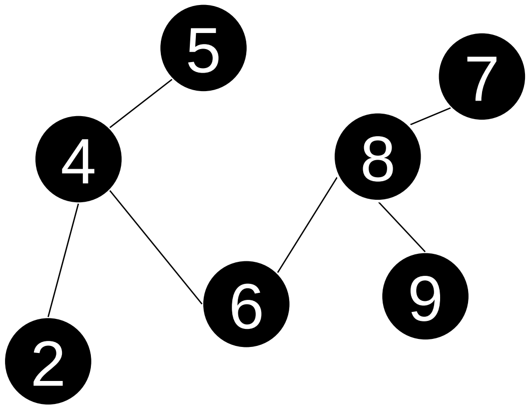
Tree theory

Trees are connected acyclic graphs.

Tree Definition



Is this graph a tree?



Iff (if and only if) there is exactly one path between any two vertices, it is a tree.

Path:
A walk between two vertices where every vertex and edge is distinct.

Nodes are vertices in graph theory. They can be the root, an inner node or a leaf in a tree.

Nodes

Node:

- Nodes are what we called vertices in graph theory.

Root

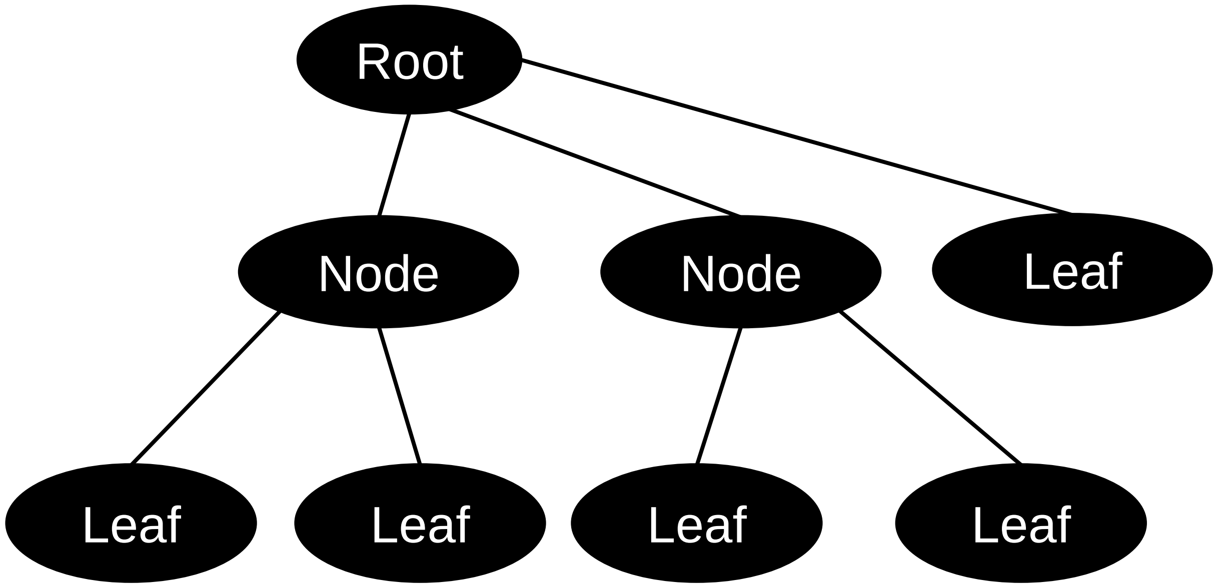
- The only vertex without any predecessors, („Beginning of the tree“)

inner Node

- Node with a predecessor and n successor(s)

Leaf

- Node with a predecessor and 0 successor(s)



A node is in different relationships with other nodes.

Relationships

Parent

- Direct predecessor

Ancestor

- Predecessor of any predecessor of the node

Child

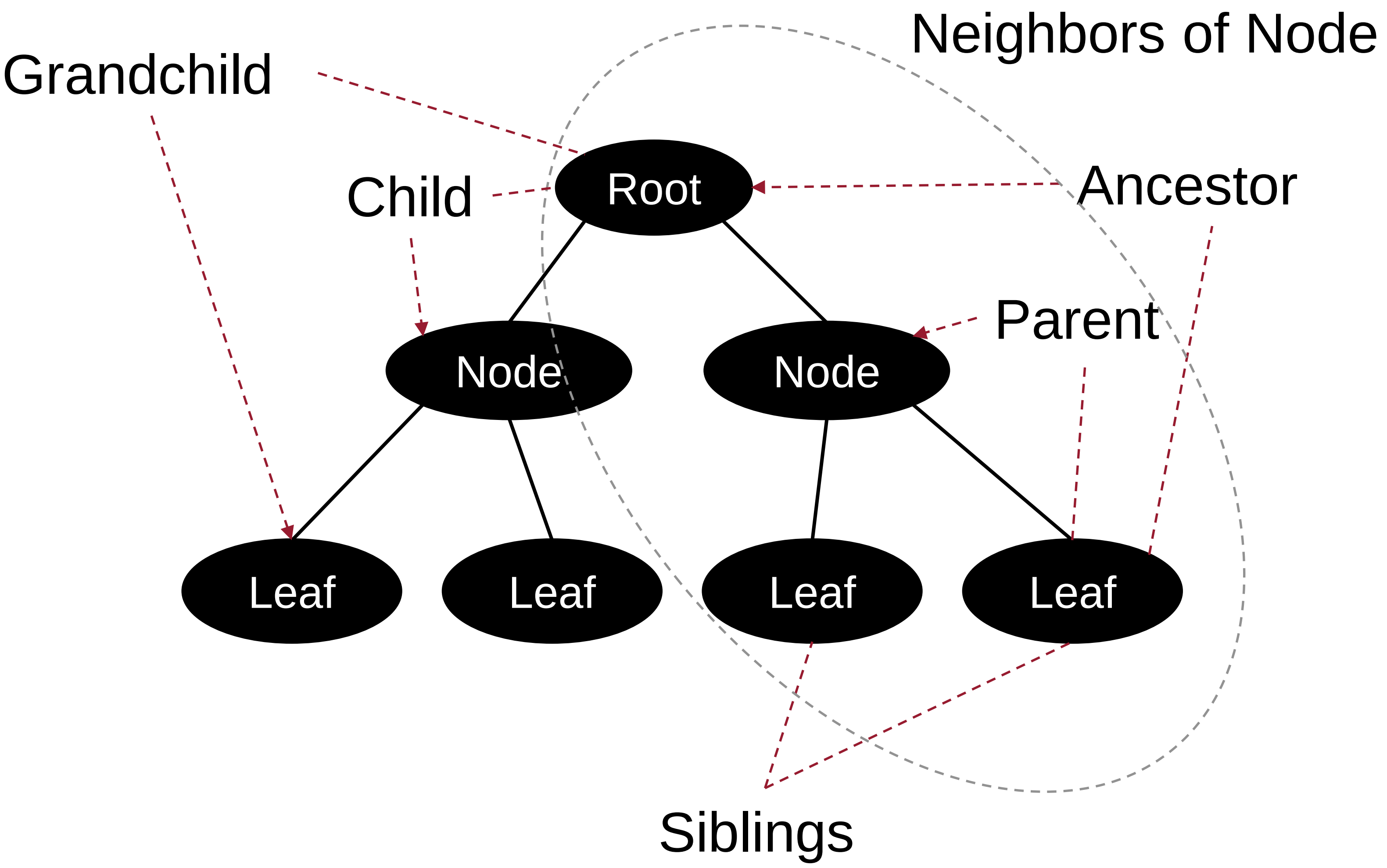
- Direct successor

Grand child

- Successor in the latter of the tree

Siblings

- Nodes with the same Parent



There are several properties to describe a tree or a node. The most prominent ones are height and depth.

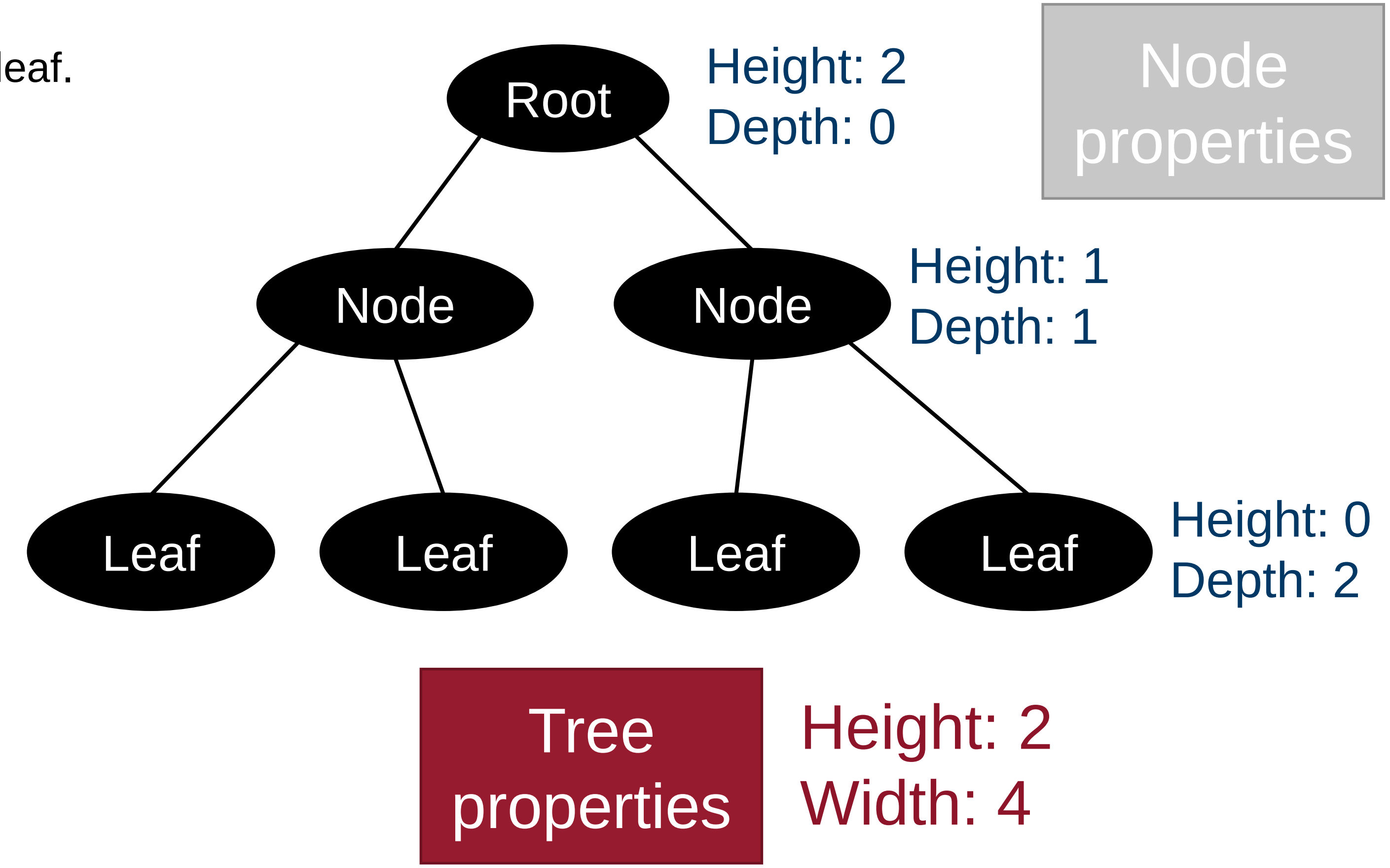
Describing trees

Node properties

- **Height**
  The number of edges to walk from the node to a leaf.
- **Depth**
  The number of edges to walk
  from a node to the root.

Tree properties

- **Height**
  The height of the root node
- **Width**
  The longest path between two leafs



Properties specific to trees are whether they are full, complete, balanced or perfect.

Tree properties

Full

- Every node has either n or 0 children

(left) Complete

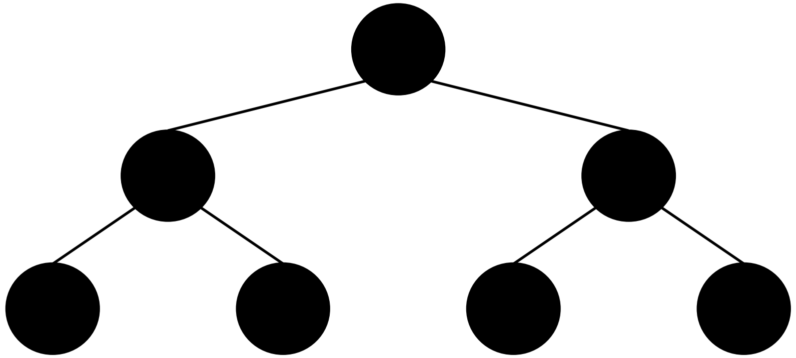
- A complete binary tree is filled at least down to the leaf level.

Balanced

- Height balanced:
  The difference of heights between a node's subtrees is $< \Delta\mathrm{h}$ (for us ∓ 1)
- Fully balanced:
  The difference of nodes in each subtree is $< \Delta n$ (for us ∓ 1)

Perfect

- Complete, Full and completely balanced

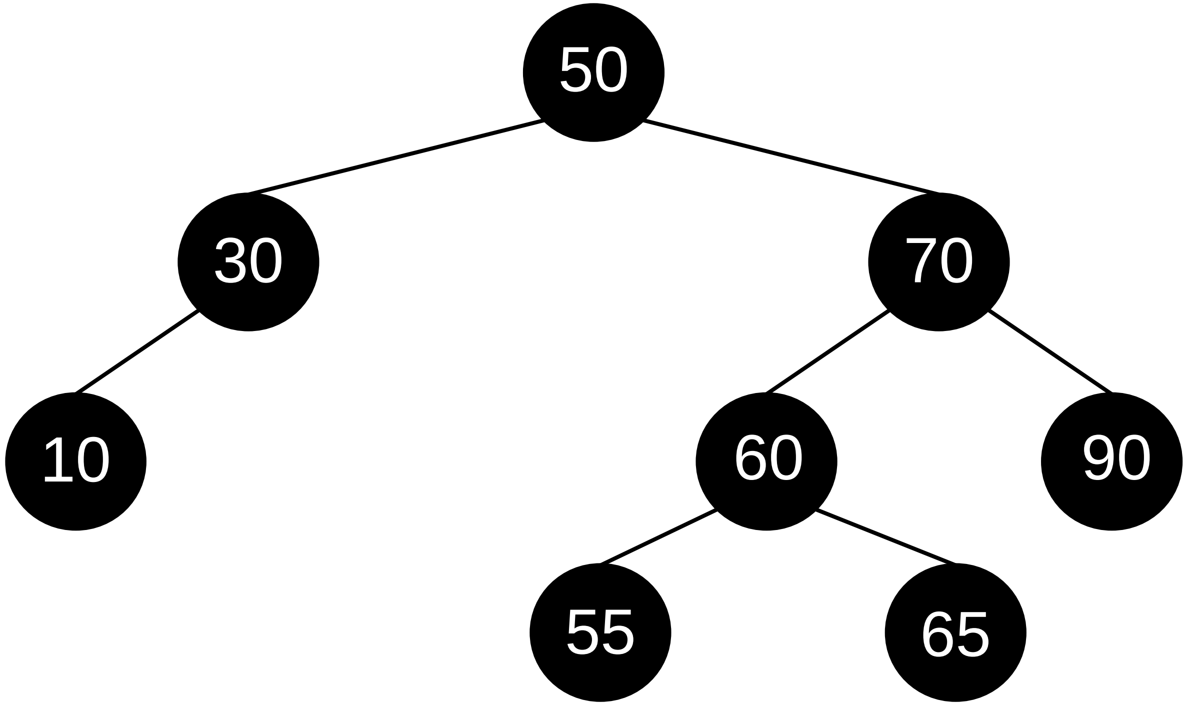


A perfect tree

Let’s classify the following binary search tree.

Example Classification

Classify the following tree?

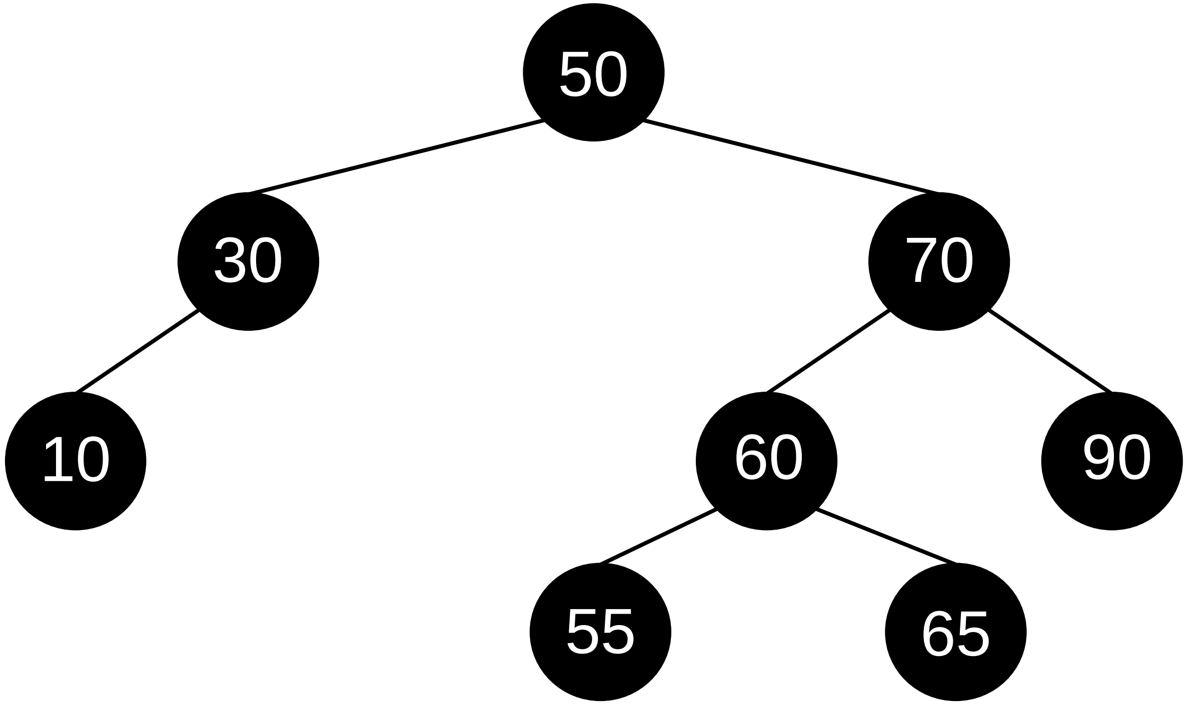


The tree is only height balanced.

Example Classification

Classify the following tree?

Neither full nor complete; but, height balanced.



Binary trees and binary search trees

Binary trees are defined recursively: If both children of a tree node are binary trees, then it is also a binary tree.

Binary Tree Definition

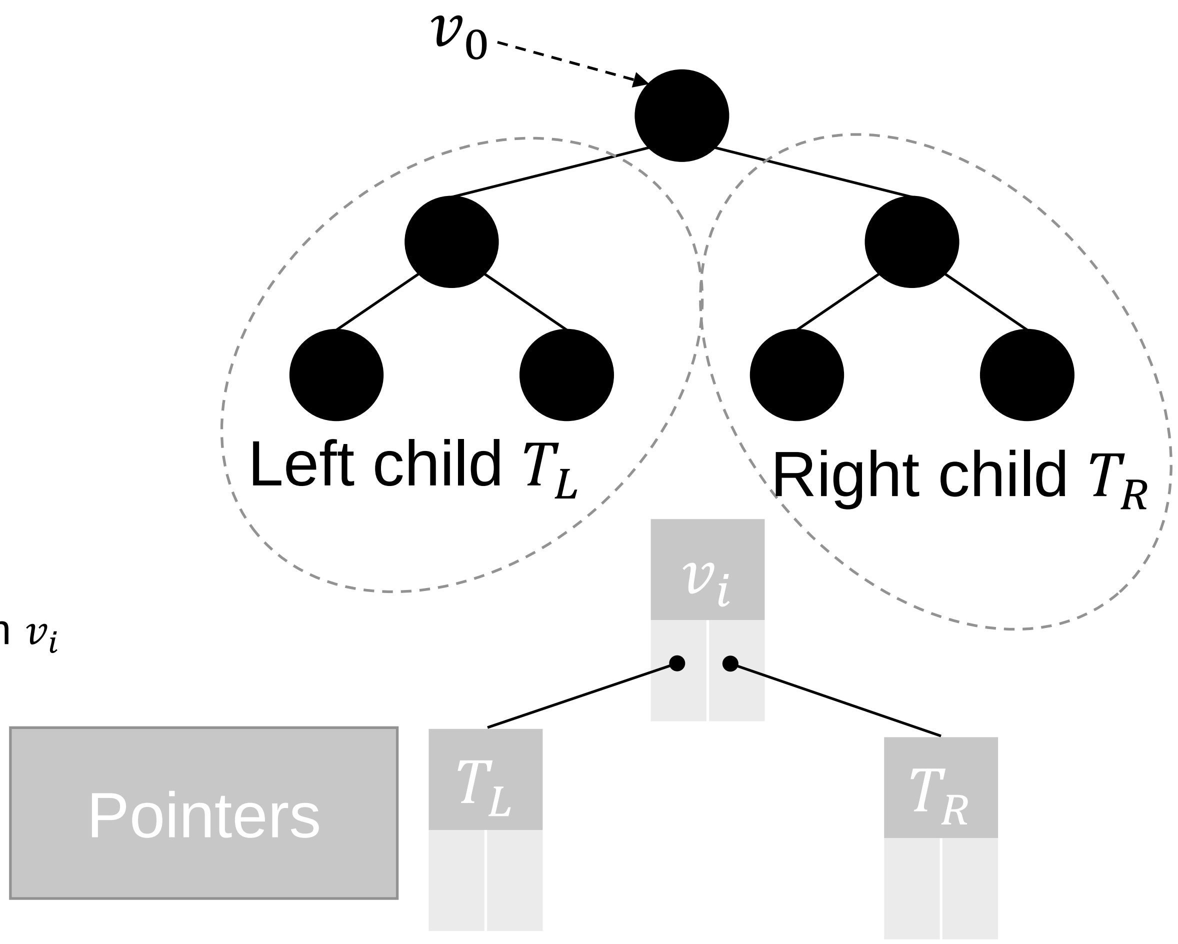
Mathematically, a binary tree, can be defined as the triple

- $T_B = (T_L, v_i, T_R)$
  - Where T is a tree
  - Where v is the root of the ith subtree

A binary tree is always a binary tree when the two children are binary trees.

– **Note:** A binary tree can be empty

To implement binary trees, we can use structs with pointers from v_i to the childs, arrays, or object-oriented programming



The most concise implementation of a binary tree is in a normal array.

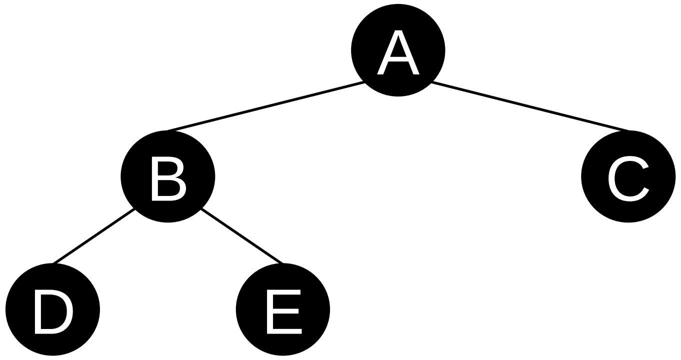
Binary Tree implementation

Two-dimensional array

A	B	C	D	E
0	1	2	3	4
1	3	-1	-1	-1
2	4	-1	-1	-1

One-dimensional array

A	B	C	D	E
0	1	2	3	4



$$\text{node}_i = \text{array}[i]$$
$$\text{left child}_i = \text{array}[2i + 1]$$
$$\text{right child}_i = \text{array}[2i + 2]$$

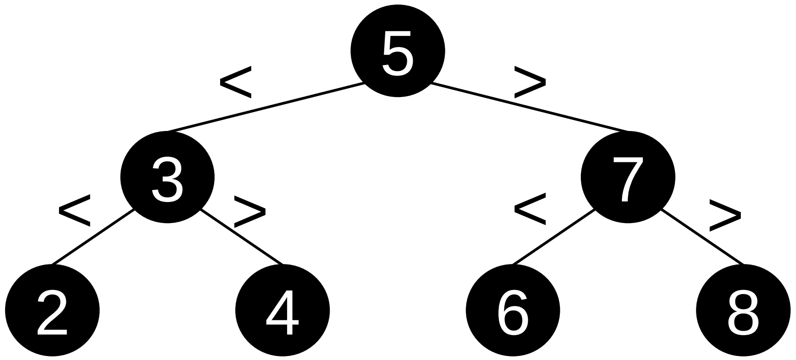
The binary search tree is a data structure that inherently displays the binary search algorithm.

Binary Search Tree

We already got to know one famous divide and conquer approach, the binary search algorithm.

In an average binary search tree, we eliminate half of the search space in one operation

- In a binary search tree, the element are inserted using operators (e.g., ‘<’)
- Every element in a binary search tree is unique
- Every right child is larger than the node and every left child is lower than the node



Read from left to right

Finding an element in a binary search tree is O(log n).

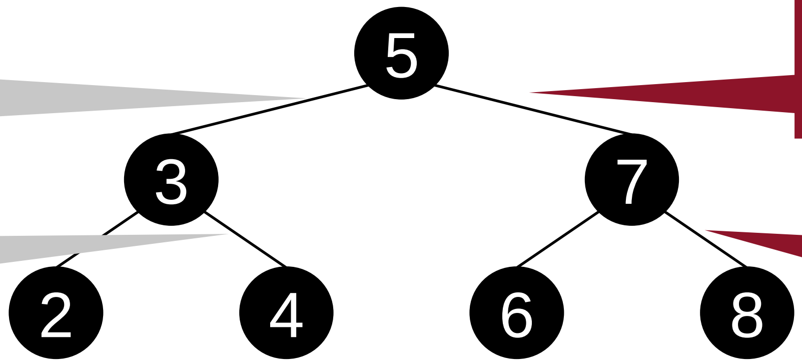
Finding in Binary Search Trees

Find the 4 in the tree

Go left because 4 < 5

Go right because 4 > 3

return true;



Find the 9 in the tree

Go right because 9 > 5

Go right because 9 > 7

return false;

Let’s construct a binary search tree.

Creating a binary search tree

Create a binary search tree with the following values!
50, 30, 70, 60, 10, 20, 90, 40

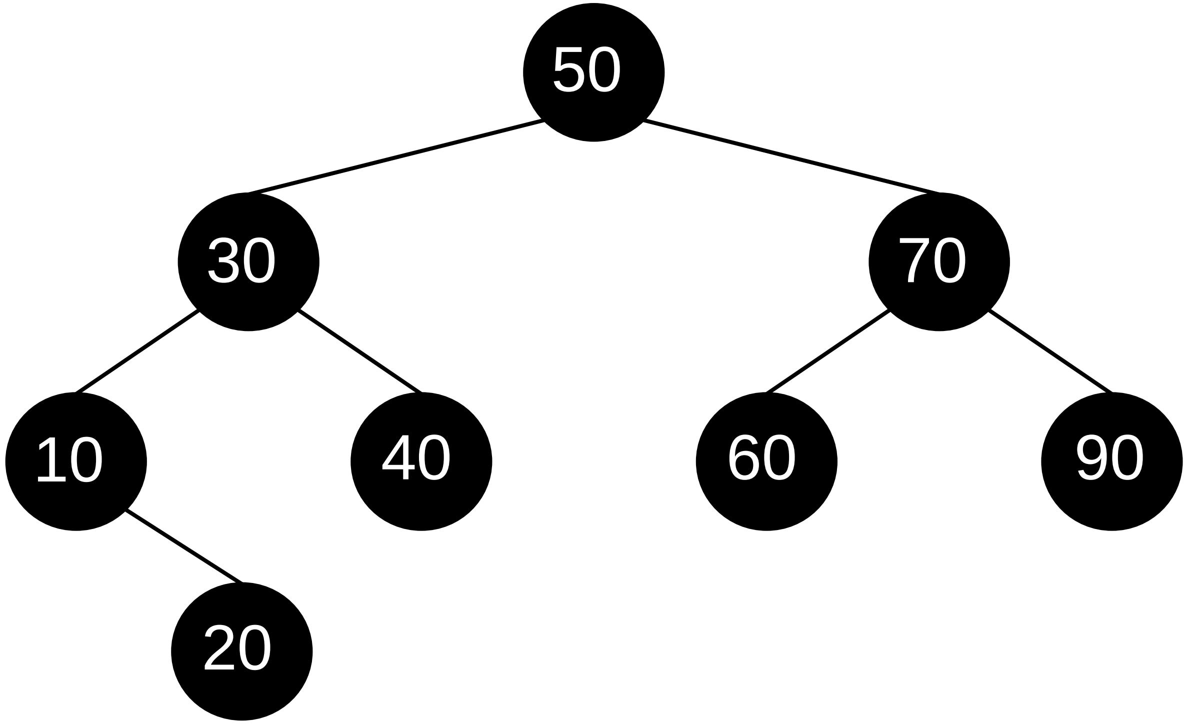
Binary Search Trees

Creating a binary search tree

Let’s create a binary search tree with the following values
50, 30, 70, 60, 10, 20, 90, 40

- The first element we add is always the root.

50	30	70	60	10	20	90	40



The mere creation of a binary search tree from a list by using only the operator can cause a problem.

Problem: Creating Binary Search Trees

In Lecture 5 – Data structures you heard of a problem when creating BSTs … do you remember it?

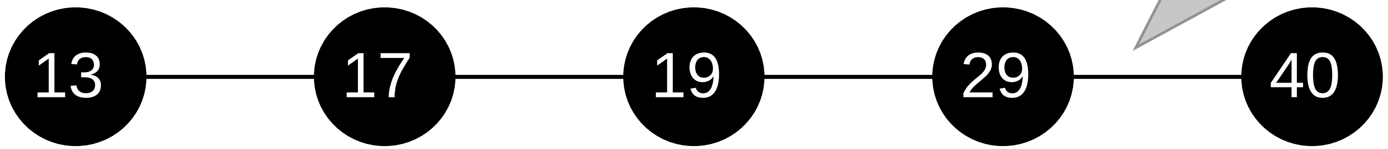
When degenerating a binary search tree, we receive a linked list.

Creating binary search trees

What might cause problems when creating binary search trees?

We lost all efficiency of binary search trees by inserting sorted elements

Create the following binary tree and insert the elements in the order they come:
13, 17, 19, 29, 40



Degenerate
every parent has only one child ~
Linked List

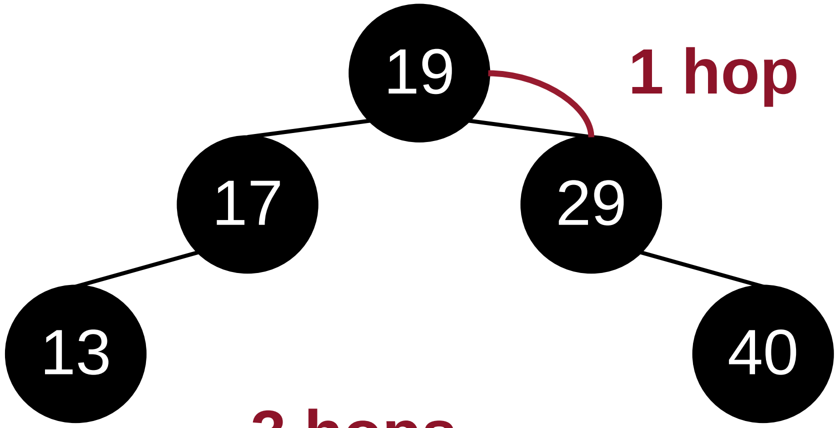
We like balanced binary trees better than unbalanced ones.

Balanced vs. not Balanced

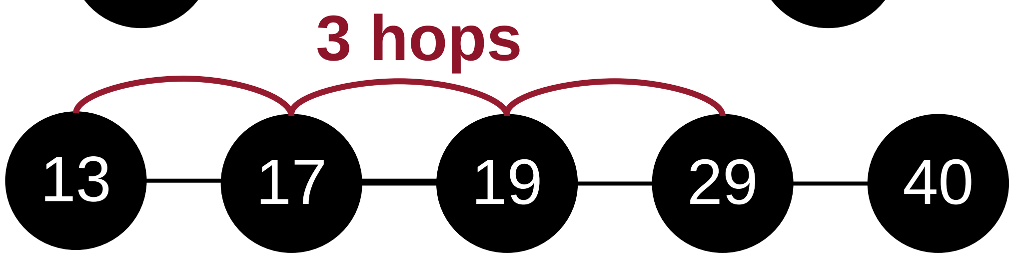
Average runtime of searching:

Find 29 in both data structures

$O(\log n)$



O(n)



The deletion of a leaf just removes the leaf. The deletion of an inner node with one child replaces the node to be deleted with its child.

Deleting Nodes

We need to distinguish three cases:

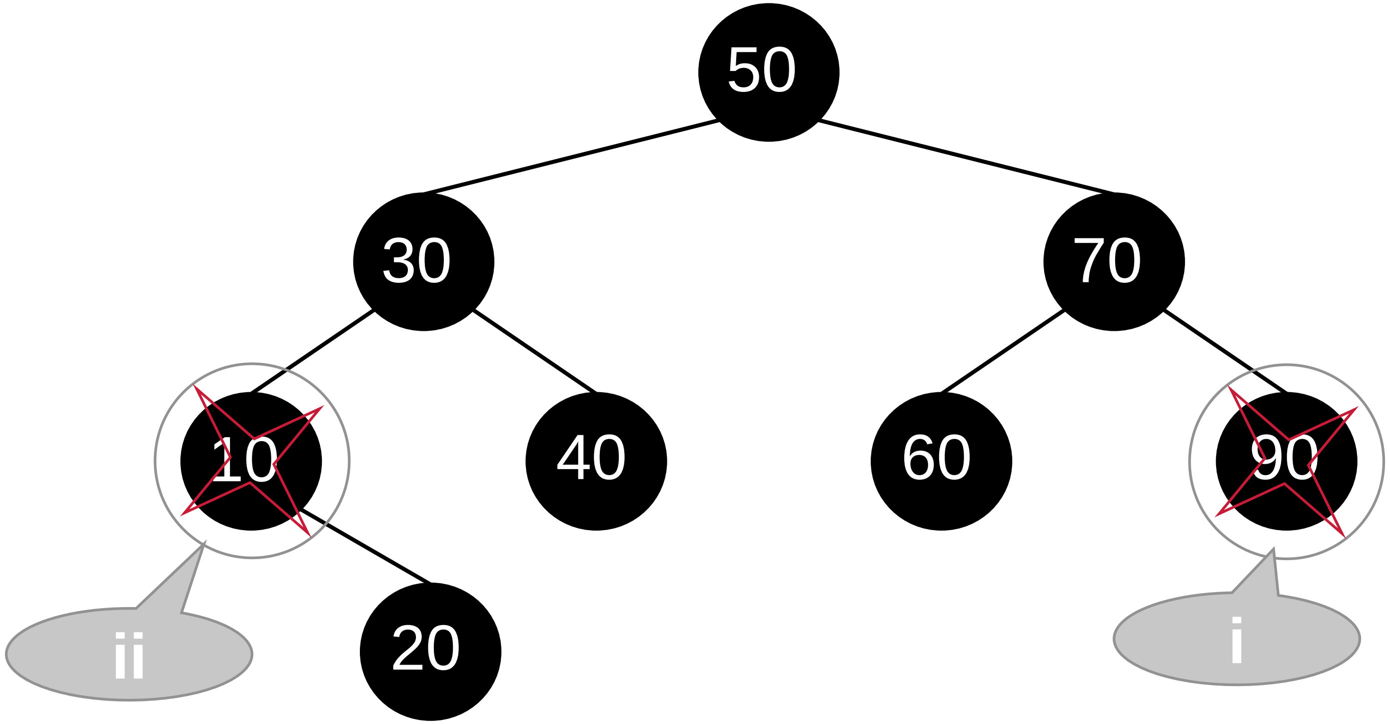
i. **Deleting a leaf**

Just delete the node

ii. **Deleting a node with one child**

Swap child to own position

iii. **Deleting a node with two children**



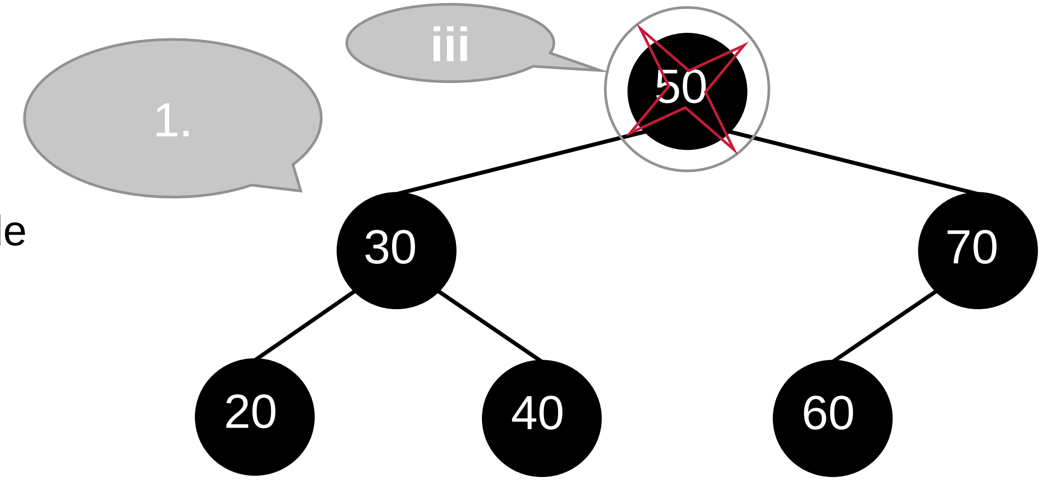
There are two strategies for deleting a node with two children.

Deleting Nodes

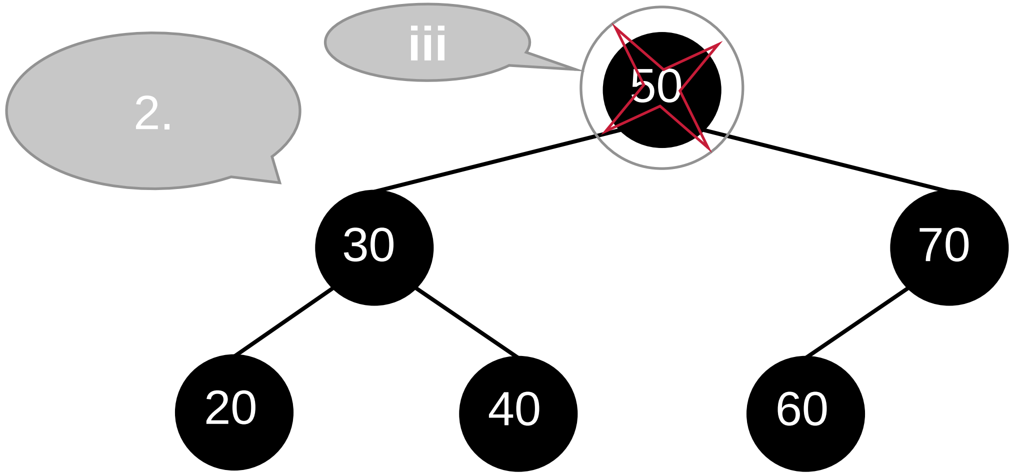
iii. Deleting a node with two children

Two strategies:

1. Find minimum of right subtree and replace with the deleted node



2. Find maximum of left subtree and replace with the deleted node



Inorder-traversal traverses a binary search tree in sorted order.

Traversal

Preorder: “5 3 1 2 4 7 6 9 8”

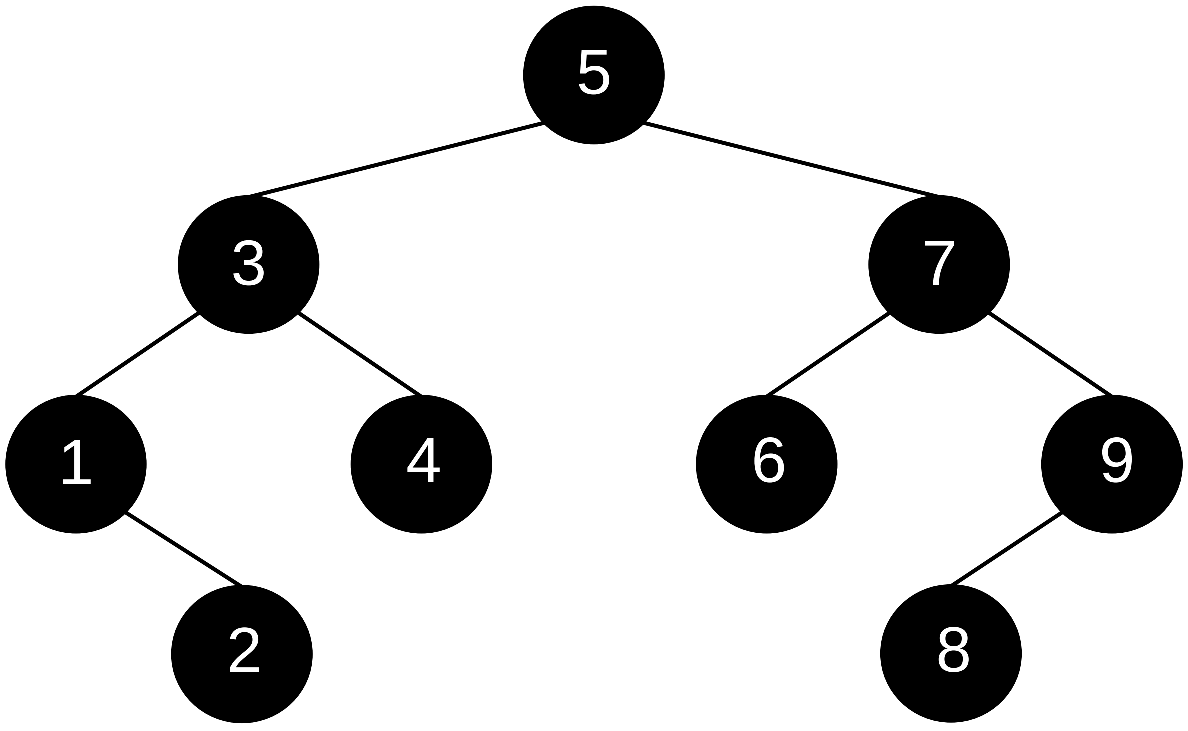
1. Print value
2. Go to left child
3. Go to right child

Inorder: “1 2 3 4 5 6 7 8 9”

1. Go to left child
2. Print value
3. Go to right child

Postorder: “2 1 4 3 6 8 9 7 5”

1. Go to left child
2. Go to right child
3. Print value

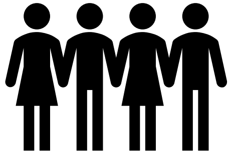


Binary Tree Visualizer

There is a tree data structure which can automatically balance a tree.

Motivation

If balanced trees are so much better than unbalanced trees, why don’t we get self-balancing trees?



Ok then, let’s take a look at self-balancing trees.

Adelson Velsky and Landis (AVL-) trees

AVL-Trees define a structural invariant that expresses that a tree must be height balanced.

Adelson-**V**elsky and **L**andis (AVL) Trees

- A self-balancing tree data structure
- Searching, Inserting and Deleting is $O(\log n)$ in the average and worst case
- **Idea**:
  Define a structural invariant. Every time one updates (delete or inserts) the tree check the invariant, and if required enforce it.
- In natural words: an AVL Tree's structural invariant says that the tree must be height balanced.

An AVL tree rebalances by specific rotation rules to achieve a tree that is always at least height balanced.

AVL Definition

The Invariant that AVL trees enforce is as follows:

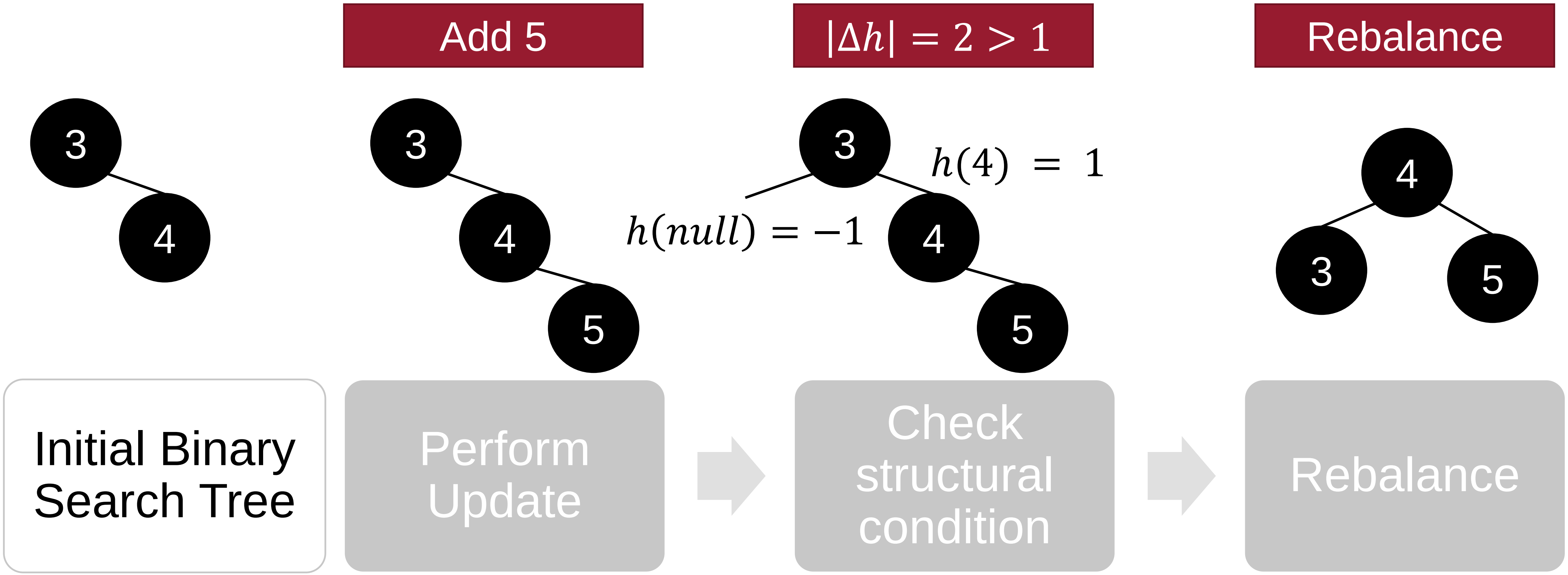
Let v be any node in a binary search tree and $h(v)$ be the function to determine its height.

- The height of both children $v.leftchild$ and $v.rightchild$ differ by 1 at most.
- A non-existing node always has a height difference of -1.

Structural Invariant (AVL):
$$|h(v.left) - h(v.right)| \leq 1$$

Whenever an update violates the AVL invariant, the tree "**rebalances**".

AVL Trees reevaluate their structural invariant after every update, i.e., addition or deletion of a node.



After this lecture, students understand the theory of graphs and trees

Learning Objectives

- **Formally define a graph or tree**
- **Can distinguish directed and graphs, trees, binary search trees**
- **Name basic properties of graphs and tress**
- **Know how to store graphs and trees computationally**
- **Know traversal algorithms (i.e. DFS & BFS, and Pre-, Post- and Inorder)**
- **Heard of self-balancing AVL-Trees**

Friedrich-Alexander-Universität
School of Business,
Economics and Society

Chair of Digital Industrial Service Systems

Prof. Dr. Martin Matzner
Friedrich-Alexander-Universität Erlangen-Nürnberg
School of Business, Economics and Society | WiSo
wiso-is-kontakt@fau.de
twitter.com/ismama
www.is.rw.fau.eu