Discrete mathematics

Graphs, trees, lists

Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Graphs

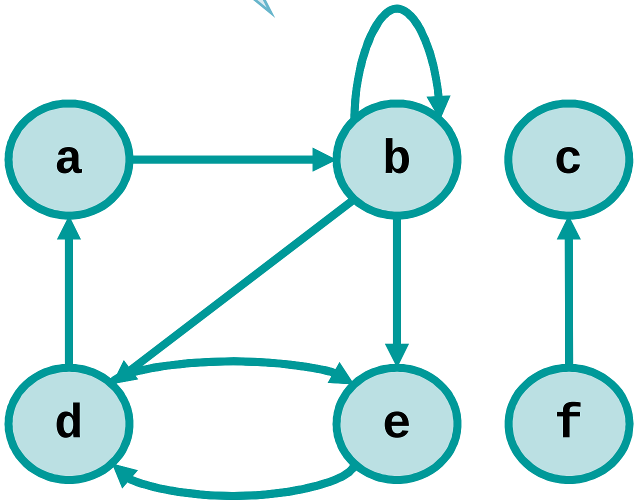
- Definition
  - G = (V, E)
    - V = Finite and non empty set of vertices (simple or complex data)
    - E = Finite set of edges, that define a binary relation on V
- Directed/Undirected graphs
  - Directed
    - Edge = sorted pair of vertices $(u, v) \in E$ and $u, v \in V$
  - Undirected
    - Edge = unsorted pair of vertices $(u, v) \in E$ and $u, v \in V$

Applications

Domain	Vertex	Edge
communications	phone, computer	fiber optic, cable
circuits	gate, register, processor	wire
mechanics	joint	spring
finance	stocks, currencies	transactions
transports	airoport, station	air corridor, railway line
games	position on board	legal move
social networks	person	friendship
neural networks	neuron	synapsis
chemical compounds	molecules	link

Example: Directed graph

```
G = {V, E}
V = {a, b, c, d, e, f}
E = {(a,b),(b,b),(b,d),(b,e),
     (d,a),(d,e),(e,d),(f,c)}
```

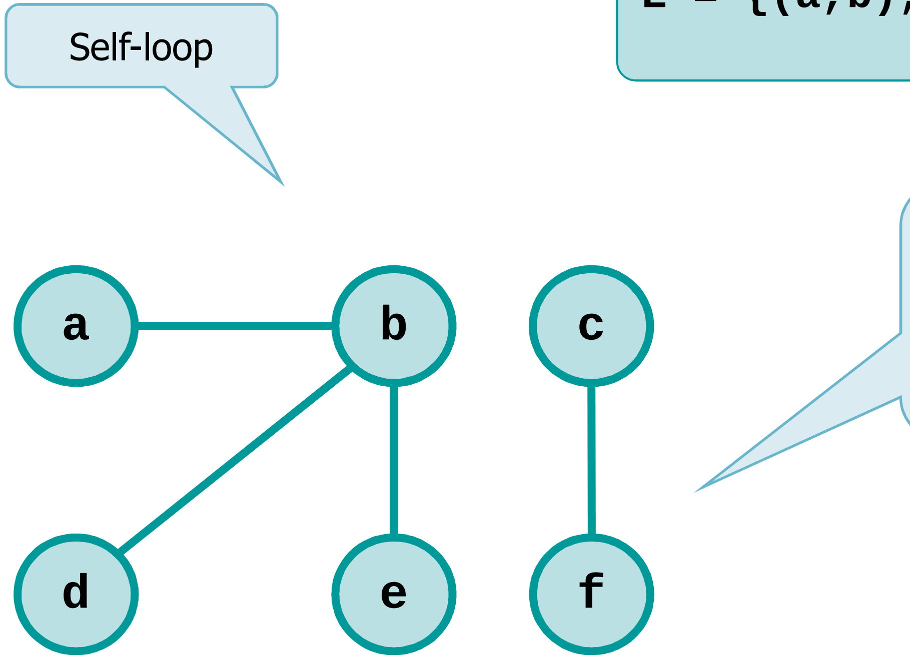


Self-loop

In some contexts self-loops may be forbidden.
If the context allows loops, but the graph is self-loop-free, it is called **simple**

Example: Undirected graph

```
G = {V, E}
V = {a, b, c, d, e, f}
E = {(a,b),(b,d),(b,e),(c,f)}
```



Self-loop

In some contexts self-loops may be forbidden.
If the context allows loops, but the graph is self-loop-free, it is called **simple**

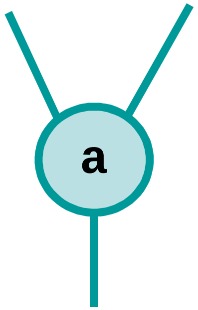
Edges

- Edges
  - An edge (a, b) can be
    - Incident from vertex a
    - Incident in vertex b
    - Incident on vertices a and b

  - Vertices a and b are adjacent
    - $a \rightarrow b \Leftrightarrow (a, b) \in E$

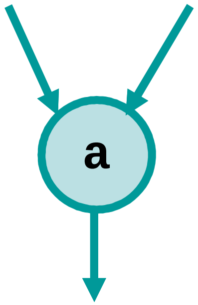
Edges

- Undirected graph
  - Degree (a) = number of incident edges



Degree (a) = 3

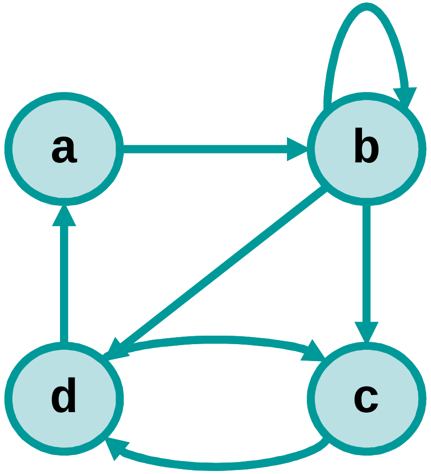
- Directed graph
  - In-degree (a) = number of incoming edges
  - Out-degree (a) = number of outgoing edges
  - Degree (a) = in-degree(a) + out-degree(a)



In-degree (a) = 2
Out-degree (a) = 1
Degree (a) = 3

Paths

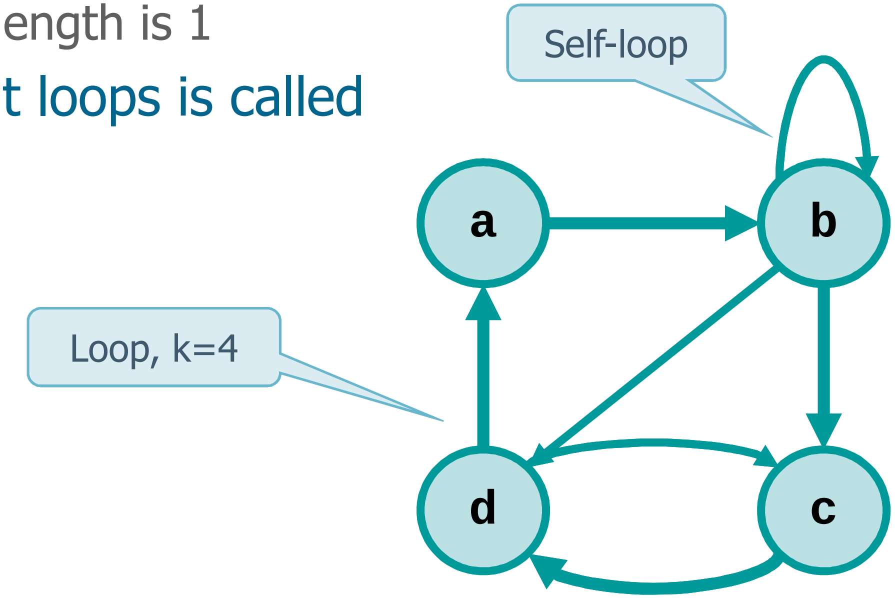
- Paths
  - A path p, u $\rightarrow_p$ u', is defined in G = (V, E) as
    - $\exists(v_0,v_1,v_2,\ldots,v_k) \mid u=v_0,\ u'=v_k,\ \forall i = 1,2,\ldots,k\ (v_{i-1},v_i) \in E$
  - k = length of the path
  - u' is reachable from u $\Leftrightarrow$ $\exists$ p: u $\rightarrow_p$ u'
  - Simple path p: distinct $(v_0,v_1,v_2,\ldots,v_k) \in$ p



G = (V, E)
p: a $\rightarrow_p$ d : (a, b), (b, c), (c, d)
k = 3
d is reachable from a
p is a simple path

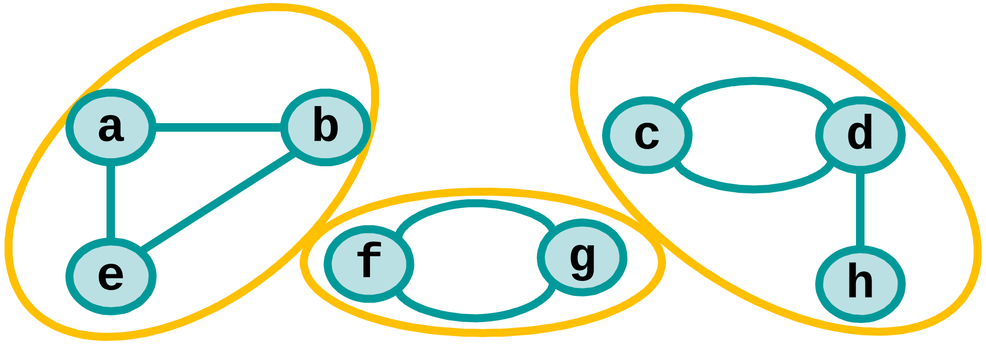
Loops

- Loops
  - A loop is defined as a path where
    - $v_0=v_k$, the starting and arrival vertices do coincide
  - Self-loop
    - Loops whose length is 1
  - A graphs without loops is called **acyclic**



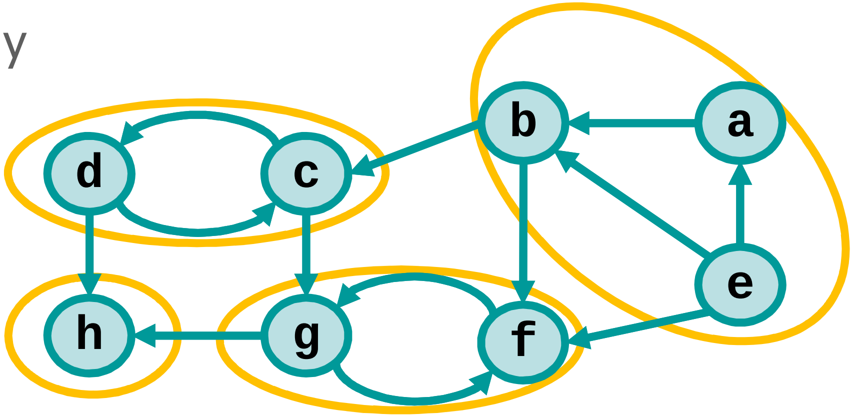
Connection in undirected graphs

- An undirected graph is said to be connected iff
  - $\forall v_i, v_j \in V$ there exists a path p such that $v_i \rightarrow_p v_j$
- In an undirected graph
  - Connected component
    - Maximal connected subgraph, that is, there is no superset including it which is connected
  - Connected undirected graph
    - Only one connected component



Connection in directed graphs

- A directed graph is said to be strongly connected iff
  - $\forall v_i, v_j \in V$ there exists two paths p, p′ such that $v_i \rightarrow_p v_j$ and $v_j \rightarrow_{p'} v_i$
- In a directed graph
  - Strongly connected component
    - Maximal strongly connected subgraph
  - Strongly connected directed graph
    - Only one strongly connected component



Dense/sparse graphs

- Given a graph
  - $G = (V, E)$

with

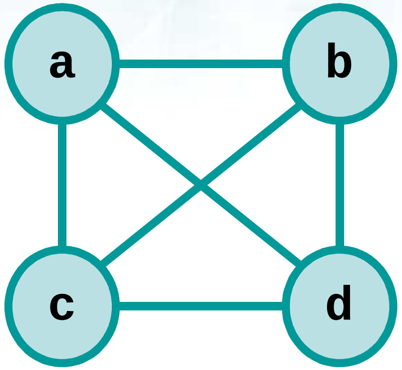
  - $|V|$ = cardinality of set V
  - $|E|$ = cardinality of set E
- We define
  - Dense graph
    - $|E| \cong |V|^2$

A lot of edges

  - Sparse graph
    - $|E| << |V|^2$

Few edges

Complete graph



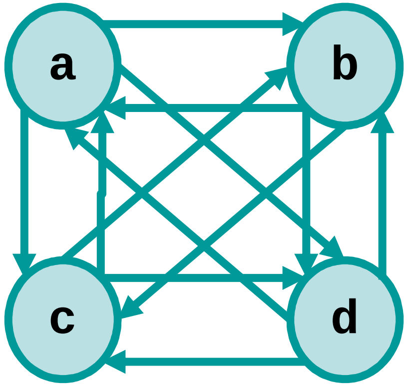
- Definition
  - $\forall v_i, v_j \in V \quad \exists (v_i, v_j) \in E$
- How many edges there are in a complete undirected graph?
  - |E| is given by the number of **combinations** of |V| elements taken 2 by 2
    - $|E| = \frac{|V|!}{(|V|-2)! \cdot 2!} = \frac{|V| \cdot (|V|-1) \cdot (|V|-2)!}{(|V|-2)! \cdot 2!} = \frac{|V| \cdot (|V|-1)}{2}$

Combinations:
Order does not
matter

Complete graph

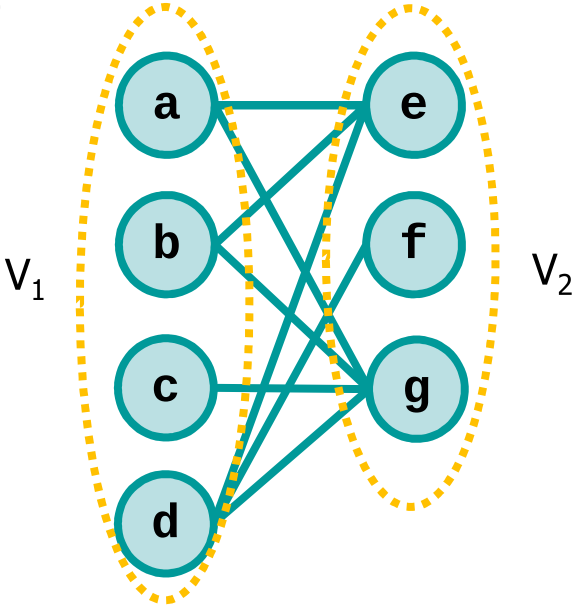
- How many edges there are in a complete directed graph?
  - |E| is the number of **dispositions** of |V| elements taken 2 by 2
    - $|E| = \frac{|V|!}{(|V|-2)!} = \frac{|V| \cdot (|V|-1) \cdot (|V|-2)!}{(|V|-2)!} = |V| \cdot (|V|-1)$

Dispositions: Order matters



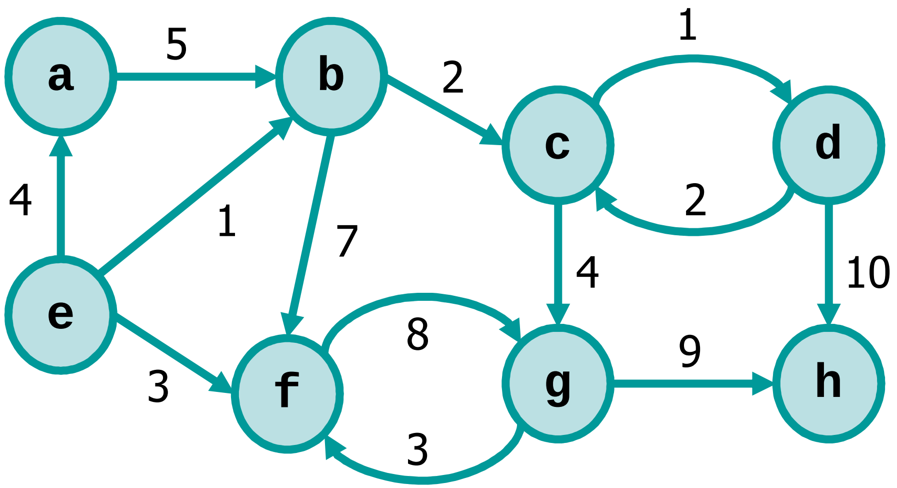
Bipartite graph

- Definition
  - Undirected graph where the V set may be partitioned in 2 subsets V_1 and V_2, such that
    - $\forall(v_i, v_j) \in E$ and $(v_i \in V_1$ and $v_j \in V_2)$ or $(v_j \in V_1$ and $v_i \in V_2)$

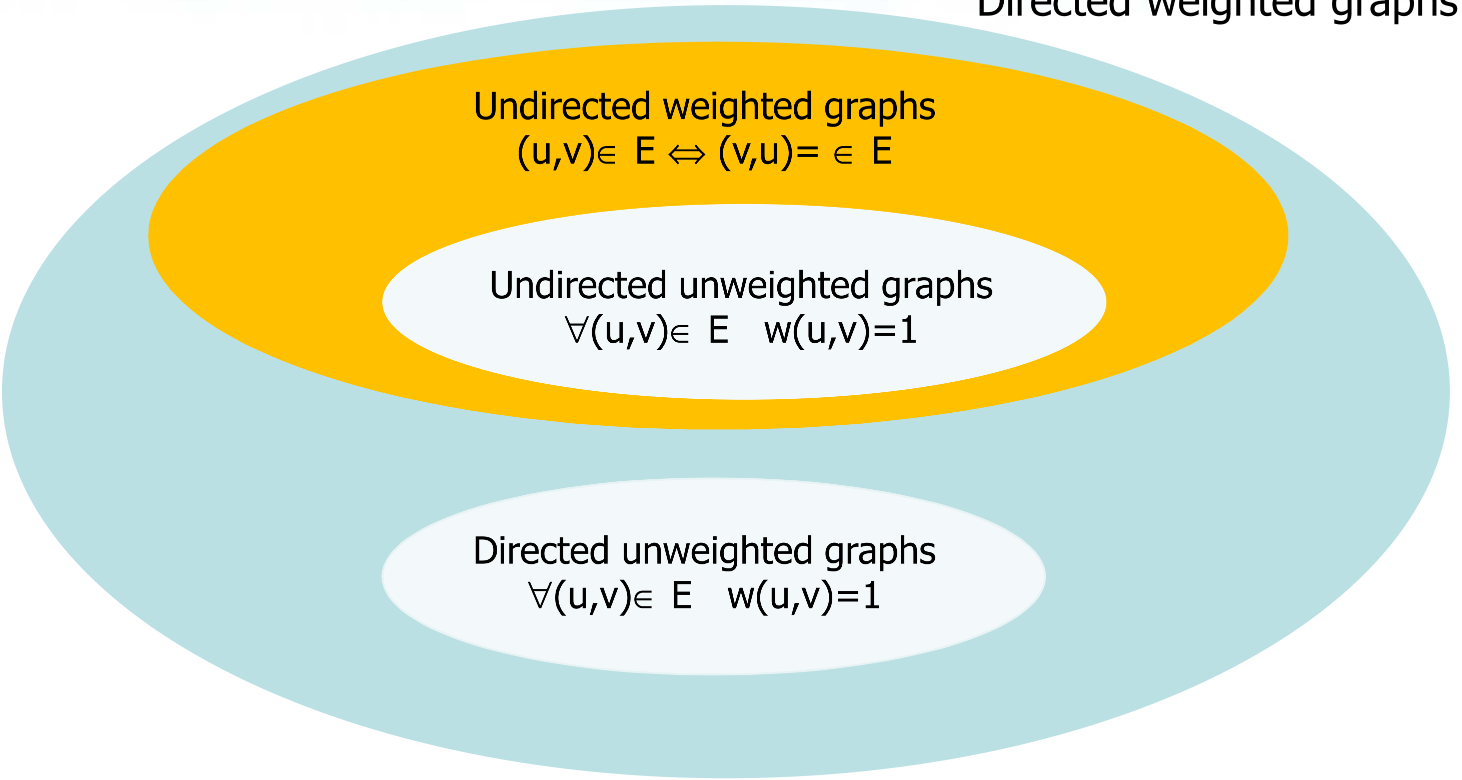


Weighted graph

- A weighted graph is a graph whose edges have a weight, i.e.,
  - ∃ w: E → R | w(u,v) = weight of edge (u, v)
  - In practice, weights may be integers, reals, positive or negative values, etc.



Types of Graphs



Directed weighted graphs

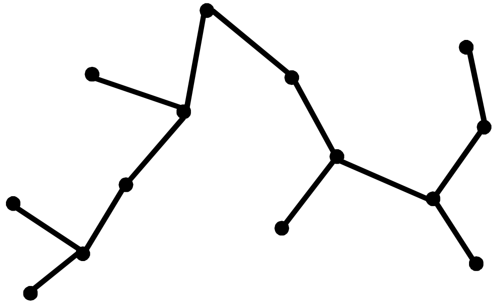
Undirected weighted graphs
$(u,v) \in E \Leftrightarrow (v,u) = \in E$

Undirected unweighted graphs
$\forall (u,v) \in E \quad w(u,v)=1$

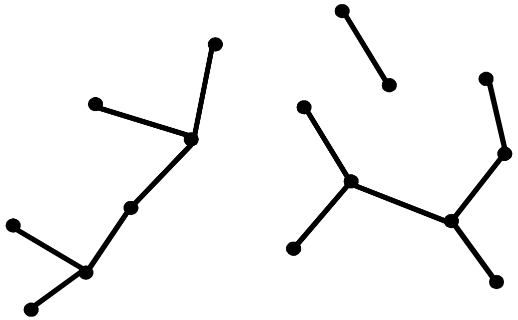
Directed unweighted graphs
$\forall (u,v) \in E \quad w(u,v)=1$

Non rooted trees

- A non rooted tree is an
  - Undirected, connected, acyclic graph



- A forest is a
  - Undirected acyclic graph

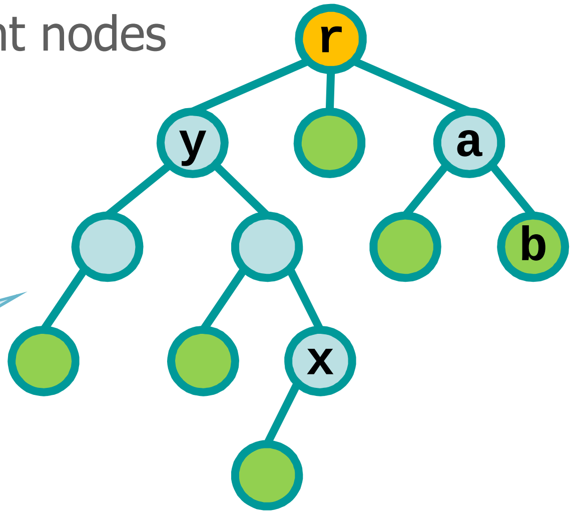


Properties of non rooted trees

- A non rooted tree G = (V, E) with |E| edges and |V| satisfies the following properties
  - Every pair of nodes is connected by a single simple path
  - G is connected
    - Removing an edge disconnects the graph
  - G connected and |E| = |V| - 1
  - G acyclic and |E| = |V| - 1
  - G acyclic
    - Adding an edge introduces a loop

Rooted trees

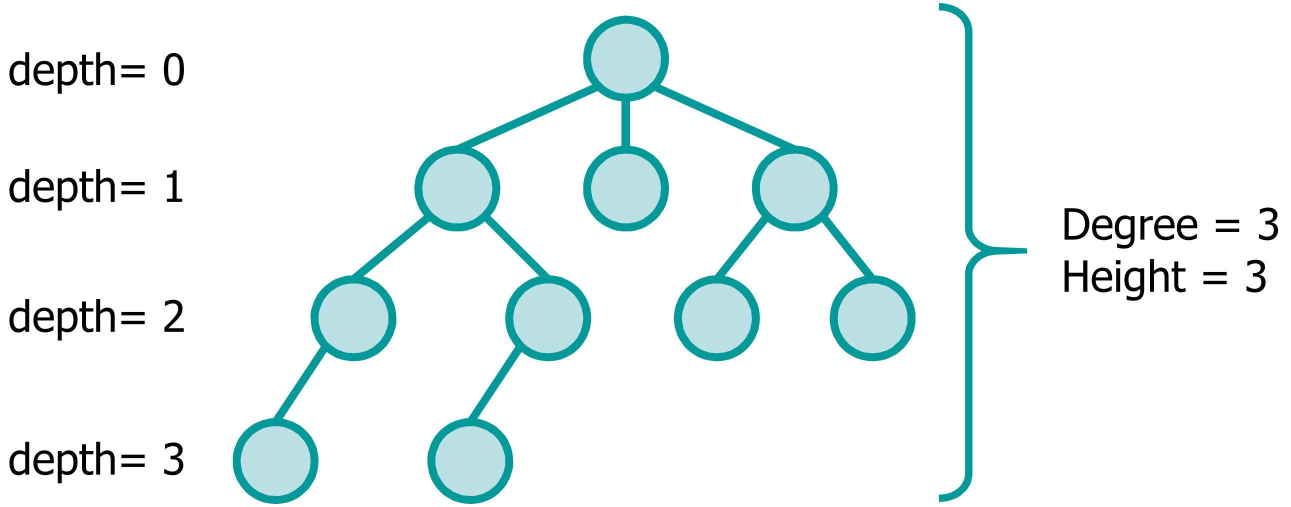
- A rooted tree is a tree where there is a node r called root
  - Parent/child relationship
    - y is an ancestor of x if y belongs to the path from r to x. In this case x is a descendant of y
    - y is a proper ancestor of x iff x ≠ y
    - Parent and a child are adjacent nodes
  - The root has no parent r
  - Leaves have no children ●



y ancestor of di x
x descendant of y
a parent of b
b child of a

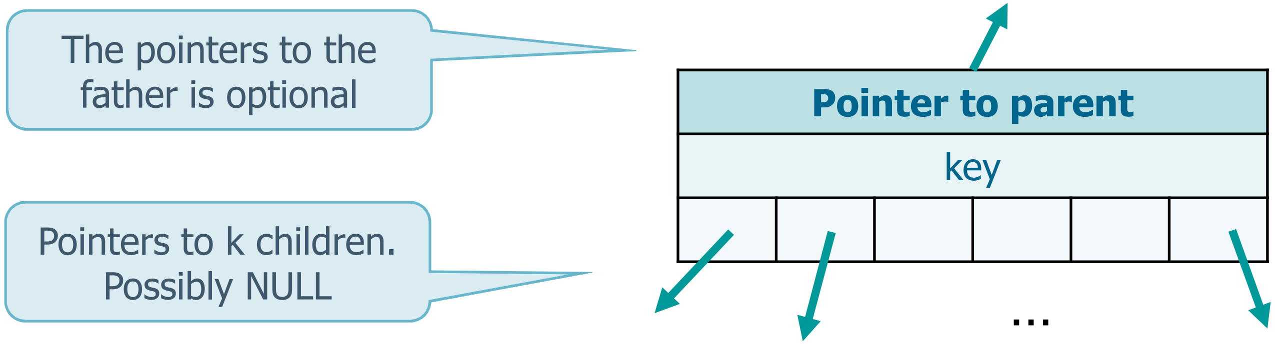
Properties of a rooted tree

- Given a rooted tree T the following are common definitions
  - Degree (T) = maximum number of children
  - Depth (x) = length of the path from the root to x
  - Height (T) = maximum depth of a node



Representation of a tree

- There are at least two representations for nodes of a tree of degree k
  - Each node may store a pointer to the parent, the key, and k pointers to k children

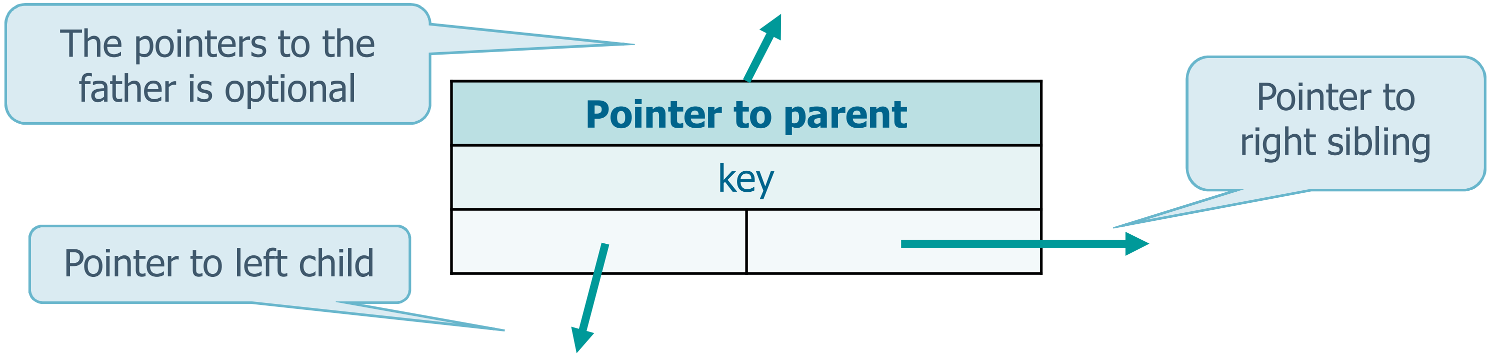

The pointers to the father is optional

Pointers to k children. Possibly NULL

  - Unefficient if only few nodes have indeed degree k
    - Space is allocated for all k pointers, but many are NULL)

Representation of a tree

- Each node may also store a pointer to parent, the key, 1 pointer to left child, 1 pointer to right sibling



The pointers to the father is optional

Pointer to parent

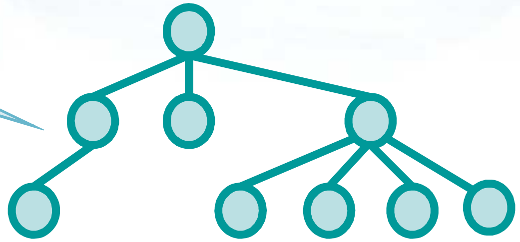
key

Pointer to right sibling

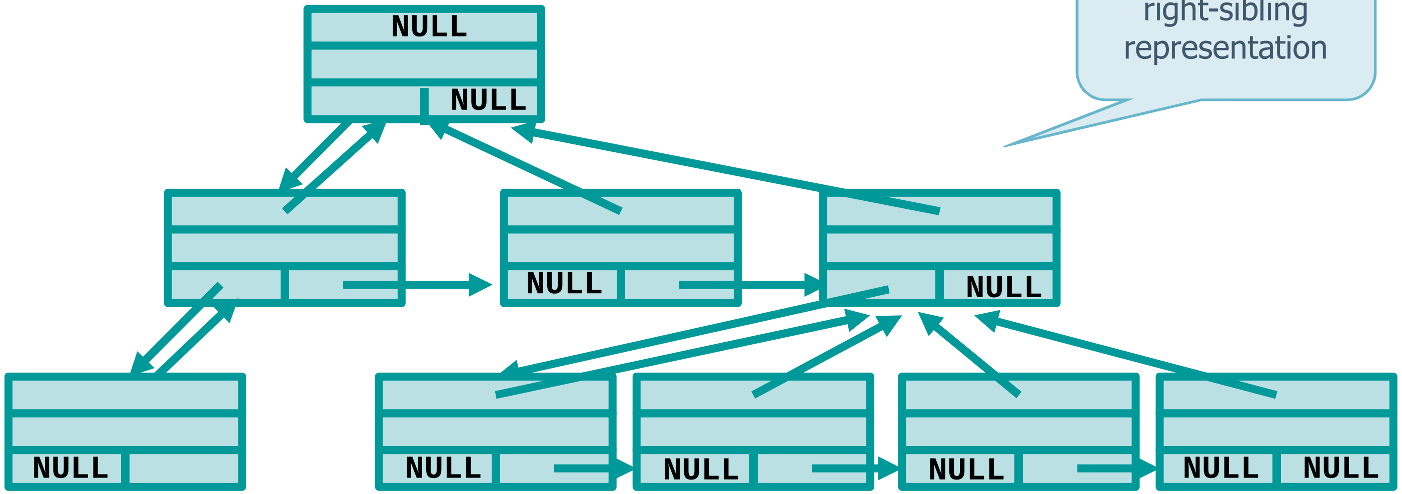
Pointer to left child

  - Efficient, as each node specifies always 2 pointers, no matter the degree of the tree

Representation of a tree



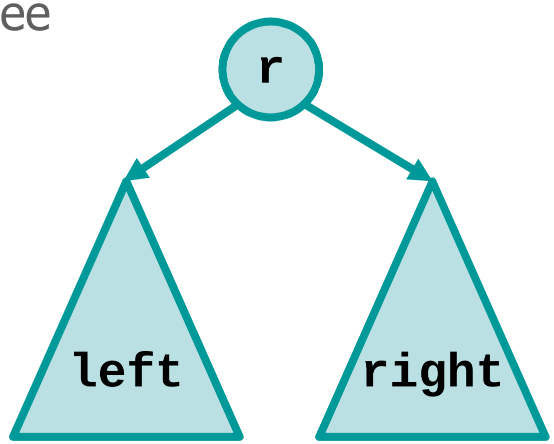
Standard representation



Left-child right-sibling representation

Binary trees

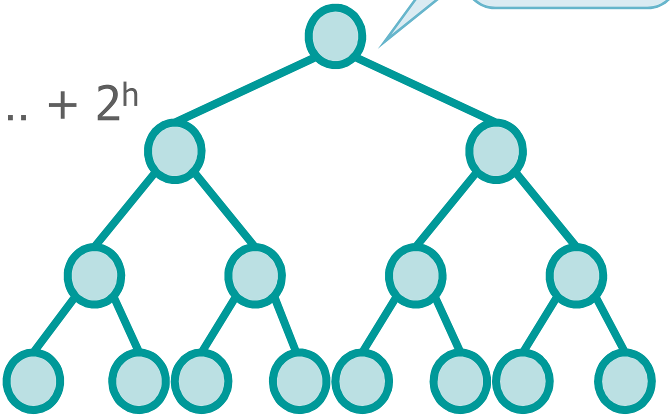
- Definition
  - Tree of degree 2
  - Recursively T is
    - Empty set of nodes
    - Root, left subtree, right subtree



Complete Binary Trees

- A complete binary tree must satisty two conditions
    - All leaves have the same depth
    - Every node is either a leaf or it has 2 children
- In a complete binary tree of height h
    - The number of leaves is 2^h
    - The number of nodes is
        - $\Sigma_{0 \leq i \leq h} 2^i = 2^0 + 2^1 + 2^2 \ldots + 2^h$
          $= 2^{h+1} - 1$

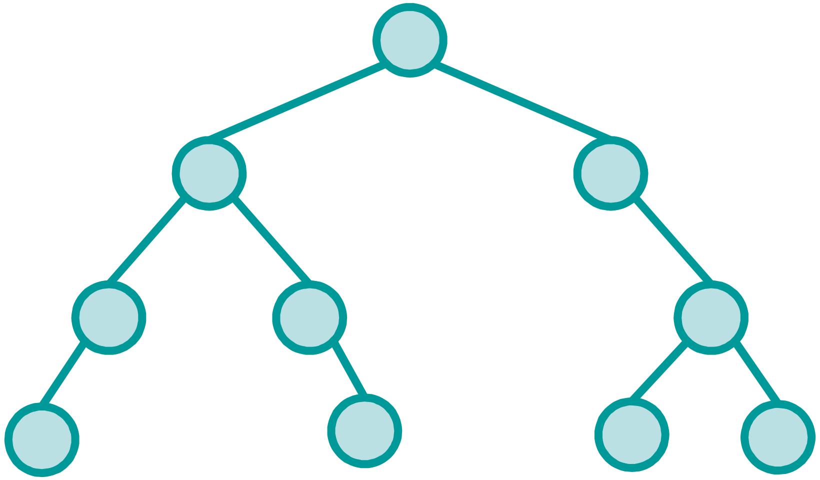
Finite geometric progression with ratio = 2



h = 3
8 leaves
15 nodes

Balanced binary trees

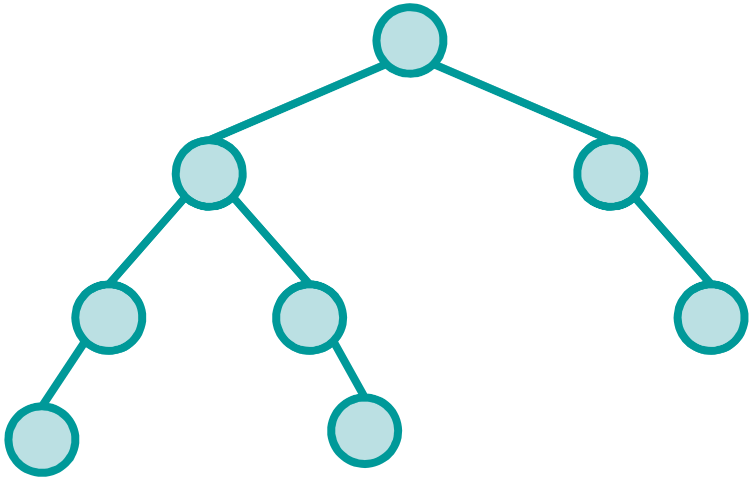
- In a balanced binary tree all paths root-leaves have the same length



  - If T is complete, then T is also balanced
  - The opposite is not necessarily true

Balanced binary trees

- A binary tree is said to be almost balanced if the length of all paths from root to leaves differs at most by 1



Linear Sequences

- A linear sequence is a finite set of consecutive elements
  - A unique index is associated to each element
    - $a_0, a_1, \dots, a_i, \dots, a_{n-1}$
  - A predecessor/successor relation is defined on couples of elements
    - $a_{i+1} = succ(a_i)$
    - $a_i = pred(a_{i+1})$
- A linear sequence can be stored using different underlying data structures
  - Array
  - List

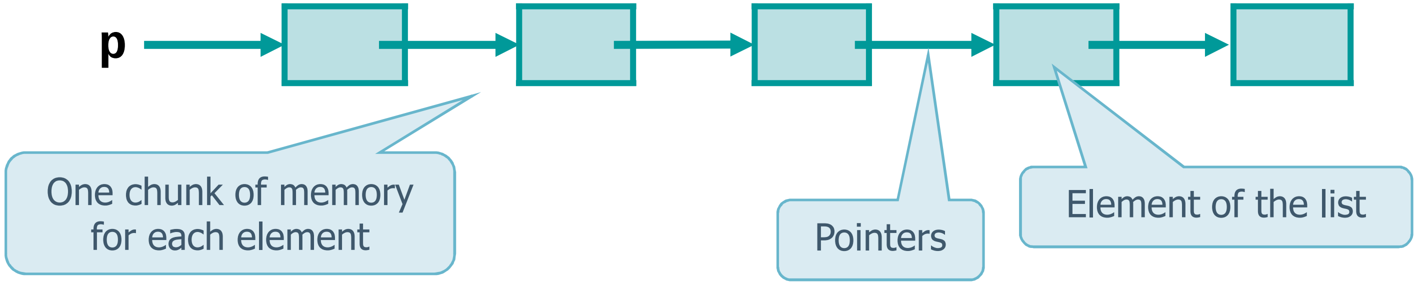
Linear Sequences

- An array stores element contiguously in memory
- Array enables direct access
  - Given index i, we access element a_i without any need for scanning the whole sequence
  - The cost of an access does not depend on the position of the element in the linear sequence, thus it is **O(1)**

One unique chunk of memory

Linear Sequences

- A list stores element non contiguously in memory
- List only allows sequential access
  - Given index i, we access element a_i scanning the linear sequence starting from one of its boundaries, usually the left one
  - The access cost depends on the position of the element in the linear sequence, thus it is **O(n)** in the worst case



Lists

- Operations on lists
  - Search
    - Scan the list looking for an element whose key field equals a given key
  - Insert an element
    - At the head of an unsorted list
    - At the tail of an unsorted list
    - At a position such as to guarantee that the invariance property of a sorted list is satisfied
  - Extract an element
    - From the head of an unsorted list
    - That has a field whose contents equals a deletion key (such an operation usually requires a search for the element to be deleted)

Lists

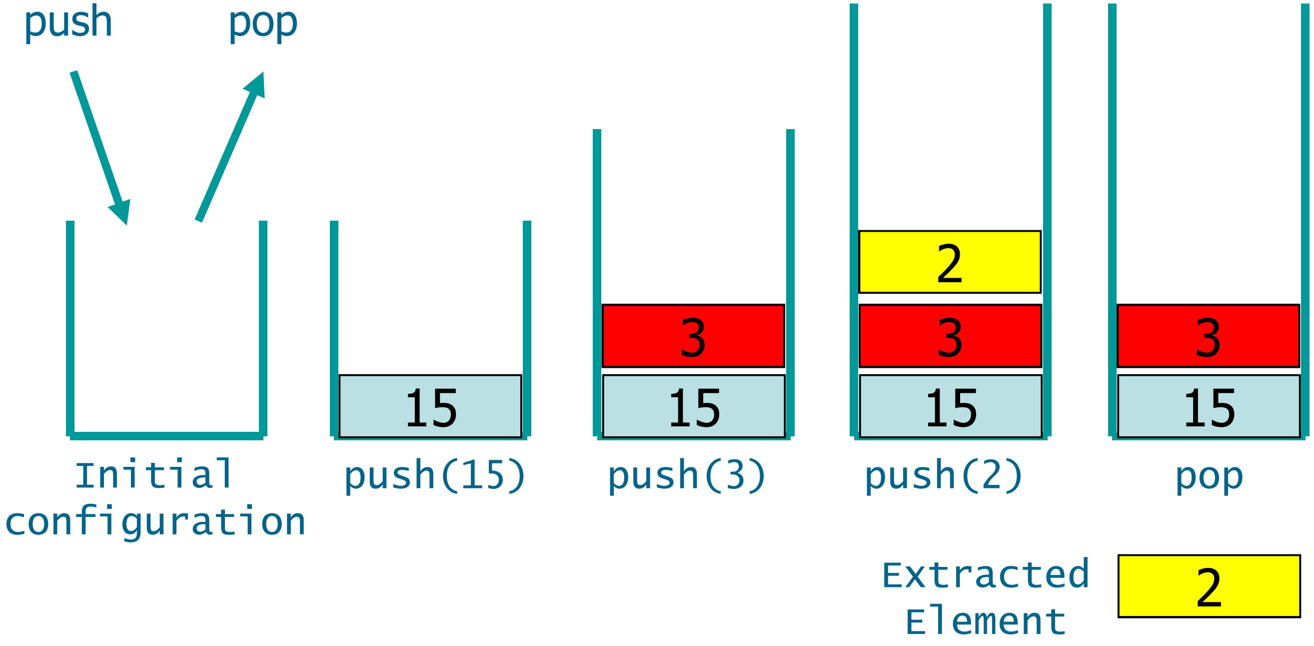
- Lists can be generalized into collections of data
- Data are inserted and deleted using different logics suited to obtained the desired result
- Among collections we recall
  - Stacks
  - Queues
  - Priority Queues

Stacks

- Criteria to extract elements
  - Extract the most-recently inserted element
    - **LIFO** policy: Last-In First-Out
    - Insertions are usually referred as push
    - Extractions are usually referred as pop
    - Pushes and pops are performend onto the structure head, usually referred as top of the stack or tos
    - The data structure is manipulated using an index (or a pointer) to the tos

Stacks

- A stack is usually represented as a pile of objects
  - A stack usually grows upward (towards smaller memory addresses)



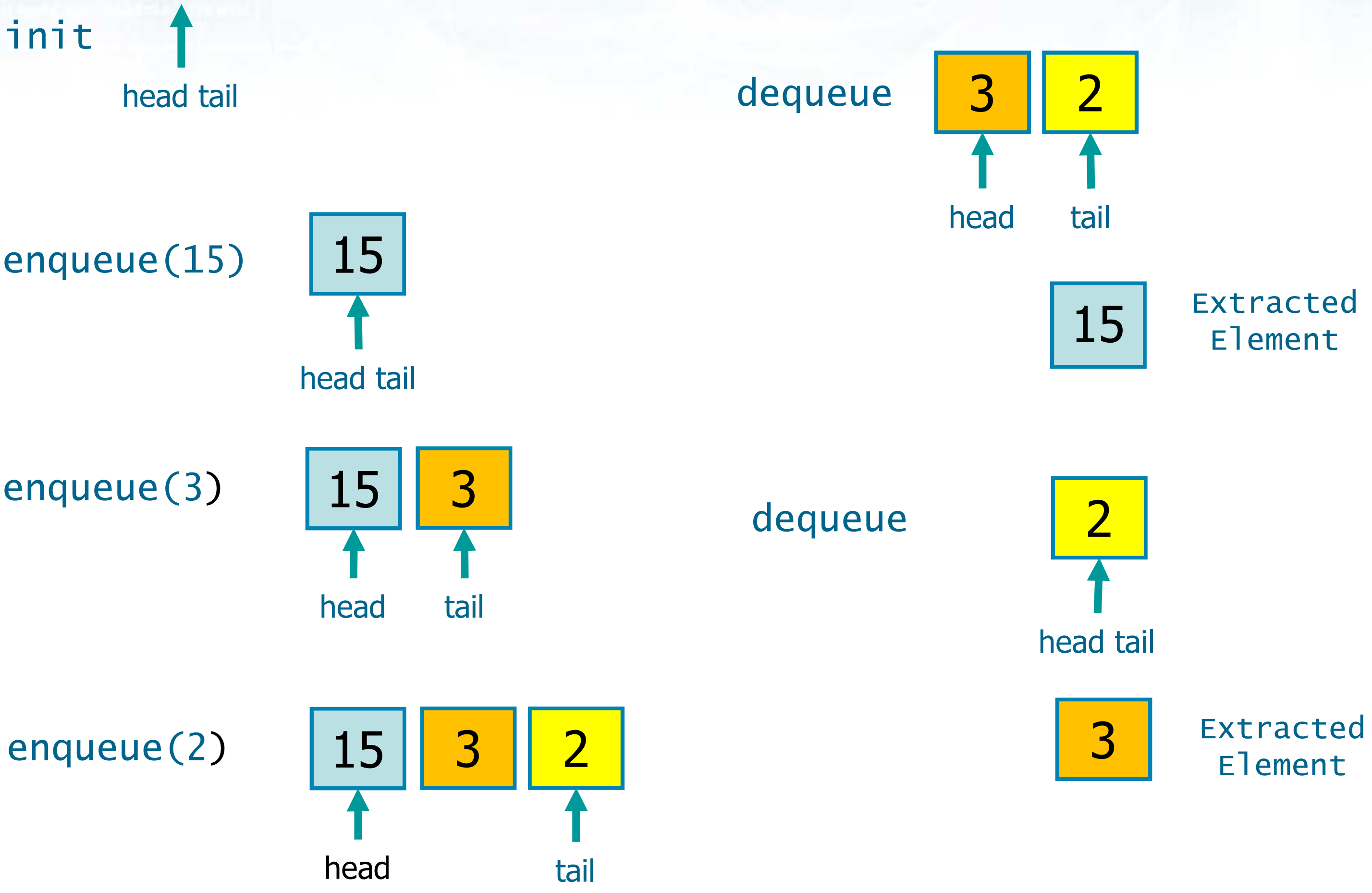
Queues

- Criteria to extract elements
  - Extract the least-recently inserted element
    - **FIFO** policy: First-In First-Out
    - Insertions are usually referred as enqueue
    - Extractions are usually referred as dequeue
    - Enqueues are performend onto the structure tail, usually referred as tail
    - Dequeues are performend onto the structure head, usually referred as head
    - The data structure is manipulated using two indexes (or pointers) to the head and tail

dequeue

enqueue

Queues



init

head tail

enqueue(15)

15

head tail

enqueue(3)

15 3

head tail

enqueue(2)

15 3 2

head tail

dequeue

3 2

head tail

15 Extracted Element

dequeue

2

head tail

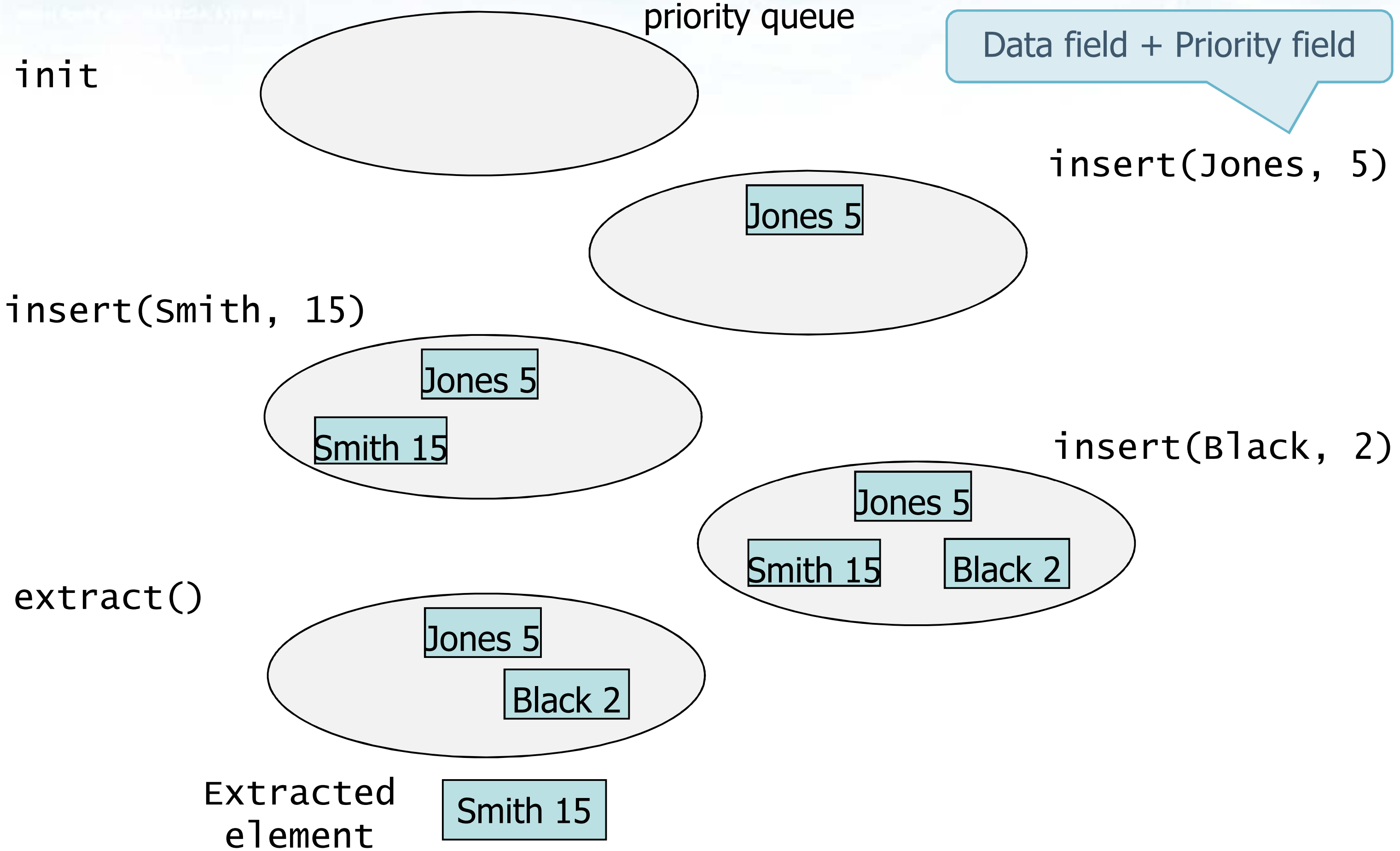
3 Extracted Element

Priority Queues

- Criteria to extract elements
  - Each element as an associated priority value
  - During each extraction, the highest (or lowest) priority element is extracted
    - The insertion logic and the used structure have to guarantees that property

Max (min) priority queue

Priority Queues



priority queue

Data field + Priority field

`init`

`insert(Jones, 5)`

Jones 5

`insert(Smith, 15)`

Jones 5

Smith 15

`insert(Black, 2)`

Jones 5

Smith 15

Black 2

`extract()`

Jones 5

Black 2

Extracted element

Smith 15