- Constant ≈ 1
- Logarithmic ≈ $\log n$
- Linear ≈ n
- N-Log-N ≈ $n \log n$
- Quadratic ≈ n^2
- Cubic ≈ n^3
- Exponential ≈ 2^n

Properties of powers:

$a^{(b+c)} = a^b a^c$

$a^{bc} = (a^b)^c$

$a^b / a^c = a^{(b-c)}$

$b = a^{\log_a b}$

$b^c = a^{c * \log_a b}$

Properties of logarithms:

$\log_b(xy) = \log_b x + \log_b y$

$\log_b(x/y) = \log_b x - \log_b y$

$\log_b xa = a\log_b x$

$\log_b a = \log_x a / \log_x b$

To perform the asymptotic analysis

- We find the worst-case number of primitive operations executed as a function of the input size
- We express this function with big-Oh notation

Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that $|f(n)| \leq c|g(n)|$ for $n \geq n_0$

Binary Search Runs in O(log n) time.

```
public static boolean binarySearch(int[ ] data, int target, int low, int high) {
  if (low > high)
    return false;                          // interval empty; no match
  else {
    int mid = (low + high) / 2;
    if (target == data[mid])
      return true;                         // found a match
    else if (target < data[mid])
      return binarySearch(data, target, low, mid − 1);  // recur left of the middle
    else
      return binarySearch(data, target, mid + 1, high); // recur right of the middle
  }
}
```

RECURSION → linear | binary | multiple
→ head | tail

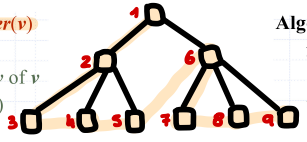
Tree ADT

Generic methods:
- integer size()
- boolean isEmpty()
- Iterator iterator()
- Iterable positions()

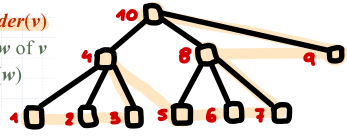
Accessor methods:
- position root()
- position parent(p)
- Iterable children(p)
- Integer numChildren(p)

Query methods:
- boolean isInternal(p)
- boolean isExternal(p)
- boolean isRoot(p)

Algorithm *preOrder(v)*
visit(v)
for each child *w* of *v*
preorder (w)



Algorithm *postOrder(v)*
for each child *w* of *v*
postOrder (w)
visit(v)



Binary Trees

def. 1 :
- Each internal node has at most two children (exactly two for proper binary trees)
- The children of a node are an ordered pair

def. 2 (recursive):
- a tree consisting of a single node, or
- a tree whose root has an ordered pair of children, each of which is a binary tree

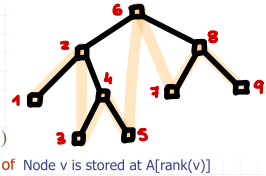
Additional methods:
- position left(p)
- position right(p)
- position sibling(p)

Properties:

- $e = i + 1$
- $n = 2e - 1$
- $h \leq i$
- $h \leq (n - 1)/2$
- $e \leq 2^h$
- $h \geq \log_2 e$
- $h \geq \log_2 (n + 1) - 1$

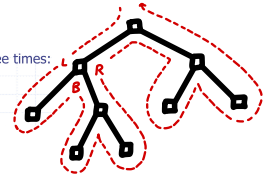
n number of nodes
e number of external nodes
i number of internal nodes
h height

Algorithm *inOrder(v)*
if *left* $(v) \neq$ **null**
inOrder (*left* (v))
visit(v)
if *right*$(v) \neq$ **null**
inOrder (*right* (v))



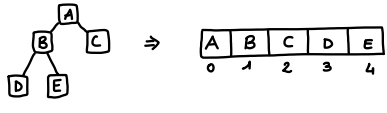
Euler Tour Traversal

Walk around the tree and visit each node three times:
- on the left (preorder)
- from below (inorder)
- on the right (postorder)



Array-Based Representation of Binary Trees

Node v is stored at A[rank(v)]
- rank(root) = 0
- if node is the left child of parent(node), rank(node) = 2 · rank(parent(node)) + 1
- if node is the right child of parent(node), rank(node) = 2 · rank(parent(node)) + 2



Priority Queue ADT

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Keys in a priority queue can be arbitrary objects on which an order is defined → total order relation $\leq$
  - Comparability property: either $x \leq y$ or $y \leq x$
  - Antisymmetric property: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

Main methods

- insert(k, v) inserts an entry with key k and value v
- removeMin() removes and returns the entry with smallest key

O(1) con lista NON ordinata
O(n) con lista ordinata
O(1) con lista ordinata
O(n) con lista NON ordinata

Additional methods

- min() returns, but does not remove, an entry with smallest key
- size(), isEmpty()

Entry ADT

- Methods:
  - getKey: returns the key for this entry
  - getValue: returns the value associated with this entry

Comparator ADT

- compare(x, y): returns an integer i such that
  - i < 0 if a < b,
  - i = 0 if a = b
  - i > 0 if a > b
  - An error occurs if a and b cannot be compared.

PQ Sorting

Algorithm *PQ-Sort(S, C)*

Input sequence S, comparator C for the elements of S
Output sequence S sorted in increasing order according to C
$P \leftarrow$ priority queue with comparator C
while $\neg S.isEmpty()$
  $e \leftarrow S.remove(S.first())$
  $P.insert(e, e)$
while $\neg P.isEmpty()$
  $e \leftarrow P.removeMin().getKey()$
  $S.addLast(e)$

→ il RUNNING TIME dipende dall'implementazione della PQ

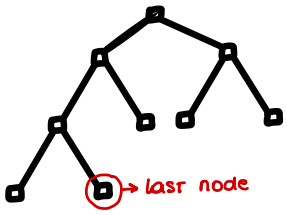
- PQ implementata con sequenza ordinata ⇒ Insertion-Sort $O(n^2)$
- PQ implementata con sequenza NON ordinata ⇒ Selection-Sort $O(n^2)$

Heaps

Binary tree + properties

- **Heap-Order:** for every internal node v other than the root, $key(v) \geq key(parent(v))$
- **Complete Binary Tree:** let h be the height of the heap
  - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
  - at depth $h-1$, the internal nodes are to the left of the external nodes

The last node of a heap is the rightmost node of maximum depth



h=0 ⇒ 2^0 = 1 node

h=1 ⇒ 2^1 = 2 nodes

h=2 ⇒ 2^2 = 4 nodes

Theorem: A heap storing n keys has height $O(\log n)$

Insertion into a Heap

1. TROVARE nuovo nodo per inserimento
2. INSERIRE valore
3. RISTABILIRE ordine con UPHEAP $O(\log n)$

The insertion node can be found by traversing a path of $O(\log n)$ nodes

- Go up until a left child or the root is reached
- If a left child is reached, go to the right child
- Go down left until a leaf is reached

: Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node

Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k

Removal from a Heap

↳ RemoveMin()

1. SCAMBIARE la key della radice con la key dell'ultimo nodo w
2. ELIMINARE w
3. RISTABILIRE ordine con DOWNHEAP $O(\log n)$

: Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root

downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k

Heap-Sort

sort a sequence of n elements in $O(n \log n)$

Bottom-up Heap Construction $O(n)$

- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with $2^i - 1$ keys are merged into heaps with $2^{i+1} - 1$ keys

Merge Sort $O(n \log n)$

Divide-and conquer is a general algorithm design paradigm:

- Divide: divide the input data S in two disjoint subsets S_1 and S_2
- Recur: solve the subproblems associated with S_1 and S_2
- Conquer: combine the solutions for S_1 and S_2 into a solution for S

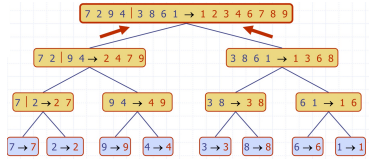
Merge-sort on an input sequence S with n elements consists of three steps:

- Divide: partition S into two sequences S_1 and S_2 of about $n/2$ elements each
- Recur: recursively sort S_1 and S_2
- Conquer: merge S_1 and S_2 into a unique sorted sequence

Algorithm *mergeSort*(S)
  Input sequence S with n elements
  Output sequence S sorted according to C
  if $S.size() > 1$
    $(S_1, S_2) \leftarrow partition(S, n/2)$
    mergeSort(S_1)
    mergeSort(S_2)
    $S \leftarrow merge(S_1, S_2)$

Algorithm *merge*(A, B)
  Input sequences A and B with $n/2$ elements each
  Output sorted sequence of $A \cup B$
  $S \leftarrow$ empty sequence
  while $\neg A.isEmpty() \wedge \neg B.isEmpty()$
    if $A.first().element() < B.first().element()$
      $S.addLast(A.remove(A.first()))$
    else
      $S.addLast(B.remove(B.first()))$
  while $\neg A.isEmpty()$
    $S.addLast(A.remove(A.first()))$
  while $\neg B.isEmpty()$
    $S.addLast(B.remove(B.first()))$
  return S

Merge-Sort Tree

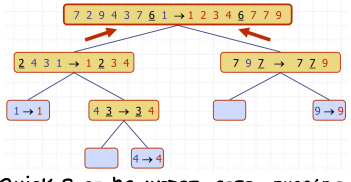


Quick-Sort $O(n^2)$ / $O(n \log n)$ exp.

- Divide: pick a random element x (called pivot) and partition S into
  - L elements less than x
  - E elements equal x
  - G elements greater than x
- Recur: sort L and G
- Conquer: join L, E and G

Algorithm *partition*(S, p)
  Input sequence S, position p of pivot
  Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
  $L, E, G \leftarrow$ empty sequences
  $x \leftarrow S.remove(p)$
  while $\neg S.isEmpty()$
    $y \leftarrow S.remove(S.first())$
    if $y < x$
      $L.addLast(y)$
    else if $y = x$
      $E.addLast(y)$
    else { $y > x$ }
      $G.addLast(y)$
  return L, E, G

Quick-Sort Tree



Quick-Sort ha worst-case running time $O(n^2)$ ma expected running time $O(n \log n)$ su una "Good Call", cioè una chiamata in cui L e G hanno dimensioni $< \frac{3}{4}s$ (s sequence size)

In-Place Quick-Sort

Algorithm *inPlaceQuickSort*(S, l, r)
  Input sequence S, ranks l and r
  Output sequence S with the elements of rank between l and r rearranged in increasing order
  if $l \geq r$
    return
  $i \leftarrow$ a random integer between l and r
  $x \leftarrow S.elemAtRank(i)$
  $(h, k) \leftarrow inPlacePartition(x)$
  inPlaceQuickSort(S, l, $h-1$)
  inPlaceQuickSort(S, $k+1$, r)

Bucket-Sort $O(n+N)$

Let be S be a sequence of n (key, element) items with keys in the range $[0, N-1]$

Bucket-sort uses the keys as indices into an auxiliary array B of sequences (buckets)

Phase 1: Empty sequence S by moving each entry (k, o) into its bucket $B[k]$ → $O(n)$

Phase 2: For $i = 0, \ldots, N-1$, move the entries of bucket $B[i]$ to the end of sequence S → $O(n+N)$

Algorithm bucketSort(S):
Input: Sequence S of entries with integer keys in the range [0, N − 1]
Output: Sequence S sorted in nondecreasing order of the keys
let B be an array of N sequences, each of which is initially empty
for each entry e in S **do**
  k = the key of e
  remove e from S
  insert e at the end of bucket B[k]
for i = 0 to N−1 **do**
  for each entry e in B[i] **do**
    remove e from B[i]
    insert e at the end of S

Lexicographic-Sort $O(dT(n))$

Let C_i be the comparator that compares two tuples by their i-th dimension

Let *stableSort*(S, C) be a stable sorting algorithm that uses comparator C

Lexicographic-sort sorts a sequence of d-tuples in lexicographic order by executing d times algorithm *stableSort*, one per dimension

Lexicographic-sort runs in $O(dT(n))$ time, where $T(n)$ is the running time of *stableSort*

Algorithm *lexicographicSort*(S)
  Input sequence S of d-tuples
  Output sequence S sorted in lexicographic order
  for $i \leftarrow d$ **downto** 1
    stableSort(S, C_i)

Example:

(7,4,6) (5,1,5) (2,4,6) (2, 1, 4) (3, 2, 4)

(2, 1, 4) (3, 2, 4) (5,1,5) (7,4,6) (2,4,6)

(2, 1, 4) (5,1,5) (3, 2, 4) (7,4,6) (2,4,6)

(2, 1, 4) (2,4,6) (3, 2, 4) (5,1,5) (7,4,6)

Radix-Sort $O(d(n+N))$

Radix-sort is a specialization of lexicographic-sort that uses bucket-sort as the stable sorting algorithm in each dimension

Radix-sort is applicable to tuples where the keys in each dimension i are integers in the range $[0, N-1]$

Algorithm *radixSort*(S, N)
  Input sequence S of d-tuples such that $(0, \ldots, 0) \leq (x_1, \ldots, x_d)$ and $(x_1, \ldots, x_d) \leq (N-1, \ldots, N-1)$ for each tuple $(x_1, \ldots, x_d)$ in S
  Output sequence S sorted in lexicographic order
  for $i \leftarrow d$ **downto** 1
    bucketSort(S, N)

Radix-Sort for Binary Numbers $O(bn)$

Consider a sequence of n b-bit integers

$x = x_{b-1} \ldots x_1 x_0$

We represent each element as a b-tuple of integers in the range $[0, 1]$ and apply radix-sort with $N = 2$

Algorithm *binaryRadixSort*(S)
  Input sequence S of b-bit integers
  Output sequence S sorted
  replace each element x of S with the item $(0, x)$
  for $i \leftarrow 0$ **to** $b-1$
    replace the key k of each item (k, x) of S with bit x_i of x
    bucketSort(S, 2)

Maps

A map models a searchable collection of key-value entries. Multiple entries with the same key are not allowed

The main operations of a map are for searching, inserting, and deleting items

Map ADT

- get(k): if the map M has an entry with key k, return its associated value; else, return null → O(n) con list based map
- put(k, v): insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k → O(1) con list based map
- remove(k): if the map M has an entry with key k, remove it from M and return its associated value; else, return null → O(n) con list based map
- size(), isEmpty()
- entrySet(): return an iterable collection of the entries in M
- keySet(): return an iterable collection of the keys in M
- values(): return an iterator of the values in M

Hash Functions and Hash Tables

A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$ → ex. $h(x) = x \bmod N$ (HASH VALUE of x)

A hash function is usually specified as the composition of two functions:

Hash code: → applied 1st
h_1: keys → integers

Compression function: → applied 2nd
h_2: integers → $[0, N-1]$

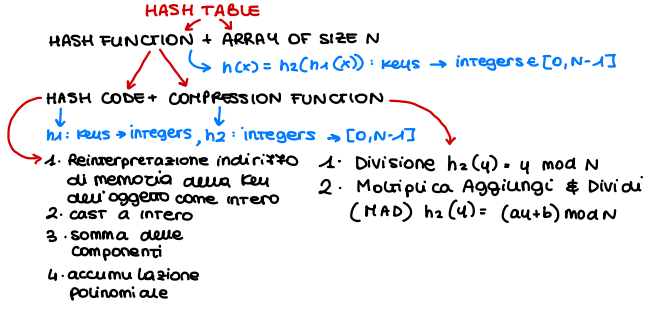
A hash table for a given key type consists of

- Hash function h
- Array (called table) of size N

When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Collision occur when different elements are mapped to the same cell

Separate Chaining: let each cell in the table point to a linked list of entries that map there



Open addressing: the colliding item is placed in a different cell of the table

Linear probing: handles collisions by placing the colliding item in the next (circularly) available table cell

Each table cell inspected is referred to as a "probe"

Colliding items lump together, causing future collisions to cause a longer sequence of probes

CON LINEAR PROBING:

1. get(k) O(n)

- We start at cell $h(k)$
- We probe consecutive locations until one of the following occurs
  - An item with key k is found, or
  - An empty cell is found, or
  - N cells have been unsuccessfully probed

Algorithm *get(k)*
$i \leftarrow h(k)$
$p \leftarrow 0$
repeat
  $c \leftarrow A[i]$
  if $c = \emptyset$
    return *null*
  else if *c.getKey*() $= k$
    return *c.getValue*()
  else
    $i \leftarrow (i + 1) \bmod N$
    $p \leftarrow p + 1$
until $p = N$
return *null*

2. remove(k) O(n)

- We search for an entry with key k
- If such an entry (k, o) is found, we replace it with the special item $DEFUNCT$ and we return element o
- Else, we return *null*

3. put(k, o) O(n)

- We throw an exception if the table is full
- We start at cell $h(k)$
- We probe consecutive cells until one of the following occurs
  - A cell i is found that is either empty or stores $DEFUNCT$, or
  - N cells have been unsuccessfully probed
- We store (k, o) in cell i

Double Hashing

Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series

$(i + jd(k)) \bmod N$ for $j = 0, 1, \ldots, N-1$

The secondary hash function $d(k)$ cannot have zero values

The table size N must be a prime to allow probing of all the cells

Common choice of compression function for the secondary hash function:

$d_2(k) = q - k \bmod q$

where

- $q < N$
- q is a prime

The possible values for $d_2(k)$ are $1, 2, \ldots, q$

EXPECTED RUNNING TIME OF ALL ADT OPERATIONS IS O(1)
(VERY FAST PROVIDED THAT THE LOAD FACTOR IS NOT CLOSE TO 100%)

Binary Search Trees

A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:

- Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$

External nodes do not store items

Search O(h), h : height

Algorithm *TreeSearch*(k, v)
- **if** *T.isExternal* (v)
  - **return** v
- **if** $k < key(v)$
  - **return** *TreeSearch*$(k, left(v))$
- **else if** $k = key(v)$
  - **return** v
- **else** { $k > key(v)$ }
  - **return** *TreeSearch*$(k, right(v))$

si inizia dalla root poi si scende in base al confronto con k. se si arriva a una foglia la key non è stata trovata

→ visita in order per visitare in ordine crescente

Insertion O(h)

To perform operation put(k, o), we search for key k (using TreeSearch)

Assume k is not already in the tree, and let w be the leaf reached by the search

We insert k at node w and expand w into an internal node

Deletion O(h)

To perform operation remove(k), we search for key k

Assume key k is in the tree, and let v be the node storing k

If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent

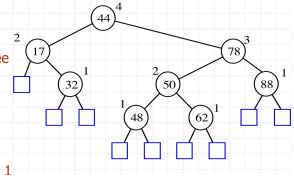
We consider the case where the key k to be removed is stored at a node v whose children are both internal

- we find the internal node w that follows v in an inorder traversal
- we copy $key(w)$ into node v
- we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)

AVL Trees

AVL trees are balanced

An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1



Insertion O(log n)

Insertion is as in a binary search tree

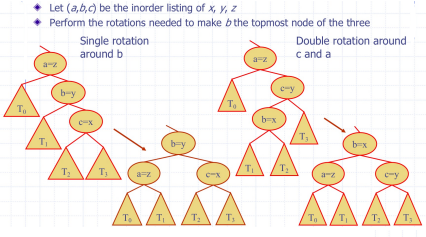
Always done by expanding an external node.

Removal O(log n)

Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.

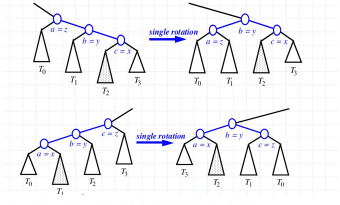
Trinode Restructuring

- Let (a,b,c) be the inorder listing of x, y, z
- Perform the rotations needed to make b the topmost node of the three



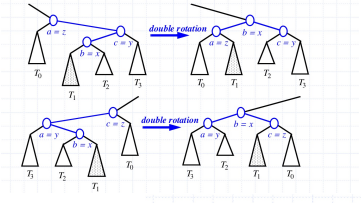
Restructuring (as Single Rotations)

- Single Rotations:



Restructuring (as Double Rotations)

- double rotations:

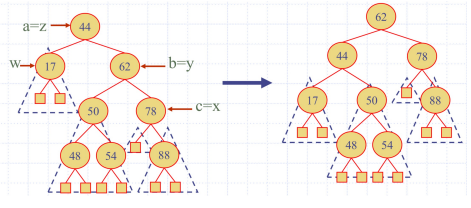


Rebalancing after a Removal

Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height

We perform a trinode restructuring to restore balance at z

As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



AVL tree storing n items

- The data structure uses O(n) space
- A single restructuring takes O(1) time
  - using a linked-structure binary tree
- Searching takes O(log n) time
  - height of tree is O(log n), no restructures needed
- Insertion takes O(log n) time
  - initial find is O(log n)
  - restructuring up the tree, maintaining heights is O(log n)
- Removal takes O(log n) time
  - initial find is O(log n)
  - restructuring up the tree, maintaining heights is O(log n)

Graphs

A graph is a pair (V, E), where

- V is a set of nodes, called vertices
- E is a collection of pairs of vertices, called edges
- Vertices and edges are positions and store elements

Directed edge
- ordered pair of vertices (u,v)
- first vertex u is the origin
- second vertex v is the destination
- e.g., a flight

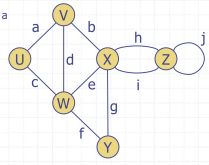
Undirected edge
- unordered pair of vertices (u,v)
- e.g., a flight route

⇒ Directed graph → DIGRAPH
- all the edges are directed
- e.g., route network

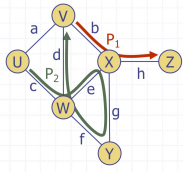
⇒ Undirected graph
- all the edges are undirected
- e.g., flight network

Terminology

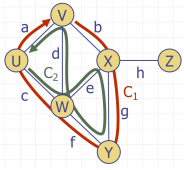
- End vertices (or endpoints) of an edge
  - U and V are the endpoints of a
- Edges incident on a vertex
  - a, d, and b are incident on V
- Adjacent vertices
  - U and V are adjacent
- Degree of a vertex
  - X has degree 5
- Parallel edges
  - h and i are parallel edges
- Self-loop
  - j is a self-loop



- Path
  - sequence of alternating vertices and edges
  - begins with a vertex
  - ends with a vertex
  - each edge is preceded and followed by its endpoints
- Simple path
  - path such that all its vertices and edges are distinct
- Examples
  - P_1=(V,b,X,h,Z) is a simple path
  - P_2=(U,c,W,e,X,g,Y,f,W,d,V) is a path that is not simple



- Cycle
  - circular sequence of alternating vertices and edges
  - each edge is preceded and followed by its endpoints
- Simple cycle
  - cycle such that all its vertices and edges are distinct
- Examples
  - C_1=(V,b,X,g,Y,f,W,c,U,a,-) is a simple cycle
  - C_2=(U,c,W,e,X,g,Y,f,W,d,V,a,-) is a cycle that is not simple



Properties

1. $\sum_v \deg(v) = 2m$ → sommatoria dei deg di tutti i vertici
2. In an undirected graph with no self-loops and no multiple edges $m \leq n\,(n-1)/2$

con n = numero vertici
m = numero lati
deg(v) = grado del vertice v

vertici e lati sono implementati attraverso 2 tipi di dato: Vertex (che memorizza un elemento inserito da utente) e Edge (che memorizza un elemento associato, richiamato col metodo element()). Il grafo è modellato con il tipo di dato Graph.

Graph ADT

- numVertices(): Returns the number of vertices of the graph.
- vertices(): Returns an iteration of all the vertices of the graph.
- numEdges(): Returns the number of edges of the graph.
- edges(): Returns an iteration of all the edges of the graph.
- getEdge(u, v): Returns the edge from vertex u to vertex v, if one exists; otherwise return null. For an undirected graph, there is no difference between getEdge(u, v) and getEdge(v, u).
- endVertices(e): Returns an array containing the two endpoint vertices of edge e. If the graph is directed, the first vertex is the origin and the second is the destination.
- opposite(v, e): For edge e incident to vertex v, returns the other vertex of the edge; an error occurs if e is not incident to v.
- outDegree(v): Returns the number of outgoing edges from vertex v.
- inDegree(v): Returns the number of incoming edges to vertex v. For an undirected graph, this returns the same value as does outDegree(v).
- outgoingEdges(v): Returns an iteration of all outgoing edges from vertex v.
- incomingEdges(v): Returns an iteration of all incoming edges to vertex v. For an undirected graph, this returns the same collection as does outgoingEdges(v).
- insertVertex(x): Creates and returns a new Vertex storing element x.
- insertEdge(u, v, x): Creates and returns a new Edge from vertex u to vertex v, storing element x; an error occurs if there already exists an edge from u to v.
- removeVertex(v): Removes vertex v and all its incident edges from the graph.
- removeEdge(e): Removes edge e from the graph.

performance based on data structure

• n vertices, m edges • no parallel edges • no self-loops	Edge List	Adjacency List	Adjacency Matrix
Space	$n+m$	$n+m$	n^2
incidentEdges(v)	m	deg(v)	n
areAdjacent (v, w)	m	min(deg(v), deg(w))	1
insertVertex(o)	1	1	n^2
insertEdge(v, w, o)	1	1	1
removeVertex(v)	m	deg(v)	n^2
removeEdge(e)	1	1	1

Subgraphs

SOTTOGRAFO S di G: (V',E') tc V'⊆V, E' ⊆ E

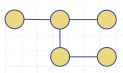
SOTTOGRAFO RICOPRENTE (SPANNING) S di G: contiene tutti i vertici di G

Connectivity

- A graph is connected if there is a path between every pair of vertices
- A connected component of a graph G is a maximal connected subgraph of G

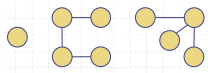
Trees and Forests

- A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycles



Tree

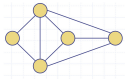
- A forest is an undirected graph without cycles
- The connected components of a forest are trees



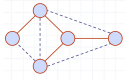
Forest

Spanning Trees and Forests

- A spanning tree of a connected graph is a spanning subgraph that is a tree
- A spanning tree is not unique unless the graph is a tree
- A spanning forest of a graph is a spanning subgraph that is a forest



Graph



Spanning tree

Depth-First Search O(n+m)

- Depth-first search (DFS) is a general technique for traversing a graph
- A DFS traversal of a graph G
  - Visits all the vertices and edges of G
  - Determines whether G is connected
  - Computes the connected components of G
  - Computes a spanning forest of G
- DFS on a graph with n vertices and m edges takes $O(n + m)$ time
- DFS can be further extended to solve other graph problems
  - Find and report a path between two given vertices
  - Find a cycle in the graph
- Depth-first search is to graphs what Euler tour is to binary trees

Algorithm DFS(G, u):
Input: A graph G and a vertex u of G
Output: A collection of vertices reachable from u, with their discovery edges
Mark vertex u as visited.
for each of u's outgoing edges, $e = (u, v)$ **do**
  if vertex v has not been visited **then**
    Record edge e as the discovery edge for vertex v.
    Recursively call DFS(G, v).

Properties of DFS

1. *DFS*(G, v) visits all the vertices and edges in the connected component of v
2. The discovery edges labeled by *DFS*(G, v) form a spanning tree of the connected component of v

Path Finding

We call *DFS*(G, u) with u as the start vertex

We use a stack S to keep track of the path between the start vertex and the current vertex

As soon as destination vertex z is encountered, we return the path as the contents of the stack

```
Algorithm pathDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  if v = z
    return S.elements()
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        S.push(e)
        pathDFS(G, w, z)
        S.pop(e)
      else
        setLabel(e, BACK)
  S.pop(v)
```

Cycle Finding

We use a stack S to keep track of the path between the start vertex and the current vertex

As soon as a back edge (v, w) is encountered, we return the cycle as the portion of the stack from the top to vertex w

Trova i simple cycles

```
Algorithm cycleDFS(G, v, z)
  setLabel(v, VISITED)
  S.push(v)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      S.push(e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        pathDFS(G, w, z)
        S.pop(e)
      else
        T ← new empty stack
        repeat
          o ← S.pop()
          T.push(o)
        until o = w
        return T.elements()
  S.pop(v)
```

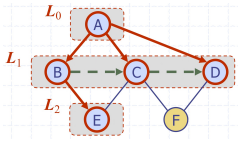
DFS for an Entire Graph

```
Algorithm DFS(G)
  Input graph G
  Output labeling of the edges of G
      as discovery edges and
      back edges
  for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
  for all e ∈ G.edges()
    setLabel(e, UNEXPLORED)
  for all v ∈ G.vertices()
    if getLabel(v) = UNEXPLORED
      DFS(G, v)
```

```
Algorithm DFS(G, v)
  Input graph G and a start vertex v of G
  Output labeling of the edges of G
      in the connected component of v
      as discovery edges and back edges
  setLabel(v, VISITED)
  for all e ∈ G.incidentEdges(v)
    if getLabel(e) = UNEXPLORED
      w ← opposite(v,e)
      if getLabel(w) = UNEXPLORED
        setLabel(e, DISCOVERY)
        DFS(G, w)
      else
        setLabel(e, BACK)
```

Breadth-First Search $O(n+m)$

BFS on a graph with n vertices and m edges takes $O(n+m)$ time



Algorithm *BFS*(*G*)
  Input graph *G*
  Output labeling of the edges and partition of the vertices of *G*
  for all *u* ∈ *G.vertices*()
    setLabel(*u*, *UNEXPLORED*)
  for all *e* ∈ *G.edges*()
    setLabel(*e*, *UNEXPLORED*)
  for all *v* ∈ *G.vertices*()
    if *getLabel*(*v*) = *UNEXPLORED*
      BFS(*G*, *v*)

Algorithm *BFS*(*G*, *s*)
  L_0 ← new empty sequence
  L_0.*addLast*(*s*)
  setLabel(*s*, *VISITED*)
  i ← 0
  while ¬L_i.*isEmpty*()
    L_{i+1} ← new empty sequence
    for all *v* ∈ L_i.*elements*()
      for all *e* ∈ *G.incidentEdges*(*v*)
        if *getLabel*(*e*) = *UNEXPLORED*
          w ← *opposite*(*v*,*e*)
          if *getLabel*(*w*) = *UNEXPLORED*
            setLabel(*e*, *DISCOVERY*)
            setLabel(*w*, *VISITED*)
            L_{i+1}.*addLast*(*w*)
          else
            setLabel(*e*, *CROSS*)
    i ← *i* +1

Properties

Notation

G_s: connected component of s

Property 1

BFS(*G*, *s*) visits all the vertices and edges of G_s

Property 2

The discovery edges labeled by *BFS*(*G*, *s*) form a spanning tree T_s of G_s

Property 3

For each vertex v in L_i

- The path of T_s from s to v has i edges
- Every path from s to v in G_s has at least i edges

DFS vs. BFS

Applications	DFS	BFS
Spanning forest, connected components, paths, cycles	√	√
Shortest paths		√
Biconnected components	√	

Digraphs

If G is simple, $m \leq n \cdot (n-1)$

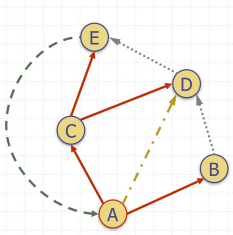
Directed DFS

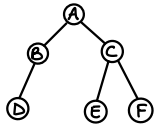
We can specialize the traversal algorithms (DFS and BFS) to digraphs by traversing edges only along their direction

In the directed DFS algorithm, we have four types of edges

- discovery edges
- back edges
- forward edges
- cross edges

A directed DFS starting at a vertex s determines the vertices reachable from s





DFS: A-B-D-C-E-F

Strong Connectivity

Each vertex can reach all other vertices

Strong Connectivity Algorithm

Pick a vertex v in G

Perform a DFS from v in G

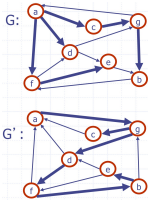
- If there's a w not visited, print "no"

Let G' be G with edges reversed

Perform a DFS from v in G'

- If there's a w not visited, print "no"
- Else, print "yes"

Running time: O(n+m)

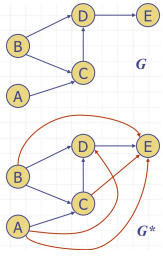


Transitive Closure

Given a digraph G, the transitive closure of G is the digraph G^* such that

- G^* has the same vertices as G
- if G has a directed path from u to v ($u \neq v$), G^* has a directed edge from u to v

The transitive closure provides reachability information about a digraph



Floyd-Warshall's Algorithm

- Number vertices $v_1, \ldots, v_n$
- Compute digraphs $G_0, \ldots, G_n$
  - $G_0 = G$
  - G_k has directed edge (v_i, v_j) if G has a directed path from v_i to v_j with intermediate vertices in $\{v_1, \ldots, v_k\}$
- We have that $G_n = G^*$
- In phase k, digraph G_k is computed from G_{k-1}
- Running time: $O(n^3)$, assuming areAdjacent is $O(1)$ (e.g., adjacency matrix)

Algorithm *FloydWarshall*(*G*)
  Input digraph *G*
  Output transitive closure *G** of *G*
  i ← 1
  for all *v* ∈ *G.vertices*()
    denote *v* as v_i
    i ← *i* + 1
  G_0 ← *G*
  for *k* ← 1 **to** *n* **do**
    G_k ← G_{k-1}
    for *i* ← 1 **to** *n* (*i* ≠ *k*) **do**
      for *j* ← 1 **to** *n* (*j* ≠ *i*, *k*) **do**
        if G_{k-1}.*areAdjacent*(v_i, v_k) ∧ G_{k-1}.*areAdjacent*(v_k, v_j)
          if ¬G_k.*areAdjacent*(v_i, v_j)
            G_k.*insertDirectedEdge*(v_i, v_j, *k*)
  return G_n

DAGs and Topological Ordering

- A directed acyclic graph (DAG) is a digraph that has no directed cycles
- A topological ordering of a digraph is a numbering

  $v_1, \ldots, v_n$

Theorem A digraph admits a topological ordering if and only if it is a DAG

Topological Sorting $O(n+m)$

Number vertices, so that (u,v) in E implies u < v

Algorithm TopologicalSort(*G*)
  H ← *G* // Temporary copy of *G*
  n ← *G.numVertices*()
  while *H* is not empty **do**
    Let *v* be a vertex with no outgoing edges
    Label *v* ← *n*
    n ← *n* − 1
    Remove *v* from *H*

Algorithm *topologicalDFS*(*G*)
  Input dag *G*
  Output topological ordering of *G*
  n ← *G.numVertices*()
  for all *u* ∈ *G.vertices*()
    setLabel(*u*, *UNEXPLORED*)
  for all *v* ∈ *G.vertices*()
    if *getLabel*(*v*) = *UNEXPLORED*
      topologicalDFS(*G*, *v*)

Algorithm *topologicalDFS*(*G*, *v*)
  Input graph *G* and a start vertex *v* of *G*
  Output labeling of the vertices of *G* in the connected component of *v*
  setLabel(*v*, *VISITED*)
  for all *e* ∈ *G.outEdges*(*v*)
    { outgoing edges }
    w ← *opposite*(*v*,*e*)
    if *getLabel*(*w*) = *UNEXPLORED*
      { *e* is a discovery edge }
      topologicalDFS(*G*, *w*)
    else
      { *e* is a forward or cross edge }
  Label *v* with topological number *n*
  n ← *n* - 1

Shortest Paths

- In a weighted graph, each edge has an associated numerical value, called the weight of the edge
- Given a weighted graph and two vertices u and v, we want to find a path of minimum total weight between u and v.

Property 1:
A subpath of a shortest path is itself a shortest path

Property 2:
There is a tree of shortest paths from a start vertex to all the other vertices

Dijkstra's Algorithm $O((n+m)\log n)$, $O(m \log n)$

- Assumptions:
  - the graph is connected
  - the edges are undirected
  - the edge weights are nonnegative
- We grow a "cloud" of vertices, beginning with s and eventually covering all the vertices
- We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices
- At each step
  - We add to the cloud the vertex u outside the cloud with the smallest distance label, $d(u)$
  - We update the labels of the vertices adjacent to u

Algorithm ShortestPath(G, s):
Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.
Output: The length of a shortest path from s to v for each vertex v of G.

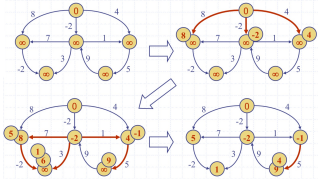
Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
Let a priority queue Q contain all the vertices of G using the D labels as keys.
while Q is not empty **do**
  {pull a new vertex u into the cloud}
  $u =$ value returned by Q.remove_min()
  for each vertex v adjacent to u such that v is in Q **do**
    {perform the ***relaxation*** procedure on edge (u, v)}
    if $D[u] + w(u, v) < D[v]$ **then**
      $D[v] = D[u] + w(u, v)$
      Change to $D[v]$ the key of vertex v in Q.
return the label $D[v]$ of each vertex v

Bellman-Ford Algorithm $O(nm)$

- Works even with negative-weight edges
- Must assume directed edges (for otherwise we would have negative-weight cycles)
- Iteration i finds all shortest paths that use i edges.

Algorithm *BellmanFord*(G, s)
  for all $v \in G.vertices()$
    if $v = s$
      setDistance$(v, 0)$
    else
      setDistance(v, ∞)
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    for each $e \in G.edges()$
      { relax edge e }
      $u \leftarrow G.origin(e)$
      $z \leftarrow G.opposite(u,e)$
      $r \leftarrow getDistance(u) + weight(e)$
      if $r < getDistance(z)$
        setDistance(z,r)

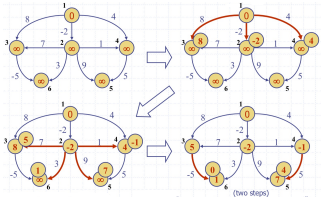
Nodes are labeled with their d(v) values



DAG-based Algorithm $O(n+m)$

- Works even with negative-weight edges
- Uses topological order

Algorithm *DagDistances*(G, s)
  for all $v \in G.vertices()$
    if $v = s$
      setDistance$(v, 0)$
    else
      setDistance(v, ∞)
  { Perform a topological sort of the vertices }
  for $u \leftarrow 1$ **to** n **do** {in topological order}
    for each $e \in G.outEdges(u)$
      { relax edge e }
      $z \leftarrow G.opposite(u,e)$
      $r \leftarrow getDistance(u) + weight(e)$
      if $r < getDistance(z)$
        setDistance(z,r)



Minimum Spanning Trees

Spanning subgraph
- Subgraph of a graph G containing all the vertices of G

Spanning tree
- Spanning subgraph that is itself a (free) tree

Minimum spanning tree (MST)
- Spanning tree of a weighted graph with minimum total edge weight

Cycle Property:

- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Partition Property:

- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Prim-Jarnik's Algorithm $O(m \log n)$

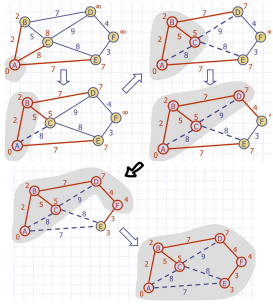
We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s

We store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud

At each step:
- We add to the cloud the vertex u outside the cloud with the smallest distance label
- We update the labels of the vertices adjacent to u

Algorithm PrimJarnik(G):
Input: An undirected, weighted, connected graph G with n vertices and m edges
Output: A minimum spanning tree T for G

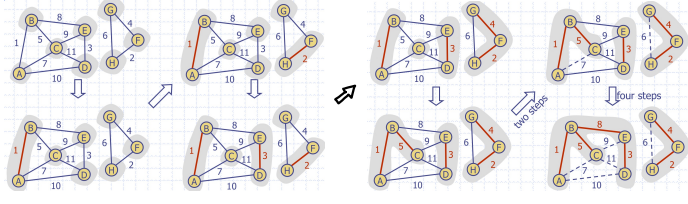
Pick any vertex s of G
$D[s] = 0$
for each vertex $v \neq s$ **do**
  $D[v] = \infty$
Initialize $T = \emptyset$.
Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
while Q is not empty **do**
  $(u, e) =$ value returned by Q.remove_min()
  Connect vertex u to T using edge e.
  for each edge $e' = (u, v)$ such that v is in Q **do**
    {check if edge (u, v) better connects v to T}
    if $w(u, v) < D[v]$ **then**
      $D[v] = w(u, v)$
      Change the key of vertex v in Q to $D[v]$.
      Change the value of vertex v in Q to (v, e').
return the tree T



Kruskal's Approach

Algorithm Kruskal(G):
Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G

for each vertex v in G **do**
  Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of the MST}
while T has fewer than $n - 1$ edges **do**
  $(u, v) =$ value returned by Q.remove_min()
  Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
  if $C(u) \neq C(v)$ **then**
    Add edge (u, v) to T.
    Merge $C(u)$ and $C(v)$ into one cluster.
return tree T




codice

ARRAY-BASED PQ

```java
/** An implementation of a priority queue using an array-based heap. */
public class HeapPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  protected ArrayList<Entry<K,V>> heap = new ArrayList<>();
  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public HeapPriorityQueue() { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public HeapPriorityQueue(Comparator<K> comp) { super(comp); }
  // protected utilities
  protected int parent(int j) { return (j-1) / 2; }          // truncating division
  protected int left(int j) { return 2*j + 1; }
  protected int right(int j) { return 2*j + 2; }
  protected boolean hasLeft(int j) { return left(j) < heap.size(); }
  protected boolean hasRight(int j) { return right(j) < heap.size(); }
  /** Exchanges the entries at indices i and j of the array list. */
  protected void swap(int i, int j) {
    Entry<K,V> temp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, temp);
  }
  /** Moves the entry at index j higher, if necessary, to restore the heap property. */
  protected void upheap(int j) {
    while (j > 0) {                 // continue until reaching root (or break statement)
      int p = parent(j);
      if (compare(heap.get(j), heap.get(p)) >= 0) break;   // heap property verified
      swap(j, p);
      j = p;                                    // continue from the parent's location
    }
  }
  /** Moves the entry at index j lower, if necessary, to restore the heap property. */
  protected void downheap(int j) {
    while (hasLeft(j)) {                  // continue to bottom (or break statement)
      int leftIndex = left(j);
      int smallChildIndex = leftIndex;              // although right may be smaller
      if (hasRight(j)) {
          int rightIndex = right(j);
          if (compare(heap.get(leftIndex), heap.get(rightIndex)) > 0)
            smallChildIndex = rightIndex;           // right child is smaller
      }
      if (compare(heap.get(smallChildIndex), heap.get(j)) >= 0)
        break;                                  // heap property has been restored
      swap(j, smallChildIndex);
      j = smallChildIndex;                      // continue at position of the child
    }
  }

  // public methods
  /** Returns the number of items in the priority queue. */
  public int size() { return heap.size(); }
  /** Returns (but does not remove) an entry with minimal key (if any). */
  public Entry<K,V> min() {
    if (heap.isEmpty()) return null;
    return heap.get(0);
  }
  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);       // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    heap.add(newest);                           // add to the end of the list
    upheap(heap.size() - 1);                    // upheap newly added entry
    return newest;
  }
  /** Removes and returns an entry with minimal key (if any). */
  public Entry<K,V> removeMin() {
    if (heap.isEmpty()) return null;
    Entry<K,V> answer = heap.get(0);
    swap(0, heap.size() - 1);                   // put minimum item at the end
    heap.remove(heap.size() - 1);               // and remove it from the list;
    downheap(0);                                // then fix new root
    return answer;
  }
}
```

UNSORTED LIST PQ

```java
/** An implementation of a priority queue with an unsorted list. */
public class UnsortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public UnsortedPriorityQueue() { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public UnsortedPriorityQueue(Comparator<K> comp) { super(comp); }

  /** Returns the Position of an entry having minimal key. */
  private Position<Entry<K,V>> findMin() {    // only called when nonempty
    Position<Entry<K,V>> small = list.first();
    for (Position<Entry<K,V>> walk : list.positions())
      if (compare(walk.getElement(), small.getElement()) < 0)
        small = walk;          // found an even smaller key
    return small;
  }

  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);     // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    list.addLast(newest);
    return newest;
  }

  /** Returns (but does not remove) an entry with minimal key. */
  public Entry<K,V> min() {
    if (list.isEmpty()) return null;
    return findMin().getElement();
  }

  /** Removes and returns an entry with minimal key. */
  public Entry<K,V> removeMin() {
    if (list.isEmpty()) return null;
    return list.remove(findMin());
  }

  /** Returns the number of items in the priority queue. */
  public int size() { return list.size(); }
}
```

SORTED LIST PQ

```java
/** An implementation of a priority queue with a sorted list. */
public class SortedPriorityQueue<K,V> extends AbstractPriorityQueue<K,V> {
  /** primary collection of priority queue entries */
  private PositionalList<Entry<K,V>> list = new LinkedPositionalList<>();

  /** Creates an empty priority queue based on the natural ordering of its keys. */
  public SortedPriorityQueue() { super(); }
  /** Creates an empty priority queue using the given comparator to order keys. */
  public SortedPriorityQueue(Comparator<K> comp) { super(comp); }

  /** Inserts a key-value pair and returns the entry created. */
  public Entry<K,V> insert(K key, V value) throws IllegalArgumentException {
    checkKey(key);     // auxiliary key-checking method (could throw exception)
    Entry<K,V> newest = new PQEntry<>(key, value);
    Position<Entry<K,V>> walk = list.last();
    // walk backward, looking for smaller key
    while (walk != null && compare(newest, walk.getElement()) < 0)
      walk = list.before(walk);
    if (walk == null)
      list.addFirst(newest);                    // new key is smallest
    else
      list.addAfter(walk, newest);              // newest goes after walk
    return newest;
  }

  /** Returns (but does not remove) an entry with minimal key. */
  public Entry<K,V> min() {
    if (list.isEmpty()) return null;
    return list.first().getElement();
  }

  /** Removes and returns an entry with minimal key. */
  public Entry<K,V> removeMin() {
    if (list.isEmpty()) return null;
    return list.remove(list.first());
  }

  /** Returns the number of items in the priority queue. */
  public int size() { return list.size(); }
}
```

ABSTRACT HASH MAP (JAVA)

```java
public abstract class AbstractHashMap<K,V> extends AbstractMap<K,V> {
  protected int n = 0;                    // number of entries in the dictionary
  protected int capacity;                 // length of the table
  private int prime;                      // prime factor
  private long scale, shift;              // the shift and scaling factors
  public AbstractHashMap(int cap, int p) {
    prime = p;
    capacity = cap;
    Random rand = new Random();
    scale = rand.nextInt(prime-1) + 1;
    shift = rand.nextInt(prime);
    createTable();
  }
  public AbstractHashMap(int cap) { this(cap, 109345121); }  // default prime
  public AbstractHashMap() { this(17); }                     // default capacity
  // public methods
  public int size() { return n; }
  public V get(K key) { return bucketGet(hashValue(key), key); }
  public V remove(K key) { return bucketRemove(hashValue(key), key); }
  public V put(K key, V value) {
    V answer = bucketPut(hashValue(key), key, value);
    if (n > capacity / 2)                 // keep load factor <= 0.5
      resize(2 * capacity - 1);           // (or find a nearby prime)
    return answer;
  }
  // private utilities
  private int hashValue(K key) {
    return (int) ((Math.abs(key.hashCode()*scale + shift) % prime) % capacity);
  }
  private void resize(int newCap) {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>(n);
    for (Entry<K,V> e : entrySet())
      buffer.add(e);
    capacity = newCap;
    createTable();                        // based on updated capacity
    n = 0;                                // will be recomputed while reinserting entries
    for (Entry<K,V> e : buffer)
      put(e.getKey(), e.getValue());
  }
  // protected abstract methods to be implemented by subclasses
  protected abstract void createTable();
  protected abstract V bucketGet(int h, K k);
  protected abstract V bucketPut(int h, K k, V v);
  protected abstract V bucketRemove(int h, K k);
}
```

PROBE HASH MAP (JAVA)

```java
public class ProbeHashMap<K,V> extends AbstractHashMap<K,V> {
  private MapEntry<K,V>[] table;        // a fixed array of entries (all initially null)
  private MapEntry<K,V> DEFUNCT = new MapEntry<>(null, null);  //sentinel
  public ProbeHashMap() { super(); }
  public ProbeHashMap(int cap) { super(cap); }
  public ProbeHashMap(int cap, int p) { super(cap, p); }
  /** Creates an empty table having length equal to current capacity. */
  protected void createTable() {
    table = (MapEntry<K,V>[]) new MapEntry[capacity];   // safe cast
  }
  /** Returns true if location is either empty or the "defunct" sentinel. */
  private boolean isAvailable(int j) {
    return (table[j] == null || table[j] == DEFUNCT);
  }
  /** Returns index with key k, or -(a+1) such that k could be added at index a. */
  private int findSlot(int h, K k) {
    int avail = -1;                     // no slot available (thus far)
    int j = h;                          // index while scanning table
    do {
      if (isAvailable(j)) {             // may be either empty or defunct
        if (avail == -1) avail = j;     // this is the first available slot!
        if (table[j] == null) break;    // if empty, search fails immediately
      } else if (table[j].getKey().equals(k))
        return j;                       // successful match
      j = (j+1) % capacity;             // keep looking (cyclically)
    } while (j != h);                   // stop if we return to the start
    return -(avail + 1);                // search has failed
  }
  /** Returns value associated with key k in bucket with hash value h, or else null. */
  protected V bucketGet(int h, K k) {
    int j = findSlot(h, k);
    if (j < 0) return null;             // no match found
    return table[j].getValue();
  }
  /** Associates key k with value v in bucket with hash value h; returns old value. */
  protected V bucketPut(int h, K k, V v) {
    int j = findSlot(h, k);
    if (j >= 0)                         // this key has an existing entry
      return table[j].setValue(v);
    table[-(j+1)] = new MapEntry<>(k, v);   // convert to proper index
    n++;
    return null;
  }
  /** Removes entry having key k from bucket with hash value h (if any). */
  protected V bucketRemove(int h, K k) {
    int j = findSlot(h, k);
    if (j < 0) return null;             // nothing to remove
    V answer = table[j].getValue();
    table[j] = DEFUNCT;                 // mark this slot as deactivated
    n--;
    return answer;
  }
  /** Returns an iterable collection of all key-value entries of the map. */
  public Iterable<Entry<K,V>> entrySet() {
    ArrayList<Entry<K,V>> buffer = new ArrayList<>();
    for (int h=0; h < capacity; h++)
      if (!isAvailable(h)) buffer.add(table[h]);
    return buffer;
  }
}
```

SORTED MAP USING AVL (JAVA)

```java
/** An implementation of a sorted map using an AVL tree. */
public class AVLTreeMap<K,V> extends TreeMap<K,V> {
  /** Constructs an empty map using the natural ordering of keys. */
  public AVLTreeMap() { super(); }
  /** Constructs an empty map using the given comparator to order keys. */
  public AVLTreeMap(Comparator<K> comp) { super(comp); }
  /** Returns the height of the given tree position. */
  protected int height(Position<Entry<K,V>> p) {
    return tree.getAux(p);
  }
  /** Recomputes the height of the given position based on its children's heights. */
  protected void recomputeHeight(Position<Entry<K,V>> p) {
    tree.setAux(p, 1 + Math.max(height(left(p)), height(right(p))));
  }
  /** Returns whether a position has balance factor between -1 and 1 inclusive. */
  protected boolean isBalanced(Position<Entry<K,V>> p) {
    return Math.abs(height(left(p)) - height(right(p))) <= 1;
  }
  /** Returns a child of p with height no smaller than that of the other child. */
  protected Position<Entry<K,V>> tallerChild(Position<Entry<K,V>> p) {
    if (height(left(p)) > height(right(p))) return left(p);          // clear winner
    if (height(left(p)) < height(right(p))) return right(p);         // clear winner
    // equal height children; break tie while matching parent's orientation
    if (isRoot(p)) return left(p);                                   // choice is irrelevant
    if (p == left(parent(p))) return left(p);                        // return aligned child
    else return right(p);
  }
  protected void rebalance(Position<Entry<K,V>> p) {
    int oldHeight, newHeight;
    do {
      oldHeight = height(p);                                         // not yet recalculated if internal
      if (!isBalanced(p)) {                                          // imbalance detected
        // perform trinode restructuring, setting p to resulting root,
        // and recompute new local heights after the restructuring
        p = restructure(tallerChild(tallerChild(p)));
        recomputeHeight(left(p));
        recomputeHeight(right(p));
      }
      recomputeHeight(p);
      newHeight = height(p);
      p = parent(p);
    } while (oldHeight != newHeight && p != null);
  }
  /** Overrides the TreeMap rebalancing hook that is called after an insertion. */
  protected void rebalanceInsert(Position<Entry<K,V>> p) {
    rebalance(p);
  }
  /** Overrides the TreeMap rebalancing hook that is called after a deletion. */
  protected void rebalanceDelete(Position<Entry<K,V>> p) {
    if (!isRoot(p))
      rebalance(parent(p));
  }
}
```

MERGE-SORT + MERGE (JAVA)

```java
/** Merge-sort contents of array S. */
public static <K> void mergeSort(K[ ] S, Comparator<K> comp) {
  int n = S.length;
  if (n < 2) return;                          // array is trivially sorted
  // divide
  int mid = n/2;
  K[ ] S1 = Arrays.copyOfRange(S, 0, mid);    // copy of first half
  K[ ] S2 = Arrays.copyOfRange(S, mid, n);    // copy of second half
  // conquer (with recursion)
  mergeSort(S1, comp);                        // sort copy of first half
  mergeSort(S2, comp);                        // sort copy of second half
  // merge results
  merge(S1, S2, S, comp);                 // merge sorted halves back into original
}
```

```java
/** Merge contents of arrays S1 and S2 into properly sized array S. */
public static <K> void merge(K[ ] S1, K[ ] S2, K[ ] S, Comparator<K> comp) {
  int i = 0, j = 0;
  while (i + j < S.length) {
    if (j == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
      S[i+j] = S1[i++];                   // copy ith element of S1 and increment i
    else
      S[i+j] = S2[j++];                   // copy jth element of S2 and increment j
  }
}
```

QUICK-SORT

```java
/** Quick-sort contents of a queue. */
public static <K> void quickSort(Queue<K> S, Comparator<K> comp) {
  int n = S.size();
  if (n < 2) return;                          // queue is trivially sorted
  // divide
  K pivot = S.first();                        // using first as arbitrary pivot
  Queue<K> L = new LinkedQueue<>();
  Queue<K> E = new LinkedQueue<>();
  Queue<K> G = new LinkedQueue<>();
  while (!S.isEmpty()) {                      // divide original into L, E, and G
    K element = S.dequeue();
    int c = comp.compare(element, pivot);
    if (c < 0)                                // element is less than pivot
      L.enqueue(element);
    else if (c == 0)                          // element is equal to pivot
      E.enqueue(element);
    else                                      // element is greater than pivot
      G.enqueue(element);
  }
  // conquer
  quickSort(L, comp);                         // sort elements less than pivot
  quickSort(G, comp);                         // sort elements greater than pivot
  // concatenate results
  while (!L.isEmpty())
    S.enqueue(L.dequeue());
  while (!E.isEmpty())
    S.enqueue(E.dequeue());
  while (!G.isEmpty())
    S.enqueue(G.dequeue());
}
```

IN-PLACE QUICK-SORT

```java
/** Sort the subarray S[a..b] inclusive. */
private static <K> void quickSortInPlace(K[ ] S, Comparator<K> comp,
                                                        int a, int b) {
  if (a >= b) return;       /|/ subarray is trivially sorted
  int left = a;
  int right = b-1;
  K pivot = S[b];
  K temp;                   // temp object used for swapping
  while (left <= right) {
    // scan until reaching value equal or larger than pivot (or right marker)
    while (left <= right && comp.compare(S[left], pivot) < 0) left++;
    // scan until reaching value equal or smaller than pivot (or left marker)
    while (left <= right && comp.compare(S[right], pivot) > 0) right--;
    if (left <= right) {    // indices did not strictly cross
      // so swap values and shrink range
      temp = S[left]; S[left] = S[right]; S[right] = temp;
      left++; right--;
    }
  }
  // put pivot into its final place (currently marked by left index)
  temp = S[left]; S[left] = S[b]; S[b] = temp;
  // make recursive calls
  quickSortInPlace(S, comp, a, left - 1);
  quickSortInPlace(S, comp, left + 1, b);
}
```

DFS

```java
/** Performs depth-first search of Graph g starting at Vertex u. */
public static <V,E> void DFS(Graph<V,E> g, Vertex<V> u,
                             Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  known.add(u);                          // u has been discovered
  for (Edge<E> e : g.outgoingEdges(u)) {  // for every outgoing edge from u
    Vertex<V> v = g.opposite(u, e);
    if (!known.contains(v)) {
      forest.put(v, e);                  // e is the tree edge that discovered v
      DFS(g, v, known, forest);          // recursively explore from v
    }
  }
}
```

FLOYD-WARSHALL'S TRANSITIVE CLOSURE

```java
/** Converts graph g into its transitive closure. */
public static <V,E> void transitiveClosure(Graph<V,E> g) {
  for (Vertex<V> k : g.vertices())
    for (Vertex<V> i : g.vertices())
      // verify that edge (i,k) exists in the partial closure
      if (i != k && g.getEdge(i,k) != null)
        for (Vertex<V> j : g.vertices())
          // verify that edge (k,j) exists in the partial closure
          if (i != j && j != k && g.getEdge(k,j) != null)
            // if (i,j) not yet included, add it to the closure
            if (g.getEdge(i,j) == null)
              g.insertEdge(i, j, null);
}
```

TOPOLOGICAL SORT

```java
/** Returns a list of verticies of directed acyclic graph g in topological order. */
public static <V,E> PositionalList<Vertex<V>> topologicalSort(Graph<V,E> g) {
  // list of vertices placed in topological order
  PositionalList<Vertex<V>> topo = new LinkedPositionalList<>();
  // container of vertices that have no remaining constraints
  Stack<Vertex<V>> ready = new LinkedStack<>();
  // map keeping track of remaining in-degree for each vertex
  Map<Vertex<V>, Integer> inCount = new ProbeHashMap<>();
  for (Vertex<V> u : g.vertices()) {
    inCount.put(u, g.inDegree(u));       // initialize with actual in-degree
    if (inCount.get(u) == 0)             // if u has no incoming edges,
      ready.push(u);                     // it is free of constraints
  }
  while (!ready.isEmpty()) {
    Vertex<V> u = ready.pop();
    topo.addLast(u);
    for (Edge<E> e : g.outgoingEdges(u)) {  // consider all outgoing neighbors of u
      Vertex<V> v = g.opposite(u, e);
      inCount.put(v, inCount.get(v) - 1);   // v has one less constraint without u
      if (inCount.get(v) == 0)
        ready.push(v);
    }
  }
  return topo;
}
```

PATH FINDING (JAVA)

```java
/** Returns an ordered list of edges comprising the directed path from u to v. */
public static <V,E> PositionalList<Edge<E>>
constructPath(Graph<V,E> g, Vertex<V> u, Vertex<V> v,
              Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Edge<E>> path = new LinkedPositionalList<>();
  if (forest.get(v) != null) {           // v was discovered during the search
    Vertex<V> walk = v;                  // we construct the path from back to front
    while (walk != u) {
      Edge<E> edge = forest.get(walk);
      path.addFirst(edge);               // add edge to *front* of path
      walk = g.opposite(walk, edge);     // repeat with opposite endpoint
    }
  }
  return path;
}
```

BFS

```java
/** Performs breadth-first search of Graph g starting at Vertex u. */
public static <V,E> void BFS(Graph<V,E> g, Vertex<V> s,
                             Set<Vertex<V>> known, Map<Vertex<V>,Edge<E>> forest) {
  PositionalList<Vertex<V>> level = new LinkedPositionalList<>();
  known.add(s);
  level.addLast(s);                      // first level includes only s
  while (!level.isEmpty()) {
    PositionalList<Vertex<V>> nextLevel = new LinkedPositionalList<>();
    for (Vertex<V> u : level)
      for (Edge<E> e : g.outgoingEdges(u)) {
        Vertex<V> v = g.opposite(u, e);
        if (!known.contains(v)) {
          known.add(v);
          forest.put(v, e);              // e is the tree edge that discovered v
          nextLevel.addLast(v);          // v will be further considered in next pass
        }
      }
    level = nextLevel;                   // relabel 'next' level to become the current
  }
}
```

DIJKSTRA'S SHORTEST PATH

```java
/** Computes shortest-path distances from src vertex to all reachable vertices of g. */
public static <V> Map<Vertex<V>, Integer>
shortestPathLengths(Graph<V,Integer> g, Vertex<V> src) {
  // d.get(v) is upper bound on distance from src to v
  Map<Vertex<V>, Integer> d = new ProbeHashMap<>();
  // map reachable v to its d value
  Map<Vertex<V>, Integer> cloud = new ProbeHashMap<>();
  // pq will have vertices as elements, with d.get(v) as key
  AdaptablePriorityQueue<Integer, Vertex<V>> pq;
  pq = new HeapAdaptablePriorityQueue<>();
  // maps from vertex to its pq locator
  Map<Vertex<V>, Entry<Integer,Vertex<V>>> pqTokens;
  pqTokens = new ProbeHashMap<>();

  // for each vertex v of the graph, add an entry to the priority queue, with
  // the source having distance 0 and all others having infinite distance
  for (Vertex<V> v : g.vertices()) {
    if (v == src)
      d.put(v,0);
    else
      d.put(v, Integer.MAX_VALUE);
    pqTokens.put(v, pq.insert(d.get(v), v));          // save entry for future updates
  }
  // now begin adding reachable vertices to the cloud
  while (!pq.isEmpty()) {
    Entry<Integer, Vertex<V>> entry = pq.removeMin();
    int key = entry.getKey();
    Vertex<V> u = entry.getValue();
    cloud.put(u, key);                                // this is actual distance to u
    pqTokens.remove(u);                               // u is no longer in pq
    for (Edge<Integer> e : g.outgoingEdges(u)) {
      Vertex<V> v = g.opposite(u,e);
      if (cloud.get(v) == null) {
        // perform relaxation step on edge (u,v)
        int wgt = e.getElement();
        if (d.get(u) + wgt < d.get(v)) {              // better path to v?
          d.put(v, d.get(u) + wgt);                   // update the distance
          pq.replaceKey(pqTokens.get(v), d.get(v));   // update the pq entry
        }
      }
    }
  }
  return cloud;            // this only includes reachable vertices
}
```

KRUSKAL'S MST

```java
/** Computes a minimum spanning tree of graph g using Kruskal's algorithm. */
public static <V> PositionalList<Edge<Integer>> MST(Graph<V,Integer> g) {
  // tree is where we will store result as it is computed
  PositionalList<Edge<Integer>> tree = new LinkedPositionalList<>();
  // pq entries are edges of graph, with weights as keys
  PriorityQueue<Integer, Edge<Integer>> pq = new HeapPriorityQueue<>();
  // union-find forest of components of the graph
  Partition<Vertex<V>> forest = new Partition<>();
  // map each vertex to the forest position
  Map<Vertex<V>,Position<Vertex<V>>> positions = new ProbeHashMap<>();

  for (Vertex<V> v : g.vertices())
    positions.put(v, forest.makeGroup(v));

  for (Edge<Integer> e : g.edges())
    pq.insert(e.getElement(), e);

  int size = g.numVertices();
  // while tree not spanning and unprocessed edges remain...
  while (tree.size() != size - 1 && !pq.isEmpty()) {
    Entry<Integer, Edge<Integer>> entry = pq.removeMin();
    Edge<Integer> edge = entry.getValue();
    Vertex<V>[] endpoints = g.endVertices(edge);
    Position<Vertex<V>> a = forest.find(positions.get(endpoints[0]));
    Position<Vertex<V>> b = forest.find(positions.get(endpoints[1]));
    if (a != b) {
      tree.addLast(edge);
      forest.union(a,b);
    }
  }

  return tree;
}
```