Lecture 3

Literature: Chapter 4.5-4.7 + slides

- Greedy algorithms
- Minimum spanning trees
- Minimum spanning tree property
  (called cut-property in book)
- Kruskals algorithm
- Merge-Find sets (MF-sets)
  (also called Union-Find sets)
- (Priority queues)
- Clustering

(You should have seen most of this in a datastructure course so it's a lot of repetition.
New stuff is perhaps:
the proofs, the MF-set structure and clustering)

Next time:

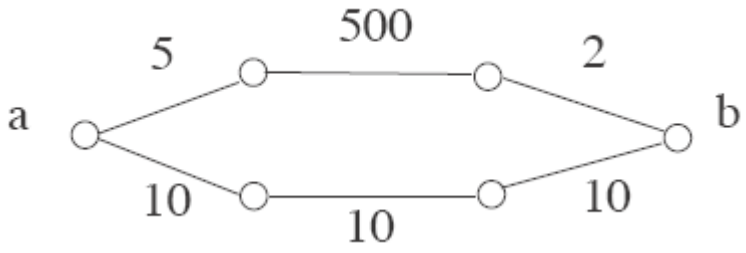
unweighted interval scheduling and "greed

vs. weighted interval scheduling and dynamic programming"

Greedy algorithms

A globally optimal solution can be arrived at by making locally optimal (greedy) choices.

- Greedy algorithms builds the solution in small steps. At every step it always chooses what seems to be the best choice "right now" in a local context without looking at the consequences further ahead.



- The choice made by a greedy algorithm may depend on choices so far, but it cannot depend on any future choices or on the solutions to sub-problems i.e.

  - the decision/choise is based on some insight about the problem e.g. a heuristic.

  - once it has made a decision it never changes it.

So make whatever choice seems best at the moment and then solve the sub-problem arising after the choice is made.

Of course, we must prove that a greedy choice at each step yields a globally optimal solution.

Typical examples of well known greedy algorithms with optimal results are:

- **Kruskal** and **Prims** algorithms for MST.
- **Dijkstras** algorithm for shortest paths in a graph.
- **Huffmans** algorithm that generates code for optimal message lengths.

Important: greed doesn't always work!

=> the correctness proof is important

What influence optimality of greedy alg.?

1 - What you are greedy with

Given n divisible objects

(say 1 ton of sand, 1 ton of salt...)

and a "knapsack"

(a wheel-barrow, a truck ...)

Object i has weight w_i and benefit b_i

and the knapsack has capacity W.

If a part of object i, say x_i, $0 \leq x_i \leq 1$, is put in the knapsack we get a profit of $b_i * x_i$.

Fill the knapsack to maximize the profit.

What to be greedy with?

Let's look at an example:

Assume that $n = 3$, $W = 20$,
$b_i = (25, 24, 15)$ and $w_i = (18, 15, 10)$.

a) Choose the object with **largest benefit**

i.e. choose as much as possible from the first (b_1=25), and then from the second (b_2=24)...

all of the first (18) plus $2/15$ of the second=>
$25+2/15*24=28.2$

b) Choose the object with **smallest weight**

i.e. as much as possible from the last ($w_3 = 10$)
and then from the second ($w_2 = 15$)...

=> $15+2/3*24 = 31$

c) Choose a balance between benefit and weight
i.e. the object with **maximal bnefit/weight**

$b_i/w_i = (1.39, 1.6, 1.5)$

i.e. as much as possible from the second=>
$24+1/2*15 = 31.5$ (optimal in this case)

What influence optimality of greedy alg.?

2 - What input looks like

Assume that we have coin values 1, 5, 10 and 25 "krona" and we want to give back change to a buyer with as few coins as possible.

Assume the change is 63 "krona"

The greedy approach

"choose the largest coin smaller than what is left"

works fine here. We get

$2*25$, $1*10$, $3*1$ "krona",

(63-25=38, 38-25=13, 13-10=3, 3-1=2, ...)

But optimality depends on the coins values

If we have the coins 1, 5 and 11 "krona"

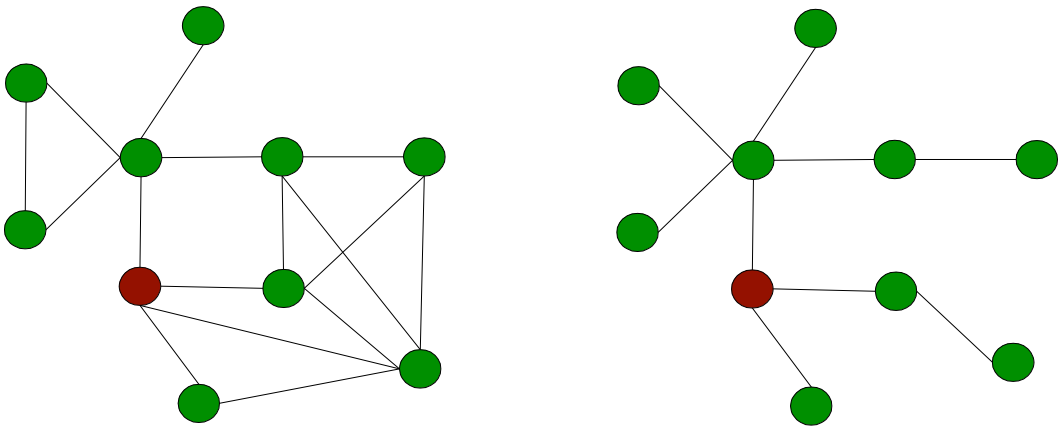
(sigh) it doesn't work

For ex. change for 15 "krona" gives us

$11+1+1+1+1$ but

$5+5+5$ is better.

Free trees



A free tree is a connected graph without cycles.

it has exactly n-1 edges

add one edge => cycle

remove one edge => not connected

Spanning trees

A spanning tree is a free tree connecting all nodes in a graph.

We are interested in

minimum spanning trees (MST)

(minimum Σedgecost)

Minimum spanning trees:

An old problem (algorithms from 1926!)

Given a graph G = (V, E),
find a subset of the edges T $\subseteq$ E such that
(V, T) is a spanning tree with min. edge cost.

Interesting because:

- frequently occurring problem
- can be used to solve many problems
  - minimum cost for different networks
  - cluster problems
  - approximation algorithm for TSP
- good example of a greedy algorithm and
- easy to find greedy algorithms for MST!
- an "exchange of argument" proof
- you need interesting data structures

A general graph has exponentially many spanning trees so it would be great to have an idea for how to generate one...

Minimum Spanning Tree property

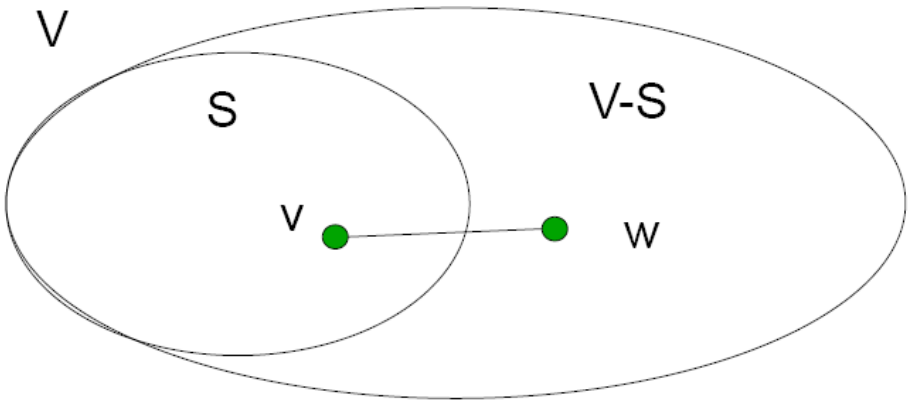
(4.17) Let G = (V, E) be a connected graph with positive edge costs.

Assume all costs are distinct.

Let S be any subset of nodes from G = (V, E) that is neither empty nor equal to V, and

let edge e = (v, w) be the minimum-cost edge with one end in S and the other in V-S,

Then **every** minimum spanning tree contains the edge e.



Thanks to
the MST property it's easy to find minimum spanning trees.

(MST property = cut property in the book)

Greedy proof strategies

- **Induction.**
- **Contradiction.**

- **Stay ahead.** Show that at each step the greedy algorithm makes a decision which is at least as good as another algorithm which we have deemed optimal. So, the greedy algorithm "stays ahead" of any optimal solution.
  Example: Interval scheduling

- **Exchange argument.** Gradually transform any solution, without violating its optimality, to the one found by the greedy algorithm . Example: MST-property, Minimizing lateness

- **Structural.** Establish a lower bound on the optimality of any solution for a particular problem. Then show that your greedy algorithm always achieves this lower bound. Example: Interval partitioning.

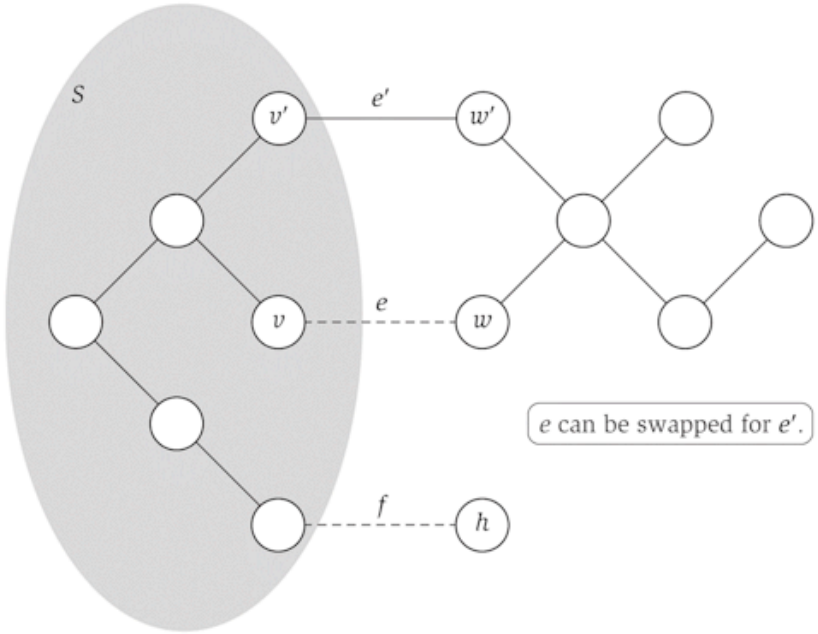
Proof of MST property (cut-property)

Idea: Variation of exchange arguments.

let edge e = (v, w) be the minimum-cost edge with one end in S and the other in V-S

Assume T is a spanning tree that does **not** contain e.

We need to show that T does not have minimum cost.



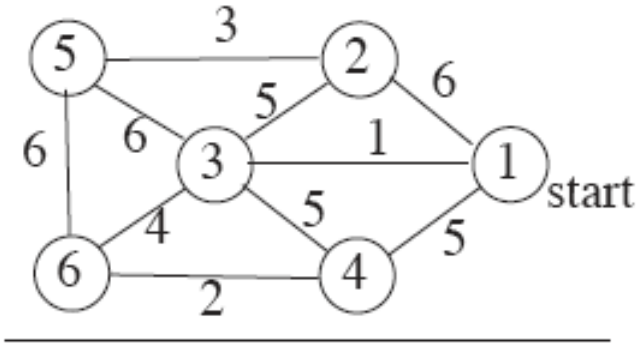
So find an edge e' in T that is more expensive than e and exchange them; show that resulting tree is:

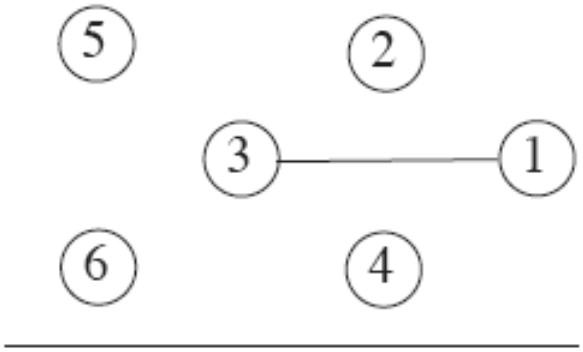
connected, acyclic and cheaper...

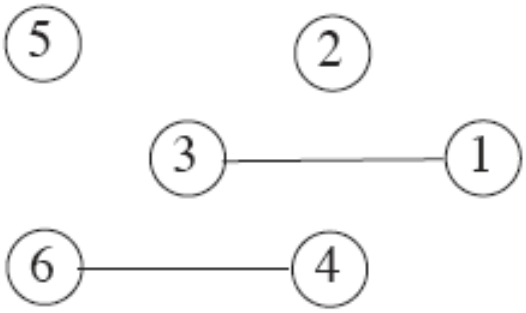
proof on BB and in book

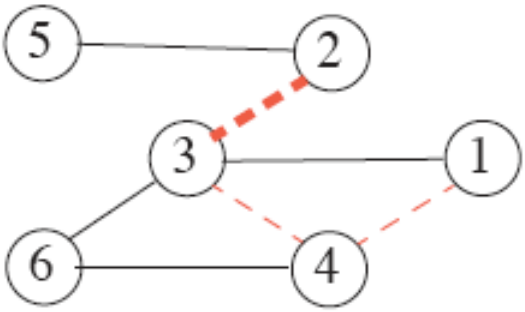
Kruskals algorithm

Idéa: (based on the MST property) always choose the cheapest edge that does not create a cycle









first pseudocode:

While there are edges left {

Find cheapest edge

Add to the mst if it does not create a cycle

}

Kruskal grows a set of connected components (cc)

- How to prove that it works? →

Optimality of Kruskals given the MST prop.

Claim: Given a connected graph G, Kruskals alg. produces a minimum spanning tree of G.

(4.18) Show that

(1) the result is a spanning tree and

(2) the edges added are the cheapest

(There is a difference between proving that

-an algorithm idea works i.e. produce a optimum answer and

- proving that some Java code works.

We need to do the first of these.)

(1) The output of Kruskal is a spanning tree

- it contains no cycles;
  that's how the algorithm works
- it's connected if G is connected
  since the algorithm doesn't stop until all nodes are in S.

Alternative 1:

(2) edges added are the cheapest

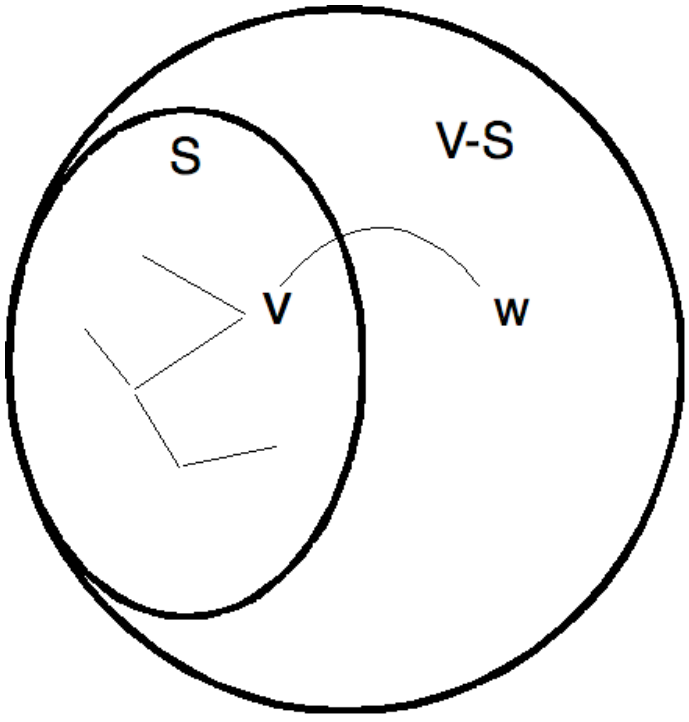
Let S be the set of all nodes to which v has a path at the moment before e = (v, w) is added by Kruskal.

Then $v \in S$ but $w \notin S$ since adding e does not create a cycle.

No other edge from S to V-S is encountered yet.

$\therefore$ e is the cheapest edge from S to V-S

so by 4.17 (MST prop.) it belongs to the mst



Alternative 2: Use proof by contradiction.

Suppose Kruskal's algorithm does not always give the minimum cost spanning tree on some graph then there is a graph on which it fails.

And if so, there must be a first edge (x, y) Kruskal adds such that the set of edges cannot be extended into a minimum spanning tree.

When we added (x, y) there previously was no path between x and y, or it would have created a cycle

Thus if we add (x, y) to the optimal tree it must create a cycle.

At least one edge in this cycle must have been added after (x, y), so it must have a heavier weight.

Deleting this heavy edge leave a better MST than the optimal tree? => A contradiction

Ref: www.cs.sunysb.edu/~skiena/373/newlectures/lecture13.pdf

Kruskal - first pseudocode

While there are edges left {

Find cheapest edge

Add to the mst if it does not create a cycle

}

- How to prove that it works? √ check

Many implement. details are left to deal with:

- **How to keep track of cycles?** →use MFsets

We start with a graph without edges where every node is a connected component.

In every step we choose the cheapest edge that connects two nodes in different connected components. When every node is in the same cc where done.

- How to enumerate all edges?
- How to find cheapest edge?
- How to add to the MST?
- What is the complexity?

Short intro to Merge- Find sets

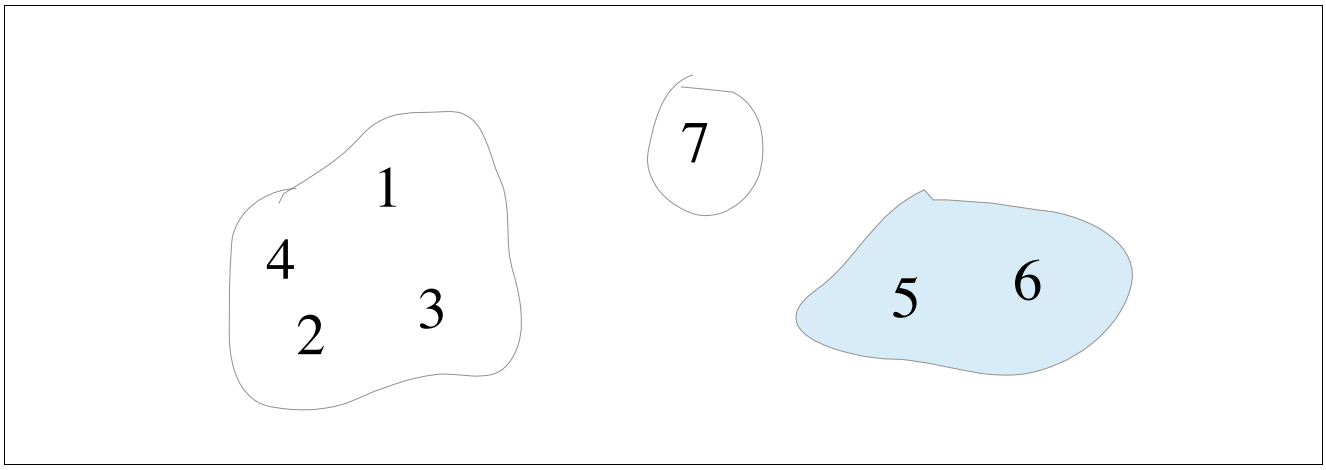
We have some kind of objects.

Could be computers in a network, web pages on the internet, transistors in a computer, pixels in a digital photo and more abstract things like assignment statements.

We want to create disjoint sets of objects that are connected.

Find queries: are two objects in the same set?

Union commands: replace sets containing two items by their union.



Find and union commands may be intermixed.
Number of operations M can be huge.
Number of objects N can be huge.

What is an efficient algorithm for MF-sets?

second pseudocode version

adding more detail: using Mfsets with each node in separate component

```
Cc = Connected components (the MF-set)
while nbrOfCc > 1 loop
   (v, w) = get cheapest edge
   if v and w are in different Cc then
      add (v, w) to T
      merge the cc of v and w
      nbrOfCc--
   end if
end while
```

ToDo list:

- How to **implement** Cc? → later
- How to enumerate all edges? → use p-queue
- How to find cheapest edge?
- How to add to the MST? → use a list
- What is the complexity? need more detail

Refined "version two": Pseudocode

```
kruskal(G: graph(V, E)) return set of edges
   MFset cc          // a Merge-Find set
   Set mst = ∅       //the growing spanning tree
   nodes v, w
   int nbrOfCc = n // where n= nbr of nodes
1  insert all edges in a priority queue
2  while nbrOfCc > 1 loop
3     (v, w) = deletemin(edges)
4     ucomp = find(v, cc), vcomp = find(w, cc)
5     if ucomp ≠ vcomp then
6        merge(ucomp, vcomp, cc)
7        nbrOfCc = nbrOfCc - 1
8        add (v, w) to mst
      end if
   end while
   return mst
end kruskal
```

If find $\in O(\log n)$, merge $\in O(1)$, deletemin $\in O(\log e)$ then altogether this is: $O(e \log e) = O(e \log n)$ since $n-1 <= e <= n^2$.

Complexity

it is likely that n<e and that e<n^2 (n-1<=e<=n^2)

MFset/UFset: assume tree implementation =>

- union/merge ∈ O(1)
- find ∈ O(logn) (if the tree is balanced)

Deletemin: put all edges in a priority queue (impl. with a heap).

- deletemin, insert/add ∈ O(loge)

row	what	cost	why
0	init	n for init. of cc, 1 or e for pq	
1	insert in pq	eloge +	
2	while	e*	each edge only once
3	deletemin	(loge +	see above
4	2*find	2loge +	logn<=loge if n<e
5	if	1 +	comparing numbers
6	merge	1 +	se above
7	decrease	1 +	trivially
8	insert/add	1)	a list

n + e + eloge + e(loge+2loge+4) ∈ O(eloge)

this is also O(elogn) because e<n^2

How to implement a MF-set? Try 1

A MFset contains a number of disjunctive sets

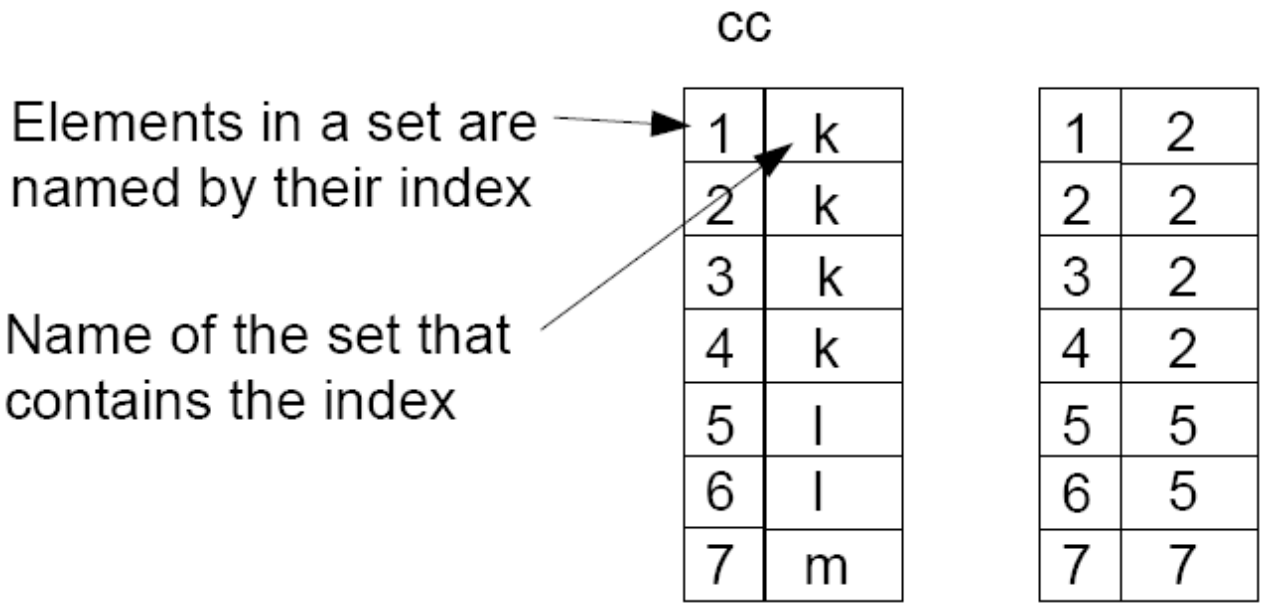
find(x) gives the set where x is

merge(A, B) is A = A∪B and A∩B = ∅

Example MF-set:

k = {1, 2, 3, 4}, l = {5, 6}, m = {7}

Try 1: quick-find with **an array**



initialize: ∀x, cc(x) = x, ∈ O(n)

find(x): cc(x) ∈ O(1)

merge(a, b): ∈ **O(n)**, n-1 merge give us **O(n²)**

```
for i in 1..nbrOfElements loop
    if cc(i) == b then
        cc(i) = a  // result in a
```

find(x): $\in O(1)$ => **quick find**

merge(a, b): $\in O(n)$, => slow merge

so m merge give us **O(m*n)**

Huge problems are common

say 10^{10} edges connecting 10^9 nodes:

=> qf-sm takes more than 10^{19} operations

If we assume

10^9 operations per second.

10^9 words of main memory.

$10^{19}/10^9/60/60/24/365$ = 317years

=> 300+ years of computer time!

And quadratic algorithms get worse with faster computers!

- New computer say 10 times as fast.
- But, also has 10 times as much memory so problem may be 10 times bigger.
- With quadratic algorithm, takes 100 times as long!

Try 2: a list

Link all members in a list, keep track of the size of the lists and always add the smaller list to the bigger.

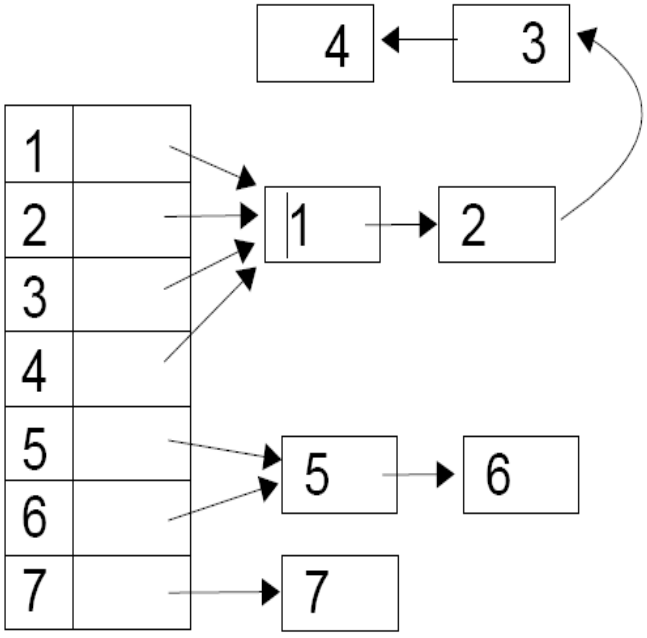
It's easy to add one list to another.

In Java you should not use "addAll" for this since we also need to walk through the shorter list and move all pointers from the array to the shorter list to point to the longer list.

find(x): cc(x) $\in$ O(1)

merge(a, b): $\in$ O(length of shortest list)

(It's the moving of pointers that are costly...)
Still to expensive.



Try 3: tree with pointers to parents

find(x) gives the set where x is; ∈ O(loge)

merge(A,B) is A = A∪B and A∩B = ∅; ∈ O(1)

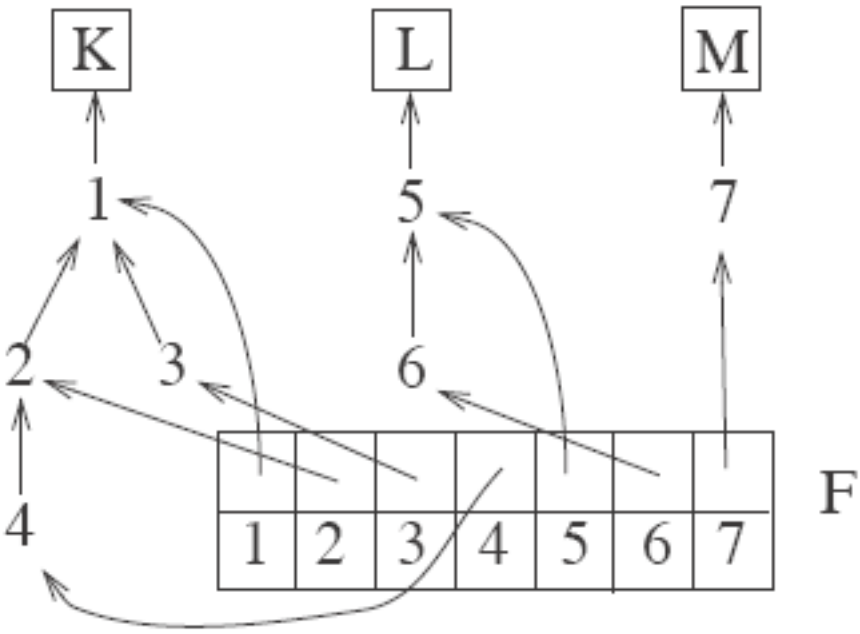
Try 3: tree with pointers to parents

Ex:

K = {1,2,3,4},

L = {5,6},

M = {7}



find(x):

look up F(x) and follow the pointers to the root which is O(logn) in wc:

Every merge at least doubles the size of the smallest tree; the trees depth increase with at most one, logn times so max depth of tree is logn.

merge(A,B):

let the root of the smaller tree be a child of the other. O(k)

We could also compact the tree in find.

Try 4: get rid of the "tree"

The trees can also be "stored" in the array!

Let F(i) represent the parent to i.
If i is a root then F(i)=0.

At start all nodes are in a set by themselves
index = set name
if F(i)=0
content = parent

1	2	3	4	5	6	7
0	0	0	0	0	0	0

Our example:
k = {1, 2, 3, 4},
l = {5, 6},
n = {7}

1	2	3	4	5	6	7
0	1	1	2	0	5	0
k				l		n

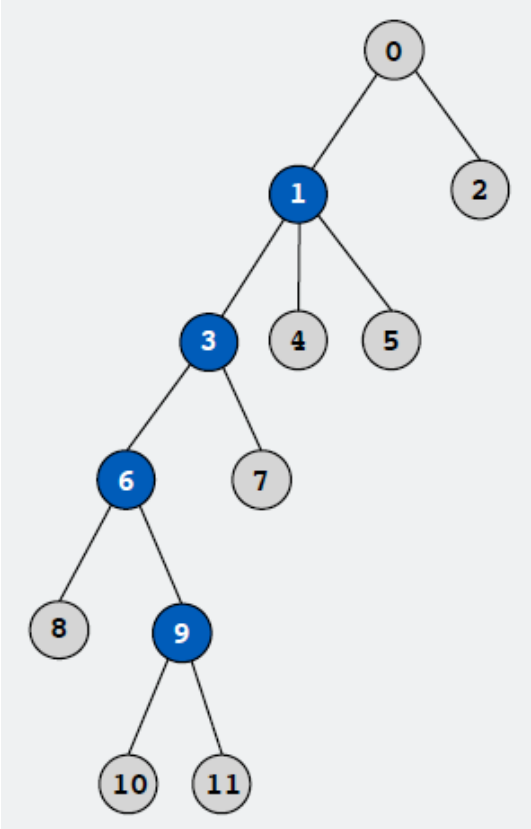
To keep track of the sizes we can replace all zeroes with the negative size of the trees

1	2	3	4	5	6	7
-4	1	1	2	-2	5	-1

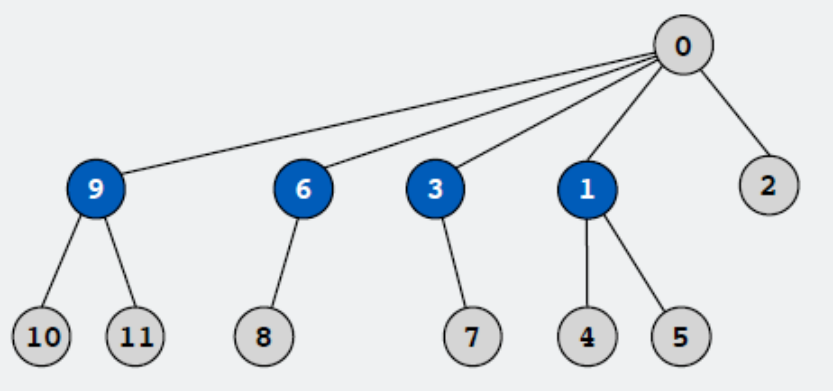
find(x): ∈ O(logn) but usually smaller
follow cc(i) to a root, return the index

merge(A,B): ∈ O(1)
merge(1,5): cc(1) = cc(5)+cc(1), cc(5) = 1

Add path compression



after find(9)



find(x): $\in$ O(k) - O(logn) => **quick find**

merge(a, b): $\in$ O(1), => **quick merge**

so m merge and find give us **O(m)**

or in wc O(mlogn)

Again: Huge problems are common

say 10^{10} edges connecting 10^9 nodes:

=> now takes some 10^{10} operations

=> $10^{10}/10^9$ = 10 seconds of computer time!

If we assume

10^9 operations per second.

10^9 words of main memory.

- New computer say 10 times as fast.
- also has 10 times as much memory so problem may be 10 times bigger.
- With linear algorithm, takes 10 times as long!

Some steps to developing an usable algorithm:

Define the problem.
Find an (any) algorithm to solve it.
Prove it works.

Analyse the algorithm
Fast enough? ("enough" is relative)
If not, figure out why,
- Look at the complexity analysis.
- Better datastructures?
- Optimizations in the algorithm?
- Completely new algorithm?

i.e find a way to address the problem.
Iterate until satisfied.

Repetition of Priority queue

A set with the (only) operations

insert and **deletemin**

is called a priority queue.

Representations:

It can be represented in the same ways that a set is represented except with a hash table.

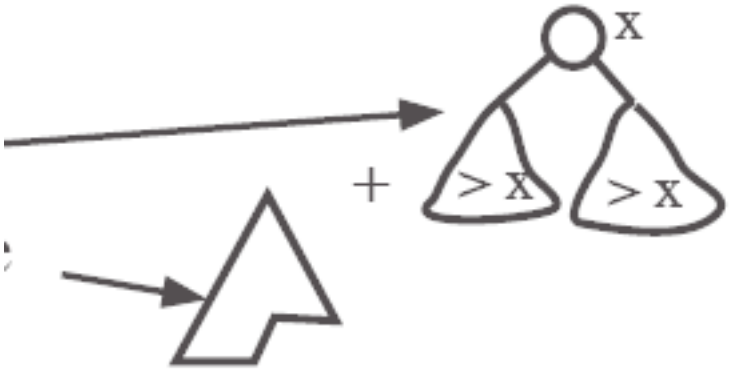
Usually we use:

- sorted or unsorted array/list O(1) - O(n)
- partially ordered leftbalanced (POL) tree

The advantage with this representation is that both insert and deletemin will take O(logn)

partially ordered

leftbalanced



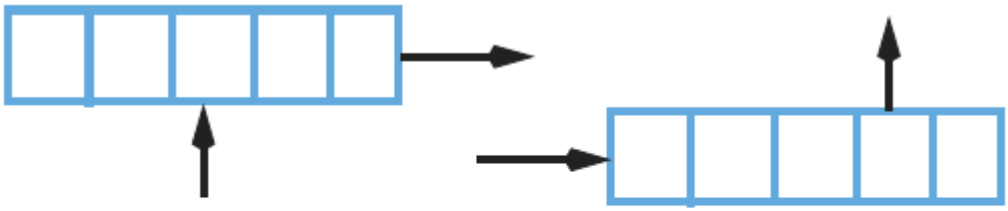
(leftbalanced is usually called complete)

”Incompatible” operations is typical

The way you implement an ADT affect how fast the operations can be performed.

This is very obvious with priority queues.

In a priority queue represented with a array you can get fast insert **or** deletemin depending on which algorithm you choose.



insert in the right place (slow) => delete is fast

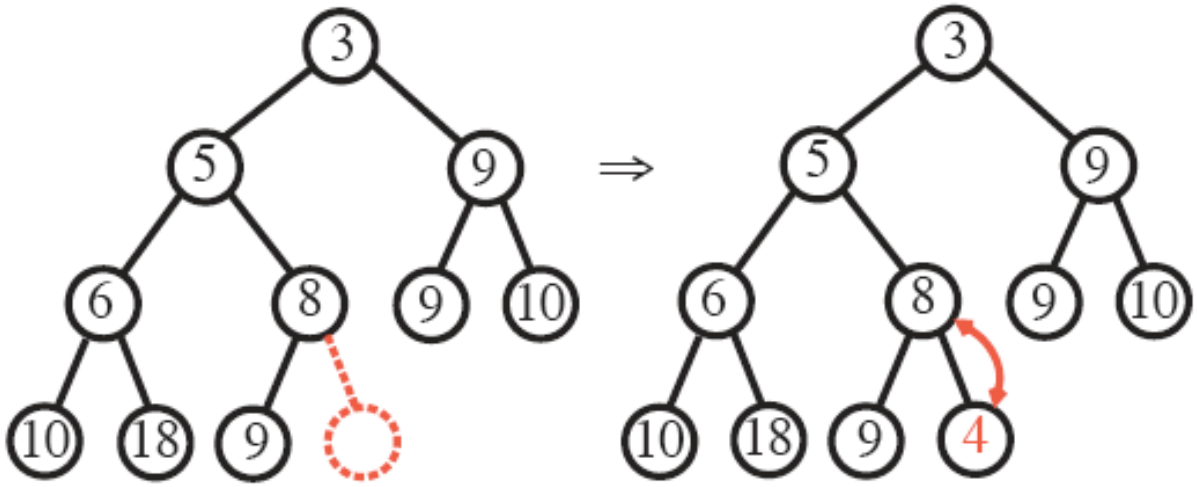
insert last (fast) => delete takes time

In a partially ordered leftbalanced (POL) tree both insert and deletemin can have O(logn) behaviour.

By selecting the implementation (representation and algorithms) we can influence performance and we can choose what to optimise.

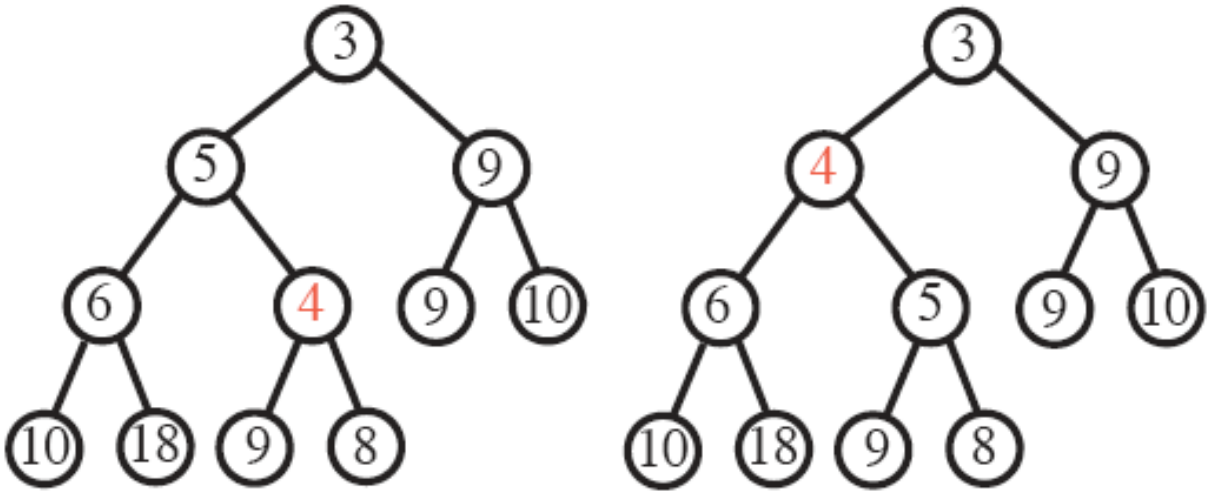
P-queue represented with a POL tree

Insert: assume that we have the following POL tree and that we want to insert the new element "4". We insert the element as far as possible to the left on the lowest level. If it's full we start a new level.

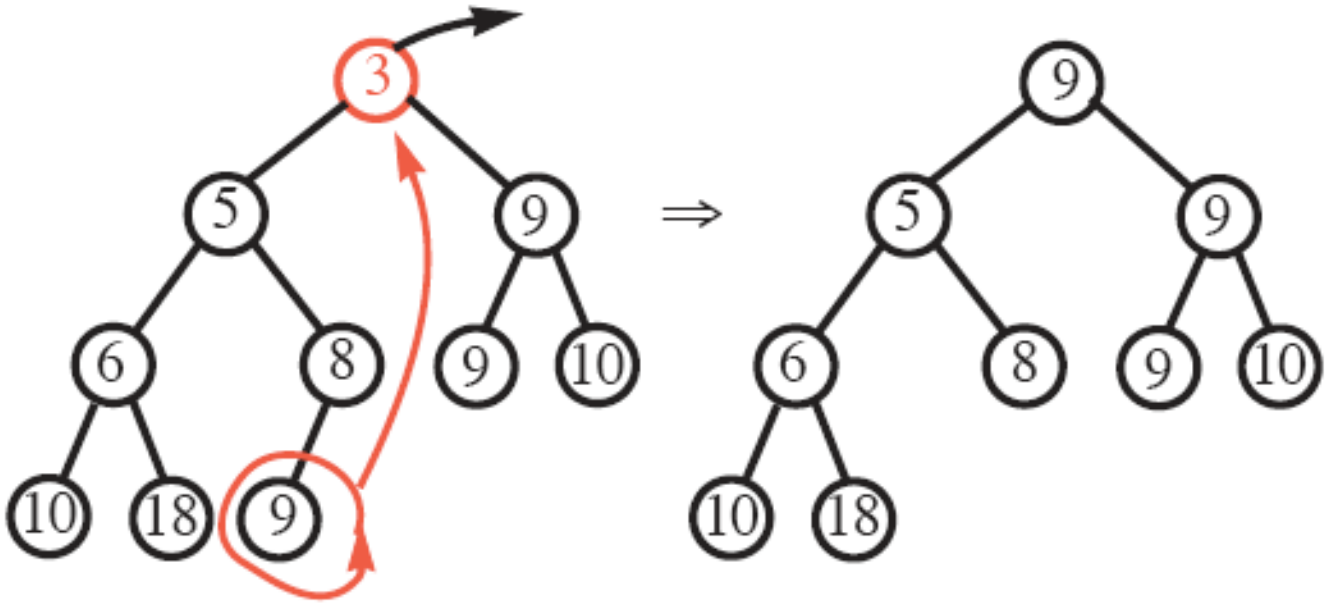


If the new element has a lower priority than it's parent we exchange them. We repeat this until the element is in the "right" place.

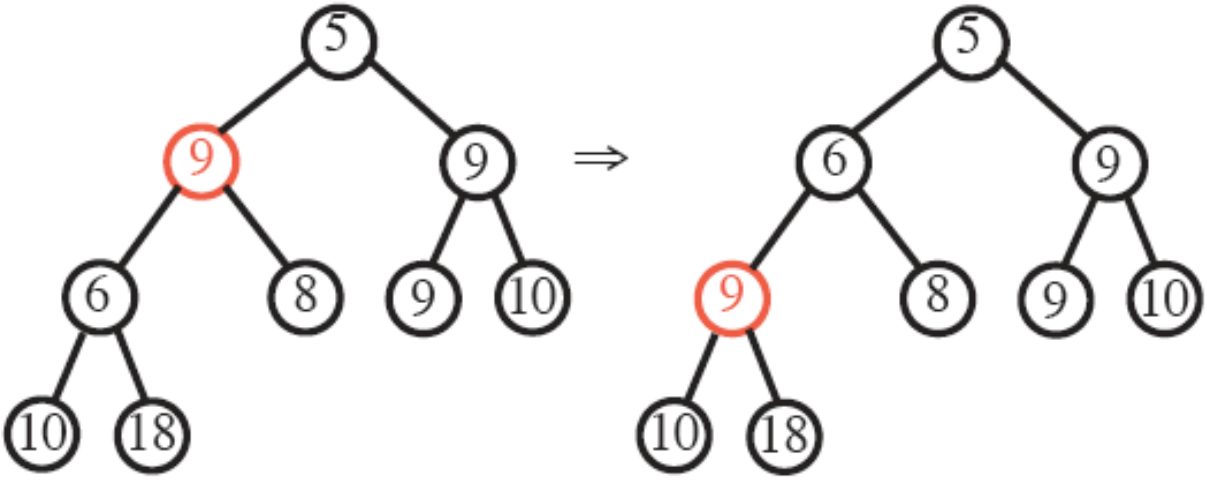
Complexity = O(logn)



Deletemin: We delete the smallest element. The smallest element is in the root. Move the rightmost leaf on the lowest level to the former root position. Also O(logn).



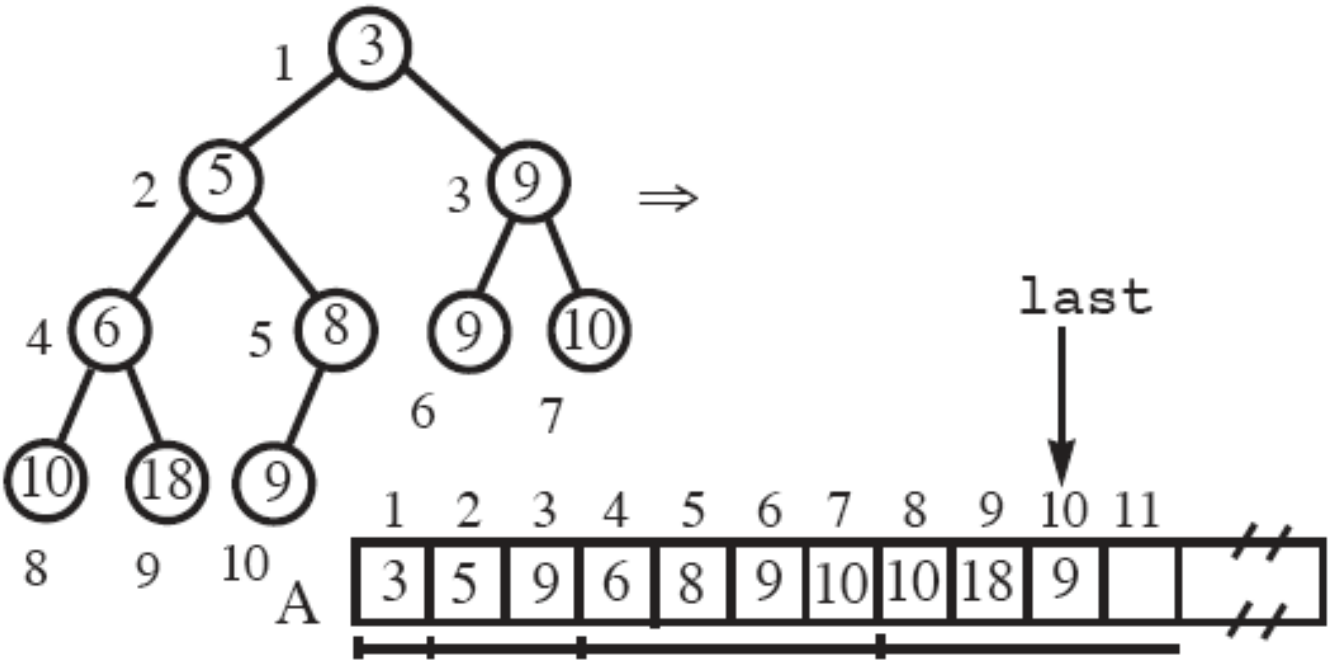
This element is then moved down the tree to it's "right" position by exchanging it with it's child with lowest priority.



How do we represent POL trees?

Pointers: - many pointers since we want to move both up and down the tree.

Heap: since we have a binary and leftbalanced tree we can store it level by level in a array.



A(1) contains the root.

If the child's exist we have:

Left child to A(i) is in A(2*i).
Right child to A(i) is in A(2*i+1)

Father to A(i) is in A(i div 2).

Ex: A(4)s children are in A(8) and A(9).
A(4)s father is in A(2)

Insert in the pq with a heap

PriorityQueue is represented with:

```
int last = 0;
array 1..max of <E> theData
```

Pseudocode:

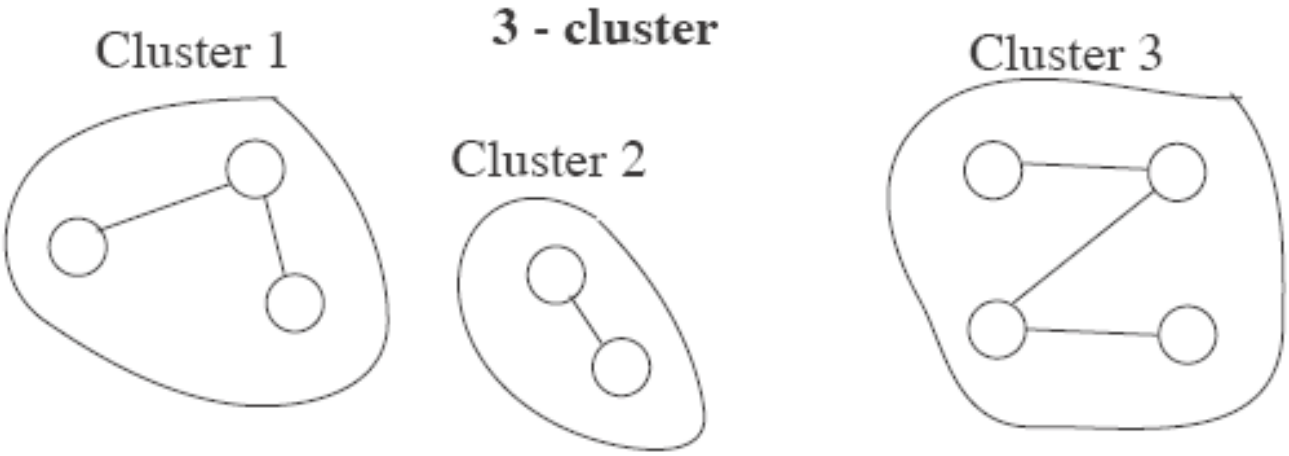
```
insert(<E> x)
    if last >= max then
        throw Pqueue_full
    else
        last = last+1;
        // theData(last) = x; // no don't add last
        int j = last-1;  // j
        // create a empty slot for the new data
        while j > 1 && x < theData(j/2)
            theData(j) = theData (j/2));
            j = j/2;
        end loop
        theData(j) = x; // now add the new one
    end if
end insert
```

Clustering - organizing similar object in groups by some distance function

A large distance <=> less similar

Example of distance:

- the number of years since two species diverged in the course of evolution
- the number of corresponding pixels at which two images differ



Given a distance function $d(p_i, p_j)$ on the objects divide them into k groups s.t. objects in the same group are close and objects in different groups are far apart.

Clustering - formal problem

A k-clustering is a partition of the objects into k non-empty sets $C_1, C_2, C_3, \ldots, C_k$

Assume that we

- have n objects $p_1, p_2, p_3, \ldots, p_n$ with a distance function $d(p_i, p_j)$ between each pair of objects
- like to divide them into k groups for given k
- seek a k-clustering with maximum spacing

The spacing of a k-cluster is the minimum distance between any pair of points in different clusters.

Kruskals algorithm can be used!

Stop once we have obtained k connected components i.e. just before Kruskal adds it's last k-1 edges or delete the k-1 most expensive edges from the mst produced by Kruskal.