Data Structures

November 9

Graphs

Objectives

Discuss the following topics:

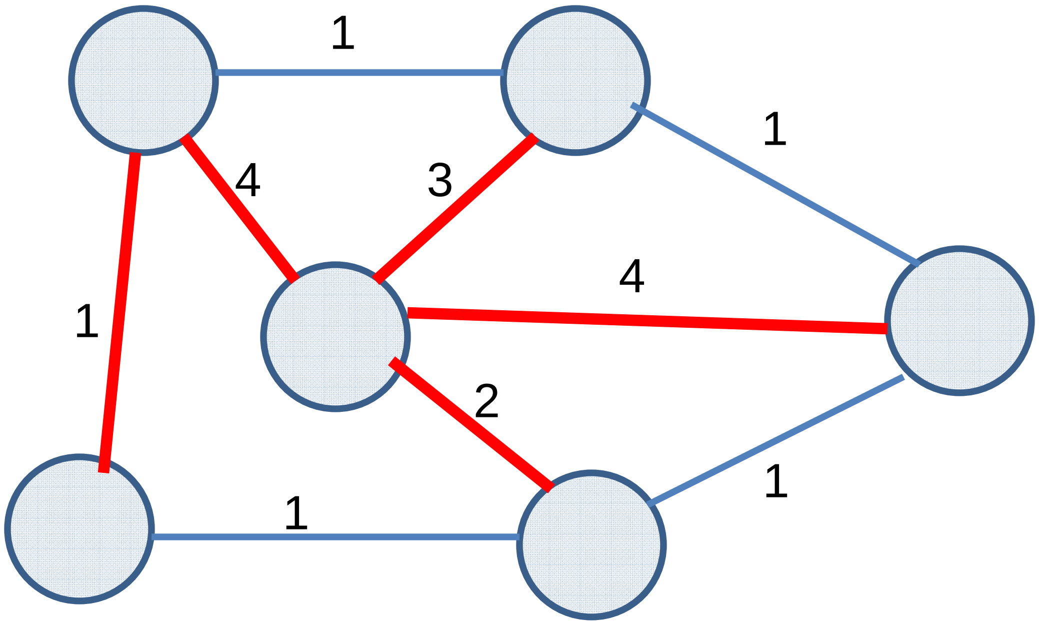
- Graphs; Graphs as ADT
- Graph Representation
- Graph Traversals (breadth first, depth first)
- Connectivity
- Bipartiteness
- Topological Sort (aka topological ordering)
- **Minimum Spanning Trees (Kruskal's and Prim's algorithms)**
- **Union-find data structure**
- **Priority queues for implementation of Prim's algorithm**
- **Clustering**
- **Shortest Paths**

Minimum Spanning Tree Problem

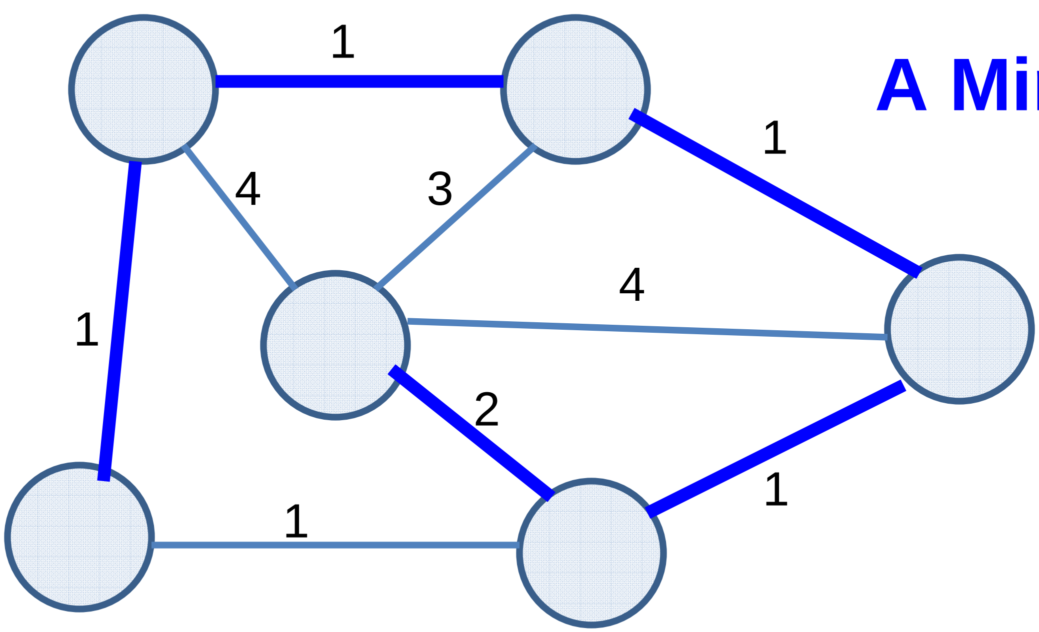
- Abstractly: $G=(V,E, w)$ is a weighted graph (i.e., $w: E \rightarrow \mathbf{R}^+$ (strictly positive real numbers, say)). Sometimes denoted by c (for cost).
- Find a subset T of E such that (V, T) is *connected*, and the total weight/cost $w(e_1)+w(e_2)+\ldots+w(e_s)$, where $T=\{e_1,e_2,\ldots, e_s\}$ is minimal. (Of course, we assume that G is connected from the outset, otherwise there are no solutions
- Statement: if T is a minimum weight (minimum cost) solution, then (V,T) is a tree – in other words, have MST

Minimum Spanning Tree Problem

- Also the weight function (cost function) could be non-negative, that is, the weights are either positive or zero can also be dealt with
- Assume first that no two weights are equal; a simple argument will also take care of the case of equal weights, once you know how to deal with mutually unequal weights.

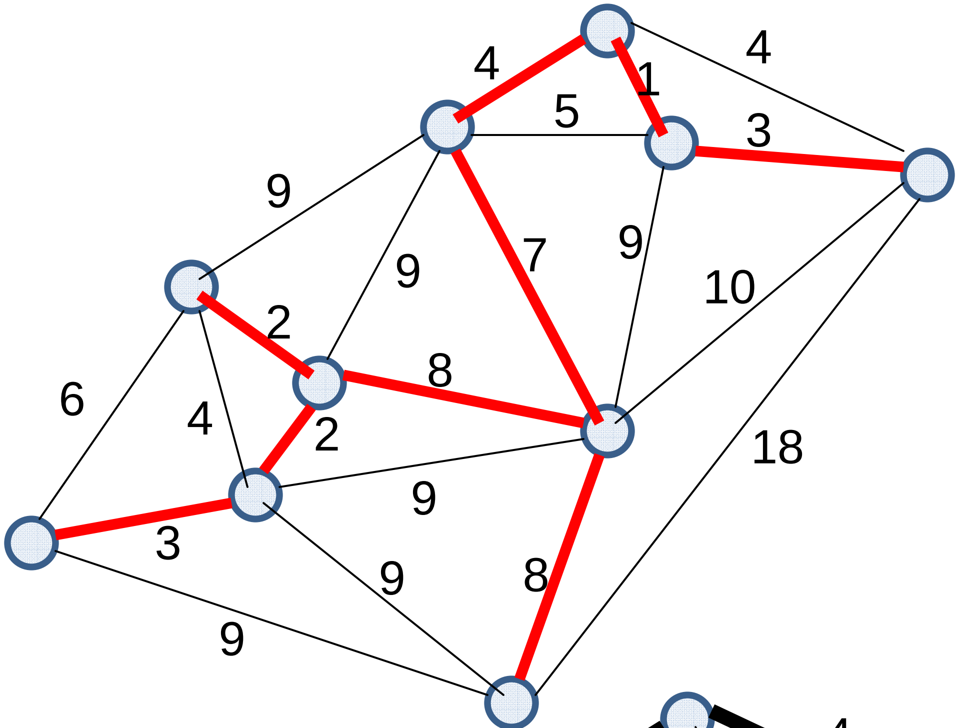


A Spanning Tree

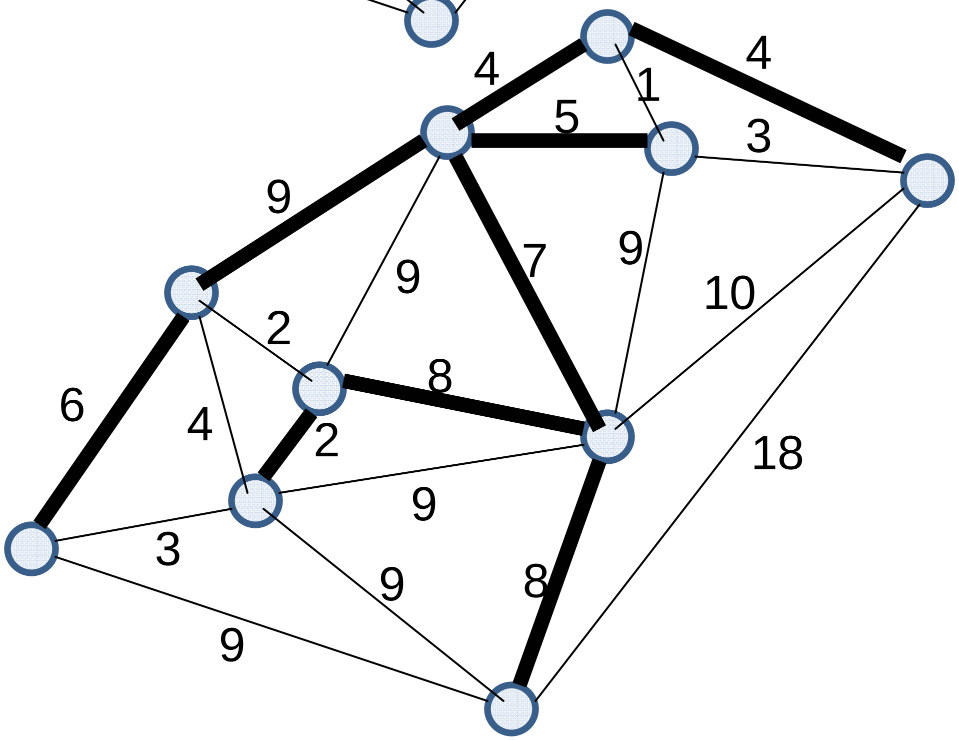


A Minimum Spanning Tree (aka MST)

(there are more MSTs and spanning trees)



An MST



A Spanning Tree

(More elaborate examples)

Definition: Spanning Tree and MST

- Let G = (V,E) be connected undirected graph: a ***spanning tree*** of the graph G is a subgraph G' = (V', E') of G such that V' = V and G' is a tree.
- In case G is also weighted: A ***minimum spanning tree*** or ***minimum weight spanning tree*** is then a spanning tree with weight less than or equal to the weight of every other spanning tree. (Weight of a tree: sum of the weights of all its edges.)

Some reasons to study MSTs

- Gives rise to one of the methods to cluster data (will be discussed next time)
- Connection with Traveling Salesman Problem: find a Hamilton Cycle in a graph with positive weights assigned to its edges that the sum of the weights of its edges of the cycle is minimal (Hamilton cycle of a graph is a cycle which visits each node of the graph). It is not known whether this problem possesses a polynomial time algorithm. (Since day and age a polynomial time algorithm is dubbed efficient algorithm by computer scientists.) Let the length/weight of minimal Hamilton cycle be denoted by h and by m we denote the weight of an MST, then $m \leq h$. It is easy to see that an MST gives rise to a tour with weight/length 2m; in this tour each road is traveled twice. (An MST of a graph can be computed efficiently as we will see below.)
- And many more apps

Minimum Spanning Tree Problem

- Many greedy algorithms
- Kruskal: start without any edges, build spanning tree by successively inserting edges from E in order of increasing cost – always insert an edge unless it creates a cycle, in this case discard the edge and proceed with the next cheapest edge available.
- Prim: start with root s and grow a tree from s outward. At each step add the node that can be attached as cheaply as possibly to the partial tree we already have.

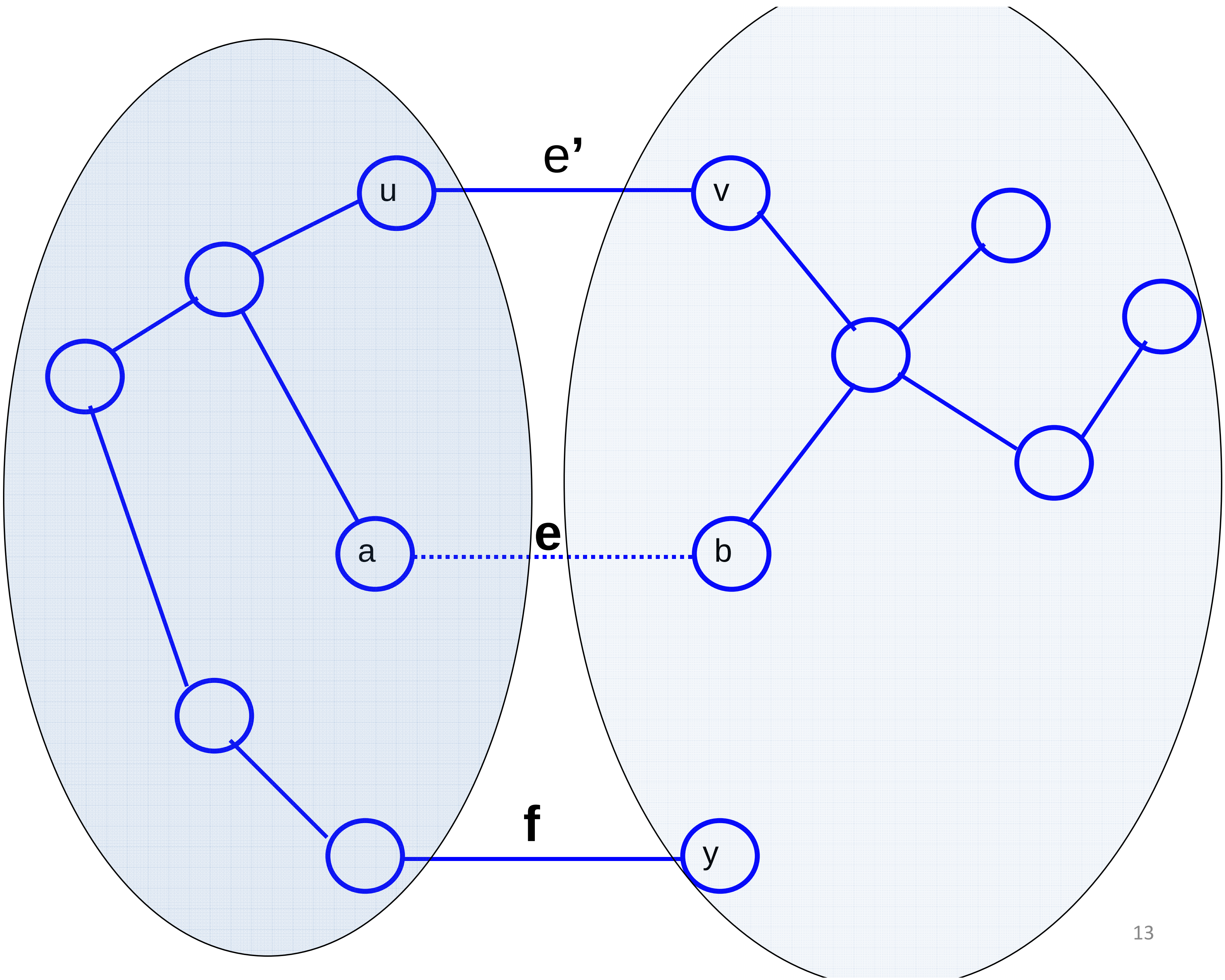
Minimum Spanning Tree Problem

- Prim: start with root node s and grow a tree from s outward. At each step add the node that can be attached as cheaply as possibly to the partial tree we already have.
- Maintain a subset S of V on which a spanning tree T_S has been constructed so far. Initially, S={s}. In each iteration, S grows by *one node* that has the *smallest* attachment cost:

Given S we consider: A_S = { weight((u,v)) | u in S and (u,v) is an edge of G and v not in S}. Let (u_S,v_S) in E with u_S in S and v_S in V\S be such that weight $((u_S,v_S))$ = min(A_S). Then we add v_S to S and the edge (u_S,v_S) is added to the growing spanning tree T_S.

Minimum Spanning Tree Problem

- Third way: start with the full graph (V,E) and begin deleting edges in order of decreasing cost; starting from the most expensive edge, we encounter edge e; delete it, if the remains are still connected
- **Cut Property**: assume all edge weights are different. Let S be a subset of V such that $S \neq \emptyset$ and $V \setminus S \neq \emptyset$; among edges with one end in S and the other in $V \setminus S$, the edge e has the smallest weight. Then e belongs to every minimum spanning tree


e'
u
v
a
e
b
f
y

Correctness of Kruskal (in words; skipped in the lecture; see pictures further on)

- Using the Cut property it is not difficult to show that Kruskal's and Prim's algorithms are correct
- Correctness Kruskal's algorithm (see pictures on the chalk board!): Let $G=(V,E)$ – (we assume that $|V| > 1$) -- be a undirected, connected graph. Clearly a connected, undirected graph has a spanning tree (for instance, take the bsf-tree); hence, in case the graph is weighted, the graph will have an Minimum Spanning Tree (MST). (In case the weights on the edges are mutually distinct, we will have a unique MST for the graph; a fact we are not going to use in our correctness proof.) Pick an MST of G: $M=(V, E_{mst})$. Kruskal algo takes as the set of vertices also V, and let $E_K = \{e_1,e_2,\ldots, e_r\}$ be set of edges produced by Kruskal algo. We will show that E_K is a subset of E_{mst}. Without loss of generality we assume $weight(e_1) < weight(e_2) < \ldots < weight(e_r)$. (This is the order in which Kruskal has added the edges.)

Correctness of Kruskal (in words; skipped in the lecture; see pictures further on)

- Let $e_1 = (v_1,w_1)$ be the first edge chosen by Kruskal. This means that e_1 is the edge with the smallest weight. Claim: e_1 is in E_{mst}. We will show this by using the cut property: Let $S=\{v_1\}$ and $V\setminus S$; since V is not a singleton, we also have that $V\setminus S$ is non-empty. Since e_1 is the edge with the smallest weight in the graph, it will a forteriori be, among the edges with one end point in S and the other in $V\setminus S$, the one with the smallest weight; by the cut property it belongs to any MST; hence e_1 is in E_{mst}.

Correctness of Kruskal **(in words; skipped in the lecture; see pictures further on)**

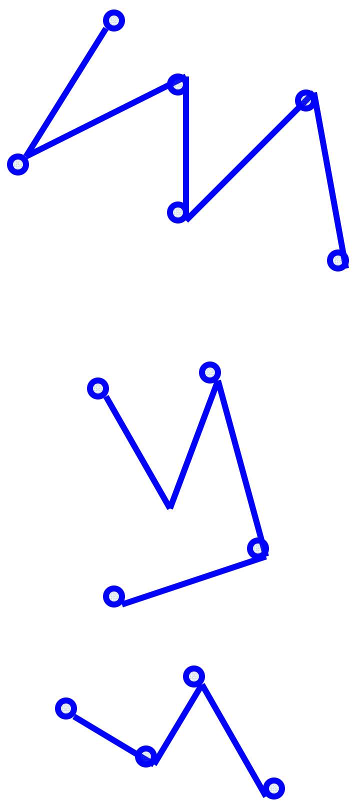
- Next we consider (i+1)st edge e_{i+1} = (v,w) which is chosen by Kruskal. Prior to this Kruskal has chosen the edges $e_1, e_2, \ldots, e_i$. Let N = { n in V | (n,x)= e_j or (x,n)=e_j for some j, 1≤ j ≤ i }. “all nodes which belong to one of the edges e_j ,1≤ j ≤ I”. Consider N U {v}. Let S be the connected component containing v of (N U {v}, {$e_1, e_2, \ldots, e_i$ }). Clearly w does not belong to S (for at stage i+1, (v,w) is chosen and w in S means that (v,w) will create a cycle).Thus e_{i+1} =(v,w) is an edge with v in S and w in V\S. We will show that e_{i+1} is the edge with the smallest weight among all the edges with one end in S and the other in V\S: let e’=(p,q) be an edge of G such that *p is in S* and weight(e**’**) < weight(e_{i+1}). Kruskal has considered e’ earlier than e_{i+1}; if q is not in S, then it would not create a cycle by adding it now or earlier (!) (earlier S may have been split up into more subcomponents which all the more would favor e’ for addition!); in short: such an e’ must have been already added. Thus: e_{i+1} is the edge with the smallest weight among all the edges with one end in S and the other in V\S; hence: e_{i+1} is in E_{mst}. Or *E_K is a subset of E_{mst}*

Correctness of Kruskal (in words; skipped in the lecture; see pictures further on)

- We have shown that E_K is a subset of E_{mst} . Secondly, (V, E_K) does not have cycles (Kruskal always adds edges which do not introduce cycles!) and thirdly (V, E_K) is connected. (Suppose by way of contradiction it is not, then V = S U V\S with S ≠ Ø and V\S≠ Ø. Since G is connected, there is an edge in G from S to V\S. But then such an edge with the lowest weight would have been added by Kruskal to E_K.)
- So (V, E_K) is a spanning tree and therefore an MST (since E_K is a subset of E_{mst} of some MST).
- qed

Correctness of Kruskal in pictures: show E_K is subset of E_{MST} for some MST of G

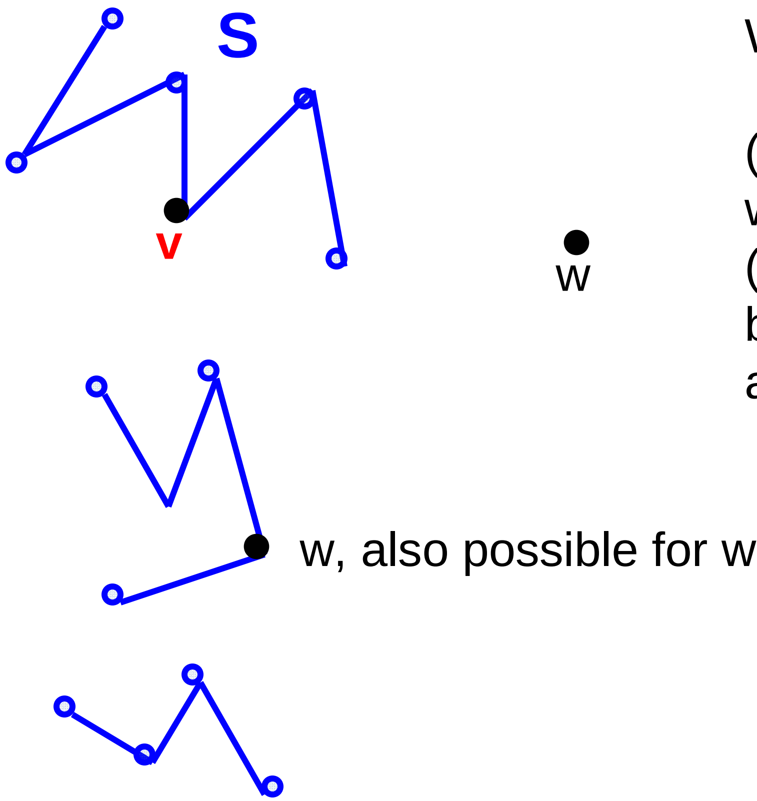
$e_1, e_2, \ldots, e_i$ (blue edges below) are already added (Kruskal is about to add edge e_{i+1} =(v,w), thus adding does not introduce cycles):



Two cases: a) v is already a node in the blue
b) v is not part of the blue picture

Correctness of Kruskal in pictures:

$e_1, e_2, \ldots, e_i$ (blue edges below) are already added (Kruskal is about to add edge e_{i+1} =(**v**,w), thus adding does not introduce cycles):



CASE a):
Where is w?

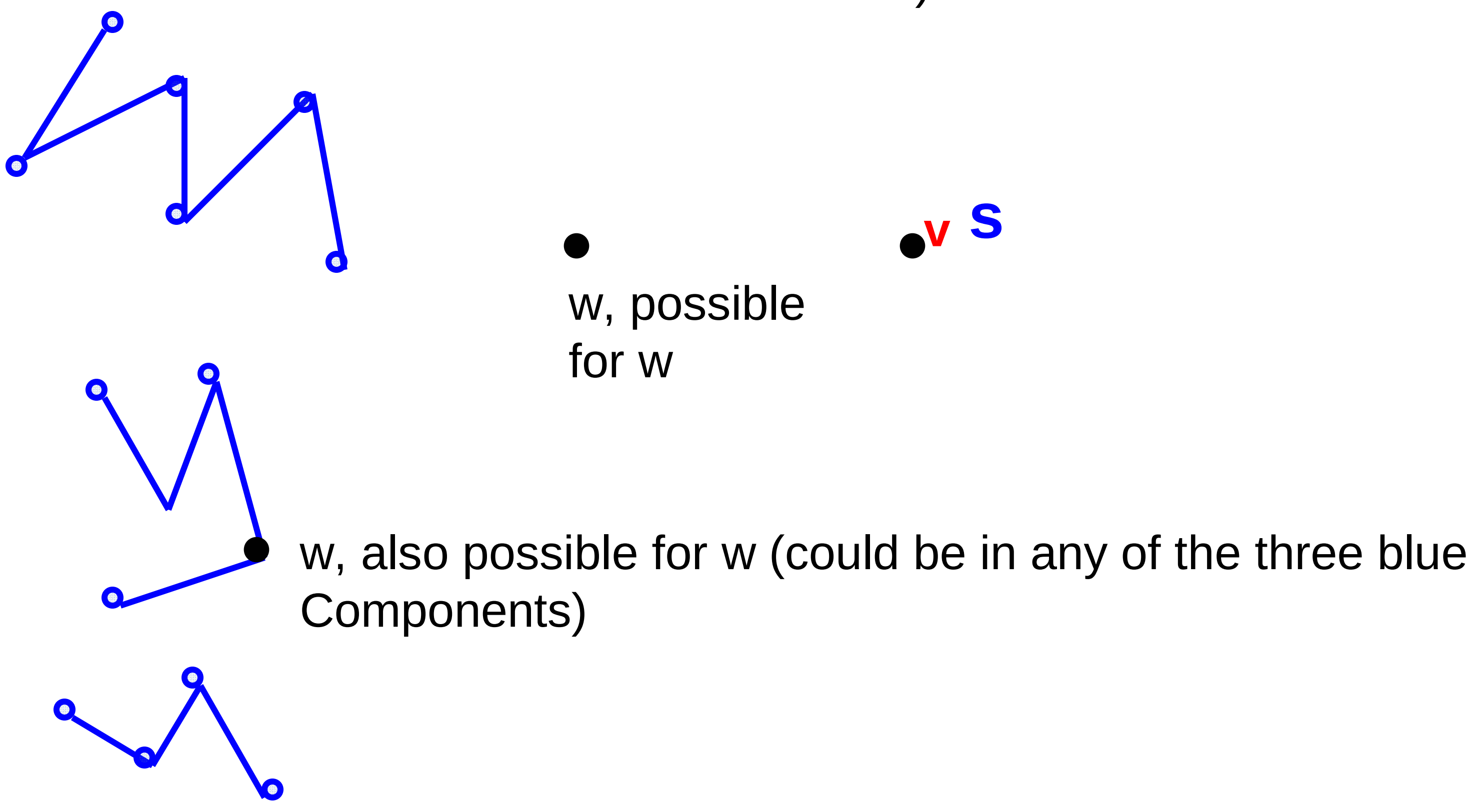
((**not** in the same blue component
where **v** resides
((**v**,w) is approved for adding
by Kruskal (if w in the same blue component
as v, adding edge (v,w) would introduce a cycle))

We consider the connected component of **v** made with the blue edges = S= upper blue component

Correctness of Kruskal in pictures:

$e_1,e_2,\ldots, e_i$ (blue edges below) are already added (Kruskal is about to add edge e_{i+1} =(**v**,w), thus adding does not introduce cycles):

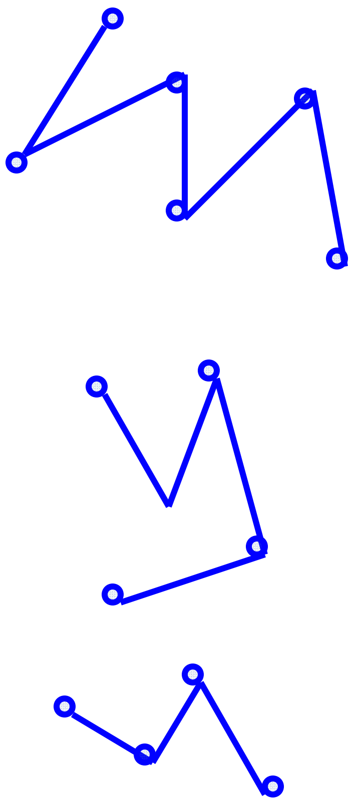
CASE b): v in none of the blue components



We consider the connected component of **v** made with the blue edges = S= {**v**}, will be a singleton

Correctness of Kruskal in pictures:

$e_1, e_2, \ldots, e_i$ (blue edges below) are already added (Kruskal is about to add edge e_{i+1} =(**v**,w), thus adding does not introduce cycles):



Summary: in all cases we get
v in S and w in V\S, and no edge with one end in S
and its other in V\S has been encountered before
by Kruskal

(i.e., none of the edges e' with
weight(e') < weight (e_{i+1})
are straddled over S and V\S),
otherwise Kruskal would have already added them,
since such and edge e' would not have created a cycle
at the moment, and a forteriori also not earlier in the game;)

That means: (v,w) is the cheapest edge with v in S and w in V\S
, by the Cut Property (v,w) belongs to any MST of the graph G

Still need to show:
(V, E_K) does not cycles (by construction) and also connected (easy)

Correctness of Prim

- It is straightforward from Cut Property that Prim's algorithm adds edges belonging to every MST.
- Recall that at each stage the set S and tree T_S are grown as follows:

Given S we consider: A_S = { weight((u,v)) | u in S and (u,v) is an edge of G and v not in S }. Let (u_S,v_S) in E with u_S in S and v_S in V\S be such that weight $((u_S,v_S)) = \min(A_S)$. Then we add v_S to S and the edge (u_S,v_S) is added to the growing spanning tree T_S.

This means: Given S we look at the edge e with one end in S and the other in V\S such that its weight is the smallest among edges of G with one in S and the other in V\S. And by the Cut Property we know that e belongs to any MST. In other words, the set of edges produced by Prim is a subset of the edges of any MST.

Clearly at each stage no cycles are introduced, and the T_S of each stage is connected as well. Thus upon termination T_S is connected and has no cycles. Upon termination all nodes of G have been included in S (T_S). Upon termination we have a spanning tree T_S whose weight is smaller or equal to the weight of any MST. Thus T_S is an MST.

Implementation of Kruskal: use Union-find (aka disjoint set data structure)

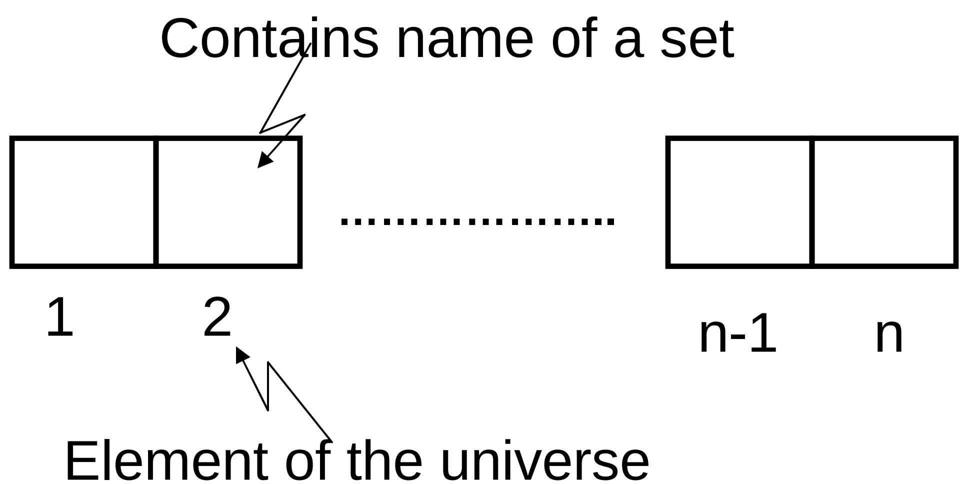
- Kruskal algorithm and the **Union-Find data structure** which can be used for the implementation
- **Union-Find**. Maintain disjoint sets: given a node u the operation find(u) will return the name of the set containing u.
- Find can be used to test whether nodes u and v are in the same set (test: find(v) == find(u)).
- The data structure has also an implementation of an operation union(A,B) which takes two sets A and B and merges them to a single set.
- These operation can be used to maintain connected components of an evolving graph.

Union-find and Kruskal

- For a node u, find(u) will return the connected component containing u
- If we add an edge (u,v) to the graph (that is, forest in Kruskal's algorithm), test first whether u and v are in the same connected component (test f(u)==f(v)). If not, then merge find(u) and find(v): union(find(v),find(u)).
- In summary Union-Find data structure:
  – makeUnionFind(S) on a set S: returns a data structure where all elements of S are in separate sets. Make it O(n) with n=|S|.
  – find(u), in O(log(n))
  – union(A,B) (in O(log(n)))

Discussion of three implementations Union-find (aka disjoint set data structure)

- As an array: (universe has n elements: {1,2,..,n})

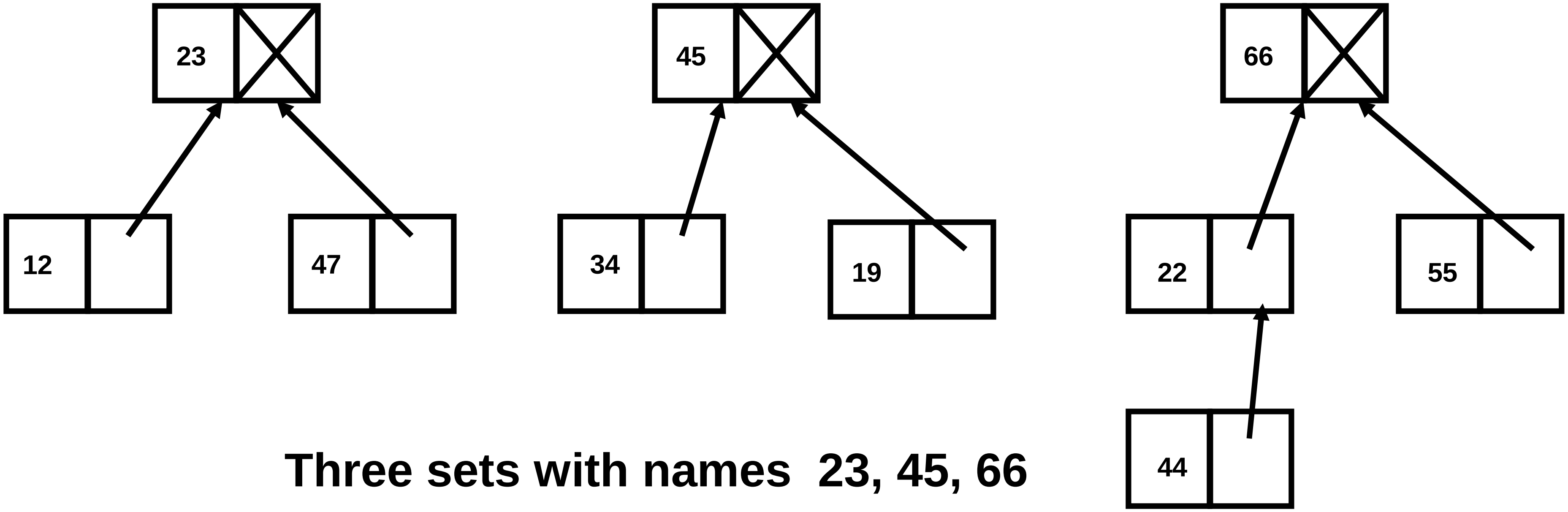


Name of array: component

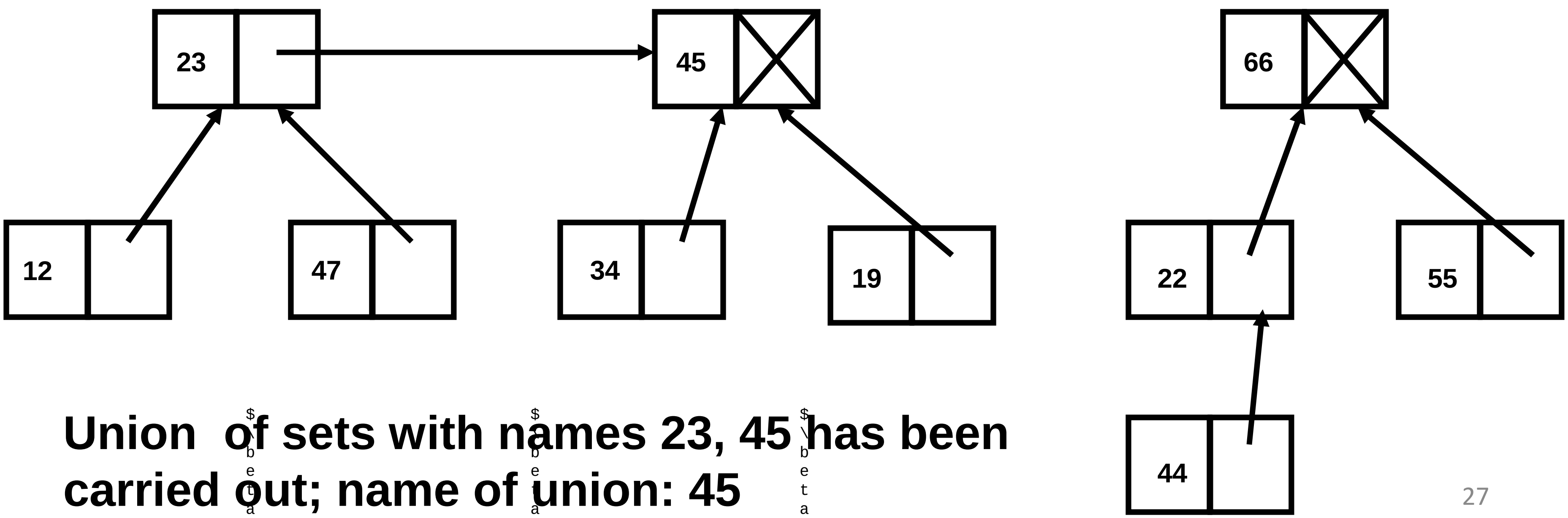
Find: O(1)
makeUnionFind: O(n)
Union: any sequence of k operations
Takes at most O(k log(k))

Discussion of three implementations Union-find (second way)

- Alternate implementation: use pointers.
- Each node v of the universe S will be contained in a record with pointer to the name of the set containing v.
- As names use elements of S (universe)
- MakeUnionFind(S): intialize a record for each element v in S with a pointer to itself (or if you wish a null pointer), to indicate that v is in its own set
- Union for sets A and B: assume for the name of A we used node v in A; name of set B is node u in B; we let either v or u be the name of the combined set; assume v will be the name: we update u's pointer to point to v – don't update pointers at the other nodes of set B.
- Union is now O(1)
- Find O(log(n))

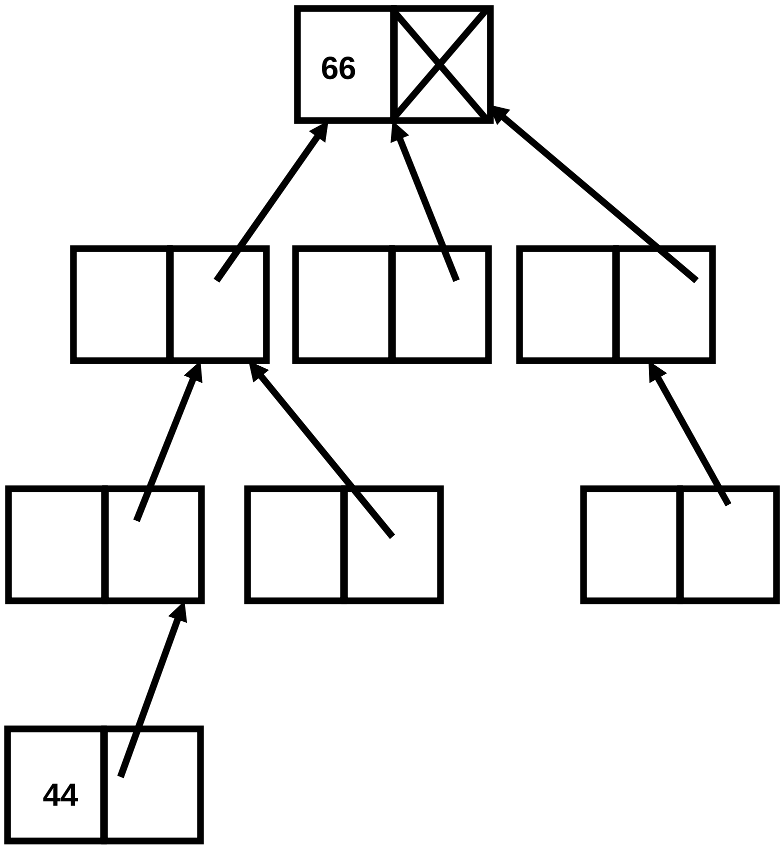


Three sets with names 23, 45, 66

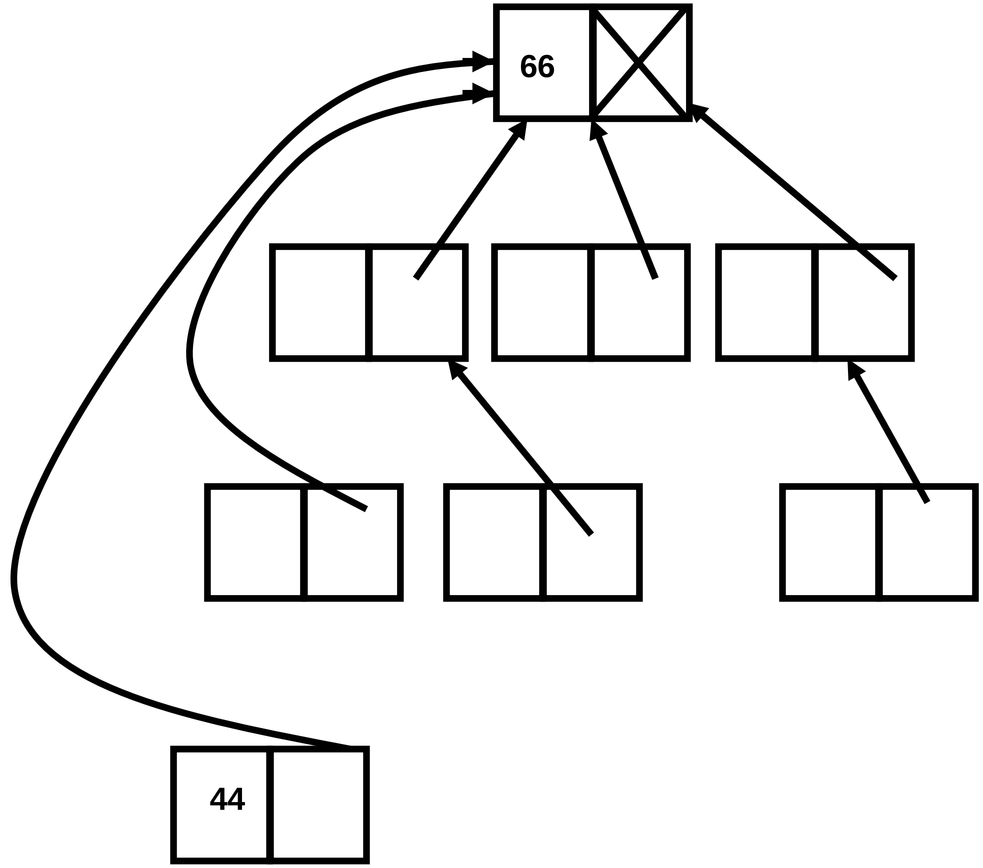


Union of sets with names 23, 45 has been carried out; name of union: 45

Discussion of three implementations
Union-find (third way: optimization in the second way)



Instance of union find data structure



After operation find(44) with path compression

Implementation of Prim's MST algo

- In order to find the node v not in currently in S (and edge) which needs to be added to the current S (respectively current tree T_S) quickly, maintain the attachment cost

a(v):= **min** $_{\textbf{e=(u,v) with u in S, e in E}}$ **weight(u,v)**, for each node v in V\S. Keep the nodes v in a priority queue with attachment costs a(v) as keys.

In case a node v in V\S is added to the current S: update the keys of w in V\SU{v} as follows: if

(v,w) is not an edge a(w) of G a(w) is the same as before; if (v,w) is an edge of the graph G, then update as follows

a(w) $\leftarrow$ min(a(w), weight((v,w)))

::: use priority queue implemented with a heap; use also the operation exchangeKey (and extractMin, insert)

Eliminating the assumption of mutually unequal weights in the MST algo's

- Suppose we have a graph in which it happens that some edges have the *same* weight
- What we can do add extremely small numbers to edges of equal weight so that they will have different weight
- ((Example: suppose you have three edges with weight 1 and all other edges have mutually different weights. Let m be the minimum absolute difference between unequal weights (suppose 1,1,1, 3, 4.5, 7, 10; then minimum: is 1.5 among pairs of unequal weight). 1,

$1+2^{-L}1.5$, $2^{-(L+1)}1.5$, 3, 4.5, 7, and 10 are the new weights for extremely large number L.))

Denote the old weight function by weight(..) and the new by

$weight_P$(..). We now find MST T with respect to the perturbed weight function $weight_P$(..). Now suppose T**'** is a spanning tree and suppose with respect to the old weight function weight(T) > weight(T**'**) but then it **cannot** be $weight_P(T) \leq weight_P(T')$ for a small enough perturbation. In other words T is not an MST wrt to the $weight_P$(..) function, contradiction. Thus T is MST also with respect to old weight function.