ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

M.Sc. Program in Computer Science
Department of Informatics

Design and Analysis of Algorithms

Minimum Spanning Trees

Vangelis Markakis

markakis@gmail.com

Spanning Trees

- Suppose a telecommunications company wants to build a network
- The network needs to connect some designated locations $V = \{v_1, v_2, ..., v_n\}$
- For each pair of locations (v_i, v_j) either we build a direct link or we would have to ensure there is a path through other nodes that connects them
- If we build the link (v_i, v_j), it has a cost of w_{ij}
- Requirement: Find which links to establish so that for every pair of nodes there is a path connecting them
- Optimization: Build the network as cheaply as possible

Spanning Trees

- Modeling this as a graph problem:
- G = (V, E), undirected graph, with weights on the edges
- Goal: Find a subset of the edges $T \subseteq E$, so that the subgraph (V, T) is connected, and such that T is of minimum cost

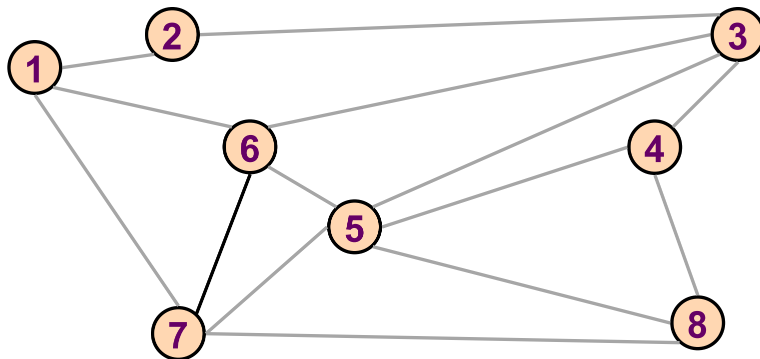
Implications:

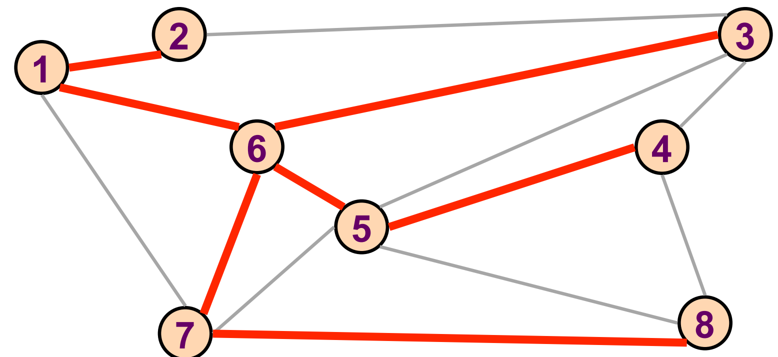
- T must be a tree
- No use in having cycles
- If there exists a cycle, remove an edge: the graph would still be connected with a cheaper solution
- How do trees look like?

Spanning Trees

Let T = (V, A) be a sub-graph of G = (V, E). The following are equivalent :

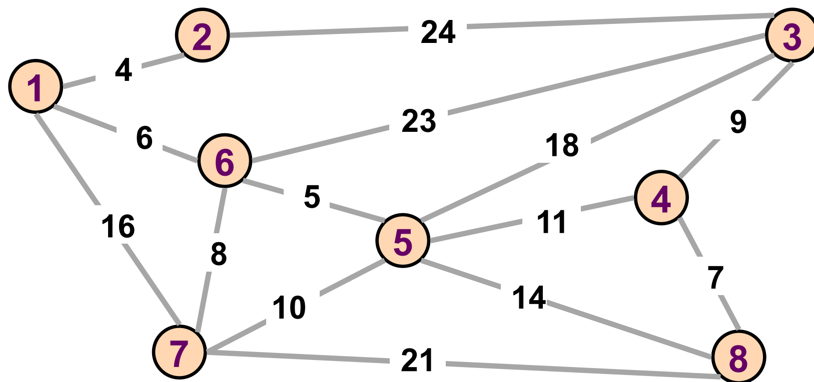
- T is a spanning tree of G.
- T is acyclic and connected.
- T is connected and has |V| - 1 edges.
- T is acyclic and has |V| - 1 edges.
- T is minimally connected: removal of any edge disconnects it.
- T is maximally acyclic: addition of any edge creates a cycle.
- T has a unique simple path between every pair of vertices.



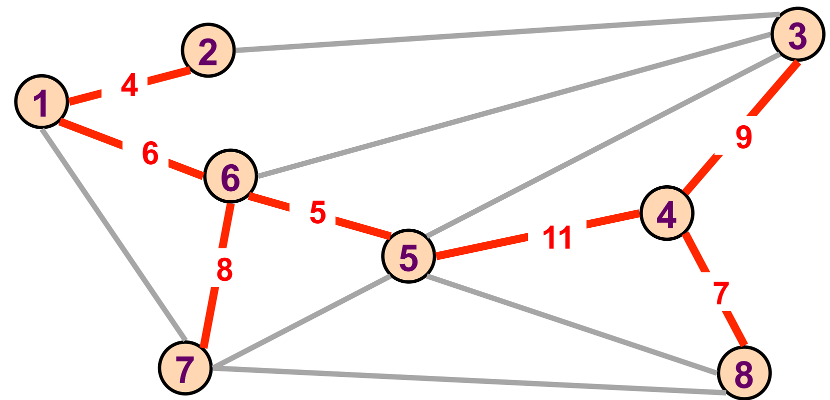


Minimum Spanning Tree

A MST of a weighted graph G is a spanning tree of G whose sum of edge weights is minimized



G = (V, E)



T = (V, A) w(T) = 50

Theorem [Cayley, 1889]: There are n^{n-2} possible spanning trees

➔ Exhaustive search is out of the question!

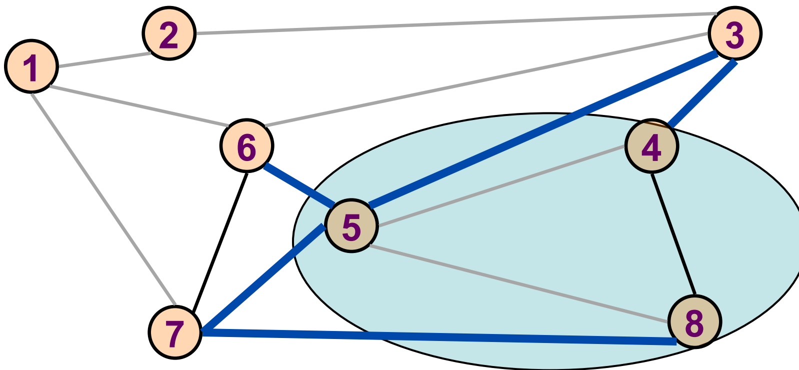
Minimum Spanning Tree

- We will try to design a greedy algorithm
- As we will see, many greedy approaches work here
- Need first to understand the structure of the problem
- The algorithms we will see are iterative, picking one edge in each round
- Each edge makes some more progress towards eventually building a spanning a tree

Cuts in graphs

A cut of a graph G = (V,E):

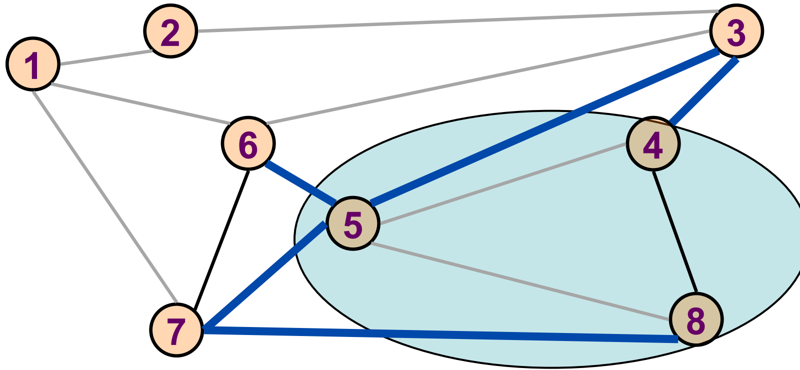
- A partition of the vertices into two groups S and V-S
- C = the set of edges with one endpoint in S and the other in V-S
- The cut is also denoted as (S, V-S)



S = {4, 5, 8}
C = { (5, 6), (5, 7), (3, 4), (3, 5), (7, 8) }

Obviously, for any cut (S, V-S), there must be an edge of a MST crossing the cut

A useful property



S = {4, 5, 8}
C = { (5, 6), (5, 7), (3, 4), (3, 5), (7, 8) }

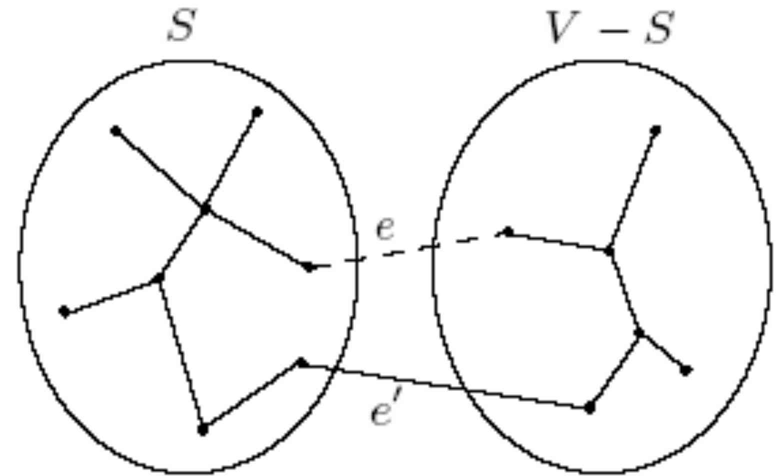
The Cut property:

- Let X be a subset of edges that belong to some MST
- Let C be any cut partitioning the graph into (S, V-S) such that X has no edges in C
- Let e be the cheapest edge in C

Then X U {e} is part of some MST

Proof of the Cut Property

- Edges in X are part of some MST T
- If e ∈ T then we are done
- Assume that e ∉ T
- Add e to T
  - e together with T creates a cycle K
  - there is one more edge e’ ∈ C ∩ K
  - w(e’) ≥ w(e)
- Remove e’ from T
- Consider T’ = T U {e} – {e’}
- Claim: T’ is a spanning tree
  - It is connected and has n-1 edges
- W(T’) = W(T) + w(e) – w(e’) ≤ W(T)



Since T is a MST, it must be that W(T) = W(T’) and T’ is also a MST

A Generic MST Algorithm

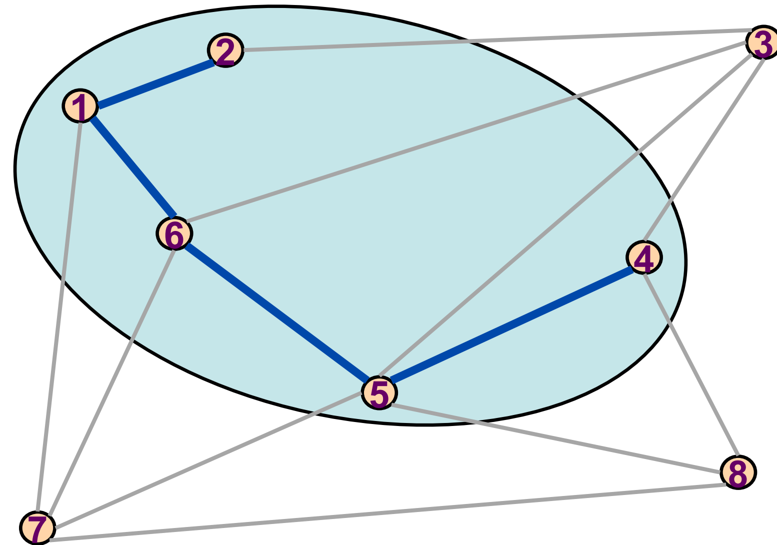
```
X:= ∅ ; //edges of a MST selected so far
while |X| < |V|-1 do
    { Choose a subset S of V such that X has
      no edges between S and V-S;
      Let e be the minimum weight edge
      between S and V-S;
      X:= X ∪ {e}          }
```

Prim's MST Algorithm

- Apply the cut property for the cut S, V-S, where S is the set of vertices of a subtree of a MST (X is the set of edges of this subtree)
- Initialize S to be any vertex

In each step
an edge and a vertex
are added to the subtree

Repeatedly select the vertex with
the minimum distance
from the current subtree



Prim's MST algorithm

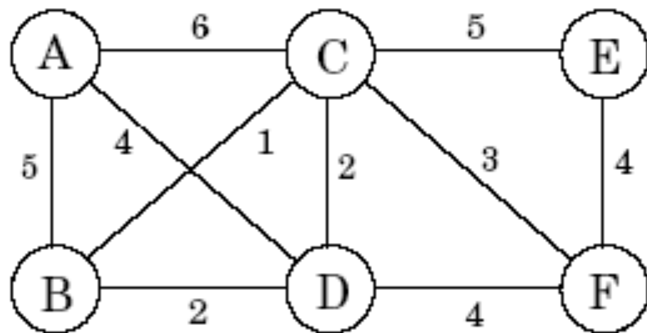
```
X:= ∅;//edges selected so far
S:={s} //s is any vertex
 while |X| < |V|- 1 do
        { Find the minimum weight edge,
          e=(u,v), between S and V-S;
          //u ∈ S, v ∈ V-S
          S:=  S U {v};
          X:= X U {e}           }
```

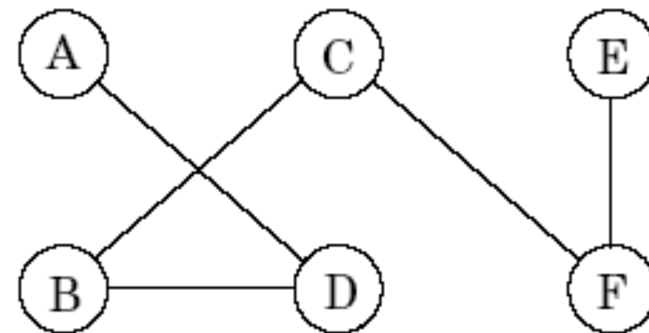
- How can this be implemented efficiently ?
- Need to select the vertex with the minimum distance from S

Prim's MST Algorithm

- A priority queue is again the right data structure
- Algorithm reminiscent of Dijkstra
- Maintain a priority queue for nodes not in S yet
- The key for each node v, not in S, is the minimum edge connecting it to S
- When we include a new node v in S:
  - Need to update the key for every neighbor of v outside S
  - Exactly same idea as in Dijkstra

Prim's MST Algorithm - example





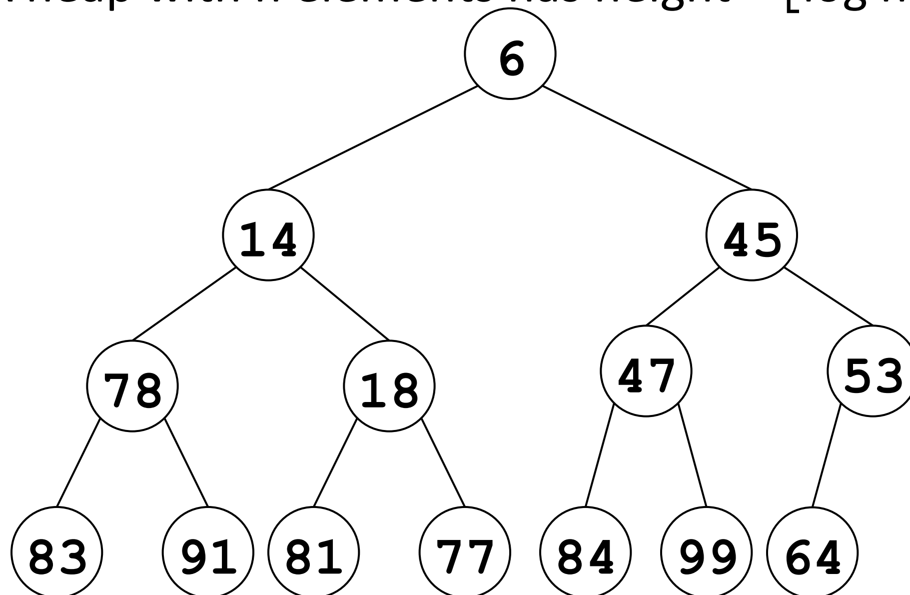
Set S	A	B	C	D	E	F
{}	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$5/A$	$6/A$	$4/A$	∞/nil	∞/nil
A, D		$2/D$	$2/D$		∞/nil	$4/D$
A, D, B			$1/B$		∞/nil	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

Priority Queues

Operation	Description
`min(Q)`	returns (a pointer to) the element of Q with minimum key
`insert(Q,x)`	adds element x to the queue Q
`delete_min(Q)`	deletes the element of minimum key from Q
`union(Q1,Q2)`	Combines queues Q1 and Q2 into one
`decrease_key(Q,x,d)`	Updates the queue by decreasing the key of element x by d
`delete(Q,x)`	deletes element x from Q

Binary Heaps

- A way to implement a priority queue
- A complete (but not necessarily full) binary tree
  - Filled on all levels, except possibly the last, where it is filled from left to right
- Min-heap property: the key of every child greater than or equal to parent's key (also say that the elements are *min-heap ordered*)
- Hence, min element is in root
- A heap with n elements has height = $\lfloor \log n \rfloor$, that is O(log n)



n=14

height=3

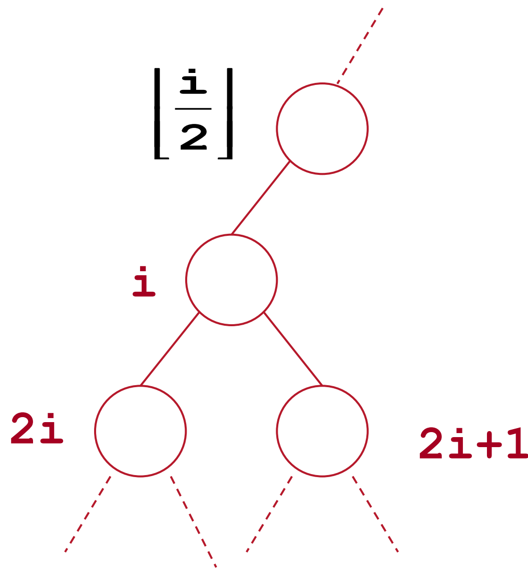
n=8
Height=3

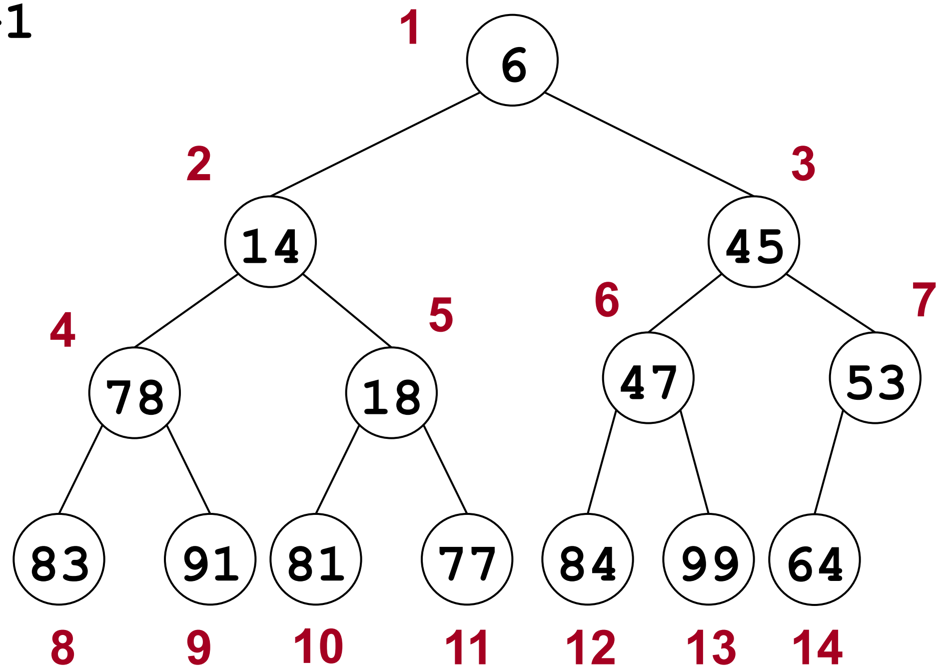
Binary heaps - array implementation

Use an array: start at position 1

no need for explicit parent or child pointers

- `Parent(i) = ⌊i/2⌋`
- `Left(i) = 2i`
- `Right(i) = 2i+1`

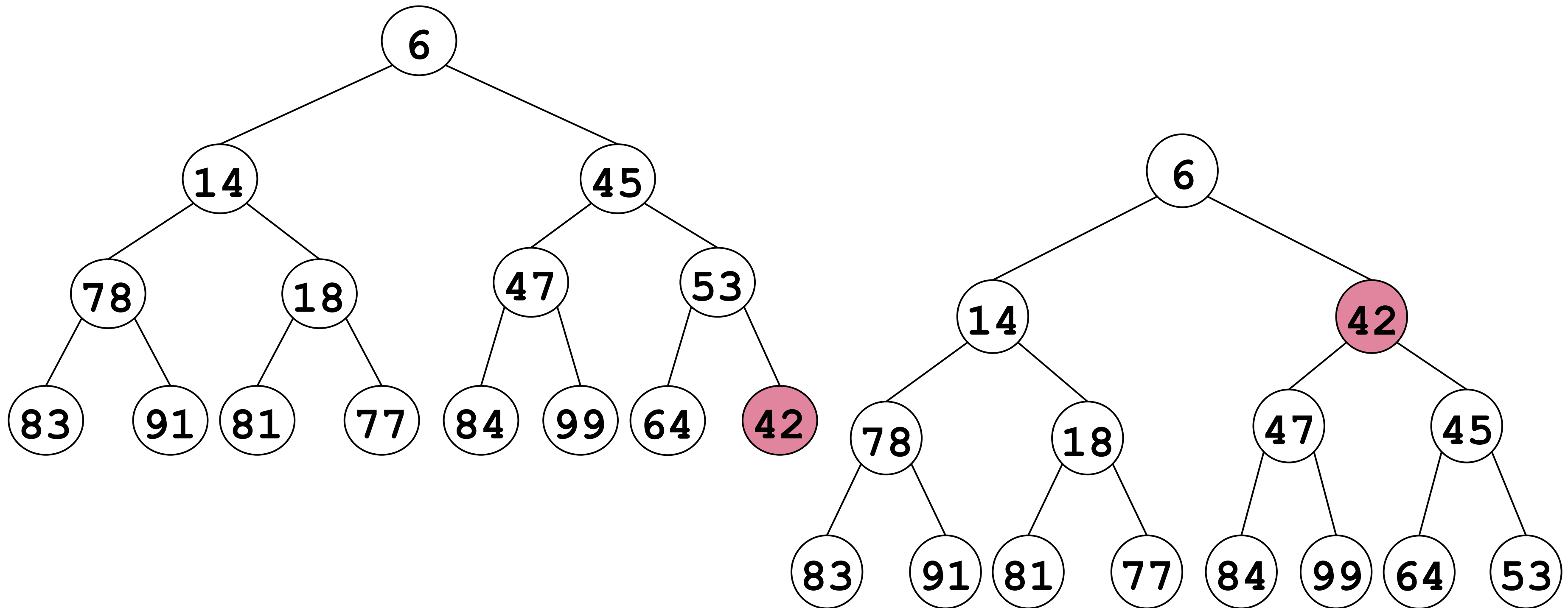




Binary heaps – `insert`

To insert a new element:

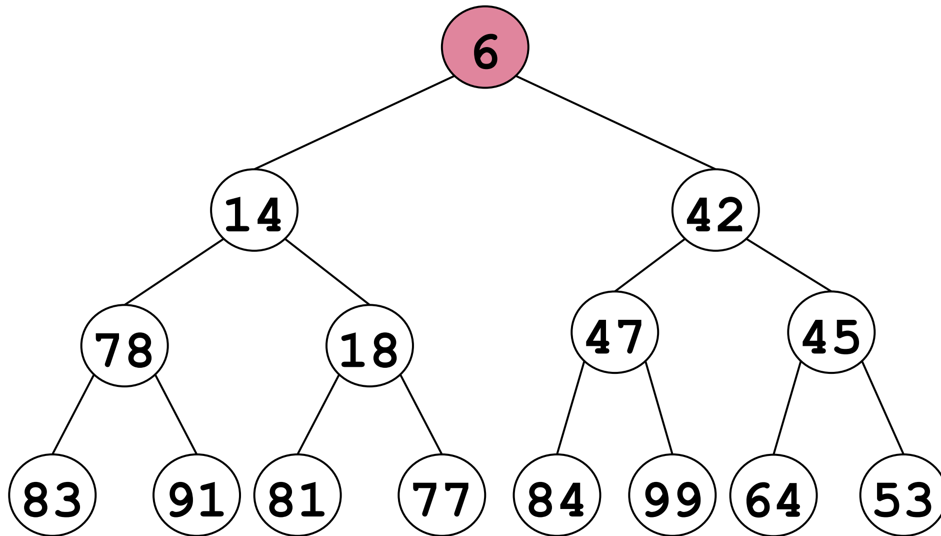
- First insert it into the next available slot at the end of the tree
- Bubble up until the heap property is restored

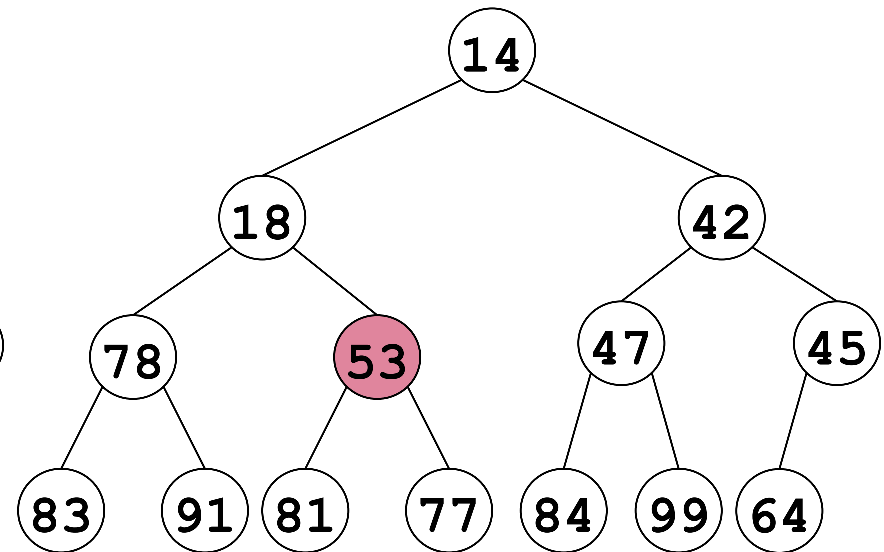


O(log n) operations

Binary heaps – `delete_min`

- Exchange root with rightmost leaf (last active position in the array)
- Bubble root down until it's heap-ordered





O(log n) operations

Binary heaps – etc

`union(Q1,Q2)`

- Suppose Q1 has n_1 elements and Q2 has n_2 elements
- No direct solution
- O(n) operations (with $n := n_1 + n_2$)

`decrease_key(Q,x,d)`

- Decrease key of x by d
- Bubble up until it's heap-ordered
- O(log n) operations

`delete(Q,x)`

- Decrease key of x to $-\infty$
- Run delete_min
- O(log n) operations

Binary heaps – summary

Operation	**Worst case**
`min(Q)`	**O(1)**
`insert(Q,x)`	**O(log n)**
`delete_min(Q)`	**O(log n)**
`union(Q1,Q2)`	**O(n)**
`decrease_key(Q,x,d)`	**O(log n)**
`delete(Q,x)`	**O(log n)**

Priority Queues – Summary

Operation	Binary Worst	Binomial Worst	Binomial Amortized	Fibonacci Amortized
min(Q)	O(1)	O(1)	O(1)	O(1)
insert(Q,x)	O(logn)	O(logn)	O(1)	O(1)
delete_min(Q)	O(logn)	O(logn)	O(logn)	O(logn)
union(Q1,Q2)	O(n)	O(logn)	O(logn)	O(1)
decrease_key(Q,x,d)	O(logn)	O(logn)	O(logn)	O(1)
delete(Q,x)	O(logn)	O(logn)	O(logn)	O(logn)

Prim's MST Algorithm: Complexity

# of Operations		Heap implementation		
		Binary	Binomial*	Fibonacci*
Insert:	n	O(logn)	O(1)	O(1)
delete_min:	n	O(logn)	O(logn)	O(logn)
Decrease_key:	m	O(logn)	O(logn)	O(1)

Binary heap: n logn + n logn + m logn ~ O(m logn)

Binomial heap: n · O(1) + n logn + m logn ~ O(m logn)

Fibonacci heap: n · O(1) + n logn + m · O(1) ~ O(m + n logn)

*** amortized**

A Generic MST Algorithm

The cut property implies that any algorithm of the following form works !

```
X:= ∅ ; //edges of a MST selected so far
while |X| < |V|-1 do
    { Choose a subset S of V such that X has
      no edges between S and V-S;
      Let e be the minimum weight edge
      between S and V-S;
      X:= X ∪ {e}          }
```

Kruskal's MST Algorithm

- Again we proceed in rounds
- This time the current solution after each round does not form a tree
- It forms a forest
- In every step, we pick an appropriate edge to add to the forest
- The addition of each edge connects 2 trees from the forest
- Until eventually we have a spanning tree
- Data structure for representing currently connected components: Union-find

Kruskal's MST Algorithm

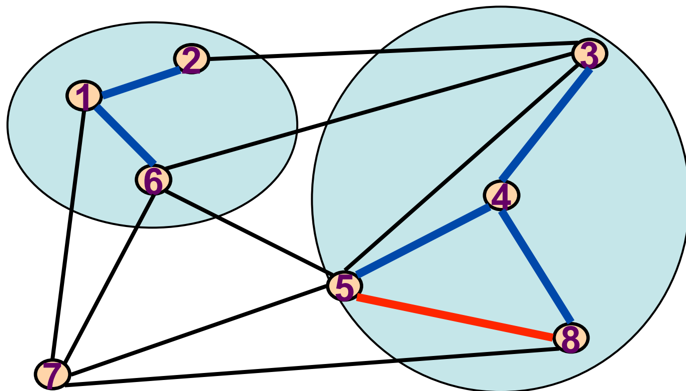
Suppose we consider adding an edge e = (u, v) during the algorithm

Let S_u = set of vertices in the tree that u belongs to (similar definition for S_v)

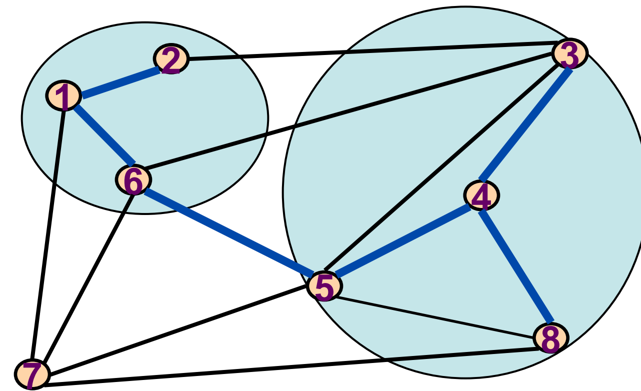
Case 1: If adding e creates a cycle, then ignore e

Case 2: else apply the cut property for the cut $(S_u, V - S_u)$ (or for S_v)

Hence: Repeatedly add the lightest edge that does not produce a cycle with the current subtrees



Case 1: e=(5, 8)



Case 2: e=(5, 6)

Kruskal's MST Algorithm

Case 2: Can we order the edges such that any edge $e = (u, v)$,
considered in this order, is the lightest edge
in the cut $(S_u, V - S_u)$ or the cut $(S_v, V - S_v)$?

YES: Simply consider edges in increasing order of weights:

Let w_i = weight of edge e_i

Sort the edges so that

$w_1 \leq w_2 \leq \ldots \leq w_m$

Kruskal's MST algorithm

```
X:= ∅;//edges selected so far
 Sort edges in increasing order of weights:
        w1 ≤ w2 ≤...≤ wm;
 i:=1;//ei = edge of weight ci
 while |X| < |V|-1 or i > m do
          { if ei does not create a cycle then
          //if ei=(u,v), then it is the lightest
          //edge in the cut (Su, V-Su)(same for Sv)
                 X:= X ∪ {ei};
                 i:=i+1       }
```

How can this be implemented efficiently ?

The Union-Find Data Structure

A data structure for maintaining a collection $\{S_1, S_2, ..., S_k\}$ of DISJOINT dynamic sets

Notation: if x is an element of some set, then S_x:= the set which contains x

<u>Supported Operations</u>

`make_set(x)`

Creates a new set S_x containing only element x;

x is not contained in any other existing set

`union(x,y)`

Unifies the disjoint sets containing elements x and y (S_x and S_y) into a new set;

Sets S_x and S_y are destroyed

`find(x)`

Returns the set S_x (from a collection of disjoint sets) containing x;

If $x, y \in S_i$, then `find(x)=find(y)`

Kruskal's MST algorithm

```
Sort edges in increasing order of weights: w1≤w2≤...≤wm;
X = ∅;
for each u ∈ V do make_set(u);
for i = 1 to m do
 {  let ei = (u,v) ;
    if (find(u) ≠ find(v)) then
        {    union(u,v);
             X ← X ∪ {ei}  }  }
```

UNION-FIND

`make_set(u)`: creates a set containing element u

`union(u,v)`: unifies the disjoint sets containing elements u and v

`find(u)`: returns the set (from a collection of disjoint sets) containing element u

Kruskal's MST algorithm

```
Sort edges in increasing order of weights: w1≤w2≤...≤wm;
X = ∅;
for each u ∈ V do make_set(u);
for i = 1 to m do
 {  let ei = (u,v) ;
    if (find(u) ≠ find(v)) then
        {    union(u,v);
             X ← X ∪ {ei}     }  }
```

Complexity

Sorting: O(|E| log |E|), that is O(| E| log |V|)

Union-Find: |V| make_set operations,
2|E| find operations,
|V|-1 union operations

In total: ??

Implementing Union-Find: Up-tree representation of sets

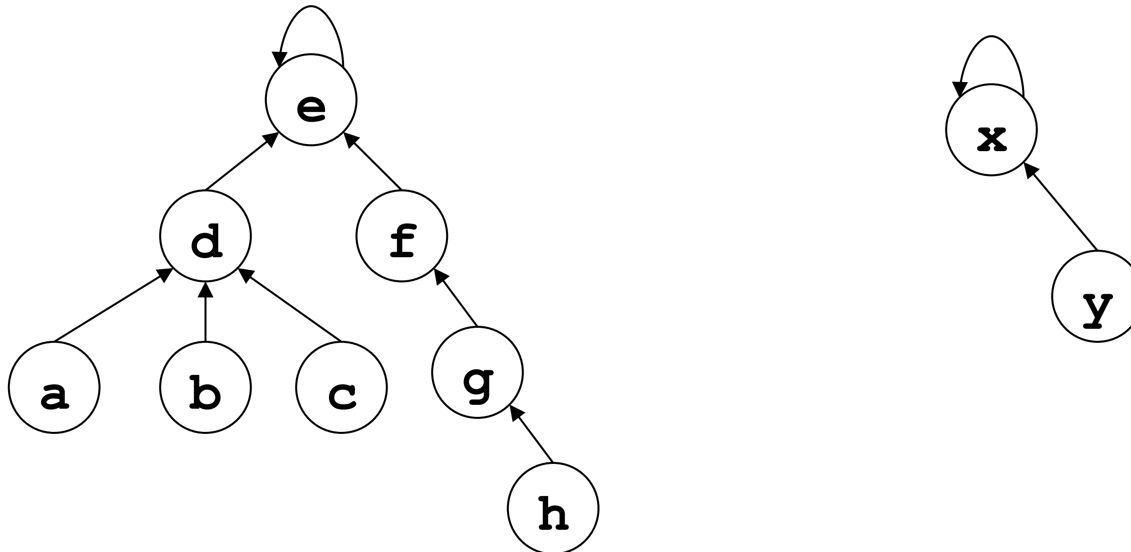
Every set S_i is represented by an up-tree

The root of the tree is the representative of S_i

Every node has a pointer to its parent

The root of the tree has a pointer to itself

p[x]: the parent of element x



1st approach

```
make_set(x)
{   p[x]:=x;    }
```

O(1), n operations

```
find(x)
{   while x!=p[x] do x:=p[x];
    return x;   }
```

O(n)

```
union(x,y)
{   a:=find(x);
    b:=find(y);
    p[a]:=b;  //or p[b]=a   }
```

O(n)

Hard `find`, easy `union`

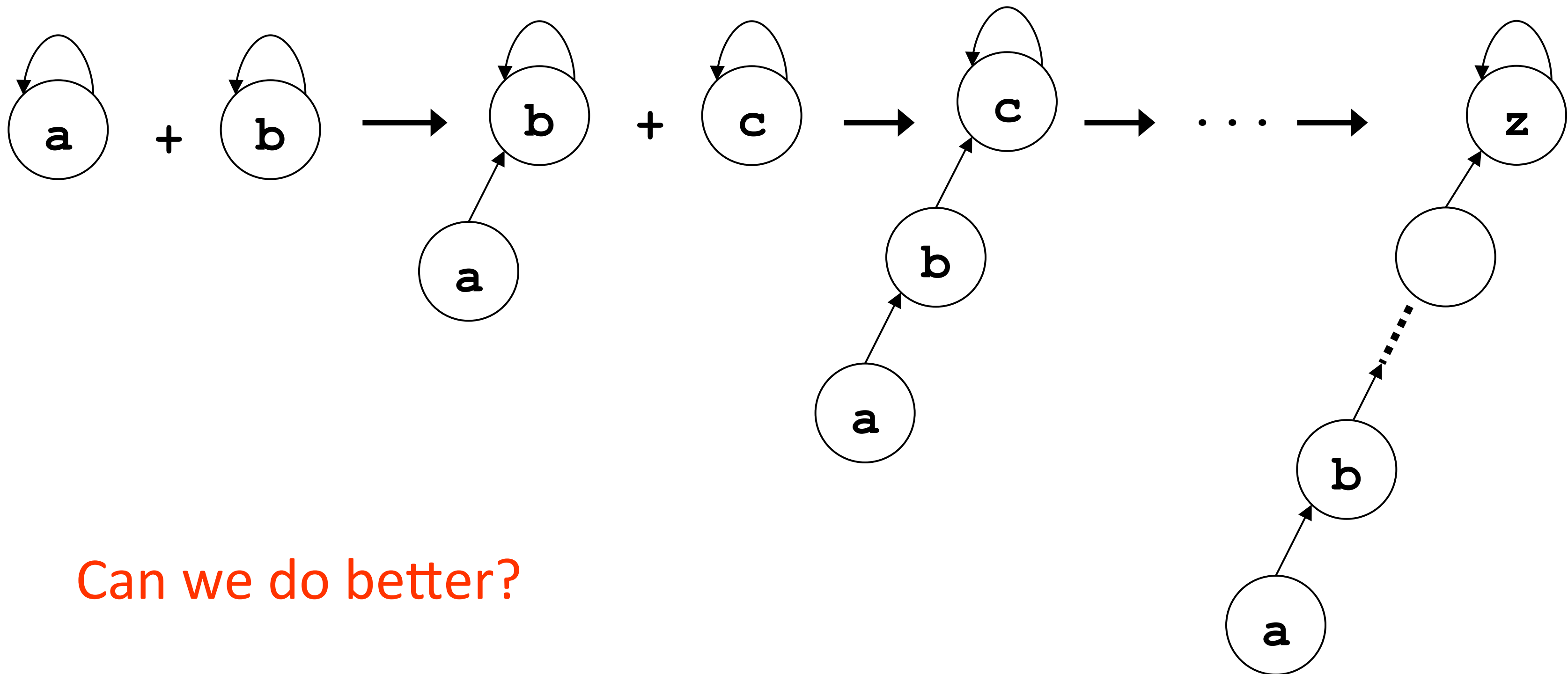
O(n) time per operation

1st approach

find(x,y) is not efficient

A tree may degenerate to a chain

Example



Can we do better?

2nd approach - Union by Rank

Make the root of the "larger" tree the parent of the root of the "smaller" tree...

<u>Union by rank</u>

`R[x]`: the rank of element x

a measure of the size of the tree "rooted" at x

`R[x]` is initiated to 0 by `make_set`

`R[x]` is updated only by `union`

if `R[a]=R[b]` then `R[new root]=R[b]+1`

We choose `R[x]` to be the height of the tree rooted at x

2nd approach - Union by Rank

The roots of the trees maintain their ranks, we do not update the rank of nodes that stop being roots

```
make_set(x)
{   p[x]:=x;
    R[x]:=0;    }
```

```
find(x)
{   while x!=p[x] do x:=p[x];
    return x;   }
```

```
union(x,y)
{   a:=find(x);
    b:=find(y);
    if R[a]>R[b] then p[b]:=a;
                 else { p[a]:=b;
                        if R[a]=R[b] then
                              R[b]:=R[b]+1; }
```

2nd approach - Union by Rank

`make_set(x)`	$O(1)$
`find(x)`	$O(R[root(S_x)]-R[x]) \leq O(R[root(S_x)])$
`union(x,y)`	$O(\max\{R[root(S_x)], R[root(S_y)]\})$

To estimate the complexity, we need to bound the height of the created trees, i.e., the rank of the roots in the trees that emerge throughout the execution of union and find operations

2nd approach - Union by Rank

Given a call of find(x) or union(x, y):
S_x: tree that x belongs to (respectively S_y for y)
p: root of the tree S_x
q: root of the tree S_y
size(x): # of nodes of S_x (resp. size(y) for y)

Lemma: for any element x, size(x) $\geq 2^{R[p]}$

Proof: Consider the tree S_x that x belongs to

- By structural induction
  - The trees are growing only by union operations
- Basis (no UNION, single-node tree): R[p]=0, size(x)=1 : $1 \geq 2^0 = 1$
- Hypothesis: The lemma holds before a union(x,y) operation

2nd approach - Union by Rank

Inductive step: Let p' = new root.

Three cases in union(x,y):

1) `R[p]<R[q]  (R[p'] = R[q])`

$$size(p') = size(x) + size(y) \geq 2^{R[p]} + 2^{R[q]} \geq 2^{R[q]} = 2^{R[p']}$$

2) `R[p]>R[q] (R[p'] = R[p])`

same analysis as above

3) `R[p]=R[q] (R[p'] = R[q]+1)`

$$size(p') = size(x) + size(y) \geq 2^{R[p]} + 2^{R[q]} = 2 \cdot 2^{R[q]} = 2^{R[q]+1} = 2^{R[p']}$$

2nd approach - Union by Rank

- Thus, if n is the number of elements, then for any x:

$$n \geq size(x) \geq 2^{R[p]}$$

- ➔ $R[p] \leq \log n$

- Each union and find operation takes time O(logn)

- Implication for Kruskal's algorithm: O(mlogn)

- Can we do better?

- YES, much better !!! (but in amortized time)

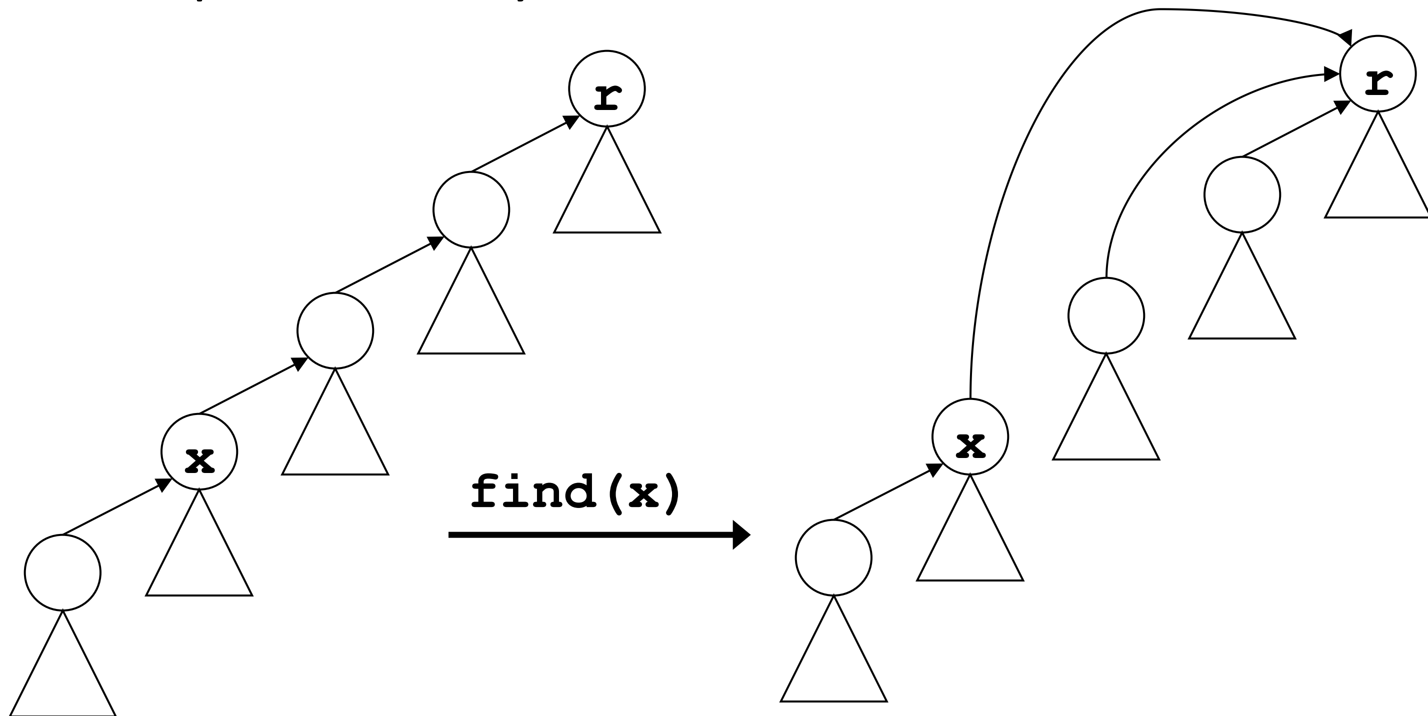
Amortized complexity

T(n,m) = complexity of a sequence of m make_set, union and find operations
on a collection of sets containing n elements in total

- We will have n make_set calls to begin with
- m-n union or find calls
- Amortized complexity per operation: T(n,m)/m

Path compression

- We will try to maintain trees with short height
- Trees grow because of the union operation
- **Idea:** modify find(x) so that the nodes we traverse from x to the root point directly to the root



Path compression

```
find(x)
{  if x!=p[x] then p[x]:=find(p[x]);
   return p[x];  }
```

- We trace the path x, a_1, a_2, ..., a_k, r from x to the root r
- At the same time, we also make all the parent pointers of x, a_1, a_2,..., a_k point to r directly

Theorem:

$T(n,m) = O(m\,\alpha(n))$ if union by rank and path compression are used (<u>now the rank is not the height of the tree</u>)

Ackerman, inverse Ackerman and $\log^* n$

- `A(i,j), i,j ≥ 1` Ackerman function
  - `A(1,j) = 2`^{j}`,` `j ≥ 1`
  - `A(i,1) = A(i-1,2),` `i ≥ 2`
  - `A(i,j) = A(i-1,A(i,j-1)),` `i,j ≥ 2`

- An extremely fast growing function
  - `A(2,1) = A(1, 2) = 4, A(2,2) = A(1,A(2,1)) = A(1,4) = 16`
  - `A(4,1)` $= 2^{65536} \approx 10^{80}$ **(more than the estimated number of atoms in the observable universe)**

- $\alpha(n) = \min\{k \geq 1: A(k,1) > n\}$ "inverse" Ackerman function

  e.g. if $n=2^{65536}-1$, then $A(4,1)=2^{65536}>n \Rightarrow k=4$

  for all practical purposes, we can say $\boldsymbol{\alpha(n) \leq 4 = O(1)}$

Ackerman, inverse Ackerman and $\log^* n$

- **log*n = min{i≥0 : log$^{(i)}$n ≤ 1}** Iterated logarithm

  e.g. for all n in {17,18,…,65536=2^{16}): log*n=4

  for all n in {65537,65538,…,2^{65536}=2^{64K}): log*n=5

- for all practical purposes, we can say **log*n ≤ 5**

- It holds that either α(n)=log*n-1 or α(n)=log*n, that is, α(n) ≤ log*n