Chapter 5
Data Structures

Algorithm Theory
WS 2016/17

Fabian Kuhn

Priority Queue / Heap

- Stores (*key,data*) pairs (like dictionary)
- But, different set of operations:

- **Initialize-Heap**: creates new empty heap
- **Is-Empty**: returns true if heap is empty
- **Insert**(*key,data*): inserts (*key,data*)-pair, returns pointer to entry
- **Get-Min**: returns (*key,data*)-pair with minimum *key*
- **Delete-Min**: deletes minimum (*key,data*)-pair
- **Decrease-Key**(*entry,newkey*): decreases *key* of *entry* to *newkey*
- **Merge**: merges two heaps into one

Fibonacci Heap

rootlist nodes are unmarked

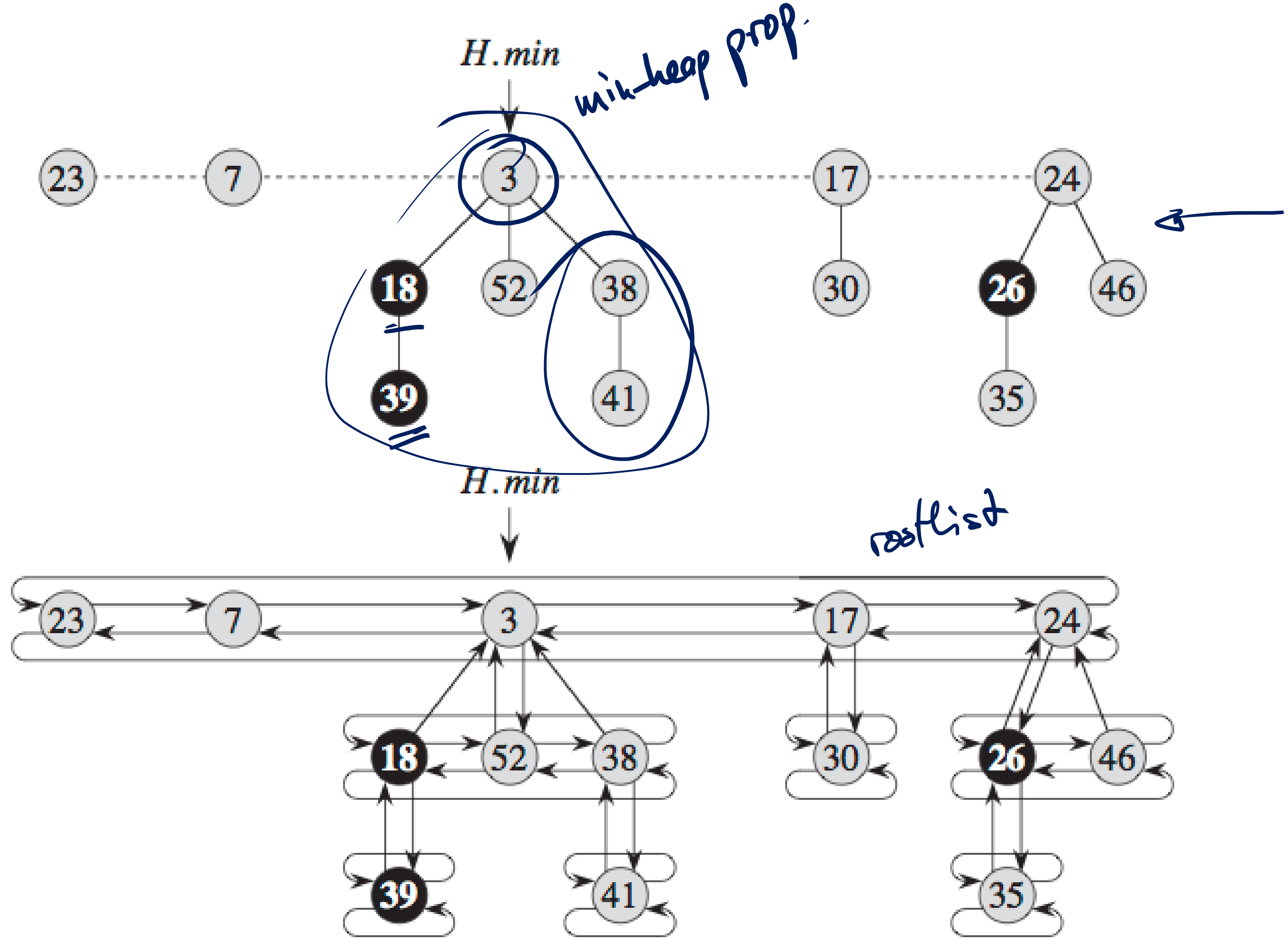


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

- $H.rootlist := H.min := null$

Merge heaps H and H':

- concatenate root lists
- update $H.min$

Insert element e into H:

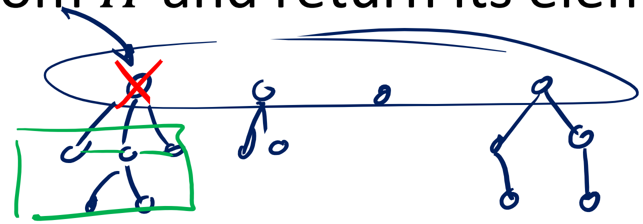
- create new one-node tree containing e → H'
  - mark of root node is set to **false**
- merge heaps H and H'

Get minimum element of H:

- return $H.min$

Operation Delete-Min

Delete the node with minimum key from H and return its element:



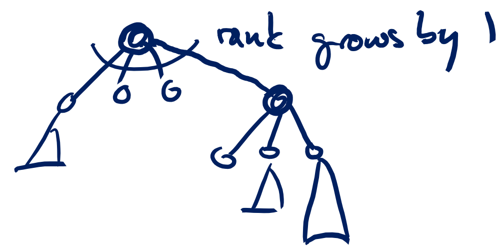
1. $m \coloneqq H.min;$
2. **if** $H.size > 0$ **then**
3. remove $H.min$ from $H.rootlist;$
4. add $H.min.child$ (list) to $H.rootlist$

remove marks of roots

5. $\boldsymbol{H.Consolidate}();$

// Repeatedly merge nodes with equal degree in the root list
// until degrees of nodes in the root list are distinct.
// Determine the element with minimum key

6. **return** m



Rank and Maximum Degree

Ranks of nodes, trees, heap:

Node v:

- $rank(v)$: degree of v (number of children of v)

Tree T:

- $rank(T)$: rank (degree) of root node of T

Heap H:

- $rank(H)$: maximum degree (#children) of any node in H

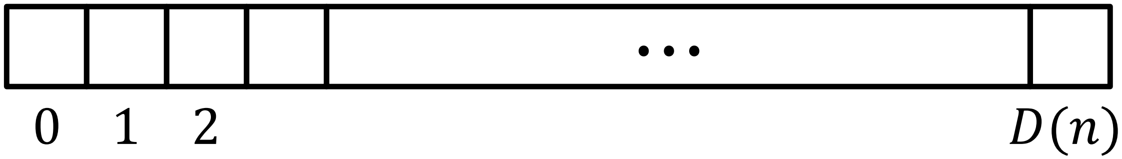
Assumption (n: number of nodes in H):

$$rank(H) \leq D(n)$$

– for a known function $D(n)$

Consolidation of Root List

Array A pointing to find roots with the same rank:



Consolidate:

1. **for** $i := 0$ **to** $D(n)$ **do** $A[i] :=$ null;
2. **while** $H.rootlist \neq$ null **do**
3. $\quad T :=$ "delete and return first element of $H.rootlist$"
4. $\quad$ **while** $A[rank(T)] \neq$ null **do**
5. $\qquad T' := A[rank(T)];$
6. $\qquad A[rank(T)] := null;$
7. $\qquad T := link(T, T')$
8. $\quad A[rank(T)] := T$
9. Create new $H.rootlist$ and $H.min$

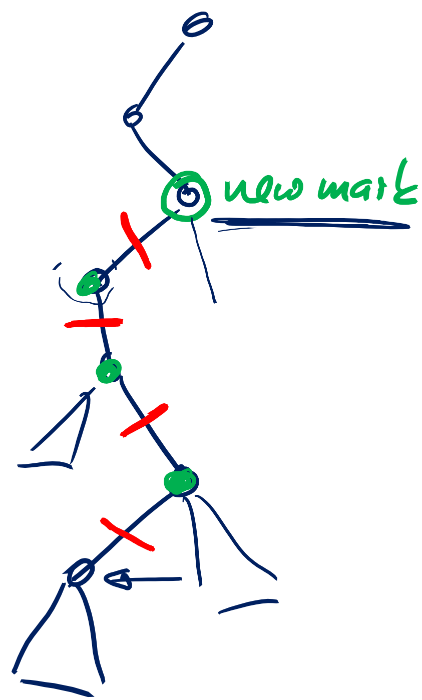
Time:
$O(|H.rootlist| + D(n))$

before consolidate

Operation Decrease-Key

Decrease-Key$(\boldsymbol{v}, \boldsymbol{x})$**:** (decrease key of node v to new value x)

1. **if** $x \geq v.key$ **then return**;
2. $v.key := x$; update $H.min$;
3. **if** $v \in H.rootlist \ \lor \ x \geq v.parent.key$ **then return**
4. **repeat**
5. $\quad parent := v.parent$;
6. $\quad \boldsymbol{H.cut(v)}$;
7. $\quad v := parent$;
8. **until** $\neg(\boldsymbol{v.mark}) \ \lor \ v \in H.rootlist$;
9. **if** $v \notin H.rootlist$ **then** $\boldsymbol{v.mark} := \mathbf{true}$;



Fibonacci Heap Marks

History of a node v:

v is being linked to a node → $\boldsymbol{v.mark = \textbf{false}}$

a child of v is cut → $\boldsymbol{v.mark \coloneqq \textbf{true}}$

a second child of v is cut → $\boldsymbol{H.cut(v);}$
$\boldsymbol{v.mark \coloneqq false}$

- Hence, the boolean value $v.mark$ indicates whether node v has lost a child since the last time v was made the child of another node.
- Nodes v in the root list always have $v.mark = \text{false}$

Cost of Delete-Min & Decrease-Key

Delete-Min:

1. Delete min. root r and add $r.child$ to $H.rootlist$

   time: $O(1)$ ($O(D(n))$ time to set marks to false)

2. Consolidate $H.rootlist$

   time: $O(\text{length of } H.rootlist + D(n))$

- Step 2 can potentially be linear in n (size of H)

Decrease-Key (at node v):

1. If new key < parent key, cut sub-tree of node v

   time: $O(1)$

2. Cascading cuts up the tree as long as nodes are marked

   time: $O(\text{number of consecutive marked nodes})$

- Step 2 can potentially be linear in n

Exercises: Both operations can take $\Theta(n)$ time in the worst case!

Fibonacci Heaps Complexity

- Worst-case cost of a single delete-min or decrease-key operation is $\Omega(n)$

- Can we prove a small worst-case amortized cost for delete-min and decrease-key operations?

Recall:

- Data structure that allows operations $O_1, \dots, O_k$

- We say that operation O_p has amortized cost a_p if for every execution the total time is

$$T \leq \sum_{p=1}^{k} n_p \cdot a_p \,,$$

where n_p is the number of operations of type O_p

Amortized Cost of Fibonacci Heaps

- **Initialize-heap**, **is-empty**, **get-min**, **insert**, and **merge** have **worst-case cost $O(1)$**
- **Delete-min** has **amortized cost $O(\log n)$**
- **Decrease-key** has **amortized cost $O(1)$**
- Starting with an empty heap, any sequence of n operations with at most n_d delete-min operations has total cost (time)

$$T = O(n + n_d \log n).$$

delete-min

- We will now need the marks…

$O(|E| \cdot \log|V|)$

- Cost for Dijkstra: $O(|E| + |V| \log |V|)$

Fibonacci Heaps: Marks

Cycle of a node:

1. Node v is removed from root list and linked to a node

   $\boldsymbol{v.mark = \text{false}}$

2. Child node u *of* v is cut and added to root list

   $\boldsymbol{v.mark := \text{true}}$

3. Second child of v is cut

   node $\boldsymbol{v}$ is cut as well and moved to root list
   $\boldsymbol{v.mark := \text{false}}$

The boolean value $v.mark$ indicates whether node v has lost a child since the last time v was made the child of another node.

Potential Function

$\phi = R + M$

(D.S.)

System state characterized by two parameters:

- $\boldsymbol{R}$: number of trees (length of $H.rootlist$)
- $\boldsymbol{M}$: number of marked nodes (not in the root list)

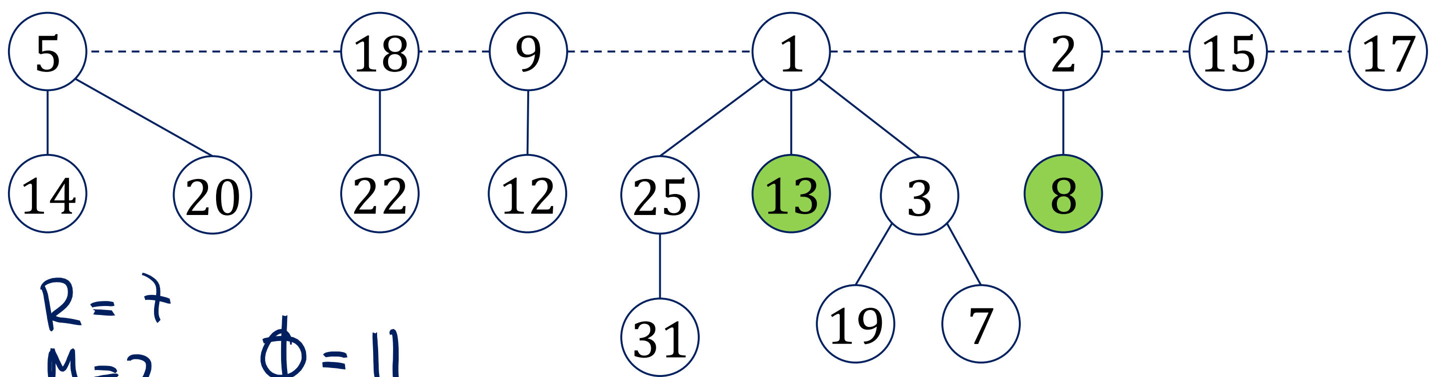
Potential function:

$a_i := t_i + \phi_i - \phi_{i-1}$

$$\boldsymbol{\Phi := R + 2M}$$

$\phi \geq 0$

Example:



$R = 7$
$M = 2$

$\phi = 11$

- $R = 7, M = 2 \;\rightarrow\; \Phi = 11$

Actual Time of Operations

- Operations: ***initialize-heap***, ***is-empty***, ***insert***, ***get-min***, ***merge***

  actual time: $O(1)$

  – Normalize unit time such that

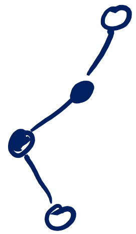
  $$t_{init}, t_{is-empty}, t_{insert}, t_{get-min}, t_{merge} \leq 1$$

- Operation ***delete-min***:

  – Actual time: $O(\text{length of } H.rootlist + D(n))$

  – Normalize unit time such that

  $$t_{del-min} \leq D(n) + \text{ length of } H.rootlist$$



- Operation **descrease-key**:

  – Actual time: $O(\text{length of path to next unmarked ancestor})$

  – Normalize unit time such that

  $$t_{decr-key} \leq \text{length of path to next unmarked ancestor}$$

Amortized Times

Assume operation i is of type:

- **initialize-heap:**
  - actual time: $t_i \leq 1$, potential: $\Phi_{i-1} = \Phi_i = 0$
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

- **is-empty, get-min:**
  - actual time: $t_i \leq 1$, potential: $\Phi_i = \Phi_{i-1}$ (heap doesn't change)
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

- **merge:**
  - Actual time: $t_i \leq 1$
  - combined potential of both heaps: $\Phi_i = \Phi_{i-1}$
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

Amortized Time of Insert

Assume that operation i is an *insert* operation:

- **Actual time:** $t_i \leq 1$
- **Potential function:**
  - M remains unchanged (no nodes are marked or unmarked, no marked nodes are moved to the root list)
  - R grows by 1 (one element is added to the root list)

$$M_i = M_{i-1}, \qquad R_i = R_{i-1} + 1$$
$$\Phi_i = \Phi_{i-1} + 1$$

- **Amortized time:**

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2}$$

Amortized Time of Delete-Min

Assume that operation i is a *delete-min* operation:

Actual time: $t_i \leq D(n) + \overbrace{|H.rootlist|}^{R_{i-1}}$

Potential function $\boldsymbol{\Phi = R + 2M}$:

- R: changes from $|H.rootlist|$ to at most $D(n)$ $+1$
- M: (# of marked nodes)
  - Number of marks does not ~~change~~ increase

$R_{i-1} = |H.rootlist|$

$R_i \leq D(n)+1$

$$M_i \leq M_{i-1}, \qquad R_i \leq R_{i-1} + D(n) + 1 - |H.rootlist|$$

$$\Phi_i \leq \Phi_{i-1} + D(n) + 1 - |H.rootlist|$$

Amortized Time:

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq 2D(n) + 1} = O(D(n))$$

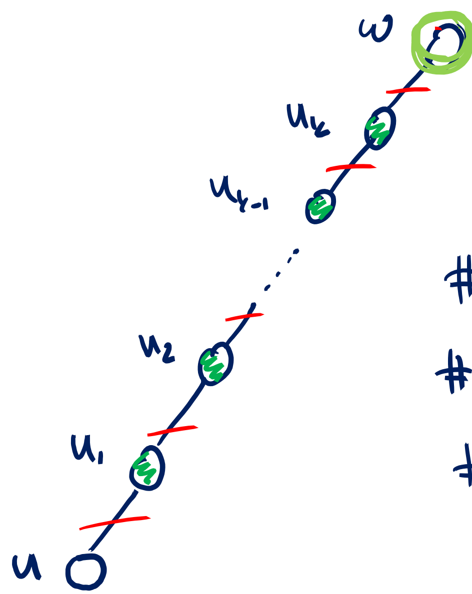
Amortized Time of Decrease-Key

Assume that operation i is a *decrease-key* operation at node u:

Actual time: $t_i \leq$ length of path to next unmarked ancestor v

Potential function $\Phi = R + 2M$:

- Assume, node u and nodes $u_1, \dots, u_k$ are moved to root list
  - $u_1, \dots, u_k$ are marked and moved to root list, $v.\text{mark}$ is set to true



new mark

$t_i = \underline{k+1}$

$\#$marks removed $\geq k$

$\#$marks added ≤ 1

$\#$cuts $= k+1$

Amortized Time of Decrease-Key

Assume that operation i is a *decrease-key* operation at node u:

Actual time: $t_i \leq \underbrace{\text{length of path to next unmarked ancestor } v}_{k+1}$

Potential function $\Phi = R + 2M$:

- Assume, node u and nodes $u_1, \dots, u_k$ are moved to root list
  - $u_1, \dots, u_k$ are marked and moved to root list, $v.$mark is set to true
- $\geq k$ marked nodes go to root list, ≤ 1 node gets newly marked
- R grows by $\leq k + 1$, M grows by 1 and is decreased by $\geq k$

$$R_i \leq R_{i-1} + k + 1, \qquad M_i \leq M_{i-1} + 1 - k$$
$$\Phi_i \leq \Phi_{i-1} + (k+1) - 2(k-1) = \Phi_{i-1} + 3 - k$$

Amortized time:

$$\boldsymbol{a_i = t_i + \Phi_i - \Phi_{i-1} \leq k + 1 + 3 - k = 4} = O(1)$$

Complexities Fibonacci Heap

- **Initialize-Heap**: $\boldsymbol{O(1)}$
- **Is-Empty**: $\boldsymbol{O(1)}$
- **Insert**: $\boldsymbol{O(1)}$
- **Get-Min**: $\boldsymbol{O(1)}$
- **Delete-Min**: $\boldsymbol{O(D(n))}$
- **Decrease-Key**: $\boldsymbol{O(1)}$

amortized

- **Merge** (heaps of size m and n, $m \leq n$): $\boldsymbol{O(1)}$

- **How large can $D(n)$ get?**

Rank of Children

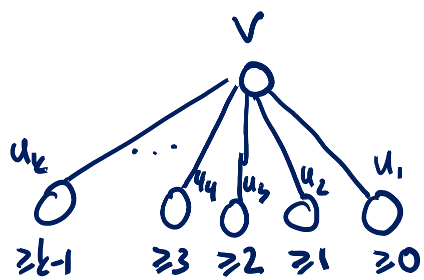
Lemma:

Consider a node v of rank k and let $u_1, \dots, u_k$ be the children of v in the order in which they were linked to v. Then,

$$\boldsymbol{rank(u_i) \geq i - 2}.$$

Proof:

rank when adding child



when u_i was added

$rank(u_i) \geq i-1$

Size of Trees

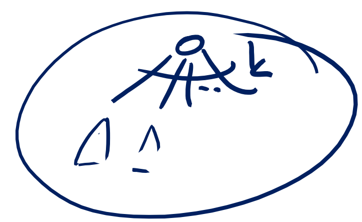
0, 1, 1, 2, 3, 5, 8, 13, 21

Fibonacci Numbers:

$$F_0 = 0, \qquad F_1 = 1, \qquad \forall k \geq 2: F_k = F_{k-1} + F_{k-2}$$

Lemma:

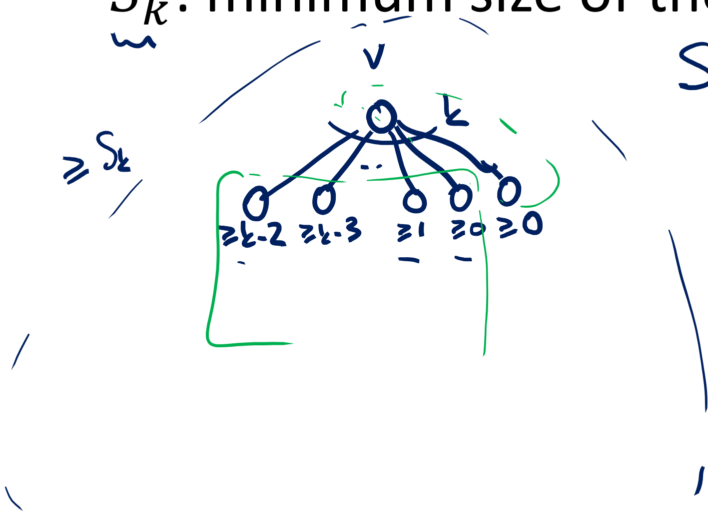
In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2}.



size $\geq F_{k+2}$

Proof:

- S_k: minimum size of the sub-tree of a node of rank k



$S_0 = 1$, $S_1 = 2$

$\underline{k \geq 2:}$ $S_k \geq 2 + \sum_{i=0}^{k-2} S_i$

(using prev. lemma)

Size of Trees

$$S_0 = 1, \qquad S_1 = 2, \qquad \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i$$

- Claim about Fibonacci numbers:

$$\forall k \geq 0: F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

Proof: (ind. on k)

$k=0$: $F_2 = 1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1$ ✓

$k>0$: $F_{k+2} = F_k + \underbrace{F_{k+1}}_{\text{I.H.}:\ = 1 + \sum_{i=0}^{k-1} F_i} = F_k + 1 + \sum_{i=0}^{k-1} F_i = 1 + \sum_{i=0}^{k} F_i$ ✓

$\square$

Size of Trees

$$S_0 = 1, S_1 = 2, \forall k \geq 2: S_k \geq 2 + \sum_{i=0}^{k-2} S_i, \qquad F_{k+2} = 1 + \sum_{i=0}^{k} F_i$$

- Claim of lemma: $S_k \geq F_{k+2}$

Ind. on k:

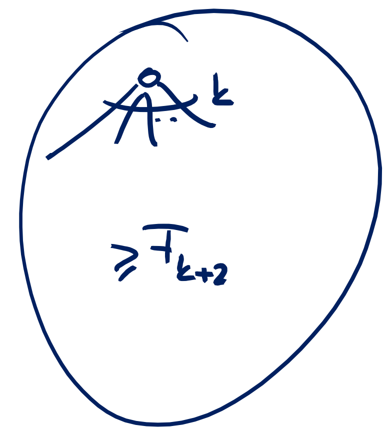
base: $S_0 \geq F_2 = 1$ ✓ $\quad S_1 \geq F_3 = 2$ ✓

step: $k \geq 2$:

$$S_k \geq 2 + \sum_{i=0}^{k-2} S_i \overset{(IH)}{\geq} 2 + \sum_{i=0}^{k-2} F_{i+2}$$

$$= 2 + \sum_{j=2}^{k} F_j$$

$$= 1 + \sum_{j=0}^{k} F_j = F_{k+2} \qquad \square$$



Size of Trees

Lemma:
In a Fibonacci heap, the size of the sub-tree of a node v with rank k is at least F_{k+2}.

Theorem:
The maximum rank of a node in a Fibonacci heap of size n is at most

$$\boldsymbol{D(n) = O(\log n)}\,.$$

Proof:

- The Fibonacci numbers grow exponentially:

$$F_k = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2}\right)^k - \overbrace{\left(\frac{1-\sqrt{5}}{2}\right)^k}^{\to 0} \right)$$

- For $D(n) \geq k$, we need $n \geq F_{k+2}$ nodes.

Summary: Binomial and Fibonacci Heaps

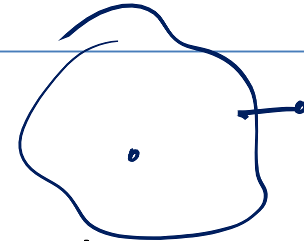
	Binary Heap	Fibonacci Heap
initialize	$O(1)$	$O(1)$
insert	$O(\log n)$	$O(1)$
get-min	$O(1)$	$O(1)$
delete-min	$O(\log n)$	$O(\log n)$ *
decrease-key	$O(\log n)$	$O(1)$ *
merge	$O(m \cdot \log n)$	$O(1)$
is-empty	$O(1)$	$O(1)$

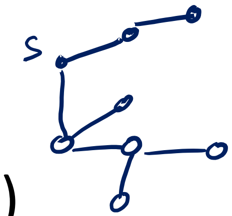
Dijkstra in time: $O(m + n \log n)$

* **amortized time**

Minimum Spanning Trees

Prim Algorithm:





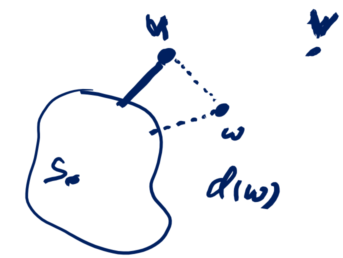
1. Start with any node v (v is the initial component)
2. In each step:
   Grow the current component by adding the minimum weight edge e connecting the current component with any other node

Kruskal Algorithm:

1. Start with an empty edge set
2. In each step:
   Add minimum weight edge e such that e does not close a cycle

Implementation of Prim Algorithm

Start at node s, very similar to Dijkstra's algorithm:



1. Initialize $d(s) = 0$ and $d(v) = \infty$ for all $v \neq s$
2. All nodes $s \geq v$ are unmarked

   add all nodes to an empty priority queue (d(v): key)

3. Get unmarked node u which minimizes $d(u)$:

   get-min

   d(v) + w(e)

4. For all $e = \{u, v\} \in E$, $d(v) = \min\{d(v), \boxed{w(e)}\}$

   potentially update d(v) of neighbors : decrease-key

5. mark node u

   delete-min

6. Until all nodes are marked

Implementation of Prim Algorithm

Implementation with Fibonacci heap:

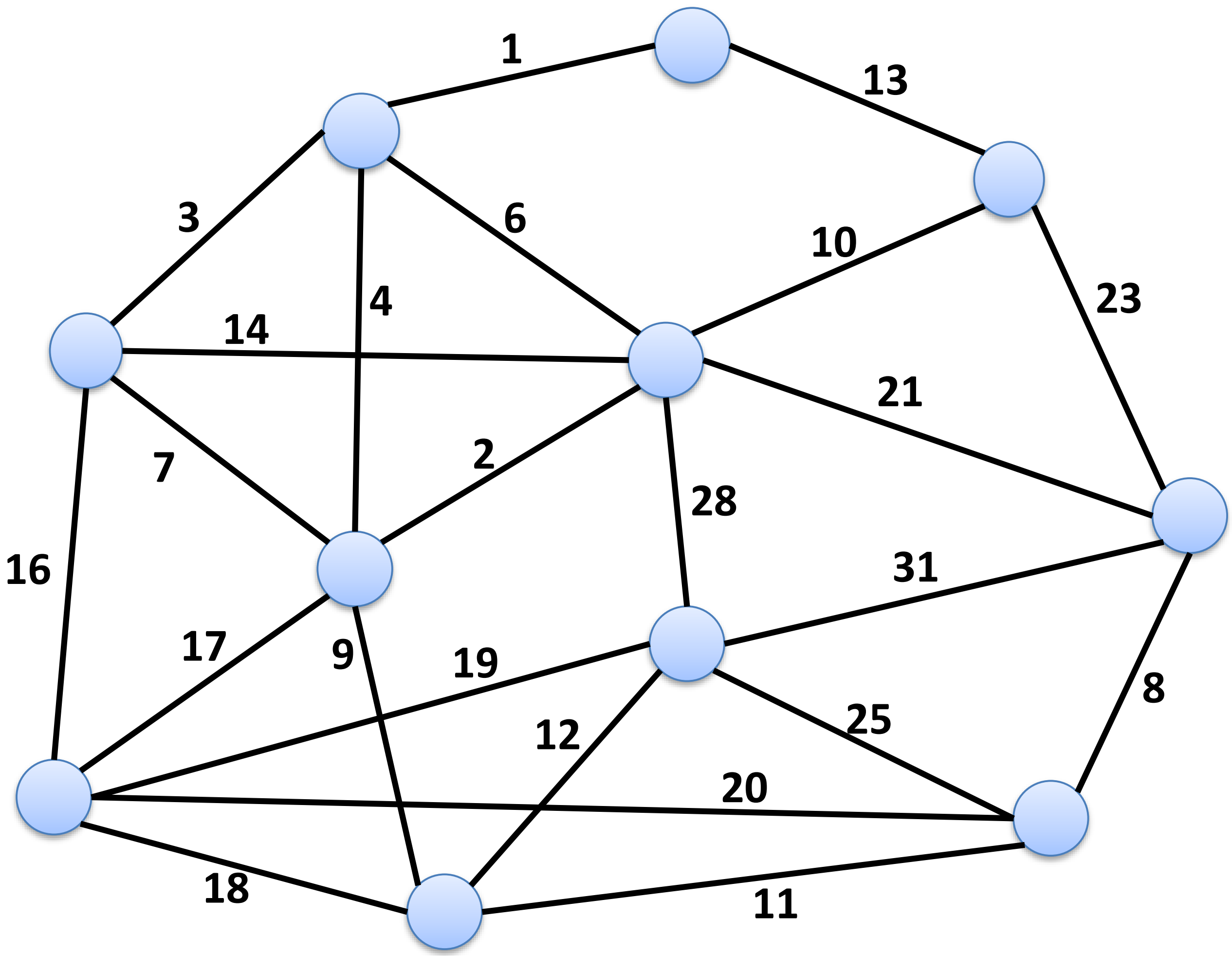
- Analysis identical to the analysis of Dijkstra's algorithm:

  $O(n)$ insert and delete-min operations

  $O(m)$ decrease-key operations

- Running time: $\boldsymbol{O(m + n \log n)}$

Kruskal Algorithm



1. Start with an empty edge set
2. In each step: Add minimum weight edge *e* such that *e* does *not* close a cycle

Implementation of Kruskal Algorithm

$\log m \leq 2 \log n \qquad n-1 \leq m \leq \binom{n}{2}$

1. Go through edges in order of increasing weights

   Sort edges by weight $\qquad$ $\underline{O(m \log n)}$

2. For each edge e: $= \{u,v\}$

   if ***e*** **does not close a cycle then**

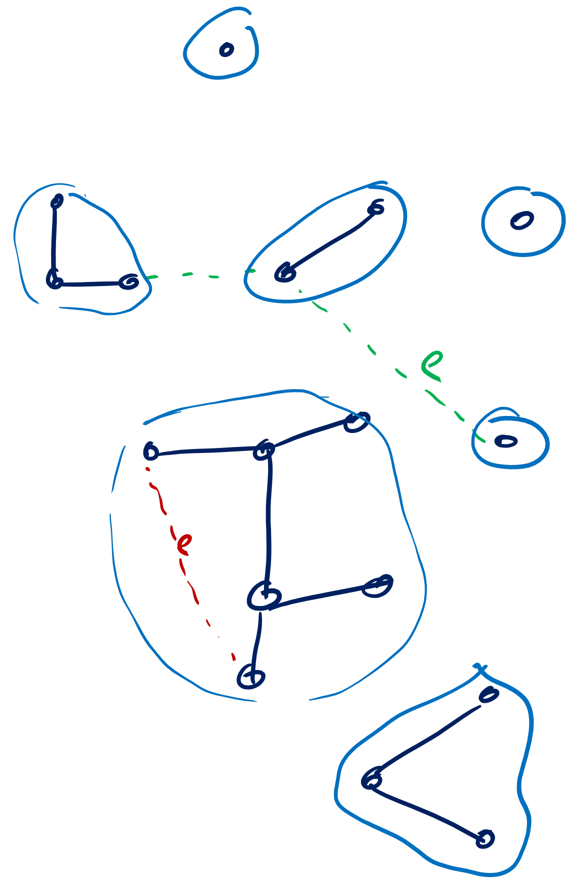
   need to check whether $\{u,v\}$ closes a cycle

   $\Longleftrightarrow$ check whether u & v are in the same component

   add ***e*** **to the current solution**

   add $\{u,v\}$

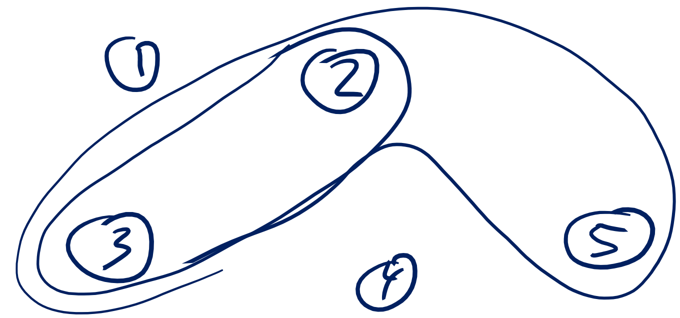
   need to merge comp. of u & v



Union-Find Data Structure

Also known as **Disjoint-Set Data Structure**...

Manages partition of a set of elements
- set of disjoint sets



Operations:

- **make_set(x):** create a new set that only contains element x
- **find(x):** return the set containing x
- **union(x, y):** merge the two sets containing x and y

Implementation of Kruskal Algorithm

1. Initialization:
   For each node v: $\mathrm{make\_set}(v)$

2. Go through edges in order of increasing weights:
   Sort edges by edge weight

3. For each edge $e = \{u, v\}$:

   if $\mathbf{find}(\boldsymbol{u}) \neq \mathbf{find}(\boldsymbol{v})$ **then**

   add e to the current solution

   $\mathbf{union}(\boldsymbol{u}, \boldsymbol{v})$