Chapter 4

Data Structures

Fibonacci Heaps, Amortized Analysis

Algorithm Theory
WS 2012/13

Fabian Kuhn

Fibonacci Heaps

Lacy-merge variant of binomial heaps:

- Do not merge trees as long as possible…

Structure:

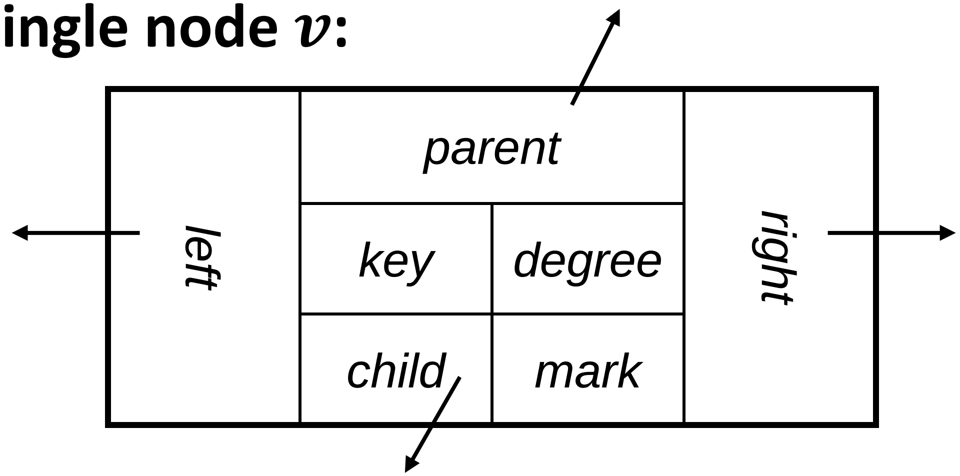
A Fibonacci heap H consists of a collection of trees satisfying the min-heap property.

Variables:

- $H.min$: root of the tree containing the (a) minimum key
- $H.rootlist$: circular, doubly linked, unordered list containing the roots of all trees
- $H.size$: number of nodes currently in H

Trees in Fibonacci Heaps

Structure of a single node v:



- $v.child$: points to circular, doubly linked and unordered list of the children of v
- $v.left, v.right$: pointers to siblings (in doubly linked list)
- $v.mark$: will be used later…

Advantages of circular, doubly linked lists:

- Deleting an element takes constant time
- Concatenating two lists takes constant time

Example

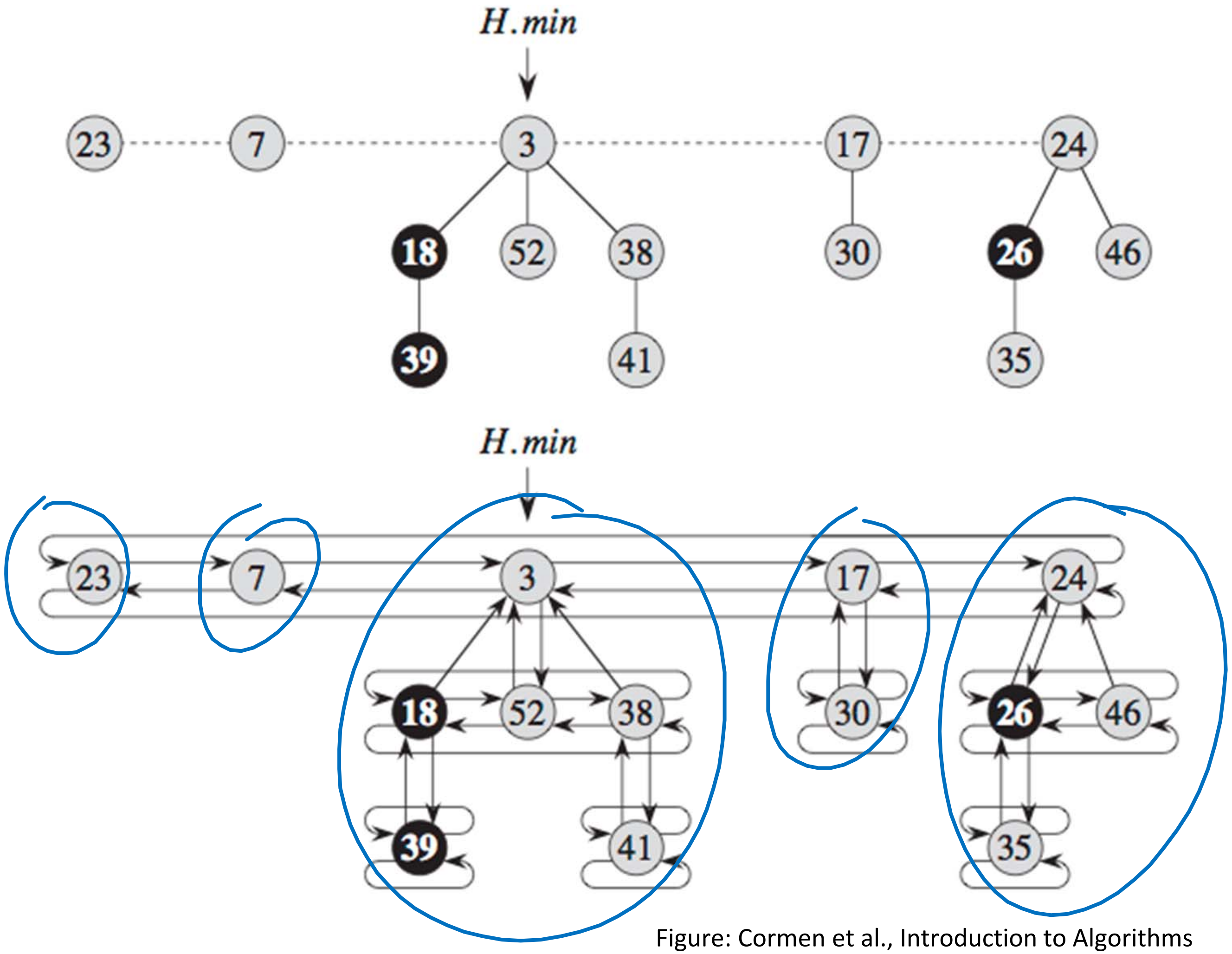


Figure: Cormen et al., Introduction to Algorithms

Simple (Lazy) Operations

Initialize-Heap H:

- $H.rootlist \coloneqq H.min \coloneqq null$

Merge heaps H and H':

- concatenate root lists
- update $H.min$

Insert element e into H:

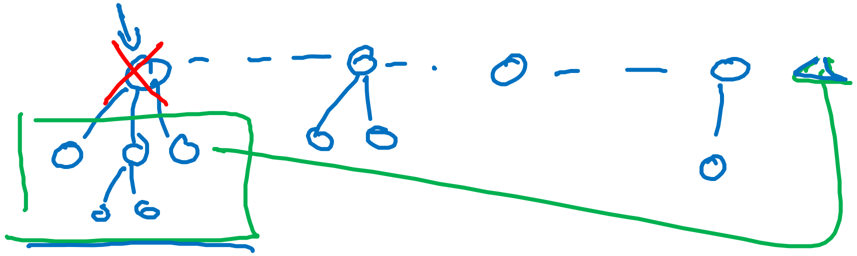
- create new one-node tree containing e → H'
- merge heaps H and H'

Get minimum element of H:

- return $H.min$

Operation Delete-Min

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



```
1.  $m := H.min$;
2.  if $H.size > 0$ then
3.        remove $H.min$ from $H.rootlist$;
4.        add $H.min.child$ (list) to $H.rootlist$
5.  $H.Consolidate()$;
    // 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

Rank and Maximum Degree

Ranks of nodes, trees, heap:

Node v:

- $rank(v)$: degree of v

Tree T:

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

Heap H:

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

Assumption (n: number of nodes in H):

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

- for a known function $D(n)$

Merging Two Trees

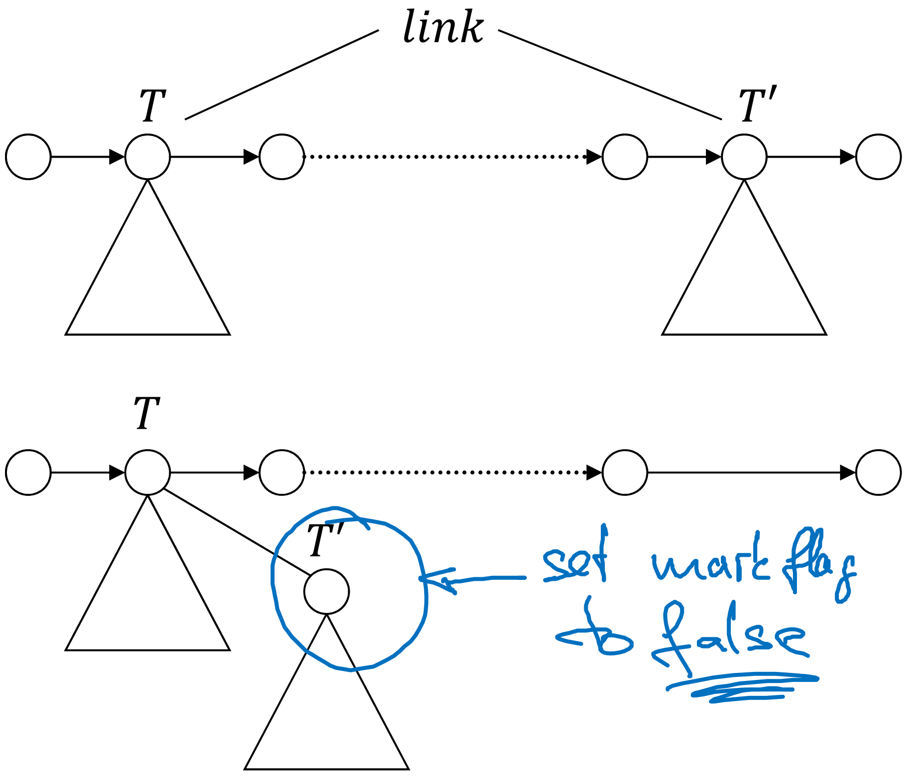
Given: Heap-ordered trees T, T' with $rank(T) = rank(T')$

- Assume: min-key of $T <$ min-key of T'

Operation $\boldsymbol{link(T, T')}$:

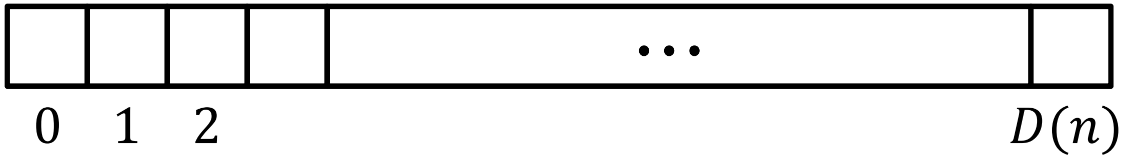
- Removes tree T' from root list and adds T' to child list of T
- $rank(T) := rank(T) + 1$
- $T'.mark :=$ **false**

each node has mark flag



Consolidation of Root List

Array A pointing to find roots with the same rank:



Consolidate:

```
for $i := 0$ to $D(n)$ do $A[i] :=$ null;
while $H.rootlist \neq$ null do
    $T :=$ "delete and return first element of $H.rootlist$"
    while $A[rank(T)] \neq$ null do
        $T' := A[rank(T)]$;
        $A[rank(T)] := null$;
        $T := link(T, T')$
    $A[rank(T)] := T$
Create new $H.rootlist$ and $H.min$
```

Time:
$\Theta(\underbrace{|H.rootlist|}_{\text{length}} + \underline{D(n)})$

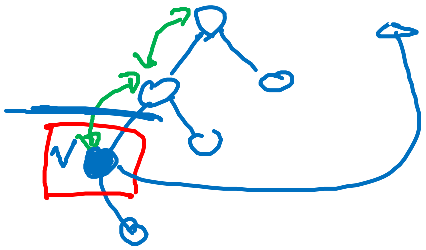
cost of del-min

$O(\text{length of root list} + D(n))$

Operation Decrease-Key

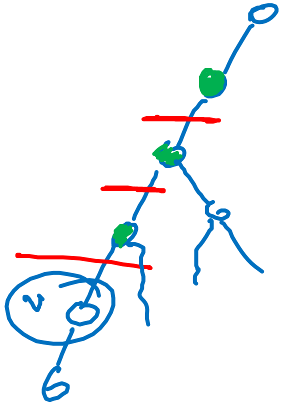
Decrease-Key$(\boldsymbol{v}, \boldsymbol{x})$: (decrease key of node v to new value x)

```
if $x \geq v.key$ then return;
$v.key := x$; update $H.min$;
if $v \in H.rootlist \lor x \geq v.parent.key$ then return
repeat
    $parent := v.parent$;
    $H.cut(v)$;
    $v := parent$;
until $\neg(v.mark) \lor v \in H.rootlist$;
if $v \notin H.rootlist$ then $v.mark := true$;
```



if parent is marked then also cut parent

repeat

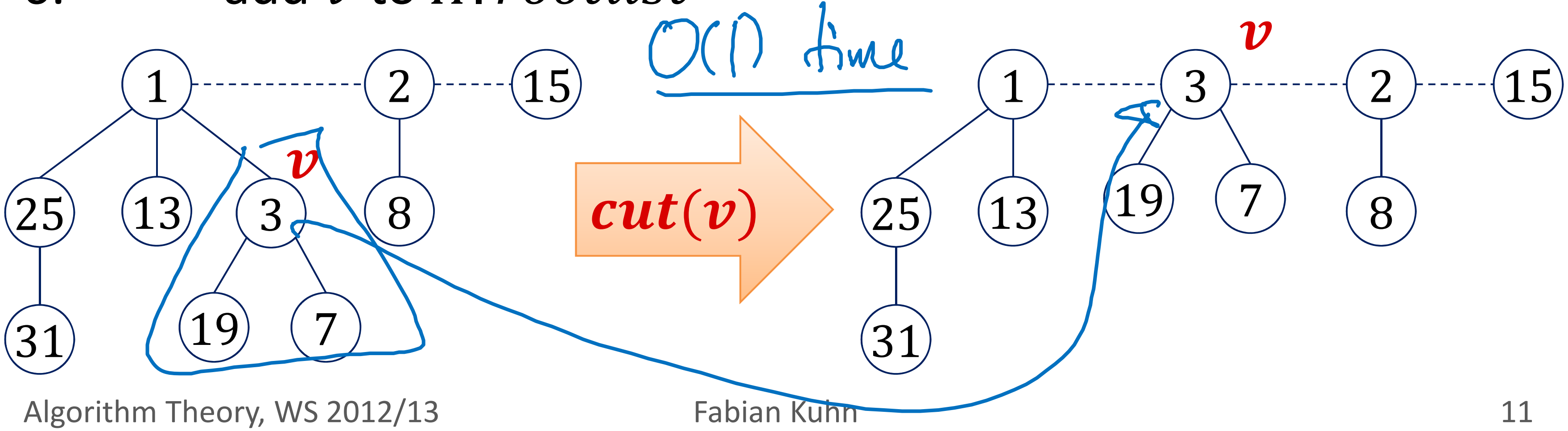


Operation Cut(v)

Operation $H.cut(v)$:

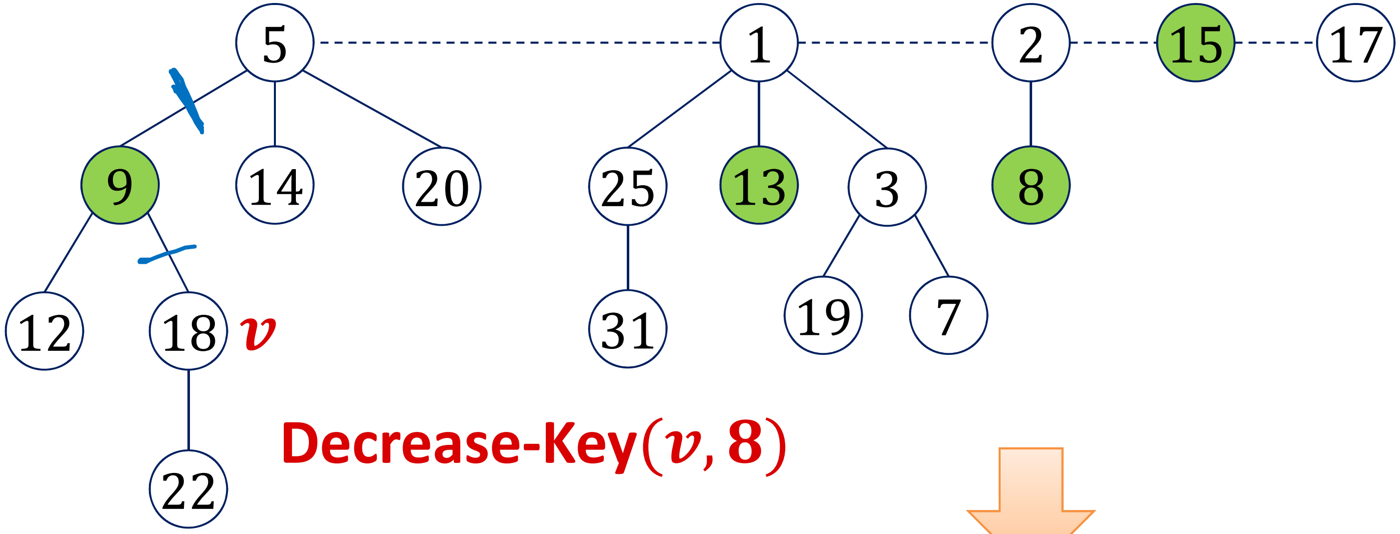
- Cuts v's sub-tree from its parent and adds v to rootlist

```
1. if $v \notin H.rootlist$ then
2.     // cut the link between $v$ and its parent
3.     $rank(v.parent) := rank(v.parent) - 1;$
4.     remove $v$ from $v.parent.child$ (list)
5.     $v.parent := \text{null};$
6.     add $v$ to $H.rootlist$
```

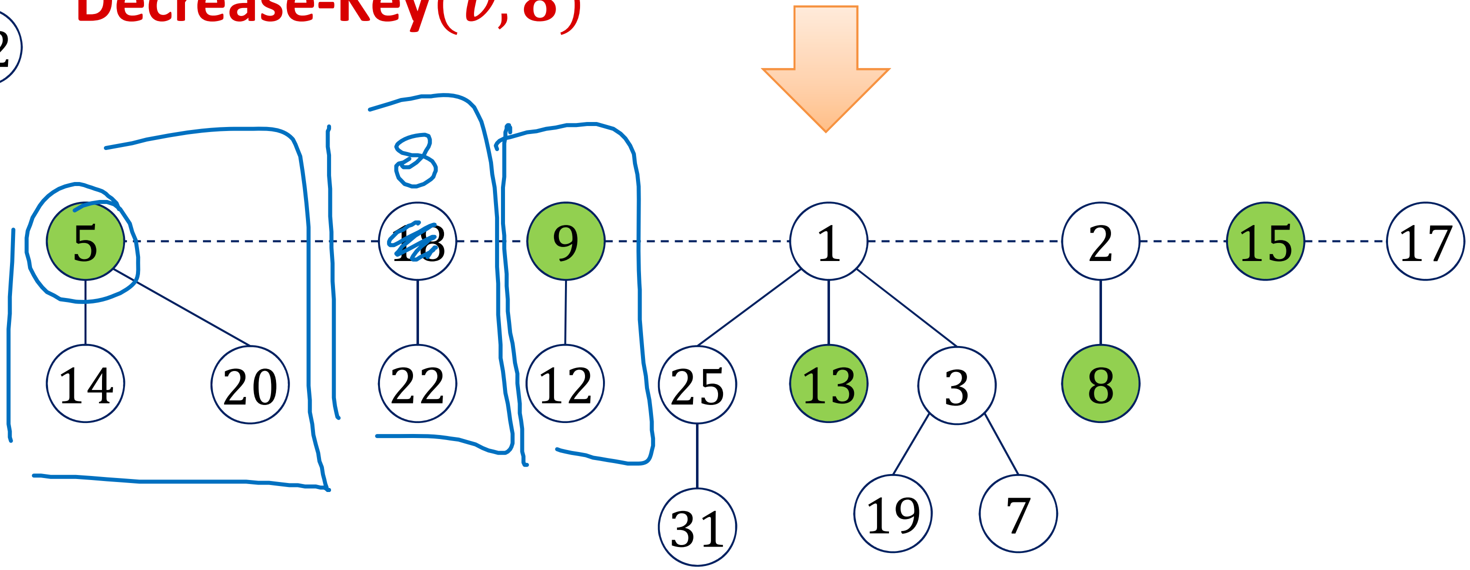


Decrease-Key Example

- Green nodes are marked

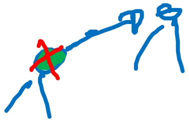


Decrease-Key$(v, 8)$

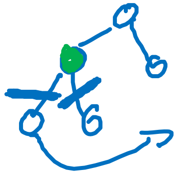


Fibonacci Heap Marks

History of a node v:



v is being linked to a node ⇨ $\underline{v.mark := \textbf{false}}$



a child of v is cut ⇨ $v.mark := \textbf{true}$

a second child of v is cut ⇨ $\underline{H.cut(v)} \rightarrow$ root list

- 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.

Cost of Delete-Min & Decrease-Key

Delete-Min:

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

   time: $O(1)$

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!

Cost of Delete-Min & Decrease-Key

- Cost of delete-min and decrease-key can be $\Theta(n)$…
  - Seems a large price to pay to get insert and merge in $O(1)$ time

- Maybe, the operations are efficient most of the time?
  - It seems to require a lot of operations to get a long rootlist and thus, an expensive consolidate operation
  - In each decrease-key operation, at most one node gets marked: We need a lot of decrease-key operations to get an expensive decrease-key operation

- Can we show that the average cost per operation is small?

- We can → requires **amortized analysis**

Amortization

- Consider sequence $o_1, o_2, \dots, o_n$ of n operations (typically performed on some data structure D)

- $\boldsymbol{t_i}$: execution time of operation o_i
- $\boldsymbol{T \coloneqq t_1 + t_2 + \cdots + t_n}$: total execution time

- The execution time of a single operation might

  vary within a large range (e.g., $t_i \in [1, O(i)]$)

- The worst case overall execution time might still be small

  → average execution time per operation might be small in the worst case, even if single operations can be expensive

$\underbrace{\frac{T}{n}}_{O(1)}$: amortized exec. time of an operation

Analysis of Algorithms

- Best case
- Worst case worst exec. time of an operation
- Average case hard complexity of a typical execution
- Amortized worst case

What it the average cost of an operation in a worst case sequence of operations?

Example: Binary Counter

Incrementing a binary counter: determine the bit flip cost:

Operation	Counter Value	Cost
	00000	
1	00001	1
2	00010	2
3	00011	1
4	00100	3
5	00101	1
6	00110	2
7	00111	1
8	01000	4
9	01001	1
10	01010	2
11	01011	1
12	01100	3
13	01101	1

$0\underbrace{111111}$

$10\ldots0$

$2n$

Accounting Method

Observation:

- Each increment flips exactly one 0 into a 1

$$0010001111 \Longrightarrow 0010010000$$

Idea:

- Have a bank account (with initial amount 0)
- Paying x to the bank account costs x
- Take "money" from account to pay for expensive operations

Applied to binary counter:

- Flip from 0 to 1: pay 1 to bank account (cost: 2)
- Flip from 1 to 0: take 1 from bank account (cost: 0)
- Amount on bank account = number of ones

  → We always have enough "money" to pay!

Accounting Method

Op.	Counter	Cost	To Bank	From Bank	Net Cost	Credit
	0 0 0 0 0					
1	0 0 0 0 1	1	1	0	2	1
2	0 0 0 1 0	2	1	1	2	1
3	0 0 0 1 1	1	1	0	2	2
4	0 0 1 0 0	3	1	2	2	1
5	0 0 1 0 1	1	1	0	2	2
6	0 0 1 1 0	2	1	1	2	2
7	0 0 1 1 1	1	1	0	2	3
8	0 1 0 0 0	4	1	3	2	1
9	0 1 0 0 1	1	1	0	2	2
10	0 1 0 1 0	2	1	1	2	2

$$\sum \leq \sum$$

Potential Function Method

- Most generic and elegant way to do amortized analysis!
  - But, also more abstract than the others…

- State of data structure / system: $S \in \mathcal{S}$ (state space)

  Potential function $\boldsymbol{\Phi}: \mathcal{S} \to \mathbb{R}_{\geq 0}$

- **Operation $\boldsymbol{i}$:**
  - $\boldsymbol{t_i}$: actual cost of operation i
  - $\boldsymbol{S_i}$: state after execution of operation i (S_0: initial state)
  - $\boldsymbol{\Phi_i} \coloneqq \Phi(S_i)$: potential after exec. of operation i
  - $\boldsymbol{a_i}$: amortized cost of operation i:

$$\boldsymbol{a_i \coloneqq \underline{t_i} + \underline{\Phi_i} - \underline{\Phi_{i-1}}}$$

Potential Function Method

Operation i:

actual cost: t_i amortized cost: $a_i = t_i + \Phi_i - \Phi_{i-1}$

Overall cost:

$$T := \sum_{i=1}^{n} t_i = \left(\sum_{i}^{n} a_i\right) + \overbrace{\Phi_0 - \Phi_n}^{\text{if } \Phi_0 = 0 \text{ neg}}$$

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

$$\begin{aligned}
\sum t_i = {} & a_1 + \Phi_0 - \cancel{\Phi_1} \\
& + a_2 \qquad + \cancel{\Phi_1} - \cancel{\Phi_2} \\
& + a_3 \qquad\qquad + \cancel{\Phi_2} - \cancel{\Phi_3} \\
& \quad \vdots \\
& \qquad\qquad\qquad\qquad - \cancel{\Phi_{n-1}} \\
& + a_n \qquad\qquad\qquad + \cancel{\Phi_{n-1}} - \Phi_n
\end{aligned}$$

Binary Counter: Potential Method

- **Potential function:**

  Φ: number of ones in current counter

- Clearly, $\underline{\Phi_0 = 0}$ and $\underline{\Phi_i \geq 0}$ for all $i \geq 0$

- Actual cost t_i:
  - 1 flip from 0 to 1
  - $t_i - 1$ flips from 1 to 0

$\Phi_i = \Phi_{i-1} + 1 - (t_i - 1)$

- Potential difference: $\underline{\Phi_i - \Phi_{i-1}} = 1 - (t_i - 1) \underline{= 2 - t_i}$

- Amortized cost: $a_i = \underline{t_i} + \underline{\Phi_i - \Phi_{i-1}} \underline{\underline{= 2}}$

Back to Fibonacci Heaps

- 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?

Remark:

- 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

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

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

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 $\underline{O(1)}$**

- Starting with an empty heap, any sequence of n operations with at most $\underline{n_d}$ delete-min operations has total cost (time)

$$T = O(\overset{\swarrow}{n} + \underline{n_d \log n}).$$

- We will now need the marks…

- Cost for Dijkstra: $O(\underbrace{|E|}_{\text{amort. cost of decr. key } O(1)} + \underline{|V| \log |V|})$

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 v is cut as well and moved to root list

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

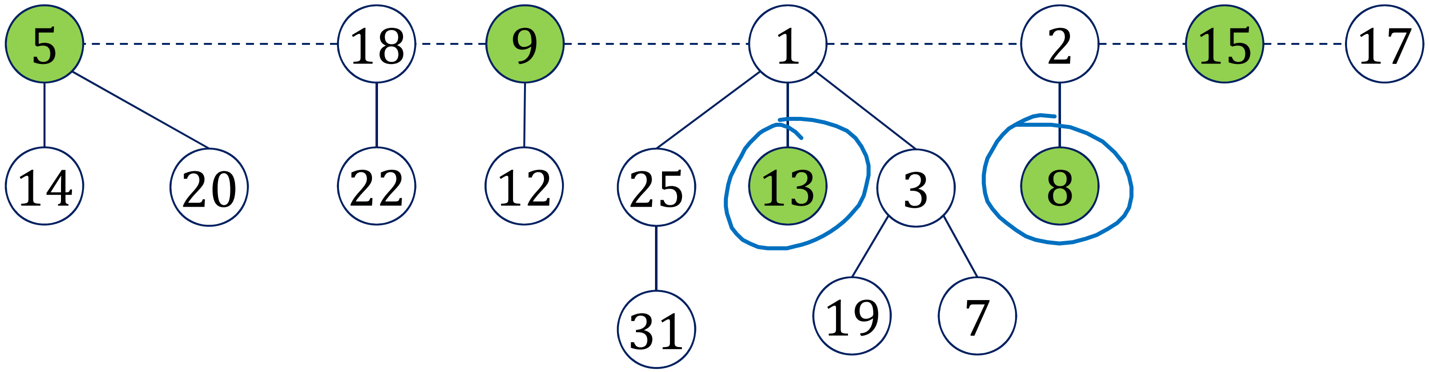
System state characterized by two parameters:

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

Potential function:

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

Example:



- $R = 7, M = 2 \rightarrow \underline{\Phi = 11}$

Actual Time of Operations

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

  actual time: $\underline{O(1)}$

  - Normalize unit time such that

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

- Operation ***delete-min***:
  - Actual time: $O\left(\underline{\text{length of } H.rootlist} + \underline{D(n)}\right)$
  - Normalize unit time such that

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

- Operation **descrease-key**:
  - Actual time: $O(\text{length of path to next unmarked ancestor})$
  - Normalize unit time such that

    #cuts

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

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

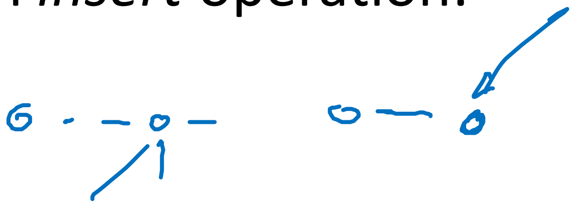
- **is-empty, get-min:**
  - actual time: $t_i \leq 1$, potential: $\Phi_i = \Phi_{i-1}$ (heap doesn't change)
  - amortized time: $a_i = t_i + \Phi_i - \Phi_{i-1} \leq 1$

- **merge:**
  - Actual time: $\underline{t_i \leq 1}$
  - 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:**

$\Phi_i = R_i + 2M_i$

  - 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)

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

- **Amortized time:**

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

Amortized Time of Delete-Min

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

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

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

- R: changes from $H.rootlist$ to at most $\underline{D(n)}$
- M: (# of marked nodes that are not in the root list)
  - no new marks
  - if node v is moved away from root list, $v.mark$ is set to false → value of M does not change!

$$\underline{M_i \le M_{i-1},} \qquad R_i \le \underline{R_{i-1}} + \underbrace{\overset{\swarrow}{D(n)} - \overbrace{|H.rootlist|}^{R_{i-1}}}$$

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

Amortized Time:

$D(n)+|H.rootlist|+D(n)-|H.rootlist|$

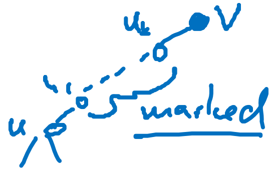
$$\boldsymbol{a_i = \underline{t_i} + \Phi_i - \Phi_{i-1} \le \underline{2D(n)}}$$

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



$t_i = k+1$

- 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, $v.\mathrm{mark}$ is set to true
- $\geq k$ marked nodes go to root list, ≤ 1 node gets newly marked

  $M_i = M_{i-1} - k$ $\quad$ $M_i = M_{i-1} + 1$
- R grows by $\leq k + 1$, M grows by 1 and is decreased by $\geq k$

$$\boxed{R_i \leq R_{i-1} + k + 1,} \quad \boxed{M_i \leq M_{i-1} + 1 - k}$$

$$\Phi_i \leq \Phi_{i-1} + (k+1) - 2(k-1) = \Phi_{i-1} + 3 - k$$

Amortized time:

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

Complexities Fibonacci Heap

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

amortized

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

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

$D(n) = O(\log n)$

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:

Size of Trees

Fibonacci Numbers:

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

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

Proof:

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

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:

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

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: $S_k \geq F_{k+2}$

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:

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

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

Summary: Binomial and Fibonacci Heaps

	Binomial 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	$O(\log n)$	$O(1)$
is-empty	$O(1)$	$O(1)$

* **amortized time**