



Chapter 4

Data Structures

Algorithm Theory
WS 2014/15

Fabian Kuhn

Amortized Analysis

- Consider sequence o_1, o_2, \dots, o_n of n operations (typically performed on some data structure D)
- t_i : execution time of operation o_i
- $T := t_1 + t_2 + \dots + t_n$: 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

Example: Binary Counter

Incrementing a binary counter: determine the bit flip cost:

Operation	Counter Value	Cost
	00000	
1	0000 1	1
2	000 10	2
3	000 11	1
4	00 100	3
5	0010 1	1
6	001 10	2
7	001 11	1
8	01000	4
9	0100 1	1
10	010 10	2
11	010 11	1
12	01 100	3
13	0110 1	1

Accounting Method



Op.	Counter	<u>Cost</u>	<u>To Bank</u>	From Bank	<u>Net Cost</u>	Credit
	00000					
1	0000 1	1	1	0	2	1
2	000 1 0	2	1	1	2	1
3	0001 1	1	1	0	2	2
4	00 1 00	3	1	2	2	
5	0010 1	1	1	0	2	
6	001 1 0	2	1	1	1	
7	0011 1	1	1	0	1	
8	0 1 000	4	1	3	1	
9	0100 1	1	1	0	1	
10	010 1 0	2	1	1		

C T F $N = C + T - F$ $N \leq C$

amount on bank at the end ≥ 0

Potential Function Method

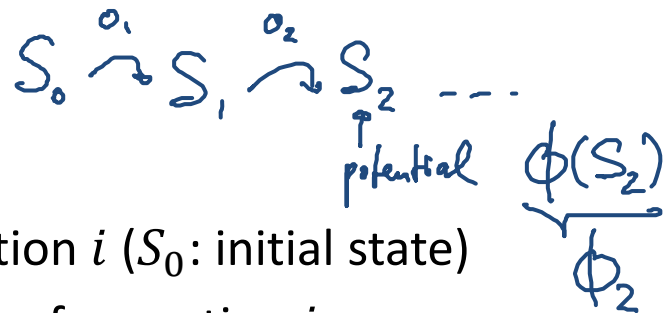
- Often a more **elegant** way to do amortized analysis!
 - But, also more abstract...

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

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

- **Operation i :**

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



$$a_i := t_i + \underbrace{\Phi_i - \Phi_{i-1}}_{\Delta\Phi}$$

often:
 $\Phi_0 = 0$

Potential Function Method

Operation i :

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

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

Overall cost:

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

$$\begin{aligned} \sum_{i=1}^n t_i &= \sum_{i=1}^n (a_i + \phi_{i-1} - \phi_i) \\ &= a_1 + \cancel{\phi_0} - \cancel{\phi_1} \\ &\quad + a_2 + \cancel{\phi_1} - \cancel{\phi_2} \\ &\quad + a_3 + \cancel{\phi_2} - \cancel{\phi_3} \\ &\quad \vdots \\ &\quad + a_n + \cancel{\phi_{n-1}} - \cancel{\phi_n} \end{aligned}$$

if $\phi_0 = 0$
 $\sum t_i \leq \sum a_i$
 because $\phi_n \geq 0$

Binary Counter: Potential Method

- Potential function:

Φ : number of ones in current counter

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

- Actual cost t_i :

- 1 flip from 0 to 1
- $t_i - 1$ flips from 1 to 0



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

- Amortized cost: $a_i = t_i + \underbrace{\Phi_i - \Phi_{i-1}}_{(2-t_i)} = 2$

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

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

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)

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

- We will now need the marks...

$\leq O(|V|)$ op. of type ins, is-empty, ...
~~del.-min~~

- Cost for Dijkstra: $O(|E| + |V| \log |V|)$ $\leq |E|$ decr. -key op

previously: $O(|E| \log |V|)$

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 $e \rightarrow H'$
- merge heaps H and H'
(appending a one-node tree to the rootlist)

Get minimum element of H :

- return $H.min$

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 $D(n)$

n elements

Operation Delete-Min

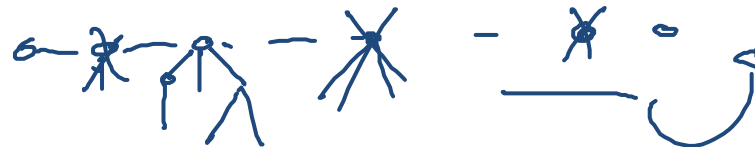
Delete the node with minimum key from H and return its element:

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. **$H.Consolidate();$**

time: $O(|H.rootlist| + \underline{D(u)})$

// 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

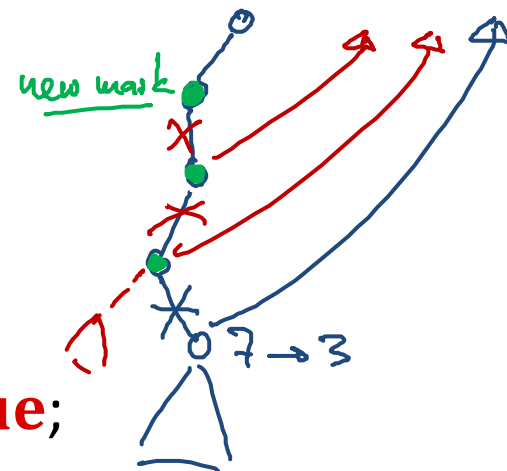
6. **return** m



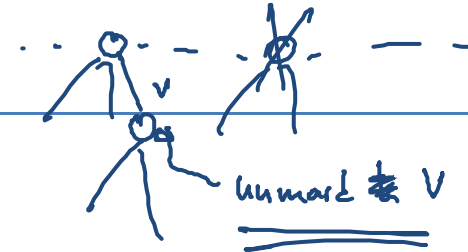
Operation Decrease-Key

Decrease-Key(v, 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** $v \in H.rootlist \vee x \geq v.parent.key$ **then return**
4. **repeat**
5. $parent := v.parent$;
6. **$H.cut(v)$;**
7. $v := parent$;
8. **until** $\neg(v.mark) \vee v \in H.rootlist$;
9. **if** $v \notin H.rootlist$ **then** $v.mark := true$;



Fibonacci Heaps: Marks



Cycle of a node:

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



$v.mark = false$

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

$v.mark = true$

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.

Actual Time of Operations

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

actual time: $O(1)$

- Normalize unit time such that

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

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

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

- Normalize unit time such that

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

↑ before op.

- Operation ***decrease-key***:

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

- Normalize unit time such that

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



Potential Function

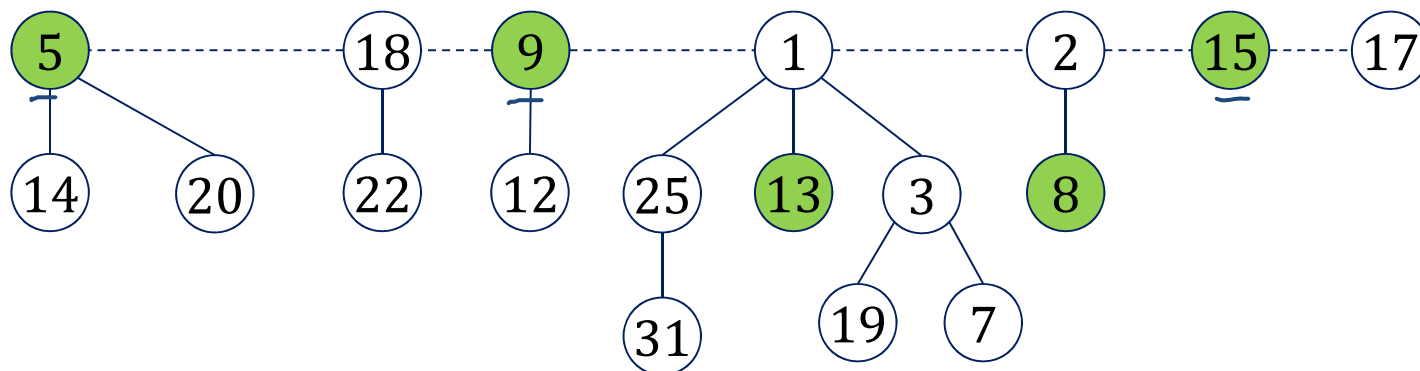
System state characterized by two parameters:

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

Potential function:

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

Example:



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

Amortized Times

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



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)

$$\begin{array}{l} M_i = M_{i-1}, \quad R_i = R_{i-1} + 1 \\ \Phi_i = \Phi_{i-1} + 1 \end{array}$$

- **Amortized time:**

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

Amortized Time of Delete-Min $\phi = R + 2M$

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

Actual time: $t_i \leq \underline{D(n) + |H.rootlist|}$ $R_i \leq \underline{D(n) + 1} + R_{i-1} - \underbrace{R_{i-1}}_{|H.rootlist|}$

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

- R : changes from $H.rootlist$ to at most $\underline{D(n) + 1}$
- 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 increase!

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

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

Amortized Time:

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

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 . 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 $\geq k$

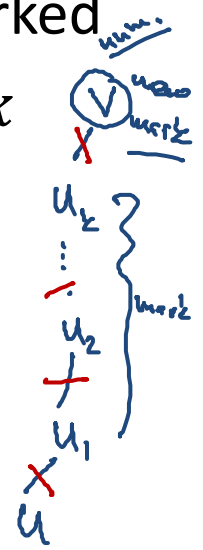
$$R_i \leq R_{i-1} + k + 1, \quad M_i \leq M_{i-1} + 1 - k$$

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


Amortized time:

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

act. cost
 $\Delta\Phi$



Complexities Fibonacci Heap

- Initialize-Heap: $O(1)$
 - Is-Empty: $O(1)$
 - Insert: $O(1)$
 - Get-Min: $O(1)$
 - Delete-Min: $O(D(n))$
 - Decrease-Key: $O(1)$
 - Merge (heaps of size m and $n, m \leq n$): $O(1)$
 - How large can $D(n)$ get?
- amortized**
- 

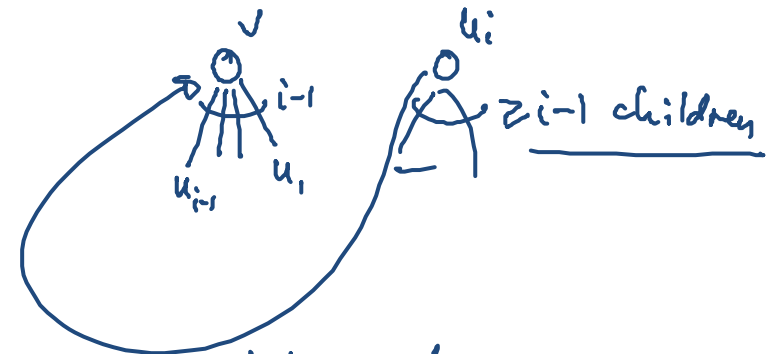
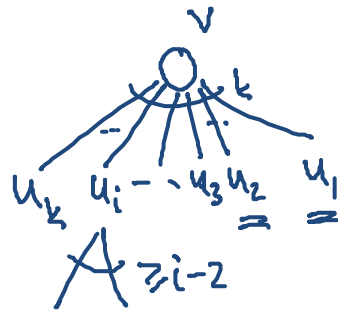
Rank of Children

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,

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

Proof:



when u_i is moved to v , it has $\geq i-1$ children

u_i can only lose 1 child before being cut

Size of Trees

$$F_2=1, F_3=2, F_4=3, F_5=5, F_6=8, F_7=13$$



Fibonacci Numbers:

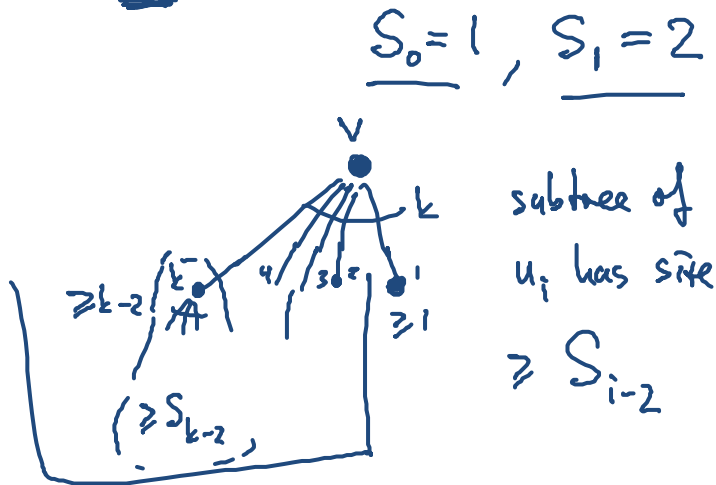
$$F_0 = 0, \quad F_1 = 1, \quad \forall k \geq 2: F_k = F_{k-1} + F_{k-2}$$

Lemma:

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

Proof:

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



$$S_0 = 1, \quad S_1 = 2$$

subtree of u_i has size $\geq S_{i-2}$

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

(by prev. lemma)

Size of Trees

0, 1, 1, 2, 3
 F_2

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

- Claim about Fibonacci numbers:

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

Proof: by ind. on k

$k=0$: $F_2 = 1$

$k > 0$: $F_{k+2} = F_k + F_{k+1}$
 \equiv $\underbrace{F_{k+1}}_{\text{by ind.}} = 1 + \sum_{i=0}^{k-1} F_i = 1 + \sum_{i=0}^k F_i$

Size of Trees



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

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

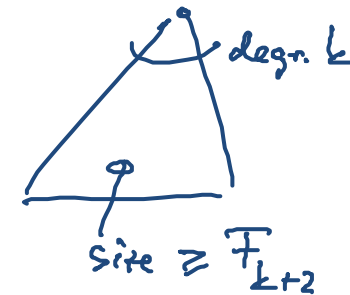
Ind. on k

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

step: $S_k \geq 2 + \sum_{i=0}^{k-2} S_i \stackrel{(I.H.)}{\geq} 2 + \sum_{i=0}^{k-2} F_{i+2}$

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

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



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

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

Proof:

- The Fibonacci numbers grow exponentially:

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

↖ 0

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

Summary: Binomial and Fibonacci Heaps



	Binomial Heap	Fibonacci Heap
<i>initialize</i>	$O(1)$	$O(1)$
<i>insert</i>	$O(\log n)$	$O(1)$
<i>get-min</i>	$O(1)$	$O(1)$
<i>delete-min</i>	$O(\log n)$	$O(\log n)^*$
<i>decrease-key</i>	$O(\log n)$	$O(1)^*$
<i>merge</i>	$O(\log n)$	$O(1)$
<i>is-empty</i>	$O(1)$	$O(1)$

* amortized time