Chapter 4
Data Structures

Algorithm Theory
WS 2014/15

Fabian Kuhn

Amortized Analysis

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

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

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	0001**1**	1
4	00**100**	3
5	0010**1**	1
6	001**10**	2
7	0011**1**	1
8	0**1000**	4
9	0100**1**	1
10	010**10**	2
11	0101**1**	1
12	01**100**	3
13	0110**1**	1

Accounting Method

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

Potential Function Method

- Often a more elegant way to do amortized analysis!
  - But, also more abstract…
- State of data structure / system: $\mathcal{S} \in \mathcal{S}$ (state space)

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

- **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:

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

Potential Function Method

Operation i:

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

Overall cost:

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

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

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

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

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)

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

- We will now need the marks…

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

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 → H'
- merge heaps H and H'

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 $D(n)$

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 $H.rootlist$
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

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

Fibonacci Heaps: Marks

Cycle of a node:

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

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

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

   $\boldsymbol{v.mark} = \textbf{true}$

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

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

- 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

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

Potential Function

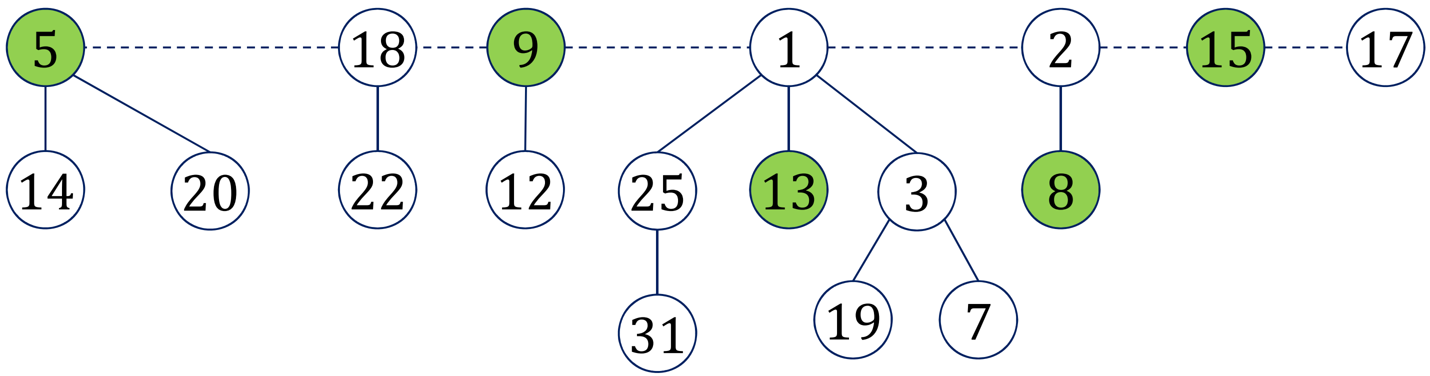
System state characterized by two parameters:

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

Potential function:

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

Example:



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

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: $t_i \leq D(n) + |H.rootlist|$

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

- R: changes from $H.rootlist$ to at most $D(n)$
- 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 increase!

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

Amortized Time:

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

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

- 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
- $\geq k$ 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:

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

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 $\boldsymbol{D(n)}$ 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:

Size of Trees

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}.

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**