ETH zürich

Exercise Session 04 – Amortized Analysis

Data Structures and Algorithms

These slides are based on those of the lecture, but were adapted and extended by the teaching assistant Adel Gavranović

Today's Schedule

Intro
Follow-up
Feedback for **code** expert
Learning Objectives
Entry Quiz
Amortized Analysis
Code-Example: Dynamically Sized Array
Tips for **code** expert
Old Exam Question
Outro

n.ethz.ch/~agavranovic

- Exercise Session Material
- Adel's Webpage
- Mail to Adel

Comic of the Week

xkcd

1. Intro

Intro

Intro

- You get the XP points to unlock the bonus tasks with way fewer than all points (i.e. 1/3 usually suffices)

2. Follow-up

Follow-up from last exercise session

Follow-up from last exercise session

- The code examples (we skipped last week) are good exam prep

Follow-up from last exercise session

- The code examples (we skipped last week) are good exam prep
- If time allows, we'll finish the rest of the previous session

Follow-up from last exercise session

- The code examples (we skipped last week) are good exam prep
- If time allows, we'll finish the rest of the previous session
- If time allows, we'll have a look at an exam question

3. Feedback for **code** expert

Task "Some Proofs"

Task "Some Proofs"

- use counterexamples whenever you can – they're easy to proof and even easier to correct ;)

Task "Some Proofs"

- use counterexamples whenever you can – they're easy to proof and even easier to correct ;)
- the majority seems to grap the concepts well, but the "mathy proofs" are lacking – make sure to study the master solutions

Questions regarding **code** expert from your side?

4. Learning Objectives

Learning Objectives

- ☐ Understand the basics of the three *Amortized Analysis* methods
  - ☐ Aggregate Analysis
  - ☐ Account Method
  - ☐ Potential Method
- ☐ Be prepared for Double Ended Queue exercise on **code** expert

5. Entry Quiz

Quiz

Among a huge number (n) of students present, a price will be awarded to the student with the median Legi number. There is an argument what kind of algorithm shall be used to find this student. Mark the correct statements.

(1) In order to have a worst case runtime of $\mathcal{O}(n \log n)$, we use

- BubbleSort
- Selection Sort
- Mergesort
- Quicksort

Quiz

Among a huge number (n) of students present, a price will be awarded to the student with the median Legi number. There is an argument what kind of algorithm shall be used to find this student. Mark the correct statements.

(1) In order to have a worst case runtime of $\mathcal{O}(n \log n)$, we use

- BubbleSort
- Selection Sort
- Mergesort ☺
- Quicksort

Quiz

Among a huge number (n) of students present, a price will be awarded to the student with the median Legi number. There is an argument what kind of algorithm shall be used to find this student. Mark the correct statements.

(2) We use Quickselect with random pivot choice. Then we have

- a worst case running time of $\mathcal{O}(n \log n)$
- a worst case running time of $\mathcal{O}(n)$
- an expected running time of $\mathcal{O}(\log n)$
- an expected running time of $\mathcal{O}(n)$

Quiz

Among a huge number (n) of students present, a price will be awarded to the student with the median Legi number. There is an argument what kind of algorithm shall be used to find this student. Mark the correct statements.

(2) We use Quickselect with random pivot choice. Then we have

- a worst case running time of $\mathcal{O}(n \log n)$
- a worst case running time of $\mathcal{O}(n)$
- an expected running time of $\mathcal{O}(\log n)$
- an expected running time of $\mathcal{O}(n)$ ☺

6. Amortized Analysis

Amortized Analysis

Three Methods

Amortized Analysis

"Amortized constant runtime"

→ $O(1)$

Three Methods

- Aggregate analysis
- Account Method
- Potential Method

Example: simple multi-set

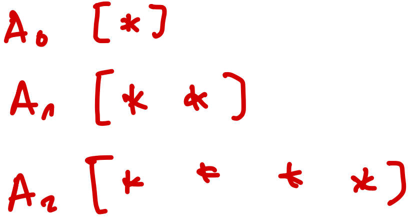
Supports operations `Insert` and `Find`.

Example: simple multi-set

Supports operations `Insert` and `Find`.

Idea:

- Collection of arrays A_i with Length 2^i

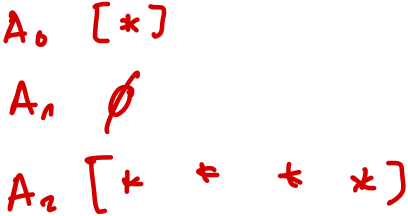


Example: simple multi-set

Supports operations `Insert` and `Find`.

Idea:

- Collection of arrays A_i with Length 2^i
- Every array is either entirely empty or entirely full and stores items in a sorted order



Example: simple multi-set

Supports operations `Insert` and `Find`.

Idea:

- Collection of arrays A_i with Length 2^i
- Every array is either entirely empty or entirely full and stores items in a sorted order
- Between the arrays there is no further relationship

Example: simple multi-set

Supports operations `Insert` and `Find`.

Idea:

- Collection of arrays A_i with Length 2^i
- Every array is either entirely empty or entirely full and stores items in a sorted order
- Between the arrays there is no further relationship

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$
\begin{aligned}
A_0 &: \quad [50] \\
A_1 &: \quad [8, 99] \\
A_2 &: \quad \emptyset \\
A_3 &: \quad [1, 10, 18, 20, 24, 36, 48, 75]
\end{aligned}
$$

We use 0-indexing, such that for the lengths $|A_i| = 2^i$.

Example: simple multi-set

For any $n \in \mathbb{N}$, we can store exactly n elements in our multi set, without partially-filled arrays. Intuition: binary representation of n.

$$
\begin{aligned}
\text{\#elements in multi-set} &= |A_k| + |A_{k-1}| + \ldots + |A_0| \\
&= b_k 2^k + b_{k-1} 2^{k-1} + \ldots + b_0 2^0 \\
&= (b_k \; b_{k-1} \; \ldots \; b_0)_2
\end{aligned}
$$

Where $b_i = 0$ if $|A_i| = 0$, and 1 if $|A_i| = 2^i$.

Example: simple multi-set

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$A_0: \quad [50]$$
$$A_1: \quad [8, 99]$$
$$A_2: \quad \emptyset$$
$$A_3: \quad [1, 10, 18, 20, 24, 36, 48, 75]$$

Algorithm `Find`:

Example: simple multi-set

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{aligned} A_0 &: \quad [50] \\ A_1 &: \quad [8, 99] \\ A_2 &: \quad \emptyset \\ A_3 &: \quad [1, 10, 18, 20, 24, 36, 48, 75] \end{aligned}$$

Algorithm `Find`: Perform a binary search on each array

Worst-case Runtime:

Example: simple multi-set

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{aligned} A_0&: [50] \\ A_1&: [8, 99] \\ A_2&: \emptyset \\ A_3&: [1, 10, 18, 20, 24, 36, 48, 75] \end{aligned}$$

Algorithm `Find`: Perform a binary search on each array
Worst-case Runtime: $\Theta(\log^2 n)$,

Example: simple multi-set

Data $\{1, 8, 10, 18, 20, 24, 36, 48, 50, 75, 99\}$, $n = 11$

$$\begin{aligned} A_0&: \quad [50] \\ A_1&: \quad [8, 99] \\ A_2&: \quad \emptyset \\ A_3&: \quad [1, 10, 18, 20, 24, 36, 48, 75] \end{aligned}$$

Algorithm `Find`: Perform a binary search on each array
Worst-case Runtime: $\Theta(\log^2 n)$,

$$\log 1 + \log 2 + \log 4 + \cdots + \log 2^k = \sum_{i=0}^{k} \log_2 2^i = \frac{k \cdot (k+1)}{2} \in \Theta(\log^2 n).$$

$(k = \lfloor \log_2 n \rfloor)$

Example: simple multi-set

Algorithm `Insert(x)`:

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$

A'_0 [x]

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i + 1$

```
while ( Ai ≠ ∅){
    set A'i+1 = Merge(Ai, A'i);
    Ai ← ∅;
    i++ ;
}
```

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i + 1$
- Set $A_i \leftarrow A'_i$

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i + 1$
- Set $A_i \leftarrow A'_i$

Insert(11)

Pre-insert

A_0: $[50]$ ∅

A_1: $[8, 99]$ ∅

A_2: $\emptyset$

A_3: $[1, 10, 18, \ldots, 75]$

A'_0: [11]

A'_1: [11, 50]

A'_2: [8, 11, 50, 99]

Post-insert

A_0 ∅

A_1 ∅

A_2 [8 11 50 99]

A_3 [1, 10, 18 ...]

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i + 1$
- Set $A_i \leftarrow A'_i$

Insert(11)

	Pre-insert	Temporary
A_0:	$[50]$	
A_1:	$[8, 99]$	
A_2:	$\emptyset$	
A_3:	$[1, 10, 18, \ldots, 75]$	

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Insert(11)

	Pre-insert		Temporary
A_0:	$[50]$	A'_0:	$[11]$
A_1:	$[8, 99]$		
A_2:	$\emptyset$		
A_3:	$[1, 10, 18, \ldots, 75]$		

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Insert(11)

	Pre-insert		Temporary
A_0:	$[50]$	A'_0:	$[11]$
A_1:	$[8, 99]$	A'_1:	$[11, 50]$
A_2:	$\emptyset$		
A_3:	$[1, 10, 18, \ldots, 75]$		

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i+1$
- Set $A_i \leftarrow A'_i$

Insert(11)

	Pre-insert		Temporary
A_0:	$[50]$	A'_0:	$[11]$
A_1:	$[8, 99]$	A'_1:	$[11, 50]$
A_2:	$\emptyset$	A'_2:	$[8, 11, 50, 99]$
A_3:	$[1, 10, 18, \ldots, 75]$		

Example: simple multi-set

Algorithm `Insert(x)`:

- New array $A'_0 \leftarrow [x]$, $i \leftarrow 0$
- while $A_i \neq \emptyset$, set A'_{i+1} =Merge(A_i, A'_i), $A_i \leftarrow \emptyset$, $i \leftarrow i + 1$
- Set $A_i \leftarrow A'_i$

Insert(11)

Pre-insert		Temporary			Post-insert	
A_0:	$[50]$	A'_0:	$[11]$		A_0:	$\emptyset$
A_1:	$[8, 99]$	A'_1:	$[11, 50]$	$\Longrightarrow$	A_1:	$\emptyset$
A_2:	$\emptyset$	A'_2:	$[8, 11, 50, 99]$		A_2:	$[8, 11, 50, 99]$
A_3:	$[1, 10, 18, \ldots, 75]$				A_3:	$[1, 10, 18, \ldots, 75]$

Costs insert

In the following example: $n = 2^k$, $k = \log_2 n$

Costs insert

In the following example: $n = 2^k$, $k = \log_2 n$

Assumption: creating new array A'_i with length 2^i (and, for $i > 0$ subsequent merge of A'_{i-1} and A_{i-1}) has costs $\Theta(2^i)$

Costs insert

In the following example: $n = 2^k$, $k = \log_2 n$

Assumption: creating new array A'_i with length 2^i (and, for $i > 0$ subsequent merge of A'_{i-1} and A_{i-1}) has costs $\Theta(2^i)$

In the worst case, inserting an element into the data structure provides $\log_2 n$ such operations.

Costs insert

In the following example: $n = 2^k$, $k = \log_2 n$

Assumption: creating new array A'_i with length 2^i (and, for $i > 0$ subsequent merge of A'_{i-1} and A_{i-1}) has costs $\Theta(2^i)$

In the worst case, inserting an element into the data structure provides $\log_2 n$ such operations.

$\Rightarrow$ **Worst-case Costs Insert**:

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \in \Theta(n).$$

Aggregate analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*, *]$
2	4	$[*, *, *, *]$
3	8	$\emptyset$
4	16	$[*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Aggregate analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*, *]$
2	4	$[*, *, *, *]$
3	8	$\emptyset$
4	16	$[*, *, *, *, *, *, *, *, *, *, *, *, *, *, *, *]$

Observation: Starting with an empty container, an insertion sequence reaches level 0 each time, level 1 (with costs 2) every second time, level 2 (with costs 4) every fourth time, etc.

Aggregate analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*,*]$
2	4	$[*,*,*,*]$
3	8	$\emptyset$
4	16	$[*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*]$

Observation: Starting with an empty container, an insertion sequence reaches level 0 each time, level 1 (with costs 2) every second time, level 2 (with costs 4) every fourth time, etc.

- Total costs: $1 \cdot \frac{n}{1} + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \cdots + 2^k \cdot \frac{n}{2^k} =$

Aggregate analysis

Level	Costs	Example Array
0	1	$[*]$
1	2	$[*,*]$
2	4	$[*,*,*,*]$
3	8	$\emptyset$
4	16	$[*,*,*,*,*,*,*,*,*,*,*,*,*,*,*,*]$

Observation: Starting with an empty container, an insertion sequence reaches level 0 each time, level 1 (with costs 2) every second time, level 2 (with costs 4) every fourth time, etc.

- Total costs: $1 \cdot \frac{n}{1} + 2 \cdot \frac{n}{2} + 4 \cdot \frac{n}{4} + \cdots + 2^k \cdot \frac{n}{2^k} = (k+1)n$
  This is in $\Theta(n \log n)$ because $k = \log_2 n$.
- **Amortized cost per operation**: $\Theta((n \log n)/n) = \Theta(\log n)$.

Account method

Ideas?

Account method

- Every element i $(1 \leq i \leq n)$ pays $a_i = \log_2 n$ coins when it is inserted into the data structure.

Account method

- Every element i $(1 \leq i \leq n)$ pays $a_i = \log_2 n$ coins when it is inserted into the data structure.
- The element pays the allocation of the first array and every subsequent merge-step that can occur until the element has reached array A_{k+1} $(k = \lfloor \log_2 n \rfloor)$.

Account method

- Every element i $(1 \leq i \leq n)$ pays $a_i = \log_2 n$ coins when it is inserted into the data structure.
- The element pays the allocation of the first array and every subsequent merge-step that can occur until the element has reached array A_{k+1} $(k = \lfloor \log_2 n \rfloor)$.
- The account provides enough credit to pay for all Merge operations of the n elements.

Account method

"guess a cost" and proof that account never runs out of credit

- Every element i $(1 \leq i \leq n)$ pays $a_i = \log_2 n$ coins when it is inserted into the data structure.
- The element pays the allocation of the first array and every subsequent merge-step that can occur until the element has reached array A_{k+1} $(k = \lfloor \log_2 n \rfloor)$.
- The account provides enough credit to pay for all Merge operations of the n elements.

$\Rightarrow$ **Amortized costs** for insertion $\mathcal{O}(\log n)$

Potential method

Ideas?

Potential method

We know from the account method that **each element on the way to higher levels requires $\log n$ coins**, i.e. that an element on level i still needs to posess $k - i$ coins.

Potential method

We know from the account method that **each element on the way to higher levels requires** $\log n$ **coins**, i.e. that an element on level i still needs to posess $k - i$ coins. We use the **potential**

Potential method

We know from the account method that **each element on the way to higher levels requires** $\log n$ **coins**, i.e. that an element on level i still needs to posess $k - i$ coins. We use the **potential**

$$\Phi_j = \sum_{0 \le i \le k : A_i \neq \emptyset} (k - i) \cdot 2^i$$

Potential method

For the **change of the potential** $\Phi_j - \Phi_{j-1}$ we only have to consider the lower l levels that are occupied at time point $j-1$ (in analogy to the binary counter). Let l be the smallest index such that array A_l is empty.

Potential method

For the **change of the potential** $\Phi_j - \Phi_{j-1}$ we only have to consider the lower l levels that are occupied at time point $j-1$ (in analogy to the binary counter). Let l be the smallest index such that array A_l is empty.

After merging arrays $A_0 \ldots A_{l-1}$, array A_l is full and arrays $A_i (0 \leq i < l)$ are now empty. Therefore:

Potential method

For the **change of the potential** $\Phi_j - \Phi_{j-1}$ we only have to consider the lower l levels that are occupied at time point $j-1$ (in analogy to the binary counter). Let l be the smallest index such that array A_l is empty.

After merging arrays $A_0 \ldots A_{l-1}$, array A_l is full and arrays $A_i (0 \leq i < l)$ are now empty. Therefore:

$$\Phi_j - \Phi_{j-1} = (k-l) \cdot 2^l - \sum_{i=0}^{l-1} (k-i) \cdot 2^i$$

Potential method

For the **change of the potential** $\Phi_j - \Phi_{j-1}$ we only have to consider the lower l levels that are occupied at time point $j-1$ (in analogy to the binary counter). Let l be the smallest index such that array A_l is empty.

After merging arrays $A_0 \ldots A_{l-1}$, array A_l is full and arrays $A_i (0 \leq i < l)$ are now empty. Therefore:

$$\Phi_j - \Phi_{j-1} = (k-l) \cdot 2^l - \sum_{i=0}^{l-1} (k-i) \cdot 2^i$$

Real costs:

$$t_j = \sum_{i=0}^{l} 2^i = 2^{l+1} - 1$$

Potential method

Potential method

$$\begin{aligned}\Phi_j - \Phi_{j-1} &= (k-l)\cdot 2^l - \sum_{i=0}^{l-1}(k-i)\cdot 2^i \\ &= (k-l)\cdot 2^l - k\cdot(2^l-1) + \sum_{i=0}^{l-1} i\cdot 2^i \\ &= (k-l)\cdot 2^l - k\cdot(2^l-1) + l\cdot 2^l - 2^{l+1} + 2 \\ &= k - 2^{l+1} + 2\end{aligned}$$

$$\implies \Phi_j - \Phi_{j-1} + t_j = k - 2^{l+1} + 2 + 2^{l+1} - 1 = k + 1 \in \Theta(\log n)$$

Potential method

$$
\begin{aligned}
\Phi_j - \Phi_{j-1} &= (k-l)\cdot 2^l - \sum_{i=0}^{l-1}(k-i)\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l - 1) + \sum_{i=0}^{l-1} i\cdot 2^i \\
&= (k-l)\cdot 2^l - k\cdot(2^l - 1) + l\cdot 2^l - 2^{l+1} + 2 \\
&= k - 2^{l+1} + 2
\end{aligned}
$$

$$
\implies \Phi_j - \Phi_{j-1} + t_j = k - 2^{l+1} + 2 + 2^{l+1} - 1 = k + 1 \in \Theta(\log n)
$$

See CLRS Chapter 16.

$\sum i \cdot \lambda^i$

Always the same trick:

$\sum i \cdot \lambda^i$

Always the same trick:

$$\lambda \cdot \sum_{i=0}^{n} i \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=0}^{n} i \cdot \lambda^{i+1} - \sum_{i=0}^{n} i \cdot \lambda^i = \sum_{i=1}^{n+1} (i-1) \cdot \lambda^i - \sum_{i=0}^{n} i \cdot \lambda^i$$

$$= n \cdot \lambda^{n+1} + \sum_{i=1}^{n} (i-1) \cdot \lambda^i - i \cdot \lambda = n \cdot \lambda^{n+1} - \sum_{i=1}^{n} \lambda^i$$

$$= n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1$$

$$\implies (\lambda - 1) \cdot \sum_{i=0}^{n} i \cdot \lambda^i = n \cdot \lambda^{n+1} - \frac{\lambda^{n+1} - 1}{\lambda - 1} + 1$$

For $\lambda = 2$:

$$\sum_{i=0}^{n} i \cdot 2^i = n \cdot 2^{n+1} - 2^{n+1} + 1 + 1 = (n-1) \cdot 2^{n+1} + 2$$

Quiz

```
void g(unsigned n){
  for (unsigned k = 1; k != n ; ++k){
     // what does the following code do?
    unsigned prev = k-1;
    for (unsigned num = k; num != 0; num /= 2){
      if (num % 2 != prev % 2)
        f();
      prev /= 2;
    }
  }
}
```

Q: Asymptotic number of calls of f?

Quiz

```
void g(unsigned n){
  for (unsigned k = 1; k != n ; ++k){
     // call f for all bits that toggle from k-1 to k
    unsigned prev = k-1;
    for (unsigned num = k; num != 0; num /= 2){
      if (num % 2 != prev % 2)
        f();
      prev /= 2;
    }
  }
}
```

k-1 10111

k 11000

Q: Asymptotic number of calls of f?

Quiz

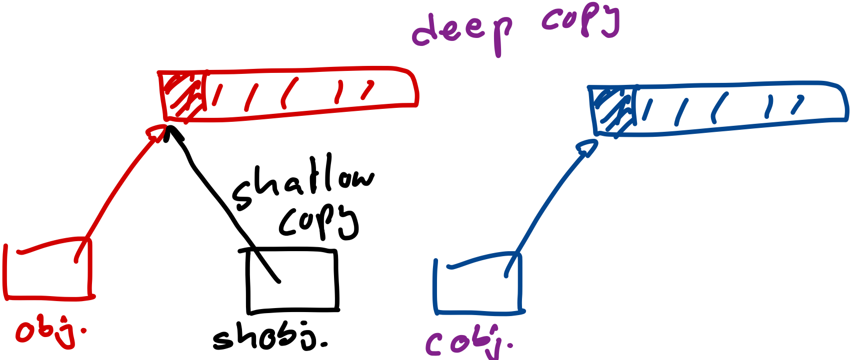
```
void g(unsigned n){
  for (unsigned k = 1; k != n ; ++k){
     // call f for all bits that toggle from k-1 to k
    unsigned prev = k-1;
    for (unsigned num = k; num != 0; num /= 2){
      if (num % 2 != prev % 2)
        f();
      prev /= 2;
    }
  }
}
```

Q: Asymptotic number of calls of f?

A: $\Theta(n)$ (Counting example from class).

Recap dynamically allocated memory

Important: Every `new` needs its `delete` and only one!



Recap dynamically allocated memory

Important: Every `new` needs its `delete` and only one!

Therefore “Rule of three”:

- constructor
- copy constructor
- destructor

Recap dynamically allocated memory

Important: Every `new` needs its `delete` and only one!

Therefore “Rule of three”:

- constructor
- copy constructor
- destructor

Being lazy “Rule of two”:

- never copy (unsafe)
- make copy constructor private (safe) or deleted

7. Code-Example: Dynamically Sized Array

Preparation for **code** expert exercise *Double Ended Queue*

8. Tips for **code** expert

Tips for next **code** expert exercises

Task "Stable and In-Situ Sorting"

Tips for next **code** expert exercises

Task "Stable and In-Situ Sorting"

- "...in their unmodified form..."

Tips for next code expert exercises

Task "Stable and In-Situ Sorting"

- "...in their unmodified form..."

Task "Amortized Analysis: Dynamic Array"

Tips for next code expert exercises

Task "Stable and In-Situ Sorting"

- "...in their unmodified form..."

Task "Amortized Analysis: Dynamic Array"

- Ottman/Widmayer, Chapter 3.3 (depending on version)
- Cormen et al, Chapter 17 (or 16 depending on version)

Tips for next code expert exercises

Task "Stable and In-Situ Sorting"

- "...in their unmodified form..."

Task "Amortized Analysis: Dynamic Array"

- Ottman/Widmayer, Chapter 3.3 (depending on version)
- Cormen et al, Chapter 17 (or 16 depending on version)

Task "Double Ended Queue"

Tips for next code expert exercises

Task "Stable and In-Situ Sorting"

- "...in their unmodified form..."

Task "Amortized Analysis: Dynamic Array"

- Ottman/Widmayer, Chapter 3.3 (depending on version)
- Cormen et al, Chapter 17 (or 16 depending on version)

Task "Double Ended Queue"

- Takes time – make sure to start early!
- Dynamic data types and memory management (fun!)
- By the way: *the name Double Ended Queue may be misleading because it suggests to be implemented with a linked list. This would make it hard, if not impossible, to fulfill the requirements stated above. Rather think of something like a vector and extend it with* `push_front()`

9. Old Exam Question

Recurrence Equation

Gegeben sei die folgende Rekursionsgleichung:

Consider the following recursion equation:

$$T(n) = \begin{cases} 4T(n/2) + 3n, & n > 1 \\ 3 & n = 1 \end{cases}$$

not just asymptotic!

Leiten Sie eine geschlossene (nicht rekursive), einfache Formel für $T(n)$ her. Nehmen Sie an, dass es ein $k \in \mathbb{N}$ gibt mit $2^k = n$. Zeigen Sie mit vollständiger Induktion, dass Ihr Ergebnis stimmt.

Derive a closed (non-recursive), simple formula for $T(n)$. Assume that there is some $k \in \mathbb{N}$ for which $2^k = n$. Prove by induction that your solution is correct.

optional

Hinweis: Es gilt

Hint: it holds that

$$4^{\log_2 n} = n^2 \text{ und } \sum_{i=0}^{k-1} 2^i = 2^k - 1$$

(D&A Exam 25.8.2022)

Recurrence Equation – Solution I

$$
\begin{aligned}
T(n) &= 4T(n/2) + 3n \\
&= 4(4T(n/4) + 3n/2) + 3n \\
&= 4(4(4T(n/8) + 3n/4) + 3n/2) + 3n \\
&= \ldots \\
&= T(1) \cdot 4^k + 3n \cdot \sum_{i=0}^{k-1} 2^i \\
&= 3n^2 + 3n(2^k - 1) \\
&= 3n^2 + 3n(n - 1) \\
&= 3n(2n - 1)
\end{aligned}
$$

(D&A Exam 25.8.2022)

Recurrence Equation – Solution II

Let $f(n) = 3n(2n-1)$

We show that $f(n) = T(n)$ for all n such that there is some $k \in \mathbb{N}$ for which $2^k = n$.

Induction base: It holds that $f(1) = 3 = T(1)$.

Induction step: Assume that $T(n) = f(n)$ (induction hypothesis). We now show that $T(2n) = f(2n)$.

$$
\begin{aligned}
T(2n) &= 4T(n) + 6n \\
&\overset{i.h.}{=} 12n(2n-1) + 6n \\
&= 6n(2(2n-1)+1) \\
&= 3 \cdot 2n(2 \cdot 2n - 1) \\
&= f(2n).
\end{aligned}
$$

(D&A Exam 25.8.2022)

10. Outro

General Questions?

See you next time

Have a nice week!