Algoritmer og Datastrukturer

Amortiseret Analyse [CLRS, kapitel 17]

compact1, compact2, compact3

java.util

Class ArrayList<E>

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:

AttributeList, RoleList, RoleUnresolvedList

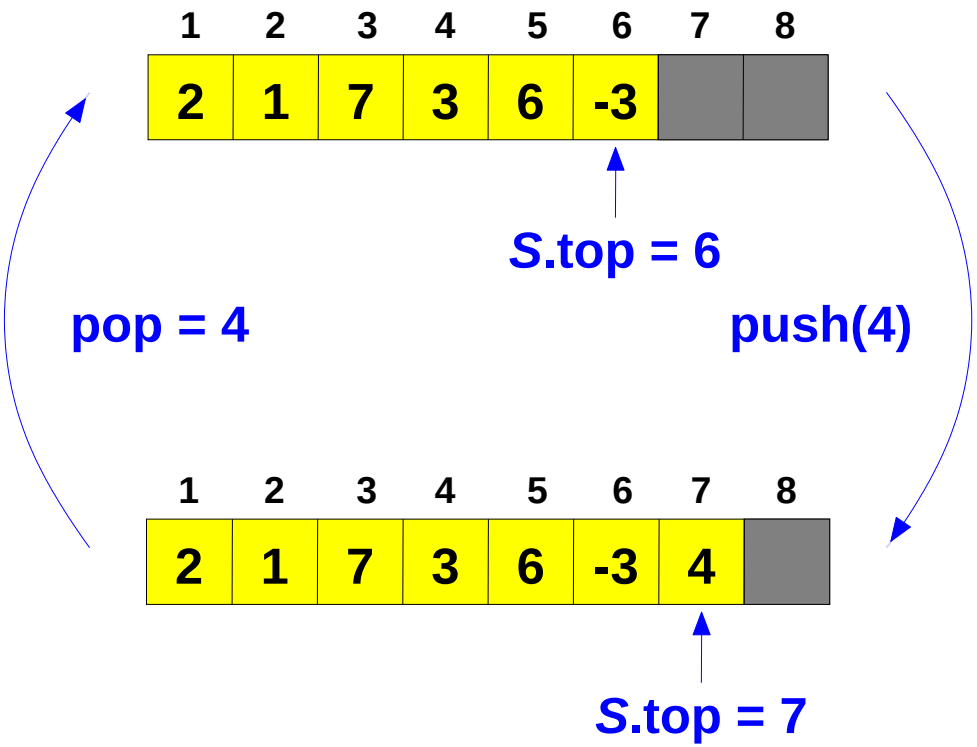
```
public class ArrayList<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`. In addition to implementing the `List` interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to `Vector`, except that it is unsynchronized.)

The `size`, `isEmpty`, `get`, `set`, `iterator`, and `listIterator` operations run in constant time. The add operation runs in *amortized constant time*, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the `LinkedList` implementation.

Stak

Stak : Array Implementation



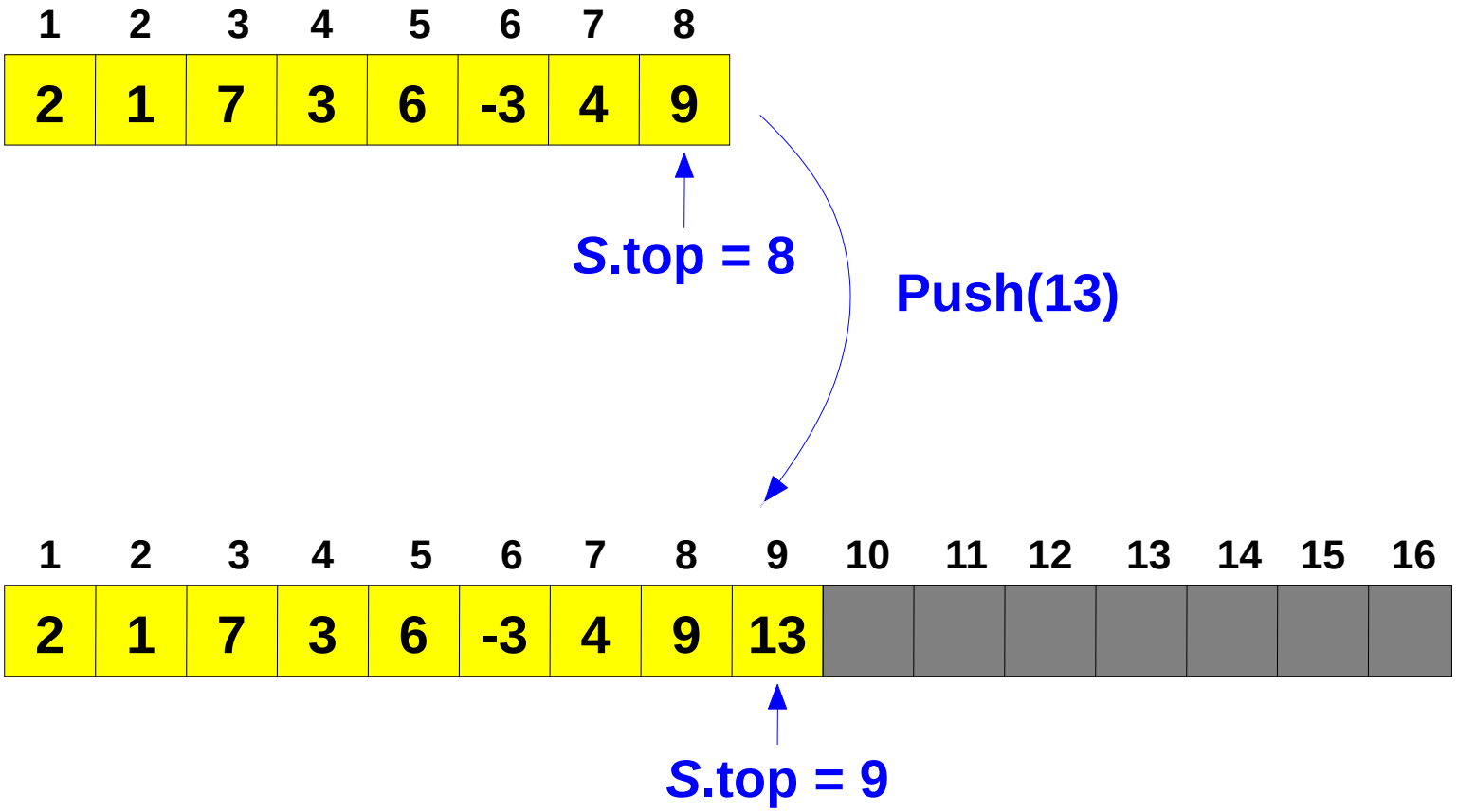

```
STACK-EMPTY(S)
if S.top == 0
    return TRUE
else return FALSE
```

```
PUSH(S, x)
S.top = S.top + 1
S[S.top] = x
```

```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

Stack-Empty, Push, Pop : O(1) **tid**

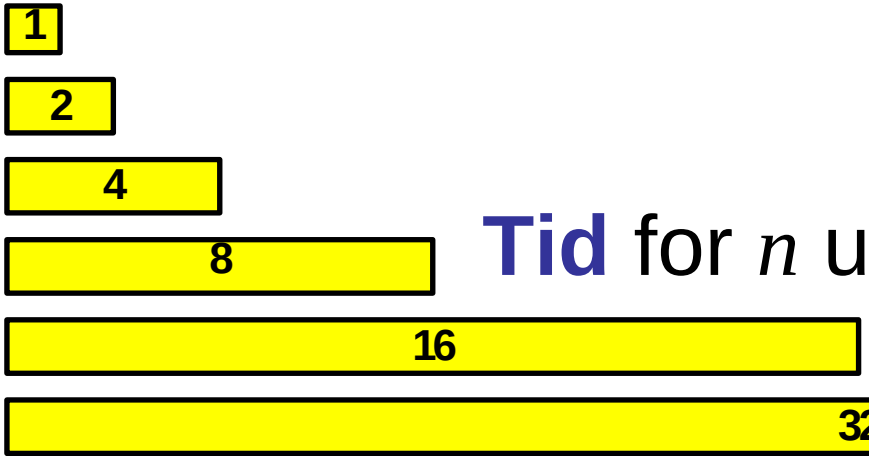
Stak : Overløb



Array fordobling : $O(n)$ **tid**

Array Fordobling

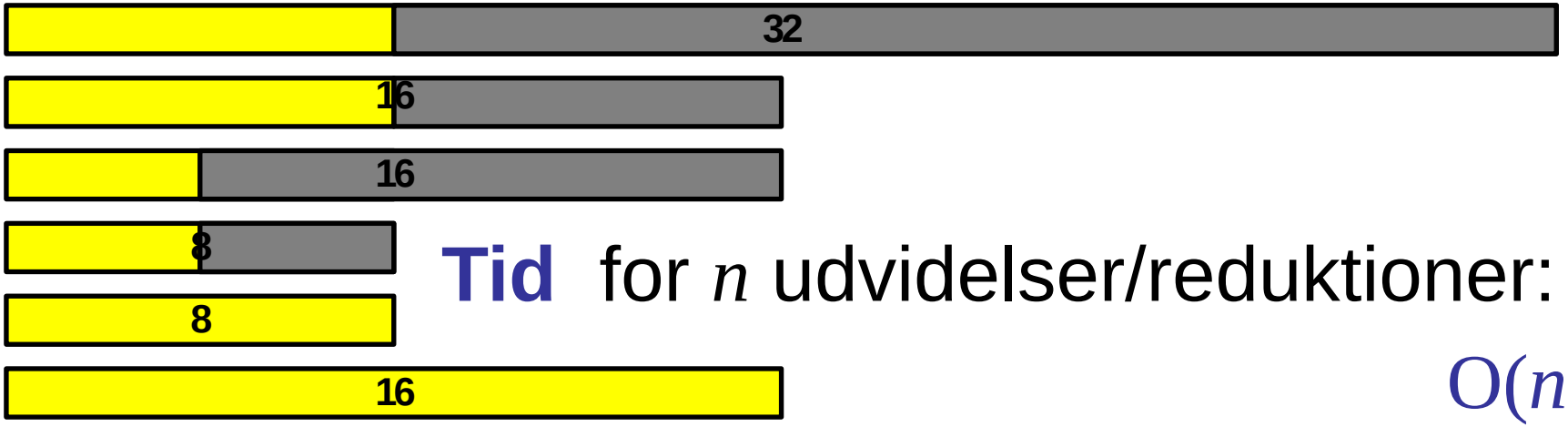
Fordoble arrayet når det er fuld



Tid for n udvidelser:

$$1+2+4+\cdots+n/2+n = O(n)$$

Halver arrayet når det er <1/4 fyldt



Tid for n udvidelser/reduktioner:

$$O(n)$$

Array Fordobling + Halvering
– en generel teknik

Tid for n udvidelser/reduktioner er $O(n)$

Plads $\leq$ 4 $\cdot$ aktuelle antal elementer

Array implementation af Stak:
n push og pop operationer tager $O(n)$ tid

Analyse teknik ønskes...

Krav

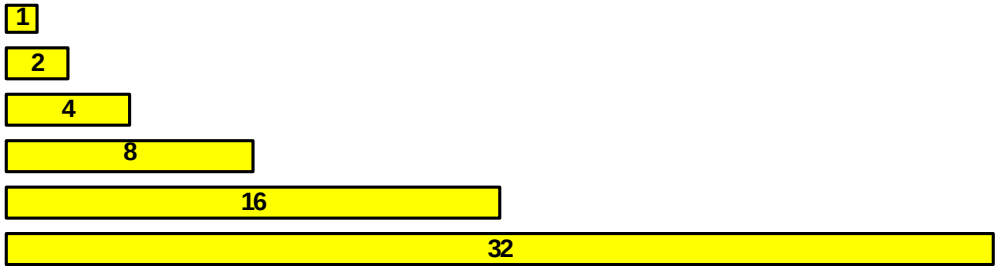
- Analysere **worst-case** tiden for en **sekvens** af operationer
- Behøver kun at analysere den **enkelte operation**

Fordel

- Behøver **ikke** overveje andre operationer i sekvens og deres **indbyrdes påvirkninger**
- Gælder for alle sekvenser med de givne operationer

Intuition

- Der findes ”**gode**”/”**balancerede**” tilstande og ”**dårlige**”/”**ubalancerede**”
- At komme fra en ”**dårlig**” tilstand til en ”**god**” tilstand er **dyrt**
- Det tager mange operationer fra en ”**god**” tilstand før man er i en ”**dårlig**”
- For de (mange) **billige** operationer ”betaler” vi lidt ekstra for senere at kunne lave en **dyr** operation næsten **gratis**

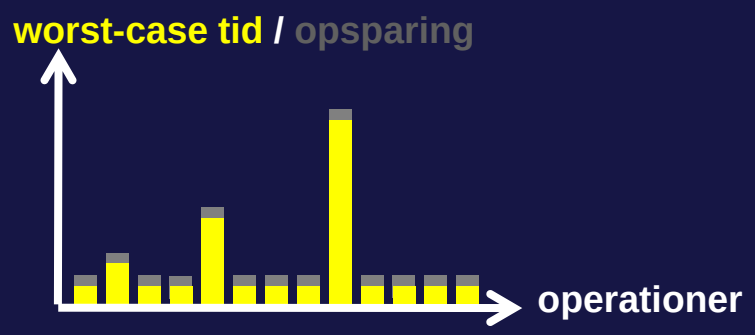


Amortiseret Analyse

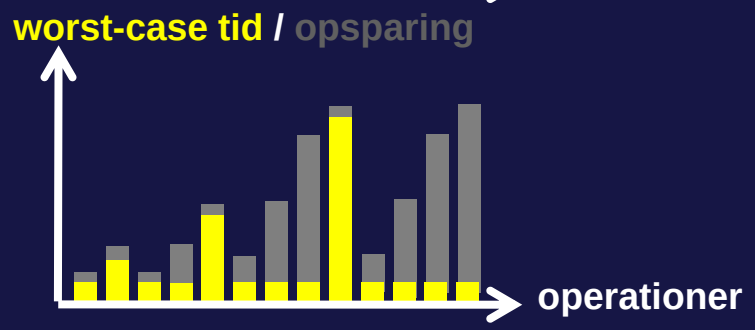
- **1 €** kan betale for $O(1)$ **arbejde**
- En operation der tager tid $O(t)$ koster t €
- Hvornår vi betaler/sparer op er ligegyldigt – bare pengene er der når vi skal bruge dem!
- Opsparing = Potentiale = Φ
- Vi kan ikke låne penge, dvs. vi skal spare op før vi bruger pengene, $\Phi \geq 0$
- Amortiseret tid for en operation = hvad vi er **villige** til at betale – men vi skal have råd til operationen!
- Brug **invarianter** til at beskrive sammenhængen mellem **opsparingen** og **datastrukturens tilstand**

Sammenhæng Mellem Worst-case Tid og Opsparingen Φ

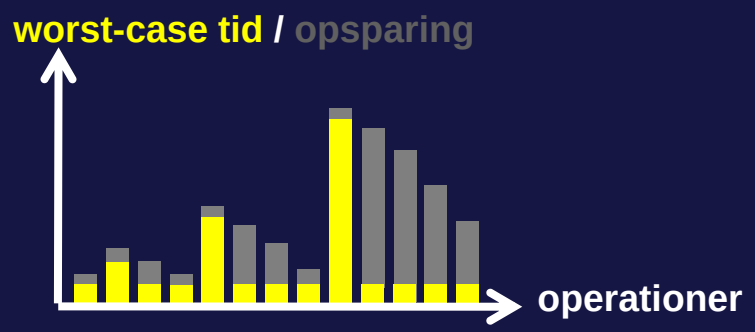
a)



b)



c)

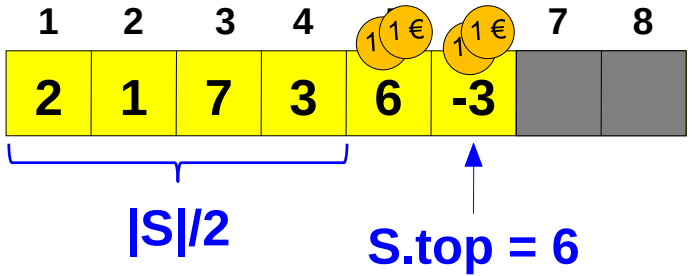


d) Ved ikke

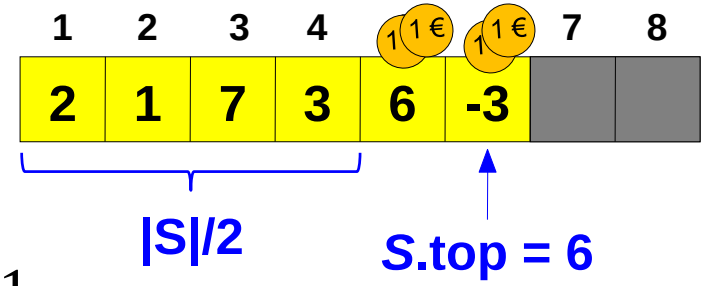
Eksempel: Stak

- En **god** stak er halv fuld – kræver ingen opsparing
- Invariant : $\Phi = 2 \cdot |\, S.top - |S|/2 \,|$
- Antag: 1 € per element indsættelse/kopiering
- Amortiseret tid per push: 3 **€ ?**

  (har vi altid penge til at udføre operationen?)
- Hvis ja: n push operationer koster $\leq 3n$ €



Eksempel: Stak
Push = Amortiseret 3€



- Push uden kopiering:
  - Et nyt element : **1 €**
  - $||S|/2\text{-top}[S]|$ vokser med højst 1, så invarianten holder hvis vi sparer **2 €** op
  - Amortiseret tid: $1+2 =$ **3 €**
- Push med kopiering
  - Kopier S: **$|S|$ €**
  - Indsæt nye element: **1 €**
  - Φ før = $|S|$, Φ efter = 2, dvs **$|S|$-2 € frigives**
  - Amortiseret tid: $|S|+1-(|S|-2) =$ **3 €**

Invariant: $\Phi = 2\cdot|\, S.\text{top}-|S|/2\,|$

Binær Tæller

0000
1 bit
0001
2 bit
0010
1
0011
3
0100
1
0101
2
0110
1
0111
4
1000
...
10101110111111
7
10101111000000

Hvad er en god opsparing- / potentiale- / Φ-funktion ?

a) Positionen af mest betydende 1-tal

b) Positionen af det højreste 0

c) Antal 1'er i det binære tal

d) Antal 0'er i det binære tal

e) Ved ikke

Amortiseret Analyse

- Teknik til at argumenter om **worst-case** tiden for en sekvens af operationer
- Behøver kun at analysere operationerne enkeltvis
- **Kunsten**: Find den rigtige invariant for Φ

Eksempel: Rød-Sorte Træer

Insert(x)

=

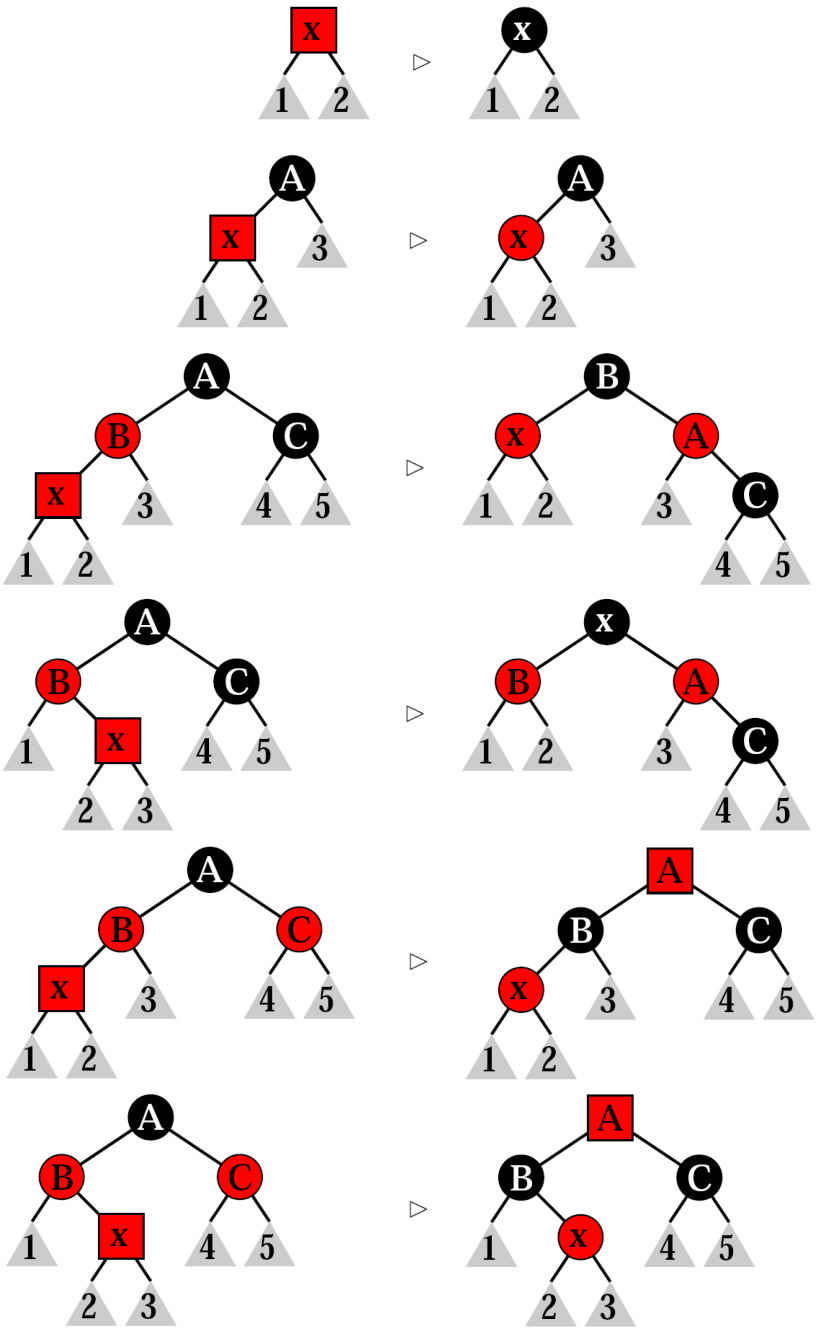
Søgning ← $O(\log n)$

+

Opret nyt rødt blad ← $O(1)$

+

Rebalancering ← # transitioner



transitioner = amortiseret O(1)

Φ = # røde knuder

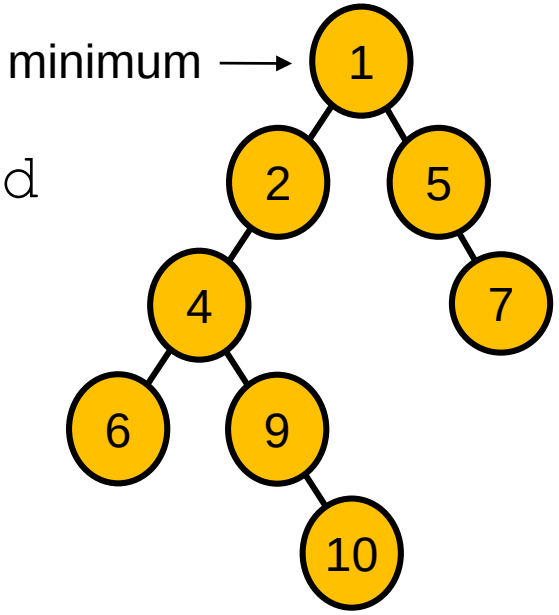
Korollar: Indsættelse i rød-sorte træer tager amortiseret O(1) tid, hvis indsættelsespositionen er kendt

Ikke pensum

Skew heaps – ”selvbalancerende” heaps

Sleator og Tarjan, *Self-Adjusting Heaps*, SIAM Journal on Computing 15(1): 52–69, 1986

- Prioritetskø:
  - `FindMin`, `Insert`, `DeleteMin`, `Meld`
- Heap-ordnet binært træ
  - *Ingen krav til struktur eller dybde*
- Knude: `left`, `right`, `element`
- Prioritetskø = pointer til roden

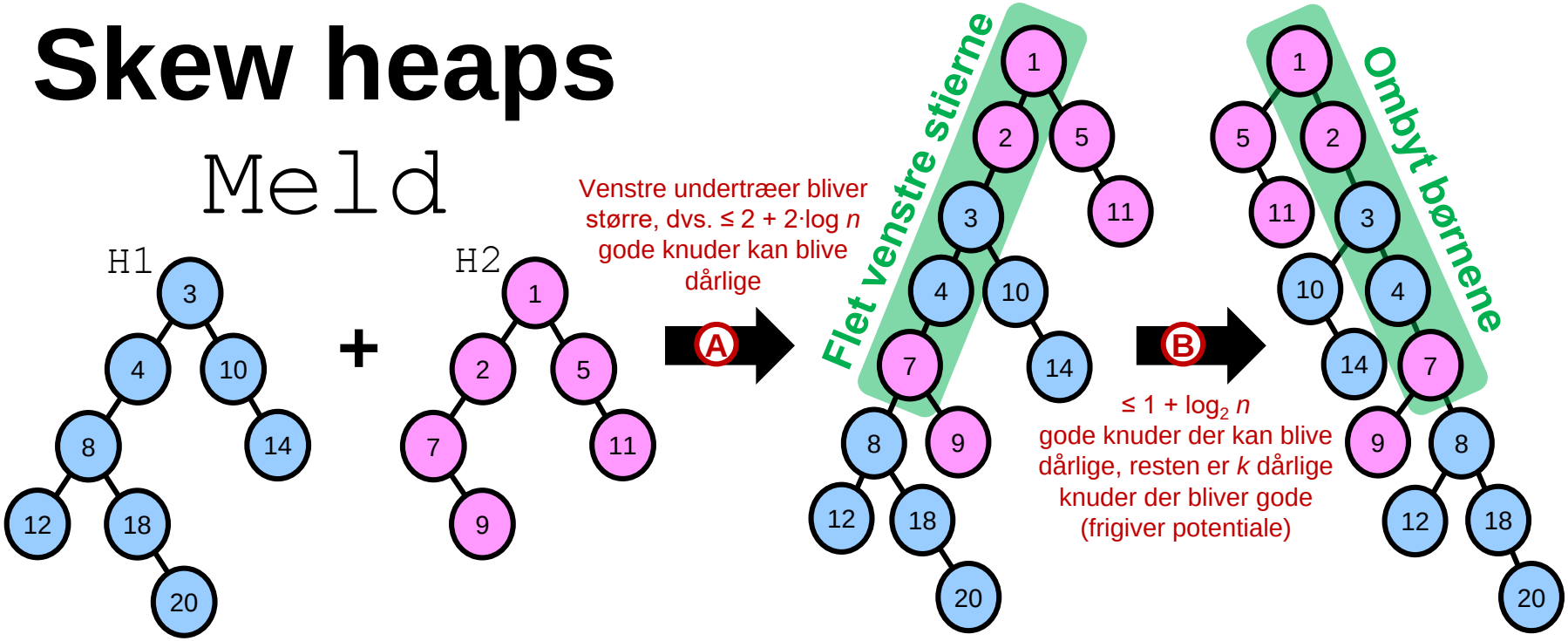



```
proc FindMin(H)
    return H.element

Proc Insert(H, e)
    v = new Node(e, Null, Null)
    return Meld(H, v)

proc DeleteMin(H)
    return Meld(H.left, H.right)
```

(`Meld` next slide)

Skew heaps

Meld




```
proc Meld(H1, H2)
    if H1 = Null then return H2
    if H2 = Null then return H1
    if H1.element < H2.element then
        return new Node(H1.element, H1.right, Meld(H1.left, H2))
    else
        return new Node(H2.element, H2.right, Meld(H2.left, H1))
```

Amortiseret analyse:

Definition God knude `v` : |`v.left`| ≤ |`v`| / 2
Lemma # gode knuder på venstre stien ≤ $1 + \log_2 n$
Potentiale Φ = # dårlige knuder

`Meld` **amortiseret omkostning:**

$$\underbrace{\leq 2 + 2\cdot\log_2 n}_{\Delta\Phi \text{ fra } A} + \underbrace{1 + \log_2 n - k}_{\Delta\Phi \text{ fra } B} + \underbrace{k + 1 + \log_2 n}_{\text{faktisk arbejde}}$$
$= O(\log n)$

⇒ Alle operationer amortiseret time O(log n)

Ikke pensum

Selvbalancerende Datastrukturer

med amortiserede udførselstider

	Skew heaps	Fibonacci heaps	Splay trees
Minimum	$O(1)$	$O(1)$	$O_{AM}(1)^*$
Insert	$O_{AM}(\log n)$	$O_{AM}(1)$	$O_{AM}(\log n)$
DeleteMin	$O_{AM}(\log n)$	$O_{AM}(\log n)$	$O_{AM}(1)^*$
Delete		$O_{AM}(\log n)$	$O_{AM}(\log n)$
Meld	$O_{AM}(\log n)$	$O_{AM}(1)$	
DecreaseKey	$O_{AM}(\log n)$	$O_{AM}(1)$	$O_{AM}(\log n)$
Search			$O_{AM}(\log n)$

Skew Heaps Sleator og Tarjan, SICOMP 1986

Fibonacci heaps Fredman og Tarjan, JACM 1987 [CLRS, kapitel 19]

Splay trees Sleator og Tarjan, JACM 1985 + Cole*, SICOMP 2006

$O_{AM} \equiv$ amortiseret tid