

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
```

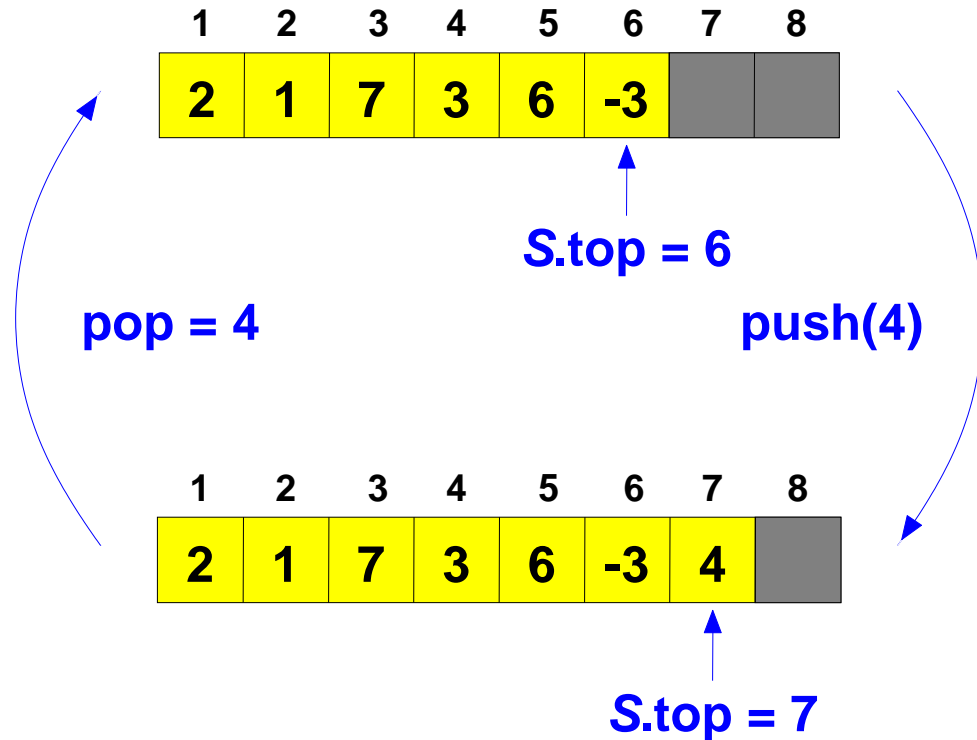
Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class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

```
1  if  $S.top == 0$ 
2      return TRUE
3  else return FALSE
```

PUSH(S, x)

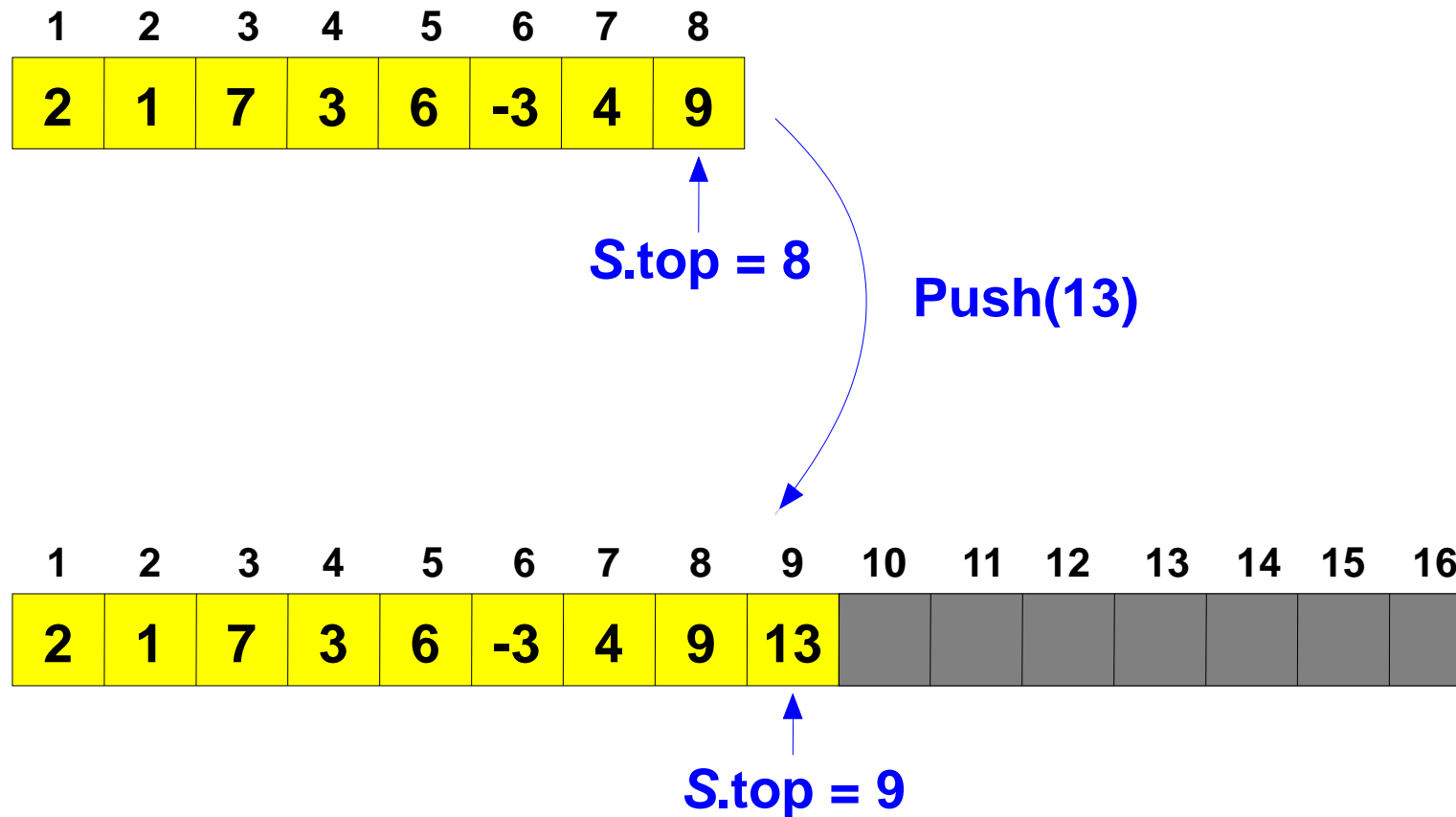
```
1   $S.top = S.top + 1$ 
2   $S[S.top] = x$ 
```

POP(S)

```
1  if STACK-EMPTY( $S$ )
2      error "underflow"
3  else  $S.top = S.top - 1$ 
4      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

1

2

4

8

16

32

Halver arrayet når det er $< 1/4$ fyldt

32

16

16

8

8

16

Tid for n udvidelser:
 $1+2+4+\dots+n/2+n = O(n)$

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

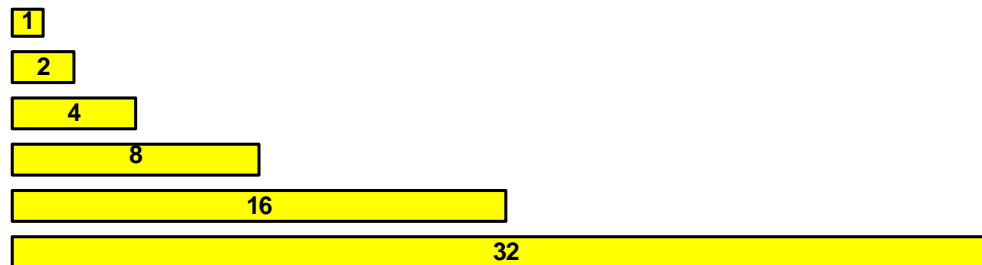
- Analysere **worst-case** tiden for en **sekvens** af operationer
- Behøver kun at analysere den **enkelte operation**

Fordel

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

Intuition

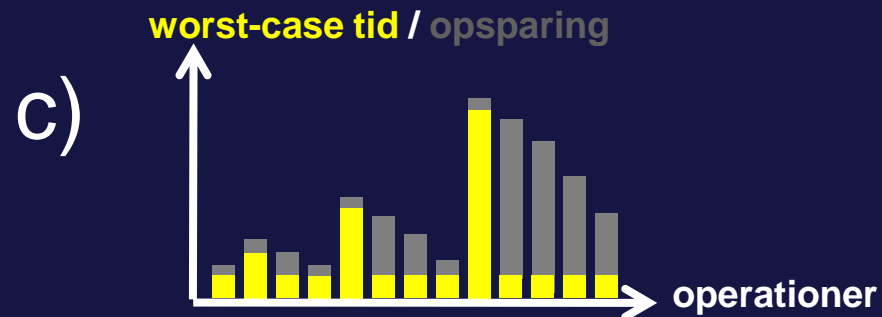
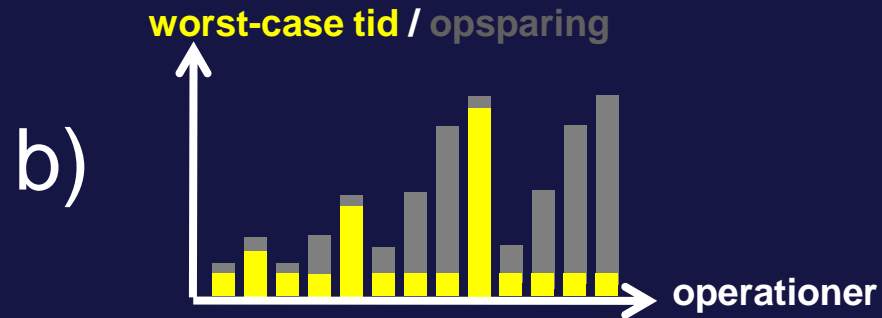
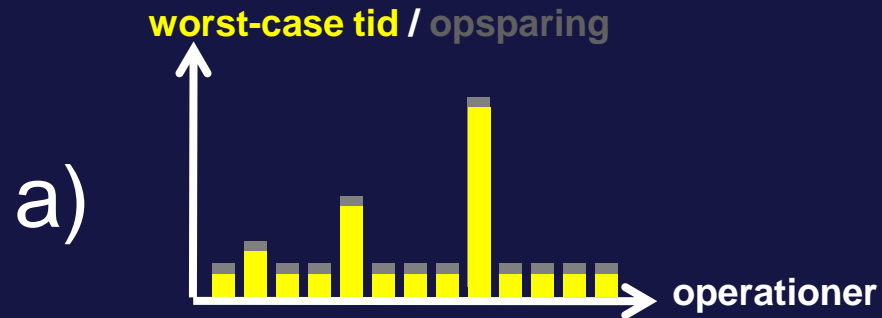
- Der findes ”**gode**”/”**balancerede**” tilstande og ”**dårlige**”/”**ubalancerede**”
- At komme fra en ”**dårlig**” tilstand til en ”**god**” tilstand er **dyrt**
- Det tager mange operationer fra en ”**god**” tilstand før man er i en ”**dårlig**”
- For de (mange) **billige** operationer ”betaler” 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**

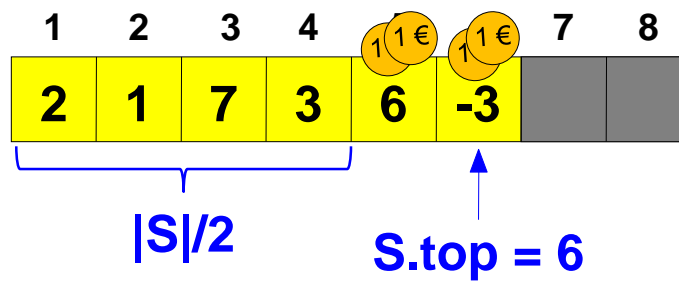
Sammenhæng Mellem Worst-case Tid og Opsparingen Φ



d) Ved ikke

Eksempel: Stak

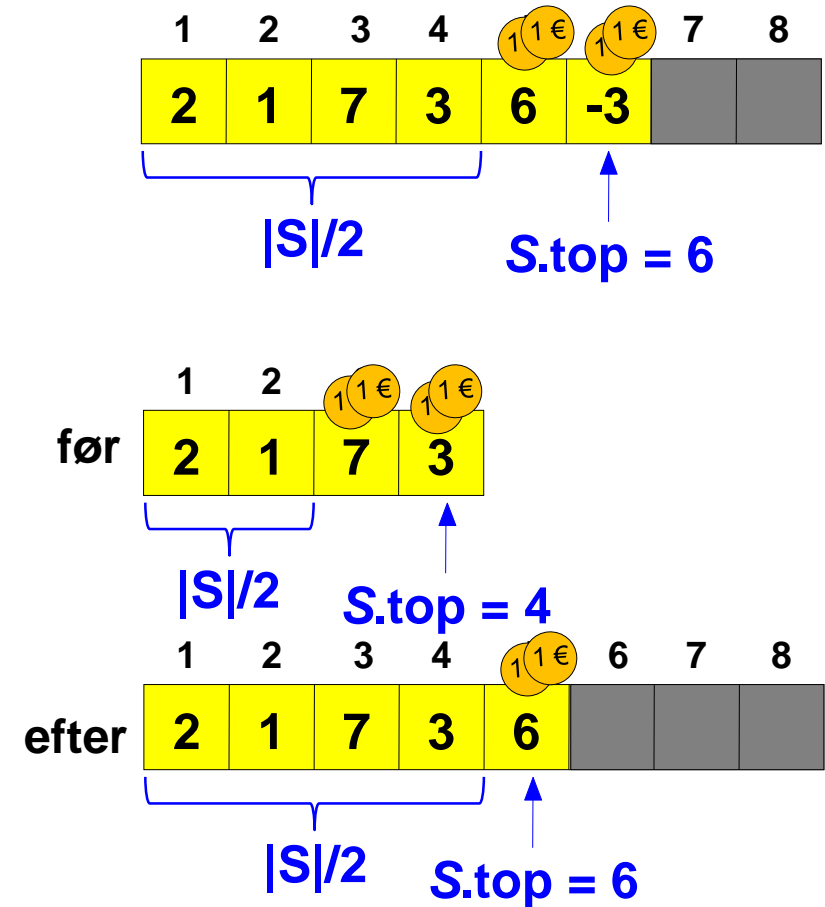
- En **god** stak er halv fuld – kræver ingen opsparing
- Invariant : $\Phi = 2 \cdot |S.top - |S|/2|$
- Antag: 1 € per element indsættelse/kopiering
- Amortiseret tid per push: 3 € ?
(har vi altid penge til at udføre operationen?)
- Hvis ja: n push operationer koster $\leq 3n$ €



Eksempel: Stak

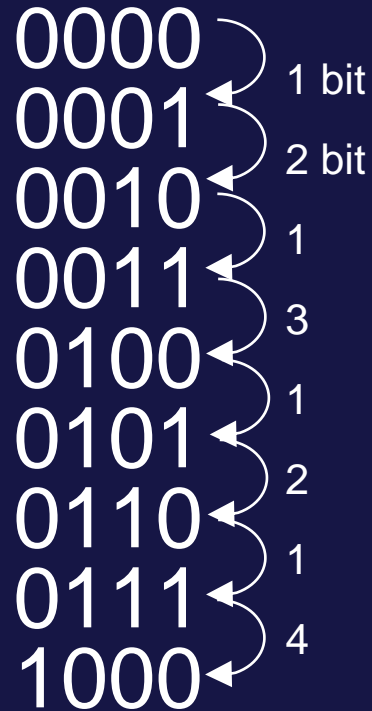
Push = Amortiseret 3€

- 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: $1+2 = \mathbf{3 €}$
- Push med kopiering
 - Kopier S : $|S| €$
 - Indsæt nye element: **1 €**
 - Φ før = $|S|$, Φ efter = 2, dvs $|S|-2 €$ frigives
 - Amortiseret tid: $|S|+1-(|S|-2) = \mathbf{3 €}$



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


Binær Tæller




...
10101110111111
10101111000000

7

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

- Positionen af mest betydende 1-tal
- Positionen af det højreste 0
- Antal 1'er i det binære tal 
- Antal 0'er i det binære tal
- Ved ikke

Amortiseret Analyse

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

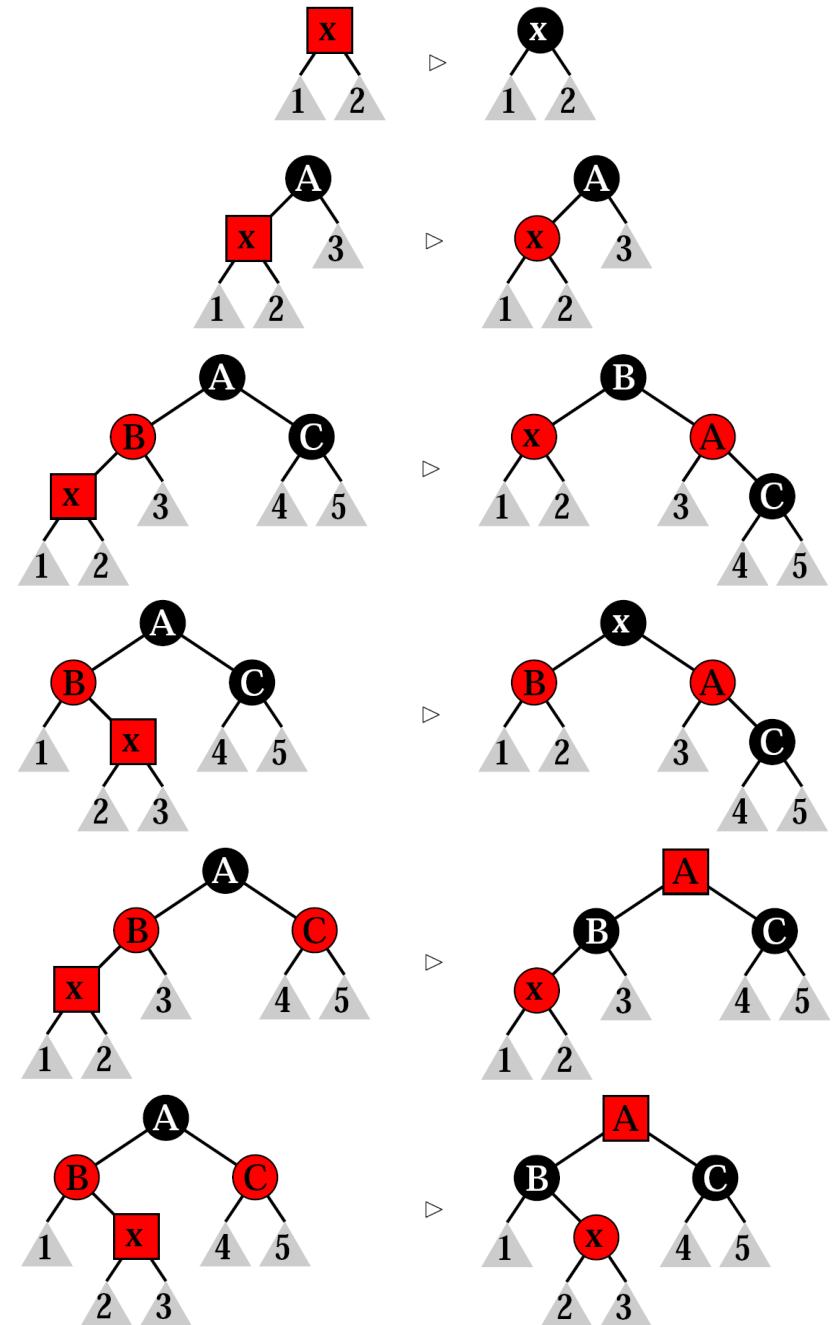
Eksempel: Rød-Sorte Træer

Insert(x)
 =
 Søgning $\leftarrow O(\log n)$
 +
 Opret nyt **rødt** blad $\leftarrow O(1)$
 +
 Rebalancering $\leftarrow \# \text{ transitioner}$

transitioner = *amortiseret* $O(1)$

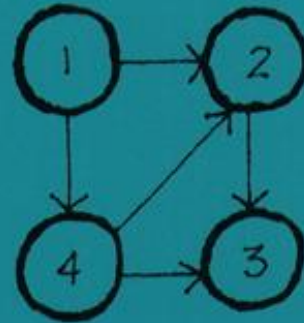
$\Phi = \# \text{ røde knuder}$

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

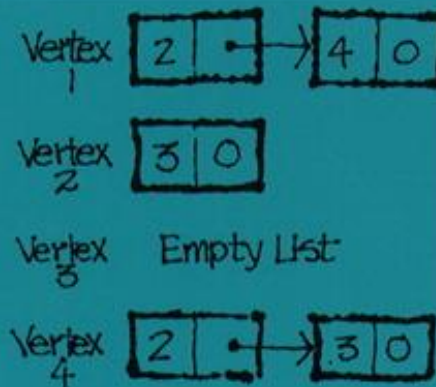


The Design and Analysis of Computer Algorithms

AHO | HOPCROFT | ULLMAN



	1	2	3	4
1	0	1	0	1
2	0	0	1	0
3	0	0	0	0
4	0	1	1	0



	Head	Next
Vertices 1		5
Vertices 2		7
Vertices 3		0
Vertices 4		8
Edges 5	2	6
Edges 6	4	0
Edges 7	3	0
Edges 8	2	9
Edges 9	3	0

Union-Find med Sti-Komprimering (uden rank)

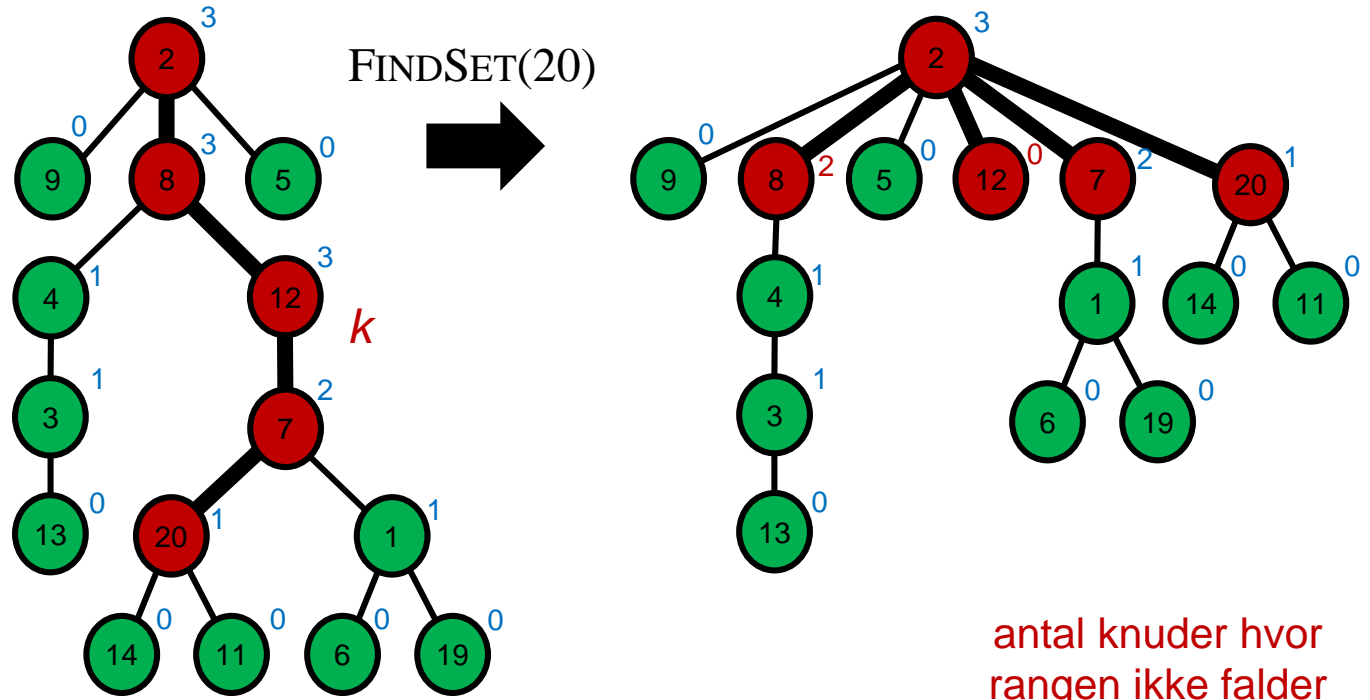
```

MAKESET(x)
1  x.p = x

UNION(x, y)
1  LINK(FINDSET(x), FINDSET(y))

LINK(x, y)
1  y.p = x

FINDSET(x)
1  if x ≠ x.p do
2     x.p = FINDSET(x.p)
3  return x.p
    
```



antal knuder hvor rangen ikke falder

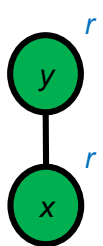
	Faktisk arbejde	$\Delta\Phi$	Amortiseret arbejde (faktisk arbejde + $\Delta\Phi$)
MAKESET	$O(1)$	0	$O(1)$
LINK	$O(1)$	$\leq \log n$	$O(\log n)$
FINDSET	k	$\leq -k + 2 + \log n$	$O(\log n)$
UNION = 2 x FINDSET + LINK			$O(\log n)$

Amortiseret analyse

Definition $rank(x) = \lfloor \log_2 |T(x)| \rfloor$

Lemma $rank(x.p) \geq rank(x)$

Potentiale $\Phi = \sum_x rank(x)$



Observation

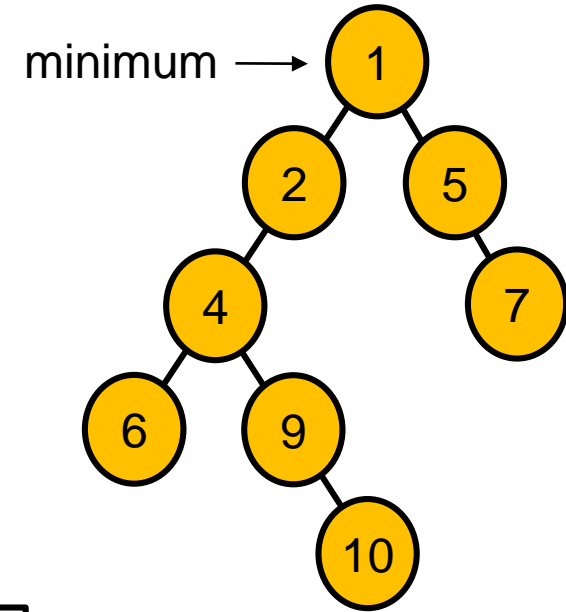
Antag x og y har rang r, dvs. begge undertræer har størrelse $[2^r, 2^{r+1}[$

Hvis x fjernes, så får y rang $< r$

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

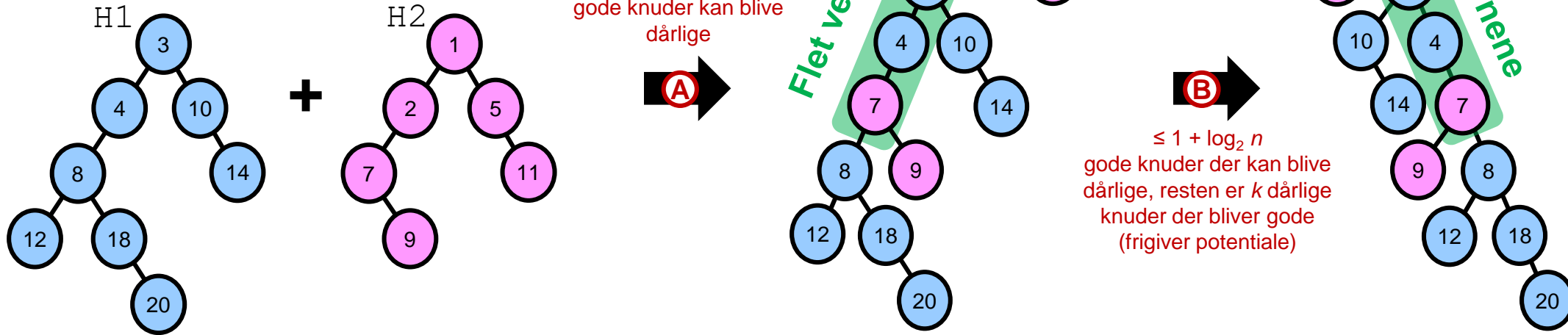
Proc Insert(H, e)
    v = new Node(e, Null, Null)
    return Meld(H, v)

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

Skew heaps

Ikke pensum

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 $v : |v.left| \leq |v| / 2$
- Lemma** # gode knuder på venstre stien $\leq 1 + \log_2 n$
- Potentiale** $\Phi = \#$ dårlige knuder

Meld amortiseret omkostning:

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

\Rightarrow Alle operationer **amortiseret time** $O(\log n)$

Selvbalancerende Datastrukturer

med amortiserede udførelstider

	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

$O_{AM} \equiv$ amortiseret tid

Fibonacci heaps

Fredman og Tarjan, [JACM](#) 1987 [CLRS, kapitel 19]

Splay trees

Sleator og Tarjan, [JACM](#) 1985 + Cole*, [SICOMP](#) 2006