

Grundlæggende Algoritmer og Datastrukturer

Amortiseret Analyse [CLRS, kapitel 17]

OVERVIEW PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

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
```

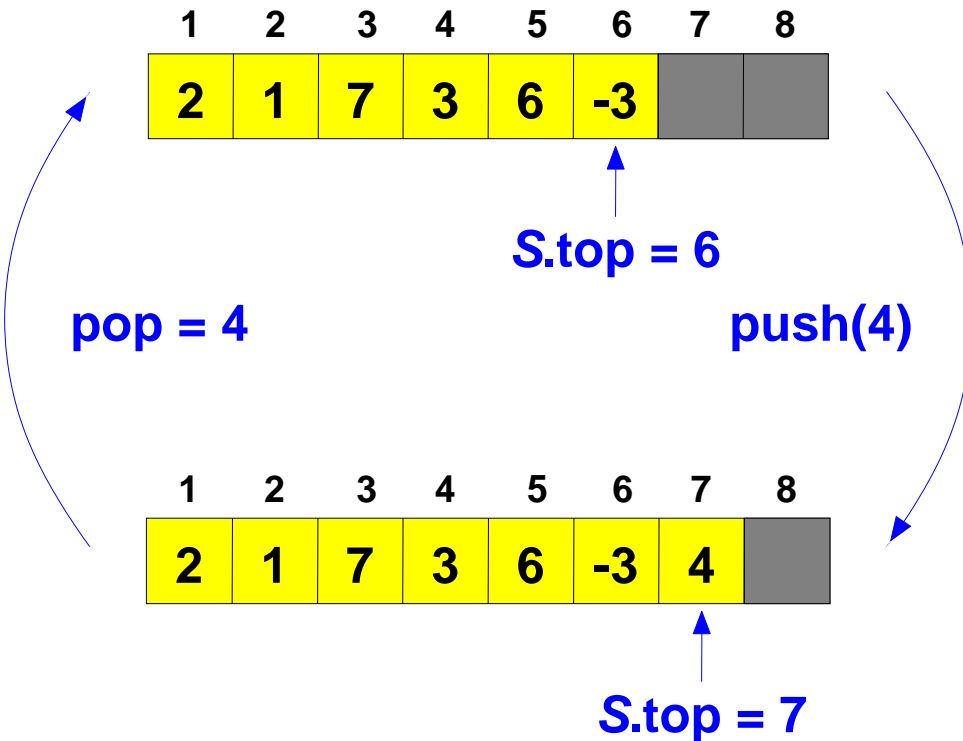
Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class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

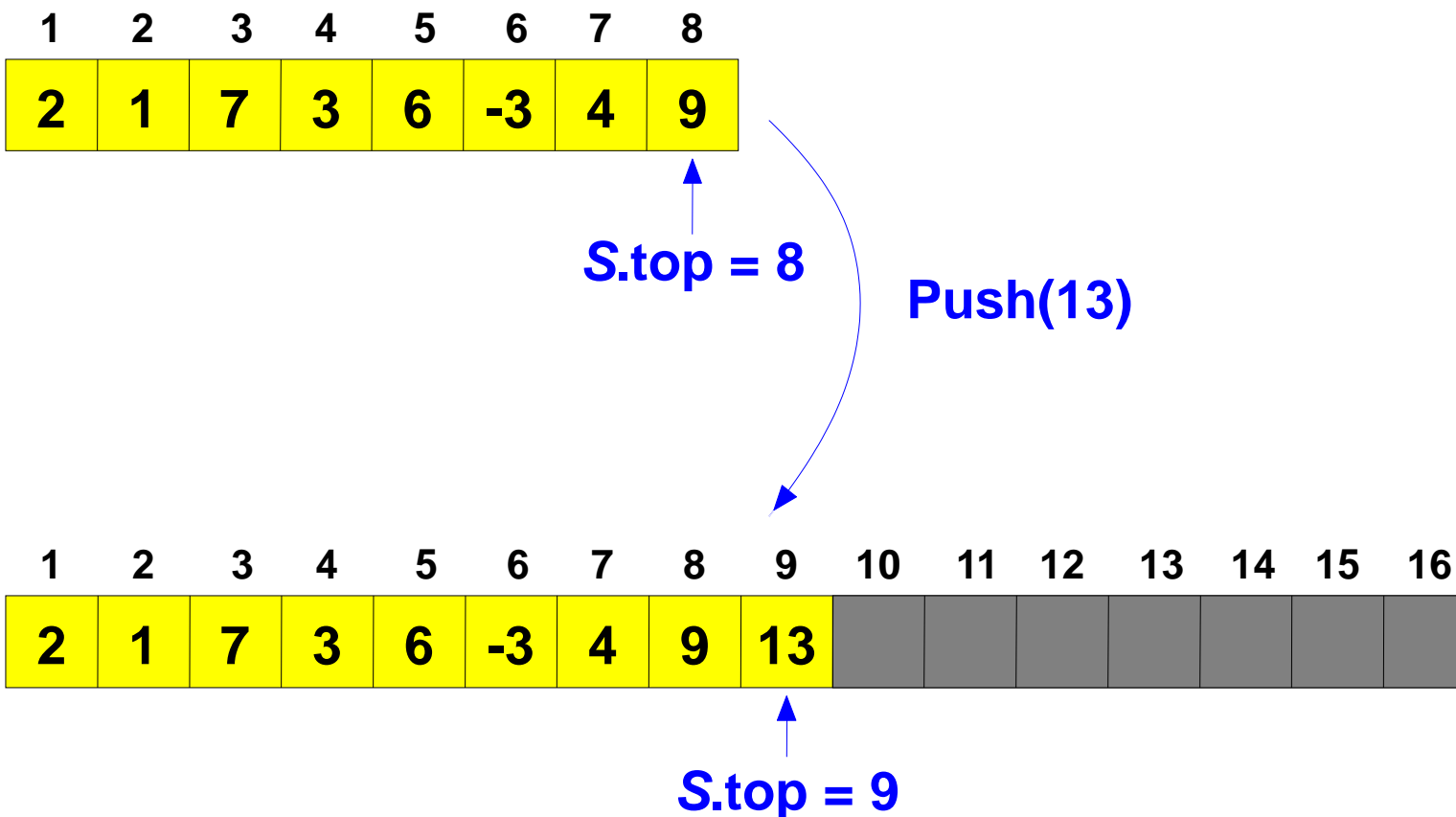
```
1   $S.top = S.top + 1$ 
2   $S[S.top] = x$ 
```

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

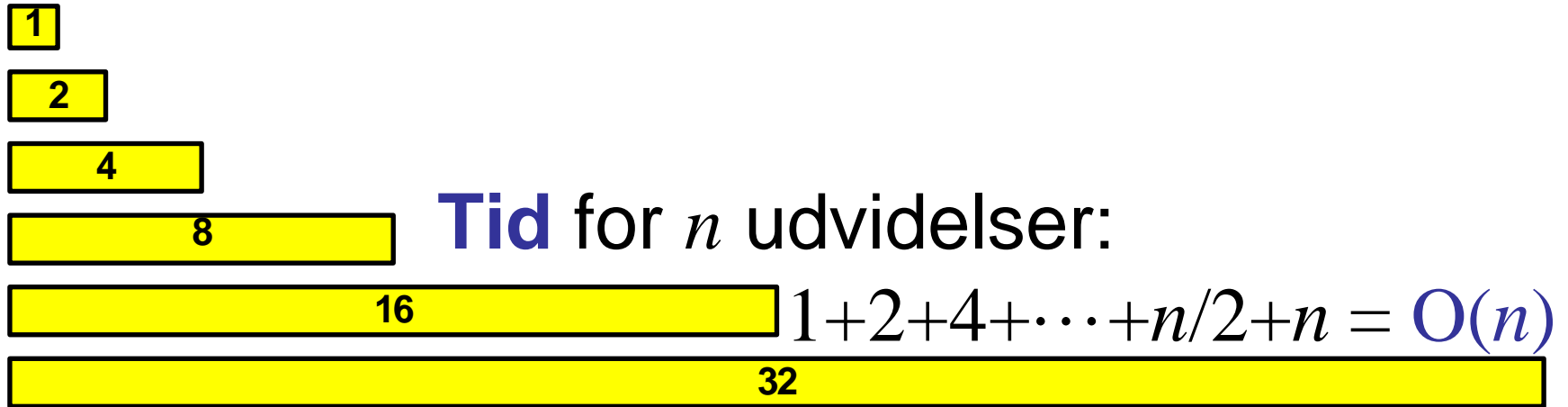
Stak : Overløb



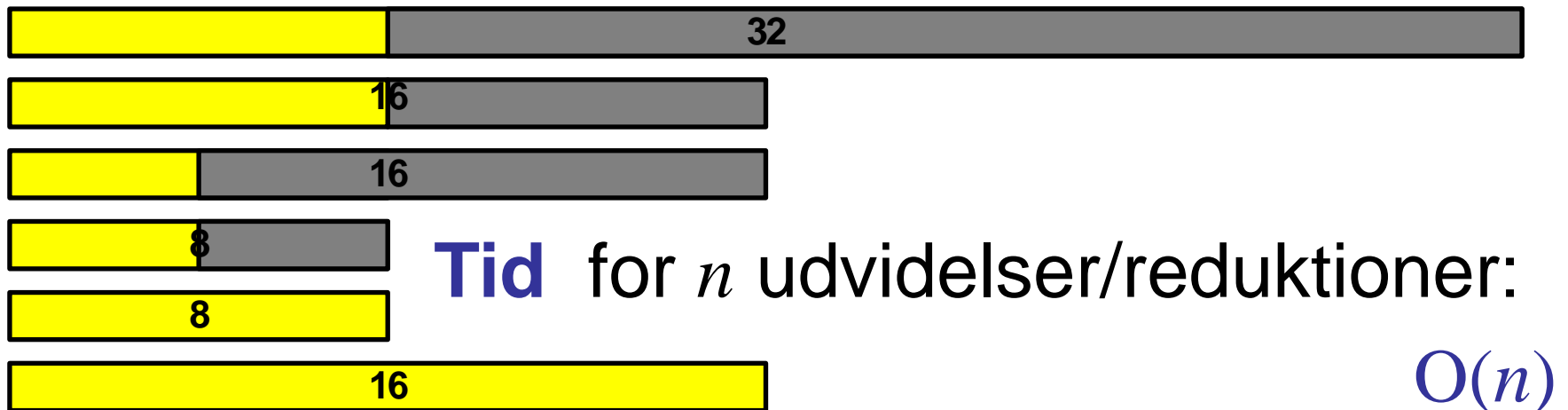
Array fordobling : $O(n)$ tid

Array Fordobling

Fordoble arrayet når det er fuld



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



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

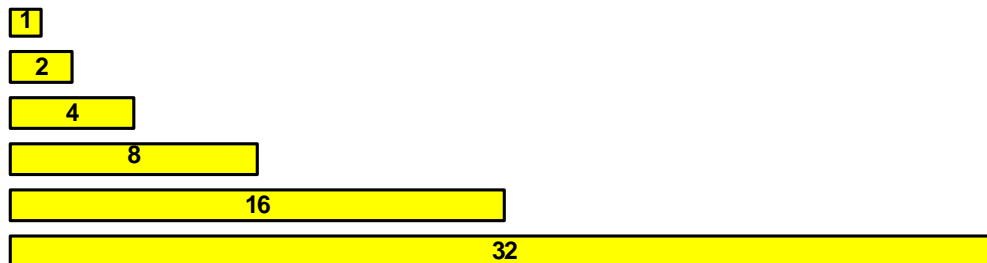
- Analysere **worst-case** tiden for en **sekvens** af operationer
- Behøver kun at analysere den **enkelte operation**

Fordel

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

Intuition

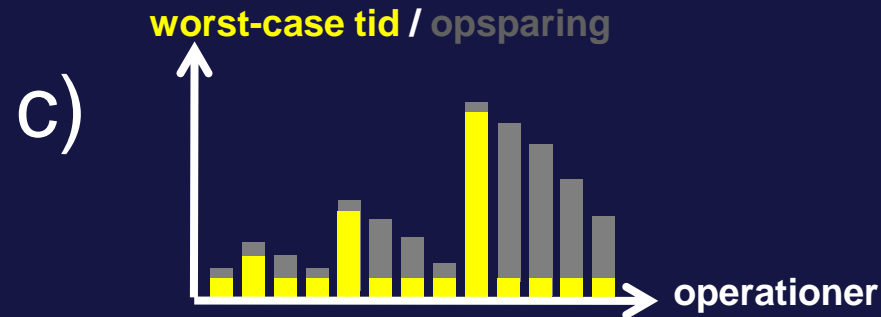
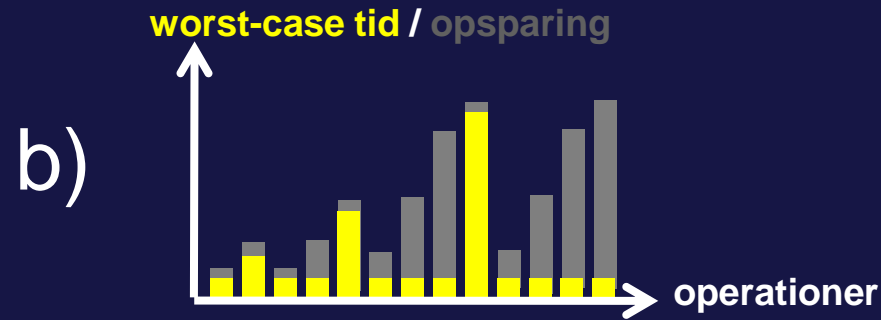
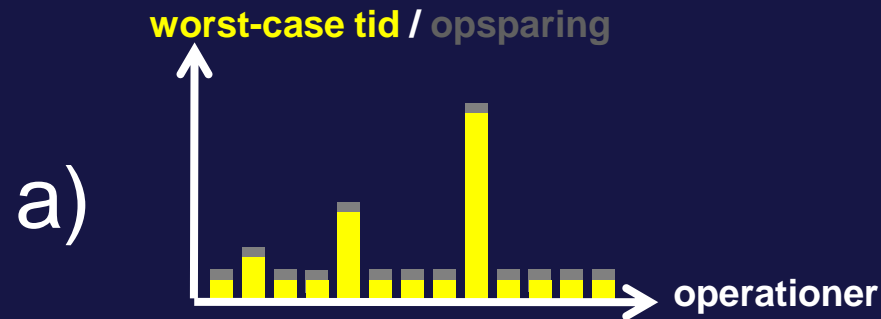
- Der findes ”**gode**”/”**balancerede**” tilstande og ”**dårlige**”/”**ubalancerede**”
- At komme fra en ”**dårlig**” tilstand til en ”**god**” tilstand er **dyrt**
- Det tager mange operationer fra en ”**god**” tilstand før man er i en ”**dårlig**”
- For de (mange) **billige** operationer ”betaler”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

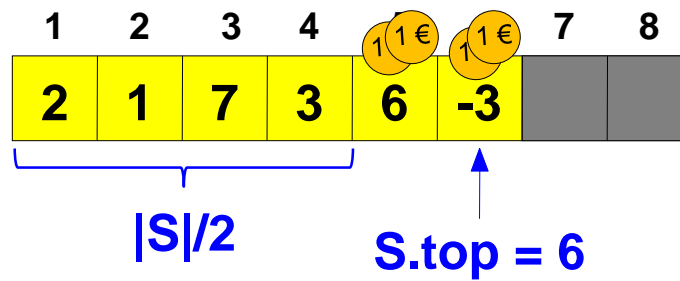
Sammenhæng Mellem Worst-case Tid og Opsparingen Φ



d) Ved ikke

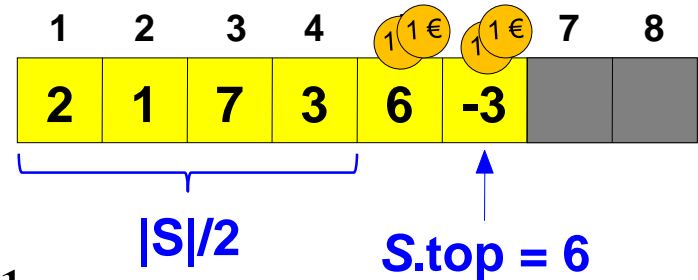
Eksempel: Stak

- 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

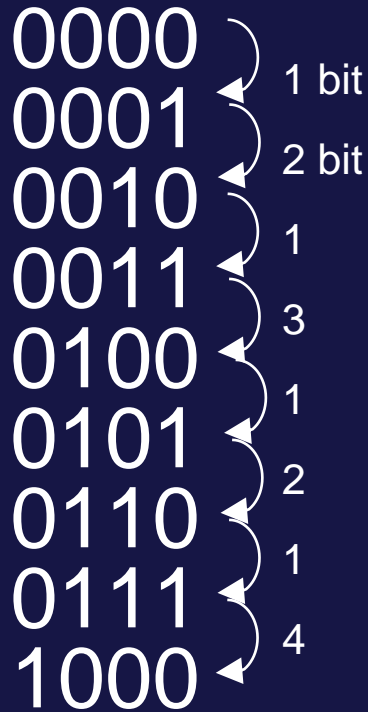
Push = Amortiseret 3€



- Push uden kopiering:
 - Et nyt element : **1 €**
 - $||S|/2 - top[S]|$ 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: $\Phi = 2 \cdot |S.top - |S|/2|$

Binær Tæller




...

10101110111111
10101111000000 ← 7

**Hvad er en god opsparring- /
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 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

← $O(\log n)$

+

Opret nyt **rødt** blad ← $O(1)$

+

Rebalancering ← # transitioner

transitioner = amortiseret $O(1)$

$\Phi = \#$ røde knuder

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

