

# ID1020: Stackar och Köer

Dr. Jim Dowling  
[jdowling@kth.se](mailto:jdowling@kth.se)

kap. 1.3

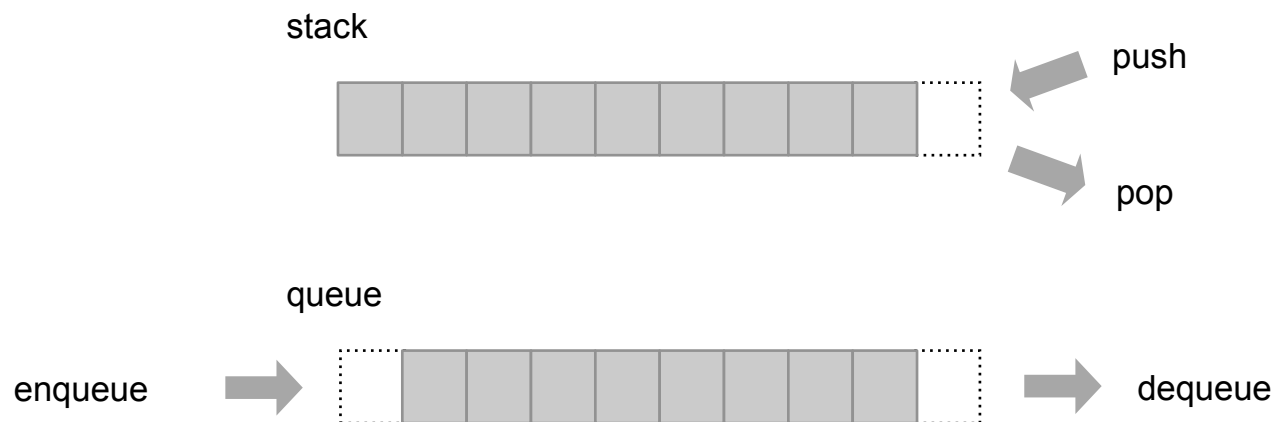


Slides adapted from Algorithms 4<sup>th</sup> Edition, Sedgewick.

# Stackar och köer

- Grundläggande datatyper.

- Kollektion (collection) av objekt.
- Operationer: **insert**, **remove**, **iterate**, testa om null (*if empty*).
- Det är uppenbart hur man lägger till (insert) ett objekt.
- Men, vilket element tar man bort (remove)?



- **Stack.** Examinera elementet *most recently added*

← LIFO = "last in first out"

- **Kö.** Examinera elementet *least recently added*.

← FIFO = "first in first out"

# Klient, implementation, interface

- Separar interface och implementation.

T.ex. stack, queue, bag, priority queue, symbol table, union-find, ....

- Fördelar.

Implementationen kan förbättras utan att påverka klienten

- Klienten får inte veta detaljer om implementationen => klienten får välja från flera olika implementationer.
- Implementationen får inte veta detaljer om klientens behov => många klienter kan återanvända samma implementationen.
- **Design:** skapar modular, återanvändbara bibliotek.
- **Prestanda:** använd en optimerade implementation när den behövs.

Klient:	programmera med operationer definierade i interface:n.
---------	--

Implementation:	kod som implementerar operationerna och datastrukturer.
-----------------	---

Interface:	beskrivningen av grundläggande operationer.
------------	---

bags, stackar ock köer

# Stack API

## Exempel API. Stack av String datatyp.

```
public class StackOfStrings
```

---

```
    StackOfStrings (  
    )
```

*create an empty stack*

```
    void push (String  
              item)
```

*insert a new string onto stack*

```
    String pop ()
```

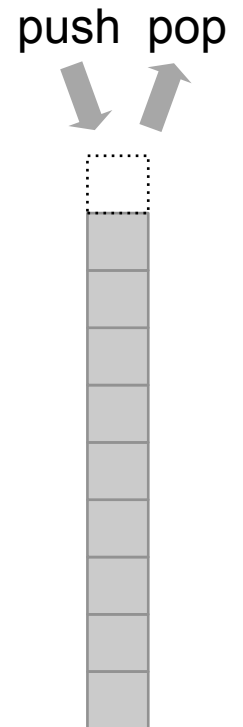
*remove and return the string  
most recently added*

```
    boolean isEmpty ()
```

*is the stack empty?*

```
    int size ()
```

*number of strings on the stack*

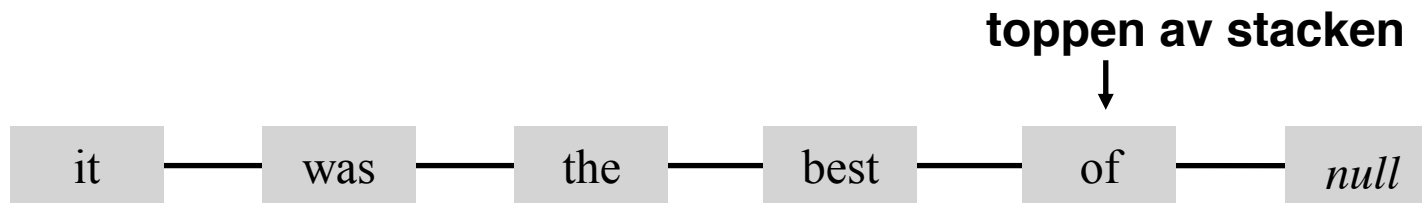


**Exempel klient.** Omvänd sekvens av strängar från standard input.

# Hur implementerar man en stack med en länkad lista?

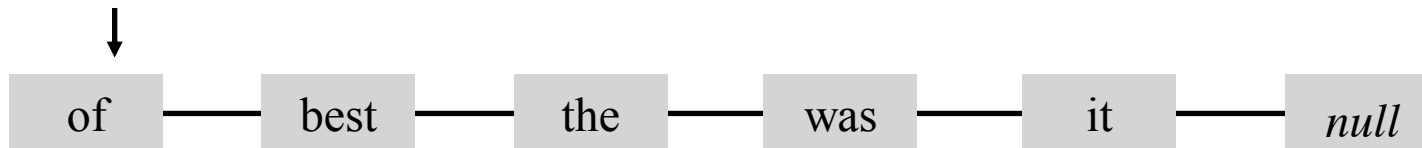
A. Det går inte att implementera med en enkellänkad lista på ett effektivt sätt.

B.



C.

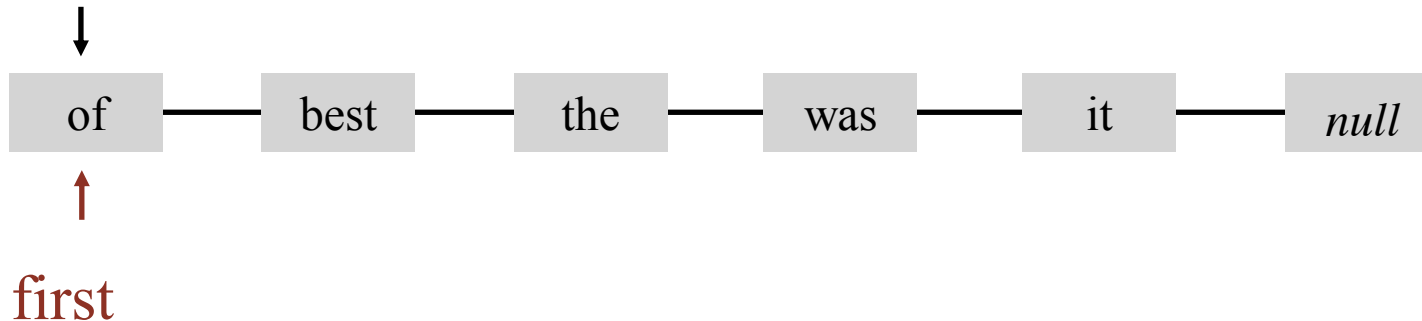
toppen av stacken



# Stack: länkad lista implementation

- Spara pekare `first` till första noden i en enkellänkad lista.
- Push ett nytt element innan `first` och uppdatera `first`.
- Pop ett nytt element från `first`.

toppen av stacken



# Stack pop: länkad lista implementation

## inner class

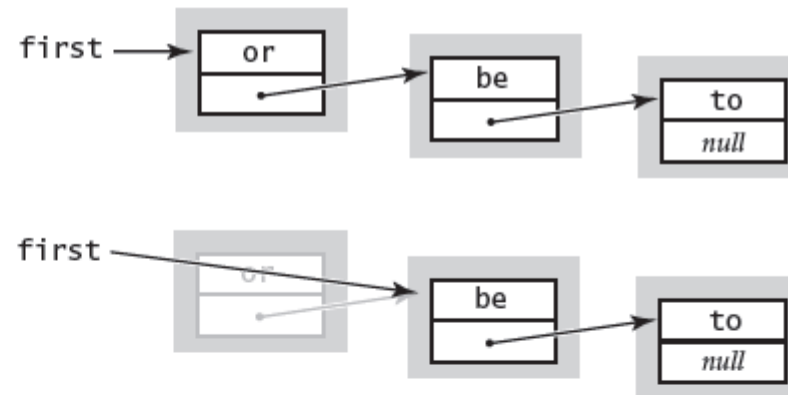
```
private class Node
{
    String item;
    Node next;
}
```

## save item to return

```
String item = first.item;
```

## delete first node

```
first = first.next;
```



## return saved item

```
return item;
```



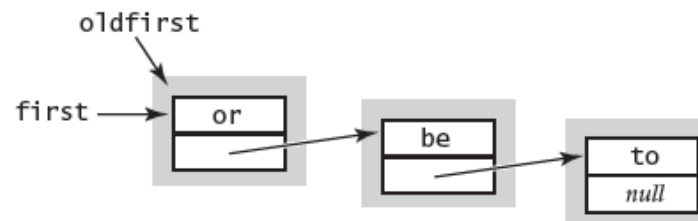
# Stack push: länkad lista implementation

## inner class

```
private class Node
{
    String item;
    Node next;
}
```

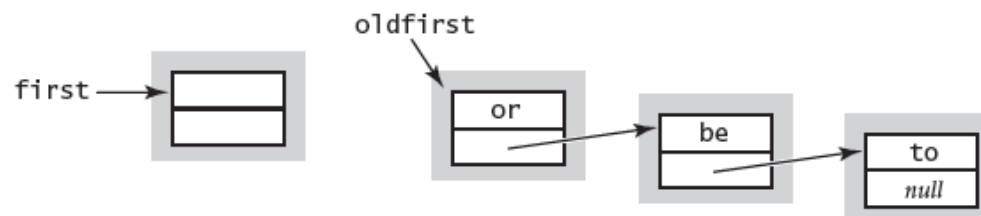
## save a link to the list

```
Node oldfirst = first;
```



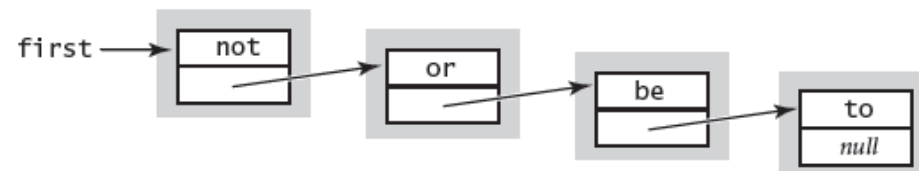
## create a new node for the beginning

```
first = new Node();
```



## set the instance variables in the new node

```
first.item = "not";
first.next = oldfirst;
```



# Stack: länkad lista implementation i Java

```
public class LinkedStackOfStrings {  
    private Node first = null;  
    private class Node  
    {  
        String item;  
        Node next;  
    }  
  
    public boolean isEmpty()  
    { return first == null; }  
  
    public void push(String item) {  
        Node oldfirst = first;  
        first = new Node();  
        first.item = item;  
        first.next = oldfirst;  
    }  
  
    public String pop() {  
        String item = first.item;  
        first = first.next;  
        return item;  
    }  
}
```

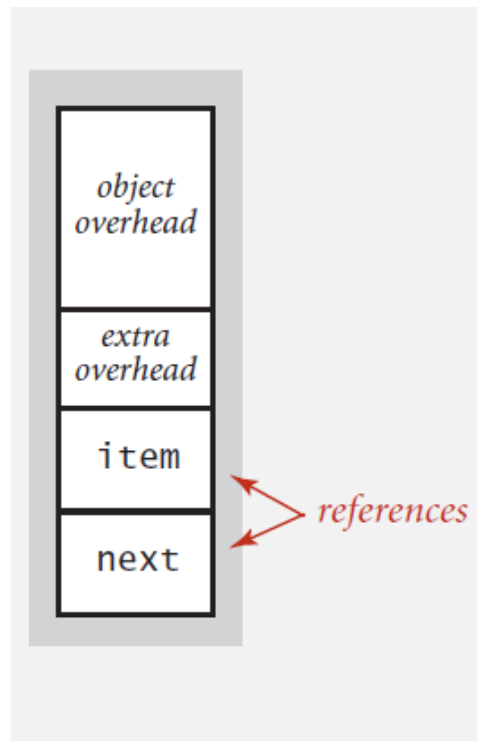
← innre klass (access modifiers spelar  
ingen roll för attributer i en innre klass)

# Stack: länkad lista implementation prestanda

- **Sats.** Varje operation tar konstant tid i worst case.
- **Sats.** En stack med  $N$  element använder  $\sim 40 N$  bytes.

inner class

```
private class Node {  
    String item;  
    Node next;  
}
```



16 bytes (object overhead)

8 bytes (inner class extra overhead)

8 bytes (reference to String)

8 bytes (reference to Node)

40 bytes per stack node

- **Anmärkning.** Beräkningen inkluderar bara minnet av stacken (inte minnet som behövs för strängerna själva, som klienten äger).

# Hur implementerar man en fixed-kapacitet stack med en array?

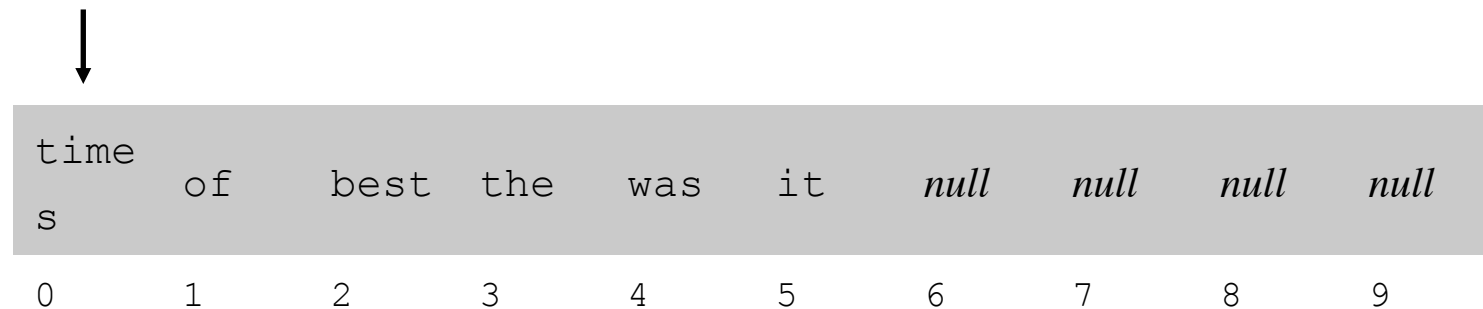
Det går inte att implementera med en array på ett effektivt sätt.

B.



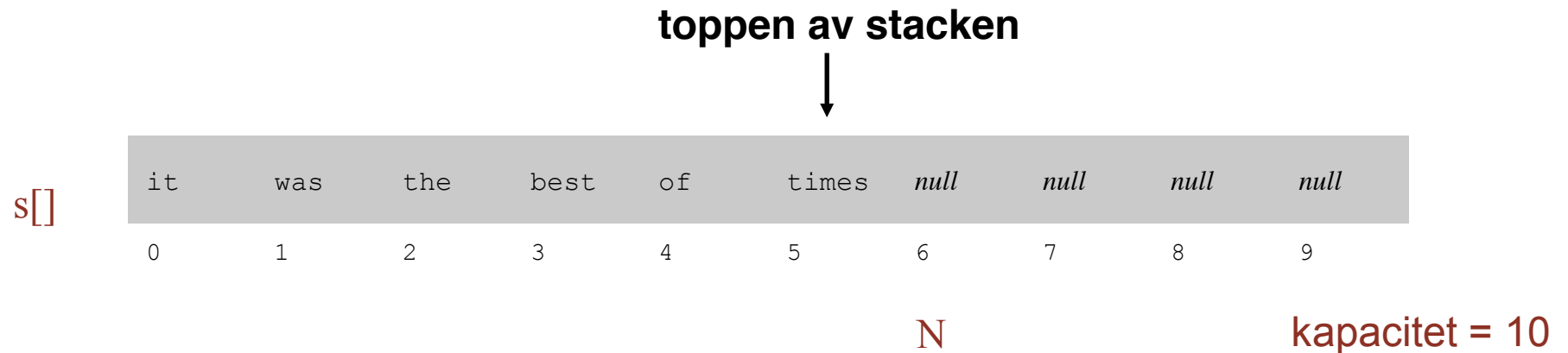
C.

toppen av stacken



# Fixed-kapacitet stack: array implementation

- Använd en array  $s[]$  för att lagra  $N$  poster på stacken.
  - `push()`: lägg till en ny post vid  $s[N]$ .
  - `pop()`: tar bort från  $s[N-1]$ .



**Defekt.** Stack-overflow inträffar när  $N$  är större än kapacitet.

# Fixed-kapacitet stack: array implementation

```
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity)
    {   s = new String[capacity];   }

    public boolean isEmpty()
    {   return N == 0;   }

    public void push(String item) {
        s[N++] = item;
    }

    public String pop()
    {   return s[--N];   }
}
```

används för att indexera  
in i array:en: sedan för att  
inkrementera N

decrement N; sedan används  
för att indexera in i array

# Att tänka på...

- **Overflow och underflow.**
  - Underflow: kasta en exception om klienten anropar *pop* på en tom stack.
  - Overflow: använd en *resizing array* för array implementationen.
- **Null element:** det är tillåtet att lägga till ett null element.
- **Loitering.** *Loitering* (söla) händer när någon objekt behåller en referens till ett annat objekt när referensen inte längre behövs.

```
public String pop() {  
    return s[--N];  
    //Vi glömde nollställa s[N]!  
}
```

*loitering*

```
public String pop() {  
    String item = s[--N];  
    s[N] = null;  
    return item;  
}
```

**Denna version undviker loitering.**

**Garbage collectorn kan bara få tillbaka  
minnet från ett objekt om det inte längre finns  
referenser kvar till objektet.**

# Stackar: resizing arrayer



# Stack: resizing-array implementation

- **Problem.** API:et är trasig om det behöver klienten att specificera kapaciteten!
- Hur ökar och minskar man storleken (dvs. längden) av array:n?

- **Första försöket.**

- `push()`: öka storleken av array `s[]` med 1.
- `pop()`: minska storleken av array `s[]` med 1.

- **För dyrt.**

- Vi behöver kopiera alla element till en ny array för varje *push* operation.

Antal array accesser för att lägga till första  $N$  element =

$$N + (2 + 4 + \dots + 2(N-1)) \sim N^2.$$

↑                      ↑  
1 array access      2(k-1) array accesser för att öka längden med k  
per push            (ignorera kostnaden att skapa den nya array:n)

omöjlig för stort N



- **Utmaning.** Skriva om programmet så att array "resizing" sällan inträffar.

# Stack: resizing-array implementation

- Hur kan man öka storleken av array:n?
- Om array:n är full, skapa en ny array **två gånger större**, och kopiera elementen.

"repeated doubling"

```
public ResizingArrayStackOfStrings() { s = new String[1]; }  
public void push(String item) {  
    if (N == s.length) resize(2 * s.length);  
    s[N++] = item;  
}  
private void resize(int capacity) {  
    String[] copy = new String[capacity];  
    for (int i = 0; i < N; i++) {  
        copy[i] = s[i];  
    }  
    s = copy;  
}
```

- Resultat: att lägga till första  $N$  element tar tid proportionellt mot  $N$  inte  $N^2$

# Stack: resizing-array implementation

- Hur minskar man storlekan av array:n?
- Första försöket.
  - `push()`: fördubbla storlekan av array `s[]` när array är full.
  - `pop()`: halvera storlekan av array `s[]` när array är halv-full.
- För dyrt i worst case.
  - Överväga push-pop-push-pop-... sekvensen när array:n är full.
  - Varje operation tar tid proportionell mot  $N$ .

N = 5	to	be	or	not	to	<i>null</i>	<i>null</i>	<i>null</i>
N = 4	to	be	or	not				
N = 5	to	be	or	not	to	<i>null</i>	<i>null</i>	<i>null</i>
N = 4	to	be	or	not				

# Stack: resizing-array implementation

- Hur minskar man storlekan av array:n?
- Effektiv lösning.
  - `push()`: fördubbla storlekan av array `s[]` när array är full.
  - `pop()`: halvera storlekan av array `s[]` när array är **en kvart-full**.

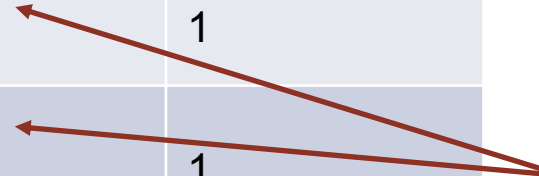
```
public String pop() {  
    String item = s[--N];  
    s[N] = null;  
    if (N > 0 && N == s.length/4) {  
        resize(s.length/2);  
    }  
    return item;  
}
```

- Invariant. array `s[]` är nu mellan 25% och 100% full.

# Stack resizing-array implementation: prestanda

- **Amorterad analys.** Börja från en tom datastruktur, medel körtiden per operation över en worst-case sekvens av operationer.
- **Sats.** Om man börjar med en tom stack, alla möjliga sekvenser av  $M$  push och pop operationer tar tid proportionell mot  $M$ .

	best	worst	amortized
<b>construct</b>	1	1	1
<b>push</b>	1	$N$	1
<b>pop</b>	1	$N$	1
<b>size</b>	1	1	1



fördubbling och halvering operationer

Ökningen av körtiden när man kör resize på en stack med  $N$  element

# Stack resizing-array implementation: minneskomplexitet

- **Sats.** Använder mellan  $\sim 8 N$  och  $\sim 32 N$  bytes för att representera en stack med  $N$  element.
  - $\sim 8 N$  när full.
  - $\sim 32 N$  när en kvart full.

```
public class ResizingArrayStackOfStrings
{
    private String[] s;
    private int N = 0;
    ...
}
```

8 bytes (refers till array)

24 bytes (array overhead)

← 8 bytes × array storlek

← 4 bytes (int)

← 4 bytes (padding)

- **Anmärkning.** Beräkningen inkluderar bara minnet av stacken (inte minnet som behövs för strängerna själv, som klienten äger).

# Stack implementations: resizing array vs. länkad lista

- **Tradeoffs.** Man kan implementera en stack med antingen en resizing array eller en länkad lista; klient kan byta ut den ena mot den andra.

- Vilken är bättre?

In computer science, amortized analysis is a method for analyzing a given algorithm's time complexity, or how much of a resource, especially time or memory, it takes to execute. The motivation for amortized analysis is that looking at the worst-case run time per operation can be too pessimistic

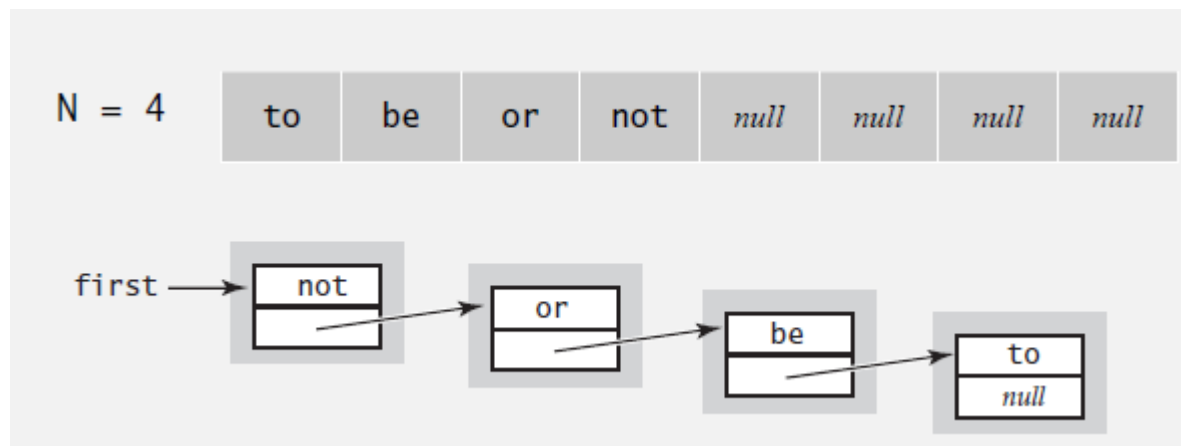
- **Länka lista implementation.**

- Varje operation tar konstant tid i **worst case**.
- Använder extra tid och minne för att hantera länkerna.

- **Resizing-array implementation.**

- Varje operation tar konstant **amorterad** tid.
- Mindre slösseri med minne.

Amortized time looks at an algorithm from the viewpoint of total running time rather than individual operations. We don't care how long one insert takes, but rather the average time of all the calls to insert.



# Resizing arrayer: köer



# Queue (kö) API

```
public class QueueOfStrings
```

---

```
    QueueOfStrings()
```

*create an empty queue*

```
    void enqueue(String item)
```

*insert a new string onto queue*

```
    String dequeue()
```

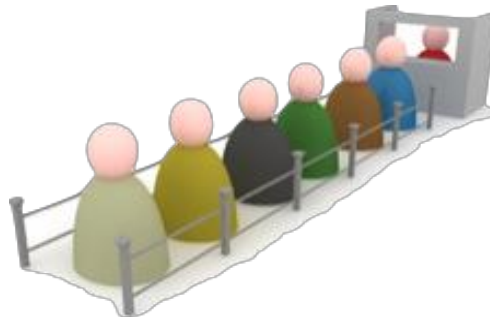
*remove and return the string  
least recently added*

```
    boolean isEmpty()
```

*is the queue empty?*

```
    int size()
```

*number of strings on the queue*



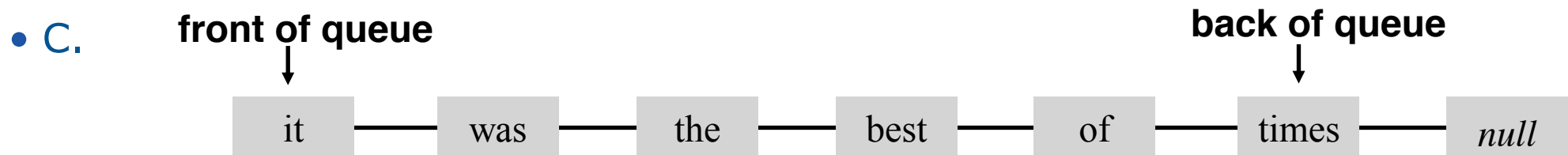
enqueue



dequeue

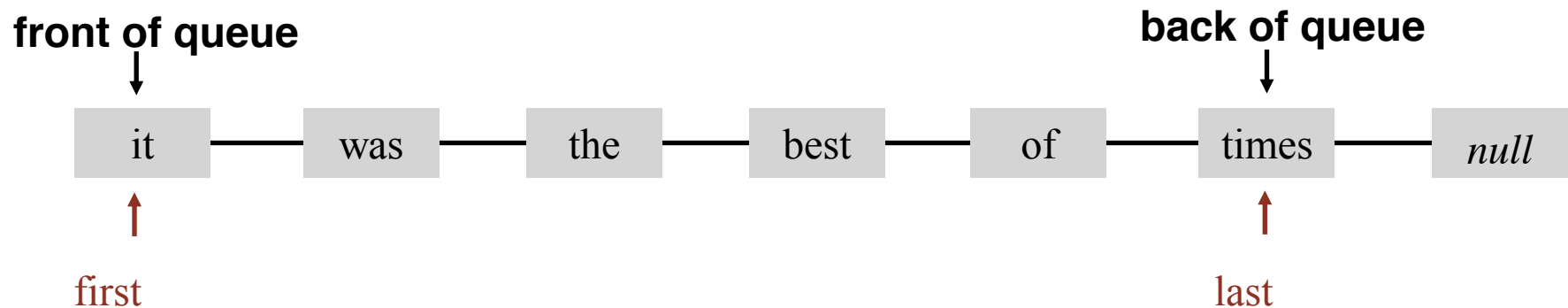
# Hur implementeras en kö med en länkad lista?

- A. Det går inte att implementera med en enkellänkad lista på ett effektivt sätt.



# Kö: länkad lista implementation

- Spara en pekare `first` till första noden i en enkellänkad lista.
- Spara en till pekare `last` till sista noden.
- Dequeue (*ta bort*) från `first`.
- Enqueue (*lägg till*) efter `last`.



# Kö dequeue: länkad lista implementation

**save item to return**

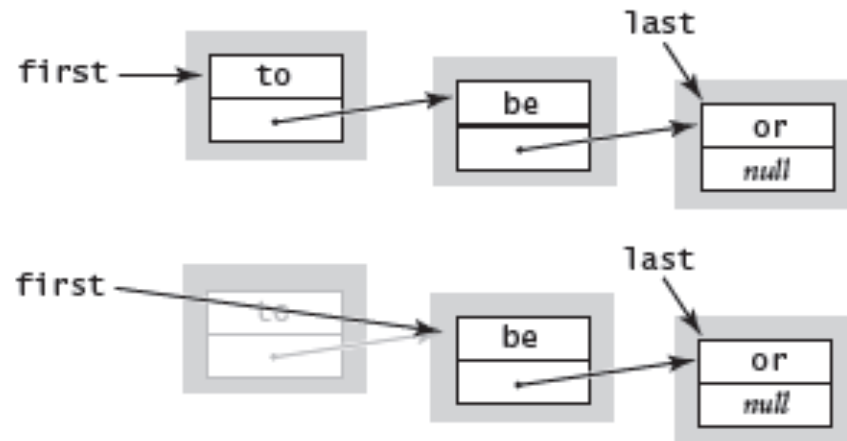
```
String item = first.item;
```

**delete first node**

```
first = first.next;
```

innre klass

```
private class Node {  
    String item;  
    Node next;  
}
```



**return saved item**

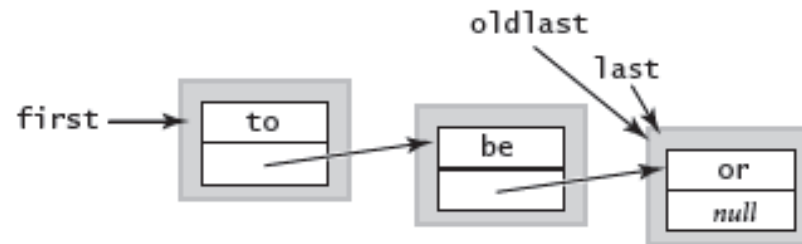
```
return item;
```

- **Anmärkning.** Samma kod som länkad lista stack `pop()`.

# Kö enqueue: länkad lista implementation

save a link to the last node

```
Node oldlast = last;
```

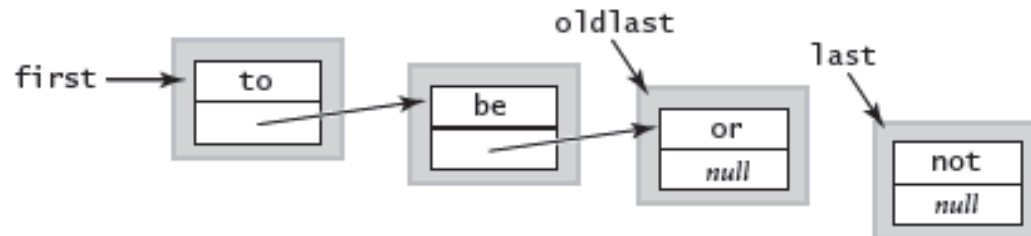


inre klass

```
private class Node {  
    String item;  
    Node next;  
}
```

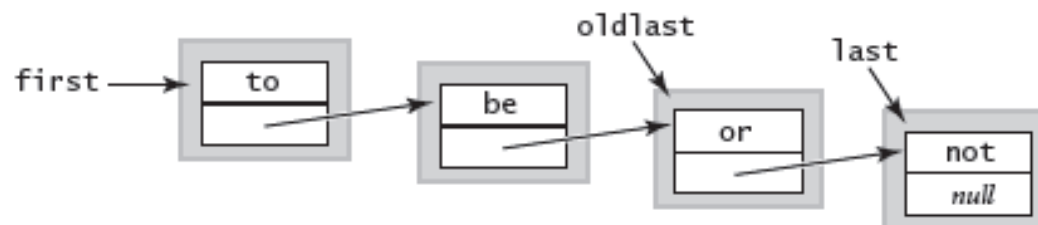
create a new node for the end

```
last = new Node();  
last.item = "not";
```



link the new node to the end of the list

```
oldlast.next = last;
```



# Kö: länkad lista implementation in Java

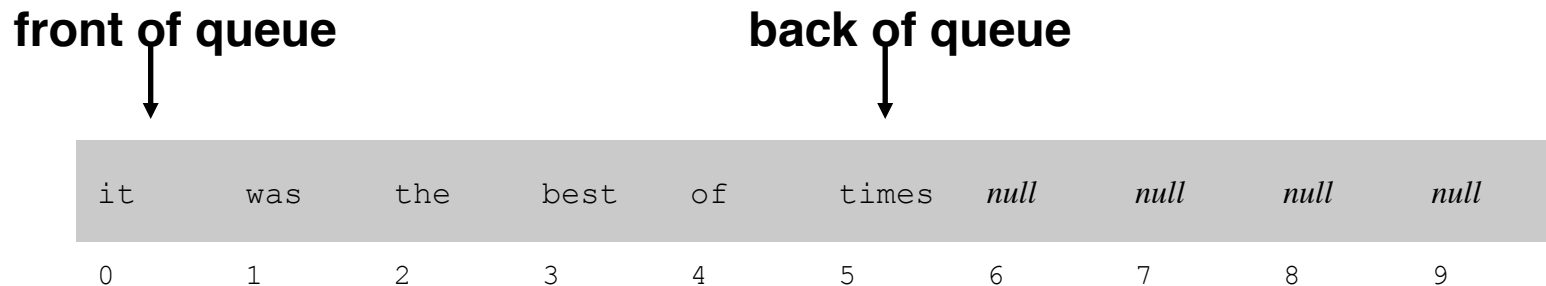
```
public class LinkedQueueOfStrings {  
    private Node first, last;  
    private class Node { /* same as in LinkedStackOfStrings */ }  
    public boolean isEmpty() { return first == null; }  
  
    public void enqueue(String item) {  
        Node oldlast = last;  
        last = new Node();  
        last.item = item;  
        last.next = null;  
        if (isEmpty()) {  
            first = last;  
        } else {  
            oldlast.next = last;  
        }  
    }  
  
    public String dequeue() {  
        String item = first.item;  
        first = first.next;  
        if (isEmpty()) { last = null; }  
        return item;  
    }  
}
```

specialla fall för tomma köer

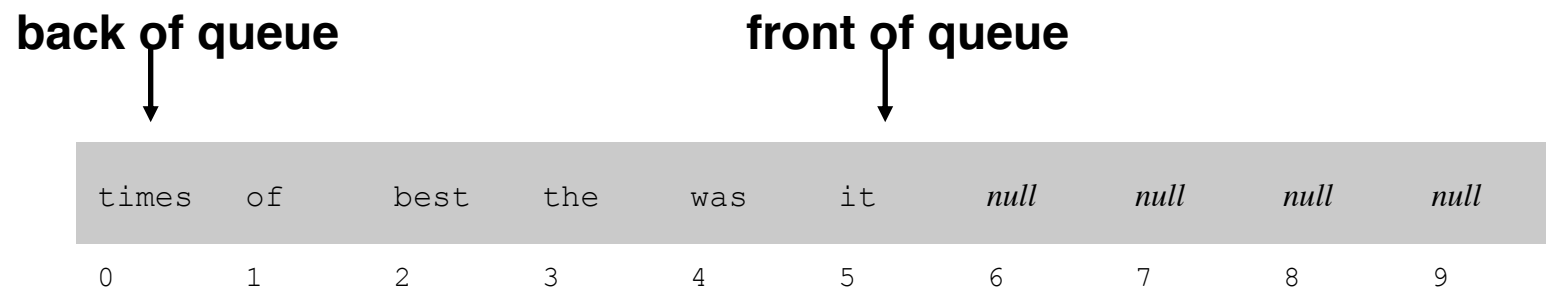
# Hur implementerar man en fixed-kapacitet kö med en länkad lista?

- A. Det går inte att implementera med en enkellänkad lista på ett effektivt sätt.

• B.

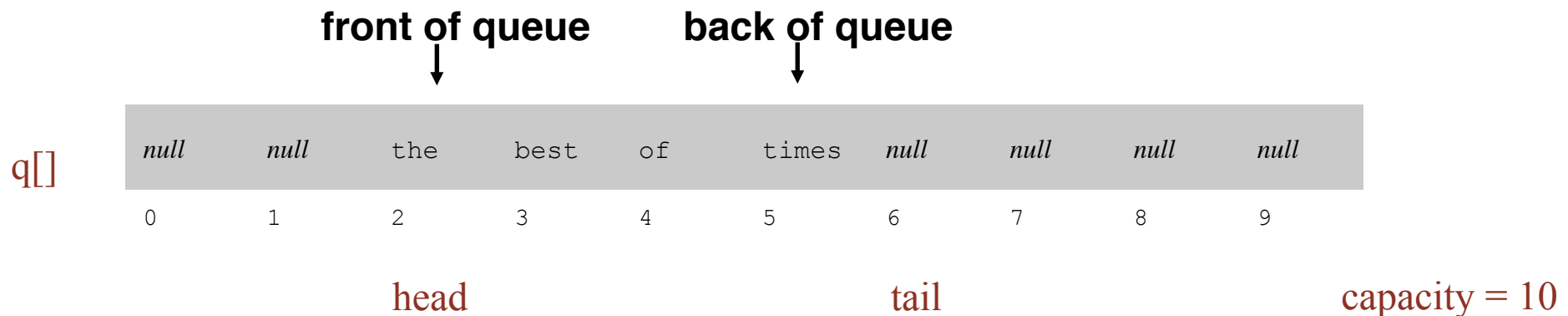


• C.



# Kö: resizing-array implementation

- Använd array  $q[]$  för att lagra element i kön.  
    `enqueue()`: lägg till ett nytt element vid  $q[\text{tail}]$ .  
    `dequeue()`: ta bort ett element från  $q[\text{head}]$ .
- Uppdatera `head` och `tail` modulo kapacitetet.
- Lägg till en "resizing array".



- Hur implementerar man *resize* operationen?



generics

# Parameterized stack

- Vi har implementerad: `StackOfStrings`.
- Vi vill också ha: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ....
- **Försök 1.** Implementera en separat stack klass för varje typ.
  - Att skriva om koden är jobbig och kan lätt introducera buggar.
  - Att underhålla kopiera-och-klistra kod är jobbig och kan lätt introducera buggar.




# Parameterized stack

- Vi har implementerad: `StackOfStrings`.
- Vi vill också ha: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ....
- **Försök 2.** Implementera en stack med element av typ `Object`.
  - Klienten behöver kasta från `Object` typen till mål typen.
  - Att kasta kan lätt introducera buggar: körtidsfel (*runtime error*) om typerna inte matcha varandra.

```
StackOfObjects s = new StackOfObjects();  
Apple a = new Apple();  
Orange b = new Orange();  
s.push(a);  
s.push(b);  
a = (Apple) (s.pop());
```

ClassCastException



# Parameterized stack

- Vi har implementerad: `StackOfStrings`.
- Vi vill också ha: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ....
- **Attempt 3.** Java generics.
  - Unvik att kasta i klienten.
  - Upptäck "type mismatch" vid kompileringstiden istället för körtiden.

```
Stack<Apple> s = new Stack<Apple>();  
Apple a = new Apple();  
Orange b = new Orange();  
s.push(a);  
s.push(b);  
a = s.pop();
```

typ parameter

kompileringsfel

- **Fördel.** Kompileringsfel är bättre än körtidsfel.

# Generic stack: länkad lista implementation

```
public class LinkedStackOfStrings {
    private Node first = null;
    private class Node {
        String item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(String item) {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public String pop() {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

```
public class Stack<Item> {
    private Node first = null;

    private class Node {
        Item item;
        Node next;
    }

    public boolean isEmpty()
    { return first == null; }

    public void push(Item item) {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public Item pop() {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

generic -typ

# Generic stack: array implementation

@#\$\*! generic arrayer är inte tillåtet i Java



```
public class FixedCapacityStackOfStrings{
    private String[] s;
    private int N = 0;

    public StackOfStrings(int capacity)
    {   s = new String[capacity];   }

    public boolean isEmpty()
    {   return N == 0;   }

    public void push(String item)
    {   s[N++] = item;   }

    public String pop()
    {   return s[--N];   }
}
```

```
public class FixedCapacityStack<Item>{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(
        int capacity)
    {   s = new Item[capacity];   }

    public boolean isEmpty()
    {   return N == 0;   }

    public void push(Item item)
    {   s[N++] = item;   }

    public Item pop()
    {   return s[--N];   }
}
```

# Generic stack: array implementation

Ugh. Vi kaster här.

```
public class FixedCapacityStackOfStrings{
    private String[] s;
    private int N = 0;

    public StackOfStrings(int capacity)
    { s = new String[capacity]; }

    public boolean isEmpty()
    { return N == 0; }

    public void push(String item)
    { s[N++] = item; }

    public String pop()
    { return s[--N]; }
}
```

```
public class FixedCapacityStack<Item> {
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(
int capacity) {
    s = (Item[]) new Object[capacity]; }

    public boolean isEmpty()
    { return N == 0; }

    public void push(Item item)
    { s[N++] = item; }

    public Item pop()
    { return s[--N]; }
}
```

# Unchecked Cast

```
% javac FixedCapacityStack.java
```

Note: FixedCapacityStack.java uses unchecked or unsafe operations.

Note: Recompile with -Xlint:unchecked for details.

```
% javac -Xlint:unchecked FixedCapacityStack.java
```

```
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
```

```
found   : java.lang.Object[]
```

```
required: Item[]
```

```
    a = (Item[]) new Object[capacity];
```

```
        ^
```

```
1 warning
```



- Varför behöver jag kasta i Java (eller använda reflektion)?
- Kort svar. bakåtkompatibilitet.
- Långt svar. Vi behöver diskutera **type erasure** och **covariant arrayer**.



# Generic-datatyper: autoboxing

Generics funkar inte med primitivtyper.

Vad gör man med primitivtyper?

- Wrapper-typ.

- Varj primitivtyp har en **wrapper** object typ.
- T.ex.: `Integer` är en wrappertyp för `int`.

- Autoboxing är en automatisk kast mellan en primitivtyp och sin wrapper.

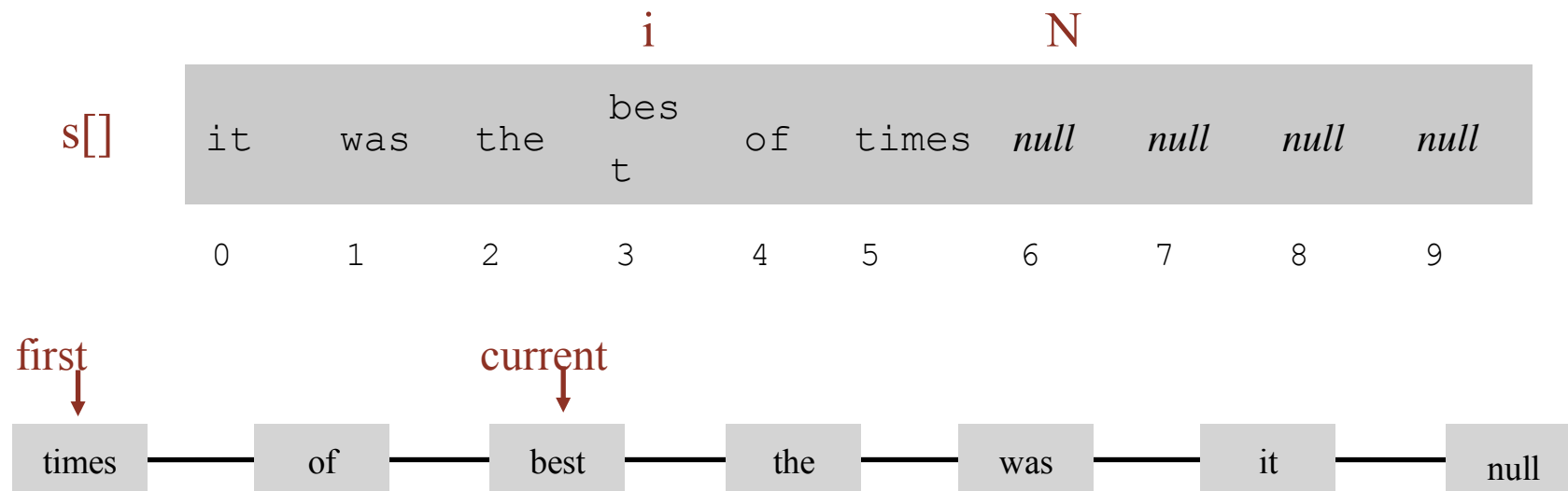
```
Stack<Integer> s = new Stack<Integer>();  
s.push(17);           // s.push(Integer.valueOf(17));  
int a = s.pop();      // int a = s.pop().intValue();
```

**Resultat.** Klient code kan använda en generic stack för alla datatyper.

# Iteratorer

# Iteration

- **Design utmaning:** Hur kan vi stödja iteration över element i en stack, utan att avslöja den interna representationen av stacken?



- **Java lösningen.** Skriv stack klassen så att den implementerar `java.lang.Iterable` interface.

# Iteratorer

- Vad är en `Iterable` ?
- En interface med en metod som returnerar en `Iterator`.
- Vad är en `Iterator` ?
- En interface med metoderna `hasNext()` och `next()`.
- Varför designer man datastrukturer som är `Iterable` ?
- Java stödjer eleganta klient kod.

```
java.lang.Iterable interface
```

```
public interface Iterable<Item> {  
    Iterator<Item> iterator();  
}
```

```
java.util.Iterator interface
```

```
public interface Iterator<Item> {  
    boolean hasNext();  
    Item next();  
    void remove(); ← optional; use  
                                     at your own  
                                     risk  
}
```

```
equivalent code (longhand)
```

```
Iterator<String> i = stack.iterator();  
while (i.hasNext()) {  
    String s = i.next();  
    StdOut.println(s);  
}
```

“foreach” statement (shorthand)

```
for (String s : stack) {  
    StdOut.println(s);  
}
```

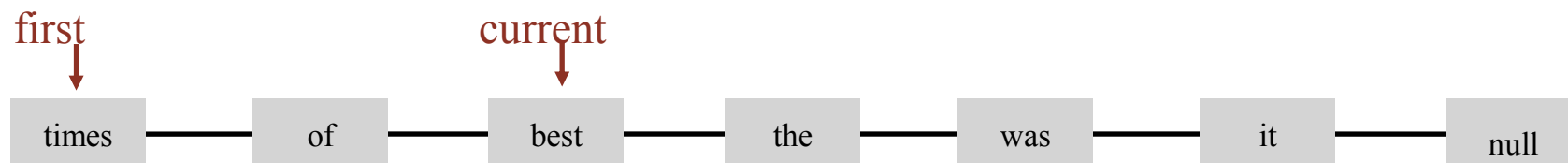
# Stack iterator: länkad lista implementation

```
import java.util.Iterator;
public class Stack<Item> implements Iterable<Item> {
    ...
    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item> {
        private Node current = first;
        public boolean hasNext() { return current != null; }
        public void remove() { /* not supported */ }
        public Item next() {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

kasta UnsupportedOperationException

kasta NoSuchElementException  
om inga items finns kvar

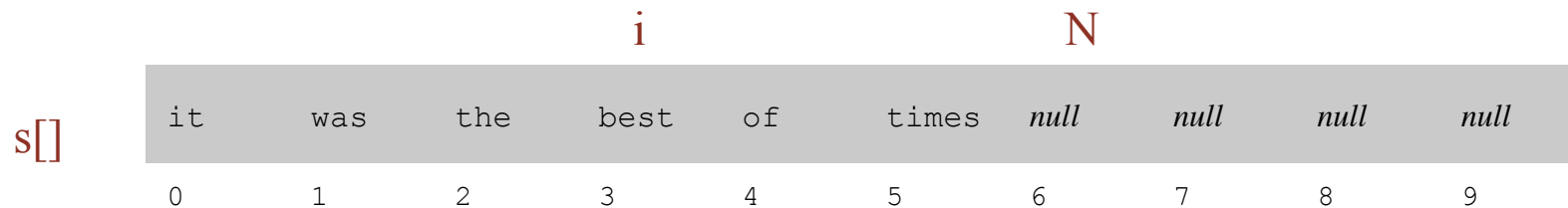


# Stack iterator: array implementation

```
import java.util.Iterator;
public class Stack<Item> implements Iterable<Item> {
    public Iterator<Item> iterator() {
        return new ReverseArrayIterator();
    }

    private class ReverseArrayIterator implements Iterator<Item> {
        private int i = N;

        public boolean hasNext() { return i > 0; }
        public void remove() { /* not supported */ }
        public Item next() { return s[--i]; }
    }
}
```



# Iteration: concurrent modifieringar

- Vad händer om klienten modifierar datastrukturen medans vi itererar över den?
- En *"fail-fast"* iterator kaster en `java.util.ConcurrentModificationException`.

```
for (String s : stack) {  
    stack.push(s);  
}
```

- Hur kan man upptäcka concurrent modifieringar med en iterator?
  - Räkna antal `push()` och `pop()` operationer i `Stack`.
  - Spara sammanräkningar i en `Iterator` subklass när instanser skapas.
  - Om, när vi anropar `next()` eller `hasNext()`, och de nuvarande sammanräkningar inte likar de nuvarande räkningar, kaster man en exception.

# Applikationer



# Java collections bibliotek

- **List interface.** `java.util.List` är ett API för en sekvens av element.

```
public interface List<Item> implements Iterable<Item>
```

---

<code>List()</code>	<i>create an empty list</i>
---------------------	-----------------------------

<code>boolean isEmpty()</code>	<i>is the list empty?</i>
--------------------------------	---------------------------

<code>int size()</code>	<i>number of items</i>
-------------------------	------------------------

<code>void add(Item item)</code>	<i>append item to the end</i>
----------------------------------	-------------------------------

<code>Item get(int index)</code>	<i>return item at given index</i>
----------------------------------	-----------------------------------

<code>Item remove(int index)</code>	<i>return and delete item at given index</i>
-------------------------------------	--

<code>boolean contains(Item item)</code>	<i>does the list contain the given item?</i>
--	--

<code>Iterator&lt;Item&gt; iterator()</code>	<i>iterator over all items in the list</i>
--	--

...

**Implementations.** `java.util.ArrayList` använder en resizing array;

`java.util.LinkedList` använder en länkad lista.

# Java collections bibliotek

- `java.util.Stack`
  - Den stöder `push()`, `pop()`, och iteration.
  - Den subklasser `java.util.Vector`, som i sin tur implementerar `java.util.List` interface, inklusiv `get()` och `remove()` metoderna.
  - Stack API:t är ett för-bred och dålig designad API (varför?)

Java 1.3 bug report (June 27, 2001)

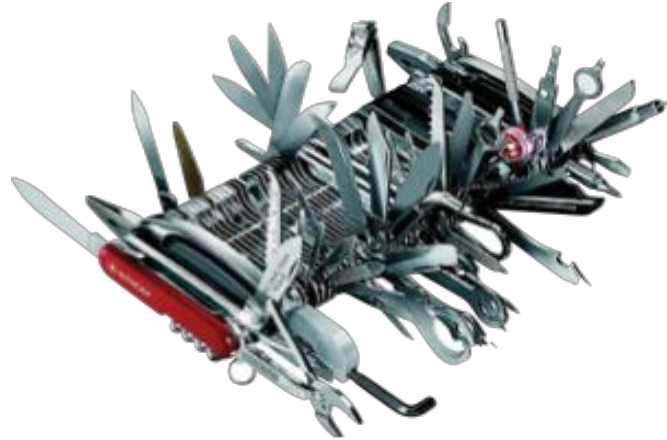
The iterator method on `java.util.Stack` iterates through a Stack from the bottom up. One would think that it should iterate as if it were popping off the top of the Stack.

status (closed, will not fix)

It was an incorrect design decision to have Stack extend Vector ("is-a" rather than "has-a"). We sympathize with the submitter but cannot fix this because of compatibility.

# Java collections bibliotek

- `java.util.Stack`.
  - Den stöder `push()`, `pop()`, och iteration.
  - Den subklasser `java.util.Vector`, som i sin tur implementerar `java.util.List` interface, inklusiv `get()` och `remove()` metoderna.

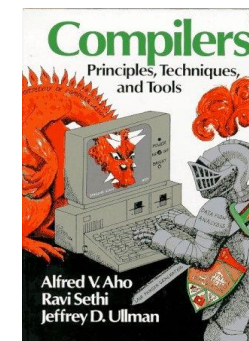
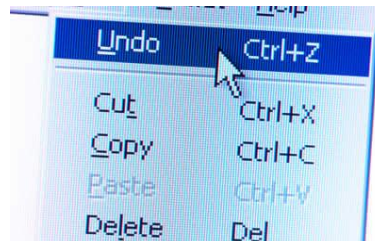


`java.util.Queue`. Ett interface, inte en implementation av en kö.

**Bästa praxis i ID1020.** Återanvänd implementationer av `Stack`, `Queue`, och `Bag` från kursboken.

# Stack applikationer

- Parsing i en kompilator
- Java virtuell maskin.
- Undo i MS-Word.
- Back knapp i en Web browser.
- Implementeringen av funktionsanrop i en kompilatorn.
- ...



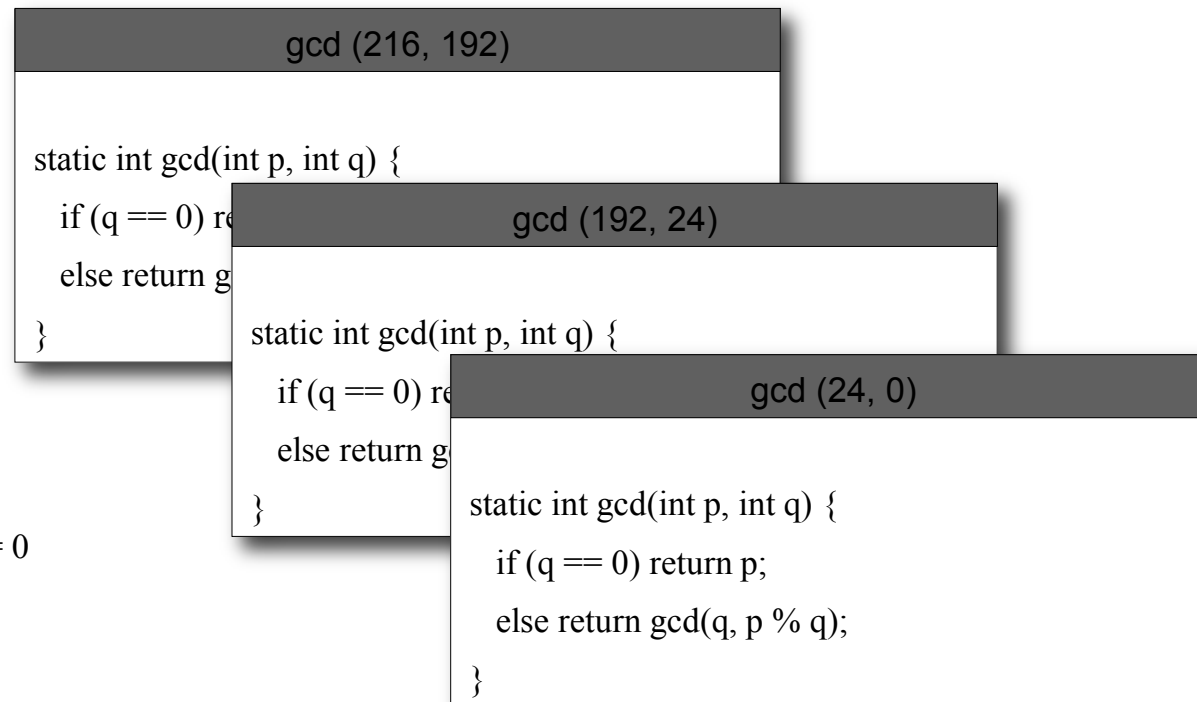
# Funktionsanrop

- Hur en kompilator implementerar en function.
  - Funktionsanrop: **push** lokala stackframe och returadress.
  - Returnera: **pop** returnera adressen och den lokala stackframe.
- Rekursiv funktion. En funktion som anropar sig själv.
- Anmärkning. Kan alltid använda en explicit stack för att ta bort rekursion.

p = 216, q = 192

p = 192, q = 24

p = 24, q = 0



# Aritmetisk funktion utvärdering

- **Mål.** Utvärdera infix uttryck.

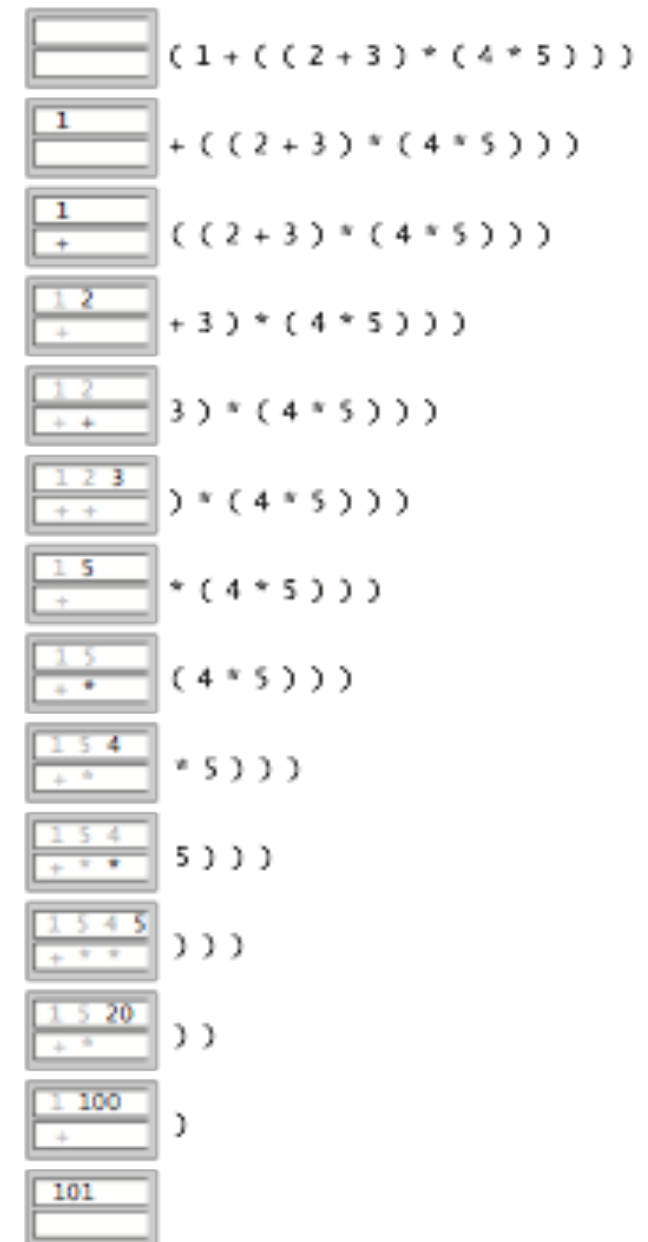
$(1 + ((2 + 3) * (4 * 5)))$

operand      operatorn

**operand stack**  
**operator stack**

- **Två-stack algoritm.** [E. W. Dijkstra]
  - Värde: lägg på operand-stack.
  - Operator: lägg på operator stack.
  - Vänster parentes: ignorera.
  - Höger parentes: `pop` operator och två operander; lägg resultaten man får när man kör operatorn med dem två värden som ligger på operand stacken.

- **Kontext.** En interpreter!



# Dijkstra's två stack algoritm demo



value stack



operator stack

**infix expression**

**(fully parenthesized)**



operand

operator

# Dijkstra's två stack algoritm: aritmetisk funktion utvärdering

```
public class Evaluate {  
    public static void main(String[] args)    {  
        Stack<String> ops  = new Stack<String>();  
        Stack<Double> vals = new Stack<Double>();  
        while (!StdIn.isEmpty()) {  
            String s = StdIn.readString();  
            if      (s.equals("("))                ; // do nothing  
            else if (s.equals("+")) ops.push(s);  
            else if (s.equals("*")) ops.push(s);  
            else if (s.equals(")")) {  
                String op = ops.pop();  
                if      (op.equals("+")) vals.push(vals.pop() + vals.pop());  
                else if (op.equals("*")) vals.push(vals.pop() * vals.pop());  
            }  
            else vals.push(Double.parseDouble(s));  
        }  
        StdOut.println(vals.pop());  
    }  
}
```

```
% java Evaluate  
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )  
101.0
```



# Korrekthet

- Varför är algoritmen korrekt?
- När algoritmen möter en operator omgiven av två värden inom parenteser, lägger den resultatet på operand stacken.

$(1 + ((2 + 3) * (4 * 5)))$

- som original input hade varit:

$(1 + (5 * (4 * 5)))$

- Upprepa argumenten:

$(1 + (5 * 20))$

$(1 + 100)$

101

- Bygge ut. Flera operationer, ordning av operator, *associativity* .

# Stack-based programming languages

- **Observation 1.** Dijkstra's två-stack algoritm beräknar samma värde om operatören kommer efter de två värden.

$(1((23+)(45*)*)+)$

- **Observation 2.** Alla paranteser är redundant!

$1\ 2\ 3\ +\ 4\ 5\ *\ * +$



Jan Lukasiewicz

- **Slutsatsen.** Postfix eller "omvänd Polish" notation.