ID1020: Stacks and Queues

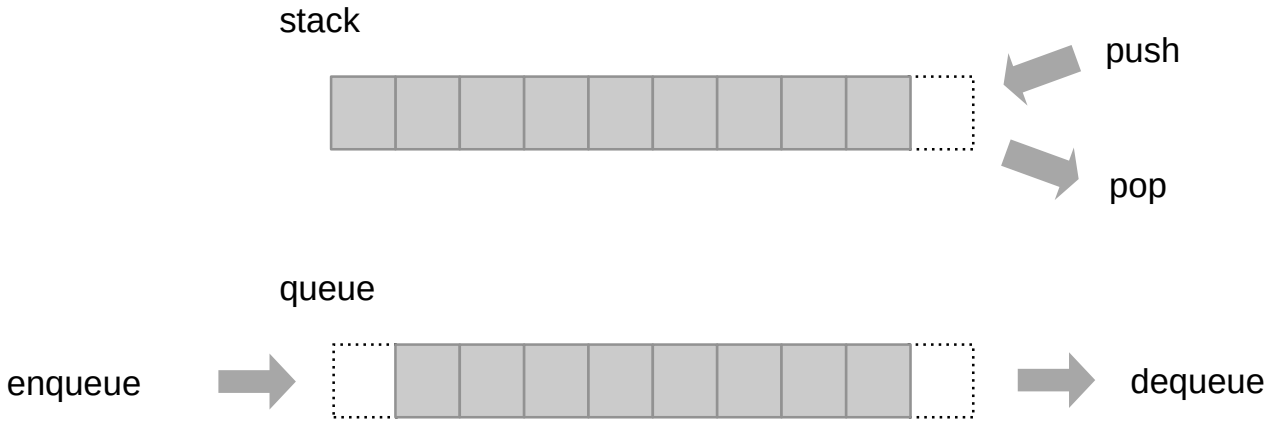
Dr. Per Brand
pbrand@kth.se

chap. 1.3

Slides adapted from Algorithms 4th Edition, Sedgewick.

Stacks and queues

- Fundamental datatypes.
  - Collections of objects.
  - Operations: insert, remove, iterate, check if empty.
  - Obvious what it means to insert an object.
  - But which element should be removed?



- Stack. Remove *most recently added* ← LIFO = "last in first out"
- Kö. Remove *least recently added*. ← FIFO = "first in first out"

Client, implementation, interface

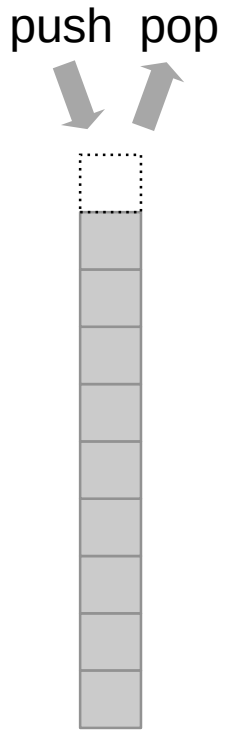
- Separate the interface from the implementation.
  For example, stack, queue, priority queue, symbol table, union-find, ….
- Advantages.
  - Client does not need and should not know the details of the implementation
  - Implementation may be changed without impacting client
  - Client can choose from different implementations.
  - Implementationen does and should not know the details of client needs
  - Different and diverse clients can use the same implementation.
  - Design: create modules, reusable libraries.
  - Performance: can use an optimized implementation when needed.

Client:	programs using operations defined in the interface.
Implementation:	code that implements the operations.
Interface:	describes the operations or services offered..

Bags, stacks and queues

Stack API

Exemple API. Stack of the String datatype.



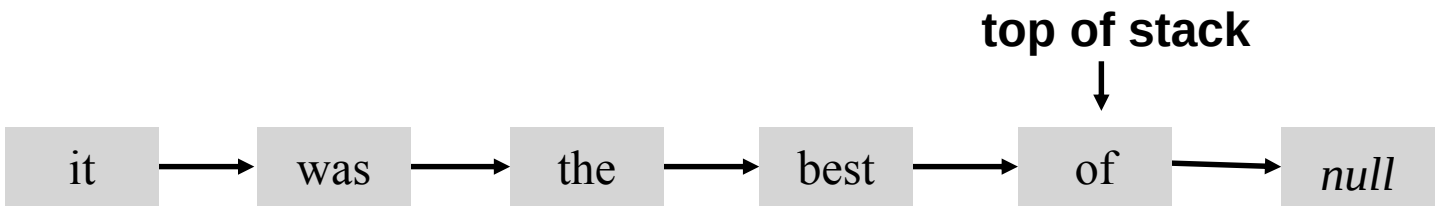
public class StackOfStrings	
StackOfStrings()	*create an empty stack*
void push(String item)	*insert a new string onto stack*
String pop()	*remove and return the string most recently added*
boolean isEmpty()	*is the stack empty?*
int size()	*number of strings on the stack*

Example. Reverse a sequence of strings from standard input.

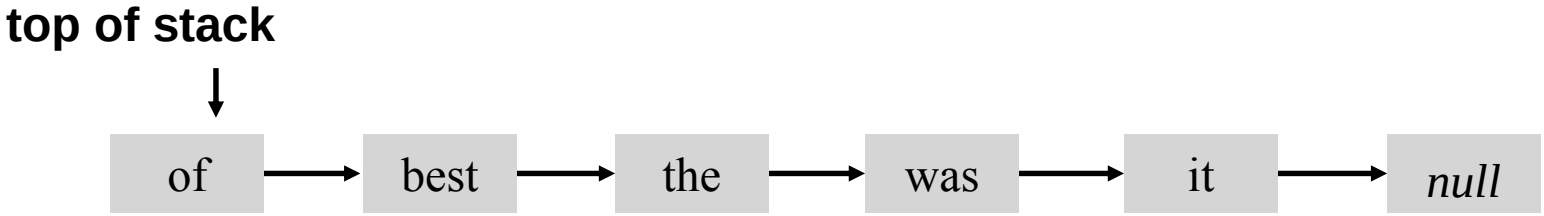
How can we implement a stack with a linked list?

A. It cannot be done efficiently.

B.

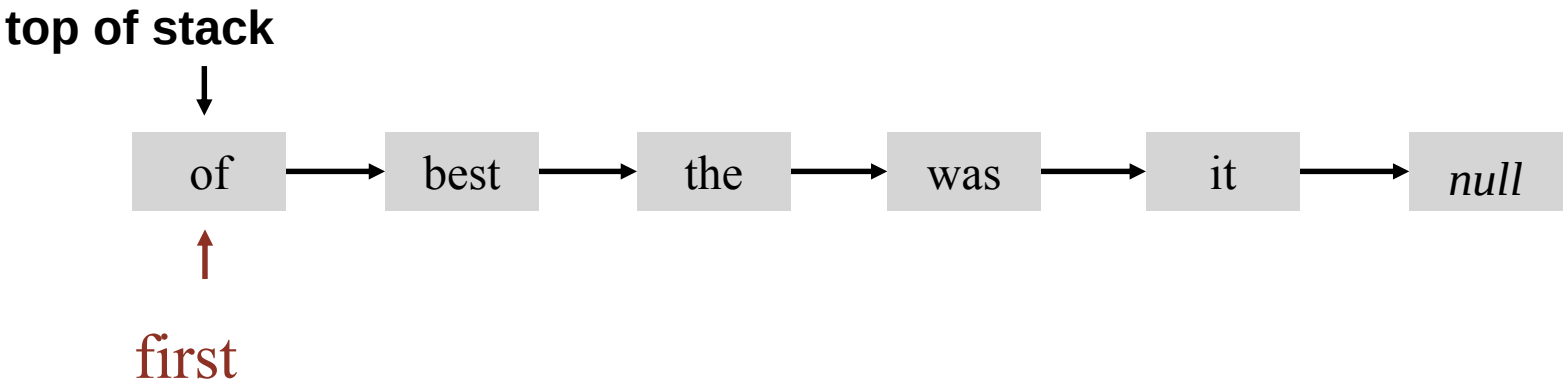


C.



Stack: linked list implementation

- Use `first` to reference the first node in a linked list (singly-linked).
- `Push` create a new element to place before `first` and uppdate `first`.
- `Pop` return the first element and update `first`.



Stack pop: linked list implementation

inner class

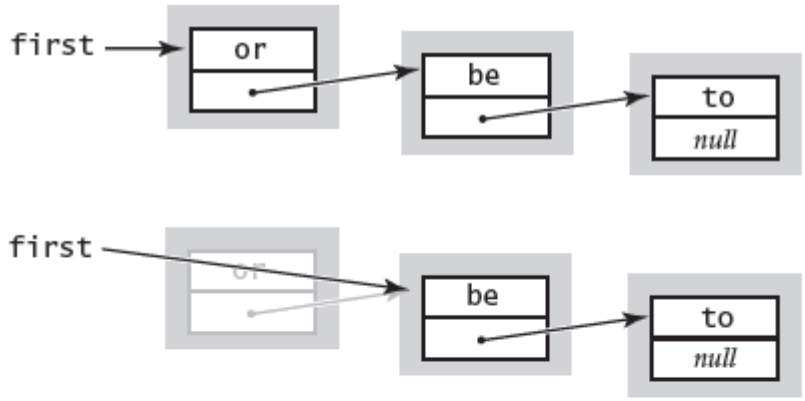
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

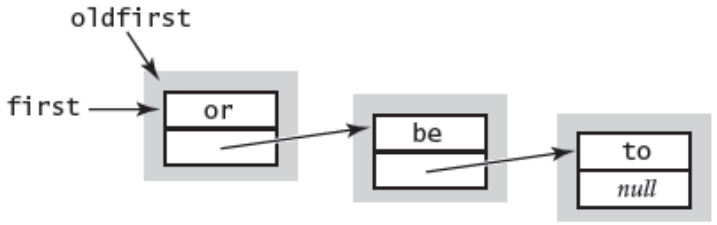
Stack push: linked list implementation

inner class

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

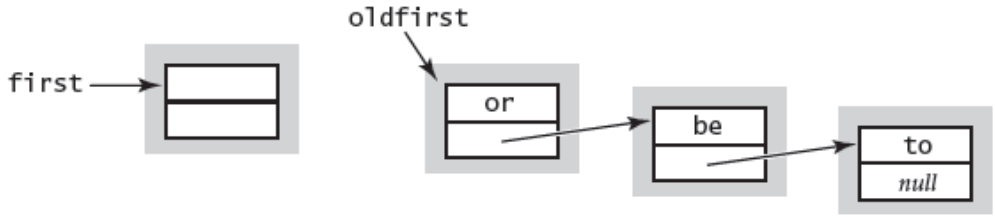
save a link to the list

```
Node oldfirst = first;
```



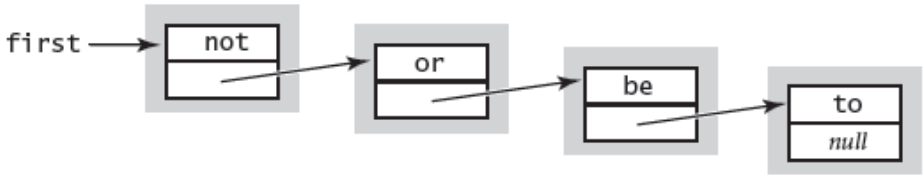
create a new node for the beginning

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



set the instance variables in the new node

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



Stack: linked list implementation in 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;
   }
}
```

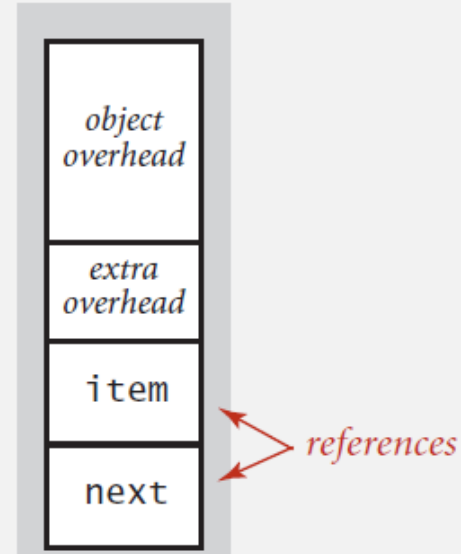
Within inner class - access modifiers unnecessary

Stack: linked list implementation performance

- Theorem. Every operation can be done in constant time even in worst case.
- Theorem A stack with N element uses ~ $40\,N$ bytes of memory

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)

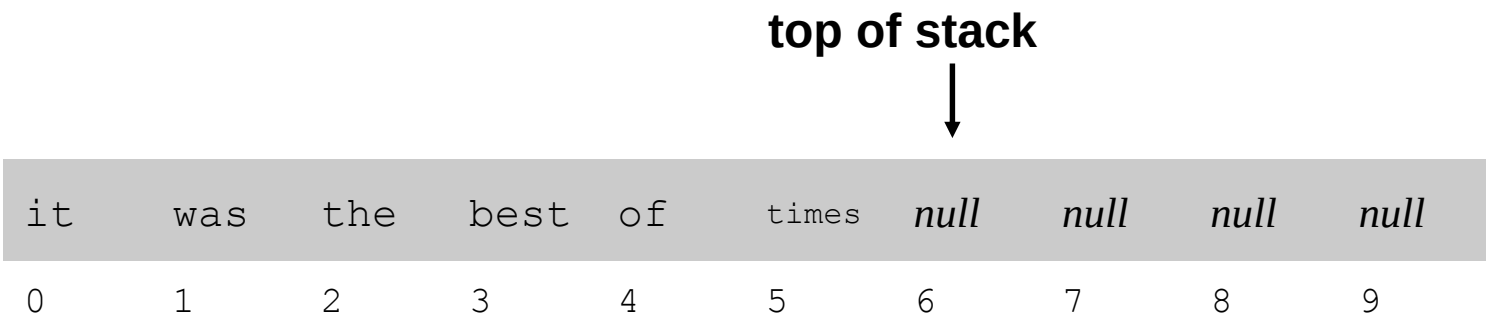
40 bytes per stack node

- Note. Only stack memory is taken into account, i.e.not the memory for the strings themselves which are owned by the client.

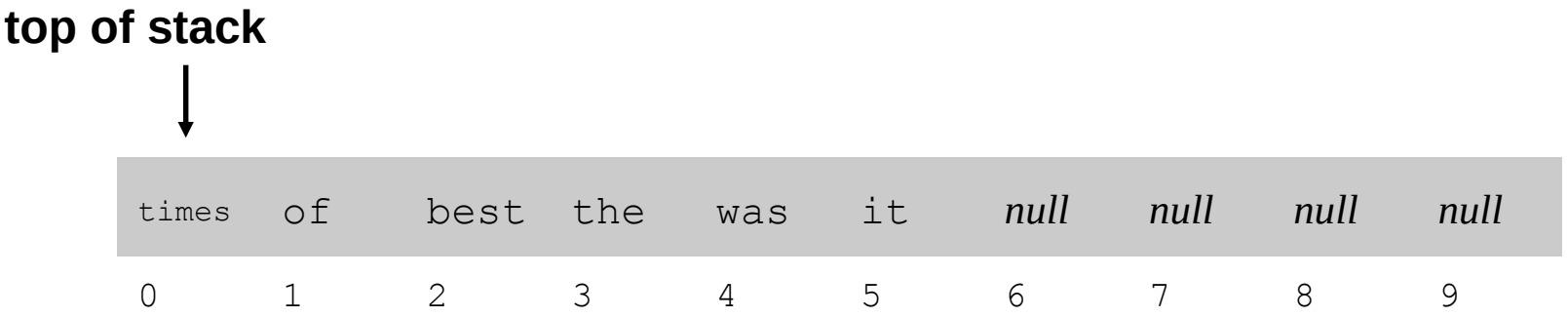
How could we implement a fixed-capacity stack using an array?

A. Cannot be done efficiently.

B.

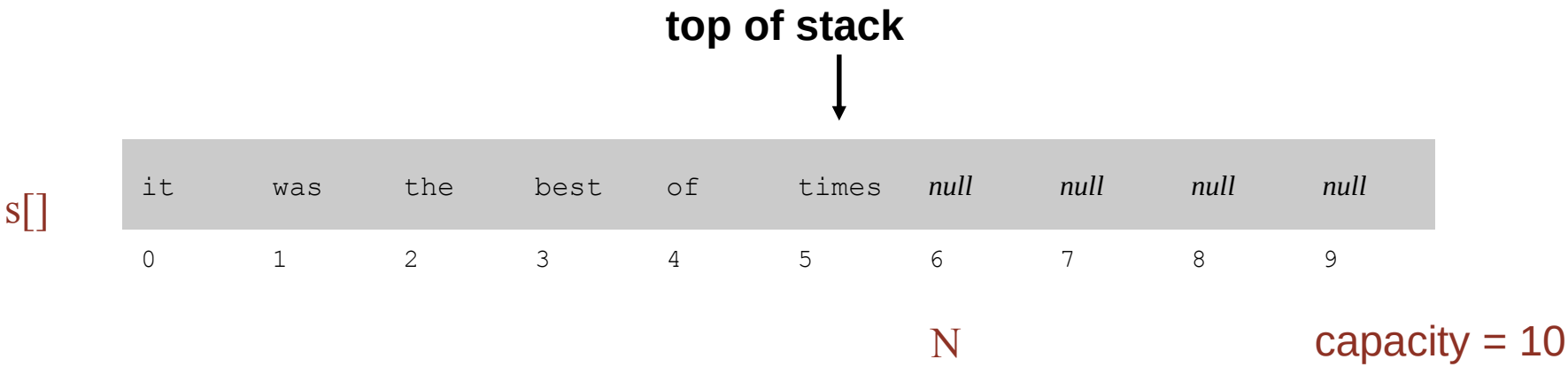


C.



Fixed-capacity stack: array implementation

- Use an array `s[]` to store the `N` elements on the stack
  - `push()`: inserts at `s[N]`.
  - `pop()`: removes at `s[N-1]`.



Defect. Stack-overflow will occur when `N` is larger than capactiy

Fixed-capacity 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];  }

}
```

Why N++ and not ++N?

Why -- N and not N--?

Design considerations

- Overflow and underflow.
  - Underflow: throw and exception if client calls *pop* on an empty stack.
  - Overflow: use a *resizing array* for the array implementation.

- Null element: it is permitted to insert null elements.

- Loitering. *Loitering* happens when references to objects that by program logic can never again be accessed remain live in view of the runtime system and cannot be garbage-collected.

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

loitering

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

This version avoids loitering.

From the poiint-of-view of the garbage collection as long as the array is live then all its elements are !!

Stacks: resizing arrays

Stack: resizing-array implementation

- Problem. API is broken if client has to specify capacity!

- When and by how much should be expand and shrink the array?

- First attempt.
  - `push()`: increase size by `1`.
  - `pop()`: diminish size by `1`.

- To expensive.
  - We need to copy all elements to a new array for each *push* operation.

  Number of array accesses to push N element =

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

1 array access per push

2(k–1) array accesses to increase length to k (iand this does not take into account the cost of creating new arrays and all the work that the garbage collector needs to do)

- Challenge. Ensure that array "resizing" rarely occurs.

Stack: resizing-array implementation

- By how much should the array size be increased? "repeated doubling"
- If the array is full, create a new array twice as large och copy the elements

```
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;
}
```

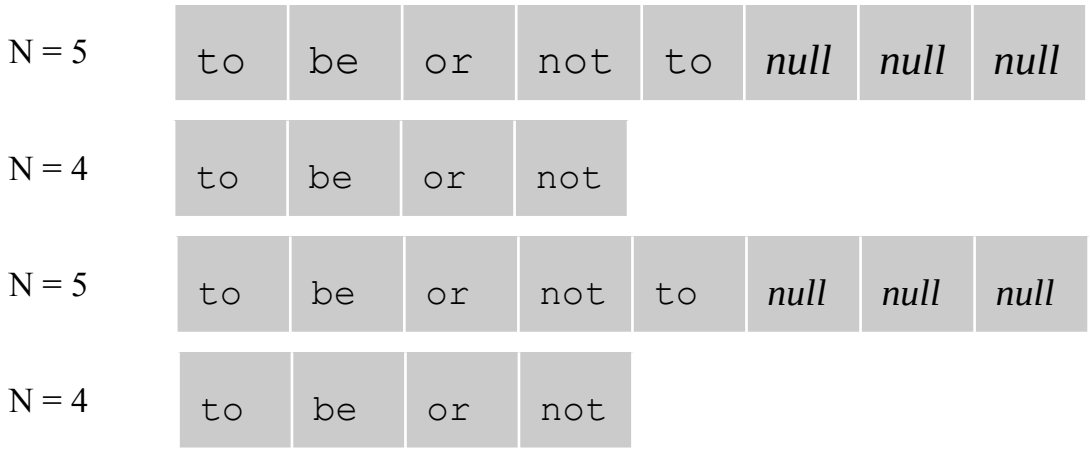
- Result: to time needed to push N element is now proportional to N and not N^2

Stack: resizing-array implementation

- How to diminsh the size of the array?

- First attempt.
  - `push()`: double the size when the array is full.
  - `pop()`: halve the size when the array is half-full.

- To expensive in worst case.
  - Consider the sequence push-pop-push-pop-… when the array is half full.
  - The time for every operation is proportional to N.



Stack: resizing-array implementation

- Efficient solution.
  - `push()`: double the size when the array is full.
  - `pop()`: halve the size when the array is one quarte full.

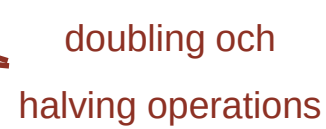
```
public String pop()  {
   String item = s[--N];
   s[N] = null;
   if (N > 0 && N == s.length/4) {
        resize(s.length/2);
   }
   return item;
}
```

- Invariant. Array is always between 25% och 100% full.

Stack resizing-array implementation: performance

- Amorterized analys. Begin with an empty stack, take the average time per operation over a worst-case sequence of operations.
- Theorem. Starting with an empty stack, all possible sequences of *M* push and pop operations consume time that is proportional to N

	best	worst	amortized
construct	1	1	1
push	1	*N*	1
pop	1	*N*	1
size	1	1	1



Stack resizing-array implementation: memory usage

- Theorem. Uses between ~ 8 *N* och ~ 32 *N* bytes for a stack of *N* elements.
  - ~ 8 *N* when full.
  - ~ 32 *N* when a quarter full.
  - ~20 N on average (compare this to 40N for linked list implmentation)

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

8 bytes (reference to array)

24 bytes (array overhead)

8 bytes × array storlek

4 bytes (int)

4 bytes (padding)

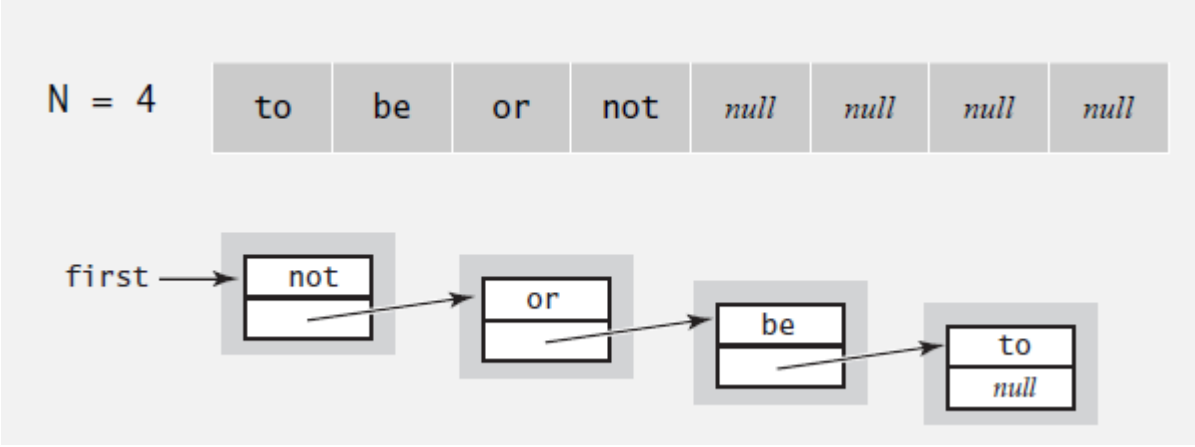
- Note The calculation only takes into account memory usage of the stack (i.e., not the memory of the strings themselves which are owned by the client).

Stack implementations: resizing array vs. linked list

- Choice. Stacks can be implemented using resizing arrays or with linked lists; the client can choose on or the other.
- Which is the better? Classical time vs memory tradeoff

- Linked list implementation.
  - Every operation takes constant time even in worst case.
  - Uses more memory, and push/pop operations more expensive than for an array (where no resizing is done)

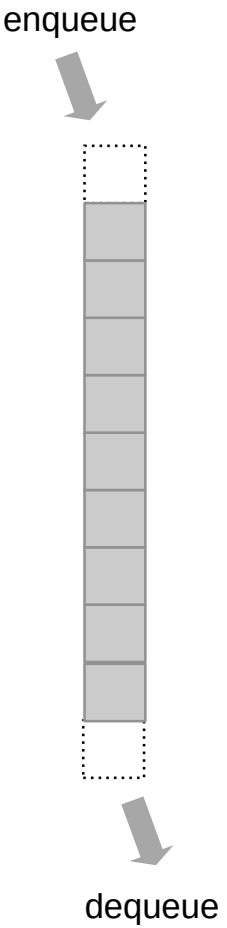
- Resizing-array implementation.
  - Every operation takes constant amortized time.
  - Less memory.

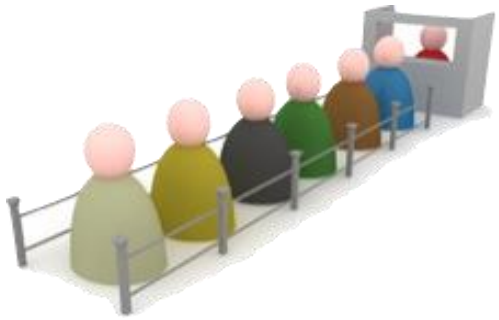


Resizing arrays: queues

Queue API

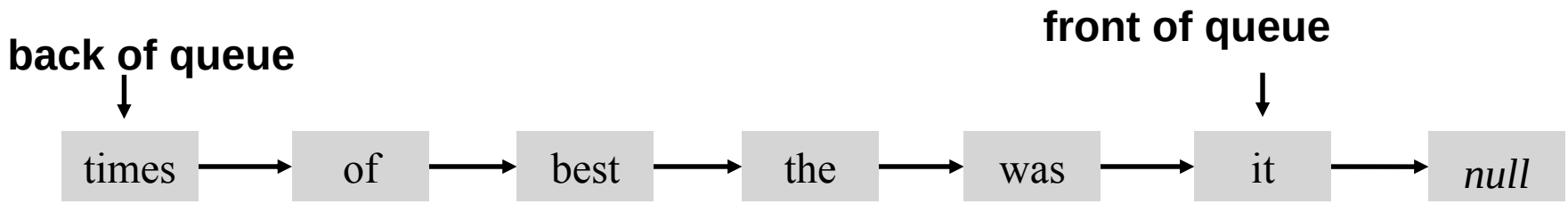
public class QueueOfStrings		
	`QueueOfStrings()`	*create an empty queue*
`void`	`enqueue(String item)`	*insert a new string onto queue*
`String`	`dequeue()`	*remove and return the string least recently added*
`boolean`	`isEmpty()`	*is the queue empty?*
`int`	`size()`	*number of strings on the queue*



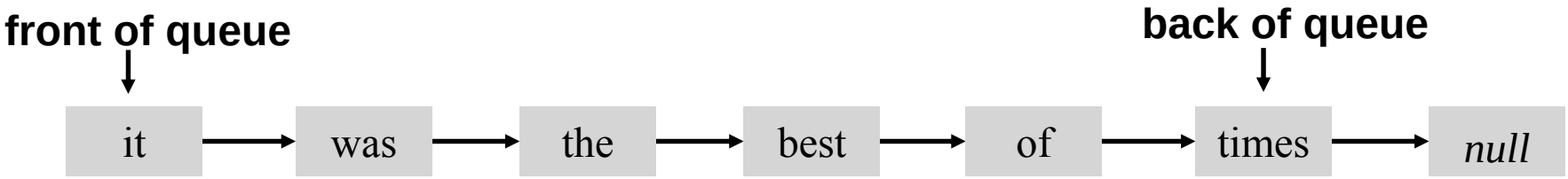


How do we implement a queue using a linked list?

- A. Cannot be done efficiently.
- B.



- C.



Dequeue: linked list implementation

innre klass

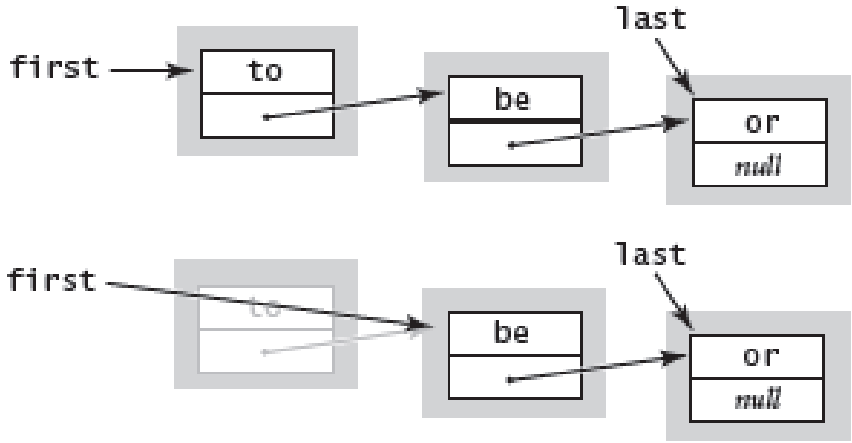
```
private class Node {
   String item;
   Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

- Note. Identical code to linked list stack `pop()`.

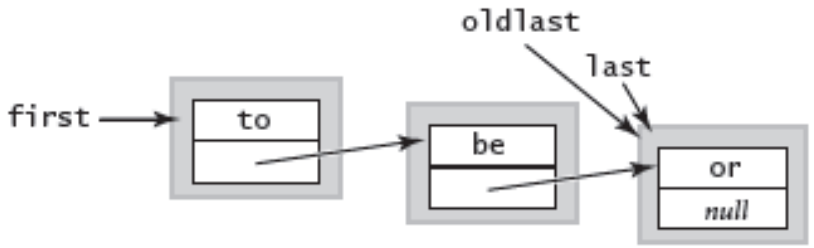
Enqueue: linked list implementation

innre klass

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

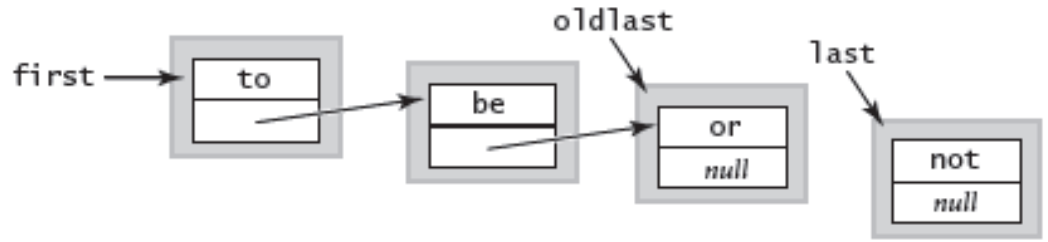
save a link to the last node

```
Node oldlast = last;
```



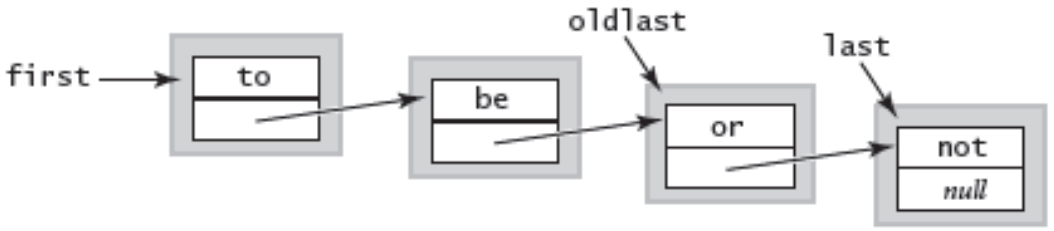
create a new node for the end

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



link the new node to the end of the list

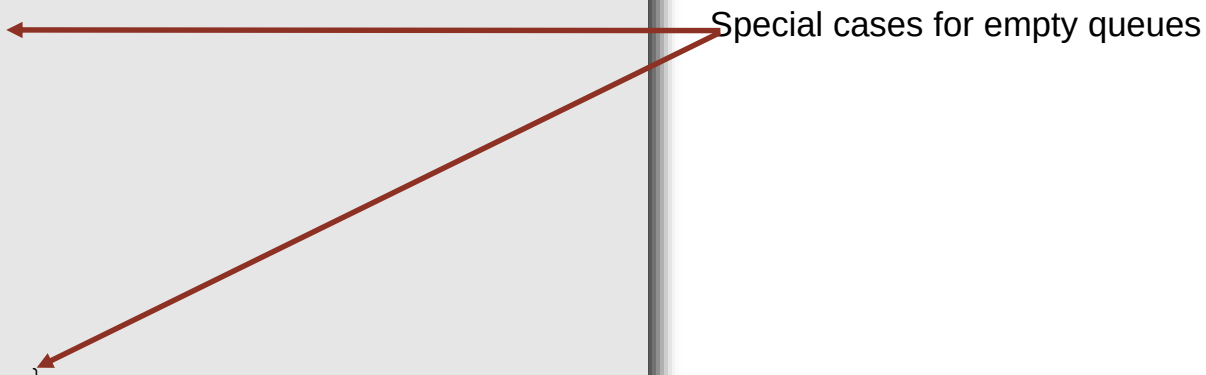
```
oldlast.next = last;
```



Queue: linked list implementation in Java

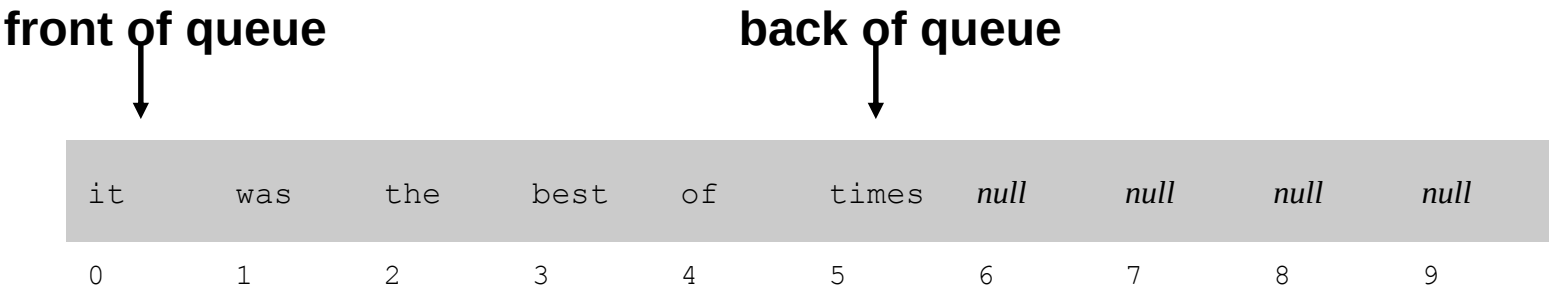
```
public class LinkedQueueOfStrings {
   private Node first, last;
   private class Node {  /* same as in LinkedStackOfStrings */ }
   public boolean isEmpty()  {  return first == null;  }

   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;
   }
}
```

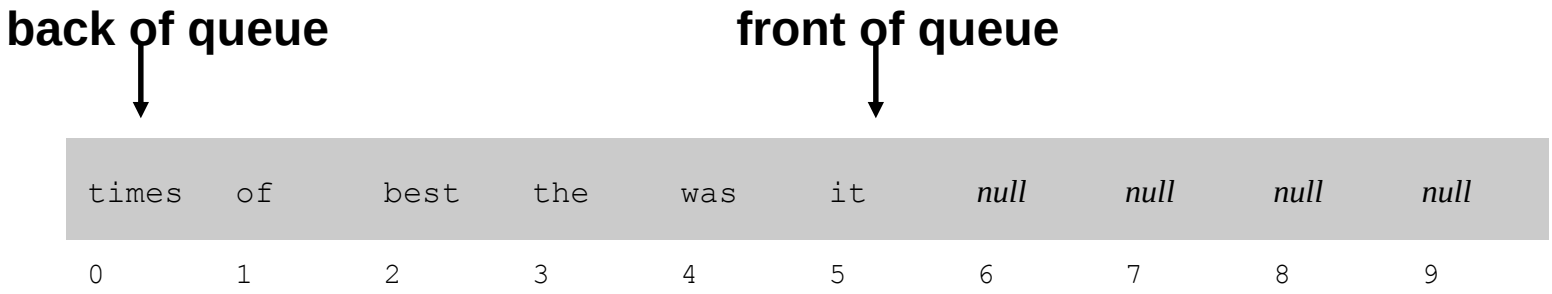


Hur can we implement a queue with an array?

- A. Cannot be done efficiently.
- B.



- C.



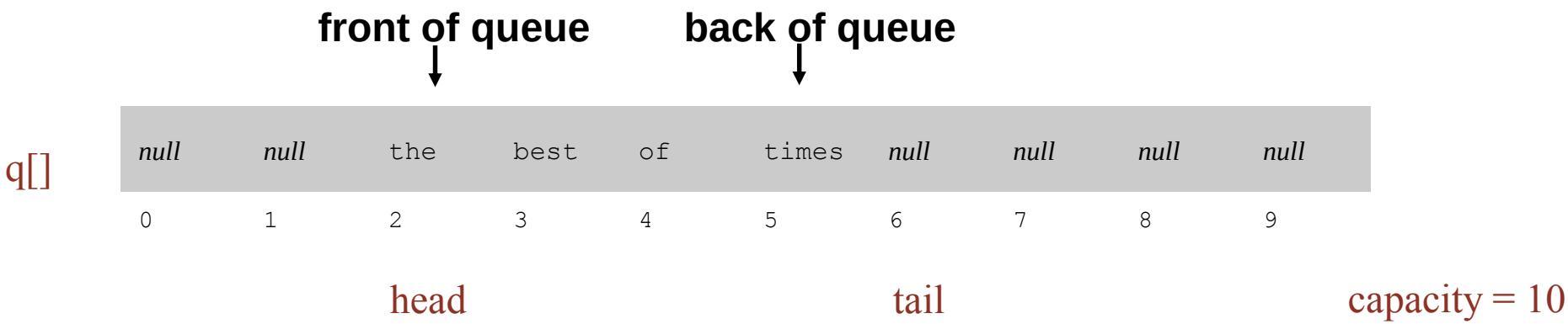
Queue: array implementation

- Use an array `q[]` to store the elements.

  `enqueue()`: Insert new element at `q[tail]`.

  `dequeue()`: Remove element at `q[head]`.

- Uppdate `head` och `tail`.
- Note that the live entries will move to the right (
  - either use wraparound or amoritized move to front)
- Can add a ”resizing array".



- Java code left as exercise.

generics

Parameterized stack

- We have implemented: `StackOfStrings`.
- We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ….

- Attempt 1. Implement a separate class for each data type.
  - BUT this leads to code explosion
    - Lost of extra work – bugs easily introduced
    - Extra maintainence

Parameterized stack (2)

- We have implemented: `StackOfStrings`.
- We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ….

- Attempt 2. Implement a stack with elements of type `Object`.
  - Client will need to cast from `Object` to the specific type.
  - To case tends to introduce errors: *runtime errors* when types do not match.

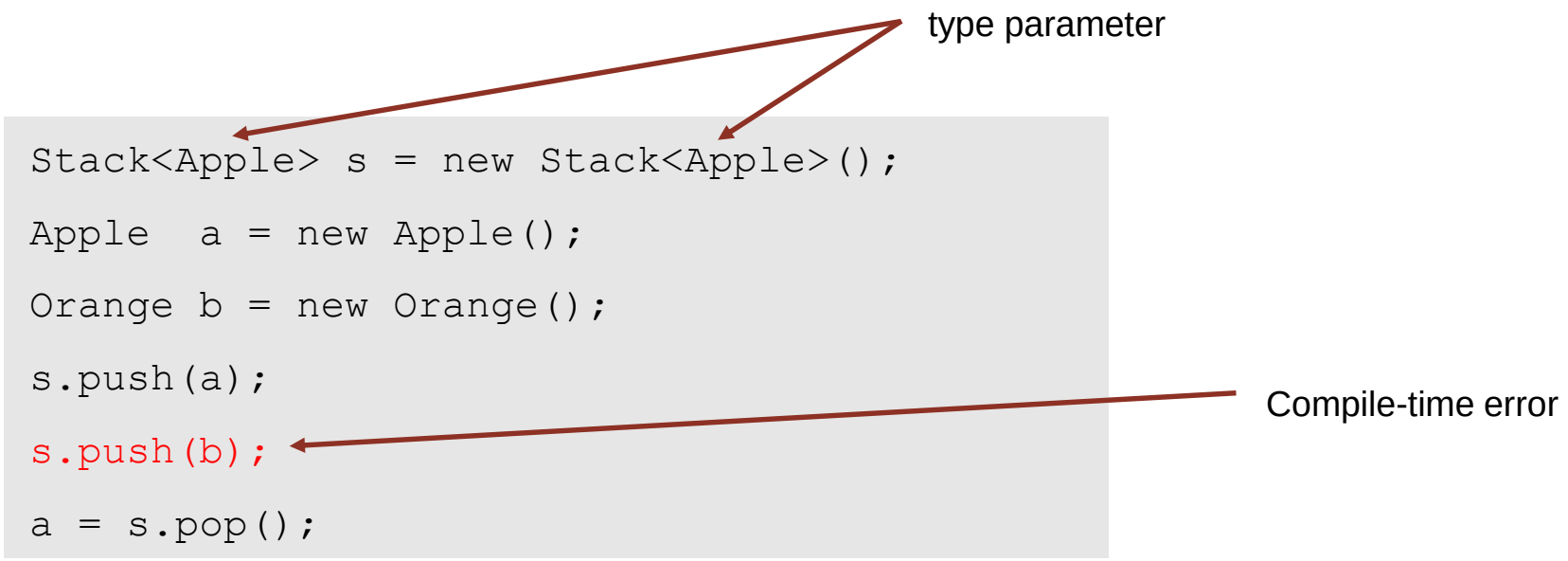
```
StackOfObjects s = new StackOfObjects();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

ClassCastException

Parameterized stack (3)

- We have implemented: `StackOfStrings`.
- We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ….

- Attempt 3. Java generics.
  - Client does not need to catch.
  - Type mismatch detected at compile time instead of at runtime.



- Advantage. Compilation errors better than runtime errors. Special case of handle errors as soon as possible

Generic stack: linked list 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 -type

Generic stack: array implementation that doesn't work

@#$*! generic arrays are not allowed in 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 that does work

```
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];  }
}
```

Ugh. Vi kaster här.

```
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
```

- Why do we need to cast in Java ?
- Short answer: backwards compatability
- Long answer. Beyond scope of book (involving type erasure och covariant arrayer).

Generic-datatypes: autoboxing

At first glance generics won't work with primtive data types (which are not objects)

- Wrapper-type.
  - Every primitive type has a wrapper object type.
  - E.g. `Integer` is a wrapper for `int`.

- Autoboxing is an automaticc cast between primitive types and their wrappers.

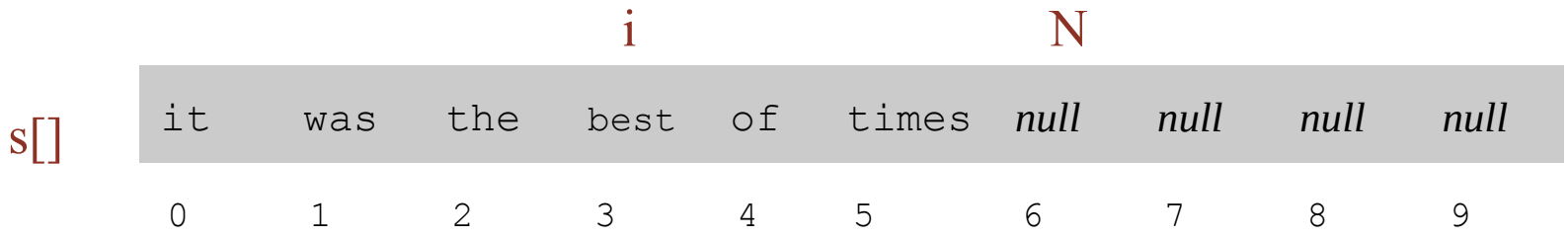
```
Stack<Integer> s = new Stack<Integer>();
s.push(17);          // s.push(Integer.valueOf(17));
int a = s.pop();     // int a = s.pop().intValue();
```

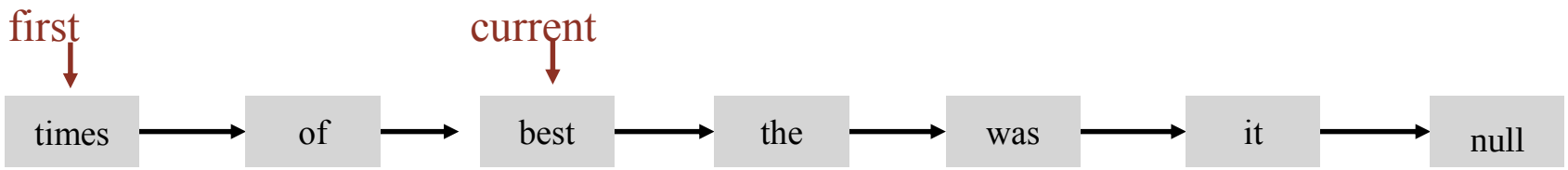
Result Client code can use generic data structures like stacks even for primitive types.

Iterators

Iteration

- Design challenge: How can we support iteration over the elements in a stack, (or queue or many other data structures) without revealing the implementation representation?





- Java solution. Design the class so that it implements iterators as given by the `java.lang.Iterable` interface.

Iterators

- What is an `Iterable` ?
- An inferface with a method that returns an `Iterator`.

- What is an `Iterator` ?
- An inferface with the metods `hasNext()` and `next()`.

- Why use these?
- Makes for elegant client code.

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

"foreach" statement (shorthand)

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

equivalent code (longhand)

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

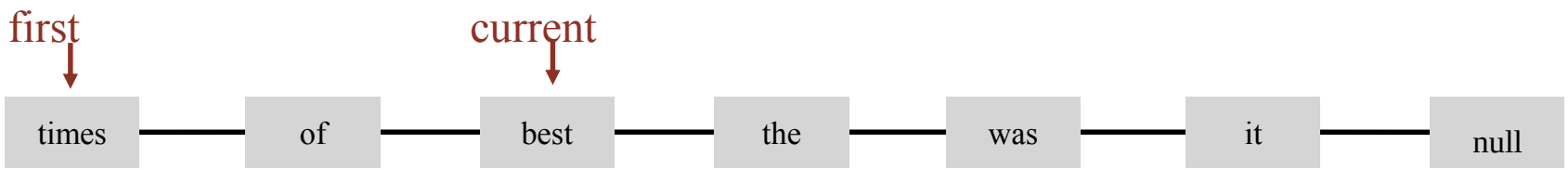
Stack iterator: linked list implementation

```
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;
        }
   }
}
```

thows UnsupportedOperationException

Throws NoSuchElementException

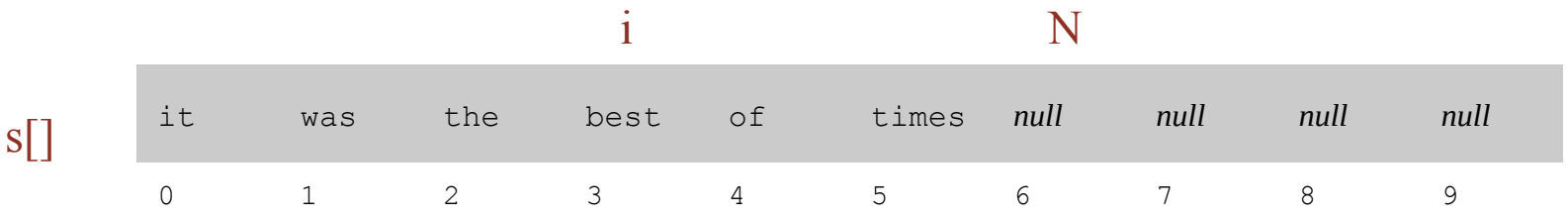


Stack iterator: array implementation

```
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];        }
    }
}
```



Applications

Java collection libraries

- List interface. `java.util.List` is an API for sequences of elements.

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

	`List()`	*create an empty list*
`boolean`	`isEmpty()`	*is the list empty?*
`int`	`size()`	*number of items*
`void`	`add(Item item)`	*append item to the end*
`Item`	`get(int index)`	*return item at given index*
`Item`	`remove(int index)`	*return and delete item at given index*
`boolean`	`contains(Item item)`	*does the list contain the given item?*
`Iterator<Item>`	`iterator()`	*iterator over all items in the list*
	`...`	

Implementations. `java.util.ArrayList` uses a resizing array; `java.util.LinkedList` uses a linked list.

Java collection libraries (2)

- `java.util.Stack`
  - Supports `push()`, `pop()`, och iteration.
  - It subclasses `java.util.Vector`, which in turn implements `java.util.List` interface, including the `get()` and `remove()` methods.
  - The stack API is to broad and poorly designed.

Java 1.3 bug report (June 27, 2001)

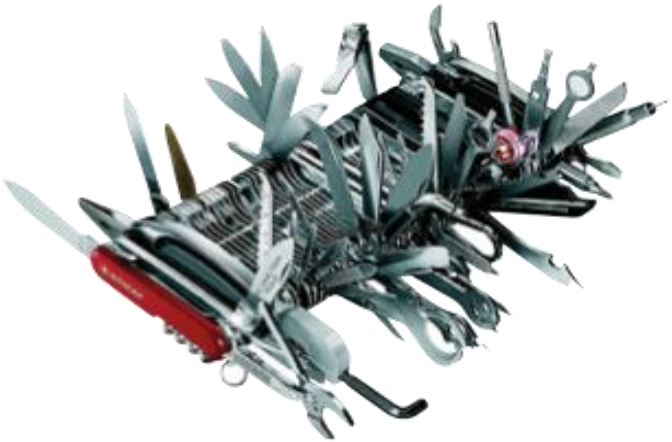
> The iterator method on java.util.Stack iterates through a Stack from the bottom up. One would think that it should iterate as if it were popping off the top of the Stack.

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 collection libraries (3)

- `java.util.Stack`.

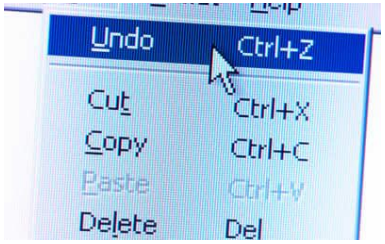


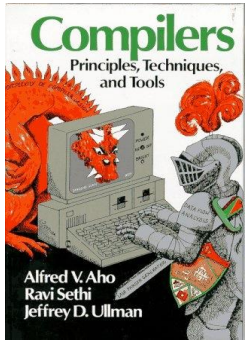
`java.util.Queue`. An interface, not a queue implementation

Best practice in ID1020. Reuse implementations of `Stack`, `Queue`, and `Bag` from the course.

Stack applications

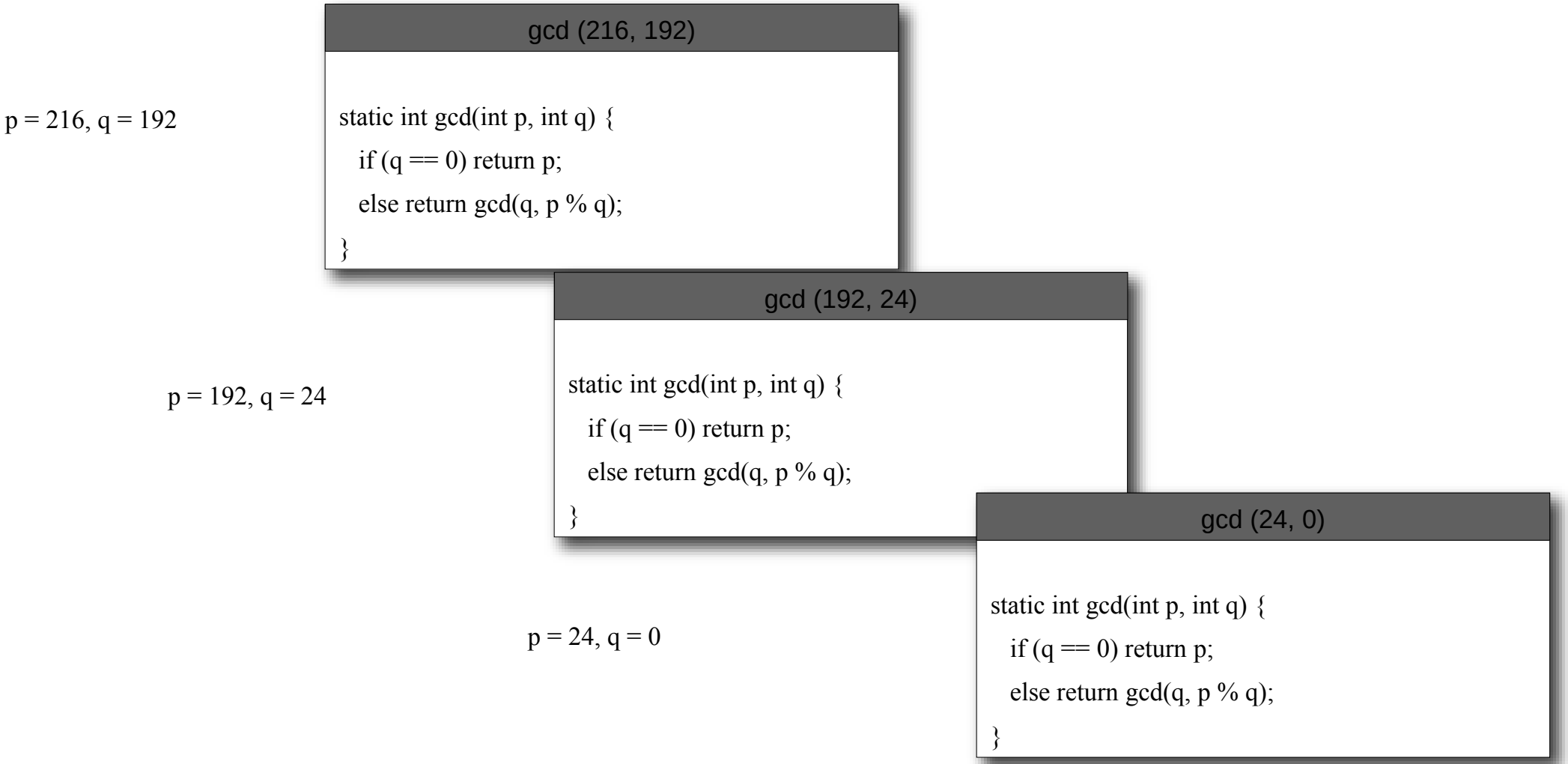
- Parsing in a compiler
- Java VM (Virtual Machine)
- Undo in MS-Word.
- Back button in a Web browser.
- Procedure/method/function calls in a runtime system
- ...





Procedure calls

- How a runtime system implements a procedure call.
  - Call: push the local stackframe and the return address.
  - Return: pop returnes the address and the relevant stackframe.



Evaluation of an arithmetic expression

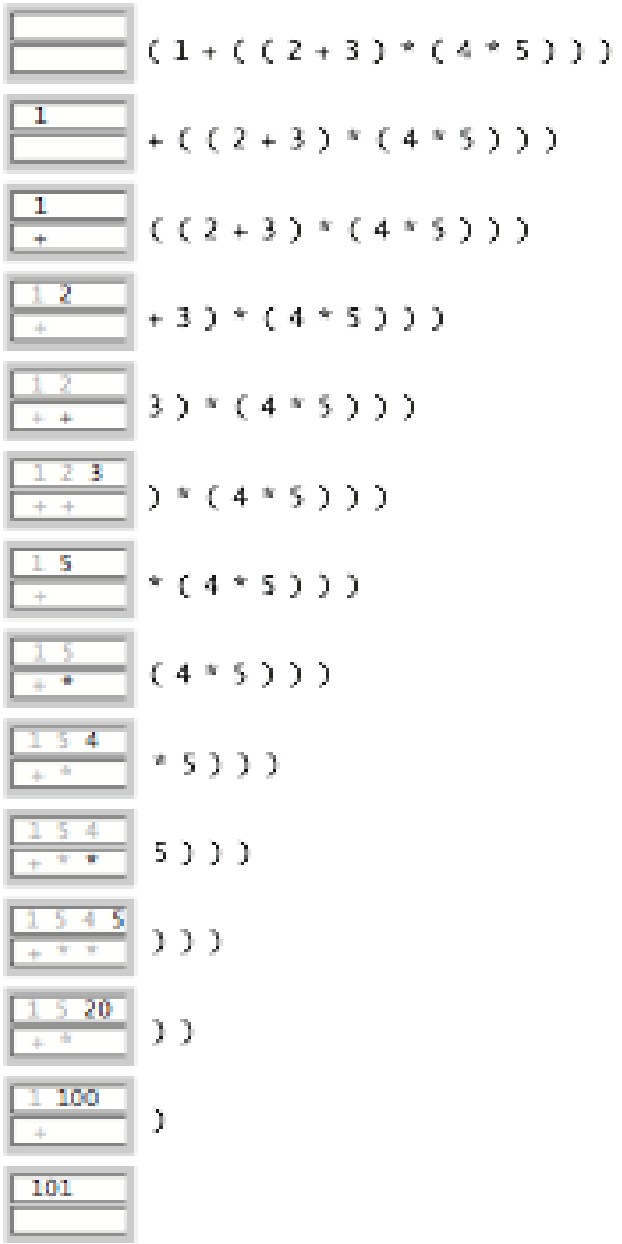
- Mål. Evaluate infix .

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

operand operatorn

- Två-stack algorithm. [E. W. Dijkstra]
  - Value: push on the operand-stack.
  - Operator: push on the operator stack.
  - Left parenthesis: ignore
  - Right parenthesis: `pop` operator och två operands Evaluate. Push the result on the operand stack.

- Context. An interpreter!



Stack-based programming languages

- Observation 1. Dijkstra's två-stack algorithm will get the same results if the operator comes after the operands.

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

- Observation 2. This makes all parentheses unnecessary!

1 2 3 + 4 5 * * +

Jan Lukasiewicz

- Conclusion. Can use postfix or "reverse Polish" notation.