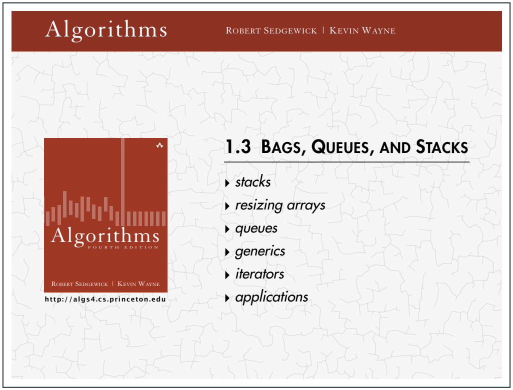
CS2010: ALGORITHMS AND DATA STRUCTURES

Lectures 5 & 6: Abstract Data Types - Stack & Queue

Vasileios Koutavas

School of Computer Science and Statistics
Trinity College Dublin

THIS LECTURE


Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

Algorithms FOURTH EDITION

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

1.3 BAGS, QUEUES, AND STACKS

- *stacks*
- *resizing arrays*
- *queues*
- *generics*
- *iterators*
- *applications*

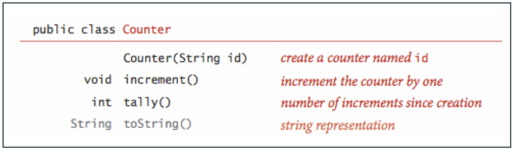
→ Abstract Data Types

→ Stacks and Queues *

→ S&W 1.2 and 1.3

Abstract Data Types

ABSTRACT DATA TYPE

Example: 

public class Counter

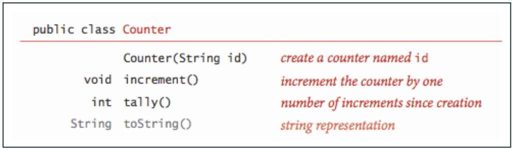
	`Counter(String id)`	*create a counter named* `id`
`void`	`increment()`	*increment the counter by one*
`int`	`tally()`	*number of increments since creation*
`String`	`toString()`	*string representation*

A Data Type is

- → A set of values
  - → in example: all counter objects at state 0, 1, 2, ...
- → A set of operations on those values
  - → in example: `constructor`, `increment`, `tally`, `toString`

ABSTRACT DATA TYPE

Example:



`public class Counter`

	`Counter(String id)`	*create a counter named* `id`
`void`	`increment()`	*increment the counter by one*
`int`	`tally()`	*number of increments since creation*
`String`	`toString()`	*string representation*

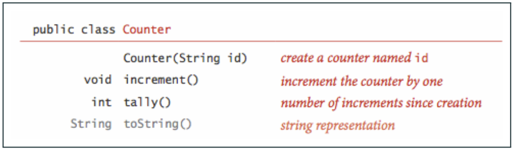
A Data Type is

→ A set of values
  → in example: all counter objects at state 0, 1, 2, ...
→ A set of operations on those values
  → in example: `constructor`, `increment`, `tally`, `toString`

An Abstract Data Type (ADT) is

→ A Data Type whose implementation is unknown to the client of the ADT

ABSTRACT DATA TYPE

Example: 

public class Counter		
	Counter(String id)	*create a counter named* id
void	increment()	*increment the counter by one*
int	tally()	*number of increments since creation*
String	toString()	*string representation*

A Data Type is

→ A set of values
  → in example: all counter objects at state 0, 1, 2, ...

→ A set of operations on those values
  → in example: `constructor`, `increment`, `tally`, `toString`

An Abstract Data Type (ADT) is

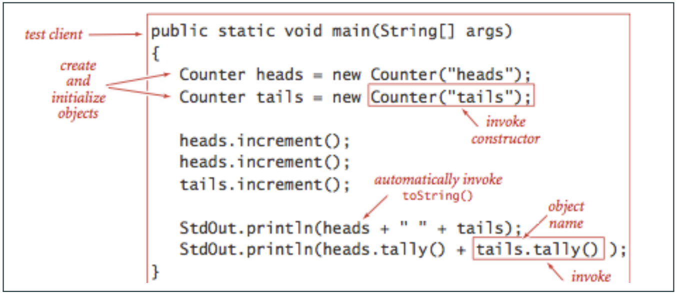
→ A Data Type whose implementation is unknown to the client of the ADT

An Application Programming Interface (API) is

→ A list and informal description of the operations of an ADT

WHO IS THE CLIENT OF AN ADT?

Example:



```
public static void main(String[] args)
{
   Counter heads = new Counter("heads");
   Counter tails = new Counter("tails");

   heads.increment();
   heads.increment();
   tails.increment();

   StdOut.println(heads + " " + tails);
   StdOut.println(heads.tally() + tails.tally() );
}
```

test client
create and initialize objects
invoke constructor
automatically invoke toString()
object name
invoke

→ Client: the rest of the program, using the ADT

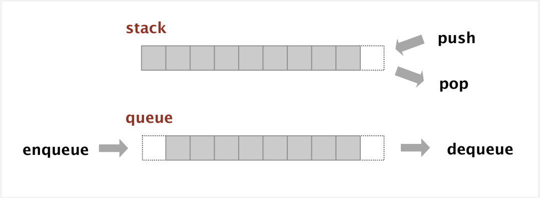
Stacks & Queues

STACKS & QUEUES

Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Operations: insert, remove, iterate, test if empty.
- Intent is clear when we insert.
- Which item do we remove?



Stack. Examine the item most recently added. $\longleftarrow$ LIFO = "last in first out"

Queue. Examine the item least recently added. $\longleftarrow$ FIFO = "first in first out"

Client, implementation, interface

Separate interface and implementation.

Ex: stack, queue, bag, priority queue, symbol table, union-find, ...

Benefits.

- Client can't know details of implementation ⇒ client has many implementation from which to choose.
- Implementation can't know details of client needs ⇒ many clients can re-use the same implementation.
- Design: creates modular, reusable libraries.
- Performance: use optimized implementation where it matters.

Client: program using operations defined in interface.

Implementation: actual code implementing operations.

Interface: description of data type, basic operations.

http://algs4.cs.princeton.edu

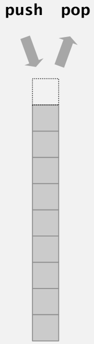
1.3 Bags, Queues, and Stacks

- *stacks*
- *resizing arrays*
- *queues*
- *generics*
- *iterators*
- *applications*

Stack API

Warmup API. Stack of strings data type.

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*

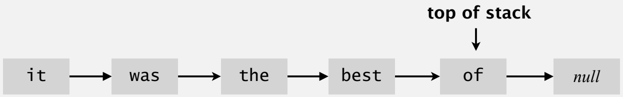


Warmup client. Reverse sequence of strings from standard input.

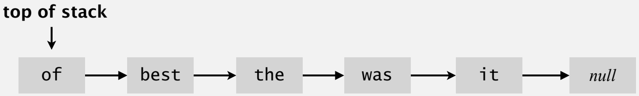
How to implement a stack with a linked list?

A. Can't be done efficiently with a singly-linked list.

B.

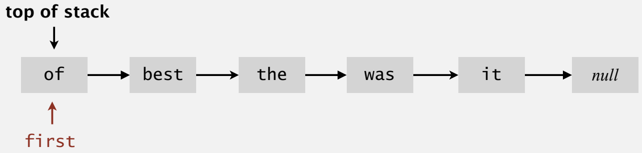


C.



Stack: linked-list implementation

- Maintain pointer `first` to first node in a singly-linked list.
- Push new item before `first`.
- Pop item from `first`.



STACK: LINKED LIST IMPLEMENTATION

http://dsvproject.github.io/dsvproject/code/stackLinkedList.html

Stack pop: linked-list implementation

inner class

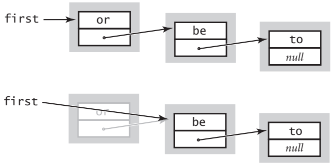
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

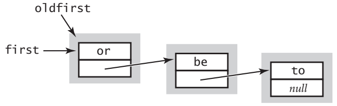
Stack push: linked-list implementation

inner class

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

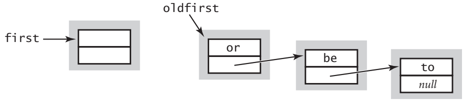
save a link to the list

```
Node oldfirst = first;
```



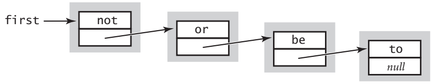
create a new node for the beginning

```
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;
   }
}
```

private inner class
(access modifiers for instance variables don't matter)

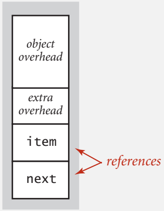
Stack: linked-list implementation performance

Proposition. Every operation takes constant time in the worst case.

Proposition. A stack with N items uses $\sim 40\,N$ 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)

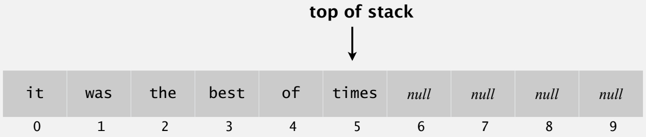
40 bytes per stack node

Remark. This accounts for the memory for the stack
(but not the memory for strings themselves, which the client owns).

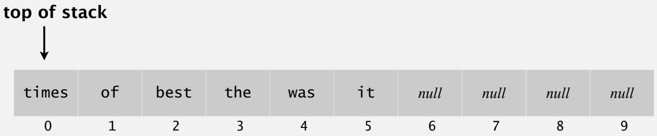
How to implement a fixed-capacity stack with an array?

A. Can't be done efficiently with an array.

B.

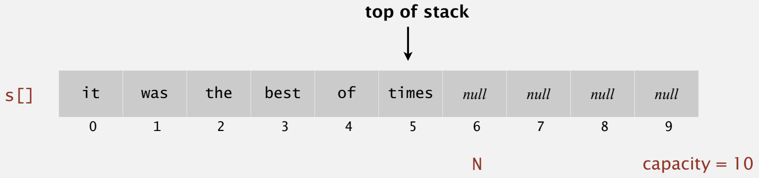


C.



Fixed-capacity stack: array implementation

- Use array `s[]` to store `N` items on stack.
- `push()`: add new item at `s[N]`.
- `pop()`: remove item from `s[N-1]`.



Defect. Stack overflows when `N` exceeds capacity. [stay tuned]

STACK: ARRAY IMPLEMENTATION

`http://dsvproject.github.io/dsvproject/code/stackArray.html`

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];  }
}
```

a cheat
(stay tuned)

use to index into array;
then increment N

decrement N;
then use to index into array

Stack considerations

Overflow and underflow.

- Underflow: throw exception if pop from an empty stack.
- Overflow: use resizing array for array implementation. [stay tuned]

Null items. We allow null items to be inserted.

Loitering. Holding a reference to an object when it is no longer needed.

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

loitering

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

this version avoids "loitering": garbage collector can reclaim memory for an object only if no outstanding references

A BETTER ARRAY IMPLEMENTATION OF STACKOFSTRINGS

1.3 Bags, Queues, and Stacks

- *stacks*
- *resizing arrays*
- *queues*
- *generics*
- *iterators*
- *applications*

Algorithms

Robert Sedgewick | Kevin Wayne

http://algs4.cs.princeton.edu

Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- `push()`: increase size of array `s[]` by `1`.
- `pop()`: decrease size of array `s[]` by `1`.

Too expensive.

- Need to copy all items to a new array, for each operation.
- Array accesses to insert first N items = $\underset{\substack{\uparrow\\ \text{1 array access}\\ \text{per push}}}{N} + \underset{\substack{\uparrow\\ \text{2(k–1) array accesses to expand to size k}\\ \text{(ignoring cost to create new array)}}}{(2 + 4 + \ldots + 2(N-1))} \sim \overset{\substack{\text{infeasible for large N}\\ \downarrow}}{N^2}$.

Challenge. Ensure that array resizing happens infrequently.

INCREASE SIZE BY 1

We start with an empty array of size 1
Array accesses (AC):

- → 1st push: 1 AC (store new item)
- → 2nd push: 1 AC + 2 AC (read-write previous item(s) to new array)
- → 3rd push: 1 AC + 4 AC
- → 4th push: 1 AC + 6 AC
- → ...
- → Nth push: 1 AC + $2(N-1)$ AC

$\sim N^2$ **array accesses to insert N items starting from the empty stack**

Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.

- `push()`: increase size of array `s[]` by `1`.
- `pop()`: decrease size of array `s[]` by `1`.

Too expensive.

- Need to copy all items to a new array, for each operation.
- Array accesses to insert first N items = $\underbrace{N}_{\text{1 array access per push}} + (\underbrace{2 + 4 + \ldots + 2(N-1)}_{\text{2(k–1) array accesses to expand to size k (ignoring cost to create new array)}}) \sim \overset{\text{infeasible for large N}}{N^2}$.

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

Q. How to grow array?

"repeated doubling"

A. If array is full, create a new array of twice the size, and copy items.

```
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;
}
```

Array accesses to insert first N = 2^i items. $\underbrace{N}_{\text{1 array access per push}} + (2 + 4 + \underbrace{8}_{\text{k array accesses to double to size k (ignoring cost to create new array)}} + \ldots + N) \sim 3N.$

Stack: resizing-array implementation

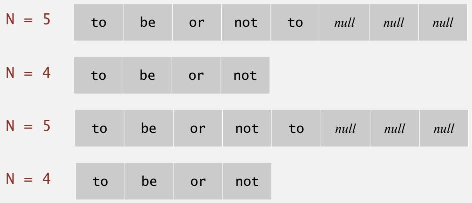
Q. How to shrink array?

First try.

- `push()`: double size of array `s[]` when array is full.
- `pop()`: halve size of array `s[]` when array is one-half full.

Too expensive in worst case.

- Consider push-pop-push-pop-... sequence when array is full.
- Each operation takes time proportional to N.



N = 5 | to | be | or | not | to | null | null | null
N = 4 | to | be | or | not
N = 5 | to | be | or | not | to | null | null | null
N = 4 | to | be | or | not

Stack: resizing-array implementation

Q. How to shrink array?

Efficient solution.

- push(): double size of array s[] when array is full.
- pop(): halve size of array s[] when array is one-quarter 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 between 25% and 100% full.

AMORTIZED RUNNING TIME

Assume implementation of ADT with operations A, B, C

How do we calculate the amortized running time of the operations of the ADT?

- → Consider the ADT to be empty
- → Consider all feasible sequences of N operations
  - → A, A, A, ...
  - → A, B, A, C, ...
  - → ...
- → calculate total running time for each of these sequences take the largest one (worst case sequence)
- → Amortized running time of ADT operations = worst-case running time of N operations / N

Stack resizing-array implementation: performance

Amortized analysis. Starting from an empty data structure, average running time per operation over a worst-case sequence of operations.

Proposition. Starting from an empty stack, any sequence of M push and pop operations takes time proportional to M.

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

doubling and halving operations

order of growth of running time for resizing stack with N items

RESIZING-ARRAY IMPLEMENTATION OF STACKOFSTRINGS ADT

Operations (push/pop/construct/resize) have

O(1) amortized running time

Stack resizing-array implementation: memory usage

Proposition. Uses between $\sim 8N$ and $\sim 32N$ bytes to represent a stack with N items.

- $\sim 8N$ when full.
- $\sim 32N$ when one-quarter full.

```
public class ResizingArrayStackOfStrings
{
   private String[] s;   ←——  8 bytes × array size
   private int N = 0;
   …
}
```

Remark. This accounts for the memory for the stack
(but not the memory for strings themselves, which the client owns).

Stack implementations: resizing array vs. linked list

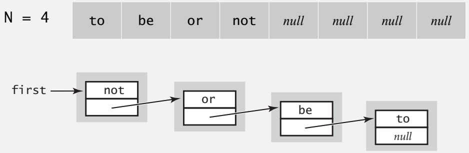
Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.

- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.

Resizing-array implementation.

- Every operation takes constant amortized time.
- Less wasted space.



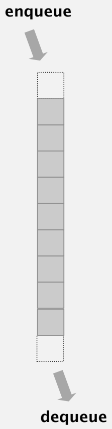
http://algs4.cs.princeton.edu

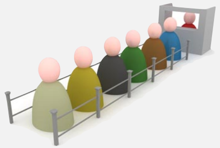
1.3 Bags, Queues, and Stacks

- *stacks*
- *resizing arrays*
- *queues*
- *generics*
- *iterators*
- *applications*

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*

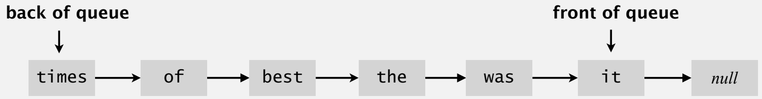




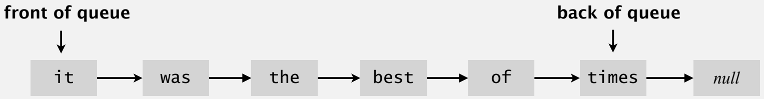
How to implement a queue with a linked list?

A. Can't be done efficiently with a singly-linked list.

B.

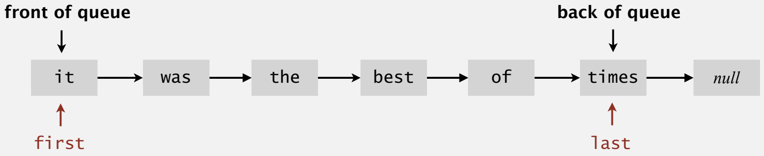


C.



Queue: linked-list implementation

- Maintain one pointer `first` to first node in a singly-linked list.
- Maintain another pointer `last` to last node.
- Dequeue from `first`.
- Enqueue after `last`.



Visualisation:

`http://www.cs.usfca.edu/~galles/visualization/QueueLL.html`

Queue dequeue: linked-list implementation

inner class

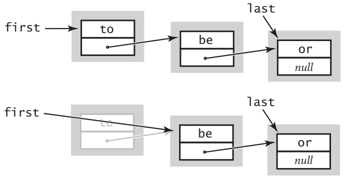
```
private class Node
{
   String item;
   Node next;
}
```

save item to return

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

delete first node

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



return saved item

```
return item;
```

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

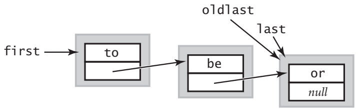
Queue enqueue: linked-list implementation

inner class

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

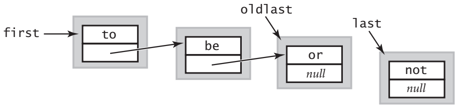
save a link to the last node

```
Node oldlast = last;
```



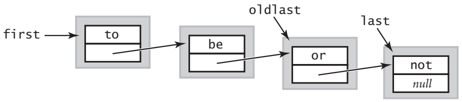
create a new node for the end

```
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

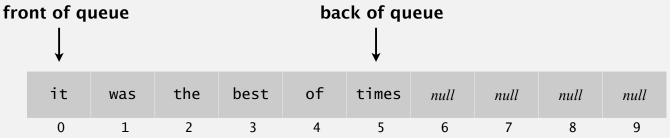
```
public class LinkedQueueOfStrings
{
   private Node first, last;

   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;
   }
}
```

special cases for empty queue

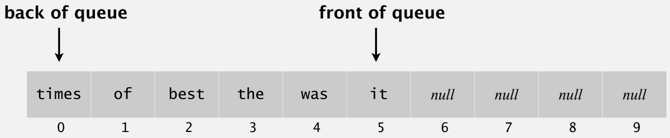
How to implement a fixed-capacity queue with an array?

A. Can't be done efficiently with an array.

B.

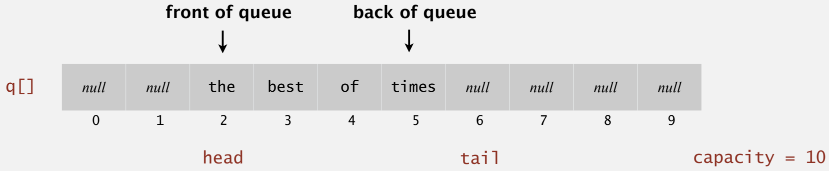


C.



Queue: resizing-array implementation

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.
- Add resizing array.



Q. How to resize?