1.3 Stacks and Queues

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

see precept
Stacks and queues

**Fundamental data types.**
- Value: *collection* of objects.
- Operations: *add, remove, iterate, test if empty.*
- Intent is clear when we add.
- Which item do we remove?

**Stack.** Examine the item most recently added.  
**Queue.** Examine the item least recently added.

LIFO = “last in first out”
FIFO = “first in first out”
Separate client and implementation via API.

**Client**: program that uses the API operations.

**Implementation**: code that implements the API operations.

**API**: operations that characterize the behavior of a data type.

**Benefits.**
- **Design**: create modular, reusable libraries.
- **Performance**: substitute faster implementations.

**Ex.** Stack, queue, bag, priority queue, symbol table, union–find, ....
1.3 STACKS AND QUEUES

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

**Warmup API.** Stack of strings data type.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackOfStrings()</td>
<td>create an empty stack</td>
</tr>
<tr>
<td>void push(String item)</td>
<td>add a new string to stack</td>
</tr>
<tr>
<td>String pop()</td>
<td>remove and return the string most recently added</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the stack empty?</td>
</tr>
<tr>
<td>int size()</td>
<td>number of strings on the stack</td>
</tr>
</tbody>
</table>

**Performance requirements.** All operations take constant time.

**Warmup client.** Reverse sequence of strings from standard input.
How to implement a stack with a singly linked list?

A. least recently added
   ↓
   it → was → the → best → of → null

B. most recently added
   ↓
   of → best → the → was → it → null

C. Both A and B.

D. Neither A nor B.
Stack: linked-list implementation

- Maintain pointer $\text{first}$ to first node in a singly linked list.
- Push new item before $\text{first}$.
- Pop item from $\text{first}$. 

![Diagram showing a stack implementation using a linked list]
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;
}

**create a new node for the beginning**
first = new Node();

**set the instance variables in the new node**
first.item = "not";
first.next = oldfirst;

**save a link to the list**
Node oldfirst = first;
Stack: linked-list implementation

```java
public class LinkedStackOfStrings {
    private Node first = null;

    private class Node {
        private String item;
        private 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;
    }
}
```
Stack: linked-list implementation performance

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

**Proposition.** A stack with $n$ items uses $\sim 40n$ bytes.

**Remark.** This counts the memory for the stack (but not the memory for the strings themselves, which the client owns).
How to implement a fixed-capacity stack with an array?

A. least recently added

<table>
<thead>
<tr>
<th>it</th>
<th>was</th>
<th>the</th>
<th>best</th>
<th>of</th>
<th>times</th>
<th>null</th>
<th>null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

B. most recently added

<table>
<thead>
<tr>
<th>times</th>
<th>of</th>
<th>best</th>
<th>the</th>
<th>was</th>
<th>it</th>
<th>null</th>
<th>null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

C. Both A and B.

D. Neither A nor B.
Fixed-capacity stack: array implementation

- Use array \( s[] \) to store \( n \) items on stack.
- \texttt{push}(): add new item at \( s[n] \).
- \texttt{pop}(): remove item from \( s[n-1] \).

\begin{center}
\begin{tabular}{lcccccccc}
 & \texttt{it} & \texttt{was} & \texttt{the} & \texttt{best} & \texttt{of} & \texttt{times} & \texttt{null} & \texttt{null} & \texttt{null} & \texttt{null} \\
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{tabular}
\end{center}

\textbf{Defect.} Stack overflows when \( n \) exceeds capacity. [stay tuned]
Fixed-capacity stack: array implementation

```java
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];
    }
}
```

- Use `n` as index into array; then increment `n`.
- Decrement `n`; then use as index into array.
- A cheat (stay tuned)
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 added.

Duplicate items. We allow an item to be added more than once.

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

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

loitering

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

no loitering
1.3 **Stacks and Queues**

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
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 add first `n` items = `n + (2 + 4 + ... + 2(n − 1)) \sim n^2`.

**Challenge.** Ensure that array resizing happens infrequently.
Stack: resizing-array implementation

Q. How to grow array?
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 add first \( n = 2^i \) items. \( n + (2 + 4 + 8 + \ldots + n) \sim 3n \).
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`.

<table>
<thead>
<tr>
<th>Status</th>
<th>to</th>
<th>be</th>
<th>or</th>
<th>not</th>
<th>push(&quot;to&quot;)</th>
<th>pop()</th>
<th>push(&quot;be&quot;)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>null</code></td>
<td><code>null</code></td>
<td><code>null</code></td>
<td><code>null</code></td>
<td><code>null</code></td>
<td>to</td>
<td>be</td>
<td>or</td>
</tr>
</tbody>
</table>
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 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 \( m \).

<table>
<thead>
<tr>
<th></th>
<th>typical</th>
<th>worst</th>
<th>amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>construct</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>push</td>
<td>1</td>
<td>( n )</td>
<td>1</td>
</tr>
<tr>
<td>pop</td>
<td>1</td>
<td>( n )</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

order of growth of running time for resizing array stack with \( n \) items
Stack resizing-array implementation: memory usage

**Proposition.** A ResizingArrayStackOfStrings uses between $\sim 8n$ and $\sim 32n$ bytes of memory for a stack with $n$ items.

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

**Remark.** This counts the memory for the stack (but not the memory for the 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.

\[
\begin{array}{cccccccc}
\text{null} & \text{null} & \text{null} & \text{null} \\
\end{array}
\]

```
Inserting a new node at the beginning of a linked list
```

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

\[
\begin{array}{cccccccc}
\text{null} & \text{null} & \text{null} & \text{null} \\
\end{array}
\]

```
set the instance variables in the new node
```

```java
first.item = "not";
```

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

```
to
be
or
not
```
1.3 **Stacks and Queues**

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Queue API

Warmup API. Queue of strings data type.

```java
public class QueueOfStrings {
    QueueOfStrings() // create an empty queue
    void enqueue(String item) // add a new string to 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
}
```

Performance requirements. All operations take constant time.
How to implement a queue with a singly linked linked list?

A. most recently added

```
times -> of -> best -> the -> was -> it -> null
```

B. least recently added

```
it -> was -> the -> best -> of -> times -> null
```

C. Both A and B.

D. Neither A nor B.
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`.

```
least recently added
↓
  it → was → the → best → of → times → null
  ↑ first
most recently added
↓
  null → times → of → best → the → was → it
  ↑ last
```
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;

remark

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

A. least recently added

B. most recently added

C. Both A and B.

D. Neither A nor B.
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.

Q. How to resize?
**Queue With Two Stacks**

**Problem.** Implement a queue with two stacks so that:

- Each queue op uses a constant amortized number of stack ops.
- At most constant extra memory (besides two stacks).

**Applications.**

- Job interview.
- Implement an immutable or persistent queue.
- Implement a queue in a (purely) functional programming language.

[Haskell] (https://www.haskell.org/)
[Lisp] (https://www.lisp.org/)
[OCaml] (https://ocaml.org/)
1.3 Stacks and Queues

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfApples, StackOfOranges, ....

Solution in Java: generics.

```
Stack<Apple> stack = new Stack<Apple>();
Apple apple = new Apple();
Orange orange = new Orange();
stack.push(apple);
stack.push(orange); // compile-time error
...}
```
Generic stack: linked-list implementation

```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;
    }
}
```

Stack of strings (linked list)

```java
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 stack (linked list)
Generic stack: array implementation

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]; }
}

stack of strings (fixed-length array)

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 (fixed-length array)

@#$! generic array creation not allowed in Java
Generic stack: array implementation

**Stack of strings (fixed-length array)**

```java
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];
    }
}
```

**Generic stack (fixed-length array)**

```java
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];
    }
}
```

the ugly cast
Unchecked cast

Q. Why does Java require a cast (or reflection)?
Short answer. Backward compatibility.
Long answer. Need to learn about type erasure and covariant arrays.
Which of the following is the correct way to declare and initialize an empty stack of integers?

A. Stack stack = new Stack<int>();

B. Stack<int> stack = new Stack();

C. Stack<int> stack = new Stack<int>();

D. None of the above.
Generic data types: autoboxing and unboxing

Q. What to do about primitive types?

Wrapper type.
- Each primitive type has a wrapper object type.
- Ex: Integer is wrapper type for int.

Autoboxing. Automatic cast from primitive type to wrapper type.
Unboxing. Automatic cast from wrapper type to primitive type.

Bottom line. Client code can use generic stack for any type of data.
1.3 **Stacks and Queues**

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

*see precept*
Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.

resizing-array representation

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s[]</td>
<td>it</td>
<td>was</td>
<td>the</td>
<td>best</td>
<td>of</td>
<td>times</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

linked-list representation

first

\[ \text{times} \rightarrow \text{of} \rightarrow \text{best} \rightarrow \text{the} \rightarrow \text{was} \rightarrow \text{it} \rightarrow \text{null} \]

Java solution. Use a foreach loop.
Java provides elegant syntax for iteration over collections.

**“foreach” loop (shorthand)**

```
Stack<String> stack;
...
for (String s : stack)
...
```

**equivalent code (longhand)**

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

To make user-defined collection support foreach loop:

- Data type must have a method named `iterator()`.
- The `iterator()` method returns an object that has two core method.
  - `hasNext()` method returns `false` when there are no more items
  - `next()` method returns the next item in the collection
Iterators

To support foreach loops, Java provides two interfaces.

- **Iterator interface**: `next()` and `hasNext()` methods.
- **Iterable interface**: `iterator()` method that returns an `Iterator`.
- Both should be used with generics.

```java
java.util.Iterator interface

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

```java
java.lang.Iterable interface

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

Type safety.

- Implementation must use these interfaces to support foreach loop.
- Client program won’t compile unless implementation do.
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;
        }
    }
}
Stack iterator: array implementation

```java
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]; }
    }
}
```

```
<table>
<thead>
<tr>
<th>s[]</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>it</td>
<td>was</td>
<td>the</td>
<td>best</td>
<td>of</td>
<td>times</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```
1.3 **Stacks and Queues**

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

- Java virtual machine.
- Parsing in a compiler.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.
- ...
Queue applications

Familiar applications.

- Spotify playlist.
- Data buffers (iPod, TiVo, sound card, streaming video, ...).
- Asynchronous data transfer (file IO, pipes, sockets, ...).
- Dispensing requests on a shared resource (printer, processor, ...).

Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.
Java collections library

**List interface.** `java.util.List` is API for a sequence of items.

```java
public interface List<Item> extends Iterable<Item>
{
    List() create an empty list
    boolean isEmpty() is the list empty?
    int size() number of items
    void add(Item item) add item to the end
    Iterator<Item> iterator() iterator over all items in the list
    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?
    ...
}
```

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

*Caveat:* not all operations are efficient!
Java collections library

**java.util.Stack.**

- Supports `push()`, `pop()`, and iteration.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.

---

**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 library

`java.util.Stack`.  
- Supports `push()`, `pop()`, and iteration.  
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.

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

**Best practices.** Use our Stack and Queue for stacks and queues; use `java.util.ArrayList` or `java.util.LinkedList` when appropriate.
Generate random open sites in an $n$-by-$n$ percolation system.

- Jenny: pick $(row, col)$ at random; if already open, repeat. Takes $\sim c_1 n^2$ seconds.
- Kenny: create a `java.util.ArrayList` of $n^2$ closed sites. Pick an index at random and delete. Takes $\sim c_2 n^4$ seconds.

**Lesson.** Don’t use a library until you understand its API!

**This course.** Can’t use a library until we’ve implemented it in class.