1.3 **Stacks and Queues**

- stacks
- resizing arrays
- queues
- generics
- iterators see precept
- applications
Both are *collections* of objects.
Both support *add*, *remove*, *iterate*, test if empty.

Intent is clear when we add.
Difference between stack and queue: which item to remove.

**Stack.** Remove the item most recently added.  
LIFO = “last in first out”

**Queue.** Remove the item least recently added.  
FIFO = “first in first out”
Client, implementation, API

Separate client and implementation via API.

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

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

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

**Benefits.**

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

**Ex.** Stack, queue, bag, priority queue, symbol table, union-find, ....
Layers in a computer system

Program
Libraries
Programming language
Operating system
Hardware

API
Java libraries include stacks and queues but in this course we’ll prefer our own implementations
1.3 Stacks and Queues

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

https://algs4.cs.princeton.edu
Stack API

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

```java
public class StackOfStrings {
    StackOfStrings() { create an empty stack }
    void push(String item) { add a new string to 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 }
}
```

**Performance requirements.** All operations must take constant time.
Either data type can be implemented using either data structure

<table>
<thead>
<tr>
<th></th>
<th>Linked list</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Queue</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Client ➔ API ➔ Implementation
How to implement a stack with a singly linked list?
Recall: we only keep track of the head of the list.

A. "first" (least recently added)

  it → was → the → best → of → null

B. "first" (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 \texttt{first} to first node in a singly linked list.
- Push new item before \texttt{first}.
- Pop item from \texttt{first}.

```

```

```
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 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 = item;
first.next = oldfirst;
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 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 \texttt{s[]} to store \( n \) items on stack.
- \texttt{push()}: add new item at \texttt{s}[n].
- \texttt{pop()}: remove item from \texttt{s}[n-1].

\begin{center}
\begin{tabular}{cccccccc}
\textbf{least recently added} & & & & & & & \\
\texttt{s[]} & it & was & the & best & of & times & null \\
\hline
0 & 1 & 2 & 3 & 4 & 5 & 6 & null \\
1 & 2 & 3 & 4 & 5 & 6 & 7 & null \\
2 & 3 & 4 & 5 & 6 & 7 & 8 & null \\
3 & 4 & 5 & 6 & 7 & 8 & 9 & null \\
\end{tabular}
\end{center}

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;
        n++;
    }

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

Overflow and underflow.
- Underflow: throw exception if \texttt{pop()} from an empty stack.
- Overflow: use “resizing array” for array implementation. [stay tuned]

Null items. We allow \texttt{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()
{
    n--;  
    return s[n];
}
```

\textit{loitering}

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

\textit{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 $= n + (2 + 4 + \ldots + 2(n-1)) \sim n^2$.

Challenge. Ensure that array resizing happens infrequently.
Q. How to grow array?
A. If array is full, create a new array of twice the size, and copy items.

```java
public ResizingArrayStackOfStrings()
{  s = new String[1]; }

public void push(String item)
{
    if (n == s.length) resize(2 * s.length);
    n++;
    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. 

\[
\text{Array accesses} = n + (2 + 4 + 8 + \ldots + n) \sim 3n .
\]
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-push-push-push-…` sequence when array is full.
- Each operation takes time proportional to `n`.

<table>
<thead>
<tr>
<th></th>
<th>full</th>
<th>to</th>
<th>be</th>
<th>or</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(&quot;to&quot;)</td>
<td></td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
</tr>
<tr>
<td>pop()</td>
<td></td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
</tr>
<tr>
<td>push(&quot;be&quot;)</td>
<td></td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
</tr>
</tbody>
</table>
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.

```java
public String pop()
{
    n--;  
    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.
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 $m$.

<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

doubling and halving operations
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).
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 \textit{worst case}.
- Uses extra time and space to deal with the links.

Resizing-array implementation.
- Every operation takes constant \textit{amortized} time.
- Less wasted space.
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();
    void enqueue(String item);
    String dequeue();
    boolean isEmpty();
    int size();
}
```

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

A. Most recently added
   \[\text{times} \rightarrow \text{of} \rightarrow \text{best} \rightarrow \text{the} \rightarrow \text{was} \rightarrow \text{it} \rightarrow \text{null}\]

B. Least recently added
   \[\text{it} \rightarrow \text{was} \rightarrow \text{the} \rightarrow \text{best} \rightarrow \text{of} \rightarrow \text{times} \rightarrow \text{null}\]

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`.

```
first     (least recently added)
  ↓       ↓
  it → was → the → best → of → times → null

last    (most recently added)```
Queue dequeue: linked-list implementation

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

```java
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;
```
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

```
<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.
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?
1.3 Stacks and Queues

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

**We implemented:** StackOfStrings.

**We also want:** StackOfURLs, StackOfInts, StackOfApples, StackOfVans, ....

**Solution in Java:** generics.

```java
Stack<Apple> stack = new Stack<Apple>();
Apple apple = new Apple();
stack.push(apple);
Van van = new Van();
stack.push(van);
...
```

*type parameter*  
(use syntax both to specify type and to call constructor)  

*compile-time error*
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)
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];  }
}

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 array creation not allowed in Java
Generic 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];  }
}

stack of strings (fixed-length array)

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];  }
}

generic stack (fixed-length array)

the ugly cast
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 \textit{wrapper} object type.
- Ex: \texttt{Integer} is wrapper type for \texttt{int}.

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

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

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

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

[https://algs4.cs.princeton.edu](https://algs4.cs.princeton.edu)
Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.

### Iteration

#### resizing-array representation

```
 s[]    i    n
<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>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>
<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
  ↓
times → of → best → the → was → it → null
current
  ↓
```

Java solution. Use a `foreach` loop.
Foreach loop

Java provides elegant syntax for iteration over collections.

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.
  – the `hasNext()` methods returns `false` when there are no more items
  – the `next()` method returns the next item in the collection

```
“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();
    ...
}
```
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.util.Iterator interface

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

```
java.lang.Iterable interface

public interface Iterable
{
    Iterator iterator();
}
```

**Type safety.**

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

```java
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;
        }
    }
}
```

- `first`: initial node
- `current`: current node
- `times`: number of times
- `of`: context of
- `best`: best
- `the`: context of
- `was`: past
- `it`: context of
- `null`: null
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]; }
    }

    }

    s[]

    i    n

    0 1 2 3 4 5 6 7 8 9

    it was the best of times null null null null null
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.
Unchecked cast

% javac -Xlint:unchecked FixedCapacityStack.java
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
   s = (Item[]) new Object[capacity];  
      ^
   required: Item[]
   found:    Object[]
   where Item is a type-variable:
      Item extends Object declared in class FixedCapacityStack
1 warning

Q. Why does Java require a cast (or reflection)?

Short answer. Backward compatibility.

Long answer. Need to learn about type erasure and covariant arrays.