12. Stacks and Queues
12. Stacks and Queues

- APIs
- Clients
- Strawman implementation
- Linked lists
- Implementations
Data types and data structures

Data types
- Set of values.
- Set of operations on those values.
- Some are built in to Java: int, double, String, ...
- Most are not: Complex, Picture, Charge, ...

Data structures
- Represent data.
- Represent relationships among data.
- Some are built in to Java: 1D arrays, 2D arrays, ...
- Most are not: linked list, circular list, tree, ...

Design challenge for every data type: Which data structure to use?
- Resource 1: How much memory is needed?
- Resource 2: How much time do data-type methods use?
Stack and Queue APIs

A collection is an ADT whose values are a multiset of items, all of the same type.

Two fundamental collection ADTs differ in just a detail of the specification of their operations.

**Stack operations**
- Add an item to the collection.
- Remove and return the item *most* recently added (LIFO).
- Test if the collection is empty.
- Return the size of the collection.

**Queue operations**
- Add an item to the collection.
- Remove and return the item *least* recently added (FIFO).
- Test if the collection is empty.
- Return the size of the collection.

Stacks and queues both arise naturally in countless applications.

A key characteristic. **No limit** on the size of the collection.
Example of stack operations

**Push.** Add an item to the collection.

**Pop.** Remove and return the item *most* recently added.
Example of queue operations

**Enqueue.** Add an item to the collection.

**Dequeue.** Remove and return the item *least* recently added.
Parameterized data types

**Goal.** Simple, safe, and clear client code for collections of any type of data.

Java approach: Parameterized data types (generics)
- Use placeholder type name in definition.
- Substitute concrete type for placeholder in clients.

---

**Stack API**

```java
public class Stack<Item> {
  Stack<Item>() // create a stack of items, all of type Item
  void push(Item item) // add item to stack
  Item pop() // remove and return the item most recently pushed
  boolean isEmpty() // is the stack empty?
  int size() // # of objects on the stack
}
```

**Queue API**

```java
public class Queue<Item> {
  Queue<Item>() // create a queue of items, all of type Item
  void enqueue(Item item) // add item to queue
  Item dequeue() // remove and return the item least recently enqueued
  boolean isEmpty() // is the queue empty?
  int size() // # of objects on the queue
}
```
Performance specifications

**Challenge.** Provide guarantees on performance.

**Goal.** Simple, safe, clear, and *efficient* client code.

**Performance specifications**

- All operations are constant-time.
- Memory use is linear in the size of the collection, when it is nonempty.
- No limits within the code on the collection size.

**Java.** Any implementation of the API implements the stack/queue abstractions.

**RS+KW.** Implementations that do not meet performance specs *do not* implement the abstractions.

Typically required for client code to be *scalable*. 
12. Stacks and Queues

- APIs
- Clients
- Strawman implementation
- Linked lists
- Implementations
Stack and queue applications

Queues
- First-come-first-served resource allocation.
- Asynchronous data transfer (StdIn, StdOut).
- Dispensing requests on a shared resource.
- Simulations of the real world.

Stacks
- Last-come-first-served resource allocation.
- Function calls in programming languages.
- Basic mechanism in interpreters, compilers.
- Fundamental abstraction in computing.
Queue client example: Read all strings from StdIn into an array

**Challenge**
- Can’t store strings in array before creating the array.
- Can’t create the array without knowing how many strings are in the input stream.
- Can’t know how many strings are in the input stream without reading them all.

**Solution:** Use a Queue<String>.

```java
public class QEx {
    public static String[] readAllStrings() {
        /* See next slide. */
    }

    public static void main(String[] args) {
        String[] words = readAllStrings();
        for (int i = 0; i < words.length; i++)
            StdOut.println(words[i]);
    }
}
```

% java QEx < moby.txt
moby
dick
herman melville
call me ishmael some years ago never mind how long precisely having little or no money ...
Queue client example: Read all strings from StdIn into an array

Solution: Use a Queue<String>.
- Store strings in the queue.
- Get the size when all have been read from StdIn.
- Create an array of that size.
- Copy the strings into the array.

```java
public class QEx
{
    public static String[] readAllStrings()
    {
        Queue<String> q = new Queue<String>();
        while (!StdIn.isEmpty())
            q.enqueue(StdIn.readString());
        int N = q.size();
        String[] words = new String[N];
        for (int i = 0; i < N; i++)
            words[i] = q.dequeue();
        return words;
    }
    public static void main(String[] args)
    {
        String[] words = readAllStrings();
        for (int i = 0; i < words.length; i++)
            StdOut.println(words[i]);
    }
}
```
Stack example: "Back" button in a browser

Typical scenario
- Visit a page.
- Click a link to another page.
- Click a link to another page.
- Click "back" button.
- Click "back" button.
- Click "back" button.
Autoboxing

**Challenge.** Use a *primitive* type in a parameterized ADT.

**Wrapper types**
- Each primitive type has a wrapper reference type.
- Wrapper type has larger set of operations than primitive type. Example: `Integer.parseInt()`.
- Instances of wrapper types are objects.
- Wrapper type can be used in a parameterized ADT.

<table>
<thead>
<tr>
<th>primitive type</th>
<th>wrapper type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

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

**Auto-unboxing.** Automatic cast from wrapper type to primitive type.

**Simple client code (no casts)**

```java
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);  // Autobox  (int -> Integer)
int a = stack.pop();  // Auto-unbox (Integer -> int)
```
Stack client example: Postfix expression evaluation

**Infix.** Standard way of writing arithmetic expressions, using parentheses for precedence.

Example. \((1 + ((2 + 3)(4 * 5))) = (1 + (5 * 20)) = 101\)

**Postfix.** Write operator *after* operands (instead of in between them).

Example. 1 2 3 + 4 5 * * +  

also called "reverse Polish" notation (RPN)

**Remarkable fact.** No parentheses are needed!

There is only one way to parenthesize a postfix expression.

\[
\begin{align*}
1 & \quad 2 \quad 3 \quad + \quad 4 \quad 5 \quad * \quad * \quad + \\
1 & \quad (2 + 3) \quad 4 \quad 5 \quad * \quad * \quad + \\
1 & \quad ((2+3) \quad *(4 \quad 5)) + \\
(1 + ((2 + 3) \quad *(4 \quad 5)))
\end{align*}
\]

find first operator, convert to infix, enclose in ()
iterate, treating subexpressions in parentheses as atomic

**Next.** With a stack, postfix expressions are easy to evaluate.
Postfix arithmetic expression evaluation

Algorithm
- While input stream is nonempty, read a token.
- Value: Push onto the stack.
- Operator: Pop operand(s), apply operator, push the result.

\[
\begin{array}{cccccc}
1 & 2 & 3 & + & 4 & 5 \quad * \quad * \quad + \\
\end{array}
\]

\[
= 5 \quad = 20 \quad 100 \quad 101
\]
Stack client example: Postfix expression evaluation

```java
public class Postfix {
    public static void main(String[] args) {
        Stack<Double> stack = new Stack<Double>();
        while (!StdIn.isEmpty()) {
            String token = StdIn.readString();
            if (token.equals("*"))
                stack.push(stack.pop() * stack.pop());
            else if (token.equals("+"))
                stack.push(stack.pop() + stack.pop());
            else if (token.equals("-"))
                stack.push(-stack.pop() + stack.pop());
            else if (token.equals("/"))
                stack.push((1.0 / stack.pop()) * stack.pop());
            else if (token.equals("sqrt"))
                stack.push(Math.sqrt(stack.pop()));
            else
                stack.push(Double.parseDouble(token));
        }
        StdOut.println(stack.pop());
    }
}
```

% java Postfix
1 2 3 + 4 5 * * +
101.0

% java Postfix
1 5 sqrt + 2 /
1.618033988749895

1 + \sqrt{5} / 2

Perspective
- Easy to add operators of all sorts.
- Can do infix with two stacks (see text).
- Could output machine language code.
- Indicative of how Java compiler works.
Real-world stack application: PostScript

PostScript (Warnock-Geschke, 1980s): A turtle with a stack.
- Postfix program code (push literals; functions pop arguments).
- Add commands to drive virtual graphics machine.
- Add loops, conditionals, functions, types, fonts, strings....

PostScript code

\[
\begin{array}{c}
100 & 100 & \text{moveto} \\
100 & 300 & \text{lineto} \\
300 & 300 & \text{lineto} \\
300 & 100 & \text{lineto} \\
\text{stroke}
\end{array}
\]

A simple virtual machine, but not a toy
- Easy to specify published page.
- Easy to implement on various specific printers.
- Revolutionized world of publishing.

Another stack machine: The JVM (Java Virtual Machine)!
Image sources

http://upload.wikimedia.org/wikipedia/commons/2/20/Cars_in_queue_to_enter_Gibraltar_from_Spain.jpg
12. Stacks and Queues

• APIs
• Clients
• Strawman implementation
• Linked lists
• Implementations
Strawman ADT for pushdown stacks

Warmup: simplify the ADT
- Implement only for items of type String.
- Have client provide a stack capacity in the constructor.

values

Strawman API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StrawStack(int max)</td>
<td>create a stack of capacity max</td>
</tr>
<tr>
<td>void push(String item)</td>
<td>add item to stack</td>
</tr>
<tr>
<td>String pop()</td>
<td>return the string most recently pushed</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>

Rationale. Allows us to represent the collection with an array of strings.
Strawman implementation: Instance variables and constructor

Data structure choice. Use an array to hold the collection.

```java
public class StrawStack {
    private String[] a;
    private int N = 0;

    public StrawStack(int max)
    { a = new String[max]; }

    ...}
```

"upside down" representation of
Strawman stack implementation: Test client

```
public static void main(String[] args)
{
    int max = Integer.parseInt(args[0]);
    StrawStack stack = new StrawStack(max);
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (item.equals("-"))
            StdOut.print(stack.pop());
        else
            stack.push(item);
    }
    StdOut.println();
}
```

What we expect, once the implementation is done.

What to be or not to be -- that -- is not.

% java StrawStack 20 < tobe.txt
% more tobe.txt

to be or not to be -- that -- is not that or be
**Pop quiz 1 on stacks**

**Q.** Can we always insert `pop()` commands to make items come out in sorted order?

| Example 1. | 6 5 4 3 2 1 - - - - - - |
| Example 2. | 1 - 2 - 3 - 4 - 5 - 6 - |
| Example 3. | 4 1 - 3 2 - - - 6 5 - - |

```
1
2 3 4
5 6
```

```
1

2

3

3

3

6

5

6
```
Methods define data-type operations (implement APIs).

```java
public class StrawStack {
    ...
    public boolean isEmpty() {
        return (N == 0);
    }

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

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

    public int size() {
        return N;
    }
    ...
}
```

Methods

- `isEmpty()`: Checks if the stack is empty.
- `push(String item)`: Puts an item onto the stack.
- `pop()`: Removes an item from the stack.
- `size()`: Returns the number of items in the stack.

All constant-time one-liners!
Strawman pushdown stack implementation

```java
public class StrawStack {

  private String[] a;
  private int N = 0;

  public StrawStack(int max) {
    a = new String[max];
  }

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

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

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

  public int size() {
    return N;
  }

  public static void main(String[] args) {
    int max = Integer.parseInt(args[0]);
    StrawStack stack = new StrawStack(max);
    while (!StdIn.isEmpty()) {
      String item = StdIn.readString();
      if (item.equals("-"))
        StdOut.print(stack.pop() + " ");
      else
        stack.push(item);
    }
    StdOut.println();
  }
}
```

instance variables
class
methods
test client

% more tobe.txt
to be or not to - be -- that - -- is
% java StrawStack 20 < tobe.txt
to be not that or be
Trace of strawman stack implementation (array representation)

<table>
<thead>
<tr>
<th>push</th>
<th>to</th>
<th>be</th>
<th>or</th>
<th>not</th>
<th>to</th>
<th>-</th>
<th>be</th>
<th>-</th>
<th>-</th>
<th>that</th>
<th>-</th>
<th>-</th>
<th>-</th>
<th>is</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| stack contents after operation |

| a[0] | to | to | to | to | to | to | to | to | to | to | to | to | to | to | is |
| a[1] |    | be | be | be | be | be | be | be | be | be | be | be | be | be | is |
| a[2] |    | or | or | or | or | or | or | or | or | or | or | or | or | or |
| a[3] |    | not | not | not | not | not | not | not | not | not | not | not | not | not |
| a[4] |    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[5] |    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[6] |    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[7] |    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[8] |    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[9] |    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[10]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[11]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[12]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[13]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[14]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[15]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[16]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[17]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[18]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |
| a[19]|    |              |              |              |              |              |              |              |              |              |              |              |              |              |

Significant wasted space when stack size is not near the capacity (typical).
Benchmarking the strawman stack implementation

StrawStack implements a *fixed-capacity collection that behaves like a stack* if the data fits.

It does *not* implement the stack API or meet the performance specifications.

<table>
<thead>
<tr>
<th>Stack API</th>
<th>StrawStack requires client to provide capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public class Stack&lt;Item&gt;</code></td>
<td><img src="Image" alt="Circle" /> <img src="Image" alt="Cross" /></td>
</tr>
<tr>
<td><code>Stack&lt;Item&gt;()</code></td>
<td><em>create a stack of items, all of type Item</em> <img src="Image" alt="Cross" /></td>
</tr>
<tr>
<td><code>void push(Item item)</code></td>
<td><em>add item to stack</em></td>
</tr>
<tr>
<td><code>Item pop()</code></td>
<td><em>remove and return the item most recently pushed</em> <img src="Image" alt="Cross" /></td>
</tr>
<tr>
<td><code>boolean isEmpty()</code></td>
<td><em>is the stack empty?</em></td>
</tr>
<tr>
<td><code>int size()</code></td>
<td><em># of items on the stack</em></td>
</tr>
</tbody>
</table>

**StrawStack works only for strings**

**Performance specifications**

- All operations are constant-time. ✓
- Memory use is linear in the size of the collection, when it is nonempty. ![Cross](Image)
- No limits within the code on the collection size. ![Cross](Image)

Nice try, but need a new *data structure.*
12. Stacks and Queues

- APIs
- Clients
- Strawman implementation
- Linked lists
- Implementations
Data structures: sequential vs. linked

**Sequential data structure**
- Put objects next to one another.
- Machine: consecutive memory cells.
- Java: array of objects.
- Fixed size, arbitrary access. \( i \)th element

**Linked data structure**
- Associate with each object a link to another one.
- Machine: link is memory address of next object.
- Java: link is reference to next object.
- Variable size, sequential access. next element
- Overlooked by novice programmers.
- Flexible, widely used method for organizing data.
Simplest singly-linked data structure: linked list

Linked list
- A recursive data structure.
- **Def.** A *linked list* is null or a reference to a *node*.
- **Def.** A *node* is a data type that contains a reference to a node.
- Unwind recursion: A linked list is a sequence of nodes.

Representation
- Use a private *nested class* **Node** to implement the node abstraction.
- For simplicity, start with nodes having two values: a **String** and a **Node**.

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

A linked list

```
first: "Alice" → "Bob" → "Carol" → null
```

- **item**:
- **next**:
Singly-linked data structures

Even with just one link (○→) a wide variety of data structures are possible.

Linked list (this lecture)

Circular list (TSP)

Tree

Rho

General case

Multiply linked structures: many more possibilities!

From the point of view of a particular object, all of these structures look the same.
Building a linked list

Node third = new Node();
third.item = "Carol";
third.next = null;

Node second = new Node();
second.item = "Bob";
second.next = third;

Node first = new Node();
first.item = "Alice";
first.next = second;
List processing code

Standard operations for processing data structured as a singly-linked list
- Add a node at the beginning.
- Remove and return the node at the beginning.
- Add a node at the end (requires a reference to the last node).
- Traverse the list (visit every node, in sequence).

An operation that calls for a *doubly*-linked list (slightly beyond our scope)
- Remove and return the node at the end.
**List processing code: Remove and return the first item**

**Goal.** Remove and return the first item in a linked list `first`.

```java
item = first.item;
first = first.next;
return item;
```

![Diagram](image)
**List processing code: Add a new node at the beginning**

**Goal.** Add item to a linked list first.

```java
Node second = first;

first = new Node();

first.item = item;
first.next = second;
```

- **first**: 
  - "Alice" → "Bob" → "Carol" → null

- **second**: 
  - "Alice" → "Bob" → "Carol" → null

- **item**: "Dave"
**List processing code: Traverse a list**

**Goal.** Visit every node on a linked list **first**.

```java
Node x = first;
while (x != null)
{
    StdOut.println(x.item);
    x = x.next;
}
```

Diagram:

```
first → "Alice" → "Bob" → "Carol" → null
```

```
StdOut
Alice
Bob
Carol
```
Pop quiz 1 on linked lists

Q. What is the effect of the following code (not-so-easy question)?

```java
... 
Node list = null;
while (!StdIn.isEmpty())
{
    Node old = list;
    list = new Node();
    list.item = StdIn.readString();
    list.next = old;
}
for (Node t = list; t != null; t = t.next)
    StdOut.println(t.item);
... 
```
Pop quiz 2 on stacks

Q. Give code that uses a stack to print the strings from StdIn on StdOut, in reverse order.
Pop quiz 2 on linked lists

Q. What is the effect of the following code (not-so-easy question)?

```java
...  
Node list = new Node();  
list.item = StdIn.readString();  
Node last = list;  
while (!StdIn.isEmpty())  
{  
    last.next = new Node();  
    last = last.next;  
    last.item = StdIn.readString();  
}  
...  
```
14. Stacks and Queues

- APIs
- Clients
- Strawman implementation
- Linked lists
- Implementations
Data structure choice. Use a linked list to hold the collection.

```java
public class Stack<Item>
{
    private Node first = null;
    private int N = 0;

    private class Node
    {
        private Item item;
        private Node next;
    }
    ...
}
```

Objects on stack

Annoying exception (not a problem here).
Can't declare an array of Item objects (don't ask why).
Need cast: Item[] a = (Item[]) new Object[N]
public static void main(String[] args) {
    Stack<String> stack = new Stack<String>();
    while (!StdIn.isEmpty()) {
        String item = StdIn.readString();
        if (item.equals("-"))
            System.out.println(stack.pop() + " ");
        else
            stack.push(item);
    }
    StdOut.println();
}
Stack implementation: Methods

**Methods** define data-type operations (implement the API).

```java
public class Stack<Item>
{
   ...
    public boolean isEmpty()
    { return first == null; }
    public void push(Item item)
    {
        Node second = first;
        first = new Node();
        first.item = item;
        first.next = second;
        N++;
    }
    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        N--;
        return item;
    }
    public int size()
    { return N; }
    ...
}
```

might also use N == 0

add a new node to the beginning of the list

remove and return first item on list

instance variable

local variable in push()
public class Stack<Item> {
    private Node first = null;
    private int N = 0;

    private class Node {
        private Item item;
        private Node next;
    }

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

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

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

    public int size() {
        return N;
    }

    public static void main(String[] args) {
        // See earlier slide
    }
}
Trace of stack implementation (linked list representation)

<table>
<thead>
<tr>
<th>push</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>to</td>
<td></td>
</tr>
<tr>
<td>be</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td></td>
</tr>
<tr>
<td>- to</td>
<td></td>
</tr>
<tr>
<td>be</td>
<td></td>
</tr>
<tr>
<td>- be</td>
<td></td>
</tr>
<tr>
<td>- not</td>
<td></td>
</tr>
<tr>
<td>that</td>
<td></td>
</tr>
<tr>
<td>- that</td>
<td></td>
</tr>
<tr>
<td>- or</td>
<td></td>
</tr>
<tr>
<td>- be</td>
<td></td>
</tr>
<tr>
<td>is</td>
<td></td>
</tr>
</tbody>
</table>

**Push to the beginning**

- to •
- be •
- or •
- not •
- to •
- not •
- or •
- be •
- to •
- not •
- or •
- be •
- to •
- not •
- or •
- be •
- to •
- be •
- not •
- or •
- be •
- to •
- be •
- not •
- or •
- be •
- to •
- be •
- to •

**Pop from the beginning**

- is •
- to •
Benchmarking the stack implementation

Stack implements the stack abstraction.

It *does* implement the API and meet the performance specifications.

### Stack API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack&lt;Item&gt;()</td>
<td>create a stack of items, all of type Item</td>
</tr>
<tr>
<td>void push(Item item)</td>
<td>add item to stack</td>
</tr>
<tr>
<td>Item pop()</td>
<td>remove and return the item most recently pushed</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the stack empty?</td>
</tr>
<tr>
<td>int size()</td>
<td># of items on the stack</td>
</tr>
</tbody>
</table>

- All operations are constant-time. ✔
- Memory use is linear in the size of the collection, when it is nonempty. ✔
- No limits within the code on the collection size. ✔

Made possible by *linked data structure*.

Also possible to implement the *queue* abstraction with a singly-linked list (see text).
Summary

Stacks and queues
- Fundamental collection abstractions.
- Differ only in order in which items are removed.
- Performance specifications: Constant-time for all operations and space linear in the number of objects.

Linked structures
- Fundamental alternative to arrays.
- Enable implementations of the stack/queue abstractions that meet performance specifications.

Next: Symbol tables
12. Stacks and Queues