14. Stacks and Queues
14. 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.

---

<table>
<thead>
<tr>
<th>Stack API</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>public class Stack&lt;Item&gt;</strong></td>
</tr>
<tr>
<td>Stack&lt;Item&gt;()</td>
</tr>
<tr>
<td>void push(Item item)</td>
</tr>
<tr>
<td>Item pop()</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
</tr>
<tr>
<td>int size()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue API</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>public class Queue&lt;Item&gt;</strong></td>
</tr>
<tr>
<td>Queue&lt;Item&gt;()</td>
</tr>
<tr>
<td>void enqueue(Item item)</td>
</tr>
<tr>
<td>Item dequeue()</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
</tr>
<tr>
<td>int size()</td>
</tr>
</tbody>
</table>

stay tuned for examples
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 proportional to 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.
14. 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 (printer, processor).
- Simulations of the real world (guitar string, traffic analysis, ...)

Stacks
- Last-come-first-served processes (browser, e-mail).
- Function calls in programming languages.
- Basic mechanism in interpreters, compilers.
- ...
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 a link to another page.
- Click "back" button.
- Click "back" button.
- Click "back" button.

http://introcs.cs.princeton.edu/java/43stack/
http://introcs.cs.princeton.edu/java/40algorithms/
http://introcs.cs.princeton.edu/java/home/
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()`.
- Values 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>long</td>
<td>Long</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.

```java
Stack<Integer> stack = new Stack<Integer>();
stack.push(17); // Autobox (int -> Integer)
int a = stack.pop(); // Auto-unbox (Integer -> int)
```

Simple client code (no casts)
Stack client example: Postfix expression evaluation

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

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

**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!

1 2 3 + 4 5 * * +

Find first operator, convert to infix, enclose in ()

There is only one way to parenthesize a postfix expression.

1 (2 + 3) 4 5 * * +

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.

```
1 2 3 + 4 5 * * +

= 5
= 20 100 101
```
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

% 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 TOY program.
- 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

```
100 100 moveto
100 300 lineto
300 300 lineto
300 100 lineto
stroke
```

call "moveto" (takes args from stack)
draw the path

define a path

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)!
14. 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.

<table>
<thead>
<tr>
<th>Strawman API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class StrawStack</td>
<td></td>
</tr>
<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.
Data structure choice. Use an array to hold the collection.
Strawman stack implementation: Test client

```java
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("-"))
            stack.push(item);
        else
            StdOut.print(stack.pop());
    }
    StdOut.println();
}
```

% more tobe.txt
```
to be or not to - be -- that - - - is
% java StrawStack 20 < tobe.txt
to be not that or be
```

What we expect, once the implementation is done.
**Self-assessment 1 on stacks**

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

<table>
<thead>
<tr>
<th>Example 1.</th>
<th>Example 2.</th>
<th>Example 3.</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 5 4 3 2 1 - - - - -</td>
<td>1 - 2 - 3 - 4 - 5 - 6 -</td>
<td>4 1 - 3 2 - - - 6 5 - -</td>
</tr>
<tr>
<td><img src="image.png" alt="Diagram" /></td>
<td><img src="image.png" alt="Diagram" /></td>
<td><img src="image.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Methods define data-type operations (implement APIs).

```
public class StrawStack {
    ...
    public boolean isEmpty() {
        return (N == 0);
    }
    public void push(Object item)
    {  a[N++] = item; }
    public String pop() {
        return a[--N];
    }
    public int size() {
        return N;
    }
    ...
}
```
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.compareTo("-") != 0)
                stack.push(item);
            else
                StdOut.print(stack.pop());
        }
        StdOut.println();
    }
}
```

instance variables

constructor

methods

% more tobe.txt
to be or not to - be -- that -- is

% java StrawStack 20 < tobe.txt
to be not that or be

test client
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>

<table>
<thead>
<tr>
<th>stack</th>
<th>contents after operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>a[0]</td>
<td>to</td>
</tr>
<tr>
<td>a[1]</td>
<td>to</td>
</tr>
<tr>
<td>a[2]</td>
<td>be</td>
</tr>
<tr>
<td>a[3]</td>
<td>be</td>
</tr>
<tr>
<td>a[4]</td>
<td>be</td>
</tr>
<tr>
<td>a[5]</td>
<td>be</td>
</tr>
<tr>
<td>a[6]</td>
<td>be</td>
</tr>
<tr>
<td>a[7]</td>
<td>be</td>
</tr>
<tr>
<td>a[8]</td>
<td>be</td>
</tr>
<tr>
<td>a[9]</td>
<td>be</td>
</tr>
<tr>
<td>a[10]</td>
<td>to</td>
</tr>
<tr>
<td>a[11]</td>
<td>to</td>
</tr>
<tr>
<td>a[12]</td>
<td>to</td>
</tr>
<tr>
<td>a[13]</td>
<td>to</td>
</tr>
<tr>
<td>a[14]</td>
<td>to</td>
</tr>
<tr>
<td>a[15]</td>
<td>to</td>
</tr>
<tr>
<td>a[16]</td>
<td>to</td>
</tr>
<tr>
<td>a[17]</td>
<td>to</td>
</tr>
<tr>
<td>a[18]</td>
<td>to</td>
</tr>
<tr>
<td>a[19]</td>
<td>to</td>
</tr>
</tbody>
</table>

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

StrawStack requires client to provide capacity.

**StrawStack works only for strings**

- All operations are constant-time. ✓
- Memory use is proportional to the size of the collection, when it is nonempty. ✗
- No limits within the code on the collection size. ✗

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

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

#### Sequential data structure
- Put objects next to one another.
- TOY: consecutive memory cells.
- Java: array of objects.
- Fixed size, arbitrary access.  \( \text{ith element} \)

#### Linked data structure
- Associate with each object a link to another one.
- TOY: link is memory address of next object.
- Java: link is reference to next object.
- Variable size, sequential access.  \( \text{next element} \)
- Overlooked by novice programmers.
- Flexible, widely used method for organizing data.

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>C1</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>C2</td>
<td>&quot;Carol&quot;</td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td></td>
</tr>
<tr>
<td>C6</td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td></td>
</tr>
<tr>
<td>C9</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td></td>
</tr>
<tr>
<td>CB</td>
<td></td>
</tr>
</tbody>
</table>

Array at C0

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>&quot;Carol&quot;</td>
</tr>
<tr>
<td>C1</td>
<td>null</td>
</tr>
<tr>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>C5</td>
<td>CA</td>
</tr>
<tr>
<td>C6</td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td></td>
</tr>
<tr>
<td>C9</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>CB</td>
<td>C0</td>
</tr>
</tbody>
</table>

Linked list at C4
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.

A linked list

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

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>C0</td>
<td>&quot;Carol&quot;</td>
</tr>
<tr>
<td>C1</td>
<td>null</td>
</tr>
<tr>
<td>C2</td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>&quot;Alice&quot;</td>
</tr>
<tr>
<td>C5</td>
<td>CA</td>
</tr>
<tr>
<td>C6</td>
<td></td>
</tr>
<tr>
<td>C7</td>
<td></td>
</tr>
<tr>
<td>C8</td>
<td></td>
</tr>
<tr>
<td>CA</td>
<td>&quot;Bob&quot;</td>
</tr>
<tr>
<td>CB</td>
<td>C0</td>
</tr>
</tbody>
</table>
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`.

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

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

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

Node second = first;

first = new Node();

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

<table>
<thead>
<tr>
<th>first</th>
<th>second</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Alice&quot; →</td>
<td>&quot;Bob&quot; →</td>
</tr>
<tr>
<td>&quot;Carol&quot;</td>
<td></td>
</tr>
</tbody>
</table>

item
"Dave"

<table>
<thead>
<tr>
<th>first</th>
<th>second</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Dave&quot; →</td>
<td>&quot;Alice&quot; →</td>
</tr>
<tr>
<td>&quot;Bob&quot; →</td>
<td>&quot;Carol&quot;</td>
</tr>
</tbody>
</table>

Node second = first;

first = new Node();

first.item = item;
first.next = second;
**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 showing traversal of a linked list from first to Alice, Bob, and Carol](image)
Self-assessment 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);
...
Self-assessment 2 on stacks

Q. Give code that uses a stack to print the strings from StdIn on StdOut, in reverse order.
Self-assessment 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
**ADT for pushdown stacks: review**

A *pushdown stack* is an idealized model of a LIFO storage mechanism.

An **ADT** allows us to write Java programs that use and manipulate pushdown stacks.

```java
public class Stack<Item>

Stack<Item>() create a stack of objects, 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
```

**API**

**Performance specifications**

- All operations are constant-time.
- Memory use is proportional to the size of the collection, when it is nonempty.
- No limits within the code on the collection size.
Data structure choice. Use a linked list to hold the collection.

```
public class Stack<Item> {
    private Node first = null;
    private int N = 0;
    
    private class Node {
        private Item item;
        private Node next;
    }
    ...
}
```

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.print(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++; // Increment the size
    }
    public Item pop() {
        Item item = first.item;
        first = first.next;
        N--; // Decrement the size
        return item;
    }
    public int size() {
        return N;
    }
    ...
}
```

**Diagram:**
- **Instance variables:**
  - `first`
  - `second`
- **Constructors:**
  - `push()`: Add a new node to the beginning of the list.
  - `pop()`: Remove and return the first item on the list.
  - Local variable in `push()`.
- **Test client:**
Stack implementation

```java
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
    }
}
```

- **instance variables**
- **nested class**
- **methods**
- **test client**
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**

1. to •
2. be ← to •
3. or ← be ← to •
4. not ← or ← be ← to •
5. to • ← not • ← or • ← be • ← to •
6. not • ← or • ← be • ← to •
7. be ← not • ← or • ← be • ← to •
8. not • ← or • ← be • ← to •
9. be ← not • ← or • ← be • ← to •
10. not • ← or • ← be • ← to •
11. or • ← be • ← to •
12. that ← or • ← be • ← to •
13. or • ← be • ← to •
14. be • ← to •
15. to •

**Pop from the beginning**

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

### Performance specifications

- All operations are constant-time. ✓
- Memory use is proportional to 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).

dequeue(): same code as pop()
enqueue(): slightly more complicated
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 proportional to number of objects.

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

Next: Symbol tables