1.3 Bags, Queues, and Stacks

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

Stacks and queues

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

Stack. Examine the item most recently added. LIFO = "last in first out"
Queue. Examine the item least recently added. FIFO = "first in first out"

Client, implementation, interface

Separate interface and implementation.
Ex: stack, queue, bag, priority queue, symbol table, union-find, ....

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

Client: program using operations defined in interface.
Implementation: actual code implementing operations.
Interface: description of data type, basic operations.
Stack API

Warmup API. Stack of strings data type.

```java
public class StackOfStrings {
    StackOfStrings() { create an empty stack }
    void push(String item) { insert a new string onto stack }
    String pop() { remove and return the string most recently added }
    boolean isEmpty() { is the stack empty? }
    int size() { number of strings on the stack }
}
```

Warmup client. Reverse sequence of strings from standard input.

Stack test client

Read strings from standard input.
- If string equals "-", pop string from stack and print.
- Otherwise, push string onto stack.

```java
public static void main(String[] args) {
    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty()) {
        String s = StdIn.readString();
        if (s.equals("-")) StdOut.print(stack.pop());
        else stack.push(s);
    }
}
```

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

Stack: linked-list representation

Maintain pointer to first node in a linked list; insert/remove from front.

Stack pop: linked-list implementation

Maintain pointer to first node in a linked list; insert/remove from front.
**Stack push: linked-list implementation**

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

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

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

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

---

**Stack: linked-list implementation in Java**

```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: array implementation**

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

Remark. This accounts for the memory for the stack (but not the memory for strings themselves, which the client owns).
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: 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.
• Inserting first N items takes time proportional to \(1 + 2 + \ldots + N \sim N^2/2\).

Challenge. Ensure that array resizing happens infrequently.

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

Null items. We allow null items to be inserted.

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

Stack considerations
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.

```java
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;  
}
```

Consequence. Inserting first $N$ items takes time proportional to $N$ (not $N^2$).

Stack: amortized cost of adding to a stack

Cost of inserting first $N$ 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>$N = 5$</th>
<th>to be</th>
<th>or not</th>
<th>to null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = 4$</td>
<td>to be</td>
<td>or not</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$N = 5$</td>
<td>to be</td>
<td>or not</td>
<td>to null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>$N = 4$</td>
<td>to be</td>
<td>or not</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Invariant. Array is between 25% and 100% full.
Stack: resizing-array implementation trace

<table>
<thead>
<tr>
<th>push()</th>
<th>pop()</th>
<th>N</th>
<th>a.length</th>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>1</td>
<td>1</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>be</td>
<td>2</td>
<td>2</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>or</td>
<td>3</td>
<td>4</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>not</td>
<td>4</td>
<td>4</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>5</td>
<td>8</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>8</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>3</td>
<td>8</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
<td>4</td>
<td>to</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>1</td>
<td>2</td>
<td>to</td>
<td>null</td>
</tr>
<tr>
<td>is</td>
<td>2</td>
<td></td>
<td>to</td>
<td>is</td>
</tr>
</tbody>
</table>

Trace of array resizing during a sequence of push() and pop() operations

Stack resizing-array implementation: performance

Amortized analysis. 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 \( M \).

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

- \( \sim 8N \) when full.
- \( \sim 32N \) when one-quarter full.

public class ResizingArrayStackOfStrings {
    private String[] s;
    private int N = 0;

    // 8 bytes (reference to array) 24 bytes (array overhead)
    // 8 bytes \times array size 4 bytes (int)
    // 4 bytes (padding)
}

Remark. This accounts for the memory for the stack (but not the memory for 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 worst case.
- Uses extra time and space to deal with the links.

Resizing-array implementation.
- Every operation takes constant amortized time.
- Less wasted space.

N = 4

Stack implementations: resizing array vs. linked list
1.3 Bags, Queues, and Stacks

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

Queue API

```java
public class QueueOfStrings {
    public QueueOfStrings() {  // create an empty queue
        // constructor...
    }
    void enqueue(String item) {  // insert a new string onto queue
        // method...
    }
    String dequeue() {  // remove and return the string least recently added
        // method...
    }
    boolean isEmpty() {  // is the queue empty?
        // method...
    }
    int size() {  // number of strings on the queue
        // method...
    }
}
```

Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list; insert/remove from opposite ends.

Queue dequeue: linked-list implementation

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

Remark. Identical code to linked-list stack pop().
Queue enqueue: linked-list implementation

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

create a new node for the end
last = new Node();
last.item = "not";

link the new node to the end of the list
oldlast.next = last;

Queue: linked-list implementation in Java

public class LinkedQueueOfStrings
{
private Node first, last;

private class Node
{
// same as in StackOfStrings */
}

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;
}
}

Queue: resizing array implementation

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

Q. How to resize?
Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 1. Implement a separate stack class for each type.
- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@$^$! most reasonable approach until Java 1.5.

Attempt 2. Implement a stack with items of type Object.
- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 3. Java generics.
- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.

Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Guiding principles. Welcome compile-time errors; avoid run-time errors.

Generic stack: linked-list implementation

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;
    }
}

Guiding principles. Welcome compile-time errors; avoid run-time errors.

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: array implementation

```java
public class FixedCapacityStackOfStrings {
    private String[] sk;
    private int m = 0;

    public FixedCapacityStackOfStrings(int capacity) {
        sk = new String[capacity];
    }

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

    public void push(String item) {
        m += 1;
        sk[m] = item;
    }

    public String pop() {
        return sk[--m];
    }
}
```

@#$" generic array creation not allowed in Java

## the way it should be

```java
public class FixedCapacityStack<Item> {
    private Item[] s;
    private int m = 0;

    public FixedCapacityStack(int capacity) {
        s = new Item[capacity];
    }

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

    public void push(Item item) {
        m += 1;
        s[m] = item;
    }

    public Item pop() {
        return s[--m];
    }
}
```

## the way it is

```java
public class FixedCapacityStackOfStrings {
    private String[] sk;
    private int m = 0;

    public FixedCapacityStackOfStrings(int capacity) {
        s = new String[capacity];
    }

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

    public void push(String item) {
        m += 1;
        s[m] = item;
    }

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

the ugly cast

## Unchecked cast

```java
% javac FixedCapacityStack.java
Note: FixedCapacityStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

% javac -Xlint:unchecked FixedCapacityStack.java
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
found : java.lang.Object[
required: Item[]
    a = (Item[]) new Object[capacity];
1 warning
```

## Generic data types: autoboxing

**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 between a primitive type and its wrapper.

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

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

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

#### Iterators

**Q.** What is an **Iterable**?
**A.** Has a method that returns an **Iterator**.

**Q.** What is an **Iterator**?
**A.** Has methods **hasNext**() and **next**().

**Q.** Why make data structures **Iterable**?
**A.** Java supports elegant client code.

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

#### Java solution
Make stack implement the **java.lang.Iterable** interface.

### 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 Item next() {
            Item item = current.item;
            current = current.next;
            return item;
        }
        public void remove() { throw new UnsupportedOperationException(); }
    }
}
```

#### Iterable interface

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

#### Iterator interface

```java
public interface Iterator<Item> {
    boolean hasNext();
    Item next();
    void remove();
}
```

#### "foreach" statement (shorthand)

```java
for (String s : stack)
    StdOut.println(s);
```

#### equivalent code (longhand)

```java
Iterator<String> i = stack.iterator();
while (i.hasNext())
    String s = i.next();
    StdOut.println(s);
```
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]; }
    }
}
```

Iteration: concurrent modification

**Q.** What if client modifies the data structure while iterating?
**A.** A fail-fast iterator throws a java.util.ConcurrentModificationException.

```java
for (String s : stack)    
    stack.push(s);
```

To detect:
- Count total number of push() and pop() operations in Stack.
- Save counts in Iterator subclass upon creation.
- If, when calling next() and hasNext(), the current counts do not equal the saved counts, throw exception.

Java collections library

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

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

**Implementations.** java.util.ArrayList uses resizing array;
java.util.LinkedList uses linked list.  caveat: only some operations are efficient
**Java collections library**

`java.util.Stack`

- Supports `push()`, `pop()`, and `iteration`.
- Extends `java.util.Vector`, which implements `java.util.List` interface from previous slide, including `get()` and `remove()`.
- Bloated and poorly-designed API (why?)

**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.

**War story (from Assignment 1)**

Generate random open sites in an $N$-by-$N$ percolation system.
- Jenny: pick $(i, j)$ 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.

**Stack applications**

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

How a compiler implements a function.
• Function call: push local environment and return address.
• Return: pop return address and local environment.

Recursive function. Function that calls itself.
Note. Can always use an explicit stack to remove recursion.

---

Arithmetic expression evaluation

Goal. Evaluate infix expressions.

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

Two-stack algorithm. [E. W. Dijkstra]
• Value: push onto the value stack.
• Operator: push onto the operator stack.
• Left parenthesis: ignore.
• Right parenthesis: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!

---

Dijkstra's two-stack algorithm demo

---

Arithmetic expression evaluation

```
public class Evaluate
{
    public static void main(String[] args)
    {
        Stack<String> ops = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();
        while (!StdIn.isEmpty()) {
            String s = StdIn.readString();
            if (s.equals("("))
                ;
            else if (s.equals("+"))
                ops.push(s);
            else if (s.equals("*"))
                ops.push(s);
            else if (s.equals(")"))
                {
                    String op = ops.pop();
                    if (op.equals("+"))
                        vals.push(vals.pop() + vals.pop());
                    else if (op.equals("*"))
                        vals.push(vals.pop() * vals.pop());
                    else vals.push(Double.parseDouble(s));
                }
        }
        StdOut.println(vals.pop());
    }
}
```

---
Correctness

Q. Why correct?
A. When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

\[
(1 + ((2 + 3) \ast (4 \ast 5)))
\]

as if the original input were:

\[
(1 + (5 \ast (4 \ast 5)))
\]

Repeating the argument:

\[
(1 + ((5 \ast 20))
\]

\[
(1 + 100)
\]

101

Extensions. More ops, precedence order, associativity.

Stack-based programming languages

Observation 1. Dijkstra's two-stack algorithm computes the same value if the operator occurs after the two values.

\[
(1 ((2 + 3) \ast (4 \ast 5)) + )
\]

Observation 2. All of the parentheses are redundant!

\[
1 2 3 + 4 5 \ast \ast +
\]

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, …