1.3 Bags, Queues, and Stacks

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

Stacks and queues

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

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

Interfaces can be ambiguous

Stacks and queues.
- Value: collection of objects.
- Operations: add, remove, iterate, test if empty.

Q. What are two ways in which the semantics of iteration can be ambiguous?
A.  
- What order to iterate in: same as removal order or does client not care?
- What happens if collection is modified during iteration?

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.
Ambiguity in the semantics of interfaces leads to bugs

Example: Mars climate orbiter.
Lost due to metric vs. imperial mishap in contract between NASA & Lockheed

Layers in a computer system

Program
Libraries
Programming language
Operating system
Hardware

Stack API

Warmup API. Stack of strings data type.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>StackOfStrings()</td>
<td>create an empty stack</td>
</tr>
<tr>
<td>void push()</td>
<td>add a new string to stack</td>
</tr>
<tr>
<td>String pop()</td>
<td>remove and return the string</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>

1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Stacks quiz 1

How to implement a stack with a singly-linked list?

A. least recently added

B. most recently added

C. None of the above.

D. I don’t know.

Stack: linked-list implementation

- Maintain pointer first to first node in a singly-linked list.
- Push new item before first.
- Pop item from first.

Stack pop: linked-list implementation

Stack push: linked-list implementation
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;
    }
}
```

private inner class (access modifiers for instance variables don’t matter)

Stack: linked-list implementation performance

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

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

Fixed-capacity stack: array implementation

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

Defect. Stack overflows when `n` exceeds capacity. [stay tuned]
Fixed-capacity stack: array implementation

```java
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int n = 0;
    public FixedCapacityStackOfStrings(int capacity) {
        s = new String[capacity];
    }
    public boolean isEmpty() {
        return n == 0;
    }
    public void push(String item) {
        s[n++] = item;
    }
    public String pop() {
        return s[--n];
    }
}
```

Stack considerations

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

Null items. We allow null items to be added.
Duplicate items. We allow an item to be added more than once.
Loitering. Holding a reference to an object when it is no longer needed.

Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!
Q. How to grow and shrink array?

First try.
- push(): increase size of array s[] by 1.
- pop(): decrease size of array s[] by 1.

Too expensive.
- Need to copy all items to a new array, for each operation.
- Array accesses to add first n items = n + (2 + 4 + ... + 2(n - 1)) ~ n^2.

Infeasible for large n

Challenge. Ensure that array resizing happens infrequently.
Why isn’t array resizing easier?

Q. Why no way to increment array size except by creating new one?
A. Because Java doesn’t let us
   Because OS doesn’t let us safely write past memory bounds
   Because memory is fragmented at the hardware level

Stack: resizing-array implementation

Q. How to grow array?
A. If array is full, create a new array of twice the size, and copy items.

```
public void push(String item)
{
  if (n == s.length) resize(2 + s.length);
  s[n++] = item;
}
```

Array accesses to add first \( n = 2^i \) items. \( n + (2 + 4 + 8 + \ldots + n) \sim 3n. \)

```
private void resize(int capacity)
{
  String[] copy = new String[capacity];
  for (int i = 0; i < n; i++)
    copy[i] = s[i];
  s = copy;
}
```

Stack: resizing-array implementation

Q. How to shrink?

First try.
- `push()`: double size of array \( s[] \) when array is full.
- `pop()`: half 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 \( n \).

```
full to be or* not
push("no") to be or* not to null null null
pop() to be or* not
push("be") to be or* not be null null
```

Efficient solution.
- `push()`: double size of array \( s[] \) when array is full.
- `pop()`: half size of array \( s[] \) when array is one-quarter full.

```
public String pop()
{
  String item = s[--n];
  s[n] = null;
  if (n > 0 && n == s.length/4) resize(s.length/2);
  return item;
}
```

Invariant. Array is between 25% and 100% full.
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 \( 8n \) to \( 32n \) bytes of memory for a stack with \( n \) items.
- \( 8n \) when full.
- \( 32n \) when one-quarter full.

public class ResizingArrayStackOfStrings {
    private String[] s;  // 8 bytes x array size
    private int n = 0;
    ...
}

Stack implementations: resizing array vs. linked list

Tradeoffs. Can implement a stack with either resizing array or linked list; client can use interchangeably. Which one is better?

Linked-list implementation.
- Every operation takes constant time in the worst case.
- Uses extra time and space to deal with the links.

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

Exercise: complete this argument.

Exercise: would proof hold if we'd had different constants instead of \( \frac{1}{2} \) and \( \frac{1}{4} \) (say \( \frac{3}{5} \) & \( \frac{1}{5} \))? What are the factors to consider in picking the constants?

Remark. This accounts for the memory for the stack (but not the memory for strings themselves, which the client owns).
1.3 Bags, Queues, and Stacks

Queues quiz 1

How to implement a queue with a singly-linked linked list?

A.  most recently added
1. times → of → best → the → was → it → null

B.  least recently added
1. it → was → the → best → of → times → null

C. None of the above.

D. I don’t know.
Queue.dequeue: linked-list implementation

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

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

Queue.enqueue: linked-list implementation

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

Queue: linked-list implementation in Java

```java
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;
    return item;
  }
}
```

Queues quiz 2

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

A. least recently added

```
0 1 2 3 4 5 6 7 8 9
```

B. most recently added

```
0 1 2 3 4 5 6 7 8 9
```

C. None of the above.

D. I don't know.
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.
- Add resizing array.

<table>
<thead>
<tr>
<th>least recently added</th>
<th>most recently added</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td></td>
</tr>
</tbody>
</table>

Q. How to resize?

Queue with two stacks

Job interview problem. Implement a queue with two stacks so that:
- Each queue op uses a constant amortized number of stack ops.
- At most constant extra memory (besides the two stacks).

Solution. Call the two stacks incoming and outgoing.
- enqueue: push to incoming
- dequeue: pop from outgoing
  - if outgoing is empty, first “pour” incoming into outgoing (O(N)).
- isEmpty: check if both stacks are empty

Analysis: correctness. Left as exercise.

Analysis: efficiency. Consider the lifecycle of each item:
- pushed into incoming, popped from incoming,
- pushed into outgoing, popped from outgoing.
At most 4 stack operations per item. [Exercise: complete this argument.]

Parameterized stack

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

Solution in Java: generics.

1.3 BAGS, QUEUES, AND STACKS

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

Parameterized stack

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

Solution in Java: generics.

Parameterized stack

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

Solution in Java: generics.

Parameterized stack

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

Solution in Java: generics.

Parameterized stack

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

Solution in Java: generics.

Parameterized stack

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

Solution in Java: generics.

Parameterized stack

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

Solution in Java: generics.
Generic stack: linked-list implementation

```java
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;
    }
}
```

Generic stack: array implementation

```java
class FixedCapacityStackOfStrings {
    private String[] s;
    private int n = 0;
    public ...StackOfStrings(int capacity) {
        s = new String[capacity];
    }
    public boolean isEmpty() { return n == 0; }
    public void push(String item) {
        s[n++] = item;
    }
    public String pop() {
        return s[--n];
    }
}
```

Generic data types: autoboxing and unboxing

**Q.** What to do about primitive types?

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

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

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

```java
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 any type of data.
1.3 Bags, Queues, and Stacks

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

Iterators

To support for-each 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.

For a user-defined collection, do this to enable looping over it with for-each:
- Data type must have a method named iterator().
- The iterator() method returns an object that has two core methods:
  - the hasNext() method returns false when there are no more items
  - the next() method returns the next item in the collection

Type safety.
- Data type must use these interfaces to support for-each loop.
- Client program won’t compile if implementation doesn’t.

Iteration

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

resizing-array representation

<table>
<thead>
<tr>
<th>s[]</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>it</td>
<td>was</td>
<td>the</td>
<td>best</td>
<td>of</td>
<td>time</td>
<td>stack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

linked-list representation

```
for...each loop
```

Java solution. Use a for-each loop.
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;
        }

        throw new UnsupportedOperationException();
        throw new UnsupportedOperationException();
        if (no more items in iteration)
    }
}
```

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 void 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);
}
```

Q. How to detect concurrent modification?
A.
- Count total number of push() and pop() operations in Stack.
- Save counts in Iterator subclass upon creation.
- If, when calling either next() or hasNext(), the current counts do not equal the saved counts, throw exception.

1.3 Bags, Queues, and Stacks
For this course, use our Stack and Queue implementations instead of Java's.

### Arithmetic expression evaluation

**Goal.** Evaluate infix expressions using two stacks.

\[
\text{( 1 + ( ( 2 + 3 ) \times ( 4 \times 5 ) ) )}
\]

**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
  - apply operator to the two values
  - push the result to value stack.

### 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.
- ...

### Dijkstra's two-stack algorithm demo

**Infix expression** (fully parenthesized)

\[
\text{( 1 + ( ( 2 + 3 ) \times ( 4 \times 5 ) ) )}
\]

- **Operand**
- **Operator**
Arithmetic expression evaluation

```java
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("/")) ops.push(s);
            else if (s.equals("*")) ops.push(s);
            else if (s.equals("+")) ops.push(s);
            else if (s.equals("-")) ops.push(s);
            String op = ops.pop();
            if (op.equals("/")) vals.push(vals.pop() / vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
            else if (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("-")) vals.push(vals.pop() - vals.pop());
            double result = Double.parseDouble(s);
            vals.push(result);
        }
        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.

As if the original input were:

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

Repeating the argument:

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

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.

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

Observation 2. All of the parentheses are redundant!

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

Bottom line. Postfix or "reverse Polish" notation.
Applications. Postscript, Forth, calculators, Java virtual machine, ...