1.3 Stacks and Queues

> stacks
> dynamic resizing
> queues
> generics
> iterators
> applications

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

 **Stack**. Remove the item most recently added.
 **Analogy**. Cafeteria trays, Web surfing.
 **LIFO** = “last in first out”

 **Queue**. Remove the item least recently added.
 **Analogy**. Registrar’s line.
 **FIFO** = “first in first out”

 Client, implementation, interface

 Separate interface and implementation.
 Ex: stack, queue, 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.
Warmup. Stack of strings objects.

```
public class StackOfStrings

    StackOfStrings() create an empty stack
    void push(String s) insert a new item onto stack
    String pop() remove and return the item most recently added
    boolean isEmpty() is the stack empty?
    int size() number of items on the stack
```

Challenge. Reverse sequence of strings from standard input.

```
public class StackOfStrings

    StackOfStrings() create an empty stack
    void push(String s) insert a new item onto stack
    String pop() remove and return the item most recently added
    boolean isEmpty() is the stack empty?
    int size() number of items on the stack
```

Stack pop: linked-list implementation

```
save item to return
    String item = first.item;

save item to return
    first = first.next;

return saved item
    return item;
```

Stack push: linked-list implementation

```
create a new node for the beginning
    first = new Node();
    Node oldfirst = first;

set the instance variables in the new node
    first.item = "not";
    first.next = oldfirst;

```
Stack: linked-list implementation in Java

```java
public class StackOfStrings {
    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() {
        if (isEmpty()) throw new RuntimeException();
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

Proposition. Using a linked-list implementation of a stack, every operation takes constant time in the worst case.

Proposition. Uses ~ 40 \(N\) bytes to represent a stack with \(N\) items.
  - assume 64-bit machine (8 bytes per reference)
  - extra overhead for inner class

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

Lesson. "Swollen" pointers can use up memory on 64-bit machines!

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 \(s[N-1]\).

```java
public class StackOfStrings {
    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];
    }
}
```

Defect. Stack overflows when \(N\) exceeds capacity. [stay tuned]
Stack: dynamic-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 s[] by 1.
- pop(): decrease size of s[] by 1.

Too expensive.
- Need to copy all item to a new array.
- Inserting first N items takes time proportional to \(1 + 2 + \ldots + N \sim N^2/2\), infeasible for large N

**Challenge.** Ensure that array resizing happens infrequently.

---

**Stack: dynamic-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 StackOfStrings() { 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. \(N + (2 + 4 + 8 + \ldots + N) \sim 3N\).

<table>
<thead>
<tr>
<th>one gray dot for each operation</th>
<th>red dots give cumulative average</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost (array/accesses)</td>
<td>number of push() operations</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>128</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td></td>
</tr>
</tbody>
</table>

Cost of array resizing is now \(2 + 4 + 8 + \ldots + N \sim 2N\).
Stack: dynamic-array implementation

Q. How to shrink array?

First try.
• push(): double size of a[] when array is full.
• pop(): halve size of a[] when array is one-half full.

Too expensive.
• Consider push-pop-push-pop-... sequence when array is full.
• Takes time proportional to \( N \) per operation in worst case.

```
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 dynamic-array implementation: performance

Amortized analysis. Average running time per operation over a worst-case sequence of operations.

Proposition. Starting from empty stack (with dynamic resizing), any sequence of \( M \) push and pop operations takes time proportional to \( M \).

<table>
<thead>
<tr>
<th>Operation</th>
<th>best</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>
Stack dynamic array implementation: memory usage

**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.

```java
public class DoublingStackOfStrings {
    private String[] s;
    private int N = 0;
    ...
}
```

8 bytes \times array size
4 bytes

**Remark.** Analysis includes memory for the stack (but not the strings themselves, which the client owns).

Stack implementations: dynamic array vs. linked list

**Tradeoffs.** Can implement a stack with either dynamic 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.

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

Queue API
Queue test client

```java
public static void main(String[] args) {
    QueueOfStrings q = new QueueOfStrings();
    while (!StdIn.isEmpty()) {
        String item = StdIn.readString();
        if (item.equals('-')) StdOut.print(q.dequeue());
        else                  q.enqueue(item);
    }
}
```

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

Queue dequeue: linked-list implementation

```java
save item to return
String item = first.item;
```

Queue enqueue: linked-list implementation

```java
save a link to the last node
Node oldlast = last;
```

Queue: linked-list implementation in Java

```java
public class QueueOfStrings {
    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;
    }
}
```

```java
class Node {
    /* same as in StackOfStrings */
}
class QueueOfStrings {
    private Node first, last;
```

```java
```
Queue: linked-list trace

Queue: dynamic 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 dynamic resizing.

Parameterized stack

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

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

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.

```java
StackOfObjects s = new StackOfObjects();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

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

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

```java
Stack<Apple> s = new Stack<Apple>();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = s.pop();
```

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

Generic stack: linked-list implementation

```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;
    }
}
```

Generic stack: array implementation

```java
public class ArrayStackOfStrings
{
    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];  }
}
```

The way it should be

```java
public class ArrayStack<Item>
{
    private Item[] s;
    private int N = 0;
    public Stack<Item> s = new Stack<Item>();
    public boolean isEmpty()
    {  return N == 0;  }
    public void push(Item item)
    {  s[N++] = item;  }
    public Item pop()
    {  return s[--N];  }
}
```

@#$^1 generic array creation not allowed in Java
Generic stack: array implementation

```java
public class ArrayStack<Item>
{
    private Item[] s;
    private int N = 0;
    public ArrayStack(int capacity)
    {  s = new Object[capacity]; }
    public boolean isEmpty()
    {  return N == 0;  }
    public void push(Item item)
    {  s[N++] = item;  }
    public Item pop()
    {  return s[--N];  }
}
```

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.

Syntactic sugar. Behind-the-scenes casting.

Bottom line. Client code can use generic stack for any type of data.

Iteration

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

Java solution. Make stack implement the Iterable interface.
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.

---

Iterables

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

---

Stack iterator: linked-list implementation

```
import java.util.Iterator;
public class Stack<Item> implements Iterable<Item> {
    ...
    public Iterator<Item> iterator() { return new ListIterator(); }
    private class ListIterator implements Iterator<Item> {
        private Node current = first;
        public boolean hasNext() {  return current != null;  }
        public void remove() { optional; use at your own risk }
        public Item next() { Item item = current.item; current = current.next; return item; }
    }
}
```

---

```
import java.util.Iterator;
public class Stack<Item> implements Iterable<Item> {
    ...
    public Iterator<Item> iterator() { return new ArrayIterator(); }
    private class ArrayIterator 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 `ConcurrentModificationException`.

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

To detect:
- Count total number of `push()` and `pop()` operations in `Stack`.
- Save current count in `Iterator` subclass upon creation.
- Check that two values are still equal when calling `next()` and `hasNext()`.

---

```
for (String s : stack) StdOut.println(s);
```
Three ADTs for processing collections of objects:
- Stack
- Queue
- Bag

Generic implementations
- reusable code (don’t need new implementation for new type)
- compile-time type checks

Iteration
- stack: LIFO order
- queue: FIFO order
- bag: arbitrary order

Bag

```java
public class Bag<Item> implements Iterable<Item> {
    Bag() {
        create an empty bag
    }
    void add(Item x) {
        add an item
    }
    int size() {
        number of items in bag
    }
    Iterable<Item> iterator() {
        iterator for all items in bag
    }
}
```

Sweet spot: Save for iteration where order doesn’t matter.

Queue

```java
public class Queue<Item> implements Iterable<Item> {
    Queue() {
        create an empty queue
    }
    void enqueue(Item x) {
        add an item
    }
    Item dequeue() {
        remove the least recently added item
    }
    int size() {
        number of items in queue
    }
    Iterable<Item> iterator() {
        iterator for all items in queue
    }
}
```

Typical client (put the ints in a file into an array)

```java
public static int[] readInts(String name) {
    In in = new In(name);
    Queue<Integer> q = new Queue<Integer>();
    while (!in.isEmpty())
        q.enqueue(in.readInt());
    int N = q.size();
    int[] a = new int[N];
    for (int i = 0; i < N; i++)
        a[i] = q.dequeue();
    return a;
}
```

Sweet spot: Save for later use where order does matter.

Summary

- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications
Stack

public class Stack<Item> implements Iterable<Item>
{
    Stack()
    void push(Item x)
    Item pop()
    int size()
    Iterable<Item> iterator()
}

Sweet spot: Support recursive computation (stay tuned).

Java collections library

List interface. java.util.List is API for ordered collection of items.

public interface List<Item> implements Iterable<Item>
{
    List()
    boolean isEmpty()
    int size()
    void add(Item item)
    Item get(int index)
    Item remove(int index)
    boolean contains(Item item)
    Iterator<Item> iterator()
    ...
}

Implementations. java.util.ArrayList uses dynamic array;
java.util.LinkedList uses linked list.

Java collections library

java.util.Stack.
• Supports push(), pop(), size(), isEmpty(), and iteration.
• Also implements java.util.List interface from previous slide,
  including get(), remove(), and contains().
• Bloated and poorly-designed API => don’t use.

java.util.Queue. An interface, not an implementation of a queue.

Best practices. Use our implementations of Stack, Queue, and Bag.

War story (from COS 226)

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.LinkedList of \( N^2 \) closed sites.
  Pick an index at random and delete.
  Takes \( \sim c_2 N^4 \) seconds.

Why is my program so slow?

Why is my program so slow?

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.
- ...

Context. An interpreter!

Arithmetic expression evaluation

Goal. Evaluate infix expressions.

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

Context. An interpreter!

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.
- Not. Can always use an explicit stack to remove recursion.
Correctness

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

\[
\{ 1 + ( ( 2 + 3 ) \times ( 4 + 5 ) ) \}
\]
as if the original input were:

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

Repeating the argument:

\[
\begin{align*}
&\{ 1 + ( 5 \times 20 ) \} \\
&\{ 1 + 100 \} \\
&101
\end{align*}
\]

Extensions. More ops, precedence order, associativity.

Stack-based programming languages

Observation 1. The 2-stack algorithm computes the same value if the operator occurs after the two values.

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

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, …

PostScript

PostScript. [Warnock-Geschke 1980s]
- Postfix program code.
- Turtle graphics commands.
- Variables, types, text, loops, conditionals, functions, ...

Define a path

\[
\begin{align*}
&100 100 moveto \\
&100 300 lineto \\
&300 300 lineto \\
&300 100 lineto \\
&stroke
\end{align*}
\]

Simple virtual machine, but not a toy.
- Easy to specify printed page.
- Easy to implement in printers.
- Revolutionized the publishing world.

Page description language.
- Explicit stack.
- Full computational model
- Graphics engine.

Basics.
- \%! "I am a PostScript program."
- Literal: "push me on the stack."
- Function calls take arguments from stack.
- Turtle graphics built in.

\[
\begin{align*}
&61 \\
&72 72 moveto \\
&0 72 rlineto \\
&72 0 rlineto \\
&0 -72 rlineto \\
&-72 0 rlineto \\
&2 setlinewidth \\
&stroke
\end{align*}
\]

Its output
PostScript

Data types.
- Basic: integer, floating point, boolean, ...
- Graphics: font, path, curve, ....
- Full set of built-in operators.

Text and strings.
- Full font support.
- `show` (display a string, using current font).
- `cvs` (convert anything to a string).

%!
/Helvetica-Bold findfont 16 scalefont setfont
72 168 moveto
(Square root of 2:) show
72 144 moveto
2 sqrt 10 string cvs show

Variables (and functions).
- Identifiers start with `/`.
- `def` operator associates id with value.
- Braces.
- `args` on stack.

%!
/box {
  /sz exch def
  0 sz rlineto
  sz 0 rlineto
  0 sz neg rlineto
  sz neg 0 rlineto
  } def
72 144 moveto
72 box
288 288 moveto
144 box
2 setlinewidth
stroke

For loop.
- "from, increment, to" on stack.
- Loop body in braces.
- `for` operator.

If-else conditional.
- Boolean on stack.
- Alternatives in braces.
- `if` operator.

... (hundreds of operators)

PostScript applications


 Algorithms, 4th edition. Figures created using enhanced version of `StdDraw` that saves to PostScript for vector graphics.

See page 218
Queue applications

Familiar applications.
- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.
- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

M/M/1 queuing model

M/M/1 queue.
- Customers arrive according to Poisson process at rate of $\lambda$ per minute.
- Customers are serviced with rate of $\mu$ per minute.

Q. What is average wait time $W$ of a customer in system?
Q. What is average number of customers $L$ in system?

M/M/1 queuing model: example simulation

```
public class MM1Queue {
    public static void main(String[] args) {
        double lambda = Double.parseDouble(args[0]); // arrival rate
        double mu = Double.parseDouble(args[1]); // service rate
        double nextArrival = StdRandom.exp(lambda);
        double nextService = nextArrival + StdRandom.exp(mu);
        Queue<Double> queue = new Queue<Double>();
        Histogram hist = new Histogram("M/M/1 Queue", 60);
        while (true) {
            while (nextArrival < nextService) {
                queue.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            }
            double arrival = queue.dequeue();
            double wait = nextService - arrival;
            hist.addDataPoint(Math.min(60, (int) (Math.round(wait))));
            if (queue.isEmpty()) nextService = nextArrival + StdRandom.exp(mu);
            else nextService = nextService + StdRandom.exp(mu);
        }
    }
}
```

M/M/1 queuing model: event-based simulation

```
public class MM1Queue{
    public static void main(String[] args) {
        double lambda = ... Exp(lambda);
        double mu = ... Exp(mu);
        Queue<Double> queue = new Queue<Double>();
        Histogram hist = new Histogram("M/M/1 Queue", 60);
        while (true) {
            while (nextArrival < nextService) {
                queue.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            }
            double arrival = queue.dequeue();
            double wait = nextService - arrival;
            hist.addDataPoint(Math.min(60, (int) (Math.round(wait))));
            if (queue.isEmpty()) nextService = nextArrival + StdRandom.exp(mu);
            else nextService = nextService + StdRandom.exp(mu);
        }
    }
}
```
**Observation.** If service rate $\mu$ is much larger than arrival rate $\lambda$, customers get good service.

```java
% java MM1Queue .2 .333
```

**Observation.** As service rate $\mu$ approaches arrival rate $\lambda$, services go to h***.

```java
% java MM1Queue .2 .25
```

**Observation.** As service rate $\mu$ approaches arrival rate $\lambda$, services go to h***.

```java
% java MM1Queue .2 .21
```

**M/M/1 queuing model: analysis**

**M/M/1 queue.** Exact formulas known.

Little's Law

$W = \frac{1}{\mu - \lambda}$, $L = \lambda W$

More complicated queueing models. Event-based simulation essential!

Queueing theory. See ORF 309.