1.3 BAGS, QUEUES, AND STACKS

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

Stacks and queues

**Fundamental 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”

Client, implementation, interface

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

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

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

1.3 BAGS, QUEUES, AND STACKS

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

Warmup API. Stack of strings data type.

```java
public class StackOfStrings {
    public StackOfStrings() {
        // create an empty stack
    }
    void push(String item) {
        // add a new string to 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.

Stacks quiz 1

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

A. least recently added
   ↓
   of ─→ best ─→ the ─→ best ─→ of ─→ null

B. most recently added
   ↓
   of ─→ best ─→ the ─→ was ─→ it ─→ null

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`.

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

public class Stack {
    private Node first;

    public Stack() {
        // create an empty stack
    }
    void push(String item) {
        // add a new string to stack
        first = new Node();
        first.item = item;
        first.next = first;
    }
    String pop() {
        // remove and return the string most recently added
        if (first == null) {
            return null;
        }
        String item = first.item;
        first = first.next;
        return item;
    }
    boolean isEmpty() {
        // is the stack empty?
        return first == null;
    }
    int size() {
        // number of strings on the stack
        if (first == null) {
            return 0;
        }
        int count = 1;
        Node current = first.next;
        while (current != first) {
            count += 1;
            current = current.next;
        }
        return count;
    }
}
```

Stack pop: linked-list implementation

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

delete first node
first = first.next;

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

```
first = first.next;
```

```
return saved item
return item;
```
Stack push: linked-list implementation

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

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

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

Stack push: linked-list implementation

save a link to the list
Node oldfirst = first;

Stack: linked-list implementation in Java

public class LinkedStackOfStrings
{
   private Node first = null;

   private class Node
   {
      private String item;
      private 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: linked-list implementation performance

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

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

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

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

Stacks quiz 2

How to implement a fixed-capacity stack with an array?

A. least recently added

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

B. most recently added

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

C. None of the above.

D. I don't know.
Fixed-capacity stack: array implementation

- Use array \( s[] \) to store \( N \) items on stack.
- \( \text{push()} \): add new item at \( s[N] \).
- \( \text{pop()} \): remove item from \( s[N-1] \).

Least recently added

\[
s[] ^ {\text{least recently added}}
\]

\[
\begin{array}{cccccccccc}
\text{it} & \text{was} & \text{the} & \text{best} & \text{of} & \text{times} & \text{null} & \text{null} & \text{null} & \text{null} \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\( N \), capacity = 10

**Defect.** Stack overflows when \( N \) exceeds capacity. [stay tuned]

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.

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

```java
public String pop() {
    String item = s[--N];
    s[N] = null;
    return item;
}
```

This version avoids "loitering": garbage collector can reclaim memory for an object only if no remaining references

**1.3 Bags, Queues, and Stacks**

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
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 = $N + (2 + 4 + \ldots + 2(N-1)) \sim N^2$.

Challenge. Ensure that array resizing happens infrequently.

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>full</th>
<th>to</th>
<th>be</th>
<th>or</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(&quot;to&quot;)</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
</tr>
<tr>
<td>pop()</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
</tr>
<tr>
<td>push(&quot;be&quot;)</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
</tr>
</tbody>
</table>

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;
}
```

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

Stack: resizing-array implementation

Q. How to shrink array?

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

```java
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 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.

**Stack resizing-array implementation: memory usage**

**Proposition.** A ResizingArrayStackOfStrings uses $\sim 8N$ to $\sim 32N$ bytes of memory for a stack with $N$ items.
- $\sim 8N$ when full.
- $\sim 32N$ when one-quarter full.

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

8 bytes x array size

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

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

Queue: linked-list implementation

- Maintain one pointer `first` to first node in a singly-linked list.
- Maintain another pointer `last` to last node.
- Dequeue from `first`.
- Enqueue after `last`.

Queue dequeue: linked-list implementation

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

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

Queues quiz 1

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

A. **most recently added**
   ```
   times → of → best → the → it → null
   ```

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

C. None of the above.

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

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

Queue: linked-list implementation in 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 = "not";
            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

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.

Queues quiz 2

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

A. least recently added

<table>
<thead>
<tr>
<th></th>
<th>it</th>
<th>was</th>
<th>the</th>
<th>best</th>
<th>of</th>
<th>times</th>
<th>null</th>
<th>null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

B. most recently added

<table>
<thead>
<tr>
<th></th>
<th>times</th>
<th>of</th>
<th>best</th>
<th>the</th>
<th>was</th>
<th>it</th>
<th>null</th>
<th>null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>1 1</td>
<td>2 2</td>
<td>3 3</td>
<td>4 4</td>
<td>5 5</td>
<td>6 6</td>
<td>7 7</td>
<td>8 8</td>
<td>9 9</td>
<td></td>
</tr>
</tbody>
</table>

C. None of the above.

D. I don’t know.
**Queue With Two Stacks**

**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 two stacks).

**Applications.**
- Job interview.
- Implement an immutable or persistent queue.
- Implement a queue in a (purely) functional programming language.

---

**Parameterized stack**

**We implemented:** Stack0fStrings.
**We also want:** Stack0fURLs, Stack0fInts, Stack0fApples, Stack0fOranges, ....

**Solution in Java:** generics.

```java
public class LinkedStack0fStrings {
    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: linked-list implementation**

```java
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;
    }
}
```

---

**HTTP URL:** [http://algs4.cs.princeton.edu](http://algs4.cs.princeton.edu)
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];
    }
}

@#$! generic creation not allowed in Java

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

Unchecked cast

% 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];
A
1 warning

Q. Why does Java make me cast (or use reflection)?
Short answer. Backward compatibility.
Long answer. Need to learn about type erasure and covariant arrays.

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.

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.
Stacks quiz 5

Which of the following is the correct way to declare and initialize an empty stack of characters?

A. Stack<Character> stack = new Stack();

B. Stack stack = new Stack<Character>();

C. Stack<Character> stack = new Stack<Character>();

D. Stack<Char> stack = new Stack<Char>();

E. None of the above.

1.3 BAGS, QUEUES, AND STACKS

Java provides elegant syntax for iteration over collections.

"foreach" loop (shorthand)

Stack<String> stack;
...
for (String s : stack)
...

equivalent code (longhand)

Stack<String> stack;
...
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
    String s = i.next();
    ...
}

To make user-defined collection support foreach loop:

• 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

Iteration

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

resizing-array representation

i
N
s[]

0 1 2 3 4 5 6 7 8 9
it was the best of times null null null null

linked-list representation

first current

times of best the was it null

Java solution. Use a foreach loop.
Iterators

To support foreach 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.

```java
java.util.Iterator interface
public interface Iterator<Item> {
  boolean hasNext();
  Item next();
  void remove(); ← optional; use at your own risk
}
```

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

Type safety.

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

Stack iterator: linked-list implementation

First

Times

Best

The

Was

It

Null

Stack iterator: array implementation

Bag API

Main application. Adding items to a collection and iterating (when order doesn’t matter).

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

Implementation. Stack (without pop) or queue (without dequeue).
1.3 Bags, Queues, and Stacks

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

Java collections library

Java.util.Stack
- Supports push(), pop(), and iteration.
- Inherits from java.util.Vector, which implements java.util.List interface.

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

Best practices. Use our implementations of Stack and Queue.
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. Cannot use a library until we’ve implemented it in class.

Why is my program so slow?

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

Infix expression
(fully parenthesized)

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

Correctness

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

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

as if the original input were:

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

Repeating the argument:

\[(1 + (5 \times (4 \times 5)))\]
\[(1 + 100)\]
101

Extensions. More ops, precedence order, associativity.

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("=")) /* noop */;
            else if (s.equals("+")) { ops.push(s);
            else if (s.equals("-")) { ops.push(s);
            else if (s.equals("*")) { vals.push(s);
            else if (s.equals("/")) { vals.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 vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}
```

\[% java Evaluate (1 + ((2 + 3) \times (4 \times 5)))
101.0\]

Stack-based programming languages

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

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

Observation 2. All of the parentheses are redundant!

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

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