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.

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 {
    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 quiz

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

A. Can't be done efficiently with a singly-linked list.

B. it was the best of null

C. of best the was it null

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

Stack pop: linked-list implementation

```java
String item = first.item;
first = first.next;
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();
oldfirst = first;

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

Stack: linked-list implementation in 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: 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;
}

16 bytes (object overhead)
8 bytes (inner class extra overhead)
8 bytes (reference to String)
8 bytes (reference to Node)
40 bytes per stack node

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

Stack quiz

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

A. Can't be done efficiently with an array.

B. It was the best of times null null null null

C. Times of best the was it null null null null

D. I don't know.
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 $s[N-1]$.

```
least recently added
↓
$s[]$
```

<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>

$N$ 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 inserted.

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

```
pUBLIC String pop()
{ return s[--N]; }  // loitering
```

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

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 insert first \(N\) items: \(N = 2 + 4 + \ldots + 2(N - 1) \approx N^2\).

Challenge. Ensure that array resizing happens infrequently.

Stack: resizing-array implementation

Q. How to grow array?
A. If array is full, create a new array of \textit{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 insert first \(N = 2^i\) items. \(N + (2 + 4 + 8 + \ldots + N) \approx 3N\).

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>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>

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.

N = 4

| First | to be or not | null | null | null | null |

1.3 Bags, Queues, and Stacks
Queue API

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

Queue quiz 1

How to implement a queue with a linked list?

A. Can't be done efficiently with a singly-linked list.

B. Times of best the was it null

C. It was the best of times it null

D. I don't know.

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

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

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

delete first node
first = first.next;

return saved item
```

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

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

queue enqueue
Node oldlast = last;
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 LinkedStackOfStrings */

        public boolean isEmpty()
        { return first == null; }

        public void enqueue(String item)
        {
            Node oldlast = last;
            last = new Node();
            last.item = item;
            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.

Queue quiz 2

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

A. Can't be done efficiently with an array.

least recently added

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

most recently added

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

D. I don't know.

Q. How to resize?
1.3 Bags, Queues, and Stacks

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.

Parameterized stack

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

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());
```

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.

```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.
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.

```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.

### Iteration

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

```
<table>
<thead>
<tr>
<th>i</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>it</td>
</tr>
</tbody>
</table>
```

First current
```
times  of  best  the  was  it  null
```

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

### Iterators

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

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

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

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

```
"foreach" statement (shorthand)
for (String s : stack)
  StdOut.println(s);
```

```
equivalent code (longhand)
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
  String s = i.next();
  StdOut.println(s);
}
```

java.lang.Iterable interface
java.util.Iterator interface

http://algs4.cs.princeton.edu
ROBERT SEDGEWICK  |  KEVIN WAYNE
Algorithms
‣ stacks
‣ resizing arrays
‣ queues
‣ generics
‣ iterators
‣ applications
1.3  BAGS, QUEUES, AND STACKS
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 void remove() {
            throw UnsupportedOperationException;
        }

        public Item next() {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

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() {
            throw UnsupportedOperationException;
        }

        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.

```
 concurrent modification
 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.
Java collections library

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

```
public interface List<Item> extends Iterable<Item>
```

- create an empty list
- is the list empty?
- number of items
- append item to the end
- return item at given index
- return and delete item at given index
- does the list contain the given item?
- 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.

Java collections library

java.util.Queue.

An interface, not an implementation of a queue.

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

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!
# 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>();
        String s = StdIn.readString();
        if (s.equals("")) {
            while (!StdIn.isEmpty()) {
                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());
                else vals.push(Double.parseDouble(s));
            } StdOut.println(vals.pop());
        }
    }
}
```

% java Evaluate
```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

101.0

---

# 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 ) * ( 4 * 5 ) ) )
```

as if the original input were:

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

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

---

# 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, ...