1.3 Bags, Queues, and Stacks

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

→ Abstract Data Types
→ Stacks and Queues *
→ S&W 1.2 and 1.3
ABSTRACT DATA TYPES
A Data Type is

→ A set of values
  → in example: all counter objects at state 0, 1, 2, ...
→ A set of operations on those values
  → in example: constructor, increment, tally, toString

Example:

```
public class Counter

  Counter(String id)  // create a counter named id
  void increment()    // increment the counter by one
  int tally()         // number of increments since creation
  String toString()   // string representation
```
A **Data Type** is

→ A set of values
  → in example: all counter objects at state 0, 1, 2, ...

→ A set of operations on those values
  → in example: `constructor`, `increment`, `tally`, `toString`

An **Abstract Data Type (ADT)** is

→ A Data Type whose implementation is unknown to the **client** of the ADT
ABSTRACT DATA TYPE

Example:

A **Data Type** is

→ A set of values
  → in example: all counter objects at state 0, 1, 2, ...
→ A set of operations on those values
  → in example: `constructor`, `increment`, `tally`, `toString`

An **Abstract Data Type (ADT)** is

→ A Data Type whose implementation is unknown to the **client** of the ADT

An **Application Programming Interface (API)** is

→ A list and informal description of the operations of an ADT
Example:

→ **Client:** the rest of the program, using the ADT
Stacks & Queues
STACKS & QUEUES
Stacks and queues

Fundamental data types.

- Value: collection of objects.
- Intent is clear when we insert.
- Which item do we remove?

![Diagram of stack and queue](image)

**Stack.** Examine the item most recently added. \(\rightarrow\) LIFO = "last in first out"

**Queue.** Examine the item least recently added. \(\leftarrow\) 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 \( \Rightarrow \)
  client has many implementation from which to choose.
• Implementation can't know details of client needs \( \Rightarrow \)
  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.

```
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.
How to implement a stack with a linked list?

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

B. ![Diagram with moves from it to null]

C. ![Diagram with moves from of to null]
Stack: linked-list implementation

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

```
  top of stack
     ↓
   of → best → the → was → it → null
     ↑
  first
```
http://dsvproject.github.io/dsvproject/code/stackLinkedList.html
inner class
private class Node
{
    String item;
    Node next;
}

save item to return
String item = first.item;

delete first node
first = first.next;

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

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

```java
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).
How to implement a fixed-capacity stack with an array?

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

B. 

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></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>
</tr>
</tbody>
</table>

C. 

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>it</th>
<th></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>
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]
http://dsvproject.github.io/dsvproject/code/stackArray.html
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]; }
}
```

use to index into array; then increment N

decrement N; then use to index into array

a cheat (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.

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

```
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 outstanding references
1.3 BAGS, QUEUES, AND STACKS

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

Challenge. Ensure that array resizing happens infrequently.
We start with an empty array of size 1

Array accesses (AC):

→ 1st push: 1 AC (store new item)
→ 2nd push: 1 AC + 2 AC (read-write previous item(s) to new array)
→ 3rd push: 1 AC + 4 AC
→ 4th push: 1 AC + 6 AC
→ ... 
→ Nth push: 1 AC + 2(N − 1) AC

$\sim N^2$ array accesses to insert N items starting from the empty stack
Stack: resizing-array implementation

**Problem.** Requiring client to provide capacity does not implement API!

**Q.** How to grow and shrink array?

**First try.**
- `push(A)`: increase size of array `s[]` by 1.
- `pop(A)`: 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 = \( N + (2 + 4 + \ldots + 2(N - 1)) \) \( \sim N^2 \).

**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 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. \[ N + (2 + 4 + 8 + \ldots + N) \sim 3N. \]
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></th>
<th></th>
<th></th>
<th></th>
<th>to</th>
<th>be</th>
<th>or</th>
<th>not</th>
<th>to</th>
<th>null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N = 4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$N = 5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
<td>to</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>$N = 4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.
AMORTIZED RUNNING TIME

Assume implementation of ADT with operations A, B, C

How do we calculate the amortized running time of the operations of the ADT?

→ Consider the ADT to be empty
→ Consider all feasible sequences of N operations
  → A, A, A, ...
  → A, B, A, C, ...
  → ...
→ calculate total running time for each of these sequences take the largest one (worst case sequence)
→ Amortized running time of ADT operations = worst-case running time of N operations / N
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>

doubling and halving operations

order of growth of running time for resizing stack with $N$ items
Operations (push/pop/construct/resize) have $O(1)$ amortized running time
Stack resizing-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 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).
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

```
| to | be | or | not | null | null | null | null |
```

```
first

not ────> or ────> be ────> to
```
1.3 Bags, Queues, and Stacks

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

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
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 → null
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`.
Visualisation:
http://www.cs.usfca.edu/~galles/visualization/QueueLL.html
Queue dequeue: linked-list implementation

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

**save item to return**

```java
String item = first.item;
```

**delete first node**

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

**return saved item**

```java
return item;
```

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

```java
class Node {
    String item;
    Node next;
}
...
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;
        if (isEmpty()) last = null;
        return item;
    }
}
How to implement a fixed-capacity queue with an array?

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

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>front of queue</td>
<td>back of queue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>it</td>
<td>was</td>
<td>the</td>
<td>best</td>
<td>of</td>
<td>times</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<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>
</tr>
</tbody>
</table>

B.

C.
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.

Q. How to resize?