1.3 **Stacks and Queues**

- stacks
- resizing arrays
- queues
- generics
- iterators

[see next lecture and precept](https://algs4.cs.princeton.edu)
Stacks and queues

Fundamental data types.

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

**Stack.** Remove the item most recently added. \[\text{LIFO} = \text{“last in first out”}\]

**Queue.** Remove the item least recently added. \[\text{FIFO} = \text{“first in first out”}\]
Programming assignment 2

Deque. Remove either the most recently or the least recently added item.

Randomized queue. Remove a random item.

Your job.

- Identify a data structure that meets the performance requirements.
- Implement it from scratch.
Data type design: API, client, and implementation

Separate client and implementation via API.

**API**: operations that characterize the behavior of a data type.

**Client**: code that uses a data type through its API.

**Implementation**: code that implements the API operations.

**Benefits.**

- **Design**: develop and maintain reusable code.
- **Performance**: substitute faster implementations.

**Ex.** Stack, queue, priority queue, symbol table, set, union–find, ...
1.3 Stacks and Queues

- stacks
- resizing arrays
- queues
- generics
- iterators

https://algs4.cs.princeton.edu
Stack API

**Warmup API.** Stack of strings data type.

```java
public class StackOfStrings
{
    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
}
```

**Performance goals.** Every operation takes $\Theta(1)$ time; stack with $n$ items uses $\Theta(n)$ memory.

**Warmup client.** Reverse a stream of strings from standard input.
Function-call stack demo

```java
public static double square(double a) {
    return a*a;
}
```

```
public static double hypotenuse(double a, double b) {
    return Math.sqrt(square(a) + square(b));
}
```

```
public static void main(String[] args) {
    double a = 3.0;
    double b = 4.0;
    double c = hypotenuse(a, b);
    System.out.println("hypotenuse = " + c);
}
```
How to implement efficiently a stack with a singly linked list?

A. least recently added

```
I -> have -> a -> dream -> today -> null
```

B. most recently added

```
today -> dream -> a -> have -> I -> null
```

C. Both A and B.

D. Neither A nor B.
Stack: linked-list implementation

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

```
! → today → dream → a → have → I → null
```

_{most recently added}_
Stack pop: linked-list implementation

singly linked list

save item to return

String item = first.item;

delete first node

first = first.next;

return saved item

return item;

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

nested class

garbage collector reclaims memory when no remaining references
Stack push: linked-list implementation

save a link to the list
Node oldFirst = first;

create a new node at the front
first = new Node();

initialize the instance variables in the new Node
first.item = "dream";
first.next = oldFirst;

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

nested class
Stack: linked-list implementation

```java
public class LinkedListStackOfStrings {
    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;
    }
}
```

private nested class (access modifiers for instance variables of such a class don’t matter)

no Node constructor explicitly defined ⇒ Java supplies default no-argument constructor
**Stack: linked-list implementation performance**

**Proposition.** Every operation takes $\Theta(1)$ time.

**Proposition.** A `LinkedStackOfStrings` with $n$ items has $n$ `Node` objects and uses $\sim 40n$ bytes.

**Remark.** This counts the memory for the stack itself, including the string references. 

[ but not the memory for the string objects, which the client allocates ]
How to implement efficiently a fixed-capacity stack with an array?

A.  
least recently added

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>have</th>
<th>a</th>
<th>dream</th>
<th>today</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>

B.  
most recently added

<table>
<thead>
<tr>
<th></th>
<th>!</th>
<th>today</th>
<th>dream</th>
<th>a</th>
<th>have</th>
<th>I</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>null</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

C.  
Both A and B.

D.  
Neither A nor B.
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]
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

**Underflow.** Throw exception if `pop()` called when stack is empty.

**Overflow.** Use “resizing array” for array implementation. [next section]

**Null items.** We allow null items to be added.

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

<table>
<thead>
<tr>
<th>I</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>have</td>
<td>have</td>
<td>a</td>
<td>dream</td>
<td>today</td>
<td>!</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

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

loitering

```java
public String pop() {
    String item = s[n-1];
    s[n-1] = null;
    n--;
    return item;
}
```

no loitering
1.3 Stacks and Queues

- stacks
- resizing arrays
- queues
- generics
- iterators
Stack: resizing-array implementation

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

Q. How to grow and shrink array?

Naive approach.
- Push: increase length of array \( s[] \) by 1.
- Pop: decrease length of array \( s[] \) by 1.

Too expensive.
- Need to copy all items to a new array, for each push/pop.
- Array accesses to add item \( k \): \( 1 + 2(k - 1) \)
- Array accesses to add first \( n \) items: \( n + (0 + 2 + 4 + 6 + \ldots + 2(n - 1)) \sim n^2 \).

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 length, and copy items.

```java
public class ResizingArrayStackOfStrings {
    private String[] s;
    private int n = 0;

    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;
    }
}
```

“repeated doubling”

double size of array if full

helper method does the resizing
Stack: resizing-array implementation

Q. How to grow array?

A. If array is full, create a new array of twice the length, and copy items.

Cost is reasonable.

• Still need to copy all items to a new array but, now, only infrequently.
• Total array accesses to add first \( n = 2^i \) items: \( n + (2 + 4 + 8 + 16 + \ldots + n) \sim 3n \).

“repeated doubling”
Stack: resizing-array implementation

Q. How to shrink array?

First try.

- Push: double length of array $s[]$ when array is full.
- Pop: halve length of array $s[]$ when array is one-half full.

Too expensive for some sequences of operations.

- Consider alternating sequence of push and pop operations when array is full.
- Each operation takes $\Theta(n)$ time.
Stack: resizing-array implementation

Q. How to shrink array?

Efficient solution.

- Push: double length of array \( s[] \) when array is full.
- Pop: halve length 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;
}
```

so, on average, each operation takes \( \Theta(1) \) time

Proposition. Starting from an empty stack, any sequence of \( m \) push/pop operations takes \( \Theta(m) \) time.
Amortized analysis

Worst-case analysis. Worst-case running time for an individual operation.

Amortized analysis. Worst-case running time for a sequence of operations.
- Amortized cost per operation = total cost / # operations.
- Provides more realistic analysis when some operations are expensive but rare.

Bob Tarjan
(1986 Turing award)

<table>
<thead>
<tr>
<th></th>
<th>worst</th>
<th>amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>construct</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>push</td>
<td>$n$</td>
<td>1</td>
</tr>
<tr>
<td>pop</td>
<td>$n$</td>
<td>1</td>
</tr>
<tr>
<td>size</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

order of growth of running time for resizing-array stack with $n$ items
Stack resizing-array: memory usage

Proposition. A ResizingArrayStackOfStrings with $n$ items use between $\sim 8n$ and $\sim 32n$ bytes of memory.
   • Always between 25% and 100% full.
   • $\sim 8n$ when full. [ array length = $n$ ]
   • $\sim 32n$ when one-quarter full. [ array length = $4n$ ]

Remark. This counts the memory for the stack itself, including the string references.
   [ but not the memory for the string objects, which the client allocates ]
Tradeoffs. Can implement a stack with either resizing array or linked list. Which is better?

Linked-list implementation.
- Stronger performance guarantee.
- More memory.

Resizing-array implementation.
- Weaker performance guarantee.
- Less memory.
- Better use of cache.

accessing adjacent memory locations (e.g., in an array) is much faster than accessing nonadjacent memory locations (e.g., in a linked list)
1.3 Stacks and Queues

- stacks
- resizing arrays
- queues
- generics
- iterators
Queue of strings API

```
public 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
    boolean isEmpty() is the queue empty?
    int size() number of strings on the queue
```

Performance goals. Every operation takes $\Theta(1)$ time; queue with $n$ items uses $\Theta(n)$ memory.
How to implement efficiently a queue with a singly linked list?

A. 

least recently added

I -> have -> a -> dream -> today -> null

most recently added

B. 

most recently added

today -> dream -> a -> have -> I -> null

least recently added

C. Both A and B.

D. Neither A nor B.
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

**Remark.** Code is identical to `pop()`.  

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

singly linked list

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

delete first node

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

return saved item

```java
return item;
```
Queue enqueue: linked-list implementation

save a link to the list
Node oldLast = last;

create a new node at the end
last = new Node();
last.item = "dream";

link together
oldLast.next = last;

```java
private class Node {
    private String item;
    private Node next;
}
```
Queue: linked-list implementation

```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;
        if (isEmpty()) last = null;
        return item;
    }
}
```

- **corner case**: add to an empty queue
- **corner case**: remove down to an empty queue
Goal. Implement a queue using a resizing array so that, starting from an empty queue, any sequence of \( m \) operations takes \( \Theta(m) \) time.
**Goal.** Implement a **queue** using a **resizing array** so that, starting from an empty queue, any sequence of \( m \) operations takes \( \Theta(m) \) time.
1.3 **Stacks and Queues**

- stacks
- resizing arrays
- queues
- generics
- iterators
Parameterized stack

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

Solution in Java: generics.
Guiding principle: prefer compile-time errors to run-time errors.

```java
Stack<Apple> stack = new Stack<Apple>();
Apple apple = new Apple();
stack.push(apple);
Orange orange = new Orange();
stack.push(orange);
...  // compile-time error
```
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: linked-list implementation

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;
    }
}
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 array 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];
    }
}
Generic stack: array implementation

The way it should be.

```java
public class FixedCapacityStackOfStrings {
    private String[] s;
    private int n = 0;

    public FixedCapacityOfStrings(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];
    }
}
```

```
public class FixedCapacityStack<Item> {
    private Item[] s;
    private int n = 0;

    public FixedCapacityStack(int capacity) {
        s = (Item[]) new Object[capacity];
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(Item item) {
        s[n++] = item;
    }

    public Item pop() {
        return s[--n];
    }
}
```

stack of strings (fixed-length array)
generic stack (fixed-length array)
Q. Why does Java require a cast (or reflection)?

Short answer. Backward compatibility.

Long answer. Need to learn about type erasure and covariant arrays.
How to declare and initialize an empty stack of integers in Java?

A. Stack stack = new Stack\<int\>();

B. Stack\<int\> stack = new Stack();

C. Stack\<int\> stack = new Stack\<int\>();

D. None of the above.
Q. What to do about primitive types?

Wrapper type.
- Each primitive type has an associated “wrapper” reference type.
- Ex: `Integer` is wrapper type associated with `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.
Caveat. Performance overhead for primitive types.
Java collections framework

Java's library of collection data types.

- `java.util.LinkedList`  
  [doubly linked list]
- `java.util.ArrayList`  
  [resizing array]

This course. Implement from scratch (once).

Beyond. Basis for understanding performance guarantees.

Best practices.

- Use `Stack` and `Queue` in `algs4.jar` for stacks and queues to improve design and efficiency.
- Use `java.util.ArrayList` or `java.util.LinkedList` when other ops needed.
  (but remember that some ops are inefficient)
Stacks and queues summary

Fundamental data types.

• Value: collection of objects.
• Operations: add, remove, iterate, size, test if empty.

Stack. [LIFO] Remove the item most recently added.
Queue. [FIFO] Remove the item least recently added.

Efficient implementations.

• Singly linked list.
• Resizing array.

Next time. Advanced Java features (including iterators for collections).
<table>
<thead>
<tr>
<th>image</th>
<th>source</th>
<th>license</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment Logo</td>
<td>Kathleen Ma ’18</td>
<td>by author</td>
</tr>
<tr>
<td>Stack of Books</td>
<td>Adobe Stock</td>
<td>Education License</td>
</tr>
<tr>
<td>No Loitering</td>
<td>signsworldwide.com</td>
<td></td>
</tr>
<tr>
<td>Bob Tarjan</td>
<td>Heidelberg Laureate</td>
<td></td>
</tr>
<tr>
<td>Long Queue Line</td>
<td>Adobe Stock</td>
<td>Education License</td>
</tr>
<tr>
<td>Stack of Apples</td>
<td>Adobe Stock</td>
<td>Education License</td>
</tr>
<tr>
<td>Stack of Fruit</td>
<td>Adobe Stock</td>
<td>Education License</td>
</tr>
<tr>
<td>Queue of People</td>
<td>Adobe Stock</td>
<td>Education License</td>
</tr>
<tr>
<td>Stack of Sweaters</td>
<td>Adobe Stock</td>
<td>Education License</td>
</tr>
<tr>
<td>ChatGPT Phone</td>
<td>Adobe Stock</td>
<td>Editorial Use</td>
</tr>
</tbody>
</table>
A final thought

“Linked lists, nodes connected with care,
Arrays resizing, with memory to spare.
Organizing data, their only need,
Helping us, with efficiency indeed.”