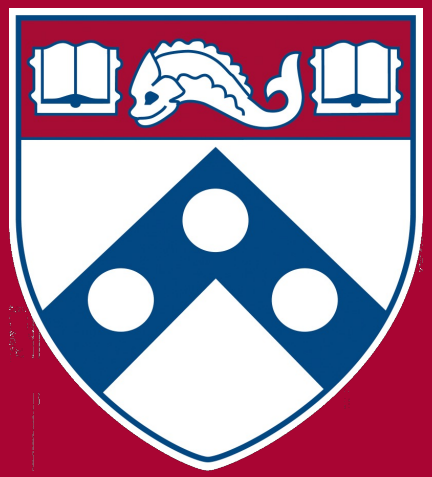
Stacks & Queues

CIT5940

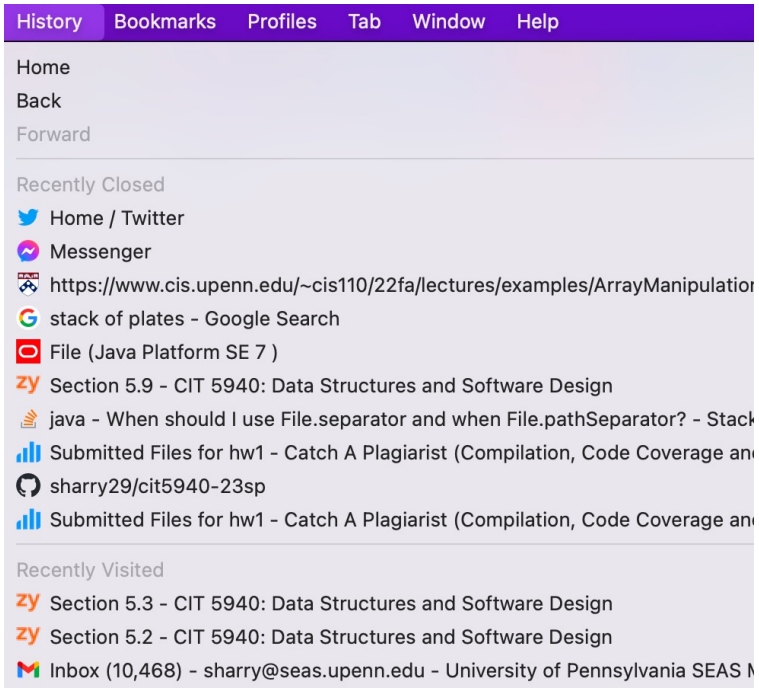


Stacks

- The ***stack*** is a list-like structure in which elements may be inserted or removed from only one end.
- "***LIFO***" list: "Last-In, First-Out”

Stack

- The ***stack*** is a list-like structure in which elements may be inserted or removed from only one end.
- "***LIFO***" list: "Last-In, First-Out”



Activity

Suppose that an intermixed sequence of push and pop operations are performed. The pushes push the integers 0 through 9 in order; the pops print out the return value. Which of the following sequences could not occur?

(a) 4 3 2 1 0 9 8 7 6 5

(b) 4 6 8 7 5 3 2 9 0 1

(c) 2 5 6 7 4 8 9 3 1 0

(d) 4 3 2 1 0 5 6 7 8 9

Activity

Suppose that an intermixed sequence of push and pop operations are performed. The pushes push the integers 0 through 9 in order; the pops print out the return value. Which of the following sequences could not occur?

(a) 4 3 2 1 0 9 8 7 6 5

(b) 4 6 8 7 5 3 2 9 0 1

(c) 2 5 6 7 4 8 9 3 1 0

(d) 4 3 2 1 0 5 6 7 8 9

Stack ADT: Operations

- `push`: push an element onto the top of the stack
- `pop`: remove and return the element at the top of the stack
- `peek`: return a copy of the top element
- `empty`: return true if and only if the stack contains no elements

Linked Stack

- Uses *dynamic memory allocation*
- Elements' value stored in "Node" objects
- `head` variable points to the node at the top of the stack

```
public class LinkedListStack<E>
implements Stack<E> {

    private Node head;
    private int size;


    ...
}
```

Linked Stack: `push`

```
@Override
public E push(E item) {
    Node newNode = new Node(item);
    newNode.next = head;
    head = newNode;
    size++;
    return item;
}
```

-Runtime analysis: ???

Linked Stack: `push`

```
@Override
public E push(E item) {
    Node newNode = new Node(item);
    newNode.next = head;
    head = newNode;
    size++;
    return item;
}
```

-Runtime analysis: $O(1)$

Linked Stack: `pop`

```
@Override
public E pop() {
    if (empty()) {
        throw new EmptyStackException();
    }
    E data = head.data;
    head = head.next;
    size--;
    return data;
}
```

- Runtime analysis: O(??)

Linked Stack: `pop`

```
@Override
public E pop() {
    if (empty()) {
        throw new EmptyStackException();
    }
    E data = head.data;
    head = head.next;
    size--;
    return data;
}
```

- Runtime analysis: $O(1)$

Linked Stack: `peek`

```
@Override
public E peek() {
    if (empty()) {
        throw new EmptyStackException();
    }
    return head.data;
}
```

- Runtime analysis: O(??)

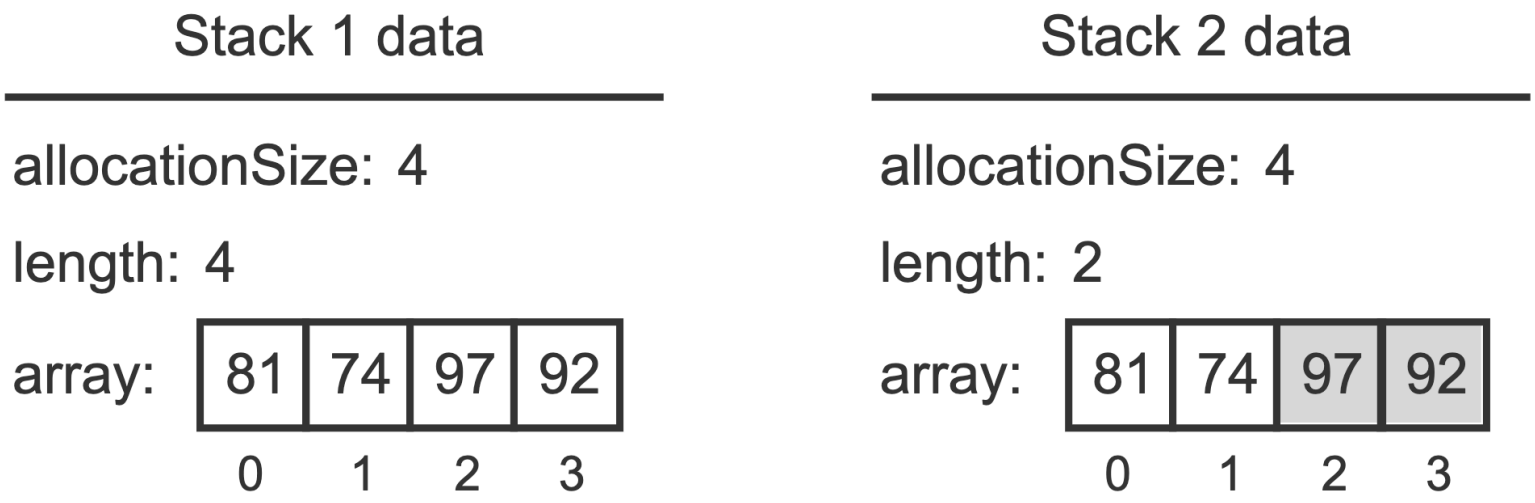
Linked Stack: `peek`

```
@Override
public E peek() {
    if (empty()) {
        throw new EmptyStackException();
    }
    return head.data;
}
```

- Runtime analysis: $O(1)$

Array-Based Stack

- Array is used internally to store elements
- `length` variable keeps track of the position of the element at the top of the stack
- Can be **bounded** or **unbounded** (we'll focus on unbounded)



Array-Based Stack: `push`

```
@Override
public E push(E item) {
    if (length == array.length) {
        grow();
    }
    array[length] = item;
    length++;
    return item;
}
```

- Runtime analysis: $O(1)$

Array-Based Stack: `pop`

```
@Override
public E pop() {
    if (empty()) {
        throw new EmptyStackException();
    }
    E item = (E) array[length - 1];
    length--;
    return item;
}
```

-Runtime analysis: $O(1)$

Array-Based Stack: `peek`

```
@Override
public E peek() {
    if (empty()) {
        throw new EmptyStackException();
    }
    return (E) array[length - 1];
}
```

-Runtime analysis: $O(1)$

Comparison of Stack Implementations

- All operations take constant time for the array-based and linked stack implementations

Stack ADT: Java's Implementations

- Array-based stack: ArrayDeque, Stack
- Linked stack: LinkedList

Activity: Browser Back Button

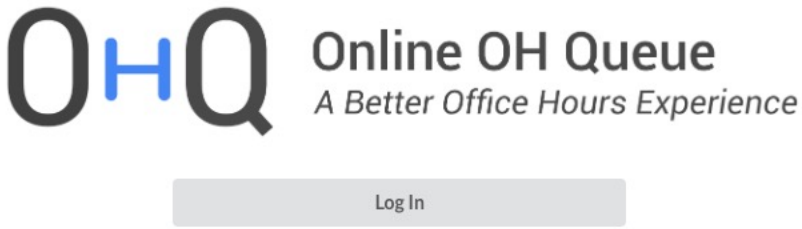
- Complete the **visit** and **back** methods in the BrowserHistory class to mimic the behavior of a browser's back button.

```
public static void main(String[] args) {
    BrowserHistory myHistory = new
BrowserHistory("Google");
    myHistory.visit("Gradescope");
    myHistory.visit("cis.upenn.edu/~cit5940");
    myHistory.visit("cis1100.com");
    myHistory.back();
    myHistory.visit("Canvas");
    myHistory.back();
    myHistory.back();
}
```

```
visiting: Gradescope
visiting: cis.upenn.edu/~cit5940
visiting: cis1100.com
returning to: cis.upenn.edu/~cit5940
visiting: Canvas
returning to: cis.upenn.edu/~cit5940
returning to: Gradescope
```

Queues

- Queue elements may only be:
  - inserted at the back: *enqueue* operation
  - removed from the front: *dequeue* operation
- ”***FIFO***" list: ”First-In, First-Out.”



Queue ADT: Operations

- `Enqueue / offer / push:` insert an element into the queue
- `Dequeue / poll / pop:` remove and return the element at the head of the queue
- `frontValue / peek:` return a copy of the head element

Array-Based Queue

- Circular array is used internally to store elements
- `frontIndex` variable keeps track of the position of the element at the front of the queue. Initialized at 0
- `size` variable tracks the number of elements contained in the queue. Initialized at 0

```
public class ResizingArrayQueue<E> implements Queue<E> {
    private Object[] array;
    private int frontIndex;
    private int size;

    public ResizingArrayQueue() {
        int allocationSize = 1;
        array = new Object[allocationSize];
        size = 0;
        frontIndex = 0;
    }
```

Array-Based Queue: `offer`

```
public boolean offer(E e) {
    // Resize if length equals allocation size
    if (size == array.length) {
        resize();
    }

    // Enqueue the item and return true
    int itemIndex = (frontIndex + size) % array.length;
    array[itemIndex] = e;
    size++;
    return true;
}
```

- Runtime analysis: $O(1)$

Array-Based Queue: `dequeue`

```
public E poll() {
    // Get the item at the front of the queue
    E toReturn = (E) array[frontIndex];

    // Decrement length and advance frontIndex
    size--;
    frontIndex = (frontIndex + 1) % array.length;

    // Return the front item
    return toReturn;
}
```

Runtime analysis: $O(1)$

Linked Queue

- Elements' value stored in "Node" objects
- `front` variable points to the front (node) of the queue
- `rear` variable points to the rear (node) of the queue

Linked Queue: `offer`

```
@Override
public boolean offer(E e) {
    if (head == null) {
        head = new Node(e);
        tail = head;
    } else {
        tail.next = new Node(e);
        tail = tail.next;
    }
    size++;
    return true;
}
```

-Runtime analysis: $O(1)$

Linked Queue: `poll`

```
@Override
public E poll() {
    if (empty()) {
        throw new NoSuchElementException();
    }
    E toReturn = head.data;
    head = head.next;
    size--;
    if (head == null) {
        tail = null;
    }
    return toReturn;
}
```

-Runtime analysis: $O(1)$

Comparison of Queue Implementations

- All operations take constant time for the array-based and linked queue implementations

Queue ADT: Implementations

- Array-based queue: ArrayDeque
- Linked queue: LinkedList

Activity: OHQ

- Write an OHQ class that maintains a Queue of students getting help in OH. When a student joins the queue, let them know what their estimated wait time will be.