CS 10: Problem solving via Object Oriented Programming

Keeping order

Main goals

- Implement stacks and queues

Agenda

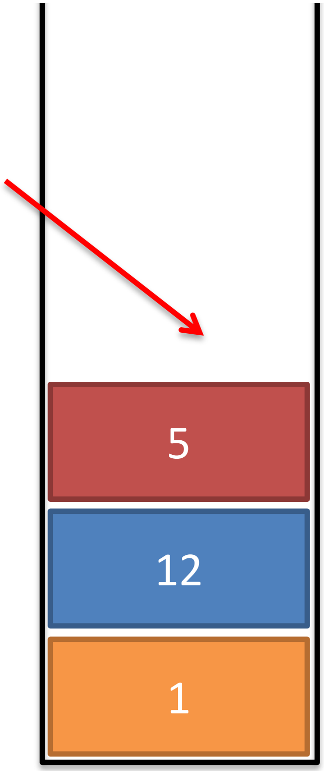
1. Stacks

2. Queues

Stacks add and remove from top, Queues add to back, remove from front

Items inserted in order: 1, 12, 5

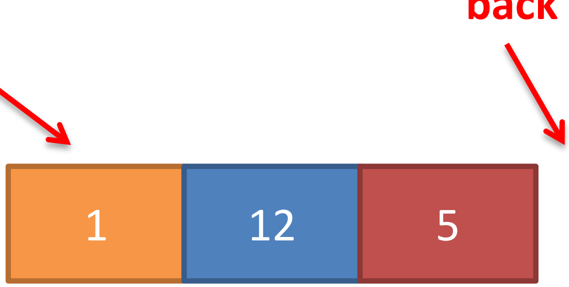
Add and remove from top



Stack (LIFO)

Remove from front

Add at back



Queue (FIFO)

Stacks are a Last In, First Out (LIFO) data structure

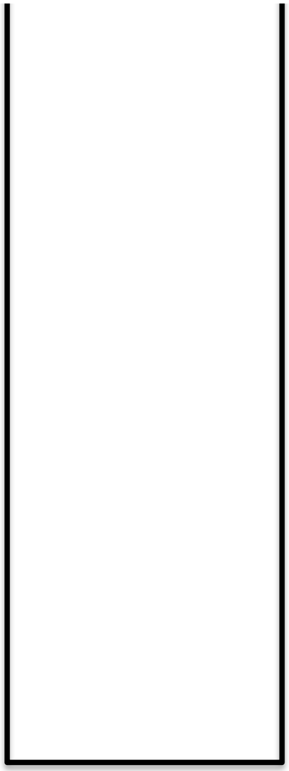
Stack overview

- Think of stack of dinner plates (or Pez dispenser)
- Add item to the top, others move down
- To remove, take top item (last one inserted)
- Commonly used in CS – function calls, parenthesis matching, reversing items in collection…
- **Operations**
  - `push` – add item to top of stack
  - `pop` – remove top item and return it
  - `peek` – return top item, but don’t remove it
  - `isEmpty` – true if stack empty, false otherwise

NOTE: There is no *size* method in a Stack as classically defined (Java’s implementation does have size)

Stack adds to top only, removes from top only; Last In First Out (LIFO)

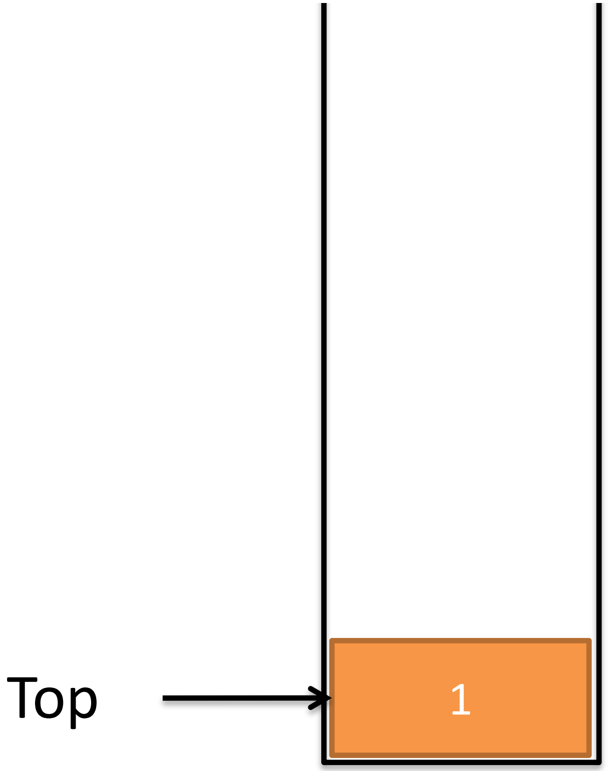
Initially empty



Stack

Stack adds to top only, removes from top only; Last In First Out (LIFO)

push(1)



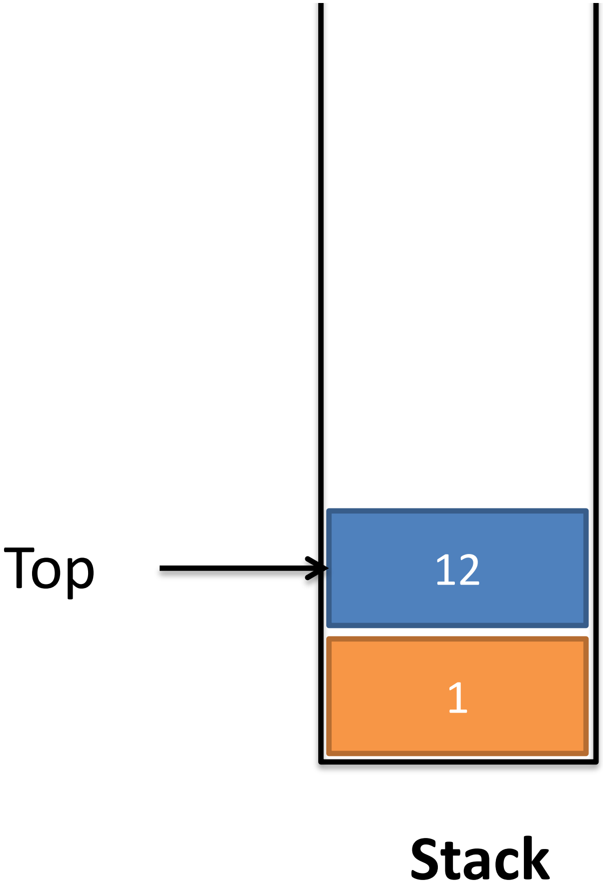
Stack

Operations

Push 1

Stack adds to top only, removes from top only; Last In First Out (LIFO)

push(12)



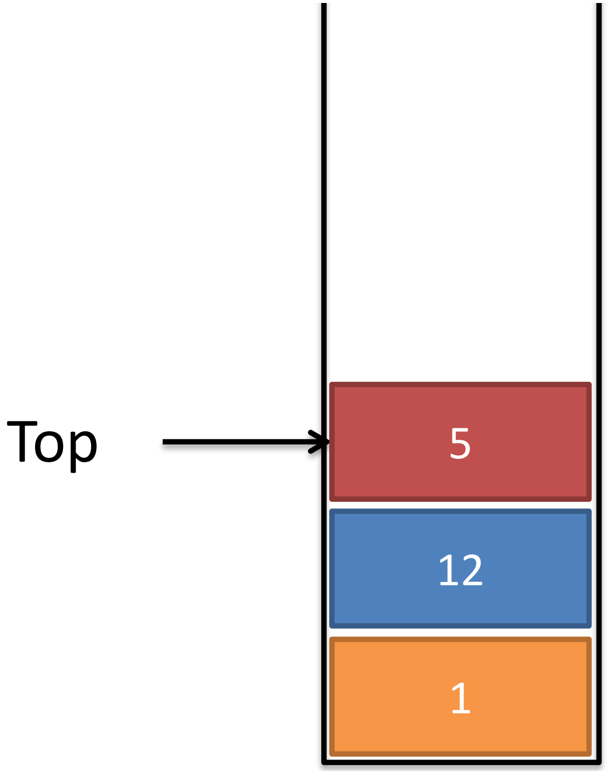
Operations

Push 1

Push 12

Stack adds to top only, removes from top only; Last In First Out (LIFO)

push(5)



Stack

Operations

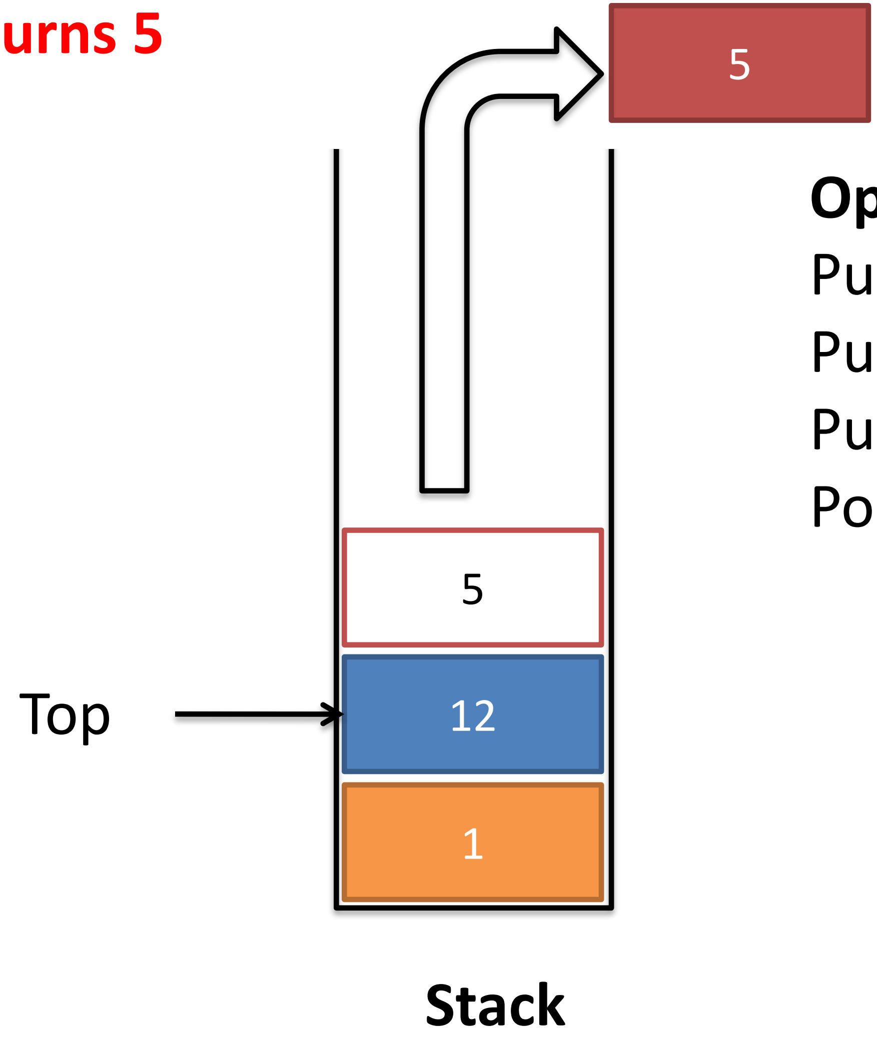
Push 1

Push 12

Push 5

Stack adds to top only, removes from top only; Last In First Out (LIFO)

pop() –> returns 5



Operations

Push 1

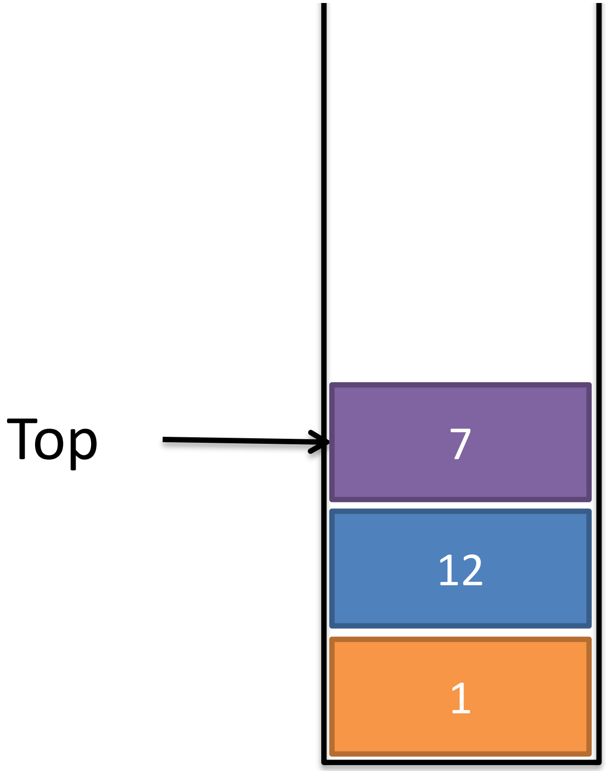
Push 12

Push 5

Pop – returns 5

Stack adds to top only, removes from top only; Last In First Out (LIFO)

push(7)



Stack

Operations

Push 1

Push 12

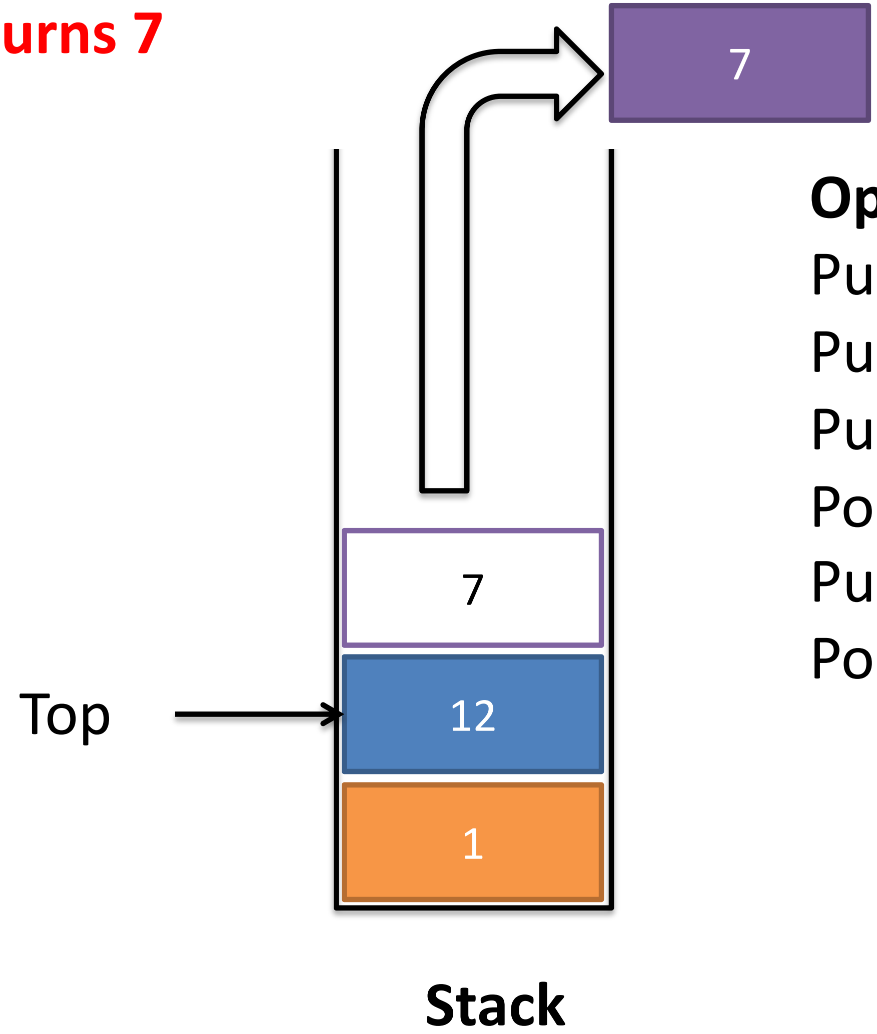
Push 5

Pop – returns 5

Push 7

Stack adds to top only, removes from top only; Last In First Out (LIFO)

pop() –> returns 7



Operations

Push 1

Push 12

Push 5

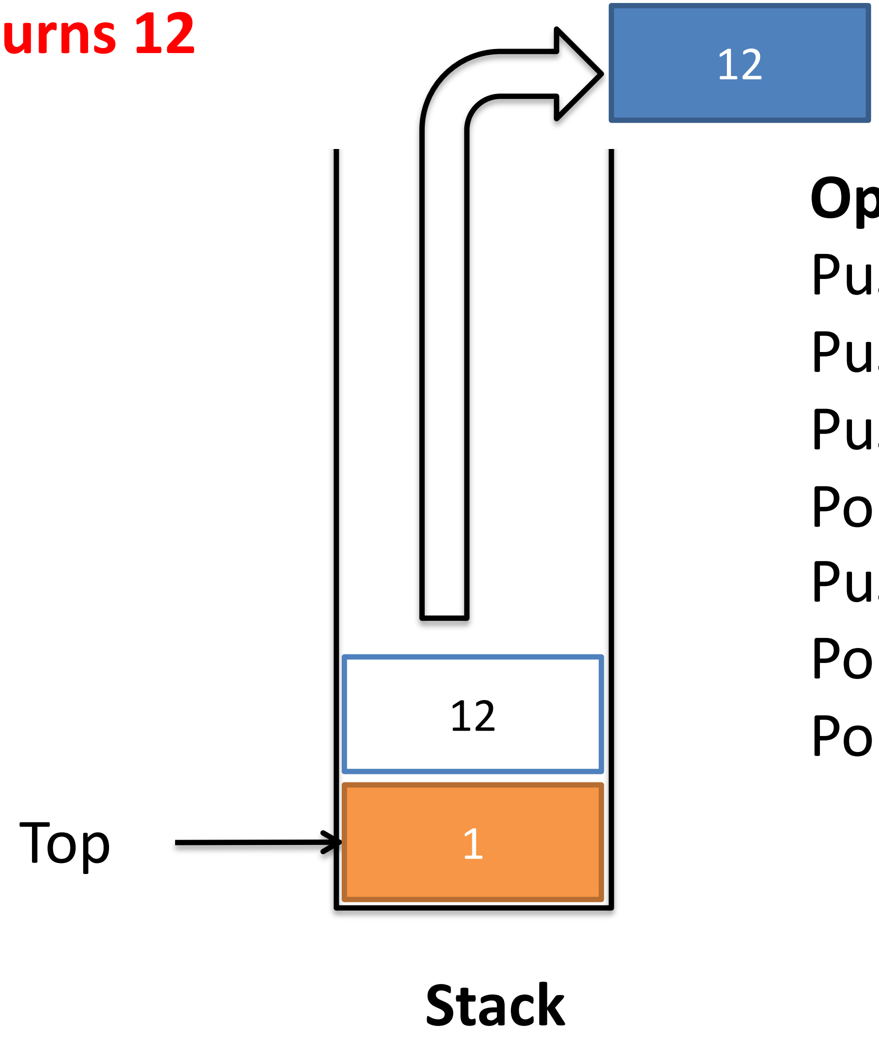
Pop – returns 5

Push 7

Pop – returns 7

Stack adds to top only, removes from top only; Last In First Out (LIFO)

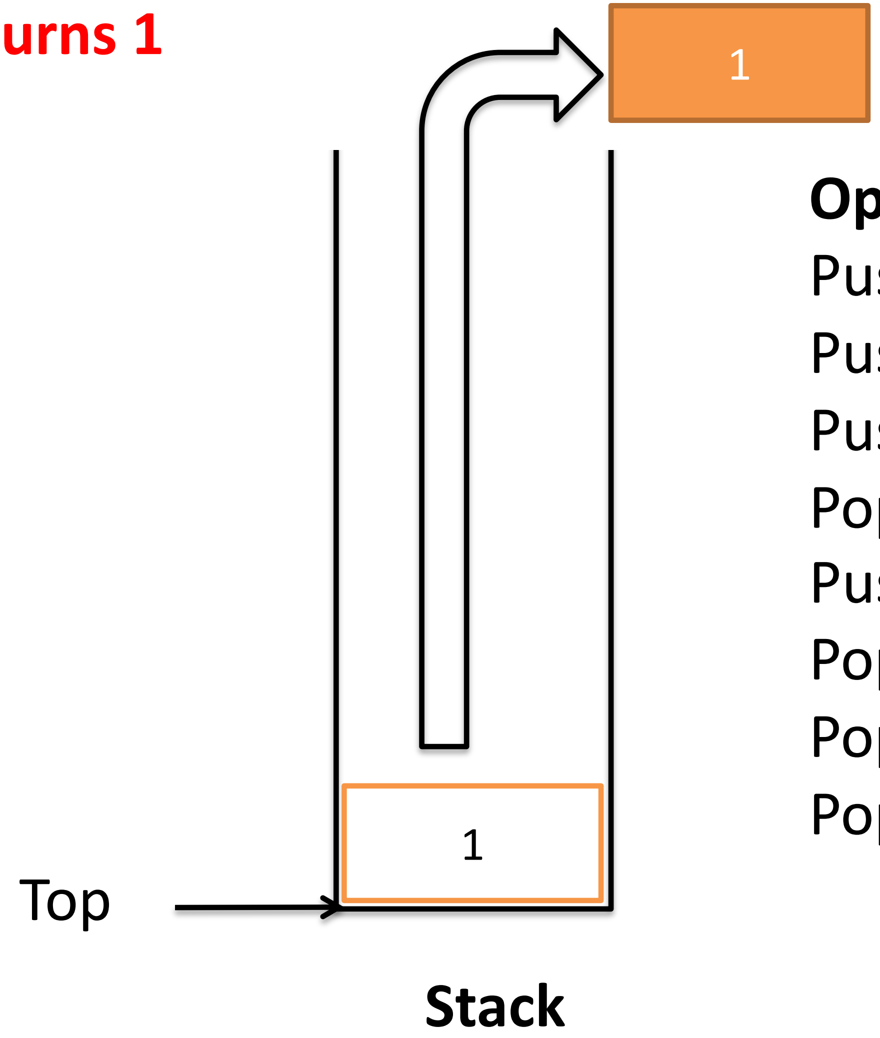
pop() –> returns 12



Operations
Push 1
Push 12
Push 5
Pop – returns 5
Push 7
Pop – returns 7
Pop – returns 12

Stack adds to top only, removes from top only; Last In First Out (LIFO)

pop() –> returns 1



Operations

Push 1

Push 12

Push 5

Pop – returns 5

Push 7

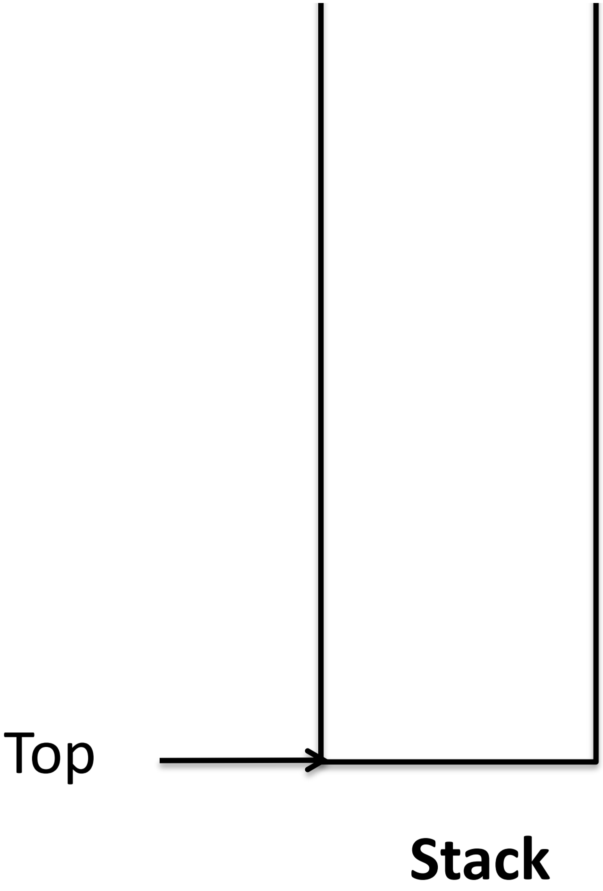
Pop – returns 7

Pop – returns 12

Pop – returns 1

Stack adds to top only, removes from top only; Last In First Out (LIFO)

pop() –> throw exception



Operations

Push 1

Push 12

Push 5

Pop – returns 5

Push 7

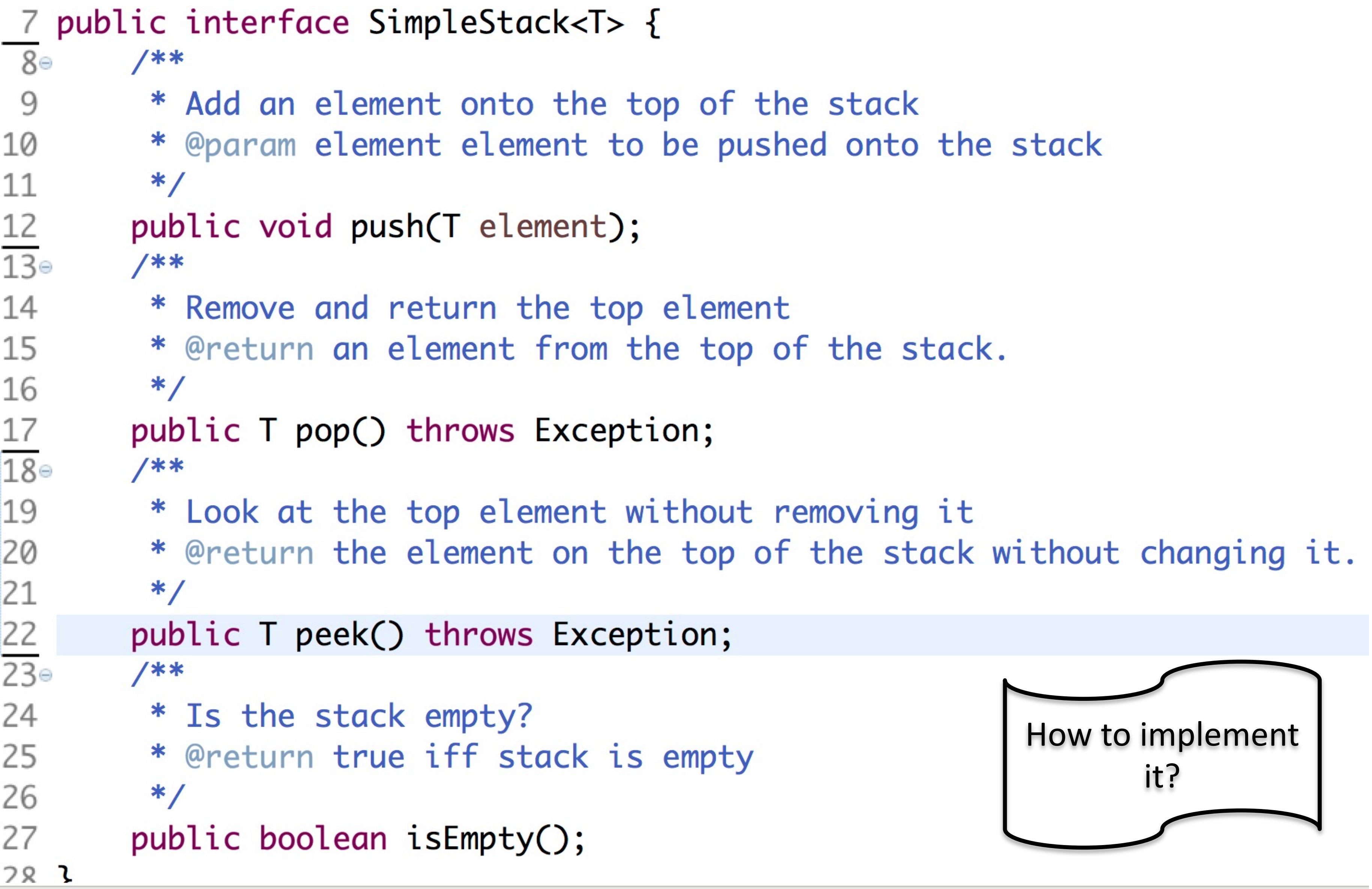
Pop – returns 7

Pop – returns 12

Pop – returns 1

Pop – throw exception

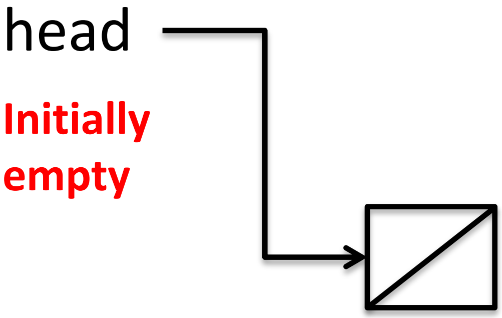
SimpleStack.java: Interface defining Stack operations



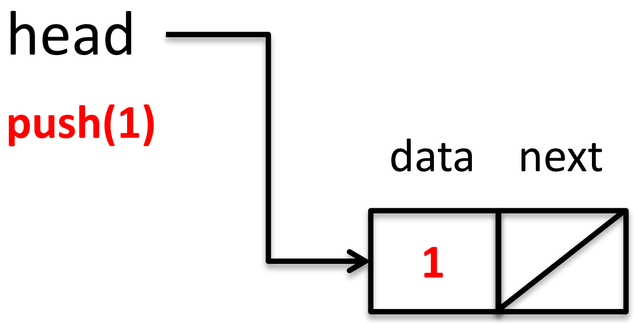
```java
public interface SimpleStack<T> {
    /**
     * Add an element onto the top of the stack
     * @param element element to be pushed onto the stack
     */
    public void push(T element);
    /**
     * Remove and return the top element
     * @return an element from the top of the stack.
     */
    public T pop() throws Exception;
    /**
     * Look at the top element without removing it
     * @return the element on the top of the stack without changing it.
     */
    public T peek() throws Exception;
    /**
     * Is the stack empty?
     * @return true iff stack is empty
     */
    public boolean isEmpty();
}
```

How to implement it?

A Singly Linked List works well for a Stack, using top as head of list



A Singly Linked List works well for a Stack, using top as head of list

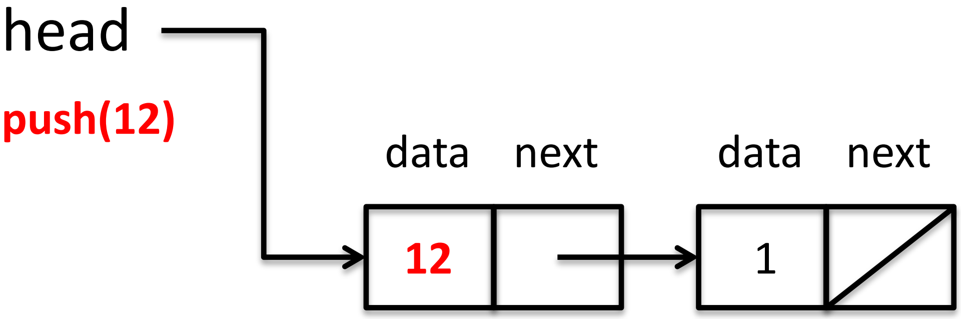


Add at front of linked list

Set new element *next* to *head* (null)

Set *head* to new element (1)

A Singly Linked List works well for a Stack, using top as head of list

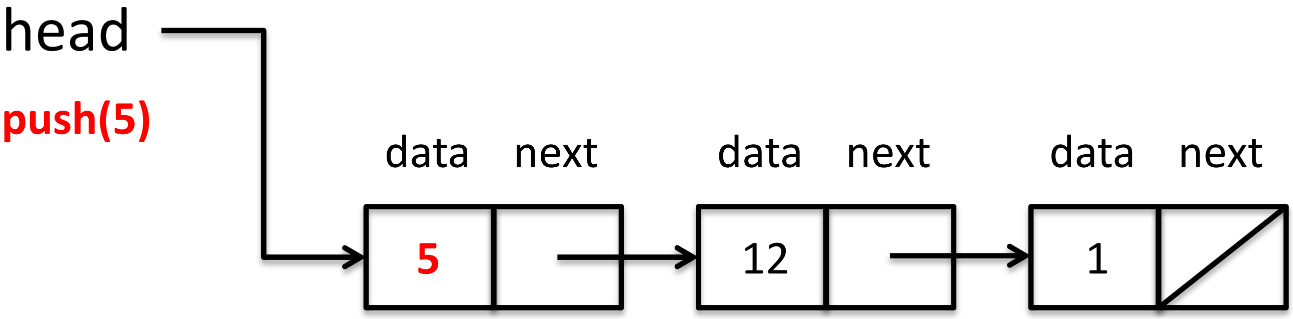


Add at front of linked list

Set new element *next* to *head* (1)

Set *head* to new element (12)

A Singly Linked List works well for a Stack, using top as head of list

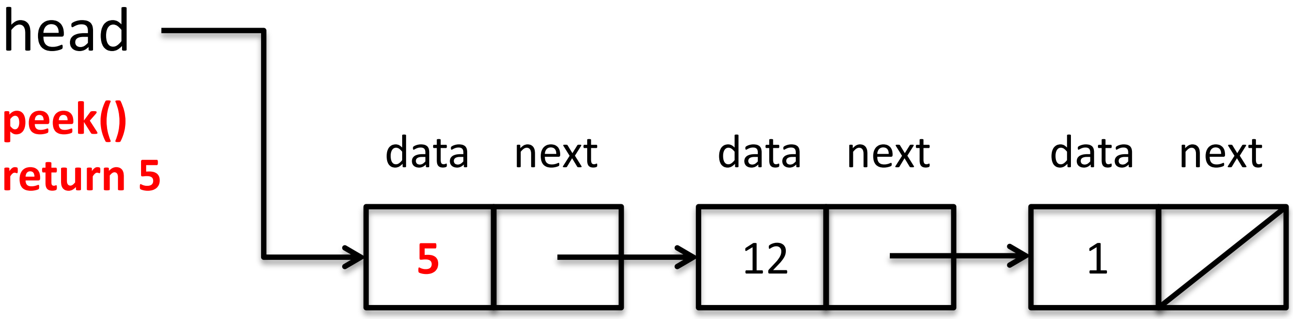


Add at front of linked list

Set new element *next* to *head* (12)

Set *head* to new element (5)

A Singly Linked List works well for a Stack, using top as head of list



head

peek()
return 5

data next

5

data next

12

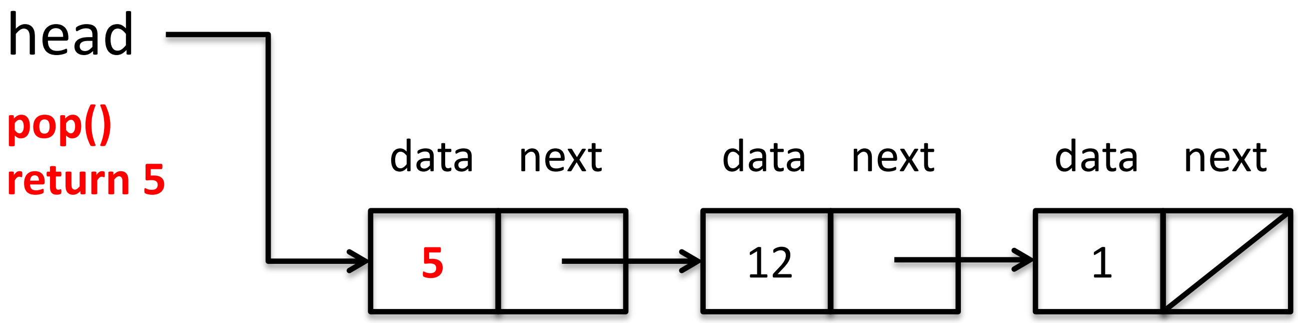
data next

1

Add at front of linked list

Peek returns data from first element or throw exception if empty

A Singly Linked List works well for a Stack, using top as head of list



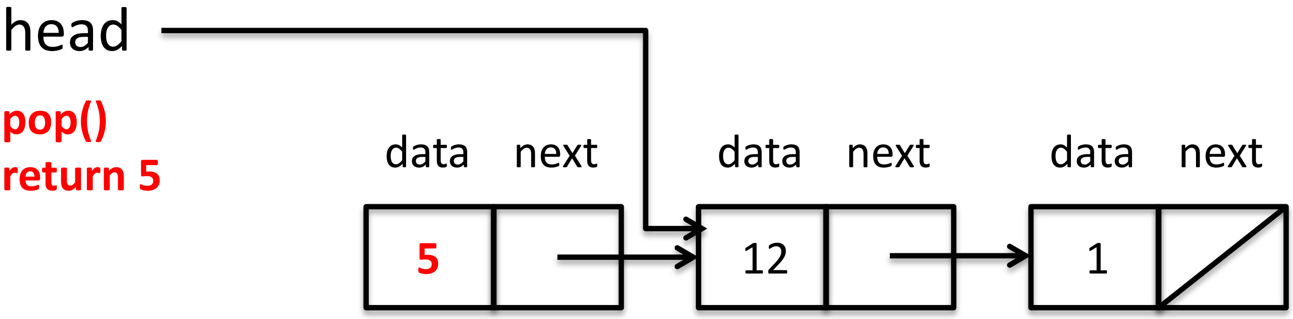
Add at front of linked list

Peek returns data from first element or throw exception if empty

Pop from front of list

Get *data* from *head* (5)

A Singly Linked List works well for a Stack, using top as head of list



Add at front of linked list

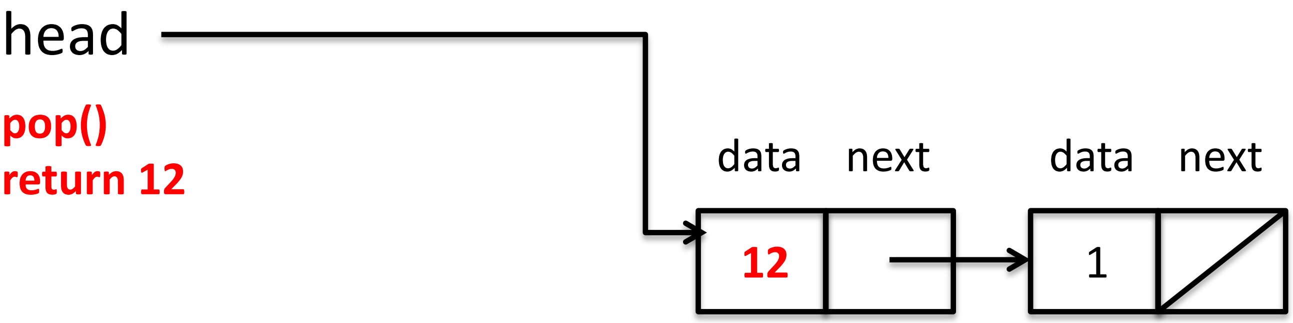
Peek returns data from first element or throw exception if empty

Pop from front of list

Get *data* from *head* (5)

Set *head* = *head.next* (12)

A Singly Linked List works well for a Stack, using top as head of list



pop()
return 12

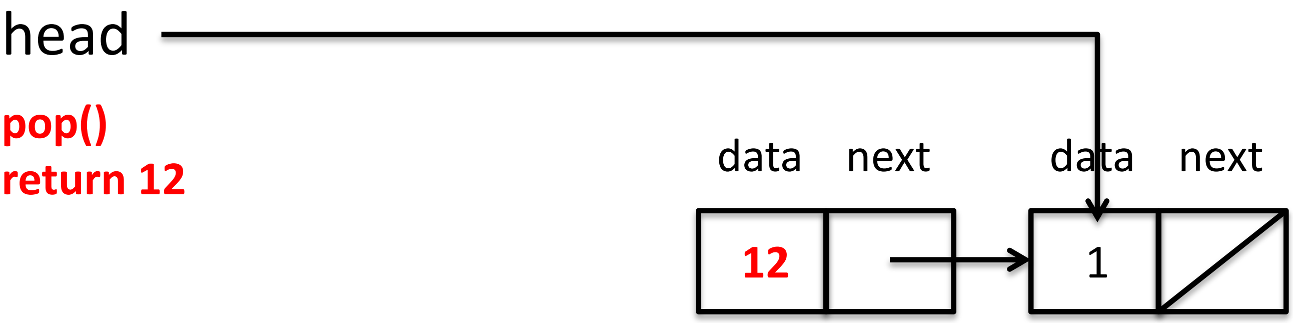
Add at front of linked list

Peek returns data from first element or throw exception if empty

Pop from front of list

Get *data* from *head* (12)

A Singly Linked List works well for a Stack, using top as head of list

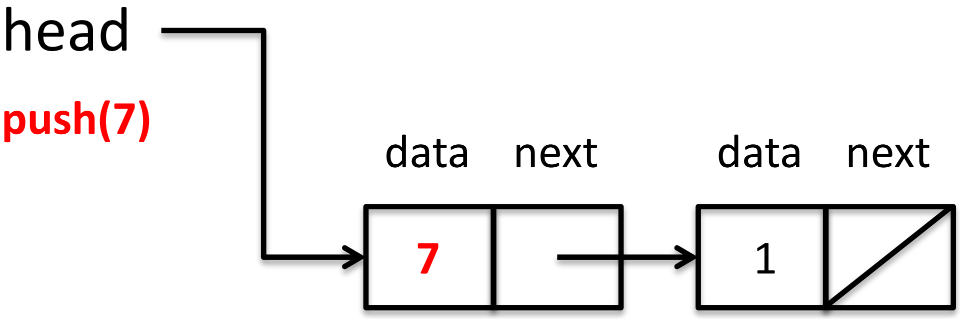


pop()
return 12

Add at front of linked list
Peek returns data from first element or throw exception if empty
Pop from front of list

Get *data* from *head* (12)
Set *head* = *head.next* (1)

A Singly Linked List works well for a Stack, using top as head of list



Add at front of linked list

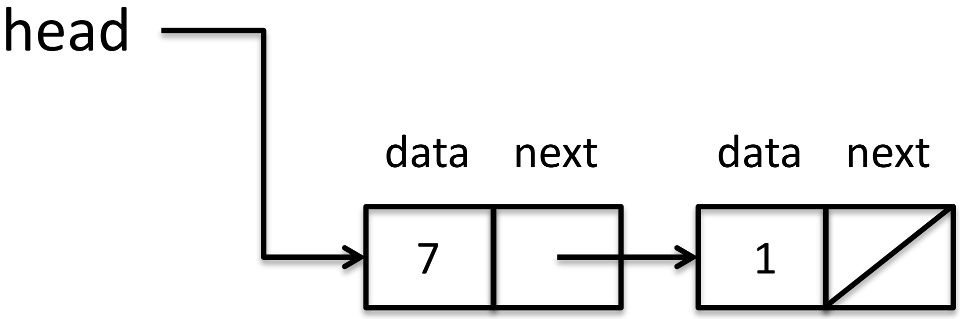
Set new element *next* to *head* (1)

Set *head* to new element (7)

Peek returns data from first element or throw exception if empty

Pop from front of list

A Singly Linked List works well for a Stack, using top as head of list



Add at front of linked list
Peek returns data from first element or throw exception if empty
Pop from front of list

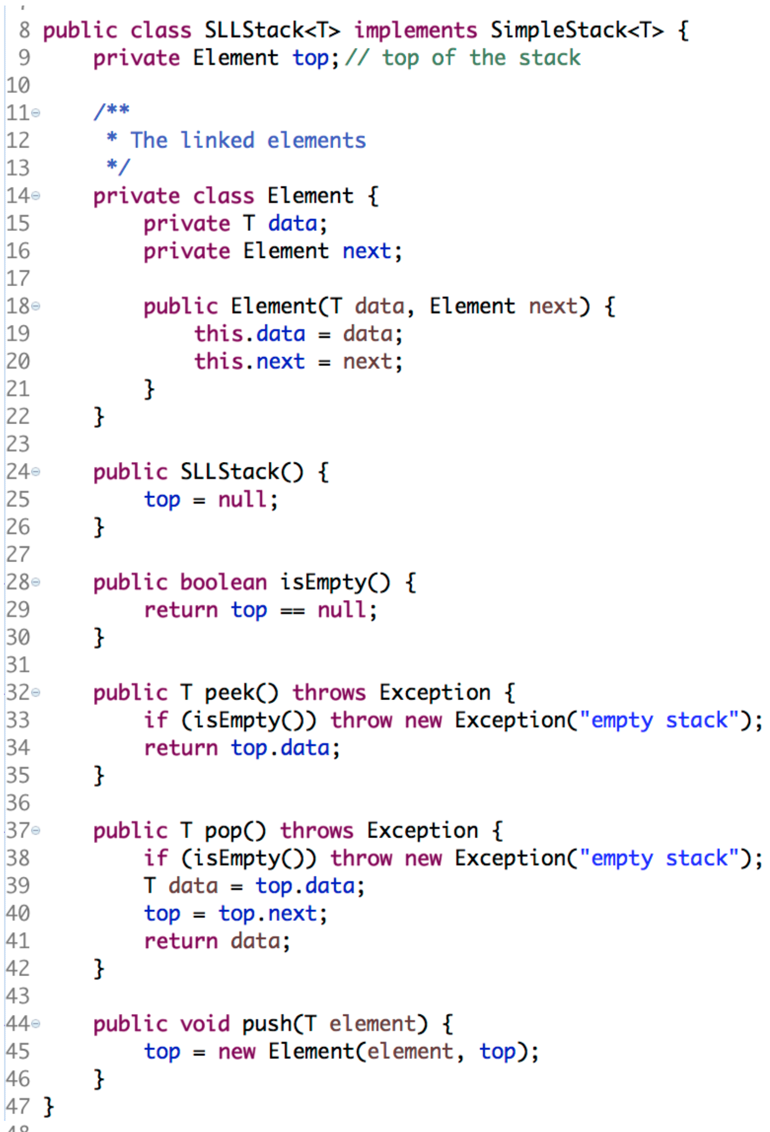
Always operating from *head*
Never need to traverse list
All operations Θ(1)

If you had a tail pointer, could you implement a Stack by adding at the tail?

- **Adding at tail is easy (you did so in SA-4)**
- **How would you handle *pop*?**
- **No easy way to move tail pointer back one element**
- **Could use a doubly linked list, but easy to implement Stack with singly linked list by operating at head**

A Singly Linked List works well for a Stack, using top as head of list

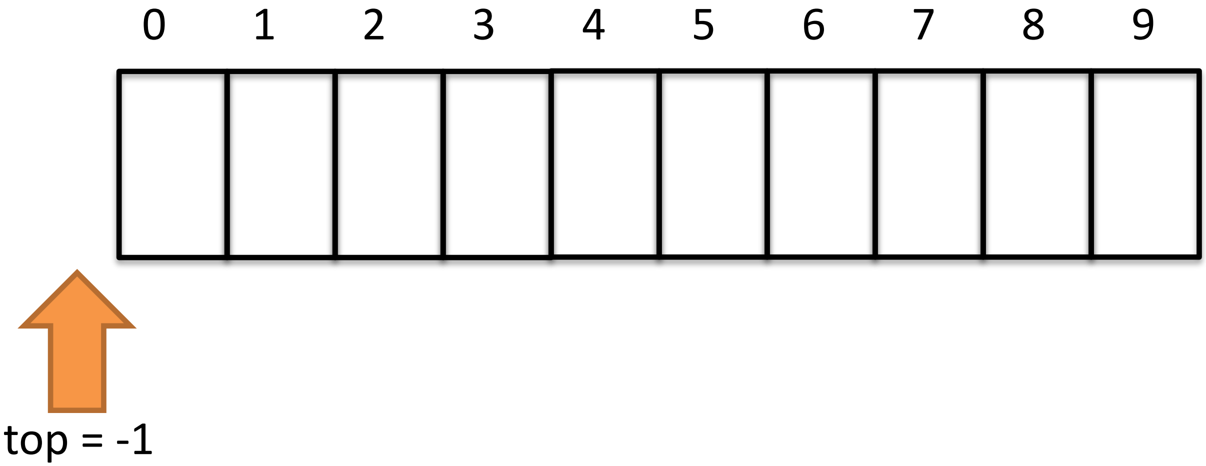
SLLStack.java



```java
public class SLLStack<T> implements SimpleStack<T> {
    private Element top;// top of the stack

    /**
     * The linked elements
     */
    private class Element {
        private T data;
        private Element next;

        public Element(T data, Element next) {
            this.data = data;
            this.next = next;
        }
    }

    public SLLStack() {
        top = null;
    }

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

    public T peek() throws Exception {
        if (isEmpty()) throw new Exception("empty stack");
        return top.data;
    }

    public T pop() throws Exception {
        if (isEmpty()) throw new Exception("empty stack");
        T data = top.data;
        top = top.next;
        return data;
    }

    public void push(T element) {
        top = new Element(element, top);
    }
}
```

We can implement a Stack using an array

Stack array implementation

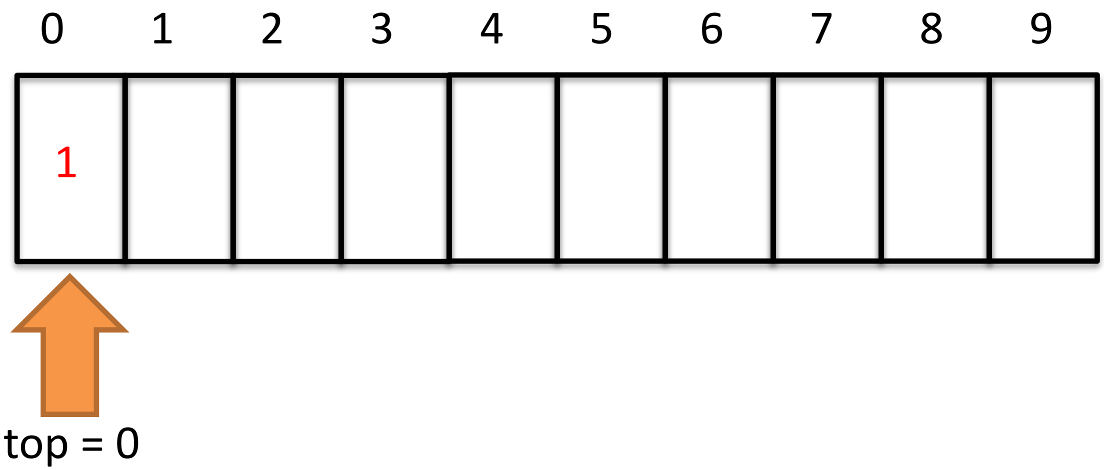


Create array and set *top* = -1

We can implement a Stack using an array

Stack array implementation

push(1)



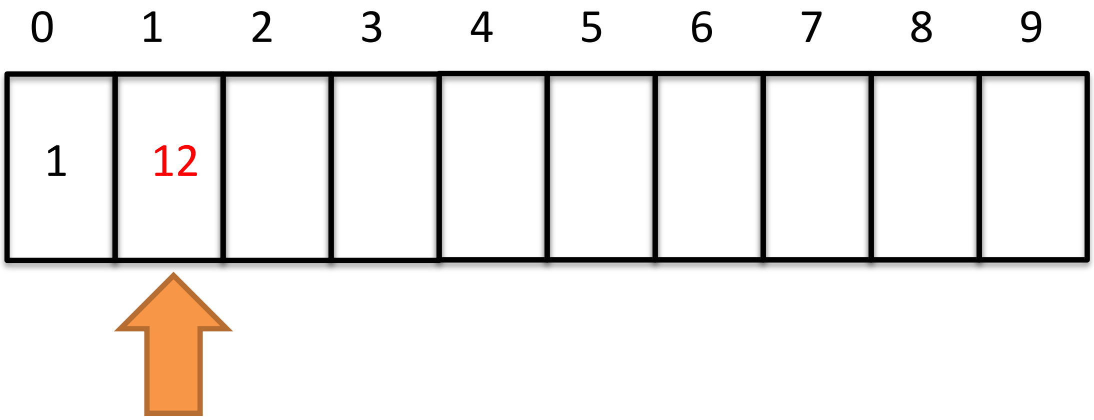
Create array and set *top* = -1

To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

We can implement a Stack using an array

Stack array implementation

push(12)



0	1	2	3	4	5	6	7	8	9
1	12								

top = 1

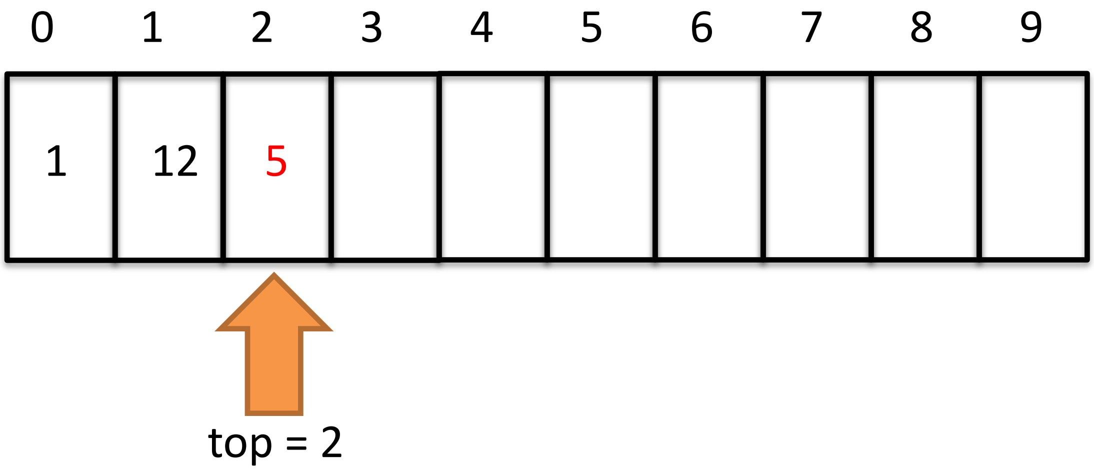
Create array and set *top* = -1

To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

We can implement a Stack using an array

Stack array implementation

push(5)



0	1	2	3	4	5	6	7	8	9
1	12	5							

top = 2

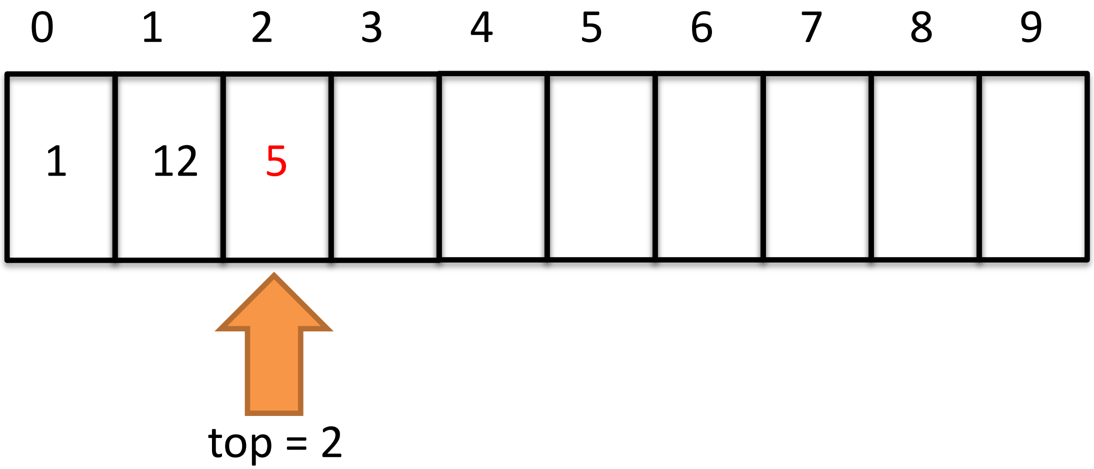
Create array and set *top* = -1

To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

We can implement a Stack using an array

Stack array implementation

peek()
return 5



0	1	2	3	4	5	6	7	8	9
1	12	5							

top = 2

Create array and set *top* = -1

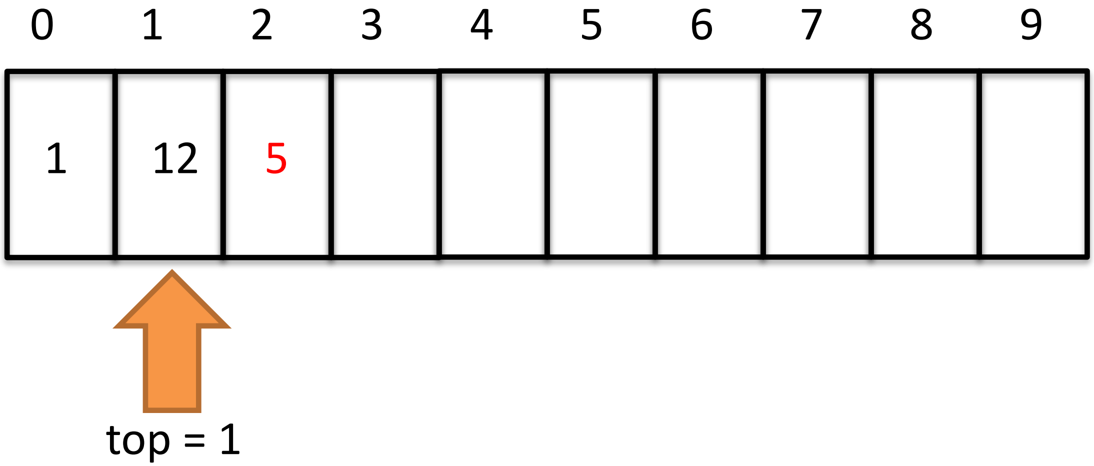
To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

To *peek()* if *top >=0* return *stack[top],* else throw exception

We can implement a Stack using an array

Stack array implementation

pop()
return 5



0	1	2	3	4	5	6	7	8	9
1	12	5							

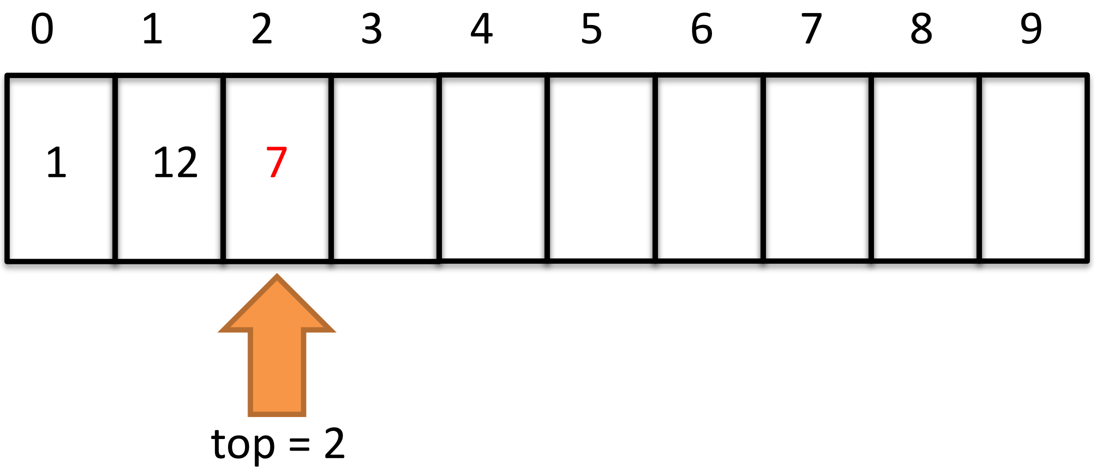
top = 1

Create array and set *top* = -1
To *push(T elmt),* add 1 to *top* and *stack[top] = elmt*
To *peek()* if *top >=0* return *stack[top],* else throw exception
To *pop()*, do *peek()* and set *top -= 1*

We can implement a Stack using an array

Stack array implementation

push(7)



0	1	2	3	4	5	6	7	8	9
1	12	7							

top = 2

Create array and set *top* = -1

To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

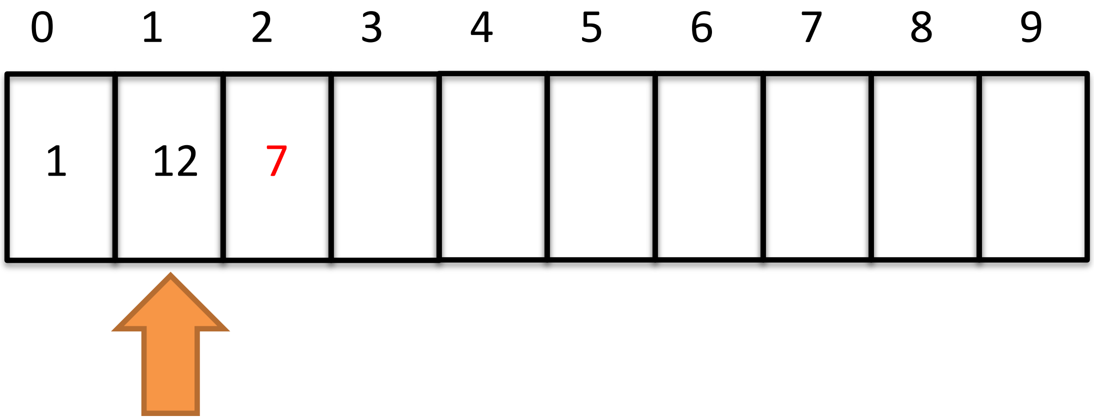
To *peek()* if *top >=0* return *stack[top],* else throw exception

To *pop()*, do *peek()* and set *top -= 1*

We can implement a Stack using an array

Stack array implementation

pop()
return 7



0	1	2	3	4	5	6	7	8	9
1	12	7							

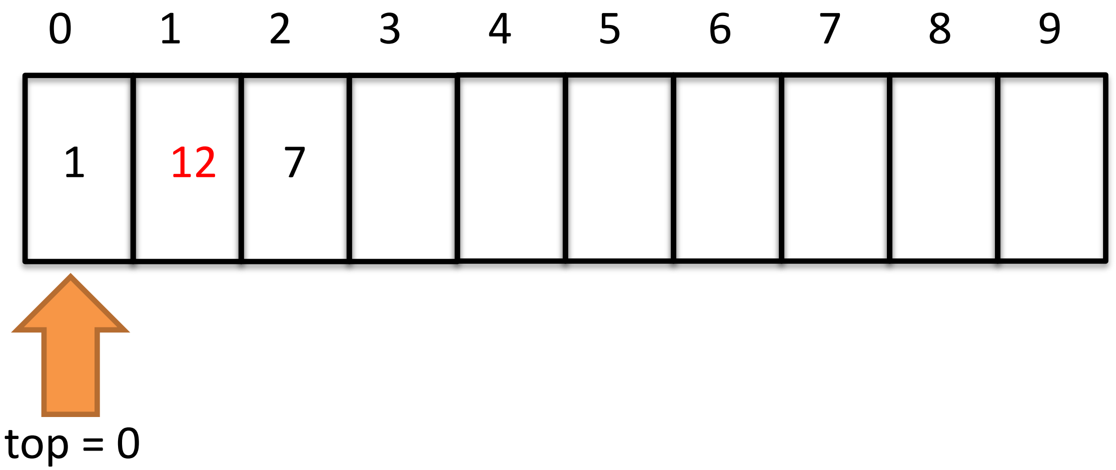
top = 1

Create array and set *top* = -1
To *push(T elmt),* add 1 to *top* and *stack[top] = elmt*
To *peek()* if *top >=0* return *stack[top],* else throw exception
To *pop()*, do *peek()* and set *top -= 1*

We can implement a Stack using an array

Stack array implementation

pop()
return 12



0	1	2	3	4	5	6	7	8	9
1	12	7							

top = 0

Create array and set *top* = -1

To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

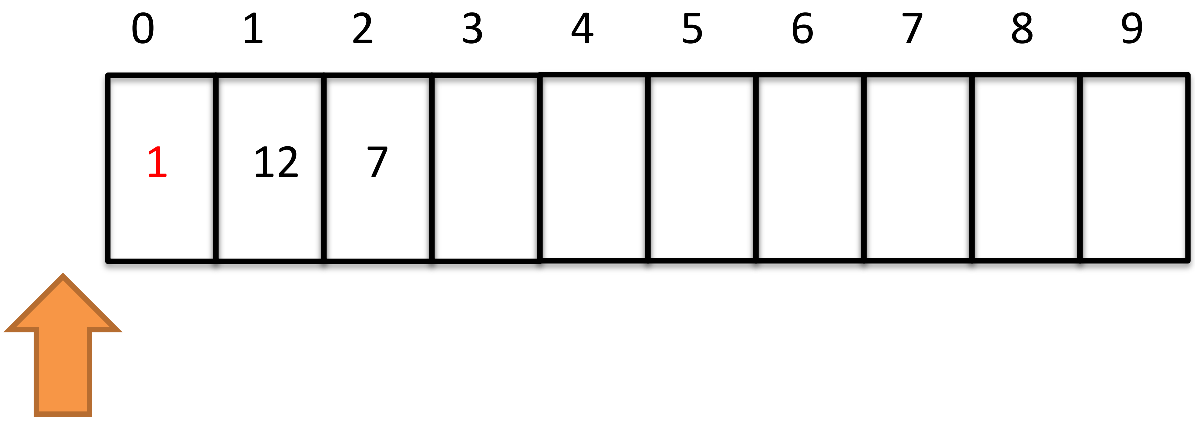
To *peek()* if *top >=0* return *stack[top]*, else throw exception

To *pop()*, do *peek()* and set *top -= 1*

We can implement a Stack using an array

Stack array implementation

pop()
return 1



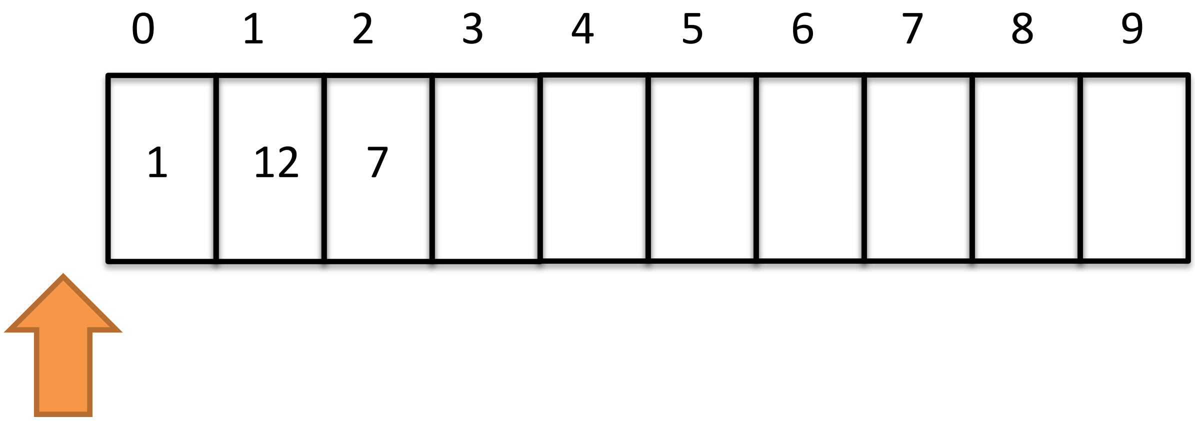
0	1	2	3	4	5	6	7	8	9
1	12	7							

top = -1

Create array and set *top* = -1
To *push(T elmt),* add 1 to *top* and *stack[top] = elmt*
To *peek()* if *top >=0* return *stack[top],* else throw exception
To *pop()*, do *peek()* and set *top -= 1*

We can implement a Stack using an array

Stack array implementation



0	1	2	3	4	5	6	7	8	9
1	12	7							

pop()
throw
exception

top = -1

Create array and set *top* = -1

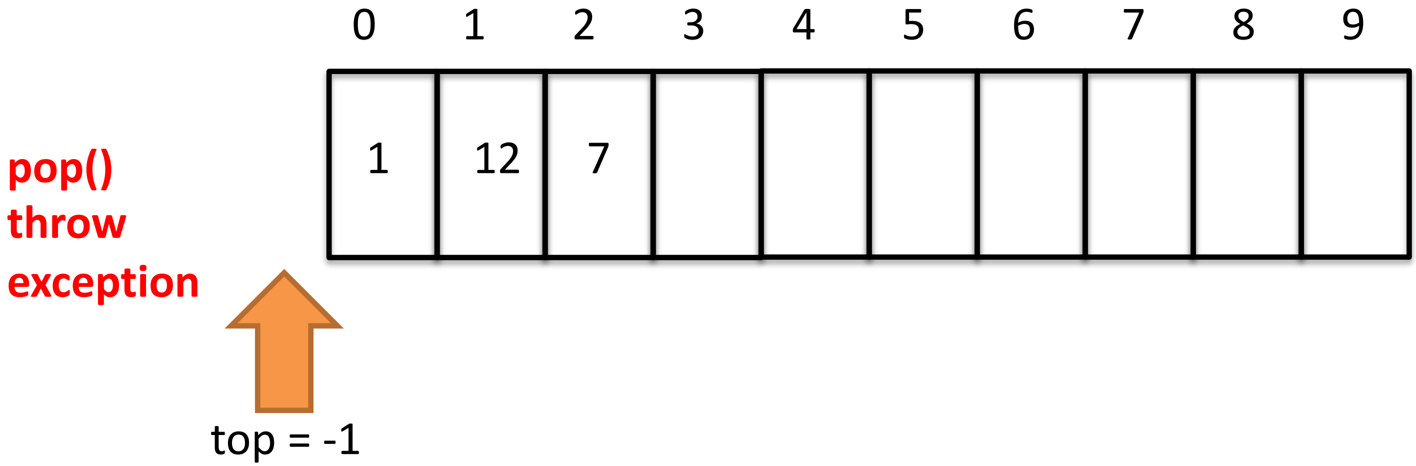
To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

To *peek()* if *top >=0* return *stack[top]*, **else throw exception**

To *pop()*, do *peek()* and set *top -= 1*

We can implement a Stack using an array

Stack array implementation



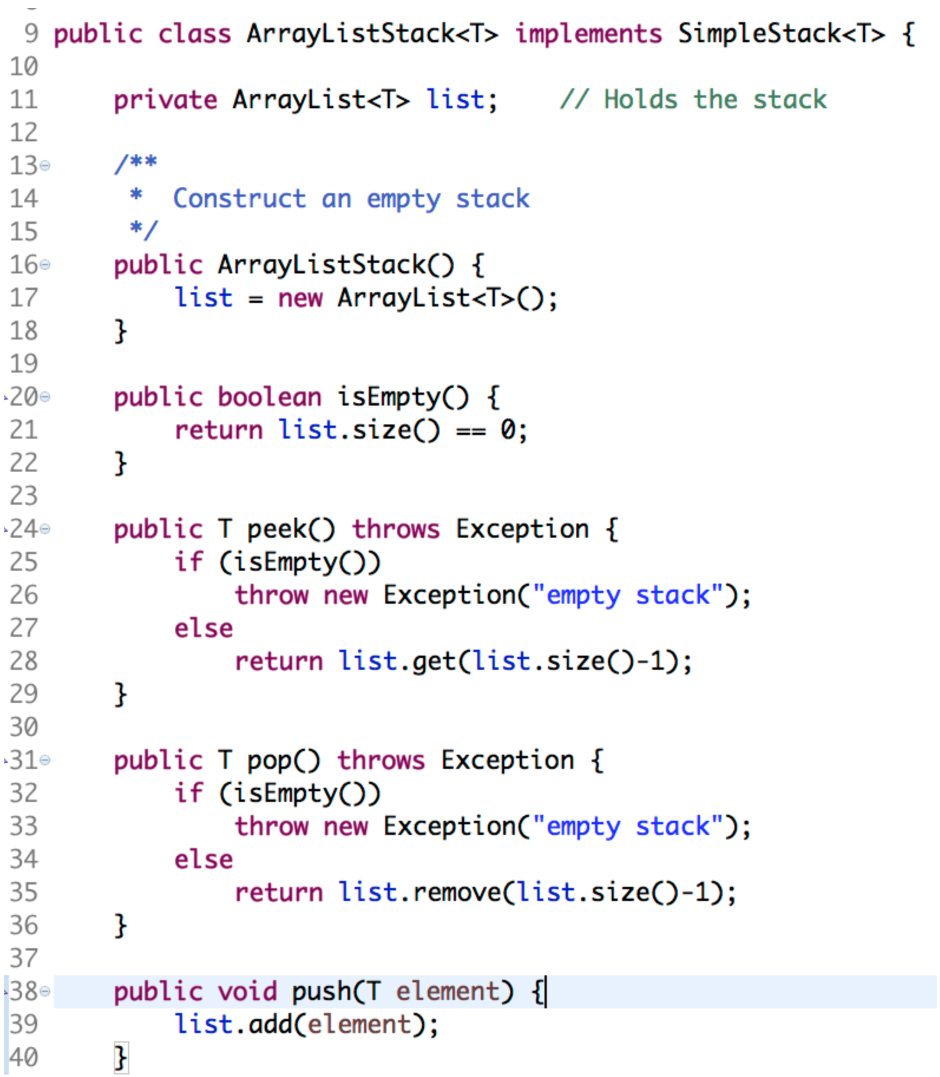
0	1	2	3	4	5	6	7	8	9
1	12	7							

Create array and set *top* = -1
To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*
To *peek()* if *top >=0* return *stack[top],* else throw exception
To *pop()*, do *peek()* and set *top -= 1*

Implementation is O(1) for all operations, never need to move items
Might run out of space using an array, but can grow in amortized O(1) time
Can use ArrayList and not run out of space
As shown, leaves data in memory – security implications!

An ArrayList implementation makes sure the Stack does not run out of space

ArrayListStack.java



```java
public class ArrayListStack<T> implements SimpleStack<T> {

    private ArrayList<T> list;    // Holds the stack

    /**
     *  Construct an empty stack
     */
    public ArrayListStack() {
        list = new ArrayList<T>();
    }

    public boolean isEmpty() {
        return list.size() == 0;
    }

    public T peek() throws Exception {
        if (isEmpty())
            throw new Exception("empty stack");
        else
            return list.get(list.size()-1);
    }

    public T pop() throws Exception {
        if (isEmpty())
            throw new Exception("empty stack");
        else
            return list.remove(list.size()-1);
    }

    public void push(T element) {
        list.add(element);
    }
```

Agenda

1. Stacks
2. Queues

Queues are a First In, First Out (FIFO) data structure

Queue overview

- Think of line at a store, join in back, leave from front
- Used in simulations, queuing print jobs, running jobs, could have used it for PS-1 to visit neighbor pixels
- **Operations**
  - `enqueue` – add item at rear of queue
  - `dequeue` – remove and return first item in queue
  - `front` – return first item, but don’t remove it
  - `isEmpty` – true if queue empty, false otherwise
- Java uses different names (first ones throw exceptions; second ones return false if unable to complete)
  - `enqueue` == `add`*()* and `offer`*()*
  - `dequeue` == `remove`*() and* `poll`*()*
  - `front` == `element`*()* and `peek`*()*

Queues add to back, remove from front; First In First Out (FIFO)

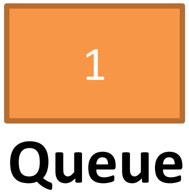
Initially empty

Queue

Queues add to back, remove from front; First In First Out (FIFO)

enqueue(1)

enqueue() adds to back



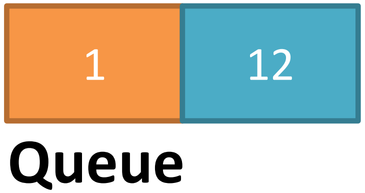
1

Queue

Queues add to back, remove from front; First In First Out (FIFO)

enqueue(12)

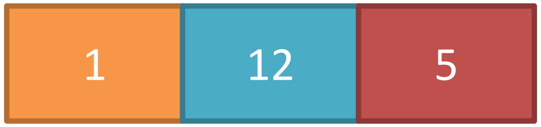
enqueue() adds to back



Queues add to back, remove from front; First In First Out (FIFO)

enqueue(5)

enqueue() adds to back

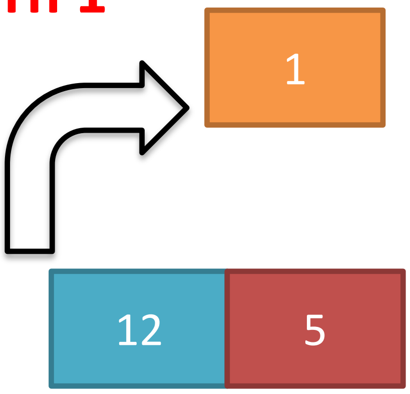


Queue

Queues add to back, remove from front; First In First Out (FIFO)

dequeue()

Return 1



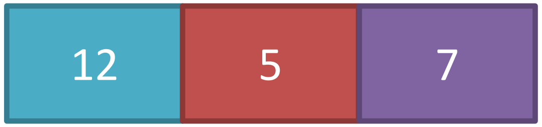
Queue

dequeue() removes from front

Queues add to back, remove from front; First In First Out (FIFO)

enqueue(7)

enqueue() adds to back



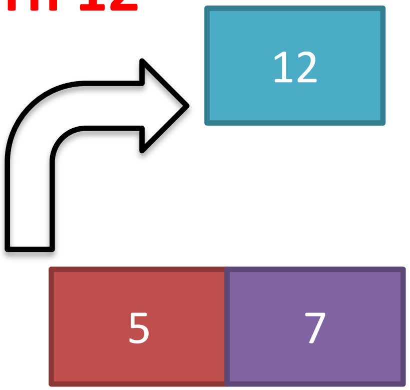
12 5 7

Queue

Queues add to back, remove from front; First In First Out (FIFO)

dequeue()

Return 12



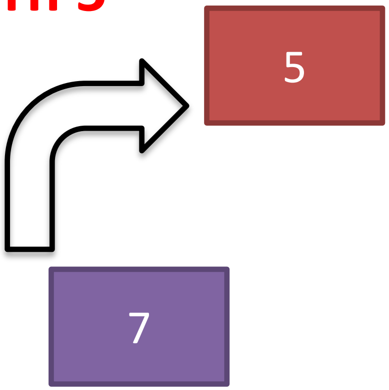
Queue

dequeue() removes from front

Queues add to back, remove from front; First In First Out (FIFO)

dequeue()

Return 5



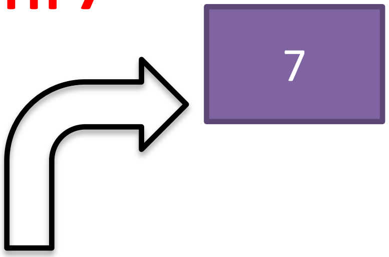
Queue

dequeue() removes from front

Queues add to back, remove from front; First In First Out (FIFO)

dequeue()

Return 7

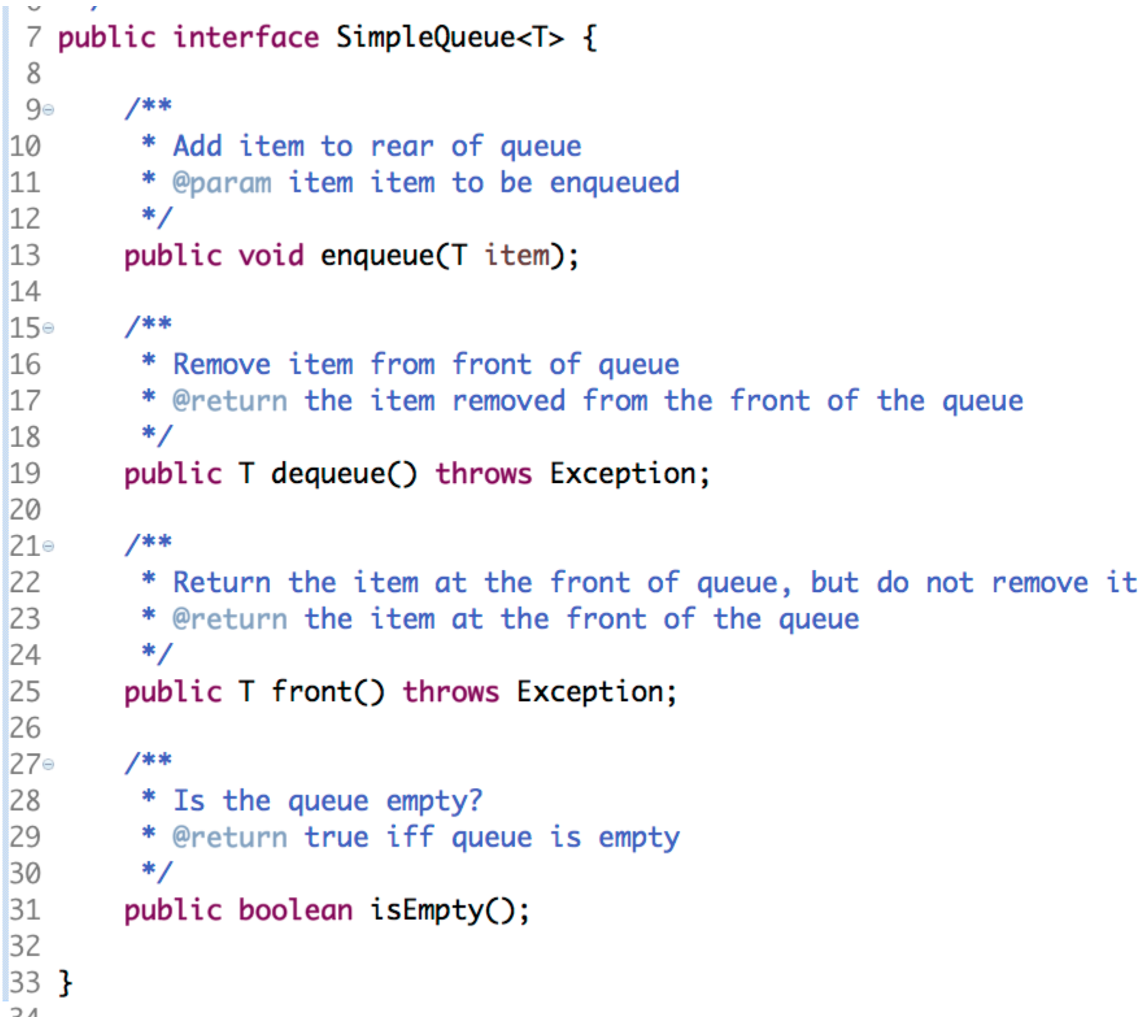


dequeue() removes from front

Queue

SimpleQueue.java: Interface defining Queue operations

SimpleQueue.java



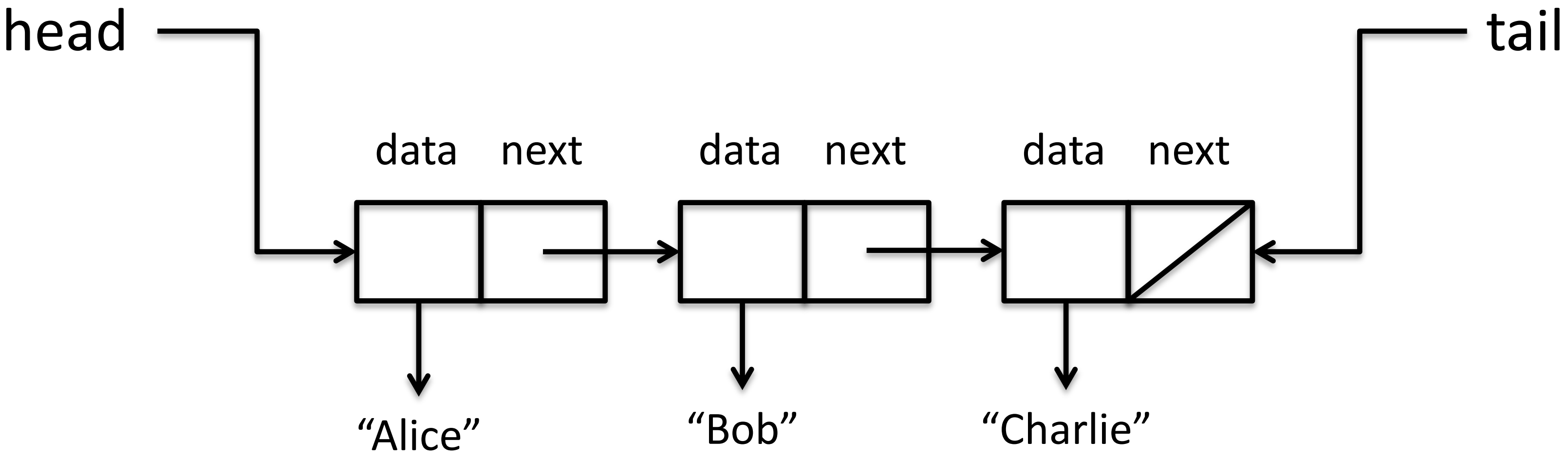
```java
public interface SimpleQueue<T> {

    /**
     * Add item to rear of queue
     * @param item item to be enqueued
     */
    public void enqueue(T item);

    /**
     * Remove item from front of queue
     * @return the item removed from the front of the queue
     */
    public T dequeue() throws Exception;

    /**
     * Return the item at the front of queue, but do not remove it
     * @return the item at the front of the queue
     */
    public T front() throws Exception;

    /**
     * Is the queue empty?
     * @return true iff queue is empty
     */
    public boolean isEmpty();

}
```

How to implement it?

Queues can be implemented with Singly Linked List using *head* and *tail* pointers

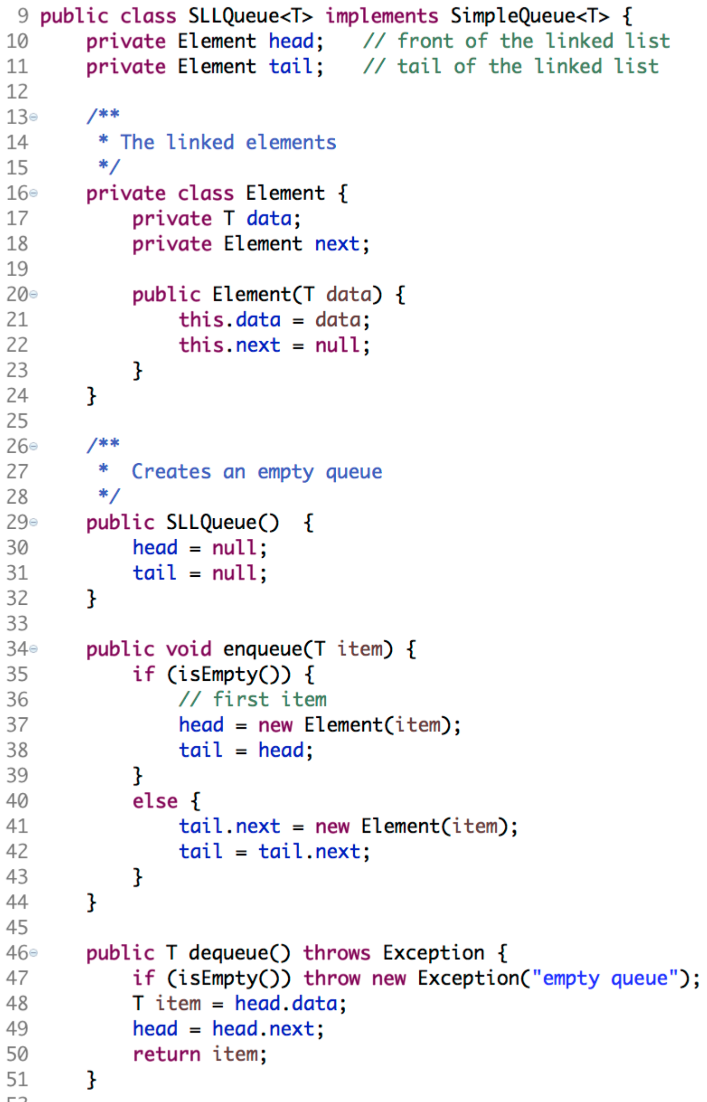
Queue implementation

- Easy to get/remove from *head*
- Use *tail* to add to back of queue
  - Set new element *next* to null
  - Set *tail.next* to new element
  - Move *tail* to new element (*tail = tail.next*)
- All operations Θ(1)



Queues can be implemented with Singly Linked List using head and tail pointers

SLLQueue.java



```java
public class SLLQueue<T> implements SimpleQueue<T> {
    private Element head;    // front of the linked list
    private Element tail;    // tail of the linked list

    /**
     * The linked elements
     */
    private class Element {
        private T data;
        private Element next;

        public Element(T data) {
            this.data = data;
            this.next = null;
        }
    }

    /**
     *  Creates an empty queue
     */
    public SLLQueue()  {
        head = null;
        tail = null;
    }

    public void enqueue(T item) {
        if (isEmpty()) {
            // first item
            head = new Element(item);
            tail = head;
        }
        else {
            tail.next = new Element(item);
            tail = tail.next;
        }
    }

    public T dequeue() throws Exception {
        if (isEmpty()) throw new Exception("empty queue");
        T item = head.data;
        head = head.next;
        return item;
    }
```

Arrays are seemingly unpromising as a Queue data structure, but can work well

Array implementation

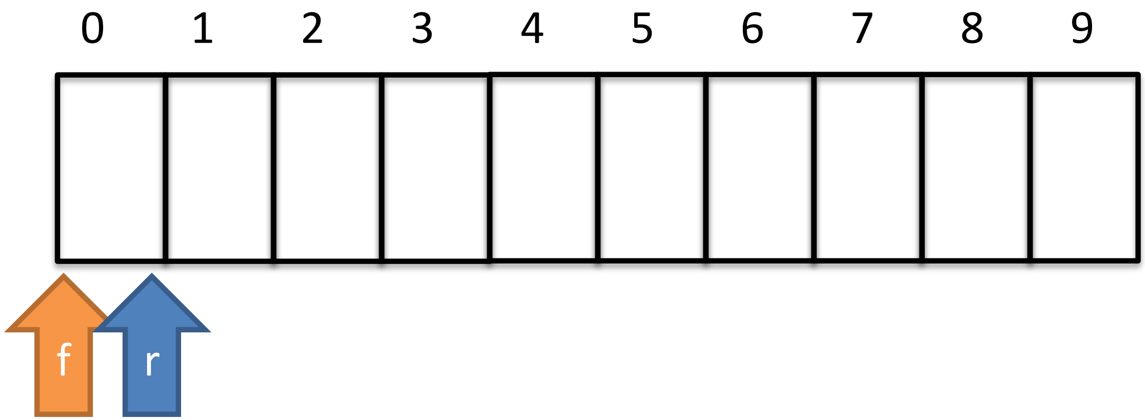
- Could `enqueue` at back, `dequeue` from front
  - `enqueue` is fast, just add item to end O(1)
  - `dequeue` must move all elements left one space O(n)
- Could `enqueue` at front and `dequeue` from back
  - `enqueue` must move all elements right one space O(n)
  - `dequeue` is fast, just take last item O(1)
- Could track `front (f)` and `rear (r)` indexes (circular array)
  - `enqueue` at `r`, then increment `r`
  - `dequeue` at `f`, then increment `f`
  - If `f` or `r > m-1`, wrap around to empty spaces at front
  - Full or empty when `f==r` (full if *size* !=0)
  - `enqueue` and `dequeue` O(1)

Array implementing a Queue using index for front and rear

Array implementing Queue

Empty *f=r=0*

***size* = 0**



***front (f)* is index of first element**

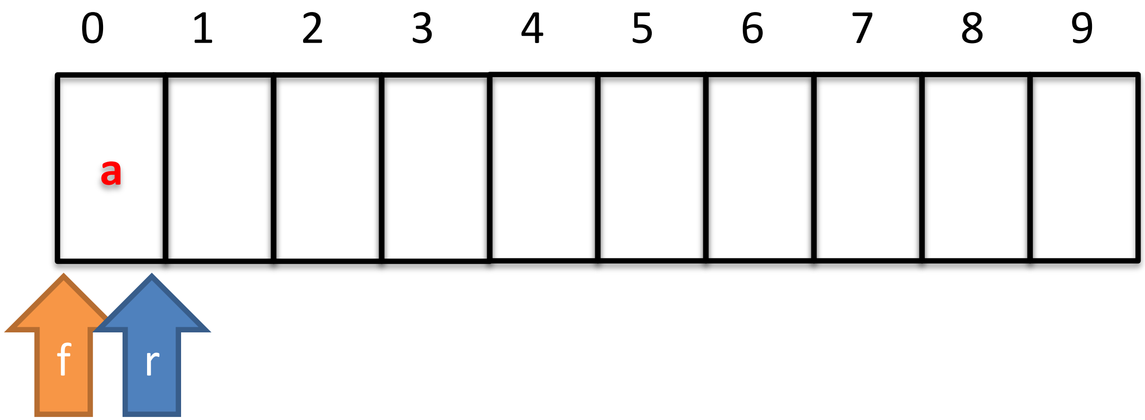
***rear (r)* is index of next free space (initially 0)**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(a)

size = 0



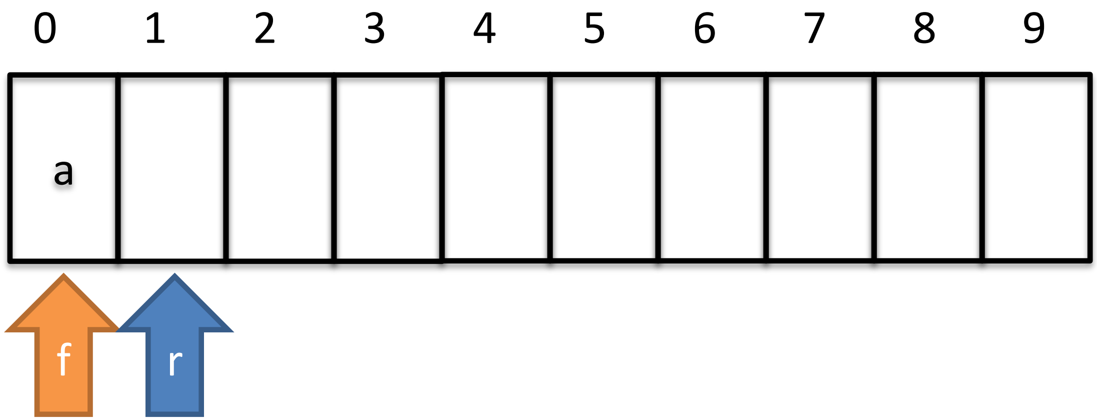
Set *a* at position *rear*

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(a)

size = 1



Set *a* at position *rear*

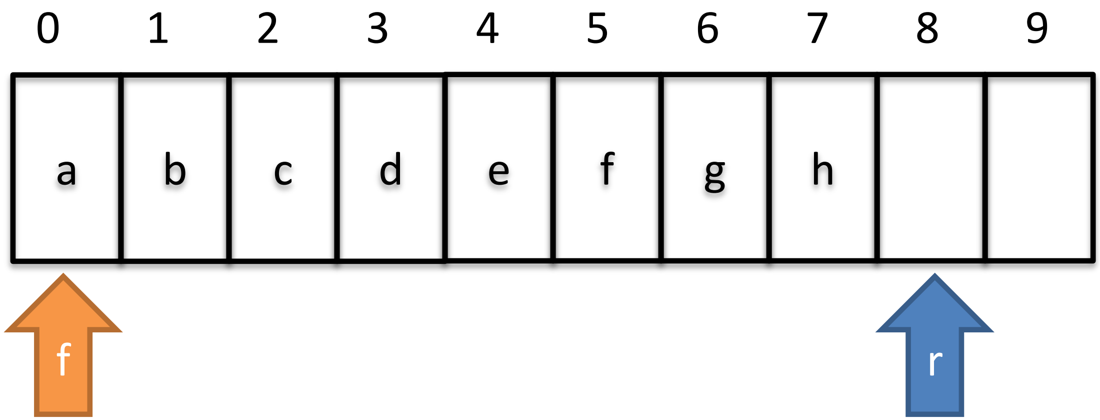
Set *rear* +=1, *size* +=1

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(b) **through** ***enqueue(h)***

size = 8



0	1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h		

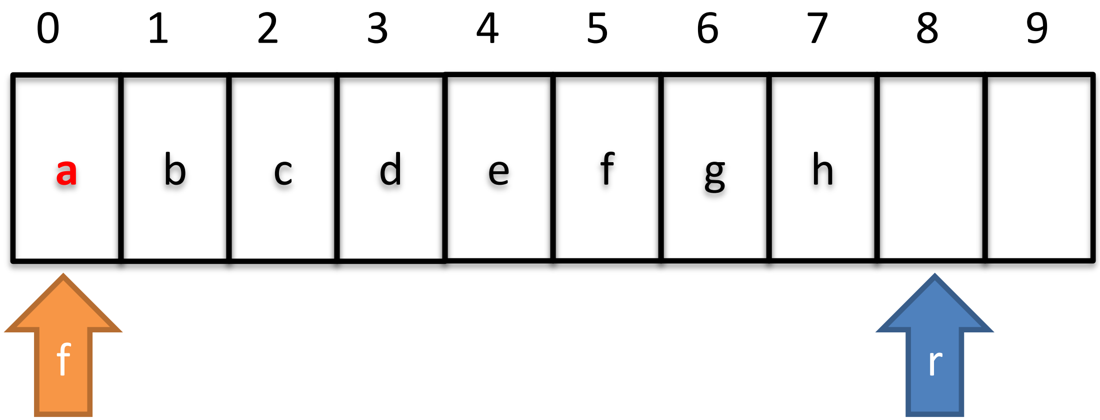
front **stays at 0,** ***rear*** **+=1 and** ***size*** **+=1 on each** ***enqueue()***

Array implementing a Queue using index for front and rear

Array implementing Queue

dequeue()

size = 8



0	1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h		

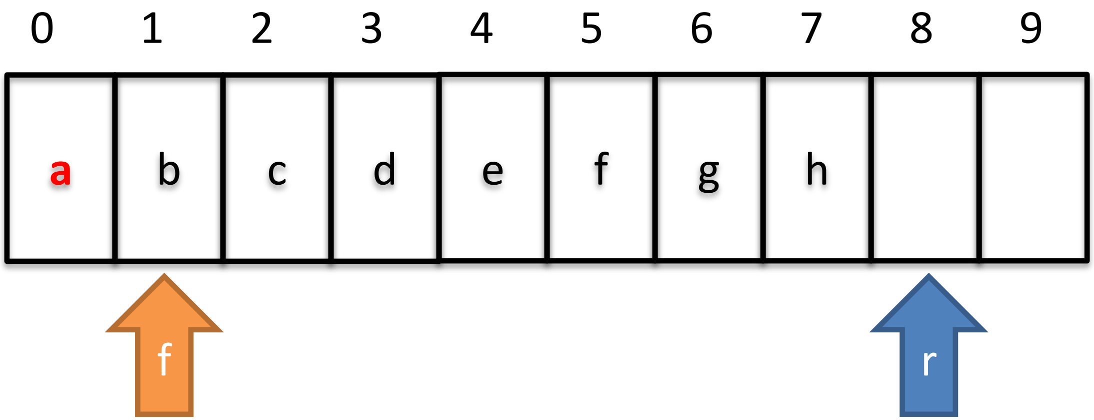
Return item at *front* -> a

Array implementing a Queue using index for front and rear

Array implementing Queue

dequeue()

size = 7



Return item at *front* -> a

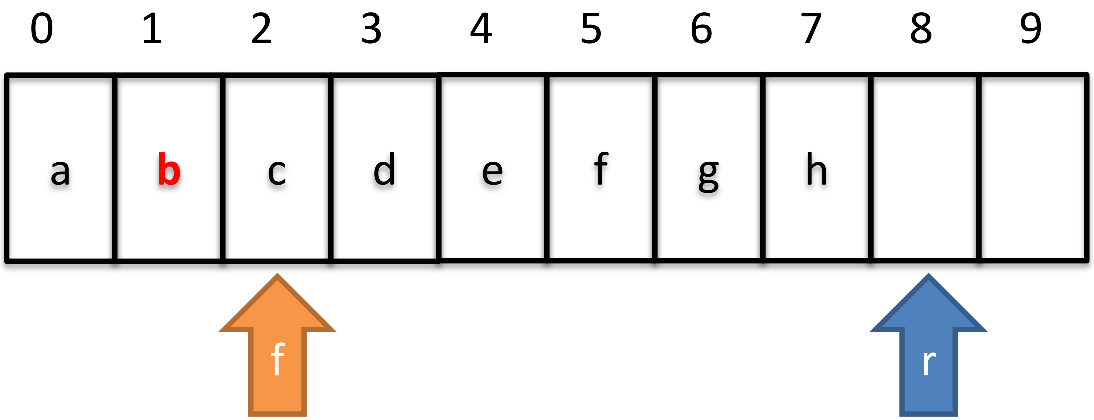
***front* +=1, *size* -=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

dequeue()

size = 6



0	1	2	3	4	5	6	7	8	9
a	**b**	c	d	e	f	g	h		

Return item at *front* -> b

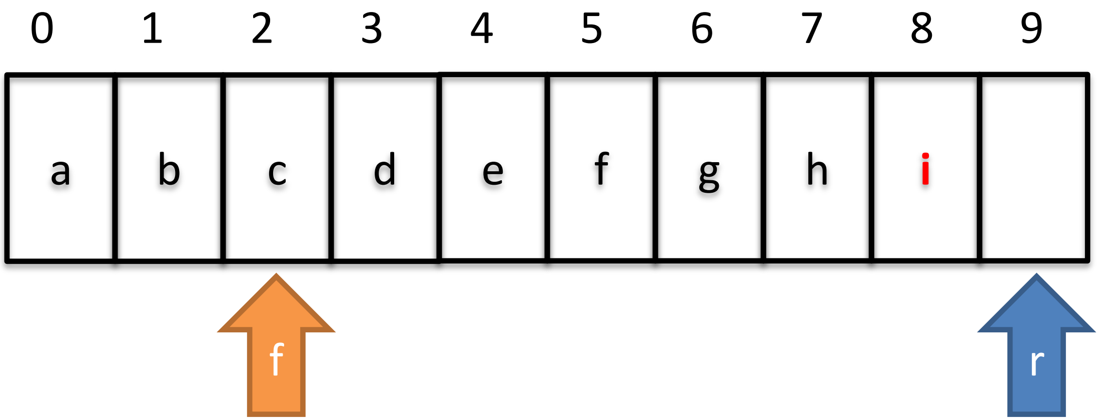
***front* +=1, *size* -=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(i)

size = 7



Set *i* at position *rear*

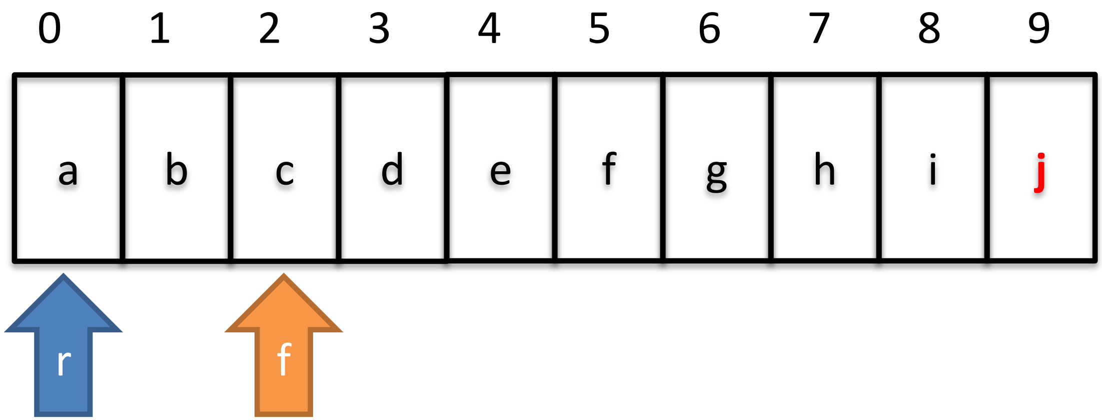
***rear* +=1, *size* +=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(j)

size = 8



0	1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h	i	j

Set *j* at position *rear*

***rear* +=1, wrap around to index 0**

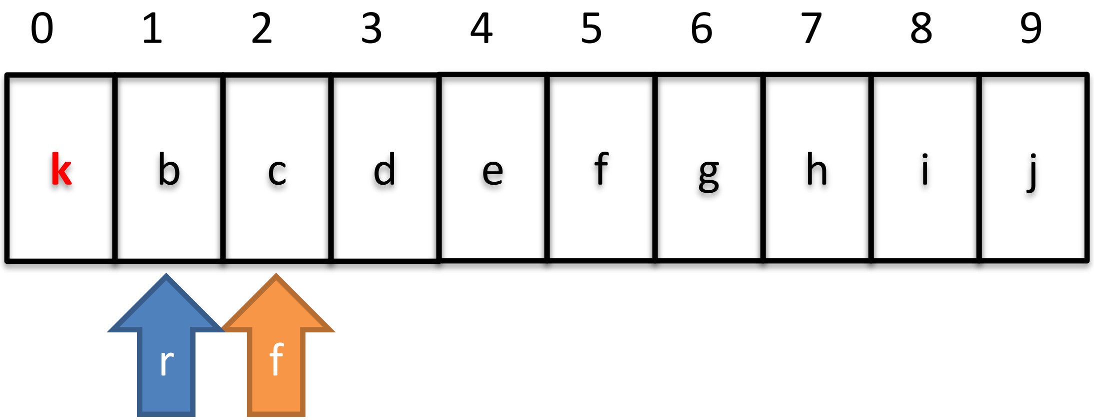
***size* +=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(k)

size = 9



0	1	2	3	4	5	6	7	8	9
k	b	c	d	e	f	g	h	i	j

Set *k* at position *rear*

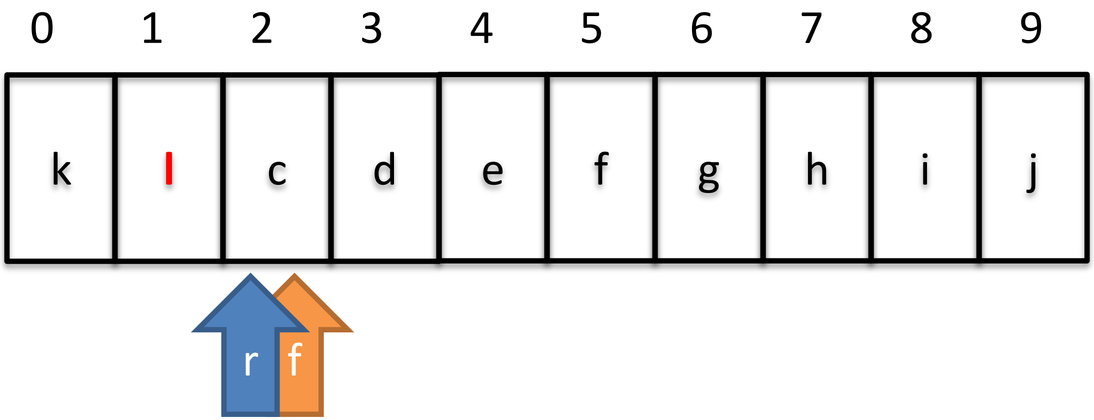
***rear* +=1, *size* +=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(l)

size = 10



0	1	2	3	4	5	6	7	8	9
k	l	c	d	e	f	g	h	i	j

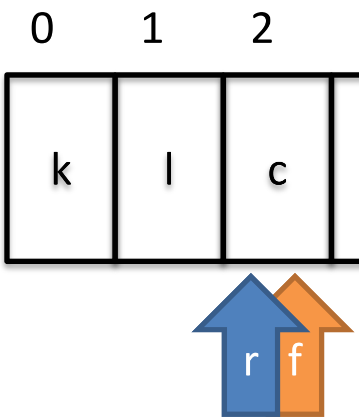
Set *l* at position *rear*

***rear* +=1, *size* +=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

size = 10



0	1	2	3	4	5	6	7	8	9
k	l	c	d	e	f	g	h	i	j

Array is full (*f==r* and *size != 0*), now what?

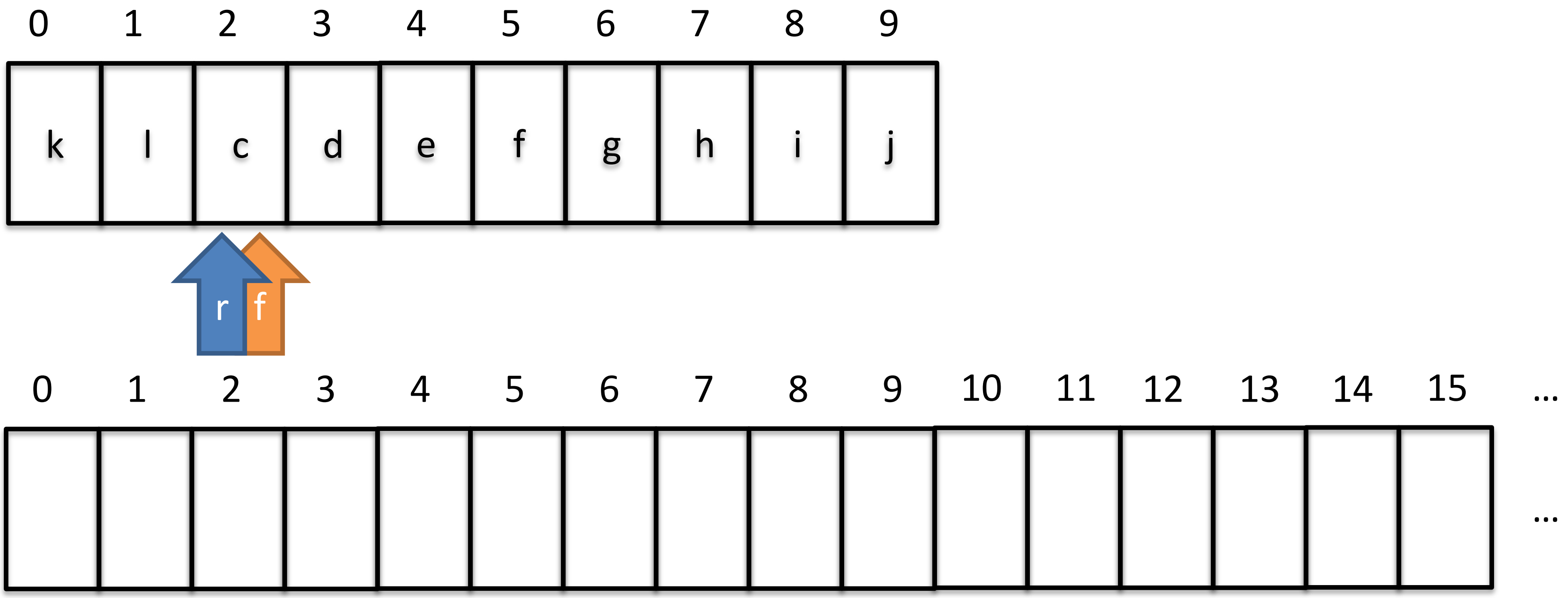
Grow by creating new larger array

Copy elements from old array into new array

How if *front* != 0 due to *dequeue* operations?

Array implementing a Queue using index for front and rear

Array implementing Queue



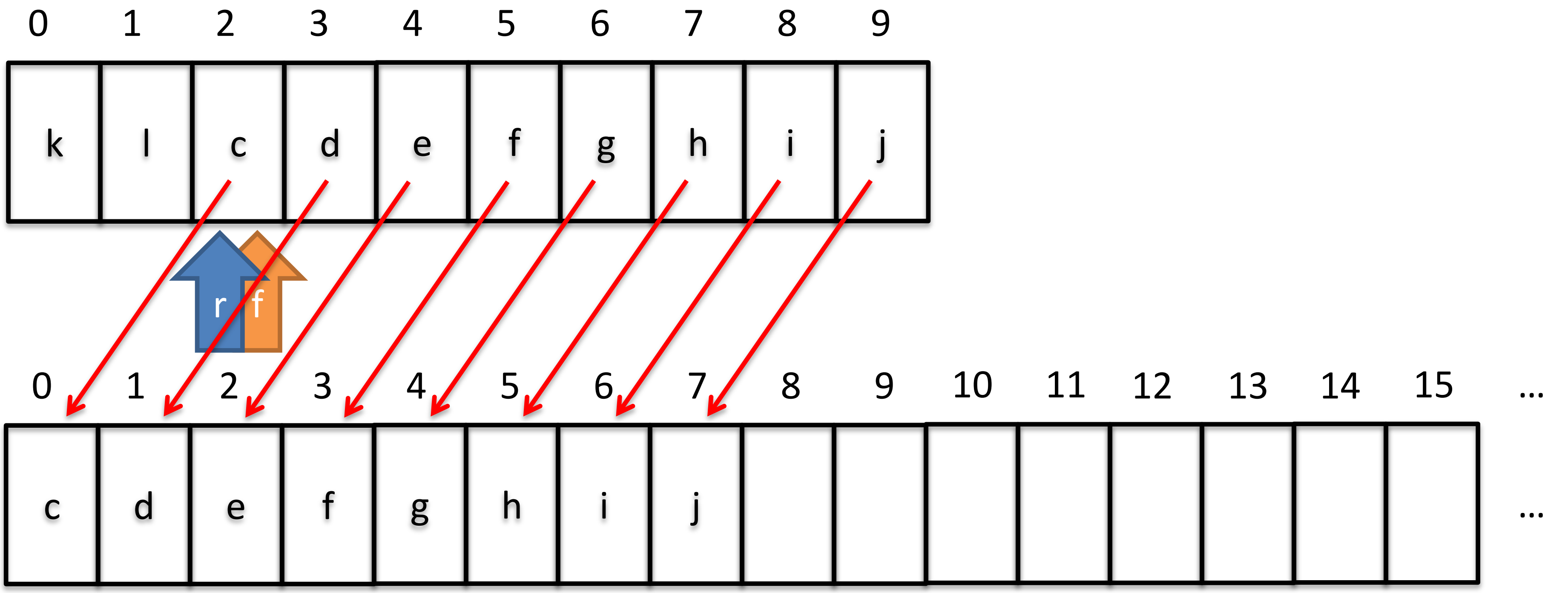
0	1	2	3	4	5	6	7	8	9
k	l	c	d	e	f	g	h	i	j

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
																…

Copy old array from *front to size-1* into new array starting at index 0

Array implementing a Queue using index for front and rear

Array implementing Queue



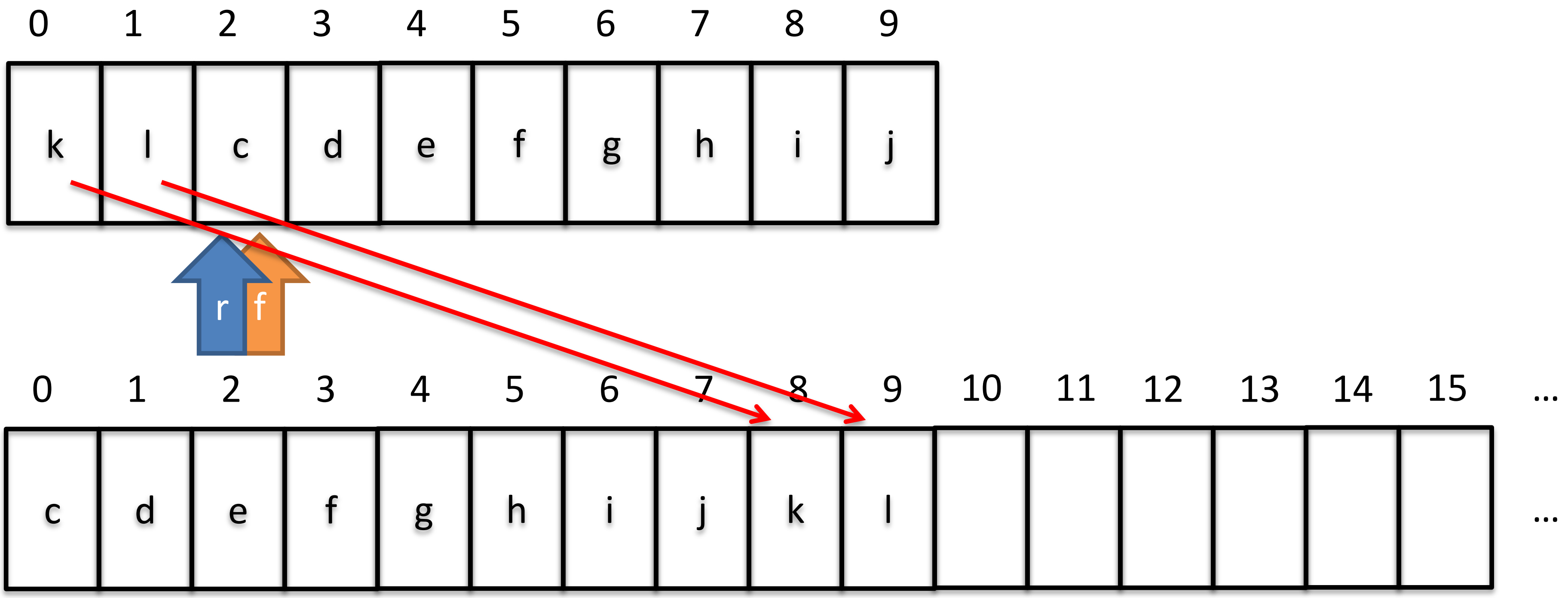
0	1	2	3	4	5	6	7	8	9
k	l	c	d	e	f	g	h	i	j

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
c	d	e	f	g	h	i	j									…

Copy old array from *front to size-1* into new array starting at index 0

Array implementing a Queue using index for front and rear

Array implementing Queue



0	1	2	3	4	5	6	7	8	9
k	l	c	d	e	f	g	h	i	j

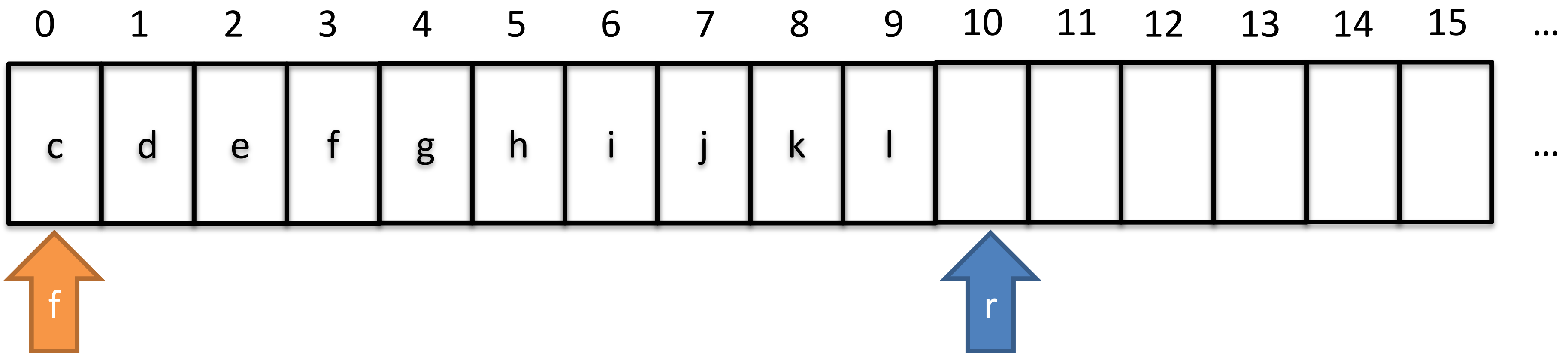
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
c	d	e	f	g	h	i	j	k	l							…

Copy old array from index 0 to *front-1* into new array starting at last index

Array implementing a Queue using index for front and rear

Array implementing Queue

size = 10



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	…
c	d	e	f	g	h	i	j	k	l							…

Set *front = 0* and *rear = size*

Set array to new array

Summary

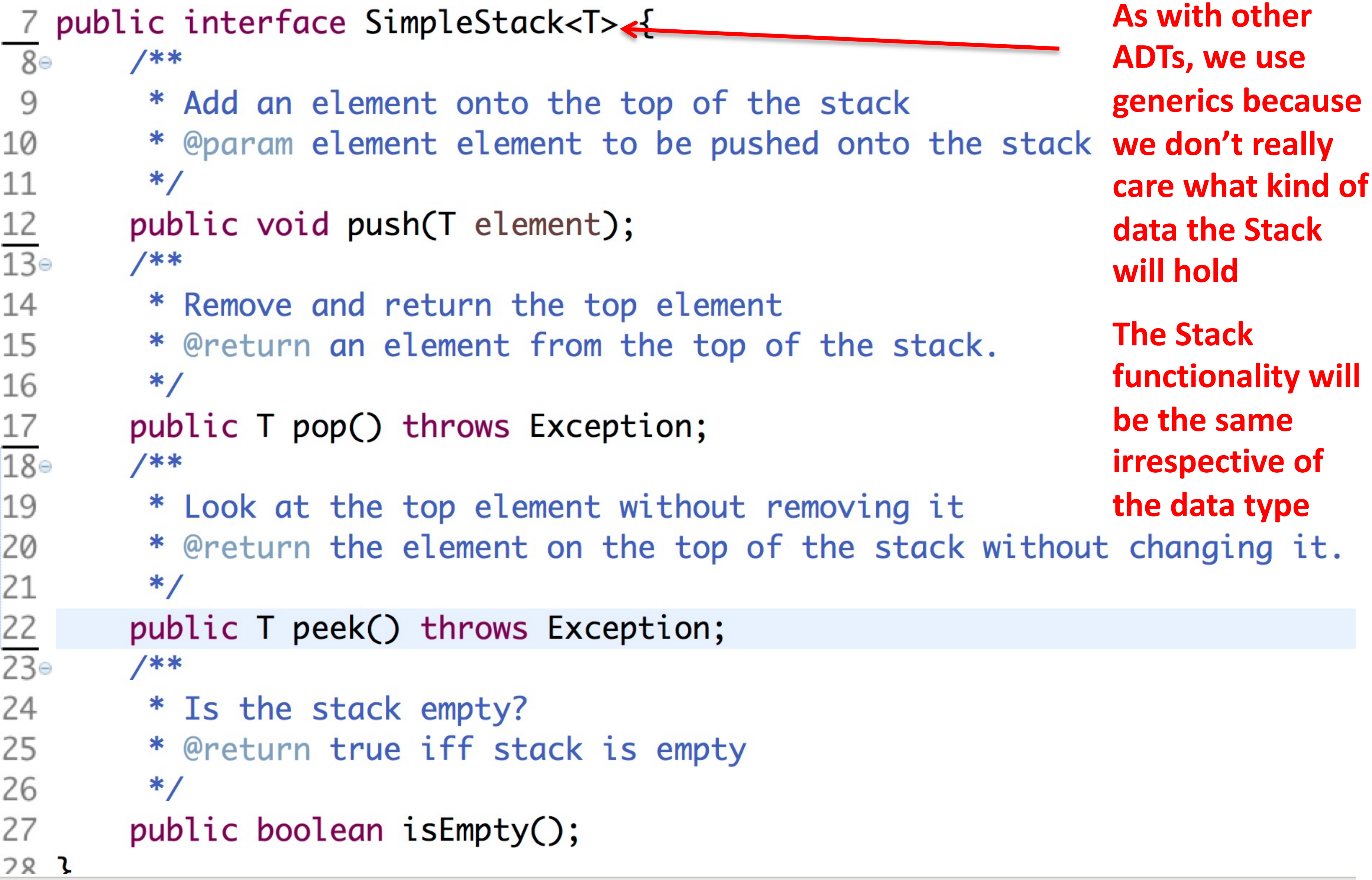
- Stacks Last in, First out
  - Singly linked list can work as an implementation
  - Or array
  - Both with proper insertion and removal can have constant time
- Queue First in, First Out
  - Singly linked list (with tail) can work as implementation
  - Array with two pointers and growing
  - Also here constant time

Additional Resources

SimpleStack.java

ANNOTATED SLIDES

SimpleStack.java: Interface defining Stack operations



```java
public interface SimpleStack<T> {
    /**
     * Add an element onto the top of the stack
     * @param element element to be pushed onto the stack
     */
    public void push(T element);
    /**
     * Remove and return the top element
     * @return an element from the top of the stack.
     */
    public T pop() throws Exception;
    /**
     * Look at the top element without removing it
     * @return the element on the top of the stack without changing it.
     */
    public T peek() throws Exception;
    /**
     * Is the stack empty?
     * @return true iff stack is empty
     */
    public boolean isEmpty();
}
```

As with other ADTs, we use generics because we don’t really care what kind of data the Stack will hold

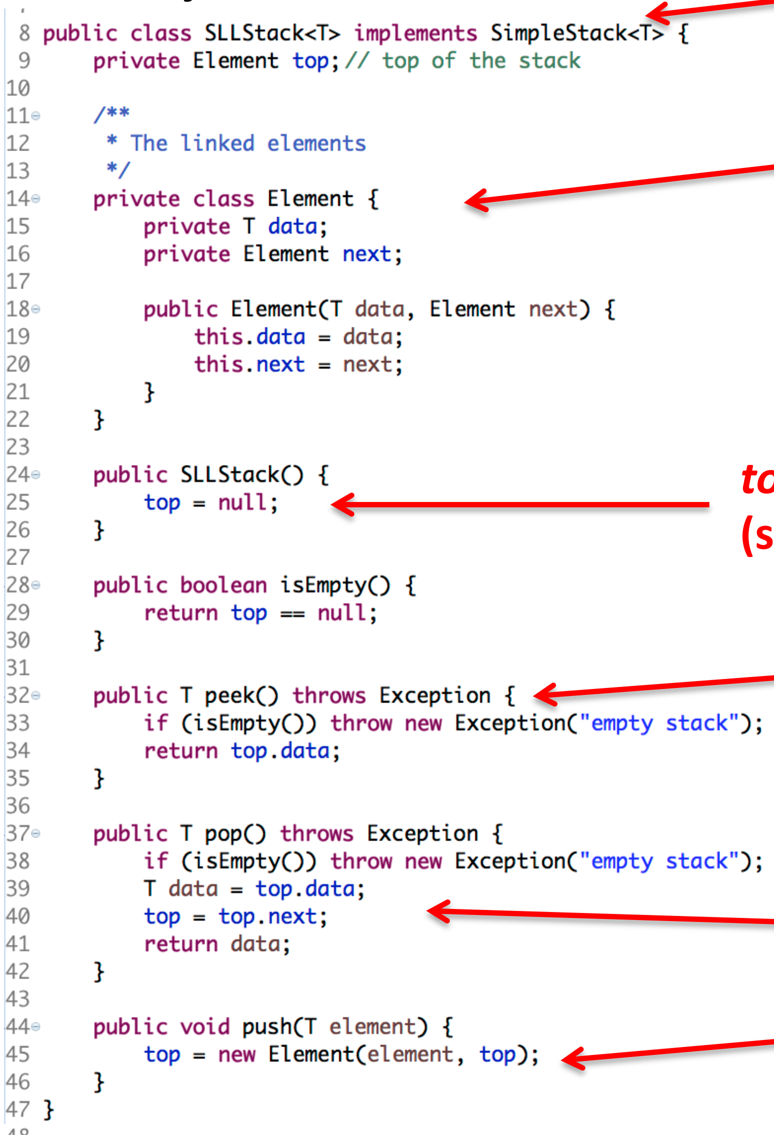
The Stack functionality will be the same irrespective of the data type

SLLStack.java

ANNOTATED SLIDES

A Singly Linked List works well for a Stack, using top as head of list

SLLStack.java



```java
public class SLLStack<T> implements SimpleStack<T> {
    private Element top;// top of the stack

    /**
     * The linked elements
     */
    private class Element {
        private T data;
        private Element next;

        public Element(T data, Element next) {
            this.data = data;
            this.next = next;
        }
    }

    public SLLStack() {
        top = null;
    }

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

    public T peek() throws Exception {
        if (isEmpty()) throw new Exception("empty stack");
        return top.data;
    }

    public T pop() throws Exception {
        if (isEmpty()) throw new Exception("empty stack");
        T data = top.data;
        top = top.next;
        return data;
    }

    public void push(T element) {
        top = new Element(element, top);
    }
}
```

Implements SimpleStack interface, so must implment its methods

Private Element class as we've seen before

Data is of generic type T

- All operations Θ(1)
- Unlike an array, this does not run out of space

top keeps track of top of stack (same as *head* did), initially null

peek() returns *data* of first Element in list but does not remove it

pop() gets *data* from first Element in list, then sets *top* to *next*

push() adds new Element at *top*

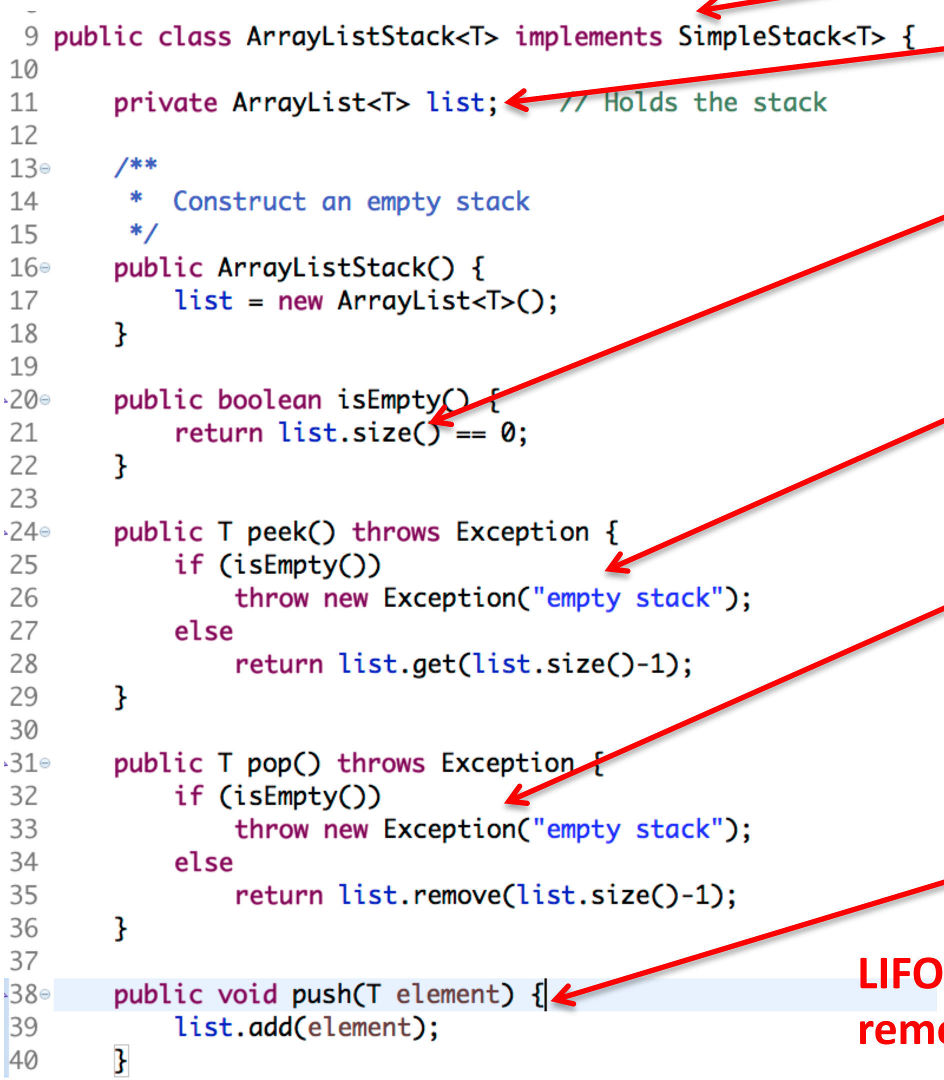
Sets new Element next to *top's* prior value

ArrayListStack.java

ANNOTATED SLIDES

An ArrayList implementation makes sure the Stack does not run out of space

ArrayListStack.java



```java
public class ArrayListStack<T> implements SimpleStack<T> {

    private ArrayList<T> list;    // Holds the stack

    /**
     *  Construct an empty stack
     */
    public ArrayListStack() {
        list = new ArrayList<T>();
    }

    public boolean isEmpty() {
        return list.size() == 0;
    }

    public T peek() throws Exception {
        if (isEmpty())
            throw new Exception("empty stack");
        else
            return list.get(list.size()-1);
    }

    public T pop() throws Exception {
        if (isEmpty())
            throw new Exception("empty stack");
        else
            return list.remove(list.size()-1);
    }

    public void push(T element) {
        list.add(element);
    }
```

Implements SimpleStack interface

ArrayList as stack

ArrayList *size* keeps track of *isEmpty()*

***peek()* returns value of last item but does not change stack**
Throws exception if stack empty

***pop()* removes and returns last item**
Throws exception if stack empty

***push()* adds element to stack at end**
List add method grows array if needed, O(1)

LIFO: add to end (top) and remove from end -> O(1)

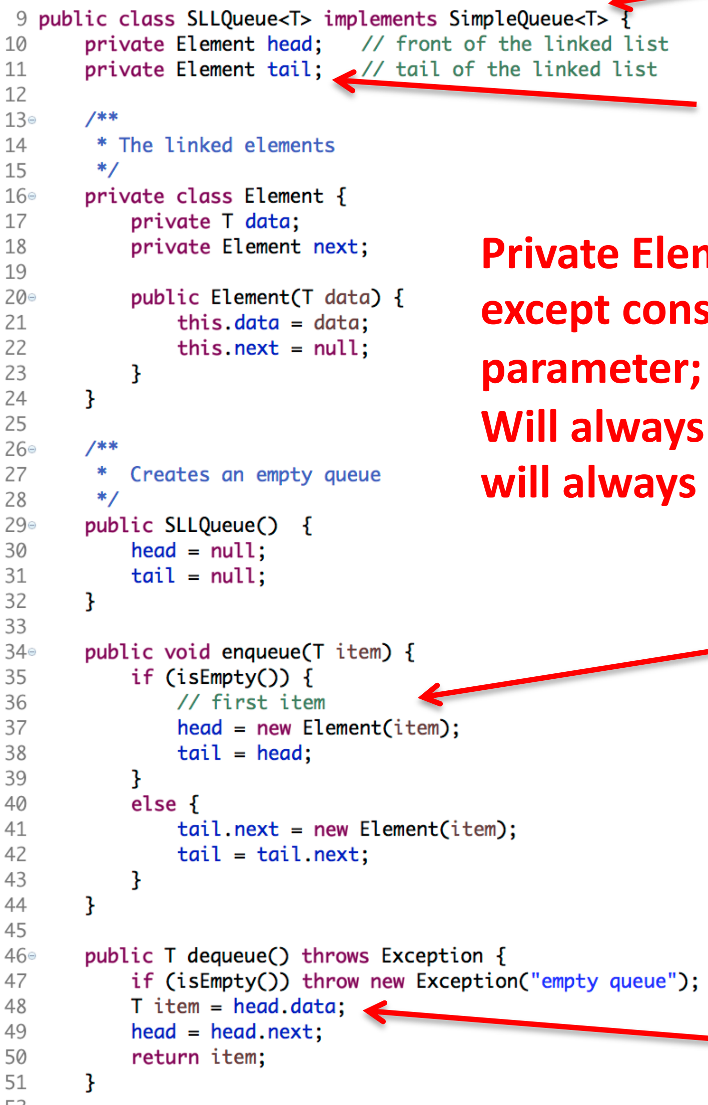
SLLQueue.java

ANNOTATED SLIDES

Queues can be implemented with Singly Linked List using head and tail pointers

SLLQueue.java

Implements SimpleQueue interface



```java
public class SLLQueue<T> implements SimpleQueue<T> {
    private Element head;    // front of the linked list
    private Element tail;    // tail of the linked list

    /**
     * The linked elements
     */
    private class Element {
        private T data;
        private Element next;

        public Element(T data) {
            this.data = data;
            this.next = null;
        }
    }

    /**
     *  Creates an empty queue
     */
    public SLLQueue()  {
        head = null;
        tail = null;
    }

    public void enqueue(T item) {
        if (isEmpty()) {
            // first item
            head = new Element(item);
            tail = head;
        }
        else {
            tail.next = new Element(item);
            tail = tail.next;
        }
    }

    public T dequeue() throws Exception {
        if (isEmpty()) throw new Exception("empty queue");
        T item = head.data;
        head = head.next;
        return item;
    }
```

Keep a pointer to *head* (for dequeue) and a pointer to *tail* (for enqueue)

Private Element class, same a before, except construct doesn't take *next* parameter; why?

Will always set *next* to null because will always add at end

Check if first item

enqueue()* at end of queue using *tail

dequeue()* from front of queue using *head

EXAMPLE OF USE OF STACK

We can use the simple stack to easily match parens in a string

JSON String

```
Students: [
{“id”: 123, “name” : “Alice”}
{“id”: 987, “name” : “Bob”}
]
```

Open Parens: [, {, (, <

Close parens:], },), >

Define matching open and close parens

Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Pseudo code ensures matching parens

We can use the simple stack to easily match parens in a string

JSON String

```
Students: [
{“id”: 123, “name” : “Alice”}
{“id”: 987, “name” : “Bob”}
]
```

Open Parens: [, {, (, <
Close parens:], },), >

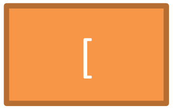
Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Current character

[

Stack



[

We can use the simple stack to easily match parens in a string

JSON String
Students: **[**
{“id”: 123, “name” : “Alice”}
{“id”: 987, “name” : “Bob”}
]

Open Parens: [, {, (, <
Close parens:], },), >

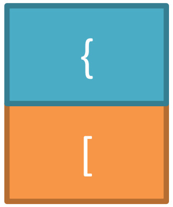
Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Current character

{

Stack



We can use the simple stack to easily match parens in a string

JSON String

```
Students: [
{“id”: 123, “name” : “Alice”}
{“id”: 987, “name” : “Bob”}
]
```

Open Parens: [, {, (, <

Close parens:], },), >

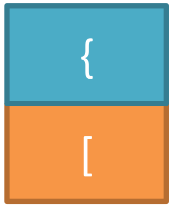
Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Current character

}

Stack



We can use the simple stack to easily match parens in a string

JSON String

```
Students: [
{“id”: 123, “name” : “Alice”}
{“id”: 987, “name” : “Bob”}
]
```

Open Parens: [, {, (, <
Close parens:], },), >

Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack -> {   matches current }
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Current character

}

Stack

[

We can use the simple stack to easily match parens in a string

JSON String
Students: **[**
{“id”: 123, “name” : “Alice”**}**
{“id”: 987, “name” : “Bob”}
]

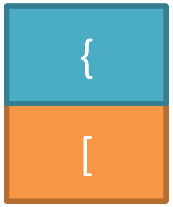
Open Parens: [, {, (, <
Close parens:], },), >

Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Current character
{

Stack



We can use the simple stack to easily match parens in a string

JSON String

```
Students: [
{“id”: 123, “name” : “Alice”}
{“id”: 987, “name” : “Bob”}
]
```

Open Parens: [, {, (, <

Close parens:], },), >

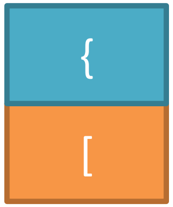
Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Current character

}

Stack



We can use the simple stack to easily match parens in a string

JSON String
Students: **[**
{“id”: 123, “name” : “Alice”**}**
{“id”: 987, “name” : “Bob”**}**
]

Open Parens: [, {, (, <
Close parens:], },), >

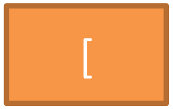
Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack -> {   matches current }
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Current character

}

Stack



We can use the simple stack to easily match parens in a string

JSON String
Students: [
{“id”: 123, “name” : “Alice”}
{“id”: 987, “name” : “Bob”}
]

Open Parens: [, {, (, <
Close parens:], },), >

Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Current character

]

Stack

[

We can use the simple stack to easily match parens in a string

JSON String
Students: **[**
{“id”: 123, “name” : “Alice”**}**
{“id”: 987, “name” : “Bob”**}**
]

Open Parens: [, {, (, <
Close parens:], },), >

Pseudo code

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack -> [   matches current ]
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

Current character
]

Stack

We can use the simple stack to easily match parens in a string

JSON String

Students: **[**
{“id”: 123, “name” : “Alice”**}**
{“id”: 987, “name” : “Bob”**}**
]

Open Parens: [, {, (, <
Close parens:], },), >

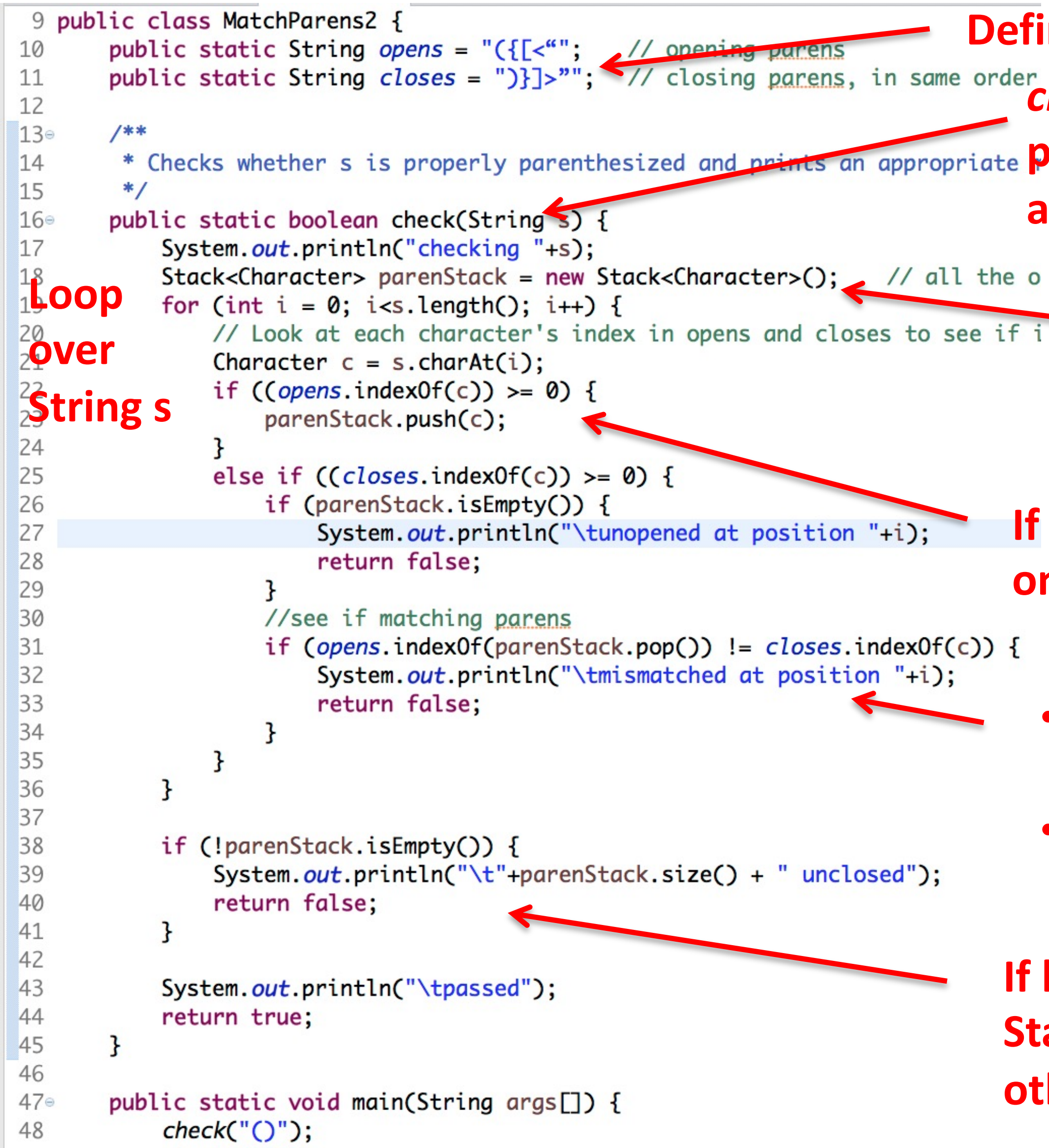
Pseudo code

Current character

Stack

```
Parse each letter
If open paren, add to stack
If close paren
    If stack empty then invalid (close without an open)
    Pop stack
    Invalid if popped element doesn’t match close paren
If end of string and empty stack, valid, else not valid
```

MatchParens2.java uses Java’s Stack to check matching parenthesis



```java
public class MatchParens2 {
    public static String opens = "({[<“";    // opening parens
    public static String closes = ")}]>”";   // closing parens, in same order

    /**
     * Checks whether s is properly parenthesized and prints an appropriate
     */
    public static boolean check(String s) {
        System.out.println("checking "+s);
        Stack<Character> parenStack = new Stack<Character>();    // all the o
        for (int i = 0; i<s.length(); i++) {
            // Look at each character's index in opens and closes to see if i
            Character c = s.charAt(i);
            if ((opens.indexOf(c)) >= 0) {
                parenStack.push(c);
            }
            else if ((closes.indexOf(c)) >= 0) {
                if (parenStack.isEmpty()) {
                    System.out.println("\tunopened at position "+i);
                    return false;
                }
                //see if matching parens
                if (opens.indexOf(parenStack.pop()) != closes.indexOf(c)) {
                    System.out.println("\tmismatched at position "+i);
                    return false;
                }
            }
        }

        if (!parenStack.isEmpty()) {
            System.out.println("\t"+parenStack.size() + " unclosed");
            return false;
        }

        System.out.println("\tpassed");
        return true;
    }

    public static void main(String args[]) {
        check("()");
```

Define open and matching close parens

check() will see if a string s is properly formatted with open and close parens

Loop over String s

Create new Stack of Characters to hold open parens

If find open paren character, push it onto Stack

- If find close paren character, make sure Stack not empty, and *pop()*
- Check popped open Character matches close paren character

If handled all characters, see if Stack empty, fail if not empty, otherwise pass