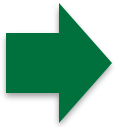
CS 10:
Problem solving via Object Oriented Programming

Keeping order

Agenda



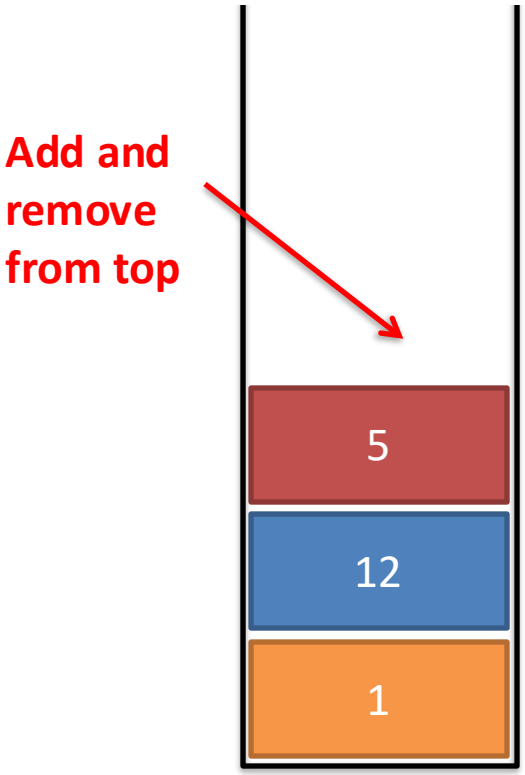
1. Stacks
2. Queues

Key points:

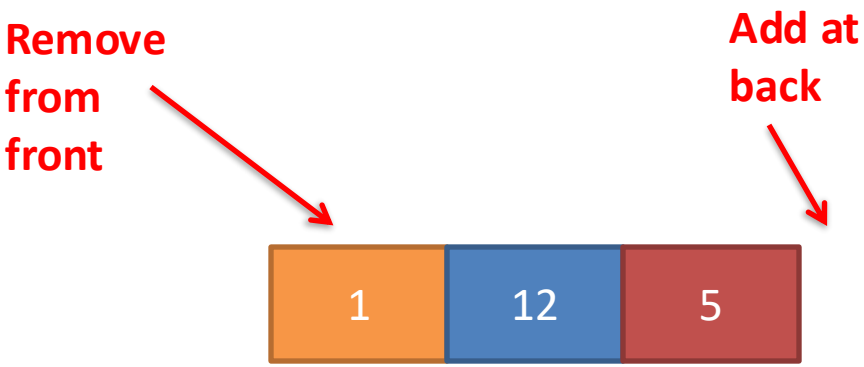
1. **Stacks are LIFO (last in, first out)**
2. **Use for recursive calls, parenthesis matching, reversing items in collection**
3. **Stacks are easy to implement with a linked list or an array**

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

Items inserted in order: 1, 12, 5



Stack (LIFO)



Queue (FIFO)

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

Stack overview

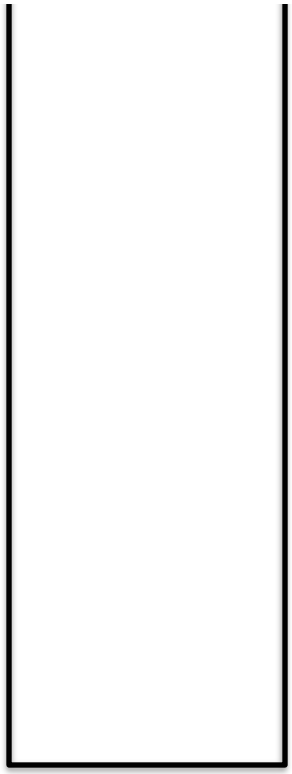
- Think of stack of dinner plates (or Pez dispenser)
- Add item to the top, others move down
- To remove, take top item (last one inserted)
- **Operations**
  - `push` – add item to top of stack
  - `pop` – remove top item and return it
  - `peek` – return top item, but don't remove it
  - `isEmpty` – true if stack empty, false otherwise

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

- Commonly used in CS – recursive function calls, parenthesis matching, reversing items in collection…

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

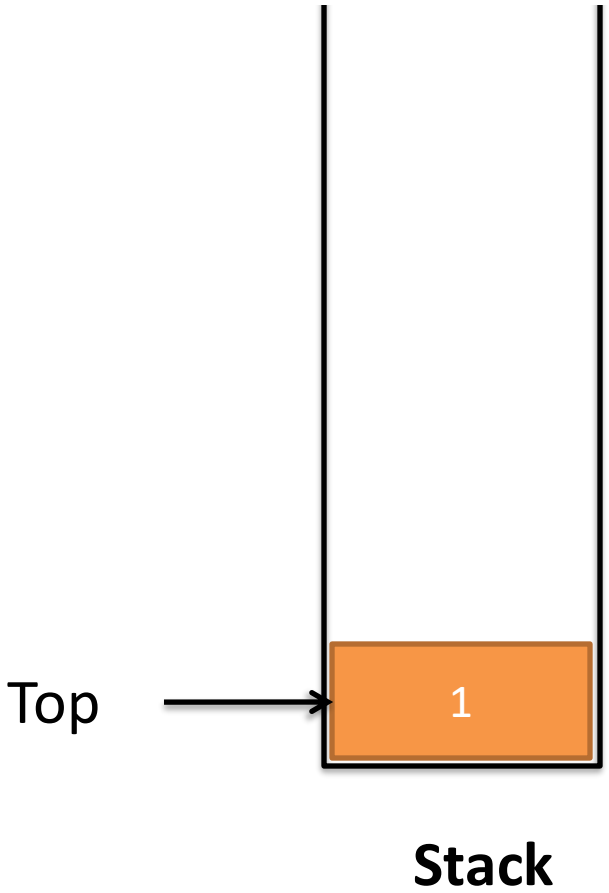
Initially empty



Stack

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

push(1)

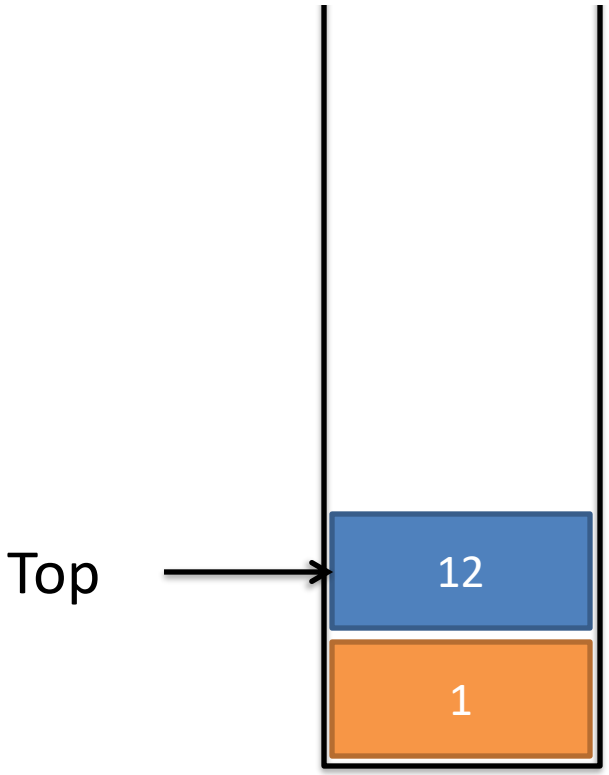


Operations

Push 1

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

push(12)



Stack

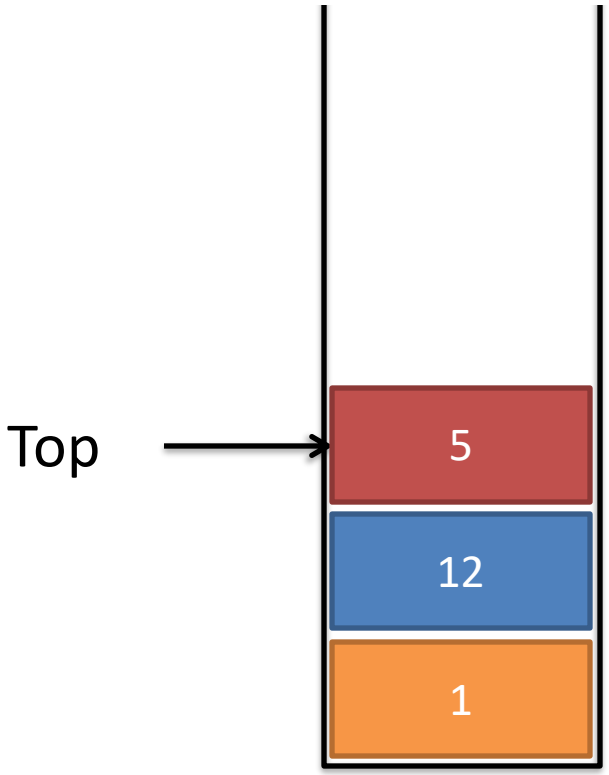
Operations

Push 1

Push 12

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

push(5)



Stack

Operations

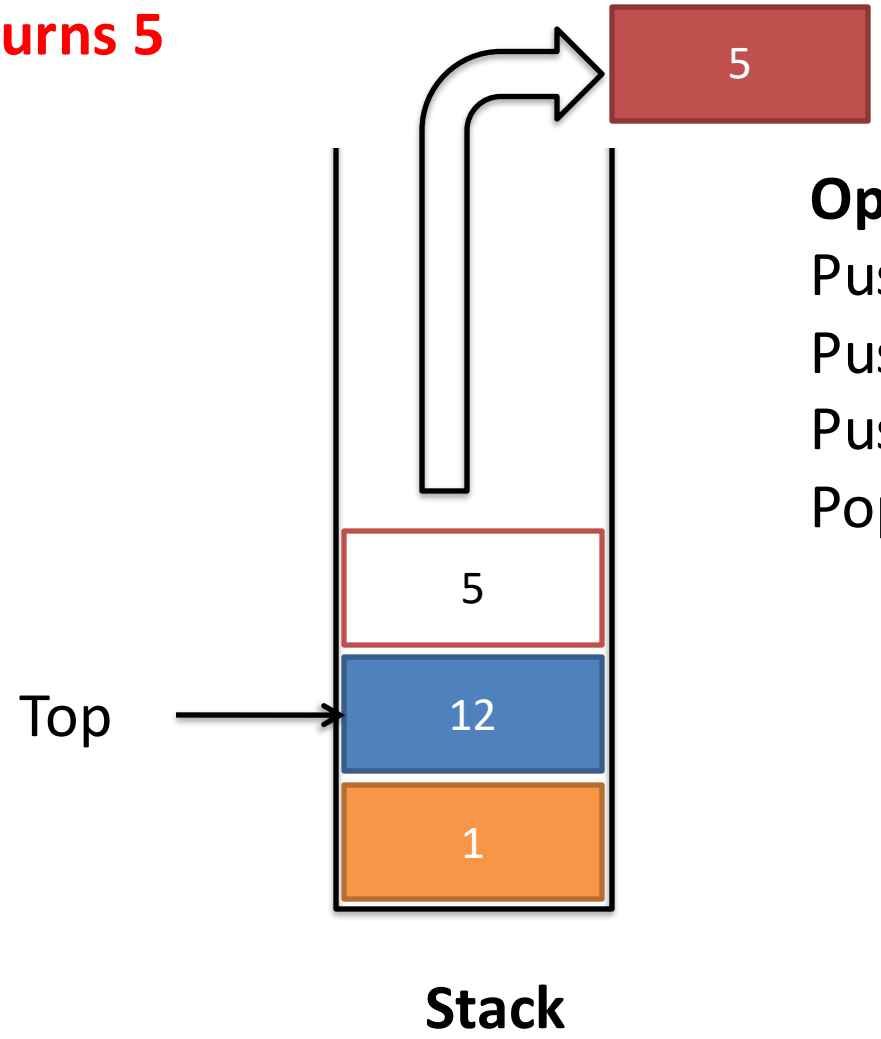
Push 1

Push 12

Push 5

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

pop() –> returns 5



Stack

Operations

Push 1

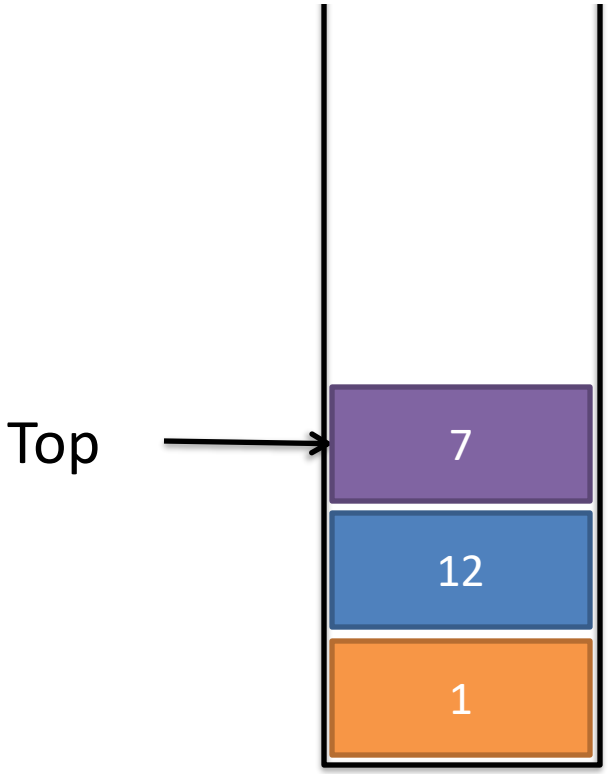
Push 12

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

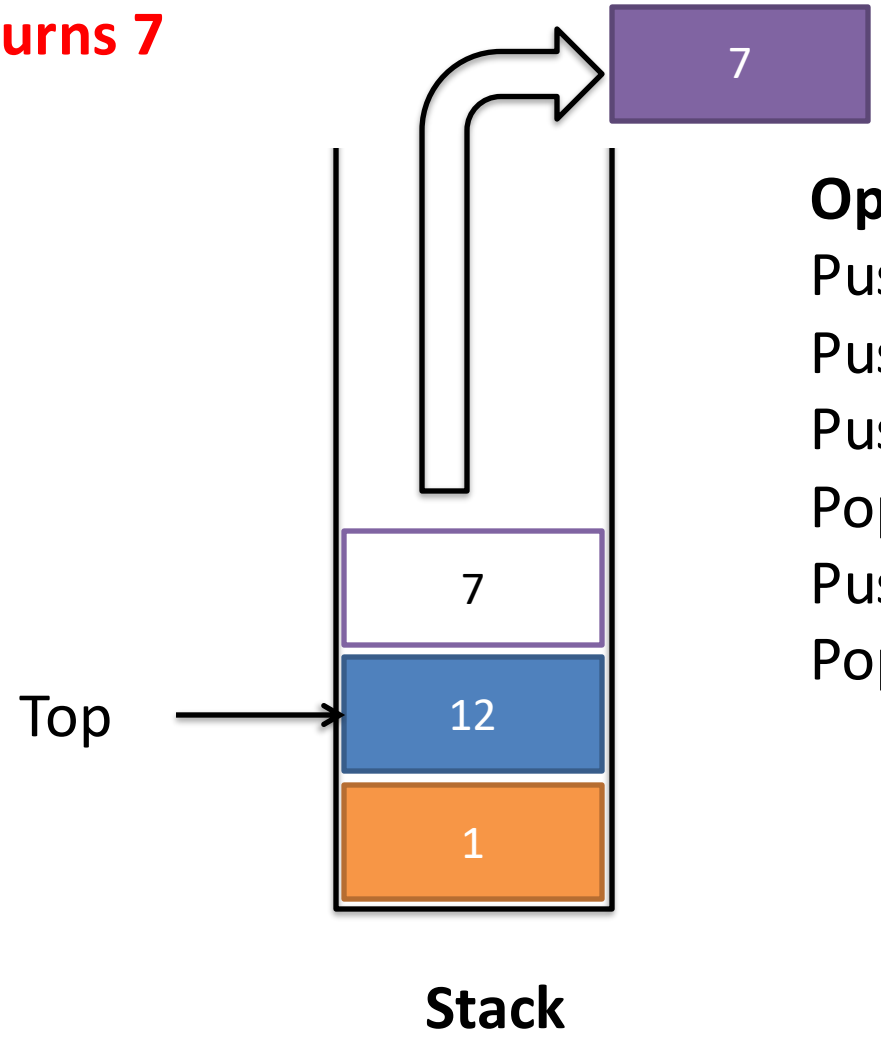
Push 5

Pop – returns 5

Push 7

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

pop() –> returns 7



Stack

Operations

Push 1

Push 12

Push 5

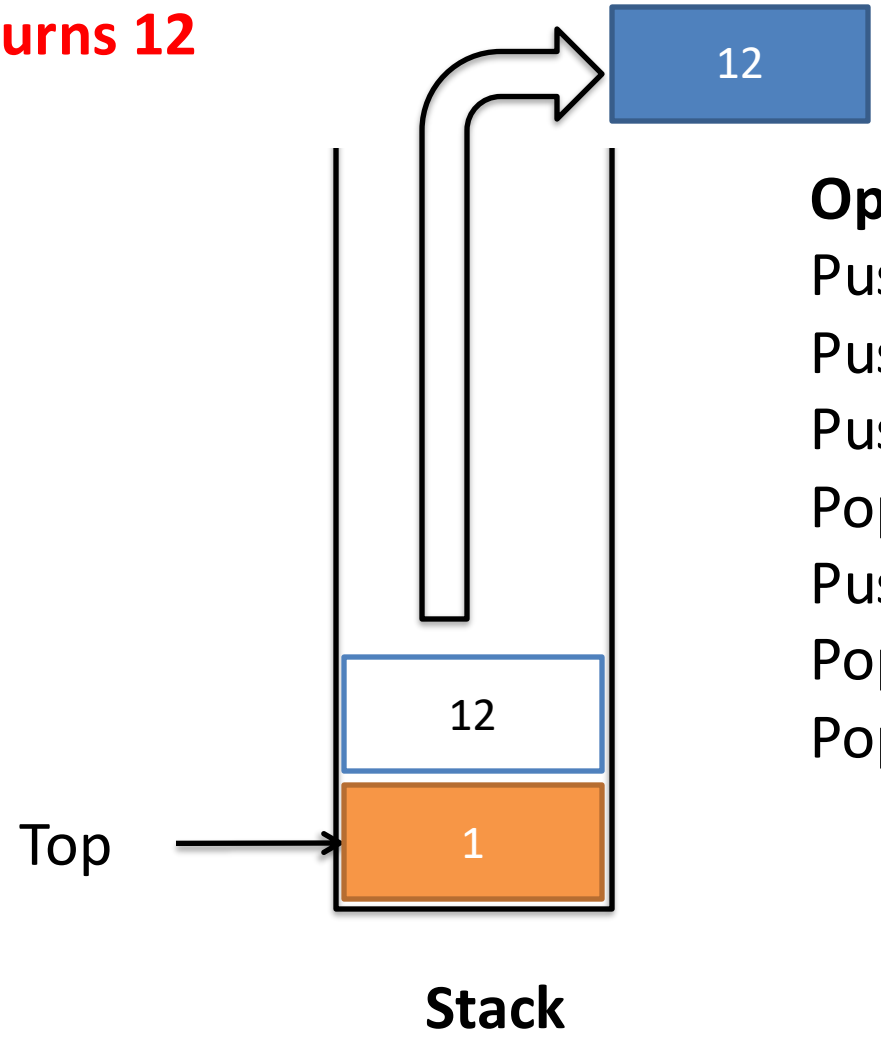
Pop – returns 5

Push 7

Pop – returns 7

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

pop() –> returns 12



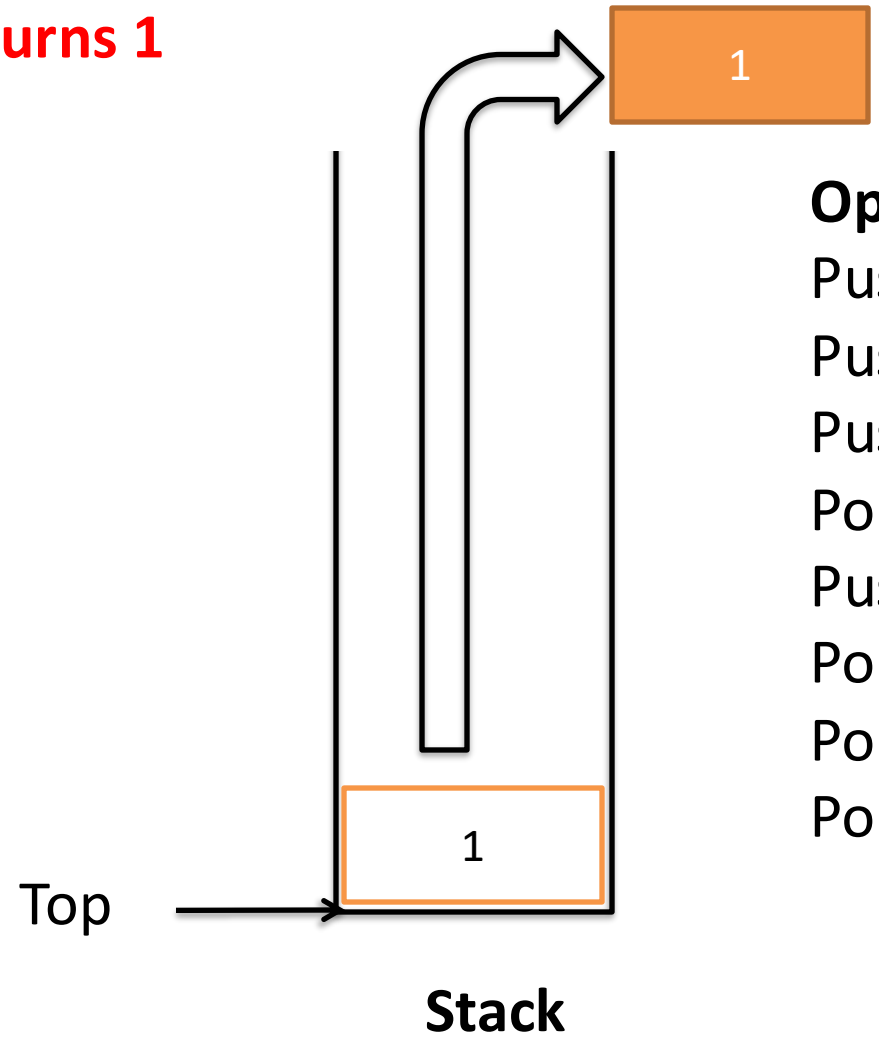
Stack

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

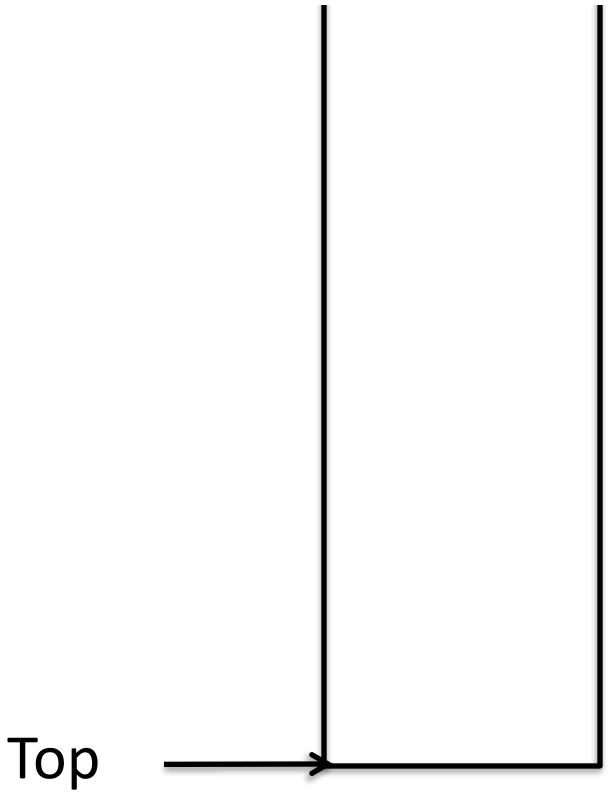
Pop – returns 7

Pop – returns 12

Pop – returns 1

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

pop() –> throw exception



Stack

Operations

Push 1
Push 12
Push 5
Pop – returns 5
Push 7
Pop – returns 7
Pop – returns 12
Pop – returns 1
Pop – throw exception

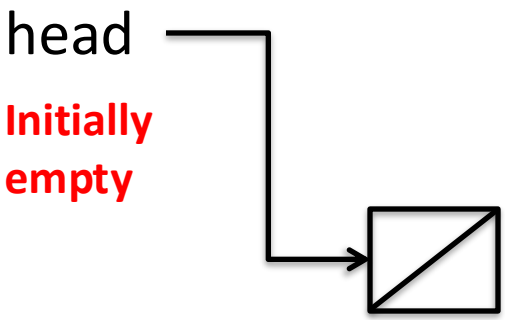
SimpleStack.java: Interface defining Stack operations

```
 7 public interface SimpleStack<T> {
 8     /**
 9      * Add an element onto the top of the stack
10      * @param element element to be pushed onto the stack
11      */
12     public void push(T element);
13     /**
14      * Remove and return the top element
15      * @return an element from the top of the stack.
16      */
17     public T pop() throws Exception;
18     /**
19      * Look at the top element without removing it
20      * @return the element on the top of the stack without changing it.
21      */
22     public T peek() throws Exception;
23     /**
24      * Is the stack empty?
25      * @return true iff stack is empty
26      */
27     public boolean isEmpty();
28 }
```

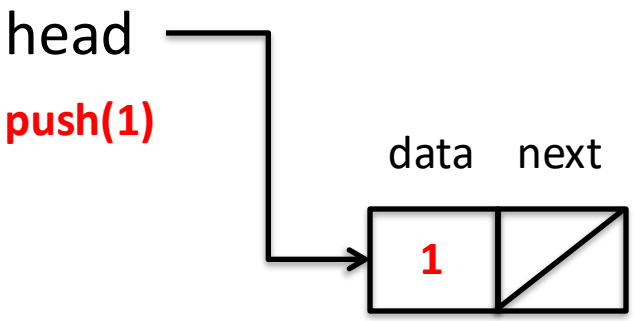
As with other ADTs, we use generics because we don't really care what kind of data the Stack will hold

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

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

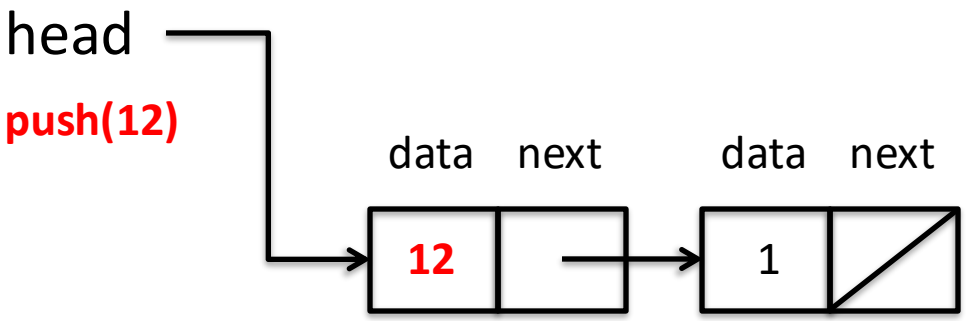


Add at front of linked list

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

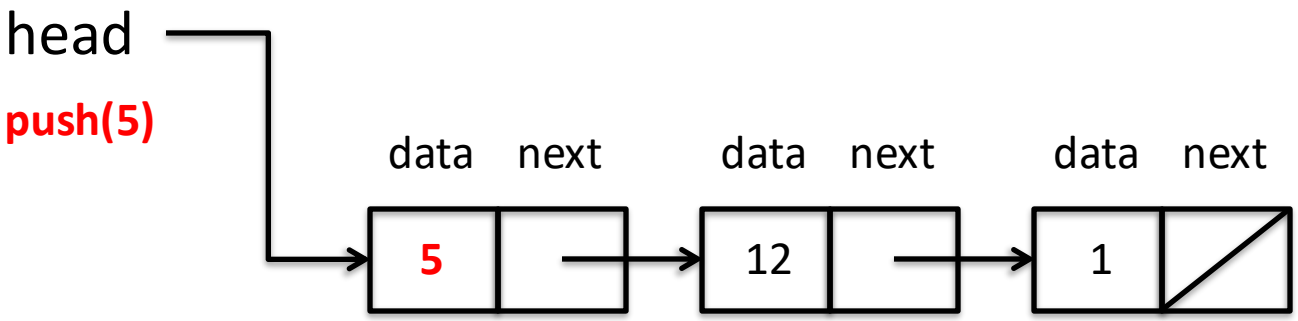


Add at front of linked list

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

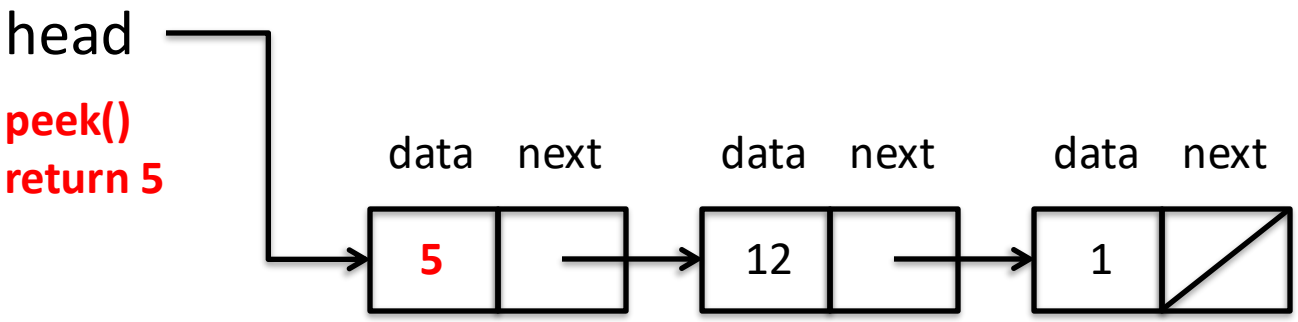


Add at front of linked list

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

Set *head* to new element (5)

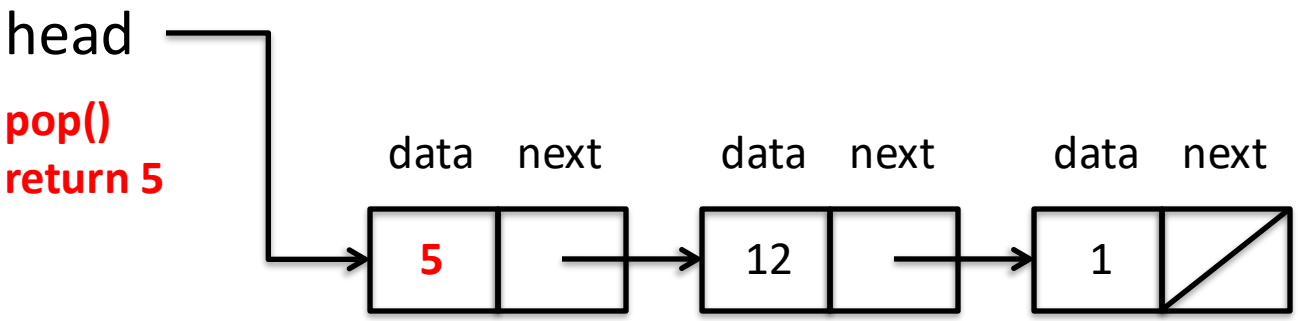
A Singly Linked List works well for a Stack, using top as head of list



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



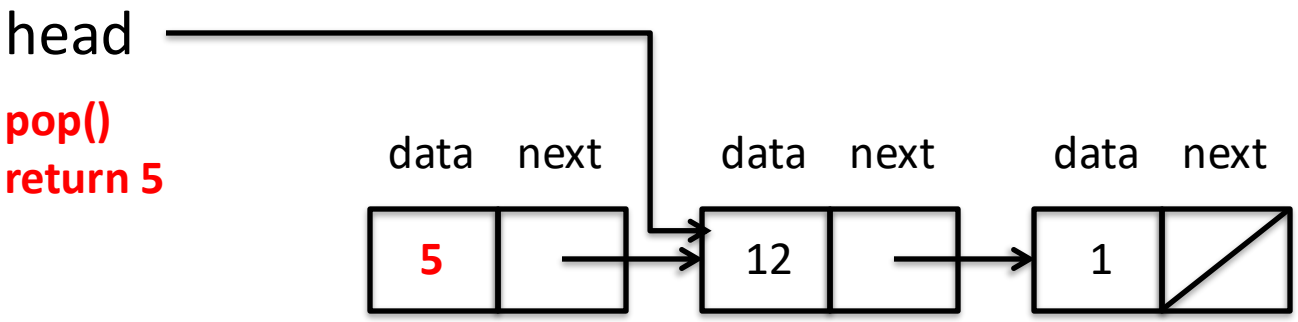
Add at front of linked list

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

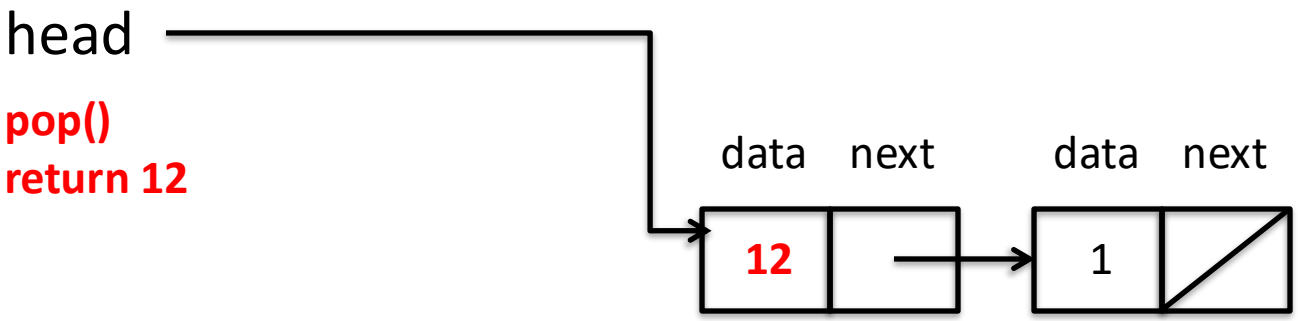
Peek returns data from first element or throw exception if empty

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



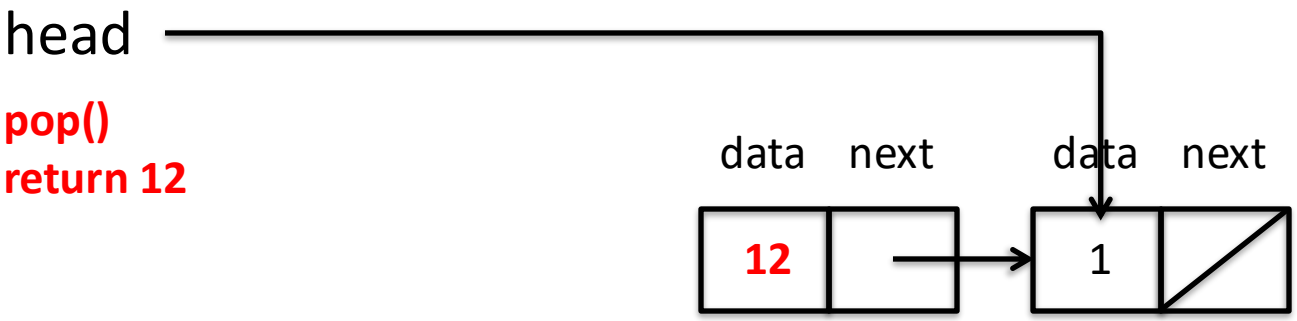
Add at front of linked list

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



Add at front of linked list

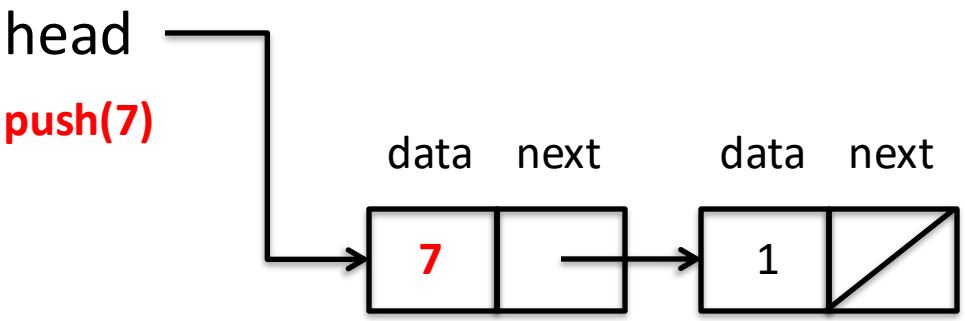
Peek returns data from first element or throw exception if empty

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

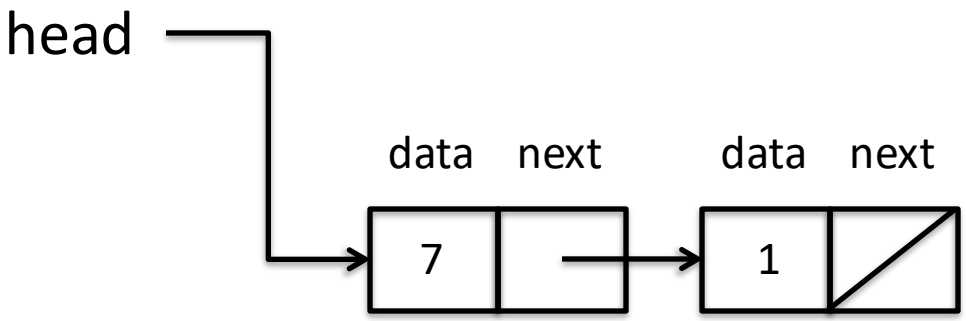
Set new element *next* to *head* (1)

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 O(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

```
 8 public class SLLStack<T> implements SimpleStack<T> {
 9     private Element top;// top of the stack
10
11     /**
12      * The linked elements
13      */
14     private class Element {
15         private T data;
16         private Element next;
17
18         public Element(T data, Element next) {
19             this.data = data;
20             this.next = next;
21         }
22     }
23
24     public SLLStack() {
25         top = null;
26     }
27
28     public boolean isEmpty() {
29         return top == null;
30     }
31
32     public T peek() throws Exception {
33         if (isEmpty()) throw new Exception("empty stack");
34         return top.data;
35     }
36
37     public T pop() throws Exception {
38         if (isEmpty()) throw new Exception("empty stack");
39         T data = top.data;
40         top = top.next;
41         return data;
42     }
43
44     public void push(T element) {
45         top = new Element(element, top);
46     }
47 }
```

Implements SimpleStack interface, so must implement its methods

Private Element class as we've seen before
Data is of generic type T

- All operations at the head, O(1)
- Unlike an array, SLL does not run out of space

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

isEmpty check head == null

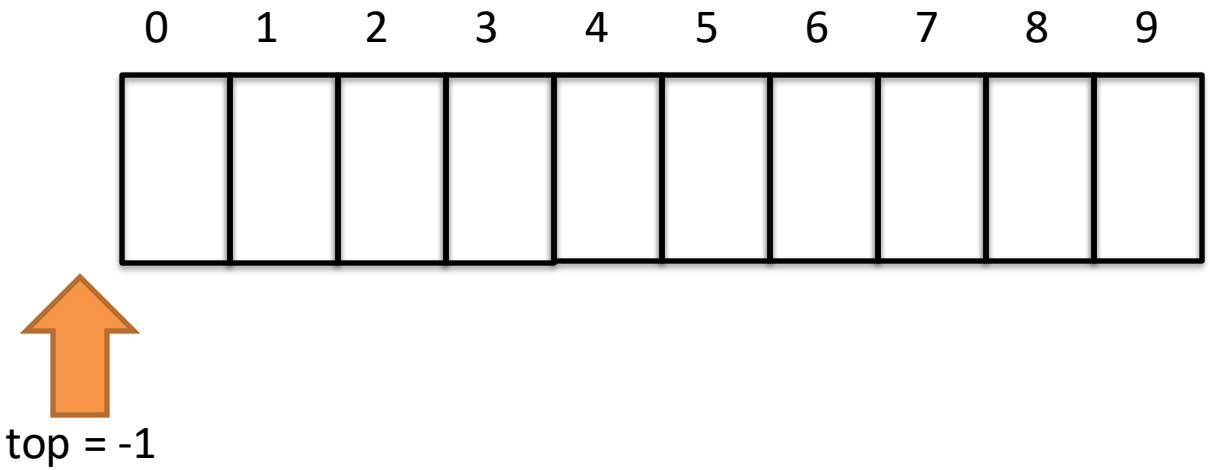
peek() returns *data* of first Element in list but does not remove it

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

We can implement a Stack using an array

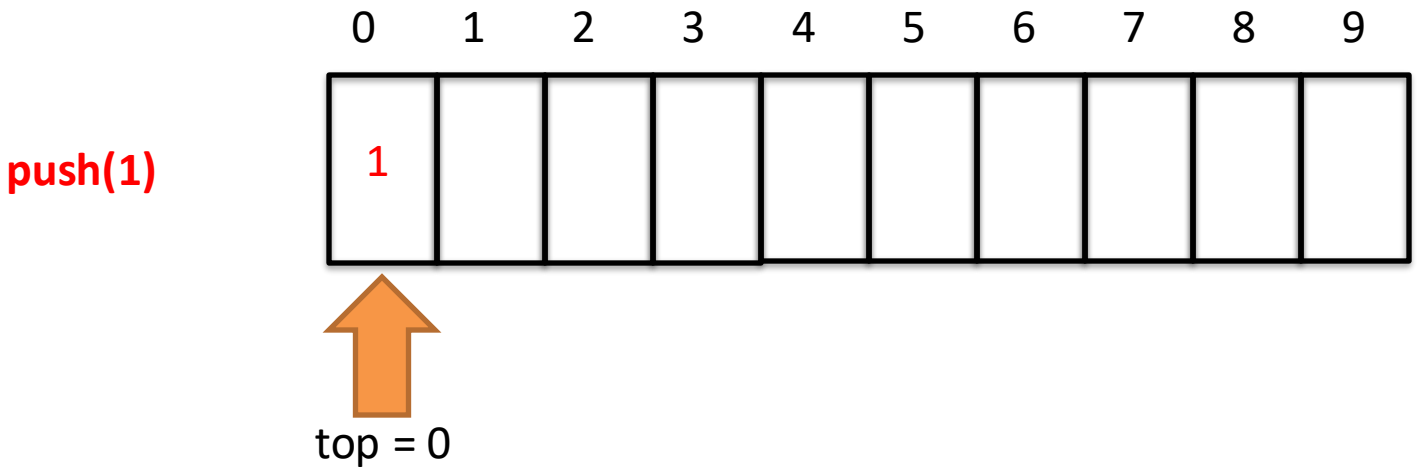
Stack array implementation



Create array and set *top* = -1

We can implement a Stack using an array

Stack array implementation

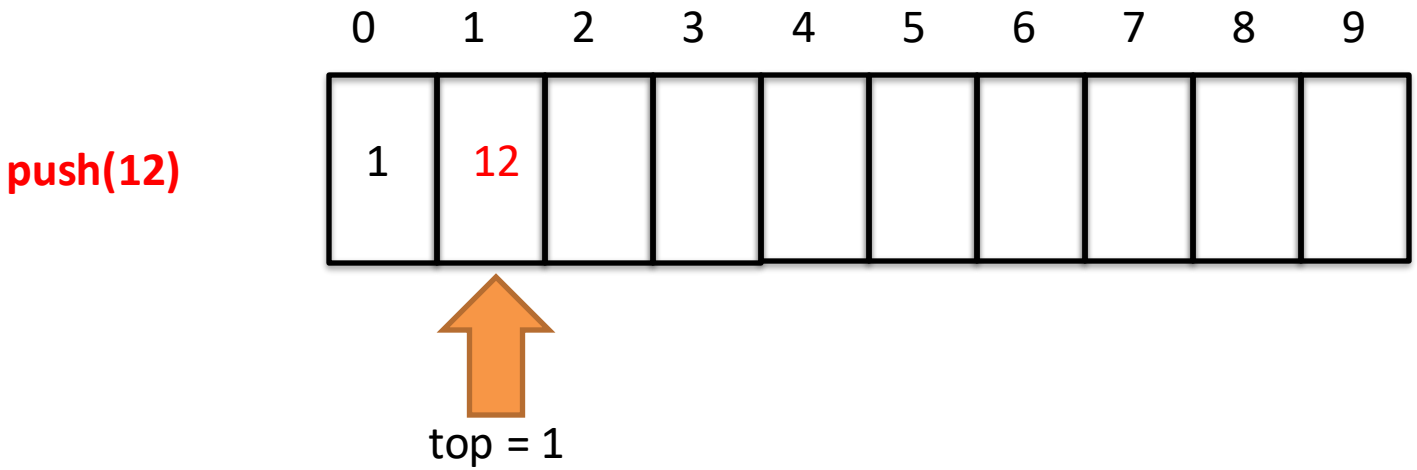


Create array and set *top* = -1

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

We can implement a Stack using an array

Stack array implementation

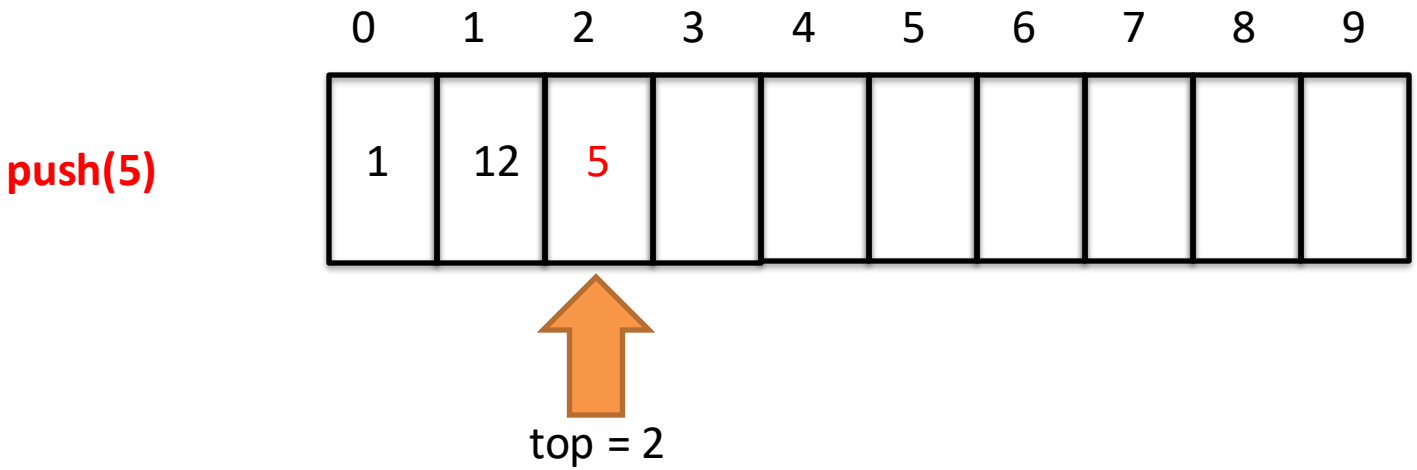


Create array and set *top* = -1

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

We can implement a Stack using an array

Stack array implementation

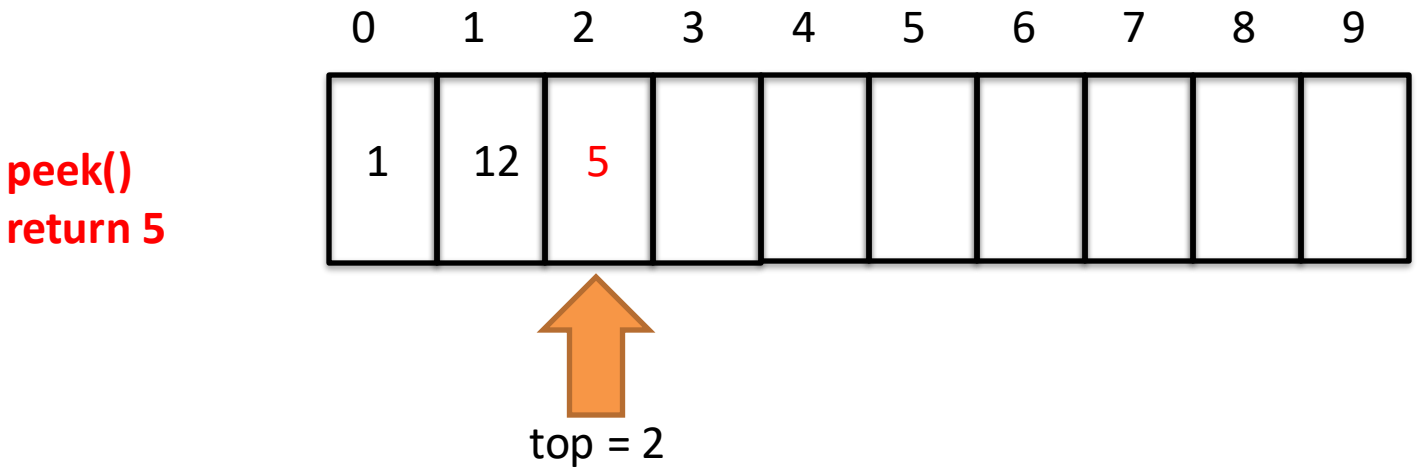


Create array and set *top* = -1

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

We can implement a Stack using an array

Stack array implementation



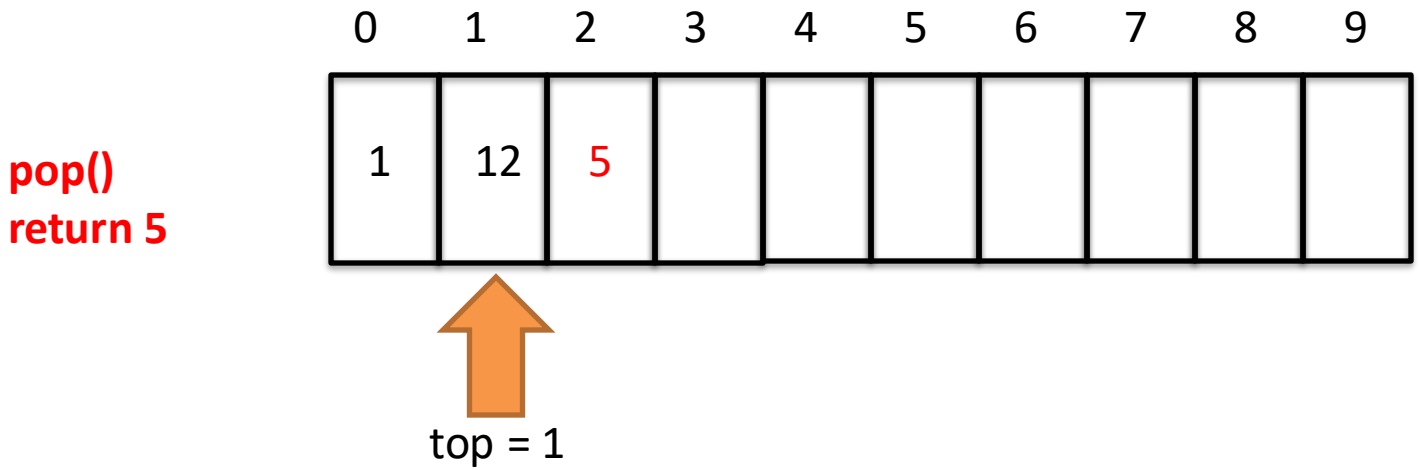
Create array and set *top* = -1

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



Create array and set *top* = -1

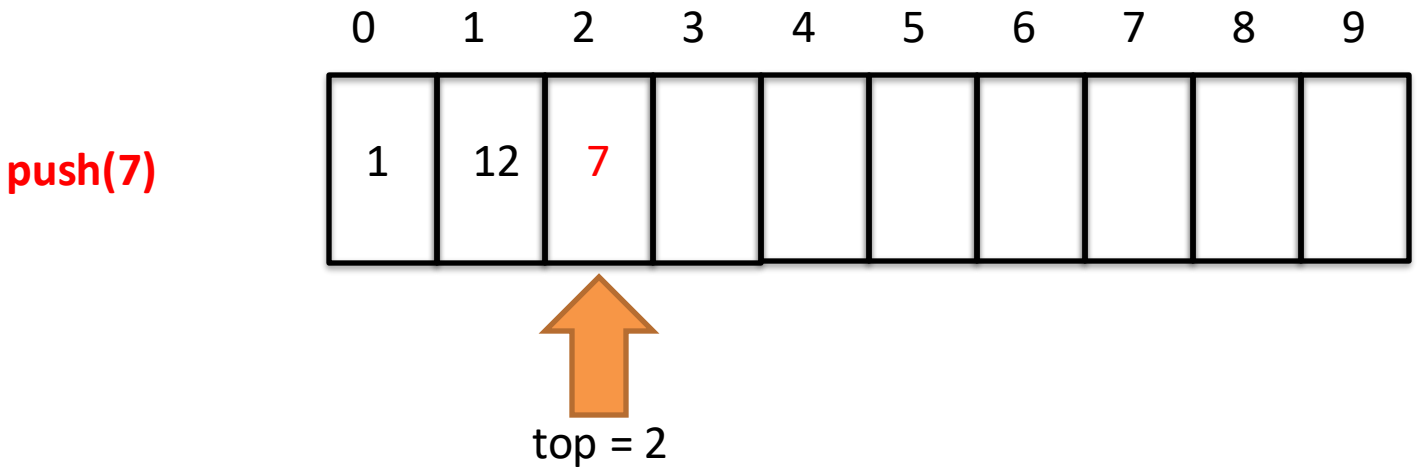
To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

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



Create array and set *top* = -1

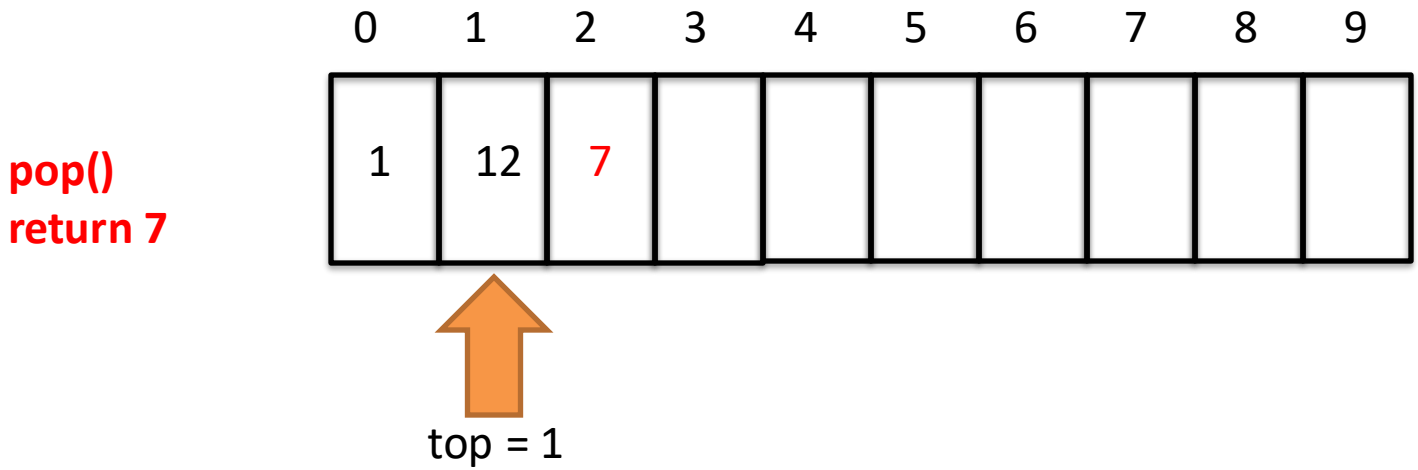
To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

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



Create array and set *top* = -1

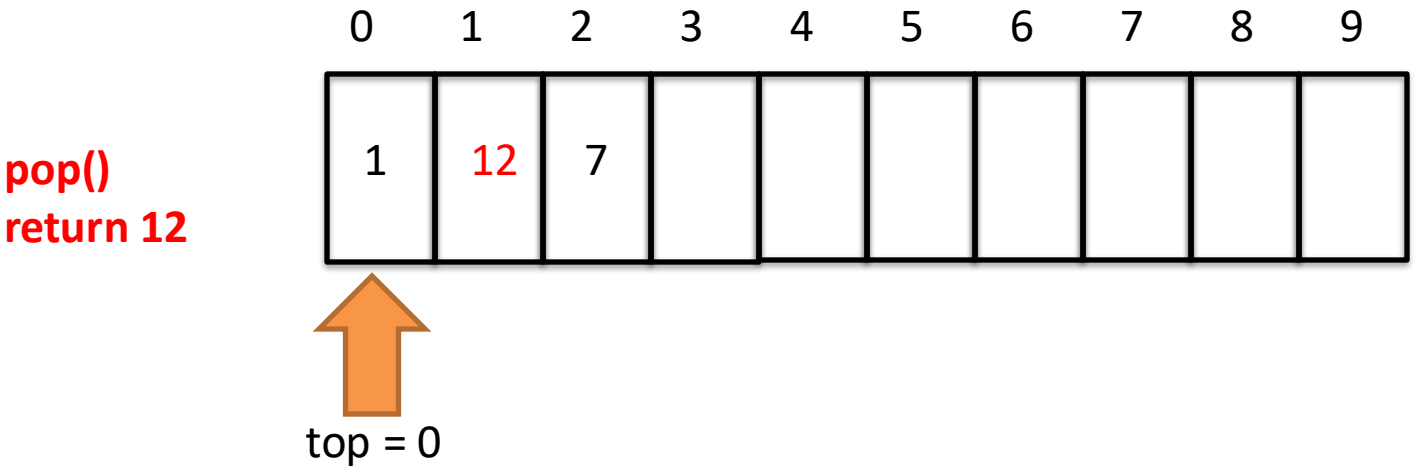
To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

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



Create array and set *top* = -1

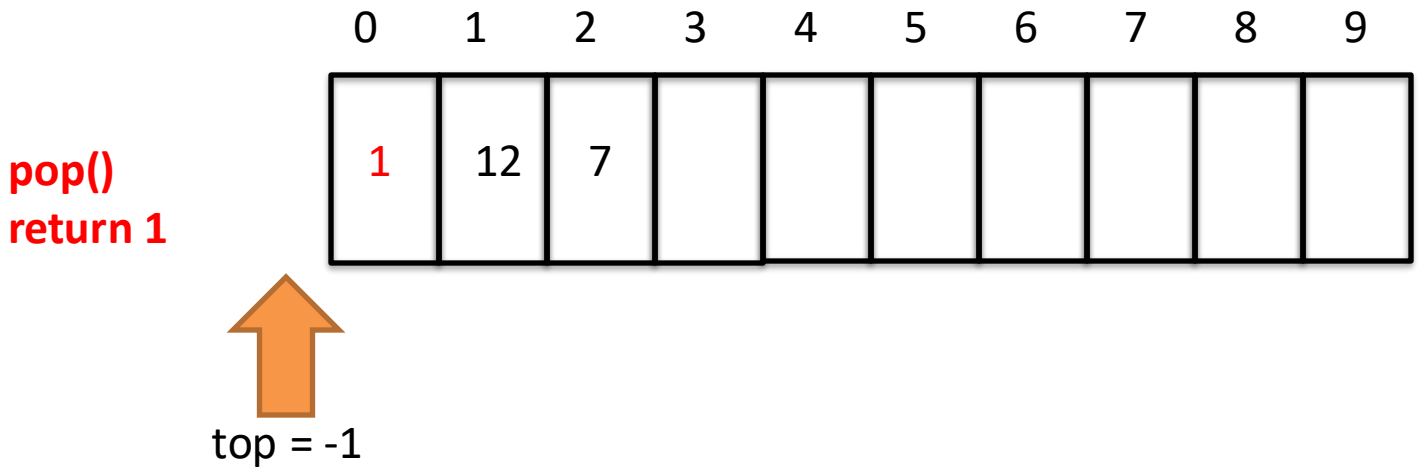
To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

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



Create array and set *top* = -1

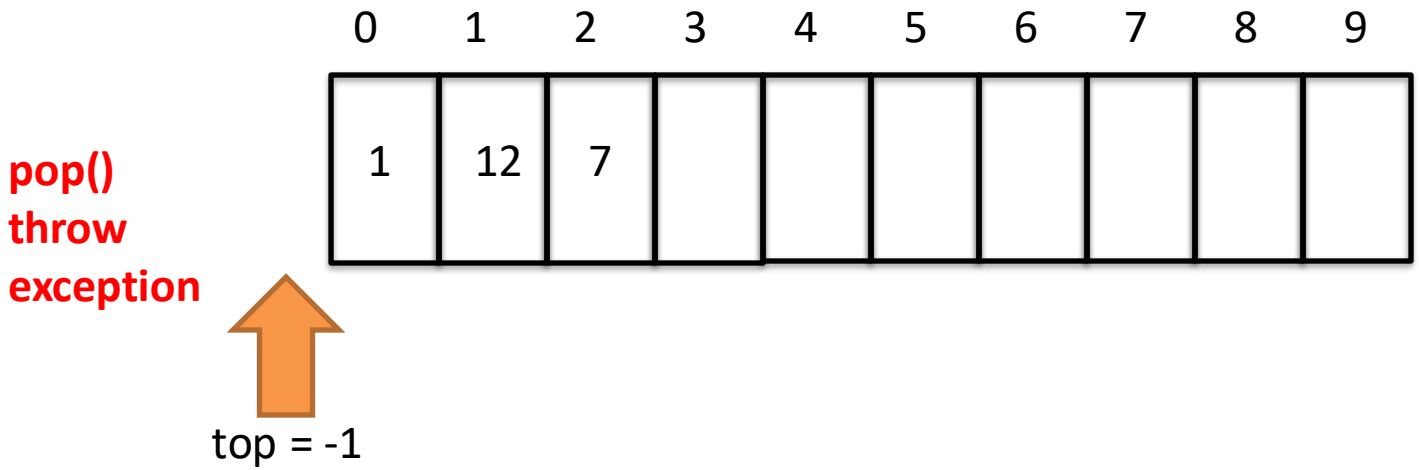
To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

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



Create array and set *top* = -1

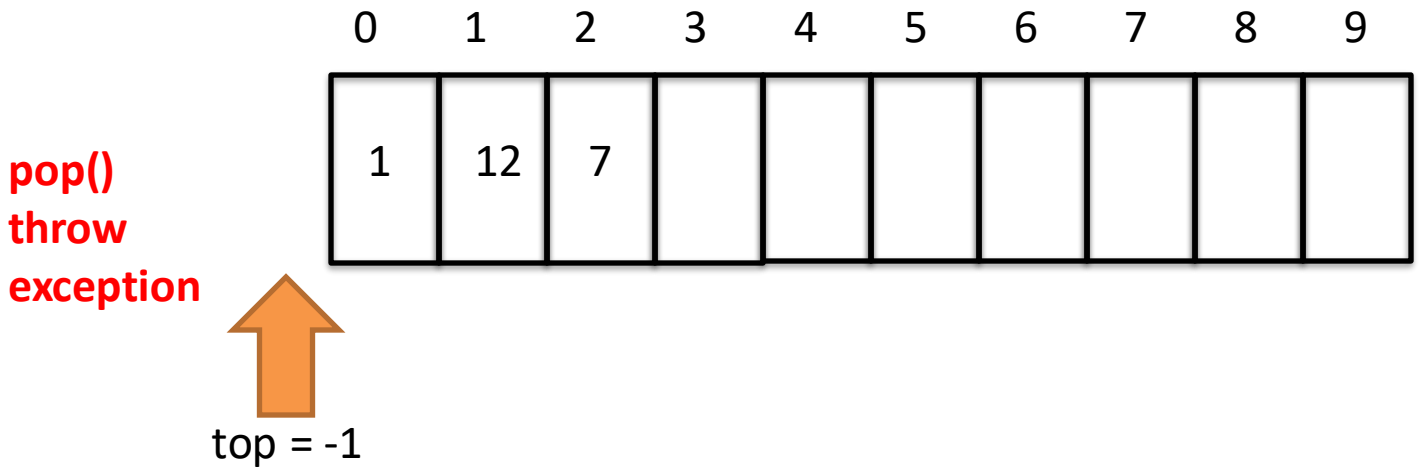
To *push(T elmt)*, add 1 to *top* and *stack[top] = elmt*

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



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

```
 9 public class ArrayListStack<T> implements SimpleStack<T> {
10
11     private ArrayList<T> list;   // Holds the stack
12
13     /**
14      *  Construct an empty stack
15      */
16     public ArrayListStack() {
17         list = new ArrayList<T>();
18     }
19
20     public boolean isEmpty() {
21         return list.size() == 0;
22     }
23
24     public T peek() throws Exception {
25         if (isEmpty())
26             throw new Exception("empty stack");
27         else
28             return list.get(list.size()-1);
29     }
30
31     public T pop() throws Exception {
32         if (isEmpty())
33             throw new Exception("empty stack");
34         else
35             return list.remove(list.size()-1);
36     }
37
38     public void push(T element) {
39         list.add(element);
40     }
```

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)

Agenda

1. Stacks
2. Queues

Key points:

1. **Queues are FIFO (first in, first out)**
2. **Use for modeling things like grocery checkout lines**
3. **Queues are easy to implement with a linked list but more difficult with an array**
4. **Use a circular array to implement a queue**

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

Queue

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

enqueue(12)

enqueue() adds to back

Queue

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

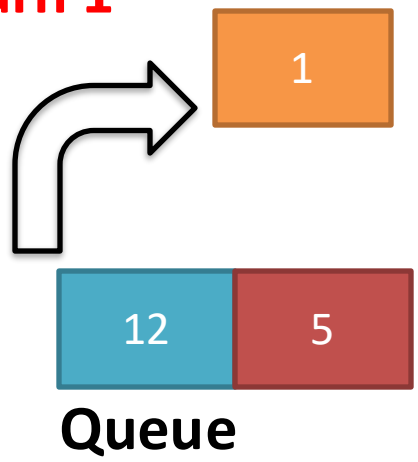
enqueue(5)

enqueue() adds to back

Queue

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

dequeue()
Return 1



dequeue() removes from front

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

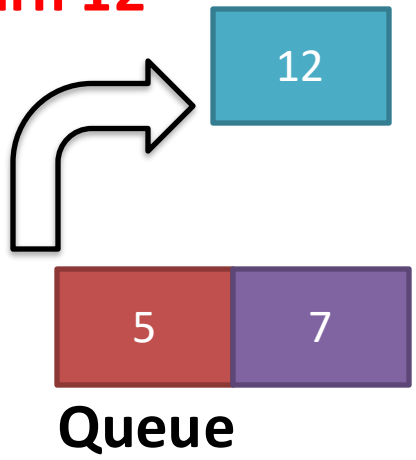
enqueue(7)

enqueue() adds to back

Queue

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

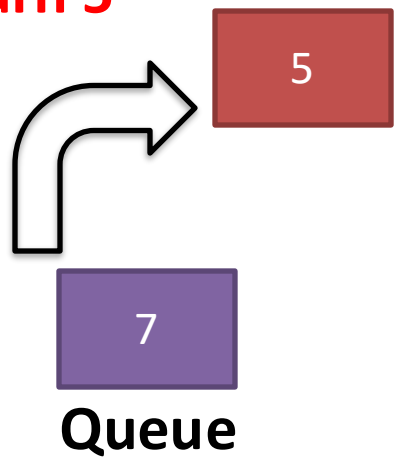
dequeue()
Return 12



dequeue() removes from front

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

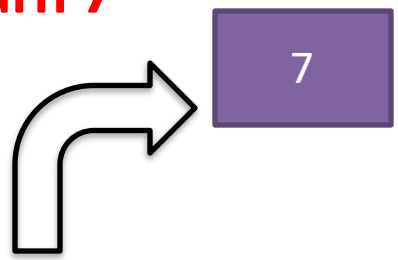
dequeue()
Return 5



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

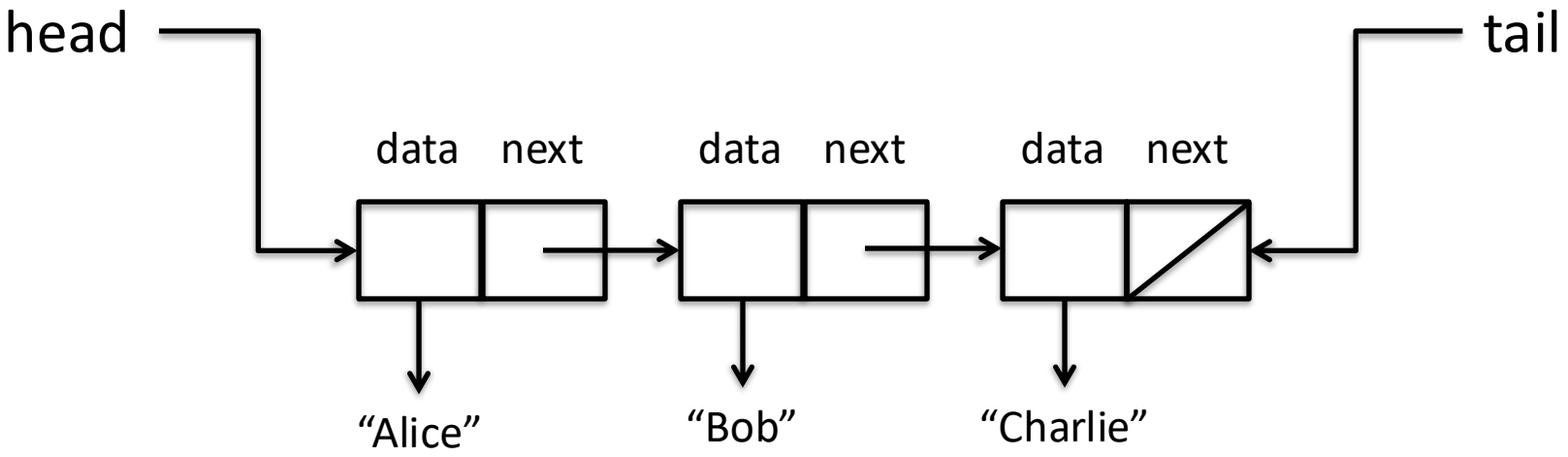
SimpleQueue.java

```
 7 public interface SimpleQueue<T> {
 8
 9     /**
10      * Add item to rear of queue
11      * @param item item to be enqueued
12      */
13     public void enqueue(T item);
14
15     /**
16      * Remove item from front of queue
17      * @return the item removed from the front of the queue
18      */
19     public T dequeue() throws Exception;
20
21     /**
22      * Return the item at the front of queue, but do not remove it
23      * @return the item at the front of the queue
24      */
25     public T front() throws Exception;
26
27     /**
28      * Is the queue empty?
29      * @return true iff queue is empty
30      */
31     public boolean isEmpty();
32
33 }
```

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 O(1)



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

SLLQueue.java

```
 9 public class SLLQueue<T> implements SimpleQueue<T> {
10     private Element head;    // front of the linked list
11     private Element tail;    // tail of the linked list
12
13     /**
14      * The linked elements
15      */
16     private class Element {
17         private T data;
18         private Element next;
19
20         public Element(T data) {
21             this.data = data;
22             this.next = null;
23         }
24     }
25
26     /**
27      *  Creates an empty queue
28      */
29     public SLLQueue()  {
30         head = null;
31         tail = null;
32     }
33
34     public void enqueue(T item) {
35         if (isEmpty()) {
36             // first item
37             head = new Element(item);
38             tail = head;
39         }
40         else {
41             tail.next = new Element(item);
42             tail = tail.next;
43         }
44     }
45
46     public T dequeue() throws Exception {
47         if (isEmpty()) throw new Exception("empty queue");
48         T item = head.data;
49         head = head.next;
50         return item;
51     }
```

Implements SimpleQueue interface

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

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

Array implementation

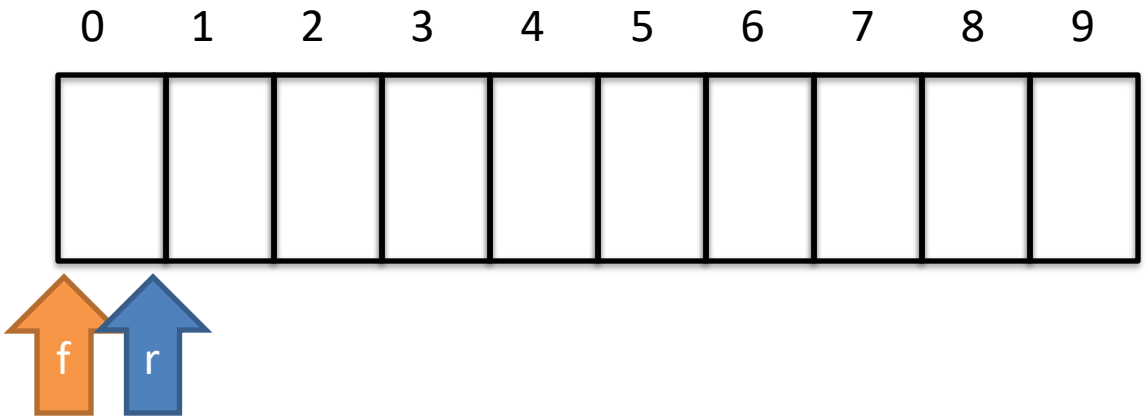
- Could `enqueue` at back, `dequeue` from front
  - `enqueue` is fast, just add item to end O(1)
  - `dequeue` must move all elements left one space O(n)
- Could `enqueue` at front and `dequeue` from back
  - `enqueue` must move all elements right one space O(n)
  - `dequeue` is fast, just take last item O(1)
- Could track `front (f)` and `rear (r)` indexes
  - `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)
  - This approach is called a *circular array*

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**

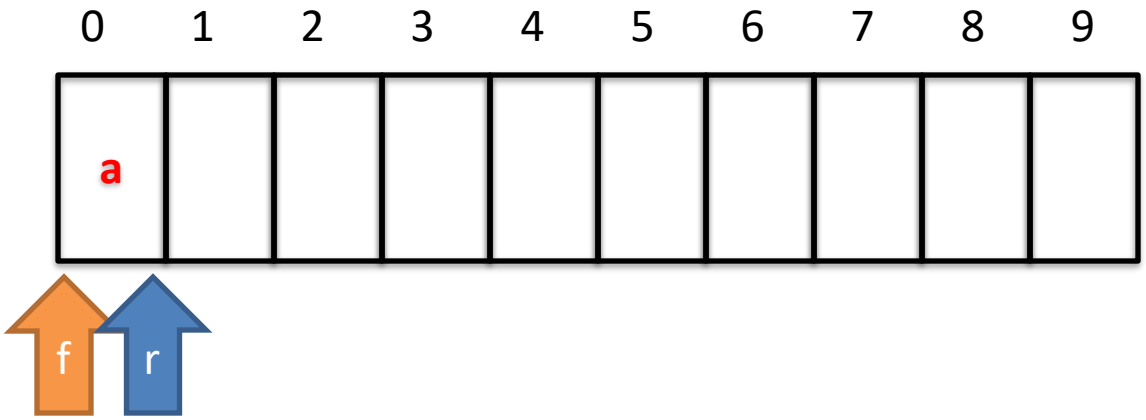
***rear (r)* is index of next free space (initially 0)**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(a)

size = 0



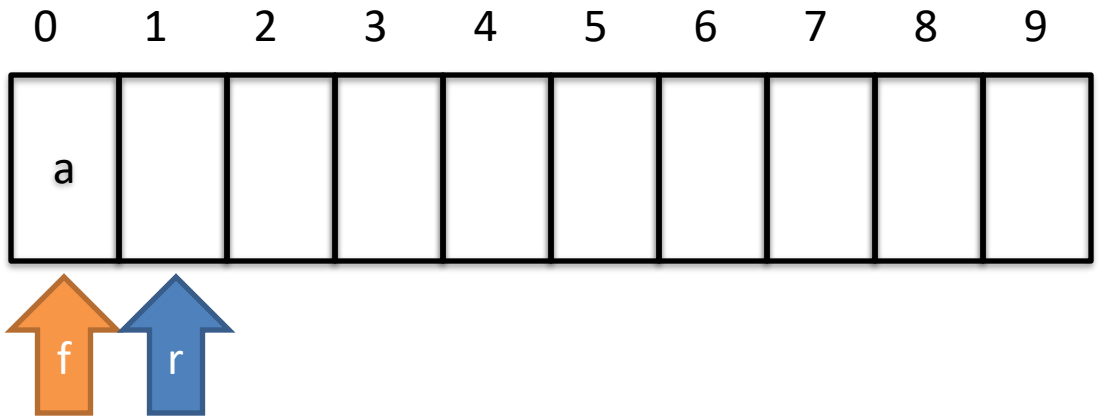
Set *a* at position *rear*

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(a)

size = 1



Set *a* at position *rear*

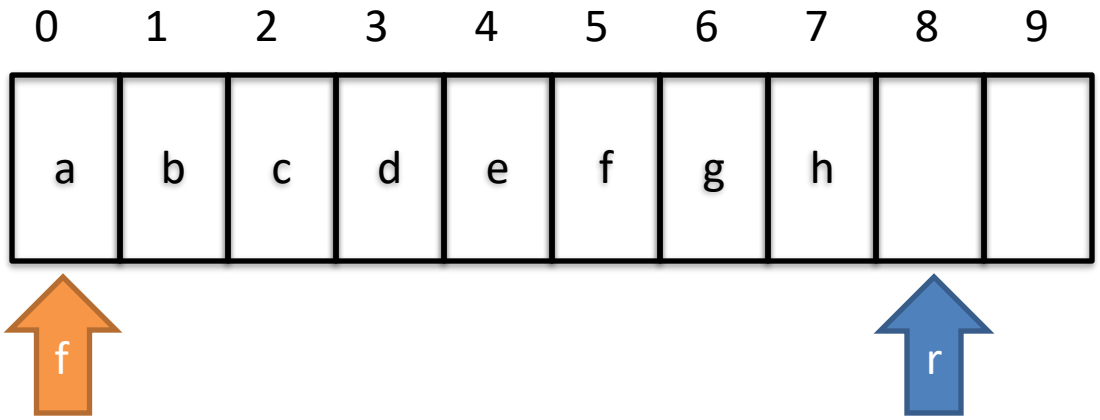
Set *rear* +=1, *size* +=1

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(b)* through *enqueue(h)

size = 8



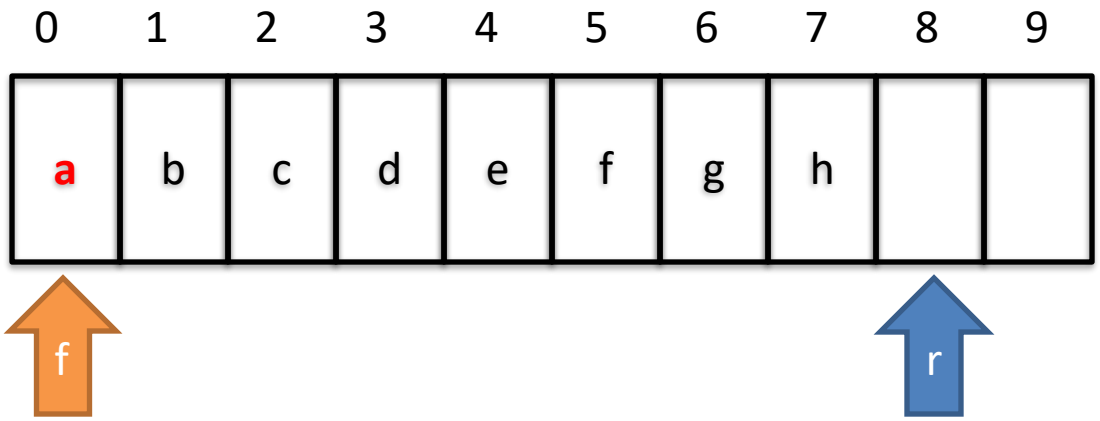
front* stays at 0, *rear* +=1 and *size* +=1 on each *enqueue()

Array implementing a Queue using index for front and rear

Array implementing Queue

dequeue()

size = 8



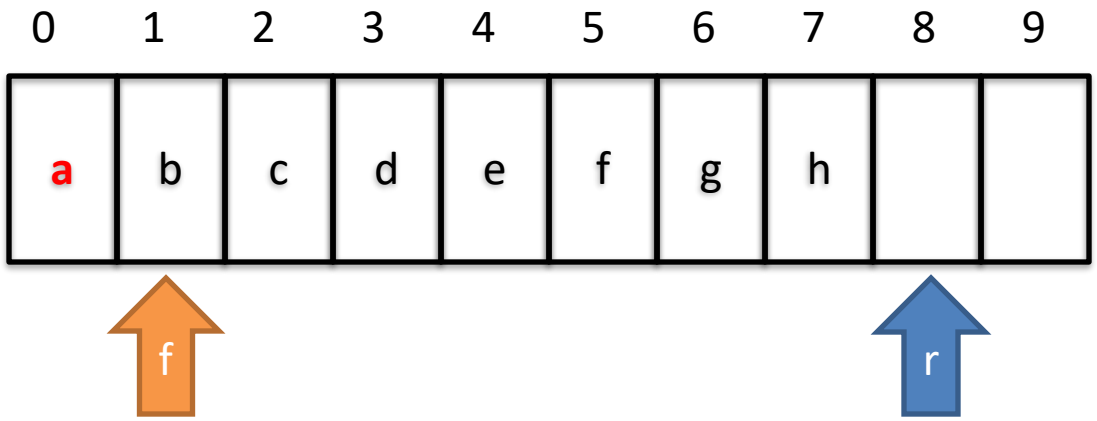
Return item at *front* -> a

Array implementing a Queue using index for front and rear

Array implementing Queue

dequeue()

size = 7



Return item at *front* -> a

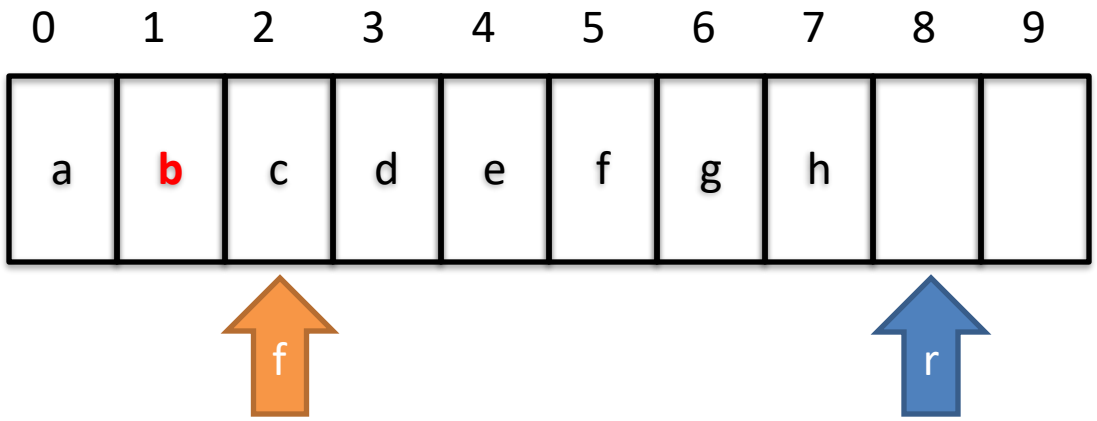
***front* +=1, *size* -=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

dequeue()

size = 6



Return item at *front* -> b

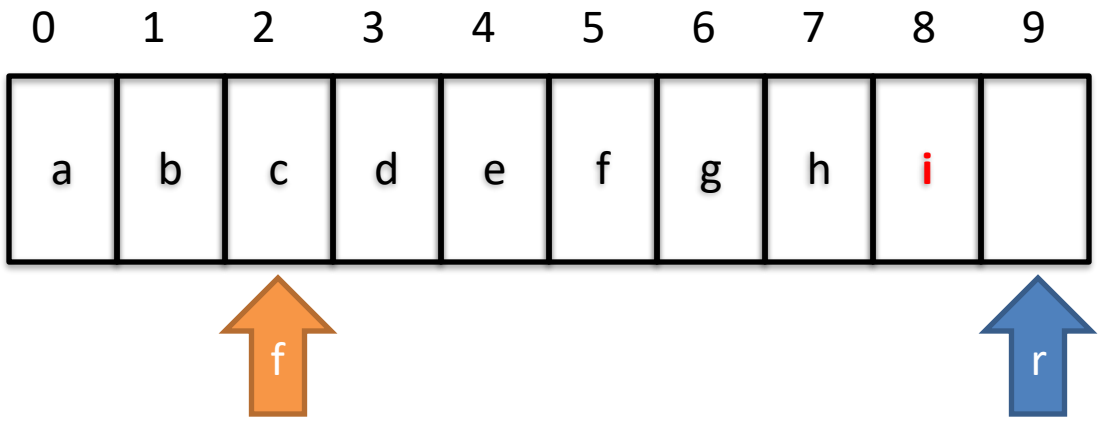
***front* +=1, *size* -=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(i)

size = 7



Set *i* at position *rear*

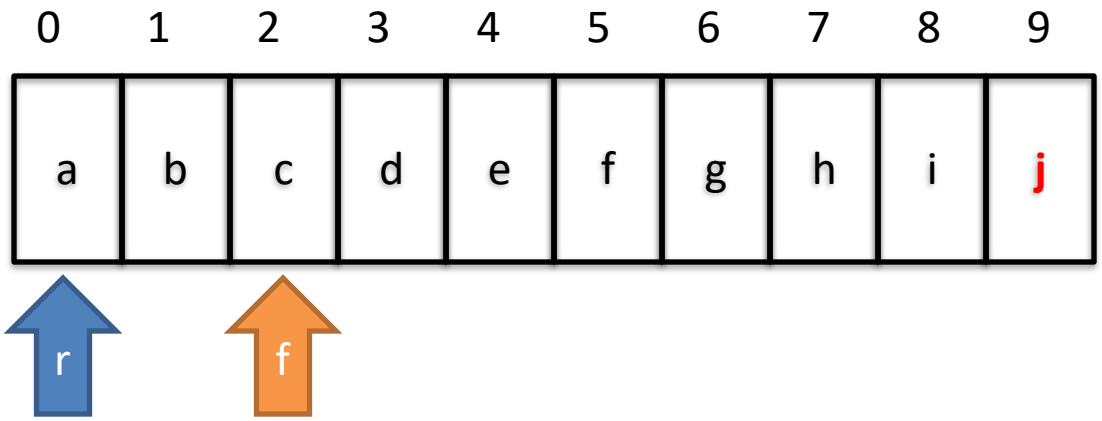
***rear* +=1, *size* +=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(j)

size = 8



Set *j* at position *rear*

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

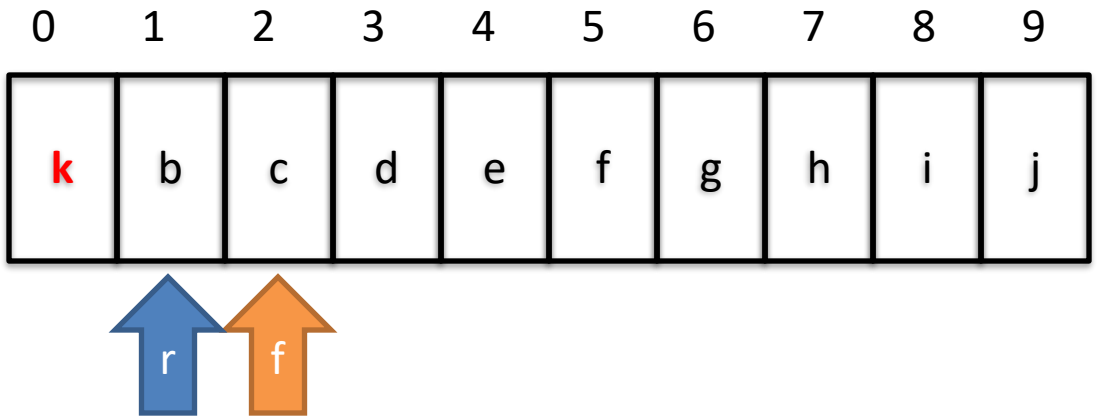
***size* +=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(k)

size = 9



Set *k* at position *rear*

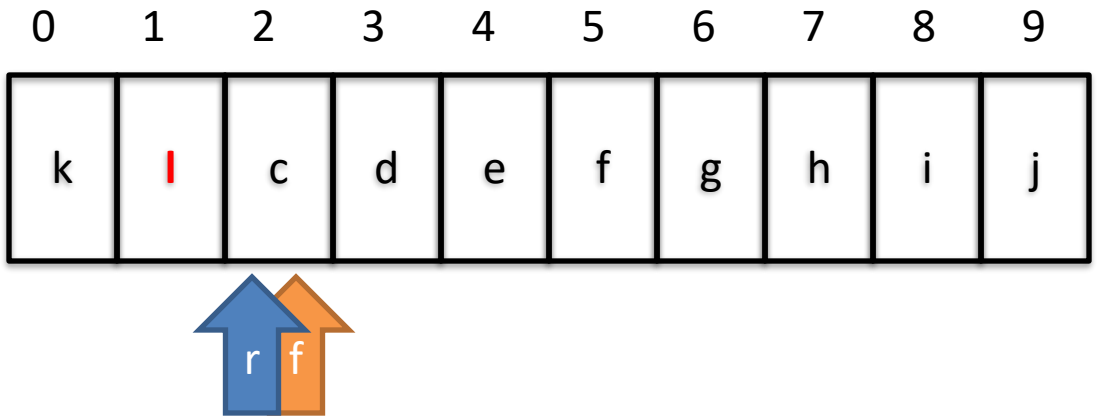
***rear* +=1, *size* +=1**

Array implementing a Queue using index for front and rear

Array implementing Queue

enqueue(l)

size = 10



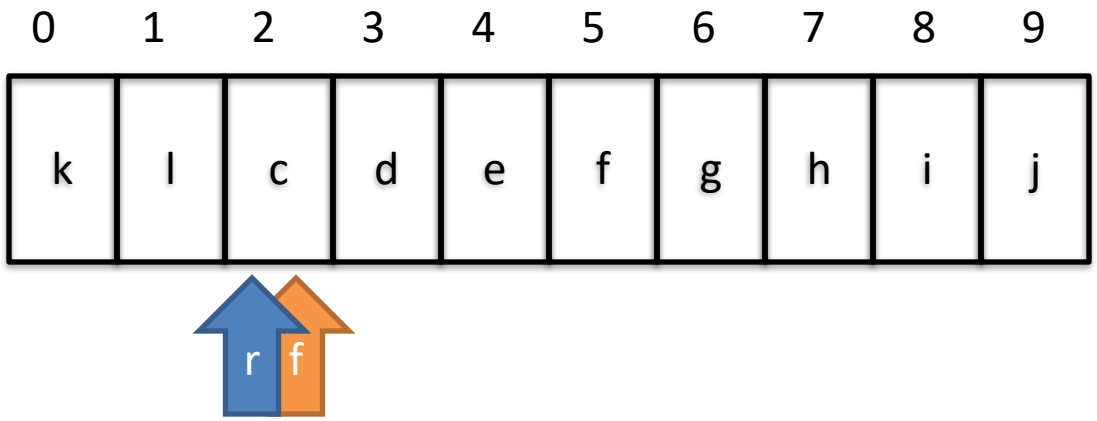
Set *l* at position *rear*

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

Array implementing a Queue using index for front and rear

Array implementing Queue

size = 10



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

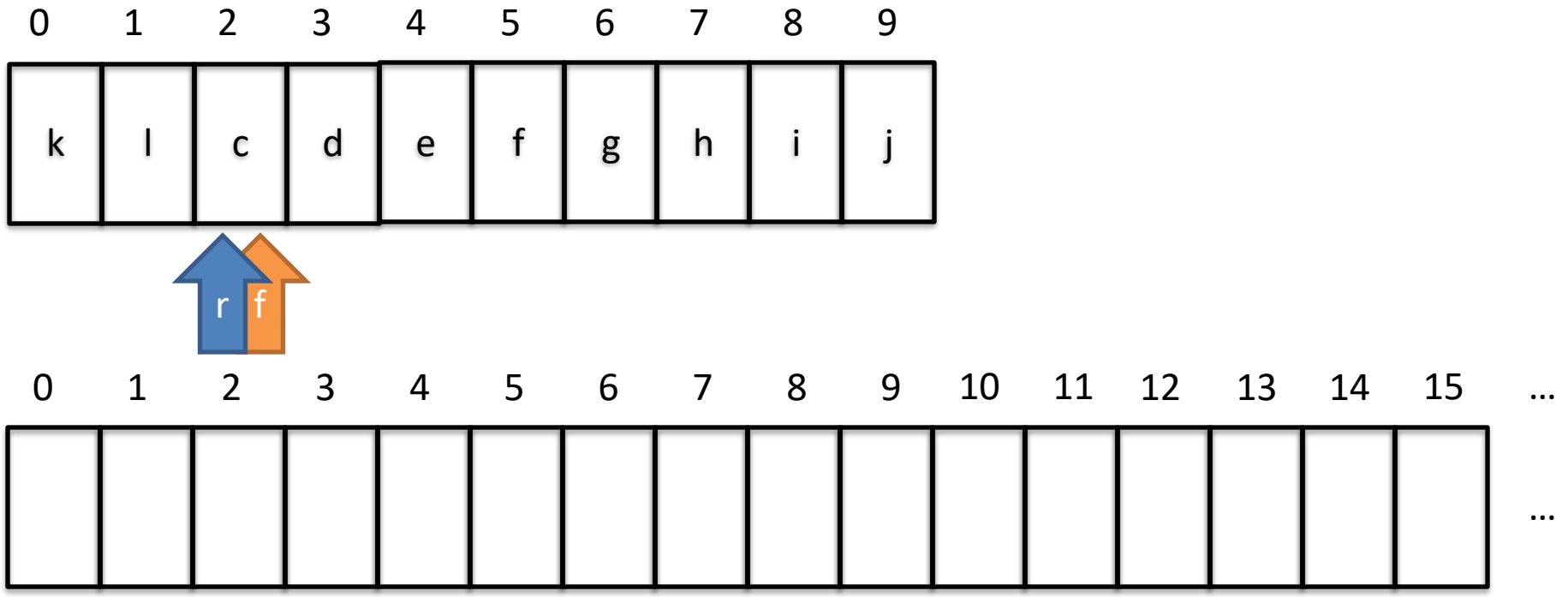
Grow by creating new larger array

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

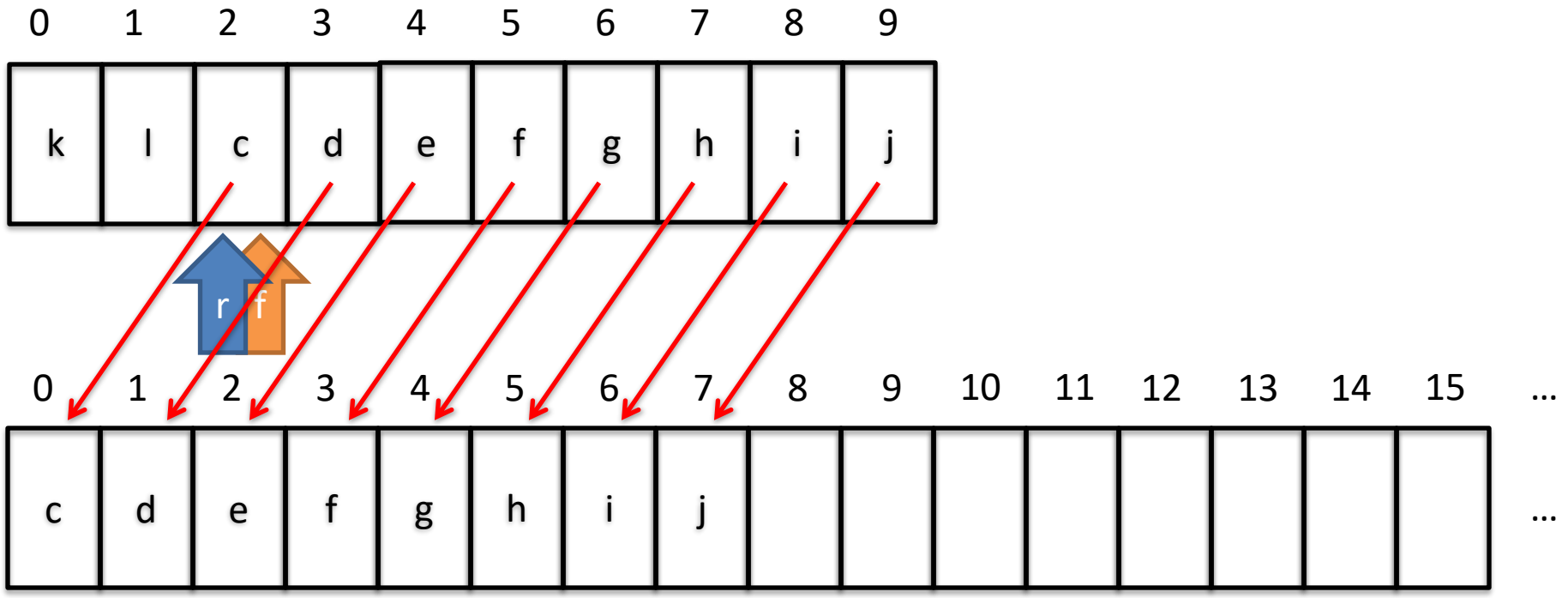
Array implementing Queue



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

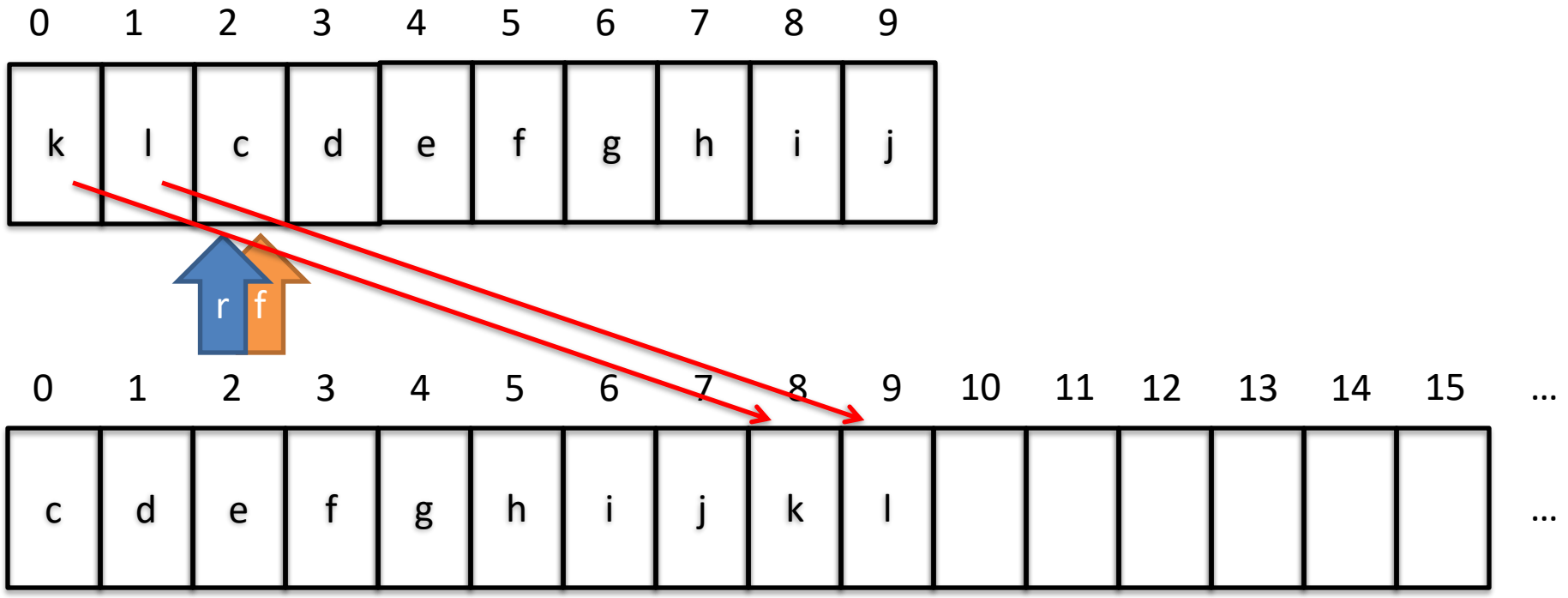
Array implementing Queue



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

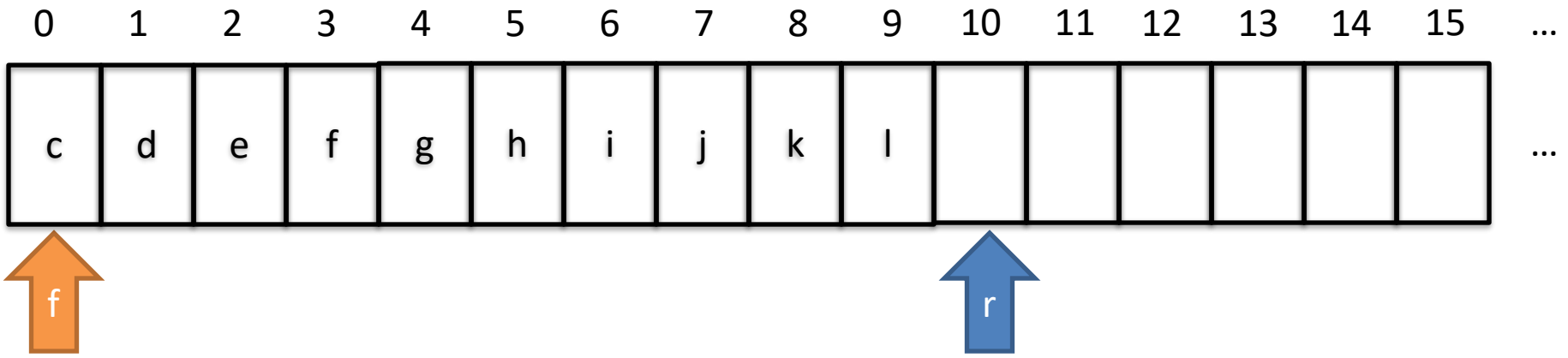


Copy old array from index 0 to *front-1* into new array starting at last index

Array implementing a Queue using index for front and rear

Array implementing Queue

size = 10



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

Set array to new array

Key points

1. Stacks are LIFO (last in, first out)
2. Use for recursive calls, parenthesis matching, reversing items in collection
3. Stacks are easy to implement with a linked list or an array
4. Queues are FIFO (first in, first out)
5. Use for modeling things like grocery checkout lines
6. Queues are easy to implement with a linked list but more difficult with an array
7. Use a circular array to implement a queue

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 ensures matching 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

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

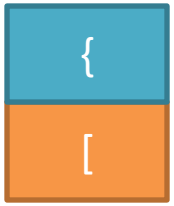
JSON String
Students: **[**
{“id”: 123, “name” : “Alice”}
{“id”: 987, “name” : “Bob”}
]

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

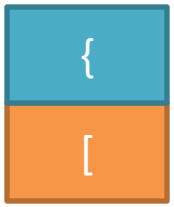
JSON String
Students: **[**
{“id”: 123, “name” : “Alice”**}**
{“id”: 987, “name” : “Bob”}
]

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

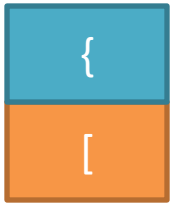
JSON String
Students: **[**
{“id”: 123, “name” : “Alice”**}**
{“id”: 987, “name” : “Bob”}
]

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

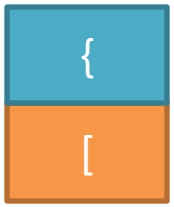
JSON String
Students: **[**
{“id”: 123, “name” : “Alice”**}**
{“id”: 987, “name” : “Bob”**}**
]

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
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

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

```
9  public class MatchParens2 {
10     public static String opens = "({[<";    // opening parens
11     public static String closes = ")}]>"";  // closing parens, in same order
12
13     /**
14      * Checks whether s is properly parenthesized and prints an appropriate p
15      */
16     public static boolean check(String s) {
17         System.out.println("checking "+s);
18         Stack<Character> parenStack = new Stack<Character>();    // all the o
19         for (int i = 0; i<s.length(); i++) {
20             // Look at each character's index in opens and closes to see if i
21             Character c = s.charAt(i);
22             if ((opens.indexOf(c)) >= 0) {
23                 parenStack.push(c);
24             }
25             else if ((closes.indexOf(c)) >= 0) {
26                 if (parenStack.isEmpty()) {
27                     System.out.println("\tunopened at position "+i);
28                     return false;
29                 }
30                 //see if matching parens
31                 if (opens.indexOf(parenStack.pop()) != closes.indexOf(c)) {
32                     System.out.println("\tmismatched at position "+i);
33                     return false;
34                 }
35             }
36         }
37
38         if (!parenStack.isEmpty()) {
39             System.out.println("\t"+parenStack.size() + " unclosed");
40             return false;
41         }
42
43         System.out.println("\tpassed");
44         return true;
45     }
46
47     public static void main(String args[]) {
48         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