Lecture 2: Stacks and Queues

CSE 373: Data Structures and Algorithms

Warm Up

1. Grab a worksheet
2. Introduce yourself to your neighbors ☺
3. Discuss the answers
4. Log onto www.socrative.com
5. Click “student login”
6. Enter “CSE373” as a room name
7. Enter your name Last, First
8. Answer question
9. Get extra credit!

List ADT tradeoffs

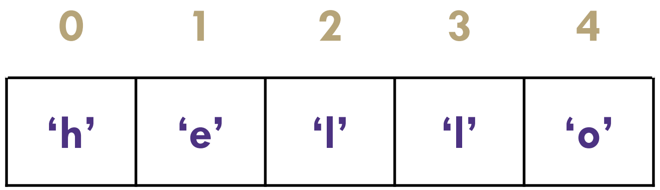
Time needed to access i-th element:
- Array: O(1) constant time
- LinkedList: O(n) linear time

Time needed to insert at i-th element
- Array: O(n) linear time
- LinkedList: O(n) linear time

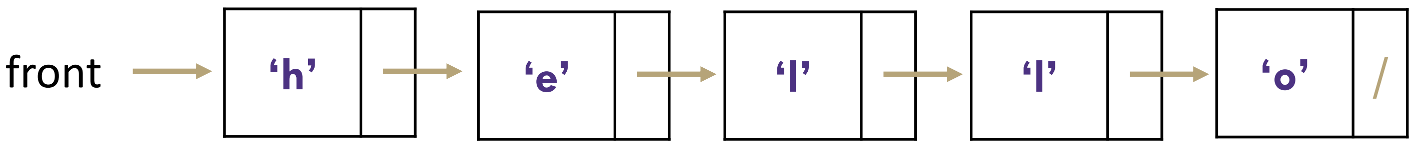
Amount of space used overall
- Array: sometimes wasted space
- LinkedList: compact

Amount of space used per element
- Array: minimal
- LinkedList: tiny extra

```
char[] myArr = new char[5]
```




```
LinkedList<Character> myLl = new LinkedList<Character>();
```



Design Decisions

Discuss with your neighbors: How would you implement the List ADT for each of the following situations? For each consider the most important functions to optimize.

Situation #1: Write a data structure that implements the List ADT that will be used to store a list of songs in a playlist.

LinkedList – optimize for growth of list and movement of songs

Situation #2: Write a data structure that implements the List ADT that will be used to store the history of a bank customer's transactions.

ArrayList – optimize for addition to back and accessing of elements

Situation #3: Write a data structure that implements the List ADT that will be used to store the order of students waiting to speak to a TA at a tutoring center

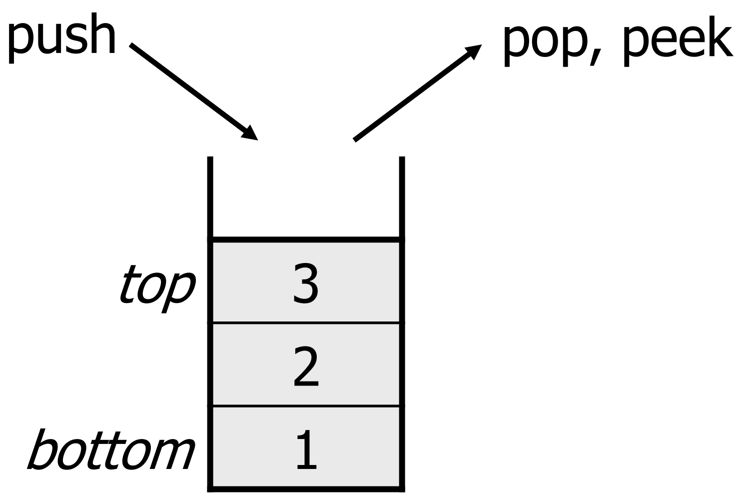
LinkedList - optimize for removal from front

ArrayList – optimize for addition to back

Review: What is a Stack?

stack: A collection based on the principle of adding elements and retrieving them in the opposite order.

- Last-In, First-Out ("LIFO")
- Elements are stored in order of insertion.
  - We do not think of them as having indexes.
- Client can only add/remove/examine the last element added (the "top").



Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

supported operations:

- **push(item)**: Add an element to the top of stack
- **pop()**: Remove the top element and returns it
- **peek()**: Examine the top element without removing it
- **size():** how many items are in the stack?
- **isEmpty():** true if there are 1 or more items in stack, false otherwise

Implementing a Stack with an Array

Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

ArrayStack<E>

state

```
data[]
size
```

behavior

```
push data[size] = value, if
out of room grow data
pop return data[size - 1],
size-1
peek return data[size - 1]
size return size
isEmpty return size == 0
```

Big O Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	O(1) Constant or worst case O(N) linear

```
push(3)
push(4)
pop()
push(5)
```

0	1	2	3
3	5		

`numberOfItems = 2`

Implementing a Stack with Nodes

Stack ADT

state

Set of ordered items
Number of items

behavior

push(item) add item to top
pop() return and remove item at top
peek() look at item at top
size() count of items
isEmpty() count of items is 0?

LinkedStack<E>

state

```
Node top
size
```

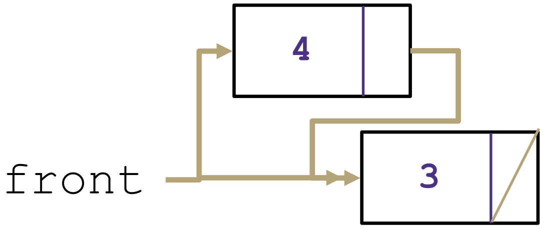
behavior

```
push add new node at top
pop return and remove node at top
peek return node at top
size return size
isEmpty return size == 0
```

Big O Analysis

`pop()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`push()`	O(1) Constant

```
push(3)
push(4)
pop()
```



`numberOfItems = 2`

Review: What is a Queue?

queue: Retrieves elements in the order they were added.

- First-In, First-Out ("FIFO")
- Elements are stored in order of insertion but don't have indexes.
- Client can only add to the end of the queue, and can only examine/remove the front of the queue.

	front		*back*	
remove, peek ←	1	2	3	← add

Queue ADT
state Set of ordered items Number of items **behavior** <u>add(item)</u> add item to back <u>remove()</u> remove and return item at front <u>peek()</u> return item at front <u>size()</u> count of items <u>isEmpty()</u> count of items is 0?

supported operations:

- **add(item):** aka "enqueue" add an element to the back.
- **remove():** aka "dequeue" Remove the front element and return.
- **peek()**: Examine the front element without removing it.
- **size():** how many items are stored in the queue?
- **isEmpty():** if 1 or more items in the queue returns true, false otherwise

Implementing a Queue with an Array

Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?

ArrayQueue<E>

```
state
  data[]
  Size
  front index
  back index
behavior
 add - data[size] = value, if
 out of room grow data
 remove - return data[size -
 1], size-1
 peek - return data[size - 1]
 size - return size
 isEmpty - return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(1) Constant or worst case O(N) linear

```
add(5)
add(8)
add(9)
remove()
```

0	1	2	3	4
5	8	9		

```
numberOfItems = 3
front = 1
back = 2
```

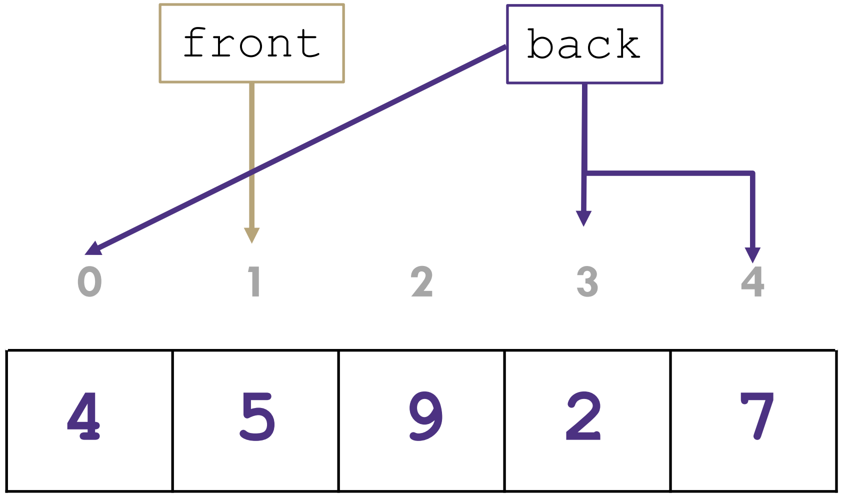
Implementing a Queue with an Array

> Wrapping Around

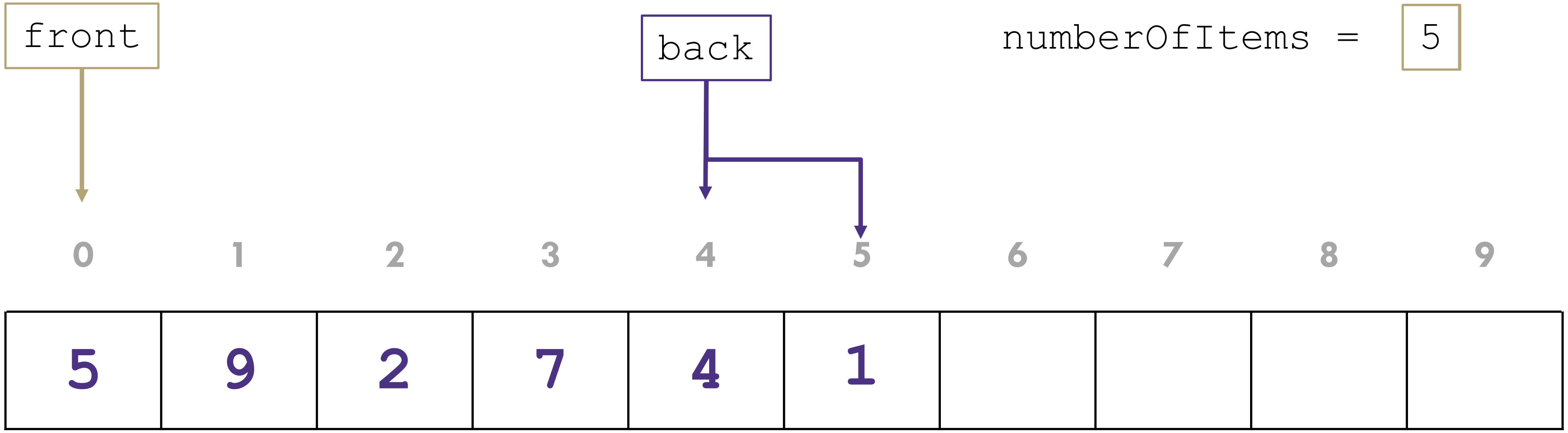
add(7)

add(4)

add(1)



numberOfItems = 5



Implementing a Queue with Nodes

Queue ADT

state

Set of ordered items
Number of items

behavior

add(item) add item to back
remove() remove and return item at front
peek() return item at front
size() count of items
isEmpty() count of items is 0?

LinkedQueue<E>

state

```
Node front
Node back
size
```

behavior

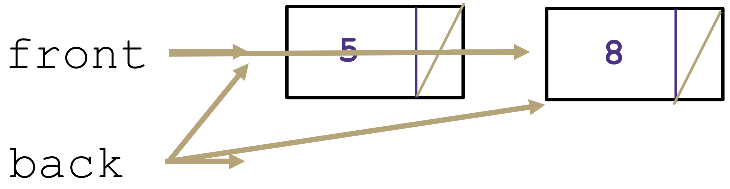
```
add - add node to back
remove - return and remove node at front
peek - return node at front
size - return size
isEmpty - return size == 0
```

Big O Analysis

`remove()`	O(1) Constant
`peek()`	O(1) Constant
`size()`	O(1) Constant
`isEmpty()`	O(1) Constant
`add()`	O(1) Constant

```
add(5)
add(8)
remove()
```

`numberOfItems = 2`



Review: Generics

```
// a parameterized (generic) class
public class name<TypeParameter> {
     ...
}
```

- Forces any client that constructs your object to supply a type
  - Don't write an actual type such as String; the client does that
  - Instead, write a type variable name such as `E` (for "element") or `T` (for "type")
  - You can require multiple type parameters separated by commas
- The rest of your class's code can refer to that type by name

```
public class Box {
   private Object object;
   public void set(Object object) {
      this.object = object;
   }
   public Object get() {
      return object;
   }
}
```

```
public class Box<T> {
   private T t;
   public void set(T t) {
      this.t = t;
   }
   public T get() {
      return t;
   }
}
```

More details: https://docs.oracle.com/javase/tutorial/java/generics/types.html

Implementing a Generic Stack