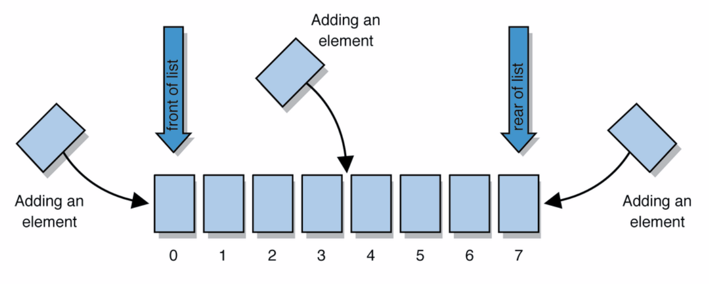
Lecture 2: Stacks and Queues

CSE 373: Data Structures and Algorithms

Warm Up

Q: Which data structure implementation of the List ADT would you choose to optimize for the “delete” function?



List ADT

state
Set of ordered items
Count of items

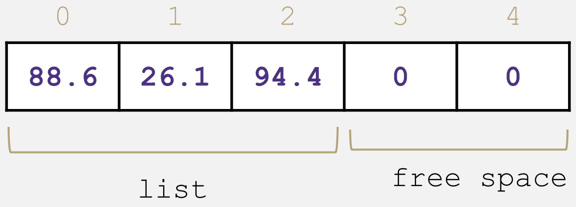
behavior
get(index) return item at index
set(item, index) replace item at index
append(item) add item to end of list
insert(item, index) add item at index
delete(index) delete item at index
size() count of items

ArrayList
uses an Array as underlying storage

ArrayList<E>

state
data[]
size

behavior
get return data[index]
set data[index] = value
append data[size] = value, if out of space grow data
insert shift values to make hole at index, data[index] = value, if out of space grow data
delete shift following values forward
size return size



LinkedList
uses nodes as underlying storage

LinkedList<E>

state
Node front
size

behavior
get loop until index, return node’s value
set loop until index, update node’s value
append create new node, update next of last node
insert create new node, loop until index, update next fields
delete loop until index, skip node
size return size

Instructions

Take 3 Minutes

1. Introduce yourself to your neighbors ☺
2. Discuss your answers
3. Log onto Poll Everywhere
   1. Go to PollEv.com/champk
   2. Text CHAMPK to 22333 to join session, text “1” or “2” to select your answer
4. Get extra credit!

Administrivia

Course Stuff

- Class webpage: cs.washington.edu/373
- Piazza: piazza.com/washington/spring2019/cse373

Homework 1 Live!

- Individual assignment
- 14x content review
- GitLab/IntelliJ setup
  - You will be created a git lab repo (TODAY)

Important Dates

- Midterm – Friday May 4th in class
- Final – Tuesday June 11th 8:30-10:20am

Homework 2 out next week, partner project

- You are responsible for finding your own partner
- Lecture, section, piazza, office hours
- Fill out partner form so we can generate repos

Review: "Big Oh"

efficiency: measure of computing resources used by code.
- can be relative to speed (time), memory (space), etc.
- most commonly refers to run time

Assume the following:
- Any single Java statement takes same amount of time to run.
- A method call's runtime is measured by the total of the statements inside the method's body.
- A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.

We measure runtime in proportion to the input data size, N.
- **growth rate**: Change in runtime as N gets bigger. How does this algorithm perform with larger and larger sets of data?

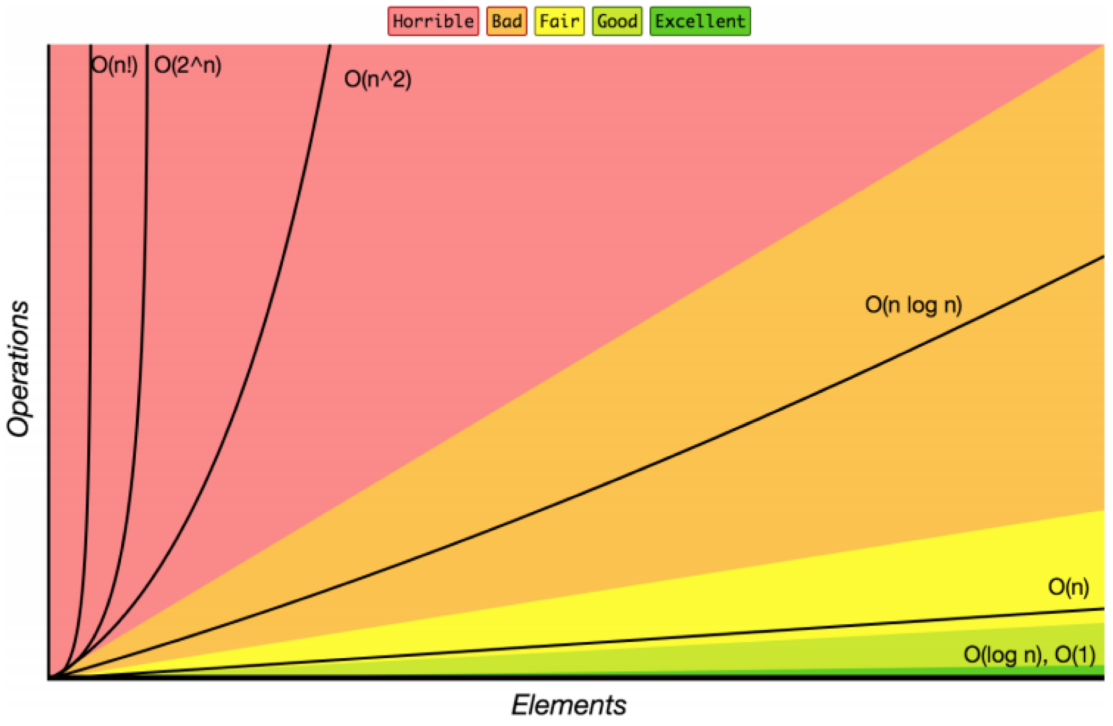
```
b = c + 10;

for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        dataTwo[j][i] = dataOne[i][j];
        dataOne[i][j] = 0;
    }
}
for (int i = 0; i < N; i++) {
    dataThree[i] = b;
}
```

This algorithm runs $\mathbf{2N^2 + N + 1}$ statements.
- We ignore constants like 2 because they are tiny next to N.
- The highest-order term (N^2) "dominates" the overall runtime.
- We say that this algorithm runs "on the order of" N^2.
- or **O(N^2)** for short ("**Big-Oh** of N squared")

Review: Complexity Class

complexity class: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Complexity Class	Big-O	Runtime if you double N	Example Algorithm
constant	O(1)	unchanged	Accessing an index of an array
logarithmic	$O(\log_2 N)$	increases slightly	Binary search
linear	O(N)	doubles	Looping over an array
log-linear	$O(N \log_2 N)$	slightly more than doubles	Merge sort algorithm
quadratic	$O(N^2)$	quadruples	Nested loops!
…	…	…	…
exponential	$O(2^N)$	multiplies drastically	Fibonacci with recursion



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

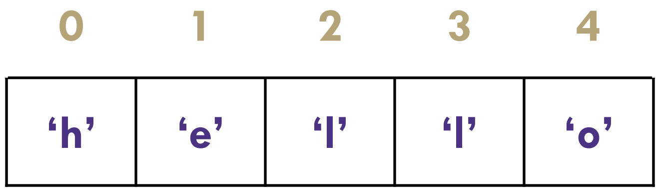
Amount of space used overall

- Array: sometimes wasted space
- LinkedList: compact

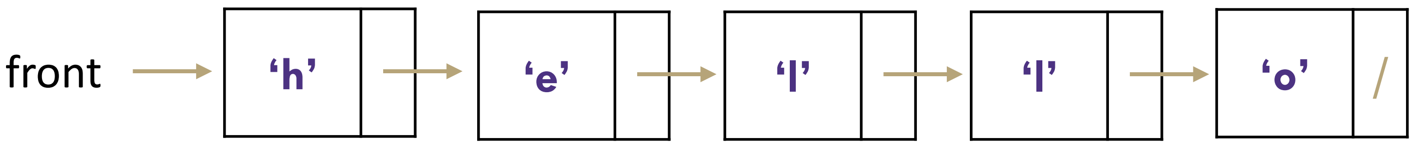
Amount of space used per element

- Array: minimal
- LinkedList: tiny extra

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




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



Design Decisions

Take 3 Minutes

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

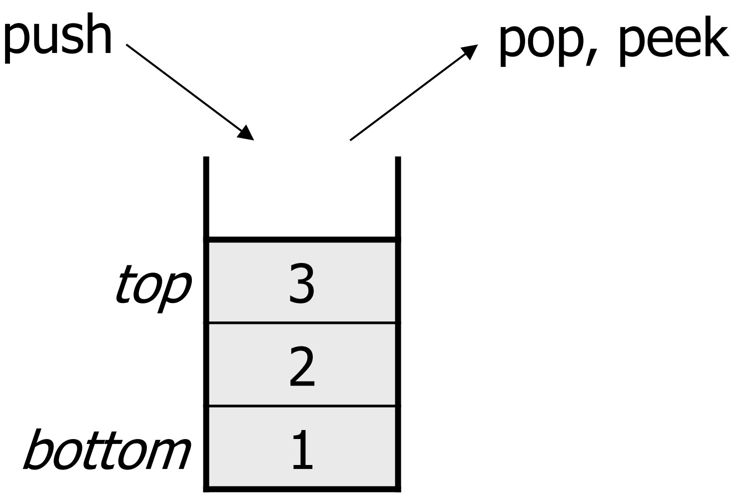
LinkedList - optimize for removal from front

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>

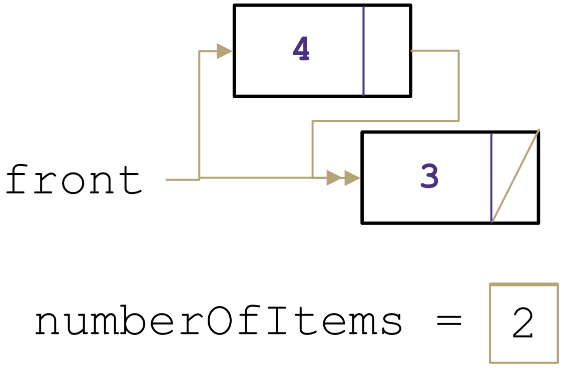
state
```
Node top
size
```

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

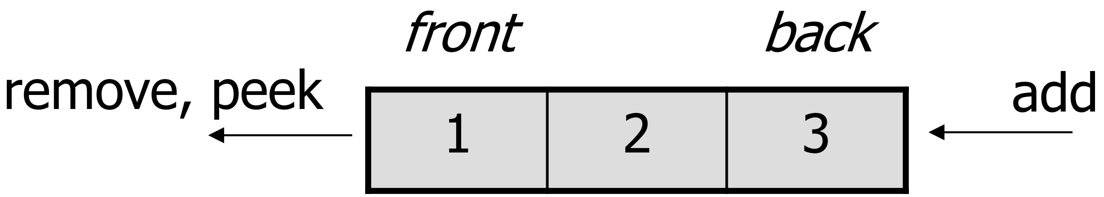
```
push(3)
push(4)
pop()
```



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.



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?

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

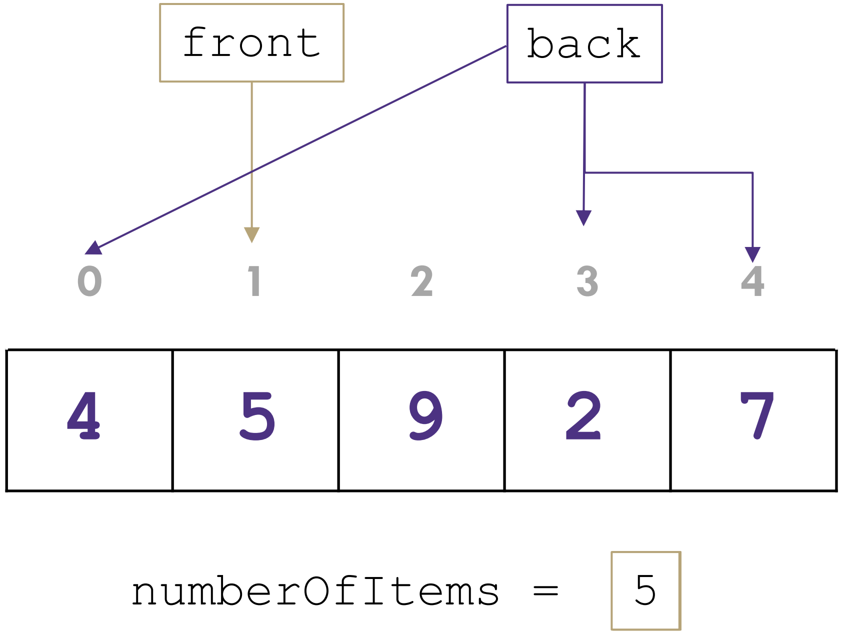
```
add(5)
add(8)
add(9)
remove()
```

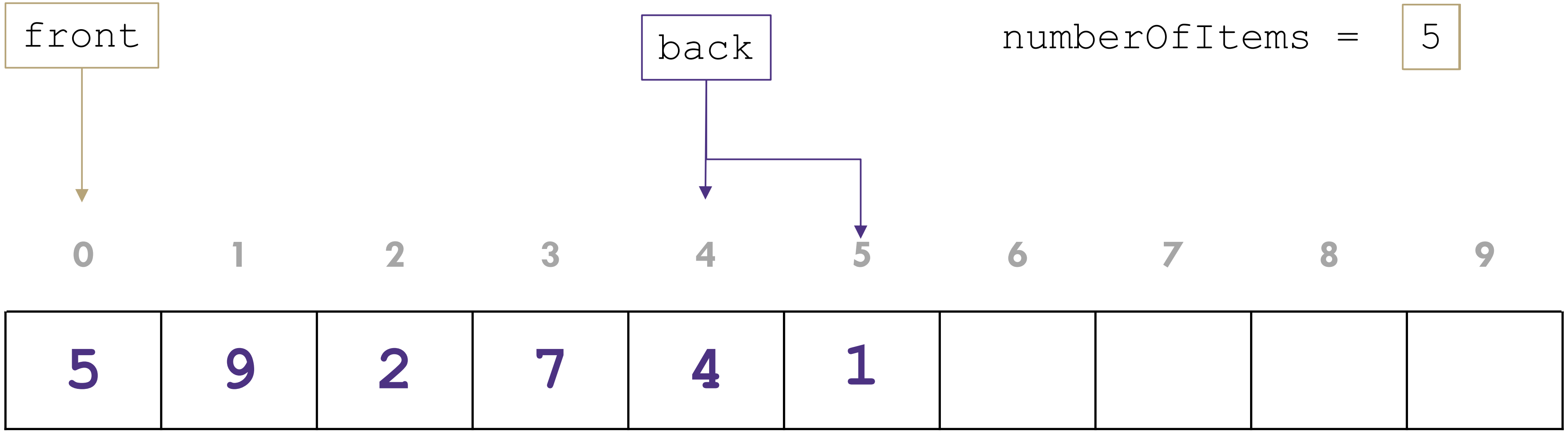
0	1	2	3	4
5	8	9		

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

Implementing a Queue with an Array
> Wrapping Around

add(7)
add(4)
add(1)





Implementing a Queue with Nodes

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?

LinkedQueue<E>

state

```
Node front
Node back
size
```

behavior

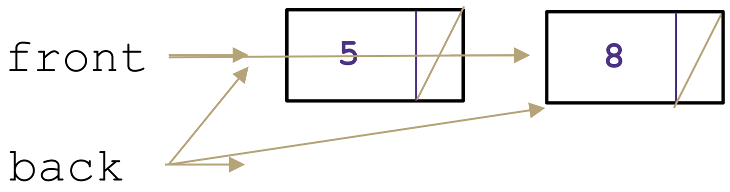
```
add - add node to back
remove - return and remove node at front
peek - return node at front
size - return size
isEmpty - return size == 0
```

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



Design Decisions

Take 3 Minutes

Discuss with your neighbors: For each scenario select the appropriate ADT and implementation to best optimize for the given scenario.

Situation #1: You are writing a program to manage a todo list with a very specific approach to tasks. This program will order tasks for someone to tackle so that the **most recent** task is addressed first. How would you store the transactions in appropriate order?

Stack – First in Last out

Nodes – make addition and removal of tasks very easy

Situation #2: You are writing a program to schedule jobs sent to a laser printer. The laser printer should process these jobs in the order in which the requests were received. How would you store the jobs in appropriate order?

Queue – First in First out

Array – want easy access to all items in queue in case you need to cancel a job

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