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]

0 1 2 3 4
'h' 'e' 'l' 'l' 'o'
```

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

```
front

'h' 'e' 'l' 'l' 'o' /
```
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 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

---

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>**

<table>
<thead>
<tr>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>data[]</td>
</tr>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

**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**

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

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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**

- push() O(1) Constant
- peek() O(1) Constant
- size() O(1) Constant
- isEmpty() O(1) Constant
- pop() 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.

---

**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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

numOfItems = 3
front = 1
back = 2
Implementing a Queue with an Array

> Wrapping Around

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

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
```

`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

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

```
front
5  8
back
```

`numberOfItems = 2`
// 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

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