Lecture 2: Stacks and Queues
Q: Which data structure implementation of the List ADT would you choose to optimize for the “delete” function?

ArrayList
uses an Array as underlying storage

ArrayList<E>

Linked List
uses nodes as underlying storage

LinkedList<E>

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(index) add item at index
delete(index) delete item at index
size() count of items

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

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

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?

```java
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 $2N^2 + N + 1$ 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 $N$.

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

```java
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 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(value) = 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
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()  

```
4  
3
```

```
front
```

```
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**

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of ordered items</td>
<td>add(item) add item to back</td>
</tr>
<tr>
<td>Number of items</td>
<td>remove() remove and return item at front</td>
</tr>
<tr>
<td></td>
<td>peek() return item at front</td>
</tr>
<tr>
<td></td>
<td>size() count of items</td>
</tr>
<tr>
<td></td>
<td>isEmpty() count of items is 0?</td>
</tr>
</tbody>
</table>

**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`

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

numberOfItems = 3
front = 1
back = 2

**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
Implementing a Queue with an Array

> Wrapping Around

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

numberOfItems = 5

front
back

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(item)` add node to back
- `remove()` remove and return 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

numberOfItems = 2

add(5)
add(8)
remove()
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

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