Lecture 3: Stacks, Queues, and Dictionaries
Warm Up

Q: Would you use a LinkedList or ArrayList implementation for each of these scenarios?

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

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

**Situation #3**: Choose 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.

Instructions

1. [www.pollev.com/cse373activity](http://www.pollev.com/cse373activity) for participating in our active learning questions. For this particular question label your answer with
   - what situation #
   - ArrayList/LinkedList
   - why.

2. [https://www.pollev.com/cse373studentqs](https://www.pollev.com/cse373studentqs) to ask your own questions.
Design Decisions

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

*ArrayList – I want to be able to shuffle play on the playlist*

**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*
List ADT tradeoffs

Last time: we used “slow” and “fast” to describe running times. Let’s be a little more precise.

Recall these basic Big-O ideas from 14X: Suppose our list has N elements
- If a method takes a constant number of steps (like 23 or 5) its running time is $O(1)$
- If a method takes a linear number of steps (like $4N+3$) its running time is $O(N)$

For ArrayLists and LinkedLists, what is the $O()$ for each of these operations?
- Time needed to access $N^{th}$ element:
- Time needed to insert at end (the array is full!)

What are the memory tradeoffs for our two implementations?
- Amount of space used overall
- Amount of space used per element
List ADT tradeoffs

Time needed to access $N$th element:
- *ArrayList*: O(1) constant time
- *LinkedList*: O(N) linear time

Time needed to insert at $N$th element (the array is full!)
- *ArrayList*: O(N) linear time
- *LinkedList*: O(N) linear time

Amount of space used overall
- *ArrayList*: sometimes wasted space
- *LinkedList*: compact

Amount of space used per element
- *ArrayList*: minimal
- *LinkedList*: tiny extra
Administrivia

- reminder: P0 released on Wednesday, due next Wednesday. Lots of good setup questions on Piazza, continue to use that resource

- 4/1 Lecture is a panopto recording because we had to make some edits -- sorry for the inconvenience. If it's not published immediately in the future, it'll also be on Panopto and a little bit delayed.

- Section -- video/recording is being worked on and edited by our TAs right now, should be up soon. Handouts/Solutions/Slides always posted on the website -- the solutions/slides are up now. We'll make a Piazza announcement when the video is up.

- Student Slack -- Totally optional chatroom for y'all to join to build a community for this class (beyond piazza which is kinda rigid). You can use it to find a partner/coordinate with them, plan study sessions, or just chat with people in general! We won’t be monitoring this closely, but please be kind and respectful to others. Feel free to report any incidents and we can look into it. Everyone registered should have received an invite.

- Office Hours today, see the calendar on course website for schedule and the Zoom tab on Canvas for meeting link
Questions?
**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

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of ordered items</td>
<td>push(item) add item to top</td>
</tr>
<tr>
<td>Number of items</td>
<td>pop() return and remove item at top</td>
</tr>
<tr>
<td></td>
<td>peek() look at item at top</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:**
- **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

<table>
<thead>
<tr>
<th>Method</th>
<th>Big O</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop()</td>
<td>O(1)</td>
</tr>
<tr>
<td>peek()</td>
<td>O(1)</td>
</tr>
<tr>
<td>size()</td>
<td>O(1)</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1)</td>
</tr>
<tr>
<td>push()</td>
<td>O(N) linear if you have to resize O(1) otherwise</td>
</tr>
</tbody>
</table>

Take 1 min to respond to activity

www.pollev.com/cse373activity
What do you think the worst possible runtime of the “push()” operation will be?

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` number of items

- **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

---

Take 1 min to respond to activity

www.pollev.com/cse373activity

What do you think the worst possible runtime of the “push()” operation will be?
Question Break
**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>
</tr>
</thead>
<tbody>
<tr>
<td>Set of ordered items</td>
</tr>
<tr>
<td>Number of items</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(item)</td>
</tr>
<tr>
<td>remove()</td>
</tr>
<tr>
<td>peek()</td>
</tr>
<tr>
<td>size()</td>
</tr>
<tr>
<td>isEmpty()</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

Big O Analysis

remove() O(1) Constant
peek() O(1) Constant
size() O(1) Constant
isEmpty() O(1) Constant
add() O(N) linear if you have to resize
O(1) otherwise

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

0 1 2 3 4
5 8 9 0

numberOfItems = 3
front = 1
back = 2

www.pollev.com/cse373activity
What do you think the worst possible runtime of the “add()” operation will be?
Implementing a Queue with an Array

> Wrapping Around

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

class Queue:
    def __init__(self, capacity):
        self.front = None
        self.back = None
        self.number_of_items = 0
        self.items = [None] * capacity

    def add(self, item):
        if self.number_of_items == len(self.items):
            return
        self.back = (self.back + 1) % len(self.items)
        self.items[self.back] = item
        self.number_of_items += 1

    def remove(self):
        if self.number_of_items == 0:
            return None
        self.front = (self.front + 1) % len(self.items)
        self.number_of_items -= 1
        return self.items[self.front]
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<br/> `<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()
```

```
front
5 8
back
```

**Take 1 min to respond to activity**

What do you think the worst case runtime of the “add()” operation will be?

www.pollev.com/cse373activity
Questions?
Design Decisions

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

Situation: 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. There are busy and slow times for requests that can have large differences in the volume of jobs sent to the printer. Which ADT and what implementation would you use to store the jobs sent to the printer?

ADT options:
- List
- Stack
- Queue

Implementation options:
- array
- linked nodes
Breakout Instructions

1. Instructor will trigger breakout rooms
2. Accept the invite that pops up
3. Work with your partners to answer the question on slide 16
4. TAs will be coming in and out. Fill out this form to request a TA’s assistance: https://forms.gle/b9NiC1s11FKBcpm89
5. Instructor will end the breakouts in 5 minutes

For detailed instructions on how breakouts work: https://docs.google.com/presentation/d/15HiAPu6yYz2WWBkonRejBtUcq_FFhmoW FYT2l25G06o/edit#slide=id.g8289eae46a_0_694
Discuss in your Breakouts: For each scenario select the appropriate ADT and implementation to best optimize for the given scenario.

Situation: 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. There are busy and slow times for requests that can have large differences in the volume of jobs sent to the printer. Which ADT and what implementation would you use to store the jobs sent to the printer?

ADT options:
- List
- Stack
- Queue

Implementation options:
- array
- linked nodes
Dictionaries
Dictionaries (aka Maps)

Every Programmer’s Best Friend

You’ll probably use one in almost every programming project.
- Because it’s hard to make a big project without needing one sooner or later.

// two types of Map implementations supposedly covered in CSE 143
Map<String, Integer> map1 = new HashMap<>();
Map<String, String> map2 = new TreeMap<>();
**Review: Maps**

**map**: Holds a set of distinct *keys* and a collection of *values*, where each key is associated with one value.
- a.k.a. "dictionary"

**supported operations:**
- **put**(*key*, *value*): Adds a given item into collection with associated key,
  - if the map previously had a mapping for the given key, old value is replaced.
- **get**(*key*): Retrieves the value mapped to the key
- **containsKey**(*key*): returns true if key is already associated with value in map, false otherwise
- **remove**(*key*): Removes the given key and its mapped value

### Dictionary ADT

**state**
- Set of items & keys
- Count of items

**behavior**
- put(*key*, *item*): add item to collection indexed with key
- get(*key*): return item associated with key
- containsKey(*key*): return if key already in use
- remove(*key*): remove item and associated key
- size(): return count of items
Implementing a Dictionary with an Array

**ArrayDictionary< K, V >**

**state**
Pair<K, V>[] data

**behavior**
- **put** find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
- **get** scan all pairs looking for given key, return associated item if found
- **containsKey** scan all pairs, return if key is found
- **remove** scan all pairs, replace pair to be removed with last pair in collection
- **size** return count of items in collection

```
containsKey('c')
get('d')
put('b', 97)
put('e', 20)
```

**Big O Analysis**
- **put()** O(N) linear
- **get()** O(N) linear
- **containsKey()** O(N) linear
- **remove()** O(N) linear
- **size()** O(1) constant

**Big O Analysis** (if key is the first one looked at)
- **put()** O(1) constant
- **get()** O(1) constant
- **containsKey()** O(1) constant
- **remove()** O(1) constant
- **size()** O(1) constant
Implementing a Dictionary with Nodes

**Dictionary ADT**

**state**
Set of items & keys  
Count of items

**behavior**
put(key, item) add item to collection indexed with key  
get(key) return item associated with key  
containsKey(key) return if key already in use  
remove(key) remove item and associated key  
size() return count of items

**containsKey(‘c’)  
get(‘d’)  
put(‘b’, 20)**

**LinkedDictionary<K, V>**

**state**
front  
size

**behavior**
put if key is unused, create new with pair, add to front of list, else replace with new value  
get scan all pairs looking for given key, return associated item if found  
containsKey scan all pairs, return if key is found  
remove scan all pairs, skip pair to be removed  
size return count of items in dictionary

Big O Analysis – (if key is the last one looked at / not in the dictionary)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Big O</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>O(N) linear</td>
</tr>
<tr>
<td>get()</td>
<td>O(N) linear</td>
</tr>
<tr>
<td>containsKey()</td>
<td>O(N) linear</td>
</tr>
<tr>
<td>remove()</td>
<td>O(N) linear</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) constant</td>
</tr>
</tbody>
</table>

Big O Analysis – (if the key is the first one looked at)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Big O</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>get()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>containsKey()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>remove()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) constant</td>
</tr>
</tbody>
</table>