Lecture 3: Stacks, Queues, and Dictionaries

Please fill out the Poll at pollev.com/21sp373
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.

Please fill out the Poll at pollev.com/21sp373
Announcements

Course website live with slides

HW 0 – 143 Review Project
- Live on website
- Due Wednesday April 7th

Find a partner by next Wednesday
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<sup>th</sup> 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

ArrayList<Character> myArr

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘h’</td>
<td>‘e’</td>
<td>‘l’</td>
<td>‘l’</td>
<td>‘o’</td>
</tr>
</tbody>
</table>

LinkedList<Character> myLl

front → ‘h’ → ‘e’ → ‘l’ → ‘l’ → ‘o’ /
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
**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(log₂ 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 log₂ N)</td>
<td>slightly more than doubles</td>
<td>Merge sort algorithm</td>
</tr>
<tr>
<td>quadratic</td>
<td>O(N²)</td>
<td>quadruples</td>
<td>Nested loops!</td>
</tr>
<tr>
<td>exponential</td>
<td>O(2^N)</td>
<td>multiplies drastically</td>
<td>Fibonacci with recursion</td>
</tr>
</tbody>
</table>

Note: You don’t have to understand all of this right now – we’ll dive into it soon.
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

- `pop()` O(1) Constant
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `push()` O(N) linear if you have to resize O(1) otherwise

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?
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?

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

**Big O Analysis**
- `pop()` \(O(1)\) Constant
- `peek()` \(O(1)\) Constant
- `size()` \(O(1)\) Constant
- `isEmpty()` \(O(1)\) Constant
- `push()` \(O(1)\) Constant

```
numberOfItems = 2
```

Take 1 min to respond to activity

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

www.pollev.com/cse373activity
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>
<th>behavior</th>
<th>supported operations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of ordered items</td>
<td>add(item): add item to back</td>
<td>- add(item): aka “enqueue” add an element to the back.</td>
</tr>
<tr>
<td>Number of items</td>
<td>remove(): remove and return item at front</td>
<td>- remove(): aka “dequeue” Remove the front element and return.</td>
</tr>
<tr>
<td></td>
<td>peek(): return item at front</td>
<td>- peek(): Examine the front element without removing it.</td>
</tr>
<tr>
<td></td>
<td>size(): count of items</td>
<td>- size(): how many items are stored in the queue?</td>
</tr>
<tr>
<td></td>
<td>isEmpty(): count of items is 0?</td>
<td>- isEmpty(): if 1 or more items in the queue returns true, false otherwise</td>
</tr>
</tbody>
</table>
Implementing a Queue with an Array

Que e 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

Take 1 min to respond to activity

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)

number of items = 5

front
back

0 1 2 3 4 5 6 7 8 9

5 9 2 7 4 1

CSE 373 SP 18 - KASEY CHAMPION 15
Implementing a Queue with Nodes

Queues are used to represent a single linked list (where the queue is empty if the head is null)

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

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

```
front -> 5 -> 8 -> back
```

**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
```

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

www.pollev.com/cse373activity
Questions?
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_FFhmoWFyT2I25G06o/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"

**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

**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
Implementing a Dictionary with an Array

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

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 dictionary

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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>('a', 1)</td>
<td>('b', 97)</td>
<td>('c', 3)</td>
<td>('d', 4)</td>
<td>('e', 20)</td>
</tr>
</tbody>
</table>

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

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

Big O Analysis – (if the 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**

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of items &amp; keys</td>
<td>put(key, item) add item to collection indexed with key</td>
</tr>
<tr>
<td>Count of items</td>
<td>get(key) return item associated with key</td>
</tr>
<tr>
<td></td>
<td>containsKey(key) return if key already in use</td>
</tr>
<tr>
<td></td>
<td>remove(key) remove item and associated key</td>
</tr>
<tr>
<td></td>
<td>size() return count of items</td>
</tr>
</tbody>
</table>

**LinkedDictionary<K, V>**

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
<td>put if key is unused, create new with pair, add to front of list, else replace with new value</td>
</tr>
<tr>
<td>size</td>
<td>get scan all pairs looking for given key, return associated item if found</td>
</tr>
<tr>
<td></td>
<td>containsKey scan all pairs, return if key is found</td>
</tr>
<tr>
<td></td>
<td>remove scan all pairs, skip pair to be removed</td>
</tr>
<tr>
<td></td>
<td>size() return count of items in dictionary</td>
</tr>
</tbody>
</table>

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

- put() \(\text{O(N) linear}\)
- get() \(\text{O(N) linear}\)
- containsKey() \(\text{O(N) linear}\)
- remove() \(\text{O(N) linear}\)
- size() \(\text{O(1) constant}\)

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

- put() \(\text{O(1) constant}\)
- get() \(\text{O(1) constant}\)
- containsKey() \(\text{O(1) constant}\)
- remove() \(\text{O(1) constant}\)
- size() \(\text{O(1) constant}\)