Lecture 3: ADT Implementation

CSE 373: Data Structures and Algorithms
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.
Warm Up

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

**Features to consider:**
- add or remove songs from list
- change song order
- shuffle play

**Why ArrayList?**
- optimized element access makes shuffle more efficient
- accessing next element faster in contiguous memory

**Why LinkedList?**
- easier to reorder songs
- memory right sized for changes in size of playlist, shrinks if songs are removed

**Q:** Would you use a LinkedList or ArrayList implementation for this scenario? *Discuss with those around you!*

Slido Event #2048464
https://app.sli.do/event/hnnjzEubx73ASDfQCzhPXw
Agenda

Design Decisions Review
Stacks
Queues
Dictionaries
Questions
Announcements

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

Monday – Exercise 1 to be released
Design Decisions Review

Stacks
Queues
Dictionaries
Questions
Design Decisions

For every ADT there are lots of different ways to implement them.

Based on your situation you should consider:

- Memory vs Speed
- Generic/Reusability vs Specific/Specialized
- One Function vs Another
- Robustness vs Performance

This class is all about implementing ADTs based on making the right design tradeoffs!

A common topic in interview questions!
A quick aside: Types of memory

Arrays - contiguous memory: when the “new” keyword is used on an array the operating system sets aside a single, right-sized block of computer memory.

```java
int[] array = new int[3];
array[0] = 3;
array[1] = 7;
array[2] = 3;
```

Nodes - non-contiguous memory: when the “new” keyword is used on a single node the operating system sets aside enough space for that object at the next available memory location.

```java
Node front = new Node(3);
front.next = new Node(7);
front.next.next = new Node(3);
```
Design Decisions

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

**Features to consider:**
- adding a new transaction
- reviewing/retrieving transaction history

**Why ArrayList?**
- optimized element access makes reviewing based on order easier
- contiguous memory more efficient and less waste than usual array usage because no removals

**Why LinkedList?**
- if structured with front pointing to most recent transaction, addition of transactions constant time
- memory right sized for large variations in different account history size

Q: Would you use a LinkedList or ArrayList implementation for this scenario? Discuss with those around you!
Real-World Scenarios: Lists

**LinkedList**
- Image viewer
  - Previous and next images are linked, hence can be accessed by next and previous button
- Dynamic memory allocation
  - We use linked list of free blocks
- Implementations of other ADTs such as Stacks, Queues, Graphs, etc.

**ArrayList**
- Maintaining Database Records
  - List of records you want to add / delete from and maintain your order after
- Implementations of other ADTs such as Stacks, Queues, Graphs, etc.
Questions?
Design Decisions Review

Stacks

Queues

Dictionaries

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`:
  - Set `data[size] = value`, if out of room grow data
- `pop`:
  - Return and remove item at top
- `peek`:
  - Return item at top
- `size`:
  - Return size
- `isEmpty`:
  - Return size == 0

---

**Big O Analysis**

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pop()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>peek()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>O(1) Constant</td>
</tr>
</tbody>
</table>
| `push()`  | O(N) Linear if a resize is required  
|           | O(1) Otherwise  |

---

```
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**

<table>
<thead>
<tr>
<th>Method</th>
<th>Big O</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pop()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>peek()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>size()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>isEmpty()</code></td>
<td>O(1) Constant</td>
</tr>
<tr>
<td><code>push()</code></td>
<td>O(1) Constant</td>
</tr>
</tbody>
</table>

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

```
4
front

numberOfItems = 2
```
Real-World Scenarios – Stacks

- **Undo/Redo Feature in editing software**
  - push for every action
  - pop for every click of “undo”

- **Matching tags/curl braces**
  - push at every opening
  - pop at every closing, check if there’s a match

- **DNA Parsing Implementation**
  - stack is able to record combinations of two different DNA signals, release the signals into solution in reverse order, and then re-record
  - social implications + ethical concerns?
    - performance of stack dependent on efficiency of “washing steps” between stack operations
    - what if certain DNA needs more stack operations to parse than other? what kind of inequalities can this create between more common and more rare DNA? what are some social consequences of using a stack for DNA sequencing?
Design Decisions Review

Stacks

Queues

Dictionaries

Questions
**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

Big O Analysis

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>peek()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>add()</td>
<td>O(N) Linear if a resize is required O(1) Otherwise</td>
</tr>
</tbody>
</table>

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 (Wrap around)

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

front
back

numberOfItems = 6
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Big O Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>peek()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>O(1) Constant</td>
</tr>
<tr>
<td>add()</td>
<td>O(1) Constant</td>
</tr>
</tbody>
</table>

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

numberOfItems = 2

![Queue diagram]
Real-World Examples

Serving requests on a single shared resource
- e.g. a printer, CPU task scheduling, etc.

Call Center phone systems use Queues to hold people calling them in order, until a service representative is free.

Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, i.e. first come first served.
Questions?
Design Decisions Review
Stacks
Queues
Dictionaries
Questions
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 123
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)**: Adds a given item into collection indexed with 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

### 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

```java
map.get("the")
```
Dictionary ADT

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

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

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

---

Implementing a Dictionary with an Array

<table>
<thead>
<tr>
<th>ArrayDictionary&lt;K, V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>state</strong></td>
</tr>
<tr>
<td>Pair&lt;K, V&gt;[] data</td>
</tr>
<tr>
<td><strong>behavior</strong></td>
</tr>
<tr>
<td>put</td>
</tr>
<tr>
<td>get</td>
</tr>
<tr>
<td>containsKey</td>
</tr>
<tr>
<td>remove</td>
</tr>
<tr>
<td>size</td>
</tr>
</tbody>
</table>

---

Big O Analysis: if the key is the last one looked at / is not in the dictionary

<table>
<thead>
<tr>
<th>Method</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>Method</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>
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 the key is the last one looked at / is not in the dictionary

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</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>Method</th>
<th>Complexity</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>
Real-World Examples

- Symbol table for compilers
  - Key = symbol, Value = command meaning

- Database indexing
  - Data stored in databases is generally of the key-value format which is typically implemented using a HashTable data structure Dictionary.

- Computer File Managing
  - Each file has two very crucial information that is, filename and file path, in order to make a connection between the filename to its corresponding file path hash tables are used
Design Decisions

**Discuss with your neighbors:** For the following scenario select the appropriate ADT and implementation and explain why they are optimal for this situation.

**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?

<table>
<thead>
<tr>
<th>ADT options:</th>
<th>Implementation options:</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>array</td>
</tr>
<tr>
<td>Stack</td>
<td>linked nodes</td>
</tr>
<tr>
<td>Queue</td>
<td></td>
</tr>
</tbody>
</table>

**Kasey’s Answer**

**LinkedQueue**

This will maintain the original order of the print jobs, but allow you to easily cancel jobs stuck in the middle of the queue. This will also keep the space used by the queue at the minimum required levels despite the fact the queue will have very different lengths at different times.
Design Decisions Review
Stacks
Queues
Dictionaries
Questions?
That’s all!

Have a great weekend!