Lecture 3: Design Tradeoffs

Please log onto https://edstem.org/us/courses/21257/discussion/1336583 to submit live lecture questions

Please log onto PollEv.com/champk to answer the daily lecture participation question
Warm Up

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

**ArrayList** uses an Array as underlying storage

```
state
  data[]
  size
behavior
  get return data[index]
  set data[index] = value
  add 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
```

**LinkedList** uses nodes as underlying storage

```
state
  Node front
  size
behavior
  get loop until index, return node’s value
  set loop until index, update node’s value
  add 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
```

**List ADT**

- **get(index)** return item at index
- **set(item, index)** replace item at index
- **append(item)** add item to end of list
- **insert(item, index)** add item at index
- **delete(index)** delete item at index
- **size()** count of items

---

**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/champk
Announcements

Course website live with slides

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

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

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:
  - 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

What are the memory tradeoffs for our two implementations?
- Amount of space used overall
  - ArrayList: sometimes wasted space
  - LinkedList: compact
- Amount of space used per element
  - ArrayList: minimal
  - LinkedList: tiny extra
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
- Music Player – Songs in music player are linked to previous and next song, you can play songs either from starting or ending of the list
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** – optimize for the ability to shuffle play on the playlist
- **LinkedList** – optimize for adding/removing/changing order 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 accessing of elements and adding to back
- **LinkedList** – if structured backwards, could optimize for addition to front

**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
Questions?
Two more ADTs: Stacks and Queues
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(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?

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<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
- pop() O(1) Constant
- peek() O(1) Constant
- size() O(1) Constant
- isEmpty() O(1) Constant
- push() O(1) Constant

push(3)
push(4)

front

numberOfItems = 2

pop()

Stack ADT

push(3)
push(4)
pop()
Real-World Scenarios – Stacks

- Undo/Redo Feature in editing software
- DNA Parsing Implementation
  - [https://www.nature.com/articles/s41467-021-25023-6](https://www.nature.com/articles/s41467-021-25023-6)
  - 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?
    - accuracy limits
    - 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?
**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(N) linear if you have to resize
  O(1) otherwise

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

```
<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`
Implementing a Queue with an Array

> Wrapping Around

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

```plaintext
numberOfItems = 5
front
back
```

```
add(4)
add(1)
```

```plaintext
front
back
```

```
4 5 9 2 7
0 1 2 3 4
```

```
5 9 2 7 4 1
0 1 2 3 4
```
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

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

**Big O Analysis**
- `remove()` O(1) Constant
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `add()` O(1) Constant

**Take 1 min to respond to activity**

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

ADT options:  Implementation options:
- List        - array
- Stack       - linked nodes
- Queue

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.
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 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)`: 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

**supported operations:**

- `map.get("the")`: 56
  - `map.get("you")`: 22
  - `map.get("in")`: 37

**Table:**

<table>
<thead>
<tr>
<th>KEYS</th>
<th>VALUES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jan</td>
<td>327.2</td>
</tr>
<tr>
<td>Feb</td>
<td>368.2</td>
</tr>
<tr>
<td>Mar</td>
<td>197.6</td>
</tr>
<tr>
<td>Apr</td>
<td>178.4</td>
</tr>
<tr>
<td>May</td>
<td>100.0</td>
</tr>
<tr>
<td>Jun</td>
<td>69.9</td>
</tr>
<tr>
<td>Jul</td>
<td>32.3</td>
</tr>
<tr>
<td>Aug</td>
<td>37.3</td>
</tr>
<tr>
<td>Sep</td>
<td>19.0</td>
</tr>
<tr>
<td>Oct</td>
<td>37.0</td>
</tr>
<tr>
<td>Nov</td>
<td>73.2</td>
</tr>
<tr>
<td>Dec</td>
<td>110.9</td>
</tr>
<tr>
<td>Annual</td>
<td>1551.0</td>
</tr>
</tbody>
</table>
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)

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Big O Analysis</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 Analysis</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

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 with new value
- `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

**Example**
- `containsKey('c')`
- `get('d')`
- `put('b', 20)`

**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
Real-World Examples

- Used for fast data lookups & hashing
- Symbol table for compilers
- Database indexing
  - Data stored in databases is generally of the key-value format which is done through hash tables.
- Caches
- Unique data representation
- Every time we type something to be searched in google chrome or other browsers, it generates the desired output based on the principle of hashing
- Message Digest
  - a function of cryptography also uses hashing for creating output in such a manner that reaching to the original input from that generated output is almost next to impossible
- 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
Questions?