CS234
MIDTERM REVIEW

• What’s the format?
• What’s on the midterm?

Updated: 2018-06-14
When and Where?

- Tuesday, June 19\textsuperscript{th}, 2018, 4:30P – 6:20P

- Seats are assigned through Odyssey
Coverage

Modules 1-9:
- Abstraction and ADT
- Arrays and Dynamic Arrays (Python Lists)
- Algorithm Analysis / Big-O Notation
- Set and Map ADTs
- Linked Lists (singly and doubly)
- Searching and Sorting
- Stack and Queue ADTs
- Hash Tables
Format

Similar to the assignments:

i.e. Some programming, some written.

Since time is limited, the written will be more focused on conceptual things and less on the more puzzle-like “Find an algorithm to do _” (which can take a while to wrap your head around)
How to Study

Course Slides are a good start

What I wrote on the board and/or said about the slides is good to know

The examples I post are sometimes more complete than what I did in class

You could try re-implementing some ADTs from the examples, or implementing ones I didn’t.
How to Study

Feel free to ask during office hours:

• If you’re just seeing this on Tuesday, it’s too late!

• Special Monday Exam Hours: 12:30 – 2:30
• Tuesdays 12:30 – 2:00
OK, Crash Course Time. Module 1

Learn Python. If you just ate the 0s on A1 + A2 you’re going to get hit hard on the Midterm, too.
Module 2, Abstraction

- Separating the properties of the object, so we can focus on the relevant properties and ignore the rest
- For ADTs: Focus on what what rather than the how
- Abstraction lets us focus on solving the problem
Module 2, ADTs

• ADTs:
  • Let the User Focus on their problem, not on ours (the implementation)
  • Let Us change the implementation without the User needing to know (Flexibility)
  • Let Us avoid logical errors caused by misuse of data (e.g. if the user cannot access the nodes of our linked list, they cannot leave it in an inconsistent state)
Module 3, Arrays

An Array is a sequence of sequential “buckets”, with index 0 through N – 1 (N being the length)

Usually all the same type (homogeneous)
• In Python this is arguably true, “Python Object” is the type. An Int and a Str are both subtypes of Python Object.

• Has a Fixed Length

• A Python List is an Array, but does not have fixed length
Module 3, Dynamic Arrays

• A Python List uses fixed length arrays. If the array is full then a new one is created with twice the length.
  • This gives $O(1)$ amortized complexity for list.append

• This is called a dynamic array.
• The strategy of doubling has the creative name: array doubling
Module 3, 2D Arrays

• A 2D array could be an Array of Arrays
• It could also be flattened using row-major form. Where you store Row 0, then Row 1, then Row 2, …

  • Array2D[row,col] is stored at Array1D[row * column_count + col]

  • Only works if each row has the same column count (e.g. a matrix)
Module 4, Sets and Maps

This was a light one, wasn’t it?

A Set contains **unique** values. Usually mathematical, so it provides union, intersection, subtract, symmetric difference.

A Map contains **unique** keys and maps them to values (not necessarily unique).

- Python’s dict type is a Map
Implementing Sets

Sorted Array = decent
  • $O(n)$ insert, $O(\log n)$ member, $O(n+m)$ union, intersection

Hash Tables are better, in the average case, worse in the worst case.

Self Balancing BST we haven’t covered yet, but they’re better than a sorted array
Set Practice

Try to implement a Set ADT (insert, __contains__, remove, union) using a **sorted Array**

I posted code that uses an unsorted array. Improve it!
Implementing Maps

Not really any different than implementing sets, honestly.

You have keys AND values. Just need a second array. Or, an array of objects with 2 fields. (Or an array of length 2 lists…)

Module 5, Algorithm Analysis

In CS234 we care about two things when it comes to an algorithm (in addition to “does it work?” of course)

1. How fast is it?
2. How much memory does it use?

Typically we measure these worst case and sufficiently large n.
Module 5, Big-O

We use Big-O notation to handle the “sufficiently large n” as well as ensure we do not need to care about counting every last operation / every last variable of memory.

\[ O(f(n)) = \{ g | \exists c, n_0 \forall n > n_0 g(n) \leq c \cdot f(n) \} \]
Big-O, Less Math

O(n) is the set of all functions f(n) where, for sufficiently large n, doubling n doubles f(n) (roughly)

It means “on the order of”

Big-O Addition

O(f(n)) + O(g(n)) =
O(f(n) + g(n)) =
O(max(f(n), g(n)))
\[ O(f(n)) + O(g(n)) = O(f(n) + g(n)) \]

Let \( h_1 \in O(f(n)), h_2 \in O(g(n)) \)
\[ h_1(n) \leq c_1 \cdot f(n), \text{ when } n > n_{0,1} \]
\[ h_2(n) \leq c_2 \cdot f(n), \text{ when } n > n_{0,2} \]

Therefore:

\[ h_1 + h_2 \leq c_1 \cdot f(n) + c_2 \cdot g(n) \text{ when } n > \max(n_{0,1}, n_{0,2}) \]
\[ c_1 \cdot f(n) + c_2 \cdot g(n) \leq c(f(n) + g(n)) \text{, for } c = \max(c_1, c_2) \]

There, \( h_1 + h_2 \in O(f(n) + g(n)) \)
## The Orders (a selection)

<table>
<thead>
<tr>
<th>Order</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>Log-Linear</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential (technically each base is its own order)</td>
</tr>
</tbody>
</table>
The Big O Family

O(n^2) – Really means “Quadratic or better”. Look at the definition. Log n is a member of O(n^2)

There are other sets in the family:
Big-Omega
Ω(n^2) Things that are “Quadratic or worse”. Definition is the same as Big-O, but it’s a > not a <.

Theta of n^2, or Θ(n^2) means both Big-O and Big-Omega of n^2
Module 6, Linked Lists

A sequence of **nodes**, which are a simple class of Object that keeps track of whatever node is **next** (or None if there is no next value)

class Node:
    def __init__(self, item):
        self.data = item
        self.next = None
A Node With No Name

We call the first node in a linked list the “head”.
Sometimes we also call the last node the “tail”.

We rarely call any of the other nodes by a name. They are only found by following the chain.
A Linked List

Head might be a variable we have, or the field of a wrapper Object
Wrappers

• A Wrapper Object lets us make an ADT out of a linked list.

• User does not know about nodes, so they cannot do something silly that makes two Linked Lists that share nodes in common

• User does not need to know anything about Linked Lists to use the Sequence ADT from A2

• The wrapper can have additional fields
  • Tail – Useful for add_back
  • Length – Useful for O(1) __len__ function
Doubly Linked List

If we have a “prev” field in our nodes, we can move backward as well as forward.

Let us to remove_back in O(1) time (when combined with a “tail” field in our wrapper).

Uses more memory (need 2 references per node, not 1)
- $2n$ or $1n$, it’s still $O(n)$ memory.
Skip Lists

• By having multiple layers of linked list, we can achieve $O(\log n)$ search instead of needing linear search.

• To do this “online” need to use a random algorithm for picking the heights

• Average Case performance is $O(\log n)$, worst case is still $O(n)$
Searching

Linear Search = Linear Time, O(n)

Binary Search = O(log n) probes * cost of probing index i

- For an array, the cost is O(1), so we end up with O(log n)
- For a linked list, the cost of probing index i is O(i), so we end up with O(n log n), worse than linear search!
  - Use a Skip List, if that’s what you want to do with your linked list!
Searching by Key

You don’t need to have your search criteria be “Equal to”, you can use any function you want

• Search a list of students for people who are in their 2B term
• Search a list of numbers for odd numbers

To binary search for this, they must be sorted according to the criteria

• If students are sorted by name, you cannot binary search for their terms
Sorting

Bubble Sort – (n-1) traversals.
   Each traversal checks all (n-i) adjacent pairs and swaps ones that are out of place.
   Cost: O(n^2)

Selection Sort – (n-1) traversals.
   Each traversal finds the smallest value that hasn’t been placed, and then places it at the front.
   Cost: O(n^2)

Fewer swaps than bubble, exactly as many comparisons.
Sorting

Insertion Sort: \((n-1)\) traversals.

Each traversal inserts the value into a sorted list (maintaining sorted-ness)

Cost: \(O(n^2)\), but \(O(n)\) in the best case (insert can be cheap if the value is already in the right spot)
Mergesort

Requires merge helper function: Consumes two sorted lists, returns a single sorted list containing all elements from both inputs

Cost: $O(n+m)$

Mergesort:

1. Split List in half.
2. Mergesort two halves.
3. Merge sorted Halves
Mergesort Cost

\[ T(n) = O(n) + 2T(n/2) = O(n \log n) \]

\( O(n) \) from split and merge operations, \( 2T(n/2) \) are 2 recursive calls.

Substitute in:
\[ T(n) = O(n) + 2(O(n/2)) + 4T(n/4) = 2O(n) + 4T(n/4) \]
And again and again and again
\[ T(n) = i O(n) + 2^iT(n/2^i). \] Reaches base case when \( i = \log n \)

\[ T(n) = O(n \log n) + n O(1) = O(n \log n) \]
Quicksort

1. Pick a pivot
2. Partition the list into [less than pivot] and [greater or equal to pivot]
3. Put the pivot between the two
4. Sort the two partitions recursively
5. The list is now sorted
Quicksort - Cost

In the worst case, one of the partitions contains n-1 values (everything but the pivot). This requires n recursive calls to complete, so the cost will be $O(n^2)$

- n calls, each call involves partitioning, which is $O(n)$
Stacks

• Push – Adds to Top
• Pop – Removes from top

LIFO – Last in, first out

• Used to keep track of function calls
• Useful for evaluating expressions (prefix and postfix are straightforward, infix notation more complex, but still involves a stack)
• Useful for reversing things

Implement with: Linked List or Dynamic Array
Queues

- Enqueue – Add to Back
- Dequeue – Remove From Front

FIFO – First In, First Out

Useful for algorithms we haven’t got to yet
Useful for a literal queue (phone queue, login queue)

Implement with: Linked List or Circular Array (can be tricky)
Hash Tables

An array of Buckets

Hash Function $h(x)$ tells you what bucket to store the values in

You can store several values in one bucket (Separate Chaining) or use the hash function as a starting point (Open Addressing)
Open Addressing

If a bucket is full, try another one. Keep going until you find a free one. Important to use a good “probe” (method of picking the next bucket)

Deleting is hard (need to rearrange the whole table, or leave a “deleted” marker)

Load factor ($\alpha$): Percentage of buckets that are full
    As it gets higher, performance slows down greatly
Probes

Linear Probe: Add 1 (or another constant c) to the bucket each time you probe a full bucket.
   Works as long as c and n (length of array) are mutually prime

Quadratic Probes:
\[ p(i) = h(x) + (-1)^i i^2 \quad \text{if } n \text{ is prime (and } n \% 4 == 3) \]
\[ p(i) = h(x) + i(i+1)/2 \quad \text{if } n \text{ is a power of 2} \]
Double Hashing

Linear probe, but the coefficient is picked by a secondary hash function.

Works if: second hash function always makes things mutually prime with $n$.

Example 1: $n$ is prime, secondary hash function does not make multiples of $n$.

Example 2: $n$ is a power of 2, secondary hash function makes odd numbers.
Separate Chaining

Each Bucket is independent (separate)

Don’t need to worry as much about load factor

- Data structure is more complicated (uses more memory, potentially)
- Add and delete are more straightforward