EECS 268 Notes

1/21 :)

Review

- var_name = initial_value
- # single line comment
- ‘ ‘ ‘ -> to create multi-line comments
- Types:
    - Integers
    - Floats
    - Strings
    - Boolean

1/24 :)

- More review stuff

1/31

Exceptions!

- Exception handling:

1. The simplest form of exception handling:
    - *try:*
        - *Attempt something risky*
    - *except:*
        - *Code that runs if an exception occurred in the try block*

    - NOTE: For functions raising an exception is an alternative to returning a value
    - When an exception is raised…
        1) If the exception was raised during an assignment, the assignment is aborted
        2) You go directly to the except block, any other code in the try block is skipped

2. try except else
    - *try*:
        - *Something risky*
    - *except:*
        - *response to the exception*
    - *else:*
        - *Code that runs if no exceptions were raised in the try block*

2/7

<u>Linked Structures</u>

- We're going to make Stacks, Queues, Linked Lists that all have the same building block:
  - Node
- What's a Node?
  - It contains an item (e.g. number, string, anything)
  - It also can "link" to another Node
- Recall how variables work in python and learn a couple of new keywords.

  list1 = [1, 2, 3, 4]
  list2 = [1, 2, 3, 4]
  list1 == list2
          >> True
  list1 is list2
          >> False
  list1 = list2
  list1 is list2
          >> True
- None is a keyword that represents not referring to anything

2/9

<u>Stacks</u>

- When adding to a stack, you add to the **<u>top</u>**
- When you remove, you remove from the **<u>top</u>**
- You can only interact with the topmost box.

- ***Push***:
  - Adding to a stack
- ***Pop***:
  - Removing from a stack
- ***Peek***:
  - Look at the value at the top of the stack
- ***is_empty***:
  - Is the stack empty?

#driver.py

- from stack import Stack
- def main():
  - num_stack = Stack() #empty stack
  - num_stack.push(5)
  - num_stack.push(10)
  - num_stack.push(15)

2/11

Going Back to Exceptions

- You can choose to raise an exception

```
def my_div(num1, num2):
    if num2 != 0:
        return num1/num2
    else:
        raise ZeroDivisionError('num2 is zero')
def main():
    try:
        ans = my_div(5, 0)
    except ZeroDivisionError as error:
        print(error) #prints custom message
    else:
        print(ans)
```

- You can have as many except blocks for different errors as you want

New Data Structure: Queue

- ***enqueue*** - add to the back of a queue
- ***dequeue*** - remove the front of a queue
- ***peek_front*** - look at the front of the queue
- ***is_empty*** - is it empty?

2/14 <3

More Queues

2/16

Stack Application

- Stacks:
    - Assume I push 5, 10, 15
    - 5 is at the bottom
    - 15 is at the top
- A stack is First In Last Out (FILO/LIFO) data structures

- Queues:
    - Assume I enqueue 5, 10, 15
    - 5 is at the front
    - 15 is at the back
- A Que is a First in First Out (FIFO/LILO)

- Goal: Design a parenthesis balance checker
- What's balanced?

- You need a matching right parenthesis for every left parenthesis

New Data Structure: Lists!

- A generic list allows for arbitrary access (reading, writing, inserting anywhere in the list)
- Our implementation of a List will be a node-based implementation
  - It's known as a Linked List (singly linked, one link)
- A linked list has a single entry point: front
- How do we traverse a link?
- 0   1   2
- A ->   B ->   C
- for i in range(num_jumps):
  - temp_jumper = temp_jumper.next

2/23

Inserting into a linked list

- Inserting will be the only way to add
- Insert takes an entry and an index
- We can insert at:
  - Index 1 (middle case)
  - Index 0 (front case)
  - Index length (end case)
- DON'T forget to increase your length!!

2/25

MIDTERM MARCH 9TH

Remove from a linked list

- Valid range? → 0 to (length - 1)
- Common case: remove somewhere in the middle of the list
- Special cases:
  - Removing the front
  - Invalid index

3/2

Recursion

- What happens if a function calls itself?

Scope/Call Stack review

- Recall function scope basics:
- Q: What variables "belong" to a function?
  - Parameters
  - Locally declared variables
- When does a function (either parameter or locally declared variable) fall out of scope/leave the call stack?
  - When the function returns/ends

3/4

- A note about last class's notes:
    - Instead of reassigning your parameter, typically you just pass a new value into the recursive call.

Recursive Strategies

- Identify the "bases case(s)"
    - When do I NOT have to recurse? (When am I finished?)
- Remember the goal of a single recursive call is to take a "little bite" out of the problem and then let the recursion deal with the rest.
- Your recursive call must be working towards hitting your base case

3/7

<u>Midterm Review</u>

Format

- Conceptual:
    - Short answer
    - T/F
    - Multiple Choice
- Code tracing:
    - given some code either describe what it does or how it does it (e.g. talk about call stack and heap)
- Code writing:
    - Just like our board works

Topics

- Everything! (minus recursion)
- The C++ to Python Guide (focus on python)
    - EECS 168 skills are fair game
    - Content of Lab 1
- Exception handling
    - Raise exceptions
    - try-except blocks
- Linked structures
    - Node class
        - Our building block for all linked structures
    - Stack (node-based implementation)
        - Understand how push, pop, peek, is_empty work
        - What ordering does a stack provide?
    - Queue (node-based implementation)
        - Understand how enqueue, dequeue, peek_front, is_empty work
        - What ordering does a stack provide?
    - LinkedList
        - Be comfortable traversing across a series of nodes
        - The front is the one and only entry point

    - Be able to discuss similarities and differences between our linked structures.
- No recursion on the exam

Top 10 (or fewer) ways to study for the exam to guarantee an A (maybe, it's up to you)

- #1 best is to come to class and do board work (either you did this or didn't at this point)
- Use the lecture archive
- Redo old board works (all in archive)
- Practice with as few aids as possible
- Keep your focus on the lecture material and what we've done in the lab
- Assuming they've been graded, reading the lab of a friend

3/23/22

Recursions with Backtracking

- The n queen problem
- Backtracking:
    - If you hit a dead-end, tell a previous "queen" to try a different "space."

3/25/22

Solve a text-based maze

- Contents of our text maze:
    - W - walls
    - P - passages
    - E - exit
- Rules for the maze:
    - You cannot walk through walls
    - You cannot move diagonally
- Moving algorithm
    - Check for valid moves in this order: UP, RIGHT, DOWN LEFT
    - As soon as we see a move, take it
    - When we move to a new position, you must "mark" it.

3/28/2022

Let's sketch out a MazeWalking class

- Data (i.e. member variables)
    - The Maze contents (all those Ps, Ws, E)
        - List of strings?
        - List of lists? Each sublist has characters
        - Separate class for the Maze?
        - A lot of choices
    - The visited data
        - Similar/same storage means as our Maze
    - The current step

- Recursive walk
    - def walk_maze(row, col):
        - mark(row, col)
        - if is_exit(row, col):
            - return True
        - #check up
        - elif is_valid_move(row - 1, col):
            - is_exit_found = walk_move(row-1, col)
            - if is_exit_found:
                - return True
        - #check right
        - elif is_valid_move(row, col+1):
            - is_exit_found = walk_move(row, col+1)
            - if is_exit_found:
                - return True
        - #check down
        - #check left (similar to previous checks)
        - #if we make it to this spot in the method, none of the directions led to an exit. This is our second base: dead-end
        - unmark(row, col):
            - return False

3/30/2022

Complexity Analysis

- How does an algorithm scale in the following categories:
    - Time
        - Number of instructions executed
        - NOT clock-on-the-wall time
        - Clock-on-the-wall time varies based on the hardware
    - Space
        - Memory allocation
- Big-O notation
    - If an algorithm has constant complexity, we would say it has a time/space complexity of **O(1)**
    - For linear, **O(n)**
    - For n^2, **O(n^2)**
    - And so on…
    - Big-O only retains the most influencing factor of n. All coefficients and other factors of n are ignored.
    - Assume an algorithm had 4n^2 + 3n + 9 instructions, the Big-O is still O(n^2)

- Let’s consider some of the algorithms that we created for our data structures
- Stack

        - What is the time complexity of a push?
            - Make a node, put a value in it, keep track of what's on the top, move the top, set next of the new top to the old top
- Board Work:
    - O(1), O(1), O(n), O(1), O(n)

04/01/2022! :]}

<u>Space Complexity</u>
- Memory allocation
- There's "typical allocation (e.g. list of size n)
- Don't forget that recursion requires space too
    - Specifically on the stack
- Our factorial function requires n calls, then it has O(n) space complexity

<u>Sorts</u>
- Iterative sorts
    - Bubble sort
        - for n-1 elements:
            - Loop through all neighbors and if any neighbors are out of order, swap them
    - Selection sort
        - for n-1 indices:
            - find the min value from that index onward and swaps it into the current index
    - Insertion sort
        - Expand a sorted section one index at a time
        - With each addition to the sorted section, shift it into the correct position.
    - Bogo sort
        - 1) Shuffle all values
        - 2) See if it's sorted
        - 3) Repeat as needed
        - WARNING: NEVER USE

04/04/2022

<u>Recursive Sorts</u>
- **Merge Sort**
    - Utilizes a non-recursive helper function, merge
    - Helper function: merge
        - Take two sorted collections (e.g. lists, arrays)
        - Merges the numbers into a single collection
        - Complexities of merge
            - We get two lists to merge into a list of size n (each parameter size n/2)
            - Time complexity: O(n)

            - The “buffer” is size (n/2) + (n/2)
            - Space complexity: O(n)
    - Recursion in Merge Sort
        - “Break” the collection in half over and over again
        - Merge the “broken” collections back together
    - How many levels of recursion are required to break the original collection into single elements?
        - $n/2^k = 1 \rightarrow n = 2^k \rightarrow \mathbf{log_2 n = k}$
        - We merge n elements on each recursive level.
        - If merge takes O(n) time and we have $\log_2(n)$ levels.
        - What is the complexity of Merge Sort?
            - Time complexity: $O(n*\log_2(n))$
            - Space complexity: O(n)

4/6/2022

- Back to the space complexity of merge sort
    - Which is worse: O(n) or $O(\log_2(n))$?
        - O(n) will have more instructions to get through

- **Quick Sort**
    - Much like merge sort, it uses an iterative helper function: partition

    - Partition
        - Iterative
        - Takes in a collection of numbers, chooses a pivot (today we’ll just pick the last value), arranges all values in the collection such that…
            - Values less than the pivot are left of the pivot
            - Values greater than or equal to the pivot are right of the pivot
    - The recursive part of quick sort simply recursively partitions over and over
    - All of the partitionings are “in-place” they affect the list/array/collection that’s given to it.

4/8/22

<u>New Data Structure: Trees</u>

- Up until now, all of our data structures were linear, meaning there was one path through a stack or a queue or a linked list
- The only time we’ve had a choice in direction was in MazeWalking or more generally graphs
- Trees are in fact a type of graph, but they are special
    - Trees are acyclic graphs, meaning there are no cycles
    - While trees are graphs, we won’t need to mark as we go
- Tree info:
    - The starting point/entry is called a **root**

    - An N-ary tree is where the number of connections from parent to children differs.
    - Nodes with no children are called “**leaves**”

- Binary Trees
    - Consist of Binary Nodes: nodes with at most 2 children
    - A binary node (**parent**) can have a **left child** and a **right child**
    - We will use recursion to traverse our trees
    - Everything on the left side is the left subtree of the whole tree
    - Everything on the right side is the right subtree of the whole tree.
- Traversal Orders:
    - Pre-Order
        - 1) Visit (e.g searching, counting, printing the entry in the node)
        - 2) Traverse into the LST
        - 3) Traverse into the RST
    - In-Order
        - 1) Traverse into the LST
        - 2) Visit (e.g searching, counting, printing the entry in the node)
        - 3) Traverse into the RST
    - Post-Order
        - 1) Traverse into the LST
        - 2) Traverse into the RST
        - 3) Visit (e.g searching, counting, printing the entry in the node)

4/11/2022

- Today we’re going to “sketch” out the implementation of a “plain” binary tree
- We’re going to make a node-based implementation
- For any of our trees, we’re going to traverse recursively

```
def main():
        mytree = BinaryTree()
        mytree.add('A')
        #repeat B, C, D
        if mytree.search('D'):
                #yay it's there
```

- Think about how someone using a BinaryTree would search for a value.
- Our tree implementation will contain “public-facing” non-recursive methods that call recursive helper methods that actually traverse the tree.

4/13/2022

- Count all instances of something in the tree
- If it is at the top
    - 1+ count of lst + count of rst
- If not at the top
    - 0 + count of lst + count of rst

<u>Adding to a Binary Search Tree</u>

- If the subtree being added to is empty, just add it
- When adding to a non-empty tree/subtree, compare the value you're adding to the node you're on
    - If the value being added is less than the value of the current node, add it to its LST (Strings can be compared via ASCII!!)
    - If the value being added is greater than the value of the current node, add it to its RST
    - No duplicates allowed!

4/15/22

Traversals in BST

- What do we notice about traversals? (after doing the board work)
    - In-order — it's in order!
    - Pre-order — can recreate an existing BST!
        - (you get the same tree as the original)
    - Post-order — will be useful when you clean up the nodes allocated (e.g. C++)

Traits of a BST

- Smallest value:
    - Go as left as you can
- Largest value:
    - Start at the root and go as far right as you can

Let's talk about the "search" of BST

- Everything starts at the root
- Recall in "plain" binary trees we did an exhaustive search where we searched the LST and RST
- Searching is simpler

4/18

<u>BSTs: Searching</u>

- Think about how you search in the "real world"
- We want to gain information when we search, not just confirm that a value is or isn't in a BST

- We will use two types in our BSTs:
    - Key type: use for searching
    - Item type: the entries in the nodes

- Let's fill a BST with KU courses
    - Assume there is a KUCourse class
    - These objects have department, number, and amount of credit hours
    - Example KUCourse object:

        - EECS, 268, 4.0
    - Assume KUCourse class has defined __lt__, __gt__, __eq__, for comparing one KUCourse to another
    - Assume the comparison is defined to compare the course number
- Add the following KUCourses to a BST
    - EECS 168 4.0
    - EECS 268 4.0
    - EECS 140 4.0
    - EECS 368 3.0
    - EECS 448 4.0

- To search:
    - The keytype is going to be a piece of your item type.
    - Keys must be unique because BST cannot contain duplicates
    - The KUCourse will also need the ability to compare KUCourse to ints
- You get the object out of the search rather than just a confirmation of whether it exists.

4/20

BSTs: Adding and Common Pitfalls

- Recall the rules:
    - If a subtree is empty, just put the value there
    - If the subtree is not empty, then compare the new value to the value in the current node and figure out whether to go left or right

- Add 10, 5, 15 to a BST
- Recall some of those edge cases/corner cases/ special cases from lists, queues, stacks

- #sketch of a recursive add in my BST class
- def_rec_add(self, entry, cur_node):
    - If cur_node == None:
        - cur_node == BNode(entry)
        - #BUT this won't change the left or right passed in! So it's not right
    - #compare and then recurse left or right

- You cannot pass control of a left or right member variable (or any variable) to a function
- Our add, unlike search or other tree methods we've written that can go all the way to None without an issue, needs to stay one "level" away from None.

4/22

BST: Removing

- Zero Child Case (aka removing a leaf)
    - Recall that every node has one reference to it. This reference must be updated.
    - Set the parent's right/left child to None
- One Child Case

    - Doesn’t matter which child (left/right)
    - Values still must be in the correct subtree
    - The one child, takes the place of the target
- Two Child Case (aka the doom bringer)
    - Finding a replacement candidate and pick ONE:
        - 1) The largest value in the LST (found by going the most right)
        - 2) The smallest value in the RST (found by going the most left)
    - Perform a “remove” on the candidate but don’t lose track of it.
    - It will replace the original target

4/25

- Hints for implementing remove:
    - 1) Don’t forget that every node has exactly one thing referencing it. That reference needs to be updated.
    - 2) Don’t forget that when you recurse, you can use recursion to return important information to previous spots
    - 3) Think of each call as “sitting” on a single node

<u>Heaps</u>

- **Min-heaps**
    - Binary trees
    - Rules about where values can be placed
    - Rules about the structure of the tree
    - Heaps are *complete* binary trees
    - *Complete* binary trees: filled in level by level, from left to right
        - (Left child first then right child)

- Rules for value placement in Min-heaps
    - A node’s value must be less than OR equal to both of its children individually
    - Its children must also be min-heaps
    - WARNING: remember these aren’t BSTs!

- Adding to a min-heap:
    - 1) New value is placed in a position to keep the tree complete.
        - It’s placed in the leftmost position in the current unfilled level
    - 2) Upheap the newly added value
        - Compare the value against its parent
        - Swap if needed
        - Continue upheaping as needed

4/27

- Removing from a min-heap:
    - 1) Remove the root
    - 2) Take the lowest rightmost node and make it the temporary root

        - This keeps the tree complete
    - 3) Downheap the new root
        - Compare against the smaller of the children
        - Swap if needed

4/29

- Are there other ways to implement a binary tree other than with nodes?
    - Yes
- How to implement a binary tree with a python list?
    - No BNode class
    - How does one access any element of a list?
        - An index
    - We'll use various index formulas to emulate being binary tree
        - root index: zero
        - left child of index i: 2*i + 1
        - right child of index i: 2*i + 2
        - parent of index i: (i - 1) // 2
    - This is not helpful with incomplete trees because there is a lot of empty space left over that is required but unuseful.
    - This is helpful for complete binary trees

In order to use as heap

- We need to quickly add new values to the leftmost position in the shallowest level
    - This is just an append!!
- When removing
    - 1) Find the root
    - 2) Replace the root with the deepest/lowest level's rightmost 'node'
        - Swap value at index length - 1 (aka index[-1]) with the root
        - Pop last index
        - Downheap using formulas

Heaps. What's the point?

- We can only remove from the root
- We don't have any say on where values get placed in a heap
- Heaps are self organizing
    - Min heaps keep the smallest value at the root
    - Max heaps keep the largest value at the root
- Any data type that supports comparison operators (<, <=, >, >=, ==, !=)
- Use cases:
    - Sort data added in an arbitrary order
    - Implement a priority queue: a queue where higher priority elements are pushed towards the front
        - Examples of priority queues:
            - Airport passenger boarding orders

                - Hospitals and patient priorities (aka triage)

5/2/22

<u>Inheritance</u>

- Relationship we have been able to model with Classes
- The relationship that we've modeled so far is known as the "has a" relationship
    - Example: A Car has Tires
- The next relationship we want to model is the "is a"
    - Example: A Dog is an Animal

Class Hierarchy

- Animal class
    - Base class
    - Super class
- Dog class
    - Derived class
    - Sub class
        - The subclass add more specialized functionality (e.g. do_trick in Dog)
    - 'Dog inherits from Animal'
    - Dog → Animal
    - Poodle → Dog → Animal

5/4/22

<u>Final Review</u>

- NOTE: 2.5 hour exam, but not written to take all 2.5 hours — 3.75 hours for me
- Format
    - Very similar to the midterm
    - Conceptual
    - Visualization (e.g. draw a BST)
    - Code writing
- Topics
    - Heavy focus on post-midterm topics
    - **Recursion**
        - Basic recursive functions just mimicked loops
        - Identifying base cases
        - Recursive functions that return values
    - **Recursion with Backtracking**
        - N-Queens problem, Knight tour
        - Maze Walking / Blob lab
        - Marking/unmarking visited spaces
    - **Complexity Analysis**
        - Don't worry about python specific functions for timing
        - Do be aware of why we're not interested in "clock on the wall" time

        - Supply the Big-O complexity for (time or space) for various methods we’ve written
    - **Sorts**
        - Iterative sorts: bubble, insertion, selection
        - Recursive sorts: merge and quick
        - Won’t have to code any sorts from scratch
        - Compare and contrast the different sorts
    - **TREES**
        - Tree vocab (e.g. root, subtree, leaf)
        - Binary Trees
            - Left and right children
            - Exercises in “plain” binary trees
            - Public facing method that class a “hidden” recursive method
        - Binary Search Trees
            - Rules for adding
            - How the rules for adding affect placement of values (e.g. where is the largest value in BST)
            - How searching a BST was more efficient than searching a “plain” Binary Tree
            - Key Types VS Item Type
            - Traversal orders
        - **Heaps**
            - We didn’t have a heap lab, but we did talk about how an implementation could work
            - Binary Trees
            - Rules for value placement
            - Rules for structure: completeness
            - How to add and remove
            - Upheaping and downheaping
            - List-based implementation, index formulas
            - Why heaps are compatible with list-based implementation as opposed to a random “plain” binary tree
        - **Inheritance**
            - Difference between the “is a” relationship versus the “has a” relationship
            - Syntax: how to inherit, key function super()