# Exam 2 Study Guide

## Week 7 and Week 8: Lists, Tuples, Searching, Sorting, Complexity

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | ‘a’ | 5.4 | 6 | ‘b’ | Value |
| 0 | 1 | 2 | 3 | 4 | Index |

* Lists: list =
  + Ordered, changeable collection
  + Empty list 🡪 list = []
  + Same indexing as string
  + Time Complexities
    - Access element by index: O(1)
      * array[4]
    - Append element to end: O(1)
      * array.append(4)
    - Insert element at front: O(N)
      * array.insert(0, 2)

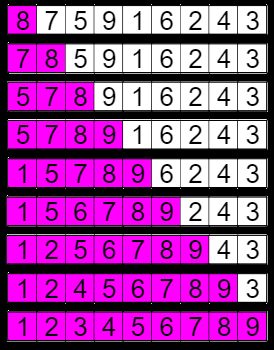
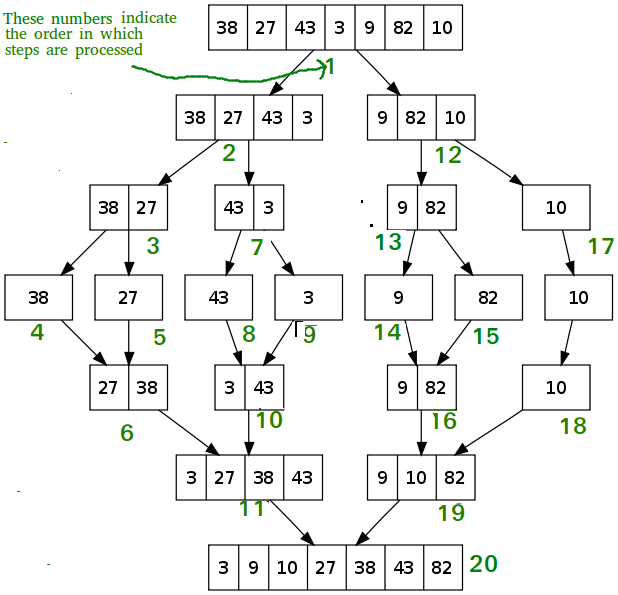
|  |  |  |
| --- | --- | --- |
| **Function** | **Return** | **Description** |
| len(list) | 5 | returns length of list |
| list[2] | 5.4 | returns value at index 2 |
| list[2] = value | none | changes value at index 2 |
| list.append(value) | None | adds value to list |
| list.remove(value) | None | removes value from list |
| list2 = str.split(‘,’) | str = ‘8,6’  **list2 = [8 , 6]** | creates a list from values split up by commas in str |
| list.extend(other\_list) | other\_list = [6,9]  **lst = [1,’a```’,5.4,6,’b’,6,9]** | Extends list by adding elements of other\_list to the end of list and returning that finished list |

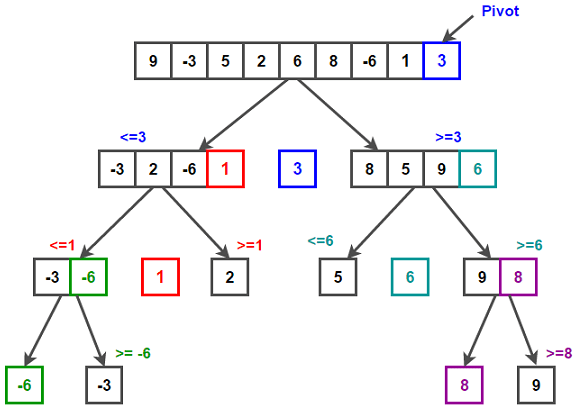
* Tuples: tuple = (1, 'a', 5.4, 6, 'b')
  + Ordered, unchangeable collection
    - Unchangeable = cannot change any values after declaring tuple
  + Empty tuple 🡪 tuple = ()
    - Tuple with one value 🡪 tuple = (‘b’,) \*MUST INCLUDE COMMA\*
  + Same indexing as list and string
  + Functions
    - len(tuple) = return length of tuple
* Can loop through lists and tuples:
  + For i in range(len(list)):

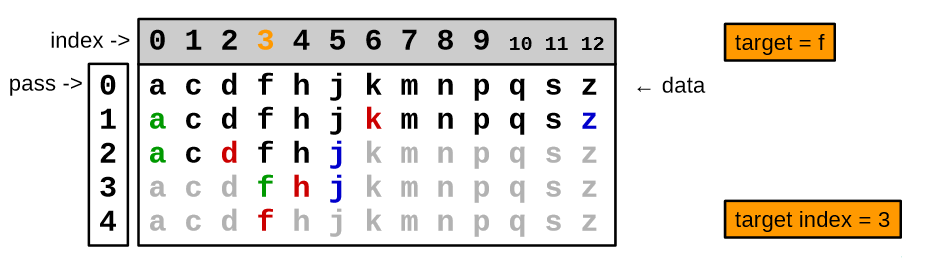
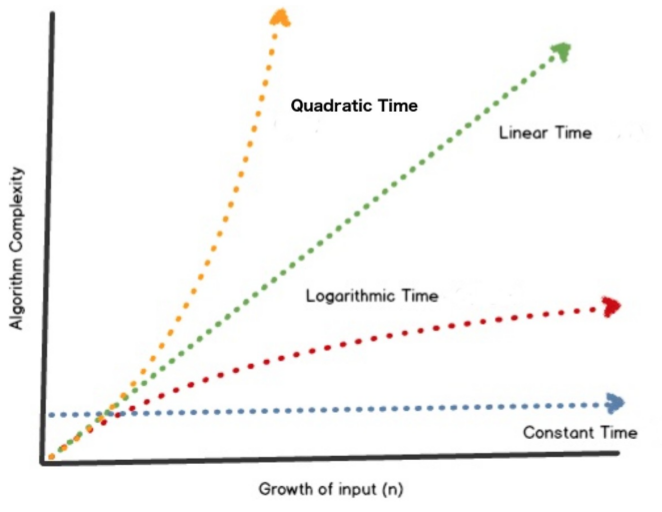
print(list[i])

* + For value in list: (for-each loop)

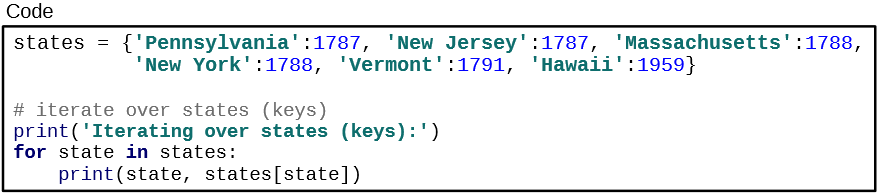
print(value)

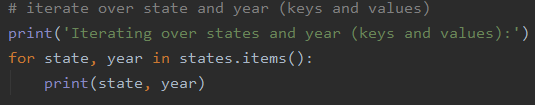
* Sorting Algorithms
  + Insertion Sort
    - 2 parts:
      * Left part = seen
      * Right part = unseen
    - Sort left part first
    - Time Complexity
      * Worst Case (reverse order)
        + O(N2) – swapping is maximized
      * Best Case (sorted)
        + O(N) = just has to run through the list
  + Merge Sort
    - Splitting the list into even and odd indices – down to the smallest index. We then sort the these split lists, sort them, and then merged them back together
    - Can be sorted in place or with additional sorts
    - Think “Divide and Conquer”
    - Time Complexity
      * Best Case: O (N log N)
        + Split = log N
        + Merge = N
      * Same applies for average and worse case
        + This is because, in all cases, you are splitting and merging
  + Quick Sort
    - There is a pivot
    - Partition list into three parts: values greater than pivot, values less than pivot, and values same as pivot
    - Repeatedly sort the less and greater lists and recombine them via concatenation with the equal lists
    - Time Complexity:
      * Best Case: O(N log N)
        + less and greater partitions at each level are evenly sized
      * Average Case: O(N log N)
      * Worst Case: O(N2)
        + Sorted array and pivot is last element every time



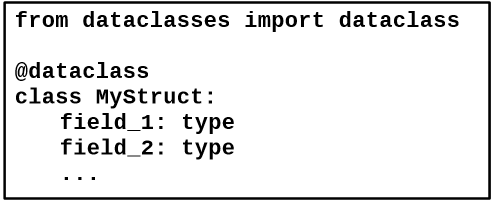
* Searching Algorithms
  + Binary Search
    - finds the index of a target element in a sorted list of values
    - At each pass, we cut the list in half and there are three possible outcomes:
      * If the target equals the middle value, we return the middle index
      * If the target is less than the middle value
        + Cut the list in half, and disregard the greater half
        + Search the left half
      * If the target is greater than the middle value:
        + Cut the list in half, and disregard the lesser half
        + Search the right half
    - Time Complexities
      * Best Case: O(1)
      * Worst Case: O(log n)
  + Linear Search
    - Finds the index of a target value by checking every element sequentially in an unordered list
    - Time Complexities
      * Best case: O(1)
        + The target is the first element
      * Worst case: O(N)
        + The target is not even in the list
* Time Complexities (in order from Fastest to Slowest)
  + Constant, O(1)
    - Accessing an element in a list by index
  + Logarithmic, O(log N)
    - Binary search for the index of an element in a sorted list
  + Linear, O(N)
    - Accessing all elements in a list
  + Log-linear, O(N log N)
    - Optimal sort time using merge sort or average case quick sort
  + Quadratic, O(N2)
    - Worst case insertion sort and quick sort

## Week 9: Dictionaries, Data Classes

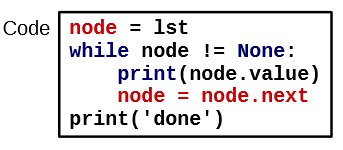
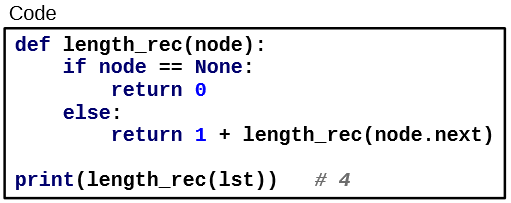
* Dictionaries
  + an unsorted collection that associates a unique key with any value
  + The insertion order is maintained
  + dictionary = {key: value, key: value, key: value,…}
  + dict = {‘Name’: Nihal, ‘Age’: 18, ‘Class’: ‘Freshman} OR dict = dict()
  + Time Complexities
    - All the following are O(1) operations
      * Key membership
      * Lookup of value based on key
      * Add a new entry
      * Update an existing entry
  + Iterating through Dictionaries
    - Iterating through the dictionary by key
    - Iterating through the dictionary by values (use dictionary in snippet above)



|  |  |  |
| --- | --- | --- |
| **Function** | **Return** | **Description** |
| dict[‘Name’] | Nihal | returns value at key |
| dict[‘Name’] = Neil | None | Sets value at key ‘Name’ to Neil |
| Name in dict | Boolean: True | Checks to see if an key is in a dictionary |
| len(dict) | 3 | Returns number of items in dictionary |
| dict.keys() | [‘Name’, ‘Age’, ‘Class’] | Returns a list of keys in dict |
| dict.values() | [‘Nihal’, 18, ‘Freshman’] | Returns a list of values in dict |
| dict.get(“Name”) | Nihal | Returns the value at key “Name” |
| dict.pop("Name") | None  Dict = {‘Age’: 18, ‘Class’: ‘Freshman} | Removes the item (as well as the key) with the key Name |
| dict.popitem() | None | Removes the last inserted item |
| dict.update({‘Major’:’ CS’}) | None  Dict = {‘Name’: Nihal, ‘Age’: 18, ‘Class’: ‘Freshman, ‘Major’:’ CS’} | Adds the dictionary in () to the current dict |

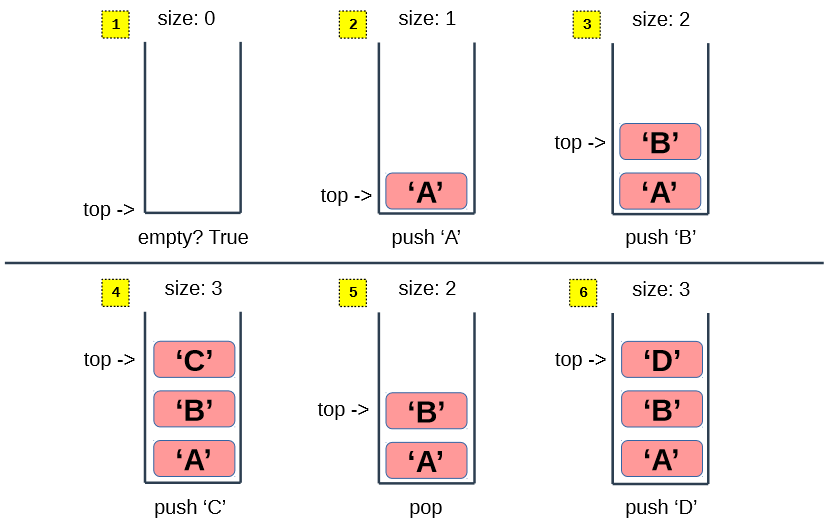
* Sets
  + an unordered collection with no duplicate elements
  + Dictionary of ONLY keys
  + NOT THE SAME AS A DICTIONARY
  + Set = (‘Name’, ‘Age’, ‘Class’)
  + Time Complexities
    - Checking whether an element exists in a set or not is an O(1) operation
* Data Classes
  + Accessing the fields
    - MyStrict.field\_1
  + Defining new MyStruct object
    - mystruct1 = MyStruct(field\_1, field\_2)

## Week 10: Linked Lists

* Designed as a dataclass
* Using the Node dataclass
  + Chain together many nodes to create a linked list
* Two ways to iterate through linked list
  + While loop
  + Recursion
* Time Complexities
  + Access element by index: O(N)
    - Worst case is accessing the last element
  + Append an element to the end: O(N)
  + Insert element at front: O(1)
    - Takes constant time to create and link a new node at the front

## Week 11: Stacks and Queues

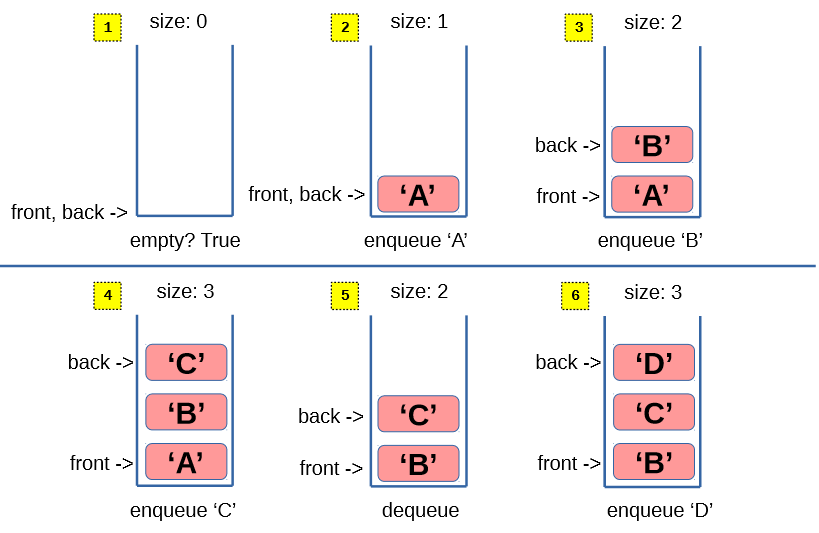
* Stacks
  + A stack is a data structure where elements are added and removed from only one end - the top
    - Last In First Out (LIFO) / First In Last Out (FILO)
  + Uses a variation of linked lists in which the dataclass is preferably mutable
  + Dataclass has two fields
    - size: int
    - top: Union[None, Node]



|  |  |  |
| --- | --- | --- |
| **Function** | **Return** | **Description** |
| make\_empty\_stack() | Queue(0, None, None) | Add an element to the top of the stack. The stack state changes. |
| push(stack, element) | None | Insert an element into the back of the queue. |
| top(stack) | None | Return top element on stack. Does not change stack |
| pop(stack) | Queue’s front value | Remove the top element in the stack and returns the removed value. The stack state changes. |
| size(stack) | Size of stack | Return the # of elements |
| is\_empty(queue) | Bool: True if the queue is empty, False if it isn’t | Is the queue empty? |

* Queues
  + A queue is a data structure where elements are added to the back and removed from the front
    - First In First Out (FIFO)
  + Uses a variation of linked lists in which the dataclass is preferably mutable
  + Dataclass has three fields
    - size: int
    - front: Union[None, Node]
    - back: Union[None, Node]

|  |  |  |
| --- | --- | --- |
| **Function** | **Return** | **Description** |
| make\_empty\_queue() | Queue(0, None, None) | Returns a new queue with size initialized to zero and back fields initialized to the empty sequence. |
| enqueue(queue, element) | None | Insert an element into the back of the queue. |
| dequeue(queue) | Removed value | Remove the front element from the queue |
| front(queue) | Queue’s front value | Access and return the first element in the queue without removing it. |
| back(queue) | Queue’s back value | Access and return the last element in the queue without removing it |
| is\_empty(queue) | Bool: True if the queue is empty, False if it isn’t | Is the queue empty? |



* All operations with Stacks and Queues have a time complexity of O(N)

### Additional Notes

* Go over following code
  + Stacks
  + Queues
  + Linked Lists (both recursion and iteration)
  + Sorting Algorithms
  + Searches