**Python Memory Management:**

*-* Every piece of data is an object

*-* Variable is a pointer to an object

*-* When no variable points to an object of data the object is

orphaned and there is no way to access it.

*-* When orphaned, python will eventually notice that it's inaccessible

and reclaim the allocated memory (i.e. garbage collection)

*-* Every object has a unique identifier

*-* Blocks of memory called pages

*-* OS handles requests to read and write memory

*-* Python code and memory management handled by Python application

*-* Python implementation converts code to instructions (bytecode)

that it runs on a virtual machine

*-* CPython converts python code to bytecode

*-* Python types in CPython are represented by a struct called

PyObject which every object in Python uses and contains two

attributes ob\_refcnt (reference count) and ob\_type (pointer to another type)

*-* Each object has its own object-specific memory allocator that

knows how to get the memory to store that object.

*-* Python GIL locks the interpreter to prevent threads from

accessing the same memory

**Python Commonly Used Collections:**

*-* Collections are containers used for storing data

(i.e. data structures)

*- DefaultDict:*

*-* Like a dict

*-* Difference is that it does not give an exception/key

error when you try to access non-existent key

[**python**]

from collections import defaultddict

nums = defaultdict(int)

nums['one'] = 1

print(nums['two']) #prints 0

[**end**]

*- Counter:*

*-* Counts the occurrences of each value present in an

array or list

[**python**]

from collections import Counter

list = [1,2,3,4,2]

answer = Counter()

answer = Counter(list)

print(answer[2]) #prints 2

[**end**]

*- Deque:*

*-* optimal version of lists

*-* can add or remove items from start to end of list

[**python**]

from collections import deque

#initialization

list = ["a","b","c"]

deq = deque(list)

print(deq)

#insertion

deq.append("z")

deq.appendleft("g")

print(deq)

#removal

deq.pop()

deq.popleft()

print(deq)

[**end**]

*- Namedtuple:*

*-* Solves the problem of remembering the index of each field

of a tuple object

*-* Returns with names for each position in the tuple

[**python**]

from collections import namedtuple

Student = namedtuple('Student', 'fname, lname, age')

s1 = Student('Peter', 'James', '13')

print(s1.fname)

[**end**]

*- ChainMap:*

*-* combines a lot of dictionaries together and returns a

list of dictionaries

[**python**]

import collections

dictionary1 = { 'a' : 1, 'b' : 2 }

dictionary2 = { 'c' : 3, 'b' : 4 }

chain\_Map = collections.ChainMap(dictionary1, dictionary2)

print(chain\_Map.maps)

[**end**]

best) search through multiple dictionaries at a time

*- OrderedDict:*

*-* ensures order is maintained even if the value of the

key is changed

[**python**]

from collections import OrderedDict

order = OrderedDict()

order['a'] = 1

order['b'] = 2

order['c'] = 3

print(order)

#unordered dictionary

unordered=dict()

unordered['a'] = 1

unordered['b'] = 2

unordered['c'] = 3

print("Default dictionary", unordered)

[**end**]

**Python Libraries:**

*- Pandas:* working with large datasets

*- Numpy:* complex math and common statistical operations

**Python vs C++:**

*-* Python runs through an interpreter vs C++ is pre-compiled

*-* Python has garbage collection

**Asymptotic Notation:**

*-* Algorithm's running time

*-* Measured by dropping the constant coefficients and less significant terms and

focusing on the rate of growth

*-* 3 forms = Big Theta, Big O, Big Omega

*- Big O Notation:* the biggest it gets

*- Big Omega Notation:* the smallest it gets

*- Big Theta Notation:* the running time as n gets larger

//Data structures (defining details, how they're implemented, runtimes, best scenarios)

**Iterators:**

contains a countable number of values that can be traversed through

strings, lists, dicts, and tuples are iterble objects

[**python**]

iter(myList)

print(next(myList)) #prints the first element

print(next(myList)) #prints the second element

[**end**]

ex) for loops create an iterator object and executes the next() method for each loop *#for x in myList:*

**Arrays:**

sequential arrangment paired with index of the data

best) dont have to append and insert, need to have data than can be refered by an index, need to refer iterate through it

*-* Lists []

*-* Dicts {'...' : '...', }

*-* Tuples ()

O(1) random access

Omega(1)

ex)

**Linked Lists:**

elements contain its data and a link to the other data

best) fast insertion and deletion if you are already at the location of the item (good for a lot of adding and deleting)

useful when you need a stack or queue

*#append*

[**python**]

def append(self, NewVal):

NewNode = Node(NewVal)

NewNode.next = None

if self.head is None:

NewNode.prev = None

self.head = NewNode

return

last = self.head

while (last.next is not None):

last = last.next

last.next = NewNode

NewNode.prev = last

return

[**end**]

O(n) random access

O(1) Insertion/deletion with reference to location

**Stacks:**

FILO order of operation

Only can operate at the top/end of the stack

Use Python Lists

*- Push:* self.stack.append(val)

*- Pop:* self.stack.pop()

*- Peek:* self.stack[**-1**]

**Queues:**

FIFO order of operation

Operates at both ends

Use Linked Lists

*- Push:*

*- Pop:*

*- Peek:*

**Dequeue:**

Adds and removes from both ends

**Hash Tables:**

Made of arrays associated with a hash function

Values retrieved from keys rather than index

*- Hash Function Should:*

*-* map large keys to small keys

*-* generate values between 0 to m-1 where m is the size of the hash table

*-* uniformly distribute large keys into hash table slots

*- Collision Handling:*

*- Chaining:*

Each cell of hash table point to a linked list of records that land there

*- Open Addressing:*

Linear Probing: go to the next empty slot

Quadratic Probing: probe for i^2th slot and then find next empty slot

*- Performance Measure:*

Load factor = Total keys / length of table

Expected time to search = O(1 + lf)

Expted time to insert/delete = O(1 + lf)

[**python**]

class hashTable:

def \_\_init\_\_(self):

#Create empty Hash Table with Chaining in mind

self.table = [[] for \_ in range(10)]

def hashing(self, val):

return val % 10

def insert(self, val):

self.table[self.hashing(val)].append(val)

def search(self, val):

index = self.hashing(val)

for v in self.table[index]:

if(v == val):

return index

return -1

[**end**]

**Trees:**

Root Nodes connected to children nodes

*- Binary Tree:*

each node connected to maximum of two other nodes

starts with a root node

left child node is <

right child nodes is >

O(n) searching and insertion/deletion

*- AVL Tree:*

left and right subtree height can not differ by more than 1

O(log\_2n) for searching and insertion/deletion

*- Trie Tree:*

*- Traversal:* for checking every node in the tree

*- In-order:* left,root,right

[**python**]

def inorderTraversal(self, root):

res = []

if root:

res = self.inorderTraversal(root.left)

res.append(root.data)

res = res + self.inorderTraversal(root.right)

return res

[**end**]

*- Pre-order:* root,left,right

[**python**]

def preorderTraversal(self,root):

res = []

if root:

res.append(root.data)

res = res + self.preorderTraversal(root.left)

res = res + self.preorderTraversal(root.right)

return res

[**end**]

*- Post-order:* left,right,root

[**python**]

def PostorderTraversal(self, root):

res = []

if root:

res = self.PostorderTraversal(root.left)

res = res + self.PostorderTraversal(root.right)

res.append(root.data)

return res

[**end**]

**Heaps:**

Special Trees where the nodes hierarchy is determined by the value of the data

*- Min:* parent is *<=* child node

*- Max:* parent is >= child node

**Graphs:**

Arrangement of nodes and edges

[**python**]

myGraph = {"a": ["b", "c"],

"b": ["a", "c"],

"c": ["a", "b"]}

[**end**]

**Sets:**

Like a list without indexes and unordered

Highly OPTIMIZED way of checking whether a specific

element is contained in the set compared to another set !using hash tables

[**python**]

{

# Python program to demonstrate working# of

# Set in Python

# Creating two sets

set1 = set()

set2 = set()

# Adding elements to set1

for i in range(1, 6):

set1.add(i)

# Adding elements to set2

for i in range(3, 8):

set2.add(i)

print("Set1 = ", set1)

print("Set2 = ", set2)

print("\n")

# Union of set1 and set2

set3 = set1 | set2# set1.union(set2)

print("Union of Set1 & Set2: Set3 = ", set3)

# Intersection of set1 and set2

set4 = set1 & set2# set1.intersection(set2)

print("Intersection of Set1 & Set2: Set4 = ", set4)

print("\n")

# Checking relation between set3 and set4

if set3 > set4: # set3.issuperset(set4)

print("Set3 is superset of Set4")

elif set3 < set4: # set3.issubset(set4)

print("Set3 is subset of Set4")

else : # set3 == set4

print("Set3 is same as Set4")

# displaying relation between set4 and set3

if set4 < set3: # set4.issubset(set3)

print("Set4 is subset of Set3")

print("\n")

# difference between set3 and set4

set5 = set3 - set4

print("Elements in Set3 and not in Set4: Set5 = ", set5)

print("\n")

# checkv if set4 and set5 are disjoint sets

if set4.isdisjoint(set5):

print("Set4 and Set5 have nothing in common\n")

# Removing all the values of set5

set5.clear()

print("After applying clear on sets Set5: ")

print("Set5 = ", set5)

}

[**end**]

*# add and checking for an item*

O(1) on average, worst case O(n)

**Recursion/Recursive Strings:**

*-* Needs an exit condition

*-* Method for \*starting at the back\*

//algorithms

*- Sort:* data searching can be optimized a lot

*- #Bubble Sort:* !never use

*-* compares each pair of adjacent elements

*-* for x in range(length -1, 0, -1)

*-* for y in range(x)

*-* check & swap

*- #Merge Sort:* O(nlogn) consistent (if worried about worst case use this)

!expensive memory usage

*-* Exit condition check if list len *<=* 1 then return list

*-* Divide list in half recursively

*-* return merge list

loop until left list or right list is empty

if left[**0**] < right[**0**]

list.append(left[**0**])

left.popLeft()

else

list.append(right[**0**])

right.popLeft()

if len(left) == 0:

list = list + right

else

list = list + left

*- #Quick Sort:* good but merge sort is better (better space complexity tho)

*-* Picks a pivot and partions around pivot

*- Insertion Sort:* use when item count is low and mostly sorted

*- Selection Sort:* !never use

*- Heap Sort:* only if youre REALLY worried about worst case and space complexity but quick sort is faster

*- Search:*

*- Binary Search:*

*-* Takes a sorted list of elements and starts looking at middle of list,

if the value matches return,

if the value is less than middle then get the middle of the first half,

if the value is more than middle then get the middle of the second half.

*-* Keeps going while idx0 *<=* idxn

*- Breadth First Search:*

*-* Check every node at the same level

*-* Use queues

1. Start with root

2. Loop through queue while not empty

3. pop queue

4. print item

5. add its children to the queue

*- Depth First Search:*

*-* Move as far down a path until a dead end is reached, then

backtrack until a nonexplored path has been found

*-* Use stacks

1. Start with root

**DS Tradeoffs:**

*- Python List:*

indexes (random access), fast insertions and deletions at tail

useful when you need a \*stack\* or \*hash table\*

! has to shift everything over on insertion/deletion i.e. O(n)

\*random access with index\* & \*insertion/deletions at tail\*

*- Linked List:*

fast insertion and deletion if you are already at the location of the item (good for a lot of adding and deleting)

useful when you need a \*queue\*

! O(n) time to traverse

\*insertion and deletion with location of the item\*

*- Hash Table:*

\*Search with no indexes and not sorted\* and \*insertion/deletion with no location\*

! O(1) on average but worst case O(n)

*- Binary Tree:*

! O(n) Searching and insertion

**Sort Tradeoffs:**

*-* Timsort > mergesort > quicksort

**Search Tradeoffs:**

//Design Patterns

**Singleton:**

**Factor:**

**Adapter:**

**Bridge:**

**Visitor:**

**Command:**

**Proxy:**

**Observer:**

//Distributed Computing

**Map-reduce:**

**Service Oriented Architectures:**

**Distributed Caching:**

**Load Balancing:**

https://www.khanacademy.org/computing/code-org

https://www.khanacademy.org/computing/computer-science

https://medium.com/@ratulsaha/preparing-for-programming-interview-as-a-phd-student-with-python-5f8af8b40d5f

https://realpython.com/python-coding-interview-tips/

https://leetcode.com/discuss/interview-question/875845/amazon-sde-ii-interview-experience