Study Notes:

1. Data types:

List: time complexity for basic operation

<https://wiki.python.org/moin/TimeComplexity>

Array: good for indexing (for example, return the middle of an array); Not good with insertion and deletion

Stack: LIFO

Queue: FIFO

Deque is good for both ends

Linked-List: good for insertion/deletion

1. Binary search (sorted array, O(log n))
2. Recursion (Two key parts: (1) Base Cases to avoid infinite recursion (2) the recursion part )
3. Sort:
4. Basic sort method and their time complexity

Bubble sort:

def bubbleSort(arr):

    n = len(arr)

    # Traverse through all array elements

    for i in range(n):

        # Last i elements are already in place

        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1

            # Swap if the element found is greater

            # than the next element

            if arr[j] > arr[j+1] :

                arr[j], arr[j+1] = arr[j+1], arr[j]

Merge Sort: (Divide and Conquer)

# Python program for implementation of MergeSort

# Merges two subarrays of arr[].

# First subarray is arr[l..m]

# Second subarray is arr[m+1..r]

def merge(arr, l, m, r):

    n1 = m - l + 1

    n2 = r- m

    # create temp arrays

    L = [0] \* (n1)

    R = [0] \* (n2)

    # Copy data to temp arrays L[] and R[]

    for i in range(0 , n1):

        L[i] = arr[l + i]

    for j in range(0 , n2):

        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]

    i = 0     # Initial index of first subarray

    j = 0     # Initial index of second subarray

    k = l     # Initial index of merged subarray

    while i < n1 and j < n2 :

        if L[i] <= R[j]:

            arr[k] = L[i]

            i += 1

        else:

            arr[k] = R[j]

            j += 1

        k += 1

    # Copy the remaining elements of L[], if there

    # are any

    while i < n1:

        arr[k] = L[i]

        i += 1

        k += 1

    # Copy the remaining elements of R[], if there

    # are any

    while j < n2:

        arr[k] = R[j]

        j += 1

        k += 1

# l is for left index and r is right index of the

# sub-array of arr to be sorted

**def mergeSort(arr,l,r):**

**if l < r:**

**# Same as (l+r)/2, but avoids overflow for**

**# large l and h**

**m = (l+(r-1))/2**

**# Sort first and second halves**

**mergeSort(arr, l, m)**

**mergeSort(arr, m+1, r)**

**merge(arr, l, m, r)**

QuickSort

def partition(arr,low,high):

i = ( low-1 ) # index of smaller element

pivot = arr[high] # pivot

for j in range(low , high):

# If current element is smaller than or

# equal to pivot

if arr[j] <= pivot:

# increment index of smaller element

i = i+1

arr[i],arr[j] = arr[j],arr[i]

arr[i+1],arr[high] = arr[high],arr[i+1]

return ( i+1 )

# The main function that implements QuickSort

# arr[] --> Array to be sorted,

# low --> Starting index,

# high --> Ending index

# Function to do Quick sort

def quickSort(arr,low,high):

if low < high:

# pi is partitioning index, arr[p] is now

# at right place

pi = partition(arr,low,high)

# Separately sort elements before

# partition and after partition

quickSort(arr, low, pi-1)

quickSort(arr, pi+1, high)

1. Whether or not a in-place sorting algorithm, (talk about space complexity)
2. Maps and hashing (Look-up in constant time O(1))
3. A good example of a dictionary with (key, value) in which the value is another dictionary

locations = {'North America': {'USA': ['Mountain View']}}

locations['North America']['USA'].append('Atlanta')

locations['Asia'] = {'India': ['Bangalore']}

locations['Asia']['China'] = ['Shanghai']

locations['Africa'] = {'Egypt': ['Cairo']}

print 1

usa\_sorted = sorted(locations['North America']['USA'])

for city in usa\_sorted:

print city

print 2

asia\_cities = []

for countries, cities in locations['Asia'].iteritems():

city\_country = cities[0] + " - " + countries

asia\_cities.append(city\_country)

asia\_sorted = sorted(asia\_cities)

for city in asia\_sorted:

print city

1. Tree

**Example 1:**

**class** **BinaryTree**(object):

**def** **\_\_init\_\_**(self, root):

self.root = Node(root)

**def** **search**(self, find\_val):

**return** self.preorder\_search(tree.root, find\_val)

**def** **print\_tree**(self):

**return** self.preorder\_print(tree.root, "")[:-1]

**def** **preorder\_search**(self, start, find\_val):

**if** start:

**if** start.value == find\_val:

**return** **True**

**else**:

**return** self.preorder\_search(start.left, find\_val) **or** self.preorder\_search(start.right, find\_val)

**return** **False**

**def** **preorder\_print**(self, start, traversal):

**if** start:

traversal += (str(start.value) + "-")

traversal = self.preorder\_print(start.left, traversal)

traversal = self.preorder\_print(start.right, traversal)

**return** traversal

**Example 2**

**class** **BST**(object):

**def** **\_\_init\_\_**(self, root):

self.root = Node(root)

**def** **insert**(self, new\_val):

self.insert\_helper(self.root, new\_val)

**def** **insert\_helper**(self, current, new\_val):

**if** current.value < new\_val:

**if** current.right:

self.insert\_helper(current.right, new\_val)

**else**:

current.right = Node(new\_val)

**else**:

**if** current.left:

self.insert\_helper(current.left, new\_val)

**else**:

current.left = Node(new\_val)

**def** **search**(self, find\_val):

**return** self.search\_helper(self.root, find\_val)

**def** **search\_helper**(self, current, find\_val):

**if** current:

**if** current.value == find\_val:

**return** **True**

**elif** current.value < find\_val:

**return** self.search\_helper(current.right, find\_val)

**else**:

**return** self.search\_helper(current.left, find\_val)

**return** **False**

1. Graph

**def** **get\_edge\_list**(self):

edge\_list = []

**for** edge\_object **in** self.edges:

edge = (edge\_object.value, edge\_object.node\_from.value, edge\_object.node\_to.value)

edge\_list.append(edge)

**return** edge\_list

**def** **get\_adjacency\_list**(self):

max\_index = self.find\_max\_index()

adjacency\_list = [**None**] \* (max\_index + 1)

**for** edge\_object **in** self.edges:

**if** adjacency\_list[edge\_object.node\_from.value]:

adjacency\_list[edge\_object.node\_from.value].append((edge\_object.node\_to.value, edge\_object.value))

**else**:

adjacency\_list[edge\_object.node\_from.value] = [(edge\_object.node\_to.value, edge\_object.value)]

**return** adjacency\_list

**def** **get\_adjacency\_matrix**(self):

max\_index = self.find\_max\_index()

adjacency\_matrix = [[0 **for** i **in** range(max\_index + 1)] **for** j **in** range(max\_index + 1)]

**for** edge\_object **in** self.edges:

adjacency\_matrix[edge\_object.node\_from.value][edge\_object.node\_to.value] = edge\_object.value

**return** adjacency\_matrix

**def** **find\_max\_index**(self):

max\_index = -1

**if** len(self.nodes):

**for** node **in** self.nodes:

**if** node.value > max\_index:

max\_index = node.value

**return** max\_index