okeeffed Update most cheat sheets with a TOC 91856dc · 5 years ago History

1849 lines (1394 loc) · 55.7 KB

Data Structures with Python

    - 17.3: Binary Search
    - 17.4: Implementation of a Binary Search
    - 17.5: Hashing
    - 17.6: Implementation of a Hash Table
    - 17.7: Sorting Overview
    - 17.8: Visualizing Sorting Algorithms
    - 17.9: Implementing the Bubblesort Sort Method
    - 17.10: Implementing the Selection Sort
- [2, 3, 4, 10]
- [2, 3, 4, 10]
- [2, 3, 4, 10, 12, 35]
- output [2,2,4,5,7,11,12,23,56]
- quickSort recursively calls
- Vertex() - create a new vertice with an id and what it is connected to
- addNeighbour() - create a neighbour that it is connected to
- getWeight() returns the weight of the edge from this vertex
- Graph() - create new, empty graph
- addVertex(vert) - create new instance of a vertex
- addEdge(fromVert, toVert, weight)
- addEdge (fromtVert, toVert) - without weight
- getVertex(vertKey) - return vertex
- getVertices() - return list of all vertices
- in - returns True for a statement of the form vertex in graph, if the given vertex is in the graph, False otherwise
- gives back dict of vertices
- [0: , 1: ...]
- prints list with edge connected from 0 to 1

PYDS-12.0: Arrays

- Introduction to Arrays
- Low Level Arrays
- Dynamic Arrays and Amotization
- Array based "mini project"
- Several Array Interview Problems

Array Sequences

- List
- Tuple
- String

All of which support indexing.

PYDS-12.1: Low Level Arrays

- Focus on low level computer theory

Low-level comp architecture

- Memory stored in bits, stored in units called bytes
- Stores these bytes in an address
- Just as easy to retrieve or stored in O(1) time
- Arrays can be a contiguous portion of the computers memory - eg. String is consecutive - each location within the array is a *cell* - calcs done by start address + (cellsize * index)

Referential Arrays

- 100 student id names, each needs to have the same number of bytes. - We can use an array of object *References* - This helps the constant time access
- A single list instance may include multiple references to the same objects
- Single object can be an element of two or more lists - Changing the element reference to another point

Copying Arrays

- `backup = list(primes)`
- This produces a new list that is a `shallow copy` in that it references the same elements as in the first list. - If the contents of the list were of a mutable type, a `deep copy`, meaning a new list with new elements, can be produced by using the deepcopy function from the copy module
- counters = [0] * 8 - All 8 cells reference the same object! - We rely on the object being mutable - counters[2] += 1 does not change the value of the existing int instances - computes a new integer
- `primes.extend(extras)` will add the references to the first list

Review

- Basic computer architecture
- Low-level array representation
- Referential arrays

PYDS 12.2: Amotization

Using amortization, we can show that performing a sequence of such append operations on a dynamic array is actually quite efficient!

Amotized Anaylsis

1. Allocate memory for a larger array of size, typically twice the old array
2. Copy the contents of old array to new array
3. Free the old array

Once we hit a full array in items being asserted, we conclude an overflow and we implement the doubling.

With amortization, after we have continually doubled the size at an overflow, cost when we are not in overflow is a cost of 1 whereas the cost with inserting for overflow is the `n`.

```
Amortized Cost = ( 1 + 2 + 3 + 5 + 1 + 1 + 9 + 1 ... ) / n
                                = [(1 + 1 + 1 + ... ) + ( 1 + 2 + 4 +
...)]
                                = [n + 2n] / n
                                = 3
                                = O(1)
```

PYDS-13.0: Stacks, Queues and Deques

What is a stack?

- Ordered collection of items where additional and removal occur at the same end
- End is referred to as the "top"
- Opposite is the "base"
- Items near the base have been in the stack the longest
- Recently added are in position to be removed first - LIFO
- Fundamentally important as it can reverse the stack easily
- Similar to a list

Stack implementation

```
# Stack() creates a new empty stack
# push(item) add to stack
# pop() removes item from the top
# peek() shows you the top but does not remove
# isEmpty() bool
# size() return item size
```

```
class Stack(object):
        def __init__(self):
                self.items = []

        def isEmpty(self):
                return self.items == []

        def push(self, item):
                self.items.append(item)

        def pop(self):
                self.items.pop()

        def peek(self):
                return self.items[len(self.items)-1]

        def size(self):
                return len(self.items)

s = Stack()
print s.isEmpty()       # true
s.push('two')
s.peek()
s.push(True)
s.size()                        # 1
s.isEmpty()             # false
s.pop()                         # 'two'
```

PYDS-13.1: Queues Overview

What are Queues?

- Ordered collection of items where items addition happens at the end "rear"
- Removal happens from the "front"
- Item entered and waits in queue to be removed
- Longest item at the front - FIFO implementation
- "Enqueue" and "Dequeue" to the adding to the rear and removing the front
- "Push" and "pop" refers to the queue.

Queue Implementation

```
# Queue() to create a queue
# enqueue(item) to add to the rear
# dequeue() removes from the front
# isEmpty() is the bool
# size() returns the size

class Queue(object):
        def __init__(self):
```

```
                self.items = []

        def isEmpty(self):
                return self.items == []

        def enqueue(self, item):
                self.items.insert(0, item) # insert for FIFO

        def dequeue():
                return self.items.pop()

        def size():
                return len(self.items)

q = Queue()
q.size()                # 0
q.enqueue(1)
q.enqueue(2)
q.dequeue()     # 1
```

PYDS-13.2: Deques Overview

What is a deque?

- A deque is a double-ended queue
- Also has a front and an end and the items are position within the collection
- Unrestrictive nature for adding items - add to front OR rear!
- Same for removal
- Does not require LIFO/FIFO enforced data structure design

Implement a deque

```
# Deque() create a deque
# addFront(item)
# addRear(item)
# removeFront()
# removeRear()
# isEmpty()
# size()

class Deque(object):
        def __init__:
                self.items = []

        def isEmpty(self):
                return self.items == []

        # rear is the first index
        def addRear(self, item):
```

```
                self.items.insert(0, item)

        # front is the len(self.items) index
        def addFront(self, item):
                self.items.append(item)

        def removeFront(self):
                return self.items.pop()

        def removeRear(self):
                return self.items.pop(0)

        def size(self):
                return len(self.items)

d = Deque()
d.addFront('hello')
d.addRear('world')
d.size()
print d.removeFront() + ' ' + d.removeRear()    # 'hello world'
d.size()
```

PYDS-14.0: Singly Linked Lists

What is a singly linked list?

- Singly Linked List is a collection of nodes that form a linear sequence
- Each node stores a reference to the next node
- The first and last node of the list are known as the "head" and the "tail" of the list
- Process of moving through the list is "traversing"
- Each node stores a reference to the element and the next node (except the tail)
- How do we add a new element?
- Example to append to the Head (inverse can be done for appending to the Tail) - We create a new node - Set its element to the new element - Set the next link to refer to the current head - Set the list's head to point to the new node
- Removing an element from the Head is essentially the reverse operation to adding the item
- We cannot easily remove the last node - to do so efficiently requires a `doubly linked list`
- O(k) time to access elements
- Constant time insertions and deletions in any position, arrays require O(n) time
- Linked Lists can expand without having to specify their size ahead of time!

Implementation of a singly linked list

```
class Node(object):
        def __init__(self, value):
                self.value = value
                self.nextNode = None

a = Node(1)
b = Node(2)
c = Node(3)

# how to link the nodes?
a.nextNode = b
b.nextNode = c
```

PYDS-14.1: Doubly Linked Lists

What is a doubly linked list?

- `next` and `prev` for references to nodes that are both next and what precedes it
- "dummy" nodes are known as the header sentinel and trailer sentinel for both the beginning and end of a list respectively
- Each insertion happens between a pair of existing nodes - eg. Add between header and what is after to add to the front

Implementation of a Doubly Linked List

```
class Node(object):
        def __init__(self, value):
                self.value = value
                self.nextNode = None
                self.prevNode = None

a = Node(1)
b = Node(2)
c = Node(3)

a.nextNode = b
b.prevNode = a
b.nextNode = c
c.prevNode = b
```

PYDS-15.0: Recursion

What is recursion?

- Two instances - First when recursion is used a technique in which a func makes one or more calls to itself - Second is when data structures use smaller instances of the

exact type of DS when it represents itself
- Powerful alternative to repetitive tasks in which a loop is not ideal
- Great tool for building out particular DS

PYDS-15.1: Memoization

- Remembers results of method calls based on the method inputs and then remembering them again.

PYDS-16.0: Trees

Tree Section

1. Tree Data Structures
2. Implementing with Lists
3. Implement with OOP
4. Implemenet with priority queue
5. Only covers ADT (Abstract Data Types)

What are trees?

- Has a root, branches and leaves
- Root at the top, leaves at the bottom
- Children of one node are independent of children of another
- Each leaf node is unique
- File systems are structured as a tree
- Consists of a set of nodes and edges that connect pairs of nodes
- Trees that have a max of two children are referred to as a *binary tree*

Nodes in the tree

- Can have a name "key"
- May also have additional "payload" info
- One incoming edge, 0-to-many outgoing
- Path: Order list of nodes connected by edges
- Level "n" refers to number of edges from the root node
- Height of the tree is maxHeight(Tree)

Recursive Definition of a tree

- either empty or consists of a root and zero or more subtrees which are also a tree
- the root of each subtree is connected to the root of the parent by an edge

PYDS-16.1: Implementing a Tree as a List of Lists

- Store value of root node as first element
- Second element will be a list that represents the left subtree
- Third element will be a list of another list representing the right subtree

```
# what we are aiming for
myTree = ['a',          # root
                ['b',           # left subtree
                        ['d', [], []],
                        ['e', [], []],
                ],
                ['c', [], []]   # right subtree
]
```

Implementing a Tree using a List of Lists

```
# define the style of tree with a root and left/right empty child
lists
def BinaryTree(r):
        return [r, [], []]

def insertLeft(root, newBranch):
        t = root.pop(1)

        if len(t) > 1:
                root.insert(1, [newBranch, t, []])
        else:
                root.insert(1, [newBranch], [], [])
        return root

def insertRight(root, newBranch):
        t = root.pop(1)

        if len(t) > 1:
                # reordered compared to above
                root.insert(2, [newBranch, [], t)
        else:
                root.insert(2, [newBranch], [], [])
        return root

def getRootVal(root):
        return root[0]

def setRootVal(root, newVal):
        root[0] = newVal

def getLeftChild(root):
        return root[1]
```

```
def getRightChild(root):
        return root[2]

r = BinaryTree(3)
insertLeft(r, 4)
# [3, [4, [], []], []]
insertLeft(r, 5)
# [3, [5 [4, [], []], []], []]
insertRight(r, 6)
# [3, [5 [4, [], []], []], [6, [], []]]
insertRight(r, 7)
# [3, [5 [4, [], []], []], [7, [], [6, [], []]]]
l = getLeftChild(r)
print l
# [5 [4, [], []], []]
setRootVal(1, 9)
# [3, [9 [4, [], []], []], [7, [], [6, [], []]]]
```

PYDS-16.1: Node and Node References Implementation

- In this case, define a class that has attributes for the root value as well as left and right subtrees
- Since rep more closely follows OOP, we will continue with this representation

```
class BinaryTree(object):
        def __init__(self, rootObj):
                self.key = rootObj
                self.leftChild = None
                self.rightChild = None

        def insertLeft(self, newNode):
                if self.leftChild == None:
                        self.leftChild = BinaryTree(newNode)
                else:
                        t = BinaryTree(newNode)
                        t.leftChild = self.leftChild
                        self.leftChild = t

        def insertRight(self, newNode):
                if self.rightChild == None:
                        self.rightChild = BinaryTree(newNode)
                else:
                        t = BinaryTree(newNode)
                        t.rightChild(self.rightChild)
                        self.rightChild=(t)

        # bring back Object Address values
        def getRightChild(self):
                return self.rightChild
```

```
        def getLeftChild(self):
                return self.leftChild

        def setRootVal(self, obj):
                self.key = obj

        def getRootVal(self):
                return self.key

r = BinaryTree('a')
r.getRootVal()
# 'a'
print r.getLeftChild()
# None
r.insertLeft('b')
r.getLeftChild()
# get address of another binary tree
r.getLeftChild().getRootVal()
# 'b'
```

PYDS-16.2: Tree Traversals

3 Main Methods

1. Preorder
2. Inorder
3. Postorder

- Commonly used patterns
- Difference is the order in which nodes are visited
- Preorder - We visit the root node first, before a recursive preorder traversal of the left subtree followed by the same for the right subtree
- Inorder - We recursively do an inorder traversal of left subtree, then visit the root node, then a recusive inorder traversal of the right subtree
- Postorder - Recursively postorder traversal of the left subtree and the right subtree followed by a visit to the root node

How to use "Preorder"

- Think of a tree with a Book as the root, Chapters 1 and 2 as the children and sections as the children of the chapters
- Preorder can "read it" from front to book
- Read the "Book" node, then recusively go down the left child eg. Chapter One and each recursive left subtree from there

Preorder implementation

- Base case to check if tree exists
- If parameter is None, then the function returns without taking any action
- This can be implemented as a method of the `BinaryTree` class - Must check for the existence of the left and the right children before making the recursive call to preorder - In this case, probably better implementing it as an external function - The reason is that you rarely just want to traverse the tree - Most cases you want to accomplish something else during traversal

```
def preorder(tree):
        if tree != None:
                print(tree.getRootVal())
                preorder(tree.getLeftChild())
                preorder(tree.getRightChild())

# implementation as a BinaryTree method
# generally not what you will want to do
def preorder(self):
        print(self.key)
        if self.leftChild:
                self.leftChild.preorder()
        if self.rightChild:
                self.rightChild.preorder()
```

Postorder Implementation

- Nearly identical to preorder except that we move the call to print to the end

```
def postorder(tree):
        if tree != None:
                preorder(tree.getLeftChild())
                preorder(tree.getRightChild())
                print(tree.getRootVal())
```

Inorder Implementation

- In inorder traversal we visit the left subtree, followed by the root and finally the right subtree
- Notice that in all three of the traversal functions we are simply changing the position of the print statement with respect to the two recursive function calls

```
def inorder(tree):
        if tree != None:
                inorder(tree.getLeftChild())
                print(tree.getRootVal())
                inorder(tree.getRightChild())
```

PYDS-16.3: Priority Queues with Binary Heaps

What are Binary Queues?

- One important variation of a queue is called a *Priority Queue*
- A priority queue acts like a queue in that you dequeue an item by removing it from the front
- However, the logical order of items inside a queue is determined by their priority
- The classic way to implement this is using a *Binary Heap*
- Binary heap allows us both enqueue and dequeue items in O(log n) time!

What are Binary Heaps

- Two common variations - "min heap": the smallest key is always at the front - "max heap": in which the largest key value is always at the front

Implementing a Binary Heap

- to make the heap work efficiently, we use logarithmic nature of binary tree
- must keep the tree balanced - we do this by creating a *complete binary tree* - if we know it is a complete list, we can find the parent/child relationship `2p` and `2p+1` - we can use this to make an efficient implementation of the tree - index 0 is set as 0 and then that math operation works
- heap will init with one element 0, but the current size will be 0
- most efficient way is to append to the list - likely violate the heap structure property by comparing with the parent - if new item is less than parent, we can swap the parent and child and repeat!

```
# BinaryHeap() - create new heap
# insert(k) - adds a new item to the heap
# findMin() - returns the item with the minimum key value, leaving item
# delMin() - returns the item with the minimum key value, removing item
        # requires that we take the last position and set it to the root
        # restore order by pushing down new root node
        # swap new root with smallest child recursively
# isEmpty()
# size
# buildHeap(list) builds a new heap from a list of keys
        # we can build a heap in O(n) operations

class BinaryHeap(object):
        def __init__(self):
                self.heapList = [0]
                self.currentSize = 0

        def percUp(self, i):
                while i // 2 > 0:
```

```
                if self.heapList[i] < self.heapList[i // 2]:
                        tmp = self.heapList[i//2]
                        self.heapList[i//2] = self.heapList[i]
                        self.heapList[i] = tmp
                i = i // 2

        def insert(self, k):
                self.heapList.append(k)
                self.currentSize = self.currentSize + 1
                self.percUp(self.currentSize)

        def percDown(self, i):
                while (i * 2) <= self.currentSize:
                        mc = self.minChild(i)
                        if self.heapList[i] > self.heapList[mc]:
                                tmp = self.heapList[i]
                                self.heapList[i] = self.heapList[mc]
                                self.heapList[mc] = tmp
                        i = mc

        def minChild(self, i):
                if i * 2 + 1 > self.currentSize:
                        return i * 2
                else:
                        if self.heapList[i*2] < self.heapList[i*2+1]:
                                return i * 2
                        else:
                                return i * 2 + 1

        def delMin(self):
                retVal = self.heapList[1]
                self.heapList[1] = self.heapList[self.currentSize]
                self.currentSize = self.currentSize - 1
                self.heapList.pop()
                self.percDown(1)
                return retVal

        def buildHeap(self, aList):
                i = len(aList) // 2
                self.currentSize = len(aList)
                self.heapList = [0] + aList[:]
                while (i > 0):
                        self.percDown(i)
                        i = i - 1
```

More on Heap Operations

Generally implemented as an array due t the nature of accessing children.

---- Inserting an element (up-heap, percolate-up)

1. Add element to the bottom level of the heap
2. Compare the added element with its parent; if they are correct, stop
3. If not, swap the element with its parent and return to the previous step

We do not need to check the other child in the end, as the `transistive relation` will ensure that it holds.

---- Extract (down-heap, percolate-down)

The procedure for deleting the root form the heap.

1. Replace the root of the heap with the last element on the last level
2. Compare the new root with its children; if the order is correct, stop
3. If not, swap the element with one of its children and return to the previous step (swap with smaller child in min-heap and larger child in max-heap) - always swap the child that has the larger and correct difference

PYDS-16.4: Binary Search Trees

- We've seen two different ways to get key-value pairs in a collection
- These collections implement the `map abstract data type`
- Two implementations talked about so far were binary search on a list and hash tables.

Implementation of Binary Search Trees

- relies on the property that keys that are less than the parent are found in the left subtree, while those greater are in the right subtree. - left/right subtree implementation is the `bst` property - refers just to direct parent - understand when something becomes to left and right tree
- implementation will use two classes! *BinarySearchTree* and *TreeNode* - since we need to be able to create and work with an empty tree - *BinarySearchTree* has a reference to the *TreeNode* that is the root of the binary search tree

```
# order of insert data = [70,31,93,94,14,23,73]
# put moethod – check if tree already has a root
        # if not, create a new TreeNode and install it as the root of th
        # if root node in place, call private, helper function put
                # start at root of tree, search tree comparing new key t
                # if less, search left, if more, search right

# get method is easier since it searches tree recursively until it gets
```

```
        # when matching key found, the value stored in the payload of th

# delete is more difficult
        # if tree has more than one node, we search using the _get metho
        # if single node, remove root but must check if root key == para
        # if we find node, 3 options to consider
                # does node to delete have children?
                        # remove reference to parent -> set [left|right]
                #  does node to delete have single child?
                        # slightly more complex -> promote child to take

class BinarySearchTree:
        def __init__(self):
                self.root = None
                self.size = 0

        def length(self):
                return self.size

        # allows to call len(<BST object>)
        def __len__(self):
                return self.size

        def __iter__(self):
                return self.root.__iter__()

        def _put(self, key, val, currentNode):
                if key < currentNode.key:
                        if currentNode.hasLeftChild():
                                self._put(key,val,currentNode.leftChild)
                        else:
                                currentNode.leftChild = TreeNode(key,val
                else:
                        if currentNode.hasRightChild():
                                self._put(key,val,currentNode.rightChild
                        else:
                                currentNode.rightChild = TreeNode(key, v

        def __setitem__(self, k, v):
                self.put(k,v)

        def put(self, key, val):
                if self.root:
                        self._put(key, val, self.root)
                else:
                        self.root = TreeNode(key, val)
                self.size = self.size + 1

        # _ for code refactoring reasons as help
        def _put(self, key, val, currentNode):
                if key < currentNode.key:
                        if currentNode.hasLeftChild():
                                self._put(ley, val, currentNode.leftChil
```

```
                else:
                        currentNode.leftChild = TreeNode(key, va
        else:
                if currentNode.hasRightChild():
                        self._put(key, val, currentNode.rightChi
                else:
                        currentNode.rightChild = TreeNode(key, v

def get(self, key):
        if self.root:
                res = self._get(key,self.root)
                if res:
                        return res.payload
                else:
                        return None
        else:
                return None

def _get(self, key, currentNode):
        if not currentNode:
                return None
        elif currentNode.key == key:
                return currentNode
        elif key < currentNode.key:
                return self._get(key, currentNode.leftChild)
        else:
                return self._get(key, currentNode.rightChild)

def __getitem__(self, key):
        return self.get(key)

def __contains__(self, key):
        if self._get(key, self.root):
                return True
        else:
                return False

def delete(self, key):
        if self.size > 1:
                nodeToRemove = self._get(key, self.root)
                if nodeToRemove:
                        self.remove(nodeToRemove)
                        self.size = self.size-1
                else:
                        raise KeyError('Error, key not in tree')
        elif self.size == 1 and self.root.key == key:
                self.root = None
                self.size = self.size - 1
        else:
                raise KeyError('Error, key not in tree')

def __delitem__(self, key):
        self.delete(key)
```

```
    def spliceOut(self):
        if self.isLeaf():
            if self.isLeftChild():
                self.parent.leftChild = None
            else:
                self.parent.rightChild = None
        elif self.hasAnyChildren():
            if self.hasLeftChild():
                if self.isLeftChild():
                    self.parent.leftChild = self.lef
                else:
                    self.parent.rightChild = self.le
                    self.leftChild.parent = self.par
        else:
            if self.isLeftChild():
                self.parent.leftChild = self.rightChild
            else:
                self.parent.rightChild = self.rightChild
                self.rightChild.parent = self.parent

    def findSuccessor(self):
        succ = None
        if self.hasRightChild():
            succ = self.rightChild.findMin()
        else:
            if self.parent:
                if self.isLeftChild():
                    succ = self.parent
                else:
                    self.parent.rightChild = None
                    succ = self.parent.findSuccessor
                    self.parent.rightChild = self
        return succ

    def findMin(self):
        current = self
        while current.hasLeftChild():
            current = current.leftChild
        return current

    def remove(self, currentNode):
        if currentNode.isLeft(): #Leaf
            if currentNode == currentNode.parent.leftChild:
                currentNode.parent.leftChild = None
            else:
                currentNode.parent.rightChild = None
        elif currentNode.hasBothChildren(): #interior
            succ = currentNode.findSuccessor()
            succ.spliceOut()
            currentNode.key = succ.key
            currentNode.payload = succ.payload
        else: # this node has one child
```

```
                        if currentNode.hasLeftChild():
                                if currentNode.isLeftChild():
                                        currentNode.leftChild.parent = c
                                        currentNode.parent.leftChild = c
                                elif currentNode.isRightChild():
                                        currentNode.leftChild.parent = c
                                        currentNode.parent.rightChild =
                                else:
                                        currentNode.replaceNodeData(curr
                                                        currentNode.left
                                                        currentNode.left
                                                        currentNode.left
                        else:
                                if currentNode.isLeftChild():
                                        currentNode.rightChild.parent =
                                        currentNode.parent.leftChild = c
                                elif currentNode.isRightChild():
                                        currentNode.rightChild.parent =
                                        currentNode.parent.rightChild =
                                else:
                                        currentNode.replaceNodeData(curr
                                                        currentNode.righ
                                                        currentNode.righ
                                                        currentNode.righ

class TreeNode:
        def __init__(self, key, val, left=None, right=None, parent=None)
                self.key=key
                self.val=val
                self.left=left
                self.right=right
                self.parent=parent

        def hasLeftChild(self):
                return self.leftChild

        def hasRightChild(self):
                return self.rightChild

        def isLeftChild(self):
                return self.parent and self.parent.leftChild == self

        def isRightChild(self):
                return self.parent and self.parent.rightChild == self

        def isRoot(self):
                return not self.parent

        def isLeaf(self):
                return not (self.rightChild or self.leftChild)

        def hasAnyChildren(self):
                return self.rightChild or self.leftChild
```

```
        def hasBothChildren(self):
                return self.rightChild and self.leftChild

        def replaceNodeData(self, key, value, lc, rc):
                self.key = key
                self.payload = value
                self.leftChild = lc
                self.rightChild = rc
                if self.hasLeftChild():
                        self.leftChild.parent = self
                if self.hasRightChild():
                        self.rightChild.parent = self
```

Notes:

- Deletion is one of the more difficult things we can do for a binary search tree
- If both children are present, then we need to decide on a successor
- The successor is guaranteed to have no more than one child, so we know how to remove it using the two cases for deletion that we have already implemented
- Once the successor has been removed, we simply put it in the tree in place of the node to be deleted
- This is an inorder traversal from largest to smallest - If right child, the successor is the findMin() of that right subtree - If no right child, then successor is the parent
- Remember, left most child will be the smallest of a BST
- The iterator method itself takes a bit more work - yield keyword freezes the state of the function - iterators vs generators in Python

PYDS-16.5: Common Tree Questions

Given a binary tree, check whether it's a binary search tree or not.

- Tree traversal should lead to sorted order
- Another solution is to keep track of the min and max values a node can take

```
# Solution 1
tree_vals = []
# traversal should lead to sorted order
def inorder(tree):
        if tree != None:
                inorder(tree.getLeftChild())
                treeVals.append(tree.getRootVal())
                inorder(tree.getRightChild())

def sortCheck(treeVals):
        return treeVals == sorted(treeVals)
```

```
inorder(tree)
sortCheck(treeVals)

# Solution 2
# class Node
        # def treeMax - if not node, return -inf
        # def tree Min - not node, return inf
        # def verify
                - if not node, return True
                - if tree.max and tree.min and verify(node.right) return
                - else return False
```

17.0: Searching and Sorting

- First half is search
- Second half is sorting

Note: In python, we can use `in` to check if an element is in a list.

```
15 in [5, 4, 15] # true
```

How does this work? What is the best way to search?

17.1: Sequential Search

Basic searching technique. Sequentially go through a data subject and compare as you go along.

Eg. traversing an unordered list for 50 and comparing as you go.

If 50 was not present, we still had to check every element in the array.

But what if it was ordered?

If the array was sorted, then we only have to search until we get a match or find something greater than our search target.

Ordered vs Unordered sequential search

Average time for unordered will be `n`, whereas for ordered it will be `n/2`

17.2: Implementation of a Sequential Search

Unordered List

```
def seqSearch(arr, el):
        pos = 0
        found = False

        while pos < len(arr) and not found:
                if arr[pos] == el:
                        found = True
                else:
                        pos += 1

        return found
```

Ordered List

```
def orderedSeqSearch(arr, el):
        pos = 0
        found = False
        stopped = False

        while pos < len(arr) and not found and not stopped:
                if arr[pos] == el:
                        found = True
                else:
                        if arr[pos] > el:
                                stopped = True
                        else:
                                pos += 1

        return found
```

17.3: Binary Search

If the list is ordered, we can do a binary search!

This item starts from the middle.

If the item is greater, we know the entire lower half of the list can be ignore.

Then, we can repeat the process with the upper half.

So it uses `Divide and Conquer` - divide to smaller pieces, solve the smaller pieces and then repeat.

- Each comparison eliminates about half of the remaining items from consideration

- What is the maximum number of comparisons this algorithm will require to check the entire list?

```
// Comparisons

1: n/2 comparisons left
2: n/4 comparisons left
3: n/8 comparisons left
...
i: n/2^i comparisons left
```

17.4: Implementation of a Binary Search

2 Versions - iterative and recusive

Iterative

```
def binarySearch(arr, el):
        first = 0
        last = len(arr)-1

        found = False

        while first <= last and not found:
                mid = (first+last)/2
                if arr[mid] == el:
                        found = True
                else:
                        if el < arr[mid]:
                                last = mid-1
                        else:
                                first = mid+1
        return found

arr = [1,2,3,4...] # must be sorted

binarySearch(arr, 4) # True
binarySeach(arr, 13) # False
```

Recursive

Remember: Must always have a base case!

```
def recBinSearch(arr, el):
        if len(arr) == 0:
                return False
        else:
                mid = len(arr)/2
```

```
    if arr[mid] == el:
            return True
    else:
            if el < arr[mid]:
                    return recBinSearch(arr[:mid], el)
            else:
                    return recBinSearch(arr[mid+1:], el])
```

17.5: Hashing

- Hashing
- Hash Tables
- Hash Functions
- Collision Resolution
- Implementing a Hash Table

Hashing

We've seen how we can improve search by knowing about structures beforehand.

We can build a data structure that can be accessed in O(1) time - this is hashing!

A `hash table` is a collection of items that are stored in such a way that it becomes easy to find them later.

- Each position of the hash table, `slots` , can hold an item and is named by an integer value starting at 0.
- For example, we will have a slot named 0, a slot named 1, a slot named 2 and so on.
- Initially, the hash table contains no items so every slot is empty

Hash Tables

The mapping between an item and the slot were that item belongs in the hash table is called the `hash function`

- The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1.
- So how should we use hash functions to map items to slots?

One `hash function` we can use is the remainder method. When preseted with an item, the hash function is the item divided by the table size, this is then its slot number.

Example:

- Assume we have 54, 26, 93, 17, 77 and 31
- We've preassigned an empty hash table of m=11

- Our remainder hash function then is `h(item)=item%11`

Let's see the results!

```
Item    | Hash Value
--------------------
54              | 10
26              | 4
93              | 5
17              | 6
77              | 0
31              | 9
```

We are now ready to occupy 6 out of the 11 slots

- This is referred to as the `load factor` and is commonly denoted by `lambda = number of items / table size`
- Here we have `lambda = 6/11`

Our Hash Table has now been loaded.

When we want to search, we just need to use the hash function to compute the slot name for the item and then chack the hash table to see if it is present.

Therefore the operation is O(1), since a constant amount o time is required to compute the hash value and then index the hash table at that location.

What if we have two items that have the same location? This is known as a `collision` (sometimes a `clash`).

We can talk about this resolution soon.

Hash Function

A hash func that maps each item into a unique slot is referred to as a `perfect hash function` .

Our goal is to create a hash func that minimizes collisions, so it is easy to compute and evenly distributes the items in the hash table.

Folding Method

This method for constructing hash functions begins by dividing the item into equal-size pieces (last piece may not be of equal size).

These pieces are then added together to give the resulting hash value.

Example:

Give the number `436-555-4601` - we can divide these numbers into groups of two.

After add the numbers now, we get 210.

If we assume our hash table has 11 slots, we need to perform the extra step of dividing by 11 and keeping its remainder.

210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1.

Mid-square method

We first square the item, and then extract some portion of the resulting digits.

Example, if it were 44, we computer 44^2 = 1936.

By extracting the middle two digits, 93, and performing the remainder step, we get 93%11 = 5

Non-Integer elements We can also create hash funcs for character-based items.

The word `cat` can be thought of as a sequence of ordinal values.

If you use Python, you can just use the function `ord('a')` and then get the values.

So `ord('a')` = 97 % 11 = 11

For cat, you can sum up all the ordinal values, get 312 % 11 = 4.

Collision Resolution

One method for resolving it to look into the hash table and find another open slot to hold the item that caused the collision. This is known as `Open Addressing` .

By systematically visiting every spot, we are doing something known as `linear probing` .

Example if we had 77, 44, and 55, we then move 44 and 55 up until it finds something that fits.

One way to deal with clustering is to skip slots, thereby more evenly distributing the items that have caused collisions.

`rehashing` is the general process of looking for another slot.

`quadratic probing` is a variation of the linear probing idea.

Instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9 and so on.

- If the first have value is `h` , successive values are `h+1` , `h+4` , `h+9` , `h+16`

Alternative Option

We can also allow each slot to hold a reference to a collection (or chain) or items.

`Chaining` allows many items to exist at the same location in the hash table.

When collisions happen, the item is still placed in the proper slot of the hash table.

As more items hash to the same location, the difficulty for finding the item also increases.

17.6: Implementation of a Hash Table

Map

The idea of a dictionary used as a hash table to get and retrieve items using `keys` is often referred to as a mapping.

```
# HashTable() - Create a new, empty map. It returns an empty map collecti
# put(key, val) - Add a new key-value pair to the map. If the key is alr
# get(key) - Give a key, return the value stored in the map or None
# del - Delete the key-value pair from the map using a statement of the
# len() - Return the number of key-value pairs stored
# in the map Return True for a statement of the form `key in map`, if th

#
# Note - you can take this and play around with other hash functions
#

class HashTable(object):
        def __init__(self, size):
                self.size = size
                self.slots = [None] * self.size         # list with an e
                self.data = [None] * self.size

        def put(self, key, data):
                hashvalue = self.hashfunction(key, len(self.slots))
                if self.slots[hashvalue] == None:
                        self.slots[hashvalue] = key
                        self.data[hashvalue] = data
                else:
                        if self.slots[hashvalue] == key:
                                self.data[hashvalue] = data
                        else:
                                nextslot = self.rehash(hashvalue, len(se

                                while self.slots[nextslot] != None and s
                                        nextslot = self.rehash(nextslot,

                                if self.slots[nextslot] == None:
                                        self.slots[nextslot] = key
                                        self.data[nextslot] = data

                                else:
```

```
                                        self.data[nextslot] = data

        def hashfunction(self, key, size):
                return key%size

        def rehash(self, oldhash, size):
                return (oldhash+1)%size

        def get(self,key):
                startslot = self.hashfunction(key, len(self.slots))
                data = None
                stop = False
                found = False
                position = startslot

                while self.slots[position] != None and not found and not
                        if self.slots[position] == key:
                                found = True
                                data = self.slots[position]
                        else:
                                position = self.rehash(position, len(sel
                                if position == startslot:
                                        stop = True
                return data

        def __getitem__(self, key):
                return self.get(key)

        def __setitem__(self, key, data):
                self.put(key, data)

h = HashTable(5)
h[1] = 'one'
h[2] = 'two'
h[3] = 'three'
h[1] # 'one'
h[2] # 'two'
```

17.7: Sorting Overview

We've discussed how to search for items, but now we will look at how to sort!

Explanations and Implementations:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Shell Sort

5. Merge Sort
6. Quick Sort

Common interview questions consist of being asked to implement a sorting algorithm.

The best way to understand algorithms is to understand two things:

1. The underlying principle behind the algorithm
2. What a simple visualization of what the algorithm looks like

17.8: Visualizing Sorting Algorithms

Examples for seeing them:

Sorting Algorithms website Visualgo

17.9: Implementing the Bubblesort Sort Method

The `bubble sort` makes multiple passes through a list

- It compares adjacent items and exchanges those that are out of order
- Each pass through the list places the next largest value in its proper place
- Each item "bubbles" up to the location where it belongs

In the visualization of the sort, imagine the comparision beside each other to see if they are out of order.

If so, you swap and repeat (exchange), if not (no exchange), set the element to compare to the next number.

There can be multiple passes.

```
def bubbleSort(arr):
        for n in range(len(arr)-1, 0, -1):
                # print n
                for k in range(n):
                        # print k
                        if arr[k] > arr[k+1]:
                                temp = arr[k]
                                arr[k] = arr[k+1]
                                arr[k+1] = temp

arr = [5, 3, 7, 2]
bubbleSort(arr)
arr                     # [2 3, 5, 7]
```

17.10: Implementing the Selection Sort

Selection sort improves on the bubble sort by making only one exchange for every pass through the list.

It looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location.

The process continues and requires `n-1` passes to sort `n` items since the final item is placed on the nth pass.

Example: [10, 4, 3, 2]

We notice 10 is the largest, so we swap last place with it. Then 4, so second last place. Repeat.

Resources:

[cs.armstrong]

Implementation of the Selection Sort

```python def selectionSort(arr): for fillslot in range(len(arr)-1, 0, -1): positionOfMax = 0

```
        for location in range(1, fillslot+1):
                if arr[location] > arr[positionOfMax]:
                        positionOfMax = location

        temp = arr[fillslot]
        arr[fillslot] = arr[positionOfMax]
        arr[positionOfMax] = temp
```

arr = [10, 4, 3, 2] selectionSort(arr)

# [2, 3, 4, 10]

```
17.11: Insertion Sort

Always maintains a sorted sublist in the lower positions of the list.

Each new item is inserted back into the previous sublist such that the
sorted sublist is one item larger.

- Begin by assuming list with one item (position 0) is already sorted.
- On each pass pass, one for each item 1 through n-1, the current item
is checked against those in the already sorted sublist.
- As we look back into the already sorted sublist, we shift those
```
```

```
items that are greater to the right.
- When we reach a smaller item or the end of the sublist, the current
item can be inserted.

Insertion sort builds the final sorted array one item at a time. It is
much less effecient on large lists than more advanced algorithms such
as quicksort, heapsort or merge sort.

This runs in O(n^2) but a best case costs 1 on each pass. This
requires a third of the processing power though!

**Implementation of the Insertion Sort**

```python
def insertionSort(arr):
        for i in range(1, len(arr)):
                currentValue = arr[i]
                position = i

                while position > 0 and arr[position-1] > currentValue:
                        arr[position] = arr[position-1]
                        position = position-1

                arr[position] = currentValue

arr = [10, 4, 3, 2]
insertionSort(arr)
# [2, 3, 4, 10]
```

17.12: Shell Sort

Shell sort improves on the insertion sort by breaking the original list into a number of smaller sublists.

The unique way that these sublists are chosen is the key to the shell sort.

Instead of breaking lists into contiguous sublists, shell sort uses an increment `i` to create a sublist by choose all items that are `i` items apart.

If we have 9 items and we have an increment of 3, each n+3 form the sublist.

After completing these sublists, we've moved these closer to where they belong.

Then if we do the final sort with an increment of one (so in this case, just a standard insertion sort).

Implementation of Shell Sort
```

```
def shellSort(arr):
        sublistCount = len(arr)/2

        # While we still have sub lists
        while sublistCount > 0:
                for start in range(sublistCount):
                        # Use a gap insertion
                        gapInsertionSort(arr, start, sublistCount)

                print 'After increments of size: ', sublistCount
                print 'Current array: ', arr
                sublistCount = sublistCount / 2

def gapInsertionSort(arr, start, gap):
        for i in range(start+gap, len(arr), gap):
                currentValue = arr[i]
                position = i

                # Using the gap
                while position >= gap and arr[position - gap] > currentV
                        arr[position] = arr[position - gap]
                        position = position - gap

                # Set current value
                arr[position] = currentValue

arr = [10, 4, 3, 2, 12, 35]
shellSort(arr)
# [2, 3, 4, 10, 12, 35]
```

17.12: Merge Sort

Merge sort is a recursive algorithm that continual splits a list a half. Going with a `divide and conquer` strategy.

If the list is empty or has one item, it is sorted by definition (the base case).

If the list has more than one item, we split the list and recursively invoke a merge sort on both halves.

- Once the two halves are sorted, the fundamental operation, called a `merge` is performed.
- Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list.
- Continually `split` and at the end, recusively merge!

Implementation of a Merge Sort

```
def mergesort(arr):
        if len(arr) > 1:
                mid = len(arr) / 2
                lefthalf = arr[:mid]
                righthalf = arr[mid:]

                mergesort(lefthalf)
                mergesort(righthalf)

                i = 0
                j = 0
                k = 0

                while i < len(lefthalf) and j < len(righthalf):
                        if lefthalf[i] < righthalf[i]:
                                arr[k] = lefthalf[i]

                                i += 1

                        else:
                                arr[k] = righthalf[j]
                                j += 1

                        k += 1

                while i < len(lefthalf):
                        arr[k] = lefthalf[i]
                        i += 1
                        k += 1

                while j < len(righthalf):
                        arr[k] = righthalf[j]
                        j += 1
                        k += 1
        print 'Merging', arr

arr = [11,2,5,4,7,56,2,12,23]
mergesort(arr)
# output [2,2,4,5,7,11,12,23,56]
```

17.13: Quick Sort

- First selects a value, called the `pivot value` and there will be a `left mark` and `right mark`
- The role of the pivot value is to assist with splitting the list
- After selecting the `pivot value`, the `partition` process happens next - It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value

- The actual position where the pivot value belongs in the final sorted list, commonly called the `split point`, will be used to divide the list for subsequent calls to the quick sort
- `left mark` will check if greater than `pivot value` and `right mark` will check if less than
- Once decided, switch the marks
- When the `right mark` is < than the `left mark` on the array, we call this the `split point` - Once crossed, we swap right mark with the `pivot value`

Implementation of the Quick Sort

You can choose different pivot values, but this implementation will choose the first item in the list.

```
def quickSort(arr):
        quickSortHelper(arr, 0, len(arr-1))

# quickSort recursively calls
def quickSortHelper(arr, first, last):
        if first < last:
                splitPoint = partition(arr, first, last)

                quickSortHelper(arr, first, splitPoint-1)
                quickSortHelper(arr, splitPoint+1, last)

def partition(arr, first, last):
        pivotValue = arr[first]
        leftmark = first+1
        rightmark = last

        done = False

        while not done:
                while leftmark <= rightmark and arr[leftmark] <= pivotVa
                        leftmark += 1
                while arr[rightmark] >= pivotValue and rightmark >= lefti
                        rightmark -= 1
                if rightmark < leftmark:
                        done = True
                else:
                        temp = arr[leftmark]
                        arr[leftmark] = arr[rightmark]
                        arr[rightmark] = temp

        temp = arr[first]
        arr[first] = arr[rightmark]
        arr[rightmark] = temp
```

```
        return rightmark
```

18.0: Graphs

- Learn about graphs
- Implement the graph abstract data type using multiple internal representations
- See how graphs solve a wide variety of problems
- With a good graph implementation, we can then use these to solve problems which at first see difficult
- Vertices and Edges - Edges can be directed (one-way/digraph) or two-way - Edges can be weighted
- Cycle in a directed graph is a path that starts and ends at the same vertex - No cycles is called `acyclic` - we will see that we can solve several important problems if the problem can be represented as a `directed acyclic graph`

18.1: Adjacency Matrices and Lists

Adjacency Matrix

- One of the easiest ways to implement a graph is to use a two-dimensional matrix
- In this matrix implementation, each of the rows and columns represent a vertex in the graph
- If two vertices are connected by an edge, we say they are `adjacent`
- A matrix is not a very efficient way to store sparse data
- It's good to use when the number of edges is large

Adjacency List

- A more space-efficient way to implement a sparsely connected graph
- This is a list with an object that gives the adjacent vertices and their values

Implementation of a Graph as an Adjacency List

```
# Vertex() - create a new vertice with an id and what it is connected to
# addNeighbour() - create a neighbour that it is connected to
# getWeight() returns the weight of the edge from this vertex

class Vertix:
        def __init__(self, key):
                self.key = key
                self.connectedTo = {}
```

```
        def addNeighbour(self, nbr, weight=0):
                self.connectedTo[nbr] = weight

        def getConnections(self):
                return self.connectedTo.keys()

        def getId(self):
                return self.id

        def getWeight(self, nbr):
                return self.connectedTo[nbr]

        def __str__(self):
                return str(self.id) + ' connected to: ' + str([x.id for

# Graph() - create new, empty graph
# addVertex(vert) - create new instance of a vertex
# addEdge(fromVert, toVert, weight)
# addEdge (fromtVert, toVert) - without weight
# getVertex(vertKey) - return vertex
# getVertices() - return list of all vertices
# in - returns True for a statement of the form vertex in graph, if the

class Graph:
        def __init__(self):
                # dict, but modelled after adjacency list
                self.vertList = {}
                self.numVert = 0

        def addVertex(self, key):
                self.numVertices = self.numVertices + 1
                newVertex = Vertix(key)
                self.vertList[key] = newVertex
                return newVertex

        def getVertex(self, n):
                if n in self.vertList:
                        return self.vertList[n]
                else:
                        return None

        # f: from, t: to, cost: weight
        def addEdge(self, f, t, cost=0):
                if f not in self.vertList:
                        nv = self.addVertex(f)
                if t not in self.vertList:
                        nv = self.addVertex(t)

                self.vertList[f].addNeighbour(self.vertList[t], cost)

        def getVertices(self):
                return self.vertList.keys()
```

```
        def __iter__(self):
                return iter(self.vertList.values())

        def __contains__(self, n):
                return n in self.vertList

g = Graph()

for i in range(6):
        g.addVertex(i)

g.vertList
# gives back dict of vertices
# [ 0: <memory pos>, 1: ...]
g.addEdge(0,1,2)
for vertex in g:
        print vertex
        print vertex.getConnections()
        print '\n'

# prints list with edge connected from 0 to 1
```

18.1: BFS - Breadth First Search

- One of the easiest algorithms for searching a graph
- Given graph `G` and starting node `s` is that explores all vertices that are distance `k` from `s` before finding any that are `k+1` from `s`

Algorithm for exploration if vertex is unexplored:

1. The new, unexplored vertex `nbr` is coloured gray
2. The predecessor of `nbr` is set to the current node `currentVert`
3. Distance to `nbr` is set to the distance to `currentVert + 1`
4. `nbr` is added to the end of a queue - adding `nbr` to the end of the queue effectively schedules this node for further exploration - but not until all the other vertices on the adjacency list of `currentVert` have been explored

- Ultimately, the implementation will construct a tree! - This will help solve issues about finding the path! - Now we can also find any short word later back to the root from any vertex!

```
def bfs(g, start):
        # would need to implement setDistance and setPred
        start.setDistance(0)
        start.setPred(None)
        vertQueue = Queue()
```

```
        vertQueue.enqueue(start)
        while (vertQueue.size() > 0)
                currentVert = vertQueue.dequeue()
                for nbr in currentVert.getConnections():
                        if (ngr.getColor() == 'white'):
                                nbr.setColor('gray')
                                nbr.setDistance(currentVert.getDistance(
                                nbr.setPred(currentVert)
                                vertQueue.enqueue(nbr)
                currentVert.setColor('black')
```

18.2: DFS - Depth First Search

Knight's Tour Problem

On a chess board, how can the knight move? And from there, making more moves, how can we move?

Again, ajacency matrix would be sparse - so we certainly want an adjaency list.

DFS will explore each node as deeply as possible. That being said, the nodes can be visited more than once.

It will then return as far back as it can to find the next legal move.

The `knightTour` function takes four parameters:

1. `n` - the current depth of the search tree
2. `path` - a list of vertices visited up to this point
3. `u` - the vertex in the graph we wish to explore
4. `limit` - the number of nodes in the path

The function itself is also recursive.

We use a queue to keep a list of what vertice to visit next.

```
def knightTour(n, path, u, limit):
        u.setColor('gray')
        path.append(u)
        if n < limit:
                nbrList = list(u.getConnections())
                i = 0
                done = False
                while i < len(nbrList) and not done:
                        if nbrList[i].getColor() == 'white':
                                done = knightTour(n+1, path,
nbrList[i], limit)
                        i = i + 1
```

```
                if not done: # prepare to backtrack - haven't reach
limit
                        path.pop()
                        u.setColor('white')
        else:
                done = True
        return done
```

DFS Overview

- Knight's tour is a special case where the goal is to create the deepest depth first tree without any branches
- Really the general DFS is more about searching as deep as possible and then branching where needed

DFS

- As with the BFS, our DFS makes use of predecessor links to construct the tree
- In addition, DFS will make use of tw additional instance variables in the Vertex class
- New instance variables are the discovery and finish times
- Discovery time tracks the number of steps in the algorithm before a vertex is first encountered
- Finish time is the number of steps in the algorithm before a vertex is coloured black

```
class DFSGraph(Graph):
        def __init__(self):
                super().__init__()
                self.time = 0

        def dfs(self):
                for aVertex in self:
                        aVertex.setColor('white')
                        aVertex.setPred(-1)
                for aVertex in self:
                        if aVertex.getColor() == 'white':
                                self.dfsvisit(aVertex)

        # dfsvisit uses a stack
        def dfsvisit(self, startVertex):
                startVertex.setColor('gray')
                self.time += 1
                startVertex.setDiscovery(self.time)
                for nextVertex in startVertex.getConnections():
                        if nextVertex.getColor() == 'white':
                                nextVertex.setPred(startVertex)
                                self.dfsvisit(nextVertex)
                startVertex.setColor('black')
```

```
                self.time += 1
                startVertex.setFinish(self.time)
```

- Start and Finish times display a property called the `parenthesis property`
- This means that all children of a particular node in DFS have a later discovery time and an earlier finish time than their parent