Overview

Over the summer I went through an introductory course on data structures and algorithms. This course went included lectures, readings on a textbook, and problems to solve. I got to somewhat experience what a college student in a csa course would experience. The difficulty of the course was greater than I expected, despite that I've found myself excited to experience the real thing when I get to college. Here is a collection of the things I learnt from the class.

Key terms:
Algorithm: Set of instructions designed to accomplish a task
Data structure: Way or organizing day to be stored
Array: Data structure that consists of a collection of elements each identified by a key
Root: First node of a tree
Binary Tree: Data structure with a root, and nodes, with each node having at most two children
Recursion: A method of solving a problem that depends on the solution of the same method just with a smaller input.
Iterate: Repetition through a set of instructions, to go through some type of array for example
Set: A mathematical collection of different things/elements
Key: An identifier to access values in an object/data structure
Stack: data type acting as a collection of elements which adds or removes most recent elements

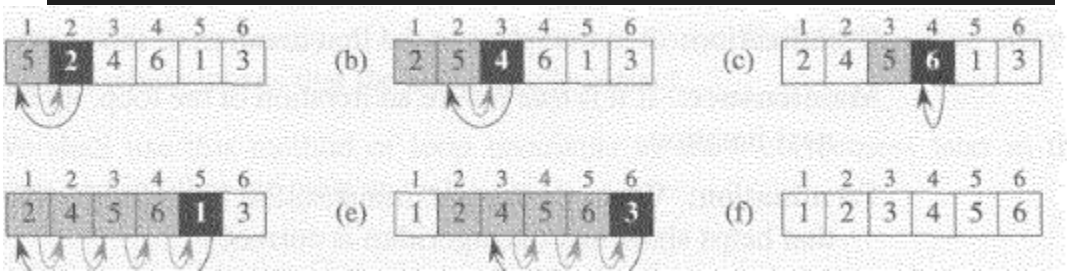
Sorting and Trees

About:

The concept of Trees as well as easier/simpler concepts of sorting through a set of data. Several different types of sorts with a baseline example for a tree likely to be seen and expanded on in the future.

Point:

- Insertion Sort
    - Efficient algorithm for sorting a small amount of elements
    - Start from the left and go down, moving elements into their corresponding location based on the size of said element.

-
```
#Insertion Sort
A = [5,2,4,6,1,3]
for j in range(1,len(A)):
    key = A[j]
    i = j-1
    while i >=0 and A[i] > key:
        A[i+1] = A[i]
        i -=1
    A[i+1] = key
print(A)
```
- 

    - **Loop Invariant**
        - Assume that A[0,j-1] is sorted.
        - Initialization: A[1,1] is sorted
        - Maintenance:
            - Assume A[0,j-1] is sorted, it takes the next element and moves it into the correct spot, making it so A[0,j] is sorted.
        - Termination
            - Loop runs until j = len(A)), which would mean A is sorted since the maintenance is sorting inside the loop.
- Binary Search Trees
    - (Pseudocode will be written in notes with my implementation at the bottom)
    - Data Structure organized in a binary tree.
    - Represent said tree through a linked data structure where a node is an object.
        - Each Node contains a key and attributes describing it's left and right children as well as its parent
            - If said thing is missing, represented as None or NIl
    - Keys always stored to satisfy the **binary search tree property**
        - Let x be a node in a binary search tree. If y is a node in the left subtree, then y.key <= x.key. If y is a node in the right subtree of x then y.key >=x.
        - This property allows one to print all keys in the tree through an **inorder tree walk**, which is a simple recursive algorithm.
            - Prints the key of the root of a subtree between values in left and right subtrees.
                - InorderTreeWalk(x):

            - if x not NIL:
                - InorderTreeWalk(x.left)
                - print x.key
                - IndorderTreewalk(x.right)
- Searching
    - Given a pointer to the **root** of the tree, and a key k, can return a pointer to the node with key k if it exists
        - TreeSearch(x,k)
            - if x is nil or k is x.key
                - return x
            - if k < x.key
                - return treeSearch(x.left, k)
            - else return treeSearch(x.right,k)
    - Iterating instead of recursion is usually more efficient here though
        - TreeSearch(x,k)
            - while x isn't NIL and k isn't x.key
                - if k < x.key
                    - x is x.left
                - else x is x.right
            - return x
- Min/Max
    - Simply start with an x and then follow it left or right completely and you'll find a min or max for it respectively.
- Insertion
    - Loops through until finding where x is NIL, and has y as a parent. Y is what x is assuming it isn't NIL, or if tree is empty y would stay NIL. Then attaches onto left or right of y depending if it's < or > /
    - Y is there because can't define anything if x is NIL. It trails behind x basically.
- Deletion
    - 3 Basic Cases for del node z from tree T
        - if z has 0 children, remove by modifying parent to replace z with NIL
        - If z has 1 child, elevate child to z position in tree by changing z's parent to replace z with z's child
            - Basically take a look at z, remember it's child delete z, insert z's child onto z's parent.
            - I think
        - If z has 2 children, find z's successor y.
            - y must be in the z.right subtree.
            - Y take pos of z
            - rest of right tree becomes y's right tree and z left becomes y left
        - Define a function to "transplant" one node subtrees to another.
            - Transplant(T,u,v)
                - if u.p == NIL
                    - T.root = v

        - elseif u == =u.p.left:
            - u.p.left = v
        - else u .p.right = v
        - if v not NIL
            - v.p = u.p
- Using Transplant, can now properly delete a node (z) from BST T
    - TDel(T,z)
        - if z.left == NIl
            - Transplant(T,z,z.right)
        - elif z.right == NIL
            - Transplant(T,z,z.left)
        - else y = TreeMin(z.right) (Min was from following all the way to the left)
            - if y.p not z
                - Transplant(t,y,y.right)
                - y.right = z.right
                - y.right.p = y
            - Transplant(t,z,y)
            - y.eft = z.left
            - y.left.p = y
    - Handles things by dealing with if
        - no left child
        - no right child but has left child
        - cases where z has 2 children

-
```
#BST
class BinarySearchTree:
    def __init__(self):
        self.root= None
    def Insert(self,z):
        newnode = Node(z)
        y = None
        x = self.root
        while x != None:
            y =x
            if newnode.key < x.key:
                x = x.left
            else: x = x.right
        newnode.parent = y
        if y == None:
            self.root = newnode
        elif newnode.key < y.key:
            y.left =newnode
        else: y.right = newnode


    def Min(self,x):
        if x.left == None:
            return x
        return self.Min(x.left)
    def Max(self,x):
        if x.right == None:
            return x
        return self.Max(x.right)
    def TreeSearch(self,x,k):
        while x != None and k != x.key:
            if k<x.key:
                x= x.left
            else: x = x.right
        return x

    def InorderTreeWalk(self,x):
        if x != None:
            self.InorderTreeWalk(x.left)
            print(x.key)
            self.InorderTreeWalk(x.right)

class Node:
    def __init__(self,key):
        self.key = key
        self.left = None
        self.right = None
        self.parent= None
```

- Counting Sort
    - Assumes that each of the n input elements is an int in range 0 - k.
    - Determines number of elements less than x, uses this info to place x in correct position in array
    - Assume input is array A[1..n], where len(A) = n.
        - 2 other arrays used. B which holds sorted output and C[0..k] which is temp. working storage

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i - 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] - 1
```

        -
- **Heaps**
    - Data structure, an array object close to a full **binary tree**
    - Each node contains an element
        - Tree full on every level except the lowest, with the lowest being filled to a point

- An array representing a heap as two properties
  - A.length: Gives the length of A
  - A.heap-size: How many elements of the heap are in the array
    - A[1..A.length] can contain numbers but the only part of the Heap is [1…A.heap-size], note that heap size will be less than A.length i.e: 0<=A.heap-size<= A.length
- The root is the first element in the array i.e A[0] or A[1]
- Given an index i of a node one can find it's parent and both left and right children
  - Parent: i/2
  - Left Child: 2i
  - Right child 2i+1
- Two types of **binary** heaps
  - Each type satisfies a specific heap property(depends on heap type)
  - Max heaps
    - Heap property: For every node i other than root, the parent node will be greater than or equal to i
    - For heapsort, max heap is used
  - Min Heaps
    - Heap property: For every node i other than root, the parent node will be less than or equal to i
    - used in things like priority queues
- As a tree the height of a node in a heap is the number of edges on the longest downward path from node to a leaf, with the root being the height of the heap
- Maintaining a heap
  - Assumes binary trees at left(i) and right(i) are max heaps.
  - Lets value at A[i] go down in the max heap so the tree at i obeys the property of being a max heap
  - Max-Heapify(A,i)
  - l= left(i)
  - r = right(i)
  - if l <= heap size and A[l] > A[i]
    - largest is l
  - else largest is i
  - if r <= heap size and A[r] ? A[largest]
    - largest = r
  - if largest not i:
    - exchange A[i] with A[largest]
    - Amex heapify(A,leagest)
- Creation of a heap
  - Given an array A
  - A.heap-seize = A.length
  - for i from A.length * .5 to 1:
    - Max Heapify(A,i)
- Heapsort

        - Uses creation of max heap on array A[1..n], n= A.length
        - Max element at A[1], can be placed at correct location by exchange with A[n]
        - Discard n, observe children of root remain max heaps, but new root may violate.
        - To fix this, call max heapify on the array and the first element which makes a max heap for A[1..n-1].
        - Repeats until last part removed array is sorted
    - Priority Queue
        - data structure for maintaining a set S of elements, each of which has a key
        - Supported operations
            - Insert(S,x): Inserts element x into set S i.e (S= S $\cup$ {x})
            - Maximum(S) returns element of S with largest key
            - Extract-Max(S) removes element of S with largest key
            - Increase-Key(S,x,k): Increases value of x's key to new value k which is assumed to be >= x's current key
        - Scheduling is a prime example of priority queue usage
            - Once a job is complete it will choose the highest priority job.
        -
    -

React:

I've found that the world of sorting numbers and storing them in a way is more complex than I thought. But it's quite interesting how it's done. Depending on the purpose of what you're doing and your inputs there are many different ways to sort your data with different efficiency between them. Even if I haven't explored a super large amount of them here doesn't mean I won't learn about them from practical experience or simply future research

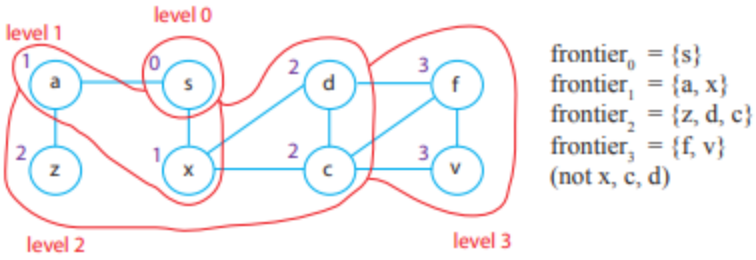
Graphs

About

Point

- **Graph**:
    - G = (V,E)
        - V = {set of vertices}
        - E = {set of edges}
            - Unordered pairs: undirected
            - Ordered pairs: Directed
    - Applications of Graph search
        - Web crawling
        - Social Networking
        - network broadcasting

        - garbage collection(in things like C/C++ manual, modern languages auto have this)
    - Representation
        - Adjacency Lists:
            - Array obj of |v|
                - each elemnts pointer to a linked list
            - Indexed by vertex
                - for each vertex u in V:
                    - Adj[u] stores neighbors(vertices you can reach in one step)
                    - Adj[u] = set of all vertices V such that (u,v) is a edge
        - Obj-oriented
            - object v
                - v.neighbors = Adj[v]
            - cleaner for single graph
- BreadthFirst Search
    - Start at some node, look at nodes reachable from there, repeat, till done
    - Looks at all vertices reachable from some node
    - Observe nodes reachable in 0 moves({s}), 1 move, 2 moves….n moves
    - Avoid duplicates
        - def BFS(s, Adj):
        - level = {s: 0} # Initialize the level of the source node s as 0
        - parent = {s: None} # The source node has no parent
        - i = 1 # This will track the level of the BFS
        - frontier = [s] # Start with the source node
        -
        - # Continue exploring the frontier as long as it's not empty
        - while frontier:
        - next_frontier = [] # The next level of the frontier
        - for u in frontier:
        - # Explore all neighbors of u
        - for v in Adj[u]:
        - # If v hasn't been visited (isn't in level dictionary)
        - if v not in level:
        - level[v] = i # Assign level i to node v
        - parent[v] = u # Set u as the parent of v
        - next_frontier.append(v) # Add v to the next frontier
        - frontier = next_frontier # Move to the next level
        - i += 1 # Increment level counter
        -

- 

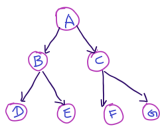
- Depth First Search
  - follows a path until stuck
  - then goes back until reaches somewhere unexplored
  - recruseive(calls itself)
  - make sure not go back to a previously visited vertex
  - parent = {s:None}
  - DFS - visit(V,adj,s):
    - for v in adj[s]:
      - if v not in parent:
        - parent[v]= s
        - DFS - visit(V,adj,v)
  - DFS(V,Adj):
    - parent = {}
    - for s in V:
      - if s not in parent:
        - parent[s] = None
        - DFS-visit(V,Adj,s)

React

Personal Research

About: This is general information that I got when trying to understand something or new things needed for project

Point:

- DFS\BFS[1]
  - 
    - BFS would go through: ABCDEFG
    - DFS would go through: ABDECFG

    - BFS traverses level by level. In a sense it could be thought of as width first. I.e breadth first search
    - DFS on the other hand, goes deep, it goes until there's a leaf than goes back up to the next part.
    - BFS helpful:
        - Shortest path from souce to destination
        - See if there is a path between two nodes
    - DFS helpful:
        - Exhaust all possibilities and choose which is best
        - Count all paths from source to destination
        - Back track/
- DFS[2]
    - In a graph, initalize a stack
    - Mark the first node as visited and insert into stack. Visit nonvisted vertices from S. Mark as visited and add to node
    - Visit the next thing from that S+1 node.
    - Keep visiting things going down.
    - In a directed graph things are more specific in comparison to a undirected graph where there are more edges/unspecific connections.

Product Plan

Goals: Make a cube solver that can solve a 3x3 rubiks cube in either under 60 moves or that can put out a solution based on human solving methods

Subgoal: IF main goal is completed either implement human solving methods, add a different computational method, or make a calculator

Materials: My personal computer which I already have, so no price needed to pay either

Shadowing: Went through a data structures and algorithms course through MIT which helped me understand some of the basic algorithms used in the algorithms for solving it.

Productive: I'll know it's effective if it works and solves a variety of scrambles correctly. If I can take the same solution and do it by hand then I'l know if that's correct.

Data: I'll either generate or take several scrambles and plug them into the solver. If they solve it in less moves than me and less moves than the average for human methods than I'll know it's correct. Optomilally there will be no solves greater than 20(gods number) when using one of the algorithms.

Sources

[1] Govind R. 2021 Feb 26. A begunners guid to BFS and DFS. leetcode.com https://leetcode.com/discuss/study-guide/1072548/A-Beginners-guid-to-BFS-and-DFS.

[2] Data Structure - Depth First Traversal - Tutorialspoint. wwwtutorialspointcom. https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm.

Chen J. 2022. Different Algorithms to Solve a Rubik's Cube. iopscience org. https://iopscience.iop.org/article/10.1088/1742-6596/2386/1/012018/pdf#:~:text=Overall%2C%20Korf's%20algorithm%20is%20the,solutions%20that%20are%20more%20optimized.&text=%5B1%5D%20Botto%2C%20B.,(2021)..