Data Structures & Algorithms in Python 1/

1. Given 2 strings, check if anagrams
2. Given an array, find pairs that sum K
3. Find missing element in 2 arrays
4. Largest continuous sum in an array
5. Reverse words in a sentence
6. String compression
7. Unique characters in string

Arrays

BASICS
- lists, tuples, strings
- Elements can be: mutable: lists, sets, dicts; immutable: numbers, strings, tuples
- append() vs extend() (referential)

Referential copies
- slicing
- Re-assignment creates new object if elements immutable

from copy import deepcopy

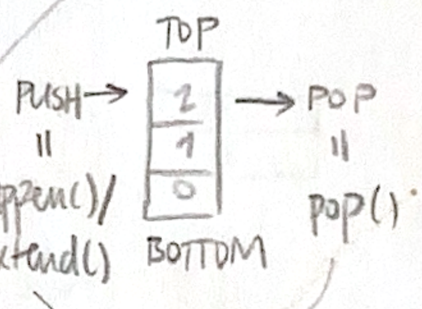
Dynamic arrays: capacity doubled if limit reached, O(1)* amortized pushing / extending

another way(queue):
- push() = append()
- pop() = pop(0)

0. Implement Stack & Queue
1. Check if parentheses balanced
2. Implement a Queue using 2 Stacks

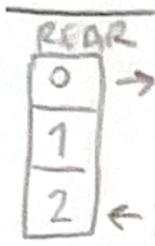
Stacks & Queues

① Stacks: LIFO



both operations at TOP! → Can reverse order!

② Queues: FiFo



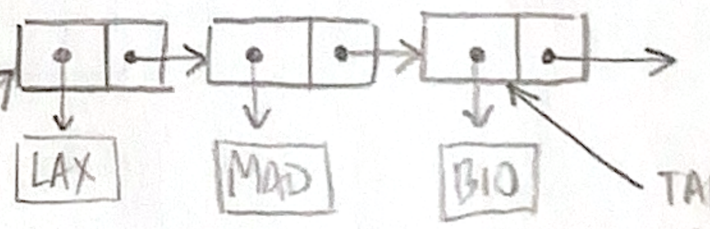
both operations at different ends! Usually push at pos ∅!

③ Deques: double ended queues
- addFront() = append()/extend()
- addRear() = insert(0, ·)
- removeFront() = pop()
- removeRear() = pop(0)

0. Implement singly/doubly linked lists
1. Check if cycle in a singly linked list
2. Reverse a singly linked list
3. Given a list head and a value n, return the nth node before the tail

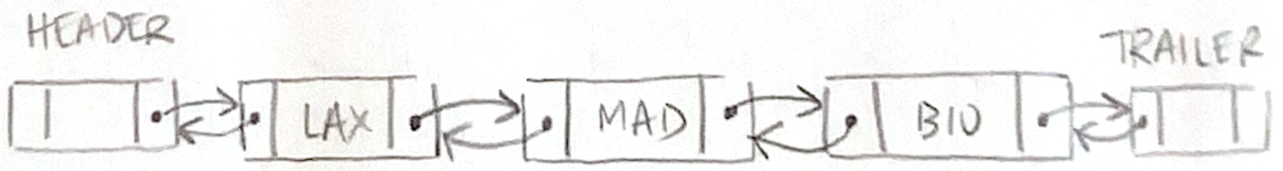
Linked Lists

① Singly linked lists: nodes that contain 1) value or ref to obj 2) ref to next node



- Access item K: O(k)
- Change/rm/insert in pos K once there: O(1)

② Doubly linked lists: like singly linked, but 1 node + 2 refs



Advantage vs. singly linked: we can traverse from tail/trailer to head

0. Basics: factorial, $\sum_{i}^{n} i$, sum of number digits
1. Split a stream of chars in words from a vocabulary
2. Memoize factorial: with function / with class
3. Reverse a string
4. All possible permutations of a string ⚠
5. Fibonacci sequence: recursively, dynamically, iteratively
6. Given an amount n and a list of coins, what's the fewest amount of coins necessary to reach n ⚠

Recursion

Recursion: when a function calls itself - a way to avoid loops
- Base case: when recursion stops
- Recursive case: when function is called again

Memoization: caching/storing expensive solutions to re-use, eg. in a dict

Dynamic programming: when memoization used with recursion. Exploding solutions become feasible

2 ways of memoizing a function:
1. with class wrapping recursive function using a dict
2. with function using a recursive fun + dict.

careful: the rec. function calls the dyn. fun!

0. Tree implementation with lists / classes
1. Implement level order traversal = breadth first.

Trees (next page)

2. Binary heap implementation

Trees

node: a: 1 | left | right — root node: no parents, no incoming edges; children (in this case ≤ 2: binary)

edge

b: 252 | left | right

c: 10 | left | right — key/name: value/payload

d: 75 | left | right — leaf node: no children

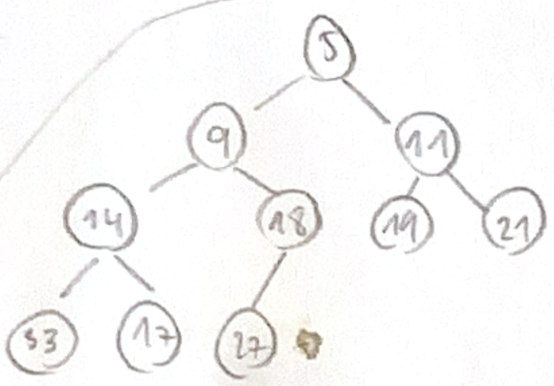
subtree

- Level of a node: # edges on the path from root to node
- Height of a tree: max level of a node in a tree
- Implementations: lists / classes (node, node, tree)
- Traversals:
  - preorder: current, left, right
  - inorder: left, current, right
  - postorder: left, right, current
  - levelorder = breadth first

Binary Heaps = Priority queues: elements ordered according to relevance

↳ represented by ARRAYS, but in reality complete trees: all nodes filled in, in last level all pushed to left; O(log n) insert/pop

Values increase/decrease



p	0	1	2	3	4	5	6	7	8	9	10
	0	5	9	11	14	18	19	21	53	17	27

X — left unused

2p+C — children position from parent position (C=0, C=1)

P//2 — parent position from child position

ROOT: most important item, will be min/max value, depending on min/max heap

↳ Typical use case: we insert new values, which are ordered and have always the min/max ready.

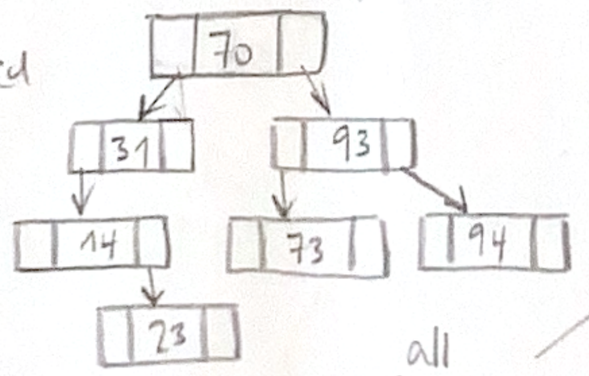
- Insert items at rear
- Percolate up / swap to keep heap property: swap if parent larger/smaller
- Remove min/max value = root
- Replace root with last item & percolate/swap down: swap if children larger/smaller

↳ To build a heap from a list: percolate down from node i = len(list)//2 : 0, ie., only non-leaves are percolated

0. Binary search Tree implementation
1. Check if a tree is a BST ⚠
2. Level order print of a BST = breadth-first; FIFO list/queue used
3. Trim a BST for values (min, max)

Binary Search Trees — There're 2 main key-value mappings < hash tables / BST

↳ BST property: keys that are smaller than the parent are on the left subtree



↳ an inorder traversals gives sorted values! Very important property!

↳ RECALL: a tree can be implemented only with the Node class!

↳ Typical use case: search values for a key in O(log n)

↳ Important operations (all recursive)

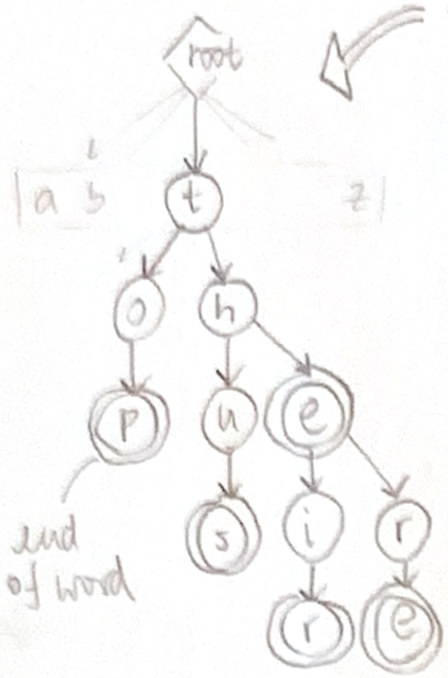
- Insert a node: put()
- Retrieve a value given a key: get()
- Delete a node: remove()
  1) Case 1: node is leaf
  2) Case 2: node has only 1 child
  3) Case 3: — " — 2 children

Data Structures & Algorithms in Python 2/

Tries

tree/graph-like structure well suited for lexicographical operations: order, check, ...

0. Implement a trie
1. Count words in a trie
2. Print words in a trie
3. Sort words in a list
4. Check if a word can be a combination of words in a dictionary.



Example: words in trie: [top, thus, the, their, there]

- Each node can have 26 children: 26 = number of characters
- Each inserted word is marked with end-of-word = True flag in the node of its last character.
- A unique word has a unique path
- Words with same prefixes / suffixes share parts of paths

Methods { insert(word), O(n), n = # chars; search(word), O(n), n = # chars; delete(word), O(n), n = # chars } for all 3, 3 sub-cases are defined

0. Find the largest palindrome in a string of digits.

Subsets

Permutations: count all possible arrays; order matters

$$P(n,K) = \frac{n!}{(n-K)!}$$

{ A, B, C, D, E, F, G, H (n=8): people; gold, silver, bronze (K=3): medals }

→ all possible medal permutations?

$$P(n=8, K=3) = \frac{8!}{5!} = 336$$

Combinations w/o replacement: count all possible unordered groups: order unimportant

$$C(n,K) = \frac{P(n,k)}{K!} = \frac{n!}{K!(n-k)!}$$

→ all possible top-3 teams?

$$C(n=8, K=3) = \frac{8!}{3!5!} = 56$$

Combinations with replacement

Example: You have n=5 ice-creams: A, B, C, D, E. You can take K=3 scoops and can repeat flavour: 1, 2, 3. How many varieties do you have?

$$C_R(n,K) = C(n+K+1, K) = \frac{(n+K-1)!}{K!(n-1)!}$$

$$C_R(n=5, K=3) = \frac{7!}{3!4!} = 35$$

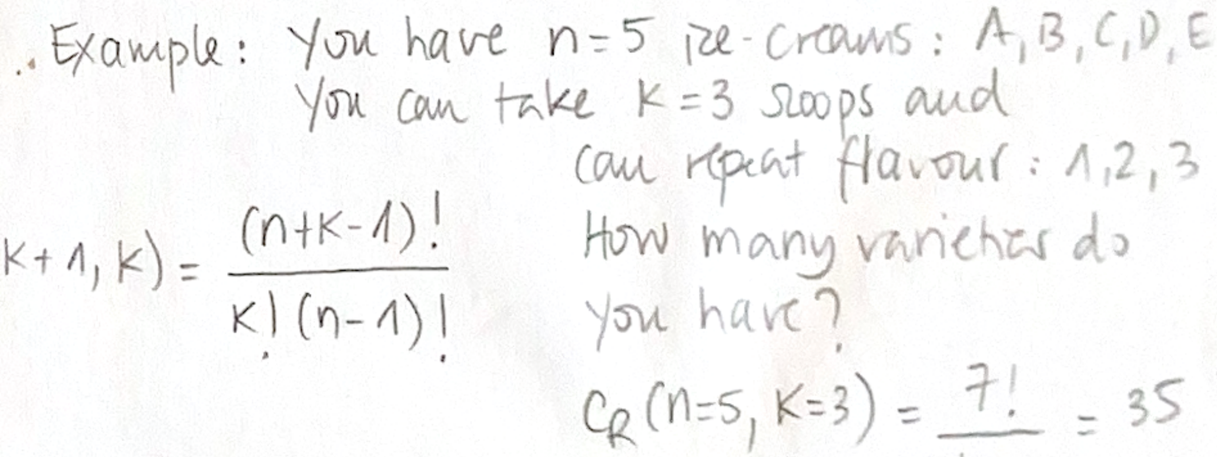
Search

0. Implement sequential search

Sequential search: Visit all items in a collection linearly until we find our queried item; O(n) worst case. For unordered collections.

0. Implement binary search

Binary Search: For ordered collections. Iteratively/recursively check the midpoint of the proper half; worst case O(log n)

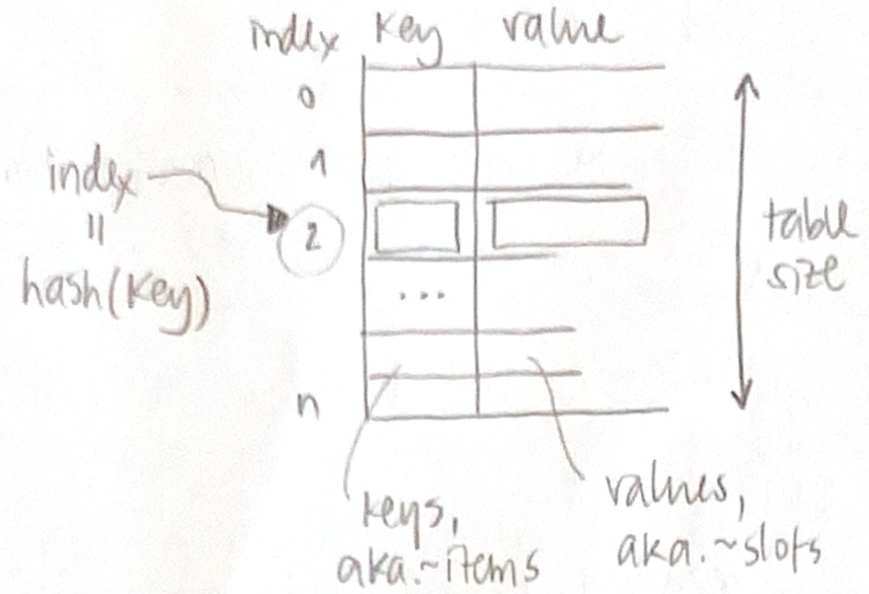


related to binary search trees, but no tree!

Hash tables (next page)

0. Implement a hash table

Hash tables Tables / arrays which map keys to values using a hash function. O(1) for insert, delete, search!



- keys need to be hashable
- common hash functions:
  - Remainder: $h(key) = key \% table\_size$
  - Folding: split item and sum
  - Mid-square: square item and pick middle digits
  - String ordinals: split into chars and get ordinals

- Load factor: $\lambda = \frac{\text{num items}}{\text{table size}}$

- Collisions: when the slot of a new key is already taken; there are several resolution approaches:
  - Open addressing: find next free slot
    - Linear probing: move sequentially until next free found $(i+1) \% table\_size$
    - Quadratic probing: jump n quadratic steps to avoid clusters characteristic from
    - double hashing: apply new hash
  - Other approaches
    - Rehashing: create a new hash table; expensive
    - Chaining: allow storing a collection of key-value pairs per slot in a linked list.
- Python:
  - dictionaries are hash tables
  - make a class hashable with __hash__ and __eq__