Data Structures

Built-in python data structures and relevant notes:

Structure	Python	Relevant Notes
Vector	list()	append, pop, insert, remove, extend, index, clear
HashMap	dict(), collections.defaultdict(lambda: 0)	d[k]=v, d.pop(k). CPython uses open addressing and random probing to solve collisions.
HashSet	set()	add, update, remove, clear, union, intersection
Stack	list()	
Dequeue	collections.deque	rotate, append, appendleft, pop, popleft, extend, extendleft
Priority Queue	heapq	heapify, heappush, heappop, nlargest, nsmallest

Function (Python)	Relevant Notes
sorted(iterable, key=key, reverse=reverse)	Ascending sort of an iterable collection
reversed(sequence)	Reverses a sequence (lists, strings, tuples, ...)
bin(number)	Binary string representation of a number

Tree

- acyclic graph (root + children)
- given the height of tree as H:
  - O(H) lookup
  - O(H) insert
  - O(H) delete

Binary Tree

- a tree with at most 2 children
- no certainty regarding tree height, hence:
  - O(H) lookup
  - O(H) insert
  - O(H) delete

Binary Search Tree

- a binary tree where left < root < right
- no certainty regarding tree height, hence:
    - O(H) lookup
    - O(H) insert
    - O(H) delete

Balanced Binary Search Tree

- a binary search tree where the height difference between subtrees is at most 1
- insertions and deletions possibly make the tree unbalanced, self-balancing trees correct this through rotations (e.g. AVL)
- the height H is balanced, hence with N nodes height is log N, thus:
    - O(log N) lookup
    - O(log N) insert
    - O(log N) delete

Trie

- trees of characters
- terminal nodes (leaves) represent words
- allows caching of current prefix and current node for efficient search
- given the prefix length of K:
  - O(K) lookup
  - O(K) insert
  - O(K) delete

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.terminal = False

    def insert(self, word):
        cur = self
        for c in word:
            if c not in cur.children:
                cur.children[c] = TrieNode()
            cur = cur.children[c]
        cur.terminal = True

    def remove(self, word):
        cur = self
        for c in word:
            if c not in cur.children:
                break
            cur = cur.children[c]
        cur.terminal = False

    def search(self, word) -> bool:
        cur = self
        for c in word:
            if c not in cur.children:
                return False
            cur = cur.children[c]
        return cur.terminal
```

Heap (Max)

- balanced binary tree
- root is bigger than children (recursive definition meaning maximum is at the top)
- insertion is done by inserting new element in the last spot and bubbling it up, swapping with parent if needed
- deletion is done by removing element and replacing by the last element added, swapping it down with the max child
- balanced binary tree:
    - O(1) max lookup
    - O(log N) insert
    - O(log N) delete

Disjoint Set

- keeps track of multiple sets of elements, disjoint at first
- allows fast check of disjoint sets of elements
- union(x, y) should set x and y to the same set
- find(x) should return the set x belongs to
- O(log N) union-find by tracking the size and chain to the smallest

```python
class DisjointSet:
    def __init__(self):
        self.groups = dict()
        self.sizes = dict()

    def find(self, x):
        if x not in self.groups:
            self.groups[x] = x
            self.sizes[x] = 1

        while x != self.groups[x]:
            x = self.groups[x]
        return x

    def union(self, x, y):
        x = self.find(x)
        y = self.find(y)
        if x == y:
            return

        shorter = x if self.sizes[x] < self.sizes[y] else y
        longer = y if self.sizes[x] < self.sizes[y] else x
        self.groups[shorter] = longer
        self.sizes[longer] += self.sizes[shorter]
```

Graph

- collection of vertices (V) and edges (E)
- adjacency matrix representation (good for dense graphs): V * V matrix with distances (0, inf, x)
- adjacency list representation (good for sparse graphs): list of lists of neighbors

Algorithms

Example tree:

```
      A
    /   \
   B     C
  / \   /
 D  E  F
      /
     G
```

Tree Traversal

Method	Order	Example
Pre (dfs)	**root** left right	A B D E C F G
In	left **root** right	D B E A G F C
Post	left right **root**	D E B G F C A

```python
def preorder(root):
    if root == None: return
    print(root)
    preorder(root.left)
    preorder(root.right)


def inorder(root):
    if root == None: return
    inorder(root.left)
    print(root)
    inorder(root.right)


def postorder(root):
    if root == None: return
    postorder(root.left)
    postorder(root.right)
    print(root)
```

Minimum Spanning Tree (MST)

- a tree that contains all nodes of the original one with a minimal sum of edge weights

Kruskal's Algorithm

- select minimum cost edges that do not form a cycle
- pop them one by one, using those that do not connect two already used vertices (disjoint set)
- stop when all vertices are connected
- O(E * log V)

```python
def kruskal(edges):
    edges.sort()

    mst = []
    disjoint_set = DisjointSet()
    while len(edges) > 0:
        cost, src, dst = edges.pop(0)

        # disjoint set keeps track of connectivity
        if disjoint_set.find(src) != disjoint_set.find(dst):
            disjoint_set.union(src, dst)
            mst.append((cost, src, dst))

    return mst
```

Binary Search

- cut the search space in half each iteration (logarithmic complexity)
- requires a sorted collection
- O(log N)

```python
def bin_search(nums, target):
    lb, ub = 0, len(nums) - 1
    while lb <= ub:
        mid = lb + (ub - lb) // 2
        if nums[mid] < target:
            lb = mid + 1
        elif nums[mid] > target:
            ub = mid - 1
        else:
            return mid
    return -1
```

Depth-First Search (DFS)

- LIFO approach
- search leftmost first, backtracking when needed

```python
# recursive
def dfs(root):
    print(root)
    for child in root.children:
        dfs(child)


# iterative
def dfs(root):
    stack = [root]
    while len(stack) > 0:
        top = stack.pop()
        print(top)
        for child in reversed(top.children):
            stack.append(child)
```

Breadth-First Search (BFS)

- FIFO approach
- explore all nodes in a "level" before going deeper

```python
from collections import deque

def bfs(root):
    queue = deque([root])
    while len(queue) > 0:
        front = queue.popleft()
        print(front)
        for child in front.children:
            queue.append(child)
```

Dijkstra

- greedy algorithm to find the shortest path from one node to all others
- no negative weight edges allowed
- O(E * log V)

```python
from heapq import heappush, heappop

def dijkstra(graph, src):
    dists = [float("inf")] * len(graph)
    dists[src] = 0

    visited = set()
    pq = [(0, src)]
    while len(pq) > 0:
        (_, cur) = heappop(pq)

        if cur in visited:
            continue
        visited.add(cur)

        # for each neighbor check if the cost of going
        # from current to neighbor is lower than neighbor distance
        for (neighbor, cost) in enumerate(graph[cur]):
            alt = dists[cur] + cost
            if alt < dists[neighbor]:
                dists[neighbor] = alt
            heappush(pq, (dists[neighbor], neighbor))

    return dists
```

Bellman-Ford

- finds the shortest path from one node to all others
- works for negative edges
- relaxes edges V-1 times
- can quit early if nothing improves
- can detect negative cycles
- O(VE)

```python
def bellman_ford(graph, src):
    n_vertices = len(graph)

    dists = [float("inf")] * n_vertices
    dists[src] = 0

    for _ in range(n_vertices - 1):
        # for each neighbor check if the cost of going
        # from current to neighbor is lower than neighbor distance
        for i in range(n_vertices):
            for j in range(n_vertices):
                alt = dists[i] + graph[i][j]
                if alt < dists[j]:
                    dists[j] = alt

    return dists
```

Floyd-Warshall

- shortest path between all nodes
- works for negative edges
- O(V³)

```python
def floyd_warshall(graph):
    n_vertices = len(graph)

    dists = [[float("inf") for _ in range(n_vertices)] for _ in
range(n_vertices)]
    for i in range(n_vertices):
        for j in range(n_vertices):
            dists[i][j] = graph[i][j]

    for intermediate in range(n_vertices):
        for i in range(n_vertices):
            for j in range(n_vertices):
                alt = dists[i][intermediate] + dists[intermediate][j]
                if alt < dists[i][j]:
                    dists[i][j] = alt

    return dists
```

Cycle Detection

- DFS: check if a node has been visited twice

```python
def has_cycle(root):
    visited = set()
    stack = [root]
    while len(stack) > 0:
        top = stack.pop()

        if top in visited:
            return True

        for child in reversed(top.children):
            stack.append(child)

    return False
```

- Disjoint Set: union nodes for each edge and quit if same set is found

```python
def has_cycle(edges):
    disjoint_set = DisjointSet()

    for src, dst in edges:
        if disjoint_set.find(src) == disjoint_set.find(dst):
            return True
        disjoint_set.union(src, dst)

    return False
```

- Bellman-Ford: run an extra cycle and if it improves there is a cycle

```python
def has_cycle(graph, src):
    n_vertices = len(graph)
    dists = bellman_ford(graph, src)

    # run an extra cycle to see if anything improves
    for i in range(n_vertices):
        for j in range(n_vertices):
            alt = dists[i] + graph[i][j]
            if alt < dists[j]:
                return True

    return False
```

- Tortoise & Hare: if both pointers meet, there is a cycle

```python
def has_cycle(root):
    slow, fast = root, root
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False
```

Dynamic Programming

- appliable when optimal solution depends on the optimal solution for subproblems
- bottom-up: solve base cases and compound results
- top-down: memoization, cache results and avoid recomputation, easily applied to recursive solutions

Quick Sort

- recursively sort halves, partitioned by a pivot
- swap left and right elements of the pivot and call quick sort on both halves
- O(N * log N)

```python
def quicksort(collection):
    return _quicksort(collection, 0, len(collection) - 1)


def _quicksort(collection, left, right):
    if left >= right:
        return

    pivot = collection[(left + right) // 2]
    split = partition(collection, left, right, pivot)
    _quicksort(collection, left, split - 1)
    _quicksort(collection, split, right)
    return collection


def partition(collection, left, right, pivot):
    while left <= right:
        while collection[left] < pivot:
            left += 1

        while collection[right] > pivot:
            right -= 1

        if left <= right:
            tmp = collection[left]
            collection[left] = collection[right]
            collection[right] = tmp
            left += 1
            right -= 1

    return left
```

Merge Sort

- recursively sort halves, call merge sort on each
- copy elements in order to a new array
- O(N * log N)

```python
def mergesort(collection):
    if len(collection) <= 1:
        return collection

    middle = len(collection) // 2
    left = mergesort(collection[:middle])
    right = mergesort(collection[middle:])
    merged = merge(left, right)
    return merged


def merge(left, right):
    merged = []

    l, r = 0, 0
    while l < len(left) and r < len(right):
        if left[l] < right[r]:
            merged.append(left[l])
            l += 1
        else:
            merged.append(right[r])
            r += 1

    while l < len(left):
        merged.append(left[l])
        l += 1

    while r < len(right):
        merged.append(right[r])
        r += 1

    return merged
```

Heap Sort

- build an heap (heapify O(N))
- keep popping the min element into a new array
- O(N * log N)

```python
from heapq import heapify, heappush, heappop

def heapsort(collection):
    heapify(collection)
    return [heappop(collection) for _ in range(len(collection))]
```

Object Oriented Programming (OOP)

SOLID Principles

Single Responsibility - classes should do one thing and do it well, having one reason to change

Open-Closed - classes should be open for extension and closed for modifications

Liskov Substition - classes should be substitutable for parent classes or interfaces they implement

Interface Segregation - keep interfaces thin, split big ones into smaller contracts, each client implements what is needed

Dependency Inversion - entities depend on abstractions and not on concretions

Design Patterns

Typical solutions for common software OOP design problems.

Creational - objects' creation

- **Factory** - interface for creating objects, simplifying and centralizing logic

```python
class Burger:
    def __init__(self, ingredients):
        self.ingredients = ingredients

class BurgerFactory:
    @classmethod
    def create_cheese_burger(cls):
        return Burger(["bun", "cheese", "beef-patty"])

    @classmethod
    def create_deluxe_burger(cls):
        return Burger(["bun", "cheese", "beef-patty", "tomatoe",
"lettuce"])
```

- **Builder** - construct complex objects step by step

```python
class Burger:
    def __init__(self):
        self.buns = None
        self.patty = None

    def set_buns(self, buns):
        self.buns = buns

    def set_patty(self, patty):
        self.patty = patty

class BurgerBuilder:
    def __init__(self):
        self.burger = Burger()

    def build(self):
        return self.burger

    def add_buns(self, buns):
        self.burger.set_buns(buns)
        return self

    def add_patty(self, patty):
        self.burger.set_patty(patty)
        return self
```

- **Singleton** - ensure a single instance of a class

```python
class Singleton:
    _instance = None

    @classmethod
    def instance(cls):
        if cls._instance == None:
            cls._instance = cls()
        return cls._instance
```

Behavioral - objects' communication (events / state changes)

- **Iterator** - defines how the values in a collection are iterated through

```python
class LinkedList:
    def __init__(self):
        self.head = None
        self.cur = None

    def __iter__(self):
        self.cur = self.head
        return self

    def __next__(self):
        if self.cur == None:
            raise StopIteration

        val = self.cur.val
        self.cur = self.cur.next
        return val
```

- **Command** - turns actions into objects (e.g. useful for queues, delays, undo/redo, event sourcing, ...)

```python
class Command:
    def execute(self):
        pass

class KillCommand(Command):
    def __init__(self, program):
        self.program = program

    def execute(self):
        self.program.kill()

class RestartCommand(Command):
    def __init__(self, program):
        self.program = program

    def execute(self):
        self.program.restart()
```

- **Observer** - subscription/notification of objects to events

```python
class Subscriber:
    def notify(self, event):
        pass

class Publisher:
    def __init__(self):
        self.subscribers = []

    def subscribe(self, sub: Subscriber):
        self.subscribers.append(sub)

    def notify(self, event):
        for sub in self.subscribers:
            sub.notify(event)
```

- **Strategy** - define a family of interchangeable algorithms

```python
class FilterStrategy:
    def filter(self, val):
        pass

class FilterPositives(FilterStrategy):
    def filter(self, val):
        return val > 0

class FilterNegatives(FilterStrategy):
    def filter(self, val):
        return val < 0

def filter_fn(values, strategy: FilterStrategy):
    return [x for x in values if strategy.filter(x)]
```

Structural - objects' assembly

- **Facade** - a wrapper used to abstract lower-level details

```python
class VideoConverter:
    # inner workings and system interactions abstracted
```

- **Adapter** - allow objects with incompatible interfaces to communicate

```python
class SquarePeg:
    def __init__(self, width: float):
        self.width = width

class RoundPeg:
    def __init__(self, radius: float):
        self.radius = radius

class RoundHole:
    def __init__(self, radius: float):
        self.radius = radius

    def fits(self, peg: RoundPeg):
        return self.radius >= peg.radius

class SquarePegAdapter(RoundPeg):
    def __init__(self, square_peg: SquarePeg):
        self.square_peg = square_peg
        self.radius = self.square_peg.width * math.sqrt(2) / 2
```

- **Decorator** - wrap objects with additional functionality

```python
class Text:
    def __init__(self, text):
        self.text = text

    def render(self):
        return self.text

class UnderlineText(Text):
    def __init__(self, wrapped):
        self.wrapped = wrapped

    def render(self):
        return "<u>" + self.wrapped.render() + "<u>"

class BoldText(Text):
    def __init__(self, wrapped):
        self.wrapped = wrapped

    def render(self):
        return "<b>" + self.wrapped.render() + "<b>"
```