01_Aggregate

```python
arr=[8,3,1,9,8,7,3,6]
stack=[]
stack_len=8
unit=[]
operations_count=0
def push(element):
    global operations_count,unit,stack_len
    if len(stack)<=stack_len:
        stack.append(element)
        operations_count+=1
        unit.append(1)
        print(f"Stack after pushing {element} is {stack} and the unit array is {unit}")
def pop():
    if len(stack)<=0:
        return
    else:
        top_element=stack[-1]
        stack.pop()
def multipop(k):
    global operations_count,unit
    num_of_elements=0
    for i in range(k):
        if len(stack)!=0:
            pop()
            num_of_elements+=1
            operations_count+=1
    unit.append(num_of_elements)
    print(f"Stack after popping {k} elements is {stack} and the unit array is {unit}")
    return num_of_elements
for i in arr:
    print(f"element {i} : - ")
    if i<=len(stack):
        print(f"since {i} is less than equal to {len(stack)}, therefore we multipop {i} element/s")
        multipop(i)
    push(i)
```

```
print(f"T(n)={sum(unit)} and num of operations is {operations_count}")
print(f"Time complexity O({sum(unit)//operations_count})")
```

02_Accounting

```
def accounting(n):
    size=1
    total=0
    dcost=0
    icost=0
    bank=0
    print("Elements\tDoubling Copying Cost\tInsertion Cost\tTotal Cost\tBank\t\tSize")
    for i in range(1,n+1):
        icost=1
        if i>size:
            size*=2
            dcost=i-1
        total=icost+dcost
        bank+=(3-total)
        print(i,"\t\t\t",dcost,"\t\t\t",icost,"\t",total,"\t\t",bank,"\t\t",size)
        icost=0
        dcost=0

print("-------------------Dynamic Array-------------------")
n=int(input("Enter number of elements:"))
print("Accounting method")
accounting(n)

class AccountingStack:
    def __init__(self):
        self.stack=[]
        self.cost=0
        self.balance=0
        self.op=0
```

```python
    def push(self,item):
        self.stack.append(item)
        self.cost+=1
        self.balance+=1
        self.op+=1
        self.printstack()

    def pop(self):
        self.stack.pop()
        self.cost+=1
        self.balance-=1
        self.op+=1
        self.printstack()

    def multipop(self,k):
        for i in range(k):
            self.pop()

    def printstack(self):
        print(self.stack,"\nBalance",self.balance,"\n")

print("\n-------------------Stack Operation-------------------")

s=AccountingStack()

s.push(1)
s.push(2)
s.push(3)
s.pop()
s.printstack()
s.multipop(2)

print("Total operations= ",s.op)
print("Amortized cost= ",s.cost)
```

03_Potential

```
doubling_costs = []
current_length = 1
potential = []
for i in range(1, 11):
    if current_length < i:
        current_length *= 2
        doubling_costs.append(i-1)
    else:
        doubling_costs.append(0)
    potential.append(2*i - current_length)

total_cost = [x+1 for x in doubling_costs]

print('Doubling Cost\t Iteration\t Total Cost\t Potential\t Amortized Cost')

print(f'{doubling_costs[0]}\t\t {1}\t\t {total_cost[0]}\t\t {potential[0]}\t\t {total_cost[0] + potential[0]}')

for j in range(1, 10):
    amortized_cost = total_cost[j] + potential[j] - potential[j-1]
    print(f'{doubling_costs[j]}\t\t {1}\t\t {total_cost[j]}\t\t {potential[j]}\t\t {amortized_cost}')
```

04_Hiring

```
candidates = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print("Candidates: ", candidates)
interviewed_candidates = []
hired_candidates = []

# Interview candidates in order
for candidate in candidates:
    interviewed_candidates.append(candidate)
    if not hired_candidates or candidate > max(hired_candidates):
        hired_candidates.append(candidate)
firing_cost = len(hired_candidates) - 1  # Since the last candidate is the best
```

```python
print("Normal way : ")
print("Interviewed candidates:", interviewed_candidates)
print("Hired candidates:", hired_candidates)
print("Number of candidates hired:", len(hired_candidates))
print("Firing cost:", firing_cost)

import random

candidates = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
interviewed_candidates = []
hired_candidates = []

print("\nRandomized Approach")
# Randomly select and interview candidates
for i in range(len(candidates)):
    temp = random.choice(candidates)
    interviewed_candidates.append(temp)
    candidates.remove(temp)

# Hire the best candidate so far
max = -1
for i in interviewed_candidates:
    if i > max:
        max = i
        hired_candidates.append(i)

# Calculate firing cost
firing_cost = len(hired_candidates) - 1  # Since the last candidate is the best
print("Interviewed candidates in randomized order:", interviewed_candidates)
print("Hired candidates:", hired_candidates)
print("Number of candidates hired:", len(hired_candidates))
print("Firing cost:", firing_cost)
```

05_Randomized-QuickSort

```python
import random
c1,c2 = 0,0
def randomizedqs(arr):
    global c1
    if len(arr) <= 1:
        return arr
    else:
        pivot = random.choice(arr)
        left = []
        right = []
        for i in range(len(arr)):
            if arr[i] < pivot:
                left.append(arr[i])
                c1+=1
            elif arr[i] > pivot:
                right.append(arr[i])
                c1+=1
        return randomizedqs(left) + [pivot] + randomizedqs(right)

def quicksort(arr):
    global c2
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        left = []
        right = []
        for i in range(1, len(arr)):
            if arr[i] < pivot:
                left.append(arr[i])
                c2+=1
            else:
                right.append(arr[i])
                c2+=1
        return quicksort(left) + [pivot] + quicksort(right)
```

```
arr = [i for i in range(500)]
print('Normal Quicksort')
print("Sorted Array:", quicksort(arr))
print("Number of Comparisons:", c2)
print("Randomized QS")
print("Sorted Array:", randomizedqs(arr))
print("Number of Comparisons:", c1)
```

08_KD-Balanced

```
class Node:
    def __init__(self, nums):
        self.nums = nums
        self.level = 0
        self.left = None
        self.right = None

def create_node(nums):
    return Node(nums)

def traverse_in_order(curr):
    if curr is None:
        return
    traverse_in_order(curr.left)
    print(f"({', '.join(map(str, curr.nums))}) ", end="")
    traverse_in_order(curr.right)

def make_kd_tree(seq, depth=0):
    if len(seq) == 0:
        return None

    k = len(seq[0]) #no of dimensions
    dim = depth % k

    seq.sort(key=lambda x: x[dim])
    mid = len(seq) // 2
```

```
    mid_elem = seq[mid]

    root = create_node(mid_elem)

    left_sub_arr = seq[:mid]
    right_sub_arr = seq[mid+1:]

    root.level = depth
    root.left = make_kd_tree(left_sub_arr, depth+1)
    root.right = make_kd_tree(right_sub_arr, depth+1)

    return root

if __name__ == "__main__":
    seq = [[6,2], [7,1], [2,9], [3,6], [4,8], [8,4], [5,3], [1,5], [9,5]]
    root = make_kd_tree(seq)
    print("Inorder Traversal: ",end='')
    traverse_in_order(root)
```

09_KD-Unbalanced

```
class KDNode:
    def __init__(self, data):
        self.data= data
        self.depth = 0
        self.left = None
        self.right = None

def insert(node,point):
    if node is None:
        return KDNode(point)

    dim=node.depth%2

    if point[dim] < node.data[dim]:
        node.left=insert(node.left,point)
        node.left.depth=node.depth+1
```

```
    else:
        node.right=insert(node.right,point)
        node.right.depth=node.depth+1

    return node

def inorder(node):
    if node is None:
      return
    inorder(node.left)
    print(f"({','.join(map(str,node.data))}) ",end="")
    inorder(node.right)

unbalanced_points = [[6, 2], [7, 1], [2, 9], [3, 6], [4, 8], [8, 4], [5, 3], [1, 5], [9, 5]]
unbalanced_root = KDNode(unbalanced_points[0])

for point in unbalanced_points[1:]:
  insert(unbalanced_root, point)

print("Initial tree:")
# print("Unbalanced KD-Tree (inorder traversal):")
inorder(unbalanced_root)
insert(unbalanced_root, [3,5])
print("\nAfter insertion of point (3,5):")
inorder(unbalanced_root)
```

10_FordFulkerson

```
class Graph:
    def __init__(self, graph):
        self.graph = graph
        self.ROW = len(graph)

    def bfs(self, s, t, parent):
        visited = [False] * self.ROW
        queue = []
```

```
        queue.append(s)
        visited[s] = True
        while queue:
            u = queue.pop(0)
            for ind, val in enumerate(self.graph[u]):
                if not visited[ind] and val > 0:
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u
        return visited[t], parent

    def ford_fulkerson(self, source, sink):
        max_flow = 0
        parent = [-1] * self.ROW
        while True:
            found_path, parent = self.bfs(source, sink, parent)
            if not found_path:
                break
            path_flow = float("Inf")
            s = sink
            while s != source:
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]
            max_flow += path_flow

            # Print the augmented path and its minimum value
            path = [sink]
            v = sink
            while v != source:
                u = parent[v]
                path.insert(0, u)
                v = u
            print("Augmented path: ", " -> ".join(str(x) for x in path), " Minimum flow: ", path_flow)
            # print("Graph: ", self.graph)
```

```
            v = sink
            while v != source:
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = u

        return max_flow

#            0   1   2   3   4   5
#            S,  B,  P,  M,  K,  D
graph = [   [0,  0,  0,  0,  0,  0],  #S
            [17, 0,  0,  0,  10, 0],  #B
            [6,  7,  0,  0,  0,  0],  #P
            [0,  12, 0,  0,  14, 0],  #M
            [0,  0, 10,  6,  0,  0],  #K
            [0,  0,  0,  8,  14, 0],  #D
        ]

g = Graph(graph)
source = 5
sink = 0
print("Max Flow: %d " % g.ford_fulkerson(source, sink))
```

11_Convex_Hull

```
class Graph:
    def __init__(self, graph):
        self.graph = graph
        self.ROW = len(graph)

    def bfs(self, s, t, parent):
        visited = [False] * self.ROW
        queue = []
        queue.append(s)
```

```
        visited[s] = True
        while queue:
            u = queue.pop(0)
            for ind, val in enumerate(self.graph[u]):
                if not visited[ind] and val > 0:
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u
        return visited[t], parent

    def ford_fulkerson(self, source, sink):
        max_flow = 0
        parent = [-1] * self.ROW
        while True:
            found_path, parent = self.bfs(source, sink, parent)
            if not found_path:
                break
            path_flow = float("Inf")
            s = sink
            while s != source:
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]
            max_flow += path_flow

            # Print the augmented path and its minimum value
            path = [sink]
            v = sink
            while v != source:
                u = parent[v]
                path.insert(0, u)
                v = u
            print("Augmented path: ", " -> ".join(str(x) for x in path), " Minimum flow: ", path_flow)
            # print("Graph: ", self.graph)

            v = sink
            while v != source:
```

```
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = u

        return max_flow

#            0   1   2   3   4   5
#            S,  B,  P,  M,  K,  D
graph = [   [0,  0,  0,  0,  0,  0],  #S
            [17, 0,  0,  0,  10, 0],  #B
            [6,  7,  0,  0,  0,  0],  #P
            [0,  12, 0,  0,  14, 0],  #M
            [0,  0, 10,  6,  0,  0],  #K
            [0,  0,  0,  8,  14, 0],  #D
        ]

g = Graph(graph)
source = 5
sink = 0
print("Max Flow: %d " % g.ford_fulkerson(source, sink))
```

06_RedBlack-Insertion

```
# RB tree insertion
class Node:
  def __init__(self, val, color):
    self.val = val
    self.color = color
    self.left = None
    self.right = None
    self.parent = None

class RedBlackTree:
    def __init__(self):
```

```
        self.root = None

    def insert(self, val):
        new_node = Node(val, "RED")
        if not self.root:
            self.root = new_node
            new_node.color = "BLACK"
            return

        curr = self.root
        parent = None
        while curr:
            parent = curr
            if val < curr.val:
                curr = curr.left
            else:
                curr = curr.right

        new_node.parent = parent
        if val < parent.val:
            parent.left = new_node
        else:
            parent.right = new_node

        self._fix_violations(new_node)

    def _fix_violations(self, node):
        while node.parent and node.parent.color == "RED":
            if node.parent == node.parent.parent.left:
                uncle = node.parent.parent.right
                if uncle and uncle.color == "RED":
                    node.parent.color, uncle.color, node.parent.parent.color = "BLACK", "BLACK", "RED"
                    node = node.parent.parent
                else:
                    if node == node.parent.right:
                        node = node.parent
```

```
                    self._left_rotate(node)
                node.parent.color, node.parent.parent.color = "BLACK", "RED"
                self._right_rotate(node.parent.parent)
        else:
            uncle = node.parent.parent.left
            if uncle and uncle.color == "RED":
                node.parent.color, uncle.color, node.parent.parent.color = "BLACK", "BLACK", "RED"
                node = node.parent.parent
            else:
                if node == node.parent.left:
                    node = node.parent
                    self._right_rotate(node)
                node.parent.color, node.parent.parent.color = "BLACK", "RED"
                self._left_rotate(node.parent.parent)

    self.root.color = "BLACK"

def _left_rotate(self, node):
    right_child = node.right
    node.right = right_child.left

    if right_child.left:
        right_child.left.parent = node
    right_child.parent = node.parent

    if not node.parent:
        self.root = right_child
    elif node == node.parent.left:
        node.parent.left = right_child
    else:
        node.parent.right = right_child

    right_child.left = node
    node.parent = right_child
```

```
    def _right_rotate(self, node):
        left_child = node.left
        node.left = left_child.right
        if left_child.right:
            left_child.right.parent = node
        left_child.parent = node.parent
        if not node.parent:
            self.root = left_child
        elif node == node.parent.right:
            node.parent.right = left_child
        else:
            node.parent.left = left_child
        left_child.right = node
        node.parent = left_child

    def inorder_traversal(self, node):
        if node:
            self.inorder_traversal(node.left)
            print(f"{node.val} ({node.color})", end=" ")
            self.inorder_traversal(node.right)

# Example usage
tree = RedBlackTree()
for val in [8,18,5,15,17,25,40,80]:
  tree.insert(val)
print("Inorder traversal of Red Black Tree:");

tree.inorder_traversal(tree.root)
```

07_RedBlack-Delete

```
import sys
# Node creation
class Node():
    def __init__(self, item):
        self.item = item
```

```
        self.parent = None
        self.left = None
        self.right = None
        self.color = 1

class RedBlackTree():
    def __init__(self):
        self.TNULL = Node(0)
        self.TNULL.color = 0
        self.TNULL.left = None
        self.TNULL.right = None
        self.root = self.TNULL

    # Preorder
    def pre_order_helper(self, node):
        if node != TNULL:
            sys.stdout.write(node.item + " ")
            self.pre_order_helper(node.left)
            self.pre_order_helper(node.right)

    # Inorder
    def in_order_helper(self, node):
        if node != TNULL:
            self.in_order_helper(node.left)
            sys.stdout.write(node.item + " ")
            self.in_order_helper(node.right)

    # Postorder
    def post_order_helper(self, node):
        if node != TNULL:
            self.post_order_helper(node.left)
            self.post_order_helper(node.right)
            sys.stdout.write(node.item + " ")
```

```
    # Search the tree
    def search_tree_helper(self, node, key):
        if node == TNULL or key == node.item:
            return node
        if key < node.item:
            return self.search_tree_helper(node.left, key)
        return self.search_tree_helper(node.right, key)

    # Balancing the tree after deletion
    def delete_fix(self, x):
        while x != self.root and x.color == 0:
            if x == x.parent.left:
                s = x.parent.right
                if s.color == 1:
                    s.color = 0
                    x.parent.color = 1
                    self.left_rotate(x.parent)
                    s = x.parent.right
                if s.left.color == 0 and s.right.color == 0:
                    s.color = 1
                    x = x.parent
                else:
                    if s.right.color == 0:
                        s.left.color = 0
                        s.color = 1
                        self.right_rotate(s)
                        s = x.parent.right

                    s.color = x.parent.color
                    x.parent.color = 0
                    s.right.color = 0
                    self.left_rotate(x.parent)
                    x = self.root
            else:
                s = x.parent.left
                if s.color == 1:
```

```
                    s.color = 0
                    x.parent.color = 1
                    self.right_rotate(x.parent)
                    s = x.parent.left
                if s.right.color == 0 and s.right.color == 0:
                    s.color = 1
                    x = x.parent
                else:
                    if s.left.color == 0:
                        s.right.color = 0
                        s.color = 1
                        self.left_rotate(s)
                        s = x.parent.left
                    s.color = x.parent.color
                    x.parent.color = 0
                    s.left.color = 0
                    self.right_rotate(x.parent)
                    x = self.root
        x.color = 0

    def __rb_transplant(self, u, v):
        if u.parent == None:
            self.root = v
        elif u == u.parent.left:
            u.parent.left = v
        else:
            u.parent.right = v
        v.parent = u.parent

    # Node deletion
    def delete_node_helper(self, node, key):
        z = self.TNULL
        while node != self.TNULL:
            if node.item == key:
                z = node
            if node.item <= key:
```

```
                node = node.right
            else:
                node = node.left
        if z == self.TNULL:
            print("Cannot find key in the tree")
            return
        y = z
        y_original_color = y.color
        if z.left == self.TNULL:
            x = z.right
            self.__rb_transplant(z, z.right)
        elif (z.right == self.TNULL):
            x = z.left
            self.__rb_transplant(z, z.left)
        else:
            y = self.minimum(z.right)
            y_original_color = y.color
            x = y.right
            if y.parent == z:
                x.parent = y
            else:
                self.__rb_transplant(y, y.right)
                y.right = z.right
                y.right.parent = y
            self.__rb_transplant(z, y)
            y.left = z.left
            y.left.parent = y
            y.color = z.color
        if y_original_color == 0:
            self.delete_fix(x)

    # Balance the tree after insertion
    def fix_insert(self, k):
        while k.parent.color == 1:
            if k.parent == k.parent.parent.right:
                u = k.parent.parent.left
```

```
                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    if k == k.parent.left:
                        k = k.parent
                        self.right_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.left_rotate(k.parent.parent)
            else:
                u = k.parent.parent.right


                if u.color == 1:
                    u.color = 0
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    k = k.parent.parent
                else:
                    if k == k.parent.right:
                        k = k.parent
                        self.left_rotate(k)
                    k.parent.color = 0
                    k.parent.parent.color = 1
                    self.right_rotate(k.parent.parent)
            if k == self.root:
                break
        self.root.color = 0

    # Printing the tree
    def __print_helper(self, node, indent, last):
        if node != self.TNULL:
            sys.stdout.write(indent)
```

```
            if last:
                sys.stdout.write("R----")
                indent += "     "
            else:
                sys.stdout.write("L----")
                indent += "|    "


            s_color = "RED" if node.color == 1 else "BLACK"
            print(str(node.item) + "(" + s_color + ")")
            self.__print_helper(node.left, indent, False)
            self.__print_helper(node.right, indent, True)

    def preorder(self):
        self.pre_order_helper(self.root)

    def inorder(self):
        self.in_order_helper(self.root)

    def postorder(self):
        self.post_order_helper(self.root)

    def searchTree(self, k):
        return self.search_tree_helper(self.root, k)

    def minimum(self, node):
        while node.left != self.TNULL:
            node = node.left
        return node

    def maximum(self, node):
        while node.right != self.TNULL:
            node = node.right
        return node

    def successor(self, x):
```

```
        if x.right != self.TNULL:
            return self.minimum(x.right)
        y = x.parent
        while y != self.TNULL and x == y.right:
            x = y
            y = y.parent
        return y

    def predecessor(self,  x):
        if (x.left != self.TNULL):
            return self.maximum(x.left)
        y = x.parent
        while y != self.TNULL and x == y.left:
            x = y
            y = y.parent
        return y

    def left_rotate(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.TNULL:
            y.left.parent = x
        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

    def right_rotate(self, x):
        y = x.left
        x.left = y.right
        if y.right != self.TNULL:
```

```
            y.right.parent = x
        y.parent = x.parent
        if x.parent == None:
            self.root = y
        elif x == x.parent.right:
            x.parent.right = y
        else:
            x.parent.left = y
        y.right = x
        x.parent = y

    def insert(self, key):
        node = Node(key)
        node.parent = None
        node.item = key
        node.left = self.TNULL
        node.right = self.TNULL
        node.color = 1
        y = None
        x = self.root
        while x != self.TNULL:
            y = x
            if node.item < x.item:
                x = x.left
            else:
                x = x.right
        node.parent = y
        if y == None:
            self.root = node
        elif node.item < y.item:
            y.left = node
        else:
            y.right = node
        if node.parent == None:
            node.color = 0
            return
```

```python
        if node.parent.parent == None:
            return
        self.fix_insert(node)

    def get_root(self):
        return self.root

    def delete_node(self, item):
        self.delete_node_helper(self.root, item)

    def print_tree(self):
        self.__print_helper(self.root, "", True)

if __name__ == "__main__":
    bst = RedBlackTree()
    bst.insert(55)
    bst.insert(40)
    bst.insert(65)
    bst.insert(60)
    bst.insert(75)
    bst.insert(57)
    bst.print_tree()

    print("\nAfter deleting element 40")
    bst.delete_node(40)
    bst.print_tree()

    print("\nAfter deleting element 57")
    bst.delete_node(57)
    bst.print_tree()
```