INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, NAGPUR

Department of Computer Science and Engineering

CSL 210 – Data Structures with Applications

Module 2 Complex Coding Assignments For Practice

1. Write a program for AVL tree having functions for the following operations:

Insert an element (no duplicates are allowed),

Delete an existing element,

Traverse the AVL to search for an element

INPUT:

Line 1 contains an integer NQ, the number of queries.

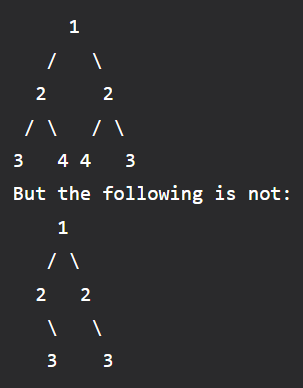
The next NQ lines contain queries and are of the form 'i x' (Insert x into an AVL tree) or

'd x' (Delete x from an AVL tree).

SAMPLE INPUT: 8 i 1 i 2 i 3 i 4 i 5 i 6 d 4 d 5

Show the output at each insertion and and deletion.

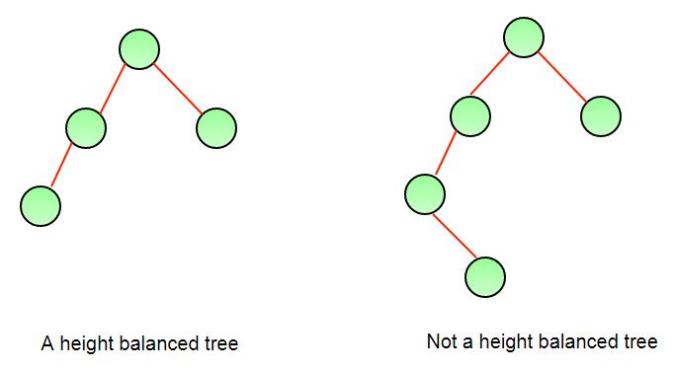
1. Given two trees with certain keys. Two trees are identical when they have the same data and the arrangement of data is also the same. Write a function to determine if two trees are identical or not.
2. Given a binary tree, check whether it is a mirror of itself. For example, this binary tree is symmetric:



The idea is to write a recursive function isMirror() that takes two trees as an argument and returns true if trees are the mirror and false if trees are not mirrored.

The isMirror() function recursively checks two roots and subtrees under the root.

1. The tree on the left is a height balanced binary tree. Whereas the tree on the right is not a height balanced tree. Because the left subtree of the root has a height which is 2 more than the height of the right subtree.



Write a program to check whether the tree is height balanced or not. Naive Approach to check if a tree is height-balanced is to get the height of left and right subtrees using dfs traversal. Return true if the difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.

1. Suppose you are implementing a B-tree with a maximum degree of t. Write a function split\_node(node) that takes a node as input and splits it into two nodes, following the rules of B-trees. Your function should split the node into two nodes, left\_node and right\_node.

The original node's median key should be moved up to its parent.

If the original node is a leaf node, both left\_node and right\_node should also be leaf nodes.

If the original node is an internal node, left\_node should contain the t-1 smallest keys and their corresponding children, and right\_node should contain the t-1 largest keys and their corresponding children.

Ensure that the parent node's keys are properly adjusted after the split.

1. Design a phone directory using a Red-Black tree. Each entry should store a person's name and phone number. Provide operations to add, delete, and search for a person.

You'll need a structure to represent each entry in the phone directory, containing fields for the person's name, phone number, and pointers to the left and right child nodes.

Operations:

Insertion:

To add a person's entry, you'll perform an insertion operation on the Red-Black tree.

Use the person's name as the key for insertion.

After insertion, you might need to perform rotations and color adjustments to maintain the Red-Black tree properties.

Deletion:

To delete a person's entry, you'll perform a deletion operation on the Red-Black tree.

Use the person's name to locate and remove the entry.

After deletion, you'll need to rebalance the tree.

Search:

To search for a person's entry, you'll perform a search operation on the Red-Black tree.

Use the person's name as the search key.

Return the phone number if found, or indicate that the person is not in the directory.

1. Suppose you are building a database system, and you decide to use a B-tree to store and manage the data. You want to implement a feature that allows users to find the k-th smallest element in the database efficiently. Write a function find\_kth\_smallest(k) that finds and returns the k-th smallest element in the B-tree.

Your function should efficiently find and return the k-th smallest element in the B-tree.

The B-tree may contain duplicates, so if there are multiple occurrences of the k-th smallest element, return any of them.

1. You are building a cache system for a web server that needs to efficiently retrieve and store web pages. To achieve this, you decide to use a Splay Tree as the underlying data structure for the cache.

Write a class **SplayTreeCache** that implements a cache using a Splay Tree. The cache should have the following methods:

**get(key):** Retrieves the value associated with the given key from the cache. If the key is present, the corresponding value should be returned and the accessed node should be splayed to the root. If the key is not present, return -1.

**put(key, value):** Inserts the key-value pair into the cache. If the cache is at its maximum capacity, the least recently used item should be evicted before inserting the new item. After insertion, the newly inserted node should be splayed to the root.

**print\_cache():** Prints out the keys and values of all nodes in the cache.

Assume the keys are unique

The cache should have a maximum capacity, and if it exceeds that capacity, it should evict the least recently used item.

The get operation should splay the accessed node to the root.

The put operation should splay the newly inserted node to the root.

1. You are building a task scheduler for a computer system. Tasks have dependencies, and you need to schedule them in such a way that no task executes before its dependencies have completed. Write a function schedule\_tasks(tasks, dependencies) that returns an ordered list of tasks to be executed.

tasks is a list of task IDs.

dependencies is a list of tuples, where each tuple represents a dependency relationship in the format (dependent\_task, dependency). For example, (B, A) means task B depends on task A.

The function should return a list of task IDs in the order they should be executed.

tasks = ['A', 'B', 'C', 'D']

dependencies = [('B', 'A'), ('C', 'A'), ('D', 'B'), ('D', 'C')]

result = schedule\_tasks(tasks, dependencies)

print(result) # Output: ['A', 'B', 'C', 'D']

In this example, tasks A and D have no dependencies, so they can be executed first. Tasks B and C depend on A, so they are scheduled next. The final order is ['A', 'B', 'C', 'D'].

1. You are building an event scheduling system for a conference. Each event has a start time and an end time, and you want to efficiently schedule as many events as possible. Write a function schedule\_events(events) that takes a list of events and returns a list of scheduled events with no overlapping times.

Requirements:

events is a list of tuples, where each tuple represents an event in the format (start\_time, end\_time).

The function should return a list of scheduled events.

events = [

(1, 3),

(2, 4),

(3, 6),

(5, 7),

(8, 9)

]

result = schedule\_events(events)

print(result) # Output: [(1, 3), (5, 7), (8, 9)]

In this example, events (1, 3), (5, 7), and (8, 9) do not overlap, so they can be scheduled.

1. Priority Task Execution

You are building a task execution system where each task has a priority level. You want to efficiently execute tasks with higher priority levels first. Write a class PriorityTaskExecutor that implements this system using a heap.

The class signature could look like this:

Requirements:

The priority task executor should use a heap to efficiently manage tasks based on their priority levels.

The add\_task method should add a task with a given priority to the executor.

The execute\_next\_task method should remove and return the task with the highest priority.

1. You are building a task scheduling system, and you decide to use a Red-Black Tree (RB Tree) as the underlying data structure for managing the tasks. Each task has a unique ID and a priority level (an integer). The tasks need to be scheduled in such a way that tasks with higher priority levels are executed first.

Write a class RBScheduler that implements a scheduler using a Red-Black Tree. The scheduler should have the following methods:

schedule(task\_id, priority): Adds a task with the given task\_id and priority to the scheduler. The task should be inserted into the RB Tree according to its priority.

execute\_next(): Removes and returns the task with the highest priority from the scheduler. If there are multiple tasks with the highest priority, return any of them. If the scheduler is empty, return None.

print\_schedule(): Prints out the task IDs and priorities of all tasks in the scheduler in ascending order of priorities.

The RB Tree should maintain the properties of a Red-Black Tree.

The schedule operation should insert tasks according to their priority.

The execute\_next operation should remove and return the task with the highest priority.