|  |  |
| --- | --- |
|  | **[Design & Analysis of Algorithm]**  **[BSCS – 5 A]**  **Department of Computer Science**  **Bahria University, Lahore Campus** |

**Solution Assignment: 4**

**Part1: CLO 2: Analysis**

**Reading Task: Explore and briefly discuss time & space complexity of AVL Trees and Red-Black Trees from following sources, you can also use other resources to study. [0.5]**

1. <https://iq.opengenus.org/time-and-space-complexity-of-red-black-tree/>
2. <https://iq.opengenus.org/time-complexity-of-avl-tree/>

**Part 1**

**A)**

AVL trees, being self-balancing Binary Search Trees (BSTs), boast impressive time complexities for fundamental operations like insertion, deletion, and search. Here's a breakdown:

**Time Complexity:**

* **Search:** O(log n) - In the worst-case scenario, searching involves traversing the tree's height, which in an AVL tree is guaranteed to be logarithmic (log n) to the number of nodes (n) due to self-balancing.
* **Insertion:** O(log n) - Similar to searching, insertion also takes O(log n) time in the worst case. This is because finding the insertion location using BST logic takes O(log n) due to the balanced nature of the tree, and the actual insertion and any necessary rebalancing (rotations) are constant time operations (O(1)).
* **Deletion:** O(log n) - Deletion follows a similar logic to insertion. Traversing to find the node for deletion takes O(log n) in the worst case, and the deletion itself along with potential rebalancing are O(1) operations.

**Intuition behind the Efficiency:**

AVL trees maintain a balance between their left and right subtrees. This ensures the tree's height stays proportional to the logarithm of the number of nodes, leading to efficient search, insertion, and deletion operations.

**Part 1**

**B)**

Red-black trees offer efficient insertion, deletion, and search operations with a guaranteed logarithmic time complexity in the worst case. Here's a breakdown:

* **Time Complexity:**
  + Search: O(log n)
  + Insert: O(log n)
  + Delete: O(log n)

Here's why:

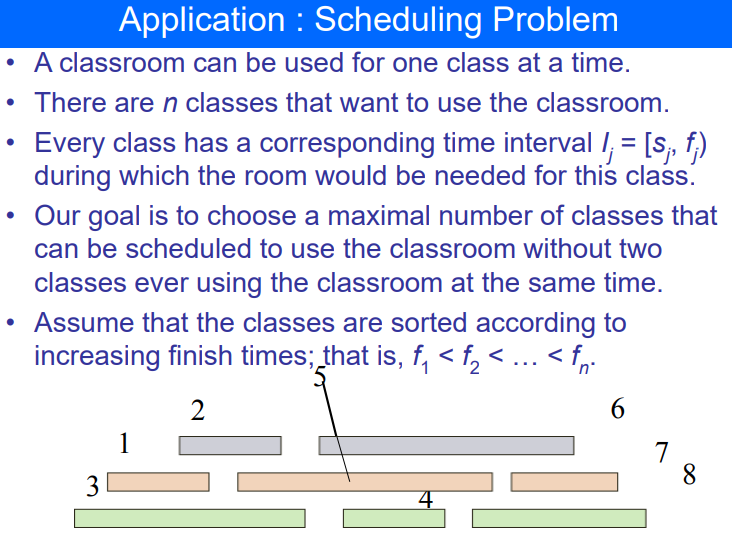
* Red-black trees are self-balancing, ensuring the tree's height is always proportional to the logarithm of the number of nodes (n).
* Search, insertion, and deletion operations traverse the tree based on its structure. Since the height is logarithmic, these operations take at most O(log n) time in the worst case.

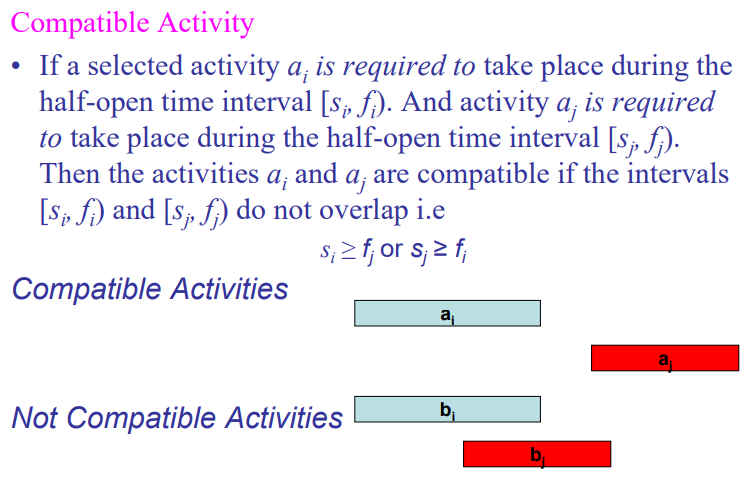
It's important to note that the rebalancing operations (rotations and color changes) that maintain the balance of the tree can also contribute to the time complexity. However, these rebalancing operations are designed to be efficient, with an average time complexity of O(1). In the worst case, rebalancing might take O(log n) time, but this doesn't affect the overall guarantee of logarithmic time complexity for search, insert, and delete.

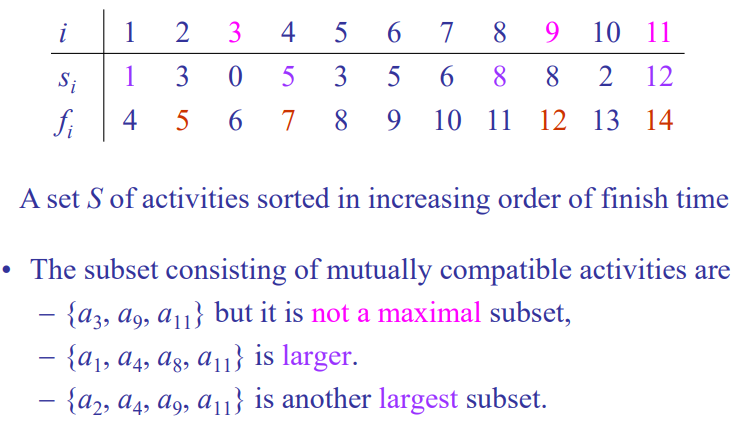
**Part 2: CLO 3: Design algorithms [4.5]**

The problem involves scheduling of several competing activities that require exclusive use of common resource Problem Statement. Suppose we have a set: ***S = {a1 , a2 , ..., an }*** of ***n*** proposed activities. Each activity wish to use a resource which can be used by only one activity at a time. Each activity ***ai*** has starting time ***si*** , finishing time ***fi*** where, 0 ≤ si < f i < ∞

Objective in activity-selection problem is to select a maximum-size subset of mutually compatible activities. Its application is explained through scheduling example given below:

****

****

****

