Coding+Theory 3: CS2233

13th October, 2025

Instructions: Please strictly follow the input and output format specified for each problem. Your code should run in an infinite loop, continuously waiting for the query number, and **reading queries until EOF**.
Note: Failure to follow the input and output format exactly may result in 0 marks for that problem.
Initialize the tree by inserting n elements from the array given in the second line (in order). **Duplicates must not be inserted** (see messages below). After initialization, the program reads queries until EOF.

Query types:

- **1** — *search* query
- **2** — *insert* query
- **3** — *delete* query
- **4** — *print level-order*

Input format

- First line will contain n, which indicates the number of elements to be inserted into the tree.
- Second line will contain n one-space-separated integers, which are your tree elements. Assume this to be an array of n integers, inserted in order to build the initial tree (ignore duplicates).
- Queries: Each subsequent line is either:
  - Two integers: `<type> <value>` for types **1** (search), **2** (insert), **3** (delete), or
  - One integer: `4` to print the tree level-by-level.

Output messages (uniform across Problems 1–3):

- Search: `‘<x> present’’` or `‘<x> not present’’`.
- Insert: if duplicate `‘<x> already present. So no need to insert.’’`; else `‘<x> inserted’’`.

- Delete: if present ‘<x> deleted’’; else ‘<x> not present. So it can not be deleted’’.
- Level-order print (type 4): *Each level on a new line, nodes left-to-right, single spaces between nodes.*

Example:
Input and Output:
17
9 8 5 4 99 78 31 34 89 90 21 23 45 77 88 112 32
1 56
(Output for this query. From now on, here, blue text represents the expected output)56 not present
2 21
21 already present. So no need to insert.
2 56
56 inserted
2 90
90 already present. So no need to insert
3 51
51 not present. So it can not be deleted
1 32
32 present
3 32
32 deleted
4
Level-order (one line per level). Example format:
<level 0 nodes>
<level 1 nodes>
<level 2 nodes>

Note:- Code should run in infinite loop expecting the query.

Max Marks 65

1 Coding Problems:

1. Construct an AVL tree.

   The AVL tree should be constructed by **repeatedly calling your** `insert (root, key)` on the given keys (left to right).

   Each node of the tree should use the following `struct` data type:

```
struct node
{
int data;
int height; /* stores the height of the current node */
```

```
struct node *left;
struct node *right;
};
```

You can assume that you have stored the pointer to the `root` node. Please, write the functions for:

(a) `search (root, key)` – this function takes the pointer to the `root` node, and `key` as input, and outputs `'key present'` if found, else `'key not present'`.

(b) `insert (root, key)` – this function takes the pointer to the `root` node, and key as input, and inserts the node at the appropriate position (ignore duplicates as per the global output messages).

(c) `delete (root, key)` – this function takes the pointer to the `root` node, and the key as input, and deletes the corresponding node (output messages as specified globally).

(d) Use query type **4** to output the tree by printing the nodes level-by-level.

2+3+3+2=10 Marks

2. Construct a `(2-4)`-tree.

The `(2-4)`-tree should be constructed by calling your `insert (root, key)` on the given keys. Each node of the tree should use the following `struct` data type:

```
struct node
{
int keys[3];
int count; /* number of valid keys in this node:  1..3 */
struct node *children[4];
};
```

Use the global query interface (types 1–4) and output messages defined above. In particular, print the tree level-by-level only when query type **4** is issued.

2+3+3+2=10 Marks

3. Construct a `B`-tree.

The `B`-tree should be constructed by calling your `insert (root, key)` on the given keys. Each node of the tree should use the following `struct` data type, with order $t \geq 2$, $MIN = t - 1$, $MAX = 2t - 1$ (keys per node in $[MIN, MAX]$ except root):

```
struct BTreeNode
{
```

```
int key[MAX]; /* store keys at indices 0..count-1 (0-based) */
int count; /* current number of keys in this node:  0..MAX */
struct BTreeNode *children[MAX + 1];
};
```

Use the global query interface (types 1–4) and output messages defined above. Print the tree level-by-level only for query type **4**.

2+3+3+2=10 Marks

4. The AVL tree should support insertion and deletion of integer keys. Each node of the tree should use the following struct data type:

```
struct node
{
int data;
int height; /* stores the height of the current node */
struct node *left;
struct node *right;
};
```

Standalone batch I/O for this problem (does not use the global query loop).

You can assume that you have stored the pointer to the root node. Please, write the functions for:

(a) search(root, key) – this function takes the pointer to the root node, and key as input, and returns the pointer to the node where key is present. If key is not present in the AVL tree, then the code should output an error message.

(b) insert(root, key) – this function takes the pointer to the root node, and key as input, and inserts the node at the appropriate position. During insertion, whenever a rotation (single or double) occurs to maintain AVL balance, it should be counted and recorded.

(c) delete(root, key) – this function takes the pointer to the root node, and key as input, and deletes the corresponding node. During deletion, whenever a rotation (single or double) occurs to maintain AVL balance, it should be counted and recorded.

(d) After performing all the operations, output:

- In-order traversal of the final tree.
- Total number of single rotations performed (LL or RR).
- Total number of double rotations performed (LR or RL).

Input Format:

```
N
op1 x1
op2 x2
...
opN xN
```

Each operation op_i is either:

- 'I" for insertion (e.g., I 15)
- 'D" for deletion (e.g., D 10)

Output Format (printed once after all N operations):

```
Inorder: <list of keys in ascending order>
Single rotations: <count>
Double rotations: <count>
```

Rotation Definitions for AVL Trees:

- **Single Rotation:** Used when a node becomes unbalanced in one direction, and the imbalance can be fixed by a **single rotation**.
  - **Left-Left (LL) case:** Right rotation on the unbalanced node.
  - **Right-Right (RR) case:** Left rotation on the unbalanced node.
- **Double Rotation:** Used when a node becomes unbalanced, but the imbalance is caused by the opposite child of the heavy subtree. Requires **two rotations**.
  - **Left-Right (LR) case:** Left rotation on left child, then right rotation on the unbalanced node.
  - **Right-Left (RL) case:** Right rotation on right child, then left rotation on the unbalanced node.

Example:

Input:

```
7
I 10
I 20
I 30
I 25
I 28
D 10
D 25
```

Output:

```
Inorder: 20 28 30
Single rotations: 3
Double rotations: 1
```

2+3+3+2=10 Marks

5. Consider the problem of augmenting `red-black`-trees with an operation $RB-ENUMERATE(x, a, b)$ that output all keys k such that $a \leq k \leq b$ in a `red-black` tree rooted x. Write an algorithm that implements $RB-ENUMERATE(x, a, b)$ in $\Theta(m+\log n)$ time, where m is the number of keys that are outputted, and n is the number of internal nodes in the tree.

   Also, give a write to prove the correctness and efficiency of your algorithm.

   `Hint: See Theorem 14.1 of CLRS book.`

   Note: The global coding I/O specifications do not apply in this question.

   7 (Coding) +3 (theory)=10 Marks

2 Theory questions:

1. Let us define a relaxed `red-black` tree as a binary search tree that satisfies the following properties. In other words, the `root` may be either red or black. Consider a relaxed `red-black` tree T whose root is `red`. If we color the `root` of T black but make no other changes to `T`, is the resulting tree a `red-black` tree?

   (a) Every node is either `red` or `black`.

   (b) Every `leaf` (NIL) is `black`.

   (c) If a node is `red`, then both its children are `black`.

   (d) For each node, all simple paths from the node to descendant leaves contain the same number of `black` nodes.

   5 Marks

2. Show that the longest simple path from a node x in a `red-black` tree to a descendant leaf has a length at most twice that of the shortest simple path from node x to a descendant leaf. 5 Marks

3. Give a scenario (specific node heights or shape) where a *double rotation* is necessary during insertion in an AVL tree. Explain why a single rotation cannot restore balance in that case. 5 Marks

   Instructions on theory problems: Please neatly write the solution using pen-paper 1) to give a proof of the algorithm in Question 5 in Section 1, and 2) Questions 1, 2 and 3 of this section; and submit the scan copy in the google classroom page.