Assignment 7 — Solutions for Parts 1, 3, and 4

Sean Balbale

April 15, 2025

Part 1: AVL Trees

In this part we work with AVL Trees.

(a) AVL Tree Insertion

We insert the keys in the order:

$$9, 27, 50, 15, 2, 21, 36.$$

Below we detail the process step by step.

Step 1. Insert 9:

The tree is initially empty so 9 becomes the root.

$$9$$

Step 2. Insert 27:

Since $27 > 9$, insert 27 as the right child of 9.

$$9 — 27$$

Step 3. Insert 50:

Following BST rules, 50 is placed as the right child of 27. Now the tree appears as:

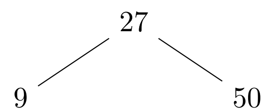
$$9 \to 27 \to 50.$$

This yields an imbalance at 9 (balance factor -2) because the right subtree has height 2. This is a **Right-Right case** requiring a **left rotation** at node 9.

After rotation the tree becomes:

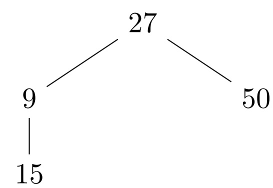
27

9 50



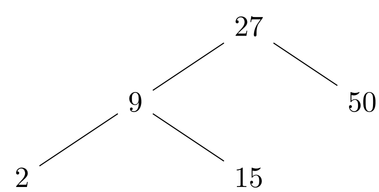
Step 4. Insert 15:

15 is less than 27 so we go left to 9; since $15 > 9$, insert 15 as the right child of 9.



Step 5. Insert 2:

2 is less than 27; at 9, since $2 < 9$, insert 2 as the left child of 9.



Step 6. Insert 21:

Traverse: $21 < 27$ (go left), then at node 9: $21 > 9$ (go right) to node 15, and $21 > 15$ so insert as right child of 15. Now the subtree rooted at 9 is:

9

2 15

21

This causes an imbalance at 27 (with a balance factor of +2) because node 9's right subtree (via node 15) becomes taller than its left subtree. Moreover, node 9 has a balance factor of -1 (right heavy). This is a **Left-Right (LR) case**. The remedy is to perform a *double rotation*: first a left rotation at node 9, then a right rotation at node 27.

Left Rotation at 9:

The subtree rooted at 9 becomes:

15

9 21

2

Right Rotation at 27:

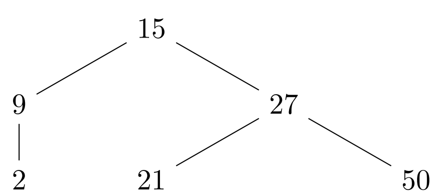
After this rotation the tree becomes:

15

9 27

2 21 50

Here 9 has left child 2, and 27 has left child 21 and right child 50.



Step 7. Insert 36:

Traverse: $36 > 15$, so move right to node 27; then $36 > 27$ so move to its right subtree; at 50, $36 < 50$, hence insert 36 as the left child of 50.

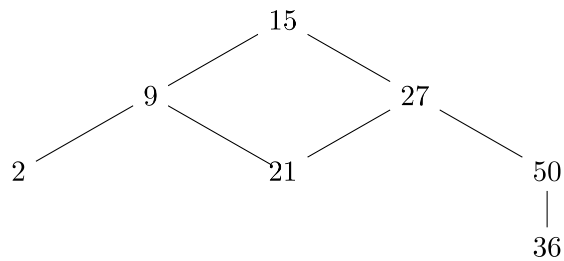
The final AVL tree is:

15

9 27

2 21 50

36

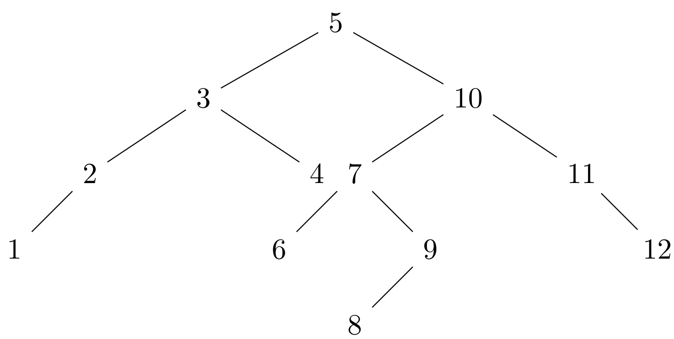


Summary of Part (a):

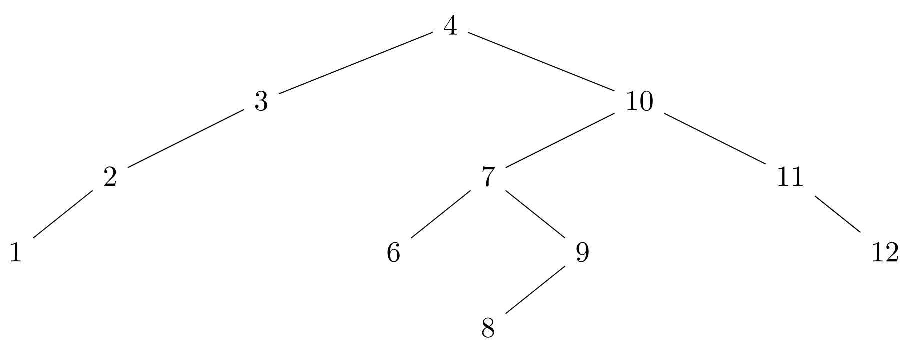
After inserting 9, 27, 50, 15, 2, 21, and 36—with appropriate rotations (a left rotation at 9 and a double LR rotation at 27)—the final balanced AVL tree is as shown above.

(b) Deletion (Before Rebalancing)

Consider the following given AVL tree:



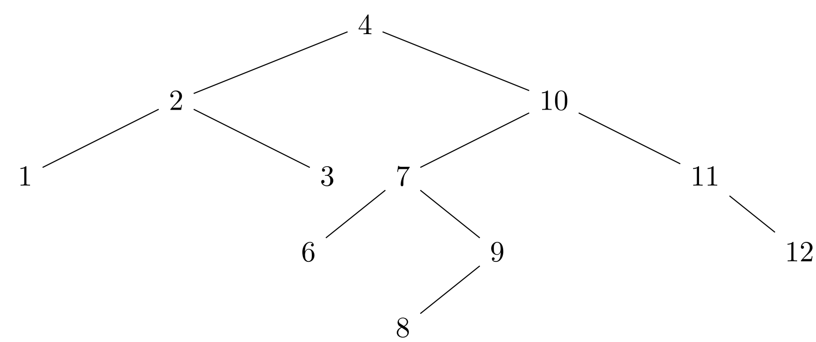
After deleting node 5, we replace it with its inorder predecessor (node 4). The resulting tree is:



(c) Rebalancing After Deletion

In the above tree, notice that the left subtree of the root shows an imbalance: node **3** has a balance factor of +2. This indicates a left–left (LL) case, which can be corrected by performing a right rotation at node **3**. After the rotation, node **2** becomes the new parent for that subtree, with node **1** as its left child and node **3** as its right child. The right subtree remains unchanged.

Below is the rebalanced tree (balance factors are omitted):



Part 3: Graph Traversals

Graph 1: Depth-First Search (DFS) Traversal

Given a graph where the adjacency lists (neighbors) are arranged in ascending order, we perform a DFS starting at node 1. The DFS is implemented using a stack, and the procedure is as follows:

Procedure Overview

1. **Initialization:**
   - Mark all nodes as unvisited.
   - Create an empty stack.
   - Push the start node (**1**) onto the stack and mark it as visited.
2. **While the stack is not empty:**
   - Let the top of the stack be the current node.
   - If the current node has an unvisited neighbor, push that neighbor onto the stack and mark it as visited.

- If the current node has no unvisited neighbors, pop it off the stack.

Step-by-Step DFS Execution

1. Start at node **1**. **Stack:** [1] **Visited Order:** 1
2. From node 1, the neighbors (in ascending order) are: $\{0, 2, 3, 5\}$. The first unvisited neighbor is **0** $\rightarrow$ push 0. **Stack:** [1, 0] **Visited Order:** 1, 0
3. At node 0, the only neighbor is $\{1\}$, which is already visited $\rightarrow$ pop node 0. **Stack:** [1]
4. Back at node 1, the next unvisited neighbor is **2** $\rightarrow$ push 2. **Stack:** [1, 2] **Visited Order:** 1, 0, 2
5. At node 2, assume its neighbors are $\{1, 4, 5\}$. The first unvisited neighbor is **4** $\rightarrow$ push 4. **Stack:** [1, 2, 4] **Visited Order:** 1, 0, 2, 4
6. At node 4, assume its neighbors are $\{2, 6\}$. The first unvisited neighbor is **6** $\rightarrow$ push 6. **Stack:** [1, 2, 4, 6] **Visited Order:** 1, 0, 2, 4, 6
7. At node 6, assume its neighbors are $\{4, 7\}$. The first unvisited neighbor is **7** $\rightarrow$ push 7. **Stack:** [1, 2, 4, 6, 7] **Visited Order:** 1, 0, 2, 4, 6, 7
8. At node 7, suppose its only neighbor is $\{6\}$ (already visited) $\rightarrow$ pop node 7. **Stack:** [1, 2, 4, 6]
9. Back at node 6, all neighbors have been visited $\rightarrow$ pop node 6. **Stack:** [1, 2, 4]
10. Back at node 4, all neighbors have been visited $\rightarrow$ pop node 4. **Stack:** [1, 2]
11. Returning to node 2, the next unvisited neighbor is **5** $\rightarrow$ push 5. **Stack:** [1, 2, 5] **Visited Order:** 1, 0, 2, 4, 6, 7, 5
12. At node 5, if its neighbors are $\{1, 2\}$ (both visited) $\rightarrow$ pop node 5. **Stack:** [1, 2]
13. Back at node 2, no more unvisited neighbors $\rightarrow$ pop node 2. **Stack:** [1]

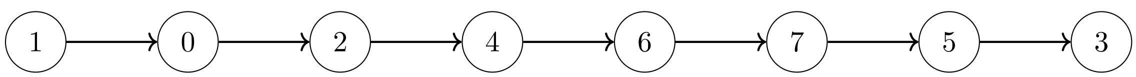
14. Back at node 1, the final unvisited neighbor is **3** $\rightarrow$ push 3. **Stack:** [1, 3] **Visited Order:** 1, 0, 2, 4, 6, 7, 5, 3
15. At node 3, if the only neighbor is $\{1\}$ (already visited) $\rightarrow$ pop node 3. **Stack:** [1]
16. Back at node 1, all neighbors are now visited $\rightarrow$ pop node 1. **Stack:** Empty.

Thus, the DFS traversal order is:

$$\boxed{1 \rightarrow 0 \rightarrow 2 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 5 \rightarrow 3.}$$

Visual Representation of DFS Order

The following diagram illustrates the final order in which nodes are visited.



Final DFS Visitation Order: $\boxed{1,\ 0,\ 2,\ 4,\ 6,\ 7,\ 5,\ 3.}$

Graph 2: Breadth-First Search (BFS) Traversal

In this example, we perform a BFS traversal on a graph starting from node 0. We assume that for each node the neighbors are considered in ascending numerical order. The BFS traversal is implemented using a queue as follows:

BFS Procedure:

1. **Initialization:**
   - Mark all nodes as unvisited.
   - Create an empty queue.
   - Enqueue the start node (node **0**) and mark it as visited.
2. **Iteration:** While the queue is not empty:
   - Dequeue the front of the queue (let this be the current node).
   - For each neighbor of the current node (in ascending order):
     - If the neighbor is unvisited, mark it as visited and enqueue it.

Step-by-Step Execution:

1. **Start at node 0:** Enqueue **0**.
   Visited set: $\{0\}$
   Queue: [0]
   BFS Order: 0

2. **Dequeue 0:** Suppose the neighbors of 0 are $\{1, 2\}$.
   Enqueue 1 and 2 (both unvisited).
   Visited set: $\{0, 1, 2\}$
   Queue: [1, 2]
   BFS Order: 0

3. **Dequeue 1:** Suppose the neighbors of 1 are $\{0, 2, 3, 5\}$.
   Nodes 0 and 2 are already visited.
   Enqueue 3 and 5.
   Visited set: $\{0, 1, 2, 3, 5\}$
   Queue: [2, 3, 5]
   BFS Order: 0, 1

4. **Dequeue 2:** Suppose the neighbors of 2 are $\{1, 4, 6\}$.
   1 is already visited; enqueue 4 and 6.
   Visited set: $\{0, 1, 2, 3, 4, 5, 6\}$
   Queue: [3, 5, 4, 6]
   BFS Order: 0, 1, 2

5. **Dequeue 3:** Suppose node 3's neighbor(s) are $\{1\}$.
   1 is already visited.
   Queue: [5, 4, 6]
   BFS Order: 0, 1, 2, 3

6. **Dequeue 5:** Suppose the neighbors of 5 are $\{1, 2, 3\}$.
   All are visited.
   Queue: [4, 6]
   BFS Order: 0, 1, 2, 3, 5

7. **Dequeue 4:** Suppose the neighbors of 4 are $\{2\}$.
   2 is visited.
   Queue: [6]
   BFS Order: 0, 1, 2, 3, 5, 4

8. **Dequeue 6:** Suppose the neighbors of 6 are $\{7\}$.
   Enqueue 7 (if unvisited).

Visited set: $\{0, 1, 2, 3, 4, 5, 6, 7\}$
Queue: [7]
BFS Order: 0, 1, 2, 3, 5, 4, 6

9. **Deque 7:** Suppose node 7 has no unvisited neighbors.
   Queue: []
   BFS Order: 0, 1, 2, 3, 5, 4, 6, 7

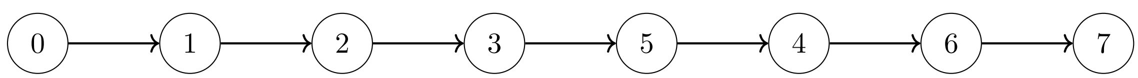
Final BFS Visitation Order:

$$\boxed{0 \to 1 \to 2 \to 3 \to 5 \to 4 \to 6 \to 7.}$$

Unreachable Vertices:
Since every node $\{0, 1, 2, 3, 4, 5, 6, 7\}$ was visited during the BFS, there are no vertices unreachable from node 0.

Visual Representation of BFS Order

The following TikZ diagram illustrates the BFS order using arrows:



Conclusion:
The BFS traversal starting at node 0 visits the nodes in the order

$$\boxed{0,\ 1,\ 2,\ 3,\ 5,\ 4,\ 6,\ 7,}$$

and all vertices of the graph are reachable from node 0.

1 Part 4: Graph Representations

Assume we are given a directed graph with 5 vertices, labeled $0, 1, 2, 3, 4$, and the following edges (with weights):

- $0 \to 1$ (weight 3)
- $0 \to 4$ (weight 4)
- $4 \to 1$ (weight 1)
- $1 \to 3$ (weight 3)
- $2 \to 4$ (weight 1)
- $2 \to 3$ (weight 7)

1. Graph Representations

(a) Adjacency Matrix

The adjacency matrix $A = [a_{ij}]$ is constructed with rows and columns labeled $0, 1, 2, 3, 4$ such that

$$a_{ij} = \begin{cases} \text{edge weight } w, & \text{if there is an edge } i \to j, \\ 0, & \text{otherwise.} \end{cases}$$

For our graph, the matrix is:

	0	1	2	3	4
0	0	3	0	0	4
1	0	0	0	3	0
2	0	0	0	7	1
3	0	0	0	0	0
4	0	1	0	0	0

(b) Adjacency Graph (Adjacency List)

The adjacency list stores for each vertex a linked list of all outgoing edges. For our graph the lists are:

- **Vertex 0:** $0 \to 1$ (weight 3), $0 \to 4$ (weight 4)
- **Vertex 1:** $1 \to 3$ (weight 3)
- **Vertex 2:** $2 \to 4$ (weight 1), $2 \to 3$ (weight 7)
- **Vertex 3:** No outgoing edges.
- **Vertex 4:** $4 \to 1$ (weight 1)

Each edge node stores:

- Destination vertex index: 2 bytes
- Edge weight: 2 bytes
- Pointer to the next edge: 4 bytes

Thus, each edge node uses $2 + 2 + 4 = 8$ bytes.

Also, a vertex record (in the vertex array) stores:

- Vertex index: 2 bytes
- Pointer to its edge list: 4 bytes

That is $2 + 4 = 6$ bytes per vertex.

2. Memory Requirements

We are given the following size assumptions:

- Vertex index: 2 bytes
- Pointer: 4 bytes
- Edge weight: 2 bytes

(a) Directed Graph

Adjacency Matrix: The matrix has $n = 5$ vertices, so there are $5 \times 5 = 25$ entries.

$$\text{Total Memory} = 25 \times 2 = 50 \text{ bytes.}$$

Adjacency List:

- **Vertex Array:** 5 vertices $\times$ 6 bytes $= 30$ bytes.
- **Edge Nodes:** There are 6 edges. Each edge node uses 8 bytes.

$$6 \times 8 = 48 \text{ bytes.}$$

So, the total memory for the adjacency list is:

$$30 + 48 = 78 \text{ bytes.}$$

(b) Undirected Graph

For an undirected graph, each edge is stored twice in the adjacency list.

Adjacency Matrix: The matrix remains unchanged:

$$25 \times 2 = 50 \text{ bytes.}$$

Adjacency List:

- **Vertex Array:** Remains 30 bytes.
- **Edge Nodes:** Each of the original 6 edges appears twice, so there are $6 \times 2 = 12$ edge nodes.

$$12 \times 8 = 96 \text{ bytes.}$$

Thus, total memory is:

$$30 + 96 = 126 \text{ bytes.}$$

Summary of Results

Directed Graph:

- **Adjacency Matrix:** 50 bytes.
- **Adjacency List:** 78 bytes.

Undirected Graph:

- **Adjacency Matrix:** 50 bytes.
- **Adjacency List:** 126 bytes.