LAB-9

Binary Trees, BSTs & AVL in C

Q1) Write a program to construct a binary tree using user input (level-order insertion). Perform preorder, inorder, and postorder traversals.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
   int data;
   struct Node *left, *right;
} Node;

Node* createNode(int x) {
   if(x == -1) return NULL;
   Node* n = (Node*)malloc(sizeof(Node));
   n->data = x;
   n->left = n->right = NULL;
   return n;
}

// Build tree from level-order array
Node* build(int arr[], int n) {
   if(n == 0) return NULL;

   Node* nodes[n];
   for(int i=0; i<n; i++)
      nodes[i] = createNode(arr[i]);

   for(int i=0; i<n; i++) {
      if(nodes[i] != NULL) {
         int l = 2*i + 1;
         int r = 2*i + 2;
         if(l < n) nodes[i]->left = nodes[l];
         if(r < n) nodes[i]->right = nodes[r];
      }
   }
   return nodes[0];
```

```
}

// Traversals
void preorder(Node* root) {
   if(!root) return;
   printf("%d ", root->data);
   preorder(root->left);
   preorder(root->right);
}

void inorder(Node* root) {
   if(!root) return;
   inorder(root->left);
   printf("%d ", root->data);
   inorder(root->right);
}

void postorder(Node* root) {
   if(!root) return;
   postorder(root->left);
   postorder(root->right);
   printf("%d ", root->data);
}

int main() {
   int n;
   printf("Enter number of nodes: ");
   scanf("%d", &n);

   int arr[n];
   printf("Enter values in level order (-1 for NULL):\n");
   for(int i=0; i<n; i++)
      scanf("%d", &arr[i]);

   Node* root = build(arr, n);

   printf("\nPreorder: ");
   preorder(root);

   printf("\nInorder: ");
   inorder(root);
```

```
    printf("\nPostorder: ");
    postorder(root);

    printf("\n");
    return 0;
}
```

Input & Output

```
Enter number of nodes: 3
Enter values in level order (-1 for NULL):
20  20
2034
3420


Preorder: 20 2034 3420
Inorder: 2034 20 3420
Postorder: 2034 3420 20
```

Q2) Given a binary tree, write functions to compute:
a) Height of the tree b) Total number of leaf nodes

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

Node* createNode(int x) {
    if(x == -1) return NULL;
    Node* n = (Node*)malloc(sizeof(Node));
    n->data = x;
    n->left = n->right = NULL;
    return n;
}
```

```
// Build binary tree from level-order input
Node* buildTree(int arr[], int n) {
   if(n == 0) return NULL;

   Node* nodes[n];
   for(int i = 0; i < n; i++)
      nodes[i] = createNode(arr[i]);

   for(int i = 0; i < n; i++) {
      if(nodes[i] != NULL) {
         int left = 2*i + 1;
         int right = 2*i + 2;
         if(left < n) nodes[i]->left = nodes[left];
         if(right < n) nodes[i]->right = nodes[right];
      }
   }
   return nodes[0];
}

// Height of tree
int height(Node* root) {
   if(root == NULL) return 0;
   int lh = height(root->left);
   int rh = height(root->right);
   return (lh > rh ? lh : rh) + 1;
}

// Count leaf nodes
int countLeaves(Node* root) {
   if(root == NULL) return 0;
   if(root->left == NULL && root->right == NULL) return 1;
   return countLeaves(root->left) + countLeaves(root->right);
}

int main() {
   int n;
   printf("Enter number of nodes: ");
   scanf("%d", &n);

   int arr[n];
   printf("Enter %d node values in level order (-1 for NULL):\n", n);
   for(int i = 0; i < n; i++){
```

```
    scanf("%d", &arr[i]);
  }

  Node* root = buildTree(arr, n);

  printf("\nHeight of Tree: %d\n", height(root));
  printf("Leaf Nodes: %d\n", countLeaves(root));

  return 0;
}
```

Input & Output

```
Enter number of nodes: 3
Enter 3 node values in level order (-1 for NULL):
20
34
3420


Height of Tree: 2
Leaf Nodes: 2
```

Q3) Given any binary tree, write a program to verify whether it satisfies BST properties using the min–max recursive method.

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

typedef struct Node {
  int data;
  struct Node *left, *right;
} Node;

Node* createNode(int val) {
  if (val == -1) return NULL;
  Node* n = (Node*)malloc(sizeof(Node));
  n->data = val;
  n->left = n->right = NULL;
```

```
    return n;
}

// Build tree from level-order array
Node* buildTree(int arr[], int n) {
    if (n == 0) return NULL;

    Node* nodes[n];
    for (int i = 0; i < n; i++)
        nodes[i] = createNode(arr[i]);

    for (int i = 0; i < n; i++) {
        if (nodes[i] != NULL) {
            int l = 2*i + 1;
            int r = 2*i + 2;
            if (l < n) nodes[i]->left = nodes[l];
            if (r < n) nodes[i]->right = nodes[r];
        }
    }
    return nodes[0];
}

// Check if BST using min–max method
int isBST(Node* root, int min, int max) {
    if (root == NULL) return 1;

    if (root->data <= min || root->data >= max)
        return 0;

    return isBST(root->left, min, root->data) &&
           isBST(root->right, root->data, max);
}

int main() {
    int n;
    printf("Enter number of nodes: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d values in level order (-1 for NULL):\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);
```

```
    Node* root = buildTree(arr, n);

    if (isBST(root, INT_MIN, INT_MAX))
        printf("\nThis is a BST.\n");
    else
        printf("\nThis is NOT a BST.\n");

    return 0;
}
```

Input & Output

```
Enter number of nodes: 3
Enter 3 values in level order (-1 for NULL):
3420
3 2034
20


This is NOT a BST.
```

Q4) Perform level-order (breadth-first) traversal of a binary tree using a queue.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

typedef struct Queue {
    Node* arr[100];
    int front, rear;
} Queue;

void initQueue(Queue* q) {
    q->front = q->rear = -1;
}
```

```
int isEmpty(Queue* q) {
   return q->front == -1;
}

void enqueue(Queue* q, Node* x) {
   if (q->rear == 99) return;
   if (q->front == -1) q->front = 0;
   q->arr[++q->rear] = x;
}

Node* dequeue(Queue* q) {
   if (isEmpty(q)) return NULL;
   Node* x = q->arr[q->front];
   if (q->front == q->rear)
      q->front = q->rear = -1;
   else
      q->front++;
   return x;
}

Node* createNode(int val) {
   if (val == -1) return NULL;
   Node* n = (Node*)malloc(sizeof(Node));
   n->data = val;
   n->left = n->right = NULL;
   return n;
}

// Build binary tree from level-order
Node* buildTree(int arr[], int n) {
   if (n == 0) return NULL;

   Node* nodes[n];
   for (int i = 0; i < n; i++)
      nodes[i] = createNode(arr[i]);

   for (int i = 0; i < n; i++) {
      if (nodes[i] != NULL) {
         int l = 2*i + 1;
         int r = 2*i + 2;
         if (l < n) nodes[i]->left = nodes[l];
```

```
        if (r < n) nodes[i]->right = nodes[r];
      }
    }
    return nodes[0];
}

// Level-Order Traversal using queue
void levelOrder(Node* root) {
    if (root == NULL) return;

    Queue q;
    initQueue(&q);
    enqueue(&q, root);

    printf("\nLevel-order: ");

    while (!isEmpty(&q)) {
        Node* cur = dequeue(&q);
        printf("%d ", cur->data);

        if (cur->left)  enqueue(&q, cur->left);
        if (cur->right) enqueue(&q, cur->right);
    }
}

int main() {
    int n;
    printf("Enter number of nodes: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter %d values in level order (-1 for NULL):\n", n);
    for (int i = 0; i < n; i++)
        scanf("%d", &arr[i]);

    Node* root = buildTree(arr, n);

    levelOrder(root);

    return 0;
}
```

Input & Output

```
Enter number of nodes: 5
Enter 5 values in level order (-1 for NULL):
20
34
2034
3420
680


Level-order: 20 34 2034 3420 680 [?2004h
```

Q5) Insert nodes into a Binary Search Tree (BST) and display its inorder traversal.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
   int data;
   struct Node *left, *right;
} Node;

Node* createNode(int val) {
   Node* n = (Node*)malloc(sizeof(Node));
   n->data = val;
   n->left = n->right = NULL;
   return n;
}

// Insert into BST
Node* insert(Node* root, int val) {
   if (root == NULL)
      return createNode(val);

   if (val < root->data)
      root->left = insert(root->left, val);
   else
```

```
        root->right = insert(root->right, val);

    return root;
}

// Inorder traversal
void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    printf("%d ", root->data);
    inorder(root->right);
}

int main() {
    int n, val;
    Node* root = NULL;

    printf("Enter number of nodes: ");
    scanf("%d", &n);

    printf("Enter %d values:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        root = insert(root, val);
    }

    printf("\nInorder Traversal (Sorted): ");
    inorder(root);
    printf("\n");

    return 0;
}
```

Input & Output

```
Enter number of nodes: 5

Enter 5 values:

34

20

3420

2034
```

```
68- 0


Inorder Traversal (Sorted): 20 34 680 2034 3420
```

Q6) Given a BST, write functions to find:
A) k-th smallest element B) k-th largest element
(Without storing traversal in an array)

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
   int data;
   struct Node *left, *right;
} Node;

Node* createNode(int val) {
   Node* n = (Node*)malloc(sizeof(Node));
   n->data = val;
   n->left = n->right = NULL;
   return n;
}

Node* insert(Node* root, int val) {
   if (root == NULL) return createNode(val);
   if (val < root->data)
      root->left = insert(root->left, val);
   else
      root->right = insert(root->right, val);
   return root;
}

// ------------------ K-th SMALLEST ------------------

int kthSmallUtil(Node* root, int *k) {
   if (root == NULL) return -1;

   int left = kthSmallUtil(root->left, k);
   if (left != -1) return left;
```

```
    (*k)--;
    if (*k == 0) return root->data;

    return kthSmallUtil(root->right, k);
}

int kthSmallest(Node* root, int k) {
    return kthSmallUtil(root, &k);
}

// ------------------ K-th LARGEST ------------------

int kthLargeUtil(Node* root, int *k) {
    if (root == NULL) return -1;

    int right = kthLargeUtil(root->right, k);
    if (right != -1) return right;

    (*k)--;
    if (*k == 0) return root->data;

    return kthLargeUtil(root->left, k);
}

int kthLargest(Node* root, int k) {
    return kthLargeUtil(root, &k);
}


int main() {
    int n, val, k;
    Node* root = NULL;

    printf("Enter number of nodes: ");
    scanf("%d", &n);

    printf("Enter %d values:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &val);
        root = insert(root, val);
    }
```

```
    printf("Enter k for k-th smallest: ");
    scanf("%d", &k);
    printf("k-th Smallest = %d\n", kthSmallest(root, k));

    printf("Enter k for k-th largest: ");
    scanf("%d", &k);
    printf("k-th Largest = %d\n", kthLargest(root, k));

    return 0;
}
```

Input & Output

```
Enter number of nodes: 5
Enter 5 values:
34
4 4 3420
2034
680
20
Enter k for k-th smallest: 3
k-th Smallest = 680
Enter k for k-th largest: 2
k-th Largest = 2034
```

Q7) Given a sorted array, build a balanced BST using divide-and-conquer. Print level-order traversal of the new tree.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node{
    int data;
    struct Node *left,*right;
} Node;

Node* newNode(int x){
```

```
   Node* p = (Node*)malloc(sizeof(Node));
   p->data = x;
   p->left = p->right = NULL;
   return p;
}

Node* buildBalancedBST(int arr[], int l, int r){
   if(l>r) return NULL;
   int mid = (l + r) / 2;
   Node* root = newNode(arr[mid]);
   root->left  = buildBalancedBST(arr, l, mid-1);
   root->right = buildBalancedBST(arr, mid+1, r);
   return root;
}

void levelOrder(Node* root, int n){
   if(root==NULL){ printf("Tree is empty.\n"); return; }
   Node** q = (Node**)malloc(sizeof(Node*) * n);
   int front = 0, rear = 0;
   q[rear++] = root;
   while(front < rear){
      Node* cur = q[front++];
      printf("%d ", cur->data);
      if(cur->left)  q[rear++] = cur->left;
      if(cur->right) q[rear++] = cur->right;
   }
   printf("\n");
   free(q);
}

int main(){
   int n;
   printf("Enter number of elements (n): ");
   if(scanf("%d",&n)!=1 || n<=0){ printf("Invalid n\n"); return 0; }

   int *arr = (int*)malloc(sizeof(int)*n);
   printf("Enter %d sorted integers (ascending) separated by space/newline:\n",
n);
   for(int i=0;i<n;i++){
      if(scanf("%d",&arr[i])!=1){ printf("Invalid input\n"); free(arr); return 0; }
   }
```

```
    Node* root = buildBalancedBST(arr, 0, n-1);
    printf("Level-order traversal of the balanced BST:\n");
    levelOrder(root, n);

    free(arr);
    return 0;
}
```

Input & Output

```
Enter number of elements (n): 5
Enter 5 sorted integers (ascending) separated by space/newline:
20 34      0
34
2034
3420
6800
Level-order traversal of the balanced BST:
2034 20 3420 34 6800
```

Q8) Given a binary tree, write a program to check whether it is height-balanced (AVL property). Also print the balance factor of each node.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node *left, *right;
} Node;

Node* newNode(int x) {
    Node* p = (Node*)malloc(sizeof(Node));
    p->data = x;
    p->left = p->right = NULL;
    return p;
}
```

```
/* ---------- LEVEL ORDER INSERTION ---------- */
Node* insertLevelOrder(Node* root, int x) {
   Node* node = newNode(x);

   if (root == NULL)
      return node;

   // queue for BFS
   Node* q[1000];
   int front = 0, rear = 0;

   q[rear++] = root;

   while (front < rear) {
      Node* temp = q[front++];

      if (temp->left == NULL) {
         temp->left = node;
         break;
      }
      else if (temp->right == NULL) {
         temp->right = node;
         break;
      }
      else {
         q[rear++] = temp->left;
         q[rear++] = temp->right;
      }
   }

   return root;
}

/* ---------- HEIGHT ---------- */
int height(Node* root) {
   if (!root) return 0;

   int lh = height(root->left);
   int rh = height(root->right);

   return 1 + (lh > rh ? lh : rh);
}
```

```
/* ---------- CHECK BALANCE & PRINT BALANCE FACTOR ---------- */
int checkBalanced(Node* root) {
   if (!root) return 1;

   int lh = height(root->left);
   int rh = height(root->right);

   int bf = lh - rh;
   printf("Node %d → Balance Factor = %d\n", root->data, bf);

   if (abs(bf) > 1)
      return 0;

   return checkBalanced(root->left) && checkBalanced(root->right);
}

int main() {
   int n, x;

   printf("Enter number of nodes: ");
   scanf("%d", &n);

   if (n <= 0) {
      printf("Invalid size\n");
      return 0;
   }

   Node* root = NULL;

   printf("Enter %d values (level-order):\n", n);
   for (int i = 0; i < n; i++) {
      scanf("%d", &x);
      root = insertLevelOrder(root, x);
   }

   printf("\nBalance Factors:\n");
   int ok = checkBalanced(root);

   printf("\nTree is %s\n", ok ? "HEIGHT-BALANCED (AVL)" : "NOT
BALANCED");
```

```
    return 0;
}
```

Input & Output

```
Enter number of nodes: 5
Enter 5 values (level-order):
20
34
680
2034
3420

Balance Factors:
Node 20 → Balance Factor = 1
Node 34 → Balance Factor = 0
Node 2034 → Balance Factor = 0
Node 3420 → Balance Factor = 0
Node 680 → Balance Factor = 0

Tree is HEIGHT-BALANCED (AVL)
```

Q9) Implement AVL Tree insertion, after each insertion display:
A) Balance Factor B) Rotation Performed

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    int data;
    struct Node *left, *right;
    int height;
} Node;

/* ---------- CREATE NODE ---------- */
Node* newNode(int x) {
```

```
    Node* p = (Node*)malloc(sizeof(Node));
    p->data = x;
    p->left = p->right = NULL;
    p->height = 1;
    return p;
}

/* ---------- HEIGHT FUNCTION ---------- */
int height(Node* root) {
    return root ? root->height : 0;
}

/* ---------- BALANCE FACTOR ---------- */
int getBF(Node* root) {
    if (!root) return 0;
    return height(root->left) - height(root->right);
}

/* ---------- ROTATIONS ---------- */
Node* rightRotate(Node* y) {
    printf("Rotation: Right Rotation (LL Case)\n");

    Node* x = y->left;
    Node* T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = 1 + (height(y->left) > height(y->right) ?
                height(y->left) : height(y->right));
    x->height = 1 + (height(x->left) > height(x->right) ?
                height(x->left) : height(x->right));

    return x;
}

Node* leftRotate(Node* x) {
    printf("Rotation: Left Rotation (RR Case)\n");

    Node* y = x->right;
    Node* T2 = y->left;
```

```
  y->left = x;
  x->right = T2;

  x->height = 1 + (height(x->left) > height(x->right) ?
            height(x->left) : height(x->right));
  y->height = 1 + (height(y->left) > height(y->right) ?
            height(y->left) : height(y->right));

  return y;
}

/* ---------- INSERT IN AVL TREE ---------- */
Node* insert(Node* root, int key) {
  if (root == NULL) {
    printf("Inserted %d\n", key);
    return newNode(key);
  }

  if (key < root->data)
    root->left = insert(root->left, key);
  else if (key > root->data)
    root->right = insert(root->right, key);
  else {
    printf("Duplicate ignored: %d\n", key);
    return root;
  }

  root->height = 1 + (height(root->left) > height(root->right) ?
              height(root->left) : height(root->right));

  int bf = getBF(root);

  /* ---------- ROTATION CASES ---------- */
  if (bf > 1 && key < root->left->data) {
    return rightRotate(root);       // LL
  }

  if (bf < -1 && key > root->right->data) {
    return leftRotate(root);        // RR
  }

  if (bf > 1 && key > root->left->data) {  // LR
```

```
        printf("Rotation: Left-Right Rotation (LR Case)\n");
        root->left = leftRotate(root->left);
        return rightRotate(root);
    }

    if (bf < -1 && key < root->right->data) { // RL
        printf("Rotation: Right-Left Rotation (RL Case)\n");
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}

void printBF(Node* root) {
    if (!root) return;
    printBF(root->left);
    printf("Node %d → BF = %d\n", root->data, getBF(root));
    printBF(root->right);
}

int main() {
    int n, x;
    Node* root = NULL;

    printf("Enter number of insertions: ");
    scanf("%d", &n);

    printf("Enter %d values:\n", n);

    for (int i = 0; i < n; i++) {
        scanf("%d", &x);
        printf("\nInserting %d\n", x);
        root = insert(root, x);

        printf("Balance Factors now:\n");
        printBF(root);
        printf("\n");
    }

    return 0;
}
```

Input & Output

Enter number of insertions: 5

Enter 5 values:

20

Inserting 20

Inserted 20

Balance Factors now:

Node 20 → BF = 0

34

Inserting 34

Inserted 34

Balance Factors now:

Node 20 → BF = -1

Node 34 → BF = 0

680

Inserting 680

Inserted 680

Rotation: Left Rotation (RR Case)

Balance Factors now:

Node 20 → BF = 0

Node 34 → BF = 0

Node 680 → BF = 0

2034

Inserting 2034

Inserted 2034

```
Balance Factors now:
Node 20 → BF = 0
Node 34 → BF = -1
Node 680 → BF = -1
Node 2034 → BF = 0


3420


Inserting 3420
Inserted 3420
Rotation: Left Rotation (RR Case)
Balance Factors now:
Node 20 → BF = 0
Node 34 → BF = -1
Node 680 → BF = 0
Node 2034 → BF = 0
Node 3420 → BF = 0
```

Q10)Write a program to implement AVL tree deletion, after each deletion print: A) Type of Rotation B) Height of the tree

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
   int data;
   struct Node *left, *right;
   int height;
} Node;

/* Create Node */
Node* newNode(int x) {
   Node* n = (Node*)malloc(sizeof(Node));
   n->data = x;
   n->left = n->right = NULL;
```

```
    n->height = 1;
    return n;
}

/* Height */
int height(Node* root) {
    return root ? root->height : 0;
}

/* Balance Factor */
int BF(Node* root) {
    return height(root->left) - height(root->right);
}

/* Update Height */
void updateHeight(Node* root) {
    root->height = 1 + (height(root->left) > height(root->right)
                        ? height(root->left) : height(root->right));
}

/* ---------- Rotations ---------- */
Node* rightRotate(Node* y) {
    printf("Rotation: Right (LL)\n");
    Node* x = y->left;
    Node* T2 = x->right;
    x->right = y;
    y->left = T2;

    updateHeight(y);
    updateHeight(x);

    return x;
}

Node* leftRotate(Node* x) {
    printf("Rotation: Left (RR)\n");
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;

    updateHeight(x);
```

```
  updateHeight(y);

  return y;
}

/* ---------- AVL Insert ---------- */
Node* insert(Node* root, int key) {
  if (!root) return newNode(key);

  if (key < root->data) root->left = insert(root->left, key);
  else if (key > root->data) root->right = insert(root->right, key);
  else return root;  // no duplicates

  updateHeight(root);

  int bf = BF(root);

  // LL
  if (bf > 1 && key < root->left->data)
    return rightRotate(root);

  // RR
  if (bf < -1 && key > root->right->data)
    return leftRotate(root);

  // LR
  if (bf > 1 && key > root->left->data) {
    printf("Rotation: Left-Right (LR)\n");
    root->left = leftRotate(root->left);
    return rightRotate(root);
  }

  // RL
  if (bf < -1 && key < root->right->data) {
    printf("Rotation: Right-Left (RL)\n");
    root->right = rightRotate(root->right);
    return leftRotate(root);
  }

  return root;
}
```

```
/* Find inorder successor */
Node* minValueNode(Node* root) {
  while (root->left) root = root->left;
  return root;
}

/* ---------- AVL Delete ---------- */
Node* deleteNode(Node* root, int key) {
  if (!root) return root;

  if (key < root->data) root->left = deleteNode(root->left, key);
  else if (key > root->data) root->right = deleteNode(root->right, key);
  else {
    if (!root->left || !root->right) {
      Node* temp = root->left ? root->left : root->right;
      if (!temp) {
        free(root);
        return NULL;
      } else {
        *root = *temp;
        free(temp);
      }
    } else {
      Node* temp = minValueNode(root->right);
      root->data = temp->data;
      root->right = deleteNode(root->right, temp->data);
    }
  }

  if (!root) return root;

  updateHeight(root);

  int bf = BF(root);

  // LL
  if (bf > 1 && BF(root->left) >= 0)
    return rightRotate(root);

  // LR
  if (bf > 1 && BF(root->left) < 0) {
    printf("Rotation: Left-Right (LR)\n");
```

```
    root->left = leftRotate(root->left);
    return rightRotate(root);
  }

  // RR
  if (bf < -1 && BF(root->right) <= 0)
    return leftRotate(root);

  // RL
  if (bf < -1 && BF(root->right) > 0) {
    printf("Rotation: Right-Left (RL)\n");
    root->right = rightRotate(root->right);
    return leftRotate(root);
  }

  return root;
}

/* ---------- Print Inorder ---------- */
void inorder(Node* root) {
  if (!root) return;
  inorder(root->left);
  printf("%d ", root->data);
  inorder(root->right);
}

int main() {
  Node* root = NULL;
  int n, x, d;

  /* Insert section */
  printf("Enter number of insertions: ");
  scanf("%d", &n);

  printf("Enter values:\n");
  for (int i = 0; i < n; i++) {
    scanf("%d", &x);
    printf("\nInserting %d...\n", x);
    root = insert(root, x);
    printf("Current tree height: %d\n", height(root));
  }
```

```
    /* Delete section */
    printf("\nEnter number of deletions: ");
    scanf("%d", &d);

    printf("Enter values to delete:\n");
    for (int i = 0; i < d; i++) {
        scanf("%d", &x);
        printf("\nDeleting %d...\n", x);
        root = deleteNode(root, x);
        printf("Height after deletion: %d\n", height(root));
    }

    printf("\nFinal Tree (Inorder): ");
    inorder(root);
    printf("\n");

    return 0;
}
```

Input & Output

```
Enter number of insertions: 20  5
Enter values:
20

Inserting 20...
Current tree height: 1
34

Inserting 34...
Current tree height: 2
2034

Inserting 2034...
Rotation: Left (RR)
Current tree height: 2
3420
```

```
Inserting 3420...
Current tree height: 3
680

Inserting 680...
Current tree height: 3

Enter number of deletions: 2
Enter values to delete:
680

Deleting 680...
Height after deletion: 3
34

Deleting 34...
Height after deletion: 2

Final Tree (Inorder): 20 2034 3420
```