Section b Q1

- **BST**: organized by the first letter of the names.
- **Successor**: the next node in **in-order traversal**.

✅ **Simple C++ Program**

```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    char nameInitial;
    Node* left;
    Node* right;
};

// Create new node
Node* newNode(char c) {
    Node* n = new Node;
    n->nameInitial = c;
    n->left = n->right = nullptr;
    return n;
}

// Insert node into BST
Node* insert(Node* root, char c) {
    if (!root) return newNode(c);
    if (c < root->nameInitial) root->left = insert(root->left, c);
    else root->right = insert(root->right, c);
    return root;
```

```
}

// Find minimum node in BST
Node* minNode(Node* root) {
   while (root->left) root = root->left;
   return root;
}

// Find successor of a node with given key
Node* successor(Node* root, char key) {
   Node* succ = nullptr;
   while (root) {
      if (key < root->nameInitial) {
         succ = root;
         root = root->left;
      } else root = root->right;
   }
   return succ;
}

int main() {
   char names[] = {'Q','S','R','T','M','A','B','P','N'};
   int n = 9;
   Node* root = nullptr;

   // Create BST
   for (int i = 0; i < n; i++)
      root = insert(root, names[i]);
```

```
    // Find successor of M
    Node* succ = successor(root, 'M');
    if (succ)
        cout << "Successor of M is: " << succ->nameInitial << endl;
    else
        cout << "M has no successor." << endl;

    return 0;
}
```

🔑 **How It Works**

1. BST is built by **comparing letters**.
2. Successor = smallest node **greater than M**.
3. For given letters, **successor of M = N**.

Q2 let's implement **all three standard BST traversals** in a simple C++ program.

✅ **C++ Program: BST Traversals**

```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Create a new node
Node* newNode(int val) {
```

```
    Node* n = new Node;
    n->data = val;
    n->left = n->right = nullptr;
    return n;
}

// Insert node into BST
Node* insert(Node* root, int val) {
    if (!root) return newNode(val);
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}

// In-Order Traversal (Left, Root, Right)
void inOrder(Node* root) {
    if (!root) return;
    inOrder(root->left);
    cout << root->data << " ";
    inOrder(root->right);
}

// Pre-Order Traversal (Root, Left, Right)
void preOrder(Node* root) {
    if (!root) return;
    cout << root->data << " ";
    preOrder(root->left);
    preOrder(root->right);
}
```

```
// Post-Order Traversal (Left, Right, Root)
void postOrder(Node* root) {
    if (!root) return;
    postOrder(root->left);
    postOrder(root->right);
    cout << root->data << " ";
}

int main() {
    Node* root = nullptr;

    // Sample data
    int values[] = {50, 30, 70, 20, 40, 60, 80};
    int n = 7;

    // Build BST
    for (int i = 0; i < n; i++)
        root = insert(root, values[i]);

    cout << "In-Order Traversal: ";
    inOrder(root);
    cout << endl;

    cout << "Pre-Order Traversal: ";
    preOrder(root);
    cout << endl;

    cout << "Post-Order Traversal: ";
```

```
    postOrder(root);
    cout << endl;

    return 0;
}
```

🔑 **Key Notes**

1. **In-Order** → prints BST in ascending order.
2. **Pre-Order** → useful for copying the tree or prefix expression.
3. **Post-Order** → useful for deleting/freeing the tree or postfix expression.

Q3**simplest C++ program** to search an element in a BST:

✅ **C++ Program: BST Search**

```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Create new node
Node* newNode(int val) {
    Node* n = new Node;
    n->data = val;
    n->left = n->right = nullptr;
```

```
    return n;
}

// Insert node into BST
Node* insert(Node* root, int val) {
    if (!root) return newNode(val);
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}

// Search in BST
bool search(Node* root, int key) {
    if (!root) return false;         // not found
    if (root->data == key) return true; // found
    if (key < root->data) return search(root->left, key);
    else return search(root->right, key);
}

int main() {
    Node* root = nullptr;

    // Sample BST
    int values[] = {50, 30, 70, 20, 40, 60, 80};
    int n = 7;
    for (int i = 0; i < n; i++)
        root = insert(root, values[i]);

    int key;
```

```
    cout << "Enter element to search: ";
    cin >> key;

    if (search(root, key))
        cout << key << " found in BST." << endl;
    else
        cout << key << " not found in BST." << endl;

    return 0;
}
```

🔑 **Notes**

- Works recursively.
- Complexity: **O(h)**, where h = height of the BST.
- Simplest possible version without extra functions or loops

Q4.

the **simplest C++ program** to build a BST from the given roll numbers and display them in ascending order using **in-order traversal**:

✅ **C++ Program**

```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
};
```

```
// Create a new node
Node* newNode(int val) {
    Node* n = new Node;
    n->data = val;
    n->left = n->right = nullptr;
    return n;
}

// Insert node into BST
Node* insert(Node* root, int val) {
    if (!root) return newNode(val);
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}

// In-Order Traversal (ascending order)
void inOrder(Node* root) {
    if (!root) return;
    inOrder(root->left);
    cout << root->data << " ";
    inOrder(root->right);
}

int main() {
    int rolls[] = {45, 12, 78, 34, 23, 89, 67, 11, 90, 54};
    int n = 10;
```

```
    Node* root = nullptr;

    // Construct BST
    for (int i = 0; i < n; i++)
        root = insert(root, rolls[i]);

    // Display roll numbers in ascending order
    cout << "Roll numbers in ascending order: ";
    inOrder(root);
    cout << endl;

    return 0;
}
```

Q5 . the **simplest C++ program** to delete a node from a BST:

✅ **C++ Program: Delete Node in BST**

```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* left;
    Node* right;
};

// Create a new node
Node* newNode(int val) {
    Node* n = new Node;
```

```
    n->data = val;
    n->left = n->right = nullptr;
    return n;
}

// Insert node into BST
Node* insert(Node* root, int val) {
    if (!root) return newNode(val);
    if (val < root->data) root->left = insert(root->left, val);
    else root->right = insert(root->right, val);
    return root;
}

// Find minimum value node in BST
Node* minNode(Node* root) {
    while (root && root->left) root = root->left;
    return root;
}

// Delete a node from BST
Node* deleteNode(Node* root, int key) {
    if (!root) return nullptr;

    if (key < root->data)
        root->left = deleteNode(root->left, key);
    else if (key > root->data)
        root->right = deleteNode(root->right, key);
    else {
        // Node with only one child or no child
```

```
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            Node* temp = root->left;
            delete root;
            return temp;
        }

        // Node with two children: get inorder successor
        Node* temp = minNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// In-Order Traversal
void inOrder(Node* root) {
    if (!root) return;
    inOrder(root->left);
    cout << root->data << " ";
    inOrder(root->right);
}

int main() {
    int rolls[] = {50, 30, 70, 20, 40, 60, 80};
    int n = 7;
```

```
    Node* root = nullptr;

    // Construct BST
    for (int i = 0; i < n; i++)
        root = insert(root, rolls[i]);

    int key;
    cout << "Enter roll number to delete: ";
    cin >> key;

    root = deleteNode(root, key);

    cout << "BST after deletion (In-Order): ";
    inOrder(root);
    cout << endl;

    return 0;
}
```

🔑 **How It Works**

1. **No child:** Simply delete the node.
2. **One child:** Replace node with its child.
3. **Two children:** Replace node with **in-order successor** (smallest node in right subtree).

Q6let's implement a **simple BST-based family tree** in C++ with all the requested features.

We'll keep it **simple**:

- BST is organized by **names alphabetically**.
- Each node stores a **name**.
- Supports **insert**, **search**, and **traversals**.

✅ **C++ Program**

```
#include <iostream>
#include <string>
using namespace std;

// Node structure
struct Node {
    string name;
    Node* left;
    Node* right;
};

// Create new node
Node* newNode(string name) {
    Node* n = new Node;
    n->name = name;
    n->left = n->right = nullptr;
    return n;
}

// Insert node into BST
Node* insert(Node* root, string name) {
    if (!root) return newNode(name);
    if (name < root->name) root->left = insert(root->left, name);
    else root->right = insert(root->right, name);
    return root;
}

// Search for a name
```

```
bool search(Node* root, string name) {
    if (!root) return false;
    if (root->name == name) return true;
    if (name < root->name) return search(root->left, name);
    else return search(root->right, name);
}

// In-Order Traversal
void inOrder(Node* root) {
    if (!root) return;
    inOrder(root->left);
    cout << root->name << " ";
    inOrder(root->right);
}

// Pre-Order Traversal
void preOrder(Node* root) {
    if (!root) return;
    cout << root->name << " ";
    preOrder(root->left);
    preOrder(root->right);
}

// Post-Order Traversal
void postOrder(Node* root) {
    if (!root) return;
    postOrder(root->left);
    postOrder(root->right);
    cout << root->name << " ";
```

```
}

int main() {
    Node* root = nullptr;

    // Insert family members
    string members[] = {"John", "Alice", "Bob", "Mary", "David", "Sophia"};
    int n = 6;
    for (int i = 0; i < n; i++)
        root = insert(root, members[i]);

    // Display hierarchy
    cout << "In-Order Traversal: ";
    inOrder(root);
    cout << endl;

    cout << "Pre-Order Traversal: ";
    preOrder(root);
    cout << endl;

    cout << "Post-Order Traversal: ";
    postOrder(root);
    cout << endl;

    // Search for a member
    string name;
    cout << "Enter a family member name to search: ";
    cin >> name;
    if (search(root, name))
```

```
        cout << name << " found in the family tree." << endl;
    else
        cout << name << " not found in the family tree." << endl;

    return 0;
}
```

🔑 **Key Points**

1. **Insert:** Alphabetical order (BST property).
2. **Search:** Recursive lookup based on name.
3. **Traversals:**
    - **In-Order:** names in ascending order.
    - **Pre-Order:** shows hierarchy starting from root.
    - **Post-Order:** useful for bottom-up processing.