```cpp
//AVL Tree
#include <iostream>

using namespace std;

struct AVLNode {
    int key;
    AVLNode* left, * right;
    int height;
};

int getHeight(AVLNode* root) {
    if (!root) return 0;
    return root->height;
}

void fixHeight(AVLNode*& root) {
    root->height = 1 + std::max(getHeight(root->left), getHeight(root-
      >right));
}

void leftRotate(AVLNode*& root) {
    AVLNode* B = root->right;
    AVLNode* Y = B->left;

    B->left = root;
    root->right = Y;

    fixHeight(root);
    fixHeight(B);

    root = B;
}

void rightRotate(AVLNode*& root) {
    AVLNode* B = root->left;
    AVLNode* Y = B->right;

    B->right = root;
    root->left = Y;

    fixHeight(root);
    fixHeight(B);

    root = B;
}

int getBalanceFactor(AVLNode* node) {
    return getHeight(node->left) - getHeight(node->right);
```

```cpp
}

void insertAVLNode(AVLNode*& root, int key) {
    if (!root) {
        root = new AVLNode{ key, nullptr, nullptr, 1 };
        return;
    }

    if (key < root->key) {
        insertAVLNode(root->left, key);
    }
    else if (key > root->key) {
        insertAVLNode(root->right, key);
    }
    else return;

    fixHeight(root);

    int bf = getBalanceFactor(root);

    if (bf > 1 && key < root->left->key) {
        rightRotate(root);
        return;
    }

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

    if (bf > 1 && key > root->left->key) {
        leftRotate(root->left);
        rightRotate(root);
        return;
    }

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

AVLNode* findPredecessor(AVLNode* node) {
    if (!node->left) {
        std::cout << "This node does not have predecessor!";
        return nullptr;
    }

    AVLNode* y = node->left;
    while (y->right) y = y->right;
    return y;
}

void deleteAVLNode(AVLNode*& root, int key) {
    if (!root) return;
    else if (key < root->key) deleteAVLNode(root->left, key);
    else if (key > root->key) deleteAVLNode(root->right, key);
    else {
        if (!root->left) {
            AVLNode* temp = root->right;
            delete root;
            root = temp;
            return;
        }
        else if (!root->right) {
            AVLNode* temp = root->left;
            delete root;
            root = temp;
            return;
        }
        else {
            AVLNode* pred = findPredecessor(root);
            root->key = pred->key;

            deleteAVLNode(root->left, pred->key);
        }
    }

    fixHeight(root);

    int bf = getBalanceFactor(root);

    if (bf > 1 && getBalanceFactor(root->left) >= 0) {
        rightRotate(root);
    }

    else if (bf < -1 && getBalanceFactor(root->right) <= 0) {
        leftRotate(root);
    }

    else if (bf > 1 && getBalanceFactor(root->left) < 0) {
        leftRotate(root->left);
        rightRotate(root);
    }

    else if (bf < -1 && getBalanceFactor(root->right) > 0) {
        rightRotate(root->right);
        leftRotate(root);
    }
}
```

```cpp
//BINARY TREE
#include <iostream>
#include <stack>
#include <queue>

using namespace std;

struct Node {
    int key;
    int items;
    Node* left, * right;

    Node(int k = 0) : key(k), items(0), left(nullptr), right(nullptr) {}
};

//Tim trung vi hieu qua
int getItems(Node* node) {
    if (!node) return 0;
    return node->items;
}

int findKth(Node* root, int k) {
    if (!root) return -1;
    int l = getItems(root->left);
    if (l == k) return root->key;
    if (l > k) return findKth(root->left, k);
    else return findKth(root->right, k - l - 1);
}
//


void preOrderNoRecursion(Node* root) {
    stack<Node*> st;
    if (root) st.push(root);

    while (!st.empty()) {
        Node* node = st.top();
        st.pop();

        cout << node->key << " ";
        if (node->right) {
            st.push(node->right);
        }
        if (node->left) {
            st.push(node->left);
        }
    }
}

```

```cpp
void inOrderNoRecursion(Node* root) {
    stack<Node*> st;

    Node* q = root;
    while (q || !st.empty()) {
        while (q) {
            st.push(q);
            q = q->left;
        }
        q = st.top();
        st.pop();
        cout << q->key << " ";
        q = q->right;
    }
}

void postOrderNoRecursion(Node* root) {
    stack<Node*> st1, st2;
    if (root) st1.push(root);

    while (!st1.empty()) {
        Node* node = st1.top();
        st1.pop();

        st2.push(node);
        if (node->left) st1.push(node->left);
        if (node->right) st1.push(node->right);
    }

    while (!st2.empty()) {
        cout << st2.top()->key << " ";
        st2.pop();
    }
}

void removeNodeNoRecursion(Node*& root, int key) {
    Node* z = root, * prev = nullptr;
    while (z && z->key != key) {
        prev = z;
        if (key < z->key) z = z->left;
        else z = z->right;
    }

    if (!z) return;

    Node* y; // y: node that su bi xoa, prev: node cha cua y
    if (!z->left || !z->right) y = z;
    else {
        y = z->left, prev = z;
        while (y->right) prev = y, y = y->right;
    }

    z->key = y->key;

    if (!prev) root = nullptr;
    else if (prev->left == y) prev->left = (y->left ? y->left : y->right);
    else prev->right = (y->left ? y->left : y->right);

    delete y;
}
```

```cpp
//HASH TABLE

#include <iostream>

using namespace std;

struct Node {
    int key;
    Node* next;
};

struct HashTable {
    Node** head;
    int size;
};

HashTable createHashTable(int size = 0) {
    HashTable newTable;
    newTable.size = size;
    newTable.head = new Node * [size];

    for (int i = 0; i < size; i++) newTable.head[i] = nullptr;

    return newTable;
}

int calculateHash(int id) {
    return id % 10;
}

void insertTableNode(Node*& head, int key) {
    head = new Node{ key, head };
}

void insertItem(HashTable& table, int key) {
    int index = calculateHash(key);
    insertTableNode(table.head[index], key);
}

void deleteItem(HashTable& table, int key) {
    int index = calculateHash(key);
    Node* prev = nullptr, * cur = table.head[index];
    while (cur && cur->key != key) {
        prev = cur, cur = cur->next;
    }

    if (!cur) return;

    if (!prev) {
        table.head[index] = table.head[index]->next;
    }
    else {
        prev->next = cur->next;
    }
    delete cur;
}

void displayHashTable(HashTable table) {
    for (int i = 0; i < table.size; i++) {
        cout << i << " -> ";
        for (Node* p = table.head[i]; p; p = p->next) {
            cout << p->key << " ";
        }
        cout << endl;
    }
}
```

```cpp
//Priority Queue
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

// Structure representing an object with an ID, order, and priority
struct Object {
    string id;
    int order;
    int priority;
};

// Structure representing a priority queue implemented using a min-heap
struct PriorityQueueHeap {
    vector<Object> arr;
};

// Function to check if the priority queue is empty
bool isEmpty(const PriorityQueueHeap& pq) {
    return pq.arr.empty();
}

// Function to maintain the heap property while moving an element up the heap
void heapifyUp(vector<Object>& arr, int index) {
    while (index > 0) {
        int parentIndex = (index - 1) / 2;
        if (arr[index].priority < arr[parentIndex].priority) {
            swap(arr[index], arr[parentIndex]);
            index = parentIndex;
        }
        else {
            break;
        }
    }
}

// Function to maintain the heap property while moving an element down the heap
void heapifyDown(vector<Object>& arr, int index) {
    int size = arr.size();
    while (true) {
        int leftChild = 2 * index + 1;
        int rightChild = 2 * index + 2;
        int smallest = index;

        if (leftChild < size && arr[leftChild].priority < arr[smallest].priority) {
            smallest = leftChild;
        }

        if (rightChild < size && arr[rightChild].priority < arr[smallest].priority) {
            smallest = rightChild;
        }

        if (smallest != index) {
            swap(arr[index], arr[smallest]);
            index = smallest;
        }
        else {
            break;
        }
    }
}

// Function to insert an object into the priority queue
void insert(PriorityQueueHeap& pq, const Object& obj) {
    pq.arr.push_back(obj);
    heapifyUp(pq.arr, pq.arr.size() - 1);
}

// Function to extract the object with the highest priority from the priority queue
Object extract(PriorityQueueHeap& pq) {
    if (isEmpty(pq)) {
        cerr << "Error: Priority queue is empty." << endl;
        // Handle error accordingly, here we just return an Object with an empty string.
        return Object{ "", 0, 0 };
    }

    Object result = pq.arr[0];
    pq.arr[0] = pq.arr.back();
    pq.arr.pop_back();
    heapifyDown(pq.arr, 0);

    return result;
}

// Function to remove an object with a given ID from the priority queue
void remove(PriorityQueueHeap& pq, const string& objectId) {
    int index = -1;
    for (int i = 0; i < pq.arr.size(); ++i) {
        if (pq.arr[i].id == objectId) {
            index = i;
            break;
        }
    }

    if (index == -1) {
        cerr << "Error: Object with id " << objectId << " not found." << endl;
        return;
    }

    pq.arr[index].priority = INT_MIN; // Set priority to negative infinity
    heapifyUp(pq.arr, index);
    (void)extract(pq);
}

// Function to change the priority of an object with a given ID
void changePriority(PriorityQueueHeap& pq, const string& objectId, int newPriority) {
    int index = -1;
    for (int i = 0; i < pq.arr.size(); ++i) {
        if (pq.arr[i].id == objectId) {
            index = i;
            break;
        }
    }

    if (index == -1) {
        cerr << "Error: Object with id " << objectId << " not found." << endl;
        return;
    }

    int oldPriority = pq.arr[index].priority;
    pq.arr[index].priority = newPriority;

    if (newPriority < oldPriority) {
        heapifyUp(pq.arr, index);
    }
    else {
        heapifyDown(pq.arr, index);
    }
}
```

```cpp
//Red black Tree
#include <iostream>

using namespace std;

// Enum representing the color of a node in the red-black tree
enum Color {
    RED, BLACK,
};

// Node structure for the red-black tree
typedef struct RBNode* Ref;
struct RBNode {
    int key;
    Color color;
    Ref parent, left, right;
};

// Global variable representing the nil (sentinel) node
Ref nil;

// Function to create a new node with given key, color, and nil reference
Ref createNode(int key, Color color, Ref nil) {
    Ref p = new RBNode{ key, color, nil, nil, nil };
    return p;
}

// Left rotation operation in the red-black tree
void leftRotate(Ref& root, Ref x) {
    Ref y = x->right;
    x->right = y->left;

    if (y->left != nil) {
        y->left->parent = x;
    }
    y->parent = x->parent;

    if (x->parent == nil) {
        root = y;
    }
    else {
        if (x == x->parent->left) {
            x->parent->left = y;
        }
        else {
            x->parent->right = y;
        }
    }

    y->left = x;
    x->parent = y;
}

// Right rotation operation in the red-black tree
void rightRotate(Ref& root, Ref x) {
    Ref y = x->left;
    x->left = y->right;

    if (y->right != nil) {
        y->right->parent = x;
    }
    y->parent = x->parent;

    if (x->parent == nil) {
        root = y;
    }
    else {
        if (x == x->parent->right) {
            x->parent->right = y;
        }
        else {
            x->parent->left = y;
        }
    }

    y->right = x;
    x->parent = y;
}

// Binary Search Tree (BST) insertion operation
void BST_Insert(Ref& root, Ref x) {
    Ref y = nil, z = root;
    while (z != nil) {
        y = z;

        if (x->key < z->key) z = z->left;
        else if (x->key > z->key) z = z->right;
        else return; // Key already exists, do nothing
    }

    x->parent = y;
    if (y == nil) root = x;
    else {
        if (x->key < y->key) y->left = x;
        else y->right = x;
    }
}

```

```cpp
// Adjustments after left child insertion in the red-black tree
void insertionLeftAdjust(Ref& root, Ref& x) {
    Ref u = x->parent->parent->right;
    if (u->color == RED) {
        x->parent->color = u->color = BLACK;
        x->parent->parent->color = RED;
        x = x->parent->parent;
    }
    else {
        if (x == x->parent->right) {
            x = x->parent;
            leftRotate(root, x);
        }
        x->parent->color = BLACK;
        x->parent->color = BLACK;
        x->parent->parent->color = RED;
        rightRotate(root, x->parent->parent);
    }
}

// Adjustments after right child insertion in the red-black tree
void insertionRightAdjust(Ref& root, Ref& x) {
    Ref u = x->parent->parent->left;
    if (u->color == RED) {
        x->parent->color = u->color = BLACK;
        x->parent->parent->color = RED;
        x = x->parent->parent;
    }
    else {
        if (x == x->parent->left) {
            x = x->parent;
            rightRotate(root, x);
        }
        x->parent->color = BLACK;
        x->parent->color = BLACK;
        x->parent->parent->color = RED;
        leftRotate(root, x->parent->parent);
    }
}

// Fix-up routine after insertion in the red-black tree
void insertionFixUp(Ref& root, Ref x) {
    while (x->parent->color == RED) {
        if (x->parent == x->parent->parent->left) {
            insertionLeftAdjust(root, x);
        }
        else {
            insertionRightAdjust(root, x);
        }
    }
    root->color = BLACK;
}

// Insert a key into the red-black tree
void Insert(Ref& root, int key) {
    Ref x = createNode(key, RED, nil);
    BST_Insert(root, x);
    insertionFixUp(root, x);
}

// Create a red-black tree from an array of keys
Ref createTree(int a[], int n) {
    Ref root = nil;

    for (int i = 0; i < n; i++) {
        Insert(root, a[i]);
    }

    return root;
}

// Adjustments after left child deletion in the red-black tree
void deleteLeftAdjust(Ref& root, Ref& x) {
    Ref w = x->parent->right;

    if (w->color == RED) {
        w->color = BLACK;
        x->parent->color = RED;
        leftRotate(root, x->parent);
        w = x->parent->right;
    }

    if ((w->right->color == BLACK) && (w->left->color == BLACK)) {
        w->color = RED;
        x = x->parent;
    }
    else {
        if (w->right->color == BLACK) {
            w->left->color = BLACK;
            w->color = RED;
            rightRotate(root, w);
            w = x->parent->right;
        }
        w->color = x->parent->color;
        x->parent->color = w->right->color = BLACK;
        leftRotate(root, x->parent);
        x = root;
    }
```

```cpp
}

// Adjustments after right child deletion in the red-black tree
void deleteRightAdjust(Ref& root, Ref& x) {
    Ref w = x->parent->left;

    if (w->color == RED) {
        w->color = BLACK;
        x->parent->color = RED;
        leftRotate(root, x->parent);
        w = x->parent->left;
    }

    if ((w->left->color == BLACK) && (w->right->color == BLACK)) {
        w->color = RED;
        x = x->parent;
    }
    else {
        if (w->left->color == BLACK) {
            w->right->color = BLACK;
            w->color = RED;
            leftRotate(root, w);
            w = x->parent->left;
        }
        w->color = x->parent->color;
        x->parent->color = w->left->color = BLACK;
        rightRotate(root, x->parent);
        x = root;
    }
}

// Fix-up routine after deletion in the red-black tree
void deleteFixUp(Ref root, Ref x) {
    while ((x->color == BLACK) && (x != root)) {
        if (x == x->parent->left) deleteLeftAdjust(root, x);
        else deleteRightAdjust(root, x);
    }
    x->color = BLACK;
}

// Search for a key in the red-black tree and return the corresponding
  node
Ref lookup(Ref root, int key) {
    Ref p = root;
    while (p != nil) {
        if (key == p->key) return p;

        if (key < p->key) p = p->left;
        else p = p->right;
    }

    return nil;
}

// Find the predecessor of a given node in the red-black tree
Ref findPredecessor(Ref z) {
    if (z->left == nil) {
        std::cout << "This node does not have predecessor!";
        return nullptr;
    }

    Ref y = z->left;
    while (y->right != nil) y = y->right;
    return y;
}

// Remove a key from the red-black tree
void Remove(Ref& root, int k) {
    Ref z = lookup(root, k);
    if (z == nil) return;

    Ref y = (z->left == nil) || (z->right == nil) ? z : findPredecessor
      (z);

    Ref x = (y->left == nil) ? y->right : y->left;

    x->parent = y->parent;
    if (y->parent == nil) root = x;
    else {
        if (y == y->parent->left) y->parent->left = x;
        else y->parent->right = x;
    }

    if (y != z) z->key = y->key;
    if (y->color == BLACK) {
        deleteFixUp(root, x);
    }

    delete y;
}

struct Node {
    int key;
    Node* left, * right;
    int height;
};

bool isGreater(int a, int b) {
    return a > b;
}

bool isSmaller(int a, int b) {
    return a < b;
}

bool isEqual(int a, int b) {
    return a == b;
}

int getHeight(Node* root) {
    if (!root) return 0;
    return root->height;
}

void fixHeight(Node*& root) {
    root->height = 1 + std::max(getHeight(root->left), getHeight(root->right));
}

void leftRotate(Node*& root) {
    Node* B = root->right;
    Node* Y = B->left;

    B->left = root;
    root->right = Y;

    fixHeight(root);
    fixHeight(B);

    root = B;
}

void rightRotate(Node*& root) {
    Node* B = root->left;
    Node* Y = B->right;

    B->right = root;
    root->left = Y;

    fixHeight(root);
    fixHeight(B);

    root = B;
}

int getBalanceFactor(Node* node) {
    return getHeight(node->left) - getHeight(node->right);
```

```cpp
}

void insertNode(Node*& root, int key) {
    if (!root) {
        root = new Node{ key, nullptr, nullptr, 1 };
        return;
    }

    if (isSmaller(key, root->key)) {
        insertNode(root->left, key);
    }
    else if (isGreater(key, root->key)) {
        insertNode(root->right, key);
    }
    else return;

    fixHeight(root);

    int bf = getBalanceFactor(root);

    if (bf > 1 && isSmaller(key, root->left->key)) {
        rightRotate(root);
        return;
    }

    if (bf < -1 && isGreater(key, root->right->key)) {
        leftRotate(root);
        return;
    }

    if (bf > 1 && isGreater(key, root->left->key)) {
        leftRotate(root->left);
        rightRotate(root);
        return;
    }

    if (bf < -1 && isSmaller(key, root->right->key)) {
        rightRotate(root->right);
        leftRotate(root);
        return;
    }
}

Node* findPredecessor(Node* node) {
    if (!node->left) {
        std::cout << "This node does not have predecessor!";
        return nullptr;
    }

    Node* y = node->left;
    while (y->right) y = y->right;
    return y;
}

void deleteNode(Node*& root, int key) {
    if (!root) return;
    else if (isSmaller(key, root->key)) deleteNode(root->left, key);
    else if (isGreater(key, root->key)) deleteNode(root->right, key);
    else {
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            root = temp;
            return;
        }
        else if (!root->right) {
            Node* temp = root->left;
            delete root;
            root = temp;
            return;
        }
        else {
            Node* pred = findPredecessor(root);
            root->key = pred->key;

            deleteNode(root->left, pred->key);
        }
    }

    fixHeight(root);

    int bf = getBalanceFactor(root);

    if (bf > 1 && getBalanceFactor(root->left) >= 0) {
        rightRotate(root);
    }

    else if (bf < -1 && getBalanceFactor(root->right) <= 0) {
        leftRotate(root);
    }

    else if (bf > 1 && getBalanceFactor(root->left) < 0) {
        leftRotate(root->left);
        rightRotate(root);
    }

    else if (bf < -1 && getBalanceFactor(root->right) > 0) {
        rightRotate(root->right);
        leftRotate(root);
    }
}
```

```cpp
//TOPO SORT

//(1,2)(1,3)(2,3)(6,9)(5,4)(3,7)(0,7)(9,8)(3,0)(5,0)(2,6)(1,8) => 5 4 1 2 3 0 7 6 9 8
//(9,1)(5,6)(5,4)(4,8)(0,1)(7,2)(7,3)(9,4)(5,7)(0,2)(1,3)(0,6) => 0 5 6 7 2 9 1 3 4 8

#include <iostream>
#include <fstream>
#include <vector>

// Forward declaration of Leader and Trailer structures
typedef struct Leader* lref;
typedef struct Trailer* tref;

// Structure representing a Leader node
struct Leader {
    int key;        // Key of the leader
    int count;      // Number of incoming precedences
    lref next;      // Pointer to the next leader node in the list
    tref trails;    // Pointer to the list of trailers
};

// Structure representing a Trailer node
struct Trailer {
    lref id;        // Pointer to the leader node
    tref next;      // Pointer to the next trailer node in the list
};

// Function to find the leader with key x; if not exist yet, add to the end of the leader list
lref findLeader(lref& head, lref& tail, int x) {
    lref p = head;

    tail->key = x;

    while (p->key != x) {
        p = p->next;
    }

    if (p == tail) {
        tail = new Leader;

        p->count = 0;
        p->trails = nullptr;
        p->next = tail;
    }

    return p;
```

```cpp
}

// Function to split leaders with no precedences from the leader list
void splitLeaderWithNoPrecedence(lref& head, lref& tail) {
    lref p = head;
    head = nullptr;

    while (p != tail) {
        lref tmp = p->next;

        if (p->count == 0) {
            p->next = head;
            head = p;
        }

        p = tmp;
    }
}

// Function to add a new order x < y
void addOrder(lref& head, lref& tail, int x, int y) {
    lref xNode = findLeader(head, tail, x);
    lref yNode = findLeader(head, tail, y);

    tref xTrail = new Trailer{ yNode, xNode->trails };
    xNode->trails = xTrail;

    // Increase the number of precedences
    yNode->count++;
}

// Function to create leaders from pairs of orders
void createLeadersFromPairs(lref& head, lref& tail,
  std::vector<std::pair<int, int>> orders) {
    head = new Leader{ -1, 0, nullptr, nullptr };
    tail = head;

    for (int i = 0; i < orders.size(); i++) {
        addOrder(head, tail, orders[i].first, orders[i].second);
    }
}

// Function to perform topological sort based on the given orders
void topoSort(std::vector<std::pair<int, int>> orders) {
    lref head, tail;
    createLeadersFromPairs(head, tail, orders);

    splitLeaderWithNoPrecedence(head, tail);

    lref p = head;

    while (p) {
        std::cout << p->key << " ";

        tref t = p->trails;

        p = p->next;

        for (tref q = t; q; q = q->next) {
            lref succNode = q->id;

            succNode->count--;

            if (succNode->count == 0) {
                succNode->next = p;

                p = succNode;
            }
        }
    }
}

// Function to parse orders from a file and return a vector of pairs
std::vector<std::pair<int, int>> parseFile(std::string fileName) {
    std::vector<std::pair<int, int>> orders;

    std::pair<int, int> p;
    char ch1, ch2, ch3;

    std::ifstream inFile;
    inFile.open(fileName);

    while (!inFile.eof()) {
        inFile >> ch1 >> p.first >> ch2 >> p.second >> ch3;

        if (inFile.eof()) {
            break;
        }

        orders.push_back(p);
    }

    inFile.close();

    return orders;
}

// Main function
```

```cpp
int main() {
    // Parse orders from the input file
    std::vector<std::pair<int, int>> orders = parseFile("input.txt");

    // Perform topological sort and print the result
    topoSort(orders);

    return 0;
}

```