

```
1 //AVL Tree
2 #include <iostream>
3
4 using namespace std;
5
6 struct AVLNode {
7     int key;
8     AVLNode* left, * right;
9     int height;
10 };
11
12 int getHeight(AVLNode* root) {
13     if (!root) return 0;
14     return root->height;
15 }
16
17 void fixHeight(AVLNode*& root) {
18     root->height = 1 + std::max(getHeight(root->left), getHeight(root->right));
19 }
20
21 void leftRotate(AVLNode*& root) {
22     AVLNode* B = root->right;
23     AVLNode* Y = B->left;
24
25     B->left = root;
26     root->right = Y;
27
28     fixHeight(root);
29     fixHeight(B);
30
31     root = B;
32 }
33
34 void rightRotate(AVLNode*& root) {
35     AVLNode* B = root->left;
36     AVLNode* Y = B->right;
37
38     B->right = root;
39     root->left = Y;
40
41     fixHeight(root);
42     fixHeight(B);
43
44     root = B;
45 }
46
47 int getBalanceFactor(AVLNode* node) {
48     return getHeight(node->left) - getHeight(node->right);
```

```
49 }
50
51 void insertAVLNode(AVLNode*& root, int key) {
52     if (!root) {
53         root = new AVLNode{ key, nullptr, nullptr, 1 };
54         return;
55     }
56
57     if (key < root->key) {
58         insertAVLNode(root->left, key);
59     }
60     else if (key > root->key) {
61         insertAVLNode(root->right, key);
62     }
63     else return;
64
65     fixHeight(root);
66
67     int bf = getBalanceFactor(root);
68
69     if (bf > 1 && key < root->left->key) {
70         rightRotate(root);
71         return;
72     }
73
74     if (bf < -1 && key > root->right->key) {
75         leftRotate(root);
76         return;
77     }
78
79     if (bf > 1 && key > root->left->key) {
80         leftRotate(root->left);
81         rightRotate(root);
82         return;
83     }
84
85     if (bf < -1 && key < root->right->key) {
86         rightRotate(root->right);
87         leftRotate(root);
88         return;
89     }
90 }
91
92 AVLNode* findPredecessor(AVLNode* node) {
93     if (!node->left) {
94         std::cout << "This node does not have predecessor!";
95         return nullptr;
96     }
97 }
```

```
198     AVLNode* y = node->left;
199     while (y->right) y = y->right;
200     return y;
201 }
202
203 void deleteAVLNode(AVLNode*& root, int key) {
204     if (!root) return;
205     else if (key < root->key) deleteAVLNode(root->left, key);
206     else if (key > root->key) deleteAVLNode(root->right, key);
207     else {
208         if (!root->left) {
209             AVLNode* temp = root->right;
210             delete root;
211             root = temp;
212             return;
213         }
214         else if (!root->right) {
215             AVLNode* temp = root->left;
216             delete root;
217             root = temp;
218             return;
219         }
220         else {
221             AVLNode* pred = findPredecessor(root);
222             root->key = pred->key;
223
224             deleteAVLNode(root->left, pred->key);
225         }
226     }
227
228     fixHeight(root);
229
230     int bf = getBalanceFactor(root);
231
232     if (bf > 1 && getBalanceFactor(root->left) >= 0) {
233         rightRotate(root);
234     }
235
236     else if (bf < -1 && getBalanceFactor(root->right) <= 0) {
237         leftRotate(root);
238     }
239
240     else if (bf > 1 && getBalanceFactor(root->left) < 0) {
241         leftRotate(root->left);
242         rightRotate(root);
243     }
244
245     else if (bf < -1 && getBalanceFactor(root->right) > 0) {
246         rightRotate(root->right);
```

```
147         leftRotate(root);  
148     }  
149 }
```

```
1  //BINARY TREE
2  #include <iostream>
3  #include <stack>
4  #include <queue>
5
6  using namespace std;
7
8  struct Node {
9      int key;
10     int items;
11     Node* left, * right;
12
13     Node(int k = 0) : key(k), items(0), left(nullptr), right(nullptr) {}
14 };
15
16 //Tim trung vi hieu qua
17 int getItems(Node* node) {
18     if (!node) return 0;
19     return node->items;
20 }
21
22 int findKth(Node* root, int k) {
23     if (!root) return -1;
24     int l = getItems(root->left);
25     if (l == k) return root->key;
26     if (l > k) return findKth(root->left, k);
27     else return findKth(root->right, k - l - 1);
28 }
29 //
30
31
32 void preOrderNoRecursion(Node* root) {
33     stack<Node*> st;
34     if (root) st.push(root);
35
36     while (!st.empty()) {
37         Node* node = st.top();
38         st.pop();
39
40         cout << node->key << " ";
41         if (node->right) {
42             st.push(node->right);
43         }
44         if (node->left) {
45             st.push(node->left);
46         }
47     }
48 }
49
```

```
50 void inOrderNoRecursion(Node* root) {
51     stack<Node*> st;
52
53     Node* q = root;
54     while (q || !st.empty()) {
55         while (q) {
56             st.push(q);
57             q = q->left;
58         }
59         q = st.top();
60         st.pop();
61         cout << q->key << " ";
62         q = q->right;
63     }
64 }
65
66 void postOrderNoRecursion(Node* root) {
67     stack<Node*> st1, st2;
68     if (root) st1.push(root);
69
70     while (!st1.empty()) {
71         Node* node = st1.top();
72         st1.pop();
73
74         st2.push(node);
75         if (node->left) st1.push(node->left);
76         if (node->right) st1.push(node->right);
77     }
78
79     while (!st2.empty()) {
80         cout << st2.top()->key << " ";
81         st2.pop();
82     }
83 }
84
85 void removeNodeNoRecursion(Node*& root, int key) {
86     Node* z = root, * prev = nullptr;
87     while (z && z->key != key) {
88         prev = z;
89         if (key < z->key) z = z->left;
90         else z = z->right;
91     }
92
93     if (!z) return;
94
95     Node* y; // y: node that su bi xoa, prev: node cha cua y
96     if (!z->left || !z->right) y = z;
97     else {
98         y = z->left, prev = z;
```

```
99         while (y->right) prev = y, y = y->right;
100     }
101
102     z->key = y->key;
103
104     if (!prev) root = nullptr;
105     else if (prev->left == y) prev->left = (y->left ? y->left : y->right);
106     else prev->right = (y->left ? y->left : y->right);
107
108     delete y;
109 }
```

```
1  //HASH TABLE
2
3  #include <iostream>
4
5  using namespace std;
6
7  struct Node {
8      int key;
9      Node* next;
10 };
11
12 struct HashTable {
13     Node** head;
14     int size;
15 };
16
17 HashTable createHashTable(int size = 0) {
18     HashTable newTable;
19     newTable.size = size;
20     newTable.head = new Node * [size];
21
22     for (int i = 0; i < size; i++) newTable.head[i] = nullptr;
23
24     return newTable;
25 }
26
27 int calculateHash(int id) {
28     return id % 10;
29 }
30
31 void insertTableNode(Node*& head, int key) {
32     head = new Node{ key, head };
33 }
34
35 void insertItem(HashTable& table, int key) {
36     int index = calculateHash(key);
37     insertTableNode(table.head[index], key);
38 }
39
40 void deleteItem(HashTable& table, int key) {
41     int index = calculateHash(key);
42     Node* prev = nullptr, * cur = table.head[index];
43     while (cur && cur->key != key) {
44         prev = cur, cur = cur->next;
45     }
46
47     if (!cur) return;
48
49     if (!prev) {
```



```
50     table.head[index] = table.head[index]->next;
51 }
52 else {
53     prev->next = cur->next;
54 }
55 delete cur;
56 }
57
58 void displayHashTable(HashTable table) {
59     for (int i = 0; i < table.size; i++) {
60         cout << i << " -> ";
61         for (Node* p = table.head[i]; p; p = p->next) {
62             cout << p->key << " ";
63         }
64         cout << endl;
65     }
66 }
```

```
1 //Priority Queue
2 #include <iostream>
3 #include <string>
4 #include <vector>
5 #include <algorithm>
6
7 using namespace std;
8
9 // Structure representing an object with an ID, order, and priority
10 struct Object {
11     string id;
12     int order;
13     int priority;
14 };
15
16 // Structure representing a priority queue implemented using a min-heap
17 struct PriorityQueueHeap {
18     vector<Object> arr;
19 };
20
21 // Function to check if the priority queue is empty
22 bool isEmpty(const PriorityQueueHeap& pq) {
23     return pq.arr.empty();
24 }
25
26 // Function to maintain the heap property while moving an element up the heap ➤
27 void heapifyUp(vector<Object>& arr, int index) {
28     while (index > 0) {
29         int parentIndex = (index - 1) / 2;
30         if (arr[index].priority < arr[parentIndex].priority) {
31             swap(arr[index], arr[parentIndex]);
32             index = parentIndex;
33         }
34         else {
35             break;
36         }
37     }
38 }
39
40 // Function to maintain the heap property while moving an element down the heap ➤
41 void heapifyDown(vector<Object>& arr, int index) {
42     int size = arr.size();
43     while (true) {
44         int leftChild = 2 * index + 1;
45         int rightChild = 2 * index + 2;
46         int smallest = index;
47
```

```
48     if (leftChild < size && arr[leftChild].priority < arr
        [smallest].priority) {
49         smallest = leftChild;
50     }
51
52     if (rightChild < size && arr[rightChild].priority < arr
        [smallest].priority) {
53         smallest = rightChild;
54     }
55
56     if (smallest != index) {
57         swap(arr[index], arr[smallest]);
58         index = smallest;
59     }
60     else {
61         break;
62     }
63 }
64 }
65
66 // Function to insert an object into the priority queue
67 void insert(PriorityQueueHeap& pq, const Object& obj) {
68     pq.arr.push_back(obj);
69     heapifyUp(pq.arr, pq.arr.size() - 1);
70 }
71
72 // Function to extract the object with the highest priority from the
    priority queue
73 Object extract(PriorityQueueHeap& pq) {
74     if (isEmpty(pq)) {
75         cerr << "Error: Priority queue is empty." << endl;
76         // Handle error accordingly, here we just return an Object with an
        empty string.
77         return Object{ "", 0, 0 };
78     }
79
80     Object result = pq.arr[0];
81     pq.arr[0] = pq.arr.back();
82     pq.arr.pop_back();
83     heapifyDown(pq.arr, 0);
84
85     return result;
86 }
87
88 // Function to remove an object with a given ID from the priority queue
89 void remove(PriorityQueueHeap& pq, const string& objectId) {
90     int index = -1;
91     for (int i = 0; i < pq.arr.size(); ++i) {
92         if (pq.arr[i].id == objectId) {
```

```
93         index = i;
94         break;
95     }
96 }
97
98 if (index == -1) {
99     cerr << "Error: Object with id " << objectId << " not found." <<  ↵
100     endl;
101     return;
102 }
103
104 pq.arr[index].priority = INT_MIN; // Set priority to negative infinity
105 heapifyUp(pq.arr, index);
106 (void)extract(pq);
107 }
108 // Function to change the priority of an object with a given ID
109 void changePriority(PriorityQueueHeap& pq, const string& objectId, int  ↵
    newPriority) {
110     int index = -1;
111     for (int i = 0; i < pq.arr.size(); ++i) {
112         if (pq.arr[i].id == objectId) {
113             index = i;
114             break;
115         }
116     }
117
118     if (index == -1) {
119         cerr << "Error: Object with id " << objectId << " not found." <<  ↵
120         endl;
121         return;
122     }
123
124     int oldPriority = pq.arr[index].priority;
125     pq.arr[index].priority = newPriority;
126
127     if (newPriority < oldPriority) {
128         heapifyUp(pq.arr, index);
129     }
130     else {
131         heapifyDown(pq.arr, index);
132     }
133 }
```

```
1 //Red black Tree
2 #include <iostream>
3
4 using namespace std;
5
6 // Enum representing the color of a node in the red-black tree
7 enum Color {
8     RED, BLACK,
9 };
10
11 // Node structure for the red-black tree
12 typedef struct RBNode* Ref;
13 struct RBNode {
14     int key;
15     Color color;
16     Ref parent, left, right;
17 };
18
19 // Global variable representing the nil (sentinel) node
20 Ref nil;
21
22 // Function to create a new node with given key, color, and nil reference
23 Ref createNode(int key, Color color, Ref nil) {
24     Ref p = new RBNode{ key, color, nil, nil, nil };
25     return p;
26 }
27
28 // Left rotation operation in the red-black tree
29 void leftRotate(Ref& root, Ref x) {
30     Ref y = x->right;
31     x->right = y->left;
32
33     if (y->left != nil) {
34         y->left->parent = x;
35     }
36     y->parent = x->parent;
37
38     if (x->parent == nil) {
39         root = y;
40     }
41     else {
42         if (x == x->parent->left) {
43             x->parent->left = y;
44         }
45         else {
46             x->parent->right = y;
47         }
48     }
49 }
```

```
50     y->left = x;
51     x->parent = y;
52 }
53
54 // Right rotation operation in the red-black tree
55 void rightRotate(Ref& root, Ref x) {
56     Ref y = x->left;
57     x->left = y->right;
58
59     if (y->right != nil) {
60         y->right->parent = x;
61     }
62     y->parent = x->parent;
63
64     if (x->parent == nil) {
65         root = y;
66     }
67     else {
68         if (x == x->parent->right) {
69             x->parent->right = y;
70         }
71         else {
72             x->parent->left = y;
73         }
74     }
75
76     y->right = x;
77     x->parent = y;
78 }
79
80 // Binary Search Tree (BST) insertion operation
81 void BST_Insert(Ref& root, Ref x) {
82     Ref y = nil, z = root;
83     while (z != nil) {
84         y = z;
85
86         if (x->key < z->key) z = z->left;
87         else if (x->key > z->key) z = z->right;
88         else return; // Key already exists, do nothing
89     }
90
91     x->parent = y;
92     if (y == nil) root = x;
93     else {
94         if (x->key < y->key) y->left = x;
95         else y->right = x;
96     }
97 }
98
```

```
99 // Adjustments after left child insertion in the red-black tree
100 void insertionLeftAdjust(Ref& root, Ref& x) {
101     Ref u = x->parent->parent->right;
102     if (u->color == RED) {
103         x->parent->color = u->color = BLACK;
104         x->parent->parent->color = RED;
105         x = x->parent->parent;
106     }
107     else {
108         if (x == x->parent->right) {
109             x = x->parent;
110             leftRotate(root, x);
111         }
112         x->parent->color = BLACK;
113         x->parent->color = BLACK;
114         x->parent->parent->color = RED;
115         rightRotate(root, x->parent->parent);
116     }
117 }
118
119 // Adjustments after right child insertion in the red-black tree
120 void insertionRightAdjust(Ref& root, Ref& x) {
121     Ref u = x->parent->parent->left;
122     if (u->color == RED) {
123         x->parent->color = u->color = BLACK;
124         x->parent->parent->color = RED;
125         x = x->parent->parent;
126     }
127     else {
128         if (x == x->parent->left) {
129             x = x->parent;
130             rightRotate(root, x);
131         }
132         x->parent->color = BLACK;
133         x->parent->color = BLACK;
134         x->parent->parent->color = RED;
135         leftRotate(root, x->parent->parent);
136     }
137 }
138
139 // Fix-up routine after insertion in the red-black tree
140 void insertionFixUp(Ref& root, Ref x) {
141     while (x->parent->color == RED) {
142         if (x->parent == x->parent->parent->left) {
143             insertionLeftAdjust(root, x);
144         }
145         else {
146             insertionRightAdjust(root, x);
147         }
148     }
149 }
```

```
148     }
149     root->color = BLACK;
150 }
151
152 // Insert a key into the red-black tree
153 void Insert(Ref& root, int key) {
154     Ref x = createNode(key, RED, nil);
155     BST_Insert(root, x);
156     insertionFixUp(root, x);
157 }
158
159 // Create a red-black tree from an array of keys
160 Ref createTree(int a[], int n) {
161     Ref root = nil;
162
163     for (int i = 0; i < n; i++) {
164         Insert(root, a[i]);
165     }
166
167     return root;
168 }
169
170 // Adjustments after left child deletion in the red-black tree
171 void deleteLeftAdjust(Ref& root, Ref& x) {
172     Ref w = x->parent->right;
173
174     if (w->color == RED) {
175         w->color = BLACK;
176         x->parent->color = RED;
177         leftRotate(root, x->parent);
178         w = x->parent->right;
179     }
180
181     if ((w->right->color == BLACK) && (w->left->color == BLACK)) {
182         w->color = RED;
183         x = x->parent;
184     }
185     else {
186         if (w->right->color == BLACK) {
187             w->left->color = BLACK;
188             w->color = RED;
189             rightRotate(root, w);
190             w = x->parent->right;
191         }
192         w->color = x->parent->color;
193         x->parent->color = w->right->color = BLACK;
194         leftRotate(root, x->parent);
195         x = root;
196     }
```



```
197 }
198
199 // Adjustments after right child deletion in the red-black tree
200 void deleteRightAdjust(Ref& root, Ref& x) {
201     Ref w = x->parent->left;
202
203     if (w->color == RED) {
204         w->color = BLACK;
205         x->parent->color = RED;
206         leftRotate(root, x->parent);
207         w = x->parent->left;
208     }
209
210     if ((w->left->color == BLACK) && (w->right->color == BLACK)) {
211         w->color = RED;
212         x = x->parent;
213     }
214     else {
215         if (w->left->color == BLACK) {
216             w->right->color = BLACK;
217             w->color = RED;
218             leftRotate(root, w);
219             w = x->parent->left;
220         }
221         w->color = x->parent->color;
222         x->parent->color = w->left->color = BLACK;
223         rightRotate(root, x->parent);
224         x = root;
225     }
226 }
227
228 // Fix-up routine after deletion in the red-black tree
229 void deleteFixUp(Ref root, Ref x) {
230     while ((x->color == BLACK) && (x != root)) {
231         if (x == x->parent->left) deleteLeftAdjust(root, x);
232         else deleteRightAdjust(root, x);
233     }
234     x->color = BLACK;
235 }
236
237 // Search for a key in the red-black tree and return the corresponding node
238 Ref lookup(Ref root, int key) {
239     Ref p = root;
240     while (p != nil) {
241         if (key == p->key) return p;
242
243         if (key < p->key) p = p->left;
244         else p = p->right;
```

```
245     }
246
247     return nil;
248 }
249
250 // Find the predecessor of a given node in the red-black tree
251 Ref findPredecessor(Ref z) {
252     if (z->left == nil) {
253         std::cout << "This node does not have predecessor!";
254         return nullptr;
255     }
256
257     Ref y = z->left;
258     while (y->right != nil) y = y->right;
259     return y;
260 }
261
262 // Remove a key from the red-black tree
263 void Remove(Ref& root, int k) {
264     Ref z = lookup(root, k);
265     if (z == nil) return;
266
267     Ref y = (z->left == nil) || (z->right == nil) ? z : findPredecessor  ↗
        (z);
268
269     Ref x = (y->left == nil) ? y->right : y->left;
270
271     x->parent = y->parent;
272     if (y->parent == nil) root = x;
273     else {
274         if (y == y->parent->left) y->parent->left = x;
275         else y->parent->right = x;
276     }
277
278     if (y != z) z->key = y->key;
279     if (y->color == BLACK) {
280         deleteFixUp(root, x);
281     }
282
283     delete y;
284 }
285
286 struct Node {
287     int key;
288     Node* left, * right;
289     int height;
290 };
291
292 bool isGreater(int a, int b) {
```

```
293     return a > b;
294 }
295
296 bool isSmaller(int a, int b) {
297     return a < b;
298 }
299
300 bool isEqual(int a, int b) {
301     return a == b;
302 }
303
304 int getHeight(Node* root) {
305     if (!root) return 0;
306     return root->height;
307 }
308
309 void fixHeight(Node*& root) {
310     root->height = 1 + std::max(getHeight(root->left), getHeight(root-
311     >right));
312 }
313
314 void leftRotate(Node*& root) {
315     Node* B = root->right;
316     Node* Y = B->left;
317
318     B->left = root;
319     root->right = Y;
320
321     fixHeight(root);
322     fixHeight(B);
323
324     root = B;
325 }
326
327 void rightRotate(Node*& root) {
328     Node* B = root->left;
329     Node* Y = B->right;
330
331     B->right = root;
332     root->left = Y;
333
334     fixHeight(root);
335     fixHeight(B);
336
337     root = B;
338 }
339
340 int getBalanceFactor(Node* node) {
341     return getHeight(node->left) - getHeight(node->right);
```

```
341 }
342
343 void insertNode(Node*& root, int key) {
344     if (!root) {
345         root = new Node{ key, nullptr, nullptr, 1 };
346         return;
347     }
348
349     if (isSmaller(key, root->key)) {
350         insertNode(root->left, key);
351     }
352     else if (isGreater(key, root->key)) {
353         insertNode(root->right, key);
354     }
355     else return;
356
357     fixHeight(root);
358
359     int bf = getBalanceFactor(root);
360
361     if (bf > 1 && isSmaller(key, root->left->key)) {
362         rightRotate(root);
363         return;
364     }
365
366     if (bf < -1 && isGreater(key, root->right->key)) {
367         leftRotate(root);
368         return;
369     }
370
371     if (bf > 1 && isGreater(key, root->left->key)) {
372         leftRotate(root->left);
373         rightRotate(root);
374         return;
375     }
376
377     if (bf < -1 && isSmaller(key, root->right->key)) {
378         rightRotate(root->right);
379         leftRotate(root);
380         return;
381     }
382 }
383
384 Node* findPredecessor(Node* node) {
385     if (!node->left) {
386         std::cout << "This node does not have predecessor!";
387         return nullptr;
388     }
389 }
```

```
390     Node* y = node->left;
391     while (y->right) y = y->right;
392     return y;
393 }
394
395 void deleteNode(Node*& root, int key) {
396     if (!root) return;
397     else if (isSmaller(key, root->key)) deleteNode(root->left, key);
398     else if (isGreater(key, root->key)) deleteNode(root->right, key);
399     else {
400         if (!root->left) {
401             Node* temp = root->right;
402             delete root;
403             root = temp;
404             return;
405         }
406         else if (!root->right) {
407             Node* temp = root->left;
408             delete root;
409             root = temp;
410             return;
411         }
412         else {
413             Node* pred = findPredecessor(root);
414             root->key = pred->key;
415
416             deleteNode(root->left, pred->key);
417         }
418     }
419
420     fixHeight(root);
421
422     int bf = getBalanceFactor(root);
423
424     if (bf > 1 && getBalanceFactor(root->left) >= 0) {
425         rightRotate(root);
426     }
427
428     else if (bf < -1 && getBalanceFactor(root->right) <= 0) {
429         leftRotate(root);
430     }
431
432     else if (bf > 1 && getBalanceFactor(root->left) < 0) {
433         leftRotate(root->left);
434         rightRotate(root);
435     }
436
437     else if (bf < -1 && getBalanceFactor(root->right) > 0) {
438         rightRotate(root->right);
```

```
439         leftRotate(root);  
440     }  
441 }
```

```

1 //TOPO SORT
2
3 //(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
4 //(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
5
6 #include <iostream>
7 #include <fstream>
8 #include <vector>
9
10 // Forward declaration of Leader and Trailer structures
11 typedef struct Leader* lref;
12 typedef struct Trailer* tref;
13
14 // Structure representing a Leader node
15 struct Leader {
16     int key;           // Key of the leader
17     int count;         // Number of incoming precedences
18     lref next;         // Pointer to the next leader node in the list
19     tref trails;       // Pointer to the list of trailers
20 };
21
22 // Structure representing a Trailer node
23 struct Trailer {
24     lref id;           // Pointer to the leader node
25     tref next;        // Pointer to the next trailer node in the list
26 };
27
28 // Function to find the leader with key x; if not exist yet, add to the ↗
   end of the leader list
29 lref findLeader(lref& head, lref& tail, int x) {
30     lref p = head;
31
32     tail->key = x;
33
34     while (p->key != x) {
35         p = p->next;
36     }
37
38     if (p == tail) {
39         tail = new Leader;
40
41         p->count = 0;
42         p->trails = nullptr;
43         p->next = tail;
44     }
45
46     return p;

```

```
47 }
48
49 // Function to split leaders with no precedences from the leader list
50 void splitLeaderWithNoPrecedence(lref& head, lref& tail) {
51     lref p = head;
52     head = nullptr;
53
54     while (p != tail) {
55         lref tmp = p->next;
56
57         if (p->count == 0) {
58             p->next = head;
59             head = p;
60         }
61
62         p = tmp;
63     }
64 }
65
66 // Function to add a new order x < y
67 void addOrder(lref& head, lref& tail, int x, int y) {
68     lref xNode = findLeader(head, tail, x);
69     lref yNode = findLeader(head, tail, y);
70
71     tref xTrail = new Trailer{ yNode, xNode->trails };
72     xNode->trails = xTrail;
73
74     // Increase the number of precedences
75     yNode->count++;
76 }
77
78 // Function to create leaders from pairs of orders
79 void createLeadersFromPairs(lref& head, lref& tail,
80     std::vector<std::pair<int, int>> orders) {
81     head = new Leader{ -1, 0, nullptr, nullptr };
82     tail = head;
83
84     for (int i = 0; i < orders.size(); i++) {
85         addOrder(head, tail, orders[i].first, orders[i].second);
86     }
87
88 // Function to perform topological sort based on the given orders
89 void topoSort(std::vector<std::pair<int, int>> orders) {
90     lref head, tail;
91     createLeadersFromPairs(head, tail, orders);
92
93     splitLeaderWithNoPrecedence(head, tail);
94 }
```



```
95     lref p = head;
96
97     while (p) {
98         std::cout << p->key << " ";
99
100        tref t = p->trails;
101
102        p = p->next;
103
104        for (tref q = t; q; q = q->next) {
105            lref succNode = q->id;
106
107            succNode->count--;
108
109            if (succNode->count == 0) {
110                succNode->next = p;
111
112                p = succNode;
113            }
114        }
115    }
116 }
117
118 // Function to parse orders from a file and return a vector of pairs
119 std::vector<std::pair<int, int>> parseFile(std::string fileName) {
120     std::vector<std::pair<int, int>> orders;
121
122     std::pair<int, int> p;
123     char ch1, ch2, ch3;
124
125     std::ifstream inFile;
126     inFile.open(fileName);
127
128     while (!inFile.eof()) {
129         inFile >> ch1 >> p.first >> ch2 >> p.second >> ch3;
130
131         if (inFile.eof()) {
132             break;
133         }
134
135         orders.push_back(p);
136     }
137
138     inFile.close();
139
140     return orders;
141 }
142
143 // Main function
```

```
144 int main() {
145     // Parse orders from the input file
146     std::vector<std::pair<int, int>> orders = parseFile("input.txt");
147
148     // Perform topological sort and print the result
149     topoSort(orders);
150
151     return 0;
152 }
153
```