ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

2η Εργασία του Μαθήματος Δομές Δεδομένων

Π21127 Ματίνα Παπαδάκου

Περιγραφή Κλάσεων και Μεθόδων:

Εισαγωγή:

Ο κώδικας που παρουσιάζεται υλοποιεί ένα AVL δέντρο σε γλώσσα προγραμματισμού C++. Ένα AVL δέντρο είναι ένας ειδικός τύπος δυαδικού δέντρου αναζήτησης που διατηρεί την ισορροπία του μέσω των περιστροφών. Αυτό το δέντρο είναι σχεδιασμένο να εκτελεί τις βασικές λειτουργίες (εισαγωγή, διαγραφή, αναζήτηση) με πολύ αποδοτικό τρόπο, εξασφαλίζοντας ότι το ύψος του δέντρου παραμένει λογαριθμικό ως προς τον αριθμό των στοιχείων του.

Δομή του Κώδικα:

Δομή Κόμβου (Node):

```
// Δομή ενός κόμβου στο AVL δέντρο
struct Node {
    int key;        // Το κλειδί του κόμβου
    Node* left;     // Δείκτης προς το αριστερό παιδί
    Node* right;    // Δείκτης προς το δεξί παιδί
    int height;     // Ύψος του κόμβου
    Node(int k) : key(k), left(nullptr), right(nullptr), height(1) {} // Αρχικοποίηση κόμβου
};
```

Ο κάθε κόμβος του δέντρου έχει ένα κλειδί (key), δείκτες προς το αριστερό (left) και το δεξί παιδί (right), και το ύψος του κόμβου (height). Ο constructor του κόμβου αρχικοποιεί τις τιμές αυτές.

Κλάση AVLTree – public Μέθοδοι:

```
class AVLTree {
public:
    AVLTree() : root(nullptr) {}

    // Εισαγωγή νέου κλειδιού στο AVL δέντρο
    void insert(int key) {
        root = insert(root, key);
    }

    // Αφαίρεση κλειδιού από το AVL δέντρο
    void remove(int key) {
        root = remove(root, key);
    }

    // Αναζήτηση κλειδιού στο AVL δέντρο
    bool search(int key) {
        return search(root, key) != nullptr;
    }

    // Συνένωση δύο AVL δέντρων με ένα ενδιάμεσο κλειδί
    void join(AVLTree& t1, int k, AVLTree& t2) {
        root = join(t1.root, k, t2.root);
    }

    // Διάσπαση του AVL δέντρου σε δύο δέντρα βάσει ενός κλειδιού
    void split(int k, AVLTree& t1, AVLTree& t2) {
        split(root, k, t1.root, t2.root);
    }

    // Εύρεση κλειδιών σε συγκεκριμένο εύρος [k1, k2]
    vector<int> find(int k1, int k2) {
        vector<int> result;
        find(root, k1, k2, result);
        return result;
    }

    // Εύρεση γειτονικού κλειδιού (προκατόχου ή διαδόχου) βάσει κατεύθυνσης
    int findNeighbor(int k, string direction) {
        Node* neighbor = nullptr;
        if (direction == "predecessor") {
            findPredecessor(root, k, neighbor);
        }
        else if (direction == "successor") {
            findSuccessor(root, k, neighbor);
        }
        return neighbor ? neighbor->key : -1; // Επιστροφή -1 αν δεν βρεθεί γείτονας
    }
```

AVLTree(): Ο κατασκευαστής αρχικοποιεί τη ρίζα του δέντρου σε nullptr.

Βασικές Λειτουργίες:

void insert(int key): Εισάγει ένα νέο κλειδί στο δέντρο.

void remove(int key): Αφαιρεί ένα κλειδί από το δέντρο.

bool search(int key): Αναζητά ένα κλειδί στο δέντρο και επιστρέφει true αν βρεθεί.

Συνένωση και Διάσπαση:

void join(AVLTree& t1, int k, AVLTree& t2): Συνενώνει δύο AVL δέντρα t1 και t2 με ένα ενδιάμεσο κλειδί k.

void split(int k, AVLTree& t1, AVLTree& t2): Διασπά το δέντρο σε δύο δέντρα t1 και t2 βάσει του κλειδιού k.

Εύρεση και Γειτονικά Κλειδιά:

vector<int> find(int k1, int k2): Επιστρέφει όλα τα κλειδιά στο εύρος [k1, k2].

int findNeighbor(int k, string direction): Επιστρέφει τον προκατόχο ή τον διάδοχο ενός κλειδιού k ανάλογα με την κατεύθυνση (predecessor ή successor).

Κλάση AVLTree – private Μέθοδοι και Μεταβλητές:

int height(Node* n): Επιστρέφει το ύψος ενός κόμβου.

```
// Επιστροφή του ύψους ενός κόμβου
int height(Node* n) {
    return n ? n->height : 0;
}
```

int balanceFactor(Node* n): Υπολογίζει τον συντελεστή ισορροπίας ενός κόμβου.

```
// Υπολογισμός του συντελεστή ισορροπίας ενός κόμβου
int balanceFactor(Node* n) {
    return n ? height(n->left) - height(n->right) : 0;
}
```

Node* rotateRight(Node* y), Node* rotateLeft(Node* x): Δεξιά και αριστερή περιστροφή αντίστοιχα. Αυτή η διαδικασία βοηθά στην αποκατάσταση της ισορροπίας του δέντρου όταν το υποδέντρο με ρίζα τον κόμβο y/x έχει υπερβολικά μεγάλο ύψος στο αριστερό/δεξιό του υποδέντρο.

Ανάλυση δεξιάς περιστροφής: Η δεξιά περιστροφή χρησιμοποιείται όταν το αριστερό υποδέντρο είναι βαρύτερο από το δεξί, και συγκεκριμένα στην περίπτωση που έχουμε μια ανισορροπία τύπου "LL" (Left-Left). Αυτή η περιστροφή μειώνει το ύψος του αριστερού υποδέντρου και αποκαθιστά την ισορροπία.

Ανάλυση αριστερής περιστροφής: Η αριστερή περιστροφή χρησιμοποιείται όταν το δεξί υποδέντρο είναι βαρύτερο από το αριστερό, και συγκεκριμένα στην περίπτωση που έχουμε μια ανισορροπία τύπου "RR" (Right-Right). Αυτή η περιστροφή μειώνει το ύψος του δεξιού υποδέντρου και αποκαθιστά την ισορροπία.

```
// Δεξιά περιστροφή ενός υποδέντρου με ρίζα τον y
Node* rotateRight(Node* y) {
    Node* x = y->left;
    Node* T2 = x->right;
    x->right = y;
    y->left = T2;
    // Ενημέρωση των υψών
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

// Αριστερή περιστροφή ενός υποδέντρου με ρίζα τον x
Node* rotateLeft(Node* x) {
    Node* y = x->right;
    Node* T2 = y->left;
    y->left = x;
    x->right = T2;
    // Ενημέρωση των υψών
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
}
```

Node* insert(Node* node, int key): Εισάγει ένα νέο κλειδί στο υποδέντρο με ρίζα τον node.

```
// Εισαγωγή νέου κλειδιού στο δέντρο με διατήρηση της ισορροπίας
Node* insert(Node* node, int key) {
    // Αν ο κόμβος είναι κενός, δημιουργία νέου κόμβου
    if (!node) return new Node(key);

    // Καθορισμός της θέσης εισαγωγής
    if (key < node->key) {
        node->left = insert(node->left, key);
    }
    else if (key > node->key) {
        node->right = insert(node->right, key);
    }
    else {
        return node; // Διπλά κλειδιά δεν επιτρέπονται
    }

    // Ενημέρωση του ύψους του κόμβου
    node->height = 1 + max(height(node->left), height(node->right));

    // Έλεγχος και διόρθωση του συντελεστή ισορροπίας
    int balance = balanceFactor(node);

    // LL περίπτωση
    if (balance > 1 && key < node->left->key) {
        return rotateRight(node);
    }

    // RR περίπτωση
    if (balance < -1 && key > node->right->key) {
        return rotateLeft(node);
    }

    // LR περίπτωση
    if (balance > 1 && key > node->left->key) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    // RL περίπτωση
    if (balance < -1 && key < node->right->key) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}
```

Ο παρακάτω κώδικας υλοποιεί την εισαγωγή ενός νέου κλειδιού σε ένα AVL δέντρο με τη διατήρηση της ισορροπίας του. Αυτό σημαίνει ότι μετά την εισαγωγή ενός νέου κόμβου, ελέγχονται οι συντελεστές ισορροπίας σε όλους τους κόμβους από τον εισαγόμενο κόμβο προς τη ρίζα του δέντρου και γίνονται αναγκαίες περιστροφές για να διορθωθεί η ισορροπία αν αυτή έχει διαταραχθεί.

- Αρχικά ελέγχεται αν ο κόμβος είναι κενός. Αν ο κόμβος node είναι nullptr, τότε δημιουργείται ένας νέος κόμβος με το κλειδί key και επιστρέφεται αυτός ο νέος κόμβος ως η νέα ρίζα του υποδέντρου.
- Στην συνέχεια καθορίζεται η θέση εισαγωγής. Το κλειδί key συγκρίνεται με το κλειδί του τρέχοντος κόμβου node->key. Αν το key είναι μικρότερο από το node->key, τότε εισάγεται στο αριστερό υποδέντρο του node. Αν είναι μεγαλύτερο, εισάγεται στο δεξί υποδέντρο. Αν είναι ίσο, τότε επιστρέφεται απευθείας ο τρέχον κόμβος node χωρίς να γίνει εισαγωγή (διπλά κλειδιά δεν επιτρέπονται στο AVL δέντρο).

- Μετά από την εισαγωγή, ενημερώνεται το ύψος του κόμβου `node` με βάση τα ύψη των αριστερού και δεξιού του υποδέντρου.
- Υπολογίζεται ο συντελεστής ισορροπίας του κόμβου node και διορθώνεται η ισορροπία του.
- Γίνονται οι διορθώσεις με τις περιστροφές.
    - LL περίπτωση: Αν ο συντελεστής ισορροπίας είναι μεγαλύτερος από 1 και το key είναι μικρότερο από το κλειδί του αριστερού παιδιού του node, τότε γίνεται δεξιά περιστροφή (rotateRight) για να επαναφερθεί η ισορροπία.
    - RR περίπτωση: Σε αυτήν την περίπτωση, γίνεται αριστερή περιστροφή (rotateLeft) στον κόμβο node, προκειμένου να διορθωθεί η ανισορροπία και να επαναφερθεί η ισορροπία του AVL δέντρου. Η αριστερή περιστροφή συνίσταται στην αλλαγή των ριζών των δύο υποδέντρων και στην ενημέρωση των υψών των κόμβων.
    - LR περίπτωση: Αν ο συντελεστής ισορροπίας είναι μεγαλύτερος από 1 και το key είναι μεγαλύτερο από το κλειδί του αριστερού παιδιού του node, τότε γίνεται αριστερή περιστροφή (rotateLeft) στο αριστερό παιδί και στη συνέχεια δεξιά περιστροφή (rotateRight) στον κόμβο node για να επαναφερθεί η ισορροπία.
    - RL περίπτωση: Αν ο συντελεστής ισορροπίας είναι μικρότερος από -1 και το key είναι μικρότερο από το κλειδί του δεξιού παιδιού του node, τότε γίνεται δεξιά περιστροφή (rotateRight) στο δεξιό παιδί και στη συνέχεια αριστερή περιστροφή (rotateLeft) στον κόμβο node για να επαναφερθεί η ισορροπία.

Node* minValueNode(Node* node): Η συνάρτηση minValueNode χρησιμοποιείται για την εύρεση του κόμβου με την ελάχιστη τιμή κλειδιού σε ένα δέντρο αναζήτησης, όπως είναι το AVL δέντρο. Εξετάζει συνεχώς το αριστερό παιδί κάθε κόμβου μέχρι να φτάσει στον κόμβο με το ελάχιστο κλειδί του δέντρου και τον επιστρέφει.

```
// Εύρεση του κόμβου με την ελάχιστη τιμή κλειδιού σε ένα δέντρο
Node* minValueNode(Node* node) {
    Node* current = node;
    // Διασχίζουμε προς τα αριστερά για να βρούμε το μικρότερο κλειδί
    while (current->left != nullptr) {
        current = current->left;
    }
    return current;
}
```

1. Αρχικοποίηση Μεταβλητής: Ξεκινάμε με τον κόμβο node ως αρχικό κόμβο από όπου θέλουμε να ξεκινήσουμε την αναζήτηση του ελάχιστου κλειδιού.

2. Διάσχιση Προς τα Αριστερά: Χρησιμοποιούμε μια βρόχο while για να διασχίσουμε το δέντρο προς τα αριστερά. Η συνθήκη current->left != nullptr ελέγχει αν ο τρέχον κόμβος έχει ένα αριστερό παιδί. Όσο υπάρχει αριστερό παιδί, μετακινούμε τον δείκτη current στο αριστερό παιδί του τρέχοντος κόμβου.
3. Επιστροφή Ελάχιστου Κόμβου: Όταν δεν υπάρχει πλέον αριστερό παιδί (current->left == nullptr), ο current δείχνει στον κόμβο που περιέχει το ελάχιστο κλειδί στο δέντρο. Στη συνέχεια, η συνάρτηση επιστρέφει τον κόμβο current.

Node* remove(Node* root, int key): Η συνάρτηση remove υλοποιεί την αφαίρεση ενός κλειδιού από ένα AVL δέντρο με διατήρηση της ισορροπίας του

```
// Αφαίρεση ενός κλειδιού από το δέντρο με διατήρηση της ισορροπίας
Node* remove(Node* root, int key) {
    if (!root) return root;

    // Εύρεση του κόμβου που θα αφαιρεθεί
    if (key < root->key) {
        root->left = remove(root->left, key);
    }
    else if (key > root->key) {
        root->right = remove(root->right, key);
    }
    else {
        // Ο κόμβος βρέθηκε
        if (!root->left || !root->right) {
            Node* temp = root->left ? root->left : root->right;
            if (!temp) {
                // Ο κόμβος δεν έχει παιδιά
                temp = root;
                root = nullptr;
            }
            else {
                // Ένας από τους δύο κόμβους είναι null
                *root = *temp;
            }
            delete temp;
        }
        else {
            // Ο κόμβος έχει δύο παιδιά
            Node* temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = remove(root->right, temp->key);
        }
    }

    if (!root) return root;

    // Ενημέρωση του ύψους του κόμβου
    root->height = 1 + max(height(root->left), height(root->right));

    // Έλεγχος και διόρθωση του συντελεστή ισορροπίας
    int balance = balanceFactor(root);

    // LL περίπτωση
    if (balance > 1 && balanceFactor(root->left) >= 0) {
        return rotateRight(root);
    }
```

```
    // LR περίπτωση
    if (balance > 1 && balanceFactor(root->left) < 0) {
        root->left = rotateLeft(root->left);
        return rotateRight(root);
    }

    // RR περίπτωση
    if (balance < -1 && balanceFactor(root->right) <= 0) {
        return rotateLeft(root);
    }

    // RL περίπτωση
    if (balance < -1 && balanceFactor(root->right) > 0) {
        root->right = rotateRight(root->right);
        return rotateLeft(root);
    }

    return root;
}
```

Αρχικά ο κώδικας:

1. Ελέγχει Αν ο Κόμβος Είναι Κενός: Αν ο κόμβος root είναι κενός (nullptr), η συνάρτηση επιστρέφει αμέσως τον κόμβο root, καθώς δεν χρειάζεται να γίνει κάποια αλλαγή.
2. Αναζητάει το Κλειδί για Αφαίρεση: Αρχίζουμε την αναζήτηση του κλειδιού που θέλουμε να αφαιρέσουμε στο AVL δέντρο. Αν το key είναι μικρότερο από το root->key, τότε την αναζήτηση τη συνεχίζουμε στο αριστερό υποδέντρο (root->left). Αν είναι μεγαλύτερο, στο δεξί υποδέντρο (root->right). Αν το key ισούται με το root->key, τότε έχουμε βρει τον κόμβο που θέλουμε να αφαιρέσουμε.
3. Αν βρεθεί ο κόμβος που θέλουμε να αφαιρέσουμε:
   - Αν ο κόμβος έχει μηδενικό ή ένα παιδί: Δημιουργούμε έναν προσωρινό κόμβο temp που θα είναι είτε το αριστερό είτε το δεξί παιδί του root. Αν αμφότερα είναι nullptr, τότε θέτουμε τον temp ίσο με τον root και θέτουμε τον root σε nullptr (δηλαδή διαγράφουμε τον κόμβο).
   - Αν ο κόμβος έχει δύο παιδιά: Βρίσκουμε τον επόμενο κόμβο με τη μικρότερη τιμή στο δεξιό υποδέντρο του root με τη βοήθεια της συνάρτησης minValueNode. Αντιγράφουμε το κλειδί αυτού του επόμενου κόμβου στον root και στη συνέχεια αφαιρούμε αυτόν τον επόμενο κόμβο από το δέντρο αποκαθιστώντας την ισορροπία.
4. Ενημερώνουμε το Ύψος: Αφού έχει γίνει η αφαίρεση, ενημερώνουμε το ύψος του κόμβου root υπολογίζοντας το μέγιστο ύψος των παιδιών του.
5. Γίνεται Έλεγχος και Διόρθωση της Ισορροπίας: Υπολογίζουμε τον συντελεστή ισορροπίας του root και ελέγχουμε αν χρειάζεται να γίνει κάποια διόρθωση ισορροπίας σε κάποια από τις περιπτώσεις LL, LR, RR, RL χρησιμοποιώντας τις αντίστοιχες περιστροφές.

Αυτή η διαδικασία επαναλαμβάνεται αναδρομικά μέχρι να επιστραφεί ο τροποποιημένος κόμβος root με την κατάλληλη ισορροπία. Με αυτόν τον τρόπο, η συνάρτηση remove αφαιρεί ένα κλειδί από το AVL δέντρο ενώ διατηρεί την ισορροπία του.

Node* search(Node* root, int key): Η συνάρτηση search υλοποιεί την αναζήτηση ενός κλειδιού σε ένα AVL δέντρο.

```
// Αναζήτηση ενός κλειδιού στο δέντρο
Node* search(Node* root, int key) {
    // Αν ο κόμβος είναι κενός ή βρέθηκε το κλειδί
    if (!root || root->key == key) return root;
    if (key < root->key) return search(root->left, key);
    return search(root->right, key);
}
```

1. Έλεγχος Αν ο Κόμβος Είναι Κενός ή το Κλειδί Βρέθηκε:

   Η συνάρτηση πρώτα ελέγχει αν ο κόμβος root είναι κενός (nullptr). Αν ναι, τότε επιστρέφει τον root, καθώς το κλειδί δεν υπάρχει στο δέντρο.

   Αν ο κόμβος root δεν είναι κενός, τότε ελέγχει αν το κλειδί του κόμβου (root->key) ισούται με το key. Αν είναι ίσο, τότε έχει βρεθεί ο κόμβος που περιέχει το ζητούμενο κλειδί, και η συνάρτηση επιστρέφει αυτόν τον κόμβο.

2. Αναζήτηση στο Αριστερό Υποδέντρο:

   Αν το key είναι μικρότερο από το root->key, τότε η συνάρτηση καλείται αναδρομικά για το αριστερό υποδέντρο (root->left). Αυτό σημαίνει ότι συνεχίζει την αναζήτηση στο αριστερό υποδέντρο του τρέχοντος κόμβου, επειδή τα κλειδιά στο αριστερό υποδέντρο είναι μικρότερα από το κλειδί του τρέχοντος κόμβου.

3. Αναζήτηση στο Δεξί Υποδέντρο:

   Αν το key είναι μεγαλύτερο από το root->key, τότε η συνάρτηση καλείται αναδρομικά για το δεξί υποδέντρο (root->right). Αυτό σημαίνει ότι συνεχίζει την αναζήτηση στο δεξί υποδέντρο του τρέχοντος κόμβου, επειδή τα κλειδιά στο δεξί υποδέντρο είναι μεγαλύτερα από το κλειδί του τρέχοντος κόμβου.

Η διαδικασία αυτή συνεχίζεται αναδρομικά μέχρι να βρεθεί ο κόμβος που περιέχει το ζητούμενο κλειδί ή μέχρι να φτάσουμε σε έναν κενό κόμβο, που σημαίνει ότι το κλειδί δεν υπάρχει στο δέντρο. Στην περίπτωση που το κλειδί βρεθεί, η συνάρτηση επιστρέφει τον κόμβο που περιέχει το κλειδί, αλλιώς επιστρέφει nullptr.

Node* join(Node* T1, int k, Node* T2):

```
// Συνένωση δύο δέντρων με ένα ενδιάμεσο κλειδί
Node* join(Node* T1, int k, Node* T2) {
    if (!T1) return insert(T2, k);
    if (!T2) return insert(T1, k);
    Node* newRoot = new Node(k);
    newRoot->left = T1;
    newRoot->right = T2;
    newRoot->height = 1 + max(height(newRoot->left), height(newRoot->right));
    return balance(newRoot);
}
```

1. **Έλεγχος για Κενά Δέντρα**:

   - Αν το δέντρο T1 είναι κενό (nullptr), τότε η συνάρτηση επιστρέφει το αποτέλεσμα της εισαγωγής του κλειδιού k στο δέντρο T2. Δηλαδή, το δέντρο T2 με το κλειδί k εισαγόμενο σε αυτό.

- Αν το δέντρο T2 είναι κενό (nullptr), τότε η συνάρτηση επιστρέφει το αποτέλεσμα της εισαγωγής του κλειδιού k στο δέντρο T1. Δηλαδή, το δέντρο T1 με το κλειδί k εισαγόμενο σε αυτό.

2. **Δημιουργία Νέας Ρίζας**:

Αν και τα δύο δέντρα δεν είναι κενά, η συνάρτηση δημιουργεί έναν νέο κόμβο (newRoot) με κλειδί k. Αυτός ο κόμβος θα είναι η νέα ρίζα του συνενωμένου δέντρου.

3. **Εισαγωγή των Δέντρων στα Παιδιά του Νέου Κόμβου**:

- Ορίζει το αριστερό παιδί του νέου κόμβου (newRoot->left) να είναι το δέντρο T1.
- Ορίζει το δεξί παιδί του νέου κόμβου (newRoot->right) να είναι το δέντρο T2.

4. **Ενημέρωση του Ύψους του Νέου Κόμβου**:

Ενημερώνει το ύψος του νέου κόμβου (newRoot) ως το μέγιστο ύψος ανάμεσα στο αριστερό και το δεξί παιδί του, συν ένα.

5. **Ισορροπία του Νέου Δέντρου**:

Καλεί τη συνάρτηση balance για να εξασφαλίσει ότι το νέο δέντρο με ρίζα τον newRoot παραμένει ισορροπημένο, δηλαδή ικανοποιεί τις ιδιότητες του AVL δέντρου.

4. Τελική Επιστροφή:

Η συνάρτηση επιστρέφει τη ρίζα του νέου, ισορροπημένου AVL δέντρου που προκύπτει από τη συνένωση των δέντρων T1 και T2 με το ενδιάμεσο κλειδί k.

void split(Node* T, int k, Node*& T1, Node*& T2): Η συνάρτηση split διασπά ένα AVL δέντρο T σε δύο υποδέντρα T1 και T2 βάσει ενός κλειδιού k.

```
// Διάσπαση του δέντρου σε δύο υποδέντρα βάσει ενός κλειδιού
void split(Node* T, int k, Node*& T1, Node*& T2) {
    if (!T) {
        T1 = T2 = nullptr;
        return;
    }
    if (k < T->key) {
        T2 = T;
        split(T->left, k, T1, T2->left);
        T2->height = 1 + max(height(T2->left), height(T2->right));
    }
    else {
        T1 = T;
        split(T->right, k, T1->right, T2);
        T1->height = 1 + max(height(T1->left), height(T1->right));
    }
    T1 = balance(T1);
    T2 = balance(T2);
}
```

1. **Αρχικός έλεγχος**:
    - Εάν το δέντρο T είναι κενό (nullptr), τότε και τα δύο υποδέντρα T1 και T2 ορίζονται ως κενά (nullptr) και η συνάρτηση επιστρέφει.
2. **Διάσπαση βάσει του κλειδιού k**:
    - Εάν το κλειδί k είναι μικρότερο από το κλειδί του τρέχοντος κόμβου T:
        - Το δέντρο T2 ορίζεται ως το τρέχον δέντρο T.
        - Η συνάρτηση καλείται αναδρομικά για το αριστερό υποδέντρο του T, ενημερώνοντας το T1 και το αριστερό παιδί του T2.
        - Ενημερώνεται το ύψος του κόμβου T2.
    - Εάν το κλειδί k είναι μεγαλύτερο ή ίσο με το κλειδί του τρέχοντος κόμβου T:
        - Το δέντρο T1 ορίζεται ως το τρέχον δέντρο T.
        - Η συνάρτηση καλείται αναδρομικά για το δεξί υποδέντρο του T, ενημερώνοντας το T1 και το δεξί παιδί του T2.
        - Ενημερώνεται το ύψος του κόμβου T1.
3. **Εξισορρόπηση**:
    - Εξισορροπούνται τα υποδέντρα T1 και T2 καλώντας τη συνάρτηση balance για το καθένα, ώστε να διατηρηθούν οι ιδιότητες ισορροπίας του AVL δέντρου.

void find(Node* T, int k1, int k2, vector<int>& result): η συνάρτηση find διασχίζει το δέντρο T και προσθέτει όλα τα κλειδιά που βρίσκονται μέσα στο εύρος [k1, k2] στη λίστα result.

```
// Εύρεση όλων των κλειδιών σε ένα εύρος [k1, k2]
void find(Node* T, int k1, int k2, vector<int>& result) {
    if (!T) return;
    if (k1 < T->key) find(T->left, k1, k2, result);
    if (k1 <= T->key && k2 >= T->key) result.push_back(T->key);
    if (k2 > T->key) find(T->right, k1, k2, result);
}
```

1. **Αρχικός έλεγχος**:

    Ελέγχει αν το δέντρο T είναι κενό. Αν ναι, η συνάρτηση επιστρέφει χωρίς να κάνει τίποτα.

2. **Διάσχιση του αριστερού υποδέντρου**:

   Αν το κλειδί k1 είναι μικρότερο από το κλειδί του τρέχοντος κόμβου T->key, τότε αναδρομικά καλεί τη συνάρτηση find για το αριστερό υποδέντρο του T. Αυτό σημαίνει ότι μπορεί να υπάρχουν κλειδιά στο εύρος [k1, k2] στο αριστερό υποδέντρο.

3. **Προσθήκη του κλειδιού στη λίστα αποτελεσμάτων**:

   Αν το κλειδί του τρέχοντος κόμβου T->key βρίσκεται μέσα στο εύρος [k1, k2], τότε το προσθέτει στη λίστα result.

4. **Διάσχιση του δεξιού υποδέντρου**:

   Αν το κλειδί k2 είναι μεγαλύτερο από το κλειδί του τρέχοντος κόμβου T->key, τότε αναδρομικά καλεί τη συνάρτηση find για το δεξί υποδέντρο του T. Αυτό σημαίνει ότι μπορεί να υπάρχουν κλειδιά στο εύρος [k1, k2] στο δεξί υποδέντρο.

void findPredecessor(Node* T, int k, Node*& predecessor:

```
// Εύρεση προκατόχου ενός κλειδιού
void findPredecessor(Node* T, int k, Node*& predecessor) {
    // Αν το δέντρο είναι κενό, επιστροφή
    if (!T) return;

    // Αν το κλειδί του τρέχοντος κόμβου είναι μικρότερο από το κλειδί k,
    // ο τρέχων κόμβος μπορεί να είναι ο πρόγονος του κλειδιού k
    if (T->key < k) {
        // Αποθηκεύουμε τον τρέχοντα κόμβο ως πιθανό πρόγονο
        predecessor = T;
        // Συνεχίζουμε την αναζήτηση στο δεξί υποδέντρο
        findPredecessor(T->right, k, predecessor);
    }
    else {
        // Αν το κλειδί του τρέχοντος κόμβου δεν είναι μικρότερο από το κλειδί k,
        // συνεχίζουμε την αναζήτηση στο αριστερό υποδέντρο
        findPredecessor(T->left, k, predecessor);
    }
}
```

1. **Έλεγχος κενού δέντρου**:

   Αν το δέντρο T είναι κενό (nullptr), η συνάρτηση επιστρέφει χωρίς να κάνει τίποτα. Αυτό σημαίνει ότι δεν υπάρχει κόμβος για να εξεταστεί περαιτέρω.

2. **Σύγκριση κλειδιού τρέχοντος κόμβου με το κλειδί k**:

   Αν το κλειδί του τρέχοντος κόμβου T->key είναι μικρότερο από το κλειδί k, τότε:

   - Αποθηκεύεται ο τρέχων κόμβος T ως προσωρινός προκατόχους (predecessor).
   - Συνεχίζει την αναζήτηση για τον προκατόχου στο δεξί υποδέντρο του τρέχοντος κόμβου καλώντας αναδρομικά την ίδια συνάρτηση findPredecessor για το δεξί υποδέντρο T->right.

3. **Αλλιώς**:

   Αν το κλειδί του τρέχοντος κόμβου T->key δεν είναι μικρότερο από το κλειδί k, σημαίνει ότι ο προκατόχους του κλειδιού k θα είναι κάπου στο αριστερό υποδέντρο.

   - Συνεχίζει την αναζήτηση για τον προκατόχου στο αριστερό υποδέντρο του τρέχοντος κόμβου καλώντας αναδρομικά την ίδια συνάρτηση findPredecessor για το αριστερό υποδέντρο T->left.

void findSuccessor(Node* T, int k, Node*& successor): συνάρτηση χρησιμοποιείται για την εύρεση του διαδόχου ενός κλειδιού k σε ένα δυαδικό αναζητούμενο δέντρο (BST)

```
// Εύρεση διαδόχου ενός κλειδιού
void findSuccessor(Node* T, int k, Node*& successor) {
    if (!T) return;
    if (T->key > k) {
        successor = T;
        findSuccessor(T->left, k, successor);
    }
    else {
        findSuccessor(T->right, k, successor);
    }
}
```

1. **Έλεγχος κενού δέντρου**:

   - Αν το δέντρο T είναι κενό (nullptr), η συνάρτηση επιστρέφει χωρίς να κάνει τίποτα. Αυτό σημαίνει ότι δεν υπάρχει κόμβος για να εξεταστεί περαιτέρω.

2. **Σύγκριση κλειδιού τρέχοντος κόμβου με το κλειδί k**:

   - Αν το κλειδί του τρέχοντος κόμβου T->key είναι μεγαλύτερο από το κλειδί k, τότε:

     - Αποθηκεύεται ο τρέχων κόμβος T ως προσωρινός διάδοχος (successor), γιατί μπορεί να είναι ο πιθανός διάδοχος, αλλά θα χρειαστεί να ελεγχθούν και άλλοι κόμβοι για τον πραγματικό διάδοχο.

     - Συνεχίζει την αναζήτηση για τον διάδοχο στο αριστερό υποδέντρο του τρέχοντος κόμβου καλώντας αναδρομικά την ίδια συνάρτηση findSuccessor για το αριστερό υποδέντρο T->left.

3. **Αλλιώς**:

   - Αν το κλειδί του τρέχοντος κόμβου T->key δεν είναι μεγαλύτερο από το κλειδί k, σημαίνει ότι ο διάδοχος του κλειδιού k θα είναι κάπου στο δεξί υποδέντρο.

- Συνεχίζει την αναζήτηση για τον διάδοχο στο δεξί υποδέντρο του τρέχοντος κόμβου καλώντας αναδρομικά την ίδια συνάρτηση findSuccessor για το δεξί υποδέντρο T->right.

Κατά την εκτέλεση της συνάρτησης, η μεταβλητή successor που περνάει ως αναφορά αλλάζει τιμή καθώς η αναζήτηση προχωράει στο δέντρο μέχρι να βρει τον επόμενο κατά σειρά κόμβο μεγαλύτερο από το κλειδί k.

Node* balance(Node* node):

```
// Ισορροπία ενός υποδέντρου βάσει του συντελεστή ισορροπίας
Node* balance(Node* node) {
    if (!node) return node; // Αν ο κόμβος είναι κενός, επιστρέφουμε τον κόμβο

    // Ενημέρωση του ύψους του κόμβου
    node->height = 1 + max(height(node->left), height(node->right));

    // Υπολογισμός του συντελεστή ισορροπίας για τον κόμβο
    int balance = balanceFactor(node);

    // LL περίπτωση (ανισορροπία αριστερά-αριστερά)
    if (balance > 1 && balanceFactor(node->left) >= 0) {
        return rotateRight(node);
    }

    // LR περίπτωση (ανισορροπία αριστερά-δεξιά)
    if (balance > 1 && balanceFactor(node->left) < 0) {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }

    // RR περίπτωση (ανισορροπία δεξιά-δεξιά)
    if (balance < -1 && balanceFactor(node->right) <= 0) {
        return rotateLeft(node);
    }

    // RL περίπτωση (ανισορροπία δεξιά-αριστερά)
    if (balance < -1 && balanceFactor(node->right) > 0) {
        node->right = rotateRight(node->right);
        return rotateLeft(node);
    }

    // Επιστροφή του κόμβου αν δεν απαιτείται περιστροφή
    return node;
}
```

Η συνάρτηση `balance` εξισορροπεί ένα υποδέντρο βάσει του συντελεστή ισορροπίας του, πραγματοποιώντας τις απαραίτητες περιστροφές (δεξιά ή αριστερά) για να διατηρήσει την ισορροπία ενός AVL δέντρου. Αρχικά, ενημερώνει το ύψος του κόμβου και υπολογίζει τον συντελεστή ισορροπίας. Αν ο συντελεστής ισορροπίας δείχνει ότι το δέντρο είναι ανισόρροπο (με διαφορά ύψους μεγαλύτερη από 1 ή μικρότερη από -1), η συνάρτηση εφαρμόζει τη σωστή περιστροφή βάσει της περίπτωσης ανισορροπίας: LL (αριστερά-αριστερά), LR (αριστερά-δεξιά), RR (δεξιά-δεξιά) ή RL (δεξιά-αριστερά). Στο τέλος, επιστρέφει τον ισορροπημένο κόμβο.

int main():

```
int main() {
    AVLTree tree;

    // Εισαγωγή στοιχείων στο AVL δέντρο
    tree.insert(10);
    tree.insert(20);
    tree.insert(30);
    tree.insert(40);
    tree.insert(50);
    tree.insert(25);

    // Αναζήτηση στοιχείων στο AVL δέντρο
    cout << "Search 100: " << (tree.search(100) ? "Found" : "Not Found") << endl;
    cout << "Search 50: " << (tree.search(50) ? "Found" : "Not Found") << endl;

    // Αφαίρεση στοιχείου από το AVL δέντρο
    tree.remove(20);
    cout << "Search 20 after removal: " << (tree.search(20) ? "Found" : "Not Found") << endl;

    // Δημιουργία δύο AVL δέντρων για τη λειτουργία join
    AVLTree tree1;
    tree1.insert(1);
    tree1.insert(2);
    tree1.insert(3);

    AVLTree tree2;
    tree2.insert(5);
    tree2.insert(6);
    tree2.insert(7);

    // Λειτουργία join
    AVLTree joinedTree;
    joinedTree.join(tree1, 4, tree2);
    cout << "Joined Tree contains 4: " << (joinedTree.search(4) ? "Found" : "Not Found") << endl;

    // Λειτουργία split
    AVLTree splitTree1, splitTree2;
    joinedTree.split(4, splitTree1, splitTree2);
    cout << "Split Tree1 contains 3: " << (splitTree1.search(3) ? "Found" : "Not Found") << endl;
    cout << "Split Tree2 contains 5: " << (splitTree2.search(5) ? "Found" : "Not Found") << endl;

    // Λειτουργία find
    vector<int> foundElements = joinedTree.find(3, 6);
    cout << "Elements in range [3, 6]: ";
```

```
    for (int elem : foundElements) {
        cout << elem << " ";
    }
    cout << endl;

    // Λειτουργία find_neighbor
    int predecessor = joinedTree.findNeighbor(4, "predecessor");
    int successor = joinedTree.findNeighbor(4, "successor");

    // Χρήση ostringstream για μετατροπή αριθμών σε συμβολοσειρές
    ostringstream oss;
    oss << predecessor;
    string predecessorStr = predecessor != -1 ? oss.str() : "None";
    oss.str(""); // Καθαρισμός του buffer
    oss.clear(); // Επαναφορά της κατάστασης του stream
    oss << successor;
    string successorStr = successor != -1 ? oss.str() : "None";

    cout << "Predecessor of 4: " << predecessorStr << endl;
    cout << "Successor of 4: " << successorStr << endl;

    return 0;
}
```

Αυτός ο κώδικας παρουσιάζει τη χρήση μιας δομής δεδομένων AVL δέντρου, επιδεικνύοντας διάφορες λειτουργίες του. Πρώτα, δημιουργείται ένα AVL δέντρο και εισάγονται σε αυτό τα

στοιχεία 10, 20, 30, 40, 50, και 25. Στη συνέχεια, εκτελείται αναζήτηση για τα στοιχεία 100 και 50, ακολουθούμενη από την αφαίρεση του στοιχείου 20 και την επανεξέταση της ύπαρξής του στο δέντρο. Ακολούθως, δημιουργούνται δύο νέα AVL δέντρα, τα οποία εισάγουν στοιχεία και στη συνέχεια ενώνονται σε ένα νέο δέντρο με τη χρήση της λειτουργίας join. Το νέο δέντρο διασπάται σε δύο υποδέντρα βάσει του στοιχείου 4. Επιπλέον, εκτελείται εύρεση στοιχείων σε εύρος [3, 6] και εκτυπώνονται τα αποτελέσματα. Τέλος, βρίσκονται και εκτυπώνονται ο προκάτοχος και ο διάδοχος του στοιχείου 4 χρησιμοποιώντας τη λειτουργία findNeighbor.