პროგრამირების აბსტრაქციები
სემინარის ამოცანები #25

1 ხის დაბალანსება შემთხვევითი რიცხვებით

ორობითი ძებნის ხის ერთ-ერთი მარტივი ბალანსირების სტრატეგია არის თითოეული ჩასმული წვეროსთვის შემთხვევითი პრიორიტეტის მინიჭება და წვეროების სიღრმის პრიორიტეტის მიხედვით განსაზღვრა. კერძოდ, პრიორიტეტების მიხედვით ჩვენი სტრუქტურა "heap"-ისავით უნდა იყოს დალაგებული, ანუ ყოველი წვეროს ორივე შვილს უფრო მაღალი (ან ტოლი) რიცხვით გამოსახული პრიორიტეტი უნდა ჰქონდეს ვიდრე მოცემულ წვეროს.

იმისთვის რომ ეს წესი შევინარჩუნოთ ახალი წვეროს ჩასმისას, ჯერ ჩვეულებრივათ უნდა ჩავსვათ ეს წვერო ხეში და შემდეგ, სანამ მისი პრიორიტეტი ნაკლები იქნება მშობლის პრიორიტეტზე, ხე ისე უნდა დავატრიალოთ, რომ ეს წვერო მის მშობელზე უფრო მაღლა მოექცეს.

შეცვალეთ ადრე დაწერილი ხის კლასი ისე, რომ ის დაბალანსების ოპერაციებს მოიცავდეს. ხიდან წვეროს ამოღების ოპერაციის გაკეთება არ არის საჭირო.

interface

```
#ifndef BST_H_
#define BST_H_

class bst {
private:
    struct Node {
        int val;
        int priority;
        Node* left;
        Node* right;
    };
    Node* root;
    void deleteSubtree(Node* subtree);
    Node* findNode(Node* subtree, int value);
    void rotateLeft(Node*& subtree);
    void rotateRight(Node*& subtree);
    void insertInSubtree(Node*& subtree, Node* newNode);
public:
    bst();
    ~bst();
    bool contains(int value);
    void insert(int newVal);
};

#endif BST_H_
```

implementation (ჩვეულებრივი ორობითი ძებნის ხის მეთოდები)

```
#include <cstdlib>
#include "bst.h"

bst::bst() {
    root = NULL;
}

void bst::deleteSubtree(Node* subtree) {
    if (subtree != NULL) {
        deleteSubtree(subtree -> left);
        deleteSubtree(subtree -> right);
        delete subtree;
    }
}

bst::~bst() {
    deleteSubtree(root);
}

bst::Node* bst::findNode(Node* subtree, int value) {
    if (subtree == NULL || subtree -> val == value)
        return subtree;
    if (subtree -> val >= value)
        return findNode(subtree -> left, value);
    else
        return findNode(subtree -> right, value);
}

bool bst::contains(int value) {
    return (findNode(root, value) != NULL);
}
```

implementation (ხის დატრიალება და ელემენტის ჩასმა)

```
#include "random.h"
#include "limits.h"

void bst::rotateLeft(Node*& subtree) {
    Node* beta = subtree -> right -> left;
    subtree -> right -> left = subtree;
    subtree = subtree -> right;
    subtree -> left -> right = beta;
}

void bst::rotateRight(Node*& subtree) {
    Node* beta = subtree -> left -> right;
    subtree -> left -> right = subtree;
    subtree = subtree -> left;
    subtree -> right -> left = beta;
}

void bst::insertInSubtree(Node*& subtree,
                          Node* newNode) {
    if (subtree == NULL) {
        subtree = newNode;
        return;
    }

    if (newNode -> val < subtree -> val) {
        insertInSubtree(subtree -> left, newNode);
        if (subtree -> left -> priority < subtree -> priority)
            rotateRight(subtree);
    } else if (newNode -> val > subtree -> val) {
        insertInSubtree(subtree -> right, newNode);
        if (subtree -> right -> priority < subtree -> priority)
            rotateLeft(subtree);
    } else {
        delete newNode;
    }
}

void bst::insert(int newVal) {
    Node* newNode = new Node;
    newNode -> val = newVal;
    newNode -> priority = randomInteger(0, INT_MAX);
    newNode -> left = NULL;
    newNode -> right = NULL;
    insertInSubtree(root, newNode);
}
```