**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**ОТЧЕТ**

**по лабораторной работе № 3**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: «КОМБИНИРОВАННЫЕ СТРУКТУРЫ ДАННЫХ**

**И СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 3311 | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Сапронов К. Д.  Землякова С. А. |
|  | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Преподаватель |  | Манирагена Валенс |

Санкт-Петербург

2025

**Текст индивидуального задания (вариант 40)**

Реализовать индивидуальное задание темы «Множества + последовательности» в виде программы, используя контейнер 1-2 дерева. Для операций с контейнером рекомендуется использовать возможности библиотеки алгоритмов. Программа должна реализовывать цепочку операций над множествами: (A \ B) ∪ C ⊕ D ∩ E, и операциями MERGE, CONCAT, ERASE с последовательностью. Результат каждого шага цепочки операций выводится на экран.

Реализация каждой операции должна обеспечивать расширенные гарантии устойчивости к исключениям.

**Описание контейнера**

Структура данных:  
1-2 дерево — это разновидность сбалансированного дерева поиска, где каждый узел может содержать 1 или 2 ключа и соответственно 2 или 3 поддерева.

Ключевые особенности:

* Балансировка: Автоматически поддерживается при вставке за счёт разделения узлов с 3 ключами на три узла с 1 ключом.
* Итераторы: Реализован однонаправленный итератор (myiter) для совместимости с STL-алгоритмами.
* Методы:
  + insert(k) – вставка элемента с гарантией балансировки
  + erase(k) – удаление элемента с сохранением структуры дерева
  + begin(), end() – итераторы для обхода в порядке возрастания.
* Визуализация: Метод Display() выводит дерево в псевдографическом виде с учётом вложенности.

Совместимость с STL:  
Контейнер поддерживает стандартные алгоритмы (set\_union, set\_intersection и др.) благодаря:

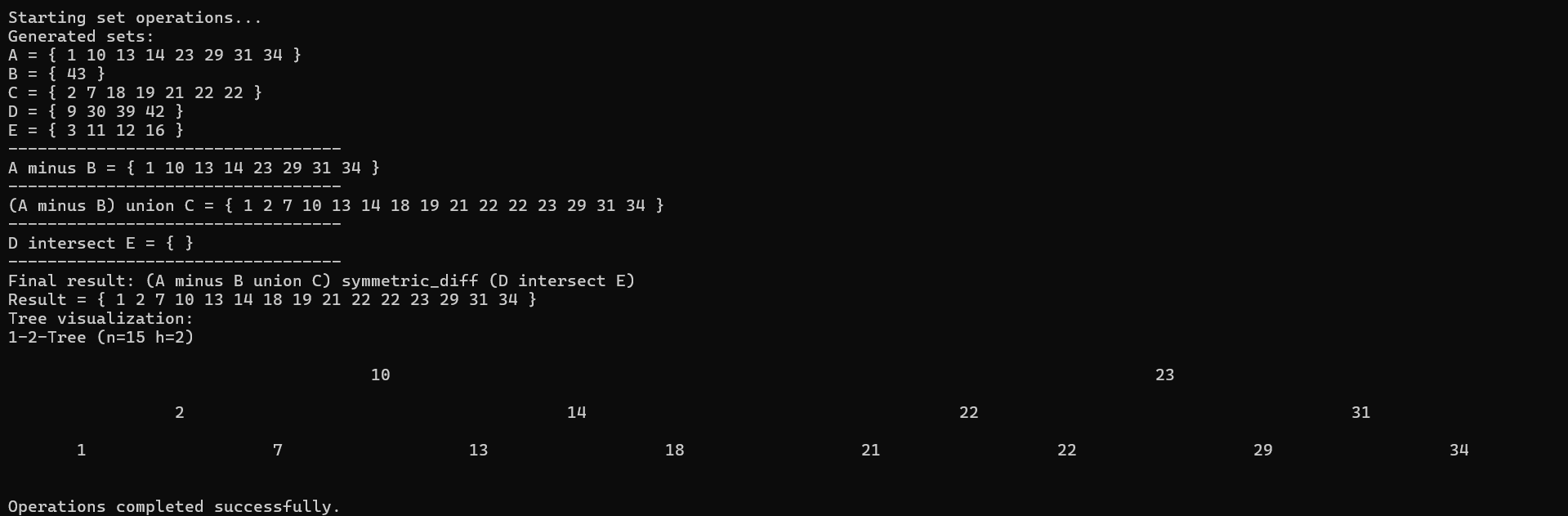
* Итераторам с категорией forward\_iterator\_tag.
* Методам insert и erase, соответствующим STL-конвенциям.

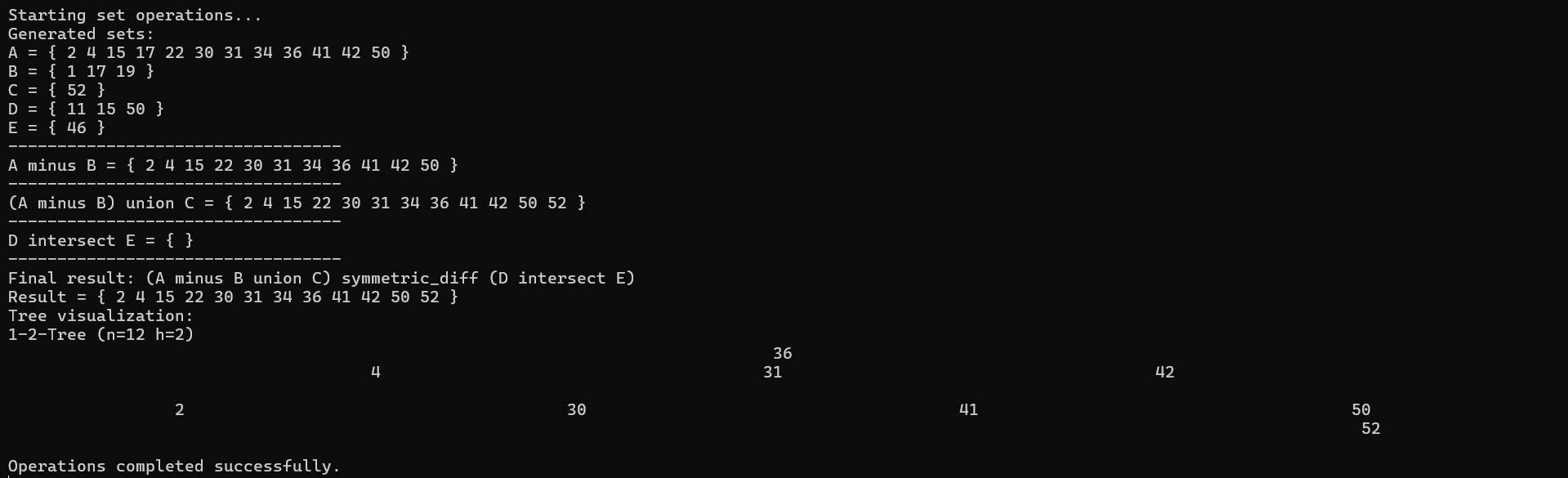
Гарантии исключений:

* Базовая: При исключении структура дерева остаётся корректной.
* Расширенная: Вставка либо завершается успешно, либо не изменяет контейнер

**Примеры выполнения программы:**

1. Множества

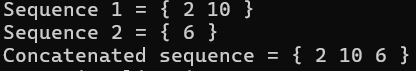




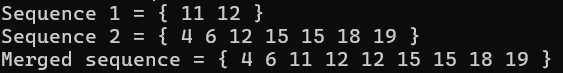
1. Последовательности

Для последовательностей вывод дерева некорректен, а для concat невозможен, так как дерево автоматически сортируется и балансируется

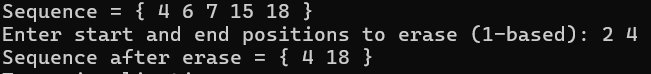
Merge



Concat



Erase



**Выводы:**

Оценка временной сложности:

* 1. Вставка/удаление в 1-2 дереве: O(log n).
  2. Разность/объединение/пересечение: O(n + m) (линейная от размера множеств).
  3. Общая сложность цепочки: O(n log n)

**Контрольные вопросы**

**Ответы на 3.3.1.**

1. Почему для хранения произвольной последовательности структуру данных для множества (хеш-таблицу или ДДП) приходится дорабатывать?    
Множества хранят элементы без учета порядка (в ДДП — сортировка по ключу, в хеш-таблице — произвольный порядок). Для работы с последовательностями нужна фиксация порядка элементов, поэтому требуется дополнительный механизм хранения позиций (например, список указателей или массив индексов).  
  
---  
  
2. Какие доработки возможны?    
- ДДП:    
  - Добавление поля order в узлы для хранения позиции в последовательности.   
  - Поддержка двусвязного списка итераторов для обхода в порядке вставки.   
- Хеш-таблица:    
  - Отдельный массив или список, хранящий элементы в порядке добавления.   
  - Использование unordered\_map + vector (ключи в хеше, порядок — в векторе).   
  
---  
  
3. Можно ли предложить оптимальный вариант доработки?    
Да. Для ДДП:   
- Хранить в узле поле sequence\_pos (позиция в последовательности).   
- Поддерживать двусвязный список итераторов для быстрого доступа к элементам в порядке вставки.   
Для хеш-таблицы:   
- Использовать std::unordered\_map + std::vector (хеш — для поиска, вектор — для порядка).   
  
---  
  
4. Влияет ли доработка структур данных для множеств для поддержки последовательностей на временную сложность операций над множествами?    
- ДДП:    
  - Вставка/удаление остаются O(log n), но обновление списка итераторов требует O(1).    
- Хеш-таблица:    
  - Поиск остается O(1), но вставка в вектор — O(n) в худшем случае (если нужен сдвиг элементов).   
- Вывод: Влияет, но не критично, если доработка сделана аккуратно.   
  
---  
  
5. Какую структуру данных проще дорабатывать — хеш-таблицу или ДДП?    
- Хеш-таблица проще:   
  - Достаточно добавить вектор для порядка.   
  - Не требует балансировки, как ДДП.   
- ДДП сложнее:   
  - Нужно поддерживать порядок при поворотах дерева.   
  
---  
  
6. Какова оптимальная доработка структуры данных и временная сложность для операции исключения части последовательности между указанными позициями?    
- Доработка: Использовать vector или двусвязный список для хранения порядка.   
- Сложность `ERASE(p1, p2)`:    
  - Для вектора: O(n) (сдвиг элементов).   
  - Для списка: O(1) (если известны итераторы на p1 и `p2`).   
  
---  
  
7. То же — для операции вставки с указанной позиции.    
- Доработка:    
  - Вектор: O(n) (сдвиг).   
  - Список: O(1) (вставка между узлами).   
- Оптимально: Двусвязный список + хеш-таблица для быстрого поиска.   
  
---  
  
8. То же — для замены.    
- Доработка:    
  - Вектор: O(1) (доступ по индексу).   
  - Список: O(1) (если есть итератор).   
- Сложность `CHANGE(p, new\_seq)`:    
  - Зависит от длины new\_seq. В среднем O(k), где k — размер заменяемого фрагмента.

**Ответы на контрольные вопросы (3.2.1)**  
1. Каким способом следует разметить дерево, чтобы в нём был возможен двоичный поиск? Как его следует нагрузить?    
- Разметка:    
  - Каждый узел содержит ключ и два поддерева (левое и правое).   
  - Для всех узлов выполняется условие:   
    - Левые поддеревья содержат ключи меньшие текущего.   
    - Правые поддеревья содержат ключи не меньшие текущего.   
- Нагрузка:    
  - Ключи должны быть сравнимыми (например, числа, строки).   
  - Дерево должно быть сбалансированным (иначе вырождается в список).   
  
---  
  
2. Почему для операций с двоичным деревом дают две оценки сложности — «в худшем случае» и «в среднем»? Почему не рассматривается «лучший» случай?    
- Худший случай: Дерево вырождается в линейный список → сложность O(n).    
- Средний случай: Дерево сбалансировано → сложность O(log n).    
- Лучший случай не рассматривается, так как он тривиален (например, поиск корня за `O(1)`), но не отражает реальную производительность.   
  
---  
  
3. Отчего деревья двоичного поиска вырождаются?    
- При последовательной вставке отсортированных данных (например, `1, 2, 3, 4`).   
- В результате дерево превращается в односторонний список с высотой n.    
  
---  
  
4. Можно ли воспрепятствовать вырождению ДДП?    
Да. Использовать:   
- Автобалансирующиеся деревья (AVL, красно-черные, 1-2 деревья).   
- Рандомизированную вставку (например, Treap).   
- Периодическую перестройку (например, при достижении глубины `2 log n`).   
  
---  
  
5. Каков оптимальный алгоритм двуместной операции над множествами, представленными деревьями двоичного поиска?    
1. Обход деревьев в порядке возрастания (через рекурсию или стек).   
2. Слияние двух отсортированных последовательностей (аналогично merge в сортировке слиянием).   
3. Построение нового ДДП из результата.   
- Сложность: O(n + m) (линейная от размеров деревьев).   
  
---  
  
6. Какова временная сложность такой операции и как сказывается на ней возможное вырождение деревьев?    
- Сбалансированные деревья: O(n + m).    
- Вырожденные деревья:    
  - Обход требует O(n²) (из-за рекурсии).   
  - Слияние остается O(n + m), но построение нового дерева может ухудшиться до O(n²).    
  
---  
  
7. Может ли при двуместной операции над множествами в ДДП получиться вырожденное дерево-результат?    
Да, если:   
- Исходные деревья были вырождены.   
- Новое дерево строится последовательной вставкой (без балансировки).   
  
---  
  
8. Можно ли хранить в дереве двоичного поиска множество с повторениями?    
Да. Варианты:   
1. Разрешить дубликаты в правом поддереве (`>=` вместо `>`).   
2. Хранить счетчик повторов в узле.   
3. Использовать `multiset` (стандартный контейнер C++).   
  
---  
  
9. Какая структура данных требует больше памяти для хранения множества: хеш-таблица или дерево двоичного поиска?    
- Хеш-таблица:    
  - Дополнительная память на разрешение коллизий (цепочки или открытая адресация).   
- ДДП:    
  - Хранит указатели на потомков (`2n` указателей для n элементов).   
- Вывод: ДДП обычно требует больше памяти.   
  
---  
  
10. Какая из них быстрее работает?    
- Хеш-таблица:    
  - Вставка/поиск/удаление — O(1) в среднем.   
- ДДП:    
  - Те же операции — O(log n) в сбалансированном случае.   
- Итог: Хеш-таблица быстрее, но ДДП сохраняет порядок элементов.   
  
---  
  
11. Как сделать не вырождающееся ДДП? Зачем оно может понадобиться?    
- Способы:    
  - AVL-деревья, красно-черные деревья, B-деревья.   
  - Treap (дерево + куча).   
- Применение:    
  - Базы данных, файловые системы, std::map в C++.   
  
---  
  
12. Какая структура данных является оптимальной для хранения дерева двоичного поиска?    
- На практике:    
  - Стандартные контейнеры (`std::set`, std::map в C++).   
  - Самобалансирующиеся деревья (AVL, RB-tree).   
- Для учебных целей:    
  - Ручная реализация с указателями (если требуется контроль над балансировкой).

1. Что такое стандартный контейнер библиотеки *STL*? Чем он отличается от обычного объекта?

Стандартный контейнер STL — это шаблонный класс, предоставляющий структуру данных (вектор, список, множество) с унифицированным интерфейсом (итераторы, методы size(), insert() и т.д.). Отличается от обычного объекта поддержкой STL-алгоритмов и гарантиями исключений.

2. Какой стандартный контейнер можно считать наиболее подходящим для работы с множествами?

Наиболее подходящий контейнер для множеств — std::set (балансированное дерево) или std::unordered\_set (хеш-таблица), так как они обеспечивают уникальность элементов и быстрый поиск.

3. Можно ли использовать стандартные контейнеры для множеств, на которых не определено отношение полного порядка?

Множества без полного порядка можно хранить в std::unordered\_set, где порядок не требуется (используется хеш-функция).

4. Существуют ли ограничения на применение стандартных алгоритмов двуместных операций над множествами в контейнерах?

Да, например, им требуются отсортированные диапазоны, и итераторы должны поддерживать сравнение (<).

5. Можно ли реализовать двуместную операцию над множествами в контейнерах без применения стандартного алгоритма?

Да, но с большим объёмом кода (например, вручную сравнивать элементы через итераторы).

6. Можно ли выполнять операции над последовательностями для множеств, хранящихся в стандартном контейнере?

Да, например, std::transform или std::accumulate для std::set.

7. Можно ли обеспечить поддержку произвольных последовательностей в контейнере для множеств?

Да, если контейнер реализует итераторы произвольного доступа (как вектор) или хотя бы однонаправленные.

8. Какова ожидаемая временная сложность при выполнении стандартным алгоритмом операции объединения двух множеств в стандартных контейнерах *set*?

O(n + m), так как оба множества уже отсортированы.

9. С какой целью может понадобиться оформить пользовательскую структуру данных как стандартный контейнер?

Совместимость с STL-алгоритмами и повторное использование кода.

10. Каков минимально необходимый набор средств для превращения пользовательской структуры данных в полноценный контейнер?

Итераторы и методы begin(), end(), size(), insert().

11. Зачем нужны гарантии устойчивости алгоритмов относительно исключений?

Предотвращение утечек ресурсов и сохранение данных при ошибках.

12. Почему базовых гарантий устойчивости алгоритма к исключениям может быть недостаточно?

Базовые гарантии сохраняют инварианты, но могут оставить программу в промежуточном состоянии (например, частично изменённые данные).

13. В чём заключаются расширенные гарантии устойчивости алгоритма к исключениям и как их можно обеспечить?

Либо операция завершается успешно, либо состояние остаётся неизменным. Такой подход обеспечивается через откат изменений при исключении, или внесение изменений только после положительной проверки на отсутствие исключений.

Код программы:

// 1-2-Tree.h

#ifndef TREE\_1\_2\_H

#define TREE\_1\_2\_H

#include <iostream>

#include <string>

#include <stack>

#include <utility>

#include <algorithm>

#include <vector>

#include <random>

// Node structure for 1-2 Tree

struct Node {

    int key;

    int count = 1;  // Number of duplicates

    Node \*left, \*right;

    bool next = false;

    Node(int k, Node \*l = nullptr, Node \*r = nullptr)

        : key(k), left(l), right(r) {}

    ~Node() { delete left; delete right; }

    void Display(int level, int col, int& max\_level) const;

    void Out(int row, int col) const;

};

using MyStack = std::stack<std::pair<Node\*, int>>;

// Iterator for Tree class

struct myiter {

    using iterator\_category = std::forward\_iterator\_tag;

    using value\_type = int;

    using difference\_type = std::ptrdiff\_t;

    using pointer = int\*;

    using reference = int&;

    myiter(Node \*p = nullptr) : Ptr(p), current\_count(0) {

        if (p) current\_count = 1;

    }

    myiter(Node \*p, MyStack St) : Ptr(p), St(St), current\_count(0) {

        if (p) current\_count = 1;

    }

    bool operator==(const myiter &Other) const { return Ptr == Other.Ptr; }

    bool operator!=(const myiter &Other) const { return !(\*this == Other); }

    myiter& operator++();

    myiter operator++(int) { myiter tmp(\*this); ++\*this; return tmp; }

    reference operator\*() const { return Ptr->key; }

    pointer operator->() const { return &Ptr->key; }

    Node \*Ptr;

    MyStack St;

    int current\_count = 1;

};

// 1-2 Tree class implementation

class Tree {

    Node \*root = nullptr;

    int count = 0;

    int tree\_height = 0;

    bool is\_set = true; // Flag to distinguish between set and sequence

public:

    using value\_type = int;

    Tree(bool is\_set = true) : is\_set(is\_set) {}

    ~Tree() { delete root; }

    std::pair<myiter, bool> insert(int k, myiter where = myiter());

    bool erase(int k);

    bool erase\_all(int k);

    void clear() { delete root; root = nullptr; count = 0; tree\_height = 0; }

    int size() const { return count; }

    int height() const { return tree\_height; }

    myiter begin() const;

    myiter end() const { return myiter(); }

    void Display() const;

    template<typename Iter>

    Tree(Iter first, Iter last, bool is\_set = true) : is\_set(is\_set) {

        for (; first != last; ++first) insert(\*first);

    }

    void set\_mode(bool set\_mode) { is\_set = set\_mode; }

};

// Output iterator adapter for set operations

template <typename Container>

class outiter {

    Container& container;

    typename Container::value\_type value;

    myiter iter;

public:

    using iterator\_category = std::output\_iterator\_tag;

    explicit outiter(Container& c, myiter it) : container(c), iter(it) {}

    outiter& operator\*() { return \*this; }

    outiter& operator++() { return \*this; }

    outiter& operator++(int) { return \*this; }

    outiter& operator=(const typename Container::value\_type& val) {

        value = val;

        iter = container.insert(value, iter).first;

        return \*this;

    }

};

template <typename Container>

inline outiter<Container> outinserter(Container& c, myiter it) {

    return outiter<Container>(c, it);

}

int random\_int(int min, int max) {

    static std::random\_device rd;

    static std::mt19937 gen(rd());

    std::uniform\_int\_distribution<> distrib(min, max);

    return distrib(gen);

}

// Function declarations

void execute\_set\_operations();

void execute\_sequence\_operations();

template<typename Iter>

void printSet(const std::string& name, Iter begin, Iter end);

#endif

// 1-2-Tree.cpp

#include "1-2-Tree.h"

#include <vector>

#include <algorithm>

#include <iterator>

#include <iostream>

// Constants for tree display

const int FIRSTCOL = 80;

const int MAXCOL = 160;

const int OFFSET[] = {40, 20, 10, 5, 2, 1};

const int MAXROW = 30;

const int SHIFT = 3;

std::string SCREEN[MAXROW];

// Display methods for Node with bounds checking

void Node::Out(int row, int col) const {

    if (row >= MAXROW || col < 0 || col >= MAXCOL) return;

    std::string keyStr = std::to\_string(key);

    if (count > 1) {

        keyStr += "(" + std::to\_string(count) + ")";

    }

    // Ensure we don't write past screen boundaries

    int len = std::min(static\_cast<int>(keyStr.length()), MAXCOL - col);

    if (len > 0) {

        SCREEN[row].replace(col, len, keyStr.substr(0, len));

    }

}

// Simplified tree display method

void Node::Display(int level, int col, int& max\_level) const {

    if (level > max\_level) max\_level = level;

    // Calculate offset based on level with bounds checking

    int current\_offset = (level < 6) ? OFFSET[level] : OFFSET[5];

    // Display current node

    Out(level\*2, col);

    // Recursively display left and right subtrees

    if (left) left->Display(level+1, col - current\_offset, max\_level);

    if (right) right->Display(level+1, col + current\_offset, max\_level);

}

// Tree display method with proper initialization

void Tree::Display() const {

    // Clear the screen buffer

    std::fill\_n(SCREEN, MAXROW, std::string(MAXCOL, ' '));

    int max\_level = 0;

    if (root) {

        root->Display(0, FIRSTCOL, max\_level);

    } else {

        SCREEN[0].replace(FIRSTCOL, 9, "Empty Tree");

    }

    // Display only the relevant portion of the screen

    for (int i = 0; i <= max\_level\*2+2 && i < MAXROW; ++i) {

        std::string line = SCREEN[i];

        // Trim trailing spaces

        line.erase(line.find\_last\_not\_of(' ') + 1);

        std::cout << line << '\n';

    }

}

// Iterator increment operator

myiter& myiter::operator++() {

    if (!Ptr) return \*this;

    if (current\_count < Ptr->count) {

        current\_count++;

        return \*this;

    }

    if (Ptr->right) {

        St.push({Ptr, 1});

        Ptr = Ptr->right;

        while (Ptr->left) {

            St.push({Ptr, 0});

            Ptr = Ptr->left;

        }

    } else {

        auto pp = std::make\_pair(Ptr, 1);

        while (!St.empty() && pp.second) {

            pp = St.top();

            St.pop();

        }

        Ptr = pp.second ? nullptr : pp.first;

    }

    current\_count = 1;

    return \*this;

}

// Tree iterator methods

myiter Tree::begin() const {

    MyStack St;

    Node\* p = root;

    while (p && p->left) {

        St.push({p, 0});

        p = p->left;

    }

    return myiter(p, St);

}

// Tree insertion method

std::pair<myiter, bool> Tree::insert(int k, myiter where) {

    Node \*t = root;

    bool cont = true, up = false;

    MyStack St;

    int current\_depth = 0;

    if (!where.Ptr) {

        if (!t) {

            root = new Node(k);

            count = 1;

            tree\_height = 1;

            return {myiter(root), true};

        }

        St.push({root, 1});

        current\_depth = 1;

    } else {

        t = where.Ptr;

        St = std::move(where.St);

        current\_depth = St.size();

    }

    while (cont) {

        if (k == t->key) {

            if (is\_set) {

                return {myiter(t, std::move(St)), false};

            } else {

                t->count++;

                count++;

                return {myiter(t, std::move(St)), false};

            }

        }

        if (k < t->key) {

            if (t->left) {

                St.push({t, 2});

                t = t->left;

                current\_depth++;

            } else {

                t->left = new Node(k);

                cont = false;

                if (current\_depth + 1 > tree\_height) {

                    tree\_height = current\_depth + 1;

                }

            }

        } else {

            if (!t->right) {

                t->right = new Node(k);

                t->next = true;

                cont = false;

                if (current\_depth + 1 > tree\_height) {

                    tree\_height = current\_depth + 1;

                }

            } else if (t->next) {

                if (k < t->right->key) {

                    if (t->right->left) {

                        St.push({t, 3});

                        t = t->right->left;

                        current\_depth++;

                    } else {

                        t->right->left = new Node(k);

                        cont = false;

                        if (current\_depth + 2 > tree\_height) {

                            tree\_height = current\_depth + 2;

                        }

                    }

                } else {

                    if (t->right->right) {

                        St.push({t, 4});

                        t = t->right->right;

                        current\_depth++;

                    } else {

                        t->right->right = new Node(k);

                        up = t->right->next = true;

                        cont = false;

                        if (current\_depth + 2 > tree\_height) {

                            tree\_height = current\_depth + 2;

                        }

                    }

                }

            } else {

                if (t->right) {

                    St.push({t, 3});

                    t = t->right;

                    current\_depth++;

                } else {

                    t->right = new Node(k);

                    t->next = true;

                    cont = false;

                    if (current\_depth + 1 > tree\_height) {

                        tree\_height = current\_depth + 1;

                    }

                }

            }

        }

        while (up) {

            std::swap(t->key, t->right->key);

            std::swap(t->count, t->right->count);

            Node \*t1 = t->right;

            t->next = t1->next = false;

            t->right = t1->right;

            t1->right = t1->left;

            t1->left = t->left;

            t->left = t1;

            t1 = t;

            if (St.empty()) {

                up = false;

                tree\_height++;

            } else {

                t = St.top().first;

                switch (St.top().second) {

                    case 2:

                        std::swap(t->key, t1->key);

                        std::swap(t->count, t1->count);

                        t->left = t1->left;

                        t1->left = t1->right;

                        t1->right = t->right;

                        t->right = t1;

                        up = t1->next = t->next;

                        break;

                    case 3:

                        if (t->next) {

                            t->right->left = t1->right;

                            t1->right = t->right;

                            t->right = t1;

                            t1->next = true;

                        } else {

                            t->next = true;

                            up = t1->next = false;

                            St.pop();

                        }

                        break;

                    case 4:

                        t->right->next = true;

                        t1->next = false;

                        break;

                }

                if (up) St.pop();

            }

        }

    }

    ++count;

    return {myiter(t, std::move(St)), true};

}

// Tree erase method (removes one occurrence)

bool Tree::erase(int k) {

    Node \*p = nullptr, \*q = root;

    int a = 0;

    bool cont = (q != nullptr);

    while (cont && (k != q->key)) {

        p = q;

        a = (k > q->key);

        q = a ? q->right : q->left;

        cont = (q != nullptr);

    }

    if (!cont) return false;

    if (!is\_set || q->count > 1) {

        q->count--;

        count--;

        if (q->count == 0) {

            // Should never happen for sets

            Node \*r = q->right;

            if (r && r->left) {

                p = q;

                a = 1;

                do {

                    p = r;

                    r = r->left;

                    a = 0;

                } while (r->left);

                q->key = r->key;

                q->count = r->count;

                p->left = r->right;

                q = r;

            }

            else if (r) {

                (a ? r->right : r->left) = q->left;

                if (p) (a ? p->right : p->left) = r;

                else root = r;

            }

            else {

                if (p) {

                    if (a) {

                        p->right = q->left;

                        p->next = false;

                    }

                    else {

                        p->left = q->left;

                    }

                }

                else {

                    root = q->left;

                }

            }

            q->left = q->right = nullptr;

            delete q;

        }

        return true;

    }

    Node \*r = q->right;

    if (r && r->left) {

        p = q;

        a = 1;

        do {

            p = r;

            r = r->left;

            a = 0;

        } while (r->left);

        q->key = r->key;

        q->count = r->count;

        p->left = r->right;

        q = r;

    }

    else if (r) {

        (a ? r->right : r->left) = q->left;

        if (p) (a ? p->right : p->left) = r;

        else root = r;

    }

    else {

        if (p) {

            if (a) {

                p->right = q->left;

                p->next = false;

            }

            else {

                p->left = q->left;

            }

        }

        else {

            root = q->left;

        }

    }

    q->left = q->right = nullptr;

    --count;

    delete q;

    return true;

}

// New method to remove all occurrences

bool Tree::erase\_all(int k) {

    Node \*p = nullptr, \*q = root;

    int a = 0;

    bool cont = (q != nullptr);

    while (cont && (k != q->key)) {

        p = q;

        a = (k > q->key);

        q = a ? q->right : q->left;

        cont = (q != nullptr);

    }

    if (!cont) return false;

    int removed\_count = q->count;

    Node \*r = q->right;

    if (r && r->left) {

        p = q;

        a = 1;

        do {

            p = r;

            r = r->left;

            a = 0;

        } while (r->left);

        q->key = r->key;

        q->count = r->count;

        p->left = r->right;

        q = r;

    }

    else if (r) {

        (a ? r->right : r->left) = q->left;

        if (p) (a ? p->right : p->left) = r;

        else root = r;

    }

    else {

        if (p) {

            if (a) {

                p->right = q->left;

                p->next = false;

            }

            else {

                p->left = q->left;

            }

        }

        else {

            root = q->left;

        }

    }

    q->left = q->right = nullptr;

    count -= removed\_count;

    delete q;

    return true;

}

// Utility function to print a set

template<typename Iter>

void printSet(const std::string& name, Iter begin, Iter end) {

    std::cout << name << " = { ";

    for (; begin != end; ++begin) {

        std::cout << \*begin << " ";

    }

    std::cout << "}\n";

}

Tree generate\_random\_set(int max\_size) {

    Tree result(true); // Set mode

    int size = random\_int(1, max\_size);

    while (result.size() < size) {

        int num = random\_int(1, 52);

        result.insert(num);

    }

    return result;

}

Tree generate\_random\_sequence(int max\_size) {

    Tree result(false); // Sequence mode

    int size = random\_int(1, max\_size);

    for (int i = 0; i < size; ++i) {

        int num = random\_int(1, 20); // Smaller range to increase duplicates

        result.insert(num);

    }

    return result;

}

// Fixed sequence operations implementation

void execute\_sequence\_operations() {

    try {

        int choice = 0;

        std::cout << "\nSequence Operations:\n";

        std::cout << "1. Merge two sequences\n";

        std::cout << "2. Concatenate two sequences\n";

        std::cout << "3. Erase range from sequence\n";

        std::cout << "Enter your choice: ";

        std::cin >> choice;

        // Generate only one sequence for erase operation

        Tree seq1 = generate\_random\_sequence(10);

        if (choice == 3) {

            // Handle erase range operation

            std::cout << "\nGenerated sequence:\n";

            printSet("Sequence", seq1.begin(), seq1.end());

            int start, end;

            std::cout << "Enter start and end positions to erase (1-based): ";

            std::cin >> start >> end;

            if (start < 1 || end > seq1.size() || start > end) {

                std::cout << "Invalid range!\n";

                return;

            }

            // Convert to vector for easier range deletion

            std::vector<int> elements(seq1.begin(), seq1.end());

            elements.erase(elements.begin() + start - 1, elements.begin() + end);

            // Rebuild the tree

            seq1.clear();

            for (int val : elements) {

                seq1.insert(val);

            }

            printSet("Sequence after erase", seq1.begin(), seq1.end());

            std::cout << "Tree visualization:\n";

            seq1.Display();

        }

        else {

            // Handle merge and concat operations

            Tree seq2 = generate\_random\_sequence(8);

            std::cout << "\nGenerated sequences:\n";

            printSet("Sequence 1", seq1.begin(), seq1.end());

            printSet("Sequence 2", seq2.begin(), seq2.end());

            Tree result(false); // Sequence mode

            if (choice == 1) {

                // Merge operation

                std::vector<int> temp1(seq1.begin(), seq1.end());

                std::vector<int> temp2(seq2.begin(), seq2.end());

                std::sort(temp1.begin(), temp1.end());

                std::sort(temp2.begin(), temp2.end());

                std::merge(temp1.begin(), temp1.end(),

                         temp2.begin(), temp2.end(),

                         outinserter(result, result.begin()));

                printSet("Merged sequence", result.begin(), result.end());

            }

            else if (choice == 2) {

                // for (auto it = seq1.begin(); it != seq1.end(); ++it) {

                //     result.insert(\*it);

                // }

                // for (auto it = seq2.begin(); it != seq2.end(); ++it) {

                //     result.insert(\*it);

                // }

                std::vector<int> result;

                std::copy(seq1.begin(), seq1.end(), std::back\_inserter(result));

                std::copy(seq2.begin(), seq2.end(), std::back\_inserter(result));

                printSet("Concatenated sequence", result.begin(), result.end());

            }

            else {

                std::cout << "Invalid choice!\n";

                return;

            }

            std::cout << "Tree visualization:\n";

            result.Display();

        }

    }

    catch (const std::exception& e) {

        std::cerr << "Error in sequence operations: " << e.what() << std::endl;

    }

}

// Main set operations function

void execute\_set\_operations() {

    try {

        Tree A = generate\_random\_set(15);

        Tree B = generate\_random\_set(10);

        Tree C = generate\_random\_set(8);

        Tree D = generate\_random\_set(5);

        Tree E = generate\_random\_set(5);

        std::cout << "Generated sets:\n";

        printSet("A", A.begin(), A.end());

        printSet("B", B.begin(), B.end());

        printSet("C", C.begin(), C.end());

        printSet("D", D.begin(), D.end());

        printSet("E", E.begin(), E.end());

        std::cout << "----------------------------------\n";

        Tree A\_minus\_B(true);

        std::set\_difference(A.begin(), A.end(), B.begin(), B.end(),

                          outinserter(A\_minus\_B, A\_minus\_B.begin()));

        printSet("A minus B", A\_minus\_B.begin(), A\_minus\_B.end());

        std::cout << "----------------------------------\n";

        Tree A\_minus\_B\_union\_C(true);

        std::set\_union(A\_minus\_B.begin(), A\_minus\_B.end(),

                      C.begin(), C.end(),

                      outinserter(A\_minus\_B\_union\_C, A\_minus\_B\_union\_C.begin()));

        printSet("(A minus B) union C", A\_minus\_B\_union\_C.begin(), A\_minus\_B\_union\_C.end());

        std::cout << "----------------------------------\n";

        Tree D\_intersect\_E(true);

        std::set\_intersection(D.begin(), D.end(),

                             E.begin(), E.end(),

                             outinserter(D\_intersect\_E, D\_intersect\_E.begin()));

        printSet("D intersect E", D\_intersect\_E.begin(), D\_intersect\_E.end());

        std::cout << "----------------------------------\n";

        Tree result(true);

        std::set\_symmetric\_difference(A\_minus\_B\_union\_C.begin(), A\_minus\_B\_union\_C.end(),

                                     D\_intersect\_E.begin(), D\_intersect\_E.end(),

                                     outinserter(result, result.begin()));

        std::cout << "Final result: (A minus B union C) symmetric\_diff (D intersect E)\n";

        printSet("Result", result.begin(), result.end());

        std::cout << "Tree visualization:\n";

        result.Display();

    }

    catch (const std::exception& e) {

        std::cerr << "Error in set operations: " << e.what() << std::endl;

        throw;

    }

}

// Main function

int main() {

    try {

        int choice = 0;

        while (true) {

            std::cout << "\nMenu:\n";

            std::cout << "1. Set operations\n";

            std::cout << "2. Sequence operations\n";

            std::cout << "3. Exit\n";

            std::cout << "Enter your choice: ";

            std::cin >> choice;

            switch (choice) {

                case 1:

                    execute\_set\_operations();

                    break;

                case 2:

                    execute\_sequence\_operations();

                    break;

                case 3:

                    return 0;

                default:

                    std::cout << "Invalid choice. Please try again.\n";

            }

        }

    }

    catch (const std::exception& e) {

        std::cerr << "Fatal error: " << e.what() << std::endl;

        return 1;

    }

}