МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра Вычислительной техники

ОТЧЕТ

по лабораторной работе № 3

по дисциплине «Алгоритмы и структуры данных»

Тема: КОМБИНИРОВАННЫЕ СТРУКТУРЫ ДАННЫХ И СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ

Студентка гр. 2308 _____ Рымарь М.И.

Преподаватель _____ Манирагена В.

Санкт-Петербург
2023

Цель работы.

Изучение комбинированных структур данных и стандартной библиотеки шаблонов.

Задание.

Реализовать индивидуальное задание темы «Множества + последовательности» в виде программы, используя свой контейнер для заданной структуры данных (хеш-таблицы или одного из вариантов ДДП), и доработать его для поддержки операций с последовательностями. Для операций с контейнером рекомендуется использовать возможности библиотеки алгоритмов. Программа должна реализовывать цепочку операций над множествами, имеющимися в выражении, указанном на рисунке 1, с базовым контейнером и операциями с последовательностью на рисунке 2. Результат каждого шага цепочки операций выводится на экран.

Мощность множества	Что надо вычислить
26	$(A \oplus B \setminus C) \cup D \cap E$

Рисунок 1 – Выражение

Базо-вая СД	Дополнительные операции
2-3д	MERGE, ERASE, SUBST

Рисунок 2 – Структура данных и дополнительные операции

Описание работы программы.

За основу для написания пользовательского контейнера взято 2-3-дерево, в котором каждый управляющий узел хранит в себе наибольшие ключи из своих поддеревьев. Концепция 2-3 дерева схематично представлена на рисунке 3.

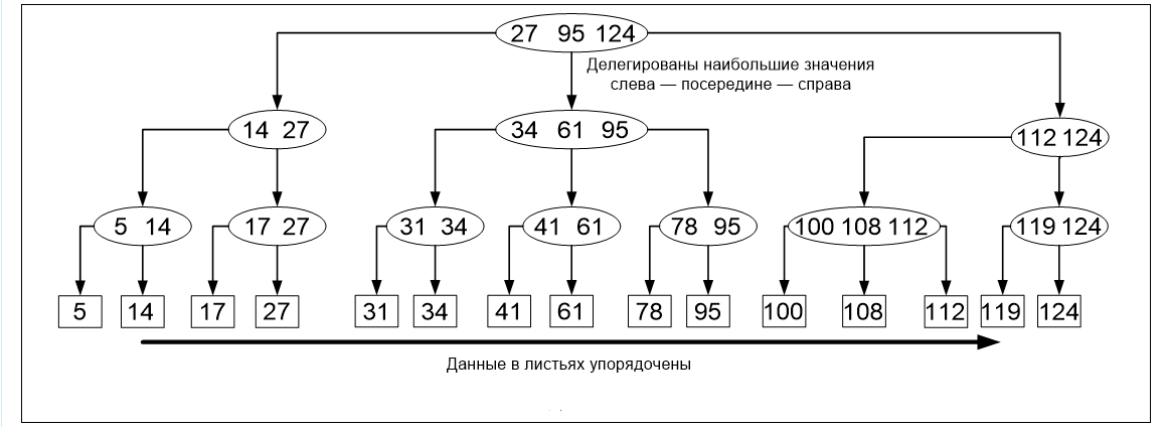

Рисунок 3 – Концепция 2-3 дерева

Пример вывода 2-3 дерева командой *D.display()* представлен на рисунке 4.

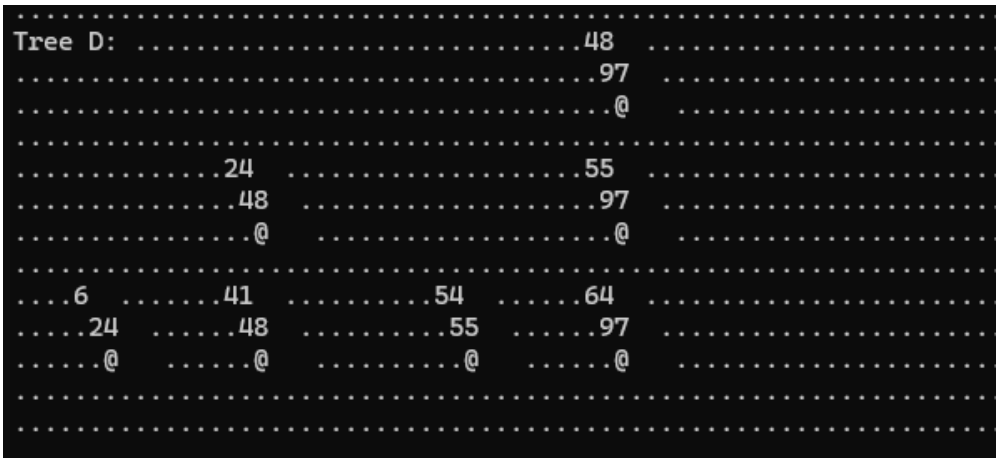

Рисунок 4 - Пример 2-3-дерева

Узлы хранят в себе значение и указатели на узел снизу и справа. Архитектура узла представлена на рисунке 5.

```
class Node {
private:
    int key;     //Ключ
    Node *next, *down;      //Указатели на узел справа и ниже
```

Рисунок 5 - Архитектура узла

Объекты класса дерево хранят в себе указатель на корневой узел, высоту дерева, его название и количество узлов. Архитектура дерева указана на рисунке 6.

```
class TwoThreeTree {    //Класс 2-3-дерева
    Node* root;
    int height;
    char name;
    int count;
    friend ReadIterator;
public:
    using value_type = int;
    TwoThreeTree(char n = 'T') : root(nullptr), height(0), name(n), count(0) {};
    const TwoThreeTree& operator = (const TwoThreeTree&);
    const TwoThreeTree& operator | (const TwoThreeTree&) const;
    const TwoThreeTree& operator & (const TwoThreeTree&) const;
    const TwoThreeTree& operator ^ (const TwoThreeTree&) const;
    const TwoThreeTree& operator / (const TwoThreeTree&) const; // вместо \ (разность)
    bool find(int num) const;
    void display();    //Вывод на экран
    int build(int);    //Построение дерева
    void genSet();     //Генерация данных
    int erase(int k);
    int getSize() const { return count; };
    ReadIterator begin();
    std::pair<ReadIterator, bool> insert(int k, ReadIterator where = ReadIterator());
    ~TwoThreeTree();
};
```

Рисунок 6 - Архитектура дерева

У класса дерева перегружены операторы для имитации математических операций над деревьями. Во всех перегрузках используется пользовательский итератор чтения, 2 экземпляра которого (для левого и правого деревьев-операндов) параллельно обходят деревья (постоянно «догоняют» друг друга), чтобы передавать в метод build, строящий деревья по возрастающей последовательности, ключи из листьев обоих деревьев в порядке возрастания. Пример перегрузки оператора представлен на рисунке 7.

```
const TwoThreeTree& TwoThreeTree::operator &(const TwoThreeTree& rightTreeOperand) const
{
    TwoThreeTree* temp = new TwoThreeTree;
    ReadIterator leftTreeIterator(*this);
    ReadIterator rightTreeIterator(rightTreeOperand);
    bool temporaryPointer = false;
    while (leftTreeIterator.ptr != NULLNODE && rightTreeIterator.ptr != NULLNODE) {
        while (leftTreeIterator.ptr != NULLNODE && rightTreeIterator.ptr != NULLNODE && *leftTreeIterator < *rightTreeIterator)
            leftTreeIterator++;
        while (leftTreeIterator.ptr != NULLNODE && rightTreeIterator.ptr != NULLNODE && *rightTreeIterator < *leftTreeIterator)
            rightTreeIterator++;
        if (leftTreeIterator.ptr != NULLNODE && rightTreeIterator.ptr != NULLNODE && *rightTreeIterator == *leftTreeIterator) {
            temp->build(*rightTreeIterator);
            if (!temporaryPointer) temporaryPointer = true;
            leftTreeIterator++;;
            rightTreeIterator++;
        }
    }
    if (temporaryPointer)
        temp->build(0);
    return *temp;
}
```

Рисунок 7 - Пример перегрузки оператора с использованием итератора чтения

Итератор чтения унаследован от std::iterator и имеет в себе указатель на текущий элемент и стек обхода. Архитектура итератора чтения представлена на рисунке 8.

```
struct ReadIterator : public std::iterator<std::forward_iterator_tag, int> {
    Node* ptr;
    std::stack<std::pair<Node*, int>> stack;
    ReadIterator(const TwoThreeTree& tree);
    ReadIterator(Node* p = nullptr) : ptr(p) { }
    ReadIterator(Node* p, const std::stack<std::pair<Node*, int>>&& St) : ptr(p), stack(std::move(St)) {}
    bool operator == (const ReadIterator& other) const { return ptr == other.ptr; }
    bool operator != (const ReadIterator& other) const { return !(*this == other); }
    ReadIterator& operator++();
    ReadIterator operator++(int) { ReadIterator temp(*this); ++* this; return temp; }
    pointer operator->() const { return &ptr->getKeyReference(); }
    reference operator*() const { return ptr->getKeyReference(); }
};
```

Рисунок 8 - Архитектура итератора чтения

«Надстройкой» над классом 2-3-дерева является класс последовательность. Архитектура класса последовательности представлена на рисунке 9.

```
class Sequence {
    TwoThreeTree keysTree;
    std::vector<int> values;
public:
    Sequence(std::initializer_list<int> valuesList);
    Sequence& merge(const Sequence& rightOperand);
    Sequence& substitute(const Sequence& rightOperand, int fromPosition);
    Sequence& erase(int fromPosition, int toPosition);
    void printTree();
};
#endif
```

Рисунок 9 - Архитектура класса последовательности

Класс последовательности хранит в себе массив ключей, которые хранятся в 2-3-дереве. В классе последовательности реализованы методы merge, substitute и erase, в которых использован итератор вставки (который в свою очередь использует итератор чтения). Реализация методов класса последовательности представлена на рисунке 10.

```
Sequence& Sequence::merge(const Sequence& rightOperand) {
    ReadIterator readIterator = keysTree.begin();
    InsertIterator<TwoThreeTree> insertIterator = inserter(keysTree, readIterator);
    for (int rightSequenceValues : rightOperand.values) {
        insertIterator = rightSequenceValues;
        values.push_back(rightSequenceValues);
    }
    return *this;
}

Sequence& Sequence::substitute(const Sequence& rightOperand, int fromPosition) {
    ReadIterator readIterator = keysTree.begin();
    InsertIterator<TwoThreeTree> insertIterator = inserter(keysTree, readIterator);
    for (size_t i = fromPosition; i < rightOperand.values.size(); ++i) {
        insertIterator = rightOperand.values[i];
        values.push_back(rightOperand.values[i]);
    }
    return *this;
}

Sequence& Sequence::erase(int fromPosition, int toPosition) {
    for (size_t index = fromPosition; index < toPosition; index++)
    {
        keysTree.erase(values[fromPosition]);
        values.erase(values.begin() + fromPosition);
    }
    return *this;
}
```

Рисунок 10 - Реализация методов класса последовательности

Примеры работы программы.

1. Создадим последовательность, в которой проверим реакцию контейнера на дублированный элемент. Сотрем 2 элемента с нулевой позиции и объединим ее с другой последовательностью, в которой проверим правильность построения дерева по невозрастающей последовательности. Часть программы указана на рисунке 11. Пример работы программы показан на рисунке 12.

```
    system("cls");
    Sequence T = { 2, 4, 5, 4, 7, 8, 10 };
    T.display();
    T.erase(0, 2);
    std::cout << "Sequence T after erase method: \n";
    T.display();
    Sequence D = { 1, 15, 6 };
    T.merge(D);
    std::cout << "Sequence T after merging with D: \n";
    T.display();
```

Рисунок 11 – Тестирование функций на последовательностях

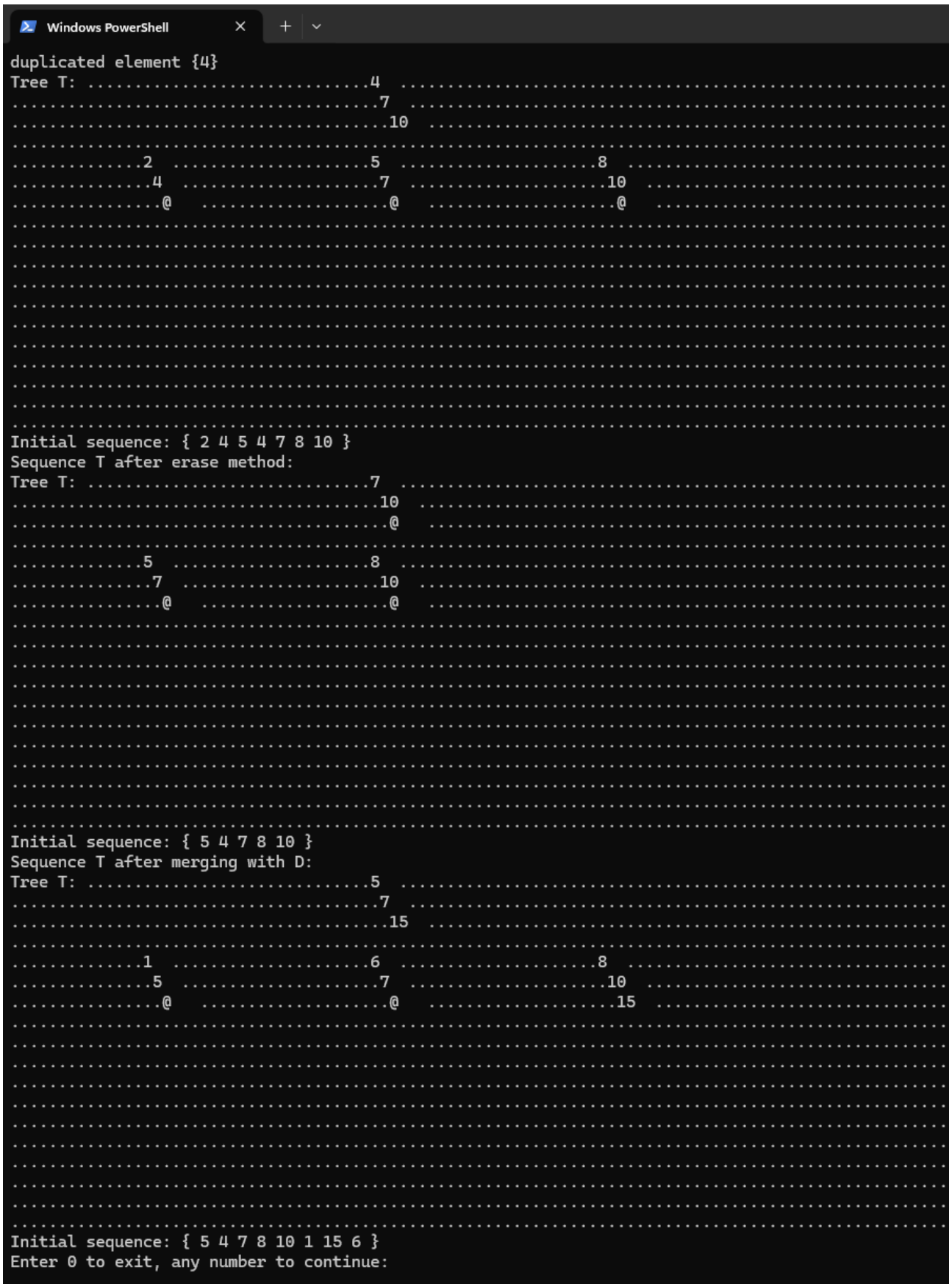

Рисунок 12 – Пример работы программы с последовательностями

2. Создадим случайные деревья и выведем результаты операций (A ⊕ (B \ C)) ∪ (D ⋂ E) на экран. На рисунке 13 представлен код программы. Вывод программы указан на рисунках 14-17.

```
A.display();
B.display();
C.display();
D.display();
E.display();
std::cout << "(B / C)\n";
(B / C).display();
std::cout << "(A ^ (B / C))\n";
(A ^ (B / C)).display();
std::cout << "(D & E)\n";
(D & E).display();
std::cout << "((A ^ (B / C)) | (D & E)\n";
((A ^ (B / C)) | (D & E)).display();
std::cout << "Enter 0 to exit, any number to continue: ";
```

Рисунок 13 – Тестирование программы

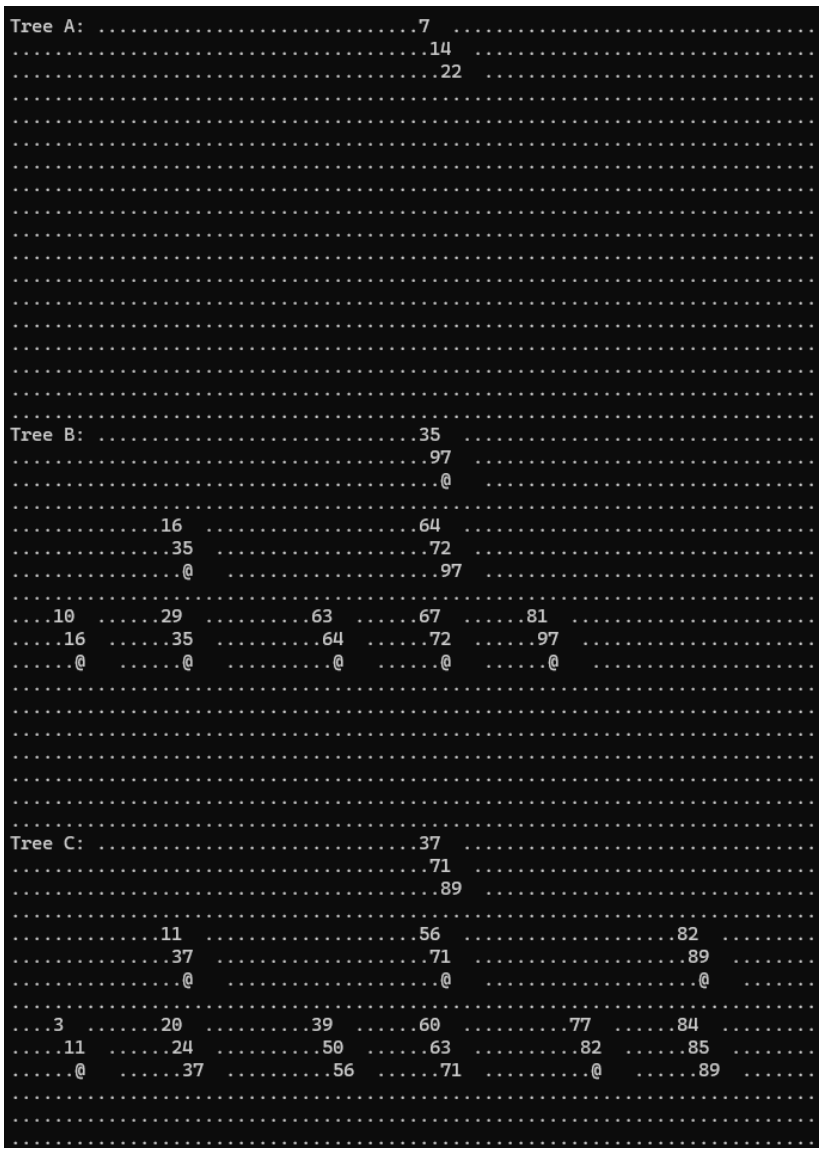

Рисунок 14 – Тестирование программы (часть 1/4)

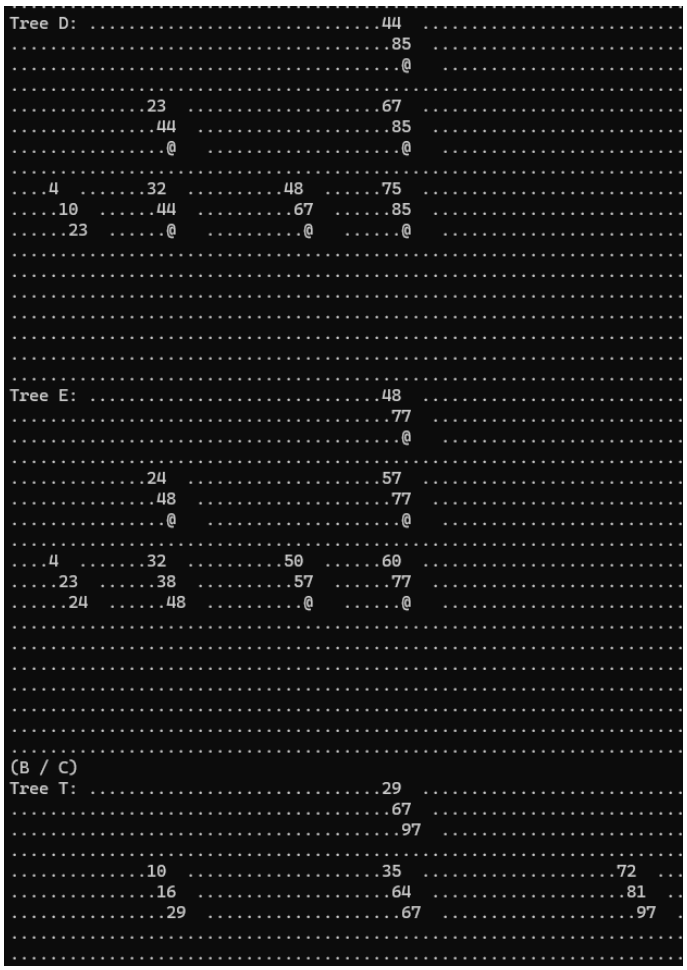

Рисунок 15 – Тестирование программы (часть 2/4)

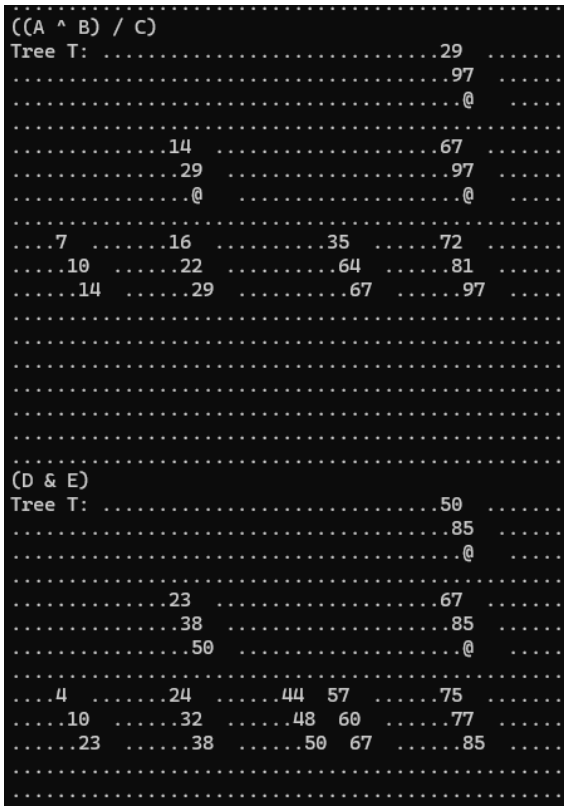

Рисунок 16 – Тестирование программы (часть 3/4)

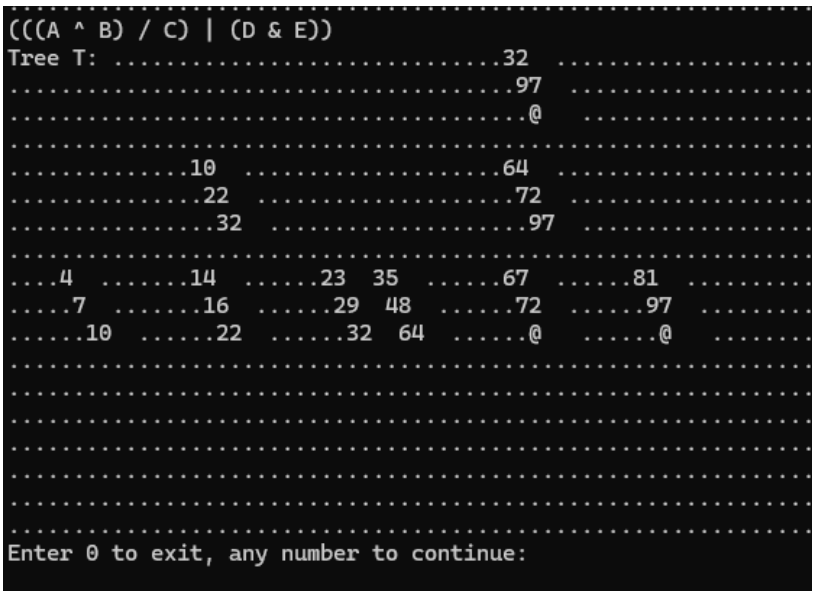

Рисунок 17 – Тестирование программы (часть 4/4)

Выводы.

В ходе выполнения лабораторной работы был создан эффективный контейнер для работы с множествами и последовательностями на основе структуры данных 2-3 дерева. Программа успешно выполняет цепочку операций над множествами, обеспечивая корректный вывод результатов на каждом этапе. Использование возможностей библиотеки алгоритмов позволило оптимизировать выполнение операций и улучшить читаемость кода.

Структура данных 2-3 дерево была выбрана благодаря своим свойствам сбалансированности и эффективности при выполнении операций вставки, удаления и поиска. Эти свойства обеспечили высокую производительность и устойчивость к изменениям в данных, что особенно важно для операций с большими наборами данных.

Работа продемонстрировала важность правильного выбора структур данных и алгоритмов для эффективного выполнения операций над множествами и последовательностями. Полученные результаты подтверждают, что поставленные цели достигнуты, и разработанный контейнер может быть эффективно использован в приложениях, требующих обработки множеств и последовательностей.

Ответы на контрольные вопросы.

1. Каким способом следует разметить дерево, чтобы в нём был возможен двоичный поиск? Как его следует нагрузить?

Для возможности двоичного поиска дерево должно быть организовано как бинарное дерево поиска (BST), где для каждого узла выполняется свойство: все значения в левом поддереве меньше значения узла, а все значения в правом поддереве больше значения узла. Нагрузить его следует таким образом, чтобы минимизировать дисбаланс, то есть стремиться к равномерному распределению узлов, что достигается, например, при использовании самобалансирующихся деревьев (например, AVL-дерево или красно-чёрное дерево).

2. Почему для операций с двоичным деревом дают две оценки сложности — «в худшем случае» и «в среднем»? Почему не рассматривается «лучший» случай?

Дают две оценки, потому что структура дерева может сильно варьироваться: в худшем случае (вырожденное дерево) время выполнения операций может быть O(n), в среднем случае (сбалансированное дерево) — O(logn). Лучший случай, когда все операции выполняются за O(1), редко имеет практическое значение, так как он маловероятен и не отражает реальную производительность структуры данных.

3. Отчего деревья двоичного поиска вырождаются?

Деревья двоичного поиска вырождаются из-за последовательных вставок элементов в уже отсортированном порядке, что приводит к образованию цепочки узлов, а не сбалансированного дерева.

4. Можно ли воспрепятствовать вырождению ДДП?

Да, можно. Для этого используют самобалансирующиеся деревья, такие как AVL-дерево или красно-чёрное дерево, которые автоматически поддерживают балансировку при вставках и удалениях, предотвращая вырождение.

5. Каков оптимальный алгоритм двуместной операции над множествами, представленными деревьями двоичного поиска?

Оптимальный алгоритм для двуместной операции (например, объединение, пересечение) над множествами, представленными деревьями двоичного поиска, заключается в обходе обоих деревьев одновременно (например, симметричным обходом) и выполнении нужной операции. Это позволяет эффективно обрабатывать элементы и сохранять свойства бинарного дерева поиска.

6. Какова временная сложность такой операции и как сказывается на ней возможное вырождение деревьев?

Временная сложность двуместной операции над сбалансированными деревьями двоичного поиска обычно составляет O(m+n), где m и n — количество элементов в деревьях. Если деревья вырождены, сложность может ухудшиться до O(m×n), так как приходится выполнять линейный поиск по вырожденным структурам.

7. Может ли при двуместной операции над множествами в ДДП получиться вырожденное дерево-результат?

Да, при выполнении двуместных операций (например, объединение или пересечение) результат может быть вырожденным деревом, особенно если исходные деревья сильно несбалансированы.

8. Можно ли хранить в дереве двоичного поиска множество с повторениями?

В стандартном дереве двоичного поиска (BST) нельзя хранить повторяющиеся элементы. Для хранения множеств с повторениями используют модифицированные структуры, такие как мультимножества или деревья поиска с подсчетом количества элементов.

9. Какая структура данных требует больше памяти для хранения множества: хеш-таблица или дерево двоичного поиска?

Хеш-таблица требует больше памяти, так как использует дополнительные структуры для управления коллизиями и поддержания производительности. Дерево двоичного поиска обычно требует меньше памяти, так как не требует дополнительной памяти для хеш-функций и коллизий.

10. Какая из них быстрее работает?

Хеш-таблица обычно работает быстрее для операций вставки, удаления и поиска в среднем случае O(1). Дерево двоичного поиска имеет временную сложность O(logn) для сбалансированных деревьев. Однако, если деревья вырождены, операции могут занимать O(n).

11. Как сделать не вырождающееся ДДП? Зачем оно может понадобиться?

Чтобы предотвратить вырождение, используются самобалансирующиеся деревья, такие как AVL-дерево или красно-чёрное дерево. Они автоматически поддерживают балансировку при вставках и удалениях. Несбалансированные деревья необходимы для обеспечения гарантированного быстрого доступа, вставки и удаления элементов.

12. Какая структура данных является оптимальной для хранения дерева двоичного поиска?

Оптимальной структурой для хранения дерева двоичного поиска является самобалансирующееся дерево, такое как AVL-дерево или красно-чёрное дерево. Эти структуры данных обеспечивают высокую производительность и гарантируют логарифмическую сложность операций в худшем случае.

13. Почему для хранения произвольной последовательности структуру данных для множества (хеш-таблицу или ДДП) приходится дорабатывать?

Множества, реализованные с использованием хеш-таблиц или деревьев двоичного поиска, не поддерживают порядок элементов, в то время как для произвольной последовательности важен порядок. Доработка необходима для хранения и управления упорядоченной информацией.

14. Какие доработки возможны?

Возможные доработки включают добавление индексов для поддержки упорядоченности, использование структур данных, которые поддерживают последовательности (например, сбалансированные деревья поиска с поддержкой индексов), или комбинирование структур, таких как связные списки с хеш-таблицами.

15. Можно ли предложить оптимальный вариант доработки?

Оптимальным вариантом может быть использование структур данных, таких как дерево поиска с поддержкой рангов (Ranked AVL Tree), которые поддерживают эффективные операции над последовательностями, включая вставку, удаление и доступ по индексу.

16. Влияет ли доработка структур данных для множеств для поддержки последовательностей на временную сложность операций над множествами?

Да, доработка может повлиять на временную сложность операций. Например, добавление индексов может увеличить сложность вставки и удаления, так как необходимо поддерживать порядок элементов. Однако для многих структур данных, таких как сбалансированные деревья поиска, увеличение сложности минимально.

17. Какую структуру данных проще дорабатывать — хеш-таблицу или ДДП?

Древовидные структуры данных (например, ДДП) проще дорабатывать для поддержки последовательностей, так как они изначально поддерживают некоторый порядок элементов. Хеш-таблицы, в свою очередь, требуют значительных изменений для поддержания порядка.

18. Какова оптимальная доработка структуры данных и временная сложность для операции исключения части последовательности между указанными позициями?

Оптимальная доработка для исключения части последовательности — использование сбалансированного дерева поиска с поддержкой индексов (например, AVL-дерево с рангами). Временная сложность удаления части последовательности в таком дереве составит O(logn+k), где k — количество удаляемых элементов.

19. То же — для операции вставки с указанной позиции.

Для вставки с указанной позиции в сбалансированном дереве поиска с поддержкой индексов временная сложность будет O(logn+k), где k — количество вставляемых элементов.

20. То же — для замены.

Для замены части последовательности в сбалансированном дереве поиска с поддержкой индексов временная сложность будет O(logn+k), где k — количество заменяемых элементов.

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

Стандартный контейнер библиотеки STL (Standard Template Library) — это шаблонный класс, который предоставляет способы хранения и управления коллекциями данных (например, векторы, списки, множества). Он отличается от обычного объекта тем, что предоставляет стандартные методы и интерфейсы для работы с элементами, а также интеграцию со стандартными алгоритмами STL.

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

Наиболее подходящими стандартными контейнерами для работы с множествами являются std::set и std::unordered_set. std::set хранит элементы в отсортированном порядке, а std::unordered_set использует хеш-таблицу для быстрого доступа.

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

Да, можно использовать std::unordered_set, который не требует отношения полного порядка, так как использует хеш-функции для организации элементов.

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

Да, существуют ограничения. Некоторые алгоритмы требуют, чтобы контейнеры были отсортированы или поддерживали отношение порядка. Например, алгоритмы из <algorithm> для работы с std::set предполагают, что контейнеры отсортированы.

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

Да, двуместные операции можно реализовать вручную, используя итераторы и методы контейнеров. Это может быть полезно, если требуется специфическая логика, не реализованная в стандартных алгоритмах.

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

Да, можно выполнять операции над последовательностями, используя стандартные алгоритмы STL или вручную написанные функции, обрабатывающие элементы множества.

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

Да, можно использовать контейнеры, такие как std::vector или std::list, которые поддерживают произвольные последовательности, в сочетании с множествами для управления упорядоченными данными.

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

Ожидаемая временная сложность для операции объединения двух отсортированных множеств std::set — O(n+m), где n и m — размеры множеств.

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

Оформление пользовательской структуры данных как стандартного контейнера позволяет использовать её с алгоритмами и функциями STL, улучшает совместимость и упрощает интеграцию с остальным кодом.

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

Для превращения пользовательской структуры данных в полноценный контейнер необходимо реализовать типы и методы, такие как iterator, const_iterator, begin(), end(), size(), empty(), и, возможно, insert(), erase(), и другие, в зависимости от потребностей.

31. Зачем нужны гарантии устойчивости алгоритмов относительно исключений?

Гарантии устойчивости алгоритмов к исключениям необходимы для обеспечения надежности и предсказуемости работы программы. Они позволяют уверенно работать с данными, даже если возникают ошибки или исключения.

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

Базовые гарантии лишь предотвращают утечки ресурсов и поддерживают целостность данных, но не гарантируют, что состояние программы останется неизменным. В критических системах требуются более строгие гарантии для предотвращения некорректного поведения или потери данных.

33. В чём заключаются расширенные гарантии устойчивости алгоритмах исключениям и как их можно обеспечить?

Расширенные гарантии включают строгую гарантию, что при возникновении исключения состояние программы останется прежним, или предоставляют возможность откатиться к предыдущему состоянию. Их можно обеспечить, используя техники копирования данных перед модификацией, транзакционные методы и тщательно обрабатывая все потенциальные исключения.

34. Отличия 2-3 деревьев от 1-2 деревьев, ДДП с сохранением высоты, Хэш-таблиц.

Сравнение структур данных представлено в таблице на рисунке 18.

Ключевые отличия

- Сбалансированность: 2-3 дерево и AVL-дерево всегда сбалансированы, красно-черное дерево приближено к сбалансированному, хеш-таблица не сбалансирована.

- Сложность операций: Все деревья имеют логарифмическую сложность операций, хеш-таблица имеет среднюю константную сложность.

- Структура: Различные структуры для поддержания баланса и организации данных.

- Применение: Зависит от требований к скорости операций, допустимой памяти и особенностей данных.

2-3 дерево

- Сбалансированность: Всегда идеально сбалансировано.

- Операции: Поиск, вставка и удаление за O(log n).

- Структура: Внутренние узлы могут иметь 2 или 3 детей.

- Применение: Хорошо подходит для файловых систем и баз данных.

1-2 дерево (Красно-черное дерево)

- Сбалансированность: Приближено к идеальному балансированию, высота не превышает 2log(n).

- Операции: Поиск, вставка и удаление за O(log n).

- Структура: Узлы окрашены в красный или черный цвет для поддержания баланса.

- Применение: Широко используется в стандартных библиотеках (например, std::map в C++).

Хеш-таблица

- Сбалансированность: Не сбалансировано.

- Операции: Поиск, вставка и удаление за O(1) в среднем случае, O(n) в худшем.

- Структура: Использует массив и хеш-функцию для распределения данных.

- Применение: Эффективно для ситуаций, где требуется быстрый доступ по ключу, например, кэширование.

Дерево динамического поиска с сохранением высоты (AVL-дерево)

- Сбалансированность: Всегда строго сбалансировано, разница высот поддеревьев не более 1.

- Операции: Поиск, вставка и удаление за O(log n).

- Структура: Узлы хранят информацию о высоте поддеревьев для поддержания баланса.

- Применение: Полезно в приложениях, где критичен худший случай поиска.

Характеристика	2-3 дерево	1-2 дерево (Красно-черное дерево)	Хеш-таблица	Дерево динамического поиска с сохранением высоты (AVL-дерево)
Сбалансированность	Всегда идеально сбалансировано	Приближено к идеальному балансированию	Не сбалансировано	Всегда строго сбалансировано
Сложность операций	O(log n)	O(log n)	O(1) в среднем, O(n) в худшем случае	O(log n)
Структура	Узлы имеют 2 или 3 детей	Узлы окрашены в красный или черный	Массив и хеш-функция	Узлы хранят информацию о высоте поддеревьев
Поддержание баланса	Перестройки при вставке и удалении	Перекрашивание и повороты	Не применяется	Повороты и обновление высот узлов
Применение	Файловые системы, базы данных	Стандартные библиотеки (например, std::map в C++)	Кэширование, быстрый доступ по ключу	Приложения с критичным худшим случаем поиска

Рисунок 18 – Сравнение структур данных