**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»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3311 | Пасечный Л.В.  Загуменнов И.М. |  |
| Преподаватель | Манирагена В. |  |

Санкт-Петербург

2025

**Введение**

**Цель работы:**

Целью данной лабораторной работы является практическое освоение комбинированных структур данных, совмещающих свойства упорядоченных деревьев и последовательных контейнеров. В рамках работы исследуется реализация пользовательского контейнера SetContainer, поддерживающего базовые и двуместные операции над множествами: объединение, пересечение, разность, а также модификации через подстановку, замену и удаление элементов. Дополнительно оцениваются временные характеристики этих операций. Работа направлена на развитие навыков проектирования эффективных структур данных, использования стандартной библиотеки шаблонов C++, а также закрепление знаний о деревьях двоичного поиска и их применении в задачах обработки множеств.

**Контрольные примеры:**

**Изображение выглядит как текст, Шрифт, снимок экрана

Автоматически созданное описание**

**Контрольные вопросы  
Хeш-таблицы:**

1. **Какой объём памяти нужно выделять под хеш-таблицу для хранения множеств со средней мощностью 50?**  
Следует выделять память минимум в 2 раза больше средней мощности множества — около 100 ячеек. Это снижает вероятность коллизий и повышает производительность операций.

2. **Хеш-таблица какого типа расходует больше памяти для хранения множества — с открытой адресацией или с цепочками переполнения?**  
Таблица с цепочками переполнения использует больше памяти, так как каждая ячейка содержит указатель на отдельный список значений, а не единственное значение.

3. **Каким требованиям должна удовлетворять «хорошая» хеш-функция?**  
Она должна равномерно распределять ключи по всем ячейкам таблицы, быть достаточно быстрой и независимой от специфики распределения входных данных.

4. **Можно ли построить хеш-таблицу, в которой не будет коллизий?**  
Да, но только если известны все ключи заранее и используется идеальная хеш-функция. В общем случае полностью исключить коллизии невозможно.

5. **Каков оптимальный алгоритм выполнения двуместной операции над множествами в хеш-таблице? Какова его временная сложность?**  
Алгоритм предполагает проход по одному множеству с поиском каждого элемента во втором. Его средняя временная сложность — O(n), где n — размер меньшего множества.

6. **Можно ли хранить в хеш-таблице множество с повторениями?**  
Да, можно. Хеш-таблица не запрещает повторения ключей — они просто попадают в цепочку коллизий, если такие есть.

7. **Какова временная сложность операций вставки и удаления элемента для хеш-таблицы?**  
В среднем — O(1), в худшем случае — O(n), если произошла сильная коллизия и элементы сгруппировались в одной цепочке.

8. **Почему для операций с хеш-таблицей дают две оценки временной сложности — сложность в худшем случае и сложность в среднем?**  
Потому что производительность сильно зависит от распределения ключей и качества хеш-функции. При плохом распределении возможна деградация до линейной сложности.

9. **Что такое вырождение хеш-таблицы и как его избежать?**  
Вырождение — это ситуация, когда большинство элементов попадает в одну ячейку, превращая таблицу в список. Чтобы избежать этого, нужно выбирать хорошую хеш-функцию и увеличивать размер таблицы при необходимости.

10. **Что нужно делать, если хеш-таблица переполнилась?**  
Необходимо увеличить размер таблицы (обычно вдвое) и пересчитать хеши для всех элементов с использованием новой хеш-функции или новых параметров.  
**Операции с деревом двоичного поиска (ДДП):**

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

Следует использовать внутренний (симметричный) обход. Узлы дерева должны быть нагружены ключами так, чтобы левое поддерево содержало только меньшие значения, а правое — большие или равные.

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

Лучший случай слишком специфичен и маловероятен, чтобы быть практически полезным. Анализируют «средний» и «худший» случаи для оценки устойчивости и предсказуемости алгоритмов.

3. **Отчего деревья двоичного поиска вырождаются?**

Они вырождаются при последовательной вставке уже упорядоченных ключей. Это превращает дерево в односторонний линейный список.

4. **Можно ли воспрепятствовать вырождению ДДП?**

Да, можно. Для этого используют алгоритмы балансировки — например, AVL-, красно-чёрные или 2–3-деревья.

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

Обход обоих деревьев в порядке возрастания и слияние полученных последовательностей с последующим построением нового сбалансированного дерева.

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

Сложность — O(n), если результат строится из отсортированных списков. При вырождении исходных деревьев и поэлементной вставке сложность возрастает до O(n²).

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

Да, если результат формируется путём последовательной вставки элементов. Это приводит к несбалансированному дереву.

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

Можно, но требуется доработка: например, хранить счётчик повторов или использовать списки значений при одинаковых ключах.

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

Хеш-таблица, особенно с цепочками переполнения, обычно требует больше памяти за счёт внутреннего запаса и хранения коллизий.

10. **Какая из них быстрее работает?**

Хеш-таблица быстрее при хорошем распределении хеш-функции. Но при плохом хешировании или переполнении эффективность падает.

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

Нужно использовать сбалансированные деревья (AVL, красно-чёрные и т. д.). Это обеспечивает предсказуемое логарифмическое время на операции.

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

Оптимальна структура с балансировкой и поддержкой дополнительной информации — например, мощность поддерева или высота.

**Хранение произвольных последовательностей:**1. **Почему для хранения произвольной последовательности структуру данных для множества (хеш-таблицу или ДДП) приходится дорабатывать?**

Они изначально не учитывают порядок элементов. Последовательность требует поддержки позиции и индексного доступа.

2. **Какие доработки возможны?**

Добавление индексов, хранения порядка, поддержки двусвязных элементов или деревьев с хранимой мощностью для позиционного доступа.

3. **Можно ли предложить оптимальный вариант доработки?**

Да, можно объединить дерево с массивом или использовать дерево с подсчётом мощностей для поддержки позиции и диапазонов.

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

Да, может увеличить сложность некоторых операций — например, вставки или удаления с позиции — до O(log n) вместо O(1).

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

Проще дорабатывать ДДП, так как в нём естественно поддерживается порядок и связи между элементами.

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

Использование сбалансированного дерева с мощностями поддеревьев. Сложность — O(log n + k), где k — длина подотрезка.

7. **То же — для операции вставки с указанной позиции.**

Также O(log n + k), если используется буфер вставки или дерево с индексами. Вставка требует обновления структурных связей.

8. **То же — для замены.**

Замена — это удаление и вставка, поэтому её сложность аналогична: O(log n + k). Оптимально реализуется в индексированных деревьях.

**Заключение**

В результате выполнения лабораторной работы была реализована структура SetContainer, совмещающая дерево двоичного поиска для эффективного поиска и вставки, а также вектор значений для последовательной обработки данных. Были реализованы операции объединения, пересечения, разности множеств, а также функции модификации набора элементов, такие как подстановка и замена подотрезков.

Полученные результаты подтвердили правильность логики операций и продемонстрировали устойчивую работу контейнера при выполнении множественных действий. Проведённый эксперимент показал, что реализация контейнера позволяет достигать высокой производительности при выполнении базовых и двуместных операций над множествами, особенно в случаях сбалансированного наполнения. Работа способствовала углублению понимания принципов организации комбинированных структур данных и их использования в прикладных задачах обработки информации.

**Источники:**

- Учебно-методическое пособие Колинько П. Г. – Последнее посещение 9.04 18:20  
- https://ru.stackoverflow.com/questions/1468952/%D0%9F%D1%80%D0%BE%D0%B1%D0%BB%D0%B5%D0%BC%D0%B0-%D1%81%D0%BE%D0%B7%D0%B4%D0%B0%D0%BD%D0%B8%D1%8F-%D0%BA%D0%BE%D0%BD%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D0%BE%D1%80%D0%B0-%D1%85%D1%8D%D1%88-%D1%82%D0%B0%D0%B1%D0%BB%D0%B8%D1%86%D1%8B-c- Последнее посещение 15.04 18:20  
- https://habr.com/ru/articles/267855/ - Последнее посещение 16.04 10:05

- https://ru.stackoverflow.com/questions/427915/%D0%9A%D0%B0%D0%BA-%D0%BF%D1%80%D0%B0%D0%B2%D0%B8%D0%BB%D1%8C%D0%BD%D0%BE-%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D1%8C-%D0%B0%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC-%D0%B3%D0%B5%D0%BD%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8-%D1%81%D0%BF%D0%B5%D1%86%D0%B8%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B9-%D0%BF%D0%BE%D1%81%D0%BB%D0%B5%D0%B4%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D0%B8-%D1%87%D0%B8%D1%81%D0%B5 - Последнее посещение 12.04 13:55

**Код программы**

#include <iostream>

#include <vector>

#include "SetContainer.h"

int main(){

    SetContainer A, B, C, D, E;

    A.insert(1); A.insert(2); A.insert(3);

    B.insert(2); B.insert(3); B.insert(4);

    C.insert(3); C.insert(4); C.insert(5);

    D.insert(4); D.insert(5); D.insert(6);

    E.insert(5); E.insert(6); E.insert(7);

    cout << "A :";

    A.printSet();

    cout << "B :";

    B.printSet();

    cout << "C :";

    C.printSet();

    cout << "D :";

    D.printSet();

    cout << "E :";

    E.printSet();

    SetContainer BC = B.intersection(C);

    std::cout << "B with C: ";

    BC.printSet();

    SetContainer AB\_C = A.unionSet(BC);

    std::cout << "A + (B with C): ";

    AB\_C.printSet();

    SetContainer AB\_C\_D = AB\_C.difference(D);

    std::cout << "(A + (B with C)) \\ D: ";

    AB\_C\_D.printSet();

    SetContainer finalResult = AB\_C\_D.intersection(E);

    std::cout << "Final Result: ";

    finalResult.printSet();

    A.setValues(A.erase(1, 2));

    A.printSet();

    B.setValues(B.subst(C.getValues(), 3));

    B.printSet();

    C.setValues(C.change(E.getValues(), 0));

    C.printSet();

    return 0;

}

#include "SetContainer.h"

#include <vector>

#include <iostream>

using namespace std;

void SetContainer::insert(int value){

    root = insertRecursive(this->root, value);

}

Node\* SetContainer::insertRecursive(Node\* node, int value){

    if (node == nullptr){

        node = new Node();

        node->value = value;

        node->left = nullptr;

        node->right = nullptr;

        this->values.push\_back(value);

        return node;

    }

    if (value < node->value){

        node->left = insertRecursive(node->left, value);

    }else if (value > node->value){

        node->right = insertRecursive(node->right, value);

    }

    return node; // Удаляем этот вызов, так как он добавляет значение в вектор даже если оно уже существует

}

vector<int> SetContainer::erase(int start, int end) {

    vector<int> result;

    result.insert(result.end(), this->values.begin(), this->values.begin() + start);

    result.insert(result.end(), this->values.begin() + end + 1, this->values.end());

    return result;

}

vector<int> SetContainer::subst(const vector<int>& B, int pos) {

    vector<int> result;

    result.insert(result.end(), this->values.begin(), this->values.begin() + pos);

    result.insert(result.end(), B.begin(), B.end());

    result.insert(result.end(), this->values.begin() + pos, this->values.end());

    return result;

}

vector<int> SetContainer::change(const vector<int>& B, int pos) {

    if (pos + B.size() > this->values.size()) {

        cerr << "Error: Out of bounds" << endl;

        return this->values;

    }

    vector<int> result = this->values;

    copy(B.begin(), B.end(), result.begin() + pos);

    return result;

}

Node\* SetContainer::getRoot(){

    return this->root;

}

bool SetContainer::find(int value) const{

    return findRecursive(root, value);

}

bool SetContainer::findRecursive(Node\* node, int value) const{

    if (node == nullptr){

        return false;

    }

    if (value == node->value){

        return true;

    }

    if (value < node->value){

        return findRecursive(node->left, value);

    }else{

        return findRecursive(node->right, value);

    }

}

void SetContainer::remove(int value){

    root = removeRecursive(root, value);

}

Node\* SetContainer::removeRecursive(Node\* node, int value){

    if (node == nullptr){

        return nullptr;

    }

    if (value < node->value){

        node->left = removeRecursive(node->left, value);

    }else if (value > node->value){

        node->right = removeRecursive(node->right, value);

    }else{

        if (node->left == nullptr && node->right == nullptr){

            delete node;

            return nullptr;

        }else if (node->left == nullptr){

            Node\* temp = node->right;

            delete node;

            return temp;

        }else{

            Node\* minNode = findMin(node->right);

            node->value = minNode->value;

            node->right = removeRecursive(node->right, minNode->value);

        }

    }

    return node;

}

Node\* SetContainer::findMin(Node\* node) const{

    while (node->left != nullptr){

        node = node->left;

    }

    return node;

}

vector<int> SetContainer::getValues() const{

    return this->values;

}

SetContainer SetContainer::unionSet(const SetContainer& other) const {

    SetContainer result;

    for (int value : this->values) {

        result.insert(value);

    }

    for (int value : other.getValues()) {

        result.insert(value);

    }

    return result;

}

SetContainer SetContainer::intersection(const SetContainer& other) const {

    SetContainer result;

    for (int value : this->values) {

        if (other.find(value)) {

            result.insert(value);

        }

    }

    return result;

}

SetContainer SetContainer::difference(const SetContainer& other) const {

    SetContainer result;

    for (int value : this->values) {

        if (!other.find(value)) {

            result.insert(value);

        }

    }

    return result;

}

void SetContainer::printSet(){

    if (this->root == nullptr){

        cout << "Set is empty" << endl;

        return;

    }

    cout << "{";

    for (size\_t i = 0; i < this->values.size(); i++){

        cout << " " << values[i];

        if (i < this->values.size() - 1) {

            cout << ",";

        }

    }

    cout << " }" << endl;

}

void SetContainer::setValues(vector<int> values){

    this->values = values;

}

#include <vector>

using namespace std;

struct Node{

    int value;

    Node\* left;

    Node\* right;

};

class SetContainer{

    private:

        Node\* root;

        Node\* insertRecursive(Node\* node, int value);

        bool findRecursive(Node\* node, int value) const;

        Node\* removeRecursive(Node\* node, int value);

        Node\* findMin(Node\* node) const;

        vector<int> values;

    public:

        SetContainer() : root(nullptr){}

        void insert(int value);

        bool find(int value) const;

        void remove(int value);

        SetContainer unionSet(const SetContainer& other) const;

        SetContainer intersection(const SetContainer& other) const;

        SetContainer difference(const SetContainer& other) const;

        Node\* getRoot();

        void printSet();

        vector<int> getValues() const;

        vector<int> erase(int start, int end);

        void setValues(vector<int> values);

        vector<int> subst(const vector<int>& B, int pos);

        vector<int> change( const vector<int>& B, int pos);

};