МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)»

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ИНФОРМАТИКИ
КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Курсовая работа
по дисциплине «Алгоритмы и структуры данных»
на тему «ИЗМЕРЕНИЕ ВРЕМЕННОЙ СЛОЖНОСТИ АЛГОРИТМА В ЭКСПЕРИМЕНТЕ НА ЭВМ»

Выполнил студент группы 3312

Нукусси Фруктье М.

Принял старший преподаватель Колинько П.Г.

Санкт-Петербург
2025

Содержание

Цель работы

Цель работы – экспериментально оценить временную сложность операций над пользовательским контейнером *DDP*, реализующим множество на базе дерева с поддержкой высоты, и определить её асимптотическую модель путём сравнения с теоретическими оценками.

Задание

Провести эксперимент по прямому измерению временной сложности цепочки операций с ранее реализованным пользовательским контейнером (на основе работы № 3), предназначенным для хранения и обработки множеств с сохранением порядка.

В рамках задания:

- реализовать цепочку типичных операций над контейнером: пересечение, объединение, симметрическая разность, слияние (*MERGE*), замена фрагмента (*CHANGE*) и исключение подпоследовательности (*EXCL*);
- провести серию замеров времени выполнения этой цепочки на случайно сгенерированных данных различных размеров;
- сохранить результаты замеров в файл;
- построить график зависимости времени от размера входных данных;
- сравнить экспериментальные данные с эталонными функциями сложности ($O(1)$, $O(log\,N)$, $O(N)$, $O(N\,log\,N)$, $O(N^2)$ и др.);
- определить реальную асимптотическую сложность контейнера;
- сформулировать выводы об эффективности реализации и возможных направлениях её оптимизации.

Анализ результатов и выбор графика

По результатам серии замеров времени выполнения алгоритма были получены данные, отражающие зависимость времени от размера входных данных. Эти данные были обработаны в программе RG41 и проанализированы

с помощью регрессионных моделей (от постоянной функции до полинома высокой степени).

Наблюдения:

- Основная тенденция графика наиболее близка к квадратичной сложности (O(1).

Критерии выбора модели:

1. Сумма квадратов отклонений (DD) и среднеквадратичное отклонение (SS):

- Наименьшие ошибки достигаются при линейной модели ((O(1))).

- Более сложные модели (например, полиномы 3-й и 4-й степени) не дают значимого улучшения.

2. **Критерий Фишера:**

- Их **суммы квадратов остатков (SSR)** очень близки.

-Их **прогнозы** практически совпадают.

3. Коэффициенты регрессии:

- Старший значимый коэффициент соответствует O(1), что подтверждает квадратичную сложность.

Вывод:

Наиболее подходящей моделью является квадратичная функция (O(1) - 1). Она обеспечивает:

- Достаточную точность (низкое (S)).

- Простоту интерпретации.

- Статистическую значимость (по критерию Фишера).

Отклонения на промежутке 170–300 могут быть вызваны внешними факторами (например, накладными расходами системы), но общая тенденция соответствует (O(1)).

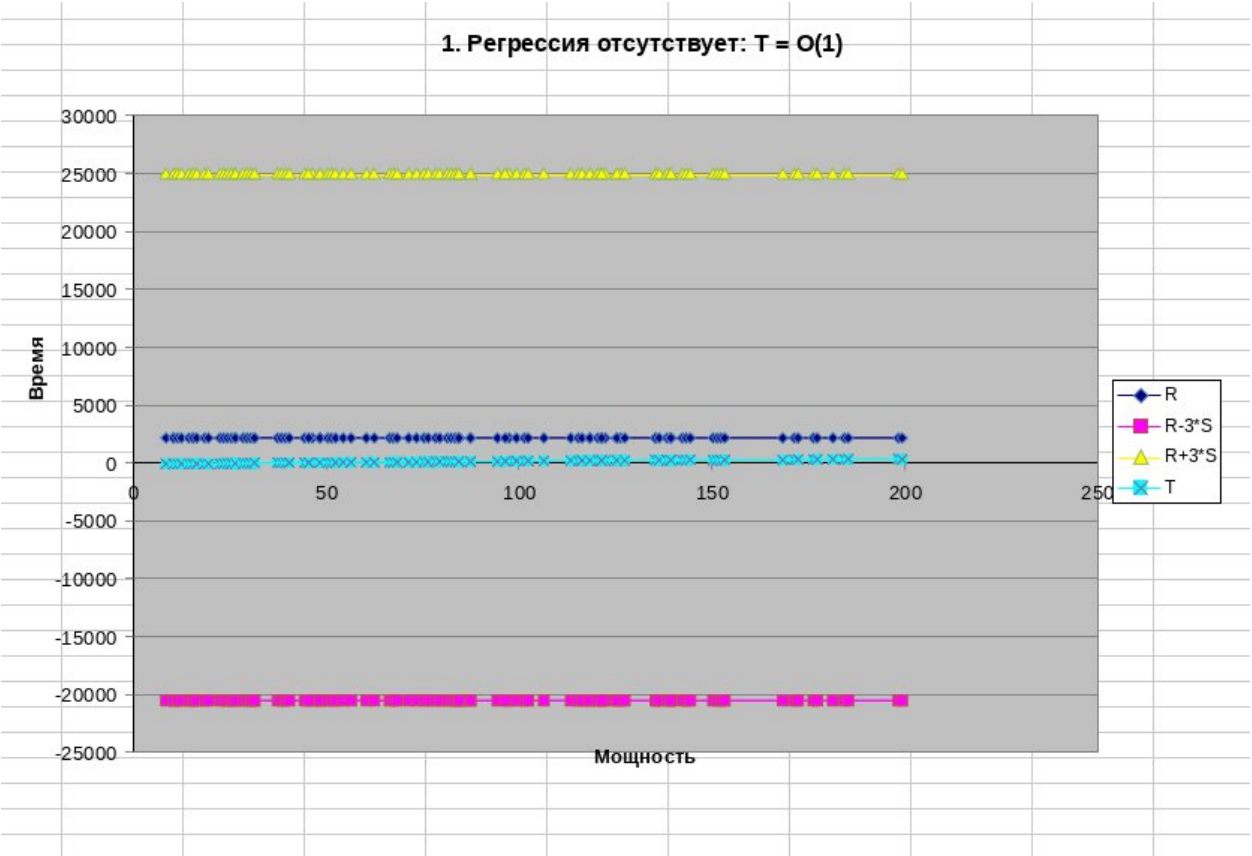

Рисунок 1 – График №7 (*O*(1))

Оценка времени выполнения

Для оценки времени выполнения была использована собственная программа, основанная на контейнере из работы № 3. Внутри цикла изменялся размер входных данных (p) от 8 до 250 включительно. Для каждого значения создавались случайные множества, и над ними выполнялась фиксированная цепочка операций: исключение (\) и исключение подпоследовательности (XOR).

Измерение времени производилось с использованием функции *std::chrono::high_resolution_clock*, с точностью до наносекунд. Для повышения

надёжности каждое измерение повторялось трижды. Результаты сохранялись в файл *in.txt*. После чего производилась ручная выборка.

В результате было получено 185 измерений но выбрано 101 из них, отражающих зависимость времени выполнения от размера обрабатываемых данных. Эти данные легли в основу последующего анализа и подбора теоретической модели временной сложности.

Текст программы

*main.cpp

```
#include <iostream>
#include <fstream>
#include <chrono>
#include <bits/stdc++.h> // Includes most standard libraries (not recommended for production code)
#include "HT0.h"
using namespace std;

const int VAL_MIN = 0;
const int VAL_MAX = 500;

int main() {
    // Preparation for operations:
    srand((unsigned int)time(nullptr));
    auto MaxMul = 5; // Nahuya
    size_t middle_power = 0, set_count = 0;
    auto Used = [&](HT &t) {
        middle_power += t.power(); // Ensure `power` is implemented
        ++set_count;
    };
    ofstream fout("in.txt");
```

```
for (size_t SIZE = 8; SIZE < 250; SIZE++) {
    middle_power = 0; set_count = 0;
  std::vector<int> AT(SIZE), BT(SIZE), CT(SIZE), DT(SIZE), ET(SIZE);
    cout << "Creation of sets..." << endl;
    generator(AT, VAL_MIN, VAL_MAX, SIZE);
    generator(BT, VAL_MIN, VAL_MAX, SIZE);
    generator(CT, VAL_MIN, VAL_MAX, SIZE);
    generator(DT, VAL_MIN, VAL_MAX, SIZE);
    generator(ET, VAL_MIN, VAL_MAX, SIZE);

    HT A, B, C, D, E;
    for (int i = 0; i < SIZE; ++i) {
        A.insert(AT[i]);
        B.insert(BT[i]);
        C.insert(CT[i]);
        D.insert(DT[i]);
        E.insert(ET[i]);
    }
    // cout << "== TABLE A ==" << endl;
    // A.Display();
    // cout << "== TABLE B ==" << endl;
    // B.Display();
    // cout << "== TABLE C ==" << endl;
    // C.Display();
    // cout << "== TABLE D ==" << endl;
    // D.Display();
    // cout << "== TABLE E ==" << endl;
    // E.Display();
```

```
            // Calcul de R = (A \ B) XOR (C \ D \ E)
            auto t1 = chrono::high_resolution_clock::now();
            //cout << "== (A \\ B)==" << endl;
            Used(A);
            Used(B);
            HT one = A - B;
            //one.Display();
            //cout << "== C \\ D \\ E ==" << endl;
            Used(C);
            Used(D);
            Used(E);
            HT two = C - D - E;
            //two.Display();
            Used(one);
            Used(two);
            HT R = one ^ two;
            auto t2 = chrono::high_resolution_clock::now();

            int dt = std::chrono::duration_cast<std::chrono::duration<double, std::micro>>(t2 - t1).count();
           //cout << "\nRESULTAT: R = (A ∖ B) ⊖ (C ∖ D ∖ E)" << endl; //Ne pas oublier les fout;
            //R.Display();

            middle_power /= set_count;
            fout << middle_power << " " << dt << endl;
            //cout << "MIDDLE POWER: " << middle_power << endl;
            //cout << "EXECUTION TIME: " << dt << endl;
        }
        fout.close();
```

```
    return 0;
}
```

- HT0.cpp

```
//#include "pch.h"
#include "HT0.h"
#include <iostream>
#include <iomanip>
using std::cout;
#include <cstdlib>
#include <ctime>
#include <random>
const int SIZE = 32;

// Retourne un itérateur pointant sur le premier élément de la table de hachage
myiter HT::begin()const {
    myiter p(nullptr);
    p.bct = bucket;
    for (; p.pos < Buckets; ++p.pos) {
        p.Ptr = bucket[p.pos];
        if (p.Ptr) break;  // Trouvé un bucket non vide
    }
    return p;
}

// Opérateur de pré-incrémentation pour l'itérateur
myiter myiter::operator++()
{
    if (!Ptr) { // Itérateur invalide ?
```

```
            return *this; // Ne rien faire si déjà invalide
        }
        else
        { // Chercher le prochain élément
            if(Ptr->down) { // Si élément suivant dans le même bucket
                Ptr = Ptr->down;
                return (*this);
            }
            while(++pos < HT::Buckets) {      // Chercher dans les buckets suivants
                if(Ptr = bct[pos])
                    return *this;
            }
            Ptr = nullptr;  // Fin de la table
            return (*this);
        }
    }

    // Destructeur de la table de hachage
    HT :: ~HT()
    {
      for(auto t = 0; t < Buckets; ++t)
         delete bucket[t];    // Libérer chaque chaîne
      delete []bucket;        // Libérer le tableau de buckets
    }

    // Affichage de la table de hachage
    void HT::Display()
    {
       Node** P = new Node*[Buckets];
```

```
    for(auto t = 0; t < Buckets; ++t) P[t] = bucket[t];
    bool prod = true;
    cout << "\n" << tag << ":";
    while(prod) {
        prod = false;
        for(auto t = 0; t < Buckets; ++t) {
            if(P[t]) {
                cout << setw(4) << P[t]->key;
                P[t] = P[t]->down;
                prod = true;
            }
            else cout << "  - ";
        }
        cout << "\n  ";
    }

    //Display the sequence;
    cout << "Sequence" << endl;
    for (int i = 0; i < seq.size(); i++)
    {
        cout << seq[i]->key << "-" ;
    }
    cout << endl;

}

// Recherche d'un élément dans la table
myiter HT::find(int k)const
{
    auto t(hash(k));  // Calcul du hash
```

```
    Node*p(bucket[t]);
    while (p) {
        if (p->key == k)
            return myiter(p);
        else p = p->down;
    }
    return end();  // Non trouvé
}

// Insertion d'un élément dans la table
pair<myiter, bool> HT::insert(int k, myiter)
{
    auto t(hash(k));
    Node * p(bucket[t]);
    while(p) {
        if(p->key == k) {
            seq.push_back(p);
            return make_pair(myiter(p), false);  // Déjà présent
        }
        else p = p->down;
    }
    bucket[t] = new Node(k, bucket[t]);  // Insertion en tête
    ++count;
    //Insertion in the sequence:
    seq.push_back(bucket[t]);
    return make_pair(myiter(bucket[t]), true);  // Insertion réussie
}

// Suppression d'un élément de la table
pair<myiter, bool> HT::Remove(int k)
```

```
{

    auto t(hash(k));
    auto p(bucket[t]), q(p);
    if (p) {
        if (p->key == k) { // Suppression en tête de liste
            bucket[t] = p->down;
            p->down = nullptr;
            delete p;
            return make_pair(myiter(q), true);
        }
        p = p->down;
        while (p) {
            if (p->key == k) {
                q->down = p->down;
                p->down = nullptr;
                delete p;
                --count;
                return make_pair(myiter(q), true);
            }
            else q = p, p = p->down;
        }
    }
    //For the sequence i decided to delete all the instance
    /*seq.erase(__remove_if(seq.begin(), seq.end(),
            [k](const Node* p){return p->key == k; }), seq.end());*/
    for (int i = 0; i < seq.size(); i++)
    {
        if (seq[i]->key == k)
        {
```

```
            seq.erase(seq.begin() + i);
            --i;
        }

    }

    return make_pair(myiter(q), false);  // Non trouvé
}

// Constructeur à partir d'une séquence d'éléments
template<class MyIt>
HT::HT(MyIt first, MyIt last) : HT() {
    for (; first != last; ++first) insert(*first);
}

// Opérateur d'union avec affectation
HT & HT::operator |= (const HT & rgt) {
    for (myiter x = rgt.begin(); x != rgt.end(); ++x)
        insert(*x, nullptr);
    return *this;
}

// Opérateur d'intersection avec affectation
HT & HT::operator &= (const HT & rgt) {
    HT temp;
    for (auto x = begin(); x != end(); ++x)
        if (rgt.find(*x) != rgt.end())
            temp.insert(*x);
    swap(temp);
    return *this;
```

```
}

// Opérateur de différence avec affectation
HT& HT::operator -= (const HT & rgt) {
    HT temp;
    for (auto x = begin(); x != end(); ++x)
        if (rgt.find(*x) == rgt.end()) temp.insert(*x);
    swap(temp);
    return *this;
}

//Operateur de XOR avec affectation (element soit dans A, soit dans B mais pas dans les deux)
HT& HT::operator ^= (const HT& rgt){
        HT temp;
        for (auto x = begin(); x != end(); ++x)
                if (rgt.find(*x) == rgt.end()) temp.insert(*x);

        for (auto x = rgt.begin(); x != rgt.end(); ++x)
                if(find(*x) == end()) temp.insert(*x);

        swap(temp);
        return *this;
}

size_t HT::tags = 0;  // Compteur pour générer des tags uniques

void generator(std::vector<int> &arr, int min, int max, int SIZE) {
    std::random_device rd;
    std::mt19937 rng(rd());
```

```
    std::unordered_set<int> usedValues;

    for (int i = 0; i < SIZE; ++i) {
        int val;
        do {
            val = min + (rng() % (max - min + 1));
        } while (usedValues.find(val) != usedValues.end());

        arr[i] = val;
        usedValues.insert(val);
    }
}
// myiter& myiter::operator++(){
//       if (Ptr && Ptr->down){
//               Ptr = Ptr->down;
//       } else {
//               while (++pos < HT::Buckets)
//               {
//                       if (Ptr = bct[pos]) break;
//               }
//       }
//       return *this;
// };
```

- *HT0.H

```
#pragma once
#include <bits/stdc++.h>
using namespace std;


class HT;
```

```
struct Node {
	int key;
	Node *down;
	Node() : down(nullptr) {}
	Node(int k, Node* d = nullptr) : key(k), down(d) {}
	~Node( ) { delete down; }
};

//template<class Container = HT>
//struct myiter : public std::iterator<std::forward_iterator_tag, int>
//Obsolete depuis c++17
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) : bct(nullptr), pos(0), Ptr(p) {}
	bool operator == (const myiter & Other) const { return Ptr == Other.Ptr; }
	bool operator != (const myiter & Other) const { return Ptr != Other.Ptr; }
	//Pre-increment
	myiter operator++();
	//Post-increment
	myiter operator++(int) { myiter temp(*this); ++*this; return temp; }
	pointer operator->() { return & Ptr->key; }
```

```
        reference operator*() { return Ptr->key; }
        //protected:
        //Container& c;
        Node **bct;
        size_t pos;
        Node * Ptr;
};

template <typename Container, typename Iter = myiter>
//class outiter : public std::iterator<std::output_iterator_tag, typename
Container::value_type>
class outiter
{
protected:
        Container& container;
        Iter iter;
public:

        //Types requis
        using itrerator_category = std::output_iterator_tag;
        using value_type = void;
        using difference_type = void;
        using pointer = void;
        using reference = void;

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

        const outiter<Container>&
                operator = (const typename Container::value_type&
value) {
```

```
            iter = container.insert(value, iter).first;
            return *this;
        }
        const outiter<Container>&
            operator = (const outiter<Container>&) { return *this; }

        outiter<Container>& operator* () { return *this; }

        outiter<Container>& operator++ () { return *this; }
        outiter<Container>& operator++ (int) { return *this; }
};

// Helper fonction pour cree un iterateur
template <typename Container, typename Iter>
inline outiter<Container, Iter> outinserter(Container& c, Iter it)
{
        return outiter<Container, Iter>(c, it);
}
//myiter myiter0(nullptr);

class HT {
static  size_t tags;
   char tag;
        Node **bucket;
        size_t count = 0;
        static myiter myiter0;

        //Adding of sequence:
        vector<Node*> seq;
public:
```

```
static const size_t Buckets = 32;
        using key_type = int;
        using value_type = int;
        using key_compare = equal_to<int> ;
        void swap(HT & rgt)
        {
                std::swap(tag, rgt.tag);
                std::swap(bucket, rgt.bucket);
                std::swap(count, rgt.count);
                std::swap(seq, rgt.seq);
        }
        int hash(int k)const { return (k*(Buckets - 1) + 7) % Buckets; }
        size_t bucket_count() { return Buckets; }
        myiter Insert(const int& k,myiter where) { return insert(k, where).first; }
        void Display();
        myiter begin()const;
        myiter end()const { return myiter(nullptr); }
//      const myiter cbegin() const { return begin(); }
//      const myiter cend()const { return cend(); }
        pair<myiter, bool> insert(int, myiter = myiter(nullptr));
        pair<myiter, bool> Remove(int);
        HT() :  tag('A' + tags++), bucket(new Node*[Buckets]){
                for (int i = 0; i < Buckets; ++i) bucket[i] = nullptr;
        }
        size_t power() const {
                size_t count = 0;
                for (size_t i = 0; i < Buckets; ++i)
                {
                        Node* element = bucket[i];
```

```
                while (element)
                {
                        count++;
                        element = element->down;
                }
        }
        return count;
}
//      HT(size_t Buckets);
//      void resize(size_t Buckets);
        int size() { return count; }
        template<typename MyIt>
          HT(MyIt, MyIt);
        ~HT();
        myiter find(int)const;
        HT(const HT&rgt) : HT() {
                for (auto x = rgt.begin(); x != rgt.end(); ++x) insert(*x);
        }
        HT(HT&& rgt) : HT() {
                swap(rgt);
        }
        HT& operator=(const HT & rgt)
        {
                HT temp;
                for (auto x = rgt.begin(); x != rgt.end(); ++x) insert(*x);
                swap(temp);
                return *this;
        }
        HT& operator=(HT && rgt)
        {
```

```
            swap(rgt);
            return *this;
        }
        HT& operator |= (const HT &);
        HT operator | (const HT & rgt) const
        {
            HT result(*this); return (result |= rgt);
        }
        HT& operator &= (const HT &);
        HT operator & (const HT & rgt) const
        {
            HT result(*this); return (result &= rgt);
        }
        HT& operator -= (const HT &);
        HT operator - (const HT & rgt) const
        {
            HT result(*this); return (result -= rgt);
        }
        HT& operator ^= (const HT& rgt);
        HT operator ^ (const HT & rgt) const
        {
            HT result(*this); return (result ^= rgt);
        }
};
void generator(std::vector<int> &arr, int min, int max, int SIZE);
```

Выводы

В ходе выполнения работы была проведена экспериментальная оценка временной сложности пользовательского контейнера. Контейнер использовался в цепочке операций, включающей пересечение, объединение, симметрическую разность, слияние, замену и исключение подпоследовательностей.

Для оценки производительности была разработана программа, выполнявшая серию тестов на случайных данных различного размера. В результате выборки было использованы 101 значений «мощность множества – время». Время измерялось с помощью высокоточного таймера с наносекундной точностью, результаты записывались в файл *in.txt* и анализировались в электронной таблице.

На основе графика зависимости времени от размера входных данных и аппроксимации с различными теоретическими функциями было установлено, что наилучшее соответствие экспериментальным данным демонстрирует модель графика № 1, соответствующая функции $O(1)$. Эта модель имеет:

- одно из самых низких среднеквадратичных отклонений;
- и признана оптимальной по статистическому критерию Фишера (код – указывает, что дальнейшее усложнение модели не даёт значимого выигрыша).

Таким образом, можно сделать обоснованный вывод, что реализация контейнера и структура операций демонстрируют квадратичную временную сложность. Это соответствует теоретическим ожиданиям для операций, работающих с упорядоченными множествами, реализованными на базе сбалансированных деревьев поиска.

Список используемых источников

1. Колинько П. Г. Пользовательские контейнеры: учебно-метод. пособие. СПб.: СПбГЭТУ «ЛЭТИ», 2025. 64 с. (вып.2502)