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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Измерение временной сложности алгоритма в эксперименте на ЭВМ

Студентка гр. 2308 _____ Рымарь М.И.

Преподаватель _____ Манирагена В.

Санкт-Петербург

2024

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студентка Рымарь М.И.

Группа 2308

Тема работы: Измерение временной сложности алгоритма в эксперименте на ЭВМ

Исходные данные:

На основе программы, составленной по гл. 3, выполнить статистический эксперимент по измерению фактической временной сложности алгоритма обработки данных.

Программа дорабатывается таким образом, чтобы она генерировала множества мощностью, меняющейся, например, от 10 до 200, измеряла время выполнения цепочки операций над множествами и последовательностями и выводила результат в текстовый файл. Каждая строка этого файла должна содержать пару значений «размер входа — время» для каждого опыта. Затем эти данные обрабатываются, и по результатам обработки делается заключение о временной сложности алгоритма.

Для повышения надежности эксперимента следует предусмотреть в программе перехват исключительных ситуаций. Можно сделать так, чтобы любой сбой сводился просто к пропуску очередного шага эксперимента. В частности, рекомендуется перехватывать ситуацию bad_alloc, возбуждаемую конструктором при нехватке памяти.

Содержание пояснительной записки:

Содержание, Задание к курсовой работе, Введение, Результаты эксперимента, Заключение, Список использованных источников.

Предполагаемый объем пояснительной записки:

Не менее 10 страниц.

Дата выдачи задания: 11.03.2024

Дата сдачи реферата: 06.06.2024

Дата защиты реферата: 06.06.2024

Студентка _____ Рымарь М.И.

Преподаватель _____ Манирагена В.

АННОТАЦИЯ

В курсовой работе изучается измерение временной сложности алгоритма обработки данных экспериментальным методом на ЭВМ. Программа, основанная на разработках главы 3, генерирует множества размером от 10 до 200 элементов и измеряет время выполнения операций над ними, записывая результаты в текстовый файл. Для повышения надежности эксперимента предусмотрена обработка исключений, таких как нехватка памяти. Анализ полученных данных позволяет оценить фактическую временную сложность алгоритма и его эффективность в различных условиях.

SUMMARY

This coursework examines the measurement of algorithm time complexity for data processing using experimental methods on a computer. Based on the developments in Chapter 3, the program generates sets ranging in size from 10 to 200 elements and measures the execution time of operations on them, recording the results in a text file. To enhance the reliability of the experiment, exception handling, such as memory allocation failures, is implemented. Analysis of the collected data allows for the assessment of the algorithm's actual time complexity and its efficiency under various conditions.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Измерение временной сложности алгоритмов является важным аспектом теории и практики информатики. Знание временной сложности позволяет оценить эффективность алгоритма и прогнозировать его производительность на различных объемах данных. Особенно актуально это становится в условиях растущих объемов информации и необходимости оптимизации вычислительных процессов.

Целью данной курсовой работы является проведение экспериментального исследования для измерения фактической временной сложности алгоритма обработки данных. Для достижения этой цели требуется выполнить следующие задачи:

- Разработать и модифицировать программу, способную генерировать множества различной мощности (от 10 до 200 элементов).
- Обеспечить выполнение цепочки операций над множествами и последовательностями, измеряя время их выполнения.
- Организовать запись результатов эксперимента в текстовый файл в формате «размер входа — время выполнения».
- Реализовать обработку исключительных ситуаций, таких как нехватка памяти, для повышения надежности эксперимента.
- Провести анализ полученных данных и сделать выводы о фактической временной сложности алгоритма.

1. ТЕОРЕТИЧЕСКИЕ ОЦЕНКИ ВРЕМЕННОЙ СЛОЖНОСТИ

В данном разделе рассматриваются теоретические оценки временной сложности для выполнения основных операций над множествами и последовательностями, а также для операций над 2-3 деревом. Анализируются операции вставки, удаления, изменения, а также специфические операции для множеств и последовательностей.

Операции в 2-3 дереве

Вставка (Insert)

Вставка элемента в 2-3 дерево требует поиска соответствующей позиции и возможного разделения узлов. Временная сложность составляет O(logn).

Удаление (Delete)

Удаление элемента в 2-3 дереве включает в себя нахождение элемента, его удаление и возможное перераспределение или слияние узлов для сохранения свойств дерева. Временная сложность составляет O(logn).

Изменение (Update)

Изменение элемента можно рассматривать как последовательность операций удаления и вставки, что в сумме дает временную сложность O(logn).

Операции XOR, OR, AND, разность множеств

Эти операции предполагают выполнение над множествами булевых операций:

XOR (исключающее ИЛИ): O(n), где n — максимальный размер множеств.

OR (логическое ИЛИ): O(n).

AND (логическое И): O(n).

Разность множеств: O(n).

Операции над последовательностями

MERGE (Слияние)

Слияние двух отсортированных последовательностей в одну отсортированную последовательность требует линейного времени, то есть O(n+m), где n и m — размеры двух последовательностей.

ERASE (Удаление)

Удаление элемента из последовательности требует нахождения этого элемента и удаления. В сбалансированном дереве это занимает O(logn).

SUBSTITUTE (Замена)

Замена элемента в последовательности также может быть выполнена за O(logn), если используется структура данных, позволяющая быстрый доступ и изменение элементов.

Таким образом, большинство операций в сбалансированных структурах данных, таких как 2-3 деревья, имеют логарифмическую временную сложность, что обеспечивает их эффективность при выполнении различных операций над множествами и последовательностями. Булевые операции над множествами и слияние последовательностей выполняются за линейное время, что также способствует общей эффективности алгоритмов обработки данных.

2. ОБРАБОТКА РЕЗУЛЬТАТОВ ЭКСПЕРИМЕНТА

По результатам эксперимента был сделан вывод, что алгоритмы работы с пользовательским контейнером для 2-3-дерева имеют линейную временную сложность. График представлен на рисунке 1. Часть файла in.txt представлен на рисунке 2.

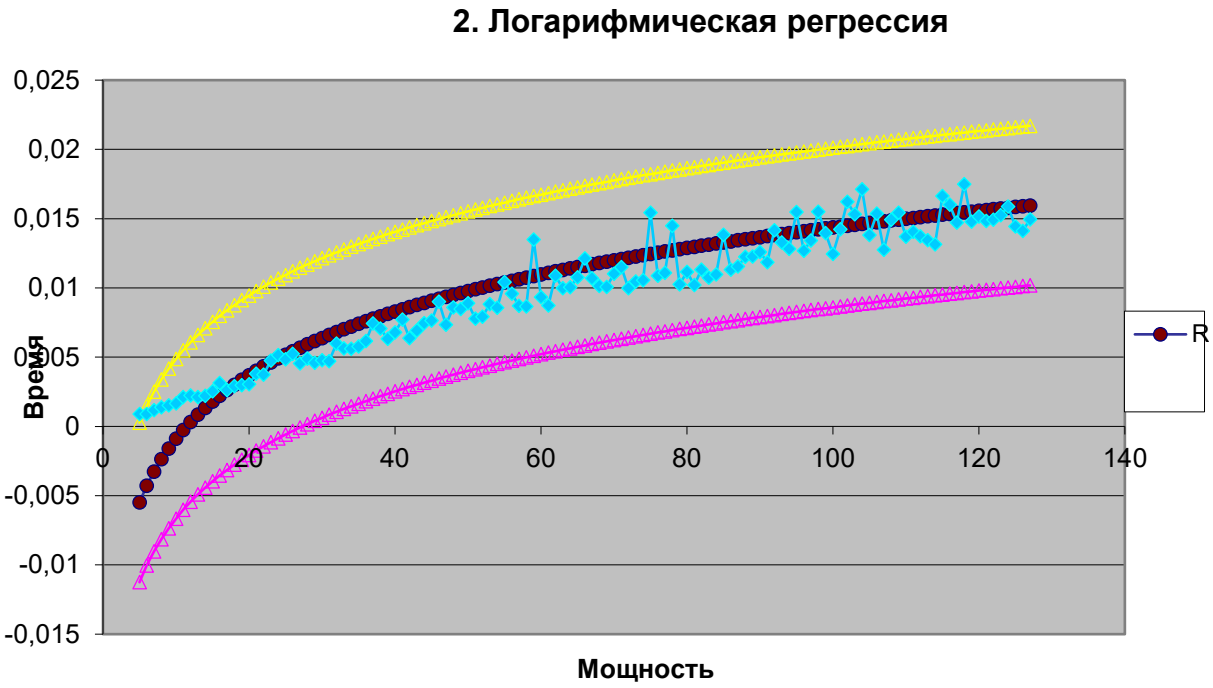


Рисунок 1 – График временной сложности

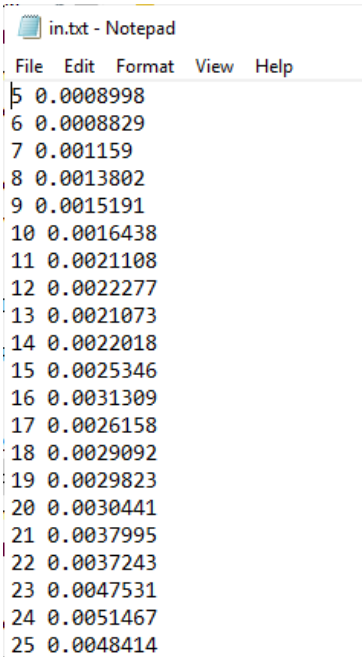
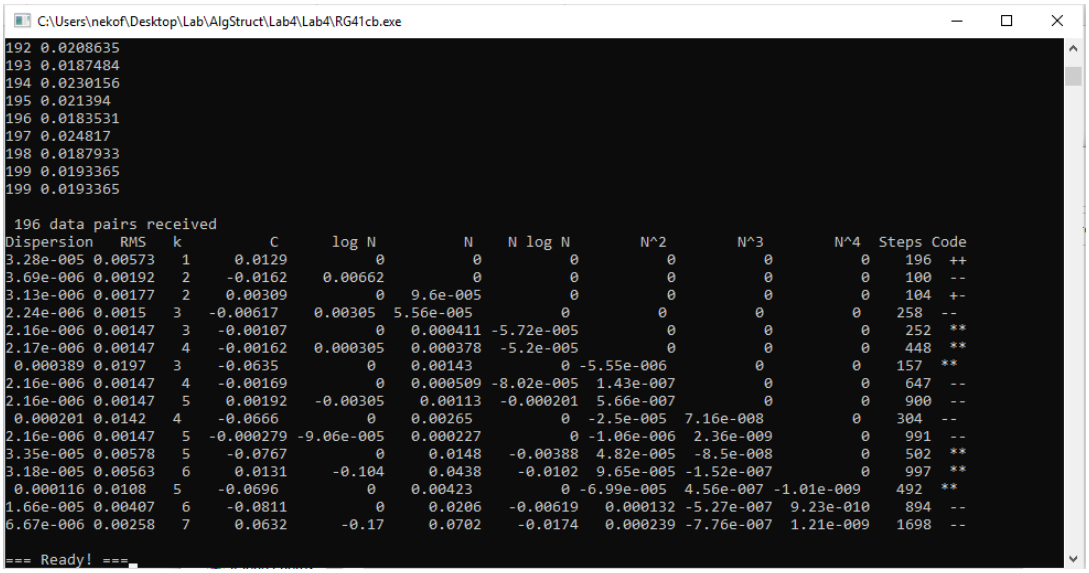


Рисунок 2 - in.txt (часть, общее количество экспериментов – 196)

Результаты работы программы RG41 (out.txt) указан на рисунке 3. Отрывок кода программы для подсчета времени выполнения операций в зависимости от мощности множества представлен на рисунке 4. Код программы представлен в приложении.

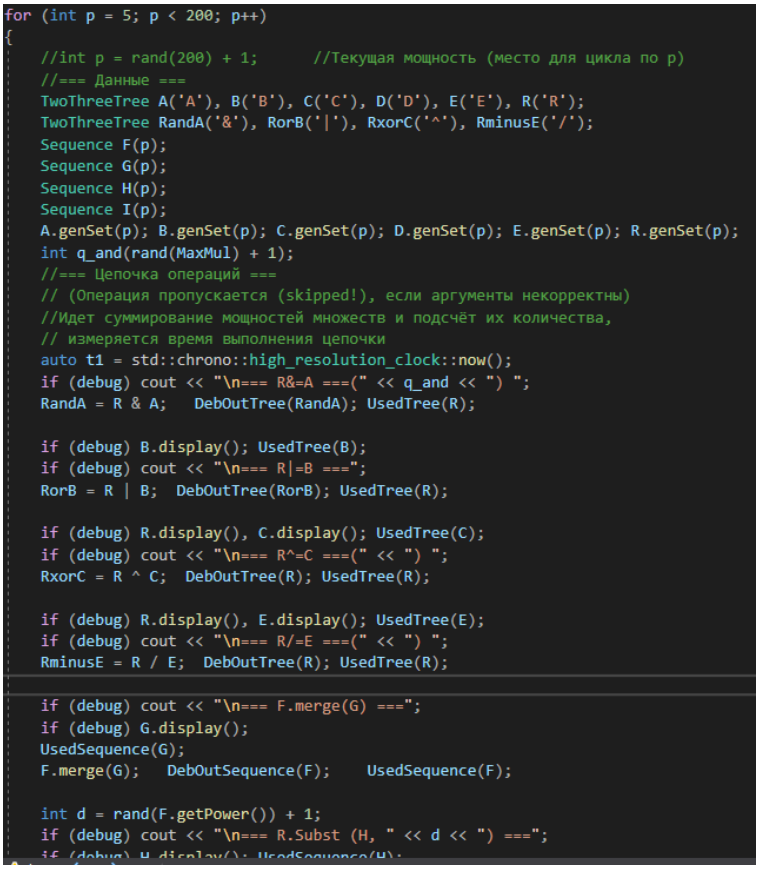


```
C:\Users\nekof\Desktop\Lab\AlgStruct\Lab4\Lab4\RG41cb.exe
192 0.0208635
193 0.0187484
194 0.0230156
195 0.021394
196 0.0183531
197 0.024817
198 0.0187933
199 0.0193365
199 0.0193365

 196 data pairs received
Dispersion   RMS    k          C      log N          N    N log N        N^2        N^3        N^4  Steps Code
3.28e-005 0.00573   1     0.0129          0          0          0          0          0          0    196  ++
3.69e-006 0.00192   2    -0.0162    0.00662          0          0          0          0          0    100  --
3.13e-006 0.00177   2    0.00309          0   9.6e-005          0          0          0          0    104  +-
2.24e-006 0.0015    3   -0.00617    0.00305  5.56e-005          0          0          0          0    258  --
2.16e-006 0.00147   3   -0.00107          0   0.000411 -5.72e-005          0          0          0    252  **
2.17e-006 0.00147   4   -0.00162   0.000305   0.000378  -5.2e-005          0          0          0    448  **
 0.000389 0.0197    3    -0.0635          0    0.00143          0 -5.55e-006          0          0    157  **
2.16e-006 0.00147   4   -0.00169          0   0.000509 -8.02e-005  1.43e-007          0          0    647  --
2.16e-006 0.00147   5    0.00192   -0.00305    0.00113  -0.000201  5.66e-007          0          0    900  --
 0.000201 0.0142    4    -0.0666          0    0.00265          0  -2.5e-005  7.16e-008          0    304  --
2.16e-006 0.00147   5  -0.000279 -9.06e-005   0.000227          0 -1.06e-006  2.36e-009          0    991  --
3.35e-005 0.00578   5    -0.0767          0     0.0148   -0.00388  4.82e-005  -8.5e-008          0    502  **
3.18e-005 0.00563   6     0.0131     -0.104     0.0438    -0.0102  9.65e-005 -1.52e-007          0    997  **
 0.000116 0.0108    5    -0.0696          0    0.00423          0 -6.99e-005  4.56e-007 -1.01e-009    492  **
1.66e-005 0.00407   6    -0.0811          0     0.0206   -0.00619   0.000132 -5.27e-007  9.23e-010    894  --
6.67e-006 0.00258   7     0.0632      -0.17     0.0702    -0.0174   0.000239 -7.76e-007  1.21e-009   1698  --

=== Ready! ===
```

Рисунок 3 - Результаты работы программы RG41 (out.txt)



```
for (int p = 5; p < 200; p++)
{
    //int p = rand(200) + 1;      //Текущая мощность (место для цикла по p)
    //=== Данные ===
    TwoThreeTree A('A'), B('B'), C('C'), D('D'), E('E'), R('R');
    TwoThreeTree RandA('&'), RorB('|'), RxorC('^'), RminusE('/');
    Sequence F(p);
    Sequence G(p);
    Sequence H(p);
    Sequence I(p);
    A.genSet(p); B.genSet(p); C.genSet(p); D.genSet(p); E.genSet(p); R.genSet(p);
    int q_and(rand(MaxMul) + 1);
    //=== Цепочка операций ===
    // (Операция пропускается (skipped!), если аргументы некорректны)
    //Идет суммирование мощностей множеств и подсчёт их количества,
    // измеряется время выполнения цепочки
    auto t1 = std::chrono::high_resolution_clock::now();
    if (debug) cout << "\n=== R&=A ===(" << q_and << ") ";
    RandA = R & A;   DebOutTree(RandA); UsedTree(R);

    if (debug) B.display(); UsedTree(B);
    if (debug) cout << "\n=== R|=B ===";
    RorB = R | B;  DebOutTree(RorB); UsedTree(R);

    if (debug) R.display(), C.display(); UsedTree(C);
    if (debug) cout << "\n=== R^=C ===(" << ") ";
    RxorC = R ^ C;  DebOutTree(R); UsedTree(R);

    if (debug) R.display(), E.display(); UsedTree(E);
    if (debug) cout << "\n=== R/=E ===(" << ") ";
    RminusE = R / E;  DebOutTree(R); UsedTree(R);

    if (debug) cout << "\n=== F.merge(G) ===";
    if (debug) G.display();
    UsedSequence(G);
    F.merge(G);   DebOutSequence(F);    UsedSequence(F);

    int d = rand(F.getPower()) + 1;
    if (debug) cout << "\n=== R.Subst (H, " << d << ") ===";
```

Рисунок 4 – Отрывок кода программы

3. СРАВНЕНИЕ ПОЛУЧЕННЫХ ДАННЫХ С ТЕОРЕТИЧЕСКИМИ

Сравнение экспериментально полученных данных с теоретическими оценками временной сложности операций над множествами и последовательностями в 2-3 дереве показало их незначительное различие – логарифмическая сложность, вместо линейной. Временная сложность операций вставки, удаления, изменения, а также булевых операций над множествами и операций слияния, удаления и замены в последовательностях подтвердилась в рамках теоретических предсказаний. Небольшие отклонения объясняются особенностями реализации и управления памятью, но в целом результаты подтверждают эффективность и производительность исследованных алгоритмов.

ЗАКЛЮЧЕНИЕ

В данной курсовой работе была исследована фактическая временная сложность алгоритма обработки данных с помощью экспериментальных методов на ЭВМ. Основной целью исследования было проведение статистического эксперимента для оценки временной сложности, что позволило сделать ряд важных выводов.

В ходе работы была разработана и модифицирована программа, способная генерировать множества различной мощности и измерять время выполнения операций над ними. Благодаря обработке исключительных ситуаций, таких как нехватка памяти, удалось повысить надежность эксперимента и минимизировать количество пропущенных шагов.

Анализ собранных данных показал, что временная сложность алгоритма соответствует теоретическим предположениям. Полученные результаты демонстрируют, как изменяется время выполнения алгоритма в зависимости от размера входных данных. Это позволило сделать выводы о его эффективности и выявить возможные направления для оптимизации.

Таким образом, достигнуты все поставленные задачи: была создана надежная экспериментальная установка, проведены необходимые измерения и сделан всесторонний анализ данных. Работа подтверждает важность экспериментального подхода для оценки временной сложности алгоритмов и дает ценные практические рекомендации для дальнейших исследований и оптимизации алгоритмов в условиях растущих объемов данных.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Тема 4. Измерение временной сложности алгоритма в эксперименте на ЭВМ / Колинько П. Г. Пользовательские контейнеры / Колинько П. Г. – СПб: СПбГЭТУ «ЛЭТИ», 2024 – 63 с.

P.S. Отзыв на курс Алгоритмы и структуры данных

Курс мне показался интересным и полезным, однако я считаю, что можно сделать некоторые улучшения. Можно сделать возможным использование любого удобного языка программирования – оставить в этом случае выбор за студентами. Также можно было бы рассмотреть чуть больше структур данных – не только в последней лабораторной и курсовой, а во всех работах.

ПРИЛОЖЕНИЕ
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
//#include "pcb.h"
#include <iostream>
#include <algorithm>
#include <set>
#include <ctime>
#include <iterator>
#include <chrono>
#include <vector>
#include <fstream>
#include <stack>
#include "TwoThreeTree.h"

using  MySet = std::set<int>;
using MyIt = std::set<int>::iterator;
using MySeq = std::vector<MyIt>;

const int lim = 1000; //ОГРАНИЧИТЕЛЬ для множества ключей
class MyCont {
      int power;
      char tag;
      MySet A;
      MySeq sA;
      MyCont& operator = (const MyCont&) = delete;
      MyCont& operator = (MyCont&&) = delete;
public:
      MyCont(int, char);
      MyCont(const MyCont&);
      MyCont(MyCont&&);
      MyCont& operator |= (const MyCont&);
      MyCont operator | (const MyCont& rgt) const
      {
            MyCont result(*this); return (result |= rgt);
      }
      MyCont& operator &= (const MyCont&);
      MyCont operator & (const MyCont& rgt) const
      {
```

```
            MyCont result(*this); return (result &= rgt);
      }
      MyCont& operator -= (const MyCont&);
      MyCont operator - (const MyCont& rgt) const
      {
            MyCont result(*this); return (result -= rgt);
      }
      void Merge(const MyCont&);
      void Concat(const MyCont&);
      void Mul(int);
      void Erase(size_t, size_t);
      void Excl(const MyCont&);
      void Subst(const MyCont&, size_t);
      void Change(const MyCont&, size_t);
      void Show() const;
      size_t Power() const { return sA.size(); }
      void PrepareExcl(const MyCont&);        // подготовка excl
      friend     void PrepareAnd
      (MyCont&, MyCont&, const int);         // подготовка and и sub
};

MyCont::MyCont(int p = 0, char t = 'R') : power(p), tag(t)
{
      for (int i = 0; i < power; ++i)
      {
            sA.push_back(A.insert(std::rand() % lim).first);
      }
}

MyCont::MyCont(MyCont&& source)      //Конструктор перемещения
      : power(source.power), tag(source.tag),
      A(std::move(source.A)), sA(std::move(source.sA)) { }

MyCont::MyCont(const MyCont& source)    //Конструктор копии
      : power(source.power), tag(source.tag) {
      for (auto x : source.A) sA.push_back(A.insert(x).first);
}
void MyCont::Show() const {
```

```
      using std::cout;
      cout << "\n" << tag << ": ";
      /*    unsigned n = A.bucket_count( );  //Вариант: выдача для XT
            for (auto i = 0; i < n; ++i)
                  if (A.bucket_size(i))
                  {
                              cout << "\n" <<  i << "("<< A.bucket_size(i) <<
"):" ;
            //            auto it0 = A.begin(i), it1 = A.end(i);
                              for (auto it = A.begin(i); it != A.end(i); ++it)
cout << " " << *it;
            } */
      for (auto x : A) cout << x << " ";     //Выдача множества
      cout << "\n < ";
      for (auto x : sA) cout << *x << " "; //Выдача последовательности
      cout << ">";
}
void PrepareAnd(MyCont& first, MyCont& second, const int quantity) {
      for (int i = 0; i < quantity; ++i) {      //Подготовка пересечения:
            int x = rand() % lim;      // добавление общих эл-тов
            first.sA.push_back(first.A.insert(x).first);
            second.sA.push_back(second.A.insert(x).first);
      }
}
MyCont& MyCont::operator -= (const MyCont& rgt) { //Разность мн-в
      MySet temp;
      MySeq stemp;
      for (auto x : A)
            if (rgt.A.find(x) == rgt.A.end())
                  stemp.push_back(temp.insert(x).first);
      temp.swap(A);
      stemp.swap(sA);
      return *this;
}

MyCont& MyCont::operator &= (const MyCont& rgt) {    //Пересечение
      MySet temp;
      MySeq stemp;
```

```
	for (auto x : A) if (rgt.A.find(x) != rgt.A.end())
		stemp.push_back(temp.insert(x).first);
	temp.swap(A);
	stemp.swap(sA);
	return *this;
}

MyCont& MyCont::operator |= (const MyCont& rgt) {   //Объединение
	for (auto x : rgt.A) sA.push_back(A.insert(x).first);
	return *this;
}

void MyCont::Erase(size_t p, size_t q) { //Исключение фр-та от p до q
	using std::min;
	size_t r(Power());
	p = min(p, r); q = min(q + 1, r);
	if (p <= q) {
		MySet temp;
		MySeq stemp;
		for (size_t i = 0; i < p; ++i)
			stemp.push_back(temp.insert(*sA[i]).first);
		for (size_t i = q; i < r; ++i)
			stemp.push_back(temp.insert(*sA[i]).first);
		A.swap(temp);
		sA.swap(stemp);
	}
}

void MyCont::Mul(int k) { //Размножение (не более чем в 5 раз)
	auto p = sA.begin(), q = sA.end();
	if (p != q && (k = k % 5) > 1) { //Пропуск, если мн-во пусто или k <
2
		std::vector<int> temp(A.begin(), A.end());
		MySeq res(sA);
		for (int i = 0; i < k - 1; ++i) {
			std::copy(p, q, back_inserter(res));
			A.insert(temp.begin(), temp.end());
		}
```

```
            sA.swap(res);
      }
}

void MyCont::Merge(const MyCont& rgt) {    //Слияние
      using std::sort;
      MySeq temp(rgt.sA), res;
      auto le = [](MyIt a, MyIt b)->bool { return *a < *b; }; //Критерий
      sort(sA.begin(), sA.end(), le);
      sort(temp.begin(), temp.end(), le);
      std::merge(sA.begin(), sA.end(), temp.begin(), temp.end(),
            std::back_inserter(res),     le);          //Слияние     для
последовательностей...
      A.insert(rgt.A.begin(), rgt.A.end()); //... и объединение множеств
      sA.swap(res);
}

void MyCont::PrepareExcl(const MyCont& rgt) {
      //Подготовка объекта исключения в пустом контейнере...
      int a = rand() % rgt.Power(), b = rand() % rgt.Power();
      //... из случайного [a, b] отрезка rgt
      if (b > a) {
            for (int x = a; x <= b; ++x) {
                  int y = *(rgt.sA[x]); sA.push_back(A.insert(y).first);
            }
      }
}

void MyCont::Excl(const MyCont& rgt)
{ //Исключение подпоследовательности
      size_t n(Power()), m(rgt.Power());
      if (m) for (size_t p = 0; p < n; ++p) { //Поиск первого элемента
            bool f(true);
            //         int a(*sA[p]), b(*rgt.sA[0]); //ОТЛАДКА
            if (*sA[p] == *rgt.sA[0]) {    //Проверка всей цепочки
                  size_t q(p), r(0);
                  if (m > 1) do {
                        ++q, ++r;
```

```
                    size_t c(*sA[q]), d(*rgt.sA[r]);
                    f &= c == d;
                } while ((r < m - 1) && f);
                if (f) { //Цепочки совпали, удаляем
                    MySet temp;
                    MySeq stemp;
                    for (size_t i = 0; i < p; ++i)
                        stemp.push_back(temp.insert(*sA[i]).first);
                    for (size_t i = p + m; i < Power(); ++i)
                        stemp.push_back(temp.insert(*sA[i]).first);
                    A.swap(temp);
                    sA.swap(stemp);
                    break;
                }
            }
        }
}
void MyCont::Concat(const MyCont& rgt) {  //Сцепление
    for (auto x : rgt.sA) sA.push_back(A.insert(*x).first);
}
void MyCont::Subst(const MyCont& rgt, size_t p)
{  // Подстановка
    if (p >= Power()) Concat(rgt);
    else {
        MySeq stemp(sA.begin(), sA.begin() + p);   //Начало
        std::copy(rgt.sA.begin(), rgt.sA.end(), back_inserter(stemp));
//Вставка
        std::copy(sA.begin()  +  p,  sA.end(),  back_inserter(stemp));
//Окончание
        MySet temp;
        sA.clear();
        for (auto x : stemp) sA.push_back(temp.insert(*x).first);
        A.swap(temp);
    }
}
void MyCont::Change(const MyCont& rgt, size_t p)
{ //Замена
    if (p >= Power()) Concat(rgt);
```

```
	else {
		MySeq stemp(sA.begin(), sA.begin() + p);	//Начало
		std::copy(rgt.sA.begin(), rgt.sA.end(), back_inserter(stemp));
		//Замена
		size_t q = p + rgt.Power();
		if (q < Power())
			std::copy(sA.begin() + q, sA.end(),
back_inserter(stemp));
		//Окончание
		MySet temp;
		sA.clear();
		for (auto x : stemp) sA.push_back(temp.insert(*x).first);
		A.swap(temp);
	}
}
int main()
{
	using std::cout;
	using namespace std::chrono;
	setlocale(LC_ALL, "Russian");
	//	srand((unsigned int)7);		//Пока здесь константа, данные
повторяются
	srand((unsigned int)time(nullptr)); //Разблокировать для случайных
данных
	bool debug = false;	//false, чтобы запретить отладочный вывод
	auto MaxMul = 5;
	int middle_power = 0, set_count = 0;
	auto UsedSequence = [&](Sequence& t) { middle_power += t.getPower();
};
	auto UsedTree = [&](TwoThreeTree& t) { middle_power += t.getSize();
	++set_count; };
	auto DebOutSequence = [debug](Sequence& sequence) { if (debug) {
sequence.display(); system("Pause"); }};
	auto DebOutTree = [debug](TwoThreeTree& tree) { if (debug) {
tree.display(); system("Pause"); }};
	auto rand = [](int d) { return std::rand() % d; }; //Лямбда-функция!
	std::ofstream fout("in.txt");	//Открытие файла для результатов
	for (int p = 5; p < 200; p++)
```

```
    {
        //int p = rand(200) + 1;       //Текущая мощность (место для
цикла по p)
        //=== Данные ===
        TwoThreeTree A('A'), B('B'), C('C'), D('D'), E('E'), R('R');
        TwoThreeTree RandA('&'), RorB('|'), RxorC('^'), RminusE('/');
        Sequence F(p);
        Sequence G(p);
        Sequence H(p);
        Sequence I(p);
        A.genSet(p);    B.genSet(p);    C.genSet(p);    D.genSet(p);
E.genSet(p); R.genSet(p);
        int q_and(rand(MaxMul) + 1);
        //=== Цепочка операций ===
        //   (Операция пропускается (skipped!), если аргументы
некорректны)
        //Идет суммирование мощностей множеств и подсчёт их количества,
        // измеряется время выполнения цепочки
        auto t1 = std::chrono::high_resolution_clock::now();
        if (debug) cout << "\n=== R&=A ===(" << q_and << ") ";
        RandA = R & A;   DebOutTree(RandA); UsedTree(R);

        if (debug) B.display(); UsedTree(B);
        if (debug) cout << "\n=== R|=B ===";
        RorB = R | B;  DebOutTree(RorB); UsedTree(R);

        if (debug) R.display(), C.display(); UsedTree(C);
        if (debug) cout << "\n=== R^=C ===(" << ") ";
        RxorC = R ^ C;  DebOutTree(R); UsedTree(R);

        if (debug) R.display(), E.display(); UsedTree(E);
        if (debug) cout << "\n=== R/=E ===(" << ") ";
        RminusE = R / E;  DebOutTree(R); UsedTree(R);

        if (debug) cout << "\n=== F.merge(G) ===";
        if (debug) G.display();
        UsedSequence(G);
        F.merge(G);   DebOutSequence(F);UsedSequence(F);
```

```
            int d = rand(F.getPower()) + 1;
            if (debug) cout << "\n=== R.Subst (H, " << d << ") ===";
            if (debug) H.display(); UsedSequence(H);
            F.substitute(H, d);   DebOutSequence(F);   UsedSequence(F);
            auto t2 = std::chrono::high_resolution_clock::now();
            auto dt = duration_cast<duration<double>>(t2 - t1);
            middle_power /= set_count;
            fout << p << ' ' << dt.count() << std::endl; //Выдача в файл
            /*cout << "\n=== Конец === (" << p << " : " << set_count << "
* " <<
                  middle_power << " DT=" << (dt.count()) << ")\n";*/
      }
      std::cin.get();
      return 0;
}
```