Минобрнауки России

Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им В. И. Ульянова (Ленина)»

Факультет компьютерных технологий и информатики

Кафедра вычислительной техники

Зачётная работа № 1

по дисциплине «Алгоритмы и структуры данных»

на тему «Множества в памяти ЭВМ»

Выполнили студенты группы 3312:

Зайцев И.С.

Мохно Д.А.

Принял: старший преподаватель Колинько П.Г.

Санкт-Петербург

2024

Оглавление

1.Цель работы

Сравнительное исследование четырёх способов хранения множеств в памяти ЭВМ.

2. Задание

Множество русских букв (кириллица), содержащее все буквы множеств *A* и *B*, за исключением букв, содержащихся в *C*, а также все буквы из *D*

3. Формула для вычисления пятого множества

Формализация задания: E = (A∪B∪D)\C

Программа генерирует четыре случайные строки, состоящие из строчных букв русского алфавита, и выполняет операции объединения и разности этих строк с использованием четырех различных методов: односвязные списки, массивы, битовые векторы и 64-битные целые числа (машинные слова). Она измеряет и сравнивает время выполнения каждой из операций, а затем выводит результаты, демонстрируя эффективность каждого метода в обработке строк. В алфавите символ “ё” идёт 34, 33 символ – “è”. При обработке массивов 33 символ пропускается. Так же программа должна быть мультиплатформенная (windows\unix системы).

4. Контрольные тесты

Ниже представлены примеры работы программы (рис. 1 – 4). Используя генератор подмножеств заданной мощности, формируем подмножества с случайной мощностью от 1 до 33 и считаем время выполнения операций, произошедших за период обработки подмножеств в различных формах их представления. Результат выводится в консоль.

```
Введите мощность множеств:
4
a: уждь
b: двоц
c: кжцв
d: оясз
array                удьоясз,        за время:       1.6 наносекунд
linked list          ясзоудь,        за время:       3 наносекунд
universe             дзосуья,        за время:       1.3 наносекунд
machine word         дзосуья,        за время:       0.1 наносекунд
```

Рис. 1

```
Введите мощность множеств:
15
a: ьшжхтзеъщяынёиу
b: ъжшытуьдэювейоц
c: ьёыифрзюмтшбщнц
d: чгёфжэкувиеъбзь
array                жхеъяудэвйочгк, за время:       4.3 наносекунд
linked list          чгкдэвйожхеъяу, за время:       12.5 наносекунд
universe             вгдежйкоухчъэя, за время:       1.9 наносекунд
machine word         вгдежйкоухчъэя, за время:       0.1 наносекунд
```

Рис. 2

```
Введите мощность множеств:
7
a: бшцффыё
b: чргерцф
c: юцождчк
d: ьтнковц
array                бшфыёргеьтнв,   за время:       2 наносекунд
linked list          ьтнвргебшфыё,   за время:       3.4 наносекунд
universe             бвгенртфшыьё,   за время:       1.5 наносекунд
machine word         бвгенртфшыьё,   за время:       0.1 наносекунд
```

Рис. 3

```
Введите мощность множеств:
19
a: епжкювоаэшьзёяйлчуы
b: дшновухъзтасфпяжьлы
c: илбфосрнюшгжцяэавщз
d: кщрзлаёюуъхтфйжпгдв
array                 епкьёйчуыдхът,  за время:       4.2 наносекунд
linked list           дхътепкьёйчуы,  за время:       7.9 наносекунд
universe              дейкптухчъыьё,  за время:       1.5 наносекунд
machine word          дейкптухчъыьё,  за время:       0.1 наносекунд
```

Рис. 4

5. Временная сложность

Таблица 1. Способы представления и временная сложность обработки

Способ представления	Временная сложность	
	Ожидаемая	Фактическая
Массив символов	$O(n^2)$	$O(n^2)$
Список		$O(n^2)$
Универсум	O(|U|)	O(|U|)
Машинное слово	O(1)	O(1)

Пояснения:

Для множества, представленного набором элементов (массив символов или список), двуместная операция требует проверки всех комбинаций элементов множества, которых для множеств мощностью n будет $O(n^2)$. Общая сложность получится такой же, если вычисление заданного выражения свести к последовательности двуместных операций.

Для множеств, представленных отображением на универсум ожидаемое количество шагов двуместной операции равно мощности универсума.

Поскольку вычисление заданного выражения выполнено как последовательность двуместных операций, каждая из которых реализована двойным циклом по мощности множества, фактическая временная сложность алгоритма вычислений совпала с ожидаемой. Для отображения на универсум, мощность которого фиксирована, сложность вычисления можно считать константной.

Использование машинных слов (например, 64-битных чисел) для представления множеств позволяет каждой букве соответствовать одному биту. Операции объединения и разности множеств сводятся к битовым операциям (ИЛИ, И НЕ) над машинными словами. Это позволяет достичь константного времени выполнения O(1), так как операция выполняется за один шаг. В коде строки преобразуются в машинные слова, а операции выполняются за счет битовых операций, что делает их более эффективными по сравнению с классическими методами, где перебираются элементы.

6. Результат измерения времени обработки для каждого из способов

Таблица 2. Результаты измерения времени обработки

Мощность множеств	Время (в наносекундах), требуемое при обработке множеств различными способами представления			
	Массив символов	Список	Массив битов	Машинное слово
2	0.9	1.8	0.7	0.1
4	1.7	1.4	0.8	0.1
6	1.9	5.3	1.4	0.1
8	3	4.5	1.5	0.1
10	3	5.8	1.5	0.1
12	3	4.8	1.4	0.1
14	3.6	4.9	1.4	0.1
16	4	11.3	1.1	0.1
18	4.7	12.1	1.4	0.1
20	5.3	17.7	1.4	0.1
22	5.3	13.8	1.5	0.1
24	5.5	12.9	1.4	0.1
26	5.3	14.6	1.7	0.1

28	5.8	14.4	1.4	0.1
30	4.1	5.9	1.1	0.1
32	5.9	9.3	1.3	0.1

При представлении множеств в виде массива символов и списка заметно, что время обработки множеств с увеличением мощности также увеличивается. Для универсума и машинного слова время обработки практически неизменно, т.е. не зависит от размера входа.

7. Выводы

В результате работы было установлено, что наиболее быстрым методом представления множеств для их обработки является использование машинного слова. Этот способ рекомендуется применять, когда существует простая функция для отображения элемента множества в соответствующий номер бита, и размер универсума не превышает разрядности слова. Указанные ранее аспекты, касающиеся представления множеств в виде машинного слова, также применимы к битовым векторам, однако в этом случае мощность универсума также должна быть ограниченной.

Наиболее медленной оказалась обработка односвязных списков. Этот метод следует использовать, когда мощность создаваемого множества неизвестна, и выделение памяти под всё множество заранее не представляется возможным.

Представление множества в виде массива рекомендуется использовать, если возможно с достаточной точностью определить размер массива, а мощность универсума слишком велика для применения битового вектора или машинного слова.

8. Список используемых источников

1. П. Г. Колинько Методические указания по дисциплине «Алгоритмы и структуры данных, часть 1» Вып. 2408, 2024 год

2. Курс на платформе “Stepik” Основы программирования на C/C++ https://stepik.org/course/55918/promo

9. Приложение. Текст программы

```
#include <iostream>
#include <cwchar>    // Для работы с широкими строками и функциями wcscpy,
wcslen
#include <chrono>

using namespace std;

const short UNIVERSE_SIZE = 34; // Количество букв в русском алфавите (буква ё
находится на, а+33 месте, на, а+32 почему-то находится è

#pragma region list

struct Word {
    wchar_t letter = L'\0';
    Word *next = nullptr;
};

// Функция для проверки, содержится ли символ в списке
bool contains(const Word* list, wchar_t symbol) {
    while (list) {
        if (list->letter == symbol) {
            return true;
        }
        list = list->next;
    }
    return false;
}

// Функция для добавления символа в список (если его нет)
void add_Word(Word*& head, wchar_t symbol) {
    if (!contains(head, symbol)) {
        Word* new_Word = new Word;
        new_Word->letter = symbol;
        new_Word->next = head;
        head = new_Word;
    }
}

// Функция для преобразования строки в список
Word* string_to_list(const wchar_t * str) {
    Word* head = nullptr;
    size_t len = wcslen(str);

    for (size_t i = 0; i < len; i += 1) {
        wchar_t symbol = str[i];
        add_Word(head, symbol);
    }
    return head;
}

void OR(const Word* src, Word*& dst) {
    while (src) {
        // If src->letter is not in dst, add it to dst
        if (!contains(dst, src->letter)) {
```

```
            add_Word(dst, src->letter);
        }
        src = src->next;
    }
}


void DIFF(Word** A, const Word* B) {
    Word* prev = nullptr, *cur = *A;

    while (cur) {
        // Если буква A находится в B, удаляем её из A
        if (contains(B, cur->letter)) {
            if (prev) {
                prev->next = cur->next; // Пропускаем текущий узел
                delete cur;             // Удаляем текущий узел
                cur = prev->next;       // Переходим к следующему узлу
            } else {
                // Обновляем голову, если удаляем первый узел
                Word* temp = cur;       // Сохраняем текущий узел для удаления
                *A = cur->next;         // Перемещаем голову на следующий узел
                cur = *A;               // Обновляем текущий до головы
                delete temp;            // Удаляем старую голову
            }
        } else {
            prev = cur;                 // Перемещаем prev на текущий узел
            cur = cur->next;            // Переходим к следующему узлу
        }
    }
}

// Функция для записи списка
void convert(const Word* list, wchar_t* arr) {
    int i = 0;
    while (list) {
        arr[i] =  list->letter;
        i++;
        list = list->next;
    }
    arr[i] = '\0';
}

#pragma endregion list

#pragma region arrs

void OR(const wchar_t *src, wchar_t *dst) {
    size_t len_dst = wcslen(dst);
    for (int i = 0; src[i] != '\0'; i++) {
        // note: проверка на то, чтобы символ не уже не содержался в строке, и
на то, чтобы не содержался в строке C
        if (dst && !wcschr(dst, src[i])) {
            dst[len_dst] = src[i];
            len_dst++;
        }
    }
    dst[len_dst] = '\0';
}

void DIFF(const wchar_t *A, const wchar_t *B, wchar_t *dst) {
    int i, j;
    for (i = 0, j = 0; i <= wcslen(A); i++) {
        if (!wcschr(B, A[i])) {
```

```
            dst[j] = A[i];
            j++;
        }
    }
    dst[j] = '\0';
}

#pragma endregion arrs

#pragma region universe

// Функция для отображения строки на вектор битов
void mapToUniverse(const wchar_t *str, bool *bitVector)
{
    // Инициализация вектора битов
    for (int i = 0; i < UNIVERSE_SIZE; ++i)
        bitVector[i] = false;

    // Устанавливаем бит для каждой буквы из строки
    for (int i = 0; str[i]; ++i)
    {
        // Индекс буквы относительно 'a'
        int index = str[i] - L'a';
        if (index >= 0 && index < UNIVERSE_SIZE)
        {
            bitVector[index] = true;
        }
    }
}

// Функция для объединения множеств (логическое ИЛИ)
void OR(const bool *A, const bool *B, bool *result)
{
    for (int i = 0; i < UNIVERSE_SIZE; ++i)
        result[i] = A[i] || B[i];
}

// Функция для разности множеств (A \ B)
void DIFF(const bool *A, const bool *B, bool *result)
{
    for (int i = 0; i < UNIVERSE_SIZE; ++i)
        result[i] = A[i] && !B[i];
}

// Функция для вывода множества (вектора битов) как строки
void convert(const bool *bitVector, wchar_t* res)
{
    int j = 0;
    for (int i = 0; i < UNIVERSE_SIZE; ++i)
    {
        if (bitVector[i]){
            res[j] = (wchar_t) (L'a' + i);
            j++;
        }
    }
    res[j] = '\0';
}

#pragma endregion universe

#pragma region machine word

// Функция отображения символа в позицию бита
```

```
int charToBitIndex(wchar_t ch) {
    if (ch >= L'a' && ch <= L'ё') {
        return ch - L'a'; // Индекс для русских букв
    }
    return -1; // Неизвестный символ
}

// Функция для отображения строки на машинное слово (64-битное число)
unsigned long long mapToWord(const wchar_t* str) {
    unsigned long long bitWord = 0;
    for (int i = 0; str[i]; ++i) {
        int bitIndex = charToBitIndex(str[i]);
        if (bitIndex != -1) {
            bitWord |= (1ULL << bitIndex); // Устанавливаем бит для символа
        }
    }
    return bitWord;
}

// Функция для вывода множества как строки символов
void convert(unsigned long long bitWord, wchar_t* res) {
    int j = 0;
    for (int i = 0; i < UNIVERSE_SIZE; ++i) {
        if (bitWord & (1ULL << i)) {
            res[j] = (wchar_t)(L'a' + i);
            j++;
        }
    }
    res[j] = '\0';
}

#pragma endregion machine word

void Randomizer(wchar_t *arr, size_t power) {
    const wchar_t first_lower = L'a'; // Начало строчных букв
    bool used[UNIVERSE_SIZE] = {false};  // Массив для отслеживания использован-
ных символов
    int i = 0;

    // Проверка, что power не больше размера вселенной возможных символов
    if (power >= UNIVERSE_SIZE) {
        power = UNIVERSE_SIZE-1;  // Ограничение на количество символов
    }

    while (i < power) {
        wchar_t new_char = (wchar_t)(first_lower + rand() % UNIVERSE_SIZE);

        // Проверяем, был ли символ использован и не равен ли он символу с раз-
ницей 32
        if (!used[new_char - first_lower] && (new_char - first_lower != 32)) {
            arr[i] = new_char;  // Если не был, добавляем его в массив
            used[new_char - first_lower] = true;  // Отмечаем как использованный
            i++;
        }
    }

    arr[i] = L'\0'; // Завершаем массив нулевым символом
}


int main() {
#pragma region formation
    setlocale(LC_ALL, "");
```

```
    const size_t len = 33;
    size_t power;
    wchar_t a[len], b[len], c[len], d[len];
    // получаем мощность множеств
    wcout << L"Введите мощность множеств:\n";
    cin >> power;
    // Инициализируем генератор случайных чисел
    srand(time(nullptr)); // time(nullptr)
    Randomizer(a, power);
    Randomizer(b, power);
    Randomizer(c, power);
    Randomizer(d, power);
    wcout << "a: " << a << endl;
    wcout << "b: " << b << endl;
    wcout << "c: " << c << endl;
    wcout << "d: " << d << endl;

    //const size_t size = wcslen(a) + wcslen(b) + wcslen(c) + 1;
    wchar_t e_arr[len] , e_list[len], e_uni[len], e_mw[len];

#pragma endregion formation

#pragma region list work

    // Преобразуем строки в линейные списки
    Word* list_a = string_to_list(a);
    Word* list_b = string_to_list(b);
    Word* list_c = string_to_list(c);
    Word* list_d = string_to_list(d);
    Word* list_e = nullptr;
    auto list_t1 = chrono::high_resolution_clock::now( );

    OR(list_a, list_e);
    OR(list_b, list_e);
    OR(list_d, list_e);


    DIFF(&list_e, list_c);

    auto list_t2 = chrono::high_resolution_clock::now( );
    auto list_time_res = chrono::duration_cast<chrono::duration<double, mi-
cro>>(list_t2 - list_t1).count();
    // Запись результата
    convert(list_e, e_list);
#pragma endregion list work

#pragma region arr work
    e_arr[0] = '\0';
    auto arr_t1 = chrono::high_resolution_clock::now( );

    OR(a,e_arr);
    OR(b, e_arr);
    OR(d, e_arr);
    DIFF(e_arr, c, e_arr);

    auto arr_t2 = chrono::high_resolution_clock::now( );
    auto arr_time_res = chrono::duration_cast<chrono::duration<double, mi-
cro>>(arr_t2 - arr_t1).count();
#pragma endregion arr work

#pragma region universe work
    bool bitA[UNIVERSE_SIZE], bitB[UNIVERSE_SIZE], bitC[UNIVERSE_SIZE],
bitD[UNIVERSE_SIZE];
```

```
    bool bitUnion[UNIVERSE_SIZE], bitResult[UNIVERSE_SIZE];

    // Отображаем строки на векторы битов
    mapToUniverse(a, bitA);
    mapToUniverse(b, bitB);
    mapToUniverse(c, bitC);
    mapToUniverse(d, bitD);

    auto uni_t1 = chrono::high_resolution_clock::now( );

    // (a U b U d)
    OR(bitA, bitB, bitUnion);
    OR(bitUnion, bitD, bitUnion);

    // (a U b U d) \ c
    DIFF(bitUnion, bitC, bitResult);

    auto uni_t2 = chrono::high_resolution_clock::now( );
    auto uni_time_res = chrono::duration_cast<chrono::duration<double, mi-
cro>>(uni_t2 - uni_t1).count();


    // Вывод результата
    convert(bitResult, e_uni);
#pragma endregion universe work

#pragma region machine word work
    // Отображаем строки на машинные слова
    unsigned long long wA = mapToWord(a);
    unsigned long long wB = mapToWord(b);
    unsigned long long wC = mapToWord(c);
    unsigned long long wD = mapToWord(d);

    auto mw_t1 = chrono::high_resolution_clock::now( );


    // Вычисляем E = (A U B U D) \ C
    unsigned long long wUnion = (wA | wB | wD); // (A U B U D)
    unsigned long long wResult = wUnion & ~wC;  // (A U B U D) \ C

    auto mw_t2 = chrono::high_resolution_clock::now( );
    auto mw_time_res = chrono::duration_cast<chrono::duration<double, mi-
cro>>(mw_t2 - mw_t1).count();


    // Вывод результата
    convert(wResult, e_mw);
#pragma endregion machine word work

#pragma region output results
    // Вывод результата
    wcout << "array\t\t\t"  << e_arr << L",\tза время:\t" << arr_time_res << L"
наносекунд" << endl;
    wcout << "linked list\t\t" << e_list << L",\tза время:\t" << list_time_res
<< L" наносекунд" << endl;
    wcout << "universe\t\t" << e_uni << L",\tза время:\t" << uni_time_res << L"
наносекунд" << endl;
    wcout << "machine word\t\t" << e_mw << L",\tза время:\t" << mw_time_res <<
L" наносекунд" << endl;
#pragma endregion output results

    return 0;
}
```