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:

Нищеглов С.В.

Авдеев Н. А.

Принял: старший преподаватель Колинько П. Г.

Санкт-Петербург

2024

1.Цель работы

Исследование четырёх способов хранения множеств в памяти ЭВМ.

2. Задание

Множество прописных латинских букв, содержащее все символы из множества A, за исключением символов, содержащихся в B или C, а также все символы множества D

3. Формула для вычисления пятого множества

Формализация задания: E = A ∧ (B ∨ C) ∨ D

4. Контрольные тесты

Ниже приведены примеры работы программы (рис. 1 – 4). С помощью генератора подмножеств заданного размера мы создаем подмножества с количеством элементов от 2 до 26 и измеряем время в тиках процессора, затраченное на обработку этих подмножеств в разных формах их представления. Результаты отображаются в консоли.

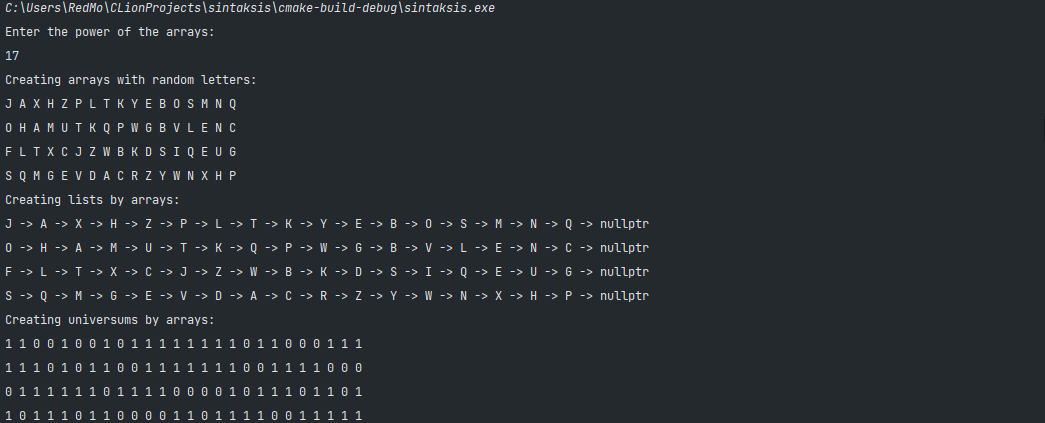
рис.1

рис.2

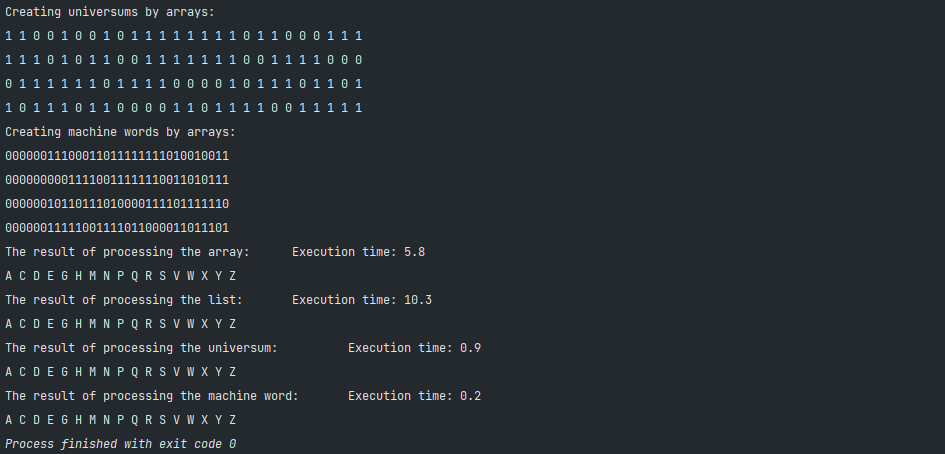


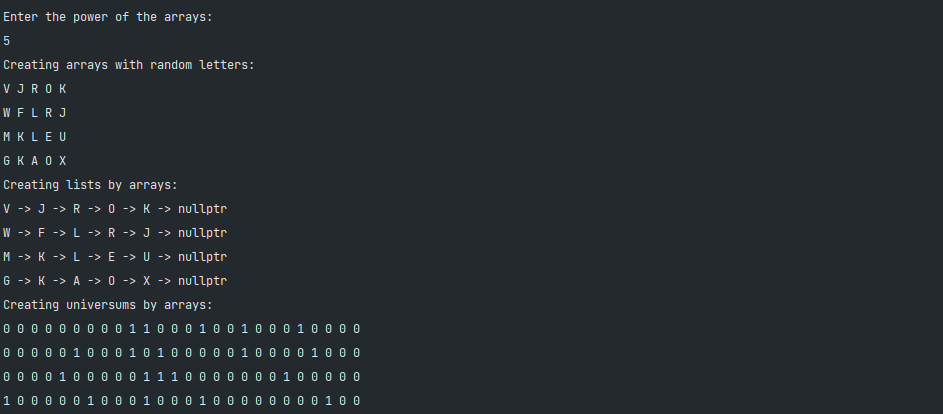
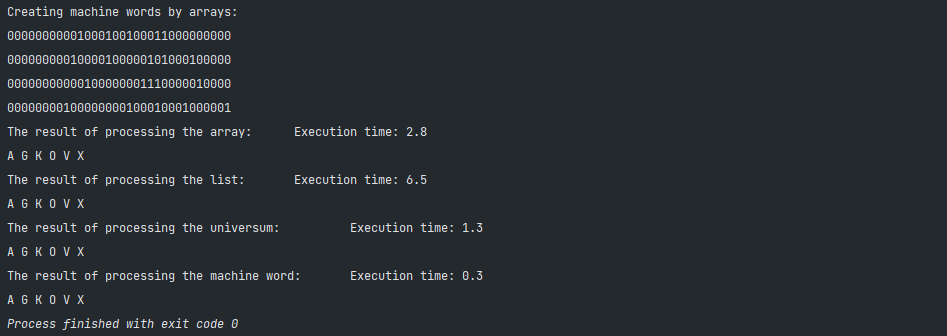
рис.3  


рис.4



5. Временная сложность

|  |  |  |
| --- | --- | --- |
| Способ представления | Временная сложность | |
| Ожидаемая | Фактическая |
| Массив символов | O() | O(n2) |
| Список | O(n2) |
| Универсум | O(|U|) | O(1) |
| Машинное слово | O(1) |

**Пояснения:**

Для множества, представленного набором элементов (в виде массива символов или списка), выполнение двуместной операции требует проверки всех возможных комбинаций элементов, что для множеств с количеством элементов n будет иметь сложность O(n²). Общая сложность останется такой же, если вычисление заданного выражения свести к последовательности двуместных операций.

Для множеств, которые представлены отображением на универсум, ожидаемое количество шагов для выполнения двуместной операции будет равно количеству элементов в универсуме. Поскольку вычисление заданного выражения осуществляется как последовательность двуместных операций, каждая из которых реализована через вложенные циклы по количеству элементов в множестве, фактическая временная сложность алгоритма вычислений совпала с ожидаемой. В случае отображения на универсум с фиксированным числом элементов, сложность вычислений может считаться константной.

6. Результат измерения времени обработки для каждого из способов

Таблица 2. Результаты измерения времени обработки (микросекунды)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Мощность множеств | Количество тиков процессора при обработке множеств при различных способах представления | | | |
| Массив символов | Список | Массив битов | Машинное слово |
| 2 | 1.3 | 12.2 | 0.5 | 0.1 |
| 4 | 1.5 | 3.7 | 0.5 | 0.2 |
| 6 | 1.9 | 6.8 | 0.8 | 0.2 |
| 8 | 2.4 | 11.2 | 0.6 | 0.1 |
| 10 | 2.8 | 8.6 | 0.8 | 0.2 |
| 12 | 4.2 | 7 | 0.9 | 0.2 |
| 14 | 4.9 | 5.9 | 0.8 | 0.2 |
| 16 | 5.5 | 8.8 | 0.9 | 0.2 |
| 18 | 6.5 | 10.8 | 0.9 | 0.2 |
| 20 | 6.9 | 15.5 | 0.6 | 0.1 |
| 22 | 7.4 | 16 | 0.5 | 0.1 |
| 24 | 8 | 17.6 | 0.4 | 0.2 |
| 26 | 9.6 | 11.6 | 0.3 | 0.1 |
| Зависимость времени обработки от размера данных | Да | Нет | Нет | Нет |

7. Выводы о результатах

В ходе данной работы было установлено, что самым эффективным способом представления множеств для их обработки является использование машинного слова. Этот метод рекомендуется применять, когда имеется простая функция, позволяющая сопоставить элемент множества с соответствующим порядковым номером бита, и размер универсума не превышает разрядности слова.

Следует отметить, что аналогичное применение машинного слова справедливо и для вектора битов, однако в этом случае также существует ограничение на мощность универсума.

Наиболее медленным оказался метод обработки списков. Этот способ представления целесообразно использовать, когда мощность создаваемого множества неизвестна, и выделить память под всё множество заранее не представляется возможным.

Представление множества в виде массива элементов оправдано в тех случаях, когда есть возможность достаточно точно определить размер массива, а мощность универсума слишком велика для применения вектора битов или машинного слова.

8. Список используемых источников

1. Колинько П. Г. Пользовательские структуры данных: Методические указания по дисциплине «Алгоритмы и структуры данных, часть 1». –– СПб.: СПбГЭТУ «ЛЭТИ», 2024. — 64 с. (вып.2408)

9. Текст программы

|  |
| --- |
| #include <iostream>  #include <vector>  #include <random>  #include <algorithm>  #include <bitset>  #include <chrono>  **const** **int** SHIFT = 65;  **const** **int** ONE = 1;  **using** **namespace** std;  **struct** Node {  **char** data;  Node\* next;  Node(**char** val) : data(val), next(**nullptr**) {}  };  **class** LinkedList {  **public**:  Node\* head;  LinkedList() : head(**nullptr**) {}  **void** append(**char** data) {  Node\* newNode = **new** Node(data);  **if** (head == **nullptr**) {  head = newNode;  **return**;  }  Node\* last = head;  **while** (last->next) {  last = last->next;  }  last->next = newNode;  }  **bool** find(**char** value) {  Node\* current = head;  **while** (current) {  **if** (current->data == value) **return** **true**;  current = current->next;  }  **return** **false**; // элемент не найден  }  // Метод для отображения линейного списка  **void** display() {  Node\* current = head;  **while** (current) {  cout << current->data << " -> ";  current = current->next;  }  cout << "nullptr" << endl;  }  **void** bubbleSort() {  **if** (head == **nullptr** || head->next == **nullptr**) {  **return**;  }  **bool** swapped;  Node\* last = **nullptr**;  **do** {  swapped = **false**;  Node\* current = head;  **while** (current->next != last) {  **if** (current->data > current->next->data) {  std::swap(current->data, current->next->data);  swapped = **true**;  }  current = current->next;  }  last = current;  } **while** (swapped);  }  // Деструктор для освобождения памяти  ~LinkedList() {  Node\* current = head;  **while** (current) {  Node\* nextNode = current->next;  **delete** current;  current = nextNode;  }  }  };  **void** rand\_array(**char** arr[], **int** size) {  **char** letters[26] = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z' };  **int** n = 26;  /\*  \* ПРИМЕЧАНИЕ: time(0) возвращает количество секунд, прошедших с 1 января 1970 года.  \* Так как в коде данная функция вызывается 4 раза подряд, между их вызовами проходит гораздо меньше времени,  \* чем 1 секунда. Поэтому srand в каждом вызове возвращает одну и ту же последовательность чисел, а все множества A,B,C,D  \* оказываются сгенерированными одинаково. Для решения данной прпоблемы была взята функция std::chrono::system\_clock,  \* берущая более точное время.  \*/  // Получаем текущее время  **unsigned** seed = std::chrono::system\_clock::now().time\_since\_epoch().count();  // Инициализируем генератор случайных чисел  srand(seed);  // Перемешиваем массив letters  **for** (**int** i = 0; i < n - 1; ++i) {  **int** j = i + rand() % (n - i); // Выбираем случайный индекс из оставшейся части массива  swap(letters[i], letters[j]); // Меняем местами текущий элемент с выбранным  }  // Копируем первые size элементов в массив arr  **for** (**int** i = 0; i < size; ++i) {  arr[i] = letters[i];  cout << arr[i] << " ";  }  arr[size] = '\0';  cout << "\n";  }  **void** getting\_list(**char** arr[], LinkedList\* list\_by\_vector, **int** size) {  **for** (**int** i = 0; i < size; i++) {  (\*list\_by\_vector).append(arr[i]);  }  (\*list\_by\_vector).display();  }  **void** getting\_bool(**const** **char** arr[], **bool**\* universum) {  **for** (size\_t i = 0; arr[i]; ++i) {  universum[(**int**)arr[i] - SHIFT] = **true**;  }  **for** (**int** i = 0; i < 26; i++) {  cout << universum[i] << ' ';  }  cout << "\n";  }  **int** getting\_machine\_word(**const** **char** arr[]) {  **int** word = 0;  **for** (size\_t i = 0; arr[i]; ++i) {  word = word | ONE << (arr[i] - SHIFT);  }  bitset<**sizeof**(**int**) \* 8> binary(word);  cout << binary << '\n';  **return** word;  }  **int** logical\_processing\_arr(**int** size, **char** A[], **char** B[], **char** C[], **char** D[], **char** E[]) {  **int** j = 0;  // Добавляем элементы из A, не находящиеся в B и C  **for** (**int** i = 0; A[i]; i++) {  **if** (!(find(B, B + size, A[i]) != (B + size) || find(C, C + size, A[i]) != (C + size))) {  E[j] = A[i];  j++;  }  }  // Добавляем элементы из D  **for** (**int** i = 0; D[i]; i++) {  **if** (find(E, E + size, D[i]) == (E + size)) {  E[j] = D[i];  j++;  }  }  E[j] = '\0';  **return** j;  }  **void** logical\_processing\_li(LinkedList\* A, LinkedList\* B, LinkedList\* C, LinkedList\* D, LinkedList\* E) {  Node\* currentA = (\*A).head;  // Добавляем элементы из A, не находящиеся в B и C  **while** (currentA) {  **if** (!((\*B).find(currentA->data) || (\*C).find(currentA->data))) {  (\*E).append(currentA->data);  }  currentA = currentA->next;  }  Node\* currentD = (\*D).head;  // Добавляем элементы из D  **while** (currentD) {  **if** (!(\*E).find(currentD->data)) {  (\*E).append(currentD->data);  }  currentD = currentD->next;  }  }  **void** logical\_processing\_bool(**const** **bool**\* A, **const** **bool**\* B, **const** **bool**\* C, **const** **bool**\* D, **bool**\* E) {  **for** (size\_t i = 0; i < 26; i++) {  E[i] = A[i] && !(B[i] || C[i]) || D[i];  }  }  **void** logical\_processing\_machine(**int** A, **int** B, **int** C, **int** D, **int**& E) {  E = A & ~(B | C) | D;  }  **char** convert\_from\_universum(**bool** element, **int** i) {  **if** (element) {  **return** **char**(i + SHIFT);  }  **else** { **return** 0; }  }  **int** main() {  **int** size;  **char** arr\_A[26];  **char** arr\_B[26];  **char** arr\_C[26];  **char** arr\_D[26];  **char** arr\_E[26];  LinkedList li\_A;  LinkedList li\_B;  LinkedList li\_C;  LinkedList li\_D;  LinkedList li\_E;  **bool** bool\_A[26] = { **false** };  **bool** bool\_B[26] = { **false** };  **bool** bool\_C[26] = { **false** };  **bool** bool\_D[26] = { **false** };  **bool** bool\_E[26] = { **false** };  **int** Mword\_A = 0;  **int** Mword\_B = 0;  **int** Mword\_C = 0;  **int** Mword\_D = 0;  **int** Mword\_E = 0;  cout << "Enter the power of the arrays:\n";  cin >> size;  **while** (size > 26) {  cout << "The size of the array cannot be more than 26, since there are only 26 uppercase Latin letters. Try again:\n";  cin >> size;  }  cout << "Creating arrays with random letters:\n";  rand\_array(arr\_A, size);  rand\_array(arr\_B, size);  rand\_array(arr\_C, size);  rand\_array(arr\_D, size);  cout << "Creating lists by arrays:\n";  getting\_list(arr\_A, &li\_A, size);  getting\_list(arr\_B, &li\_B, size);  getting\_list(arr\_C, &li\_C, size);  getting\_list(arr\_D, &li\_D, size);  cout << "Creating universums by arrays:\n";  getting\_bool(arr\_A, bool\_A);  getting\_bool(arr\_B, bool\_B);  getting\_bool(arr\_C, bool\_C);  getting\_bool(arr\_D, bool\_D);  cout << "Creating machine words by arrays:\n";  Mword\_A = getting\_machine\_word(arr\_A);  Mword\_B = getting\_machine\_word(arr\_B);  Mword\_C = getting\_machine\_word(arr\_C);  Mword\_D = getting\_machine\_word(arr\_D);  **auto** t1\_arr = chrono::high\_resolution\_clock::now();  **int** result\_size = logical\_processing\_arr(size, arr\_A, arr\_B, arr\_C, arr\_D, arr\_E);  **auto** t2\_arr = chrono::high\_resolution\_clock::now();  **auto** time\_arr = chrono::duration\_cast<chrono::duration<**double**, micro>>(t2\_arr - t1\_arr).count();  sort(arr\_E, arr\_E + result\_size);  cout << "The result of processing the array: \t Execution time: " << time\_arr << "\n";  **for** (**int** i = 0; arr\_E[i]; i++) {  cout << arr\_E[i] << ' ';  }  **auto** t1\_li = chrono::high\_resolution\_clock::now();  logical\_processing\_li(&li\_A, &li\_B, &li\_C, &li\_D, &li\_E);  **auto** t2\_li = chrono::high\_resolution\_clock::now();  **auto** time\_li = chrono::duration\_cast<chrono::duration<**double**, micro>>(t2\_li - t1\_li).count();  li\_E.bubbleSort();  Node\* currentE = li\_E.head;  cout << "\nThe result of processing the list: \t Execution time: " << time\_li << "\n";  **while** (currentE) {  std::cout << currentE->data << ' ';  currentE = currentE->next;  }  **auto** t1\_bool = chrono::high\_resolution\_clock::now();  logical\_processing\_bool(bool\_A, bool\_B, bool\_C, bool\_D, bool\_E);  **auto** t2\_bool = chrono::high\_resolution\_clock::now();  **auto** time\_bool = chrono::duration\_cast<chrono::duration<**double**, micro>>(t2\_bool - t1\_bool).count();  cout << "\nThe result of processing the universum: \t Execution time: " << time\_bool << "\n";  **for** (**int** i = 0; i < 26; i++) {  **if** (bool\_E[i]) {  cout << convert\_from\_universum(bool\_E[i], i) << ' ';  }  }  **auto** t1\_mword = chrono::high\_resolution\_clock::now();  logical\_processing\_machine(Mword\_A, Mword\_B, Mword\_C, Mword\_D, Mword\_E);  **auto** t2\_mword = chrono::high\_resolution\_clock::now();  **auto** time\_mword = chrono::duration\_cast<chrono::duration<**double**, micro>>(t2\_mword - t1\_mword).count();  cout << "\nThe result of processing the machine word: \t Execution time: " << time\_mword << "\n";  **for** (**int** i = 0; i < **sizeof**(**int**) \* 8; ++i) {  **if** (Mword\_E & 1 << i) {  cout << (**unsigned** **char**)(i + SHIFT) << ' ';  }  }  **return** 0;  } |