Минцифры РФ

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Курсовой проект

по курсу

«Структуры и алгоритмы обработки данных»

Вариант 76

Выполнил: студент группы

Юров Д.А.

Проверил: старший преподаватель кафедры ПМиК

Дьячкова И.С.

Новосибирск 2024

Содержание

1. [ПОСТАНОВКА ЗАДАЧИ 3](#_bookmark0)
2. [ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ 4](#_bookmark1)
   1. [МЕТОД СОРТИРОВКИ 4](#_bookmark2)
   2. [ДВОИЧНЫЙ ПОИСК 4](#_bookmark3)
   3. [ДЕРЕВО И ПОИСК ПО ДЕРЕВУ 5](#_bookmark4)
3. [ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ 6](#_bookmark5)
4. [ОПИСАНИЕ ПРОГРАММЫ 7](#_bookmark6)
   1. [ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ 7](#_bookmark7)
   2. [ОПИСАНИЕ ПОДПРОГРАММ 8](#_bookmark8)
5. [ТЕКСТ ПРОГРАММЫ 9](#_bookmark9)
6. [РЕЗУЛЬТАТЫ 11](#_bookmark10)
7. [ВЫВОДЫ 15](#_bookmark11)
8. ПОСТАНОВКА ЗАДАЧИ

Хранящуюся в файле базу данных загрузить в оперативную память компьютера и построить индексный массив, упорядочивающий данные **по году рождения сотрудника**, используя **метод Цифровой сортировки** в качестве метода сортировки.

Предусмотреть возможность поиска по ключу в упорядоченной базе, в результате которого из записей с одинаковым ключом формируется очередь, содержимое очереди выводится на экран.

Из записей очереди построить **Двоичное дерево оптимального поиска, по приближенному алгоритму А2 по номеру отдела**, и предусмотреть возможность поиска в дереве по запросу.

Закодировать файл базы данных статическим **кодом Гилберта-Мура**, предварительно

оценив вероятности всех встречающихся в ней символов. Построенный код вывести на экран.

Стpуктуpа записи:

ФИО сотpудника: текстовое поле 30 символа

фоpмат <Фамилия>\_<Имя>\_<Отчество>

Hомеp отдела: целое число

Должность: текстовое поле 22 символа

Дата pождения: текстовое поле 10 символов

фоpмат дд-мм-гг

Пpимеp записи из БД:

Петpов\_Иван\_Иванович\_\_\_\_\_\_\_\_\_\_\_\_

130

начальник\_отдела\_\_\_\_\_\_

15-03-46

Ваpианты условий упоpядочения и ключи поиска (К):

C = 3 - по дате(!) pождения, К = год pождения.

1. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ
   1. МЕТОД СОРТИРОВКИ

Цифровая сортировка

Пусть дана последовательность из S чисел, представленных в *m* – ичной системе счисления. Каждое число состоит из L цифр d1d2…dL, 0 ≤ di ≤ *m* – 1, i=1..L. Сначала числа из списка S распределяются по *m* очередям, причём номер очереди определяется последней цифрой каждого числа. Затем полученные очереди соединяются в список, для которого все действия повторяются, но распре- деление по очередям производится в соответствии со следующей цифрой и т.д.

Цифровой метод может успешно использоваться не только для сортировки чисел, но и для сортировки любой информации, представленной в памяти ком-пьютера. Необходимо лишь рассматривать каждый байт ключа сортировки как цифру, принимающую значения от 0 до 255. Тогда для сортировки потребуется *m*=256 очередей. Для выделения каждого байта ключа сортировки можно исполь-зовать массив Digit, наложенный в памяти компьютера на поле элемента после-довательности, по которому происходит сортировка.

Для цифровой сортировки М<const L(*m+n*). При фиксированных *m* и L М=O(*n*) при *n → ∞*, что значительно быстрее остальных рассмотренных методов. Однако если длина чисел L велика, то метод может проигрывать обычным мето-дам сортировки. Кроме того, Метод применим только, если задача сортировки сводится к задаче упорядочивания чисел, что не всегда возможно. Метод обеспечивает устойчивую сортировку. Чтобы изменить направление 36 сортировки на обратное, очереди нужно присоединять в обратном порядке.

* 1. ДВОИЧНЫЙ ПОИСК

Алгоритм двоичного поиска в упорядоченном массиве сводится к следующему. Берём средний элемент отсортированного массива и сравниваем с ключом X. Возможны три варианта:

Выбранный элемент равен X. Поиск завершён.

Выбранный элемент меньше X. Продолжаем поиск в правой половине массива. Выбранный элемент больше X. Продолжаем поиск в левой половине массива.

Из-за необходимости найти все элементы соответствующие заданному ключу поиска в курсовой работе использовалась вторая версия двоичного поиска, которая из необходимых элементов находит самый левый, в результате чего для поиска остальных требуется просматривать лишь оставшуюся правую часть массива.

Верхняя оценка трудоёмкости алгоритма двоичного поиска такова. На каждой итерации поиска необходимо два сравнение для первой версии, одно сравнение для второй версии. Количество

итераций не больше, чем . Таким образом, трудоёмкость двоичного поиска в обоих случаях



* 1. ДЕРЕВО И ПОИСК ПО ДЕРЕВУ

*Второй алгоритм* (А2) использует предварительно упорядоченный набор вершин. В качестве корня выбирается такая вершина, что разность весов левого и правого поддеревьев была минимальна. Для этого путем последовательного суммирования весов определим вершину *Vk*, для которой справедливы неравенства:

Тогда в качестве "центра тяжести" может быть выбрана вершина *Vk, Vk-1* или *Vk+1*, т. е. вершина, для которой разность весов левого и правого поддерева минималь- на. Далее действия повторяются для каждого поддерева

Приведем некоторые свойства рассмотренных приближенных алгоритмов:

1) Сложность алгоритмов как функция от *n* (количество элементов) зави- сит следующим образом: время Т = О(*n* log *n*), память М = О(*n*) при

80 *n→∞.* (Время определяется трудоемкостью сортировки элементов, а память – размером массива для хранения элементов)

2) Дерево, построенное приближенным алгоритмом А1, равносильно случайному (с точки зрения средней высоты) при *n→∞,* т.е. алгоритм А1– плохой*.*

3) Дерево, построенное приближенным алгоритмом А2, асимптотически приближается к оптимальному (с точки зрения средней высоты) при *n→∞,* т.е. алгоритм А2 является хорошим.

4) ИСДП нельзя считать даже приближением к дереву оптимального по-иска.

2.4 МЕТОД КОДИРОВАНИЯ

Алфавитный код Гилберта – Мура

Рассмотрим источник с алфавитом А={*a1,a2,…,an*} и вероятностями *p1,…pn*. Пусть символы алфавита некоторым образом упорядочены, например, *a1≤a2≤…≤an*. *Алфавитным* называется код, в котором кодовые слова лексико-графически упорядочены, т.е. *φ(a1)≤φ(a2)≤…≤φ(an).*

Е.Н. Гилбертом и Э.Ф. Муром предложили метод построения алфавитного кода, для которого *Lср*

*< H+2.* Процесс построения происходит следующим образом.

1. Составим суммы *Qi, i=1,n*, вычисленные следующим образом:

*Q1=p1/2, Q2=p1+p2/2, Q3=p1+p2+p3/2,…, Qn=p1+p2+…+pn-1+pn/2*.

1. Представим суммы *Qi* в двоичном виде.
2. В качестве кодовых слов возьмем -log2*pi* +1 младших бит в двоичном представлении *Qi*.

**Пример**. Пусть дан алфавит A={*a1, a2, a3, a4, a5, a6*} с вероятностями *p1*=0.36, *p2*=0.18, *p3*=0.18, *p4*=0.12, *p5*=0.09, *p6*=0.07. Построенный код приведен в таблице.

Таблица 1 Код *Гилберта-Мура*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *ai* | *Pi* | *Qi* | *Li* | кодовое слово |
| *a1* | 1/23≤0.18 | 0.09 | 4 | 0001 |
| *a2* | 1/23≤0.18<1/22 | 0.27 | 4 | 0100 |
| *a3* | 1/22≤0.36<1/21 | 0.54 | 3 | 100 |
| *a4* | 1/24≤0.07 | 0.755 | 5 | 11000 |
| *a5* | 1/24≤0.09 | 0.835 | 5 | 11010 |
| *a6* | 1/24≤0.12 | 0.94 | 5 | 11110 |

Средняя длина кодового слова не превышает значения энтропии плюс 2

*Lср*=4**.**0.18+4**.**0.18+3**.**0.36+5**.**0.07+5**.**0.09+5**.**0.12=3.92<2.37+2

1. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

1. З а г р у з к а и в ы в о д б а з ы д а н н ы х

Для загрузки базы данных прописана процедура *std*::ifstream file() в функции main(), в которой производится считывание записей типа *record2(Имя сотрудника).* Здесь же предусмотрена проверка на наличие файла, откуда выполняется считывание и проверка на выделение памяти для считывания.

За вывод элементов считанной базы данных отвечает процедура showPage()*.*Она предоставляет возможность просмотра базы данных по 20 элементов на странице с возможностью выхода из режима просмотра.

1. В с п о м о г а т е л ь н ы е ф у н к ц и и и п р о ц е д у р ы д л я с о р т и р о в к и д а н н ы х

Для сортировки данных используется процедура showPage(), которая очищает экран и вызывает функцию сортировки DigitalSort(). Доступ к записям базы данных осуществляется через массив array, для перевода строк в числа написана функция и их сравнение compare\_strings() (т.к. не используются стандартные процедуры сравнения двух строк).

1. О с о б е н н о с т и р е а л и з а ц и и б и н а р н о г о п о и с к а и п о с т р о е н и я о ч е р е д и

Бинарный поиск по отсортированной базе осуществляется в процедуре bynarySearchAll(), которая принимает в качестве параметра ключ. Доступ к записям ведётся через массив array, найденные записи заносятся в очередь в процедуре createAndPrintQueue(). Для вывода на монитор найденных записей используется процедура createAndPrintQueue(). При реализации бинарного поиска была использована его вторая версия, так как в результате ее выполнения возвращается номер самого левого из найденных элементов, благодаря чему легко найти и вывести остальные элементы, лишь просмотрев оставшуюся правую часть массива, пока не встретится запись, не удовлетворяющая ключу поиска.

1. О с о б е н н о с т и п о с т р о е н и я д е р е в а , е г о в ы в о д а н а э к р а н и п о и с к а

Построение дерева осуществляется в функции buildOptimalSearchTree(), возвращающей указатель на корень дерева. Внутри нее происходит непосредственно добавление каждого элемента из очереди. Для вывода дерева на экран используется процедура printTree(), которая совершает обход по дереву (ЛКП) и выводит(форматированно) данные на экран по 20 элементов. Поиск в дереве осуществляется с помощью функции searchTree(), которая, заходя в корень дерева проверяет данные на соответствие ключу, если они идентичны, то совершается поиск в левом и правом поддеревьяхи на экран выводится корень дерева. Если ключ поиска и данные в корне дерева различны, то используется поиск в левом или правом поддеревьях в зависимости от данных. Если искомые данные меньше корня, то ищем в левом поддереве, иначе – в правом поддереве. Также, когда находим дубликаты записей по ключу, то записываем их в отдельное поле next, при помощи процедуры searchTree() и выводим их при помощи процедуры printDuplicates().

1. К о д и р о в а н и е д а н н ы х

Кодирование базы данных начинается с процедуры readAndCountFrequencies(), в которой происходит построение массива встречаемых в базе символов simbols(состоит из структур, содержащих имя(порядковый номер), количества встречи символа в базе данных и вычисленной вероятности встречи), вызов построения кода generateGilbertMooreCodes(). А также форматированный вывод на экран результата, который включает в себя:

* + символы, встречающиеся в базе
  + вероятность появления символа
  + длину каждого кодового слова
  + кодовое слово каждого символа

-энтропию

-среднюю длину кодового слова

-общую сумму вероятностей.

Кодирование происходит в процедуре generateGilbertMooreCodes().

1. ОПИСАНИЕ ПРОГРАММЫ
   1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ

глобальные переменные и константы:

queue \*head = NULL, \*tail = NULL; - указатели типа queue на начало и конец списка соответственно record2 \*database; - указатель на массив структур типа record3(«Предприятие»)

struct record2 - структура, используемая для работы с базой данных «Населенный пункт».

{

char a[30]; - поле а типа char (используется для хранения ФИО) 30 символа

shirt int b[18]; - поле b типа short int (используется для хранения номера отдела)

char c[22]; - поле с типа char(используется для хранения должности сотрудника)

char d[10]; - поле е типа char(используется для хранения даты поселения в формате ДД-ММ- ГГ) 10 символов.

};

struct Node - структура, используемая для работы со списком

{

Record2 data; - поле data типа record2(используется для хранения данных типа «Предприятие»)

Node \*next; -поле, хранящее указатель типа Node(указатель на следующий элемент в списке)

Node \*prev; -поле, хранящее указатель типа Node(указатель на предыдущий элемент в списке)

};

struct TreeNode - Структура, представляющая дерево оптимального поиска (А2).

{

Record2 data; - поле данных data типа record2 для хранения данных типа «Предприятие»

TreeNode \* left; - поле, хранящее указатель типа TreeNode на левое поддерево

TreeNode \* right; - поле, хранящее указатель типа TreeNode на правое поддерево

TreeNode \* next; - поле, хранящее указатель типа TreeNode на дубликаты

TreeNode(Data d) : data(d), left(nullptr), right(nullptr), next(nullptr) {} -

конструктор, обнуляющие все поля в структуре, а поле типа char заполняет символом «пробел»

};

struct symbol - Структура, предназначенная для кодирования базы данных.

{

public:

int frequency; поле типа int, предназначенное для хранения количества раз, которое символ

встретился в тексте

double probability; поле типа double (используется для хранения вероятности встречи символа в базе данных)

};

* 1. ОПИСАНИЕ ПОДПРОГРАММ

Процедуры для обработки базы данных:

1. void push\_back(); - чтение базы данных с построением индексного массива, возвращает указатель типа record2 на указатель на индексный массив структур
2. void showPage (Node\* head); - печать базы по 20 элементов, в качестве параметра принимает указатель типа Node на указатель на список структур
3. int main(); - очистка памяти после базы, в качестве параметра принимает указатель типа Node на указатель на список структур

Функции и процедуры сортировки:

1. void DigitalSort(Node\*\* head); - сортировка базы данных, в качестве параметра принимает указатель типа Node на указатель на список структур
2. int compare\_strings(char\* str1, char\* str2, int size); - сравнение строк, в качестве параметров принимает символьные массивы str1 и str2 – сравниваемые строки, и size – количество символов, по которым происходит сравнение, возвращает , -1 если str1 < str2, если они равны - 0, 1 – иначе.

Функции и процедуры для поиска в отсортированной базе данных:

1. void binarySearchAll(Data\* array, int target, int size, int\*& foundIndices, int& foundCount); - в бинарном поиске, в качестве параметра принимает указатель типа Data на массив структур, а также сам ключ поиска, размер, индексный массив и количество найденных элементов)
2. createAndPrintQueue(Data\* array, int size, int\*& foundIndices, int& foundCount); - в качестве параметра, при вызове процедуры добавления и вывод очеред, принимает указатель типа Data на массив структур, а также сам ключ поиска, размер, индексный массив и количество найденных элементов)

Процедуры и функции построения двоичного Б-дерева и поиска в нем

1. TreeNode\* buildOptimalSearchTree(Data\* array, int\* indices, int start, int end); - добавление элемента из списка в дерево, возвращает указатель на корень дерева типа TreeNode
2. printTree(TreeNode\* root, int level = 0); - вывод дерева на экран, в качестве параметра принимает указатель на корень дерева.
3. void search\_in\_tree(tree\* root); - поиск в дереве, , в качестве параметра принимает указатель на корень дерева.
4. TreeNode\* searchTree(TreeNode\* root, short int departmentNum); - поиск в дереве, , в качестве параметров принимает указатель на корень дерева, и ключ поиска типа int

Процедуры и функции кодирования базы данных:

1. generateGilbertMooreCodes(std::vector<Symbol>& symbols); - основная функция кодирования
2. void readAndCountFrequencies(const char\* filename, std::vector<Symbol>& symbols, int& totalChars);-сортировка вероятностей, в качестве параметров принимает массив указатель на массив символов типа symbol, l и r соответственно левая и правая границы сортировки.
3. void calculateProbabilities(std::vector<Symbol>& symbols, int totalChars); - вычисление вероятности встречи символа, возвращает вероятность встречи символа в базе данных, в качестве параметров принимает указатель на массив символов типа symbol и количество символов, встречающихся в бае данных соответственно.
4. void gilbert(symbol\* array\_symbols, int n, double \*q, int \*l, bool \*\*c); функция отвечает за составление кодовых слов для каждого символа базы данных, в качестве параметров принимает:

Symbol\* symbols – указатель на массив структур символов типа Symbol int n - количество разных символов в базе данных

double \*q - указатель на массив под Q

double calculateAverageLength(const std::vector<Symbol>& symbols); - указатель на массив, в котором формируется длина кодового символа для каждого символа.

Основная программа:

1. main() – основная программа, в которой последовательно вызываются процедуры для работы с базой данных
2. ТЕКСТ ПРОГРАММЫ

// main.cpp

#include <iostream>

#include <fstream>

#include <string>

#include <cstring>

#include <vector>

#include <algorithm>

#include <cmath>

#include <iomanip>

#include <numeric>

struct Data {

char a[30];

short int b;

char c[22];

char d[10];

};

struct Node {

Data data;

Node\* next;

Node\* prev;

};

// Узел дерева поиска

struct TreeNode {

Data data;

TreeNode\* left;

TreeNode\* right;

TreeNode\* next;

TreeNode(Data d) : data(d), left(nullptr), right(nullptr), next(nullptr) {}

};

struct Symbol {

unsigned char ch;

int frequency;

double probability;

double q;

std::string code;

};

int compareStrings(const char\* str1, const char\* str2) {

int len1 = strlen(str1);

int len2 = strlen(str2);

while (len1 > 0 && str1[len1 - 1] == ' ') {

len1--;

}

for (int i = 0; i < len1 && i < len2; i++) {

if (str1[i] != str2[i]) {

return str1[i] - str2[i];

}

}

if (len1 < len2) return -1;

if (len1 > len2) return 1;

return 0;

}

void sortArrayByName(Data\* array, int\* indices, int size) {

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) {

if (compareStrings(array[indices[j]].a, array[indices[j + 1]].a) > 0) {

std::swap(indices[j], indices[j + 1]);

}

}

}

}

void sortArrayByDepartment(Data\* array, int\* indices, int size) {

for (int i = 0; i < size; i++) {

indices[i] = i;

}

for (int i = 0; i < size - 1; i++) {

for (int j = 0; j < size - i - 1; j++) {

if (array[indices[j]].b > array[indices[j + 1]].b) {

std::swap(indices[j], indices[j + 1]);

}

}

}

}

int getHeight(TreeNode\* root) {

if (!root) return 0;

return 1 + std::max(getHeight(root->left), getHeight(root->right));

}

bool isBalanced(TreeNode\* root) {

if (!root) return true;

int leftHeight = getHeight(root->left);

int rightHeight = getHeight(root->right);

if (std::abs(leftHeight - rightHeight) <= 1 &&

isBalanced(root->left) &&

isBalanced(root->right))

return true;

return false;

}

TreeNode\* buildOptimalSearchTree(Data\* array, int\* indices, int start, int end) {

if (start > end) return nullptr;

if (start == 0) {

sortArrayByDepartment(array, indices, end + 1);

}

int mid = start + (end - start) / 2;

TreeNode\* root = new TreeNode(array[indices[mid]]);

std::cout << "Building tree node with department number: " << root->data.b << std::endl;

root->left = buildOptimalSearchTree(array, indices, start, mid - 1);

root->right = buildOptimalSearchTree(array, indices, mid + 1, end);

return root;

}

void printTree(TreeNode\* root, int level = 0) {

if (!root) return;

printTree(root->right, level + 1);

std::cout << "Department: " << root->data.b << " | Name: " << root->data.a << " | Date: " << root->data.d << "\n";

printTree(root->left, level + 1);

}

void findAllDepartments(TreeNode\* root, short int departmentNum, TreeNode\*& firstFound) {

if (!root) return;

findAllDepartments(root->left, departmentNum, firstFound);

if (root->data.b == departmentNum) {

if (!firstFound) {

firstFound = root;

}

else {

TreeNode\* duplicate = new TreeNode(root->data);

duplicate->next = firstFound->next;

firstFound->next = duplicate;

}

}

findAllDepartments(root->right, departmentNum, firstFound);

}

TreeNode\* searchTree(TreeNode\* root, short int departmentNum) {

TreeNode\* firstFound = nullptr;

findAllDepartments(root, departmentNum, firstFound);

if (firstFound) {

std::cout << "\n[FOUND DEPARTMENT " << departmentNum << "]:\n";

std::cout << "Main record: " << firstFound->data.a << " | Date: " << firstFound->data.d << "\n";

if (firstFound->next != nullptr) {

std::cout << "\nDuplicate records found:\n";

TreeNode\* duplicate = firstFound->next;

int count = 1;

while (duplicate != nullptr) {

std::cout << count << ". " << duplicate->data.a

<< " | Date: " << duplicate->data.d << "\n";

count++;

duplicate = duplicate->next;

}

}

}

return firstFound;

}

void printDuplicates(TreeNode\* node) {

if (node == nullptr) {

std::cout << "Node not found\n";

return;

}

std::cout << "\n[Department " << node->data.b << " Records]:\n";

std::cout << "Main record: " << node->data.a << " | Date: " << node->data.d << "\n";

if (node->next != nullptr) {

std::cout << "\nDuplicate records:\n";

TreeNode\* current = node->next;

int count = 1;

while (current != nullptr) {

std::cout << count << ". " << current->data.a

<< " | Date: " << current->data.d << "\n";

count++;

current = current->next;

}

}

else {

std::cout << "No duplicates found.\n";

}

}

TreeNode\* insertNode(TreeNode\* root, Data data) {

if (root == nullptr) {

return new TreeNode(data);

}

if (data.b == root->data.b) {

TreeNode\* newNode = new TreeNode(data);

newNode->next = root->next;

root->next = newNode;

return root;

}

if (data.b < root->data.b) {

root->left = insertNode(root->left, data);

}

else {

root->right = insertNode(root->right, data);

}

return root;

}

void deleteTree(TreeNode\* root) {

if (!root) return;

deleteTree(root->left);

deleteTree(root->right);

delete root;

}

void push\_back(Node\*& head, Node\*& tail, const Data& data) {

Node\* newNode = new Node{ data, nullptr, tail };

if (tail) {

tail->next = newNode;

}

else {

head = newNode;

}

tail = newNode;

}

Data\* listToArray(Node\* head, int& size) {

size = 0;

Node\* current = head;

while (current) {

++size;

current = current->next;

}

Data\* array = new Data[size];

current = head;

for (int i = 0; i < size; ++i) {

array[i] = current->data;

current = current->next;

}

return array;

}

int convertDateToInt(const char\* date) {

std::string cleanedDate;

for (int i = 6; date[i] != '\0'; ++i) {

if (date[i] != '-') {

cleanedDate += date[i];

}

}

return std::atoi(cleanedDate.c\_str());

}

void binarySearchAll(Data\* array, int target, int size, int\*& foundIndices, int& foundCount) {

foundCount = 0;

int left = 0;

int right = size - 1;

int mid = -1;

while (left < right) {

mid = left + (right - left) / 2;

int midDate = convertDateToInt(array[mid].d);

if (midDate < target) {

left = mid + 1;

}

else {

right = mid;

}

}

if (convertDateToInt(array[left].d) != target) {

foundIndices = nullptr;

return;

}

int leftIdx = left;

int rightIdx = left;

while (rightIdx < size && convertDateToInt(array[rightIdx].d) == target) {

rightIdx++;

}

foundCount = rightIdx - leftIdx;

foundIndices = new int[foundCount];

for (int i = 0; i < foundCount; ++i) {

foundIndices[i] = leftIdx + i;

}

}

void createAndPrintQueue(Data\* array, int size, int\*& foundIndices, int& foundCount) {

Node\* tempHead = nullptr;

Node\* tempTail = nullptr;

std::cout << "\n";

for (int i = 0; i < foundCount; ++i) {

push\_back(tempHead, tempTail, array[foundIndices[i]]);

}

int index = 1;

Node\* current = tempHead;

while (current != nullptr) {

std::cout << "| " << index++ << ". " << current->data.a << "\t" << current->data.b

<< "\t" << current->data.c << "\t" << current->data.d << " |" << "\n";

Node\* temp = current;

current = current->next;

delete temp;

}

}

void readAndCountFrequencies(const char\* filename, std::vector<Symbol>& symbols, int& totalChars) {

std::ifstream file(filename, std::ios::binary);

if (!file) {

std::cerr << "Error opening file!" << std::endl;

return;

}

symbols.resize(256);

for (int i = 0; i < 256; i++) {

symbols[i].ch = static\_cast<unsigned char>(i);

symbols[i].frequency = 0;

}

unsigned char ch;

totalChars = 0;

while (file.read(reinterpret\_cast<char\*>(&ch), sizeof(ch))) {

symbols[ch].frequency++;

totalChars++;

}

file.close();

}

void calculateProbabilities(std::vector<Symbol>& symbols, int totalChars) {

for (auto& symbol : symbols) {

symbol.probability = static\_cast<double>(symbol.frequency) / totalChars;

}

std::sort(symbols.begin(), symbols.end(), [](const Symbol& a, const Symbol& b) {

return a.ch < b.ch;

});

}

void calculateCumulativeProbabilities(std::vector<Symbol>& symbols) {

double cumulative = 0.0;

for (auto& symbol : symbols) {

symbol.q = cumulative + (symbol.probability / 2.0);

cumulative += symbol.probability;

}

}

double calculateEntropy(const std::vector<Symbol>& symbols) {

double entropy = 0.0;

for (const auto& symbol : symbols) {

if (symbol.probability > 0) {

entropy -= symbol.probability \* std::log2(symbol.probability);

}

}

return entropy;

}

double calculateAverageLength(const std::vector<Symbol>& symbols) {

double avgLength = 0.0;

for (const auto& symbol : symbols) {

avgLength += symbol.probability \* symbol.code.length();

}

return avgLength;

}

void generateGilbertMooreCodes(std::vector<Symbol>& symbols) {

if (symbols.empty()) return;

int totalChars = 0;

for (const auto& symbol : symbols) {

totalChars += symbol.frequency;

}

calculateProbabilities(symbols, totalChars);

calculateCumulativeProbabilities(symbols);

for (auto& symbol : symbols) {

double q = symbol.q;

symbol.code = "";

int codeLength = static\_cast<int>(-std::log2(symbol.probability) + 2);

for (int i = 0; i < codeLength; ++i) {

q \*= 2;

if (q >= 1.0) {

symbol.code += "1";

q -= 1.0;

}

else {

symbol.code += "0";

}

}

}

double entropy = calculateEntropy(symbols);

double avgLength = calculateAverageLength(symbols);

double redundancy = avgLength - entropy;

std::cout << "\nGilbert-Moore Codes (Alphabetically sorted):\n";

std::cout << "Symbol\tFreq\tProb\t\tCode\t\tLength\n";

std::cout << "--------------------------------------------------------\n";

for (const auto& symbol : symbols) {

std::cout << symbol.ch << "\t"

<< symbol.frequency << "\t"

<< std::fixed << std::setprecision(6) << symbol.probability << "\t"

<< symbol.code << "\t\t"

<< symbol.code.length() << "\n";

}

std::cout << "\nCode Characteristics:\n";

std::cout << "--------------------------------------------------------\n";

std::cout << "Entropy (H): " << std::fixed << std::setprecision(6) << entropy << " bits\n";

std::cout << "Average Code Length (L): " << avgLength << " bits\n";

std::cout << "Redundancy (L - H): " << redundancy << " bits\n";

std::cout << "Relative Redundancy ((L - H)/H): " << (redundancy / entropy) \* 100 << "%\n\n";

}

void processGilbertMooreCoding(const char\* filename) {

std::vector<Symbol> symbols;

int totalChars = 0;

readAndCountFrequencies(filename, symbols, totalChars);

symbols.erase(

std::remove\_if(symbols.begin(), symbols.end(),

[](const Symbol& s) { return s.frequency == 0; }),

symbols.end()

);

generateGilbertMooreCodes(symbols);

}

void showPage(Node\* head) {

Node\* it = head;

char choice;

const int pageSize = 20;

int size = 0;

char searchTarget[10];

Data\* array = listToArray(head, size);

int processedDate;

int countSymbols;

std::vector<Symbol> symbols;

int totalChars = 0;

double entropy;

double avgLength;

while (true) {

for (int i = 0; i < pageSize && it != nullptr; ++i) {

std::cout << it->data.a << "\t" << it->data.b << "\t"

<< it->data.c << "\t" << it->data.d << "\n";

it = it->next;

}

std::cout << "N/n - next page, P/p - previous page, Q/q - quit, S/s - search, G/g - show codes" << std::endl;

std::cin >> choice;

switch (choice) {

case 'N':

case 'n':

if (it == nullptr) it = head;

break;

case 'P':

case 'p':

for (int i = 0; i < pageSize && it != nullptr; ) {

if (it->prev == nullptr) break;

it = it->prev;

++i;

}

break;

case 'G':

case 'g':

processGilbertMooreCoding("testBase2.dat");

break;

case 'S':

case 's': {

std::cout << "[INPUT YEAR]: ";

std::cin >> searchTarget;

processedDate = convertDateToInt(searchTarget);

int\* foundIndices = nullptr;

int foundCount = 0;

binarySearchAll(array, processedDate, size, foundIndices, foundCount);

if (foundCount > 0) {

std::cout << "\n[RECORDS FOUND]:\n";

std::cout << "===================================================================================";

createAndPrintQueue(array, size, foundIndices, foundCount);

std::cout << "===================================================================================";

std::cout << "\n\n";

TreeNode\* root = buildOptimalSearchTree(array, foundIndices, 0, foundCount - 1);

std::cout << "\n[OPTIMAL SEARCH TREE]:\n";

printTree(root);

std::cout << "\n";

short int searchDepartment;

std::cout << "Enter the department number:\n";

std::cin >> searchDepartment;

TreeNode\* result = searchTree(root, searchDepartment);

if (result) {

std::cout << "\n[FOUND]:\n";

std::cout << "|" << result->data.a << " " << result->data.d << "|\n\n";

printDuplicates(result);

}

else {

std::cout << "\n[NOT FOUND]: No department with the number '" << searchDepartment << "' exists in the tree.\n";

}

deleteTree(root);

delete[] foundIndices;

}

else {

std::cout << "[NO RECORDS FOUND FOR DATE]: " << searchTarget << '\n\n';

}

break;

}

case 'Q':

case 'q':

delete[] array;

return;

default:

break;

}

if (it == nullptr) it = head;

}

}

void extractDate(const char\* date, int& day, int& month, int& year) {

day = (date[0] - '0') \* 10 + (date[1] - '0');

month = (date[3] - '0') \* 10 + (date[4] - '0');

year = (date[6] - '0') \* 10 + (date[7] - '0');

}

void DigitalSort(Node\*\* head) {

if (\*head == nullptr) return;

Node\* buckets[100] = { nullptr };

Node\* tails[100] = { nullptr };

for (int j = 0; j < 3; j++) {

Node\* current = \*head;

while (current != nullptr) {

Node\* next = current->next;

int day, month, year;

extractDate(current->data.d, day, month, year);

int key;

if (j == 0) {

key = day;

}

else if (j == 1) {

key = month;

}

else {

key = year;

}

if (buckets[key] == nullptr) {

buckets[key] = current;

}

else {

tails[key]->next = current;

current->prev = tails[key];

}

tails[key] = current;

current->next = nullptr;

current = next;

}

Node\* newHead = nullptr;

Node\* newTail = nullptr;

for (int i = 0; i < 100; ++i) {

if (buckets[i] != nullptr) {

if (newHead == nullptr) {

newHead = buckets[i];

}

else {

newTail->next = buckets[i];

buckets[i]->prev = newTail;

}

newTail = tails[i];

buckets[i] = tails[i] = nullptr;

}

}

\*head = newHead;

}

}

int main() {

std::ifstream file("testBase2.dat", std::ios::binary);

if (!file.is\_open()) {

std::cout << "File isn't open!" << std::endl;

return 1;

}

Node\* head = nullptr;

Node\* tail = nullptr;

while (true) {

Data temp;

file.read(reinterpret\_cast<char\*>(&temp), sizeof(Data));

if (file.gcount() == 0) break;

push\_back(head, tail, temp);

}

DigitalSort(&head);

showPage(head);

Node\* current = head;

while (current != nullptr) {

Node\* nextNode = current->next;

delete current;

current = nextNode;

}

file.close();

return 0;

}

1. РЕЗУЛЬТАТЫ

A screenshot of a computer program

Description automatically generated

Рисунок 1. Отсортированная по ФИО и улице база данн

A screenshot of a computer

Description automatically generated

Рисунок 2. Очередь из записей, полученных в результате поиска года рождения.

A screenshot of a computer

Description automatically generated

Рисунок 4. Дерево, ключ в дереве – номер отдела работника(2 поле).

A screen shot of a computer

Description automatically generated

Рисунок 5. Поиск по дереву (элементы с одинаковым ключом).

Результаты кодирования базы данных(начальный и конечный фрагменты):

A screenshot of a computer

Description automatically generated

Рисунок 6. Примеры кодовых слов для наиболее вероятных символов в базе

A screenshot of a computer

Description automatically generated

Рисунок 7. Примеры кодовых слов для минимально вероятных символов в базе, а также вычисленная средняя длина кодового слова и энтропия источника

1. ВЫВОДЫ

В ходе выполнения курсового проекта были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, двоичного поиска, создания очереди, построения двоичного бинарного дерева, поиска по дереву, кодирования данных.

Четкая структуризация кода и грамотно подобранные имена переменных, структур данных, функций и процедур способствуют удобочитаемости программы.

Все разработанные алгоритмы расширяют возможности работы с данными и способствуют улучшению эффективности анализа и обработки данных и представляют собой минимальный набор процедур для представления и обработки базы данных.