Федеральное агентство связи

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

Курсовой проект

«Структуры и алгоритмы обработки данных»

Вариант 50

Выполнил: студент группы ИП-715

Эпов И. В.

Проверил: ассистент кафедры ПМиК

Турцев А.А.

Новосибирск 2018

Содержание

[1. ПОСТАНОВКА ЗАДАЧИ 3](#_Toc533143399)

[2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ 4](#_Toc533143400)

[2.1. МЕТОД СОРТИРОВКИ 4](#_Toc533143401)

[2.2 ДВОИЧНЫЙ ПОИСК 4](#_Toc533143402)

[2.3 ДЕРЕВО И ПОИСК ПО ДЕРЕВУ 5](#_Toc533143403)

[3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ 7](#_Toc533143404)

[4. ОПИСАНИЕ ПРОГРАММЫ 8](#_Toc533143405)

[4.1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ 8](#_Toc533143406)

[4.2. ОПИСАНИЕ ПОДПРОГРАММ 8](#_Toc533143407)

[5. ТЕКСТ ПРОГРАММЫ 11](#_Toc533143408)

[6. РЕЗУЛЬТАТЫ 22](#_Toc533143409)

[7. ВЫВОДЫ 25](#_Toc533143410)

1. ПОСТАНОВКА ЗАДАЧИ

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

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

Из записей очереди построить **АВЛ-дерево по названию**, и предусмотреть возможность поиска в дереве по запросу.

Закодировать файл базы данных статическим **кодом Шеннона**, предварительно оценив вероятности всех встречающихся в ней символов. Построенный код вывести на экран.

Библиогpафическая база данных "Жизнь замечательных людей"

Стpуктуpа записи:

Автоp: текстовое поле 12 символов

фоpмат <Фамилия>\_<буква>\_<буква>

Заглавие: текстовое поле 32 символа

фоpмат <Имя>\_<Отчество>\_<Фамилия>

Издательство: текстовое поле 16 символов

Год издания: целое число

Кол-во стpаниц: целое число

Пpимеp записи из БД:

Кловский\_В\_Б

Лев\_Hиколаевич\_Толстой\_\_\_\_\_\_\_\_\_

Молодая\_гваpдия\_

1963

864

Варианты условий упорядочения и ключи поиска (К):

по году издания и автоpу, К = год издания.

2. ОСНОВНЫЕ ИДЕИ И ХАРАКТЕРИСТИКИ ПРИМЕНЯЕМЫХ МЕТОДОВ

2.1. МЕТОД СОРТИРОВКИ

Метод Хоара

Метод Хоара или метод быстрой сортировки заключается в следующем. Возьмём произвольный элемент массива х. Просматривая массив слева, найдём элемент ai ≥x. Просматривая массив справа, найдём aj ≤x. Поменяем местами ai и aJ . Будем продолжать процесс просмотра и обмена, до тех пор пока i не станет больше j. Тогда массив можно разбить на две части: в левой части все элементы не больше х, в правой части массива не меньше х. Затем к каждой части массива применяется тот же алгоритм.

Очевидно, трудоёмкость метода существенно зависит от выбора элемента х, который влияет на разделение массива. Максимальные значения М и С для метода быстрой сортировки достигаются при сортировке упорядоченных массивов (в прямом и обратном порядке). Тогда в этом случае в одной части остаётся только один элемент (минимальный или максимальный), а во второй – все остальные элементы. Выражения для М и С имеют следующий вид

M=*3(n-1),* C=*(n2+5n+4)/2*

Таким образом, в случае упорядоченных массивов трудоёмкость сортировки имеет квадратичный порядок.

Элемент *am* называется *медианой* для элементов *aL…aR,* если количество элементов меньших *am* равно количеству элементов больших *am* с точностью до одного элемента (если количество элементов нечётно). В примере буква К- медиана для КУРАПОВАЕ.

Минимальная трудоемкость метода Хоара достигается в случае, когда на каждом шаге алгоритма в качестве ведущего элемента выбирается медиана массива. Количество сравнений в этом случае C=*(n+1)*log*(n+1)-(n+1)*. Количество пересылок зависит от положения элементов, но не может быть больше одного обмена на два сравнения. Поэтому количество пересылок – величина того же порядка, что и число сравнений. Асимптотические оценки для средних значений М и С имеют следующий вид

С=О(*n* log *n*), М=О(*n* log *n*) при *n → ∞*.

Метод Хоара неустойчив.

2.2 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



2.3 ДЕРЕВО И ПОИСК ПО ДЕРЕВУ

Дерево поиска называется *сбалансированным по высоте*, или *АВЛ – деревом*, если для каждой его вершины высоты левого и правого поддеревьев отличаются не более чем на 1.

На рисунке 39 приведены примеры деревьев, одно из которых является АВЛ-деревом, а другое – нет. В выделенной вершине нарушается баланс высот левого и правого поддеревьев.

**+**

**–**

Рисунок 39 Пример АВЛ-дерева и не АВЛ-дерева

Заметим, что ИСДП является также и АВЛ – деревом. Обратное утверждение не верно.

Адельсон – Вельский и Ландис доказали теорему, гарантирующую, что АВЛ-дерево никогда не будет в среднем по высоте превышать ИСДП более, чем на 45% независимо от количества вершин:

log(*n*+1) ≤ hАВЛ(*n*) < 1,44 log(*n*+2) – 0,328 при *n.*

Таким образом, лучший случай сбалансированного по высоте дерева – ИСДП, худший случай – плохое АВЛ – дерево. *Плохое АВЛ – дерево*это АВЛ-дерево, которое имеет наименьшее число вершин при фиксированной высоте. Рассмотрим процесс построения плохого АВЛ-дерева. Возьмём фиксированную высоту h и построим АВЛ – дерево с минимальным количеством вершин. Обозначим такое дерево через Th. Ясно, что Т0 – пустое дерево, Т1 – дерево с одной вершиной. Для построения Тh при h > 1 будем брать корень и два поддерева с минимальным количеством вершин.

h=1

**T1**

h=2

**T2**

h=3

**T3**

h=4

**T4**

T3

T2

**T5**

T3

T4

h=5

Рисунок 40 Деревья Фибоначчи

Одно поддерево должно быть высотой *h*–1, а другое высотой *h*–2. Поскольку принцип их построения очень напоминает построение чисел Фибоначчи, то такие деревья называют *деревьями Фибоначчи*: Th = < Th-1, x, Th-2 >. Число вершин в Th определяется следующим образом:

*n*0 = 0, *n*1 = 1, *nh* = *nh-*1 + 1 + *nh*-2

* 1. Повороты при балансировке

Рассмотрим, что может произойти при включении новой вершины в сбалансированное по высоте дерево. Пусть r – корень АВЛ-дерева, у которого имеется левое поддерево (ТL) и правое поддерево (TR). Если добавление новой вершины в левое поддерево приведет к увеличению его высоты на 1, то возможны три случая:

1. если hL = hR, то ТL и TR станут разной высоты, но баланс не будет нарушен;
2. если hL < hR, то ТL и TR станут равной высоты, т. е. баланс даже улучшится;
3. если hL > hR, то баланс нарушиться и дерево необходимо перестраивать.

Введём в каждую вершину дополнительный параметр Balance (показатель баланса), принимающий следующие значения:

-1, если левое поддерево на единицу выше правого;

0, если высоты обоих поддеревьев одинаковы;

1, если правое поддерево на единицу выше левого.

Если в какой-либо вершине баланс высот нарушается, то необходимо так перестроить имеющееся дерево, чтобы восстановить баланс в каждой вершине. Для восстановления баланса будем использовать процедуры поворотов АВЛ-дерева.

L

**1**

LL - поворот

p

L

q

**A**

T3

T2

T1

LL

=>

**A**

**B**

T1

T2

T3

p

0

0

-1 -2

**B**

0 -1

Рисунок 41 LL - поворот

***Алгоритм на псевдокоде***

*LL - поворот*

q := p→Left

q→Balance := 0

p→Balance := 0

p→Left := q→Right

q→Right := p

p := q

p

T1

T2

T3

T4

**B**

**A**

**C**

0

0

0

-1

+1

**2**

LR – поворот

p

q

L

R

r

T1

T2

T3

T4

LR

=>

**B**

**A**

**C**

-1

-2

0

0

+1

+1

-1

Рисунок 42 LR – поворот

***Алгоритм на псевдокоде***

*LR - поворот*

q := p→Left, r := q→Right

IF (r→Balance<0) p→Balance := +1 ELSE p→Balance := 0 FI

IF (r→Balance>0) q→Balance := –1 ELSE q→Balance := 0 FI

r→Balance := 0

p→Left := r→Right, q→Right := r→Left

r→Left := q, r→Right := p, p := r

T1

T2

RR

=>

T3

**B**

**A**

q

0

0

RR – поворот

**3**

p

T1

T2

T3

R

R

**B**

**A**

q

0

+1

+1

+2

Рисунок 43 RR – поворот

***Алгоритм на псевдокоде***

*RR - поворот*

q := p→Right

q→Balance := 0

p→Balance := 0

p→ Right:= q→ Left

q→ Left := p

p := q

RL – поворот

**4**

T1

T2

T3

T4

RL

=>

p

**A**

**B**

**C**

-1

0

0

0

+1

T1

T2

T3

T4

p

q

r

R

L

**A**

**C**

**B**

+1

+1

+2

-1

0

0

-1

Рисунок 44 RL – поворот

***Алгоритм на псевдокоде***

*RL - поворот*

q := p→ Right, r := q→ Left

IF (r→Balance>0) p→Balance := -1 ELSE p→Balance := 0 FI

IF (r→Balance<0) q→Balance := 1 ELSE q→Balance := 0 FI

r→Balance := 0

p→ Right:= r→ Left, q→ Left:= r→ Right

r→ Left := p, r→Right := q, p := r

* 1. Добавление вершины в дерево

Добавление новой вершины в АВЛ-дерево происходит следующим образом. Вначале добавим новую вершину в дерево так же как в случайное дерево поиска (проход по пути поиска до нужного места). Затем, двигаясь назад по пути поиска от новой вершины к корню дерева, будем искать вершину, в которой нарушился баланс (т. е. высоты левого и правого поддеревьев стали отличаться более чем на 1). Если такая вершина найдена, то изменим структуру дерева для восстановления баланса с помощью процедур поворотов.

***Алгоритм на псевдокоде***

*Добавление в АВЛ – дерево (D: данные; Var p: pVertex);*

Обозначим

Рост – логическая переменная, которая показывает выросло дерево или нет.

IF (p = NIL)

new(p), p→Data := D, p→Left := NIL, p→Right := NIL

p→Balance := 0, Рост := ИСТИНА

ELSE

IF (p→Data > D)

Добавление в АВЛ – дерево (D, p→Left)

IF (Рост = ИСТИНА) {выросла левая ветвь}

IF (p→Balance > 0) p→Balance := 0, Рост := ЛОЖЬ

ELSE IF (p→Balance = 0) p→Balance := -1

ELSE

IF (p→Left→Balance < 0) <LL – поворот>

ELSE <LR – поворот> Рост := ЛОЖЬ

FI

FI

ELSE IF (p→Data < D)

<аналогичные действия для правого поддерева

ELSE {p→Data = D, такая вершина уже есть}

FI

FI

FI

**Пример:** Построение АВЛ-дерева с вершинами B 9 2 4 1 7 E F A D C 3 5 8 6

RR

=>

**E**

**F**

**7**

**9**

**1**

**2**

**4**

**B**

**B**

**2**

**9**

LL

=>

LR

=>

**9**

**B**

**2**

**1**

**7**

**4**

RL

=>

RR

=>

**A**

**C**

**D**

**F**

**B**

**E**

**7**

**9**

**1**

**2**

**4**

**F**

**E**

**7**

**A**

**9**

**B**

**1**

**2**

**4**

**3**

**8**

**5**

**6**

LL

=>

**B**

**4**

**2**

**1**

**7**

**A**

**9**

**E**

**F**

**D**

**C**

**A**

**8**

**6**

**7**

**9**

**5**

**2**

**3**

**1**

**C**

**F**

**D**

**B**

**4**

**E**

Рисунок 45 Построение АВЛ-дерева

* 1. Удаление вершины из дерева

Очевидно, удаление вершины – процесс намного более сложный, чем добавление. Хотя алгоритм операции балансировки остаётся тем же самым, что и при включении вершины. Балансировка по-прежнему выполняется с помощью одного из четырёх уже рассмотренных поворотов вершин.

Удаление из АВЛ-дерева происходит следующим образом. Удалим вершину так же, как это делалось для СДП. Затем двигаясь назад от удалённой вершины к корню дерева, будем восстанавливать баланс в каждой вершине (с помощью поворотов). При этом нарушение баланса возможно в нескольких вершинах в отличие от операции включения вершины в дерево.

Как и в случае добавления вершин, введём логическую переменную *Уменьшение*, показывающую уменьшилась ли высота поддерева. Балансировка идёт, только если *Уменьшение* = истина. Это значение присваивается переменной *Уменьшение*, если обнаружена и удалена вершина или высота поддерева уменьшилась в процессе балансировки.

Введём две симметричные процедуры балансировки, т. к. они будут использоваться несколько раз в алгоритме удаления:

BL – используется при уменьшении высоты левого поддерева,

BR – используется при уменьшении высоты правого поддерева.

Рисунки 46 и 47 иллюстрируют три случая, возникающие при удалении вершины из левого (для BL) или правого (для BR) поддерева, в зависимости от исходного состояния баланса в вершине по адресу p.

***Алгоритм на псевдокоде***

BL (p: pVertex, Уменьшение: boolean)

IF (p→Bal = -1) p→Bal := 0

ELSEIF (p→Bal = 0) p→Bal := 1, Уменьшение := ЛОЖЬ

ELSEIF (p→Bal = 1)

IF (p→Left→Bal ≥ 0) <RR1-поворот>

ELSE <RL - поворот> FI

FI

\*)

p

-1

0

\*\*)

\*\*\*)

p

p

p

R

L

R

p→right

p→right

p→right

**Для BL**

p

0

1

Уменьшение := Ложь

R

1

1

0

1

-1

R

1

R

Рисунок 46 Три случая при удалении вершины из левого (для BL) поддерева

***Алгоритм на псевдокоде***

BR (p: pVertex, Уменьшение: boolean)

IF (p→Bal = 1) p→Bal := 0

ELSEIF (p→Bal = 0) p→Bal := -1, Уменьш := ЛОЖЬ

ELSEIF (p→Bal = -1)

IF (p→Left→Bal ≤ 0) <LL1 - поворот>

ELSE <LR - поворот> FI

FI

p→left

\*)

p

1

0

\*\*)

\*\*\*)

p

p

p

L

L

R

p→left

p→left

**Для BR**

p

0

-1

Уменьшение := Ложь

-1

L

-1

-1

0

1

-1

L

L

Рисунок 47 Три случая при удалении вершины правого (для BR) поддерева

При добавлении вершины не может быть случая, когда p→left→Bal = 0, поэтому LL – поворот необходимо изменить, чтобы учесть эту ситуацию.

***Алгоритм на псевдокоде***

*LL1 – поворот*

q := p→Left

IF (q→Bal = 0) p→Bal := -1, q→Bal := 1, Уменьш := false

ELSE p→Bal := 0, q→Bal := 0

p→Left := q→Right

q→Right := p

p := q

Аналогично изменяется RR – поворот, LR и RL – повороты не изменяются.

***Алгоритм на псевдокоде***

*RR1 – поворот*

q := p→Right

IF (q→Bal = 0) p→Bal := 1, q→Bal := -1, Уменьшение := ЛОЖЬ

ELSE p→Bal := 0, q→Bal := 0

FI

p→ Right:= q→ Left

q→ Left := p

p := q

***Алгоритм на псевдокоде***

*Удаление из АВЛ-дерева (x: Данные, p: pVertex, Уменьшение: boolean)*

IF (p = NIL) {ключа в дереве нет}

ELSE IF (p→Data > x) Удаление (x, p→Left, Уменьшение)

IF Уменьшение BL (p, Уменьш) FI

ELSE IF (p→Data < x) Удаление (x, p→Right, Уменьшение)

IF Уменьшение BR (p, Уменьшение) FI

ELSE IF {удаление вершины по адресу p}

q := p

IF (q→Right = NIL) p := q→Left, Уменьшение := ИСТИНА

ELSE IF (q→Left = NIL) p := q→Right, Уменьшение := ИСТИНА

ELSE del (q→Left, Уменьшение)

IF Уменьшение BL (p, Уменьшение) FI

FI

dispose(q)

FI

Используемая при удалении процедура del удаляет вершину, имеющую 2 поддерева, т. е. заменяет её на самую правую вершину из левого поддерева.

***Алгоритм на псевдокоде***

*del (r: pVertex, Уменьшение: boolean)*

IF (r→right ≠ NIL)

del (r→right, Уменьшение)

IF Уменьшение BR (r, Уменьшение) FI

ELSE q→Data := r→Data

q := r

r := r→Left

Уменьшение := ИСТИНА

FI

**Пример**: Удаление из АВЛ-дерева вершин B 9 2 4 1 7 E F

**B**

**E**

**F**

**D**

**C**

**A**

**7**

**1**

**9**

**2**

**4**

**A**

**2**

**7**

**4**

**9**

**1**

**C**

**F**

**D**

**E**

**C**

**F**

**E**

**D**

**7**

**1**

**4**

**A**

**2**

**1**

**7**

**C**

**D**

**F**

**E**

**A**

**4**

**F**

**E**

**C**

**D**

**A**

**7**

**1**

**C**

**7**

**FR**

**A**

RL

=>

R

L

**E**

**D**

**F**

**D**

**C**

**A**

**E**

**F**

**D**

**C**

**A**

**7**

**C**

**A**

**F**

**E**

**D**

LR

=>

R

L

**C**

**A**

**D**

**A**

**D**

**C**

Рисунок 48 Удаление из АВЛ-дерева

Поиск элемента с заданным ключом, включения нового элемента, удаления элемента – каждое из этих действий в АВЛ-дереве можно произвести в худшем случае за О(log *n*) операций.

Отличие между процедурами включения и удаления заключается в следующем. Включение может привести самое большое к одному повороту, исключение может потребовать поворот во всех вершинах вдоль пути поиска. Наихудшим случаем с точки зрения количества балансировок является удаление самой правой вершины у плохого АВЛ-дерева (дерева Фибоначчи). По экспериментальным оценкам на каждые два включения встречается один поворот, а при исключении поворот происходит в одном случае из пяти.

2.4 МЕТОД КОДИРОВАНИЯ

Метод Шеннона

Код Шеннонапозволяет построить почти оптимальный код с длинами кодовых слов *Li* < - log *pi*+1. Тогда *Lcp <H(p1, …,pn)*+1. Код Шеннона строится следующим образом*.*

1. Упорядочим символы исходного алфавита А={*a1,a2,…,an*} по убыванию их вероятностей: *p1≥p2≥p3≥…≥pn*.
2. Составим нарастающие суммы вероятностей *Qi*:

*Q0=0, Q1=p1, Q2=p1+p2, Q3=p1+p2+p3, … , Qn=1*.

1. Представим *Qi* в двоичной системе счисления и возьмем в качестве кодового слова первые - log2*pi* знаков после запятой .

**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 Код Шеннона

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *ai* | *Pi* | *Qi* | *Li* | кодовое слово |
| *a1*  *a2*  *a3*  *a4*  *a5*  *a6* | 1/22≤0.36<1/2  1/23≤0.18<1/22  1/23≤0.18<1/22  1/24≤0.12<1/23  1/24≤0.09<1/23  1/24≤0.07<1/23 | 0  0.36  0.54  0.72  0.84  0.93 | 2  3  3  4  4  4 | 00  010  100  1011  1101  1110 |

Построенный код является префиксным. Вычислим среднюю длину кодового слова и сравним ее с энтропией. Значение энтропии вычислено при построении кода Хаффмена (*H* = 2.37).

*Lср*= 0.36**.**2+(0.18+0.18)**.**3+(0.12+0.09+0.07)**.**4=2.92< 2.37+1

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ АЛГОРИТМОВ

В ходе выполнения курсовой работы, помимо основных алгоритмов, потребовалось реализовать также несколько вспомогательных, необходимых для корректной работы программы.

1. Интерфейс программы

Для организации интерфейса использовалась процедура void mainloop(), которая обеспечивает корректное и незатруднительное использование программы и предоставляет возможность многократного выбора различных вариантов обработки базы данных, в зависимости от задач пользователя.

2. Загрузка и вывод базы данных

Для загрузки базы данных разработана процедура Node \*load\_to\_memory(), в которой производится считывание записей типа struct Record, а из них формируется очередь Node record. Здесь же предусмотрена проверка на наличие файла, откуда выполняется считывание. Данная процедура вызывается независимо от желания пользователя, в то время как остальные он может выбрать посредствам меню.

После загрузки в динамическую память выполняется построение индексного массива по списку.

За вывод элементов считанной базы данных отвечает процедура void show\_list Она предоставляет возможность постраничного просмотра базы данных(по 20 элементов на странице), смена страниц осуществляется вводом «w» и «s». Есть возможность прервать просмотр при помощи ввода любой другой клавиши.

3. Вспомогательные функции и процедуры для сортировки данных

При сортировке базы данных потребовалось реализовать дополнительную функцию сравнения двух строк (int strcomp()), которая возвращает -1, если вторая строка больше, 1, если первая строка больше, 0, если они равны.

4. Особенности реализации бинарного поиска

Для того чтобы без проблем многократно осуществлять поиск элементов, соответствующих разным ключам создаётся индексный массив при помощи функции viod make index array, в котором можно икать множество раз.

5. Вспомогательные функции для построения АВЛ-дерева

Также как и для очереди, при неоднократном построении дерева требуется освобождать память, эту функцию выполняет процедура void rmtree. Для вывода дерева на экран используется процедура void Print\_tree, представляющая собой обход дерева слева – направо.

Аналогичная процедура void search\_in\_tree выполняет вывод результатов поиска в дереве.

6. Кодирование данных

При побуквенном кодировании существует необходимость знать вероятности встречаемости символов. Для их подсчета создана процедура void calc\_probabilities(), в которой происходит вычисление вероятностей на основе частоты встречающихся символов. Для создания кодовых слов используется функция void shannon().

4. ОПИСАНИЕ ПРОГРАММЫ

4.1. ОСНОВНЫЕ ПЕРЕМЕННЫЕ И СТРУКТУРЫ

int N = 4000;

struct Record {  
 char author[12];  
 char title[32];  
 char publisher[16];  
 short int year;  
 short int num\_of\_page;  
};

Запись, используемая для работы с базой данных «Запись».

int [N] –глобальная переменная для массива на N элементов.

struct Node {  
 Record record;  
 Node \*next;  
};

Структура (список), используемая при сортировке базы данных.

Node \*next **–** указатель на следующие элемент;

Record record **–** поле данных

struct Vertex { // структура дерева  
 Record \*data;  
 Vertex \*left;  
 Vertex \*right;  
};

Структура, представляющая дерево оптимального поиска (А2).

Vertex \*left, Vertex \*right – указатели на левое и правое поддеревья.

Record \*data - поле данных (адрес элемента в основном массиве структур).

Record records[N] – массив структур

Record \*arr[] – индексный массив.

Int \*w – массив весов.

Record \*ind\_arr[] индексный массив для построения дерева.

Record \*unsorted\_ind\_arr[N] – массив указателей на не отсортированные записи

4.2. ОПИСАНИЕ ПОДПРОГРАММ

Функция вывода меню:

1. void mainloop();

Функция отображает меню и принимает ввод пользователя, после чего запускает подпрограмму, зависящую от того, что ввёл пользователь.

Процедуры начальной обработки базы данных:

1. Node \*load\_to\_memory();
2. make\_index\_array(Record \*arr[], Node \*root, int n = N);

Load\_to\_memory – считывание базы из файла и представление ее элементов в форме вышеперечисленных структур, возвращает указатель на массив записей.

Make\_index\_array - создание массива указателей на список.

arr - индексный массив, в который будут записаны указатели на каждую из вершин. Root - список, состоящий из записей, соединённых между собой указателями.

Функции и процедуры сортировки:

1. **void quick\_sort**(Record \*array[], int n)
2. **int strcomp**(const string &str1, const string &str2, int len);

**Quick\_sort** – Функция, сортирующая список методом Хоара.

Array - сортируемый массив. Int n - количество элементов в массиве.

**strcomp** – определение операции сравнения двух строк (str1 и str2): размерностью len. Возвращает 1 в случае, если a > b и -1, если a < b и 0, если а = b.

Функции и процедуры для поиска в отсортированной базе данных:

1. **int quick\_search**(Record \*arr[], const string &key);
2. void search(Record \*arr[], int &ind, int &n);

**Quick\_search** – процедура двоичного поиска (версия 2), Record \*arr[] – указатель на массив записей, в котором осуществляется поиск, **key** – ключ поиска. Возвращает позицию найденного элемента и -1, в случае его отсутствия.

**search** – Фунцкия, вызываемая из основного меню. Запрашивает у пользователя ключ поиска и с ним вызывает функцию **Quick\_search.** Arr - отсортированный индексный массив. Ind - индекс первого вхождения ключа, n - количество вхождений.

Процедуры и функции построения АВЛ-дерева:

1. void rmtree(Vertex \*root);
2. void add\_to\_avl(Vertex \*&p, Record \*data);
3. void rl(Vertex \*&p);
4. void ll(Vertex \*&p);
5. void rr(Vertex \*&p);
6. void lr(Vertex \*&p);
7. void printTree(Vertex \*p);
8. void Search\_In\_Tree(Vertex \*root, int key);

**rmtree** – освобождение памяти для построения дерева, чтобы не возникало проблем, в случае если до этого дерево уже создавалось (**root** – указатель на корень дерева).

**Add\_to\_avl**– непосредственно построение, p - указатель на корень дерева. data - добавляемые данные.

**PrintTree** – обход дерева с корнем **p**, используемый для вывода на экран отсортированных по году элементов базы данных.

**Search\_In\_Tree** – поиск в дереве с корнем **root** элементов, соответствующих ключу key. Выводит все найденные ключи.

**lr, rl, ll, rr** - повороты в авл-дереве. p - указатель на нужную вершину

Процедуры и функции кодирования базы данных:

14. **unordered\_map**<char, int> get\_char\_counts\_from\_file(const string &file\_name, int &file\_size, int n = N);

20. vector<pair<double, char>> calc\_probabilities(const unordered\_map<char, int> &counter\_map, int count)

22. void **coding**()

23. void **shannon** (const int n, double p[], int Length[], char c[][20])

**Get\_char\_counts\_from\_file** – функция, подсчитывающая сколько раз каждый символ входит в базу данных, а также размер файла. File\_name - название файла, file\_size - размер файла, n - количество записей.

**calc\_probabilities** – Функция, подсчитывающая вероятности вхождения каждого символа. Возвращает вектор, содержащий пару: символ и количество его вхождений. Counter\_map - хеш таблица, ключом в которой является символ, а значением - сколько раз этот символ входит в БД.

**coding** – Основная функция для кодировки. Вызывается из главного меню.

**shannon**– Функция, кодирующая символы методом Шеннона, согласно их вероятностям. **N** - количество различных символов, p - массив, содержащий вероятности этих символов, **Length** - массив, содержащий длины кодовых слов, char c[][20] - массив, содержащий коды символов.

Основная программа

int **main**() – в основной программе вызывается только меню.

5. ТЕКСТ ПРОГРАММЫ

#include <iostream>

#include <iomanip>

#include <fstream>

#include <unordered\_map>

#include <vector>

#include <cmath>

#include <algorithm>

using namespace std;

int N = 4000;

struct Record {

char author[12];

char title[32];

char publisher[16];

short int year;

short int num\_of\_page;

};

struct Node {

Record record;

Node \*next;

};

void print\_head() {

cout << "Record Author Title Publisher Year Num of pages\n";

}

void print\_record(Record \*record, int i) {

cout << "[" << setw(4) << i << "] ";

cout << record->author

<< " " << record->title

<< " " << record->publisher

<< " " << record->year

<< " " << record->num\_of\_page << "\n";

}

int strcomp(const string &str1, const string &str2, int len = 10000000) {

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

if (str1[i] == '\0' and str2[i] == '\0') {

return 0;

} else if (str1[i] == ' ' and str2[i] != ' ') {

return -1;

} else if (str1[i] != ' ' and str2[i] == ' ') {

return 1;

} else if (str1[i] < str2[i]) {

return -1;

} else if (str1[i] > str2[i]) {

return 1;

}

}

return 0;

}

int compare(const Record &record1, const Record &record2) {

if (record1.year > record2.year) {

return 1;

} else if (record1.year < record2.year) {

return -1;

} else {

return strcomp(record1.author, record2.author);

}

}

void qSort(Record \*array[], int L, int R) {

while (L < R) {

Record \*x = array[(L + R) / 2];

int i = L, j = R;

while (i < j) {

while (compare(\*array[i], \*x) < 0) {

++i;

}

while (compare(\*array[j], \*x) > 0) {

--j;

}

if (i <= j) {

Record \*temp = array[i];

array[i] = array[j];

array[j] = temp;

++i;

--j;

}

}

if (j - L < R - i) {

qSort(array, L, j);

L = i;

} else {

qSort(array, i, R);

R = j;

}

}

}

void quickSort(Record \*array[], const int N) {

qSort(array, 0, N - 1);

}

string prompt(const string &str) {

cout << str;

cout << "\n> ";

string ans;

cin >> ans;

return ans;

}

void load\_to\_memory(Record records[]) {

ifstream file("testBase1.dat", ios::binary);

if (not file.is\_open()) {

cout << "Could not open file\n";

}

file.read((char \*) records, sizeof(Record) \* N);

file.close();

}

struct Vertex {

Record \*data;

Vertex \*left;

Vertex \*right;

int balance;

};

bool grow;

void ll(Vertex \*&p) {

Vertex \*q = p->left;

p->balance = q->balance = 0;

p->left = q->right;

q->right = p;

p = q;

}

void rr(Vertex \*&p) {

Vertex \*q = p->right;

p->balance = q->balance = 0;

p->right = q->left;

q->left = p;

p = q;

}

void lr(Vertex \*&p) {

Vertex \*q = p->left;

Vertex \*r = q->right;

if (r->balance < 0) {

p->balance = 1;

} else {

p->balance = 0;

}

if (r->balance > 0) {

q->balance = -1;

} else {

q->balance = 0;

}

r->balance = 0;

q->right = r->left;

p->left = r->right;

r->left = q;

r->right = p;

p = r;

}

void rl(Vertex \*&p) {

Vertex \*q = p->right;

Vertex \*r = q->left;

if (r->balance > 0) {

p->balance = -1;

} else {

p->balance = 0;

}

if (r->balance < 0) {

q->balance = 1;

} else {

q->balance = 0;

}

r->balance = 0;

q->left = r->right;

p->right = r->left;

r->right = q;

r->left = p;

p = r;

}

void add\_to\_avl(Vertex \*&p, Record \*data) {

if (!p) {

p = new Vertex{data, nullptr, nullptr, 0};

grow = true;

} else if (strcomp(p->data->title, data->title) > 0) {

add\_to\_avl(p->left, data);

if (grow) {

if (p->balance > 0) {

p->balance = 0;

grow = false;

} else if (p->balance == 0) {

p->balance = -1;

grow = true;

} else {

if (p->left->balance < 0) {

ll(p);

grow = false;

} else {

lr(p);

grow = false;

}

}

}

} else if (strcomp(p->data->title, data->title) <= 0) {

add\_to\_avl(p->right, data);

if (grow) {

if (p->balance < 0) {

p->balance = 0;

grow = false;

} else if (p->balance == 0) {

p->balance = 1;

grow = true;

} else {

if (p->right->balance > 0) {

rr(p);

grow = false;

} else {

rl(p);

grow = false;

}

}

}

} else {

std::cout << "Data already exist";

}

}

void Print\_tree(Vertex \*p, int &i) {

if (p) {

Print\_tree(p->left, i);

print\_record(p->data, i++);

Print\_tree(p->right, i);

}

}

void search\_in\_tree(Vertex \*root, const string &key, int &i) {

if (root) {

if (strcomp(key, root->data->title, key.length()) < 0) {

search\_in\_tree(root->left, key, i);

} else if (strcomp(key, root->data->title, key.length()) > 0) {

search\_in\_tree(root->right, key, i);

} else if (strcomp(key, root->data->title, key.length()) == 0) {

search\_in\_tree(root->left, key, i);

print\_record(root->data, i++);

search\_in\_tree(root->right, key, i);

}

}

}

void rmtree(Vertex \*root) {

if (root) {

rmtree(root->right);

rmtree(root->left);

delete root;

}

}

void tree(Record \*arr[], int n) {

Vertex \*root = nullptr;

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

add\_to\_avl(root, arr[i]);

}

print\_head();

int i = 1;

Print\_tree(root, i);

string key;

getline(cin, key);

do {

cout << "Input search key (title), q - exit\n> ";

getline(cin, key);

if (!key.empty() && key != "q") {

print\_head();

i = 1;

search\_in\_tree(root, key, i);

}

} while (key[0] != 'q');

rmtree(root);

}

void show\_list(Record \*records[], int n = N) {

int ind = 0;

while (true) {

print\_head();

for (int i = ind; i < ind + 20 && i < n; i++) {

Record \*record = records[i];

print\_record(record, i + 1);

}

string chose = prompt("w: Next page\t"

"q: Last page\t"

"e: Skip 10 next pages\n"

"s: Prev page\t"

"a: First page\t"

"d: Skip 10 prev pages\n"

"Any key: Exit");

switch (chose[0]) {

case 'w':

ind += 20;

break;

case 's':

ind -= 20;

break;

case 'a':

ind = 0;

break;

case 'q':

ind = n / 20 \* 20;

break;

case 'd':

ind -= 200;

break;

case 'e':

ind += 200;

break;

default:

return;

}

if (ind < 0) {

ind = 0;

}

while (ind >= n) {

ind -= 20;

}

}

}

int quick\_search(Record \*arr[], int key) {

int l = 0;

int r = N - 1;

while (l < r) {

int m = (l + r) / 2;

if (arr[m]->year < key) {

l = m + 1;

} else {

r = m;

}

}

if (arr[r]->year == key) {

return r;

}

return -1;

}

void make\_index\_array(Record arr[], Record \*ind\_arr[], int n = N) {

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

ind\_arr[i] = &arr[i];

}

}

void make\_index\_array(Record \*arr[], Node \*root, int n = N) {

Node \*p = root;

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

arr[i] = &(p->record);

p = p->next;

}

}

void search(Record \*arr[], int &ind, int &n) {

Node \*head = nullptr, \*tail = nullptr;

int key;

while (true) {

try {

key = stoi(prompt("Input search key (year)"));

break;

} catch (invalid\_argument &exc) {

cout << "Please input a number\n";

continue;

}

}

ind = quick\_search(arr, key);

if (ind == -1) {

cout << "Not found\n";

} else {

head = new Node{\*arr[ind], nullptr};

tail = head;

int i;

for (i = ind + 1; arr[i]->year == key; ++i) {

tail->next = new Node{\*arr[i], nullptr};

tail = tail->next;

}

n = i - ind;

Record \*find\_arr[n];

make\_index\_array(find\_arr, head, n);

show\_list(find\_arr, n);

}

}

void shannon(const int n, double p[], int Length[], char c[][20]) {

double q[n];

q[0] = 0;

Length[0] = -floor(log2(p[0]));

for (int i = 1; i < n; ++i) {

q[i] = q[i - 1] + p[i - 1];

Length[i] = -floor(log2(p[i]));

}

for (int i = 1; i <= n; ++i) {

for (int j = 0; j < Length[i - 1]; ++j) {

q[i - 1] \*= 2;

c[i - 1][j] = floor(q[i - 1]) + '0';

if (q[i - 1] >= 1) {

q[i - 1] -= 1;

}

}

}

}

unordered\_map<char, int> get\_char\_counts\_from\_file(const string &file\_name, int &file\_size, const int n = N) {

ifstream file(file\_name, ios::binary);

char ch\_arr[sizeof(Record) \* n];

file.read((char \*) ch\_arr, sizeof(Record) \* n);

file.close();

unordered\_map<char, int> counter\_map;

file\_size = 0;

for (auto ch : ch\_arr) {

counter\_map[ch]++;

file\_size++;

}

return counter\_map;

}

vector<pair<double, char>> calc\_probabilities(const unordered\_map<char, int> &counter\_map, int count) {

vector<pair<double, char>> probabilities;

probabilities.reserve(counter\_map.size());

for (auto i : counter\_map) {

probabilities.emplace\_back(make\_pair((double) i.second / count, i.first));

}

return probabilities;

}

void coding() {

int file\_size;

unordered\_map<char, int> counter\_map;

counter\_map = get\_char\_counts\_from\_file("testBase1.dat", file\_size);

auto probabilities = calc\_probabilities(counter\_map, file\_size);

counter\_map.clear();

sort(probabilities.begin(), probabilities.end(), greater<pair<char, int>>());

cout << "Probabil. char\n";

for (auto i : probabilities) {

cout << fixed << i.first << " | " << i.second << '\n';

}

const int n = (int) probabilities.size();

auto c = new char[n][20];

auto Length = new int[n];

auto p = new double[n];

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

p[i] = probabilities[i].first;

}

double shen = 0;

shannon(n, p, Length, c);

cout << "\nShannon Code:\n";

cout << "\nCh Prob Code\n";

double avg\_len = 0;

double entropy = 0;

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

avg\_len += Length[i] \* p[i];

entropy -= p[i] \* log2(p[i]);

printf("%c | %.5lf | ", probabilities[i].second, p[i]);

for (int j = 0; j < Length[i]; ++j) {

printf("%c", c[i][j]);

}

cout << '\n';

shen += p[i];

}

cout << "Average length = " << avg\_len << '\n'

<< "Entropy = " << entropy << '\n'

<< "Average length < entropy + 1\n"

<< "N = " << n << '\n';

}

void mainloop(Record \*unsorted\_ind\_array[], Record \*sorted\_ind\_array[]) {

int search\_ind, search\_n = -1;

while (true) {

string chose = prompt("1: Show unsorted list\n"

"2: Show sorted list\n"

"3: Search\n"

"4: Tree\n"

"5: Coding\n"

"Any key: Exit");

switch (chose[0]) {

case '1':

show\_list(unsorted\_ind\_array);

break;

case '2':

show\_list(sorted\_ind\_array);

break;

case '3':

search(sorted\_ind\_array, search\_ind, search\_n);

break;

case '4':

if (search\_n == -1) {

cout << "Please search first\n";

} else {

tree(&sorted\_ind\_array[search\_ind], search\_n);

}

break;

case '5':

coding();

break;

default:

return;

}

}

}

int main() {

Record records[N];

load\_to\_memory(records);

Record \*unsorted\_ind\_arr[N];

make\_index\_array(records, unsorted\_ind\_arr);

Record \*sorted\_ind\_arr[N];

make\_index\_array(records, sorted\_ind\_arr);

quickSort(sorted\_ind\_arr, N);

mainloop(unsorted\_ind\_arr, sorted\_ind\_arr);

}

6. РЕЗУЛЬТАТЫ

Рисунок 1. Неотсортированная база данных.



Рисунок 2. Отсортированная по году и автору база данных.

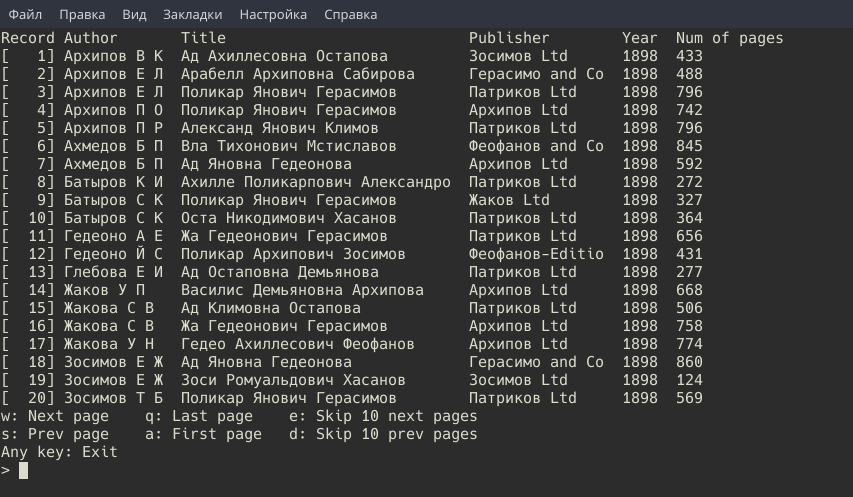


Рисунок 3. Очередь из записей, полученных в результате поиска (1984).

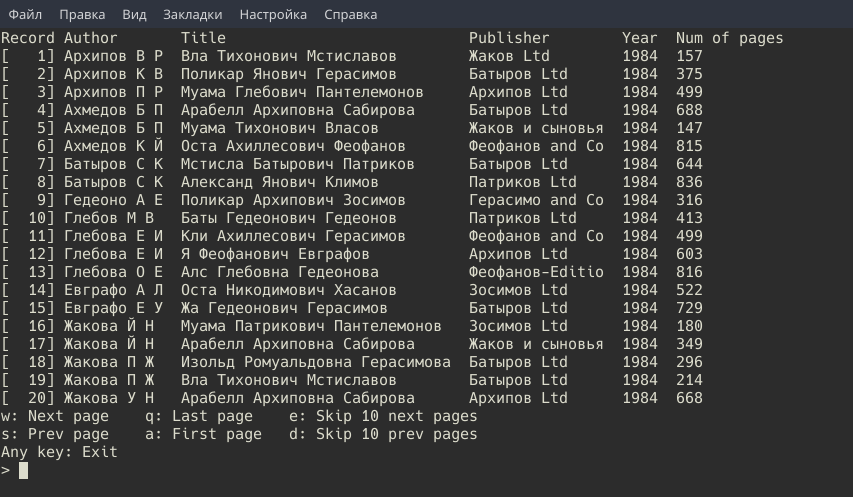


Рисунок 4. Дерево, ключ в дереве – по названию.

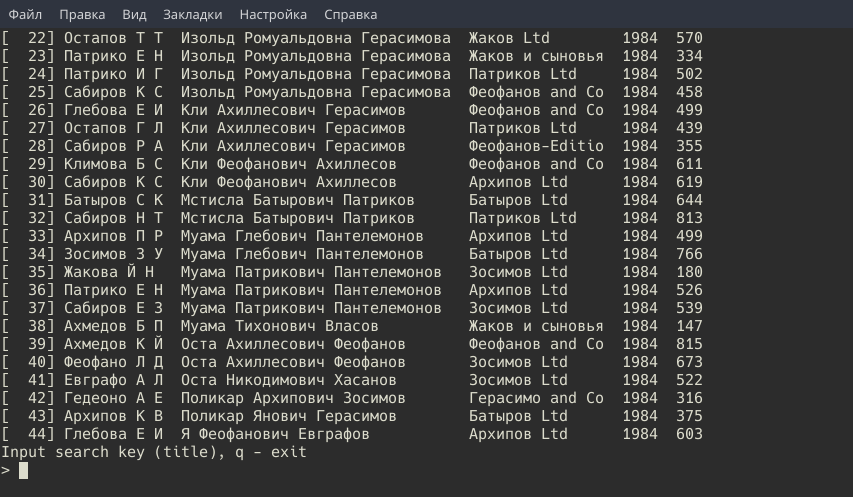


Рисунок 5. Поиск по дереву названию.

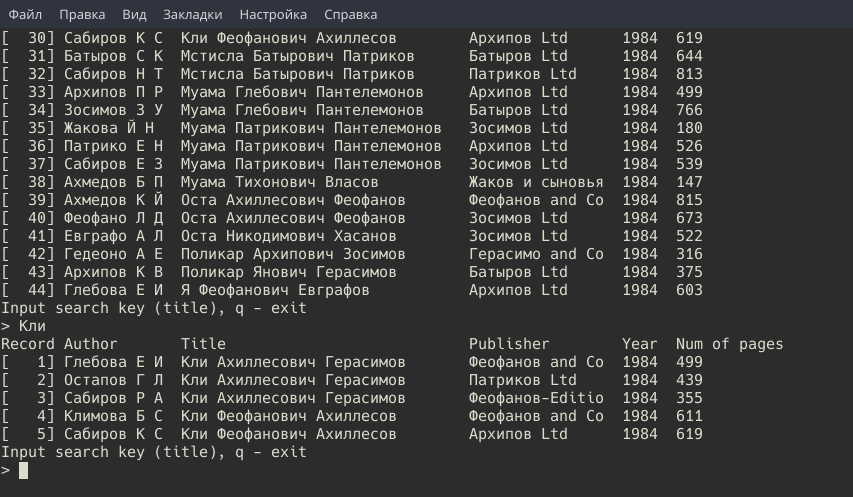
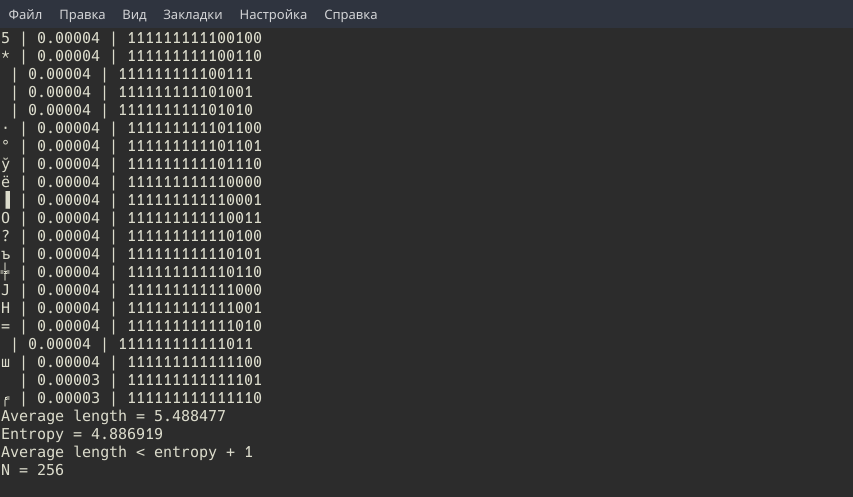


Рисунок 6. Примеры кодовых слов, энтропия и средняя длина кодового слова.



7. ВЫВОДЫ

В ходе выполнения курсовой работы были выполнены все поставленные задачи и реализованы необходимые алгоритмы: сортировки, поиска, построения дерева, поиска по дереву и кодирование базы данных.

В результате кодирования были получены данные подтверждающие теоретические сведения. К таковым относятся: величины средней длины кодового слова и энтропии (Lср ≤ H + 2).

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

Реализованные алгоритмы представляют минимальный набор функций для представления и обработки базы данных, а также отличаются достаточно высоким быстродействием и эффективностью.