Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации
ФГБОУ высшего образования
"Сибирский Государственный Университет Телекоммуникаций и Информатики"(СибГУТИ)
Кафедра прикладной математики и кибернетики

Курсовая работа
по дисциплине «Структуры и алгоритмы обработки данных»

Выполнил: студент 2 курса группы ИП-012
Маланов Роман Игоревич
Проверил: доцент кафедры ПМиК
Янченко Елена Викторовна

Вариант 47.

Новосибирск, 2021

Содержание

1 Постановка задачи

Хранящуюся в файле базу данных (база №4 "Населённый пункт") загрузить в память компьютера в виде списка, подготовить процедуры вывода на экран по 20 строк (записей) с возможностью отказа от просмотра.

Упорядочить данные, используя метод прямого слияния (Merge sort). Упорядоченные данные вывести на экран. Сортировка производится по дате поселения и названию улицы.

Предусмотреть возможность быстрого поиска по ключу в упорядоченной базе, в результате которого из результатов поиска формируется очередь, содержимое очереди выводится на экран.

Из записей очереди построить дерево поиска (двоичное Б-дерево) по ключу, отличному от ключа сортировки (используется поле "Номер дома"). Вывести на экран содержимое дерева и предусмотреть возможность поиска в дереве по запросу.

Закодировать файл базы данных статическим кодом (кодом Фано), предварительно оценив вероятности всех встречающихся символов. Построенный код вывести на экран, вычислить среднюю длину кодового слова и сравнить её с энтропией исходного файла.

2 Основные идеи и характеристики применяемых методов

2.1 Метод сортировки MergeSort

Сортировка слиянием (merge sort) – алгоритм сортировки, который упорядочивает списки или иные структуры с последовательным доступом к данным. В основе метода лежит операция слияния серий.

Сложность алгоритма определяется сложностью операции слияния серий. Асимптотическая оценка сложности: $O(n \log n)$ при $n \to \infty$.

Алгоритм обеспечивает устойчивую сортировку. При реализации со списками дополнительной памяти не требует.

2.2 Двоичный поиск

Двоичный (бинарный) поиск – классический алгоритм поиска элемента в отсортированном массиве, использующий дробление массива на половины.

Оценка сложности алгоритма: $O(\log n)$ при $n \to \infty$.

2.3 Списки и очереди

Список – абстрактный тип данных, реализующий упорядоченный набор значений. Списки отличаются от массивов последовательным, а не произвольным доступом к элементам.

В данной работе используются связные списки и очереди.

Связный список – это структура данных, представляющая собой конечное множество упорядоченных элементов, связанных друг с другом посредством указателей. Каждый элемент содержит поле с данными (или указателем на данные) и ссылку (указатель) на следующий элемент.

Очередь – это список, добавление элементов в который допустимо лишь в один его конец, а извлечение производится с другого конца (принцип организации FIFO - «первым пришёл – первым вышел»).

Сложность доступа: $O(n)$ при $n \to \infty$.

Сложность добавления элемента: $O(1)$.

2.4 Вид дерева и поиск

Двоичное Б-дерево – вид сильноветвящегося дерева, Б-дерево первого порядка. Состоит из вершин (страниц) с одним или двумя элементами. Следовательно, каждая страницм содержит две или три ссылки на поддеревья.

Двоичные Б-деревья представляют собой альтернативу АВЛ-деревьям. При этом поиск происходит как в обычном дереве.

Сложность поиска в двоичном Б-дереве и в АВЛ-дереве одинакова по порядку величины ($O(\log n)$).

2.5 Метод кодирования

Код Фано – метод построения почти оптимального кода, для которого $L_{\text{ср.}} < H(p1, ..., p_n) + 1$. Метод заключается в следующем. Упорядоченный по убыванию вероятностей список букв алфавита источника делится на две части так, чтобы суммы вероятностей букв, входящих в эти части, как можно меньше отличались друг от друга. Буквам первой части приписывается 0, а буквам из второй части – 1. Далее также поступают с каждой из полученных частей. Процесс продолжается до тех пор, пока весь список не разобьется на части, содержащие по одной букве.

3 Описание структур данных и использованных алгоритмов

3.1 Использованные структуры данных

Основные использованные в работе структуры данных, это структура, описывающая одну строку базы данных (запись - struct record), структуры списка и очереди.

- struct record

  Описывает элемент базы данных, состоящий из полей: ФИО (person), название улицы (street), номер дома (house), номер квартиры (apt), и дата заселения (date).

  В коде используется как тип данных Record.

- struct node

  Структура, описывающая элемент списка, содержащая указатель на запись в базе данных (указатель на тип Record), и ссылку на следующий элемент списка (реализует односвязный список). В коде используется как тип данных List и как элемент очереди Que.

- struct Que

  Структура, описывающая очередь. Содержит элементы-указатели на начало списка (head) и на конец списка (tail). Элементом очереди является структура struct node.

4 Описание программы

4.1 Описание подпрограмм

Программа разбита на модули. Основной файл – main.c, в котором программа запускается, содержит функции вывода на экран меню, списка записей, очереди, и функции запуска определённых частей программы.

Функция сортировки, Que mergeSort(List**, int N, tSortType), расположенная в файле mergeSort.h, принимает на вход список элементов, количество элементов, и тип сортировки (элемент перечисления tSortType). Сортировка изменяет сам список и возвращает созданную очередь для дальнейших действий в основной программе.

Функция Que search(Que que, Record** indexArr, int size, int year) выполняет бинарный поиск по году year в очереди с помощью предварительно созданного индексного массива indexArr. Создаёт и возвращает очередь из найденных элементов.

Рекурсивная функция pTree addDBTree(Record*, pTree), расположенная в файле dbtree.h, принимает на вход указатель на данные, которые необходимо добавить в дерево, и корень дерева, указатель на элемент структуры pTree (структура так же описана в файле dbtree.h и кроме указателей на левое и правое поддерево содержит указатель на список для добавления одинаковых элементов, чтобы не допускать потери данных при построении дерева).

Процедура void fano(float* P, int L, int R, int k, int** C, int* Length) выполняет построение кода Фано по массиву вероятностей float* P. Изменяет массив с готовыми кодами int** C и массив длин кодов int* Length.

5 Скриншоты

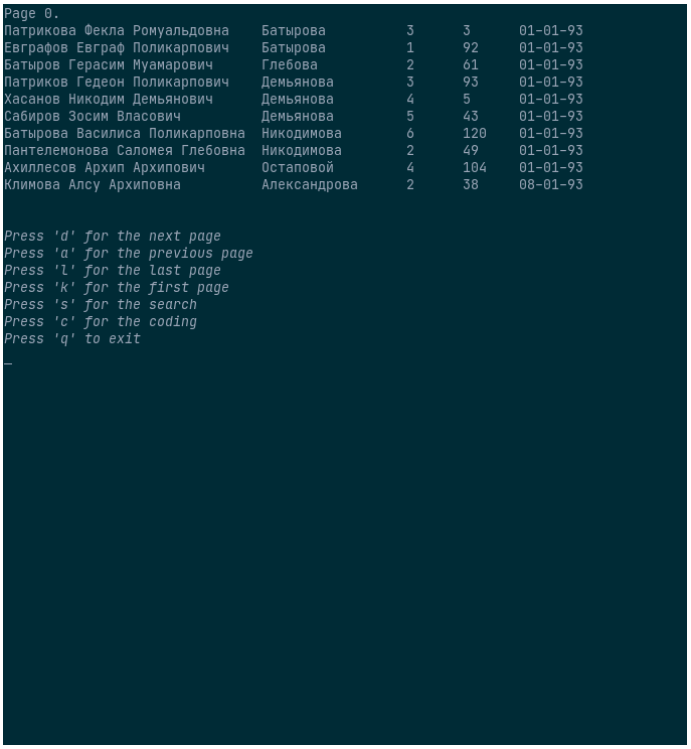

Рис. 1: Начальное состояние программы. Выведен список записей на экран с возможностью пролистывать записи и запустить подпрограммы

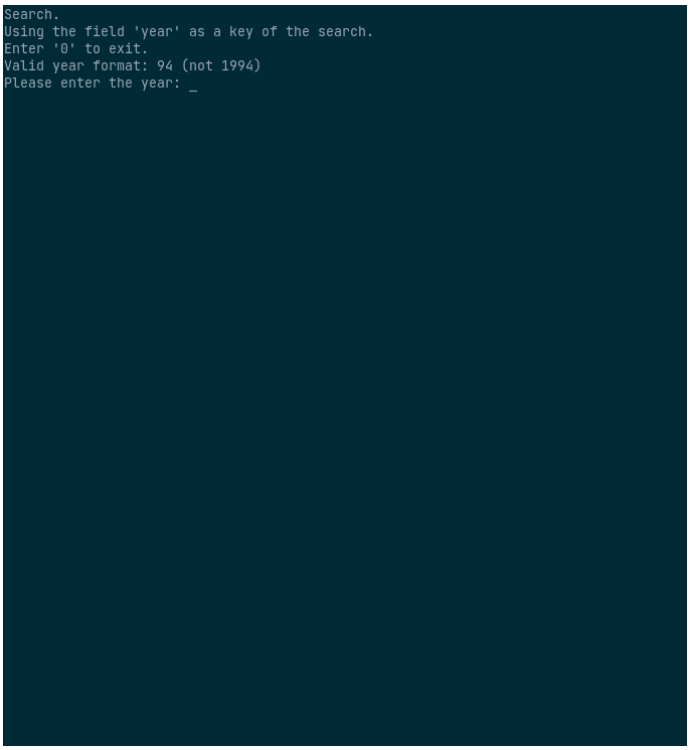

Рис. 2: Запрос года для запуска поиска

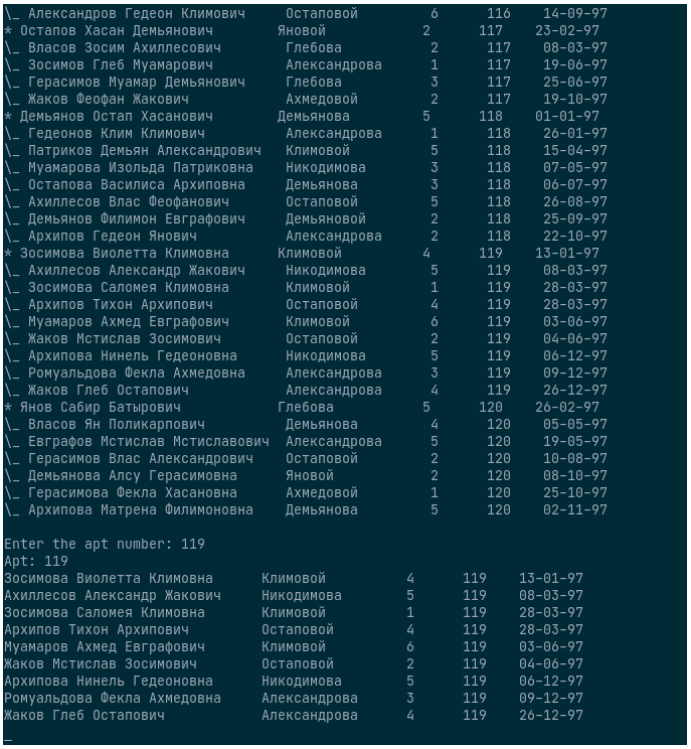

Рис. 3: Построенное дерево записей и поиск по дереву по номеру квартиры

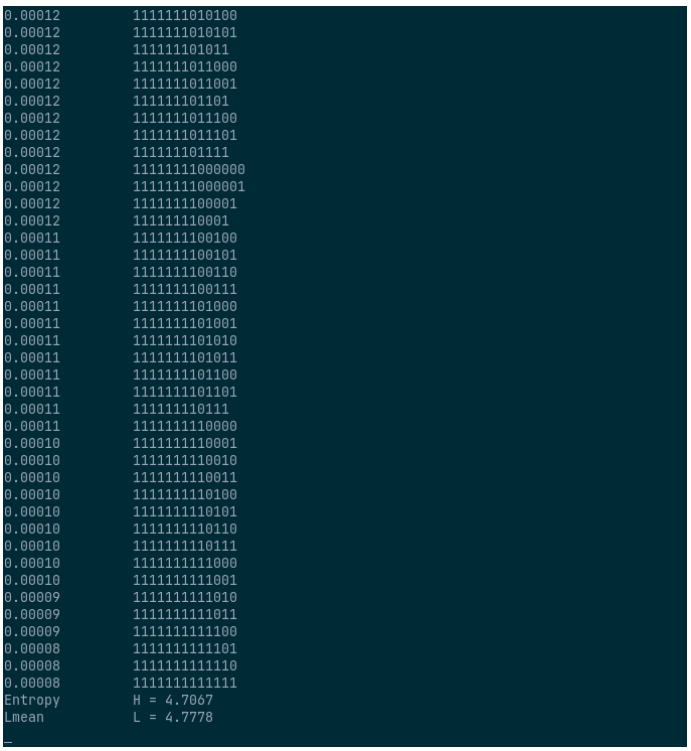

Рис. 4: Построенный код Фано для файла базы данных

6 Выводы

В результате была реализована программа, считывающая базу данных из файла, с возможностью сортировки записей, поиска по базе, построения дерева, поиска по построенному дереву, и

кодировки файла базы методом Фано.

7 Код программы

language: C

struct.h

```c
#ifndef STRUCT_H
#define STRUCT_H

struct record {
    char person[32];
    char street[18];
    short int house;
    short int apt;
    char date[10];
};

typedef struct record2 {
    char person[32*2];
    char street[18*2];
    short int house;
    short int apt;
    char date[10*2];
} Record;

typedef struct node Node;
struct node {
    Record* pdata;
    Node* next;
};

typedef enum SORTTYPE {
  sortDate,
  sortStreet,
  sortDateStreet
} tSortType;

struct dict {
  unsigned char c;
  int n;
  char* code;
};

#endif // #ifndef STRUCT_H
```

list.h

```c
#ifndef LIST_H
#define LIST_H

#include <stdlib.h>
#include <stdio.h>

#include "struct.h"

typedef Node List;

List *addList(List *head, Record *record) {
  List *new_node = calloc(1, sizeof(List));
  new_node->next = head;
```

```
14    new_node->pdata = record;
15    return new_node;
16  }
17  #endif
```

que.h

```
1   #ifndef QUEUE_H
2   #define QUEUE_H
3
4   #include <stdlib.h>
5   #include <stdio.h>
6
7   #include "struct.h"
8
9   typedef Node Qnode;
10
11  typedef struct _queue {
12    Qnode *head;
13    Qnode *tail;
14  } Que;
15
16  int isEmptyQ(Que que) {
17    if (que.tail == que.head)
18        return 1;
19    return 0;
20  }
21
22  Que createQue() {
23    Que que;
24    que.head = NULL;
25    que.tail = que.head;
26    return que;
27  }
28
29  Que addQue(Que que, Record* record) {
30    Qnode *new_node = calloc(1, sizeof(Qnode));
31    new_node->pdata = record;
32    // TODO: вынестиизфункции , используетсятолькооднажды
33    if (que.head == NULL) {
34      que.head = new_node;
35      que.tail = new_node;
36    } else {
37      que.tail->next = new_node;
38      que.tail = new_node;
39    }
40    return que;
41  }
42
43  int sizeQue(Que que) {
44    Qnode* p = que.head;
45    int size = 0;
46    while (p != que.tail && p != NULL) {
47      ++size;
48      p = p->next;
49    }
50    return size;
51  }
52
```

```
53  #endif
```

search.h

```
1   #ifndef SEARCH_H
2   #define SEARCH_H
3
4   #include "struct.h"
5   #include "que.h"
6
7   #include <stdio.h>
8
9   int getYear(Record* rec) {
10    if (rec && rec->date) {
11      return (rec->date[6]-'0')*10+(rec->date[7]-'0');
12    }
13    return 0;
14  }
15
16  Que createQueSort(Que queSort, Que mergedQue, Record** indexArr, int idxStart, int
        size) {
17    Qnode* p;
18          p = mergedQue.head;
19
20    for (int i = 0; i < idxStart; ++i)
21      p = p->next;
22    queSort.head = p;
23
24    int year = getYear(indexArr[idxStart]);
25    for (int i = idxStart; i < size; ++i, p = p->next) {
26      if (getYear(indexArr[i]) != year) {
27        queSort.tail = p;
28        break;
29      }
30    }
31    return queSort;
32  }
33
34  Que search(Que mergedQue, Record** indexArr, int size, int year) {
35    Que que = createQue();
36    int L = 0, R = size-1, m;
37    while (L < R) {
38      m = (L+R)/2;
39      if (getYear(indexArr[m]) < year)
40        L = m+1;
41      else
42        R = m;
43    }
44    if (getYear(indexArr[R]) == year) {
45      printf("Найден:\n");
46      Record* record = indexArr[R];
47      que = createQueSort(que, mergedQue, indexArr, R, size);
48    } else {
49      printf("Не найден.\n");
50    }
51    return que;
52  }
53
54  #endif
```

```
1  #ifndef MERGE_SORT
2  #define MERGE_SORT
3
4  #include "list.h"
5  #include "que.h"
6
7  #include <stdio.h>
8  #include <stdlib.h>
9
10 int isStreetLessEq(Record* a, Record* b) {
11   for (int i = 0; i < 18*2; ++i) {
12     if (a->street[i] > b->street[i])
13       return 0;
14     if (a->street[i] < b->street[i])
15       return 1;
16   }
17   return 2;
18 }
19
20 int isDateLessEq(Record* a, Record* b) {
21   for (int i = 2; i >= 0; --i) {
22     if (a->date[i*3] > b->date[i*3])
23       return 0;
24     if (a->date[i*3] < b->date[i*3])
25       return 1;
26     if (a->date[i*3+1] > b->date[i*3+1])
27       return 0;
28     if (a->date[i*3+1] < b->date[i*3+1])
29       return 1;
30   }
31   return 2;
32 }
33
34 int isDateStreetLessEq(Record* a, Record* b) {
35   int answer;
36   answer = isDateLessEq(a, b);
37   if (answer == 2)
38       answer = isStreetLessEq(a, b);
39   return answer;
40 }
41
42 void fromLtoQ(List **list, Que *que) {
43   if ((*que).head == NULL) {
44     addQue(*que, NULL);
45     (*que).head = *list;
46     (*que).tail = (*que).head;
47   } else {
48     (*que).tail->next = *list;
49     (*que).tail = *list;
50   }
51   (*list) = (*list)->next;
52 }
53
54 void mergeSeries(List **a, int q, List **b, int r, Que *c, int (*funcComp)(Record*,
      Record*)) {
55   while (q != 0 && r != 0) {
56     if (funcComp((*a)->pdata, (*b)->pdata)) {
```

```
57          fromLtoQ(a, c);
58          q--;
59        } else {
60          fromLtoQ(b, c);
61          r--;
62        }
63      }
64      while (q > 0) {
65        fromLtoQ(a, c);
66        q--;
67      }
68      while (r > 0) {
69        fromLtoQ(b, c);
70        r--;
71      }
72    }
73
74    void splitList(List **S, List **a, List **b, int N) {
75      *a = *S;
76      *b = (*S)->next;
77      int n = 0;
78      List *k = *a;
79      List *p = *b;
80
81      while (p != NULL) {
82        n++;
83        k->next = p->next;
84        k = p;
85        p = p->next;
86      }
87    }
88
89    Que mergeSort(List **S, int N, tSortType sortType) {
90      int i, m, q, r;
91
92      int p;
93      List *a, *b;
94      Que c0, c1;
95
96      splitList(S, &a, &b, N);
97
98      p = 1;
99      while (p < N) {
100       c0 = createQue();
101       c1 = createQue();
102
103       i = 0; m = N;
104       while (m > 0) {
105         q = (m >= p) ? p : m;
106         m = m - q;
107         r = (m >= p) ? p : m;
108         m = m - r;
109
110         mergeSeries(&a, q, &b, r, i ? &c1 : &c0,
111             sortType == sortDate ? isDateLessEq :
112               sortType == sortStreet ? isStreetLessEq :
113                 isDateStreetLessEq);
114         i = 1 - i;
115       }
```

```
116       a = c0.head;
117       b = c1.head;
118       p = 2*p;
119     }
120     c0.tail->next = NULL;
121     *S = c0.head;
122     return c0;
123   }
124
125   #endif
```

insertSort.h

```
1   #ifndef INSERTSORT_H
2   #define INSERTSORT_H
3
4   #include "struct.h"
5
6   int insertSort(struct dict* A, int N)
7   {
8       int i, j;
9       struct dict t;
10      for (i = 1; i < N; i++) {
11          j = i - 1;
12          t = A[i];
13          while ((j >= 0) && (t.n > A[j].n)) {
14              A[j + 1] = A[j];
15              j -= 1;
16          }
17          A[j + 1] = t;
18      }
19  }
20
21  #endif
```

dbtree.h

```
1   #ifndef BTREE_H
2   #define BTREE_H
3
4   #include "search.h"
5
6   #define YES 1
7   #define NO 0
8
9   typedef struct tree* pTree;
10  struct tree {
11    pTree left;
12    pTree right;
13    pTree next;
14    Record* pdata;
15    int bal;
16  };
17
18  int getApt(Record* rec) {
19    int apt = rec->apt;
20    return apt;
21  }
22
23  pTree addDBtree(Record* pdata, pTree p) {
```

```
24      int VR = YES;
25      int HR = YES;
26      pTree q;
27
28      if (p == NULL) {
29        p = calloc(1, sizeof(struct tree));
30        p->pdata = pdata;
31        p->left = NULL; p->right = NULL; p->next = NULL;
32        p->bal = 0;
33        VR = YES;
34      }
35      else {
36        if (getApt(p->pdata) > getApt(pdata)) {
37          p->left = addDBtree(pdata, p->left);
38          if (VR == YES) {
39            if (p->bal == 0) {
40              q = p->left;
41              p->left = q->right;
42              q->right = p;
43              p = q;
44              q->bal = 1;
45              VR = NO;
46              HR = YES;
47            }
48            else {
49              p->bal = 0;
50              HR = YES;
51            }
52          }
53          else
54            HR = NO;
55        }
56        else {
57          if (getApt(p->pdata) < getApt(pdata)) {
58            p->right = addDBtree(pdata, p->right);
59            if (VR == YES) {
60              p->bal = 1;
61              VR = NO;
62              HR = YES;
63            }
64            else {
65              if (HR == YES) {
66                if (p->bal > 0) {
67                  q = p->right;
68                  p->right = q->left;
69                  p->bal = 0;
70                  q->bal = 0;
71                  p->left = p;
72                  p = q;
73                  VR = YES; HR = NO;
74                }
75                else
76                  HR = NO;
77              }
78            }
79          } else { // if ==
80            q = p;
81            while (q->next) q = q->next;
82            q->next = calloc(1, sizeof(struct tree));
```

```
83          q = q->next;
84          q->pdata = pdata;
85          q->left = NULL; q->right = NULL; q->next = NULL;
86          p->bal = 0;
87        }
88      }
89    }
90    return p;
91  }
92
93  #endif // #ifndef BTREE_H
```

fano.h

```
1   #ifndef FANO_H
2   #define FANO_H
3
4   #include <stdio.h>
5   #include <stdlib.h>
6   #include <string.h>
7
8   int med(float* P, int L, int R) {
9     int m;
10    float SL, SR;
11    SL = 0;
12    for (int i = L; i < R; ++i)
13      SL += P[i];
14    SR = P[R];
15    m = R;
16    while (SL >= SR) {
17      --m;
18      SL -= P[m];
19      SR += P[m];
20    }
21    return m;
22  }
23
24  void fano(float* P, int L, int R, int k, int** C, int* Length) {
25    int m;
26    //printf("fano: left %d -- right %d\n", L, R);
27
28    if (L < R) {
29      ++k;
30      m = med(P, L, R);
31      //printf("Median: %d\n", m);
32      for (int i = L; i <= R; ++i) {
33        if (i <= m)   C[i][k-1] = 0;
34        else    C[i][k-1] = 1;
35        Length[i] += 1;
36      }
37      fano(P, L, m, k, C, Length);
38      fano(P, m+1, R, k, C, Length);
39    }
40  }
41
42
43  #endif // #ifndef FANO_H
```

coding.h

```c
#include "struct.h"
#include "insertSort.h"
#include "fano.h"

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int runCoding() {
  FILE *fp;
  fp = fopen("testBase4.dat", "rb");
  if (fp == NULL) {
    printf("File 'textBase4.dat' not found.\n");
    return 1;
  }

  int allSymbols = 4000*(32+18+10+2);

  printf("Size: %d\n", allSymbols);
  struct record* records = malloc(sizeof(struct record)*4000);
  unsigned char* text = malloc(sizeof(unsigned char)*allSymbols);

  int i = fread(text, sizeof(unsigned char), allSymbols, fp);
  if (i == 4000) printf("Read succesfully!\n");

  struct dict* unique = malloc(sizeof(struct dict)*allSymbols);
  int uniqueCount = 0;

  for (int i = 0; i < allSymbols; ++i) {
    int found = 0;
    for (int j = 0; j < uniqueCount; ++j) {
      if (text[i] == unique[j].c)
      {
        unique[j].n += 1;
        found = 1;
        break;
      }
    }
    if (found == 0) {
      unique[uniqueCount].c = text[i];
      unique[uniqueCount].n = 1;
      uniqueCount++;
    }
  }
  printf("Unique count: %d\n", uniqueCount);

  insertSort(unique, uniqueCount);

  float* P = calloc(sizeof(float), uniqueCount);
  for (int i = 0; i < uniqueCount; ++i) {
    P[i] = unique[i].n*1.0 / allSymbols;
  }

  int** C = calloc(sizeof(int*), uniqueCount);
  for (int i = 0; i < uniqueCount; ++i) C[i] = calloc(sizeof(int), uniqueCount);
  int* Length = calloc(sizeof(int), uniqueCount);

```

```
58    fano(P, 0, uniqueCount-1, 0, C, Length);
59
60    for (int i = 0; i < uniqueCount; ++i) {
61      printf("%.5f\t\t", P[i]);
62      for (int j = 0; j < Length[i]; ++j)
63        printf("%d", C[i][j]);
64      putchar('\n');
65    }
66
67    double H = 0.0;
68    double Lmean = 0.0;
69    for (int i = 0; i < uniqueCount; ++i) {
70      H += ( P[i] * log2(P[i]) );
71      Lmean += ( P[i] * Length[i] );
72    }
73
74    printf("Entropy \tH = %.4f\n", -H);
75    printf("Lmean \t\tL = %.4f\n", Lmean);
76
77    for (int i = 0; i < uniqueCount; ++i) {
78      unique[i].code = calloc(sizeof(char), Length[i]+1);
79      for (int j = 0; j < Length[i]; ++j)
80        unique[i].code[j] = (char)('0'+C[i][j]);
81      unique[i].code[Length[i]] = '\0';
82    }
83
84    getchar();
85
86    return 0;
87  }
```

Source-файлы:

main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <iconv.h>
5
6  #include "search.h"
7  #include "dbtree.h"
8  #include "coding.h"
9
10 #define RECNUM 4000
11
12 #ifndef RECONPAGE
13 #define RECONPAGE 10
14 #endif
15
16 #include "struct.h"
17 #include "mergeSort.h"
18
19 Que g_mergedQue;
20
21 void from_866_to_utf8(char *in, size_t *sizein, char *out, size_t *sizeout) {
22   iconv_t cd = iconv_open("UTF-8", "CP866");
23   iconv(cd, &in, sizein, &out, sizeout);
24 }
25
```

```
26  void decode(struct record2* base2, struct record* base) {
27    for (int i = 0; i < RECNUM; ++i) {
28        size_t sizein = sizeof(base[i].person);
29        size_t sizeout = sizeof(base2[i].person);
30        char *in = calloc(sizeof(char), sizein);
31        char *out = calloc(sizeof(char), sizeout);
32        strcpy(in, base[i].person);
33        from_866_to_utf8(in, &sizein, out, &sizeout);
34        strcpy(base2[i].person, out);
35        free(in);
36        free(out);
37
38        sizein = sizeof(base[i].street);
39        sizeout = sizeof(base2[i].street);
40        in = calloc(sizeof(char), sizein);
41        out = calloc(sizeof(char), sizeout);
42        strcpy(in, base[i].street);
43        from_866_to_utf8(in, &sizein, out, &sizeout);
44        strcpy(base2[i].street, out);
45        free(in);
46        free(out);
47
48        base2[i].apt = base[i].apt;
49        base2[i].house = base[i].house;
50        strcpy(base2[i].date, base[i].date);
51    }
52  }
53
54  void printRecord(Record* record) {
55    printf("%-32s %-18s %-6d %-6d %-10s\n",
56            record->person,
57            record->street,
58            record->house,
59            record->apt,
60            record->date
61            );
62  }
63
64  void printSubRecord(Record* record) {
65    printf("\\_ ");
66    printRecord(record);
67  }
68
69
70  void printHelp() {
71    printf("\n\n\e[3mPress 'd' for the next page\n" \
72            "Press 'a' for the previous page\n" \
73            "Press 'l' for the last page\n" \
74            "Press 'k' for the first page\n" \
75            "Press 's' for the search\n" \
76            "Press 'c' for the coding\n" \
77            "Press 'q' to exit\n\e[0m");
78  }
79
80  void printHelpQue() {
81    printf("\n\n\e[3mPress 'd' for the next page\n" \
82            "Press 'a' for the previous page\n" \
83            "Press 'l' for the last page\n" \
84            "Press 'k' for the first page\n" \
```

```
85            "Press 't' to create tree\n" \
86            "Press 'q' to exit from search mode.\n\e[0m");
87  }
88
89
90  void showPageList(Node* list, int page, int size) {
91    int N = RECONPAGE;
92    int idx = page*N;
93    printf("Page %d.\n", page);
94
95    int count = 0;
96    while (count < idx && count < size) {
97      list = list->next;
98      ++count;
99    }
100
101   for (int i = idx; i < (idx+N) && i < size; ++i) {
102     printRecord(list->pdata);
103     list = list->next;
104   }
105
106   printHelp();
107 }
108
109 void showPageQue(Que que, int page, int size) {
110   int N = RECONPAGE;
111   int idx = page*N;
112   printf("Search mode.\n");
113   printf("Page %d.\n", page);
114
115   int count = 0;
116   Node* p = que.head;
117   while (count < idx && count < size) {
118     p = p->next;
119     ++count;
120   }
121
122   for (int i = idx; i < (idx+N) && i < size; ++i) {
123     printRecord(p->pdata);
124     p = p->next;
125   }
126
127   printHelpQue();
128 }
129
130
131 void clearScreen() {
132   #ifndef ALL
133   system("clear");
134   #endif
135 }
136
137 void showList(Node*, int);
138
139 int nextPage(int page, int size) {
140   if ((page+1) < (size/RECONPAGE + (size%RECONPAGE ? 1 : 0))) ++page;
141   return page;
142 }
143
```

```
144  int prevPage(int page, int size) {
145    if ((page-1) >= 0) --page;
146    return page;
147  }
148
149  int lastPage(int page, int size) {
150    return size ? size/RECONPAGE+(size%RECONPAGE?1:0) - 1 : 0;
151  }
152
153  pTree createTree(Que que) {
154    pTree tree = NULL;
155    Qnode* p = que.head;
156    for (int i = 0; i < sizeQue(que); ++i) {
157      tree = addDBtree(p->pdata, tree);
158      p = p->next;
159    }
160    return tree;
161  }
162
163  void printTree(pTree p) {
164    if (p != NULL) {
165      printTree(p->left);
166      printf("* ");
167      printRecord(p->pdata);
168      pTree q = p->next;
169      while (q) {
170        printSubRecord(q->pdata);
171        q = q->next;
172      }
173      printTree(p->right);
174    }
175  }
176
177  pTree findSubTree(int apt, pTree p) {
178    pTree q = p;
179    while (q != NULL) {
180      if (q->pdata->apt > apt)
181        q = q->left;
182      else {
183        if (q->pdata->apt < apt)
184          q = q->right;
185        else
186          break;
187      }
188    }
189
190    if (q != NULL) {
191      if (q->pdata->apt == apt)
192        return q;
193    }
194
195    return NULL;
196  }
197
198  void showQueSort(Que que) {
199    char c;
200    int page = 0;
201    int N = RECONPAGE;
202    int size = sizeQue(que);
```

```
203
204     //Node* list = (Node*)que.head;
205
206     clearScreen();
207     showPageQue(que, page, size);
208     while ((c = getc(stdin)) != 'q') {
209       switch(c) {
210           case 'd':
211             clearScreen();
212             page = nextPage(page, size);
213             showPageQue(que, page, size);
214             break;
215           case 'a':
216             clearScreen();
217             page = prevPage(page, size);
218             showPageQue(que, page, size);
219             break;
220           case 'l':
221             clearScreen();
222             page = lastPage(page, size);
223             showPageQue(que, page, size);
224             break;
225           case 'k':
226             clearScreen();
227             page = 0;
228             showPageQue(que, page, size);
229             break;
230           case 't':
231             clearScreen();
232             pTree tree = createTree(que);
233             printTree(tree);
234
235             int apt;
236             printf("\nEnter the apt number: ");
237             scanf("%d", &apt);
238
239             pTree subtree = findSubTree(apt, tree);
240
241             printf("Apt: %d\n", apt);
242             if (subtree != NULL) {
243               pTree p = subtree;
244               while (p != NULL) {
245                 printRecord(p->pdata);
246                 p = p->next;
247               }
248             }
249             else {
250               printf("Not found.\n");
251             }
252             getchar();
253
254           default:
255             break;
256       }
257     }
258   }
259
260   void startSearch(Node* list) {
261
```

```
262     Record** indexArray = calloc(sizeof(Record*), 4000);
263     Node* p = list;
264     for (int i = 0; i < 4000; ++i, p = p->next)
265       indexArray[i] = p->pdata;
266 
267     printf("Using the field 'year' as a key of the search.\n");
268     printf("Enter '0' to exit.\n");
269     printf("Valid year format: 94 (not 1994)\n");
270     int year;
271     while (1) {
272       printf("Please enter the year: ");
273       scanf("%d", &year);
274       if (year == 0) return; // exit
275       if (year > 0 && year < 100) break;
276       else {
277         printf("\nHint: Use 94 if you want to search by 1994.\n");
278         printf("Year's range: (1-99)\n");
279         printf("Enter '0' to exit.\n");
280         printf("Try again!\n\n");
281       }
282     };
283 
284     Que que;
285     que = search(g_mergedQue, indexArray, 4000, year);
286 
287     showQueSort(que);
288   }
289 
290   void showList(Node* list, int size) {
291     char c;
292     int page = 0;
293     int N = RECONPAGE;
294     clearScreen();
295     showPageList(list, page, size);
296     while ((c = getc(stdin)) != 'q') {
297       switch(c) {
298           case 'd':
299             clearScreen();
300             page = nextPage(page, size);
301             showPageList(list, page, size);
302             break;
303           case 'a':
304             clearScreen();
305             page = prevPage(page, size);
306             showPageList(list, page, size);
307             break;
308           case 'l':
309             clearScreen();
310             page = lastPage(page, size);
311             showPageList(list, page, size);
312             break;
313           case 'k':
314             clearScreen();
315             page = 0;
316             showPageList(list, page, size);
317             break;
318           case 's':
319             clearScreen();
320             printf("Search.\n");
```

```
321             startSearch(list);
322             clearScreen();
323             page = 0;
324             showPageList(list , page, size);
325             break;
326           case 'c':
327             clearScreen();
328             printf("Fano coding.");
329             runCoding();
330 
331             getchar();
332             clearScreen();
333             showPageList(list , page, size);
334             break;
335           default:
336             break;
337       }
338     }
339   }
340 
341   Node* createNode() {
342     Node* node = calloc(sizeof(Node), 1);
343     node->next = NULL;
344     return node;
345   }
346 
347   Node* createList(Record* base) {
348     Node* list = createNode();
349     Node* head = list;
350     list->pdata = base;
351     for (int i = 1; i < RECNUM; ++i) {
352       list->next = createNode();
353       list = list->next;
354 
355       list->pdata = (base + i);
356     }
357     return head;
358   }
359 
360   int main() {
361     FILE *fp;
362     fp = fopen("testBase4.dat", "rb");
363 
364     struct record* base = calloc(sizeof(struct record), RECNUM);
365     struct record2* base2 = calloc(sizeof(struct record2), RECNUM);
366 
367     int i = fread((struct record *) base, sizeof(struct record), RECNUM, fp);
368     if (i != RECNUM) fprintf(stderr, "Some errors with reading from .dat file\n");
369 
370     decode(base2, base);
371     fclose(fp);
372     free(base);
373 
374     Node* list = createList((Record*)base2);
375 
376     g_mergedQue = mergeSort(&list , 4000, sortDateStreet);
377 
378     showList(list , 4000);
379 
```

```
380     return 0;
381 }
```