Федеральное агентство связи

Сибирский государственный университет телекоммуникаций и информатики

Кафедра прикладной математики и кибернетики

КУРСОВАЯ РАБОТА

"Структуры и алгоритмы обработки данных"

ВАРИАНТ 23

Выполнил: студент группы ИП-413

Шапоренко Ю.А.

Проверил: ассистент кафедры

ПМиК Климова И.В.

Новосибирск, 2015 г.

СОДЕРЖАНИЕ

1.Постановка задачи

2.Основные идеи и хаpактеpистики пpименяемых методов

2.1.Метод сортировки

2.2.Двоичный поиск

2.3.Списки и очереди

3.Описание структур данных и использованных алгоритмов

3.1.Структуры данных

3.2.Особенности реализации алгоритмов

4.Описание программы

4.1.Основные переменные и структуры

4.2.Описание подпрограмм

5.Исходный текст пpогpаммы

6.Результаты

7.Выводы

1. Постановка задачи

Хранящуюся в файле базу данных (4000 записей) загрузить динамически в оперативную память компьютера в виде списка, вывести на экран по 20 записей (строк) на странице с возможностью отказа от показа, отсортировать базу данных с использованием очередей, затем, для проведения быстрого поиска, по отсортированному списку строится индексный массив. Предусмотреть возможность поиска по ключу в упорядоченной базе, в результате которого из записей с одинаковым ключом формируется очередь, содержимое очереди выводится на экран.

B = 4 (файл testBase4.dat)

База данных "Населенный пункт"

Стpуктуpа записи:

ФИО гражданина: текстовое поле 32 символа

фоpмат <Фамилия>\_<Имя>\_<Отчество>

Название улицы: текстовое поле 18 символов

Номер дома: целое число

Номер квартиры: целое число

Дата поселения: текстовое поле 10 символов

фоpмат дд-мм-гг

Пpимеp записи из БД:

Петpов\_Иван\_Федоpович\_\_\_\_\_\_\_\_\_\_\_

Ленина\_\_\_\_\_\_\_\_\_\_\_\_\_\_

10

67

29-02-65

Условие упорядочивания и ключи поиска (К): по дате поселения и названию улицы, К = год поселения.

Метод сортировки: метод прямого слияния.

2. Основные идеи и характеристики применяемых методов

2.1. Метод сортировки

*Метод прямого слияния*

В основе метода прямого слияния лежит операция слияния серий. р-серией называется упорядоченная последовательность из р элементов.

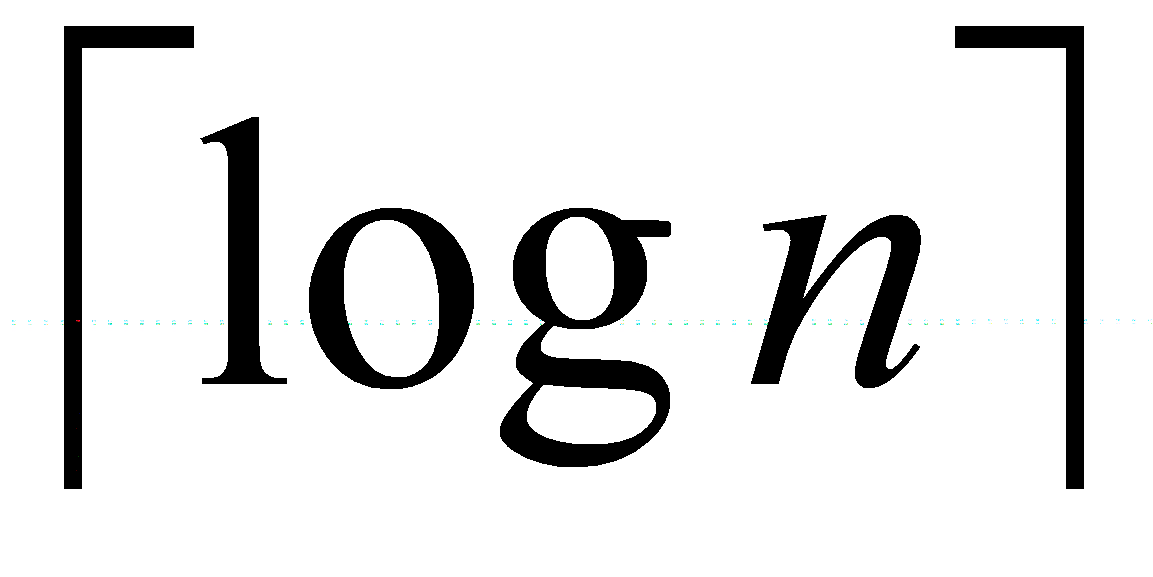
Пусть имеются две упорядоченные серии a и b длины q и r соответственно. Необходимо получить упорядоченную последовательность с, которая состоит из элементов серий a и b. Сначала сравниваем первые элементы последовательностей a и b. Минимальный элемент перемещаем в последовательность с. Повторяем действия до тех пор, пока одна из последовательностей a и b не станет пустой, оставшиеся элементы из другой последовательности переносим в последовательность с. В результате получим (q+r)-серию.

Для алгоритма слияния серий с длинами q и r необходимое количество сравнений и перемещений оценивается следующим образом

min(q, r) ≤ C ≤ q+r-1, M=q+r

Пусть длина списка S равна степени двойки, т.е. 2k, для некоторого натурального k. Разобьем последовательность S на два списка a и b, записывая поочередно элементы S в списки а и b. Сливаем списки a и b с образованием двойных серий, то есть одиночные элементы сливаются в упорядоченные пары, которые записываются попеременно в очереди c0 и c1. Переписываем очередь c0 в список a, очередь c1 – в список b. Вновь сливаем a и b с образованием серий длины 4 и т. д. На каждом итерации размер серий увеличивается вдвое. Сортировка заканчивается, когда длина серии превысит общее количество элементов в обоих списках. Если длина списка S не является степенью двойки, то некоторые серии в процессе сортировки могут быть короче.

Трудоёмкость метода прямого слияния определяется сложностью операции слияния серий. На каждой итерации происходит ровно n перемещений элементов списка и не более n сравнений. Как нетрудно видеть, количество итераций равно .Тогда



Дополнительные n перемещений происходят во время начального расщепления исходного списка. Асимптотические оценки для М и С имеют следующий вид



Метод обеспечивает устойчивую сортировку. При реализации для массивов, метод требует наличия второго вспомогательного массива, равного по размеру исходному массиву. При реализации со списками дополнительной памяти не требуется.

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 Списки и очереди

*Очередь* — это линейный список информации, работа с которой происходит по принципу "первым пришел — первым вышел" (first-in, first-out); этот принцип (и очередь как структура данных) иногда еще называется FIFO[[1]](http://lord-n.narod.ru/download/books/walla/programming/Spr_po_C/22/2201.htm" \l "11). Это значит, что первый помещенный в очередь элемент будет получен из нее первым, второй помещенный элемент будет извлечен вторым и т.д. Это единственный способ работы с очередью; произвольный доступ к отдельным элементам не разрешается.

В программировании очереди применяются при решении многих задач. Один из наиболее популярных видов таких задач — симуляция. Очереди также применяются в планировщиках задач операционных систем и при буферизации ввода/вывода.

В отличие от очереди, *связанный список* допускает гибкие способы доступа, поскольку каждый фрагмент информации имеет ссылку на следующий элемент данных в цепочке. Кроме того, операция извлечения не приводит к удалению из списка и уничтожению элемента данных. В принципе, для этой цели необходимо ввести дополнительную специальную операцию удаления.

Связанные списки могут быть односвязными и двусвязными. В односвязном списке каждый элемент информации содержит ссылку на следующий элемент списка. Каждый элемент данных обычно представляет собой структуру, которая состоит из информационных полей и указателя связи. Существует два основных способа построения односвязного списка. Первый способ — помещать новые элементы в конец списка. Второй — вставлять элементы в определенные позиции списка, например, в порядке возрастания. От способа построения списка зависит алгоритм функции добавления элемента.

Двусвязный список содержит ссылки как на последующий, так и на предыдущий элементы списка. Наличие двух ссылок вместо одной предоставляет несколько преимуществ. Вероятно, наиболее важное из них состоит в том, что перемещение по списку возможно в обоих направлениях. Это упрощает работу со списком, в частности, вставку и удаление. Помимо этого, пользователь может просматривать список в любом направлении. Еще одно преимущество имеет значение только при некоторых сбоях. Поскольку весь список можно пройти не только по прямым, но и по обратным ссылкам, то в случае, если какая-то из ссылок станет неверной, целостность списка можно восстановить по другой ссылке.

Выбор типа применяемого списка зависит от конкретной задачи.

3.Описание структур данных и использованных 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 menu()****,*** которая обеспечивает корректное и незатруднительное использование программы и предоставляет возможность многократного выбора различных вариантов обработки базы данных, в зависимости от задач пользователя. Визуальное представление пунктов меню вынесено в отдельную процедуру *void info().*

2. Загрузка и вывод базы данных

Для загрузки базы данных разработана процедура *struct qel \* LoadMem*, в которой производится считывание записей типа *struct Record,* а из них формируется очередь *struct qel.* 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.

За вывод элементов считанной базы данных отвечает процедура *void ViewBase* *Она предоставляет возможность постраничного вывода базы данных(по 20 элементов на странице), вывод следующей страницы осуществляется нажатием управляющей стрелки «вниз» на клавиатуре. Есть возможность прервать просмотр в любой момент времени нажатием клавиши «Esc».*

3. Вспомогательные функции и процедуры для сортировки данных

При сортировке базы данных потребовалось реализовать дополнительную процедуру *void InvertDate* для преобразования даты рождения в формат, соответствующий условию упорядочивания («дд-мм-гг» в «гг-мм-дд»), а также определить операцию сравнения двух строк(не используя стандартные процедуры среды) для последующего применения в алгоритме сортировки. Данную функция выполняет процедура *char compare, которая поэлементно сравнивает две строки и при первом различии возвращает результат, а при совпадении проверяет на равенство другое поле записей.*

4. Особенности реализации бинарного поиска

Для того чтобы без проблем многократно осуществлять поиск элементов, соответствующих разным ключам, требуется каждый раз создавать новую очередь, и чтобы постоянно не выделять память, процедура void FreeQueue очищает , ту что была распределена при предыдущем вызове функции построения очереди - *void MakeQueue*. Новая очередь же строится непосредственно после выполнения поиска. При его реализации была использована вторая версия двоичного поиска, 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.

4. ОПИСАНИЕ ПРОГРАММЫ

4.1. Основные переменные и структуры

**struct Record**

**{**

**char fio[32];**

**char street[18];**

**short int house;**

**short int flat;**

**char hd[10];**

**};**

Запись, используемая для работы с базой данных

**struct qel**

**{**

**struct qel \*next;**

**struct Record \*data;**

**};**

Структура(список), используемая при сортировке базы данных.

**struct qel \*next –** указатель на следующие элемент;

**struct Record \*data –** поле данных(адрес элемента в основном массиве структур.

**struct queueS**

**{**

**qel \*head;**

**qel \*tail;**

**};**

Структура (очередь), используемая при сортировке базы данных.

**qel \*head –** голова очереди;

**qel \*tail –** хвост очереди.

**struct queue**

**{**

**int index;**

**struct queue \*next;**

**}**

Структура, используемая при построении очереди из элементов, полученных в результате бинарного поиска.

**int index**– индекс элемента в базе данных;

**struct queue \*next** – указатель на следующий элемент.

**int index[4000];**

Индексный массив на 4000 элементов (по числу записей).

**Struct Record \*el[3999];**

Указатель на массив структур.

4.2. Описание подпрограмм

Процедуры, вывода меню:

**1. void menu();**

**2. void info();**

Реализуют меню, ***info*** – отвечает только за визуальное представление, ***menu*** – непосредственно за функциональное.

Процедуры начальной обработки базы данных:

**3. struct qel \* LoadMem();**

**4. void ViewBase(struct Struct \*\*el,int iN);**

***LoadMem*** – считывание базы из файла и представление ее элементов в форме вышеперечисленных структур, возвращает указатель на массив записей.

***ViewBase*** - просмотр базы. ***struct Struct \*\*el*** – указатель на первый элемент, ***iN*** – общее количество записей в базе.

Функции и процедуры сортировки:

**5. void InvertDate(char aDate[],int n);**

**6. char compare(char aDate[],char bDate[],char aStreet[],char bStreet[],int n, int m);**

**7. int Devide(qel\* &s, qel\* &a, qel\* &b);**

**8. void Merging(qel\* &a, qel\* &b, queueS &c, int q, int r);**

**9. qel\* MergeSort(qel\* s);**

***InvertDate*** – вспомогательная процедура для преобразования даты (представленной в виде символьного массива **char aData[],** размерностью ***n***) в формат соответствующий условию упорядочивания(«дд-мм-гг» в «гг-мм-дд»).

***Compare*** – определение операции сравнения двух массивов: **char aDate[],char bDate[]** размерностью ***n***. Возвращает ***1*** в случае, если ***a > b*** и ***2***, если ***a < b***. Если оба массива равны, то сравниваются два массива **char aStreet []** и **char bStreet []** размерностью **m**. Возвращает ***1*** в случае, если ***a > b*** и ***2***, если ***a < b***.

***Devide, Merging, MergeSort*** – реализация сортировки.

***Devide*** – разделение последовательности ***s*** на два списка ***a и b,*** возвращает количество элементов в списке ***s.***

***Merging*** – слияние ***q – серии*** из списка ***a*** c ***r – серией*** списка ***b***, запись результата в очередь ***c.***

***MergeSort*** – инициализация очередей, сама процедура сортировки элементов последовательности ***s,*** возвращает голову (head) первой из двух инициализированных очередей.

Функции и процедуры для поиска в отсортированной базе данных:

**10. int BinSearch(struct Record \*\*x,int N,int \*pointers,char \*value);**

**11. void FreeQueue(queue \*p);**

**12. void MakeQueue(char \*n,struct queue \*pq,int \*index,int pos);**

**13. void PrintQueue(struct queue \*p);**

***BinSearch*** – процедура двоичного поиска (версия 2), ***struct Record \*\*x*** – указатель на массив записей, в котором осуществляется поиск, ***N*** – количество записей(изначально - правая граница поиска), ***pointers*** – указатель на индексный массив, через который происходит обращение к элементам, ***value*** – ключ поиска. Возвращает позицию найденного элемента и ***-1***, в случае его отсутствия.

***FreeQueue*** – освобождение памяти для очереди ***p***, если, например, она уже создавалась.

***MakeQueue*** – построение очереди из результатов поиска. ***n*** – самый левый из найденных элементов, ***pq*** – голова очереди, ***index*** – указатель на индексный массив, ***pos*** – позиция, в которой был найден нужный элемент, от нее просматриваем массив только вправо.

***PrintQueue*** – вывод очереди ***p* (struct queue \*p –** указатель на первый элемент очереди**)** на экран.

Основная программа

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

5. ИСХОДНЫЙ ТЕКСТ ПРОГРАММЫ

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#include <iostream>

#include <conio.h>

#include <windows.h>

#include <fstream>

#include <IO.H>

#include <FCNTL.H>

#include <math.h>

using namespace std;

int index[4000];

**struct qel \* LoadMem**();

**int Devide**(qel\* &s, qel\* &a, qel\* &b);

**qel\* MergeSort**(qel\* s);

**struct queue** {

int index;

struct queue \*next;

} \*headq=NULL,\*tailq,\*spis;

struct Record

{

char fio[32];

char street[18];

short int house;

short int flat;

char hd[10];

};

struct qel{

struct qel \*next;

struct Record \*date;

};

struct Record \*el[3999];

struct queueS {

qel \*head;

qel \*tail;

};

struct qel \* LoadMem(){

int i=0, j, f;

struct qel \*head,\*p,\*tail;

printf("\n Base is loading...\n");

f = open("testBase4.dat", O\_RDONLY | O\_BINARY);

head=NULL;

tail=head;

while (!eof(f)){

p = (struct qel \*)malloc(sizeof(struct qel));

if (p==NULL){

printf("\n ERROR: Out of Memory");

getch();

exit(0);

}

p->date = (struct Record \*)malloc(sizeof(struct Record));

if (p->date==NULL){

printf("\n ERROR: Out of Memory");

getch();

exit(0);

}

read(f, p->date,sizeof(struct Record));

p->next=NULL;

if (head!=NULL) tail->next=p;

else head=p;

tail=p;

i++;

}

close(f);

printf("\n Base was loaded\n");

return(head);

}

**void info**(){

printf("\n");

printf(" [ 1 ] Print base\n");

printf(" [ 2 ] Sort base\n");

printf(" [ 3 ] Search in base\n");

printf(" [ESC] Exit\n");

}

**void ViewBase**(struct Record \*\*el,int iN){

int i,j,end;

char key;

j=0;

do {

if (iN>=20\*(j+1)) end=20\*(j+1);

else end=iN;

for (i=20\*j;i<end;i++){

printf("\n%4d|",index[i]+1);

printf("%32s",el[index[i]]->fio);

printf("|%18s ",el[index[i]]->street);

printf("|%d ",el[index[i]]->house);

printf("|%3d ",el[index[i]]->flat);

printf("|%10s|",el[index[i]]->hd);

printf("\n");

}

printf("\n ESC - back to menu, arrow down - following entries,");

key=getch();

if ((key==80)&&((j+1)\*3<iN)){ j++; }

else {

break;

}

} while(key!=27);

}

**int Devide**(qel\* &s, qel\* &a, qel\* &b){

qel \*k,\*p;

int n;

a=s;

b=s->next;

n=1;

k=a;

p=b;

while (p){

n++;

k->next=p->next;

k=p;

p=p->next;

}

return n;

}

**void InvertDate**(char aDate[],int n){

char a[n];

int i=0, j=n-2;

while (i<n){

a[i]=aDate[j];

a[++i]=aDate[j+1];

a[++i]='-';

i++;

j=j-3;

}

for (i=0;i<n;i++) aDate[i]=a[i];

}

**char compare**(char aDate[],char bDate[],char aStreet[],char bStreet[],int n, int m){

char a[n], b[n],d[m], e[m];; int i;

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

a[i]=aDate[i];

b[i]=bDate[i];

}

**InvertDate**(a,n);

**InvertDate**(b,n);

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

if(a[i]>b[i])return 2;

else if(a[i]<b[i])return 1;

}

for (i=0;i<m;i++){

d[i]=aStreet[i];

e[i]=bStreet[i];

}

if (strcmp(aStreet,bStreet) < 0) return 1;

else if (strcmp(aStreet,bStreet) > 0) return 2;

return 1;

}

**void Merging**(qel\* &a, qel\* &b, queueS &c, int q, int r){

qel \*p;

while ((q!=0)&&(r!=0)){

if (**compare**(a->date->hd,b->date->hd,a->date->street,b->date->street,8,18) == 1){

p=a;

a=a->next;

p->next=NULL;

if (c.head) c.tail->next=p;

else c.head=p;

c.tail=p;

q--;

}

else{

p=b;

b=b->next;

p->next=NULL;

if (c.head) c.tail->next=p;

else c.head=p;

c.tail=p;

r--;

}

}

while (q>0){

p=a;

a=a->next;

p->next=NULL;

if (c.head) c.tail->next=p;

else c.head=p;

c.tail=p;

q--;

}

while (r>0){

p=b;

b=b->next;

p->next=NULL;

if (c.head) c.tail->next=p;

else c.head=p;

c.tail=p;

r--;

}

}

**qel\* MergeSort**(qel\* s){

int n,p,q,r,m,i;

struct queueS Cs[2];

struct qel \*a,\*b;

n=**Devide**(s,a,b);

p=1;

while (p<n){

Cs[0].head=NULL;

Cs[1].head=NULL;

Cs[0].tail=Cs[0].head;

Cs[1].tail=Cs[1].head;

i=0;

m=n;

while (m>0){

if (m>=p) q=p;

else q=m;

m-=q;

if (m>=p) r=p;

else r=m;

m-=r;

**Merging**(a,b,Cs[i],q,r);

i=1-i;

}

a=Cs[0].head;

b=Cs[1].head;

p\*=2;

}

Cs[0].tail->next=NULL;

return Cs[0].head;

}

**int BinSearch**(struct Record \*\*x,int N,int \*pointers,char \*key){

int m,L,R;

int i; char y[2];

L=0;

R=N;

while(L<R){

m=(L+R)/2;

for(i=0;i<2;i++) y[i]=x[pointers[m]]->hd[i+6];

if (strncmp(y,key,2)<0) L=m+1;

else R=m;

}

for(i=0;i<2;i++) y[i]=x[pointers[R]]->hd[i+6];

if(strncmp(y,key,2)==0)return R;

else return -1;

}

**void FreeQueue**(queue \*p){

queue \*q;

while (p!=NULL){

q=p;

p=p->next;

free(q);

}

p=NULL;

}

**void MakeQueue**(char \*n,struct queue \*pq,int \*index,int pos){

char y[2];

headq=NULL;

while (1){

for (int i=0;i<2;i++) y[i]=el[index[pos]]->hd[i+6];

if (strncmp(y,n,2)!=0) break;

pq = (queue \*)malloc(sizeof(queue));

pq -> next = NULL;

pq -> index = index[pos];

if (headq != NULL) tailq -> next = pq;

else

headq = pq;

tailq = pq;

pos++;

if(pos==4000)break;

}

}

**void PrintQueue**(struct queue \*p){

while (p!=NULL){

printf("|%32s",el[p->index]->fio);

printf("|%18s ",el[p->index]->street);

printf("|%d ",el[p->index]->house);

printf("|%3d ",el[p->index]->flat);

printf("|%10s|",el[p->index]->hd);

printf("\n");

p=p->next;

}

}

**int main**()

{

int i;

char ch;

struct qel \*head,\*tail,\*p;

head=LoadMem();

printf("\n Press any key to back to menu...");

getch();

getch();

for (i=0;i<=3999;i++) index[i]=i;

i=0;

for (p=head;p!=NULL;p=p->next) el[index[i++]]=p->date;

while (1){

system("cls");

info();

ch=getch();

switch (ch){

case '1':

system("cls");

printf("\n BASE \n");

**ViewBase**(el,i);

printf("\n THE END OF BASE \n");

printf("\n Press any key to back to menu...");

getch();

break;

case '2':

system("cls");

i=0;

qel \*q;

for (p=**MergeSort**(head);p!=NULL;p=p->next)

{

el[index[i++]]=p->date;

}

printf("\n Base was sorted");

printf("\n Press any key to back to menu...");

getch();

break;

case '3':

system("cls");

printf("\n Enter search key (format 'yy'): ");

char x[2];

int y;

scanf("%s",x);

y=**BinSearch**(el,3999,index,x);

system("cls");

if (y==-1){ printf(" Element not found");getch();break; }

**FreeQueue**(spis);

**MakeQueue**(x,spis,index,y);

spis=headq;

printf("\n QUEUE \n");

**PrintQueue**(spis);

printf("\n THE END OF QUEUE \n");

printf("\n Press any key to back to menu...");

getch();

break;

case 27:

exit(0);

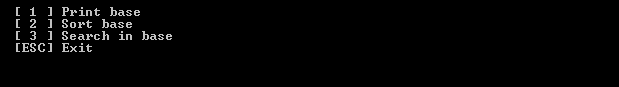
}

}

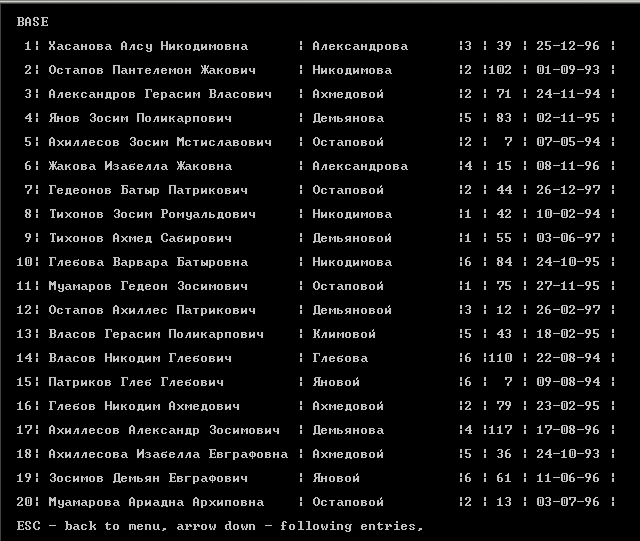
}

6. РЕЗУЛЬТАТЫ

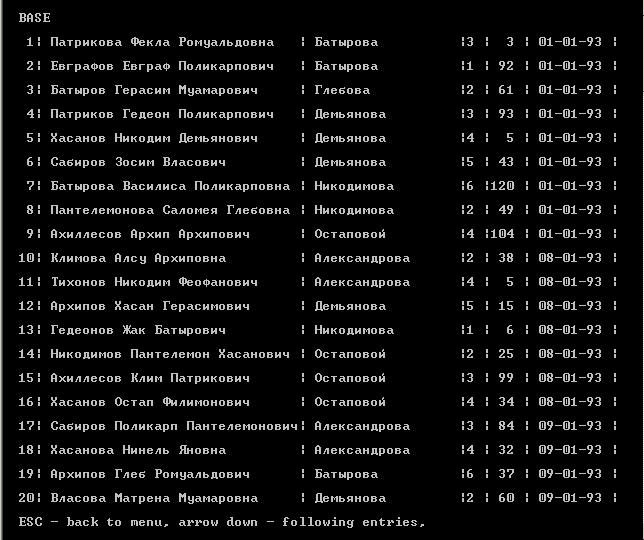
1. Меню программы



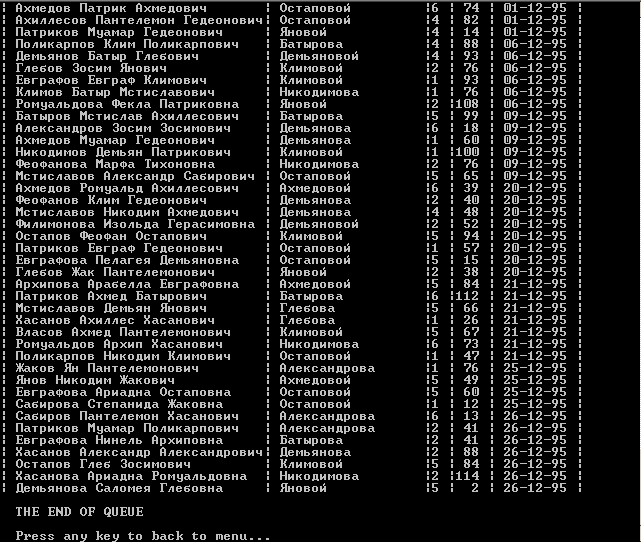
2. Неотсортированная база данных



3. Отсортированная по дате заселения и названию улицы база данных



4. Результат поиска по ключу (ключ – год поселения)



7. ВЫВОДЫ

В ходе выполнения курсовой работы были выполнены все поставленные задачи и реализованы необходимые алгоритмы: работа со списками и очередями, сортировки, поиска.

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

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