МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания 5.2

Тема: «РАБОТА С ДАННЫМИ ИЗ ФАЙЛА»

Дисциплина: Структуры и алгоритмы обработки данных

Выполнил студент Комисарик М.А.
группа ИКБО-20-23

Москва 2024

Цель работы: Получить практический опыт по применению алгоритмов поиска в таблицах данных.

Задание 1

Формулировка задачи

Создать двоичный файл из записей информации о студенте. Студент: номер зачетной книжки, номер группы, ФИО. Поле ключа записи – номер зачетной книжки. Заполнить файл данными, используя для поля ключа датчик случайных чисел. Ключи записей в файле уникальны.

Модель решения

Для создания каждой записи требуется занять для каждого поля память. Были выбраны следующие размеры полей:

Номер зачетной книжки – 4 байта (для хранения чисел типа unsigned int).

Номер группы – 10 байтов (XXXX-XX-XX 10 символов).

ФИО – 50 байтов.

Итого 64 байта.

Для доступа к конкретным полям записи используется указательная арифметика и преобразование reinterpret_cast<>(). Для записи используются файловый поток ofstream в двоичном режиме с помощью метода write(). При заполнении полей ФИО использовался файл с 10000 случайных ФИО.

Код программы

Код программы задания 1:

```
1:   ofstream file("students.sts", ios::trunc | ios::out | ios::binary);
2:
3:   unsigned int studentsAmount;
4:   cout << "Введите количество студентов: ";
5:   do
6:   {
7:       cin >> studentsAmount;
8:       if (studentsAmount > 10000)
9:       {
10:          cout << "Введите число не большее 10000.\n";
11:      }
12:  } while (studentsAmount > 10000);
```

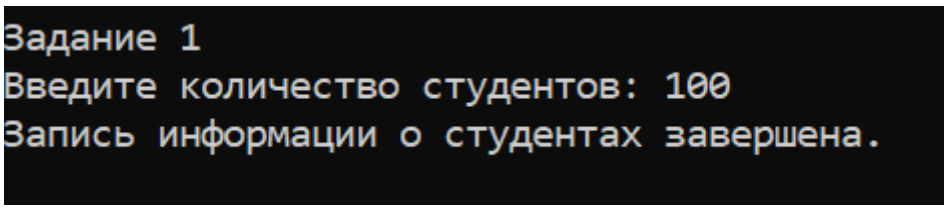
```
13:
14:  GenerateStudentsFile(file, studentsAmount, 4, 10, 50);
15:
16:  cout << "Запись информации о студентах завершена.\n";}
```

Код функции GenerateStudentsFile(…):

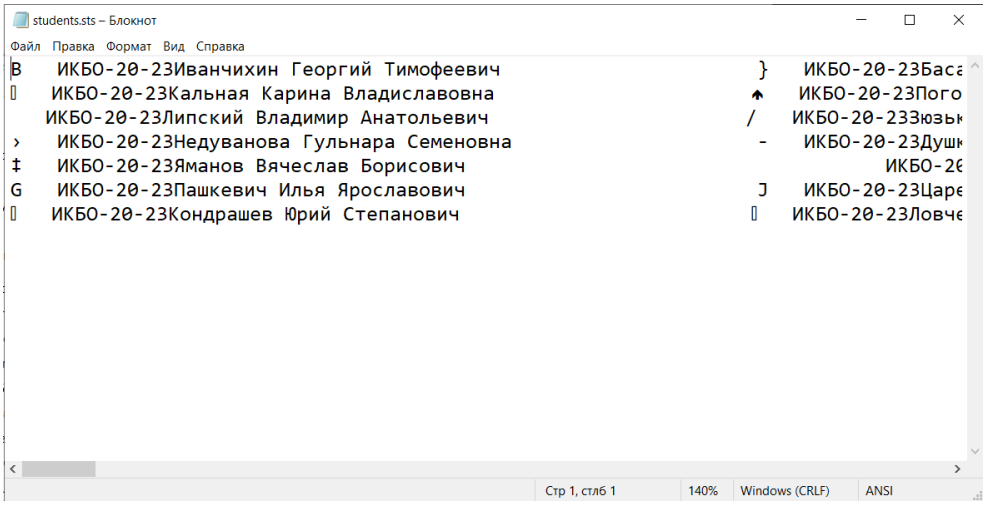
```
1:   void GenerateStudentsFile(ofstream& file, unsigned int studentsAmount,
     unsigned int keySize, unsigned int groupSize, unsigned int nameSize)
2:   {
3:       ifstream names("names.txt");
4:       string nameBuffer;
5:
6:       const unsigned int idSize = keySize + groupSize + nameSize;
7:       char* idBuffer = new char[idSize];
8:
9:       vector<unsigned int> keys;
10:      srand(time(0));
11:      GenerateRandomKeys(studentsAmount * 2, studentsAmount, keys);
12:
13:      for (unsigned int key : keys)
14:      {
15:          for (unsigned int i = 0; i < idSize; i++)
16:          {
17:              idBuffer[i] = 0;
18:          }
19:          getline(names, nameBuffer);
20:          *reinterpret_cast<unsigned int*>(idBuffer) = key;
21:          strncpy(idBuffer + keySize, "ИКБО-20-23", groupSize);
22:          strncpy(idBuffer + keySize + groupSize, nameBuffer.c_str(),
     nameBuffer.size());
23:          file.write(idBuffer, idSize);
24:      }
25:
26:      delete[] idBuffer;
27:      names.close();
28:  }
```

Результаты тестирования

Результаты тестирования программы задания 1:



Файл students.sts после заполнения 100 записями о студентах:



Задание 2

Формулировка задачи

Поиск в файле с применением линейного поиска

1. Разработать программу поиска записи по ключу в бинарном файле с применением алгоритма линейного поиска.

2. Провести практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей.

3. Составить таблицу с указанием результатов замера времени

Алгоритм

Линейный поиск (linear search) представляет тривиальный алгоритм, который используется, когда отсутствует какая-либо информация о данных, среди которых необходимо провести поиск. В данном случае процесс поиска требует простого последовательного просмотра элемент за элементом всего массива.

Он начинается с просмотра первого элемента и завершается, либо, когда искомый элемент найден, либо, когда будет просмотрен весь массив, и не было обнаружено совпадения.

Для этой цели реализовать алгоритм удобнее всего оператором цикла While (или Repeat-Until) с двойным условием. Первое условие контролирует принадлежность массиву (i < n). Второе условие - условие продолжения

поиска (a[i] = x). В теле цикла обычно записывается только один оператор, изменяющий индекс (параметр) массива a.

После выхода из цикла необходимо проверить, по какому из условий цикл завершился.

Следует обратить внимание на то, что если элемент найден, то он найден вместе с минимально возможным индексом, то есть, это первый из таких элементов в массиве. Равенство i = n свидетельствует, что элемента с искомым значением ключа в массиве не обнаружено.

Окончание цикла гарантировано, поскольку на каждом шаге значение i увеличивается, и, следовательно, за конечное число шагов оно обязательно достигнет n; фактически же, если совпадения не было, это произойдет после n шагов.

Псевдокод алгоритма:

```
1:   i ← 1 // инициализация параметра цикла
2:   While (i<n) And (a[i]<>x)
3:   Do i ← i + 1
4:   If (i = n) Then <Элемент не найден>
5:   Else <Элемент найден>
```

Код программы

Код линейного поиска:

```
1:   char* LinearFileSearch(const string& fileName, const unsigned int key, const
     unsigned int entrySize, const unsigned int keyOffset)
2:   {
3:       ifstream file(fileName, ios::binary | ios::in);
4:       if (!file)
5:       {
6:           file.close();
7:           throw invalid_argument("No file found with name " + fileName);
8:       }
9:
10:      char* buffer = new char[entrySize];
11:      while (file.read(buffer, entrySize))
12:      {
13:          unsigned int readKey = *reinterpret_cast<unsigned int*>(buffer +
     keyOffset);
14:          if (readKey == key)
15:          {
16:              file.close();
17:              return buffer;
18:          }
19:      }
20:      file.close();
21:      return nullptr;
22:  }
```

Код задания 2:

```
1:   unsigned int key;
2:   cout << "Введите ключ: ";
3:   cin >> key;
4:   char* foundEntry;
5:   chrono::time_point<chrono::steady_clock> start;
6:   chrono::time_point<chrono::steady_clock> end;
7:   try
8:   {
9:       start = chrono::high_resolution_clock::now();
10:      foundEntry = LinearFileSearch("students.sts", key, 64, 0);
11:      end = chrono::high_resolution_clock::now();
12:  }
13:  catch(invalid_argument& e)
14:  {
15:      cerr << "Файл не был найден.\n";
16:      return;
17:  }
18:  if (foundEntry == nullptr)
19:  {
20:      cout << "Ключ не найден.\n";
21:  }
22:  else
23:  {
24:      PrintStudent(foundEntry, 4, 10, 50);
25:  }
26:  auto duration = end - start;
27:  DisplayTimeDuration(duration);
```

Результаты тестирования

Результат тестирования программы на файле из 100 записей:

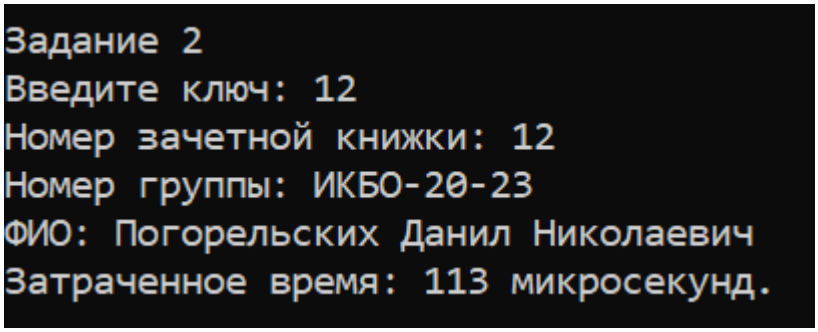


Таблица результатов:

Количество записей	Затраченное время
100	113 микросекунд
1000	173.4 микросекунд
10000	308.8 микросекунд

Задание 3

Формулировка задачи

Поиск записи в файле с применением дополнительной структуры данных, сформированной в оперативной памяти.

1. Для оптимизации поиска в файле создать в оперативной памяти структуру данных – таблицу, содержащую ключ и ссылку (смещение) на запись в файле.

2. Разработать функцию, которая принимает на вход ключ и ищет в таблице элемент, содержащий ключ поиска, а возвращает ссылку на запись в файле. Алгоритм поиска – однородный бинарный поиск с таблицей смещений.

3. Разработать функцию, которая принимает ссылку на запись в файле, считывает ее, применяя механизм прямого доступа к записям файла. Возвращает прочитанную запись как результат.

4. Провести практическую оценку времени выполнения поиска на файле объемом 100, 1000, 10 000 записей.

5. Составить таблицу с указанием результатов замера времени.

Модель решения

Каждый элемент таблицы смещений на запись в файле – целое число размером 8 байт. Первые 4 байта выделяются для ключа, а вторые – для ссылки на запись в файл. В данном случае ссылкой является номер записи в файле, который можно определить, если вести счетчик записей и увеличивать его при считывании каждой порции по 64 байта из файла (каждой записи).

Для доступа к записям файла с помощью такой ссылки используется файловый поток ifstream с бинарным способом считывания и функция seekg(…) из библиотеки fstream. Функция принимает количество байтов, которое следует отступить от начала файла (или от других точек файла, но в данном случае не требуется) и смещает указатель считывания на это количество байтов.

Алгоритм однородного бинарного поиска.

Идея алгоритма заключается в следующем. Сначала производится проверка среднего элемента массива. Если его ключ больше ключа искомого элемента, то делается проверка среднего элемента из первой половины. В

противном случае делается проверка среднего элемента из второй половины. Этот процесс повторяется до тех пор, пока не будет найден требуемый элемент или не будет проверен весь массив.

Однородный (равномерный) бинарный поиск является модификацией обычного бинарного поиска. Эта модификация заключается в следующем. Вместо трех указателей – l (нижняя граница интервала), h (верхняя граница интервала) и m (середина интервала) – используется только два: текущая позиция m и величина его изменения δ. После каждого сравнения, не давшего равенства, можно установить i ← i ± δ и δ ← δ / 2.

i ←1 c←1 // инициализация переменных алгоритма

While table[c] != 0

Do

If x = a[i] Then < элемент найден >

If x < a[i] Then i←i – table[c]

Else i←i + table[c]

c←c+1

od

Else right ←d

< элемент не найден >

Код программы

Код функции однородного бинарного поиска со смещением:

```
1:   unsigned int BinarySearchInKeyTable(const vector<unsigned long long>&
     keyTable, const unsigned int key, const vector<unsigned int>& lookupTable)
2:   {
3:       unsigned int index = lookupTable[0] - 1;
4:       unsigned int count = 0;
5:       while (lookupTable[count] != 0)
6:       {
7:           if (key == *reinterpret_cast<const unsigned int*>(&keyTable[index]))
8:           {
9:               return *(reinterpret_cast<const unsigned int*>(&keyTable[index]) +
     1);
10:          }
11:
12:          if (key < *reinterpret_cast<const unsigned int*>(&keyTable[index]))
13:          {
14:              index -= lookupTable[++count];
```

```
15:           }
16:           else
17:           {
18:               index += lookupTable[++count];
19:           }
20:       }
21:       return keyTable.size();
22:   }
```

Результаты тестирования

Результаты тестирования бинарного поиска для 10000 чисел:

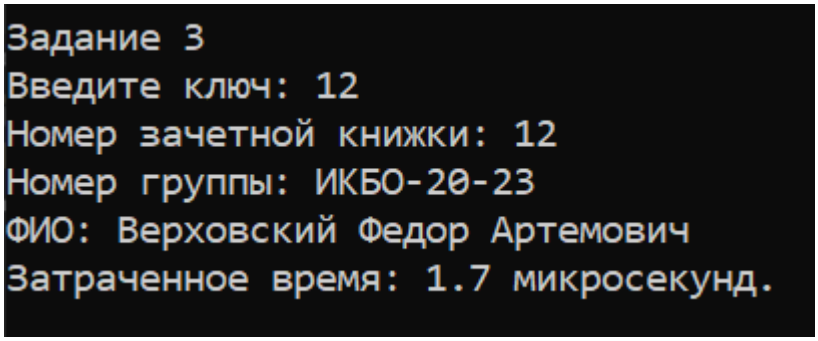


Таблица результатов:

Количество записей	Затраченное время
100	1 микросекунда
1000	1.1 микросекунд
10000	1.7 микросекунд

ВЫВОДЫ

Бинарный поиск намного быстрее линейного поиска, но недостатком бинарного поиска является необходимость отсортированности исходных данных по ключу.

Приложения

Приложение 1 – Все использованные функции.

```
1:  void GenerateStudentsFile(ofstream& file, unsigned int studentsAmount,
    unsigned int keySize, unsigned int groupSize, unsigned int nameSize)
2:  {
3:      ifstream names("names.txt");
4:      string nameBuffer;
5:
6:      const unsigned int idSize = keySize + groupSize + nameSize;
7:      char* idBuffer = new char[idSize];
8:
9:      vector<unsigned int> keys;
10:     srand(time(0));
11:     GenerateRandomKeys(studentsAmount * 2, studentsAmount, keys);
12:
13:     for (unsigned int key : keys)
14:     {
15:         for (unsigned int i = 0; i < idSize; i++)
16:         {
17:             idBuffer[i] = 0;
18:         }
19:         getline(names, nameBuffer);
20:         *reinterpret_cast<unsigned int*>(idBuffer) = key;
21:         strncpy(idBuffer + keySize, "ИКБО-20-23", groupSize);
22:         strncpy(idBuffer + keySize + groupSize, nameBuffer.c_str(),
    nameBuffer.size());
23:         file.write(idBuffer, idSize);
24:     }
25:
26:     delete[] idBuffer;
27:     names.close();
28: }
29:
30: char* LinearFileSearch(const string& fileName, const unsigned int key, const
    unsigned int entrySize, const unsigned int keyOffset)
31: {
32:     ifstream file(fileName, ios::binary | ios::in);
33:     if (!file)
34:     {
35:         file.close();
36:         throw invalid_argument("No file found with name " + fileName);
37:     }
38:
39:     char* buffer = new char[entrySize];
40:     while (file.read(buffer, entrySize))
41:     {
42:         unsigned int readKey = *reinterpret_cast<unsigned int*>(buffer +
    keyOffset);
43:         if (readKey == key)
44:         {
45:             file.close();
46:             return buffer;
47:         }
48:     }
49:     file.close();
50:     return nullptr;
51: }
52:
53: void PrintStudent(char* student, const unsigned int keySize, const unsigned
    int groupSize, const unsigned int nameSize)
54: {
55:     cout << "Номер зачетной книжки: " << *reinterpret_cast<unsigned
    int*>(student) << '\n';
```

```
56:       cout << "Номер группы: ";
57:       cout.write(student + keySize, groupSize) << '\n';
58:       cout << "ФИО: ";
59:       cout.write(student + keySize + groupSize, nameSize) << '\n';
60:   }
61:
62:   void GenerateKeyTable(vector<unsigned long long>& table, ifstream& file, const
      unsigned int entrySize, const unsigned int keyOffset)
63:   {
64:       table.clear();
65:       char* entry = new char[entrySize];
66:       unsigned int i = 0;
67:       while (file.read(entry, entrySize))
68:       {
69:           unsigned int key = *reinterpret_cast<unsigned int*>(entry +
      keyOffset);
70:           table.push_back(0);
71:           *reinterpret_cast<unsigned int*>(&table[i]) = key;
72:           *(reinterpret_cast<unsigned int*>(&table[i]) + 1) = i;
73:           i++;
74:       }
75:       SortKeyTable(table);
76:   }
77:
78:   void SortKeyTable(vector<unsigned long long>& table)
79:   {
80:       sort(table.begin(), table.end(), [](unsigned long long a, unsigned long
      long b)
81:       {
82:           return *reinterpret_cast<unsigned int*>(&a) <
      *reinterpret_cast<unsigned int*>(&b);
83:       });
84:   }
85:
86:   void GenerateLookupTable(unsigned int len, std::vector<unsigned int>& table)
87:   {
88:       table = vector<unsigned int>(static_cast<unsigned int>(log2(len)) + 3, 0);
89:       unsigned int power = 1;
90:       unsigned int count = 0;
91:
92:       do
93:       {
94:           power <<= 1;
95:           table[count] = (len + (power >> 1)) / power;
96:       }
97:       while (table[count++] != 0);
98:   }
99:
100:  unsigned int BinarySearchInKeyTable(const vector<unsigned long long>&
      keyTable, const unsigned int key, const vector<unsigned int>& lookupTable)
101:  {
102:      unsigned int index = lookupTable[0] - 1;
103:      unsigned int count = 0;
104:      while (lookupTable[count] != 0)
105:      {
106:          if (key == *reinterpret_cast<const unsigned int*>(&keyTable[index]))
107:          {
108:              return *(reinterpret_cast<const unsigned int*>(&keyTable[index]) +
      1);
109:          }
110:
111:          if (key < *reinterpret_cast<const unsigned int*>(&keyTable[index]))
112:          {
113:              index -= lookupTable[++count];
114:          }
```

```
115:         else
116:         {
117:             index += lookupTable[++count];
118:         }
119:     }
120:     return keyTable.size();
121: }
122:
123: char* AccessFileByRef(ifstream& file, const unsigned int ref, const unsigned
     int entrySize)
124: {
125:     file.clear();
126:     file.seekg(ref * entrySize, ios::beg);
127:     char* entry = new char[entrySize];
128:     file.read(entry, entrySize);
129:     return entry;
130: }
131:
132: void GenerateRandomKeys(const unsigned int maxAmount, const unsigned int
     keysAmount, vector<unsigned int>& buffer)
133: {
134:     buffer.clear();
135:     vector<bool> keyTable(maxAmount, false);
136:     for (unsigned int i = 0; i < keysAmount; i++)
137:     {
138:         unsigned int r = rand() % maxAmount;
139:         int k = 0;
140:         while (keyTable[(r + k) % maxAmount])
141:         {
142:             k *= -1;
143:             if (!keyTable[(r + k) % maxAmount])
144:             {
145:                 break;
146:             }
147:             k *= -1;
148:             k++;
149:         }
150:         buffer.push_back((r + k) % maxAmount);
151:         keyTable[(r + k) % maxAmount] = true;
152:     }
153: }
154:
155: void DisplayTimeDuration(std::chrono::steady_clock::duration duration)
156: {
157:     auto time = chrono::duration_cast<chrono::nanoseconds>(duration);
158:     string timeUnit = "наносекунд";
159:     long double convertedTime = time.count();
160:     if (convertedTime >= 1000)
161:     {
162:         convertedTime /= 1000;
163:         timeUnit = "микросекунд";
164:         if (convertedTime >= 1000)
165:         {
166:             convertedTime /= 1000;
167:             timeUnit = "милисекунд";
168:             if (convertedTime >= 1000)
169:             {
170:                 convertedTime /= 1000;
171:                 timeUnit = "секунд";
172:             }
173:         }
174:     }
175:     cout << "Затраченное время: " << convertedTime << ' ' << timeUnit <<
     ".\n";
176: }
```