Лабораторная работа № 5

Тема: *Эффективные алгоритмы сортировки и поиска.*

Цель: *ознакомиться с понятиями «сортировка» и «поиск», изучить основные алгоритмы сортировки и поиска, научиться применять полученные знания на практике.*

1 Краткая теория

1.1 ОБЩИЕ ПОНЯТИЯ.

Поиск — обработка некоторого множества данных с целью выявления подмножества данных, соответствующего критериям поиска.

Все алгоритмы поиска делятся на

- поиск в неупорядоченном множестве данных;
- поиск в упорядоченном множестве данных.

Упорядоченность – наличие отсортированного ключевого поля.

Сортировка — упорядочение (перестановка) элементов в подмножестве данных по какому-либо критерию. Чаще всего в качестве критерия используется некоторое числовое поле, называемое ключевым. Упорядочение элементов по ключевому полю предполагает, что ключевое поле каждого следующего элемента не больше предыдущего (***сортировка по убыванию***). Если ключевое поле каждого последующего элемента не меньше предыдущего, то говорят о ***сортировке по возрастанию***.

Цель сортировки — облегчить последующий поиск элементов в отсортированном множестве при обработке данных.

Все алгоритмы сортировки делятся на

- алгоритмы внутренней сортировки (сортировка массивов);
- алгоритмы внешней сортировки (сортировка файлов).

Сортировка массивов

Массивы обычно располагаются в оперативной памяти, для которой характерен быстрый произвольный доступ. Основным критерием, предъявляемым к алгоритмам сортировки массивов, является минимизация используемой оперативной памяти. Перестановки элементов нужно выполнять на том же месте оперативной памяти, где они находятся, и методы, которые пересылают элементы из массива A в массив B, не представляют интереса.

Методы сортировки массивов можно разбить на три класса:

- сортировка включениями;
- сортировка выбором;
- сортировка обменом.

Сортировка файлов

Файлы хранятся в более медленной, но более вместительной внешней памяти, на дисках. Однако алгоритмы сортировки массивов чаще всего неприменимы, если сортируемые данные расположены в структуре с

последовательным доступом, которая характеризуется тем, что в каждый момент имеется непосредственный доступ к одному и только одному компоненту.

2.1 СОРТИРОВКА ПУЗЫРЬКОМ / BUBBLE SORT

Будем идти по массиву слева направо. Если текущий элемент больше следующего, меняем их местами. Делаем так, пока массив не будет отсортирован. Заметим, что после первой итерации самый большой элемент будет находиться в конце массива, на правильном месте. После двух итераций на правильном месте будут стоять два наибольших элемента, и так далее. Очевидно, не более чем после n итераций массив будет отсортирован. Таким образом, асимптотика в худшем и среднем случае – $O(n^2)$, в лучшем случае – $O(n)$.

```
void bubblesort(int* l, int* r) {
	int sz = r - l;
	if (sz <= 1) return;
	bool b = true;
	while (b) {
		b = false;
		for (int* i = l; i + 1 < r; i++) {
			if (*i > *(i + 1)) {
				swap(*i, *(i + 1));
				b = true;
			}
		}
		r--;
	}
}
```

2.2 ШЕЙКЕРНАЯ СОРТИРОВКА / SHAKER SORT

(также известна как сортировка перемешиванием и коктейльная сортировка). Заметим, что сортировка пузырьком работает медленно на тестах, в которых маленькие элементы стоят в конце (их еще называют «черепахами»). Такой элемент на каждом шаге алгоритма будет сдвигаться всего на одну позицию влево. Поэтому будем идти не только слева направо, но и справа налево. Будем поддерживать два указателя begin и end, обозначающих, какой отрезок массива еще не отсортирован. На очередной итерации при достижении end вычитаем из него единицу и движемся справа налево, аналогично, при достижении begin прибавляем единицу и двигаемся слева направо. Асимптотика у алгоритма такая же, как и у сортировки пузырьком, однако реальное время работы лучше.

```
void shakersort(int* l, int* r) {
	int sz = r - l;
	if (sz <= 1) return;
	bool b = true;
	int* beg = l - 1;
	int* end = r - 1;
	while (b) {
		b = false;
		beg++;
		for (int* i = beg; i < end; i++) {
```

```
                if (*i > *(i + 1)) {
                    swap(*i, *(i + 1));
                    b = true;
                }
            }
            if (!b) break;
            end--;
            for (int* i = end; i > beg; i--) {
                if (*i < *(i - 1)) {
                    swap(*i, *(i - 1));
                    b = true;
                }
            }
        }
}
```

2.3 СОРТИРОВКА ВСТАВКАМИ / INSERTION SORT

Создадим массив, в котором после завершения алгоритма будет лежать ответ. Будем поочередно вставлять элементы из исходного массива так, чтобы элементы в массиве-ответе всегда были отсортированы. Асимптотика в среднем и худшем случае – $O(n^2)$, в лучшем – $O(n)$. Реализовывать алгоритм удобнее по-другому (создавать новый массив и реально что-то вставлять в него относительно сложно): просто сделаем так, чтобы отсортирован был некоторый префикс исходного массива, вместо вставки будем менять текущий элемент с предыдущим, пока они стоят в неправильном порядке.

```
void insertionsort(int* l, int* r) {
    for (int *i = l + 1; i < r; i++) {
        int* j = i;
        while (j > l && *(j - 1) > *j) {
            swap(*(j - 1), *j);
            j--;
        }
    }
}
```

2.4 СОРТИРОВКА ШЕЛЛА / SHELLSORT

Используем ту же идею, что и сортировка с расческой, и применим к сортировке вставками. Зафиксируем некоторое расстояние. Тогда элементы массива разобьются на классы – в один класс попадают элементы, расстояние между которыми кратно зафиксированному расстоянию. Отсортируем сортировкой вставками каждый класс. В отличие от сортировки расческой, неизвестен оптимальный набор расстояний. Существует довольно много последовательностей с разными оценками. Последовательность Шелла – первый элемент равен длине массива, каждый следующий вдвое меньше предыдущего. Асимптотика в худшем случае – $O(n^2)$. Последовательность Хиббарда – $2^n — 1$, асимптотика в худшем случае – $O(n^{1,5})$, последовательность Седжвика (формула нетривиальна, можете ее посмотреть по ссылке ниже) — $O(n^{4/3})$, Пратта (все произведения степеней двойки и тройки) — $O(n\log^2 n)$. Отмечу, что все эти

последовательности нужно рассчитать только до размера массива и запускать от большего от меньшему (иначе получится просто сортировка вставками). Также я провел дополнительное исследование и протестировал разные последовательности вида $s_i = a * s_{i-1} + k * s_{i-1}$ (отчасти это было навеяно эмпирической последовательностью Циура – одной из лучших последовательностей расстояний для небольшого количества элементов). Наилучшими оказались последовательности с коэффициентами a = 3, k = 1/3; a = 4, k = 1/4 и a = 4, k = -1/5.

```
void shellsort(int* l, int* r) {
	int sz = r - l;
	int step = sz / 2;
	while (step >= 1) {
		for (int *i = l + step; i < r; i++) {
			int *j = i;
			int *diff = j - step;
			while (diff >= l && *diff > *j) {
				swap(*diff, *j);
				j = diff;
				diff = j - step;
			}
		}
		step /= 2;
	}
}
void shellsorthib(int* l, int* r) {
	int sz = r - l;
	if (sz <= 1) return;
	int step = 1;
	while (step < sz) step <<= 1;
	step >>= 1;
	step--;
	while (step >= 1) {
		for (int *i = l + step; i < r; i++) {
			int *j = i;
			int *diff = j - step;
			while (diff >= l && *diff > *j) {
				swap(*diff, *j);
				j = diff;
				diff = j - step;
			}
		}
		step /= 2;
	}
}
int steps[100];
void shellsortsedgwick(int* l, int* r) {
	int sz = r - l;
	steps[0] = 1;
	int q = 1;
	while (steps[q - 1] * 3 < sz) {
		if (q % 2 == 0)
```

```
                steps[q] = 9 * (1 << q) - 9 * (1 << (q / 2)) + 1;
           else
                steps[q] = 8 * (1 << q) - 6 * (1 << ((q + 1) / 2)) +
1;
           q++;
      }
      q--;
      for (; q >= 0; q--) {
           int step = steps[q];
           for (int *i = l + step; i < r; i++) {
                int *j = i;
                int *diff = j - step;
                while (diff >= l && *diff > *j) {
                     swap(*diff, *j);
                     j = diff;
                     diff = j - step;
                }
           }
      }
}
void shellsortpratt(int* l, int* r) {
      int sz = r - l;
      steps[0] = 1;
      int cur = 1, q = 1;
      for (int i = 1; i < sz; i++) {
           int cur = 1 << i;
           if (cur > sz / 2) break;
           for (int j = 1; j < sz; j++) {
                cur *= 3;
                if (cur > sz / 2) break;
                steps[q++] = cur;
           }
      }
      insertionsort(steps, steps + q);
      q--;
      for (; q >= 0; q--) {
           int step = steps[q];
           for (int *i = l + step; i < r; i++) {
                int *j = i;
                int *diff = j - step;
                while (diff >= l && *diff > *j) {
                     swap(*diff, *j);
                     j = diff;
                     diff = j - step;
                }
           }
      }
}
void myshell1(int* l, int* r) {
      int sz = r - l, q = 1;
      steps[0] = 1;
      while (steps[q - 1] < sz) {
           int s = steps[q - 1];
```

```
            steps[q++] = s * 4 + s / 4;
        }
        q--;
        for (; q >= 0; q--) {
            int step = steps[q];
            for (int *i = l + step; i < r; i++) {
                int *j = i;
                int *diff = j - step;
                while (diff >= l && *diff > *j) {
                    swap(*diff, *j);
                    j = diff;
                    diff = j - step;
                }
            }
        }
}
void myshell2(int* l, int* r) {
        int sz = r - l, q = 1;
        steps[0] = 1;
        while (steps[q - 1] < sz) {
            int s = steps[q - 1];
            steps[q++] = s * 3 + s / 3;
        }
        q--;
        for (; q >= 0; q--) {
            int step = steps[q];
            for (int *i = l + step; i < r; i++) {
                int *j = i;
                int *diff = j - step;
                while (diff >= l && *diff > *j) {
                    swap(*diff, *j);
                    j = diff;
                    diff = j - step;
                }
            }
        }
}
void myshell3(int* l, int* r) {
        int sz = r - l, q = 1;
        steps[0] = 1;
        while (steps[q - 1] < sz) {
            int s = steps[q - 1];
            steps[q++] = s * 4 - s / 5;
        }
        q--;
        for (; q >= 0; q--) {
            int step = steps[q];
            for (int *i = l + step; i < r; i++) {
                int *j = i;
                int *diff = j - step;
                while (diff >= l && *diff > *j) {
                    swap(*diff, *j);
                    j = diff;
```

```
                    diff = j - step;
                }
            }
}}
```

2.5 СОРТИРОВКА ДЕРЕВОМ / TREE SORT

Будем вставлять элементы в двоичное дерево поиска. После того, как все элементы вставлены достаточно обойти дерево в глубину и получить отсортированный массив. Если использовать сбалансированное дерево, например красно-черное, асимптотика будет равна O(nlogn) в худшем, среднем и лучшем случае. В реализации использован контейнер multiset.

```
void treesort(int* l, int* r) {
     multiset<int> m;
     for (int *i = l; i < r; i++)
          m.insert(*i);
     for (int q : m)
          *l = q, l++;}
```

2.6 СОРТИРОВКА ВЫБОРОМ / SELECTION SORT

На очередной итерации будем находить минимум в массиве после текущего элемента и менять его с ним, если надо. Таким образом, после i-ой итерации первые i элементов будут стоять на своих местах. Асимптотика: $O(n^2)$ в лучшем, среднем и худшем случае. Нужно отметить, что эту сортировку можно реализовать двумя способами – сохраняя минимум и его индекс или просто переставляя текущий элемент с рассматриваемым, если они стоят в неправильном порядке. Первый способ оказался немного быстрее, поэтому он и реализован.

```
void selectionsort(int* l, int* r) {
     for (int *i = l; i < r; i++) {
          int minz = *i, *ind = i;
          for (int *j = i + 1; j < r; j++) {
               if (*j < minz) minz = *j, ind = j;
          }
          swap(*i, *ind);}}
```

2.7 БЫСТРАЯ СОРТИРОВКА / QUICKSORT

Выберем некоторый опорный элемент. После этого перекинем все элементы, меньшие его, налево, а большие – направо. Рекурсивно вызовемся от каждой из частей. В итоге получим отсортированный массив, так как каждый элемент меньше опорного стоял раньше каждого большего опорного. Асимптотика: O(nlogn) в среднем и лучшем случае, $O(n^2)$. Наихудшая оценка достигается при неудачном выборе опорного элемента. Моя реализация этого алгоритма совершенно стандартна, идем одновременно слева и справа, находим пару элементов, таких, что левый элемент больше опорного, а правый меньше, и меняем их местами. Помимо чистой быстрой сортировки, участвовала в сравнении и сортировка, переходящая при малом количестве элементов на сортировку вставками. Константа подобрана тестированием, а сортировка

вставками — наилучшая сортировка, подходящая для этой задачи (хотя не стоит из-за этого думать, что она самая быстрая из квадратичных).

```
void quicksort(int* l, int* r) {
     if (r - l <= 1) return;
     int z = *(l + (r - l) / 2);
     int* ll = l, *rr = r - 1;
     while (ll <= rr) {
          while (*ll < z) ll++;
          while (*rr > z) rr--;
          if (ll <= rr) {
               swap(*ll, *rr);
               ll++;
               rr--;     }}
     if (l < rr) quicksort(l, rr + 1);
     if (ll < r) quicksort(ll, r);}
void quickinssort(int* l, int* r) {
     if (r - l <= 32) {
          insertionsort(l, r);
          return;}
     int z = *(l + (r - l) / 2);
     int* ll = l, *rr = r - 1;
     while (ll <= rr) {
          while (*ll < z) ll++;
          while (*rr > z) rr--;
          if (ll <= rr) {
               swap(*ll, *rr);
               ll++;
               rr--;}}
     if (l < rr) quickinssort(l, rr + 1);
     if (ll < r) quickinssort(ll, r);}
```

2.8 СОРТИРОВКА СЛИЯНИЕМ / MERGE SORT

Сортировка, основанная на парадигме «разделяй и властвуй». Разделим массив пополам, рекурсивно отсортируем части, после чего выполним процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из этих двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель, соответствующий минимуму. Слияние работает за O(n), уровней всего logn, поэтому асимптотика O(nlogn). Эффективно заранее создать временный массив и передать его в качестве аргумента функции. Эта сортировка рекурсивна, как и быстрая, а потому возможен переход на квадратичную при небольшом числе элементов.

```
     void merge(int* l, int* m, int* r, int* temp) {
     int *cl = l, *cr = m, cur = 0;
     while (cl < m && cr < r) {
          if (*cl < *cr) temp[cur++] = *cl, cl++;
          else temp[cur++] = *cr, cr++;}
     while (cl < m) temp[cur++] = *cl, cl++;
     while (cr < r) temp[cur++] = *cr, cr++;
     cur = 0;
     for (int* i = l; i < r; i++)
```

```
            *i = temp[cur++];}
void _mergesort(int* l, int* r, int* temp) {
     if (r - l <= 1) return;
     int *m = l + (r - l) / 2;
     _mergesort(l, m, temp);
     _mergesort(m, r, temp);
     merge(l, m, r, temp);}
void mergesort(int* l, int* r) {
     int* temp = new int[r - l];
     _mergesort(l, r, temp);
     delete temp;}
void _mergeinssort(int* l, int* r, int* temp) {
     if (r - l <= 32) {
          insertionsort(l, r);
          return;}
     int *m = l + (r - l) / 2;
     _mergeinssort(l, m, temp);
     _mergeinssort(m, r, temp);
     merge(l, m, r, temp);}
void mergeinssort(int* l, int* r) {
     int* temp = new int[r - l];
     _mergeinssort(l, r, temp);
     delete temp;}
```

3.1 БИНАРНЫЙ ПОИСК

Бинарный поиск производится в упорядоченном массиве.

При бинарном поиске искомый ключ сравнивается с ключом среднего элемента в массиве. Если они равны, то поиск успешен. В противном случае поиск осуществляется аналогично в левой или правой частях массива. Алгоритм может быть определен в рекурсивной и нерекурсивной формах. Бинарный поиск также называют поиском методом ***деления отрезка пополам*** или ***дихотомии***.

Количество шагов поиска определится как

$$\log_2 n\uparrow,$$

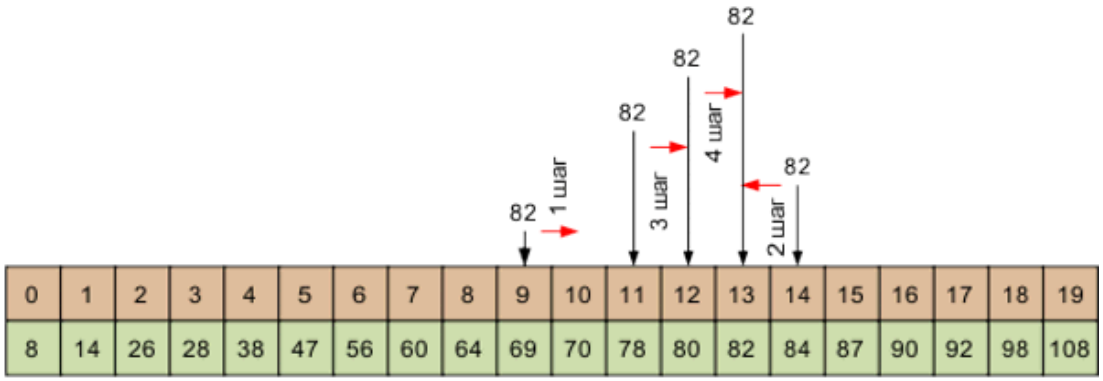
где **n**-количество элементов,

↑ — округление в большую сторону до ближайшего целого числа.

На каждом шаге осуществляется поиск середины отрезка по формуле

$$mid = (left + right)/2$$

Если искомый элемент равен элементу с индексом **mid**, поиск завершается. В случае если искомый элемент меньше элемента с индексом mid, на место mid перемещается правая граница рассматриваемого отрезка, в противном случае — левая граница.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
8	14	26	28	38	47	56	60	64	69	70	78	80	82	84	87	90	92	98	108

- ***Подготовка***. Перед началом поиска устанавливаем левую и правую границы массива:

left = 0, right = 19

- ***Шаг 1***. Ищем индекс середины массива (округляем в меньшую сторону):

mid = (19+0)/2=9

Сравниваем значение по этому индексу с искомым:

69 < 82

Сдвигаем левую границу:

left = mid = 9

- ***Шаг 2***. Ищем индекс середины массива (округляем в меньшую сторону):

mid = (9+19)/2=14

Сравниваем значение по этому индексу с искомым:

84 > 82

Сдвигаем правую границу:

right = mid = 14

- ***Шаг 3***. Ищем индекс середины массива (округляем в меньшую сторону):

mid = (9+14)/2=11

Сравниваем значение по этому индексу с искомым:

78 < 82

Сдвигаем левую границу:

left = mid = 11

- ***Шаг 4***. Ищем индекс середины массива (округляем в меньшую сторону):

mid = (11+14)/2=12

Сравниваем значение по этому индексу с искомым:

80 < 82

Сдвигаем левую границу:

left = mid = 12

- ***Шаг 5***. Ищем индекс середины массива (округляем в меньшую сторону):

mid = (12+14)/2=13

Сравниваем значение по этому индексу с искомым:

82 = 82

Решение найдено!

Чтобы уменьшить количество шагов поиска можно сразу смещать границы поиска на элемент, следующий за серединой отрезка:

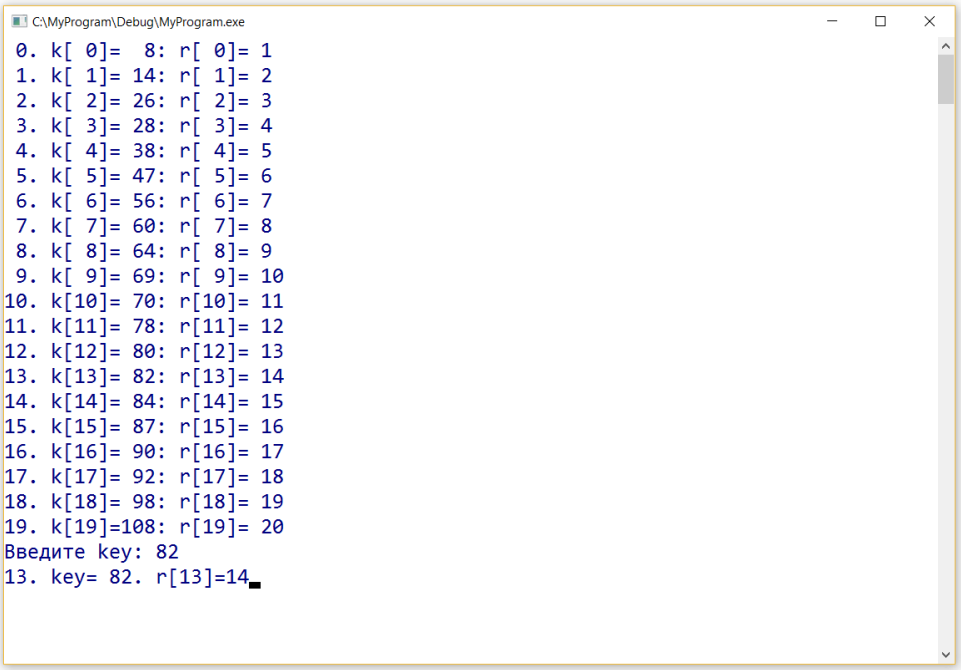
left = mid + 1

right = mid — 1

Реализация бинарного поиска на Си

```c
#define _CRT_SECURE_NO_WARNINGS // для корректной работы scanf()
#include <stdio.h>
#include <stdlib.h> // для использования функций system()
int main()
{
  int k[20]; // массив ключей основной таблицы
  int r[20]; // массив записей основной таблицы
  int key, i;
  system("chcp 1251"); // перевод русского языка в консоли
  system("cls");     // очистка окна консоли
  // Инициализация ключевых полей упорядоченными значениями
  k[0] = 8;  k[1] = 14;
  k[2] = 26;  k[3] = 28;
  k[4] = 38;  k[5] = 47;
  k[6] = 56;  k[7] = 60;
  k[8] = 64;  k[9] = 69;
  k[10] = 70; k[11] = 78;
  k[12] = 80; k[13] = 82;
  k[14] = 84; k[15] = 87;
  k[16] = 90; k[17] = 92;
  k[18] = 98; k[19] = 108;
  // Ввод записей
  for (i = 0; i < 20; i++)
  {
    printf("%2d. k[%2d]=%3d: r[%2d]= ", i, i, k[i], i);
    scanf("%d", &r[i]);
  }
  printf("Введите key: "); // вводим искомое ключевое поле
  scanf("%d", &key);
  int left = 0; // задаем левую и правую границы поиска
  int right = 19;
  int search = -1; // найденный индекс элемента равен -1 (элемент не найден)
  while (left <= right) // пока левая граница не "перескочит" правую
  {
    int mid = (left + right) / 2; // ищем середину отрезка
    if (key == k[mid]) {  // если ключевое поле равно искомому
      search = mid;     // мы нашли требуемый элемент,
      break;            // выходим из цикла
    }
    if (key < k[mid])     // если искомое ключевое поле меньше найденной середины
      right = mid - 1;  // смещаем правую границу, продолжим поиск в левой части
    else                  // иначе
      left = mid + 1;   // смещаем левую границу, продолжим поиск в правой части
  }
  if (search == -1)     // если индекс элемента по-прежнему -1, элемент не найден
    printf("Элемент не найден!\n");
  else          // иначе выводим элемент, его ключ и значение
    printf("%d. key= %d. r[%d]=%d", search, k[search], search, r[search]);
  getchar(); getchar();
  return 0;
}
```

Результат выполнения



3.2 ИНДЕКСНО-ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК.

Для индексно-последовательного поиска в дополнение к отсортированной таблице заводится вспомогательная таблица, называемая ***индексной***.

Каждый элемент индексной таблицы состоит из ключа и указателя на запись в основной таблице, соответствующей этому ключу. Элементы в индексной таблице, как элементы в основной таблице, должны быть отсортированы по этому ключу.

Если индекс имеет размер, составляющий 1/8 от размера основной таблицы, то каждая восьмая запись основной таблицы будет представлена в индексной таблице.

Если размер основной таблицы — **n**, то размер индексной таблицы — **ind_size = n/8**.

Достоинство алгоритма индексно-последовательного поиска заключается в том, что сокращается время поиска, так как последовательный поиск первоначально ведется в индексной таблице, имеющей меньший размер, чем основная таблица. Когда найден правильный индекс, второй последовательный поиск выполняется по небольшой части записей основной таблицы.

Реализация индексно-последовательного поиска на языке Си

```c
#define _CRT_SECURE_NO_WARNINGS // для корректной работы scanf()
#include <stdio.h>
#include <stdlib.h> // для использования функций system()
int main()
{
  int k[20]; // массив ключей основной таблицы
  int r[20]; // массив записей основной таблицы
  int i, j, ind_size;
  int key;
  int kindex[3]; // массив ключей индексной таблицы
  int pindex[3]; // массив индексов индексной таблицы
  system("chcp 1251"); // перевод русского языка в консоли
  system("cls");     // очистка окна консоли
  // Инициализация ключевых полей упорядоченными значениями
  k[0] = 8;  k[1] = 14;
  k[2] = 26;  k[3] = 28;
  k[4] = 38;  k[5] = 47;
  k[6] = 56;  k[7] = 60;
  k[8] = 64;  k[9] = 69;
  k[10] = 70; k[11] = 78;
  k[12] = 80; k[13] = 82;
  k[14] = 84; k[15] = 87;
  k[16] = 90; k[17] = 92;
  k[18] = 98; k[19] = 108;
  // Ввод записей
  for (i = 0; i < 20; i++)
  {
    printf("%2d. k[%2d]=%3d: r[%2d]= ", i, i, k[i], i);
    scanf("%d", &r[i]);
  }
```

```
31    // Формирование индексной таблицы
32    for (i = 0, j = 0; i < 20; i = i + 8, j++)
33    {
34      kindex[j] = k[i]; // переносим каждый 8-й ключ в индексную таблицу
35      pindex[j] = i;    // запоминаем текущий индекс
36    }
37    ind_size = j;  // запоминаем размер индексной таблицы
38    pindex[j] = 20; // последний индекс равен 20
39    // Поиск
40    printf("Введите key: "); // вводим ключевое поле
41    scanf("%d", &key);
42    // Просматриваем элементы индексной таблицы
43    for (j = 0; j < ind_size; j++)
44    {
45      if (key < kindex[j]) // если находим ключ меньше введенного,
46        break;           // выходим из цикла - мы нашли нужную область основной таблицы
47    }
48    if (j == 0)  i = 0;      // присваиваем i начальный индекс диапазона поиска в основ
49    else  i = pindex[j - 1];
50    for (i; i<pindex[j]; i++) // осуществляем поиск в основной таблице
51    {                         //до следующего индекса индексной таблицы
52      if (k[i] == key)  // если найдено введенное значение, выводим его
53        printf("%2d. key=%3d. r[%2d]=%3d", i, k[i], i, r[i]);
54    }
55    getchar(); getchar();
56    return 0;
57  }
```

Результат выполнения

```
C:\MyProgram\Debug\MyProgram.exe
 0. k[ 0]=  8: r[ 0]= 7
 1. k[ 1]= 14: r[ 1]= 3
 2. k[ 2]= 26: r[ 2]= 4
 3. k[ 3]= 28: r[ 3]= 2
 4. k[ 4]= 38: r[ 4]= 6
 5. k[ 5]= 47: r[ 5]= 5
 6. k[ 6]= 56: r[ 6]= 8
 7. k[ 7]= 60: r[ 7]= 9
 8. k[ 8]= 64: r[ 8]= 1
 9. k[ 9]= 69: r[ 9]= 12
10. k[10]= 70: r[10]= 14
11. k[11]= 78: r[11]= 10
12. k[12]= 80: r[12]= 55
13. k[13]= 82: r[13]= 34
14. k[14]= 84: r[14]= 25
15. k[15]= 87: r[15]= 19
16. k[16]= 90: r[16]= 78
17. k[17]= 92: r[17]= 90
18. k[18]= 98: r[18]= 86
19. k[19]=108: r[19]= 100
Введите key: 87
15. key= 87. r[15]= 19
```

! Дополнительный материал !

1. Алгоритмы сортировки и поиска: *https://prog-cpp.ru/algorithm-sort/*
2. Алгоритмы в C++: *https://purecodecpp.com/algoritmy-v-c*

2 Задания

Вариант 1 – нечетный номер в списке всей группы.
Вариант 2 – четный номер в списке всей группы.

Обязательное 1: *Повторить примеры из раздела 3 данной лабораторной работы.*

Обязательное 2: *Написать программу для сортировки массива (заполнение массива рандомно).*

Вариант 1*: сортировка пузырьком и Шелла.*

Вариант 2*: сортировка вставками и Шелла.*

! Контрольные вопросы !

1. Определение понятия поиск.
2. На какие группы делятся алгоритмы поиска?
3. Определение понятия сортировка.
4. На какие группы делятся алгоритмы сортировки?
5. Принцип работы сортировки пузырьком.
6. Принцип работы сортировки выбором.
7. Принцип работы сортировки вставками.
8. Принцип работы сортировки Шелла.
9. Принцип работы индексно-последовательного поиска.
10. Принцип работы бинарного поиска.

Содержание отчёта

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Ответы на контрольные вопросы.
6. Выводы.