11. ИССЛЕДОВАНИЕ АЛГОРИТМОВ ПОИСКА

Цель работы – изучить основные алгоритмы поиска, овладеть практическими навыками разработки, программирования и применения алгоритмов поиска.

11.1 Подготовка к работе

При подготовке к работе необходимо ознакомиться с организацией циклов, изучить операторы для реализации циклов, алгоритмы поиска.

4.2 Теоретические сведения

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.

Поиск в неупорядоченном множестве данных.

Линейный, последовательный поиск (также известен как поиск методом полного перебора) — алгоритм нахождения заданного значения произвольной функции на некотором отрезке. Данный алгоритм является простейшим алгоритмом поиска и, в отличие, например, от двоичного поиска, не накладывает никаких ограничений на функцию и имеет простейшую реализацию. Поиск значения функции осуществляется простым сравнением очередного рассматриваемого значения и, если значения совпадают (с той или иной точностью), то поиск считается завершённым.

Пример. Поиск первого правого вхождения заданного значения в массив.

```
void main()
{
	setlocale (LC_CTYPE,"rus");
	int a[100], n;   // n - количество элементов массива
//формирование массива a
	int key;
	cout<<"\nKey =?";
	cin>> key;
	int i = 0;
	while ( (i< n) && (a[i] != key) )	i++;
	if (i==n)
	cout<<"\nthere is no such element!";
	else
	cout<<"\nnom="<<i+1;
	_getch();	// ждать нажатия любой клавиши
}
```

При необходимости нахождения последнего вхождения заданного значения в массив проверку надо организовать, начиная с последнего элемента массива:

```
	int i = n-1;
	while ( (i>=0) && (a[i] != key) ) i--;
```

Данный код представляет собой самый простой поиск элемента в массиве. Однако в нем не рассматривается случай, когда в массиве не единственный элемент равный *key*. Будем считать все случаи, включая наихудший, в котором все элементы массива равны *key*. Введем массив *b*, в который будем сохранять позиции элементов массива, значения которых равны *key*.

```
void main()
{
	setlocale (LC_CTYPE,"rus");
	int a[100],b[100];
	int n,i;
	//формирование массива a
	int key;
	cout<<"\nKey =?";
	cin>> key;
	int i, nom=-1;
	for(i=0; i<n; i++)
		if(a[i]== key) b[++nom]=i;
	if(nom !=-1)
	{
		cout<<"\nnom=";
		for(i=0; i<=nom; i++) cout<<b[i]<<"  ";
	}
	else
		cout<<"\nthere is no such element! ";
	_getch();	// ждать нажатия любой клавиши
}
```

Поиск в упорядоченном множестве данных.

Двоичный (бинарный) поиск (также известен как метод деления отрезка пополам и дихотомия) — классический алгоритм поиска элемента в отсортированном массиве (векторе), использующий дробление массива на половины. Используется в информатике, вычислительной математике и математическом программировании.

Алгоритм. Массив делится пополам *s=(l+r)/2 (*где *l, r* – левая и правая границы массива соответственно*)* и определяется, в какой части массива находится нужный элемент *key*. Т.к. массив упорядочен, то если *a[s] < key*, то искомый элемент находится в правой части массива, иначе – находится в левой части. Производится соответствующая коррекция границ массива. Выбранную часть массива снова надо разделить пополам и т.д., до тех пор, пока границы отрезка *l* и *r* не станут равны.

Для того, чтобы найти нужную запись в таблице, в худшем случае нужно $\log_2(N)$ сравнений (округление производится в большую сторону до ближайшего целого числа). Это значительно лучше, чем при последовательном поиске.

Приведем иллюстрация бинарного поиска на примерах.

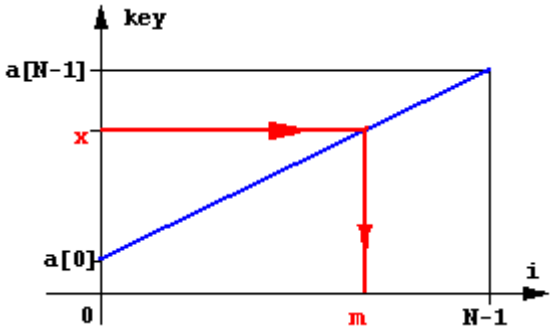
1) Вариант 1.

```
void main()
{
	setlocale (LC_CTYPE,"rus");
	int a[100];
	int n,I;
```

```
//формирование массива a
        cout<<"\nEnter the size of array:";
        cin>>n;
        for( I=0; I<n; I++) cin>>a[I];
        int key;
        cout<<"\nKey =?";
        cin>> key;
        int l=0, r=n-1, s;
        bool Found = false;             // флаг
        while ( (l <=r) && !Found)   // цикл, пока интервал поиска не
// сузиться до 0
{
                s = ( l + r ) / 2;          // середина интервала
                if ( a[s] == key )
                Found = true;                       // ключ найден
                else
                        if ( a[s] < key )
                                l = s + 1;                  // поиск в правом подинтервале
                        else
                                r = s - 1;                  // поиск в левом подинтервале
        }
         if (Found == true)  cout<< "Элемент найден на позиции "<< s+1<< " \n";
         else cout<< "Элемент не найден!\n";

        _getch();       // ждать нажатия любой клавиши
}


        2) Вариант 2.

void main()
{
        setlocale (LC_CTYPE,"rus");
        int a[100];
        int n,I;
        //формирование массива a
        cout<<"\nEnter the size of array:";
         cin>>n;
        for( I=0; I<n; I++) cin>>a[I];
        int key;
        cout<<"\nKey =?";
        cin>> key;
        int l=0, r=n-1, s;
        do
        {
                s=(l+r)/2;                  //найти средний элемент
                if (a[s]< key) l=s+1;   //перенести левую границу
                else r=s;                   //перенести правую границу
        }
        while(l!=r);
        if(a[l]== key) cout<< "Элемент найден на позиции "<< l+1<< " \n";
        else cout<< "Элемент не найден!\n";
```

```
    _getch();        // ждать нажатия любой клавиши
}
```

Метод интерполяции



Если нет никакой дополнительной информации о значении ключей, кроме факта их упорядочения, то можно предположить, что значение *key* увеличиваются от *a[0]* до *a[N-1*] более или менее "равномерно". Это означает, что значение среднего элемента *a[N/2]* будет близким к среднему арифметическому между наибольшим и наименьшим значением. Но, если искомое значение *key* отличается от указанного, то есть некоторый смысл для проверки брать не средний элемент, а "средне-пропорциональный", то есть тот, номер которого пропорционален значению *key*:

```
s=l+(key-a[l])*(r-l)/(a[r]-a[l]);        // "средне-пропорциональный"
```

Выражение для текущего значения i получено из пропорциональности отрезков на рисунке: (a[r]-key)/(key-a[l])=(r-s)/(s-l);

В среднем этот алгоритм должен работать быстрее бинарного поиска, но в худшем случае будет работать гораздо дольше.

11.3 Варианты заданий

Выполнить поиск заданного значения в массиве, используя линейный и бинарный алгоритмы поиска в соответствии с вариантом.

Таблица 11.1 – Варианты заданий

№ вар.	Тип и размер массива	Линейный поиск	Двоичный поиск (классический алгоритм)	Двоичный поиск (метод интерполяции)
1.	целые числа *X(n)*	первое вхождение	последнее вхождение	-
2.	действительные числа *B(20)*	последнее вхождение	-	первое вхождение
3.	целые числа *A(k)*	все вхождения	последнее вхождение	-
4.	целые числа *C(25)*	первое вхождение	все вхождения	-
5.	целые числа *D(n)*	последнее вхождение	первое вхождение	-
6.	действительные числа *B(20)*	все вхождения	последнее вхождение	-
7.	целые числа *A(15)*	первое вхождение	-	все вхождения
8.	целые числа *C(25)*	последнее вхождение	-	первое вхождение

9.	целые числа *K(n)*	все вхождения	первое вхождение	-
10.	действительные числа *L(20)*	первое вхождение	последнее вхождение	
11.	целые числа *D(k)*	последнее вхождение	все вхождения	-
12.	целые числа *M(15)*	все вхождения	первое вхождение	-
13.	целые числа *Y(n)*	первое вхождение	-	последнее вхождение
14.	действительные числа *B(10)*	последнее вхождение	-	все вхождения
15.	целые числа *X(n)*	все вхождения	первое вхождение	-
16.	действительные числа *A(20)*	первое вхождение	последнее вхождение	-
17.	целые числа *A(k)*	последнее вхождение	-	первое вхождение
18.	целые числа *E(25)*	все вхождения	-	последнее вхождение
19.	целые числа *X(n)*	первое вхождение	последнее вхождение	-
20.	действительные числа *L(20)*	последнее вхождение	все вхождения	-
21.	целые числа *X(n)*	все вхождения	-	последнее вхождение
22.	действительные числа *F(20)*	первое вхождение	-	все вхождения
23.	целые числа *A(k)*	последнее вхождение	-	первое вхождение
24.	целые числа *K(20)*	все вхождения	-	первое вхождение
25.	целые числа *F(n)*	первое вхождение	все вхождения	-

11.4 Контрольные вопросы

1. Классификация алгоритмов поиска.
2. Алгоритм линейного поиска.
3. Преимущества и недостатки линейного поиска.
4. Какие алгоритмы поиска в отсортированном массиве Вам известны?
5. Алгоритм бинарного поиска.
6. Применение алгоритмов бинарного поиска?
7. Алгоритм индексно-последовательного поиска.