ШАГ компьютерная академия

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С

Урок №10

Програмиране на език C

Съдържание

1. Линейно търсене

В дадения урок ще говорим за алгоритмите за търсене и сортировка, Вие вероятно, вече сте успели да се сблъснете с необходимостта да подредите своя масив или бързо да намерите в него някакви данни.

За начало, ще разгледаме най-простите от методите за търсене на данни – линейно търсене.

Дадения алгоритъм сравнява всеки елемент на масива с ключа, предоставен за търсене. Нашия експериментален масив не е подреден и може да се получи ситуация в която търсеното значение да се окаже първо в масива. Но, като цяло, програмата реализираща линейното търсене, ще сравни с ключа за търсене половината елементи на масива.

```
#include <iostream>

using namespace std;

int LinearSearch (int array[], int size,
   int key){ for(int i=0;i<size;i++)
      if(array[i] ==
         key) return i;
   return -1;
}

void main()
{
   const int arraySize=100;
   int a[arraySize], searchKey, element;
   for(int x=0;x<arraySize;x++)
         a[x]=2*x;

    //Следващия ред изкарва на екрана
    съобщението //Моля въведете ключ:
    cout<<"Please, enter the key:       ";
```

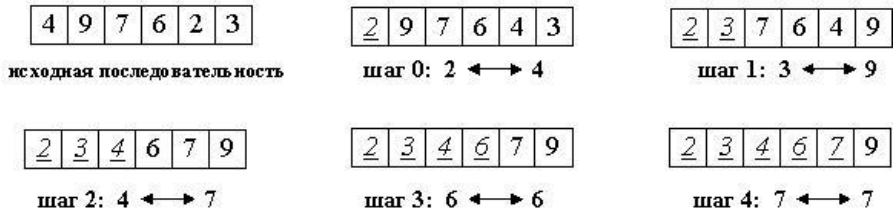
```
   cin>>searchKey;
   element=LinearSearch(a, arraySize, searchKey);

   if(element!=-1)
      //Следващия ред изкарва на екрана
      съобщение //Намерено значение в елемента
     cout<<"\nThe key was found in element "<<element<<'\n';

    //Следващия ред изкарва на екрана
    съобщението //Няма намерено значение
   else
     cout<<"\nValue not found ";

}
```

Ще отбележим, че алгоритъма на линейното търсене отлично работи само за неголеми или не подредени масиви и е абсолютно надежно.

2. Сортиране чрез пряка селекция

Идеята на дадения метод се състои в това, че за да създава сортирана последователност по пътя на присъединението към нея на един елемент след друг в правилен ред.

Сега с вас ще пробваме да построим готова последователност, започвайки от левия край на масива. Алгоритъма се състои от n последователни стъпки, започвайки от нулевия и завършвайки с (n -1). На стъпката i-м избираме най-малкия от елементите a[i] ... a[n] и му сменяме местата с a[i]. Последователността на стъпките при n=5 е изобразена на рисунките долу.



Независимо от номера на текущата стъпка i, последователността a[0] …a[i] е подредена. По такъв начин, на стъпката (n-1) цялата последователност, освен a[n] се оказва сортирана, а a[n] стои на последно място по право: всичките по-малки елементи са отишли в ляво. Да разгледаме пример, реализиращ дадения метод:

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
template <class T>
void selectSort(T a[], long size) {
    long i, j, k;
    T x;
                                  // i - номер на текущата
    for(i=0;i<size;i++) {         стъпка
       k=i;
       x=a[i];
        // цикъл за избор на най-малкия елемент
       for(j=i+1;j<size;j++)
          if(a[j]<x){
            k=j;
            x=a[j];
               // k - индекс на най-малкия елемент
          }
       a[k]=a[i];    // сменяме местата на най-малкия с
       a[i]=x;       a[i]
   }
}
void main(){
        srand(time(NULL));
        const long SIZE=10;
        int ar[SIZE];
          // преди сортирането
        for(int i=0;i<SIZE;i++){
           ar[i]=rand()%100;
           cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
        selectSort(ar,SIZE);

           // след сортирането
        for(int i=0;i<SIZE;i++){
           cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
}
```

Основни принципи на метода

1. За намиране на най-малкия елемент от n+1 алгоритъма извършва n сравнения. По такъв начин, след като числото за обмяна е винаги по-малко от числото за сравнение, времето за сортиране расте относително с количествата елементи.

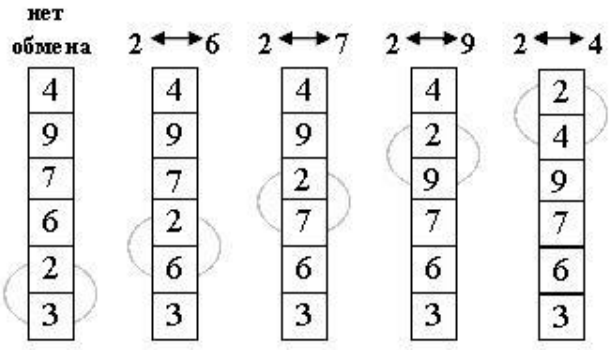
2. Алгоритъма не използва допълнителна памет: всички операции протичат „на място“.

Нека да определим, колко е устойчив дадения метод. Да разгледаме последователността от три елемента, всеки от които има две полета, а сортирането върви по първото от тях. Резултата от нейното сортиране може да се види след стъпка 0, след като повече обмени няма да има. Реда на ключовете 2a, 2b е променен на 2b, 2a.

Таким образом, входная последовательность почти упорядочена, то сравнений будет столько же, значит алгоритм ведет себя не очень оптимально. Однако, такую сортировку можно использовать для массивов имеющих небольшие размеры.

3. Метод на мехурчето

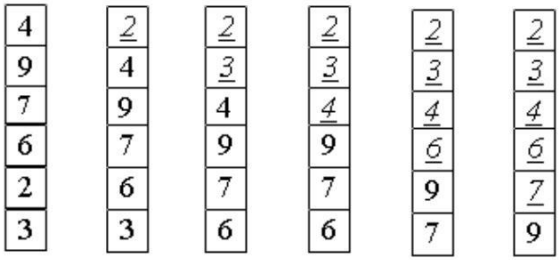
Идеята на метода се състои в следното: стъпка от сортирането се състои от преминаването от долу на горе по масива. По пътя се разглеждат двойки от съседните елементи. Ако елементите на някоя двойка се намират в неправилен порядък, то им сменяме местата. За реализация ще разположим масива от горе на долу, от нулевия елемент към последния.. След нулевото преминаване по масива най-отгоре се оказва най-„лекия“ елемент – от тук идва и аналогията с балончетата. Следващото преминаване стига до втория от горе на долу елемент, по този начин втория по величина елемент се качва на правилната позиция



Нулевой проход, сравниваемые пары выделены

Правим преходи по все смаляващата се долна част на масива докато в нея не остане само един елемент. Тогава сортирането приключва.

Пример за код:




```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
template <class T>
void bubbleSort(T a[], long size){
    long i, j;
    T x;                          // i - номер на
    for(i=0;i<size;i++){          прехода
                                 // вътрешен цикъл на
       for(j=size-1;j>i;j--){    прехода
          if(a[j-1]>a[j]){
            x=a[j-1];
            a[j-1]=a[j];
            a[j]=x;
          }
       }
    }
}

void main(){
    srand(time(NULL));
    const long SIZE=10;
    int ar[SIZE];
     // преди сортирането
    for(int i=0;i<SIZE;i++){
       ar[i]=rand()%100;
       cout<<ar[i]<<"\t";
    }
    cout<<"\n\n";
    bubbleSort(ar,SIZE);

       // след сортирането
    for(int i=0;i<SIZE;i++){
       cout<<ar[i]<<"\t";
    }
    cout<<"\n\n";
}
```

Основни принципи на метода.

Средното число за сравнение и обмен има квадратичен порядък на растеж, от тук може и да се изхожда, че балонения алгоритъм е много бавен и неефективен. Въпреки това, той има огромен плюс: той е елементарен и може да бъде подобряван по всякакъв начин. С което сега ще се и заемем.

Да разгледаме ситуация в която на който и да е от преходите не се е случил нито един обмен. Това означава, че всички двойки са разположени в правилен ред, така че масива вече е сортиран. Да се продължава процеса няма смисъл. По такъв начин, първата стъпка за оптимизация е запомнянето, дали на дадения преход се е случил някакъв обмен. Ако не, алгоритъма приключва работата си.

Процеса за оптимизация може да продължи, запомняйки не само обмена като факт, но и индекса на последния обмен k. Действително: всички двойки на съседните елементи с индекс, по-малък от k, вече са разположени в нужния ред. Следващите преходи могат да приключват на индекс k, вместо да се движат до установената по-рано горна граница i.

Друго качествено подобрение на алгоритъма може да се получи от следното наблюдение. Въпреки че „лекото“ балонче от долу се качва нагоре за един преход, „тежките“ балони се спускат с минимална скорост: една стъпка за итерация. Така че масива 2 3 4 5 6 1 ще бъде сортиран за 1 преход, а сортирането на последователността 6 1 2 3 4 5 ще заеме 5 прехода.

За да с е избегне подобен ефект, може да се променя направлението на всеки следващ един след друг преход. Получения алгоритъм понякога се нарича „шейкър-сортиране“.

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;
template <class T>
void shakerSort(T a[], long
  size) { long j, k=size-1;
 long lb=1, ub=size-1; // граници на несортираните части на масива
  T x;

  do{
       // преход от долу на
      горе for(j=ub;j>0;j--){
         if(a[j-1]>a[j]){
           x=a[j-1]; a[j-
           1]=a[j];
           a[j]=x;
           k=j;
         }
      }
      lb = k+1;
       // преход от горе на
      долу
      for(j=1;j<=ub;j++){
         if(a[j-1]>a[j]){
           x=a[j-1]; a[j-
           1]=a[j];
           a[j]=x;
           k=j;
         }
       }
      ub=k-1;
  }while (lb<ub);
}
void main(){
   srand(time(NULL));
   const long SIZE=10;
   int ar[SIZE];
    // преди сортирането
   for(int i=0;i<SIZE;i++){
      ar[i]=rand()%100;
      cout<<ar[i]<<"\t";
   }
   cout<<"\n\n";
   shakerSort(ar,SIZE);
      // след сортирането
   for(int i=0;i<SIZE;i++){
      cout<<ar[i]<<"\t";
   }
   cout<<"\n\n";
}
```

4. Сортиране чрез вмъкване

Сортиране с елементарни вмъквания до някаква степен прилича на методите описани в предишните раздели на урока. По аналогичен начин се правят преходи по части от масива, и по аналогичен начин в неговото начало израства сортирана последователност.

Обаче в балоненото сортиране или изборното може да се зави точни, че на стъпката i елементите a[0] …a[i] стоят на правилните места и никъде повече няма да се преместят. Тук същото утвърждение ще бъде по-слабо: последователността a[0] …a[i] е подредена. При това по хода на алгоритъма в нея ще бъдат вмъквани всички нови елементи.

Ще разглобим алгоритъма и ще разгледаме действията му на стъпка i. Както е казано горе, последователността до този момент е разделена на две части: готова a[0]...a[i]
и неподредена a[i+1]...a[n]. На следующем, (i+1)-м каждом шаге алгоритма берем a[i+1] и вмъкваме на нужното място в готовата част на масива. Търсенето на подходящо място за поредния елемент на входната последователност се осъществява чрез последователни сравнения с елемента, стоящ пред него. В зависимост от резултата, сравнението на елемента или остава на текущото място, или си сменят местата и процеса се повтаря .

Вставка числа 2 в отсортированную подпоследовательность. Сравниваемые пары выделены.

По такъв начин, в процеса на вмъкването ние „отсейваме“ елемента x към началото на масива, спирайки се в случаите, когато е намерен елемент по-малък от x или е достигнато началото на последователността.

Реализация на метода

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

template <class T>
void insertSort(T a[], long
       size) { T x;
       long i, j;
         // цикъл на прехода, i –
       номер на прехода
       for(i=0;i<size;i++){
          x=a[i];

             // търсене мястото на елемента в готова
          последователност for(j=i-1;j>=0&&a[j]>x;j--)
               // местим елемента на дясно докато
            не стигнем a[j+1]=a[j];
               // мястото е намерено да се вмъкне елеметна
            a[j+1] = x;
       }
}

void main(){
       srand(time(NULL));
       const long SIZE=10;
       int ar[SIZE];
```

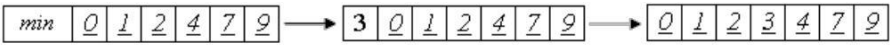
```
        // преди сортирането
        for(int i=0;i<SIZE;i++){
                ar[i]=rand()%100;
                cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
        shakerSort(ar,SIZE);

        // след сортирането
        for(int i=0;i<SIZE;i++){
                cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
}
```

Принципи на метода

Добър показател за сортиране е едно съвсем естествено поведение: почти сортирания масив ще бъде досортиран много бързо. Това в съвкупност с устойчивостта на алгоритъма, прави метода добър избор в съответстващите ситуации. Обаче, алгоритъма може винаги да бъде подобрен. Ще отбележим, че на всяка стъпка от вътрешния цикъл се проверяват 2 условия. Може .

Тогда при j=0 будет заведомо верно a[0] <=x. Цикъла ще спре на нулевия елемент, което е било и целта на условието j>=0. По такъв начин, сортирането ще минава по правилен начин, а във вътрешния цикъл ще стане едно сравнение по-малко. Обаче сортирания масив ще бъде непълен, понеже от него е изчезнало първото число. За

завършване на сортирането, това число трябва да се върне назад, а след това да се вмъкне в сортираната последователност a[1]...a[n].




```
#include <iostream>
#include <stdlib.h>
#include <time.h>
using namespace std;

template <class T>
void setMin(T a[],long
       size){ T min=a[0];
       for(int i=1;i<size;i++)
              if(a[i]<min)
                     min=a[i];
       a[0]=min;
}

template <class T>
void insertSortGuarded(T a[], long
       size) { T x;
       long i, j;
        // да се съхрани стария първи елемент
       T backup = a[0];
        // заменить на минимальный
       setMin(a,size);
        // сортиране на масива
       for(i=1;i<size;i++){
          x = a[i];
          for(j=i-1;a[j]>x;j-
             -) a[j+1]=a[j];
          a[j+1] = x;
       }
        // вмъкване на backup на правилното
       място for(j=1;j<size&&a[j]<backup;j++)
          a[j-1]=a[j];
        // вмъкване на елемента
       a[j-1] = backup;
}
```

```
void main(){
        srand(time(NULL));
        const long SIZE=10;
        int ar[SIZE];
          // преди сортирането
        for(int i=0;i<SIZE;i++){
                ar[i]=rand()%100;
                cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
        insertSortGuarded(ar,SIZE);

          // след сортирането
        for(int i=0;i<SIZE;i++){
                cout<<ar[i]<<"\t";
        }
        cout<<"\n\n";
}
```

5. Домашна работа

1. Даден е масив от числа с размер 10 елемента. Да се напише функция, която да сортира масива по нарастване или по смаляване, в зависимост от третия параметър на функцията. Ако той е равен на 1, сортирането да е по смаляване, ако е 0, тогава по нарастване. Първите 2 параметъра на функцията – това са масива и неговия размер, третия параметър по подразбиране е равен на 1.
2. Даден е масив от случайни числа в диапазона от -20 до +20. Да се намерят позициите на крайните отрицателни елементи (най-левия отрицателен елемент и най-десния отрицателен елемент) и да се сортират елементите намиращи се между тях.
3. Даден е масив от 20 цели числа със значения от 1 до 20:

Необходимо е:

- да се напише функция, разхвърляща елементите на масива по произволен начин;
- да се създаде случайно число от същия диапазон и да се намери позицията на това число в масива;
- да се сортират елементите на масива, намиращи се ляво от намерената позиция, по смаляване, а елементите намиращи се от дясно, по нарастване.