ХИМИКОТЕХНОЛОГИЧЕН И МЕТАЛУРГИЧЕН УНИВЕРСИТЕТ - СОФИЯ

Алгоритми и структури данни

лектор: доц. д-р инж. Атанас Атанасов
Катедра “Информатика”

Лекция 9

АЛГОРИТМИ ЗА ТЪРСЕНЕ

Алгоритми за търсене

- **Въведение**
- **Видове търсене**
  - Линейно търсене
  - Двоично търсене
  - Търсене чрез интерполиране
  - Троично търсене
  - Търсене чрез Хаш таблица

Алгоритми за търсене - Въведение

- Търсенето е процедура, при която елементите на дадено множество или структура от данни като масив, списък, дърво и др. се обхождат с цел намиране на съвпадение по стойност между елементите на множеството и зададен елемент.
- В стандартния случай се изисква елементите на множеството да са предварително сортирани
- При търсенето е възможно да се намерят едно или няколко съвпадения или търсеният елемент да не се намери.
- Търсенето се използва за намиране на някаква информация по ключова стойност. Например при търсене на данни за човек по ЕГН или на думи в речник и др.

Линейно търсене

Линейното търсене е най-простият алгоритъм за търсене. При него се извършва последователно търсене по всички елементи, един по един, на даден масив. Всеки елемент на масива се проверява и ако се намери съвпадение, тогава този конкретен елемент, както и поредния му номер се връщат като резултат, в противен случай търсенето продължава до края на масива.

Масивът, в който се извършва търсенето може да не е сортиран.

Алгоритъм

Linear Search (Array A, Value x)
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Линейно търсене

Линейното търсене е най-простият алгоритъм за търсене. При него се извършва последователно търсене по всички елементи, един по един, на даден масив. Всеки елемент на масива се проверява и ако се намери съвпадение, тогава този конкретен елемент, както и поредния му номер се връщат като резултат, в противен случай търсенето продължава до края на масива.

Масивът, в който се извършва търсенето може да не е сортиран.

Алгоритъм

Linear Search (Array A, Value x)
Step 1: Set i to 1
Step 2: if i > n then go to step 7
Step 3: if A[i] = x then go to step 6
Step 4: Set i to i + 1
Step 5: Go to Step 2
Step 6: Print Element x Found at index i and go to step 8
Step 7: Print element not found
Step 8: Exit

Линейно търсене

Пример 1:

Да се състави програма за линейно търсене на дадено число в елементите на даден масив.

```
#include <iostream>
using namespace std;
int linear_search(int arr[], int x, int n);
int main(){
    int x[100],  y,  i, n, rez;
    cout<<"Broj na elementite na masiva n<=100: ";   cin>>n;
    for(i=0;i<n;i++ ){
      cout<<"Vavedete X["<<i<<"] = ";  cin>>x[i];     }
    cout<<" Searched value ="; cin>>y;
    rez=linear_search(x,y,n);
if (rez >= 0 ) {cout<<" Index = "<<rez<<" Value =
"<<x[rez]<<endl;}
else { cout <<"Value not found in array";}

return 0;
}
int linear_search(int arr[] , int x, int n)
{     for(int i = 0 ; i < n; i++){
        if(arr[i]==x){ return i; }
      }
return -1;
}
```

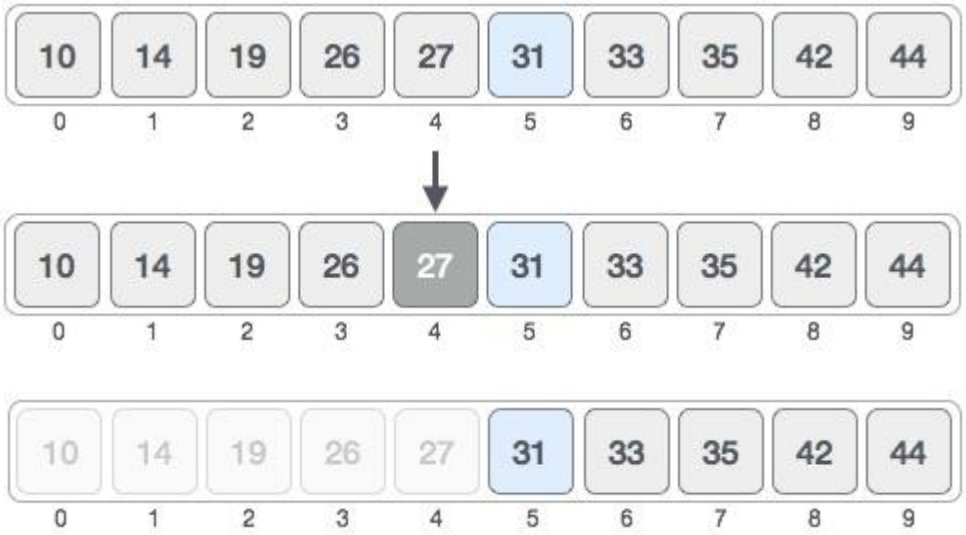
Двоично търсене

Двоичното търсене търси определен елемент чрез сравняване със средния елемент на масива. Ако се получи съвпадение, индексът на намерения елемен се връща. Ако средният елемент е по-голям от търсения, тогава елементът се търси в подмасива вляво от средния елемент. В противен случай елементът се търси в подмасива вдясно от средния елемент. Този процес се прилага и върху всеки нов под-масив, докато размерът на под-масива се намали до нула.

Масивът, в който се извършва търсенето **трябва** да е сортиран.

Ако в масива по-долу търсим числото 31 последователността е следната:

Намираме средата на масива
mid = low + (high - low) / 2
mid = 0 + (9 - 0) / 2 = 4 (индекс 4)
31 е по-голямо от 27 и търсим вдясно
low = mid + 1 = 5
mid = low + (high - low) / 2
mid = 5 + (9 - 5)/2 = 7



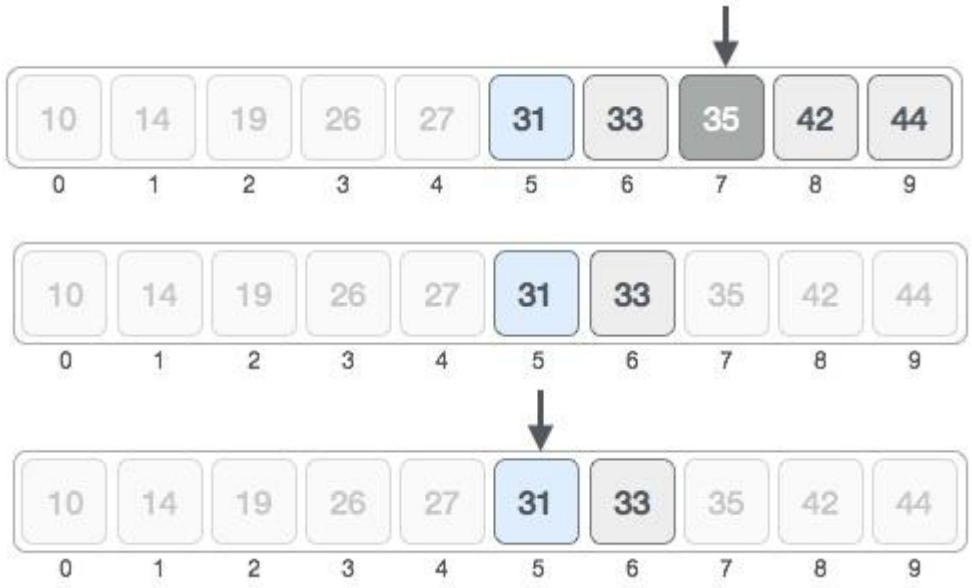
Двоично търсене

Двоичното търсене търси определен елемент чрез сравняване със средния елемент на масива. Ако се получи съвпадение, индексът на намерения елемен се връща. Ако средният елемент е по-голям от търсения, тогава елементът се търси в подмасива вляво от средния елемент. В противен случай елементът се търси в подмасива вдясно от средния елемент. Този процес се прилага и върху всеки нов под-масив, докато размерът на под-масива се намали до нула.

Масивът, в който се извършва търсенето **трябва** да е сортиран.

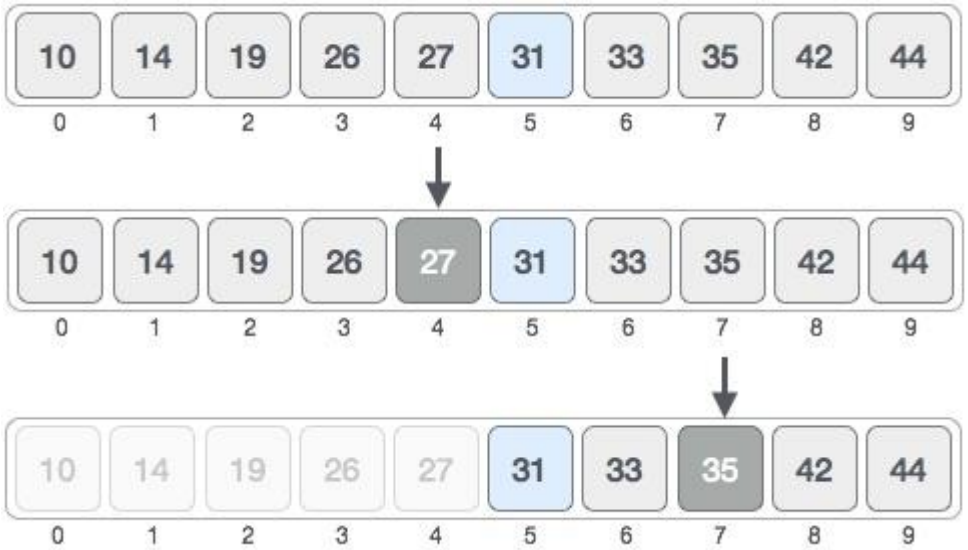
Ако в масива по-долу търсим числото 31 последователността е следната:

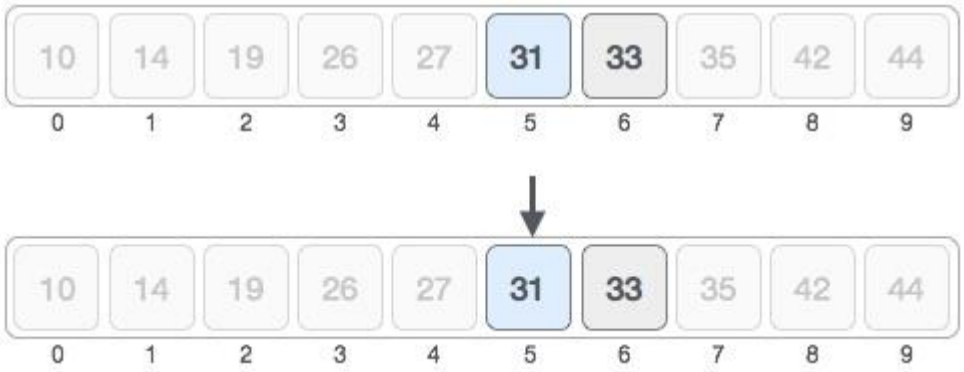
Намираме mid
low = mid + 1 = 5
mid = low + (high - low) / 2
mid = 5 + (9 - 5)/2 = 7
Сравняваме 31 с елемент 7 (35)
Трябва да търсим вляво
high = mid - 1 = 6
mid = low + (high - low) / 2
mid = 5 + (6 - 5)/2 = 5
Сравняваме 31 с елемент 5 (31)
Елементът е намерен в позиция (индекс) 5



Двоично търсене

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched
   Set lowerBound = 1
   Set upperBound = n
   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.

      set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

      if A[midPoint] < x
         set lowerBound = midPoint + 1

      if A[midPoint] > x
         set upperBound = midPoint - 1

      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while
end procedure
```





Двоично търсене 1/2

Пример 1:

Да се състави програма за двоично търсене на дадено число в елементите на даден масив.

!!! Елементите на масива трябва да сортирани или всеки един да е по-голям от предходния.

```
#include <iostream>
using namespace std;

int BinarySearch(int array[], int left, int right, int element)
{
   int middle;
   while (left <= right){
      middle = left + (right - left)/2;
      if (array[middle] == element)
         return middle;
      if (array[middle] < element)
         left = middle + 1;
      else
         right = middle - 1;
   }
   return -1;
}
```

Двоично търсене 2/2

Пример 1:

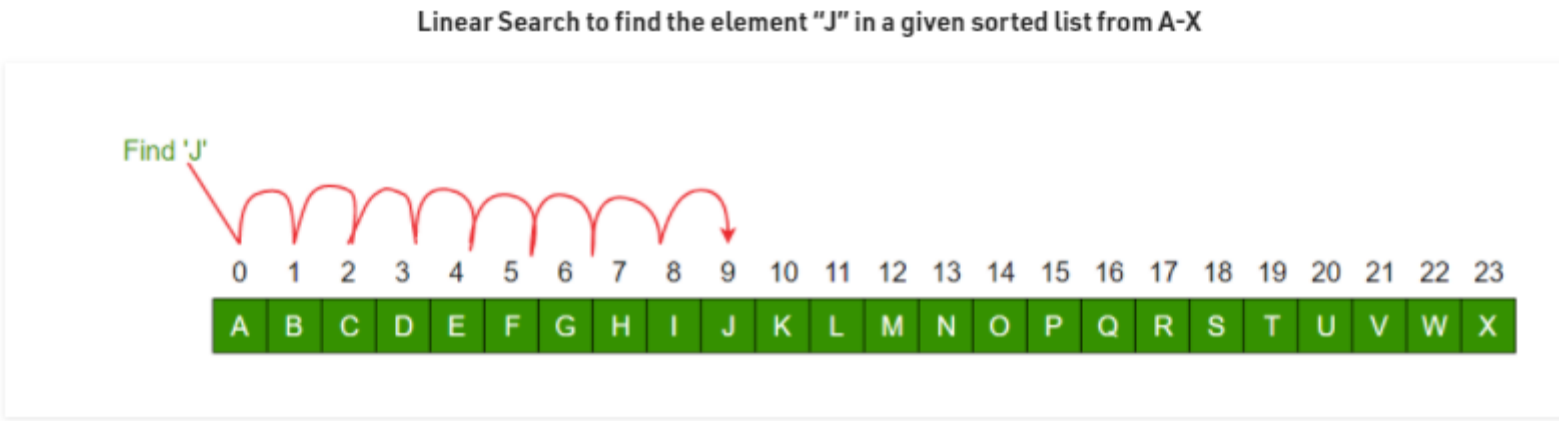
Да се състави програма за двоично търсене на дадено число в елементите на даден масив.

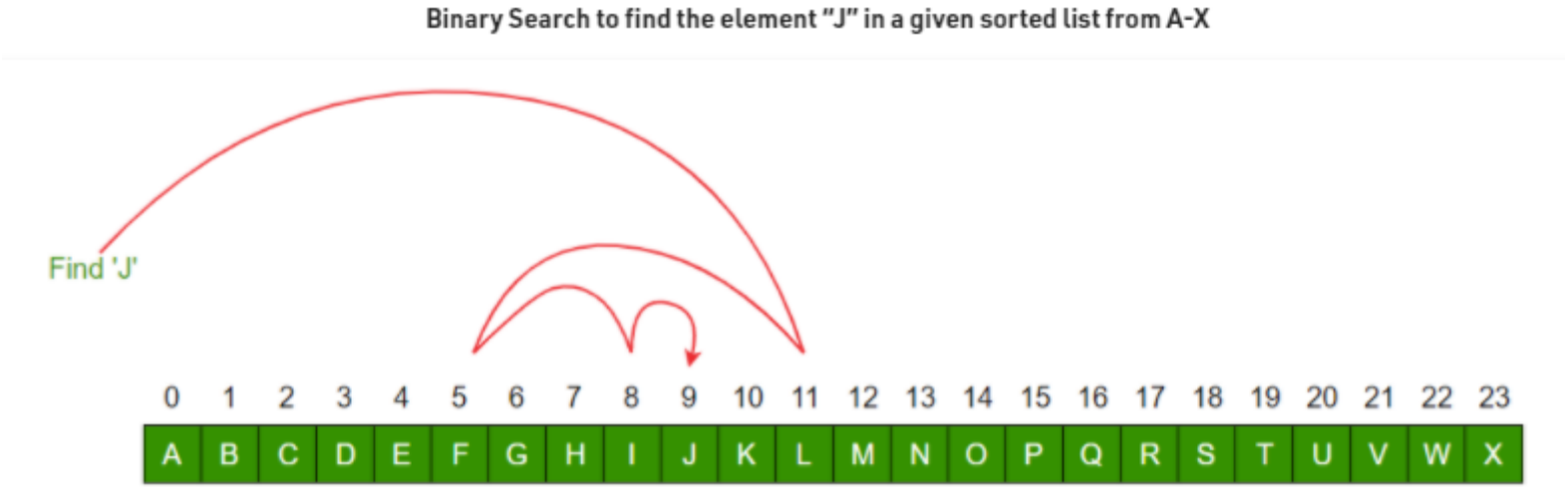
!!! Елементите на масива трябва да сортирани или всеки един да е по-голям от предходния.

```
// продължение на програмата
int main(){
   int array[100] ;
   int i, n, element ;
   int index;
   cout<<"Broj na elementite na masiva n<=100: ";   cin>>n;
    for(i=0;i<n;i++ ){
          cout<<"Vavedete array["<<i<<"] = ";
          cin>>array[i];
    }
    cout<<" Searched value =";
    cin>>element;
    index = BinarySearch(array, 0, n-1, element);
    if( index == -1 ) {
         cout<<"Element not found in the array ";
    }
    else {
         cout<<"Element found at index "<<index;
    }
   return 0;
}
```

Двоично търсене

Сравнение на алгоритмите на линейно и двоично търсене





Търсене чрез интерполиране

Търсенето чрез интерполиране е вариация на двоичното търсене. При това търсене позицията на средния елемент в масива, на базата на който масивът се разделя на два подмасива се определя по формулата:

mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

където :

A е масив,

X е търсеният елемент

Lo е най-малкият индекс на масива

Hi е най-големият индекс на масива

A[n] е стойността в позиция n на масива A

Изборът на ляв или десен под масив е аналогичен на този при двоичното търсене и продължава докато дължината му стане 0.

Масивът, в който се извършва търсенето **трябва** да е сортиран.

Търсене чрез интерполиране

```
A → Array
N → Size of A
X → Target Value
Procedure Interpolation_Search()
  Set Lo  →  0
  Set Mid → -1
  Set Hi  →  N-1
  While X does not match
      if Lo equals to Hi OR A[Lo] equals to A[Hi]
      EXIT: Failure, Target not found
    end if
    Set Mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])
    if A[Mid] = X
      EXIT: Success, Target found at Mid
    else
      if A[Mid] < X
        Set Lo to Mid+1
      else if A[Mid] > X
        Set Hi to Mid-1
      end if
    end if
  End While
End Procedure
```

Търсене чрез интерполиране1/2

Пример 1:

Да се състави програма за търсене чрез интерполиране на дадено число в елементите на даден масив.

!!! Елементите на масива трябва да сортирани или всеки един да е по-голям от предходния.

```
#include<bits/stdc++.h>
using namespace std;

int interpolationSearch(int arr[], int n, int x)
{   // Find indexes of two corners
    int lo = 0;      int hi = (n - 1);    int pos;
    while (lo <= hi && x >= arr[lo] && x <= arr[hi])
    {
        if (lo == hi)
        {   if (arr[lo] == x) return lo;
            return -1;   }
        pos = lo+(((double)(hi-lo)/(arr[hi]-arr[lo]))*(x-arr[lo]));
        if (arr[pos] == x)  return pos;
         // If x is larger, x is in upper part
        if (arr[pos] < x)
            lo = pos + 1;
         // If x is smaller, x is in the lower part
        else
            hi = pos - 1;
    }
    return -1;
}
```

Търсене чрез интерполиране 2/2

Пример 1:

Да се състави програма за търсене чрез интерполиране на дадено число в елементите на даден масив.

!!! Елементите на масива трябва да сортирани или всеки един да е по-голям от предходния.

```
int main()
{
    int arr[] = {10, 12, 13, 16, 18, 19, 20, 21,
                    22, 23, 24, 33, 35, 42, 47};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x ;
    cout<<"Element to be searched= ";
    cin>> x;
    int index = interpolationSearch(arr, n, x);
    // If element was found
    if (index != -1)
        cout << "Element found at index " << index;
    else
        cout << "Element not found.";
    return 0;
}
```

Троично търсене

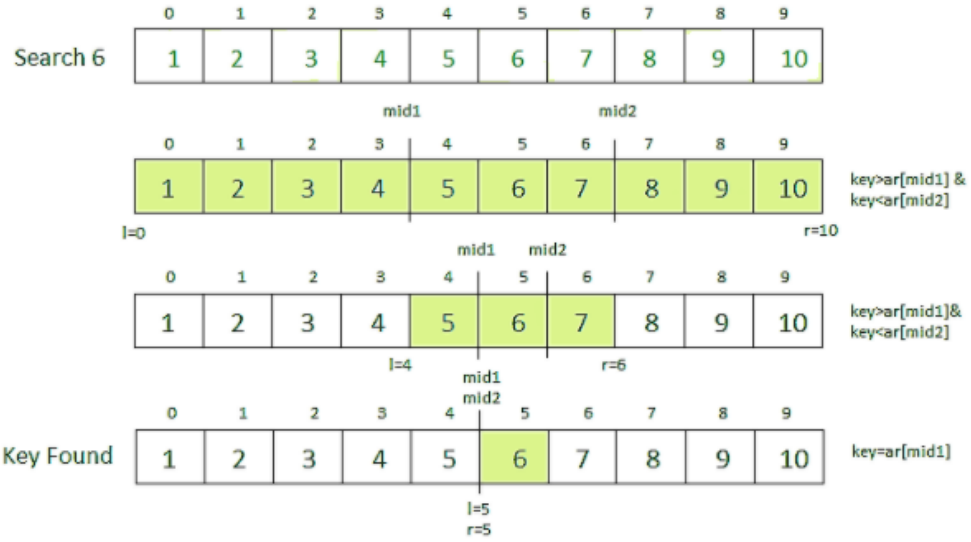
Троичното или тернарното търсене е алгоритъм от типа разделяй и владей. Подобно е на бинарното търсене, където масива се разделя на две части, при този алгоритъм дадения масив се разделя на три части и се определя в коя част е ключа (търсеният елемент). Масивът може да се раздели на три части, като вземем mid1 и mid2, които могат да бъдат изчислени, както е показано по-долу.

mid1 = l + (r-l)/3

mid2 = r – (r-l)/3

Масивът, в който се извършва търсенето **трябва** да е сортиран.

.



Троично търсене

Алгоритъм

1.Сравняваме ключа с елемента с индекс mid1.

Ако има съвпадение, резултатът е mid1.

2. Ако не, тогава сравняваме ключа с елемента с индекс mid2.

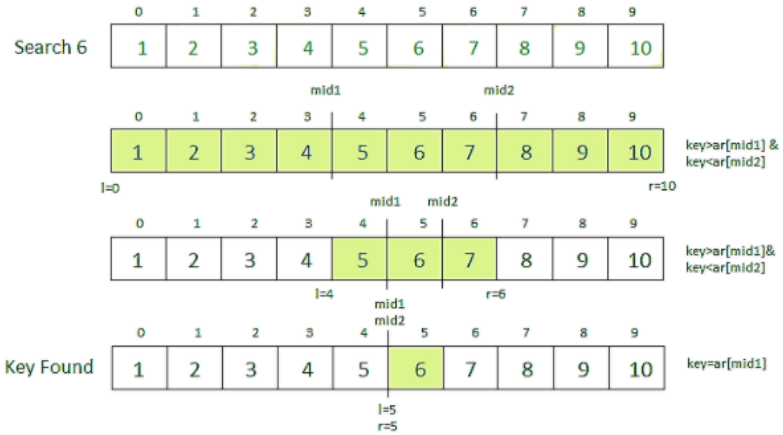
Ако има съвпадение, резултатът е mid2.

3. Ако не, тогава проверяваме дали ключът е по-малък от елемента в mid1. Ако да, тогава ключът се търси в първата част на масива, където: **mid2=mid1-1**, а **mid1=l** и се преминава към стъпка 1.

4. Ако не, тогава проверяваме дали ключът е по-голям от елемента в mid2. Ако да, тогава ключът се търси в третата част на масива, където: **mid1= mid2-1,** а **mid2=r** и се преминава към стъпка 1.

5. Ако не, тогава ключът се търси във втората (средната) част на масива, където mid1=mid1+1, а mid2=mid2-1 и се преминава към стъпка 1.

Търсенето продължава докато двете граници не съвпаднат



Троично търсене 1/2

Пример 1:

Да се състави програма за троично търсене на дадено число в елементите на даден масив.

!!! Елементите на масива трябва да сортирани или всеки един да е по-голям от предходния.

```
#include<bits/stdc++.h>
using namespace std;

int ternarySearch(int l, int r, int key, int ar[])
{   if (r >= l) {
        int mid1 = l + (r - l) / 3;
        int mid2 = r - (r - l) / 3;
        if (ar[mid1] == key) return mid1;
        if (ar[mid2] == key) return mid2;
         if (key < ar[mid1]) {
            return ternarySearch(l, mid1-1, key, ar); }
         else
            if (key > ar[mid2]) {
               return ternarySearch(mid2+1, r, key, ar); }
            else { return ternarySearch(mid1+1, mid2-1, key, ar); }
      }

   // Key not found
   return -1;
}
```

Троично търсене 2/2

Пример 1:

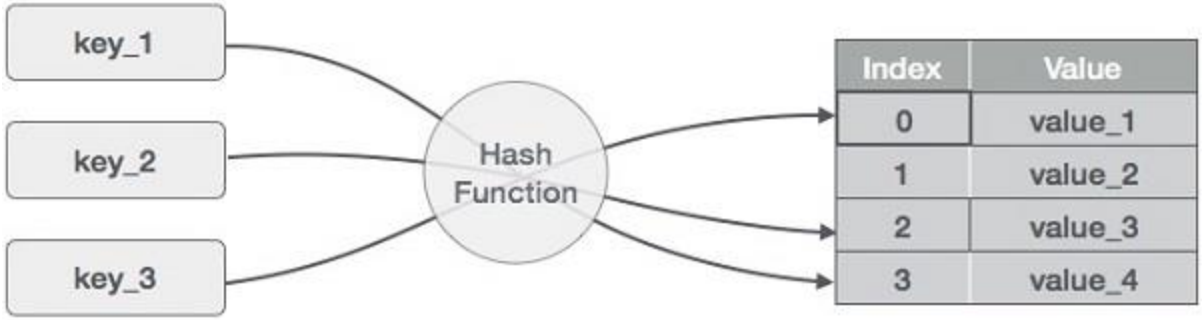
Да се състави програма за троично търсене на дадено число в елементите на даден масив.

!!! Елементите на масива трябва да сортирани или всеки един да е по-голям от предходния.

```
int main()
{
    int l, r, p, key;
    int ar[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    //Left index of array
    l = 0;
    // Right index of array
    r = 9;
    // Key to be searched in the array
    cout<<"Searched key:";
    cin>>key    // key = 5

// Search the key using ternarySearch recursive function
    p = ternarySearch(l, r, key, ar);

if (p==-1)  { cout<<"Index not found";}
    else {cout << "Index of " << key<< " is " << p << endl;}
 return 0;
}
```

Търсене чрез хеш таблица

- Хеш таблицата е структура от данни, която съхранява данни по асоциативен начин. В хеш таблица данните се съхраняват под формата на масив, където всяка стойност на данните има своя уникална стойност на индекса. Достъпът до данни става много бързо, ако знаем индекса на желаните данни.
- По този начин хеш таблицата се превръща в структура от данни, при която операциите по вмъкване и търсене са много бързи, независимо от размера на данните. Хеш таблицата използва масив като носител за съхранение и използва хеш техника, за да генерира индекс, където даден елемент да бъде вмъкнат или да бъде разположен от него.
- Хеширането е техника за преобразуване на диапазон от ключови стойности в диапазон от индекси на масив. За получаване на набор от ключови стойности се използва делене по модул.



Търсене чрез хеш таблица

- Ако имаме хеш таблица с размер 20 то ще ни трябва масив от елементи във формат (ключ, стойност).
- По-долу е да пример за генериране на индексите на нехешираните елементи на масив от 9 елемента (вляво) и получаването на индекса в хеш таблицата.
- Както се вижда, може да се случи техниката на хеширане да се генерира вече използван индекс на масива.

- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Търсене чрез хеш таблица

- В такъв случай можем да търсим следващото празно местоположение в масива, като проверим в следващата му клетка, докато намерим празна клетка. Тази техника се нарича линейно сондиране.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18