DUOMENŲ STRUKTŪROS IR ALGORITMAI

Paieškos algoritmai:

Nuoseklios paieškos, šuoliuojantis, dvejetainis, interpoliacinis

Praeitos paskaitos santrauka

Lėtieji rūšiavimo algoritmai

- Burbulo
- Įterpimo
- Išrinkimo

Spartieji rūšiavimo algoritmai:

Piramidės algoritmas

- Kokie pagrindiniai žingsniai?

Radix algoritmas

- Ar galima juo naudojantis rūšiuoti simbolius, 8-tainius, 16-tainius skaičius?

Kas yra paieška?

Paieška – tai dažniausiai atliekamas veiksmas kompiuteriuose, o ypač dirbant su duomenų bazėmis.

Apibrėžimas

Paieška – tai procedūra, kurios tikslas nustatyti ar elementas su tam tikra reikšme yra aibės elementas.

Paieškos procedūros metu galima ieškoti:

- Elemento su konkrečia tikslia reikšme;
- Elementų, kurių reikšmės patenka į tam tikrą reikšmių intervalą.

Paieškos pagrindinis atliekamas veiksmas – lyginimas.

Paieška

Paieškos rezultatai:

- *Sėkminga paieška* – aibės elementas su ieškoma reikšme egzistuoja t. y. $k = L$
- *Nesėkminga paieška* - aibės elementas su ieškoma reikšme neegzistuoja t. y. $k \; != L$

Paieškos algoritmai skirstomi į tris grupes:

1. Nuoseklios paieškos.
2. Tiesioginės prieigos pagal raktą (hashing).
3. Paieška medžio tipo struktūrose.

Nuosekli paieška

Nuosekli paieška - tai pats lėčiausias algoritmas. Jis tikrina kiekvieną aibės elementą iš eilės ir baigia darbą, kai randamas ieškomas elementas.

Aibėje, kurioje yra **n** elementų, nuosekli paieška blogiausiu atveju atliks **n** lyginimo veiksmų t.y. bus patikrinti visi aibės elementai.

Nuosekli paieška

```
void main(void)
{     int array [10] = { 30, 70, 90, 10, 40, 50, 60, 80, 90, 20 };
      int key;
           cout << “Ivesk ieškomą skaičių(10...100):”;
           cin >> key;
      int flag = 0;

      for (int i = 0; i < 10; i++)
     {    if ( array[i] == key)
           {       flag = 1;
                   break ;
            }      }

      if ( flag )
          cout << “Ieskomas skaičius rastas. Jo indekso numeris“ <<  i << endl;;
      else
           cout << “Ieskomas skaičius nerastas \n“;
 }
```

Paieška surūšiuotose masyvuose

Norint pagreitinti paiešką, reiktų naudoti surūšiuotas aibes.

Jei surūšiuoto masyvo elementas **L** yra didesnis nei ieškomas skaičius **K**, tai reiškia, kad **K** reikšmę turintis masyvo elementas gali būti tik mažesnių už elementą **L** masyvo elementų dalyje.

Šis principas yra ne kas kita, kaip skaldyk ir valdyk metodas.

Pastaba:
Blogiausiu atveju paieškos sudėtingumas yra **O(n)**.

Šuoliuojanti paieška

Turėdami surūšiuotą masyvą galime paiešką pradėti ne nuo 1 elemento, bet nuo tolesnio. Palyginę jį su ieškoma reikšme **K**, galime nuspręsti, į kurią pusę tęsti paiešką – didesnę ar mažesnę.

Šuoliuojančio algoritmo idėja yra ta, kad šuoliais yra skaldomas masyvas į mažesnius ir toks skaldymas tęsiamas tol, kol randamas intervalas, kuriame yra ieškomasis elementas **K**.

Po to pradedama nuosekli paieška rastame duomenų intervale.

Šuoliuojanti paieška

Algoritmas

1. Pasirenkamas dydis *j*, tikriname kiekvieną *j-ajį* elementą t.y. **L[j]**, **L[2*j]** masyve **L**, kol tenkinama sąlyga **L[m* j] < K**.

2. Kai pasiekiama reikšmė **L[m*j] > K**, pradedama nuosekli paieška intervale [(m-1)*j; m*j].

Jei masyvo elementų skaičius **n,** o ieškomo elemento **K** indekso numeris **i** yra intervale **m*j < i < (m+1)*j**, tada paieškos trukmė lygi **m+j-1.**

$$\mathbf{T}(n, j) = m + j - 1 = \left\lfloor \frac{n}{j} \right\rfloor + j - 1.$$

Šuoliuojanti paieška

Koks optimalus šuolio dydis?

Atsakymas: Norint rasti optimalų ***j*** , reikia minimizuoti T(n):

$$\min_{1 \le j \le n} \left\{ \left\lfloor \frac{n}{j} \right\rfloor + j - 1 \right\}$$

Sprendimas:

Randama funkcijos išvestinė ir prilyginama nuliui:

$$f\,'(j) = 0$$

Apskaičiavus gaunama, kad $j = \sqrt{n}.$

Geriausias sudėtingumas T(n) = O($\sqrt{n}.$)

Pavyzdys

```
#define min(x, y) ( (x) < (y) ? (x) : (y) )
int JumpSearch (int a[ ], int n, int  K )
{
        int t = 0;
        int b = (int)sqrt(n);
        while ( a [ min(b, n) - 1] <  K) {
            t = b;
            b = b + (int)sqrt(n);
            if ( t >= n)  return  -1 ;
        }
        while (a[ t ] < K ) {
            t = t+1;
            if ( t == min(b,n) )
                return   -1  ;
            if ( a[t] == K )  {
                return t;
            }
        }
```

```
int main() {
    int  a[7] = {3,7,9,12,14,15,16};
    int K =14;
    int rasti;
    rasti = JumpSearch(a, 7, K);
    if ( rasti >= 0 )
     cout <<“Elementas rastas. Jo
     indekso numeris“ <<  rasti <<  endl;
     else
      cout << “Toks elementas nerastas”;
      return 0;
}
```

Dvejetainė paieška

Dvejetainė paieškos principas - rekursyviai tikrinami surūšiuoto masyvo dalių viduriniai elementai, kiekviename žingsnyje vis mažinant masyvo dalies dydį.

Pirmas žingsnis

- Jei **L[mid] = K**, paieška sustabdoma.
- Jei **L[mid] > K**, paieška tęsiama kairėje masyvo **L** pusėje.
- Jei **L[mid] < K**, paieška tęsiama dešinėje masyvo **L** pusėje.

Dvejetainė paieška

Antras žingsnis

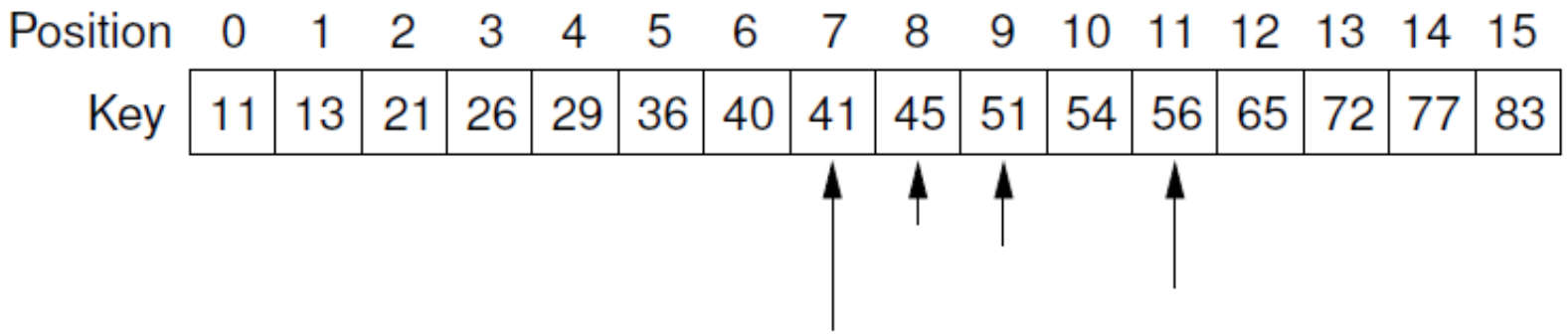
1. Imamas vidurinis elementas iš tos masyvo dalies, kurioje turi būti elementas su reikšme **K**.
2. Išrenkama nauja sumažinto masyvo pusė, kurioje randasi K.
3. Toks procesas kartojamas, kol randama ieškoma reikšmė arba nustatoma, kad nėra elemento su ieškoma reikšme K.

Sudėtingumas:

Blogiausias atvejis: $\mathbf{T(n) = T(n/2) + 1}$

Vidutinis atvejis: $\mathbf{T(n) = \log_2 n}$

Dvejetainė paieškos pavyzdys



Ieškoma reikšmė K = 45.

Pavyzdys

```
int BinarySearch (int a[ ], int n, int K) {
    int low = -1;                    // masyvo dalies pirmo elemento indeksas
    int high = n;                    // masyvo dalies paskutinio elemento indeksas
    while ( low +1 != high) {
        int i = (low + high)/2;      // Randamas viduriniojo elemento indeksas
        if ( K < A[ i ] )
            high = i;
        if ( K == A[ i ] )
            return i;
        if ( K > A[ i ] )
            low = i;
    }
     return -1;
}
```

Interpoliacinė paieška

Jei iš anksto nežinome apie surūšiuotų elementų reikšmių pasiskirstymą, tuomet dvejetainė paieška yra geriausias algoritmas.

Tačiau, kartais mes žinome arba numanome apie reikšmių pasiskirstymą surūšiuotame masyve.

Pavyzdys

Žodyne ieškodami žodžio „*saulė*“ vertimo į anglų kalbą, iš karto praverčiame ~3/4 lapų, nes „*s*“ raidė yra abėcėlės paskutinio ketvirčio raidė. Po to pradedame nuoseklią paiešką. Tokiu būdu sutaupome paieškai skirtą laiką.

Interpoliacinė paieška

Interpoliacinės paieškos metu, ieškomas masyvo elementas su **L[i]**, kuris sutampa su ieškoma reikšme **K.** Elemento pozicija masyve **p** apytiksliai apskaičiuojama taip:

$$p = \frac{K - \mathbf{L}[1]}{\mathbf{L}[n] - \mathbf{L}[1]} * n$$

Kiekviename algoritmo žingsnyje apskaičiuojama sekančio tikrinamo elemento pozicijos numeris pagal aukščiau pateiktą formulę prieš tai pasirenkant judėjimo kryptį, kaip ir dvejetainės paieškos atveju. Priešingai nei dvejetainėje paieškoje, kur visada intervalas dalinamas pusiau, čia jis gali būti didesnis arba mažesnis nei pusė.

Sudėtingumas: blogiausiu atveju **T(n) = O(*n*)**

vidutiniu atveju **T(n) = O(log log*n*)**

Pavyzdys

```
int InterpolationSearch (int a[ ], int n, int K) {
  int low = 0;                          // masyvo dalies pirmo elemento indeksas
  int high = n;                         // masyvo dalies paskutinio elemento indeksas
  int mid;                              //  elemento indeksas, kuris bus tikrinamas
  while ( a[low] <= K && a[high] >= K)  {
     mid = low + ( ( K - a[low] ) * (high - low)) / ( a[high] - a[low]);
     if ( a[mid] < K )
          low = mid + 1;
     else  if ( a[mid] > K)
          high = mid - 1;
   else
        return mid;
  }
 if (a[low] == K)
        return low;
  else
        return -1;
}
```