<u>АЛГОРИТМИ ЗА СОРТИРАНЕ</u>

Сортирането е процес на подреждане на множество елементи по определен ключ или ключове. Сортирането може да се класифицира като *вътрешно*, когато подрежданите елементи се намират в оперативната памет, и като *външно*, когато всички елементи или част от тях се намират във външни файлове.

ВЪТРЕШНО СОРТИРАНЕ

Подрежданите елементи представляват записи с една или повече ключови стойности. Записите са организирани в масив, който се намира в ОП. За улеснение в изложението долу елементите са целочислени стойности, представляващи ключ, по който се извършва подреждането:

A[1], A[2], ..., A[i], A[j], ..., A[k], ..., A[n]

Възможно е при сортирането два елемента да имат еднакви ключови стойности, т. е. да съществуват такива индекси **i** и **k**, за които **A[i]=A[k]** при **i < > k**. Метод за сортирането се нарича *устойчив*, ако той не размества такива **A[i]** и **A[k]**, в противен случай методът е *неустойчив*.

Известни са следните *групи методи* за сортиране:

- Сортиране чрез вмъкване
- Сортиране чрез размяна
- Сортиране чрез избор
- Сортиране чрез сливане
- Сортиране чрез разпределение

Ще разгледаме няколко алгоритъма от изброените групи методи и тяхна реализация. За оценка на ефективност на един алгоритъм за сортиране се използват следните означения: **C** - брой на сравненията на ключовете и **M** - брой на разместванията на елементите. Те се явяват функция на **n** - брой на елементите, които се подреждат. Простите и очевидните методи имат сложност **$O(n^2)$**, по-ефективните – **$O(n * Log(n))$**.

Сортиране чрез вмъкване

Към тази група се отнасят методите: сортиране чрез просто вмъкване, сортиране чрез двоично вмъкване, сортиране по метода на Шел, сортиране чрез вмъкване в списък, сортиране чрез изчисляване на адреса и др.

1. *Сортиране чрез просто вмъкване (Straight insertion sort)*

Идеята на метода е много проста: масивът от подрежданите елементи се дели на две части, първата част съдържа елементите от **A[1]** до **A[i-1]** и вече е подредена, втората част - само елемента **A[i]**, който трябва да се включи към вече подредения масив. Следват две примерни рекурсивни реализации на метода.

Включване чрез полуобмен:

```
void ins_p(int a[], int i)
{ int j; int c=a[i];
  j=i-1;
  while ((j>=0)&&(a[j]>c))    { a[j+1]=a[j--];}
  a[j+1]=c;
  if (i<num-1) ins_p(a, i+1);
}
```

Включване чрез разместване:

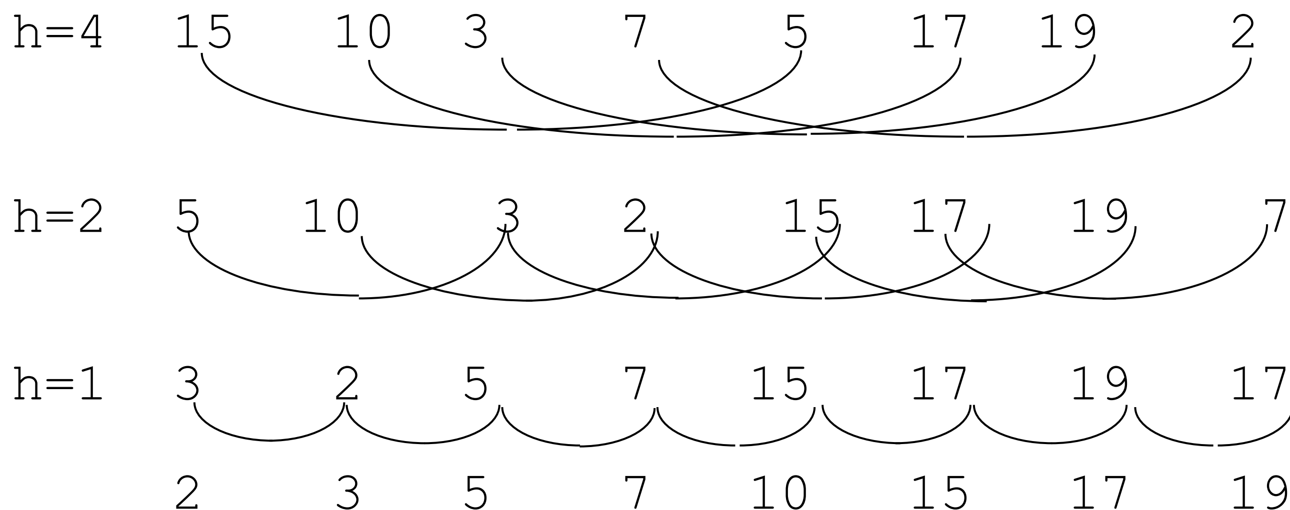
```
void ins_r(int a[], int i)
{
   int rab, j;
   j=i-1;
   while ((j>=0)&&(a[j]>a[j+1]))
    {
        rab=a[j];
        a[j]=a[j+1];
        a[j+1]=rab; j--;
    }
   if (i<num-1)
      ins_r(a, i+1);
}
```

Методът е устойчив. Ефективен е, когато масивът е частично подреден. Препоръчва се при сравнително малки ***n***. Сложността на метода се оценява като ***O(n²)***.

Insertion	Минимален	Среден	Максимален
C (бр. сравнения)	n-1	$(n^2+n-2)/4$	$(n^2+n)/2-1$
M (бр. размени)	0	$(n^2+9n-10)/4$	$(n^2+3n-4)/2$

2. Сортиране по метода на Шел (Shell sort)

Методът е предложен през 1959 г. от Д. Л. Шел и носи името на своя автор. Шел подобрява метода на просто вмъкване чрез групиране на подрежданите елементи, които се намират на разстояние ***h*** един от друг. Всяка такава група се нарича серия и се подрежда отделно. След това стъпката ***h*** се намалява и сортирането се повтаря. Процесът продължава до получаване на $h<1$.



Препоръчва се началната стъпка да има стойност приблизително ***n/9***.
Стъпките образуват следната поредица: $h_{k-1}=3h_k+1$
D.Knuth препоръчва: 1 4 13 40 121 364 1093 3280 9841 и т.н.

Сортиране по метода на Шел:

```
void shellsort(int a[], int left, int right)
       //left - най-малкия индекс на масива
       //right - най-големия индекс на масива
{
   int h;     //стъпка
   for (h=1; h<=(right-left)/9; h=3*h+1)//изчисляване на h
     for(;h>0;h/=3)     //сортиране по серии със стъпка h
        for (int i=left+h; i<=right; i++)
             {
               int j=i; int v=a[i];
               while (j>=left+h && v<a[j-h])
                 {a[j]=a[j-h];j-=h;}
               a[j]=v;
             }
}
```

Методът е неустойчив. Сложността на алгоритъма се оценява като ***O(n * Log(n)).*** Препоръчва се при сравнително големи масиви.

Сортиране чрез размяна (*Exchange sort)*

Към тази група се отнасят: метод на мехурчето, усъвършенстван метод на мехурчето, бързата сортировка, обменна поразредена сортировка и др.

1. Сортиране чрез пряка размяна или метод на мехурчето (Bubble sort).

Идеята на метода се състои в следното: сравняват се и при необходимост се разместват всеки два съседни елемента. Названието на метода идва от аналогията с въздушни мехурчета - при последователни прегледждания на масива с данни "по-леките" (с по-малък ключ) елементи постепенно "изплуват" нагоре - в началото на масива.

```
void Bubblesort1(int a[], int n)
{   int r, i, j;
    for (i=0; i<n-1; i++)
    for (j=n-1; j>=i; j--)
       if (a[j-1]>a[j])
        {r=a[j];   a[j]=a[j-1];   a[j-1]=r; }
 }
```

Сортиране по метода на мехурчето

```
void Bubblesort2(int a[], int i)
{ int inv, r;
  do
  { inv=0; //флаг за инверсия или необходимост от размяна
    for (int j=0; j<i-1; j++)
    if (a[j]>a[j+1])
        {
          inv=1;
          r=a[j];
          a[j]=a[j+1];
          a[j+1]=r;
        }
  }
  while (inv);
}
```

Методът има сложност ***O(n²) .*** Препоръчва се при сравнително малки ***n***. Методът е подобрен чрез въвеждането на допълнителна логическа променлива, която следи за наличието на инверсии, изискващи размяна на съседните елементи в масива. Ако при поредния преглед не е открита нито една инверсия, алгоритмът завършва своята работа.

Основните характеристики на сортирането по метода на мехурчето са:

Bubblesort	Минимален	Среден	Максимален
C (бр. сравнения)	n(n-1)/2	n(n-1)/2	n(n-1)/2
M (бр. размени)	0	$3(n^2-n)/4$	$3(n^2-n)/2$

Методът може да бъде подобрен още веднъж по следния начин: безсмислено е да преглеждаме елементите, за които със сигурност се знае, че са на окончателните си места. Сортирането се извършва в обратна посока: започваме от "дъното" в края на масива и най-тежките елементи потъват към него. В променливата ***k*** ще съхраняваме максималния индекс, при който е била извършена размяна на предходната итерация. Елементите, разположени вдясно от последната размяна, са на окончателните си места.

```
void Bubblesort3(int a[], int n)
{int r, i, j, k;
 for (i=n; i>0; i=k)
 for (k=j=0; j<i; j++)
       if (a[j]>a[j+1])
        {r=a[j]; a[j]=a[j+1]; a[j+1]=r;
         k=j;}
}
```

2. Сортиране чрез клатене

Въпреки ниската си ефективност метода на мехурчето дава възможност за редица подобрения. Вече разгледахме две от тях (чрез добавяне на флаг и поддържане на променлива, указваща максималния индекс на размяна от предходната стъпка).

Методът има една особеност: отделен лек елемент в края на масива ще изплува нагоре за една-единствена итерация, докато един тежък елемент, разположен близо до повърхността, ще потъва надолу само с една позиция при всяка стъпка на алгоритъма. Ако обърнем посоката на циклите (както сме направили във втория алгоритъм) ситуацията ще се обърне, но асиметрията ще се запази. Идеята на предлагания подход е посоката на сортиране да се обръща на всяка стъпка. Това е трето подобрение на алгоритъма. Прилагането на трите подобрения едновременно дава нов алгоритъм - *сортиране чрез клатене*.

```
void shakersort (int a[], int n)
{unsigned k=n, R=n-1;
 unsigned L=1, j, rab;
 do{
    for (j=R; j>=L; j--)
      if (a[j-1]>a[j])
       {rab=a[j]; a[j]=a[j-1]; a[j-1]=rab; k=j;}
    L=k+1;
    for (j=L; j<=R; j++)
       if (a[j-1]>a[j])
        {rab=a[j]; a[j]=a[j-1]; a[j-1]=rab;k=j;}
    R=k-1;
   }
 while (L<=R);
}
```

В най-добрия случай горния алгоритъм извършва ***n-1*** сравнения, а в средния случай - n^2. Макар и има известно подобрение в сравнение с класическия метод на мехурчето, общата ефективност на алгоритъма не се подобрява съществено, т. к. сортирането чрез клатене не намалява броя на извършваните размени. А размяната по-принцип е неколкократно по-тежка операция от сравнението.

Основният недостатък на метода е в "микроскопичния" му подход - винаги се сравняват два съседни елемента. Този недостатък е отстранен в следващия разгледан метод.

3. Сортиране чрез сегментиране или бързо сортиране, или сортиране на Хоар (Quicksort)

Целта на предстоящото подобрение на основната идея е да се минимизира броя на размените в масива. От последното подобрение (или сортиране чрез клатене) стана ясно, че той ще бъде редуциран, ако разменяме първия елемент с последния, втория - с предпоследния и т. н., сортирайки целия масив за **[*n*/2]** размени. Изводът: колкото на по-големи разстояния се извършват размените, толкова е по-голяма тяхната ефективност.

Идеята на алгоритъма, предложен от Хоар, е да изберем някакъв елемент ***x*** и да разделим масива на две части: лява, в която елементите са по-малки от ***x***, и дясна, в която са по-големи. Положението на ***x*** (т. е. индекса му в масива) след преобразуването се фиксира и повече не се променя. Прилагаме същия алгоритъм за лявата и дясната части, стеснявайки постепенно границите на разглежданите подмасиви, докато не достигнем до интервали (сегменти), съдържащи единствен елемент (оттук и названието - сортиране чрез сегментиране). След приключване на работата на алгоритъма масивът е подреден.

Да означим с ***q*** индекса на ***x*** в масива, т.е ***x*** =***a[q]***.

a[i]<=x	***x***	***a[i]>=x***

0 q-1 q q+1 n-1

Нека сортираме подмасива ***a [L, L+1,…, R]*** и нека *partition()* разделя подмасива на две части, извършва в него съответните преобразования и връща като резултат ***q*** (но не търси в масива ***x***, а го определя по определени правила! Как?).

Най-общо бързото сортиране може да се опише така:

```
void quicksort (int L, int R)
{ int q;
   if (L < R)
        {  q=partition(L, R);
             quicksort(L, q);
             quicksort(q+1, R);
         }
}
```

Как да извършим разделянето на масива?
Има два различни подхода или два варианта на *partition()* .
<u>Вариант 1:</u> избира се елемент ***x***, по който се извършва разделянето. В лявата част на масива трябва да останат елементи, по-малки или равни на ***x***, а в дясната - строго по-големи от ***x*** . Текущия дял от масива ***a[L, L+1, …, R]*** се разглежда отляво надясно, като при това в лявата му част се изгражда постоянно разширяваща се област от елементи, по-малки или равни на ***x*** . Десният край на областта се определя от ***q***. Когато ***j*** достигне края ***R*** на дяла, ***q*** ще сочи границата между двете области.

```
unsigned partition1  (int L, int R)
{ int q, j, x, rab;
    q=L-1; x=a[R];
    for  (j=L; j<=R, j++)
         if (a[j]<=x)
             { rab=a[q]; a[q]=a[j]; a[j]=rab;
                 q++;
             }
    if (j==R)     q--;
   return q;
}
```

Вариант 2: използват се два индекса - ***j*** и ***i*** , показващи границите на две непрекъснато разширяващи се области от двата края към центъра. На всяка стъпка на алгоритъма се прави опит за разширяване на лявата област надясно, докато това е възможно, т. е. докато вдясно от нея стои елемент, по-малък или равен на ***x*** . Същото се извършва и за дясната област (двата *while*-цикъла). След това се разменят местата на двата елемента-прегради за лявата и дясната области, които са спрели разширяването им, и процесът се повтаря отначало. Приключва при "срещане" на двете области, т. е. когато двете им граници се разминат. Индексът ***i*** сочи десния край на лявата област, а ***j*** - левия край на дясната:

```
unsigned partition2 (int L, int R)
{ int i, j, x, rab;
   i=L; j=R; x=a[L];
   do
      {  while (x>a[i])     i++;
          while (x<a[j])     j--;
          if (i<=j)
             {rab=a[i]; a[i]=a[j]; a[j]=rab; i++; j--; }
      }
   while (i <=j);
   return j;
}
```

За сортиране на двете части на масива се извършват две последователни рекурсивни обръщения. Т. к. второто винаги се отлага (двете обръщения не могат да се извършат едновременно!), то може да се реализира итеративно - чрез цикъл.

Възможно е алгоритъмът изцяло да бъде реализиран итеративно - чрез използване на специално създаден за целта стек. На всяка стъпка лявата част на масива ще се обработва директно, докато дясната част ще се поставя в стека, като обработката й ще се отлага до момента, когато се приключи окончателно с лявата част и всички свързани с нея подзадачи.

Остава открит въпроса: как да се избира средния елемент - ***x***? Най-добре на всяка стъпка (при всяко сегментиране) да се избира *средния по големина* елемент. Тогава общия брой сравнения ще бъде $n*\log_2 n$. Очакваният брой размени при всяко разделяне е от порядъка на $n/6$. Оттук броят на размените в оптималния случай е:

$$(n/6)*\ n*\log_2 n$$

Вероятността да улучим средния елемент е 1/***n***, но средната ефективност на бързото сортиране не зависи само от ***n***.

Все пак, как да избираме елемента ***x***, относно който разделяме масива? Един възможен подход е да вземаме винаги фиксиран елемент от масива, например първия, последния или средния (с индекс [***n***/2]). Изборът на средния елемент е изключително важен за за скоростта на алгоритъма: колкото по-близък е избрания елемент до средния *по големина* елемент за дяла, толкова по-равномерно става разбиването на масива на две части, а оттук - по-малка дълбочина на рекурсията, т. е. по-ефективна работа на алгоритъма. Можем ли да избираме за разделящ елемент медианата на два или повече елементи? Да, но това е свързано с допълнителни операции и води до забавяне, което не винаги се компенсира с извлечените ползи и не води задължително до подобрение.

Бързото сортиране е най-бързия *универсален* алгоритъм за сортиране. Средната му сложност е $\mathbf{O}(n*\log_2 n)$. Но той крие и редица изненади. Сортирането на Хоар е доста непостоянно и, при неподходящ избор на средния елемент ***x***, ефективността му пада до $\mathbf{O}(n^2)$.

Съществуват и други фактори, влияещи на ефективността на алгоритъма:

- силата на бързото сортиране е в неговата “хазартност”. Колкото по-разбъркан е масивът, толкова по-голяма е ефективността на метода. Авторът му Хоар препоръчва изборът на ***x*** да се прави по случаен начин или дори като средно аритметично на три или повече случайно избрани елемента;

- алгоритъмът е ефективен само при големи ***n***. Затова при получаване на сегмент с големина <=15 елемента се препоръчва да се премине към някой друг алгоритъм и сортирането да се довърши чрез използване на някой друг алгоритъм. Подмасивът е почти сортиран и е добър вход за сортирането чрез вмъкване: вмъкванията ще се извършват в рамките на съответната област, т. е. гарантирано на малки разстояния.

Библиотечната функция *qsort*

Езикът C/ C++ има библиотечна функция *qsort*, която може да сортира произволен масив:

```
# include <stdlib.h>
      …
 qsort(a, n, sizeof(T), compare);
```

където *a* е подреждания масив, *n* е броят на елементи в него, *T* е типа на елементите в масива, *compare* е наша собствена функция, показваща как трябва да бъдат сравнявани два елемента *a[i]* и *a[j]* от масива. Тази функция е подобна на добре известната функция *strcmp*, която връща 0 при равенство, някакво отрицателно число при "по-малко" и някакво положително число при "по-голямо".

__Пример:__ сортиране на масив от цели числа с библиотечната функция *qsort*

```
#include <iostream>
#include <stdlib.h>
using namespace std;
const int n=100;
int a[n], num;
void input(int a[])
{
 cout<<"\nInput num<100:";
 cin>>num; cout<<"\n"<<num;
 cout<<"\nInput items:\n";
 for (int i=0; i<num; i++)
 {      cout<<"=";
        cin>>a[i];
 }
}
```

```
void out(int a[])
{
  cout<<"\n Array:\n";
  for (int i=0;i<num; i++)
   cout<<a[i]<<" ";
}
int  compare (const void *p, const void *q)
{
 return *(int*)p < *(int*)q ? -1: *(int*)p > *(int*)q;
}
void main()
{
  input(a);
  out(a);
  qsort(a, num, sizeof(int), compare);
  out(a);

}
```

Сортиране чрез избор (извличане) (Selection sort)

Към тази група се отнасят: сортиране чрез прост избор, избор от дърво, пирамидална сортировка и др.

1. Сортиране чрез прост избор (пряка селекция)

Идеята на метода е следната: търси се елемент с най-малкия (или най-големия) ключ; намереният елемент се премества в началото (или в края) на масива и се изключва от разглеждането; действието се повтаря, докато не се изберат всички ***n*** елемента. Методът е аналогичен на метода на простото вмъкване - и в двата случая в левия край на масива се образува поредица от вече подредените елементи.

Сложността на метода е ***O(n²)***. Както и при метода на прекия избор, масивът се разделя на сортирана и несортирана част, като на всяка стъпка на алгоритъма сортираната област се разширява отдясно с един елемент. На първата стъпка се намира минималния елемент на масива и се разменя с първия. На втората минималния измежду останалите елементи се разменя с втория елемент на масива и т. н. На всяка следваща стъпка минималния елемент от несортираната част на масива се разменя мястото с първия елемент от несортираната част, разширявайки по този начин сортираната част, докато тя не обхване целия масив:

```
void straight_selection (int a[], unsigned n)
{ unsigned  i, j;
   int rab;
   for (i=0; i<n-1; i++)
       for (j=i+1; j<=n; j++)
           if (a[i] > a[j])
                {rab=a[i]; a[i]=a[j]; a[j]=rab;}
 }
```

Широко разпространен е и леко модифицирания вариант на разгледаната реализация: във вътрешния цикъл е спестено адресирането на ***j***-ия елемент като ***a[j]*** и за целта е въведена променливата ***x:***

```
void straight_selection (int a[], unsigned n)
{ unsigned  i, j, ind;
  int  x;
  for (i=0; i<n-1; i++)
     for (x=a[ind=i], j=i+1; j<=n; j++)
        if (a[j] < x)
           { x=a[ind=j];
             a[ind]=x;
             a[i]=x;
           }
}
```

Подобно на сортирането по метода на мехурчето, алгоритъмът на пряката селекция винаги извършва ***n****(***n***-1)/2 сравнения. Но той е за предпочитане пред метода на мехурчето по броя на извършваните размени:

Straight Selection	Минимален	Среден	Максимален
C (бр. сравнения)	n(n-1)/2	n(n-1)/2	n(n-1)/2
M (бр. размени)	0	n ln n	$[n^2/4] + 3(n-1)$

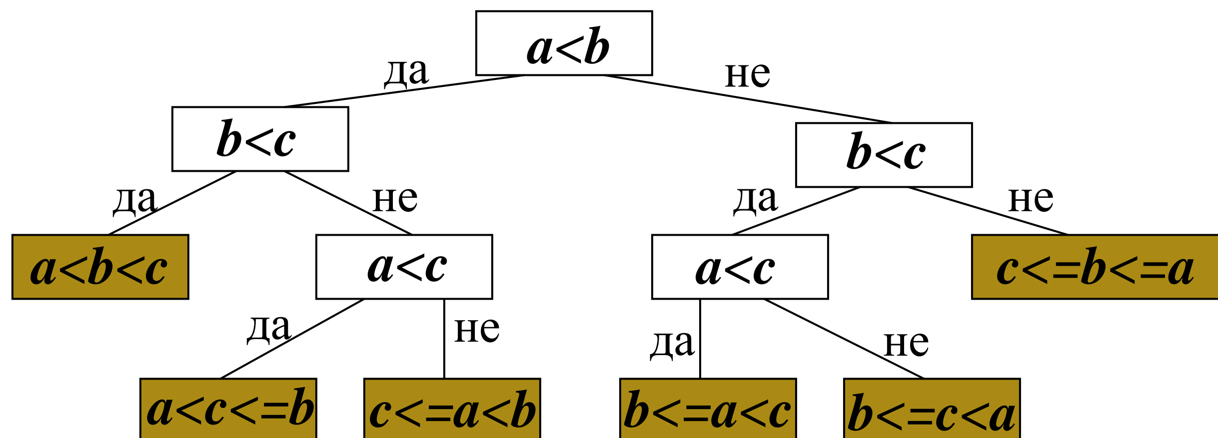
2. Пирамидално сортиране (Heapsort)

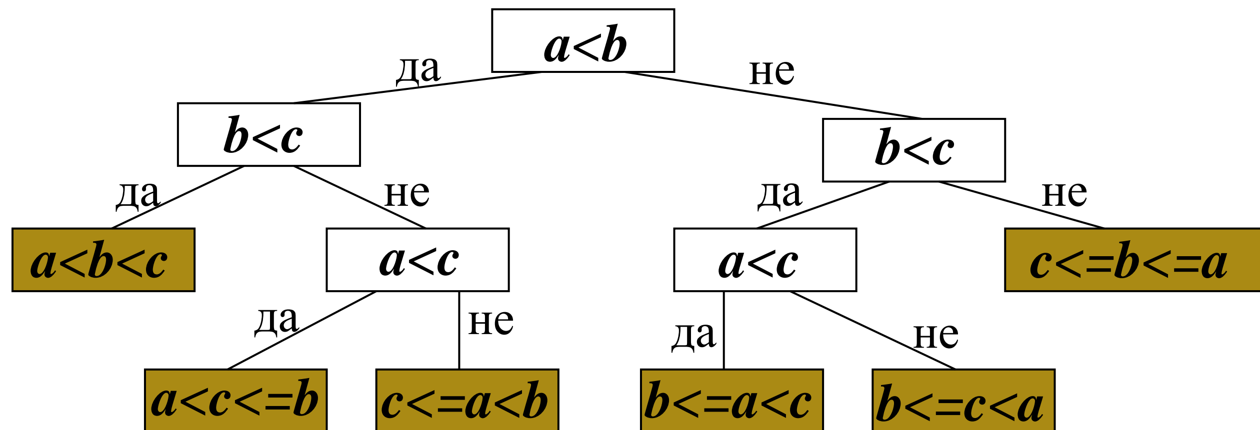
Сортирането чрез пряк избор се нарежда сред най-неефективните, подобно на метода на мехурчето и метода на прякото вмъкване. Основен негов недостатък е, че на ***i***-та стъпка винаги се извършват точно ***n-i*** сравнения, независимо от входните данни.

Методът се базира на последователното избиране на най-малкия ключ измежду ***n*** елемента, за което са необходими ***n-1*** сравнения. След това на втората стъпка на алгоритъма най-малкия ключ измежду ***n-1*** елемента се избира с ***n-2*** сравнения и т.н. Бихме могли да направим извода, че на ***i***-та ще ни потрябват ***n-i*** сравнения. Но това не е съвсем вярно, понеже в резултат на прилагането на първата стъпка получаваме не само най-малкия елемент, но и редица съотношения между двойки от останалите елементи, които можем да съхраним и да използваме на втората стъпка. Ясно е, че колкото повече допълнителни съотношения получим, толкова по-малко сравнения ще ни бъдат необходими на следващите стъпки, т. е. колкото е по-ниско и по-разклонено *дървото на сравненията*, толкова повече информация ще ни носи то.

За да се поясни последния извод, нека да разгледаме ново понятие - *двоично дърво на сравненията*.

Например, искаме да сравним три елемента - ***a, b*** и ***c***. Това е най-елементарния метод за сортиране - чрез сравнение. Основава се на поредица от сравнения, всяко от които ни носи допълнителна информация. Процесът продължава до пълното сортиране на множеството **{*a, b. c*}*.*** В зависимост от резултата от сравнението получаваме различни съотношения между елементите и оттук - различно продължение на процеса. Всяко сравнение има два изхода - "да" и "не", което ни дава възможност процесът на сравнение да се изобрази като двоично дърво, в чиито листа се намират сортираните последователности, а в останалите възли - сравнения между двойки елементи от множеството:





Дървото ни дава следния алгоритъм за сортиране на множеството:

```
if (a<b)
    if (b<c)   cout<<a<<b<<c;
        else
           if (a<c)   cout<<a<<c<<b;
               else    cout<<c<<a<<b;
else
     if (b<c)
         if (a<c) cout<<b<<a<<c;
              else   cout<<b<<c<<a;
     else    cout<<c<<b<<a;
```

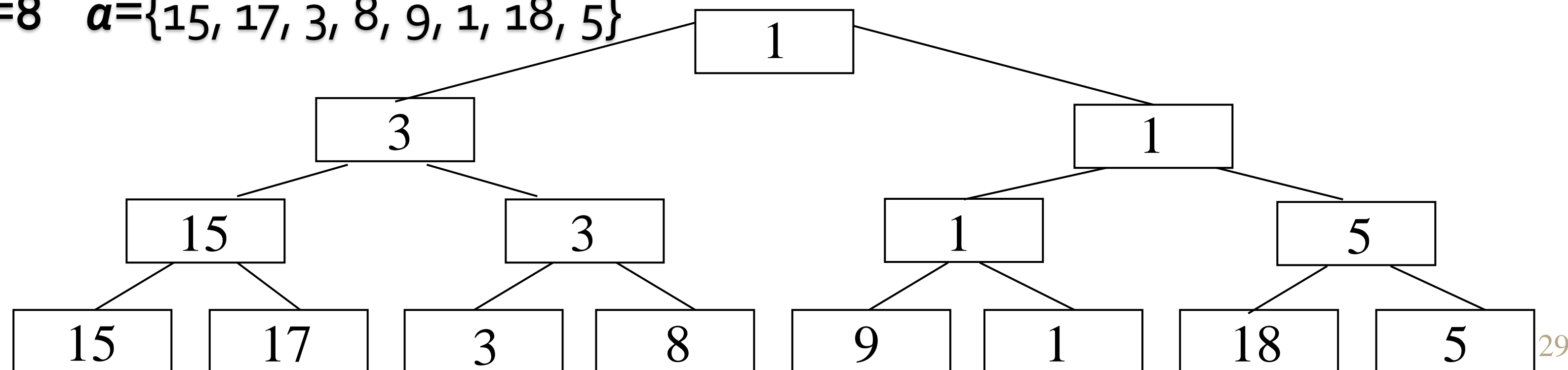
Алгоритъмът е много нагледен, но е почти неприложим: броят на възможните изходи (листата на дървото) е ***n!*** (колкото възможните пермутации на изходното множество).

Затова се налага да се търсят други по-ефективни методи за сортиране. Но идеята на дървото може да послужи като математически модел за изграждане на други алгоритми.

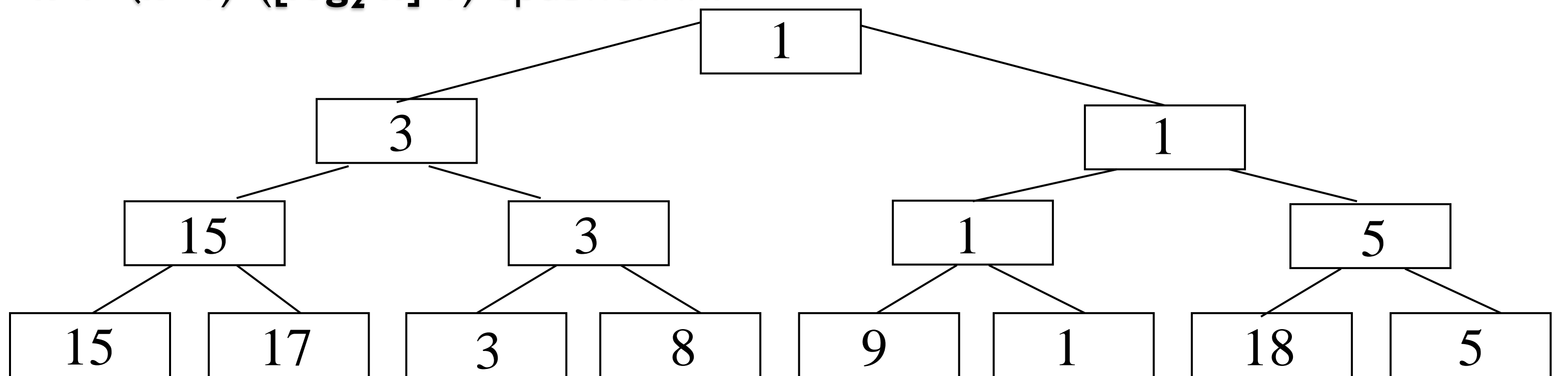
Можем да изградим дърво на сравненията, използвайки механизма на *турнира с елиминиране*, построяващ дървото на сравненията от листата към корена. В първия кръг на турнира се играят срещите: $x_1{:}x_2, x_3{:}x_4, \ldots, x_{n-1}{:}x_n$. Във втория кръг играят двойки победители от предишния кръг и т. н. На финала се срещат двама участници и определят шампиона на турнира . В случай, че в ***i-***ия кръг на турнира броят на участниците е нечетен (а това ще се случи поне веднъж, ако ***n*** не е степен на 2), то един от играчите почива, като автоматично се класира за следващия кръг. Използвайки посочената стратегия за определяне на минималния елемент на масива получаваме следното: в първия кръг на турнира с ***n/2*** сравнения се определя минималния елемент за всяка двойка, с още ***n/4*** сравнения се определя минималния елемент между две двойки (т. е. 4 елемента) и т. н.

Изложеният алгоритъм построява дървото на турнира, т. е. определя минималния елемент на масива с точно ***n-1*** сравнения:

n=8 ***a***={15, 17, 3, 8, 9, 1, 18, 5}



Сега, заменяйки най-малкия елемент ("шампиона") с ∞ в съответното му листо и преизчислявайки съдържанието на върховете по пътя до корена, във върха на дървото ще получим следващия по големина елемент. Т. к. дървото има височина [$\log_2 n$], то описаният процес на определяне на втория по големина елемент ("вицешампиона") изисква [$\log_2 n$]-1 сравнения вместо n-2, както изисква алгоритъма на пряката селекция. Така процесът на цялостно сортиране на масива изисква не повече от $n-1+(n+1)*([\log_2 n]-1)$ сравнения:



Изложеният алгоритъм е сериозно подобрение на алгоритъма на пряката селекция, но и той има редица недостатъци: все още голям брой на извършваните размени; стойностите ∞, които водят до излишни сравнения, като накрая запълват цялото дърво; изгражданото дърво, макар и идеално балансирано, не дава оптимални резултати, т. к. позволява листата да се появяват на всички нива: участниците в турнира са разположени в листата, а останалите върхове на дървото само ги дублират.

Естествено желание е да се минимизира размера на необходимата памет, която при алгоритъма на турнира с елиминиране е от порядъка на $2n-1$. За целта ще се търси ефективен механизъм за линеаризация на дървото, който позволява да се пазят само листата, без да се губи известните съотношения между тях.

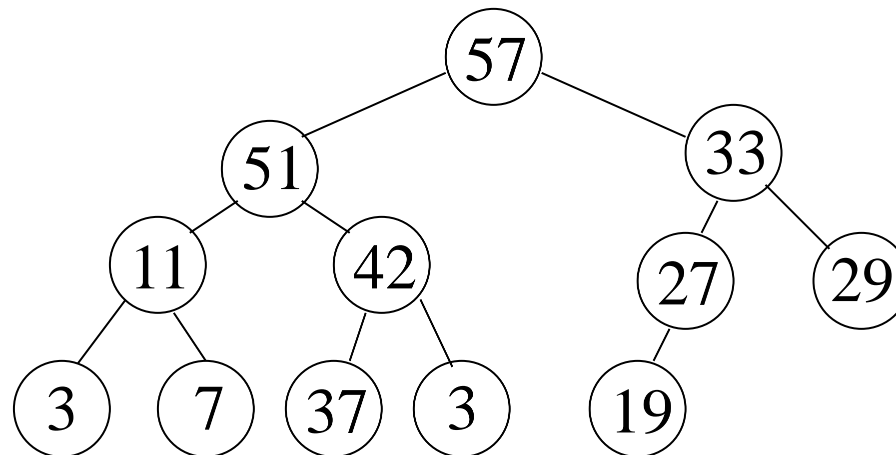
Такъв механизъм е предложен от Дж. Уилямс и се нарича *пирамида*.

Пирамидата е балансирано двоично дърво с височина h, едновременно отговаряща на следните условия:

1) всички листа са на ниво h и $h-1$;

2) всички наследници на даден връх са по-малки от него;

3) всички наследници от ниво h са максимално изместени вляво.



Пирамидата може да бъде ефективно реализирана не само чрез дърво, но и чрез n-елементен масив, в който наследствените връзки между елементите се определят като проста линейна функция на позициите им.

Как точно?

Пирамидата е еквивалентна на редица от стойности (или ключове) $a_{lf}, a_{lf+1}, \ldots, a_{rt}$ (1<=*lf*<=*rt*<=n) $a_i>=a_{2i}$ и $a_i>=a_{2i+1}$, $i=lf, lf+1, \ldots, rt/2$.

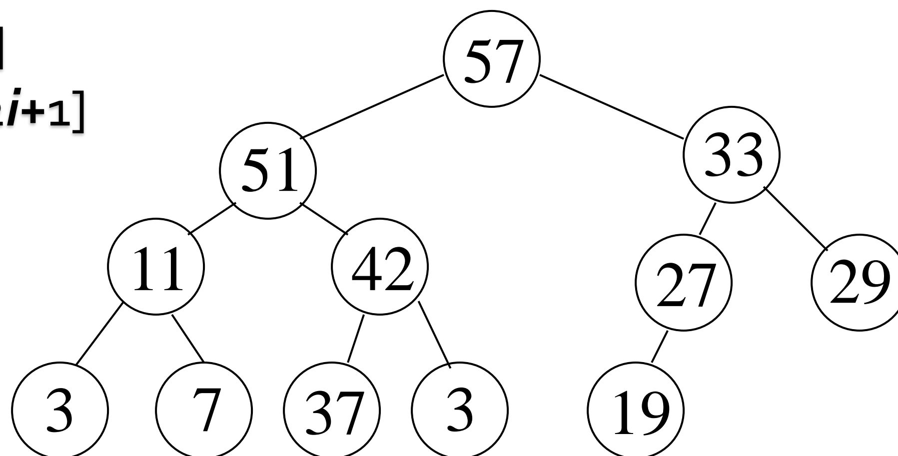
Т.е. представянето е аналогично на статичното представяне на двоично дърво:

a[1]	***a***[2]	***a***[3]	***a***[4]	***a***[5]	***a***[6]	***a***[7]	***a***[8]	***a***[9]	***a***[10]	***a***[11]	***a***[12]
57	51	33	11	42	27	29	3	7	37	3	19

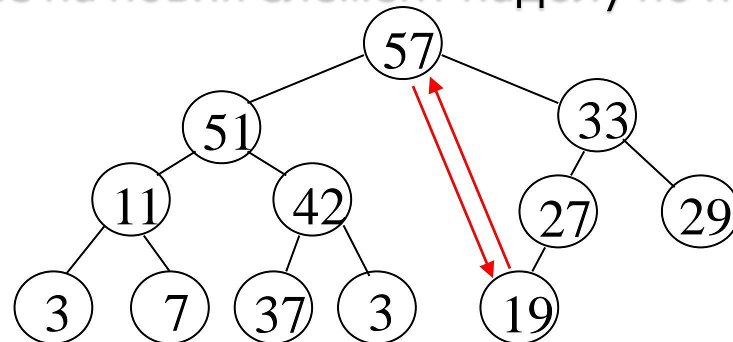
Левият наследник на ***a***[***i***] е ***a***[2***i***]

Десният наследник на ***a***[***i***] е ***a***[2***i***+1]

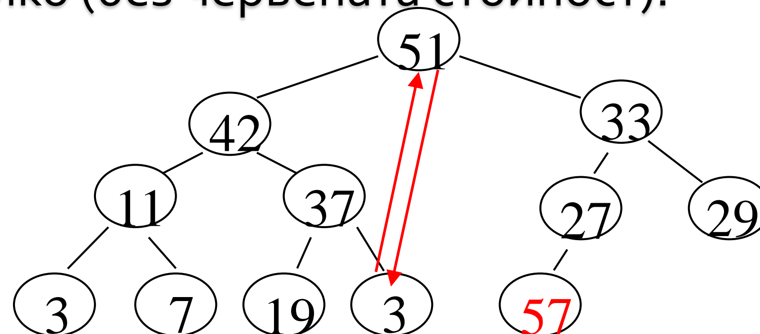
Върхът на пирамидата съдържа най-големия елемент.



Нека да предположим, че разполагаме с ефективен алгоритъм за изграждане на пирамида, реализиран с *bild_heap()*, и с алгоритъм за възстановяване на пирамида от ***k*** елемента *restore_heap()*, след като върхът й е бил заменен с произволен елемент. Тогава, след като сме изградили пирамида с ***n*** елемента, можем да разменим върха й ***a***[1] (най-големия й елемент) с последния й елемент ***a***[***n***], при което стария връх заема окончателната си позиция - става последен в сортирания масив. Т. к. тази операция нарушава формулираните за пирамидата правила, то се налага отсяване на новия елемент надолу по пирамидата:



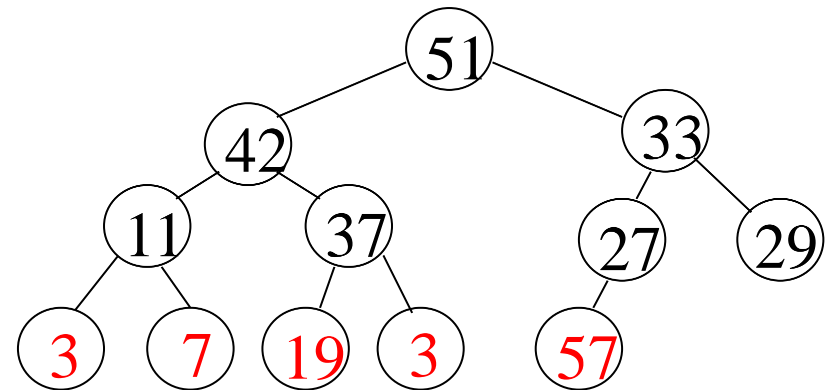
В резултат получаваме нова пирамида a1, a2, ..., an-1, съдържаща един елемент по-малко (без червената стойност):



Следва нова размяна на върха на пирамидата с последния (***n***-1) елемент, последвана от ново отсяване. Описаният процес се повтаря (***n***-1) пъти (няма нужда от ***n***-та стъпка, защото накрая ***n***-ия елемент си е на място).

Алгоритъмът за пирамидално сортиране е:

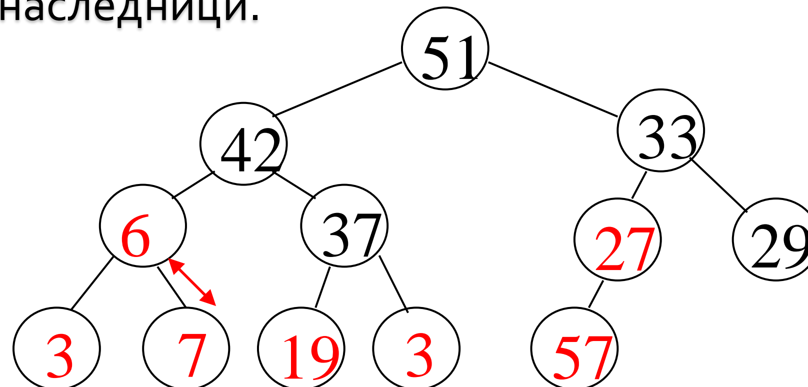
```
...
build_heap()
for (i=n; i>=2; i--)
{
    rab=a[1]; a[1]=a[i]; a[i]=rab;
    restore_heap(i-1);
 }
...
```



Как да изградим пирамидата?

Нека ***n*** да е четно. Тогава за елементите ***a***[***n***/2+1], ***a***[***n***/2+2], …, ***a***[***n***] не се изисква никакво съотношение за наредбата, т. к. за никои два елемента i и j не е изпълнено j=2i и j=2i+1. Те са листата и за тях не важат зададените правила.

Нека да разширим пирамидата отляво с един елемент. Новодобавеният елемент ***x*** ще бъде от предпоследното ниво и ще има поне един наследник. Трябва да се проверят съотношенията и при необходимост ***x*** да се размени с един от своите наследници.



Сега елементите ***a***[***n***/2], ***a***[***n***/2+1], ***a***[***n***/2+2], …, ***a***[***n***] пак образуват пирамида. Следва ново разширяване на пирамидата отляво, като процесът продължава до включване на всички елементи. При нечетно ***n*** листата са ***a***[***n***/2+1], ***a***[***n***/2+2], …, ***a***[***n***]. (В реализацията на C/C++ няма нужда от това разграничение, т. к. операцията / е целочислена.)

Отсяването на новодобавения елемент ***x*** надолу по пирамидата може да протича на няколко етапа, т. е. след размяна на ***x*** с някой от наследниците може да се окаже, че правилата не са изпълнени. Така ***x*** се отсява надолу по пирамидата, като на всяка стъпка потъва с едно ниво надолу, докато не заеме мястото си.

Тук може да се направи едно очевидно подобрение, ускоряващо процеса, а именно - ***x*** да се разменя с по-малкия от двата наследника. Освен това размяната на ***x*** с наследниците му може да се отлага дотогава, докато не бъде намерено точното му място, като вместо това в процеса на търсене на крайната му позиция се извършват еднопосочни присвоявания.

Както построяването, така и възстановяването изискват отсяване на върха надолу по пирамидата. Това означава, че може да се реализира обща функция за отсяване *sift()*:

```
void sift (int a[], unsigned lf, unsigned rt)
{ unsigned i=lf, j=i+i;
  int x=a[i];
  while (j<=rt)
     { if (j<rt)
          if (a[j]<a[j+1])    j++;
        if (x>a[j])  break;
        a[i]=a[j];
        i=j;
        j<<=1;// умножение по 2
     }
   a[i]=x;
 }
```

Построяването на пирамидата (функция build_heap()) може да се извърши така:

```
for (k=n/2+1; k>1; k--)
   sift(a, k-1, n);
```

За сортирането - многократна размяна на върха на пирамидата с последния й елемент, последвана от отсяване (функция restore_heap()) получаваме:

```
for (k=n; k<1; k--)
 { rab=a[1]; a[1]=a[k]; a[k]=rab;
    sift(a, 1, k-1);
  }
```

Окончателният вариант на пирамидалното сортиране е:

```
void heapsort(int a, unsigned n)
{  unsigned k;
    for (k=n/2+1; k>1; k--)    // построяване на пирамидата
         sift(a, k-1, n);
    for (k=n; k<1; k--)        //построяване на сортирана последователност
      { rab=a[1]; a[1]=a[k]; a[k]=rab;
         sift(a, 1, k-1);
       }
}
```

Сортиране чрез сливане *(Mergesort)*

В основата на тази група алгоритми за сортиране стои вече разгледания метод "разделяй и владей": подрежданата поредица се разделя на две части, всяка от които се сортира, след което подредените подмасиви се обединяват (или се сливат).

1. Сливане на сортирани масиви

Нека да видим, как можем да обединим два вече подредени по големина масива ***a***[ ] и ***b***[ ].

```
...
const int an=100, bn=120;
int a[an], b[bn], c[an+bn];
...
```

Функция, реализираща тази операция:

```
void merge (int a[], int na, int b[], int nb, int c[])
{  int ia=0, ib=0, ic=0;
   while (ia<na&&ib<nb)
      c[ic++]=((a[ia]<b[ib])?a[ia++]:b[ib++]);
   while (ia<na) c[ic++]=a[ia++];
   while (ib<nb) c[ic++]=b[ib++];
}
```

Функцията има един съществен недостатък: не сортира на място, а изисква допълнителна памет, колкото за двата сливани масива.

2. Сортиране чрез сливане

Разгледаната функция може да служи за основа на следния прост алгоритъм за сортиране чрез сливане:

```
void mergesort (int a[], int n)
//функцията разделя масива рекурсивно на две части
//след което функцията merge ги обединява
{
    if(n<2) return;
    int nleft=n/2, nright=n-nleft;
    mergesort(a, nleft);
    mergesort(a+nleft, nright);
    int *p=new int [n];
    merge (a, nleft, a+nleft, nright, p);
    for (int i=0; i<n; i++)
       a[i]=p[i];
    delete []p;
}
```

Т. к. между операциите *new* и *delete* няма рекурсивно обръщение, временно необходимото място в паметта не надминава *n*size (T)* байта (в нашия случай ***T*** е ***int*** и необходимия обем памет ще бъде не повече от ***4n***).

Алгоритъмът за сортиране чрез сливане е един от най-ефективните известни алгоритми за сортиране. Гарантираната му средна времева сложност е $O(n*\log_2 n)$, което теоретически е най-добрата алгоритмична сложност за универсален алгоритъм за сортиране.

Друго основно предимство на сортирането чрез сливане е, че достъпът до елементите се извършва строго последователно, поради което е подходящо за сортиране на списъци, последователни файлове и др.

Още едно положително качество на алгоритъма - сортирането чрез сливане е устойчиво, т. е. запазва относителната наредба на елементите с еднакви ключове. (Например, Quicksort е неустойчив алгоритъм.)

И още - алгоритъмът почти не зависи от предварителното нареждане на елементите в масива (за разлика, например от Quicksort).

Основен недостатък на сортирането чрез сливане е това, че изисква допълнителна памет, пропорционална на ***n***: за допълнителен масив или за указатели.

Подобно повечето съвременни методи за сортиране, сортирането чрез сливане се държи лошо при малки ***n***. Както и в случая на Quicksort, добри резултати се получават при комбинирането му с други методи. Стандартен подход е, когато броят на елементите на разглежданото множество падне примерно под 15-20 да се използва сортиране чрез вмъкване. Към групата алгоритми, реализиращи идеята за сортиране чрез сливане, се отнася и четно-нечетно сортиране чрез сливане на Батчер.

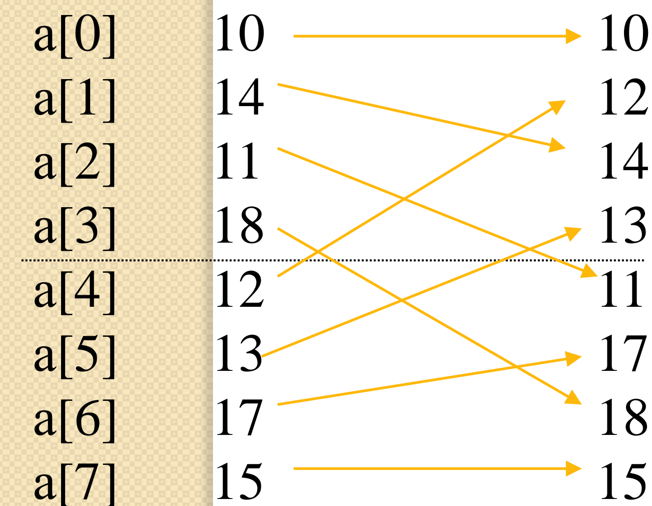
3. Алгоритъм на Бетчер
(или Четно-нечетно сортиране чрез сливане по метода на Бетчер - Batcher' odd-even mergesort)

Алгоритъмът, разработен през 1968 г. от Бетчер, действа по принципа "разделяй и владей" и е основан само на две абстрактни операции:

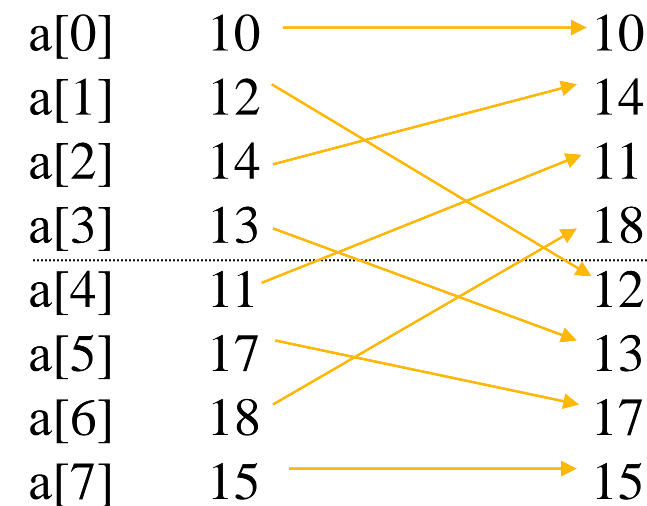
- сравнение-размяна;
- перфектно "карторазбъркване" (и обратния й еквивалент - перфектно "картоподреждане") за преместване на данните.

Първата операция може да бъде реализирана от всеки алгоритъм за сортиране. Втората операция представлява следното:

- перфектно "карторазбъркване"



- перфектно "картоподреждане"



Всички елементи от първата половина на масива ще имат четни индекси, от втората - нечетни. При обратната операция - елементите с четните индекси отиват в първата половина на масива, с нечетните - във втората.

Перфектното карторазбъркване пренарежда масива по начин, съответстващ на начина, по който тесте карти би могло да се пренареди, когато се разбърква от професионалист. То се разцепва прецизно на две половини, после се прави “розетка”, така че картите от двете половини се вземат алтернативно, за да направят разбърканото тесте. Първата карта винаги се взема от горната половина. Ако броят на картите е четен, двете половини имат еднакъв брой карти; ако броят на картите е нечетен, допълнителната карта е в края на горната половина.

Функция *shuffle*, извършваща перфектно карторазбъркване:

```
void shuffle (int a[], int lf, int rt)
{
    int i, j, m=(lf+rt)/2;
    static aux[maxn];              // временен масив
    for (i=lf, j=0; i <=rt; i+=2, j++)
        { aux[i]=aux[lf+j]; aux[i+1]=a[m+1+j];}
    for (i=lf; i<=rt; i++)     a[i]=aux[i];
}
```

Функция *unshuffle*, извършваща перфектно картоподреждане:

```
void unshuffle (int a[], int lf, int rt)
{   int i, j, m=(lf+rt)/2;
     static aux[maxn]; // временен масив
     for (i=lf, j=0; i <=rt; i+=2, j++)
            { aux[lt+j]=aux[i]; a[m+1+j] = aux[i+1];}
     for (i=lf; i<=rt; i++)     a[i]=aux[i];
}
```

Нечетно-четно сливане на Бетчер:

```
merge_B(int a[], int lf,  int rt)
{ int i, m=(lf+rt)/2, rab;
   if (rt==lf+1)   {rab=a[lf]; a[lf]=a[rt]; a[rt]=rab;}
   if (rt<lf+2)   return;
   unshuffle(a, lf, rt);
   merge_B(a, lf, m);
   merge_B(a, m+1, rt);
   shuffle(a, lf, rt);
   for (i=lf+1; i<rt; i+=2)
         {rab=a[i]; a[i]=a[i+1]; a[i+1]=rab;}
}
```

Самото сливане на Бетчер е рекурсивен метод "разделяй и владей". За да направим сливане 1-към-1, просто използваме операция "сравнение-размяна". От друга страна, за да направим ***n***-към-***n*** сливане, ние извършваме картоподреждане, за да получим две ***n/2***-към-***n/2*** задачи за сливане, после ги решаваме рекурсивно, за да получим два сортирани масива. Като извършим карторазбъркване на тези два масива, получаваме масив, който е почти сортиран - всичко, което е необходимо да направим, е да преминем още веднъж през масива с ***n/2 - 1*** на брой независими операции "сравнение-размяна": между елементи ***2i*** и ***2i+1*** за ***i*** от 1 до ***n/2 - 1*** .

Подробно описание на алгоритъма - в книгата на Робърт Седжуик, *Алгоритми на С, том 2*, Софтпрес, С., 2002.

Сортиране чрез разпределение

Методът, за разлика от разгледаните досега, не изисква сравнение на целите ключове.

Предполага се, че множеството на ключовете не е голямо и всеки ключ има следния вид:

$$a_1 \ a_2 \ \ldots \ a_{k-1} \ a_k$$

където всеки компонент a_i е елемент на някакво крайно множество (например - латинската азбука).

Компонентите се сравняват отдясно наляво в лексикографски ред:

s d a f g z < a s b f g z
a s b f g z

Самото сортиране на ключовете се извършва с помощта на друг устойчив алгоритъм за сортиране.

Сортиране чрез трансформация

За сортиране чрез сравнение:

в най-лошия случай - $O(n^2)$

в най-добрия случай - $O(n*log_2n)$

Възможно ли е да не се използва сравнение?

Каква сложност може да се постигне?

1. Сортиране чрез множество

Алгоритмът е приложим върху множество ***M*** с ***n*** на брой елементи, отговарящо на следните условия:

- елементите на множеството са естествени числа от даден затворен интервал ***[a, b]***, съдържащ ***k = b-a+1*** цели числа. Не е задължително всяко число от интервала ***[a, b]*** да бъде елемент на множеството, но всички елементи на множеството трябва да са от интервала ***[a, b]***;
- елементите не са повтарят (***M*** е множество!)

Разполагаме с масив ***m[ ],*** чиито елементи искаме да подредим. Сортирането става линейно, посредством едно преминаване през елементите на масива, последвано от обхождане на числата от интервала. В началото инициализираме с ***0*** елементите от множество ***set[ ]*** (това също е масив, чрез който представяме нашето множество ***M***. Масивът нека да е от тип ***char*** – ще го използваме като масив със стойности ***0*** и ***1***). След това преминаваме през масив ***m*** и за всеки негов елемент установяваме в ***1*** съответния елемент на множеството.

На втората стъпка извършваме проверка последователно за всички числа от интервала, дали принадлежат на ***set[ ]***. В началото на втория пас считаме, че ***m[]*** не съдържа нито един елемент и го изграждаме наново, този път вече като сортирана последователност:

```
#include <iostream>
using namespace std;
const int  MaxValue=10;
void input (int m[], int n);
void output (int m[], int n);
void setSort (int m[], int n);
```

```
void main()
{           int m[MaxValue];
            int n;
            do
                 {     cout <<"\nInput n: “;     cin>>n; }
            while (n>MaxValue);
            input (m,  n);
            setSort (m, n);
            output ( m,  n);
}
void input (int m[], int n)
{        for (int i=0; i<n; i++)
            {
                        cout<<"\nm["<<i<<"]=";
                        cin>>m[i];
            }
}
```

```
void setSort (int m[], int n)
{           char set[MaxValue];
            int i, j;
            for (i=0; i<MaxValue;i++)
                        set[i]=0;
            for (j=0; j<n; j++)
                        set[m[j]]=1;
            for (i=j=0; i<MaxValue; i++)
                        if (set[i])
                                    m[j++]=i;
}

void output (int m[], int n)
{
            for (int i=0; i<n; i++)
                        cout<<m[i]<<"  ";
}
```

Сложността на алгоритма е ***O(k+n)*** и при стойности на ***n***, близки до ***k***, би могла да се разглежда като линейна относно ***n***.
Но това не е „безплатно":

- има тежко допълнително предположение за стойностите и типа на ключовете на сортираните елементи (например, така не можем да сортираме реални числа);
- допълнителна памет от порядъка на ***n*** елемента (за масива, описващ работното множество ***set[ ]***).

2. Сортиране чрез броене

Представлява развитие на сортирането чрез множество, основният недостатък на който беше уникалност на ключа (множеството не допуска повторения!).

Да предположим, че искаме да сортираме елементите на множество от естествени числа от заден интервал, всяко от които може да се среща не повече от ***d*** на брой пъти (пак ще сортираме цели числа, а не записи). Дефинираме масив ***cnt[ ]***, като ***cnt[i]*** съдържа броя на срещанията на числото ***i***.

Започваме с нулиране на елементите на ***cnt[ ]***, след което преминаваме през елементите на ***m[ ]*** и намираме броя срещания на всяко число от ***[a, b]***. На втората стъпка преминаваме през елементите на ***cnt[ ],*** включвайки в сортираната последователност ***cnt[x]*** на брой пъти на числото ***x,*** за всяко цяло ***x***$\in$ ***[a, b]***.

```
#include <iostream.h>
const int  MaxValue=10;
void input(int m[], int n);
void output(int m[], int n);
void CountSort(int m[], int n);
```

```
void main()
{           int m[MaxValue];
            int n;
            do
            {           cout <<"\nInput n: ";
                        cin>>n;
            }
            while (n>MaxValue);
            input(m, n);
            CountSort (m, n);
            output (m, n);
}
void input (int m[], int n)
{          for (int i=0; i<n; i++)
            {           cout<<"\nm["<<i<<"]=";
                        cin>>m[i];
            }
}
```

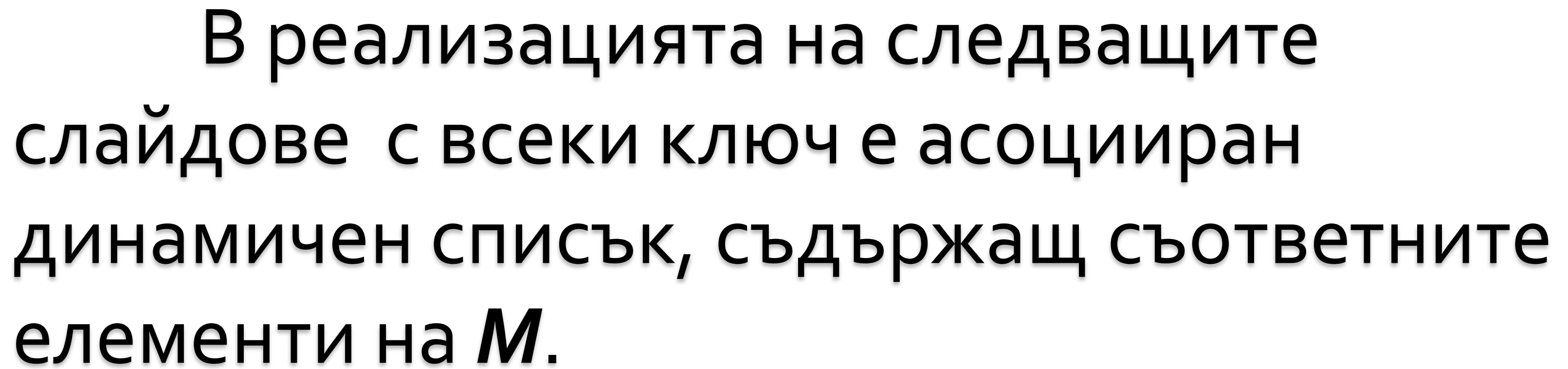
```
void CountSort (int m[], int n)
{           char cnt[MaxValue];
            int i, j;
            for(i=0; i<MaxValue; i++)
                        cnt[i]=0;
            for(j=0; j<n; j++)
                        cnt[m[j]]++;
            for(i=j=0; i<MaxValue; i++)
                        while (cnt[i]--)
                                    m[j++]=i;
}

 void output (int m[], int n)
{
            for (int i=0; i<n; i++)
                        cout<<m[i]<<"  ";
}
```

Алгоритмът работи добре при предположението, че всяко число се среща не повече от ***d*** пъти, за някаква разумна предварително дефинирана константа ***d***. Теоретично областта на приложение на този алгоритъм остава ограничена от следните изисквания:

- елементите да бъдат от интервала ***[a, b]*** , като не е задължително всяко число от ***[a, b]*** да бъде стойност на подреждания масив, но всички стойности на масива трябва да бъдат от ***[a, b];***
- всяко число от ***m[ ]*** се среща не повече от ***d*** пъти;
- да се сортират само числа, а не произволни данни (записи или структури).

Основният проблем е необходимостта да се пази информация не само за това, дали дадено цяло число ***x*** от ***[a, b]*** се среща като ключ на елемент от ***M***, но и стойността на този елемент.

Ще организираме единично свързан динамичен списък. Тъй като броят на срещанията на даден ключ не надвишава ***d*** , то всеки такъв списък може да съдържа най-много ***d*** елемента. Заделянето на статична памет с размер ***d*** за всеки ключ, обаче, води до излишно разхищение.

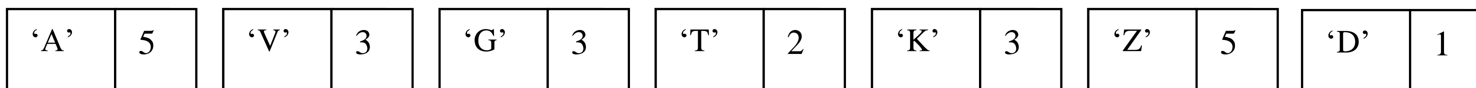


В реализацията на следващите слайдове с всеки ключ е асоцииран динамичен списък, съдържащ съответните елементи на ***М***.

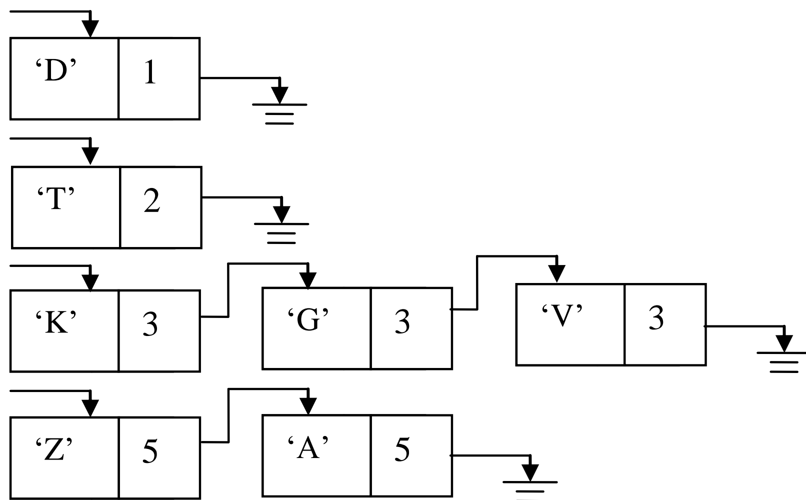
Поддържането на единствен указател (към началото на списъка) прави сортирането *неустойчиво*, защото елементите с еднакви ключове попадат в изходната последователност в ред, обратен на началния, т.к. елементите винаги се вмъкват в началото на списъка.

Например:

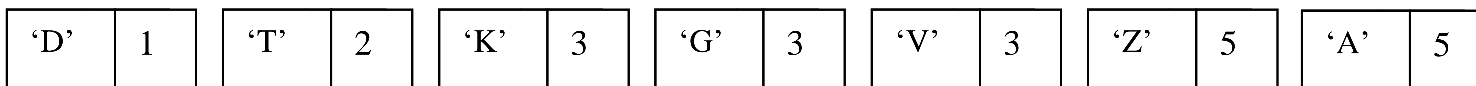
Входна последователност:



Формиране на списъци:



Изходна последователност:



Недостатъкът може да бъде премахнат, но ще се наложи допълнително обхождане на списъка и алгоритмът ще се усложни. Сложността на алгоритъма отново е ***O(k+n)*** , т.е. е линейна спрямо ***n***.

```
#include <iostream.h>
const int MaxValue=10;
struct elem
{
  char name;
  int num;
} m[MaxValue];
struct ptr
          {
                    struct elem data;
                    ptr *next;
          } *lst[MaxValue], *p;
```

```
void input(elem m[], int n);
void output(elem m[], int n);
void CountSort_List(elem m[], int n);
 void main()
{        int n;
         do
         {        cout <<"\nInput n: ";
                  cin>>n;
         }
         while (n>MaxValue);
         input (m, n);
         CountSort_List( m, n);
  output(m, n);
}
```

```
void init()
{
 for (int i=0; i<MaxValue; i++)
     lst[i]=NULL;
}
void input(elem m[], int n)
{
        for(int i=0; i<n;i++)
        {
                cout<<"\nm["<<i<<"].name=";
                cin>>m[i].name;
                cout<<"\nm["<<i<<"].num=";
                cin>>m[i].num;
        }
}
```

```
void CountSort_List(elem m[], int n)
{  int i, j;
   for (j=0; j<n; j++) // Разпределяне на елементите по списъци
                 {        p=new ptr;
                          p->data.name=m[j].name;
                          p->data.num=m[j].num;
                          p->next=lst[m[j].num];
                          lst[m[j].num]=p;
                 }
  for (i=j=0; i<MaxValue; i++) // Формиране на сорт. масив
     while (NULL!=(p=lst[i]))//Докато списъкът съдържа ел-ти
                 {       m[j++]=lst[i]->data;
                          lst[i]=lst[i]->next;
                          delete p;
                 }
}
```

```
void output(elem m[], int n)
{
        cout<<"\nSorted array: \n";
        for(int i=0; i<n; i++)
          cout<<m[i].name<<" "<<m[i].num<<"    ";
}
```

Въпреки направените подобрения, сортирането чрез броене наследява повечето недостатъци на сортирането чрез множество и не може да се смята за универсален подход.

Основният проблем на алгоритма е необходимостта от поддържане на допълнителен масив с размер, пропорционален на броя на възможните стойности на ключовете ***k*** (напомняме, че елементите на множеството ***M*** са естествени числа от даден затворен интервал ***[a, b]***, съдържащ ***k = b-a+1*** цели числа. Не е задължително всяко число от интервала ***[a, b]*** да бъде елемент на множеството, но всички елементи на множеството трябва да са от интервала ***[a, b]***). Очевидно е, че ***k*** може да бъде достатъчно голямо число и да не позволява заделяне на необходимия размер памет.

Сортирането чрез броене преминава веднъж през елементите на множеството, което изисква време ***O(n)***, и веднъж - през множеството от допустимите стойности на ключовете за време ***O(k)***. Общата алгоритмична сложност става ***O(k+n)***. Полученият резултат показва, че алгоритмът е линеен едновременно по ***k*** и ***n*** . Нас ни интересува сложността на алгоритма по отношение на броя на елементите ***n***. Но ако диапазонът на разглежданите стойности е голям, т.е. $k>n$, например, $k=n^3$, то сложността на алгоритма става $O(n^3)$. Получаваме сериозна неефективност, което трябва да се има предвид.

3. *Побитово сортиране*

Идеята на побитово сортиране до голяма степен се основава на двоичното вътрешно представяне на числата в компютъра. Нека е зададено множество от цели числа без знак, чиито елементи ще сортираме. Разделяме числата в два списъка в зависимост от стойността на най-младшия им двоичен разред. Четните числа попадат в първия списък, а нечетните – във втория. След това вторият списък се добавя към края на първия, при което се образува общ списък. За разлика от сортирането чрез броене, тук новия елемент се слага в края на съответния списък. Повтаряме операцията с предпоследния бит, след което с пред-предпоследния и т.н. до най-старшия бит. Описаният процес води до получаване на сортирана последователност:

Например, имаме следната входна последователност от данни:

5 7 2 1 4 5

Двоичните кодове са:

101 111 010 001 100 101

Разделяне и подреждане по най-младшия бит (0-вия):

2 4 + 5 7 1 5

010 100 + 101 111 001 101

Разделяне и подреждане по 1-вия бит:

4 5 1 5 + 2 7

100 101 001 101 + 010 111

Разделяне и подреждане по 2-рия (най-старшия) бит:

1 2 + 4 5 5 7

001 010 + 100 101 101 111

Методът е устойчив!

Изхождайки от вътрешното представяне на данните в компютъра, не е трудно да се съобрази, че описания метод е приложим (с минимални модификации) за сортиране на произволен тип данни (масиви, записи, низове). Но има и проблеми – сортиране на отрицателни числа (вътрешно се представят в допълнителен код) и на числа с плаваща запетая.

Реализацията може да бъде динамична (чрез списъци) или масиви.

В източника [3] - Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, може да се намери пример, реализиран чрез динамични списъци.

Побитовото сортиране е линейно и има сложност от порядъка на C*n, където C е броят на разредите на сортираните числа.

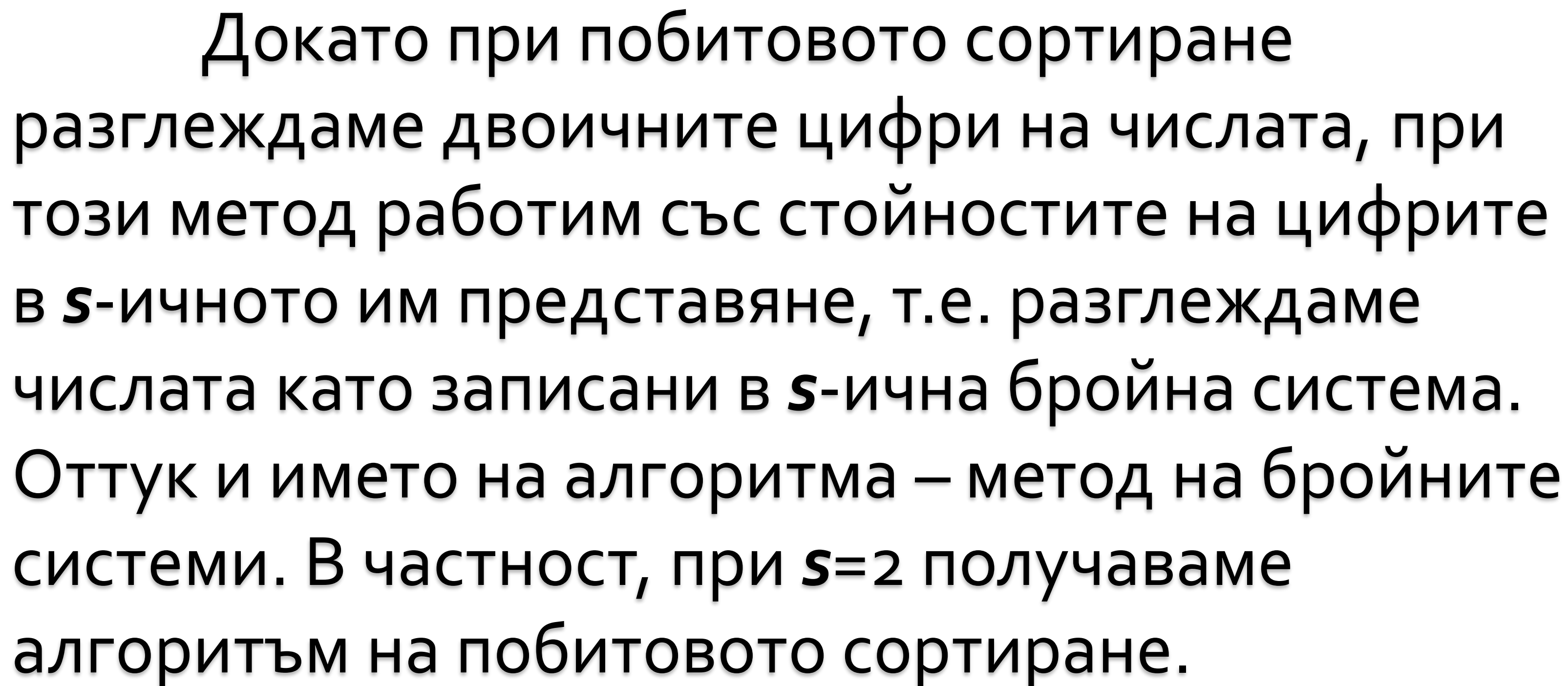
Алгоритмът може да се обърне и да преглежда битовете от най-старшия към най младшия.

Подходът е добър при относително равномерно разпределение на ключовете в интервала. В противен случай много от стъпките се оказват излишни.

4. *Метод на бройните системи*

Използва идеята на побитовото сортиране, но тук се създава по-голям брой списъци.

Нека имаме ***s*** на брой списъка. На първата стъпка елементите се разпределят по списъците в зависимост от остатъка, който дават при деление на ***s***. При това остава в сила изискването добавянето да става само в края на списъците. Следва конкатенация на списъците по реда на нарастване на остатъците. Процесът се повтаря до изчерпване на цифрите на числата.



Докато при побитовото сортиране разглеждаме двоичните цифри на числата, при този метод работим със стойностите на цифрите в ***s***-ичното им представяне, т.е. разглеждаме числата като записани в ***s***-ична бройна система. Оттук и името на алгоритма – метод на бройните системи. В частност, при ***s***=2 получаваме алгоритъм на побитовото сортиране.

По принцип няма ограничения за броя на списъците. Но все пак трябва да се внимава, тъй като по-големия брой списъци ***s*** ще влоши скоростта на сортиране за достатъчно малки ***n*** (например, за ***n***<***s***). За предпочитане този брой да бъде степен на 2 с цел по-ефективно намиране на съответните остатъци.

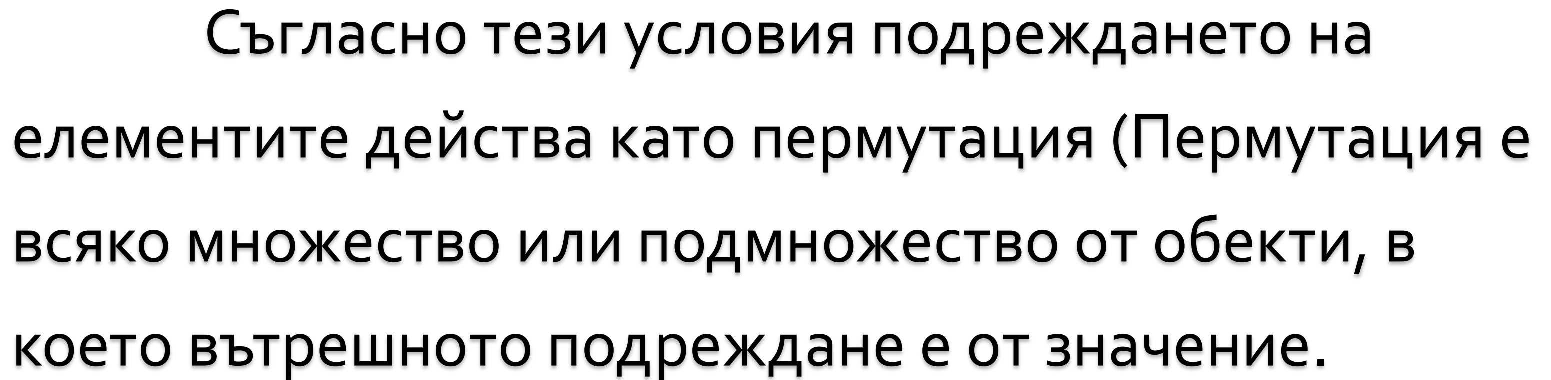
5. Сортиране чрез пермутация

Сортирането чрез пермутация е приложимо при по-силни ограничителни условия от вече разгледаните методи. Към известните условия:

- елементите на множеството са естествени числа от даден затворен интервал ***[a, b]***, съдържащ ***k = b-a+1*** цели числа. Не е задължително всяко число от интервала ***[a, b]*** да бъде елемент на множеството, но всички елементи на множеството трябва да са от интервала ***[a, b]***;
- елементите не са повтарят (***M*** е множество!)

се добавя още едно условие:

- всяко число от интервала ***[a, b]*** задължително е елемент на множеството ***M***.



Съгласно тези условия подреждането на елементите действа като пермутация (Пермутация е всяко множество или подмножество от обекти, в което вътрешното подреждане е от значение.

Пермутациите се различават от комбинациите, за които вътрешното подреждане не е от значение).

Това ни дава ефективен линеен алгоритъм, позволяващ сортиране на елементите на ***M*** на място, без използване на допълнителен масив, какъвто ни беше необходим при сортиране чрез множество или чрез броене.

В процеса на сортиране не се извършва пряко сравнение на ключовете на различните елементи. Единствената проверка е дали елементът с ключ ***i*** е на ***i*** –та позиция. В случай, че на позиция ***i*** стои елемент с ключ ***j, j=m[i], i≠j***, ще разменим ***i*** –тия и ***j*** –тия елементи, при което елементът с ключ ***j*** ще отиде на ***j***-та позиция. Ако след размяната на ***i*** –та позиция стои елемент с ключ на ***i***, то преминаваме към разглеждане на следващия (***i +1***)-ви елемент на ***m[ ].*** В противен случай извършваме нова размяна – с елемента с индекс ***m[i]*** .

Ще илюстрираме описания метод върху пермутацията 4375612. Започваме с i=1 и последователно разменяме първия елемент ***m[1] с m[m[1]]***, докато на позиция 1 не дойде елементът с ключ 1. Ще изписваме само ключовете, като ще подчертаваме елементите, които разменяме:

4	3	7	***5***	6	1	2
5	3	7	4	***6***	1	2
6	3	7	4	5	***1***	2
1	3	7	4	5	6	2

Сега числото 1 е на мястото си. Продължаваме с 2:

1	***3***	***7***	4	5	6	2
1	***7***	3	4	5	6	***2***
1	2	3	4	5	6	7

Броят на размените не надвишава ***n***.

Броят на сравненията не надвишава 2***n*** .

Защо?

Защото всеки елемент на пермутацията се преглежда най-много два пъти: веднъж при преминаване по съответния му цикъл и втори път при преминаване през крайната му позиция. В случай, че даден елемент е на мястото си в изходната пермутация, той ще участва само в едно сравнение.

Заключение:

Сортирането чрез трансформация има линейна (или близка към линейната) сложност по време.

Но! Алгоритмите не са универсални и могат да се използват само за определени специфични видове данни.

Паралелно сортиране

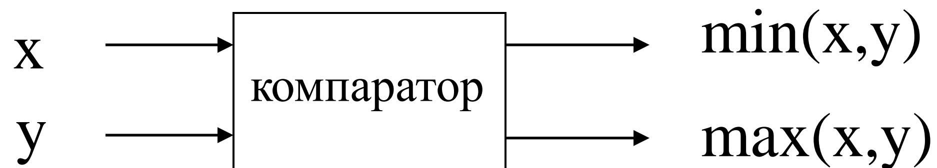
Многопроцесорни системи

Мрежови технологии

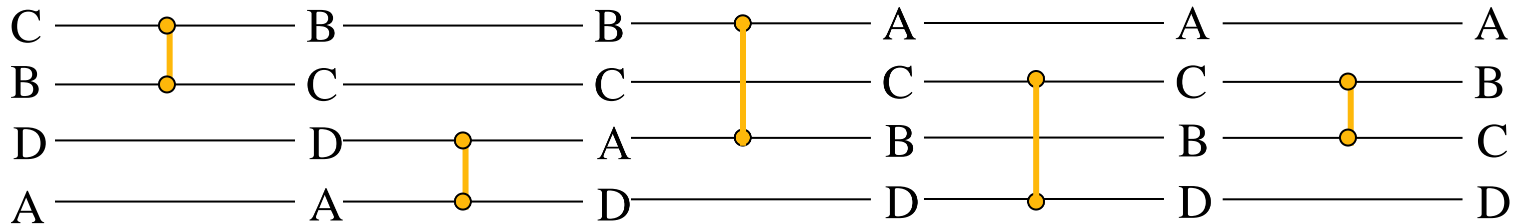
Специализирани процесори

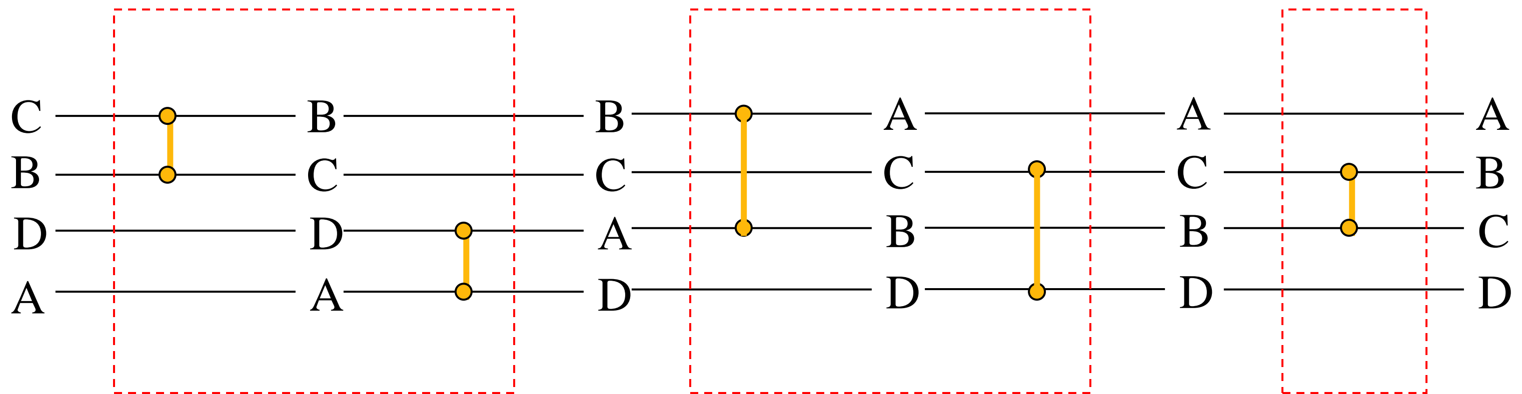
Паралелни алгоритми

Абстрактна машина, представляваща модул за сравнение-размяна или компаратор:



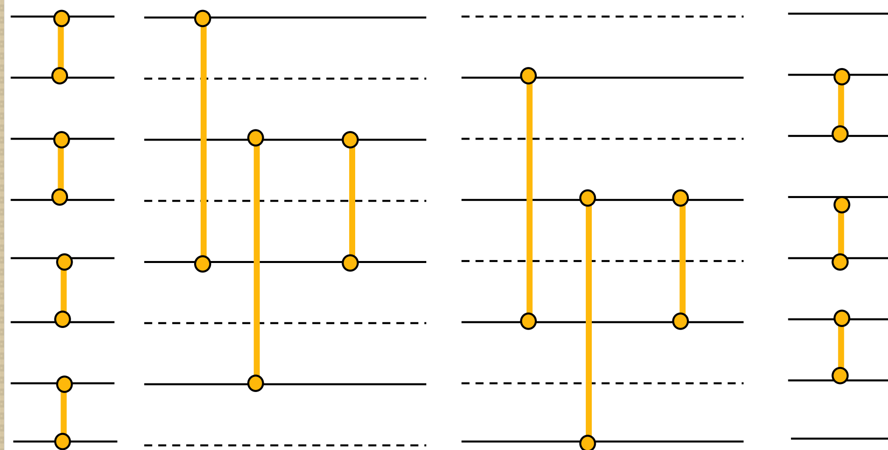
Компараторите (броят им не се ограничава) се свързват в сортираща схема или мрежа:

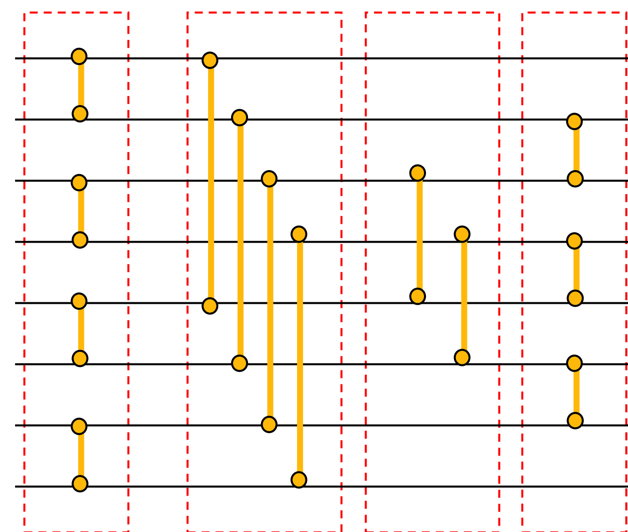




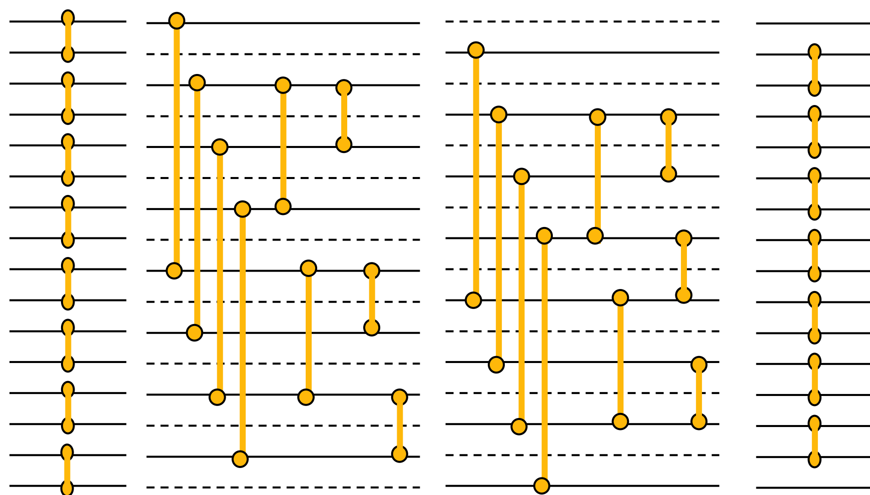
паралелни фази

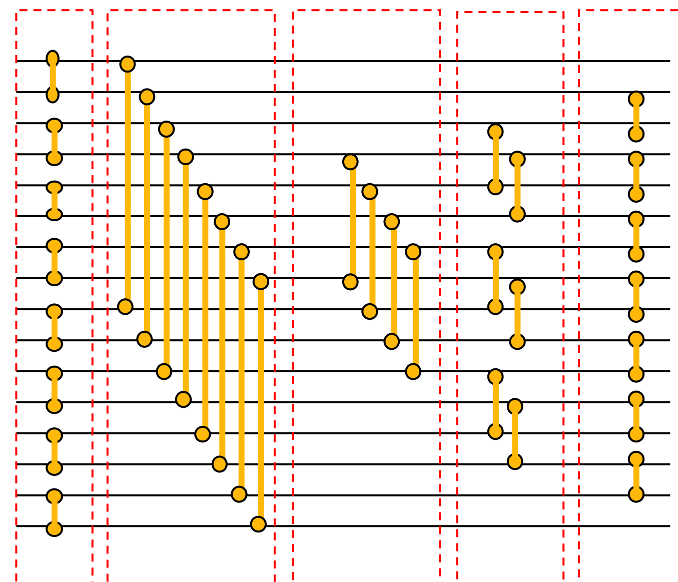
Сливаща мрежа за сортиране на 8 елемента:



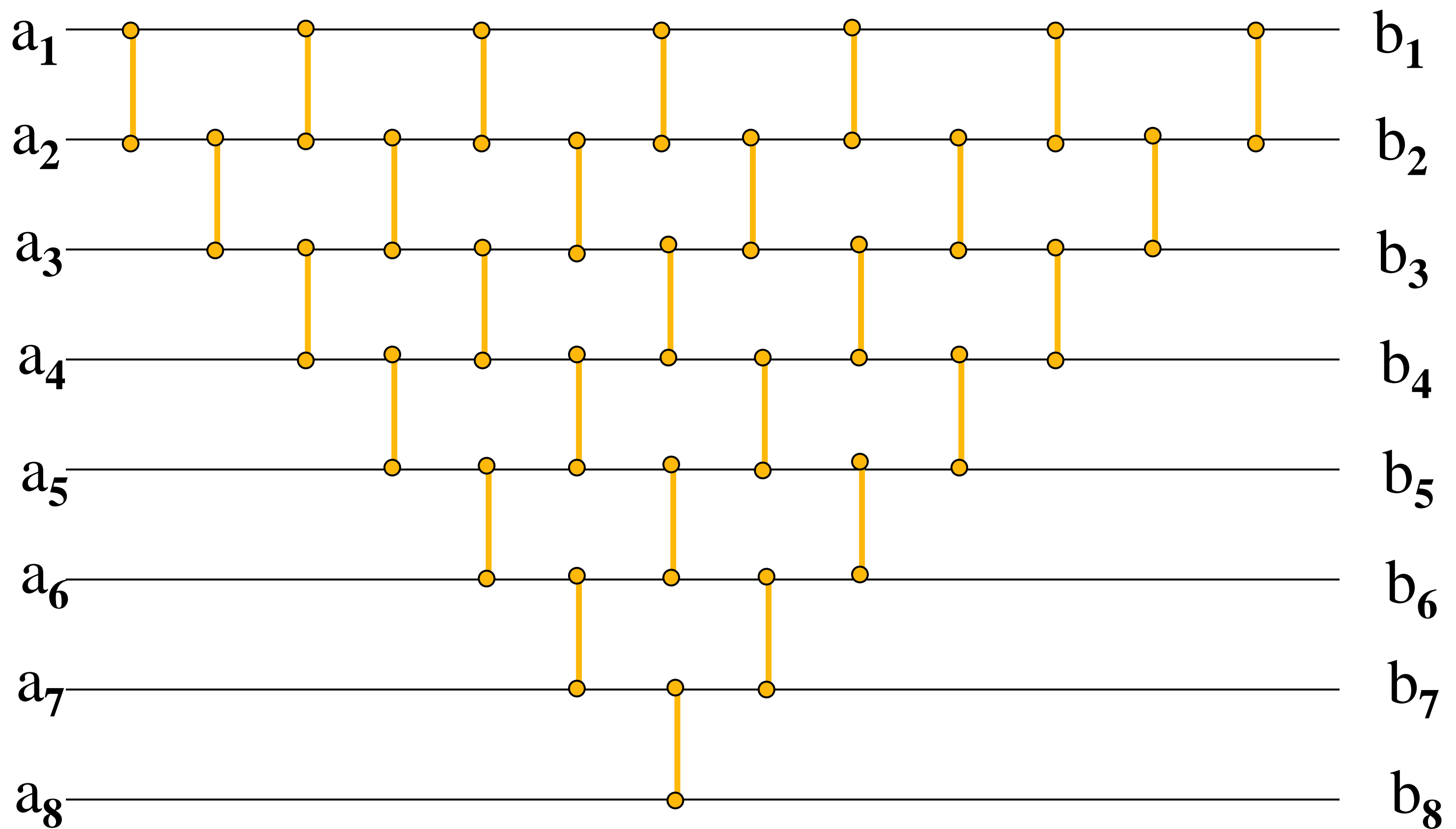


Сливаща мрежа за сортиране на 16 елемента:





Сортираща схема за сортирането чрез вмъкване:



Външно сортиране

Разгледаните методи за сортиране са предназначени за подреждане на масиви от данни, намиращи се в ОП. Но бихме могли да подреждаме и файлове, намиращи се на някакъв външен носител. За тази цел са необходими алгоритми за външно сортиране.

Разликата между методите за вътрешно и външно сортиране е по-скоро прагматична, отколкото фундаментална. Т. к. C/C++ позволява работа с файлове в режим на произволен достъп, бихме могли да адаптираме вече разгледаните методи и за файлове. Но трябва да се имат предвид няколко момента:

- Повечето методи за вътрешно сортиране сортират *in situ* (на място), т. е. използват само един масив. Както данните, които трябва да бъдат сортирани, така и подредените вече данни са в този масив и не са нужни временни масиви. От друга страна, външните методи обикновено не са толкова икономични по отношение на паметта, а използват няколко файла.
- Последователния достъп до елементите на масив обикновено не е по-бърз от произволния достъп до масив. Това не е така при файловете, т. к. при последователния достъп до файлове големи количества данни могат да бъдат съхранявани в буфер. Така че използването на последователен достъп при външно сортиране може да се окаже по-бързо.

Балансирано многоходово сливане или BMM (Balanced multiway merging)

6 временни файла или ленти:

- 3 за вход
- 3 за изход

Входният файл съдържа следните данни:

hereisyourbooktheoneyourthousandsoflettershaveaskedmetopublish

N=5

herei syour bookt heone yourt housa ndsof lette rshav easke dmeto publi sh

Лента 0: eehir eehno dfnos aeeks hs
Лента 1: orsuy ortuy eeltt demot
Лента 2: bkoot ahosu ahrsv bilpu

Лента 0: eehir eehno dfnos aeeks hs
Лента 1: orsuy ortuy eeltt demot
Лента 2: bkoot ahosu ahrsv bilpu

a[0]=e a[1]=o a[2]=b a[2]=k

Лента 3: beehikooorrstuy abdeeeiklmopstu
Лента 4: aeehhnooorstuuy hs
Лента 5: adeefhlnorssttv

Лента 0: aabdeeeeeefhhhhiklnnoooooorrrrsssstttuuuvyy
Лента 1: abdeeehiklmopsstu
Лента 2:

Окончателен резултат (Лента 3):

aaabbddeeeeeeeeefhhhhiikkllmnnoooooooprrrrsssssstttttuuuuvyy

АЛГОРИТМИ ЗА ТЪРСЕНЕ

Алгоритмите за търсене, също както и разгледаните алгоритми за сортиране, се явяват най-често срещани съставни части на системно и потребителско програмно осигуряване. *Търсенето* е процес на преглеждане на неподредена или подредена по определен ключ или ключове съвкупност от данни. Търсенето, както и сортирането на данни, може да бъде вътрешно, когато данните са разположени в оперативната памет, и външно, когато данните или част от тях се намира на някакъв външен носител.

Ние ще се ограничим с разглеждането на алгоритми за вътрешно търсене. Данните в примерите представляват цели числа, организирани в масив, или както е прието да се нарича - в таблица. Таблицата може да бъде подредена или неподредена.

Максималната сложност на алгоритмите за търсене е ***O(n)***. Това е очевидно, т. к. максималният брой необходими сравнения е ***n***.

I. Линейно (последователно) търсене (Linear Search, Sequential Search)

```
int seq_search(int a[], int k)
// a[] - неподредена последователност от цели числа
// k - търсената стойност
{   int i=0;  //работна променлива-индекс
    while ((a[i]!=k)&&(i<num))   //num - броят на
                    i++;         //елементите
    if (a[i]==k)       return 1; //елементът е намерен
       else return 0;            //елементът не е намерен
}
```

Ако търсенето е успешно, средният брой на извършваните сравнения е ***n/2***, ако предположим, че търсеният елемент равновероятно може на се намира на произволно място в масива (***n*** - броят на елементите в масива). В най-лошия случай са необходими ***n*** операции за сравнение, ако търсенето е неуспешно.

Максималната сложност на алгоритъма е ***O(n)*** в случая, ако търсеният елемент е последен или не е включен в таблицата. Минималната сложност е ***O(1)***, ако търсеният елемент е първи в таблицата.

2. Линейно (последователно) търсене в подредена таблица (Ordered Table Searching)

Ако таблицата съдържа подредени данни, не е необходимо да се преглежда цялата последователност. Търсенето може да бъде прекратено предварително.

```
int seq_order_search(int a[], int k) //a[] - подредена
                            //последователност от цели числа
                            //k - търсената стойност
{   int i=0;                 //работна променлива-индекс
    while ((a[i]<k)&&(i<n)) i++;   //n  - броят на
    if (a[i]==k)                   //елементите
        return 1;                  //елементът е намерен
             else return 0;    //елементът не е намерен
}
```

Търсенето се прекратява при достигането на търсения елемент или първия по-голям от него, което означава, че елементът не е включен в таблицата. Броят на необходимите сравнения в случай на неуспешно търсене намалява средно до ***n/2*** .

3. Последователно търсене с преподреждане

Ако разполагаме с предварителна информация относно вероятността за достъп до всеки от елементите, бихме могли да ги подредим така, че най-често търсеният да бъде в началото на масива, следващия - непосредствено след него и т. н. Използването на подобна стратегия е особено ефективно при силно неравномерно разпределение, при което малък брой елементи се търсят с много голяма вероятност.

В случай, че не разполагаме с такава предварителна информация, бихме могли сами да си я получим с помощта на прости статистически наблюдения: достатъчно е към всеки елемент да добавим брояч на достъп до него. При всяко ново търсене ще актуализираме брояча на търсения елемент. Елементите са наредени по своите броячи, така след актуализацията той ще мине по-напред в списъка. Т. к. в списъка може да има многократно дублиране на честотите на достъп, в случай на размяна се налага съответно сравнение и със съседните елементи, докато се намери точната позиция, съответстваща на новата честота на елемента. Реорганизацията става за време, пропорционално на ***n***, като в най-лошия случай може да се наложи последния елемент в масива да стане първи, разменяйки се последователно с всички преди него. Повече информация - Н. Уирт. "*Алгоритми + Структури от данни = Програми*", Техника, 1988.

Предложената схема с поддържане на броячи е доста тромава и изисква допълнителна памет от порядъка на ***n***, но тя би могла да се оптимизира. В този случай се оказва изключително ефективно прилагането на една проста стратегия за реорганизация, не изискваща поддържане на броячи и водене на статистика. Вместо това, при всяко успешно търсене на елемент той се поставя в началото на масива. Разбира се, тази стратегия не ни гарантира оптимално подреждане на елементите съгласно честотата им на достъп, но пък е лесна за поддържане и достатъчно ефективна. Реорганизацията на масива става сравнително бързо, и колкото по-често е осъществяван достъп до даден елемент, с толкова по-голяма вероятност той ще се намира сред първите елементи на масива:

```
int search_reordering(int a[], int k)
{
   int rab, i=0;
   while ((a[i]!=k)&&(i<num)) i++;
   if (a[i]==k)
      {rab=a[0]; a[0]=a[i]; a[i]=rab; return 1;}
     else    return 0;
}
```

4. Търсене със стъпка. Квадратично търсене

Ако сравним първите два алгоритъма за търсене (последователно търсене и последователно търсене в подреден масив/таблица), то може да се каже, че вторият алгоритъм не използва достатъчно пълноценно наредбата на елементите. Ще поправим тази грешка, възприемайки следния подход. Нека изберем някаква стъпка ***k*** и последователно извършим проверка дали търсеният елемент е по-голям от първия елемент в масива, от (***k*** +1)-ия елемент, от (2***k*** +1)-ия елемент, от (3***k*** +1)-ия елемент и т. н. Процесът приключва при достигане на елемент, по-голям или равен на търсения ***x***, или на края на масива.

Да разгледаме по-внимателно горната схема (която се нарича *търсене със стъпка*). Нека предположим, че, прилагайки я, сме достигнали до елемент, по-голям от ***x*** . Сега можем да използваме последователно търсене в интервала, определен от последните две проби. В случай, че сме били излезли извън масива, можем да използваме последователно търсене от предишната проба до края на масива. Подобен подход води до силно съкращаване на броя на елементите, които се преглеждат от последователно търсене. Изложеният метод е обобщение на линейното търсене (***k*=**1*)*:

a[1] a[2] a[3] … a[k+1] ... a[2k+1] a[2k+2] … a[3k+ 1] …. a[n]

В описания горе метод търсенето започва от първия елемент (или **a[**1**])**. Доказано е, че този избор, както и **a[*n*],** е лош, понеже носи минимална информация. Много по-удачно е да се започне направо с **a[*k*].** (Пак се вижда, че при ***k*** =**1** се получава линейно търсене.)

Каква е ефективността на описания метод при фиксирано ***k*** и кой е най-лошия случай? Ясно е, че линейното търсене има еднаква цена за всички интервали от вида **[a[i*k+1], a[(i+1)*k]]**, т. к. се извършва върху еднакъв брой елементи. Изключение може да прави единствено последния интервал, който евентуално може да съдържа по-малко от ***k*** елементи. В най-лошия случай търсения елемент е в последния интервал, което означава, че ще бъдат необходими **[n/k]** сравнения, за да определим нужния ни интервал, в който да приложим линейното търсене. Към тях следва да се добави дължината на интервала, която при ***n***, кратно на ***k***, е ***k***-1. Оттук следва, че в най-лошия случай при търсене със стъпка ***k*** се извършват не повече от **[n/k]** + ***k*** - **1** сравнения.

Примерна реализация на описания алгоритъм:

```
unsigned seq_search(unsigned lf, unsigned rt, int key)
{ while (lf<rt)
      if (a[lf++]==key])
          return lf-1;              //елементът е намерен
   return -1;                       // елементът не е намерен
}
unsigned jmp_search(int key, unsigned step)
{ unsigned ind;
   for (ind=0; ind<n&&a[ind]<key; ind+=step);
   return  seq_search(ind+1<step ? 0 : ind+1- step, n<ind ? n: ind, key);
}
```

Максимален брой сравнения при различни стойности на ***n*** и ***k***:

n/k	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
1	1							
2	2	2						
3	3	2	3					
4	4	3	3	4				
5	5	3	3	4	5			
6	6	4	4	4	5	6		
7	7	4	4	4	5	6	7	
8	8	5	4	5	5	6	7	8

От таблицата се вижда, че най-добрите стойности на ***k*** са близки до ***n*/2**, т. е. са разположени в средата на съответния ред на таблицата или непосредствено вляво от нея. За да определим по-точно, кои стойности на ***k*** са най-добри и как зависят от ***n***, следва да определим при каква зависимост между ***k*** и ***n*** функцията ***f(k)*** приема минимална стойност:

$$f(k)=[n/k] + k - 1$$

Може да се докаже (чрез изследване на знака на втората производна), че това е стойността $k=\sqrt{n}$. Тогава $f(k)=2\sqrt{n}+1$. В този случай търсенето се нарича *квадратично*.

Постигнахме съществено подобрение - от ***n*** до $\sqrt{n}$. Бихме ли да постигнем повече? Да, ако след определяне на интервала, преди да извършим последователно търсене, ще приложим търсене с някаква нова стъпка ***m*** ($1<m<k$) и едва след това последователно търсене. Това подобрение има смисъл, ако дължината на интервала след първото търсене със стъпка ***k*** е голяма. Първият интервал се определя с не повече от ***k*** сравнения, вторият - с не повече от ***m*** сравнения, и накрая не повече от ***n*/(*k**m)** сравнения за последователното търсене. Можем да образуваме нова функция, която да минимизираме и ще получим $k=m=n/(k*m)$, откъдето $n=k^3$. За по-големи подробности – в книгата на Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TpoTeam Co., София, 2002.

5. Двоично търсене (Binary Search, Logarithmic Search)

В случай на голям брой елементи и голям брой търсения е удобно използването на *двоично търсене*. Двоичното търсене е още един добър пример на принципа "разделяй и владей" и използва идеята, позната от пирамидалното сортиране . Обикновено се прилага към подреден масив/таблица. Методът е рекурсивен. Търсената стойност ***k*** се сравнява с ключа на средния елемент в таблицата. Ако елементът е по-голям, търсенето продължава в дясната част от таблицата, в противен случай - в лявата. При съвпадение търсенето се прекратява. На всяка стъпка размерът на полето за търсене се съкращава приблизително двойно:

```
int Bin_search(int a[], int lf, int rt, int k)
// a[] - подредена последователност от num цели числа
// k - търсената стойност
// lf - left - лявата граница на индекса
// rt - right - дясната граница на индекса
// Обръщението към функцията изглежда така:
// Bin_search(a,0,num,k)
{  if (lf>rt) return 0;  int m=(lf+rt)/2;
   if (a[m]==k) return a[m];
   if (lf==rt) return 0;
   if(a[m]>k) return Bin_search(a, lf, m-1, k);
       else   return Bin_search(a, m+1, rt, k);
}
```

Ситуацията **lf=rt** отговаря на листата на дървото. Алгоритъмът предполага максимум [$\log_2 n$] +1 сравнения. Сложността на алгоритъма е **O(log n).**

Основният недостатък на двоичното търсене е тежката операция деление, извършвана на всяка стъпка от изпълнението на алгоритъма.

Реализацията му на C/C++ позволява да се избегне тази операция, като делението на 2 се извършва с изместване на делимото с една позиция надясно, т. е. редът

```
int m=(lf+rt)/2;
```

следва да се замени с:

```
int m=(lf+rt)>>1;
```

Описаният процес е рекурсивен. На практика обаче на всяка стъпка се разглежда точно един от двата подмасива. Вземайки предвид това, не е трудно да реализираме итеративно решение:

```
int bin_search (int  k)
{
    int lf=0, rt=n-1, m;
    while (lf <=rt)
      {
        m = (lf + rt)>>1;
        if (k <a[m])
           rt = m - 1;
             else  if (k>a[m])
                      lf =  m + 1;
                         else return m;  // m - индекс на намерения елемент
      }
    return (-1);                               // елементът не е намерен
}
```

Възможни са и други подобрения на алгоритъма, които могат да се намерят в книгата на Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TpoTeam Co., София, 2002.

6. Фибоначиево търсене

Представлява модификация на двоичното търсене. За разлика от двоичното търсене, тук за сравнение се избира не средния елемент в масива, а елемент с индекс F_{k-1}, като се има предвид, че броят на елементите в масива $n = F_k - 1$.

Числата на Фибоначи се задават с рекурентната формула:

$$F_0 = 0\ ,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}\ ,\ n >= 2$$

На първа стъпка сравняваме търсената стойност ***key*** с $a[F_{k-1}]$. В случай на равенство търсенето приключва успешно. Ако $key < a[F_{k-1}]$, на следващата стъпка се разглеждат елементите с индексите $0, 1, 2, \ldots, F_{k-1}-1$. Ако пък $key > a[F_{k-1}]$, разглежданите елементи ще имат индекси $F_{k-1}+1,\ F_{k-1}+2, \ldots, n$. В първия случай за следващата стъпка получаваме последователност с дължина $F_{k-1} - 1$, а във втория - $F_{k-2} - 1$. Процесът се повтаря върху неотхвърлената част до намиране на търсения елемент или до достигане на празната последователност.

По-детайлно описание на алгоритъма може да се намери в книгата на Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, TpoTeam Co., София, 2002.

Една възможна реализация на Фибоначиево търсене изглежда така

Функция за търсене на числата на Фибоначи:

```
int fib[n];     //помощен масив, в който се съхраняват
                       //числата на Фибоначи
int findFib(int n)     //функция за търсене на числата
{  int k;  fib[0]=0; fib[1]=1;
   for (k=2; ;k++)
     if ((fib[k]=fib[k-1]+fib[k-2]) > n)
      return k-1;
  return 0;
}
```

Функция за Фибоначиево търсене:

```
int fibsearch(int key, int n)  //Фибоначиево търсене
{ int p, q, r, k;
  k=findFib(n);
  p=fib[k-1]; q=fib[k-2]; r=fib[k-3];
  if (key>a[p]) p+=n-fib[k]+1;
```

```
 while (p>0)
    if (key==a[p])   return p;
       else
         if (key<a[p])
           if (0==r)  p=0;
             else
            { int t; p-=r; t=q; q=r;  r=t-r;}
          else
            if (1==q) p=0;
              else
               { p+=r;  q-=r;  r-=q; }
              return 0;
}
```

Алгоритъмът не съдържа операцията деление, но ефективността му не е по-голяма от тази на двоичното търсене. На практика алгоритъмът на Фибоначиево търсене изгражда балансирано дърво, което в най-лошия случай с 45% е по-високо от дървото на двоичното търсене.

7. Интерполационно търсене

Подходът също така представлява модификация на двоичното търсене, при което масивът се дели на две равни части и индексът на сравняваният елемент се изчислява като

$m = (lf+rt) / 2$ (lf - левия индекс, rt - десния индекс)

или

$m = lf + (rt-lf) / 2$

Получената формула показва, че следващата позиция на разделяне на интервала се получава, като към началото му добавим половината от дължината му.

Знаейки търсената стойност ***key***, ние можем да потърсим позицията й по следната интерполационна формула:

$m = lf + k*(rt - lf)$,

където $k = (key - a[lf]) / (a[rt] - a[lf])$

Коефициентът k замества константата **1/2** (в двоичното търсене) и ни дава приблизителната позиция на търсения елемент в разглеждания интервал.

Функция за интерполационно търсене:

```
int inter_search(int key, int num)
{  unsigned lf, rt, m;//lf - началния индекс в интервала
                      //rt - крайния индекс в интервала
                      //m - индекса на сравнявания елемент
   float k;           //коефициент  lf=0; rt=num-1;
   while (lf<=rt)
   {
    if (a[rt]==a[lf])
        if (a[lf]==key)
           return lf;          //елементът е намерен
        else
           return 0;           //елементът не е намерен
    k=(float) (key-a[lf])/(a[rt]-a[lf]); //коефициент на
                                         //позицията
    if (k<0||k>1)
      return 0;                          //елементът не е намерен
    m=(unsigned)(lf+k*(rt-lf)+0.5);
    if (key<a[m])
       rt=m-1;
     else  if(key>a[m])
              lf=m+1;
              else  return m;     //елементът е намерен
  }
 return 0;                        //елементът не е намерен
}
```

Методът е реализуем, когато ключовите стойности са числа или могат лесно да се интерпретират като такива, така че коефициента *k* да попада в интервала `[0, 1]`.

Алгоритъмът е добър при равномерно разпределение на стойностите в масива. Тогава интерполационното търсене извършва по-малко от $Log_2(log_2 n) + 1$ сравнения както при успешно, така и при неуспешно търсене. Препоръчва се при големи обеми данни и големи по размер ключове, когато сравненията отнемат много време.

АЛГОРИТМИ ЗА ХЕШИРАНЕ

Разгледаните методи за търсене предполагат, че търсеният елемент се съхранява в таблица и за откриването му е необходимо да се организира преглеждане на определена част от таблицата или в най-лошия случай - на цялата таблица.

Хеширането е метод за организация на таблицата, осигуряващ бързо намиране на търсения елемент и таблично преобразуване.

Разполагането на елементи в таблицата, която се нарича *хеш-таблица*, се осъществява чрез специално преобразуване на техните ключове. Всеки елемент на хеш-таблицата се характеризира с две полета - ключ ***key*** и данни ***data***. Ключът на елемента може да бъде произволна структура, идентифицираща го уникално. Функцията, която трансформира ключ в табличен индекс, се нарича *хеш-функция*. Тя трябва да отговаря на две основни условия: да не отнема много изчислително време и да дава стойности, които относително равномерно покриват цялото множество индекси в хеш-таблицата.

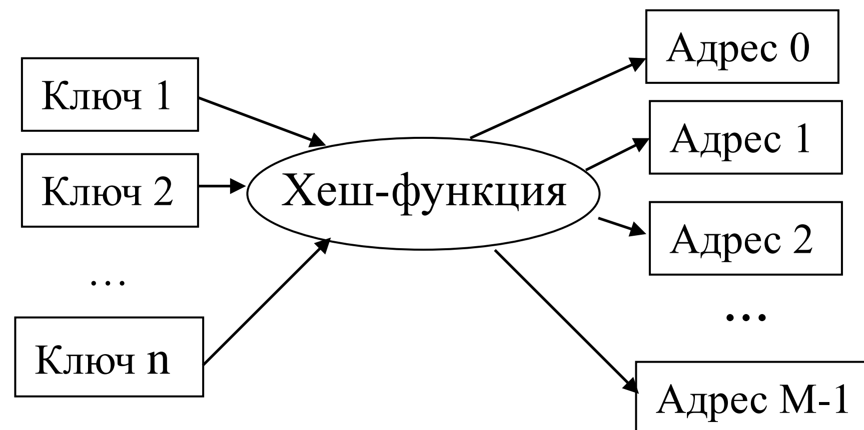
Когато ключът е целочислен, най-често хеш-функциите ***H(K)*** се задават като остатък от деление на ключа ***K*** върху ***M***, което определя размера на таблицата:

$$H(K) = K \bmod M \qquad 0 <= H(K_i) <= M-1$$

В случая, когато ключовете са символни низове, те се преобразуват в цели положителни числа чрез сумиране на ASCII-кодовете на влизащите в тях символи. Съществуват и други класически хеш-функции, които могат да с видят в специализираната литература.

Хеширането е процес, при който по зададен ключ на елемента се определя неговия адрес в хеш-таблицата:

0	…	…
	…	…
	$H(K_i)$	$A(i)$
	$H(K_j)$	$A(j)$
	…	…
	$H(K_m)$	$A(m)$
M-1	…	…



Методът е много бърз, т. к. осигурява директен достъп до търсения елемент чрез неговата хеш-функция. При използването на този подход следва да очакваме константно време за операциите *търсене* и *вмъкване* в таблицата, независимо какъв е нейния размер.

Някои автори дефинират хеш-таблицата като структура от данни, за която е характерен директен достъп до елементите, независимо от типа им.

Например, таблица, съдържаща информация за имена и факултетни номера на студенти от една студентска група. Факултетните номера уникално определят всеки елемент в таблицата, затова се използват като ключове:

Ключ	Индекс	Фак. номер	Име
026760 % 30 →	0	026760	Иван Иванов Иванов
…		…	
026775 % 30 →	15	026775	Вяра Янкова Павлова
…		…	
026728 % 30 →	28	026728	Ана Илиева Илиева
026819 % 30 →	29	026819	Петър Тодоров Пеев

026733 % 30 = 3

026793 % 30 = 3

Хеширането обаче притежава един съществен недостатък. Нека да предположим, че съществуват два елемента $A[i]$ и $A[j]$ с ключове K_i и K_j, които имат една и съща стойност на хеш-функцията: $H(K_i) = H(K_j)$. Това означава, че двата елемента би трябвало да се запишат с един и същи индекс в хеш-таблицата, което не е възможно. Подобна ситуация се нарича *колизия*.

От тук следва, че е изключително важен начина, по който се избира хеш-функция. Хеш-функцията трябва да минимизира броя на възможните колизии и да разпределя елементите равномерно в таблицата. Ако таблицата съществено по-голяма от реалния брой елементи, това ще намали вероятността от възникване на колизии, но използването й ще бъде неефективно от гледна точка на заетата памет.

Класически хеш-функции

Дали даден избор на хеш-функция е удачен се определя от две неща: тя не трябва да отнема много изчислително време и трябва да разпределя елементите колкото може по-равномерно в различните адреси на таблицата. Нека ключовете са цели числа. Ако не са , ще приемем вече споменатото правило: за ключове-символни низове ще сумираме ASCII-кодовете на влизащите в тях символи.

Нека е дадена хеш-таблица с капацитет $\boldsymbol{n}$ и елемент с ключ $\boldsymbol{k}$. Най-често срещаните в практиката хеш-функции (върху целочислен ключ) са следните:

- *Остатък при деление с размера на таблицата*

Това е прост и доста ефективен начин за хеширане: ключът $\boldsymbol{k}$ се разделя целочислено на $\boldsymbol{n}$ и се взема остатъкът от делението. При този подход не е удачно $\boldsymbol{n}$ да бъде степен на 2, т. к. в този случай хеш-кодът се определя от младшите битове на $\boldsymbol{k}$. Добре е хеш-кода да зависи от всички битове на ключа. За тази цел за $\boldsymbol{n}$ най-често се избира просто число.

- *Мултипликативно хеширане*

Това е друг широко разпространен начин. Избира се реална константа ***b***:

$$0 < b < 1$$

За даден ключ ***k*** хеш-функцията има вид:

$$H(k) = [n * \{k * b\}]$$

Тук $\{k * b\}$ означава дробната част на реалното число, т. е. $k * b - [k * b]$.

Въпреки, че изборът на константата ***b*** (при ограничението $0 < b < 1$) е произволен, за някои стойности практическите резултати са добри. Д. Кнут (D. Knuth. *The Art of Computer Programming*. *Volume 3: Sorting and Searching*, second edition, Addison-Wesley, Reading, MA, 1998) предлага за ***b*** да се използва златното сечение:

$$b = \frac{\sqrt{5} - 1}{2} = 0,6180339887...$$

- *Хеш-функции върху части от ключа*

1. Извличане на цифри

При тази схема от ключа се извличат само цифри, намиращи се на определени позиции (например - първата, третата и петата цифра от числото). Така за ключовете 123659, 425435, 546754, 676576 ще получим хеш-адреси съответно 135, 453, 565 и 667. В този случай ***n*** трябва да е 1000. Методът е добър, когато числата не съдържат много повтарящи се цифри.

2. Сгъване

Този метод се прилага най-често, когато ключовете са много големи числа. Възможни са разновидности, но като цяло всички се основават на разделяне на ключа на части и извършване на някакви аритметични действия върху получените части. Например, числото може да се раздели на две (или три и повече) части и сумата от получените числа да определя хеш-адреса.

3. Повдигане на средата в квадрат

Тази схема се основава на извличане на средните ***p*** цифри на ключа и повдигането им в квадрат. Например, за ключа 125657134280980 средните три цифри са 134, квадратът на числото е 17956. Ако резултатът надхвърля ***n***, се премахват първите няколко значещи цифри. Например, ако ***n*** =10001, то от 17956 се премахва първата цифра и полученият хеш-адрес е 7956. (Последната операция не е еквивалента на намиране на остатъка при деление на ***n***.)

- *Хеш-функции върху символни низове*

Символните низове са най-често хешираният тип данни. Намирането на добра хеш-функция за тях е сериозен проблем. Ще използваме следната схема:

result = инициализиране();
while (c = следващ_символ)
{result=комбиниране (result, c); result=вътрешно_модифициране(result);}
result=допълнително_модифициране (result);

2. Сгъване

Този метод се прилага най-често, когато ключовете са много големи числа. Възможни са разновидности, но като цяло всички се основават на разделяне на ключа на части и извършване на някакви аритметични действия върху получените части. Например, числото може да се раздели на две (или три и повече) части и сумата от получените числа да определя хеш-адреса.

3. Повдигане на средата в квадрат

Тази схема се основава на извличане на средните ***p*** цифри на ключа и повдигането им в квадрат. Например, за ключа 12565713428098 0 средните три цифри са 134, квадратът на числото е 17956. Ако резултатът надхвърля ***n***, се премахват първите няколко значещи цифри. Например, ако ***n*** =10001, то от 17956 се премахва първата цифра и полученият хеш-адрес е 7956. (Последната операция не е еквивалента на намиране на остатъка при деление на ***n***.)

- *Хеш-функции върху символни низове*

Символните низове са най-често хешираният тип данни. Намирането на добра хеш-функция за тях е сериозен проблем. Ще използваме следната схема:

result = инициализиране();
while (c = следващ_символ)
{result=комбиниране (result, c); result=вътрешно_модифициране(result);}
result=допълнително_модифициране (result);

Адитивно хеширане е най-често разпространеното (класическо), но за съжаление най-неефективното хеширане. Сумират се ASCII-кодовете на символите и сумата се взета по модул размера на масива:

```
unsigned long hash_function(const char *key, unsigned long size)
{ unsigned long result = 0;
  while (*key)
    result+=(unsigned char) *key++;
  return result % size;
}
```

Обикновено в дължината на низа се включва сумата, за да може изрично да влияе на хеш-кода:

```
unsigned long hash_function(const char *key, unsigned long size)
{ unsigned long result = strlen(key);
  while (*key)
    result+=(unsigned char) *key++;
  return result % size;
}
```

Дължината на символния низ би могла да се подава като параметър, което спестява обръщението към функцията *strlen(key)*. Или просто да се получи като разлика между текущата позиция преди и след обхождането на низа:

```
unsigned long hash_function(const char *key, unsigned long size)
{  const char * save_key=key;
   unsigned long result = 0;
   while (*key)
      result+=(unsigned char) *key++;
   result +=save_key - key;
   return result % size;
}
```

Размерът на масива най-често се избира да бъде просто число. Понякога обаче се използва масив с размер степен на 2, при което последния ред може да се опрости до:

```
return result & (size - 1);
```

Други подходи при работа със символни низове:

- ротирано хеширане (вместо сумиране се използват само поразредени операции. Размерът на масива трябва да бъде просто число.);
- хеширане едно по едно (вариация на горното, но с допълнителни манипулации върху *result*);
- хеширане по Пиърсън (използва допълнителен масив, съдържащ пермутации на числата от 0 до 255. Полученият хеш-код е еднобайтов: от 0 до 255. По-голям код може да се получи, ако функцията се извиква няколко пъти с различни масиви, при което всяко извикване ще дава един байт от резултата;

- хеширане по CRC (стойностите в масива се генерират като псевдослучайна
двоична последователност, удовлетворяваща определени условия);
- други класически хеш-функции.

За по-големи подробности - Преслав Наков, Панайот Добриков. *Програмиране = ++Алгоритми*, ТроTeam Co., София, 2002.

Каквато и хеш-функция да изберем, в най-лошия случай тя ще предизвика ***n*** колизии, където ***n*** е размерът на хеш-таблицата. В най-добрия случай ще възникнат 0 колизии. За да се предпазим от колизии, трябва да гарантираме равномерно разпределение на множеството от допустимите стойности на ключовете върху множеството от индексите на масива/таблицата. Тогава вероятността за съпоставяне на един и същи индекс в масива за два различни ключа, т. е. за възникване на колизия, ще бъде 1/ ***n*** .

Една от възможностите е да се комбинират няколко хеш-функции, които разпределят елементите равномерно.

И все пак, как да се справим с колизиите, когато те възникват?

Справяне с колизии

Нека е дадена хеш-таблица с размер ***n***=10.
Ще използваме проста хеш-функция, която се основава на извличане на цифри - първата цифра на ключа ще дава хеш-адреса на елемента. Например, искаме да включим в таблицата елементите 234, 235, 567, 123, 534 и 647.

Всички методи условно могат да бъдат разделени на две групи:

- затворено хеширане
- отворено хеширане

Затворено хеширане

1. Линейно пробване

При този подход разполагаме с едно основно място, където се съхраняват данните (***n*** слота). Когато настъпи колизия, пробвам последователно да променяме получения хеш-адрес, докато стигнем до свободен слот (вход в таблицата). Промяната се прави по предварително дефинирана схема.

Например, възможно е да увеличаваме адреса с някакво естествено число ***s*** $(0 < s < n)$. Ако адресът стане равен на ***n***, ще продължим търсенето в началото на хеш-таблицата, като вземаме остатъка по модул ***n***. За да може при подобно увеличаване да се обходят всички адреси на хеш-таблицата, трябва ***n*** и ***s*** да бъдат взаимно прости, т. е. НГОД(***n*** , ***s***)=1.

Очевидно, при този подход не могат да се включват повече елементи, колкото е капацитетът на хеш-таблицата. И когато тя се препълни, е необходимо разширяване (заделяне на памет за по-голям масив и освобождаване на заетата от текущия масив памет). На практика запълването на таблицата при този вид хеширане не трябва да превишава 1/2.

Последователно включване на елементи с разрешаване на колизиите чрез линейно пробване със стъпка 1:

		234	235	567	534	123	647
0							
1						123	123
2		234	234	234	234	234	234
3			235	235	235	235	235
4							
5				567	567	567	567
6					534	534	534
7							647
8							
9							

При търсенето на елемент по ключ се процедира по аналогичен начин - първо се изчислява хеш-адресът с помощта на хеш-функцията и в случай, че не се попадне на търсения ключ, се прави линейно пробване.

Линейното пробване се състои в променяне на адреса с 1, докато се стигне до търсеният елемент (като се предполага, че ключът е в таблицата).

В този случай можем да попаднем на ключове, при които хеширането да се окаже неефективно. Например, когато елементите образуват групи с близки хеш-кодове (т. н. клъстери). Тогава в процеса на разрешаване на колизия възникват серия нови (вторични) колизии.

2. Квадратично пробване

При квадратичното пробване се използва стъпка от вида

$$c_1 i + c_2 i^2 \text{ при } c_2 <> 0$$

Отместването зависи квадратично от поредния номер на пробата i. Този метод работи значително по-ефективно от линейното пробване, въпреки че и при него може да възникне вторично струпване на ключове (клъстери). Основният му проблем е, че не гарантира обхождане на цялата таблица.

3. Двойно хеширане

Двойното хеширане е подобно на вече разгледаните методи, но използва две хеш-функции:

$$h(k, i)= h_1(k) + i*h_2(k)$$

Тук ***i*** е номерът на пробата след първата колизия. Втората хеш-функция се използва само в случаите, когато резултатът от първата води до колизия. Като ключ при повторно хеширане може да се използва както оригиналния ключ на елемента, така и хеш-адресът, получен при прилагане на първата хеш-функция.

4. Рехеширане

Прилага се при голямо запълване на таблицата (над 1/2). Създава се нова хеш-таблица с двойно по-голям размер (обикновено се взема първото просто число, по-голямо от 2n). В новата таблица се прехвърлят ключовете от първата.

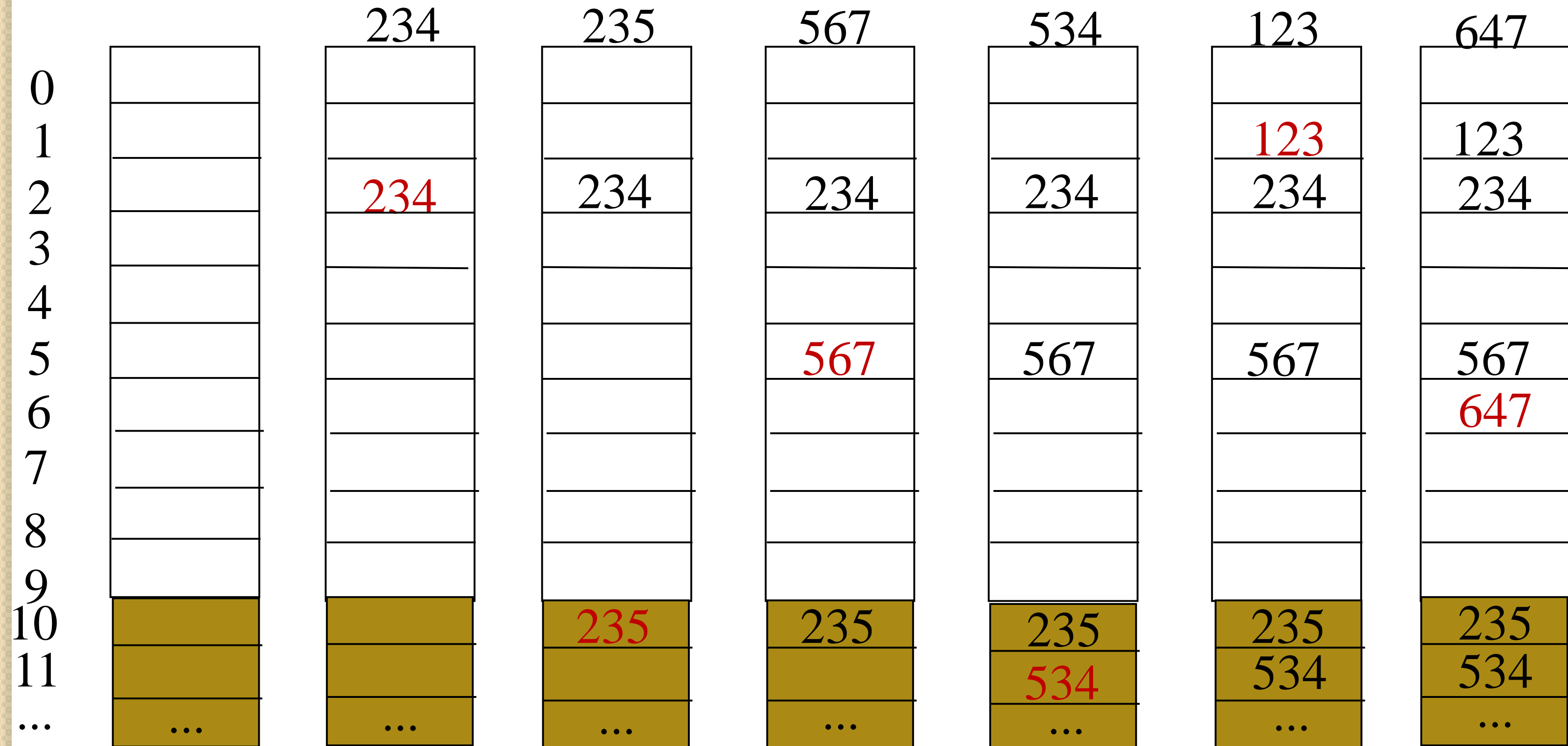
5. Разширено хеширане

Използва се тогава, когато хеш-таблицата не може да се помести в оперативната памет. В този случай таблицата се зарежда в ОП на част (по блокове). Хеширането трябва да е такова, че да има минимален среден брой обръщения към външна памет при търсене на ключ в таблицата.

<u>Отворено хеширане</u>

1. Допълнителна памет за колизии

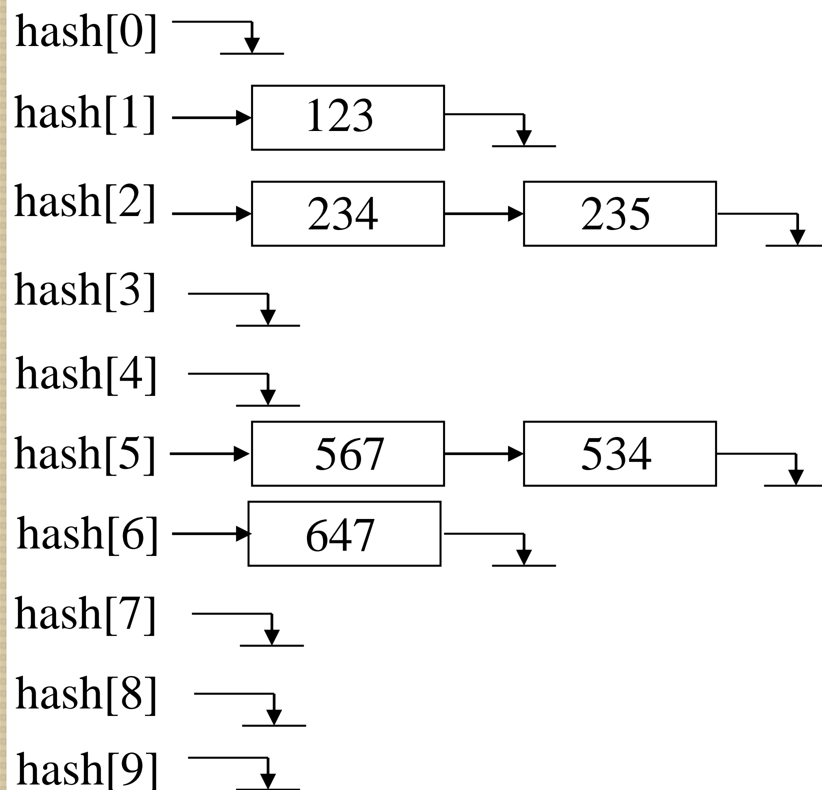
При тази схема се отделя допълнителна памет за обработване на колизии. Всеки елемент, попаднал в колизия, се разполага на първото свободно място в допълнително отделената памет:



При търсене в хеш-таблицата първо се проверява адресът, получен чрез хеш-функцията. Ако елементът с търсения ключ не е там, се претърсва цялата допълнителна памет.

2. Списък на препълванията

Възможно е да се използва и динамично заделяна памет (*heap*). Елементите със съвпадащи хеш-адреси се разполагат в списъци на препълванията. Всеки слот на хеш-таблицата е указател към динамичен списък, съдържащ само членове на съответния клас синоними. Самата хеш-таблица е масив от указатели. При включване на нов елемент той се добавя в началото на списъка, намиращ се на слота, определен от хеш-функцията. Проблемът с колизии отпада.



При търсене се преглежда списъкът, определен от хеш-функцията. Сложността на операцията търсене зависи от броя на елементите, с които дадения е в колизия, т. е. от дължината на съответния списък.

Елементите могат да се пазят и в друга динамична структура, например - в BST.

Пример за създаване хеш-таблица и търсене в нея. Данните в таблицата са имена и факултетни номера на студенти от една студентска група. Ключовите стойности на елементите са факултетните номера. Възможните колизии се отстраняват чрез използване на списъците на препълванията.

```
#include <iostream>
using namespace std;
const int m=31;                     //размер на хеш-таблицата
                                    //(студентската група)
struct ptr
{char name[10]; int fn; ptr *next;} *hash[m]; //структура
                                              //на данните
//hash[m] - хеш-таблица, реализирана като масив от
//указатели,сочещи към списъците на препълванията
void init()          //инициализация на таблицата
{
  for (int i=0; i<m; i++)
   hash[i]=NULL;
}
```

```
void create()          //създаване на хеш-таблицата
{
  int f, h;
 // f - текущ Ф.№, h - текуща стойност на хеш-функцията
  cout<<"\nЗадайте Ф.№: ";
  cin>>f;
  h=f%m;
  ptr *q=hash[h];     //работен указател
  ptr *s=new ptr;     //работен указател
  s->fn=f;
  cout<<"\nВъведете име: ";
  cin>>s->name;
  s->next=NULL;
  if (!q)                //няма колизия, списъкът е празен
       hash[h]=s;
    else                 //списъкът не е празен
     { while (q->next)  q=q->next;
       q->next=s;
     }
}
```

```
void seek()          //търсене в хеш-таблицата
{
  int fl=1;           //флаг
  int f, h;
  cout<<"\nЗадайте търсения Ф.№:";
  cin>>f;
  h=f % m;            //изчисляване на хеш-функцията
  ptr *q=hash[h];
  while (q&&fl)
     if (q->fn==f)
       {fl=0; cout<<q->name<<"  "<<q->fn;}
      else q=q->next;
  if (fl)
     cout<<"\nВ групата няма студент с Ф.№ "<<f<<"!";
}
```

```
void main()
{init();                  //инициализация
 char c;                  //работна променлива
 do
 {
   create();  //добавяне на елемент в таблицата
   cout<<"Ще добавяте ли друг студент (Y/N):";
   cin>>c;
 }
 while (c!='N'&&c!='n');
 do
 {
   seek();
   cout<<"\nИскате ли да потърсете друг студент (Y/N):";
   cin>>c;
 }
 while (c!='N'&&c!='n');
}
```

Заключение:

Съотношението между броя ключовете, записани в хеш-таблицата, и нейния размер се нарича *фактор на запълване* (или на *натоварване*).

При отвореното хеширане факторът трябва да има стойност, близка до 1 (може и по-голяма от 1).

При затвореното хеширане факторът на запълване не трябва да надвишава 1/2.

Къде се прилага хеширането?

Навсякъде, където се извършва търсене и запис на данни, когато е от значение времето, необходимо за реализация на тези две операции.

Например:

- при компилаторите (в т.н. символна таблица, за да се съхраняват в нея идентификатори в изходния код на компилираната програма;
- в програмите за игри - в т.н. транспозиционни таблици;
- в програмите за проверка на правописа - при търсене на думите в речник