# ХЕШ-ТАБЛИЦИ И ХЕШИРАНЕ

### judge.openfmi.net/mediawiki, Програми=++Алгоритми, Галина Момчева, pomagalo.com, referati.com и др.

## 1. Хеш-таблица

### Хеш таблицата е структура от данни, чийто елементи се състоят от *ключ* и *данни*. Ключът е уникален за разлика от данните – не може да съществуват два елемента с един и същи ключ. Ключът определя мястото на елемента в хеш таблицата. Хеш таблицата се характеризира с директен достъп до елементите. Броят на елементите в таблицата се нарича капацитет на таблицата. Когато полето данни отсъства, т.е. ключа и данните са едно и също, структурата се нарича хеш-множество.

Като частен случай на хеш-таблицата може да се разглежда n-елементен масив, в който индексът i (i=0,1,…..,n-1) е ключ на i- тия елемент. На практика при реализацията на хеш-таблица често се използва масив.

### Основните операции за работа хеш-таблица са:

* търсене на конкретен елемент в хеш-таблицата
* включване на елемент в хеш-таблицата
* премахване на елемент от хеш-таблицата

Сложността на елементарните операции по ключ в общия случай е константна, което я прави изключително полезна.

### 

### 2. Хеширане. Xeш-функция

*Хеширане* – процесът, при който по зададен ключ на елемент, посредством константни преобразования, се определя неговият адрес в хеш таблицата. Хеширането е необратим процес, т.е. ако вие хеширате даден низ, то от резултата няма как да получите обратно първоначалния низ. С други думи, хеширането е процес, при който на дадени данни (ключът) се съпоставят други, посредством правила, но така че възстановяването им в първоначален вид е невъзможно. Точно затова бихте могли да съхраните паролите на 1000 човека, но няма да узнаете самите пароли.

За хеширане се използва функция, която на всеки от възможните ключове съпоставя еднозначно валиден адрес. Самата функция се нарича *хеш-функция*.

Например, нека представянето на хеш-таблицата ни е n-елементен масив. При включването на елемент той трябва да се разположи на валиден адрес – между 0 и n-1. Една възможна функция, в случай че ключа е число, е остатък при деление на n. Така си гарантираме, че полученият резултат винаги е валиден.

Не трябва да се бърка криптиране с хеширане, тъй като криптирането е двупосочен процес – предполага се, че ако се разполага с нужните ключове, обратната операция е възможна. Това означава, че използвайки ключа, от криптираното съобщение можем да получим изходното съобщение. В случая със хеш функциите, това въобще не е така – хеширането е еднопосочно, необратимо преобразование.

**3. Колизии**

Ако два различни елемента след хеширането с дадена хеш-функция получат едни и същи хеш стойности, то те ще бъдат разпределени в еднакъв адрес. В такъв случай казваме, че е налице *колизия*. Елементите, които участват в колизията, наричаме *синоними*.

Дори да се състави перфектна хеш-функция, колизия ще настъпи в момента, в който броят на включените елементите надхвърли капацитета на хеш-таблицата. Когато ключовете на елементите станат повече от размера на таблицата, при опит да се добави нов елемент функцията ще разпредели еднакъв адрес за елементи с различни ключове и така ще настъпи колизия.

Всъщност колизии настъпват много по-рано от запълването на адресите.

Например, ако хеш-функцията взема остатъка при деление на 17, тогава 18 и 35 ще имат един и същи хеш стойности, тъй като 18%17=1 и 35%17=1 и ще настъпи колизия.

Също, ако хеш стойността се получава като се сумират ASCII кодовете на всички символи на стринга, то два стринга с едни и същи знаци, подредени по различен начин, също ще имат едни и същи хеш стойности и ще настъпи колизия.

Колизиите са основния проблем пред хеш-таблиците. За съжаление, няма как да не се получават колизии, дори хеш-функцията да разпределя постъпващите елементи достатъчно равномерно по адресите. Така че освен избора на подходяща функция е важно да изберем и подходящ алгоритъм за справяне с колизиите.

## 4. Класически хеш-функции

Хеш-функцията трябва да бъде избрана така, че да не отнема много изчислително време, като същевременно разпределя елементите достатъчно равномерно. Може да приемем, че ключовете ни винаги са число. В случай че не са, можем да им съпоставяме число по някакво определено правило. Например, ако са стрингове можем да ги преобразуваме до числа като сумираме кодовете на символите им (за hello получаваме 104 + 101 + 108 + 108 + 111 + 111 = 532), или като умножим всеки код на символ по позицията му и след това ги съберем и приемем, че низът е число в p-ична бройна система, където p е броя символи допустими в низа, и др.

**а/ класически хеш-функции върху числа**

### остатък при деление

Може да се каже, че това е най-простият и сравнително ефективен метод за хеширане. Взима се остатъка при целочисленото деление на ключа с размера на таблицата.

hash(key) = key % n

Например, за числото 25 при размер n = 10 се получава hash(25) = 25%10=5.

При този алгоритъм не е удачно n да е степен на двойката, тъй като в такъв случай хеш кода на даден ключ, ще бъдат младшите му  битове, а алгоритъмът се държи лошо при голям брой съвпадащи младши битове.

Например, ако n = 16, hash(173)=173%16=13, което е 1101(2) и представлява младшите битове на 173, което е 10101101(2).

Затова най-често за размер на таблицата се избира просто число.

### мултипликативно хеширане

При мултипликативното хеширане избираме реална константа a между 0 и 1 и образуваме функцията:

hash (key) = [n.{key.a}], където {key.a} е дробната част на произведението на key и а, [] означават цялата част на полученото число

За да я пресметнем, най-напред намираме дробната част от произведението key\*a и я умножаваме по размера на таблицата, след това вземаме цялата част на получения резултат. А самата дробна част се намира като от числото извадим цялата му част: {key.a} = key.a – [key.a]. Изразено това на езика С++ има следния вид:

hash(key) = floor(n \* ((key \* a) - floor(key \* a)))

Въпреки че изборът на константата а е произволен, при някой стойности алгоритъмът се държи по-добре. Кнут (1968) препоръчва да се използва златното сечение : а = = 0.6180339887…

### хеш функции върху части от ключа

Такива функции обикновено се използват когато ключът може да е много голямо число.

При някои от тях се обработва само част от ключа - използват се само цифрите стоящи на определени позиции **/извличане на цифри/**. Например за ключовете 123569, 425435, 546754, 676576 ще вземем цифрите, стоящи на 1,3 и 5-то място и ще получим съответно 136, 453, 565, 667. В случая n = 1000 (тъй като при извличане на 3 цифри можем да получим число в интервала [0…999]). Методът има добра успеваемост, когато числата не съдържат много повтарящи се цифри.

Други използват т.нар. **сгъване** – разделяме ключа на части и прилагаме някаква аритметична операция върху тях. Например числото 123569425435 можем да разделим на 2 чести, т.е. на 123569 и 425435, да извлечем цифрите от 1, 3 и 5-то място, т.е. 136 и 453 и да ги съберем. Окончателно получаваме 589.

При трети вид се прилага **повдигане на средата в квадрат** – извличате средните р цифри и ги повдигате в квадрат. Например числото 12567134280980 има среда 134, т.е. кода е 134.134 = 17956. Ако резултатът надхвърля n,  се премахват първите няколко значещи цифри. Т.е. ако n = 10001, то от 17956 се премахва първата цифра и резултатът е 7956. /последната операция не е еквивалентна на намиране на остатък при делене/.

**б/ класически хеш функции върху низове**

Низовете са едни от най-използваните ключове при хеширането. Разбира се, можем да представим всеки символен низ като числен и да приложим изброените вече хеш-функции за числа, но има и специални хеш функции за символни низове. Общата схема по която работят хеш-функциите върху низове е:

hash( key )

{

резултат = начална стойност

за всеки символ c от key

резултат = комбиниране( резултат, c )

резултат = модифициране( резултат )

резултат = допълнително\_модифициране( резултат )

}

### адитивно хеширане

Това е най-простата функция за хеширане на стрингове, но и най-неефективна. При него сумираме ASCII кодовете на символите, участващи в низа, и вземаме остатъка от делението на сумата с размера на таблицата.

unsigned long hash (const char \*key, unsigned long size)

{

unsigned long result = 0;

while (\*key)

result += (unsigned char) \*key++;

return result % size;  
}

Обикновено дължината на низа се включва в сумата, за да може изрично да влияе на хеш кода, затова и първия ред във функцията се заменя с: unsigned long result = strlen(key);

Проблем при това хеширане е, че редът в който са символите не е от значение, което означава, че пермутациите на един низ ще имат еднакви xeш-кодове. Това може лесно да се избегне например като умножим кода на символа по позицията му или на всяка стъпка удвояваме резултата преди да го комбинираме със символа.

### ротирано хеширане

Тук няма събиране, а само поразредни операции. Размерът на масива трябва да е просто число.

unsigned long hash (const char \*key, unsigned long size)

{

unsigned long result = strlen(key);

while (\*key)

result = (result<<4)^(result>>8)^((unsigned char) \*key++);

return result % size;  
}

### хеширане едно по едно

### Аналогично на ротираното, но манипулациите върху променливата result са повече и са извършвани поотделно

unsigned long hash (const char \*key, unsigned long size)

{

unsigned long result = 0;

while (\*key)

{

result += (unsigned char) \*key++;

result += result<<10;

result ^= result>>6;

}

result += result<<3;

result ^= result>>11;

result += result<<15;

return result % size;  
}

### хеширане по Пиърсън

Въвеждаме допълнителен масив tab[], съдържащ пермутация на числата от 0 до 255. Полученият хеш код е еднобайтов: от 0 до 255. За да бъдат повече от еднобайтови резултатите, може да извикате алгоритъма няколко пъти с различни масиви tab[], т.е. различни наредби на числата, при което всяко извикване ще дава един байт от резултата. Ето кода:

unsigned char hash (const char \*key, unsigned long size, const unsigned char tab[])

{

unsigned long result = strlen(key);

while (\*key)

   result = tab[ result ^ ( (unsigned char) \* key++) ];

return result;

}

## 5. Справяне с колизии

За да намалим вероятността от настъпване на колизии, трябва да не се допуска напълване на хеш таблицата повече от половината, т.е. ако искаме да работим с максимум 1000 елемента, трябва да изберем хеш таблица с поне 2000 елемента. Това не значи, че в такава таблица няма да има колизии! Ще има. Просто достатъчният размер на таблицата ще държи колизиите в допустими граници, които не се отразяват на скоростта на алгоритъма.

Когато колизиите все пак настъпят, трябва да се търси алгоритъм за решаване на проблема. Има няколко начина да се справим с колизиите. Всички те имат своите предимства и недостатъци и най-общо трябва да се избира в зависимост от конкретната задача. Ето няколко такива:

### 

### а/ затворено хеширане

При затвореното хеширане разполагаме с само с наличната хеш таблица и не се предвижда допълнително място в паметта за справяне с колизиите. В случай на колизия, т.е. на мястото където искаме да запишем елемента има вече друг елемент със същият хеш, се правят последователни проби за промяна на хеш-адреса до достигане на свободен адрес. Промяната се прави по предварително дефинирана схема:

- линейно пробване – за намиране на нов свободен адрес използваме стъпка s, с която увеличаваме адреса докато намерим свободен. При опит да увеличим адреса и той надхвърли капацитета на таблицата взимаме остатъка му при целочислено деление с размера на таблицата. Ако s и n са взаимно прости си гарантираме, че можем да обходим цялата таблица. Това е важно, тъй като от своя страна гарантира, че винаги когато има свободен адрес в таблицата ще можем да го използваме при необходимост. При този подход не могат да се включат повече елементи, отколкото 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).



**Пример:** дадени са ключове 234, 235, 567, 123, 534, 647 и проста хеш-функция – извличане на цифра(първата). При колизия ще използваме линейно пробване със стъпка 1.

hash(234) = 2 – запълваме позиция 2

hash(235) = 2 – позиция 2 е заета, пробваме линейно със стъпка 1 и намираме свободна позиция с индекс 3

hash(567) = 5 – запълваме позиция 5

hash(123) = 1 – запълваме позиция 1

hash(534) = 2 – позиция 5 е заета, пробваме линейно със стъпка 1 и намираме свободна позиция 6

hash(647) = 6 – позиция 6 е заета, пробваме линейно със стъпка 1 и намираме свободна позиция 7

Търсенето на елемент се осъществя по същата схема – прилага се хеш-функцията и ако ключът намиращ се на получения адрес не съвпада с търсения се прилага горната схема докато се той се намери или се попадне на незает адрес. Последното би означавало, че елемент с заявения ключ не е в таблицата. При наличие на колизия този алгоритъм се превръща в обикновено сканиране и това може драстично да свали скоростта на целият алгоритъм.

- квадратично пробване – аналогично на предходното, но се използва стъпка s = a\*i + b\*i2, b != 0, която зависи квадратично от поредния номер на пробата. Този по-ефективен от линейното пробване, но има недостатъка, че не винаги гарантира обхождане на цялата таблица, т.е може да не намери свободно място, макар да има такова.

- двойно хеширане – подобно е на предходните два метода, като тук се използват две хеш-функции:

h(key, i) = h1(key) + i\*h2(key), където i е номерът на пробата след първата колизия.

Втората хеш-функция се използва само когато вжрезултатът от първата хеш-функция е настъпила колизия. Като ключ при повторното хеширане може да се използва както оригиналният ключ на елемента, така и хеш-адреса, получен при прилагане на първата хеш-функция.

### б/ отворено хеширане

### При този метод има допълнително заделена памет, която се използва за справяне с колизиите.



#### Допълнителна памет за колизии

При този метод допълнителната памет е разположена след адресируемата част на таблицата. При колизия елементът се добавя на първото свободно място в частта за препълвания.

При търсене, ако ключа на елемента разположен на адреса, получен след хеширане, не съвпада с търсения ключ, се проверят елементите от допълнителната част. При тях няма никаква наредба, което е неефективно, тъй като налага да се проверят ключовете на всички елементи, попаднали в колизия до момента.

#### Списък на препълванията

При този метод се използва външна динамично заделяна памет. Елементите със съвпадащи хеш-адреси се разполагат в някаква стандартна структура от данни, най-често свързан списък. Всеки елемент на хеш-таблицата е указател към динамичен свързан списък, съдържащ само елементите, които имат еднакъв хеш-адрес, т.е. хеш-таблицата не съдържа хешираните елементи, а указатели към свързани списъци от елементи с равен хеш-код.



При постъпване на нов елемент, той се добавя в началото на списъка Така могат да запишат неограничено количество елементи в таблицата, без да настъпи колизия.

При търсене се преглежда списъка, определен от хеш-функцията. Сложността на операцията зависи от дължината на този списък и от метода на търсене – дали ще го сканираме елемент по елемент или по някакъв друг начин - например с двоично търсене.

На фигурата вдясно е илюстриран списък с препълвания при ключове 234, 235, 567, 123, 534, 647 и хеш-функция – извличане на първата цифра.

Въпреки че най-често употребяваната структура е списък, този вид хеширане може да се реализира чрез двоично дърво за търсене, т.е.елементите в колизия се пазят в подредено двоично дърво,където търсенето е по-бързо. Тогава всяко поле от хеш-таблицата ще бъде указател към корена на отделно двоично дърво.

## 6. Приложения

**а/ за реализация на асоциативни масиви**

Повечето съвременни компилатори поддържат хеш-таблици за реализация на асоциативни масиви /познати още като асоциативни списъци, map, dictionary/. Асоциативният списък е абстрактен тип данни, съставен от двойка (ключ, стойност), като ключът е уникален. Асоциативните масиви се характеризират с това, че индексите им може да не са числа. За целта компилатора използва подходящи хеш-функции за различните типове данни използвани за индексиране.

**б/ пароли, идентификация**

Другата голяма област на приложение на хеш функциите е идентификацията на потребители и разпознаване на пароли. Трябва да пазим имената на потребителите и паролите и да ги сравняваме с тези, които потребителят въвежда. Но ако пазим имената и паролите във файл, всеки който има достъп до файла с паролите, ще може да ги види и използва. Криптирането също не е решение, защото всеки криптиращ алгоритъм може да се декриптира.

Решението е да хешираме потребителските пароли и да пазим хеш стойността, вместо самата парола. Когато потребителят въведе име и парола, тази парола също се хешира и нейната хеш стойност се търси в базата данни. Дори и някой да има достъп до базата данни, той няма да може да разбере какви са паролите на потребителите, защото хеш функцията не може да се инвертира.

**в/ търсене на стрингове в големи масиви**

Друго приложение на хеш функциите е за търсене на стрингове в големи масиви. Сравняването на стрингове е много скъпа операция, която вътрешно се реализира със цикли, и сравняване символ по символ на двата стринга, а сравняването на числа е евтина операция, която се реализира с 1, 2 инструкции на процесора. Затова при въвеждане на низа изчисляваме неговата хеш-стойност и я запазваме. След това при търсенето на даден низ, изчисляваме неговата хеш стойност и обхождаме масива, сравнявайки хеш стойностите. Трябва да предвидим колизиите - ако хешовете съвпадат, трябва да направим задължително сравнение директно между стринговете, за да сме сигурни че търсеният стринг е намерен - но тук имаме само едно сравнение, а не хиляди.

Това решение е много по-ефективно, но все пак остава сканиране на масива, макар и сравнявайки числа, а не стрингове. Скоростта на алгоритъма е O(n) - тоест времето нараства линейно с увеличаване на размера на масива. Можем да направим търсенето още по-бързо, като премахнем обхождането на масива и така скоростта няма да зависи от размера на масива, а ще бъде константа. Това става чрез използване на допълнителен масив за търсене, който ще бъде нашата хеш-таблица. Размерът му трябва да е съобразен с броя на различните хеш-стойности.При въвеждането на всеки стринг изчисляваме хеш-стойността и в съответният елемент от хеш таблицата записваме указател към стринга. След това при търсене на даден стринг ще е достатъчно да изчислим хеш стойността му и да видим, дали на съответната позиция има 0 или указател.

**г/ изчисляване на контролната сума**

Cyclic redundancy check (CRC) е вид хеш функция, която се използва за изчисляване на контролната сума (checksum) на блок от данни, като например пакет от мрежовия трафик. Контролната сума има фиксиран брой битове. След пресмятането й, тя се добавя към информацията преди да се съхрани или изпрати. Получателят сам пресмята CRC на данните, които е получил и сравнява получената контролна сума с тази в пакета. Ако съобщението се е променило по време на предаването, то двете хеш стойности няма да съвпаднат и пакетът се отхвърля.

**Примерна реализация**

По-долу е дадена примерна реализация на основните операции върху хеш таблица с отворено хеширане (списък на препълванията) на C++. Използваме наготово функции за работа със списъци, за да не утежняваме кода с тяхната реализация.

void OpenHashTable::Put(HashTable::KeyType key, HashTable::DataType data)

{

HashedKeyType i = HashFunction(key);

list \*l = listFind(mOpenTable[i], key);

if (l) l->data = data;

else

{

if (mOpenTable[i])

++mCollisionsCount;

listInsert(&mOpenTable[i], key, data);

}

}

HashTable::DataType OpenHashTable::Get(HashTable::KeyType key)

{

HashedKeyType i = HashFunction(key);

list \*l = listFind(mOpenTable[i], key);

return l ? l->data : NOT\_FOUND;

}

void OpenHashTable::Remove(HashTable::KeyType key)

{

HashedKeyType i = HashFunction(key);

if (mOpenTable[i])

listDelete(&mOpenTable[i], key);

}

*Забележка:* При операцията put се прави проверка дали елемент с дадения ключ вече съществува и ако е така се променя полето му с данни. По този начин се имитира поведението на асоциативен масив. В зависимост от търсения ефект може да се подходи и по друг начин - да се изведе съобщение за грешка или въобще да не се прави проверка. Ако се добави елемента без предварителна проверка може да се наруши свойството на хеш-таблицата на един ключ да отговаря единствена стойност (данни). В такъв случай определянето на стойност по ключ е нееднозначно и трябва да зависи и от някакво допълнително правило - например данните на последният записан елемент да се считат за достоверни.

**Пример 1. 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 списъци на препълвания.

#include <iostream>

using namespace std;

const int m=30; //размер на хеш-таблицата (брой студенти )

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()

{

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<<"Fak. Nr.: ";

cin>>f;

h=f%m;

ptr \*q=hash[h]; //работен указател

ptr \*s=new ptr; //работен указател

s->fn=f;

cout<<"Name: ";

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<<"Fak. Nr.:";

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<<endl;}

else q=q->next;

if (fl)

cout<<"No student with Fek. Nr "<<f<<"!"<<endl;

}

int main()

{

init();

char c; //работна променлива

do

{

create(); //добавяне на елемент в таблицата

cout<<"Enter another student (Y/N):";

cin>>c;

}

while (c!='N'&&c!='n');

do

{

seek();

cout<<"Search another student (Y/N):";

cin>>c;

}

while (c!='N'&&c!='n');

return 0;

}

Пример 2.

В първата имплементация ще разглеждаме целочислен ключ (това е важно при реализацията на хеш-функцията — тя ще бъде на принципа на деление с остатък. С колизиите ще се справяме, като използваме динамичен свързан списък). С този пример ще илюстрираме как бързо и лесно може да се построи ефективна структура от данни (обърнете внимание, че като изключим имплементацията на свързан списък, допълнителният за хеш-таблицата код е съвсем кратък).

1 начин: *hash.cpp*

#include "stdio.h"

#define n 211

typedef int data;

typedef long int keyType;

typedef struct listnode {

keyType key;

data info;

listnode \*next;

} tlist;

typedef tlist \*list;

list hashTable[n];

/\* включва елемент в началото на свързан списък \*/

void insertBegin(list &L, keyType key, data x){

list temp;

if (!(temp = new listnode)) {

printf("Недостатъчно памет\n");

return;

}

temp->next = L;

L=temp;

L->key = key;

L->info=x;

}

/\* изключва елемент от свързан списък \*/

void deleteNode(list &L, keyType key) {

list current = L;

if (L->key == key) { // изтриване на първия елемент

current = L->next;

delete L;

L=current;

return;

}

// намира елемента, който ще се изтрива

while (current->next->key != key && current->next!=NULL) {

current = current->next;

}

if (current->next == NULL) {

printf("Елементът, който ще се трие не еч в хеш-таблицата!\n");

return;

} else {

list save = current->next;

current->next = current->next->next;

delete save;

}

}

/\* претърсва свързан списък \*/

list searchList(list L, keyType key){

while (L != NULL) {

if (L->key == key) return L;

L=L->next;

}

return NULL;

}

int hashFunction(long int key) {

return (key % n);

}

void initHashTable() {

for (int i=0; i<n; i++) hashTable[i]=NULL;

}

void put(keyType key, data x) {

int place = hashFunction(key);

insertBegin(hashTable[place], key, x);

}

data get(keyType key) {

int place = hashFunction(key);

list l = searchList(hashTable[place], key);

return l->info;

}

void main() {

initHashTable();

put(1234, 100); // -> on slot 34

put(1774, 120); // -> on slot 74

put(34, 180); // -> on slot 24 -> collision

printf("Get data for key 34 : %d \n",get(34));

printf("Get data for key 1234 : %d \n",get(1234));

}

2 начин:

#include <iostream>

using namespace std;

const int n=100;

typedef struct listnode {

int key;

int info;

listnode \*next;

} tlist;

typedef tlist \*list;

list hashTable[n];

/\* включва елемент в началото на свързан списък \*/

void insertBegin(list &L, int key, int x){

list temp = new listnode;

temp->next = L;

L=temp;

L->key = key;

L->info=x;

}

// претърсва свързан списък

list searchList(list L, int key){

while (L != NULL) {

if (L->key == key) return L;

L=L->next;

}

return NULL;

}

void put(int key, int x) {

int place = key % n;

insertBegin(hashTable[place], key, x);

}

int get(int key) {

int place = key % n;

list l = searchList(hashTable[place], key);

return l->info;

}

void main() {

for (int i=0; i<n; i++) hashTable[i]=NULL;

put(1234, 100); // -> on slot 34

put(1774, 120); // -> on slot 74

put(34, 180); // -> on slot 24 -> collision

printf("Get data for key 34 : %d \n",get(34));

printf("Get data for key 1234 : %d \n",get(1234));

}

**Пример 3:**

***Задача***: Даден е списък от думи (например зададени в текстов файл, разделени с един интервал). Да се намери броя на срещанията на всяка дума.

Описаната задача възниква често като част от по-сложни практически проблеми. Така например, индексирането на документ (за търсеща машина, преди каквато и да е по-нататъшна обработка) се състои именно в намиране на броя на срещанията на всяка дума от документа.

За да бъде решена задачата ефективно (разбира се, като използваме хеш-таблица) трябва да се обърне внимание на няколко основни неща:

* Ключът на хеш-таблицата ще бъде символен низ — всяка дума от документа. Допълнителна информация (допълните данни) ще бъде броят на срещанията на думата. В някои задачи е възможно да се разглежда дори ситуация, при която няма никакви допълнителни данни.
* Колизиите ще решаваме, като използваме линейно пробване. Ще използваме следния подход: Големината на хеш таблицата *n* ще бъде число, степен на 2. Стъпката *s*, с която ще извършваме пробването, ще бъде просто число (така гарантираме, че е изпълнено свойството *n* и *s* да бъдат взаимно прости).
* Ако предварително определената големина за хеш-таблицата се окаже недостатъчна, ще разширяваме таблицата, чрез заделяне на памет за още елементи (ще удвояваме големината й, за да запазим това число да е степен на 2-ката).

*hash2.cpp*

#include <stdio.h>

#include <string.h>

#define s 107 // The word hash table step in case of collision

long n = 32; // initial hash size, we dont use define because of resize

struct singleWord {

char \*word; // The word

unsigned long freq; // Global word occurrence frequency

} \*ht; // Words hash table

unsigned long cnt;

unsigned long hashFuntion(const char \*key) {

unsigned long result = 0;

while (\*key) {

result += result + (unsigned char) \*key++;

}

return result & (n-1);

}

void initHashtable() {

cnt = 0;

ht = new singleWord[n];

for (unsigned long i = 0; i<n; i++) {

ht[i].word = NULL;

}

}

// Look up a string in the hash table - 0 if not found, 1 if found

char get(const char \*key, unsigned long &ind) {

unsigned long k = ind = hashFuntion(key);

do {

if (ht[ind].word == NULL) return 0; // The string is missing

if (strcmp(key, ht[ind].word) == 0) return 1; // The string has been found

ind = (ind + s) & (n-1);

} while (ind != k);

return 0;

}

void resize() {

unsigned long ind, hashInd;

singleWord \*oldHashTable;

// 1. Save a pointer to the original hash table

oldHashTable = ht;

// 2. Resize the hash table: The other things are updated automatically

n <<= 1;

// 3. Allocate memory for the new table

ht = new singleWord [n];

for (ind = 0; ind < n; ind++)

ht[ind].word = NULL;

// 4. Move the records into the new hash table (using the new hash function)

for (ind = 0; ind < (n >> 1); ind++)

if (oldHashTable[ind].word != NULL) {

// Move the record to the new location

if (!get(oldHashTable[ind].word, hashInd))

ht[hashInd] = oldHashTable[ind];

else

printf("ERROR while resizing the hash table!\n");

}

// 5. Switch the pointers

delete [] oldHashTable;

}

// Add a new string to the hash table

void put(char \*key) {

unsigned long ind;

if (!get(key, ind)) { // The string has not been found

ht[ind].word = new char[strlen(key)+1];

strcpy(ht[ind].word, key);

ht[ind].freq = 1;

if (++cnt > ((unsigned long) n\*0.9))

resize();

} else ht[ind].freq++;

}

void printAll() {

for (unsigned long ind = 0; ind < n; ind++)

if (ht[ind].word != NULL)

printf("%s %ld \n", ht[ind].word, ht[ind].freq);

}

void destroy() {

for (unsigned long ind = 0; ind < n; ind++)

if (ht[ind].word != NULL)

delete [] ht[ind].word;

delete [] ht;

}

inline void UpCase(char &c) { if (c >= 'a' && c <= 'z') c -= 'a' - 'A'; }

int main() {

initHashtable();

put("reload");

put("crush tour");

put("room service");

put("load");

put("reload");

put("reload");

printAll();

unsigned long find;

printf("Lookup 'reload': %d \n", (int)get("reload", find));

printf("Lookup 'download': %d\n", (int)get("download", find));

}

*Задача за упражнение*: Реализирайте функция за изключване на елемент от хеш-таблица, по даден ключ. Сложността трябва да бъде константа. Обърнете внимание, че не е достатъчно единствено да намерим елемента със зададения ключ и да го изтрием (както при реализацията със списък на препълване). Последното ще се наруши последователността от елементите (имащи същият хеш-код) след този, който изключваме.

**Пример 4:** В текстов файл са зададени следните данни – име и телефон на 7 души. Да се напише програма, която илюстрира използването на хеш-таблици.

Petrov 9123456

Todorov 9943937

Kolev 9121212

Penev 9929292

Lekov 9767688

Dineva 9346523

Tonova 9999222

Решение: Програмата преобразува данните в хеш-таблица с K елемента. Ако K<7, непременно ще има колизии. Ако K≥7, тогава отново може да има колизии. Хеш-функцията ще дефинираме като public член-функция на класа HashTable с ключ от тип const char\*, задаващ името на притежателя на телефон, по следния начин:

unsigned HashTable::hash(const char\* str) const

{unsigned sum = 0;

for(int i = 0; str[i]; i++)

sum +=(i+1)\*str[i];

return sum % K;

}

При стойност 5 за K, получаваме:

hash(“Petrov”) = (1\*’P’ + 2\*’e’ + 3\*’t’ + 4\*’r’ + 5\*’o’ + 6\*’v’) % 5

= (1\*80 + 2\*101 + 3\*116 + 4\*114 + 5\*111 + 6\*118) % 5 = 4

Тъй като хеш-функцията връща стойност 4, записът с ключ “Petrov” ще бъде записан в свързан списък с индекс 4 в хеш-таблицата.

1 начин: реализира хеширане със списък с препълване.

#include <iostream.h>

#include <iomanip.h>

#include <fstream.h>

#include <string.h>

#include "LList.cpp"

struct rec

{char\* Name;

unsigned Num;

rec(const char\* s = "", unsigned n = 0)

{Name = new char[strlen(s)+1];

strcpy(Name, s);

Num = n;

}

~rec()

{delete Name;

}

rec& operator=(const rec& r)

{if(this!=&r)

{delete Name;

Name = new char[strlen(r.Name)+1];

strcpy(Name, r.Name);

Num = r.Num;

}

return \*this;

}

};

typedef LList<rec> List\_Rec;

void LList<rec>::print()

{elem\_link1<rec> \*p = Start;

while (p)

{cout << p->inf.Name << " " << p->inf.Num << " ";

p = p->link;

}

cout << endl;

}

class HashTable

{public:

HashTable(unsigned = 1021);

~HashTable();

unsigned hash(const char\*) const;

void Insert(const char\*, unsigned);

void print();

elem\_link1<rec>\* Find(const char\*) const;

void Delete(const char\*);

private:

unsigned K;

List\_Rec \*arr;

};

HashTable::HashTable(unsigned n)

{K = n;

arr = new List\_Rec[n];

}

HashTable::~HashTable()

{delete [] arr;

}

unsigned HashTable::hash(const char\* str) const

{unsigned sum = 0;

for (unsigned i = 0; str[i]; i++)

sum += (i+1)\*str[i];

return sum % K;

}

void HashTable::Insert(const char\* str, unsigned num)

{rec r(str, num);

arr[hash(str)].ToEnd(r);

}

void HashTable::print()

{for (unsigned i = 0; i < K; i++)

{arr[i].print();

cout << endl;

}

}

elem\_link1<rec>\* HashTable::Find(const char\* str) const

{arr[hash(str)].IterStart();

elem\_link1<rec>\* p = arr[hash(str)].Iter();

while (p && strcmp(p->inf.Name, str))

p = p->link;

return p;

}

void HashTable::Delete(const char\* str)

{arr[hash(str)].IterStart();

elem\_link1<rec>\* p = arr[hash(str)].Iter();

while(p && strcmp(p->inf.Name, str))

p = p->link;

if(p)

{rec y;

arr[hash(str)].DeleteElem(p, y);

}

}

void main()

{ifstream f("c:\\zad\\Phone\_Number.txt", ios::in);

if (!f)

{ cerr << "Cannot open file Phone\_Number.txt!\n";

return;

}

char Name[100];

unsigned Num, K;

cout << "Hash table length K: "; cin >> K;

HashTable HashObj(K);

while (f >> Name >> Num)

{cout << setw(35) << setiosflags(ios::left)

<< Name << setw(20) <<Num << endl;

HashObj.Insert(Name, Num);

}

HashObj.print(); cout << endl;

elem\_link1<rec>\* p = NULL;

cout << "Enter a name or end to quit!\n";

cin >> Name;

while (strcmp(Name, "end"))

{p = HashObj.Find(Name); //HashObj.Delete(Name)

if (p) cout << "Found.\n";

else cout << "Not found.\n";

cout << "Enter a name or end to quit!\n";

cin >> Name;

}

HashObj.print();

f.close();

}

**2 начин: реализира двойно хеширане**

struct rec

{char Name[32];

unsigned Num;

};

class HashTable

{public:

HashTable(unsigned = 1021);

~HashTable();

void Insert(const char\*, unsigned);

rec\* Find(const char\*);

void Print() const;

private:

unsigned hash(const char\*); // хеш-функция

unsigned Increment() const; // втора хеш-функция

bool Search(const char\*, unsigned &); // търсене на ключ

unsigned N, // размер на таблицата

sum; // за низа "Len", sum = 1.'L'+2.'e' + 3.'n'

rec\* arr; // носител на хеш-таблицата

};

Конструкторът създава празна хеш-таблица (едномерен масив от 11 елемента от тип rec) и инициализира полето Name на всички записи на хеш-таблицата с празния низ. Чрез член-функцията hash:

unsigned HashTable::hash(const char\* str)

{sum = 0;

for(unsigned i = 0; str[i]; i++)

sum += (i+1)\*str[i];

return sum % N;

}

за всеки ключ на записите от файла намираме първичния му индекс.

Ако те са различни, всеки запис се разполага на съответното място в хеш-таблицата. Ако има колизии се прилага втора хеш-функция Increment(), която дава стойността на увеличението, чрез което се намира мястото на елемента в хеш-таблицата. В случая имаме:

Ключ Първичен индекс

Petrov 6

Todorov 10

Kolev 9

Penev 0

Lekov 5

Dineva 6

Tonova 8

Хеш-функцията пресмята еднозначно първичните индекси i на първите пет ключа. При шестия ключ възниква колизия:

hash(“Petrov”) = hash(“Dineva”) = 6

При разполагането на записите в хеш-таблицата проверката за колизия се осъществява чрез проверка дали низ, различен от празния е записан на мястото, определено от първичния индекс. В случая записът с ключ “Petrov” се записва в позиция 6 на хеш-таблицата, записът с ключ “Todorov” се записва в позиция 10 на хеш-таблицата, записът с ключ “Kolev” се записва в позиция 9 на хеш-таблицата и т.н. На записа с ключ “Dineva” е отредена позиция 6 на хеш-таблицата. Тя е заета от записа с ключ “Petrov”. Намирането на мястото за записа с ключ “Dineva” се осъществява чрез прилагане на втора хеш-функция - член-функцията Increment() на класа HashTable, която определя стойността на увеличението.

unsigned HashTable::Increment() const

{return 1 + sum % (N-2);

}

В случая sum е 2184, а unsigned incr = Increment(); свързва incr със 7. Операторът

i = (i + incr) % N

с начална стойност 6 за i се изпълнява докато (но по-малко от N пъти) се намери i, така че в позиция i на хеш-таблицата има празен запис. В случая след първото изпълнение на този оператор за i се получава 2. Тъй като позиция 2 на хеш-таблицата е празна, записът с ключ “Dineva” се записва в нея и хеш-таблицата има вида:

0 Penev 9929292

1

2 Dineva 9346523

3

4

5 Lekov 9767688

6 Petrov 9123456

7

8 Tonova 9999222

9 Kolev 9121212

10 Todorov 9943937

Ако за incr се беше получила стойност 5, следващата стойност на i е 0. Тъй като позиция 0 в хеш-таблицата е запълнена, щеше да се пресметне поредната стойност на i. Тя е 5 и тъй като позиция 5 на таблицата е запълнена, се пресмята поредната стойност на i. Тя е 10 и отново позиция 10 на таблицата е запълнена. Следващата стойност на i е 4 и тъй като позиция 4 на таблицата е свободна, в нея щеше да се запише компонентата с ключ “Dineva”.

При N равно на 11, начална стойност на i равна на 6 и стойност 5 на incr се получава следната редица от стойности за i:

0, 5, 10, 4, 9, 3, 8, 2, 7, 1, 6, 0, 5, 10, 4, 9, 3, 8, 2, 7, 1, 6, …

Първите N числа са различни и са от 0 до 10. Това не е случайно, а е следствие на факта, че 5 и 11 са взаимно прости. Следващите числа са повторение на първите N. Затова пресмятането на i се ограничава до N. Ако генерираните по описания по-горе начин N позиции в хеш-таблицата са заети, хеш-таблицата е пълна.

Ако N не беше просто число, то би могло да се дели на стъпката incr. Например, ако N е 10, i е 2, а incr е 5, за празни ще бъдат изследвани позициите: 2, 7, 2, 7,...

За да се осигури N и incr да са взаимно прости трябва по подходящ начин да бъде дефинирана член-функцията Increment(). Дефиницията 1 + sum % (N-2)

се оказва подходяща когато N и N-2 са прости числа. Наричат се още числа – близнаци. Някои примери за такива числа са:

3 и 5, 41 и 43, 101 и 103, 311 и 313, 599 и 601, 1019 и 1021, 2999 и 3001, 7127 и 7129, 10007 и 10009, 30011 и 30013, ...

#include <iostream.h>

#include <iomanip.h>

#include <fstream.h>

#include <string.h>

struct rec

{char Name[32];

unsigned Num;

};

class HashTable

{public:

HashTable(unsigned = 1021);

~HashTable();

void Insert(const char\*, unsigned);

void Delete(const char\*);

rec\* Find(const char\*);

void Print() const;

unsigned hash(const char\*);

unsigned Increment() const;

bool Search(const char\*, unsigned &);

private:

unsigned N, // размер на таблицата

sum; // за низа "Len" sum = 1.'L'+2.'e' + 3.'n'

rec\* arr; // носител на таблицата

};

HashTable::HashTable(unsigned n)

{if (n > 3) N = n; else N = 3;

arr = new rec[N];

for (unsigned i = 0; i < N; i++)

arr[i].Name[0] = '\0';

}

HashTable::~HashTable()

{delete arr;

}

bool HashTable::Search(const char\* str, unsigned &i)

{unsigned count = 0, incr;

if (strcmp(arr[i].Name, str))

{incr = Increment();

do

{count++;

if (count == N) return 0;

i = (i+incr) % N;

} while(strcmp(arr[i].Name, str));

}

return 1;

}

unsigned HashTable::hash(const char\* str)

{sum = 0;

for (unsigned i = 0; str[i]; i++)

sum += (i+1)\*str[i];

return sum % N;

}

unsigned HashTable::Increment() const

{return 1 + sum%(N-2);

}

void HashTable::Insert(const char\* str, unsigned Num)

{unsigned i = hash(str);

if (!Search("", i)){cout << "Hash table is full.\n"; return;}

strcpy(arr[i].Name, str);

arr[i].Num = Num;

}

void HashTable::Delete(const char\* str)

{unsigned i = hash(str);

if(!Search(str, i)){cout << "Error\n"; return;}

strcpy(arr[i].Name, "");

arr[i].Num = 0;

}

rec\* HashTable::Find(const char\* str)

{unsigned i = hash(str);

if (Search(str, i)) return arr + i;

return NULL;

}

void HashTable::Print() const

{for (unsigned i = 0; i < N; i++)

if (strcmp(arr[i].Name, ""))

cout << setw(5) << i

<< setw(20) << arr[i].Name

<< setw(10) << arr[i].Num << endl;

else cout << setw(5) << i << endl;

cout << endl;

}

void main()

{ifstream f("Phone\_Number.txt", ios::in);

if (!f)

{ cout << "Cannot open file Phone\_Number.txt!\n";

return;

}

char Name[16];

unsigned Num, N;

cout << "Hash table length N - a prime number: ";

cin >> N;

HashTable PhoneTable(N);

while(f >> Name >> Num)

{cout << setw(20) << setiosflags(ios::left)

<< Name << setw(10) << Num << endl;

PhoneTable.Insert(Name, Num);

}

PhoneTable.Print();

cout << endl;

rec\* p = 0;

cout << "Enter a name or end to quit!\n";

cin >> Name;

while(strcmp(Name, "end"))

{p = PhoneTable.Find(Name);

if(p) cout << "Found.\n";

else cout << "Not found.\n";

cout << "Enter a name or end to quit!\n";

cin >> Name;

}

PhoneTable.Print();

f.close();

}

## Разрешаването на колизии чрез пряко свързване

Да предположим, че искаме да създадем програма-телефонен указател, която съхранява в масив данните (име и телефонен номер) на група абонати на дадена телефонна централа и реализира операциите търсене, вмъкване и изтриване по ключ. За ключ ще използваме името на абоната.

За целта ще създадем масив от указатели MasSp. Размерът на масива може да бъде равен на броя на лицата, но може и да е по-малък и тогава колизиите са неизбежни. Всеки указател-елемент на масива ще разглеждаме като указател към свързан списък, който първоначално e празен.

В програма, реализираща тази идея, основните операции добавяне, търсене и изтриване ще се реализират по следния начин:

*а) Добавяне данните за абонат:*

* въвеждаме името и телефонния номер на абонат;
* намираме индекса на елемента от масива, който съответства на името (ключа) на абоната, като използваме хеш-функцията;
* към свързания списък на намерения елемент от масива добавяме нов елемент и в него записваме данните на абоната. Колизия няма да настъпи, защото във всеки списък от масива от списъци винаги можем да добавим още един възел. По този начин някои от списъците в масива от списъци ще останат празни, други ще имат по един възел, трети ще имат по два и повече възли.

*б) Търсене данните за абонат:*

* намираме индекса на елемента от масива, който съответства на името (ключа), като използваме хеш-функцията;
* в свързания списък, указван от елемента - указател, търсим възел със зададения ключ.

*в) Изтриване данните за абонат:*

* по ключа, по описания начин, намираме възела, който трябва да изтрием;
* изтриваме намерения възел по правилата за изтриване на възел в свързан списък.

Фиг. 17. Хеш-таблица

Следващата програма, реализира изложената идея. Тя е в два файла: hash1.cpp и hash1ap.cpp.

Във файла hash1.cpp са дадени:

* клас Info, включващ член-данните за едно лице и член-функциите на класа;
* структурата ElemSp, включваща член-данните на възел (елемент) от свързан списък;
* клас Spisak, включваща указател към свързан списък и член-функциите на класа;
* клас HashTabl, описващ цялата хеш таблица “Телефонен указател” и операциите в нея.

В този файл специално внимание трябва да отделим на хеш-функцията. В литературата могат да се намерят много идеи за създаване на хеш-функция. В настоящата програма идеята е следната:

Ключът ime е масив от символи s[]. Всеки символ s[i] умножаваме по i+1. Намираме сумата от тези произведения. Намираме индекса, като остатък от целочисленото деление на сумата на N, където N е броят на елементите в масива-таблица. N може да е число, по-малко от броя на лицата, включени в телефонния указател (защо). Препоръчва се N да е просто число.

**Програма 3.1.** Тя реализира телефонен указател и изпълнява следените операции:

* създава празна хеш-таблица;
* добавя към хеш-таблицата данните за едно лице;
* прави справки - намира тел. номер по зададено име;
* изтрива абонати от таблицата.

Файл hash1.cpp от програма телефонен указател

#include <iostream.h>

#include <stdlib.h>

#include <string.h>

//Клас Info, описващ данните за един обект от инф. система

class klasInfo{

public:

char ime[51];

unsigned long tel\_nom;

void getData();

void display();

};

void klasInfo::getData()

{

cin.sync();

cout<<"Въведете име: "; cin.getline(ime,50);

cout<<"Въведете тел. номер: "; cin>>tel\_nom;

}

void klasInfo::display()

{

cout<<ime<<' '<<tel\_nom<<" ";

}

//Структура, представляваща елемент от списъка

struct ElemSp{ klasInfo Info; ElemSp \*next;};

//Клас на списъците-елементи на масива

class Spisak {

public:

ElemSp \*first; //Указател към списък

Spisak() { first=0; }//Конструктор, инициализиращ списък

~Spisak();

void DobElSp(klasInfo &info);

void IztrElSp( char im[]);

ElemSp \*TrsElSp(char im[]);

void PrintSp();

};

//Деструктор, изтриващ списък-елемент на масива

Spisak::~Spisak()

{

ElemSp \*q;

while (first){

q=first; first=first->next; delete q;

}

}

//Член-функция за изтриване на елемент от списъка

void Spisak::IztrElSp(char im[])

{

ElemSp \*p=first, \*q;

if (p==0) return;//Списъкът е празен

if (strcmp(p->Info.ime, im) == 0)

{//Изтриваме първия елемент от списъка

first = p->next;

delete p;

return;

}

for (;;) {// Търсим в списъка елемента за изтриване

q=p->next;

if (q==0) break; //Не е намерен

if (strcmp(q->Info.ime, im)==0){

p->next=q->next;

delete q;

return;

}

p=q;

}

}

//Член-функция за включване на елемент към списъка

void Spisak::DobElSp(klasInfo &info)

{

ElemSp \*p=new ElemSp;

p->Info=info;

p->next=first; first=p;

}

//Член-функция за намиране елемент от списъка, съдържащ зададено име

ElemSp \*Spisak::TrsElSp(char im[])

{

ElemSp \*p=first;

while (p && strcmp(p->Info.ime, im) != 0) p=p->next;

return p;

}

//Член-функция за извеждане на екрана данните в един списък

void Spisak::PrintSp()

{

if (first==0) {cout<<"Списъкът е празен.\n"; return;}

ElemSp \*p=first;

while (p) {

p->Info.display();

p=p->next;

}

cout<<endl;

}

//Клас "Хеш-таблица"

class HashTabl {

unsigned n; //Максимален размер на хеш-таблицата

Spisak \*MasSp;//Указател към на хеш-таблицата (масива от списъци)

public:

HashTabl(unsigned len=1021) //Конструктор, създаващ хеш-таблицата

{

n=len; MasSp=new Spisak[n];//Създава динамичния масив MasSp

}

~HashTabl(){delete[] MasSp;} //Деструктор, изтриващ масива хеш-таблица

void DobElTab(klasInfo &info);//Член-функция за добавяне елемент към таблицата

ElemSp \*TrsElTab(char im[]) //Член-функция за намиране елемент от таблицата

{

return MasSp[hash\_func(im)].TrsElSp(im);

}

//Член-функция за изтриване на елемент от таблицата

void IztrElTab( char im[])

{

MasSp[hash\_func(im)].IztrElSp(im);

}

void PrintTab();

unsigned hash\_func(char im[]);

};

//Член-функция за добавяне на нов абонат към хеш-таблицата

void HashTabl::DobElTab(klasInfo &info)

{

MasSp[hash\_func(info.ime)].DobElSp(info);

}

//Член-функция за извеждане на екрана данните на всички абонати

void HashTabl::PrintTab()

{

for (int i=0; i<n; i++) {cout<<i<<" -> "; MasSp[i].PrintSp();}

}

//Хеш-функция

unsigned HashTabl::hash\_func(char im[])

{ unsigned sum = 0;

for (int i=0; im[i]; i++) sum+=(i+1)\*im[i];

return sum % n;

}

*Файл “hash1ap.cpp” от програма телефонен указател*

#include <iostream.h>

#include <iomanip.h>

#include <conio.h>

#include <string.h>

#include "hash1.cpp"

void main()

{

char ImeM[51],//Име на лице, чийто данни търсим

KodOp; //Код на операцията, която искаме да се изпълни

klasInfo InfoM; //Обект, съдържащ данните за едно лице

unsigned N; ElemSp \*p=0;

cout << "\nБрой на редовете в таблицата : "; cin >> N;

HashTabl TablUk(N);//Създава хеш-таблица "TablUk" с N реда

do{

cout<<"\t МЕНЮ НА ОПЕРАЦИИТЕ:\n";

cout<<"\t\t1 - добавяне на елемент към таблицата\n";

cout<<"\t\t2 - справка по ключ\n";

cout<<"\t\t3 - изтриване на елемент от таблицата\n";

cout<<"\t\t4 - извеждане съдържанието на таблицата\n";

cout<<"\t Посочете операция или 0 за край:"; cin>>KodOp;

switch (KodOp){

case '0': break;

case '1':cout<<"Въведете данните за ново лице.\n";

InfoM.getData();

InfoM.display();cout<<endl;

TablUk.DobElTab(InfoM);

break;

case '2':cout<<"Въведете име: ";cin>>ImeM;

p=TablUk.TrsElTab(ImeM);

if (p) {cout<<"Намерено: ";p->Info.display();}

else cout<<"Не е намерено.\n";

getch();break;

case '3':cout<<"Въведете име: ";cin>>ImeM;

TablUk.IztrElTab(ImeM);

getch();break;

case '4':TablUk.PrintTab(); getch(); break;

default:cout<<"\n\tВъведен е недопустим код за операция. Натиснете ENTER.\n";

getch();

}

} while (KodOp!='0');

}

Горната програма илюстрира добре хеширането като ефективна техника за съхраняване и извличане на обекти, съдържащи уникален ключ, както и понятията хеш-таблица и хеш-функция. Тя обаче е неудобна за практическо използване, защото данните за абонатите се съхраняват в масив, т.е. в оперативната памет, която, както е известно, е енергозависима.

Следващата програма представлява развитие на предходнатс. Добавени са две нови възможности;

* за запазване на данните от хеш-таблицата в двоичен файл;
* за зареждане на хеш-таблицата с данните от двоичния файл.

Новата програма може да бъде използвана пряко в практиката, както и да послужи за създаване на други програми - информационни системи.

**Програма 3.2.** Тя реализира телефонен указател и изпълнява следните операции:

* създава празна хеш-таблица;
* добавя към таблицата данните за нови абонати;
* прави справки - намира тел. номер по зададено име;
* изтрива абонати от хеш-таблицата;
* запазва хеш-таблицата във двоичен файл;
* зарежда хеш-таблицата с данните от двоичния файл.

*Файл “hash2.cpp” oт програмата телефонен указател.*

#include <stdlib.h>

#include <string.h>

//Клас Info, описващ данните за един абонат

class klasInfo{

public:

char ime[51];

unsigned long tel\_nom;

void getData();

void display();

};

//Член-функция за въвеждане данните за един абонат

void klasInfo::getData()

{

cin.sync();

cout<<"Въведете име: "; cin.getline(ime,50);

cout<<"Въведете тел. номер: "; cin>>tel\_nom;

}

//Член-функция за въвеждане данните за един абонат

void klasInfo::display()

{

cout<<ime<<' '<<tel\_nom<<" ";

}

ofstream fout; ifstream fin;

//Структура, описваща възел (елемент) от списъка

struct ElemSp{ klasInfo Info; ElemSp \*next;};

//Клас на списък-елемент на масива-таблица

class Spisak {

public:

ElemSp \*first; //Указател към списък

Spisak() { first=0; } //Конструктор, инициализиращ списък

~Spisak();

void DobElSp(klasInfo &info);

void IztrElSp( char im[]);

ElemSp \*TrsElSp(char im[]);

void Spisak::ZapaziSp();

void PrintSp();

};

//Деструктор, изтриващ всички елементи на списък-елемент на масива

Spisak::~Spisak()

{

ElemSp \*q;

while (first){

q=first; first=first->next; delete q;

}

}

//Член-функция за изтриване на елемент от списъка

void Spisak::IztrElSp(char im[])

{

ElemSp \*p=first, \*q;

if (p==0) return; //Списъкът е празен

if (strcmp(p->Info.ime, im) == 0)

{//Изтриваме първия елемент от списъка

first = p->next;

delete p;

return;

}

for (;;) {// Търсим в списъка елемента за изтриване

q=p->next;

if (q==0) break; //Не е намерен

if (strcmp(q->Info.ime, im)==0){

p->next=q->next;

delete q;

return;

}

p=q;

}

}

//Член-функция за добавяне на възел (елемент) към списъка

void Spisak::DobElSp(klasInfo &info)

{

ElemSp \*p=new ElemSp;

p->Info=info;

p->next=first; first=p;

}

//Член-функция за намиране елемент от списъка, съдържащ зададено име

ElemSp \*Spisak::TrsElSp( char im[])

{

ElemSp \*p=first;

while (p && strcmp(p->Info.ime, im) != 0) p=p->next;

return p;

}

//Член-функция за запазване на хеш-таблицата в двоичен файл

void Spisak::ZapaziSp()

{

klasInfo InfoF;

if (first==0) {cout<<"Списъкът е празен.\n"; return;}

ElemSp \*p=first;

while (p) {

InfoF=p->Info; fout.write((char\*) &InfoF,sizeof(InfoF));

p=p->next;

}

cout<<endl;

}

//Член-функция за извеждане списък от хеш-таблицата на екрана

void Spisak::PrintSp()

{

if (first==0) {cout<<"Списъкът е празен.\n"; return;}

ElemSp \*p=first;

while (p) {

p->Info.display();

p=p->next;

}

cout<<endl;

}

//Клас "Хеш-таблица"

class HashTabl {

unsigned n; //Максимален размер на хеш-таблицата

Spisak \*MasSp;//Указател към на хеш-таблицата (масива от списъци)

public:

HashTabl(unsigned len=1021) //Конструктор, създаващ хеш-таблицата

{

n=len; MasSp=new Spisak[n]; //Създава динамичния масив MasSp

}

~HashTabl(){delete[] MasSp;} //Деструктор, изтриващ хеш-таблицата

void DobElTab(klasInfo &info);//Член-функция за добавяне елемент към таблицата

ElemSp \*TrsElTab(char im[]) //Член-функция за намиране елемент от таблицата

{

return MasSp[hash\_func(im)].TrsElSp(im);

}

//Член-функция за изтриване на елемент от таблицата

void IztrElTab( char im[])

{

MasSp[hash\_func(im)].IztrElSp(im);

}

void PrintTab();

unsigned hash\_func(char im[]);

void ZapaziTab();

void ZarediTab();

};

//Член-функция за добавяне данните за един абонат към хеш-таблицата

void HashTabl::DobElTab(klasInfo &info)

{

MasSp[hash\_func(info.ime)].DobElSp(info);

}

//Член-функция за зареждане хеш-таблицата с данните от двоичния файл

void HashTabl::ZarediTab()

{

klasInfo InfoF;

fin.open("Telephon", ios::in|ios::binary);

for(;;){

fin.read((char\*) &InfoF,sizeof(InfoF));

if (fin.eof()) break;

MasSp[hash\_func(InfoF.ime)].DobElSp(InfoF);

}

fin.close();

}

//Член-функция за извеждане данните от хеш-таблицата на екрана

void HashTabl::PrintTab()

{

for (int i=0; i<n; i++) {cout<<i<<" -> "; MasSp[i].PrintSp();}

}

//Член-функция за запазване данните от хеш-таблицата в двоичния файл

void HashTabl::ZapaziTab()

{

fout.open("Telephon", ios::out|ios::binary);

for (int i=0; i<n; i++) MasSp[i].ZapaziSp();

delete []MasSp;

fout.close();

}

//Хеш-функция

unsigned HashTabl::hash\_func(char im[])

{ unsigned sum = 0;

for (int i=0; im[i]; i++) sum+=(i+1)\*im[i];

return sum % n;

}

*Файл “hash2.cpp” oт програмата телефонен указател.*

#include <fstream.h>

#include <iomanip.h>

#include <conio.h>

#include <string.h>

#include "hash2.cpp"

void main()

{

char otgovor,ImeM[51],KodOp; klasInfo InfoM;

unsigned N; ElemSp \*p=0;

ifstream inf("TeleUk.txt", ios::in);

if (!inf) {cout << "Не може да се отвори файлът TeleUk.txt.\n"; return;}

cout << "\nБрой на редовете в таблицата : "; cin >> N;

HashTabl TablUk(N); //Създава хеш-таблица "TablUk" с N реда

do{

cout<<"\t МЕНЮ НА ОПЕРАЦИИТЕ:\n";

cout<<"\t\t1 - добавяне на елемент към таблицата\n";

cout<<"\t\t2 - справка по ключ\n";

cout<<"\t\t3 - изтриване на елемент от таблицата\n";

cout<<"\t\t4 - извеждане съдържанието на таблицата\n";

cout<<"\t\t5 - запазване съдържанието на таблицата във файл\n";

cout<<"\t\t6 - зареди таблицата от файл\n";

cout<<"\t Посочете операция или 0 за край:"; cin>>KodOp;

switch (KodOp){

case '0': break;

case '1': cout<<"Въведете данните за ново лице.\n";InfoM.getData();

InfoM.display();cout<<endl;

TablUk.DobElTab(InfoM);

break;

case '2': cout<<"Въведете име: ";cin>>ImeM;

p=TablUk.TrsElTab(ImeM);

if (p) {cout<<"Намерено: ";p->Info.display();}

else cout<<"Не е намерено.\n";

getch();break;

case '3': cout<<"Въведете име: ";cin>>ImeM;

TablUk.IztrElTab(ImeM);

getch();break;

case '4': TablUk.PrintTab(); getch(); break;

case '5': TablUk.ZapaziTab(); getch(); break;

case '6': TablUk.ZarediTab(); getch(); break;

default:cout<<"\n\tВъведен е недопустим код за операция. Натиснете ENTER.\n"; getch();

}

} while (KodOp!='0');

}

Пример /Trekto/: Да се напише програма, чрез която да обслужваме студентите.

За целта сме направили клас Student, който съдържа следните данни: факултетен номер, име и фамилия на студента. Факултетният номер е уникален и чрез него ще идентифицираме студента. С нашата програма трябва да можем да добавяме и премахваме студенти и да променяме техните данни. За целта ще съхраняваме обекти от тип Student в някаква структура от данни.

**Реализация**

Във файла [student.h](http://trekto.info/algoritmi-strukturi-danni/src/student.h) е поместен класът Student.

/\*\*

\* http://trekto.info/asd

\*/

#include <iostream>

#include <string>

using namespace std;

class Student {

public:

string fn;

string firstName;

string lastName;

Student() {

}

Student(string fn, string firstName, string lastName) {

this->fn = fn;

this->firstName = firstName;

this->lastName = lastName;

}

friend ostream &operator << (ostream &os, const Student &aStudent) {

os << aStudent.fn << " - " << aStudent.firstName << " " << aStudent.lastName;

return os;

}

};

Във файла [student\_list.h](http://trekto.info/algoritmi-strukturi-danni/src/student_list.h) е класът List, който съдържа методи за добавяне, премахване и търсене:

/\*\*

\* http://trekto.info/asd

\*/

#include <iostream>

#include <string>

#include <cstdlib> // нужна за използване на exit()

#include "student.h" // описва класа Student

using namespace std;

struct Elem {

Student data; // данни

Elem\* next; // указател към следващия елемент

};

class List {

private:

Elem\* pStart; // указател към началото на списъка

void copyList(const List &aList); // помощен метод за копиране на списъка

public:

List(); // конструктор по подразбиране

List(const List &aList); // копиращ конструктор

~List(); // деструктор

void add(Student aData); // добавя студент

void remove(string fn); // премахва студента с ф.н. fn

Student find(string fn); // намира и връща студента с ф.н. fn

void removeAll(); // премахва всички елементи от списъка

List &operator = (const List &aList); // оператор за присвояване

friend ostream &operator << (ostream &os, List &aList) { // оператор за извеждане

Elem\* p = aList.pStart;

while(p) {

os << p->data << '\n';

p = p->next;

}

return os;

}

};

void List:: add(Student aData) { // добавя студент

Elem\* temp = new Elem; // заделяме памет за новия елемент

temp->data = aData; // копираме в него данните

temp->next = pStart; // насочваме 'следващия' му към стария първи елемент

pStart = temp; // правим новия елемент първи

}

Student List:: find(string fn) { // намира и връща студента с ф.н. fn

Elem\* p = pStart; // насочваме указател за обхождане към началото

while(p && p->data.fn != fn) // докато списъкът не е свършил и не е намерен студентът

p = p->next; // преминаваме към следващия студент

if(!p) { // ако списъкът свърши преди да сме намерили студента

cerr << "Няма студент с ф.н.: " << fn << "!\n\n";

exit(1);

}

return p->data; // връщаме студента

}

void List:: remove(string fn) { // премахва студента с ф.н. fn

if(!pStart) { // ако списъкът е празен

cerr << "Няма студент с ф.н.: " << fn << "!\n\n";

exit(1);

}

if(fn, pStart->data.fn == fn) { // ако студентът е в началото на списъка

Elem\* temp = pStart; // насочваме помощен указател към първия елемент

pStart = pStart->next; // насочваме началото към 'следващия' му

delete temp; // изтриваме първия

return;

}

Elem\* p = pStart; // насочваме указател за обхождане към началото

while(p->next && p->next->data.fn != fn) // докато списъкът не е свършил

// и не е намерен студентът

p = p->next; // преминаваме към следващия студент

if(!p->next) { // ако списъкът свърши преди да сме намерили студента

cerr << "Няма студент с ф.н.: " << fn << "!\n\n";

exit(1);

}

// щом сме стигнали до тук p трябва да сочи към студента преди този с ф.н. fn

Elem\* temp = p->next; // насочваме временен указател към ел. който ще премахваме

p->next = p->next->next; // премахваме елемента от списъка

delete temp; // и го изтриваме

}

List:: List() {

pStart = NULL;

}

List:: List(const List &aList) {

pStart = NULL;

copyList(aList);

}

List:: ~List() {

removeAll();

}

void List:: copyList(const List &aList) {

Elem\* p = aList.pStart;

while(p) {

add(p->data);

p = p->next;

}

}

List &List:: operator = (const List &aList) {

if(this != &aList) {

removeAll();

copyList(aList);

}

return \*this;

}

void List:: removeAll() {

Elem\* temp;

while(pStart) {

temp = pStart;

pStart = pStart->next;

delete temp;

}

}

Във файла [hash.h](http://trekto.info/algoritmi-strukturi-danni/src/hash.h) е реализиран класът HashTable. Той има методи за добавяне, премахване, търсене на студент:

/\*\*

\* http://trekto.info/asd

\*/

#include <iostream>

#include "student\_list.h"

using namespace std;

class HashTable {

private:

List\* a; // масив от списъци

unsigned n; // брой списъци

public:

HashTable(unsigned n = 100) { // конструктори

this->n = n;

cout << "hash start\n";

a = new List[n]; // тук се извикват конструкторите на всички списъци от масива

cout << "hash end\n";

}

~HashTable() { // деструктор

delete[] a; // тук се извикват деструкторите на списъците

}

unsigned hash(string fn) { // хеш функция

unsigned sum = 0;

for(int i = 0; i < fn.length(); i++) {

sum += (i+1) \* fn[i];

}

return sum % n;

}

void add(Student student) { // добавя нов студент

a[hash(student.fn)].add(student);

}

void remove(string fn) { // премахва студента с ф.н. fn

a[hash(fn)].remove(fn);

}

Student find(string fn) { // намира и връща студента с ф.н. fn

return (a[hash(fn)]).find(fn);

}

friend ostream &operator << (ostream &os, HashTable &aHashTable) { // извежда hash таблицата

os << "===================================\n";

for(int i = 0; i < aHashTable.n; i++)

os << i << ": \n" << aHashTable.a[i] << "\n";

os << "===================================\n";

return os;

}

};

За всеки символ от факултетния номер, хеш функцията умножава номера на неговата позиция по ASCII стойността му, след което събира тези произведения. Получената сума се дели на броя на списъците в хеш таблицата и се взема остатъкът от делението. Идеята с остатъка върши нелоша работа, когато входните данни са сравнително равномерно разпределени.

unsigned hash(string fn) { *// хеш функция*

unsigned sum = 0;

**for**(int i = 0; i < fn.length(); i++) {

sum += (i+1) \* fn[i];

}

**return** sum % n;

}

В програмата [hash.cpp](http://trekto.info/algoritmi-strukturi-danni/src/hash.cpp) се демонстрира използването на хеш таблицата.

/\*\*

\* http://trekto.info/asd

\*/

#include <iostream>

#include "hash.h"

using namespace std;

int main() {

HashTable hashTable(5);

hashTable.add(Student("0801561302", "Веселина", "Георгиева"));

hashTable.add(Student("0801571303", "Наталия", "Бекриева"));

hashTable.add(Student("0801581304", "Милена", "Кючюкова"));

hashTable.add(Student("0801581305", "Димитър", "Димитров"));

hashTable.add(Student("0801601306", "Ангел", "Титов"));

hashTable.add(Student("0801611307", "Петко", "Петков"));

cout<<"Хеш таблицата: \n" << hashTable << "\n";

cout << "Търсене на 0801561302...\n";

cout << hashTable.find("0801561302") << "\n\n";

hashTable.remove("0801581305");

hashTable.add(Student("0901911206", "Стефан", "Младенов"));

cout << "Хеш таблицата: \n" << hashTable << "\n";

hashTable.remove("0801581005");

return 0;

// тук се вика деструкторът на hashTable, който извиква деструкторите на всички списъци

}

Пример

Да се въведат данни за работници в корпорация, които се идентифицират със своето ЕГН (10-цифрено цяло число), като се използва хешишране.

Упътване За да реализираме хеш-таблицата, ще използваме масив A[] с *n* елемента. Ако броя на работниците е по-малък или равен на *n*, то ще има достатъчно памет за записването им в масива A. За да стане обаче записването със сложност *O*(1), трябва да разполагаме с функция, която на ключа на всеки елемент съпоставя еднозначно адрес от 0 до *n*–1. Например, от произволно 10-цифрено число *k* може да се получи цяло число от 0 до *n*–1 с единствена проста операция: *k* % *n*. Остатъкът, получен при делението на *k* с капацитета на хеш-таблицата, е една възможна хеш-функция.