Речници, хеширане, множества

доц. д-р Нора Ангелова

Търсене

- Последователно търсене
- Двоично търсене
- Хеширане

Последователно търсене

$$a_0, a_1, \ldots, a_{n-1}$$

- Последователно сканиране на елементите на редицата до срещане на елемент с указаното свойство или до изчерпване на редицата без да е намерен елемент.

Последователно търсене

```
template <typename T>                    // първи вариант
int linSearch(T x, T* a, int n)
{ int i = 0;
   while (i < n-1 && x != a[i]) i++;
   if (x == a[i]) return i;
   return n;
}
```

Последователно търсене

Някои често срещани решения

```
template <typename T>                    // втори вариант
int linSearch(T x, T* a, int n)
{ int i = 0; a[n] = x;
  while (x != a[i]) i++;
  return i;
}
```

Използване на сентинел. Внимавайте за размерността

Ако x не е елемент в масива?

Последователно търсене

Някои често срещани решения

```
template <typename T>                    // трети вариант
int linSearch(T x, T* a, int n)
{ a[n] = x;
  for (int i = 0; i < n; i++)
     if (x == a[i]) break;
  return i;
}
```

При какви условия цикълът се прекратява

i == ?

Последователно търсене

```
template <typename T>
int linSearch(fstream& f, T x)
{ // намиране на броя на компонентите на f
  f.seekg(0, ios::end);
  long n = f.tellg()/sizeof(T);
  // търсене на x във файла
  long i = -1; T a;
  do
  { i++;
    f.seekg(i*sizeof(T));
    f.read((char*)&a, sizeof(T));
  } while (i < n-1 && x != a);
  if (x == a) return i;
  return n;
}
```

```
long i = -1;
do
   i++;
while (i < n-1 && x != a[i]);
if (x==a[i]) return i;
return n;
```

Двоично търсене

$$a_0, a_1, \ldots, a_{n-1}$$

- Редицата е наредена.
- Разглежда се средният елемент.
- Ако той е търсеният, сканирането се преустановява.
- В противен случай търсенето продължава в лявата или дясната половина на редицата в зависимост от това дали x е по-малък или по-голям от средния елемент.
- Действията се повтарят за съответната подредица.

Речник

- Структура, реализираща връзка между две множества от елементи – ключове и стойности

* асоциативен списък и карта (map)

Речник

Операции

- Създаване на празен речник – create()
- Търсене на стойност по ключ – lookup(key)
- Добавяне на стойност с даден ключ – add(key, value)
- Изтриване на ключ (изтрива и свързаната стойност) – remove(key)
- Обхождане на ключове и/или стойности – keys(), values()

Речник

Реализация

- Свързан списък

  Сложности:

  - Търсене на стойност по ключ
  - Добавяне на стойност с даден ключ – реализира се в края на списъка
  - Изтриване на ключ

Речник

Реализация

- Свързан списък

  Сложности:

  - Търсене на стойност по ключ – O(n)
  - Добавяне на стойност с даден ключ – O(1), реализира се в края на списъка
  - Изтриване на ключ – O(n)

Речник

Реализация

- Свързан списък, сортиран по ключове

  Обхождането е в нарастващ ред.

  Сложности:

  - Търсене на стойност по ключ – ?
  - Добавяне на стойност с даден ключ – O(n), добавя се на определено място
  - Изтриване на ключ – O(n)

Речник

Реализация

- Свързан списък, сортиран по ключове

  Обхождането е в нарастващ ред.

  Сложности:

  - Търсене на стойност по ключ – O(n), не се подобрява!
  - Добавяне на стойност с даден ключ – O(n), добавя се на определено място
  - Изтриване на ключ – O(n)

Речник

Реализация

- Сортиран динамичен масив

  Обхождането е в нарастващ ред.

  Спираме търсенето по-рано, но сложността не се променя значително

  Сложности:

  - Търсене на стойност по ключ – O(log n), може да се използва двоично търсене
  - Добавяне на стойност с даден ключ – O(n), добавя се на определено място
  - Изтриване на ключ – O(n)

Речник

Реализация

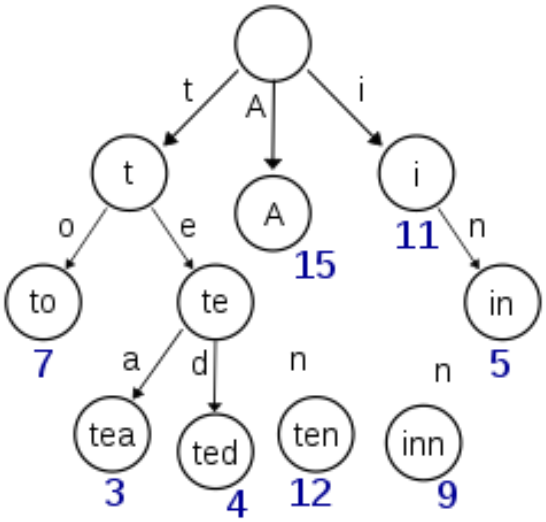
- Двоично наредено дърво
  Обхождане ЛКД – възходящо обхождане по ключове
  Сложности:
  - Търсене на стойност по ключ
  - Добавяне на стойност с даден ключ
  - Изтриване на ключ

  Средна сложност – O(log n)
  Най-лоша сложност – O(n)
- Балансирано дърво
  Сложност на операции – O(log n)
- B-дърво
  Сложност на операции – O(log n)
  Допълнително предимство: всеки възел е с фиксиран размер
  Може да доведе до увеличаване на производителността при работа с твърд диск

Речник

Реализация

- Trie



Хеширане

Техниката хеширане

Това е техника за организация на редица от записи, позволяваща ефективно вмъкване, изтриване и търсене. Записите са организирани така, че да съдържат уникален ***ключ***, чрез който се определя позицията им в редицата.

Целта е като се използва подходящо дефинирана функция, наречена ***хеш-функция*** (***hash***), дадените записи да се запишат в ***хеш-таблица***. Добре е последната да е така организирана, че основните операции над нея (вмъкване, изтриване и търсене) да са от порядъка на константа.

Хеширане

Нека наличното количество памет да е достатъчно за разполагане на не повече от *n* записа, номерирани от 0 до *n-1*. Хеш-функцията трябва да е такава, че в резултат от прилагането ѝ към ключовете на записите ѝ, да се получи цяло число от 0 до *n-1*, наречено ***първичен индекс***.

Не се иска на два различни ключа *k1* и *k2* хеш-функцията *hash* да съпоставя различни цели числа. Състояние, при което за $k1 \neq k2$ е в сила $hash(k1) = hash(k2)$, се нарича ***колизия***. Ключовете *k1* и *k2* са в колизия. Казва се и че *k1* и *k2* са ***синоними***.

Два често използвани начина за разрешаване на колизиите са ***прякото свързване*** и ***отвореното адресиране***.

Хеширане

Не е трудно да се види, че хеш-функцията задава *релация на еквивалентност*: *множеството от синоними от един и същ клас, и само те, образуват един клас на еквивалентност.*

***Принцип (на Дирихле, принцип на чекмеджетата)*:**
Дадени са n чекмеджета. Ако сложим в тях $n+1$ предмета, то в поне едно чекмедже ще има поне два предмета.

Колизиите са неизбежни. Даже да се дефинира „перфектна“ хеш-функция, колизия гарантирано ще настъпи най-късно в момента, в който броят на включените в хеш-таблицата елементите надхвърли капацитета й (съгласно принципа на Дирихле).

Хеш таблица

- Масив от n двойки ключ/стойност
- Хеш функция h: key -> [0, n)
  - Ключ -> пресмята позиция в масива
  - Колизия – h(k1) = h(k2) и k1 != k2
  - Идея: дава възможно най-добро разбъркване и намаляване на колизиите
  - Колизиите не могат да бъдат предотвратени изцяло!

Хеш таблица

- Хеш таблица

  Сложности:

  - Търсене на стойност по ключ
  - Добавяне на стойност с даден ключ
  - Изтриване на ключ

  Средна сложност – O(1)

  Най-лоша сложност – O(n)

Хеширане

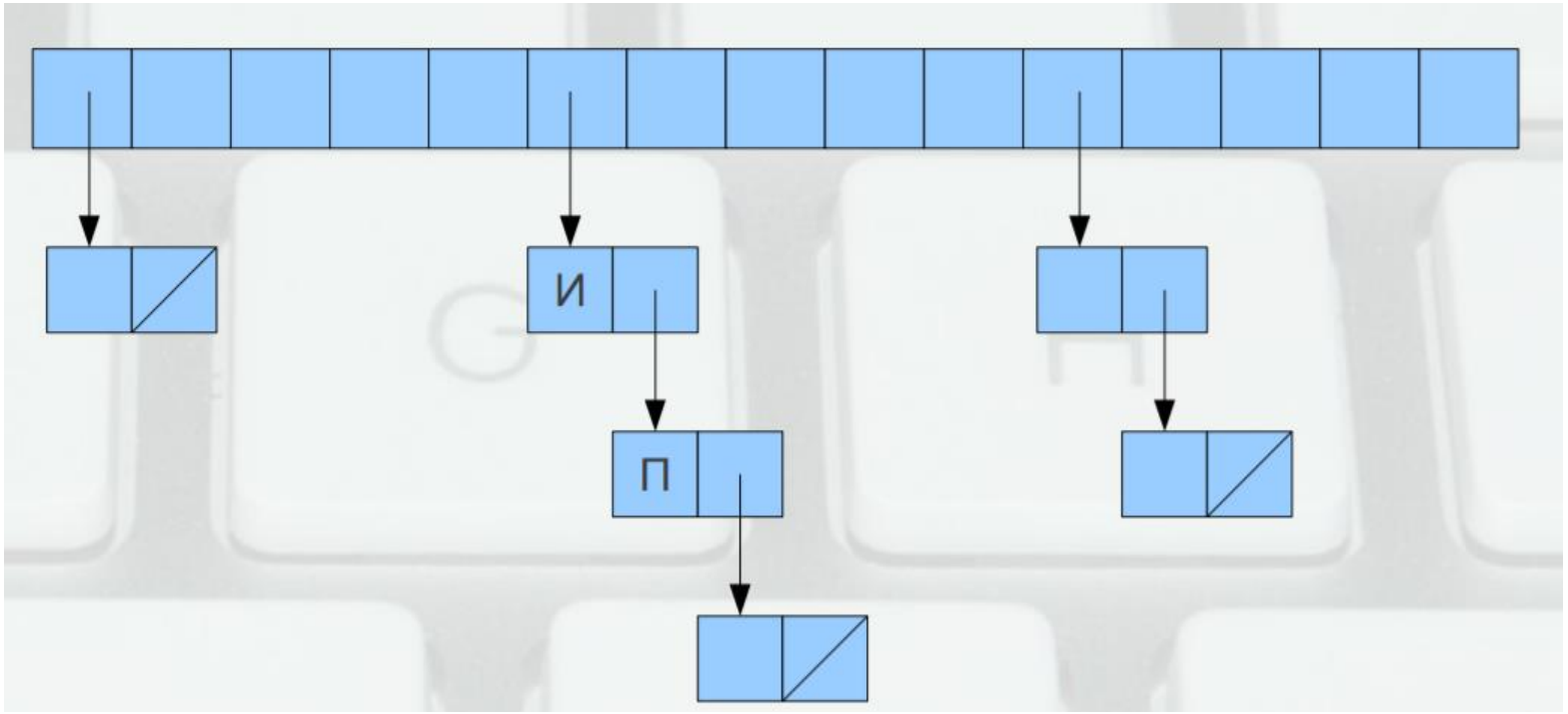
На практика колизия ще настъпи доста по-рано (като че ли по Закона на Мърфи, но всъщност статистически очаквано).

Пример с рожденните дати. Каква е вероятността измежду 25 души да има двама, родени на една и съща дата (т. е. със съвпадащи ден и месец на раждане)? Оказва се, че тази вероятност е 0.5073.

А при 100 души? – вероятността е почти 1.

Хеширане – пряко свързване

Хеш-таблицата в този случай е едномерен масив от k свързани списъка, представени с една връзка (k може да е по-голямо, равно или по-малко от n).



Хеширане – пряко свързване

Хеш-таблицата в този случай е едномерен масив от k свързани списъка, представени с една връзка (k може да е по-голямо, равно или по-малко от n).

При прилагане на хеш-функцията към ключа на запис се получава първичен индекс, по който в масива се намира линейният списък, над който ще се приложи желаната операция (търсене, изтриване или вмъкване).

Търсенето в тези списъци е възможно, тъй като ключовете са част от записите и са уникални.

Ако хеш-функцията е добра, свързаните списъци ще бъдат къси и методът ще е със сложност от порядъка на константа.

Хеширане – пряко свързване

Пример. Ще използваме следната таблица с n $(n = 7)$ записа.

Фамилия на студент	Факултетен номер
Petrov	34560
Todorov	43930
Kolev	21210
Penev	92980
Lekov	76360
Dineva	46520
Tonova	92220

Хеширане – пряко свързване

Пример.
Всеки запис означава фамилно име и факултетен номер на студент.
За ключ ще използваме символния низ, означаващ фамилията на студент. Тази таблица ще е вход на програмата, дефинирана по-долу, и ще се задава чрез текстовия файл *Student_Number.txt*
Програмата прехвърля записите на файла в хеш-таблица с k елемента.
- Ако $k < 7$, непременно ще има колизии.
- Ако $k \geq 7$, тогава отново може да има колизии.

Хеш-функцията може да се определи по различни начини. Ще я дефинираме като *public* член-функция на класа *hashTable* по следния начин:

Хеширане – пряко свързване

Дефиниране на хеш-функция

```
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;
}
```

Хеширане – пряко свързване

При $k = 5$ за ключа „Petrov“ се получава първичен индекс

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

и показва, че записът с ключ „Petrov“ ще бъде включен в линейния списък с индекс 4 на хеш-таблицата.

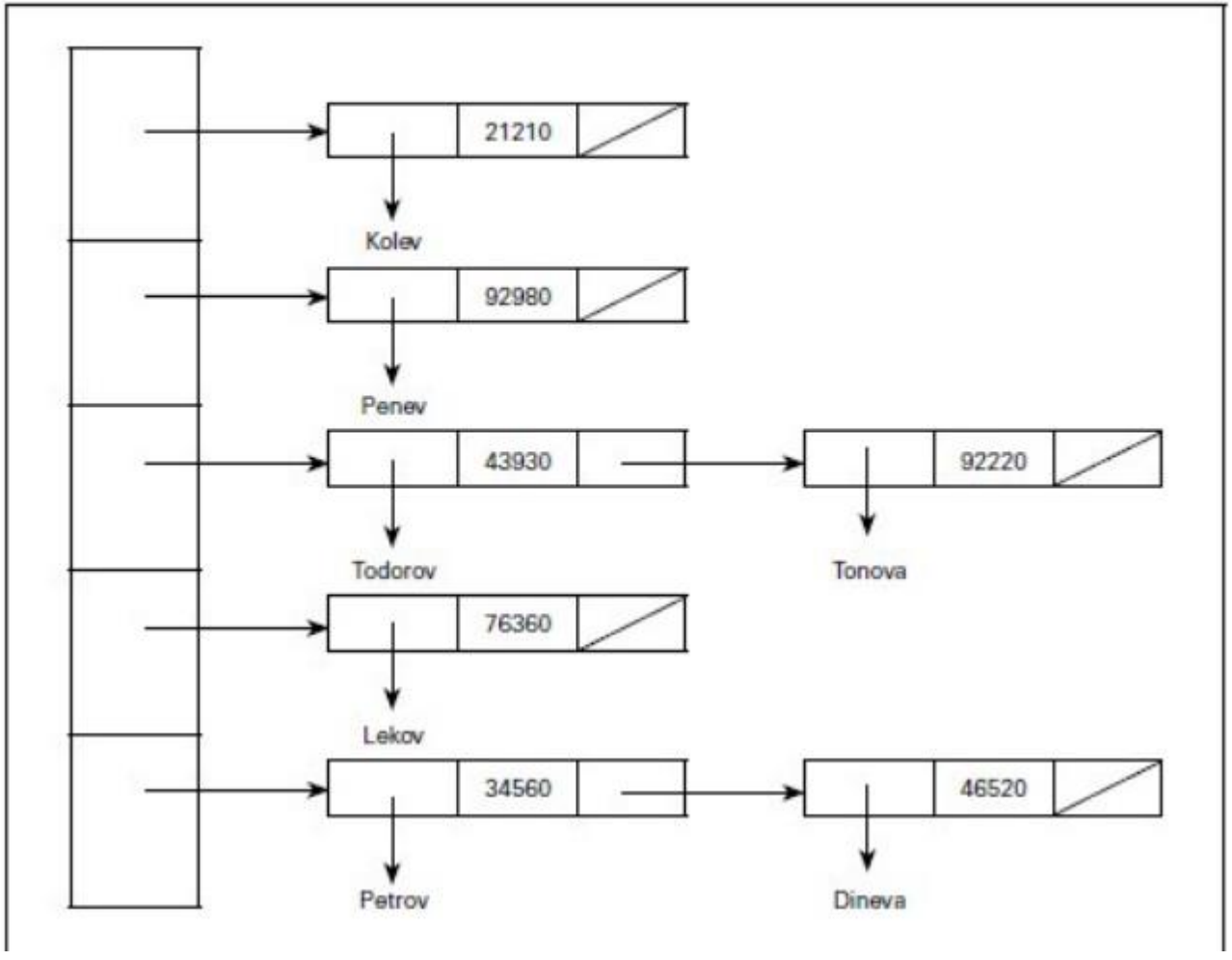
Хеширане – пряко свързване

За записите от файла *Student_Number.txt* първичните индекси са:

Ключ	**Първичен индекс**
Petrov	4
Todorov	2
Kolev	0
Penev	1
Lekov	3
Dineva	4
Tonova	2

Хеширане – пряко свързване

Хеш-таблицата има вида:



Хеширане – пряко свързване

Вместо в линеен списък, елементите, които са в колизия, могат да се пазят в структура, в която търсенето е с по-малка алгоритмична сложност от тази на линейния списък. Подходяща структура е двоично наредено дърво.

По такъв начин всеки елемент на едномерния масив, представящ хеш-таблицата, ще бъде указател към корена на двоично дърво.

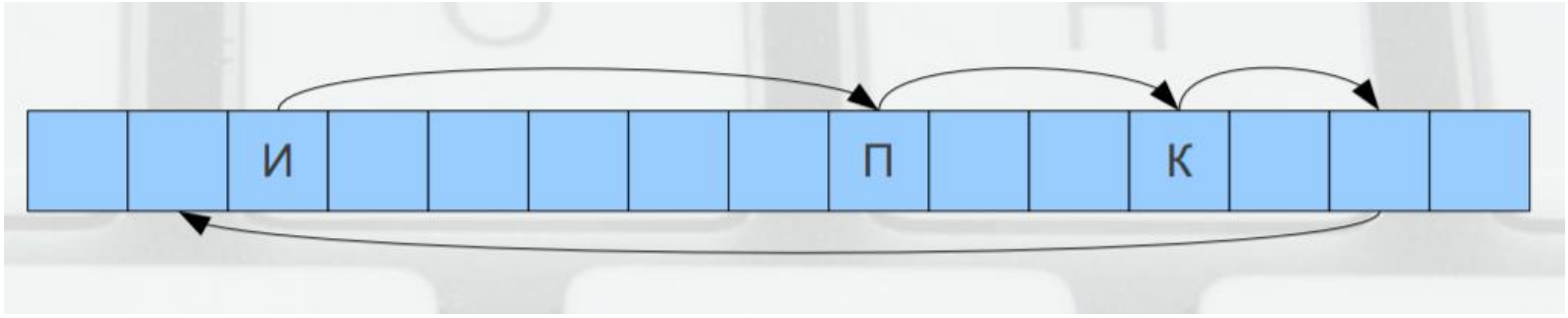
* *Сложност на операциите*

https://www.cs.usfca.edu/~galles/visualization/OpenHash.html

Хеширане – отворено адресиране

При този начин за разрешаване на колизиите ***хеш-таблицата се реализира чрез едномерен масив от записи***.

* *Използва се втора хеш функция.*



Хеширане – отворено адресиране

При този начин за разрешаване на колизиите ***хеш-таблицата се реализира чрез едномерен масив от записи***.

Предимството пред прякото свързване е, че се спестява памет за връзките при реализирането на линейните списъци. Недостатъкът е, че предварително трябва да се задели фиксирано количество памет за хеш-таблицата.

Ако хеш-таблицата е голяма, броят на колизиите намалява и се увеличава броят на елементите, които могат да се вмъкнат, преди хеш-таблицата да се е препълнила.

В този случай е възможно да се задели памет, която в действителност може да не се използва.

Хеширане – отворено адресиране

Пример. Ще използваме следната таблица със 7 записа.

Фамилия на студент	Факултетен номер
Petrov	34560
Todorov	43930
Kolev	21210
Penev	92980
Lekov	76360
Dineva	46520
Tonova	92220

Хеширане – отворено адресиране

Записите на хеш-таблицата са декларирани чрез структурата *rec*. Реализацията на хеш-таблицата и на основните операции над нея се осъществява чрез класа *hashTable*.

```
struct rec
{ char name[32];        // фамилно име
  unsigned num;         // факултетен номер
};
```

Хеширане – отворено адресиране

Дължината *n* на хеш-таблицата *arr* се определя от потребителя, но е желателно да е просто число и по-точно число-близнак (*n* и *n-2* са прости).

За *n* избираме 13 (*n* и *n-2* са прости). Конструкторът създава празна хеш-таблица (едномерен масив от 13 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 се намира първичният му индекс.

Хеширане – отворено адресиране

Ключ	Първичен индекс
Petrov	9
Todorov	5
Kolev	3
Penev	7
Lekov	8
Dineva	0
Tonova	3

- Ако първичните индекси са различни, всеки запис се разполага на съответното място в хеш-таблицата.
- Ако възникнат колизии, се прилага ***втора хеш-функция increment***, която задава стойността на увеличението, чрез което се намира мястото на записа в хеш-таблицата.

Хеширане – отворено адресиране

При седмия ключ възниква колизия:

hash("Kolev") = hash("Tonova") = 3

Намирането на мястото за записа с ключ „Tonova“ 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);
}
```

Хеширане – отворено адресиране

В случая *sum* е 2252, а

```
unsigned incr = increment();
```

свързва *incr* с 9. Операторът

```
i = (i + incr) % n;
```

с начална стойност 3 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 горния оператор за *i* се получава 12. Тъй като позиция 12 на хеш-таблицата е празна, записът с ключ „Tonova“ се записва в нея и хеш-таблицата има вида:

Хеширане – отворено адресиране

0	Dineva	46520
1		
2		
3	Kolev	21210
4		
5	Todorov	43930
6		
7	Penev	92980
8	Lekov	76360
9	Petrov	34560
10		
11		
12	Tonova	92220

Хеширане – отворено адресиране

Ако за *incr* се беше получила стойност 5, следващата стойност на *i* щеше да е 8. Тъй като позиция 8 в хеш-таблицата е запълнена, ще се пресметне следващата стойност на *i*. Тя е 0 и тъй като позиция 0 на таблицата също е запълнена, ще се пресметне следващата стойност на *i*. Тя е 5 и отново позиция 5 на таблицата е запълнена. Следващата стойност на *i* е 10 и тъй като позиция 10 на хеш-таблицата е свободна, в нея щеше да се запише записът с ключ „Tonova“.

При *n* равно на 13, начална стойност на *i* равна на 3 и стойност 9 на *incr* се получава следната редица от стойности за *i*:

3, 12, 8, 4, 0, 9, 5, 1, 10, 6, 2, 11, 7,
3, 12, 8, 4, 0, 9, 5, 1, 10, 6, 2, 11, 7, ...

Хеширане – отворено адресиране

Първите *n* числа са различни и са от 0 до 12. Това не е случайно, а е следствие на факта, че incr и n (9 и 13) са взаимно прости. Следващите *n* числа са повторение на първите *n*. Затова пресмятането на *i* се ограничава до *n* пъти.

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, началната стойност на *i* е 2, а *incr* е 5, за празни ще бъдат изследвани позициите: 2, 7, 2, 7,...

Хеширане – отворено адресиране

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***.

Примери за числа-близнаци.
3 и 5, 11 и 13, 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, ...

Хеширане – отворено адресиране

Отвореното адресиране е част от методите за разрешаване на колизиите, наречени ***затворено хеширане***.

Други подходи за затворено хеширане са: ***линейното пробване*** и ***квадратичното пробване***.

При всички тях хеш-таблицата е представена от едно основно място, където се съхраняват данните (*n* слота), за разлика от реализациите, които за разрешаване на колизиите, използват допълнителна памет).
Когато настъпи колизия, се правят опити да се намери друг адрес, представящ свободен "слот".

Промяната се прави по някаква предварително дефинирана схема.

Хеширане – линейно пробване

При линейното пробване се увеличава първичният индекс с някакво естествено число s ($0 < s < n$). Ако получената позиция стане равна на n, търсенето продължава в началото на хеш-таблицата, като се взема остатъкът 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. НОД(n, s) = 1.

Хеширане – квадратично пробване

При квадратичното пробване се използва ***стъпка*** от вида

$$c_1 i + c_2 i^2,\ c2 \neq 0.$$

Отместването зависи квадратично от поредния номер на пробата за намиране на мястото *i*.

Този метод работи значително по-ефективно от линейното пробване.

Основният му проблем е, че **не гарантира обхождане на цялата таблица, т.е. възможно е да не намери свободно място, макар да има такова.**

Хеширане – квадратично пробване

При квадратичното пробване се използва ***стъпка*** от вида

$$c_1 i + c_2 i^2,\ c2 \neq 0.$$

Отместването зависи квадратично от поредния номер на пробата за намиране на мястото *i*.

Този метод работи значително по-ефективно от линейното пробване.

Основният му проблем е, че **не гарантира обхождане на цялата таблица, т.е. възможно е да не намери свободно място, макар да има такова.**

КРАЙ

Хеширане – квадратично пробване

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)