Структуры и алгоритмы обработки данных. Часть 2

Хеширование (hashing)
Словари

Содержание

Темы семестра

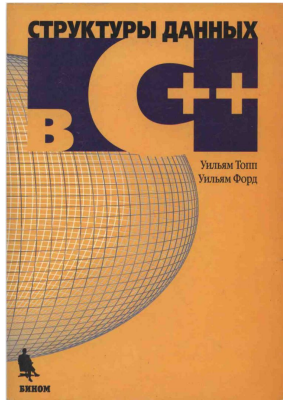
Деревья

Графы

Словарь и хеш-таблица

github.com/ivtipm
/Data-structures-and-algorithms

Литература

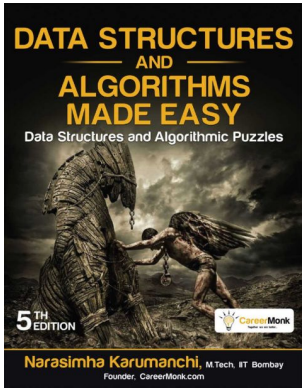


Книги

1. У. Топп, У. Форд. Структуры данных в C++: Пер. с англ. – М.: ЗАО «Издательство БИНОМ», 1999, 816с

2. Ахо А.В. Структуры данных и алгоритмы / А.В. Ахо, Д. Хопкрофт, Д.Д. Ульман. – Москва: Вильямс, 2003. – 384с.: ил.

3. stepik.org/course/579/syllabus



Дополнительно:

4. Narasimha Karumanchi. Data structures and algorithms made easy

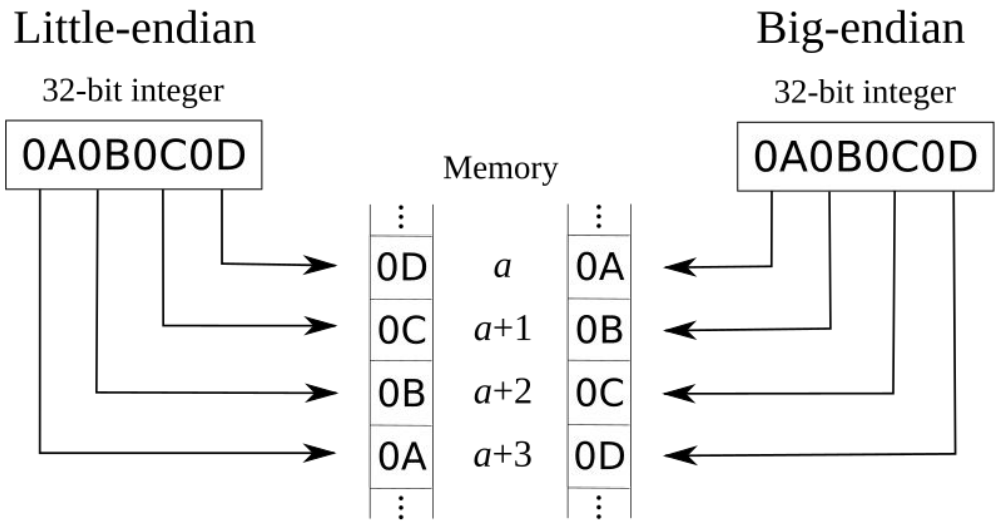
5. academy.yandex.ru/handbook/algorithms

Повторение:
размеры типов данных C++

en.cppreference.com/w/cpp/language/types

Type specifier	Equivalent type	Width in bits by data model				
		C++ standard	LP32	ILP32	LLP64	LP64
signed char	signed char	at least **8**	**8**	**8**	**8**	**8**
unsigned char	unsigned char					
short	short int	at least **16**	**16**	**16**	**16**	**16**
short int						
signed short						
signed short int						
unsigned short	unsigned short int					
unsigned short int						
int	int	at least **16**	**16**	**32**	**32**	**32**
signed						
signed int						
unsigned	unsigned int					
unsigned int						
long	long int	at least **32**	**32**	**32**	**32**	**64**
long int						
signed long						
signed long int						
unsigned long	unsigned long int					
unsigned long int						
long long	long long int (C++11)	at least **64**	**64**	**64**	**64**	**64**
long long int						
signed long long						
signed long long int						
unsigned long long	unsigned long long int (C++11)					
unsigned long long int						

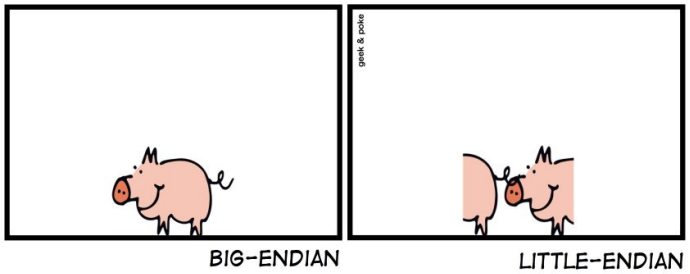
Порядок байт



Сначала старшие байты (начало с большого конца) network byte order

Big-endian порядок используется и в привычных позиционных записях числа, только здесь речь идёт не о байтах, а о цифрах.

Например
1815 — Big-Endian
5181 — Little-Endian



Повторение: Побитовые операции

Bitwise Operators		
For all examples below consider a = 10 and b = 5		
Operator	**Description**	**Example**
&	Bitwise AND	a & b gives 0
|	Bitwise OR	a | b gives 15
^	Bitwise Ex-OR	a ^ b gives 15
~	1's complement (NOT)	~a gives some negative value
<<	Left shift	a << 1 gives 20
>>	Right shift	a >> 1 gives 5

8 4 2 1

10_{10} = 1010

5_{10} = 0101

15_{10} = 1111

20_{10} = 10100

Повторение. Целое число в набор байт

```
/// Преобразует целое число в массив из байт в порядке little-endian
vector<unsigned char> intToBytes(int paramInt)
{
    vector<unsigned char> arrayOfByte(4);
    for (int i = 0; i < 4; i++){
       arrayOfByte[3 - i] = (paramInt >> (i * 8));
    return arrayOfByte;
}

   intToBytes(42);              //  42 0 0 0
   intToBytes(255);             // 255 0 0 0
   intToBytes(256 + 17);        //  11 1 0 0
   intToBytes(256*256);         //   0 0 1 0
   intToBytes(256*256 + 42);    //  42 0 1 0
```

Повторение. Произвольный тип в набор байт

```
template< typename T >
unsigned char * to_bytes( const T& object  ){
   unsigned char * bytes = new unsigned char [sizeof(T)];
   const char* begin = reinterpret_cast< const char* >( &object ) ;
   memcpy( bytes, begin, sizeof(T) ) ;
   return bytes;    }

struct ComplexNumber{
   float a, b;      };

   ComplexNumber a{1,2};

   unsigned char* bytes = to_bytes(a);
   for (int i = 0; i < sizeof(a); ++i)
       cout << (short)bytes[i] << " ";
   // 0 0 128 63  0 0 0 64
```

функция не имеет особого смысла, если значения типа T содержат указатели, т.
е. отличаются преимущественно памятью, которая хранится в поле-указателе

Указатель на функцию

```
return_type (*) (arg1_type, arg2_type, ... );     // общий вид

// Пример
using FuncIntFloat = float (*) (int);

float sqrt(int x) { return pow(x,0.5);   }
float foo(int x)  { return x*x * 22.0/7; }

/// выводит элементы массива, преобразуя их функцией f
void array_apply_n_print(int *arr, unsigned n, FuncIntFloat f){
    for (unsigned i=0; i<n; i++)
    cout << f(arr[i]) << " ";
}
```

https://raw.githubusercontent.com/VetrovSV/OOP/master/OOP_StudBook.pdf#subsection.2.2.3

Коллекция с быстрым доступом?

Проблема

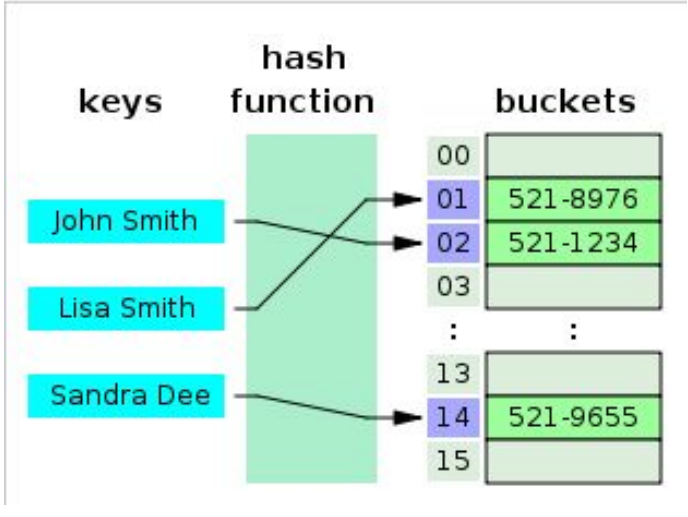
- Сложность операции поиска в массиве или списке — O(n)
- В сбалансированном BST, Skip-List или отсортированном массиве — O(log n)

Можно ли добиться константной сложности поиска в некой коллекции?

Контейнер \ операция	insert	remove	find
Array	O(N)	O(N)	O(N)
List	O(1)	O(1)	O(N)
Sorted array	O(N)	O(N)	O(logN)
Бинарное дерево поиска	O(logN)	O(logN)	O(logN)
Хеш-таблица	O(1)	O(1)	O(1)

Хеш-таблица

- Задать хорошую **хеш-функцию**, для определения индекса элемента в массиве по некому значению k
- Задать **размер** хеш-таблицы (массива)
- Определить алгоритм **решения коллизий**



Хеш функция
hash function

Хеш функция

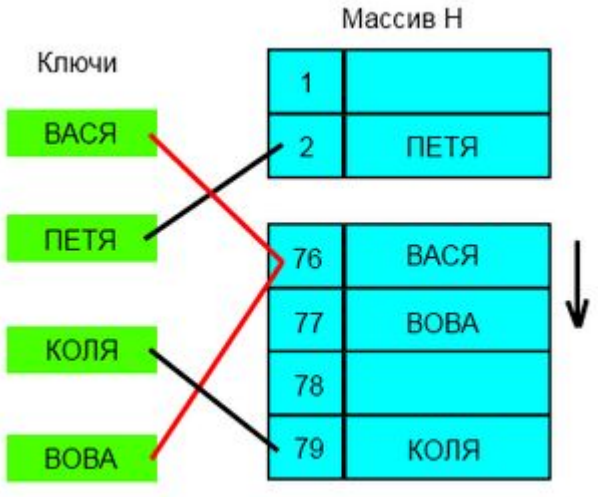
Должна выдавать для некого k его представление в виде целого числа

1. **Свойство равенства**
   Для равных l и k должно соблюдаться h(l) == h(k).
   Это необходимое требование.
2. **Свойство неравенства**
   Для различных l и k должно соблюдаться h(l) != h(k).
   Это желательное свойство.
3. **Быстрое вычисление ~ O(1)**
4. **Равномерное распределение значений**
   Все значения хеш-функции должны быть равновероятны

Коллизия (collision)

Коллизия — получение двух одинаковых ключей для различных значений.

Например, на рис. справа хеш функция выдала одно и то же значение 76 для различных ключей ВАСЯ и ВОВА



Примеры применения хеш функций

Файл для примера:
https://static1.squarespace.com/static/52b30f7ae4b067ba989438d4/t/5a7bb70724a69414063b96f4/1518057223974/Complexity+Cheatsheet.pdf

```
md5sum Complexity+Cheatsheet.pdf
d6e1 d1eb bc5f 1f37 891c 7284 74cf 5a2b

sha256sum Complexity+Cheatsheet.pdf
f938 2017 c04d caed db98 7d43 6844 b8f4
3a5c 63bb e07b 1ee5 73ee 9692 8835 829d
```

Примеры применения хеш функций

См. OpenSSL library

Какие требования соблюдают эти функции?

1. `int h(int k) { return k;}`
2. `int h(int k) { return k+42;}`
3. `int h(int k) { return k+rand();}`
4. `int h(int k) { return rand();}`
5. `int h(int k) { return 42;}`
6. `int h(int k) { return k % 42;}`
7. `int h(int k) { return (int)time(0);}`

Какие требования соблюдают эти функции?

```
unsigned int hashValue(unsigned char key) {
    return (unsigned int)key; // cast key to unsigned int
}
```

```
unsigned int hashValue(string key) {
    unsigned int val = 0;
    for(int i = 0; i < key.length(); i++) {
        val += (unsigned int)(key[i]);  }
    return val;
}
```

Примитивная хеш-функция: метод деления

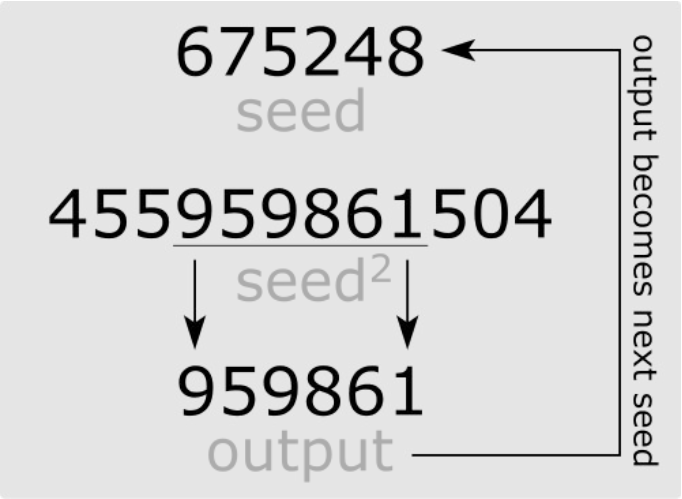
```
int h(type k) {
  int K = int(k);       // преобразование в целое число
  return k % n;   // преобраз-е хеша в индекс рекомендуется делать вне функции
}
```

n – размер хеш-таблицы

- Чем больше n тем больше разброс значений h(k)
- Распределение h(k) будет равномерным, если n – простое число

Простая хеш-функция: метод середины квадрата

1. Преобразовать k в целое число
2. Возвести в квадрат
3. Выбрать биты из середины полученного числа



Метод середины квадрата

```
/// Возвращает средние 10 бит квадрата key
int h(int key){
   key *=key;           // возведение в квадрат
   key = key >> 11;     // отбросить 11 младших бит
   return key % 1024;   // возвратить 11 младших бит
}
```

djb2

```c
unsigned long hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c;

    return hash;
}
```

cse.yorku.ca/~oz/hash.html

Written by Daniel J. Bernstein (also known as djb) in 1991

При мультипликативном методе (multiplicative method) используется случайное действительное число f в диапазоне $0 \le f < 1$. Дробная часть произведения f*key лежит в диапазоне от 0 до 1. Если это произведение умножить на n (размер хеш-таблицы), то целая часть полученного произведения даст значение хеш-функции в диапазоне от 0 до n–1.

```
// хеш-функция, использующая мультипликативный метод;
// возвращает значение в диапазоне 0...700
int HF(int key);
{
  static RandomNumber rnd;
  float f;

  // умножить ключ на случайное число из диапазона 0...1
  f = key * rnd.fRandom();
  // взять дробную часть
  f = f - int(f);
  // возвратить число в диапазоне 0...n-1
  return 701*f;
}
```

Проблема индексации h(k)%m

Хеш-таблица

Индексы в хеш-таблице

Хеш-таблица — хранит данные в массиве.

Индекс при поиске, удалении или вставке элемента массива *вычисляется* по значению хеш-функции.
Обычно индекс вычисляется так: **ind = h(x) % N**
где h(x) – значение хеш-функции для значения x
N — размер массива.

Вероятность возникновения коллизий

Вероятность коллизии при вставке нового ключа в таблицу размера N, где заполнено M ключений:
P(collision) = M/N

Вероятности противоположных событий A и ~A:P(A) = 1 - P (~A)

Вероятность возникновения коллизий

Вероятность отсутствия коллизий при вставке M ключей в таблицу размера N

$P(\text{no collision}) = 1 * (N-1)/N * (N-2)/N * \ldots * (N-M+1)/N$

https://en.cppreference.com/w/cpp/container/unordered_set

произведение т.к. рассматриваются связанные (происходящие один за другим) события (см. формулу условной вероятности)

Вероятность отсутствия коллизий, при заполнении таблицы размера 3 двумя элементами: $P_{3,2}(\text{no collision}) = 1 * ⅔ = ⅔$

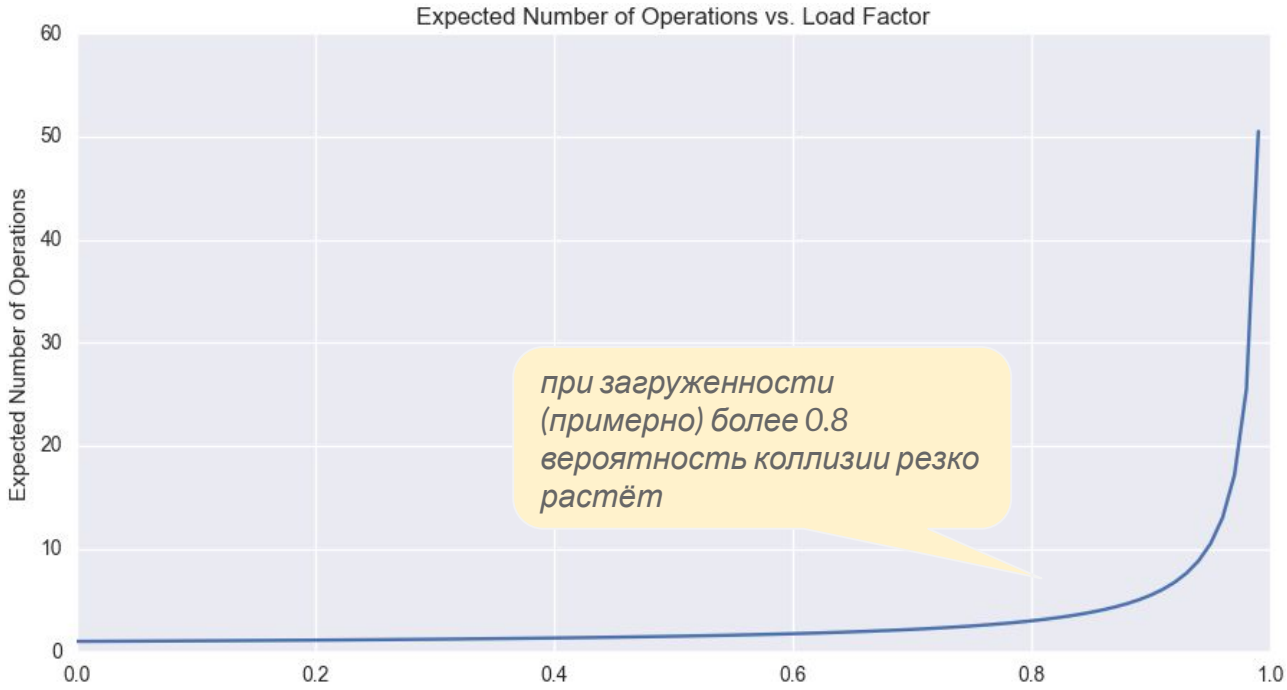
Вероятность коллизии: $P_{3,2}(\text{collision}) = 1 - P_{3,2}(\text{no collision}) = ⅓$

$P_{100,2}(\text{collision}) =$

$P_{100,50}(\text{collision}) =$

$P_{100,75}(\text{collision}) =$

коэффициент заполнения – load factor



при загруженности (примерно) более 0.8 вероятность коллизии резко растёт

N – размер таблицы;
M – количество записей в таблице

Load factor

$$\alpha = \frac{M}{N}$$

$$E(number\ of\ operations) = \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

stepik.org/lesson/31371/step/8?unit=11686

Рехеширование

Рехеширование (Rehashing) — перераспределения существующих элементов из старой таблицы в новую, увеличенного или уменьшенного размера. Соответственно это необходимо для: уменьшения вероятности коллизий, уменьшения объёма занимаемой памяти.

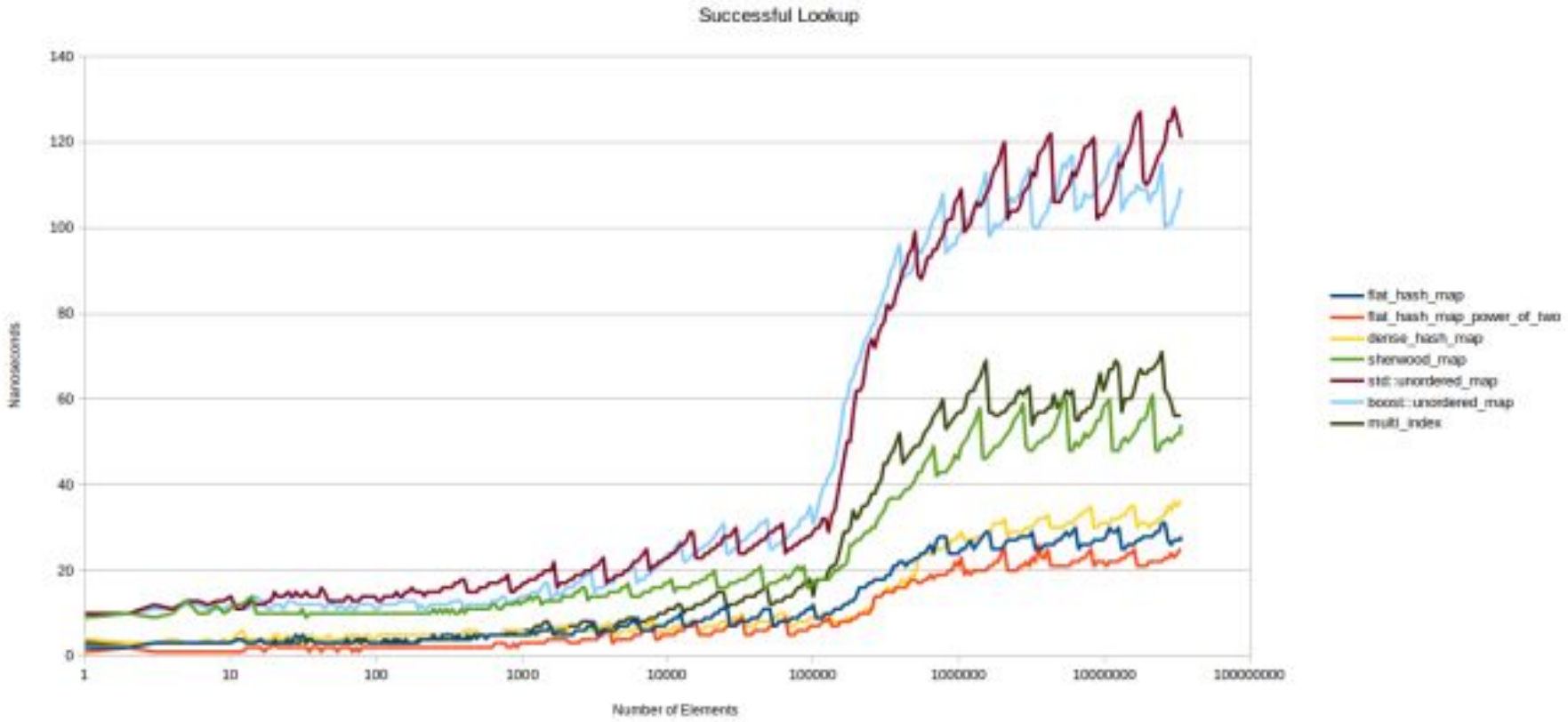
При изменении размера изменяется и значение в формуле вычисления индекса. Нужно пересчитывать индексы для всех элементов.

Изменять размер стоит ориентируясь на коэффициент заполнения (load factor):

Верхний порог (например, $\alpha > 0.7$): инициирует увеличение размера.

Нижний порог (например, $\alpha < 0.2$): инициирует уменьшение размера.

Рехеширование



habr.com/ru/companies/vk/articles/323242

Изменение размера хеш-таблицы в 2 раза

Если размер таблицы – это степень двойки, то можно вычислять индекса через быстрые побитовые операции $x\%2^n = x\&(2^n - 1)$

Если хеш-функция равномерно распределяет значения, метод минимизирует коллизии.

index = hash(key) & (size−1)

Побитовое И оставляет только младшие log2(size) битов значения хеш-функции.

Например, размер таблицы – 16 (2^4):

Для некоторого key: hash(key)=73_{10} (01001001_2)

size−1 = 15_{10} (00001111_2).

Побитовое И: 01001001 & 00001111=00001001=9 index=9.

Изменение размера таблицы до ближайшего простого

Новый размер выбирается как ближайшее большее простое число после текущего размера.

Меньшая вероятность коллизий:

- Простые числа уменьшают вероятность систематических коллизий, вызванных плохой хеш-функцией.
- Простое число гарантирует, что хеш-функция распределяет ключи более равномерно, особенно если ключи кратны определённым значениям.

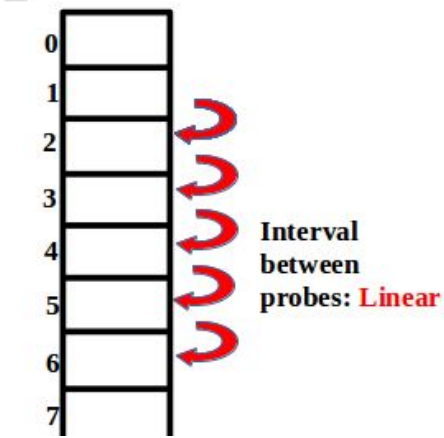
Меньший перерасход памяти: Прирост памяти более умеренный, чем при удвоении размера.

Сложность вычисления: Необходимо дополнительное вычисление для поиска следующего простого числа.

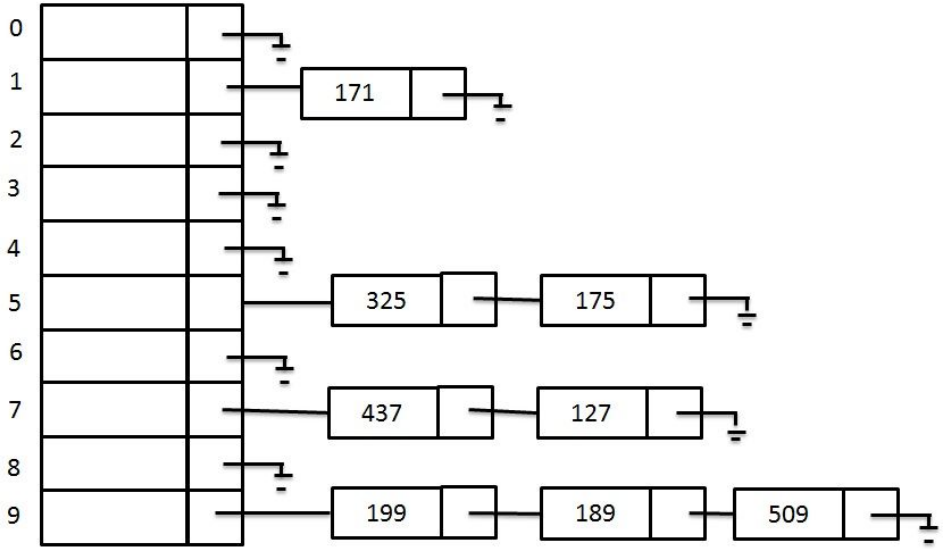
Скорость: Вычисления, связанные с остатком от деления, могут быть менее эффективными, чем побитовые операции.

Методы решения коллизий (collision resolution)

- Найти другую позицию для ключа, место которого уже занято
  - **Открытая адресация** с линейным опробованием (***open addressing with Linear Probing***)
  - Найти следующую свободную позицию в массиве

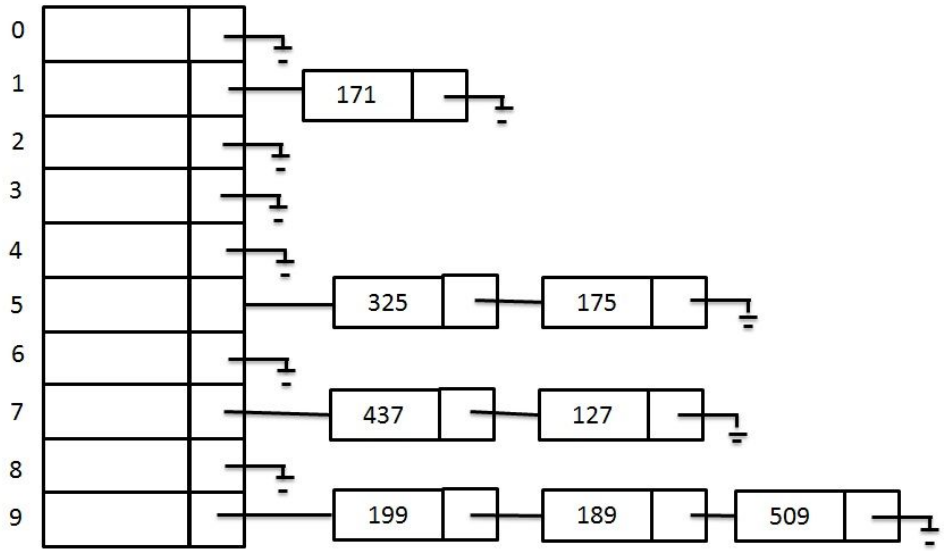


- Хранить значения для одного ключа в списке
  - Метод цепочек (**chaining**)



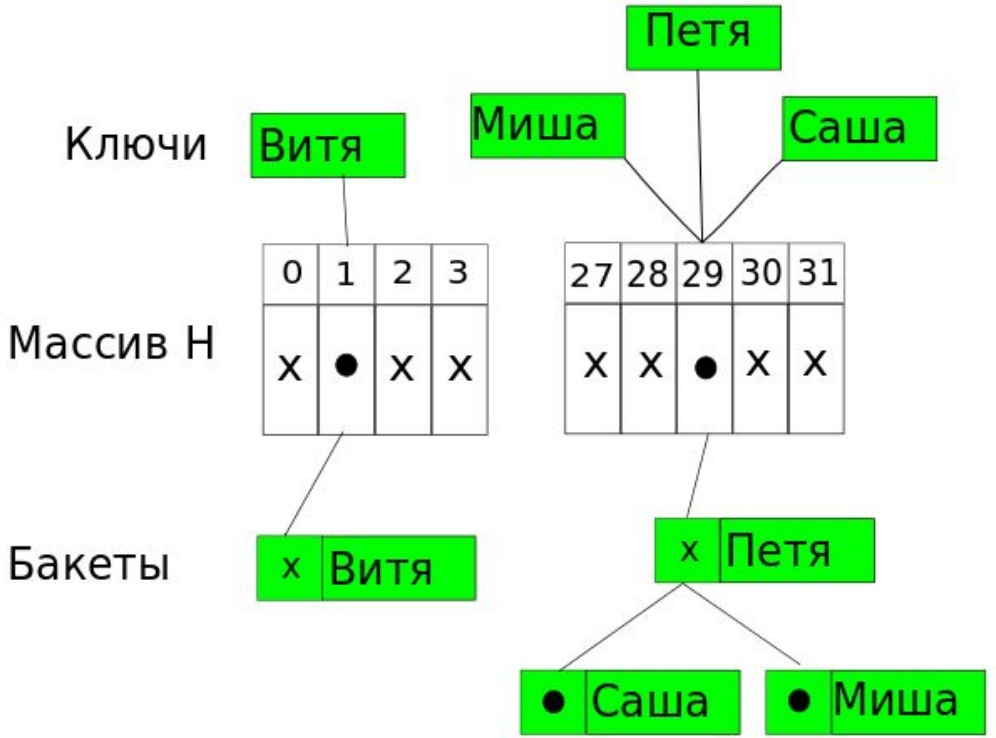
Метод цепочек

- Можно изменять размер таблицы динамически
- Не нужно просматривать всю таблицу целиком чтобы найти свободное место перед текущей ячейкой
- Таблица не должна иметь фиксированный размер n



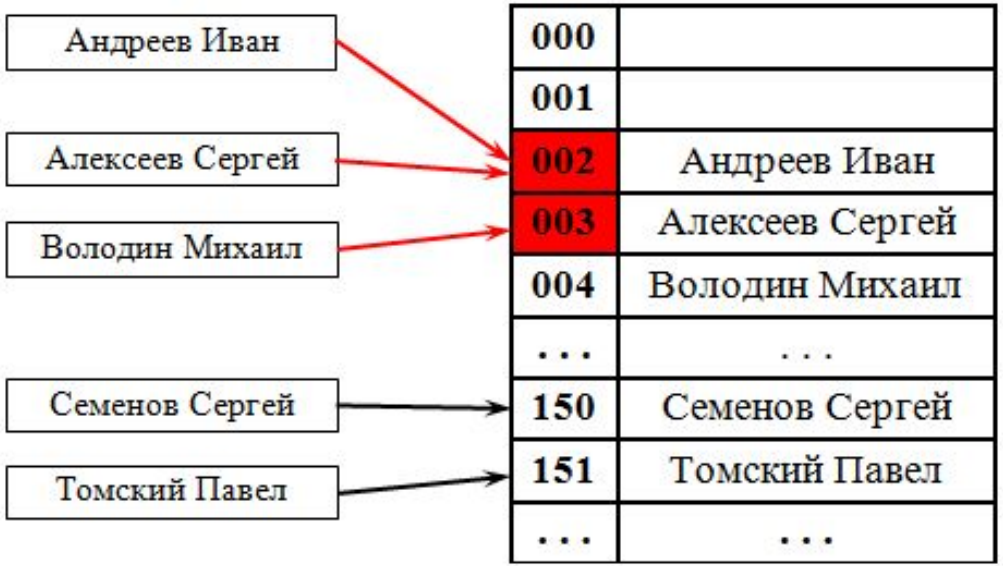
Модифицированный метод цепочек

- Если на элементах хеш-таблицы задан линейный порядок
- То можно использовать вместо линейного списка дерево поиска



Линейное разрешение коллизий: последовательный поиск

При попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки i+1,i+2,i+3 и так далее, пока не найдём свободную ячейку.



Линейное разрешение коллизий: последовательный поиск

$H(k) = k \% N$

$n = 5$

0	1	2	3	4
?	?	?	?	?

Как будет выглядеть таблица после операций, если используется последовательный поиск?

Insert: 5

Insert: 10

Insert: 11

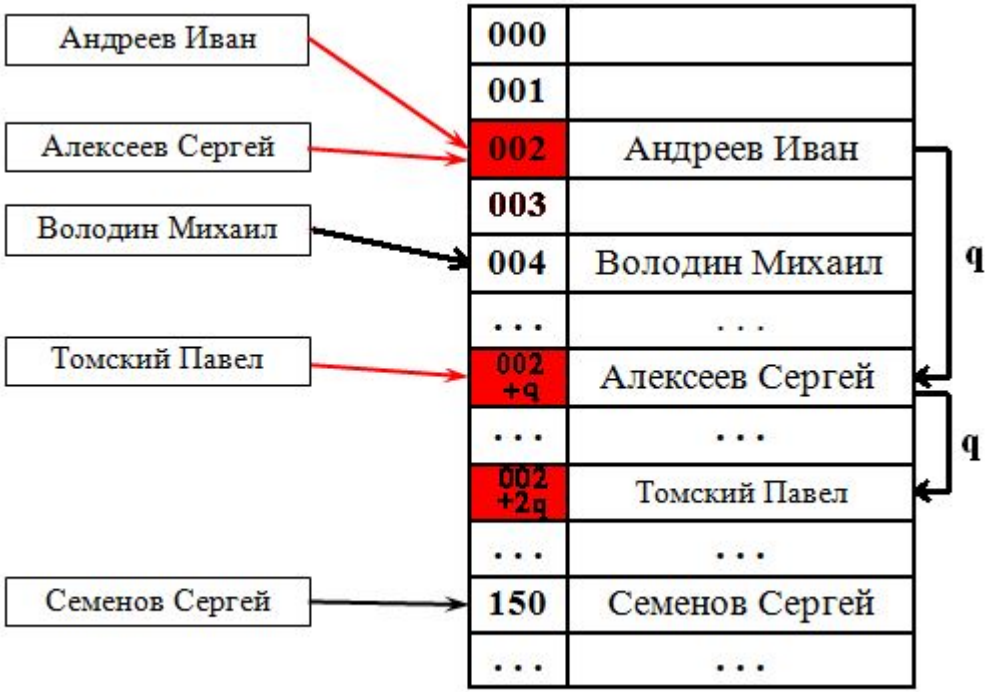
Insert: 12

Insert: 13

Линейное разрешение коллизий: линейный поиск

Выбираем шаг q.

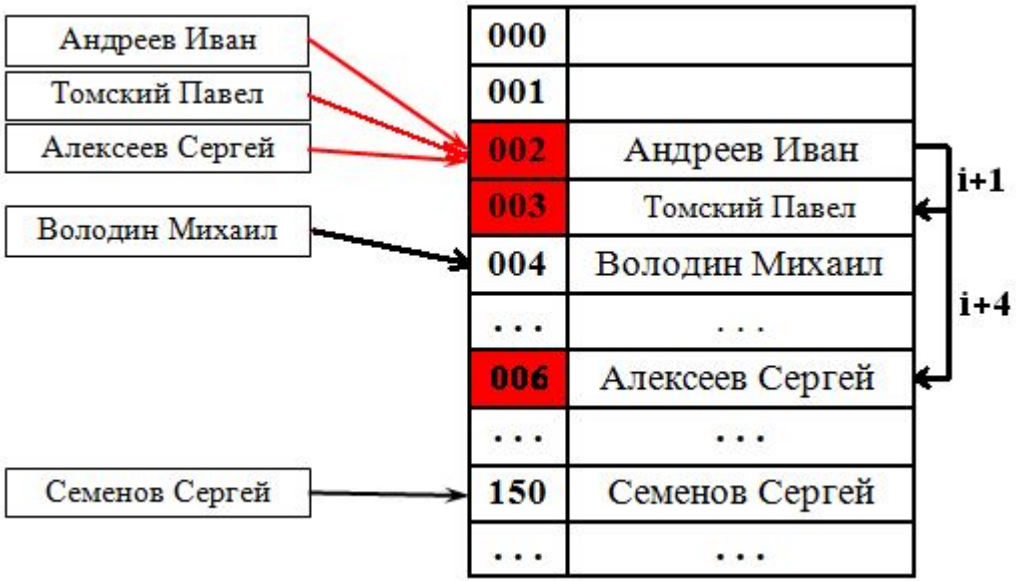
При попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки $i+(1 \cdot q), i+(2 \cdot q), i+(3 \cdot q)$ и так далее, пока не найдём свободную ячейку



Линейное разрешение коллизий: квадратичный поиск

Шаг q не фиксирован, а изменяется квадратично: q=1,4,9,16....

Соответственно при попытке добавить элемент в занятую ячейку i начинаем последовательно просматривать ячейки i+1,i+4,i+9



Двойное хеширование

Двойное хеширование (double hashing) — метод борьбы с коллизиями, возникающими при открытой адресации, основанный на использовании двух хеш-функций для построения различных последовательностей исследования хеш-таблицы.

Двойное хеширование

- используются две независимые хеш-функции h1(k) и h2(k).
- k — ключ,
  m — размер таблицы,

- Сначала исследуется ячейка с адресом h1(k) % m,
- если она уже занята, то рассматривается (h1(k)+h2(k)) % m,
- затем (h1(k)+2 · h2(k)) % m и так далее.
- В общем случае: (h1(k)+i · h2(k)) % m где i=(0,1,...,m−1)

Двойное хеширование

- в лучшем случае выполняются за O(1),
- в худшем — за O(m),
  - не отличается от обычного линейного разрешения коллизий.
- среднем, при грамотном выборе хеш-функций, двойное хеширование будет выдавать лучшие результаты, за счёт того, что вероятность совпадения значений сразу двух независимых хеш-функций ниже, чем одной.

Двойное хеширование

h1 может быть обычной хеш-функцией.

Чтобы последовательность могла охватить всю таблицу, h2 должна возвращать значения:

- не равные 0
- независимые от h1
- взаимно простые с величиной хеш-таблицы

Пример рехеширования

Условия:

1. **Хеш-функция**: Пусть хеш-функция h(x)h(x) выдаёт значения от 0 до 20.
2. **Начальный размер таблицы**: N=4.
3. **Элементы**: Пусть изначально в таблице находятся элементы с хеш-значениями 2, 6, 10, 14, 18.

индекс = h(x) % 4

Как будут расположены элементы?

$$h(2) \mod 4 = 2$$

$$h(6) \mod 4 = 2$$

$$h(10) \mod 4 = 2$$

$$h(14) \mod 4 = 2$$

$$h(18) \mod 4 = 2$$

Увеличиваем размер таблицы вдвое: N=8

$$h(2) \mod 8 = 2$$

$$h(6) \mod 8 = 6$$

$$h(10) \mod 8 = 2$$

$$h(14) \mod 8 = 6$$

$$h(18) \mod 8 = 2$$

	Открытая адресация	Метод цепочек
Использование памяти	Экономичнее, так как используется один массив.	Требуется дополнительная память для списков.
max load factor **α**	1	>1
Доступ для худшего случая	Линейное опробование: код единственная свободная ячейка находится перед ячейкой, куда предполагалось вставить элемент O(n)	Цепочка - список Все элементы попадают в одну ячейку O(n)
Среднее количество операций, при успешном поиске	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$ $\alpha \neq 1$	$1+\frac{\alpha}{2}$
Среднее количество операций, при безуспешном поиске	$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$	$e^{-\alpha}+\alpha$

Реализация

Операции HashTable

- **insert(key):** Insert *key* into the **Hash Table** (no duplicates)
- **find(key):** Return true if *key* exists in the **Hash Table**, otherwise return false
- **remove(key):** Remove *key* from the **Hash Table** (if it exists)
- **hashFunction(key):** Produce a hash value for *key* to use to map to a valid index
- **key_equality(key1, key2):** Check if two keys *key1* and *key2* are equal

// Хеш-таблица

```
class HashTable: public List<E> {
 // List<E> – абстрактный класс
 protected:
   // число блоков; представляет размер таблицы
   size_t numBuckets;

   // хеш-таблица есть массив связанных списков
   Array< LinkedList<E> > buckets;

   // хеш-функция и адрес элемента данных,
   // к которому обращались последний раз
   size_t (*hf) (E key);
   // todo: объявить отдельный тип для функции

   T *current;

public:
 // конструктор с параметрами, включающими
 // размер таблицы и хеш-функцию
 HashTable(size_t nbuckets, size_t hashf(E key));

 // основные методы
 virtual void Insert (const E& key);
 virtual int  Find   (      E& key);
 virtual void Delete (const E& key);
 virtual void Clear  ();
         void Update (const E& key);

 // дружественный итератор, имеющий доступ к
 // данным-членам
 friend class HashTableIterator<E>
};
```

Пример объявления класса из [1]

```
// Абстрактный класс для коллекций

template <class T> class List{
   // абстрактный класс для коллекций с операциями:
   // размер, проверка на пустоту, поиск, вставка, удаление, очистка
   protected:
       // фактическое число элементов коллекции,
       // обновляемое производным классом
       size_t _size;

   public:
       // конструктор
       List ();

       /// количество элементов
       virtual int size() const;
       /// коллекция пуста?
       virtual bool is_empty() const;
       /// поиск элемента item,
       /// возвр. -1 если не найден или индекс, если найден
       virtual long long find (T& item) = 0;

       /// вставка
       virtual void insert (const T& item) = 0;
       /// удаление элемента
       virtual void delete (const T& item) = 0;
       /// удаление элементов из всей коллекции
       virtual void clear  () = 0;

};
```

Итератор (пример из [1])

```
template <class T>
class HashTableIterator : public Iterator<T> {
private:
   // указатель на таблицу, подлежащую обходу
   HashTable<T> *HashTable;

   // индекс текущего просматриваемого блока
   // и указатель на связанный список
   size_t currentBucket;
   LinkedList<T> *currBucketPtr;

   // утилита для реализации метода Next
   void SearchNextNode(int cb);

public:
   // конструктор
   HashTableIterator(HashTable<T> &ht);

   // базовые методы итератора
   virtual void Next();
   virtual void Reset();
   virtual    T &Data();

   // подготовить итератор для
сканирования другой таблицы
   void SetList(HashTable<T> &lst);
};
```

Алгоритмы поиска

Алгоритм поиска	Преимущества	Недостатки	Случаи применения
Послед.	- Простой - Работает с любыми структурами - Не требует сортировки	- Медленный (**O(n)**) если n большое или поиск нужен часто	- Поиск в небольших или несортированных наборах данных
Бинарный поиск	- Быстрый на больших объемах данных (**O(log n)**) - Простая реализация	- Требуется предварительная сортировка данных (обычно O(n log n) - Неэффективен для динамически изменяемых наборов данных	- Поиск в отсортированных массивах - Применение в задачах, где важна скорость поиска
В бинарном дереве	- Быстрая скорость поиска, вставки и удаления в среднем (**O(log n)**) - Эффективное использование памяти	- Требует памяти для указателей - Может деградировать до O(n) при несбалансированном дереве - Сравнительно сложно реализовать самобаланс. дерево	- Динамически изменяемые наборы данных - Поиск, требующий сохранения порядка элементов
в хеш-таблице	- Очень высокая скорость поиска, вставки и удаления в среднем (**O(1)**) - Независимость от порядка данных	- Возможны коллизии, требующие дополнительных методов обработки - Требует дополнительной памяти для хранения	- Быстрый поиск ключей в больших наборах данных - Применение в реализациях кэшей, словарей, индексах баз данных

Хеш-таблицы в библиотеках

Пример C++

`std::unordered_set` использует хеш-таблицу для хранения значений, поэтому проверка наличия значения в коллекции очень быстрая.

Значения здесь являются и ключами.

en.cppreference.com/w/cpp/container/unordered_set

```cpp
#include <unordered_set>
void print(const auto& set){
   for (const auto& elem : set) std::cout << elem << ' ';
   std::cout << '\n'; }

int main(){
   // set - неупорядоченное множество, использует хеш-таблицу для хранения значений
   std::unordered_set<int> mySet{2, 7, 1, 8, 2, 8}; // creates a set of ints
   print(mySet);

   mySet.insert(5); // puts an element 5 in the set
   print(mySet);

   if (auto iter = mySet.find(5); iter ≠ mySet.end())
       mySet.erase(iter); // removes an element pointed to by iter

   print(mySet);
   mySet.erase(7); // removes an element 7
}
```

std::unordered_set

std::hash — специальный класс, с методом

size_t operator()(const Key &k) const

en.cppreference.com/w/cpp/utility/hash

```
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,          // функция сравнения ключей
    class Allocator = std::allocator<Key>
> class unordered_set;
```

en.cppreference.com/w/cpp/container/unordered_set

std::hash — тип для хеш-функций

```cpp
#include <functional>

// возьмём стандартную реализацию класса hash для строк
std::hash<string> hs = std::hash<string>();

cout << hs("") << "\n";          // 6142509188972423790
cout << hs("1") << "\n";         // 10159970873491820195

cout << hs("War is peace. Freedom is slavery. Ignorance is strength.") << "\n";
// 14655639112963079906
```

std::hash — пример определения варианта класса

```
class Book{
  public:
      string title;
      string author;
      size_t pages;
  // todo: ...
  string to_string() const { return std::format("{}, {}, {} p.", title, author, pages);}
};

// Определение нового варианта шаблонного класса hash
// для нового типа Book
namespace std {
     template <> struct hash<Book>{
          size_t operator()(const Book & b) const  {
               return std::hash<string>()(b.to_string());    }
     };
}

  Book b {"1984", "George Orwell", 320};
  auto h2 = hash<Book>();
  cout << h2(b);       // 5663253459914386003
```

В других языках программирования

- Python3:
  set, frozenset, and dict.
  Ключ должен быть хешируемым (hashable). Можно переопределить метод __hash__ для класса ключа, чтобы сделать его хешируемым.
- Java:
  Словарь HashMap (с методом цепочек, но если список имеет большую длину (>8) используются самобалансирующиеся деревья.
  Для класса ключа нужно переопределить метод hashCode()
  Ключи в HashMap должны быть неизменяемыми типами (immutable)

  множество HashSet – на основе HashMap

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

Словарь

Хеш таблица и словарь

- Хеш таблица хранит только ключи
  Это удобно если нужно определить, находится ли ключ в коллекции.
- Но не подходит, если нужно хранить что-то кроме ключа. Например хранить и номер сотрудника (ключ) и его персональные данные.
- В словаре с каждым ключём связан элемент – значение.
  Другие название словарей (dict) – ассоциативные массивы – map

Операции со Словарём

- **put(key,value)** – выставка, добавляет новую пару (key,value) или обновляет значение для данного ключа;
  при обновлении может возвращать старое значение ключа, при вставке — NULL
- **get(key)** → value или NULL (ключ не найден)
- **remove(key)** – удаляет пару (key,value), возвращает value или NULL (ключ не найден)
- **size()** → количество пар (key,value)
- Опционально:
  - *keys() → набор ключей*
  - *values() → набор значений*

Реализация на основе хеш-таблицы

- Хеш таблица будет хранить на ключи, а пары (ключ, значение)
- **put(key,value)** – вставка
  хеш функция вычисляется только для ключа
  в таблицу вставляется пара (ключ, значение)
- get(key)
  хеш функция вычисляется только для ключа
  выдает значение
- remove(key)
  хеш функция вычисляется только для ключа
  удаляет (ключ,значение)

Java HashMap

- insert(key,value) — вставка, при замене возвращает прежнее значение, иначе null
- find(key) — возвращает значение, можно реализовать оператором [ ]
- remove(key) — удаляет пару (*key*, *value*), возвращает value, если пара не найдена возвращает null
- hashFunction(key) — хеш-функция
- key_equality(key1, key2) — возвращает true, если хеши ключей равны
- size() – размер коллекции (количество пар)
- isEmpty() –коллекция пуста?

Пример

`std::unordered_map` использует хеш-таблицу для хранения значений и ключей.

en.cppreference.com/w/cpp/container/unordered_map

Реализация в STL

std::map. Основные операции

```cpp
#include <map>

    std::map<std::string, int> m{{"CPU", 10}, {"GPU", 15}, {"RAM", 20}};

// изменение и вставка
    m["CPU"] = 25; // update an existing value
    m["SSD"] = 30; // insert a new value

    // Using operator[] with non-existent key always performs an insert
    std::cout << "3) m[UPS] = " << m["UPS"] << '\n';     // + IPS, 0

    m.erase("GPU");

    std::erase_if(m, [](const auto& pair){ return pair.second
```

Добавляется со значением по умолчанию (вызовом конструктора по умолчанию для значения)

```cpp
    std::cout << "7) m.size() = " << m.size() << '\n';

    m.clear();
```

en.cppreference.com/w/cpp/container/map:
“Maps are usually implemented as Red-black trees”

std::map. Итерирование

C++11

```
for (const auto& n : m)
    cout << n.first << " = " << n.second << "; ";
```

Аналогично:

```
    for (const pair<const string, int>& n : m)
        cout << n.first << " = " << n.second << "; ";
```

C++17

```
    for (const auto& [key, value] : m)
        std::cout << '[' << key << "] = " << value << "; ";
```

std::unordered_map

```
// Create an unordered_map of three strings (that map to strings)
std::unordered_map<std::string, std::string> u = {
      {"RED", "#FF0000"},
      {"GREEN", "#00FF00"},
      {"BLUE", "#0000FF"}      };

cout << "size: " << u.size() << "\n";                    // 3
cout << "load factor: " << u.load_factor() << "\n";      // 0.23

u["ZabGU"] = "#326698";

cout << "load factor: " << u.load_factor() << "\n";      // 0.3
cout << "size: " << u.size() << "\n";                    // 4
```

Размер массива увеличится во время добавления нового элемента, когда load factor = 1

https://en.cppreference.com/w/cpp/container/unordered_map
Search, insertion, and removal of elements have average constant-time complexity

std::unordered_map

```
// Create an unordered_map of three strings (that map to strings)
std::unordered_map<std::string, std::string> u = {
      {"RED", "#FF0000"},
      {"GREEN", "#00FF00"},
      {"BLUE", "#0000FF"}      };

cout << "size: " << u.size() << "\n";                    // 3
cout << "load factor: " << u.load_factor() << "\n";      // 0.23

// Helper lambda function to print key-value pairs
auto print_key_value = [](const auto& key, const auto& value){
       std::cout << "Key:[" << key << "] Value:[" << value << "]\n"; };

std::cout << "Iterate and print key-value pairs of unordered_map, being\n"
               "explicit with their types:\n";

for (const std::pair<const std::string, std::string>& n : u)
       print_key_value(n.first, n.second);
```

std::unordered_map

```
// Create an unordered_map of three strings (that map to strings)
unordered_map<string, string> u = {
   {"RED", "#FF0000"},
   {"GREEN", "#00FF00"},
   {"BLUE", "#0000FF"}     };


cout << "\nIterate and print key-value pairs using C++17 structured binding:\n";

for (const auto& [key, value] : u)
   print_key_value(key, value);
```

C++ Standard Library Associative Containers

h/cpp hackingcpp.com

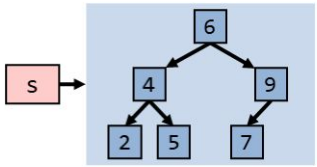
set<Key>

unique, ordered keys

multiset<K>

(non-unique) ordered keys

```
std::set<int> s;
s.insert(7); …
s.insert(5);
auto i = s.find(7);  // → iterator
if(i != s.end())     // found?
  cout << *i;        // 7
if(s.contains(7)) {…}     C++20
```



usually implemented as balanced binary tree (red-black tree)

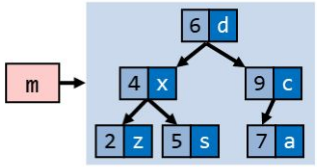
map<Key,Value>

unique key → value-pairs; ordered by keys

multimap<K,V>

(non-unique) key → value-pairs, ordered by keys

```
std::map<int,char> m;
m.insert({7,'a'});  …
m[4] = 'x';         // insert 4→x
auto i = s.find(7); // → iterator
if(i != s.end())    // found?
  cout << i->first  // 7
       << i->second; // a
if(s.contains(7)) {…}     C++20
```



usually implemented as balanced binary tree (red-black tree)

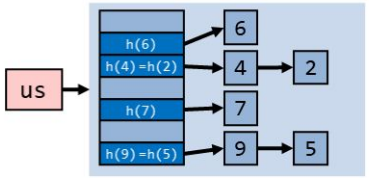
unordered_set<Key>

unique, hashable keys

unordered_multiset<Key>

(non-unique) hashable keys

```
std::unordered_set<int> us;
us.insert(7); …
us.insert(5);
auto i = us.find(7); // → iterator
if(i != us.end())    // found?
  cout << *i;        // 7
if(s.contains(7)) {…}     C++20
```



hash table for key lookup, linked nodes for key storage

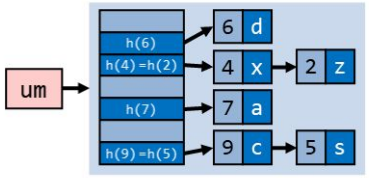
unordered_map<Key,Value>

unique key → value-pairs; hashed by keys

unordered_multimap<Key,Value>

(non-unique) key → value-pairs; hashed by keys

```
unordered_map<int,char> um;
um.insert({7,'a'});  …
um[4] = 'x';         // insert 4→x
auto i = um.find(7); // → iterator
if(i != um.end())    // found?
  cout << i->first   // 7
       << i->second; // a
if(s.contains(7)) {…}     C++20
```



hash table for key lookup, linked nodes for (key, value) pair storage

hackingcpp.com/cpp/std/associative_containers.png

Реализация словаря

Хеш таблица

- Использует массив
- \+ вставка, поиск, удаление — O(1) в среднем
- \- Нужно резервировать память
- \- Необходимость увеличивать размер массива, переносить элементы в новый массив

Самобалансирующиеся дерево

- Структура данных на основе узлов
- \- вставка, поиск, удаление — log(n) в среднем
- \+ Не нужно резервировать память
- \+ Не нужно перекопировать элементы при перевыделении памяти

Реализация словаря на основе BST

```
template <class K, class T>
class KeyValue {
protected:
   // после инициализации ключ не может быть изменен
   const K key;
public:
   // словарные данные являются общедоступными
   T value;
   KeyValue (K KeyValue, T datavalue);

   // операторы присваивания. не изменяют ключ
   KeyValue<K,T>& operator= (const KeyValue<K,T>& rhs);

   // операторы сравнения. сравнивают два ключа
   int operator== (const KeyValue<K,T>& value) const;
   int operator== (const K& keyval) const;
   int operator< (const KeyValue<K,T>& value) const;
   int operator< (const K& keyval) const;

   // метод доступа к ключу
   K Key (void) const;
};
```

Нет конструктора по умолчанию

Сравнение только ключей

пример из [1]

Словарь на основе BST

```
template <class K, class T>
class Dictionary: public BinSTree< KeyValue<K,T> >{
   // значение, присваиваемое элементу словаря по умолчанию.
   // используется оператором индексирования, а также методами
   // InDictionary и DeleteKey T defaultValue;
private:
   T defaultValue;
public:
   // конструктор
   Dictionary (const T& defval);
   // оператор индексирования
   T& operator[] (const K& index);
   // дополнительные словарные методы
   int InDictionary (const K& keyval);
   void DeleteKey(const K& keyval);
};
```

Ищет элемент типа KeyValue в дереве, если найден, то возвращает ссылку

пример из [1]

Множества

Множества

Множества на основе битовых массивов
docs.google.com/presentation/d/1e4ik9m7bWxDKmJfleoDN0XXzaF_DAx7W5EYHnyczag8/edit#slide=id.g2ce3bbf7a23_0_100

Множество на основе HashTable

```
template <typename E>
class HashSet {  // todo: наследование от абстрактного класса множество
private:
   // Внутренняя хеш-таблица для реализации множества
   HashTable<E> hashTable;
public:
   // Конструктор множества
   HashSet(size_t nbuckets, size_t (*hashf)(E key))
       : hashTable(nbuckets, hashf) {}

   // Добавление элемента в множество
   void Insert(const E& element) {           hashTable.Insert(element);    }

   // Проверка, есть ли элемент в множестве
   bool Contains(const E& element) {  return hashTable.Find(element) ≠ -1;  }

   // Удаление элемента из множества
         void Remove(const E& element) {      hashTable.Delete(element);         }

   // Очистка множества
   void Clear() {                            hashTable.Clear();    }

 // todo: итератор по множеству
};
```