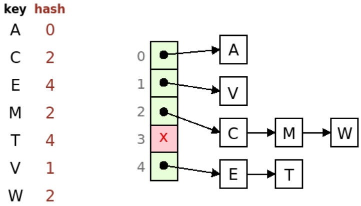
Hash Tables

L.EIC

Algorithms and Data Structures

2025/2026



P Ribeiro, AP Tomas

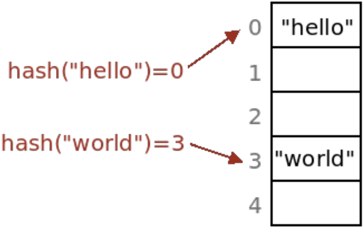
Motivation

- Consider the following simple problem: store a **set of** n **integer numbers** x_i with $0 \leq x_i < m$ and support **search**, **insert** and **remove**
- How can we solve this? What time complexities can we obtain?
  - We could use **balanced BSTs** and guarantee $\mathcal{O}(\log n)$
  - Can we do better than logarithmic time?
- We could use a **boolean array of size** m and have $\mathcal{O}(1)$ operations!
  Example: $S = \{1, 4, 95, 96, 99\}$, $m = 100$

i	0	1	2	3	4	5	...	94	95	96	97	98	99
a[i]	F	T	F	F	T	F	...	F	T	T	F	F	T

  - search(x): return position x
  - insert(x): change position x to *true*
  - remove(x): change position x to *false*
- Can we **generalize** this type of solution?
  - What if m is really large? (not enough memory...)
  - What if the keys we want to store are not integers?

Hash Tables: Key Ideas



- Save items in a **key-indexed table** (index as function of the key)
- **Hash function:** method for computing array index from key.
- Main **issues**:
  - How to compute the hash function?
  - How to handle collisions? (keys that hash to the same array index)
- Classic **space-time tradeoff**:
  - No space limitation: trivial hash function with key as index
  - No time limitation: trivial collision resolution with sequential search
  - Space and time limitations: hashing (*the real world...*)

Hash Functions: Goals

key → **hash function** → table index

- **Goals**:
  - Should be **efficiently computable**

    If we spend too much time computing, it defeats our purpose
  - Should **minimize collisions**
    - ★ It should spread the values along the table; ideally, each index should be equally likely, that is, keys are uniformly distributed
    - ★ It should use all bits of the key (otherwise almost equal keys will collide)

Example of a **"bad" hash function** for strings:

$$hash(string) = length(string)$$

All equally sized strings would hash to the same value, regardless of their content

string	hash
"john"	4
"far"	3
"life"	4
"dog"	3

0	
1	
2	
3	"far" / "dog"
4	"john" / "life"

Hash Functions: Modular Hashing

- Can be implemented in (general use) hash tables in two steps:
  1. $h = hash(key)$ *(hash could be a really large positive number)*
  2. $index = h \,\%\, table\_size$ *(convert to the size of the table, $\% =$ mod)*

- The 1st step guarantees we can use the same *hash* function for different table sizes

- The 2nd step is known as **modular hashing**, also know as **division hashing** (the image on the left shows an example)

key	key % 5
15	0
42	2
39	4
81	1

0	15
1	81
2	42
3	
4	39

- For a general case, we usually choose **a prime number as the table size**
  - Due to the mathematical properties of modular arithmetic, this might help to avoid collisions if the keys follow a biased distribution (see **this**, for example)
  - If keys follow an uniform distribution this does not matter
  - Even prime number sizes are "exploitable" if an "attacker" knows the exact hash function and table size (see **this**, for example)

Hash Functions: Implementation

- **How to create an hash function?** (assuming for now it should return an unsigned integer to be used with the modular hashing as described before)
- We will now show some **(naive) examples** of possible hash functions.
- For an **integer key** we could just trivially use the identity function

```
unsigned myHash(int i) {
  return i;
}
```

```
cout << myHash(42) << endl;
cout << myHash(-42) << endl;
```

```
42
4294967254
```

(wait: a negative value interpreted as a positive? Yes, here we are really using *underflow* arithmetic, while using all the bits. An alternative such as using *abs(i) would actually not use the entire bitspace and attribute the same hash to a number and its negative...)*

Hash Functions: Implementation

- For the **string type** we could use **polynomial hashing**:
  - A string of size k can be seen a sequence of chars $c_0, c_1, \ldots, c_{k-2}, c_{k-1}$
  - A char can be interpreted as an integer (its ascii code)
  - We choose a non-zero constant a and compute the hash as:
    $c_0 a^{k-1} + c_1 a^{k-2} + \ldots + c_{k-2} a^1 + c_{k-2}$
    (this is similar to how we interpret $1234 = 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4$, but we will be using base a instead of base 10)
  - For reasons similar to the modular hashing, choosing a as a prime might be a good choice (see **this**, for example)
  - To reduce the number of multiplications we can use **Horner's Rule**.
    (e.g. $1234 = 4 + 10 \times (3 + 10 \times (2 + 10 \times 1)))$

```
unsigned myHash(string s) {
  unsigned hash = 0;
  for (int i=0; i<(int)s.length(); i++)
    hash = 31 * hash + s[i]; // Horner's Rule
  return hash;
}
```

(we are also ignoring overflows - this is equivalent to always applying $\mod 2^{32}$ given how an unsigned is interpreted, assuming unsigned uses 32 bits)

Hash Functions: Implementation

- What if we need to hash several types (for instance a **vector** or a **class with several attributes?**)
- We could use **polynomial hashing** to combine elements or another operation such as a **XOR** (exclusive or) (see **this** for example)
  (XORing two numbers with roughly random distribution results in another number still with roughly random distribution, but which now depends on the two values)

```
class Person {
  public:
    string name;
    int age;
    Person(string n, int a) {name=n; age=a;}
};

unsigned myHash(Person p) { // naive combine
  return myHash(p.name) ^ myHash(p.age);
}
```

```
Person p("John", 42);
cout << myHash(p) << endl;
```

```
3904197
```

Hash Functions in C++ standard

- std::hash is a template with multiple specializations for common types
  It produces a *size_t* integer (64 bits on a typical 64-bit computers)

```
hash<int> hi;
hash<double> hd;
hash<string> hs;

cout << "hash(42) = " << hi(42) << endl;
cout << "hash(3.14) = " << hd(3.14) << endl;
cout << "hash(\"hello\") = " << hs("hello") << endl;
```

```
hash(42) = 42
hash(3.14) = 5464867211497793177
hash("hello") = 2762169579135187400
```

(implementation may vary from compiler to compiler; gcc with C++11 std::hash<string> used a variant of the murmur family of hash functions)

- For a more robust combine function see for instance boost::hash_combine

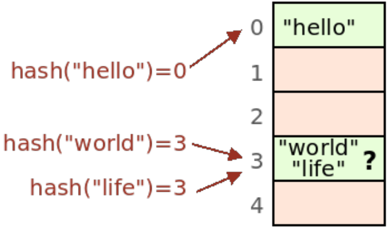
```
seed ^= hash_value(v) + 0x9e3779b9 + (seed << 6) + (seed >> 2);
```

Hash Functions

- Out of scope to do an in-depth analysis of hashing in this class
- There are **years of research** and a multitude of hash functions
- Hash functions have **other applications** besides hash tables
  - Other data structures such as **bloom filters**
  - Algorithms such as **Rabin-Karp Algorithm**
  - **Cryptographic hash functions** (e.g. for file integrity verification)
- There is **no perfect** practical *"one size fits them all"* hash function
  - It depends on the distribution of the keys, machine architecture, etc
  - Here are some empirical comparisons of existing hash functions: smhasher, strchr.com, stackexchange
- If you know the exact keys beforehand, you can devise a "perfect" hash function (for instance, using GNU's gperf tool)

Collisions in Hash Tables

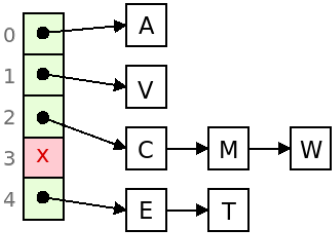
- **Collisions:** distinct keys hashing to the same index
- Collisions are almost inevitable...
- **Challenge:** how to deal with collisions efficiently



Collision Strategy: Separate Chaining

- **Separate Chaining** (also known as *open hashing*)
  Use an **array of lists**
  - **Hash:** map key to integer between 0 and $table\_size - 1$
  - **Insert:** put in front of i-th chain (if not already there)
  - **Search:** need to search only i-th chain
  - **Delete:** remove from i-th chain

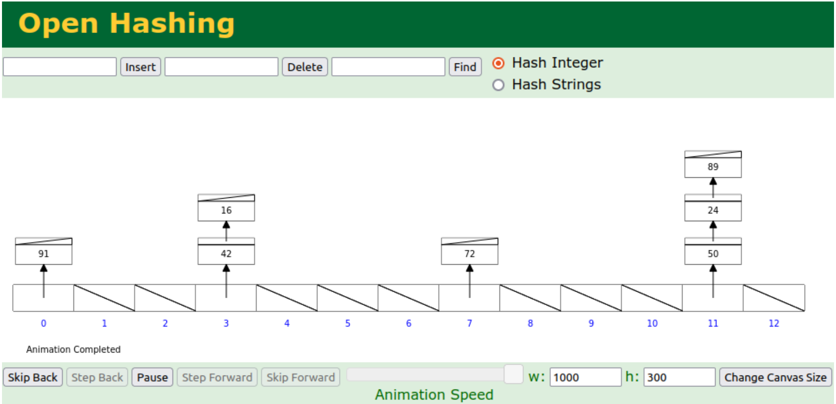
key	hash
A	0
C	2
E	4
M	2
T	4
V	1
W	2



Visualizing Separate Chaining

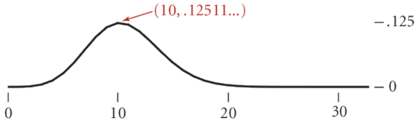
- You can try the indicated url:

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



Separate Chaining: Analysis

- Let's define the **load factor** of an hash table as $\lambda = \frac{n}{m}$
  - n: number of keys stored
  - m: size of hash table
- The **average size of a list** will be λ
- Let's assume that the hash function uniformly distributes keys.
  In this case the probability that a list size is within a constant factor of λ is extremely close to 1 (list size follows a binomial distribution)

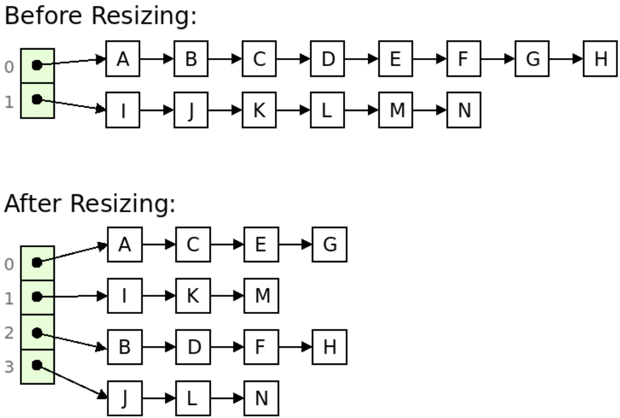


Binomial Distribution ($n = 10^4, m = 10^3, \lambda = 10$)

- **Consequence**: Number of probes for search is proportional to λ.
  - m too large $\rightarrow$ too many empty chains
  - m too small $\rightarrow$ chains too long
  - Typical choice: $\lambda \sim 4 \rightarrow$ constant-time ops (can be less if you can afford the space)

Separate Chaining: Resizing

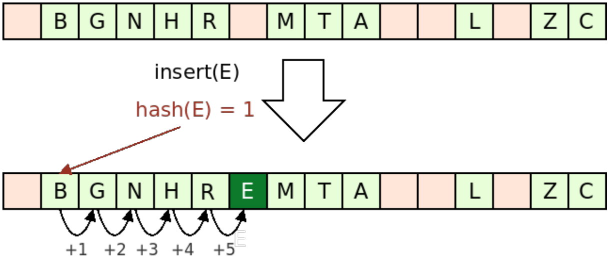
- Keep λ close to 4
  - Double size of array m when $\lambda = n/m \geq 8$
  - Halve size of array m when $\lambda = n/m \leq 2$
  - Need to rehash all keys when resizing



Collision Strategy: Open Addressing

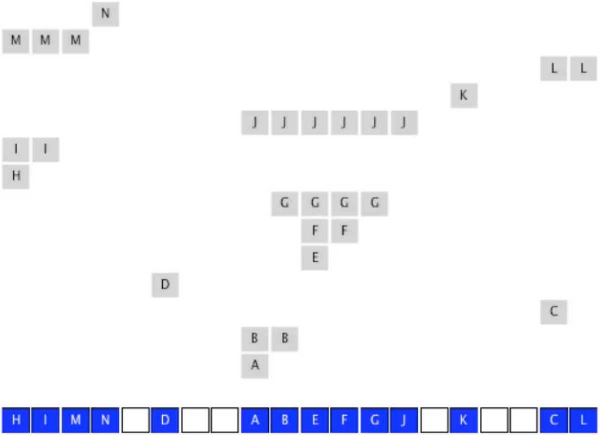
- **Open Addressing** (also known as *closed hashing*)
  Store keys on array. When a new key x collides, find an empty slot, and put it there.

- **Linear Probing:** to find an empty slot traverse consecutive positions starting on the hashed index



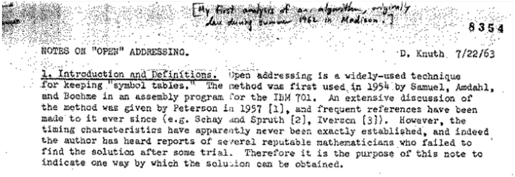
Linear Probing and Clustering

- **Cluster**: A contiguous block of items.
- New keys likely to hash into middle of big clusters (and to close gaps between clusters, forming even bigger clusters)...



Linear Probing Analysis

- Assuming a uniform distribution of keys, the average number of probes in linear probing is:
  - **Search hit:** $\sim \frac{1}{2}(1 + \frac{1}{1-\lambda})$
  - **Search miss/insert:** $\sim \frac{1}{2}(1 + \frac{1}{(1-\lambda)^2})$

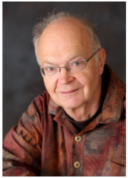


[My first analysis of an algorithm, originally done during summer 1962 in Madison.]

8354

NOTES ON "OPEN" ADDRESSING. D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



- **Table size** needs to be bigger than the number of keys ($m > n$)
  - m too large $\rightarrow$ too many empty array entries
  - m too small $\rightarrow$ search time blows up
  - Typical choice: $\lambda = \frac{1}{2}$

    # probes for search hit is about 3/2, # probes for search miss is about 5/2

Linear Probing: Resizing

- A possible **resize strategy**: keep $\lambda < 0.5$
  - Double size of array m when $\lambda = n/m \geq \frac{1}{2}$
  - Halve size of array m when $\lambda = n/m \leq \frac{1}{8}$
  - Need to rehash all keys when resizing

Before Resizing:

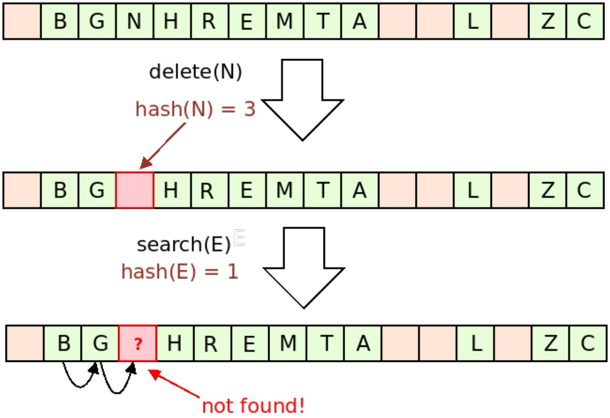
	A	B			C	D	
0	1	2	3	4	5	6	7

After Resizing:

	A				C				B				D		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

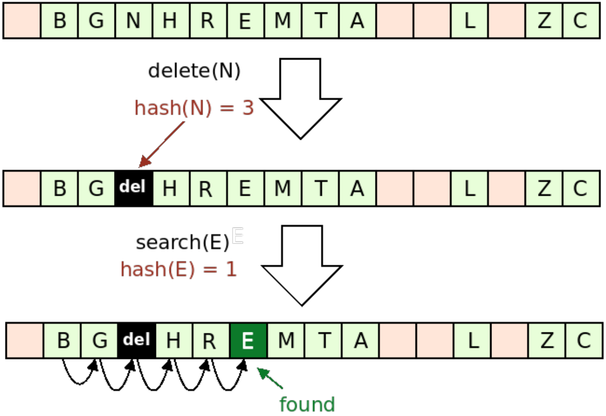
Linear Probing: Deletion

- **Deleting** a key requires some care
  - We can't just delete completely the array entry and do nothing else, as this could invalidate future searches



Linear Probing: Lazy Deletion

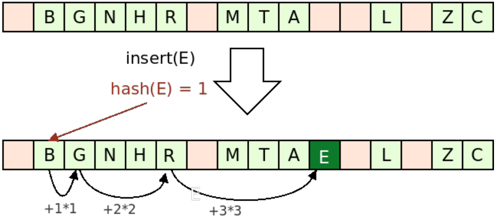
- **Idea:** leave a marker (*tombstone*) saying it was deleted so that linear probing can pass through it



- Tombstones still count towards the load factor λ
- When inserting you can occupy these indexes

Open Addressing: Other Probing Strategies

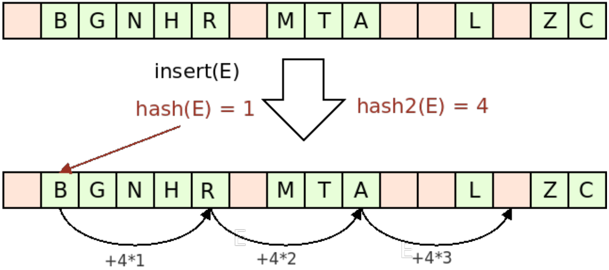
- Linear probing could be framed in a more general sequential probing framework: sequentially probe positions $H_1(x), H_2(x), H_3(x), \ldots$
- $H_i(x) = (hash(x) + f(i)) \ \% \ m$
- Under this assumption, **Linear Probing** is using $f(i) = i$
- Another possible strategy could be **Quadratic Probing:** $f(i) = i^2$



  - This eliminates "primary" clustering as existed with linear probing
  - However, it might not find an empty cell if table more than half full

Open Addressing: Other Probing Strategies

- We could also use **Double Hashing:** $f(i) = i \times hash\_function_2(x)$

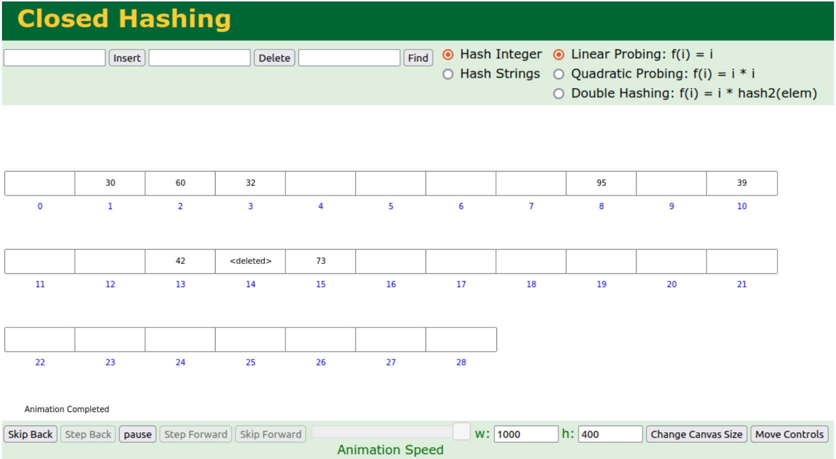


  - Effectively avoids clustering
  - Capable of using full table
  - However, it needs another good "independent" hash function and increases the cost of the hash computation

Visualizing Open Addressing

- You can try the indicated url:

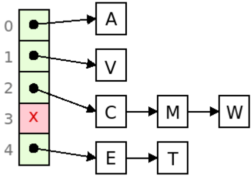
  https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html



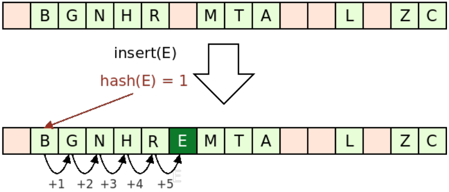
Separate Chaining vs Open Addressing

- **Separate Chaining**
  (a.k.a. *open hashing*)
  - Performance degrades gracefully
  - Clustering less sensitive to poorly-designed hash function

key	hash
A	0
C	2
E	4
M	2
T	4
V	1
W	2



- **Open Addressing**
  (a.k.a. *closed hashing*)
  - Less wasted space
  - Better cache performance



Hash Tables vs Other Data Structures

- Hash Tables can provide **constant time operations in an amortized sense** (amortized means on average even on worst possible sequence) but are very sensitive to several factors (e.g. hash function used and keys distribution)

method	**guarantee (worst case)**			**average case**			**ordered ops?**	**core interfaces**
	search	**insert**	**delete**	**search**	**insert**	**delete**		
Sequential Search (unordered list)	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	no	equality
Binary Search (ordered array)	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	yes	comparator
Balanced BST	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	yes	comparator
Hash Tables	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	no	equality hash

- **Hash Tables:**
  - Simpler to code
  - No effective alternative for unordered keys
  - Faster for simple keys (a few arithmetic ops versus $\log n$ compares)
- **Balanced BSTs:**
  - Stronger performance guarantees
  - Support for ordered operations (e.g. max, min, lower_bound, ordered iterators)
  - Easier to implement comparison correctly than equality and hash function

Hash Tables in C++ Standard

- **(Unordered) Associative Containers**
  - `unordered_set` - collection of unique keys
  - `unordered_map` - collection of key-value pairs, keys are unique
  - `unordered_multiset` - collection of keys
  - `unordered_multimap` - collection of key-value pairs

- Usual (non-ordered) operations are available:
  - Non-ordered iterator over keys
  - Lookup (find)
  - Modifiers (clear, insert, erase)

- Hash table related operations (e.g.: rehash, load_factor, bucket):

- Template class that relies on two key functions: **hash_function** and **key_eq** (already implemented for common types)

- Current implementations use **separate chaining**

Example Usage

- Let's use a **real dataset** to play a little bit
- Suppose you have a dictionary with words on a file `words.txt` (in my case 370 103 words)

```
claps
snoops
agglutination
...
```

```cpp
// Example that reads all strings from stdin and prints them (one per line)
string w;
while (cin >> w) {
  cout << w << endl;
}
```

Example compilation using gcc:

```
g++ -o example example.cpp
```

Example execution (< redirects stdin, ./ indicates current dir)

```
./example < words.txt
```

Example Usage

- Let's insert all the words into a hash table and check the load factor

```
unordered_set<string> ht;
string s;
while (cin >> s) {
  ht.insert(s);
}

cout << "nr keys: " << ht.size() << endl;
cout << "load factor: "<< ht.load_factor() << endl;

ht.rehash(400000); // rehash to at least 400 000 positions
cout << "load factor: "<< ht.load_factor() << endl;

ht.rehash(1000000); // rehash to at least 1 000 000 positions
cout << "load factor: "<< ht.load_factor() << endl;
```

```
nr keys: 370103
load factor: 0.519299
load factor: 0.900807
load factor: 0.350369
```

Example Usage

- Let's check if a word exists and test the erase method

```
string s = "algorithm";

if (ht.find(s) != ht.end()) cout << "Found: " << s << endl;
else cout << "Not found: " << s << endl;

cout << "Erasing: " << s << endl;
ht.erase(s);

if (ht.find(s) != ht.end()) cout << "Found: " << s << endl;
else cout << "Not found: " << s << endl;
```

```
Found: algorithm
Erasing: algorithm
Not found: algorithm
```

Example Application

- Let's try to determine the frequency of word terminations of size k

  For instance, for the word "algorithm" its termination of size 4 is "ithm"

```
int k = 4; // size of the word termination
string w;
unordered_map<string, int> ht; // associating frequency to termination
while (cin >> w) { // read words from standard input as before
  int len = w.length();
  if (len>=k) { // Only words with at least k chars matter
    string tmp = w.substr(len-k, k); // Extract last k chars
    if (ht.find(tmp) == ht.end()) ht[tmp] = 1; // new termination
    else ht[tmp]++; // already existing termination, increment count
  }
}
cout << "Some example frequencies:" << endl;
cout << "less " << ht["less"] << endl;
cout << "ting " << ht["ting"] << endl;
cout << "ally " << ht["ally"] << endl;
```

```
Some example frequencies:
less 1845
ting 3731
ally 4316
```

Example Application

- What if we now want to extract the 5 most frequent terminations?
  - No order on our hash table (and key is termination)
- **Idea:** combine with BSTs that provide order!
  - Use the frequency as key and the word as the value
  - Use a *multimap* (there can be several keys with the same frequency)

```
multimap<int, string> mm; // associate frequencies to words
// iterate over pairs (key, value)
for (auto i : ht) mm.insert({i.second, i.first});

// reverse iterator:  start at the maximum and not the minimum
// (natural order is increasing)
auto ri = mm.rbegin();
for (int i=0; i<5; i++) { // assuming there are at least 5
        cout << ri->second << " " << ri->first << endl;
        ri++;
}
```

```
ness 9564
tion 7245
able 4609
ally 4316
ting 3731
```

Final Notes

- **Associative Containers** (**BSTs** and **Hash Tables**) are powerful data structures that should be part of your algorithmic arsenal
- **C++** provides *ready to use* **implementations** of both
(but knowing them is important to understand what and how to use)
(and in some cases you might need a customized data structure)
- We only covered the essentials of these topics and in both there is much more to know: **never stop learning, as the algorithmic and data structures landscape is always evolving**!

Christmas Fun

- https://adventofcode.com/ - **Advent of Code**

Advent of Code is an annual set of **Christmas-themed computer programming challenges** that follow an Advent calendar. It has been running since 2015.

The programming puzzles cover a **variety of skill sets and skill levels** and **can be solved using any programming language**. Participants also compete based on speed on both global and private leaderboards.

The event was founded and is maintained by software engineer Eric Wastl.