

```
>>> 15 minutes talk.
```

```
>>> Hashtable.
```

```
Name:  Konstantin Pachuev.
```

```
Date:  July 11, 2016
```

```
>>> Outline >>>
```

1. Statement.
2. Structure.
3. Implementation.
4. Runtime Analysis.

>>> Statement of Work >>>

Implement your own `Hastable<K, V>` class without using any library classes that are already available for the language or platform you are using to implement the hashtable. E.g., implement the hashtable using just simple arrays. The hashtable should support methods to add and remove elements. You are free to choose either hashing with chaining or hashing with open addressing for your implementation. Please also provide tests. Tell us the runtime of each class method using the O calculus.¹

¹<https://www.esrlabs.com/coding-assignments/>

>>> Requirements. I >>>

1. User's.

- 1.1 Hashtable must store elements, provide insertion, retrieval and deletion of elements with basic or user defined types of keys and values.
- 1.2 Hashtable should provide unified interface for user defined hash function.

2. Functional.

- 2.1 Insertion must add an element to Hashtable or do nothing if element with such key is in Hashtable.
- 2.2 Deletion must remove element from Hashtable or do nothing if Hashtable doesn't contain element with such key.
- 2.3 Retrieval must provide access to element's value or notify user if Hashtable doesn't contain element with such key.
- 2.4 Notification must be delivered to user if user's hash function doesn't work properly.

>>> Requirements. II >>>

3. Performance.

3.1 Performance of insertion, retrieval and deletion must at least be comparable with a well-known implementation.

4. Implementation.

4.1 Hashtable must be implemented as a class using simple arrays.

4.2 Collision resolution strategy must be implemented with a separate chaining method or an open addressing method.

5. Delivery.

5.1 Testing capabilities must be provided.

5.2 Runtime analysis must be delivered.

>>> Specification. I >>>

1. Implement `Hashtable<K, V>` as a C++ template class.
2. Implement hash functions for following C++ built in types
`int`, `char`, `bool` and `std::string` type.
3. Implement following required methods
 - 3.1 `Hashtable<K, V>::add(const K& key, const V& value)`
 - 3.2 `const V* Hashtable<K, V>::get(const K& key)`
 - 3.3 `void Hashtable<K, V>::remove(const K& key)`
4. Each required method must run in $O(1)$ average case and $O(n)$ in worst case.
5. Throw an exception if user's hash function doesn't work properly.

```
>>> Specification. II >>>
```

6. Implement correctness tests for int, char, bool and std::string types.
7. Implement performance test for int and std::string types.
8. Implement correctness test for user's defined key, value and hash function.
9. All test should be implemented without using test frameworks.

>>> Structure. >>> Consept.

Hash Table uses separate chaining with linked lists method to resolve collisions ².

- * Hashtable<K, V> consists of an array of buckets³.
- * A bucket is an object of LinkedList<K, V> class.
- * LinkedList<K, V> class is an implementation of singly linked list customized to store values with unique keys.

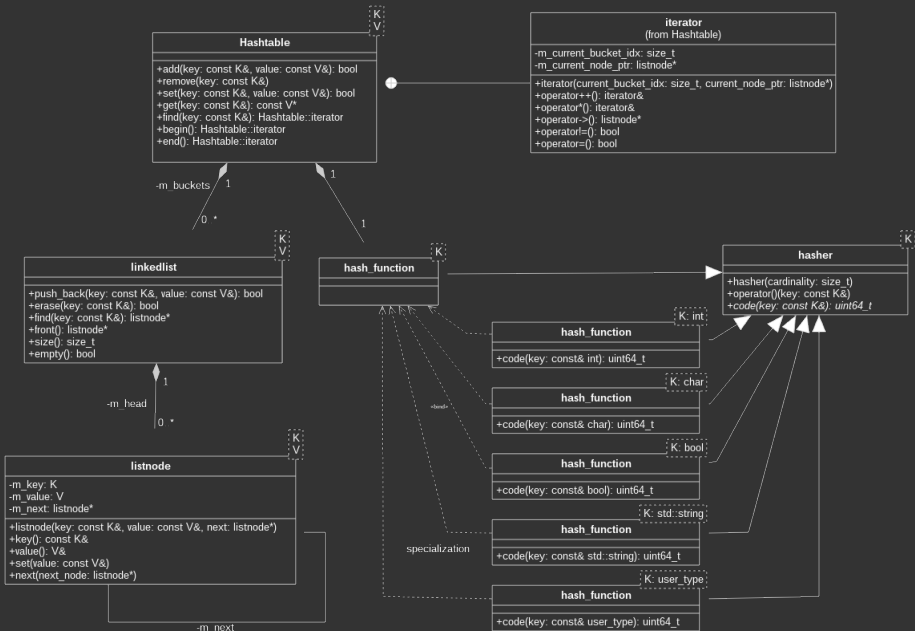
²Different keys are assigned to the same bucket.

³Bucket and chain can be used interchangeably, a bucket is implemented as a linked list.

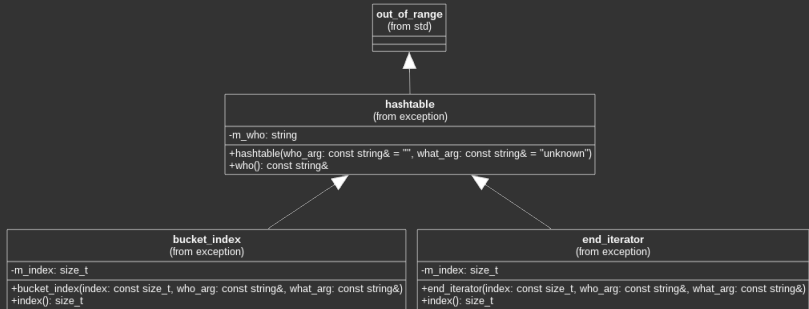
>>> Structure. >>> Classes.

```
* namespace esr
  * class Hashtable<K, V>
    * class iterator
  * class linkedlist<K, V>
  * class listnode<K, V>
  * class hasher<K>
  * class hash_function<K>
  * class hash_function<int>
  * class hash_function<char>
  * class hash_function<bool>
  * class hash_function<std::string>
  * namespace exception
    * class hashtable
    * class bucket_index
    * class end_iterator
```

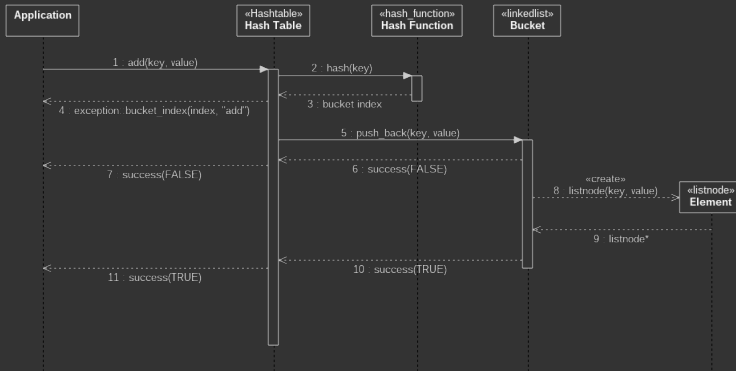
>>> Structure. >>> Classes from esr namespace.



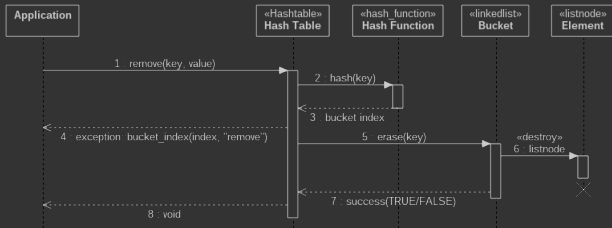
>>> **Structure.** >>> **Classes from esr::exception namespace.**



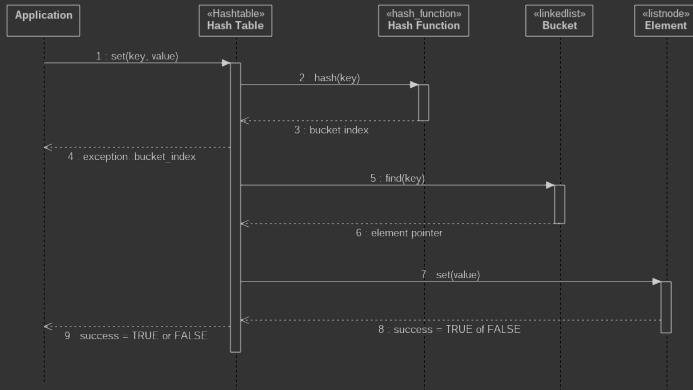
>>> Structure. >>> Insertion.



>>> Structure. >>> Deletion.



>>> Structure. >>> Set.



>>> Implementation. >>> Load Factor.

An Insertion, Deletion or Retrieval operation consists of the following steps.

1. Obtaining bucket index in bucket array using a hash function which maps key to bucket index. Running time is $O(1)$.
2. Finding an element of bucket, scanning a linked list and comparing the keys. Running time is $O(\text{size})$, size is a number of elements in list.
3. Performing a required operation. Running time is $O(1)$.

The average number of elements stored in bucket is

$$\alpha = \frac{n}{m}, \quad (1)$$

which is a Load Factor for Hashtable with m buckets that stores n elements.

>>> Implementation. I >>> Insertion.

Algorithm 1: add(key, value)

```
input      : key, value
output     : success
parameter: elementsCount, bucketsCount, loadFactorBoundUp,
           bucketArray is an array [ 0..bucketsCount ] of buckets

1 if elementsCount = 0 then
2   | RESIZE(1)
3 end
4  $loadFactor \leftarrow \frac{elementsCount}{bucketsCount}$ 
5 if loadFactor > loadFactorBoundUp then
6   | RESIZE(2·bucketsCount)
7 end
8 bucketIndex ← HASH(Key)
9 if bucketIndex ≥ bucketsCount then
10  | EXCEPTION(bucketIndex)
11 end
12 bucket ← bucketArray [bucketIndex ]
13 success ← bucket.ADDELEMENTTOBUCKET(key, value)
14 if success = TRUE then
15   | bucketsCount ← bucketsCount + 1
16 end
17 return success
```

>>> Implementation. II >>> Insertion.

Algorithm 2: AddElementToBucket(key, value)

```
input      : key, value
output     : success
parameter: front, nodesCount
1 if front = NULL then
2   | front  $\leftarrow$  { Create new list node with input key, value }
3 else
4   | foreach node n of list do
5     |   if n.key = key then
6       |     return FALSE
7     |   end
8   | end
9   | node  $\leftarrow$  { Create new list node with input key, value }
10  | node.next  $\leftarrow$  front
11  | front  $\leftarrow$  node
12 end
13 nodesCount  $\leftarrow$  nodesCount + 1
14 return TRUE
```

>>> Implementation. I >>> Deletion.

Algorithm 3: remove(key)

```
input      : key
parameter: elementsCount, bucketsCount, loadFactorBoundLow,
           bucketArray is an array [ 0..bucketsCount ] of buckets
1 if elementsCount = 0 then
2   | return
3 end
4 bucketIndex  $\leftarrow$  HASH(Key)
5 if bucketIndex  $\geq$  bucketsCount then
6   | EXCEPTION(bucketIndex)
7 end
8 bucket  $\leftarrow$  bucketArray [bucketIndex ]
9 success  $\leftarrow$  bucket.REMOVEELEMENTFROMBUCKET(key)
10 if success = FALSE then
11   | return
12 end
13 elementsCount  $\leftarrow$  elementsCount - 1
14 if elementsCount = 0 then
15   | RESIZE(0)
16   | return
17 end
18 loadFactor  $\leftarrow$   $\frac{\textit{elementsCount}}{\textit{bucketsCount}}$ 
19 if loadFactor < loadFactorBoundLow then
20   | RESIZE( $\frac{\textit{bucketsCount}}{2}$ )
21 end
```

>>> Implementation. II >>> Deletion.

Algorithm 4: RemoveElementFromBucket(key)

input : *key, value*

output : *success*

parameter: *front, nodesCount*

```
1  prev ← NULL
2  foreach node n of list do
3      if n.key = key then
4          if n = front then
5              | front ← n.next
6          else
7              | prev.next ← n.next
8          end
9          < Delete list node n >
10         nodesCount ← nodesCount - 1
11         return TRUE
12     end
13     prev ← n
14 end
15 return FALSE
```

>>> Implementation. >>> Resizing.

Algorithm 5: `resize(newBucketsCount)`

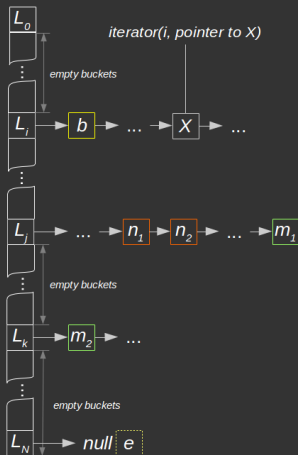
```
input      : newBucketsCount
parameter: bucketsCount,
           bucketArray is an array [ 0..bucketsCount ] of buckets

1 newBucketArray  $\leftarrow$  NULL
2 if newBucketsCount  $\neq$  0 then
3   newBucketArray  $\leftarrow$   $\langle$  Create array[0..newBucketsCount]  $\rangle$ 
4   HASH  $\leftarrow$   $\langle$  Create new Hash Function from it's family  $\rangle$ 
5 end
6 foreach bucket of bucketArray do
7   foreach element of bucket do
8     bucketIndex  $\leftarrow$  HASH(element.key)
9     if bucketIndex  $\geq$  bucketsCount then
10       $\langle$ throw out of range exception $\rangle$ 
11      success  $\leftarrow$  bucket.ADDELEMENTTOBUCKET(element.key, element.value)
12      newBucketsCount  $\leftarrow$  newBucketsCount + 1
13    end
14  end
15 end
16 if bucketArray not empty then
17    $\langle$ Delete bucketArray  $\rangle$ 
18 end
19 elementsCount  $\leftarrow$  elementsCount
20 bucketsCount  $\leftarrow$  newBucketsCount
21 bucketArray  $\leftarrow$  newBucketArray
```

>>> Implementation. >>> Iterator.

Iterator's position is defined by pair which is

- * a *bucket index*, index of linked list in an array;
- * an *element pointer*, pointer to node of linked list.



Array of N buckets

* Empty buckets

- * from L_0 to L_{i-1}
- * from L_{j+1} to L_{k-1}
- * from L_{k+1} to L_n

* Iterators

- * b is a begin iterator
- * e is an end iterator
- * n_2 is next to n_1
- * m_2 is next to m_1

>>> Implementation. >>> Iterator Begin, End, Current.

- * Begin Iterator

- * {first not empty bucket index, pointer to first element}

- * End Iterator

- * {last bucket index, NULL}

- * Current Iterator

- * {current bucket index, current element pointer}

Algorithm 6: Begin()

parameter: *bucketsCount*,

bucketArray is an array [0..*bucketsCount*] of buckets

1 *firstNotEmptyBucketIndex* \leftarrow 0

2 while *firstNotEmptyBucketIndex* < *bucketsCount* do

3 if *bucketArray* [*firstNotEmptyBucketIndex*] is not empty then

4 break

5 end

6 *firstNotEmptyBucketIndex* \leftarrow *firstNotEmptyBucketIndex* + 1

7 end

8 if *firstNotEmptyBucketIndex* = *bucketsCount* then

9 return ⟨End Iterator⟩

10 end

11 *bucket* \leftarrow *bucketArray* [*firstNotEmptyBucketIndex*]

12 return ⟨Begin Iterator⟩

>>> Implementation. >>> Iterator Advance.

Algorithm 7: Advance()

```
parameter: bucketsCount, bucketIndex, elementPointer
           bucketArray is an array [ 0..bucketsCount ] of buckets
1 elementPointer ← elementPointer.next
2 if elementPointer = NULL then
3     nextBucketIndex ← nextBucketIndex + 1
4     for i=nextBucketIndex to bucketsCount do
5         if bucketArray [nextBucketIndex] is empty then
6             | nextBucketIndex ← nextBucketIndex + 1
7         else
8             | break
9     end
10 end
11 nextNotEmptyBucketIndex ← nextBucketIndex
12 if nextNotEmptyBucketIndex < bucketsCount then
13     bucketIndex ← nextNotEmptyBucketIndex
14     bucket ← bucketArray [bucketIndex]
15     elementPointer ← bucket.front
16 else
17     bucketIndex ← bucketsCount - 1
18     elementPointer ← NULL
19 end
20 end
```

```
>>> Set/Get/Find.
```

```
>>> Tests. >>>
```

1. Correctness tests.

1.1 Insertions and Retrievals.

1.2 Copy and Assignments.

1.3 Deletion.

2. Performance tests.

2.1 Integer Keys Insertions and Retrievals.

2.2 Fixed Length String Keys Insertions and Retrievals.

2.3 Variable Length String Keys Insertions and Retrievals.

3. Application Sample: user's hash function test.

>>> Tests. >>> Correctness.

Tests functional correctness of `Hashtable<K, V>` for combinations of key type and value type, *int*, *bool*, *std::string*.

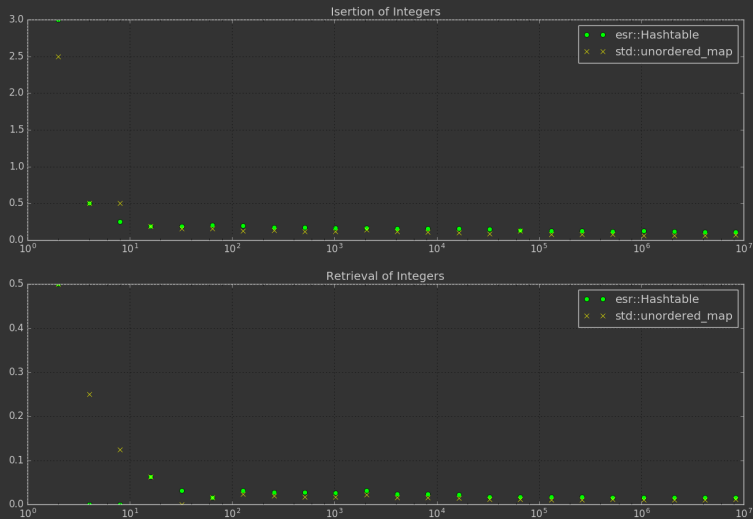
1. Insertion/Retrieval Tests ($9 \times 3 \times 2 = 48$ tests).
 - 1.1 `add()` positive/negative.
 - 1.2 `get()` positive/negative.
 - 1.3 `find()` positive/negative.
2. Copy/Assignments Tests ($9 \times 2 \times 1 = 18$ tests).
 - 2.1 Copy Constructor.
 - 2.2 Assignment Operator.
3. Deletion Tests ($9 \times 1 \times 2 = 18$ tests).
 - 3.1 `remove()` positive/negative.

>>> Tests. >>> Performance.

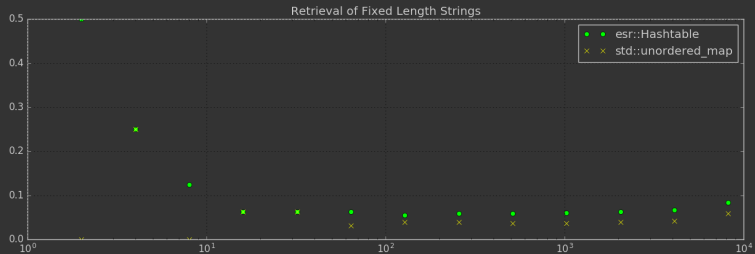
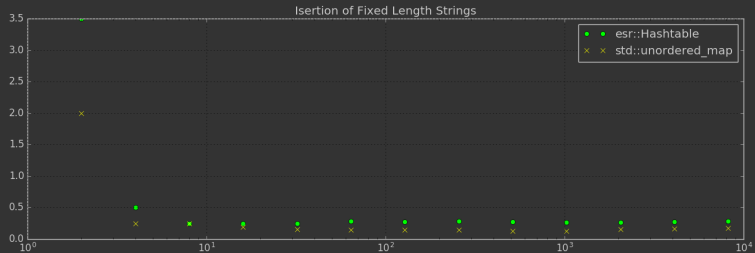
Tests performance of Hashtable<K, V> for *int* and *std::string* key types with *int* value type.

1. Integer key test. Key length is 4 bytes. Doubles number of elements.
2. Fixed Length string key test. Key length is 20 bytes. Doubles number of elements.
3. Variable Length string key test. Number of elements is 10000. Key length is 20 bytes. Doubles number of elements.

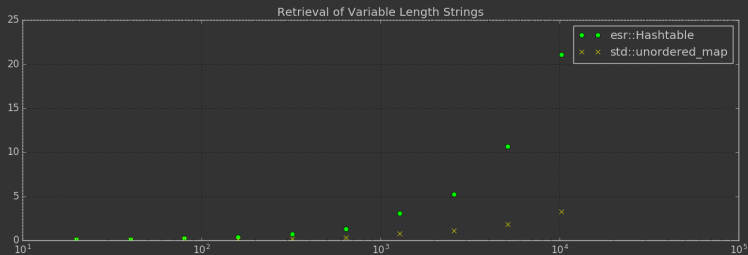
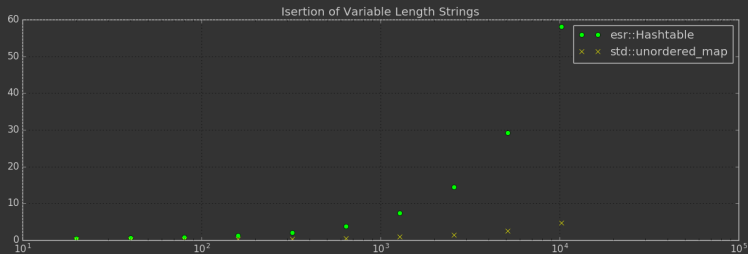
```
>>> Tests. >>> Performance. Integers.
```



>>> Tests. >>> Performance. Fixed Length Strings.



>>> Tests. >>> Performance. Variable Length Strings.



>>> Tests. >>> Performance. Summary.

Table: Comparison of `esr::Hashtable`⁴ and `std::unordered_map`⁵.

OPERATION	KEY TYPE	HT MEAN (μ s)	UM MEAN (μ s)	HT, UM DIFF (μ s)
INSERTION	integer	0.25437	0.22176	0.03261
	fixed string	0.53371	0.30994	0.22375
	var. string	11.78151	1.08067	10.70083
RETRIEVAL	integer	0.04087	0.04541	0.00454
	fixed string	0.11612	0.05375	0.06236
	var. string	4.30460	0.79793	3.50668

⁴HT stands for `esr::Hashtable`.

⁵UM stands for `std::unordered_map`.

>>> Tests. >>> Application Sample.

Goal is to check Hashtable's functional correctness with custom type of keys and custom hash function.

>>> Tests. >>> Application Sample. Structure.

Record in file: Munich; 1; 2010; 1185400; 31069; Bavaria;

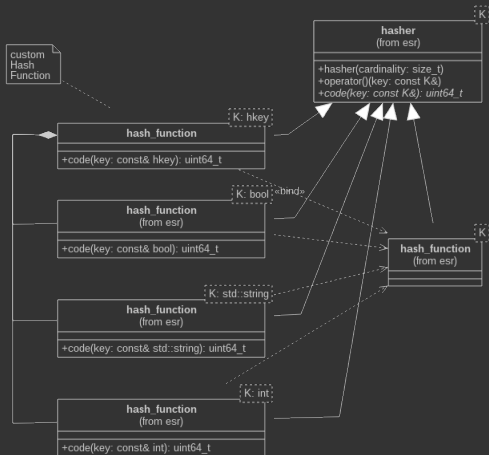
city
+name: std::string
+is_capital: bool
+year: uint16_t
+population: uint32_t
+area: uint32_t
+state: std::string

custom Hash Key

hkey
+name: std::string
+is_capital: bool
+year: uint16_t
+area: uint32_t
+state: std::string
+hkey(ct: const city&)
+operator=(other: const hkey&): bool

Element's value is a population of City with key defined by hkey.

K: hkey
V: uint_32_t
Hashtable



>>> Tests. >>> Application Sample. Custom Hash Function.

```
1 namespace esr {
2     template <>
3     class hash_function<city::hkey> : public esr::hasher<city::hkey> {
4     public:
5         explicit hash_function(size_t cardinality = 1, size_t start = 17, size_t prime = 31) :
6             esr::hasher<city::hkey>(cardinality),
7             m_bool_hasher(cardinality), m_int_hasher(cardinality),
8             m_string_hasher(cardinality),
9             m_start(start), m_prime(prime) {}
10
11     uint64_t code(const city::hkey& key) const {
12         uint64_t h = m_start;
13         h = m_prime * h + m_bool_hasher.code(key.is_capital);
14         h = m_prime * h + m_string_hasher.code(key.name);
15         h = m_prime * h + m_int_hasher.code(key.year);
16         h = m_prime * h + m_int_hasher.code(key.area);
17         h = m_prime * h + m_string_hasher.code(key.state);
18         return h;
19     }
20
21     private:
22     size_t m_start;
23     size_t m_prime;
24     esr::hash_function<bool> m_bool_hasher;
25     esr::hash_function<int> m_int_hasher;
26     esr::hash_function<std::string> m_string_hasher;
27 };
28 } // namespace esr
```

```
>>> Runtime Analysis. >>>
```

1. Runtime estimation of INSERTION.
2. Deterministic Hash Function vs. Universal Hash Function.
3. Summary of runtime estimations for Hash Table's operations.

>>> Runtime Analysis. >>> INSERTION. Naive.

Cost of insertion respective to input size is:

$$C^{insertion} = C^{hash} + C^{access} + C^{scan} + C^{create} \quad (2)$$

$$C^{hash} = O(1) \quad (3)$$

$$C^{access} = O(1) \quad (4)$$

$$C_{scan} = O(n) \quad (5)$$

$$C^{create} = O(1) \quad (6)$$

$$C^{insertion} = O(n) \quad (7)$$

Runtime is $O(n)$ because of the C^{scan} . Bucket size should be constant, to perform C^{scan} in $O(1)$ time.

>>> Runtime Analysis. >>> INSERTION with Resizing (I).

Load factor for n elements in Hash Table and m buckets is

$$\alpha = \frac{n}{m}. \quad (8)$$

Expected size for an i 'th bucket is

$$E[s_i] = \alpha. \quad (9)$$

Hashtable is implemented as a Dynamic Array of buckets to maintain α in range from 0.5 to 1. Hashtable is expanded to it's double size when load factor is close to 1.

>>> Runtime Analysis. >>> INSERTION with Resizing (II).

Cost of i 'th insertion respective to input size is

$$C_i = H + A + S_i + V + R_i \quad (10)$$

$$R_i = \begin{cases} (U + (A + (H + V) \cdot E[s_i]) \cdot (i - 1), & \alpha \geq 0.99; \\ 0, & \text{otherwise.} \end{cases} \quad (11)$$

Cost of resizing at i 'th inserion

$$R_i = \begin{cases} (U + (A + (H + V) \cdot \alpha) \cdot (i - 1), & i-1 \text{ is a power of } 2; \\ 0, & \text{otherwise.} \end{cases} \quad (12)$$

- * H is `hash()`.
- * A is access to bucket in array.
- * S is scan bucket to find an element.
- * V is cretaing an element and set key, value.
- * U is creating new hash function.

>>> Runtime Analysis. >>> INSERTION with Resizing (III).

Amortized cost of insertion of n'th element is

$$C(n) = \frac{\sum_{i=1}^n C_i}{n} = \frac{(H + A + V) \cdot n}{n} + S(n) + R(n) \quad (13)$$

Amortized cost of resizing at insertion of n'th element is

$$R(n) = \frac{\sum_{i=1}^n R_i}{n} = \frac{n + \sum_{i=1}^{\log_2(n-1)} (U + (A + (H + V) \cdot \alpha) \cdot 2^i)}{n} \quad (14)$$

$$R(n) = \left\lceil \frac{\log_2(n-1)}{n} \right\rceil \cdot U + \left[2 - \frac{4}{n} \right] \cdot (A + \alpha \cdot H + \alpha \cdot V) \quad (15)$$

Considering for load factor and bucket scan

$$\lim_{n \rightarrow \infty} \alpha = \lim_{n \rightarrow \infty} \left\lceil \frac{n}{m} \right\rceil = \frac{1}{2}, \lim_{n \rightarrow \infty} S(n) = 1 \quad (16)$$

Amortized cost of insertion

$$\lim_{n \rightarrow \infty} C(n) = 2 \cdot H + 3 \cdot A + 2 \cdot V + 1 \quad (17)$$

$$O(2 \cdot H) + O(3 \cdot A) + O(2 \cdot V) + O(1) = \textcolor{yellow}{O(1)} \quad (18)$$

>>> Runtime Analysis. >>> Deterministic Hash Function.

- * For deterministic hash function, there is a bad input on which it will have a lot of collisions.
- * Bad input is a powers of 2.
- * Hash Table's content for maximum bad input size of 64 bit unsigned integer keys.

```
0: {(32=>32) (64=>64) (128=>128) (256=>256) (512=>512) (1024=>1024) (2048=>2048) (4096=>4096) (8192=>8192) (16384=>16384) (32768=>32768) (65536=>65536) (131072=>131072) (262144=>262144) (524288=>524288) (1048576=>1048576) (2097152=>2097152) (4194304=>4194304) (8388608=>8388608) (16777216=>16777216) (33554432=>33554432) (67108864=>67108864) (134217728=>134217728) (268435456=>268435456) (536870912=>536870912) (1073741824=>1073741824)}
```

```
1: {}  
2: {(2=>2)}  
3: {}  
4: {(4=>4)}  
5: {}  
6: {}  
7: {}  
8: {(8=>8)}  
9: {}  
10: {}  
11: {}  
12: {}  
13: {}  
14: {}  
15: {}  
16: {(16=>16)}  
17: {}  
18: {}  
19: {}  
20: {}  
21: {}  
22: {}  
23: {}  
24: {}  
25: {}  
26: {}  
27: {}  
28: {}  
29: {}  
30: {}  
31: {}
```

>>> Runtime Analysis. >>> Hash function from Universal Family.

- * Hash Function selected from Universal Family of Hash Functions significantly reduces a number of collisions.
- * Input is a power of 2 is not bad one anymore.
- * Hash Table's content for maximum bad input size of 64 bit unsigned integer keys.

```
0: {(128=>128) (524288=>524288)}
1: {}
2: {}
3: {(2048=>2048)}
4: {(32=>32)}
5: {(8=>8)}
6: {(134217728=>134217728)}
7: {(33554432=>33554432)}
8: {}
9: {(1073741824=>1073741824)}
10: {(2=>2) (262144=>262144)}
11: {(268435456=>268435456)}
12: {(131072=>131072)}
13: {}
14: {}
15: {(32768=>32768)}
16: {(512=>512) (65536=>65536)}
17: {(8388608=>8388608)}
18: {(256=>256) (1024=>1024) (4096=>4096)}
19: {(4=>4)}
20: {(16777216=>16777216)}
21: {(536870912=>536870912)}
22: {(8192=>8192)}
23: {}
24: {(16384=>16384)}
25: {(4194304=>4194304)}
26: {(64=>64) (67108864=>67108864)}
27: {}
28: {(16=>16)}
29: {(2097152=>2097152)}
30: {}
31: {(1=>1) (1048576=>1048576)}
```

>>> Tests. >>> Performance. Hash Function for Integers.

Table: Comparison of `esr::Hashtable`⁶ and `std::unordered_map`⁷.

OPERATION	HASH FUNCTION	HT MEAN (μs)	UM MEAN (μs)	HT, UM DIFF (μs)
INSERTION	Deterministic	0.29295	0.24536	0.04758
	Universal	2.26267	0.24536	2.01731
RETRIEVAL	Deterministic	0.04228	0.05219	0.00991
	Universal	0.22027	0.05219	0.16808

`esr::Hashtable`'s Universal/Deterministic

1. INSERTION: 7.72367 times.

2. RETRIEVAL: 5.20892 times.

⁶HT stands for `esr::Hashtable`.

⁷UM stands for `std::unordered_map`.

>>> Runtime Analysis. >>> Summary.

OPERATION	WORST	AMORTIZED
INSERTION	$O(n)$	$O(1)$
DELETION	$O(n)$	$O(1)$
FIND	$O(n)$	$O(1)$
SET	$O(n)$	$O(1)$
GET	$O(n)$	$O(1)$
ITERATOR BEGIN	$O(n)$	$O(1)$
ITERATOR END	$O(1)$	$O(1)$
ITERATOR ADVANCE	$O(n)$	$O(1)$
ITERATOR COMPARE	$O(1)$	$O(1)$
LOAD FACTOR	$O(1)$	$O(1)$
SIZE	$O(1)$	$O(1)$

Table: Runtime estimations for Hash Table's operations.

```
>>> Conclusions. >>>
```

Implemented Hash Table meets all requirements, performing all operation in $O(1)$ amortized time.

Delivarables

- * Source Code.
- * 15-minutes talk material.