>>> 15 minutes talk.

>>> Hashtable.

Name: Konstantin Pachuev.

Date: July 12, 2016

>>> Outline >>>

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

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/

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 Isertion 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 implementetion.
4. Implementation.
   4.1 Hashtable must be implemented as a class using simple arrays.
   4.2 Colision 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. Impletent 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.

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

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

* Hashtable<K, V> consists of an array of buckets[3].
* 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.

---

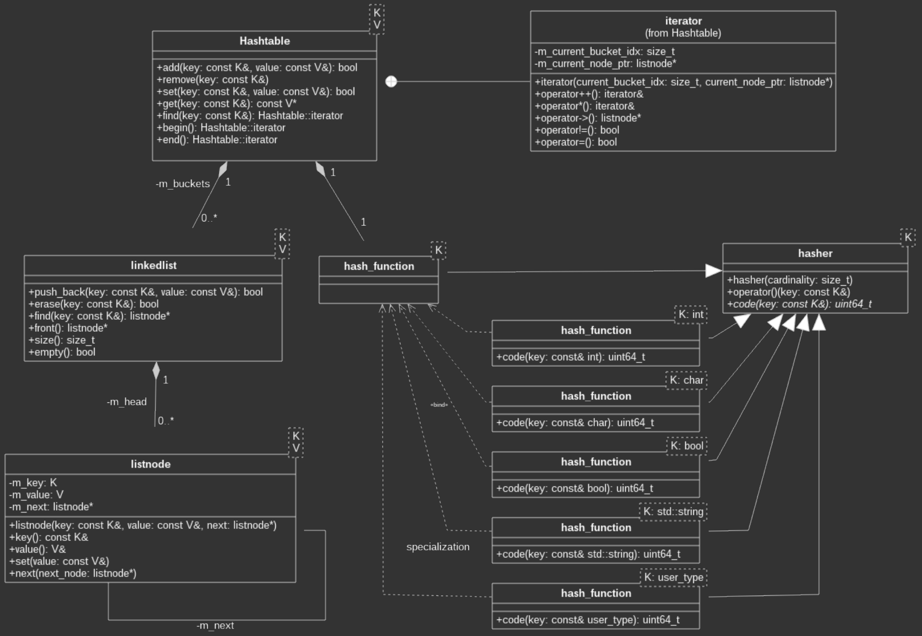
[2] Different keys are assigned to the same bucket.

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

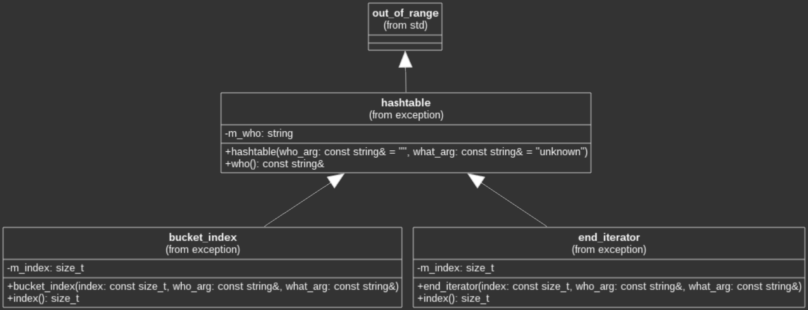
>>> Structure. >>> Classes.

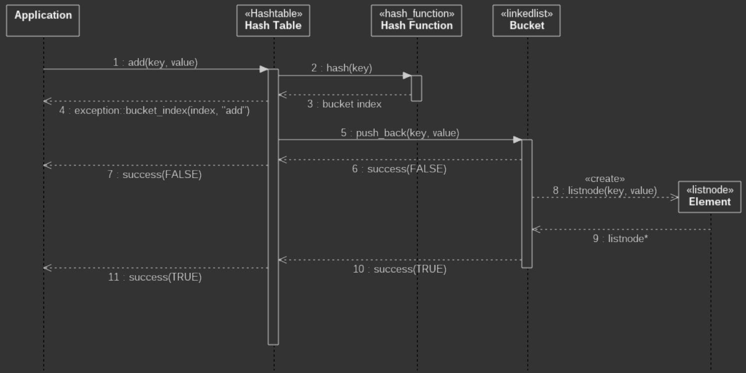
* namespace esr
  * class Hashtable<K, V>
    * class iterator
  * class linkedlist<K, V>
  * class listnode<K, V>
  * class hasher<K>
  * class hash_function<K>
  * class hash_function<int>
  * class hash_function<char>
  * class hash_function<bool>
  * class hash_function<std::string>
  * namespace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

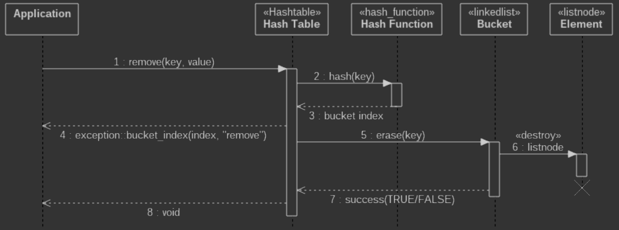
>>> Structure. >>> Classes from esr namespace.



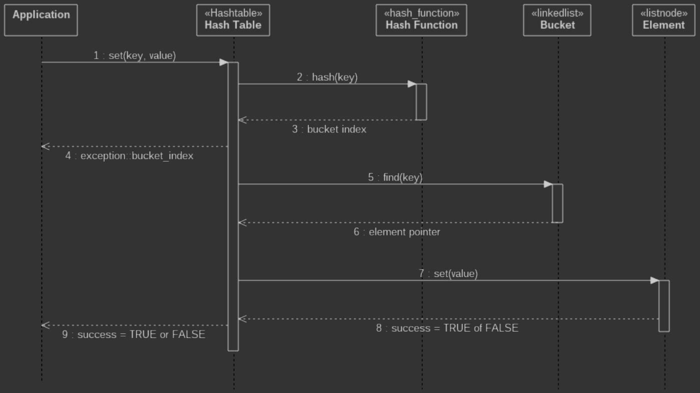
>>> Structure. >>> Classes from esr::exception namespace.

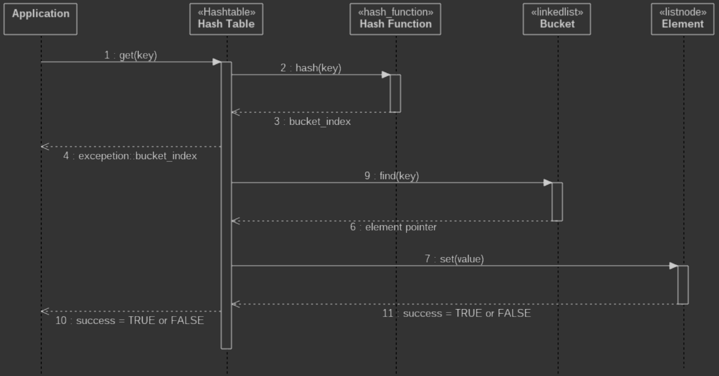



Application
«Hashtable»
Hash Table
«hash_function»
Hash Function
«linkedlist»
Bucket
1 : add(key, value)
2 : hash(key)
3 : bucket index
4 : exception::bucket_index(index, "add")
5 : push_back(key, value)
6 : success(FALSE)
7 : success(FALSE)
«create»
8 : listnode(key, value)
«listnode»
Element
9 : listnode*
10 : success(TRUE)
11 : success(TRUE)

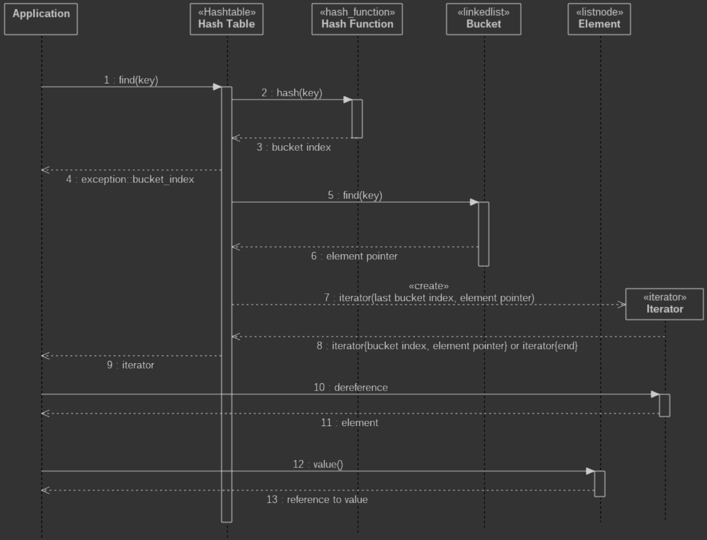

Application
«Hashtable»
Hash Table
«hash_function»
Hash Function
«linkedlist»
Bucket
«listnode»
Element
1 : remove(key, value)
2 : hash(key)
3 : bucket index
4 : exception::bucket_index(index, "remove")
5 : erase(key)
«destroy»
6 : listnode
7 : success(TRUE/FALSE)
8 : void

>>> Structure. >>> Set.




Application
«Hashtable»
Hash Table
«hash_function»
Hash Function
«linkedlist»
Bucket
«listnode»
Element
1 : get(key)
2 : hash(key)
3 : bucket_index
4 : excepetion::bucket_index
9 : find(key)
6 : element pointer
7 : set(value)
11 : success = TRUE or FALSE
10 : success = TRUE or FALSE

>>> Structure. >>> Find.



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, scaning a linked list and comparing the keys. Running time is O(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

$$\alpha = \frac{n}{m}, \tag{1}$$

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
if elementsCount = 0 then
    Resize(1)
end
loadFactor ← elementsCount / bucketsCount
if loadFactor > loadFactorBoundUp then
    Resize(2·bucketsCount)
end
bucketIndex ← Hash(Key)
if bucketIndex ≥ bucketsCount then
    Exception(bucketIndex)
end
bucket ← bucketArray [bucketIndex ]
success ← bucket.AddElementToBucket(key, value)
if success = TRUE then
    bucketsCount ← bucketsCount + 1
end
return success
```

Algorithm 2: AddElementToBucket(key, value)

```
input     : key, value
output    : success
parameter: front, nodesCount
if front = NULL then
    front ← ⟨ Create new list node with input key, value ⟩
else
    foreach node n of list do
        if n.key = key then
            return FALSE
        end
    end
    node ← ⟨ Create new list node with input key, value ⟩
    node.next ← front
    front ← node
end
nodesCount ← nodesCount + 1
return TRUE
```

>>> Implementation. I >>> Deletion.

Algorithm 3: remove(key)

```
input    : key
parameter: elementsCount, bucketsCount, loadFactorBoundLow,
           bucketArray is an array [ 0..bucketsCount ] of buckets
if elementsCount = 0 then
 |  return
end
bucketIndex ← Hash(Key)
if bucketIndex ≥ bucketsCount then
 |  Exception(bucketIndex)
end
bucket ← bucketArray [bucketIndex ]
success ← bucket.RemoveElementFromBucket(key)
if success = FALSE then
 |  return
end
elementsCount ← elementsCount - 1
if elementsCount = 0 then
 |  Resize(0)
 |  return
end
loadFactor ← elementsCount / bucketsCount
if loadFactor < loadFactorBoundLow then
 |  Resize(bucketsCount / 2)
end
```

>>> Implementation. II >>> Deletion.

Algorithm 4: RemoveElementFromBucket(key)

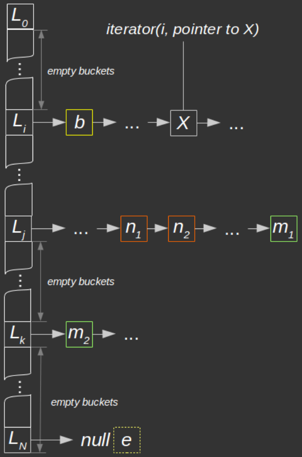
```
input    : key, value
output   : success
parameter: front, nodesCount
prev ← NULL
foreach node n of list do
    if n.key = key then
        if n = front then
            front ← n.next
        else
            prev.next ← n.next
        end
        ⟨ Delete list node n ⟩
        nodesCount ← nodesCount - 1
        return TRUE
    end
    prev ← n
end
return FALSE
```

Algorithm 5: resize(newBucketsCount)

```
input    : newBucketsCount
parameter: bucketsCount,
           bucketArray is an array [ 0..bucketsCount ] of buckets
newBucketArray ← NULL
if newBucketsCount ≠ 0 then
|   newBucketArray ← ⟨ Create array[0..newBucketsCount]⟩
|   HASH ← ⟨Create new Hash Function from it's family⟩
end
foreach bucket of bucketArray do
|   foreach element of bucket do
|   |   bucketIndex ← HASH(element.key)
|   |   if bucketIndex ≥ bucketsCount then
|   |   |   ⟨throw out of range exception⟩
|   |   |   success ← bucket.ADDELEMENTTOBUCKET(element.key, element.value)
|   |   |   newBucketsCount ← newBucketsCount + 1
|   |   end
|   end
end
if bucketArray not empty then
|   ⟨Delete bucketArray ⟩
end
elementsCount ← elementsCount
bucketsCount ← newBucketsCount
bucketArray ← newBucketArray
```

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 n1
  * 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}

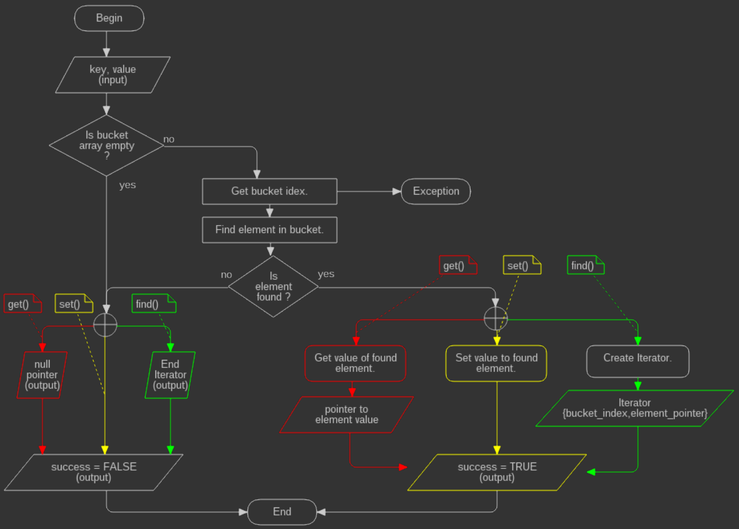
Algorithm 6: Begin()

```
parameter: bucketsCount ,
           bucketArray is an array [ 0..bucketsCount ] of buckets
firstNotEmptyBucketIndex ← 0
while firstNotEmptyBucketIndex < bucketsCount do
    if bucketArray [firstNotEmptyBucketIndex ] is not empty then
        break
    end
    firstNotEmptyBucketIndex ← firstNotEmptyBucketIndex + 1
end
if firstNotEmptyBucketIndex = bucketsCount then
    return ⟨End Iterator⟩
end
bucket ← bucketArray [firstNotEmptyBucketIndex ]
return ⟨Begin Iterator⟩
```

>>> Implementation. >>> Iterator Advance.

Algorithm 7: Advance()

```
parameter: bucketsCount , bucketIndex , elementPointer
           bucketArray is an array [ 0..bucketsCount ] of buckets
elementPointer ← elementPointer .next
if elementPointer = NULL then
    nextBucketIndex ← nextBucketIndex + 1
    for i=nextBucketIndex to bucketsCount do
        if bucketArray [nextBucketIndex ] is empty then
            nextBucketIndex ← nextBucketIndex + 1
        else
            break
        end
    end
    nextNotEmptyBucketIndex ← nextBucketIndex
    if nextNotEmptyBucketIndex < bucketsCount then
        bucketIndex ← nextNotEmptyBucketIndex
        bucket ← bucketArray [bucketIndex ]
        elementPointer ← bucket .front
    else
        bucketIndex ← bucketsCount - 1
        elementPointer ← NULL
    end
end
```


Begin
key, value
(input)
Is bucket
array empty
?
no
yes
Get bucket idex.
Exception
Find element in bucket.
no
Is
element
found ?
yes
get()
set()
find()
null
pointer
(output)
End
Iterator
(output)
success = FALSE
(output)
get()
set()
find()
Get value of found
element.
Set value to found
element.
Create Iterator.
pointer to
element value
Iterator
{bucket_index,element_pointer}
success = TRUE
(output)
End

>>> Tests. >>>

1. Linked List Correctness tests.
    1.1 Access.
    1.2 Copy and Assignments.
    1.3 Deletion.
    1.4 Fake object.
2. Hash Table Correctness tests.
    2.1 Insertions and Retrievals.
    2.2 Copy and Assignments.
    2.3 Deletion.
3. Hash Table Performance tests.
    3.1 Integer Keys Isertions and Retrievals.
    3.2 Fixed Length String Keys Isertions and Retrievals.
    3.3 Variable Length String Keys Isertions and Retrievals.
4. Application Sample: user's hash function test.

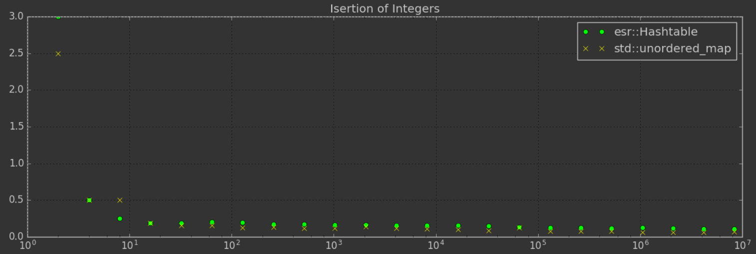
Tests functional correctness of Hashtable<K, V> for combinations of key type and value type, *int*, *bool*, *std::string*.

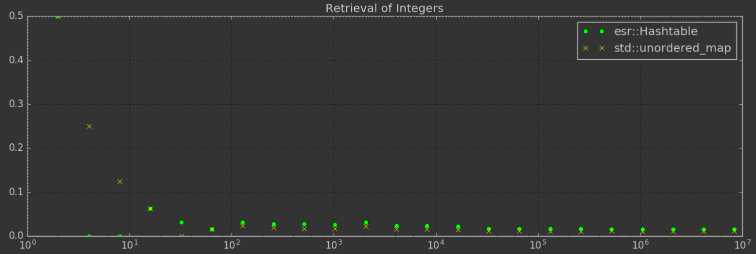
1. Isertion/Retrieval Tests (9*3*2 = 48 tests).
   1.1 add() positive/negative.
   1.2 get() positive/negative.
   1.3 find() positive/negative.
2. Copy/Assignments Tests (9*2*1 = 18 tests).
   2.1 Copy Constructor.
   2.2 Assignment Operator.
3. Deletion Tests (9*1*2 = 18 tests).
   3.1 remove() positive/negative.

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

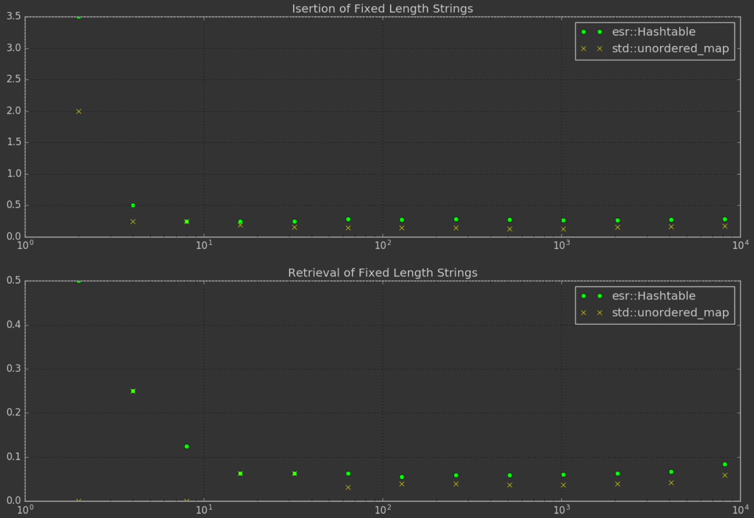
1. Integer key test. Key length is 4 bytes. Doubles number of elements.
2. Fixed Length string key test. Key length is 20 bytes. Doubles number of elements.
3. Variable Length string key test. Number of elements is 10000. Key length is 20 bytes. Doubles number of elements.

>>> Tests. >>> Hash Table Performance. Integers.





>>> Tests. >>> Hash Table Performance. Fixed Length Strings.



\>>> Tests. >>> Hash Table Performance. Variable Length Strings.

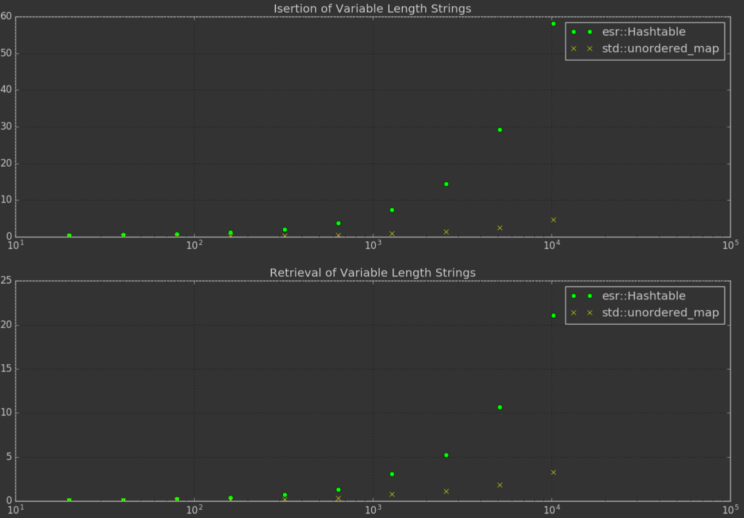
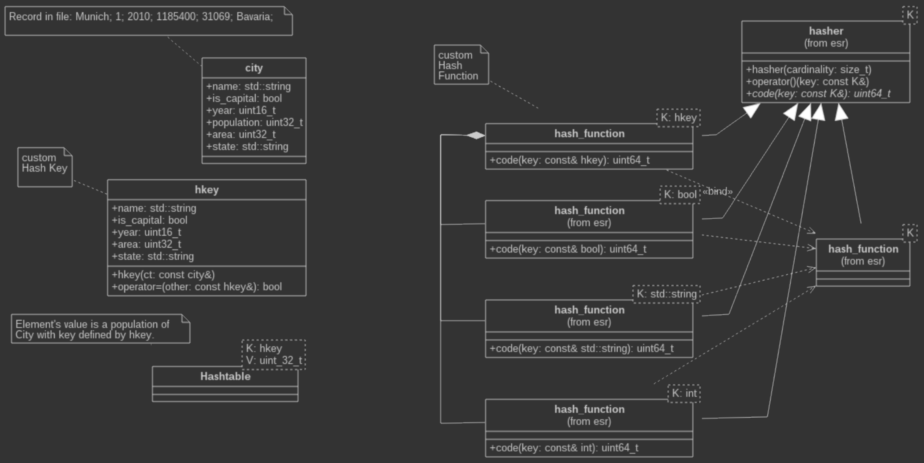


Table: Comparison of esr::Hashtable[4] and std::unordered_map[5].

OPERATION	KEY TYPE	HT MEAN (μs)	UM MEAN (μs)	HT, UM DIFF (μs)
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

[4] HT stands for esr::Hashtable.
[5] UM stands for std::unordered_map.

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


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
custom Hash Function
K
hasher
(from esr)
+hasher(cardinality: size_t)
+operator()(key: const K&)
+code(key: const K&): uint64_t
K: hkey
hash_function
+code(key: const& hkey): uint64_t
K: bool
«bind»
hash_function
(from esr)
+code(key: const& bool): uint64_t
K
hash_function
(from esr)
K: std::string
hash_function
(from esr)
+code(key: const& std::string): uint64_t
K: int
hash_function
(from esr)
+code(key: const& int): uint64_t

```cpp
namespace esr {
  template <>
  class hash_function<city::hkey> : public esr::hasher<city::hkey> {
    public:
    explicit hash_function(size_t cardinality = 1, size_t start = 17, size_t prime = 31) :
                           esr::hasher<city::hkey>(cardinality),
                           m_bool_hasher(cardinality), m_int_hasher(cardinality),
                           m_string_hasher(cardinality),
                           m_start(start), m_prime(prime) {}

    uint64_t code(const city::hkey& key) const {
      uint64_t h = m_start;
      h = m_prime * h + m_bool_hasher.code(key.is_capital);
      h = m_prime * h + m_string_hasher.code(key.name);
      h = m_prime * h + m_int_hasher.code(key.year);
      h = m_prime * h + m_int_hasher.code(key.area);
      h = m_prime * h + m_string_hasher.code(key.state);
      return h;
    }

    private:
    size_t m_start;
    size_t m_prime;
    esr::hash_function<bool> m_bool_hasher;
    esr::hash_function<int> m_int_hasher;
    esr::hash_function<std::string> m_string_hasher;
  };
}  // namespace esr
```

>>> Runtime Analysis. >>>

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

Cost of insertion respective to input size is:

$$C^{insertion} = C^{hash} + C^{access} + C^{scan} + C^{create} \tag{2}$$

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

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

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

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

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

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

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

$$\alpha = \frac{n}{m}. \tag{8}$$

Expected size for an i'th bucket is

$$E[s_i] = \alpha. \tag{9}$$

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.

Cost of i'th insertion respective to input size is

$$C_i = H + A + S_i + V + R_i \tag{10}$$

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

Cost of resizing at i'th inserion

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

* 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.

Amortized cost of insertion of n'th element is

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

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

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

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

Considering for load factor and bucket scan

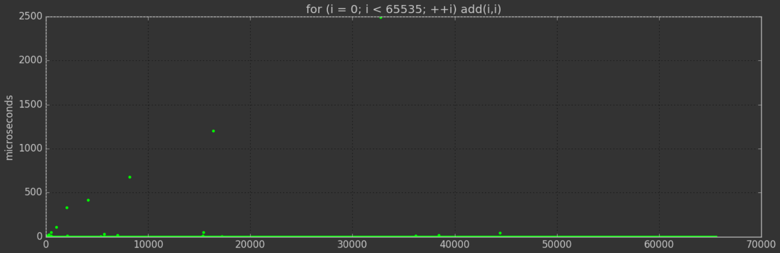
$$\lim_{n\to\infty} \alpha = \lim_{n\to\infty} \left[\frac{n}{m}\right] = \frac{1}{2}, \lim_{n\to\infty} S(n) = 1 \tag{16}$$

Amortized cost of insertion

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

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

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



>>> Runtime Analysis. >>> Determenistic 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)(335
54432=>33554432)(67108864=>67108864)(134217728=>134217728)(268435456=>268435456)(536870912=>536870912)(1073741824=>1073741824)}
  1: {(1=>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 Fucntion selected from Universal Family of Hash Functions significantly resuces a number of collisions.
* Input is a powers 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)}
```

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/Deremenistic

1. INSERTION: 7.72367 times.
2. RETRIEVAL: 5.20892 times.

[6] HT stands for esr::Hashtable.
[7] UM stands for std::unordered_map.

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.