ADT Map implementation on a hash table, collision resolution by open addressing

<u>Domain of the ADT Map:</u>

M = { *m* | *m* is a map with elements *e = (k, v)*, where $k \in TKey$ and $v \in TValue$ }

<u>Interface of the ADT Map:</u>

init(m)

descr: creates a new map
pre: true
post: $m \in M$, *m* is an empty map

destroy(m)

descr: destroys a map
pre: $m \in M$
post: *m* was destroyed

add(m, k, v)

descr: add e new key-value pair to the map
pre: $m \in M, k \in TKey, v \in TValue$
post: $m' \in M, m' = m \cup \langle k, v\rangle$
@ throws exception if *p* is already in the set of keys

remove(m, k)

descr: removes a pair with a given key from the map
pre: $m \in M, k \in TKey$
post: $v \in TValue$, where

$$remove \leftarrow \begin{cases} v', & \text{if } \exists \langle k, v'\rangle \in m \text{ and } m' \in M, \\ & \quad m' = m \setminus \langle k, v'\rangle \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

search(m, k)

descr: searches for the value associated with a given key in the map
pre: $m \in M, k \in TKey$
post: $v \in TValue$, where

$$search \leftarrow \begin{cases} v', & \text{if } \exists \langle k, v'\rangle \in m \\ 0_{TValue}, & \text{otherwise} \end{cases}$$

iterator(m, it)

descr: returns an iterator for a map
pre: $m \in M$
post: $it \in I$, *it* is an iterator over m

size(m)

descr: returns the number of pairs from the map
pre: $m \in M$
post: size ← the number of pairs from *m*

Domain of the Iterator:

I = { *it* | *it* is an iterator over $m \in M$ }

Interface of the Iterator:

init(it, m)

pre: $m \in M$

post: $it \in I$, *it* is an iterator over *m*

valid(it)

pre: $it \in I$

post:

$$valid \leftarrow \begin{cases} true, & \text{if the current element from it is a valid one} \\ false, & \text{otherwise} \end{cases}$$

next(it)

pre: $it \in I$, valid(it)

post: $it' \in I$, the current element from *it'* refers to the next element from the map *m*

getCurrent(it, pair)

pre: $it \in I$, valid(it)

post: $pair \in Pair$, *pair* is the current $<k, v>$ pair from *it*

Representation

Pair:

key: *TKey*

value: *TValue*

Map:

elems: *Pair[]*

len: *Integer*

nr: *Integer*

h1: *TFunction*

h2: *TFunction*

Iterator:

itt: *Integer*

m: *Map*

Problem statement:

Given two numbers n and m, and n correct* numbers greater than 0, verify if in the list of numbers are 2 numbers that added up are equal to m.

Also, every number has an additional information about itself, a cost c, with the meaning that to use that number in the addition to get m it costs c units.

If there exists duplicate numbers, only the first record is going to be kept.

(*)if an already existing element is added, it is not counted and another one is given, until is correct

E.g.

N : 5
M : 6
(9,3)
(2,4)
(1,5)
(4,9)
(5,1)

We can obtain m = 6 in two ways. First, 2(4) + 4(9) = 6(13). Secondly, 1(5) + 5(1) = 6(6).

So we choose 1 and 5 as their cost is less than 2 and 4.

Justification:

This problem fits our Map ADT because together with the value of the number we have an information about its cost. So, the value of the number represents the key and the cost represents the value.

Solution:

We are going to iterate the map from the beginning and search in the map M – (the value of the current element). If any pair is found such that their keys added sum up to M, we add their values and update the global minimum if possible.

Implementation

subalgorithm init(m) **is:**
    @initialize the hash functions
    @initialize the value of m
    for i ← *0*, len-1 execute:
        m.elems[i].key ← -1
end-subalgorithm
Complexity: Θ(len)

subalgorithm destroy(m) **is:**
    @m is destroyed
end-subalgorithm
Complexity: Θ(1)

subalgorithm add(m, k, v) **is:**
    if size(m) = len **then**
        @resize and rehash
    end-if
    pos ← m.h(k)
    i ← 1
    while m.elems[pos].key ≠ -1 and m.elems[pos].key ≠ -2 **execute**
        if m.elems[pos].key = k **then**
            @throw already-in-array exception
        end-if

pos ←(m.h1(k) + m.h2(k) * i) % m.len
i ← i + 1
end-while
m.nr ← m.nr + 1
m.elems[pos].key ←*k*
m.elems[pos].value ←*v*
end-subalgorithm
Best Case Complexity: Θ(1) – when there are no collisions yet, so the pair is just placed there
Worst Case Complexity: O(len) – when we make a full checking of the array

Function remove(m, k):
if m.nr = 0 **then**
remove ← -1
end-if
pos ← m.h1(k)
i ← 1
while m.elems[pos].key ≠ k **execute**
if m.elems[pos].key = -1 or i = m.len **then**
remove ← -1
end-if
pos ←(m.h1(k) + m.h2(k) * i) % m.len
i ← i + 1
end-while
value ← m.elems[pos].value
m.nr ← m.nr – 1
m.elems[pos].key ← -2
m.elems[pos].value ← -2
search ←value
end-function
Best Case Complexity: Θ(1) – when the key is on its correct position
Worst Case Complexity: Θ(len) – when the key is not in the set of keys

Suppose we have added n items into table of size len. Under the uniform hashing assumption the remove operation has expected cost of ≤ 1 / (1 – α), where α = (n / len) < 1.

probability first probe successful: (len – n) / len = p
(n bad slots, len total slots, and first probe is uniformly random)

first probe fails, probability second probe successful: (len – n) / (len – 1) >= (len – n) / len = p
2nd probe fails, probability 3rd probe successful: (len – n) / (len – 2) >= (len – n) / len = p
and so on…

Every trial, succes with probability at least p }
} Complexity : = O(1 / (1-α))
1 / p = 1 / (1 - α) <=> p = 1 - α }

Function search(m, k) **is:**
pos ← m.h1(k)
i ← 1
while m.elems[pos].key ≠ k or i = m.len **execute**
if m.elems[pos].key = -1 **then**

```
                remove ← -1
            end-if
            pos ←( m.h1(k) + m.h2(k) * i ) % m.len
            i ← i + 1
        end-while
        search ← m.elems[pos].value
end-function
```
Best Case Complexity: Θ(1) – when the key is on its correct position
Average Case Complecity: O(1 / (1-α))
Worst Case Complexity: Θ(cap) – when the key is not in the set of keys

Function size**(m) is:**
    size ←m.nr
end-function
Complexity: Θ(1)

Function iterator(m, it) **is:**
    iterator ←init(it, m)
end-function
Complexity: Θ(1)

subalgorithm init(it, m) **is:**
    it.m ←m;
    it.itt ←0;
end-subalgorithm
Complexity: Θ(1)

Function valid(it) **is:**
    if it.itt < it.m.len **then**
        valid ←*true*
    valid ←false
    end-if
end-function
Complexity: Θ(1)

Function valid(it) **is:**
    it.itt ←it.itt + 1
end-function
Complexity: Θ(1)

Function getCurrent(it) **is:**
    getCurrent ←it.m.elems[it.itt]
end-function
Complexity: Θ(1)

Implementation for the problem solution:

subalgorithm run(m, min, result, sum) **is:**
//pre: m is a map, min is the minimum sum from any 2 values whose keys sum up to ‘sum’
//**pre:** result is a Pair and sum represents the wanted sum to be computed
    iterator(m, it)

```
    while valid(it) execute
        value ← -1
        if sum > getCurrent(it).key then
            value ← search(m, sum – getCurrent(it).key)
        end-if
        if value ≠ -1 then
            maybe ← getCurrent(it).value + value
            if maybe < min and getCurrent(it).value ≠ value then
                min ← maybe
                result ← getCurrent(it)
            end-if
        end-if
        next(it)
    end-while
end-subalgorithm
```

Complexity: O(len * 1 / (1-α))

<u>Tests</u>

```
void testMap()
{
   Map m{15};
   assert(m.h1(15) == 0);
   assert(m.h1(19) == 4);
   assert(m.h2(15) == 6);
   assert(m.h2(19) == 2);
   m.add(2, 3);
   m.add(3, 2);
   m.add(18, 4);
   m.add(33, 4);

   try {
      m.add(2, 3);

   } catch (...) {
      std::cout << "The key is already existing\n";
   }

   assert(m.search(2) == 3);
   assert(m.search(4) == -1);
   assert(m.search(3) == 2);

   assert(m.nr == 4);
   assert(m.remove(2) == 3);

   assert(m.nr == 3);
   assert(m.remove(4) == -1);
   assert(m.nr == 3);

   m.add(2, 5);

   assert(m.nr == 4);
```

```
    assert(m.size() == 4);
    assert(m.remove(2) == 5);
    assert(m.size() == 3);
    assert(m.nr == 3);

    Iterator it = m.iterator();
    assert(it.getCurrent().key == -1);

    it.next();
    assert(it.valid() == true);
    assert(it.getCurrent().key == -1);

    it.next();
    it.next();
    assert(it.valid() == true);
    assert(it.getCurrent().key == 3);
}

void testProgram()
{
    Map m{5};
    m.add(9, 3);
    m.add(2, 4);

    try {
        m.add(2, 3);

    } catch (...) {
        std::cout << "The key is already existing\n";
    }

    m.add(1, 5);
    m.add(4, 9);

    try {
        m.add(4, 9);

    } catch (...) {
        std::cout << "The key is already existing\n";
    }

    m.add(5, 1);

    try {
        m.add(100, 3);

    } catch (...) {
        std::cout << "The map is full\n";
    }

    int minimum = 99999;
    Pair result;
```

```
    run(m, minimum, result, 100);
    assert( minimum == 99999 );

    run(m, minimum, result, 6);
    assert( minimum == 6 );

    minimum = 99999;
    run(m, minimum, result, 9);
    assert( minimum == 10 );

    minimum = 99999;
    run(m, minimum, result, 5);
    assert( minimum == 14 );
}
```