**ADT Multimap - implementation on a hash table, collision resolution by coalesced chaining**

A Multimap is a container where the elements are *<key, value>* pairs. We can access the values only by using the key it is associated to; this means that there are no positions in a Multimap.

Collision resolution by coalesced chaining means that each element from the hash table has a next field, that is represented on an integer and points to the position of the next element, similar to a linked list on an array.

Domain:

Multimap = { map | map is a multimap with elements e = <k,v>, where k *TKey* and v *TValue* }

Representation of the container:

The ADT Multimap is represented on a *hash table*, having as collision resolution *coalesced chaining*.

HashTable:

T: *TElement[]*

next: *Integer[]*

m: *Integer*

firstFree: *Integer*

h: *TFunction*

Interface:

* **init(mm)**
  + pre: true
  + desc: creates a new, empty multimap
  + post: mm *Multimap* and is empty
* **destroy(mm)**
  + pre: mm *Multimap*
  + desc: frees the memory occupied by the given multimap, thus destroying it
  + post: mm was destroyed, meaning that the allocated memory was freed
* **add(mm, k, v)**
  + pre: mm *Multimap*, k *TKey*, v *TElem*
  + desc: adds a new element, meaning a pair <k,v>, to the multimap
  + post: <k,v> mm’, mm’ *MM*, m <k,v>
* **remove(mm, k, v)**
  + pre: mm *Multimap*, k *TKey*, v *TElem*
  + desc: removes the pair <k,v> from the multimap, if such a pair exists
  + post: if the pair exists, it has been removed, <k,v> mm’, mm’  *MM*, otherwise, an error is thrownb
* **search(mm, k, v)**
  + pre: mm *Multimap*, k *TKey*, v *TElem*
  + desc: searches for the given value associated to a key in the multimap and returns an appropriate truth value for the result
  + post: search ← True if there is a value v associated to a key k in the multimap, False otherwise
* **iterator(mm, it)**
  + pre: mm *MM*
  + desc: creates an iterator over mm *MM*
  + post: it *I* and it is an iterator over mm

Iterator:

Domain: I = { it | it is an iterator over a map having the elements of type *TElement*, which has in its structure a *TKey* and a *TValue* field }

Representation:

Iterator:

ht: *HashTable*

currentElement: *Integer*

* **init(it, ht)**
  + pre: it *IteratorHT*, ht *HashTable*
  + desc: creates an iterator over ht *HashTable*
  + post: it *I*, is an iterator over ht and it points to the first element in ht, if ht is not empty
* **getCurrent(it)**
  + pre: it *IteratorHT*, it is valid
  + desc: returns the current element from the iterator
  + post: getCurrrent ← the current element of the iterator
* **next(it)**
  + pre: it *IteratorHT*, it is valid
  + desc: sets the current element to the next one or makes it invalid if there are no elements left
  + post: the current element from it points to the next element from the container
* **valid(it)**
  + pre: it *IteratorHT*, may or may not be valid
  + desc: checks whether the iterator is valid and returns an appropriate truth value; an iterator is considered to be valid if it points to a valid element from the container
  + post: valid ← True if the iterator is valid, False otherwise

Problem statement

Consider a directed graph having nodes indexed from 0 to a given number (n-1), which we will represent using the ADT Multimap. The <key,value> pairs will represent the the edges of the graph, such that *key* is the node from which an edge starts and *value* is the node towards which the edge goes. The application will provide the following functionalities: adding an edge to the graph, removing an edge from the graph, searching if a given edge exists and printing the graph. Each edge is unique.

Description of the solution

I have chosen to implement a directed graph using the ADT Multimap, represented on a hash table, due to the fact that it is an efficient way to store the edges of such a graph, especially in the case of sparse graph. It also allows for an easy traversal of the edges of the graph, addition, removal and searching for a specific edge, as the edges starting from the same node linked by the *next[]* field, and we use an iterator to easily access elements and parse through them.

Implementation of Multimap operations:

**subalgorithm add(key, value)**

new\_element ← new TElement

[new\_element].key ← key

[new\_element].value ←value

table.add(new\_element)

end-subalgorithm

**subalgorithm remove(key, value)**

new\_element ← new TElement

[new\_element].key ← key

[new\_element].value ←value

table.remove(new\_element)

end-subalgorithm

**function search(key, value)**

new\_element ← new TElement

[new\_element].key ← key

[new\_element].value ←value

if table.search(new\_element) = True then

search ←True

else

search ← False

end-if

end-function

Implementation for Hashtable operations:

**function hash(key):**

position ← key % [table].m

hash ← position

end-function

**subalgorithm init():**

[table].T ← new TElement[[table].m]]

[table].next ← new Integer[[table].m]

for I ← 0, [table].m execute:

[table].next[i] = -1

end-for

[table].firstFree = 0

end-subalgorithm

**subalgorithm destroy():**

free([table].T)

free([table].next)

end-subalgorithm

**subalgorithm add(new\_element):**

key ← [new\_element].getKey()

pos ← hash(key)

if [table].T[pos].isEmpty() == true then:

[table].T[pos] = new\_element

[table].next[pos] ← -1

else:

if [table].firstFree == [table].m:

@resize

end-if

current ← pos

while [table].next[current] =/= -1 then:

current ← [table].next[current]

end-whileta

[table].T[[table].firstFree] ← new\_element

[table].next[[table].firstFree] ← -1

[table].next[current] ← [table].firstFree

end-if

end-subalgorithm

**subalgorithm remove(element):**

i ← hash([element].key)

j ← -1

index ← 0

while index < [table].m and j == -1 execute:

if [table].next[index] == i then:

j ← index

else

index ← index +1

end-if

end-while

while I =/= -1 and [table].T[i] =/= execute:

j ← i

i ← [table].next[i]

end-while

if i == -1 then:

@throw exception “Key doesn’t exist”

else

over = false

do

p ← [table].next[i]

pp ← i

while p =/= -1 and hash([[table].T[p]].key =/= i then:

pp ← p

p ← [table].next[p[

end-while

if p == -1

over ← true

else

[table].T[i] ← [table].T[p]

j ← pp

i ← p

end-if

until over == true

end-do

if j =/= -1 then:

[table].next[j] ← [table].next[i]

end-if

[[table].T[i]].key ← -1

[[table].T[i]].value ← -1

[table].next[i] ← -1

if [table].firstFree > i then:

[table].firstFree ← i

end-if

end-if

end-subalgorithm

**function search(element):**

i ← hash([element[.key)

while i =/= -1 and [table].T[i] =/= element execute:

i ← [table].next[i]

end-while

if i == -1 then:

search ← false

else

search ← true

end-if

end-function

**subalgorithm resize():**

former\_size ← [table].m

[table].m ← [table].m \* 2

former\_elements ← new TElement[former\_size]

for i ← 0, former\_size execute:

former\_elements[i] = [table].T[i]

end-for

free(T)

[table].T ← new TElement[[table].m]

free(next)

[table].next ← new Integer[[table].m]

for i ← 0, former\_size execute:

if isEmpty(former\_elements[i]) == False then:

add(former\_elements[i])

end-if

end-for

free(former\_elements)

end-subalgorithm

Implementation for Iterator operations:

**function getCurrent():**

getCurrent ← getAll()[[ht].current]

end-function

**subalgorithm next():**

[ht].current ← [ht].current + 1

while isEmpty(getAll()[[ht].current]) == True execute:

[ht].current = [ht].current + 1

end-while

end-subalgorithm

**function valid():**

if [ht].current < getTableElements(ht) then:

valid ← true

else

valid ← false

end-function

Tests for the Multimap, HashTable, Iterator and Element classes:

#include "Tests.h"

#include "Element.h"

#include "HashTable.h"

#include "Multimap.h"

#include "Iterator.h"

#include <iostream>

void Tests::tests\_element()

{

TElement e1;

assert(e1.isEmpty() == true);

assert(e1.getKey() == -1);

assert(e1.getValue() == -1);

e1.setKey(14);

e1.setValue(15);

assert(e1.getKey() == 14);

assert(e1.getValue() == 15);

TElement e2{ 1,1 };

assert(e2.getKey() == 1);

assert(e2.getValue() == 1);

assert(e2 != e1);

assert((e2 != e2) == false);

assert(e1.isEmpty() == false);

}

void Tests::tests\_hashtable()

{

HashTable table;

assert(table.getSize() == 7);

assert(table.isEmpty() == true);

assert(table.getFirstEmpty() == 0);

assert(table.getNumberOfThings() == 0);

TElement el{ 1, 2 };

assert(table.getHash(el.getKey()) == 1);

TElement el2{ 1, 5 };

assert(table.getHash(el2.getKey()) == 1);

assert(table.search(el2) == false);

table.add(el);

assert(table.getNumberOfThings() == 1);

assert(table.isEmpty() == false);

assert(table.search(el) == true);

assert((table.getT()[1] != el) == false);

assert(table.getFirstEmpty() == 0);

table.add(el2);

assert(table.search(el2) == true);

assert((table.getT()[0] != el2) == false);

assert(table.getFirstEmpty() != 0);

assert(table.getFirstEmpty() == 2);

TElement el3{ 1, 6 };

assert(table.getHash(el3.getKey()) == 1);

table.add(el3);

assert(table.search(el3) == true);

assert((table.getT()[2] != el3) == false);

assert(table.getFirstEmpty() == 3);

TElement el4{ 5, 7 };

assert(table.getHash(el4.getKey()) == 5);

table.add(el4);

assert(table.search(el4) == true);

assert((table.getT()[5] != el4) == false);

assert(table.getFirstEmpty() == 3);

TElement el5{ 5, 8 };

assert(table.getHash(el5.getKey()) == 5);

table.add(el5);

assert(table.search(el5) == true);

assert((table.getT()[3] != el5) == false);

assert(table.getFirstEmpty() == 4);

TElement el6{ 5, 9 };

assert(table.getHash(el6.getKey()) == 5);

table.add(el6);

assert(table.search(el6) == true);

assert((table.getT()[4] != el6) == false);

assert(table.getFirstEmpty() == 6);

TElement el7{ 3, 9 };

assert(table.getHash(el7.getKey()) == 3);

table.add(el7);

assert(table.search(el7) == true);

assert((table.getT()[6] != el7) == false);

assert(table.getFirstEmpty() == 7);

TElement el8{ 4, 14 };

table.add(el8);

assert(table.getSize() == 14);

assert(table.search(el8) == true);

assert((table.getT()[7] != el8) == false);

assert(table.getFirstEmpty() == 8);

table.remove(el3);

assert(table.search(el3) == false);

table.remove(el7);

assert(table.search(el7) == false);

}

void Tests::tests\_multimap()

{

Multimap map;

map.add(1,3);

assert(map.getTableElements() == 7);

assert(map.search(1, 3) == true);

TElement el1{ 1,3 };

assert((map.getAll()[1] != el1) == false);

assert(map.search(1, 4) == false);

map.remove(1, 3);

assert(map.search(1, 3) == false);

}

void Tests::tests\_iterator()

{

Multimap map;

map.add(1, 3);

map.add(2, 8);

map.add(5, 6);

Iterator it{ map };

it.next();

assert(it.valid() == true);

assert((it.getCurrent() != TElement{ 1,3 }) == false);

it.next();

assert((it.getCurrent() != TElement(2, 8)) == false);

it.next();

assert(it.valid() == true);

}