ADT Multimap on a Hash Table, Collision Resolution by Coalesced Chaining

A Multimap is a container in which the elements are stored as pairs of the form (k, v), where k represents a key and v a value associated to that key. A specific property of a Multimap is the fact that it does not have positions, the information being accessible only by using the specific key. Also, to a key may be associated multiple values, contrary to a simple Map, where the pair (k, v) was unique.

A collision resolution by coalesced chaining represents a way of accessing the elements by using a next field, which contains an array of integers that point to different positions of different elements.

DOMAIN

MM = {mm | mm is a Multimap with pairs e = (k, v), k ∈ TKey, v ∈ TValue}

REPRESENTATION OF THE CONTAINER

Multimap: TElems:

- elems: TElem[] key: TKey
- cap: Integer value: TValue
- firstEmpty: Integer next: Integer
- h: TFunction

INTERFACE

init(mm, h) desc: Creates a new empty Multimap. pre: *true* post: mm ∈ MM and it is empty.	**destroy(mm)** desc: Destroys a Multimap. pre: mm ∈ MM post: The multimap was destroyed.	**add(mm, k, v)** desc: Adds a new pair to the Multimap. pre: mm ∈ MM , k ∈ TKey, v ∈ TValue post: mm0 ∈ MM , mm0 = mm ⨁ (k, v)
remove(mm, k, v) desc: Removes a key value pair from the Multimap. pre: mm ∈ MM , k ∈ TKey, v ∈ TValue post: true, if (k, v) ∈ mm, mm0 ∈ MM , mm0 = mm − (k, v) false, otherwise.	**search(mm, k)** desc: Returns a list with all the values associated to a key. pre: mm ∈ MM , k ∈ TKey post: l ∈ L, l is the list of values associated to the key k. If k is not in the Multimap, l is the empty list.	**iterator(mm, it)** desc: Returns an iterator over the Multimap. pre: mm ∈ MM post: it ∈ *I* , it is an iterator over mm, the current element from it is the first pair from mm, or, it is invalid, if mm is empty.
	size(mm) desc: Returns the size of the Multimap pre: mm ∈ MM post: The size of the Multimap	

ITERATOR

DOMAIN:

I = {it | it is an iterator over a Multimap, having the elements of type TElems, which has in its structure aTKey and a TValue}

REPRESENTATION OF THE ITERATOR

Iterator:

mm: Multimap currentElement:
Integer

init(it, mm) desc: Creates an iterator over mm ∈ MM pre: it ∈ *I* , mm ∈ MM post: it ∈ *I*, is an iterator over mm and it points to the first element in the mm, if mm is not empty.	**getCurrent(it)** desc: Returns the current element from the Multimap. pre: it ∈ *I* , it must be valid post: getCurrent ← the current element of the iterator. exceptions : Throws an exception if the iterator it is not valid.
next(it) desc: Sets the current element to the next one or it makes it invalid if there are no more elements in the hash table. pre: it ∈ *I* , it must be valid post: The current element from it points to the next element. exceptions : Throws an exception if the iterator it is not valid.	**valid(it)** desc: Checks if the iterator is valid, thus returning the *true* value, or not. pre: it ∈ *I* post: valid ← *true* if the iterator is valid, *false* otherwise.

PROBLEM STATEMENT

Let us consider a library with m different books. These books are stored in a database, where they can be accessed by a user via the name of the author. We suppose that there may be more books with the same author. The library needs an interactive program which helps the librarians to manage the database of books, using C.R.U.D. operations.

PROBLEM JUSTIFICATION

This problem is suitable for a Multimap due to the fact that we may consider the author as a unique key, but to that key may correspond different titles, making this a good analogy for a Multimap. We can use a hash table for this specific problem, because is faster than other ADTs, due to the hash function, which allocates in Θ(1) time an index where the book can be stored and is also useful for the search function.

IMPLEMENTATION IN PSEUDOCODE OF THE OPERATIONS

```
---------------------------------------------------------------------------

function add(mm, key, value) is: book.key
      <- key book.value <- value index <-
      mm.h(book.key, mm.cap) if
      mm.TElems[index].book = book then
            @The execution of the function stops end_if
      if index = m.firstEmpty then
            mm.TElems[index].book <- book while
            mm.TElems[index].book.key ≠ “” and
mm.TElems[index].book.value ≠ “” execute mm.firstEmpty
                  <- mm.firstEmpty + 1
            end_while
      else getNext <- mm.TElems[index].next
            if getNext = -1 then mm.TElems[mm.firstEmpty].book
                  <- book mm.TElems[index].next <-
                  mm.firstEmpty
            else while mm.TElems[getNext].next ≠ -1 and
mm.TElems[getNext].book ≠ book execute getNext <-
                        mm.TElems[getNext].next
                  end_while
                  if mm.TElems[mm.firstEmpty].book = book then
                        @the execution of the function stops
                  end_if mm.TElems[mm.firstEmpty].book <-
                  book mm.TElems[getNext].next <-
                  mm.firstEmpty
            end_if
            while mm.TElems[mm.firstEmpty].book.key ≠ “” execute
                  mm.firstEmpty <- mm.firstEmpty + 1
            end_while end_if
            end_function

---------------------------------------------------------------------------

function remove(mm, key, value) is:
      book.key <- key book.value <-
      value i <- mm.h(book.key,
      mm.cap) j <- -1 index <- 0
      while index < mm.cap and j = -1 execute
            if mm.TElems[index].next = i then
            j ← index
            else  index ← index + 1
            end_if  end_while
      while i ≠ -1 and mm.TElems[i] ≠ k execute j
            ← i
            i ← mm.TElems[i].next
      end_while  if i = -1
      then  remove <-
      false
      else over ← false  repeat  p ←
            mm.TELems[i].next  prev_p ←
            i
                  while p ≠ -1 and mm.h(mm.TElems[p]) ≠ i execute
                        prev_p ← p  p ← mm.TElems[p].next
                  end_while  if p =
                  -1 then  over ←
                  true
                  else  mm.TElems[i] ← mm.TElems[p]
                        j ← prev_p  i ← p
                  end_if
            until over  if j ≠ -1 then
            mm.TElems[j].next ← mm.TElems[i].next
            end_if  mm.TELems[i].book.key ←
            “empty”
            mm.TElems[i].book.value <-
            “empty” mm.TElems[i].next ← -1
            if mm.firstFree > i then
            mm.firstFree ← i
            end_if
      end_if  remove
      <- true
end_function

---------------------------------------------------------------------------
```

```
function search(mm, key) is: index
      <- mm.h(key, mm.cap) pos <-
      0
      while mm.TElems[index].next ≠ -1 execute if
            mm.TElems[index].book.key = key then list[pos]
            <- mm.TElems[index].book.value pos <- pos + 1
            end_if end_while
      if mm.TElems[index].book.key = key then list[pos]
            <- mm.TElems[index].book.value pos <- pos +
            1
      enf_if search
      <- list
end_function
```

```
function size(mm) is: size
      <- 0
      for i<-0, mm.cap execute if mm.TElems[i].book.key ≠ key and
            mm.TElems[i].book.value ≠
value and mm.TElems[i].book.key ≠ "empty" and mm.TElems[i].book.value ≠
"empty" then size <- size + 1
            end_if
      end_for size
      <- size
end_function
```

```
function init(mm, h) is: mm.cap <- 120
      mm.firstEmpty <- 0 mm.TElems <- ↑
      TElements[cap] for i <- 0, mm.cap
      execute mm.TElems[i].book.key <- ""
      mm.TElems[i].book.value <- ""
      mm.TElems[i].book.next <- -1
      end_for mm.h
      <- h
end_function
```

```
function destroy(mm) is:
      @destroy the Multimap end_function
```

```
function iterator(mm) is: iterator
      <- Iterator(↑ mm)
```

```
end_function
```

IMPLEMENTATION IN PSEUDOCODE OF THE ITERATOR

```
function init(it, mm) is: it.mm <- mm
      it.currentElement <- 0 if
      it.mm.size() = 0 then
      it.currentElement <- it.mm.cap
      else while it.currentElement < it.mm.cap and
it.mm.TElems[it.currentElement].book.key = “” and
it.mm.TElems[it.currentElement].book.value = “” execute it.currentElement
<- it.currentElement + 1
            end_while end_if
            end_function
```

```
function destroy(it) is:
      @Destroy the Iterator of the Multimap. end_function
```

```
function getCurrent(it) is: if it.valid() = true then
      getCurrent <- it.mm.TElems[it.currentElement].book
      else
            @Throw exception.
end_if end_function
```

```
function next(it) is: if
      it.valid() = false then
            @Throw exception. it.currentElement
      <- it.currentElement + 1 while
      it.currentElement < it.mm.cap and
it.mm.TElems[it.currentElement].book.key = “” and
it.mm.TElems[it.currentElement].book.value = “” execute it.currentElement
<- it.currentElement + 1
            end_while end_if
            end_function
```

```
function valid(it) is:  if
      it.currentElement < it.mm.cap then
      valid <- true
      else valid <- false
      end_if end_function
```

TESTS FOR THE OPERATIONS OF THE MULTIMAP

```
#include "Tests.h"
 int h(const string& author, int
cap)
{ int s = 0;
      for (int i = 0; i < author.size(); i++) s
            += (int)(author[i]);
      return s % cap;
}  void
Test::test_create()
{
      Multimap mm{h}; assert(mm.getCapacity()
      == 120); assert(mm.getFirstEmpty() ==
      0);
}

void Test::test_size()
{
      Multimap mm{h}; pair<string,
      string> book1; pair<string,
      string> book2; book1.first =
      "Author1"; book1.second =
      "Book1"; book2.first = "Author2";
      book2.second = "Book2";
      assert(mm.size() == 0);
      mm.add(book1.first,
      book1.second); assert(mm.size()
      == 1); mm.add(book2.first,
      book2.second); assert(mm.size()
      == 2);
}
void Test::test_add() {
      Multimap mm{h}; pair<string,
      string> book1; pair<string,
      string> book2; pair<string,
      string> book3; pair<string,
      string> book4; book1.first =
      "Author1"; book1.second =
      "Book1"; book2.first = "Author1";
      book2.second = "Book1";
      book3.first = "Author2";
      book3.second = "Book2";
      book4.first = "Author2";
      book4.second = "Book3";
      mm.add(book1.first,
      book1.second); assert(mm.size()
      == 1); mm.add(book2.first,
      book2.second); assert(mm.size()
      == 1); mm.add(book3.first,
      book3.second); assert(mm.size()
      == 2); mm.add(book4.first,
      book4.second); assert(mm.size()
      == 3);
}
void Test::test_delete()
{
      Multimap mm{h}; pair<string,
      string> book1; pair<string,
      string> book2; pair<string,
      string> book3; pair<string,
      string> book4; book1.first
      = "Author1"; book1.second =
      "Book1"; book2.first =
      "Author1"; book2.second =
      "Book1"; book3.first =
      "Author2"; book3.second =
      "Book2"; book4.first =
      "Author2"; book4.second =
      "Book3";
      mm.add(book1.first,
      book1.second);
      mm.add(book2.first,
      book2.second);
      mm.add(book3.first,
      book3.second);
      mm.add(book4.first,
      book4.second);
      assert(mm.remove(book1.firs
      t, book1.second) == true);
      assert(mm.remove(book2.firs
      t, book2.second) == false);
      assert(mm.remove(book3.firs
      t, book3.second) == true);
      assert(mm.remove(book4.firs
      t, book4.second) == true);
      assert(mm.remove(book4.firs
      t, book4.second) == false);
 } void
Test::test_search()
{
      Multimap mm{h}; pair<string, string> book1;
      pair<string, string> book2; pair<string,
      string> book3; pair<string, string> book4;
      pair<string, string> book5; book1.first =
      "Author1"; book1.second = "Book1";
      book2.first = "Author1"; book2.second =
      "Book1"; book3.first = "Author2";
      book3.second = "Book2"; book4.first =
      "Author2"; book4.second = "Book3";
      book5.first = "Author0"; book5.second =
      "Book0"; mm.add(book1.first,
      book1.second); mm.add(book2.first,
      book2.second); mm.add(book3.first,
      book3.second); mm.add(book4.first,
      book4.second); vector<string> list =
      mm.search("Author1"); assert(list.size()
      == 1); list = mm.search("Author2");
      assert(list.size() == 2); list =
      mm.search("Author0"); assert(list.size()
      == 0);
}
```

```
  void
Test::test_iterator()
{
      Multimap mm{h};
      Iterator it = mm.iterator(); try
      { while (!it.valid())
            { it.getCurrent();
                  it.next();
                  assert(false);
            }
      } catch (string&
      ex)
      { assert(true);
      }
}
```

IMPLEMENTATION IN PSEUDOCODE OF THE PROBLEM

```
function  printMenu(ui) is: print("Welcome to the library! Here are the
      tools that you need in
order to modify the database of the library:") print("1)
      Add a new book to the database.") print("2) Remove
      a book from the database.")
      print("3) Given a specific author, get a list with all the books
written by that author.") print("4) Get all the books
      from the database.") print("0) Exit the
      application.")
```

```
end_function

function populate(ui) is: book1.key <- "Ray
      Bradbury" book1.value <- "Fahrenheit
      451" book2.key <- "Orson Scott Card"
      book2.value <- "Ender's Game"
      book3.key <- "George Orwell"
      book3.value <- "1984" book4.key <-
      "Liviu Rebreanu" book4.value <- "Ion"
      book5.key <- "Liviu Rebreanu"
      book5.value <- "Padurea Spanzuratilor"
      book6.key <- "Liviu Rebreanu"
      book6.value <- "Catastrofa"
      ui.mm.add(book1.key, book1.value)
      ui.mm.add(book2.key, book2.value)
      ui.mm.add(book3.key, book3.value)
      ui.mm.add(book4.key, book4.value)
      ui.mm.add(book5.key, book5.value)
      ui.mm.add(book6.key, book6.value)
end_function

function start(ui) is:
      ui.populate() while
      true execute
      ui.printMenu option <-
      -1 @Read the option
            while option < 0 or option > 4 execute
                  @Validate the option.
            end_while if option = 0
            then
                  @Stops the execution of the function.
            switch option case 1:
                        @Read the key and the value.
                        ui.mm.add(key, value) break
                  end_case case
                  2:
                        @Read the key and the value. if
                        ui.mm.remove(key, value) = true then
                        print("The removal was successful.")
                        else print("Error! The book you are trying to
      remove does not exist.")
                        end_if break
                  end_case case
                  3:
                        @Read the key. list <-
                        ui.mm.search(key) if
                        list.size() = 0 then
                              @Print a message else for i <-
                        0, list.size() execute
                                    @Print the content of the list end_for
                        end_if
                        break
                  end_case case 4: it <-
                  ui.mm.iterator() while it.valid() =
                  true execute book <- it.getCurrent()
                              if book.key ≠ "empty" then
                                    @Print the book.
                              end_if it.next()
                        end_while  break
                        end_case case
                        0: break
                        end_case
                        default: break
                        end_switch
                        end_while
                        end_function

function readBook() is:
      @Read the key and the value for the book. read
      <- book
end_function
```

COMPLEXITIES FOR THE OPERATIONS OF THE MULTIMAP AND FOR THE ITERATOR OF THE MULTIMAP

❑ init

- ❑ Complexity: Θ(n), where is the the capacity of the Multimap. Here, we initialize every element with the pair ("", "") and we go through every single element.

❑ destroy

- ❑ Complexity: Θ(1)

❑ add

- ❑ Complexity:
  - ❑ Best Case: Θ(1), when we add on a position that is not occupied.
  - ❑ Worst Case: Θ(n), where we need to go through all the elements of the Multimap and add to the last available position and after that we need to increase the value of firstEmpty, if it is no longer valid.
  - ❑ Average Case: O(n), where we add on a position that is not empty, but it is not the last in the hash table.

- remove
    - Complexity:
        - Best Case: Θ(1), when the element that we want to remove does not exist.
        - Worst Case: Θ(n), where the element that we need to remove needs to be searched for, starting from the element at the current index and continuing using the field next unti the last position.
        - Average Case: O(n), where we need to remove a random element that is somewhere in the middle.
- search
    - Complexity: Θ(n), where we need to go through all the elements that have the same key as the one given as a parameter of the function *searched* , using the field next, where n corresponds to the number of items that have this specific key.
- size
    - Complexity: Θ(n), where we need to count every position in the Multimap that is occupied.
- iterator
    - Complexity: Θ(1). The function just returns an iterator for the Multimap mm.