**ADT Multi Map - Project Stage**

# Munteau Tudor

# Project Topic No. 2

# **Problem statement and Justification**

* Problem statement

A restaurant owner is just opening a restaurant in a city and he wishes to organize his food inventory based on the ingredient types. Help the restaurant owner by creating a program that can save an ingredient for each food category. For example:

* Fruits: apple, grapes, cranberries
* Meat: beef, pork, chicken
* Seafood: crab, lobster, shrimp

The program needs to be efficient in terms of memory and performance, as the restaurant’s server does not have the best configuration.

Your program should let the user to use the following operations:

* Add an item to a specified category
* Remove an item
* Remove a whole category
* Search for an item in a specified category
* Get all the items that belong to a category
* Get the number of items
* Show all the items from the inventory
* Justification

I will have to say that a Multi Map is a best fit for this type of requirements (multiple values that have the same key, unlimited number of pairs), it allows the insertion of several pairs which have identical keys. This will result in the program being dynamic (for example, inserting ingredients and removing a specific ingredient while the program is running).

Why is it better to use Multi Map instead of other ADTs?

1. It is efficient in terms of inserting items and finding all items that share a key (due to the implementation in memory: coalesced chaining)
2. It is efficient in terms of memory (not a lot of memory used for implementation)

Why is a Multi Map(collision resolution by coalesced chaining) inefficient?

This program is capable to dynamically extend its memory; however, this aspect is costly ifthere are many items. We will initially consider a bigger number of elements. Dynamic reallocation of memory must be avoided because of the implementation (Hash table). Ifreallocation is needed, every element must be re-hashed.

# **Interface of the container**

Class ADTMultiMap

* ADTMultiMap() – constructor, this will allocate initial memory for our data structure
* void insert(const std::string & Key, const std::string & Value) – method of our class, will insert a new entry into our Multi Map, ifneeded it will resize and rehash(every element) the Multi Map
* void search(const std::string & Key, std::vector<std::string> & Values) – method of our class, will store each value that has as key the key that was given as a parameter in the vector ‘Values’
* void deletePair(const std::string & Key, const std::string & Value) – deletes a pair<key,value> from the multimap
* void deleteKey(const std::string & Key) – deletes every pair<key,value> that has as key the key that was given as a parameter
* void clear() – deletes every pair<key,value> from the multimap
* bool containsPair(const std::string & Key, const std::string & Value) – returns true ifthere exists a pair<Key,Value> in the multimap, else false
* bool isEmpty() – returns true ifthe multimap is empty, else false
* ADT\_MultiMap\_Pair \* getArray() – returns the array that is used in the iterator
* int size() – returns the number of elements that are in the multimap
* void createIterator() – this function will create/reset the iterator of our class. Note that our class can have only one iterator
* ADTMultiMapIterator \* getIterator() – returns the iterator of our class. Will create a new iterator **if** there was no iterator created be**for**e

Class ADTMultiMapIterator

* ADTMultiMapIterator(ADTMultMap) – constructor, this will alocate initial memory for our class
* bool valid() – returns true ifthe iterator is still valid(not out of bounds)
* std::string & getCurrentKey() – returns the current key ifthe iterator is still valid
* std::string & getCurrentValue() – returns the current value ifthe iterator is still valid
* bool next() – moves the iterator to the next pair and returns true ifthe iterator is valid, either false

# **Representation of the Multi Map**

**Class ADTMultiMap**

* **ADTMultiMap\_Pair**: – struct used to organize every pair in our Multi Map
* Key: String
* Value: String
* Array: ADT\_MultiMap\_Pair[] – array with all the keys and values
* Next: Integer[] – array corresponding to ‘Array’, used to fulfill the coalesced chaining principle
* M: Integer – size of the Multi Map
* FirstEmpty: Integer – index of the first empty location in the ‘Array’
* HashFunc: ↑Function – pointer to the HashFunction

**Class ADTMultiMapIterator**

* HashTable: ↑ADTMultiMap
* currentPosition: Integer

# **Implementation**

**Class ADTMultiMap**

**subalgorithm** init(mMap) is:

allocate([mMap].Array)

allocate([mMap].Next)

[mMap].Count <- 0

[mMap].MyIterator <- NIL

[mMap].M <- SIZE\_OF\_MULTIMAP

[mMap].FirstEmpty <- [mMap].M – 1

[mMap].HashFunc <- [mMap].divisionHash

**end-subalgorithm | Time Complexity: Theta(1)**

**function** divisionHash(mMap, key) is:

sum <- 0

**for** i <- 0, i < length(key), i++ **execute**:

sum <- sum + key[i]

**end-for**

divisionHash <- sum % ADDRESS\_MAP\_OF\_VALUES

**end-function | Time Complexity: O(n), where n is the length of the key**

**subalgorithm** changeFirstEmpty(mMap) is:

**while** [mMap].FirstEmpty >= 0 **and** [mMap].Array[[mMap].FirstEmpty].Key is not empty **execute**:

[mMap].FirstEmpty <- [mMap].FirstEmpty – 1

**end-while**

**end-subalgorithm | Time Compelxity: O(M)**

**subalgorithm** destroy(mMap) is:

free([mMap].Next)

free([mMap].Array)

free([mMap].MyIterator)

**end-subalgorithm | Time Complexity: Theta(1)**

**subalgorithm** insert(mMap, Key, Value) is:

@validate Key

@validate Value

mKeyPosition <- [mMap].HashFunction(Key)

**if** [mMap].Array[mKeyPosition].Key is empty **execute**:

[mMap].Array[mKeyPosition].Key <-Key

[mMap].Array[mKeyPosition].Value <-Value

[mMap].Next[mKeyPosition] <--1

[mMap].Count <- [mMap].Count + 1

@end-subalgorithm

**end-if**

[mMap].changeFirstEmpty()

**if** [mMap].FirstEmpty is -1 **execute**:

@ reallocate

@ rehash

mKeyPosition <- [mMap].HashFunc(Key)

**if** [mMap].Array[mKeyPosition].Key is empty **execute**:

[mMap].Array[mKeyPosition].Key <-Key

[mMap].Array[mKeyPosition].Value <-Value

[mMap].Next[mKeyPosition] <--1

[mMap].Count <- [mMap].Count + 1

@end-subalgorithm

**end-if**

[mMap].changeFirstEmpty()

**end-if**

current <- mKeyPosition

**while** [mMap].Next[current] is not -1 **execute**:

current <- [mMap].Next[current]

**end-while**

[mMap].Array[[mMap].FirstEmpty].Key <- Key

[mMap].Array[[mMap].FirstEmpty].Value <- Value

[mMap].Next[[mMap].FirstEmpty] <- -1

[mMap].Next[current] <- [mMap].FirstEmpty

[mMap].Count <- [mMap].Count + 1

**end-subalgorithm | Time Complexity: Theta(1) – best case, O(M) – worst case**

**subalgorithm** search(mMap, Key, Values) is:

currentPos <- [mMap].HashFunc(Key)

**while** currentPos is not -1 **and** [mMap].Array[currentPos].Key is not empty **execute**:

@ Values.push\_back([mMap].Array[currentPos].Value)

**end-while**

**end-subalgorithm | Time Complexity: O(n), n is the number of occurs**

**subalgorithm** deletePair(mMap, Key, Value) is:

@validate Key

@validate Value

mKeyPos <- [mMap].HashFunc(Key)

lastPosition <- -1

**while** mKeyPos = -1 **and** [mMap].Array[mKeyPos].Key is not empty **execute**:

if [mMap].Array[mKeyPos].Key = Key **and** [mMap].Array[mKeyPos].Value = value execute:

[mMap].Count <- [mMap].Count – 1

**If** [mMap].Next[mKeyPos] = 1 **execute**:

@ [mMap].Array[mKeyPos].Key.clear()

**Else**

[mMap].Array[mKeyPos].Key <- [mMap].Array[[mMap].Next[mKeyPos]].Key

[mMap].Array[mKeyPos].Value <- [mMap].Array[[mMap].Next[mKeyPos]].Value

**If** lastPosition is not -1 **execute**:

[mMap].Next[lastPosition] = mKeyPos

**end-if**

@ [mMap].Array[[mMap].Next[mKeyPos]].Key.clear()

@ [mMap].Array[[mMap].Next[mKeyPos]].Value.clear()

**If** [mMap].Array[mKeyPos].Key is empty **execute**:

[mMap].Next[mKeyPos] <- -1

**Else**

[mMap].Next[mKeyPos] <- [mMap].Next[[mMap].Next[mKeyPos]]

**End-if**

@return

**End-if**

**End-if**

lastPosition <- mKeyPos

mKeyPos <- [mMap].Next[mKeyPos]

**end-while**

@throw exception(“The pair does not exists”)

**end-subalgorithm | Time Complexity: Theta(1) in most cases / O(n), n is the number of duplicates**

**subalgorithm** deleteKey(mMap, Key) is:

@validate Key

mKeyPos <- [mMap].HashFunc(Key)

lastPositon <- -1

deleted <- false

**while** mKeyPos is not -1 **and** [mMap].Array[mKeyPos].Key is not empty **execute**:

**if** [mMap].Array[mKeyPos].Key = Key **execute**:

[mMap].Count <- [mMap].Count – 1

deleted <- true

**if** [mMap].Next[mKeyPos] = - 1 **execute**:

@ [mMap].Array[mKeyPos].Key.clear()

**Else**

[mMap].Array[mKeyPos].Key <- [mMap].Array[[mMap].Next[mKeyPos]].Key

[mMap].Array[mKeyPos].Value <- [mMap].Array[[mMap].Next[mKeyPos]].Value

**If** lastPosition is not -1 **execute**:

[mMap].Next[lastPosition] = mKeyPos

**end-if**

@ [mMap].Array[[mMap].Next[mKeyPos]].Key.clear()

@ [mMap].Array[[mMap].Next[mKeyPos]].Value.clear()

lastPositon <- [mMap].Next[mKeyPos]

**If** [mMap].Array[mKeyPos].Key is empty **execute**:

[mMap].Next[mKeyPos] <- -1

**Else**

[mMap].Next[mKeyPos] <- [mMap].Next[[mMap].Next[mKeyPos]]

**End-if**

**End-if**

**Else**

mKeyPos <- [mMap].Next[mKeyPos]

lastPosition <- mKeyPos

**end-if**

**end-while**

**if** deleted is not true **execute:**

@ throw excaption(“The category does not exists”)

**End-if**

**end-subalgorithm | Time Complexity: O(n)**

**function** count(mMap) is:

count <- [mMap].Count

**end-function | Time Complexity: Theta(1)**

**function**  size(mMap) is:

size <- [mMap].M

**end-function | Time Complexity: Theta(1)**

**function** isEmpty(mMap) is:

isEmpty <- [mMap].Count is 0

**end-function | Time Complexity: Theta(1)**

**subalgorithm** clear(mMap) is:

free([mMap].Array)

free([mMap].Next)

[mMap].M <- SIZE\_OF\_MULTIMAP

[mMap].Count <- 0

[mMap].FirstEmpty <- [mMap].M – 1

allocate([mMap].Array)

allocate([mMap].Next)

**end-subalgorithm | Time Complexity: Theta(1)**

**function** getArray(mMap) is:

getArray <- [mMap].Array

**end-function | Time Complexity: Theta(1)**

**subalgorithm** createIterator(mMap) is:

**if** [mMap].MyIterator is NIL **execute**:

free([mMap].MyIterator)

**end-if**

allocate([mMap].MyIterator)

**end-subalgorithm | Time Complexity: Theta(1)**

**function** getIterator(mMap) is:

**if** [mMap].Count is 0 **execute**:

getIterator <- NIL

**end-if**

**if** [mMap].MyIterator is NIL **execute**:

[mMap].createIterator()

**end-if**

getIterator <- [mMap].MyIterator

**end-function | Time Complexity: Theta(1)**

**function** containsPair(mMap, Key, Value) is:

@validate Key

@validate Value

mKeyPos <- [mMap].HashFunc(Key)

**while** mKeyPos is not -1 and [mMap].Array[mKeyPos].Key is not empty **execute**:

**if** [mMap].Array[mKeyPos].Key is Key **execute**:

containsPair <- true

**end-if**

mKeyPos <- [mMap].Next[mKeyPos]

**end-while**

containsPair <- false

**end-function | Time Complexity: Theta(1) in case there is no duplicate, O(N) else**

**Class ADTMultiMapIterator**

**subalgorithm** init(mIterator, mMap) is:

[mIterator].mHashTable <- mMap

currentPosition <- 0

**while** [mIterator].valid() is true and [mIterator].mHashTable.getArray()[[mIterator].currentPosition].Key is empty **execute**:

[mIterator].currentPosition <- [mIterator].currentPosition + 1

**end-while**

**end-subalgorithm | Time Complexity: Theta(1) – most cases**

**function** valid(mIterator) is:

**if** [mIterator].CurrentPosition >= 0 and [mIterator].CurrentPosition < [mIterator].mHashTable.size() **execute**:

valid <- true

**end-if**

valid <- false

**end-function | Time Complexity: Theta(1)**

**function** next(mIt) is:

[mIt].CurrentPosition <- [mIt].CurrentPosition + 1

**while** [mIt].valid() is true and [mit].mHashTable.getArray()[[mIt].currentPosition].Key is not empty **execute**:

[mIt].CurrentPosition <- [mIt].CurrentPosition + 1

**end-while**

**if** [mIt].valid() is true **execute**:

next <- true

**end-if**

next <- false

**end-function | Time Complexity: O(1)**

**function** getCurrentKey(mIt) is:

**if** [mIt].valid() is not true **execute**:

@throw exception

**End-if**

getCurrentKey <- [mIt].mHashTable.getArray()[[mIt].currentPosition].Key

**end-function | Time Complexity: Theta(1)**

**function** getCurrentValue(mIt) is:

**if** [mIt].valid() is not true **execute**:

@throw exception

**End-if**

getCurrentValue <- [mIt].mHashTable.getArray()[[mIt].currentPosition].Value

**end-function | Time Complexity: Theta(1)**

# **Testing**

void testInsert(ADTMultiMap\* mMultiMap)

{

std::cout << "| Test #1 - Inserting 1000 different elements...\n";

for (int i = 0; i < 1000; i++)

{

mMultiMap->insert(std::to\_string(i), std::to\_string(i));

}

std::cout << "| Test #1 - Inserting 1000 different elements - Success\n";

std::cout << "| Test #1 - Inserting 1000 duplicates...\n";

for (int i = 0; i < 1000; i++)

{

mMultiMap->insert(std::to\_string(0), std::to\_string(0));

}

std::cout << "| Test #1 - Inserting 1000 duplicates - Success\n";

assert(mMultiMap->count() == 2000);

std::cout << "| Test #1 - Success\n";

}

void testDelete(ADTMultiMap\* mMultiMap)

{

std::cout << "| Test #2 - Removing 1000 different elements...\n";

for (int i = 0; i < 1000; i++)

{

mMultiMap->deletePair(std::to\_string(i), std::to\_string(i));

}

std::cout << "| Test #2 - Removing 1000 different elements - success\n";

std::cout << "| Test #2 - Removing 1000 duplicates...\n";

mMultiMap->deleteKey(std::to\_string(0));

std::cout << "| Test #2 - Removing 1000 duplicates - success\n";

assert(mMultiMap->count() == 0);

std::cout << "| Test #2 - Success\n";

}

void testClear(ADTMultiMap\* mMultiMap)

{

std::cout << "| Test #3 - Inserting 1000 elements...\n";

for (int i = 0; i < 1000; i++)

{

mMultiMap->insert(std::to\_string(i), std::to\_string(i));

}

std::cout << "| Test #3 - Inserting 1000 elements - success\n";

std::cout << "| Test #3 - Clearing...\n";

mMultiMap->clear();

assert(mMultiMap->count() == 0);

std::cout << "| Test #3 - Success\n";

}

void testSearch(ADTMultiMap\* mMultiMap)

{

std::cout << "| Test #4 - Inserting 1000 elements...\n";

for (int i = 0; i < 1000; i++)

{

mMultiMap->insert(std::to\_string(0), std::to\_string(i));

}

std::cout << "| Test #4 - Inserting 1000 elements - success\n";

std::cout << "| Test #4 - Searching...\n";

std::vector<std::string> mList;

mMultiMap->search(std::to\_string(0), mList);

for (int i = 0; i < 1000; i++)

{

assert(mMultiMap->containsPair(std::to\_string(0), std::to\_string(i)) != false);

}

assert(mList.size() == 1000);

std::cout << "| Test #4 - Success\n";

mMultiMap->clear();

}

void testSizes(ADTMultiMap\* mMultiMap)

{

std::cout << "| Test #5 - Inserting 1000 elements...\n";

int lastSize = SIZE\_OF\_MULTIMAP;

std::cout << "| Test #5 - Testing count() && size() && isEmpty()...\n";

for (int i = 0; i < 1000; i++)

{

if (!mMultiMap->isEmpty() && mMultiMap->count() % lastSize == 0)

lastSize \*= 2;

mMultiMap->insert(std::to\_string(0), std::to\_string(i));

assert(mMultiMap->count() == i + 1);

assert(mMultiMap->size() == lastSize);

}

mMultiMap->clear();

std::cout << "| Test #5 - Success\n";

}

void testIterator(ADTMultiMap\* mMultiMap)

{

std::cout << "| Test #6 - ADTMultiMap->getIterator()...\n";

ADTMultiMapIterator\* mIterator = mMultiMap->getIterator();

assert(mIterator == nullptr);

std::cout << "| Test #6 - Inserting elements...\n";

for (int i = 0; i < 1000; i++)

mMultiMap->insert(std::to\_string(i), std::to\_string(i));

mIterator = mMultiMap->getIterator();

assert(mIterator != nullptr);

int count = 0;

for (int i = 0; i < 2; i++)

{

do

{

count++;

assert(mIterator->getCurrentKey() == mIterator->getCurrentValue());

} while (mIterator->next());

assert(count == 1000);

mMultiMap->createIterator();

mIterator = mMultiMap->getIterator();

count = 0;

}

std::cout << "| Test #6 - Success\n";

}

void runTests()

{

std::cout << "| Status: Executing Tests..." << std::endl;

std::cout << "| Test #0 - ADTMultiMap()...\n";

ADTMultiMap\* mMultiMap = new ADTMultiMap();

std::cout << "| Test #0 - Success\n";

std::cout << "| Test #1 - ADTMultiMap->insert()...\n";

try

{

testInsert(mMultiMap);

}

catch (const std::exception& ex)

{

std::cout << "| Status: " << ex.what() << std::endl;

}

std::cout << "| Test #2 - ADTMultiMap->deleteKey() & ->deletePair()...\n";

try

{

testDelete(mMultiMap);

}

catch (const std::exception& ex)

{

std::cout << "| Status: " << ex.what() << std::endl;

}

std::cout << "| Test #3 - ADTMultiMap->clear()...\n";

try

{

testClear(mMultiMap);

}

catch (const std::exception& ex)

{

std::cout << "| Status: " << ex.what() << std::endl;

}

std::cout << "| Test #4 - ADTMultiMap->search() & ->containsPair()...\n";

try

{

testSearch(mMultiMap);

}

catch (const std::exception& ex)

{

std::cout << "| Status: " << ex.what() << std::endl;

}

std::cout << "| Test #5 - ADTMultiMap->count() & ->size() & ->isEmpty()...\n";

try

{

testSizes(mMultiMap);

}

catch (const std::exception& ex)

{

std::cout << "| Status: " << ex.what() << std::endl;

}

std::cout << "| Test #6 - ADTMultiMapIterator...\n";

try

{

testIterator(mMultiMap);

}

catch (const std::exception& ex)

{

std::cout << "| Status: " << ex.what() << std::endl;

}

std::cout << "| Status: Testing is done...\n";

delete mMultiMap;

system("pause");

system("cls");

}