Struc Razvan Nicolae 917

- **ADT MultiMap implementation on a hash table, collision resolutin by open addressing**

- **ADT MultiMap:**

  It' s a container that is a generalization of a map abstract data type in which more than one value may be associated with and returned for a given key.

  A MultiMap is a container in which the elements are pairs (key, value), and the keys don't have to be unique

  There are no positions in a MultiMap.

  - **Domain:**

  MM = {mm | mm is a multimap with elements (k, v), k∈ TKey, v∈ TValue}

  - **Interface:**

- subalgorithm init(mm):

  Description: creates a new empty multimap

  Pre: true

  Post: mm ∈ MM, mm is an empty multimap

- subalgorithm destroy(mm):

  Description: destroys a multimap

  Pre: mm ∈ MM, mm is a multimap

  Post: mm is destroyed

- subalgorithm add(mm, key, value):

  Description: add a new <key, value> pair to the multimap

  Pre: mm $\in$ MM, key $\in$ TKey, value ϵ TValue

  Post: mm' $\in$ MM, mm' = mm $\cup$ <key, value>

- subalgorithm remove(mm, key, value):

  Description: removes a given <key, value> pair from the multimap

  Pre: mm $\in$ MM, key $\in$ TKey, value $\in$ TValue

  Post: mm' $\in$ MM, mm' = mm / <key, value>

- subalgorithm iterator(mm, it)

  Description: iterates through all the elements from the multimap

  Pre: mm $\in$ MM

  Post: it $\in$ I, it is an iterator over MultiMap

- **ADT MultiMap Iterator:**

  - Domain:

    I= { it| it is an iterator over a MultiMap mm)

  - Interface:

- subalgorithm init(mm, it)

  Description: copy of the MM

  Pre: mm $\in$ MM

  Post: it $\in$ I, it is an iterator over mm

- subalgorithm valid(it)

  Description: check if the element is valid

  Pre: it $\in$ I

  Post: True, if the current element from is a valid one / False, otherwise

- subalgorithm next(it)

  Description: go to the next element

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

  Post: I' $\in$ I, the current element from it' is the next element from it

- subalgorithm getCurrent(it)

  Description: return current element

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

  Post: elem $\in$ TKey, elem is the current element from it

- **ADT Representation:**
  - **TElem:**
    - key: Integer
    - value: Integer
  - **TFunction:**
    - i: Integer
    - And a functions getPosition()returns k%m where k is the key and m the capacity
  - **MultiMap:**
    - elements: TElem [ ]
    - capacity: Integer (capacity)
    - H: TFunction
  - **Iterator MM:**

- multiMap: MultiMap
- currentPosition: Integer

MultiMap Implementation:

```
subalgorithm init(mm, cap) is:
        mm.capacity <- cap
        mm.elems <- @ an array with cap positions
        for i=0, mm.capacity-1 execute
                mm.elems[i] <-@new TElem
        end-for
end-subalgorithm


Function add(mm, h, TElem* t) is:
        h.resetI()
        pos <- h.getPosition([t].getKey(), mm.capacity)
        while [mm.elems[pos + H.getI()]].getKey != -1 execute
                if [mm.elems[pos + h.getI()]].getKey() = [t].getKey() and [mm.elems[pos +
                                h.getI()]].getValue() = [t].getValue() then
                        add <- 0
                end-if
                h.incrementI()
                if pos + h.getI >= mm.capacity then
                        h.resetI()
                        pos <- 0
                end-if
        end-while
        mm.elems[pos + h.getI()] <- t
```

```
        add <- 1
end-function

Function remove(mm, h, TElem* t) is:
        h.resetI()
        while [mm.elems[H.getI()]].getKey != [t].getKey() or [mm.elems[h.getI()]].getValue()
                                                != [t].getValue() then
                h.incrementI()
                if h.getPosition([t].getKey(), mm.capacity) + h.getI() >= mm.capacity
                        remove <- 0
                end-if
        end-while
        @free mm.elems[h.getI()]
        mm.elems[h.getI()] <- @ new TElem
        remove <- 1
end-function

subalgorithm iterator(mm, it) is:
        while it.valid() execute
                if [it.getCurrent()].getKey != -1
                        @print
                end-if
                it.next()
        end-while
end-subalgorithm
```

Iterator Implementation:

```
subalgorithm init(it, mm) is:
        it.multiMap <- mm
        it.currentPosition = 0
        it.first()
```

```
end-subalgorithm

subalgorithm first(it) is:
        if it.valid = false then
                it.next()
        end-if
end-subalgorithm

function valid(it) is:
        if it.currentPosition < it.multiMap.capacity
                valid <- true
        valid <- false
end-function

subalgorithm next(it) is:
        it.currentPosition <- it.currentPosition + 1
end-subalgorithm
```

Tests for the container:

```
void Test::testTElem()
{
   //TESTS FORM TELEM
   TElem x{ 1,2 };
   assert(x.getKey() == 1);
   assert(x.getValue() == 2);
   TElem y;
   assert(y.getKey() == -1);
   assert(y.getValue() == -1);
}
```

```
void Test::testTFunction()
{
    //TESTS FOR TFUNCTON
    TFunction H;
    assert(H.getPosition(5, 10) == 5);
    assert(H.getPosition(6, 10) + H.getI() == 6);
    H.incrementI();
    assert(H.getPosition(6, 10) + H.getI() == 7);
    H.resetI();
    assert(H.getPosition(6, 10) + H.getI() == 6);

}
```

```
void Test::testMultiMap()
{
    //TESTS FOR MULTIMAP
    MultiMap m{ 5 };
    assert(m.add(new TElem{ 3, 3 }) == 1);
    assert(m.add(new TElem{ 2, 4 }) == 1);
    assert(m.add(new TElem{ 2, 5 }) == 1);
    assert(m.add(new TElem{ 5, 5 }) == 1);
    assert(m.add(new TElem{ 5, 5 }) == 0);
    assert(m.add(new TElem{ 1, 3 }) == 1);
    TElem x{ 2,4 };
    assert(m.element(2)->getKey() == x.getKey());
    assert(m.element(2)->getValue() == x.getValue());
```

```
    m.remove(new TElem{ 2, 4 });
    assert(m.element(2)->getKey() == -1);
    m.add(new TElem{ 1, 4 });
    assert(m.getCapacity() == 5);
    TElem x2{ 1,4 };
    assert(m.element(2)->getKey() == x2.getKey());
    assert(m.element(2)->getValue() == x2.getValue());
    MultiMap m1{ 4 };
    assert(m1.add(new TElem{ 3, 3 }) == 1);
    assert(m1.add(new TElem{ 3, 4 }) == 1);
    assert(m1.add(new TElem{ 3, 5 }) == 1);
    assert(m1.add(new TElem{ 3, 6 }) == 1);
    assert(m1.remove(new TElem{ 3,12 }) == 0);
}
```

void Test::testIterator()

```
{
    //TESTS FOR ITERATOR
    MultiMap *m=new MultiMap(5);
    m->add(new TElem{ 3, 3 });
    m->add(new TElem{ 2, 4 });
    m->add(new TElem{ 3, 4 });
    m->add(new TElem{ 5, 5 });
    //m.add(1, 3);

    MultiMap::Iterator iter(*m);
    m->iterator(iter);
```

```
    assert(iter.valid() == true);
    assert(iter.getCurrent()->getKey() == 5);
    iter.next();
    assert(iter.getCurrent()->getKey() == -1);
    iter.next();
    assert(iter.getCurrent()->getKey() == 2);
    iter.next();
    assert(iter.getCurrent()->getKey() == 3);
}
```

- **Problem statement:**

We have a school where each student has a key and each course has also a key, we need a tool that allows us to add or remove a pair student,value from a list. The student key and the course key may both appear more than once in the list but the pair (student,course) must pe unique.

For example: pair 1,3; pair 1,4, pair 2,3

- **Justification**

We can solve this problem using MultiMap because we can have a student that is enrolled in more than 1 course.