Name: Savina Ruxandra
Group: 814

DSA – Project

Project topic: ADT Bag – implementation on a hash table, collision resolution by coalesced chaining. Store unique elements with their frequencies.

ADT Bag – Domain and interface

- **Domain:**

  B = {b | b is a Bag with elements of the type TElem}

- **Interface:**

  init(b)

  pre : true
  post: b ϵ *B*, b is an empty Bag

  add(b, e)

  pre: b ϵ *B*, e ϵ TElem
  post: b’ ϵ *B*, b’ = b U {e} (Telem e is added to the Bag)

  remove(b, e)

  pre: b ϵ *B*, e ϵ TElem
  post: b’ ϵ *B*, b’ = b \ {e} (one ocurrence of e was removed from the Bag)

  $$\mathrm{remove}\begin{cases}\mathrm{true, if\ an\ element\ was\ removed\ (size(b') < size(b))}\\ \mathrm{false, if\ e\ was\ not\ present\ in\ b\ (size(b') = size(b))}\end{cases}$$

  search(b, e)

  pre: b ϵ *B*, e ϵ TElem
  post: search$\begin{cases}\mathrm{true, if\ e \in B}\\ \mathrm{false, otherwise}\end{cases}$

  size(b)

  pre: b ϵ *B*
  post: size - the number of elements from b

  nrOccurrences(b, e)

  pre: b ϵ *B*, e ϵ Telem
  post: nrOccurrences – the number of occurrences of e in b

  destroy(b)

  pre: b ϵ *B*
  post: b was destroyed

  iterator(b, i)

  pre: b ϵ *B*
  post: i ϵ I, i is an iterator over b

ADT Iterator - Domain and interface

- **Domain:**

  I = {i | i is an iterator over b ϵ *B* }

- **Interface:**

<u>*init(i, b)*</u>

pre: b ϵ *B*

post: i ϵ I, i is an iterator over b. i refers to the first element of b, or it is invalid if b is empty

<u>*valid(i)*</u>

pre: i ϵ I

post: valid $\begin{cases} \text{true, if the current element from I is a valid one} \\ \text{false, otherwise} \end{cases}$

<u>*first(i)*</u>

pre: i ϵ I

post: i’ ϵ I, the current element from i’ refers to the first element from the bag or i is invalid if the bag is empty

<u>*next(i)*</u>

pre: i ϵ I, valid(i)

post: i’ ϵ I, the current element from i’ refers to the next element from the bag b

throws: exception if i is not valid

<u>*getCurrent(i, e)*</u>

pre: i ϵ I, valid(i)

post: e ϵ TElem, e is the current element from i

throws: exception if i is not valid

Representation:

<u>Bag:</u>

- elements: Integer [] //array of elements
- next: Integer[] //array of next positions when a collision occurs
- frequencies: Integer[] //array of frequencies
- firstEmpty: Integer //the first empty position from the array of elements
- capacity: Integer //the capacity of the arrays
- nrElems: Integer //total number of elements from the Bag
- h: TFunction //the function associated to an element

<u>Iterator:</u>

- bag : Bag // my bag
- currentIndex: Integer //the index of the current element
- nrOfAppearances: Integer // the frequency of the element from my current index

Statement of the problem:
Bob is a young entrepreneur and wants to start a construction firm. He needs an application to store the products for a better management. The application should to do these operations:

- Add a product
- Delete a product
- See the number of products he has in the storage
- See the quantity of a product
- Filter the products by their type. The type of the product is given by the first digit in its code (1 = DIY, 2 = thermic, 3 = electric, 4 = sanitary, from 5 to 9 = otherProducts)

The products will be stored using their codes.

Justification:
The ADT bag is a good choice for this application, because the order of the products does not matter. It is also easier to keep track of the frequency of every product, and also the total number of products. The bag is usually used for storing.

Implementation in pseudocode

ADT implementation in pseudocode

```
subalgorithm init(b) is:
    b.capacity <- MAX_CAP
    b.firstEmpty <- 0
    b.nrElems <- 0
    for i <- 0, b.capacity execute
        b.elements[i] <- -1
        b.frequencies[i] <- 0
        b.next[i] <- -1
    end-for
end-subalgorithm
```

Complexity : O(n)

```
subalgorithm add (b, e) is:
    position <- HFunction(e)
    if b.elements[position] = -1 then
        b.elements[position] <- e
        b.frequencies[position] <- b.frequencies[position] + 1
        if b.firstEmpty = position then
            b.updateFirstEmpty()
        end-if
    else
        currentPosition <- position
        foundElement <- false
        while b.next[currentPosition] ≠ -1 and foundElement = false execute
            if b.elements[currentPosition] = e then
                b.frequencies[currentPosition] <- b.frequencies[currentPosition] + 1
                foundElement = true
            end-if
            currentPosition <- b.next[currentPosition]
        end-while
        if b.elements[currentPosition] = e then
            b.frequencies[currentPosition] <- b.frequencies[currentPosition] + 1
        else
            b.elements[b.firstEmpty] <- e
            b.frequency[b.firstEmpty] <- 1
            b.next[currentPosition] <- b.firstEmpty
            b.updateFirstEmpty()
        end-if
    end-if
    b.nrElems <- b.nrElems + 1
end-subalgorithm
```

Complexity : O(n)

```
function remove(b, e) is:
    actualNode <- b.HFunction(e)
    previousNode <- -1
    index <- 0
    while b.elements[actualNode] ≠ e and actualNode ≠ -1 execute
        previousNode <- actualNode
        actualNode <- b.next[actualNode]
    end-while
    if actualNode = -1 then
        remove <- false
    end-if
    if b.frequencies[actualNode] ≠1 then
        b.frequencies[actualNode] <- b.frequencies[actualNode] – 1
        b.nrElems <- b.nrElems – 1
        remove <- true
    end-if
    doneMoving <- false
    do
        currentPosition <- b.next[actualNode]
        previousPos <- actualNode
        while currentPosition≠-1 and b.HFunction(b.elements[currentPosition]) ≠actualNode execute
            previousPos <- currentPosition
            currentPosition <- b.next[currentPosition]
        end-while
        if currentPosition = -1 then
            doneMoving <- true
        else
            b.elements[actualNode] <- b.elements[currentPosition]
            actualNode <- currentPosition
            previousNode <- previousPos
        end-if
    while doneMoving = false execute
    if previousNode ≠ -1 then
        b.next[previousNode] <-b.next[actualNode]
    end-if
    b.elements[actualNode] <- -1
    b.next[actualNode] <- -1

    if actualNode < firstEmpty then
        b.firstEmpty <- actualNode
    end-if
    b.nrElems <- b.nrElems – 1
    remove <- true
end-function
```

Complexity : O(n)

```
function search(b, e) is:
    currentPosition <- b.HFunction(e)
    while currentPosition ≠ -1 execute
        if b.elements[currentPosition] = e then
            search <- true
        end-if
        currentPosition <- b.next[currentPosition]
    end-while
    search <- false
end-function
```

Complexity : O(n)

```
function size(b) is:
    size <- b.nrElems
end-function
```

Complexity : Theta(1)

```
function nrOcurrences(b, i) is:
    currentPosition <- b.HFunction(i)
    while currentPosition ≠ -1 execute
        if b.elements[currentPosition] = i
            nrOcurrences <- b.frequencies[currentPosition]
        end-if
        currentPosition <- b.next[currentPosition]
    end-while
    nrOcurrences <- 0
end-function
```

Complexity : O(n)

```
subalgorithm iterator(b) is:
    iterator <- BagIterator{↑b}
end-subalgorithm
```

Complexity : Theta(1)

Iterator implementation in pseudocode

```
subalgorithm init(it, bag) is:
    it.bag <- bag
    it.first()
end-subalgorithm
```

Complexity : O(n)

```
function valid(it) is:
    valid <- it.currentIndex < it.bag.capacity
end-function
```

Complexity : Theta(1)

```
subalgorithm first(it) is:
    it.currentIndex <- 0
    while it.currentIndex < it.bag.capacity and it.bag.elements[it.currentIndex] = -1 execute
        it.currentIndex <- it.currentIndex + 1
    end-while
    it.nrOfAppearances <- 0
end-subalgorithm
```

Complexity : O(n)

```
subalgorithm next(it) is:
    if it.valid() = 0 then
        @thow exception "!INVALID ITERATOR!"
    end-if
    if it.nrOfAppearances < it.bag.frequences[currentIndex] – 1 then
        it.nrOfAppearances <- it.nrOfAppearances + 1
    else
        it.currentIndex <- it.currentIndex + 1
        while it.currentIndex < it.bag.capacity and it.bag.elements[it.currentIndex] = -1 execute
            it.currentIndex <- it.currentIndex + 1
        end-while
        it.nrOfAppearances <- 0
    end-if
end-subalgorithm
```

Complexity : O(n)

```
function getCurrent(it) is:
    if it.valid() = 0 then
        @thow exception “!INVALID ITERATOR!”
    end-if
    getCurrent <- it.bag.elements[it.currentIndex]
end-function
```

Complexity : Theta(1)

Solution implementation in pseudocode

```
subalgorithm printMenu(app) is:
    print "\n------------------------------------------------------------------------------------------------------------------\n"
    print "1. Add a product\n"
    print "2. Delete a product\n"
    print "3. See the number of products in the storage\n"
    print "4. See the quantity of a product\n"
    print "5. Filter by type\n"
    print "6. Show products\n"
    print "0. Exit\n"
    print "------------------------------------------------------------------------------------------------------------------\n\n”
end-subalgorithm
```

Complexity : Theta(1)

```
subalgorithm Add(app) is:
    print “Enter the code of the product: “
    read product
    app.bag.add(product)
end-subalgorithm
```

Complexity : O(n)

```
subalgorithm Delete(app) is:
    print “Enter the code of the product: ”
    read product
    if app.bag.remove(product) = 0 then
        @throw exception "There are no products in the storage with this code\n"
    end-if
end-subalgorithm
```

Complexity : O(n)

```
function NrOfProducts(app) is:
    nrOfProducts <- 0
    iterator <- app.bag.iterator()
    while iterator.valid() = 1 execute
        nrOfProducts <- nrOfProducts + 1
        iterator.next()
    end-while
    NrOfProducts <- nrOfProducts
```

```
end-function
```

Complexity : O(n)

```
function Quantity(app) is:
    quantity <- 0;
    ok <- 0
    iterator <- app.bag.iterator()
    print “Enter the code of the product: ”
    read product
    if app.bag.search(product) = 0 then
        @throw exception “There are no products in the storage with this code\n”
    end-if
    while iterator.valid() ≠ 0 and ok = 0 execute
        if iterator.getCurrent() = product then
            ok <- 1
        end-if
        iterator.next()
    end-while
    while iterator.valid() ≠ 0 and iterator.getCurrent() = product execute
        quantity <- quantity + 1
        iterator.next()
    end-while
    quantity <- quantity + 1
    Quantity <- quantity
end-function
```

Complexity : O(n)

```
subalgorithm ShowProducts(app) is:
    iterator = app.bag.iterator()
    print “List of products:\n”
    while iterator.valid() execute
        print iterator.getCurrent()
        print “ ”
        iterator.next()
    end-while
end-subalgorithm
```

Complexity : O(n)

```
function FirstDigit(app, product) is:
    while product > 10 execute
        product <- product / 10
    end-while
    FirstDigit <- product
end-function
```

Complexity : Theta(1)

```
subalgorithm Filter(app) is:
    types <- [“”, “DIY”, “thermic”, “sanitary”, “electric”, “otherProducts”]
    iterator <- app.bag.iterator()
    print “”Enter the type of products you want to filter after <DIY, thermic, electric, sanitary, otherProducts>: “
    read product
    iter = find(types.begin() + 1, types.end(), product)
    if iter = types.end() then
        @throw exception “!Invalid type!”
    end-if
    if product = “otherProducts” then
        while iterator.valid() execute
            firstDigit <- app.FirstDigit(iterator.getCurrent())
            if firstDigit > 4 and firstDigit < 10 then
                filteredProducts.push_back(iterator.getCurrent())
            end-if
            iterator.next()
        end-while
    else
        for i <- 1,4 execute
            if type[i] = product then
                type <- i
            end-if
        end-for
        while iterator.valid() ≠ 0 execute
            if app.FirstDigit(iterator.getCurrent()) = type then
                filteredProducts.pushback(iterator.getCurrent())
            end-if
            iterator.next()
        end-while
    end-if
    if filtered.size() = 0
        @throw exception “There are no products of this type”
    else
        print “\nFiltered products:\n”
        for i <- 0, filteredProducts.size() – 1 execute
            print filteredProducts[i]
            print “ ”
        end-for
    end-if
end-subalgorithm
```

Complexity : O(n)

```
subalgorithm AddProductsForStart(app) is:
    app.bag.add(1234);
    app.bag.add(4257);
    app.bag.add(6354);
    app.bag.add(2341);
    app.bag.add(5434);
    app.bag.add(3251);
    app.bag.add(9234);
    app.bag.add(1453);
    app.bag.add(2353);
    app.bag.add(1756);
    app.bag.add(1745);
    app.bag.add(3562);
    app.bag.add(4257);
    app.bag.add(1234);
    app.bag.add(4326);
end-subalgorithm
```

Complexity : O(n)

```
subalgorithm run(app) is:
    opt <- 1
    stop <- false
    AddProductsForStart();
    while stop = false execute
        try:
            app.printMenu()
            print "\n>>> "
            read opt
            case opt of
                1: app.Add()
                2: app.Delete()
                3:      if app.NrOfProducts() = 1 then
                            print "There is 1 product in the storage\n"
                        else
                            print "There are "
                            print app.NrOfProducts()
                            print " products in the storage\n"
                        end-if
                4:      print "The quantity of the product: "
                        print app.Quantity()
                5: app.Filter()
                6: app.ShowProducts()
                0: stop <- true
            others
                print "\n!Invalid command!\n"
        catch exception:
            print exception.what()
    end-while
end-subalgorithm
```

Complexity : O(p * n), where p is the number of options chosen by the user

Observation: "n" represents the maximum capacity. Also, the best case complexity for all the algorithms from the interface is Theta(1), except for the builder for the bag, which is always O(n).

Implementation for the tests in C++

```
void Test::testBag()
{
	Bag bag{};
	assert(bag.size() == 0);
	bag.add(1234);
	bag.add(2234);
	bag.add(3234);
	bag.add(4234);
	bag.add(5234);
	bag.add(6234);
	bag.add(7234);
	bag.add(8234);
	bag.add(9234);
	bag.add(9234);
	bag.add(9234);
	assert(bag.size() == 11);
	for (int i = 1; i <= 9; i++)
	{
		int number = i * 1000 + 234;
		assert(bag.search(number) == true);
	}
	assert(bag.nrOccurrences(9234) == 3);
	assert(bag.nrOccurrences(1234) == 1);
	assert(bag.nrOccurrences(7324) == 0);
	bag.remove(9234);
	bag.remove(9234);
	bag.remove(1234);
	assert(bag.size() == 8);
	for (int i = 2; i <= 9; i++)
	{
		int product = i * 1000 + 234;
		assert(bag.search(product) == true);
	}
	assert(bag.search(1234) == false);
	assert(bag.remove(1234) == false);
	bag.add(7324);
	bag.add(8932);
	bag.add(1400);
	assert(bag.size() == 11);
	assert(bag.search(7324) == true);
	assert(bag.search(8932) == true);
	assert(bag.search(1400) == true);
	bag.remove(2234);
	bag.remove(3234);
	bag.remove(4234);
	bag.remove(5234);
	bag.remove(6234);
	bag.remove(7234);
	bag.remove(8234);
	bag.remove(9234);
	bag.remove(7324);
	bag.remove(8932);
	bag.remove(1400);
	assert(bag.size() == 0);
}

void Test::testBagIterator()
{
	Bag bag{};
	bag.add(3647);
	BagIterator iteratorBuid = bag.iterator();
	bag.add(4836);
	bag.add(3636);
	bag.add(4836);
	bag.add(4836);
	bag.add(2736);
	bag.add(3849);
	bag.add(2635);
	bag.add(6543);
	bag.add(9634);
	bag.add(9634);
	BagIterator iterator = bag.iterator();
//valid and next
	while (iterator.valid())
	{
		int product = iterator.getCurrent();
		assert(bag.search(product) == true);
		iterator.next();
	}
//exception for getCurrent
	try
	{
		iterator.getCurrent();
	}catch (std::exception & exc) { assert(strcmp(exc.what(), "!INVALID ITERATOR!") == 0); }
//exception for next
	try
	{
		iterator.next();
	}catch (std::exception & exc) { assert(strcmp(exc.what(), "!INVALID ITERATOR!") == 0); }
//first
	iterator.first();
	assert(iterator.valid() == true);
}

void Test::testAll()
{
	testBag();
	testBagIterator();
}
```