**Data Structures And Algorithms Project**

**Project nr.11**

**Marcu Alexandru Daniel**

**Requirement**:

**ADT SortedMultiMap – implementation on a binary search tree**

**ADT – domain and interface**

**Domain:**

**SMM** = {smm | smm is a Sorted Multimap with pairs TKey, TValue, where we can define a relation R on the set of all possible keys}

**Interface:**

**• init (smm, R)**

pre: R – relation on the set of all possible keys

post: smm∈SMM ,smm = φ

**• destroy(smm)**

pre: smm∈ SMM

post:smm was destroyed (allocated memory was freed)

**• add(smm, k, v)**

pre: smm∈ SMM , k ∈TKey,v∈TValue

post: the pair was added into smm

**• remove(smm, k, v)**

pre: smm∈ SMM , k ∈TKey,v∈TValue

post:the pair was deleted from smm (if it was in smm)

**• search(smm, k, l)**

pre: smm∈SMM , k ∈TKey

⎧ **true** and l is the list of values associated with k, if k is in smm

post: l∈L search→⎨

⎩ **false** and l=φ otherwise

**• iterator(smm, it)**

pre: smm∈ SMM

post: it ∈ I , it is an iterator over smm

**• keySet(smm, m)**

pre: smm∈ SMM

post: m∈ M , m is the set of all keys from smm

**• size(smm)**

pre: smm∈ SMM

post: size<-s, s ∈ INTEGER , s is the number of nodes in the smm

**Iterator – domain and interface**

**Domain:**

**I** = {i | i is an iterator over smm ϵ SMM }

**Interface:**

**• init(i, smm)**

pre: smm ϵ SMM

post: i ϵ I, i is an iterator over smm

**• valid(i)**

pre: i ϵ I

⎧**true**, if the current element from i is a valid one

post: valid →⎨

⎩**false,** otherwise

**• next(i)**

pre: i ϵ I, valid(i)

post: i’ ϵ I , the current element from i’ refers to the next element from the sorted multi map smm.

**• getCurrent(i, e)**

pre: i ϵ I, valid(i)

post: e ϵ TComp, e is the current key-value pair from i

**ADT – representation**

**TComp: BSTNode: SMM:**

k:TKeyComp info:TComp root:↑BSTNode

v:TValue left: ↑BSTNode R: relation

right: ↑BSTNode

⎧**true,** if “k1<=k2”

**R(k1,k2)** →⎨ , where k1 and k2 are TKeyComp.

⎩**false,** otherwise

**Iterator – representation**

**InorderIteratorSMM:**

current: ↑BSTNode

smm:SMM

s:Stack

**Pseudocode for ADT operations**

**subalgorithm** init(smm, R) **is**:

smm.root <- NIL

smm.R <- R

**end\_subalg**

Complexity: Θ(1)

**subalgorithm** destroy(smm) **is**:

delete\_smm(smm, smm.root)

**end\_subalg**

Complexity: Θ(delete\_smm) = Θ(n), n – nr of nodes

**subalgorithm** delete\_smm(smm, root) **is**:

//pre: root is a ↑BSTNode, smm is SMM

//post: root is deallocated at each step

**if** root ≠ NIL **then**

**if** [root].left ≠ NIL **then**

**delete\_smm**(smm, [root].left)

**end\_if**

**if** [root].right ≠ NIL **then**

**delete\_smm**(smm, [root].right)

**end\_if**

**@**delete root

**end\_if**

**end\_subalg**

Complexity: Θ(n), n-nr of nodes

**function** add(smm, k, v) **is**:

smm.root <- insertNode(smm, smm.root, k, v)

**end\_function**

Complexity: O(insertNode)= O(n), n – number of nodes

**function** insertNode(smm, node, k, v) **is**:

//pre:smm is SMM, node is ↑BSTNode, k is TKeyComp, v is TValue

//post: a node containing the key and value given will be added in the smm

**if** node = NIL **then**

el <- @a dynamically allocated TComp type with key **k** and value **v**

**insertNode** <- @a dynamically alocated BSTNode with no children, but with **el** as info

**else**

**if** smm.R(k, [node].info.k) **then**

[node].left <- **insertNode**(smm, [node].left, k, v)

**else**

[node].right <- **insertNode**(smm, [node].right, k, v)

**end\_if**

**end\_if**

**insertNode** <- node

**end\_function**

Complexity: O(n), n – number of nodes

**function** remove(smm, k, v) **is**:

smm.root <- deleteNode(smm, smm.root, k, v)

**end\_function**

Complexity: O(deleteNode)= O(n), n – number of nodes

**function** deleteNode(smm, root, k, v) is:

//pre:smm is SMM, node is ↑BSTNode, k is TKeyComp, v is TValue

//post: node will be removed from smm

**if** root = NIL **then**

**deleteNode** <- root

**end\_if**

**if** v ≠ [root].info.v **then**

**if** smm.R(k, [root].info.k) **then**

[root].left <- **deleteNode**(smm, [root].left, k, v)

**else**

**if** **not** smm.R(k, [root].info.k) **then**

[root].right <- **deleteNode**(smm, [root].right, k, v)

**end\_if**

**end\_if**

**else**

**if** [root].left = NIL **then**

temp <- [root].right

**@** delete root

**deleteNode** <- temp

**else**

**if** [root].right = NIL **then**

temp = [root].left

**@** delete root

**deleteNode** <- temp

**end\_if**

**end\_if**

temp <- **minValueNode**(smm, [root].right)

[root].info <- [temp].info

new\_v <- [temp].info.v

[root].right <- **deleteNode**(smm, [root].right, [temp].info.k, new\_v)

**end\_if**

**deleteNode** <- root

**end\_function**

Complexity: O(n), n – number of nodes

**function** minValueNode(smm, node) **is**:

//pre: smm is SMM, node is ↑BSTNode

//post: minValueNodE returns the node of smm with smallest value

current <- node

**while** [current].left ≠ NIL **execute**

current <- [current].left

**end\_while**

**minValueNode** <- current

**end\_function**

Complexity: O(n), n – number of nodes

**function** search(smm, k, l) **is**:

**if** smm.root = NIL **then**

**search** <- **false**

current\_node <- smm.root

**while** current\_node ≠ NIL **execute**

**if** smm.R(k, [current\_node].info.k) **then**

**if** k = [current\_node].info.k **then**

add(l, [current\_node].info.v)

**end\_if**

current\_node <- [current\_node].left

**else**

current\_node <- [current\_node].right

**end\_if**

**end\_while**

**if** size(l) = 0 **then**

**search** <- **false**

**search** <- **true**

**end\_function**

Complexity: O(n), n – number of nodes

**subalgorithm** iterator(smm, it) **is:**

init(it, smm)

**end\_subalg**

Complexity: **(the complexity of init(it,smm) operation)** O(n), extra space complexity O(n) , n – number of nodes

**subalgorithm** keySet(smm, m) **is:**

inorderKey(smm, smm.root, m)

**end\_subalg**

Complexity: Θ(inorderKey) = Θ(n\*p), n – number of nodes in smm, p – size of list m

**subalgorithm** inorderKey(smm, root, m) **is**:

**if** root ≠ NIL **then**

**inorderKey**(smm, [root].left, m)

ok <- 1

**for** i <- 0, size(m) **do**

**if** [root].info.k = m[i] **then**

ok <- 0

**end\_if**

**end\_for**

**if** ok = 1 **then**

add(m, [root].info.k)

**end\_if**

**inorderKey**(smm, [root].right, m)

**end\_if**

**end\_subalgorithm**

Complexity: Θ(n\*p), n – number of nodes in smm, p – size of list m

**function** size(smm) **is:**

size <- sizeRecursive(smm, smm.root)

**end\_function**

Complexity: Θ(sizeRecursive) = Θ(n), n – number of nodes in smm

**function** sizeRecursive(smm, node) **is:**

**if** node = NIL **then**

**sizeRecursive** <- 0

**else**

**sizeRecursive** <- **sizeRecursive**(smm, [node].left) + **sizeRecursive**(smm, [node].right) + 1

**end\_if**

**end\_function**

Complexity: Θ(n), n – number of nodes in smm

**Pseudocode for Iterator operations**

**subalg** init(i, smm) **is:**

i.smm <- smm

init(it.s)

node <- smm.root

**while** node ≠ NIL **execute**

push(s, node)

node <- [node].left

**end\_while**

**if** not isEmpty(it.s) **then**

it.current <- top(it.s)

**else**

it.current <- NIL

**end\_if**

**end\_subalg**

Complexity: O(n), extra space complexity O(n) , n – number of nodes

**subalg** getCurrent(i, e) **is:**

e <- [i.current].info

**end\_subalg**

Complexity: Θ(1)

**function** valid(i) **is**:

**if** i.current = NIL **then**

**valid** <- **false**

**else**

**valid** <- **true**

**end\_if**

**end\_function**

Complexity: Θ(1)

**subalgorithm** next(i) **is:**

node <- pop(i.s)

**if** [node].right ≠ NIL **then**

node <- [node].right

**while** node ≠ NIL **execute**

push(i.s, node)

node <- [node].left

**end\_while**

**end\_if**

**if** **not** isEmpty(i.s) **then**

i.current <- top(i.s)

**else**

i.current <- NIL

**end\_if**

**end\_subalg**

Complexity: O(n), extra space complexity O(n) , n – number of nodes

**Complexity computations for operation size(smm)**

Operation size(smm) passes trhough all the nodes of the tree in order to count them, so we can say it`s complexity is like the complexity of a tree traversal.

Notice that **Best case, Average case and Worst case are the same, because we need to always go through all the nodes of the smm.**

Complexity function T(n) — for all problem where tree traversal is involved — can be defined as:

T(n) = T(k) + T(n – k – 1) + c

Where k is the number of nodes on one side of root and n-k-1 on the other side.

Now the computations depends on the form of the tree, but let’s do analysis of the worst case scenario, where the tree is a Skewed tree:

Skewed tree (One of the subtrees is empty and other subtree is non empty )

k is 0 in this case.  
T(n) = T(0) + T(n-1) + c  
T(n-1) = 2T(0) + T(n-2) + 2\*c  
T(n-2) = 3T(0) + T(n-3) + 3\*c  
T(n-3) = 4T(0) + T(n-4) + 4\*c

…………………………………………  
………………………………………….  
T(2) = (n-1)\*T(0) + T(1) + (n-1)\*c  
T(1) = n\*T(0) + n\*c

Value of T(0) will be some constant say d. (traversing an empty tree will take some constants time)

If we sum everything => n\*T(n) = [n\*(n+1)/2] \* (c+d)

So T(n) = [(n+1)/2] \* (c+d) =>

T(n) = n\*(c+d), and as c and d are constants =>  
=> **T(n) = Θ(n)**

**Tests for ADT and ITERATOR**

void test\_creatorSMM()

{

//testing default constructor

SMM smm = SMM();

assert(smm.getRoot() == NULL);

//testing parameter constructor

Dog d1 = Dog("Balldog", "Jack", "Marcu Daniel", 4);

TComp el2 = TComp(d1.getBreed(), d1);

BSTNode\* n2 = new BSTNode(el2, NULL, NULL);

Dog d3 = Dog("Pug", "Doug", "Nisy", 3);

TComp el3 = TComp(d3.getBreed(), d3);

BSTNode\* n3 = new BSTNode(el3, NULL, NULL);

Dog d4 = Dog("Beagle", "John", "Miruna", 2);

TComp el4 = TComp(d4.getBreed(), d4);

BSTNode\* n4 = new BSTNode(el4, n2, n3);

SMM smm2 = SMM(n4);

assert(smm2.getRoot() == n4);

assert(smm2.getRoot()->getInfo().getDog().getAge() == 2);

assert(smm2.getRoot()->getInfo().getDog().getName() == "John");

assert(smm2.getRoot()->getInfo().getDog().getBreed() == "Beagle");

assert(smm2.getRoot()->getInfo().getDog().getOwnerName() == "Miruna");

assert(smm2.getRoot()->getInfo().getKey() == "Beagle");

assert(smm2.getRoot()->getLeft() == n2);

assert(smm2.getRoot()->getRight() == n3);

assert(smm2.getRoot()->getLeft()->getInfo().getDog().getName() == "Jack");

assert(smm2.getRoot()->getRight()->getInfo().getDog().getName() == "Doug");

}

void test\_relation()

{

Dog d1 = Dog("Balldog", "Jack", "Marcu Daniel", 4);

TComp el2 = TComp(d1.getBreed(), d1);

BSTNode\* n2 = new BSTNode(el2, NULL, NULL);

Dog d3 = Dog("Pug", "Doug", "Nisy", 3);

TComp el3 = TComp(d3.getBreed(), d3);

BSTNode\* n3 = new BSTNode(el3, NULL, NULL);

Dog d4 = Dog("Beagle", "John", "Miruna", 2);

TComp el4 = TComp(d4.getBreed(), d4);

BSTNode\* n4 = new BSTNode(el4, n2, n3);

SMM smm2 = SMM(n4);

assert(smm2.R(n4->getInfo().getKey(), n3->getInfo().getKey()) == true);

assert(smm2.R(n3->getInfo().getKey(), n4->getInfo().getKey()) == false);

assert(smm2.R(smm2.getRoot()->getInfo().getKey(), n2->getInfo().getKey()) == false);

}

SMM\* give\_smm()

{

Dog d6 = Dog("Bichon", "John", "Liza", 3);

TComp e6 = TComp(d6.getBreed(), d6);

BSTNode\* n6 = new BSTNode(e6, NULL, NULL);

Dog d5 = Dog("Zabuza", "Johnuletz", "Dima", 3);

TComp e5 = TComp(d5.getBreed(), d5);

BSTNode\* n5 = new BSTNode(e5, NULL, NULL);

Dog d4 = Dog("Bichon", "Biju", "Dima", 2);

TComp e4 = TComp(d4.getBreed(), d4);

BSTNode\* n4 = new BSTNode(e4, n6, NULL);

Dog d3 = Dog("Zabuza", "Shackle", "Zoro", 3);

TComp e3 = TComp(d3.getBreed(), d3);

BSTNode\* n3 = new BSTNode(e3, n5, NULL);

Dog d2 = Dog("Beagle", "Balaur", "Sinaia", 6);

TComp e2 = TComp(d2.getBreed(), d2);

BSTNode\* n2 = new BSTNode(e2, NULL, n4);

Dog d1 = Dog("Pug", "Doug", "Marcu Daniel", 3);

TComp e1 = TComp(d1.getBreed(), d1);

BSTNode\* n1 = new BSTNode(e1, n2, n3);

SMM\* smm = new SMM(n1);

return smm;

}

void test\_search()

{

SMM\* smm = give\_smm();

std::vector<Dog> l;

assert(l.size() == 0);

assert(smm->search("Zabuza", l) == true);

assert(l.size() == 2);

delete smm;

}

void test\_reciterator()

{

SMM\* smm = give\_smm();

InorderIteratorSMM it = smm->iterator();

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

assert(it.getCurrent().getKey() == "Beagle");

it.next();

assert(it.getCurrent().getKey() == "Bichon");

delete smm;

}

void test\_add()

{

SMM\* smm = give\_smm();

Dog d = Dog("African", "Johana", "Eu", 44);

smm->add("African", d);

InorderIteratorSMM it = smm->iterator();

assert(it.getCurrent().getKey() == "African");

Dog d2 = Dog("Azumba", "Salami", "Eu", 22);

smm->add("Azumba", d2);

InorderIteratorSMM it2 = smm->iterator();

assert(it2.getCurrent().getKey() == "African");

it2.next();

assert(it2.getCurrent().getKey() == "Azumba");

delete smm;

}

void test\_remove()

{

SMM\* smm = give\_smm();

Dog d("Beagle", "Balaur", "Sinaia", 6);

smm->remove(d.getBreed(), d);

InorderIteratorSMM it = smm->iterator();

assert(it.getCurrent().getKey() == "Bichon");

it.next();

assert(it.getCurrent().getKey() == "Bichon");

it.next();

assert(it.getCurrent().getKey() == "Pug");

Dog d1 = Dog("Pug", "Doug", "Marcu Daniel", 3);

smm->remove(d1.getBreed(), d1);

Dog d3 = Dog("Zabuza", "Shackle", "Zoro", 3);

smm->remove(d3.getBreed(), d3);

InorderIteratorSMM it2 = smm->iterator();

assert(it2.getCurrent().getKey() == "Bichon");

it2.next();

assert(it2.getCurrent().getKey() == "Bichon");

it2.next();

assert(it2.getCurrent().getKey() == "Zabuza");

delete smm;

}

void test\_getKeys()

{

SMM\* smm = give\_smm();

std::vector<std::string> m;

assert(m.size() == 0);

smm->keySet(m);

assert(m.size() == 4);

assert(m[0] == "Beagle");

assert(m[1] == "Bichon");

assert(m[2] == "Pug");

assert(m[3] == "Zabuza");

delete smm;

}

void test\_size()

{

SMM\* smm = give\_smm();

assert(smm->size() == 6);

Dog d = Dog("Zabuza", "Shackle", "Zoro", 4);

smm->remove("Zabuza", d);

assert(smm->size() == 5);

delete smm;

}

void test\_SMM()

{

test\_creatorSMM();

test\_relation();

test\_search();

test\_reciterator();

test\_add();

test\_remove();

test\_getKeys();

test\_size();

}

////////////////////////////////////////////////

void test\_constriterator()

{

Dog d4 = Dog("Beagle", "John", "Miruna", 2);

TComp el4 = TComp(d4.getBreed(), d4);

BSTNode\* n4 = new BSTNode(el4, NULL, NULL);

SMM smm = SMM(n4);

InorderIteratorSMM it = InorderIteratorSMM(smm);

assert(&it.getSmm() == &smm);

assert(it.getCurrent().getKey() == n4->getInfo().getKey());

assert(it.getCurrent().getDog().getOwnerName() == n4->getInfo().getDog().getOwnerName());

/////////////// now for a smm with more than 1 node;

Dog d44 = Dog("Beagle", "John", "Miruna", 2);

TComp el44 = TComp(d44.getBreed(), d44);

BSTNode\* n44 = new BSTNode(el44, NULL, NULL);

Dog d1 = Dog("Zabuza", "Jack", "Marcu Daniel", 4);

TComp el2 = TComp(d1.getBreed(), d1);

BSTNode\* n2 = new BSTNode(el2, NULL, NULL);

Dog d3 = Dog("Pug", "Doug", "Nisy", 3);

TComp el3 = TComp(d3.getBreed(), d3);

BSTNode\* n3 = new BSTNode(el3, n44, n2);

SMM smm2 = SMM(n3);

InorderIteratorSMM it2 = InorderIteratorSMM(smm2);

assert(&it2.getSmm() == &smm2);

assert(it2.getCurrent().getKey() == n44->getInfo().getKey());

assert(it2.getCurrent().getDog().getOwnerName() == n44->getInfo().getDog().getOwnerName());

}

void test\_valid\_next()

{

Dog d4 = Dog("Beagle", "John", "Miruna", 2);

TComp el4 = TComp(d4.getBreed(), d4);

BSTNode\* n4 = new BSTNode(el4, NULL, NULL);

SMM smm = SMM(n4);

InorderIteratorSMM it = InorderIteratorSMM(smm);

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

it.next();

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

Dog d44 = Dog("Beagle", "John", "Miruna", 2);

TComp el44 = TComp(d44.getBreed(), d44);

BSTNode\* n44 = new BSTNode(el44, NULL, NULL);

Dog d1 = Dog("Zabuza", "Jack", "Marcu Daniel", 4);

TComp el2 = TComp(d1.getBreed(), d1);

BSTNode\* n2 = new BSTNode(el2, NULL, NULL);

Dog d3 = Dog("Pug", "Doug", "Nisy", 3);

TComp el3 = TComp(d3.getBreed(), d3);

BSTNode\* n3 = new BSTNode(el3, n44, n2);

SMM smm2 = SMM(n3);

InorderIteratorSMM it2 = InorderIteratorSMM(smm2);

assert(it2.valid() == true);

assert(it2.getCurrent().getKey() == "Beagle");

it2.next();

assert(it2.valid() == true);

assert(it2.getCurrent().getKey() == "Pug");

it2.next();

assert(it2.valid() == true);

assert(it2.getCurrent().getKey() == "Zabuza");

it2.next();

assert(it2.valid() == false);

}

void test\_iterator()

{

test\_constriterator();

test\_valid\_next();

}

**Problem statement**

Design an application for the organizers of a Dog Beauty Contest, an application that should help them manage their contestants. For each contestant (dog) we have to retain the Breed of the dog, the Name of the dog, the Name of the Owner and the Dog Age. The application should provide the user the following options:

* Add a contestant;
* Remove a contestant;
* Print all the contestants, organized alphabetically by dog breed;
* Show the most popular breed of dog and the number of participants from that breed at the moment; (if more such breeds exist, print the first one in alphabetic order)
* Show the least popular breed of dog and the number of participants from that breed at the moment. (if more such breeds exist, print the first one in alphabetic order)

**Justification:** Provided the fact that we need to print which is the most popular and the least popular breed, together with the numbers of contestants from those breeds, it is much better to use a Sorted Multi Map than a normal vector in order to store our contestants. Using a Sorted Multi Map would allow us to instantly have access strictly to the contestants from a given breed (each breed representing a key of the SMM), but also, we could easily count the exemplars of each breed only using a single iteration through the list of values of each key, and then counting the values of each key (values obtained using operation search for each key). We would also only need 2 variables to find out the most popular/least popular breed: currentPopular, currentPopularNr. The Multi Map must be a Sorted one, so that we can easily satisfy the third requirement, and that is to print the contestants, organized alphabetically by dog breed (using the iterator).

**Pseudocode for Problem Solution**

**function** main() **is**:

**@**call test functions, which will throw exceptions if tests are wrong

init(smm)

init(ui, smm)

run(ui)

**@**check for memory leaks

**main** <- 0

**end\_function**

Complexity: Θ(1)

**subalgorithm** run(ui) **is**:

print\_menu(ui)

choice <- 0

**@** print message “Give your choice”

**@** read choice

**if** valid\_choice(ui, choice) = 0 **then**

**@** print message “Invalid Choice!”

**end\_if**

**if** choice = 0 **then**

**@ stop subalgorithm**

**end\_if**

**if** choice = 1 **then**

add\_contestant(ui)

**end\_if**

**if** choice = 2 **then**

remove\_contestant(ui)

**end\_if**

**if** choice = 3 **then**

print\_cont\_alphabetically(ui)

**end\_if**

**if** choice = 4 **then**

show\_popular(ui)

**end\_if**

**if** choice = 5 **then**

show\_notpopular(ui)

**end\_if**

**end\_subalg**

**subalgorithm** start\_app(ui) **is**:

**@**add Dog with breed:Beagle, name:Bernard, owner name:Marcu Daniel, age:3

**@**add Dog with breed:Pug, name:Doug, owner name:Rooney McDonald, age:5

**@**add Dog with breed:Amstaff, name:Spike, owner name:Donald Trump, age:3

**@**add Dog with breed:Bulldog, name:Destroyer, owner name:Prisoner John, age:4

**@**add Dog with breed:Bulldog, name:Tanker, owner name:Prisoner John, age:5

**@**add Dog with breed:Chihuahua, name:Spetzle, owner name:Anisia Hurduzeu, age:2

**@**add Dog with breed:Pug, name:Slammy, owner name:Marcu Daniel, age:2

**end\_subalg**

Complexity: Θ(1)

**subalgorithm** print\_menu(ui) **is:**

**@** print empty line

**@** print the whole menu of functionalities from which the user can choose

**@**prin empty line

**end\_subalg**

Complexity: Θ(1)

**function** valid\_choice(ui, choice) **is:**

**if** choice ≤ 5 **or** choice ≥ 0 **then**

**valid\_choice** <- 1

**end\_if**

**valid\_choice** <- 0

**end\_function**

Complexity: Θ(1)

**subalgorithm** add\_contestant(ui) **is:**

**@** print message “Give breed of dog”

**@** read breed

**@** print message “Give name of dog”

**@** read name

**@** print message “Give name of owner”

**@** read owner\_name

**@** print message “Give age of dog”

**@** read age

**@** create a new dog with the data read above

add(ui.smm, new\_dog.breed, new\_dog)

**end\_subalgorithm**

Complexity: Θ(1)

**subalgorithm** remove\_contestant(ui) **is:**

**@** print message “Give breed of dog”

**@** read breed

**@** print message “Give name of dog”

**@** read name

**@** print message “Give name of owner”

**@** read owner\_name

**@** create a dog with the data read above

remove(ui.smm, dog.breed, dog)

**end\_subalgorithm**

Complexity: Θ(1)

**subalgorithm** print\_cont\_alphabetically(ui) **is:**

iterator(ui.smm, it)

**while** valid(it) **execute**

**@** print the current dog with the use of function **getCurrent**(it)

next(it)

**end\_while**

**end\_subalgorithm**

Complexity: Θ(n), n – number of nodes in the smm

**subalgorithm** show\_popular(ui) **is:**

currentPopular <- “”

currentPopularNr <- 0

**@**initialize **keylist** , an empty std::vector

keySet(ui.smm, keylist)

**@**initialize **dogs**, an empty std::vector

**for** k **in** keylist **do**

clear(dogs)

search(ui.smm, k, dogs)

**if** size(dogs) > currentPopularNr **then**

currentPopularNr <- size(dogs)

currentPopular <- k

**end\_if**

**end\_for**

**@**print the most popular dog breed together with the number of exemplars of that breed

**end\_subalgorithm**

Complexity: Θ(k), k – the number of keys in the smm

**subalgorithm** show\_notpopular(ui) **is:**

currentNotPopular <- “”

currentNotPopularNr <- ∞

**@**initialize **keylist** , an empty std::vector

keySet(ui.smm, keylist)

**@**initialize **dogs**, an empty std::vector

**for** k **in** keylist **do**

clear(dogs)

search(ui.smm, k, dogs)

**if** size(dogs) < currentNotPopularNr **then**

currentNotPopularNr <- size(dogs)

currentNotPopular <- k

**end\_if**

**end\_for**

**@**print the least popular dog breed together with the number of exemplars of that breed

**end\_subalgorithm**

Complexity: Θ(k), k – the number of keys in the smm