**ADT SortedBag – implementation on a binary search tree.**

**Domain: SB** = {sb | sb is a SortedBag with elements of the type TComp, where we can define a relation R on the set of all possible elements}

**Interface (set of operations):**

init(sb, R)

description: create an empty bag

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

post: sb ϵ **SB**, sb is an empty Bag

add(sb,e)

description: add an element to the bag

pre: sb ϵ **SB**, e ϵ TComp

post: sb’ ϵ **SB**, sb’ = sb U {e} (TComp e is added to the bag)

remove(sb, e)

description: remove an appearance of the element from the bag if this it is in bag

pre: sb ϵ **SB**, e ϵ TComp

post: sb’ ϵ **SB**, sb’ = sb \ {e} (one ocurrence of e was removed from the Bag). If e is not in sb, sb is not changed.

search(sb, e)

description: search an element in the bag

pre: sb ϵ **SB**, e ϵ TComp

post: true , if e ϵ **SB**, false otherwise

size(sb)

description: return the size of bag (how many elements are in bag)

pre: sb ϵ **SB**

post : size 🡨 the number of elements from **SB**

destroy(sb)

description: destroy the bag

pre: sb ϵ **SB**

post: sb was destroyed

iterator(b, i)

description: returns an iterator over the bag

pre: sb ϵ **SB**

post: i ϵ I, i is an iterator over sb

**ADT Iterator**

- Has access to the interior structure of the Bag and it has a current element from the Bag.

**Domain**: **I** = {i | i is an iterator over sb ϵ **SB** }

**Interface (set of operations):**

init(i, sb)

description: creates a new iterator over the elements of sb

pre: sb ϵ **SB**

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

valid(i)

description: verifies if the iterator is valid

pre: i ϵ **I**

post: valid 🡨 true, if the current element from I is

valid 🡨 false, otherwise

next(i)

description: takes the iterator to the next element

pre: i ϵ **I**, valid(i)

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

getCurrent(i, e)

description: initializate e with the current element from the iterator

pre: i ϵ **I**, valid(i)

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

**Representation:**

Node:

info: TComp

left: ↑Node

right: ↑Node

SB:

root: ↑Node

R: Relation

R(k1,k2) = true, if *k1->info* **<** *k2->info* ( k1 comes before k2)

false, otherwise

Iterator:

SB: Sorted Bag

Current: ↑Node

Queue: Q @ where we keep all the nodes in inOrder way

**Implementation SortedBag:**

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

@ sb will be initialized with an empty SortedBag with relation R

sb.root 🡨 NIL

**end-subalgorithm**

**function** add(sb, e) **is**:

add 🡨 add\_rec(sb.root, e)

**end-function**

**function** add\_rec(node, e) **is**:

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

node 🡨 @ a new node is created with [node].left = [node].right = NIL

[node].info = e

**else if** e < [node].info **then**

[node].left 🡨 add\_rec([node].left, e)

**else**

[node].right 🡨 add\_rec([node].right, e)

**end-if**

add\_rec 🡨 node

**end-function**

**function** remove(sb, e) **is**:

remove 🡨 remove\_rec(sb.root, e)

**end-function**

**function** remove\_rec(node, e) **is**:

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

remove\_rec🡨 NIL

**else if** e < [node].info **then**

[node].left 🡨 remove\_rec([node].left, e)

**else if** e > [node].info **then**

[node].right 🡨 remove\_rec([node].right,e)

**else**

**if** [node].right = NIL and [node].left = NIL **then**

@ node is deleted

node = NIL

**else if** [node].right = NIL **then**

aux 🡨 node

node = [node].left

@ aux is deleted

**else if** [node].left = NIL **then**

aux 🡨 node

node = [node].right

@ aux is deleted

**else**

aux 🡨 @ minimum node value from tree with root [node].right

[node].info 🡨 [aux].info

[node].right 🡨 remove\_rec([node].right, [aux].info)

**end-if**

remove\_rec 🡨 node

**end-function**

**function** search(sb, e) **is**:

search 🡨 search\_rec(sb.root, e)

**end-function**

**function** search\_rec(node, e) **is**:

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

search\_rec 🡨 false

**else** if e == [node].info **then**

searc\_rec 🡨 true

**else if** e < [node].info **then**

search\_rec 🡨 search\_rec([node].left, e)

**else**

search\_rec 🡨 search\_rec([node].right, e)

**end-if**

**end-function**

**function** size (sb, e) **is**:

size 🡨 size\_rec(sb.root, e)

**end-function**

**function** size\_rec(node, e) **is**:

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

size\_rec 🡨 0

**else**

size\_rec 🡨 size\_rec([node].left, e) + size\_rec([node].right, e) + 1

**end-if**

**end-function**

**function** iterator (sb) **is**:

iterator 🡨 @ return an iterator over SortedBag sb

**end-function**

**subalgorithm** destroy(sb) **is**:

it 🡨 sb.iterator(sb)

**while** valid(it) **do**:

getCurrent(it, e) @ e is a element what will be initialized in function getCurrent

@ delete e

next(it)

**end-subalgorithm**

**Implementation Iterator:**

**subalgorithm** init(it, sb) **is**:

@ it will be initialized with an iterator over SortedBag sb and Q is filled up with inOrder @ traversal nodes

it.curretNode 🡨 sb.root()

it.Q 🡨 inOrder(it.currentNode)

**end-subalgorithm**

**subalgorithm** inOrder (node, it) **is**:

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

inOrder([node].left, it)

it.Q.push(node)

inOrder([node].right, it)

**end-if**

**end- subalgorithm**

**function** valid (it) **is**:

valid 🡨 it.currentNode != NIL

**end-function**

**function** next(it) **is**:

**if** it.Q.empty() **then**

next 🡨 NIL

**else**

next 🡨 it.Q.pop()

**end-function**

**subalgorithm** getCurrent(it, e) **is**:

e 🡨 it.currentNode

**end-subalgorithm**

**Test Container:**

void Testing::testSortedBag()

{

Book b{ "aaa", "bbb" };

Book a{ "bbb", "bbb" };

Book c{ "ccc", "bbb" };

SortedBag sb;

sb.setRoot(sb.add(sb.getRoot(), b));

assert(sb.size(sb.getRoot()) == 1);

sb.setRoot(sb.remove(sb.getRoot(), b));

assert(sb.size(sb.getRoot()) == 0);

assert(sb.search(sb.getRoot(), b) == false);

sb.setRoot(sb.add(sb.getRoot(), b));

assert(sb.search(sb.getRoot(), b) == true);

assert(sb.getMin(sb.getRoot()) == sb.getRoot());

sb.setRoot(sb.getRoot());

sb.setRoot(sb.add(sb.getRoot(), b));

sb.setRoot(sb.add(sb.getRoot(), c));

sb.setRoot(sb.add(sb.getRoot(), a));

sb.setRoot(sb.add(sb.getRoot(), b));

sb.setRoot(sb.add(sb.getRoot(), c));

sb.setRoot(sb.add(sb.getRoot(), a));

Book d{ "DDD", "bbb" };

Node\* miny = sb.getMin(sb.getRoot());

assert(sb.getMin(sb.getRoot()) == miny);

sb.setRoot(sb.add(sb.getRoot(), d));

miny = sb.getMin(sb.getRoot());

assert(sb.getMin(sb.getRoot()) == miny);

assert(sb.search(sb.getRoot(), a) == true);

assert(sb.search(sb.getRoot(), b) == true);

assert(sb.search(sb.getRoot(), c) == true);

assert(sb.search(sb.getRoot(), d) == true);}

**Complexity:**

init(sb, R) - ***Ө(1)***

add(sb,e) - ***O(n)***

remove(sb,e) - ***O(n)***

search(sb,e) - ***O (n)***

size(sb,e) - ***Ө (n)***

destroy(sb) - ***Ө (n)***

iterator(sb,it) - ***Ө(1)***

Computation: complexity of size(sb,e) is **theta n**, where n is the number of elements because we have to traverse all the elements to get the whole size of the ADT.

**Problem Statement:**

A library needs a software to keep the track of the available books. In the library can be some book multiple times. The software has to be able to add, remove, search a book. A book has a following form: Author - Name

**Why is the Sorted Bag suitable for solving this problem?**

The bag(multiset) is a date structure that allows adding multiple elements of the same type, in our case we can have more identical books. It’s also important that this books to be sorted by author and if they have the same author, sorted by name of the book, because we can find one book more easily.

**Functions from ADT for problem solution:**

The functions that I use in order to solver my problem are : add, remove, search.

Book: Node: UI:

Author: string Info: Book sb: SortedBag

Name: string left: ↑Node

right: ↑Node

SortedBag:

root: ↑Node

R: Relation

R(book1,book2) = true, if book1.info.getAuthor() < book2.info.getAuthor()

false otherwise

**UI interface and implementation:**

i**nit**(ui, sb)

ui.sb🡨sb

readBook(ui)

@return a book with author and name inputed from keyboard

run(ui)

**while** true **do**

cmd 🡨 @input from keyboard

**if cmd** = 1 then

sb.add(sb.root(),readBook())

**else if** cmd = 2 then

sb.remove(sb.root(),readBook())

**else if** cmd = 3 then

sb.search(sb.root(),readBook())

**else**

break

**end-if**

**done**

void Testing::testBook()

{

Book x{ "AAA", "BBB" };

Book q{ "AAA", "AAA" };

assert(x.getAuthor() == "AAA");

assert(x.getName() == "BBB");

x.setAuthor("aaa");

x.setName("bbb");

assert(x.getAuthor() != "AAA");

assert(x.getName() != "BBB");

assert(x.getAuthor() == "aaa");

assert(x.getName() == "bbb");

Book y{ "CCC", "DDD" };

Book k{ "CCC", "AAA" };

Book j{ "DDD", "AAA" };

assert(x > y);

assert(y < x);

assert(y > k);

assert(!(y < k));

assert(x > q);

assert(!(x < q));

assert(!(x < y));

assert(!(y > x));

assert(y < x);

assert(!(y > j));

assert(x == x);

}

void Testing::testNode()

{

Book a{ "bbb", "aaa" };

Book b{ "aaa", "bbb" };

Node x(b);

assert(x.getInfo() == b);

assert(x.getLeft() == NULL);

assert(x.getRight() == NULL);

x.setInfo(a);

assert(x.getInfo() == a);

Node \*st = new Node;

Node \*dr = new Node;

x.setLeft(st);

x.setRight(dr);

assert(x.getLeft() == st);

assert(x.getRight() == dr);

x.printInfo();

}

The complexity of the solution is still the same like the functions from the ADT because I use use any additional function, so total complexity should be **O(n)**.