***Project documentation***

***Container***

Binary Search Tree (BST) SortedBag (SB)

A SortedBag is a container where the elements do not have to be unique and there are no positions for the elements and the elements are ordered using a relation. There are no operands that work with positions in a SortedBag.

Domain

**B**={b | b is a SortedBag with elements of type TComp}

Representation

***Node******SortedBag***

info: TComp root: ↑Node

leftChild: ↑Node relation: ↑Relation

rightChild: ↑Node

parent: ↑Node

Interface

* ***create(relation)***

description: creates a new and empty SortedBag

pre: ture

post: b ∈ **B**, b is an empty SortedBag, rel will have the ‘relation’ property

* ***destroy(root)***

description: destroys a SortedBag

pre: root: ↑Node

post: b was destroyed (the allocated memory has been freed)

* ***add(parent, root, data)***

description: adds the element data to the SortedBag b

pre: parent: ↑Node , root: ↑Node, data ∈ TComp

post: b’ ∈ B, b’ = b U {data} (TComp data is added to the SortedBag)

* ***del(root, data)***

description: removes the element data from the SortedBag b if it exists

pre: root: ↑Node, data ∈ TComp

post: b’ ∈ **B**, b’ = b \ {data} (one ocurrence of data was removed from the

SortedBag). If data is not in b, b is not changed.

* ***find(root, data)***

description: returns a pointer to a node if the element data exists in the

SortedBag, NIL otherwise

pre: root: ↑Node, data ∈ TComp

post: if data ∈ **B**, ***find*** ← ↑Node, else ***find*** ← NIL

* ***findMin(root)***

description: returns a pointer to the left most node of the given root

pre: root: ↑Node, data ∈ TComp

post: if the root does not have a left node, it returns the ↑root

* ***count(root, counter)***

description: returns the number of elements from the SortedBag

pre: : root: ↑Node, counter: Integer

post: ***count*** ← the number of elements from b

* ***iterator(root, i)***

description: iterates through all the elements in the bag

pre: b ∈ **B**

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

Pseudocode implementation of the interface operation for the SortedBag

* subalgorithm ***create(relation)*** is:

rel<-relation

root<-NIL

end-subalgorithm – complexity **O(1)**

* subalgorithm ***destroy(root)*** is:

if root !=NIL then:

destroy(root.left)

destroy(root.right)

free(root)

endif

end-subalgorithm – complexity **O(n)**

* function ***add(parent, root, data)*** is:

if root != NIL then:

root<- new(Node)

root.data<- data

root.parent<- parent

root.left<- NIL

root.right<-NIL

else if rel(data,root.data)=1 then:

root.left<- add(root,root.left,data)

else

root.right<- add(root,root.right,data)

add<- root

end-function – complexity **O(log n)**

* function ***del(root, data)*** is:

if root=NIL then:

del<- NIL

else if rel(data,root.data)=1 then:

root.left<-del(root.left,data)

else if rel(data,root.data)=-1 then:

root.right<-del(root.right,data)

else

if root.left=NIL and root.right=NIL then:

free(root)

root<-NIL

else if root.left=NIL then:

temp<-new(Node)

temp<- root.right

root.data<-root.right.data

root.left<-root.right.left

root.right<-root.right.right

free(temp)

else if root.right=NIL then:

temp<-new(Node)

temp<-root.left

root.data<-root.left.data

root.right<-root.left.right

root.left<-root.left.left

free(temp)

else:

temp<-new(Node)

temp<-findMin(root.right)

root.data<-temp.data

root.right<-del(root.right,temp.data)

endif

del<- root

end-function – complexity **O(log n)**

* function ***find(root, data)*** is:

if root!=NIL then:

if data=root.data then:

find<-root

endif

if rel(data,root.data)=1 then:

find<- find(root.left,data)

else:

find<- find(root.right,data)

endif

else

find<-NIL

end-function – complexity **O(log n)**

* function ***findMin(root)*** is:

while root.left != NIL execute:

root<-root.left

done

findMin<-root

end-function – complexity **O(log n)**

* function ***count(root, counter)*** is:

if root=NIL then:

count<-void

endif

counter<- counter +1

if root.left !=NIL then:

count(root.left,counter)

endif

if root.rigth !=NIL then:

count(root.right,counter)

end-function – complexity **O(n)**

* function ***iterator(root, i)*** is:

iterator<- Iterator(root)

end-function – complexity **O(1)**

***Iterator***

Binary Search Tree (BST) SortedBag (SB) Iterator

An iterator is a structure that is used to iterate throughout the elements of a container, in this case, a SortedBag. An iterator usually contains: a reference to the container it iterates over and another reference to a current element from the container. Iterating through the elements of the container means actually moving said current element from one element to another until the iterator becomes invalid.

Domain

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

Representation

*Iterator*

root: ↑Node

Interface

* ***init(i, root)***

description: creates a new iterator for a SortedBag

pre: b ∈ **B**

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

* ***isValid(i)***

description: returns true if the root is valid, false otherwise

pre: i ∈ **I**

post: if the root from I is a valid one, valid ← true,

else valid ← false

* ***next(i)***

description: performs the inorder traversal of the sorted bag;

jumps to the next element from the SortedBag or makes the

iterator invalid if no elements are left

pre: i ∈ **I**, valid(i)

post: i’ ∈ **I** , the current element (the root which changes constantly) from i’

refers to the next element from the SortedBag b

* ***getValue(i)***

description: returns the data of the root from the iterator

pre: i ∈ **I**, valid(i)

post: getValue ← the data from the root

* ***DeeperToLeft(root)***

description: root becomes the left most node from the sorted bag

pre: root: ↑Node

post: root is the left most node from the sorted bag

Pseudocode implementation of the interface operation for the Iterator

* subalgorithm ***init(i,root)*** *is:*

root<-root

DeeperToLeft(root.left)

end-subalgorithm - complexity **O(n)**

* function ***isValid(i)*** is:

if root!=NIL then:

isValid<-true

else

isValid<-false

end-if

end-function – complexity **O(1)**

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

if DeeperToLeft(root.right)=false then:

if root.parent!=NIL then:

while root.parent!=NIL and root=root.parent.right execute:

root<-root.parent

done

root<-root.parent

endif

endif

end-subalgorithm – complexity **O(1)**

* function ***getValue(i)*** is:

getValue<-root.data

end-function – complexity **O(1)**

* function ***DeeperToLeft(root)*** is:

if root=NIL then:

DeeperToLeft<-false

endif

if DeeperToLeft(root.left)=false then:

DeeperToLeft<-true

endif

end-function – complexity **O(log n)**

Tests for the functions written in C++

void Tests::testSearch() {

SortedBag<int> BST{ ">" };

BST.insert(5);

BST.insert(10);

BST.insert(3);

BST.insert(4);

BST.insert(1);

BST.insert(11);

assert(BST.search(5) == true);

assert(BST.search(4) == true);

assert(BST.search(2) == false);

assert(BST.search(1) == true);

assert(BST.search(12) == false);

}

void Tests::testRemove() {

SortedBag<int> BST{ ">" };

BST.insert(5);

BST.insert(10);

BST.insert(3);

BST.insert(4);

BST.insert(1);

BST.insert(11);

assert(BST.remove(5) == true);

assert(BST.remove(123) == false);

assert(BST.remove(12) == false);

assert(BST.remove(1) == true);

assert(BST.remove(11) == true);

}

void Tests::testSize() {

SortedBag<int> BST{ ">" };

BST.insert(5);

BST.insert(10);

BST.insert(3);

BST.insert(4);

BST.insert(1);

BST.insert(11);

assert(BST.size() == 6);

assert(BST.remove(5) == true);

assert(BST.size() == 5);

assert(BST.remove(10) == true);

assert(BST.remove(3) == true);

assert(BST.remove(4) == true);

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

}

void Tests::testIterator()

{

SortedBag<int> BST{ ">" };

BST.insert(5);

BST.insert(10);

BST.insert(3);

BST.insert(4);

BST.insert(1);

BST.insert(11);

Iterator<int> it = BST.iterator();

it.next();

assert(it.getValue()==3);

int k = 0;

while (k < BST.size() - 1)

{

k++;

it.next();

}

if (it.isValid() == NULL)

k++;

}

void Tests::testAll() {

testSearch();

testRemove();

testSize();

testIterator();

}

***Problem statement***

Product sales tree

A company needs to store their daily product sales in a program and may

chose to sort them in ascending order, add sales, remove sales.

We will use a SortedBag ADT (implemented on a Binary Search Tree) to

ensure the storing of similar elements.

Choice

The Sorted Bag is the best way of approaching this problem because, any element can be repeated and the position of the elements does not matter in the end, because the company wants to see how many sales they did. The elements can be sorted as the company wants, usually in ascending order.