DATA STRUCTURES AND ALGORITHMS

Name: Vanessa-Alicia Pasca

Group: 915

* **Project 25- SortedList on Binary Search Tree**
* **Specification and interface of the ADT**

A list can be seen as a sequence of elements of the same type , where there is an order of the elements, and each element has a position inside the list. The order of elements is important ( positions are relevant ).

A SortedList is a list where the elements are ordered by a given Relation.

**SortedList**={sl | sl is a SortedList with elements of type TComp, each having a unique position of type TPosition and being ordered by a Relation}

Our TPosition will be Integer.

* init(sl,rel)
  + descr: creates a new, empty sorted list
  + pre: r ∈ Relation
  + post: sl ∈ SortedList, sl is an empty sorted list
* destroy(sl)
  + descr: frees the memory occupied by the SortedList sl
  + pre: sl ∈ SortedList
  + post: sl was destroyed, memory freed
* size(sl)
  + descr: returns the number of elements from the sorted list
  + pre: sl ∈ SortedList
  + post: size ← the number of elements from sl
* isEmpty(sl)
  + descr: checks if the sorted list is empty
  + pre: sl ∈ SortedList
  + post: isEmpty ← true if sl = ∅ , false otherwise
* search( sl, e)
  + descr: search for an element in the list
  + pre: sl ∈ SortedList, e ∈ TComp
  + post: search ← true if e was found, false otherwise
* remove(sl,e)
  + descr: removes an element from the list
  + pre: sl ∈ SortedList, e ∈ TComp
  + post: e ∈ TComp, sl’ ∈ SortedList, sl’ = sl – e
* remove(sl,pos)
  + descr: removes an element from a given position
  + pre: sl ∈ SortedList, pos ∈ TPosition
  + post: e ∈ TComp, e is the element from position p from SortedList, sl’ ∈ SortedList, sl’ = sl – e
  + throws: exception if p is not valid
* add( sl,e)
  + descr: adds a new element to the list ( based on the relation)
  + pre: sl ∈ SortedList, e ∈ TComp
  + post: sl’ , sl’ = sl + e ( e was added to the list )
* position(sl,e)
  + descr: returns the position of an element
  + pre: sl ∈ SortedList, e ∈ TComp
  + post: position ← p ∈ TPosition , where p is:
    - the first position of e in sl , if e ∈ sl
    - ⊥ otherwise
* getElement(sl,pos)
  + descr: gets the element from a given TPosition
  + pre: sl ∈ SortedList, pos ∈ TPosition
  + getElement ← e ∈ TElem, e = the element from position p from l
  + throws: exception if p is not valid
* valid(sl, p)
  + descr: checks weather the position is in the range of the SortedList
  + pre: sl ∈ SortedList, p ∈ TPosition
  + valid ← true, if p is in the range of the sl , false otherwise
* iterator(sl,it)
  + descr: returns an iterator for the SortedList
  + pre: sl ∈ SortedList
  + post: :it ∈ I, it is an iterator over sl
* **Iterator SortedList**

An iterator is a structure that enables us to traverse a container, in our case a SortedList.

**I**= {it | it is an iterator over a SortedList with elements of type TComp which uses an auxiliary Stack}

**IteratorSortedList:**

stack: Stack

current: ↑Node

sl: SortedList

* init(it,sl)
  + descr: creates an iterator over sl
  + pre: sl ∈ SortedList
  + post :it ∈ I ( it is an iterator over sl )
* valid(it)
  + descr: checks if the iterator is valid
  + pre: it ∈ I
  + post: valid ← true , if the iterator is valid, false otherwise
* next(it)
  + descr: goes to the next element from the iteration
  + pre: it ∈ I , it valid
  + post: current contains now the next element of the SortedList
* getCurrent(it)
  + descr: gets the current element in the iteration
  + pre: it ∈ I , it valid
  + post: getCurrent ← e ∈ TComp , where e is the current element in the iteration

Representation of the ADT

**Node:**

info:TComp

left: ↑Node

right: ↑Node

counterRight:Integer

counterLeft:Integer

**SortedList:**

r: ↑ Relation

root: ↑Node

subalgorithm init(sl,relation)

sl.root←NIL

sl.r←relation

end-subalgorithm

Complexity: O(1)

subalgorithm listWithSameRelation(sl)

init(newOne,sl.r)

listWithSameRelation←newOne

end-subalgorithm

Complexity: O(1)

subalgorithm add(sl, e)

allocate(newNode)

[newNode].info ← e

[newNode].left ← NIL

[newNode].right ← NIL

**if sl.root=NIL then**

sl.root←newNode

**else**

ok←1

currentNode←sl.root

**while ok=1 do**

**if sl.r([newNode].info,[currentNode].info) then**

if ([currentNode].left=NIL)

[currentNode].counterLeft←[currentNode].counterLeft+1

[currentNode].left←newNode

ok←0

**else**

[currentNode].counterLeft ← [currentNode].counterLeft + 1

currentNode← [currentNode].left

**end-if**

**else**

if [currentNode].right = NIL then

[currentNode].counterRight ← [currentNode].counterRight + 1

[currentNode].right ← newNode

ok ← 0

**else**

[currentNode].counterRight ← [currentNode].counterRight + 1

currentNode ← [currentNode].right

**end - if**

**end-if**

**end-while**

**end-if**

end-subalgorithm

Complexity:

BC: θ(1) , when we add on the 1st level of the tree

WC: θ(n) , when what we have a completly unbalanced tree and we want to add on the last level

* this is actaully similar to adding at the end of a SLL: we traverse all the n elements of out tree so the complexity will be: 1\*n =n

AC:

θ(log(n))

subalgorithm findMin(sl,node)

**if node=NIL then**

findMin←NIL

**end-if**

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

findMin←findMin(sl,[node].left)

**end-if**

findMin←node

end-algorithm

Complexity:

BC: θ(1) , when the minimum is exactly the root

WC: θ(n) , when what we have a completly unbalanced tree on the left side

* we have to traverse all the elements: sum(1) , where i= 1->n

AC:

θ(log(n))

subalgorithm size(sl)

**if sl.root=NIL then**

size←0

**end-if**

size←[sl.root].counterLeft+[sl.root].counterRight +1

end-subalgorithm

Complexity: O(1)

subalgorithm isEmpty(sl)

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

isEmpty←true

**else**

isEmpty←false

**end-if**

end-subalgorithm

Complexity: O(1)

subalgorithm search(sl,e)

**if sl.root=NIL then**

search←false

**end-if**

currentNode←sl.root

**while currentNode !=NIL do**

**if sl.r(e,[currentNode].info) then**

currentNode←[currentNode].left

**else**

**if [currentNode].info=name then**

search←true

**end-if**

currentNode←[currentNode].right

**end-if**

**end-while**

search←false

end-subalgorithm

Complexity:

BC: θ(1) , when the name we are looking for is the root

WC: θ(n) , when what we have a completly unbalanced tree and we’re looking for the last element

* we have to traverse all the elements: sum(1) , where i= 1->n

AC:

θ(log(n))

Overall: O(n)

subalgorithm removeBST(sl,node,name)

enable←search(sl,name)

**if enable=true then**

**if sl.root=NIL then**

removeBST←NIL

**else**

**if sl.r(name,[node].info) then**

[node].counterLeft←[node].counterLeft-1

[node].left←removeBST(sl,[node].left,name)

**else**

**if name=[node].info then**

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

**if node=sl.root then**

sl.root←NIL

**end-if**

delete node

node←NIL

**else**

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

temp←node

**if node=sl.root then**

sl.root←[sl.root].right

**end-if**

node←[node].right

delete temp

temp←NIL

**end-if**

**else**

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

temp←node

**if node=sl.root then**

sl.root←[sl.root].left

**end-if**

node [node].left

delete temp

temp←NIL

**end-if**

**else**

temp←findMin([node].right)

[node].name←[temp].name

[node].counterRight=[node].counterLeft – 1

[node].right=removeBST([node].right,[temp].name)

**end-if**

**end-if**

**else**

[node].counterRight←[node].counterRight-1

[node].right←removeBST([node].right,name)

**end-if**

removeBST←node

**else**

@throw exception: e not in list

**end-if**

end-subalgorithm

Complexity:

In each case we add the complexity of search to the complexity of the following cases:

Leaf: θ(1)

One child: θ(1)

Two children:

BC: θ(1) , when we only have one node to the right of our aimed node

WC: θ(n) , when we have a completly unbalanced tree to the right

AC:

θ(log(n))

So, the search+ three cases complexity gives:

BC: θ(1) , search and remove are both θ(1)

WC: θ(n) , for example when search is θ(1) , we want to remove the root, but it’s right is a completly unbalanced tree

AC:

θ(log(n))

Overall: O(n)

subalgorithm remove(sl,e)

node←removeBST(sl,sl.root,e)

end-subalgorithm

Complexity: same as removeBST

subalgorithm removeFromPosition(sl,pos)

name←getElement(sl,pos)

remove(sl,name)

removeFromPosition←name

end-algorithm

Complexity:

getElement+ remove

So:

BC: θ(1) ,

WC: θ(n)

AC:

θ(log(n))

Overall: O(n)

subalgorithm position(sl,e)

currentNode←sl.root

pos←[currentNode].counterLeft+1

**if [currentNode].info=e then**

position←pos

**end-if**

enable←search(sl,e)

**if enable=true then**

ok←true

**while ok=true do**

**if [currentNode].info=e then**

position←pos

ok←false

**end-if**

**if sl.r(e,[currentNode].info) then**

currentNode←[currentNode].left

pos←pos -1 – [currentNode].counterRight

**else**

currentNode←[currentNode].right

pos←pos +1 +[currentNode].counterLeft

**end-if**

**end-while**

**else**

@throw exception: e is not in list

**end-if**

end-subalgorithm

Complexity:

BC: θ(1) , when the name we are looking for is the root

WC: θ(n) , when what we have a completly unbalanced tree and we’re looking for the last element

* we have to traverse all the elements: sum(1) , where i= 1->n

AC:

θ(log(n))

Overall: O(n)

subalgorithm getElement(sl,pos):

currentNode←sl.root

**if pos<=0 or pos>size(sl) then**

@throw exception

**end-if**

ok←true

**while ok=true do**

**if [currentNode].counterLeft+1 = pos then**

getElement←[currentNode].info

ok=false

**else**

**if [currentNode].counterLeft >= pos then**

currentNode←[currentNode].left

**else**

pos←pos – [currentNode].counterLeft -1

currentNode←[currentNode].right

**end-if**

**end-if**

**end-while**

end-algorithm

Complexity:

BC: θ(1) , when the position we are looking for is the position of the root

WC: θ(n) , when what we have a completly unbalanced tree and we’re looking for the last position or to the 1st ( depends in which side is unbalanced)

* we have to traverse all the elements: sum(1) , where i= 1->n

AC:

θ(log(n))

Overall: O(n)

subalgorithm destroy(sl)

**while size(sl)>0 do**

removeBST(sl.root,[sl.root].info)

**end-while**

end-subalgorithm

Complexity:

the nr. of elements in the SortedList\* complexity of remove operation (where search operation is always O(1) – the root)

BC: θ(1) We only have the root

WC: θ(n^2 ) . We start with the worst case scenario, for example we have an unbalanced tree at the left of the right child of the root, and a single node to the left. The first remove will have θ(n-1), the next one θ(n), and so on until we reach one => sum(i), where i goes from 1 to n-1

AC:

θ(n\*log(n))

Overall: O(n)

subalgorithm iterator(sl,it)

init(it,sl)

end-subalgorithm

Complexity: O(1) , amortized (without the init’s operation complexity). Otherwise, it’ll have the init’s operation complexity. (described just below)

ITERATOR

subalgorithm init(it,sl)

it.sl←sl

it.current←findMin(sl,sl.root)

currentNode←sl.root

**while currentNode!=NIL do**

**if it.current!=currentNode then**

it.stack.push(currentNode)

**end-if**

currentNode←[currentNode].left

**end-while**

currentNode←sl.root

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

**if [currentNode].right!=NIL then**

currentNode←[currentNode].right

**while currentNode!=NIL do**

it.stack.push(currentNode)

currentNode←[currentNode].left

**end-while**

**end-if**

**end-if**

end-subalgorithm

Complexity:

BC: θ(1) , when we have an unbalanced tree to the right

WC: θ(n) , when what we have a completly unbalanced tree to the left, meaning we have to put in the stack all the elements

* we have to traverse all the elements: sum(1) , where i= 1->n

AC:

θ(log(n))

Overall: O(n)

subalgorithm next(it)

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

currentNode←it.stack.top()

it.current←currentNode

it.stack.pop()

**if [currentNode].right!=NIL then**

currentNode←[currentNode].right

while currentNode!=NIL do

it.stack.push(currentNode)

currentNode←[currentNode].left

**end-while**

**end-if**

**else**

it.current←NIL

**end-if**

end-subalgorithm

Complexity:

BC: θ(1) ,when I have nothing in the right of my current

WC: θ(n) , when what we have a completly unbalanced tree in the left of the right node relative to current

* we have to traverse all the elements: sum(1) , where i= 1->n

AC:

θ(log(n))

Overall: O(n)

subalgorithm valid(it)

**if it.current!=NIL then**

valid←true

**else**

valid←false

**end-if**

end-subalgorithm

Complexity: O(1)

subalgorithm getCurrent(it)

getCurrent=[it.current].info

end-subalgorithm

Complexity: O(1)

* **Problem statement**

A fitness instructor holds a TRX Class every day.

He wants to keep evidence of the people coming, storing them in an alphabetically list, being able to add new people or remove people from the Class. In case a new person will join, the instructor will add it to the list; in case one person drops, he will remove his/her name from the list.

Moreover, during the training he plans to perform a group exercise, splitting the members in 3 groups, based on the position of their name in the list. He will in the list,choose the positions and the groups will be formed accordingly.

In order to solve this problem, we will use a sorted data structure to hold the people, because they need to be stored alphabetically. Because also the positions of the people are relevant, the sorted data structure will be a list. Therefore , the ADT needed will be a Sorted List.

SOLUTION

UI:

sl:SortedList

gr1:SortedList

gr2:SortedList

filename:String

subalgorithm init(ui,sl,nameOfFile)

// pre: sl is a SortedList, nameOfFile is a string

// post:

// ui belongs to UI, where ui.sl= sl and ui.filename=NameOfFile

ui.sl←sl

ui.filename←nameOfFile

ui.gr1←listWithSameRelation(sl)

ui.gr2←listWithSameRelation(sl)

end-subalgorithm

Complexity: O(1)

subalgorithm printMenu(ui)

@print the following menu

Options are:

1.Add new member

2.Remove member

3.Split member into 2 groups

4.Display all members

0.Exit

end-subalgorithm

Complexity: O(1)

subalgorithm run(ui)

//Displays the menu by calling printMenu(ui), collects the input and makes function calls in order to execute the commands

// cmd:Integer

ok←true

**while ok←true do**

printMenu(ui)

cmd←-1

**do**

@print: Give your command

@ read command in cmd

**while cmd<0 and cmd>4**

**if cmd=0 then**

@exit

**else**

**if cmd=1 then**

addUI(ui)

**end-if**

**else**

**if cmd=2 then**

removeUI(ui)

**end-if**

**else**

**if cmd=3 then**

manageGroups(ui)

**end-if**

**else**

**if cmd=4 then**

@print: “All members are:”

displayAll(ui,ui.sl)

**end-if**

**end-if**

**end-while**

end-subalgorithm

Complexity:

Number of times until we press exit is multiplyed with one of the following cases:

BC: θ(1) , cmd=0 and we exit or in case we run a command will be θ(n) complexity (function called complexity)

WC: θ(n^2) , when one of the function has complexity θ(n^2)

AC:

θ(nlog(n))

Overall: O(n)

subalgorithm displayAll(ui,sl)

//Pre: ui is a UI, sl is SortedList

//Displays the content of sl

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

@print: "No members to display!"

**else**

iterator(ui.sl,it)

i←0

**while valid(it) do**

i←i+1

name←getCurrent(it)

@print i followed by the name

next(it)

**end-while**

**end-if**

end- subalgorithm

Complexity is given by:

the init of the iterator

+ the number of elements in the stack\* init calling opperation

BC: θ(n) , when next has complexity θ(1)

WC: θ(n^2) , when next operation starts with a complexity θ(n)

AC:

θ(nlog(n))

Overall: O(n^2)

subalgorithm addUI(ui)

//Reads a name and adds it to ui.sl

//name:String

@print: Give name of the person

name←@read a name

add(ui.sl,name)

@print: All members are:

displayAll(ui,ui.sl)

writeToFile(ui)

end- subalgorithm

Complexity:

diaplayAll()+add()+writeToFile()

BC: θ(n) , when add has θ(1) and displayAll θ(n)

WC: θ(n^2) , given by displayAll’s worst case

AC:

θ(nlog(n))

without displayAll and writeToFile the complexities would be the same as add(sl,e)

Overall: O(n^2)

subalgorithm removeUI(ui)

//Removes from the sorted list of ui, by name or by position

//cmd: Integer , name:String, pos:Integer

@Print the following:

How do you want to remove?

1.By name

2.By position

**do**

@print : Enter option

cmd←@read

**while cmd<=0 or cmd>2**

**if cmd=1 then**

@print: Give name:

name←@read

**try**

remove(ui.sl,name)

@print: All members are:

displayAll(ui,ui.sl)

**catch** @exception: no member with such a name

@print a message

**end-try**

**else**

@print:Give position to be removed

pos←@read

**try**

removedName←removeFromPosition(ui.sl,pos)

@print the removed name

@print: All members are:

displayAll(ui,ui.sl)

**catch** @exception: position doesn’t exist

@print message

**end-try**

**end-if**

writeToFile(ui)

end-subalgorithm

Complexity:

diaplayAll()+add()+writeToFile()

BC: θ(n) , when remove has θ(1) and displayAll θ(n)

WC: θ(n^2) , given by displayAll’s worst case

AC:

θ(nlog(n))

without displayAll and writeToFile the complexities would be the same as remove(sl,e)

Overall: O(n^2)

subalgorithm printGroupMenu(ui)

@print the following menu:

Options are:

1.Add members to groups

2.Remove member from groups

3.Display all members of the groups

0.Exit

end- subalgorithm

Complexity: O(1)

subalgorithm manageGroups(ui)

// Displays the menu by calling printGroupMenu(ui)

//Reads a command and makes functions calls in order to execute it

//cmd:Integer

@print: We’ll gonna take care of the groups:

ok←true

**while ok=true do**

printGroupsMenu(ui)

**do**

@print: Give command:

cmd←@read

**while cmd<0 or cmd>3**

**if cmd=0 then**

run(ui)

**else**

if cmd=1 then

makeGroups(ui)

end-if

**else**

if cmd=2 then

removeFromGroups(ui)

end-if

**else**

@print: Members of group one:

displayAll(ui.gr1)

@print: Members of group two:

displayAll(ui.gr2)

**end-if**

**end-while**

end-subalgorithm

Complexity:

Number of times until we press 0 is multiplyed with one of the following cases:

BC: θ(1) , cmd=0 and we exit or in case we run a command will be θ(n) complexity (function called complexity)

WC: θ(n^2) , when one of the function has complexity θ(n^2)

AC:

θ(nlog(n))

Overall: O(n^2)

subalgorithm makeGroups(ui)

//Manages the adding to the group: reads positions and adds then , either to ui.gr1 or to ui.gr2

//pos1,pos2:Integer[] , pos:Integer

//post: ui.gr1 and ui.gr2 might get populated

@print: Give positions to be part of group one, 0 to stop

pos1←@vector of integers

pos2←@vectors of integers

pos←@read

**while pos!=0 do**

**try**

name←getElement(ui.sl,pos)

@find pos in pos1

@find pos in pos2

**if @pos was not found then**

add(ui.gr1,name)

@print: Members of gr 1 are:

displayAll(ui,ui.gr1)

@add pos to pos1

**else**

@Print: pos is Already in groups

**end-if**

**catch** @exception: no such a position

@print message

**end-try**

@print:Give the position:

pos←@read

**end-if**

@print: Give positions to be part of group two, 0 to stop

pos←@read

**while pos!=0 do**

**try**

name←getElement(ui.sl,pos)

@find pos in pos1

@find pos in pos2

**if @pos was not found then**

add(ui.gr1,name)

@print: Members of gr 2 are:

displayAll(ui,ui.gr2)

@add pos to pos2

**else**

@Print: pos is Already in groups

**end-if**

**catch** @exception: no such a position

@print message

**end-try**

@print:Give the position:

pos←@read

**end-if**

end- subalgorithm

Complexity:

Because of the displayAll operation we’ll get:

BC: θ(n) , when add has θ(1) and displayAll θ(n)

WC: θ(n^2) , given by displayAll’s worst case

AC:

θ(nlog(n))

without displayAll:

Complexities of add:

θ(1)- for example when we add just one member to the groups and this adding has θ(1)

θ(n^2)- we add all members to one group in a way that we obtain an unbalanced tree

n\*n= n^2

θ(n\*log(n))

Overall: O(n^2)

subalgorithm removeFromGroups(ui)

//Removes a given name from the groups: it identifies which group the name belongs to and calls the remove function in order to eliminate it

// res,res2: Boolean, name:String

@print: Give name to be removed:

name←@read

res←search(ui.gr1,name)

**if res=true then**

@print the name which will be removed from gr1

remove(ui.gr1,name)

@print: Members of gr 1 are:

displayAll(ui,ui.gr1)

**else**

res2←search(ui.gr2,name)

**if res2=true then**

@print the name which will be removed from gr2

remove(ui.gr2,name)

@print: Members of gr 2 are:

displayAll(ui,ui.gr2)

**else**

@print: The member is not part of any group

**end-if**

**end-if**

end- subalgorithm

Complexity:

Because of the displayAll operation we’ll get:

BC: θ(n) , when remove has θ(1) and displayAll θ(n)

WC: θ(n^2) , given by displayAll’s worst case

AC:

θ(nlog(n))

without displayAll:

Complexities of search+remove:

θ(1)

θ(n)

θ(log(n))

Overall: O(n^2)

subalgorithm readFromFile(ui)

//Reads from the file the names for the list

@open a file with the name ui.filename for reading

**while @I can read a name from the file do**

add(ui.sl,name)

**end-while**

end- subalgorithm

Complexity:

BC: θ(n\*log(n))

* we read all the n elements and we construct a balanced tree out of it

WC:

θ(n^2)

-we read the n elements but we construct an unbalanced tree

subalgorithm writeToFile(ui)

//Writes the sorted list to a file

@open a file with the name ui.filename for writing

iterator(ui.sl,it)

**while valid(it) do**

name←getCurrent(sl)

@write in file the name

next(it)

**end-while**

end- subalgorithm

Complexity: Given by the next operation\* number of members

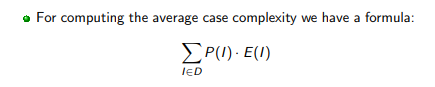
BC: θ(n), for example in an unbalanced tree to the left , where next has θ(1)

WC: θ(n^2) when the next operation has the worst case

AC: θ(nlog(n))

Overall: O(n^2)

PROOF FOR THE AVERAGE/BEST CASE:



P(I) is the probability that we will have I as an input

E(I) is the number of operations performed by the algorithm

In our case P is 1/h and E goes from 1 to h

Thus, what we get is Sum(1/h\*i) where i goes from 1 to h => O(h)

Let n be the number of nodes in a perfect binary tree and let lk denote the number of nodes on level k.

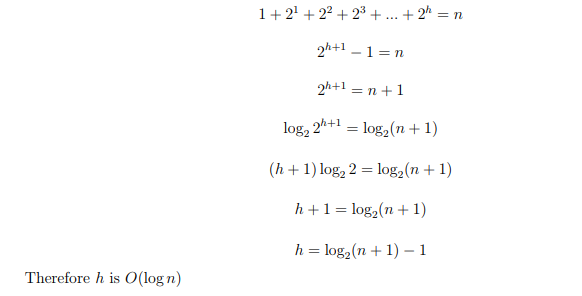
Note that:

• each level has exactly twice as many nodes as the previous level

• on the “first level” we have only one node (the root node).

• the leaves are at the last level, lh, where h is the height of the tree.

The total number of nodes in the tree is equal to the sum of the nodes on all the levels: nodes n.



And the average case will therefore be O(log(n))

bool relationTest(std::string a, std::string b) {

if (a < b)

return true;

else

return false;

}

void testAdd(SortedList& sl) {

sl.add("Maria");

sl.add("Andreea");

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

assert(sl.getRoot()->getCL() == 1);

sl.add("Dan");

assert(sl.size() == 3);

assert(sl.getRoot()->getCL() == 2);

sl.add("Tarek");

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

assert(sl.getRoot()->getCR() == 1);

sl.add("Paul");

sl.add("Rares");

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

assert(sl.getRoot()->getCR() == 3);

sl.add("Raluca");

assert(sl.size() == 7);

assert(sl.getRoot()->getCR() == 4);

sl.add("Bob");

sl.add("Ana");

assert(sl.size() == 9);

assert(sl.getRoot()->getCL() == 4);

sl.add("Will");

assert(sl.size() == 10);

assert(sl.getRoot()->getCR() == 5);

}

void testSearch(SortedList& sl) {

bool res = sl.search("Rares");

assert(res == true);

res = sl.search("Raul");

assert(res == false);

res = sl.search("Ana");

assert(res == true);

res = sl.search("Marin");

assert(res == false);

}

void testGetFromPosition(SortedList& sl) {

std::string name = sl.getElement(2);

assert(name == "Andreea");

name = sl.getElement(9);

assert(name == "Tarek");

name = sl.getElement(10);

assert(name == "Will");

name = sl.getElement(1);

assert(name == "Ana");

name = sl.getElement(3);

assert(name == "Bob");

bool exceptia = false;

std::string message;

try {

name = sl.getElement(18);

}

catch(InexistentPositionException& exc){

exceptia = true;

message= exc.what();

}

assert(exceptia == true);

assert(strstr(message.c\_str(), "range"));

}

void testPosition(SortedList& sl) {

int pos = sl.position("Rares");

assert(pos == 8);

pos = sl.position("Raluca");

assert(pos == 7);

pos = sl.position("Maria");

assert(pos == 5);

pos = sl.position("Andreea");

assert(pos == 2);

pos = sl.position("Dan");

assert(pos == 4);

pos = sl.position("Bob");

assert(pos == 3);

pos = sl.position("Tarek");

assert(pos == 9);

pos = sl.position("Paul");

assert(pos == 6);

bool exceptia = false;

std::string message;

try {

pos = sl.position("Marcel");

}

catch (InexistentMemberException& exc) {

message = exc.what();

exceptia = true;

}

assert(strstr(message.c\_str(), "No such member"));

assert(exceptia == true);

}

void testMin(SortedList& sl) {

Node \*n = sl.findMin(sl.getRoot());

assert(n->getName() == "Ana");

n= sl.findMin(sl.getRoot()->getRight());

assert(n->getName() == "Paul");

}

void testRemove(SortedList& sl) {

sl.remove("Tarek");

assert(sl.getRoot()->getRight()->getName() == "Will");

assert(sl.size() == 9);

assert(sl.getRoot()->getCR() == 4);

sl.remove("Bob");

assert(sl.size() == 8);

assert(sl.getRoot()->getCL() == 3);

sl.remove("Andreea");

assert(sl.size() == 7);

assert(sl.getRoot()->getCL() == 2);

assert(sl.getRoot()->getLeft()->getName() == "Dan");

sl.remove("Rares");

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

assert(sl.getRoot()->getCR() == 3);

sl.remove("Will");

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

assert(sl.getRoot()->getCR() == 2);

assert(sl.getRoot()->getRight()->getName() == "Paul");

sl.remove("Ana");

sl.remove("Dan");

assert(sl.getRoot()->getCL() == 0);

sl.remove("Paul");

sl.remove("Raluca");

assert(sl.getRoot()->getCR() == 0);

sl.remove("Maria");

assert(sl.isEmpty() == true);

}

void testIterator(SortedList& sl) {

Iterator it{};

sl.iterator(it);

std::string name;

name = it.getCurrent();

assert(name == "Ana");

it.next();

name = it.getCurrent();

assert(name == "Andreea");

it.next();

name = it.getCurrent();

assert(name == "Bob");

it.next();

name = it.getCurrent();

assert(name == "Dan");

it.next();

name = it.getCurrent();

assert(name == "Maria");

it.next();

name = it.getCurrent();

it.next();

it.next();

it.next();

it.next();

name = it.getCurrent();

assert(name == "Will");

it.next();

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

}

void testRemovePosition(SortedList& sl) {

assert(sl.removeFromPosition(2) == "Andreea");

assert(sl.removeFromPosition(2) == "Bob");

assert(sl.removeFromPosition(2) == "Dan");

assert(sl.removeFromPosition(2) == "Maria");

assert(sl.getRoot()->getName() == "Paul");

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

assert(sl.removeFromPosition(2) == "Paul");

assert(sl.removeFromPosition(1) == "Ana");

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

assert(sl.getRoot()->getName() == "Raluca");

assert(sl.removeFromPosition(2) == "Rares");

assert(sl.removeFromPosition(3) == "Will");

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

assert(sl.removeFromPosition(1) == "Raluca");

assert(sl.removeFromPosition(1) == "Tarek");

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

}

void testAll() {

SortedList sl{ &relationTest };

testAdd(sl);

testSearch(sl);

testGetFromPosition(sl);

testPosition(sl);

testMin(sl);

testRemove(sl);

testAdd(sl);

testRemovePosition(sl);

testAdd(sl);

testIterator(sl);

}