Project Documentation

Potop Oana-Roxana
Group 813

Project topic: ADT SortedList - implementation on a binary search tree.

Specification and interface

A ***list*** can be seen as a sequence of elements, where there is an order of the elements, and each element has a position inside the list, which is relevant.

A ***sorted list*** is a list where elements are memorized in a given order, based on a relation.

Domain:

SL = {sl | sl is a list of elements of type TComp; there is a relation defined on the set of elements: relation(c1, c2) → true, if c1 should be in front of c2 or they are equal, or false, otherwise; every element has a position}

Interface:

- init(sl, rel):
  descr: creates an empty sorted list
  pre: rel is a relation
  post: sl ∈ SL

- destructor(sl):
  descr: frees the memory occupied by the sorted list
  pre: sl ∈ SL
  post: sl has been destroyed

- insert(sl, elem):
  descr: adds an element to the sorted list

pre: sl ∈ SL, e is a TComp
post: elem has been added to the list accordingly, given the relation

- remove(sl, elem):
  descr: removes an element from the sorted list
  pre: sl ∈ SL, elem is a TComp, elem is a TComp
  post: elem has been removed from sl

- search(sl, elem):
  descr: searches for an element in the sorted list
  pre: sl ∈ SL, elem is a TComp
  post: → true, if elem is in sl
  → false, otherwise

- size(sl)
  descr: returns the number of elements from the sorted list
  pre: sl ∈ SL
  post: size = n, n is an integer

- getByPosition(sl, pos):
  descr: returns the element located at position pos
  pre: sl ∈ SL, pos is an integer, pos ≥ 1, pos ≤ size(sl)
  post: the element at position pos is returned
  throws: exception if pos is not valid

- isEmpty(sl):
  descr: checks if the sorted list is empty
  pre: sl ∈ SL
  post: → true, if sl is empty
  → false, if sl has at least one element

- iterator(sl, it):
  descr: creates an iterator over sl
  pre: sl ∈ SL

post: if ∈ I, it is an iterator over sl

Iterator

An iterator is a structure that is used to iterate through the elements of a container - a sorted list, in this case.

<u>Domain</u>:

I = {it | it is an iterator over a sorted list with elements of type TComp}

<u>Representation</u>:

stack: Stack
sl: SortedList

<u>Interface</u>:

- init(it, sl):
  descr: push to the stack deep on the left
  pre: sl is a sorted list
  post: it ∈ I, it does not contain all the elements of the list

- getCurrent(it)
  descr: returns the first element from the stack
  pre: it ∈ I, it is valid
  post: e is a TComp, e is the first element from the stack
  throws: exception if it is not valid

- next(it):
  descr: checks if the current element has a right child; if it does, it pushes it in the stack and then add to the stack all the children on the left as much as possible
  pre: it ∈ I, it is valid

post: it' ∈ I, the current element from it' points to the next element from the list, or, if there are no elements left, it' is invalid
throws: exception if it' is not valid

- first(it)
  descr: resets the iterator
  pre: it ∈ I
  post: the iterator is resetted

- valid(it):
  description: checks if it is valid (checks if the stack is empty)
  pre: it ∈ I
  post: → true, if it is valid
  → false, otherwise

ADT representation on a Binary Search Tree

BSTNode:
left: ↑BSTNode
right: ↑BSTNode
info: TElem
position: Integer (number of children on the left subtree)

SortedList:
root: ↑BSTNode

Operations in pseudocode

SortedList

subalgorithm init(sl, r) is:
sl.r ← r
sl.root ← NIL //empty list
sl.nr_elements ← 0

end-subalgorithm
//Complexity: θ(1)

subalgorithm init(sl) is:
    sl.r ← NIL
    sl.root ← NIL
    sl.nr_elements ← 0
end-subalgorithm
//Complexity: θ(1)

subalgorithm destroy(sl) is:

end-subalgorithm

function size(sl) is:
    size ← sl.nr_elements
end-function
//Complexity: θ(1)

function isEmpty(sl) is:
    if sl.nr_elements = 0 then
        isEmpty ← true
    else
        isEmpty ← false
    end-if
end-function
//Complexity: θ(1)

function getByPosition(sl, pos) is:
    if 0 > pos or sl.nr_elements ≤ pos then
        getByPosition ← Movie(m)
    end-if
    current ← sl.root
    current_position ← current.poz
    last_position ← 0
    while current_position ≠ pos execute:
        if pos > current_position then
            current_position ← current_position + [current.right].poz
            current_position ← current_position + 1
            current ← current.right

```
        else
            current_position ← current_position - 1
            current_position  ← current.left
        end-if
    end-while
    getByPosition  ← current.info
end-function
//Complexity:
    //BC: θ(1)
    //AV: θ(log2n)
    //WC: θ(n)
    //Overall complexity: O(n)
```

function remove(sl, elem) is:

```
    if sl.root = NIL then
        remove ← false
    end-if
    if [sl.root].info = elem then
        sl.nr_elements ← sl.nr_elements - 1
        if [sl.root].left = NIL and [sl.root].right = NIL then
            sl.root = NIL
        else if [sl.root].left = NIL and [sl.root].right ≠ NIL then
            sl.root ← [sl.root].left
        else if [sl.root].left ≠ NIL and [sl.root].right = NIL then
            sl.root ← [sl.root].right
        else
            min ← [sl.root].right
            min_parent ← sl.root
            while min.left ≠ NIL execute
                min_parent ← min
                min ← [min].left
            end-while
            [sl.root].info ← min.info
            [min_parent].right ← [min].right
        end-if
        remove ← true
    end-if
    current ← sl.root
    parrent ← NIL
    while [current].info ≠ elem execute
        parrent ← current
        if sl.r(elem, [current].info) then
```

```
            current ← [current].left
        else
            current ← [current].right
        end-if
        if current = NIL then
            remove ← false
        end-if
end-while
sl.nr_elements ← sl.nr_elements - 1
aux ← sl.root
while [aux].info ≠ elem execute
        if sl.r(elem, [aux].info) then
            [aux].poz ← [aux].poz - 1
            aux ← [aux].left
        else
            aux ← [aux].right
        end-if
end-while
if [current].left = NIL and [current].right = NIL then
        if [parrent].left = current then
            [parrent].left ← NIL
        else
            [parrent].right ← NIL
        end-if
else if [current].left ≠ NIL and [current].right ≠ NIL then
        min ← [current].right
        min_parrent ← current
        while [min].left ≠ NIL execute
            min_parrent ← min
            min ← min_left
        end-while
        [min_parrent].left ← [min].right
        [current].info ← [min].info
else if [current].left = NIL and [current].right ≠ NIL then
        if [parrent].right = current
            [parrent].right ← [current].right
        else
            [parrent].left ← [current].right
        end-if
else
        if [parrent].right = current then
            [parrent].right ← [current].left
```

```
            else
                  [parrent].left ← [current].right
      end-if
      remove ← true
end-function
//Complexity:
      //BC: θ(1)
      //AV: θ(log2n)
      //WC: θ(n)
      //Overall complexity: O(n)
```

function search(sl, elem) is:

```
      current ← sl.root
      while [current].info ≠ elem execute
            if sl.r(elem, [current].info) then
                  current ← [current].left
            else
                  current ← [current].right
            end-if
            if current = NIL then
                  search ← false
            end-if
      end-while
      if current ≠ NIL then
            search ← true
end-function
//Complexity:
      //BC: θ(1)
      //AV: θ(log2n)
      //WC: θ(n)
      //Overall complexity: O(n)
```

subalgorithm insert(sl, elem) is:

```
      if sl.root = NIL then
            node ← create_node (sl, elem, NIL, NIL, 0)
            sl.root ← node
            sl.nr_elements ← sl.nr_elements + 1
      end-if
      parent ← NIL
      current ← sl.root
      while [current].info ≠ elem or current ≠ NIL execute
```

```
                parrent ← current
                if sl.r(elem, [current].info) then
                        [current].poz ← [current].poz + 1
                        current ← [current].left
                else
                        current ← [current].right
                end-if
                if current = NIL then
                        break
                end-if
                if current ≠ NIL then
                        @throw exception("We already have the element")
                else
                        node ← create_node (sl, elem, NIL, NIL, 0)
                        if sl.r(elem, [parent].info) then
                                [parrent].left ← node
                        else
                                [parrent].right ← node
                        end-if
                        sl.nr_elements ← sl.nr_elements + 1
end-subalgorithm
//Complexity:
        //BC: θ(1)
        //AV: θ(log2n)
        //WC: θ(n)
        //Overall complexity: O(n)
```

function iterator(sl) is:

```
        iterator ← ListIterator(sl)
end-function
//Complexity: θ(1)
```

Operations of the iterator

<u>Iterator</u>

subalgorithm init (it, sl) is:

```
        it.nodes ← allocate BSTNode(sl.nr_elements + 1)
        it.top ← 0
        current ← sl.root
```

```
        while current ≠ NIL execute
                it.nodes[it.top] ← current
                it.top ← it.top + 1
                current ← [current].left
        end-while
        it.top ← it.top - 1
end-subalgorithm
//Complexity:
        //BC: θ(1)
        //AV: θ(log2n)
        //WC: θ(n)
        //Overall complexity: O(n)
```

subalgorithm first(it) is:

```
        it.nodes ← allocate BSTNode(sl.nr_elements)
        it.top ← 0
        current ← sl.root
        while current ≠ NIL execute
                it.nodes[it.top] ← current
                it.top ← it.top + 1
                current ← [current].left
        end-while
        it.top ← it.top - 1
end-subalgorithm
//Complexity:
        //BC: θ(1)
        //AV: θ(log2n)
        //WC: θ(n)
        //Overall complexity: O(n)
```

subalgorithm next(it) is:

```
        current_node ← it.nodes[it.top]
        if [current_node].right ≠ NIL then
                it.nodes[it.top] ← [current_node].right
                it.top ← it.top + 1
                aux ← [[current_node].right].left
                if aux ≠ NIL then
                        while aux ≠ NIL execute
                                it.nodes[it.top] ← aux
                                it.top ← it.top +1
                                aux ← [aux].left
                        end-while
```

```
			end-if
		end-if
		it.top ← it.top - 1
end-subalgorithm
//Complexity:
		//BC: θ(1)
		//AV: θ(log2n)
		//WC: θ(n)
		//Overall complexity: O(n)
```

function valid(it) is:

```
		if it.top ≠ -1 then
			valid ← true
		else
			valid ← false
		end-if
end-function
//Complexity: θ(1)
```

function getCurrent(it) is:

```
		if it.valid = false then
			@throw exception("Invalid iterator")
		end-if
		if it.nodes[it.top] ≠ NIL then
			getCurrent ← [it.nodes[it.top]].info
		end-if
end-function
//Complexity: θ(1)
```

Problem statement

In order to choose what to watch more efficiently, a person wants to create a watchlist with the movies he's interested in, and he keeps them ordered by their rating, from best to worst.

Create an application that allows the person to:

- add a movie to the watchlist (a movie has a title, a rating and additional information)
- remove a movie from the watchlist
- search a movie in the watchlist

- view the watchlist
- get the best rated movie
- show the movie from a specific position
- delete all the movies that are below a given position

Justification:

In the case of this problem, a sorted list fits the requirements perfectly because it gives us access to elements through positions and it keeps the elements sorted all the time. Whenever we insert of remove form the list, the order of the elements is updated, and the search operation is more efficient.

Solution:

UI:

sl: SortedList

```
subalgorithm run(ui) is:
    ui.print_details(ui)
    while 1 execute
        ui.print_details(ui)
        @read option
        if option = 1 then
            ui.add(ui)
        else if option = 2 then
            ui.remove(ui)
        else if option = 3 then
            ui.search(ui)
        else if option =4 then
            ui.print(ui)
        else if option = 5 then
            ui.get_best_movie(ui)
        else if option =6 then
            ui.get_movie_by_index(ui)
        else if option = 7 then
            ui.remove_all(ui)
        else if option =0 then
            run ←
        else
            @print: “Command not found”
        end-if
```

end-subalgorithm
//Complexity:
    //BC: θ(1)
    //AV: θ(log2n)
    //WC: θ(n)
    //Overall complexity: O(n)

subalgorithm config(ui) is:
    Movie m1{9, "Lord of the rings: The Fellowship of the Ring", "Amazing display of J. R. R. Tolkien's book; marks the beginning of an epic journey"}
    Movie m2 {8, "Lord of the rings: The Two Towers", "Second movie of the trilogy Lord of the Rings, epic story, amazing characters"}
    Movie m3 {10, "Lord of the rings: The Return of the King", "Third and last movie of the epic trilogy Lord of the Rings, marks the end of the story"}
    [ui.sl].insert(sl, m1)
    [ui.sl].insert(sl, m2)
    [ui.sl].insert(sl, m3)
end-subalgorithm
//Complexity: θ(1)

subalgorithm init(ui, r) is:
    ui.sl ← allocate SortedList(r)
end-subalgorithm
//Complexity: θ(1)

subalgorithm init(ui, other) is:
    it.sl ← other.sl
end-subalgorithm
//Complexity: θ(1)

subalgorithm destroy(ui) is:

end-subalgorithm
//Complexity: θ(1)

subalgorithm add(ui) is:
    @print: "Enter name"
    @read name
    @print: "Enter rating"
    @read rating
    @print: "Please enter additional info for the movie"
    @read info

```
        m{rating, name, info}
        [ui.sl].insert(sl, m)
end-subalgorithm//Complexity:
        //BC: θ(1)
        //AV: θ(log2n)
        //WC: θ(n)
        //Overall complexity: O(n)
```

subalgorithm remove(ui) is:

```
        @print: "Please enter the position of the movie you want to delete: "
        @read position
        m ← [ui.sl].getByPosition(sl, position -1)
        [ui.sl].remove(sl, m)
end-subalgorithm
//Complexity:
        //BC: θ(1)
        //AV: θ(log2n)
        //WC: θ(n)
        //Overall complexity: O(n)
```

subalgorithm remove_all(ui) is:

```
        @print: "Please insert the position"
        @read position
        size ← [ui.sl].size(sl) - position
        while s > 0 execute
                [ui.sl].remove([ui.sl].getByPosition(sl, position))
                size ← size -1
        end-while
end-subalgorithm
//Complexity:
        //BC: θ(1)
        //AV: θ(n*log2n)
        //WC: O(n^2)
        //Overall complexity: O(n^2)
```

subalgorithm search(ui) is:

```
        @print: "Please insert movie rating"
        @read rating
        dummy_movie{rating, "", "NOTHING"}
        if [ui.sl].search(sl, dummy_movie) then
                @print: "We have a movie with the given rating"
        else
```

```
            @print: “We do not have a movie with the given rating”
        end-if
end-subalgorithm
//Complexity:
        //BC: θ(1)
        //AV: θ(log2n)
        //WC: θ(n)
        //Overall complexity: O(n)
```

subalgorithm print(ui) is:

```
        it ← [ui.sl].iterator(sl)
        position ← 0
        while it.valid(it) execute
            position ← position +1
            @print: “Pos: “, position, “ “
            @print: [it.getCurrent(it)].print(sl)
            it.next(it)
        end-while
end-subalgorithm
//Complexity: θ(n)
```

subalgorithm get_best_movie (ui) is:

```
        @print [[ui.sl].getByPosition(sl, 0)].print_extended(sl)
end-subalgorithm
//Complexity: θ(1)
```

subalgorithm get_movie_by_index(ui) is:

```
        @print: “Please insert the position”\
        @read position
        @print: [[ui.sl].getByPosition(sl, position - 1)].print_extended(sl)
end-subalgorithm
//Complexity:
        //BC: θ(1)
        //AV: θ(log2n)
        //WC: θ(n)
        //Overall complexity: O(n)
```

In the main file:

```
        function relation_descending(c1, c2) is:
            if c1 ≥ c2 then
```

```
            relation_descending ← true
        else
            relation_descending ← false
        end-if
    end-function

    function main() is:
        tests()
        user_interface(relation_descending)
        [user_interface].config(user_interface)
        [user_interface].run(user_interface)
    end-function
```

Tests

```
bool test_relation(TComp c1, TComp c2) {
    if (c1 <= c2) {
        return true;
    }
    else {
        return false;
    }
}

Movie create_test_movie(int rating)
{
    Movie m{rating, "", "" };
    return m;
}

void test_add() {
    SortedList l{ test_relation };
    assert(l.isEmpty());
    for (int i = 0; i < 10; i++)
    {
        l.add(create_test_movie(i));
        l.add(create_test_movie(-i));
    }
    assert(l.size() == 20);
}
```

```
void test_remove() {
	SortedList l { test_relation };
	for (int i = 1; i < 21; i++)
	{
		l.add(create_test_movie(i));
		l.add(create_test_movie(-i));
	}
	assert(l.size() == 40);

	for (int i = 11; i < 21; i++)
	{
		assert(l.remove(create_test_movie(i)) == true);
		assert(l.remove(create_test_movie(-i)) == true);
		assert(l.remove(create_test_movie(i)) == false);


	}

	assert(l.size() == 20);


	for (int i = 1; i < 11; i++)
	{
		assert(l.remove(create_test_movie(i)) == true);
		assert(l.remove(create_test_movie(-i)) == true);
		assert(l.remove(create_test_movie(i)) == false);
	}
	assert(l.size() == 0);


	for (int i = -100; i < 100; i++)
		assert(l.remove(create_test_movie(i)) == false);

}

void test_iterator() {
	srand(time(0));
	SortedList l{ test_relation };
	for (int i = 0; i < 100; i++)
		l.add(create_test_movie(rand() % 1000));
	int counter = 0;
```

```
	ListIterator it = l.iterator();

	while (it.valid())
	{
		counter++;
		it.getCurrent();
		it.next();
	}
	assert(!it.valid());
	it.first();
	assert(it.valid());
	assert(counter == 100);


}

void tests() {
	test_add();
	test_remove();
	test_iterator();
}
```

*Note: I added some things to the problem, compared to the project stage, to better emphasize the usage of positions.