ADT SortedList – implementation on a doubly linked list on an array

(Napradean Andrei Group 812 project nr.24)

A sorted list (or ordered list) is a list in which the elements from the nodes are in a specific order, given by a relation.

This time I have to implement it on a doubly linked list on an array (without pointers)

Drawing:

elems	**20**	**21**	**11**	**42**	**25**	**24**	**30**
next	**2**	**6**	**1**	**-1**	**7**	**5**	**4**
prev	**3**	**1**	**-1**	**7**	**6**	**2**	**5**

Head: 3
Relation: <
Order of elements: 11,20,21,24,25,30,42;

ADT representation:

SDLLANode:
info: TElem
next: Integer
prev: Integer

SDLLA:
nodes: SDLLANode[]
cap: Integer
size:Integer
head: Integer
tail:Integer
firstEmpty: Integer
Relation: bool

SDLLAIterator:
list: SDLLA
currentElement: Integer

Operations:

Init:

```
subalgorithm init (sdlla, rel) is:
        //pre: rel is a relation
         //post: sdlla is SDLLA
         sdlla.head ← -1
         sdlla.tail ← -1
         sdlla.size ← NIL
         sdlla.firstEmpty ←0
         sdlla.cap ← 100
         sdlal.rel ← rel
        Node<T> elems[100]
        for int i <- 0,size
               elems[i].next <- i + 1
               elems[i].prev <- i - 1
        end-for
               elems[0].prev <- -1;
               elems[99].next <- -1
end-subalgorithm
```

First:

```
subalgorithm first() is:
       if size!=NILL
              return 0
end-subalgorithm
```

Last:

```
subalgorithm last() is:
       if size!=NILL
              return size-1
end-subalgorithm
```

Valid:

```
subalgorithm valid( pos)is:
              //pre: pos is a position
              //post: true or false if pos is valid

              if pos >= 0 && pos < size
                     return True
              else
                     return False
end-subalgorithm
```

Next:

```
Subalgorithm next(pos)is:
              if valid(pos + 1)
                     return pos + 1
```

end-subalgorithm

Previous:

```
Subalgorithm prev(pos)is:
              if valid(pos - 1)
                     return pos - 1
```

end-subalgorithm

Get Element on position:

```
       subalgorithm getElement(e, p)is:
```
//pre: e is a TElem p is a position

//post: returns in e the elem

```
              current <- head
              if valid(p)
                     for i <- 0,p
                            current <- elems[current].next
                     end-for
                     e <- elems[current].info
              end-if
       end-subalgorithm
```

Get Position:

```
       subalgorithm getPos(e)is:
```
//pre: e is a TElem

//post: returns the index of the position

```
              current <- head
              for i <- 0,size
                     if elems[current].info = e
                            return i
                     end-if
                     current <- elems[current].next
              end-for
       end-subalgorithm
```

Insert: Complexity O(n)

```
subalgorithm insert(e) is:
	//pre: e- TElem
	//post: returns nothing modifies the list
		if size = cap
			throw runtime_error "List is full"
		end-if
		elems[firstEmpty].info <- e
		if size=NILL
			elems[firstEmpty].prev <-1
			head <- firstEmpty
			tail <- head
			firstEmpty <- elems[firstEmpty].next
			elems[firstEmpty].prev <- -1
			elems[head].next = -1
			size<-size+1
			return
		end-if
		if rel(e, elems[head].info)
			aux <- elems[firstEmpty].next
			elems[firstEmpty].prev <- -1
			elems[firstEmpty].next <- head
			elems[head].prev <- firstEmpty
			head <- firstEmpty
			firstEmpty <- aux
			if aux != -1
				elems[aux].prev <- -1
			end-if
		end-if
		else
			current <- head
			while current != -1 && rel(elems[current].info, e)
				current <- elems[current].next
			end-while
			if current = -1
				elems[firstEmpty].prev <- tail
				elems[tail].next <- firstEmpty
				tail <- firstEmpty
				firstEmpty <- elems[firstEmpty].next
				if firstEmpty != -1
					elems[firstEmpty].prev <- -1
				end-if
				elems[tail].next <- -1
			end-if
			else
				aux <- elems[firstEmpty].next
				elems[firstEmpty].next <- current
				elems[firstEmpty].prev <- elems[current].prev
				elems[elems[current].prev].next <- firstEmpty
				elems[current].prev <- firstEmpty
				firstEmpty <- aux
				if aux != -1
					elems[aux].prev <- -1
				end-if
			end-if
		end-if
		size<-size+1
```

```
end-subalgorithm
```

Remove: Complexity O(n)

```
subalgorithm remove(p, e) is:
//pre: p is the position e is TElem
//post: returns nothing modifies the list

        current <- head
        if valid(p)
                for i <-0,p
                        current <- elems[current].next
                end-for
                e <- elems[current].info
                if p = 0
                        aux <-firstEmpty
                        head <- elems[head].next
                        firstEmpty <- elems[head].prev
                        elems[firstEmpty].next <- aux
                        elems[firstEmpty].prev <- -1
                        elems[head].prev <- -1
                end-if
                else if p = size - 1
                        aux <- firstEmpty
                        firstEmpty <- tail
                        tail <- elems[tail].prev
                        elems[firstEmpty].next <- aux
                        elems[firstEmpty].prev <- -1
                        elems[tail].next <- -1
                        elems[aux].prev <- firstEmpty
                end-if
                else
                        aux <- firstEmpty
                        elems[elems[current].prev].next <- elems[current].next
                        elems[elems[current].next].prev <- elems[current].prev
                        firstEmpty <- current
                        elems[current].next <- aux
                        elems[current].prev <- -1
                        elems[aux].prev <- current
                end-else
        end-if
        size<-size-1
end-subalgorithm
```

Search

```
Subalgorithm search(e)is:
```
//pre: e is a TElem

//post: returns true or false wether the elem was found or not

```
        current <- head;
        while current != -1
                if elems[current].info = e
                        return true
                end-if
                current <- elems[current].next
        end-while
        return false
end-subalgorithm
```

Complexity O(n)
Best case: O(1) element is in fist position we don't have to loop more than once => o(1)
Worst case: O(n) element is in last position we have to loop alot => o(n)
Average case: O(n)

Operations for iterator:

Init:

```
subalgorithm init(it, sdlla) is:
        //it is an Iterator, da is a sdlla
        it.sdlla ← sdlla
        it.current ← 1
 end-subalgorithm
```

Complexity: Θ(1)

Get Current:

```
subalgorithm getCurrent(it, e) is:
          e ← it.sdlla.elems[it.current]
 end-subalgorithm
```

Complexity O(1)

Get Next:

```
subalgorithm next(it) is:

    it.current ← it.current + 1

end-subalgorithm
```

Complexity O(1)

Valid:

```
function valid(it) is:
        if it.current <= it.sdlla.size then
                valid ← True
        else valid ← False
        end-if
 end-function
```

Complexity O(1)

Tests (from visual studio):

```
void tests()
{
	List<int> l;
	// for insert function
	l.insert(2);
	assert(l.size() == 1);

	// for search function
	assert(l.search(2)) == true);

	// for valid function
	assert(l.valid(0)) == true);

	// for first function
	assert(l.first() == 0);

	// for last function
	assert(l.last() == 0);

	// for remove function
	l.remove(0,2);

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

	l.insert(2);
	l.insert(3);
	l.insert(1);

	// for next function
	assert(l.next(1) == 2);

	// for prev function
	assert(l.prev(1) == 0);

	// for getPosition function
	assert(l.getPosition(2) == 1);}
```

Problem :

At a contest the students are ordered by their grades (ascendently). Find the student with grade equal with a given value, you can add students or delete stundents and the search would work after each operation.

Statement:

This problem was chosen for this ADT because I can add elements to the array on the right position (sorted) using next and previous links and when I find the position of the student with that requested grade I just have to print it.

Implementation in pseudocode:

```
template <typename T>
```
Struct Node:

T info

Int prev

Int next

End-struct

```
template <typename T>
class List

private:
	int cap <- 100
	int size <- 0
	int firstEmpty <- 0
	int tail <- -1
	int head <- -1
	Node<T> elems[100]
	bool(*r)(T, T)

public:
subalgorithm List(bool(*cmp)(T, T) = [](T a, T b) {return a < b; }) : r{ cmp } is:

	for  i <- 0,100
		elems[i].next <- i + 1;
		elems[i].prev <- i - 1;
	end-for
	elems[99].next <- -1
end-subalgorithm
```

subalgorithm first() is:

```
	if size!=NILL
		return 0
end-subalgorithm
```

subalgorithm last() is:

```
	if size!=NILL
		return size-1
end-subalgorithm

subalgorithm valid( pos)is:
		if pos >= 0 && pos < size
			return True
		else
			return False
end-subalgorithm
```

```
Subalgorithm next(pos)is:
              if valid(pos + 1)
                     return pos + 1
```

end-subalgorithm

```
Subalgorithm prev(pos)is:
              if valid(pos - 1)
                     return pos - 1
```

end-subalgorithm

```
subalgorithm getElement(e, p)is:
              current <- head
              if valid(p)
                     for i <- 0,p
                            current <- elems[current].next
                     end-for
                     e <- elems[current].info
              end-if
end-subalgorithm
```

```
subalgorithm getPos(e,)is:
              current <- head
              for i <- 0,size
                     if elems[current].info = e
                            return i
                     end-if
                     current <- elems[current].next
              end-for
end-subalgorithm
```

```
subalgorithm insert(e) is:
       //pre: e- TElem
       //post: returns nothing modifies the list
              if size = cap
                     throw runtime_error "List is full"
              end-if
              elems[firstEmpty].info <- e
              if size=NILL
                     elems[firstEmpty].prev <-1
                     head <- firstEmpty
                     tail <- head
                     firstEmpty <- elems[firstEmpty].next
                     elems[firstEmpty].prev <- -1
```

```
		elems[head].next = -1
		size<-size+1
		return
	end-if
	if rel(e, elems[head].info)
		aux <- elems[firstEmpty].next
		elems[firstEmpty].prev <- -1
		elems[firstEmpty].next <- head
		elems[head].prev <- firstEmpty
		head <- firstEmpty
		firstEmpty <- aux
		if aux != -1
			elems[aux].prev <- -1
		end-if
	end-if
	else
		current <- head
		while current != -1 && rel(elems[current].info, e)
			current <- elems[current].next
		end-while
		if current = -1
			elems[firstEmpty].prev <- tail
			elems[tail].next <- firstEmpty
			tail <- firstEmpty
			firstEmpty <- elems[firstEmpty].next
			if firstEmpty != -1
				elems[firstEmpty].prev <- -1
			end-if
			elems[tail].next <- -1
		end-if
		else
			aux <- elems[firstEmpty].next
			elems[firstEmpty].next <- current
			elems[firstEmpty].prev <- elems[current].prev
			elems[elems[current].prev].next <- firstEmpty
			elems[current].prev <- firstEmpty
			firstEmpty <- aux
			if aux != -1
				elems[aux].prev <- -1
			end-if
		end-if
	end-if
	size<-size+1
end-subalgorithm



subalgorithm remove(p, e) is:
//pre: p is the position e is TElem
//post: returns nothing modifies the list

	current <- head
	if valid(p)
		for i <-0,p
			current <- elems[current].next
		end-for
		e <- elems[current].info
		if p = 0
```

```
                        aux <-firstEmpty
                        head <- elems[head].next
                        firstEmpty <- elems[head].prev
                        elems[firstEmpty].next <- aux
                        elems[firstEmpty].prev <- -1
                        elems[head].prev <- -1
                    end-if
                    else if p = size - 1
                        aux <- firstEmpty
                        firstEmpty <- tail
                        tail <- elems[tail].prev
                        elems[firstEmpty].next <- aux
                        elems[firstEmpty].prev <- -1
                        elems[tail].next <- -1
                        elems[aux].prev <- firstEmpty
                    end-if
                    else
                        aux <- firstEmpty
                        elems[elems[current].prev].next <- elems[current].next
                        elems[elems[current].next].prev <- elems[current].prev
                        firstEmpty <- current
                        elems[current].next <- aux
                        elems[current].prev <- -1
                        elems[aux].prev <- current
                    end-else
            end-if
            size<-size-1
        end-subalgorithm

Subalgorithm search(e)is:
            current <- head;
            while current != -1
                if elems[current].info = e
                    return true
                end-if
                current <- elems[current].next
            end-while
            return false
end-subalgorithm

Subalgorithm print() is
            print "The current list is: "
            current <- head;
            while current != -1
                print elems[current].info
                current <- elems[current].next
            end-while


end-subalgorithm


        ~List()
End-class

int main()
```

```
        List<int> l

        int a,b,l,value,nr,p

        char c

    print  "Please input the instructions (enter 'E' to exit): \n A to add to list\n D
    to delete \n T to show all\n S to search the first x students with grades grater
    than value"
    l.insert(5)
    l.insert(2)
    l.insert(3)
    l.insert(7)
    l.insert(10)
    l.insert(10)
    l.insert(9)
    l.insert(7)

    do
           scan c
           switch(c)
           case ’A’: print “give element to be added: “
                  scan a
                  l.insert(a)
                  @break
           case ’D’: print “give element to be deleted: ”
                  scan a
                  print “give position: ”
                  scan p

                  remove(p,a)
                  @break
           Case ‘S’: print “give value: “
                  Scan value

                  If search(value)

                         Print”student found”
                  Else
                         Print “student not found”
                  @break
           Case ‘T’:            Print()
                  @break
           Case ‘E’:
                  Print “exit”
                  @break
           Default: print” invalid input”
    End-switch
End-do while c!=’E’
Return 0
End-main
```