Part I

Requirements

No.	ADT	Representation 1	Representation 2
3	List with current element	dynamic vector	singly linked list

Data Abstraction

In the following, we provide the **interface** / **abstract class** corresponding to the given ADT.

ListWithCurrentADT.h

```
template <typename TE>
class ListWithCurrentADT {
public:
	// default constructor
	ListWithCurrentADT() {}

	// copy constructor
	ListWithCurrentADT(ListWithCurrentADT<TE>& l) {}

	// virtual destructor
	virtual ~ListWithCurrentADT() {}


	/*
	Get the current element from the list
	@pre: the current element is valid
		the list is not empty
	@post: e:TE is the element pointed by current
	@return: the current element
	*/
	virtual TE getCurrent() = 0;

	/*
	Modify the current element from the list
	@pre: the current element is valid
		the list is not empty
	@post: the current element is modified to e:TE
	*/
	virtual void setCurrent(const TE e) = 0;
```

```cpp
/*
Remove the current element from the list
@pre: the current element is valid
         the list is not empty
@post: the current element is removed from the list
*/
virtual void delCurrent() = 0;

/*
Change the current to the first element from the list
@pre: the list is not empty
@post: current points to the first element in the list
*/
virtual void first() = 0;

/*
Change the current to the last element from the list
@pre: the list is not empty
@post: current points to the last element in the list
*/
virtual void last() = 0;

/*
Change the current to the next element from the list
@pre: the list is not empty
         current is not the last element in the list
@post: current points to the next element in the list
*/
virtual void next() = 0;

/*
Checks if the current element has a next element or not
@pre: the list is not empty
@post: -
@return: true, if current is not on the last position
              false, otherwise
*/
virtual bool hasNext() = 0;

/*
Change the current to the previous element from the list
@pre: the list is not empty
         current is not the first element in the list
@post: current points to the previous element in the list
*/
virtual void prev() = 0;

/*
Checks if the current element has a previous element or not
@pre: the list is not empty
@post: -
@return: true, if current is not on the first position
              false, otherwise
*/
virtual bool hasPrev() = 0;
```

```
/*
Check if the current element from the list is valid or not
@pre: -
@post:
@return: true, if current is valid (i.e. it can be retrieved by getCurrent())
              false, otherwise
*/
virtual bool valid() = 0;

/*
Check if a certain element is contained in the list or not
@pre: e:TE
@post: -
@return: true, if e:TE is in the list
              false, otherwise
*/
virtual bool exists(const TE e) = 0;

/*
Check if the list is empty or not
@pre: -
@post: -
@return: true, if the list is empty (i.e. it does not contain any element)
           false, otherwise
*/
virtual bool isEmpty() = 0;

/*
Get the number of elements contained in the list
@pre: -
@post: -
@return: returns the size of the list (i.e. the number of elements)
*/
virtual int getSize() = 0;

/*
Add a new element in the front of the list
@pre: e:TE
@post: e:TE is added on the first position in the list
          current is changed to the element just added
*/
virtual void addFront(const TE e) = 0;

/*
Add a new element in the back of the list
@pre: e:TE
@post: e:TE is added on the last position in the list
          current is changed to the element just added
*/
virtual void addBack(const TE e) = 0;

/*
Add a new element before the current element in the list
@pre: e:TE
@post: e:TE is added in the list just before the current element
          current is changed to the element just added
*/
virtual void addBefore(const TE e) = 0;
```

```
	/*
	Add a new element after the current element in the list
	@pre: e:TE
	@post: e:TE is added in the list just after the current element
	         current is changed to the element just added
	*/
	virtual void addAfter(const TE e) = 0;
};
```

Representation, Operation Design & Implementation

DS design (representation)

```
template <typename TE>
class DynVectLWCE : public ListWithCurrentADT<TE> {
private:
	DynamicVector<TE> elements;
	int current;
	int size;
public:
	//...
}
```

DynVectLWCE

```
    elements : DynamicVector of TE
    current  : Integer
    size     : Integer
```

```
template <typename TE>
class SLListLWCE : public ListWithCurrentADT<TE> {
private:
	SLList<TE> elements;
	SLNode<TE>* current;
	int size;
public:
	//...
}
```

SLListLWCE

```
    elements : SLList of TE
    current  : ↑SLNode of TE
    size     : Integer
```

Operation design

LWCE over Dynamic Vector	LWCE over Singly Linked List
Algorithm **getCurrent**():TE $O(1)$ **return** elements[current]	Algorithm **getCurrent**():TE $O(1)$ **return** current.data
Algorithm **setCurrent**(e:TE) $O(1)$ elements[current] ≔ e	Algorithm **setCurrent**(e:TE) $O(1)$ current.data ≔ e
Algorithm **delCurrent**() $O(n)$ elements.*del*(current) size ≔ size – 1 **if** current = size **then** current ≔ 0 **end_if**	Algorithm **delCurrent**() $O(n)$ it(current) : Iterator *for* SLList *of* TE elements.*eraseHere*(it) size ≔ size – 1 **if** current = NIL **then** current ≔ elements.head **end_if**
Algorithm **first**() $O(1)$ current ≔ 0	Algorithm **first**() $O(1)$ current ≔ elements.head

Algorithm **last**() $O(1)$
current ≔ elements.size – 1
	Algorithm **last**() $O(n)$
	if current = NIL **then**
	current ≔ elements.head
	end_if
	while current.next = NIL **do**
	current ≔ current.next
	done

Algorithm **next**() $O(1)$
  current ≔ current + 1
	Algorithm **next**() $O(1)$
	  current ≔ current.next

Algorithm **prev**() $O(1)$
  current ≔ current - 1
	Algorithm **prev**() $O(n)$
	  nod : ↑SLNode *of* TE
	  nod ≔ elements.head
	  while nod.next ≠ current **do**
	    nod ≔ nod.next
	  done
	  current = nod

Algorithm **valid**():bool $O(1)$
  if current ∈ [0, size) ∩ ℕ **then**
    return true
  end_if
  return false
	Algorithm **valid**():bool $O(1)$
	  if current ≠ NIL **then**
	    return true
	  end_if
	  return false

Algorithm **exists**(e:TE):bool $O(n)$
  for i **from** 0 **to** size - 1 **do**
    if elements[i] = e **then**
      return true
    end_if
  done
  return false
	Algorithm **exists**(e:TE):bool $O(n)$
	  @for it:Iterator **from** begin **to** end **do**
	    if it.getVal() = e **then**
	      return true
	    end_if
	  end_@for
	  return false

Algorithm **isEmpty**():bool $O(1)$
  if size = 0 **then**
    return true
  end_if
  return false
	Algorithm **isEmpty**():bool $O(1)$
	  if size = 0 **then**
	    return true
	  end_if
	  return false

Algorithm **addFront**(e:TE) $O(n)$
  elements.*insert*(0, e)
  size ≔ size + 1
  current ≔ 0
	Algorithm **addFront**(e:TE) $O(1)$
	  elements.*insertFront*(e)
	  size ≔ size + 1
	  current ≔ elements.head

Algorithm **addBack**(e:TE) $O(1)$
  elements.*push_back*(e)
  size ≔ size + 1
  current ≔ size − 1
	Algorithm **addBack**(e:TE) $O(n)$
	  elements.*insertBack*(e)
	  size ≔ size + 1
	  if current = NIL **then**
	    current ≔ elements.head
	  end_if
	  while current.next = NIL **do**
	    current ≔ current.next
	  done

Algorithm **addBefore**(e:TE) $O(n)$
  elements.*insert*(current, e)
  size ≔ size + 1
	Algorithm **addBefore**(e:TE) $O(1)$
	  it(current) : Iterator *for* SLList *of* TE
	  elements.*insertHere*(it, e)
	  size ≔ size + 1

Algorithm **addAfter**(e:TE) $O(n)$
  if current = size - 1 **then**
    elements.*push_back*(e)
  else
    elements.*insert*(current + 1, e)
  end_if
  size ≔ size + 1
  current ≔ current + 1
	Algorithm **addAfter**(e:TE) $O(1)$
	  it(current) : Iterator *for* SLList *of* TE
	  elements.*insertAfter*(it, e)
	  size ≔ size + 1
	  current ≔ current.next

Some of the most complex subalgorithms are presented in the following:

LWCE over Dynamic Vector	LWCE over Singly Linked List

Subalgorithm **insert**(vect:DynamicVector *of* TE, pos:Integer, e:TE) $O(n)$
  if vect.size = vect.capacity **then**
    vect.*resize()*
  end_if
  for i **from** size−1 **to** pos ***decreasing*** **do**
    vect.elems[i+1] ≔ vect.elems[i]
  done
  vect.size ≔ vect.size + 1
  vect.elems[i] ≔ e
	Subalgorithm **insertHere**(it:Iterator *for* SLList *of* TE, e:TE) $O(1)$
	  nod : ↑SLNode *of* TE
	  nod ≔ **new**(SLNode *of* TE)
	  nod.data ≔ it.current.data
	  it.current.data ≔ e
	  nod.next ≔ it.current.next
	  it.current.next ≔ nod

Subalgorithm **del**(vect:DynamicVector *of* TE, pos:Integer) $O(n)$
  e:TE
  e ≔ vect.elems[pos]
  for i **from** pos **to** size−1 **do**
    vect.elems[i] ≔ vect.elems[i+1]
  done
  vect.size ≔ vect.size − 1
	Subalgorithm **eraseHere**(it:Iterator *for* SLList *of* TE) $O(n)$
	  nod : ↑SLNode *of* TE
	  nod ≔ it.current
	  while nod.next ≠ NIL **do**
	    nod.data ≔ nod.next.data
	    if nod.next.next = NIL **then**
	      free(nod.next)
	      nod.next ≔ NIL
	      break
	    enf_if
	  done

Complexity of **push_back** is $O(1)$	Complexity of **insertFront** is $O(1)$ Complexity of **insertBack** is $O(n)$ Complexity of **insertAfter** is $O(1)$

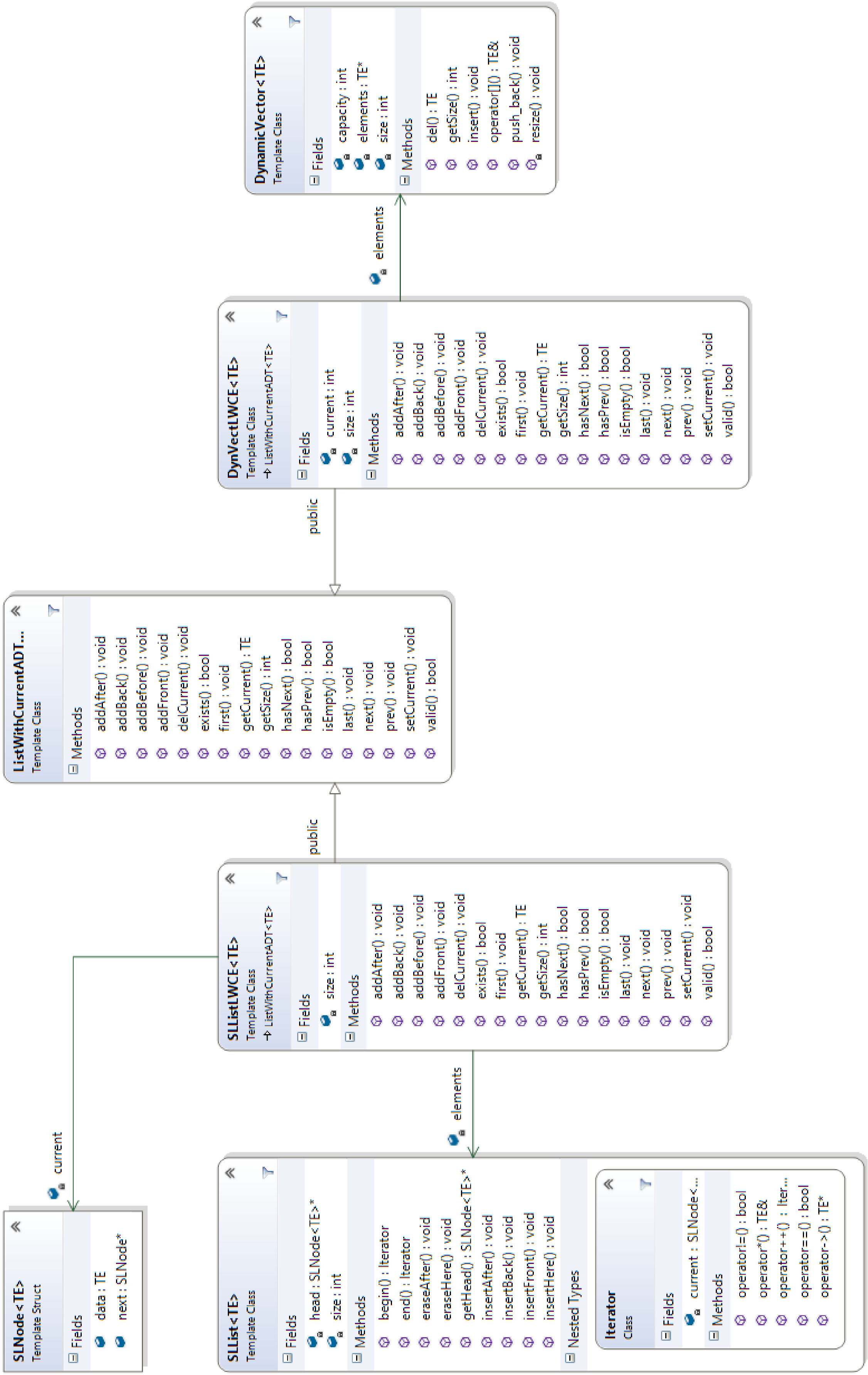
Implementation

DynVectLWCE.h

SLListLWCE.h

Unit testing

Tests.h


DynamicVector<TE>
Template Class
Fields
capacity : int
elements : TE*
size : int
Methods
del() : TE
getSize() : int
insert() : void
operator[]() : TE&
push_back() : void
resize() : void
elements
DynVectLWCE<TE>
Template Class
→ ListWithCurrentADT<TE>
Fields
current : int
size : int
Methods
addAfter() : void
addBack() : void
addBefore() : void
addFront() : void
delCurrent() : void
exists() : bool
first() : void
getCurrent() : TE
getSize() : int
hasNext() : bool
hasPrev() : bool
isEmpty() : bool
last() : void
next() : void
prev() : void
setCurrent() : void
valid() : bool
public
ListWithCurrentADT...
Template Class
Methods
addAfter() : void
addBack() : void
addBefore() : void
addFront() : void
delCurrent() : void
exists() : bool
first() : void
getCurrent() : TE
getSize() : int
hasNext() : bool
hasPrev() : bool
isEmpty() : bool
last() : void
next() : void
prev() : void
setCurrent() : void
valid() : bool
public
SLListLWCE<TE>
Template Class
→ ListWithCurrentADT<TE>
Fields
size : int
Methods
addAfter() : void
addBack() : void
addBefore() : void
addFront() : void
delCurrent() : void
exists() : bool
first() : void
getCurrent() : TE
getSize() : int
hasNext() : bool
hasPrev() : bool
isEmpty() : bool
last() : void
next() : void
prev() : void
setCurrent() : void
valid() : bool
current
elements
SLNode<TE>
Template Struct
Fields
data : TE
next : SLNode*
SLList<TE>
Template Class
Fields
head : SLNode<TE>*
size : int
Methods
begin() : Iterator
end() : Iterator
eraseAfter() : void
eraseHere() : void
getHead() : SLNode<TE>*
insertAfter() : void
insertBack() : void
insertFront() : void
insertHere() : void
Nested Types
Iterator
Class
Fields
current : SLNode<...
Methods
operator!=() : bool
operator*() : TE&
operator++() : Iter...
operator==() : bool
operator->() : TE*

Part II

Requirements

2. Given n balloons, determine the smallest number of arrows to break all the balloons. It is considered that the arrows are launched vertically. *(Solution should also present the arrow positions.)*

Problem: input/output & test data

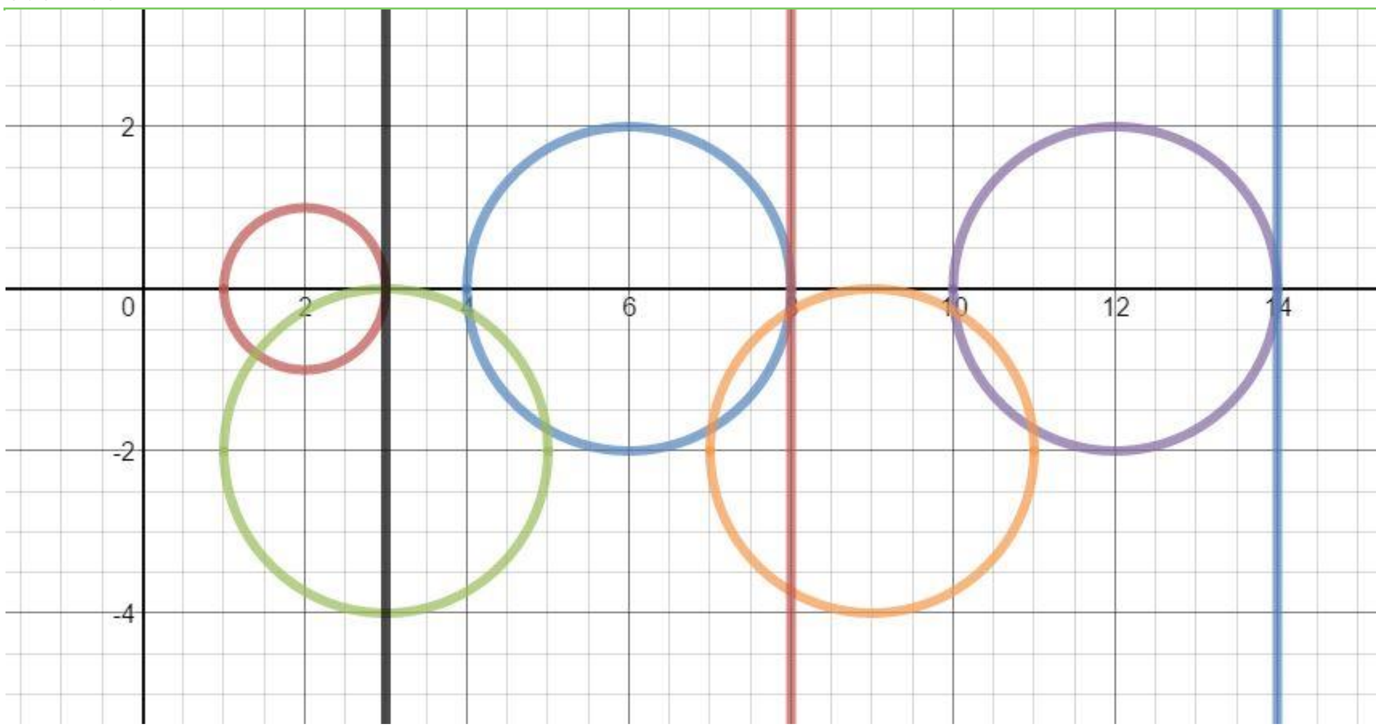
For simplicity, we consider the balloons to be spheres.

Input: each line contains two floating-point with 2 decimals numbers representing `xCoord` and `radius` (that is, the x-coordinate of the center of the balloon and its radius), separated by space.

Output: for every arrow in the solution, there is a line in the output file. The line has the following prototype: `Arrow no. <n> found at coord. <coord>`. The name of the output file is `SOL_<inputF>`.

Example:

ball1.txt (input)	**SOL_ball1.txt (output)**
2.0 1.0 3.0 2.0 9.0 2.0 12.0 2.0 6.0 2.0	Arrow no. 1 found at coord. 3 Arrow no. 2 found at coord. 8 Arrow no. 3 found at coord. 14



Test sets:

Input	Ouput
2.0 1.0 3.0 2.0 9.0 2.0 12.0 2.0 6.0 2.0	Arrow no. 1 found at coord. 3 Arrow no. 2 found at coord. 8 Arrow no. 3 found at coord. 14
4.5 3.5 8.0 4.0 27.5 1.5 16.5 6.5 16.5 3.5 6.0 1.0	Arrow no. 1 found at coord. 7 Arrow no. 2 found at coord. 20 Arrow no. 3 found at coord. 29
10.00 1.00 12.00 12.00 9.00 1.00 14.00 13.00 15.00 1.00 19.00 11.00 9.00 9.00 18.00 18.00 12.00 1.00 14.00 2.00	Arrow no. 1 found at coord. 10 Arrow no. 2 found at coord. 13 Arrow no. 3 found at coord. 16

Problem solving (application)

Idea for solving the problem

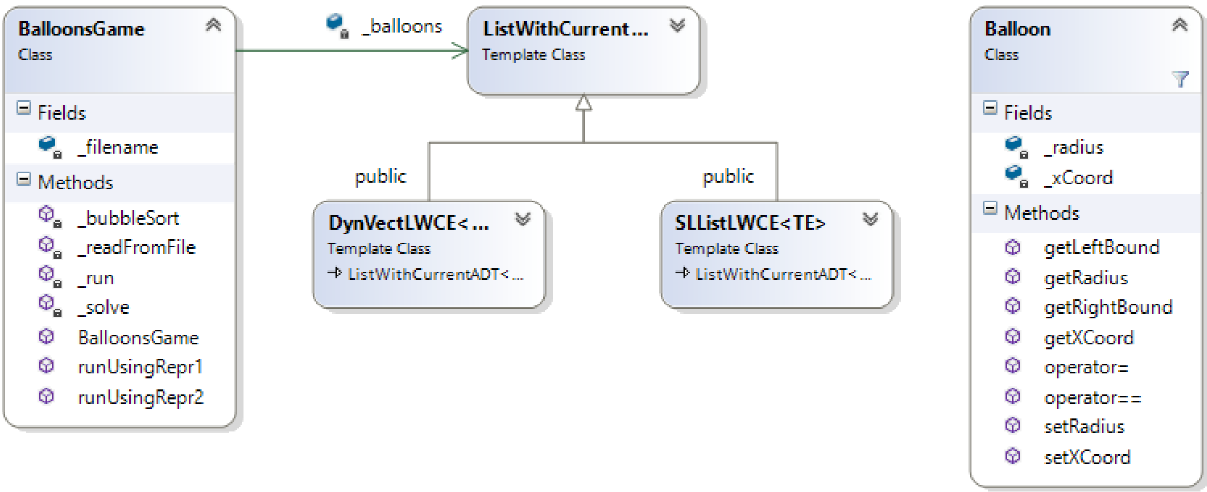
The best method to solve such a problem (which can be reduced to the *Activity Selection Problem* [Th. H. Cormen et al., *Introduction to Algorithms*, MIT, 2009]) is a **greedy approach**.

First, we sort the balloons with respect to the right-most bound (that is $x_b + r_b$). Then, from left to right, for each balloon that was not shot yet, we shoot an arrow exactly at the right-most bound coordinate of the ballon, updatind the "arrow status" of each balloon found at that coordinate.

Proof of correctness

Let $S = \{b_1, b_2, \dots, b_n\}$ be the set of balloons sorted by their right-most bound. The greedy choice is to always pick b_1 first. Let S' be the optimal solution. Suppose that b_1 is not the first element in the optimal solution. Let b_k be the first balloon in S'. Then, $\exists\ A = (S' \setminus \{b_k\}) \cup \{b_1\}$. Since $b_k \neq b_1 \Longrightarrow RB(b_k) \geq RB(b_1)$. So, b_1 can be put before b_k in the solution. Thus, we can exchange b_1 and b_k in the optimal solution. By induction, assuming that the optimal solution and the greedy one "looks alike" up to a point, we can exchange in the optimal solution the first balloon that does not appear on the same position in the greedy solution with the balloon found at that position in the greedy solution. By repeating this procedure, we end up with $S = S'$. Thus S is optimal and the algorithm is correct.

```
Algorithm solve(list:List of Balloon)
    @sort(list) w.r.t. the right-most bound of the elements
    S : Set
    nrArrows:Integer
    lastArrow:Double
    nrArrows ≔ 0
    lastArrow ≔ −∞
    for b:Balloon in list in increasing order do
        if lastArrow < b.LB then
            lastArrow ≔ b.RB
            nrArrows ≔ nrArrows + 1
            S ≔ S ∪{lastArrow}
        end_if
    done
    return S
```



Execution time

LWCE over Dynamic Vector
<u>Size of input</u> = **100**
Reading from file executed in 0.001 s..
Sorting executed in 0.007 s..
Solving the problem executed in 0.002 s..

<u>Size of input</u> = **1000**
Reading from file executed in 0.008 s..
Sorting executed in 0.842 s..
Solving the problem executed in 0.006 s..

<u>Size of input</u> = **10k**
Reading from file executed in 0.11 s..
Sorting executed in 89.509 s..
Solving the problem executed in 0.052 s..

LWCE over Singly Linked List
<u>Size of input</u> = **100**
Reading from file executed in 0.001 s..
Sorting executed in 0.007 s..
Solving the problem executed in 0.001 s..

<u>Size of input</u> = **1000**
Reading from file executed in 0.012 s..
Sorting executed in 1.581 s..
Solving the problem executed in 0.005 s..

<u>Size of input</u> = **10k**
Reading from file executed in 0.647 s..
Sorting executed in 1570.7 s..
Solving the problem executed in 0.088 s..

Part III

Which DSs are best for given ADT?

The Dynamic Vector is better if we want to add elements just to the back of the collection, while the Singly Linked List is better if we want to add them just to the front because these operations execute in linear time.

For iterating through elements, both DSs are good for forward iterating. For backward iterating, the singly linked list is not so good because the operation has complexity O(n).

In the case of this problem, in order to perform the sorting part, a bidirectional "iterator" is desired, thus the dynamic vector would be a good choice.

For the list with current element ADT, the dynamic vector is clearly better than the singly linked list (see time complexities table), but a doubly linked list would be much better because it will provide linear time execution for most operations (which is a plus), but it needs more memory space (drawback).

Which ADTs are best to be used to solve the given problem?

One important step in this problem solving is the sorting. It would be nice if we can insert elements to the collection sorted. Also, a linear time forward iterator will be helpful.

The best ADTs to solve this problem can be: sorted multiset or binary search tree (for in-order traversal).

What did I learn from this project?

- how to specify, represent and implement data structures
- how to implement an iterator (at least for SL List)
- how to overload the ++ operator (prefix and postfix)
- how to use OOP concepts such as abstraction, inheritance and polymorphism
- how to measure the execution time in C++
- understood how an greedy algorithm works
- how to deal with such problems

Opening the project

Visual Studio 2015 is required. Open the file ***ProjectDSA.sln***.

Part1 contains files related to the first part of the project: ADT interface, the two representations, dynamic bector and singly linked list DSs, as well as tests.

Part2 contains files related to the second part of the project: *Balloon* class definition and *BalloonGame* class, for solving the problem.

main.cpp contains the main function of the program.