Documentation

List & Iterator RW (read-write) add/remove only in iterator (Java ListIterator style)

List
Domain: D[L<TElement>] = the domain of the list
Operations:

- create(list) – creates an empty list
    - Data: list
    - Result: list
    - Preconditions: -
    - Postconditions: list ∈ D[L<TElement>] , list = empty list
- destroy(list) – destroys a list
    - Data: list
    - Result: -
    - Preconditions: list ∈ D[L<TElement>]
    - Postconditions: the list is destroyed and the space is freed
- create_copy(list, src) – creates a copy of a list
    - Data: list, src
    - Result: list
    - Preconditions: src ∈ D[L<TElement>]
    - Postconditions: list ∈ D[L<TElement>] , list is a copy of src
- empty(list) – checks if the list is empty
    - Data: list
    - Result: True/False
    - Preconditions:
    - Postconditions: True if list is empty, False otherwise
- size(list) – returns the size of the list
    - Data: list
    - Result: sz
    - Preconditions: list ∈ D[L<TElement>]
    - Postconditions: sz ∈ {set of natural numbers}, sz = number of elements
- clear(list) – clears the list
    - Data: list
    - Result: new_list
    - Preconditions: list ∈ D[L<TElement>]
    - Postconditions: new_list ∈ D[L<TElement>] , new_list = empty list
- begin(list) – returns an iterator pointing before the first element
    - Data: list
    - Result: it
    - Preconditions: list ∈ D[L<TElement>]
    - Postconditions: it ∈ I, pointing before the first element

Iterator
Domain: I = {i | i is an iterator on a List with elements of type TElement}

- create(it, list) – creates an iterator it on List list
    - Data: it, list
    - Result: it
    - Preconditions: list ∈ D[L<TElement>]
    - Postconditions: it ∈ I, it points before the first element in list
- destroy(it) – destroys an iterator
    - Data: it
    - Result: -
    - Preconditions: it ∈ I
    - Postconditions: the iterator is destroyed and the space is freed
- create_copy(it, src) – creates a copy of an iterator
    - Data: it, src
    - Result: it
    - Preconditions: src ∈ I
    - Postconditions: it ∈ I , it is a copy of src
- has_next(it) – checks if the iterator has elements to the right
    - Data: it
    - Result: True/False
    - Preconditions: it ∈ I
    - Postconditions: True if the iterator has elements to the right, False otherwise
- has_previous(it) – checks if the iterator has elements to the left
    - Data: it
    - Result: True/False
    - Preconditions: it ∈ I
    - Postconditions: True if the iterator has elements to the left, False otherwise
- next(it) – returns the next element in the list and advances the iterator
    - Data: it
    - Result: it' – next iterator
    - Preconditions: it ∈ I
    - Postconditions: returns next TElement, iterator advances
- prev(it) – returns the prev element in the list and moves the iterator backwards
    - Data: it
    - Result: it' – prev iterator
    - Preconditions: it ∈ I
    - Postconditions: returns prev TElement, iterator moves backwards
- add(it, e) – adds an element in the list at the current position of the iterator
    - Data: it, e
    - Result: -
    - Preconditions:
    - Postconditions: the list pointed by the iterator contains now a new element
- set(it, e) – replaces the last element returned by next() or previous() with e
    - Data: it, e
    - Result: -
    - Preconditions: it ∈ I, e ∈ D[TElement]
    - Postconditions: the list pointed by the iterator is modified
- remove(it) – removes the last element returned by next() or previous()
    - Data: it
    - Result: -
    - Preconditions: it ∈ I
    - Postconditions: the list pointed by the iterator has one element less

List Interface

```
template<typename T>
class List {
    public:
        class ListIterator;

        List();
        ~List();
        List(const List<T>& that);
        List& operator=(const List<T>& that);

        bool empty() const;
        int size() const;
        void clear();

        ListIterator begin();
};
```

ListIterator Interface

```
class List<T>::ListIterator {
       public:
            ListIterator() {}
            ~ListIterator() {}
            ListIterator(DynamicVector<T>* (or DoublyLinkedList<T>*) c);
            ListIterator(const ListIterator& that);
            ListIterator& operator=(const ListIterator& that);

            bool has_next() const;
            bool has_previous() const;
            T next();
            T previous();

            void add(T element);
            void set(T element);
            void remove();
 };
```

1. Representation with Dynamic Vector

```
template<typename T>
class DynamicVector {
    public:
        DynamicVector();
        ~DynamicVector();
        DynamicVector(const DynamicVector<T>& that);
        DynamicVector& operator=(const DynamicVector<T>& that);

        int size() const;
        void add(T element, int index);
        void set(T element, int index);
        void remove(int index);
        void clear();
        T& operator[] (int index);

    private:
        int capacity;
        int dimension;
        T* elements;

        void resize();
};

template<typename T>
class ListDynamicVector {
      private:
            DynamicVector<T>* vector;
};
```

ListDynamicVector complexities

- constructor: O(1)
- destructor: O(n)
- copy constructor: O(n)
- empty: O(1)
- size: O(1)
- clear: O(n)
- begin: O(1)

ListIterator complexities

- constructor: O(1)
- destructor: O(1)
- copy constructor: O(n)
- has_next: O(1)
- has_previous: O(1)
- next: O(1)
- previous: O(1)
- add: O(n)
- set: O(1)
- remove: O(n)

Pseudocode for non-trivial subalgorithms

```
iterator.add(element):
      // the iterator is pointing between 2 elements (A and B)
      // the new element will be added between those elements
      // the new iterator points now between element and B
      // element – the element that needs to be added
      // last_index – this index is used for the remove() and set() operations
      // current_index – represents the index of the element that would be
returned by next()

      vector.add(element, current_index)
      current_index += 1
      last_index = -1
      // this becomes -1 because the remove() operation must be done after
operations next() or previous()
```

```
vector.add(element, index):
      // element – the element that needs to be added
      // index – the index where it will be added

      if index < 0 or index > vector.size():
            throw exception(“Index out of range”)

      if vector is at full capacity:
            vector.resize()

      // starting from index, move all elements to the right
      for (i = vector.size(); i > index; i--)
            elements[i] = elements[i – 1]

      vector.size += 1
      elements[index] = element
```

```
iterator.remove():
      // the element from position last_index will be removed
      // if there is no such element, then it will throw exception
      // if current_index is bigger than last_index after the remove
      // then we will have to decrement it, because else the iterator will
      // point to a wrong element

      if last_index == -1:
            throw exception(“Neither next or previous have been called”)

      vector.remove(last_index)
      if last_index != current_index:
            current_index -= 1

      last_index = -1
```

```
vector.remove(index):
      // the element from position index will be removed

      if index < 0 or index > vector.size():
            throw exception(“Index out of range”)

      vector.size -= 1

      // starting from index, move all elements to the left
      for (i = index; i < vector.size(); i++)
            elements[i] = elements[i + 1]
```

2. Representation with Doubly Linked List

```
template<typename T>
class DoublyLinkedList {
    public:
        struct Node {
            Node* prev;
            Node* next;
            T info;

            Node() {}
            Node(T info) : info(info) {}
        };

        DoublyLinkedList();
        ~DoublyLinkedList();
        DoublyLinkedList(const DoublyLinkedList<T>& that);
        DoublyLinkedList& operator=(const DoublyLinkedList<T>& that);

        int size() const;
        void add(T element, Node* current);
        void set(T element, Node* current);
        void remove(Node* current);

    public:
        Node* pre;
        Node* post;

    private:
        int dimension;
};

template<typename T>
class ListDoublyLinkedList {
      private:
            DoublyLinkedList<T>* list;
};
```

ListDoublyLinkedList complexities

- constructor: O(1)
- destructor: O(n)
- copy constructor: O(n)
- empty: O(1)
- size: O(1)
- clear: O(n)
- begin: O(1)

ListIterator complexities

- constructor: O(1)
- destructor: O(1)
- copy constructor: O(n)
- has_next: O(1)
- has_previous: O(1)
- next: O(1)
- previous: O(1)
- add: O(n)
- set: O(1)
- remove: O(n)

Pseudocode for non-trivial subalgorithms

```
iterator.next():
      // theoretically, the iterator is pointing between 2 elements (A and B)
      // but practically, it points to B
      // so the next element is actually B
      // current – represents a node where the iterator is located
      // last – represents a node used by remove() or set()

      allocate next_node as a new Node
      next_node = current
      // we have to return this node info

      current = current.next
      // advance the iterator

      last = current
      // this will be deleted by remove() or set by set()

      return current.info
```

```
list.add(element, current):
      // element – the element that needs to be added
      // current – a node in the list
      // the new element will be added before before current

      // because at the beginning we introduced 2 sentinel nodes at the
      // start and at the end of the list, we don't need to worry about
      // particular cases

      allocate prev_node as a new Node
      prev_node = current.prev
      allocate new_node as a new Node

      prev_node.next = new_node;
      new_node.prev = prev_node;
      new_node.next = current;
      new_node.info = element;
      current.prev = new_node;
      list.size += 1
```

```
iterator.remove(current):
      // current – a node in the list that will be removed

      // because at the beginning we introduced 2 sentinel nodes at the
      // start and at the end of the list, we don't need to worry about
      // particular cases

      allocate prev_node as a new Node
      prev_node = current.prev
      allocate next_node as a new Node
      next_node = current.next

      prev_node.next = next_node;
      next_node.prev = prev_node;
      deallocate memory for current and delete it
      list.size -= 1
```

```
list.destructor():
      // pre – a sentinel node at the beginning of the list

      allocate current as a new Node
      current = pre

      while current is a valid Node:
            last = current
            current = current.next
            deallocate memory for current and delete it
```

Problem solution

Consider n balloons that are moving vertically. Select the biggest number of balloons such that they will not touch between them. (Solution should also present the list of balloons.)

We will use the greedy method. This problem can be reduced to the "Activity selection" problem. In the "Activity selection" problem we have some activities and their interval time. We need to choose a subset of these activities such that they do not intersect and the size of the subset is maximal.
In our problem, a balloon is defined by the x coordinate of the center, the y coordinate of the center and the radius. Every balloon can be transformed to an activity such that the interval corresponding to a balloon b is [b.x – b.radius, b.x + b.radius].
Now we can solve the problem in the following way: if we are to choose a single activity, choosing the one that ends first (at a time t1), will leave all the remaining time interval free for choosing other activities. If we choose any other activity instead, the remaining time interval will be shorter. This is obvious, because we will end up anyway with only one activity chosen, but at a time t2 > t1. In the first case we had available all the time span between t1 and finish and that included the time between t2 and finish. Consequently, there is no disadvantage in choosing the activity that finishes earlier. The advantage may result in the situation when we are able to insert another activity that starts between t1 and t2 and ends up before the end of any activity that starts after time t2.
We can sort the activities in increasing order after the end time of each activity and then greedily select the activity that ends the earliest. After selecting an activity, we must update the minimum start time for the next activity.

Input specification

The balloons are given in a text file.
The first line of the file is a natural number n representing the number of balloons.
Then n lines follow, each of them containing 3 natural numbers: the x coordinate of the balloon, the y coordinate of the balloon and the radius of the balloon.

Example

Input:	Output:	Explication:

4
11 2 2
3 3 2
7 5 3
14 3 1
	The biggest number of balloons is: 2
	
	X coordinate: 3
	Y coordinate: 3
	Radius: 2
	
	X coordinate: 11
	Y coordinate: 2
	Radius: 2
		The intervals are:
		[9, 13]
		[1, 5]
		[4, 10]
		[13, 15]
		
		The algorithm will choose the intervals [1, 5] and [9, 13].

Testing

Test	Number of balloons	Dynamic Vector	Doubly Linked List
test1.in	100	0.000351	0.000245
test2.in	1000	0.008205	0.005440
test3.in	10000	0.747499	0.424340
test4.in	50000	17.108337	10.975010

Analysis

The implementation with Dynamic Vector performed worse than the implementation with Doubly Linked List on this particular problem.

In this particular problem the heavy operation is the add operation on the iterator of the list. Assuming the data is random, it is clear that the linked list is better. Think about the case where an element is added in the middle of the list. In both representations, we must iterate until we find the desired position. Then, in the dynamic vector representation we need to shift all elements one position forward. But in the linked list representation we just need to insert the element and modify some pointers (next and prev). So, in average, the linked list is faster.

Assume that the problem only asked for append operations. Then clearly the dynamic vector would have been better, because we don't need to shift anything. Just add the element and we're done. Also, if we had some random access operation, the dynamic vector would have also won, because the time complexity is O(1). Meanwhile, the linked list must traverse the whole list, making the time linear.

However, in this particular problem, the linked list is the better representation. We can learn from this that we must adapt our data structures after the problem and the operations that we need to do.