DATA STRUCTURES AND ALGORITHMS

TUTORIALs AND LABs #4

Thanh-Sach LE
LTSACH@hcmut.edu.vn

April 15, 2020

Introduction

List is an data structure that helps developers to store a set/bag of data elements in an **specific order**. Word **order** does not mean ascending or descending; it means that the data elements are organized in an sequential maner, one element has **at most** one element standing on its left and one on its right. A list of points in 2D-space can be represented logically as follows: $\{(1.5, 3.2), (2.4, 1.7), \cdots, (4.5, 6.8)\}$.

List is a very important data structure, you can see its applications in many projects, from simple applications for buying tickets in movie theaters to ones in education like student and course management systems and to e-commerce ones. Therefore, all programming languages have libraries to support list with free, for examples, **java.util.ArrayList**, **java.util.LinkedList** (in **Java**), and **std::list** (in **C++**).

This document helps students:(a) **to use** existing list's implementations, for examples, **java.util.ArrayList** and **java.util.LinkedList** and then (b) **to develop** their own implementations for the list's abstract data type - **java.util.List**.

1 List's Abstract data type (ADT)

An abstract data type just tells to developers **what** features it can provide, and it hides the detail implementation; actually, it **does not contain** any implementation. In the programming view-point, **interface** (**Java**) and **abstract class** (**C++**) are the best choices for declaring abstract data type.

What features a list should provide to developers? Obviously, a list must allow developers to **add** and to **remove** data elements; a list's ADT must allow developers to be able to traverse a list to obtain its data elements to do some processings. A full list of such the features is presented in Figure 1. Actually, APIs given in Figure 1 are copied from **java.util.List**; so students should visit Oracle website for a detail specification. In Java, there are several implementations supporting **java.util.List**, for examples, **java.util.ArrayList**, **java.util.LinkedList**, and **java.util.Vector**. You can find their source code from openjdk website.

Questions in this document will ask you to add **your own** implementations to support **java.util.List**. Your implementation must stable enough to be used in other labs and assigments of this course. However, before developing your own library, you have to use APIs given in **java.util.List**. Straightfowardly, if you could not use **java.util.List** and its implementations in Java then it is certain that you can not create a similar thing.

```java
public interface List<E>{
    //Group-1: read list's properties
    public int size();
    public boolean isEmpty();

    //Group-2: add elements
    public boolean add(E e);
    public void add(int index, E element);

    //Group-3: remove elements
    public E remove(int index);
    public boolean remove(Object o);
    public void clear();

    //Group-4: set and get elements with indices
    public E get(int index);
    public E set(int index, E element);

    //Grpup-5: map an object to its index + check object existing?
    public int indexOf(Object o);
    public int lastIndexOf(Object o);
    public boolean contains(Object o);

    //Group-6: travel on lists
    public Iterator<E> iterator();
    public ListIterator<E> listIterator();
    public ListIterator<E> listIterator(int index);

    //Supplementary functionalities
    public Object[] toArray();
    public <T> T[] toArray(T[] a);
    public boolean containsAll(Collection<?> c);
    public boolean addAll(Collection<? extends E> c);
    public boolean addAll(int index, Collection<? extends E> c);
    public boolean removeAll(Collection<?> c);
    public boolean retainAll(Collection<?> c);
    public List<E> subList(int fromIndex, int toIndex);
}
```

Figure 1: ADT of lists. List's API inspired from interface **java.util.List**

.

1.1 General steps for using java.util.List

The following are general steps for using **java.util.List**:

1. **Creating a list object**: instantiate a list object by using an a specific implementation. For example, on Line 9 in Figure 2, we create an instance of class **MyArrayList** and store its reference to variable named list. Please note that you should declare list with type **List** instead of **MyArrayList** to cause the code has the minimum dependency on **MyArrayList**; thereby, if you want to change the implementation later to another, just replace **MyArrayList**; for example, replacing **MyArrayList** by **ArrayList**.

2. **Adding new data elements**: add data elements to list by using method add in **Group-2** in Figure 1 or method add in **ListIterator**, explained later.

3. **Traversing a list and processing its data elements**: now, you need to list or to visit each element in the list and do some kinds of processing with each data element.

```java
package list;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class ArrayListDemo {
    public static void main(String[] args){
        List<String> list = new MyArrayList<>();
        for(int idx=0; idx < 10; idx++){
            list.add("" + idx);
        }

        //(1)Print elements - Use Index, travel forward
        System.out.printf("%-32s", "Go forward, use index:");
        for(int idx=0; idx < list.size(); idx++){
            System.out.printf("%s ", list.get(idx));
        }
        System.out.println();

        //(2)Print elements - Use Index, travel backward
        System.out.printf("%-32s", "Go backward, use index:");
        for(int idx=list.size()-1; idx >= 0; idx--){
            System.out.printf("%s ", list.get(idx));
        }
        System.out.println();

        //(3)Print elements - Use Iterator, travel forward
        System.out.printf("%-32s", "Go forward, use Iterator:");
        Iterator<String> it = list.iterator();
        while(it.hasNext()){
            String item = it.next();
            System.out.printf("%s ", item);
        }
        System.out.println();

        //(4)Print elements - Use ListIterator, travel forward
        System.out.printf("%-32s", "Go forward, use ListIterator:");
        ListIterator<String> fwd = list.listIterator();
        while(fwd.hasNext()){
            String item = fwd.next();
            System.out.printf("%s ", item);
        }
        System.out.println();

        //(5)Print elements - Use ListIterator, travel backward
        System.out.printf("%-32s", "Go backward, use ListIterator:");
        ListIterator<String> bwd = list.listIterator(list.size());
        while(bwd.hasPrevious()){
            String item = bwd.previous();
            System.out.printf("%s ", item);
        }
        System.out.println();
    }
}
```

Figure 2: Using list: adding elements and traversing a list

.

1.2 List's Iterators

```
1  interface Iterator<E>{
2      public boolean hasNext();
3      public E next();
4      public void remove(); //optional
5  }
6  interface ListIterator<E>{
7      //for moving forward: head to tail
8      public boolean hasNext();
9      public E next();
10     public int nextIndex();
11
12     //for moving backward: tail back to head
13     public boolean hasPrevious();
14     public E previous();
15     public int previousIndex();
16
17     //for processing lists during a travelling
18     public void remove();
19     public void set(E e);
20     public void add(E e);
21 }
```

Figure 3: APIs for traversing lists with iterators, inspired from interface **java.util.Iterator** and **java.util.ListIterator**
.

List is data structure that holds a sequence of data elements. So, the question often is **how do we traverse a list to get the data elements for processing?**. Basically, with APIs in **java.util.List**, we can have the following approaches.

1. **Using index**: APIs in **Group-4** of Figure 1 can help programmers to retrieve or to change data elements by using index. Programmers can traverse **forwardly**, as shown in Figure 2(Line 16-18) or **backwardly**, as shown Figure 2 (Line 23-25).
2. **Using iterator**: As shown in Figure 1 (Line 25-27), **java.util.List** can help programmers to obtain two kinds of iterators: **java.util.Iterator** and **java.util.ListIterator**, which are defined in Figure 3.
   (a) Interface **java.util.Iterator** contains APIs to support traversal forwardly only. An example of using this interface is given in Figure 2 (Line 29-34). It is important to remember that you must ensure to call next() after a calling to hasNext() to make sure that there are more items for further reading.
   (b) Interface **java.util.ListIterator** contains APIs to support traversal forwardly and backwardly. Examples are shown in Figure 2 (Line 38-53). Programmers should call methods in groups, going-forwardly:{hasNext(), next()} and going-backwardly: {hasPrevious(), previous()}.

Using iterators, we can traverse a list and whenever arrive at a data element, we usually do the following:

- do a general processing for the application being developed,
- remove the element from the list, by invoking method remove(),
- replace the element with a new element, by invoking method set(.), see following examples for more information.

1.3 Adding elements and traversing a list

Question 1

Do the tasks below:

1. Create a Java program as shown in Figure 2. Replace **MyArrayList** by **java.util.ArrayList**, **java.util.LinkedList**, and **java.util.Vector** on Line 9.
2. Compile and Run the program, try to see the result as shown in Figure 4.

```
1 Go forward, use index:           0 1 2 3 4 5 6 7 8 9
2 Go backward, use index:          9 8 7 6 5 4 3 2 1 0
3 Go forward, use Iterator:        0 1 2 3 4 5 6 7 8 9
4 Go forward, use ListIterator:    0 1 2 3 4 5 6 7 8 9
5 Go backward, use ListIterator:   9 8 7 6 5 4 3 2 1 0
```

Figure 4: Output of examples on adding elements and traversing a list

.

1.4 Adding and removing data elements

Question 2

Do the tasks below:

1. Create a Java program as shown in Figure 5
2. Line 24-26 are to remove the odd numbers from the list and to replace the even numbers with new numbers 10 time bigger. **Can you use a for-loop to identify the data elements by their index and then remove or replace the elements as above?**

1.5 Processsing data elements

Question 3 : removing points by hit-test

Do the tasks below:

1. Develop a method with signature void removeHittedPoints(List<Point2D> list, Point2D testPoint, double radius). This method is to remove from list all the points that have the distance to testPoint less than radius.
2. Write a main function to:
   (a) Generate N points in 2D-space (using code in previous Labs).
   (b) Convert the array of points (from previous step) to one of the following list: **ArrayList**, **LinkedList**, or **Vector**.
   (c) Call removeHittedPoints with proposed arguments for testPoint and radius
   (d) Print the original list and the modified list and verify the results for different sets of testPoint and radius.

```java
package list;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.ListIterator;

public class ArrayListDemo {
    public static void main(String[] args){
        List<Integer> list = new ArrayList<>();
        //Add elements
        for(int idx=0; idx < 10; idx++){
            list.add(idx);
        }

        //(1)Print elements - Use Index, travel forward
        System.out.printf("%-25s", "Before modification:");
        for(int idx=0; idx < list.size(); idx++){
            System.out.printf("%s ", list.get(idx));
        }
        System.out.println();
        //(2)Remove odd numbers
        ListIterator<Integer> it = list.listIterator();
        while(it.hasNext()){
            int item = it.next();
            if(item % 2 != 0 ) it.remove();
            else it.set(item*10);
        }
        //(3) Print after changing
        System.out.printf("%-25s", "After modification:");
        it = list.listIterator();
        while(it.hasNext()){
            System.out.printf("%s ", it.next());
        }
        System.out.println();
    }
}
```

Figure 5: List's ADT: Adding and removing data elements. To remove the items that satisfiy some conditions, you can use remove from **Iterator**; see Line 22-27. Remember that if you just change the implementation from **ArrayList** to **LinkedList** or **Vector** on Line 9, the other code is the same.

1.6 ArrayList, LinkedList or Vector

In Java, **ArrayList** and **LinkedList** do not support synchronized calls, so they are not safe to permit multi-threads acessing concurrently. In such the requirement, using **Vector** instead or warping **ArrayList** or **LinkedList** in a new synchronized class.

ArrayList and **Vector** store the reference to its data elements continuosly in an array, so programmers can access to the elements **randomly** via APIs get(int index) or set(int index) in **List**. Meanwhile, **LinkedList** uses a data structure called **node** to store the reference of a data element; and each node has two links, one points to the previous, and the other points to the next node[1]. A node can be allocated anywhere in the memory, hence **LinkedList** does not allow randomly accessing.

Random accessing - what does it means? It means that the cost for obtaining the reference to a data element is **constant**, or it provides a **fast** method to obtain every element.

Table 1 presents the execution time for get(int index) and add(int index, E element). To measure the execution time of get(int index), the following steps were done:

[1]Java **LinkedList** is a double-linked list

1. Create three lists, each for **ArrayList**, **LinkedList** and **Vector**; and initialize each list with 10000 items (from "0" → "9999").

2. For each implementation, we try 1000 times, the execution time is the average of 1000 tries. For each try, we generate a random index (from 0 → 9999), and the use this index to obtain the data element.

Thank to randomly-accessing capability, **ArrayList** and **Vector** are faster then **LinkedList** in operations like get(int index) and set(int index, E element), as shown in Column 2 in Table 1.

However, because **ArrayList** and **Vector** store their data elements **continuosly** in the memory; so, in order to insert or to remove an element at a specific index, it needs to shift a set of data elements forward or backward one step. This shifting operation is costly in time. Meanwhile, in order to do the same operations, **LinkedList** just allocates a new node to store the new element and then inserts the new node to its current chain of nodes at the given location. Thereby, **LinkedList** is **much faster** compared to **ArrayList** and **Vector** in operations like inserting or removing an element at a specific location, as shown in Table 1, Column 3, which was obtained using following steps:

1. For each implementation, we try 1000 times of running; the reported time is the average of those number of tries.

2. For each try, we call add(int index, E element) 50000 times to add 50000 elements to the list. To test the effect of data movement, we try to call add(int index, E element) with **index = 0** for every call.

3. The list is cleared at the end of each try.

Table 1: Execution time of several implementations (in milisec)

Implementation	get(int index)	add(int index, E element)
ArrayList	0.10312	168.52047
LinkedList	28.93115	2.03744
Vector	0.02902	170.50616

Question 4 : time benchmark

Do the tasks below:

1. Develop a Java program to measure the execution time as shown in Table 1.

2. Add one more collumn to the output in previous task to show the execution time for adding new items to the end of each list, e.g., calling to add(E element).

2 Array-based List

This section presents a guidelline to implement the list's abstract data type using array. The internal data structure of this implementation is illustrated in Figure 6. To implement an array-based list (in Java), developers should maintain the following data items:

- $size$: to keep the count of elements in the list; this data field is an integer.
- $elements$: to keep the references of the elements in the list, using array in Java. As shown in Figure 6, the array contains **references** only, the actual data of elements is stored somewhere **separately**. The length of this array is referred to as $capacity$. In Java it is unneccessary to store $capacity$, because it can be retrieved using $elements.length$.

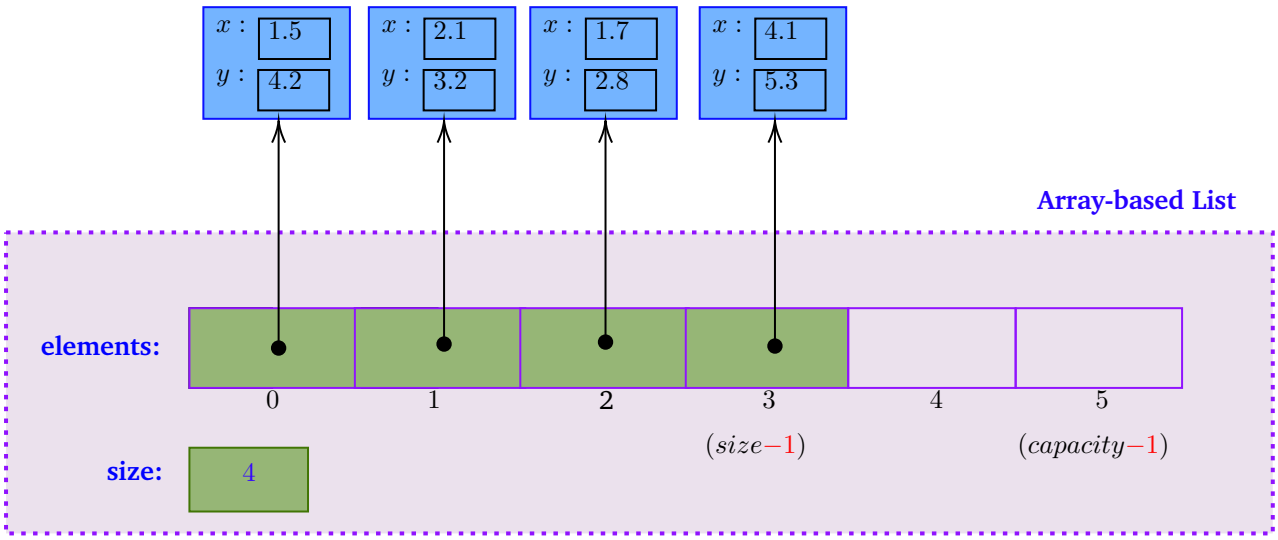


Figure 6: An example of array-based list. The element type is **Point2D**; $size = 4$: there are four elements (points) in the list; $capacity = 6$: the internel array can contain six elements maximum. $capacity$ will be **increased automatically** when the list has more elements.

.

2.1 class MyArrayList

In this tutorial, **MyArrayList** is a class implementing **java.util.List** to support the list's abstract data type. Its overview is presented in Figure 7.

- Line 1: declares that **MyArrayList** implements **java.util.List**; therefore, **MyArrayList** has the implementations for methods defined **java.util.List** (see Figure 1).
- Line 2: defines two constants called **NEXT** and **PREV** that will be used for traversing lists.
- Line 3: declares the maximum count of elements in any list instance. The capacity will be increased automatically, but when it hits this maximum an exception **OutOfMemoryError** wil be thrown.
- Line 6-7: declares internal data fields as explained above.
- Line 10-12: defines two constructors; one to allow programmers specifying an initial capacity; and the other initializes the capacity with 10.
- Line 14-15: defines utilities that facilitate the implementation of other methods.
- The remainder contains the implementation of the methods defined in **java.util.List** and two classes **MyIterator** and **MyListIterator** supporting iterator interfaces associated with **java.util.List**.

2.1.1 MyArrayList's constructors

The main task of a constructor is to **allocate** memory for the internal array and to **initialize** the list's size to 0, see Line 7-8 in Figure 8. However, it has to verify whether the input capacity is valid or not. In the case that the capacity is invalid, it throws an exception called **IllegalArgumentException**.

2.1.2 MyArrayList's utilities

1. checkValidIndex(int index, int min, int max): many other APIs need programmers passing an index of the element being processed, for examples, get(int index) and add(int index, E element). checkValidIndex helps to verify the validity of the input index; an index is valid if it satisfies $min \leq index \leq max$. To add new elements, max can be $size$ instead of $(size-1)$ as in removing existing elements. When a list is empty, add a new with $index = 0$ is fine, but there will be an exception called **IndexOutOfBoundsException** if programmers want to retrieve or to remove element at $index = 0$ with an empty list. The detail implementation of checkValidIndex is given in Figure 8.
2. checkCapacity(int minCapacity): this method is used to **ensure** that there are room for storing new elements. If the list's size is less than the current capactity (i.e., $elements.length$) then it is fine to add a new element. However, if the list's size **equals** to the current capacity; then the following must be done for adding new elements to the list:

   (a) the internal array must be re-allocated, and

   (b) the old stored in elements must be **copied** to the new memory allocated.

   These two tasks are done with API Arrays.copyOf(this.elements, newCapacity), see Line 29 in Figure 8. However, before calling to Arrays.copyOf, newCapacity must be determined. By using formula newCapacity = oldCapacity + (oldCapacity » 1), newCapacity ≡ oldCapacity + 0.5*oldCapacity if possible.

```
public class MyArrayList<E> implements java.util.List<E>{
    private static enum MoveType{ NEXT, PREV};
    private static final int MAX_CAPACITY = Integer.MAX_VALUE - 8;

    //Internal data fields
    private E[] elements;
    private int size;

    //Constructor
    public MyArrayList(int capacity) throws IllegalArgumentException{/*here
        : code*/ }
    public MyArrayList() throws IllegalArgumentException{ this(10); }

    //Utitility methods (private)
    private void checkValidIndex(int index, int min, int max){/*here: code
        */  }
    private void checkCapacity(int minCapacity){/*here: code*/ }

    //Group-1: read list's properties
    public int size() {  return this.size; }
    public boolean isEmpty() { return this.size == 0; }

    //Group-2: add elements
    public boolean add(E e) {/*here: code*/ }
    public void add(int index, E element) {/*here: code*/   }

    //Group-3: remove elements
    public E remove(int index){/*here: code*/ }
    public boolean remove(Object o) {/*here: code*/  }
    public void clear(){/*here: code*/ }

    //Group-4: set and get elements with indices
    public E get(int index){/*here: code*/ }
    public E set(int index, E element){/*here: code*/ }

    //Grpup-5: map an object to its index + check object existing?
    public int indexOf(Object o){/*here: code*/ }
    public int lastIndexOf(Object o){/*here: code*/ }
    public boolean contains(Object o){/*here: code*/ }

    //Group-6: travel on lists
    public Iterator<E> iterator(){/*here: code*/ }
    public ListIterator<E> listIterator(){/*here: code*/ }
    public ListIterator<E> listIterator(int index){/*here: code*/ }

    //Supplementary functionalities
    public Object[] toArray() {/*here: code*/ }
    public <T> T[] toArray(T[] a) {/*here: code*/ }
    public boolean containsAll(Collection<?> c) {/*here: code*/  }
    public boolean addAll(Collection<? extends E> c) {/*here: code*/ }
    public boolean addAll(int index, Collection<? extends E> c) {/*here:
        code*/ }
    public boolean removeAll(Collection<?> c) {/*here: code*/  }
    public boolean retainAll(Collection<?> c) {/*here: code*/  }
    public List<E> subList(int fromIndex, int toIndex) {/*here: code*/ }

    //Inner Classes
    public class MyIterator implements Iterator<E>{/*here: code*/  }
    public class MyListIterator extends MyIterator implements ListIterator<
        E>{/*here: code*/  }
}//End of MyArrayList
```


Figure 7: class **MyArrayList**: Overview

```java
public class MyArrayList<E> implements java.util.List<E>{
    public MyArrayList(int capacity) throws IllegalArgumentException{
        if((capacity < 0) || (capacity > MAX_CAPACITY)){
            String message = String.format("Invalid capacity (=%d)",
                capacity);
            throw new IllegalArgumentException(message);
        }
        this.elements = (E[])new Object[capacity];
        this.size = 0;
    }
    public MyArrayList() throws IllegalArgumentException{ this(10); }

    //Utitilies
    private void checkValidIndex(int index, int min, int max){
        if((index < min) || (index > max)){
            String message = String.format("Invalid index (=%d)", index);
            throw new IndexOutOfBoundsException(message);
        }
    }
    private void checkCapacity(int minCapacity){
        if((minCapacity < 0) || (minCapacity > MAX_CAPACITY))
            throw new OutOfMemoryError("Not enough memory to store the
                array");
        if(minCapacity < this.elements.length) return;
        else{
            //grow
            int oldCapacity = this.elements.length;
            int newCapacity = oldCapacity + (oldCapacity >> 1);
            if(newCapacity < 0)
            newCapacity = MAX_CAPACITY;
            this.elements = Arrays.copyOf(this.elements, newCapacity);
        }
    }
    //Here: more code of this class
}//End of MyArrayList
```

Figure 8: class **MyArrayList**: Constructors and utilities

.

2.1.3 Adding a new element to an array-based list

To add a new element at **a specific index**, we need to move all elements from $index$ to $(size - 1)$ to the right side one location, and then we copy the reference of the new element to location identified by $index$. These operations are illustrated Figure 9 and 10, Java code implementation for this method is presented in Figure 11 (Line 2-11).

Please note that the **data movement is costly**, so try to avoid adding a specific index with array-based lists as possible.

The list's ADT also has one method for adding new data element at the end of a list, i.e. **appending the new data element to the list**. The implementation for such kind of the adding is simple, as shown in Figure 11 (Line 12-18). Certainly, this kind of the adding is **cheap**, just a constant time.

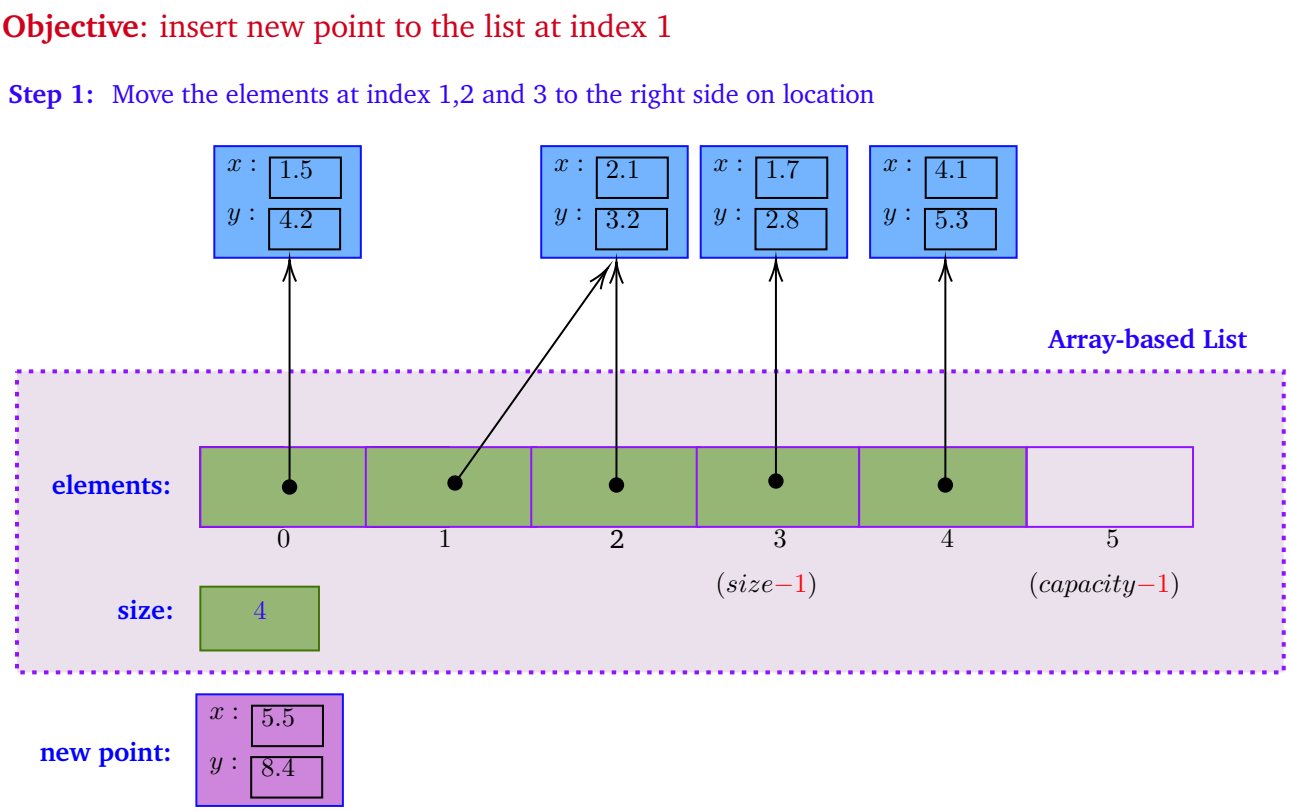
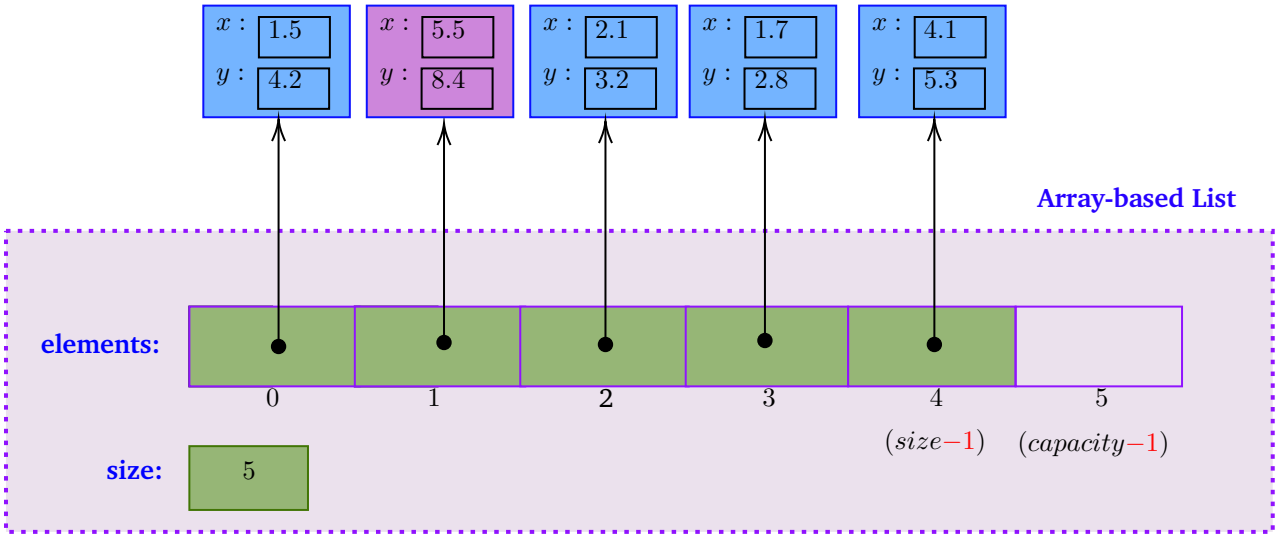


Figure 9: Adding new elements. Step 1: move elements at index 1,2,3 in Figure 6 to the right side one location

.

Objective: insert new point to the list at index 1

Step 2: Copy the reference to the new object to the array at index 1



Step 3: Increase the list's size, $size = size + 1$

Figure 10: Adding new elements. Step 2+3: Copy the reference to the new data element to location at index =1, and increase the list's size

.

```
1  public class MyArrayList<E> implements java.util.List<E>{
2      public void add(int index, E element) {
3          checkValidIndex(index, 0, size);
4          if(element == null) throw new NullPointerException("Can not add
               null pointer");
5          checkCapacity(this.size + 1);
6
7          int copyCount = (this.size-1) - index + 1;
8          System.arraycopy(this.elements, index, this.elements, index + 1,
               copyCount);
9          this.elements[index] = element;
10         this.size++;
11     }
12     public boolean add(E e) {
13         if(e == null) throw new NullPointerException("Can not add null
               pointer");
14         checkCapacity(this.size + 1);
15
16         this.elements[this.size++] = e;
17         return true;
18     }
19     //Here: more code of this class
20 }//End of MyArrayList
```

Figure 11: class **MyArrayList**: Adding new elements

.

2.1.4 Removing elements from an array-based list

To remvove an existing element at **a specific index**, we need to move all elements from $index + 1$ to $(size - 1)$ to the left side one location, as shown in Figure 12 and 13, Java code implementation for this method is presented in Figure 14 (Line 2-9).

Similar to adding at specific index, this method is time consuming; especially, in the case removing item at $index = 0$ constantly.

The list's ADT also has one more API remove(Object o) to allow programmers removing an object using the object itself instead of its index. Generally, we can search for the index and the call method remove(int index) to remove the object, as shown in Figure 14 (Line 10-15).

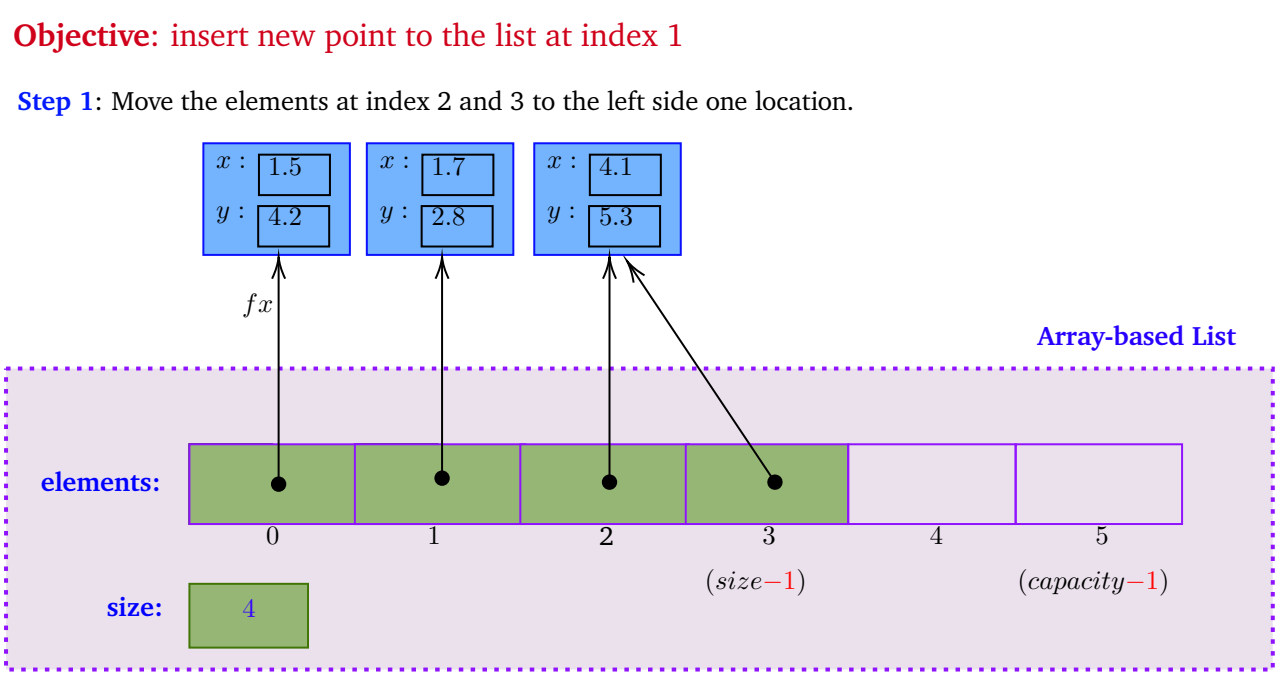
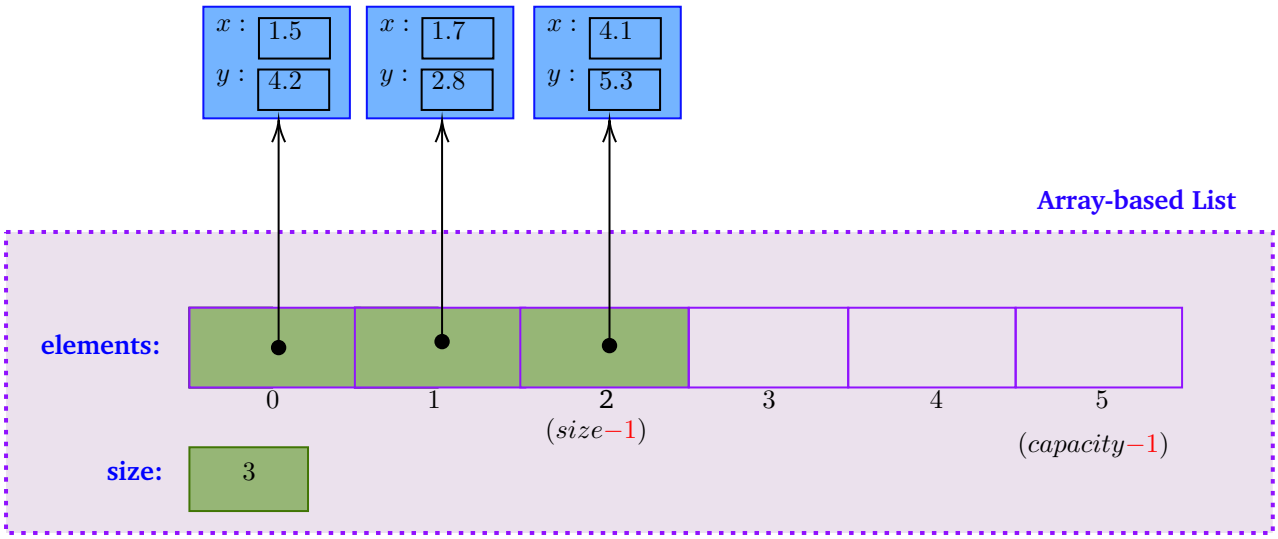


Figure 12: Removing elements. Step 1: move the elements at index 2 and 3 in Figure 6 to the left side one location

.

Objective: insert new point to the list at index 1

Step 1: Move the elements at index 2 and 3 to the left side one location.



Step 2: Assign null to the the last element (at $size - 1$); ie. at index = 3 (optional)

Step 3: Crease the list's size, $size = size - 1$; the last is at index = 2

Figure 13: Removing elements. Step 2+3: Detach the last reference from the internal array and decrease the list's size

.

```
1  public class MyArrayList<E> implements java.util.List<E>{
2      public E remove(int index) {
3          checkValidIndex(index, 0, size -1);
4          E oldElement = this.elements[index];
5          int copyCount = (this.size-1) - (index + 1) + 1;
6          System.arraycopy(this.elements, index + 1, this.elements, index,
               copyCount);
7          this.size--;
8          return oldElement;
9      }
10     public boolean remove(Object o) {
11         int index = indexOf(o);
12         if(index < 0) return false;
13         remove(index);
14         return true;
15     }
16     //Here: more code of this class
17 }//End of MyArrayList
```

Figure 14: class **MyArrayList**: Removing elements

.

2.1.5 Accessing data elements

To access data elements in array-based list we can use index with the internal array, for examples, as in 15, Line 32-41. These kinds of operations are really fast, just need a constant time.

2.1.6 Traversing the internal array

This section presents a way to traverse the internal array of a list do the following:

1. to check whether an object is in the list or not, see Figure 15, Line 22-31.
2. to get index of an existing element, see Figure 15, Line 2-21.

```java
public class MyArrayList<E> implements java.util.List<E>{
    public int indexOf(Object o) {
        int foundIdx = -1;
        for(int idx=0; idx < this.size; idx++){
            if(this.elements[idx].equals(o)){ //== not
                foundIdx = idx;
                break;
            }
        }
        return foundIdx;
    }
    public int lastIndexOf(Object o) {
        int foundIdx = -1;
        for(int idx=this.size-1; idx >=0; idx--){
            if(this.elements[idx].equals(o)){
                foundIdx = idx;
                break;
            }
        }
        return foundIdx;
    }
    public boolean contains(Object o) {
        boolean found = false;
        for(int idx=0; idx < this.size; idx++){
            if(this.elements[idx].equals(o)){
                found = true;
                break;
            }
        }
        return found;
    }
    public E get(int index) {
        checkValidIndex(index, 0, size - 1);
        return this.elements[index];
    }
    public E set(int index, E element) {
        checkValidIndex(index, 0, size-1);
        E oldElement = this.elements[index];
        this.elements[index] = element;
        return oldElement;
    }
    //Here: more code of this class
}//End of MyArrayList
```

Figure 15: class **MyArrayList**: Obtainning the index of a given element

.

2.1.7 Implementations for Iterator and ListIterator

As shown in Figure 1, instances of iterators can be obtained by APIs iterator(), listIterator(), or listIterator(int index).

What are applications of iterators? iterator objects can help programmers the following tasks:

1. to traverse the lists forwardly or backwardly;
2. and, to modify the lists; a modification maybe removing or updating elements

In this tutorial, class **MyIterator** and **MyListIterator** are typical implementations for interface **Iterator** and **ListIterator** respectively. These two classes need to access to the data fields and methods of **MyArrayList**, so it is the best choice to include **MyIterator** and **MyListIterator** as **inner classes** of **MyArrayList**, shown in Figure 16 and 17.

```
1  public class MyArrayList<E> implements java.util.List<E>{
2      public class MyIterator implements Iterator<E>{
3          int cursor = 0;
4          MoveType moveType = MoveType.NEXT;
5          boolean afterMove = false;
6
7          @Override
8          public boolean hasNext() {
9          return this.cursor != MyArrayList.this.size;
10         }
11
12         @Override
13         /*
14         Move cursor to next + return preivous element
15         */
16         public E next() {
17             cursor += 1;
18             moveType = MoveType.NEXT;
19             afterMove = true;
20             return MyArrayList.this.elements[cursor-1];
21         }
22
23         @Override
24         public void remove() {
25             if(!afterMove) return;
26             MyArrayList.this.remove(cursor - 1);
27             cursor -=1;
28             afterMove = false;
29         }
30     }//End of MyIterator
31
32     //Here: more code of this class
33 }//End of MyArrayList
```

Figure 16: class **MyArrayList**: MyIterator

.

```
public class MyArrayList<E> implements java.util.List<E>{
    public class MyListIterator extends MyIterator
                                implements ListIterator<E>{
        public MyListIterator(int index){
            cursor = index;
        }
        public boolean hasPrevious() { return this.cursor != 0; }
        public int previousIndex() { return cursor -1; }
        public E previous() {
            cursor -= 1;
            moveType = MoveType.PREV;
            afterMove = true;
            return MyArrayList.this.elements[cursor];
        }

        public int nextIndex() {
            return cursor;
        }
        public void add(E e) {
            if(!afterMove) return;
            if(moveType == MoveType.NEXT)
                MyArrayList.this.add(cursor-1, e);
            else
                MyArrayList.this.add(cursor, e);
            cursor += 1;
            afterMove = false;
        }
        public void remove() {
            if(!afterMove) return;
            if(moveType == MoveType.NEXT) super.remove();
            else{
                MyArrayList.this.remove(cursor);
                afterMove = false;
            }
        }
        public void set(E e) {
            if(!afterMove) return;
            if(moveType == MoveType.NEXT)
                MyArrayList.this.set(cursor-1, e);
            else
                MyArrayList.this.set(cursor, e);
        }

    }//End of MyListIterator

    //Here: more code of this class
}//End of MyArrayList
```

Figure 17: class **MyArrayList**: MyListIterator

.

Question 5 : create class **MyArrayList**

Do below tasks:

1. Create class **MyArrayList**, using the source code presented in previous figures. For **supplementaty methods** you need not to add your implementations; in such case, you can try throw an exception as in Figure 18.
2. Use **MyArrayList** to solve the above questions (Question 1-4).

```
public class MyArrayList<E> implements java.util.List<E>{
    public <T> T[] toArray(T[] a) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    public boolean containsAll(Collection<?> c) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    public boolean addAll(Collection<? extends E> c) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    public boolean addAll(int index, Collection<? extends E> c) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    public boolean removeAll(Collection<?> c) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    public boolean retainAll(Collection<?> c) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    //Here: more code of this class
}//End of MyArrayList
```

Figure 18: class **MyArrayList**: Unimplemented methods

.

3 Single-linked List

3.1 Data structure of Single-Linked List

The memory architecture of a single-linked list is illustrated in Figure 20 and 21. The figures show that a single-linked list has to maintain the following data items to be able to store data elements:

1. A variable called size to keep the number of data elements stored in the list.
2. Two references called head and tail; they are instances of a class called **Node**, see Figure 19. Each node contains one pointer to the next node and another pointer to the actual data attached to the node. Node head and tail have the following properties:
   - The references to the data element are assigned with **null** ($\equiv 0$);
   - head.next keeps the reference of the first node in the list, i.e., the node contains the first data element. For example, in Figure 21 head to point the node containing point $(1.5, 4.2)$. In the case of empty list, head.next = tail (see Figure 20);
   - meanwhile, tail.next keeps the reference to the last node, i.e., the node contains the last data element. In the case of empty list, tail.next = head;

```
class Node<E>{
    E element;
    Node<E> next;
    Node(Node<E> next, E element){
        this.next = next;
        this.element = element;
    }
    void update(Node<E> next, E element){
        this.next = next;
        this.element = element;
    }
}//End of Node
```

Figure 19: class **Node** for Single-Linked List

.

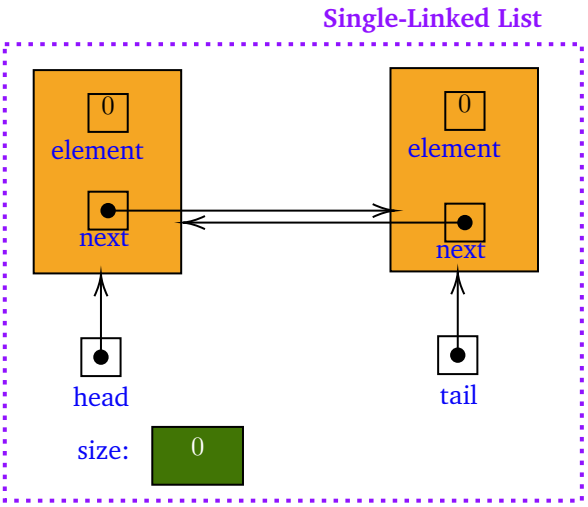


Figure 20: Single-linked list: an empty list

.

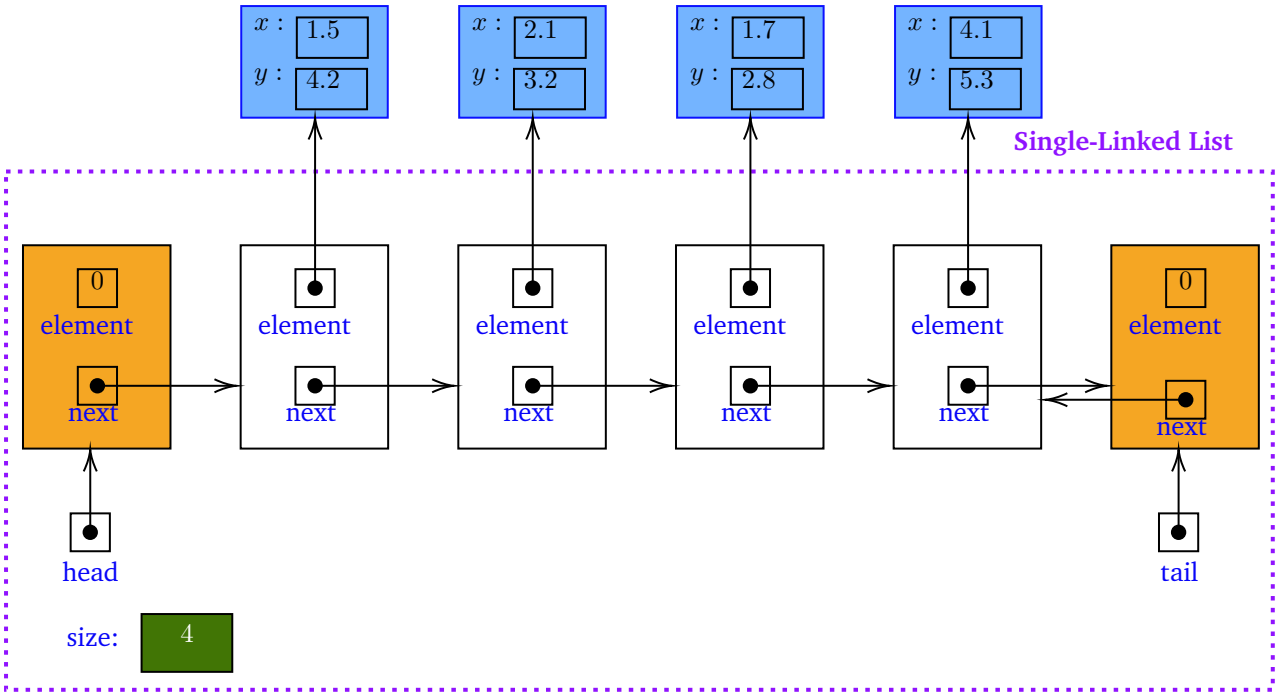


Figure 21: Single-linked list: a list contains 4 points in 2D-space

.

3.2 Class SLinkedList: an overview

Single-Linked List is implemented in class **SLinkedList**, as illustrated in Figure 22.

1. Line 1: declares that **SLinkedList** implements interface **java.util.List** to support the list's abstract data type. Therefore, **SLinkedList** must contains implementations for methods defined in **java.util.List**.
2. Line 3-4: declares the size, head and tail as explained before. **Node** is an inner class of **SLinkedList**, at Line 50; the detail definition of **Node** is presented in Figure 19.
3. Other lines: define methods to support **java.util.List** and iterators, similar to class **MyArrayList**.

3.2.1 SLinkedList's constructors

SLinkedList's constructor creates an empty list, illustrated in Figure 20; its code in Java is presented in Figure 23, Line 2-7.

3.2.2 SLinkedList's utilities

1. void checkValidIndex(int index): this method is to check whether an index is valid or not. An index is valid if it satisfies $0 \leq index \leq (size - 1)$. If the input index is invalid, an exception of **IndexOutOfBoundsException** is thrown.
2. Node<E> getDataNode(int index) : this method is to find and return the node **containing data element** idientified by index; so, the input index must satisfies $0 \leq index \leq (size - 1)$; otherwise, an exception happened.
3. Node<E> getNode(int index) : this method is to find and return the node identified by its index. **index maybe** -1, if so the head is returned. Usually, this method is used to add one node to the list. Because the list contains only one link, so in order to add one node to the list, we we need the reference to the previous node. Therefore, if we want to insert data node the beginning (i.e., index =0), we need head to do this.

```
public class SLinkedList<E> implements java.util.List<E>{
    private static enum MoveType{ NEXT, PREV };
    private Node<E> head, tail;
    private int size;
    public SLinkedList(){/*here: code*/ }
    //Utility methods (private)
    private void checkValidIndex(int index){/*here: code*/ }
    private Node<E> getDataNode(int index) {/*here: code*/ }
    private void addAfter(Node<E> afterThis, Node<E> newNode){/*here: code
        */ }
    private Node<E> removeAfter(Node<E> afterThis){/*here: code*/ }

    //Group-1: read list's properties
    public int size() {  return this.size; }
    public boolean isEmpty() { return this.size == 0; }

    //Group-2: add elements
    public boolean add(E e) {/*here: code*/ }
    public void add(int index, E element) {/*here: code*/   }

    //Group-3: remove elements
    public E remove(int index){/*here: code*/ }
    public boolean remove(Object o) {/*here: code*/  }
    public void clear(){/*here: code*/ }

    //Group-4: set and get elements with indices
    public E get(int index){/*here: code*/ }
    public E set(int index, E element){/*here: code*/ }

    //Grpup-5: map an object to its index + check object existing?
    public int indexOf(Object o){/*here: code*/ }
    public int lastIndexOf(Object o){/*here: code*/ }
    public boolean contains(Object o){/*here: code*/ }

    //Group-6: travel on lists
    public Iterator<E> iterator(){/*here: code*/ }
    public ListIterator<E> listIterator(){/*here: code*/ }
    public ListIterator<E> listIterator(int index){/*here: code*/ }

    //Supplementary functionalities
    public Object[] toArray() {/*here: code*/ }
    public <T> T[] toArray(T[] a) {/*here: code*/ }
    public boolean containsAll(Collection<?> c) {/*here: code*/  }
    public boolean addAll(Collection<? extends E> c) {/*here: code*/ }
    public boolean addAll(int index, Collection<? extends E> c) {/*here:
        code*/ }
    public boolean removeAll(Collection<?> c) {/*here: code*/  }
    public boolean retainAll(Collection<?> c) {/*here: code*/  }
    public List<E> subList(int fromIndex, int toIndex) {/*here: code*/ }

    //Inner Classes
    class Node {/*here: code*/ }
    class MyIterator implements Iterator<E>{/*here: code*/  }
    class MyListIterator extends MyIterator implements ListIterator<E>{/*
        here: code*/  }

}//End of SLinkedList
```

Figure 22: Class **SLinkedList**: An overview

```java
public class SLinkedList<E> implements java.util.List<E>{
    public SLinkedList(){
        head = new Node<>(null, null);
        tail = new Node<>(null, null);
        head.next = tail; tail.next = head;
        size = 0;
    }

    //Utilities (Private)
    private void checkValidIndex(int index){
        if((index < 0) || (index >= size) ){
            String message = String.format("Invalid index (=%d)", index);
            throw new IndexOutOfBoundsException(message);
        }
    }
    private Node<E> getDataNode(int index) {
        checkValidIndex(index);
        Node<E> curNode = head.next;
        int runIndex = 0;
        while(curNode != tail){
            if(index == runIndex) break;
            runIndex += 1;
            curNode = curNode.next;
        }
        return curNode;
    }
    //getNode: can return head
    private Node<E> getNode(int index) {
        if((index < -1) || (index >= size) ){
            String message = String.format("Invalid index (including head)
                (=%d)", index);
            throw new IndexOutOfBoundsException(message);
        }
        Node<E> curNode = head;
        int runIndex = -1;
        while(curNode != tail){
            if(index == runIndex) break;
            runIndex += 1;
            curNode = curNode.next;
        }
        return curNode;
    }
    //Here: more code of SLinkedList
}//End of SLinkedList
```

Figure 23: Class **SLinkedList**: Contructors and utility methods

.

3.3 Insertion operation

3.3.1 Insert a new node after a given node - addAfter

Assume that we want to insert a new data element at a given index or at a position pointed by reference curNode, as shown in Figure 24. We can carry out a searching process to obtain prevNode pointing to the node standing immediately before curNode.

At this point, the problem we need to solve is to insert the new data element after prevNode, and this task consists of steps as illustrated in Figure 25. Java code to do this task is presented in Figure 26, Line 2-7.

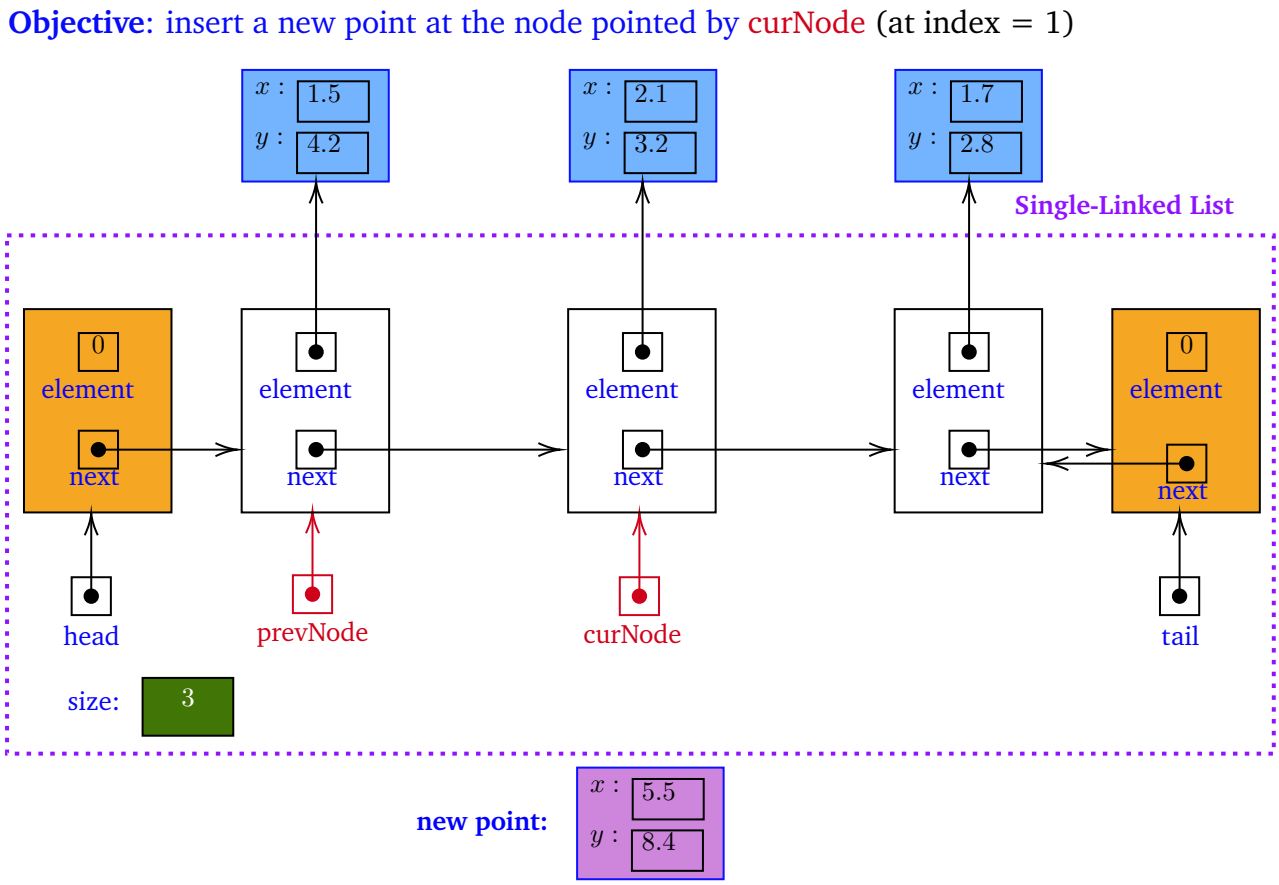


Figure 24: Single-linked list: Inserting a new data element after another element

.

3.3.2 Insert a new node at a specific index - add(int index, E element)

An implementation of add(int index, E element) is shown in Figure 26, Line 8-12. Generally, the main idea is to obtain the reference to the node at $(index - 1)$ instead of at $index$. So, if $index = 0$ then prevNode (see Figure 25) is head; hence the new data item will be the first in the list after insertion.

As shown in Figure 26 at Line 9, we need to retrieve the node at $(index-1)$ by calling to getNode($index-1$), which requires $O(size)$ in worst case for computation. Therefore, the total time complexity of add(int index, E element) is $O(size)$.

3.3.3 Append the new data element at the end of the list - add(Object o)

add(Object o) in **java.util.List** is to append a new data element at the end of the list. We can implement this method by passing the last element of the list to addAfter, as shown in Figure 26, Line 13-18.

This method is fast, because the last element is tail.next; so, the cost of this method is constant, $O(c)$.

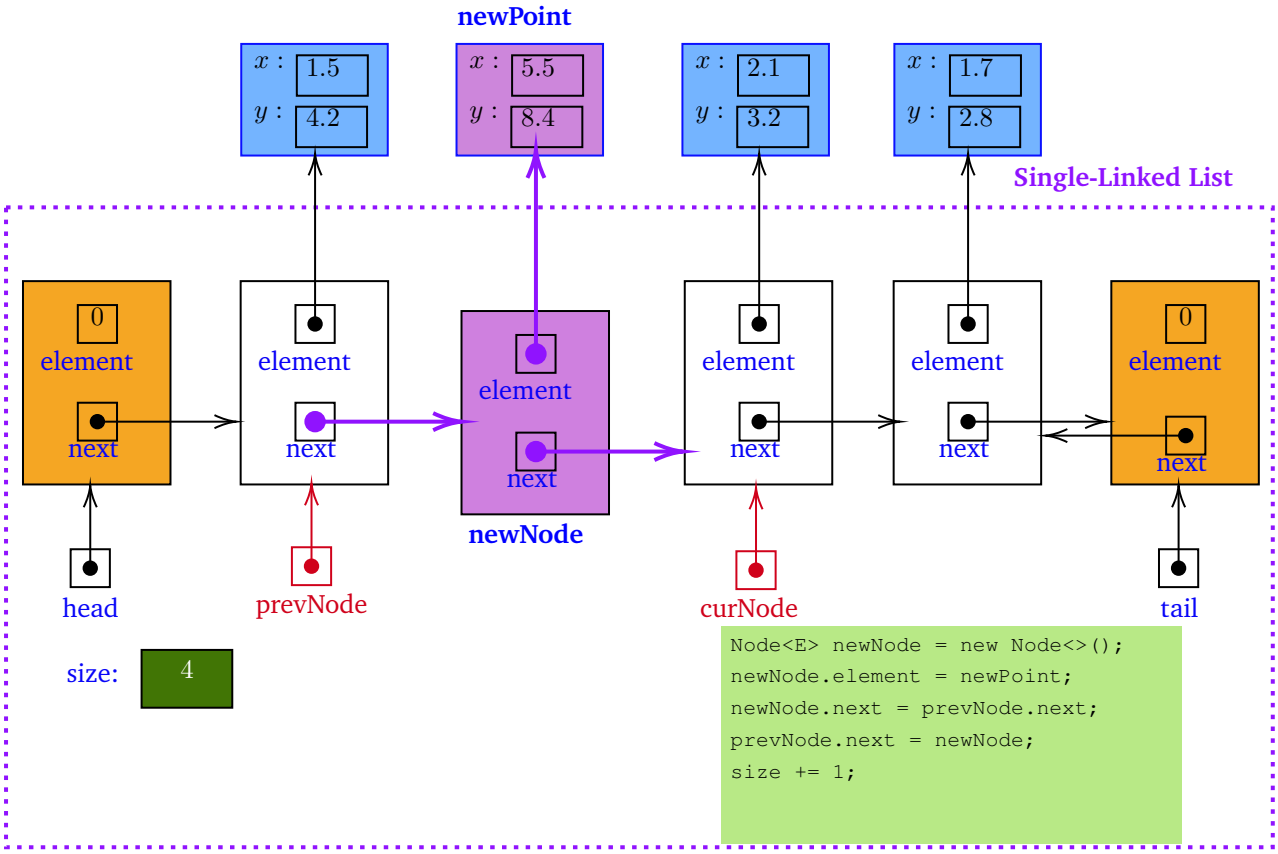


Figure 25: Single-linked list: Inserting a new data element after another element - steps to do

.

```
1  public class SLinkedList<E> implements java.util.List<E>{
2      private void addAfter(Node<E> afterThis, Node<E> newNode){
3          newNode.next = afterThis.next;
4          afterThis.next = newNode;
5          if(newNode.next == tail) tail.next = newNode;
6          size += 1;
7      }
8      public void add(int index, E element) {
9          Node<E> prevNode = getNode(index-1);
10         Node<E> newNode = new Node<>(null, element);
11         addAfter(prevNode, newNode);
12     }
13     public boolean add(E e) {
14         Node<E> newNode = new Node(null, e);
15         Node<E> lastNode = tail.next;
16         addAfter(lastNode, newNode);
17         return true;
18     }
19     //Here: more code of SLinkedList
20 }//End of SLinkedList
```

Figure 26: Class **SLinkedList**: Insertion operations

.

3.4 Removal operation

3.4.1 Removing the node following a given node - removeAfter

As shown in Figure 27, assume that we want to detach the node containing point $(2.1, 3.2)$ from the list. We can perform a searching process to assign curNode to the node being removed and prevNode with the node standing immediately before curNode, i.e. prevNode.next == curNode. Please note that if curNode points to the first data node, then prevNode is head.

Now, the problem is to remove curNode given prevNode, and steps to do this task are illustrated in Figure 28. Java code of removeAfter is presented in Figure 29 (Line 2-8). Please note that we need to update tail if we remove the last data node, this is done code at Line 5.

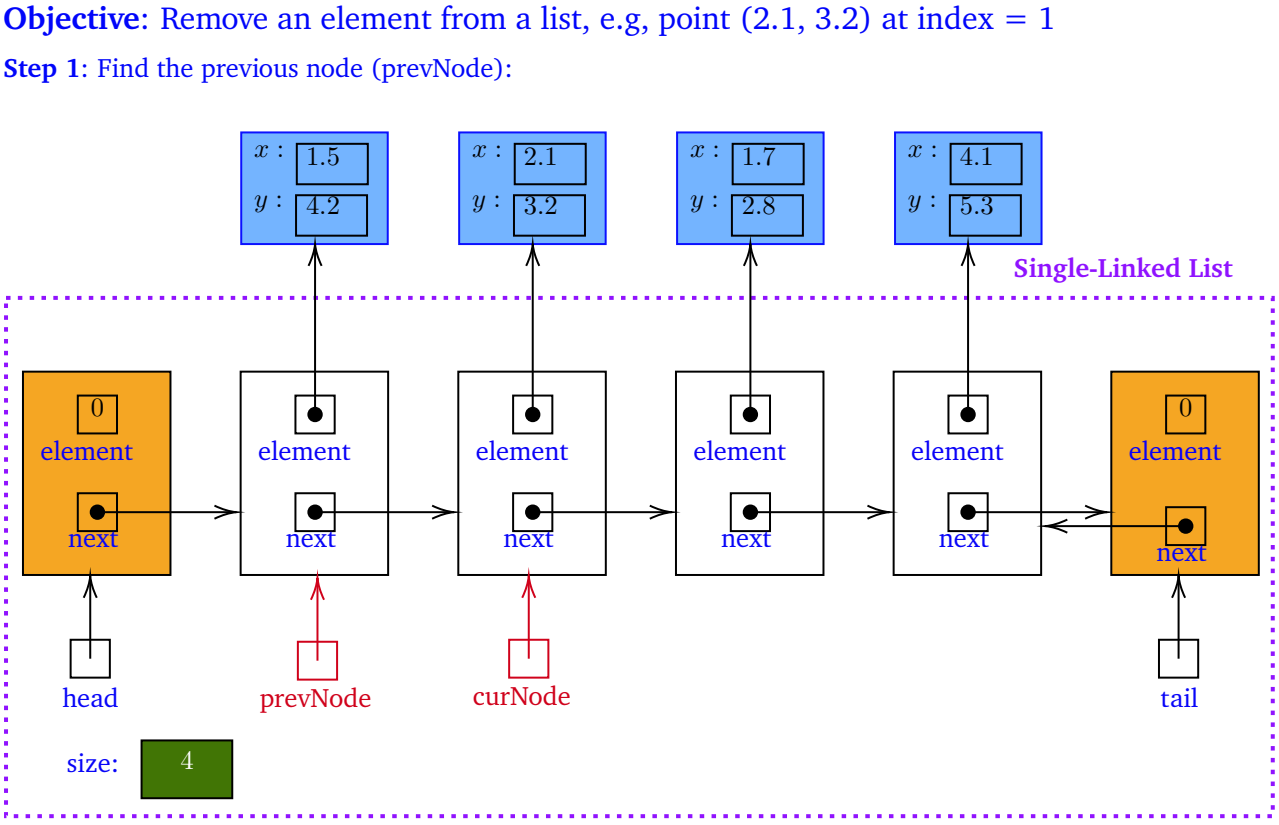


Figure 27: Single-linked list: Removing the element after a given element - Step 1

.

3.4.2 Removing an element given an index - remove(int index)

remove(int index) is a method defined in **java.util.List**. A general idea to implement this method is to retrieve a node that location $(index - 1)$ instead of $index$ and then we invoke removeAfter discussed above; see Java code in Figure 29, Line 25-34.

3.4.3 Removing an element given a data element - remove(Object o)

This functionality is also defined in **java.util.List**. Generally, we search for the data element and return **not only** the reference to node containing the data element, **but also** the reference to its previous node. Thereby, we can call removeAfter with the previous node to remove the data element out of the list; see Java code in Figure 29, Line 10-24.

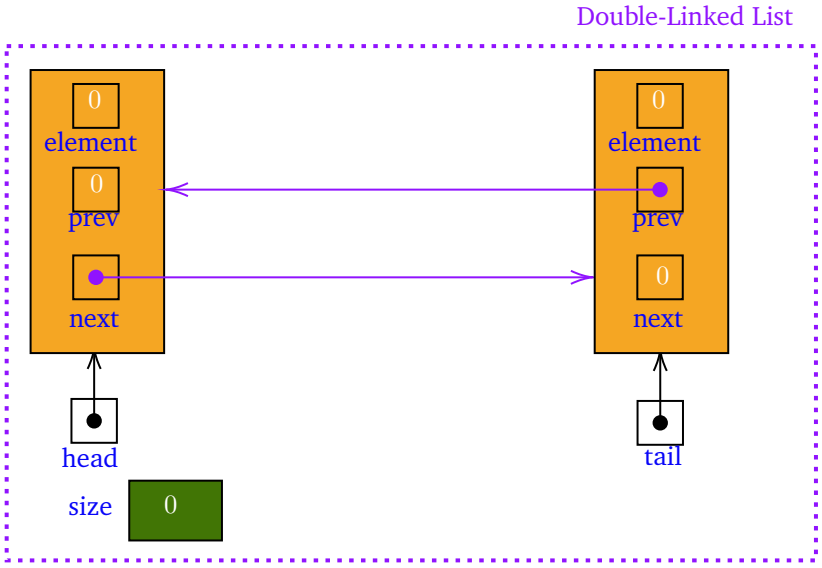


Figure 28: Single-linked list: Removing the element after a given element - Step 2

.

```java
public class SLinkedList<E> implements java.util.List<E>{
    private Node<E> removeAfter(Node<E> afterThis){
        Node<E> removedNode = afterThis.next;
        afterThis.next = removedNode.next;
        if(removedNode.next == tail) tail.next = afterThis;
        removedNode.next = null;
        return removedNode;
    }

    public boolean remove(Object o) {
        Node<E> prevNode = head;
        Node<E> curNode = head.next;
        boolean found = false;
        while(curNode != tail){
            if(curNode.element.equals(o)){
                found = true;
                removeAfter(prevNode);
                break;
            }
            curNode = curNode.next;
            prevNode = prevNode.next;
        }
        return found;
    }
    public E remove(int index) {
        if(size == 0){
            String message = String.format("Remove at %d, but the list is
               empty", index);
            throw new IndexOutOfBoundsException(message);
        }
        Node<E> prevNode = getNode(index-1);
        Node<E> curNode = prevNode.next;
        removeAfter(prevNode);
        return curNode.element;
    }
    //Here: more code of SLinkedList
}//End of SLinkedList
```

Figure 29: Class **SLinkedList**: removing elements

.

3.5 List traversal

```
public class SLinkedList<E> implements java.util.List<E>{
    public class MyIterator implements Iterator<E>{
        int cursor = 0;
        MoveType moveType = MoveType.NEXT;
        boolean afterMove = false;
        @Override
        public boolean hasNext() {
            return this.cursor != MyArrayList.this.size;
        }

        @Override
        /*
        Move cursor to next + return preivous element
        */
        public E next() {
            cursor += 1;
            moveType = MoveType.NEXT;
            afterMove = true;
            return MyArrayList.this.elements[cursor-1];
        }

        @Override
        public void remove() {
            if(!afterMove) return;
            MyArrayList.this.remove(cursor - 1);
            cursor -=1;
            afterMove = false;
        }
    }//End of MyIterator
    //Here: more code of SLinkedList
}//End of SLinkedList
```

Figure 30: Class **SLinkedList**: Inner class **MyIterator**

.

Question 6 : create class SLinkedList

Do below tasks:

1. Create class **SLinkedList**, using the source code presented in previous figures. For **supplementaty methods** you need not to add your implementations; in such case, you can try throw an exception as in Figure 18.
2. Use **SLinkedList** to solve the above questions (Question 1-4).

```
public class SLinkedList<E> implements java.util.List<E>{
    public class MyListIterator extends MyIterator implements ListIterator<
        E>{
        public MyListIterator(int index){
            cursor = index;
        }
        public boolean hasPrevious() {
            return this.cursor != 0;
        }
        public void remove() {
            if(!afterMove) return;
            if(moveType == MoveType.NEXT) super.remove();
            else{
                MyArrayList.this.remove(cursor);
                afterMove = false;
            }
        }
        public E previous() {
            cursor -= 1;
            moveType = MoveType.PREV;
            afterMove = true;
            return MyArrayList.this.elements[cursor];
        }
        public int nextIndex() {
            return cursor;
        }
        public int previousIndex() {
            return cursor -1;
        }
        public void set(E e) {
            if(!afterMove) return;
            if(moveType == MoveType.NEXT)
                MyArrayList.this.set(cursor-1, e);
            else
                MyArrayList.this.set(cursor, e);
        }
        public void add(E e) {
            if(!afterMove) return;
            if(moveType == MoveType.NEXT)
                MyArrayList.this.add(cursor-1, e);
            else
                MyArrayList.this.add(cursor, e);
            cursor += 1;
            afterMove = false;
        }
    }//End of MyListIterator
    //Here: more code of SLinkedList
}//End of SLinkedList
```

Figure 31: Class **SLinkedList**: Inner class **MyListIterator**

.

4 Double-linked List

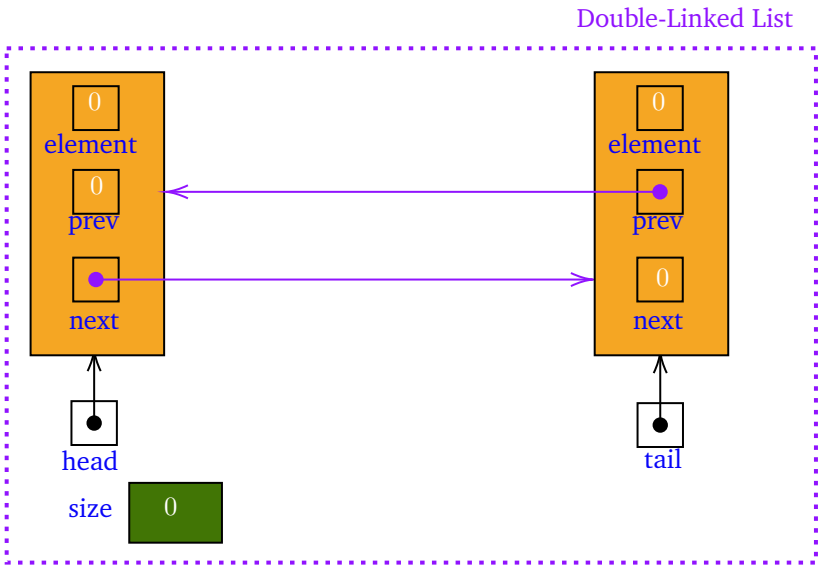


Figure 32: Dobule-linked list: An empty list

.

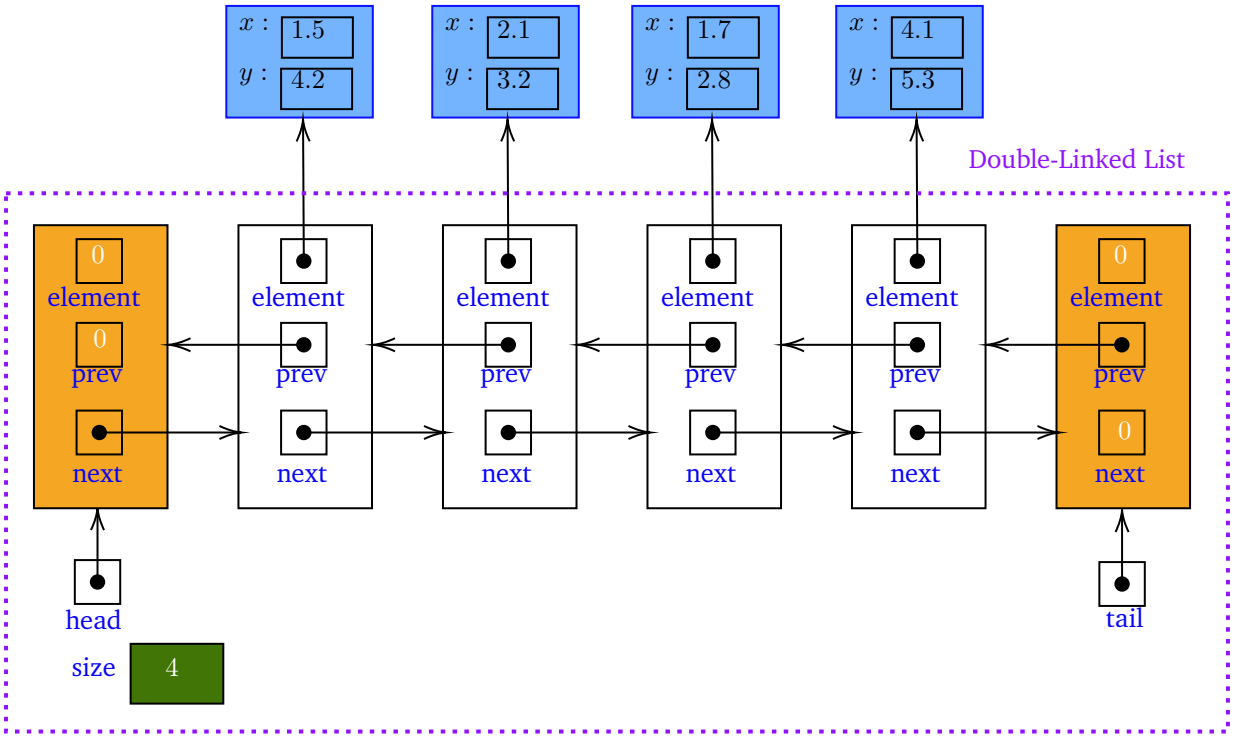


Figure 33: Dobule-linked list: An non-empty list

.

4.1 Insertion operation

Objective: insert a new element, e.g. point (2.1, 3.2) at index = 1 (i.e., = index of point (1.7, 2.8))

Step 1: Find **curNode (containing point (1.7, 2.8))** on thethe list:

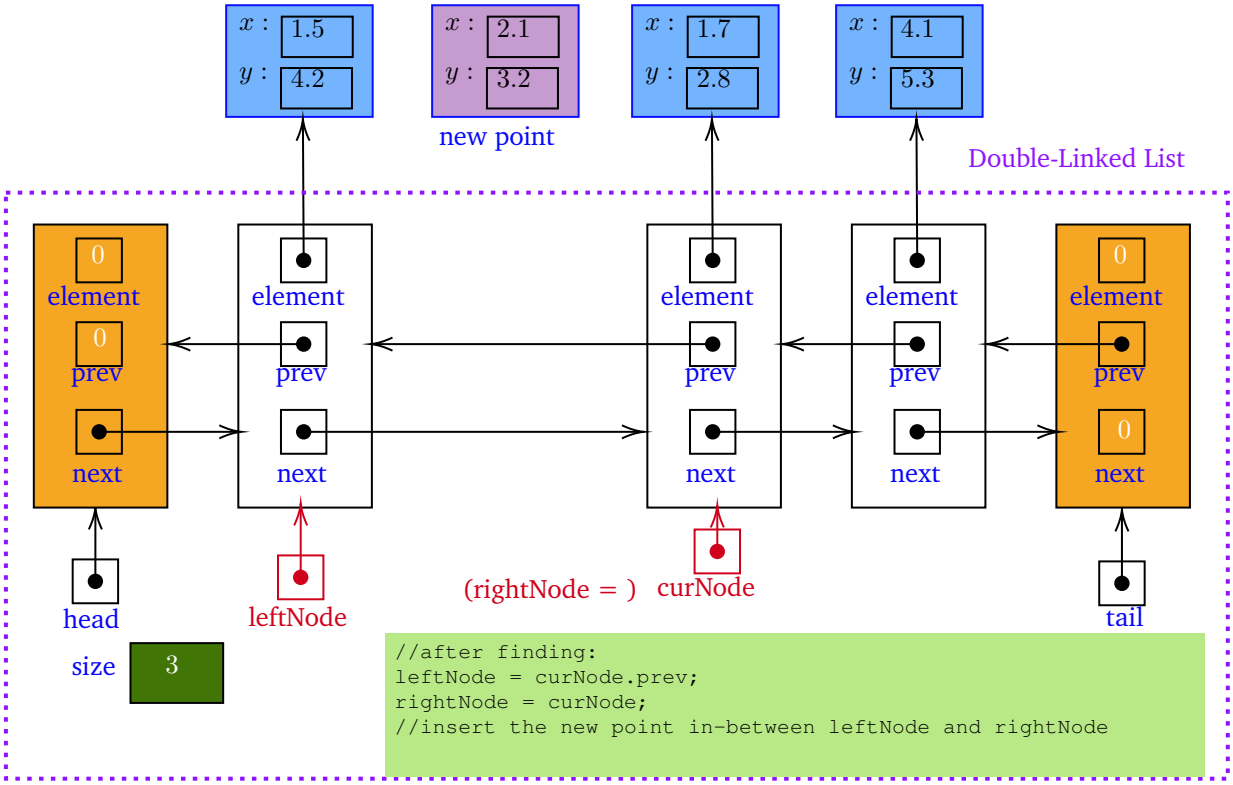


Figure 34: Dobule-linked list: Insertion operation - Step 1

.

Objective: insert a new element, e.g. point (2.1, 3.2) at index = 1 (i.e., = index of point (1.7, 2.8))

Step 2: add the new element in-between leftNode and rightNode:

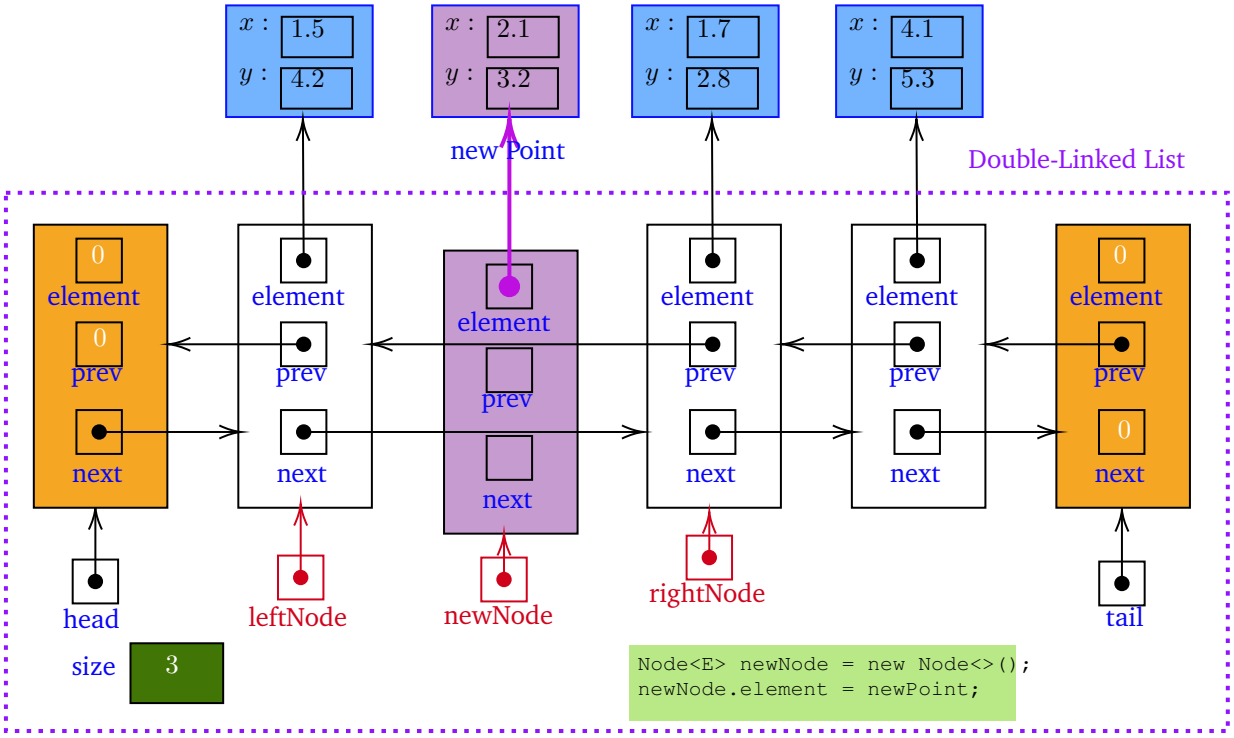


Figure 35: Dobule-linked list: Insertion operation - Step 2.1

.

Objective: insert a new element, e.g. point (2.1, 3.2) at index = 1 (i.e., = index of point (1.7, 2.8))

Step 2: add the new element in-between leftNode and rightNode:

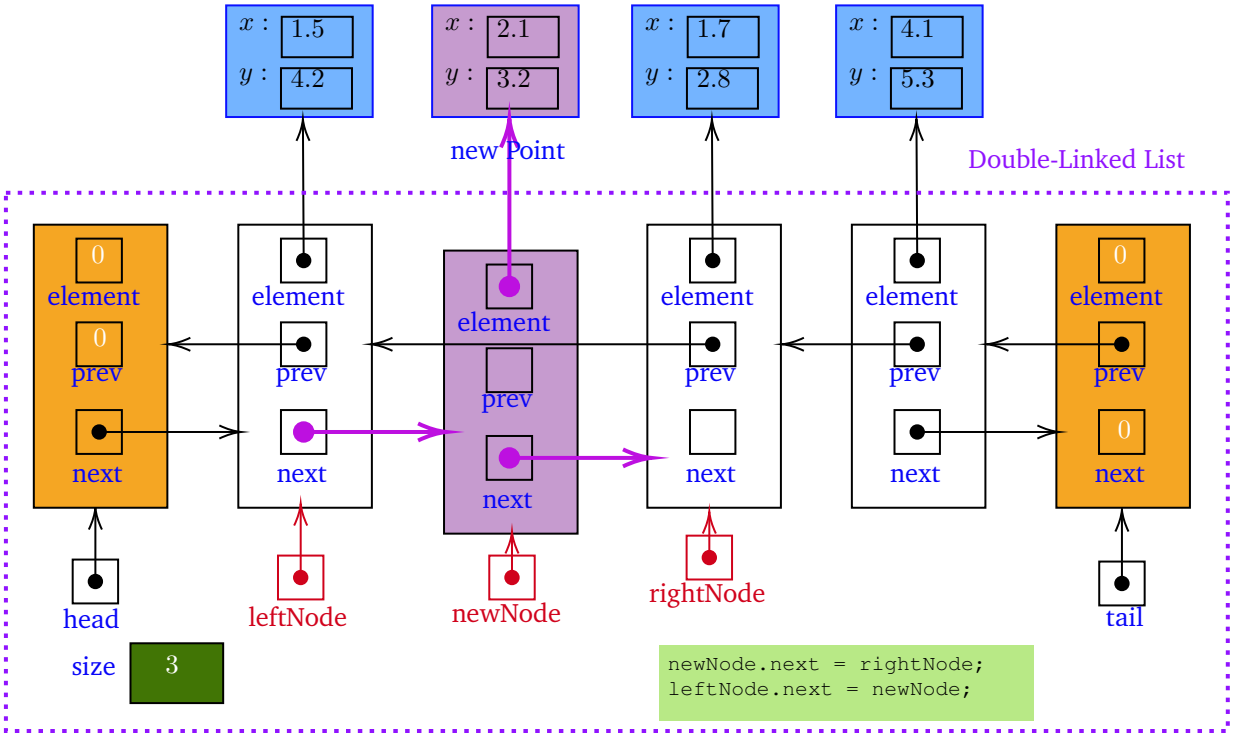


Figure 36: Dobule-linked list: Insertion operation - Step 2.2

.

Objective: insert a new element, e.g. point (2.1, 3.2) at index = 1 (i.e., = index of point (1.7, 2.8))

Step 2: add the new element in-between leftNode and rightNode:

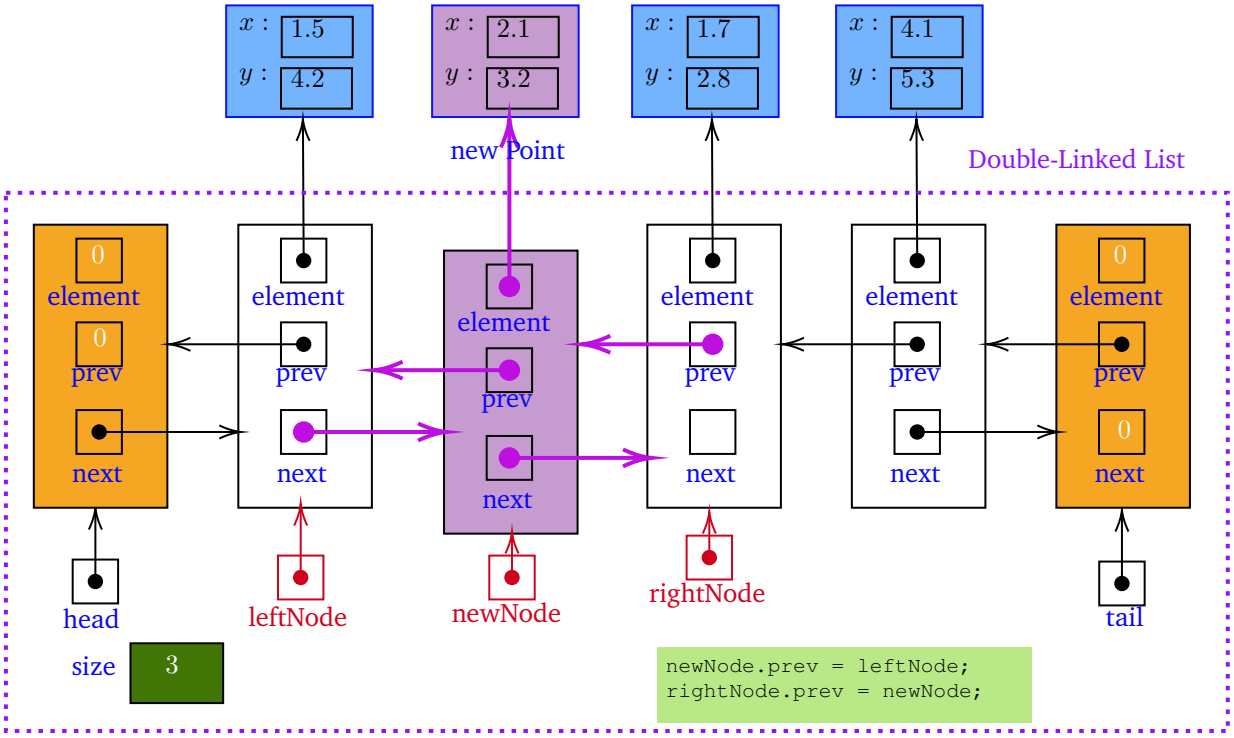


Figure 37: Dobule-linked list: Insertion operation - Step 2.3

.

Objective: insert a new element, e.g. point (2.1, 3.2) at index = 1 (i.e., = index of point (1.7, 2.8))

Step 2: add the new element in-between leftNode and rightNode:

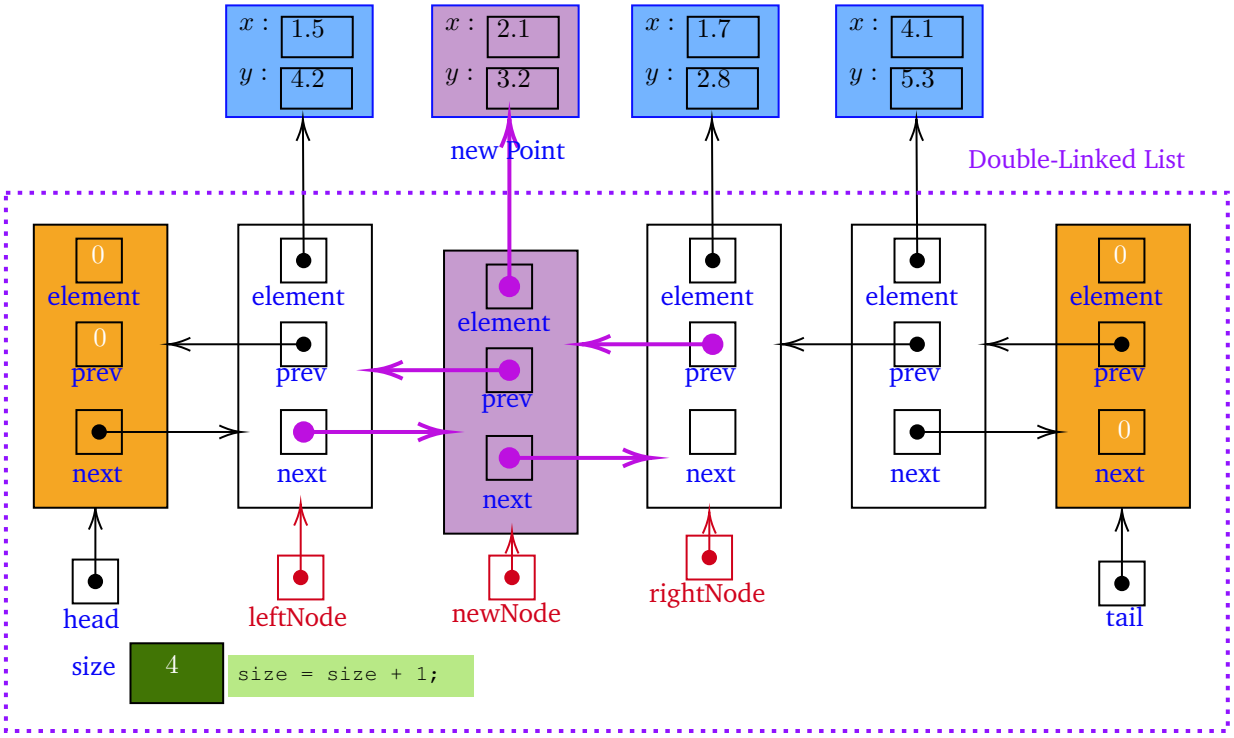


Figure 38: Dobule-linked list: Insertion operation - Step 2.4

.

4.2 Removal operation

Objective: remove an element, e.g. point (2.1, 3.2) at index = 1

Step 1: Find **curNode**:

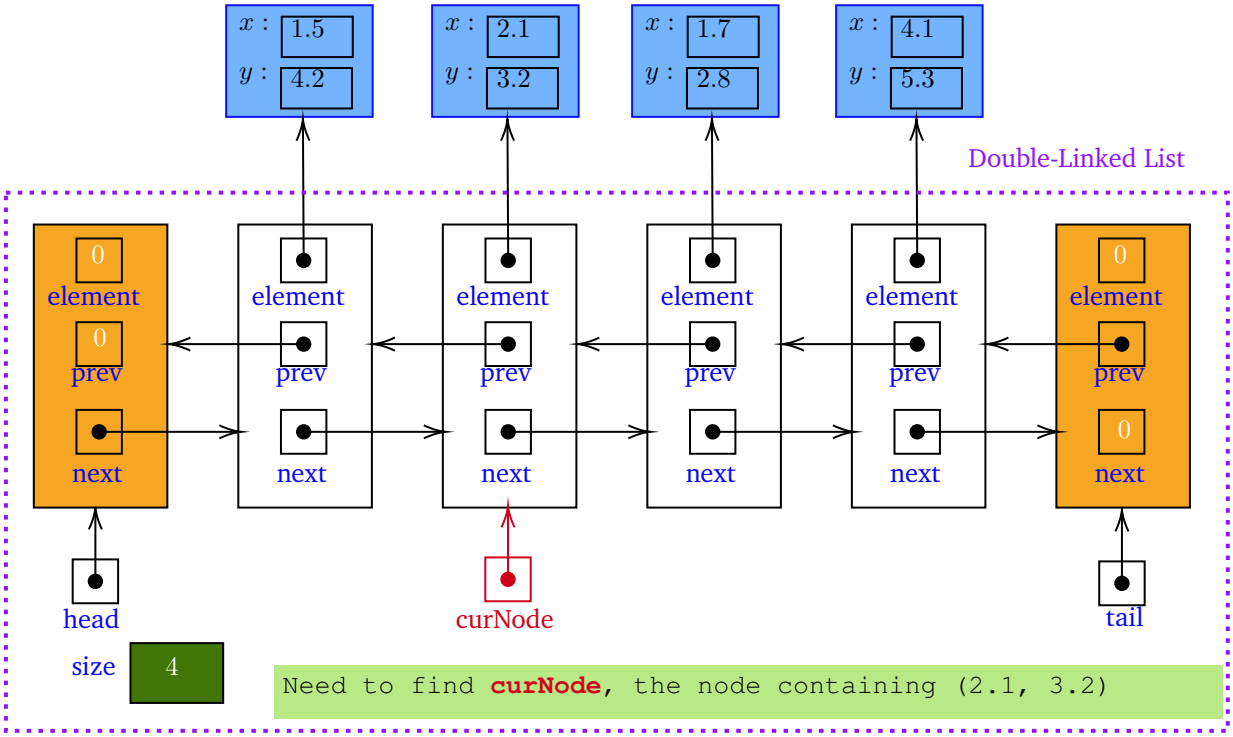


Figure 39: Dobule-linked list: Removal operation - Step 1

.

Objective: remove an element, e.g. point (2.1, 3.2) at index = 1

Step 2: Detach **curNode** from the list:

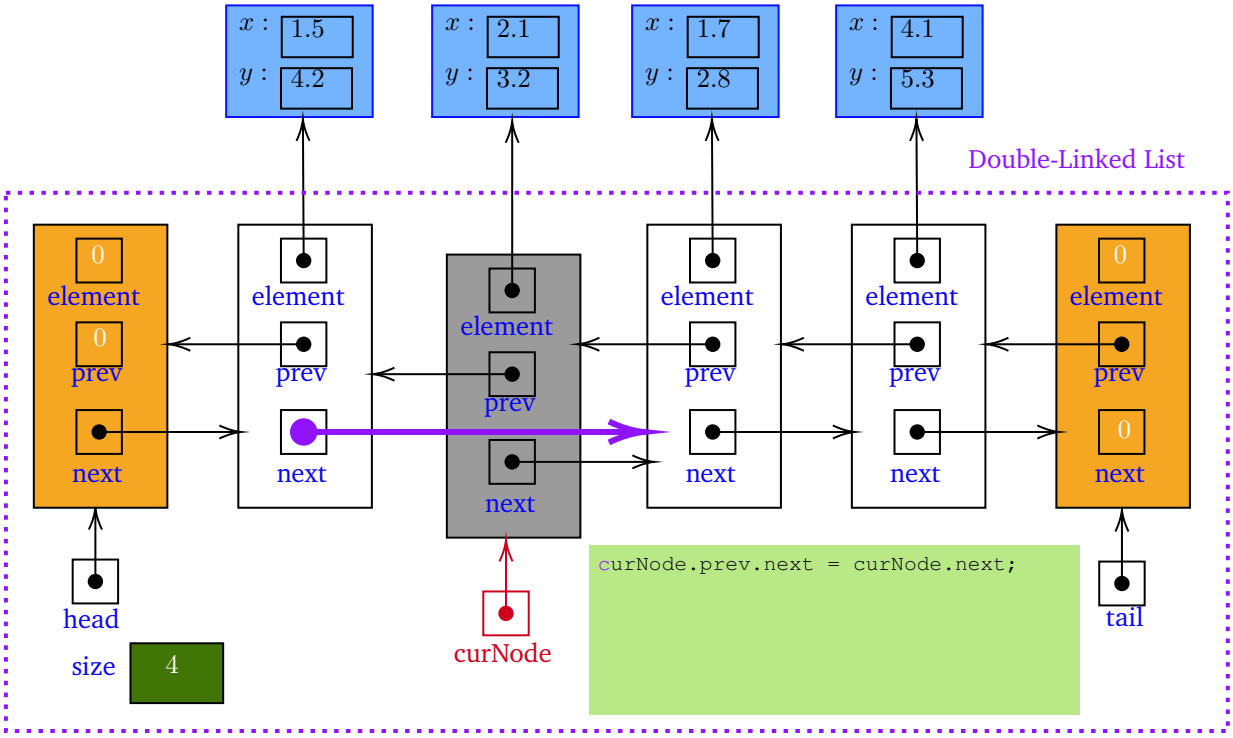


Figure 40: Dobule-linked list: Removal operation - Step 2.1

.

Objective: remove an element, e.g. point (2.1, 3.2) at index = 1

Step 2: Detach **curNode** from the list**:**

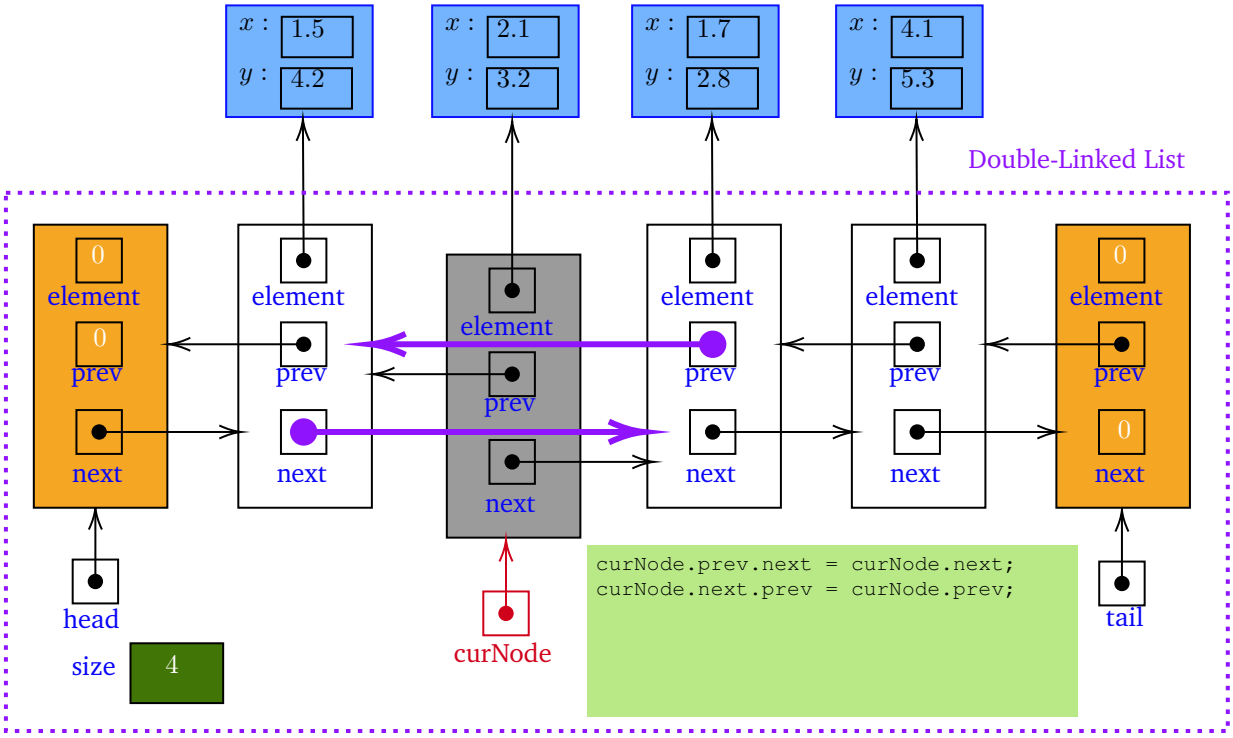


Figure 41: Dobule-linked list: Removal operation - Step 2.2

.

Objective: remove an element, e.g. point (2.1, 3.2) at index = 1

Step 2: Detach **curNode** from the list:

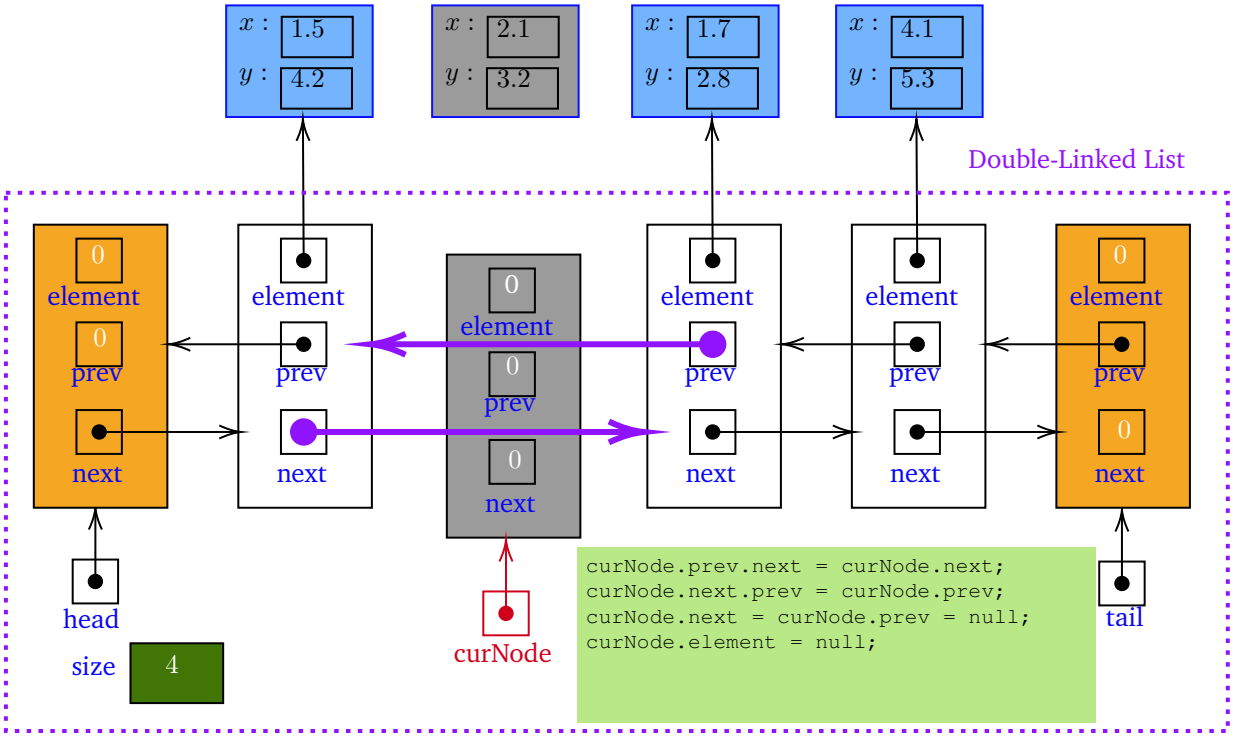


Figure 42: Dobule-linked list: Removal operation - Step 2.3

.

Objective: remove an element, e.g. point (2.1, 3.2) at index = 1

Step 2: Detach **curNode** from the list**:**

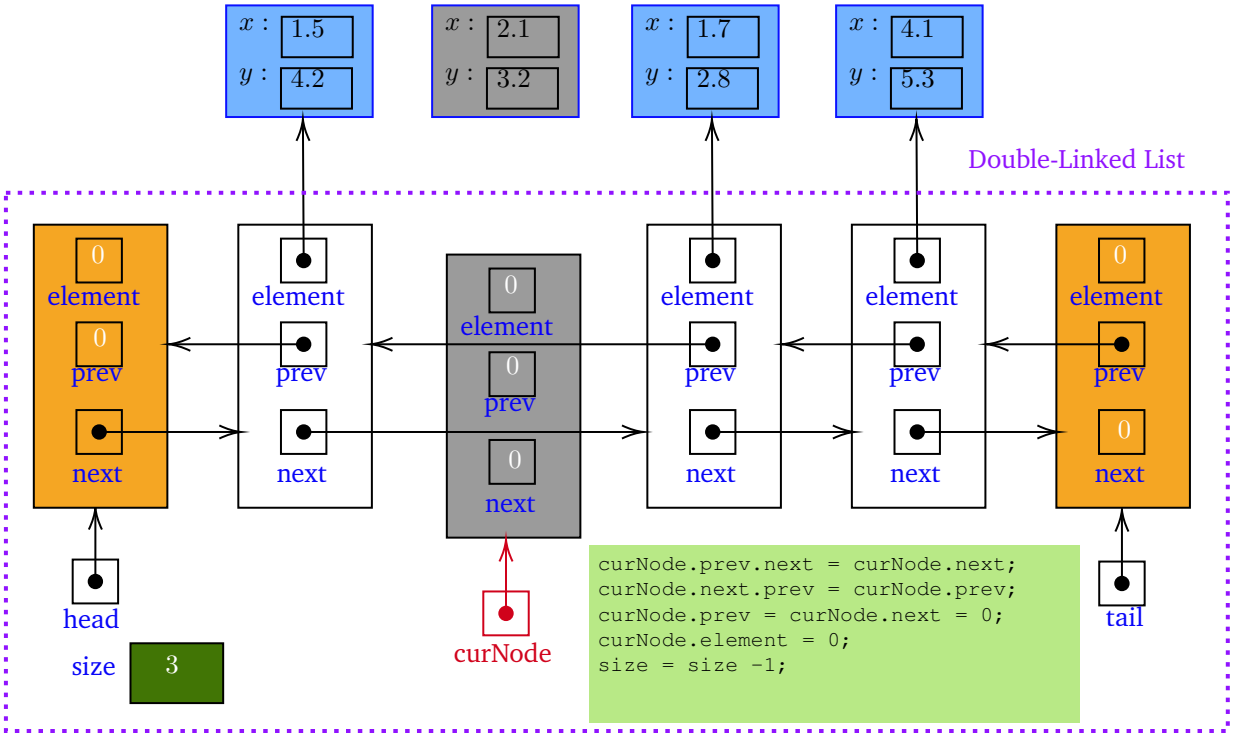


Figure 43: Dobule-linked list: Removal operation - Step 2.4

.