

A Fistful of Pointers: examples with doubly-linked lists

Programming in Java
Sergio Gutierrez-Santos



Note / disclaimer: these slides may be shared as a PDF for the sake of interoperability, but they are intended to be seen as an animation. It is recommended to disable the “Continuous View” feature of your PDF reader so you can see one slide at a time.

Legal stuff: these slides are shared under a CC by-sa license.

This means that you are free to share (copy and redistribute the material in any medium or format) and adapt (remix, transform, and build upon the material for any purpose, even commercially). The licensor cannot revoke these freedoms as long as you follow the license terms, under the following terms: Attribution (you must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use) and ShareAlike (if you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original). No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

The following examples will illustrate
some basic operations over
dynamic linked lists

The following examples will illustrate
some basic operations over
dynamic linked lists
(aka pointer-based lists)

We will use a school as a example

(a school for children, not of fish)

For the sake of simplicity,
we will assume that a school
has just one list of children.

For the sake of simplicity,
we will assume that a school
has just one list of children.

(In a realistic scenario,
a school would have several
lists, probably one per year group)

For the sake of completeness,
our list of student will be doubly-linked.
The singly-linked case is simpler.

Our main data structure unit,

Our main data structure unit,
representing the children,

Our main data structure unit,
representing the children,
looks like this:

```
public class Student {  
    private String name;  
    private int year;  
    private Student next;  
    private Student prev;
```

```
public class Student {  
    private String name;  
    private int year;  
    private Student next;  
    private Student prev;  
}
```

(In this and all the other subsequent code fragments, comments are omitted to save space, but in a real application they should be there explaining what each field is for and what every method does)

Let's add accesors (*getters*)
and mutators (*setters*)
to our student!

```
public String getName() {  
    return name;  
}  
  
public int getYear() {  
    return year;  
}  
  
public Student getNext() {  
    return next;  
}  
  
public void setNext(Student nextStudent) {  
    this.next = nextStudent;  
}  
  
public Student getPrev() {  
    return prev;  
}  
  
public void setPrev(Student prevStudent) {  
    this.prev = prevStudent;  
}  
}
```


And a constructor method, of course!

```
public Student(String name, int year) {  
    this.name = name;  
    this.year = year;  
    this.next = null;  
    this.prev = null;  
}
```

Now it is time to create the school

When created, the school
has an empty list of students.

```
public class School {  
    private Student studentListStart = null;  
    public School() {  
        this.studentListStart = null;  
    }  
}
```

Now it is time to implement
the basic features:
adding and deleting students.

We will also calculate the length of the list and print all its elements, for good measure.

As you know, this can be done
both iteratively and recursively.

We are going to do it both ways.

Phase I : iterative approach

Phase I : iterative approach
Step I : addition to the list

The time has come to enrol some students!

```
public class School {  
  
    private Student studentListStart = null;  
  
    public School() {  
        this.studentListStart = null;  
    }  
  
    public void enrolStudent(String name, int year) {  
  
        /* Write your code here */  
  
    }  
}
```

The first (trivial) step is to create
the new student

```
Student newStudent = new Student(name, year);
```

Once we have the object
of type Student,
we can add it to the list.

For the sake of simplicity, we will assume that students are always added at the end of the list.

This means that we have to
care about two cases only:

1) The list is empty,
so we create the list

2) The list is not empty,
So we move to the last element
and add the new element after it

The first case (empty list)
is very easy

```
if (this.studentListStart == null) {  
    this.studentListStart = newStudent;  
    return;  
}
```

“If the list is empty,
we create the list”

The second case requires three steps:

i) Setting an auxiliary pointer at the beginning of the list

```
Student current = studentListStart;
```

ii) moving the auxiliary pointer
to the end of the list

```
while(current.getNext() != null) {  
    current = current.getNext();  
}
```

iii) adding the new element there

```
current.setNext(newStudent);  
newStudent.setPrev(current);
```


iii) adding the new element there

```
current.setNext(newStudent);  
newStudent.setPrev(current);
```

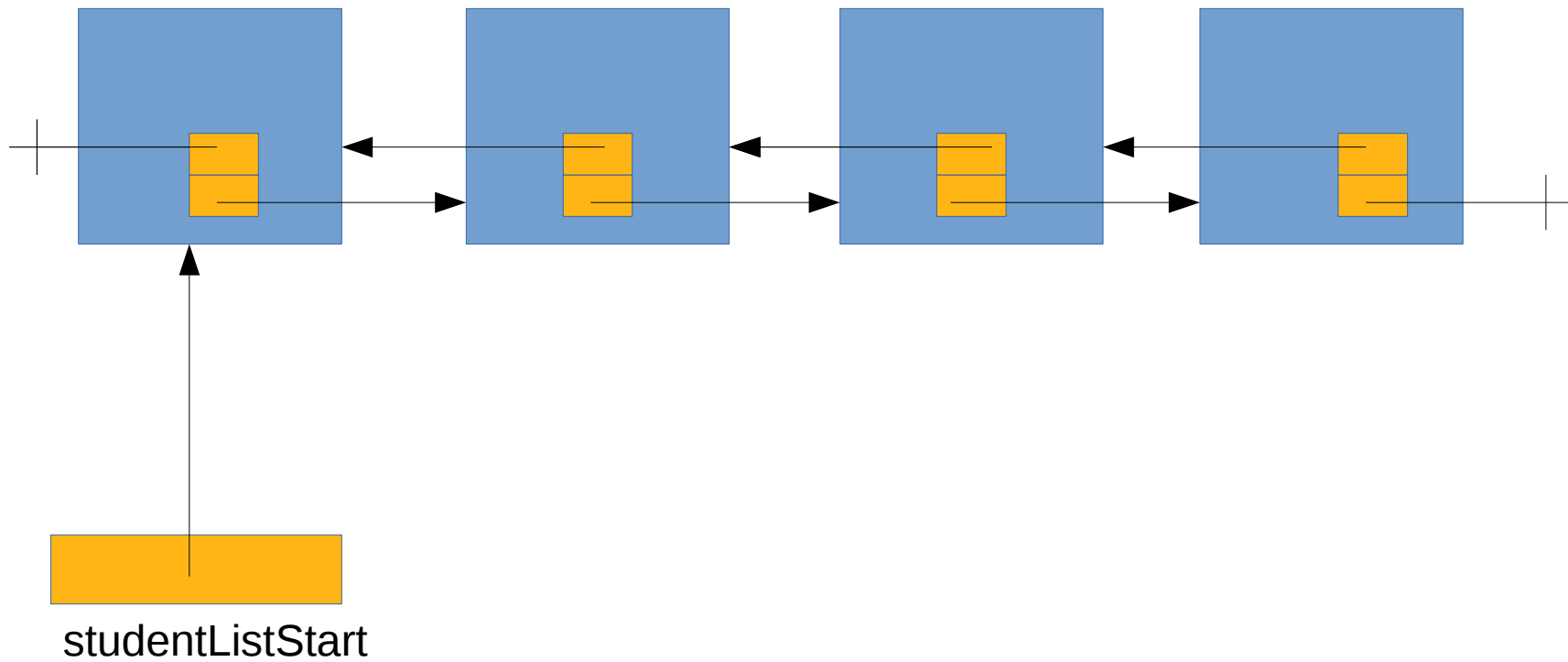
(we point from the last element to the new element, and from the new element backwards to the (previously) last element)

...and that's all, folks!

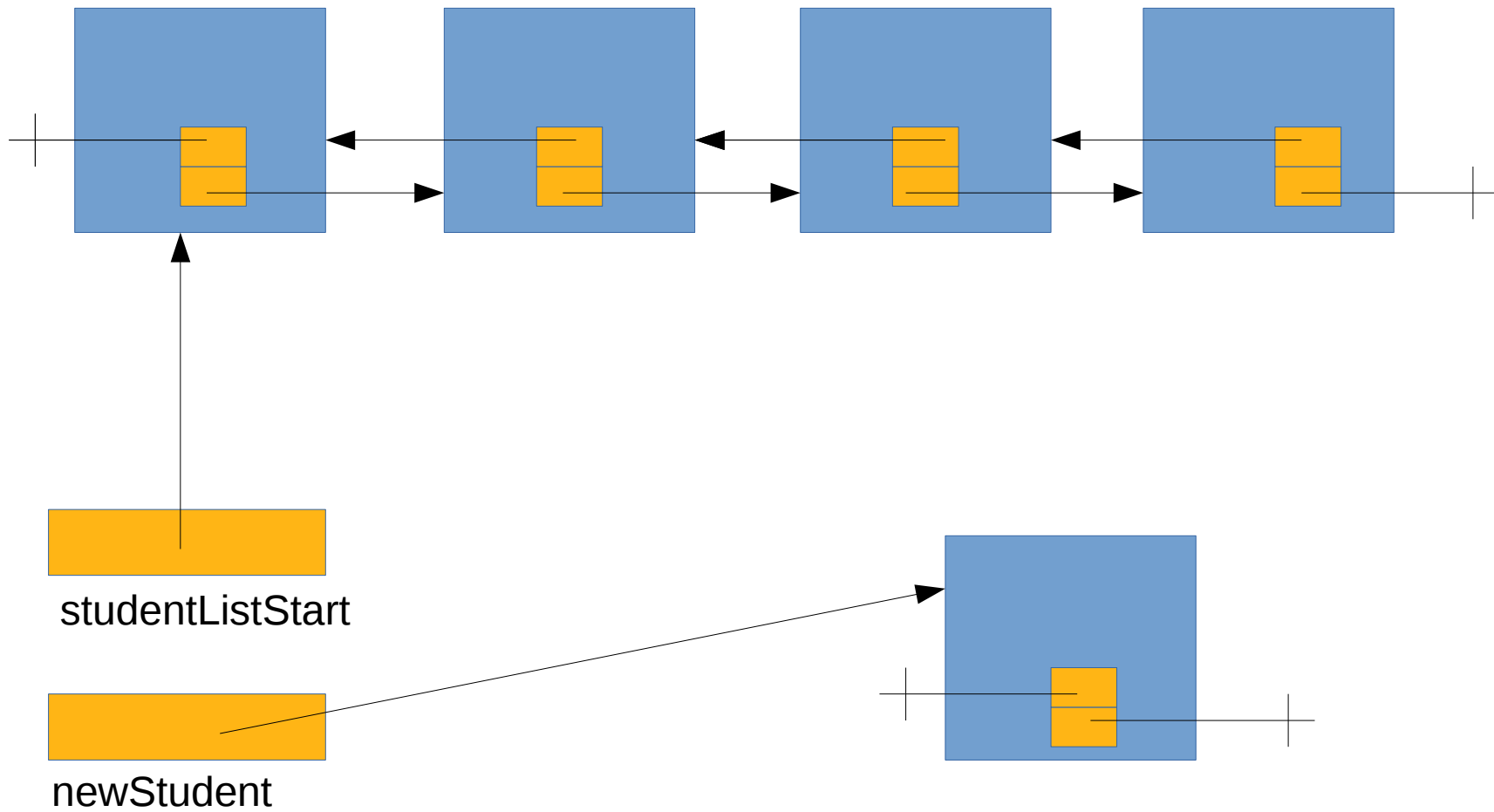
Full code for this method

```
public void enrolStudent(String name, int year) {
    Student newStudent = new Student(name, year);
    if (this.studentListStart == null) {
        this.studentListStart = newStudent;
        return;
    } else {
        Student current = studentListStart;
        while(current.getNext() != null) {
            current = current.getNext();
        }
        current.setNext(newStudent);
        newStudent.setPrev(current);
        return;
    }
}
```

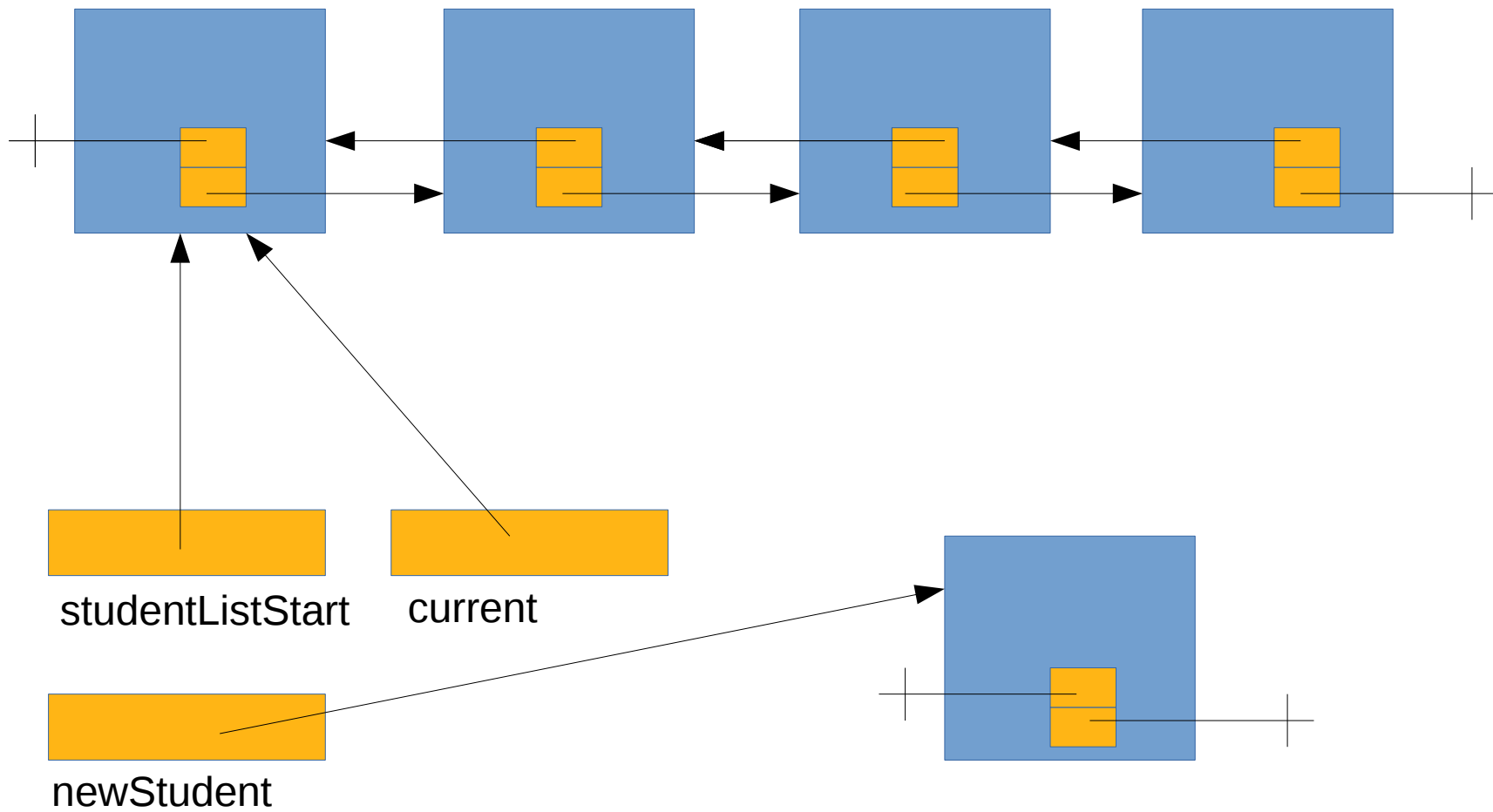
A simplified diagram to represent how this method works could look like this:



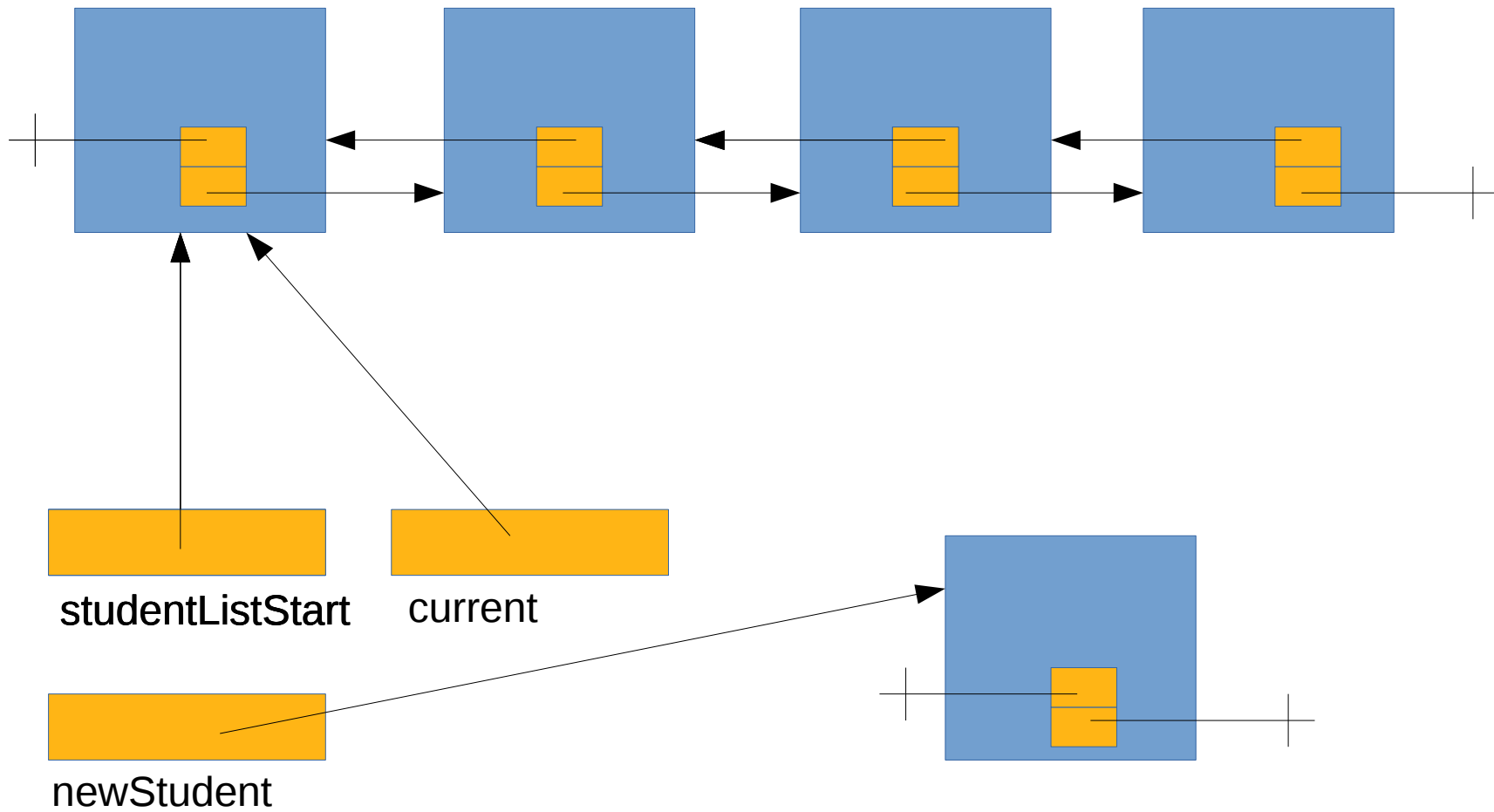
Initial situation with four students. Objects of class Student are shown as blue squares and pointers are shown as yellow squares.



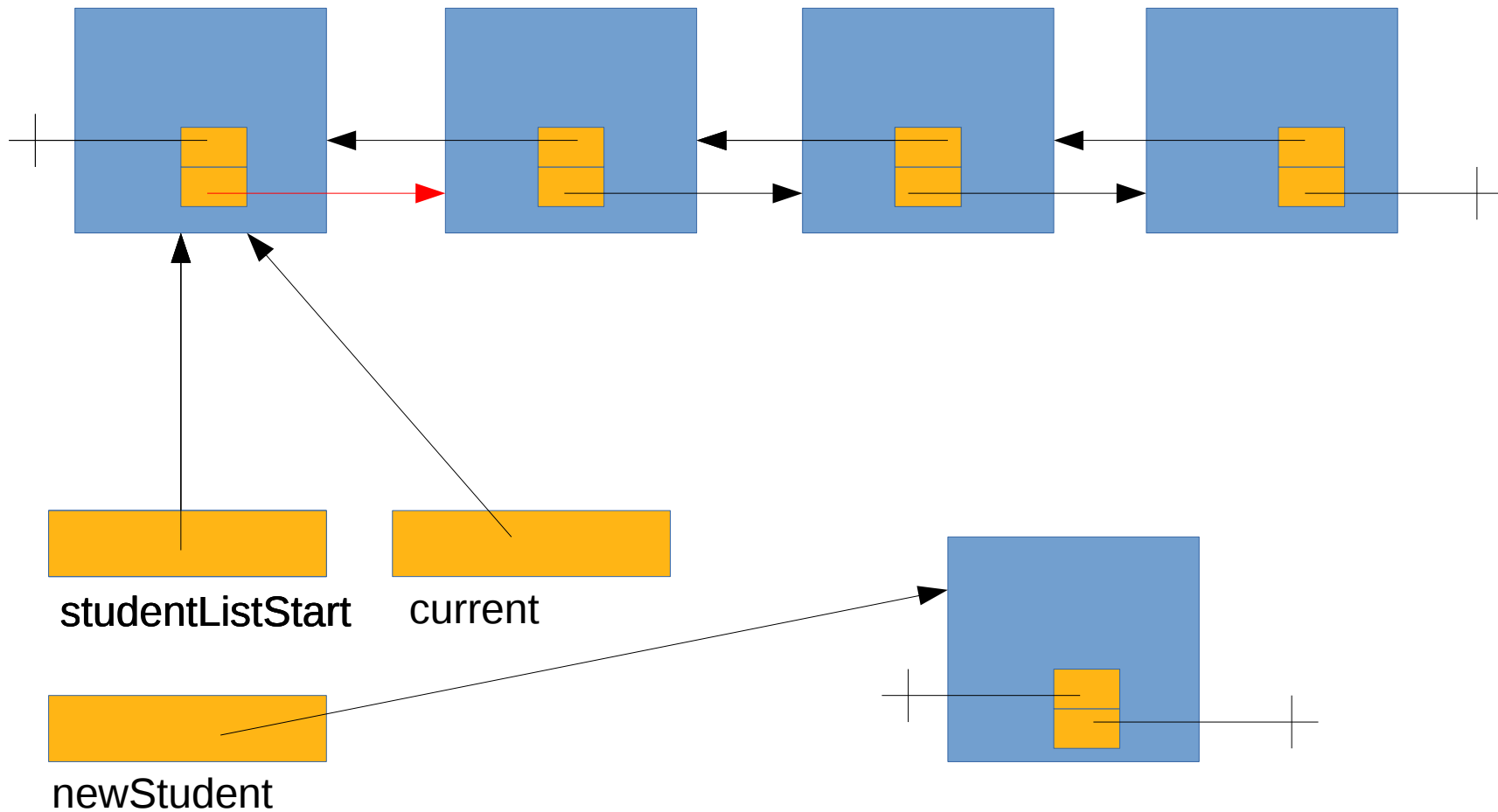
```
Student newStudent = new Student(name, year);
```



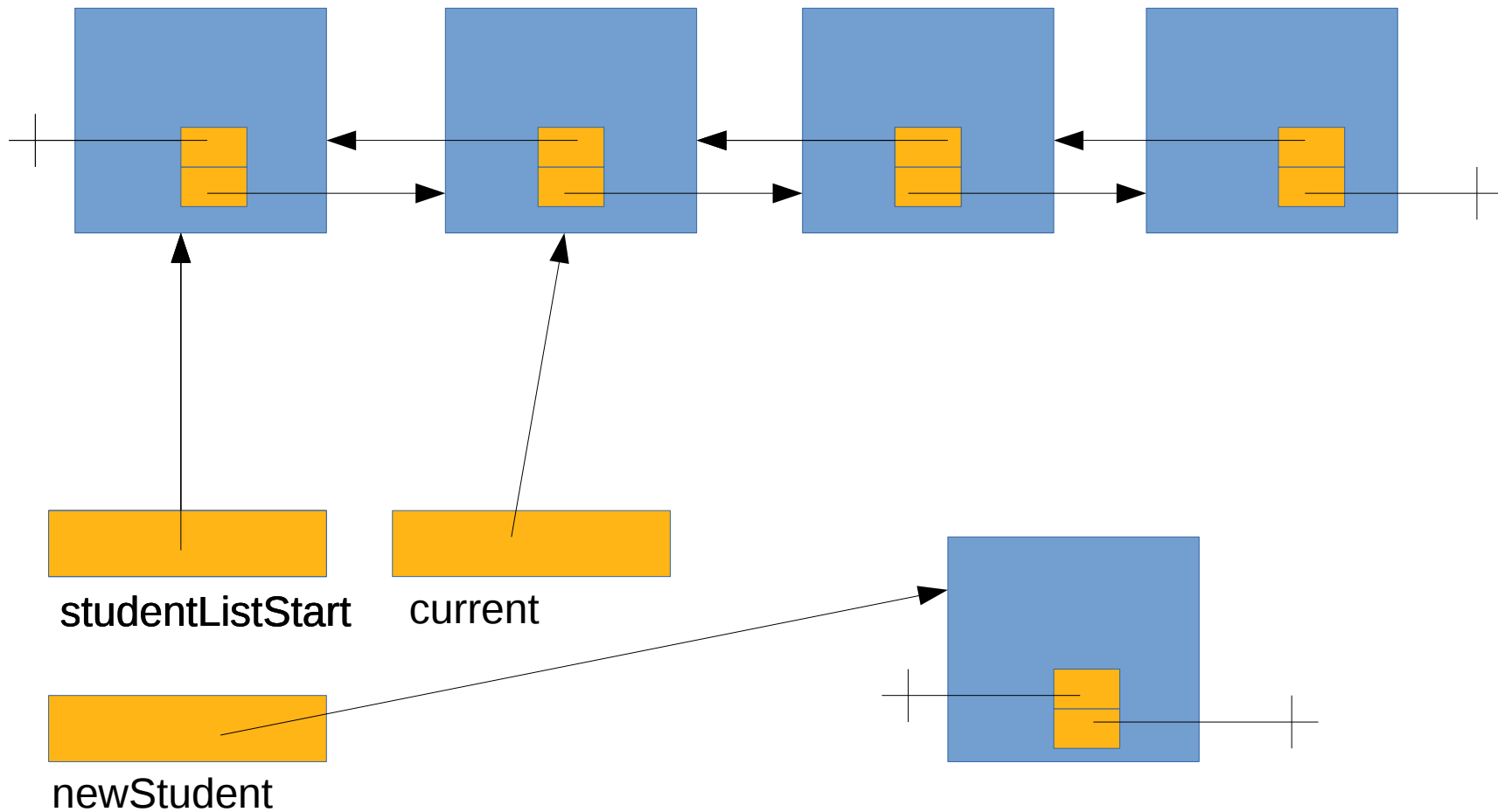
```
Student current = studentListStart;
```



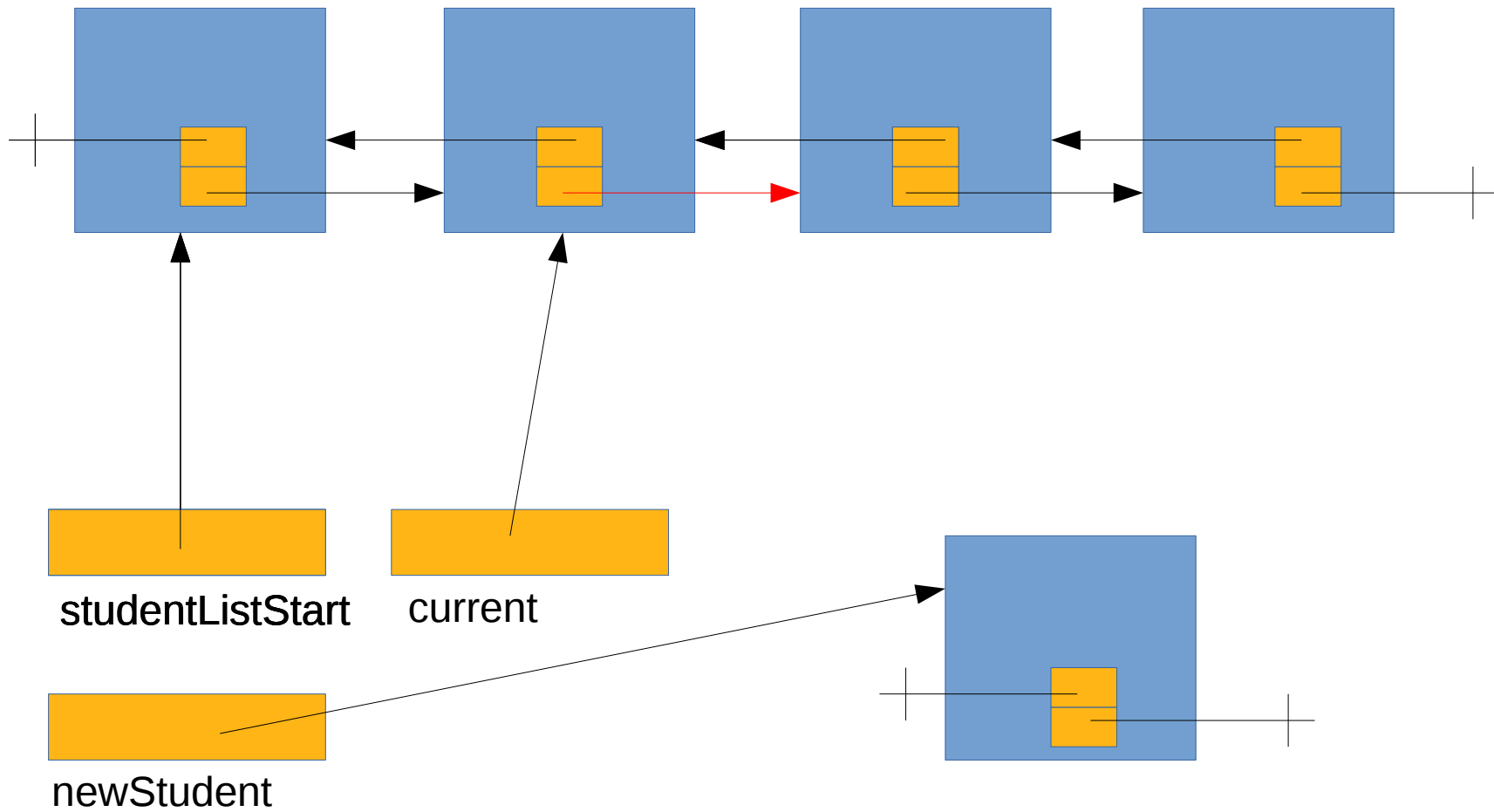
```
while(current.getNext() != null) {  
    current.setNext(current.getNext());  
}
```

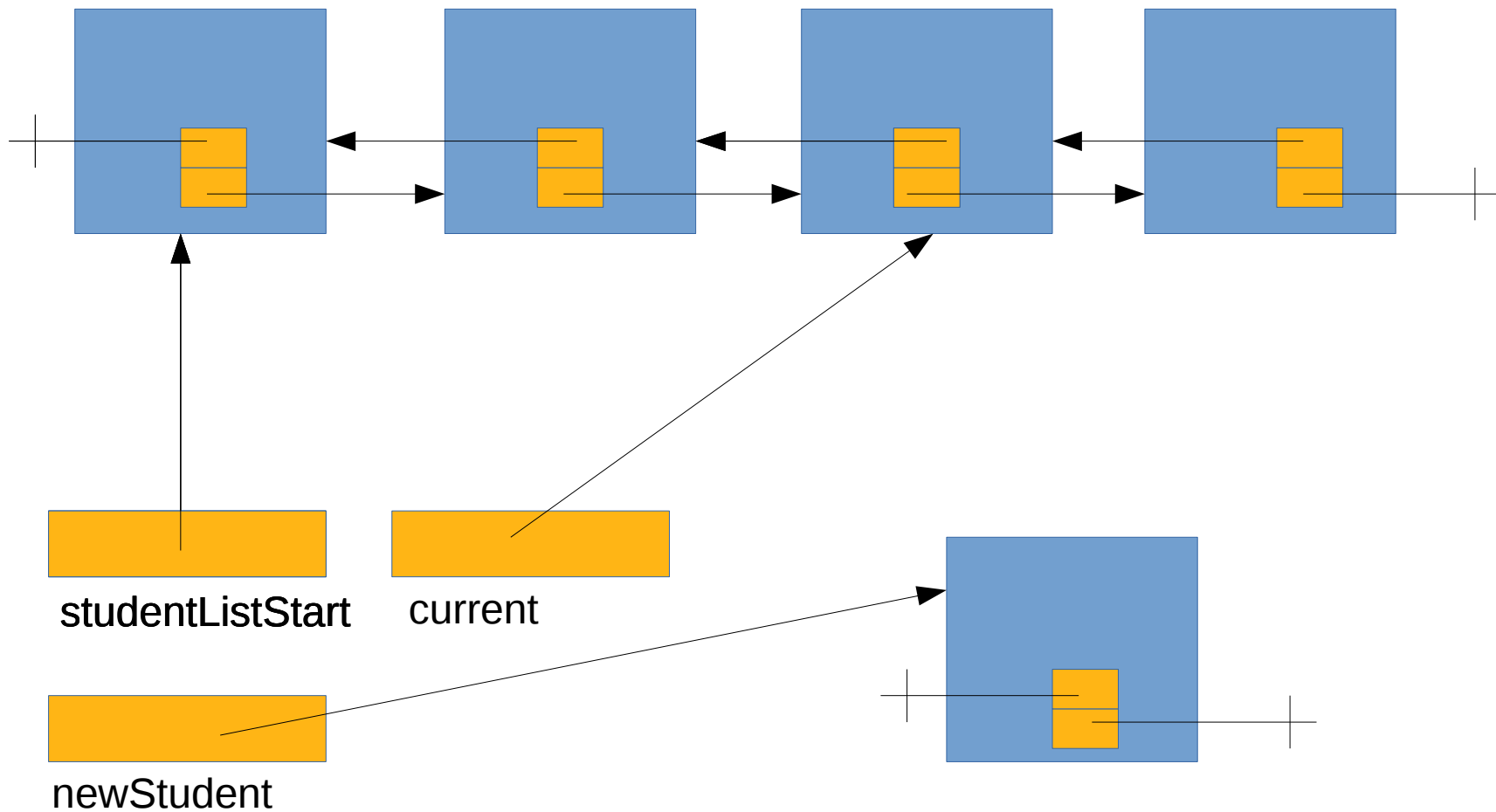
```
while(current.getNext() != null) {  
    current.setNext(current.getNext());  
}
```



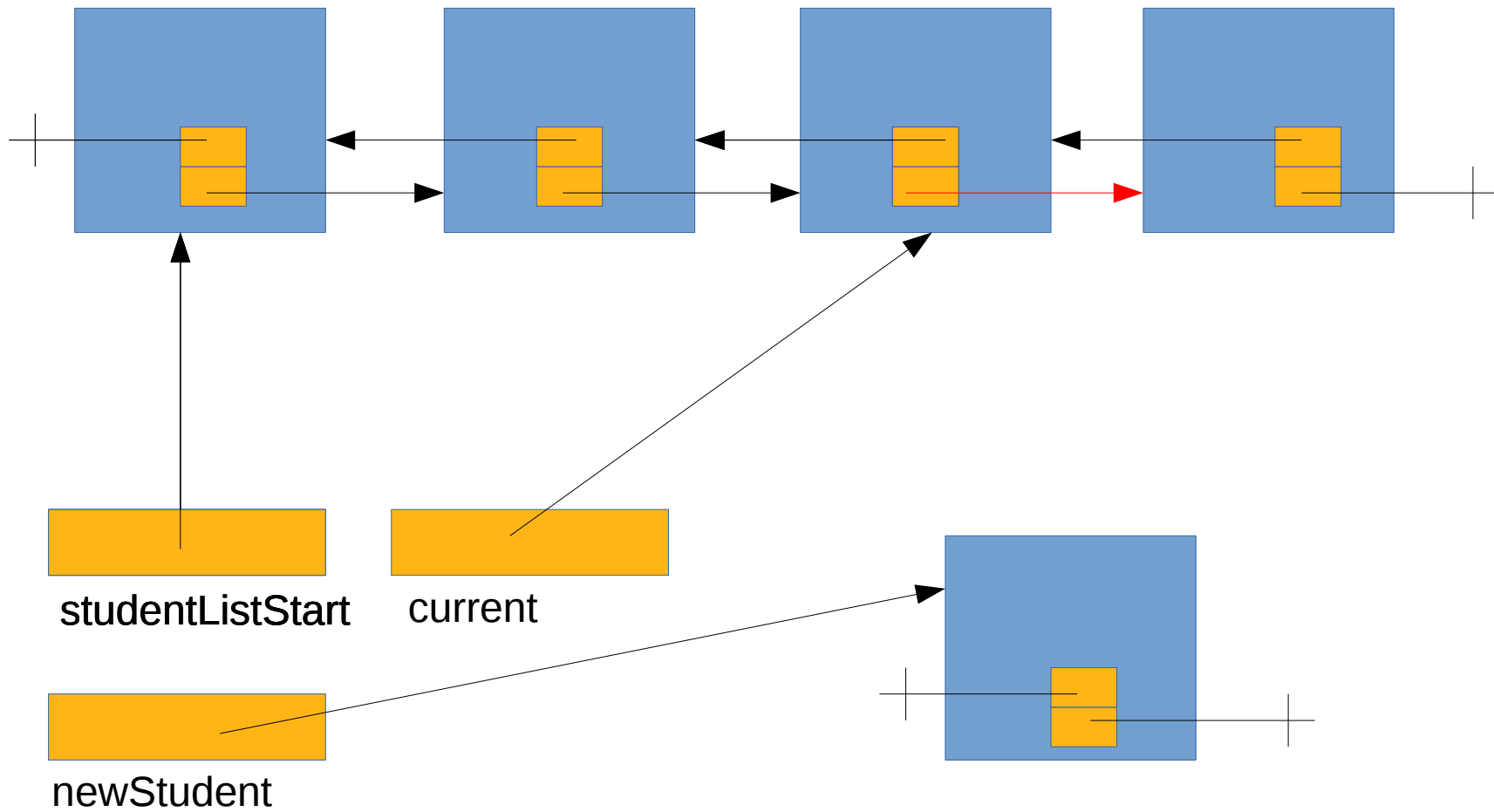
```
while(current.getNext() != null) {  
    current.setNext(current.getNext());  
}
```



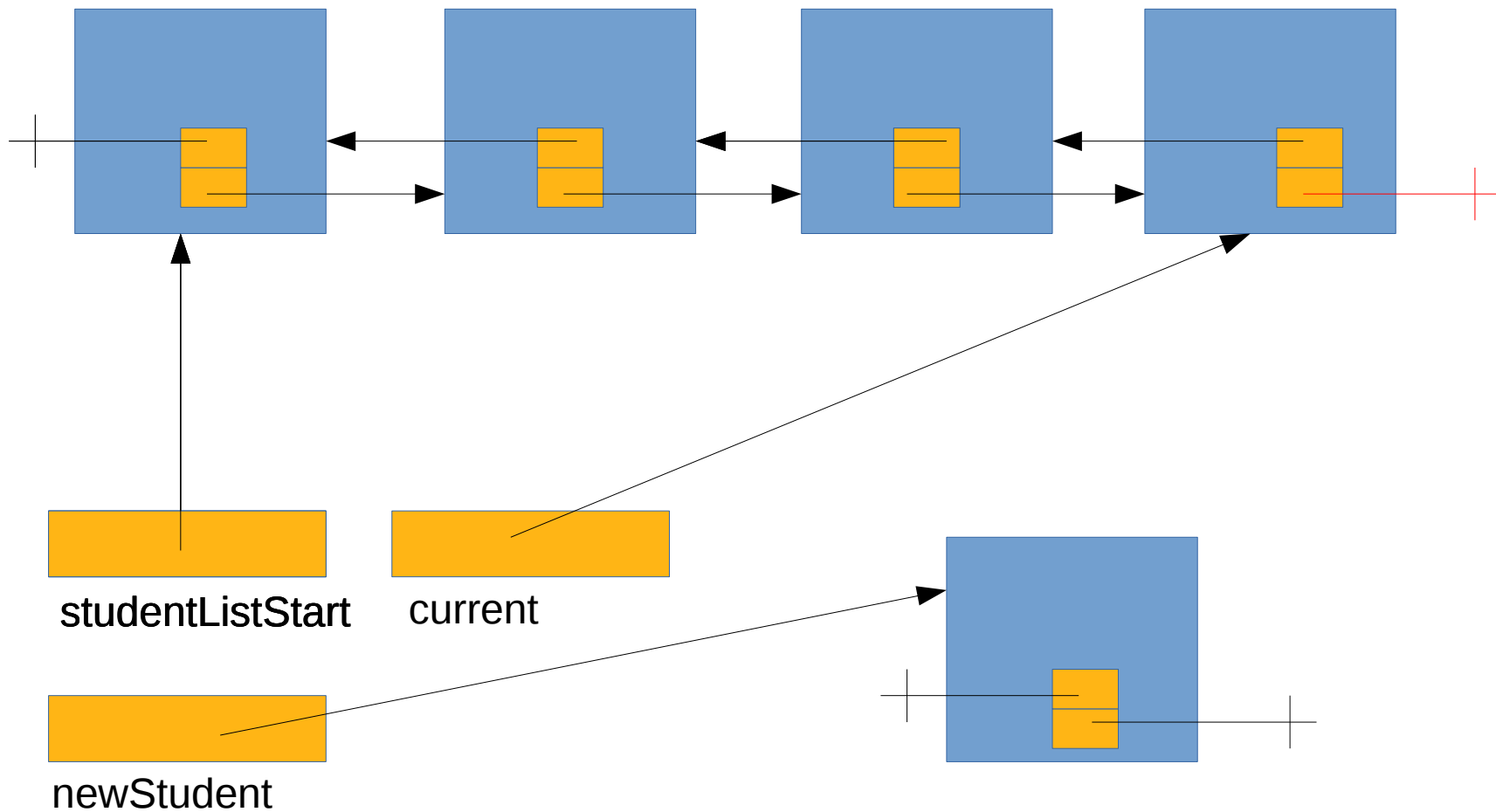
```
while(current.getNext() != null) {  
    current.setNext(current.getNext());  
}
```



```
while(current.getNext() != null) {  
    current.setNext(current.getNext());  
}
```

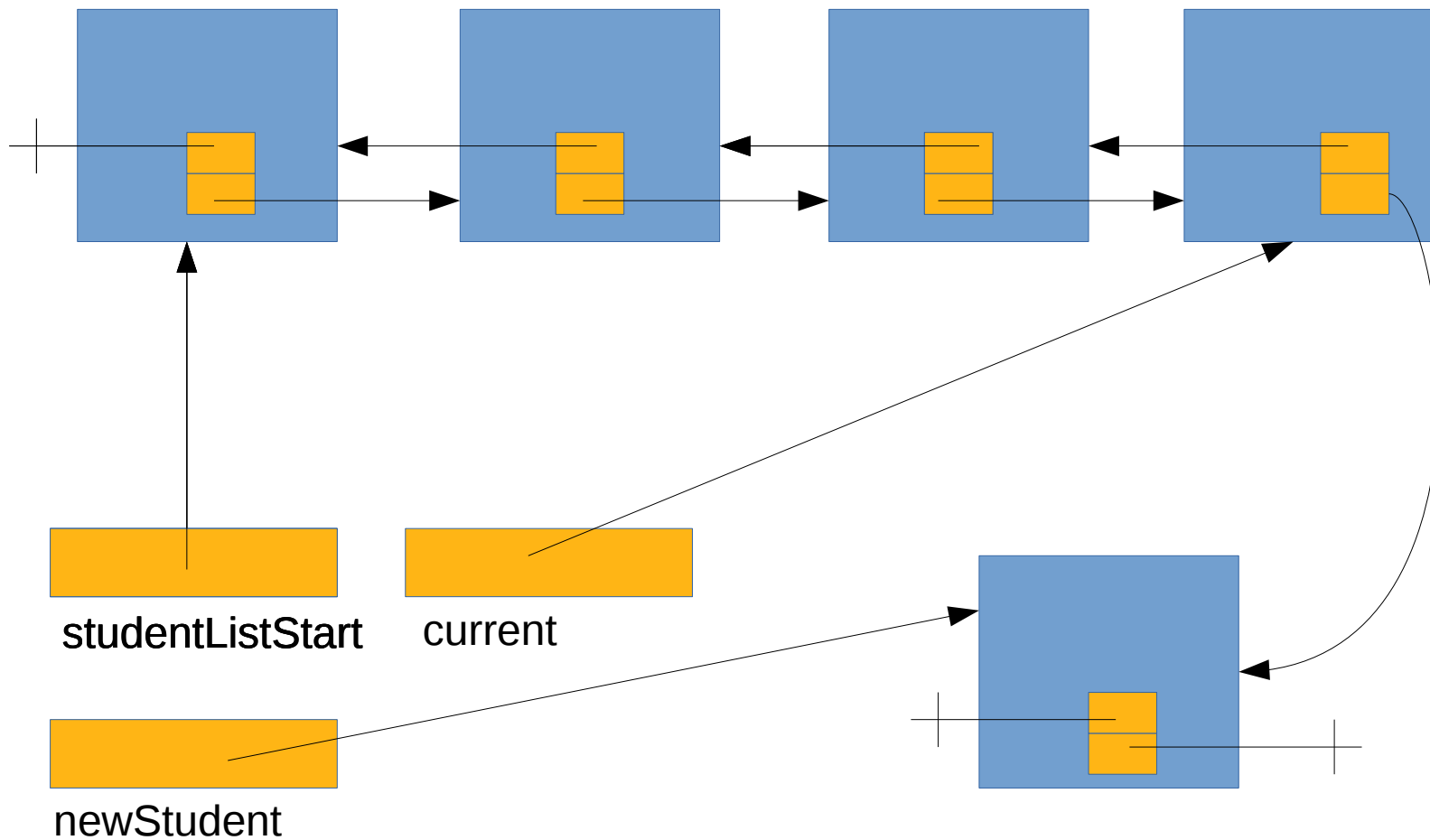


```
while(current.getNext() != null) {  
    current.setNext(current.getNext());  
}
```

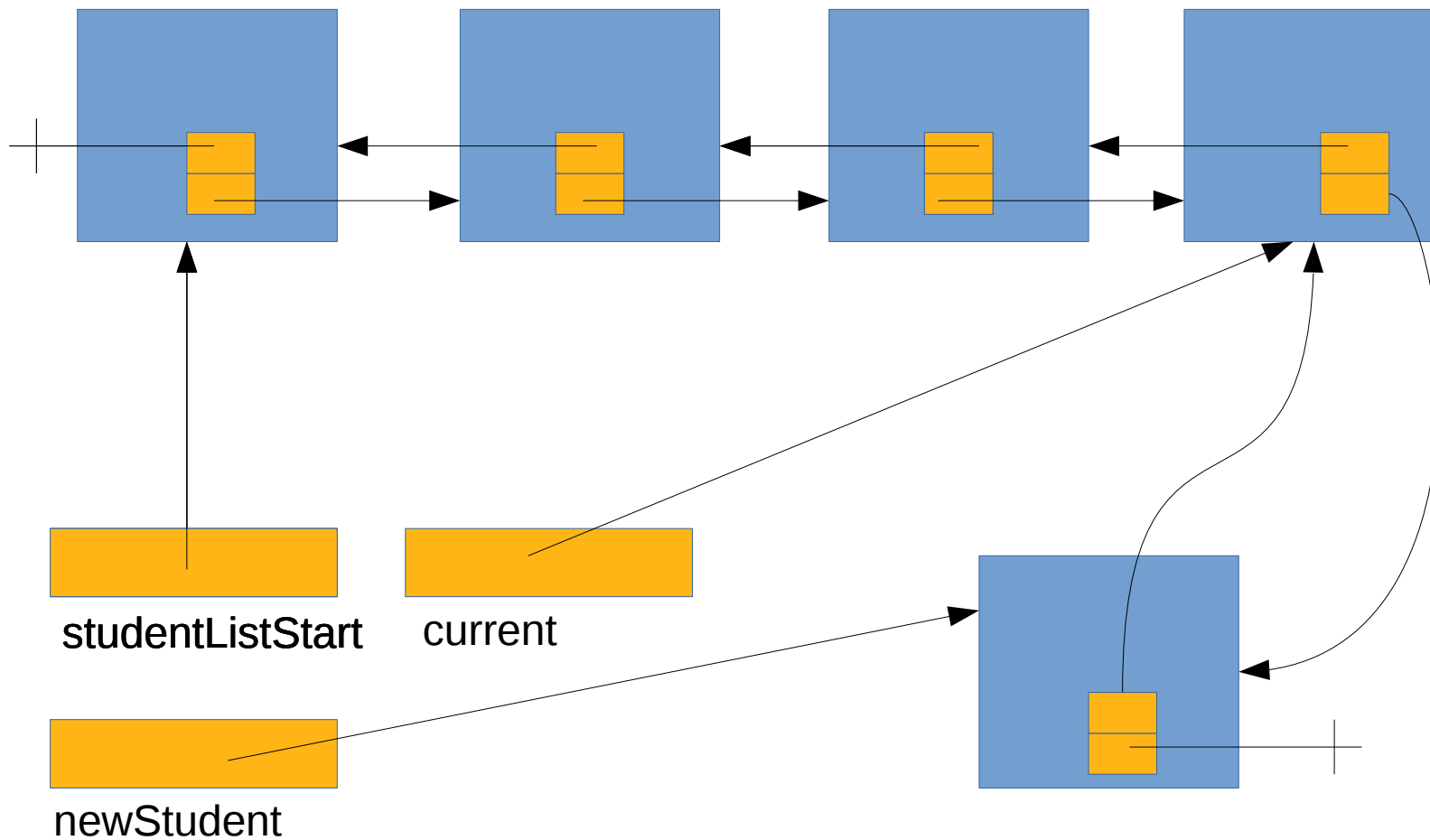


```
while(current.getNext() != null) {  
    current.setNext(current.getNext());  
}
```

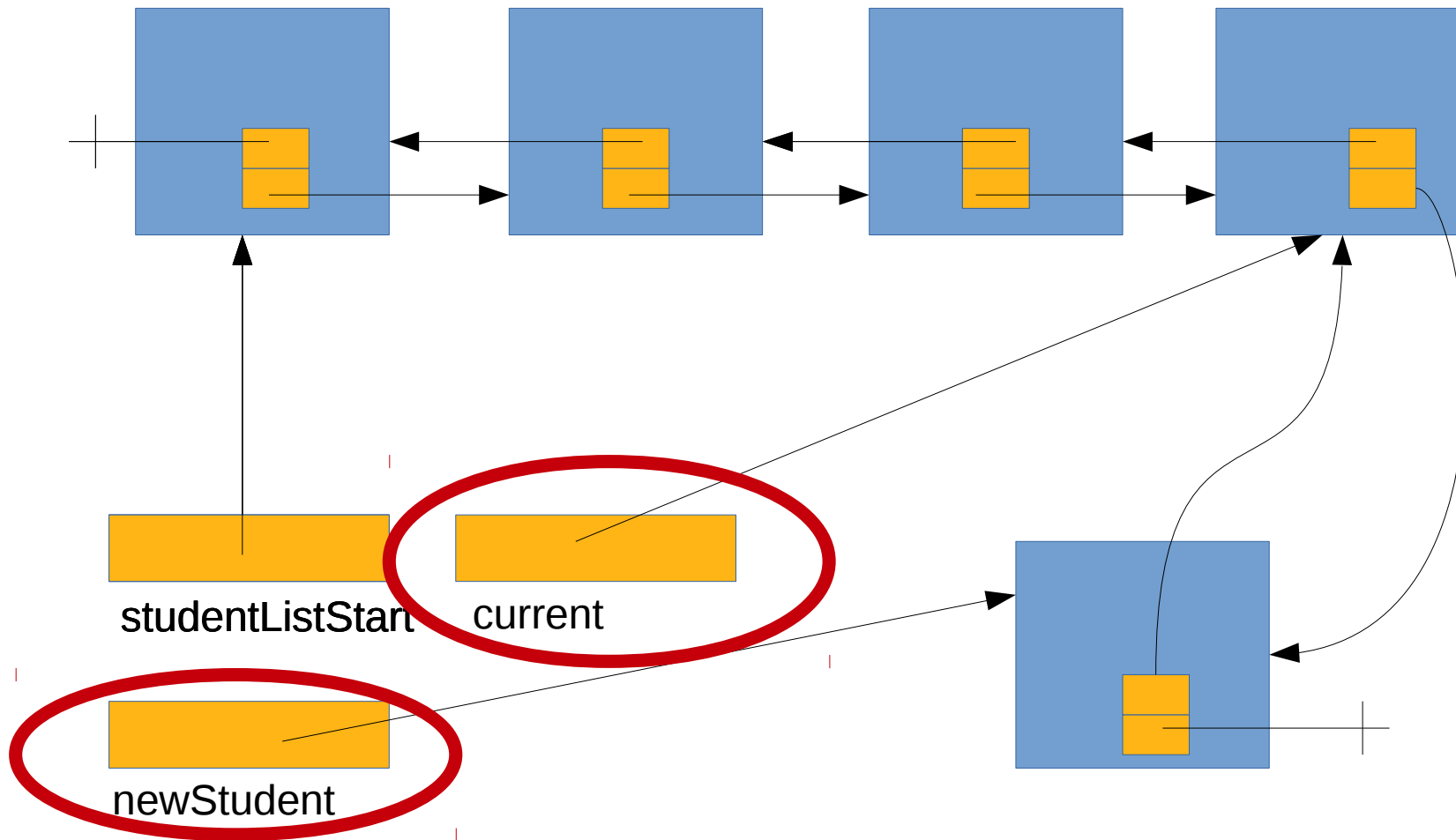
Not true
anymore, so
we exit the
loop!



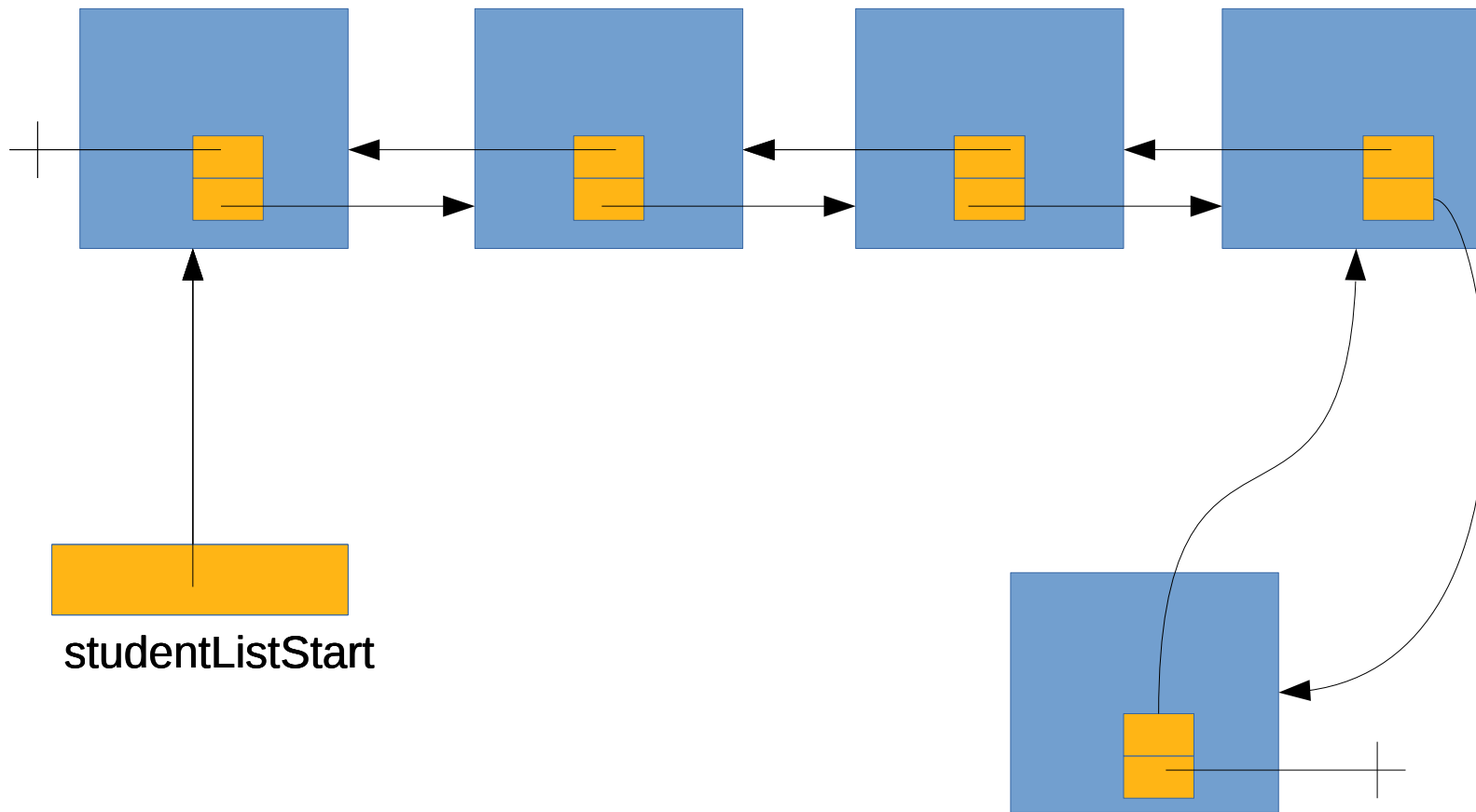
```
current.setNext(newStudent);
```



```
newStudent.setPrev(current);
```

Upon return, these two pointers will disappear because they are local variables and, therefore, they exist on the stack only while the method is running.



Situation after executing the method to add a fifth student.

Phase I : iterative approach

Phase I : iterative approach
Step II : deletion from the list

The next step is deleting students

Assuming that students' names are unique,
the method will look similar to this:

```
public void deleteStudent(String name) {  
    /* Your code here */  
}
```

Assuming that students' names are unique,
the method will look similar to this:

```
public void deleteStudent(String name) {  
    /* Your code here */  
}
```

(If names were not unique, then we would
need to give each student a unique ID,
but the main idea is the same)

Now we need to consider three cases:

Now we need to consider three cases:

- i) the student is the first one

Now we need to consider three cases:

- i) the student is the first one
- ii) the student is somewhere else

Now we need to consider three cases:

- i) the student is the first one
- ii) the student is somewhere else
- iii) the student is not in the list

Now we need to consider three cases:

- i) the student is the first one
- ii) the student is somewhere else
- iii) the student is not in the list


(actually, in the latest case we do not have
to do anything, so we can ignore it)

The first case is easy:

```
if (name.equals(studentListStart.getName())) {  
    studentListStart = studentListStart.getNext();  
    studentListStart.setPrev(null);  
}
```

The first case is easy:

```
if (name.equals(studentListStart.getName())) {  
    studentListStart = studentListStart.getNext();  
    studentListStart.setPrev(null);  
}
```



This second line is very important. The first student cannot have anything before it! We must delete the link to the former first student.

The second case involves

once again

to traverse the list
from the beginning to the end

but this time we must check at every step whether we have found the element to delete.

a) Setting an auxiliary pointer at the beginning of the list

```
Student current = studentListStart;
```

b) moving the auxiliary pointer until we either reach the end of the list or find the student to delete

```
while((current != null) && (!name.equals(current.getName()))) {  
    current = current.getNext();  
}
```

c) and at the end of the loop...

c1) if we are at the end of the list,
nothing remains to be done

```
if (current == null) {  
    return;  
}
```

The student was not on the list

c1) if we have found the student to delete, delete it.

```
} else {  
    current.getPrev().setNext(current.getNext());  
    current.getNext().setPrev(current.getPrev());  
    return;  
}
```

c1) if we have found the student to delete, delete it.

```
} else {  
    current.getPrev().setNext(current.getNext());  
    current.getNext().setPrev(current.getPrev());  
    return;  
}
```

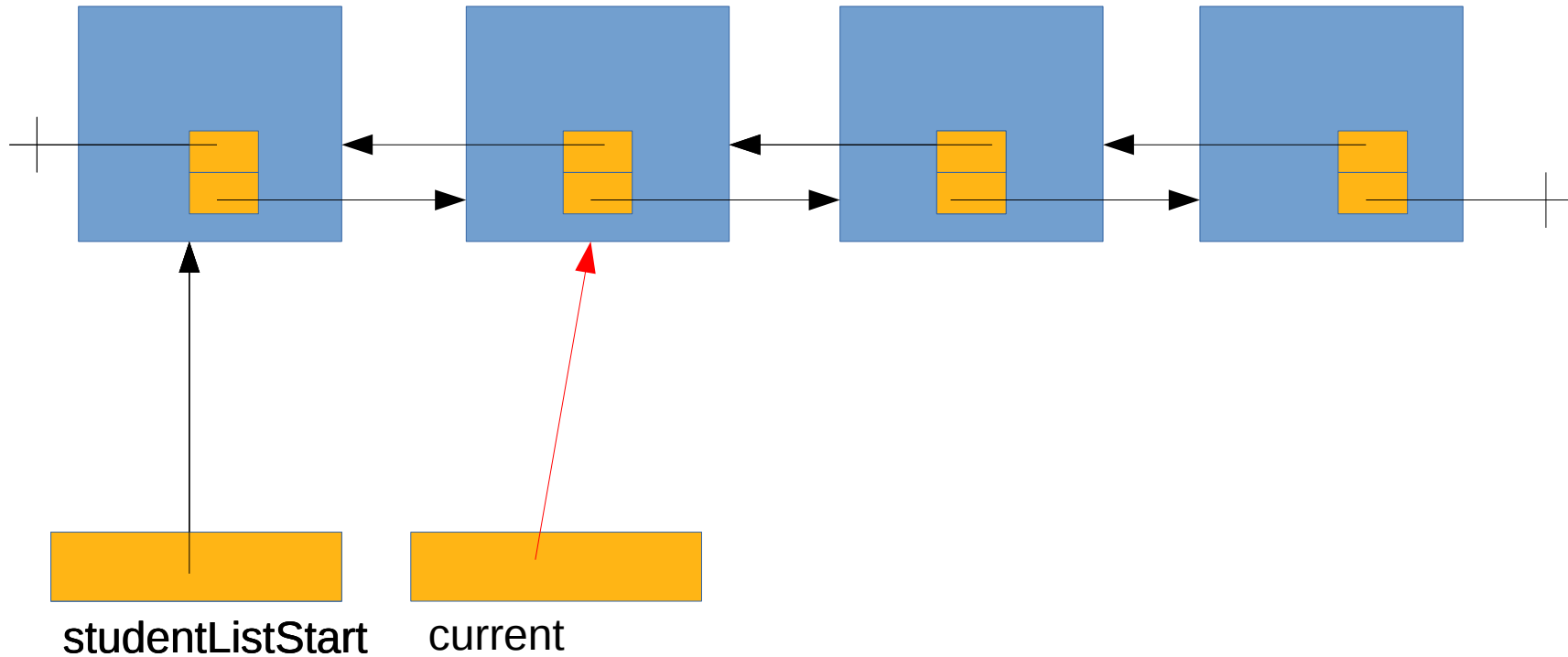
Those two lines are a bit more complex, so let's look at them in detail.

c1) if we have found the student to delete, delete it.

```
} else {  
    current.getPrev().setNext(current.getNext());  
    current.getNext().setPrev(current.getPrev());  
    return;  
}
```

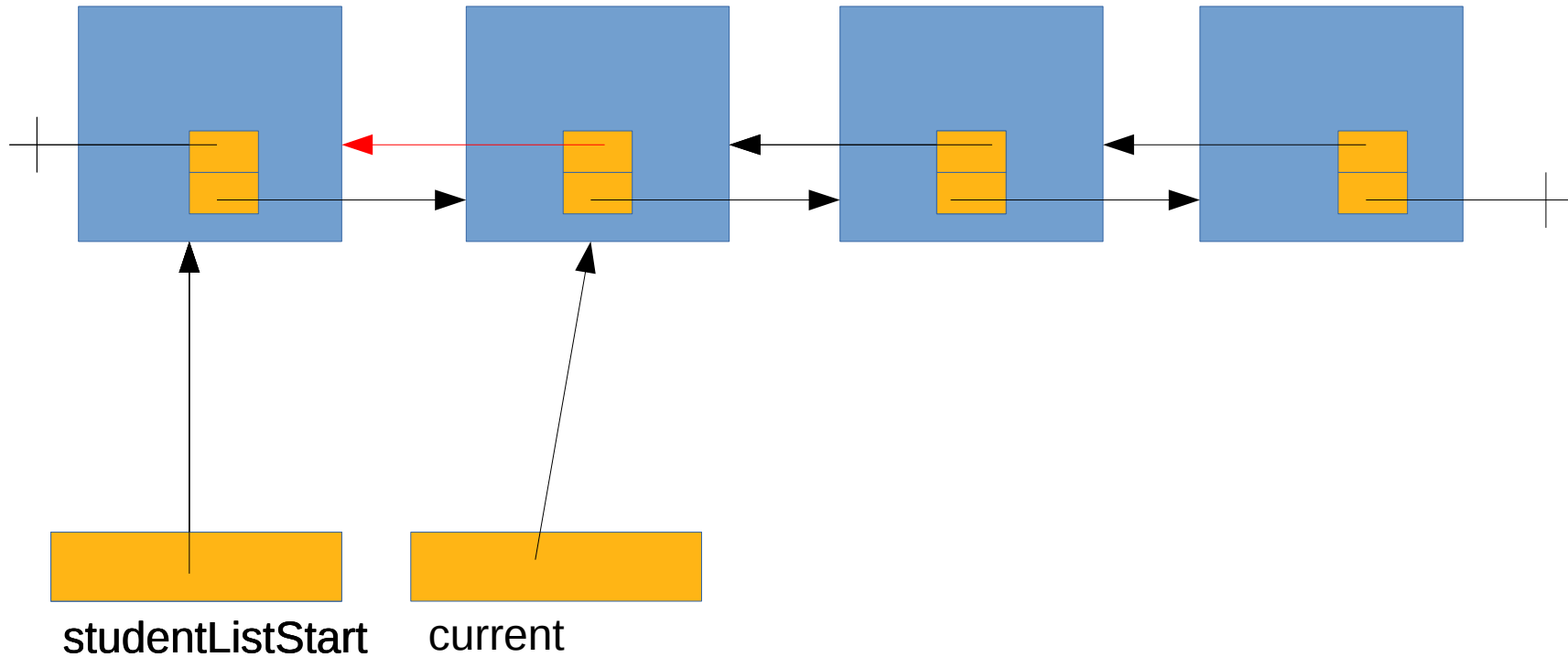
The first line sets the pointer going forward from the previous student to the next

(slow motion)



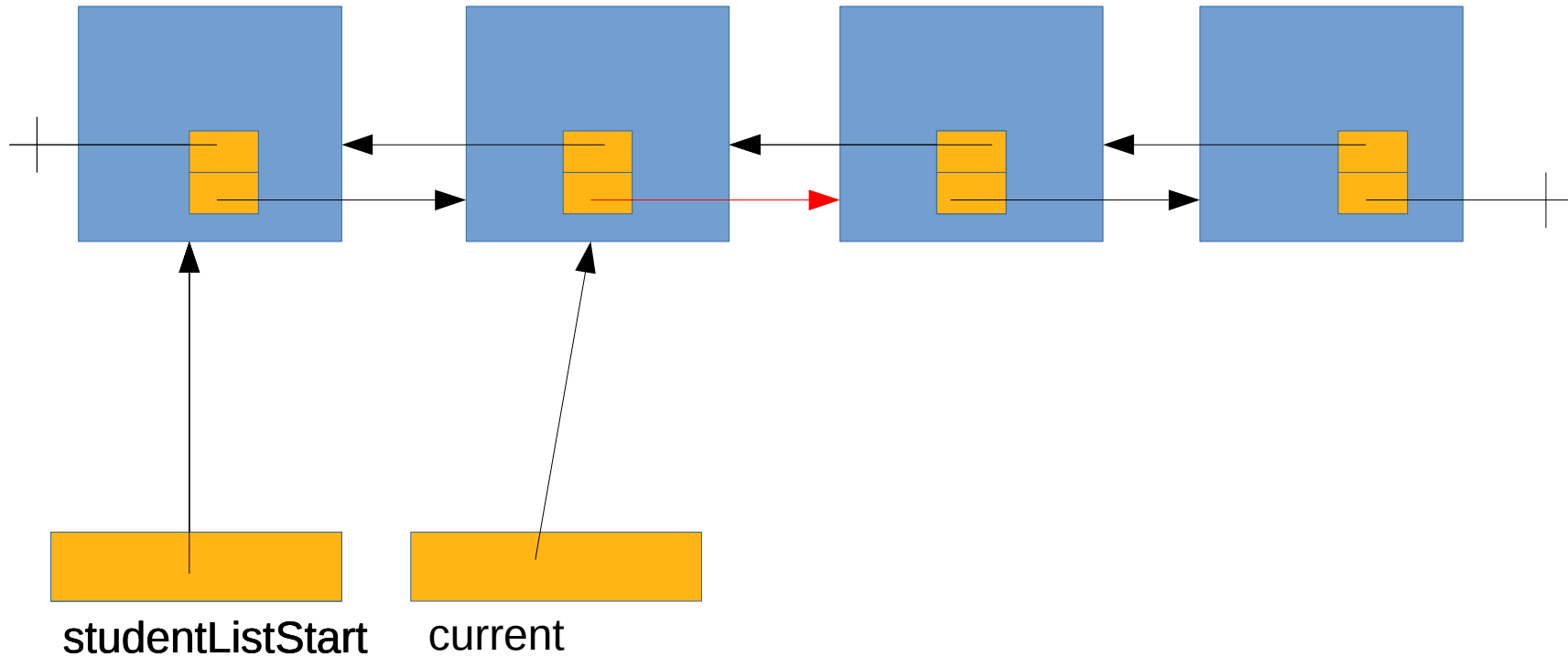
```
current.getPrev().setNext(current.getNext());
```

(slow motion)



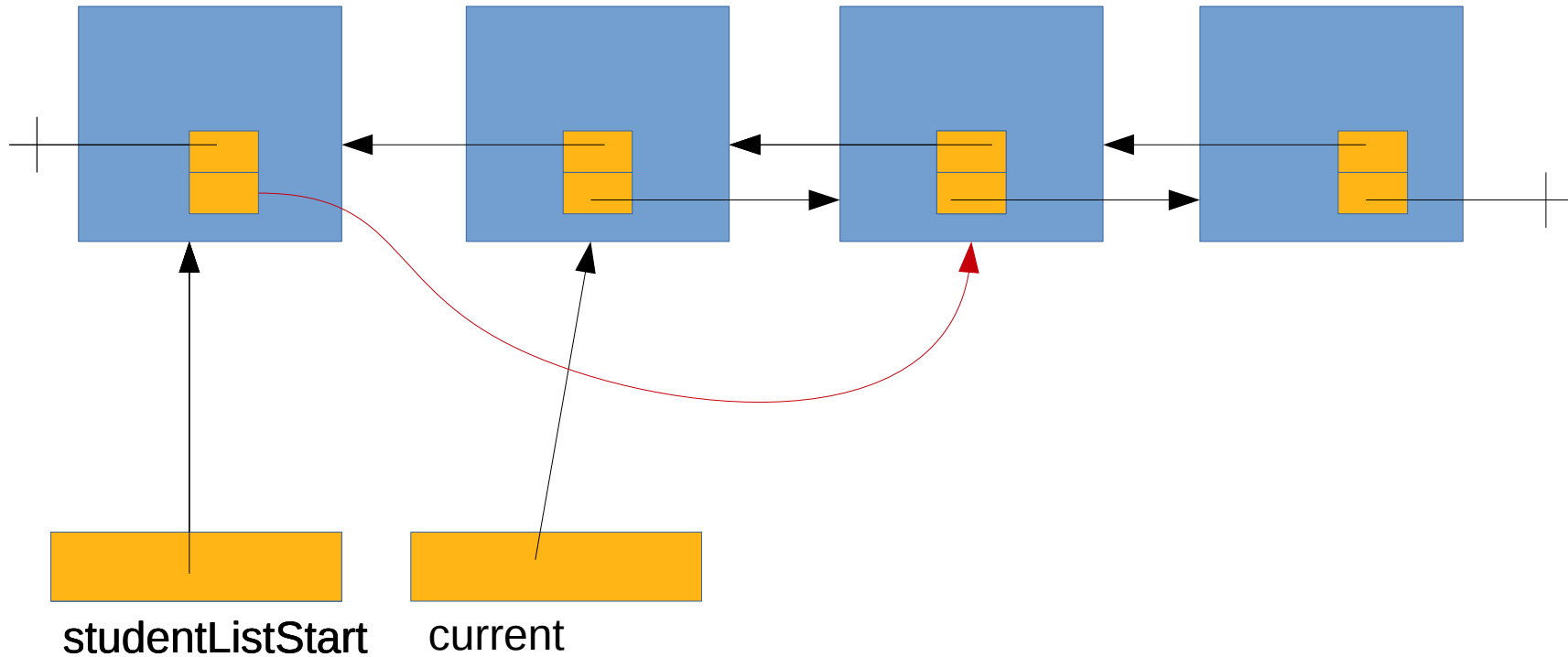
```
current.getPrev().setNext(current.getNext());
```

(slow motion)



```
current.getPrev().setNext(current.getNext());
```

(slow motion)



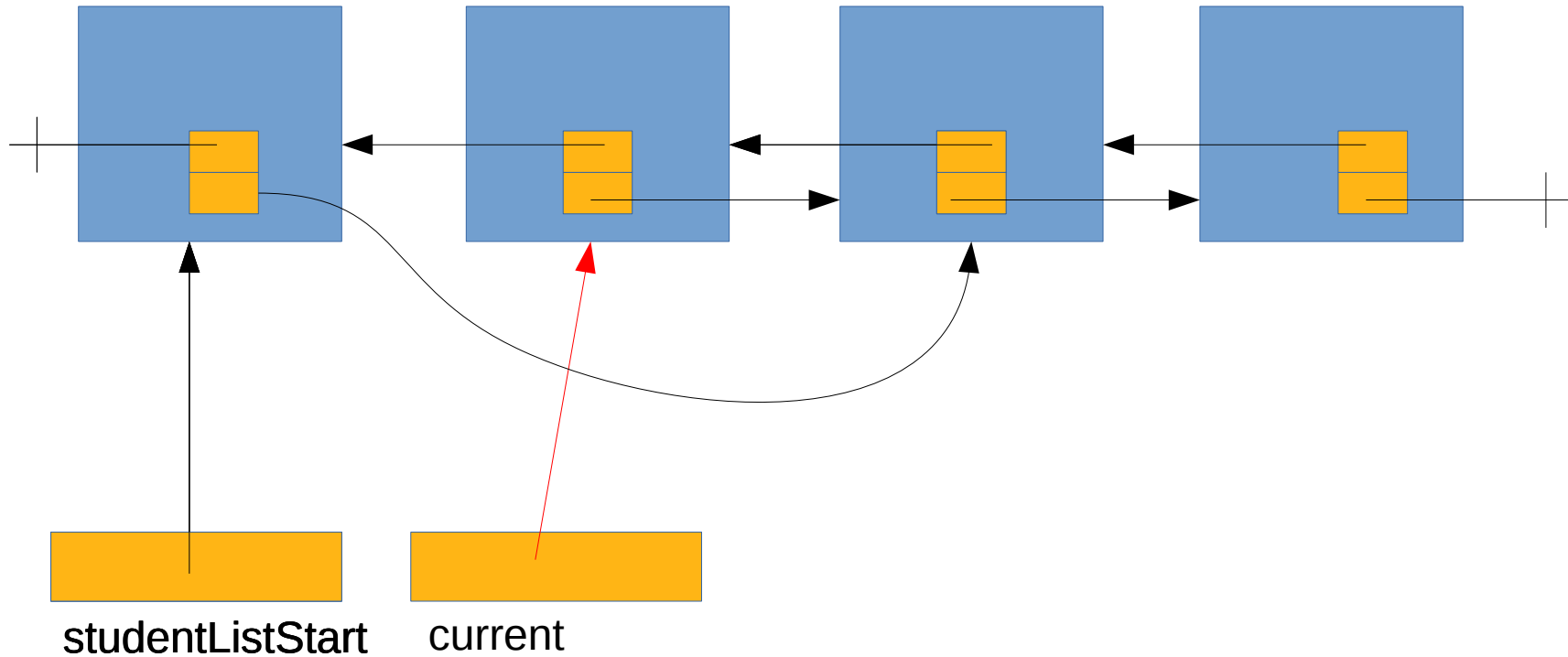
```
current.getPrev().setNext(current.getNext());
```

c1) if we have found the student to delete, delete it.

```
} else {  
    current.getPrev().setNext(current.getNext());  
    current.getNext().setPrev(current.getPrev());  
    return;  
}
```

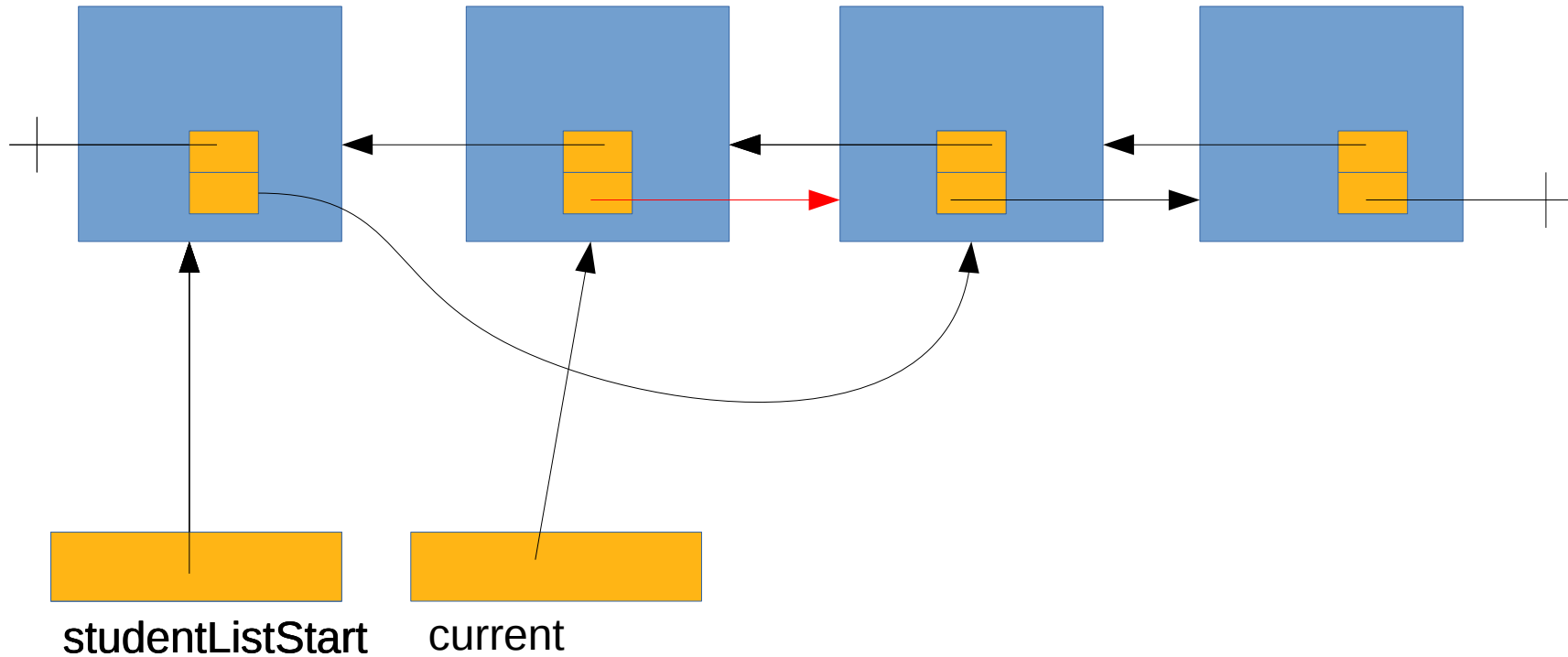
The second line sets the pointer going backwards from the next student to the previous one

(slow motion)



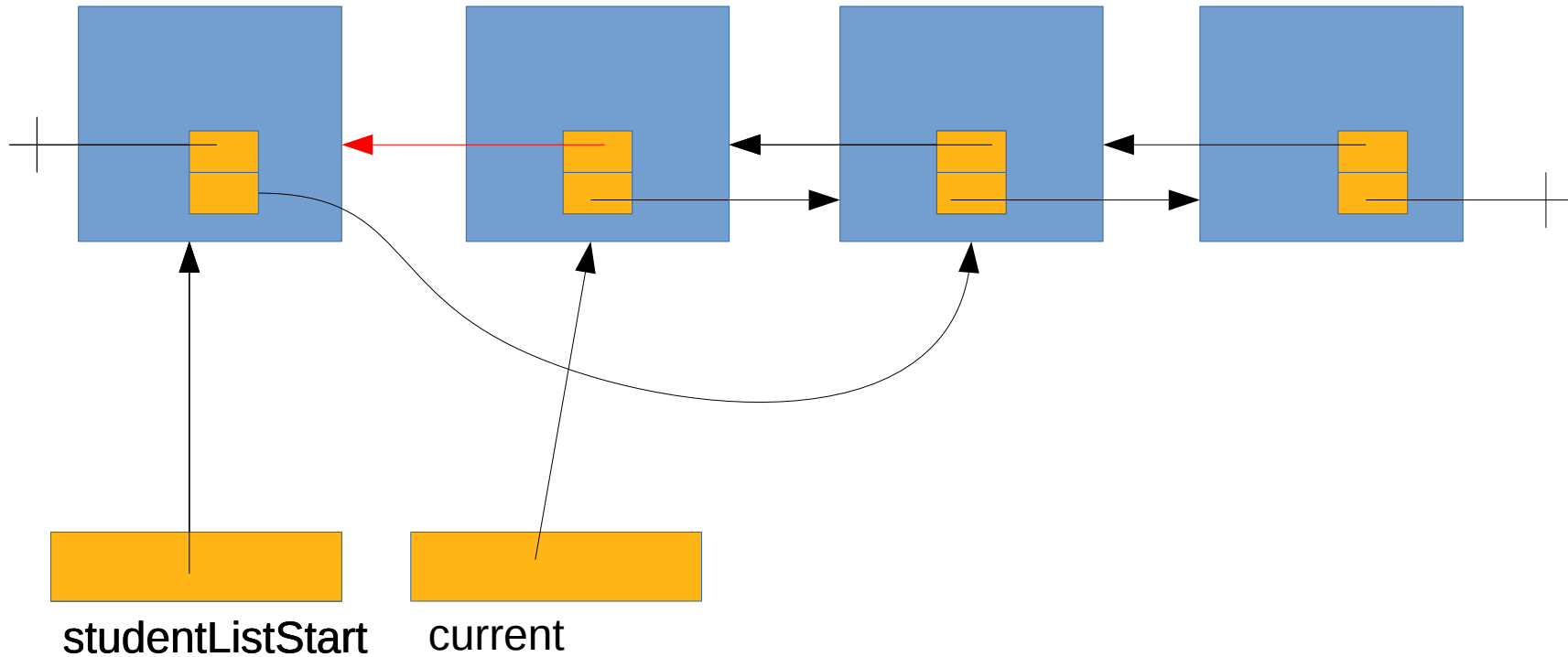
```
current.getNext().setPrev(current.getPrev());
```

(slow motion)



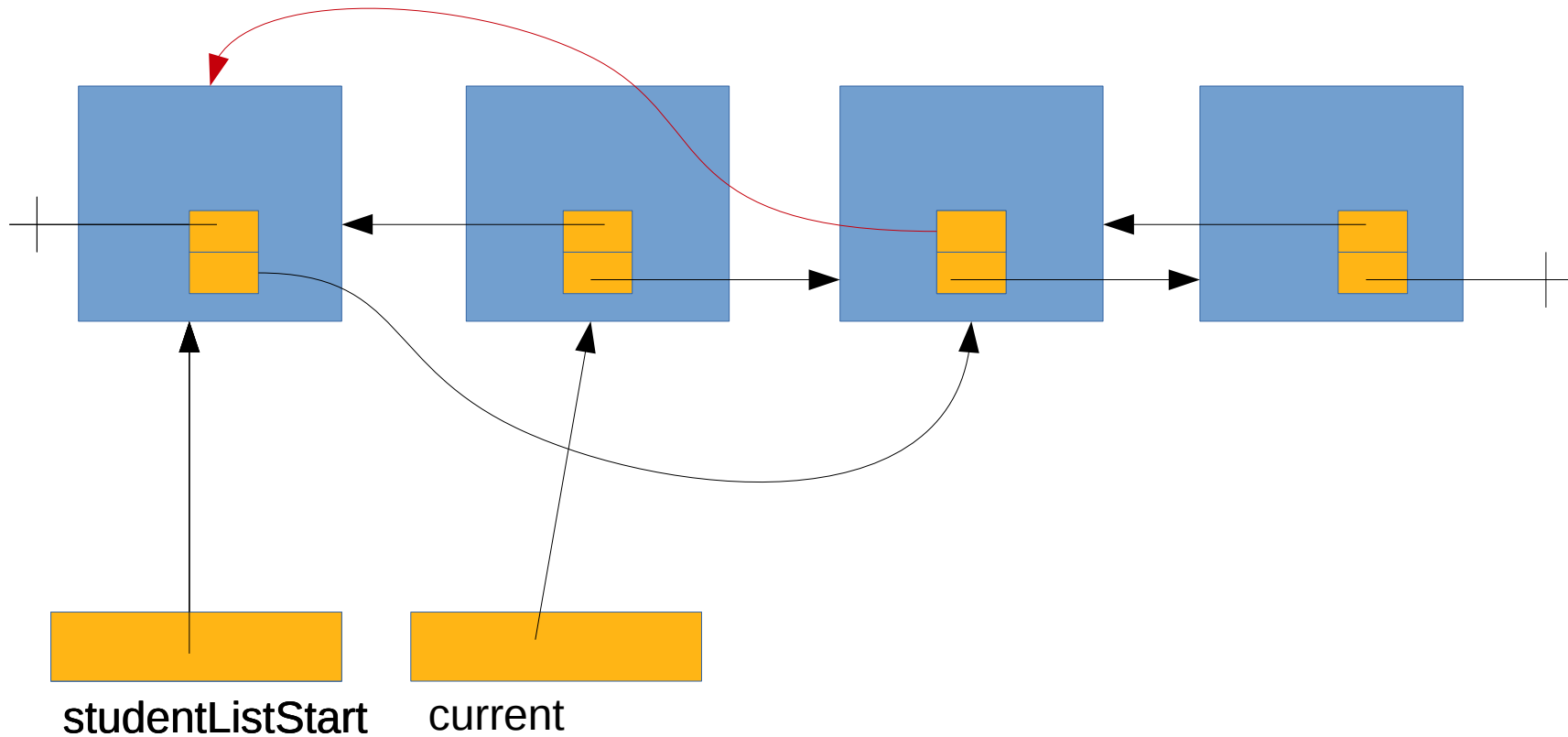
```
current.getNext().setPrev(current.getPrev());
```


(slow motion)



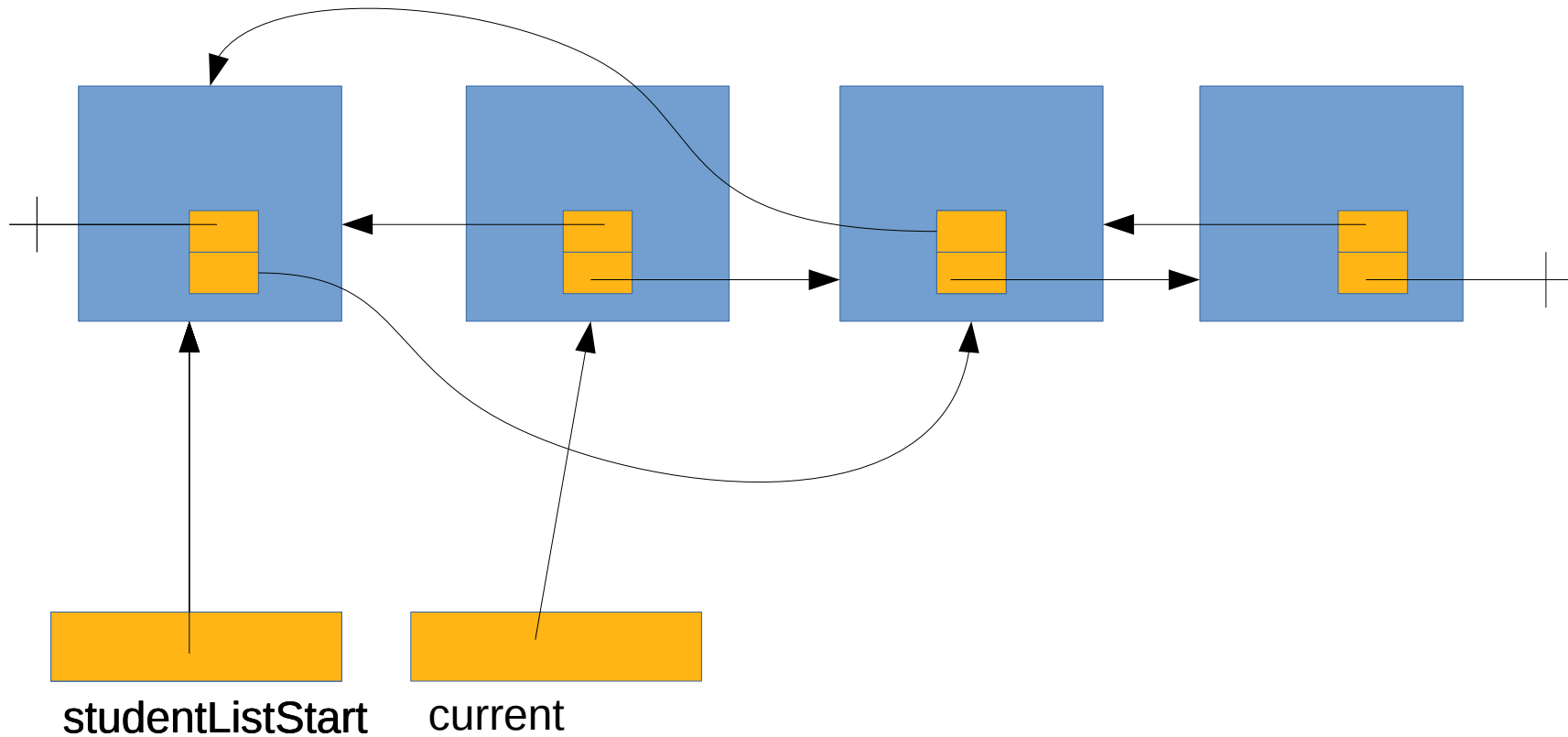
```
current.getNext().setPrev(current.getPrev());
```

(slow motion)



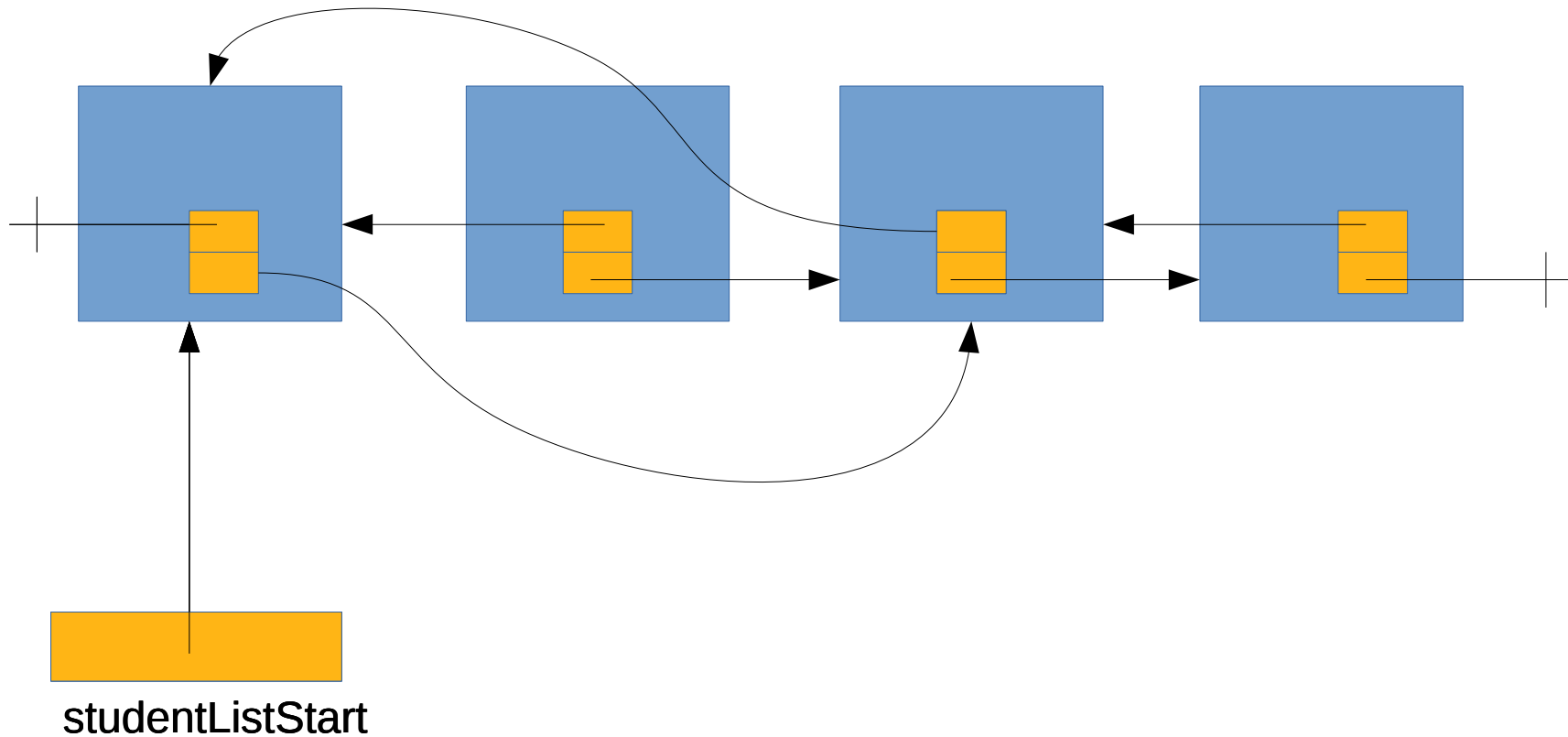
```
current.getNext().setPrev(current.getPrev());
```

(slow motion)



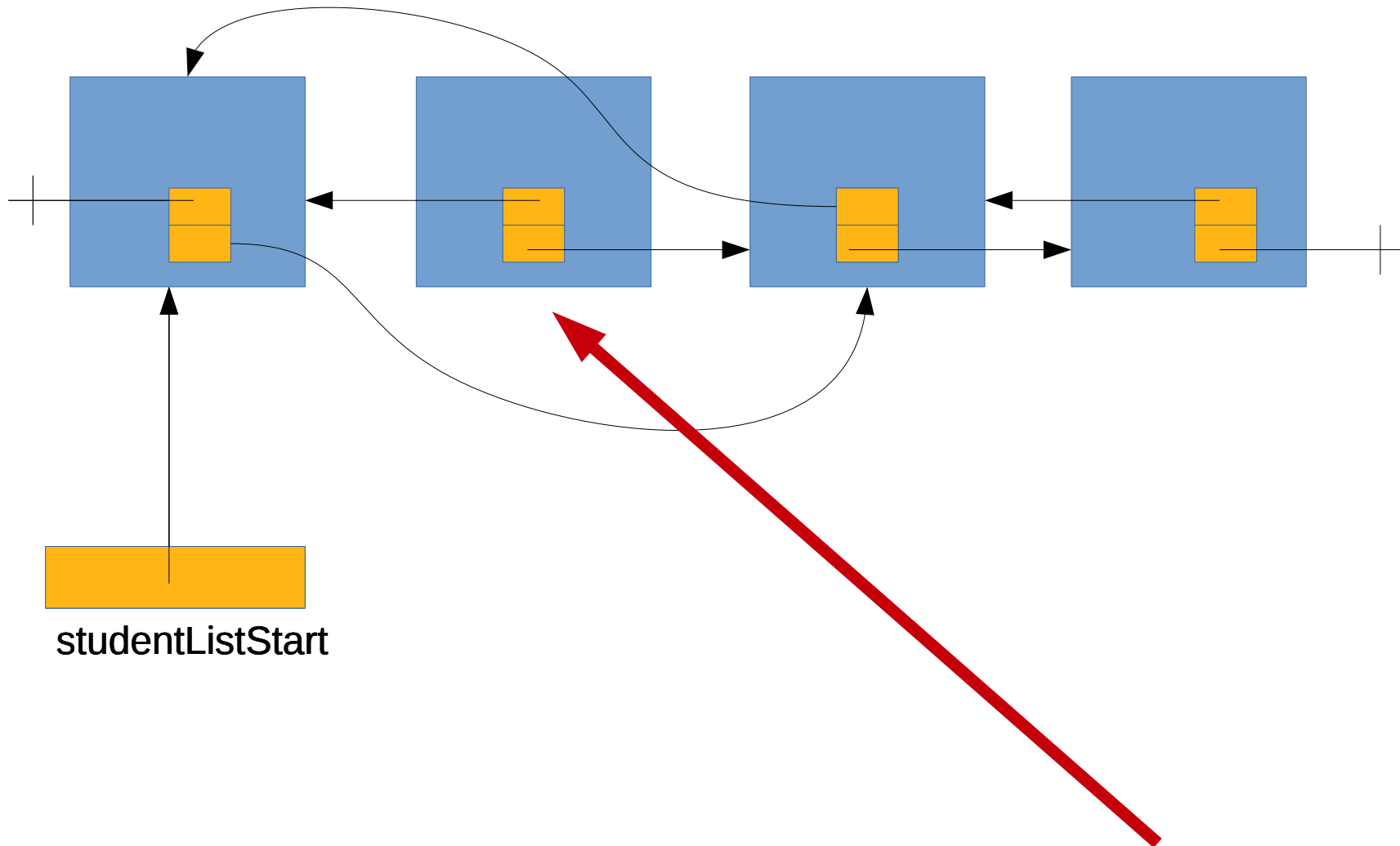
This is the result just before returning from the method

(slow motion)



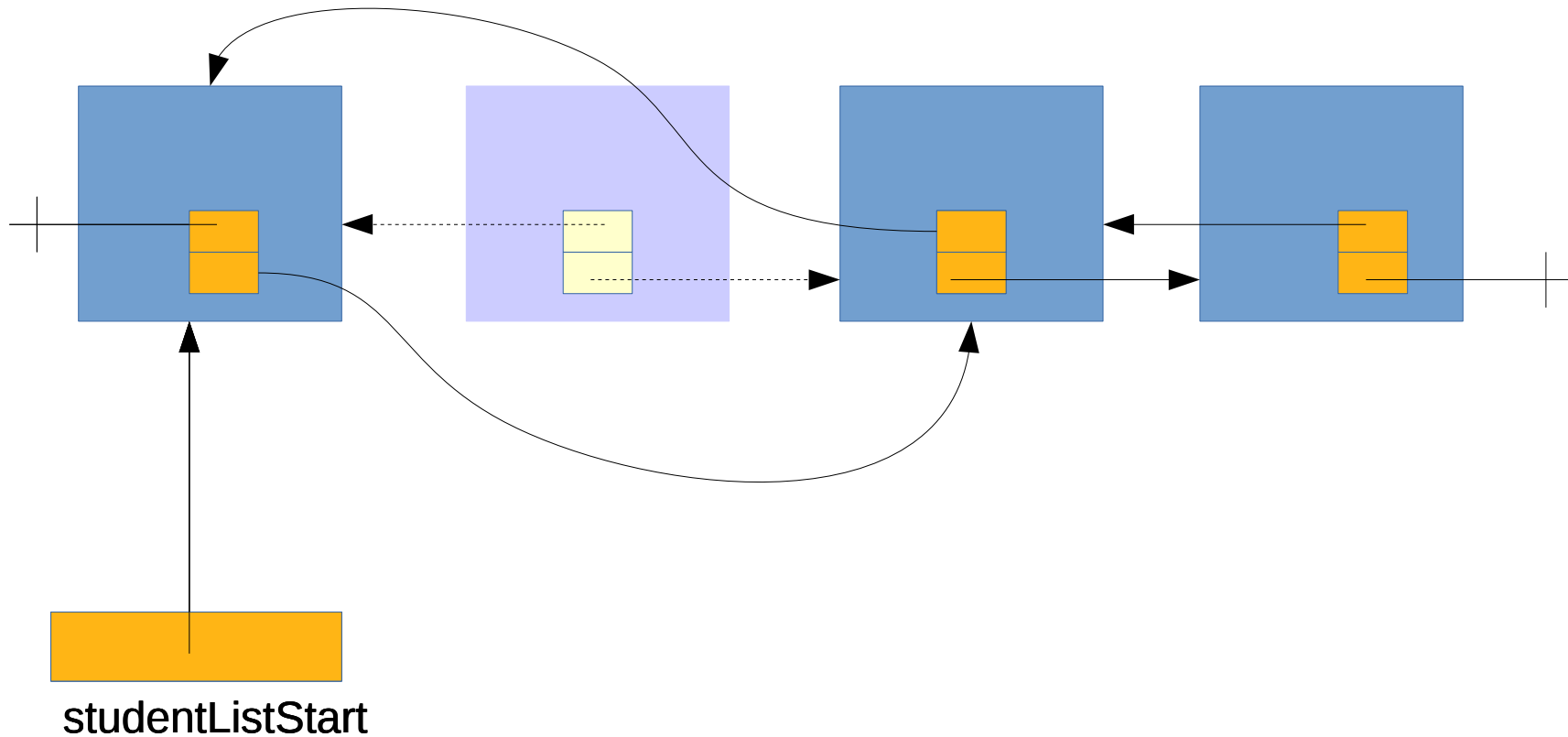
This is the result just after returning from the method:
local variables are lost.

(slow motion)



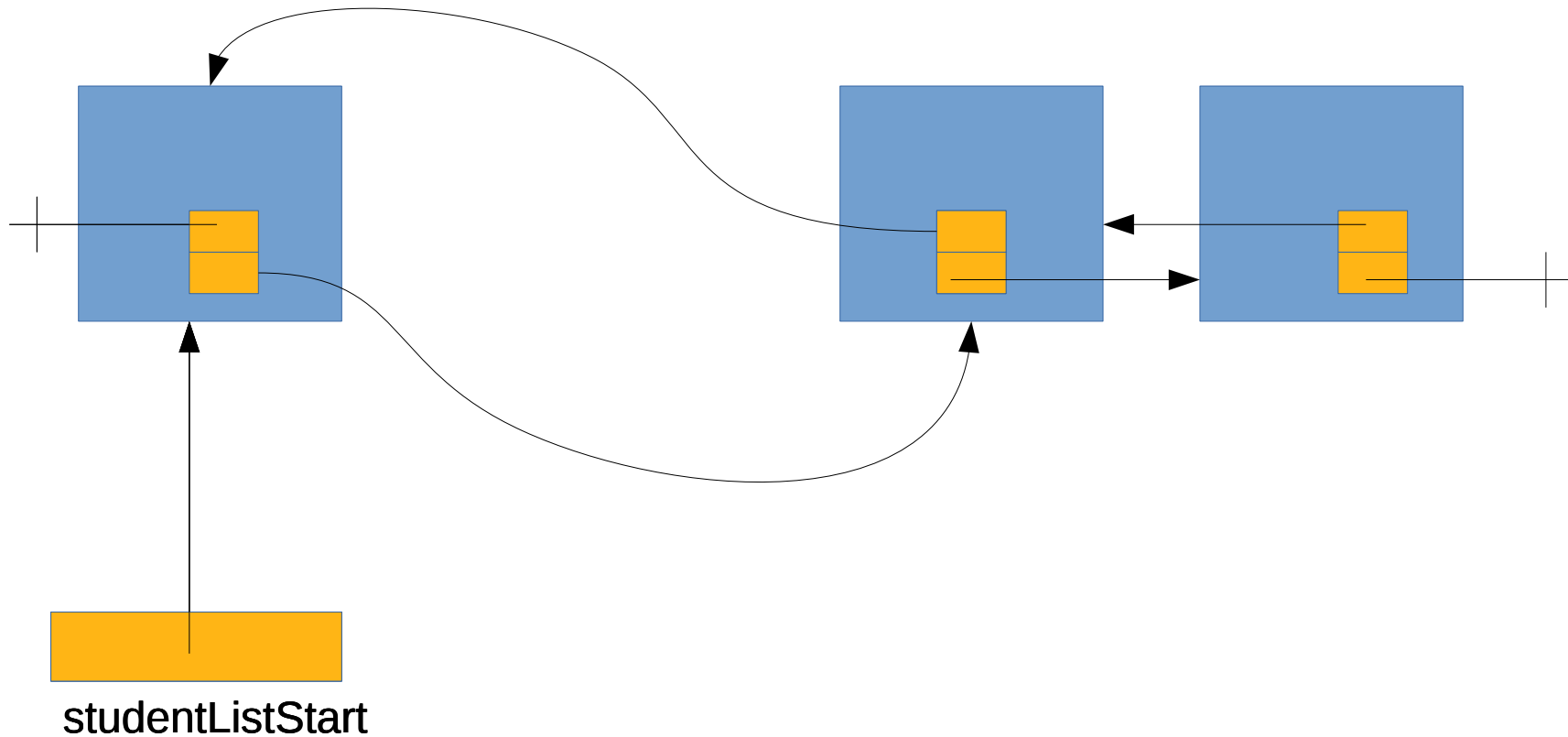
Note that there is no way we can access this element:
it is *inaccessible*!

(slow motion)



At some point, the Garbage Collector will reclaim that memory. The programmer does not need to care about it.

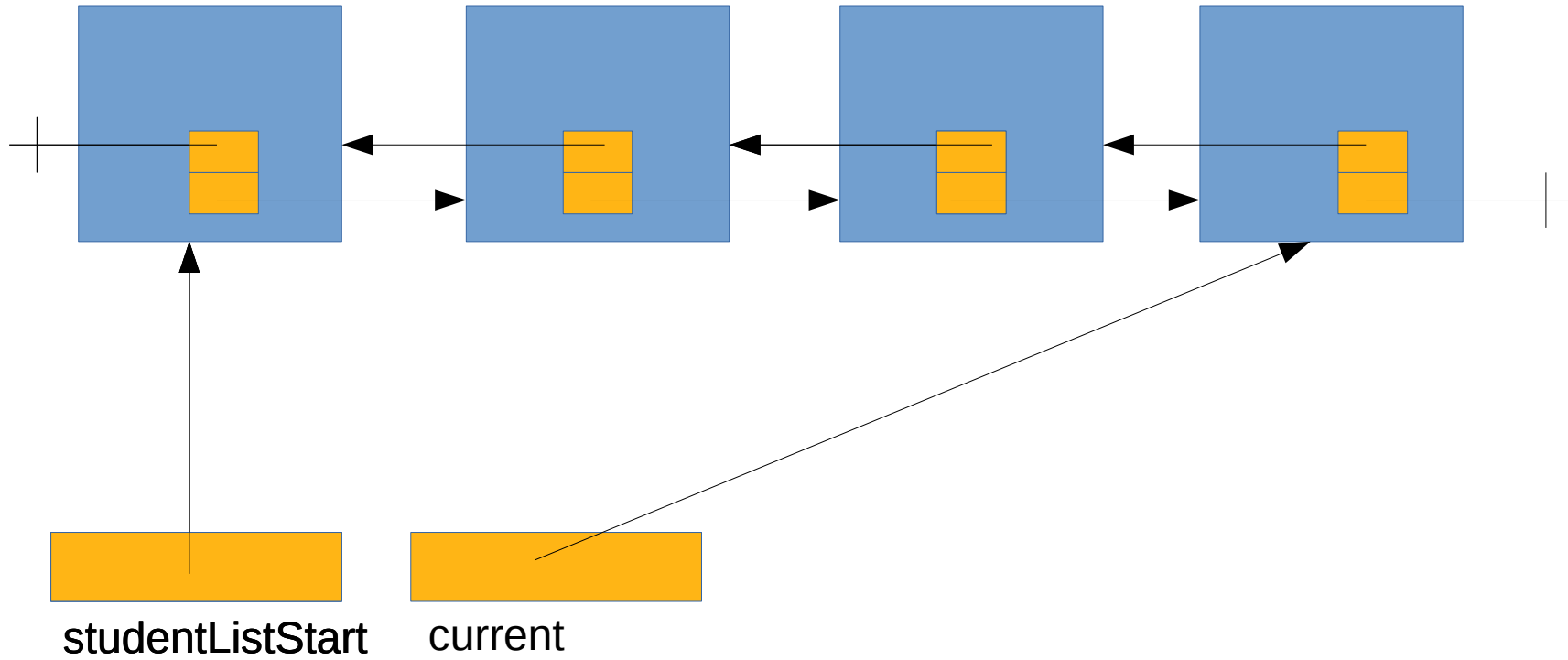
(slow motion)

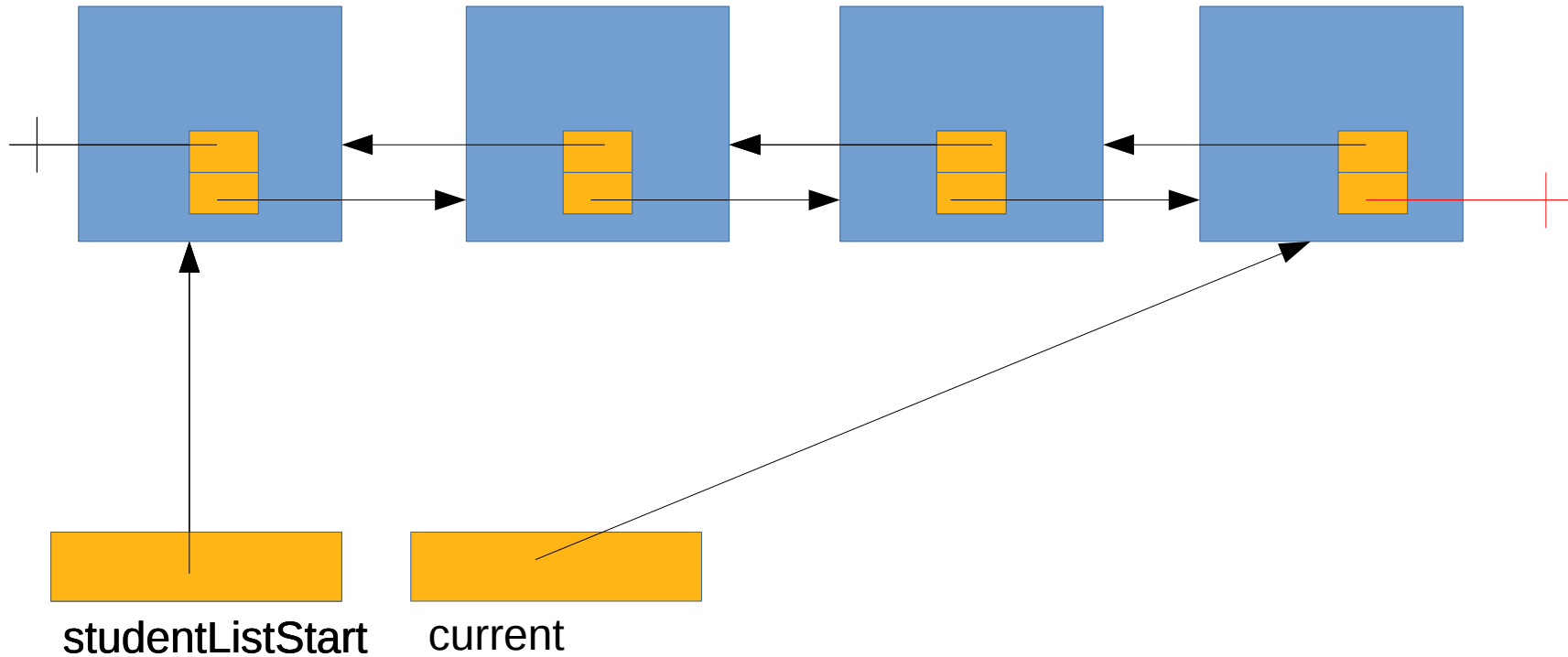


At some point, the Garbage Collector will reclaim that memory. The programmer does not need to care about it.

But wait!
We have forgotten something!

What if 'current' is the last student?





In this case, `current.getNext()` is null,
and `current.getNext().setPrev(...)` will throw
a `NullPointerException`!

Fortunately enough, this is easy to fix:

```
if (current.getNext() != null) {  
    current.getNext().setPrev(current.getPrev());  
}
```

...and that's all, folks!

Full code for this method

```
public void deleteStudent(String name) {
    if ((name == null) || "".equals(name)) {
        return;
    }
    if (name.equals(studentListStart.getName())) {
        studentListStart = studentListStart.getNext();
        studentListStart.setPrev(null);
    } else {
        Student current = studentListStart;
        while((current != null) && (!name.equals(current.getName()))) {
            current = current.getNext();
        }
        if (current == null) {
            return;
        } else {
            current.getPrev().setNext(current.getNext());
            if (current.getNext() != null) {
                current.getNext().setPrev(current.getPrev());
            }
            return;
        }
    }
}
```

Phase I : iterative approach

Phase I : iterative approach
Step III : printing the list
and calculating length

These two cases are much simpler,
so the code should be easier to understand.

Full method for calculating the number of students

```
public int getStudentCount() {  
    int result = 0;  
    Student current = studentListStart;  
    while(current != null) {  
        result++;  
        current = current.getNext();  
    }  
    return result;  
}
```

Full method for printing the students

```
public void printStudentList() {  
    Student current = studentListStart;  
    while(current != null) {  
        System.out.print("Student name: " + current.getName() + ". ");  
        System.out.println("Year: " + current.getYear() + ". ");  
        current.setNext(current.getNext());  
    }  
}
```

That's all, folks!

Full code of Student.java (1/2)

```
public class Student {  
  
    private String name;  
  
    private int year;  
  
    private Student next;  
  
    private Student prev;  
  
    public Student(String name, int year) {  
        this.name = name;  
        this.year = year;  
        this.next = null;  
        this.prev = null;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getYear() {  
        return year;  
    }  
  
    public Student getNext() {  
        return next;  
    }  
  
    public void setNext(Student nextStudent) {  
        this.next = nextStudent;  
    }  
}
```

Full code of Student.java (2/2)

```
public int getYear() {  
    return year;  
}  
  
public Student getNext() {  
    return next;  
}  
  
public void setNext(Student nextStudent) {  
    this.next = nextStudent;  
}  
  
public Student getPrev() {  
    return prev;  
}  
  
public void setPrev(Student prevStudent) {  
    this.prev = prevStudent;  
}  
}
```

Full code of School.java (1/3)

```
public class School {  
  
    private Student studentListStart = null;  
  
    public School() {  
        this.studentListStart = null;  
    }  
  
    public void enrolStudent(String name, int year) {  
        Student newStudent = new Student(name, year);  
        if (this.studentListStart == null) {  
            this.studentListStart = newStudent;  
            return;  
        } else {  
            Student current = studentListStart;  
            while(current.getNext() != null) {  
                current = current.getNext();  
            }  
            current.setNext(newStudent);  
            newStudent.setPrev(current);  
            return;  
        }  
    }  
  
    public void deleteStudent(String name) {  
        if ((name == null) || "".equals(name)) {  
            return;  
        }  
        if (name.equals(studentListStart.getName())) {
```

Full code of School.java (2/3)

```
public void deleteStudent(String name) {
    if ((name == null) || "".equals(name)) {
        return;
    }
    if (name.equals(studentListStart.getName())) {
        studentListStart = studentListStart.getNext();
        studentListStart.setPrev(null);
    } else {
        Student current = studentListStart;
        while((current != null) && (!name.equals(current.getName()))) {
            current = current.getNext();
        }
        if (current == null) {
            return;
        } else {
            current.getPrev().setNext(current.getNext());
            if (current.getNext() != null) {
                current.getNext().setPrev(current.getPrev());
            }
            return;
        }
    }
}

public int getStudentCount() {
    int result = 0;
    Student current = studentListStart;
    while(current != null) {
        result++;
        current = current.getNext();
    }
    return result;
}

public void printStudentList() {
```


Full code of School.java (3/3)

```
public int getStudentCount() {
    int result = 0;
    Student current = studentListStart;
    while(current != null) {
        result++;
        current = current.getNext();
    }
    return result;
}

public void printStudentList() {
    Student current = studentListStart;
    while(current != null) {
        System.out.print("Student name: " + current.getName() + ". ");
        System.out.println("Year: " + current.getYear() + ". ");
        current = current.getNext();
    }
}
}
```

Phase II : recursive approach

Phase II : recursive approach
Step I : addition to the list

(...under construction...)

Phase II : recursive approach

Phase II : recursive approach
Step II : deletion from the list

(...under construction...)

Phase II : recursive approach

Phase II : recursive approach
Step III : printing the list
and calculating length

(...under construction...)

Exercises for the reader

- Modify the code of School and/or Student so that:
 - Students are always in alphabetical order
 - There is more than one list, and each list contains students of one year group
 - The code of `enrolStudent(Student)` does not use an `if` (it is not really needed)
 - Do it both iteratively and recursively
 - Do it both for doubly- and singly-linked lists