LINKED NODE IMPLEMENTATIONS

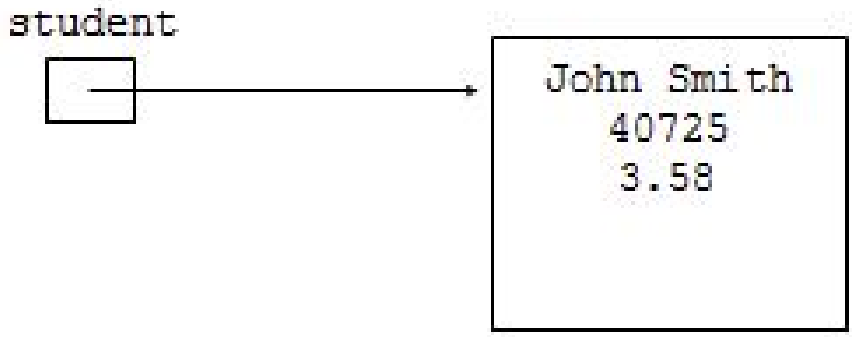
REVIEW OF OBJECT REFERENCES AND ARRAYS

Object References

- Object variables (also called *reference variables* or *object references*) are stored different from primitive variables
- Object references store the *memory address* of where the object is located
- You can also think about this as a *pointer* or *reference*

Object References

```
public class Student {
   private String name;
   private int studentID;
   private double gpa;
   ...
}
Student student = new Student("John Smith", 40725, 3.58);
```



Object References

- student stores a memory location or pointer to somewhere else in memory where all the data that describes a Student object is saved

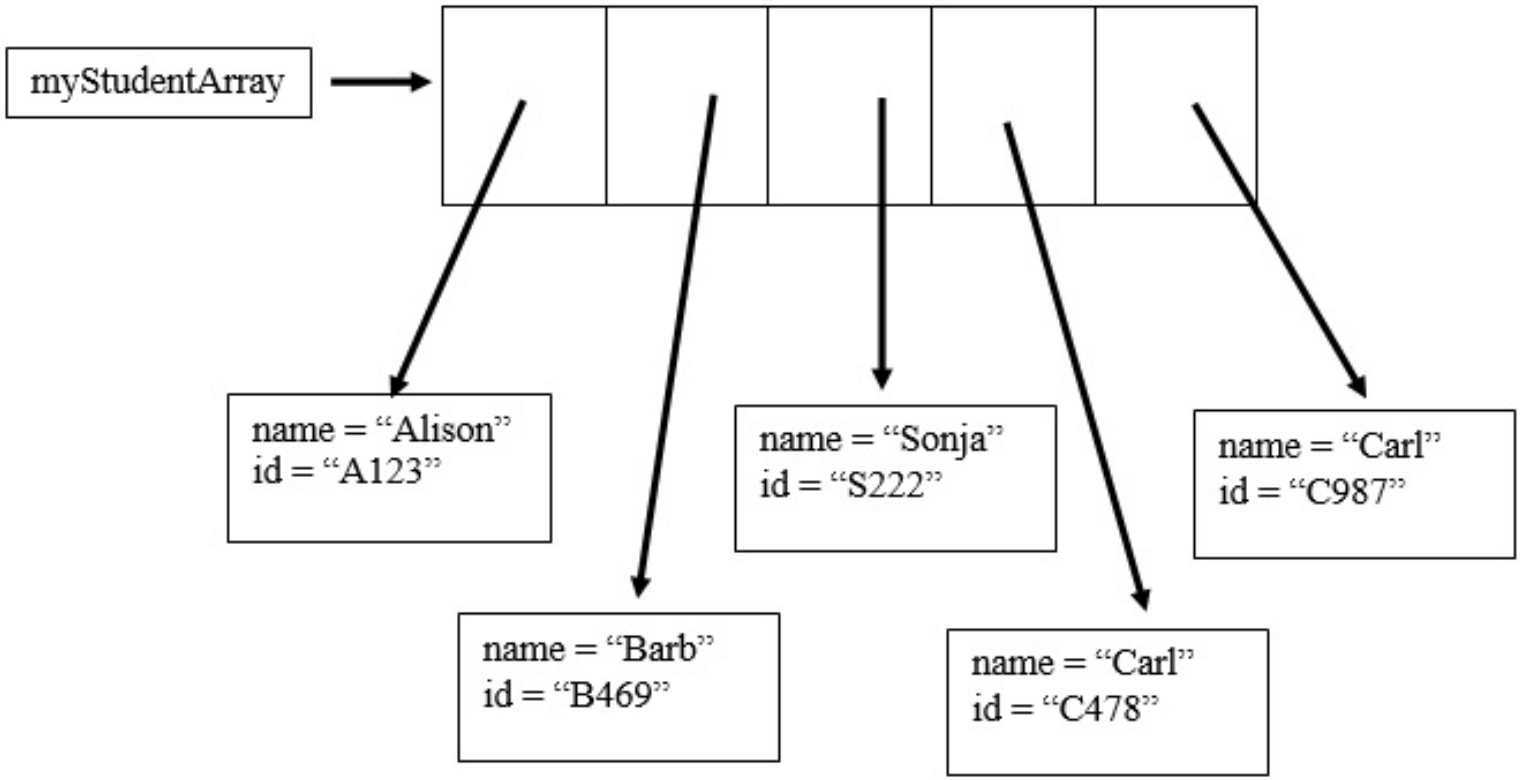
Arrays

- Arrays are *static* data structures
  - They have a fixed size
- When you declare an array, you state the size.
- This size cannot be changed.
- This can be a drawback/challenge of using an array.

Arrays of Objects in Memory

- An array is an object!
- An array variable holds a memory address to where the array is located
  - Each position in the array holds a memory address to where that object is located
- In other words, for an array of objects, what is stored is an array of memory locations.

Arrays of Objects in Memory



Static vs. Dynamic Data Structures

- Arrays are a static data structure.
- A *dynamic* data structure does not have a fixed size.
  - It can grow and shrink as required by the contents.

Dynamic Data Structures

- You can create a dynamic data structure by linking objects together.
- We can take advantage of how java stores objects to create *linked nodes*.

LINKING OBJECTS TOGETHER

Linking Objects Together

- Pretend each Student keeps track of their own information and also can link or point to another student.
- This would allow us to:
  - Create new memory when we create a new Student
  - Continually expand the collection as needed
  - Link together all of our student objects
    - studentA points to studentB, studentB points to studentC, etc.

Linking Objects Together

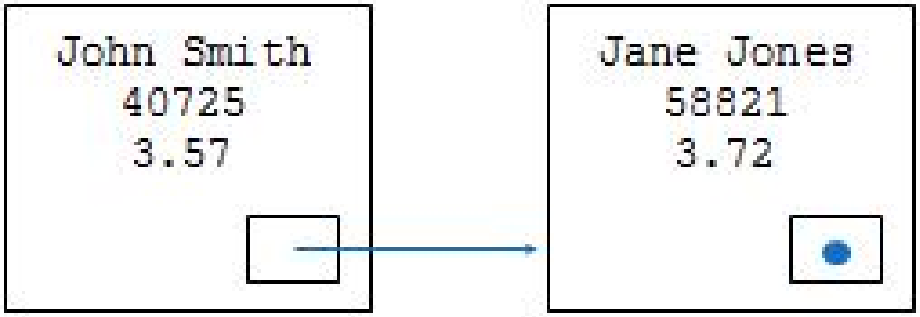
```
public class Student {
    private String name;
    private int ID;
    private double gpa;
    private Student nextStudent;
    ...
}
```

- Part of what describes a Student (part of its instance data variables) is another Student!
- The class contains instance data pointing to another Student object.

Linking Objects Together

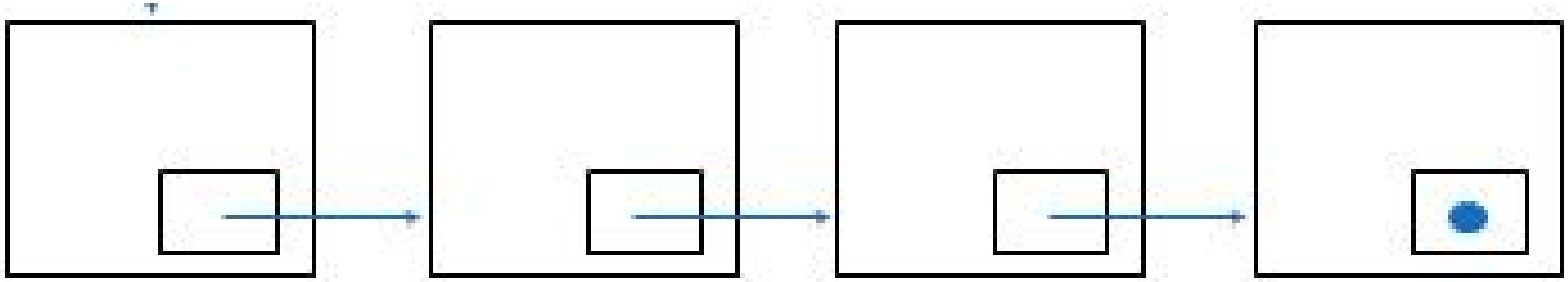
```
Student student1 =
   new Student("Jane Jones", 58821, 3.72, null);
Student student2 =
   new Student("John Smith", 40725, 3.58, student1);
```

Because student1 is an object, what the variable nextStudent holds is a *memory address* pointing to the location of student1 in memory.



Linking Objects Together

- We can create a *linked list* of Student objects.
- Each object points to the *next in line*.



Linking Objects Together

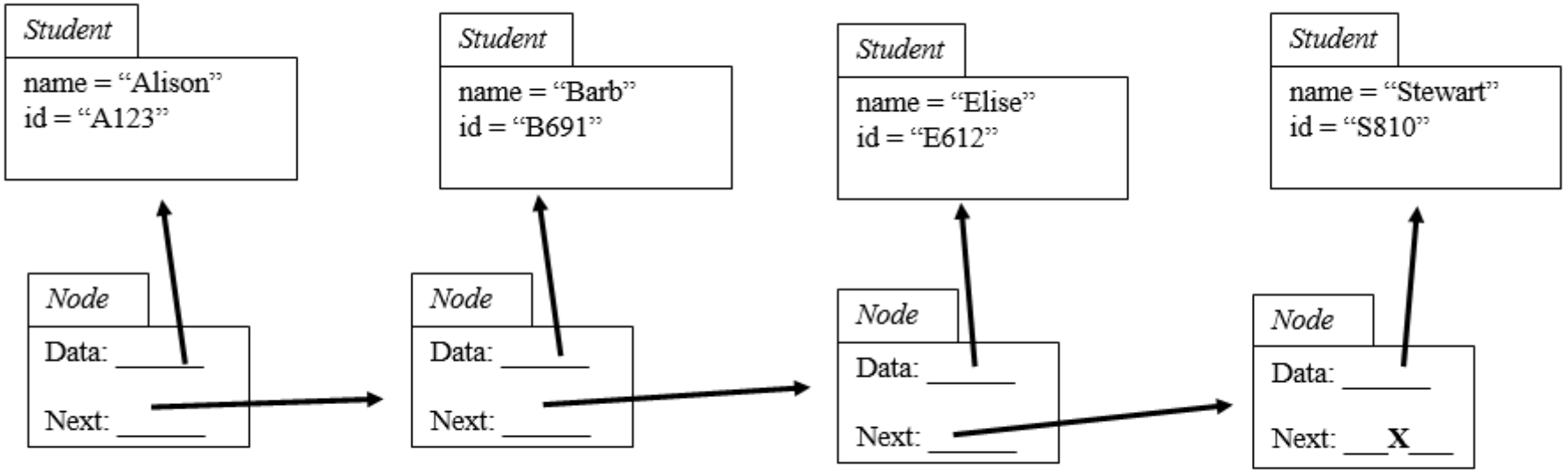
- This works to link Students together.
- But does it really make logical sense?
  - *Should* a Student object really keep track of the next student in the list?
  - A student's name, ID, and GPA *describe* the student. It makes sense to keep track of them *as part of* the Student object.
  - But does the next student in line make sense to keep track of as *part of* what describes an individual student?

NODES

Nodes

- A class used to link objects together
- Nodes contain two pieces of information
  - A reference to the current object
  - A reference (or link) to the next node in the list
- Nodes allow us to create a *linked list* of objects
  - Each node points to one object on the list and the next node in the chain
- The class that describes the object no longer needs to keep track of “next in the list”

Chain of Nodes



Node Class

```
public class Node<T> {

    private T data;
    private Node next;

    public Node(Tdata) {
        this(data, null);
    }

    public Node(T data,
                Node next) {
    this.data = data;
    this.next = next;
}
```

```
public T getData() {
    return data;
}
public void setData(T data) {
    this.data = data;
}
public Node getNext() {
    return next;
}
public void setNext(Node next){
    this.next = next;
}

}
```

Node Class

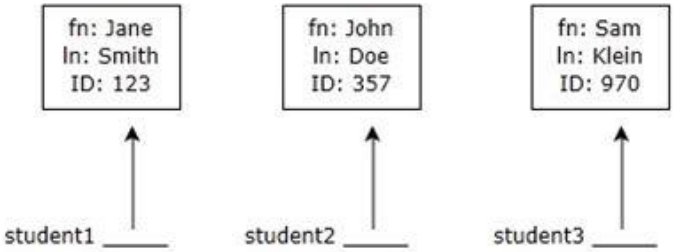
- Two constructors
  - One to set the next to null (last node in the chain)
  - One to set the next to some other node
- Common to be nested inside another class:

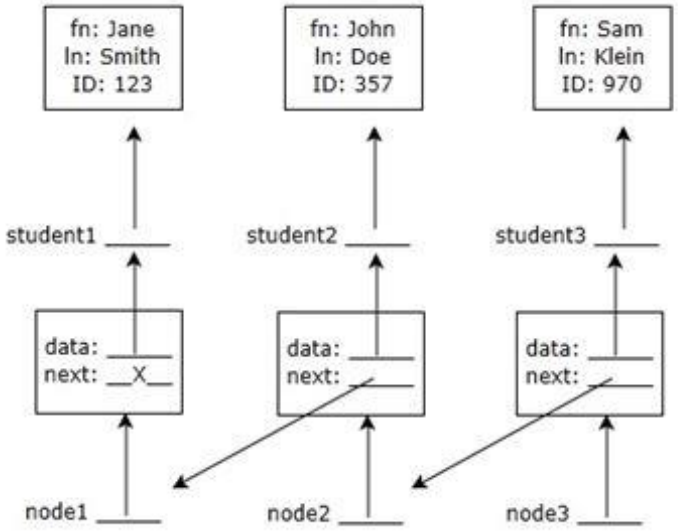
```
public class MyNodeCollectionClass<T> {
     private Node firstInCollection;

     public class Node {
          T data;
}
```

Example

```
Student student1 = new Student("Jane Smith", 123);
Student student2 = new Student("John Doe", 357);
Student student3 = new Student("Sam Klein", 970);
```

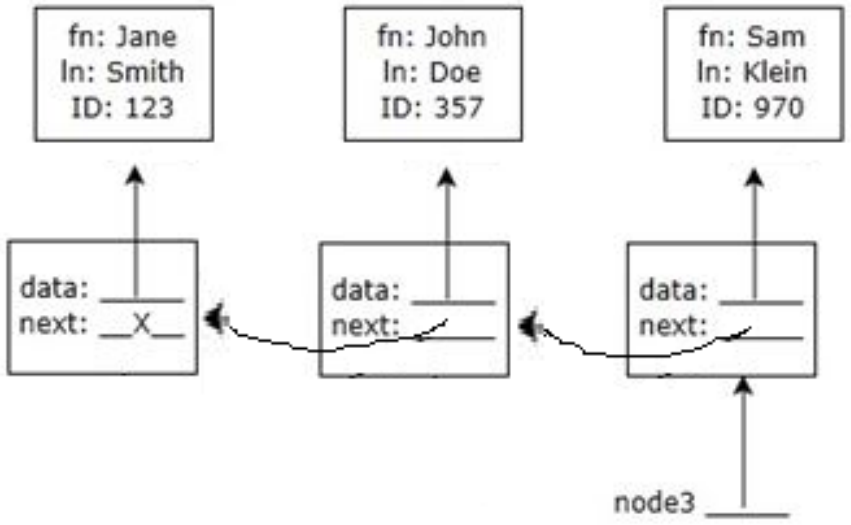





```
Node node1 = new Node(student1);
Node node2 = new Node(student2,node1);
Node node3 = new Node(student3,node2);
```

Example (cont.)

- In reality, we rarely have named Node objects or named objects in the list. Instead, the picture would look like this:



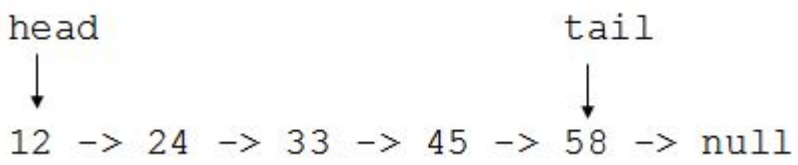
Linked Nodes

- We typically have just one or two named node references
  - Example: head, tail
  - Example: firstNode, lastNode
- We do not have access to *any* node in the chain
  - We can't directly access a middle element!
- We access the train by traversing (looping through) the links.

Example

Memory Address of the Node	Node's Data	Node's Next
1001	33	1007
1002	12	1005
1003		
1004	58	null
1005	24	1001
1006		
1007	45	1004
1008		

head 1002



USING CHAINS OF LINKED NODES

Special Cases

- With linked nodes, you have to always account for special cases of empty lists and singleton lists (one-element lists).
  - Empty list: firstNode == null
  - Singleton list: firstNode.next == null
- Avoid NullPointerExceptions!
- You always need to check for these cases to make sure your code won't crash in these situations!

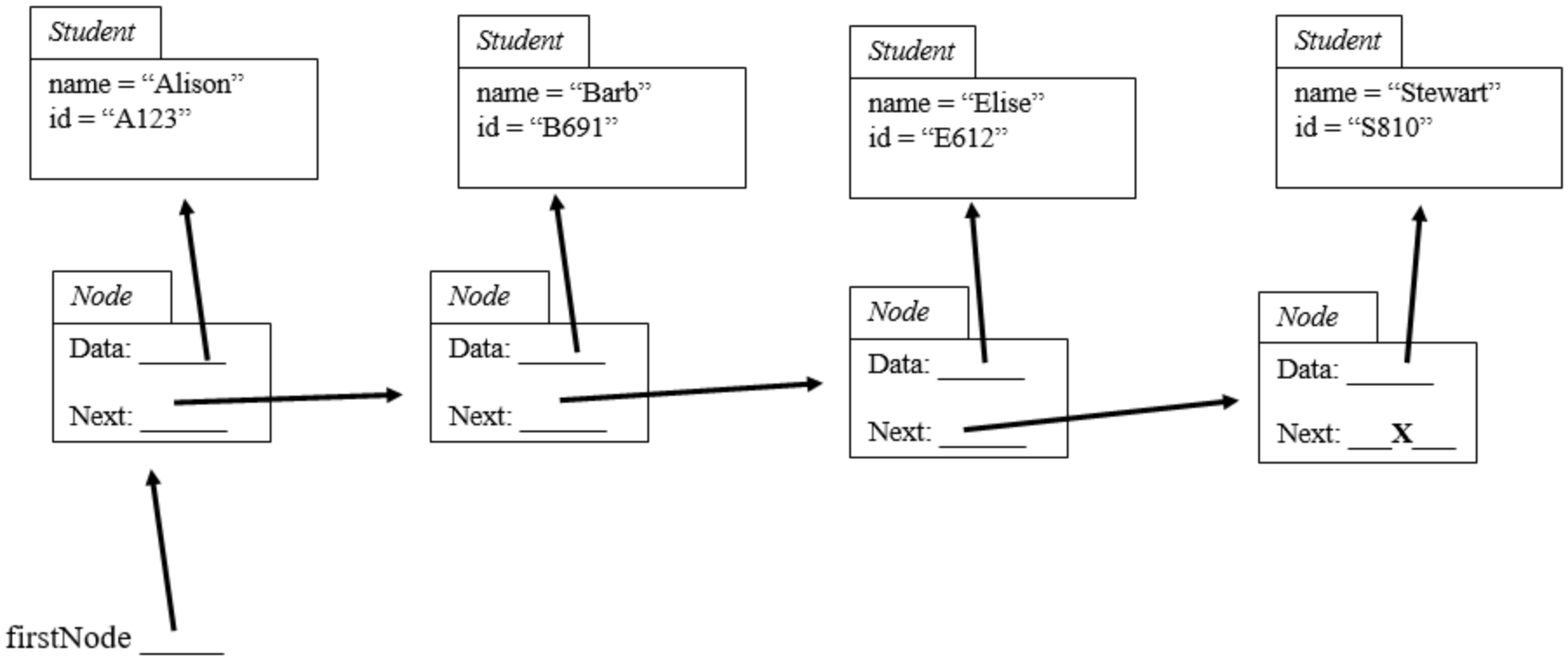
Iterating the Chain

- When we have a chain of nodes, we usually refer to the chain (or list) by just one node: the head of the chain.
  - Then within the chain, each node points to a single piece of data and the next node in the chain.
- However, we will frequently want to access all the objects in our chain.
- To do this, we use a temporary node to iterate the chain.

Iterating the Chain

- The temporary node initially points to the first node in the chain.
  - We access the data.
- Then we repeatedly point the temporary node to the next node and access that data.
- We stop when we reach the end of the list.

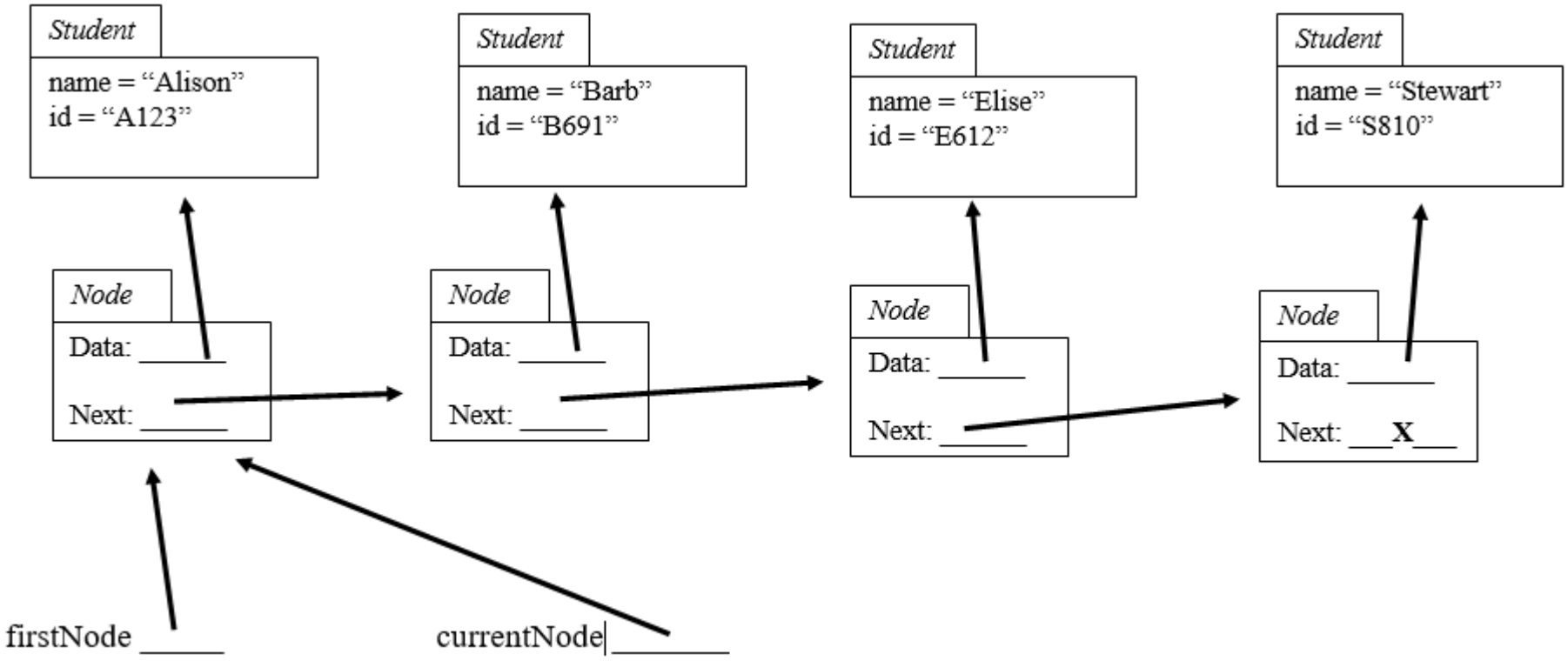
Iterating the Chain



Iterating the Chain

```
Node currentNode = firstNode;
```

- Note: this creates an **alias**- the value on the right side (a memory address) is placed into the variable on the left side. currentNode and firstNode now point to the same Node object.



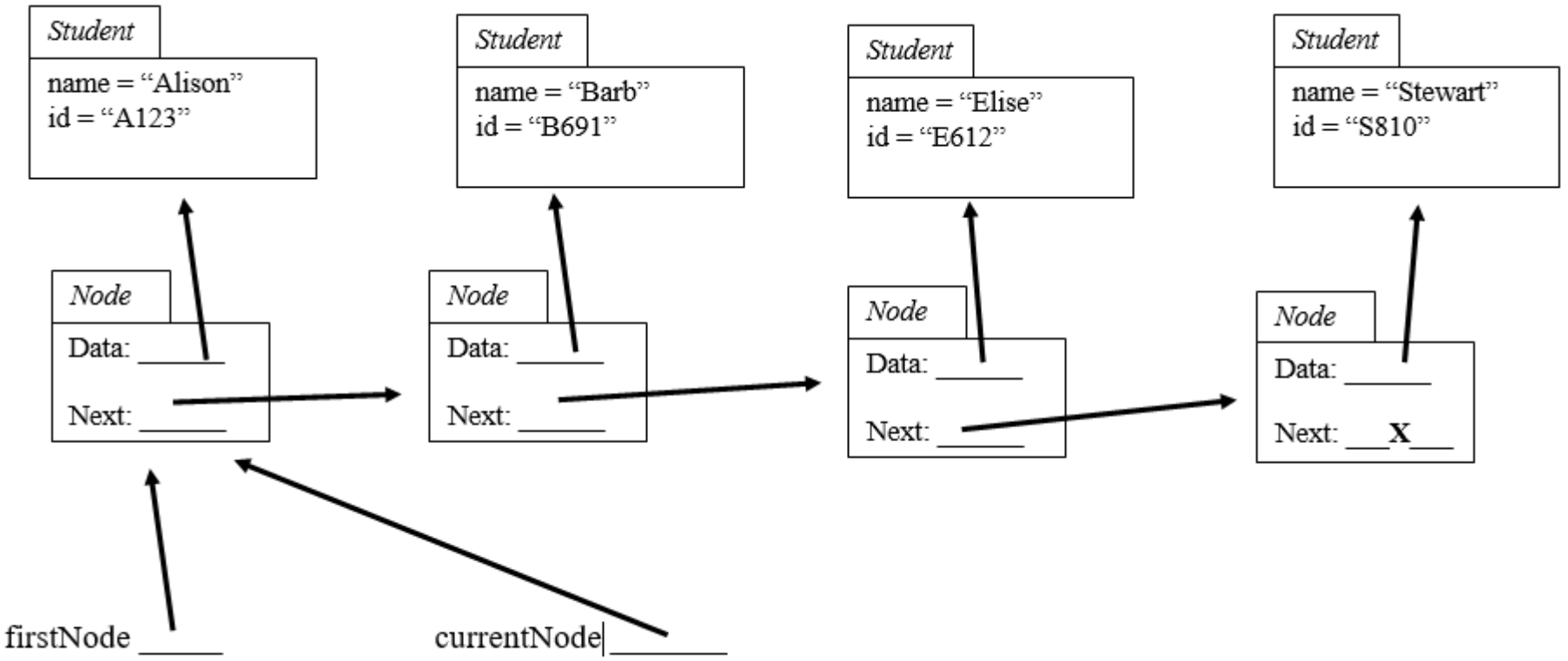
Iterating the Chain

```
while(currentNode != null) {
      Student currentStudent = currentNode.data;
      // process the Student
      currentNode = currentNode.next;
}
```

- Keep looping as long as the temporary node contains data (it is not null)
- Get the data from the current node and process it
- Advance the temporary node to the next in the list

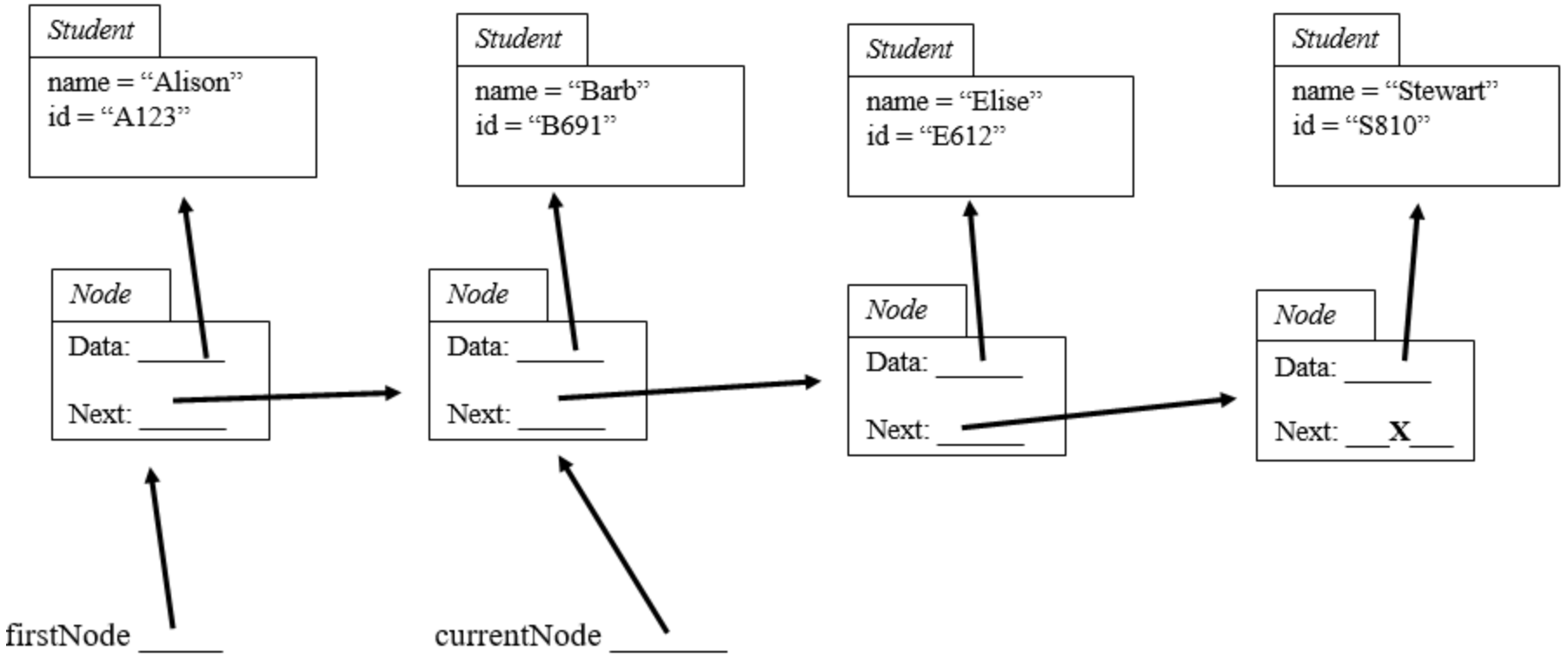
Iterating the Chain

```
while(currentNode != null) {
      Student currentStudent = currentNode.data;
      // process the Student
      currentNode = currentNode.next;
}
```



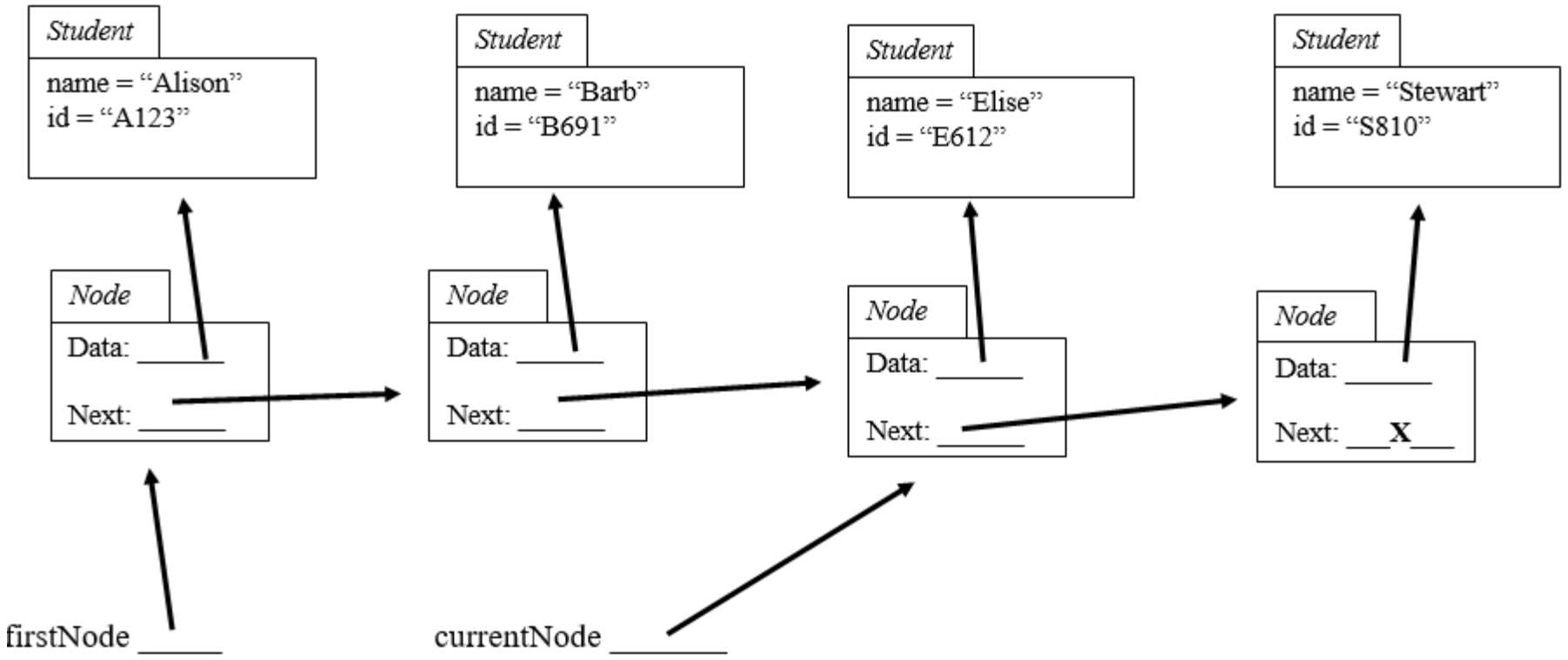
Iterating the Chain

```
while(currentNode != null) {
      Student currentStudent = currentNode.data;
      // process the Student
      currentNode = currentNode.next;
}
```



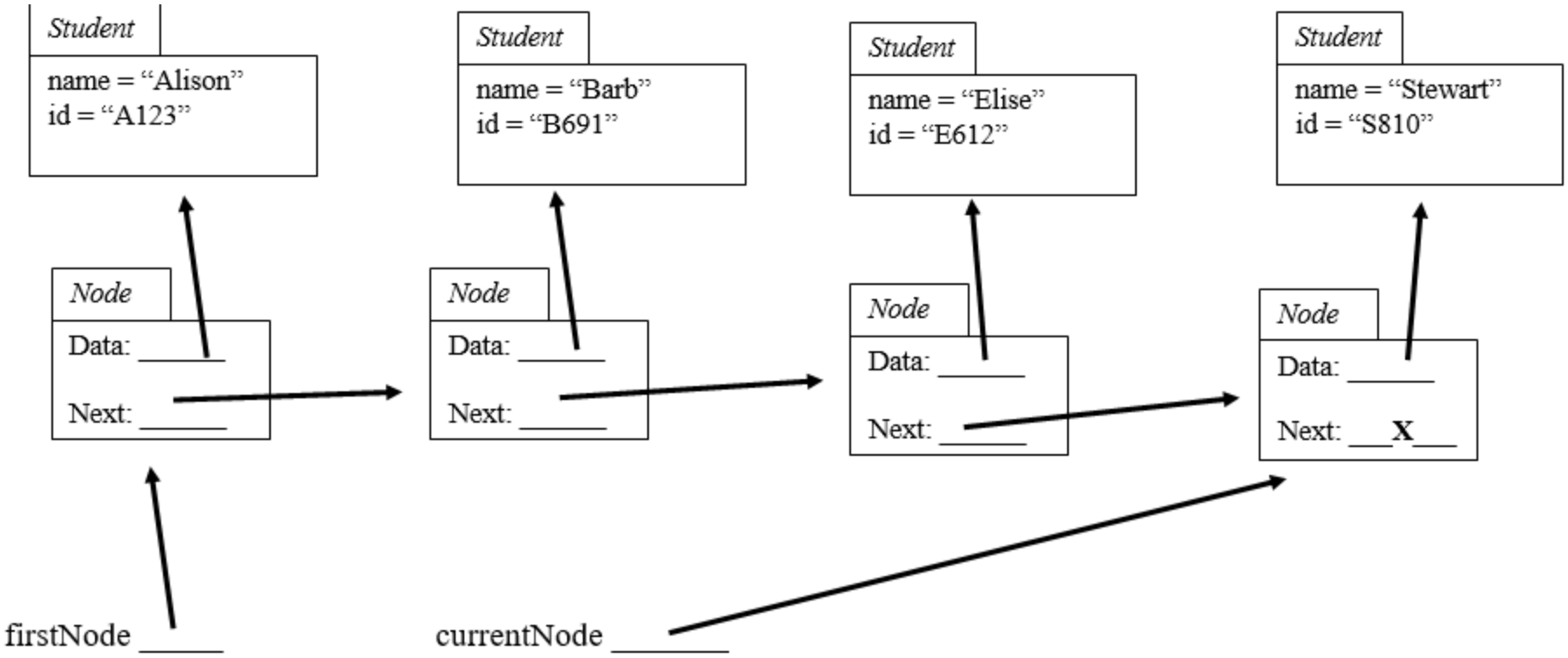
Iterating the Chain

```
while(currentNode != null) {
      Student currentStudent = currentNode.data;
      // process the Student
      currentNode = currentNode.next;
}
```



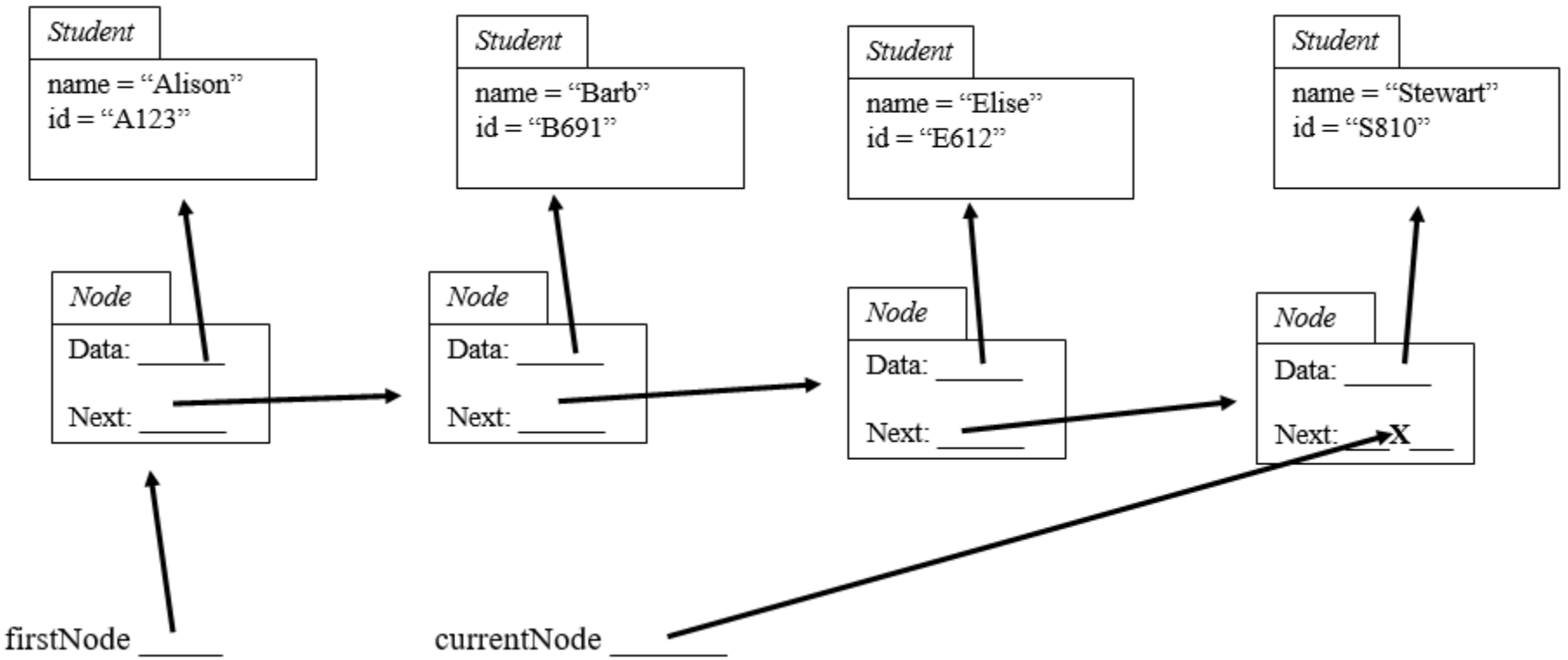
Iterating the Chain

```
while(currentNode != null) {
      Student currentStudent = currentNode.data;
      // process the Student
      currentNode = currentNode.next;
}
```



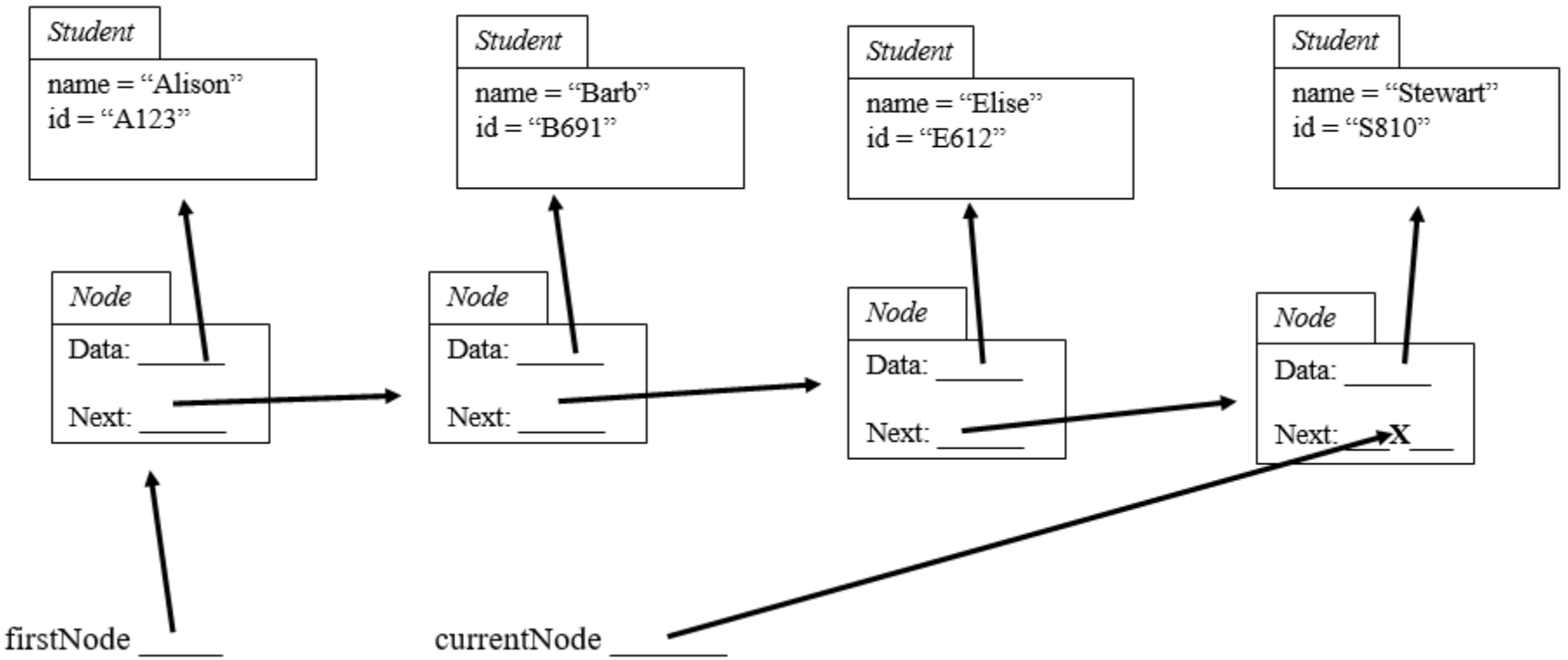
Iterating the Chain

```
while(currentNode != null) {
      Student currentStudent = currentNode.data;
      // process the Student
      currentNode = currentNode.next;
}
```



Iterating the Chain

```
while(currentNode != null) {
      Student currentStudent = currentNode.data;
      // process the Student
      currentNode = currentNode.next;
}
```



Iterating the Chain

```
Node<Student> currentNode = firstNode;

while(currentNode != null) {
     Student currentStudent = currentNode.data;
     // process the Student
     currentNode = currentNode.next;
}
```

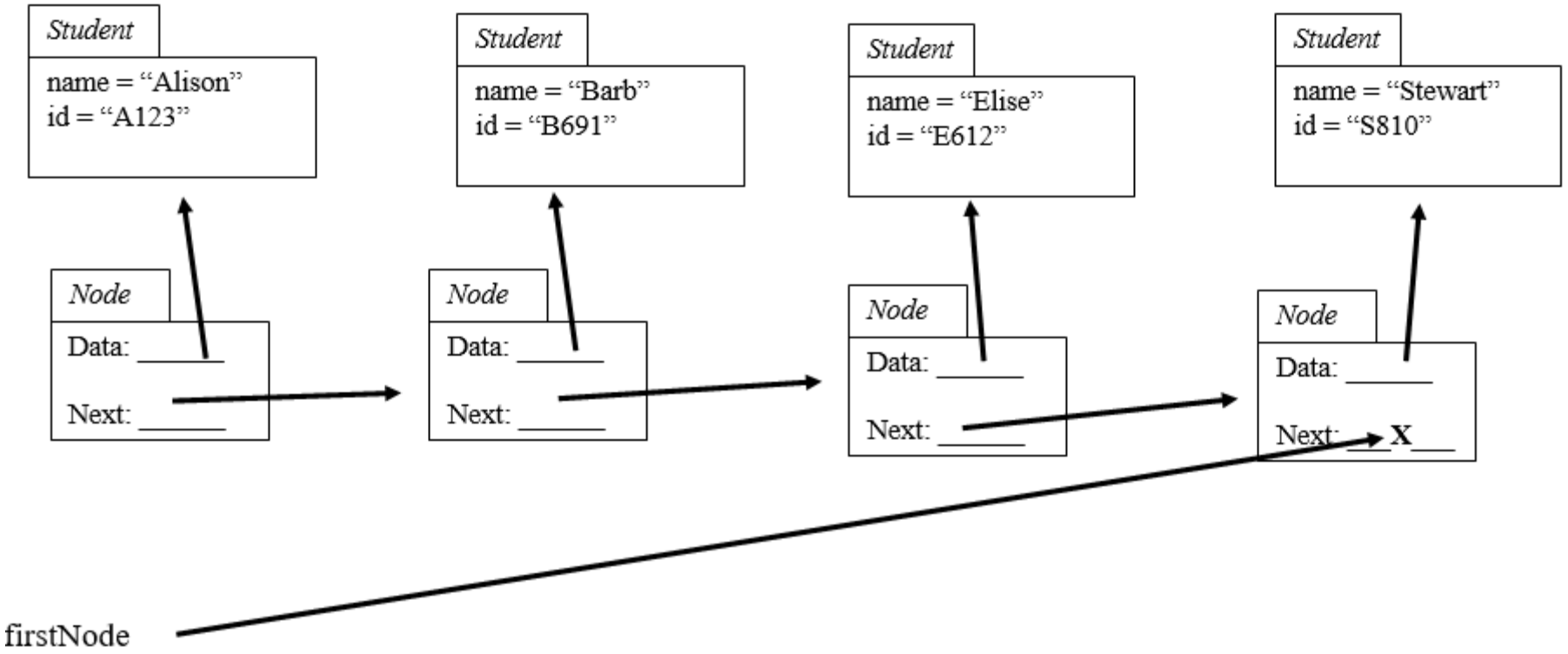
Iterating the Chain

- Error!
- Do not iterate using firstNode- this will modify the actual chain!

```
while(firstNode != null) {
      Student currentStudent = firstNode.data;
      // process the Student
       firstNode = firstNode.next;
}
```

Iterating the Chain

- Error!



Iterating the Chain

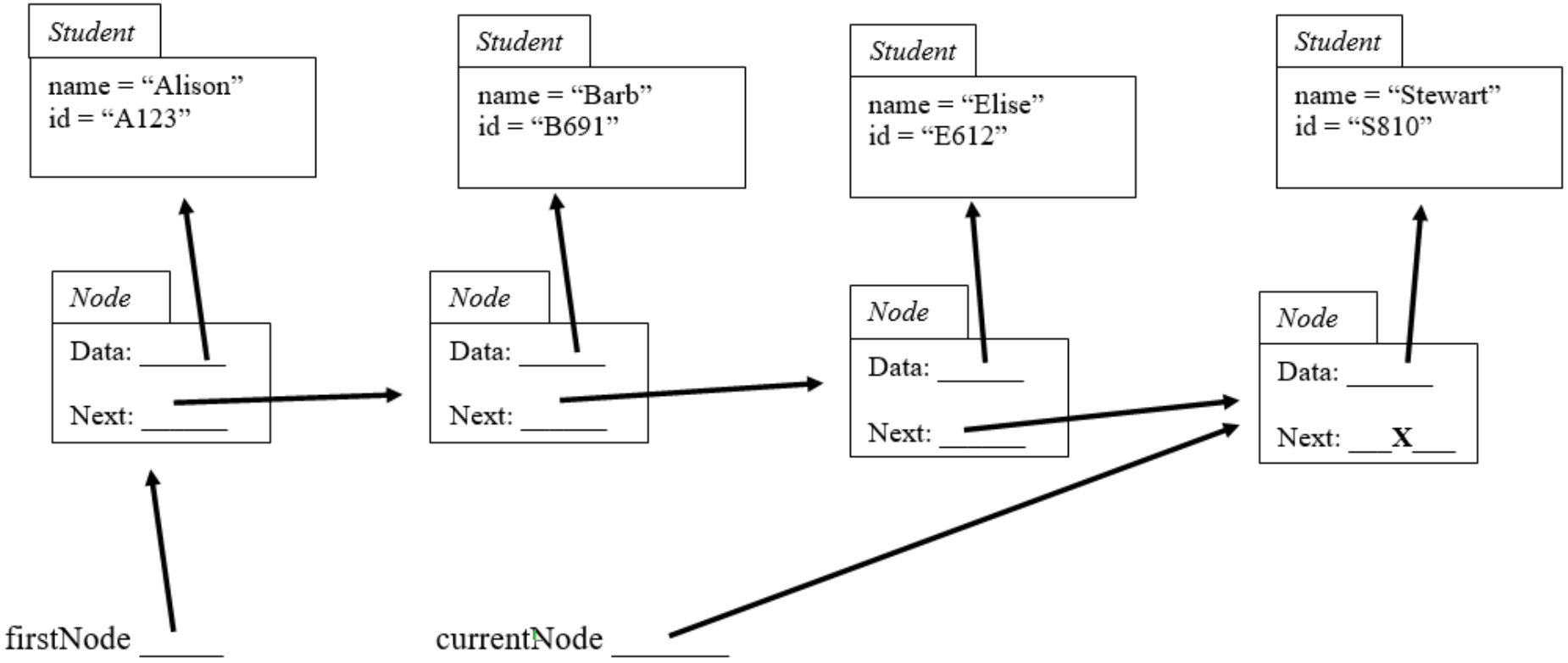
- Error!
- Do not iterate until current.next is null- this will miss the last element in the chain!

```
Node<Student> currentNode = firstNode;

while(currentNode.next != null) {
      Student currentStudent = currentNode.data;
      // process the Student
      currentNode = currentNode.next;
}
```

Iterating the Chain

- Error!



Inserting an Element

- There are three cases to consider:
  - inserting at the beginning
  - inserting in the middle
  - inserting at the end
- We must also make sure that our approaches work for empty and singleton lists!

Inserting at the End

- Approach:
  - Find the last element in the chain.
  - Insert the new node as that element's next.

```
Node lastNode = firstNode;
while(lastNode.next != null) {
// rare case when you iterate only until lastNode.next isn't null!

    lastNode = lastNode.next;
}
lastNode.next = newNode;
```

- Check special cases for crashing: empty bag, singleton bag
  - Empty list: firstNode is null, but then so is lastNode! So the code will crash.

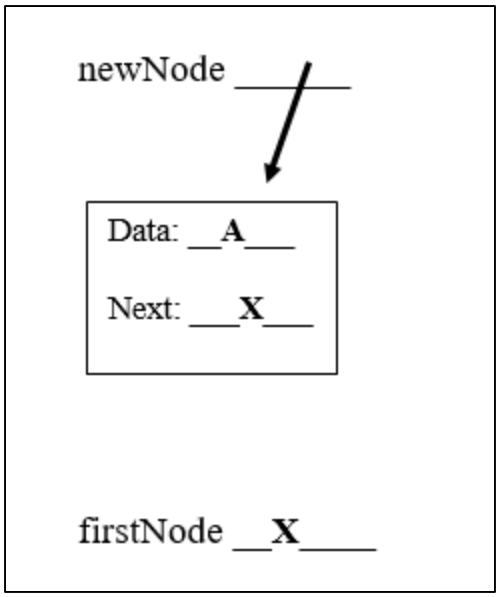
Inserting at the End

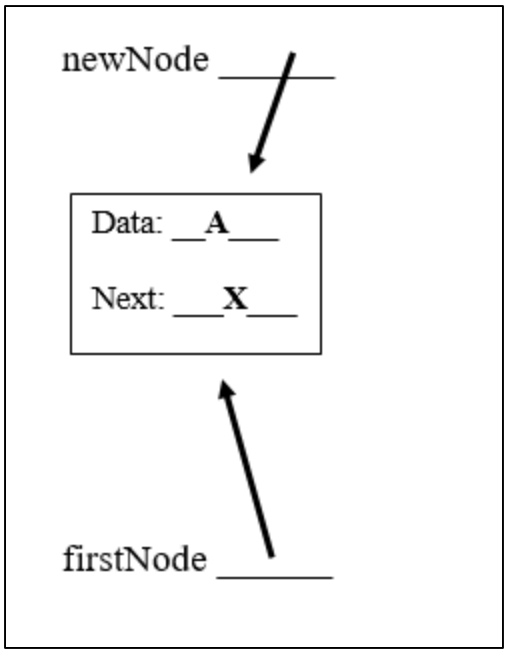
- Check special cases for crashing: empty bag, singleton bag
  - Empty bag: firstNode is null, but then so is lastNode! So the code will crash.

```
if(firstNode == null) { // empty list
    firstNode = newNode;
} else {
    Node lastNode = firstNode;
    while(lastNode.next != null) {
        lastNode = lastNode.next;
    }
    lastNode.next = newNode;
}
```

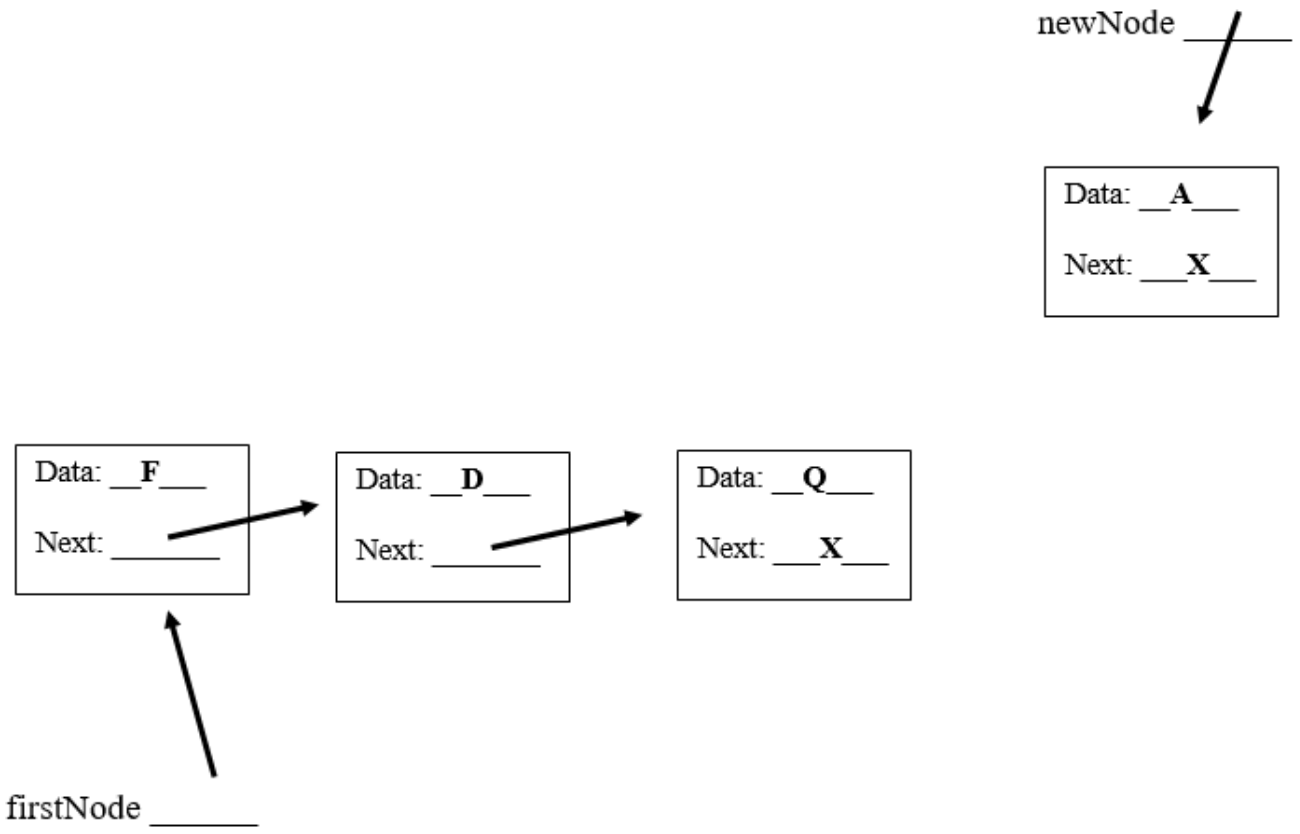
Inserting at the End- Empty

```
if(firstNode == null) { // empty list
    firstNode = newNode;
}
```



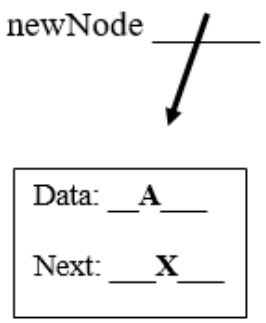


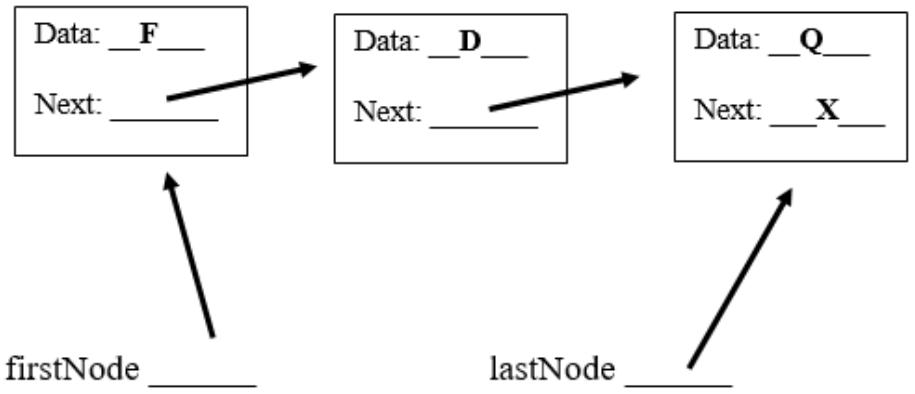
Inserting at the End- Non-Empty



Inserting at the End- Non-Empty

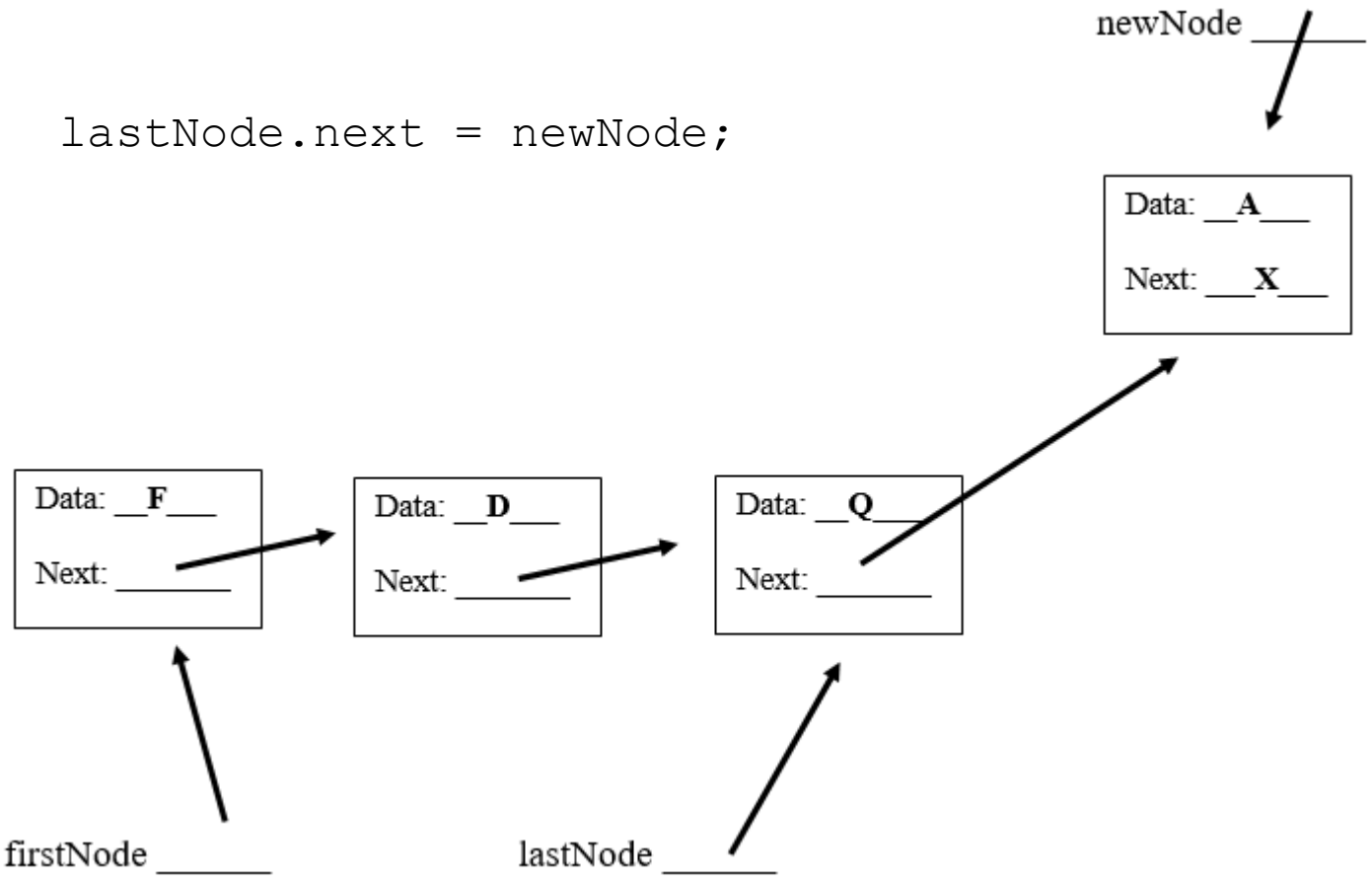
```
Node lastNode = firstNode;
while(lastNode.next != null) {
   lastNode = lastNode.next;
}
```





Inserting at the End- Non-Empty

```
lastNode.next = newNode;
```



Inserting at the Beginning

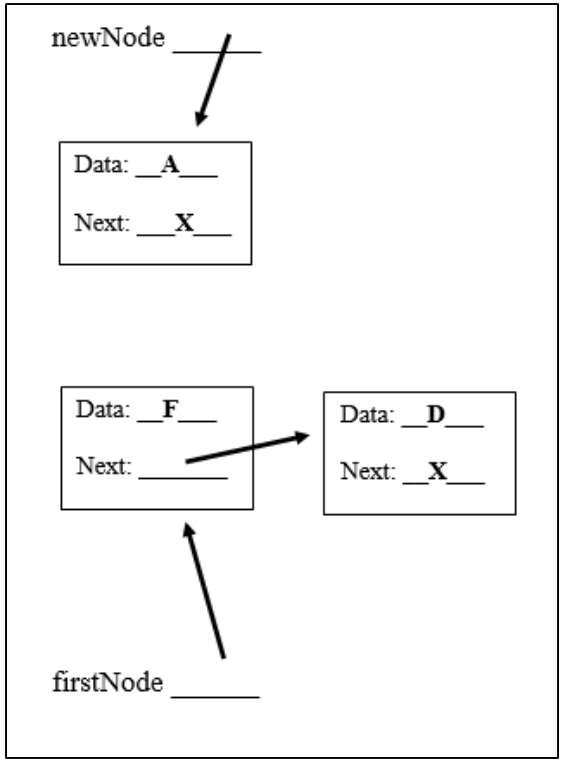
- Approach:
  - Make the new node point to the firstNode as next
  - Make firstNode refer to the new node

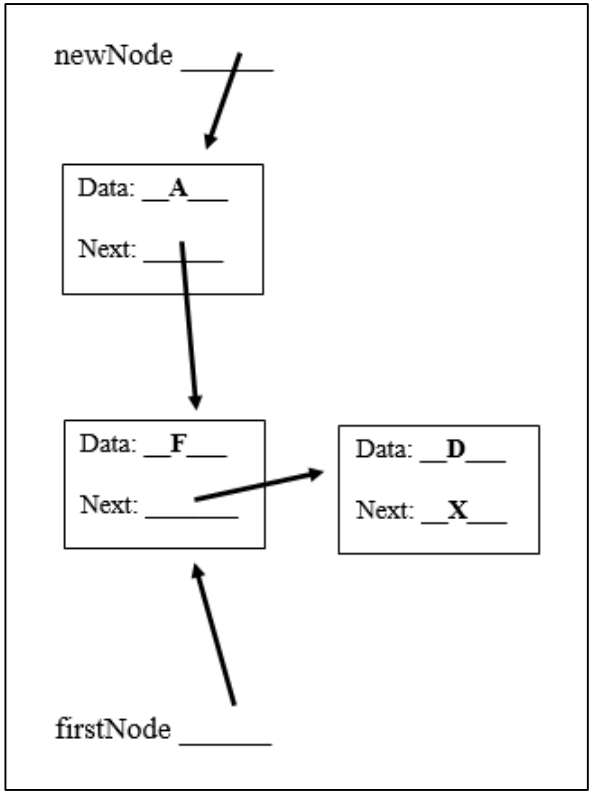
```
newNode.next = firstNode;
firstNode = newNode;
```

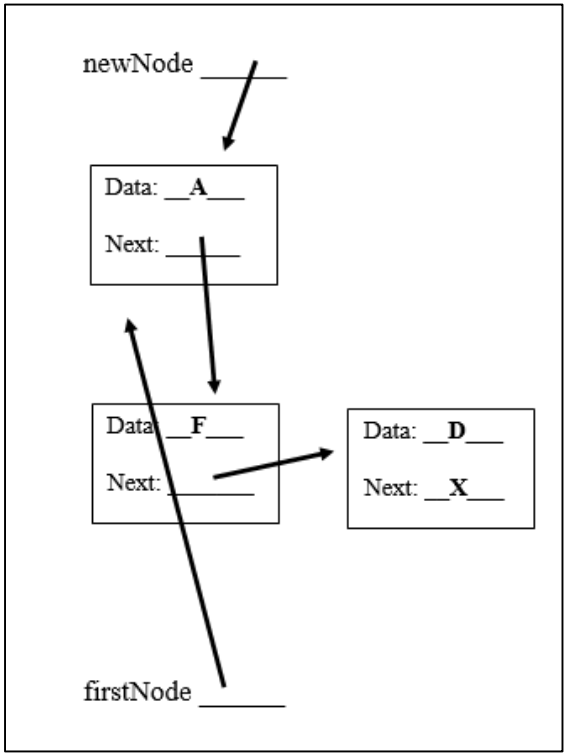
- Check special cases for crashing: empty list, singleton list
- Think about how this compares to inserting at the beginning of an array!

Inserting at the Beginning- Non-Empty

```
newNode.next = firstNode;
firstNode = newNode;
```

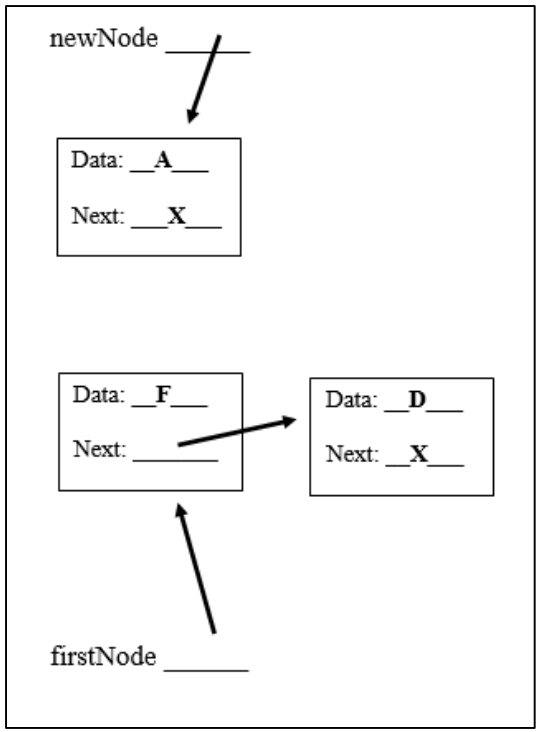


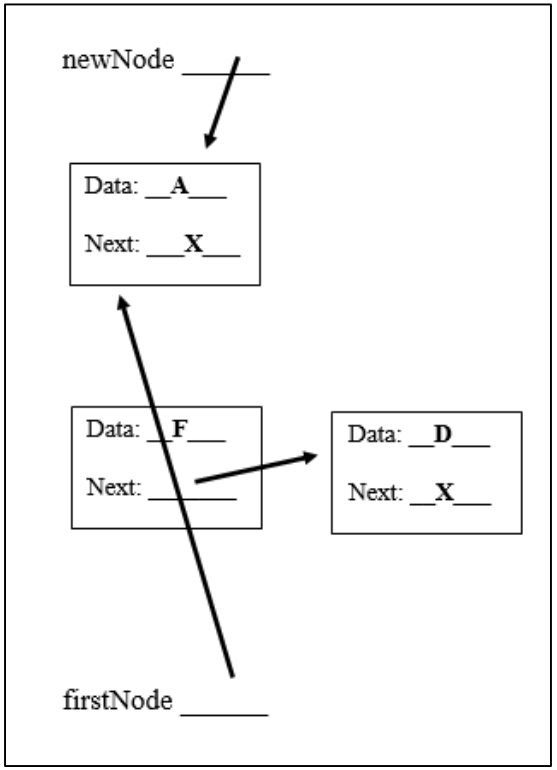


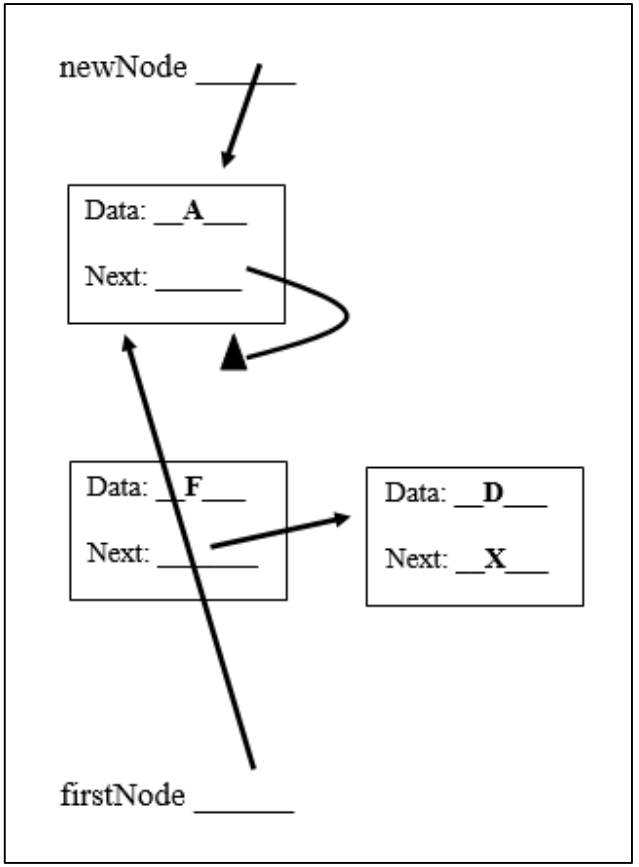


Inserting at the Beginning- Error!

```
firstNode = newNode;
newNode.next = firstNode;
```







Inserting in the Middle

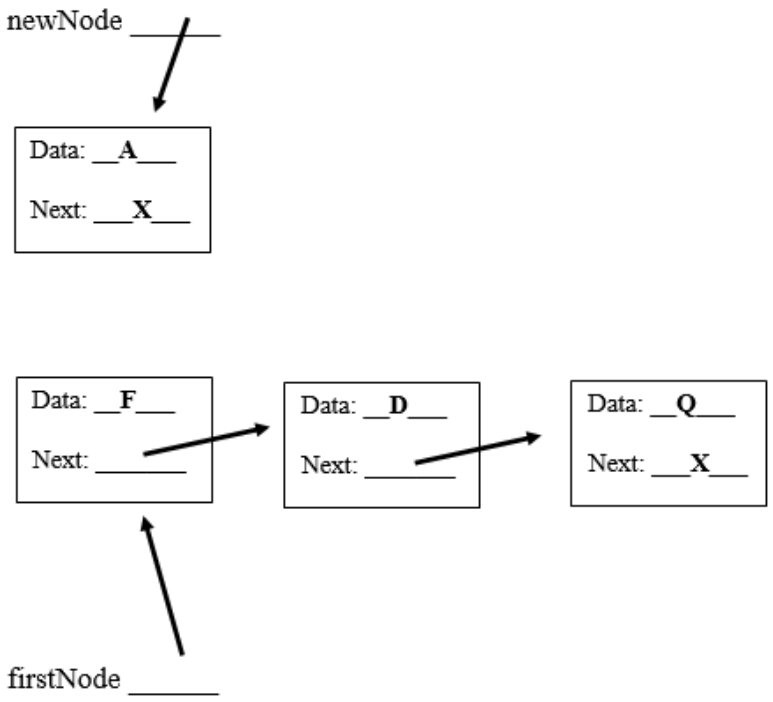
- Approach:
  - Set previous node's next to the new node
  - Set the new node's next to the previous node's old next

```
newNode.next = nodeBefore.next;
nodeBefore.next = newNode;
```

- This requires us to find nodeBefore!
  - We can traverse to the insertPosition – 1 to get this node.
- Check special cases for crashing: empty list, singleton list

Inserting in the Middle- Non-Empty

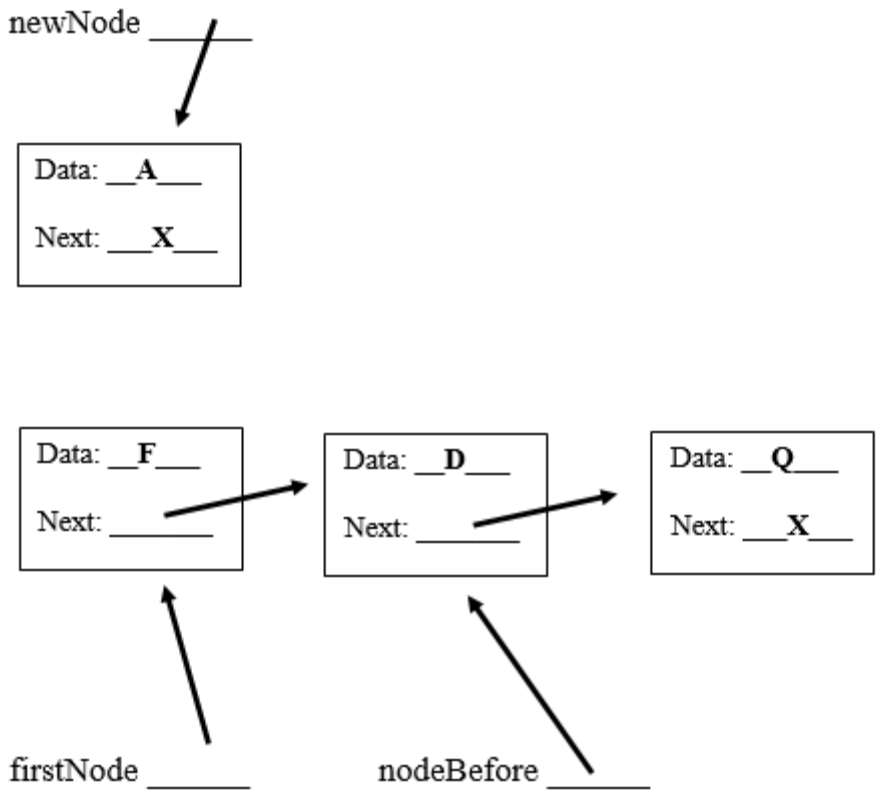
```
newNode.next = nodeBefore.next;
nodeBefore.next = newNode;
```



- Insert into position 3 (between the D and Q)

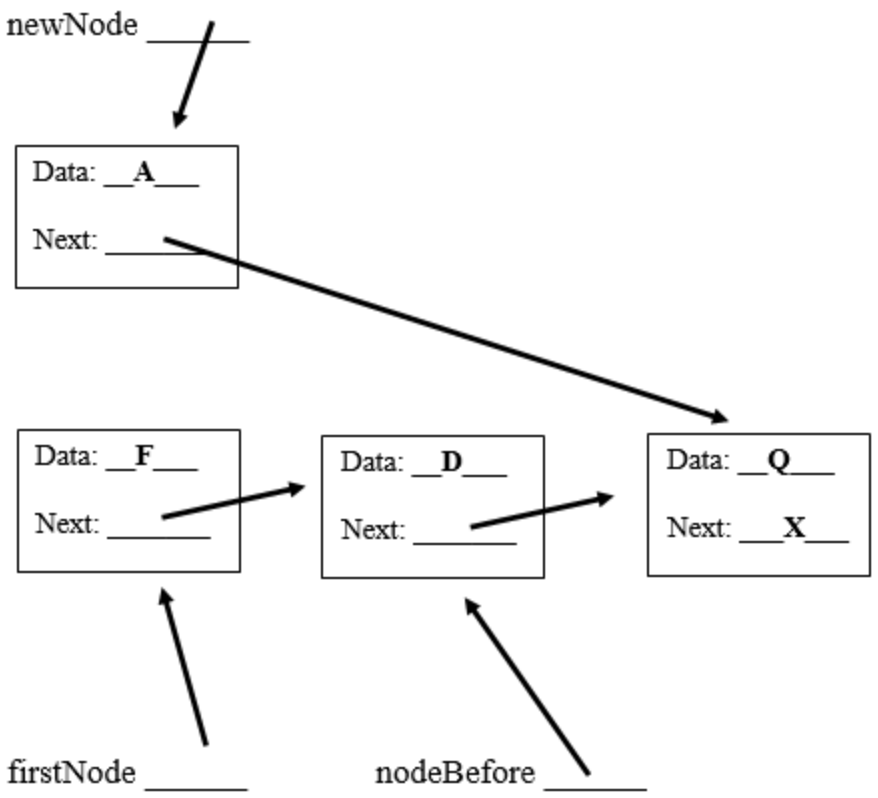
Inserting in the Middle- Non-Empty

```
newNode.next = nodeBefore.next;
nodeBefore.next = newNode;
```



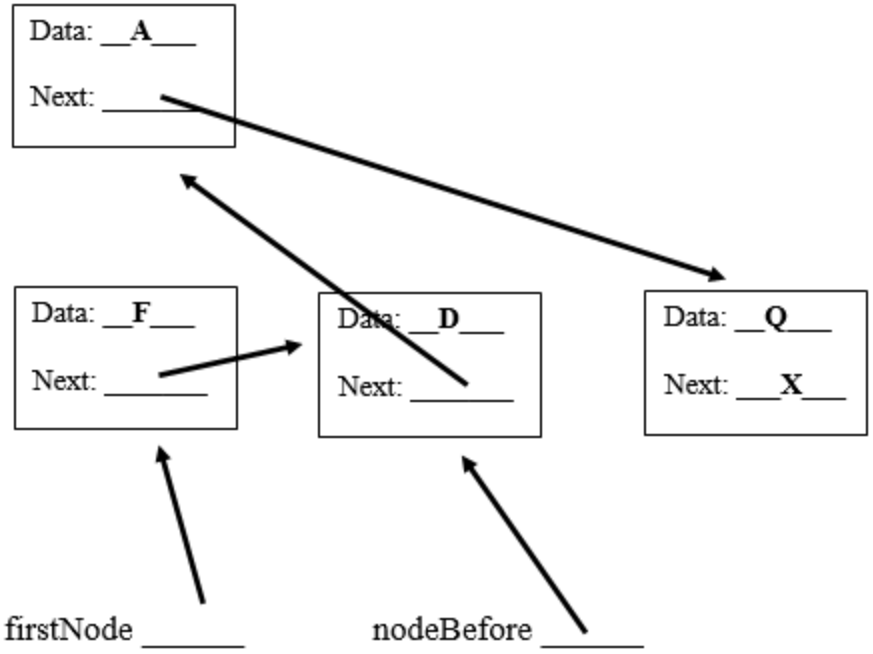
Inserting in the Middle- Non-Empty

```
newNode.next = nodeBefore.next;
nodeBefore.next = newNode;
```



Inserting in the Middle- Non-Empty

```
newNode.next = nodeBefore.next;
nodeBefore.next = newNode;
```



Removing an Element

- There are three cases to consider:
  - removing from the beginning
  - removing from the middle
  - removing from the end
- We must also make sure that our approaches work for empty and singleton lists!

Removing from the Beginning

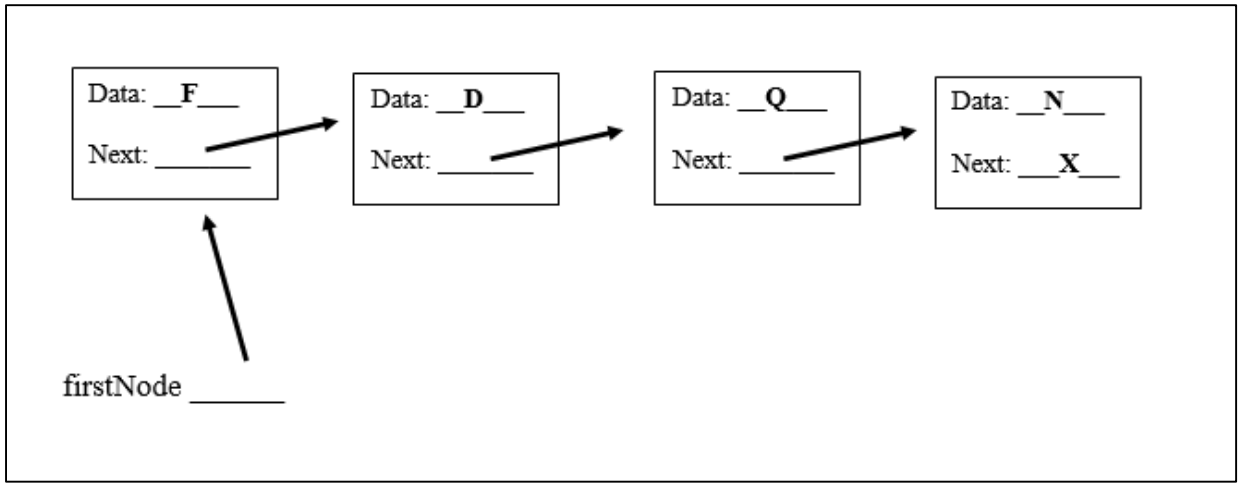
- Approach:
  - Make firstNode point to firstNode.next

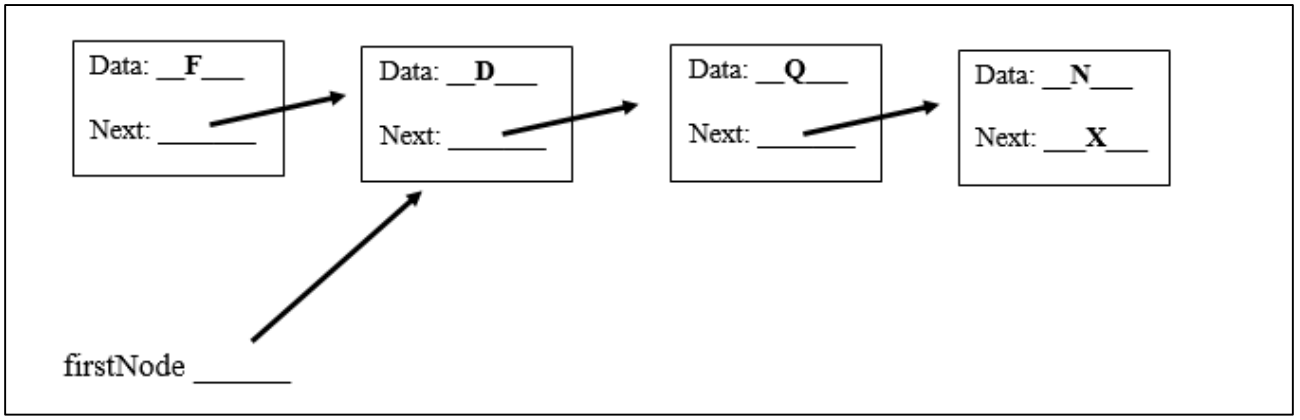
```
firstNode = firstNode.next;
```

- Check special cases for crashing: empty bag, singleton bag
  - Empty list: firstNode is null, so firstNode.next will crash
- Again think about how this compares to removing the beginning of an array!

Removing from the Beginning- Non-Empty

```
firstNode = firstNode.next;
```





Removing from the Middle

- Approach:
  - Set previous node's next to the to-be-removed node's next

```
nodeBefore.next = nodeToRemove.next;
```

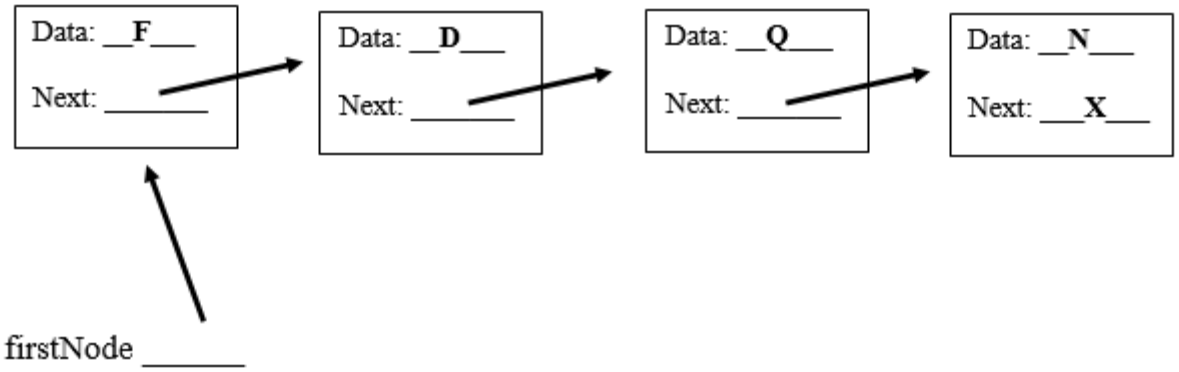
or

```
nodeBefore.next = nodeBefore.next.next;
```

- This again requires us to find nodeBefore!
  - We can traverse to the removePosition – 1 to get this node.
- Check special cases for crashing: empty list, singleton list

Removing from the Middle- Non-Empty

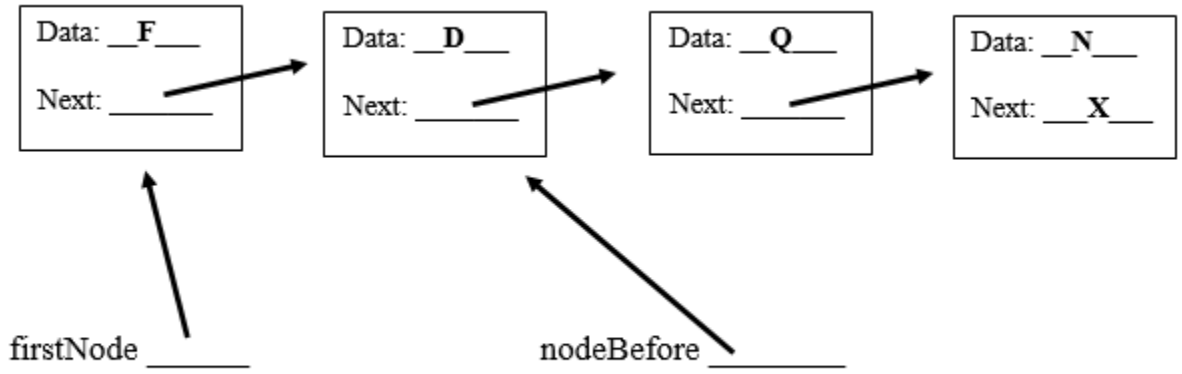
```
nodeBefore.next = nodeToRemove.next;
```



- Remove the third node (the Q node)

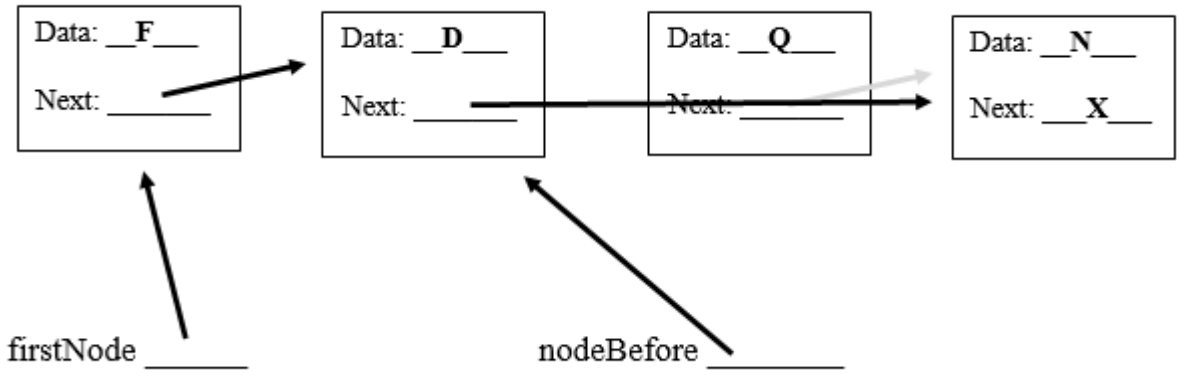
Removing from the Middle- Non-Empty

```
nodeBefore.next = nodeToRemove.next;
```



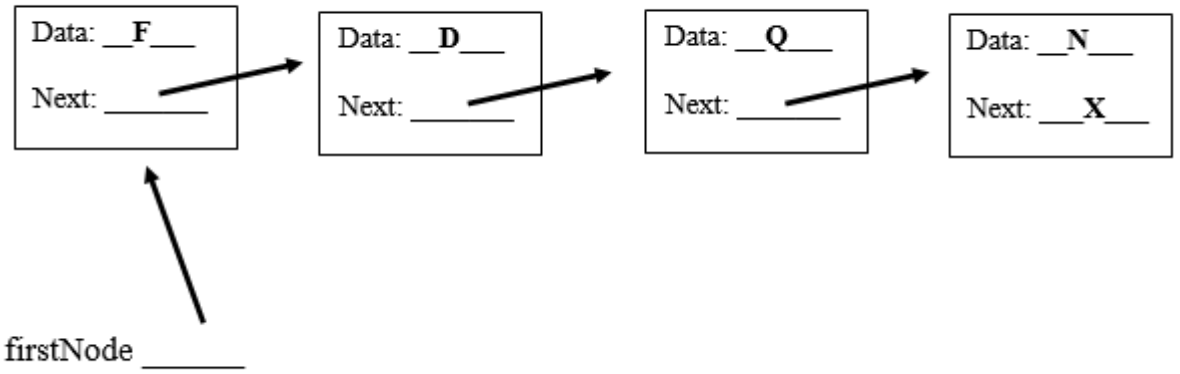
Removing from the Middle- Non-Empty

```
nodeBefore.next = nodeToRemove.next;
```

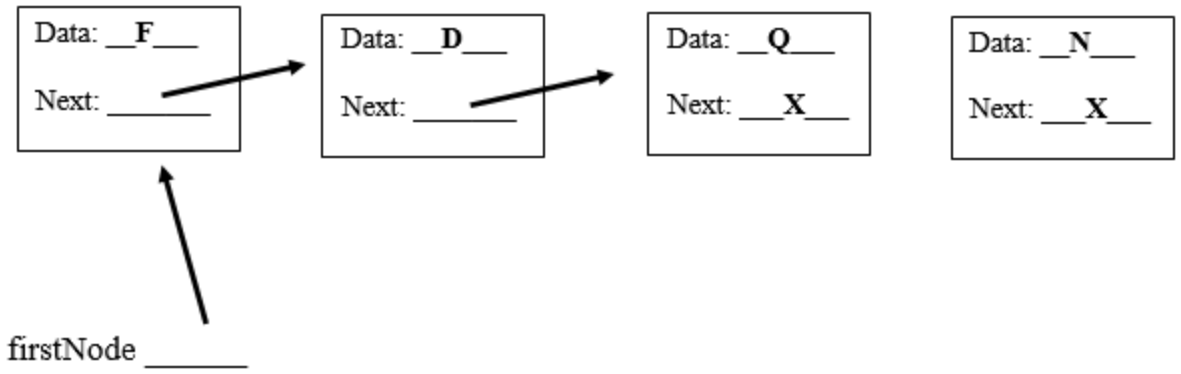


Removing from the End

- Can we reuse the same approach we used for removing from the middle?



Removing from the End- Non-Empty



Node References and Assignment

- When traversing a chain of linked nodes, you will use a temporary node reference to point to each node in the chain.

```
Node currentNode = firstNode;
```

- This line of code creates an *alias* between currentNode and firstNode.
  - This means that currentNode and firstNode point to the exact same place in memory (a Node object).
- It's critical to understand the difference between **re-assigning** currentNode to a new reference and **making changes** to the contents of currentNode.

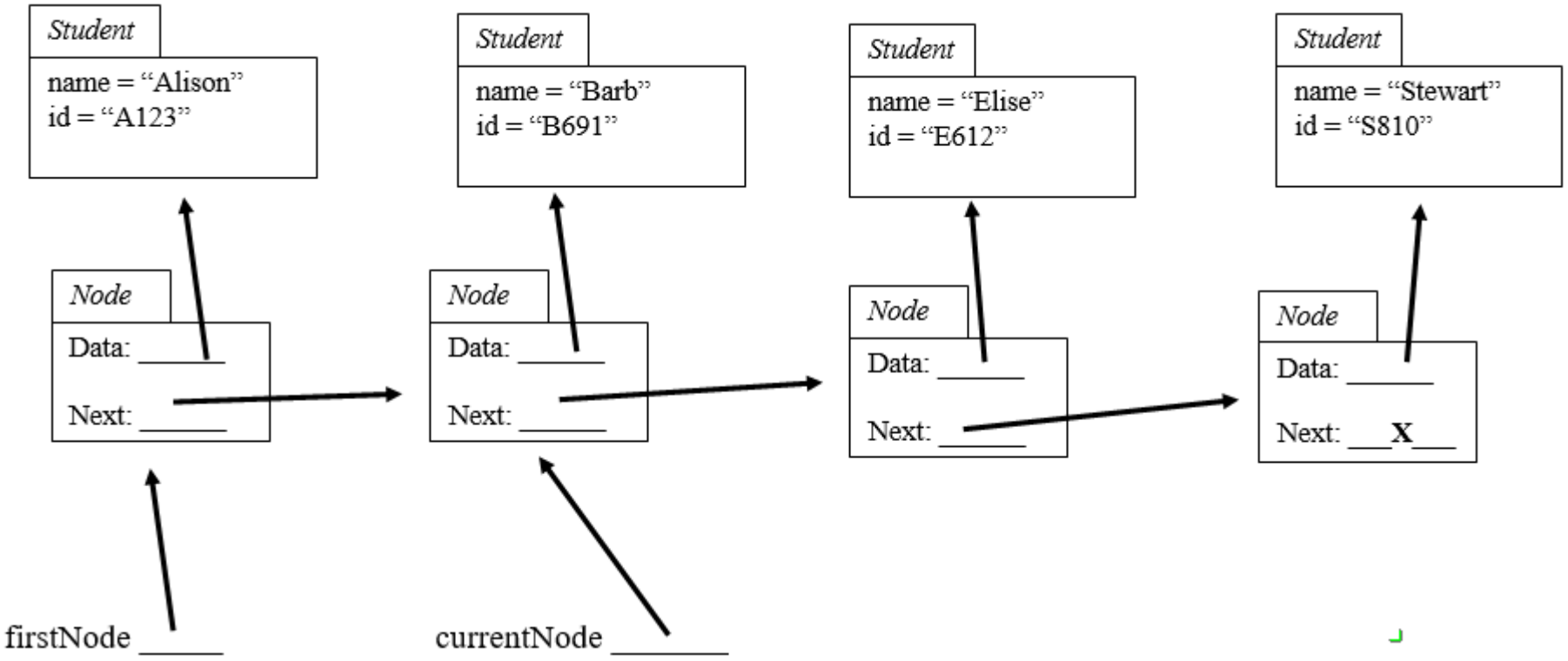
Re-Assigning currentNode

- If you re-assign currentNode to point to a different node, like this:

```
currentNode = someOtherNode;
currentNode = someOtherNode.next;
```

- This breaks the alias link between currentNode and firstNode. They are no longer aliases.
- currentNode now points to a new place in memory (another Node object).
- But firstNode has **not** been changed by this line of code! firstNode remains unchanged, although it is no longer an alias with currentNode.
- This means that the entire structure of the linked chain (which is headed by firstNode) has not changed.

Re-Assigning currentNode



Re-Assigning currentNode

- This is the correct way to iterate a chain.
- You advance the current reference through the chain, but no changes are made to the chain.
- You do this by putting currentNode on the left side of an assignment:

```
currentNode = someNode;
currentNode = currentNode.next;
currentNode = someNode.next;
```

Changing the Data through CurrentNode

- Let's say that currentNode and firstNode again start as aliases:

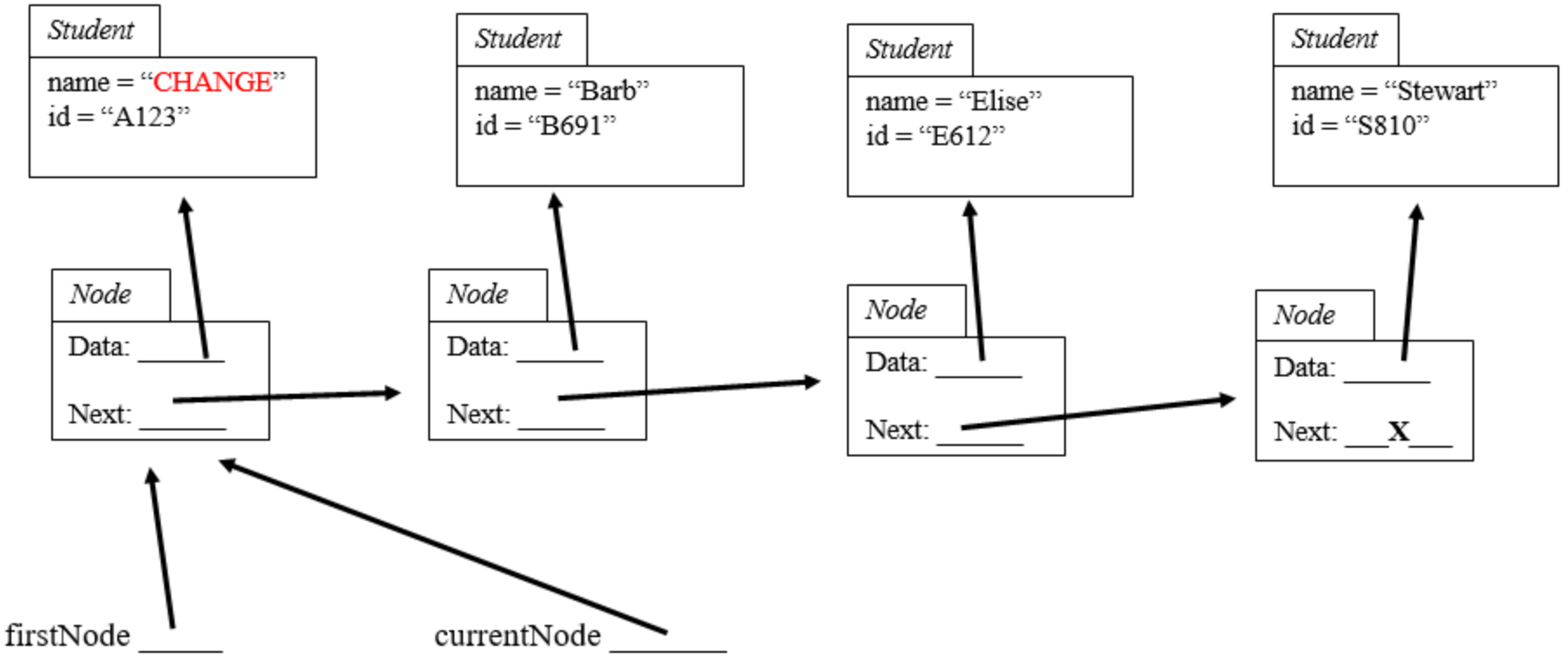
```
Node currentNode = firstNode;
```

- If you change something about the ***value*** of currentNode, like this:

```
currentNode.data = 14;
currentNode.data.setName("CHANGED!")
```

- the change will also happen to firstNode- because they are aliases!
  - Changing the data value of one reference changes it for the other- because they refer to the same place in memory!

Changing the Data through CurrentNode



Changing the Data through CurrentNode

- This is how you can change something about the chain using a temporary node.
- You advance the current reference through the chain, and then make a change to the data through currentNode.
- You do this by putting currentNode.data on the left side of an assignment:

```
currentNode.data = newData;
currentNode.data.setNewData(…);
```

Changing the Chain through CurrentNode

- Let's say that currentNode and firstNode again start as aliases:

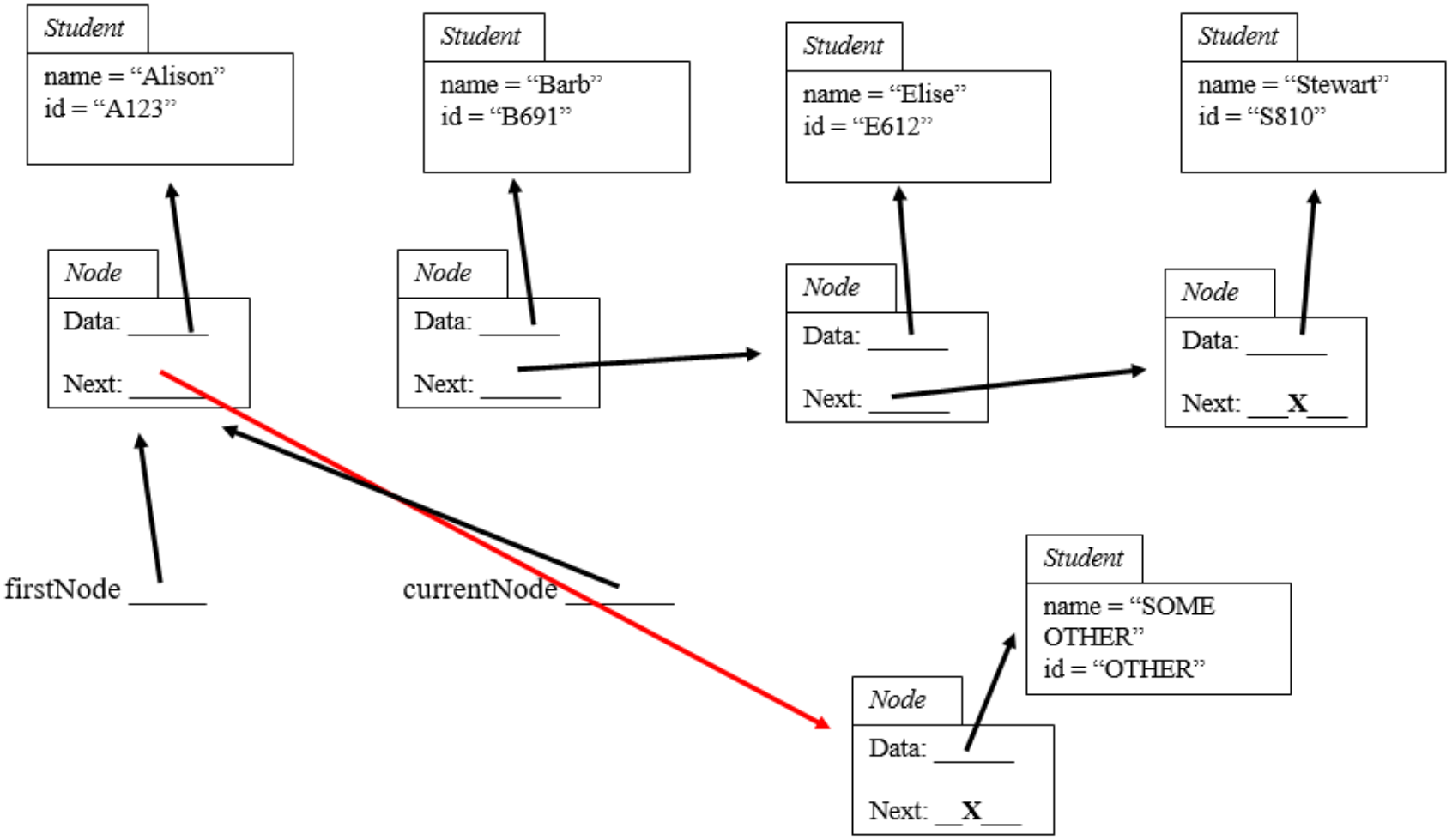
```
Node currentNode = firstNode;
```

- If you change something about the ***next reference*** of currentNode, like this:

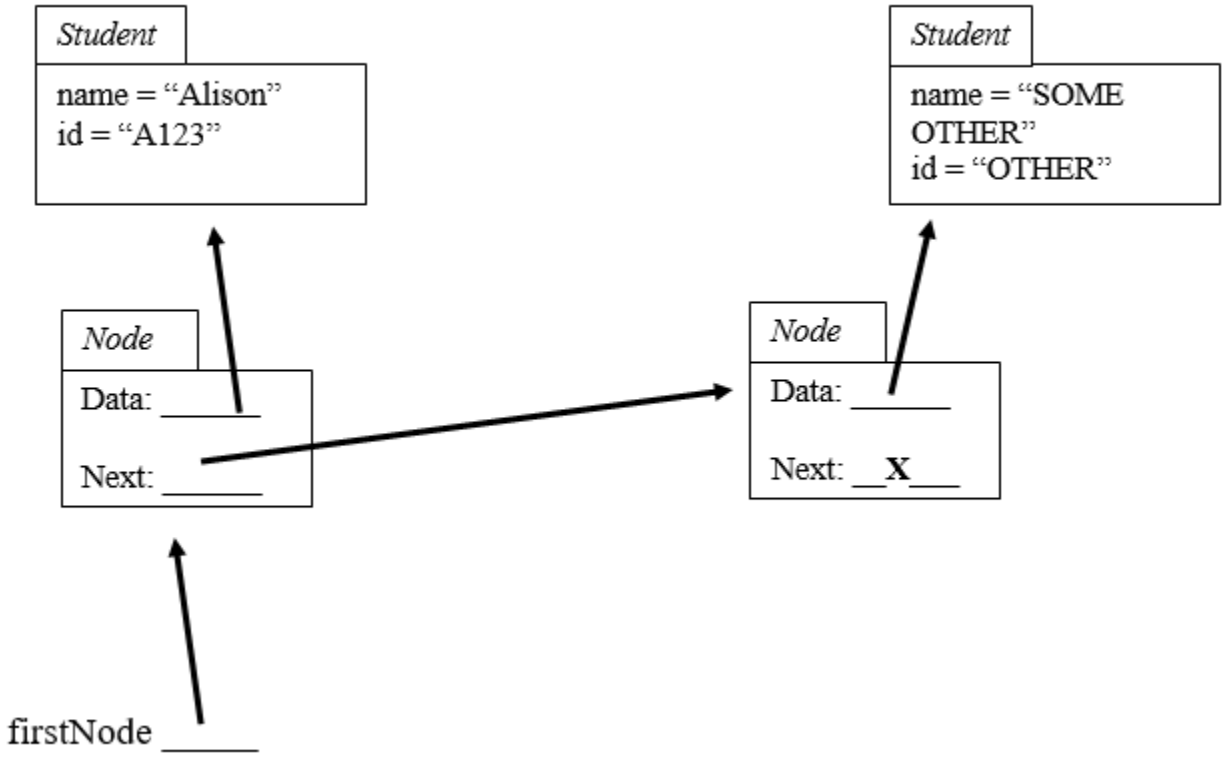
```
currentNode.next = someOtherNode;
currentNode.next = someOtherNode.next;
currentNode.next = currentNode.next.next;
```

- the change will also happen to firstNode- because they are aliases!
  - Changing the next value of one reference changes it for the other- because they refer to the same place in memory!
- This changes the actual *structure* of the chain headed by firstNode.

Changing the Chain through CurrentNode



Changing the Chain through CurrentNode



Changing the Chain through CurrentNode

- This is how you can change *the structure of the chain* using a temporary node.
- You advance the current reference through the chain, and then make a change to the next node through currentNode.
- You do this by putting currentNode.next on the left side of an assignment:

```
currentNode.next = newNode;
currentNode.next = someNode.next;
currentNode.next = currentNode.next.next;
```

Node References and Assignment

- It's critical to understand the distinction between changing the value of currentNode (pointing it to a new node) and changing the value of the instance data variables of currentNode (which changes those variables for the alias, too).

- currentNode = ... some node (could be a node or a node.next)
  - changes what currentNode refers to
  - does **not** change the actual chain

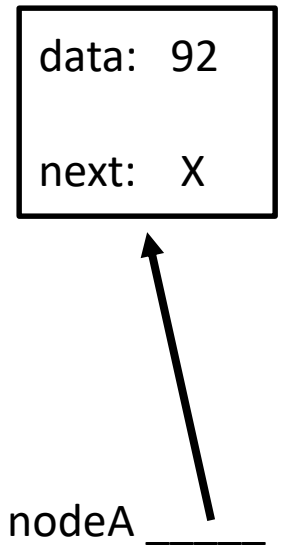
- currentNode.data = ... some data
- currentNode.data.someMethodThatModifiesTheData();
  - changes the data contained in the actual chain

- currentNode.next = ... some node (could be a node or node.next)
  - changes the structure of the actual chain

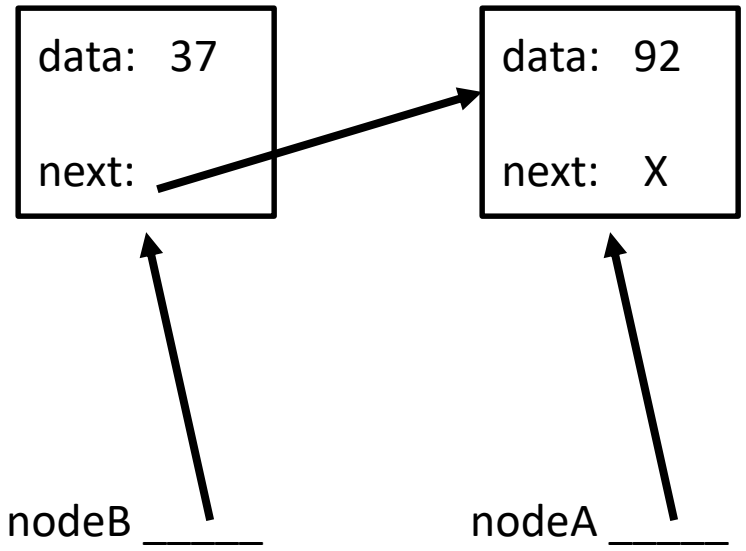
Example Node Trace

- Review a trace of the example code.
  - Remember the details about assignments statements, node references, and node assignments!

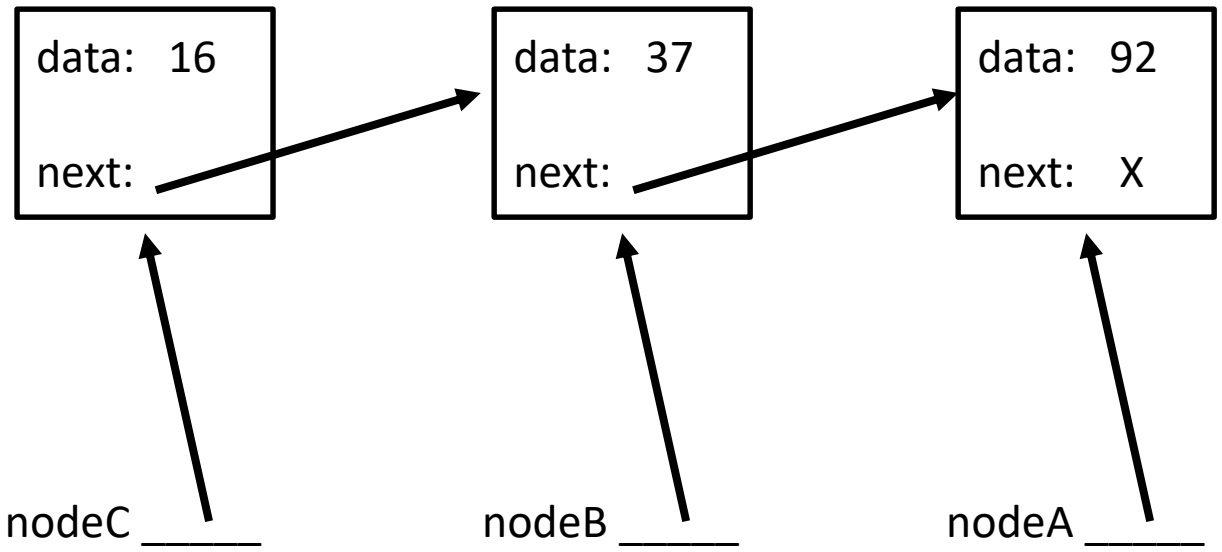
```
Node<Integer> nodeA = new Node<Integer>(92);
// creates a new, named node
```



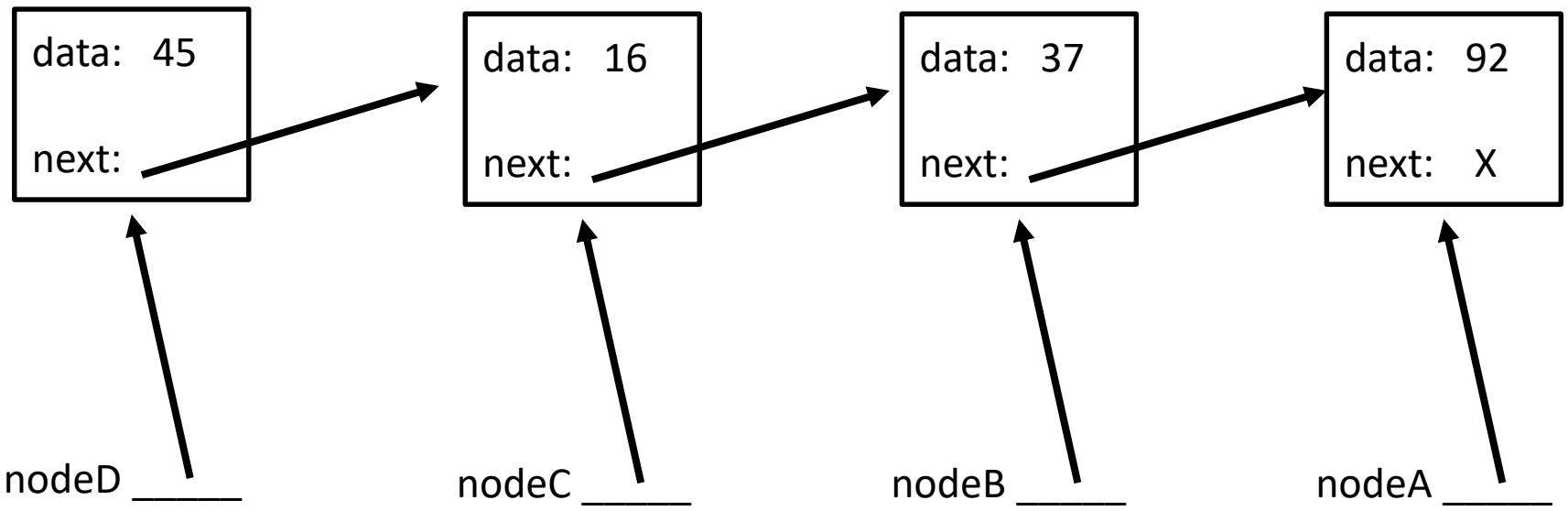
```
Node<Integer> nodeB = new Node<Integer>(37, nodeA);
// creates a new, named node with nodeA as next
```



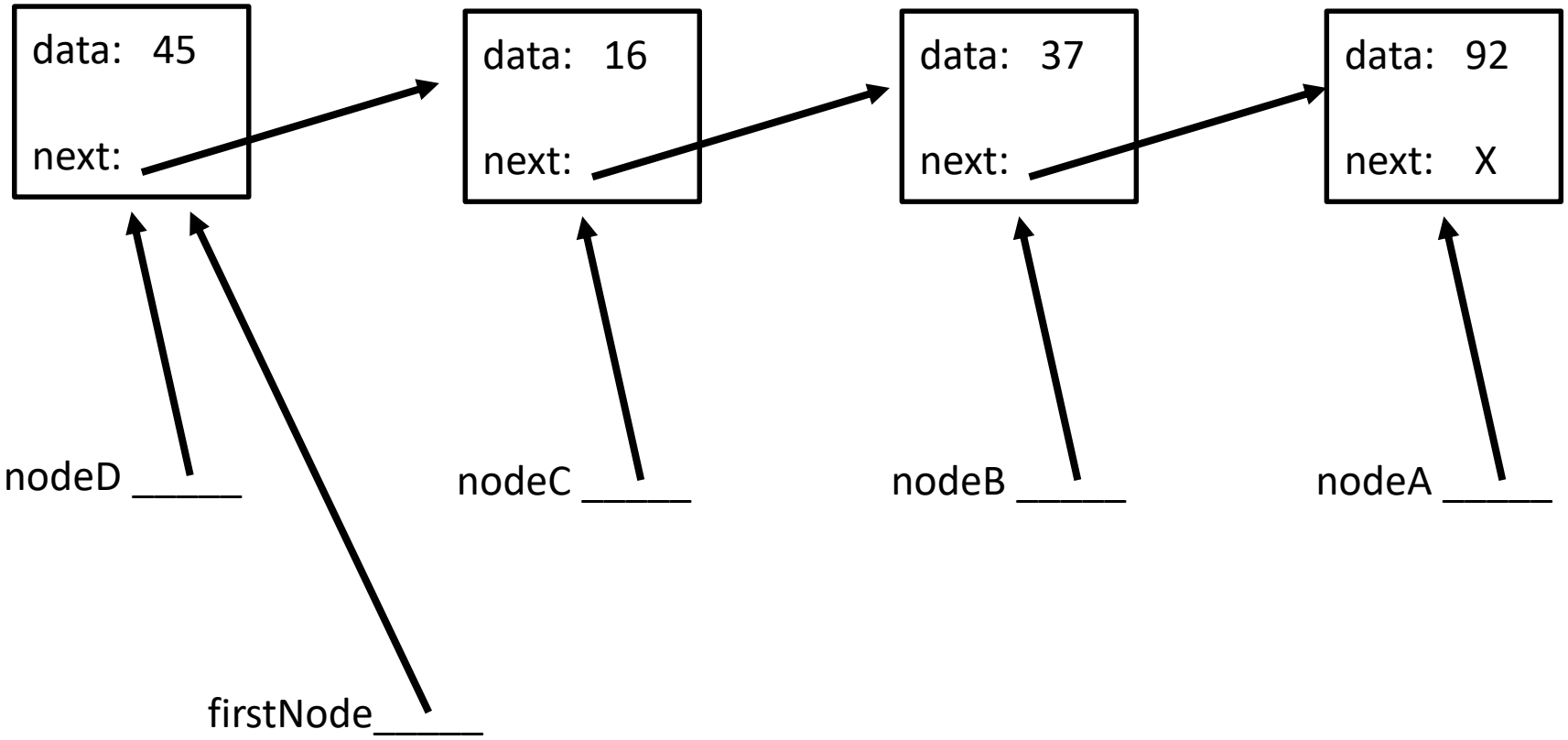
```
Node<Integer> nodeC = new Node<Integer>(16, nodeB);
// creates a new, named node with nodeB as next
```



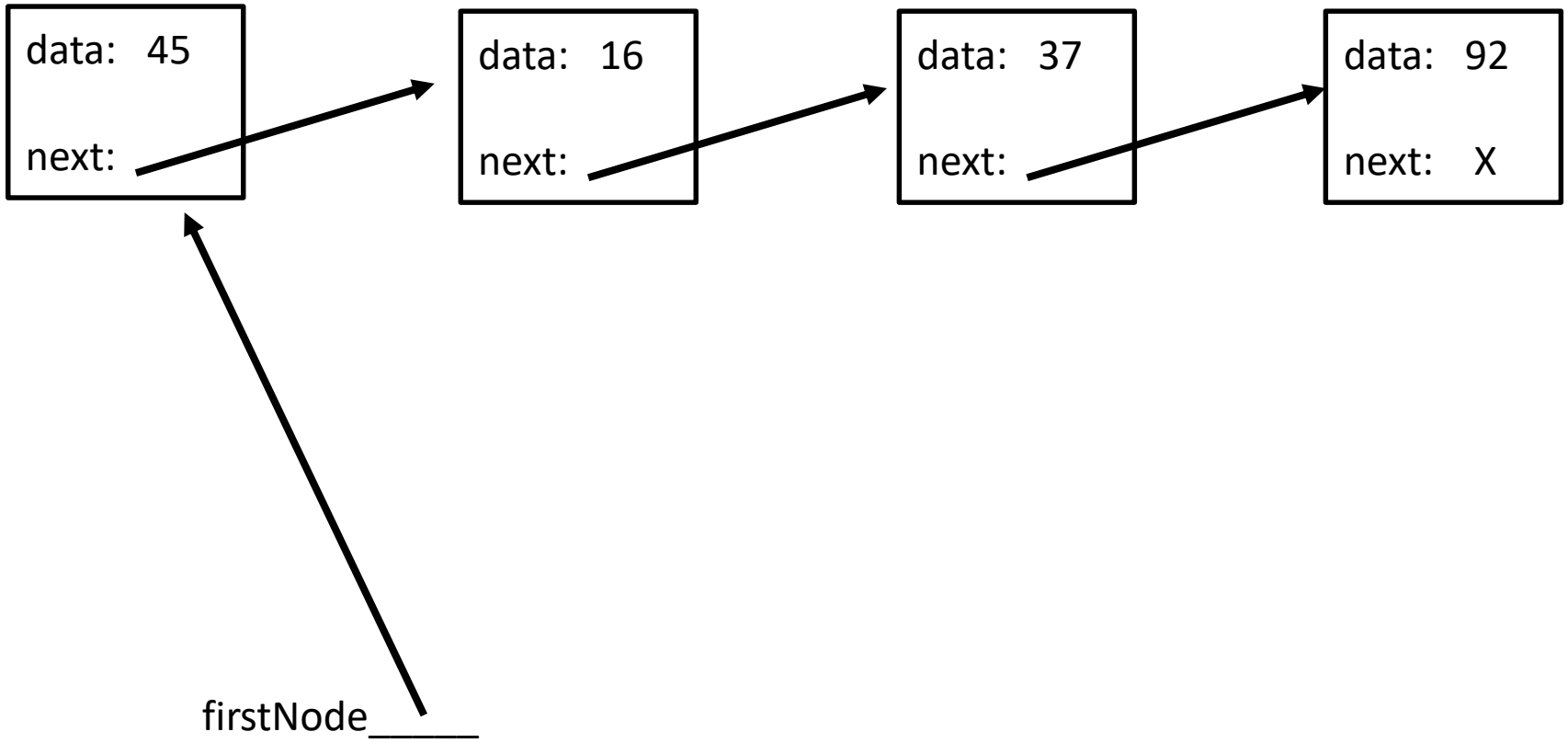
```
Node<Integer> nodeD = new Node<Integer>(45, nodeC);
// creates a new, named node with nodeC as next
```



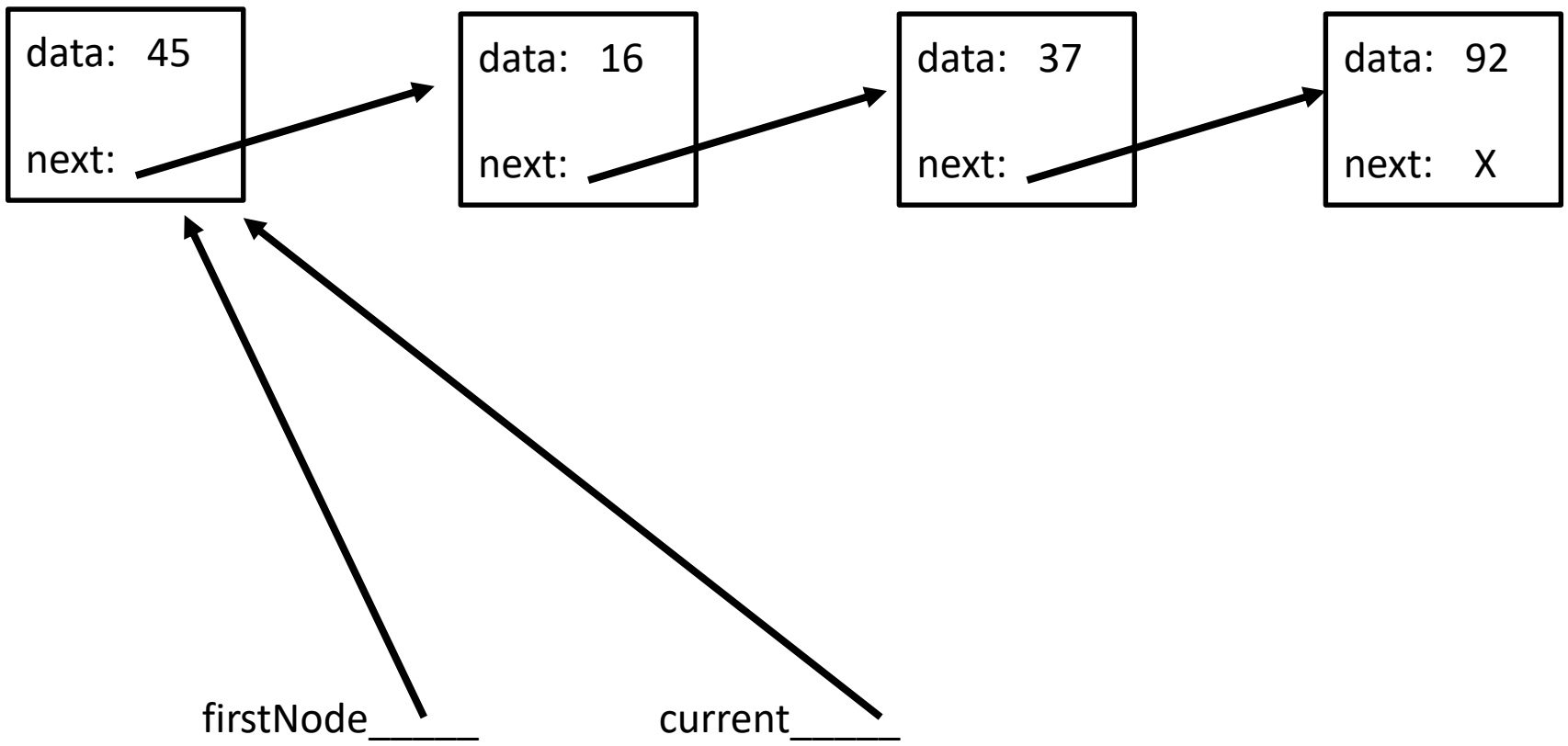
```
Node<Integer> firstNode = nodeD;
// creates an alias!
```



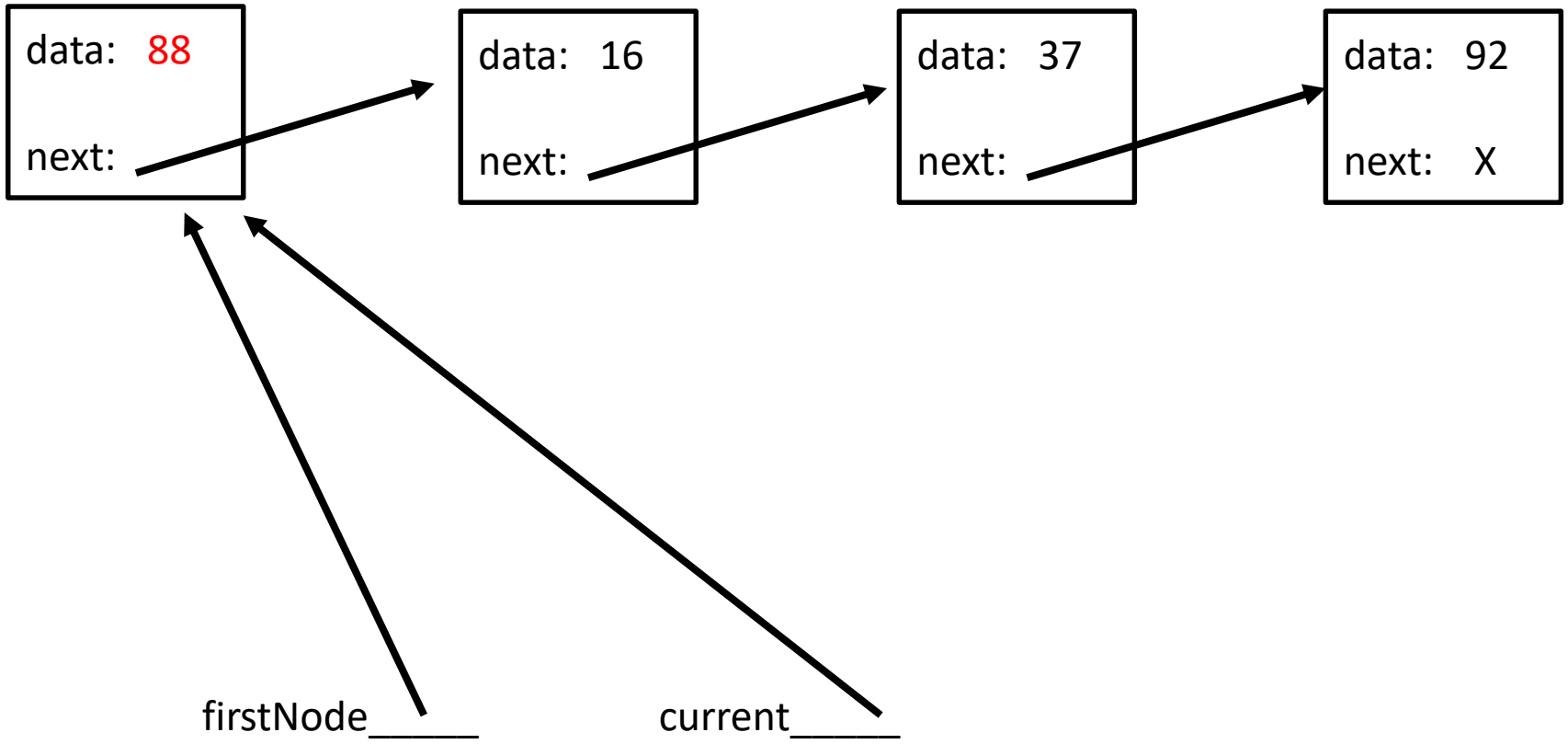
```
// removing the named node variables from our view
```



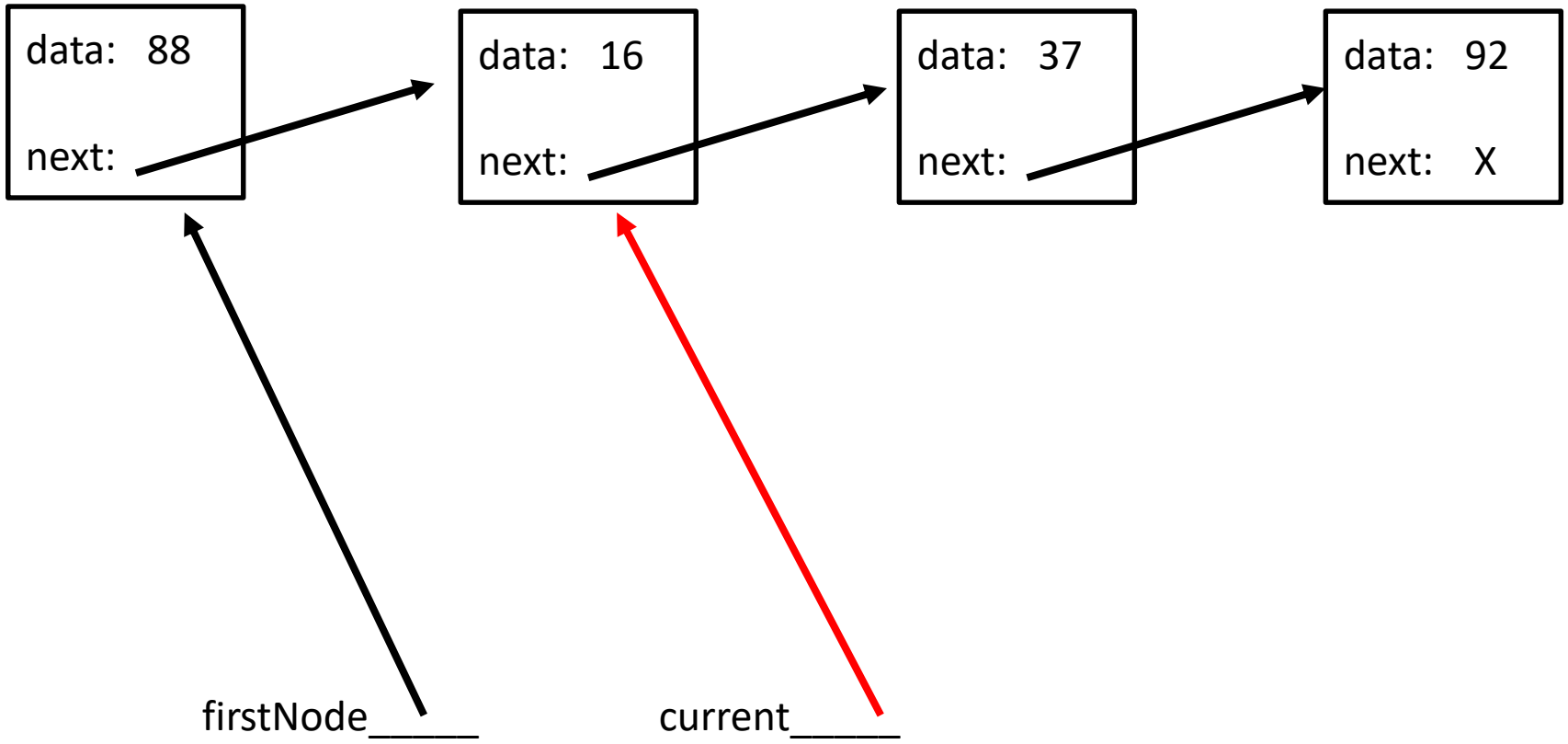
```
Node<Integer> current = firstNode;
// creates an alias
```



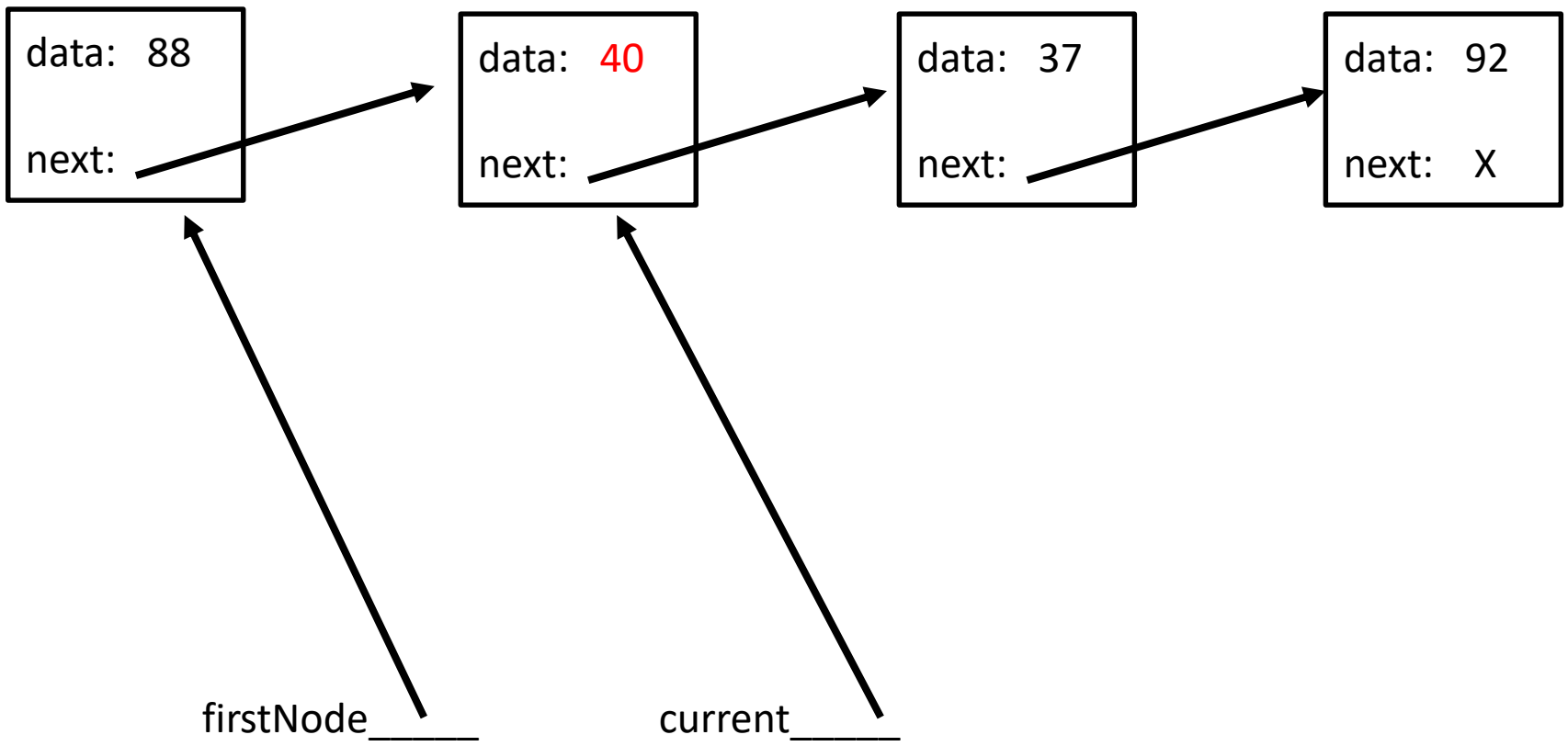
```
firstNode.data = 88;
// changes the data for firstNode; will also affect
current since they are aliases
// the structure of the chain does not change
```



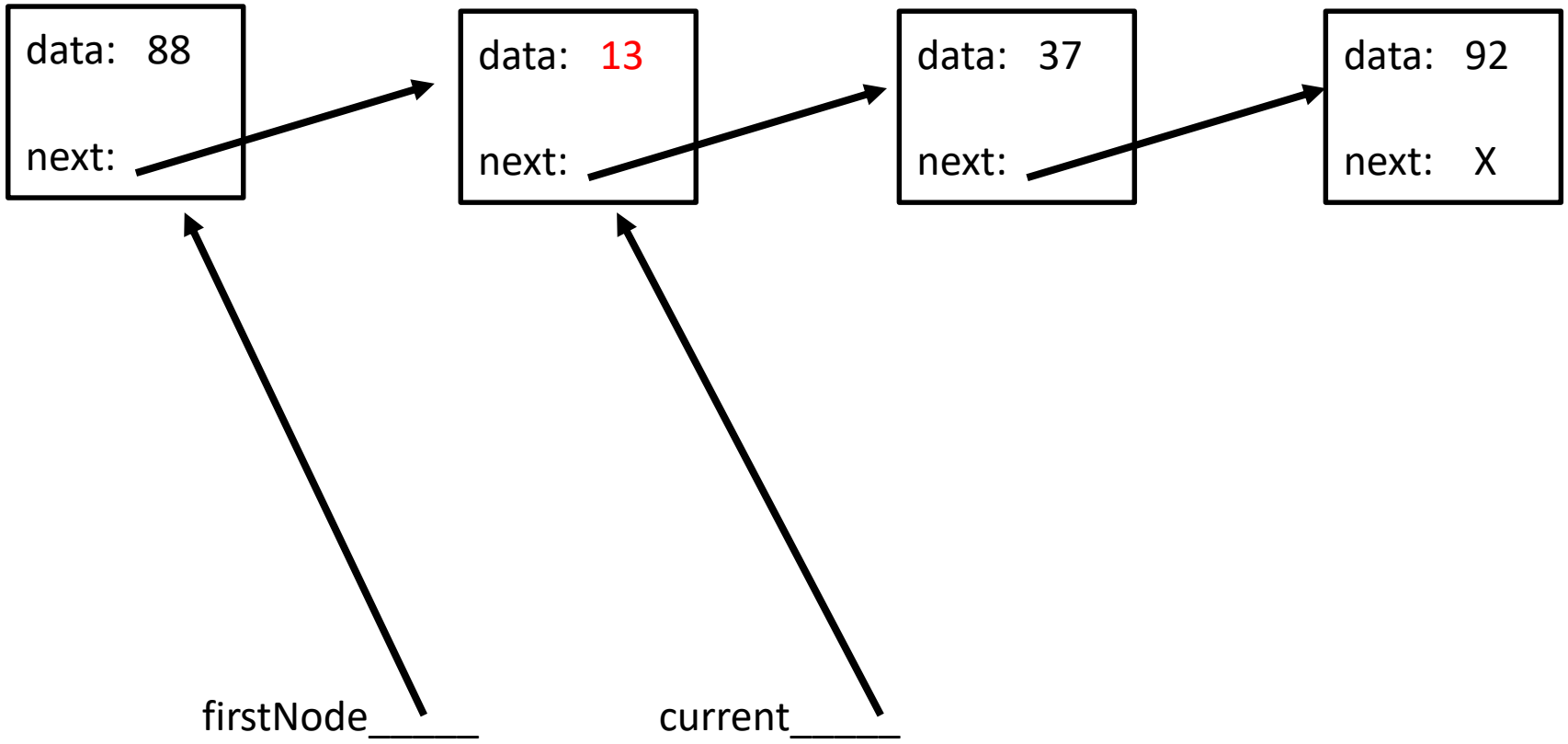
```
current = current.next;
// changes the node pointed to by current
// current and firstNode are no longer aliases
// no change to the structure of the chain
```



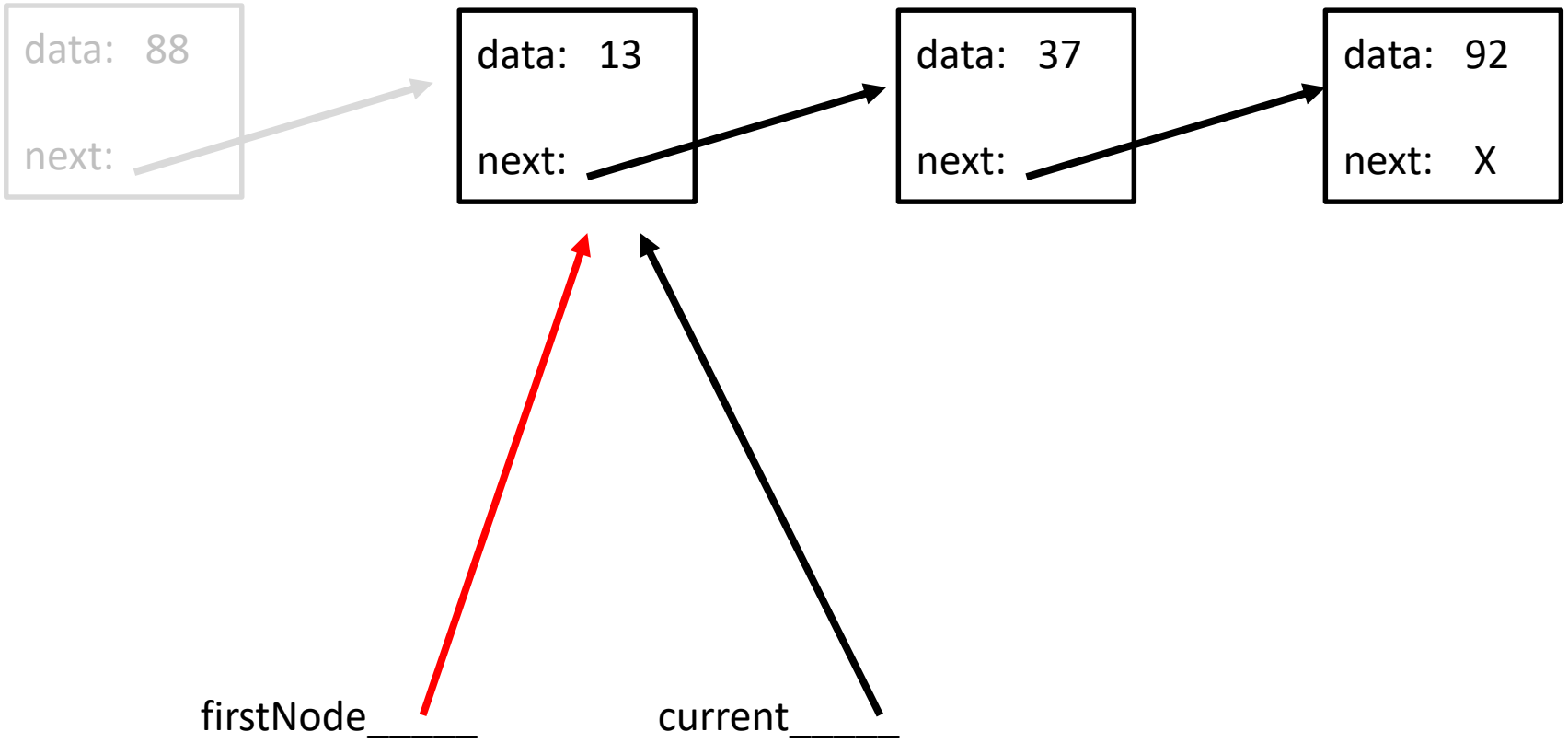
```
current.data = 40;
// changes the data in the current node
// no change to the structure of the chain
```



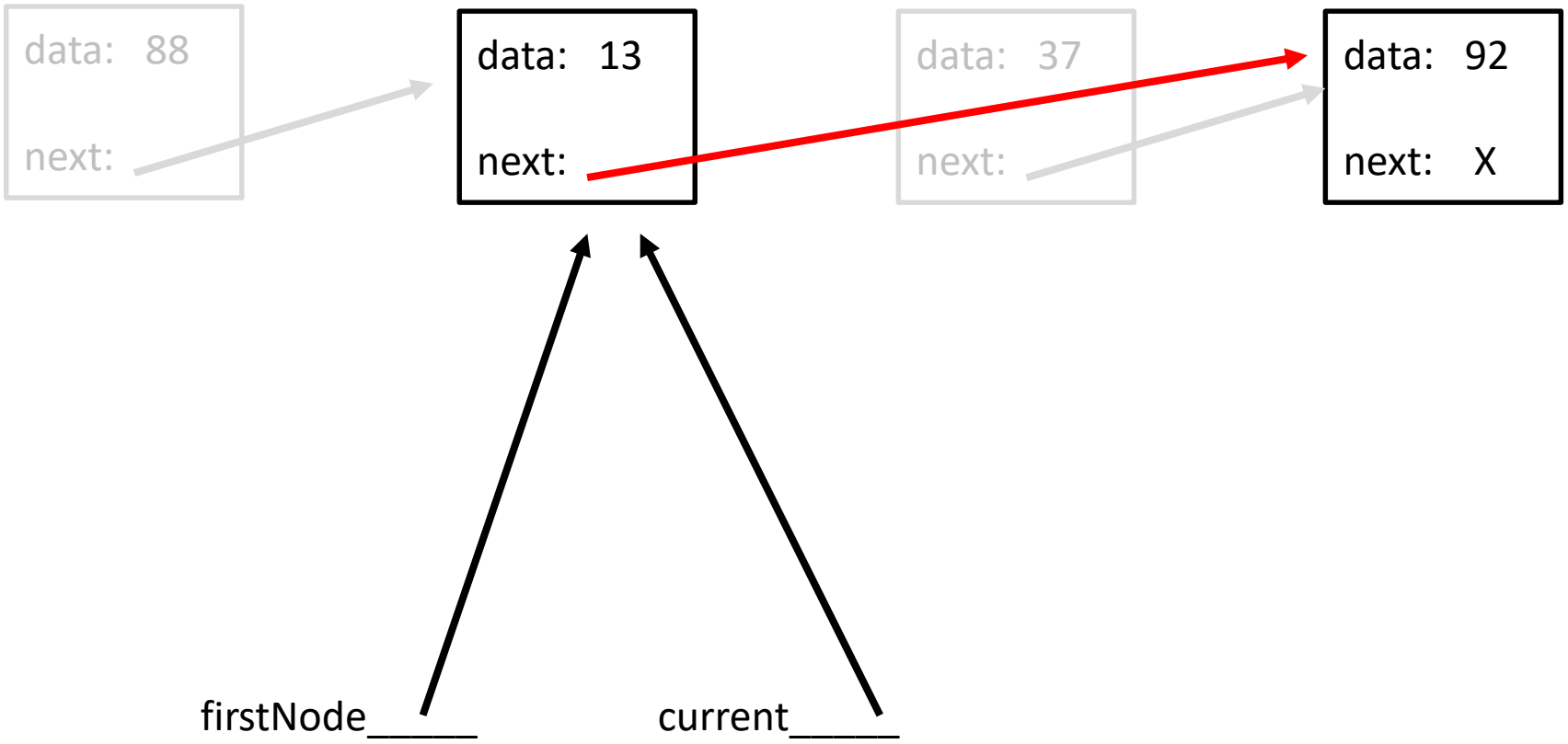
```
firstNode.next.data = 13;
// changes the data in firstNode.next
// no change to the structure of the chain
```



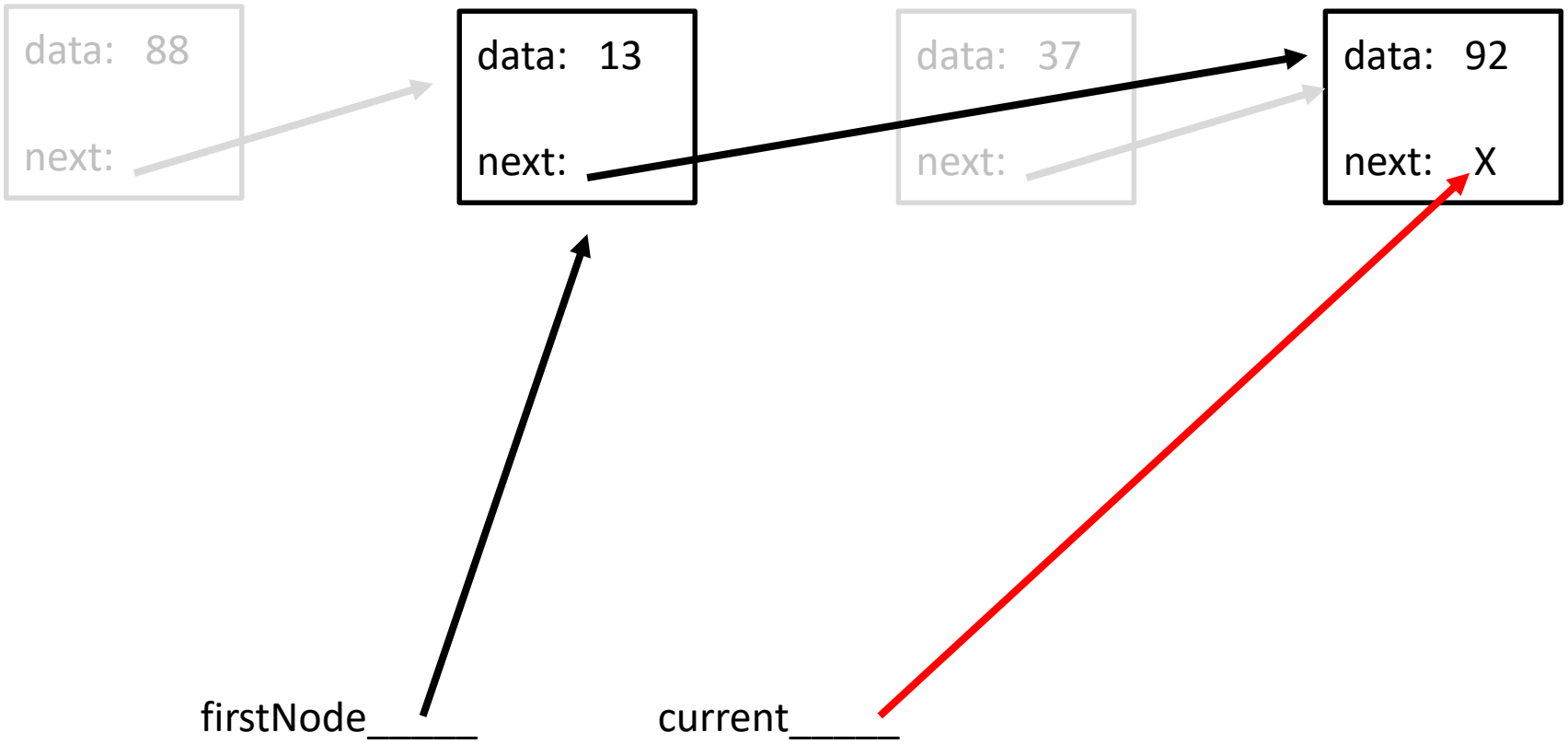
```
firstNode = firstNode.next;
// changes the node pointed to by firstNode
// changes the structure of the chain!! (assuming we
only have access to firstNode)
// firstNode and current are aliases again
```



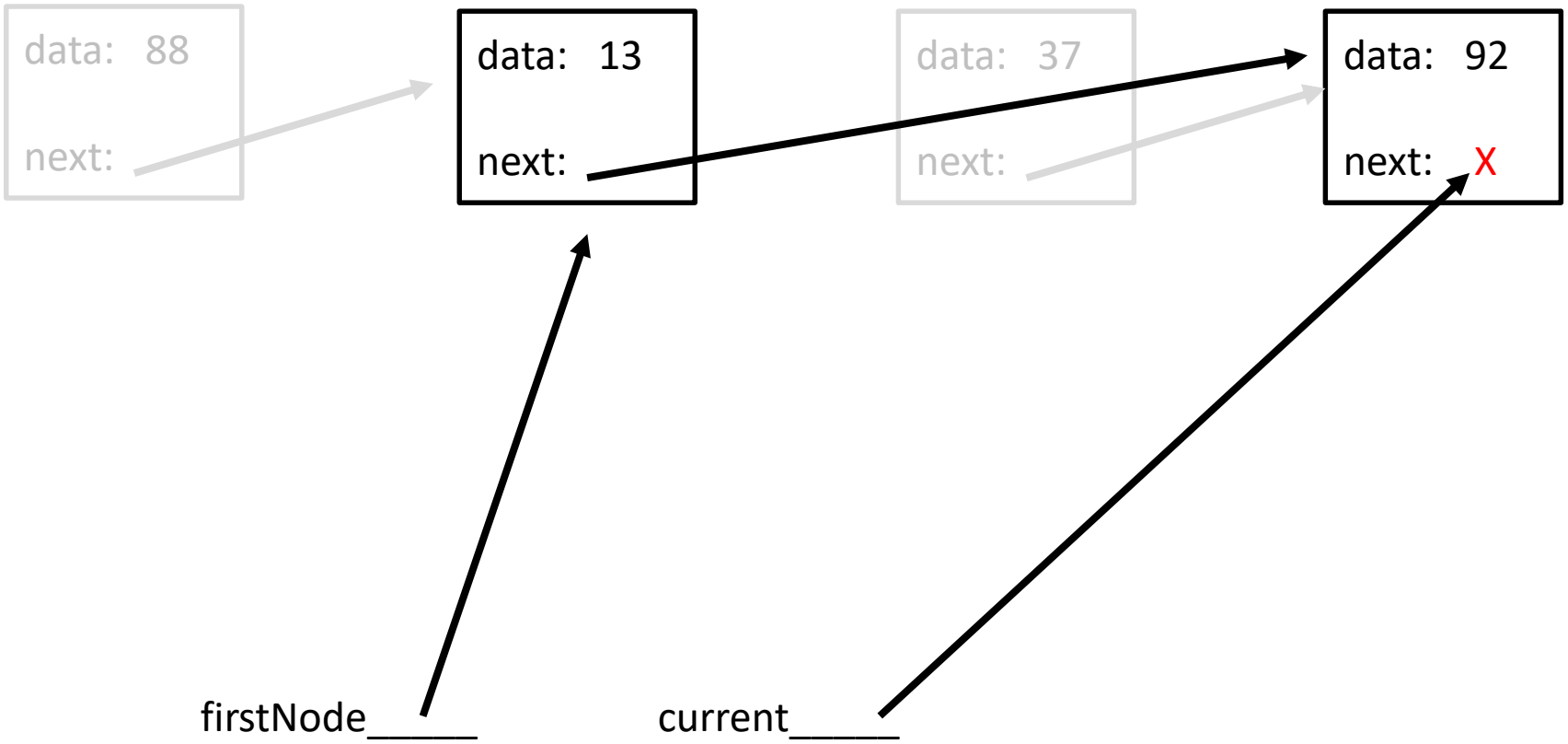
```
firstNode.next = firstNode.next.next;
// changes the next reference of first node
// changes the structure of the chain!!
```



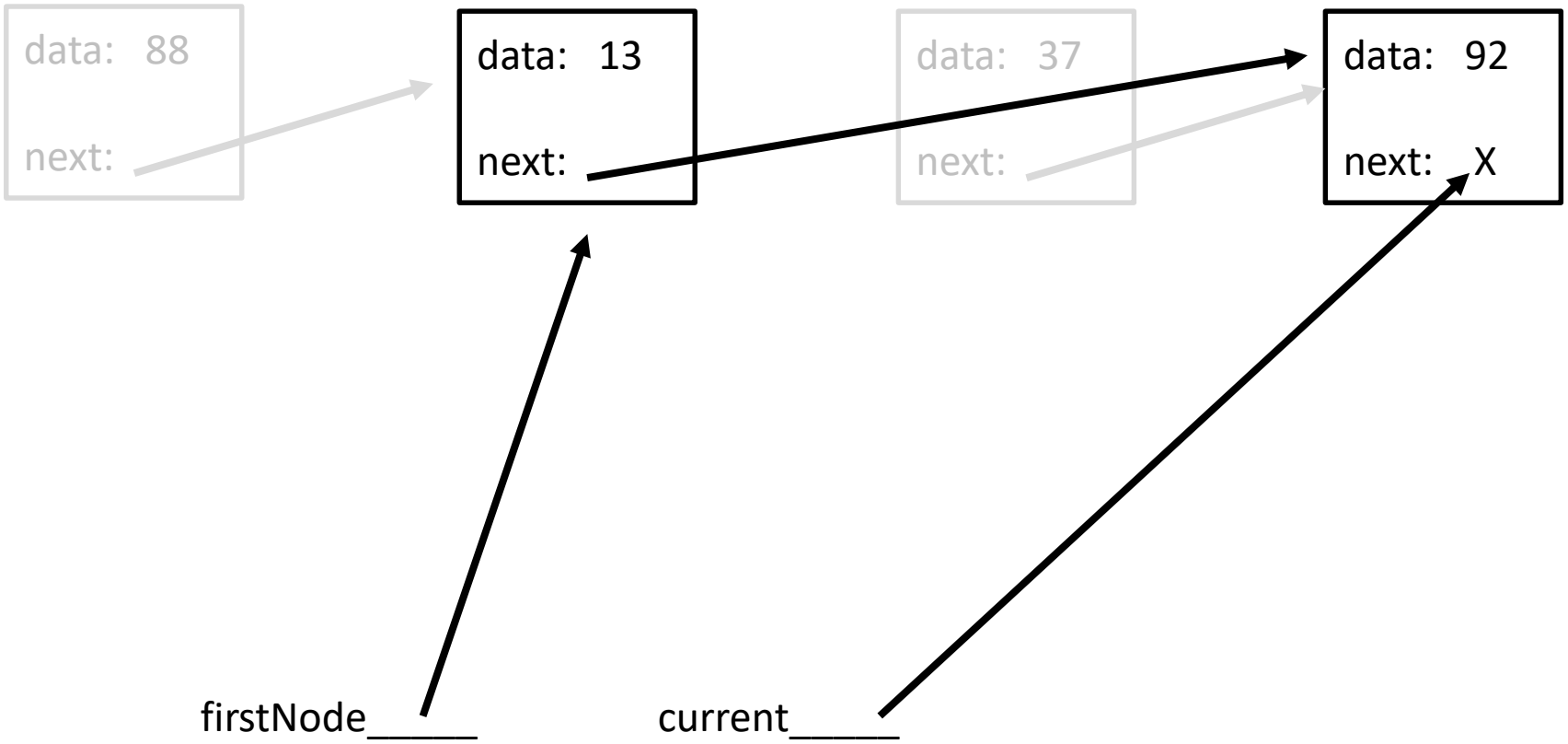
```
current = current.next.next;
// changes the node pointed to by current
// no change to the structure of the chain
```



```
System.out.println(current);
// prints null
```



```
current = current.next;
// throws an exception!
// you cannot invoke anything on null
```



Node References and Assignment: .data on the left of =

- currentNode.data = someData;
- currentNode.data.changeData();
- firstNode.data = someData;
- firstNode.data.changeData();

  - changes the data contained in the chain
  - does **not** change the structure of the chain
  - does **not** change what firstNode or currentNode points to

Node References and Assignment: node on the left of =

- currentNode = someNode;
- currentNode = someNode.next;
  - changes what currentNode points to
  - does **not** change the structure of the chain

- firstNode = someNode;
- firstNode = someNode.next;
  - changes what firstNode points to
  - since this is usually all we have, this **does** change the structure of the chain

Node References and Assignment: node.next on the left of =

- currentNode.next = someNode;
- currentNode.next = someNode.next;
- firstNode.next = someNode;
- firstNode.next = someNode.next;
  - does **not** change what currentNode or firstNode points to
  - changes the structure of the actual chain by changing which nodes are linked together

USING LINKED NODES TO IMPLEMENT COLLECTIONS

Arrays vs. Linked Nodes

- Arrays-based implementations of data structures have some disadvantages:
  - the need to shift for insertions and deletions
  - the need to handle full arrays and copy the contents over to a new array
  - wasted memory with an overly large array
- These particular disadvantages can be avoided with linked nodes.
  - Of course, there are disadvantages to linked nodes as well! (Such as no direct access!)

Using Nodes to Implement a Bag

- In any class, the instance data variables describe objects of that class.
  - private!
- For a bag, we'll use two variables:

  Node firstNode; // first object in the bag

  int numberOfEntries; // how many items
- The constructor will initialize these variables.
  - We initialize firstNode to null.
  - firstNode==null represents an empty bag!

Using Nodes to Implement a Bag

- Note that we only keep track of a single node in order to keep track of the entire chain.
- That single reference points to the first item.
  - The first item then points to the second item.
  - The second item points to the third... and so on.
- So we only keep track of a single node and, by doing so, we maintain a way to access all elements in the chain.

Adding to the Bag

- Since order doesn't matter, let's choose to add in the easiest place!
  - For an array, that was at the end (the last open space).
  - For linked nodes, the easiest place to add is at the beginning!

```
Node newNode = new Node(item);
newNode.next = firstNode;
firstNode = newNode;
```

- Check special cases for crashing: empty bag, singleton bag

Removing

- Remove an unspecified element
  - The method doesn't specify which to remove, so let's choose the easiest!
  - For linked nodes, the easiest to remove is the first in the chain.

```
T dataToRemove = firstNode.data;
firstNode = firstNode.next;
```

- Check special cases for crashing: empty bag, singleton bag
  - Empty bag: firstNode is null, so firstNode.data will crash!

Removing

- Removing a particular element
  - Check that the bag is not empty and that the bag contains the desired value
  - We could use the process for removing from the middle.
  - Or, we can swap the to-be-removed element with the first node in the chain and then delete the first node.

Other LinkedBag Methods

- The add and remove methods change the state of the bag.
- Other methods provide information about the bag, but do not alter its state.
- These methods rely on traversing the chain of nodes, using a temporary Node reference to do so:

```
Node currentNode = firstNode;
while(currentNode!=null) {
   // process/check whatever you need to check
   // about currentNode.data

   currentNode = currentNode.next;
}
```

Using Linked Nodes to Implement a List

- For a list, we'll use two variables:

  Node firstNode; // first object in the list

  int numberOfEntries; // how many items
- The constructor will initialize these variables.
  - We initialize firstNode to null.
  - firstNode==null represents an empty list!
- Remember that lists are ordered!

Adding to the End

- One add method adds to the end of the list.
- First, we need to find the last element in the chain.
- We can then add another node onto the end of the chain.
  - This requires us to find the lastNode. We can use the private helper method getNodeAt to do this by finding the node at position numberOfEntries.

```
Node lastNode = getNodeAt(numberOfEntries);
lastNode.next = newNode;
```

- Check special cases for crashing: empty list, singleton list
  - Empty bag: lastNode will be null, so the code will crash!

Adding to the Middle or End

- The other add method allows the user to specify the position.
  - It could be the end (which would use what we just wrote), or it could be the beginning or middle.
- We can use the general insertion process for both.
- We need to find the nodeBefore to do the insertion.
  - We can use the private helper method getNodeAt to do this by finding the node at position numberOfEntries-1.

Removing

- The remove method removes based on a specified position.
  - The position could be in the beginning, middle, or end.
- We can use getNodeAt(givenPosition-1) to find the nodeBefore.

Other Methods

- The replace method allows you to replace *data*
  - We don't need to change the structure of the nodes at all!

```
Node toBeReplacedNode = getEntry(givenPosition);
T oldData = toBeReplacedNode.data;
toBeReplacedNode.data = newData;
```

- Just like with LinkedBag, some of the other methods provided in LList require the traversal of the chain. You traverse a list in the same way as you traversed a bag.

Adding a Tail Reference

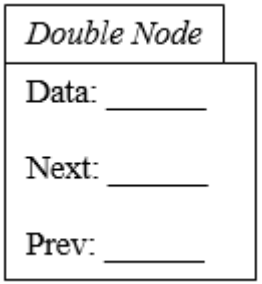
- Adding to the end of a list is not very efficient.
  - To add to the end of the chain, each time we must find the last element and then we can update lastNode.next.
- We can improve on this efficiency by adding a reference to the end of the list: the tail (lastNode).
  - Adding to the end of the list goes from O(n) to O(1)!
    - A big improvement!
- If using a tail reference, you must be careful to update firstNode **and** lastNode in all add and remove methods.
- What about deleting from the end of the list? Will this action be affected? Trace it out!

Arrays vs. Linked Nodes

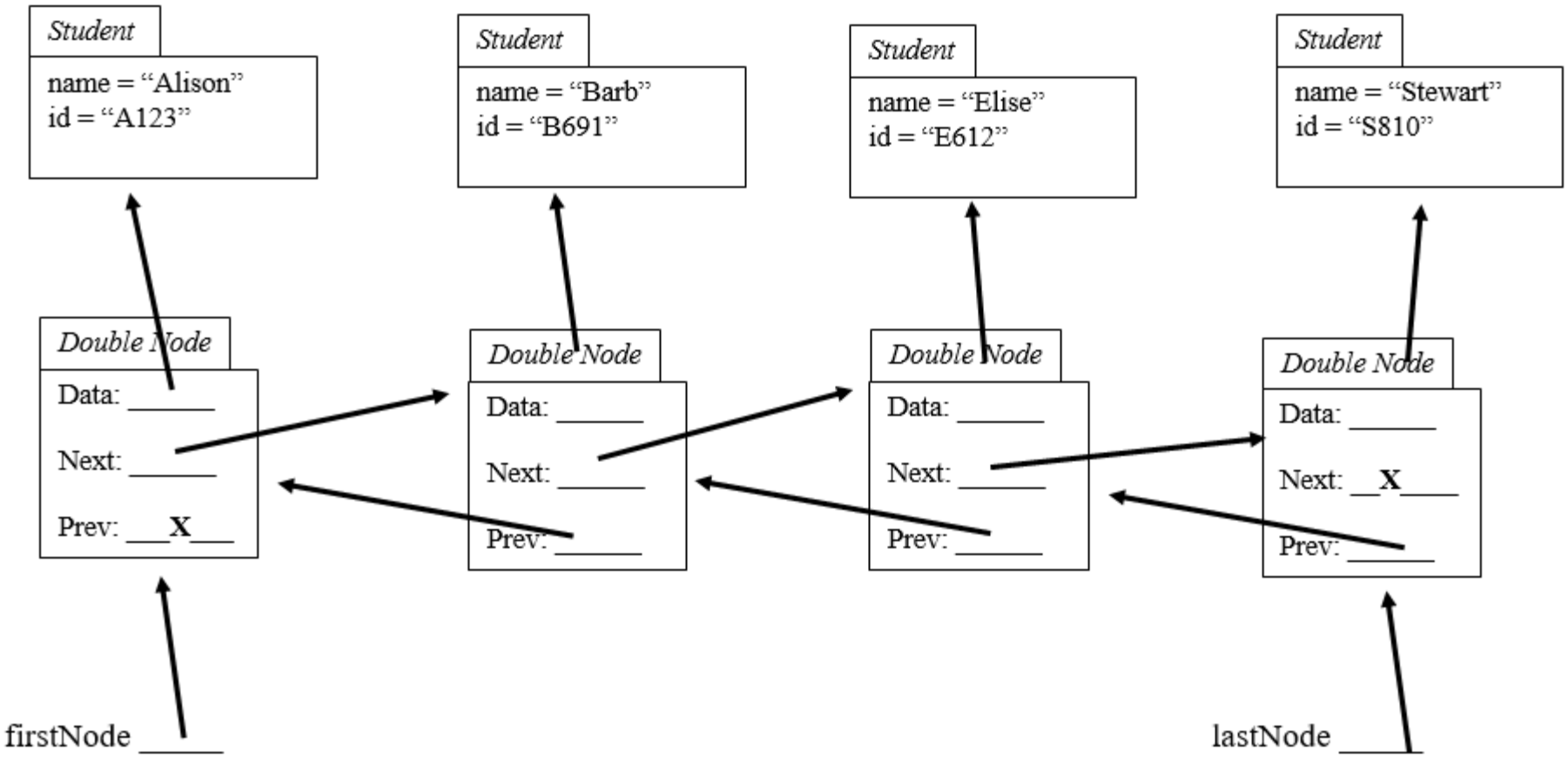
- Size of Collection
  - Arrays: fixed in size; requires copying to expand
  - Linked Nodes: can grow and shrink as needed, as long as memory is available
- Memory:
  - Linked nodes do require more memory (there are more objects)
  - Arrays often have wasted space
- Adding/Removing
  - From the beginning: advantage nodes!
  - From the middle: equivalent
  - From the end: advantage array!

List and LinkedList

- The Java standard library provides a class that implements the List interface using linked nodes behind the scenes- LinkedList
- This class actually uses a *doubly-linked list* with a head **and** tail pointer



LinkedList Class- Doubly Linked with Head and Tail Pointers



Using LinkedList

- Most of the time, use ArrayList
  - Constant-time positional access
  - “Just plain fast!”
- *Consider* LinkedList if:
  - Frequently adding elements to the beginning
  - Frequently iterate the list to delete elements from the beginning
  - Even in these cases, you should test performance!
- Source: Oracle Tutorial on List Implementations

ArrayList vs LinkedList

Action	ArrayList	LinkedList
Direct access- get(i)	O(1)	O(n)
Adding/removing beginning	O(n)	O(1)
Adding/removing middle	O(n)	O(n) O(1) with iterator
Adding/removing end	O(1)	O(1)

- Note that these are the efficiencies for ArrayList and LinkedList- **not** for AList and LList!