# Data Structures
# COMP242

Ala' Hasheesh
ahashesh@birzeit.edu

# Lists

# Lists

- A list is a finite, ordered sequence of data items known as elements ("ordered" means that each element has a position in the list)

We say that $a_{i+1}$ follows $a_i$

# Lists

- A List as an Abstract Data Type (ADT)

list is a sequence of items that supports at least the following functionality:

- accessing an item at an arbitrary position in the sequence
- adding an item at an arbitrary position
- removing an item at an arbitrary position
- determining the number of items in the list (the list's length)

# Lists

- Abstract Data Type (ADT)

specifies what a list will do, without specifying the implementation.

**Operation**

- Print List
- Find element
- Add element
- Insert element
- Delete element
- Make null

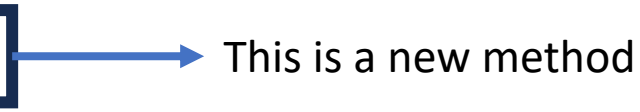
# Lists

```
public interface Listable<T> {
   void add(T element);

   void insert(int index, T element);

   T delete(int index);

   T get(int index);

   int find(T element);

   void clear();

   void print();
}
```

**We did this in lab0**

# Lists

```
public interface Listable<T> {
   void add(T element);

   void insert(int index, T element);

   T delete(int index);

   int find(T element);

   void clear();

   void print();
}
```



# Lists

```
public interface Listable<T> {
   void add(T element);

   void insert(int index, T element);

   T delete(int index);

   int find(T element);

   void clear();

   void print();
}
```

$$insert(2, 100)$$

# Lists

```
public interface Listable<T> {
   void add(T element);

   void insert(int index, T element);

   T delete(int index);

   int find(T element);

   void clear();

   void print();
}
```

$$insert(2, 100)$$


$$Shift\ elements\ to\ the\ right$$

# Lists

```
public interface Listable<T> {
   void add(T element);

   void insert(int index, T element);

   T delete(int index);

   int find(T element);

   void clear();

   void print();
}
```

$$insert(2, 100)$$

| 5 | 10 | 20 | 20 | 13 | 1 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

$$Shift\ elements\ to\ the\ right$$

# Lists

```
public interface Listable<T> {
   void add(T element);

   void insert(int index, T element);

   T delete(int index);

   int find(T element);

   void clear();

   void print();
}
```

$$insert(2, 100)$$

| 5 | 10 | 100 | 20 | 13 | 1 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |

$$a[2] = 100$$

# Lists

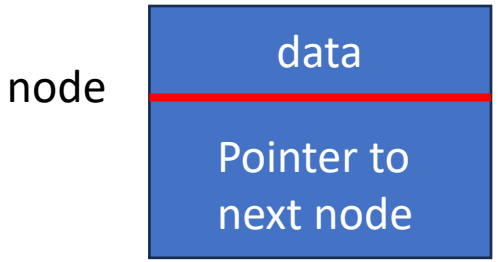
```
public interface Listable<T> {
    void add(T element);

    void insert(int index, T element);

    T delete(int index);

    int find(T element);

    void clear();

    void print();
}
```

```
public void insert(int index, T element) {
    if (count >= data.length) {
        reSize();
    }

    if (index >= count) {
        index = count - 1;
    }

    if (index < 0) {
        index = 0;
    }

    for (int i = this.count; i > index; i--) { // shift
        data[i] = data[i - 1];
    }

    data[index] = element;
    this.count++;
}
```

# Linked List

A linked list is a collection of nodes that together form a linear.

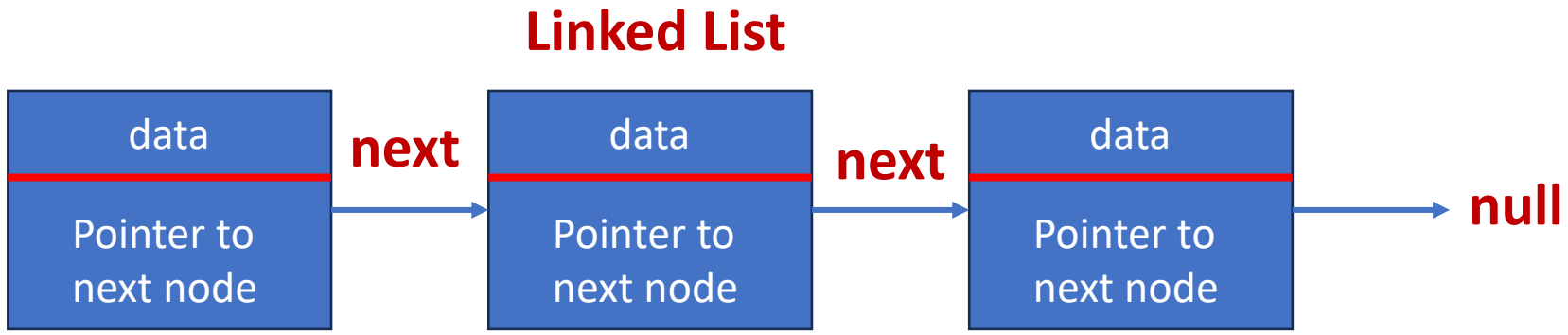
**node**: A compound object that stores the contents of the node (Element) and a pointer or reference to the next node in the list (next).



# Linked List

A linked list is a collection of nodes that together form a linear.

**node**: A compound object that stores the contents of the node (Element) and a pointer or reference to the next node in the list (next).



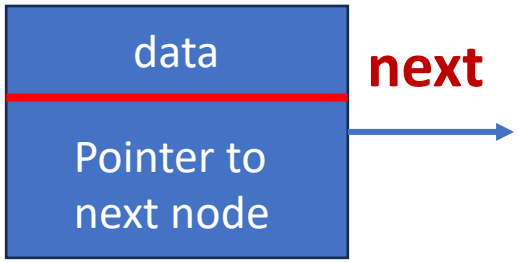
# Linked List

A linked list is a collection of nodes that together form a linear.

**node**: A compound object that stores the contents of the node (Element) and a pointer or reference to the next node in the list (next).

# Linked List

```
public class Node<T> {
    public T val;
    public Node<T> next;

    public Node(T val) {
        this(val, null);
    }

    public Node(T val, Node<T> next) {
        this.val = val;
        this.next = next;
    }
}
```



# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

```
public class Node<T> {
    public T val;
    public Node<T> next;

    public Node(T val) {
        this(val, null);
    }

    public Node(T val, Node<T> next) {
        this.val = val;
        this.next = next;
    }
}
```

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```


**How do we reach third element?**

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```


```
Node<Integer> n1 = new Node<>(5);
Node<Integer> n2 = new Node<>(10);
Node<Integer> n3 = new Node<>(1);

n1.next = n2;
n2.next = n3;

Node<Integer> head = n1;
Node<Integer> thirdElement = head.next.next;
```

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

```
public class Node<T> {
    public T val;
    public Node<T> next;

    public Node(T val) {
        this(val, null);
    }

    public Node(T val, Node<T> next) {
        this.val = val;
        this.next = next;
    }
}
```


**How do we reach 100th element?**

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

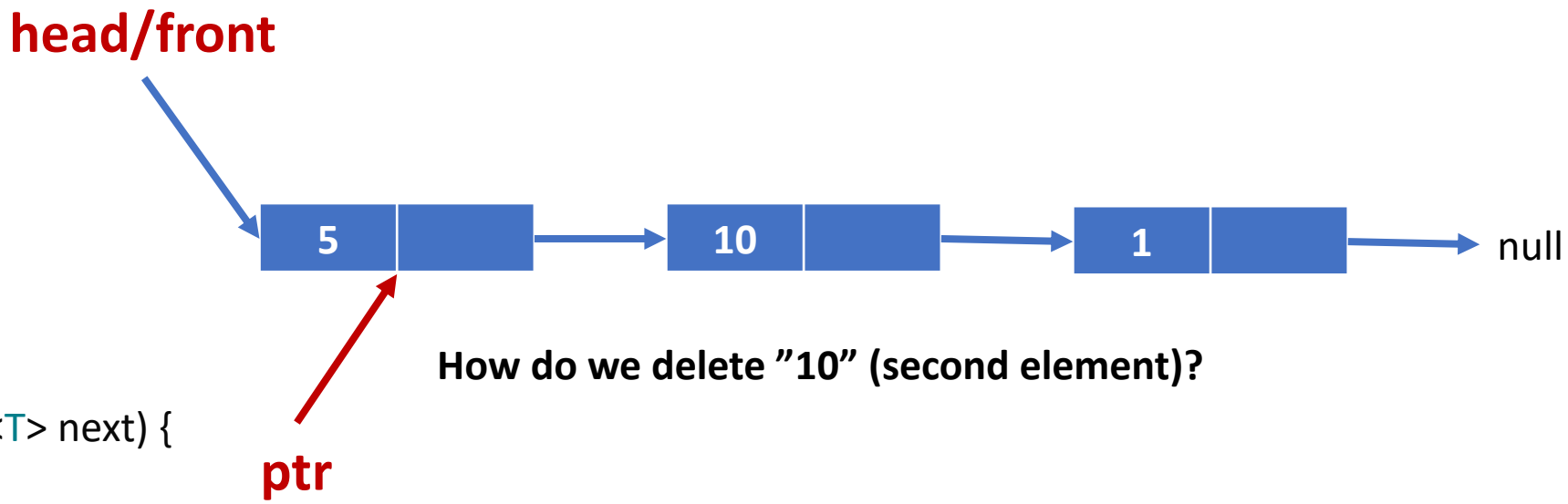
```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```


**How do we delete ”10” (second element)?**

# Linked List

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```

**In a Linked List we only need a reference to the first element (head/front)**

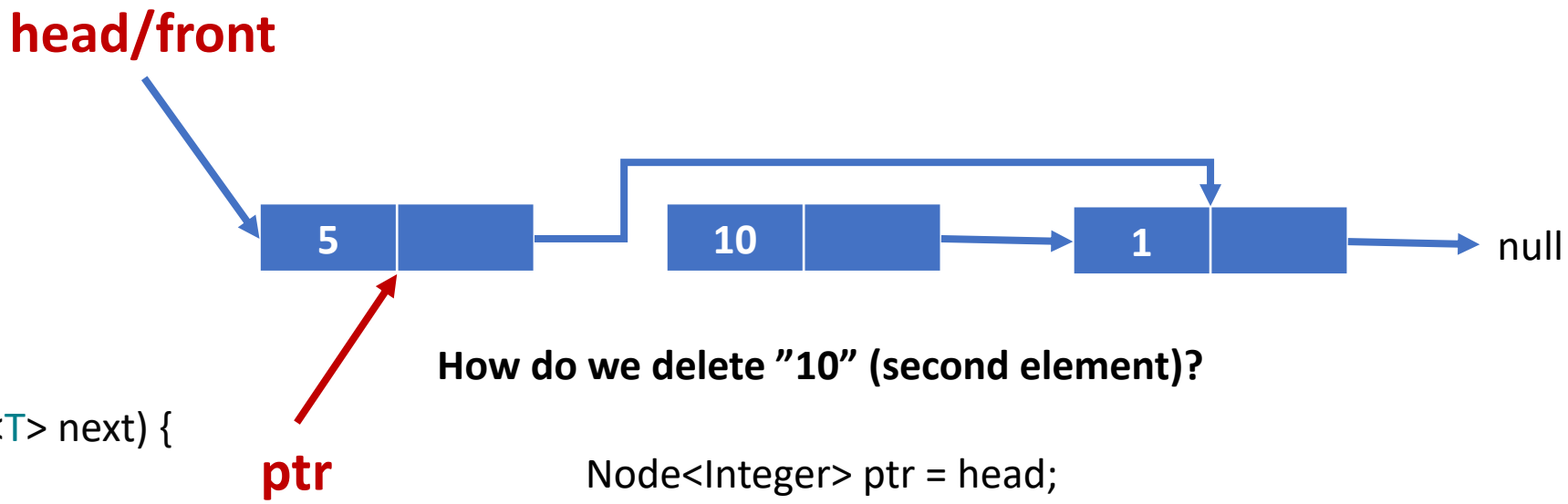



**How do we delete ”10” (second element)?**

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```



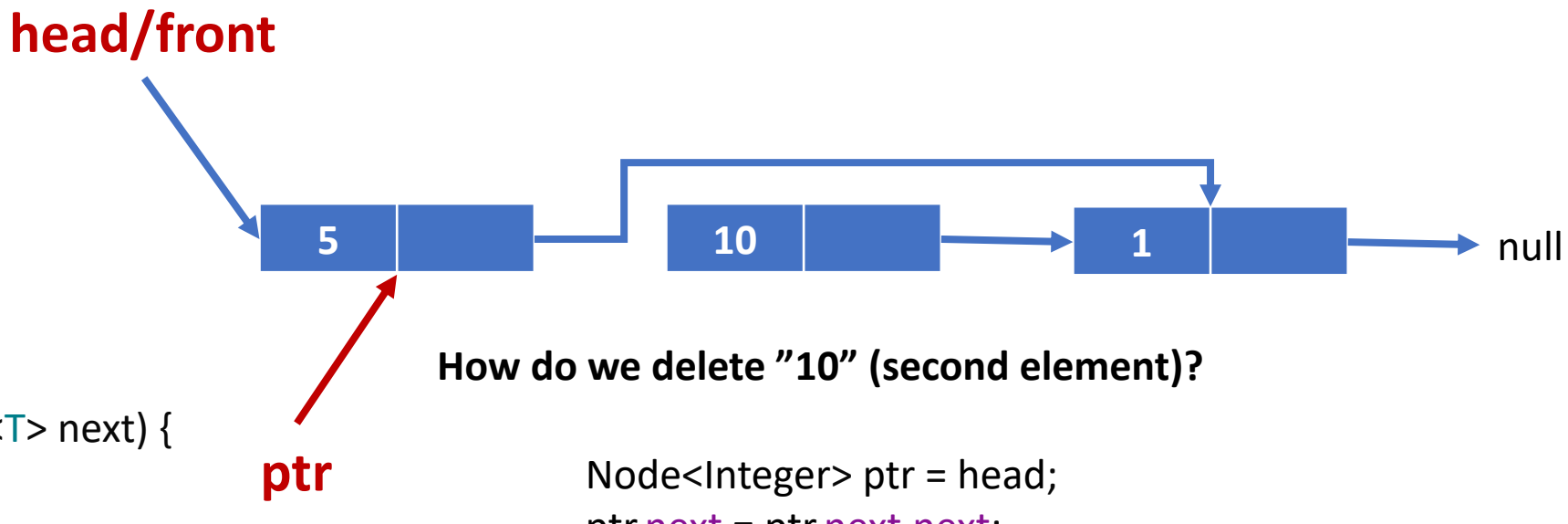

**How do we delete ”10” (second element)?**

```
Node<Integer> ptr = head;
ptr.next = ptr.next.next;
```

# Linked List

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```

**In a Linked List we only need a reference to the first element (head/front)**




**How do we delete ”10” (second element)?**

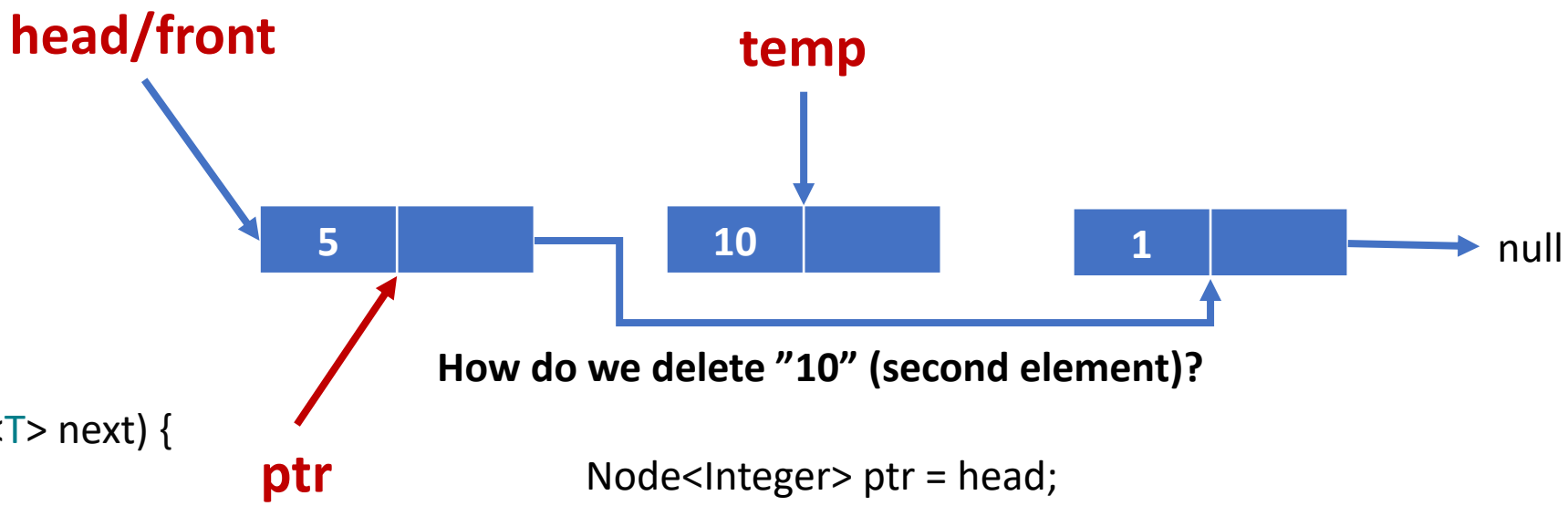
```
Node<Integer> ptr = head;
ptr.next = ptr.next.next;
```

**We didn’t need to shift anything!**

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```




**How do we delete ”10” (second element)?**

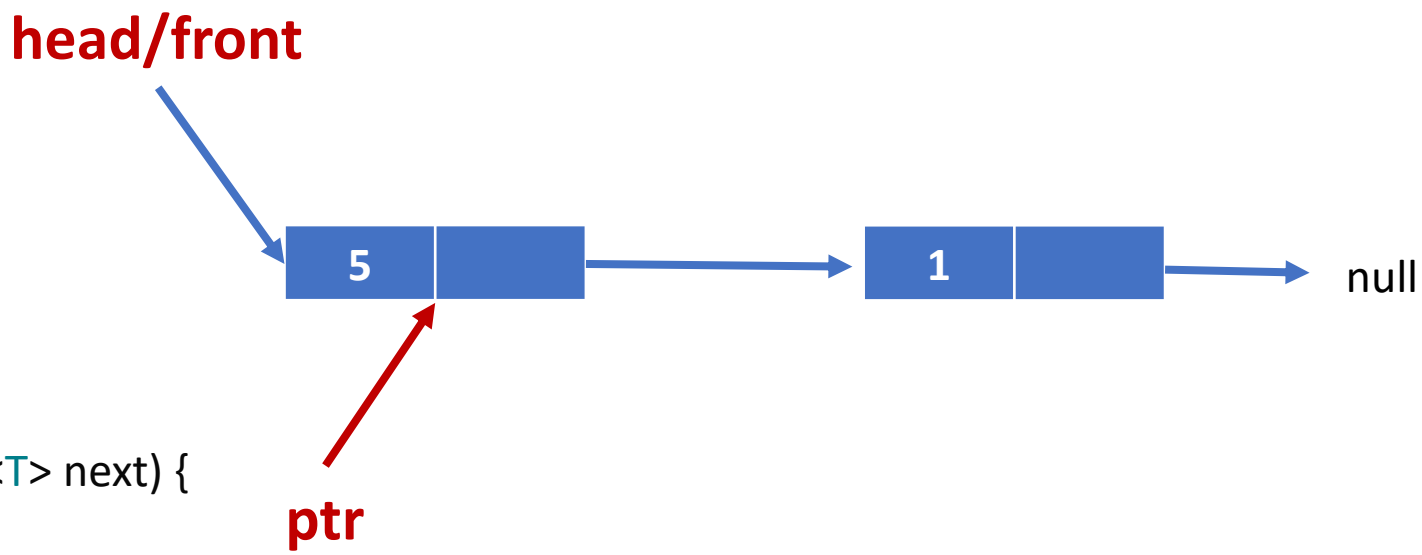
```
Node<Integer> ptr = head;
Node<Integer> temp = ptr.next;
ptr.next = temp.next;
temp.next = null;
```

**We didn’t need to shift anything!**

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

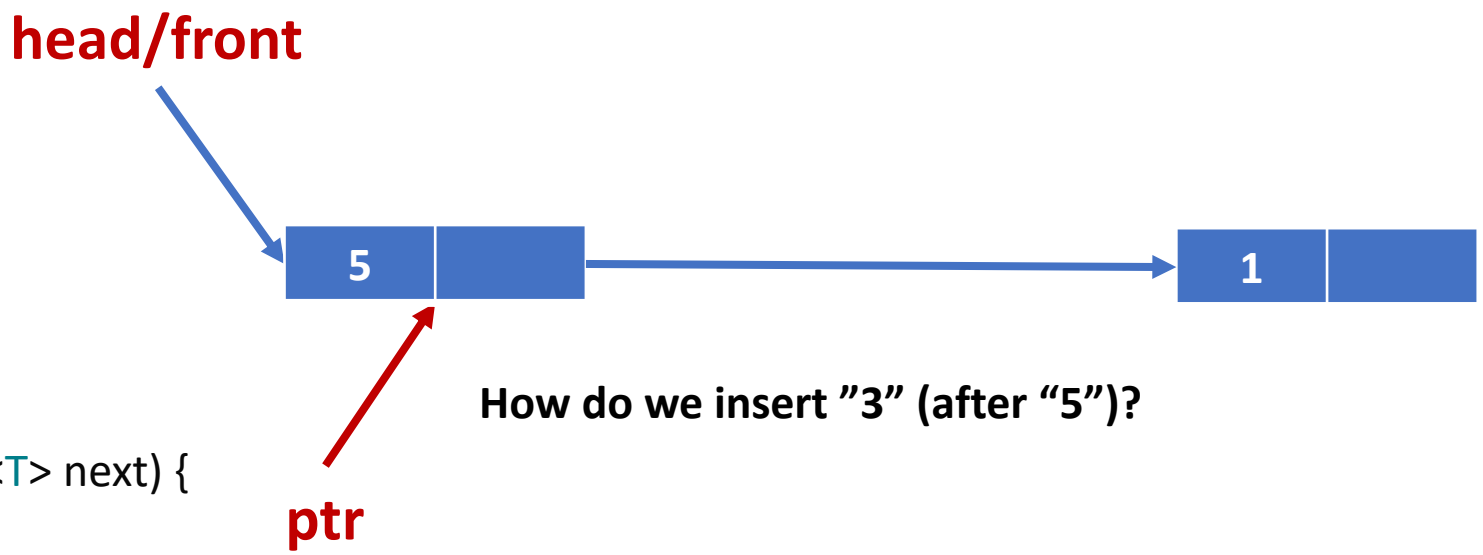
```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```



# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

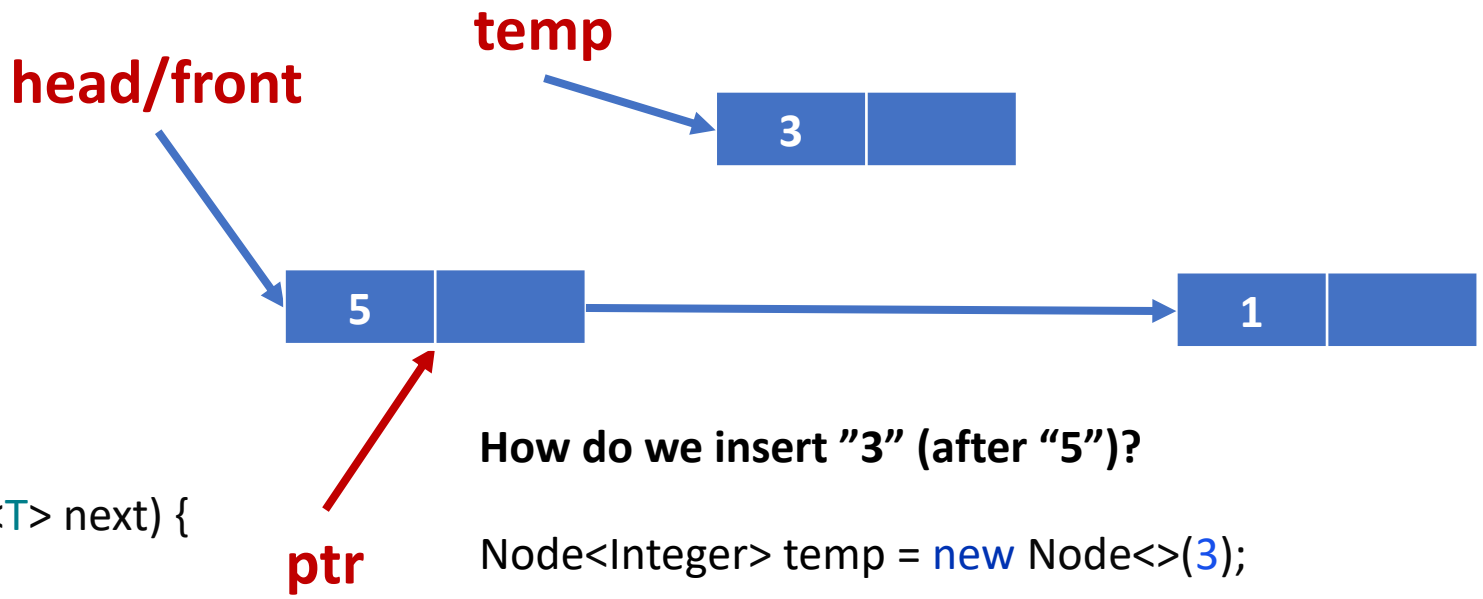
```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```



# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```



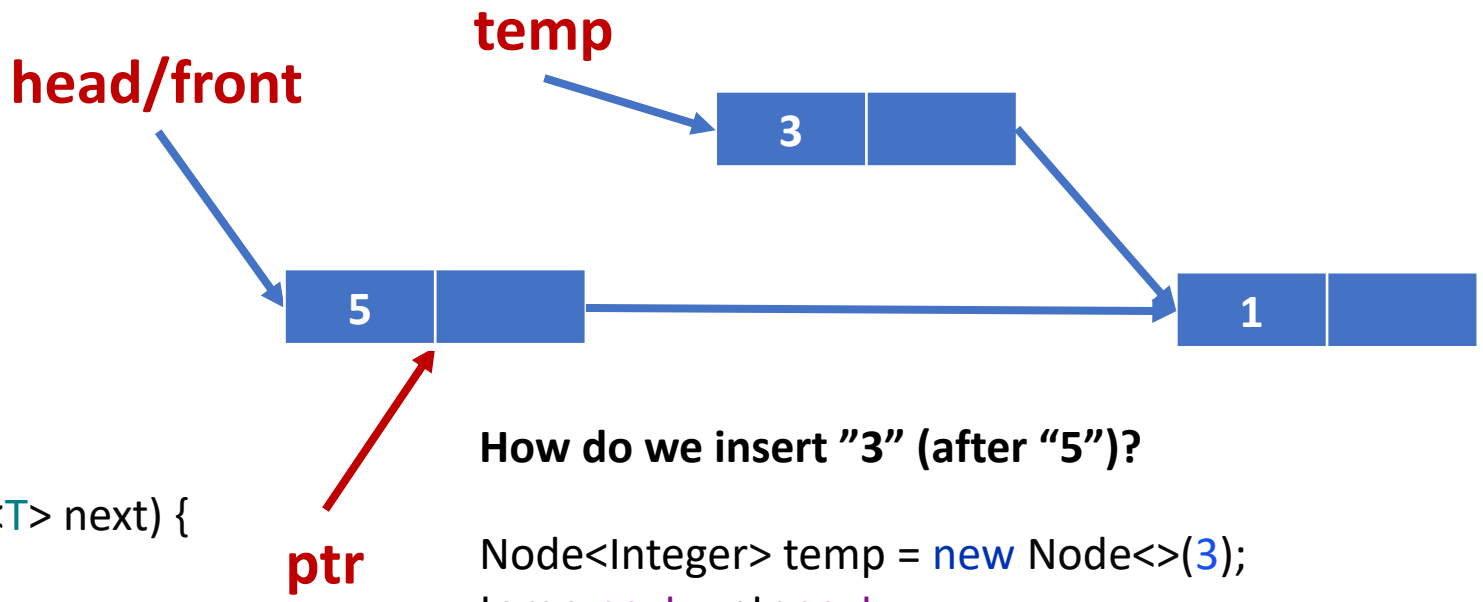

**How do we insert ”3” (after “5”)?**

```
Node<Integer> temp = new Node<>(3);
```

# Linked List

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```

**In a Linked List we only need a reference to the first element (head/front)**



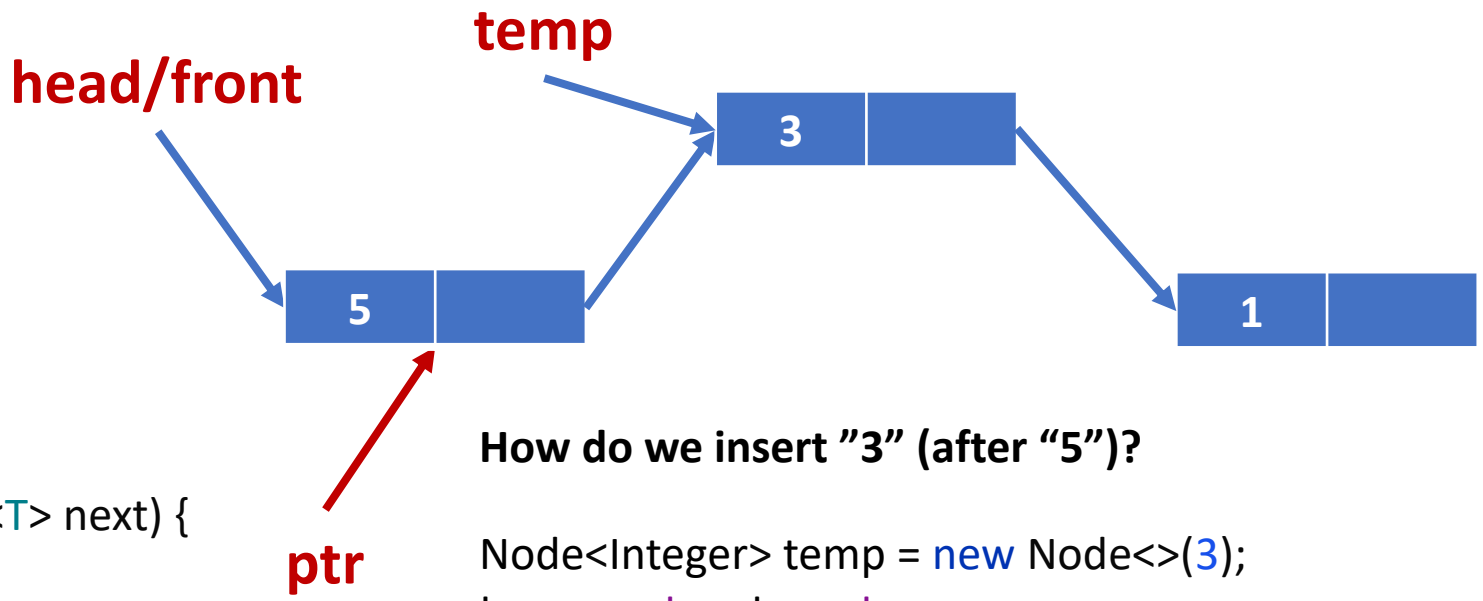

**How do we insert ”3” (after “5”)?**

```
Node<Integer> temp = new Node<>(3);
temp.next = ptr.next;
```

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```




**How do we insert ”3” (after “5”)?**

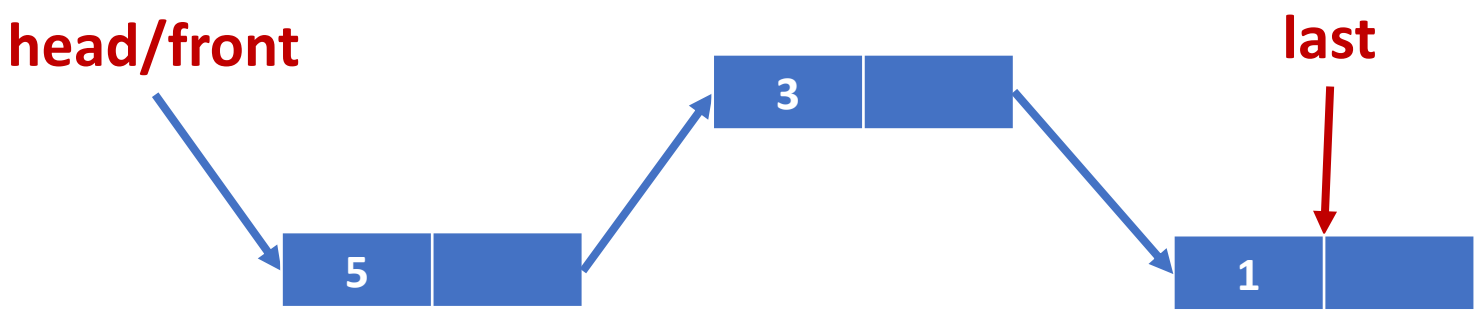
```
Node<Integer> temp = new Node<>(3);
temp.next = ptr.next;
ptr.next = temp;
```

**We didn’t need to shift anything!**

# Linked List

```
public class Node<T> {
   public T val;
   public Node<T> next;

   public Node(T val) {
      this(val, null);
   }

   public Node(T val, Node<T> next) {
      this.val = val;
      this.next = next;
   }
}
```

**In a Linked List we only need a reference to the first element (head/front)**

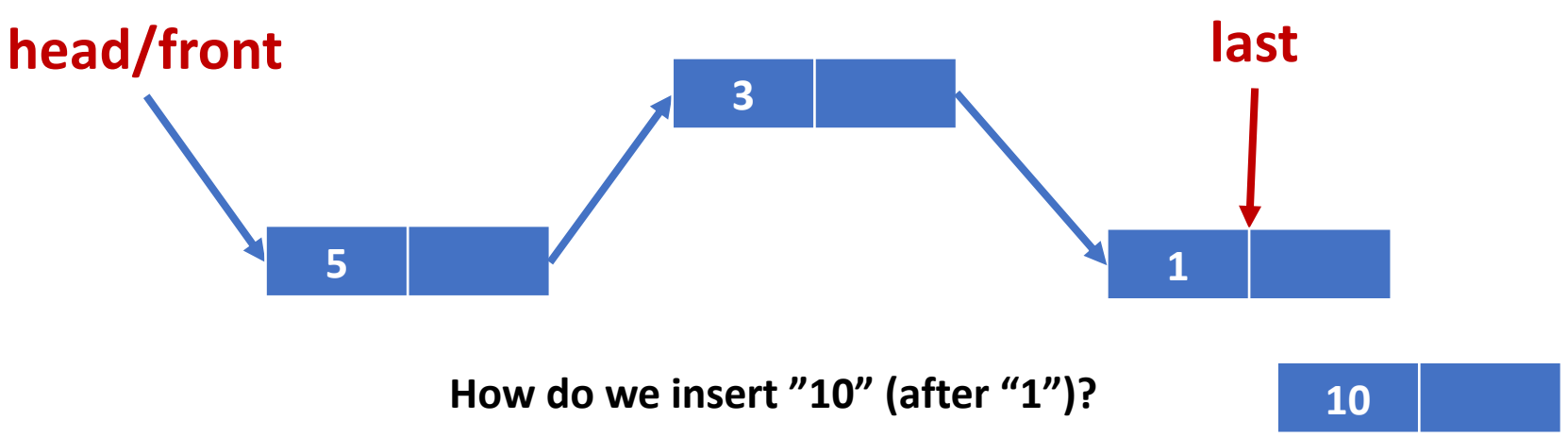



**How do we insert ”10” (after “1”)?**

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

```
public class Node<T> {
    public T val;
    public Node<T> next;

    public Node(T val) {
        this(val, null);
    }

    public Node(T val, Node<T> next) {
        this.val = val;
        this.next = next;
    }
}
```



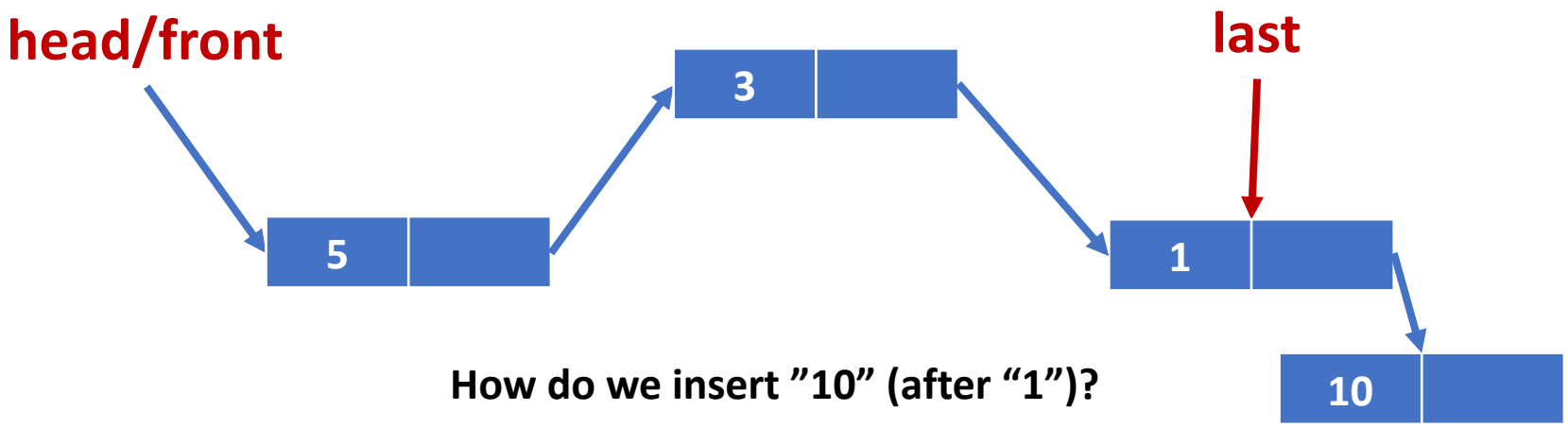

**How do we insert ”10” (after “1”)?**

```
Node<Integer> temp = new Node<>(10);
```

# Linked List

**In a Linked List we only need a reference to the first element (head/front)**

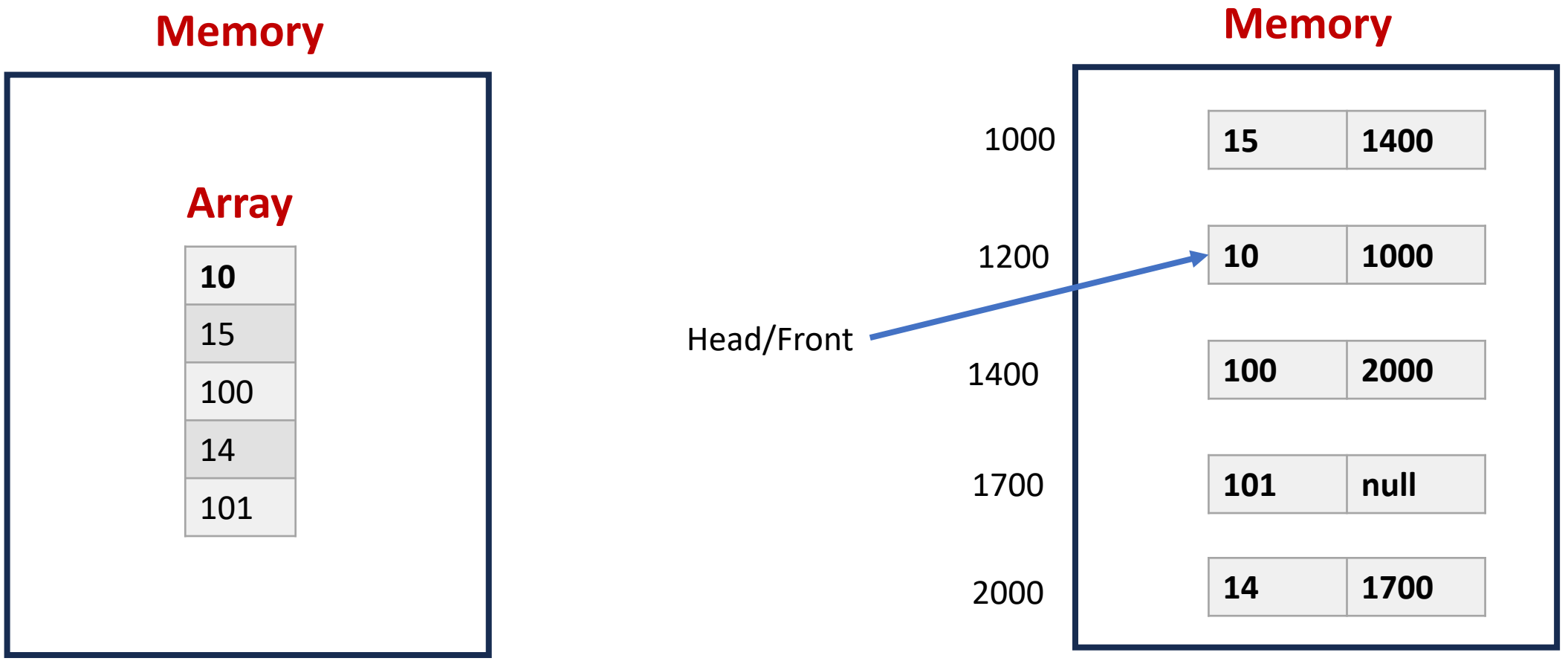
```
public class Node<T> {
    public T val;
    public Node<T> next;

    public Node(T val) {
        this(val, null);
    }

    public Node(T val, Node<T> next) {
        this.val = val;
        this.next = next;
    }
}
```




**How do we insert ”10” (after “1”)?**

```
Node<Integer> temp = new Node<>(10);
last.next = temp;
```

# Memory (Array)



# Lists (Arrays vs Linked List)

| Function | Array | Linked List |
|---|---|---|
| Insert at beginning | O(n) | O(1) |
| Insert at end | O(1) | **O(n)** |
| Insert at middle | O(n) | O(n) |
| Delete at beginning | O(n) | O(1) |
| Delete at end | O(1) | **O(n)** |
| Delete at middle | O(n) | O(n) |
| Find | O(n) | O(n) |
| Get Element at index | O(1) | O(n) |
| Print | O(n) | O(n) |
| Size | O(1) | **O(n)** |

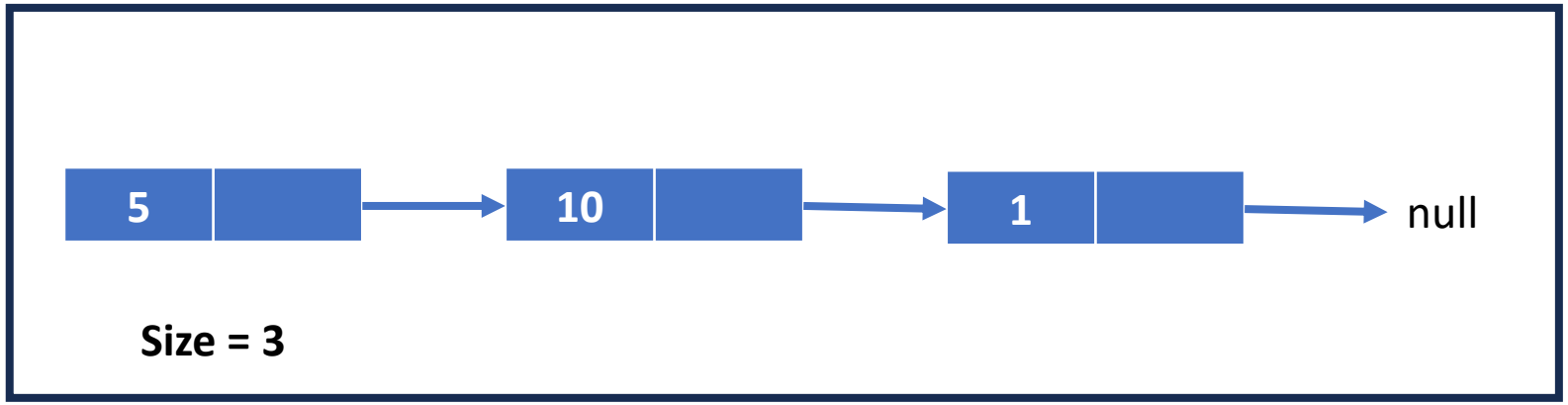
# Lists (Arrays vs Linked List)

| Function | Array | Linked List |
|---|---|---|
| Insert at beginning | O(n) | O(1) |
| Insert at end | O(1) | **O(n)** |
| Insert at middle | O(n) | O(n) |
| Delete at beginning | O(n) | O(1) |
| Delete at end | O(1) | **O(n)** |
| Delete at middle | O(n) | O(n) |
| Find | O(n) | O(n) |
| Get Element at index | O(1) | O(n) |
| Print | O(n) | O(n) |
| Size | O(1) | **O(n)** |

Can we improve time complexity for items bolded in **red**?

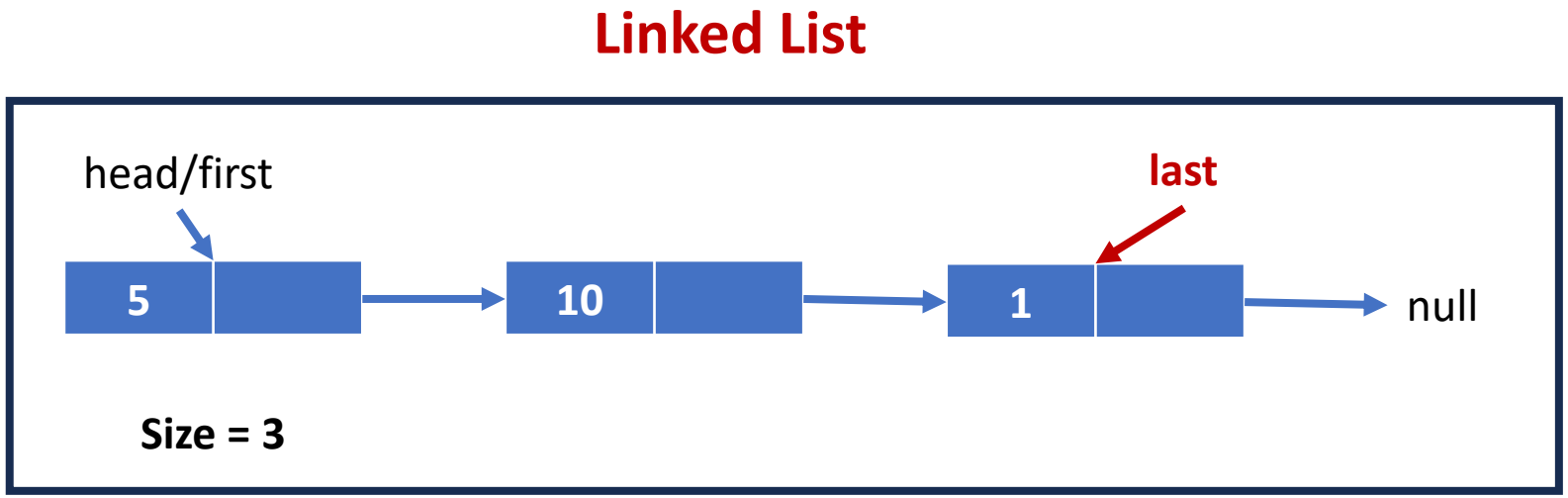
# Lists (Arrays vs Linked List)

**Linked List**




Add size field and update it every time an element is added/deleted

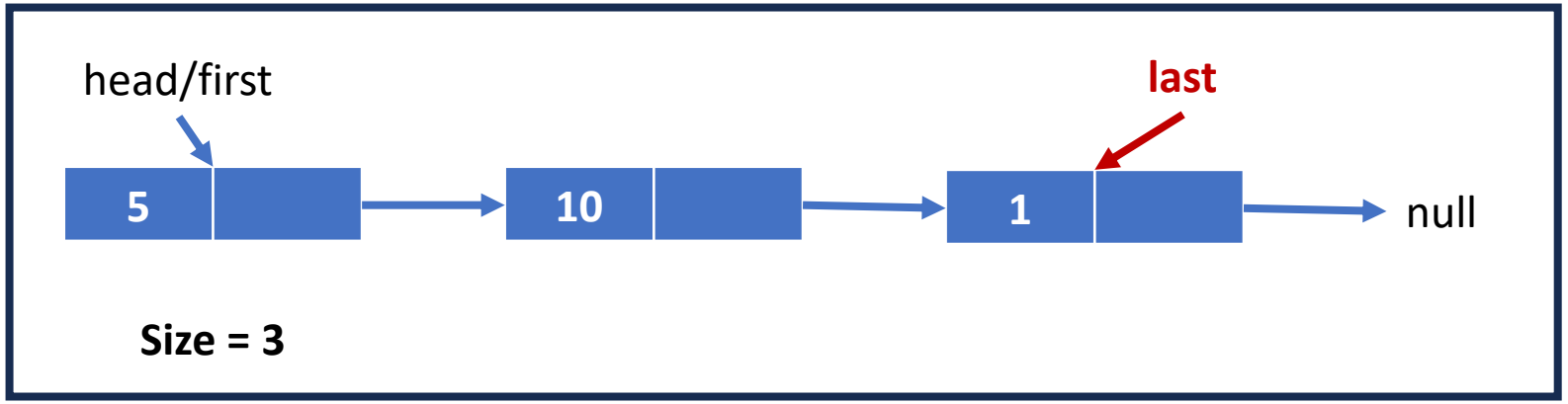
# Lists (Arrays vs Linked List)



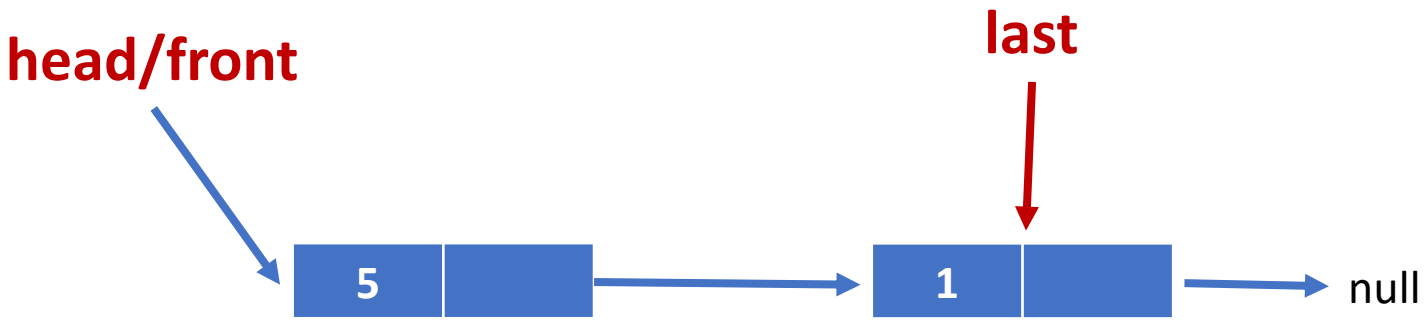

1. Add size field and update it every time an element is added/deleted.
2. Add last pointer that points to the last element!
    1. Now we can add a new node to the end in O(1)
    2. We can delete an element from the end in O(1)

# Linked List (Implementation)

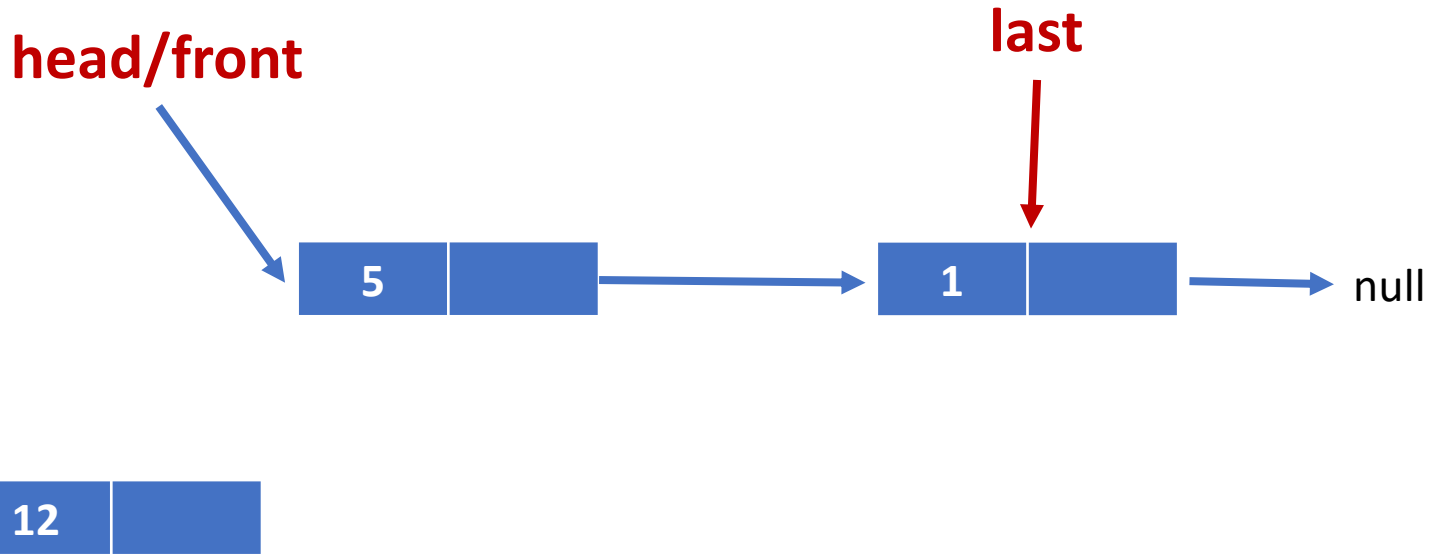
```
public class LinkedList<T> {
    Node<T> first;
    Node<T> last;
    int size;

    public LinkedList() {
        first = last = null;
        size = 0;
    }
}
```



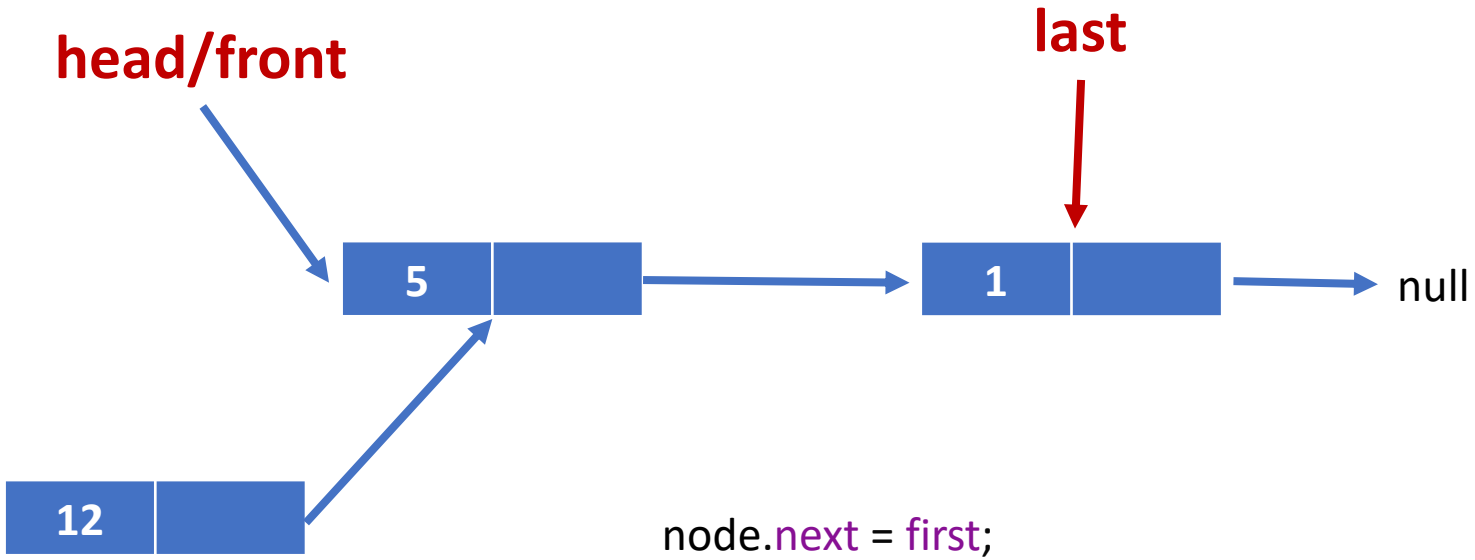
# Linked List (addFirst)



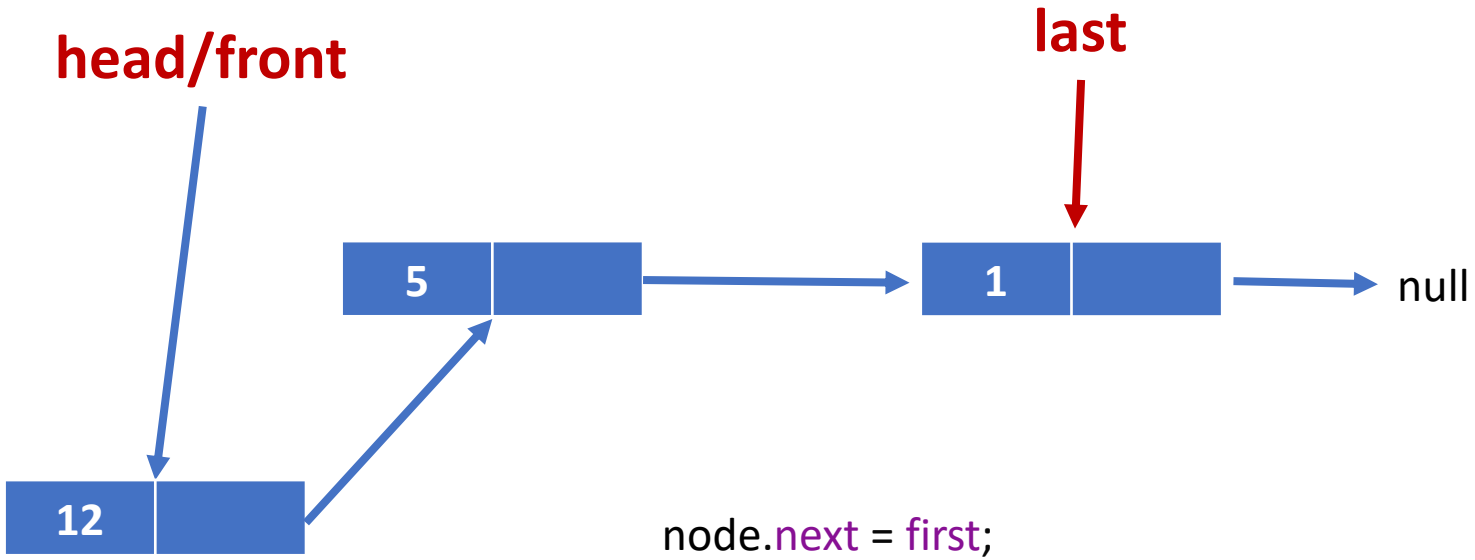
# Linked List (addFirst)



# Linked List (addFirst)



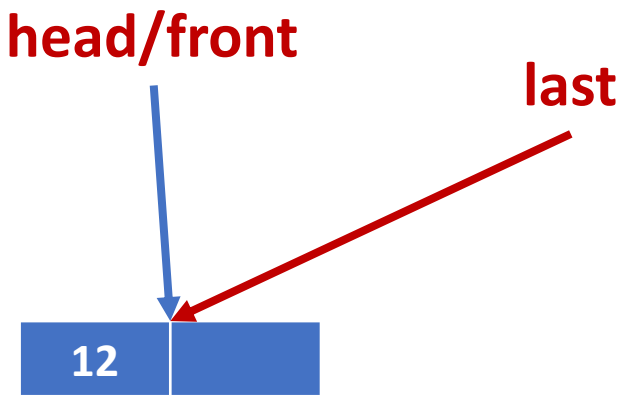
# Linked List (addFirst)




```
node.next = first;
first = node;
```

# Linked List (addFirst)

**What if it's empty?**




first = last = node;

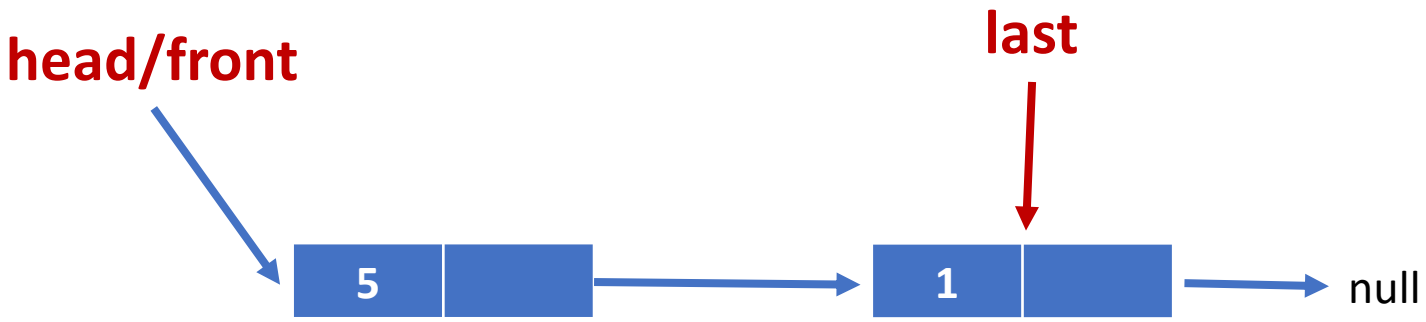
# Linked List (addFirst)

```
public void addFirst(T object) {
    Node<T> node = new Node<>(object);
    if (size == 0) {
        first = last = node;
    } else {
        node.next = first;
        first = node;
    }

    size++;
}
```

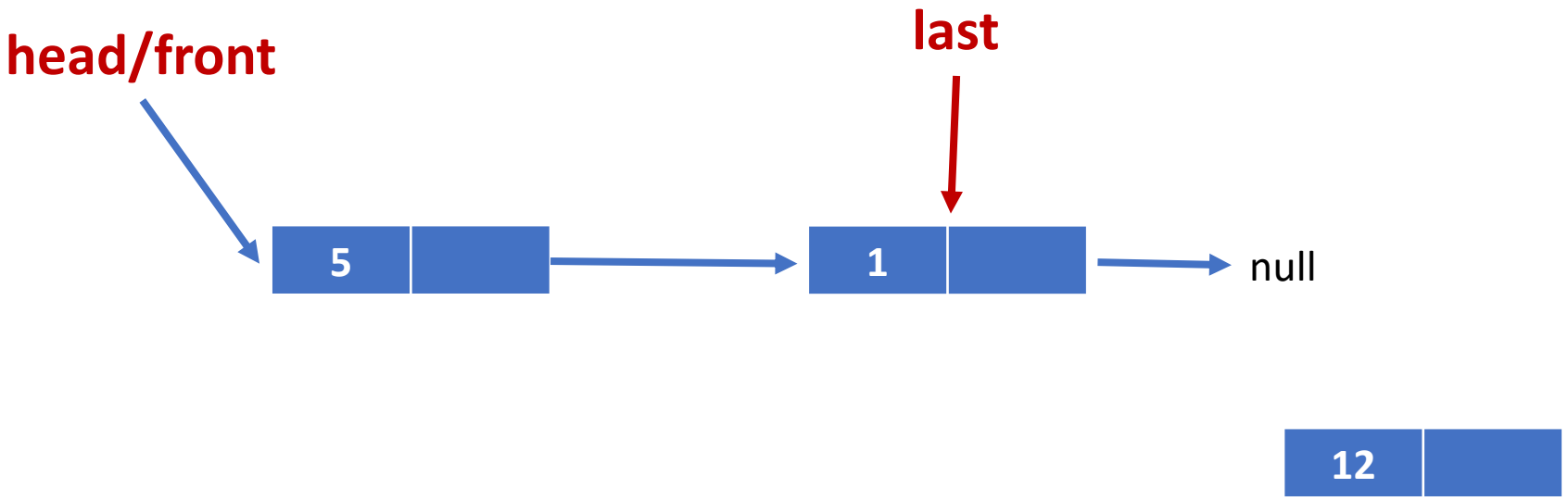
# Linked List (getFirst)

```
public T getFirst() {
    if (size == 0) {
        return null;
    }

    return first.val;
}
```

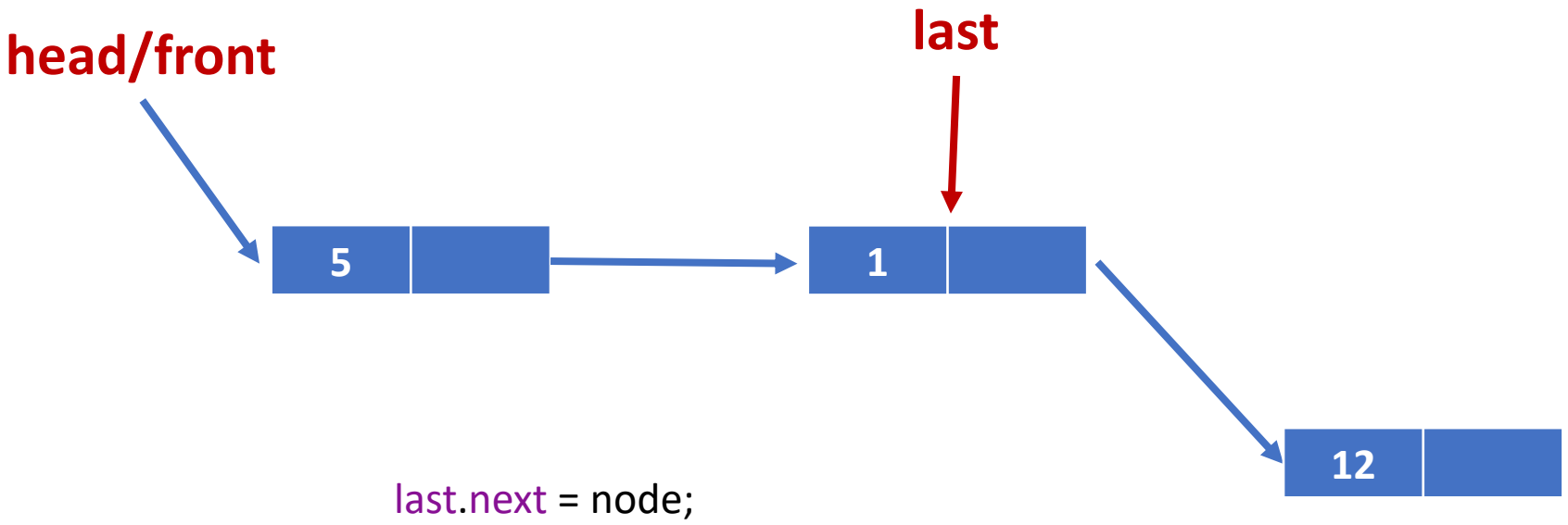
# Linked List (addLast)



# Linked List (addLast)

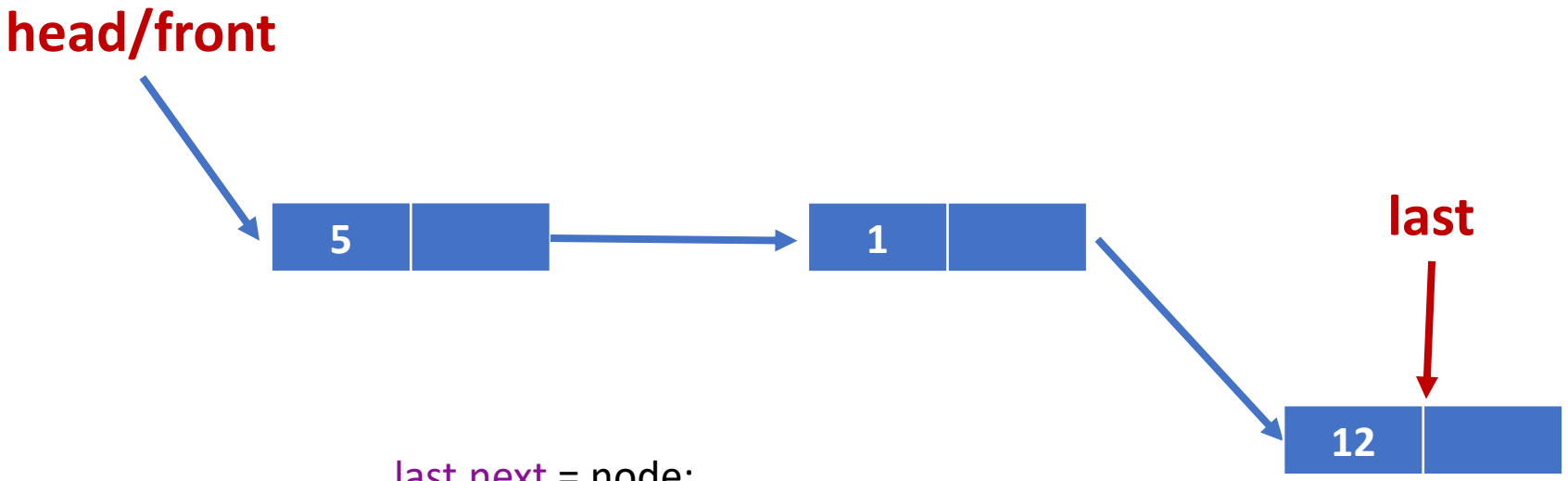


# Linked List (addLast)




last.next = node;

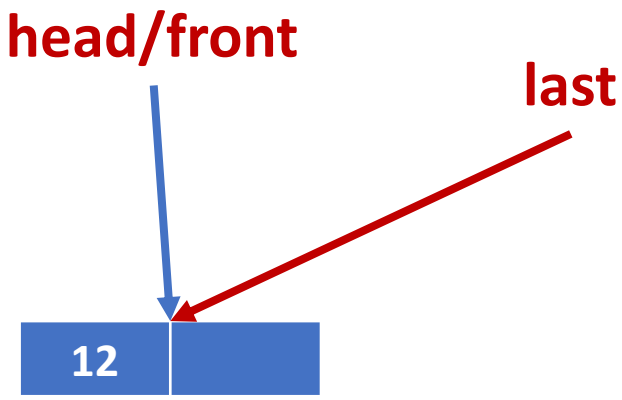
# Linked List (addLast)




```
last.next = node;
last = last.next;
```

# Linked List (addLast)

**What if it's empty?**



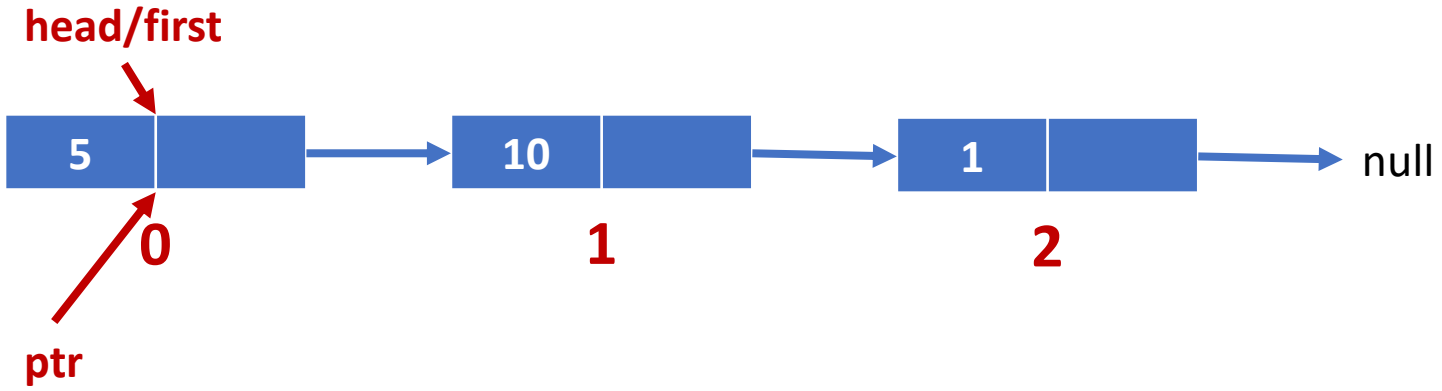
# Linked List (addLast)

```
public void addLast(T object) {
    Node<T> node = new Node<>(object);
    if (size == 0) {
        first = last = node;
    } else {
        last.next = node;
        last = last.next;
    }

    size++;
}
```

# Linked List (getLast)

```
public T getLast() {
    if (size == 0) {
        return null;
    }

    return last.val;
}
```

# Linked List (get)

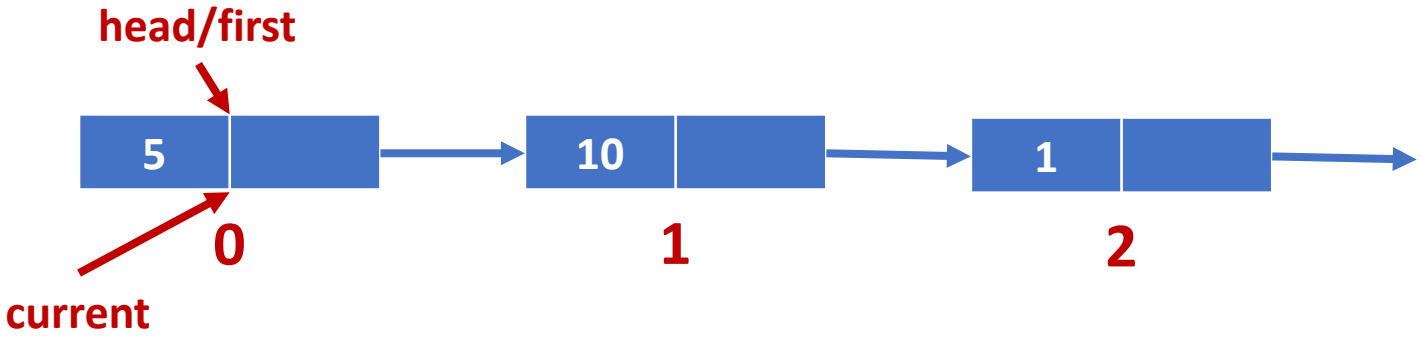



```
T get(int index);
```

# Linked List (get)

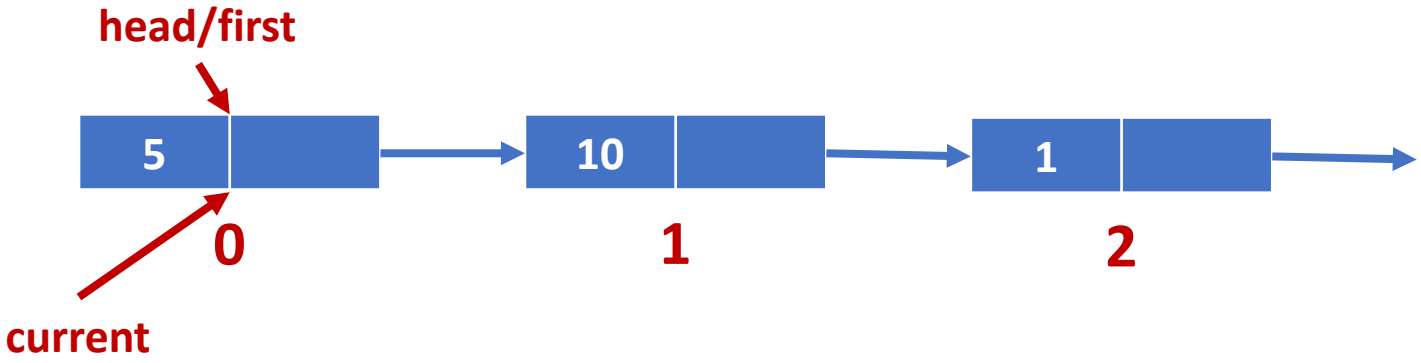
```
public T get(int index) {
    if (size == 0) {
        return null;
    }

    if (index < 0 || index >= size) {
        return null;
    }

    Node<T> current = first;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }

    return current.val;
}
```

# Linked List (add)



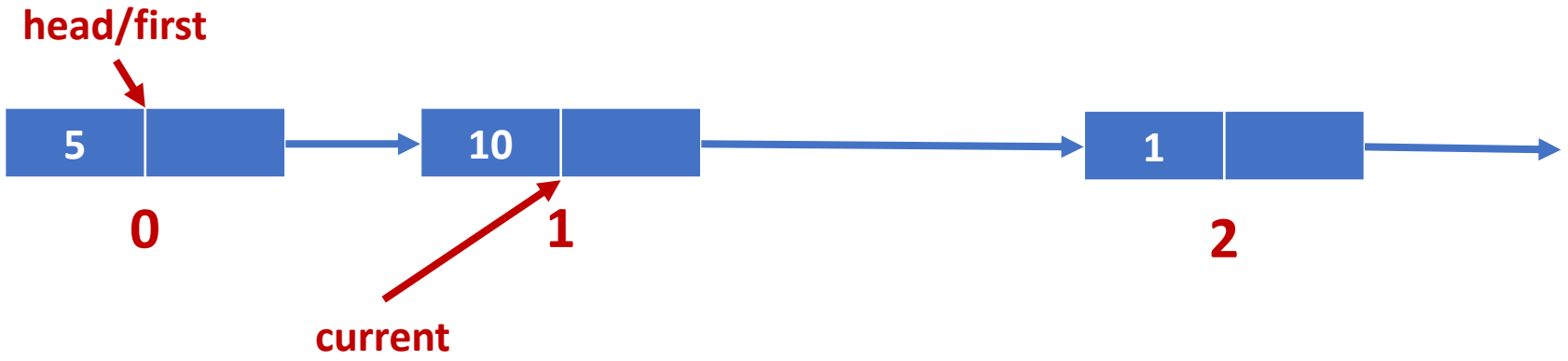

```
void add(int index, T element);
```

# Linked List (add)




```
void add(int index, T element);

add(2, 15);
```

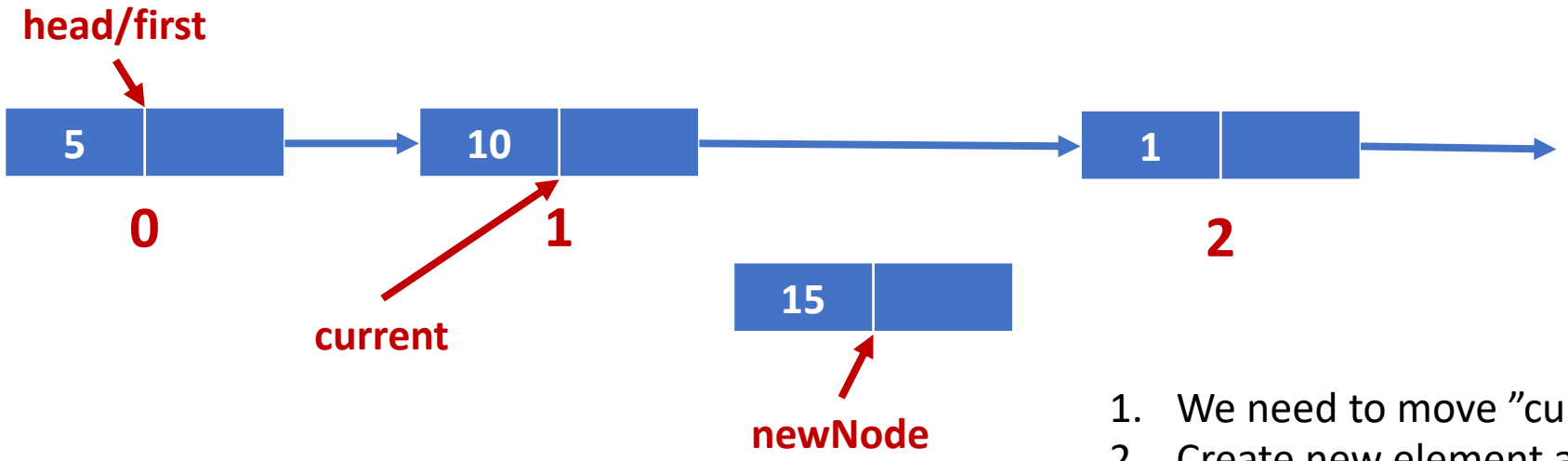
# Linked List (add)




1. We need to move ”current” to previous element

```
void add(int index, T element);

add(2, 15);
```

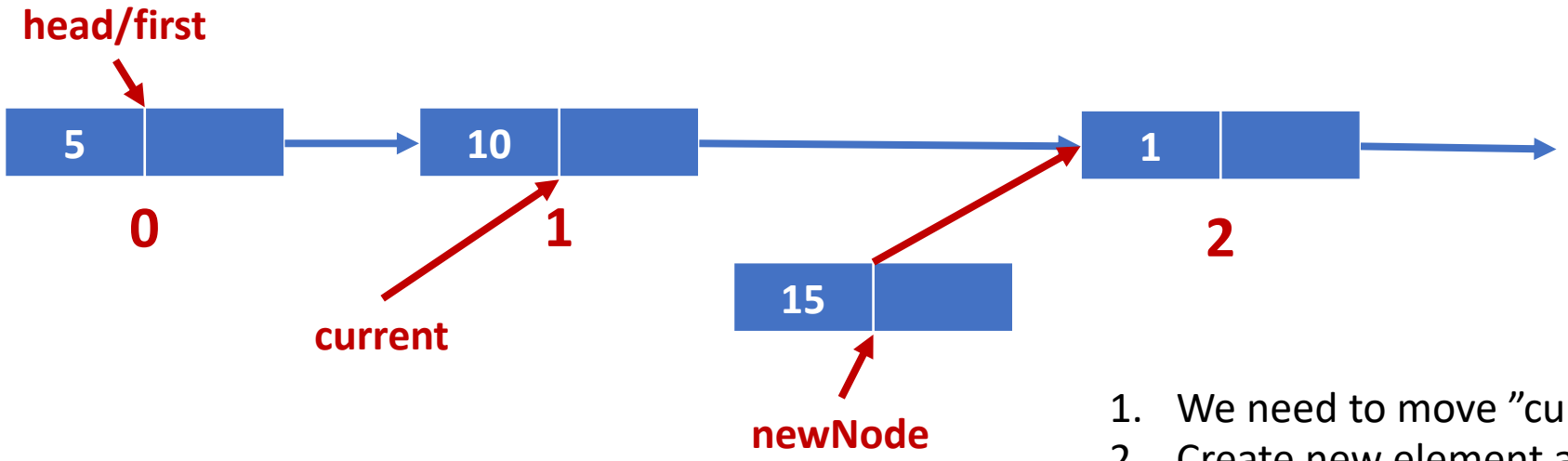
# Linked List (add)




1. We need to move ”current” to previous element
2. Create new element as “newNode”

```
void add(int index, T element);

add(2, 15);
```

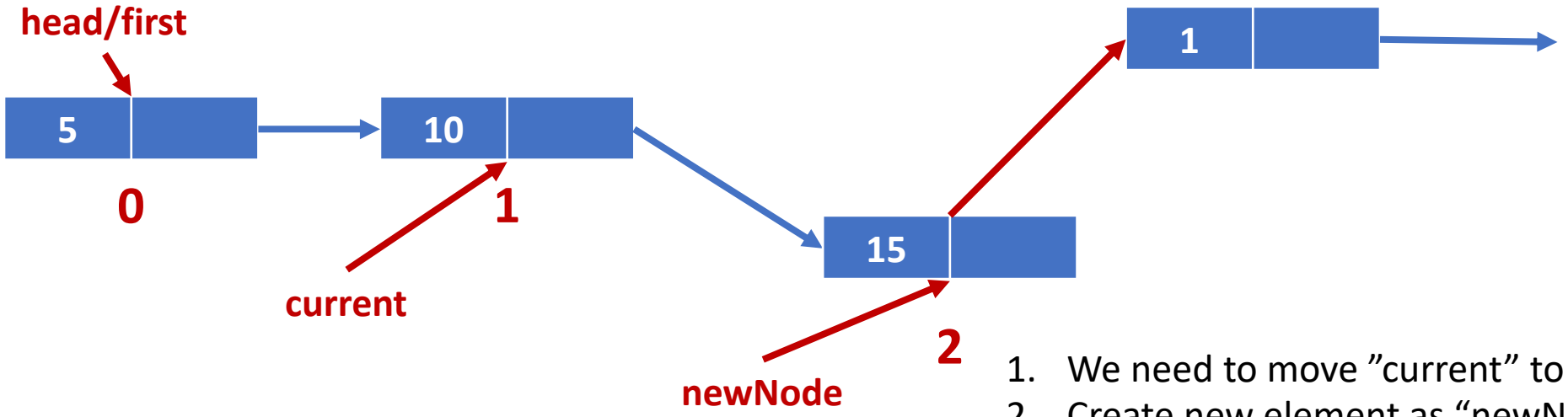
# Linked List (add)




```
void add(int index, T element);

add(2, 15);
```

1. We need to move ”current” to previous element
2. Create new element as “newNode”
3. Link new element “newNode” with the next node.
   node.next = current.next;

# Linked List (add)



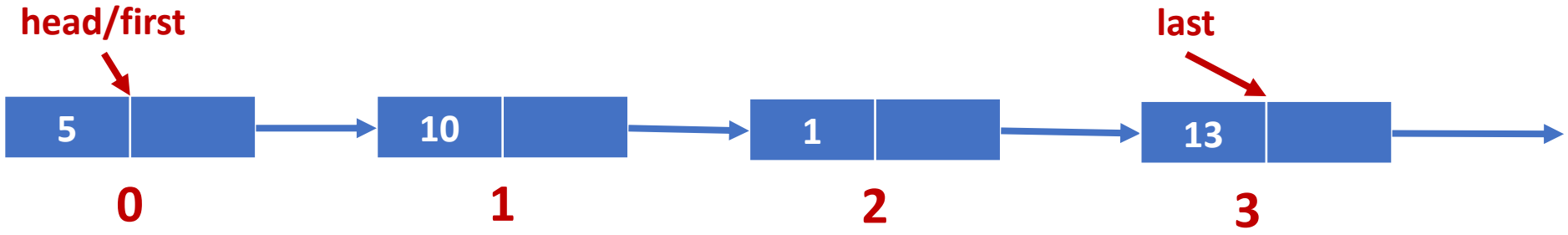

```
void add(int index, T element);

add(2, 15);
```

1. We need to move ”current” to previous element
2. Create new element as “newNode”
3. Link new element “newNode” with the next node.
   node.next = current.next;
4. Link ”current” with “newNode”
   current.next = newNode;

# Linked List (add)

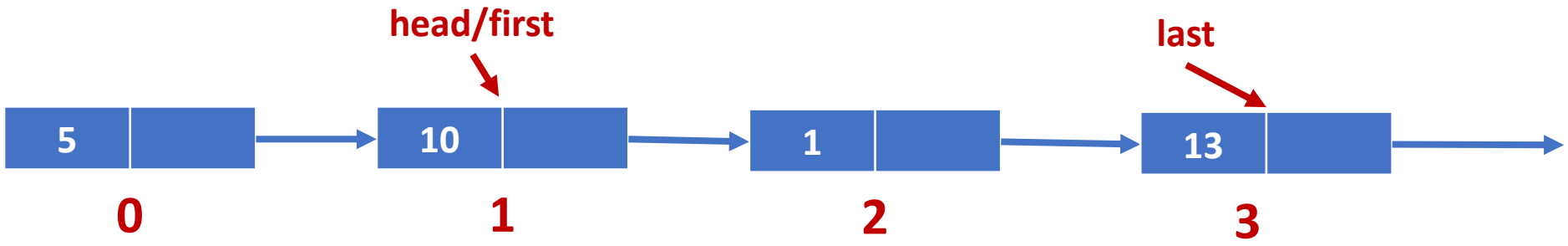
```
private void add(int index, T element) {
   if (index <= 0) {
      addFirst(element);
   } else if (index >= size) {
      addLast(element);
   } else {
      Node<T> current = first;
      for (int i = 0; i < index - 1; i++) {
         current = current.next;
      }

      Node<T> newNode = new Node<>(element);
      newNode.next = current.next;
      current.next = newNode;
      size++;
   }
}
```

# Linked List (removeFirst)




```
boolean removeFirst();
```

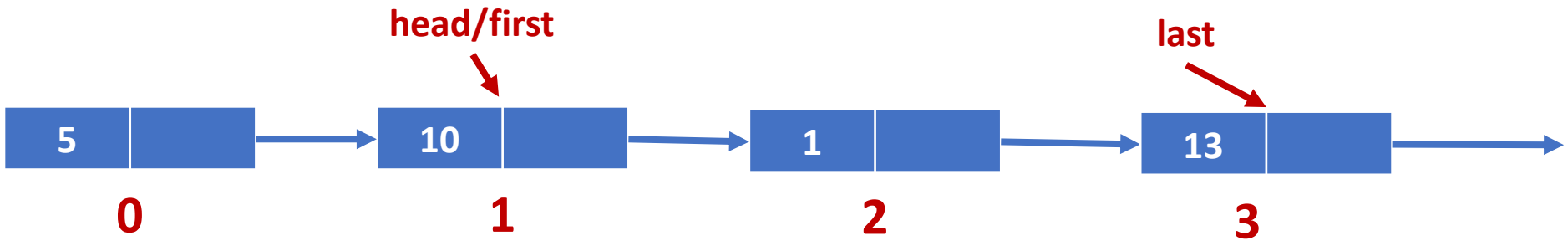
# Linked List (removeFirst)




We just need to move the head/first one step!

```
boolean removeFirst();
```

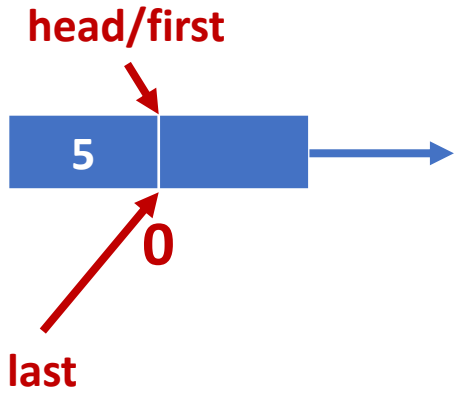
# Linked List (removeFirst)




We just need to move the head/first one step!

```
boolean removeFirst();
```

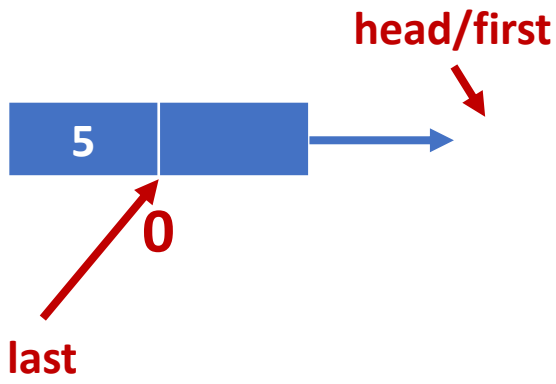
# Linked List (removeFirst)




What happens if we have one element?

```
boolean removeFirst();
```

# Linked List (removeFirst)




What happens if we have one element?

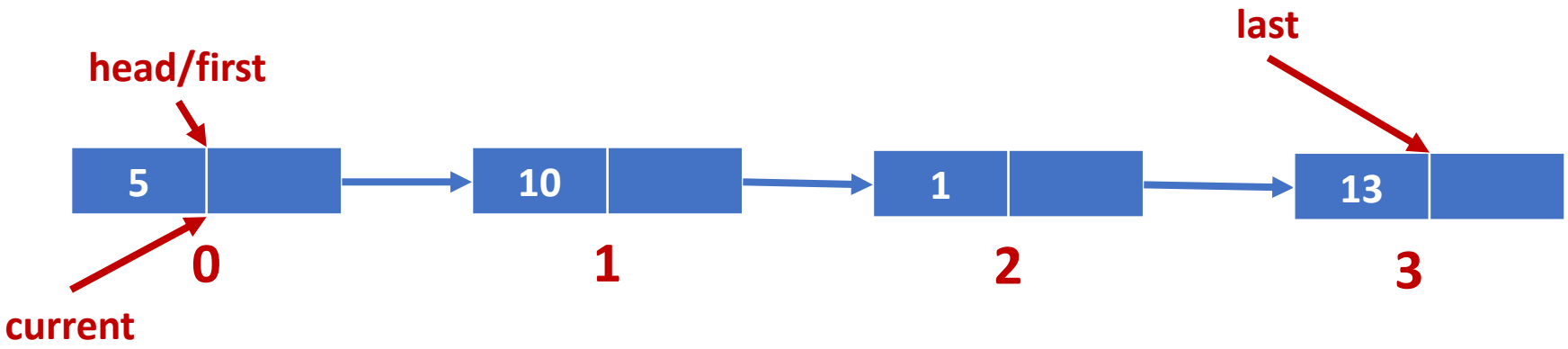
“first“ now points to “null” but last didn’t move!!!
This is a special case that we need to take care of

```
boolean removeFirst();
```

# Linked List (removeFirst)

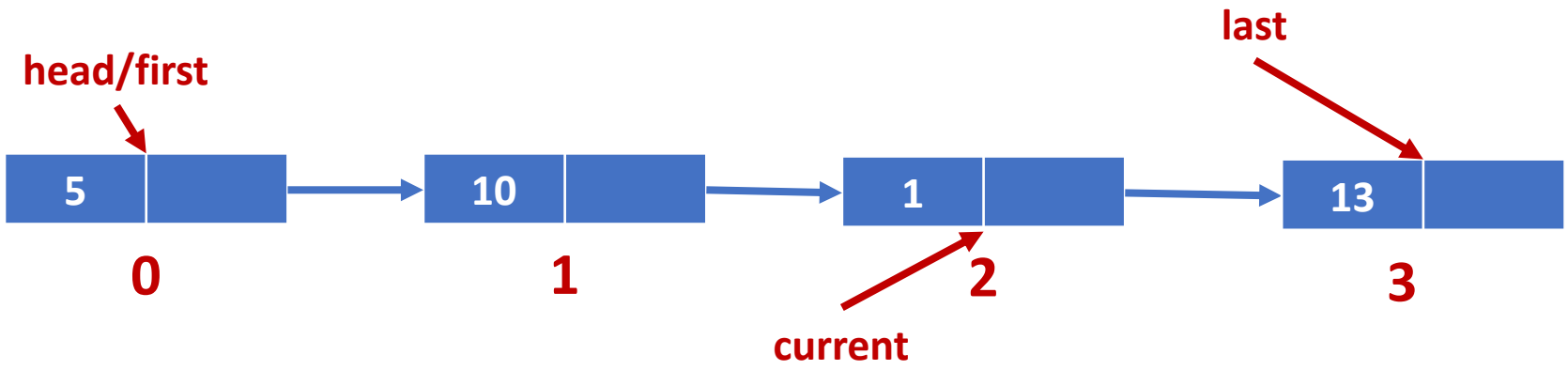
```
private boolean removeFirst() {
    if (first == null) {
        return false;
    }

    // Check if we have one element only
    if (first == last) {
        first = last = null;
        return true;
    }

    first = first.next;
    size-;

    return true;
}
```

# Linked List (removeLast)




```
boolean removeLast();
```

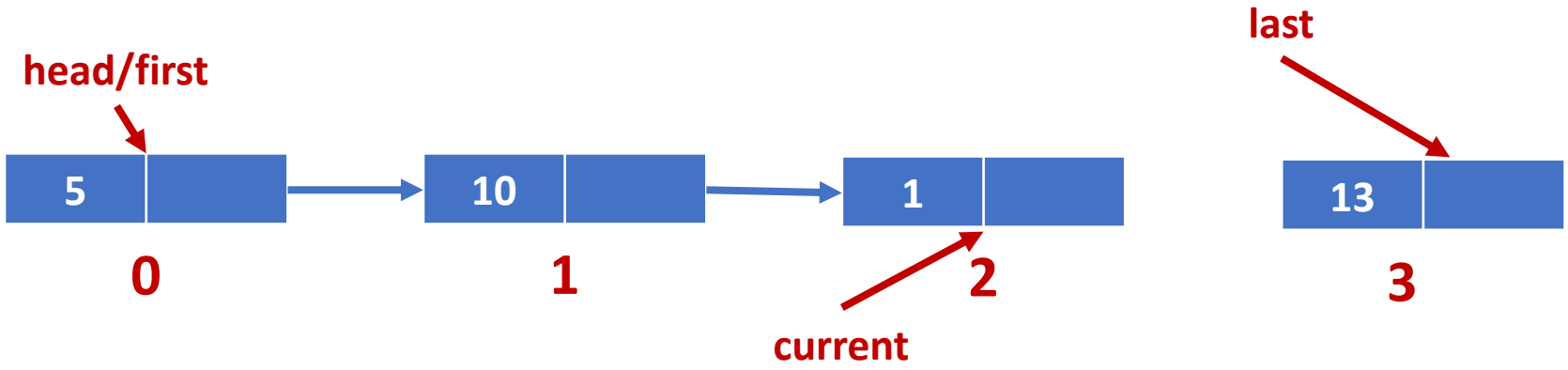
# Linked List (removeLast)




1. We need to move ”current” to previous element

```
boolean removeLast();
```

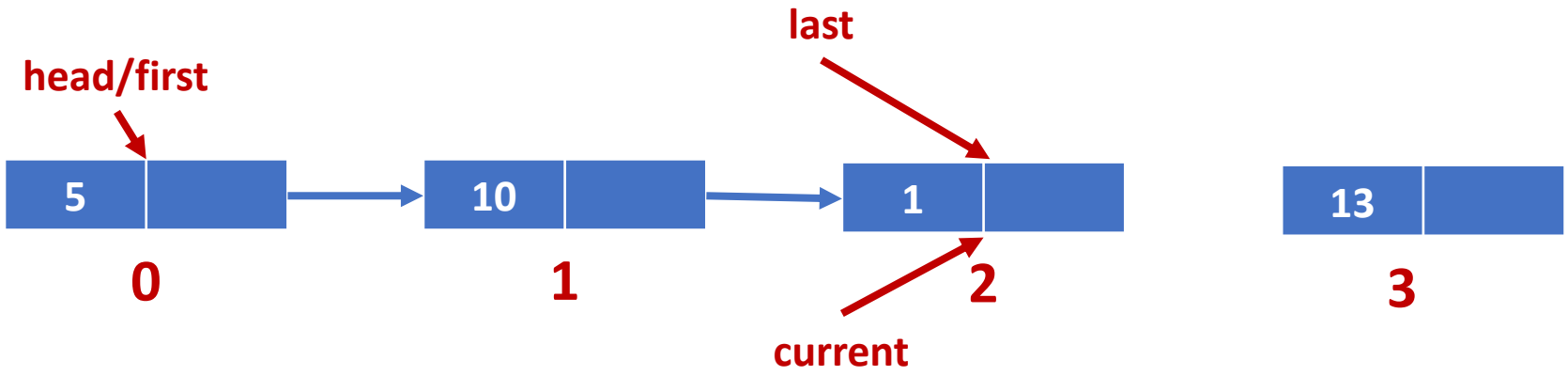
# Linked List (removeLast)




1. We need to move ”current” to previous element.
2. Set “current.next” to null

```
boolean removeLast();
```

# Linked List (removeLast)



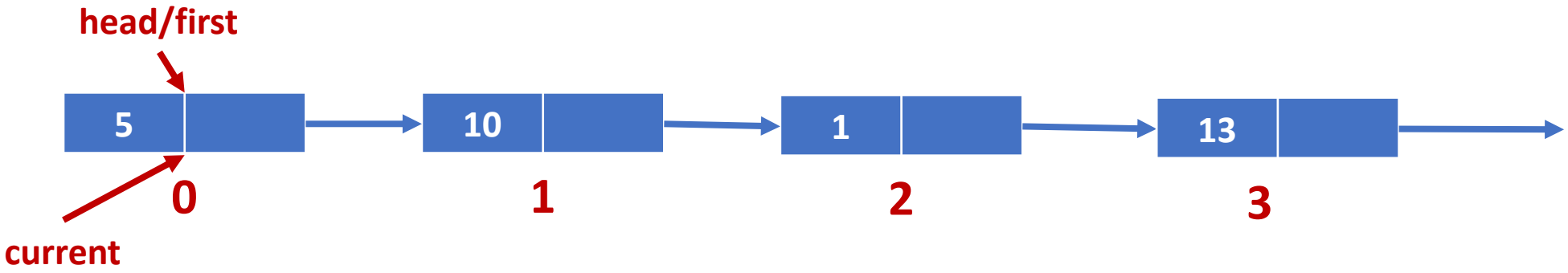

boolean removeLast();

1. We need to move ”current” to previous element.
2. Set “current.next” to null
3. Set “last“ to ”current”

# Linked List (removeLast)

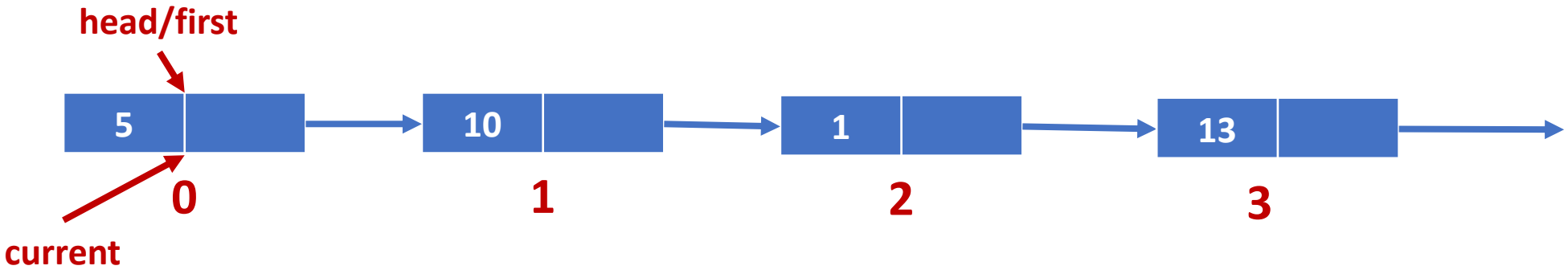
```
public boolean removeLast() {
    if (first == null) {
        return false;
    }

    // Check if we have one element only
    if (first == last) {
        first = last = null;
        return true;
    }

    Node<T> current = first;
    for (int i = 0; i < size - 2; i++) {
        current = current.next;
    }

    current.next = null;
    last = current;
    size-;

    return true;
}
```

# Linked List (remove)



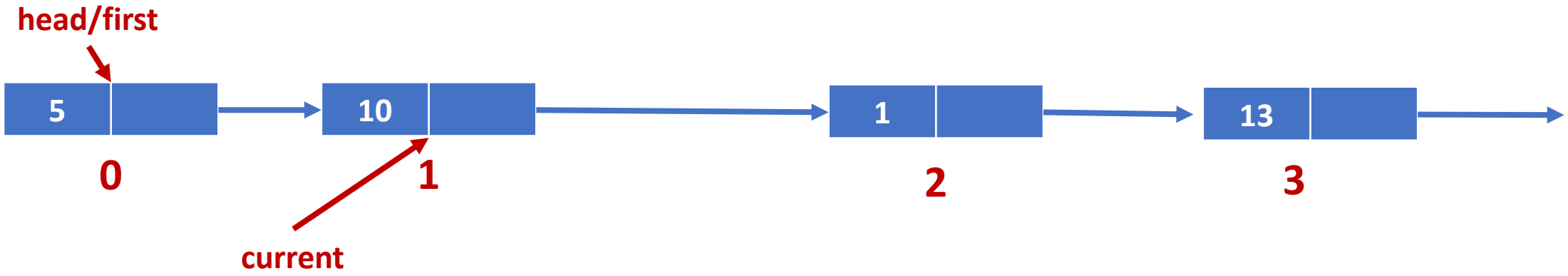

```
boolean remove(int index);
```

# Linked List (remove)




```
boolean remove(int index);

remove(2);
```

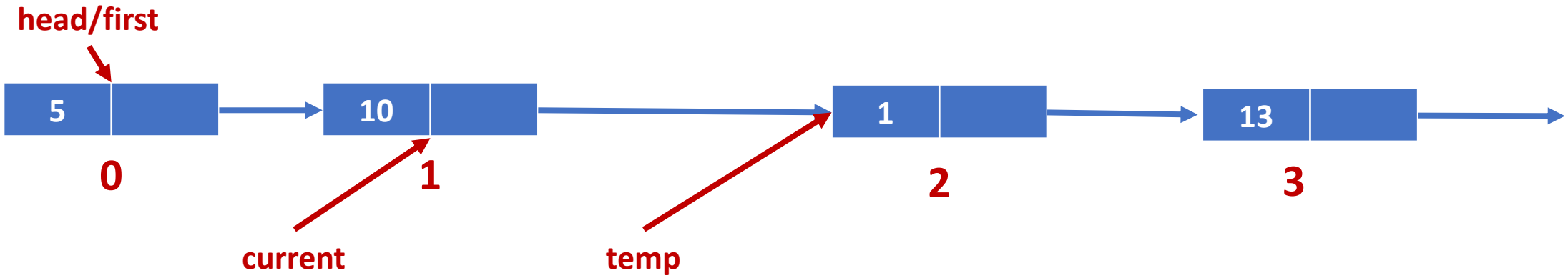
# Linked List (remove)




1. We need to move ”current” to previous element

```
boolean remove(int index);

remove(2);
```

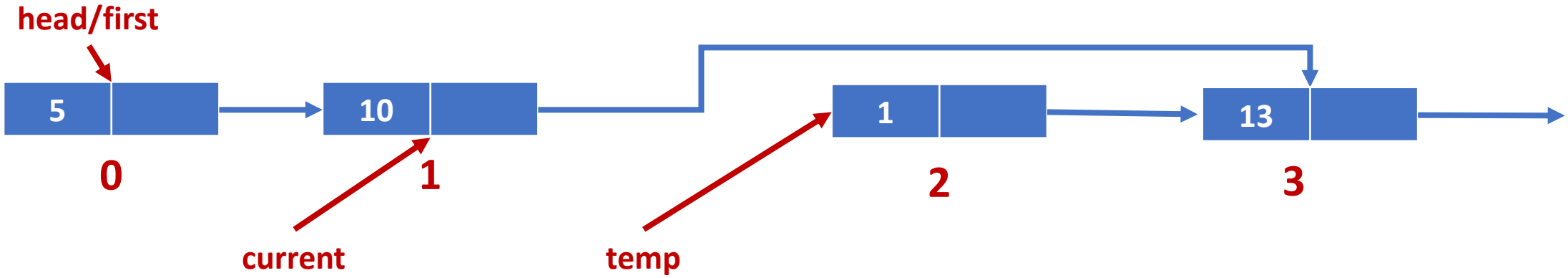
# Linked List (remove)




1. We need to move ”current” to previous element
2. Set a pointer “temp” to the element
   temp = current.next;

```
boolean remove(int index);

remove(2);
```

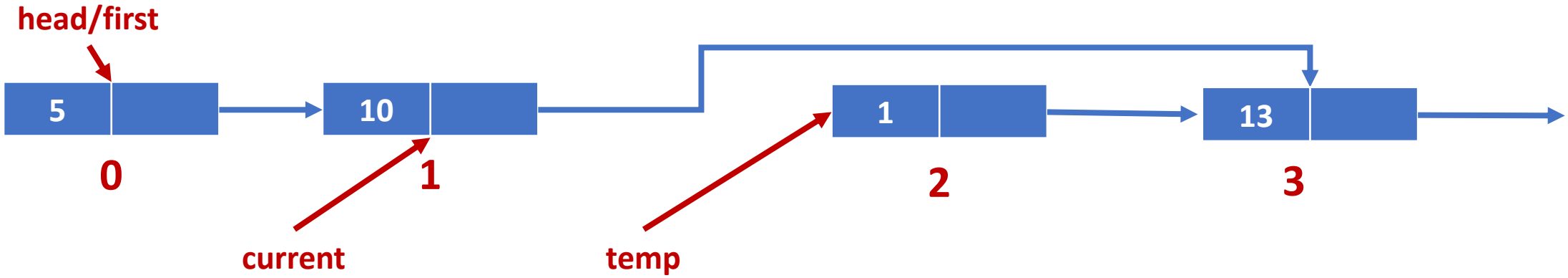
# Linked List (remove)




```
boolean remove(int index);

remove(2);
```

1. We need to move ”current” to previous element
2. Set a pointer “temp” to the element
   temp = current.next;
3. Set “current” next element to be the element after the temp element “3”
4. current.next = temp.next;

# Linked List (remove)




```
boolean remove(int index);

remove(2);
```

1. We need to move ”current” to previous element
2. Set a pointer “temp” to the element
   temp = current.next;
3. Set “current” next element to be the element after the temp element “3”
4. current.next = temp.next;

# Linked List (remove)

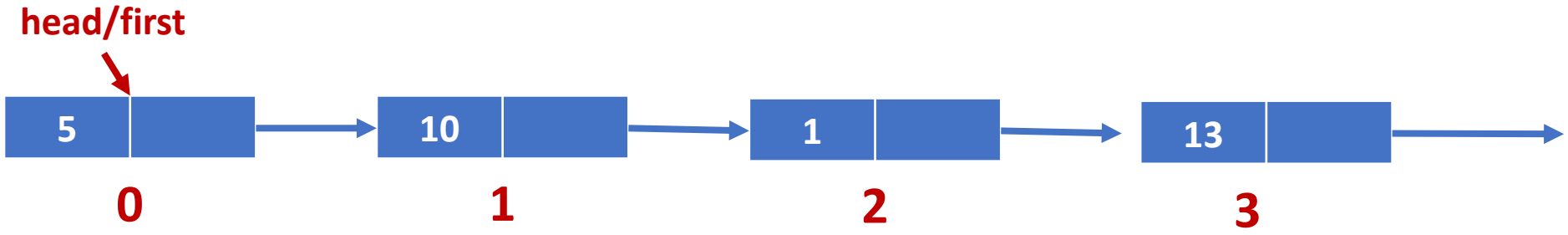
```
private boolean remove(int index) {
   if (index <= 0) {
      return removeFirst();
   } else if (index >= size) {
      return removeLast();
   } else {
      Node<T> current = first;
      for (int i = 0; i < index - 1; i++) {
         current = current.next;
      }

      Node<T> temp = current.next;
      current.next = temp.next;
      size-;

      return false;
   }
}
```

# Linked List (Returning removed element!)

```
T removeLast();

T removeFirst();

T remove(int index);

int size();

void print();

void clear();
```

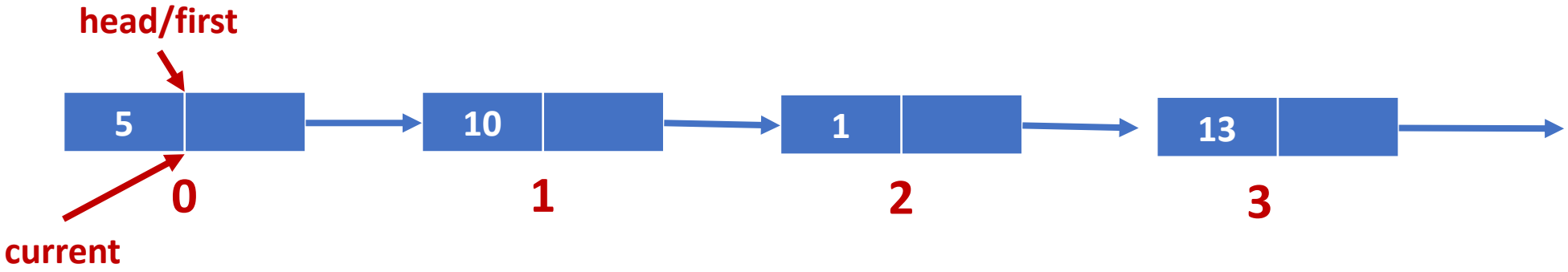
**This is an exercise!**

# Linked List (find)




```
T find(T element);
```

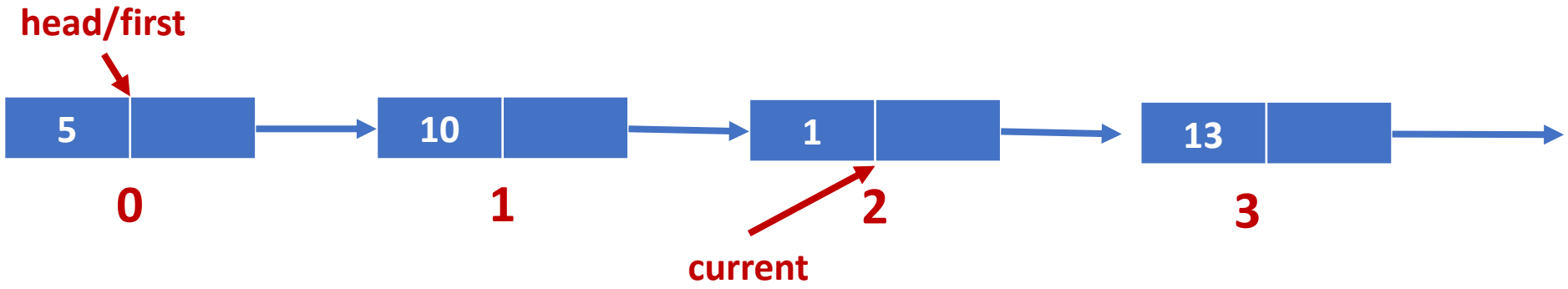
# Linked List (find)




1. Set “current“ to “first”

```
T find(T element);

T find(1);
```

# Linked List (find)




1. Set “current“ to “first”
2. Keep moving “current” until element is found!

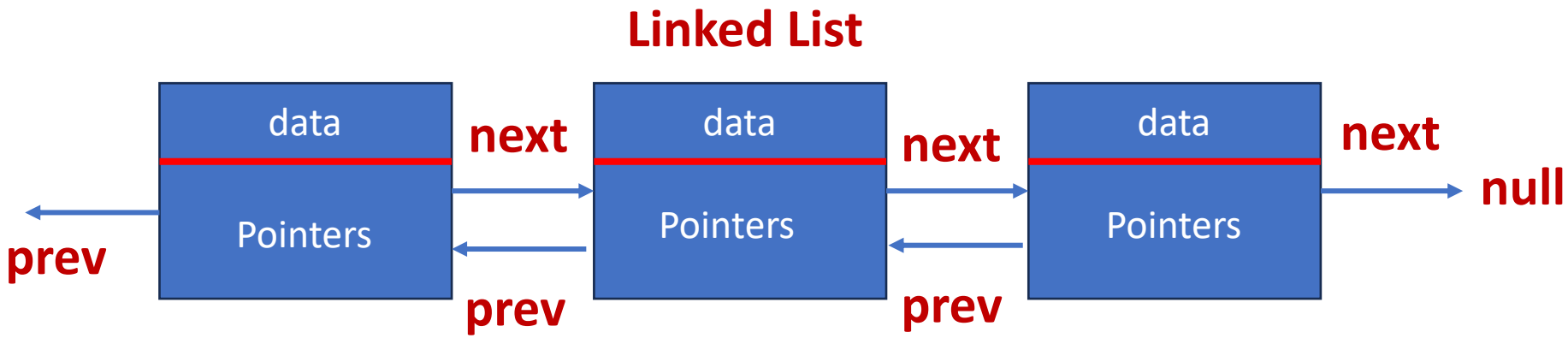
```
T find(T element);

T find(1);
```

# Linked List (find)

```
public T find(T val) {
    Node<T> current = first;
    while (current != null && current.val.equals(val)) {
        current = current.next;
    }

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

    return current.val;
}
```

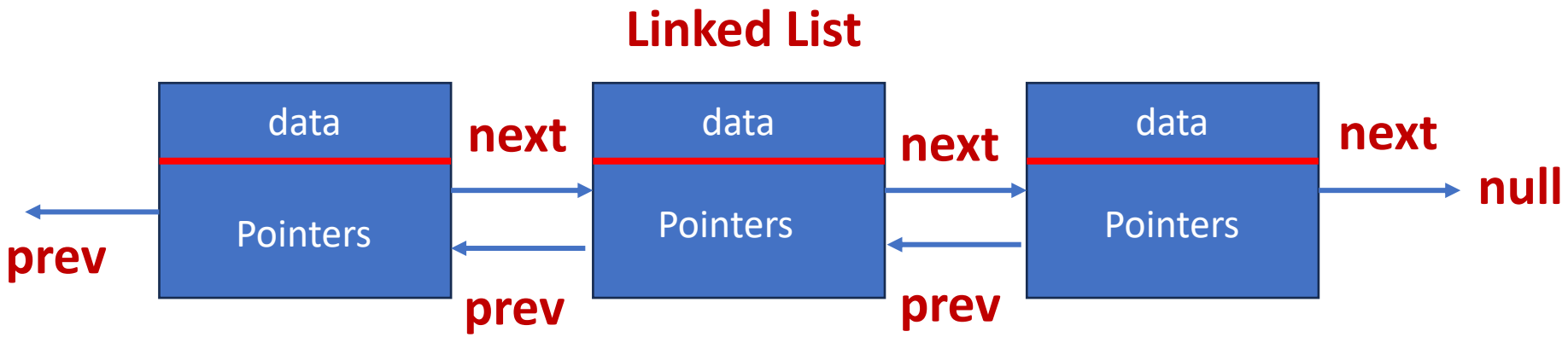
# Double-Linked List

Same as Linked List but now we have an extra pointer pointing towards the previous element "**prev**"



# Double-Linked List

Same as Linked List but now we have an extra pointer pointing towards the previous element "**prev**"

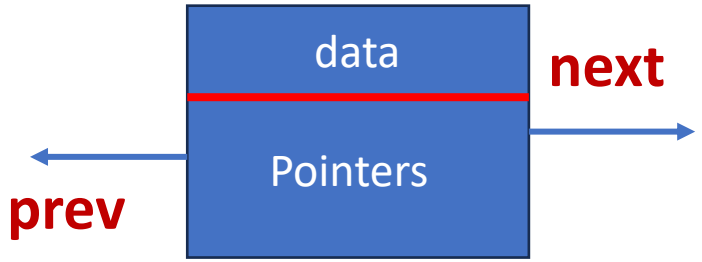



1. When adding a new node when to fix both "next" and "prev"
2. Having "prev" reduces time complexity of **removeLast** to **O(1)**!
3. Removing a node becomes easier (we don't need to find previous element)

1. Extra Space!
2. Complex programing

# Double-Linked List

```
public class Node<T> {
    public T val;
    public Node<T> next;
    public Node<T> prev;

    public Node(T val) {
        this(val, null, null);
    }

    public Node(T val, Node<T> next) {
        this(val, next, null);
    }

    public Node(T val, Node<T> next, Node<T> prev) {
        this.val = val;
        this.next = next;
        this.prev = prev;
    }
}
```

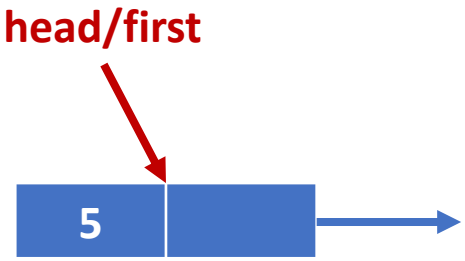


# Double-Linked List

```
public class DoubleLinkedList<T> {
    Node<T> first;
    Node<T> last;
    int size;

    public DoubleLinkedList() {
        first = last = null;
        size = 0;
    }
}
```

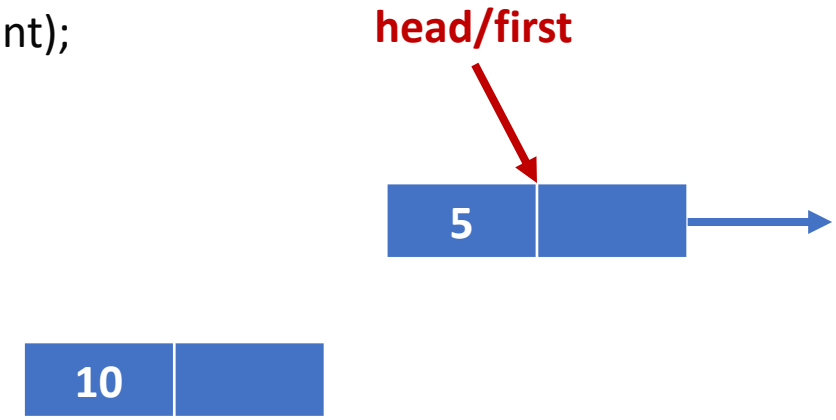
# Double-Linked List

```
public void addFirst(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        first = last = node;
    } else {
        node.next = first;
        first.prev = node;
        first = node;
    }
    size++;
}
```



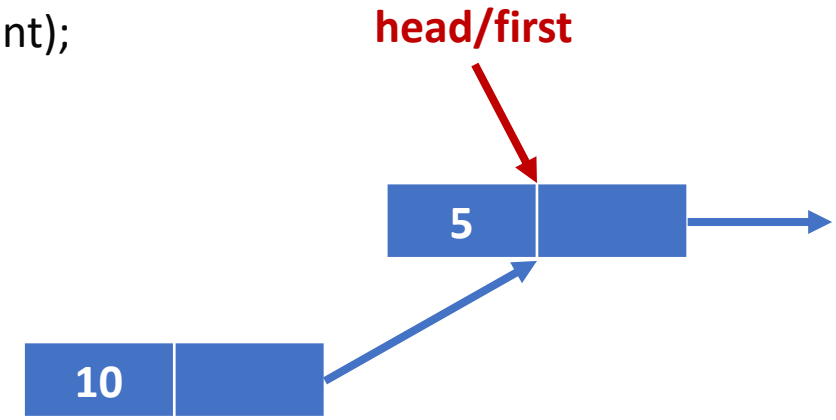
# Double-Linked List

```
public void addFirst(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        first = last = node;
    } else {
        node.next = first;
        first.prev = node;
        first = node;
    }
    size++;
}
```



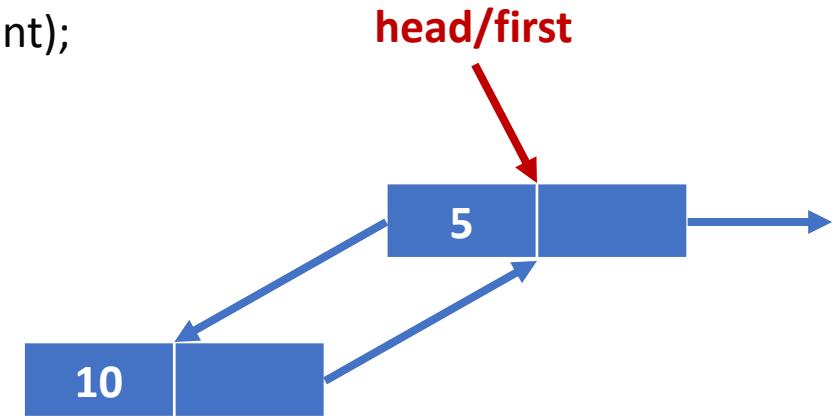
# Double-Linked List

```
public void addFirst(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        first = last = node;
    } else {
        node.next = first;
        first.prev = node;
        first = node;
    }
    size++;
}
```



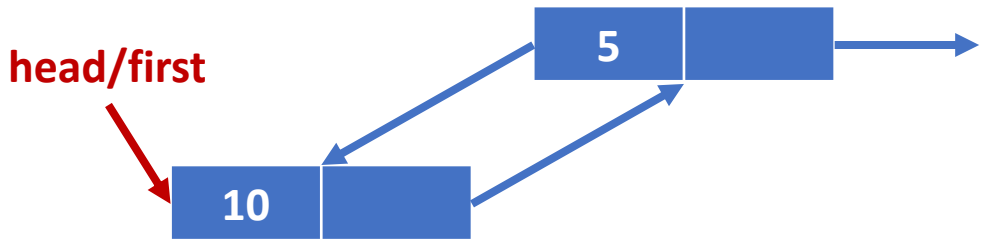
# Double-Linked List

```
public void addFirst(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        first = last = node;
    } else {
        node.next = first;
        first.prev = node;
        first = node;
    }
    size++;
}
```



# Double-Linked List

```
public void addFirst(T element) {
    Node<T> node = new Node<>(element);
    if (size == 0) {
        first = last = node;
    } else {
        node.next = first;
        first.prev = node;
        first = node;
    }
    size++;
}
```

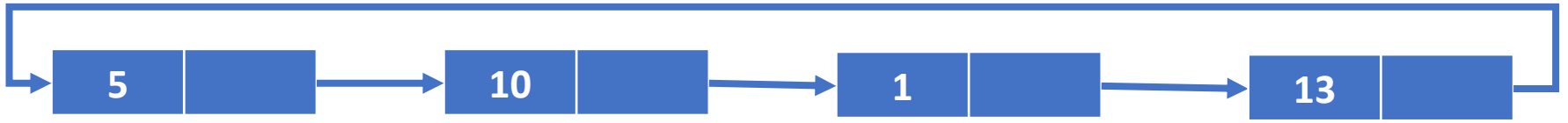


# Double-Linked List

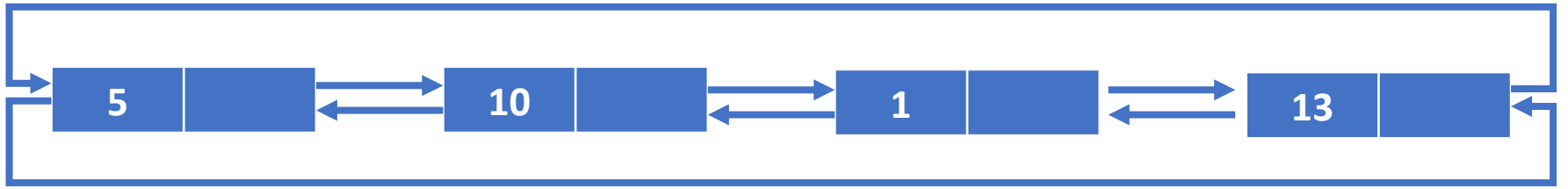
```
void addLast(T element);

void add(int index, T element);

boolean removeLast();

boolean removeFirst();

boolean remove(int index);

int find(T element);

int size();

void print();

void clear();
```

**This is an exercise!**

# Circular Linked List



# Double Circular Linked List



# Exercises

For LinkedList and DoubleLinkedList

```
boolean contains(T element);        // Returns true if list contains “element”. False otherwise

int indexOf(T element);             // Returns index of first occurrence of “element”. -1 otherwise

int lastIndexOf();                  // Returns index of last occurrence of “element”. -1 otherwise

void printReverse();                // Print list in reverse order
```

# Exercises

Write node class and List class for Circular Linked List and Double Circular LinkedList