

Linked Data Structures

Objectives

- **Understand linked structures**
- **Compare linked structures to array-based structures**
- **Understand implementations for linked structures**
- **Understand algorithms for managing a linked list**
- **Traversing linked structures**

Array Limitations

- **Fixed size**
- **Physically stored in consecutive memory locations, so to insert or delete items, may need to shift data**

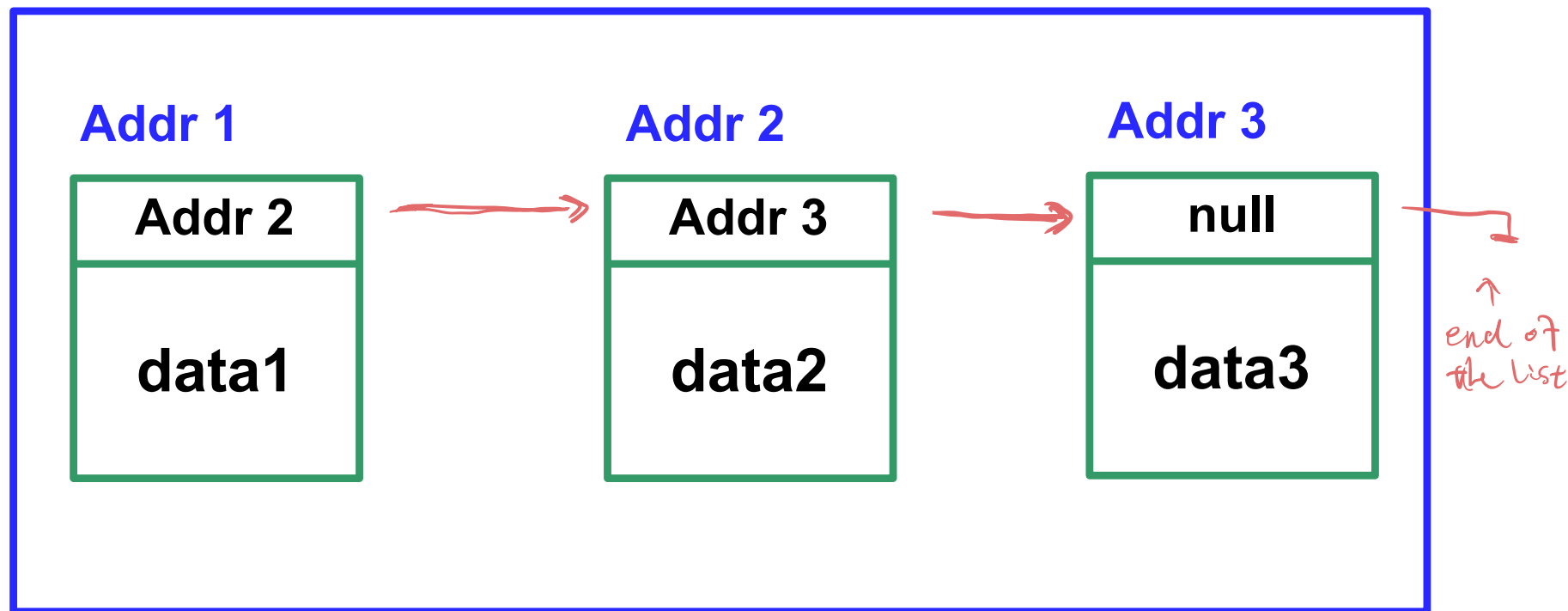


Linked Data Structures

- A **linked** data structure consists of items that are linked to other items
 - Each item points to another item

Memory

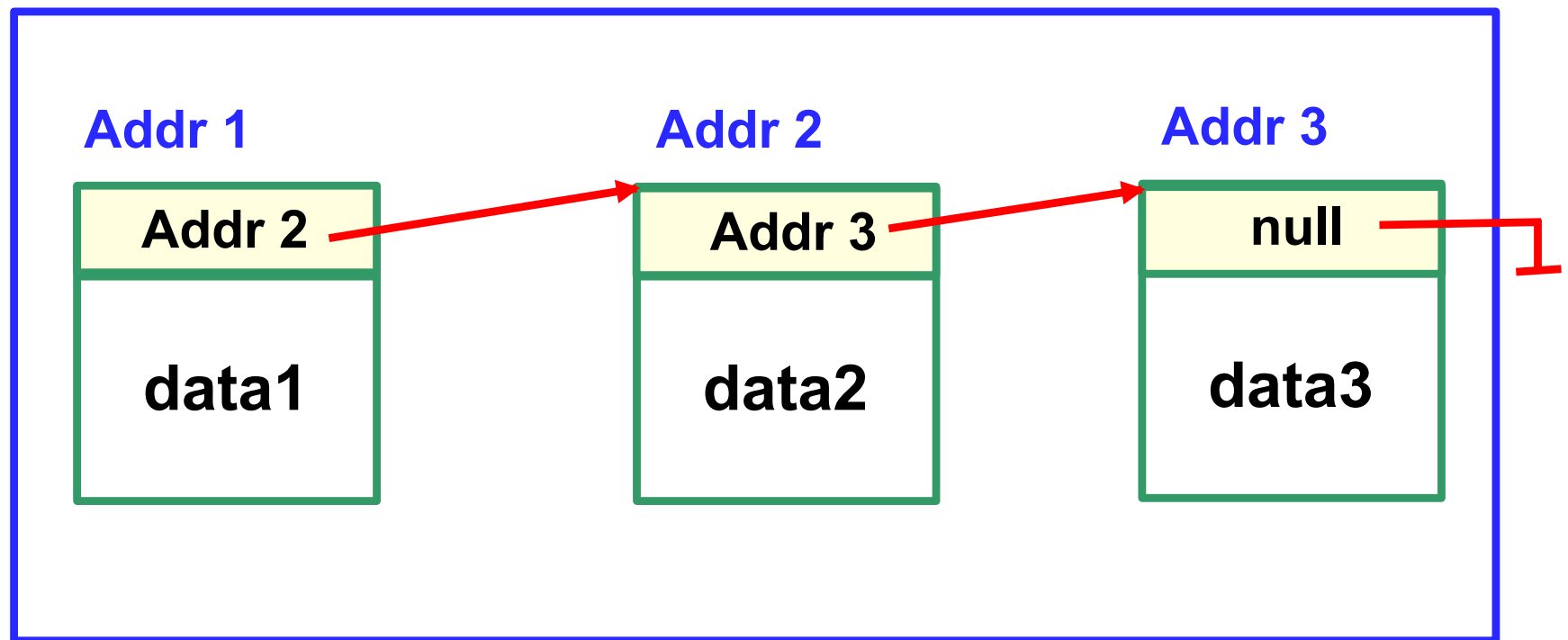
public node { Node nextNode, T data } {



Linked Data Structures

- A *linked* data structure consists of items that are linked to other items
 - Each item *points to* another item

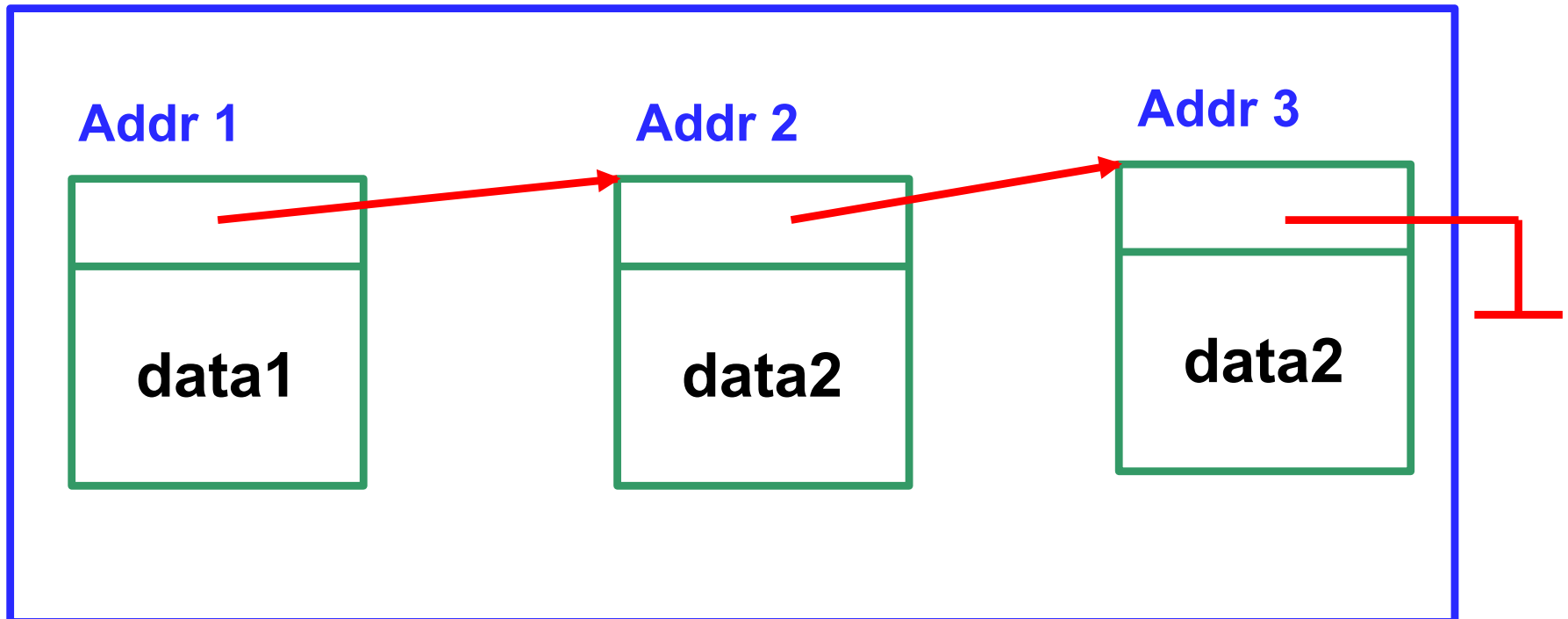
Memory



Linear Linked Data Structures

- **Singly linked list:** each item points to the next item

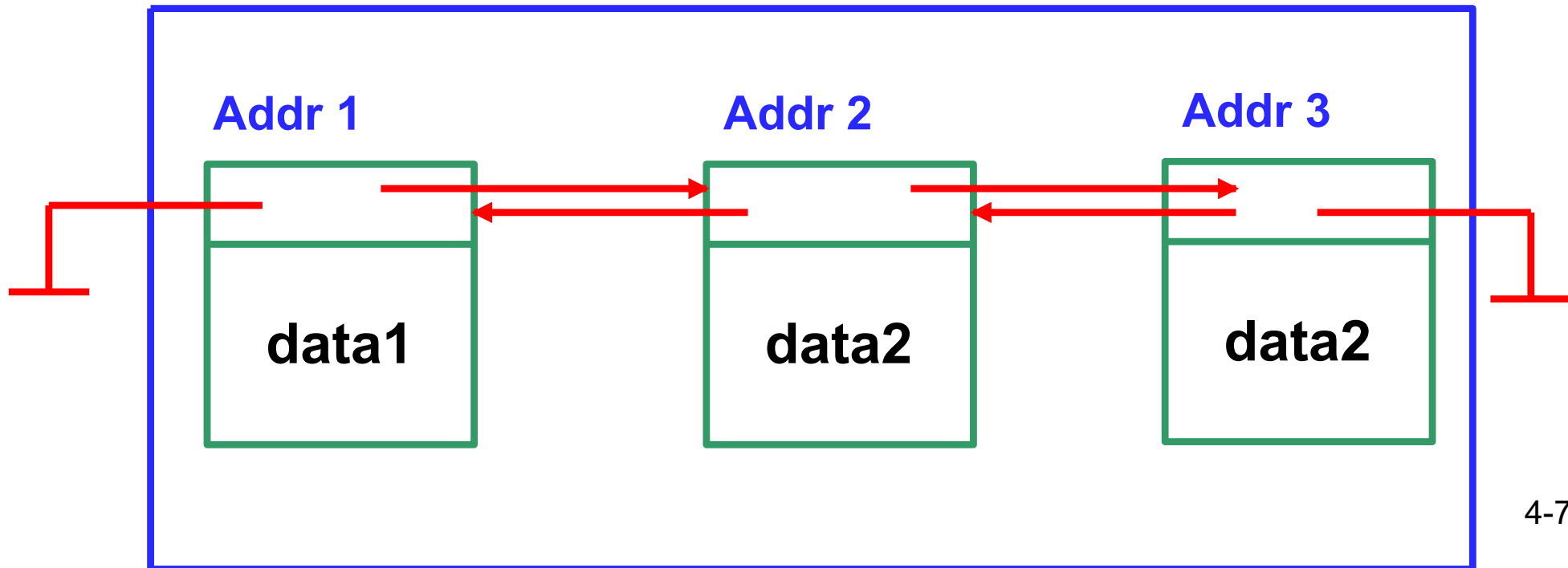
Memory



Linked Data Structures

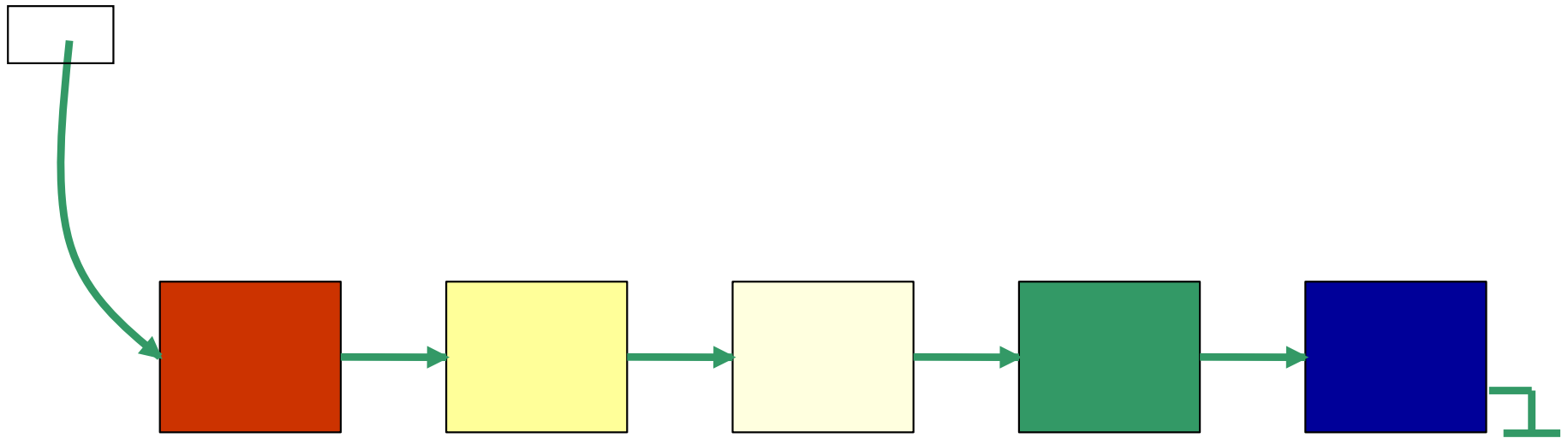
- **Doubly linked list:** each item points to the next item *and* to the previous item

Memory



Conceptual Diagram of a Singly-Linked List

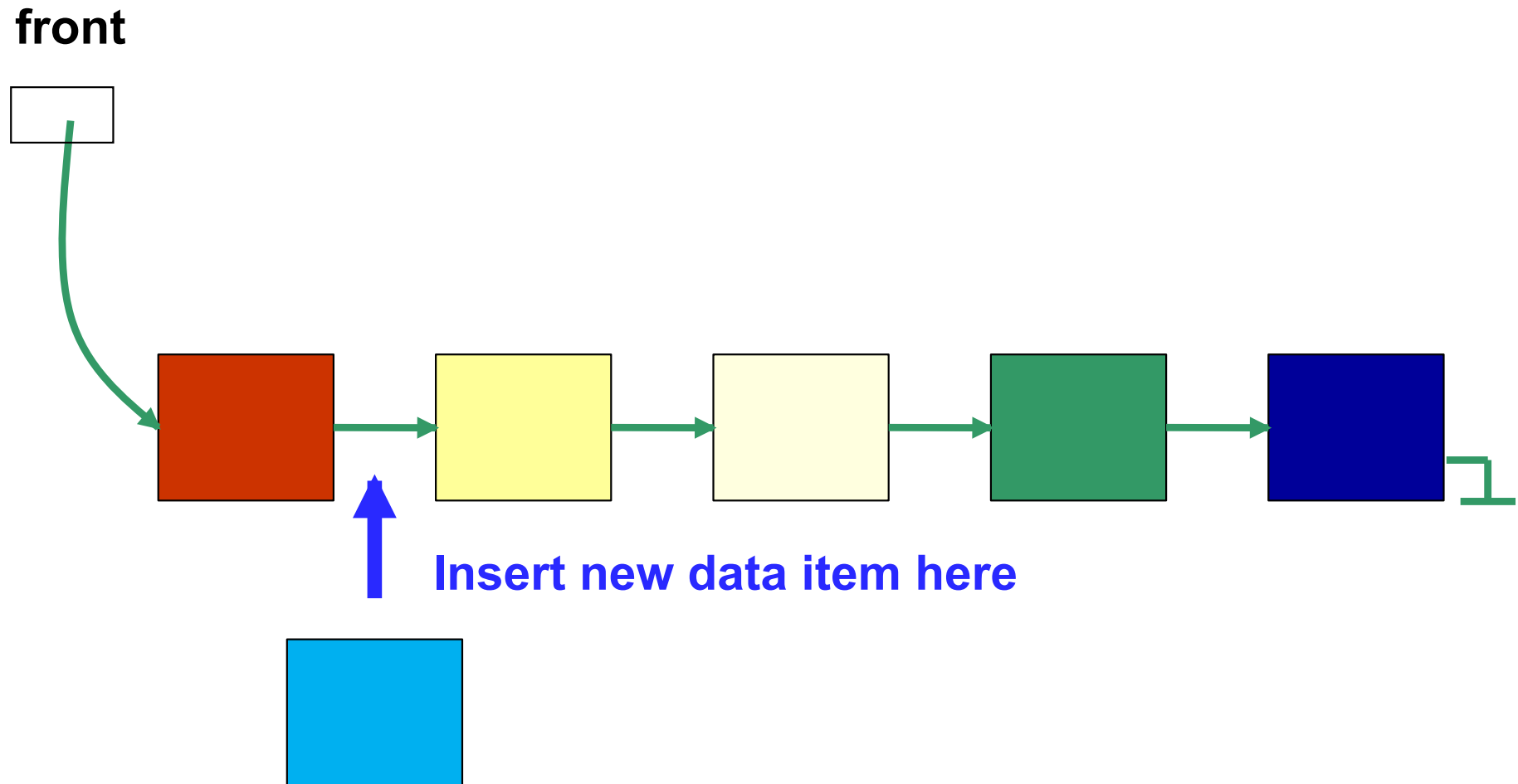
front



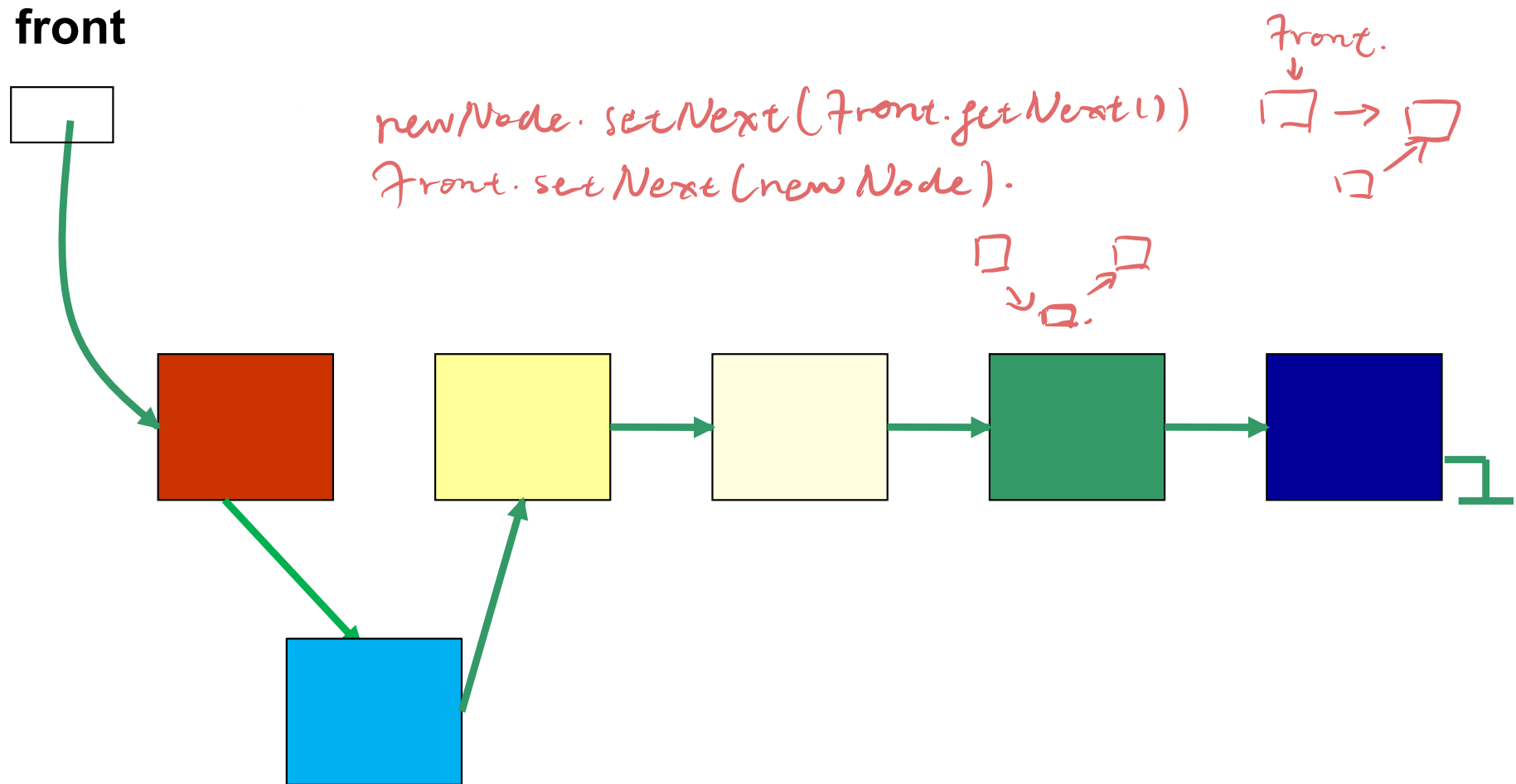
Advantages of Linked Lists

- The items do *not* have to be stored in consecutive memory locations, so we can insert and delete items without shifting data.

Advantages of Linked Lists



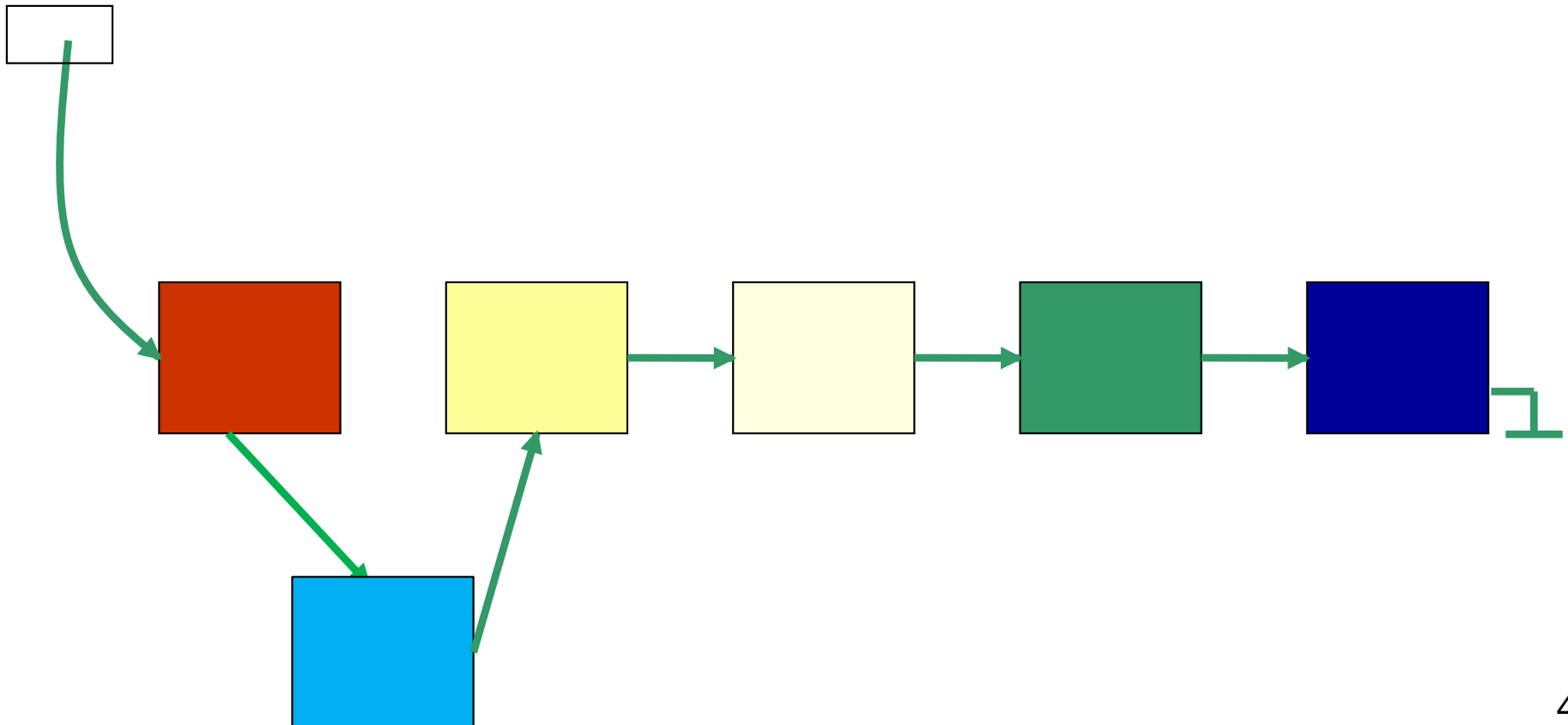
Advantages of Linked Lists



Advantages of Linked Lists

Linked lists can grow and shrink dynamically
(i.e. at run time).

front



Nodes

- A linked list is an sequence of items called *nodes*
- A *node* in a *singly linked list* consists of two fields:
 - A *data* portion
 - A *link (pointer)* to the *next* node in the structure
- The first item (node) in the linked list is accessed via a *front* or *head* pointer

front



Java Class for a Node of a Singly Linked List

```
public class LinearNode<T> {  
    private LinearNode<T> next;  
    private T dataItem;  
    public LinearNode( ) { constructor  
        next = null;  
        dataItem = null;  
    }  
    Overload to set a  
    public LinearNode (T value) { second constructor  
        next = null;  
        dataItem = value;  
    }
```

```
public LinearNode<T> getNext( ) {  
    return next;  
}  
  
public void setNext (LinearNode<T> node) {  
    next = node;  
}  
  
public T getDataItem( ) {  
    return dataItem;  
}  
  
public void setDataItem (T value) {  
    dataItem = value;  
}  
}
```

Example: Create a LinearNode Object

- Example: create a node that contains the integer 7

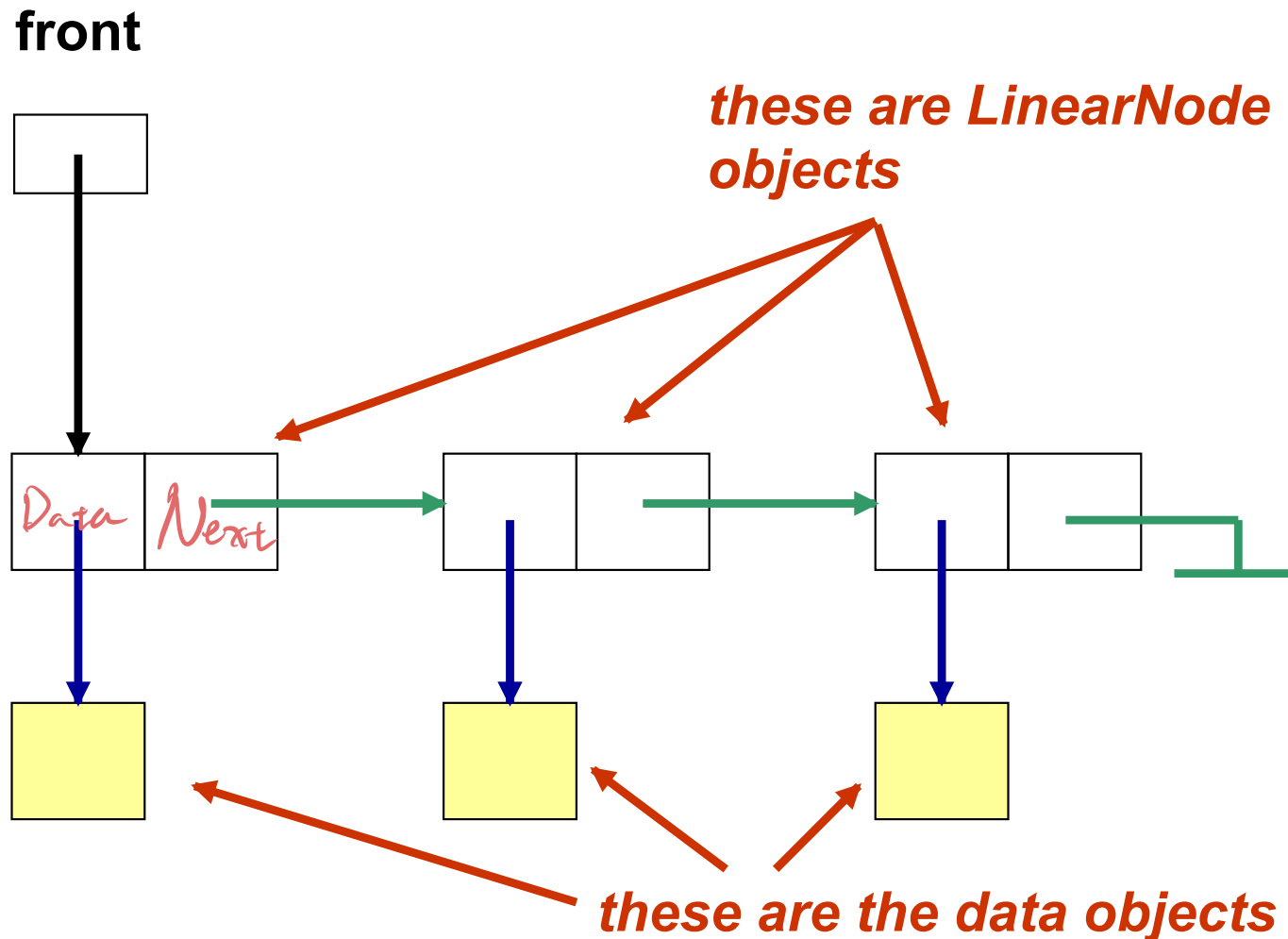
Wrapper class

```
Integer intObj = new Integer(7);  
LinearNode<Integer> inode =  
    new LinearNode<Integer>  
(intObj); second constructor.  
or
```

```
LinearNode<Integer> inode =  
    new LinearNode<Integer> (new  
Integer(7));
```

Wrapper class needed because
a generic type cannot be primitive

Linked List of Node Objects



Java Class for a Singly Linked List

```
public class SinglyLinkedList<T> {  
    private LinearNode<T> front;  
    //pointer to the front  
    public SinglyLinkedList( ) {  
        front = null;  
    }  
}
```

Linked List

Note: we will hereafter refer to a singly linked list just as a “*linked list*”

- **Traversing the linked list**

- How is the first item accessed? *front node*
- The second? *front.getNext();* indicates the first
- The last? *private Node<T> front;*

*while (front.getNext() != Null) {
 rear = front;
}*

- What does the last item point to?
 - We call this the ***null link***

Discussion

- How do we get to an item's successor? *getNext()*.
- How do we get to an item's predecessor? *PrNode.*
while (prNode.getNext().getData().equals(Front.getData()))
- How do we access, say, the 3rd item in the linked list? *count = 0*
- How does this differ from an array?
array [3] = at the position number 3.
linklist => pointer

Linked List Operations

We will now examine linked list operations:

- **Add** an item to the linked list
 - We have 3 situations to consider:
 - insert a node **at the front**
 - insert a node **in the middle**
 - insert a node **at the end**

Inserting a Node at the Front

node



node points to the new node to be inserted, **front** points to the first node of the linked list

front



node



1. Make the new node point to the first node (i.e. the node that **front** points to)

front

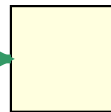


*Node.setNext(front); connect the node to front.
front = Node; update the front.*

node



front



2. Make **front** point to the new node
(i.e the node that **node** points to)

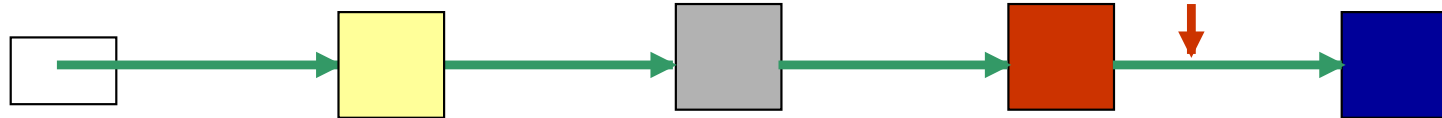
Inserting a Node in the Middle

Let's insert the new node after the *third* node in the linked list

node



front



1. Locate the node *preceding the insertion point*, since it will have to be modified (make **current** point to it)

node



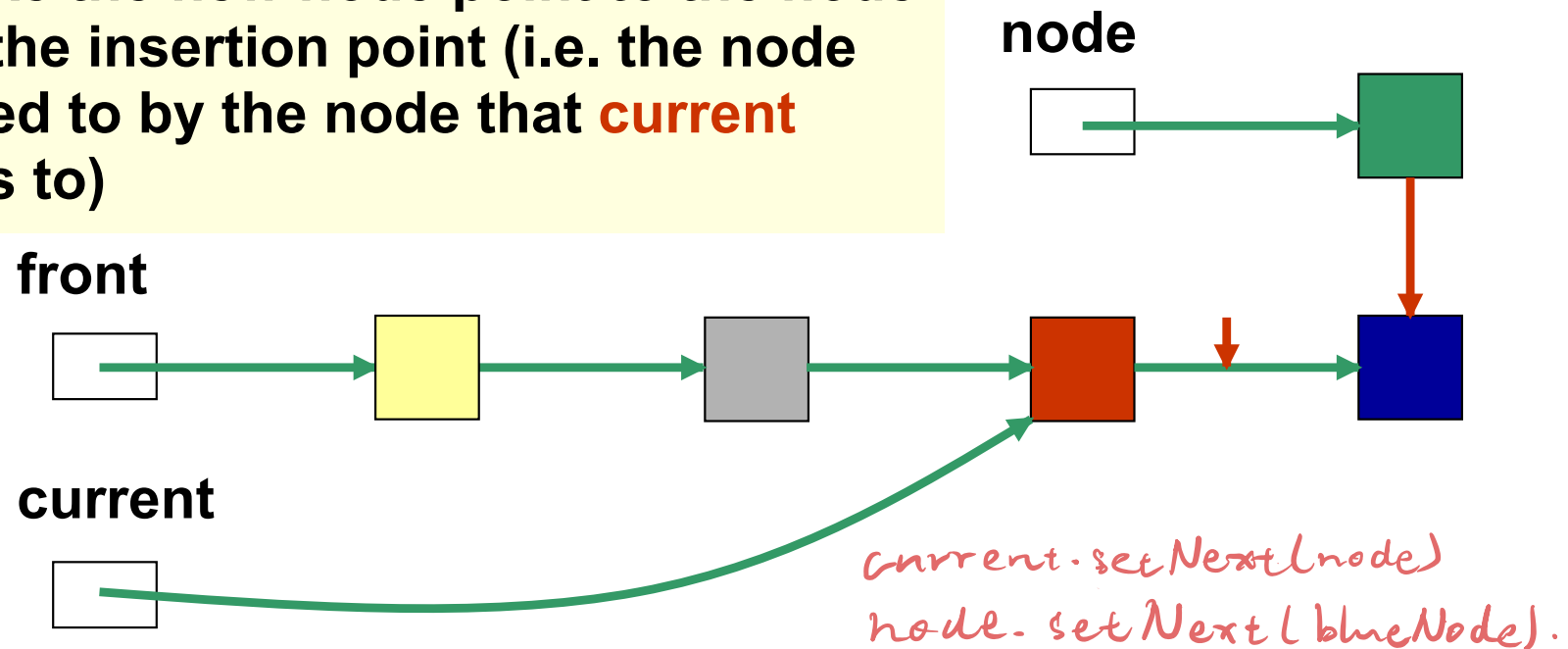
front



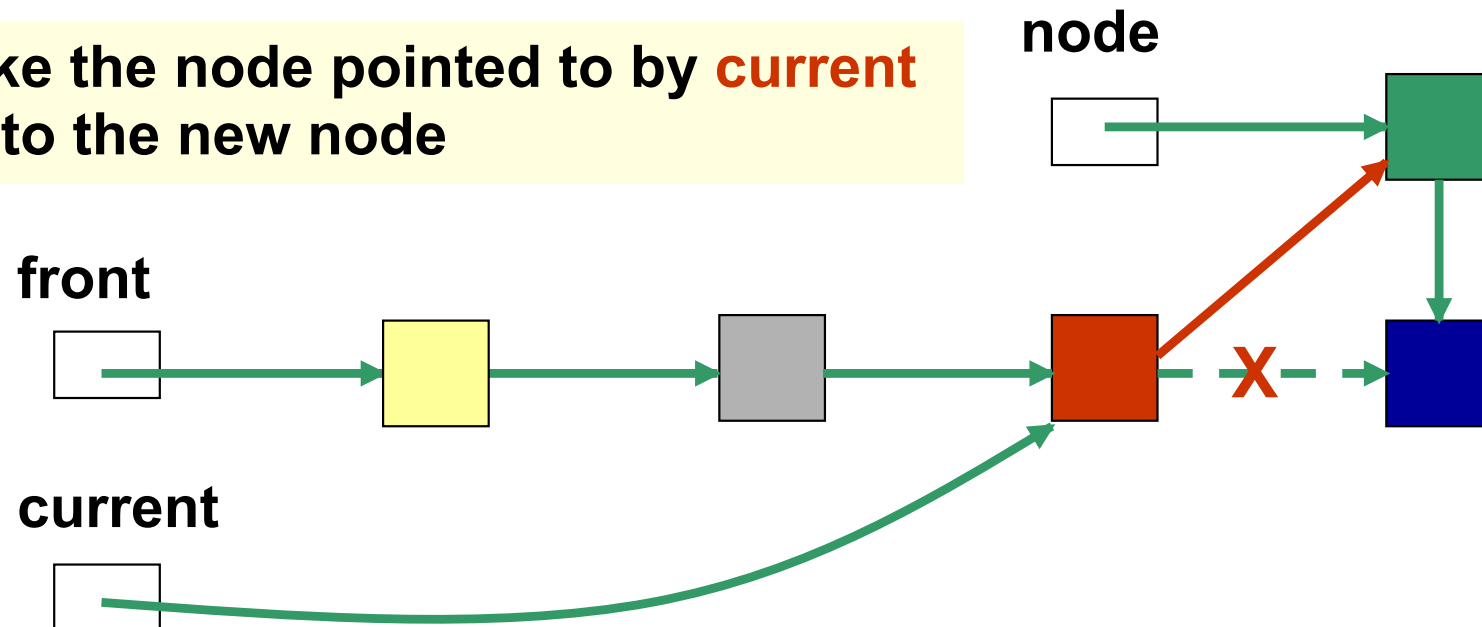
current



2. Make the new node point to the node after the insertion point (i.e. the node pointed to by the node that **current** points to)

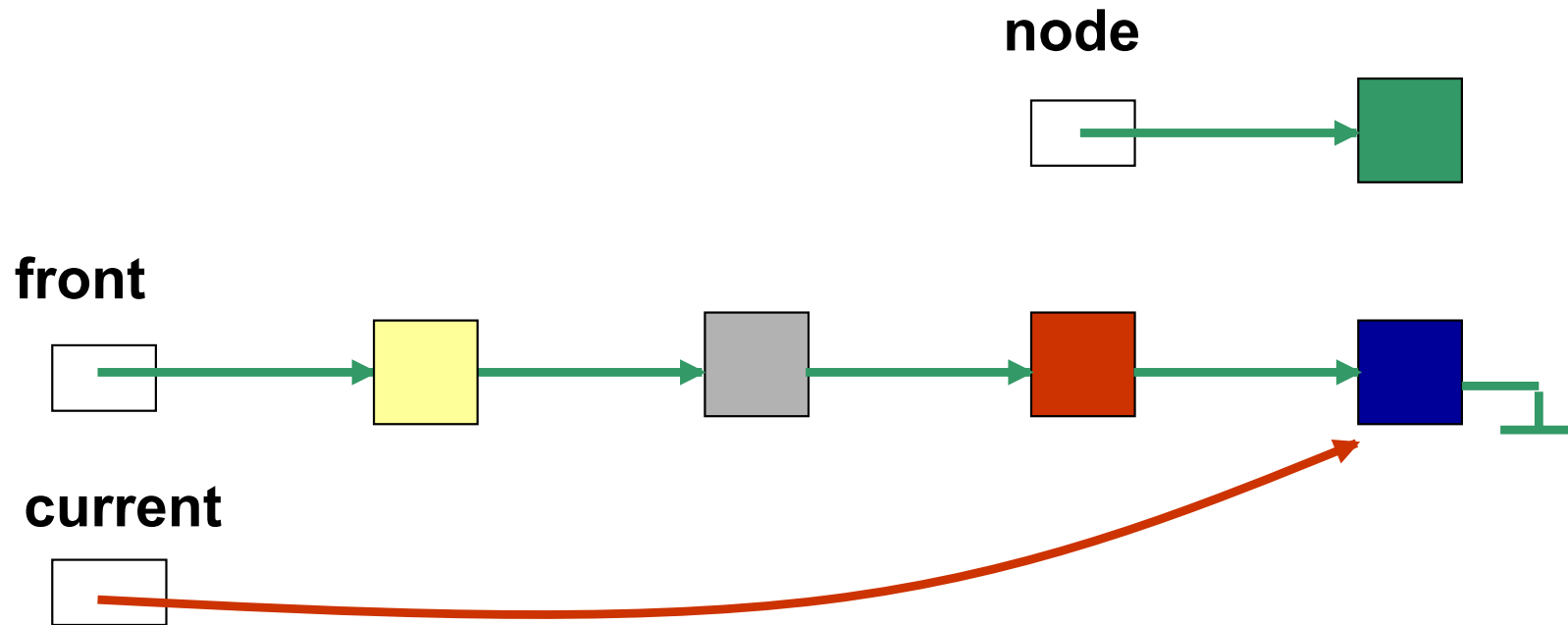


3. Make the node pointed to by **current** point to the new node



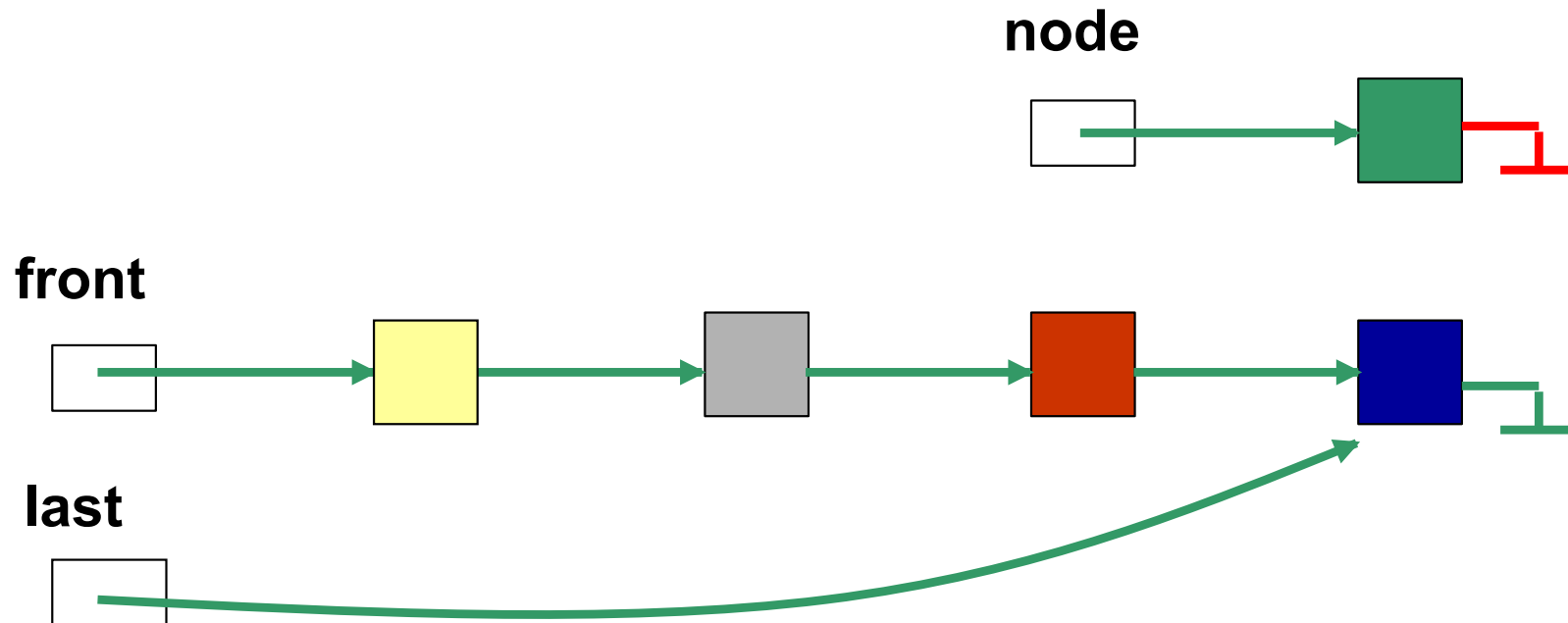
Inserting a Node at the End

1. Locate the last node



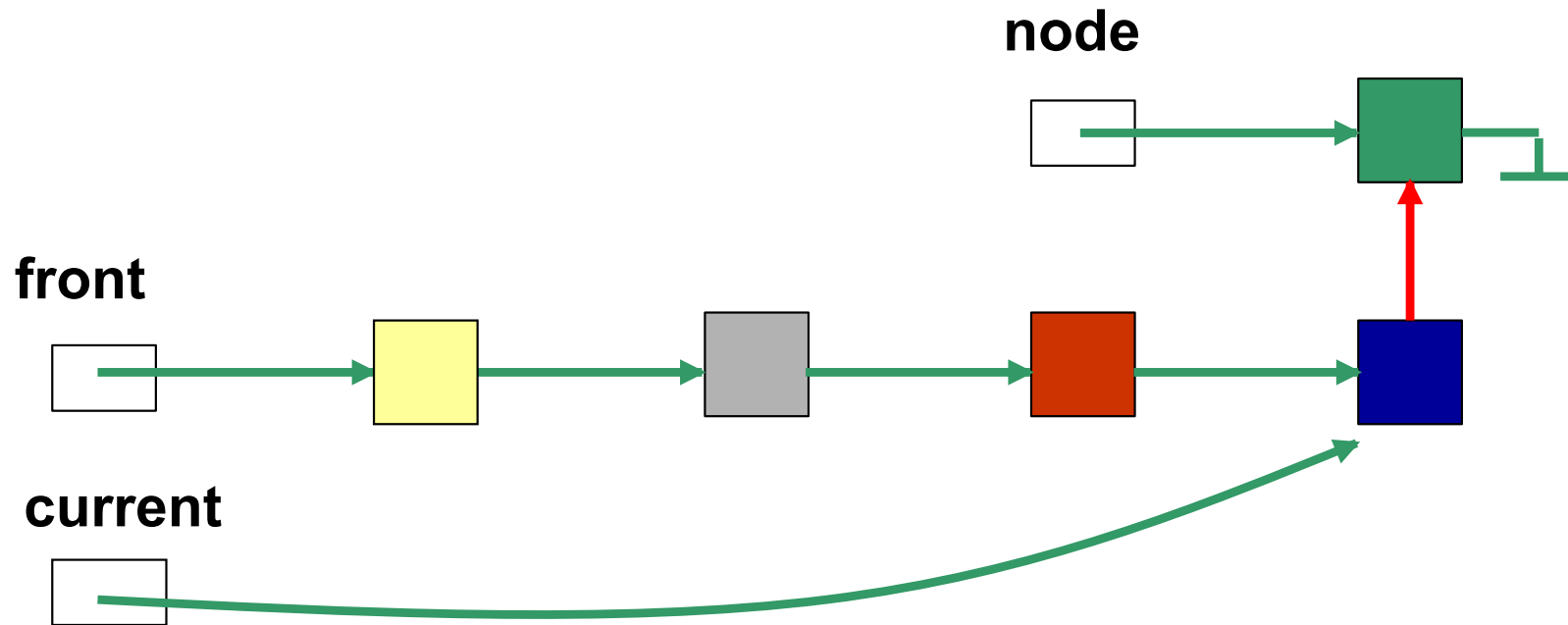
Inserting a Node at the End

2. Make new node point to null



Inserting a Node at the End

3. Make **last** point to new **node**



private node.

node.setNext(null).

current.setNext(node).


Algorithm for inserting a node in a singly linked list

Algorithm insert (*newNode*, *predecessor*)
node .

In: New node to be inserted after *predecessor*.

Out: {Insert *newNode* in linked list after *predecessor*; *newNode* must be inserted at the front of the list if predecessor is null.}

front of the List .

```
if predecessor is null then {  
    newNode.setNext(front)  
    front = newNode  
}  
else {  
    pre  succ  
    node .  
    succ = predecessor.getNext()  
    newNode.setNext(succ)  
    predecessor.setNext(newNode)  
}
```

Java implementation of algorithm for inserting a node in a singly linked list

```
public void insert (LinearNode<T> newNode,  
                  LinearNode<T> predecessor) {  
    if (predecessor == null) {  
        newNode.setNext(front);  
        front = newNode;  
    }  
    else {  
        LinearNode<T> succ = predecessor.getNext();  
        newNode.setNext(succ);  
        predecessor.setNext(newNode);  
    }  
}
```

Linked List Operations

- **Delete** an item from the linked list
 - We have 3 situations to consider:
 - delete the node **at the front**
 - delete an **interior** node
 - delete the **last** node

Deleting the First Node

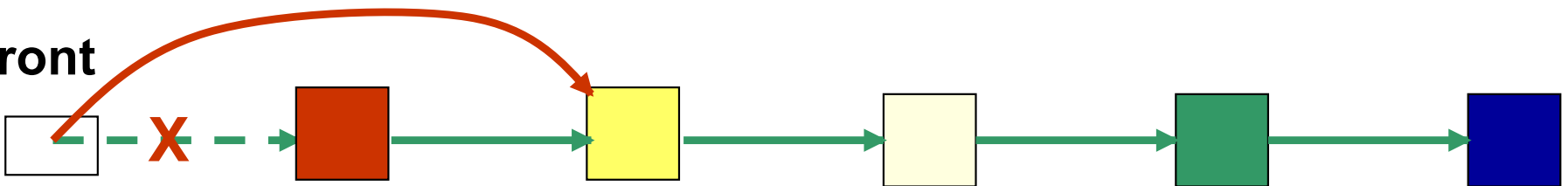
front points to the first node in the linked list, which points to the second node

front



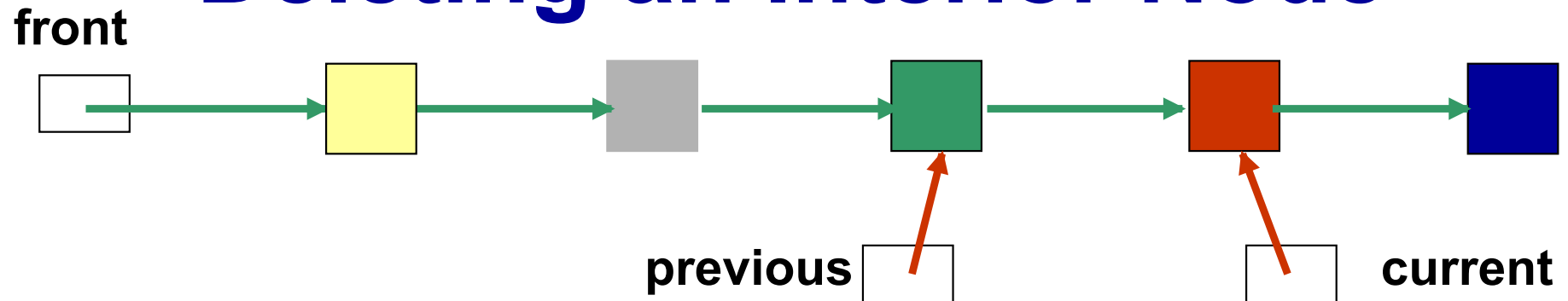
Make **front** point to the second node (i.e. the node pointed to by the first node)

front

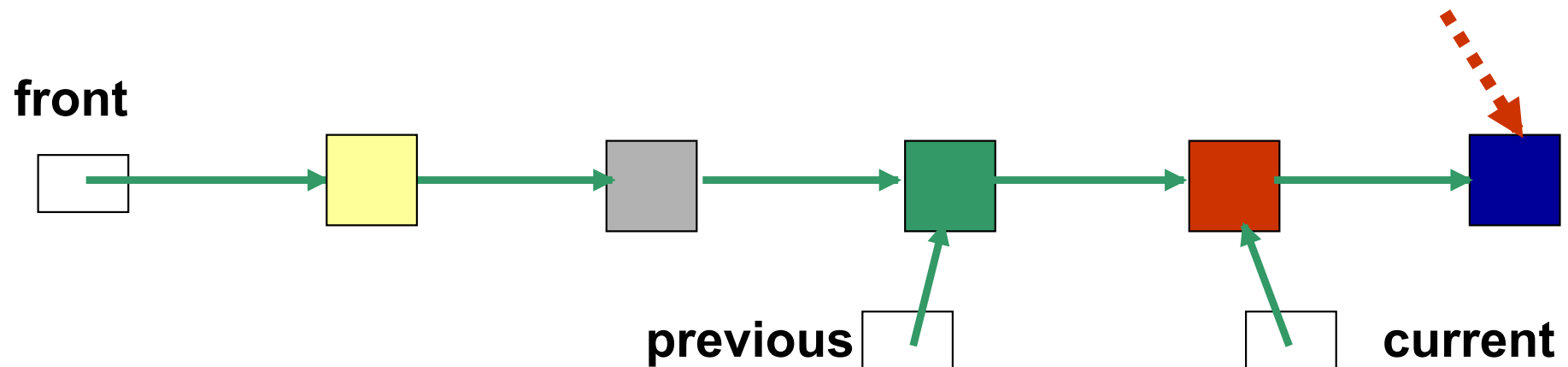


front = front.getNext()

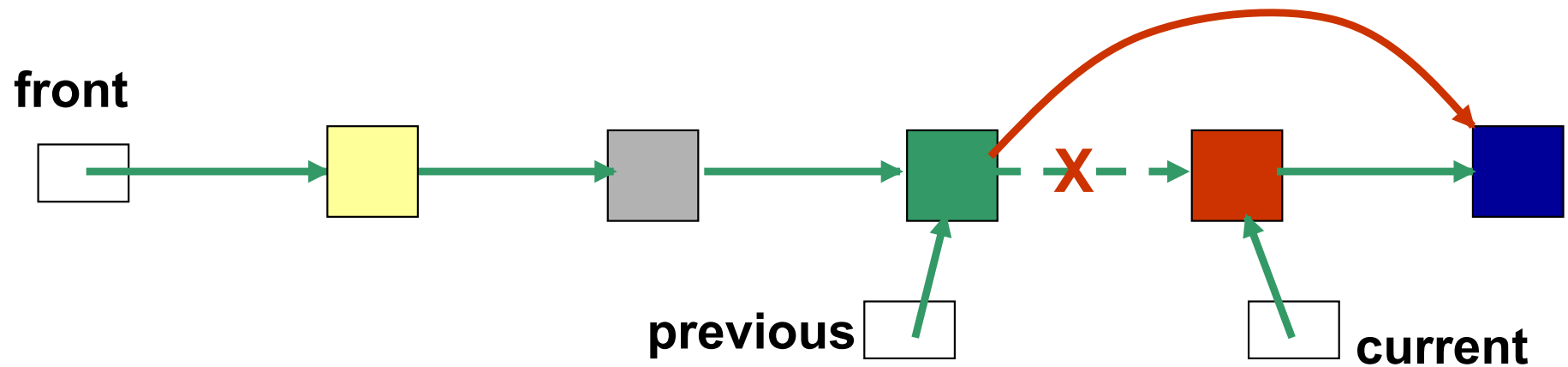
Deleting an Interior Node



1. Traverse the linked list so that **current** points to the node to be deleted and **previous** points to the node prior to the one to be deleted



2. We need to get at the node *following the one to be deleted* (i.e. the node pointed to by the node that **current** points to)

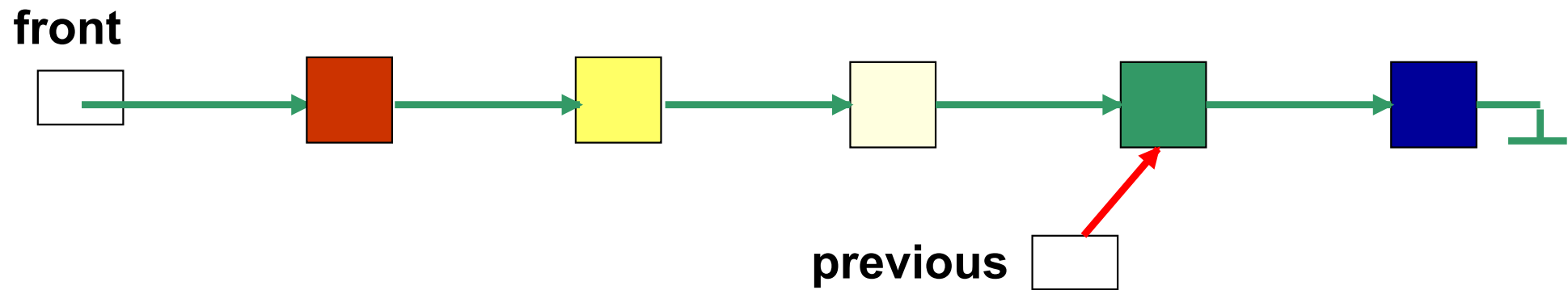


3. Make the node that **previous** points to, point to the node following the one to be deleted

*previous.setNext(current.getNext());
current.setNext(Null);
not Necessary.*

Deleting the Last Node

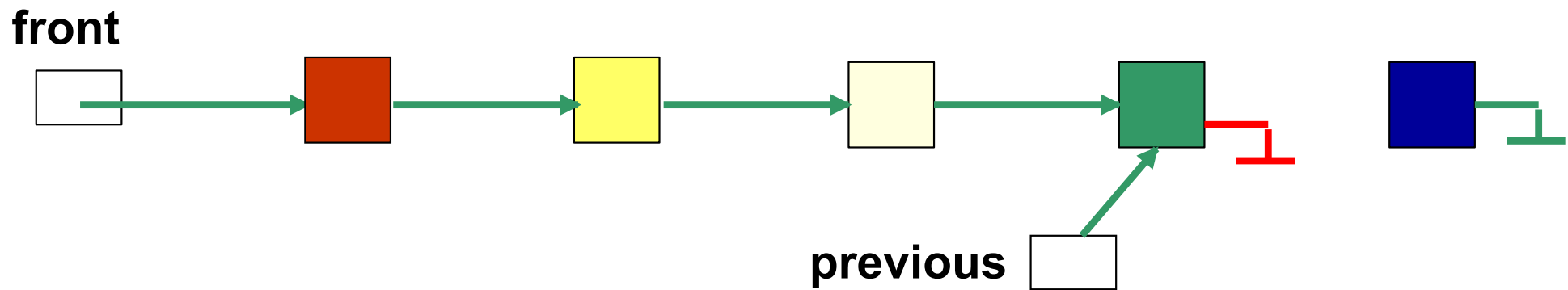
1. Find the **previous** to the last node in the linked list



previous.setNext(null);

Deleting the Last Node

1. Make **previous** point to null



Algorithm delete (*nodeToDelete*)

In: node to delete

Out: *true* if the node was deleted, *false* otherwise

current = front

predecessor = null

while (current \neq null) **and** (current \neq *nodeToDelete*) **do** {

 predecessor = current

 current = current.getNext()

}

if current is null **then return** *false*

else {

if predecessor \neq null **then**

 predecessor.setNext(current.getNext())

else front = front.getNext()

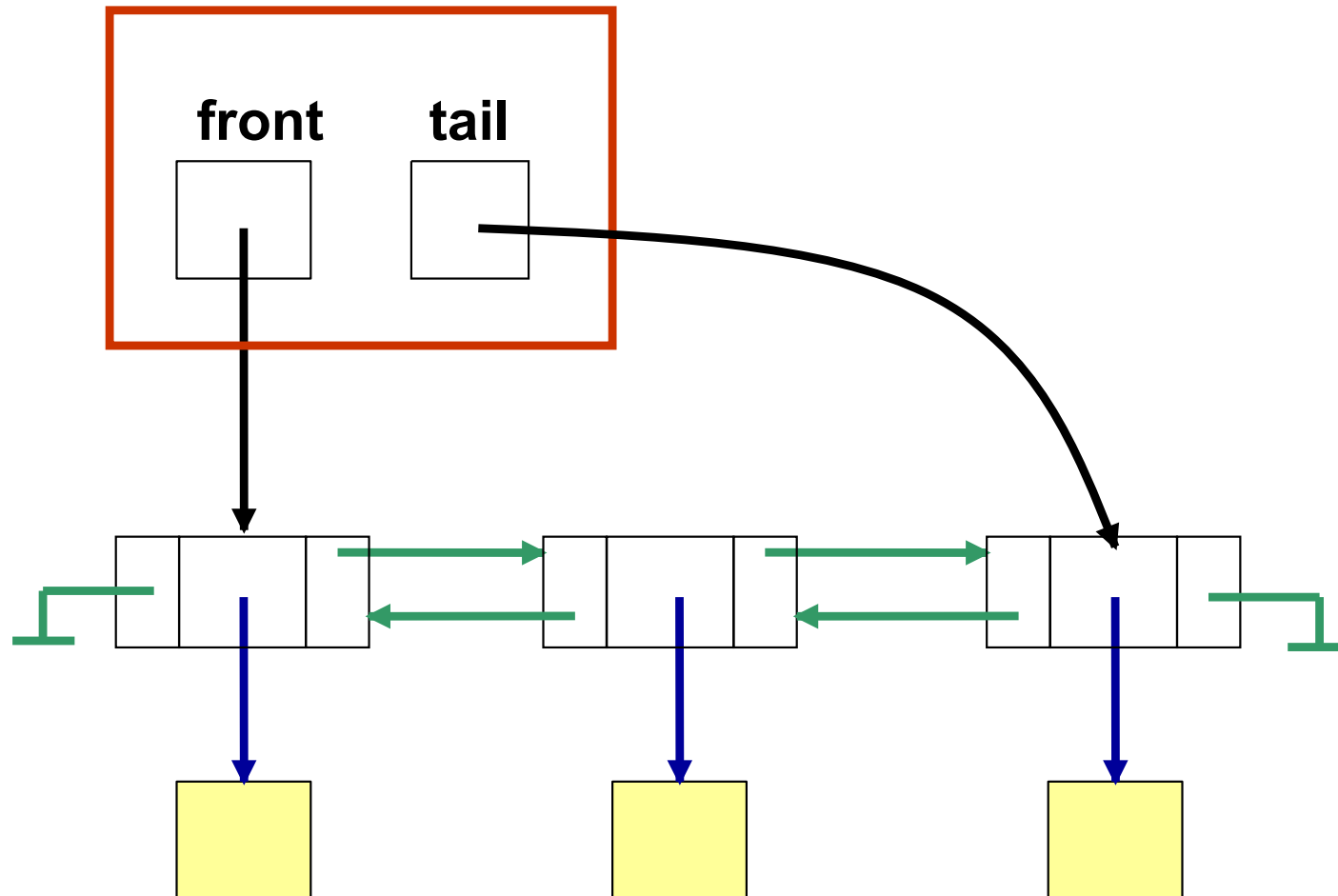
return *true*

}

Java Implementation of Above Algorithm

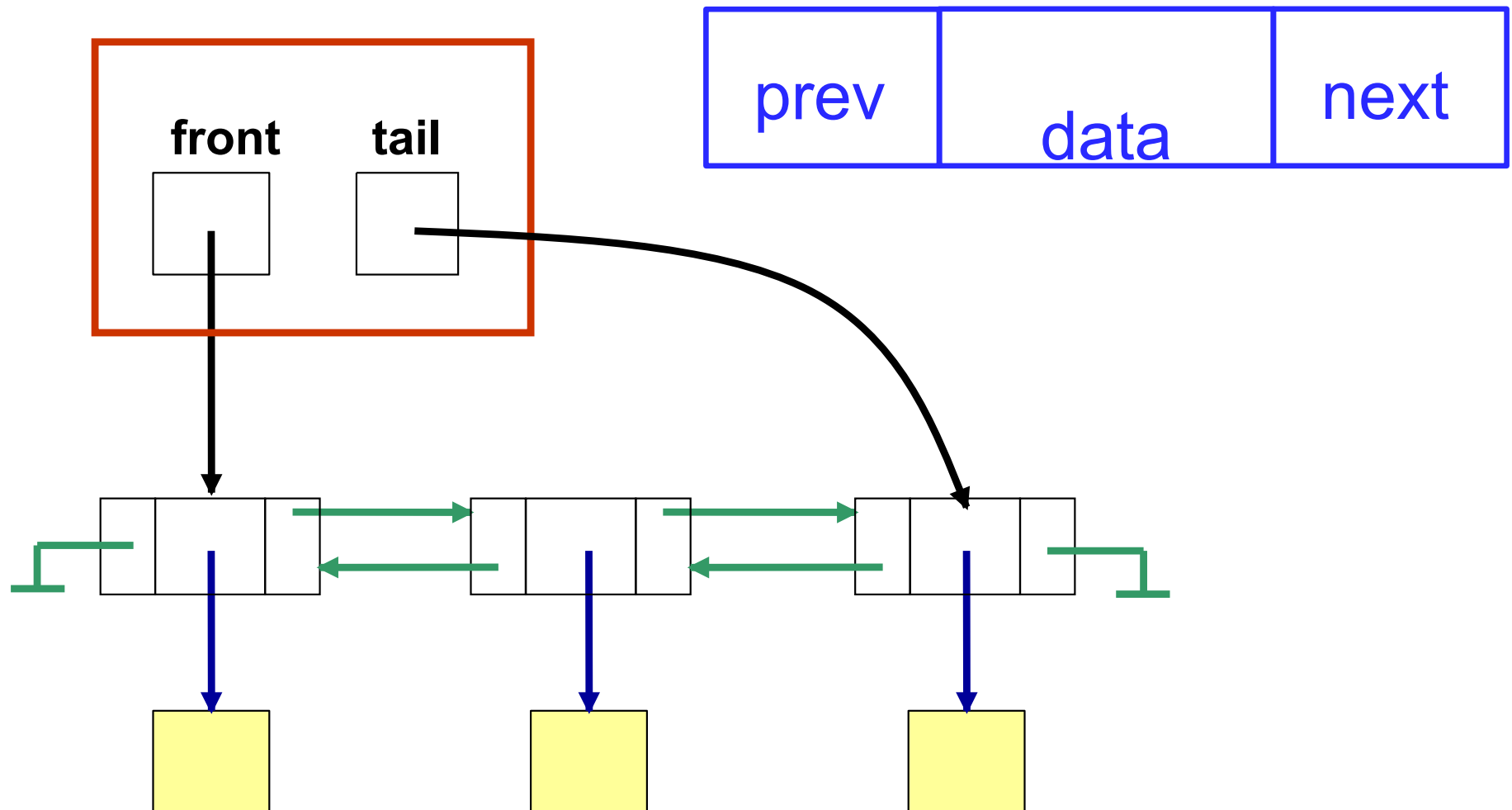
```
public boolean delete (LinearNode<T> nodeToDelete) {  
    LinearNode<T> current, predecessor;  
    current = front;  
    predecessor = null; end of the list.  
    while ((current != null) && (current != nodeToDelete)) { the case to hit.  
        predecessor = current;  
        current = current.getNext();  
    }  
    if (current == null) return false; No node found.  
    else {  
        if (predecessor != null) not the first node.  
            predecessor.setNext(current.getNext());  
        else front = front.getNext();  
        return true; first node.  
    }  
}
```

Doubly Linked List



Doubly Linked List

Node object



Java Class for a Node of a Doubly Linked List

```
public class LinearNodeDLL<T> {  
    private LinearNodeDLL<T> next;  
    private LinearNodeDLL<T> prev;  
    private T dataItem;  
  
    public LinearNodeDLL( ) {  
        next = null;  
        prev = null;  
        dataItem = null;  
    }  
  
    public LinearNodeDLL (T value) {  
        next = null;  
        prev = null;  
        dataItem = value;  
    }  
}
```

```

public LinearNodeDLL<T> getNext( ) {
    return next;
}
public void setNext (LinearNodeDLL<T> node) {
    next = node;
}
public LinearNodeDLL<T> getPrev( ) {
    return prev;
}
public void setPrev (LinearNodeDLL<T> node) {
    prev = node;
}
public T getDataItem( ) {
    return dataItem;
}
public void setDataItem (T value) {
    dataItem = value;
}
}

```

Java Class for a Doubly Linked List

```
public class DoublyLinkedList<T> {  
    private LinearNodeDLL<T> front;  
    private LinearNodeDLL<T> tail;  
  
    public DoublyLinkedList( ) {  
        front = null;  
        tail = null;  
    }  
    ...  
}
```

Write algorithms to add a new node to a doubly linked list and to remove a node from a doubly linked list.

```
public boolean remove(Node node) {  
    Node current = front;  
    while (current.getNext() != Null && current.getData().equals(  
        node.getData())) {  
        current = current.getNext();  
    }  
    if (current.getNext() == Null) { return false; }  
    else {  
        current.getPrev().setNext(current.getNext());  
        current.getNext().setPrev(current.getPrev());  
        return true;  
    }  
}
```