Basic Data Structures: Linked Lists

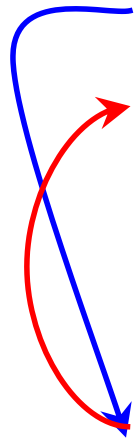
Lecture 02.03 by Marina Barsky

Linked lists: motivation

Busy day: many tasks,
1 sheet of paper

1	To do:
2	Dry cleaning
3	Bank
4	Lunch with Bob
5	Haircut

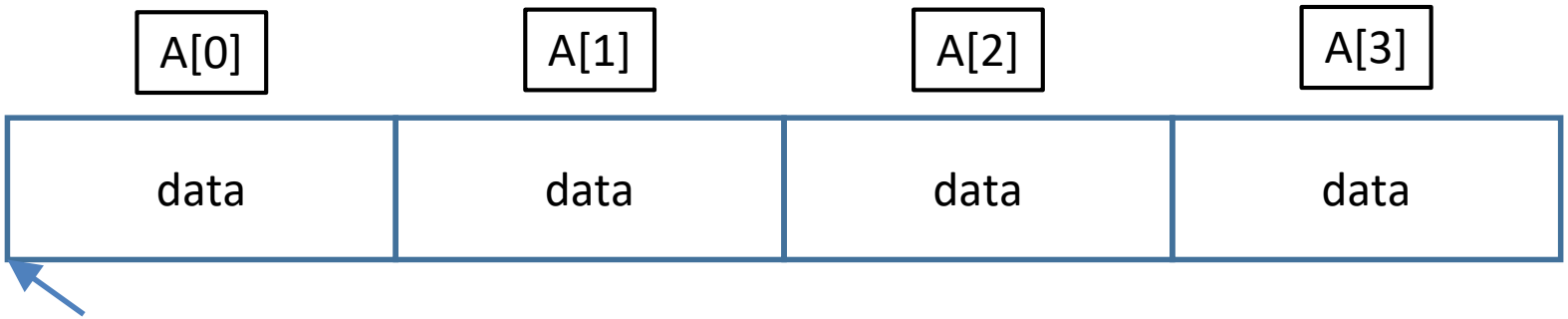
Invention:
linked list



1	To do:
2	Dry cleaning
3	Bank
4	Lunch with Bob
5	Haircut
6	Stamps
	2→6→3→4→5

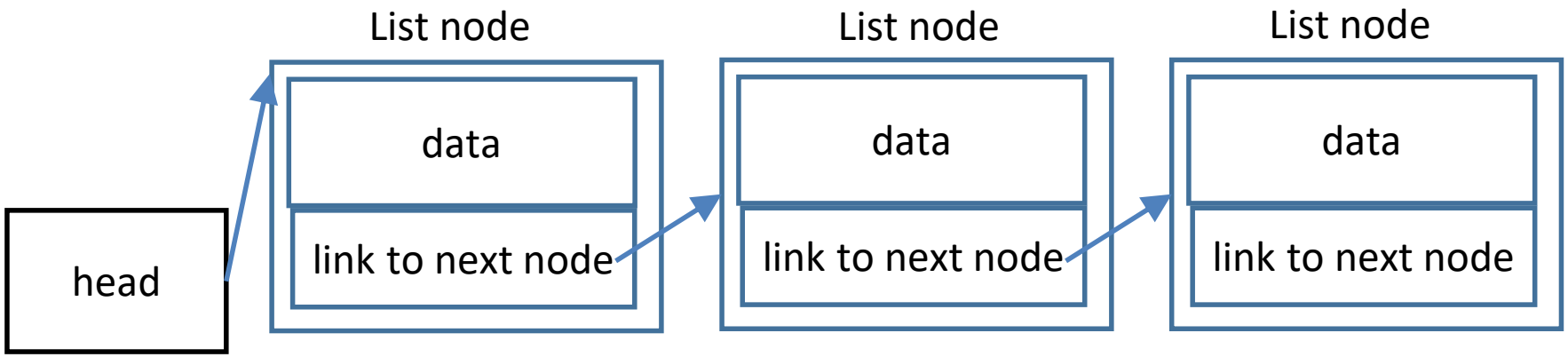
- Main motivation: flexible storage
- We want to be able to add an element in the middle of the list without moving all the data

Two ways for storing a sequence of values



Array

Pointer to (address of) the first element of an array: A [0]

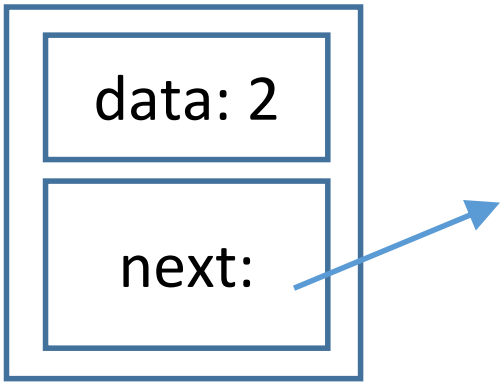


Linked list

Pointer to the first node

Linked list - recursive data structure

- Main element of the linked list: node
- Each Node contains a link to the next Node



node_addr

```
typedef struct node {
    int data;
    struct node * next;
} Node;
```

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

Linked List: main operations

`add(data)`	Add element at the end of the list
`add_in_front (data)`	
`add_after(Node prev, data)`	
`add_before(Node next, data)`	
`remove(Node node)`	
`find(data)`	
`size()`	

Linked List: main operations

`add(data)`	Add element at the end of the list
`add_in_front (data)`	Add element in front of the list
`add_after(Node prev, data)`	
`add_before(Node next, data)`	
`remove(Node node)`	
`find(data)`	
`size()`	

Linked List: main operations

`add(data)`	Add element at the end of the list
`add_in_front (data)`	Add element in front of the list
`add_after(Node prev, data)`	Insert element after a given node
`add_before(Node next, data)`	Insert element before a given node
`remove(Node node)`	
`find(data)`	
`size()`	

Linked List: main operations

`add(data)`	Add element at the end of the list
`add_in_front (data)`	Add element in front of the list
`add_after(Node prev, data)`	Insert element after a given node
`add_before(Node next, data)`	Insert element before a given node
`remove(Node node)`	Remove a given node
`find(data)`	
`size()`	

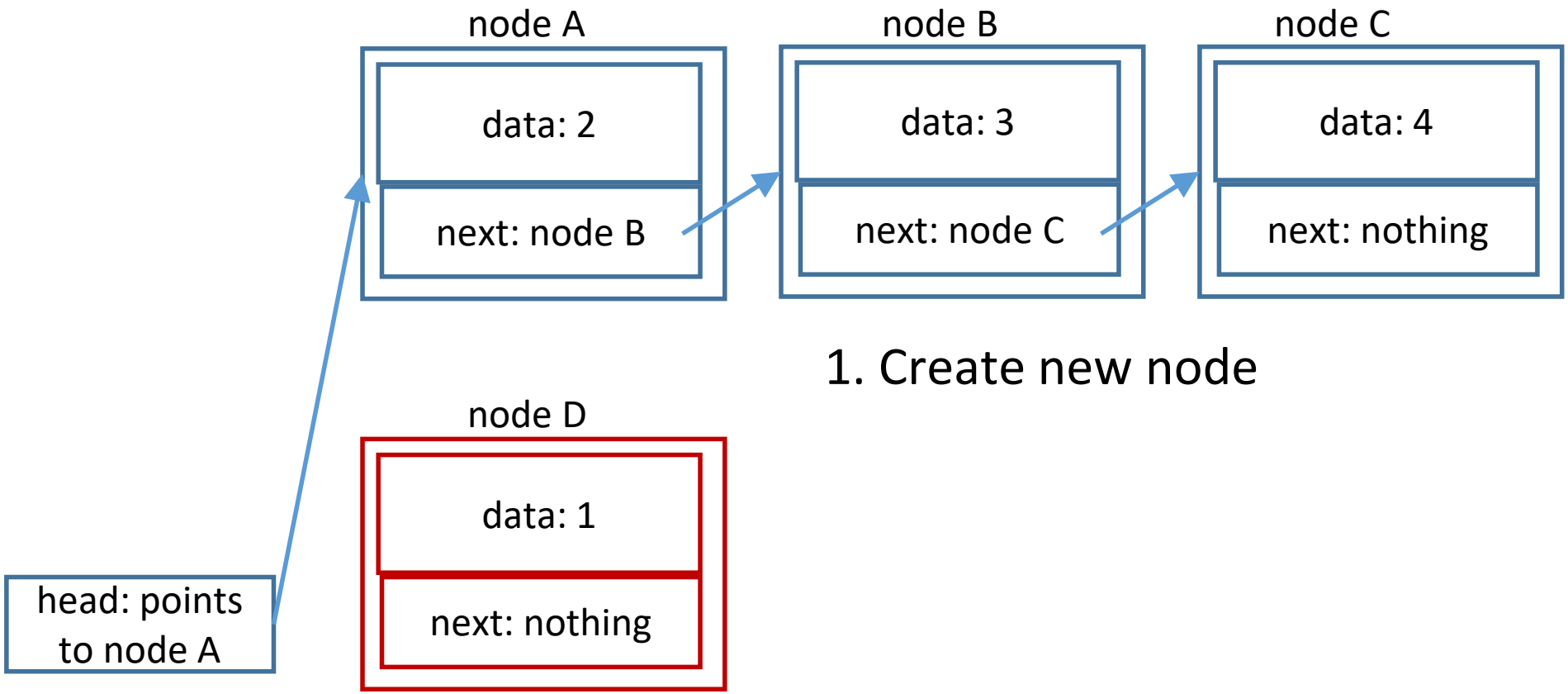
Linked List: main operations

`add(data)`	Add element at the end of the list
`add_in_front (data)`	Add element in front of the list
`add_after(Node prev, data)`	Insert element after a given node
`add_before(Node next, data)`	Insert element before a given node
`remove(Node node)`	Remove a given node
`find(data)`	Find a node which contains particular data
`size()`	

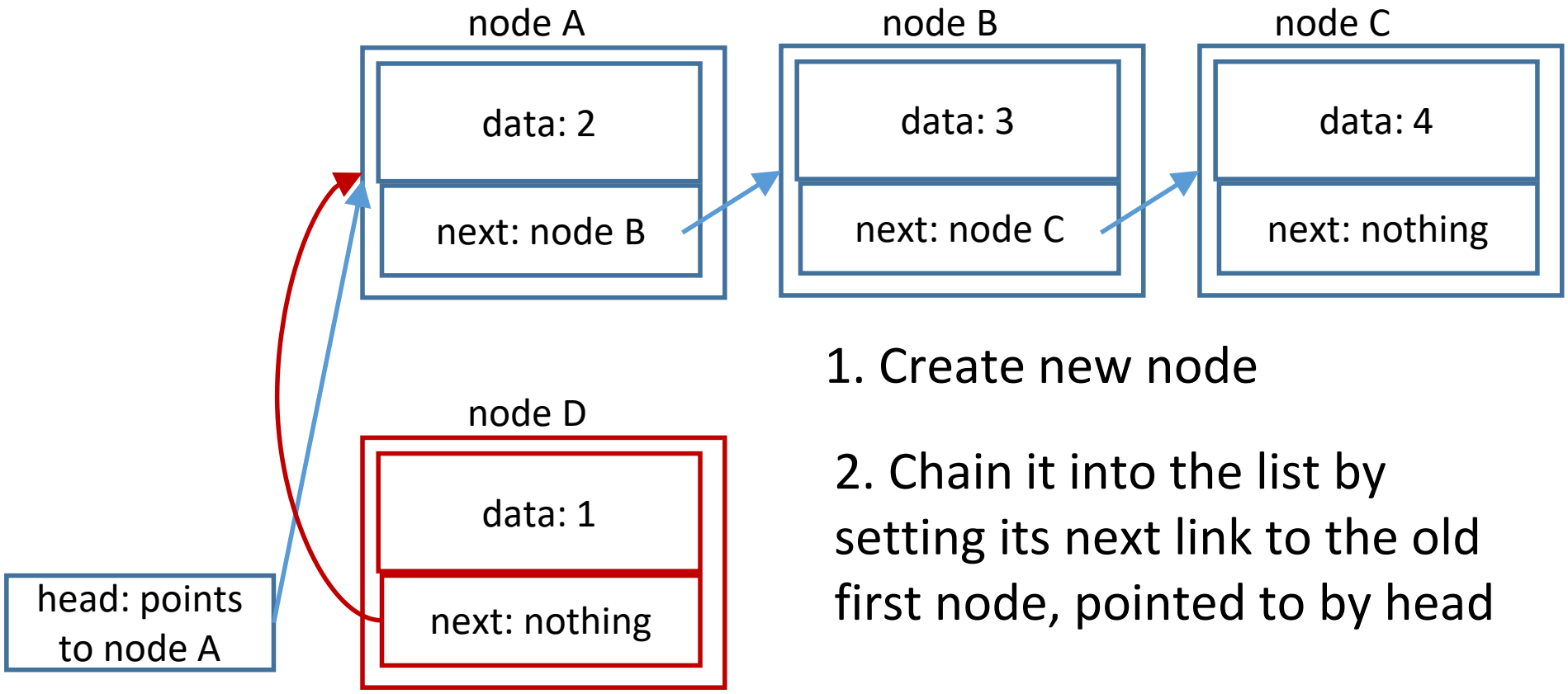
Linked List: main operations

`add(data)`	Add element at the end of the list
`add_in_front (data)`	Add element in front of the list
`add_after(Node prev, data)`	Insert element after a given node
`add_before(Node next, data)`	Insert element before a given node
`remove(Node node)`	Remove a given node
`find(data)`	Find a node which contains particular data
`size()`	Return total count of list elements

Add in front


node A
data: 2
next: node B
node B
data: 3
next: node C
node C
data: 4
next: nothing
1. Create new node
node D
data: 1
next: nothing
head: points to node A

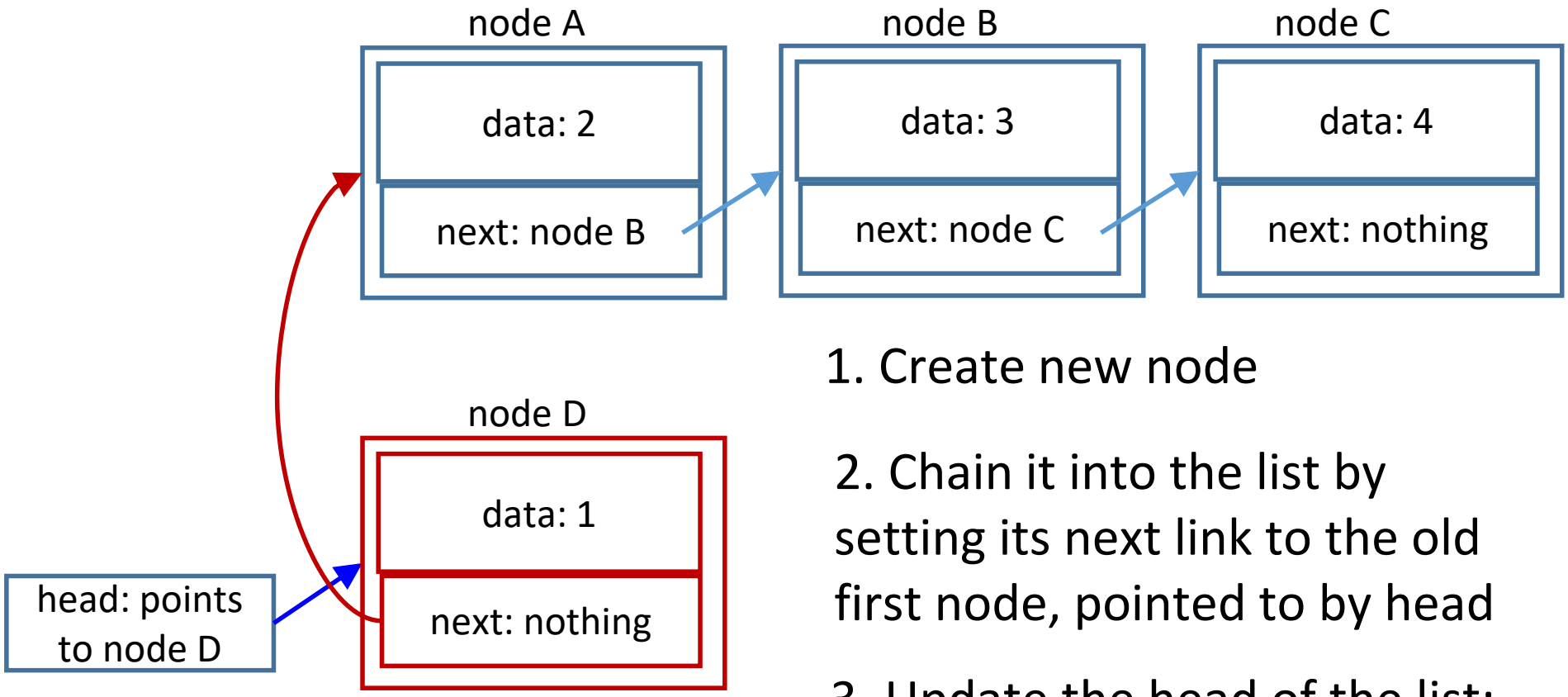
Add in front



1. Create new node

2. Chain it into the list by setting its next link to the old first node, pointed to by head

Add in front



1. Create new node

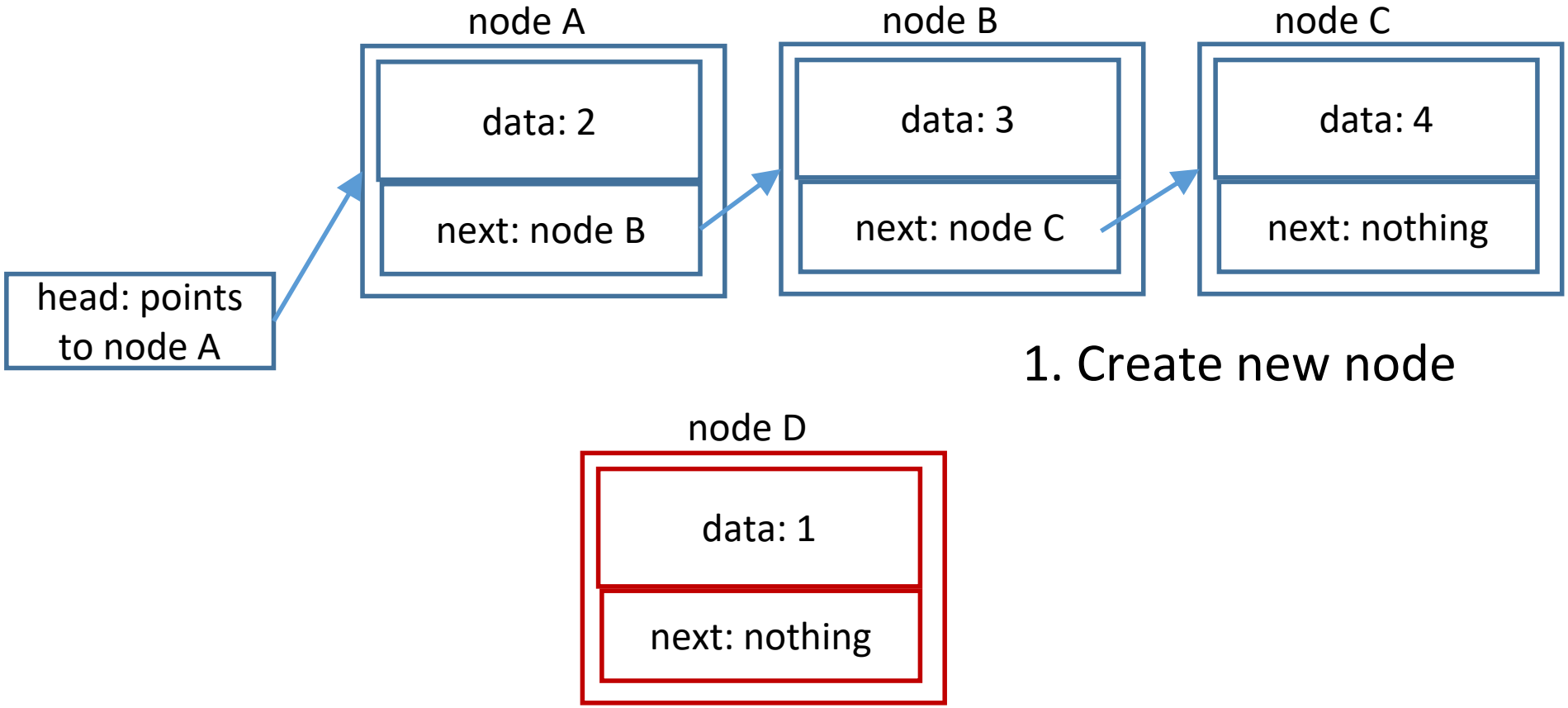
2. Chain it into the list by setting its next link to the old first node, pointed to by head

3. Update the head of the list: it is now pointing to node D

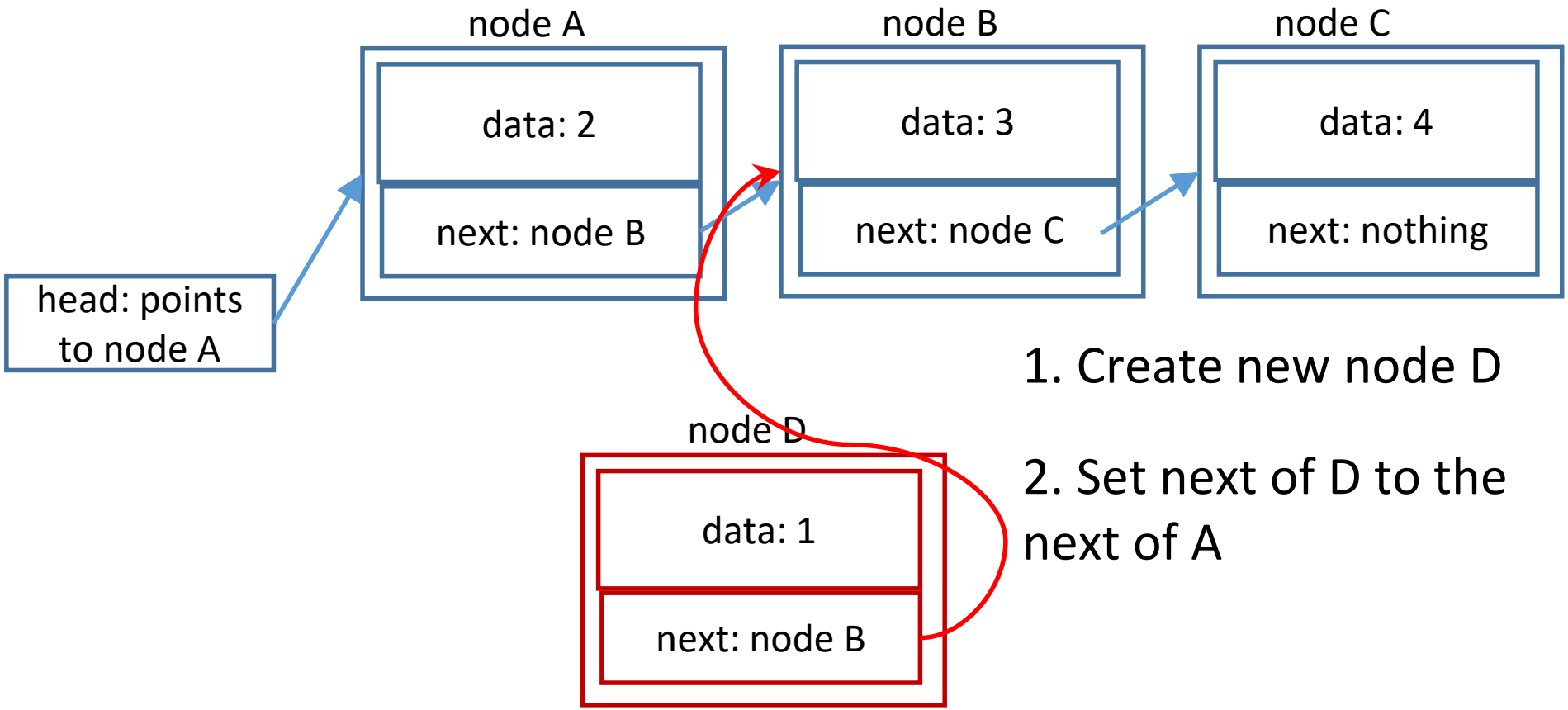
Linked List: main operations

`add(data)`	
`add_in_front (data)`	O(1)
`add_after(Node prev, data)`	
`add_before(Node next, data)`	
`remove(Node node)`	
`find(data)`	
`size()`	

Add after A


node A
data: 2
next: node B
node B
data: 3
next: node C
node C
data: 4
next: nothing
head: points to node A
1. Create new node
node D
data: 1
next: nothing

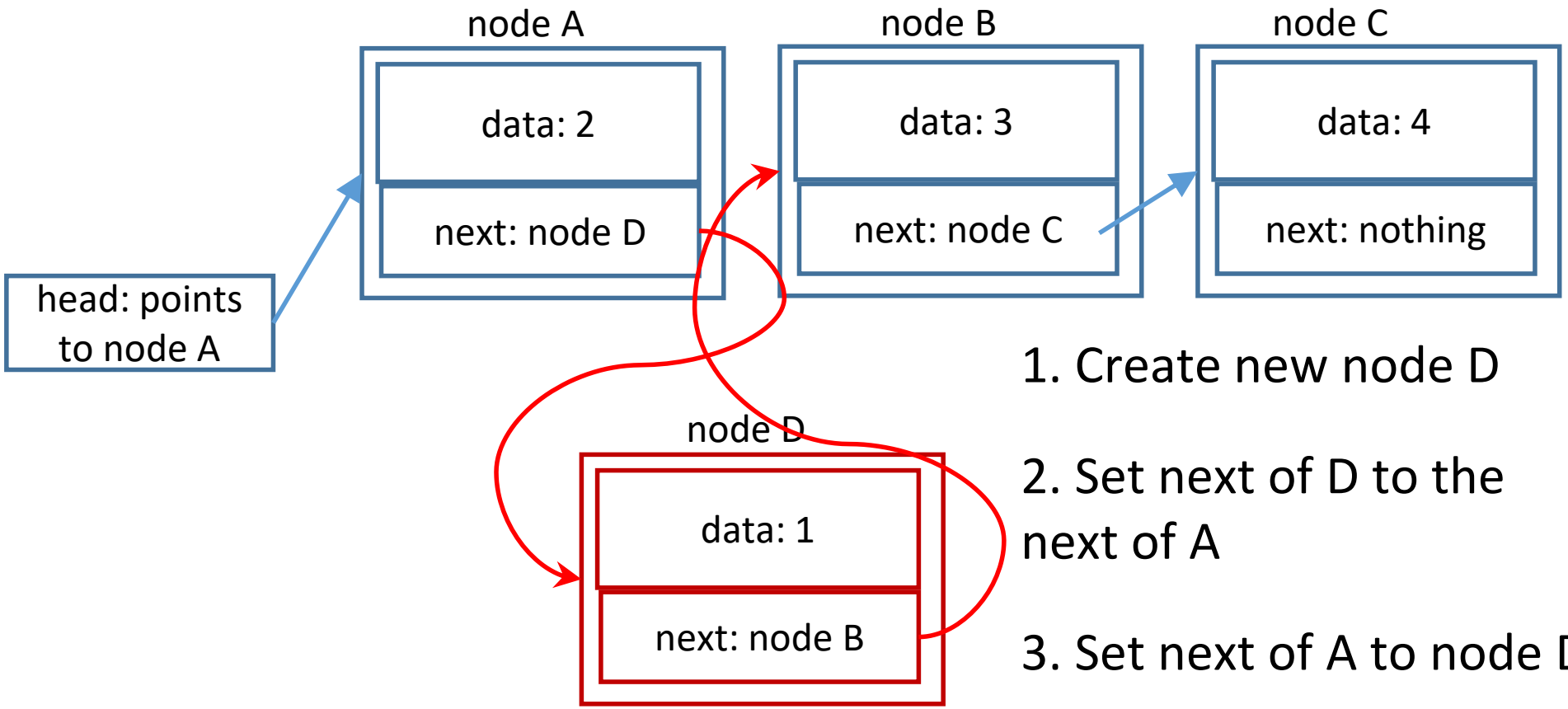
Add after A



1. Create new node D

2. Set next of D to the next of A

Add after A

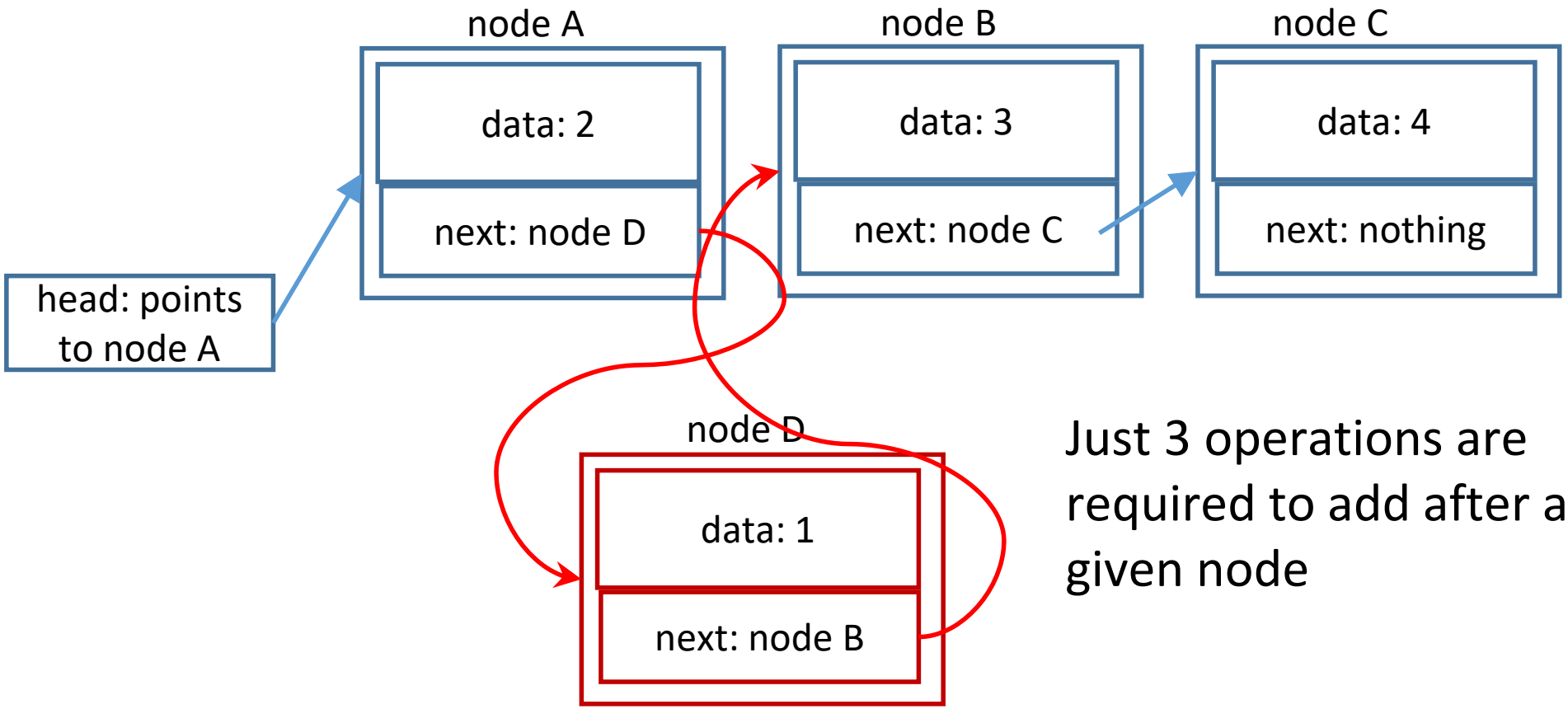


1. Create new node D

2. Set next of D to the next of A

3. Set next of A to node D

Add after A

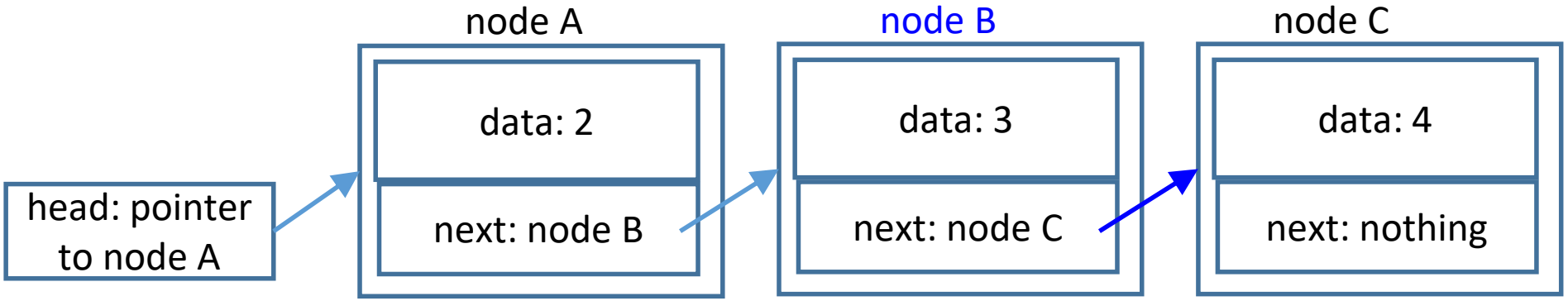


Just 3 operations are required to add after a given node

Linked List: main operations

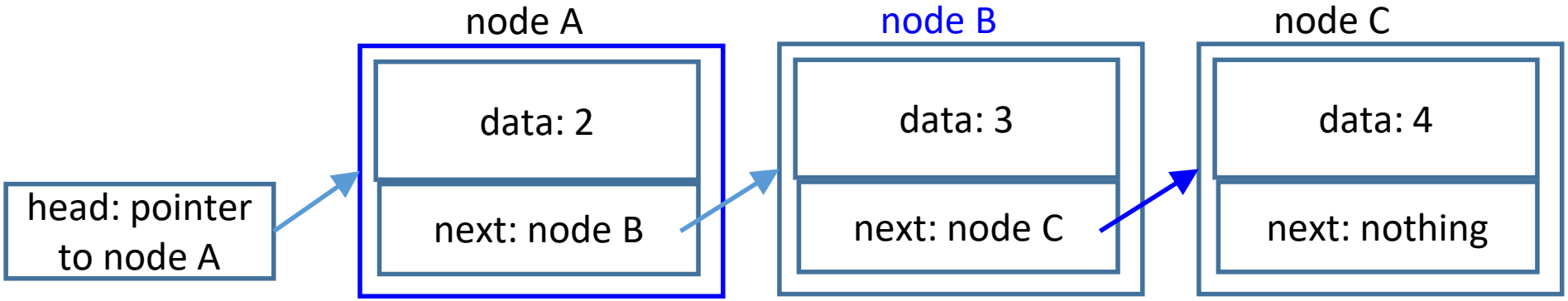
`add(data)`	
`add_in_front (data)`	O(1)
`add_after(Node prev, data)`	O(1)
`add_before(Node next, data)`	
`remove(Node node)`	
`find(data)`	
`size()`	

Remove node B



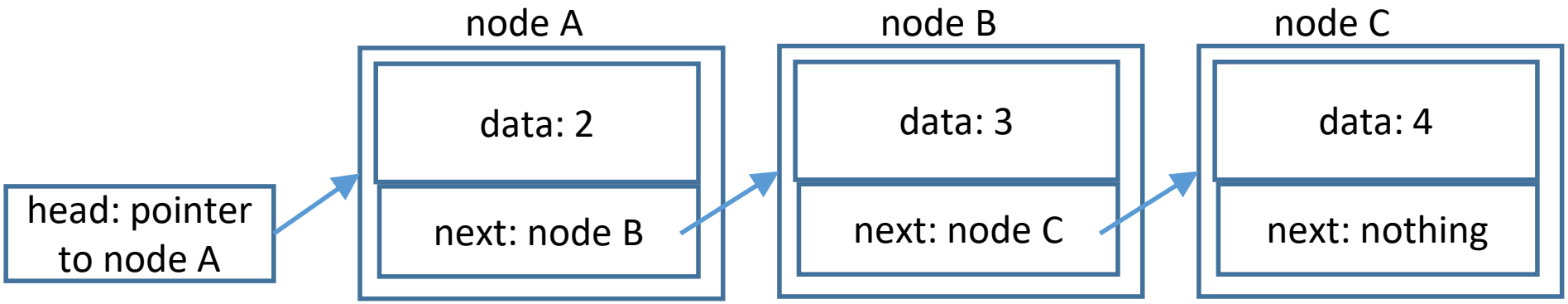
We cannot remove node B, because we do not know which node has B as its next - we have no access to the **previous** of node B!

Remove node B



1. We cannot remove node B, because we do not know which node has B as its next - we have no access to the previous of node B!
2. In order to remove node B, we need to traverse the list and look which node has B as its next
3. After we found the previous node, then O(1) ro rewire the list

Traversing the list to find the node before B

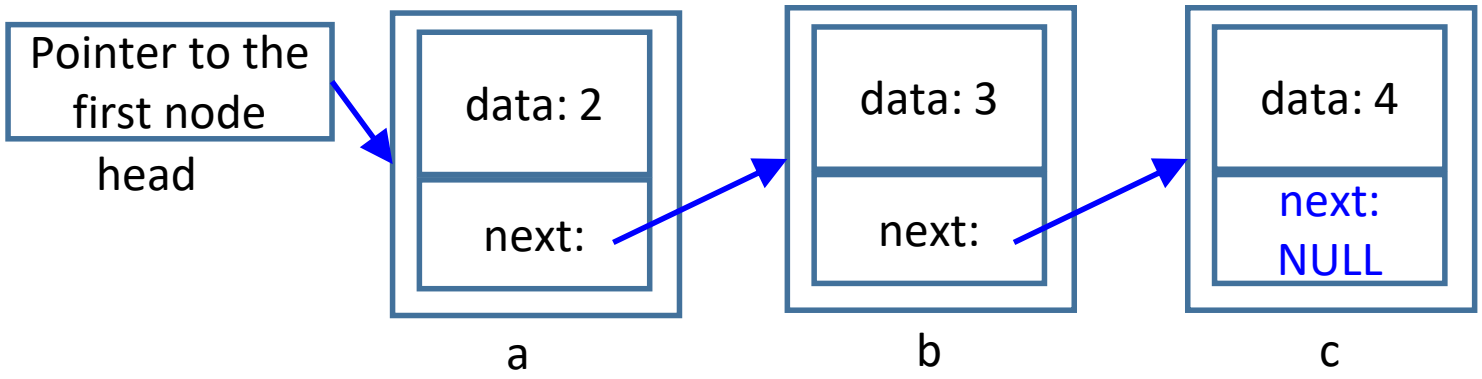


1. Head address is all we need to know
2. We follow the sequence by following the links
3. We stop when we found the node which has B as its next
4. Takes time O(n)

Linked List: main operations

`add(data)`	
`add_in_front (data)`	O(1)
`add_after(Node prev, data)`	O(1)
`add_before(Node next, data)`	O(n)
`remove(Node node)`	O(n)
`find(data)`	O(n)
`size()`	O(n)

Add to the end



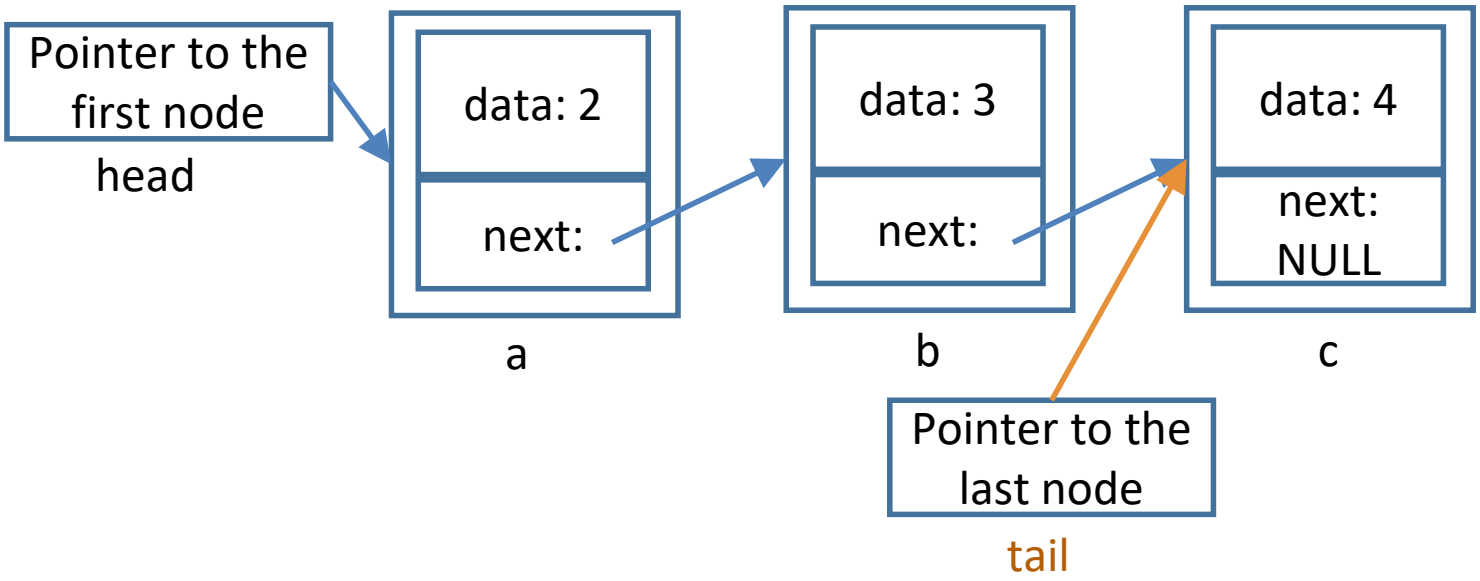
1. In order to add to the end of the list we first need to find the end
2. The traversal of the list takes time O(n)

Linked List: main operations

`add(data)`	O(n)
`add_in_front (data)`	O(1)
`add_after(Node prev, data)`	O(1)
`add_before(Node next, data)`	O(n)
`remove(Node node)`	O(n)
`find(data)`	O(n)
`size()`	O(n)

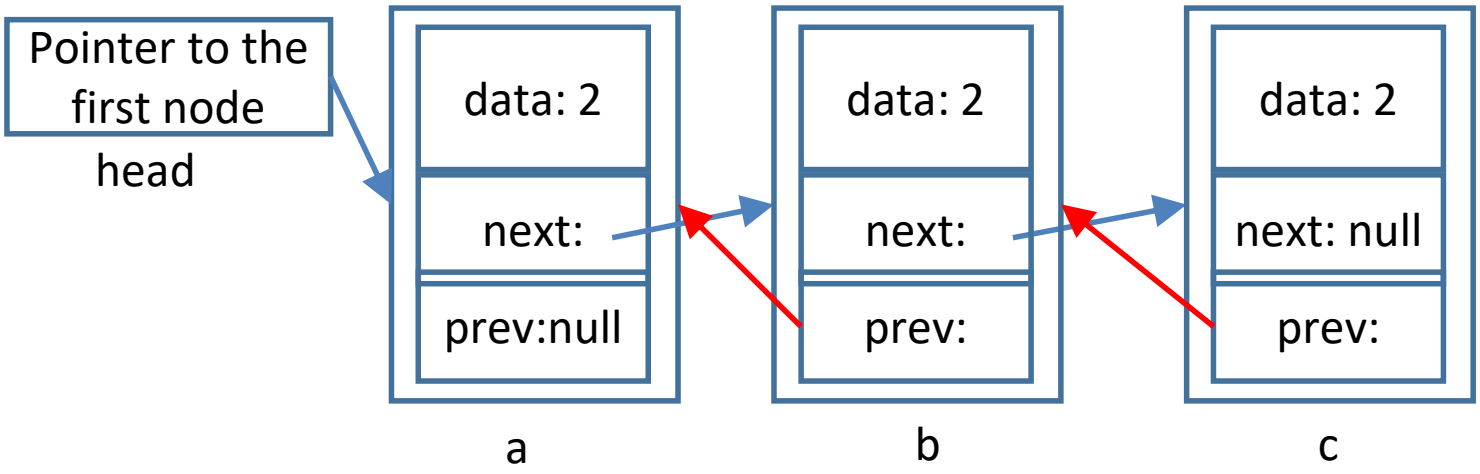
How can we modify the Linked List data structure to make *`add`*, *`add_before`* and *`remove`* work in time O(1)?

Linked List variants: tail



- The linked list we discussed so far is called a *singly-linked list*
- We can enhance it by storing the pointer to the last node - the **tail pointer** - and updating it after each insertion/deletion
- If we know the address of the last node, then we can add an element at the end of the list in time O(1) - think: how?

Linked List variants: previous



- We can enhance each node with the pointer to the previous node. Such list is called a ***doubly-linked list***
- Each time we add/remove an element, we will wire not only the link to the next, but also to the previous node in sequence
- If for each node we know the previous, then we can remove a given node in time O(1), and we can also add_before in time O(1) - think: how?

Linked List: main operations

	Singly - linked	With tail	Doubly - linked
`add(data)`	O(n)	O(1)	O(1)[with tail]
`add_in_front (data)`	O(1)	O(1)	O(1)
`add_after(Node prev, data)`	O(1)	O(1)	O(1)
`add_before(Node next, data)`	O(n)	O(n)	O(1)
`remove(Node node)`	O(n)	O(n)	O(1)
`find(data)`	O(n)	O(n)	O(n)
`size()`	O(n)	O(n)	O(n)

Can we improve *find*? Would keeping elements sorted help?

Arrays vs. Linked lists: run-time

	Array	Linked list
Add in front	O(n)	O(1)
Remove in front	O(n)	O(1)
Add in the middle	O(n)	O(1) doubly-linked
Remove in the middle	O(n)	O(1) doubly-linked
Add at the end	O(1)	O(1) with tail pointer
Remove at the end	O(1)	O(1) with tail pointer
Get element by position	O(1)	O(n)
Search for an element	O(n)/O(log n) if sorted	O(n)

Summary. Linked list

Advantages

- Can hold unlimited number of elements
- Adding/Removing in the beginning and the end is cheap
- Adding/Removing in the middle requires some work, but does not require moving other elements

Disadvantages

- No constant-time access to an element by its position
- Memory overhead (to store links)
- Cannot improve search even if the list is sorted (totally sequential access)

Python implementation: add_in_front

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def add_in_front(self, newdata):
        newnode = Node(newdata)

        newnode.next = self.head
        self.head = newnode
```

Python implementation: size()

```
class LinkedList:
    def __init__(self):
        self.head = None

    def size(self):
        count = 0
        current = self.head
        while current:
            count += 1
            current = current.next

        return count
```

Python implementation: string representation

```
class LinkedList:
    def __init__(self):
        self.head = None

    def __str__(self):
        s = ''
        current = self.head
        while current:
            s += str(current.data) + "->"
            current = current.next

        return s
```