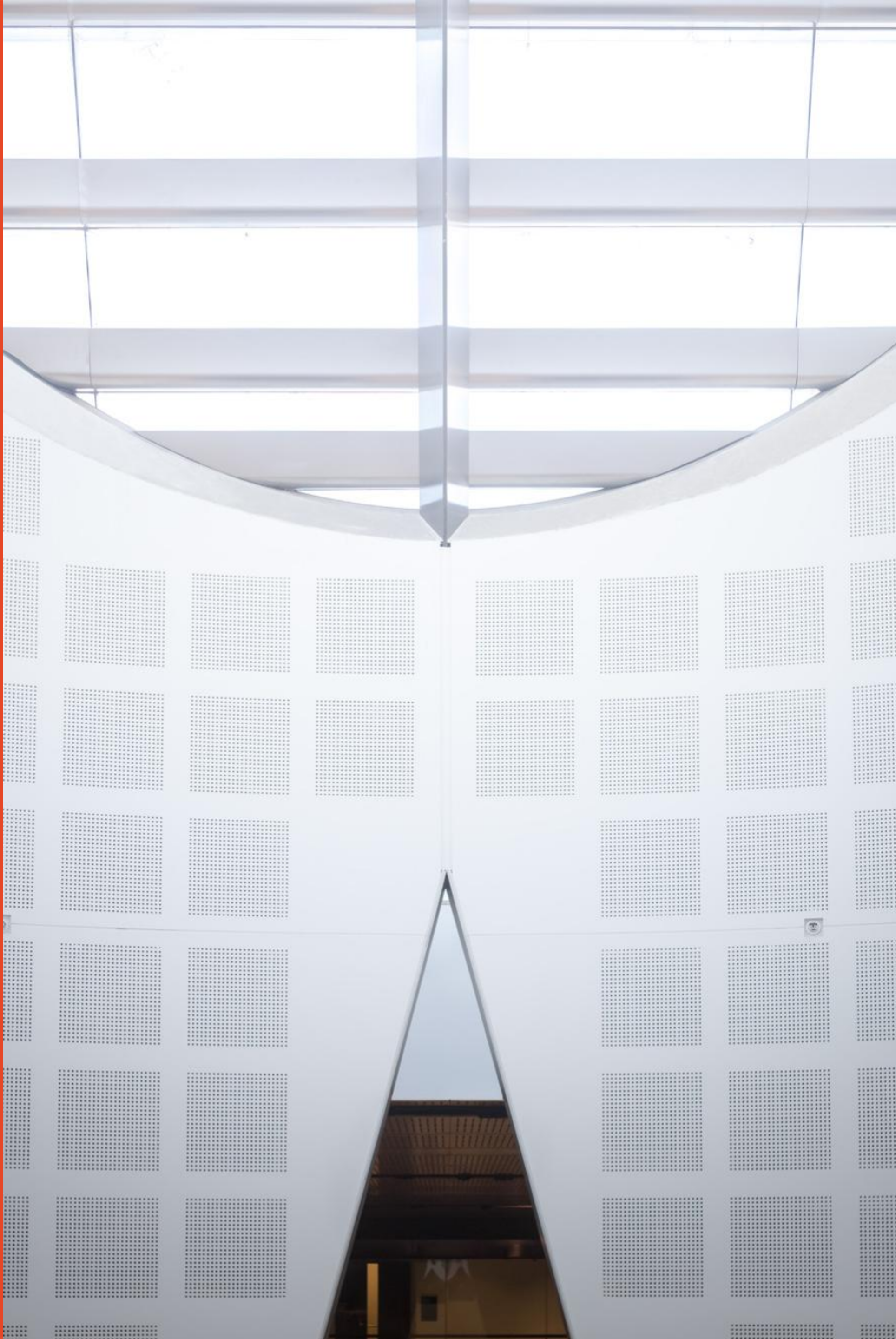
COMP9123: Data Structures & Algorithms
Lecture 02 - Lists

Dr. Ravihansa Rajapakse
Dr. Karlos Ishac
School of Computer Science

THE UNIVERSITY OF SYDNEY



Recap

- Lecture 1 covered the fundamentals of data structures, starting with an introduction to how data can be organized and managed efficiently.
- We explored the basics of Python, including variables, control flow, and functions, to build a foundation for implementing data structures.
- Big-O notation was introduced as a crucial concept for evaluating algorithm efficiency, covering common complexities to understand performance trade-offs.
- Finally, we discussed arrays as a fundamental data structure, exploring key operations such as access, insertion, deletion, and search.

Questions & Announcements

- Cannot find tutorial solutions…
  - Will be released next week
- Can I use Python in assignments?
  - Yes, but provide explanations.
- Do I need to use Latex for assignments?
  - No
- Can I use other programming languages?
  - Please use Python
- Download lecture slides AFTER the lecture for the most up-to-date version.
- Assignment answers need to include explanations.
- Check “Advice on how to do the assignment” page and “Guide to Written Assignments” on Ed

Linked Lists

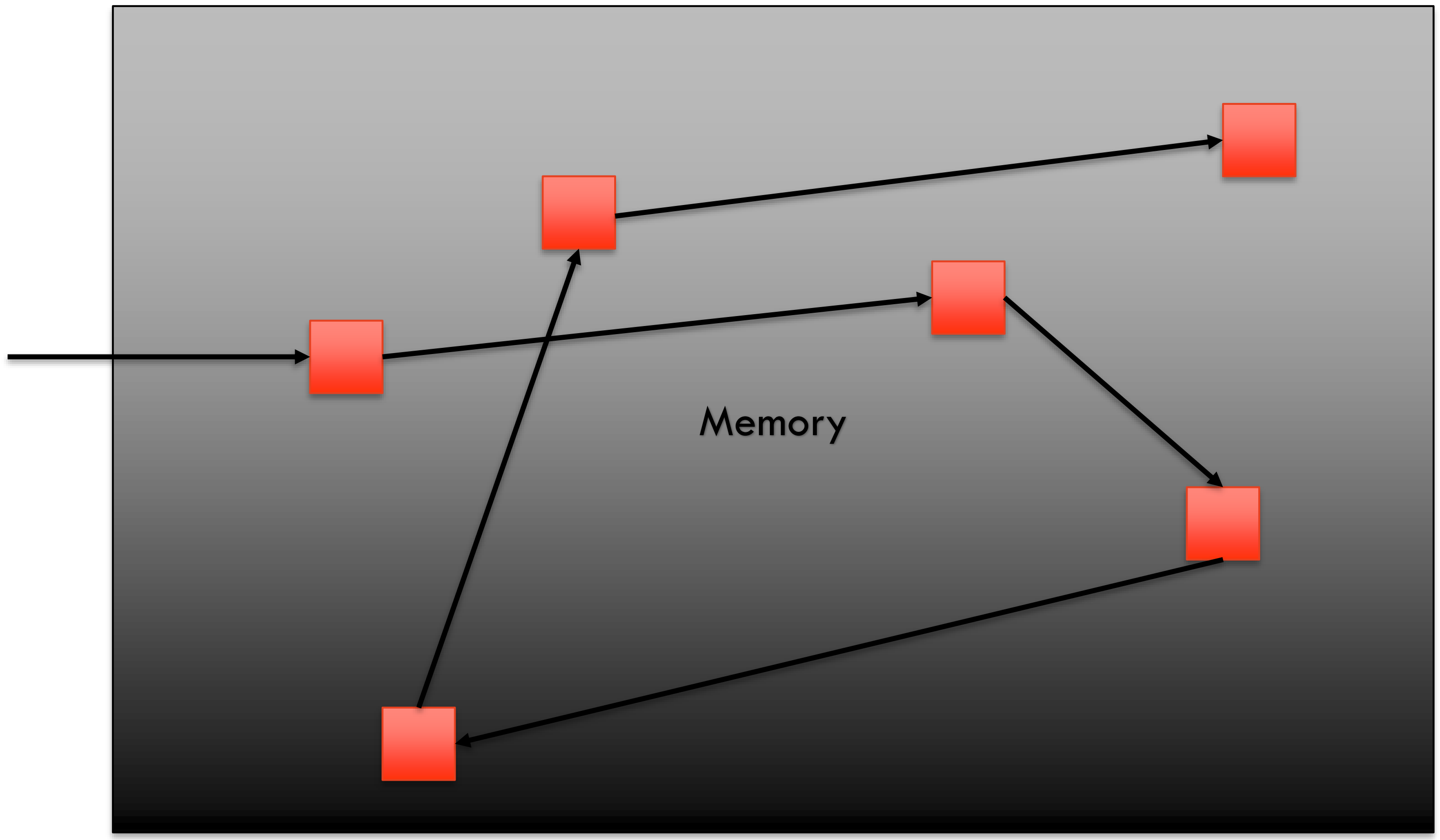
Linked List?

A Linked List is a linear data structure that is used to store a collection of data with the help of nodes.

Key Characteristics:

- The consecutive elements (nodes) are connected by pointers/references.
- The last node of the linked list points to None/Null.
- The entry point of a linked list is known as the head.
- The common types of linked lists are Singly, Doubly and Circular.

How are Linked Lists Stored?



Why Linked Lists?

Dynamic Sizing:

- Unlike arrays, linked lists do not require a predefined size. They can grow and shrink efficiently as needed without reallocating memory. The last node of the linked list points to None.

Flexible Memory Allocation:

- Linked lists use scattered memory blocks instead of requiring contiguous memory allocation like arrays. This reduces memory fragmentation and makes better use of available space.

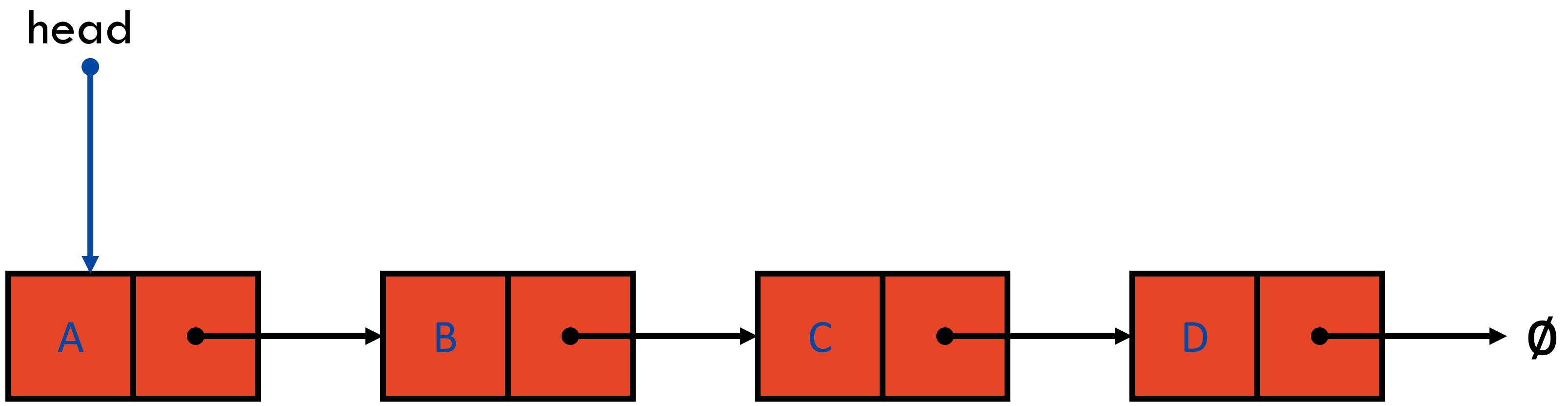
No Wasted Space

- Since linked lists allocate memory dynamically, they avoid the issue of wasted space in arrays where unused slots may remain empty.

Singly Linked Lists

Singly Linked List

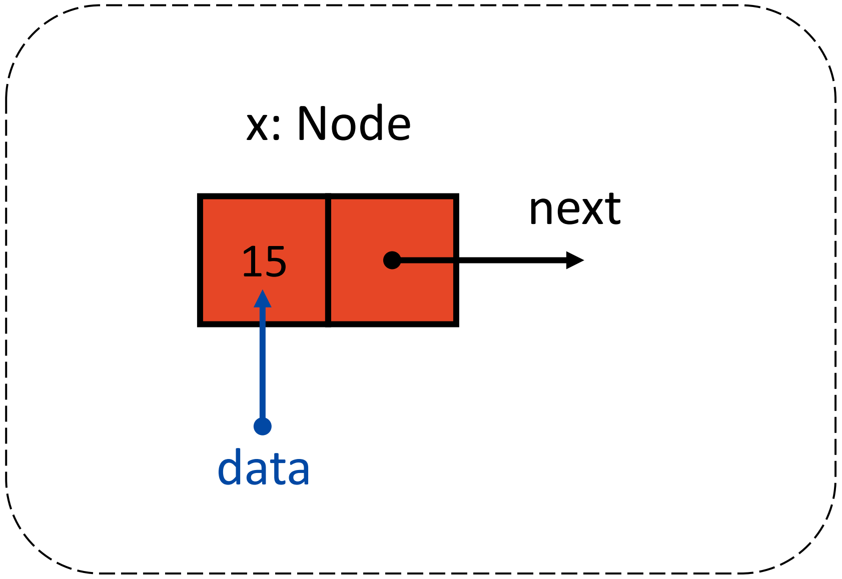
- A concrete data structure
- A sequence of Nodes, each with a reference to the next node
- List captured by reference (head) to the first Node



Node implements Position

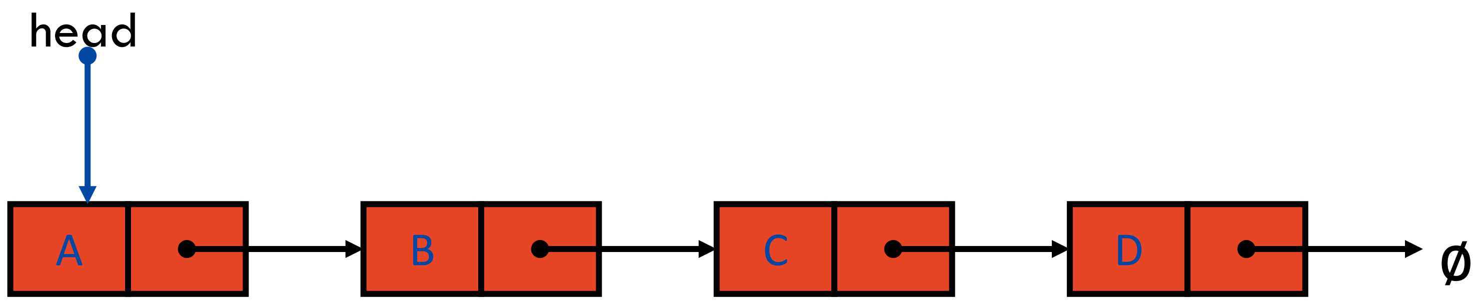
Each Node in a singly linked List stores

- its data, and
- a link to the next node.



Advice on working with linked structures

- Draw the diagram showing the state.
- Show a location where you place carefully each of the instance variables (including references to nodes).
- Be careful to step through dotted accesses.
  - Example: p.next.next
- Be careful about assignments to fields.
  - Example: p.next = q or p.next.next = r



Operations on Singly Linked List

- Insertion
  - Insert at the beginning
  - Insert at the end
  - Insert at a specific position
- Deletion
  - Delete from the beginning
  - Delete from the end
  - Delete a specific node
- Traversal

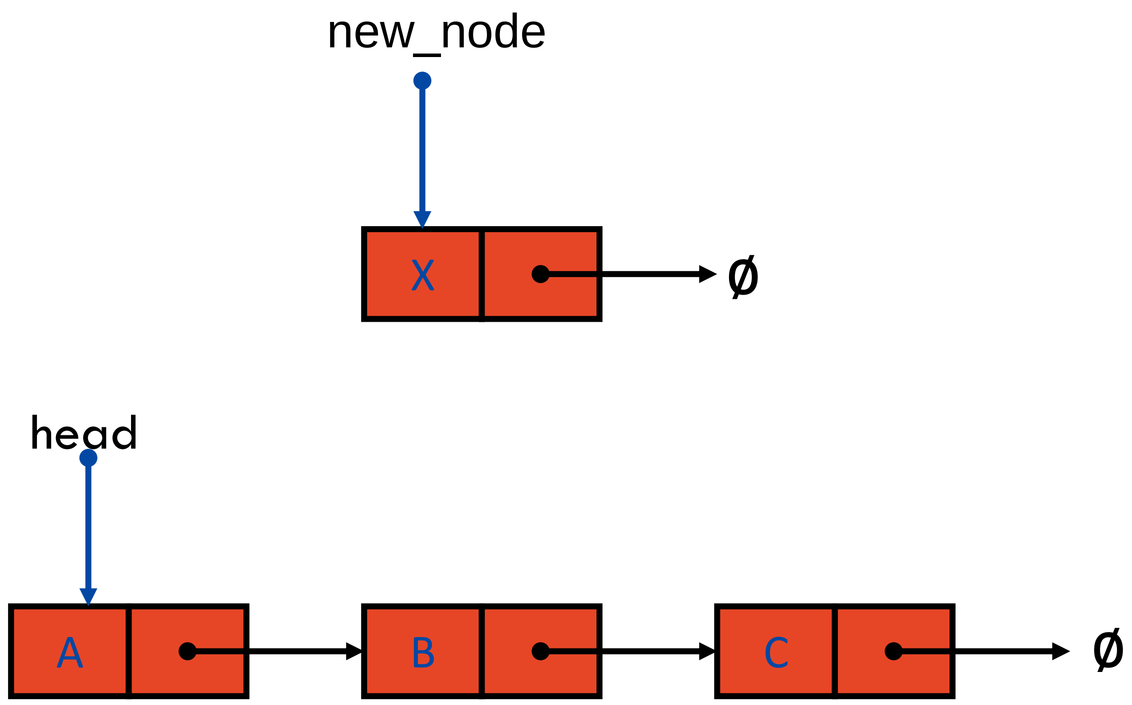
Insertion

Singly Linked List

THE UNIVERSITY OF SYDNEY

Insert at the Beginning (Step 01)

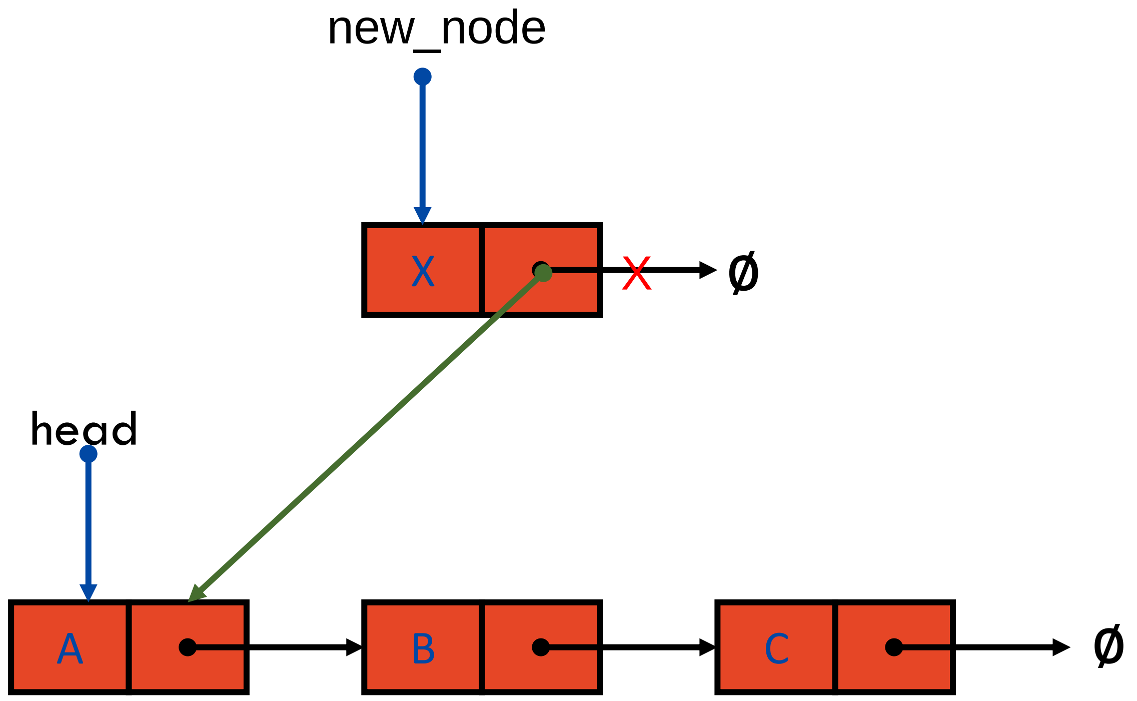
1. Create a new node with data x



Insert at the Beginning (Step 02)

2. Set the next pointer of new node to the current head

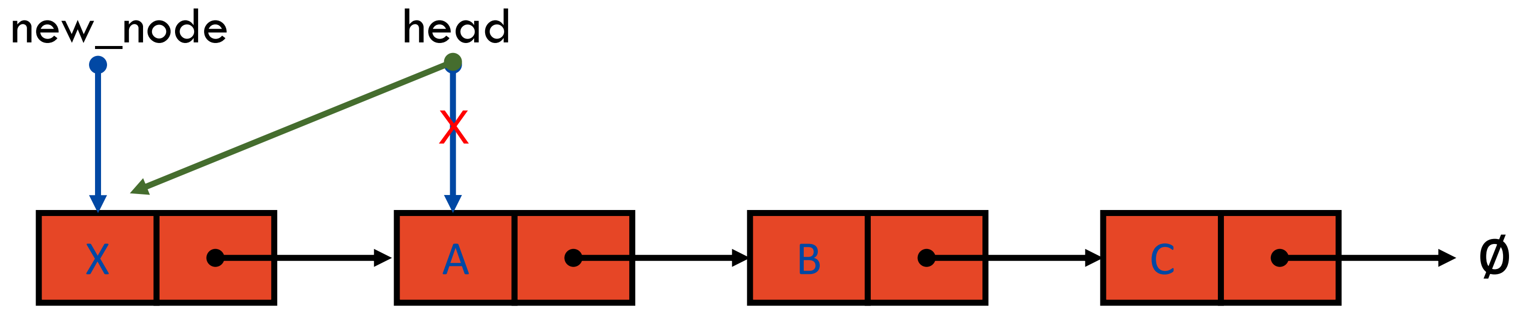
   new_node.next = head



Insert at the Beginning (Step 03)

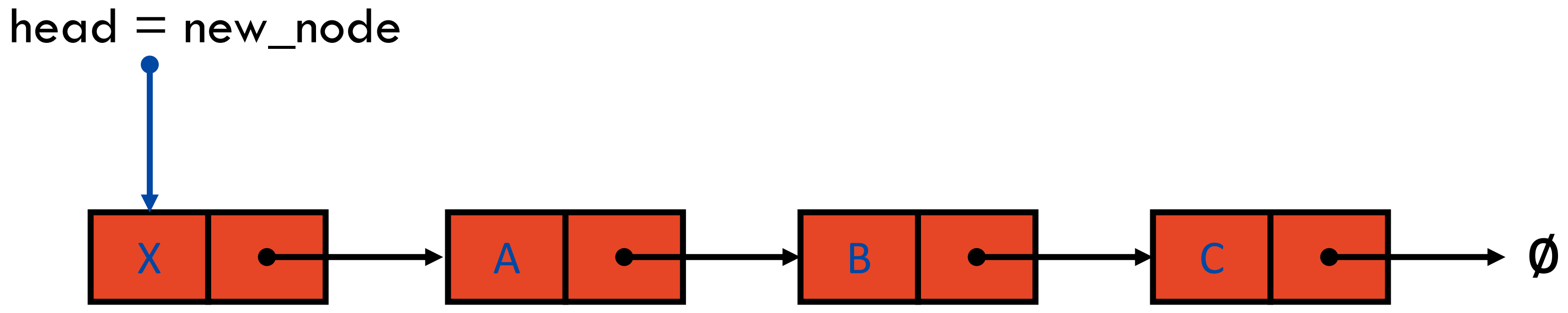
3. Move the head to point to the new node

    head = new_node



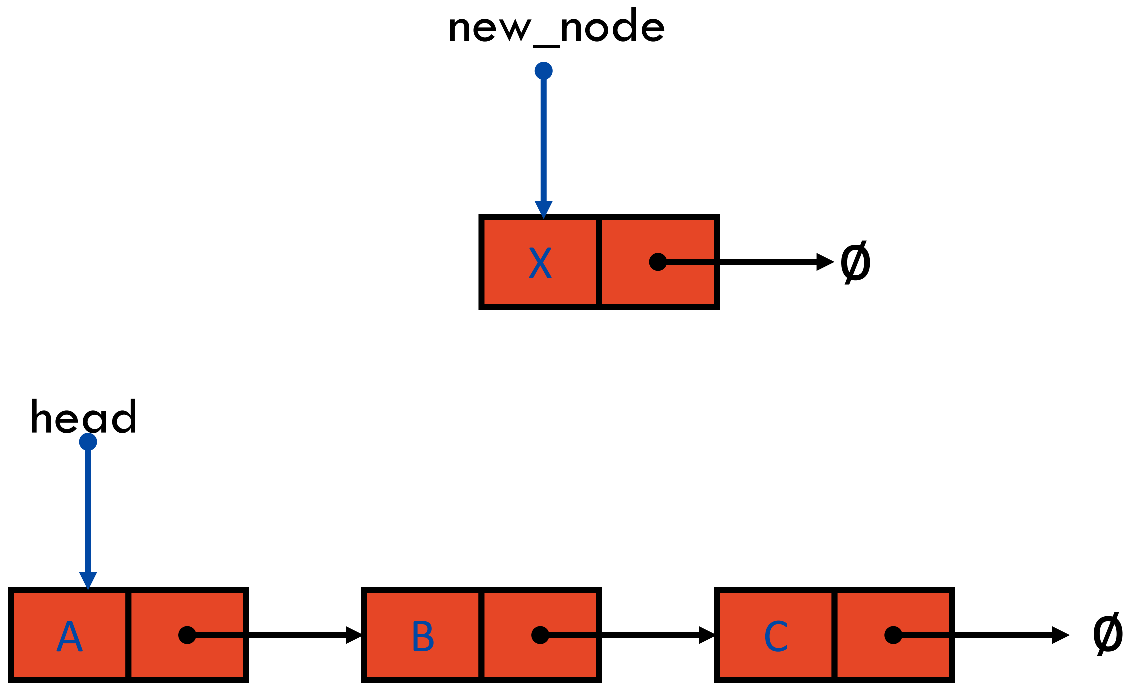
Insert at the Beginning (Result)

1. Create a new node with data x
2. Set the next pointer of new node to the current head
3. Move the head to point to the new node



Insert at the End (Step 01)

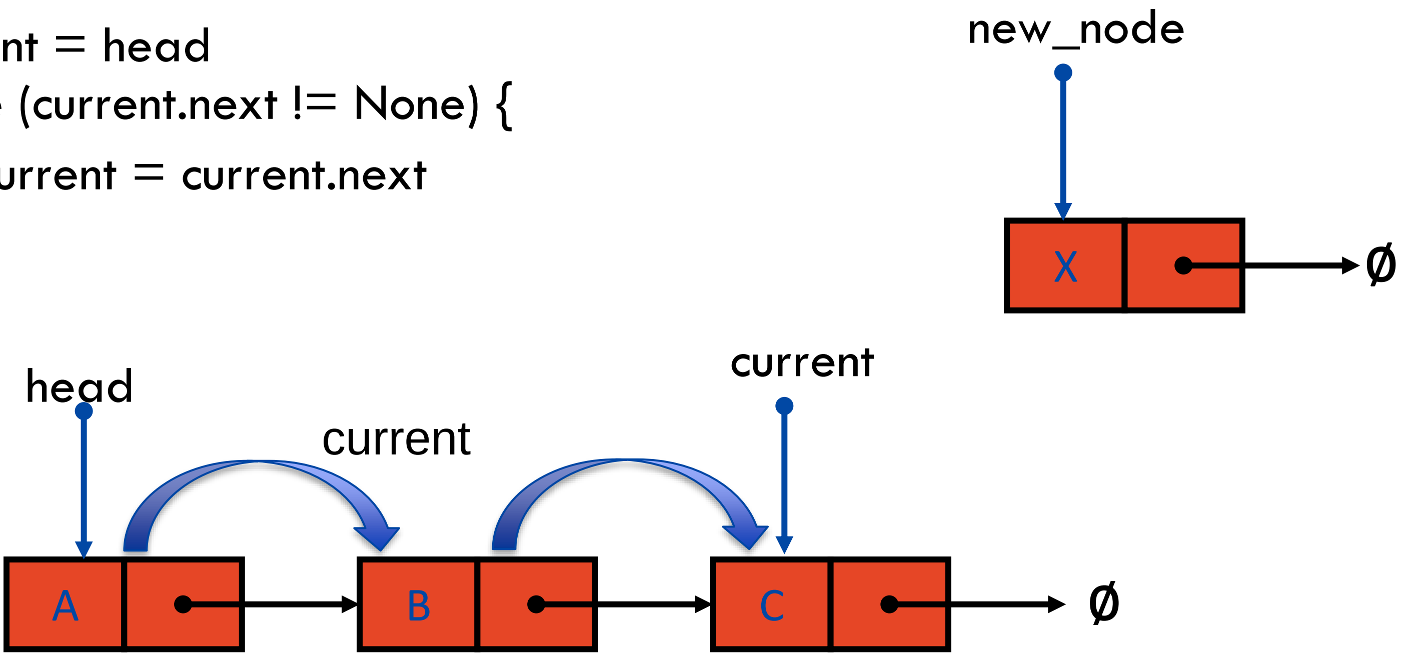
1. Create a new node with data x



Insert at the End (Step 02)

2. Traverse the list until the last node is reached

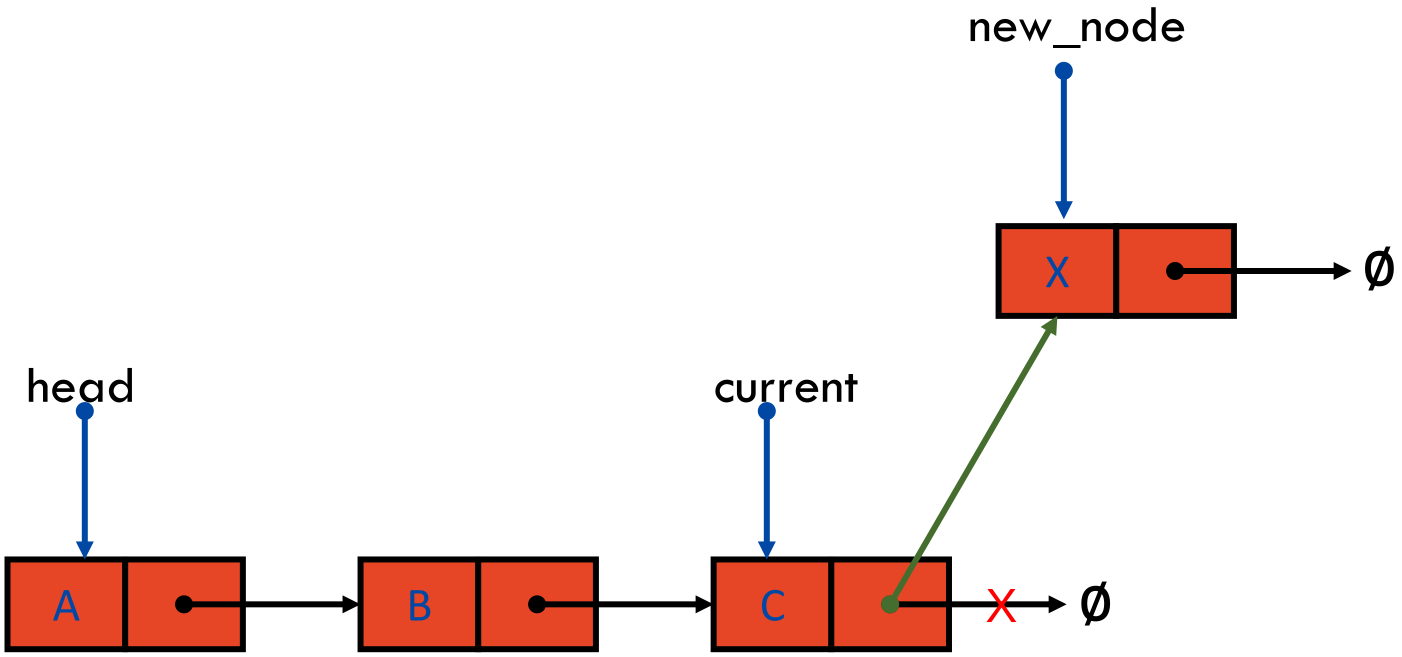
```
current = head
while (current.next != None) {
    current = current.next
}
```



Insert at the End (Step 03)

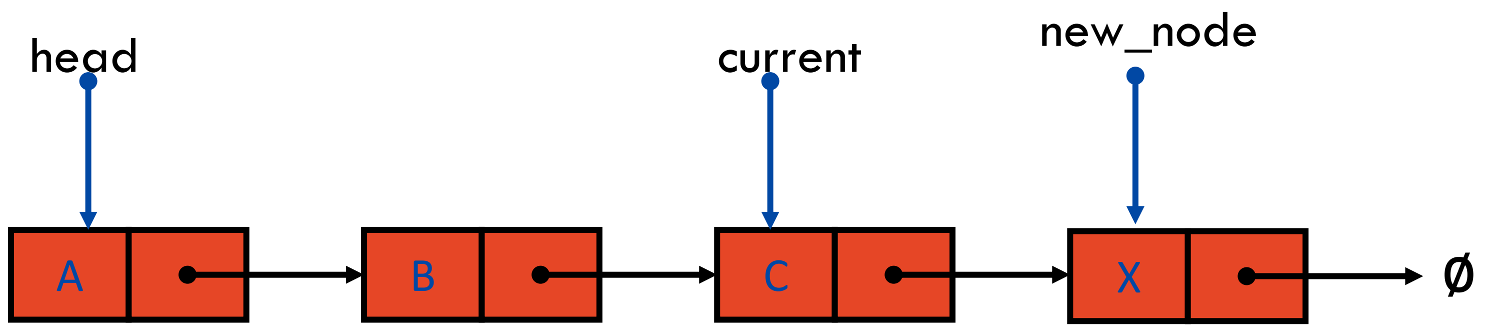
3. Link the new node to the current last node

    current.next = new_node



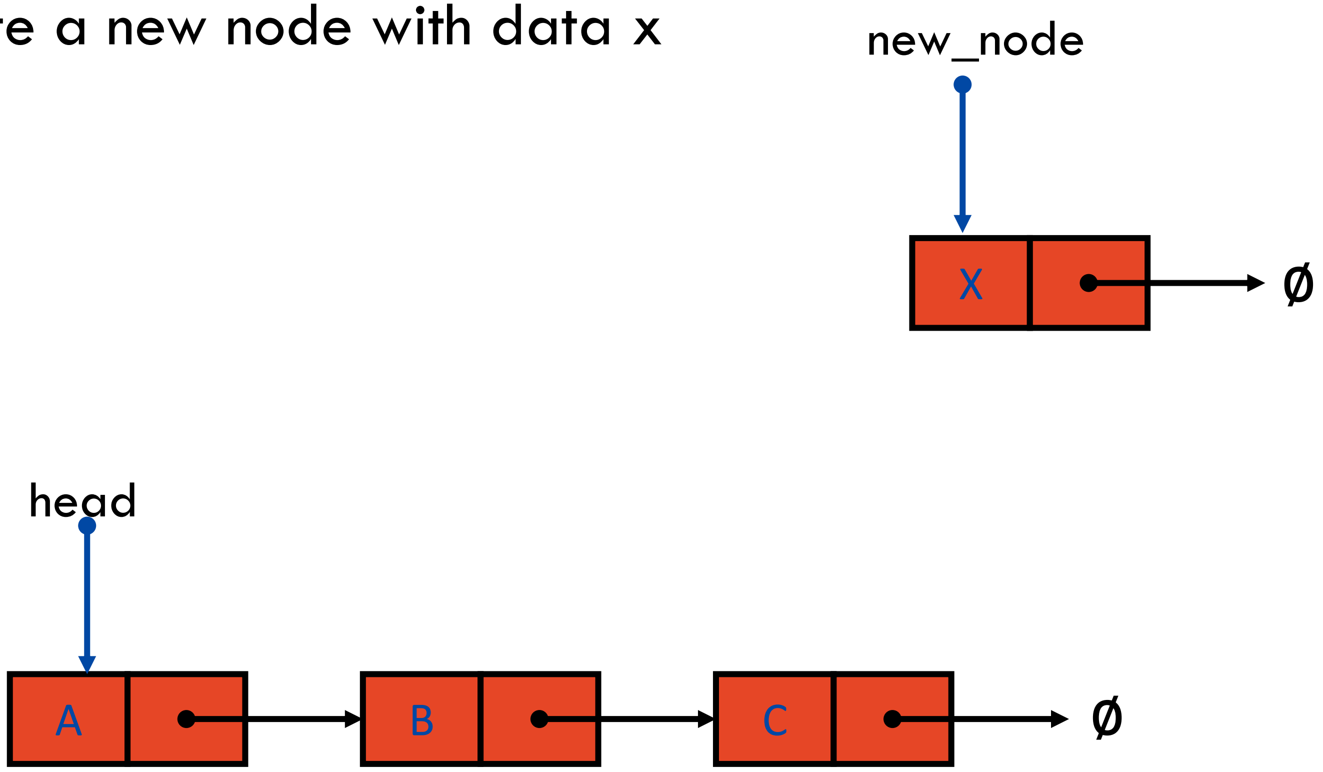
Insert at the End (Result)

1. Create a new node with data x
2. Traverse the list until the last node is reached
3. Link the new node to the current last node



Insert at a Specific Position (Step 01)

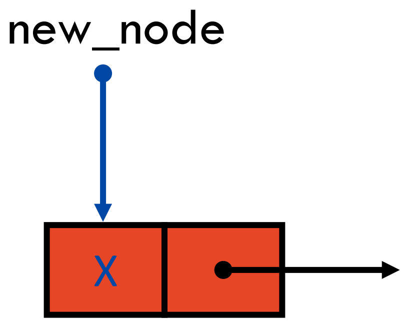
1. Create a new node with data x

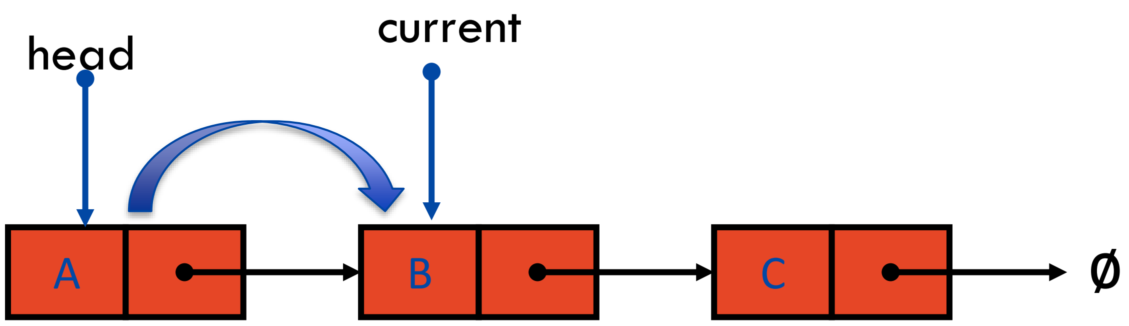


Insert at a Specific Position (Step 02)

2. Traverse the list to the desired position (i)

```
current = head
int count = 0
while (count < i-1 AND current != None) {
    current = current.next
     count++
}
```

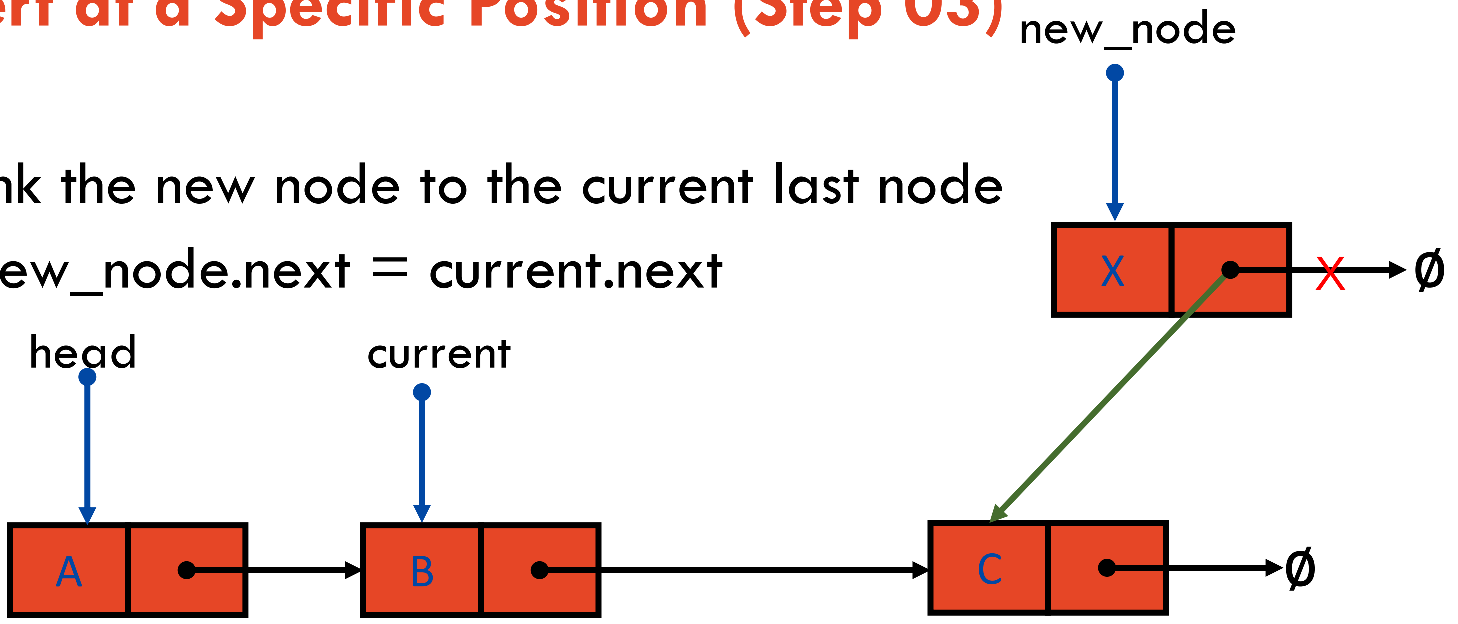




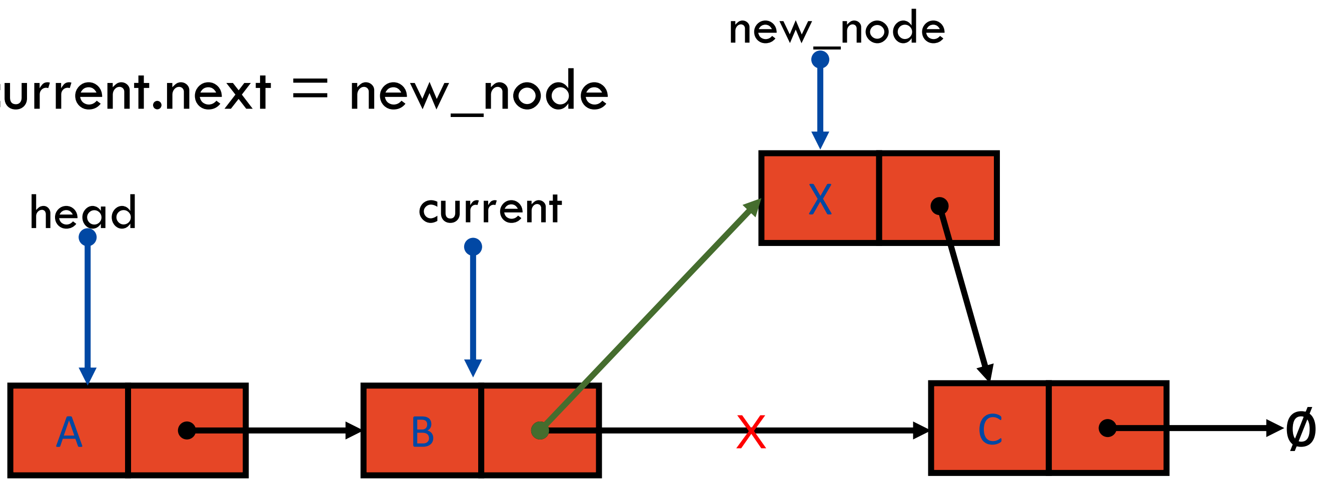
Insert at a Specific Position (Step 03)

3. Link the new node to the current last node

new_node.next = current.next

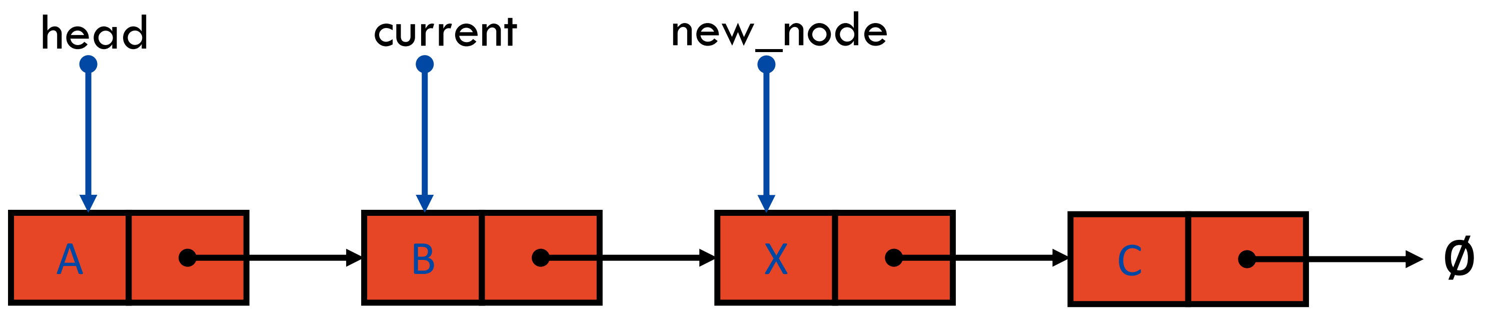


current.next = new_node



Insert at a Specific Position (Result)

1. Create a new node with data x
2. Traverse the list to the desired position (i)
3. Link the new node to the current last node



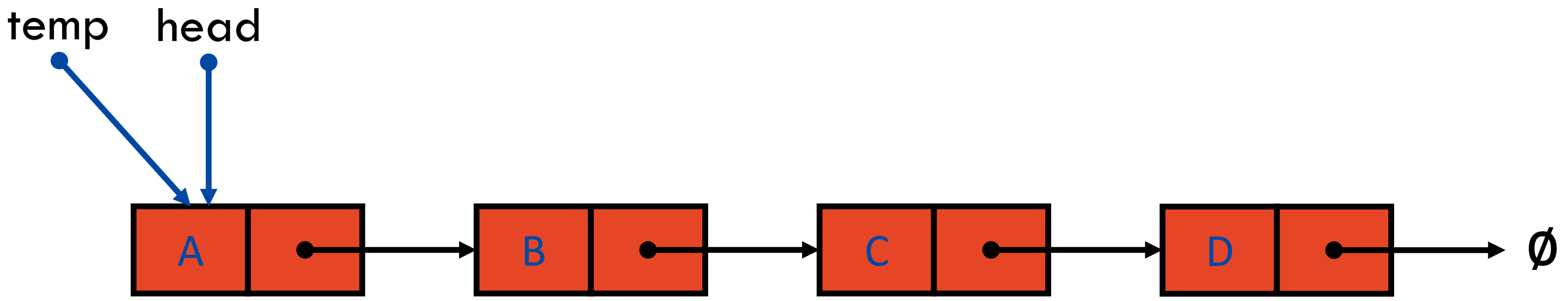
Deletion

Singly Linked List

Deletion at the Beginning (Step 01)

1. Store the current head node in a temporary variable

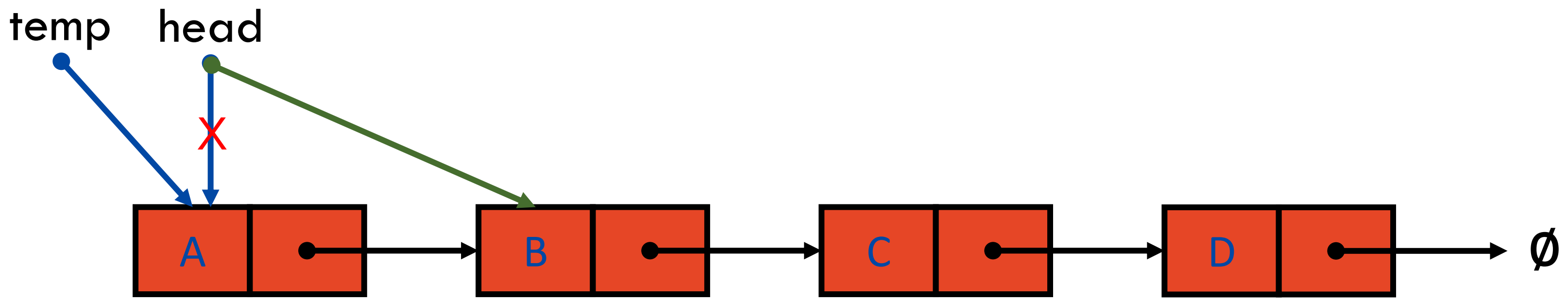
   temp = head



Deletion at the Beginning (Step 02)

2. Move the head pointer to the next node

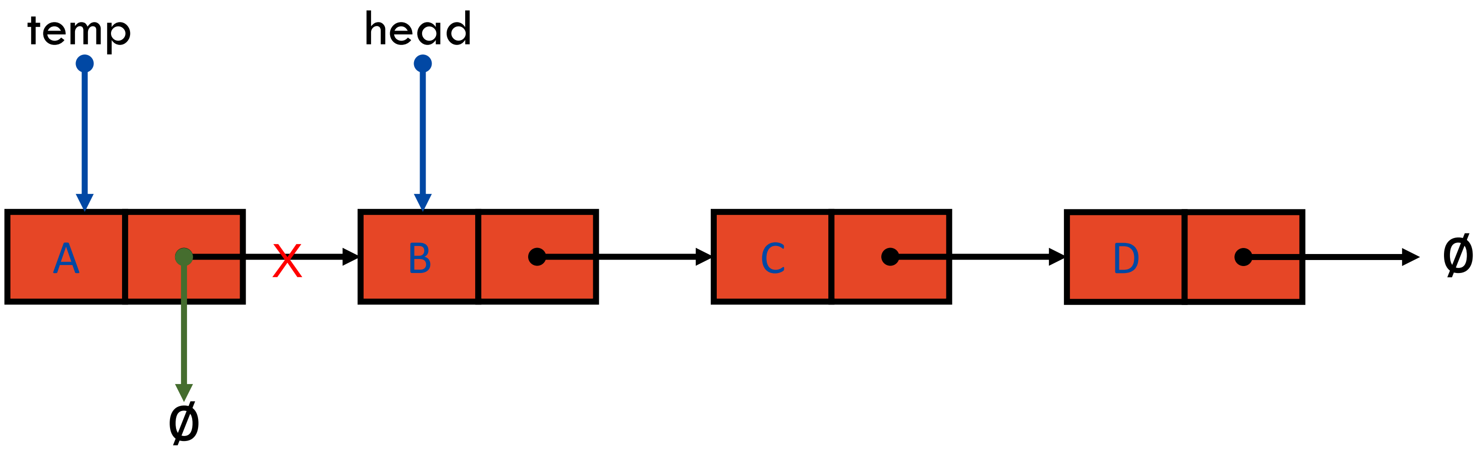
   head = head.next



Deletion at the Beginning (Step 03)

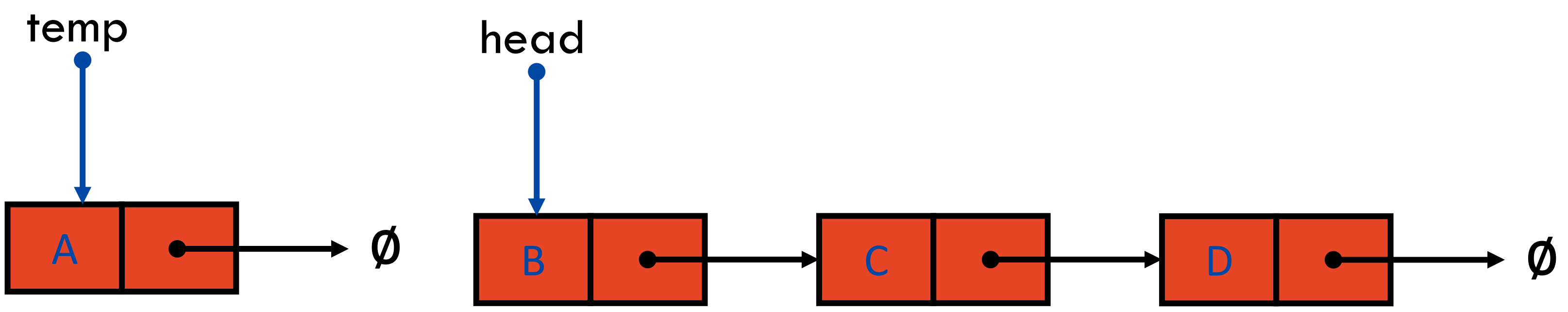
3. Remove the link from the temp node

```
        temp.next = None
```



Deletion at the Beginning (Result)

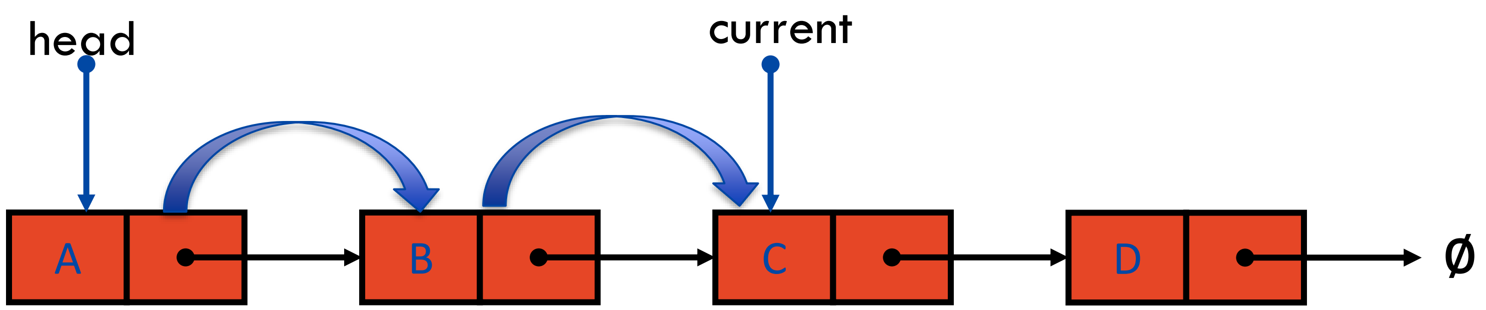
1. Store the current head node in a temporary variable
2. Move the head pointer to the next node
3. Remove the link from the temp node



Deletion at the End (Step 01)

1. Traverse the list to find the second last node

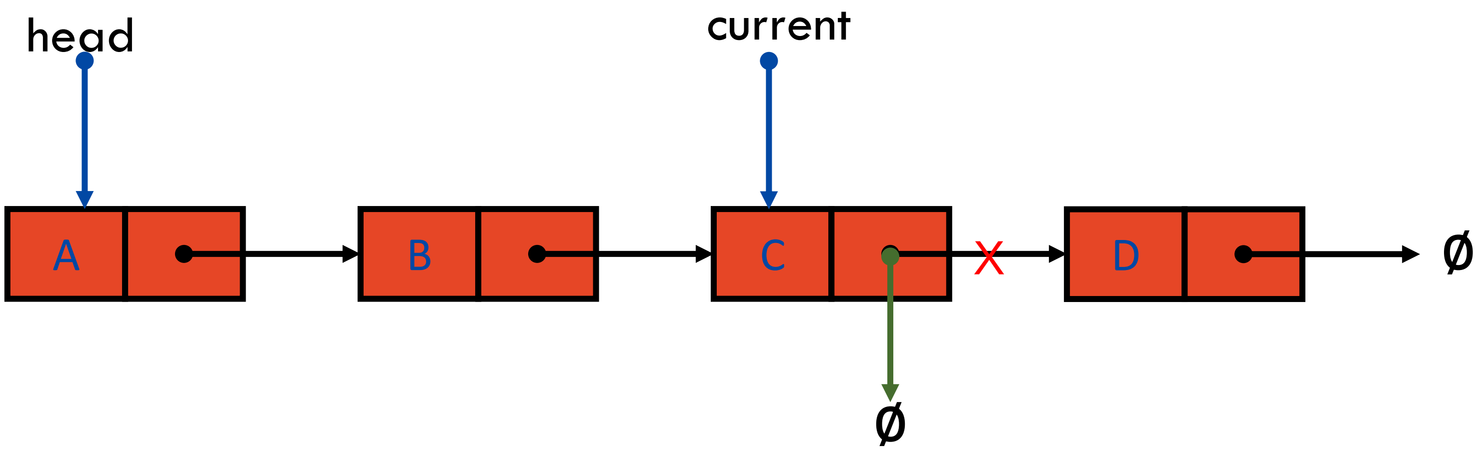
```
current = head
while (current.next.next != None){
    current = current.next
}
```



Deletion at the End (Step 02)

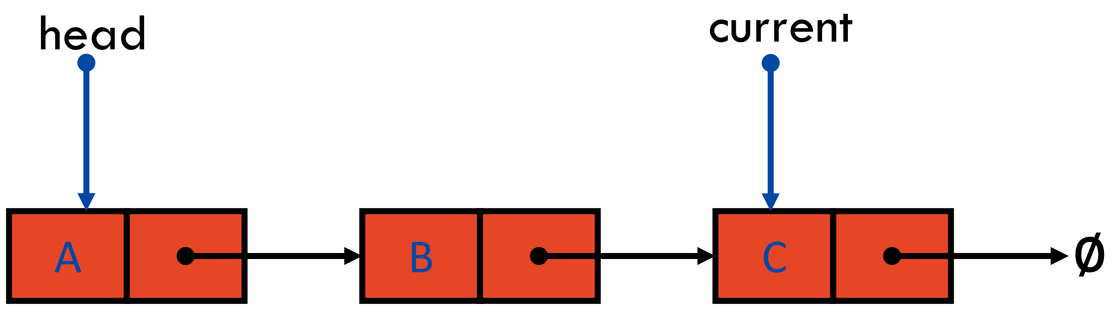
2. Remove the link to last node

current.next = None



Deletion at the End (Result)

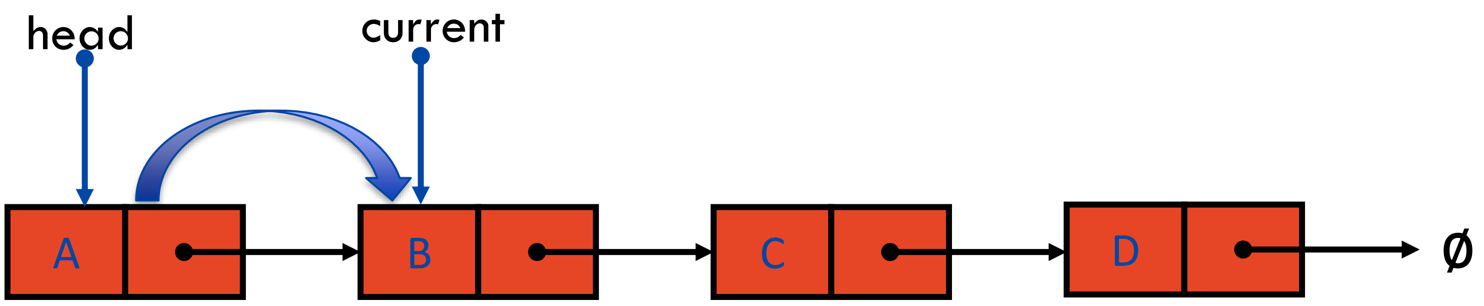
1. Traverse the list to find the second last node
2. Remove the link to last node



Deletion at Specific Position (Step 01)

1. Traverse to the node before the position to be deleted

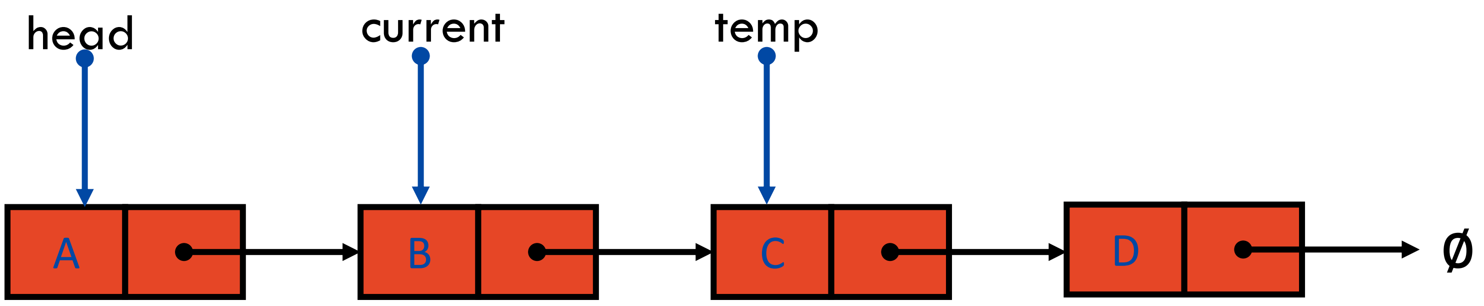
```
current = head
count = 0
while (count < index-1 AND current.next != None){
    current = current.next
    count++
}
```



Deletion at Specific Position (Step 02)

2. Store the node to be deleted

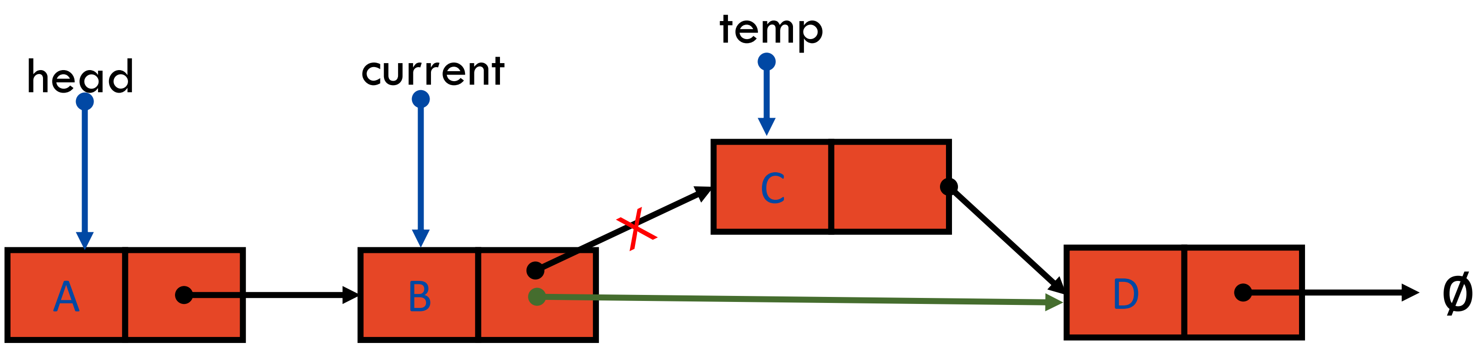
    temp = current.next



Deletion at Specific Position (Step 03)

3. Update the links to remove the node

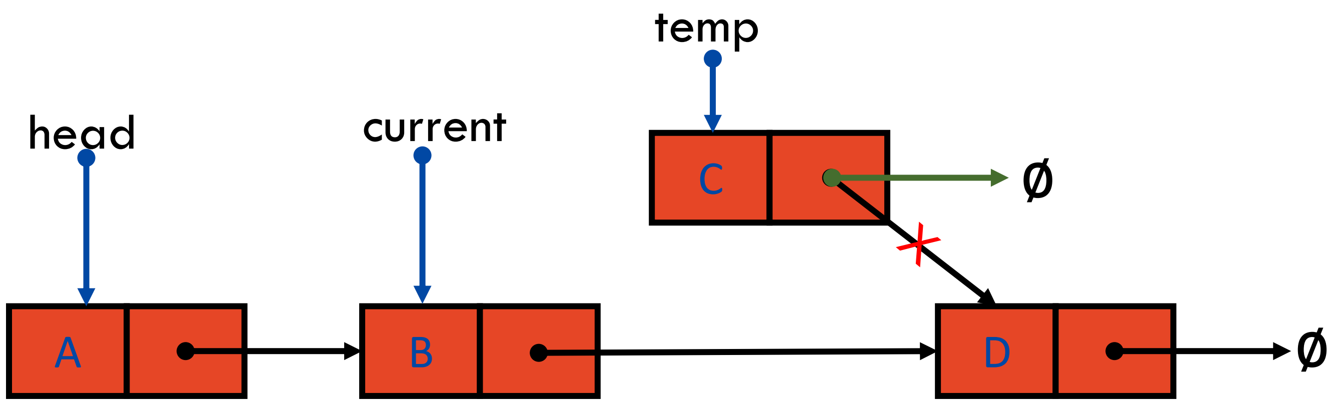
    current.next = temp.next



Deletion at Specific Position (Step 04)

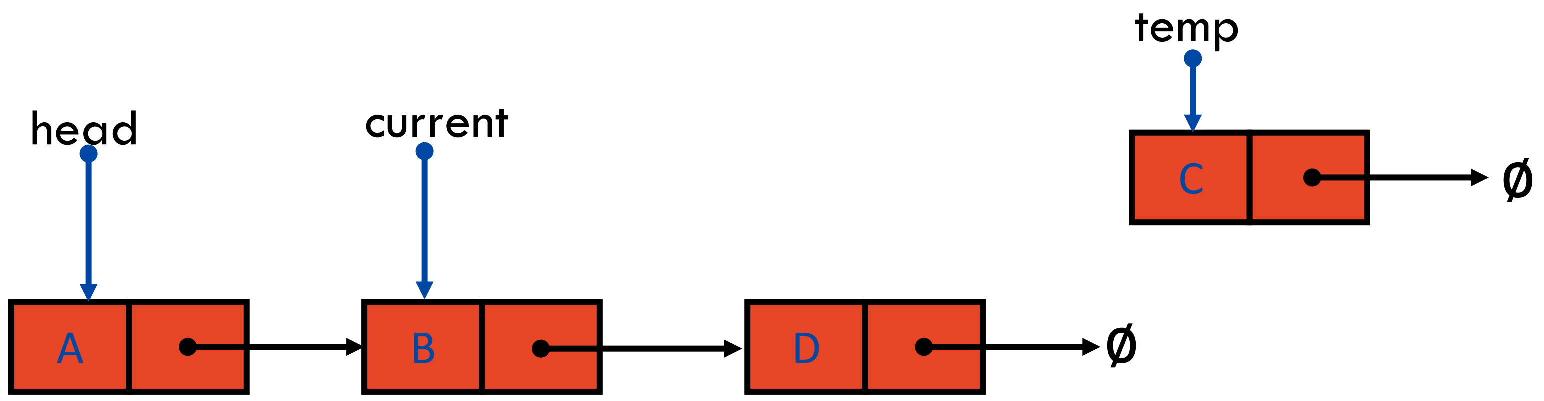
4. Remove the link from the temp node

   temp.next = None



Deletion at Specific Position (Result)

1. Traverse to the node before the position to be deleted
2. Store the node to be deleted
3. Update the links to remove the node
4. Remove the link from the temp node

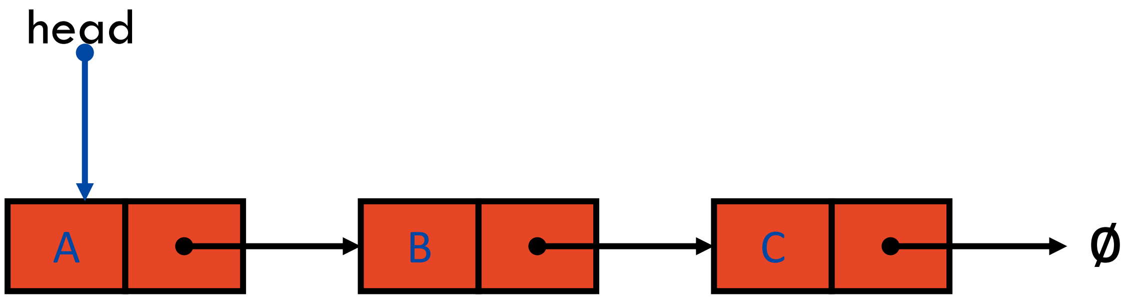


Traversal

Singly Linked List

Traversal

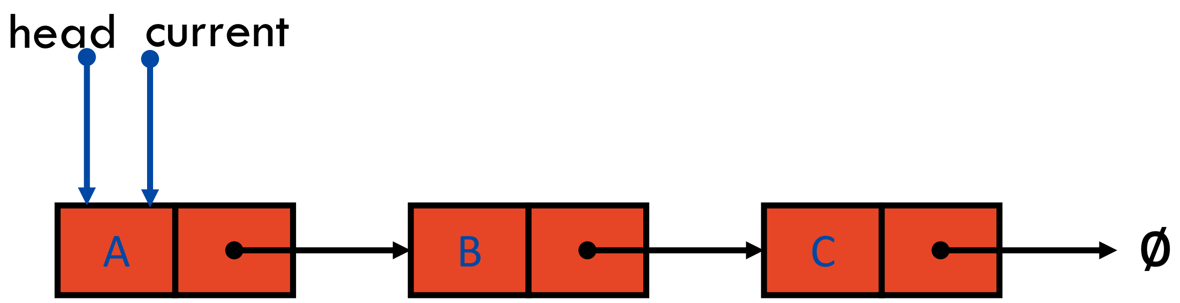
Traversal of a Singly Linked List is one of the fundamental operations, where we traverse or visit each node of the linked list.



Traversal (Step 01)

1. Store the current head node in a temporary variable

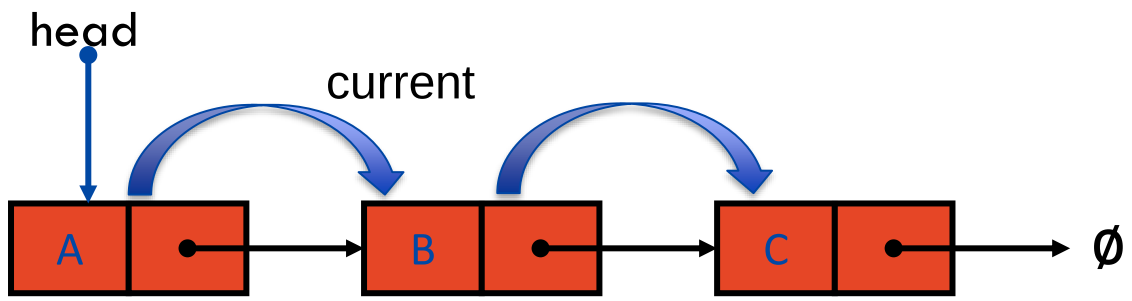
    current = head



Traversal (Step 02)

2. Traverse the list until the last node is reached

```
current = head
while (current != None) {
    print(current.data)
    current = current.next
}
```



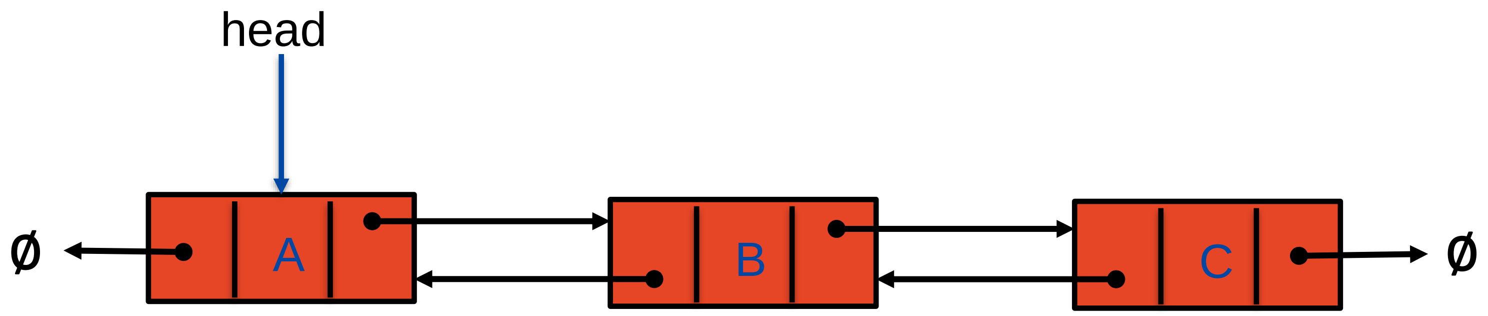
Doubly Linked Lists

Doubly Linked List

– A concrete data structure

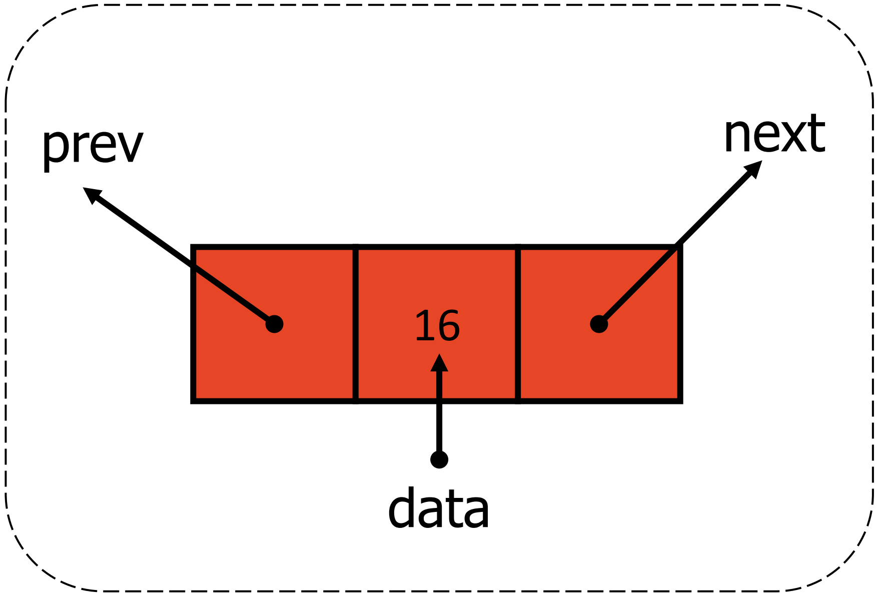
– A sequence of Nodes, each with reference to prev and to next

– List captured by reference (head) to the first Node



Doubly Linked List Node

- Doubly linked list is represented using nodes that have three fields:
  - Data
  - A pointer to the next node (next)
  - A pointer to the previous node (prev)



Operations on Doubly Linked List

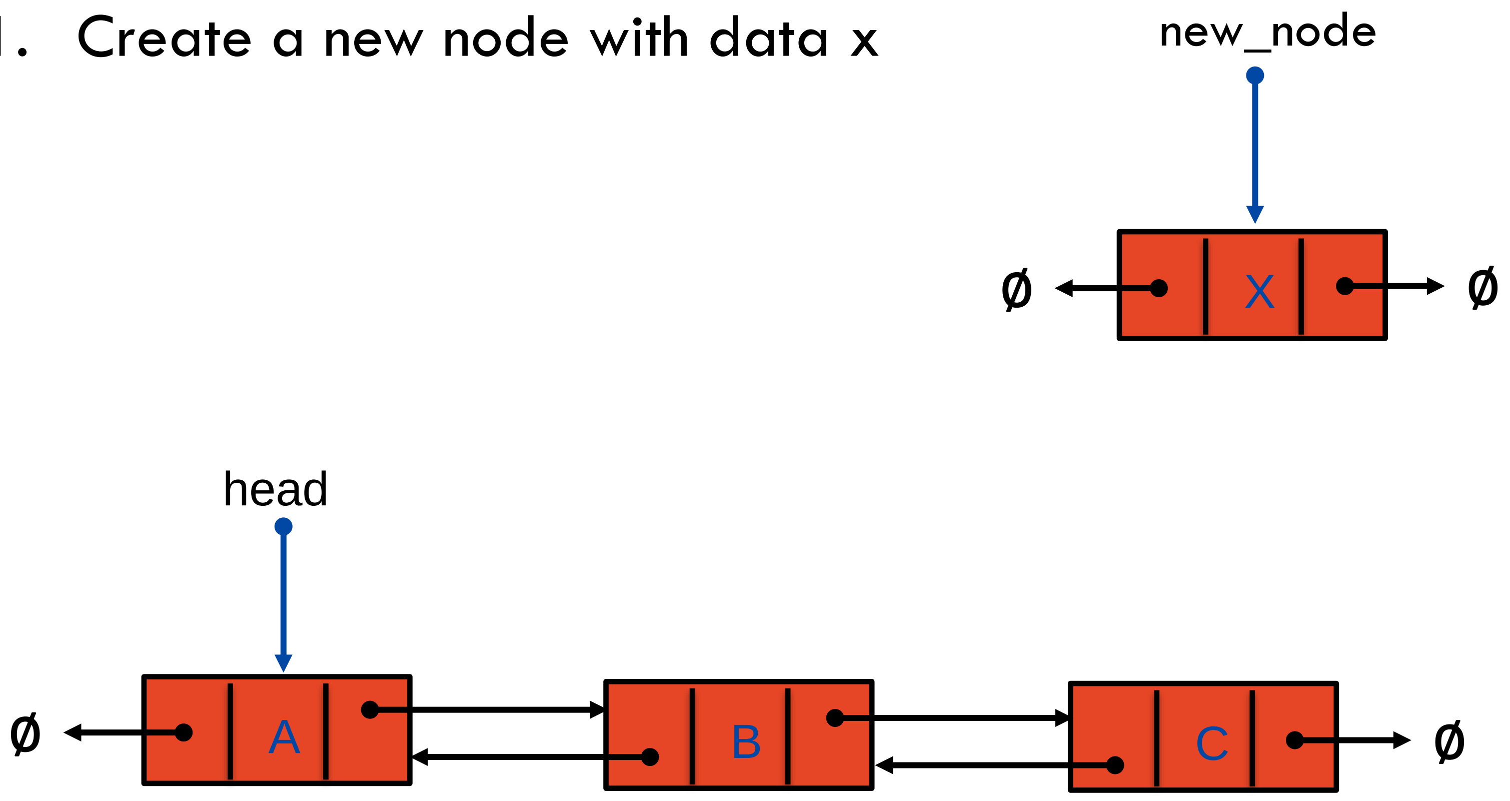
- Insertion
  - Insert at the beginning
  - Insert at the end
  - Insert at a specific position
- Deletion
  - Delete from the beginning
  - Delete from the end
  - Delete a specific node
- Traversal

Insertion

Doubly Linked List

Insert at the Beginning (Step 01)

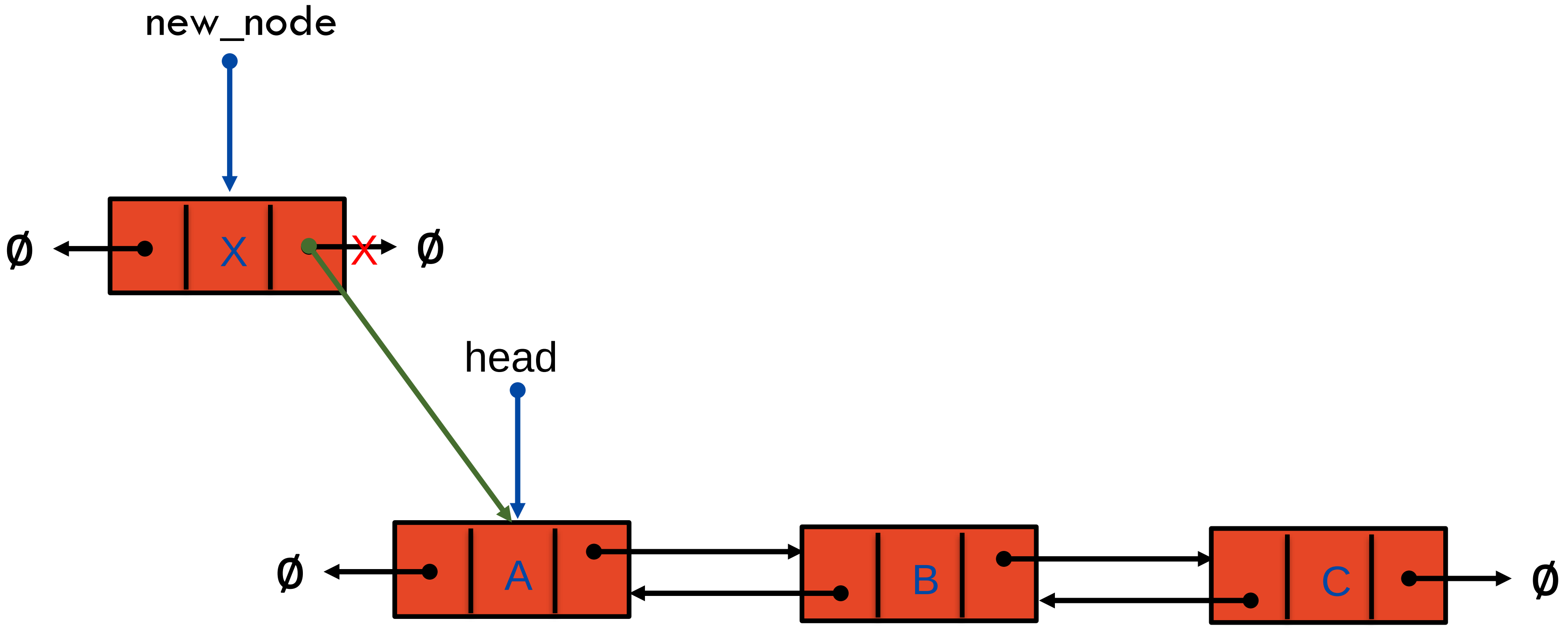
1. Create a new node with data x



Insert at the Beginning (Step 02)

2. Set the next pointer of new node to the current head

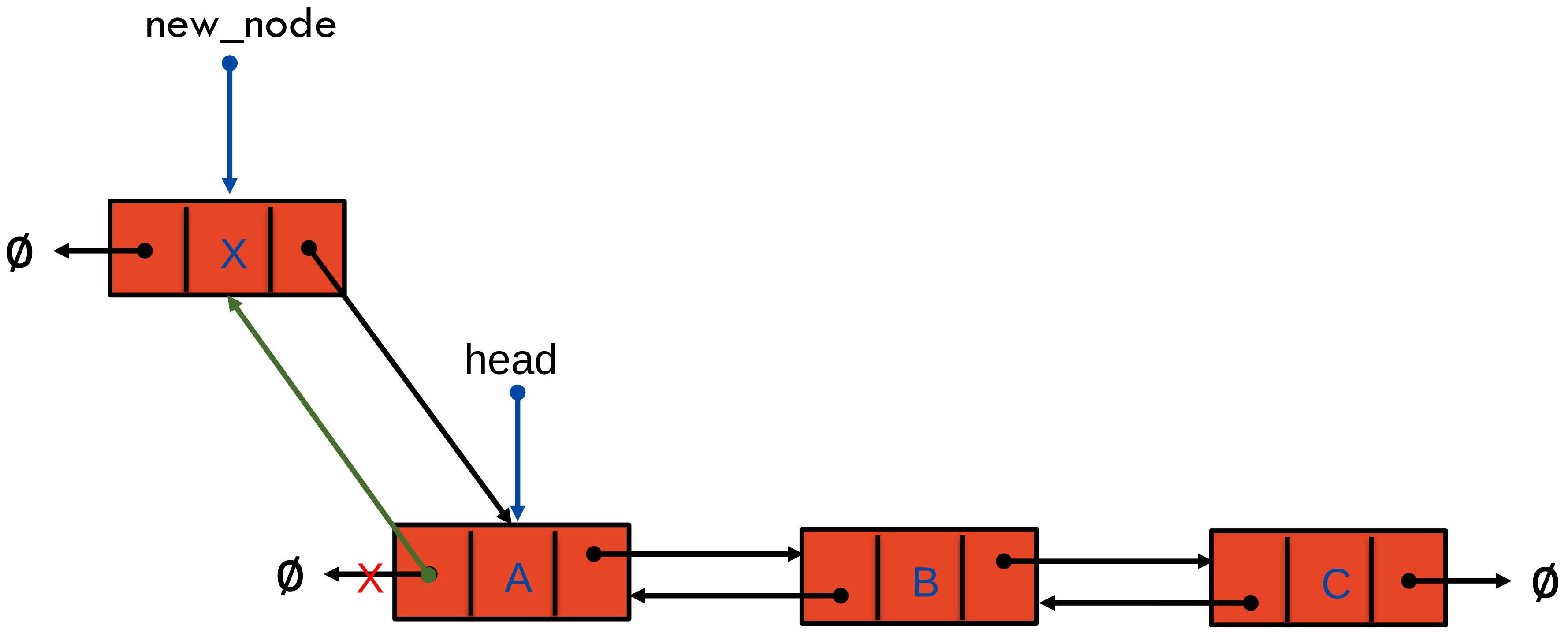
    new_node.next = head



Insert at the Beginning (Step 03)

3. Update the previous pointer of the head to the new node

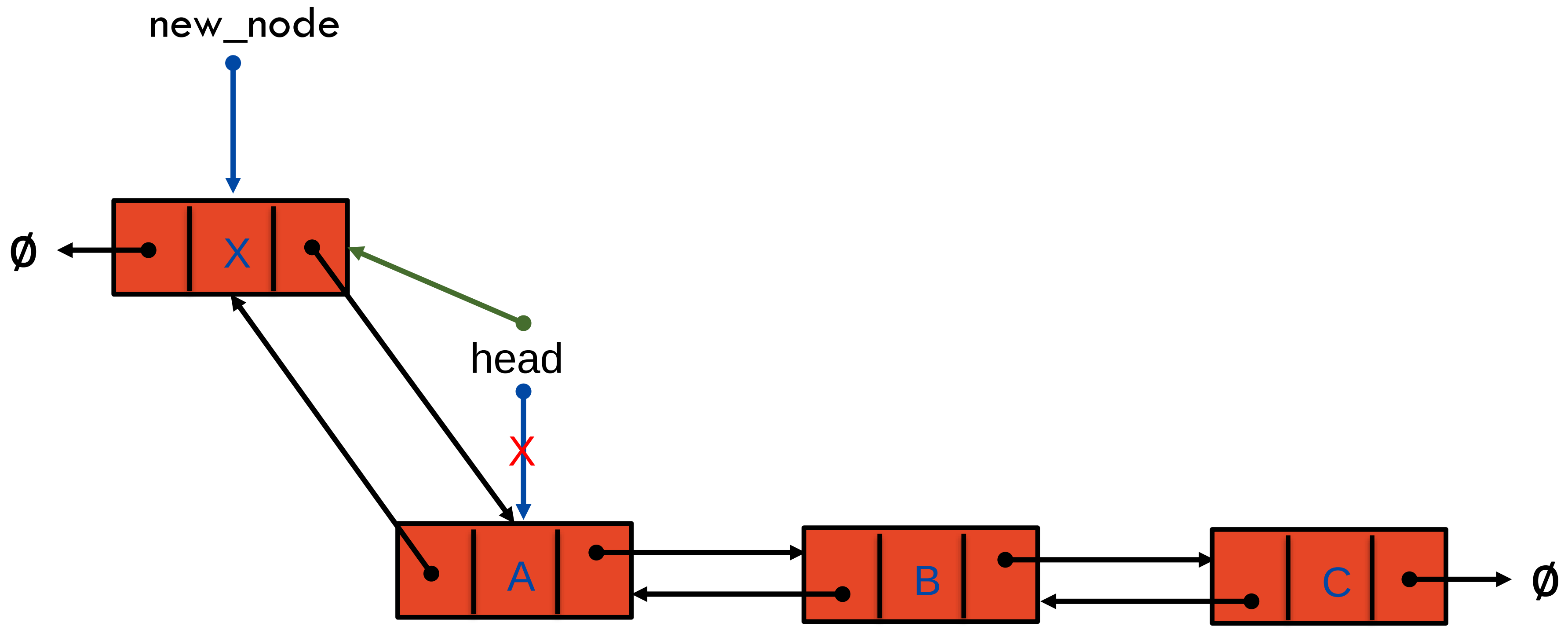
    head.prev = new_node



Insert at the Beginning (Step 04)

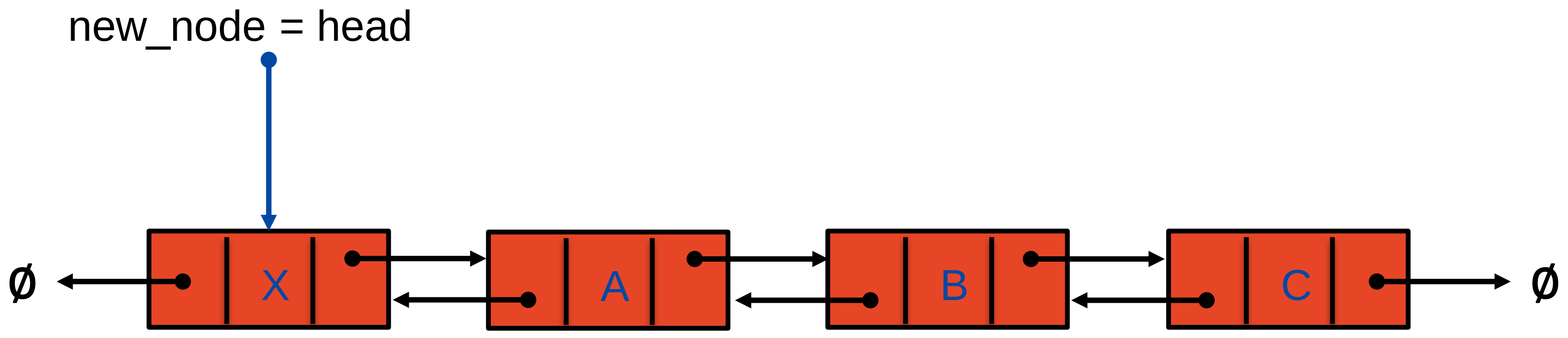
4. Move the head to point to the new node

    head = new_node



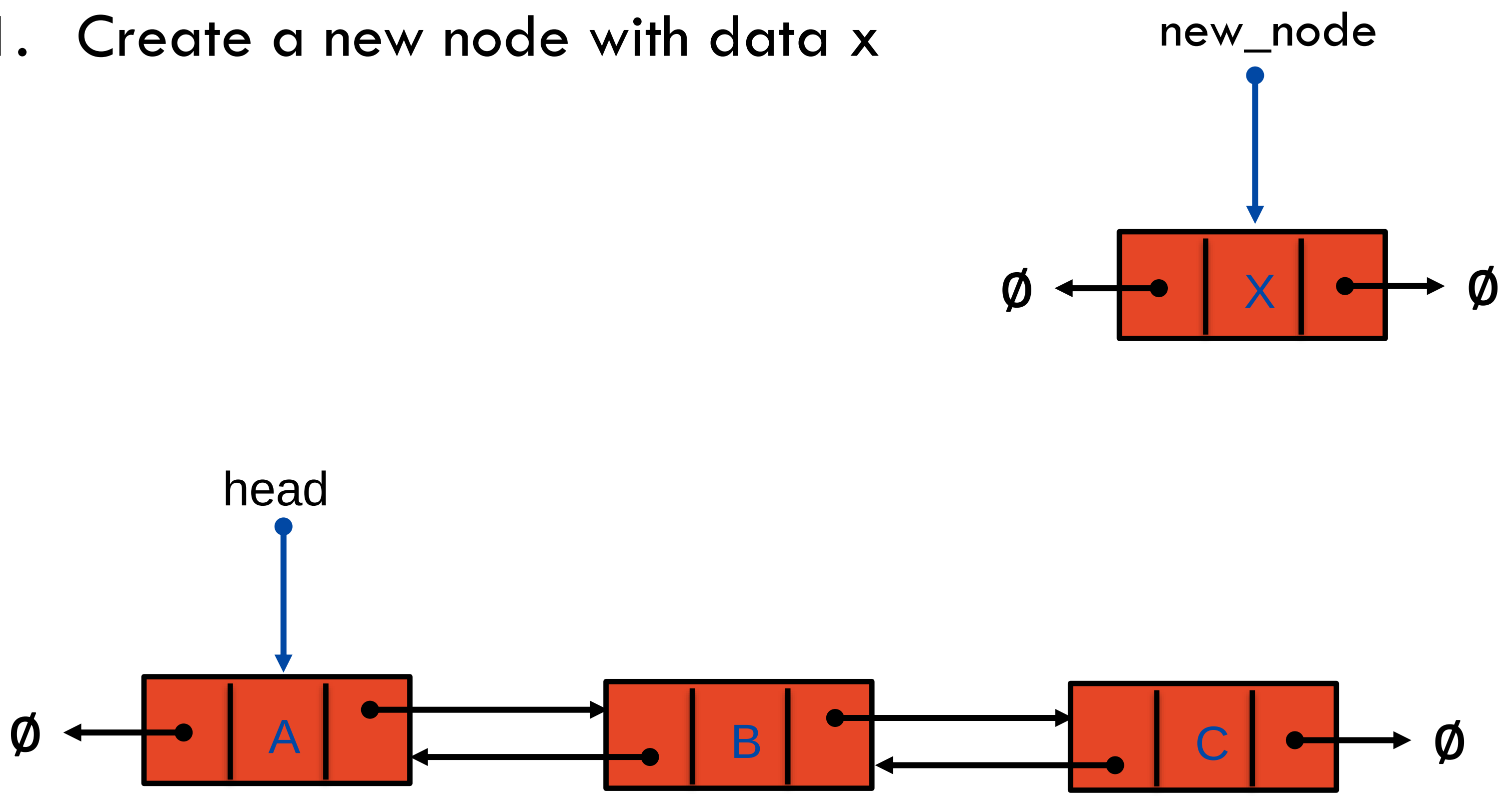
Insert at the Beginning (Result)

1. Create a new node with data x
2. Set the next pointer of new node to the current head
3. Update the previous pointer of the head to the new node
4. Move the head to point to the new node



Insert at the End (Step 01)

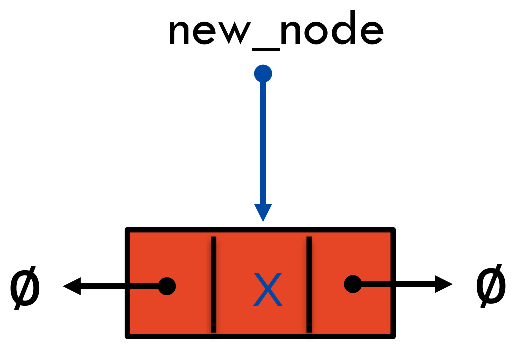
1. Create a new node with data x

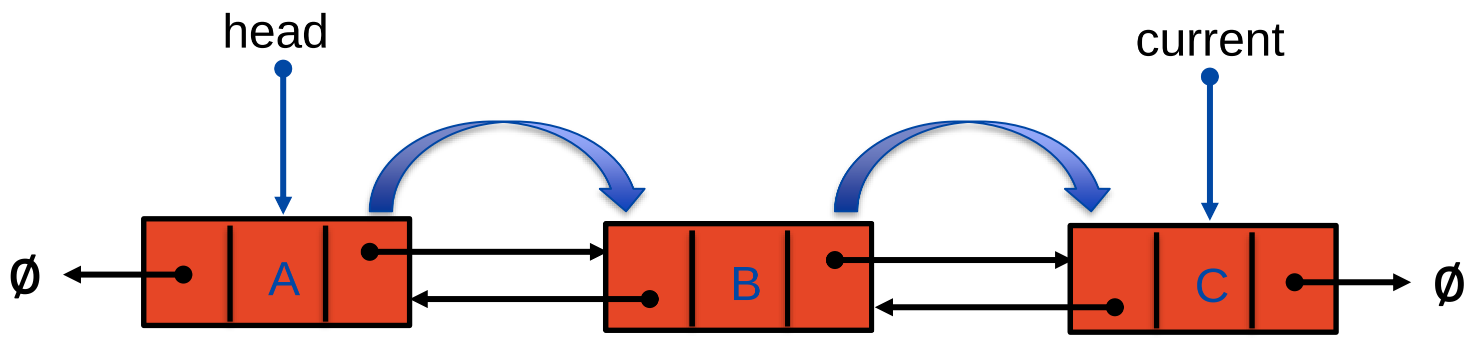


Insert at the End (Step 02)

2. Traverse the list until the last node is reached

```
current = head
while (current.next != None) {
      current = current.next
}
```

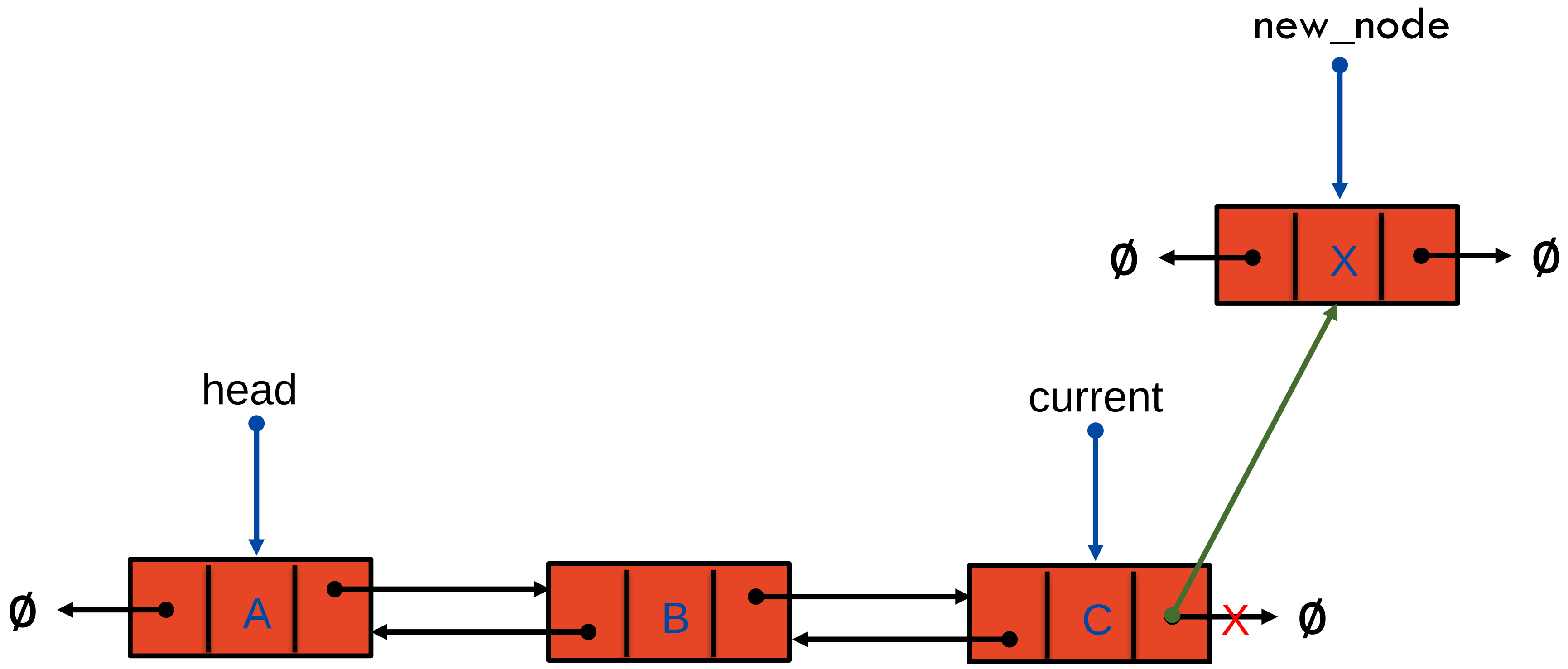




Insert at the End (Step 03)

3. Set the next pointer of last node to point to the new node

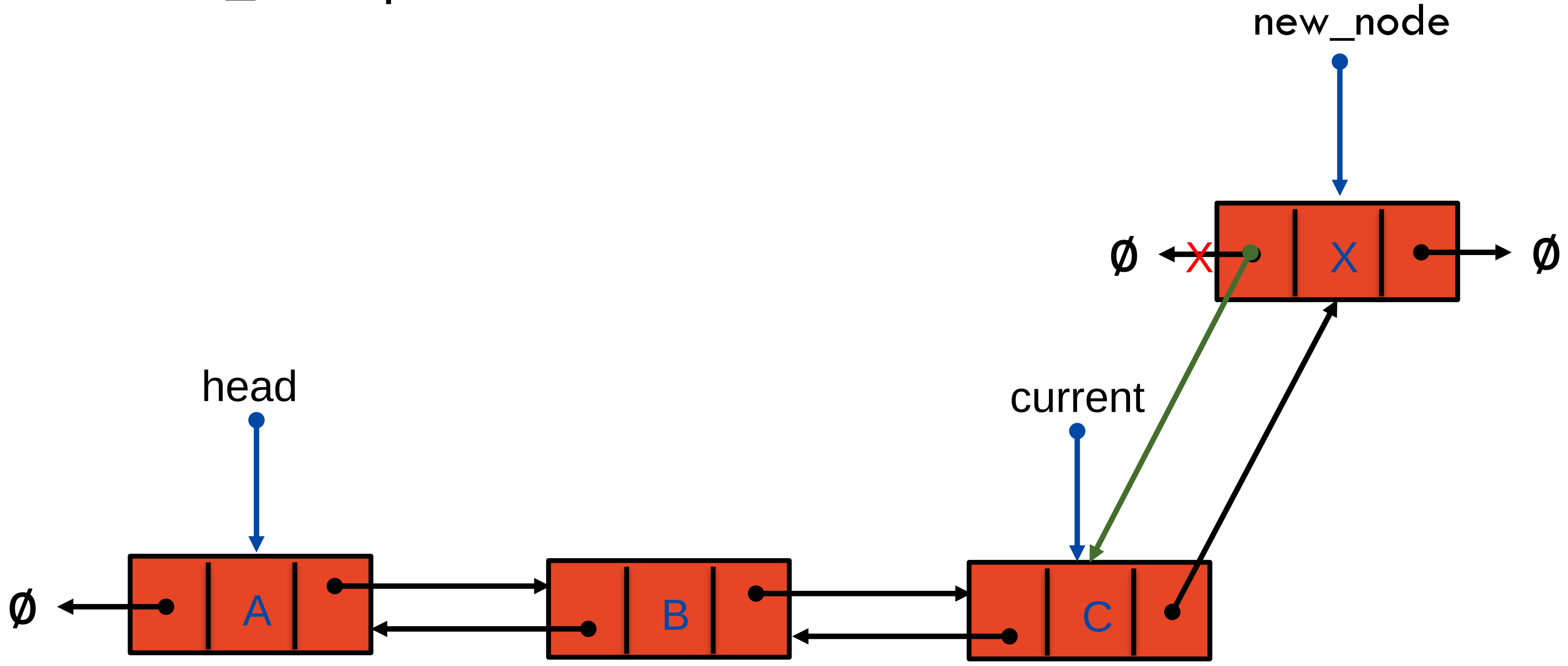
    current.next = new_node



Insert at the End (Step 04)

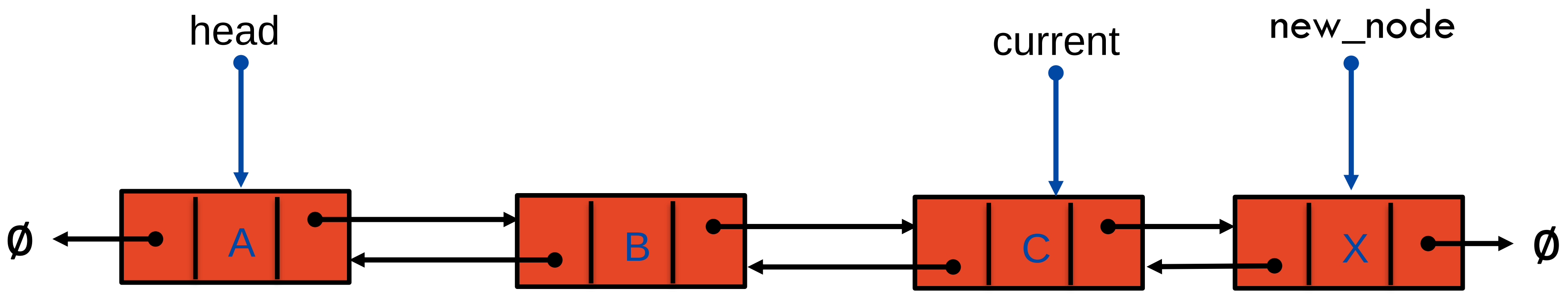
4. Set the previous pointer of the new node to point to the last node

    new_node.prev = current



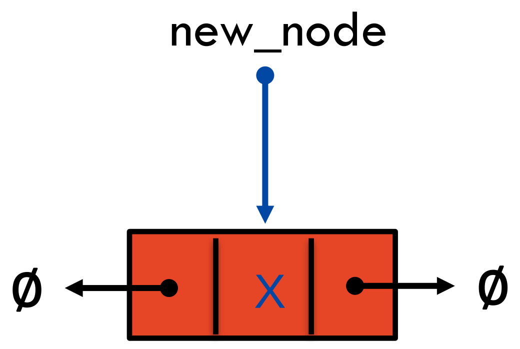
Insert at the End (Result)

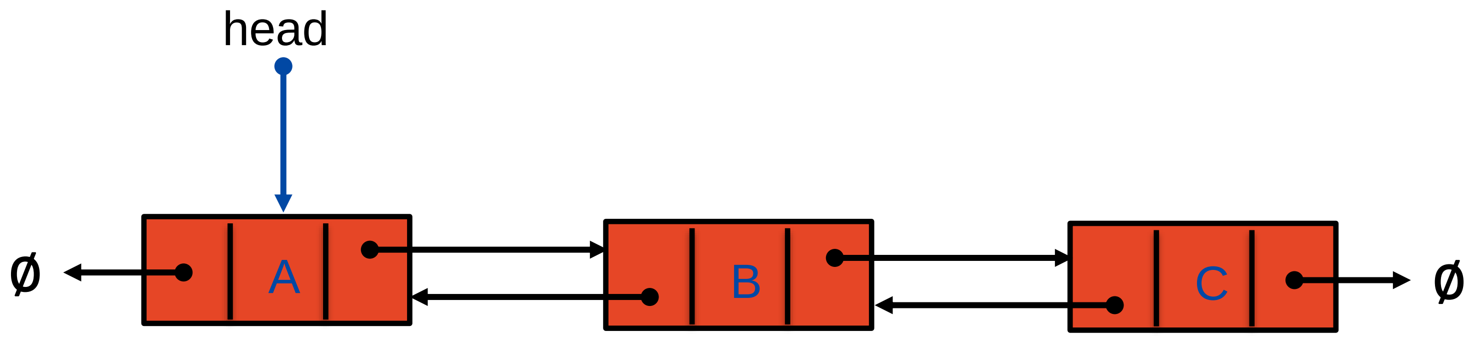
1. Create a new node with data x
2. Traverse the list until the last node is reached
3. Set the next pointer of last node to point to the new node
4. Set the previous pointer of the new node to point to the last node



Insert at a Specific Position (Step 01)

1. Create a new node with data x

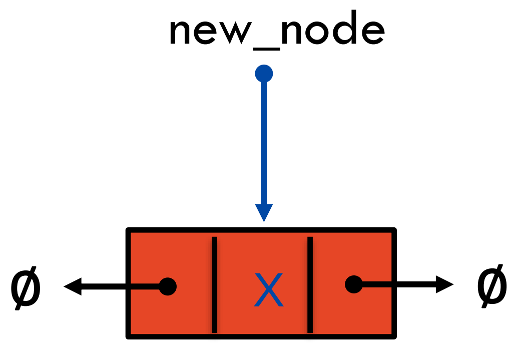


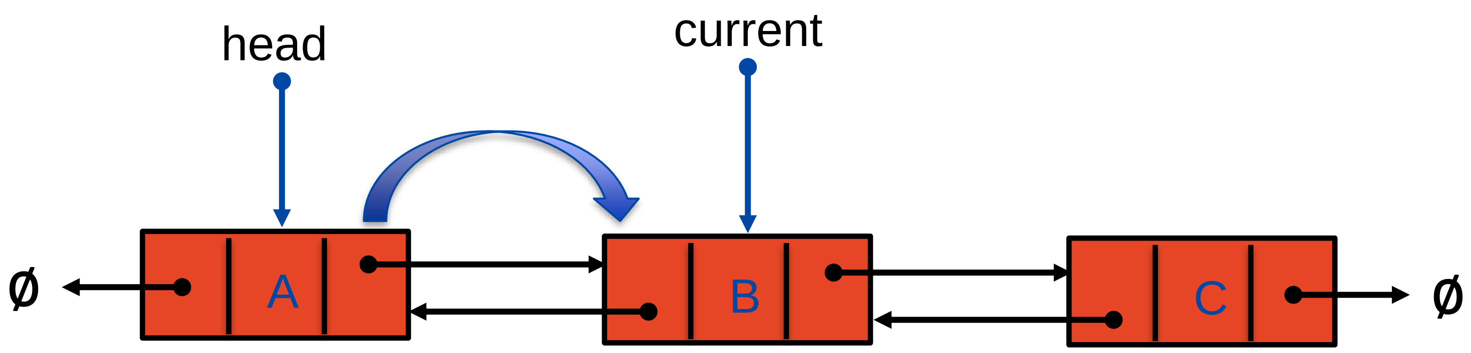


Insert at a Specific Position (Step 02)

2. Traverse the list to the desired position (i)

```
current = head
int count = 0
while (count < i-1 AND current != None) {
    current = current.next
     count++
}
```

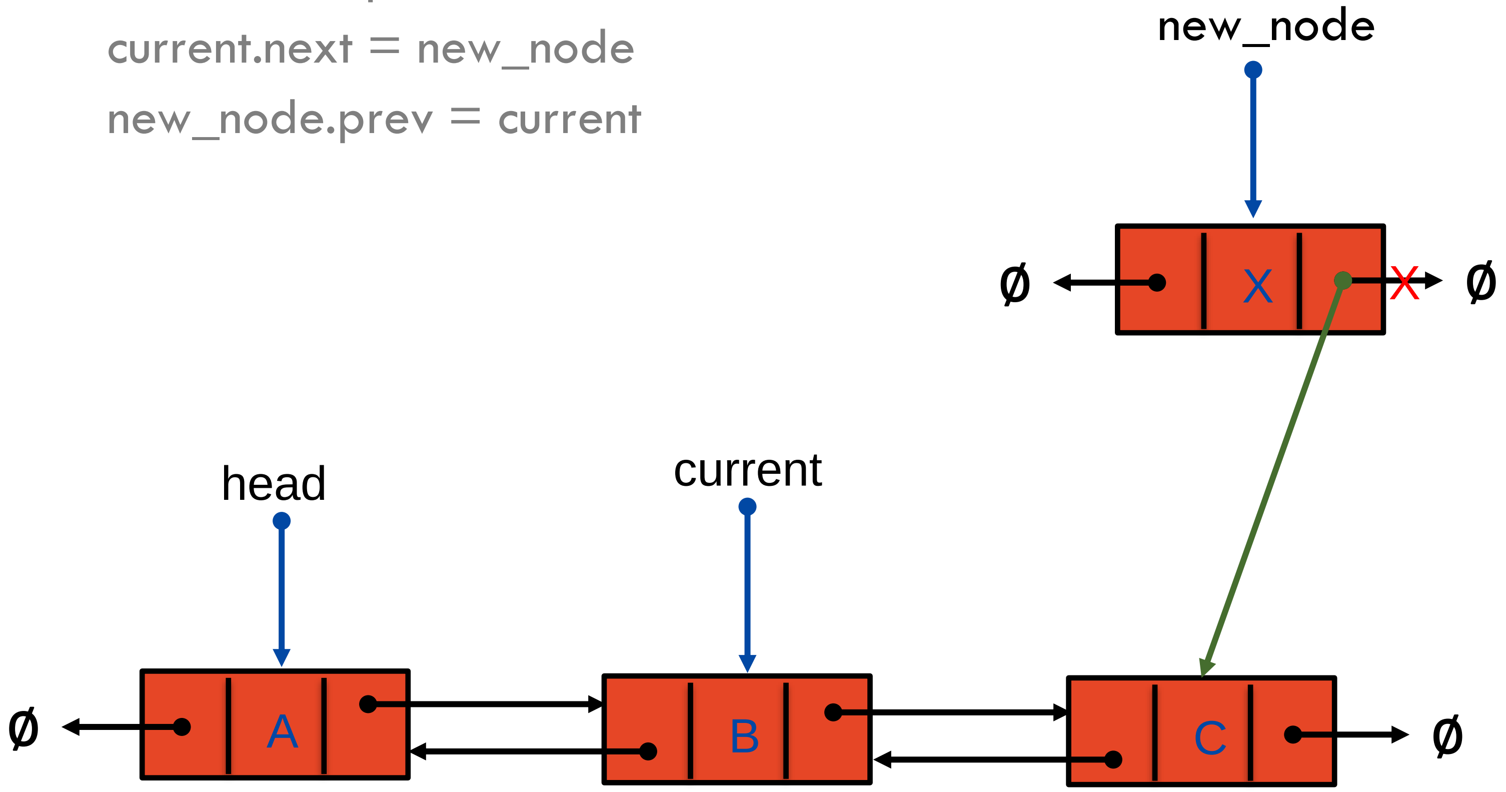




Insert at a Specific Position (Step 03)

3. Link the new node to the current last node

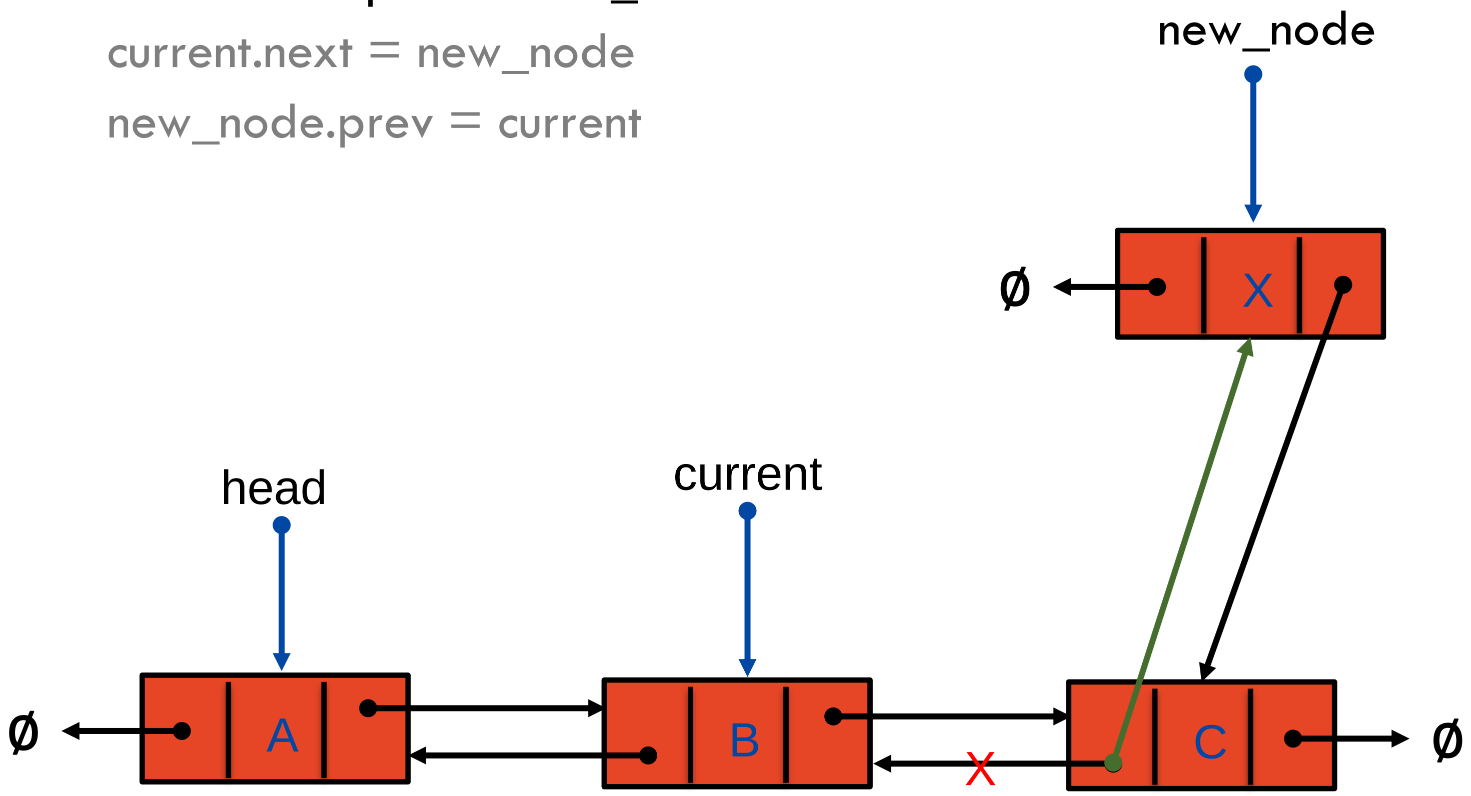
```
new_node.next = current.next
current.next.prev = new_node
current.next = new_node
new_node.prev = current
```



Insert at a Specific Position (Step 03)

3. Link the new node to the current last node

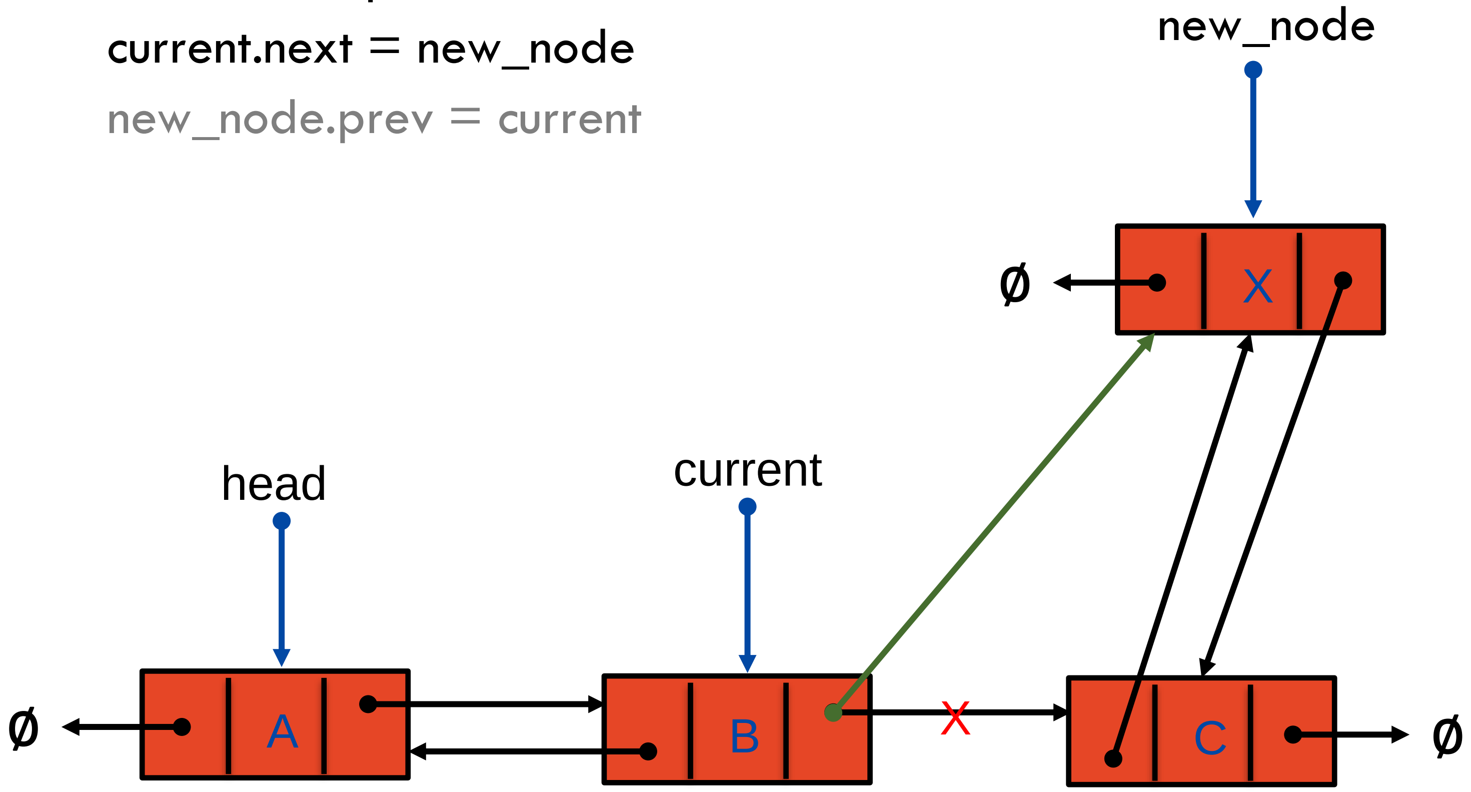
```
new_node.next = current.next
current.next.prev = new_node
current.next = new_node
new_node.prev = current
```



Insert at a Specific Position (Step 03)

3. Link the new node to the current last node

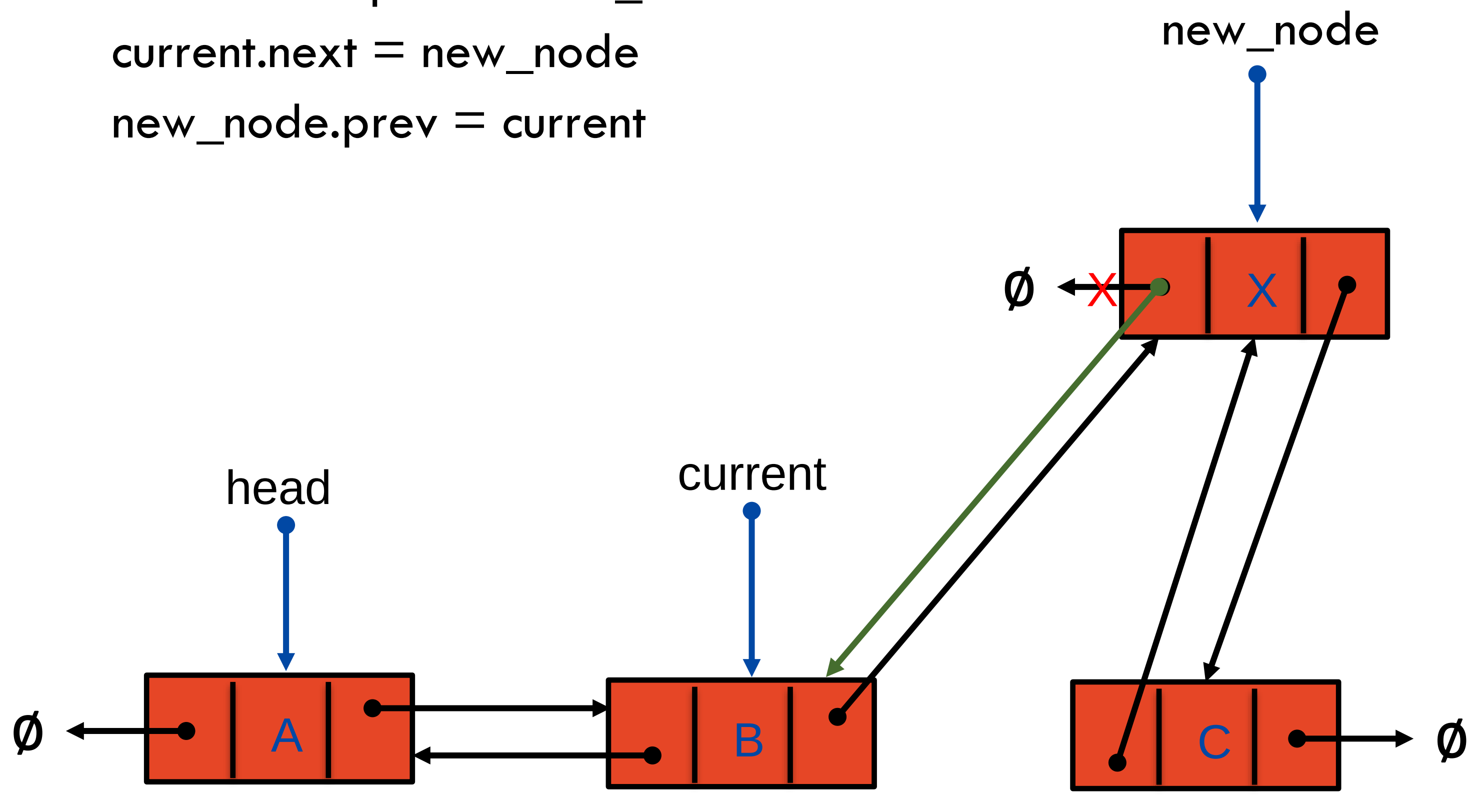
```
new_node.next = current.next
current.next.prev = new_node
current.next = new_node
new_node.prev = current
```



Insert at a Specific Position (Step 03)

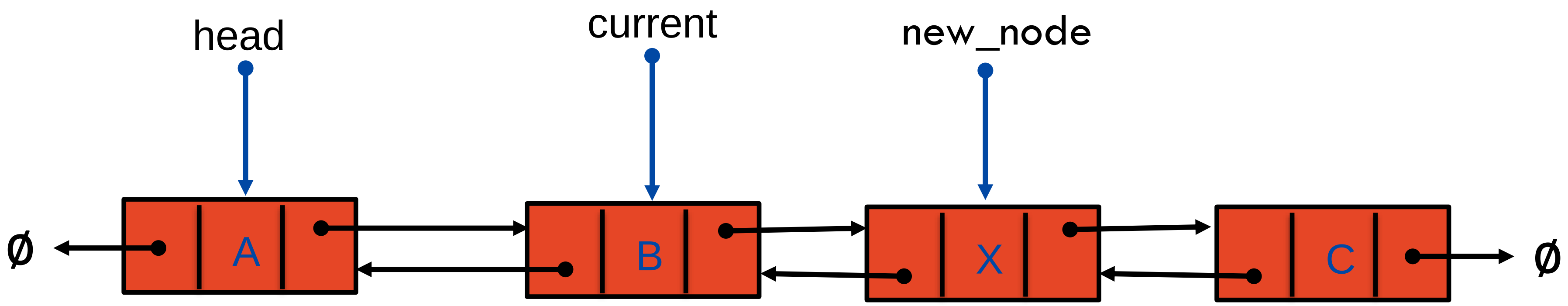
3. Link the new node to the current last node

```
new_node.next = current.next
current.next.prev = new_node
current.next = new_node
new_node.prev = current
```



Insert at a Specific Position (Result)

1. Create a new node with data x
2. Traverse the list to the desired position (i)
3. Link the new node to the current last node



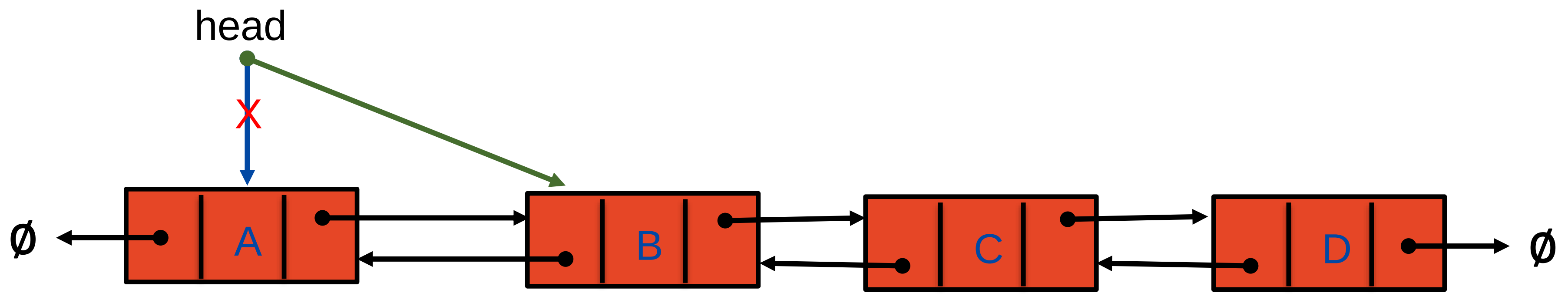
Deletion

Doubly Linked List

Deletion at the Beginning (Step 01)

1. Move the head pointer to the next node

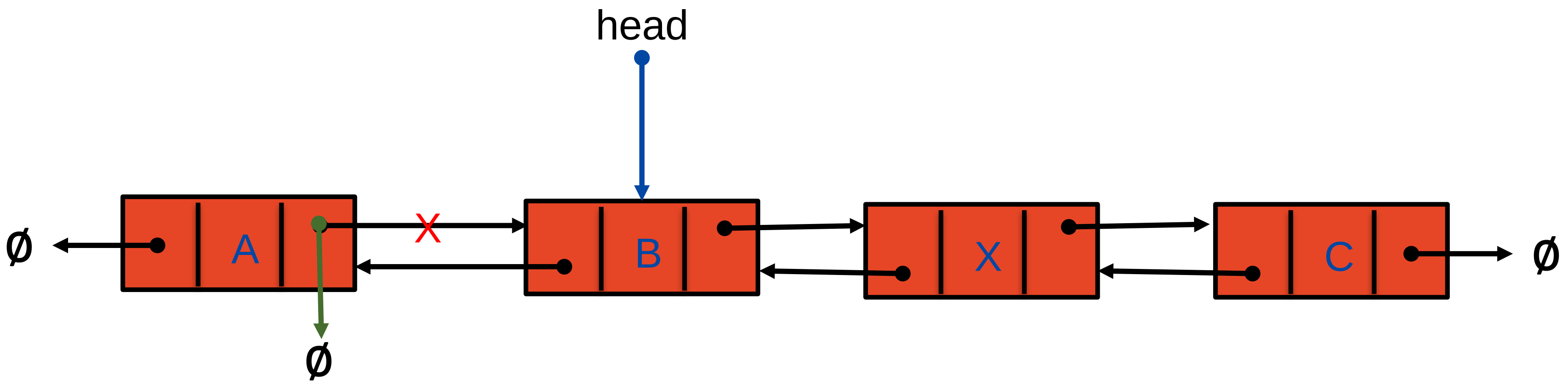
    head = head.next



Deletion at the Beginning (Step 02)

2. Remove the links from and to the first node

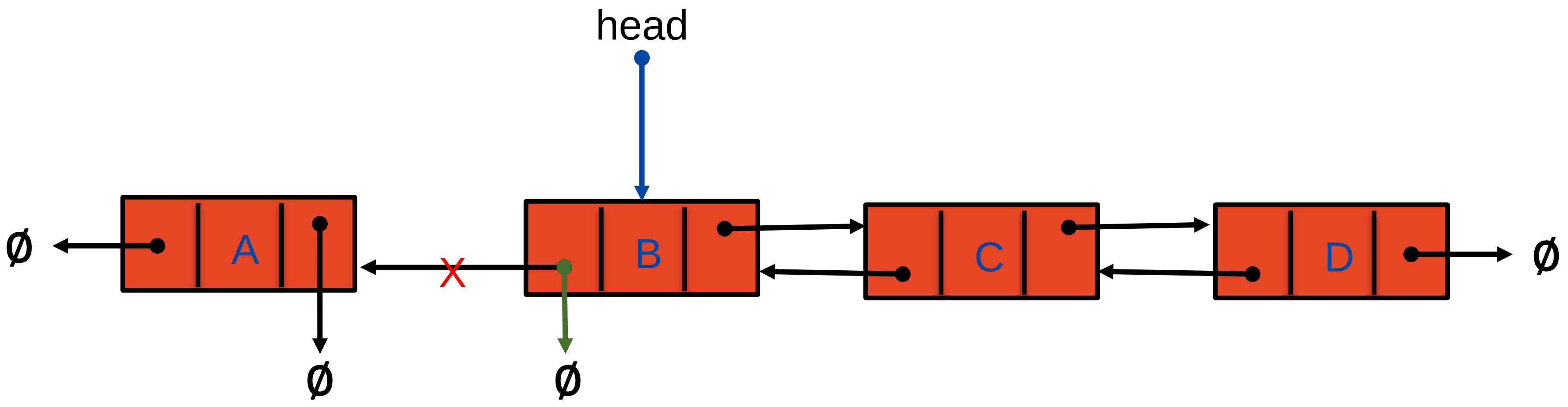
```
        head.prev.next = None
        head.prev = None
```



Deletion at the Beginning (Step 02)

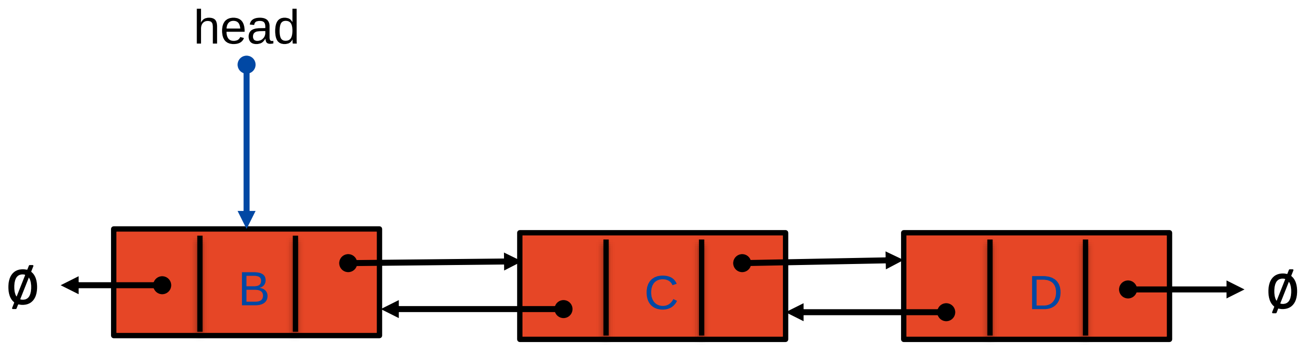
3. Remove the links from and to the first node

```
head.prev.next = None
head.prev = None
```



Deletion at the Beginning (Result)

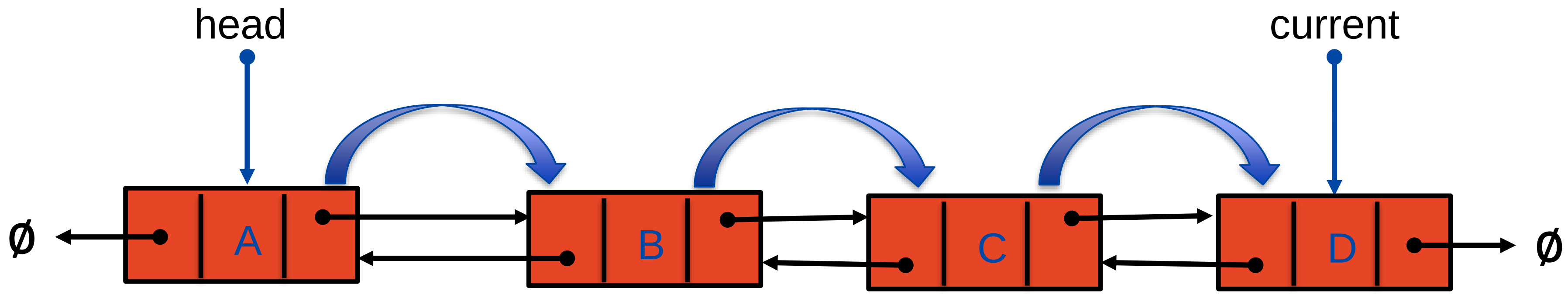
1. Move the head pointer to the next node
2. Remove the links from and to the first node



Deletion at the End (Step 01)

1. Traverse the list to find the last node

```
current = head
while (current.next != None){
        current = current.next
}
```

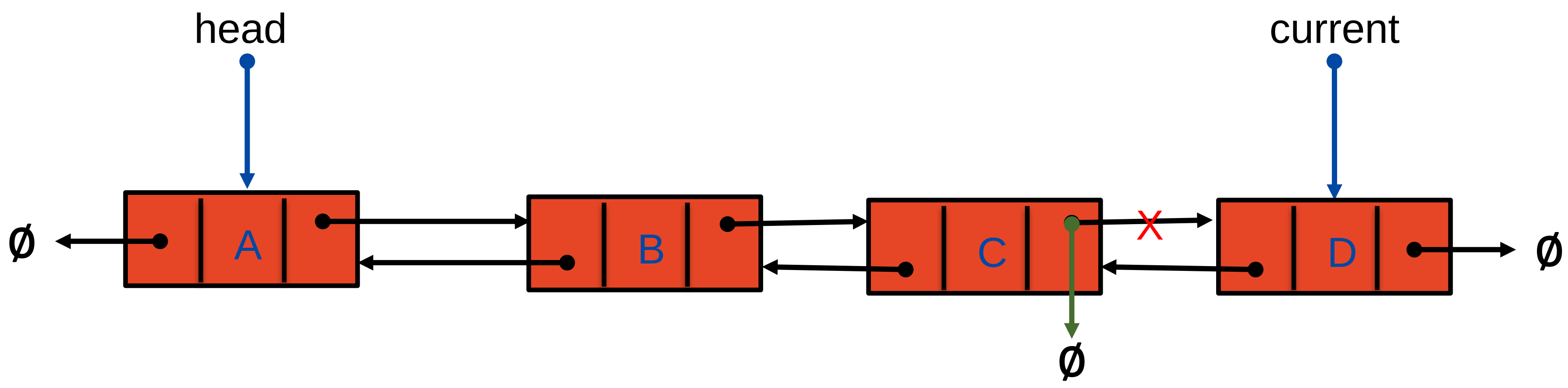


Deletion at the End (Step 02)

2. Remove the links from and to the current node

   current.prev.next = None

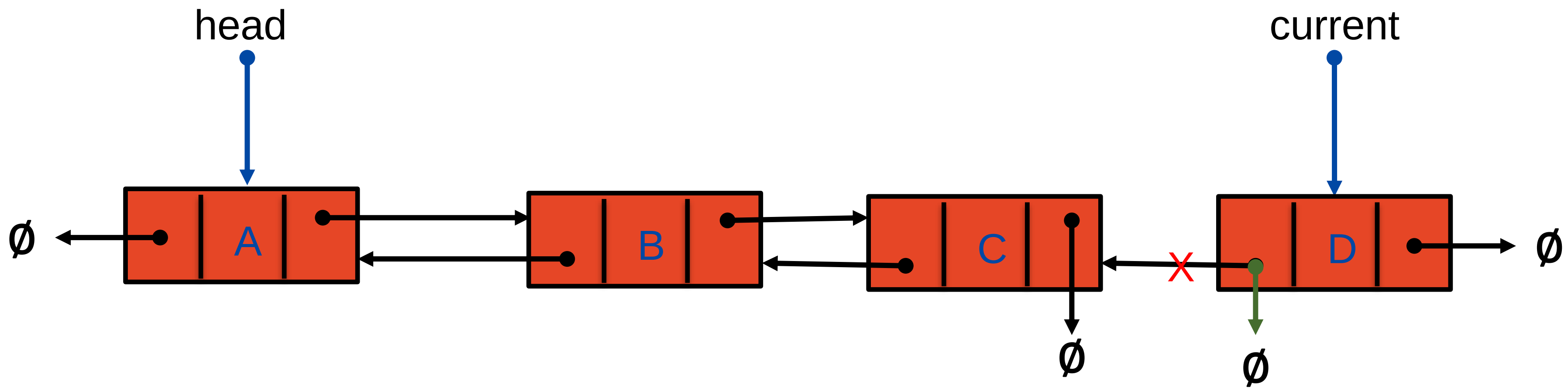
   current.prev = None



Deletion at the End (Step 02)

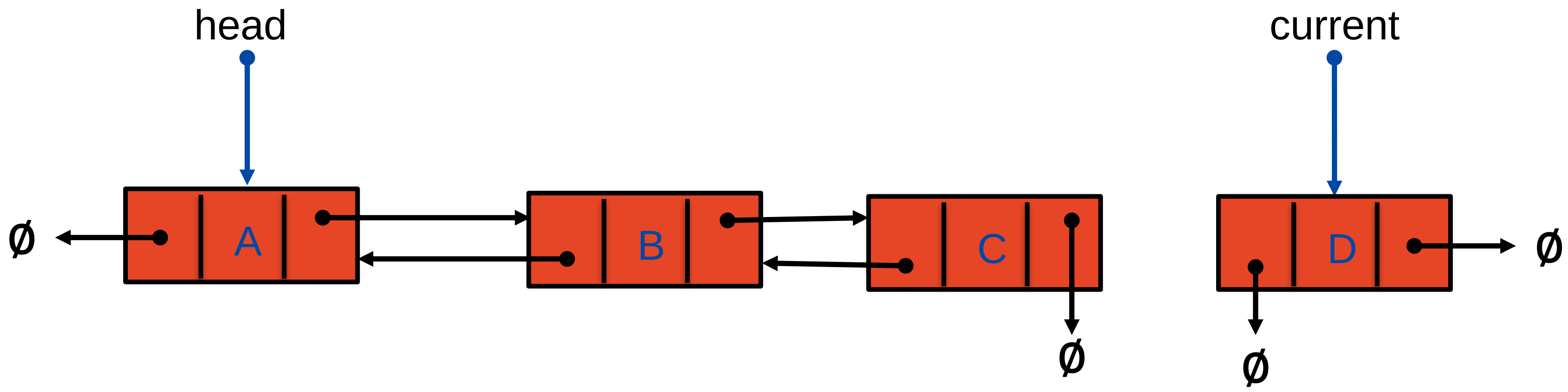
2. Remove the links from and to the current node

```
current.prev.next = None
current.prev = None
```



Deletion at the End (Result)

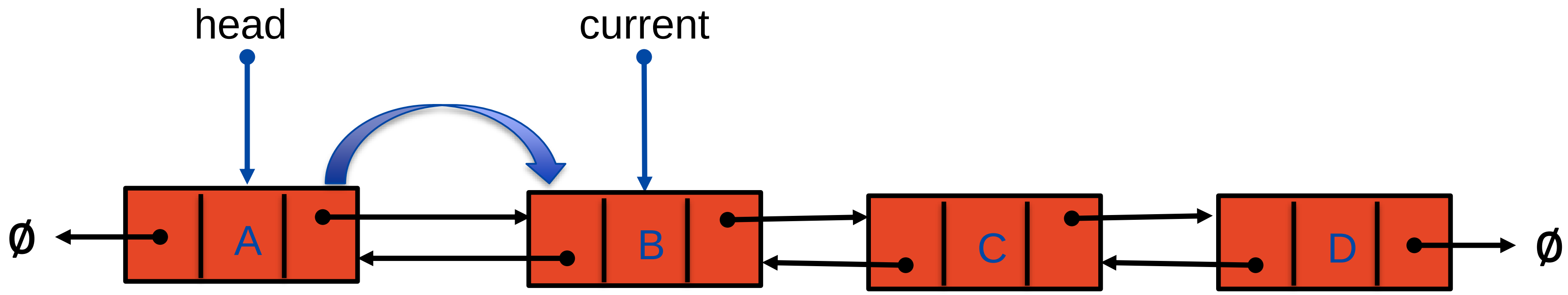
1. Traverse the list to find the last node
2. Remove the links from and to the current node



Deletion at Specific Position (Step 01)

1. Traverse to the node to be deleted

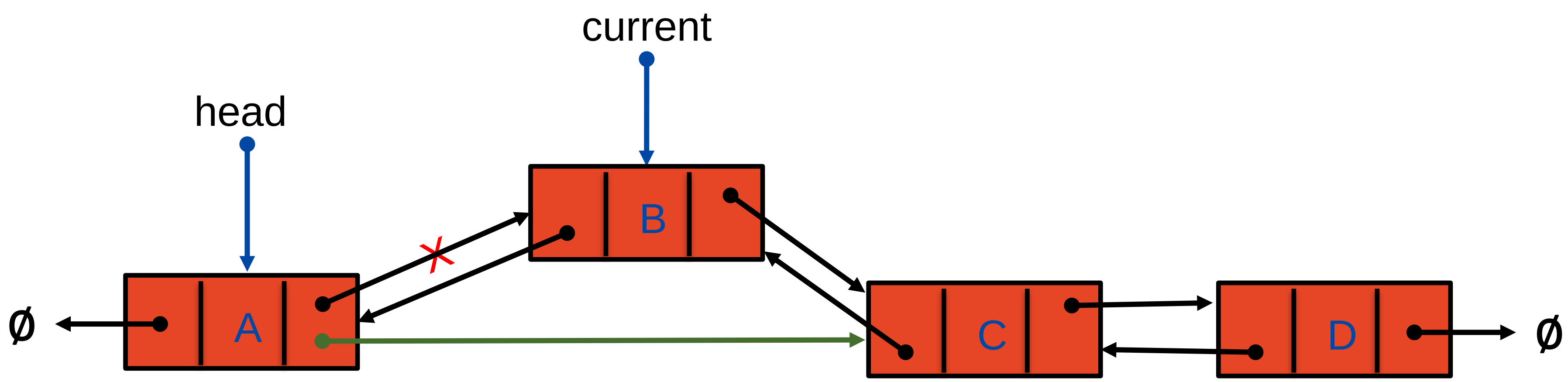
```
current = head
count = 0
while (count < index AND current != None){
    current = current.next
    count++
}
```



Deletion at Specific Position (Step 02)

2. Remove the links from and to the current node

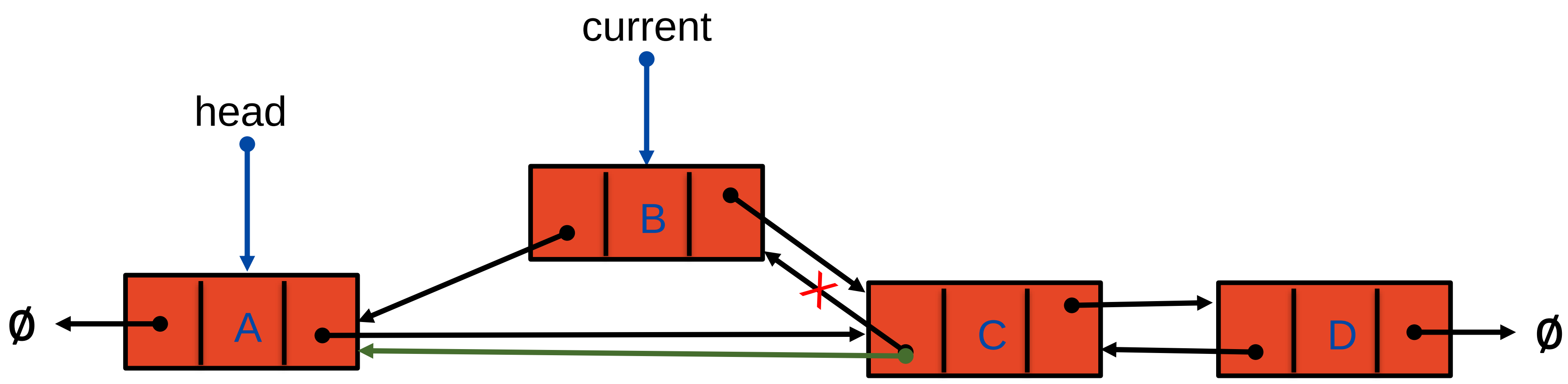
```
current.prev.next = current.next
current.next.prev = current.prev
current.prev = None
current.next = None
```



Deletion at Specific Position (Step 02)

2. Remove the links from and to the current node

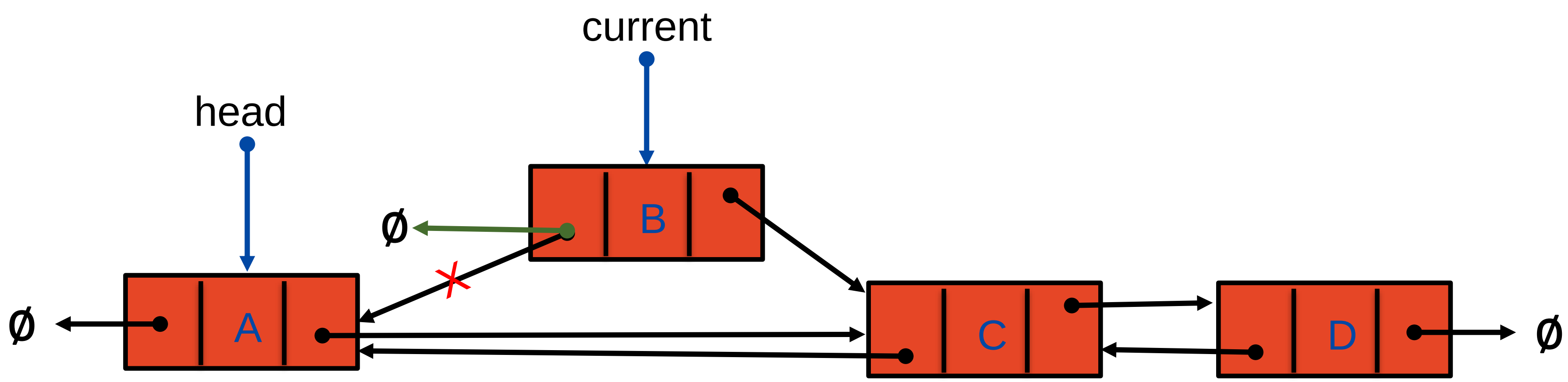
```
    current.prev.next = current.next
    current.next.prev = current.prev
    current.prev = None
    current.next = None
```



Deletion at Specific Position (Step 02)

2. Remove the links from and to the current node

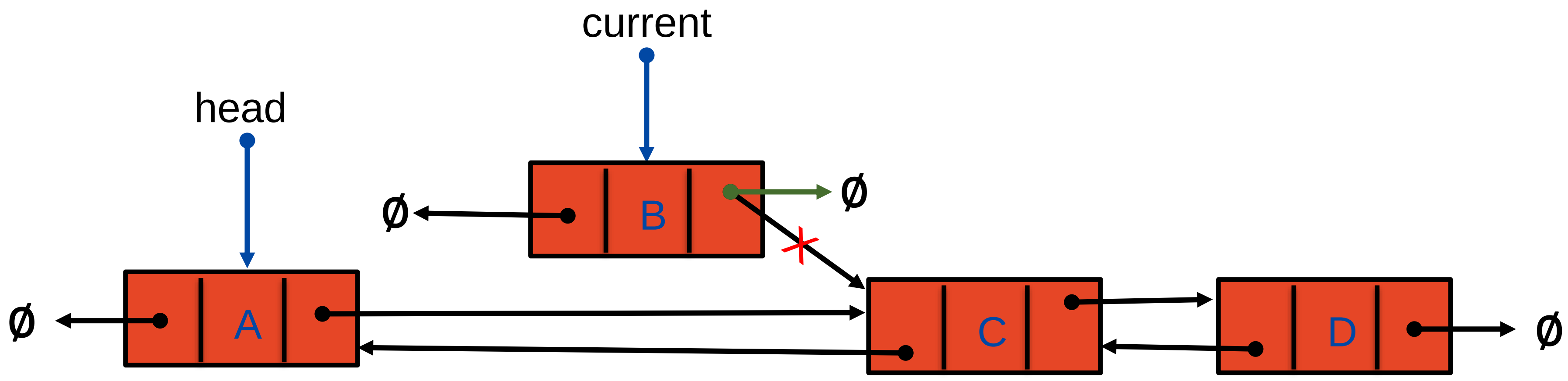
```
current.prev.next = current.next
current.next.prev = current.prev
current.prev = None
current.next = None
```



Deletion at Specific Position (Step 02)

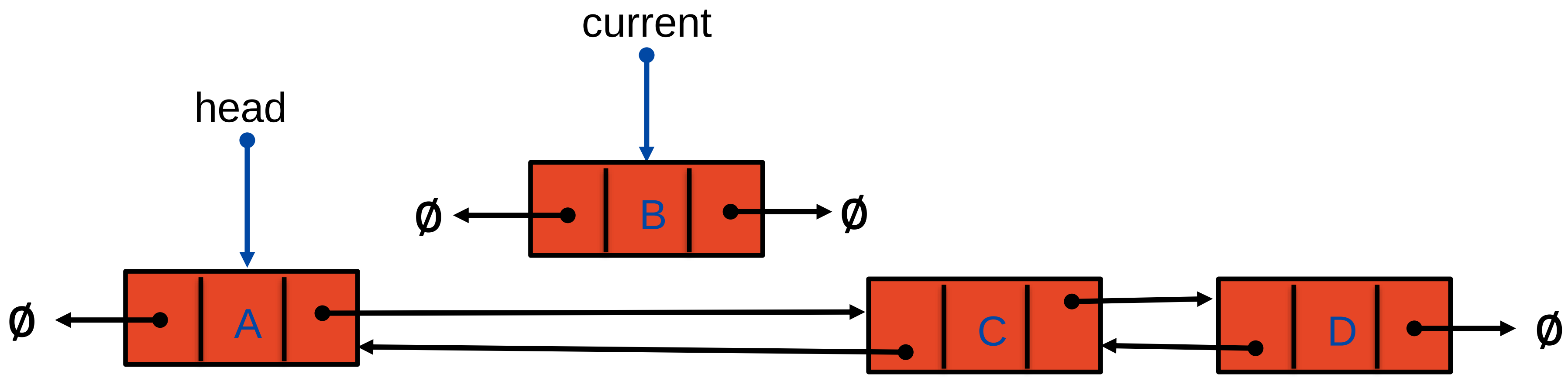
2. Remove the links from and to the current node

```
current.prev.next = current.next
current.next.prev = current.prev
current.prev = None
current.next = None
```



Deletion at Specific Position (Result)

1. Traverse to the node to be deleted
2. Remove the links from and to the current node



Traversal

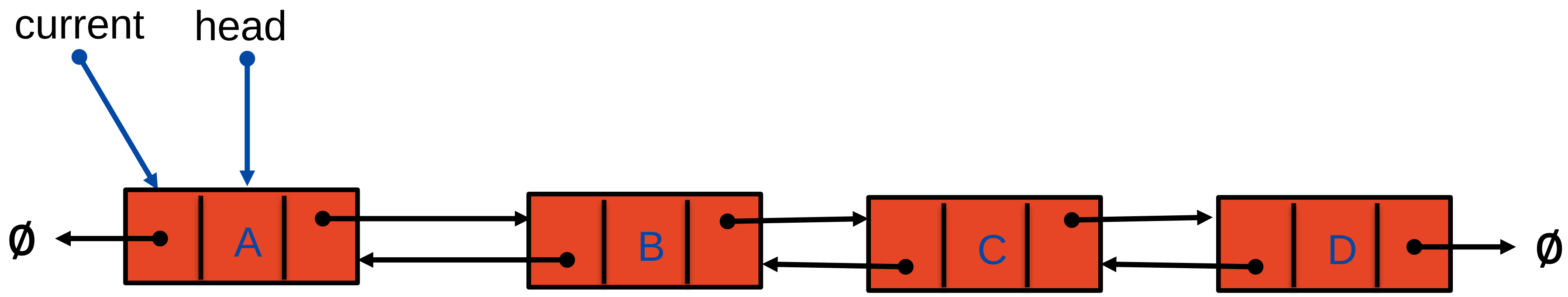
Doubly Linked List

THE UNIVERSITY OF SYDNEY

Traversal (Step 01)

1. Store the current head node in a temporary variable

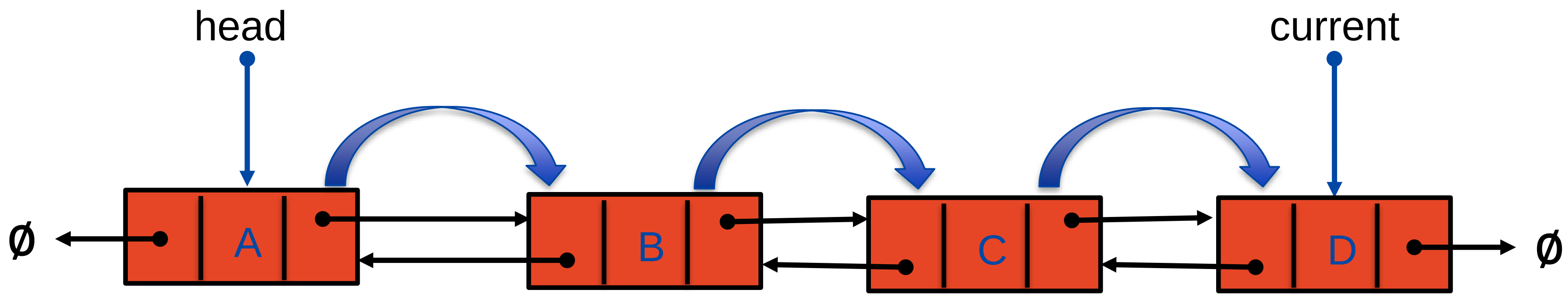
   current = head



Traversal (Step 02)

2. Traverse the list until the last node is reached

```
current = head
while (current != None) {
    print(current.data)
    current = current.next
}
```


head
current
Ø
A
B
C
D
Ø