NANYANG TECHNOLOGICAL UNIVERSITY

Data Structures & Algorithms in Python

Lecture 03 – Linked List: Variations

Dr. Owen Noel Newton Fernando

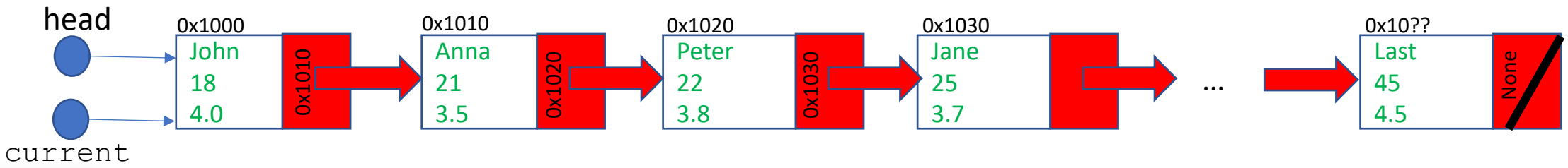
College of Engineering

School of Computer Science and Engineering

Topics

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Display each element in the linked list



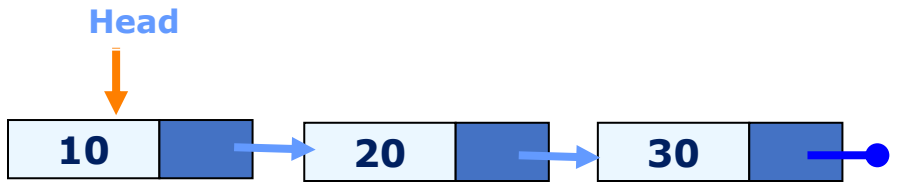

```
1def display(self):
2    current = self.head
3    while current:
4         print(current.data, end=" -> ")
5         current = current.next
6     print("None")
```

Display each element in the linked list

- Given the head pointer of the linked list
- Print all items in the linked list
- From first node to the last node

Display each element in the linked list

```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.next = None
5
6  class LinkedList:
7      def __init__(self):
8          self.head = None
9
10 # Standalone function taking head as parameter
11 def display(head):
12     current = head
13     while current:
14         print(current.data, end=" -> ")
15         current = current.next
16     print("None")
```




```
17 if __name__ == "__main__":
18
19     # Initialize empty linked list object
20     linked_list = LinkedList()
21
22     # Create nodes
23     node1 = Node(10)
24     node2 = Node(20)
25     node3 = Node(30)
26
27     # Link nodes
28     linked_list.head = node1
29     node1.next = node2
30     node2.next = node3
31
32     # Print the linked list
33     # Passes linked_list.head as argument
34     display(linked_list.head)
```

`linked_list.head` is the reference to the first node (`node1`), which acts as the starting point of the linked list.

Display each element in the linked list

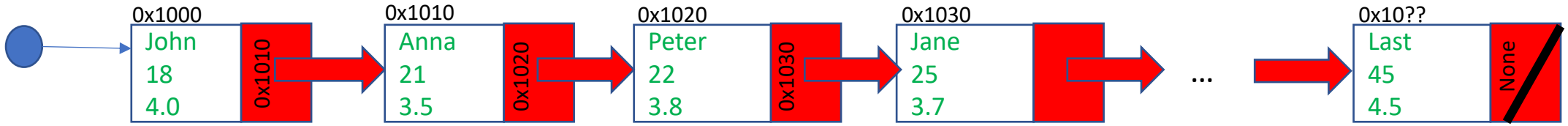
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

 # Made display a method of LinkedList
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
```

```
if __name__ == "__main__":
    # Initialize empty linked list object
    linked_list = LinkedList()

    # Create nodes
    node1 = Node(10)
    node2 = Node(20)
    node3 = Node(30)

    # Link nodes
    linked_list.head = node1
    node1.next = node2
    node2.next = node3

    # Call display method to print:
    # 10 -> 20 -> 30 -> None
    linked_list.display()
```

The key difference in this version is that display is a method of the LinkedList class:

- It's defined inside the **`LinkedList`** class.
- Being part of the class, it uses `self` to access the `LinkedList` instance's attributes, specifically `self.head`.
- The display method directly accesses the head of the linked list using **`self.head`**, which is the appropriate way to access the head of the current LinkedList instance.

Search the node at index i




```
1 def findAt(self, index):
2         current = self.head
3         if not current:
4             return None
5         while index>0:
6             current = current.next
7             if not current:
8                 return None
9             index-=1
10        return current
11
12
```

Search the node at index i

```
1   class Node:
2       def __init__(self, data):
3           self.data = data
4           self.next = None
5
6   class LinkedList:
7       def __init__(self):
8           self.head = None
9
10  # Independent function that takes
11  # head and index
12  def findAt(head, index):
13      current = head
14      if not current:
15          return None
16      while index > 0:
17          current = current.next
18          if not current:
19              return None
20          index -= 1
21      return current
```

```
22  if __name__ == "__main__":
23
24      linked_list = LinkedList()
25
26      node1 = Node(10)
27      node2 = Node(20)
28      node3 = Node(30)
29
30      linked_list.head = node1
31      node1.next = node2
32      node2.next = node3
33
34      # Find node at index 2
35      found_node = findAt(linked_list.head, 2)
36      if found_node:
37          print(f"Node at index 2: {found_node.data}")
38      else:
39          print("Index not found")
```

`linked_list.head` is the reference to the first node (`node1`), which acts as the starting point of the linked list.

Search the node at index i

```
1   class Node:
2       def __init__(self, data):
3           self.data = data
4           self.next = None
5
6   class LinkedList:
7       def __init__(self):
8           self.head = None
9
10  # Made findAt a method of LinkedList
11      def findAt(self, index):
12          current = self.head
13          if not current:
14              return None
15          while index>0:
16              current = current.next
17              if not current:
18                  return None
19              index-=1
```

```
20  if __name__ == "__main__":
21
22      linked_list = LinkedList()
23
24      node1 = Node(10)
25      node2 = Node(20)
26      node3 = Node(30)
27
28      linked_list.head = node1
29      node1.next = node2
30      node2.next = node3
31
32      # Find node at index 2
33      found_node = linked_list.findAt(2)
34      if found_node:
35          print(f"Node at index 2: {found_node.data}")
36      else:
37          print("Index not found")
```

- When you call `linked_list.findAt (2)`, Python automatically passes the instance of the class `linked_list` as the first parameter `self` when calling the method.
- Through the `self` parameter, `findAt` accesses `self.head`, which is the reference to the start of the linked list. This access is necessary to begin traversing the list.

Insert node at given index

```
def insert(self, data, index):
    # Create a new node with the given data
    new_node = Node(data)

    # If list is empty or inserting at head
    if self.head is None or index == 0:
        new_node.next = self.head
        self.head = new_node
        return True

    # Start at the head of the list
    current = self.head
    count = 0

    # Traverse until index-1 position
    # (node before where we want to insert)
    while current and count < index - 1:
        current = current.next
        count += 1

    # If current is None, index was too large
    if not current:
        print("Index out of range")
        return False

    # Insert the new node by updating pointers:
    # 1. New node points to current's next node
    # 2. Current node points to new node
    new_node.next = current.next
    current.next = new_node
    return True
```

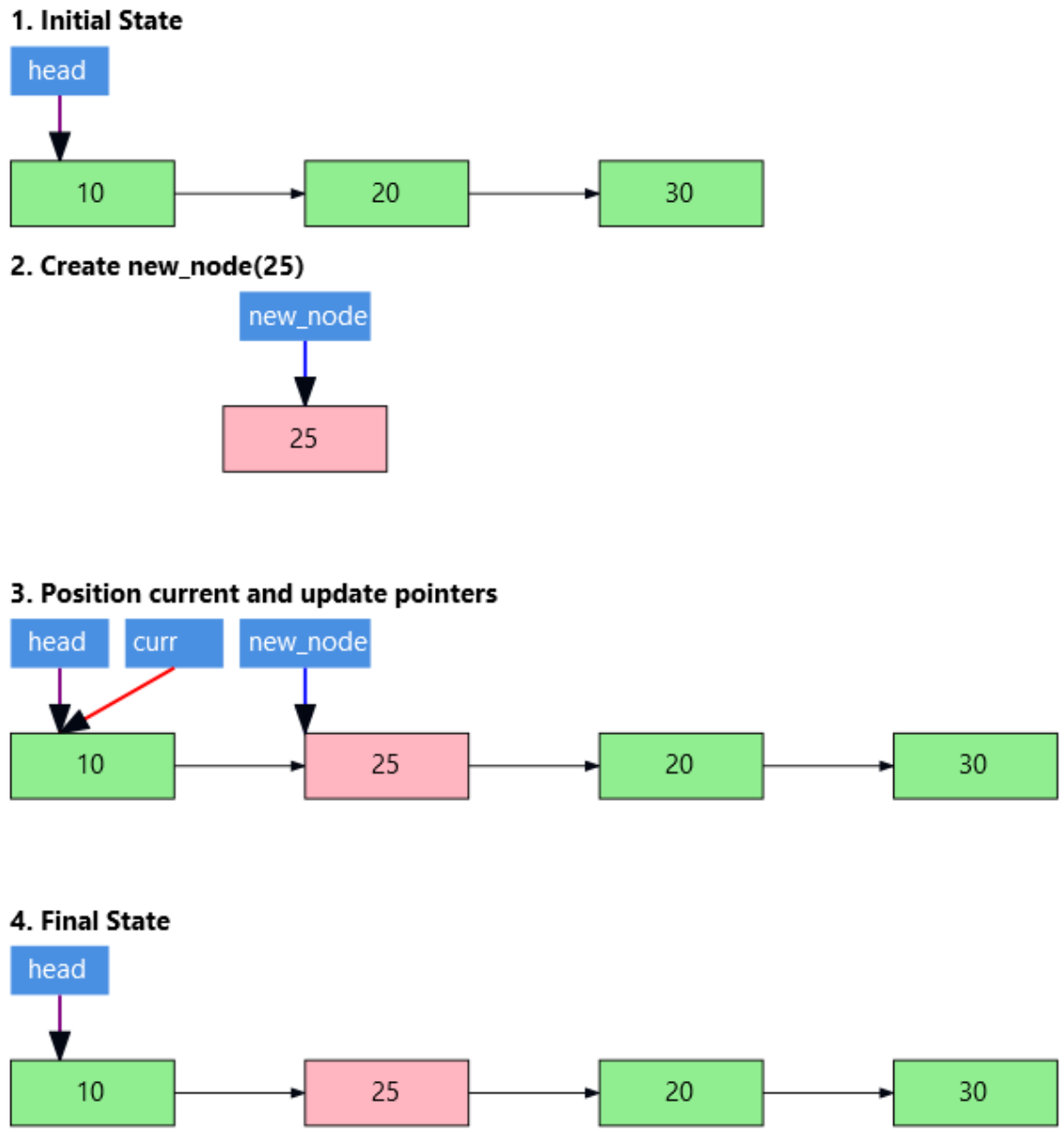
Insert a node at given index

```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.next = None
5
6  class LinkedList:
7      def __init__(self):
8          self.head = None
9
10     def display(self):
11         current = self.head
12         while current:
13             print(current.data, end=" -> ")
14             current = current.next
15         print("None")
```

```
16     def insert(self, data, index):
17         new_node = Node(data)
18
19         if self.head is None or index == 0:
20             new_node.next = self.head
21             self.head = new_node
22             return True
23
24         current = self.head
25         count = 0
26
27         while current and count < index - 1:
28             current = current.next
29             count += 1
30
31         if not current:
32             print("Index out of range")
33             return False
34
35         new_node.next = current.next
36         current.next = new_node
37         return True
```

Insert a node at given index

```
38 if __name__ == "__main__":
39
40     # Create and initialize the linked list
41     linked_list = LinkedList()
42     node1 = Node(10)
43     node2 = Node(20)
44     node3 = Node(30)
45
46     # Link nodes to form the initial list
47     linked_list.head = node1
48     node1.next = node2
49     node2.next = node3
50
51     print("Original list:")
52     linked_list.display()
53
54     # Insert 25 at index 1
55     linked_list.insert(25, 1)
56     print("After inserting 25 at index 1:")
57     linked_list.display()
```



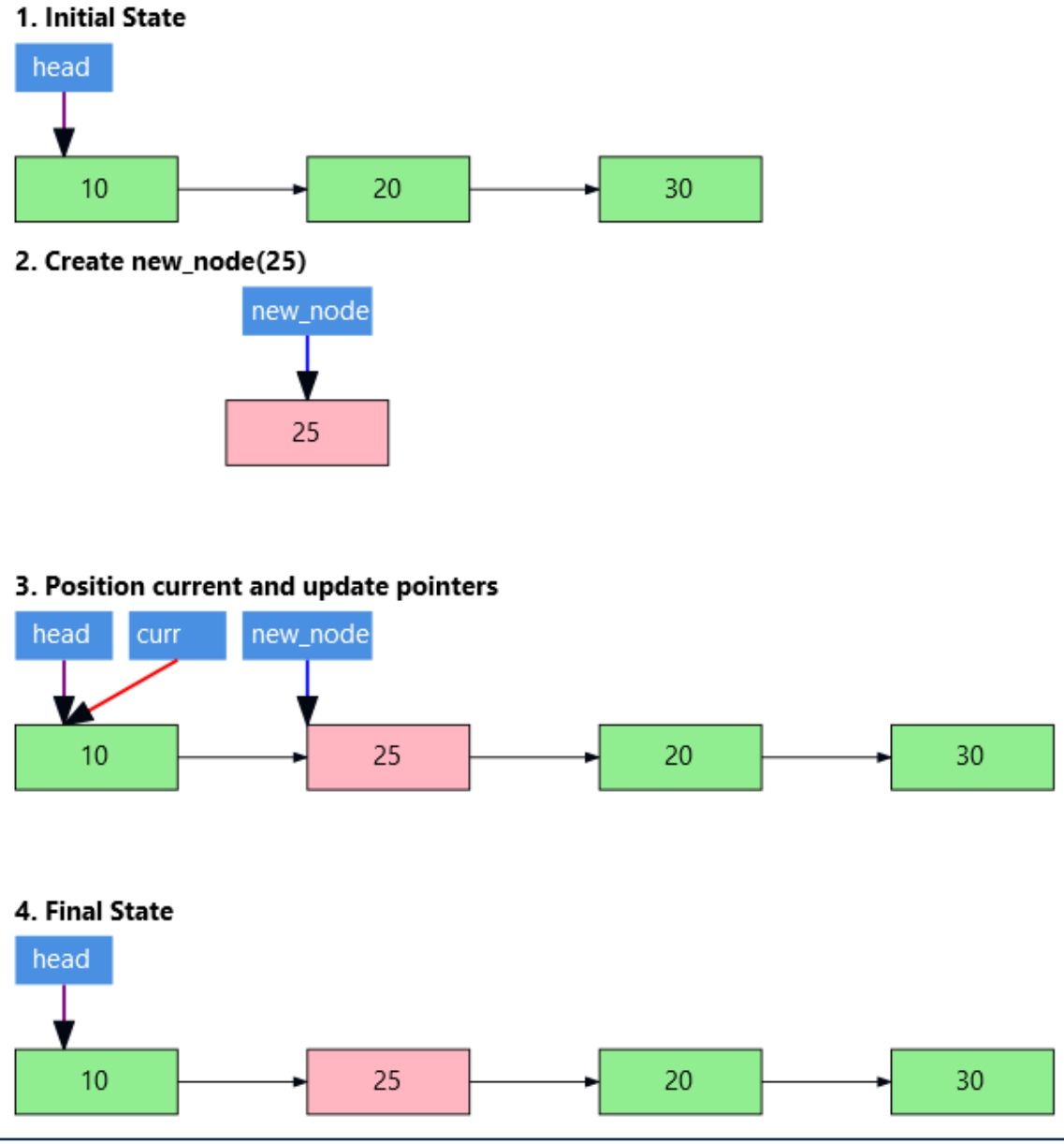
Insert a node at given index

```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.next = None
5
6  class LinkedList:
7      def __init__(self):
8          self.head = None
9
10     def display(self):
11         current = self.head
12         while current:
13             print(current.data, end=" -> ")
14             current = current.next
15         print("None")
```

```
16     def insert(self, data, index):
17         new_node = Node(data)
18
19         if self.head is None or index == 0:
20             new_node.next = self.head
21             self.head = new_node
22             return True
23
24         current = self.head
25         count = 0
26
27         while current and count < index - 1:
28             current = current.next
29             count += 1
30
31         if not current:
32             print("Index out of range")
33             return False
34
35         new_node.next = current.next
36         current.next = new_node
37         return True
```

Insert a node at given index

```
38 if __name__ == "__main__":
39
40     # Create and initialize the linked list
41     linked_list = LinkedList()
42     linked_list.insert(10, 0)
43     linked_list.insert(20, 1)
44     linked_list.insert(30, 2)
45
46     # Link nodes to form the initial list
47     # linked_list.head = node1
48     # node1.next = node2
49     # node2.next = node3
50
51     print("Original list:")
52     linked_list.display()
53
54     # Insert 25 at index 1
55     linked_list.insert(25, 1)
56     print("After inserting 25 at index 1:")
57     linked_list.display()
```



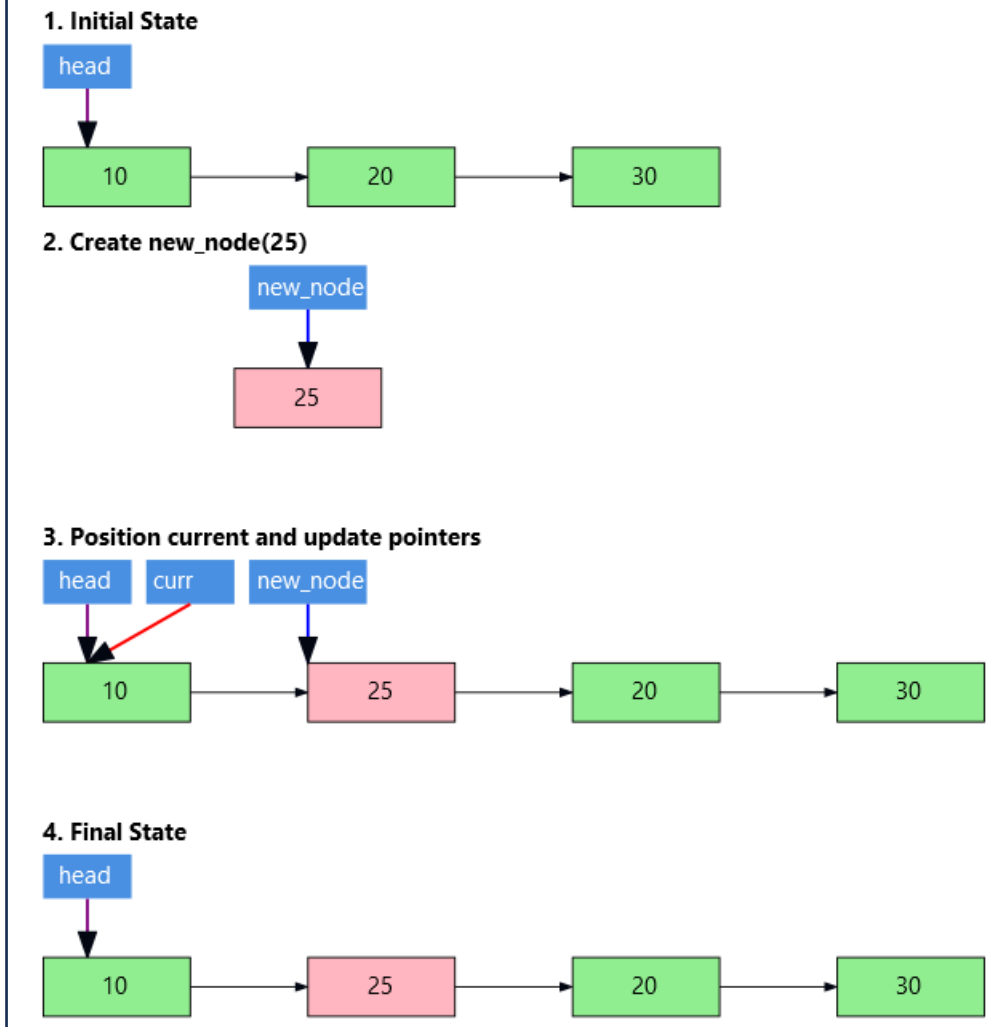
SizeList

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

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

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

    def sizeList(self):
        count = 0
        current = self.head
        while current is not None:
            count += 1
            current = current.next
        return count
```

```
    def insert(self, data, index):
        new_node = Node(data)

        if self.head is None or index == 0:
            new_node.next = self.head
            self.head = new_node
            return True

        current = self.head
        count = 0

        while current and count < index - 1:
            current = current.next
            count += 1

        if not current:
            print("Index out of range")
            return False

        new_node.next = current.next
        current.next = new_node
        return True
```

SizeList

```
46 if __name__ == "__main__":
47     # Create and initialize the linked list
48     linked_list = LinkedList()
49     linked_list.insert(10, 0)
50     linked_list.insert(20, 1)
51     linked_list.insert(30, 2)
52
53     # Display the original list
54     print("Current list:")
55     linked_list.display()
56
57     # Display initial size
58     print(f"Size of the list: {linked_list.sizeList()}")
59
60     # Insert 25 at index 1
61     linked_list.insert(25, 1)
62     print("\nAfter inserting 25 at index 1:")
63     linked_list.display()
64
65     # Display updated size
66     print(f"Size of list: {linked_list.sizeList()}")
```

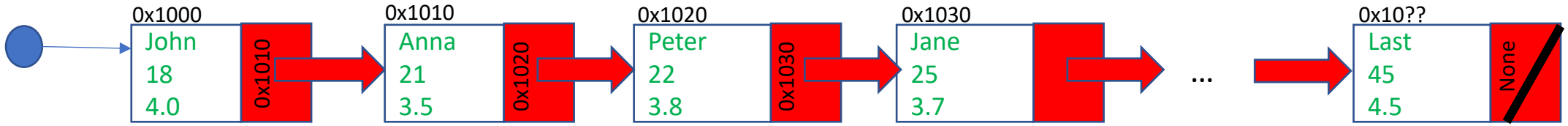


The Linked List

- Just introduce a new member in the linked list class, *size*
- Initialize *size* as zero
- When you add or remove a node, increase or decrease *size* by one accordingly

```
1   class ListNode:
2       def __init__(self, item):
3           self.item = item
4           self.next = None
5
6   class LinkedList:
7       def __init__(self):
8           self.head = None
9           self.size = 0
```

Search the node at index i



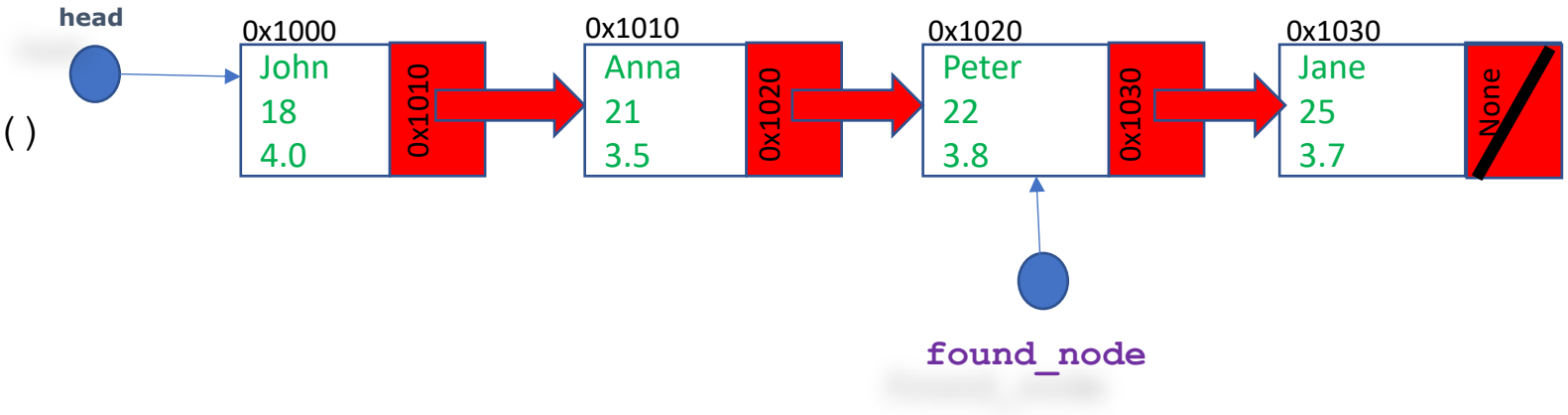

```
1 def findNode(self, index):
2         # Check if list is empty or index is invalid
3         if self.head is None or index < 0 or index >= self.size:
4             return None
5
6
7         # Start traversing from head
8         cur = self.head
9         while index > 0:
10            cur = cur.next
11            index -= 1
12        return cur
13
```

Search the node at index i

```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.next = None
5
6  class LinkedList:
7      def __init__(self):
8          self.head = None
9          self.size = 0
10
11 # Made findNode a method of LinkedList
12     def findNode(self, index):
13         # Check if list is empty or index is invalid
14         if self.head is None or index < 0 or index >= self.size:
15             return None
16
17         # Start traversing from head
18         cur = self.head
19         while index > 0:
20             cur = cur.next
21             index -= 1
22         return cur
```

Search the node at index i

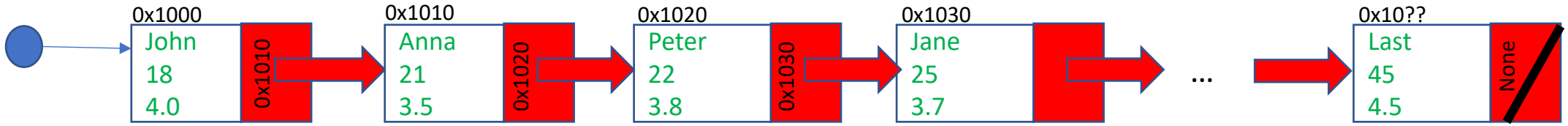
```
23 if __name__ == "__main__":
24
25     linked_list = LinkedList()
26
27     node1 = Node(10)
28     node2 = Node(20)
29     node3 = Node(30)
30
31     linked_list.head = node1
32     node1.next = node2
33     node2.next = node3
34     linked_list.size = 3  # Update size after adding 3 nodes
35
36     # Find node at index 2
37     found_node = linked_list.findNode(2)
38     if found_node:
39         print(f"Node at index 2: {found_node.data}")
40     else:
42         print("Index not found")
```



Insert node at given index

```
def insert(self, data, index):
    # Check if index is valid
    if index < 0 or index > self.size:
        print("Index out of range")
        return False

    new_node = Node(data)

    # Insert at beginning
    if index == 0:
        new_node.next = self.head
        self.head = new_node
        self.size += 1
        return True

    # Use findNode to get previous node
    prev_node = self.findNode(index - 1)
    if prev_node is not None:
        new_node.next = prev_node.next
        prev_node.next = new_node
        self.size += 1
        return True
    return False
```

SizeList




```
1   class ListNode:
2       def __init__(self, item):
3           self.item = item
4           self.next = None
5
6
7   class LinkedList:
8       def __init__(self):
9           self.head = None
10          self.size = 0
11
12
13  def sizeList(ll): # ll = LinkedList() #
14    return ll.size
```

SizeList

```
1   class Node:
2       def __init__(self, data):
3           self.data = data
4           self.next = None
5
6   class LinkedList:
7       def __init__(self):
8           self.head = None
9           self.size = 0
10
11      def findNode(self, index):
12          # Check if list is empty or index is invalid
13          if self.head is None or index < 0 or index >= self.size:
14              return None
15
16          # Start traversing from head
17          cur = self.head
18          while index > 0:
19              cur = cur.next
20              index -= 1
21          return cur
```

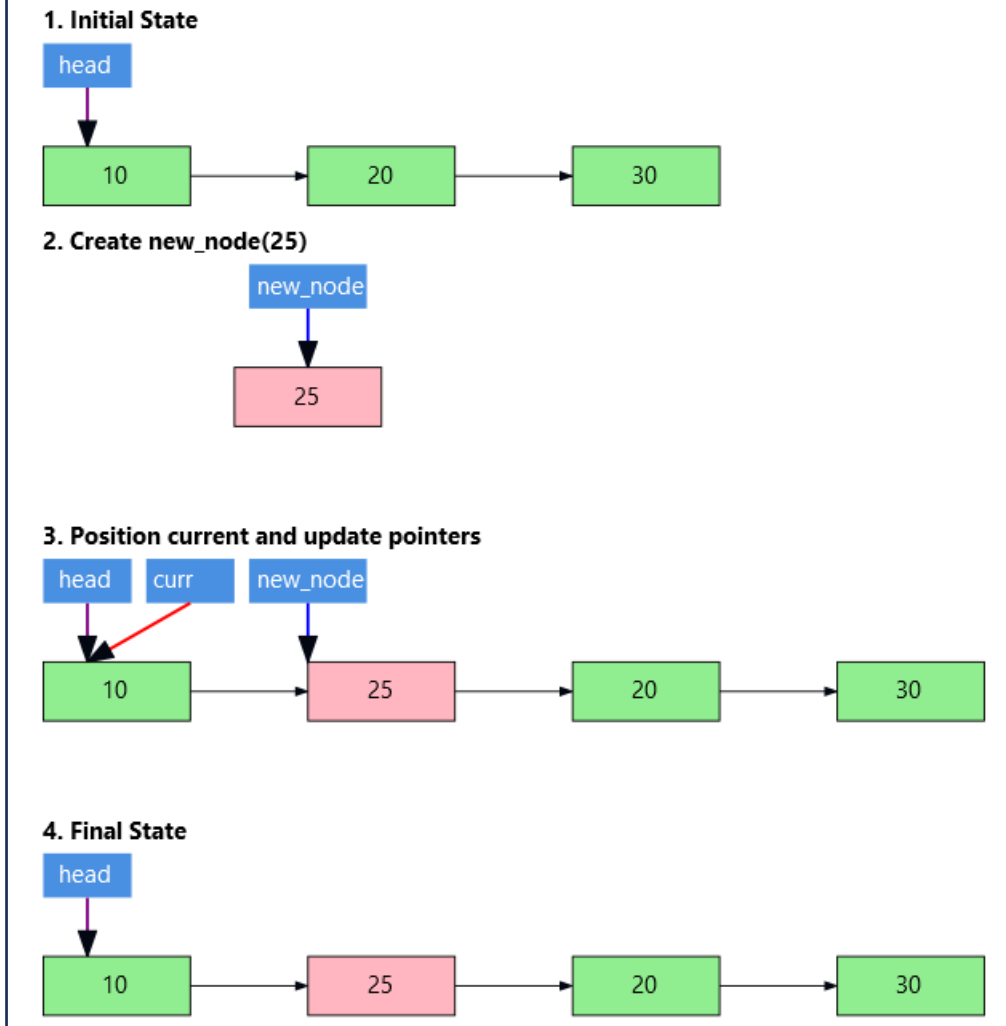
SizeList

```
22    def insert(self, data, index):
23        # Check if index is valid
24        if index < 0 or index > self.size:
25            print("Index out of range")
26            return False
27
28        new_node = Node(data)
29
30        # Insert at beginning
31        if index == 0:
32            new_node.next = self.head
33            self.head = new_node
34            self.size += 1
35            return True
36
37        # Use findNode to get previous node
38        prev_node = self.findNode(index - 1)
39        if prev_node is not None:
40            new_node.next = prev_node.next
41            prev_node.next = new_node
42            self.size += 1
44            return True
45        return False
```

```
46    def display(self):
47        current = self.head
48        while current:
49            print(current.data, end=" -> ")
50            current = current.next
51        print("None")
```

SizeList

```
46 if __name__ == "__main__":
47     # Create and initialize the linked list
48     linked_list = LinkedList()
49     linked_list.insert(10, 0)
50     linked_list.insert(20, 1)
51     linked_list.insert(30, 2)
52
53     # Display the original list
54     print("Current list:")
55     linked_list.display()
56
57     # Display initial size
58     print(f"Size of the list: {linked_list.size}")
59
60     # Insert 25 at index 1
61     linked_list.insert(25, 1)
62     print("\nAfter inserting 25 at index 1:")
63     linked_list.display()
64
65     # Display updated size
66     print(f"Size of list: {linked_list.size}")
```

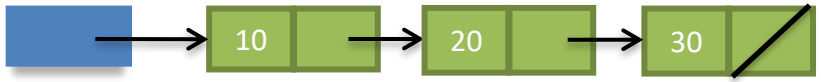


Doubly Linked List

Doubly Linked List

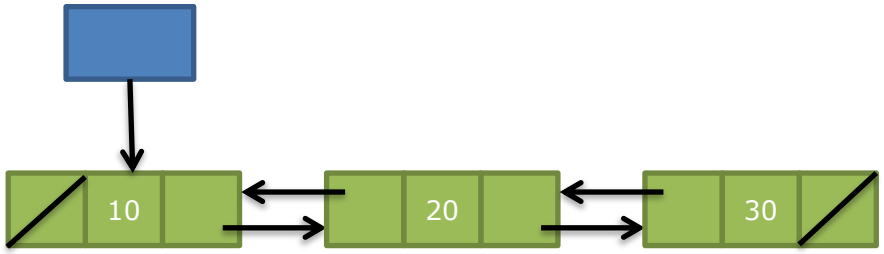
Singly Linked list: Only one link. Traversal of the list is one way only.

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



Doubly Linked List: two links in each node. It can search forward and backward.

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

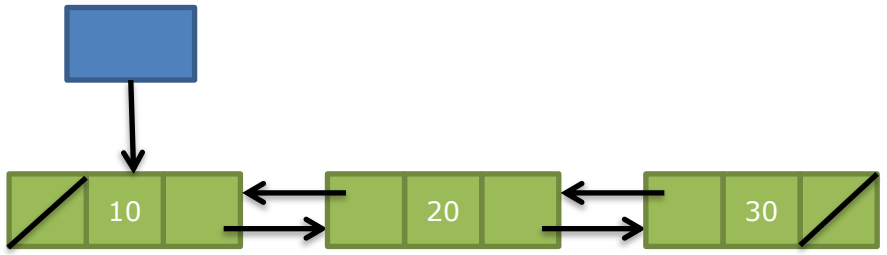


Doubly Linked List: Print

Print is similar to the Singly Linked List

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5         self.pre = None
```

```
1 def print_list(head):
2     current = head
3     while current:
4         print(current.data, end=" -> ")
5         current = current.next
6     print("None")
```

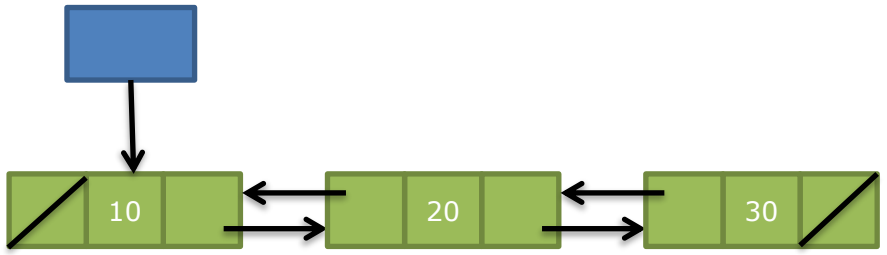


Doubly Linked List: Search

Display is similar to the Singly Linked List's

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5         self.pre = None
```

```
1 def search(self, data):
2         current = self.head
3         while current:
4             if current.data == data:
5                 return True
6             current = current.next
7         return False
```

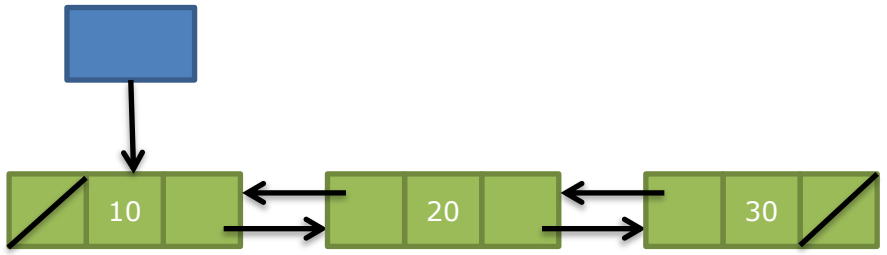


Doubly Linked List: Size (Count)

Size is similar to the Singly Linked List

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5         self.pre = None
```

```
1 def size(head):
2     count = 0
3     current = head
4     while current:
5         count += 1
6         current = current.next
7     return count
```



Doubly Linked List: Insertion

Insertion function: the solution is not unique

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

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

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre = new_node
9   new_node.pre.next = new_node
10  self.size += 1
```

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre.next = new_node
9   current.pre = new_node
10  self.size += 1
```

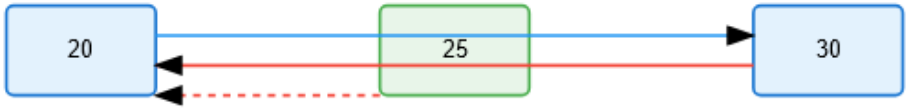
- If index = 2, the loop runs twice (i = 0 and i = 1).
- After the loop, current now points to the node at index 2.

Doubly Linked List: Insertion

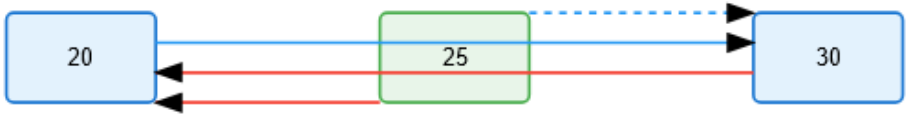
Insertion function: the solution is not unique

Step 1: Initial state

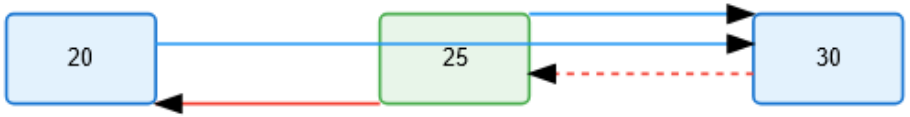
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current



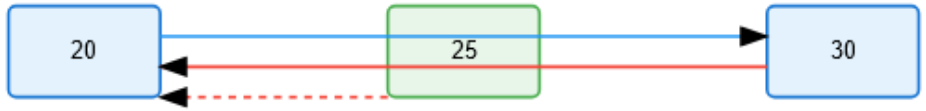
Step 4: current.pre = new_node



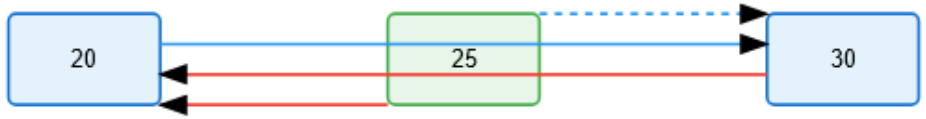
Step 5: new_node.pre.next = new_node

Step 1: Initial state

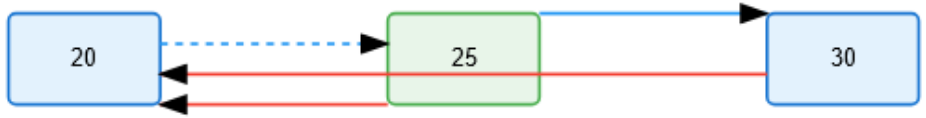
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

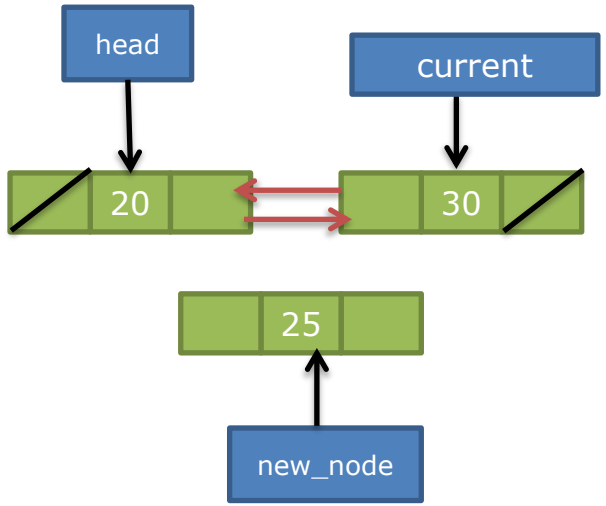


Step 4: current.pre.next = new_node



Step 5: current.pre = new_node

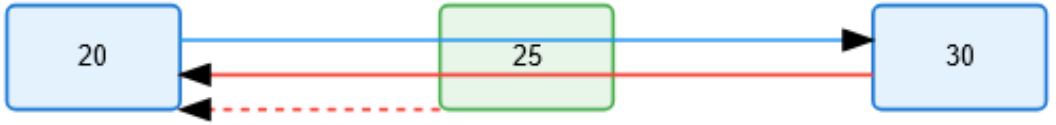
Doubly Linked List: Insertion



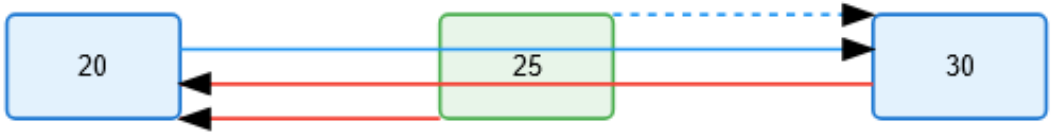

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre = new_node
9   new_node.pre.next = new_node
10  self.size += 1
```

Step 1: Initial state

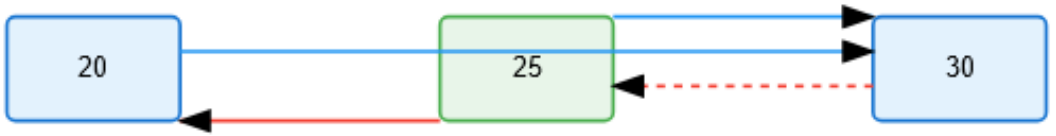
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

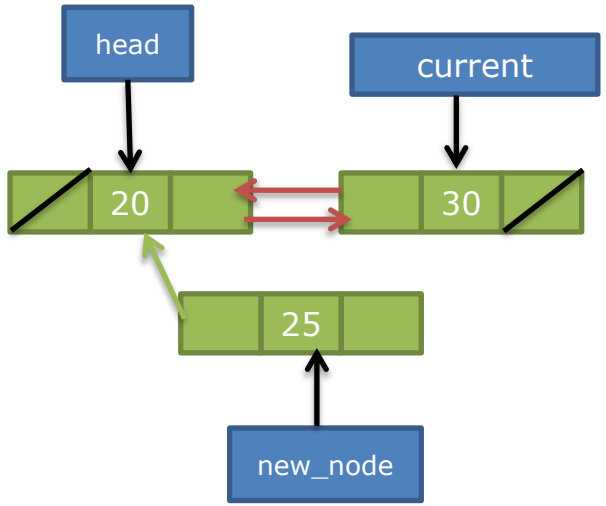


Step 4: current.pre = new_node



Step 5: new_node.pre.next = new_node

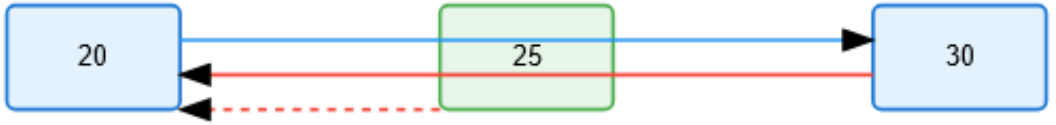
Doubly Linked List: Insertion



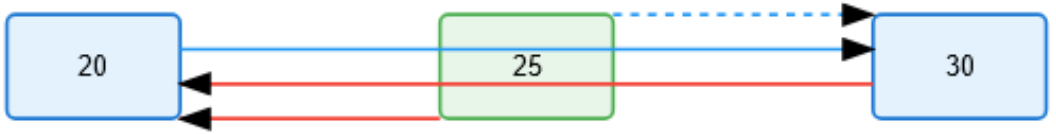

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre = new_node
9   new_node.pre.next = new_node
10  self.size += 1
```

Step 1: Initial state

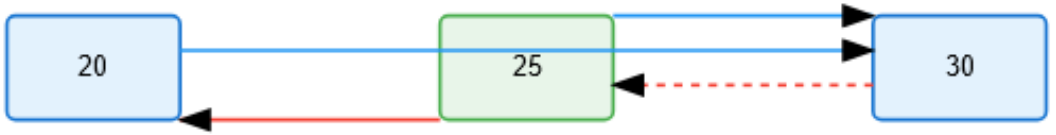
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

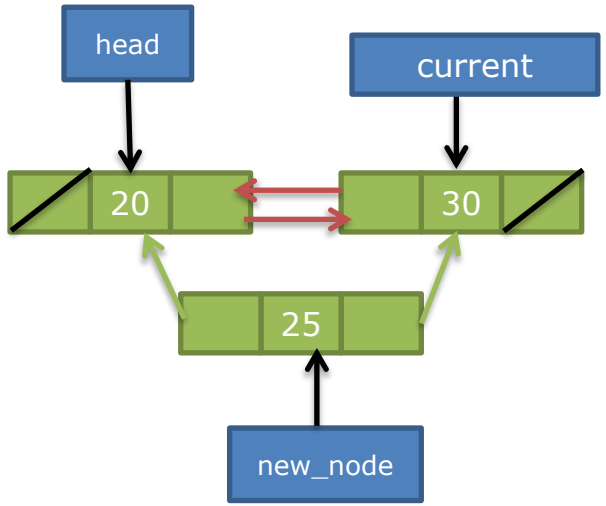


Step 4: current.pre = new_node



Step 5: new_node.pre.next = new_node

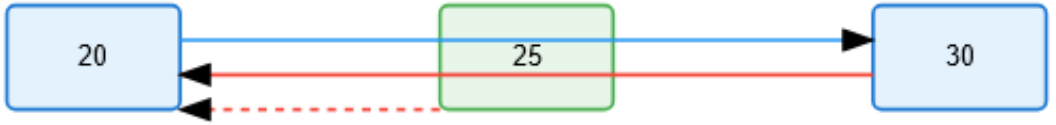
Doubly Linked List: Insertion



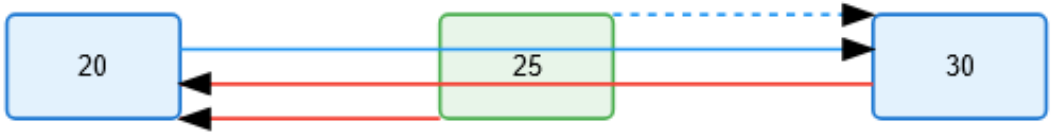

```
1   new_node = Node(data)
2   current = self.head
3   for i  in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre = new_node
9   new_node.pre.next = new_node
10  self.size += 1
```

Step 1: Initial state

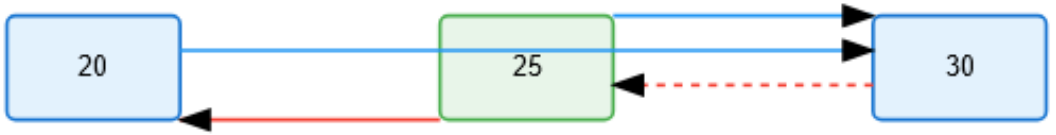
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

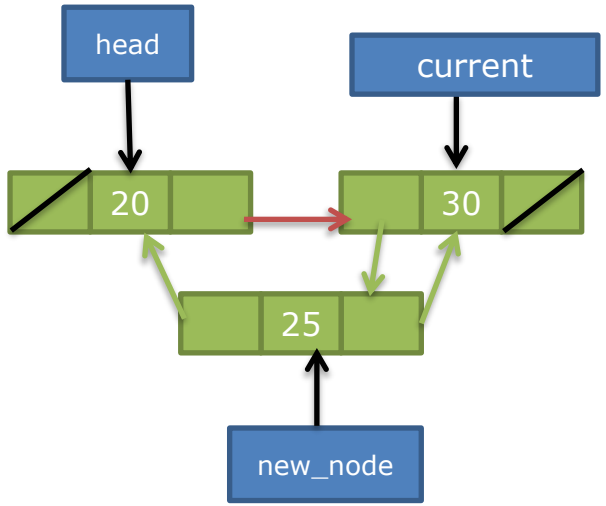


Step 4: current.pre = new_node



Step 5: new_node.pre.next = new_node

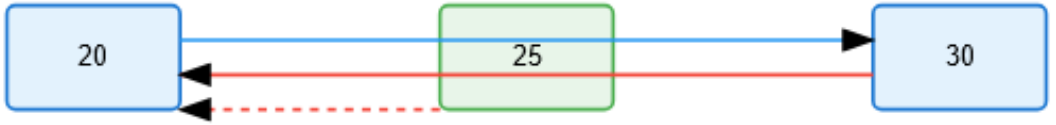
Doubly Linked List: Insertion



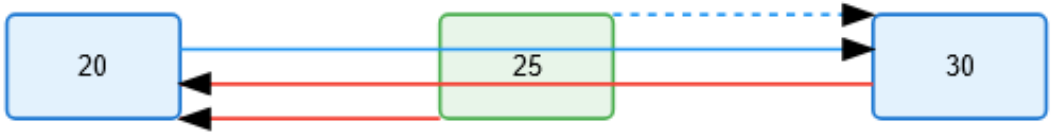

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre = new_node
9   new_node.pre.next = new_node
10  self.size += 1
```

Step 1: Initial state

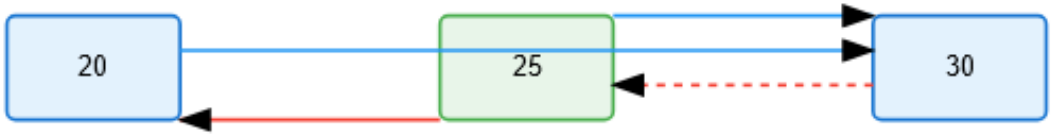
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

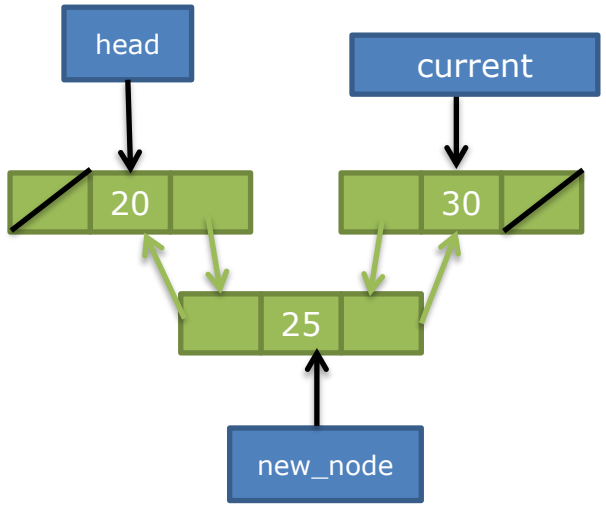


Step 4: current.pre = new_node



Step 5: new_node.pre.next = new_node

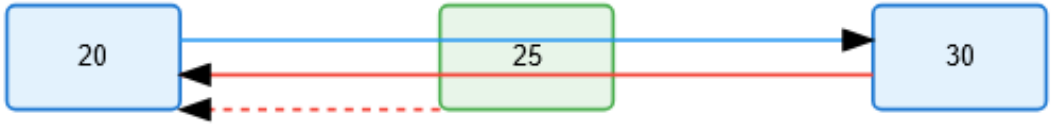
Doubly Linked List: Insertion



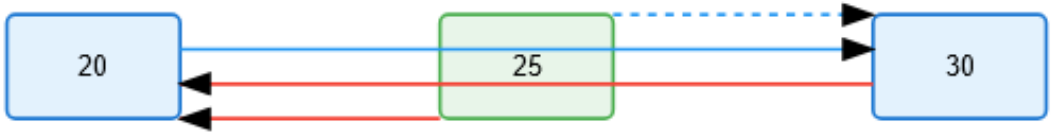

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre = new_node
9   new_node.pre.next = new_node
10  self.size += 1
```

Step 1: Initial state

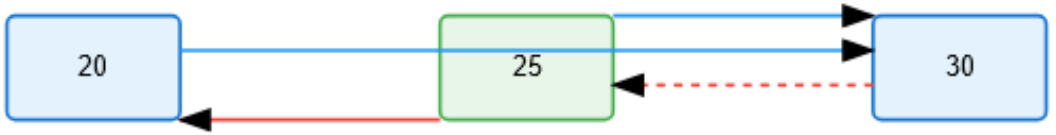
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current



Step 4: current.pre = new_node



Step 5: new_node.pre.next = new_node

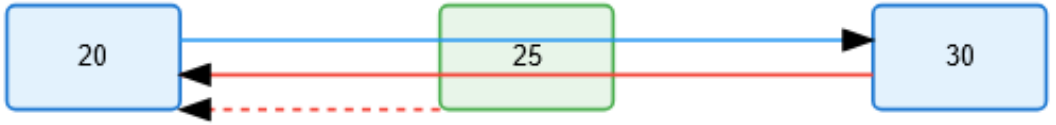
Doubly Linked List: Insertion

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

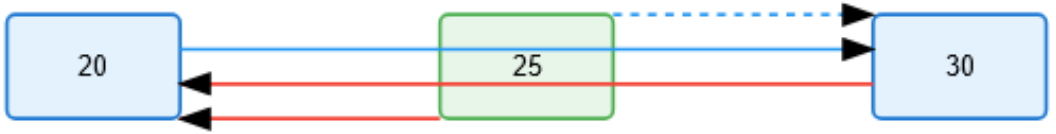
```
1  new_node = Node(data)
2  current = self.head
3  for i in range(index):
4      current = current.next
5
6  new_node.pre = current.pre
7  new_node.next = current
8  current.pre = new_node
9  new_node.pre.next = new_node
10 self.size += 1
```

Step 1: Initial state

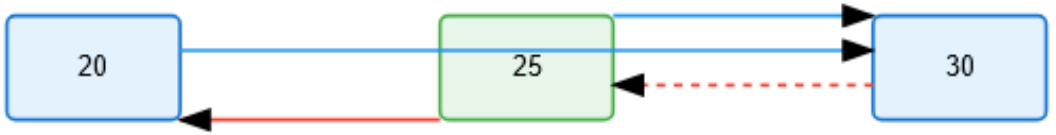
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

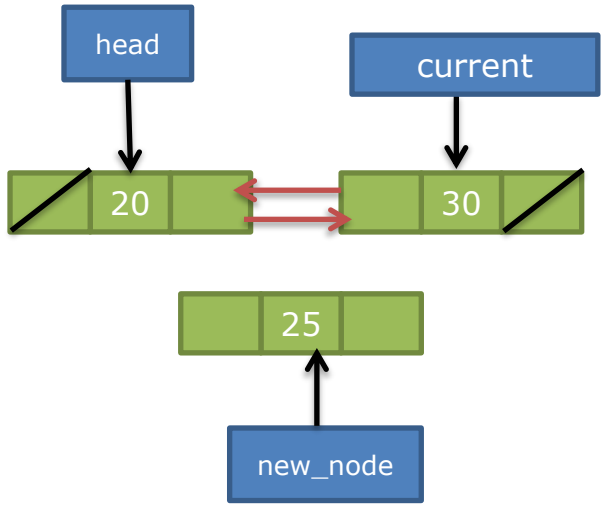


Step 4: current.pre = new_node



Step 5: new_node.pre.next = new_node

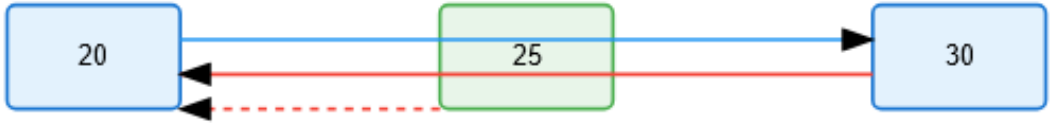
Doubly Linked List: Insertion



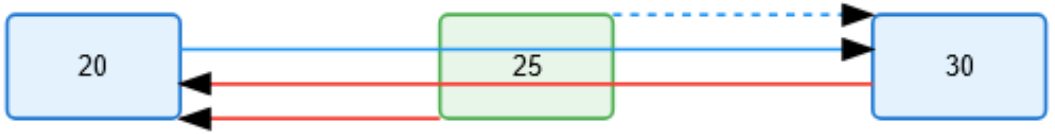

```
1   new_node = Node(data)
2   current = self.head
3   for i  in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre.next = new_node
9   current.pre = new_node
10  self.size += 1
```

Step 1: Initial state

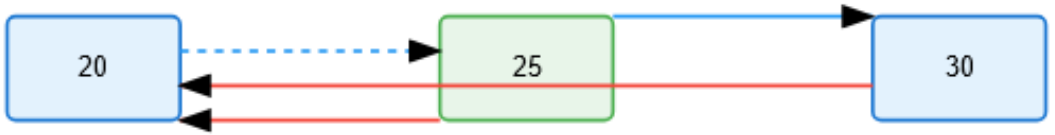
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

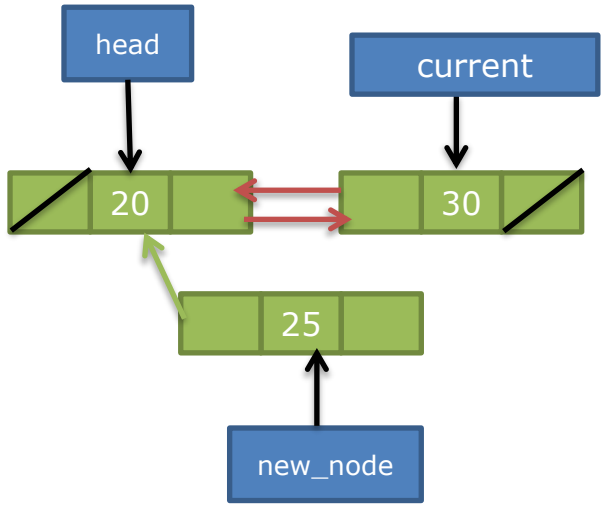


Step 4: current.pre.next = new_node



Step 5: current.pre = new_node

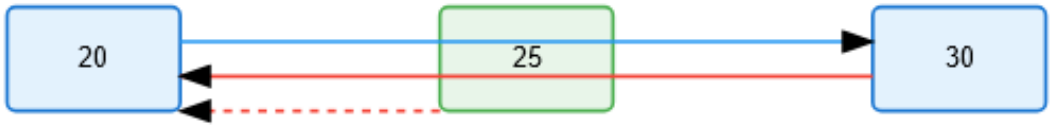
Doubly Linked List: Insertion



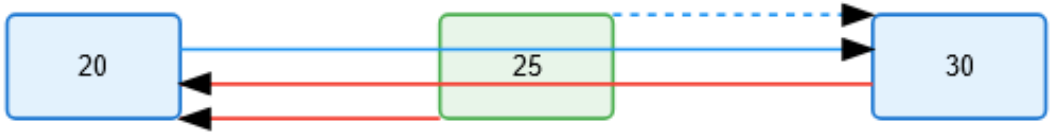

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre.next = new_node
9   current.pre = new_node
10  self.size += 1
```

Step 1: Initial state

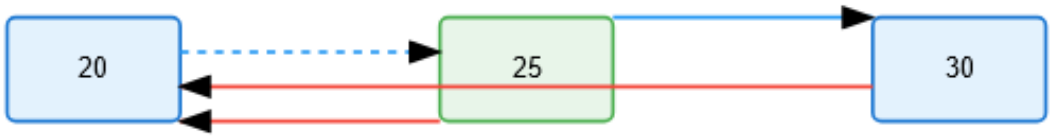
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

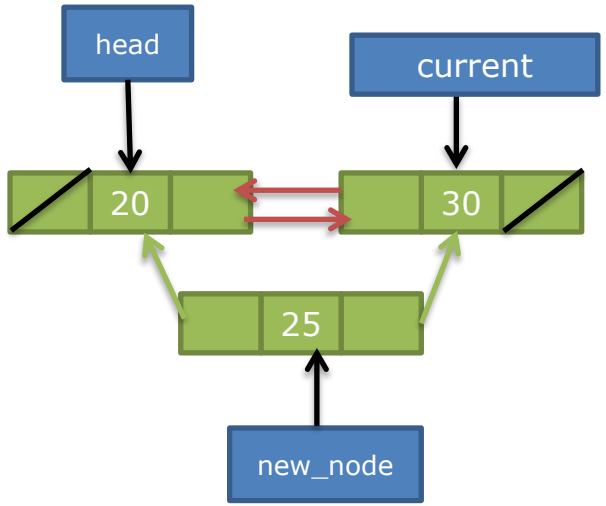


Step 4: current.pre.next = new_node



Step 5: current.pre = new_node

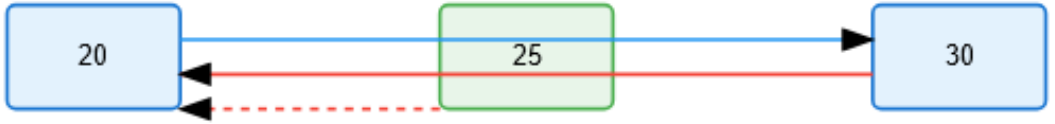
Doubly Linked List: Insertion



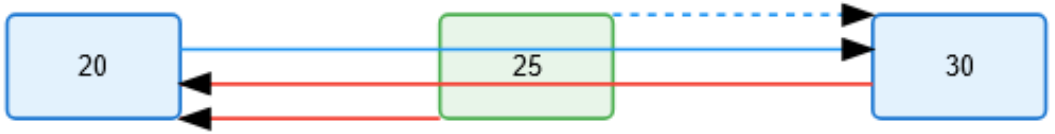

```
1   new_node = Node(data)
2   current = self.head
3   for i  in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre.next = new_node
9   current.pre = new_node
10  self.size += 1
```

Step 1: Initial state

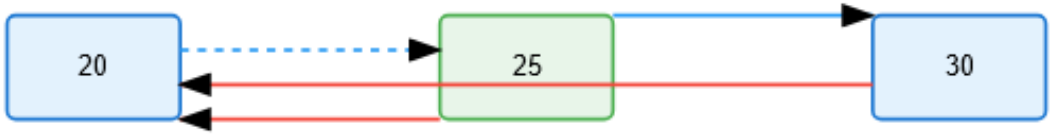
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

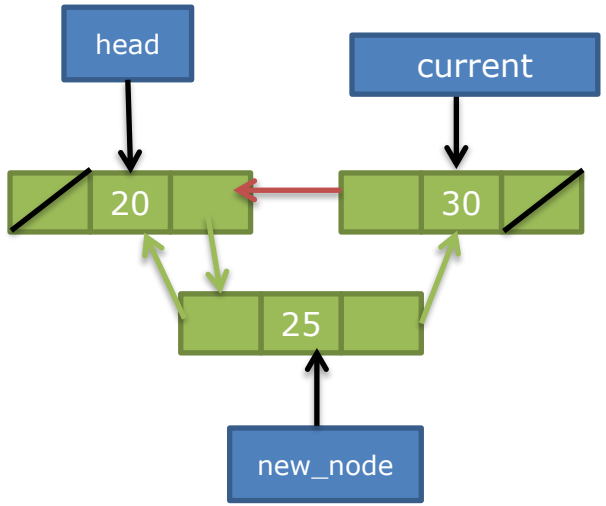


Step 4: current.pre.next = new_node



Step 5: current.pre = new_node

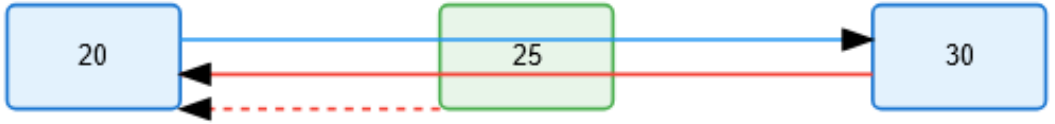
Doubly Linked List: Insertion



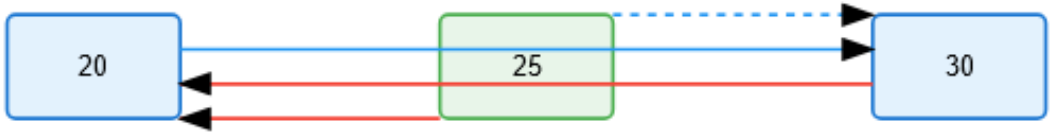

```
1   new_node = Node(data)
2   current = self.head
3   for i  in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre.next = new_node
9   current.pre = new_node
10  self.size += 1
```

Step 1: Initial state

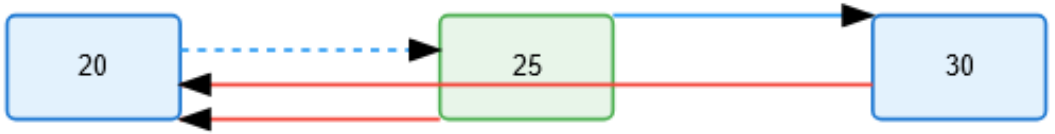
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current

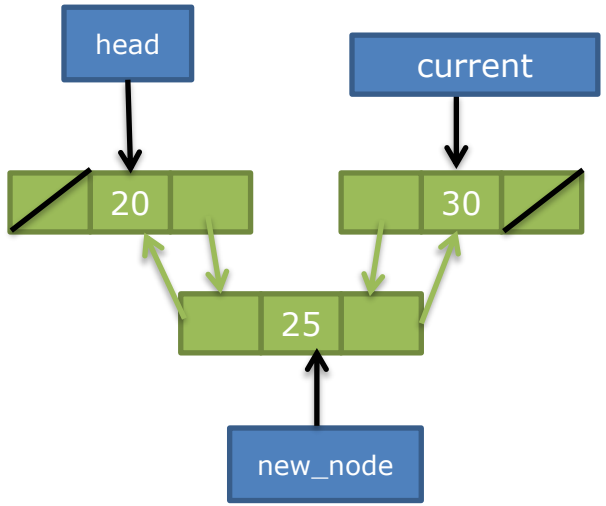


Step 4: current.pre.next = new_node



Step 5: current.pre = new_node

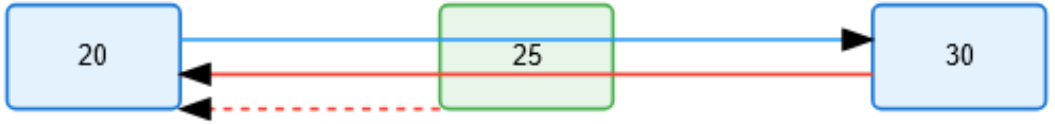
Doubly Linked List: Insertion



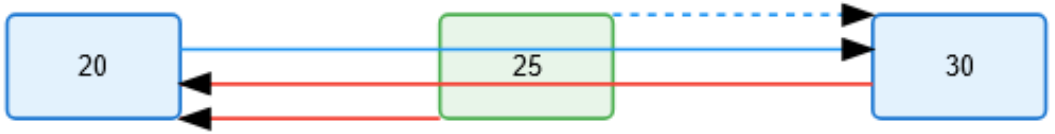

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre.next = new_node
9   current.pre = new_node
10  self.size += 1
```

Step 1: Initial state

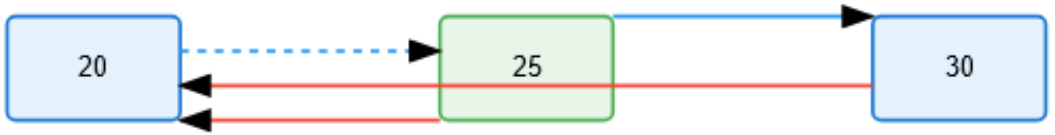
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current



Step 4: current.pre.next = new_node



Step 5: current.pre = new_node

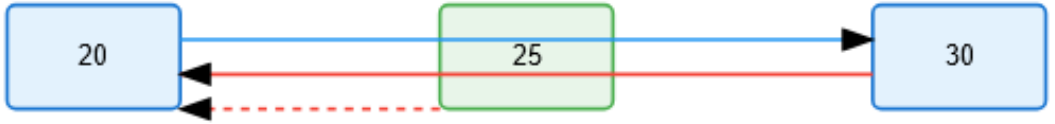
Doubly Linked List: Insertion

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

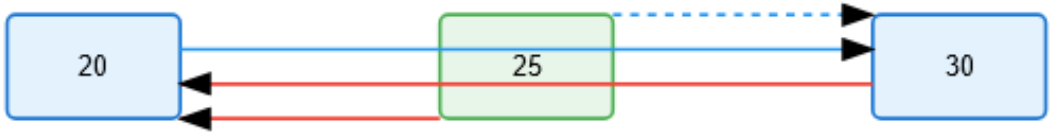
```
1  new_node = Node(data)
2  current = self.head
3  for i in range(index):
4      current = current.next
5
6  new_node.pre = current.pre
7  new_node.next = current
8  current.pre.next = new_node
9  current.pre = new_node
10 self.size += 1
```

Step 1: Initial state

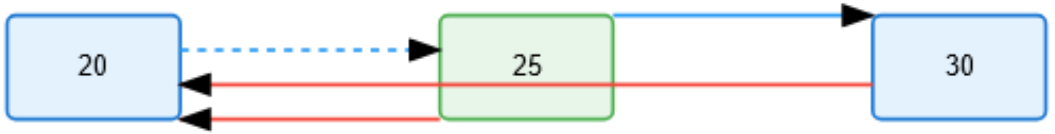
Step 2: new_node.pre = current.pre



Step 3: new_node.next = current



Step 4: current.pre.next = new_node

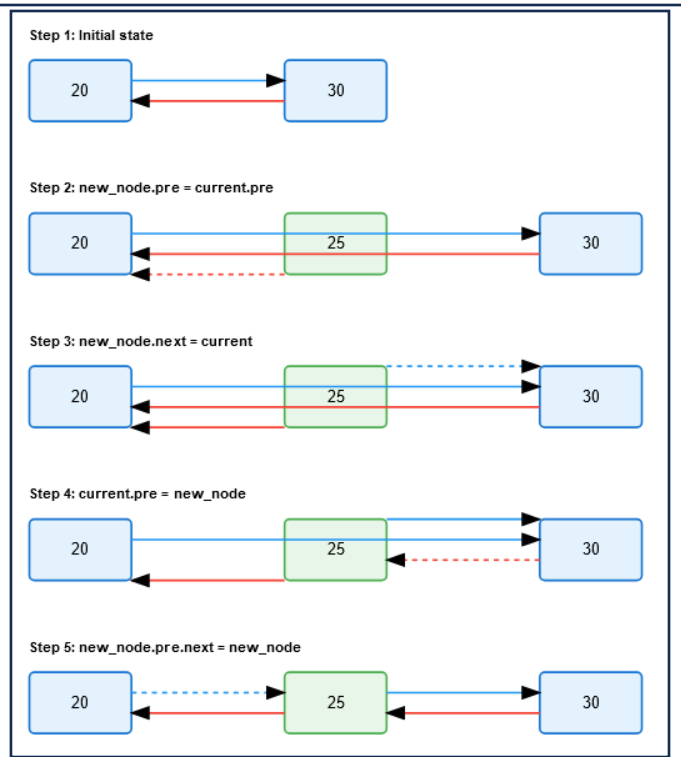


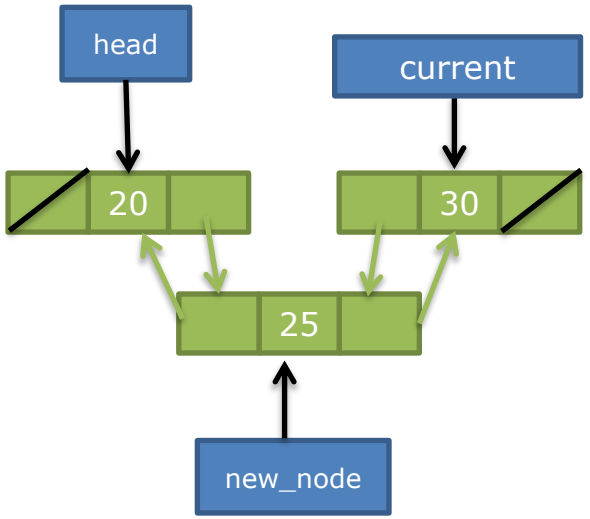
Step 5: current.pre = new_node

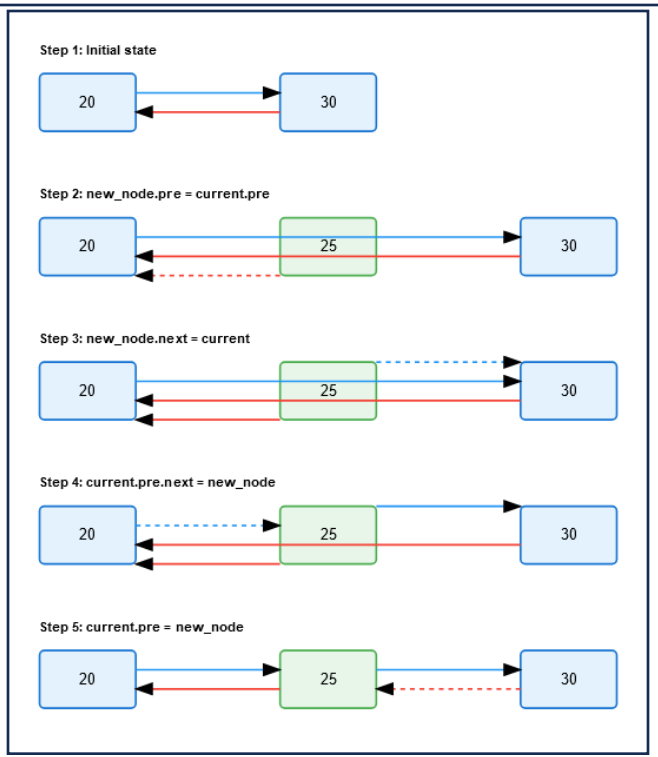
Doubly Linked List: Size (Count)

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre = new_node
9   new_node.pre.next = new_node
10  self.size += 1
```

```
1   new_node = Node(data)
2   current = self.head
3   for i in range(index):
4       current = current.next
5
6   new_node.pre = current.pre
7   new_node.next = current
8   current.pre.next = new_node
9   current.pre = new_node
10  self.size += 1
```

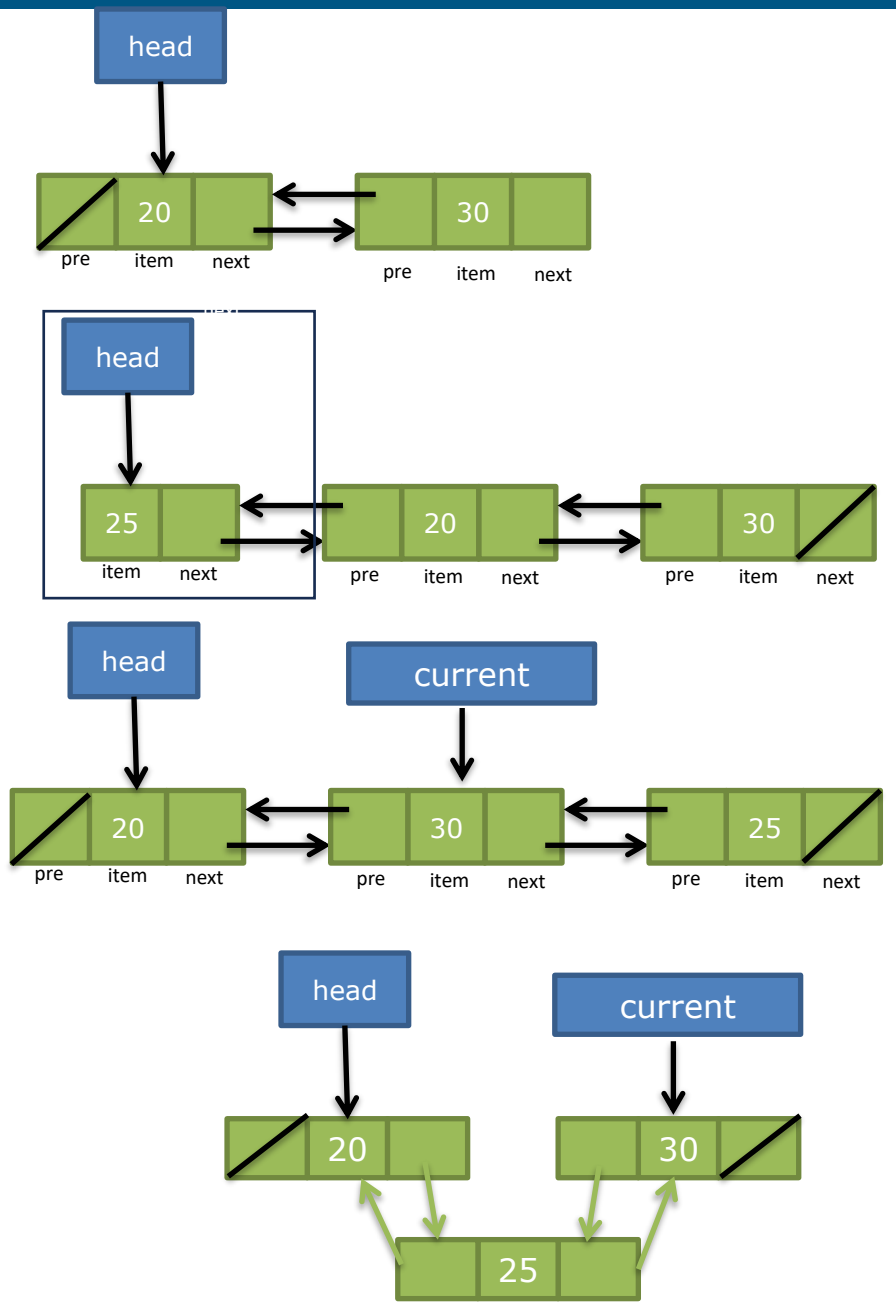






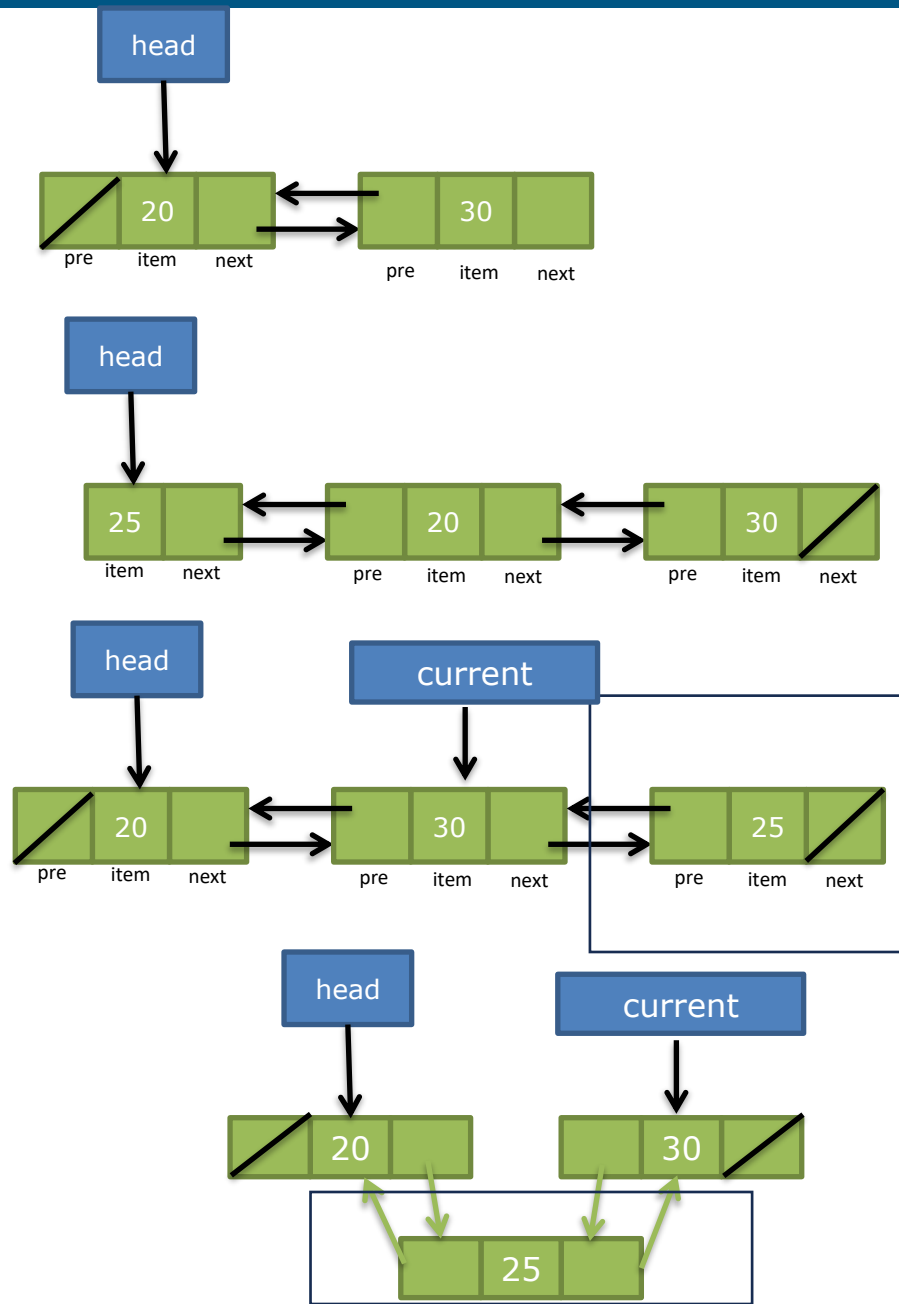
Doubly Linked List: Insertion

```
1   def insert_at(self, index, data):
2       # If index is invalid
3       if index < 0 or index > self.size:
4           raise ValueError("Invalid position")
5
6       # Create new node
7       new_node = Node(data)
8
9       # If inserting at beginning
10      if index == 0:
11          new_node.next = self.head
12          if self.head:
13              self.head.prev = new_node
14          self.head = new_node
15
16      # Inserting at middle or end
17      else:
18          current = self.head
19          # Traverse to position
20          for i in range(index-1):
21              current = current.next
22
23          # Link new node
24          new_node.prev = current
25          new_node.next = current.next
26          if current.next:
27              current.next.prev = new_node
28          current.next = new_node
28
30      self.size += 1
```



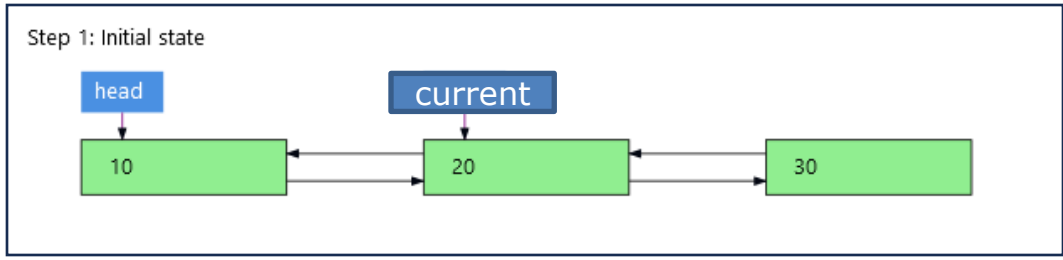
Doubly Linked List: Insertion

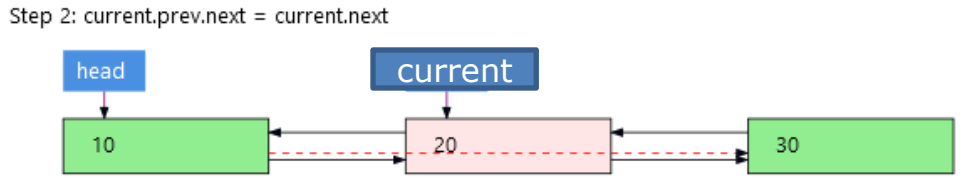
```
1   def insert_at(self, index, data):
2       # If index is invalid
3       if index < 0 or index > self.size:
4           raise ValueError("Invalid position")
5
6       # Create new node
7       new_node = Node(data)
8
9       # If inserting at beginning
10      if index == 0:
11          new_node.next = self.head
12          if self.head:
13              self.head.prev = new_node
14          self.head = new_node
15
16      # Inserting at middle or end
17      else:
18          current = self.head
19          # Traverse to position
20          for i in range(index-1):
21              current = current.next
22
23          # Link new node
24          new_node.prev = current
25          new_node.next = current.next
26          if current.next:
27              current.next.prev = new_node
28          current.next = new_node
28
30      self.size += 1
```

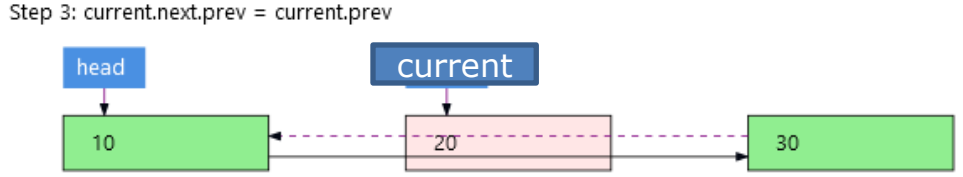


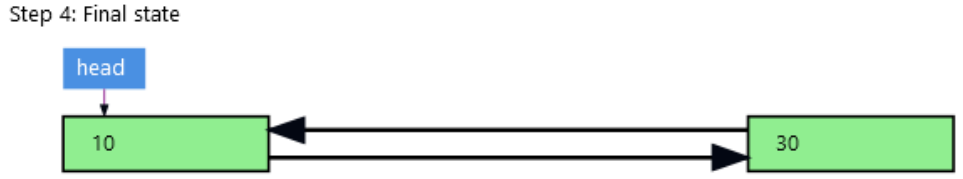
Doubly Linked List: Deletion (Data)

```
1   def delete(self, data):
2           if self.head is None:
3               raise ValueError("List is empty")
4
5           current = self.head
6           while current:
7               if current.data == data:
8                   if current.prev:
9                       current.prev.next = current.next
10                  else:
11                      self.head = current.next
12
13                  if current.next:
14                      current.next.prev = current.prev
15
16                  self.size -= 1
17                  return True
18              current = current.next
19
20          return False
```





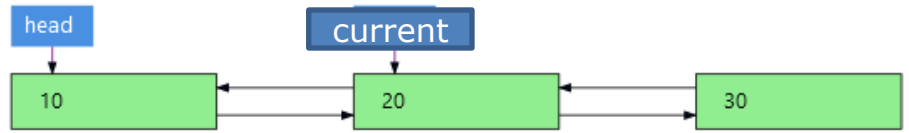




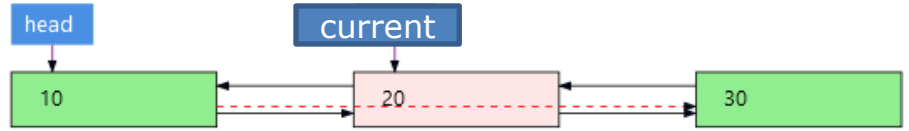
Doubly Linked List: Deletion (Data)

```
1   def delete(self, data):
2           if self.head is None:
3               raise ValueError("List is empty")
4
5           current = self.head
6           while current:
7               if current.data == data:
8                   if current.prev:
9                       current.prev.next = current.next
10                  else:
11                      self.head = current.next
12
13                  if current.next:
14                      current.next.prev = current.prev
15
16                  self.size -= 1
17                  return True
18              current = current.next
19
20          return False
```

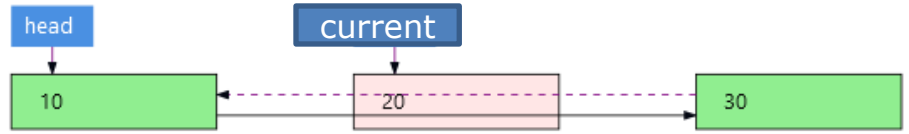
Step 1: Initial state



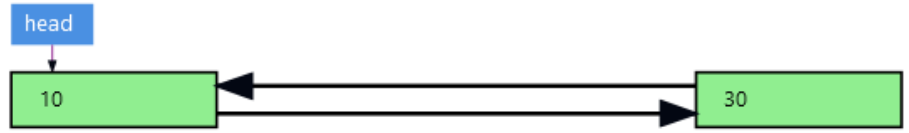
Step 2: current.prev.next = current.next



Step 3: current.next.prev = current.prev

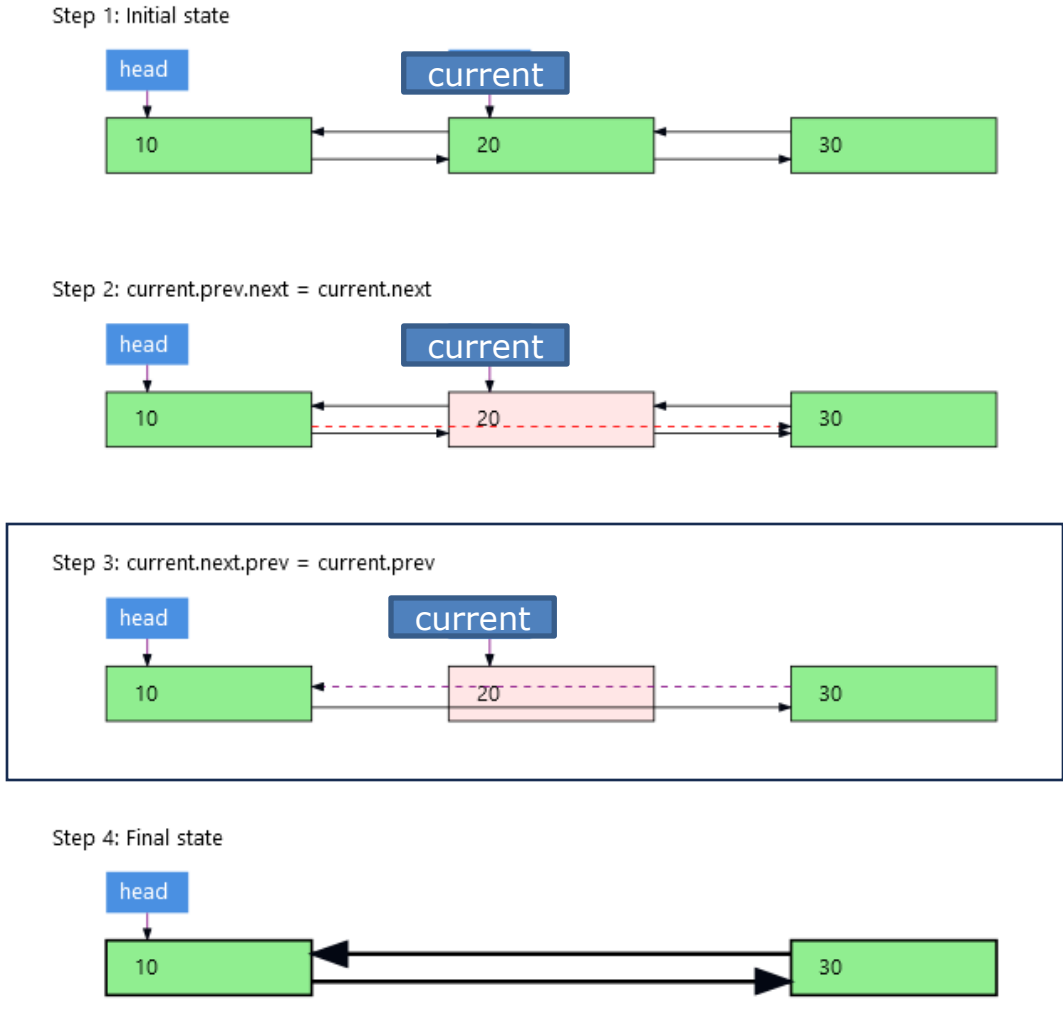


Step 4: Final state



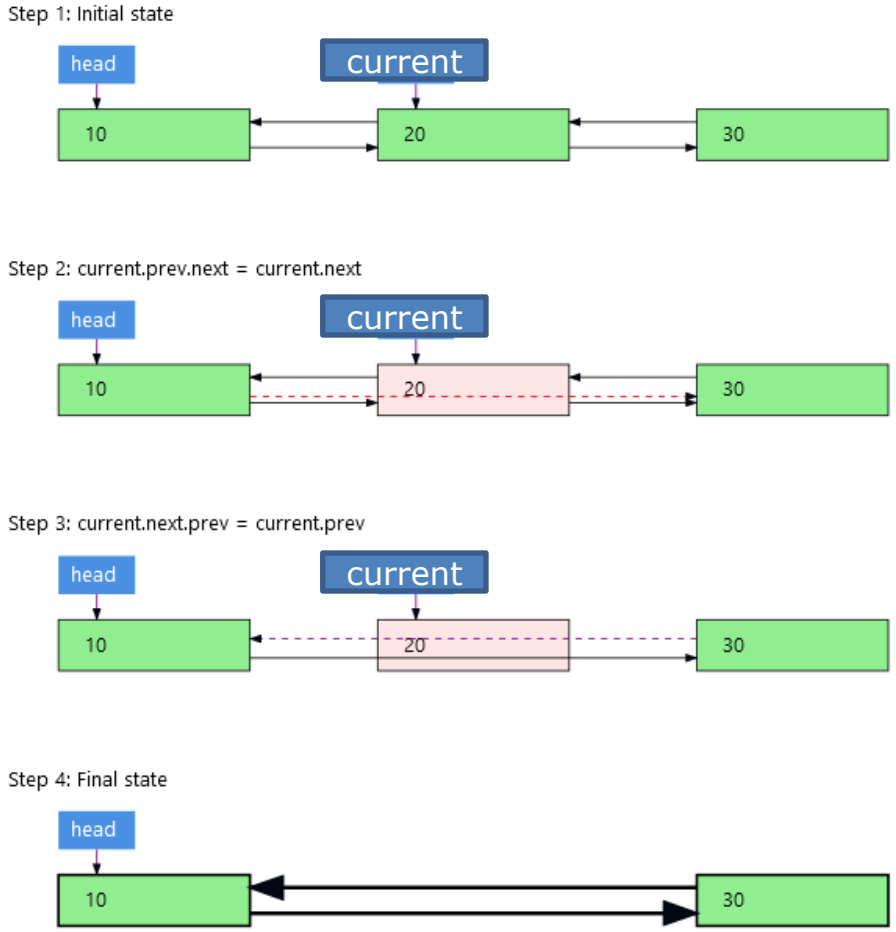
Doubly Linked List: Deletion (Data)

```
1   def delete(self, data):
2           if self.head is None:
3               raise ValueError("List is empty")
4
5           current = self.head
6           while current:
7               if current.data == data:
8                   if current.prev:
9                       current.prev.next = current.next
10                  else:
11                      self.head = current.next
12
13                  if current.next:
14                      current.next.prev = current.prev
15
16                  self.size -= 1
17                  return True
18              current = current.next
19
20          return False
```



Doubly Linked List: Deletion (Data)

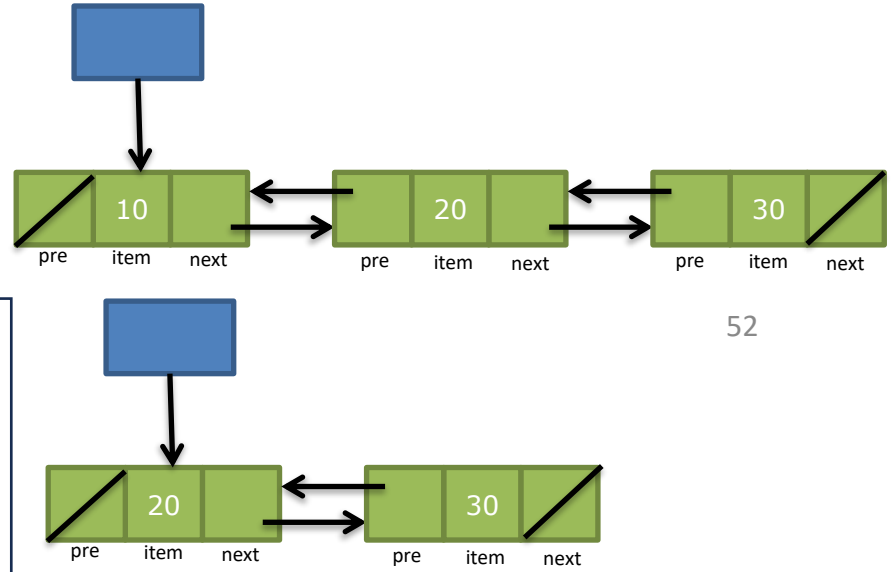
```
1    def delete(self, data):
2            if self.head is None:
3                raise ValueError("List is empty")
4
5            current = self.head
6            while current:
7                if current.data == data:
8                    if current.prev:
9                        current.prev.next = current.next
10                   else:
11                       self.head = current.next
12
13                   if current.next:
14                       current.next.prev = current.prev
15
16                   self.size -= 1
17                   return True
18               current = current.next
19
20           return False
```



- When `delete(self, data)` finishes, current goes out of scope. This means the local reference to the node is removed.
- If no other references to the node exist, its reference count drops to zero, making it eligible for garbage collection.
- Python's garbage collector will automatically reclaim the memory in a future cycle.

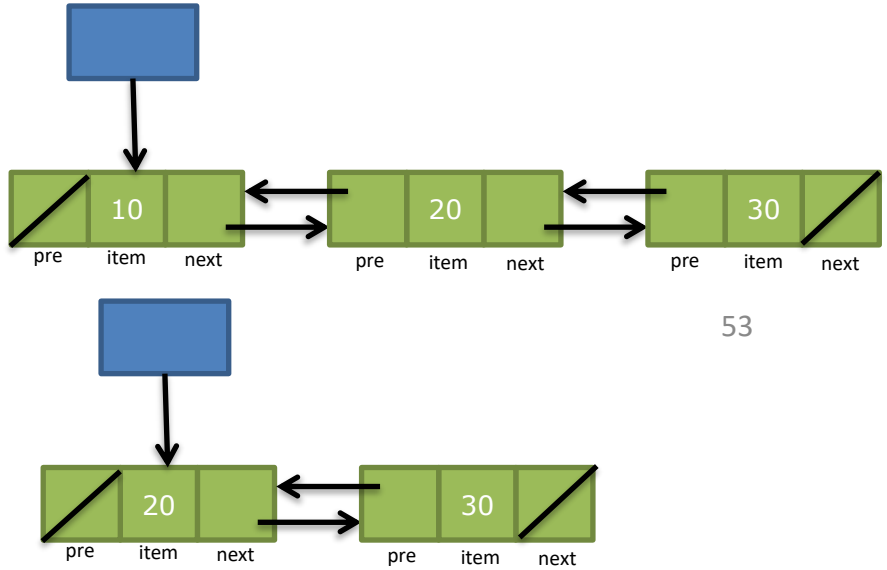
Doubly Linked List: Deletion (Index)

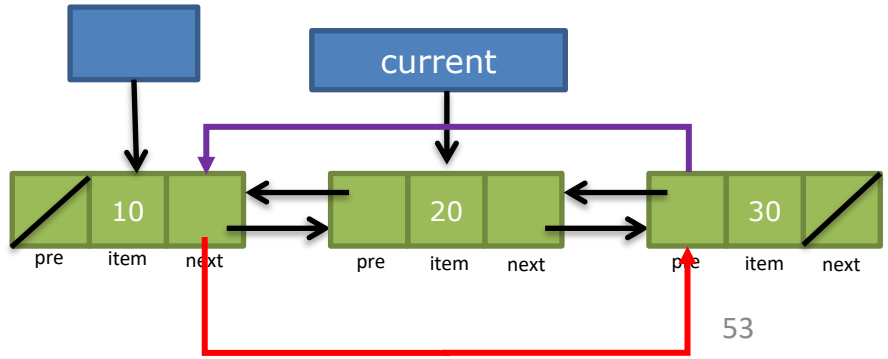
```
1 def remove_at(self, index):
2       # Case 1: Check if the list is empty
3       if self.head is None:
4           print("List is empty")
5           return False
6       # Case 2: Validate index
7       if index < 0 or index >= self.size:
8           print("Invalid index")
9           return False
10      # Case 3: Remove the first node (index 0)
11      if index == 0:
12          self.head = self.head.next
13          if self.head:  # If the list is not empty after removal
14              self.head.prev = None
15          self.size -= 1
16          return True
17      # Case 4: Remove from the middle or end
18      current = self.head
19       for i in range(index): # Traverse to the node at the given index
20          current = current.next
21      # Update pointers to remove the node
22      current.prev.next = current.next
23      if current.next:  # If it's not the last node
24          current.next.prev = current.prev
25      self.size -= 1
26      return True
```



Doubly Linked List: Deletion (Index)

```
1 def remove_at(self, index):
2       # Case 1: Check if the list is empty
3       if self.head is None:
4           print("List is empty")
5           return False
6       # Case 2: Validate index
7       if index < 0 or index >= self.size:
8           print("Invalid index")
9           return False
10      # Case 3: Remove the first node (index 0)
11      if index == 0:
12          self.head = self.head.next
13          if self.head:  # If the list is not empty after removal
14              self.head.prev = None
15          self.size -= 1
16          return True
17      # Case 4: Remove from the middle or end
18      current = self.head
19       for i in range(index): # Traverse to the node at the given index
20          current = current.next
21      # Update pointers to remove the node
22      current.prev.next = current.next
23      if current.next:  # If it's not the last node
24          current.next.prev = current.prev
25      self.size -= 1
26      return True
```

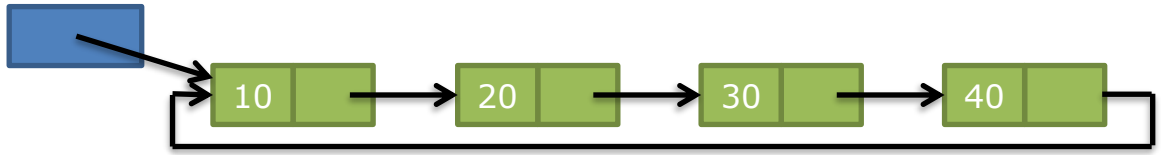




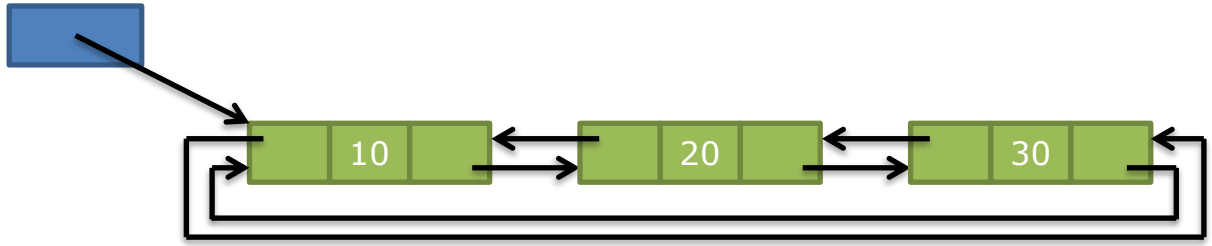
Circular Linked List

Doubly Linked List

- Circular singly linked lists
  - Last node has next pointer pointing to first node

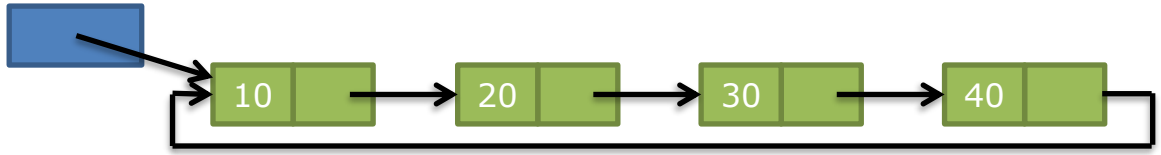


- Circular doubly linked lists
  - Last node has next pointer pointing to first node
  - First node has pre pointer pointing to last node

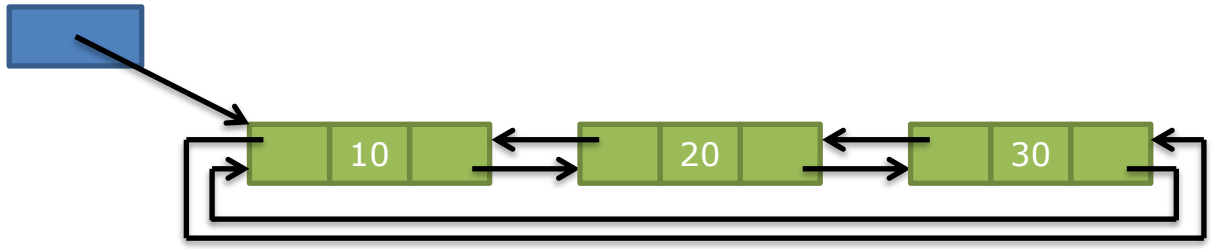


Doubly Linked List

- Circular singly linked lists
  - Last node has next pointer pointing to first node



- Circular doubly linked lists
  - Last node has next pointer pointing to first node
  - First node has pre pointer pointing to last node



- **Display, Search, Size**: the last node's link is equal to head instead of **None**. The stop criteria needs to change.
- **Insert** and **Delete**: there is no special case at first or last position. The head node may need to update if the first node is affected.

Advantages and Disadvantages: Doubly Linked Lists

- **Advantages:**
  - Can be traversed in **either direction** (may be essential for some programs)
  - Some operations, such as deletion and inserting before a link, become easier
- **Disadvantages:**
  - Requires **more space**
  - List manipulations are **slower** (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Practical Applications - Doubly Linked Lists

- **Doubly linked lists (DLLs)** are useful in applications where **bidirectional traversal**, **flexible insertion**, and **deletion** are essential. Here are some common real-world applications of DLLs:

- **Browser History Navigation**:
  - Browsers use doubly linked lists to manage page navigation history. Each page visit is a node, allowing users to move back and forth between pages seamlessly.

- **Undo and Redo Operations**:
  - In text editors and applications with undo/redo functionality, DLLs store each state change, allowing users to go forward or backward in the edit history efficiently.

- **Implementation of LRU Cache (Least Recently Used)**:
  - In memory management, doubly linked lists are used to track recently accessed items. Nodes store cache entries, allowing the system to efficiently add new entries and remove the least recently accessed ones, with both directions accessible.

Advantages and Disadvantages: Circular Linked Lists

- **Advantages:**
  - **Constant Traversal**: Traversal from any node will eventually return to the starting node, making circular lists ideal for applications needing repeated cycling.
- **Disadvantages:**
  - **Risk of Infinite Loops**: Errors in traversal logic can easily lead to infinite loops due to the circular structure, especially if stop conditions are missed.

Practical Applications - Circular Linked Lists

- **Round-Robin Scheduling**:
  - Operating systems use circular linked lists for round-robin scheduling to cycle through processes, moving to the next in a loop, which ensures each task gets its turn.

- **Circular Buffers**:
  - Circular linked lists are used in circular buffers, which are common in streaming, data buffering, and telecommunications, allowing efficient handling of incoming and outgoing data.