test2

May 3, 2025

[12]:
```python
# Now appending the linked list

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

class LinkedList:
    def __init__(self, value):
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1

    def append_list(self,value):
        new_node = Node(value)
        self.tail.next = new_node
        # self.tail = new_node
        self.length = self.length+1

    def print_list(self):
        temp = self.head
        while temp != None:
            print(temp.value)
            temp = temp.next


my_linked_list = LinkedList(13)

print(my_linked_list.head)
print(my_linked_list.tail)

print(my_linked_list.head.value)
print(my_linked_list.tail.value)
```

```
<__main__.Node object at 0x7ef8a846d0f0>
<__main__.Node object at 0x7ef8a846d0f0>
13
```

```
13
```

0.0.1 When you create a new instance of LinkedList:

The constructor of LinkedList is called. This creates a Node with a value of 13 and both head and tail refer to this node. The length of the list is initialized to 1 since there is only one node in the list at this point.

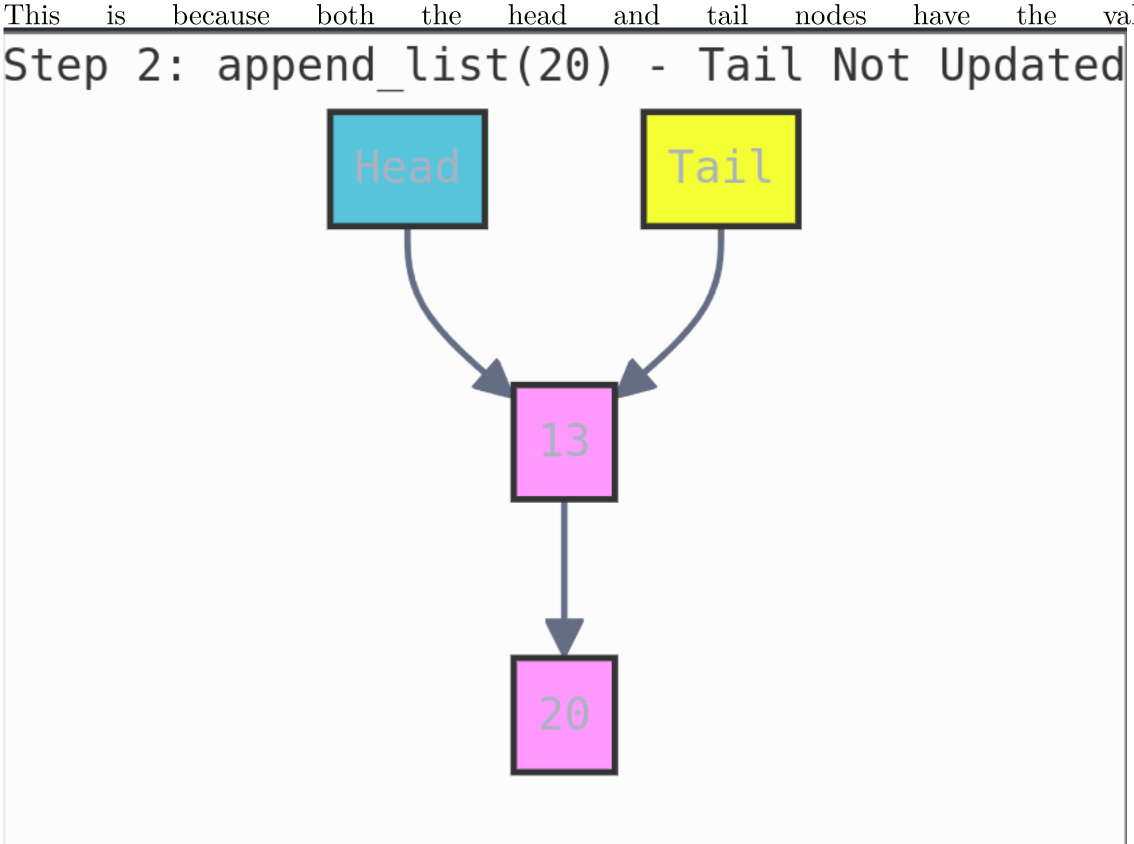
0.0.2 When you print the head and tail of the list:

These are memory addresses of the Node object where the head and tail nodes are stored.

Since both the head and tail point to the same node (the one with value 13), they have the same memory address.

0.0.3 Then, when you print the values of the head and tail:

This is because both the head and tail nodes have the value 13.



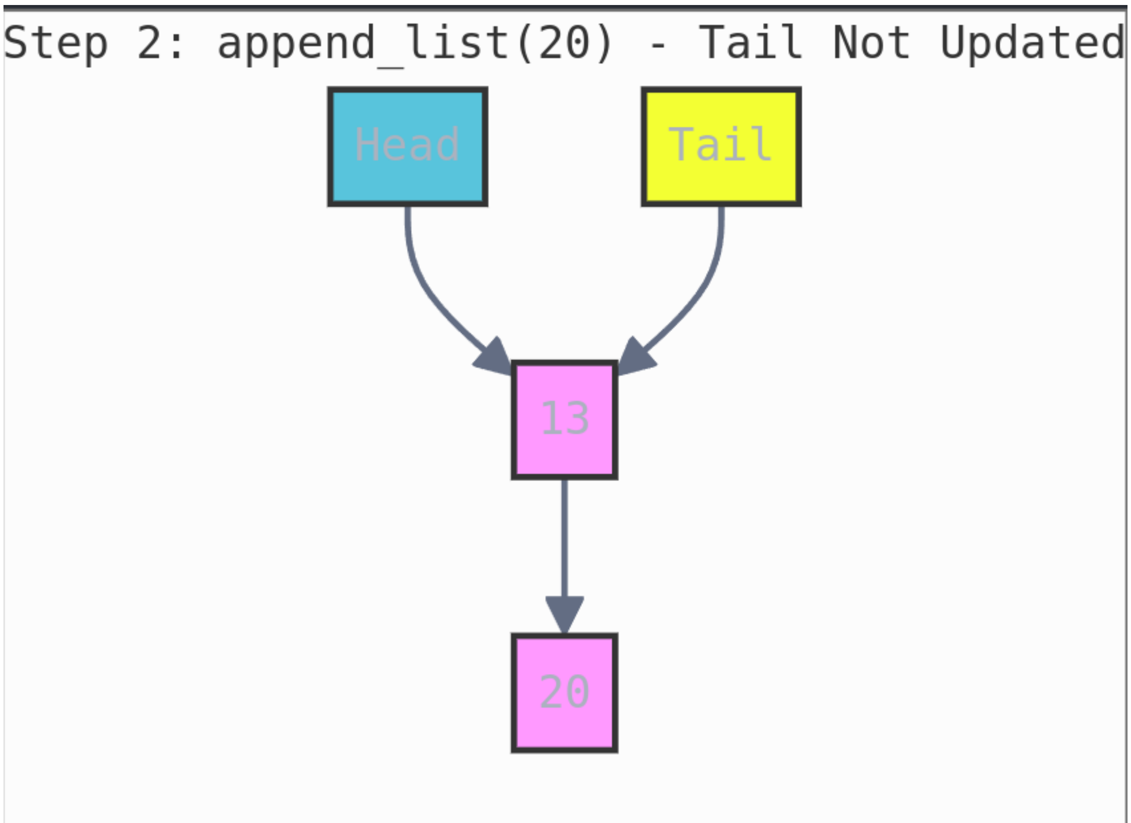
0.0.4 Appending Another Node

Now you append another value (20) to the linked list:

In the append_list method:

- A new Node is created with the value 20. –> new_node = Node(value)

- The next pointer of the current tail (which is the node with value 13) is set to point to the new node with value 20. —>self.tail.next = new_node
- However, this line is commented out; —> # self.tail = new_node However, you forgot to update the tail reference to point to this new node. The tail still points to the original node with value 13, but the new node is connected to it.



[15]:
```
my_linked_list.append_list(20)

print(my_linked_list.print_list())

print(my_linked_list.head)
print(my_linked_list.tail)

print(my_linked_list.head.value)
print(my_linked_list.tail.value)
```

```
13
20
None
<__main__.Node object at 0x7ef8a846d0f0>
<__main__.Node object at 0x7ef8a846d0f0>
13
13
```

- head still points to the node with value 13, so it will print something like:

So, this is okay value.

BUT

- tail still points to the original node with value 13, so it will also print the same memory address as head

[ ]:
```python
# Now appending the linked list

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

class LinkedList:
    def __init__(self, value):
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1

    def append_list(self,value):
        new_node = Node(value)
        self.tail.next = new_node
        self.tail = new_node
        self.length = self.length+1

    def print_list(self):
        temp = self.head
        while temp != None:
            print(temp.value)
            temp = temp.next


my_linked_list = LinkedList(13)
my_linked_list.append_list(20)

print(my_linked_list.head)
print(my_linked_list.tail)

print(my_linked_list.head.value)
print(my_linked_list.tail.value)
```

```
<__main__.Node object at 0x7ef8a8453100>
<__main__.Node object at 0x7ef8a8453850>
13
20
```

1 Now comparision of Case

Case 1: self.tail.next = new_node is executed, but self.tail is not updated (i.e., the line where self.tail = new_node is commented out).

Case 2: self.tail.next = new_node is commented out, but self.tail is updated (i.e., the line self.tail = new_node is executed without linking next).

Case 1:

[24]:
```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self, value):
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1

    def append_list(self, value):
        new_node = Node(value)
        self.tail.next = new_node  # Link current tail to new node
        # self.tail = new_node  # This line is commented out, so we don't update the tail
        self.length += 1

    def print_list(self):
        temp = self.head
        while temp != None:
            print(temp.value)
            temp = temp.next


# Case 1: self.tail.next is executed, but self.tail is not updated
my_linked_list = LinkedList(13)
my_linked_list.append_list(20)
my_linked_list.append_list(30)

# Print the list
my_linked_list.print_list()

# Check the head and tail nodes
print("Head:", my_linked_list.head.value)
print("Tail:", my_linked_list.tail.value)
```

```
13
```

```
30
Head: 13
Tail: 13
```

[28]:
```
print("Head value:", my_linked_list.head.value)

print("Next node value:", my_linked_list.head.next.value)
```

```
Head value: 13
Next node value: 30
```

In this scenario, you link the original tail to the new node (self.tail.next = new_node), but you don't update self.tail to point to the new node. This means:

The list structure is still correct — the nodes are linked together.

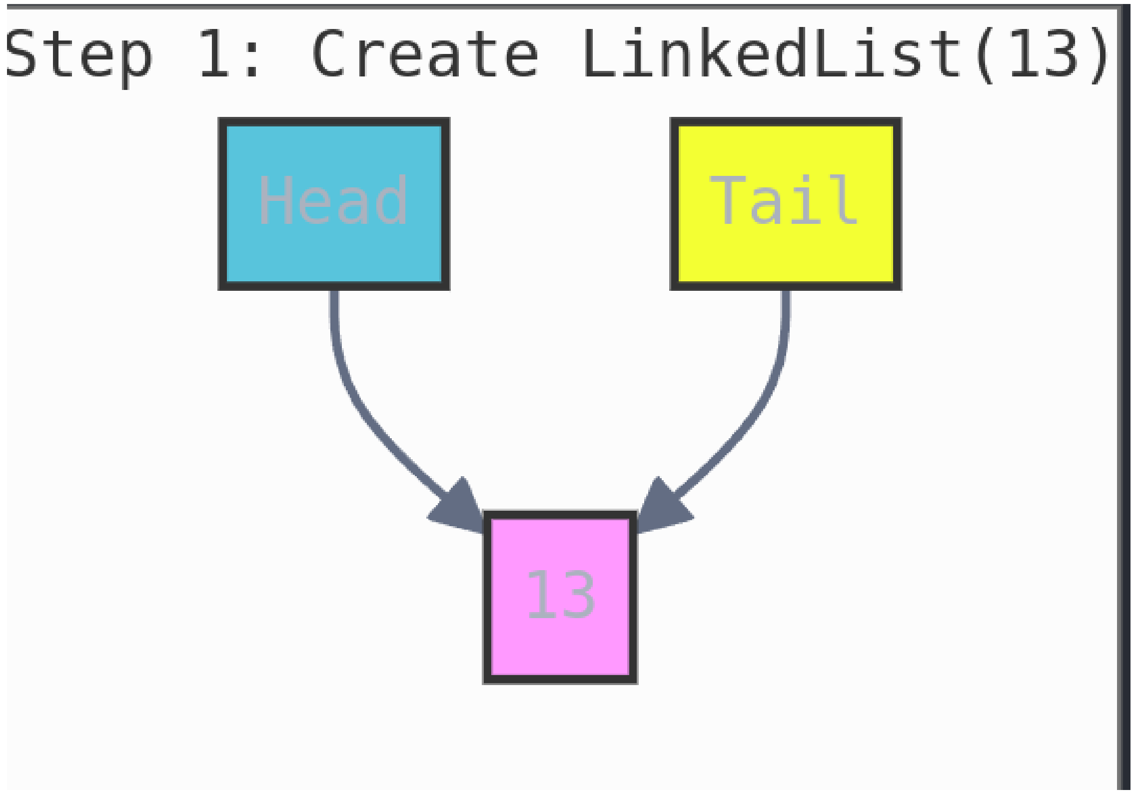
However, since self.tail still points to the original node (which was 13), the tail reference does not reflect the actual last node in the list (which would be 30 after the append).

But since self.tail.next is updated, the list is still correctly linked, and the tail reference is the main point of confusion.

1.0.1 Key Explanation:

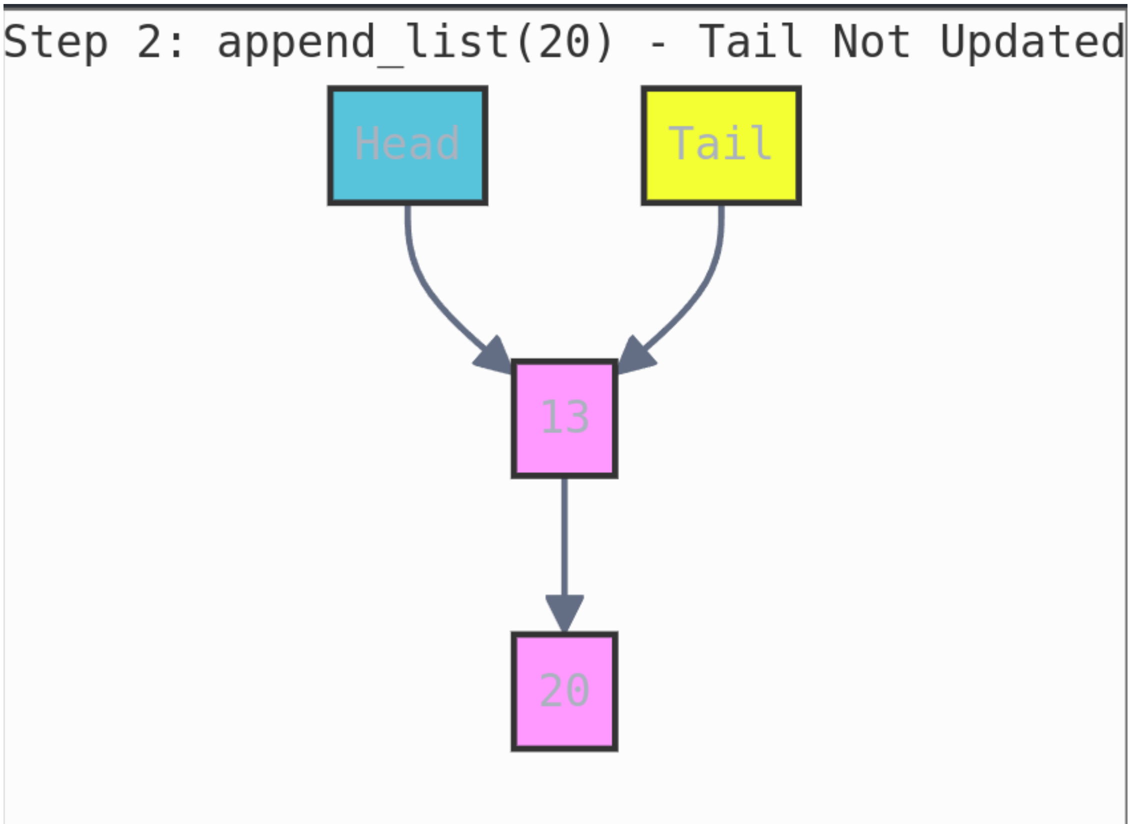
When you append the first node (13), self.head and self.tail both point to it.

- Create LinkedList(13):
- − head → [13]
- − tail → [13]



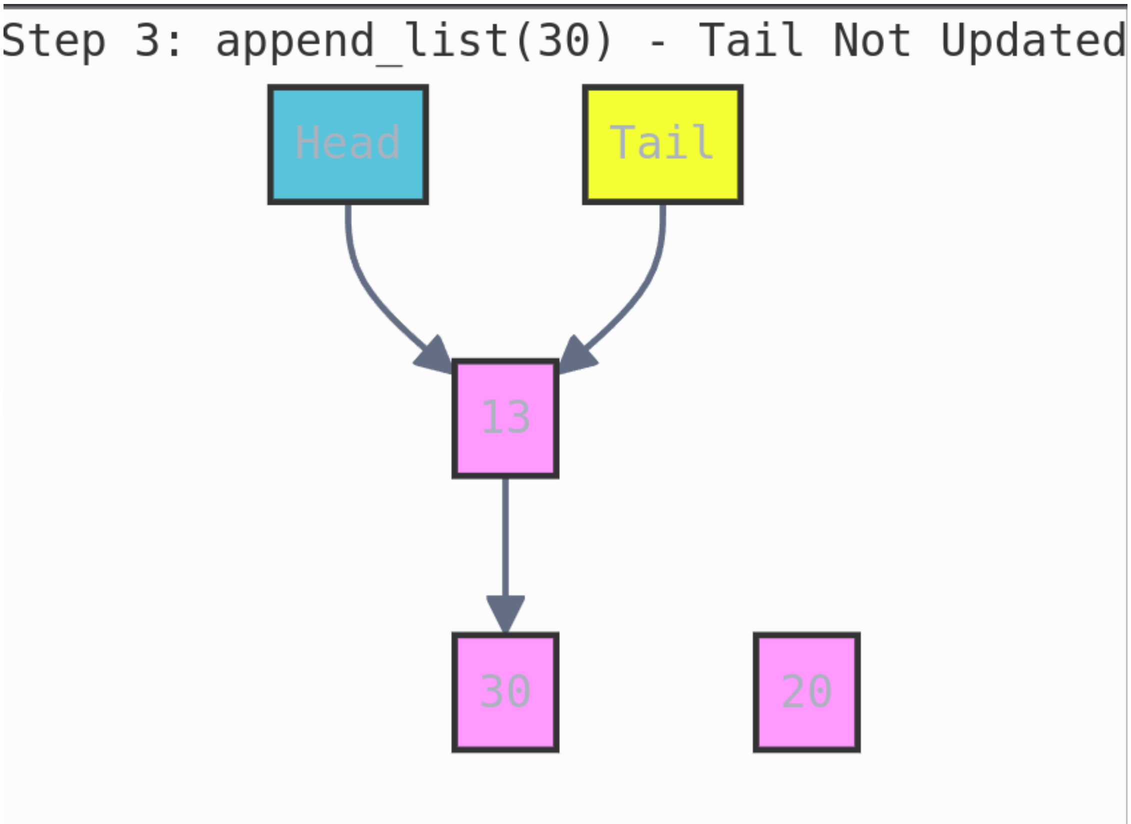
When you append 20, the self.tail.next is updated to point to 20, but self.tail itself remains pointing to 13.

- Call append_list(20):
- – Create new node [20]
- – self.tail.next = new_node means [13].next = [20]
- – tail still points to [13] (since that line is commented out)
- – Current structure: head → [13] → [20], with tail pointing to [13]



When you append 30, self.tail.next is updated to point to 30, but again, self.tail still points to the original node (13).

- Call append_list(30):
- – Create new node [30]
- – self.tail.next = new_node means [13].next = [30] (NOT [20].next!)
- – This overwrites the previous [13].next = [20]
- – tail still points to [13]
- – Final structure: head → [13] → [30], with tail pointing to [13]



In this case, the tail is not updated to the new node (30) because we commented out the line self.tail = new_node.

The key insight: Since tail never moves from [13], when you call append_list again, you're always setting [13].next to something new. (which completely overwrites the previous link) The node with value 20 still exists in memory but is no longer reachable from your list

This is why when you print the list, you only see 13 and 30. The value 20 has been "orphaned" - it's no longer in your linked list.

[22]:
```
print(my_linked_list.head.next.value)
```

30

1.1 case 2

[23]:
```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self, value):
```

```
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1

    def append_list(self, value):
        new_node = Node(value)
        # self.tail.next = new_node  # This line is commented out, so the new␣
↪node is not linked
        self.tail = new_node  # We update the tail reference to the new node
        self.length += 1

    def print_list(self):
        temp = self.head
        while temp != None:
            print(temp.value)
            temp = temp.next


# Case 2: self.tail is updated, but self.tail.next is not set
my_linked_list = LinkedList(13)
my_linked_list.append_list(20)
my_linked_list.append_list(30)

# Print the list
my_linked_list.print_list()

# Check the head and tail nodes
print("Head:", my_linked_list.head.value)
print("Tail:", my_linked_list.tail.value)
```

```
13
Head: 13
Tail: 30
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[29]:
```
class Node:
    def __init__(self, value):
```

```
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self, value):
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1
        print(f"Created list with node {value}")
        print(f"Head: {self.head.value}, Tail: {self.tail.value}")
        print(f"Structure: head → [{self.head.value}] ← tail")
        print("-" * 40)

    def append_list(self, value):
        new_node = Node(value)
        self.tail.next = new_node  # Link current tail to new node
        self.tail = new_node       # Update tail to point to the new node
        self.length += 1

        print(f"Added node {value}")
        print(f"Head: {self.head.value}, Tail: {self.tail.value}")

        # Print the current structure
        temp = self.head
        structure = "head → "
        while temp:
            structure += f"[{temp.value}]"
            if temp.next:
                structure += " → "
            temp = temp.next
        structure += " ← tail"
        print(f"Structure: {structure}")
        print("-" * 40)

    def print_list(self):
        temp = self.head
        values = []
        while temp != None:
            values.append(str(temp.value))
            temp = temp.next
        print(", ".join(values))

# Test with proper tail updates
print("PROPER IMPLEMENTATION (with tail updates):")
my_linked_list = LinkedList(13)
my_linked_list.append_list(20)
```

```python
my_linked_list.append_list(30)

print("Final list values:")
my_linked_list.print_list()
print("\n")

# Now let's simulate your original code (without tail updates)
print("BUGGY IMPLEMENTATION (without tail updates):")
class BuggyLinkedList(LinkedList):
    def append_list(self, value):
        new_node = Node(value)
        self.tail.next = new_node  # Link current tail to new node
        # self.tail = new_node     # Not updating tail!
        self.length += 1

        print(f"Added node {value}")
        print(f"Head: {self.head.value}, Tail: {self.tail.value}")

        # Print the current structure
        temp = self.head
        structure = "head → "
        while temp:
            structure += f"[{temp.value}]"
            if temp.next:
                structure += " → "
            temp = temp.next
        structure += " ← tail"
        print(f"Structure: {structure}")
        print("-" * 40)

buggy_list = BuggyLinkedList(13)
buggy_list.append_list(20)
buggy_list.append_list(30)

print("Final buggy list values:")
buggy_list.print_list()
```

```
PROPER IMPLEMENTATION (with tail updates):
Created list with node 13
Head: 13, Tail: 13
Structure: head → [13] ← tail
----------------------------------------
Added node 20
Head: 13, Tail: 20
Structure: head → [13] → [20] ← tail
----------------------------------------
Added node 30
```

```
Head: 13, Tail: 30
Structure: head → [13] → [20] → [30] ← tail
----------------------------------------
Final list values:
13, 20, 30


BUGGY IMPLEMENTATION (without tail updates):
Created list with node 13
Head: 13, Tail: 13
Structure: head → [13] ← tail
----------------------------------------
Added node 20
Head: 13, Tail: 13
Structure: head → [13] → [20] ← tail
----------------------------------------
Added node 30
Head: 13, Tail: 13
Structure: head → [13] → [30] ← tail
----------------------------------------
Final buggy list values:
13, 30
```

Table-style Loop Tracing Prompts: 1. Basic Loop Trace: Can you trace the loop in this function step by step using a table, showing variable values at each step?

2. Index-specific Loop Execution: My list has values [14, 7, 20, 30]. Can you show a table that walks through the get(3) method step-by-step, including what each variable holds during the loop?

3. Data Structure Traversal Explanation: Can you explain how the loop works in the get() method of a linked list with a table showing the current node and index at each iteration?

4. General Table Visualization Request: Please show the internal state of variables during the loop execution in table form.

Bonus Tip: You can be specific with what variables you want to track:

Show me a table of index, temp.value, and temp.next.value at each step of the loop when running get(2) on a linked list with values [10, 20, 30, 40].

[32]:
```python
# !pip install graphviz
from graphviz import Digraph

# Create a directed graph to visualize the reverse process
dot = Digraph(format='png')
dot.attr(rankdir='LR')  # Left to right layout
dot.attr('node', shape='box')

# Original List
```

```
dot.node('14', '14')
dot.node('7', '7')
dot.node('20', '20')
dot.node('30', '30')
dot.node('None1', 'None')

dot.edge('14', '7')
dot.edge('7', '20')
dot.edge('20', '30')
dot.edge('30', 'None1')

# Label for original list
dot.attr(label='Original Linked List (Before Reverse)', labelloc='top',␣
    ↪fontsize='20')
dot.render('original_linked_list', view=True)


# Now create reversed list graph
reversed_dot = Digraph(format='png')
reversed_dot.attr(rankdir='LR')
reversed_dot.attr('node', shape='box')

# Reversed List
reversed_dot.node('30', '30')
reversed_dot.node('20', '20')
reversed_dot.node('7', '7')
reversed_dot.node('14', '14')
reversed_dot.node('None2', 'None')

reversed_dot.edge('30', '20')
reversed_dot.edge('20', '7')
reversed_dot.edge('7', '14')
reversed_dot.edge('14', 'None2')

# Label for reversed list
reversed_dot.attr(label='Reversed Linked List (After Reverse)', labelloc='top',␣
    ↪fontsize='20')
reversed_dot.render('reversed_linked_list', view=True)
```

```
/snap/core20/current/lib/x86_64-linux-gnu/libstdc++.so.6: version
`GLIBCXX_3.4.29' not found (required by /lib/x86_64-linux-gnu/libproxy.so.1)
Failed to load module: /home/biswash/snap/code/common/.cache/gio-
modules/libgiolibproxy.so
```

[32]: 'reversed_linked_list.png'

```
/snap/core20/current/lib/x86_64-linux-gnu/libstdc++.so.6: version
```

```
`GLIBCXX_3.4.29' not found (required by /lib/x86_64-linux-gnu/libproxy.so.1)
Failed to load module: /home/biswash/snap/code/common/.cache/gio-
modules/libgiolibproxy.so
eog: symbol lookup error: /snap/core20/current/lib/x86_64-linux-
gnu/libpthread.so.0: undefined symbol: __libc_pthread_init, version
GLIBC_PRIVATE
eog: symbol lookup error: /snap/core20/current/lib/x86_64-linux-
gnu/libpthread.so.0: undefined symbol: __libc_pthread_init, version
GLIBC_PRIVATE
```

[ ]:

```

```