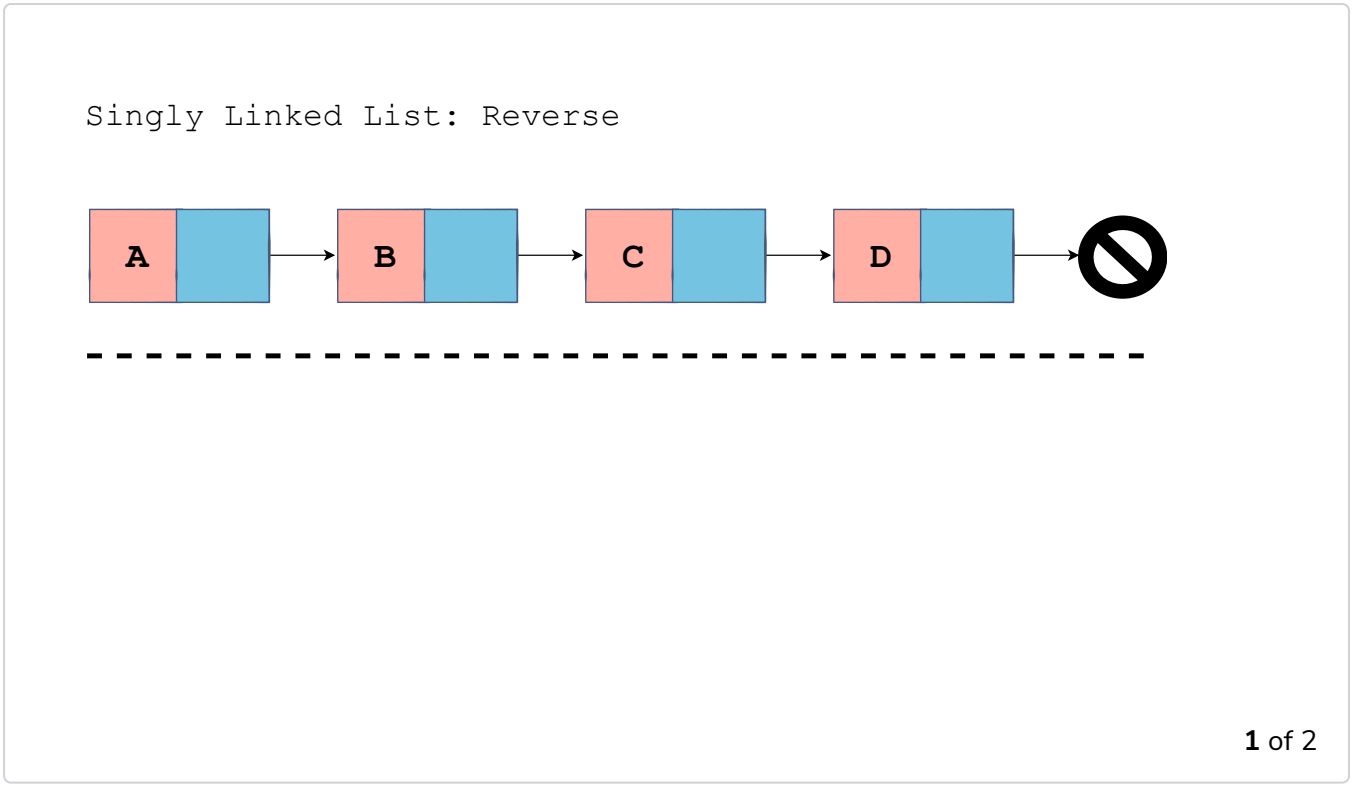
Reverse

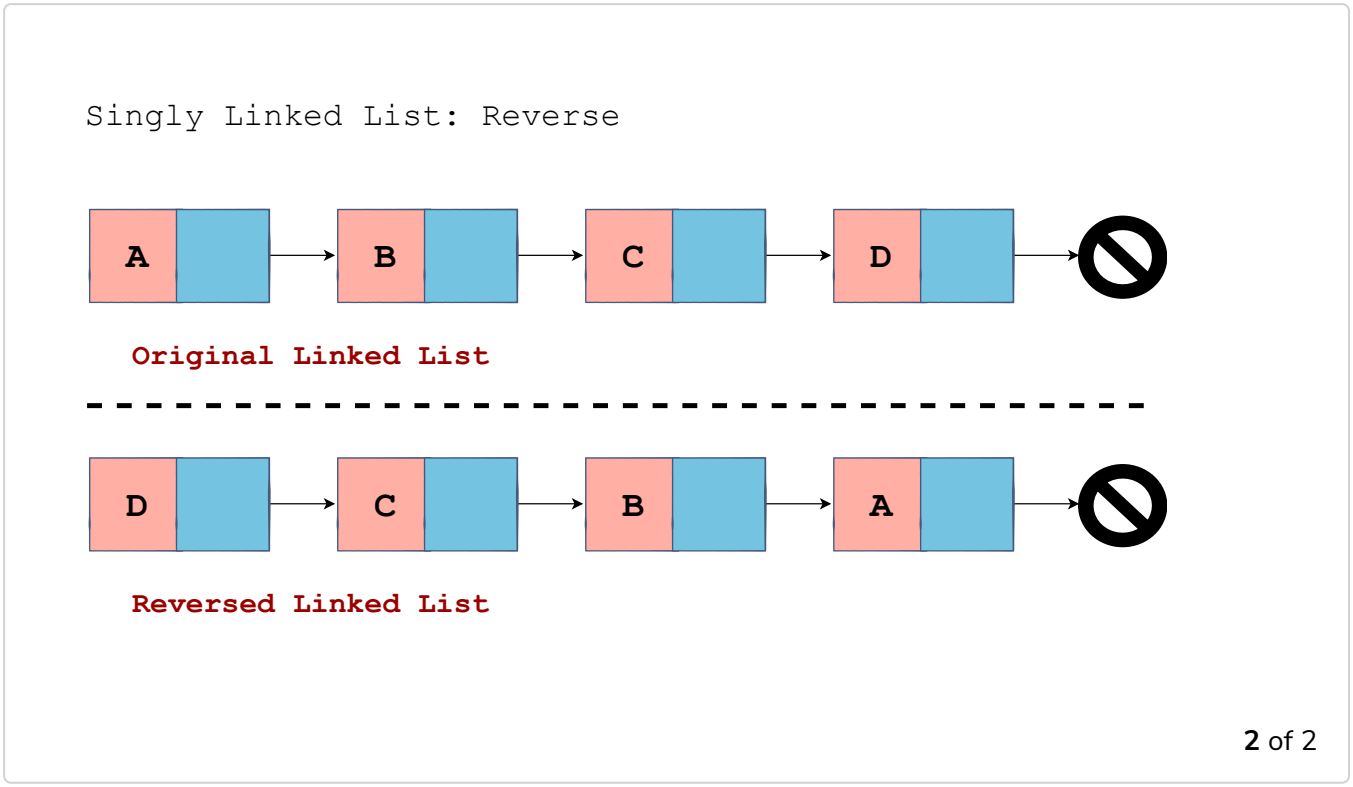
In this lesson, you will learn how to reverse a linked list in both an iterative and recursive manner.

WE'LL COVER THE FOLLOWING

In this lesson, we will look at how we can reverse a singly linked list in an iterative way and a recursive way.

Let's first be clear about what we mean by reversing a linked list. Have a look at the illustration below to get a clear idea.

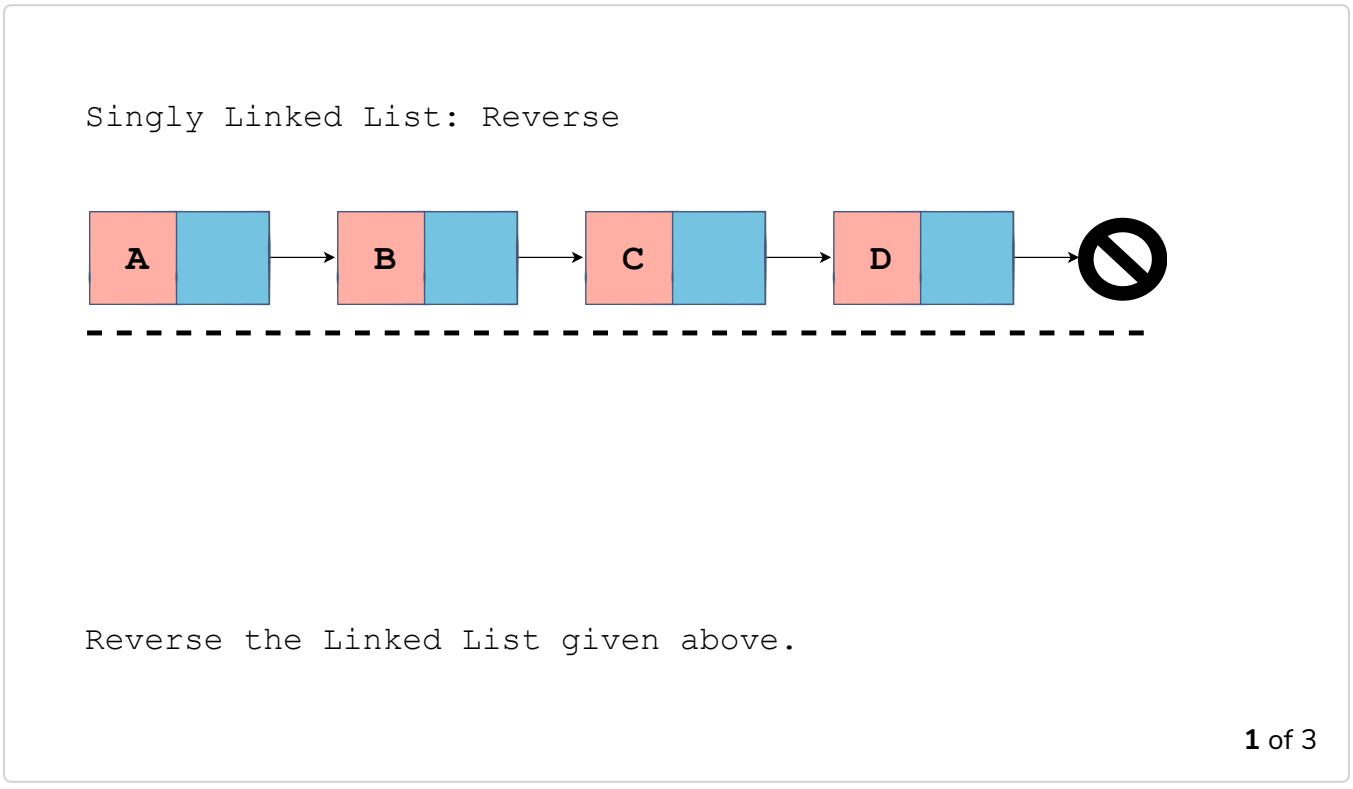




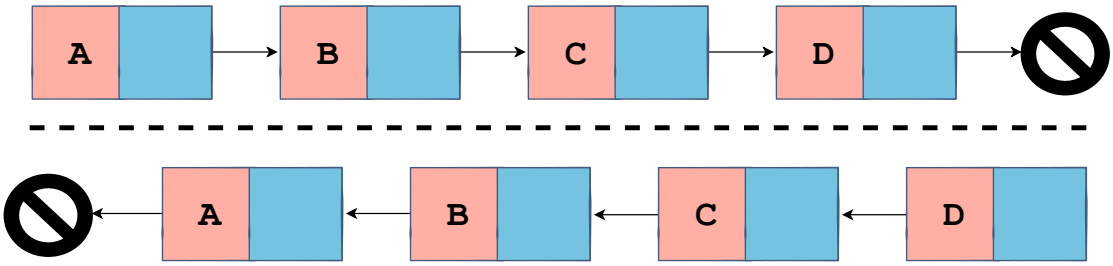
Let's discuss the algorithm to reverse the linked list as depicted in the illustration above.

Algorithm

Before jumping to the code, let's figure out the algorithm.

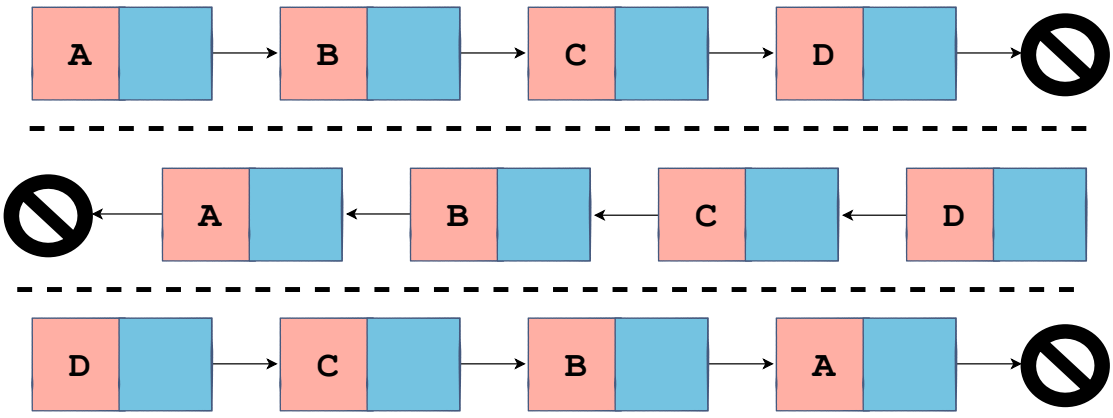


Singly Linked List: Reverse



Arrows are flipped to reverse the pointers

Singly Linked List: Reverse



Iterative Implementation

If we look at the reversal above, the key idea is that we're reversing the orientation of the arrows. For example, **node A** is initially pointing to **node B** but after we flip them, **node B** points to **node A**. The same is the case for other nodes. We can take this key observation to solve our problem and implement the method `reverse_iterative`

the method `reverse_iterative`.

```
def reverse_iterative(self):
  prev = None
  cur = self.head
  while cur:
    nxt = cur.next
    cur.next = prev
    prev = cur
    cur = nxt
  self.head = prev
```

reverse_iterative(self)

The crux of the solution will be to iterate through the linked list using the *previous and current node strategy* in the *Node swap* lesson.

On **line 2** and **line 3**, we set the initial values of `prev` and `cur` which are `None` and `self.head` respectively.

Next, we set the `while` loop on **line 4** which will run until `cur` is not equal to `None`. On **line 7** and **line 8**, we update the `prev` to `cur` and `cur` to `nxt`, i.e., `cur.next`, which help us to iterate through the linked list while keeping track of the previous and current nodes. `nxt` is used as a temporary variable to store the value of `cur.next` on **line 5** because it gets modified on **line 6**. `cur.next = prev` is the statement which does the actual work. This is because the flipping of the arrows takes place through this statement. Instead of pointing to the next node, we point the next of the current node to the previous node.

Now we just need to take care of one more thing. On **line 9**, after iterating through the linked list, `prev` is the last node in the linked list. We set `self.head` equal to the last node in the linked list. This completes our code to reverse a linked list.

Let's verify our method by making it part of the linked list implementation! You can also print out the current node and the previous node after each line in the `while` loop for a better understanding.

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

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

        self.head = None

    def print_list(self):
        cur_node = self.head
        while cur_node:
            print(cur_node.data)
            cur_node = cur_node.next

    def append(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return

        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def prepend(self, data):
        new_node = Node(data)

        new_node.next = self.head
        self.head = new_node

    def insert_after_node(self, prev_node, data):

        if not prev_node:
            print("Previous node does not exist.")
            return

        new_node = Node(data)

        new_node.next = prev_node.next
        prev_node.next = new_node

    def delete_node(self, key):

        cur_node = self.head

        if cur_node and cur_node.data == key:
            self.head = cur_node.next
            cur_node = None
            return

        prev = None
        while cur_node and cur_node.data != key:
            prev = cur_node
            cur_node = cur_node.next

        if cur_node is None:
            return

        prev.next = cur_node.next
        cur_node = None

    def delete_node_at_pos(self, pos):
        if self.head:
```

```
            cur_node = self.head

            if pos == 0:
                self.head = cur_node.next
                cur_node = None
                return

            prev = None
            count = 1
            while cur_node and count != pos:
                prev = cur_node
                cur_node = cur_node.next
                count += 1

            if cur_node is None:
                return

            prev.next = cur_node.next
            cur_node = None

    def len_iterative(self):

        count = 0
        cur_node = self.head

        while cur_node:
            count += 1
            cur_node = cur_node.next
        return count

    def len_recursive(self, node):
        if node is None:
            return 0
        return 1 + self.len_recursive(node.next)

    def print_helper(self, node, name):
        if node is None:
            print(name + ": None")
        else:
            print(name + ":" + node.data)

    def reverse_iterative(self):

        prev = None
        cur = self.head
        while cur:
            nxt = cur.next
            cur.next = prev
            prev = cur
            cur = nxt
        self.head = prev

llist = LinkedList()
llist.append("A")
llist.append("B")
llist.append("C")
llist.append("D")

llist.reverse_iterative()

llist.print_list()
```

class Node and class LinkedList

Recursive Implementation

Now that we have implemented the `reverse_iterative` method, we'll turn our attention towards the recursive implementation which is going to be similar to the iterative implementation.

```
def reverse_recursive(self):

  def _reverse_recursive(cur, prev):
    if not cur:
      return prev

    nxt = cur.next
    cur.next = prev
    prev = cur
    cur = nxt
    return _reverse_recursive(cur, prev)

  self.head = _reverse_recursive(cur=self.head, prev=None)
```

reverse_recursive(self)

The crux of any recursive solution is as follows:

- We implement the base case.
- We agree to solve the simplest problem, which in this case is to reverse just one pair of nodes.
- We defer the remaining problem to a recursive call, which is the reversal of the rest of the linked list.

Now, let's discuss the code. In `reverse_recursive` method, we define another helper method called `_reverse_recursive` with input parameters `cur` and `prev`. On **line 4**, we write the base case for the recursive method:

```
if not cur:
  return prev
```

The base case is when we'll reach the end of the linked list and `cur` is `None`, meaning `not cur` will evaluate to true. Then we'll return `prev` from the method.

The next steps on **lines 7-10** are pretty much the same as in the iterative implementation.

```
    nxt = cur.next
    cur.next = prev
    prev = cur
    cur = nxt
```

However, on **line 11**, we make a recursive call to `_reverse_recursive` method and pass `cur` and `prev` to it. This will reverse the pointers for the other nodes.

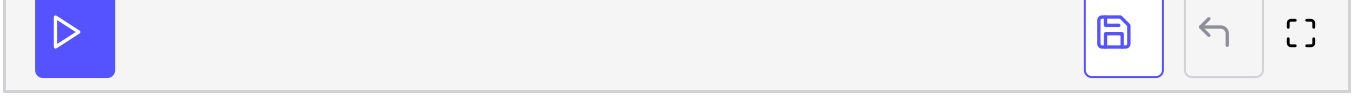
On **line 13**, we actually make a call to the helper method `_reverse_recursive` and pass `cur` as `self.head` and `prev` as `None`. Note that our base case returns `prev` which will be the last node in the linked list. Therefore, we assign `self.head` to the return argument from the `_reverse_recursive` method which will be the last node in the linked list. That concludes our implementation as the last node in the original linked list will be the head node in the reversed linked list.

Let's play around with our complete implementation and verify our method:

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


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

    def print_list(self):
        cur_node = self.head
        while cur_node:
            print(cur_node.data)
            cur_node = cur_node.next

    def append(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return

        last_node = self.head
        while last_node.next:
            last_node = last_node.next
```

```
    last_node.next = new_node

def prepend(self, data):

    new_node = Node(data)

    new_node.next = self.head
    self.head = new_node

def insert_after_node(self, prev_node, data):

    if not prev_node:
        print("Previous node does not exist.")
        return

    new_node = Node(data)

    new_node.next = prev_node.next
    prev_node.next = new_node

def delete_node(self, key):

    cur_node = self.head

    if cur_node and cur_node.data == key:
        self.head = cur_node.next
        cur_node = None
        return

    prev = None
    while cur_node and cur_node.data != key:
        prev = cur_node
        cur_node = cur_node.next

    if cur_node is None:
        return

    prev.next = cur_node.next
    cur_node = None

def delete_node_at_pos(self, pos):
    if self.head:
        cur_node = self.head

        if pos == 0:
            self.head = cur_node.next
            cur_node = None
            return

        prev = None
        count = 1
        while cur_node and count != pos:
            prev = cur_node
            cur_node = cur_node.next
            count += 1

        if cur_node is None:
            return

        prev.next = cur_node.next
        cur_node = None

def len_iterative(self):
```

```
        count = 0
        cur_node = self.head

        while cur_node:
            count += 1
            cur_node = cur_node.next
        return count

    def len_recursive(self, node):
        if node is None:
            return 0
        return 1 + self.len_recursive(node.next)

    def print_helper(self, node, name):
        if node is None:
            print(name + ": None")
        else:
            print(name + ":" + node.data)

    def reverse_iterative(self):

        prev = None
        cur = self.head
        while cur:
            nxt = cur.next
            cur.next = prev

            self.print_helper(prev, "PREV")
            self.print_helper(cur, "CUR")
            self.print_helper(nxt, "NXT")
            print("\n")

            prev = cur
            cur = nxt
        self.head = prev

    def reverse_recursive(self):

        def _reverse_recursive(cur, prev):
            if not cur:
                return prev

            nxt = cur.next
            cur.next = prev
            prev = cur
            cur = nxt
            return _reverse_recursive(cur, prev)

        self.head = _reverse_recursive(cur=self.head, prev=None)

llist = LinkedList()
llist.append("A")
llist.append("B")
llist.append("C")
llist.append("D")

llist.reverse_recursive()

llist.print_list()
```



class Node and class LinkedList

Hope everything’s clear up until now. See you in the next lesson where we are going to solve another interesting problem!