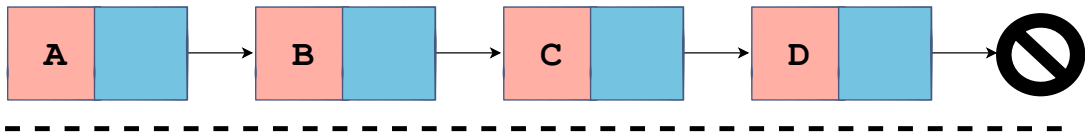
Node Swap

This lesson will teach you how to swap two nodes in a linked list.

WE'LL COVER THE FOLLOWING

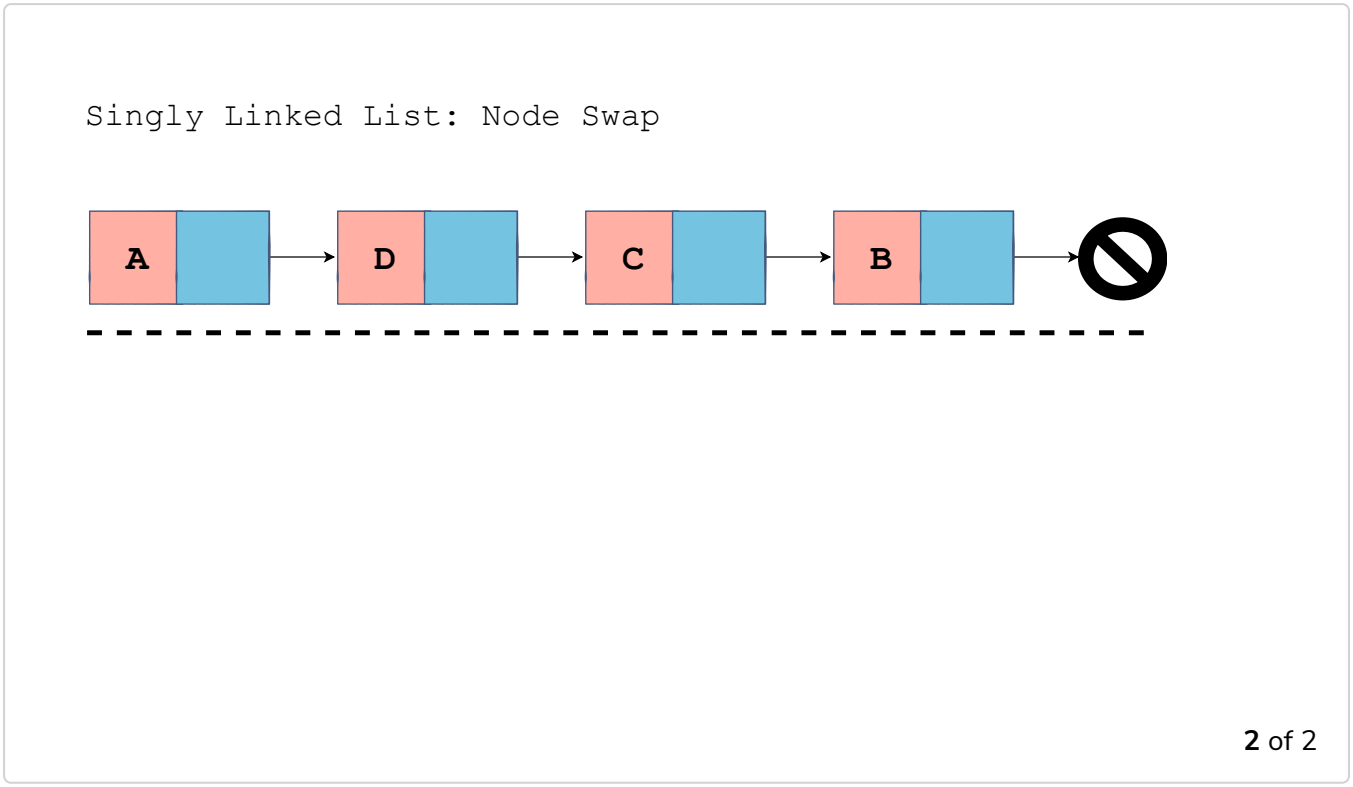
In this lesson, we will continue with our linked list implementation and focus on how to swap two different nodes in a linked list. We will give different keys corresponding to the data elements in the nodes. Now we want to swap the two nodes that contain those two keys.

Singly Linked List: Node Swap



Swap Node B with Node D

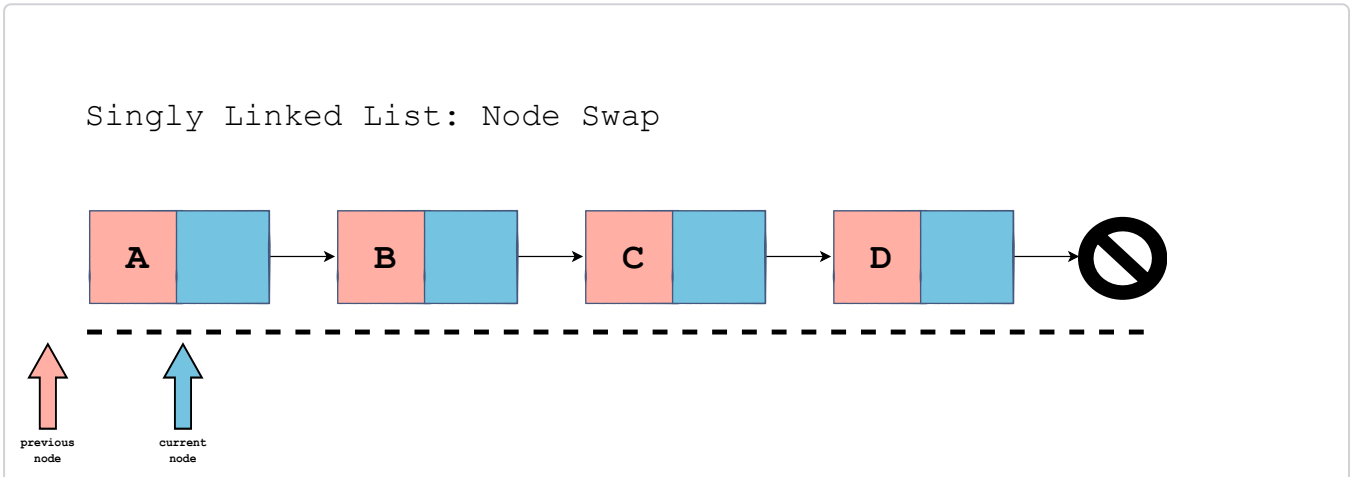
1 of 2



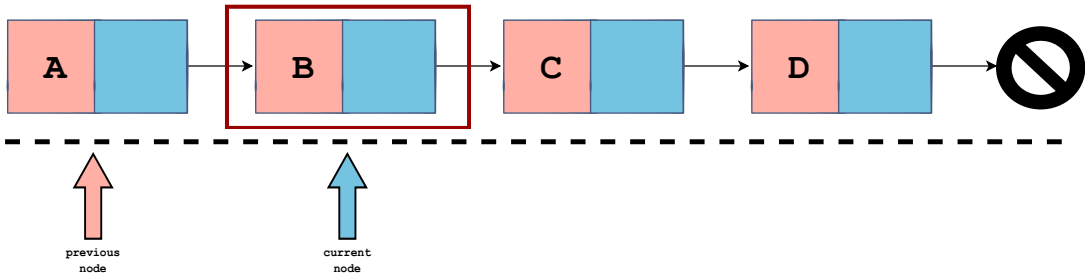
Let's go ahead and break down how we might go about solving this problem. One way to solve this is by iterating the linked list and keeping track of certain pieces of information that are going to be helpful.

Algorithm

We can start from the first node, i.e., the head node of the linked list and keep track of both the previous and the current node.

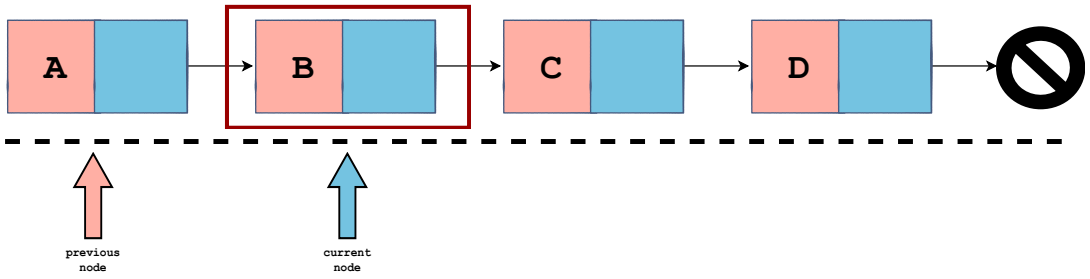


Singly Linked List: Node Swap



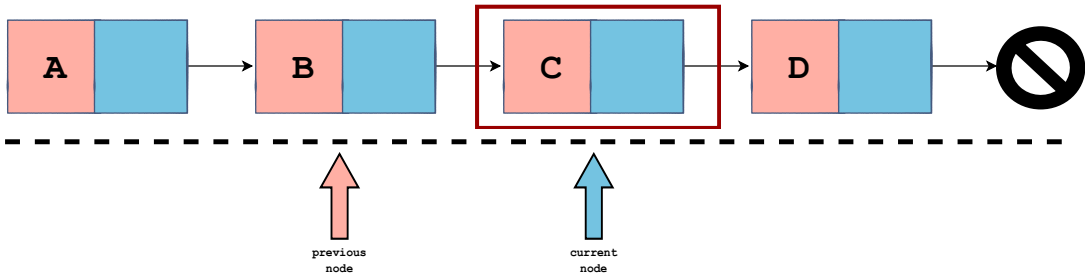
One Node Found!

Singly Linked List: Node Swap



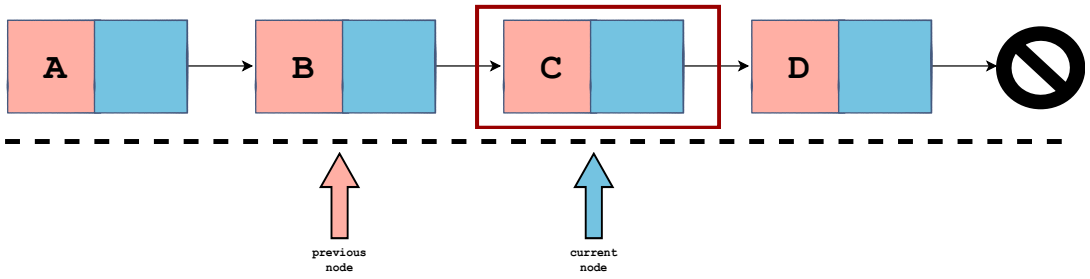
Record the current and the previous node once you find one of the nodes to be swapped while traversing
Look for the other node!

Singly Linked List: Node Swap



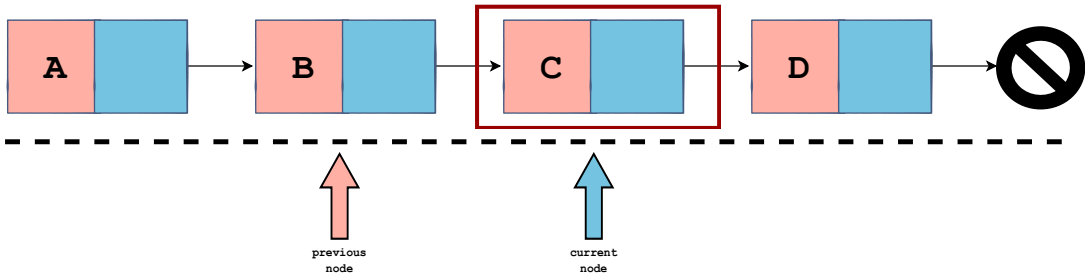
Other Node Found!

Singly Linked List: Node Swap



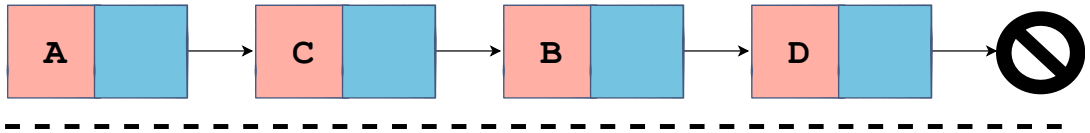
Record the current and the previous node once you find one of the nodes to be swapped while traversing

Singly Linked List: Node Swap



Now, let's swap the nodes found!

Singly Linked List: Node Swap



Node Swapped!

In the above illustration, we first set the current node to the head of the linked list and the previous node to nothing because there's no previous node to the current node. Next, we proceed through the linked list looking at the data elements and checking if the data element of the node that we're on matches one of the two keys. If we find the match, we record that information and repeat the same process for the second key that we're looking for. This is the

general way we will keep track of the information.

There are two cases that we'll have to cater for:

1. `Node 1` and `Node 2` are not head nodes.
2. Either `Node 1` or `Node 2` is a head node.

Implementation

Now let's go ahead and write up some code that will allow us to loop through this linked list and keep track of both the current and previous node for the keys given to the method.

```
def swap_nodes(self, key_1, key_2):

  if key_1 == key_2:
    return

  prev_1 = None
  curr_1 = self.head
  while curr_1 and curr_1.data != key_1:
    prev_1 = curr_1
    curr_1 = curr_1.next

  prev_2 = None
  curr_2 = self.head
  while curr_2 and curr_2.data != key_2:
    prev_2 = curr_2
    curr_2 = curr_2.next

  if not curr_1 or not curr_2:
    return

  if prev_1:
    prev_1.next = curr_2
  else:
    self.head = curr_2

  if prev_2:
      prev_2.next = curr_1
  else:
      self.head = curr_1

  curr_1.next, curr_2.next = curr_2.next, curr_1.next
```

swap_nodes(self, key_1, key_2)

Explanation

We create a method, `swap_nodes`, in the code above, which takes `key_1` and `key_2` as input parameters. First of all, we check if `key_1` and `key_2` are the

same element **(line 3)**. If they are, we return from the method on **line 4**. On **line 6** and **line 7**, we declare `prev_1` and `curr_1` to `None` and `self.head` respectively. We loop through the linked list using the `while` loop on **line 8** which runs while `curr_1` is not at the end of the linked list or it is not equal to the `key_1` that we seek. In the `while` loop, we keep updating the `prev_1` node equal to the `curr_1` and the `curr_1` to the next node in the linked list.

In the same way, we try to find if `key_2` exists in the linked list or not. We set `prev_2` equal to `None` while we set `curr_2` equal to the head of the linked list. Then again while the `curr_2` is not `None` and the `curr_2.data` is not equal to `key_2`, we update the `prev_2` and `curr_2` nodes.

On **lines 18-19**, we check to make sure that the elements we found, i.e., `curr_1` and `curr_2` actually exist or not. If either of the conditions, `not curr_1` or `not curr_2`, are not true (`curr_1` or `curr_2` is `None`) then one of them doesn't exist in the linked list. If neither key exists in the linked list or if only one of the keys exists in the linked list, we can't swap, so we `return`.

Recall the two cases that we specified while discussing the algorithm. Let's consider the case where either `curr_1` or `curr_2` is the head node. If both of the current nodes have a previous node, it implies that neither is a head node. So, we will check if the previous nodes of the current nodes exist or not. If they don't exist and are `None`, then the node without the previous node is the head node.

On **line 21**, we check if `prev_1` exists or not. If it exists, we set the `next` of `prev_1` to `curr_2` to swap it. Previously, `prev1.next` was pointing to `curr_1` but on **line 22**, we set it to point to `curr_2`. On the other hand, if `prev1` does not exist, it implies that `curr_1` is the head node and we set `self.head` to its new value, i.e., `curr_2` on **line 24**.

We repeat the same steps as above on **lines 26-29** for `prev_2` and `curr_2` and update the relevant positions with `curr_1`.

Now that we have handled the previous nodes that will point to the different nodes, we'll swap the `next` of `curr_1` with the `next` of the `curr_2` and vice versa. On **line 31**, we code this swap using the Python shorthand.

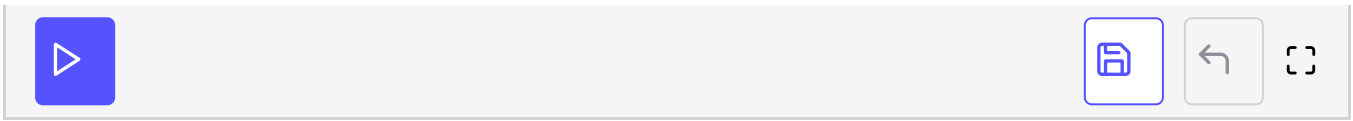
I hope you understand the above explanation. However, if the code is too hard to follow, then you can always print `curr_1`, `prev_1`, `curr_2`, or `prev_2` after

the corresponding `while` loop, so it's easy for you to follow. Now let's go ahead and verify our code in the coding widget below!

```
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 swap_nodes(self, key_1, key_2):

        if key_1 == key_2:
            return

        prev_1 = None
        curr_1 = self.head
        while curr_1 and curr_1.data != key_1:
            prev_1 = curr_1
            curr_1 = curr_1.next

        prev_2 = None
        curr_2 = self.head
        while curr_2 and curr_2.data != key_2:
            prev_2 = curr_2
            curr_2 = curr_2.next
```

```
        if not curr_1 or not curr_2:
            return

        if prev_1:
            prev_1.next = curr_2
        else:
            self.head = curr_2

        if prev_2:
            prev_2.next = curr_1
        else:
            self.head = curr_1

        curr_1.next, curr_2.next = curr_2.next, curr_1.next


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

print("Original List")
llist.print_list()


llist.swap_nodes("B", "C")
print("Swapping nodes B and C that are not head nodes")
llist.print_list()

llist.swap_nodes("A", "B")
print("Swapping nodes A and B where key_1 is head node")
llist.print_list()

llist.swap_nodes("D", "B")
print("Swapping nodes D and B where key_2 is head node")
llist.print_list()

llist.swap_nodes("C", "C")
print("Swapping nodes C and C where both keys are same")
llist.print_list()
```



class Node and class LinkedList

That's pretty much it for this lesson. `swap_nodes` is a tricky method to write because there are some edge cases that are not super obvious.

I hope this lesson was helpful for you and I'll see you in the next one.