Swapping Elements in a Linked List

Learn how to swap two nodes in a singly linked list in JavaScript.

Since singly linked lists only have pointers from each node to its next node, swapping two nodes in the list isn't as easy as doing so in an array (where you have access to the indices). You not only have to find the elements, but also reset the pointers around them to maintain the integrity of the list. This means keeping track of the two nodes to be swapped as well as the nodes preceding them.

Given an input of a linked list, `data1`, and `data2`, the general steps for doing so is as follows:

1. Iterate through the list looking for the node that matches `data1` to be swapped (`node1`), keeping track of the node's previous node as you iterate (`node1Prev`)
2. Repeat step 1 looking for the node that matches `data2` (giving you `node2` and `node2Prev`)
3. If `node1Prev` is `null`, `node1` was the head of the list, so set the list's head to `node2`
4. Otherwise, set `node1Prev`'s next node to `node2`
5. If `node2Prev` is `null`, set the list's head to `node1`
6. Otherwise, set `node2Prev`'s next node to `node1`
7. Set `node1`'s next node to `node2`'s next node
8. Set `node2`'s next node to `node1`'s next node

Finding the Matching and Preceding Nodes

Let's look at what implementing steps 1 and 2 looks like. In order to swap the two nodes, we must first find them. We also need to keep track of the nodes that precede them so that we can properly reset their pointers. (We will use the `Node` class's `.getNextNode()` method in order to access the next node.)

We will start by setting `node1` equal to the head of the list, and then creating a `while` loop that runs while `node1` isn't `null`. Inside the loop, we will check if `node1`'s data matches `data1`. If so, we `break` out of the loop as we have found the correct node. If there is no match, we update `node1Prev` to be `node1` and move `node1` to its next node:

```
function swapNodes(list, data1, data2) {
  let node1 = list.head;
```

```
  let node2 = list.head;
  let node1Prev = null;
  let node2Prev = null;

  while (node1 !== null) {
    if (node1.data === data1) {
      break;
    }
    node1Prev = node1;
    node1 = node1.getNextNode();
  }
}
```

At the end of this, we have found our matching node, and also saved its previous node, which we will use in the next step.

Updating the Preceding Nodes' Pointers

Our next step is to set `node1Prev` and `node2Prev`'s next nodes, starting with `node1Prev`. We will start by checking if `node1Prev` is `null`. If it is, then the `node1` is the head of the list, and so we will update the head to be `node2`. If `node1Prev` *isn't* `null`, then we set its next node to `node2`:

```
// Still inside the swapNodes() function
if (node1Prev === null) {
  list.head = node2;
} else {
  node1Prev.setNextNode(node2);
}
```

After this step, we have finished updating the pointers that point *to* our swapped nodes. The next step will be to update the pointers *from* them.

Updating the Preceding Nodes' Pointers

Our next step is to set `node1Prev` and `node2Prev`'s next nodes, starting with `node1Prev`. We will start by checking if `node1Prev` is `null`. If it is, then the `node1` is the head of the list, and so we will update the head to be `node2`. If `node1Prev` *isn't* `null`, then we set its next node to `node2`:

```
// Still inside the swapNodes() function
if (node1Prev === null) {
  list.head = node2;
} else {
  node1Prev.setNextNode(node2);
}
```

After this step, we have finished updating the pointers that point *to* our swapped nodes. The next step will be to update the pointers *from* them.

Updating the Nodes' Next Pointers

The last step is to update the pointers from `node1` and `node2`. This is relatively simple, and mirrors a swapping function for an array in that we will use a temporary variable.

Edge Cases

We have completed the basic swap algorithm in JavaScript! However, we haven't accounted for some edge cases. What if there is no matching node for one of the inputs? The current `swapNodes()` function will not run because we will try to access the next node of a node that is `null`. (Remember that our initial `while` loop only breaks if the matching node is found. Otherwise, it runs until the node is `null`.) Thankfully this has a quick fix. We can put in an `if` that checks if either `node1` or `node2` is `null`. If they are, we can print a statement that explains a match was not found, and `return` to end the method. We can put this right after the `while` loops that iterate through the list to find the matching nodes:

```
if (node1 === null || node2 === null) {
  console.log('Swap not possible - one or more element is not in the list')
  return;
}
```

The last edge case is if the two nodes to be swapped are the same. While our current implementation will run without error, there's no point in executing the whole function if it isn't necessary. We can add a brief check at the beginning of the function that checks if the `data1` is the same as `data2`, and then `return` to end the function:

```
if (data1 === data2) {
  console.log('Elements are the same - no swap needed.');
  return;
}
```

The Finished Function

```
const LinkedList = require('./LinkedList.js')
```

```
const testList = new LinkedList();
for (let i = 0; i <= 10; i++) {
  testList.addToTail(i);
}

testList.printList();
swapNodes(testList, 2, 5);
testList.printList();

function swapNodes(list, data1, data2) {
  console.log(`Swapping ${data1} and ${data2}:`);

  let node1Prev = null;
  let node2Prev = null;
  let node1 = list.head;
  let node2 = list.head;

  if (data1 === data2) {
    console.log('Elements are the same - no swap to be made');
    return;
  }

  while (node1 !== null) {
    if (node1.data === data1) {
      break;
    }
    node1Prev = node1;
    node1 = node1.getNextNode();
  }

  while (node2 !== null) {
    if (node2.data === data2) {
      break;
    }
    node2Prev = node2;
    node2 = node2.getNextNode();
  }

  if (node1 === null || node2 === null) {
    console.log('Swap not possible - one or more element is not in the list');
```

```
    return;
  }

  if (node1Prev === null) {
    list.head = node2;
  } else {
    node1Prev.setNextNode(node2);
  }

  if (node2Prev === null) {
    list.head = node1;
  } else {
node2Prev.setNextNode(node1);
  }

  let temp = node1.getNextNode();
  node1.setNextNode(node2.getNextNode());
  node2.setNextNode(temp);
}
```

Time and Space Complexity

The worst case for time complexity in `swapNodes()` is if both `while` loops must iterate all the way through to the end (either if there are no matching nodes, or if the matching node is the tail). This means that it has a linear big O runtime of `O(n)`, since each `while` loop has a `O(n)` runtime, and constants are dropped.

There are four new variables created in the function regardless of the input, which means that it has a constant space complexity of `O(1)`.

Two-Pointer Linked List Techniques

Learn how to approach Linked List problems with multiple iterator pointers.

Many common singly linked list problems can be solved by iterating with two pointers. This is sometimes known as the runner technique.

Two Pointers Moving in Parallel
Consider the following problem:

Create a function that returns the nth last element of a singly linked list.

In order to do this, you'll need some way of knowing how far you are from the end of the list itself. However, in a singly linked list, there's no easy way to iterate back through the list when you find the end.

If you want, you can try your hand at the problem directly, or we can walk through some approaches below.

Approaches

One thing that might first come to mind is to use an array to store a representation of the linked list. While this approach results in an easy-to-read implementation, it could also use up lots of memory maintaining a dual representation of the same data. If the linked list has one million nodes, we'll need one million pointers in an array to keep track of it! An approach like this results in an extra O(n) space being allocated.

```
const arrayNthLast = (list, n) => {
  const linkedListArray = [];
  let currentNode = list.removeHead();
  while (currentNode) {
    linkedListArray.push(currentNode);
    currentNode = currentNode.getNextNode();
  }
  return linkedListArray[linkedListArray.length - n];
}
```

Instead of creating an entire parallel list, we can solve this problem by using two pointers at different positions in the list but moving at the same rate. As in the previous example, we will use one pointer to iterate through the entire list, but we'll also move a second pointer delayed `n` steps behind the first one.

```
nthLastNodePointer = null
tailPointer = linked list head
count = 0

while tail pointer exists
  move tail pointer forward
  if count >= n
    set nthLastNodePointer to head if it's still null or move it forward
  increment count

return nthLastNodePointer
```

Implementation

Complete the `nthLastNode()` function so that it returns the correct `Node` instance (or `null` if the `linkedList` is shorter than `n` elements). Do this without creating any additional new data structures (such as an array).

You can use the `testLinkedList` to experiment yourself. It contains a 50-length linked list with `data` values from `50 -> 49 -> ... 2 -> 1`

```
const LinkedList = require('./LinkedList');
const testLinkedList = require('./testLinkedList.js');
// Complete this function
const nthLastNode = ( linkedList, n) => {
  let current = null;
  let tailSeeker = linkedList.head;
  let count = 0;
  while (tailSeeker) {
    tailSeeker = tailSeeker.next;
    if (count >= n) {
      if (!current) {
        current = linkedList.head;
      }
      current = current.next;
    }
    count++
  }
  return current;
};

// Test your function yourself:
console.log(nthLastNode(testLinkedList, 4));

// Leave this so that we can test your code:
module.exports = nthLastNode;
```

Solution

In JavaScript, we could implement the nth-last-node-finder function as such:

```
const nthLastNode = (linkedList, n) => {
  let current = null;
  let tailSeeker = linkedList.head;
  let count = 0;
  while (tailSeeker) {
    tailSeeker = tailSeeker.next;
    if (count >= n) {
      if (!current) {
        current = linkedList.head;
      }
      current = current.next;
    }
```

```
    count++
  }
  return current;
}
```

The exact variable names aren't important, and the internal implementation could be written in a number of ways, but the important part is that we are able to complete this problem efficiently–in O(n) time (we must iterate through the entire list once), and O(1) space complexity (we always use only three variables no matter what size the list is: two pointers and a counter).

Pointers at Different Speeds

Another two-pointer technique involves sending pointers through the list at different iteration "speeds".

Consider this problem:

Find the middle node of a linked list.

Approaches

As before, it's possible to find a solution by iterating through the entire list, creating an array representation, and then returning the middle index. But as before, this potentially takes up lots of extra space:

```
create array
while the linked list has not been fully iterated through
  push the current element onto array
  move forward one node
return array[length / 2]
```

Instead, we can use two pointers to move through the list. The first pointer takes two steps through the list for every one step that the second takes, so it iterates twice as fast.

```
fastPointer = list head
slowPointer = list head
while fastPointer is not null
  move fastPointer forward
  if the end of the list has not been reached
    move fastPointer forward again
    move slowPointer forward
return slowPointer
```

When the first pointer reaches the end of the list, the "slower" second pointer will be pointing to the middle element. Let's visualize the steps of the algorithm:

Starting State

```
F
S
1  2  3  4  5  6  7
```

First Tick

```
      F
   S
1  2  3  4  5  6  7
```

Second Tick

```
          F
      S
1  2  3  4  5  6  7
```
Third Tick

```
             F
       S
1  2  3  4  5  6  7
```
Final Tick

```
                F
       S
1  2  3  4  5  6  7  null
```
As long as we always move the fast pointer first and check to see that it is not null before moving it and the slow pointer again, we'll exit iteration at the proper time and have a reference to the middle node with the slow pointer.

Implementation

Complete the `findMiddle()` function and return the middle node of `linkedList`. You can assume that the list has no cycles.

Return the right-weighted middle for even-length lists. For example, in a list of 4 elements, return the element at index `2` (which would be the element `3`).

Use `generateTestLinkedList(length)` to generate linked lists with `data` from `1 -> 2 -> .. -> length` to test out your function. For instance, `generateTestLinkedList(4)` results in `1 -> 2 -> 3 -> 4`.

```
const generateTestLinkedList = require('./generateTestLinkedList');

const findMiddle = linkedList => {
  let fast = linkedList.head;
  let slow = linkedList.head;

  // As long as the end of the list is not reached
  while (fast !== null) {
    // Move the fast pointer at least one step
    fast = fast.getNextNode();
    // If it isn't at the end of the list
    if (fast !== null) {
      // Move both pointers forward once
      fast = fast.getNextNode();
      slow = slow.getNextNode();
    }
  }
```

```
  // At this point, the slow pointer is in the middle
  return slow;
};

// Test your function yourself:
console.log(findMiddle(generateTestLinkedList(7)));

// Leave this so that we can test your code:
module.exports = findMiddle;
```

Solution and Alternatives

```
const findMiddle = linkedList => {
  let fast = linkedList.head;
  let slow = linkedList.head;

  // As long as the end of the list is not reached
  while (fast !== null) {
    // Move the fast pointer at least one step
    fast = fast.getNextNode();
    // If it isn't at the end of the list
    if (fast !== null) {
      // Move both pointers forward once
      fast = fast.getNextNode();
      slow = slow.getNextNode();
    }
  }
  // At this point, the slow pointer is in the middle
  return slow;
};
```

As with the nth-to-last solution, this solution has O(n) time complexity, and O(1) space complexity, since only two nodes are created no matter the size of the input list.

Half-Speed

Another equally valid solution is to move the fast pointer once with each loop iteration but only move the slow pointer every-other iteration.

```
const findMiddleAlternate = linkedList => {
  let count = 0;
  let fast = linkedList.head;
  let slow = linkedList.head;

  while(fast !== null) {
    fast = fast.getNextNode();
    if (count % 2 !== 0) {
      slow = slow.getNextNode();
    }
    count++;
  }
  return slow;
}
```

Conclusions

Many linked list problems can be solved with the two-pointer technique. If it seems like a linked list problem requires keeping track of multiple positions or creating other data

representations (such as using an array), consider whether two pointers iterating in parallel or at different speeds could help solve the problem efficiently. We won't cover full solutions to these here, but variations on the two-pointer technique can be used to:

- Detect a cycle in a linked list
- Rotate a linked list by k places