Linked Lists

NOTES. This document was created by Sasha Avreline in the summer of 2020 as an extra study aid to BCS students taking CPSC 221. All code can be found at

`github.com/savreline/tutorials/tree/master/cpp/lists`

The problem considered in section 2.2 is LeetCode problem 206.

Contents

1 Basic Implementation Examples

Key Implementation Steps

1. Plan. Draw a diagram. Determine which nodes you need to get to and think about how to get there. Determine which pointers need to be re-linked.
2. Write the most basic implementation first. It is a good idea to give good, descriptive names to all of the variables at this point. E.g. don't use `node` or `temp` for everything, use something like `secondLastNode` instead.
3. Finally handle the edge cases. Typical edge cases are: empty list, single-element list, two-element list.

Tips

1. Make sure to handle all potential null pointers and *guard* against them. Anywhere were you are dereferencing something that is potentially a null pointer is likely to be an edge case.
2. Make sure to re-link pointers in the correct order.
3. Test simple cases first!

We will first illustrate how those steps and tips apply to simple examples such as insertions and deletions. Then we will do a much more involved example of swapping two nodes in a doubly-linked list.

In all cases we are working with roughly the following definition of a linked list, except that we do not have the previous pointer and/or the tail pointer in most cases. Note that the constructor creates the node with both next and previous pointers set to NULL by default, so we don't need to explicitly do this in the following examples.

Code for the singly-linked list can be found at
`github.com/savreline/tutorials/tree/master/cpp/lists`

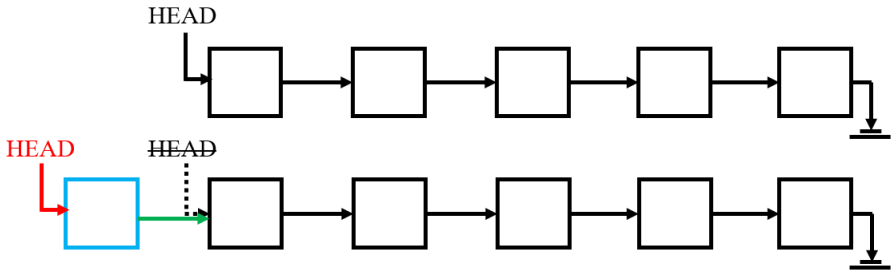
```
struct Node {
    int val;
    Node *next;
    Node *prev;
    Node(int val) :
        val(val), next(NULL), prev(NULL) {}
};
```

```
class LinkedList {
    Node *head;
    Node *tail;
    void insert(int val);
    ...
    // all other methods
}:
```

1.1 Singly-Linked List: Insertion at the Front

The diagram is show below. In this case we are inserting at the head, so we do not need to traverse the list to get to any specific node. We just need to create the new node and re-link two pointers.

STEP 1: Create the new node.
STEP 2: Point the node's next pointer to the existing head.
STEP 3: Re-assign the head pointer to point to the newly created node.




```
void insertFront(int val) {
   Node *node = new Node(val);
   node->next = head;
   head = node;
}
```

Now that we got the basic implementation ready, we should check if there are any edge cases in this case. What would happen to the above code in the case of an empty list? The existing head in that case would be null, so technically line 2 is not necessary in that case (we would be setting `node->next` to null, which is already null via the constructor). However, executing line 2 in that case wouldn't crash the code, it would just be a bit of unnecessary extra work. So, for simplicity, we just leave this line as is and do not consider the empty list as a special case here.

A word on testing: it is a good idea to test simple cases first. Once you are done with the method's implementation, first test if the method can handle very simple lists such as the ones that are empty or one-element long. That would point out potential issues with the method early on. Then move on to test complex lists.

1.2 Singly-Linked List: Insertion at the Back

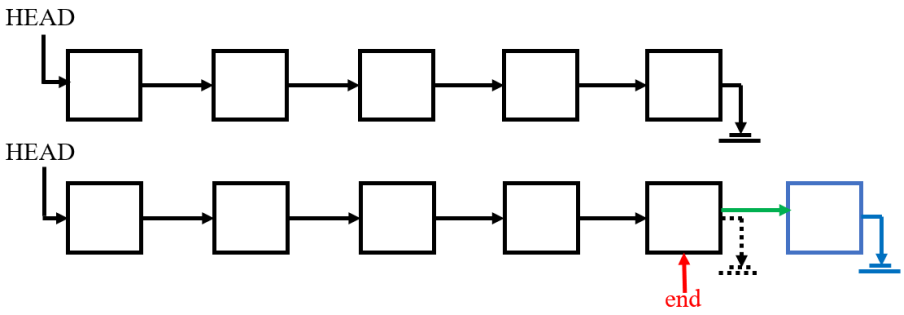
In this case we are inserting at the back, so we need to travel to the end of the list. We do this using a while loop of course (while loops are typically used with linked lists since we do not know how long the lists are). We save the head pointer into a new pointer called `end` and advance that pointer through the list until we are at the last node.

We know we are at the last node when the `end`'s next pointer is null. So we run the while loop while `end->next` is not null. We don't actually want to run a `while(end)` loop here as that would just advance the pointer beyond the last node and `end` would be null when the loop exists. Once we reach the end, we just need to point the last node's next pointer to the

new node.

Note that of course we must not use the head pointer to traverse the list - i.e. we should not run a `while(head)` loop here. That would be good enough to run this method, but then the location of the head wouldn't be correct for all methods going forward. So, the first thing we do is save a copy of the head into a new pointer called `end`. It is a good idea to name this pointer `end` as then we know what it is suppose to represent.

STEP 1: Create the new node.
STEP 2: Travel to the end of the list.
STEP 3: Point the last's node next pointer to the new node.




```
void insertBack(int val) {
   Node *node = new Node(val);

   if (!head) {
      head = node;
      return;
   }

   Node *end = head;
   while (end->next)
      end = end->next;
   end->next = node;
}
```

Note that there is an edge case here. If the list is empty, then head is null, so `end` is null too and `while(end->next)` would deference a null pointer when checking the loop condition. That would give us a segmentation fault. So we do want to handle this special case right away as is done in purple above.

Contrast this with the previous example: there we were never deferenced any pointers at all, we only set the `node->next` pointer. So anytime where we deference something that is potentially a null pointer, like we do in the `while(end->next)` line here, that hints we are looking at an edge case that we need to handle separately.

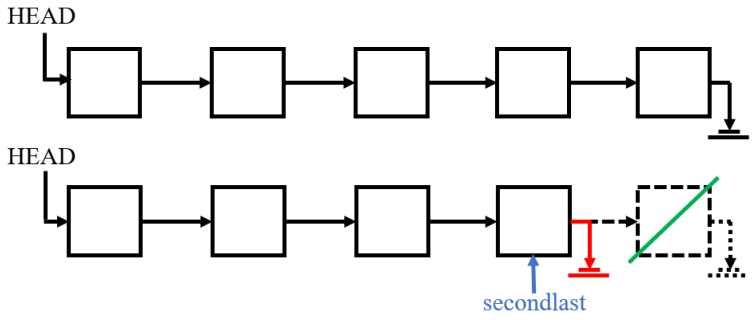
1.3 Singly-Linked List: Delete at the Back

In order to delete the last node, we need to zero-out the pointer that points to it. By looking at the diagram that we drew, we see that we need access to the *second* last node this time, as that is the one that stores the pointer that we need.

So we once again run a while loop to get to the node that we need. This time we save a copy of the head pointer into a pointer that we call `secondlast` (once again, using descriptive names). This time we want to stop when `secondlast->next->next` is null, which is equivalent to saying that the next node still exists, but its next doesn't.

Once we are at the second last node, we can just delete its next node (which is the one we wanted to delete in the first place) and zero-out the next pointer. Note that the order matters a lot here: if we were to zero-out the pointer first, we would lose access to the last node and cause a memory leak.

STEP 1: Travel to the second last node.
STEP 2: Delete the last node, which is pointed to by the second last node.
STEP 3: Set the second last node's next pointer to null.




```
void removeBack() {
   if (!head) return;
   if (!head->next) {
      delete head;
      head = NULL;
      return;
   }

   Node *secondlast = head;
   while (secondlast->next->next)
      secondlast = secondlast->next;
   delete secondlast->next;
   secondlast->next = NULL;
}
```

There are *two* edge cases here. This is hinted at by the `while(secondlast->next->next)` condition. There we have potential to deference not one, but two successive null pointers. In particular, if the head is null, then `secondlast` is null or if the `head->next` is null then

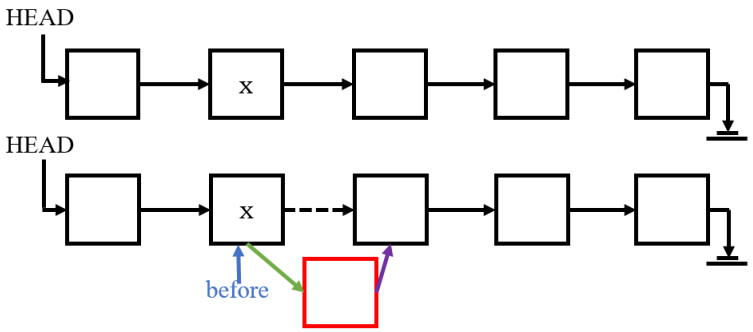
`secondlast->next` is null, so we can't deference that to the second `->next` either.

So the two edge cases are exactly those: (1) when the head is null and (2) when `head->next` is null, corresponding to case of an empty and single-element list respectively. They are handled in magenta and purple respectively. In the case of an empty list, there is nothing to delete, so we just return. In the case of a single element list, we just delete the head.

1.4 Singly-Linked List: Insertion in the Middle

In this case we would like to insert a node immediately after some node whose value is `x`. So the first step would be to locate that node whose value is `x`, as it is *its* next pointer that we would need to modify. So we save a copy of the head pointer into a pointer called `before` (once again using descriptive names here) and run the `while(before->val != x)` loop which would advance the `before` pointer through the list until we hit the node that actually has the value `x`.

Note that the loop also need to have the `while(before)` condition. Otherwise we risk dereferencing a null pointer when checking `before->val != x`. That would happen when `x` doesn't exist in the list and `before` advances to the end of the list. Here we are also using the trick of how the AND expressions are evaluated: the first condition is checked independently and if it is false, then the while loop would exist right away. The take away message; however, is that whenever there is a potential for a null pointer to show up, we always need to *guard* against that.




```
void insertAfter(int val, int x) {
   Node *before = head;
   while (before && before->val != x)
      before = before->next;

   if (before) {
      Node *node = new Node(val);
      node->next = before->next;
      before->next = node;
   }
}
```

Once we are at the correct node, we just need to re-link two pointers (purple and green). Once again, order matters, we must use `before->next` before we re-assign it. Note that

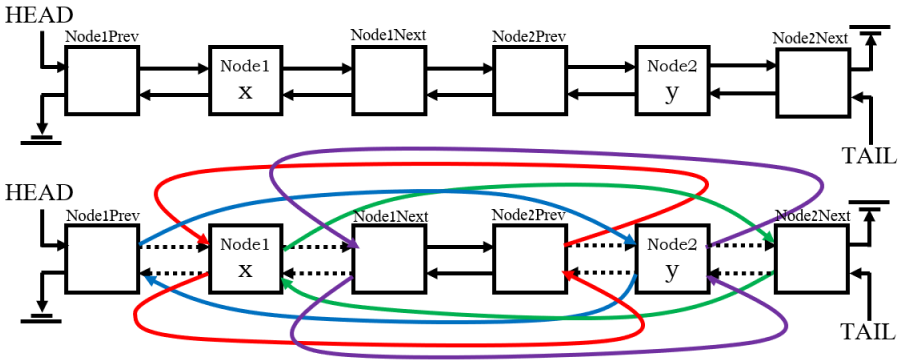
we create the new node only *inside the if statement*. Otherwise, if `before` is null and we didn't find a place to insert the new node, the node that we created wouldn't be attached anywhere, wouldn't be used and that is a memory leak!

Beyond what we have there are no other edge cases here. The edge case of a null head and an empty list are already handled with the `while(before)` clause!

1.5 Doubly-Linked List: Swap

The goal here is to swap two nodes in a doubly linked list whose values are `x` and `y`. Where do we start? A diagram would help for sure! Which diagram should we draw however? Is one of the nodes at the head, at the tail, or is this a single-element list? Let's sort of go for the most complex diagram that we can have: swapping two nodes that are both in the middle of the list and away from each other.

Based on that diagram we see that we need to re-link a total of eight pointers. In the diagram it is useful to label certain nodes as "Node1Prev", "Node1Next", etc. as this would help to keep track of things when we need to write actual code. When writing the code, we also need to first locate the nodes, and here we can just use the while loops similar to the one in the previous method. The first attempt at code is given below.




```
void swap(int x, int y) {
   // Locate the nodes
   Node *node1 = head;
   Node *node2 = head;
   while (node1 && node1->val != x)
      node1 = node1->next;
   while (node2 && node2->val != y)
      node2 = node2->next;

   // Not found
   if (!node1 || !node2)
      return;
```

```
    // Save some values
    Node *node1next = node1->next;
    Node *node1prev = node1->prev;

    node1->prev->next = node2;
    node1->next->prev = node2;
    node1->next = node2->next;
    node1->prev = node2->prev;

    node2->next->prev = node1;
    node2->prev->next = node1;
    node2->next = node1next;
    node2->prev = node1prev;
}
```

Note that order matters here as we reset the original values of `node1->next` and `node1->prev` in the first code block before we need to use the original ones in the second code block. So we must save those values into temporary pointers as is done in olive above.

Now what about the case where either of the nodes or both are at the head or at the tail? Sure we can write up a bunch of different if statements that handle all of those cases separately. That would make a really long and convoluted method! However, here we can just get away with guarding against null pointers. Essentially, if something like `node1->prev` is not null (i.e. node 1 is not the head), then it makes sense to dereference `node1->prev` to then go ahead and set `node1->prev->next`. Otherwise, we shouldn't follow `node1->prev`. The next attempt at this section of the code, with the guards, is shown below.

```
if (node1->next)
    node1->next->prev = node2;
if (node1->prev)
    node1->prev->next = node2;
node1->next = node2->next;
node1->prev = node2->prev;

if (node2->next)
    node2->next->prev = node1;
if (node2->prev)
    node2->prev->next = node1;
node2->next = node1next;
node2->prev = node1prev;
```

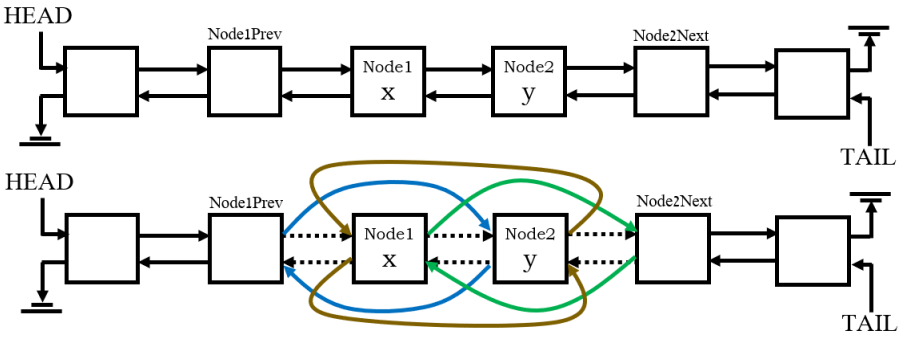
So this is where the power of guarding against null pointers and collapsing many cases into one when appropriate really simplifies the code!

We can also use this technique to catch the cases where we need to re-assign the head or the tail pointers, as is shown below.

```
if (!node1->prev)
    head = node2;
if (!node2->next)
    tail = node1;
```

Are we done? There is one edge case that we missed! If nodes 1 and 2 are adjacent, then something like the line `node1->prev = node2->prev` would point right back to node 1 (as node 1 is the previous of node 2 when they are adjacent) and result in an infinite loop!

So here we would need to draw a new diagram and write separate code to handle this special case. From the diagram we learn that there are just six pointers to re-link this time. We would also need to have some code to detect if two nodes are indeed neighbors. All of this code is presented below.




```
// Detect adjacent
bool adjacent = false;
if (node1->next == node2 || node2->next == node1)
    adjacent = true;

if (!adjacent) {
    // code as before

} else {
    if (node1->prev)
        node1->prev->next = node2;
    node1->next = node2->next;
    node1->prev = node2;

    if (node2->next)
        node2->next->prev = node1;
    node2->next = node1;
    node2->prev = node1prev;
}
```

The one problem that still remains is that in all diagrams we assumed that node 2 follows node 1. What if they were the other way around? Is there a simple fix? Yes! We could just swap the pointers to the two nodes, if need be, so that node 2 always follows node 1.

The previous example showed the need to think of all possible edge cases to test! A word on testing doubly-linked lists: it is a good idea to write methods that would print the list in both directions. Specifically, it is a good idea to have a "print forward" method that would start from head and work towards the tail and then a "print backward" method that would start from the tail and work towards the head. This way you can detect more potential issues if some pointers in just one direction weren't linked up properly.

2 Iterative vs. Recursive Approach

2.1 Construct List from An Array

Iterative Approach. We are given a primitive C array of integers and would like to construct a linked list from it. Our method is also given the `size` of the array as input, as primitive C arrays are so simple, they don't know their size.

So in this method it actually makes sense to use a for loop, as we do know how long the array is. The following code is a good start: we loop through the array and create a node for each of the array's elements.

```
void copyArray(int array[], int size) {
    for (int i = 0; i < size; i++) {
        Node *node = new Node(array[i]);
    }
}
```

However, we are of course just creating the nodes and are not linking the list. The issue is that when we create the node, on that iteration, we haven't yet created its successor to link the next pointer to. We do that on the following iteration! So we need to store some reference to the node that we created that would be available on the following iteration. Lets introduce a pointer called `before` to do that. We might as well also call the current node `curr` to keep up with using descriptive names. We make the link in red font on the next page.

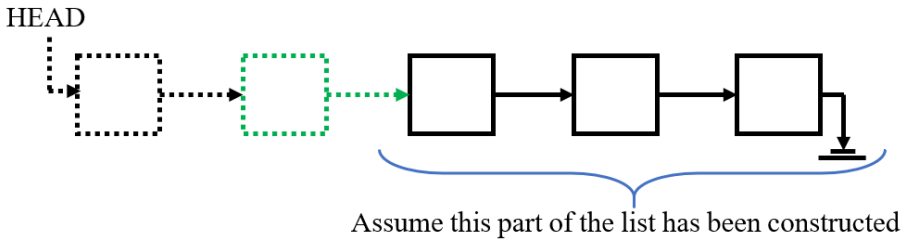
As soon as we introduce a pointer like `before`, we need to be sure we keep advancing it throughout the loop. This is done in green on the next page.

Finally, we need to set the head pointer. This is done when i is 0, on the first iteration of the loop. See purple on the next page.

Of course, everything we discussed before applies. It is important not to deference `before` when it is null, so we guard against that. It is also important to do things in the correct order: we use `before` first, then we advance it at the very end of the iteration!

```
void copyArray(int array[], int size) {
   Node *curr = NULL, *before = NULL;

   for (int i = 0; i < size; i++) {
     curr = new Node(array[i]);

     if (before)
       before->next = curr;

     if (i == 0)
       before->next = curr;

     before = curr;
   }
}
```

Recursive Approach. With linked lists, the typical assumption in recursive approaches is that when the recursive call returns, something is true about the *tail* portion of the list. In this particular case we can say that when the recursive call returns, *the tail portion of the list is fully constructed*: see the sketch below. So we just need to figure out how to add on the front portion of the list.



We extend the method's signature to keep passing the "list so far" to the recursive calls (once again, descriptive names)! So we sort of have two variables that keep changing between recursive calls. The size of the array would *shrink* with each recursive call, until we hit the base case, and the size of the "list so far" would continue to *grow*. This way we would work through the array until the array "disappears" and the list is fully constructed.

```
void copyArray(int array[], int size, Node *listsofar) {
   if (size == 0) {
     head = listsofar;
     return;
   }

   Node *node = new Node(array[size - 1]);
   node->next = listsofar;
   copyArrayRecur(array, size - 1, node);
}
```

The base case is in purple. Here, once we reach the size of zero, we can say that the entire list has been constructed, so we just set the head to point to that list.

In red we make the recursive call: we decrease the array's size by one. We also need to increase the size of the list prior to making the recursive call. This is done in green. This is also where we figure out how to attach the head onto the "list so far" (which we assume is a properly constructed list coming out of the recursive calls).

The `size - 1` in `new Node(array[size - 1])` is just to account for zero-based indexing. The initial call to the method could be done with passing in `NULL` as the argument for the "list so far": `copyArrayRecur(array, size, NULL)`.

2.2 Reverse a Linked List

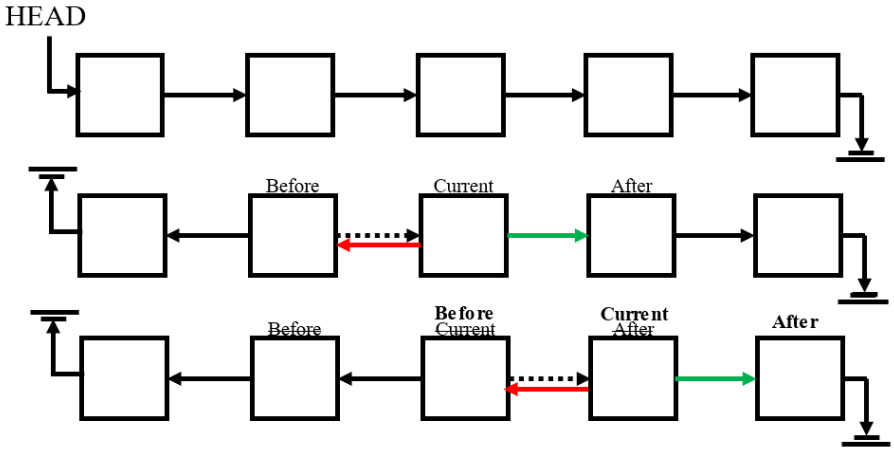
This is LeetCode problem no. 206! In this problem we are being asked to reverse a singly-linked list.

Iterative Approach. The key to this problem is once again to draw a diagram! Looking at the diagram we see that what we need to do is to flip something like the green pointer to something like the red pointer. So we see that there are *three* nodes involved.

We flip the pointer on the `current` node and point it to the `before` node. However, we also need to grab a pointer to the `after` node *first* as once we flip the green pointer back, we lose access to the `after` node (imagine if the green pointer wasn't there).

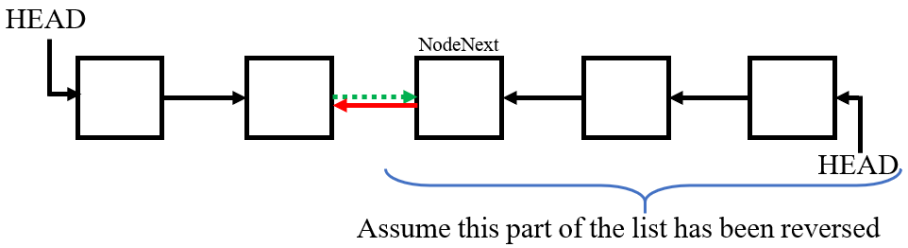
As seen on the following page, we initialize the three pointers and then just do a while loop to traverse the list. On each iteration we flip one pointer back (in red) and advance all three of our pointers (in green) - also see the third diagram below. The order in which we do things is of course very important: we advance the `after` first, then we flip the pointer. By flipping the pointer back we destroyed the link to the next node; however, we already have that saved in the `after` pointer! Finally we advance the other two pointers.

We also need to return the head, and once the entire loop is done, the head actually happens to be the `before` node (as `curr` is already past the end of the list), so we set that in purple below.



```
void reverse() {
    Node *before = NULL, *after = NULL, *curr = head;

    while (curr) {
      after = curr->next;
      curr->next = before;
      before = curr;
      curr = after;
    }

    head = before;
}
```

Recursive Approach. As before, we are going to assume something about the tail of the list. This time we are going to assume that *the tail been reversed*. So we just need to figure out how to reverse the front part of the list and work that into the recursive structure.



A big issue with this approach is that we have to start our ***work from the tail***, yet we don't have access to the tail. In the previous case we did, there we just started working with elements at the end of the array via `array[size]`. In this case we need to actually get to the tail of the list first! So we need to ***hold back the work done in the recursive calls***.

We will open up all of the recursive calls via the line in cyan, opening up one recursive call per element of the list. Once all of them have opened up, there would be one function frame on the calling stack per element of the list, holding the corresponding local `node` variable on the stack - see diagram on the following page.

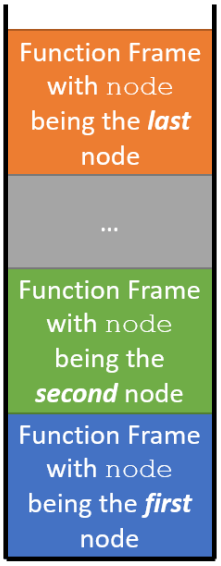
```
void reverseRecurHelper(Node *node) {
   if (!node || !node->next) {
     head = node;
     return;
   }

   reverseRecurHelper(node->next);
   node->next->next = node;
   node->next = NULL;
   return;
}
```

The base case would be reached once we are at the tail of the list and at that point we can assign our head to be there (since we are reversing the list). This is done in purple.

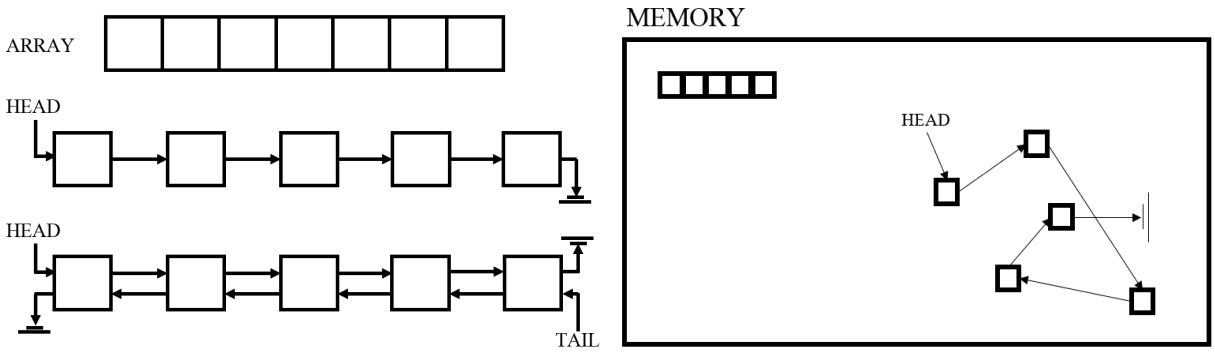
Now, when the calls start coming back, we will start flipping the pointers. This is done in red and green to match the pointers in the diagram. It is very critical that those lines are placed below the recursive calls - we want all recursive calls to open up on the stack before we start flipping pointers. Otherwise, since recursive calls can work with at most two adjacent nodes at the time (and not three nodes as we need in the iterative solution), we would break the list prematurely - i.e. we would flip the pointer but fail to advance through the list.

Finally, to call the method, we can use a trampoline, passing the head of the list as the initial parameter:

```
void reverseRecur() {
    reverseRecur1Helper(head);
}
```



3 Big-$\mathcal{O}$ Runtime Summary



Some Runtime Examples

Lists. In lists, all adds and removes involve just re-linking some pointers, all of which are just $\Theta(1)$ operations.

However, to add to a tail in $\Theta(1)$ time, we need to have access to the tail pointer. Otherwise, we would need to spend $\Theta(n)$ time to get to the tail first. If we need to remove at the tail in $\Theta(1)$, we also need to have a doubly-linked list, as we would need fast access to the preceding node to unlink the pointer.

Insertions or removals in the middle are $\Theta(n)$ as we would need to traverse $\frac{n}{2}$ elements. Finding an item is $\mathcal{O}(n)$, for example if we are looking for say the third item, that is technically $\Theta(1)$, so n is just an upper bound here.

1. Add/Remove an item from the *front* of a *singly* linked list: $\Theta(1)$
2. Add at item to the *back* of a *singly* linked list without a tail pointer: $\Theta(n)$
3. Add at item to the *back* of a *singly* linked list with a tail pointer: $\Theta(1)$
4. Remove an item from the *back* of a *singly* linked list without a tail pointer: $\Theta(n)$
5. Remove an item from the *back* of a *singly* linked list with a tail pointer: $\Theta(n)$
6. Remove an item from the *back* of a *doubly* linked list without a tail pointer: $\Theta(n)$
7. Remove an item from the *back* of a *doubly* linked list with a tail pointer: $\Theta(1)$
8. Add/remove anything in the *middle* for any linked-list: $\Theta(n)$
9. Add to the *front* of a *circular singly* linked list without a tail pointer: $\Theta(n)$
10. Locate something in a linked list: $\mathcal{O}(n)$

Addition to the front of a circular linked list without a tail pointer is $\Theta(n)$, as we would need access to the `next` pointer from the tail to link to the new head; however, to get that we would need to traverse the list!

Arrays. Arrays are stored in consecutive memory. So we would know exactly where say the fifth element is, without having to traverse a list.

However, any insertions or removals at the front or in the middle of an array would take $\Theta(n)$ time, as we would need to shuffle either all n elements over or $\frac{n}{2}$ elements over.

At the end, insertion is still $\Theta(1)$ on average. At the odd time when the array becomes full, the program would need to copy it to a new location in memory that could hold an array that is twice the size. However, if the program always just *doubles the array's size*, this happens rarely enough that insertion at the end is still $\Theta(1)$ on average (this can be shown mathematically).

1. Add an item to an array: $\Omega(1)$ to $\mathcal{O}(n)$.
   - Front or Middle: $\Theta(n)$
   - Back: $\Theta(1)$ on average
2. Remove an item from an array: $\Omega(1)$ to $\mathcal{O}(n)$.
   - Front or Middle: $\Theta(n)$
   - Back: $\Theta(1)$
3. Locate an item in the array $\Theta(1)$.

Sorted Arrays or Lists. To get either the smallest or largest element from a sorted array is $\Theta(1)$ since we know it would be either the first or the last element and we can just index that in constant time. For a linked list without a tail pointer, it would matter if the list is sorted in ascending or descending order and if we are looking for the smallest or largest element – it could be the case that we could be at the wrong end of the list (like in example 3 below)!

1. Find the smallest or largest item in a sorted array: $\Theta(1)$.
2. Find the smallest element into a singly linked list without a tail pointer that is sorted in ascending order: $\Theta(1)$.
3. Find the largest element into a singly linked list without a tail pointer that is sorted in ascending order: $\Theta(n)$.

There is actually a great summary table right here:
`https://en.wikipedia.org/wiki/Linked_list#Tradeoffs`