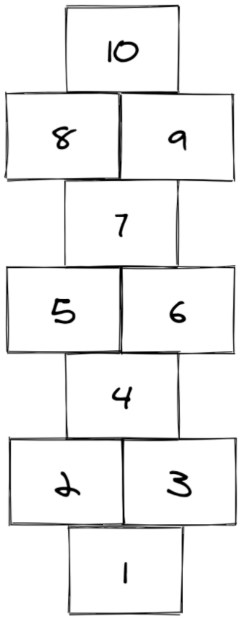
Data Structures - Linked Lists

Linked lists are not “lists”, they are an abstraction of categorizing a set of objects related to one another to understand them better. A linked list is a series of objects sitting in memory with a pointer and a value. These individual objects are linked by memory addresses (pointer) to form a linear data structure.

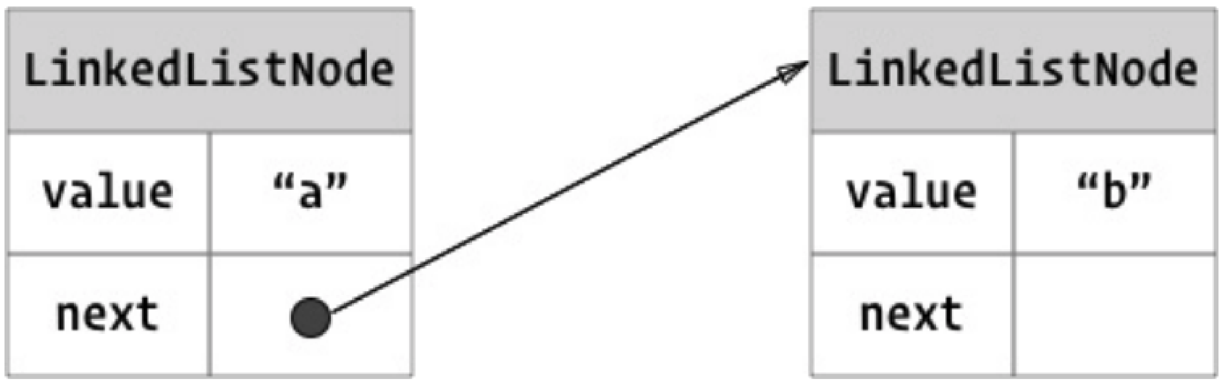


This linear data structure which consists of an ordered series of elements is usually not stored in contiguous memory like arrays have to be.

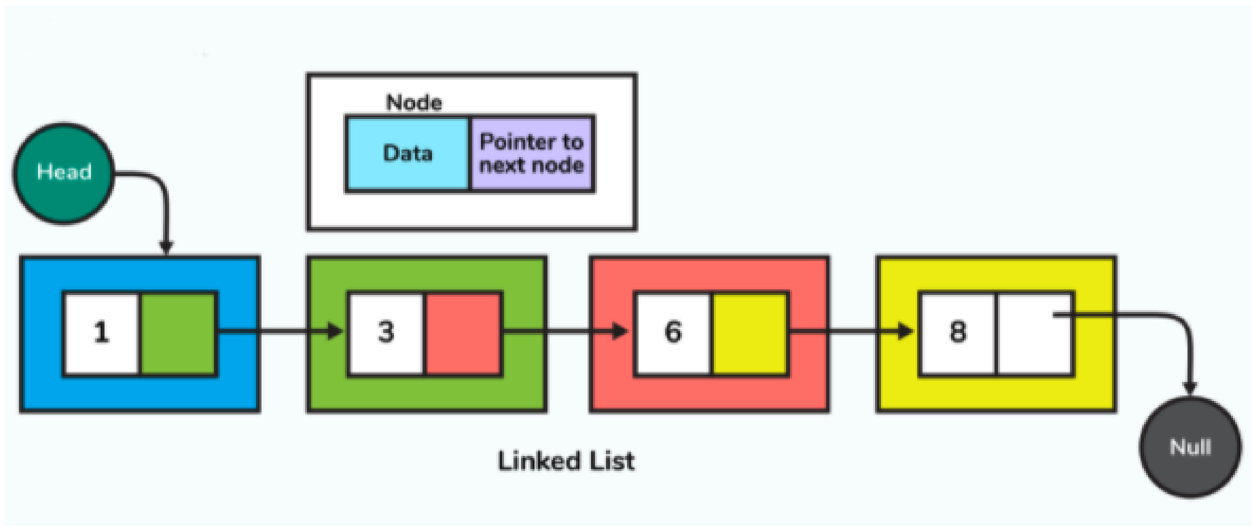
We can visualize a linear data structure like the chalk lines in a game of **hopscotch**. To get to the end of the list we have to go through all the items sequentially.

Elements in a linked list are generally referred to as nodes, although they are sometimes called elements or items, too. Each node contains a field that stores a value generally referred to as the link or **pointer**, which, links or points to the next node in the list.

The **head** of a linked list is its first node and, depending on the implementation, the **tail** is either the last element or the series of elements that follow the head.



Objects in JavaScript are dictionaries with key-value pairs. Keys are always strings and values can be a boolean or strings, numbers and objects. When we say our **next** property in our LinkedList class is a “pointer”, what we mean is that the value of **this.next** is a reference to another JavaScript object – another Node object.



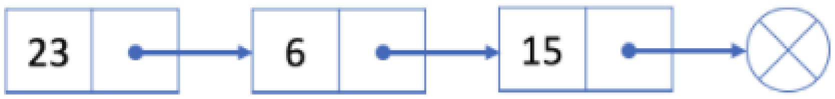
The starting point of the list is a reference to the first node, which is referred to as the **head**. The last node of the list is often referred to as its **tail**. The end of the list isn't a node but rather a node that points to null or an empty value.

Technically the node itself is not a “head” node, the Linked List itself has a local pointer variable (e.g. head pointer) which points to the first node of the list. If that pointer is NULL, then the list is empty.

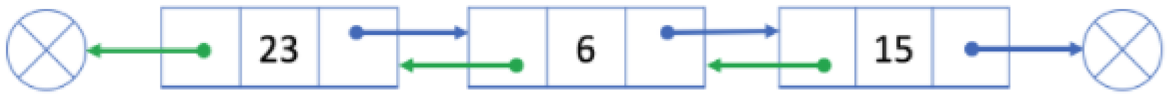
Note: The pointers themselves that are stored in each node that point to the next node can be also called **edges**. This is because a linked list is a type of tree called a **unary tree** (i.e. each node only points to one node). A binary tree has nodes that can point to two nodes (i.e. children).

There are two main types of linked lists:

- Singly linked lists: Each node contains exactly one reference to the next node.



- Doubly linked lists: Each node contains two references: a reference to the next node and a reference to the previous node.

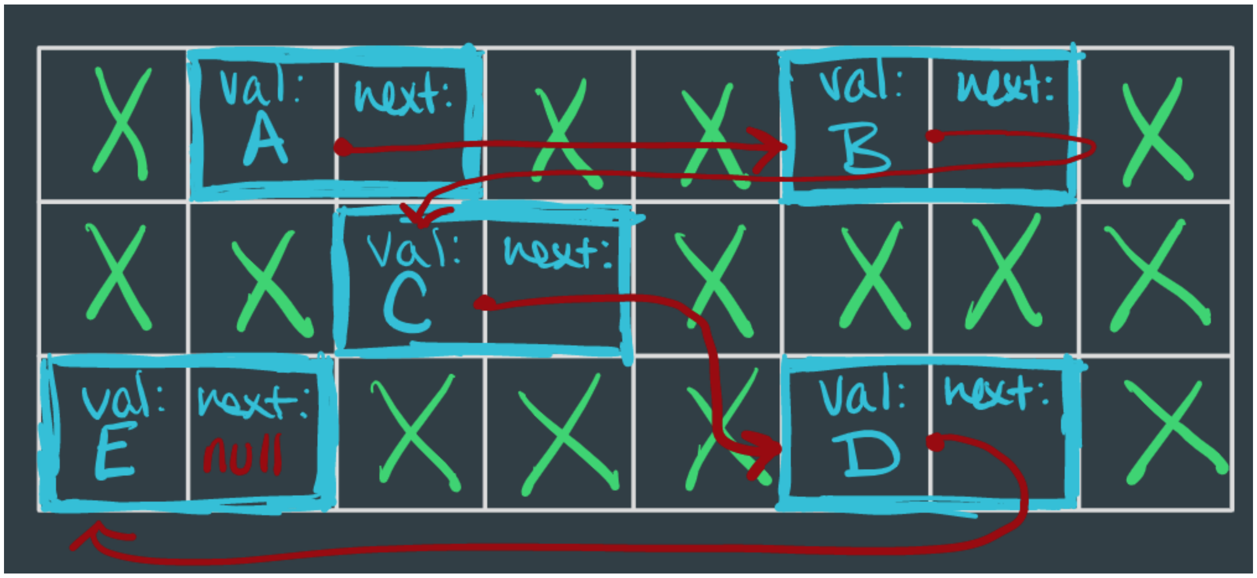


Doubly-linked lists are great for removing nodes because they provide access to the previous and the next nodes.

To remove a node from a singly-linked list, we have to iterate through the list, keeping track of the previous node so it’s a bit more complicated.

Note: a third linked list would be a curricular linked list

Linked Lists in Memory



Benefits of Linked Lists

- Nodes can be stored anywhere
- Adding nodes to the beginning of a linked list is quick - O(1)
- Implementation of linked lists can be flexible and versatile

Limits of Linked Lists

- To access a specific node we need to traverse - O(n)
- Nodes take up slightly more space than elements in an array
- Few languages come with built-in libraries to handle linked lists
- Only languages such as C++ and Java offer it natively.

Arrays vs Linked Lists

Arrays
- Indexed in order
- Insertion and deletion can be expensive because of shifting - O(n)
- Can be quickly accessed at a specific index - O(1)

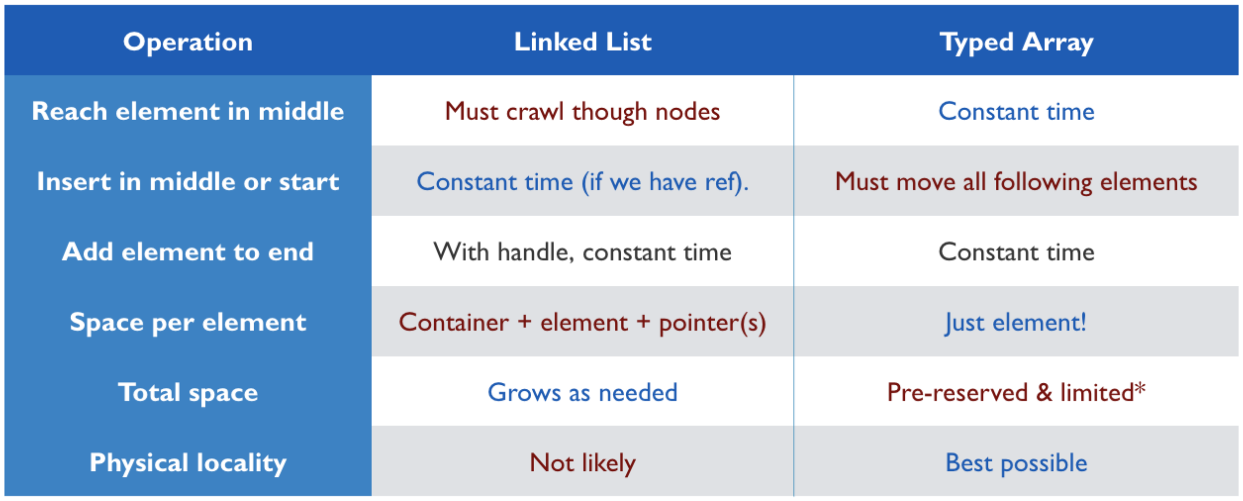
Linked Lists
- Do not have indexes
- Connected via nodes with a **next** pointer
- Random access is not allowed (aka access to a specific node)
- Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and deallocating memory.
- As the size of a linked list can grow or shrink based on the needs of the program, there is no memory wasted because it is allocated in runtime.

When to use Linked Lists

- Do not know the exact number of elements beforehand.
- There would be large number of add or remove operations.
- Less number of random access operations.
- When we want to insert items anywhere in the middle of the list, such as when implementing a priority queue, linked list is more suitable.

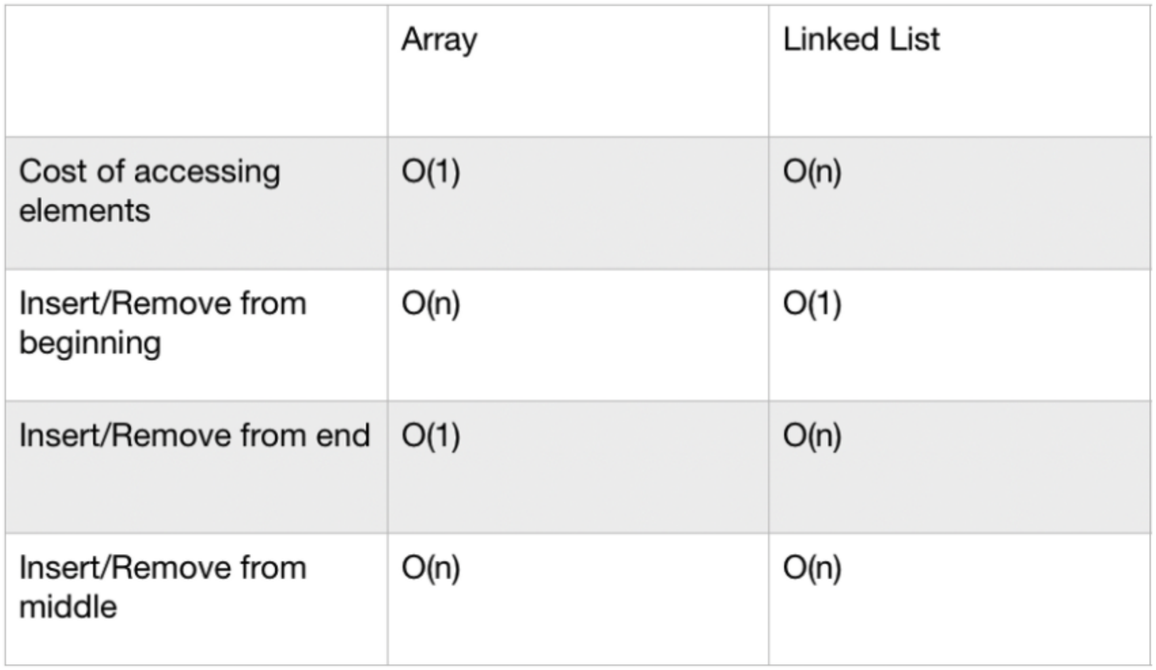
When to use Arrays

- Need to index or randomly access elements more frequently.
- Know the number of elements beforehand in allocating memory.
- Need speed while iterating over the elements in the sequence.
- Filled arrays use less memory than linked lists.
- Each element in the array indicates just the data whereas each linked list node represents the data as well as one or more pointers or references to the other elements in the linked list.



Operation	Linked List	Typed Array
Reach element in middle	Must crawl though nodes	Constant time
Insert in middle or start	Constant time (if we have ref).	Must move all following elements
Add element to end	With handle, constant time	Constant time
Space per element	Container + element + pointer(s)	Just element!
Total space	Grows as needed	Pre-reserved & limited*
Physical locality	Not likely	Best possible

Time Complexity - Array vs LinkedList



	Array	Linked List
Cost of accessing elements	O(1)	O(n)
Insert/Remove from beginning	O(n)	O(1)
Insert/Remove from end	O(1)	O(n)
Insert/Remove from middle	O(n)	O(n)

Creating a Linked List in JavaScript

To implement a linked list, you start with two **classes**:
Node and **SinglyLinkedList**

The constructor method for our Node class sets the initial values for this.val and this.next.

```
class Node {
 constructor(val) {
   this.val = val;
   this.next = null;
 }
}
```

The SinglyLinkedList class contains methods to be able to manipulate our collection of Node objects.

```
class SinglyLinkedList {
 constructor() {
   this.head = null;
   this.tail = null;
   this.length = 0;
 }
}
```

Node Class

The Node class has two components:

- Data (val)
- Pointer (next)

Data is the value you want to store in the Node. Think of it as a value at a specific index in an array. The data type can range from string to integer, to a custom class.

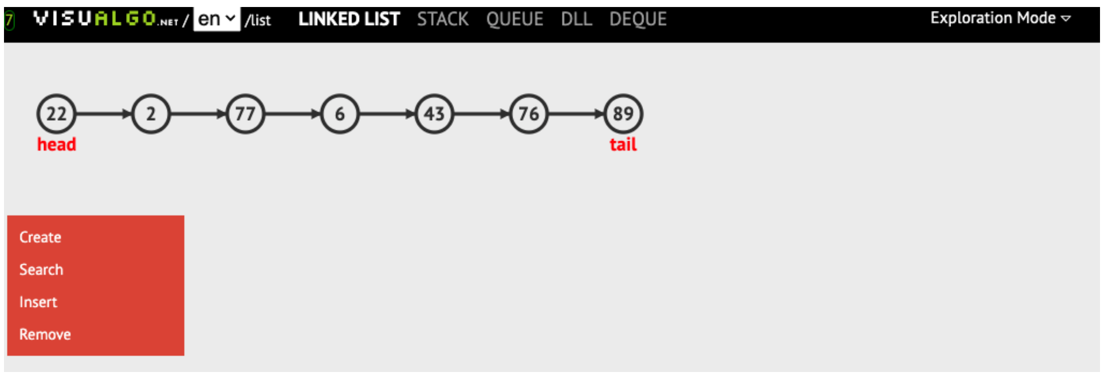
The **pointer** refers to the next Node in the list. It is essential for connectivity.

SinglyLinkedList Class

The Linked List itself is a collection of Node objects. To keep track of the list, we need a pointer to the first Node in the list.

This is where the principle of the **head** Node comes in. The head does not contain any data and only points to the beginning of the list. This means that, for any operations on the list, we need to traverse it from the head (the start of the list) to reach our desired list Node.

Use VisualAlgo to visually see how the different Linked List methods work.



append(value): O(1) - aka push

- Add a new node to the end of the Linked List
    - Accepts a value
    - Creates a new node using the value passed
    - If no head property on the list set head and tail to the newly created node; else set the next property on the tail to be the new node and set the tail property to be the newly created node.
    - Increment length by 1
    - Returns linked list

```
append(val) {
   let newNode = new Node(val);
   if (!this.head) {
     this.head = newNode;
     this.tail = this.head;
   } else {
     this.tail.next = newNode;
     this.tail = newNode;
   }
   this.length +=1;
   return this;
 }
```

deleteTail(): O(n) - aka pop

- Removing a node from the end of the Linked List.
    - **Edge case:** If there are no nodes in the list, return undefined
    - Loop list until reach the tail (but keep track of previous node)
    - Set the next property of the 2nd to last node to be null
    - Set the tail to be the 2nd to last node
    - Decrement the length of the list by 1
    - Return the value of the node removed
    - **Note:** Set head and tail to null if one node before popping

```
deleteTail() {
   if (!this.head) return undefined;
   let current = this.head;
   let newTail = current;
   while (current.next) {
     newTail = current;
     current = current.next;
   }
   this.tail = newTail;
   this.tail.next = null;
   this.length -=1;
   if (this.length === 0) {
     this.head = null;
     this.tail = null
   }
   return current;
 }
```

deleteHead(): O(1) - aka shift

- Removing node from the beginning of the Linked list.
    - **Edge case: I**f there are no nodes, return undefined
    - Store the current head property in a value
    - Set the head property to be the current head’s next property
    - Decrement the length by 1
    - Return the value of the node removed

```
deleteHead() {
   if (!this.head) return undefined
   let currentHead = this.head;
   this.head = currentHead.next;
   this.length -=1;
   return currentHead;
 }
```

prepend(value): O(1) - aka unshift

- Adding a new node to the beginning of a Linked List
    - Accepts a value
    - Create new node using value.
    - If no head property, set the head and tail to the newly created node. Otherwise set new node next property to the current head property on list
    - Set head property on the list to the newly added node.
    - Increment length by 1
    - Return the linked list

```
prepend(val) {
   let newNode = new Node(val);
   if (!this.head) {
      this.head = newNode;
      this.tail = this.head;
   }
   newNode.next = this.head;
   this.head = newNode;
   this.length +=1;
   return this;
 }
```

get(index): O(n)

- Retrieve a node by its position in the list
    - Accepts an index
    - If index is less than zero or greater than length, return null
    - Loop through list until you reach the index and return the node at a specific index

```
 get(index) {
   if (index < 0 || index >= this.length) return null;
   let counter = 0;
   let current = this.head;
   while (counter !== index) {
     current = current.next;
     counter++;
   }
   return current;
 }
}
```

set(index, value): O(n)

- Changes value of a node based on its position in the list
    - Accepts a index and a value
    - Use get method to find specific node
    - If node is not found, return false
    - If node is found, set the value of that node to the value passed and return true

```
set(index, val) {
   let foundNode = this.get(index);
   if (foundNode) {
     foundNode.val = val;
     return true;
   }
   return false;
 }
}
```

insert(index, value): O(n)

- Adding a node to list a specific position
    - Edge cases:
        - If index is less than zero or greater than length return false
        - If index is same as length, push new node to end of list
        - If index is 0, unshift new node to start of list
    - Otherwise, using get method, access node at the index -1 (right before) and set next property on that node to be new node
    - Increment length
    - Return true

```
insert(index, val) {
   if (index < 0 || index > this.length) return false;
   if (index === this.length) {
       this.append(val);
       return true;
   }
   if (index === 0) {
     this.prepend(val);
     return true;
   }
   let newNode = new Node(val);
   let prev = this.get(index - 1);
   let temp = prev.next;
   prev.next = newNode;
   newNode.next = temp;
   this.length +=1;
   return true;
```

delete(index): O(n)

- Removes a node from a list at a specific position
    - **Edge cases:**
        - If index is less than zero or greater than equal than length return undefined
        - If index is the same as the length -1, use deleteTail method
        - If the index is 0, use deleteHead method
    - Otherwise, use get method, access the node as index -1 and set the next property on the node to be the next of the next node
    - Decrement the length

```
delete (index) {
   if (index < 0 || index >= this.length) return
       undefined;
   if (index === 0) return this.deleteHead();
   if (index === this.length - 1) return
       this.deleteTail();
   let prevNode = this.get(index - 1);
   let removed = prevNode.next;
   prevNode.next = removed.next;
   this.length -=1
   return removed;
}
```

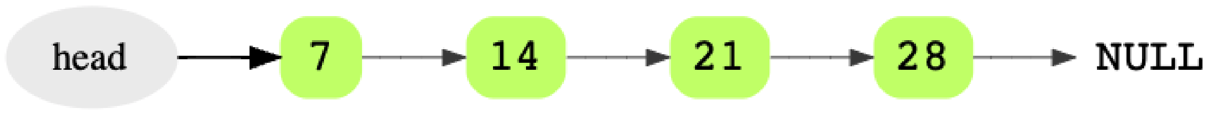
Linked List Problems

Problem #1: Reverse a Singly Linked List in place

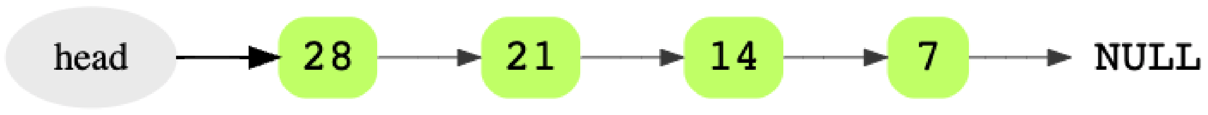
We’re given the pointer/reference to the head of a singly linked list, reverse it and return the pointer/reference to the head of the reversed linked list.

Time complexity: O(n) - linear complexity for reverse link list in single pass
Space complexity: O(1) - no extra memory is required for iterative solution

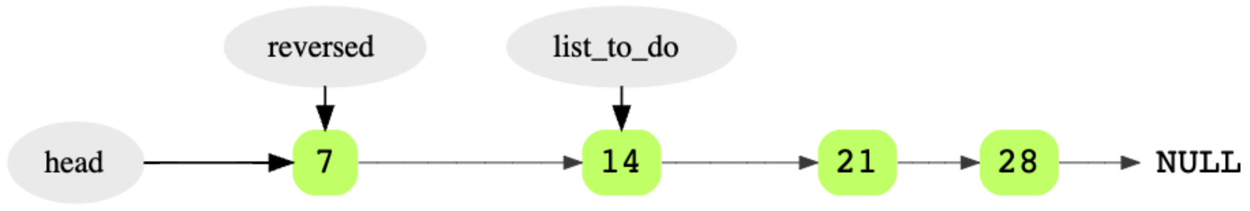
Input linked list:



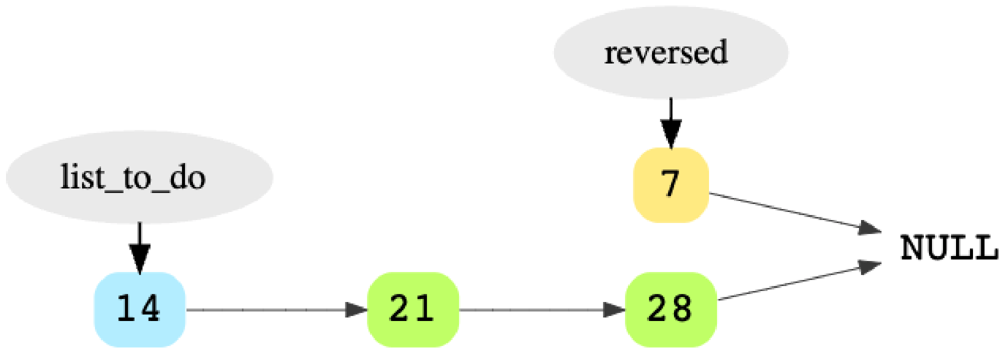
Output:



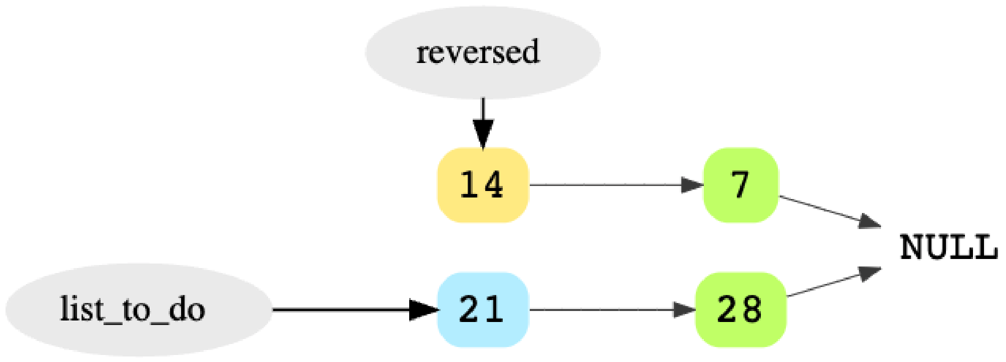
Step 1:



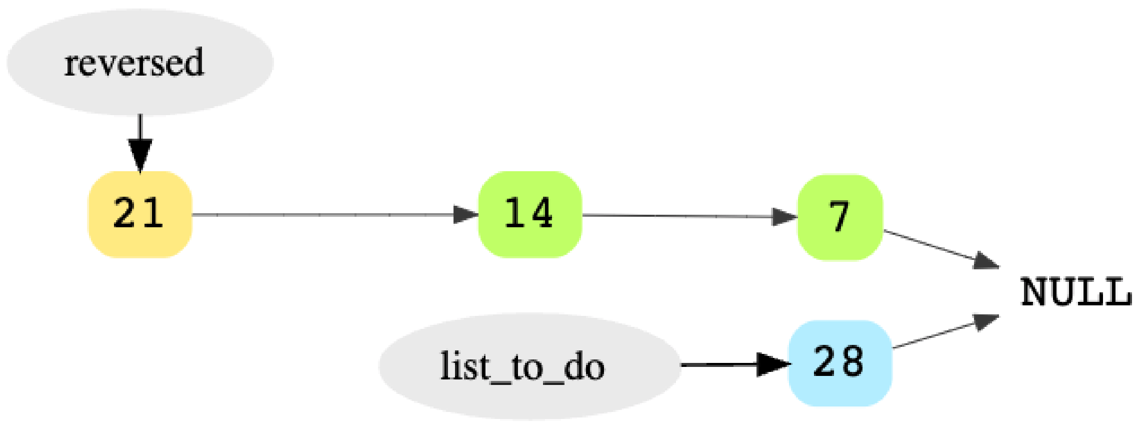
Step 2:



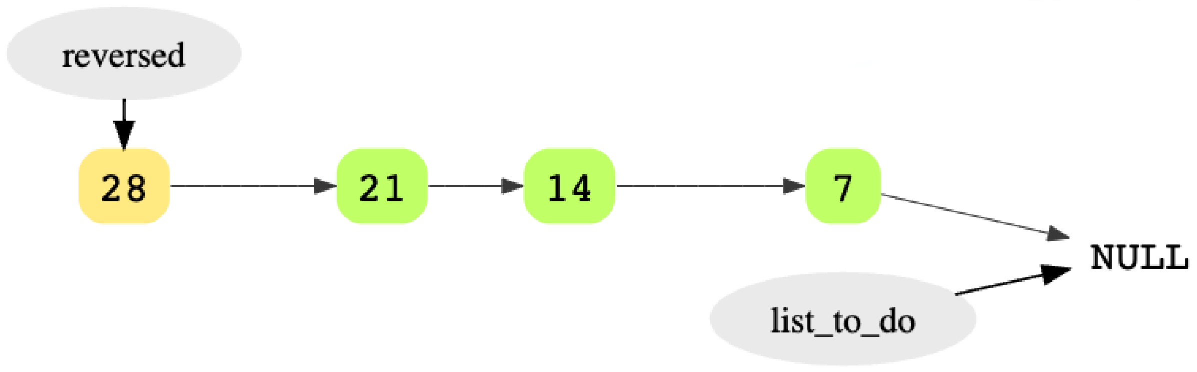
Step 3:



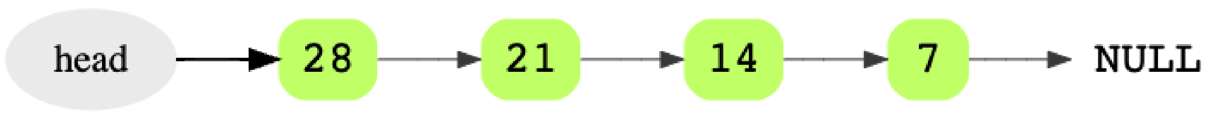
Step 4:



Step 5:



Step 6:



- Swap the head and tail
- Create a reference to the head property
- Loop through the list
    - Set next to be the next property
    - Set the next property on the node to be whatever prev will be
    - Set prev to be the value of the node variable
    - Set the node variable to be the value of the next variable

```
reverse() {
     let node = this.head;
     this.head = this.tail;
     this.tail = node;
     let next;
     let prev = null;

     for(let i = 0; i < this.length; i++){
       next = node.next;
       node.next = prev;
       prev = node;
       node = next;
     }
     return this;
   }
}
```

Problem #2: Write a function **linkedListValues** that takes in the head of a Singly Linked List as an argument and returns an array containing all the values of the nodes in the list.

```
const a = new Node("a");
const b = new Node("b");
const c = new Node("c");
const d = new Node("d");

a.next = b;
b.next = c;
c.next = d;

linkedListValues(a); // -> [ 'a', 'b', 'c', 'd' ];
```

iterative approach: time - O(n), space - O(n)

```
const linkedListValues = (head) => {
  const values = [];
  let current = head;
  while (current !== null) {
    values.push(current.val);
    current = current.next;
  }
  return values;
};
```

recursive approach: time - O(n), space - O(n)

```
const linkedListValues = (head) => {
  const values = [];
  _linkedListValues(head, values);
  return values;
};

const _linkedListValues = (head, values) => {
  if (head === null) return;
  values.push(head.val);
  _linkedListValues(head.next, values);
};
```

Problem #3: Remove Duplicates - time O(n), space O(1)

You are given the head of a Singly Linked List whose nodes are in **sorted** order with respect to their values. Write a function **removeDuplicates** that returns a modified version of the Linked List that doesn't contain any nodes with duplicate values. The Linked List should be modified in place and the modified Linked List should still have its nodes sorted with respect to their values.

Sample input:
linkedList = 1->1->3->4->4->4->5->6->->6

Sample output:
1->3->4->5->6

Note: The nested while loop is still O(n) because the outer loop is just checking for the end node (tail) so it would be O(n + n) which would be resolved to O(n). As the worst and best case would still involve iterating through each node of the Linked List.

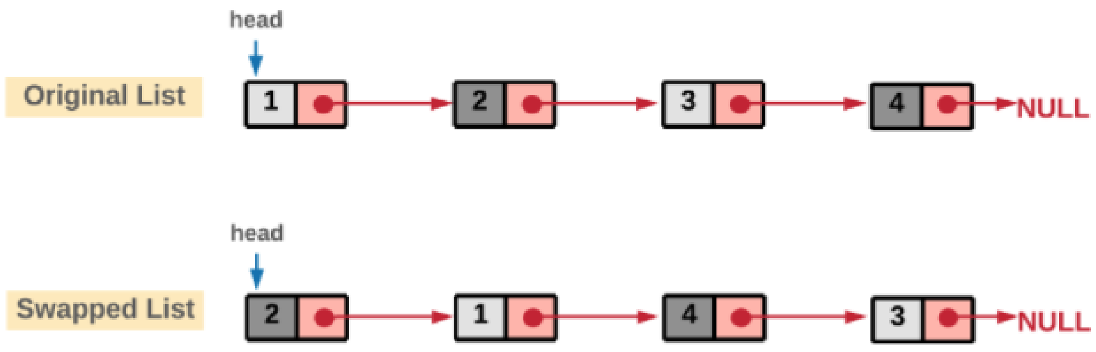
```
function removeDuplicates(head) {
    let currentNode = head;
    while(currentNode !== null) {
        let nextDistinctNode = currentNode.next;
        while(nextDistinctNode !== null &&
        nextDistinctNode.value === currentNode.value)
{
             nextDistinctnode = nextDistinctNode.next;
        }
        currentNode.next = nextDistinctNode;
        currentNode = nextDistinctNode;
    }
}
```

Problem #4: Swap Nodes in Pairs: Time - O(n), Space - O(n)

Write a recursive algorithm that swaps every two nodes in a linked list. This is often called a pairwise swap.

Sample input:
linkedList = 1->2->3->4

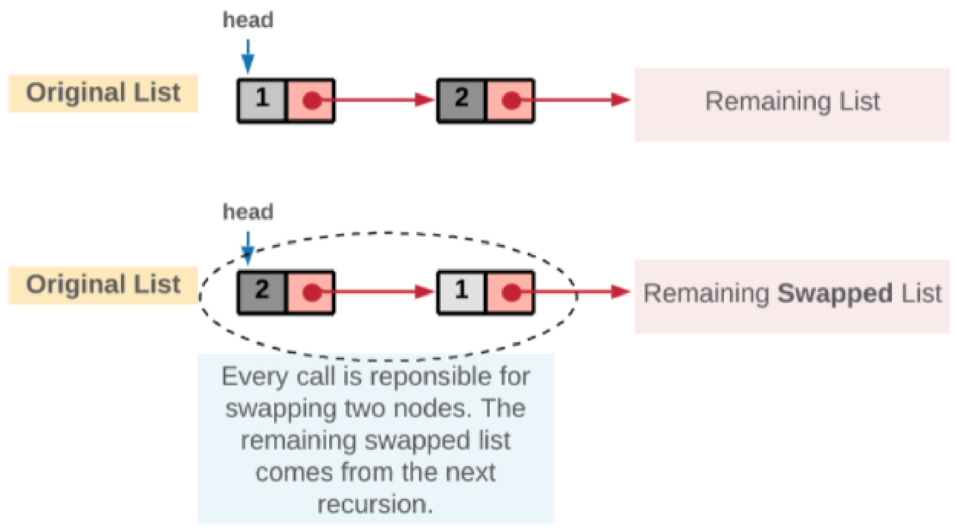
Sample output:
2->1->4->-3



```
function swapEveryTwo(head) {
  if (!head || !head.next) {
    return head;
  }

  let second = head.next;
  let third = second.next;
  second.next = head;

  head.next = swapEveryTwo(third);
  return second;
}
```


head
Original List
1
2
Remaining List
head
Original List
2
1
Remaining **Swapped** List
Every call is reponsible for swapping two nodes. The remaining swapped list comes from the next recursion.

In every function call we take out two nodes which would be swapped and the remaining nodes are passed to the next recursive call.

The reason we are adopting a recursive approach here is because a sub-list of the original list would still be a linked list and hence, it would adapt to our recursive strategy. Assuming the recursion would return the swapped remaining list of nodes, we just swap the current two nodes and attach the remaining list we get from recursion to these two swapped pairs.

Quiz Questions:

1. What is the general declaration of a node in a linked list?

```
 var Node = { data 1: next: null};
```

2. What does the code line this.head normally refer to?

```
The head node of a linked list
```

3. When adding a node to a linked list there is normally a special case condition that is identified by the following code snippet:

```
if (this._head === null) { this._head = node; }
```

What does this code snippet actually do?

```
This is a special case condition if the node to be
inserted is the first node in the linked list. If
there is no head node, the linked list is empty
```

4. Why do we track the previous node while removing?

```
To connect previous with the linked list nodes that
come after the removed node
```

5. What should the previous.next point to if we just removed the last node?

```
Null
```