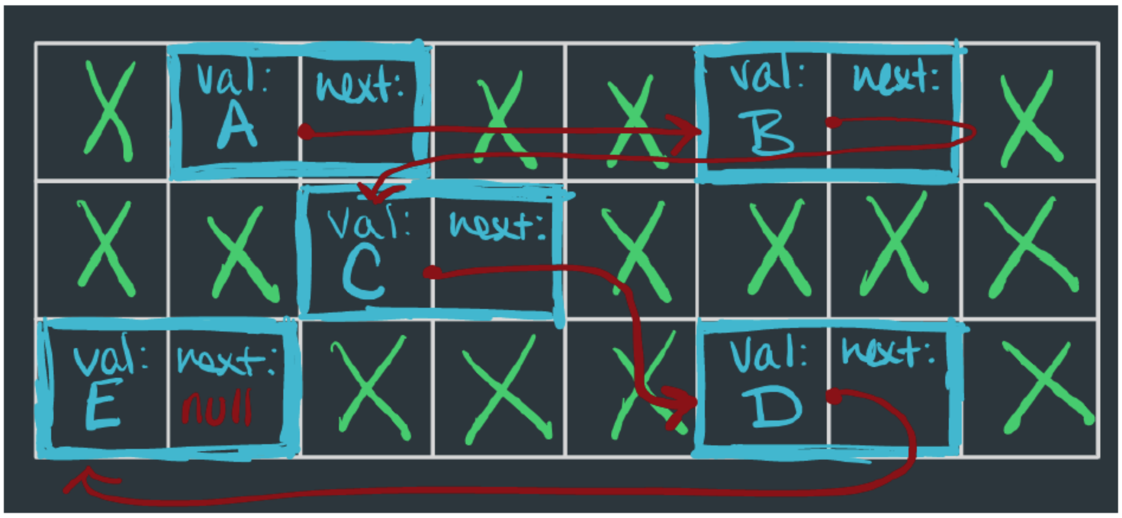
Linked Lists

A linked list is an ordered, linear data structure that's similar to an array. However, unlike in an array, elements aren't stored at a particular index; instead, they are connected through a chain of references.

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

Unlike arrays, which store data contiguously in memory, linked lists can easily insert or remove nodes from the list without reorganizing all of the data.

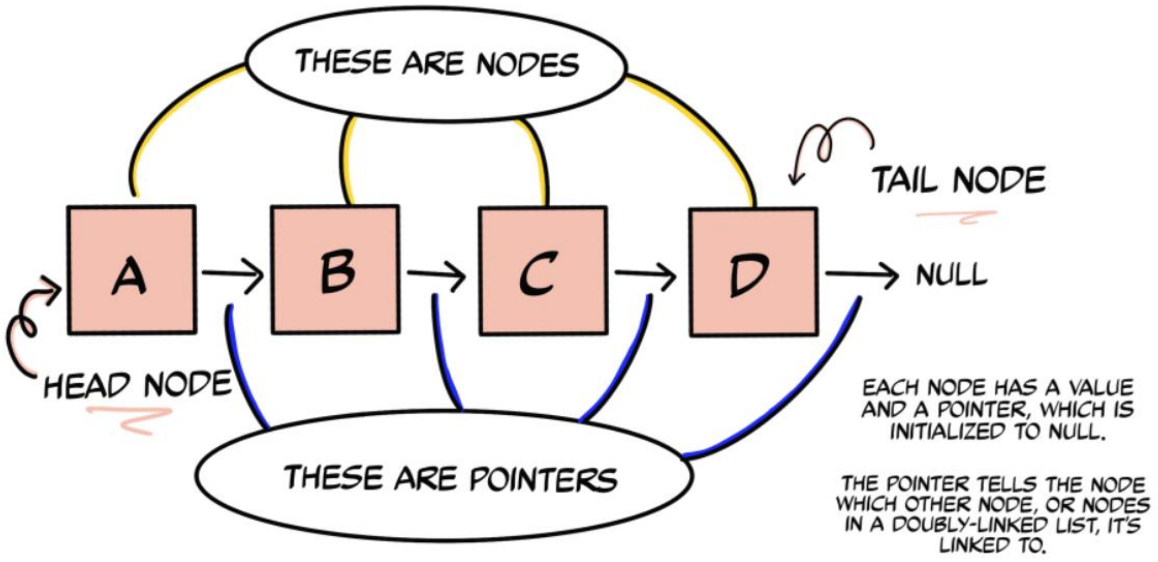


The starting point of the list is a reference to the first node, which is referred to as the **head**. The last node of the list is often referred to as its **tail**. The end of the list isn't a node but rather a node that points to **null** or an empty value.

Elements in a linked list are generally referred to as **nodes**, although they are sometimes called elements or items, too.

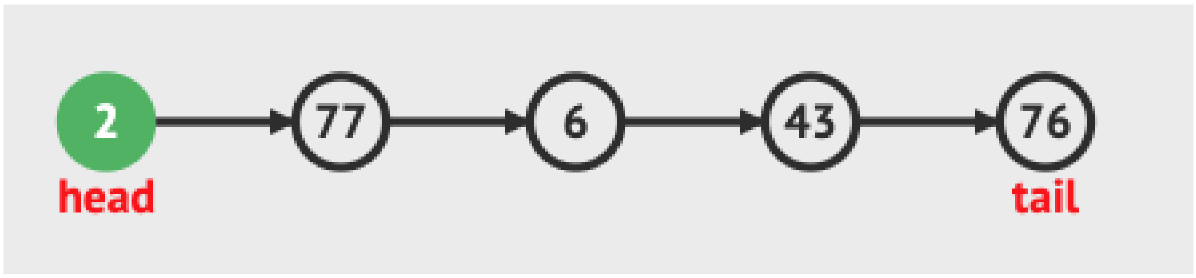
When we say our next property in our LinkedList class is a "pointer", what we mean is that the value of **this.next** is a reference to another JavaScript object – another Node object.

The 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).

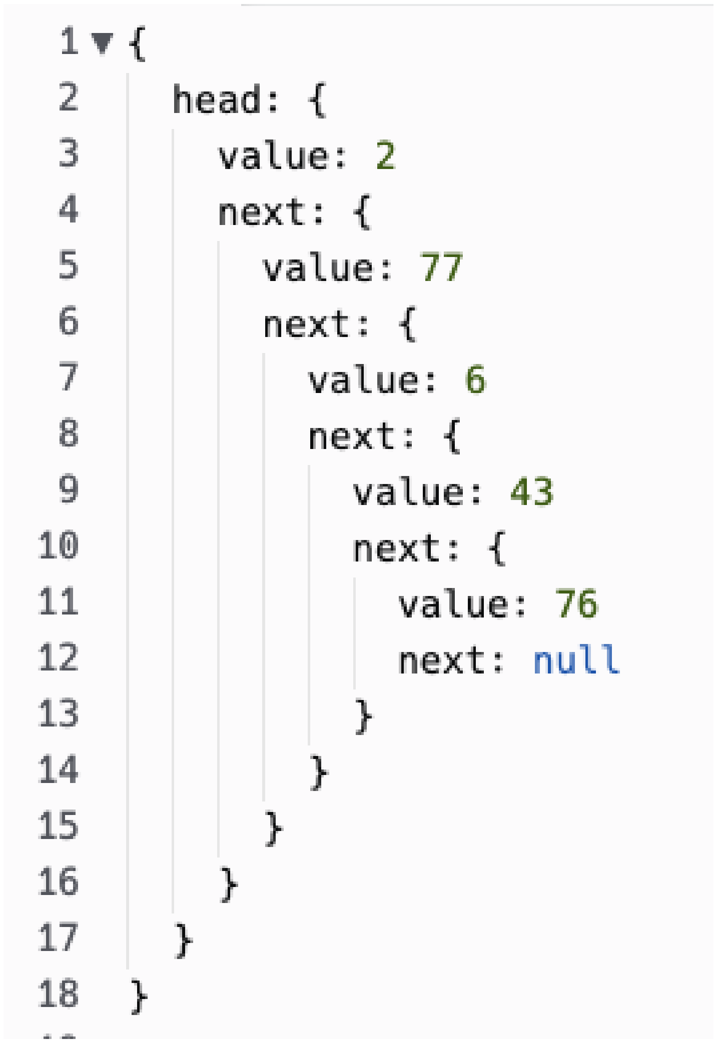

THESE ARE NODES
TAIL NODE
A
B
C
D
NULL
HEAD NODE
THESE ARE POINTERS
EACH NODE HAS A VALUE AND A POINTER, WHICH IS INITIALIZED TO NULL.
THE POINTER TELLS THE NODE WHICH OTHER NODE, OR NODES IN A DOUBLY-LINKED LIST, IT'S LINKED TO.

Additionally, linked lists and arrays perform tasks related to accessing, adding, and removing data at different speeds. Search operations on a linked list are slow because data can't

be accessed randomly; nodes in a linked list are accessed sequentially, starting from the first node.



The linked list depicted above can be written in JSON as:

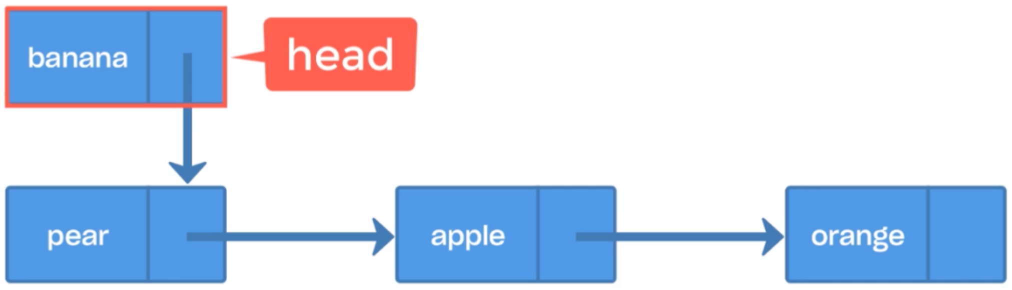


```
{
  head: {
    value: 2
    next: {
      value: 77
      next: {
        value: 6
        next: {
          value: 43
          next: {
            value: 76
            next: null
          }
        }
      }
    }
  }
}
```

Inserting new data into the list varies, depending on where you wish to insert the data. Inserting data at the head of the list is the most straightforward case.

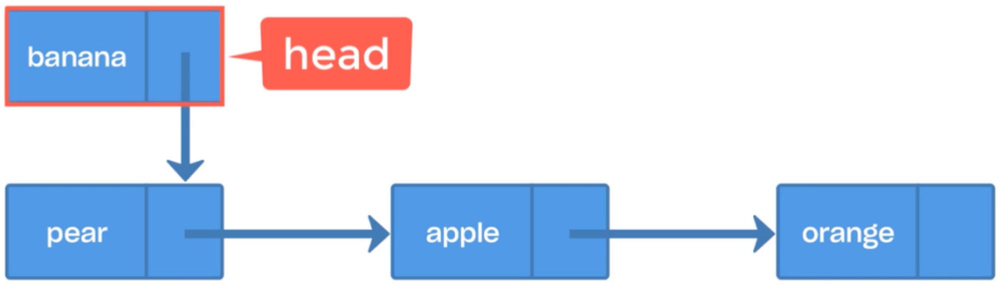
Insert at head:

- Create a new node.
- Set new node next pointer = head
- Set the head pointer = new node



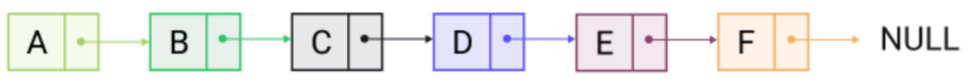
Insert after:

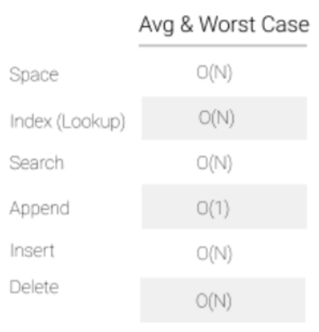
- Find the previous node
- Create new node
- Set new node next pointer = previous node next pointer
- Set previous node next pointer = new node



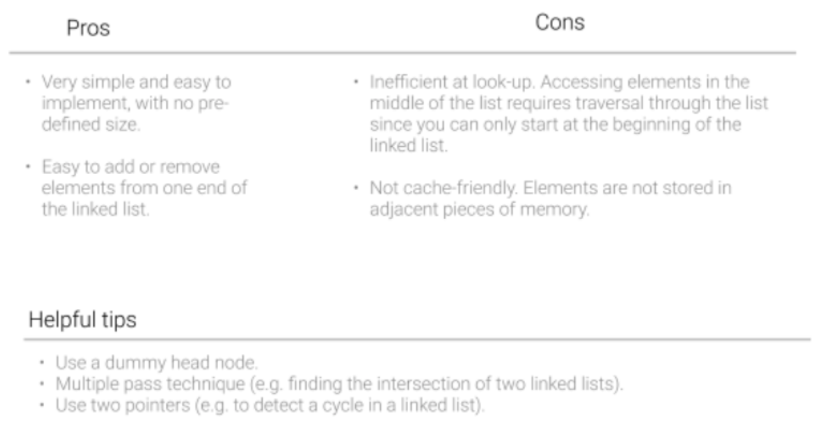
Types of Linked Lists

- **Singly linked lists** have each node containing one reference to the next node.





	Avg & Worst Case
Space	O(N)
Index (Lookup)	O(N)
Search	O(N)
Append	O(1)
Insert	O(N)
Delete	O(N)

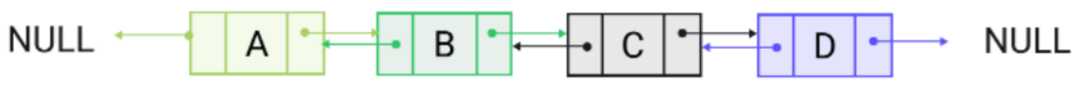


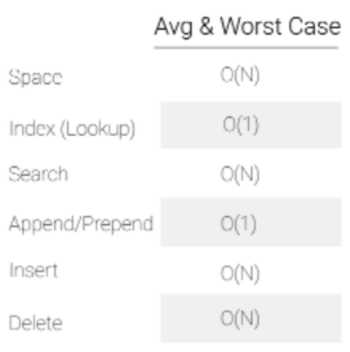
Pros	Cons
• Very simple and easy to implement, with no pre-defined size. • Easy to add or remove elements from one end of the linked list.	• Inefficient at look-up. Accessing elements in the middle of the list requires traversal through the list since you can only start at the beginning of the linked list. • Not cache-friendly. Elements are not stored in adjacent pieces of memory.

Helpful tips

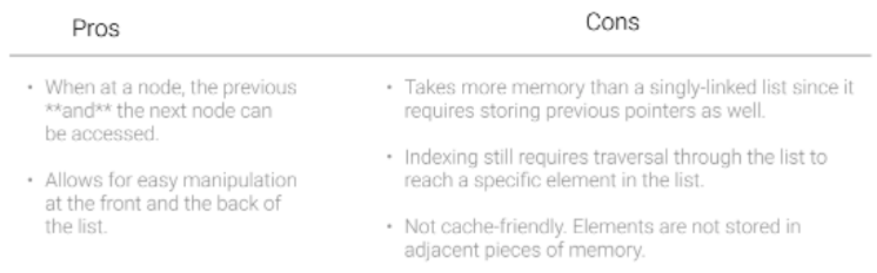
- Use a dummy head node.
- Multiple pass technique (e.g. finding the intersection of two linked lists).
- Use two pointers (e.g. to detect a cycle in a linked list).

- **Doubly linked list** has each node containing one reference to the next and previous node.





	Avg & Worst Case
Space	O(N)
Index (Lookup)	O(1)
Search	O(N)
Append/Prepend	O(1)
Insert	O(N)
Delete	O(N)



Pros	Cons
• When at a node, the previous **and** the next node can be accessed. • Allows for easy manipulation at the front and the back of the list.	• Takes more memory than a singly-linked list since it requires storing previous pointers as well. • Indexing still requires traversal through the list to reach a specific element in the list. • Not cache-friendly. Elements are not stored in adjacent pieces of memory.

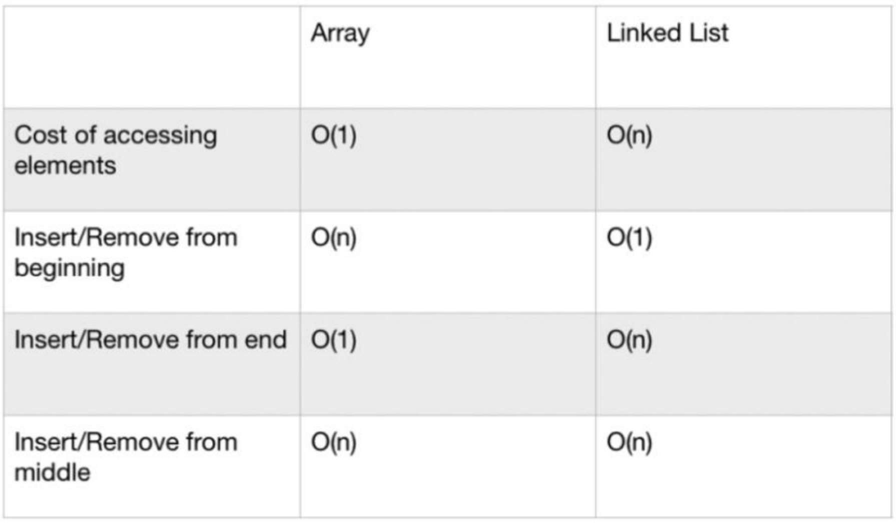
Array vs Linked List

<u>Array</u>

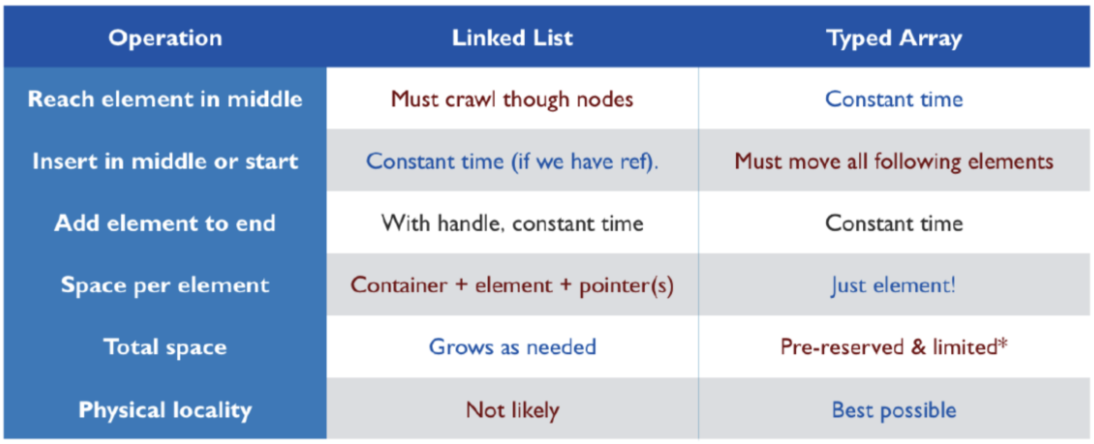
- Indexed in order
- Insertion and deletion is expensive => shifting - O(n)
- Random access at a specific index - O(1)

<u>Linked List</u>

- Do not have indexes
- Connected via nodes with a next pointer
- Random access is not allowed
- Dynamic data structure so it has less memory wasted by allocating and deallocating memory when needed.



	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)



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

Constructing a linked list (single)

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

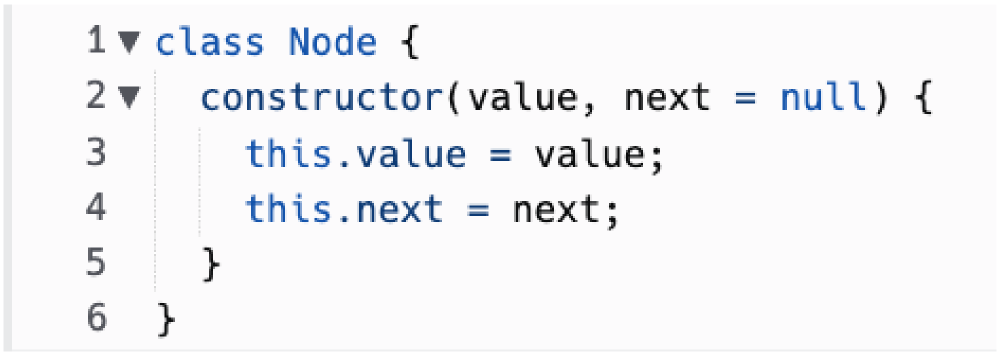
Node Class

The Node class has two components:

- Data (value)
- 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.



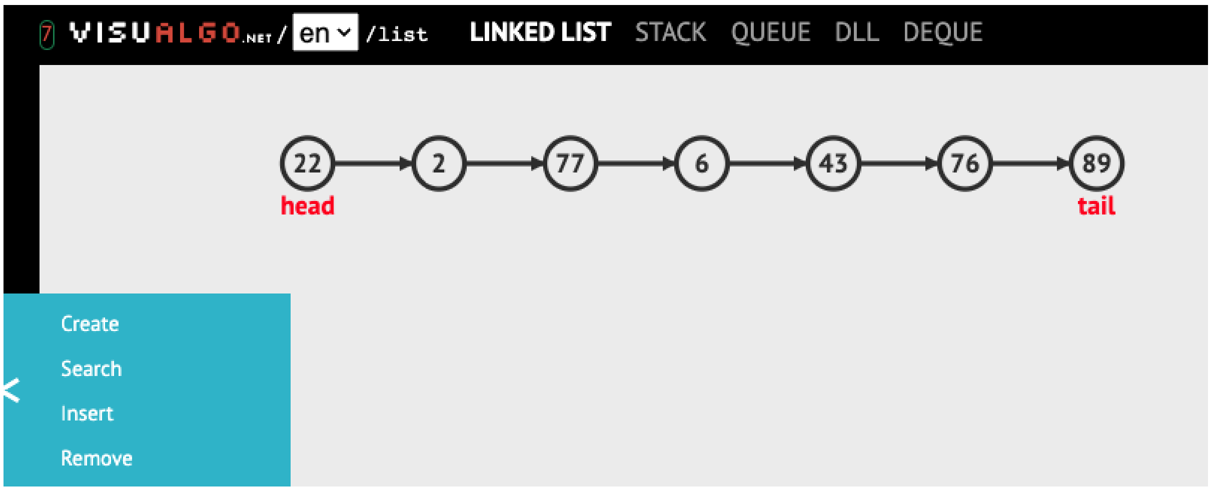
```
class Node {
  constructor(value, next = null) {
    this.value = value;
    this.next = next;
  }
}
```

SinglyLinkedList class

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

This is where the principle of the **head Node** comes in. For any operations on the list, we need to traverse it from the **head** (start of the list) to reach our desired list Node.

You can use VisualAlgo to visually see how the different Linked List methods work.

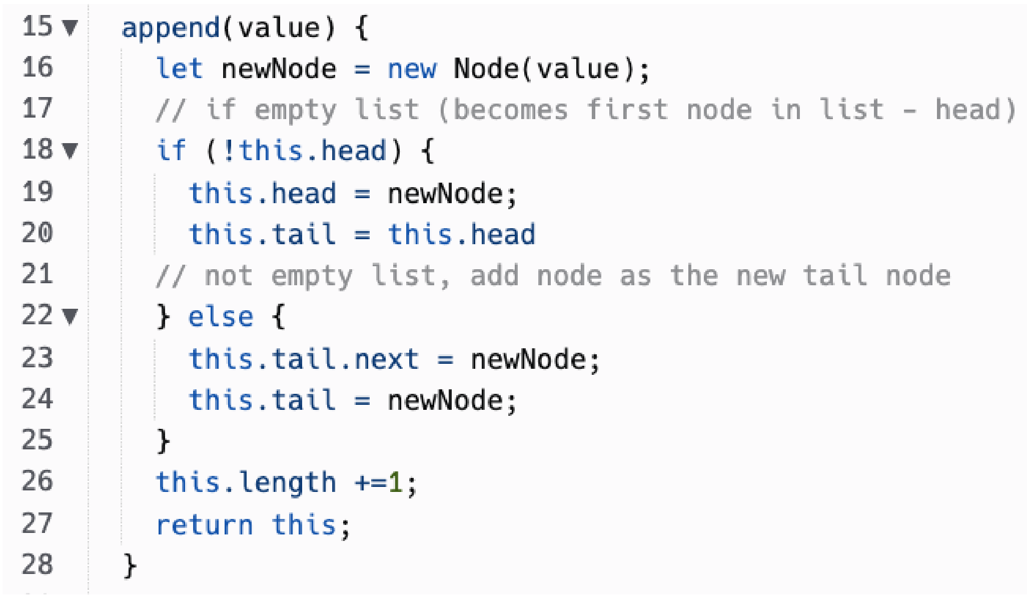


```
class SinglyLinkedList {
  constructor(head = null) {
    this.head = head;
    this.tail = null;
    this.length = 0;
  }
}
```

append(value): O(1)

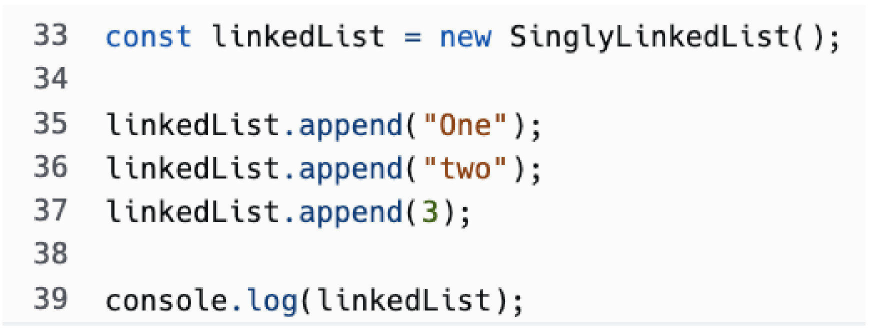
Adds a new node to the end of the Linked List

- Creates a new node using the value passed
- If empty list => node becomes head node
  If not empty list => node becomes new tail node
- Increment length property by 1

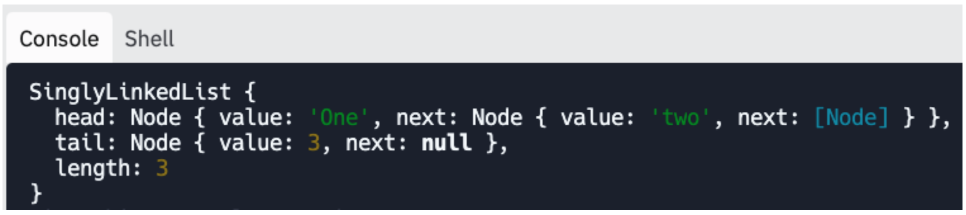


```javascript
append(value) {
  let newNode = new Node(value);
  // if empty list (becomes first node in list - head)
  if (!this.head) {
    this.head = newNode;
    this.tail = this.head
  // not empty list, add node as the new tail node
  } else {
    this.tail.next = newNode;
    this.tail = newNode;
  }
  this.length +=1;
  return this;
}
```

If we try out implementing what we have so far:



```javascript
const linkedList = new SinglyLinkedList();

linkedList.append("One");
linkedList.append("two");
linkedList.append(3);

console.log(linkedList);
```



Console Shell

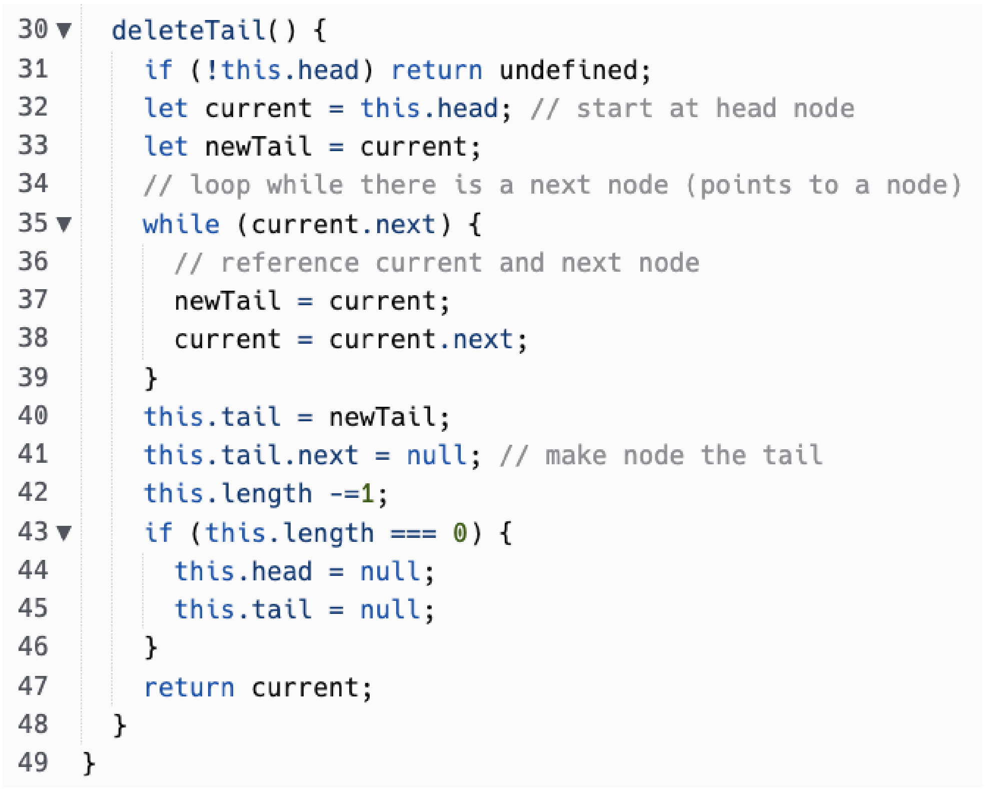
```
SinglyLinkedList {
  head: Node { value: 'One', next: Node { value: 'two', next: [Node] } },
  tail: Node { value: 3, next: null },
  length: 3
}
```

deleteTail(): O(n)

Removes a node from the end of the Linked List.

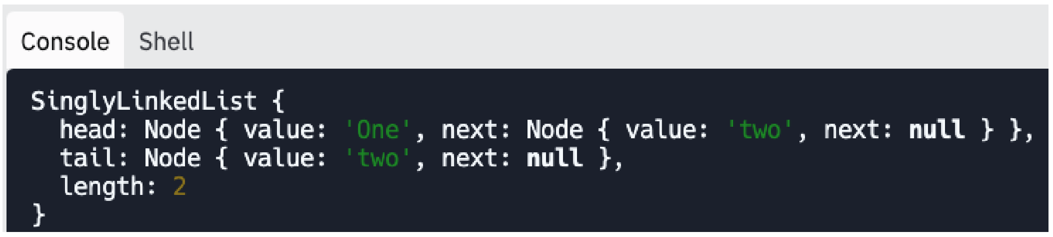
- **Edge case:** No nodes in list, return undefined
- Loop until reaching the tail (next property will be **null**)
    - Set temporary new tail to current node.
    - Reference next node (may or may not be null)
- Decrement length
- Return the value of the node removed
- **Edge case:** One node in list, set head and tail to null

Note: You are not “deleting” a node, just removing a reference to a node from a node’s next property. Garbage collection does the “removal” eventually.



```
  deleteTail() {
    if (!this.head) return undefined;
    let current = this.head; // start at head node
    let newTail = current;
    // loop while there is a next node (points to a node)
    while (current.next) {
      // reference current and next node
      newTail = current;
      current = current.next;
    }
    this.tail = newTail;
    this.tail.next = null; // make node the tail
    this.length -=1;
    if (this.length === 0) {
      this.head = null;
      this.tail = null;
    }
    return current;
  }
}
```

```javascript
const linkedList = new SinglyLinkedList();

linkedList.append("One");
linkedList.append("two");
linkedList.append(3);

linkedList.deleteTail();

console.log(linkedList);
```

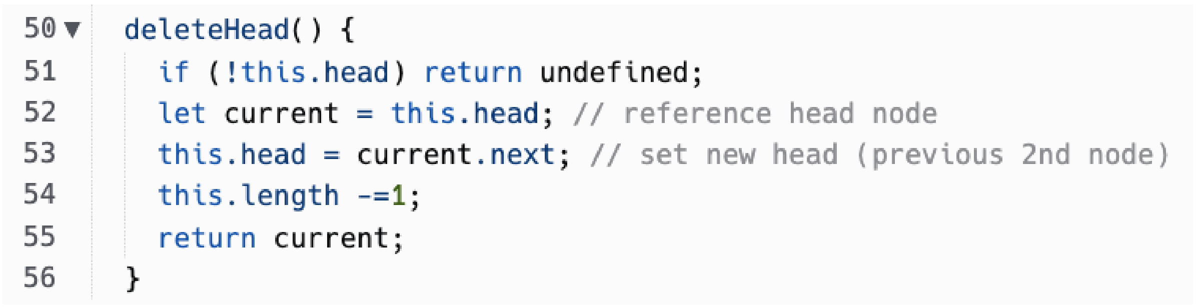


```
SinglyLinkedList {
  head: Node { value: 'One', next: Node { value: 'two', next: null } },
  tail: Node { value: 'two', next: null },
  length: 2
}
```

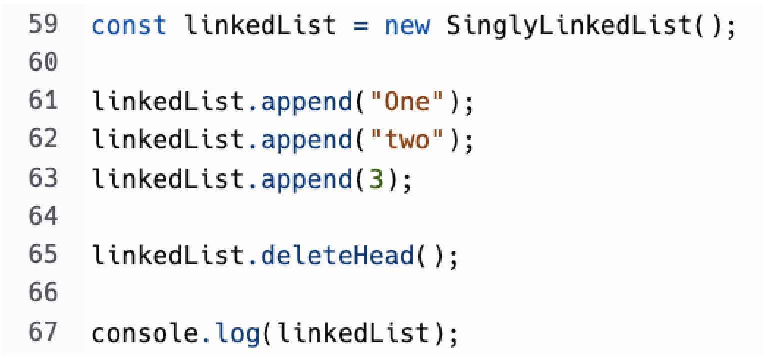
deleteHead(): O(1)

Remove node from beginning of Linked List

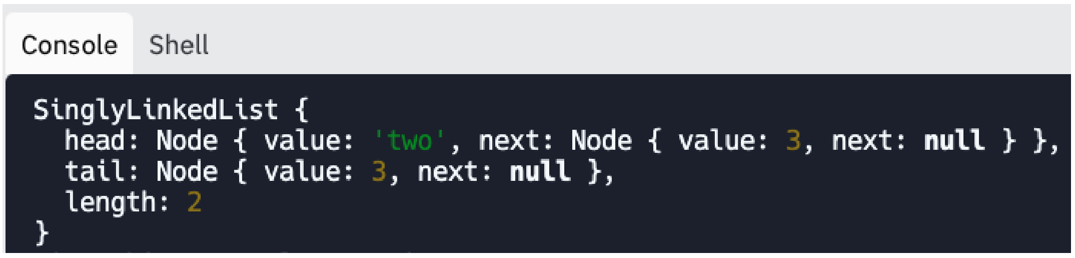
- **Edge case:** No nodes in list, return undefined
- Store current head property in a value
- Set the **head** property to be the current head's **next** property.
- Decrement length
- Return node removed



```javascript
deleteHead() {
  if (!this.head) return undefined;
  let current = this.head; // reference head node
  this.head = current.next; // set new head (previous 2nd node)
  this.length -=1;
  return current;
}
```



```javascript
const linkedList = new SinglyLinkedList();

linkedList.append("One");
linkedList.append("two");
linkedList.append(3);

linkedList.deleteHead();

console.log(linkedList);
```

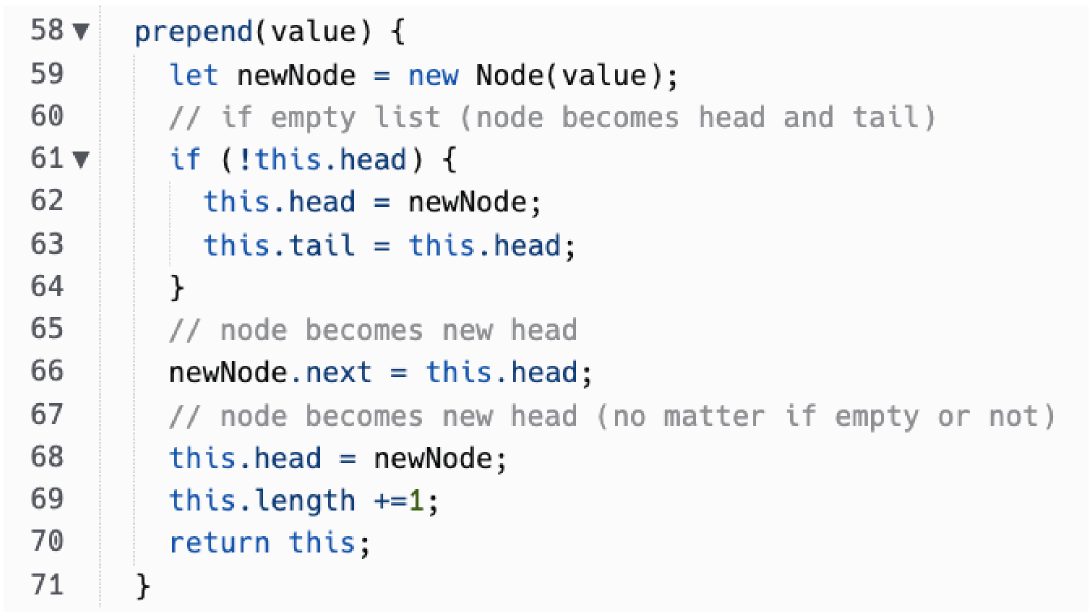


```
SinglyLinkedList {
  head: Node { value: 'two', next: Node { value: 3, next: null } },
  tail: Node { value: 3, next: null },
  length: 2
}
```

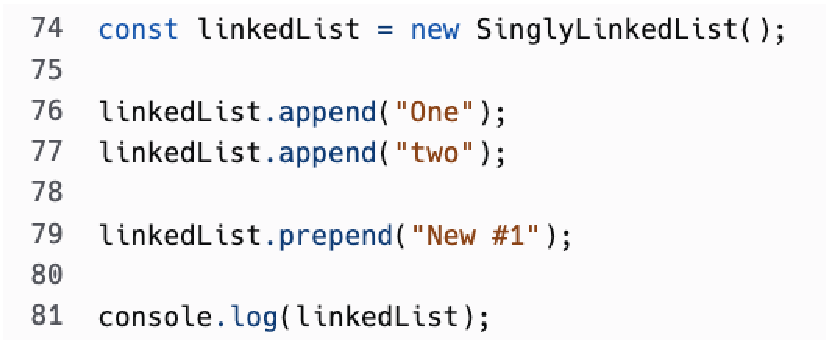
prepend(value): O(1)

Add new node to the beginning of Linked List

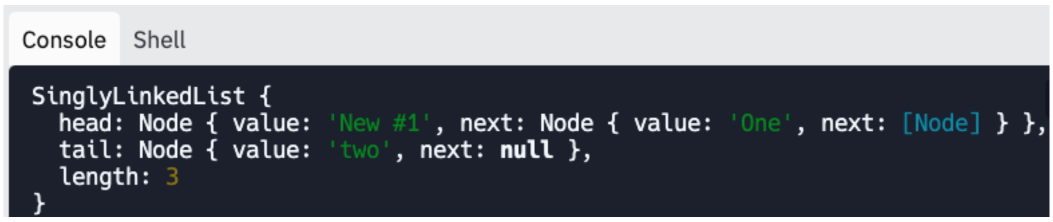
- Create new node
- **Edge case:** No nodes in list, node becomes head and tail.
- Set **next** property of new node to the current head.
- Set head to the new node.
- Increment length



```
prepend(value) {
  let newNode = new Node(value);
  // if empty list (node becomes head and tail)
  if (!this.head) {
    this.head = newNode;
    this.tail = this.head;
  }
  // node becomes new head
  newNode.next = this.head;
  // node becomes new head (no matter if empty or not)
  this.head = newNode;
  this.length +=1;
  return this;
}
```



```
const linkedList = new SinglyLinkedList();

linkedList.append("One");
linkedList.append("two");

linkedList.prepend("New #1");

console.log(linkedList);
```

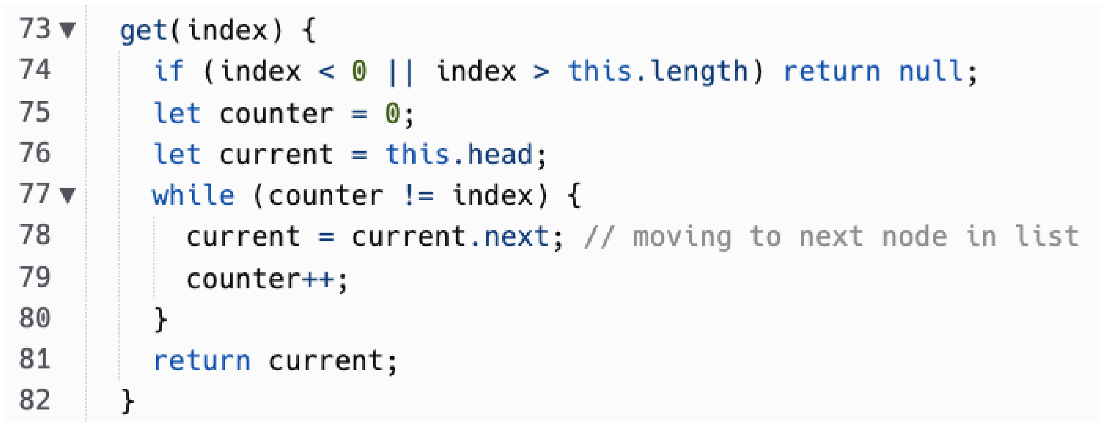

Console Shell

```
SinglyLinkedList {
  head: Node { value: 'New #1', next: Node { value: 'One', next: [Node] } },
  tail: Node { value: 'two', next: null },
  length: 3
}
```

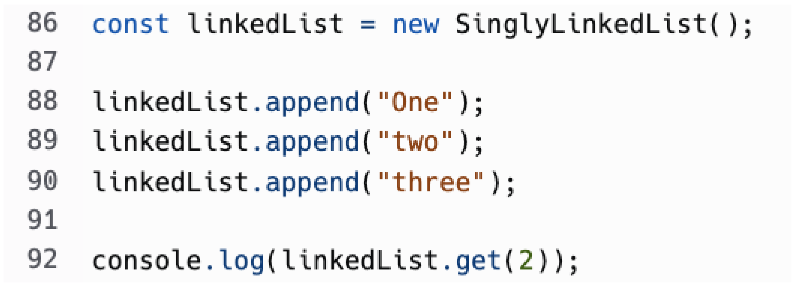
get(index): O(n)

Retrieve a node by its position in the list

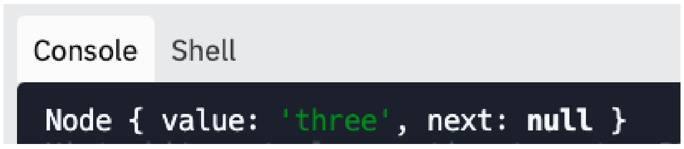
- Accept an index
- **Edge case:** 0 > index > this.length => returns null.
- Iterate through list until reach index and return node.



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



```
const linkedList = new SinglyLinkedList();

linkedList.append("One");
linkedList.append("two");
linkedList.append("three");

console.log(linkedList.get(2));
```

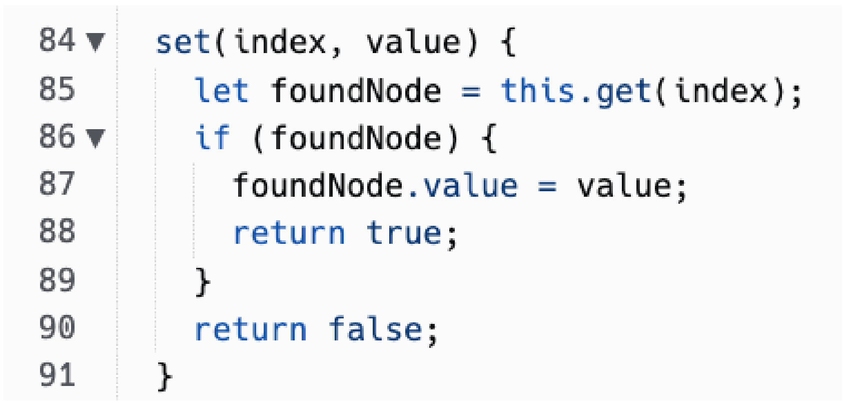

Console Shell

```
Node { value: 'three', next: null }
```

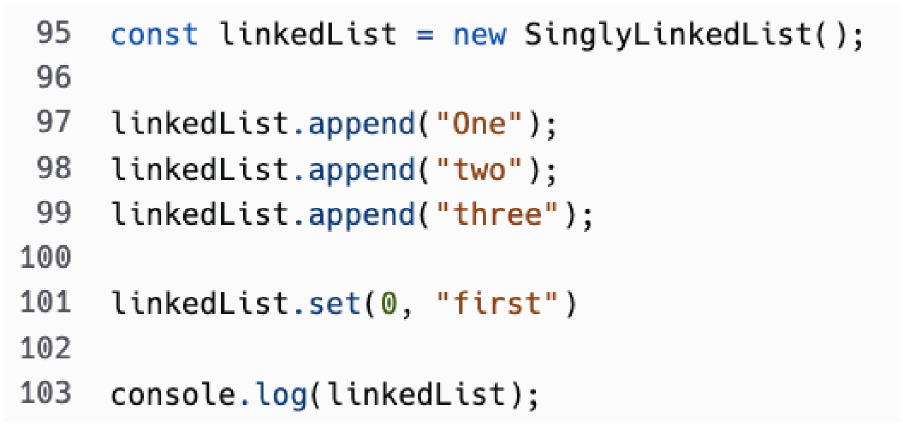
set(index, value): O(n)

Changes value of a node based on its position in the list

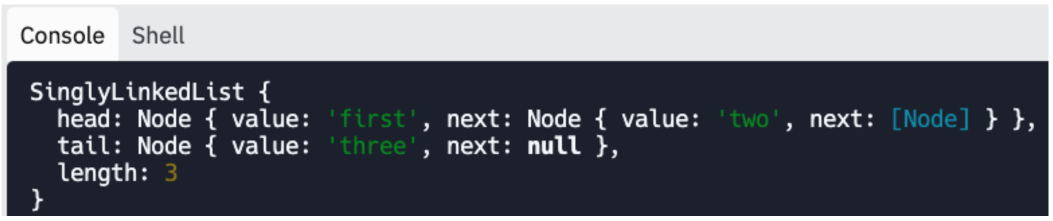
- Accepts an index and value.
- Use **get** method to find specific node
  - If not found, return false.
  - If found, set value of node to value passed.



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



```javascript
const linkedList = new SinglyLinkedList();

linkedList.append("One");
linkedList.append("two");
linkedList.append("three");

linkedList.set(0, "first")

console.log(linkedList);
```

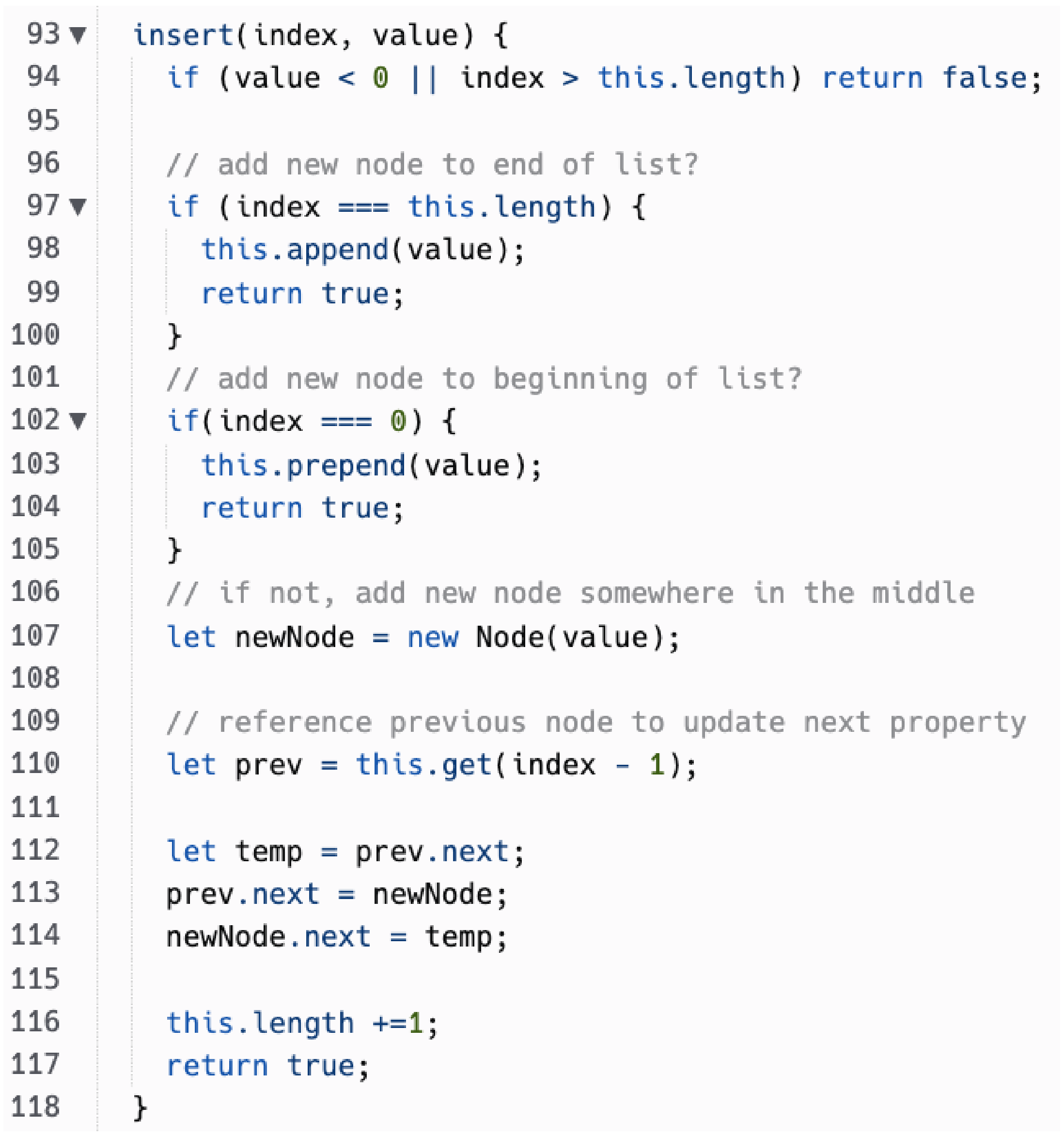


```
SinglyLinkedList {
  head: Node { value: 'first', next: Node { value: 'two', next: [Node] } },
  tail: Node { value: 'three', next: null },
  length: 3
}
```

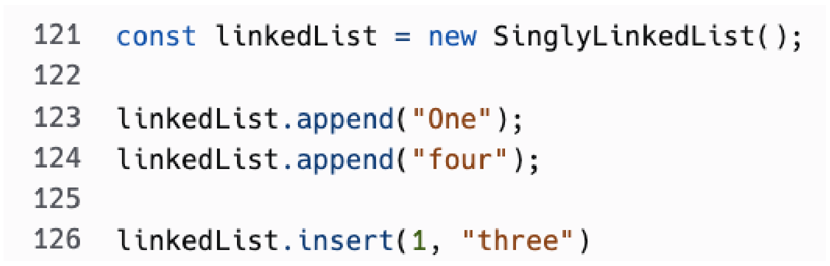
insert(index, value): O(n)

Adding a node to list at a specific position

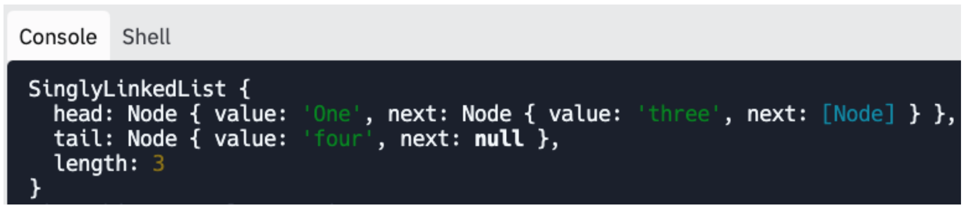
- **Edge cases:**
  - 0 < index > this.length => return false
  - index === this.length => add to end of list
  - index === 0 => add to beginning of list
- Use **get** method, access node at index.
  - Reference previous node (index -1)
    - Reference previous node **next node**.
    - Set the previous **next node** next property to new node.
    - Set the new node **next node** will be previously next node.
  - **Note:** Need to reference the previous node (index - 1) because we need to change the next property.
- Increment length.
- Return true.



```javascript
insert(index, value) {
  if (value < 0 || index > this.length) return false;

  // add new node to end of list?
  if (index === this.length) {
    this.append(value);
    return true;
  }
  // add new node to beginning of list?
  if(index === 0) {
    this.prepend(value);
    return true;
  }
  // if not, add new node somewhere in the middle
  let newNode = new Node(value);

  // reference previous node to update next property
  let prev = this.get(index - 1);

  let temp = prev.next;
  prev.next = newNode;
  newNode.next = temp;

  this.length +=1;
  return true;
}
```



```javascript
const linkedList = new SinglyLinkedList();

linkedList.append("One");
linkedList.append("four");

linkedList.insert(1, "three")
```



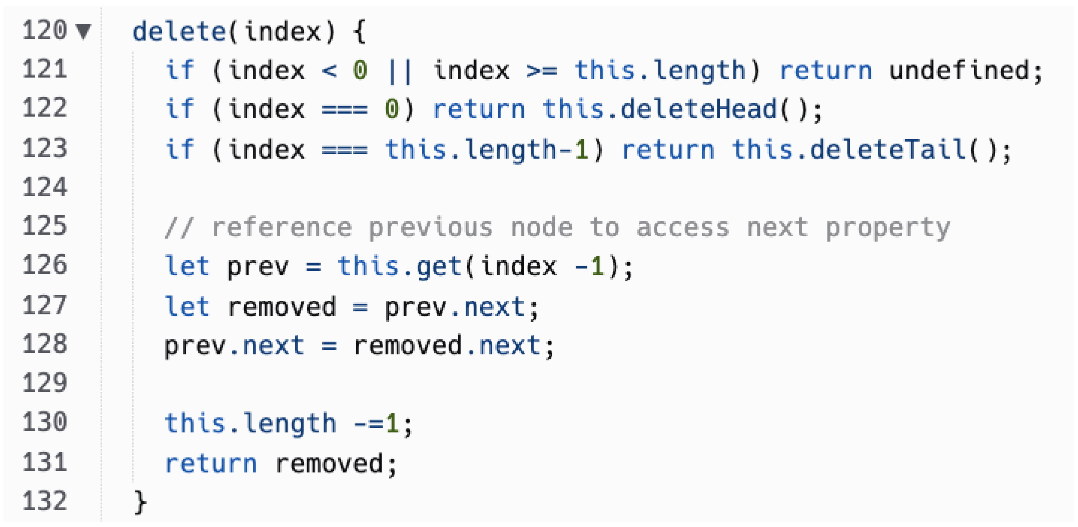
Console Shell

```
SinglyLinkedList {
  head: Node { value: 'One', next: Node { value: 'three', next: [Node] } },
  tail: Node { value: 'four', next: null },
  length: 3
}
```

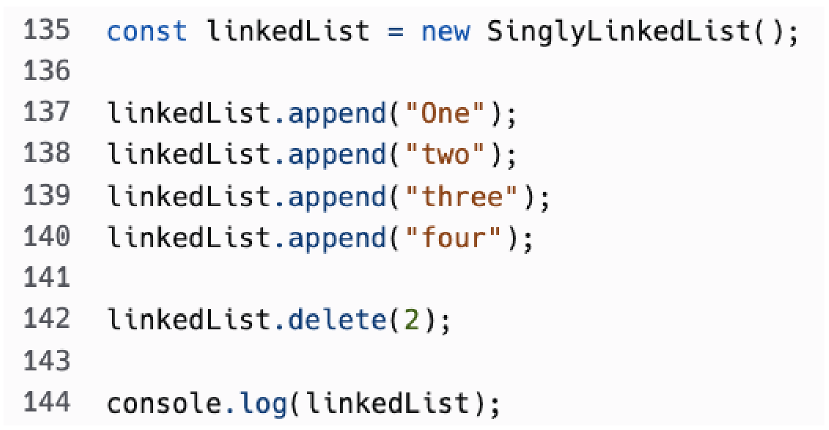
delete(index): O(n)

Removes a node from a list at a specific position

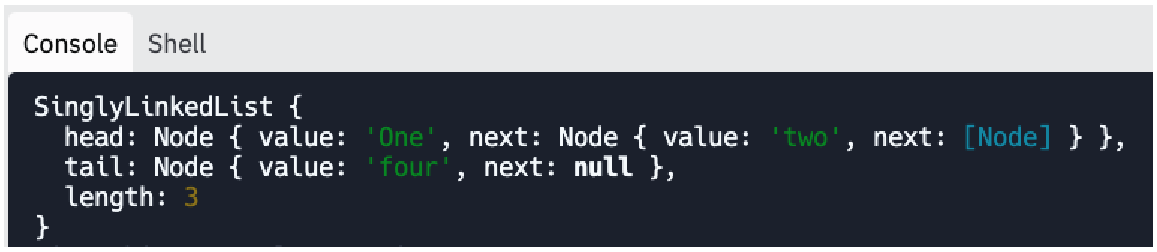
- Edge cases:
    - 0 > index >= this.length => return undefined
    - index === this.length -1 => use **deleteTail**
    - index === 0 => use **deleteHead**
- Use **get** method to previous node to reference next property.
- Update next node of previous node.
- Increment length.
- Return removed node (not referenced by next property).



```
  delete(index) {
    if (index < 0 || index >= this.length) return undefined;
    if (index === 0) return this.deleteHead();
    if (index === this.length-1) return this.deleteTail();

    // reference previous node to access next property
    let prev = this.get(index -1);
    let removed = prev.next;
    prev.next = removed.next;

    this.length -=1;
    return removed;
  }
```



```
const linkedList = new SinglyLinkedList();

linkedList.append("One");
linkedList.append("two");
linkedList.append("three");
linkedList.append("four");

linkedList.delete(2);

console.log(linkedList);
```



```
SinglyLinkedList {
  head: Node { value: 'One', next: Node { value: 'two', next: [Node] } },
  tail: Node { value: 'four', next: null },
  length: 3
}
```

Some common linked list problems

Reverse a List
https://leetcode.com/problems/reverse-linked-list/

Remove Duplicates (sorted list)
https://leetcode.com/problems/remove-duplicates-from-sorted-list/

Palindrome Linked List
https://leetcode.com/problems/palindrome-linked-list/

Merge Two Sorted Lists
https://leetcode.com/problems/merge-two-sorted-lists/