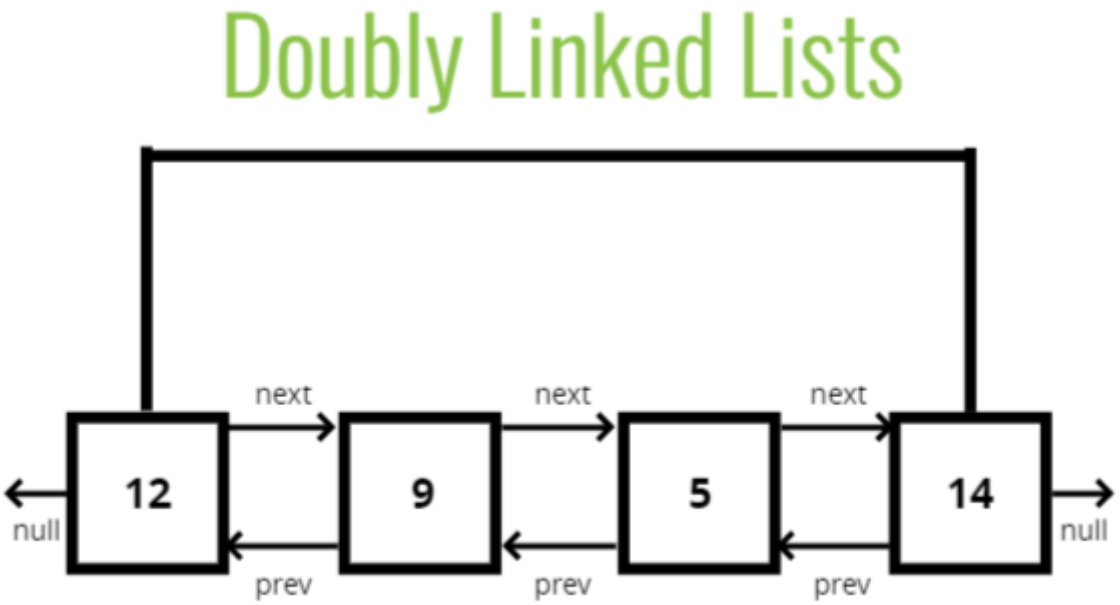
Doubly Linked List

All that we do is add a pointer to the prvious node and the next node. So each node points in two directions.

Almost identical to Singly Linked Lists, except every node has another pointer, to the previous node.

No indexing



It's a tiny change but has a large impact on some of the code we erite and the efficiency of certain operations.

Eg, when we're popping off the last item in a singly linked list, we had to go through all of the nodes because we had no other way of finding out the 2nd last node.

In doubly linked list, we can just go to the tail and get the 2nd last node from it by moving backward.

Imagine that you want to print out the singly lined list in reverse. You start at the beginning, go all the way to the end and because you can't go backward, you start over to go to the 2nd last item then do the same for 3rd last and so on. It's horribly inefficient.

Singly Linked List:

Doubly Linked List:

More memory === More Flexibility

It's **almost** always a tradeoff!

Drawback of doubly linked list is that it takes up more memory.

Node

- val
- next
- prev

Doubly Linked List

- head
- tail
- length

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

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

```
first = new Node("A")

first.next = new Node("B")

first.next.prev = first

console.log(first);
```

```
▼ Node {val: 'A', next: Node, prev: null}
  ▶ next: Node {val: 'B', next: null, prev: Node}
    prev: null
    val: "A"
  ▶ [[Prototype]]: Object
```

Pushing pseudocode

- Create a new node with the value passed to the function
- If the head property is null set the head and tail to be the newly created node
- If not, set the next property on the tail to be that node
- Set the previous property on the newly created node to be the tail
- Set the tail to be the newly created node
- Increment the length
- Return the Doubly Linked List

```
push(val) {
  let node = new Node(val);
  if (this.length == 0) {
    this.head = node;
    this.tail = node;
  } else {
    let tail = this.tail;
    tail.next = node;
    node.prev = tail;
    this.tail = node;
  }
  this.length++;
  return this;
}
```

Popping pseudocode

- If there is no head, return undefined
- Store the current tail in a variable to return later
- If the length is 1, set the head and tail to be null
- Update the tail to be the previous Node.
- Set the newTail's next to null
- Decrement the length
- Return the value removed

```
pop() {
  if (this.length === 0) {
    return undefined;
  }
  let oldTail = this.tail;
  if (this.length === 1) {
    this.head = null;
    this.tail = null;
  } else {
    let newTail = oldTail.prev;
    this.tail = newTail;
```

```
    newTail.next = null;
    oldTail.prev = null;
  }
  this.length--;
  return oldTail;
}
```

Shifting pseudocode

- If length is 0, return undefined
- Store the current head property in a variable (we'll call it old head)
- If the length is one
  - set the head to be null
  - set the tail to be null
- Update the head to be the next of the old head
- Set the head's prev property to null
- Set the old head's next to null
- Decrement the length
- Return old head

```
shift() {
  if (this.length === 0) {
    return undefined;
  }

  let oldHead = this.head;
  if (this.length === 1) {
    this.head = null;
    this.tail = null;
  } else {
    let newHead = oldHead.next;
    this.head = newHead;

    newHead.prev = null;
    oldHead.next = null;
  }
  this.length--;
  return oldHead;
}
```

Unshifting pseudocode

- Create a new node with the value passed to the function
- If the length is 0
  - Set the head to be the new node
  - Set the tail to be the new node
- Otherwise
  - Set the prev property on the head of the list to be the new node
  - Set the next property on the new node to be the head property
  - Update the head to be the new node
- Increment the length
- Return the list

```
unshift(value) {
  let newNode = new Node(value);
  if (this.length === 0) {
    this.head = newNode;
    this.tail = newNode;
  } else {
    let oldHead = this.head;
    oldHead.prev = newNode;
    newNode.next = oldHead;
    this.head = newNode;
  }
  this.length++;
  return this;
}
```

Get Pseudocode

- If the index is less than 0 or greater or equal to the length, return null
- If the index is less than the length of the list
  - Loop through the list starting from the head and loop towards the middle
  - Return the node once it is found
- If the index is greater than or equal to half the length of the list
  - Loop through the list starting from the tail and loop towards the middle
  - Return the node once it is found

```
get(i) {
  if (i < 0 || i >= this.length) {
    return undefined;
  }
  let node = null;
  if (i < this.length / 2) {
    console.log("IF");
    let count = 0;
    node = this.head;
    while (count < i) {
      node = node.next;
      count++;
    }
  } else {
    console.log("ELSE");
    let count = this.length - 1;
    node = this.tail;
    while (count > i) {
      node = node.prev;
      count--;
    }
  }
  return node;
}
```

Set pseudocode

- Create a variable which is the result of the **get** method at the index passed to the function
  - If the **get** method returns a valid node, set the value of that node to be the value passed to the function
  - Return true
- Otherwise, return false

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

Insert pseudocode

- If the index is less than zero or greater than or equal to the length return false
- If the index is 0, **unshift**
- If the index is the same as the length, **push**
- Use the **get** method to access the index -1
- Set the next and prev properties on the correct nodes to link everything together
- Increment the length
- Return true

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

  if (i === 0) return !!this.unshift(value);
  // if (i === 0) {
  //   this.unshift(value);
  //   return true;
  // }

  if (i === this.length) {
    this.push(value);
    return true;
  }

  let prevNode = this.get(i - 1);
  let newNode = new Node(value);
  let nextNode = prevNode.next;

  prevNode.next = newNode;
  newNode.prev = prevNode;
  newNode.next = nextNode;
  nextNode.prev = newNode;
  this.length++;
  return true;
}
```

Remove pseudocode

- If the index is less than zero or greater than or equal to the length return undefined
- If the index is 0, **shift**
- If the index is the same as the length-1, **pop**
- Use the **get** method to retrieve the item to be removed
- Update the next and prev properties to remove the found node from the list
- Set next and prev to null on the found node
- Decrement the length
- Return the removed node.

```
remove(i) {
  if (i < 0 || i >= this.length) {
    return undefined;
  }
  if (i === 0) {
    return this.shift();
  }
  if (i === this.length - 1) {
    return this.pop();
  }

  let removedNode = this.get(i);
  let prevNode = removedNode.prev;
  let nextNode = removedNode.next;

  removedNode.next = null;
  removedNode.prev = null;

  prevNode.next = nextNode;
  nextNode.prev = prevNode;
  this.length--;

  return removedNode;
}
```

RECAP!

- Doubly Linked Lists are almost identical to Singly Linked Lists except there is an additional pointer to previous nodes
- Better than Singly Linked Lists for finding nodes and can be done in half the time!
- However, they do take up more memory considering the extra pointer

Time Complexity

Insertion	O(1)	
Removal	O(1)	i.e, not the case for SLL
Searching	O(n)	Technically, searching is O(n/2), but that's still O(n)
Access	O(n)	

Comparing Singly And Doubly Linked List

Having double pointer does actually make certain things mush easier. If you need to access your data in reverse manner i.e, browser history.
It is often done with a DLL because every page that you're on has a next and a previous.

Also DLL are better for finding things because it can be done in half the time but it takes up more memory because of that extra pointer.