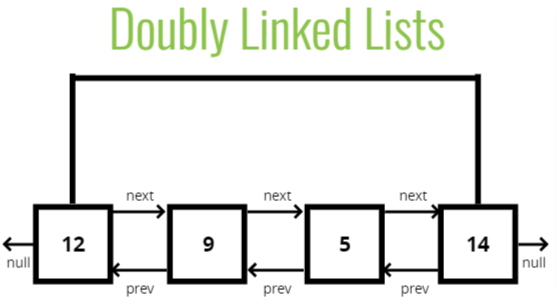
**Doubly Linked List**

**What is a Doubly linked list?**

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

****

**Comparisons B/W Singly & Doubly Linked Lists**

**Singly Linked Lists**

* It has only one pointer that points to the next node.
* It takes less memory than Doubly linked lists.
* Not better than Doubly Linked Lists for finding nodes.

**Doubly Linked Lists**

* It has 2 pointer. 1st pointer points to the next node whereas, 2nd pointer points the previous node.
* It takes more memory than Singly linked lists. Due to the presence of an extra pointer.
* Better than Singly Linked Lists for finding nodes and can be done in half the time!

**Note:**

More Memory === More Flexibility

It’s almost always a tradeoff ! (In Hindi: Samjota Karna)

**Important Note:**

**In Singly Linked List**,

* Referece of next Node stored in,
  + Head Node, it might be null or any Node.
  + Tail Node, it always be null.

**In Doubly Linked List,**

* Referece of next Node stored in,
  + Head Node, it might be null or any Node.
  + Tail Node, it always be null.
* Referece of previous Node stored in,
  + Head Node, it always be null.
  + Tail Node, it might be null or any Node

**Pushing:-**

Adding a new node to the **end** of the Doubly Linked List!

**Pushing Approach:**

* Create A Node i.e. Object consisting keys --> value: 'data' & next: null i.e. reference to the next node
* If(there is No Node i.e. head = null)

Assign head = newNode

Assign tail = newNode

Increase Length (Imagine: Creating Forward Arrow)

Return List

* If(There is some number of Nodes i.e head pointing to --> First Node)  
  So, tail has been assigning to the lastNode consisting --> value: 'data', next: null & prev = node
* Create a newNode.
  + Assign it's prev pointer to the tail node.
* Assign tail Node's next pointer to the new node.
  + Set tail to the newNode
* Increase Length (Imagine: Creating Forward Arrow)
* Return DoublyLinkedList

**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

**Pushing Code**

class Node {

constructor(val) {

this.value = val;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = this.head;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

}

const list = new DoublyLinkedList();

list.push(1);

list.push(2);

list.push(3);

list.push(4);

list.push(5);

**Output:**DoublyLinkedList {head: Node, tail: Node, length: 5}  
head: Node {value: 1, next: Node, prev: null}  
length: 5  
tail: Node {value: 5, next: null, prev: Node}

**Popping:-**

Removing a node from the **end** of the Doubly Linked List!

**Popping Approach:**

* First of all, POP Operation (Removing Elmnt from End) can only be perform if their is at least one Node in the list.
* So, lets assume there's already a created list having 5 nodes i.e. length=5, head pointing to 1st Node & tail pointing to last Node.
* Each node storing value & reference to the next node.
* For Popping:
* If(there is No Node i.e. this.head = null or this.length===0)
  + return undefined

• If(There is some number of Nodes i.e head pointing to --> First Node)

So, tail has been assigning to the lastNode might consisting --> value: 'data', next: null & prev = node

• Create a pointer remvNode.

- Set it to the tail Node.

• Assign tail Pointer to its tail.prev.

• Assign tail.next = null;

• Set remvNode.next & remvNode.prev to null.

* Edge Case:  
  If(while poping, when length of list becomes === 1)
  + Assign this.head = null;
  + Assign this.tail = null;

• decrease Length

• Return remvNode

**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

**Popping Code**

class Node {

constructor(val) {

this.value = val;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = this.head;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

pop(){

if(!this.head) return undefined;

let remvNode = this.tail;

if(this.length===1){

this.head = null;

this.tail = null;

}else{

this.tail = this.tail.prev;

this.tail.next = null;

remvNode.prev = null;

remvNode.next = null;

}

this.length--;

return remvNode;

}

}

const list = new DoublyLinkedList();

list.push(1);

list.push(2);

list.push(3);

list.push(4);

list.push(5);

list.pop(); ***//This will pop out each node per execution***

**Output:**Node {value: 5, next: null, prev: null}

**Shifting:-**

Removing a new node from the beginning of the Doubly Linked List!

**Shifting Approach:**

• If(there is No Node i.e. this.head = null or this.length===0)

-return undefined

• If(There is some number of Nodes i.e head pointing to --> First Node)

So, tail has been assigning to the lastNode might consisting --> value: 'data', next: null & prev = node

• Create a pointer remvNode.

- Set it to the head Node.

• Edge Case:

If(while poping, when length of list becomes === 1)

- Assign this.head = null;

- Assign this.tail = null;

• Assign head Pointer to its head.next.

• Assign tail.prev = null;

• Set remvNode.next & remvNode.prev to null.

• decrease Length

• Return remvNode

**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

**Shifting Code**

class Node {

constructor(val) {

this.value = val;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = this.head;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

shift(){

if(!this.head) return undefined;

let remvNode = this.head;

if(this.length===1){

this.head = null;

this.tail = null;

}else{

this.head = this.head.next;

this.head.prev = null;

remvNode.prev = null;

remvNode.next = null;

}

this.length--;

return remvNode;

}

}

const list = new DoublyLinkedList();

list.push(1);

list.push(2);

list.push(3);

list.push(4);

list.push(5);

list.shift(); ***//This will shift out each node per execution***

**Output:**Node {value: 1, next: null, prev: null}

**Unshifting:-**

Adding a new node to the beginning of the Doubly Linked List!

**Unshifting Approach**

* Create a newNode with some value.

• If(there is No Node i.e. this.head = null or this.length===0)

* Assign this.head & this.tail pointer to newNode

• If(There is some number of Nodes i.e head pointing to --> First Node)

* So, tail has been assigning to the lastNode might consisting --> value: 'data', next: null & prev = node

• Assign headNode.prev to newNode.

• Assign newNode.next to headNode;

• Set head Pointer to newNode

• Increase Length ++;

• Return this.head;

**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
* Return the linked list

**Unshifting Code**

class Node {

constructor(val) {

this.value = val;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = this.head;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

unshift(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = newNode;

}else{

this.head.prev = newNode;

newNode.next = this.head;

this.head = newNode;

}

this.length++;

return this.head;

}

}

const list = new DoublyLinkedList();

list.push(1);  
list.push(2);  
list.push(3);  
list.push(4);  
list.push(5);

list.unshift(0) ***//Execute for unshift operation***

**Output:**DoublyLinkedList {head: Node, tail: Node, length: 1}  
head: Node {value: '0', next: null, prev: null}  
length: 1  
tail: Node {value: '0', next: null, prev: null}

**Get:-**

Retrieving a node by it's position in the Doubly Linked List!

**Get Approach:**

* Get method accept, an Index(index);

• If(index <0 || index>= list's length) return null;

• Initialize a counter with 0 & Define a pointer current;

• If(index <= listLenght/2)

- Assign Current = this.head.next;

- Traverse list from Start to end, till current !== index;

- counter++;

• If(index > listLenght/2)

- Assign Current = this.tail.prev;

- counter = this.length-1;

- Traverse list from end to Start, till current !== index;

- counter--;

• Return current;

**Get Pseudocode**

* If the index is less than 0 or greater or equal to the length, return null
* If the index is less than or equal to half 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 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 Code:**

class Node {

constructor(val) {

this.value = val;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = this.head;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

get(index){

if(index<0 || index>=this.length) return null;

let counter =0, current;

if(index <= this.length/2){

current = this.head;

while(counter!==index){

current = current.next;

counter++;

}

}

if(index > this.length/2){

current = this.tail;

counter = this.length-1;

while(counter!==index){

current = current.prev;

counter--;

}

}

return current;

}

}

const list = new DoublyLinkedList();

list.push(1);

list.push(2);

list.push(3);

list.push(4);

list.push(5);

list.get(3) //***To execute get method.***

**Output:**Node {value: 4, next: Node, prev: Node}

**Set:-**

Changing the value of a node based on it's position in the Doubly Linked List

**Set Approach:**

* Set method accept, an Index & Value (index, val);

• Create a foundNode and Assign the return Node using get method at given index;

• If(Node Found)

- Change the value of the foundNode.

- return true;

else

- return false;

**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 Code:**

class Node {

constructor(val) {

this.value = val;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = this.head;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

get(index){

if(index<0 || index>=this.length) return null;

let counter =0, current;

if(index <= this.length/2){

current = this.head;

while(counter!==index){

current = current.next;

counter++;

}

}

if(index > this.length/2){

current = this.tail;

counter = this.length-1;

while(counter!==index){

current = current.prev;

counter--;

}

}

return current;

}

set(index, val){

let foundNode = this.get(index);

if(foundNode){

foundNode.value = val;

return true;

}

else

return false;

}

}

const list = new DoublyLinkedList();

list.push(1);

list.push(2);

list.push(3);

list.push(4);

list.push(5);

list.set(0, "Hello"); ***//set() method execute set operation on list.***

**Output:**true **//Changed 1 to “hello”**

**Insert:-**

Adding a node to the Doubly Linked List at a specific position

**Insert Approach:-**

* Insert method accept, an Index & Value (index, val) to Insert a newNode with some value at any position;

• If(index === 0) return unshift(val);

• If(index === this.length) return push(val);

• Create a newNode;

• If(!this.head);

- Assign head & tail pointer to newNode

• Otherwise:

- Assign the found node at index-1 position to beforeNode using get();

- Assign the beforeNode.next to afterNode

- Assign the beforeNode.next to newNode

- Assign the afterNode.prev to newNode

- Assign the newNode.prev to beforeNode

- Assign the newNode.next to afterNode

• Increase length++;

• return newNode;

**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 Code:**

class Node {

constructor(val) {

this.value = val;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = this.head;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

unshift(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = newNode;

}else{

this.head.prev = newNode;

newNode.next = this.head;

this.head = newNode;

}

this.length++;

return this.head;

}

get(index){

if(index<0 || index>=this.length) return null;

let counter =0, current;

if(index <= this.length/2){

current = this.head;

while(counter!==index){

current = current.next;

counter++;

}

}

if(index > this.length/2){

current = this.tail;

counter = this.length-1;

while(counter!==index){

current = current.prev;

counter--;

}

}

return current;

}

insert(index, val){

//Note: !! always gives true, if the function return something. Otherwise gives false.

if(index === 0) return !!this.unshift(val);

if(index === this.length) return !!this.push(val);

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = newNode;

}else{

let beforeNode = this.get(index-1);

if(beforeNode){

let afterNode = beforeNode.next;

beforeNode.next = newNode;

afterNode.prev = newNode;

newNode.prev = beforeNode;

newNode.next = afterNode;

}

else

return false;

}

this.length++;

return !!newNode;

}

}

const list = new DoublyLinkedList();

list.push(1);

list.push(2);

list.push(3);

list.push(4);

list.push(5);

list.insert(6, "last"); ***//Execute insert method .***  
list.insert(4, "Mid");  
list.insert(6, "last");

**Output:**true  
true  
true

**Remove:-**

Removing a node from the Doubly Linked List at a specific position

**Remove Approach**

* Remove method accept, only Index (index) to remove a Node at any position;

• If(index === 0) return shift();

• If(index === this.length-1) return pop();

• Create a foundNode;

• If(!this.head);

- return null;

• - Assign the found node at index position to removeNode using get();

- Assign the removeNode.next to afterNode

- Assign the removeNode.prev to beforeNode

- Assign the removeNode.next to null

- Assign the removeNode.prev to null

- Assign the beforeNode.next to afterNode

- Assign the afterNode.prev to beforeNode

- Assign the newNode.next to afterNode

• If(this.length ===1);

- this.head = null;

- this.tail = null;

• Decrease length--;

• return removeNode;

**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 Code:**

class Node {

constructor(val) {

this.value = val;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = this.head;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

pop(){

if(!this.head) return undefined;

let remvNode = this.tail;

if(this.length===1){

this.head = null;

this.tail = null;

}else{

this.tail = this.tail.prev;

this.tail.next = null;

remvNode.prev = null;

remvNode.next = null;

}

this.length--;

return remvNode;

}

shift(){

if(!this.head) return undefined;

let remvNode = this.head;

if(this.length===1){

this.head = null;

this.tail = null;

}else{

this.head = this.head.next;

this.head.prev = null;

remvNode.prev = null;

remvNode.next = null;

}

this.length--;

return remvNode;

}

get(index){

if(index<0 || index>=this.length) return null;

let counter =0, current;

if(index <= this.length/2){

current = this.head;

while(counter!==index){

current = current.next;

counter++;

}

}

if(index > this.length/2){

current = this.tail;

counter = this.length-1;

while(counter!==index){

current = current.prev;

counter--;

}

}

return current;

}

remove(index){

**//Note: !! always gives true, if the function return something. Otherwise gives false.**

if(index === 0) return !!this.shift();

if(index === this.length-1) return !!this.pop();

if(!this.head) return null;

let removeNode = this.get(index);

if(removeNode){

let beforeNode = removeNode.prev,

afterNode = removeNode.next;

removeNode.next = null;

removeNode.prev = null;

beforeNode.next = afterNode;

afterNode.prev = beforeNode;

}

else

return false;

if(this.length===1){

this.head = null;

this.tail = null;

}

this.length--;

return !!removeNode;

}

}

const list = new DoublyLinkedList();

list.push(1);

list.push(2);

list.push(3);

list.push(4);

list.push(5);

list.remove(0); ***//Execute remove method***  
list.remove(4);  
list.remove(2);

**Output:**true  
true  
true

**Reverse:-**

Reversing the Doubly Linked List in place!

**Information:**

Reversing DoublyLinkedList means, list must follow this rule (head.prev == null, head.next == node && tail.next == null, tail.prev == node) even after reverse.

- **Before Reverse**:

head tail

null <-- 0 --> 1 --> 2 --> 3 --> 4 --> 5 --> null

prev <-- <-- <-- <-- <-- nxt

arr: [0,1,2,3,4,5]

- **After Reverse**:

tail head

null <-- 0 <-- 1 <-- 2 <-- 3 <-- 4 <-- 5 --> null

next --> --> --> --> --> prev

arr: [5,4,3,2,1,0]

**Approach:**

* Swap tail pointer with head pointer.
* So, node pointer would be at tail node pointing same by head pionter after Swapping.
* Although node at tail & head pointer still acting like headNode & tailNode respectively.
* Initialize prevNode with null & counter with 0;
* Loop through the list from head pointer till counter !== this.length
* Assign node.next to node.prev;
* Set node.prev to null (As per the property of head node in Doubly Linked List)
* Set prevNode to node pointer
* Set node pointer to node.next
* Increase counter++;
* return this (DoublyLinkedList);

**Reverse Pseudocode**

* Create a variable called current and set it to be the head of the list
* Create a variable called tail and set it to be the head of the list
* Loop through the list and set the next property of the current node to be the prev property of the current node
* If there is no next property, set the tail to be the head and the head to be the current variable
* Return the list

**Reverse Code:**

class Node {

constructor(val) {

this.value = val;

this.next = null;

this.prev = null;

}

}

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val){

let newNode = new Node(val);

if(!this.head){

this.head = newNode;

this.tail = this.head;

}

else{

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

reverse(){

//Swapping

let node = this.tail;

this.tail = this.head;

this.head = node;

let prevNode = null,

counter =0;

while(counter !== this.length){

node.next = node.prev;

node.prev = prevNode;

prevNode = node;

node = node.next;

counter++;

}

return list;

}

print(){

let arr=[],

current = this.head;

while(current){

arr.push(current.value);

current = current.next;

}

return arr;

}

}

const list = new DoublyLinkedList();

list.push(1);

list.push(2);

list.push(3);

list.push(4);

list.push(5);

list.reverse(); ***//To execute reverse list***  
list.print(); ***//To print the reversed list***

**Output:**

DoublyLinkedList {head: Node, tail: Node, length: 5}  
head: Node {value: 5, next: Node, prev: null}  
length: 5  
tail: Node {value: 1, next: null, prev: Node}

[5, 4, 3, 2, 1]

**Big O of Doubly Linked Lists**

Insertion - **O(1)**

Removal - **O(1)**

Searching - **O(N)**

Access - **O(N)**

**Note:**Technically searching is O(N / 2), but that's still O(N)

**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
* Doubly linked lists are used to implement other data structures and certain types of caches