**Doubly Linked List**

*Pseudocode for Doubly Linked List:*

push(add node to end of the list):

* 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 new node
* Set the previous property of the newly created node to be the current tail
* Set the tail to be the newly created node
* Increment the length
* Return the list

pop(remove node at the end of the list):

* 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 tail to be the previous node
* Set the newTail’s next to be null
* Decrement the length by 1
* Return the value of removed node

shift(remove a node from the beginning of the list):

* If the length is 0, return undefined
* Store the current head property in a variable
* If the length is 1
  + 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 be null
* Decrement length by 1
* Return the old head

unshift(add a node to the beginning of the list):

* 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 of the new node to be the head of the list
  + Update the head to be the new node
* Increment the length by 1
* Return the list

get(access a node in a list by its position):

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

set(replace the value of a node by its position):

* Create a variable which is the result of the get method at the index passed to the function
* If the get method return a valid node, update the value and return true
* Otherwise, return false

insert(add a node at a specific position):

* If the index is less than 0 or greater than the length of the list, 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

remove(remove a node at a specific position):

* If the index is less than 0 or greater than the length of the list, return undefined
* If the index is 0, shift
* If the index is the same as the length – 1, pop
* Use the get method to access the index
* 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 by 1
* Return the removed node

*Full Code:*

// piece of data – val

// link to next node – next

// link to prev node – prev

class Node {

constructor(val) {

this.val = val;

this.next = null;

this.prev = null;

}

}

// head – tail – length

class DoublyLinkedList {

constructor() {

this.head = null;

this.tail = null;

this.length = 0;

}

push(val) {

var newNode = new Node(val);

if (!this.head) {

this.head = newNode;

this.tail = newNode;

} else {

this.tail.next = newNode;

newNode.prev = this.tail;

this.tail = newNode;

}

this.length++;

return this;

}

pop() {

if (!this.head) {

return undefined;

} else {

var popNode = this.tail;

if (this.length === 1) {

this.head = null;

this.tail = null;

} else {

this.tail = popNode.prev;

this.tail.next = null;

popNode.prev = null;

}

this.length--;

return popNode;

}

}

shift() {

if (!this.head) {

return undefined;

} else {

var oldHead = this.head;

if (this.length === 1) {

this.head = null;

this.tail = null;

} else {

this.head = oldHead.next;

this.head.prev = null;

oldHead.next = null;

}

this.length--;

return oldHead;

}

}

unshift(val) {

var newNode = new Node(val);

if (this.length === 0) {

this.head = newNode;

this.tail = newNode;

} else {

this.head.prev = newNode;

newNode.next = this.head;

this.head = newNode;

}

this.length++;

return this;

}

get(index) {

if (index < 0 || index >= this.length) {

return null;

} else {

var count;

var current;

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

count = 0;

current = this.head;

while (count != index) {

current = current.next;

count++;

}

} else {

count = this.length-1;

current = this.tail;

while (count != index) {

current = current.prev;

count--;

}

}

return current;

}

}

set(index, val) {

var foundNode = this.get(index);

if (foundNode != null) {

foundNode.value = val;

return true;

} else {

return false;

}

}

insert(index, val) {

if (index < 0 || index > this.length) {

return false;

} else if (index === 0) {

this.length++;

return !!this.unshift(val);

} else if (index === this.length) {

this.length++;

return !!this.push(val);

} else {

var newNode = new Node(val);

var prevNode = this.get(index-1);

var nextNode = prevNode.next;

prevNode.next = newNode;

newNode.prev = prevNode;

newNode.next = nextNode;

nextNode.prev = newNode;

this.length++;

return true;

}

return false;

}

remove(index) {

if (index < 0 || index > this.length) {

return undefined;

} else if (index === 0) {

this.length--;

return this.shift();

} else if (index === this.length -1) {

this.length--;

return this.pop();

} else {

var removeNode = this.get(index);

var prevNode = removeNode.prev;

var nextNode = removeNode.next;

prevNode.next = nextNode;

nextNode.prev = prevNode;

removeNode.next = null;

removeNode.prev = null;

this.length--;

return removeNode;

}

}

}

*Big O:*

Insertion – O(1)

Removal – O(1)

Search – O(N)

Access – O(N)