**Linked List**

*Definition:*

* A data structure that contains a head, tail, and length property
* Linked list consists of nodes, and each node has a value and a pointer to another node or null

*Pseudocode for method:*

push:

* This function accept a value
* Create a new node using the value passed to the function
* If there is no head property on the list, set the head and tail to be the newly created node
* Otherwise set the next property on the tail to be the new node and set the tail property on the list to be the newly created node
* Increment the length by one
* Return the linkedlist

pop:

* If there are no nodes in the list, return undefined
* Loop through the list until you reach the tail
* 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

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

* If there are no nodes, return undefined
* Store the current head property in a variable
* Set the head property to be the current head’s next property
* Decrement length by 1
* Return the value of the node removed

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

* This function should accept a value
* Create a new node using the value passed to the function
* If there is no head property on the list, set the head and tail to be the newly created node
* Otherwise set the newly created node’s next property to be the current head property on the list
* Set the head property on the list to be the newly created node
* Increment the length by 1
* Return the linked list

get(retrieve a node by its position in the Linked List):

* This function should accept an index
* If the index is less than zero or greater than or equal to the length of the list, return null
* Loop through the list until you reach the index and return the node at that specific index

set(change the value of a node based on its position in the Linked List):

* This function should accept a value and an index
* Use your “get” function to find the specific node
* If the node is not found, return false
* If the node is found, update the value of that node to be the value passed to the function and return true

insert(add a node to the Linked List at a specific position):

* This function should accept a value and an index
* If the index is less than zero or greater than the length, return false
* If the index is the same as the length, push as new node to the end of the list
* If the index is 0, unshift a new node to the start of the list
* Otherwise, using the get method, access the node at the (index-1)
* Set the next property on that node to be the new node
* Set the next property on the new node to be the previous next
* Increment the length
* Return true

remove(remove a node from the Linked List at a specific position):

* If the index is less than zero or greater than the length, return undefined
* If the index is the same as the (length-1), pop
* If the index is 0, shift
* Otherwise, use the get method, access the node at the (index-1)
* Set the next property on that node to be the next of the next node
* Decrement the length
* Return the value of the node removed

reverse(reversing the Linked List in place):

* Swap the head and tail
* Create a variable called next
* Create a variable called prev
* Create a variable called node and initialize it to the head property
* Loop through the list
* Set next to be the next property on whatever node is
* Set the next property on the node to be whatever prev is
* Set prev to be the value of the node variable
* Set the node variable to be the value of the next variable

*Starter Code:*

// piece of data – val

// reference to next node - next

class Node {

constructor(val) {

this.val = val;

this.next = null;

}

}

class SinglyLinkedList {

constructor() {

this.length = 0;

this.head = null;

this.tail = null;

}

push(val) {

var newNode = new Node(val);

if (!this.head) {

this.head = newNode;

this.tail = this.head;

} else {

this.tail.next = newNode;

this.tail = newNode;

}

this.length++;

return this;

}

pop() {

if (!this.head) return undefined;

var current = this.head;

var newTail = current;

while (current.next) {

newTail = current;

current = current.next;

}

this.tail = newTail;

this.tail.next = null;

this.length--;

if (this.length === 0) {

this.head = null;

this.tail = null;

}

return current;

}

traverse() {

var current = this.head;

while (current) {

current = current.next;

}

}

shift() {

if (!this.head) return undefined;

var currentHead = this.head;

this.head = currentHead.next;

this.length--;

if (this.length === 0) {;

this.tail = null;

}

return currentHead;

}

unshift(val) {

var newNode = new Node(val);

if (!this.head) {

this.head = newNode;

this.tail = newNode;

} else {

newNode.next = this.head;

this.head = newNode;

}

this.length++;

return this;

}

get(index) {

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

return null;

}

var counter = 0;

var current = this.head;

while (counter != index) {

current = current.next;

counter++;

}

return current;

}

set(index, value) {

var foundNode = this.get(index);

if (foundNode) {

foundNode.value = value;

return true;

}

return false;

}

insert(index, value) {

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

return false;

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

return !!this.push(value);

} else if (index === 0) {

return !!unshift(value);

} else {

var newNode = new Node(value);

var prevNode = this.get(index – 1);

var temp = prevNode.next;

prevNode.next = newNode;

newNode.next = temp;

this.length++;

return true;

}

return false;

}

remove(index) {

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

return undefined;

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

return !!this.pop();

} else if (index === 0) {

return !!this.shift();

} else {

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

var removeNode = prevNode.next;

prevNode.next = removeNode.next;

this.length--;

return removeNode;

}

}

reverse() {

var node = this.head;

this.head = this.tail;

this.tail = node;

var prev = null;

var next;

for (var i = 0; i < this.length; i++) {

next = node.next;

node.next = prev;

prev = node;

node = next;

}

return this;

}

}

*Big O:*

Push, Pop – O(1)

Removal – O(1) (remove from start) or O(N) (remove from end)

Searching – O(1)

Access – O(1)