Linked List - Notes

A linked list is a data structure in which each node points to another node. Unlike arrays, which have a fixed size, a linked list is a dynamic data structure that can allocate and deallocate memory at runtime. By the end of this chapter, you will understand how to implement and work with linked lists.

There are two types of linked lists discussed in this chapter: singly and doubly linked lists. Let's examine the singly linked list first.

Singly Linked Lists

The linked list data structure is one where each node (element) has reference to the next node

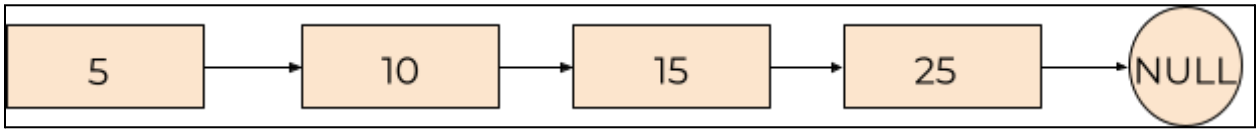


Figure for Single Linked List

A node in a singly linked list has the following properties: data and next.

data is the value for the linked list node, and **next** is a pointer to another instance of SinglyLinkedListNode.

```
function SinglyLinkedListNode(data){
this. data = data;
this.next = null;
}
```

Inserting Elements in a Linked List

The below javascript code shows how to insert into a singly linked list. If the head of the linked list is empty, the head is set to the new node. Otherwise, the old head is saved in temp, and the new head becomes the newly added node. Finally, the new head's next points to the temp (the old head).

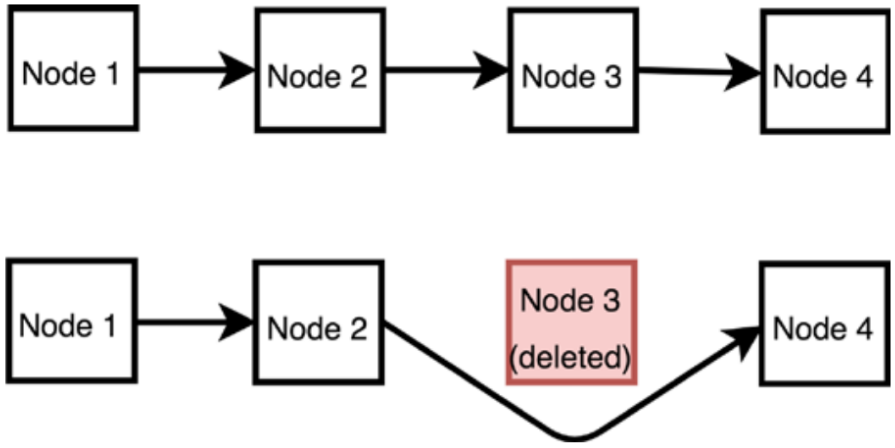
```
SinglyLinkedList.prototype.insert = function (value) {
if (this.head === null) { //If first node
this. head = new SinglyLinkedListNode(value);
```

```
}else{
var temp = this .head;
this.head = new SinglyLinkedListNode(value);
this.head.next = temp;}
this. size++;
}
var sll1 = new SinglyLinkedList();
sll1. insert (1); // linked list is now: 1 -> null
sll1. insert (12); // linked list is now: 12-> 1-> null
sll1. insert(20); // linked list is now: 20-> 12-> 1-> null
```

Time Complexity: O(1)
This is a constant time operation; no loops or traversal is required.

Deletion by Value

The deletion of a node in a singly linked list is implemented by removing the reference of that node. If the node is in the “middle” of the linked list, this is achieved by having the node with the next pointer to that node point to that node’s own next node instead.



If the node is at the end of the linked list, then the second-to-last element can dereference the node by setting its next to null.

```
SinglyLinkedList.prototype.remove = function (value) {
war currentHead = this. head;
if (currentHead .data == value) {
// just shift the head over. Head is now this new value
this. head = currentHead.next;
```

```
this. size--;
} else {
var prev = currentHead;
while (currentHead.next) {
if (currentHead.data == value) {
// remove by skipping
prev. next = currentHead.next;
prev currentHead;
currentHead= currentHead.next;
break; // break out of the loop
}
prev = currentHead;
currentHead= currentHead.next;
}
```

```
//If wasn't found in the middle or head, must be tail
if (currentHead.data == value) {
prev. next=null;
}
this. size--;
}
}
var sll1 = new SinglyLinkedList();
sll1. insert (1); // linked list is now: 1-> null
sll1. insert (12); // linked list is now: 12-> 1-> null
sll1. insert(20); // linked list is now: 20-> 12-> 1-> null
sll1.remove (12); // linked list is now: 20-> 1-> null
sll1.remove(20); // linked list is now: 1-> null
```

Time Complexity: O(n)
In the worst case, the entire linked list must be traversed.

Deletion at the Head

Deleting an element at the head of the linked list is possible in O(1). When a node is deleted from the head, no traversal is required. The implementation of this deletion is shown in the following code block. This allows the linked list to implement a stack. The last-added item (to the head) can be removed in O(1).

```
DoublyLinkedList.prototype.deleteAtHead = function() {
var toReturn = null;
if (this. head !== null) {
toReturn = this. head. data;
if (this. tail === this. head) {
this. head = null;
this. tail = null;
}
else {
this. head = this. head. next;
this.head.prev = null;}
}
this. size--;
return toReturn;
}
var s111 = new SinglyLinkedList();
sll1. insert (1); // linked list is now: 1-> null
sll1. insert (12); // linked list is now: 12-> 1-> null
sl11. insert (20); // linked list is now: 20 12-> 1-> null
sl11. deleteAtHead(); // linked list is now: 12-> 1-> null
```

Search

To find out whether a value exists in a singly linked list, simple iteration through all its next pointers is needed.

```
SinglyLinkedList.prototype.find = function(value) {
var currentHead = this. head;
while (currentHead.next) {
if (currentHead. data == value) {
return true;
}
currentHead = currentHead.next;
}
return false;
}
```

Time Complexity: O(n)
Like with the deletion operation, in the worst case, the entire linked list must be traversed.

Doubly Linked Lists

A doubly linked list can be thought of as a bidirectional singly linked list. Each node in the doubly linked list has both a next pointer and a prev pointer. The following code block implements the doubly linked list node:

```
function DoublyLinkedListNode(data) {
this.data = data;
this.next = null;
this prev = null;
}
```

In addition, a doubly linked list has a head pointer as well as a tail pointer. The head refers to the beginning of the doubly linked list, and the tail refers to the end of the doubly linked list. This is implemented in the following code along with a helper function to check whether the doubly linked list is empty:

```
function DoublyLinkedList () {
this. head = null;
this. tail = null;
this.size = 0;
}
DoublyLinkedList.prototype.isEmpty = function(){
return this. size == 0;
}
```

Each node in a doubly linked list has next and prev properties. Deletion, insertion, and search implementations in a doubly linked list are similar to that of the singly linked list. However, for both insertion and deletion, both next and prev properties must be updated. Below Figure shows an example of a doubly linked list.

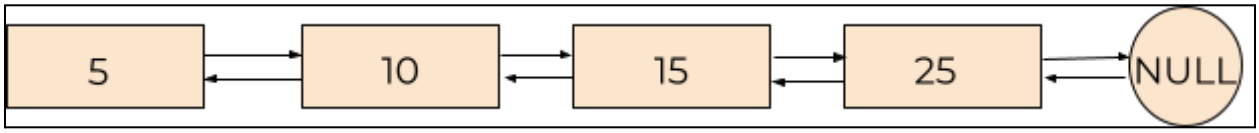


Figure For Doubly Linked List

Insertion at the Head

Inserting into the head of the doubly linked list is the same as the insertion for the singly linked list except that it has to update the prev pointer as well. The following code block shows how to insert into the doubly linked list. If the head of the linked list is empty, the head and the tail are set to the new node. This is because when there is only one element, that element is both the head and the tail. Otherwise, the temp variable is used to store the new

node. The new node's next points to the current head, and then the current head's prev points to the new node. Finally, the head pointer is updated to the new node.

```
DoublyLinkedList.prototype.addAtFront = function (value) {
if (this. head === null) { //If first node
this. head = new DoublyLinkedListNode(value);
this. tail = this. head;
} else {
var temp = new DoublyLinkedListNode(value);
temp. next = this. head;
this.head.prev temp;
this. head = temp;
}
this. size++;
}
var dll1 = new DoublyLinkedList();
dll1. insertAtHead(10); // dll1 structure: tail: 10 head: 10
dll1. insertAtHead (12); // dll1 structure: tail: 10 head: 12
dll1. insertAtHead(20); // dll1 structure: tail: 10 head: 20
```

Time Complexity: O(1)

Insertion at the Tail

Similarly, a new node can be added to the tail of a doubly linked list, as implemented in the following code block:

```
DoublyLinkedList.prototype. insertAtTail = function (value) {
if (this. tail === null) { //If first node
this. tail = new DoublyLinkedListNode(value);
this. head = this. tail;
} else {
var temp = new DoublyLinkedListNode(value);
temp.prev = this. tail;
this. tail.next temp;
this. tail = temp;
}
this.size++;
}
var dll1 = new DoublyLinkedList();
```

```
dll1. insertAtHead(10); // dll1 structure: tail: 10 head: 10
dll1. insertAtHead(12); // dll1 structure: tail: 10 head: 12
dll1. insertAtHead(20); // dll1 structure: tail: 10 head: 20
dll1. insertAtTail(30); // dll1 structure: tail: 30 head: 20
```

Time Complexity: O(1)

Deletion at the Head

Removing a node at the head from a doubly linked list can be done in O(1) time. If there is only one item in the case that the head and the tail are the same, both the head and the tail are set to null. Otherwise, the head is set to the head's next pointer. Finally, the new head's prev is set to null to remove the reference of the old head. This is implemented in the following code block. This is great because it can be used like a dequeue function from the queue data structure.

```
DoublyLinkedList.prototype.deleteAtHead = function() {
var toReturn = null;
if (this. head !== null) {
toReturn = this.head. data;
if (this. tail this.head) {
this. head = null;
this. tail = null;
} else {
this. head = this. head. next;
this.head.prev = null;
}
}
this. size--;
return toReturn;
}
```

Time Complexity: O(1)

Deletion at the Tail

Similarly to removing the node at the head, the tail node can be removed and returned in O(1) time, as shown in the following code block. By having the ability to remove at the tail as well, the doubly linked list can also be thought of as a bidirectional queue data structure. A queue can dequeue the first-added item, but a doubly linked list can dequeue either the item at the tail or the item at the head in O(1) time.

```
DoublyLinkedList.prototype.deleteAtTail = function() {
var toReturn = null;
if (this. tail !== null) {
toReturn = this.tail. data;
if (this. tail === this. head) {
this. head null;
this. tail null;
} else {
this. tail this. tail.prev;
this. tail.next = null;
}
}
this. size--;
return toReturn;
}
var dll1 = new DoublyLinkedList();
dll1. insertAtHead(10); // dll1 structure: tail: 10 head: 10
dll1. insertAtHead(12); // dll1 structure: tail: 10 head: 12
dll1. insertAtHead(20); // dll1 structure: tail: 10 head: 20
dll1. insertAtTail(30); // dll1 structure: tail: 30 head: 20
dll1. deleteAtTail(); // dll1 structure: tail: 10 head: 20
```

Time Complexity: O(1)

Table for Time Complexity for Singly and Doubly Li

Operations	Time Complexity for Singly Linked List	Time Complexity for Doubly Linked List
Insertion	O(1)	O(1)
Deletion	O(N)	O(1)
Searching	O(N)	O(N)

Applications of linked list in the real world:

1. When we are creating an image viewer to access previous and next image for preview using linked lists.
2. To move across web - pages we use linked list, like you visit google.com and next page you visit facebook.com so you move b