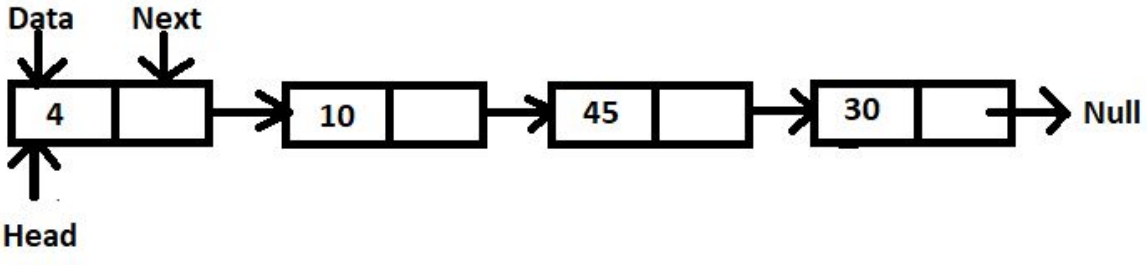
Linked Lists

Relevel
by Unacademy

Introduction to Linked Lists

A Linked List is a linear data structure used to store collections of data.
It is dynamic in nature i.e. can grow and shrink in size during program execution.
Data in linked list is stored as a node, which has two items

1. Data
2. Link or Pointer to next node (next nodes address)



- The first node of a linked list is referred to as Head.
- The last node has a reference to null (it always points to null).

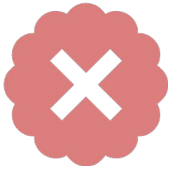
Properties of a linked list:

- Elements in Linked list are not stored at contiguous memory locations. i.e. they are stored at different locations in the memory.
- Successive elements are connected by link or pointers.
- Grows and shrinks in size during program execution and allocates memory as the list grows.

Advantages Of Linked List:

- Dynamic data structure.
- No memory wastage: Memory can be utilised efficiently as the size of the linked list increases/decreases at run time, there is no wastage of memory and pre-allocation of memory is not required.
- Implementation: Different linear data structures like stack and queues can be easily implemented using a linked list.
- Insertion and Deletion Operations: Insertion and deletion operations are comparatively easier, we can achieve it easily by updating the address in the pointer.

Disadvantages Of Linked List:



- Memory usage: In a linked list, the address of the next element is stored using a pointer which takes extra memory.
- Traversal is time-consuming: An element cannot be accessed directly in a Linked list as in an array. i.e, for accessing a node at position A, each node before A has to be traversed.
- Reverse Traversing: Accessing previous elements is not possible in a singly linked list, it can be achieved with a doubly-linked list, as it contains a pointer to the previous nodes as well.

Singly Linked Lists:

The linked list two fields data and pointer are referred as singly linked lists.

Creating a Node:

```
// Node class
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
  }
}
```

The Node class will create a Node for us which accepts and sets the data and creates a next pointer which is set to null by default.

Creating a SinglyLinkedList:

```
// linkedlist class
class SinglyLinkedList {
  constructor() {
    this.head = null;
    this.length = 0;
  }
}
```

The SinglyLinkedList class will have a head that specifies the start of the linked list. As we have not added any elements to the linked list by default head and tail will point to null. We also have a length property that will give us the current length of the linked list which is 0 now and we will increment it gradually as we add elements to the linked list.

Adding Node:

```
insertAtStart(data) {
     let newNode = new Node(data); //Create a new
node
     newNode.next = this.head; //Set next node of
new node to current head
    this.head = newNode; //Update the head pointer
to the new node
    this.length++;
  }

const ll = new SinglyLinkedList();
ll.insertAtStart(100);
ll.insertAtStart(500);
```

Traversing/Iterating and printing the linked list:

```
printLinkedList() {
    let current = this.head;

// print till current exists or current is truthy i.e. current != null
    while (current) {
     console.log(current);
      current = current.next; // update current to next node
    }
  }

ll.printLinkedList();
```

```
Node { data: 500, next: Node { data: 100, next: null } }
Node { data: 100, next: null }
```

```
// Modified print method
printLinkedList() {
    let current = this.head;

    while (current) {
      console.log(current.data);
      current = current.next;
    }
  }
```

Let us also add a method that displays the length of the linked list i.e. the number of elements currently present in the linkedlist.

```
// Size method to display length of list
size(){
      console.log("Length of Linked list is " + this.length);
  }
```

//Implementation

```
ll.insertAtStart(100);
ll.insertAtStart(500);
ll.insertAtStart(1000);

ll.printLinkedList();
ll.size();
```

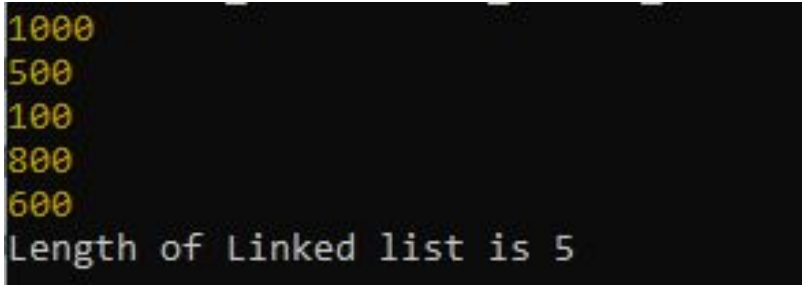
```
1000
500
100
Length of Linked list is 3
```

Insert node at the ending:

```
Insert node at the ending:

insertAtEnd(data) {
    let newNode = new Node(data); //Create a new node

    // if empty, create head
    if (this.head == null) {
      this.head = newNode;
     this.length++;
    } else {
      let current = this.head;
```

```
    // traverse till current.next != null
      while (current.next) {
        current = current.next;
      }
      current.next = newNode; //Set next of current node to new node
     this.length++;
    }
  }

ll.insertAtStart(100);
ll.insertAtStart(500);
ll.insertAtStart(1000);
ll.insertAtEnd(800);
ll.insertAtEnd(600);
```

```
ll.printLinkedList();
ll.size();
```

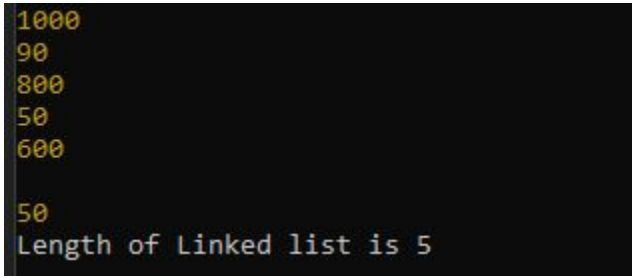


Insert node anywhere between start and end:

```
insertAtIndex(data, index) {
    // if first element
    if (index == 0) {
      this.insertAtStart(data);
    }
    // if out of range
    else if(index < 0 || index >= this.length){
        console.log("Array index out of bounds enter valid index");
    } else {
      let newNode = new Node(data); //Create a new node
      let current, previous; // maintain previous and current node
      current = this.head;
      let count = 0;
```

```
    // traverse till index - 1 as linked list has 0 based indexing
    while (count < index) {
      previous = current;
      current = current.next;
      count++;
    }
    newNode.next = current; //Set next of new node to current node
    previous.next = newNode; //Set next of previous node to new node
    this.length++;
  }
}
```

```
// Implementation
ll.insertAtStart(1000);
ll.insertAtEnd(800);
ll.insertAtEnd(600);

ll.insertAtIndex(50, 2);
ll.insertAtIndex(90, 1);

ll.printLinkedList();
ll.size();
```



```
// Modified implementaion
ll.insertAtStart(1000);
ll.insertAtEnd(800);
ll.insertAtEnd(600);

ll.insertAtIndex(50, 2);
ll.insertAtIndex(90, 1);

ll.insertAtIndex(10, 0);
ll.insertAtIndex(90, 10);

ll.printLinkedList();
ll.size();
```

```
Array index out of bounds enter valid index
10
1000
90
800
50
600

Length of Linked list is 6
```

Getting data of element at index:

```
getElement(index) {
    if (index < 0 || index >= this.length) {
      console.log('Array index out of bounds enter valid index');
    } else {
      let current = this.head;
      let count = 0;
      while (current) {
```

```
        if (count == index) {
          console.log(current.data);
        }
        count++;
        current = current.next;
      }
    }
  }
```

```
ll.insertAtStart(100);
ll.insertAtStart(500);
ll.insertAtStart(1000);
ll.insertAtEnd(800);
ll.insertAtEnd(600);

ll.printLinkedList();
console.log();
ll.getElement(3);
```

Delete element at Start:

```
removeAtStart() {
  if (!this.head) {
    return false;
  }
  let current = this.head;
  this.head = this.head.next;
  current = null;
  this.length--;
}
```

```
ll.insertAtStart(100);
ll.insertAtStart(500);
ll.insertAtStart(1000);
ll.insertAtEnd(800);
ll.insertAtEnd(600);

ll.removeAtStart();

ll.printLinkedList();
console.log();
```

```
500
100
800
600

600
Length of Linked list is 4
```

Deleting Node at index:

```
removeAt(index) {
    if (index == 0) {
      removeAtStart();
    } else if (index < 0 || index >= this.length) {
      console.log('Array index out of bounds enter valid index');
    } else {
      let current, previous;
      current = this.head;
      let count = 0;
      while (count < index) {
        count++;
        previous = current;
```

```
        current = current.next;
      }
      previous.next = current.next;
      console.log(current);
      this.length--;
    }
  }
```

```
ll.insertAtStart(100);
ll.insertAtStart(500);
ll.insertAtStart(1000);
ll.insertAtEnd(800);
ll.insertAtEnd(600);

ll.removeAt(4);

ll.printLinkedList();
console.log();
```

```
Node { data: 600, next: null }
1000
500
100
800
```

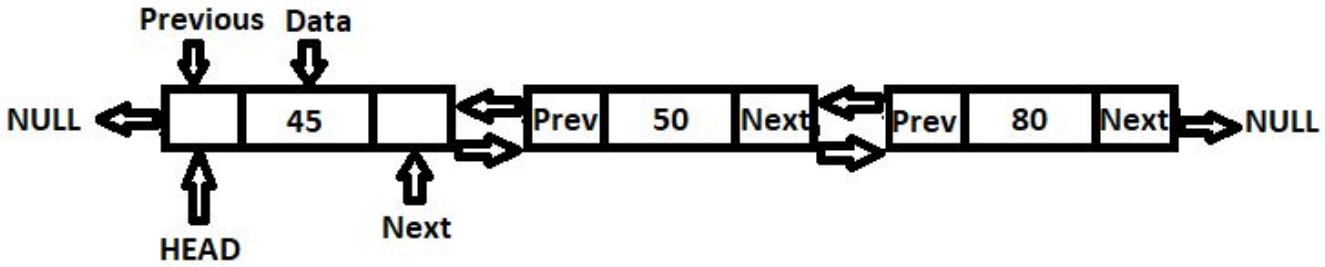
```
// Modified removeAt method
removeAt(index) {
    if (index == 0) {
      removeAtStart();
    } else if (index < 0 || index >= this.length) {
      console.log('Array index out of bounds enter valid index');
    } else {
      let current, previous;
      current = this.head;
      let count = 0;
      while (count < index) {
        count++;
        previous = current;
        current = current.next;
      }
```

```
      previous.next = current.next;
      current = null;
      console.log(current);
      this.length--;
    }
  }
```

```
ll.insertAtStart(100);
ll.insertAtStart(500);
ll.insertAtStart(1000);
ll.insertAtEnd(800);
ll.insertAtEnd(600);

ll.removeAt(4);

ll.printLinkedList();a
```

Doubly Linked Lists:

It's a two-way linked list as each node points to the next pointer as well as to the previous pointer.



Creating a Node:

```
// Node class
class Node {
  constructor(data) {
    this.data = data;
    this.previous = null;
    this.next = null;
  }
}
```

Creating a DoublyLinkedList:

```
// Doubly linked list class
class DoublyLinkedList {
  constructor() {
    this.head = null;
    this.length = 0;
  }
}
```

Adding Node:

Insert node at the beginning:

```
insertAtStart(data) {
    let newNode = new Node(data); // Create a new node
     newNode.next = this.head; // Set next node of new node to current
head
    newNode.previous = null;  // Set previous of new node as null
    // Update previous of head as new node only if had is not null
    if(this.head != null){
        this.head.previous = newNode;
    }
    this.head = newNode; // Update the head pointer to the new node
    this.length++;
}
```

Printing or traversing:

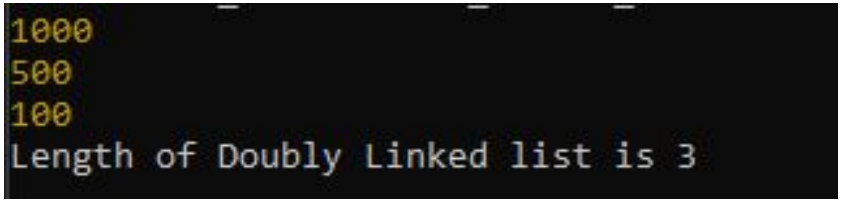
```
printLinkedList() {
    let current = this.head;

// print till current exists or current is truthy i.e. current != null
    while (current) {
     console.log(current);
      current = current.next; // update current to next node
    }
  }

const dll = new DoublyLinkedList();
dll.insertAtStart(100);
dll.insertAtStart(500);
dll.insertAtStart(1000);
dll.printLinkedList();
```

```
<ref *1> Node {
  data: 1000,
  previous: null,
  next: <ref *2> Node {
    data: 500,
    previous: [Circular *1],
    next: Node { data: 100, previous: [Circular *2], next: null }
  }
}
<ref *1> Node {
  data: 500,
  previous: Node { data: 1000, previous: null, next: [Circular *1] },
  next: Node { data: 100, previous: [Circular *1], next: null }
}
<ref *2> Node {
  data: 100,
  previous: <ref *1> Node {
    data: 500,
    previous: Node { data: 1000, previous: null, next: [Circular *1] },
    next: [Circular *2]
  },
  next: null
}
```

//Modified print method

```
printLinkedList() {
    let current = this.head;

    while (current) {
      console.log(current.data);
      current = current.next;
    }
  }

// size method to display length of list

size() {
    console.log('Length of Doubly Linked list is ' + this.length);
  }
```

```
dll.insertAtStart(100);
dll.insertAtStart(500);
dll.insertAtStart(1000);

dll.printLinkedList();
dll.size();
```

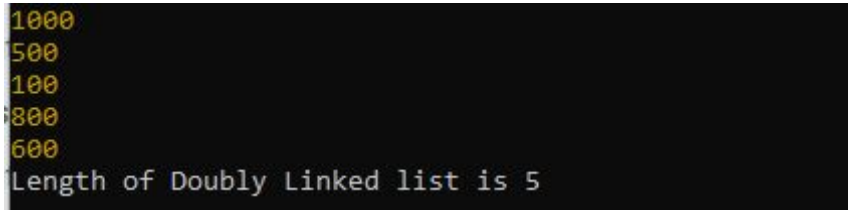


Insert node at the ending:

```
insertAtEnd(data) {
    let newNode = new Node(data); //Create a new node

    // if empty, create head
    if (this.head == null) {
      this.head = newNode;
      this.length++;
    } else {
      let current = this.head;

      // traverse till current.next != null
      while (current.next) {
        current = current.next;
      }
      current.next = newNode; // Set next of current node to new node
      newNode.previous = current; // Set previous of new node to current
node
```

```
      // Set next of new node node to null as it is the last node
     newNode.next = null;
      this.length++;
    }
  }
```

```
ll.insertAtStart(100);
dll.insertAtStart(500);
dll.insertAtStart(1000);
dll.insertAtEnd(800);
dll.insertAtEnd(600);

dll.printLinkedList();
dll.size();
```



Insert node anywhere between start and end:

```
insertAtIndex(data, index) {
    // if first element
    if (index == 0) {
      this.insertAtStart(data);
    }
    // if out of range
    else if (index < 0 || index >= this.length) {
      console.log('Array index out of bounds enter valid index');
    } else {
      let newNode = new Node(data); //Create a new node
      let current = this.head; // maintain current node
      let count = 0;

      // traverse till index - 1 as linked list has 0 based indexing
      while (count < index) {
        current = current.next;
```

```
        count++;
      }

     //Set previous of new node to previous of current node
      newNode.previous = current.previous;
      newNode.next = current; //Set next of new node to current node
      newNode.previous.next = newNode; //Set previous of new node's next
to new node
       current.previous = newNode; //Set previous of current node to new
node

      this.length++;
    }
  }
```

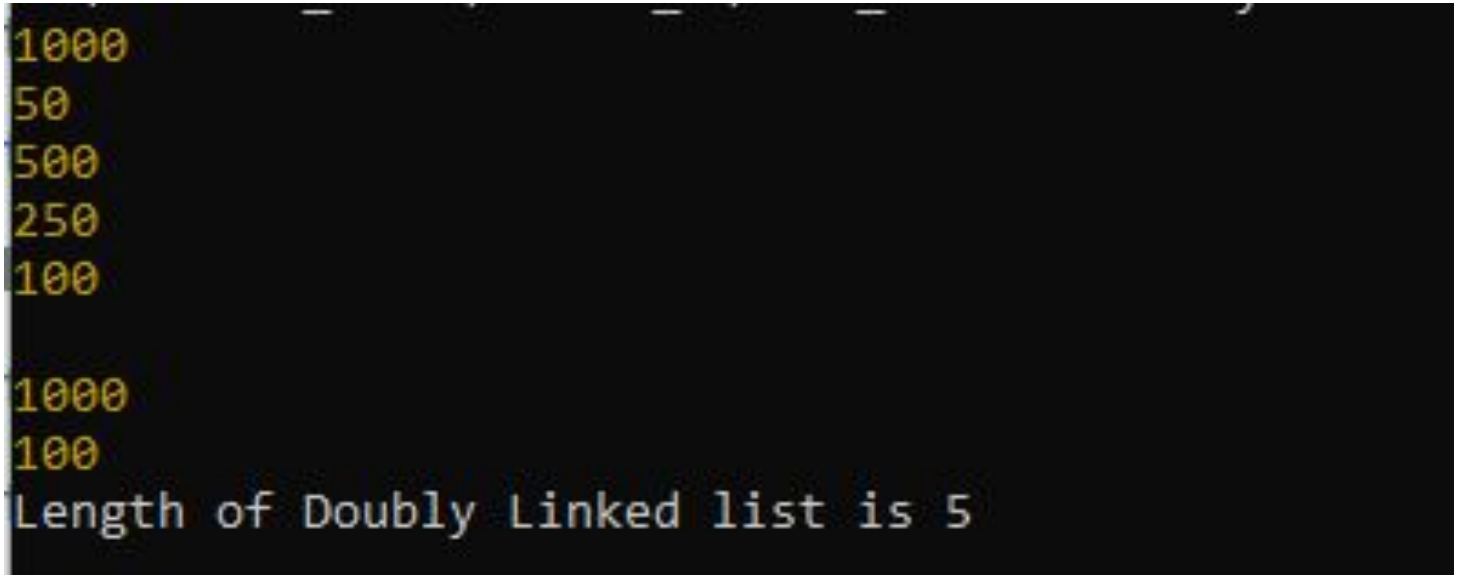
```
dll.insertAtStart(100);
dll.insertAtStart(500);
dll.insertAtStart(1000);

dll.insertAtIndex(50, 1);
dll.insertAtIndex(250, 3);
dll.printLinkedList();
dll.size();
```

```
1000
50
500
250
100
Length of Doubly Linked list is 5
```

Getting data of element at index:

```
getElement(index) {
    if (index < 0 || index >= this.length) {
      console.log('Array index out of bounds enter valid index');
    } else {
      let current = this.head;
      let count = 0;
      while (current) {
        if (count == index) {
          console.log(current.data);
        }
        count++;
        current = current.next;
      }
    }
  }
```

```
dll.insertAtStart(100);
dll.insertAtStart(500);
dll.insertAtStart(1000);
dll.insertAtIndex(50, 1);
dll.insertAtIndex(250, 3);

dll.printLinkedList();

console.log();

dll.getElement(0);
dll.getElement(4);

dll.size();
```


1000
50
500
250
100
1000
100
Length of Doubly Linked list is 5

Deleting Node at Start:

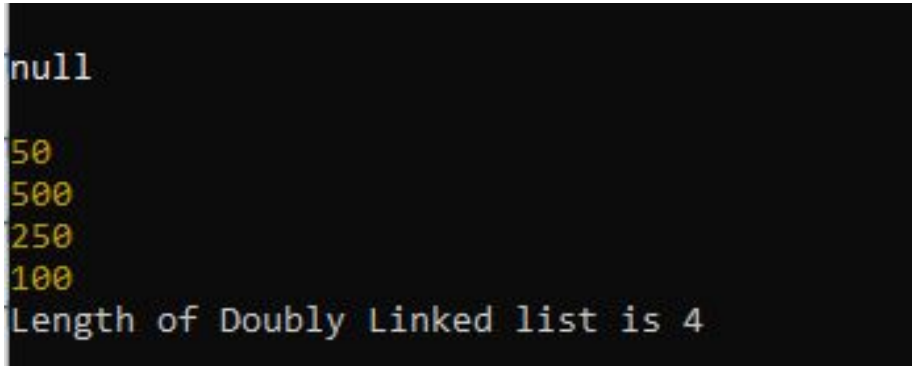
```
removeAtStart() {
    if (!this.head) {
      return false;
    }
    // store the current element to be deleted
    let current = this.head;
    // point current head to new head
    this.head = this.head.next;
    // set previous of new head to null
    this.head.previous = null;

    this.length--;
  }
```

```
ll.insertAtStart(100);
dll.insertAtStart(500);
dll.insertAtStart(1000);

dll.insertAtIndex(50, 1);
dll.insertAtIndex(250, 3);

console.log();

dll.removeAtStart();

console.log();

dll.printLinkedList();

dll.size();
```

```
Node {
  data: 1000,
  previous: null,
  next: <ref *1> Node {
    data: 50,
    previous: null,
    next: Node { data: 500, previous: [Circular *1], next: [Node] }
  }
}

50
500
250
100
Length of Doubly Linked list is 4
```

//Modified removeAtStart()

```
removeAtStart() {
    if (!this.head) {
      return false;
    }
    // store the current element to be deleted
    let current = this.head;
    // point current head to new head
    this.head = this.head.next;
    // set previous of new head to null
    this.head.previous = null;

    // set the deleted element to null
    current = null;

    console.log(current);

    this.length--;
  }
```


null
50
500
250
100
Length of Doubly Linked list is 4

Deleting Node at index:

```
removeAt(index) {
    //   if index = 0 implement removeAtStart()
    if (index == 0) {
      this.removeAtStart();
    } else if (index < 0 || index >= this.length) {
      console.log('Array index out of bounds enter valid index');
    } else {
      let current = this.head;

      let count = 0;
    //   Traverse till count is index - 1
      while (count < index) {
        count++;
        current = current.next;
      }

    //   if previous element is not null
```

```
      if (current.previous != null) {
        current.previous.next = current.next;
        current.next.previous = current.previous;
      }

    //   set deleted current element to null
      current = null;
      this.length--;
    }
  }
```

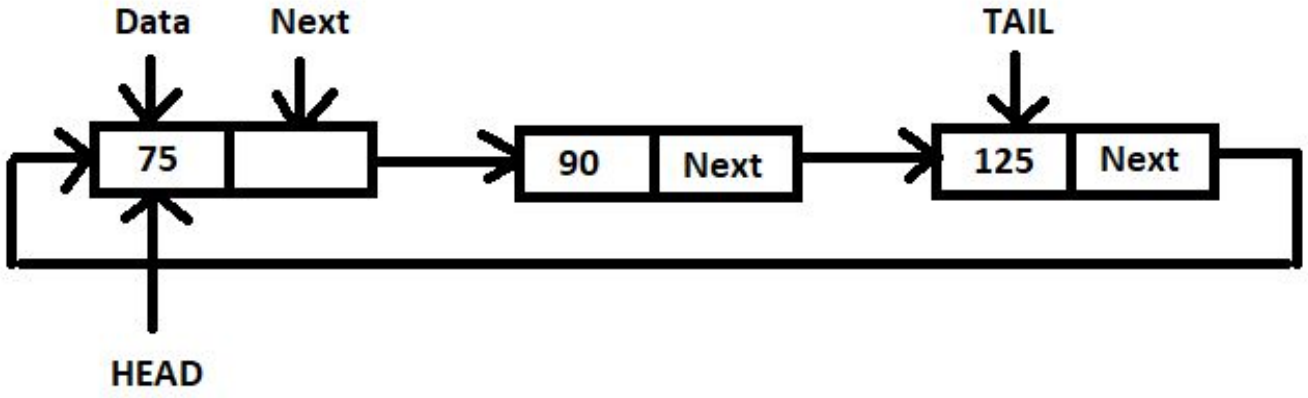
```
dll.insertAtStart(100);
dll.insertAtStart(500);
dll.insertAtStart(1000);

dll.insertAtIndex(50, 1);
dll.insertAtIndex(250, 3);

console.log();
```

```
dll.removeAt(2);
dll.removeAtStart();

console.log();

dll.printLinkedList();

dll.size();
```

```
50
250
100
Length of Doubly Linked list is 3
```

Circular Linked Lists:

Circular Linked Lists are special types of lists as the next field of a CLL is never null and the next field for the last node points to the head node.



Creating a Node:

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

Creating a CircularLinkedList class:

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

Adding Node:

Insert to empty list:

```
insertToEmpty(tail, data) {
    if (this.tail != null) {
      return this.tail;
    }

    let newNode = new Node(data);
    this.tail = newNode; // Update the head pointer to the new node
    this.tail.next = newNode;
    // console.log(this.tail);
    this.length++;

    return this.tail;
  }
```

Traversing or printing the elements:

```
printCircularLinkedList(tail) {
    let current;
    if (tail == null) {
      console.log('List is empty');
      return;
    }

    current = this.tail.next;
    do {
      console.log(current.data);
      // console.log(current);
      current = current.next;
    } while (current != this.tail.next);
  }
```

Size of CLL:

```
size() {
    console.log('Length of Circular Linked list is ' + this.length);
  }
```

//Implementation

```
let tail = null;

tail = cll.insertToEmpty(tail, 100);
cll.printCircularLinkedList(tail);

cll.size();
```

```
100
Length of Circular Linked list is 1
```

Insert node at the beginning:

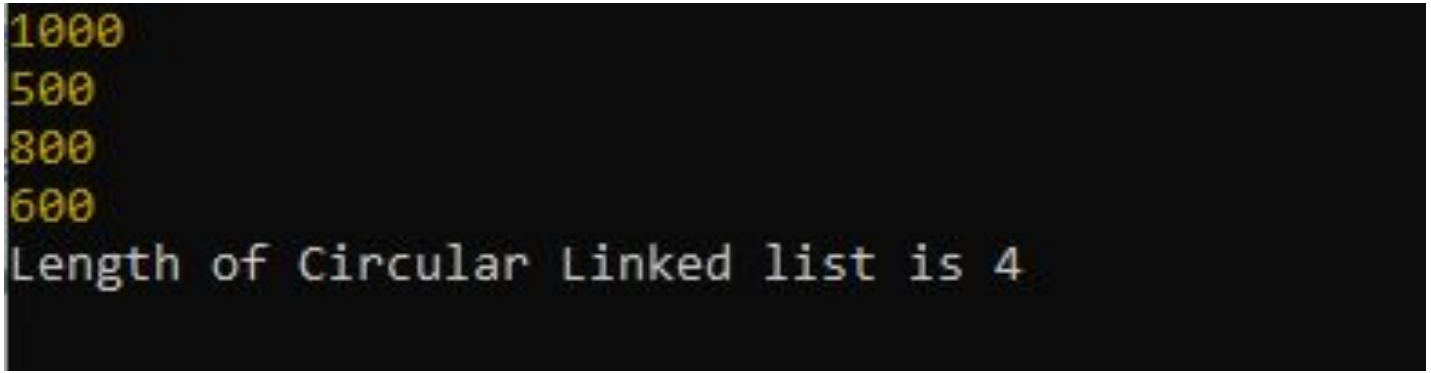
```
insertAtStart(data) {
    if (this.tail == null) {
      return this.insertToEmpty(this.tail, data);
    }
    let newNode = new Node(data); // Create a new node

      newNode.next = this.tail.next; // Set next node of new node to
current head
    this.tail.next = newNode;
    this.length++;
    return this.tail;
  }
```

```
tail = cll.insertAtStart(500);
tail = cll.insertAtStart(1000);
cll.printCircularLinkedList(tail);
cll.size();
```

Insert node at the end:

```
insertAtEnd(data) {
    if (this.tail == null) {
      return this.insertToEmpty(this.tail, data);
    }
    let newNode = new Node(data); // Create a new node

     newNode.next = this.tail.next; // Set next node of new node to
current head
    this.tail.next = newNode;
    this.tail = newNode;
    this.length++;
    return this.tail;
  }
```

```
tail = cll.insertAtStart(500);
tail = cll.insertAtStart(1000);
tail = cll.insertAtEnd(800);
tail = cll.insertAtEnd(600);
cll.printCircularLinkedList(tail);

cll.size();
```

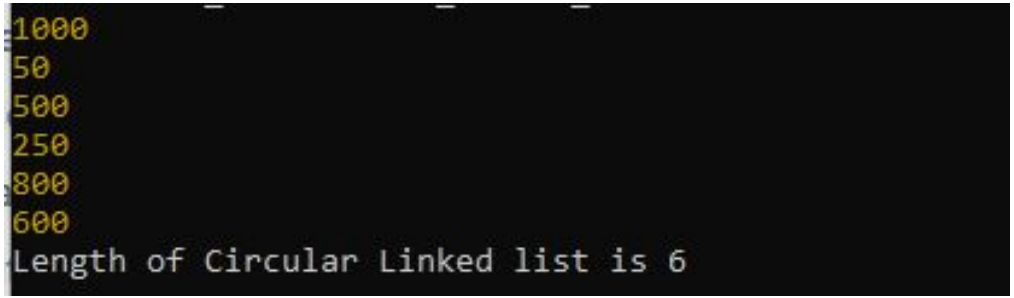


Insert node anywhere between start and end:

```
insertAtIndex(data, index) {
    // if first element
    if (index == 0) {
      this.insertAtStart(data);
    }
    // if out of range
    else if (index < 0 || index >= this.length) {
      console.log('Array index out of bounds enter valid index');
    } else {
      let newNode = new Node(data); //Create a new node
      let current, previous;
      current = this.tail.next; // maintain current node
      let count = 0;

      // traverse till index - 1 as linked list has 0 based indexing
      while (count < index) {
```

```
      while (count < index) {
        previous = current;
        current = current.next;
        count++;
      }

      newNode.next = current; //Set next of new node to current node
      previous.next = newNode; //Set previous of new node's next to new
node

      this.length++;
      return this.tail;
    }
  }
```

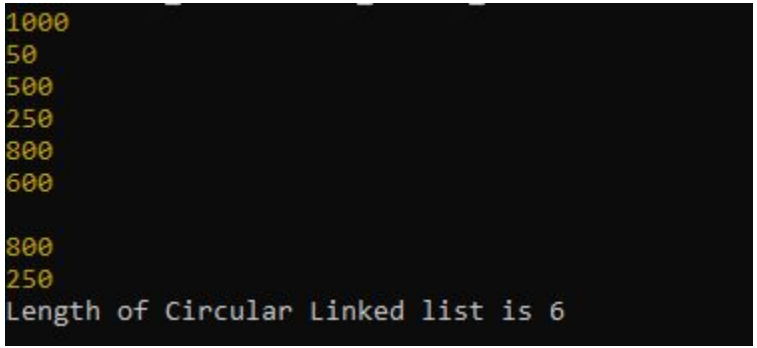
```
tail = cll.insertAtStart(500);
tail = cll.insertAtStart(1000);
tail = cll.insertAtEnd(800);
tail = cll.insertAtEnd(600);
tail = cll.insertAtIndex(50, 1);
tail = cll.insertAtIndex(250, 3);
cll.printCircularLinkedList(tail);

cll.size();
```



Getting data of element at index:

```
getElement(index) {
    if (index < 0 || index >= this.length) {
      console.log('Array index out of bounds enter valid index');
    } else {
      let current = this.tail.next;
      let count = 0;
      do {
        if (count == index && current != null) {
          console.log(current.data);
        }
        count++;
        current = current.next;
      } while (current != this.tail.next);
    }
  }
```

```
tail = cll.insertAtEnd(800);
tail = cll.insertAtEnd(600);
tail = cll.insertAtIndex(50, 1);
tail = cll.insertAtIndex(250, 3);
cll.printCircularLinkedList(tail);

console.log();
cll.getElement(4);
cll.getElement(3);

cll.size();
```

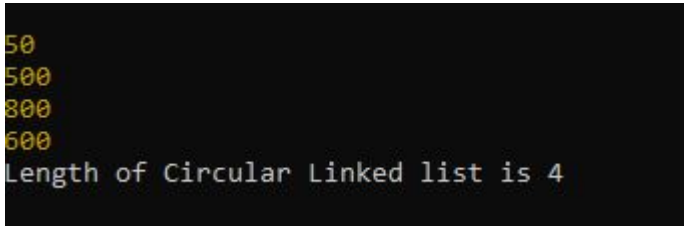


Deleting Node at index:

```
removeAt(index) {
    //   if index is 0
    if (index == 0) {
      if (!this.tail.next) {
        return false;
      }
      let current = this.tail.next;
      this.tail.next = current.next;
      current = null;

      this.length--;
    } else if (index < 0 || index >= this.length) {
      console.log('Array index out of bounds enter valid index');
    } else {
      let current, previous;
```

```
    //   current is set to head which is at tail.next
      current = this.tail.next;
      let count = 0;
      while (count < index) {
        count++;
        previous = current; // set previous to current
        current = current.next; // set current to next of current
      }

      previous.next = current.next; // set next of previous to next of
current
      current = null; // set current element to be deleted as null

      this.length--;
    }
  }
```

```
tail = cll.insertAtStart(500);
tail = cll.insertAtStart(1000);
tail = cll.insertAtEnd(800);
tail = cll.insertAtEnd(600);
tail = cll.insertAtIndex(50, 1);
tail = cll.insertAtIndex(250, 3);

console.log();

cll.removeAt(0);
cll.removeAt(2);

cll.printCircularLinkedList(tail);
cll.size();
```



Comparison of between Singly doubly and circular linked list

Type of LL	Singly LL	Doubly LL	Circular LL
Implementation	It contains data and next field. Pointer to next link is stored in next field. Head will point to start node and Tail will point to last node. Next of head will pont to next node and next of tail will point to null.	It contains data, previous and next field. Pointer to next link is stored in next field and pointer to previous node is tored in previous field. Previous of head will point to null and next of tail will point to null.	It can have singly linked list or doubly linked list implemenation. Major difference is that next node will never point to null. Next of tail will always point to head.
Traversal	Traversal - Singly linked list allows traversal elements only in one way.	Traversal - Doubly linked list allows element to traverse in backward and forward direction.	Traversal - Circular linked list allows element to traverse in forward direction being circular in nature it points to head after last node.
Memory	Singly linked list stores pointer of only one node so consumes lesser memory.	Doubly linked list stores two pointers so uses more memory per node.	Circular linked list uses memory based on its implementaion as singly circular linked list or doubly circular linked list.

Usage	Singly linked list are generally used for implementation of stacks	Doubly linked list can be used to implement stacks as well as heaps and binary trees.	Circular linked list used for implementation of queues.
Time complexity	In singly linked list the complexity of insertion and deletion at any position is O(n).	In doubly linked list the complexity of insertion and deletion at a known position is O(1) but at start and end is O(n).	In circular linked list the complexity of insertion and deletion at start and end position is O(1) but at any index other than start and end is O(n)
Performance	Singly linked list is preferred when we need to save memory and searching is not required.	For better performance while searching and memory is not a limitation doubly linked list is more preferred.	It has better performance than traditional implemenation of singly linked list or doubly linked list as tail and head can be accesed in O(1).

MCQs

1. What fields are associated with a doubly linked list?

A. next
B. data, next
C. data, previous, next
D. data, previous

1. What fields are associated with a doubly linked list?

A. next
B. data, next
C. data, previous, next
D. data, previous

Answer: C

2. What is the time complexity of inserting the element at index n of doubly linked list when previous or next is not known?

A. O(1)
B. O(log n)
C. O(n log(n))
D. O(n)

2. What is the time complexity of inserting the element at index n of doubly linked list when previous or next is not known?

A. O(1)
B. O(log n)
C. O(n log(n))
D. O(n)

Answer: D

3. Memory allocation is dynamic in Linked list.

A. True
B. False

3. Memory allocation is dynamic in Linked list.

A. True
B. False

Answer: A

4. Time complexity of insertion at last in a Circular linked list is?

A. O(1)
B. O(log n)
C. O(n log(n))
D. O(n)

4. Time complexity of insertion at last in a Circular linked list is?

A. O(1)
B. O(log n)
C. O(n log(n))
D. O(n)

Answer: A

5. Searching an element can be achieved with which of the below linked list efficiently?

A. Singly linked list
B. Doubly linked list and doubly circular linked list
C. Singly circular linked list
D. None of the above

5. Searching an element can be achieved with which of the below linked list efficiently?

A. Singly linked list
B. Doubly linked list and doubly circular linked list
C. Singly circular linked list
D. None of the above

Answer: B

Homework Problems:

1. Implement getHead(), isEmpty() methods for singly linked list.
2. Implement insertBefore(), insertAfter() and searchData() methods for doubly linked lists.

Thank You!