CHAPTER

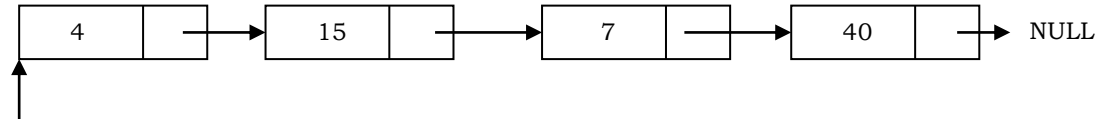
3

LINKED LISTS

3.1 What is a Linked List?

Linked list is a data structure used for storing collections of data. Linked list has the following properties.

- Successive elements are connected by pointers
- Last element points to NULL
- Can grow or shrink in size during execution of a program
- Can be made just as long as required (until systems memory exhausts)
- It does not waste memory space (but takes some extra memory for pointers)



3.2 Linked Lists ADT

The following operations make linked lists an ADT:

Main Linked Lists Operations

- Insert: inserts an element into the list
- Delete: removes and returns the specified position element from the list

Auxiliary Linked Lists Operations

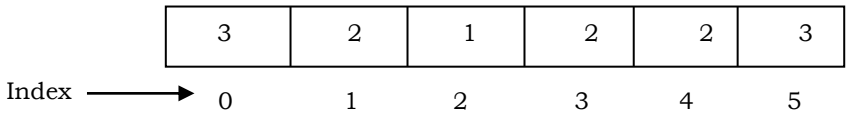
- Delete List: removes all elements of the list (disposes the list)
- Count: returns the number of elements in the list
- Find n^{th} node from the end of the list

3.3 Why Linked Lists?

There are many other data structures that do the same thing as that of linked lists. Before discussing linked lists it is important to understand the difference between linked lists and arrays. Both linked lists and arrays are used to store collections of data. Since both are used for the same purpose, we need to differentiate their usage. That means in which cases *arrays* are suitable and in which cases *linked lists* are suitable.

3.4 Arrays Overview

One memory block is allocated for the entire array to hold the elements of the array. The array elements can be accessed in a constant time by using the index of the particular element as the subscript.



Why Constant Time for Accessing Array Elements?

To access an array element, address of an element is computed as an offset from the base address of the array and one multiplication is needed to compute what is supposed to be added to the base address to get the memory address of the element. First the size of an element of that data type is calculated and then it is multiplied with the index of the element to get the value to be added to the base address. This process takes one multiplication and one addition. Since these two operations take constant time, we can say the array access can be performed in constant time.

Advantages of Arrays

- Simple and easy to use
- Faster access to the elements (constant access)

Disadvantages of Arrays

- **Fixed size:** The size of the array is static (specify the array size before using it).
- **One block allocation**: To allocate the array at the beginning itself, sometimes it may not be possible to get the memory for the complete array (if the array size is big).
- **Complex position-based insertion**: To insert an element at a given position we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning then the shifting operation is more expensive.

Dynamic Arrays

Dynamic array (also called as *growable array*, *resizable array*, *dynamic table*, or *array list*) is a random access, variable-size list data structure that allows elements to be added or removed.

One simple way of implementing dynamic arrays is, initially start with some fixed size array. As soon as that array becomes full, create the new array of size double than the original array. Similarly, reduce the array size to half if the elements in the array are less than half.

Note: We will see the implementation for *dynamic arrays* in the *Stacks*, *Queues* and *Hashing* chapters.

Advantages of Linked Lists

Linked lists have both advantages and disadvantages. The advantage of linked lists is that they can be *expanded* in constant time. To create an array we must allocate memory for a certain number of elements. To add more elements to the array then we must create a new array and copy the old array into the new array. This can take lot of time.

We can prevent this by allocating lots of space initially but then you might allocate more than you need and wasting memory. With a linked list we can start with space for just one element allocated and *add* on new elements easily without the need to do any copying and reallocating.

Issues with Linked Lists (Disadvantages)

There are a number of issues in linked lists. The main disadvantage of linked lists is *access time* to individual elements. Array is random-access, which means it takes O(1) to access any element in the array. Linked lists takes O(n) for access to an element in the list in the worst case. Another advantage of arrays in access time is *spacial locality* in memory. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods.

Although the dynamic allocation of storage is a great advantage, the *overhead* with storing and retrieving data can make a big difference. Sometimes linked lists are *hard* to *manipulate*. If the last item is deleted, the last but one must now have its pointer changed to hold a NULL reference. This requires that the list is traversed to find the last but one link, and its pointer set to a NULL reference.

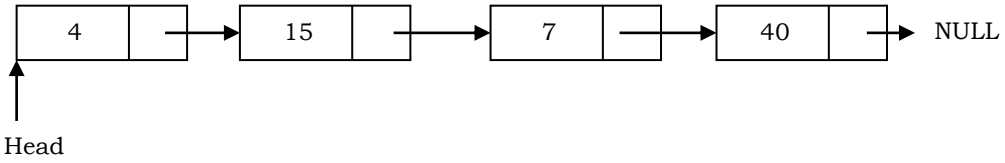
Finally, linked lists wastes memory in terms of extra reference points.

3.5 Comparison of Linked Lists with Arrays & Dynamic Arrays

Parameter	Linked list	Array	Dynamic array
Indexing	O(n)	O(1)	O(1)
Insertion/deletion at beginning	O(1)	O(n), if array is not full (for shifting the elements)	O(n)
Insertion at ending	O(n)	O(1), if array is not full	O(1), if array is not full O(n), if array is full
Deletion at ending	O(n)	O(1)	O(n)
Insertion in middle	O(n)	O(n), if array is not full (for shifting the elements)	O(n)
Deletion in middle	O(n)	O(n), if array is not full (for shifting the elements)	O(n)
Wasted space	O(n)	0	O(n)

3.6 Singly Linked Lists

Generally "linked list" means a singly linked list. This list consists of a number of nodes in which each node has a *next* pointer to the following element. The link of the last node in the list is NULL, which indicates end of the list.



Following is a type declaration for a linked list:

```
public class ListNode {
        private int data;
        private ListNode next;
        public ListNode(int data){
                this.data = data;
        }
        public void setData(int data){
                this.data = data;
        }
        public int getData(){
                return data;
        }
        public void setNext(ListNode next){
                this.next = next;
        }
        public ListNode getNext(){
                return this.next;
        }
}
```

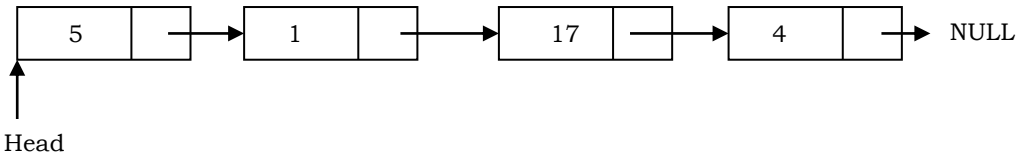
Basic Operations on a List

- Traversing the list
- Inserting an item in the list
- Deleting an item from the list

Traversing the Linked List

Let us assume that the *head* points to the first node of the list. To traverse the list we do the following.

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to NULL.



The ListLength() function takes a linked list as input and counts the number of nodes in the list. The function given below can be used for printing the list data with extra print function.

```
public int ListLength(ListNode headNode) {
        int length = 0;
        ListNode currentNode = headNode;
        while(currentNode != null){
                length++;
                currentNode = currentNode.getNext();
        }
        return length;
}
```

Time Complexity: O(n), for scanning the list of size n. Space Complexity: O(1), for creating temporary variable.

Singly Linked List Insertion

Insertion into a singly-linked list has three cases:

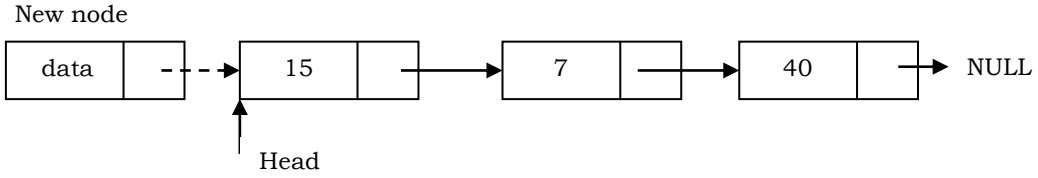
- Inserting a new node before the head (at the beginning)
- Inserting a new node after the tail (at the end of the list)
- Inserting a new node at the middle of the list (random location)

Note: To insert an element in the linked list at some position p, assume that after inserting the element the position of this new node is p.

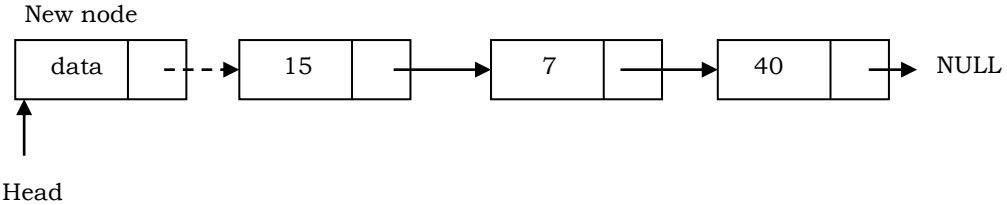
Inserting a Node in Singly Linked List at the Beginning

In this case, a new node is inserted before the current head node. *Only one next pointer* needs to be modified (new node's next pointer) and it can be done in two steps:

- Update the next pointer of new node, to point to the current head.



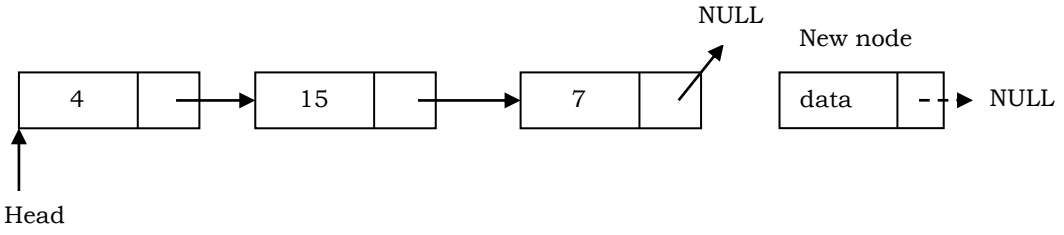
- Update head pointer to point to the new node.



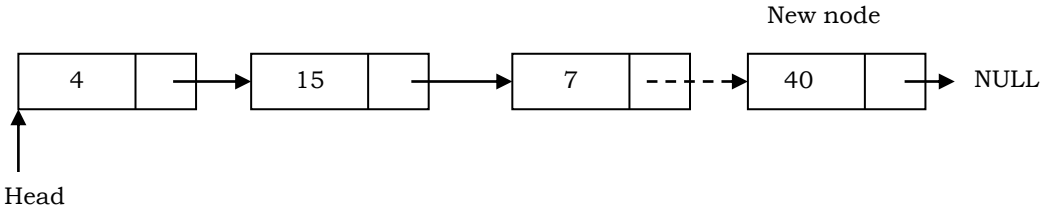
Inserting a Node in Singly Linked List at the Ending

In this case, we need to modify *two next pointers* (last nodes next pointer and new nodes next pointer).

- New nodes next pointer points to NULL.



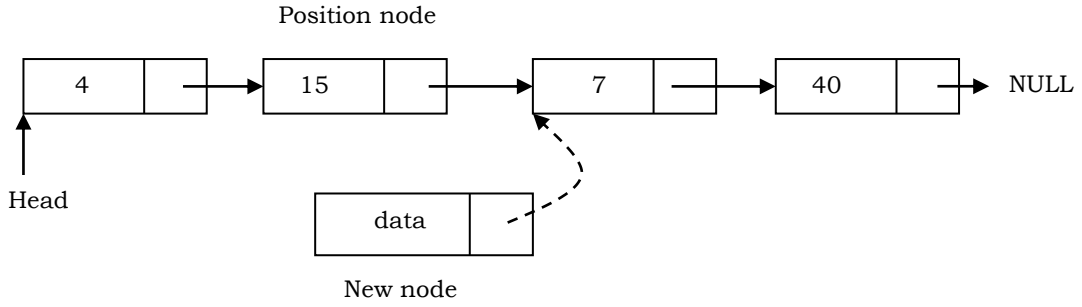
- Last nodes next pointer points to the new node.



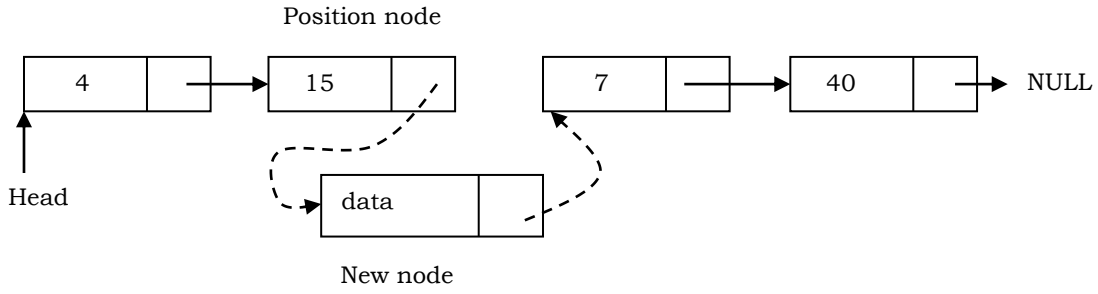
Inserting a Node in Singly Linked List at the Middle

Let us assume that we are given a position where we want to insert the new node. In this case also, we need to modify two next pointers.

- If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node. For simplicity let us assume that second node is called *position* node. New node points to the next node of the position where we want to add this node.



- Position nodes next pointer now points to the new node.



Note: We can implement the three variations of the *insert* operation separately.

Time Complexity: O(*n*). Since, in the worst we may need to insert the node at end of the list. Space Complexity: O(1), for creating one temporary variable.

Singly Linked List Deletion

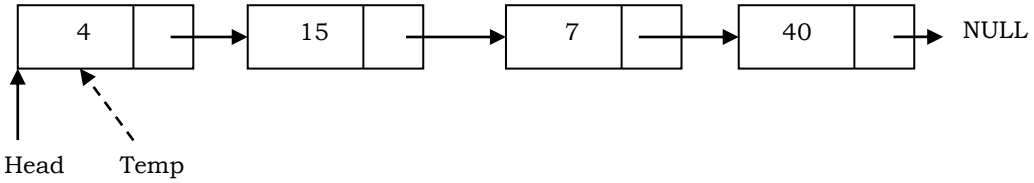
As similar to insertion here also we have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

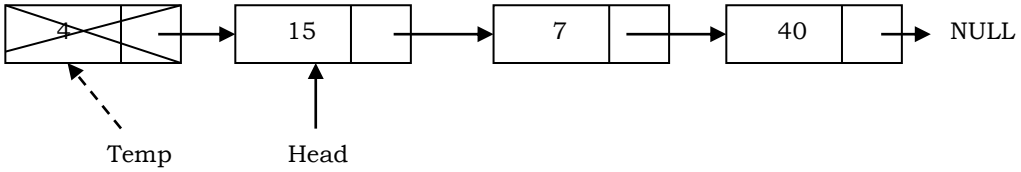
Deleting the First Node in Singly Linked List

First node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to same node as that of head.



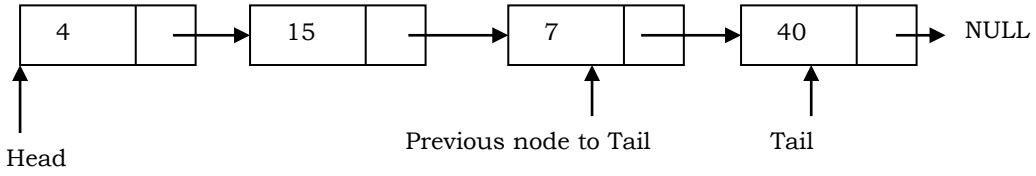
- Now, move the head nodes pointer to the next node and dispose the temporary node.



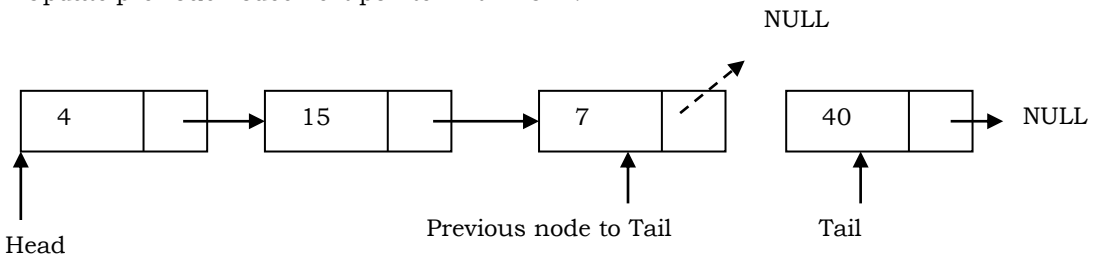
Deleting the Last node in Singly Linked List

In this case, last node is removed from the list. This operation is a bit trickier than removing the first node, because algorithm should find a node, which is previous to the tail first. It can be done in three steps:

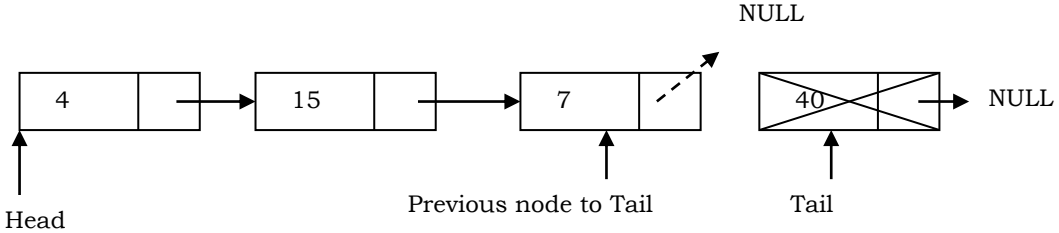
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of list, we will have two pointers one pointing to the *tail* node and other pointing to the node *before* tail node.



- Update previous nodes next pointer with NULL.



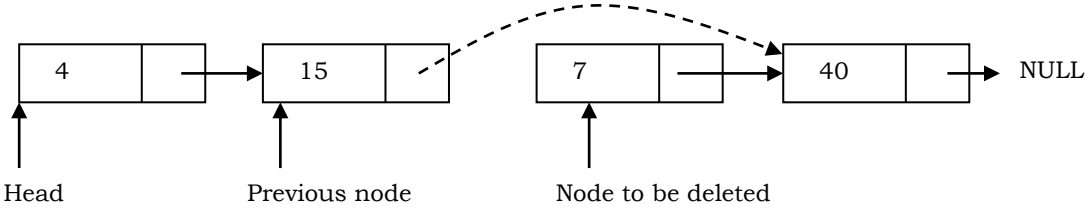
- Dispose the tail node.



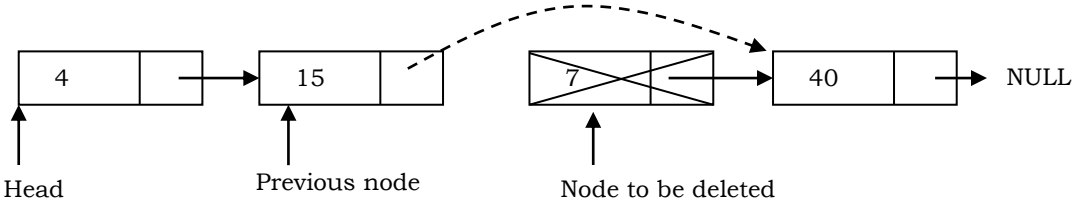
Deleting an Intermediate Node in Singly Linked List

In this case, node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- As similar to previous case, maintain previous node while traversing the list. Once we found the node to be deleted, change the previous nodes next pointer to next pointer of the node to be deleted.



- Dispose the current node to be deleted.



Time Complexity: O(n). In the worst we may need to delete the node at the end of the list. Space Complexity: O(1). Since, we are creating only one temporary variable.

Deleting Singly Linked List

This works by storing the current node in some temporary variable and freeing the current node. After freeing the current node go to next node with temporary variable and repeat this process for all nodes.

Time Complexity: O(n), for scanning the complete list of size n.
Space Complexity: O(1), for temporary variable.

Implementation

```
public class LinkedList {
        // This class has a default constructor:
        public LinkedList() {
                length = 0;
        }
        // This is the only field of the class.  It holds the head of the list
        ListNode head;
        // Length of the linked list
        private int length = 0;
        // Return the first node in the list
        public synchronized ListNode getHead() {
                return head;
        }
        // Insert a node at the beginning of the list
        public synchronized void insertAtBegin(ListNode node) {
                node.setNext(head);
                head = node;
                length ++;
        }
        // Insert a node at the end of the list
        public synchronized void insertAtEnd(ListNode node) {
                if (head == null)
                        head = node;
                else {
                        ListNode p, q;
                        for(p = head; (q = p.getNext()) != null; p = q);
                                p.setNext(node);
                }
```

```
            length ++;
        }
        // Add a new value to the list at a given position.
        // All values at that position to the end move over to make room.
        public void insert(int data, int position) {
            // fix the position
            if (position < 0) {
                position = 0;
            }
            if (position > length) {
                position = length;
            }
            // if the list is empty, make it be the only element
            if (head == null) {
                head = new ListNode(data);
            }
            // if adding at the front of the list...
            else if (position == 0) {
                ListNode temp = new ListNode(data);
                temp.next = head;
                head = temp;
            }
            // else find the correct position and insert
            else {
                ListNode temp = head;
                for (int i=1; i<position; i+=1) {
                    temp = temp.getNext();
                }
                ListNode newNode = new ListNode(data);
                newNode.next = temp.next;
                temp.setNext(newNode);
            }
            // the list is now one value longer
            length += 1;
        }
        // Remove and return the node at the head of the list
        public synchronized ListNode removeFromBegin() {
            ListNode node = head;
            if (node != null) {
                head = node.getNext();
                node.setNext(null);
            }
            return node;
        }
        // Remove and return the node at the end of the list
        public synchronized ListNode removeFromEnd() {
            if (head == null)
                return null;
            ListNode p = head, q = null, next = head.getNext();
            if (next == null) {
                head = null;
                return p;
            }
            while((next = p.getNext()) != null) {
                q = p;
                p = next;
            }
            q.setNext(null);
            return p;
        }
        // Remove a node matching the specified node from the list.
        // Use equals() instead of == to test for a matched node.
```

```
	public synchronized void removeMatched(ListNode node) {
		if (head == null)
			return;
		if (node.equals(head)) {
			head = head.getNext();
			return;
		}
		ListNode p = head, q = null;
		while((q = p.getNext()) != null) {
			if (node.equals(q)) {
				p.setNext(q.getNext());
				return;
			}
			p = q;
		}
	}
	// Remove the value at a given position.
	// If the position is less than 0, remove the value at position 0.
	// If the position is greater than 0, remove the value at the last position.
	public void remove(int position) {
		// fix position
		if (position < 0) {
			position = 0;
		}
		if (position >= length) {
			position = length-1;
		}
		// if nothing in the list, do nothing
		if (head == null)
			return;
		// if removing the head element...
		if (position == 0) {
			head = head.getNext();
		}
		// else advance to the correct position and remove
		else {
			ListNode temp = head;
			for (int i=1; i<position; i+=1) {
				temp = temp.getNext();
			}
			temp.setNext(temp.getNext().getNext());
		}
		// reduce the length of the list
		length -= 1;
	}
	// Return a string representation of this collection, in the form ["str1","str2",...].
	public String toString() {
		String result = "[";
		if (head == null) {
			return result+"]";
		}
		result = result + head.getData();
		ListNode temp = head.getNext();;
		while (temp != null) {
			result = result + "," + temp.getData();
			temp = temp.getNext();
		}
		return result + "]";
	}
	// Return the current length of the list.
```

```
        public int length() {
                return length;
        }
        // Find the position of the first value that is equal to a given value.
        // The equals method us used to determine equality.
        public int getPosition(int data) {
                // go looking for the data
                ListNode temp = head;
                int pos = 0;
                while (temp != null) {
                        if (temp.getData() == data) {
                                // return the position if found
                                return pos;
                        }
                        pos += 1;
                        temp = temp.getNext();
                }
                // else return some large value
                return Integer.MIN_VALUE;
        }
        // Remove everything from the list.
        public void clearList(){
                head = null;
                length = 0;
        }
}
```

3.7 Doubly Linked Lists

The *advantage* of a doubly linked list (also called *two – way linked list*) is that given a node in the list, we can navigate in both directions. A node in a singly linked list cannot be removed unless we have the pointer to its predecessor. But in doubly linked list we can delete a node even if we don't have previous nodes address (since, each node has left pointer pointing to previous node and we can move backward).

The primary *disadvantages* of doubly linked lists are:

- Each node requires an extra pointer, requiring more space.
- The insertion or deletion of a node takes a bit longer (more pointer operations).

As similar to singly linked list, let us implement the operations of doubly linked lists. Following is a type declaration for a doubly linked list:

```
public class DLLNode {
        private int data;
        private DLLNode prev;
        private DLLNode next;
        public DLLNode(int data) {
                this.data = data;
                prev = null;
                next = null;
        }
        public DLLNode(int data, DLLNode prev, DLLNode next) {
                this.data = data;
                this.prev = prev;
                this.next = next;
        }
        public int getData() {
                return data;
        }
        public void setData(int data) {
                this.data = data;
```

```
        }
        public DLLNode getPrev() {
                return prev;
        }
        public DLLNode getNext() {
                return next;
        }
        public void setPrev(DLLNode where) {
                prev = where;
        }
        public void setNext(DLLNode where) {
                next = where;
        }
}
```

Doubly Linked List Insertion

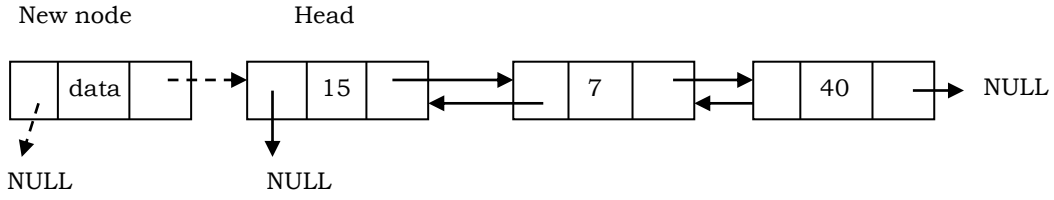
Insertion into a doubly-linked list has three cases (same as singly linked list).

- Inserting a new node before the head.
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node at the middle of the list.

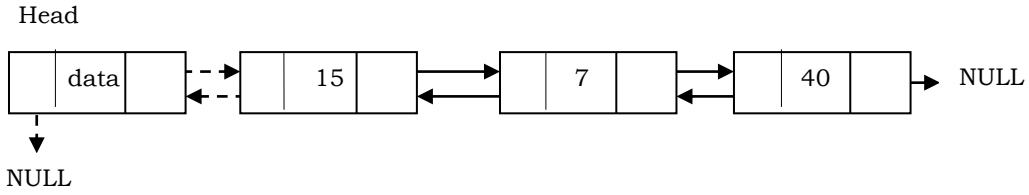
Inserting a Node in Doubly Linked List at the Beginning

In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps:

- Update the right pointer of new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.



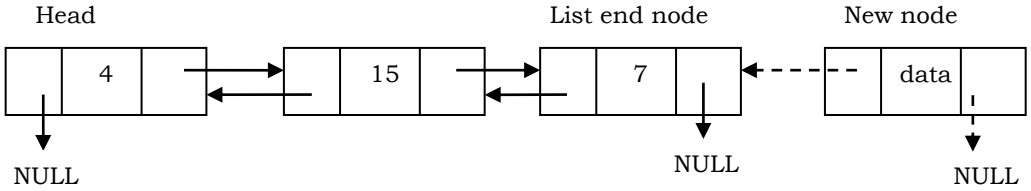
- Update head nodes left pointer to point to the new node and make new node as head.



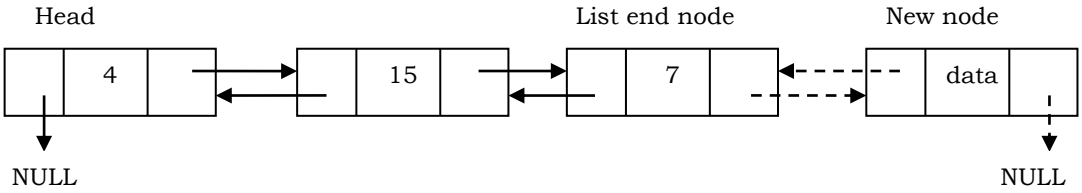
Inserting a Node in Doubly Linked List at the Ending

In this case, traverse the list till the end and insert the new node.

- New node right pointer points to NULL and left pointer points to the end of the list.



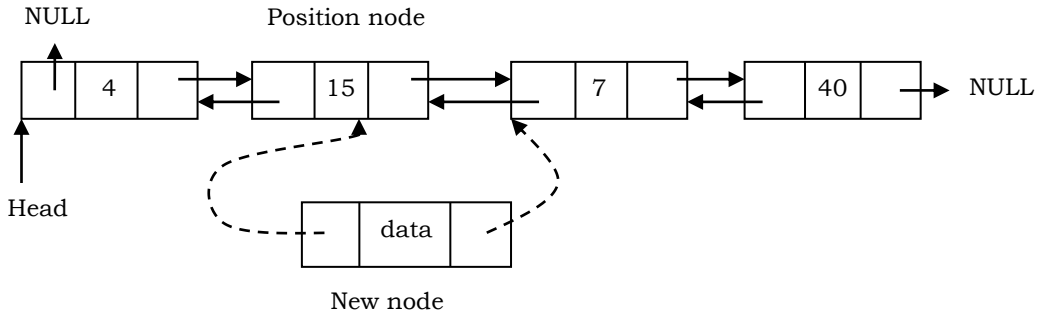
- Update right of pointer of last node to point to new node.



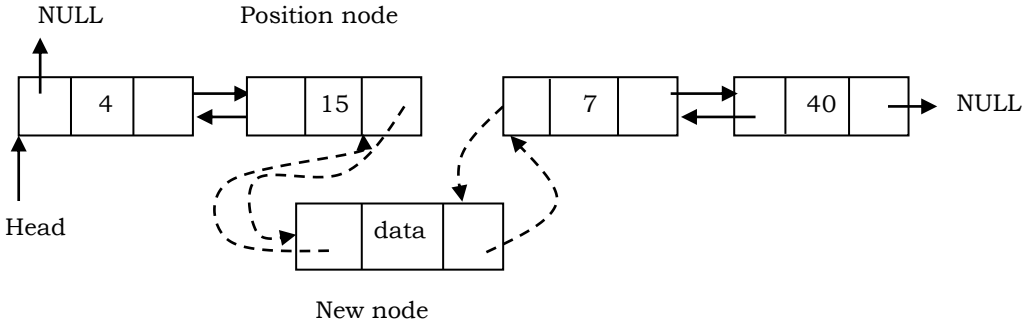
Inserting a Node in Doubly Linked List in the Middle

As discussed in singly linked lists, traverse the list till the position node and insert the new node.

- *New node* right pointer points to the next node of the *position node* where we want to insert the new node. Also, *new node* left pointer points to the *position node*.



- Position node right pointer points to the new node and the *next node* of position nodes left pointer points to new node.



Time Complexity: O(*n*). In the worst we may need to insert the node at the end of the list.
Space Complexity: O(1), for creating one temporary variable.

Doubly Linked List Deletion

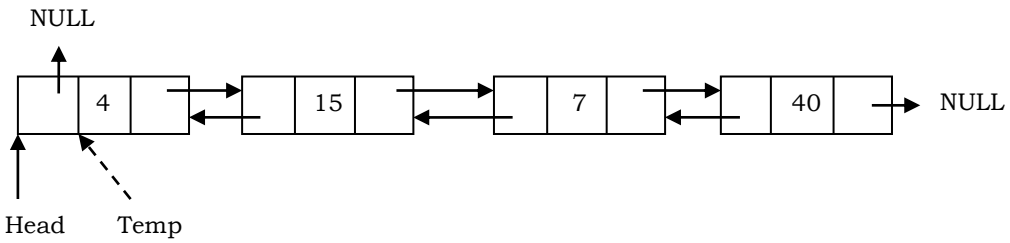
As similar to singly linked list deletion, here also we have three cases:

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

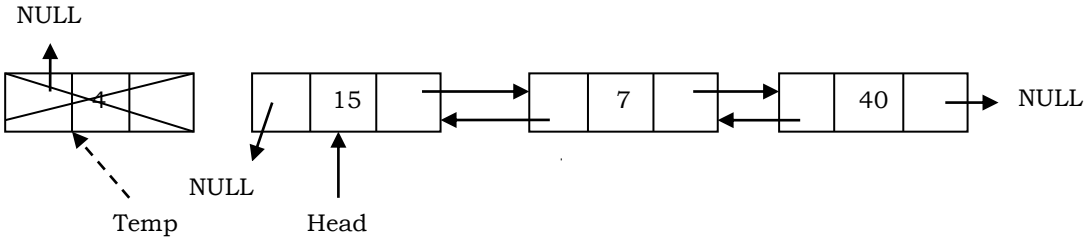
Deleting the First Node in Doubly Linked List

In this case, first node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to same node as that of head.



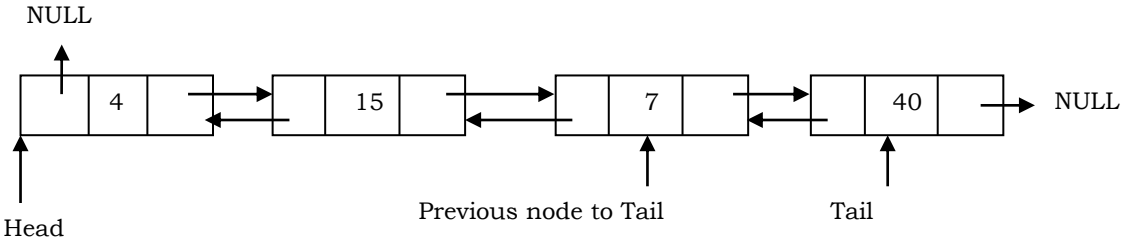
- Now, move the head nodes pointer to the next node and change the heads left pointer to NULL and dispose the temporary node.



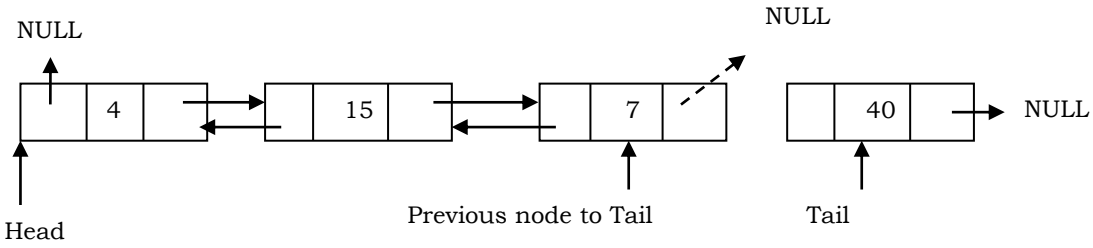
Deleting the Last Node in Doubly Linked List

This operation is a bit trickier, than removing the first node, because algorithm should find a node, which is previous to the tail first. This can be done in three steps:

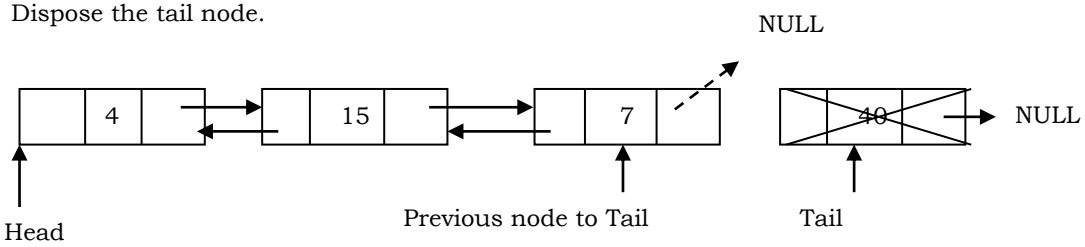
- Traverse the list and while traversing maintain the previous node address. By the time we reach the end of list, we will have two pointers one pointing to the tail and other pointing to the node before tail node.



- Update tail nodes previous nodes next pointer with NULL.



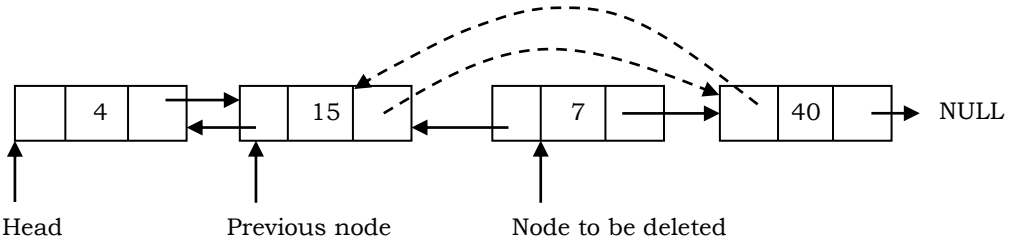
- Dispose the tail node.



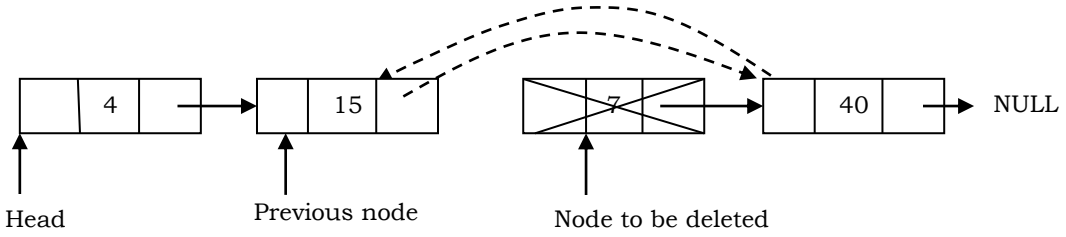
Deleting an Intermediate Node in Doubly Linked List

In this case, node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- Similar to previous case, maintain previous node also while traversing the list. Once we found the node to be deleted, change the previous nodes next pointer to the next node of the node to be deleted.



- Dispose the current node to be deleted.



Time Complexity: O(n), for scanning the complete list of size n.
Space Complexity: O(1), for creating one temporary variable.

Implementation

```
public class DoublyLinkedList{
	// properties
	private DLLNode head;
	private DLLNode tail;
	private int length;
	// Create a new empty list.
	public DoublyLinkedList() {
		head = new DLLNode(Integer.MIN_VALUE,null,null);
		tail = new DLLNode(Integer.MIN_VALUE, head, null);
		head.setNext(tail);
		length = 0;
	}
	// Get the value at a given position.
	public int get(int position) {
		return Integer.MIN_VALUE;
	}
	// Find the position of the head value that is equal to a given value.
	// The equals method us used to determine equality.
	public int getPosition(int data) {
		// go looking for the data
		DLLNode temp = head;
		int pos = 0;
		while (temp != null) {
			if (temp.getData() == data) {
				// return the position if found
				return pos;
			}
			pos += 1;
			temp = temp.getNext();
		}
		// else return some large value
		return Integer.MIN_VALUE;
	}
	// Return the current length of the DLL.
	public int length() {
		return length;
	}
	// Add a new value to the front of the list.
	public void insert(int newValue) {
		DLLNode newNode = new DLLNode(newValue,head,head.getNext());
		newNode.getNext().setPrev(newNode);
		head.setNext(newNode);
		length += 1;
	}
	// Add a new value to the list at a given position.
	// All values at that position to the end move over to make room.
```

```
	public void insert(int data, int position) {
		// fix the position
		if (position < 0) {
			position = 0;
		}
		if (position > length) {
			position = length;
		}
		// if the list is empty, make it be the only element
		if (head == null) {
			head = new DLLNode(data);
			tail = head;
		}
		// if adding at the front of the list...
		else if (position == 0) {
			DLLNode temp = new DLLNode(data);
			temp.next = head;
			head = temp;
		}
		// else find the correct position and insert
		else {
			DLLNode temp = head;
			for (int i=1; i<position; i+=1) {
				temp = temp.getNext();
			}
			DLLNode newNode = new DLLNode(data);
			newNode.next = temp.next;
			newNode.prev = temp;
			newNode.next.prev = newNode;
			temp.next = newNode;
		}
		// the list is now one value longer
		length += 1;
	}
	// Add a new value to the rear of the list.
	public void insertTail(int newValue) {
		DLLNode newNode = new DLLNode(newValue,tail.getPrev(),tail);
		newNode.getPrev().setNext(newNode);
		tail.setPrev(newNode);
		length += 1;
	}
	// Remove the value at a given position.
	// If the position is less than 0, remove the value at position 0.
	// If the position is greater than 0, remove the value at the last position.
	public void remove(int position) {
		// fix position
		if (position < 0) {
			position = 0;
		}
		if (position >= length) {
			position = length-1;
		}
		// if nothing in the list, do nothing
		if (head == null)
			return;
		// if removing the head element...
		if (position == 0) {
			head = head.getNext();
			if (head == null)
				tail = null;
		}
```

```
            // else advance to the correct position and remove
            else {
                        DLLNode temp = head;
                        for (int i=1; i<position; i+=1) {
                                    temp = temp.getNext();
                        }
                        temp.getNext().setPrev(temp.getPrev());
                        temp.getPrev().setNext(temp.getNext());
            }
            // reduce the length of the list
            length -= 1;
      }
      // Remove a node matching the specified node from the list.
      // Use equals() instead of == to test for a matched node.
      public synchronized void removeMatched(DLLNode node) {
            if (head == null)
                        return;
            if (node.equals(head)) {
                        head = head.getNext();
                        if (head == null)
                                    tail = null;
                        return;
            }
            DLLNode p = head;
            while(p != null) {
                        if (node.equals(p)) {
                                    p.prev.next = p.next;
                                    p.next.prev = p.prev;
                                    return;
                        }
            }
      }
      // Remove the head value from the list. If the list is empty, do nothing.
      public int removeHead() {
            if (length == 0)
               return Integer.MIN_VALUE;
            DLLNode save = head.getNext();
            head.setNext(save.getNext());
            save.getNext().setPrev(head);
            length -= 1;
            return save.getData();
      }
      // Remove the tail value from the list. If the list is empty, do nothing.
      public int removeTail() {
            if (length == 0)
                        return Integer.MIN_VALUE;
            DLLNode save = tail.getPrev();
            tail.setPrev(save.getPrev());
            save.getPrev().setNext(tail);
            length -= 1;
            return save.getData();
      }
      // Return a string representation of this collection, in the form: ["str1","str2",...].
      public String toString() {
            String result = "[]";
            if (length == 0)
                        return result;
            result = "[" + head.getNext().getData();
            DLLNode temp = head.getNext().getNext();
            while (temp != tail) {
                        result += "," + temp.getData();
                        temp =  temp.getNext();
```

```
                }
                return result + "]";
        }
        // Remove everything from the DLL.
        public void clearList(){
                head = null;
                tail = null;
                length = 0;
        }
}
```

3.8 Circular Linked Lists

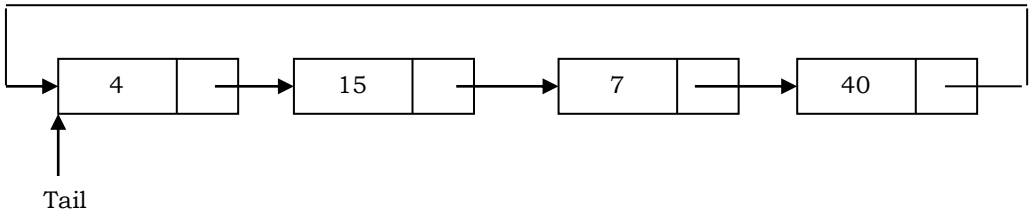
In singly linked lists and doubly linked lists the end of lists are indicated with NULL value. But circular linked lists do not have ends. While traversing the circular linked lists we should be careful otherwise we will be traversing the list infinitely. In circular linked lists each node has a successor.

Note that unlike singly linked lists, there is no node with NULL pointer in a circularly linked list. In some situations, circular linked lists are useful. For example, when several processes are using the same computer resource (CPU) for the same amount of time, we have to assure that no process accesses the resource before all other processes did (round robin algorithm).

In circular linked list we access the elements using the *head* node (similar to *head* node in singly linked list and doubly linked lists). For readability let us assume that the class name of circular linked list is CLLNode.

Counting Nodes in a Circular List

The circular list is accessible through the node marked *head*. To count the nodes, the list has to be traversed from node marked *head*, with the help of a dummy node *current* and stop the counting when *current* reaches the starting node *head*. If the list is empty, *head* will be NULL, and in that case set *count* = 0. Otherwise, set the current pointer to the first node, and keep on counting till the current pointer reaches the starting node.

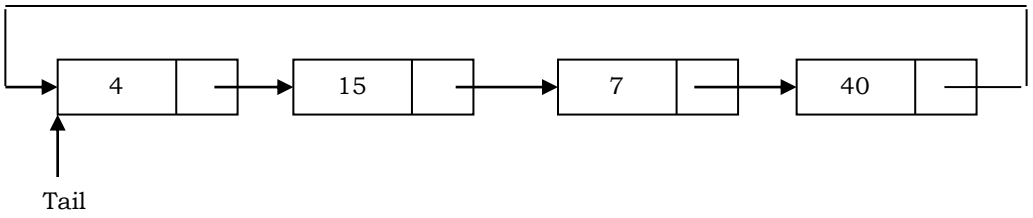



```
public int CircularListLength(CLLNode tail){
        int length = 0;
        CLLNode currentNode = tail.getNext();
        while(currentNode != tail){
                length++;
                currentNode = currentNode.getNext();
        }
        return length;
}
```

Time Complexity: O(n), for scanning the complete list of size n.
Space Complexity: O(1), for creating one temporary variable.

Printing the contents of a circular list

We assume here that the list is being accessed by its *head* node. Since all the nodes are arranged in a circular fashion, the *tail* node of the list will be the node previous to the *head* node. Let us assume we want to print the contents of the nodes starting with the *head* node. Print its contents, move to the next node and continue printing till we reach the *head* node again.



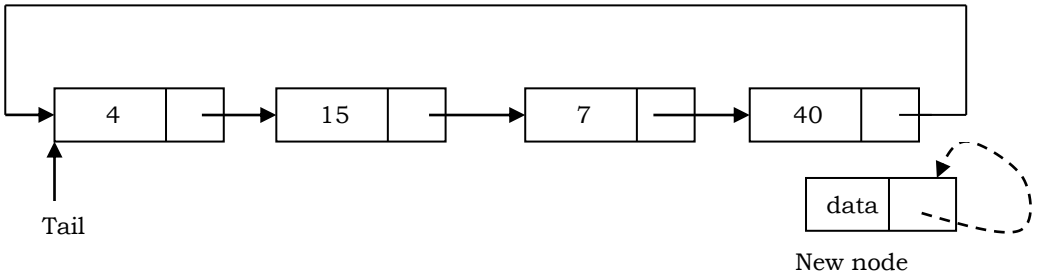

```
public void PrintCircularListData(CLLNode tail){
        CLLNode CLLNode = tail.getNext();
        while(CLLNode != tail){
                System.out.print(CLLNode.getData()+"->");
                CLLNode = CLLNode.getNext();
        }
        System.out.println("(" + CLLNode.getData() + ")headNode");
}
```

Time Complexity: O(n), for scanning the complete list of size n.
Space Complexity: O(1), for creating ne temporary variable.

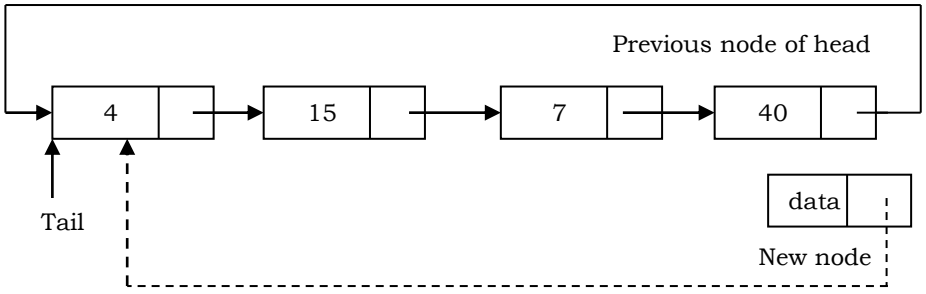
Inserting a Node at the End of a Circular Linked List

Let us add a node containing *data*, at the end of a list (circular list) headed by $head$. The new node will be placed just after the tail node (which is the last node of the list), which means it will have to be inserted in between the tail node and the first node.

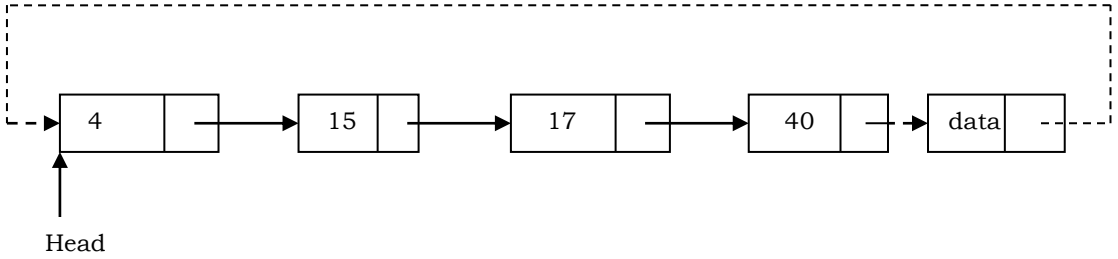
- Create a new node and initially keep its next pointer points to itself.



- Update the next pointer of new node with head node and also traverse the list until the tail. That means in circular list we should stop at a node whose next node is head.



- Update the next pointer of previous node to point to new node and we get the list as shown below.

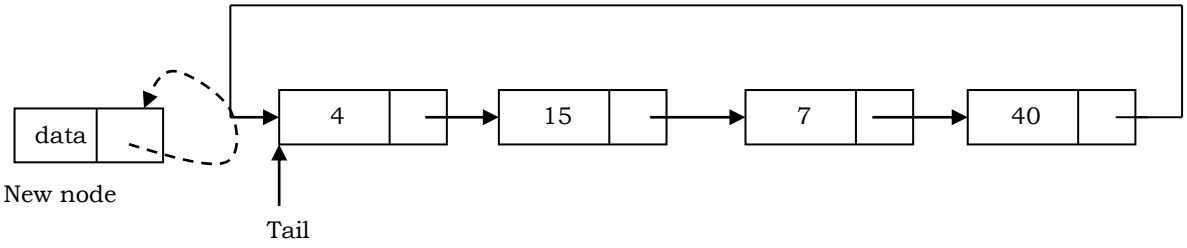


Time Complexity: O(n), for scanning the complete list of size n.
Space Complexity: O(1), for creating one temporary variable.

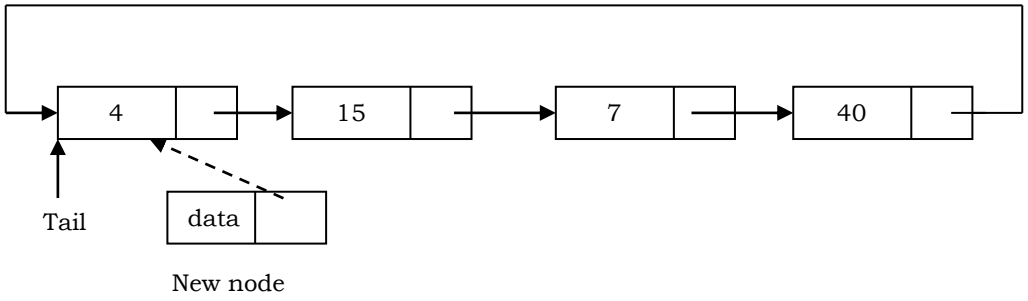
Inserting a Node at Front of a Circular Linked List

The only difference between inserting a node at the beginning and at the ending is that, after inserting the new node we just need to update the pointer. The steps for doing this is given below:

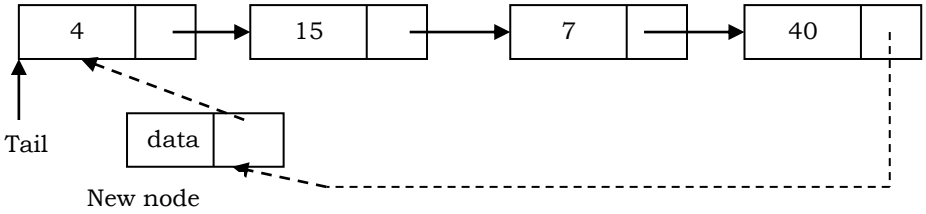
- Create a new node and initially keep its next pointer points to itself.



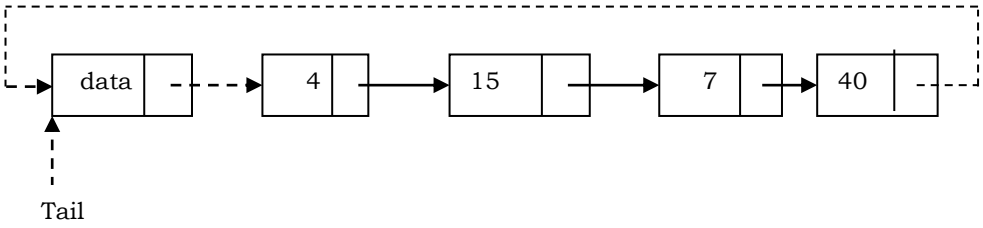
- Update the next pointer of new node with head node and also traverse the list until the tail. That means in circular list we should stop at the node which is its previous node in the list.



- Update the previous node of head in the list to point to new node.



- Make new node as head.

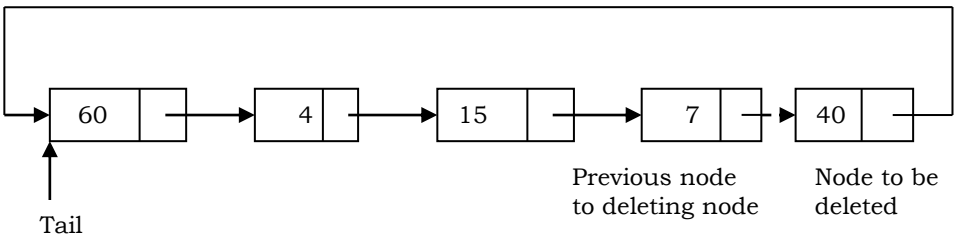


Time Complexity: O(n), for scanning the complete list of size n.
Space Complexity: O(1), for creating only one temporary variable.

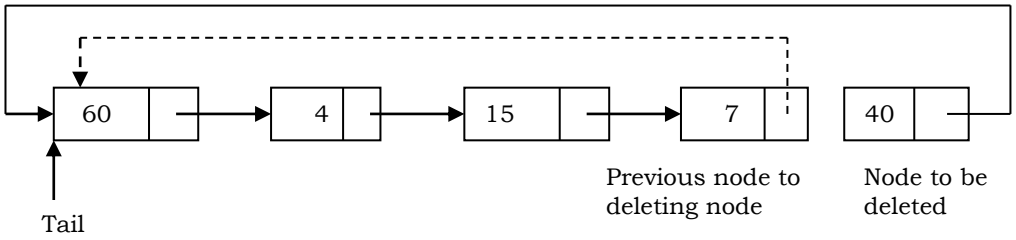
Deleting the Last Node in a Circular List

The list has to be traversed to reach the last but one node. This has to be named as the tail node, and its next field has to point to the first node. Consider the following list. To delete the last node 40, the list has to be traversed till you reach 7. The next field of 7 has to be changed to point to 60, and this node must be renamed $pTail$.

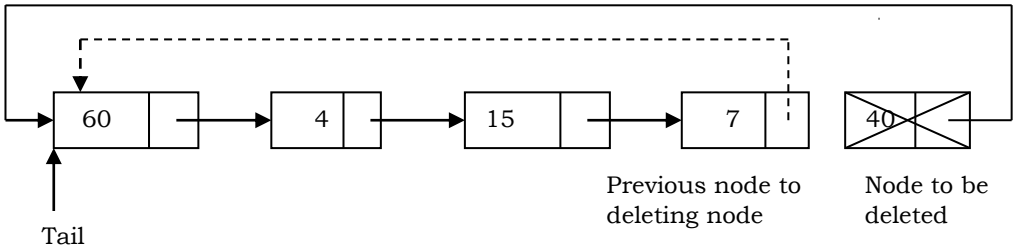
- Traverse the list and find the tail node and its previous node.



- Update the tail nodes previous node next pointer to point to head.



- Dispose the tail node.

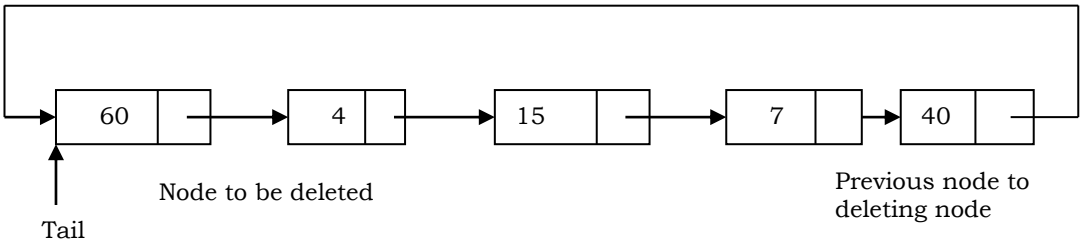


Time Complexity: O(n), for scanning the complete list of size n.
Space Complexity: O(1), for creating one temporary variable.

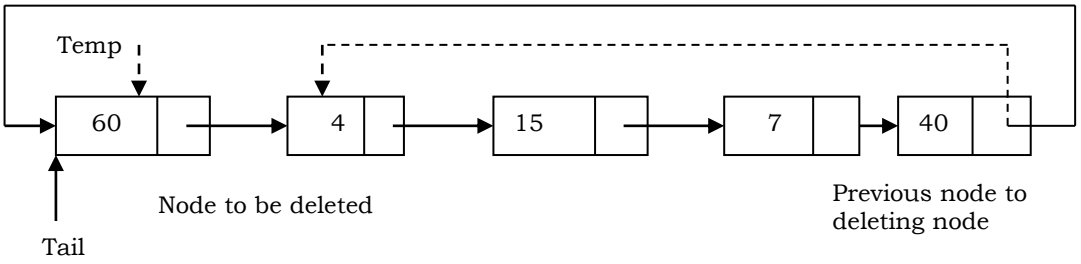
Deleting the First Node in a Circular List

The first node can be deleted by simply replacing the next field of tail node with the next field of the first node.

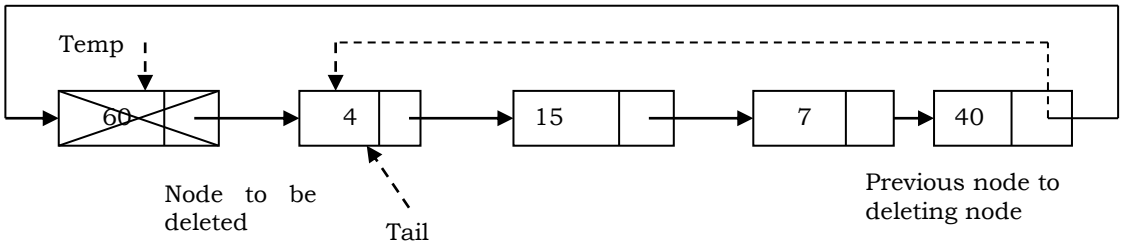
- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.



- Create a temporary node which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



- Now, move the head pointer to next node. Create a temporary node which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).



Time Complexity: O(n), for scanning complete list of size n.
Space Complexity: O(1), for creating temporary variable.

Applications of Circular List

Circular linked lists are used in managing the computing resources of a computer. We can use circular lists for implementing stacks and queues.

Implementation

```
public class CircularLinkedList{
        protected CLLNode tail;
        protected int length;
        // Constructs a new circular list
        public CircularLinkedList(){
                tail = null;
                length = 0;
        }
        // Adds data to beginning of list.
        public void add(int data){
                addToHead(data);
        }
        // Adds element to head of list
        public void addToHead(int data){
                CLLNode temp = new CLLNode(data);
                if (tail == null) { // first data added
                    tail = temp;
                    tail.setNext(tail);
                } else { // element exists in list
                    temp.setNext(tail.getNext());
                    tail.setNext(temp);
                }
                length++;
        }
        // Adds element to tail of list
        public void addToTail(int data){
                // new entry:
                addToHead(data);
                tail = tail.getNext();
        }
        // Returns data at head of list
        public int peek(){
                return tail.getNext().getData();
        }
        // Returns data at tail of list
        public int tailPeek(){
                return tail.getData();
        }
        // Returns and removes data from head of list
        public int removeFromHead(){
                CLLNode temp = tail.getNext(); // ie. the head of the list
```

```
		if (tail == tail.getNext()) {
			tail = null;
		} else {
			tail.setNext(temp.getNext());
			temp.setNext(null); // helps clean things up; temp is free
		}
		length--;
		return temp.getData();
	}
	// Returns and removes data from tail of list
	public int removeFromTail(){
		if (isEmpty()){
			return Integer.MIN_VALUE;
		}
		CLLNode finger = tail;
		while (finger.getNext() != tail) {
			finger = finger.getNext();
		}
		// finger now points to second-to-last data
		CLLNode temp = tail;
		if (finger == tail) {
			tail = null;
		} else {
			finger.setNext(tail.getNext());
			tail = finger;
		}
		length--;
		return temp.getData();
	}
	// Returns true if list contains data, else false
	public boolean contains(int data){
		if (tail == null) return false;
		CLLNode finger;
		finger = tail.getNext();
		while (finger != tail && (!(finger.getData() == data))){
			finger = finger.getNext();
		}
		return finger.getData() == data;
	}
	// Removes and returns element equal to data, or null
	public int remove(int data){
		if (tail == null) return Integer.MIN_VALUE;
		CLLNode finger = tail.getNext();
		CLLNode previous = tail;
		int compares;
		for (compares = 0; compares < length && (!(finger.getData() == data)); compares++) {
			previous = finger;
			finger = finger.getNext();
		}
		if (finger.getData() == data) {
			// an example of the pigeon-hole principle
			if (tail == tail.getNext()) {
				tail = null; }
			else {
				if (finger == tail)
					tail = previous;
				previous.setNext(previous.getNext().getNext());
			}
			// finger data free
			finger.setNext(null);	// to keep things disconnected
			length--;			// fewer elements
			return finger.getData();
		}
```

```
                else return Integer.MIN_VALUE;
        }
        // Return the current length of the CLL.
        public int size(){
                return length;
        }
        // Return the current length of the CLL.
        public int length() {
                return length;
        }
        // Returns true if no elements in list
        public boolean isEmpty(){
                return tail == null;
        }
        // Remove everything from the CLL.
        public void clear(){
                length = 0;
                tail = null;
        }
        // Return a string representation of this collection, in the form: ["str1","str2",...].
        public String toString(){
                String result = "[";
                if (tail == null) {
                        return result+"]";
                }
                result = result + tail.getData();
                CLLNode temp = tail.getNext();
                while (temp != tail) {
                        result = result + "," + temp.getData();
                        temp = temp.getNext();
                }
                return result + "]";
        }
}
```

3.9 A Memory-Efficient Doubly Linked List

In conventional implementation, we need to keep a forward pointer to the next item on the list and a backward pointer to the previous item. That means, elements in doubly linked list implementations consist of data, a pointer to the next node and a pointer to the previous node in the list as shown below.

Conventional Node Definition

```
public class DLLNode {
        private int data;
        private DLLNode next;
        private DLLNode previous;
        ……..
}
```

Recently a journal (Sinha) presented an alternative implementation of the doubly linked list ADT, with insertion, traversal and deletion operations. This implementation is based on pointer difference. Each node uses only one pointer field to traverse the list back and forth.

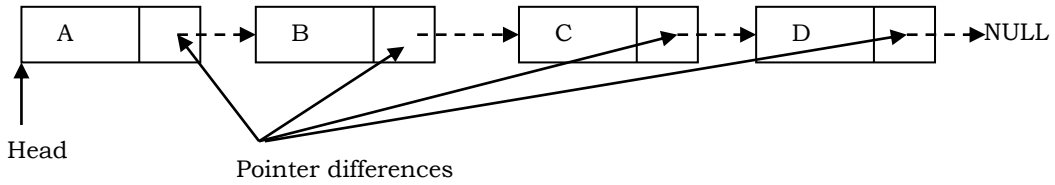
New Node Definition

```
public class ListNode {
        private int data;
        private ListNode ptrdiff;
        ……..
}
```

The *ptrdiff* pointer field contains the difference between the pointer to the next node and the pointer to the previous node. The pointer difference is calculated by using exclusive-or (⊕) operation.

$$ptrdiff = pointer\ to\ previous\ node \oplus pointer\ to\ next\ node.$$

The *ptrdiff* of the start node (head node) is the ⊕ of NULL and *next* node (next node to head). Similarly, the *ptrdiff* of end node is the ⊕ of *previous* node (previous to end node) and NULL. As an example, consider the following linked list.



In the example above,

- The next pointer of A is: NULL ⊕ B
- The next pointer of B is: A ⊕ C
- The next pointer of C is: B ⊕ D
- The next pointer of D is: C ⊕ NULL

Why does it work? To have answer for this question let us consider the properties of ⊕:

X ⊕ X = 0
X ⊕ 0 = X
X ⊕ Y = Y ⊕ X (symmetric)
(X ⊕ Y) ⊕ Z = X ⊕ (Y ⊕ Z) (transitive)

For the example above, let us assume that we are at C node and want to move to B. We know that Cs *ptrdiff* is defined as B ⊕ D. If we want to move to B, performing ⊕ on Cs *ptrdiff* with D would give B. This is due to fact that,

$$(B \oplus D) \oplus D = B \text{ (since, } D \oplus D=0)$$

Similarly, if we want to move to D, then we have to apply ⊕ to Cs *ptrdiff* with B would give D.

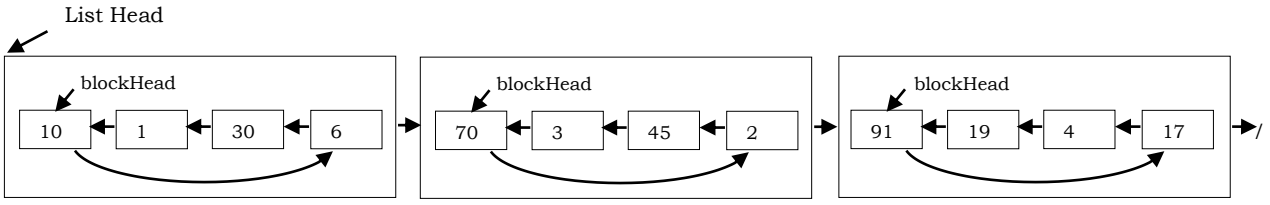
$$(B \oplus D) \oplus B = D \text{ (since, } B \oplus B=0)$$

From the above discussion we can see that just by using single pointer, we can move back and forth. A memory-efficient implementation of a doubly linked list is possible without compromising much timing efficiency.

3.10 Unrolled Linked Lists

One of the biggest advantages of linked lists over arrays is that inserting an element at any location takes only O(1) time. However, it takes O(*n*) to search for an element in a linked list. There is a simple variation of the singly linked list called *unrolled linked lists*.

An unrolled linked list stores multiple elements in each node (let us call it a block for our convenience). In each block, a circular linked list is used to connect all nodes.

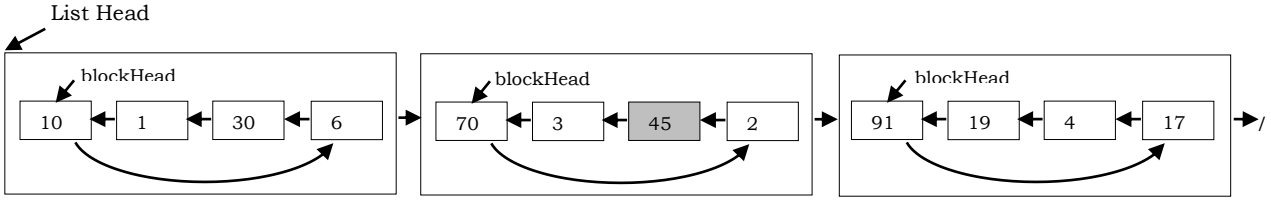


Assume that there will be no more than n elements in the unrolled linked list at any time. To simplify this problem, all blocks, except the last one, should contain exactly $\lceil\sqrt{n}\rceil$ elements. Thus, there will be no more than $\lfloor\sqrt{n}\rfloor$ blocks at any time.

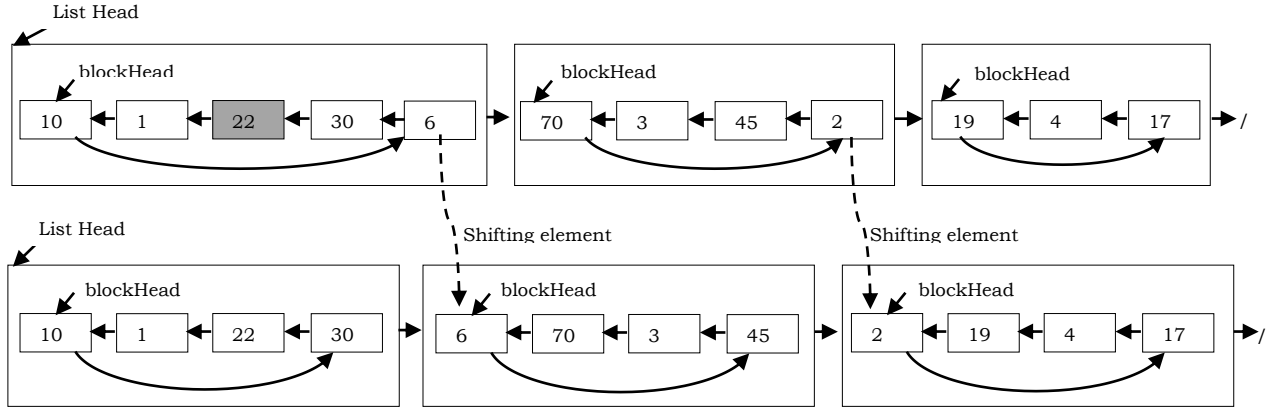
Searching for an element in Unrolled Linked Lists

In unrolled linked lists, we can find the k^{th} element in O($\sqrt{n}$):

1. Traverse on the *list of blocks* to the one that contains the k^{th} node, i.e., the $\left\lceil \frac{k}{\lceil \sqrt{n} \rceil} \right\rceil^{\text{th}}$ block. It takes O($\sqrt{n}$) since we may find it by going through no more than $\sqrt{n}$ blocks.
2. Find the $(k \bmod \lceil \sqrt{n} \rceil)^{\text{th}}$ node in the circular linked list of this block. It also takes O($\sqrt{n}$) since there are no more than $\lceil \sqrt{n} \rceil$ nodes in a single block.



Inserting an element in Unrolled Linked Lists

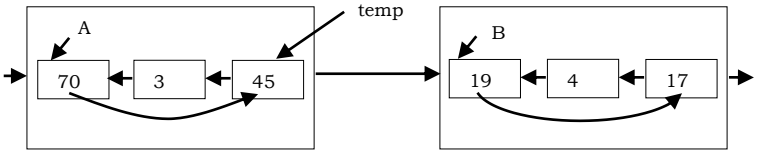


When inserting a node, we have to re-arrange the nodes in the unrolled linked list to maintain the properties previously mentioned, that each block contains $\lceil \sqrt{n} \rceil$ nodes. Suppose that we insert a node x after the i^{th} node, and x should be placed in the j^{th} block. Nodes in the j^{th} block and in the blocks after the j^{th} block have to be shifted toward the tail of the list so that each of them still have $\lceil \sqrt{n} \rceil$ nodes. In addition, a new block needs to be added to the tail if the last block of the list is out of space, i.e., it has more than $\lceil \sqrt{n} \rceil$ nodes.

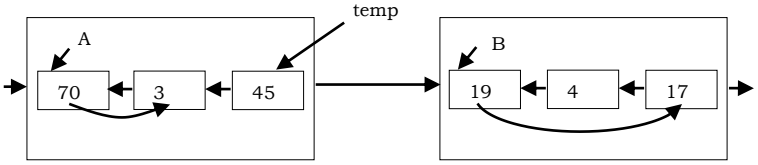
Performing Shift Operation

Note that each *shift* operation, which includes removing a node from the tail of the circular linked list in a block and inserting a node to the head of the circular linked list in the block after, takes only O(1). The total time complexity of an insertion operation for unrolled linked lists is therefore O($\sqrt{n}$); there are at most O($\sqrt{n}$) blocks and therefore at most O($\sqrt{n}$) shift operations.

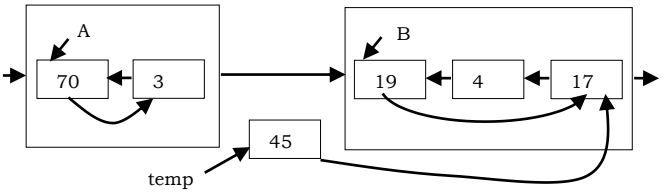
1. A temporary pointer is needed to store the tail of A.



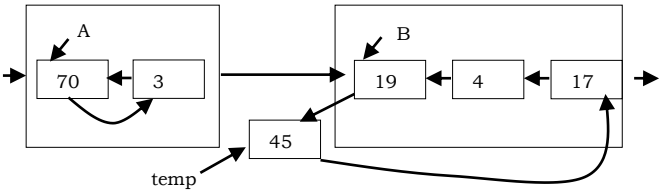
2. In block A, move the next pointer of the head node to point to the second-to-the-last node, so that the tail node of A can be removed.



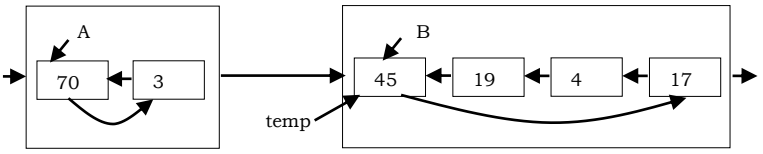
3. Let the next pointer of the node which will be shifted (the tail node of A) point to the tail node of B.



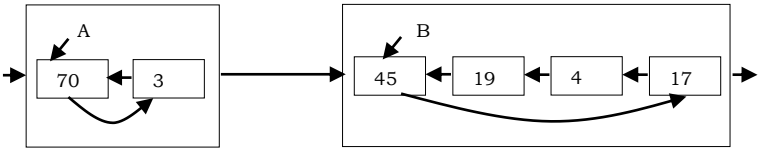
4. Let the next pointer of the head node of B point to the node temp points to.



5. Finally, set the head pointer of B to point to the node temp points to. Now the node temp points to become the new head node of B.



6. *temp* pointer can be thrown away. We have completed the shift operation to move the original tail node of A to become the new head node of B.



Performance

With unrolled linked lists, there are a couple of advantages, one in speed and one in space.

First, if the number of elements in each block is appropriately sized (e.g., at most the size of one cache line), we get noticeably better cache performance from the improved memory locality.

Second, since we have $O(n/m)$ links, where n is the number of elements in the unrolled linked list and m is the number of elements we can store in any block, we can also save an appreciable amount of space, which is particularly noticeable if each element is small.

Comparing Doubly Linked Lists and Unrolled Linked Lists

To compare the overhead for an unrolled list, elements in doubly linked list implementations consist of data, a pointer to the next node and a pointer to the previous node in the list as shown below.

Assuming we have got 4 byte pointers, each node is going to take 8 bytes. But the allocation overhead for the node could be anywhere between 8 and 16 bytes. Let's go with the best case and assume it will be 8 bytes. So, if we want to store 1K items in this list, we are going to have 16KB of overhead.

Now, let's think about an unrolled linked list node (let us call it *LinkedBlock*). It will look something like this:

Therefore, allocating a single node (12 bytes + 8 bytes of overhead) with an array of 100 elements (400 bytes + 8 bytes of overhead) will now cost 428 bytes, or 4.28 bytes per element. Thinking about our 1K items from above, it would take about 4.2KB of overhead, ,which is close to 4x better than our original list. Even if the list becomes severely fragmented and the item arrays are only 1/2 full on average, this is still an improvement. Also, note that we can tune the array size to whatever gets we the best overhead for our application.

Implementation

```
public class UnrolledLinkedList<E> extends AbstractList<E> implements List<E>, Serializable {
   //The maximum number of elements that can be stored in a single node.
   private int nodeCapacity;
```

```
//The current size of this list.
private int size = 0;
//The first node of this list.
private ListNode firstNode;
//The last node of this list.
private ListNode lastNode;
//Constructs an empty list with the specified capacity
public UnrolledLinkedList(int nodeCapacity) throws IllegalArgumentException {
    if (nodeCapacity < 8) {
        throw new IllegalArgumentException("nodeCapacity < 8");
    }
    this.nodeCapacity = nodeCapacity;
    firstNode = new ListNode();
    lastNode = firstNode;
}
public UnrolledLinkedList() {
    this(16);
}
public int size() {
    return size;
}
public boolean isEmpty() {
    return (size == 0);
}
//Returns true if this list contains the specified element.
public boolean contains(Object o) {
    return (indexOf(o) != -1);
}
public Iterator<E> iterator() {
    return new ULLIterator(firstNode, 0, 0);
}
//Appends the specified element to the end of this list.
public boolean add(E e) {
    insertIntoNode(lastNode, lastNode.numElements, e);
    return true;
}
//Removes the first occurrence of the specified element from this list,
public boolean remove(Object o) {
    int index = 0;
    ListNode node = firstNode;
    if (o == null) {
        while (node != null) {
            for (int ptr = 0; ptr < node.numElements; ptr++) {
                if (node.elements[ptr] == null) {
                    removeFromNode(node, ptr);
                    return true;
                }
            }
            index += node.numElements;
            node = node.next;
        }
    } else {
        while (node != null) {
            for (int ptr = 0; ptr < node.numElements; ptr++) {
                if (o.equals(node.elements[ptr])) {
                    removeFromNode(node, ptr);
                    return true;
                }
            }
            index += node.numElements;
            node = node.next;
        }
    }
    return false;
```

```
}
//Removes all of the elements from this list.
public void clear() {
    ListNode node = firstNode.next;
    while (node != null) {
        ListNode next = node.next;
        node.next = null;
        node.previous = null;
        node.elements = null;
        node = next;
    }
    lastNode = firstNode;
    for (int ptr = 0; ptr < firstNode.numElements; ptr++) {
        firstNode.elements[ptr] = null;
    }
    firstNode.numElements = 0;
    firstNode.next = null;
    size = 0;
}
//Returns the element at the specified position in this list.
public E get(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    ListNode node;
    int p = 0;
    if (size - index > index) {
        node = firstNode;
        while (p <= index - node.numElements) {
            p += node.numElements;
            node = node.next;
        }
    } else {
        node = lastNode;
        p = size;
        while ((p -= node.numElements) > index) {
            node = node.previous;
        }
    }
    return (E) node.elements[index - p];
}
//Replaces the element at the specified position in this list with the specified element.
public E set(int index, E element) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    E el = null;
    ListNode node;
    int p = 0;
    if (size - index > index) {
        node = firstNode;
        while (p <= index - node.numElements) {
            p += node.numElements;
            node = node.next;
        }
    } else {
        node = lastNode;
        p = size;
        while ((p -= node.numElements) > index) {
            node = node.previous;
        }
    }
    el = (E) node.elements[index - p];
    node.elements[index - p] = element;
```

```
    return el;

}
//Inserts the specified element at the specified position in this list.
//Shifts the element currently at that position (if any) and any
//subsequent elements to the right (adds one to their indices).
public void add(int index, E element) throws IndexOutOfBoundsException {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException();
    }
    ListNode node;
    int p = 0;
    if (size - index > index) {
        node = firstNode;
        while (p <= index - node.numElements) {
            p += node.numElements;
            node = node.next;
        }
    } else {
        node = lastNode;
        p = size;
        while ((p -= node.numElements) > index) {
            node = node.previous;
        }
    }
    insertIntoNode(node, index - p, element);
}
//Removes the element at the specified position in this list.
//Shifts any subsequent elements to the left (subtracts one from their indices).
public E remove(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    E element = null;
    ListNode node;
    int p = 0;
    if (size - index > index) {
        node = firstNode;
        while (p <= index - node.numElements) {
            p += node.numElements;
            node = node.next;
        }
    } else {
        node = lastNode;
        p = size;
        while ((p -= node.numElements) > index) {
            node = node.previous;
        }
    }
    element = (E) node.elements[index - p];
    removeFromNode(node, index - p);
    return element;
}
private static final long serialVersionUID = -67405230910304521 1L;
private class ListNode {
    ListNode next;
    ListNode previous;
    int numElements = 0;
    Object[] elements;
    ListNode() {
        elements = new Object[nodeCapacity];
    }
}
private class ULLIterator implements ListIterator<E> {
```

```
ListNode currentNode;
int ptr;
int index;
private int expectedModCount = modCount;
ULLIterator(ListNode node, int ptr, int index) {
    this.currentNode = node;
    this.ptr = ptr;
    this.index = index;
}
public boolean hasNext() {
    return (index < size - 1)
}
public E next() {
    ptr++;
    if (ptr >= currentNode.numElements) {
        if (currentNode.next != null) {
            currentNode = currentNode.next;
            ptr = 0;
        } else {
            throw new NoSuchElementException();
        }
    }
    index++;
    checkForModification();
    return (E) currentNode.elements[ptr];
}
public boolean hasPrevious() {
    return (index > 0);
}
public E previous() {
    ptr--;
    if (ptr < 0) {
        if (currentNode.previous != null) {
            currentNode = currentNode.previous;
            ptr = currentNode.numElements - 1;
        } else {
            throw new NoSuchElementException();
        }
    }
    index--;
    checkForModification();
    return (E) currentNode.elements[ptr];
}
public int nextIndex() {
    return (index + 1);
}
public int previousIndex() {
    return (index - 1);
}
public void remove() {
    checkForModification();
    removeFromNode(currentNode, ptr);
}
public void set(E e) {
    checkForModification();
    currentNode.elements[ptr] = e;
}
public void add(E e) {
    checkForModification();
    insertIntoNode(currentNode, ptr + 1, e);
}
private void checkForModification() {
    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
```

```
            }
        }
    }
    private void insertIntoNode(ListNode node, int ptr, E element) {
        // if the node is full
        if (node.numElements == nodeCapacity) {
            // create a new node
            ListNode newNode = new ListNode();
            // move half of the elements to the new node
            int elementsToMove = nodeCapacity / 2;
            int startIndex = nodeCapacity - elementsToMove;
            for (int i = 0; i < elementsToMove; i++) {
                newNode.elements[i] = node.elements[startIndex + i];
                node.elements[startIndex + i] = null;
            }
            node.numElements -= elementsToMove;
            newNode.numElements = elementsToMove;
            // insert the new node into the list
            newNode.next = node.next;
            newNode.previous = node;
            if (node.next != null) {
                node.next.previous = newNode;
            }
            node.next = newNode;
            if (node == lastNode) {
                lastNode = newNode;
            }
            // check whether the element should be inserted into
            // the original node or into the new node
            if (ptr > node.numElements) {
                node = newNode;
                ptr -= node.numElements;
            }
        }
        for (int i = node.numElements; i > ptr; i--) {
            node.elements[i] = node.elements[i - 1];
        }
        node.elements[ptr] = element;
        node.numElements++;
        size++;
        modCount++;
    }
    private void removeFromNode(ListNode node, int ptr) {
        node.numElements--;
        for (int i = ptr; i < node.numElements; i++) {
            node.elements[i] = node.elements[i + 1];
        }
        node.elements[node.numElements] = null;
        if (node.next != null && node.next.numElements + node.numElements <= nodeCapacity) {
            mergeWithNextNode(node);
        } else if (node.previous != null && node.previous.numElements + node.numElements <= nodeCapacity) {
            mergeWithNextNode(node.previous);
        }
        size--;
        modCount++;
    }
    //This method does merge the specified node with the next node.
    private void mergeWithNextNode(ListNode node) {
        ListNode next = node.next;
        for (int i = 0; i < next.numElements; i++) {
            node.elements[node.numElements + i] = next.elements[i];
            next.elements[i] = null;
        }
        node.numElements += next.numElements;
```

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

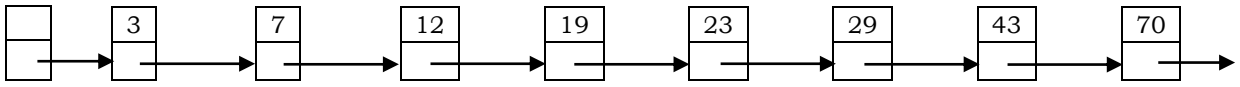
3.11 Skip Lists

Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in order, produce degenerate data structures that give very poor performance. If it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any input sequence. In most cases queries must be answered on-line, so randomly permuting the input is impractical. Balanced tree algorithms re-arrange the tree as operations are performed to maintain certain balance conditions and assure good performance.

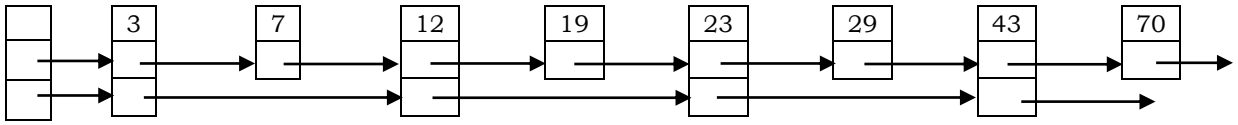
Skip lists are a probabilistic alternative to balanced trees. Skip list is a data structure that can be used as an alternative to balanced binary trees (refer *Trees* chapter). As compared to a binary tree, skip lists allow quick search, insertions and deletions of elements. This is achieved by using probabilistic balancing rather than strictly enforce balancing. It is basically a linked list with additional pointers such that intermediate nodes can be skipped. It uses a random number generator to make some decisions.

In an ordinary sorted linked list, search, insert, and delete are in O(*n*) because the list must be scanned node-by-node from the head to find the relevant node. If somehow we could scan down the list in bigger steps (skip down, as it were), we would reduce the cost of scanning. This is the fundamental idea behind Skip Lists.

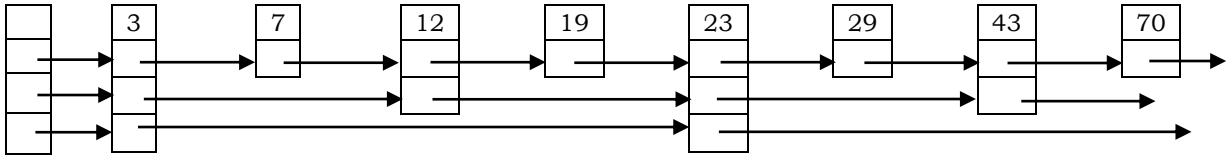
Skip Lists with One Level



Skip Lists with Two Levels



Skip Lists with Three Levels



This section gives algorithms to search for, insert and delete elements in a dictionary or symbol table. The Search operation returns the contents of the value associated with the desired key or failure if the key is not present. The Insert operation associates a specified key with a new value (inserting the key if it had not already been present). The Delete operation deletes the specified key. It is easy to support additional operations such as "find the minimum key" or "find the next key".

Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. A level i node has i forward pointers, indexed 1 through i. We do not need to store the level of a node in the node. Levels are capped at some appropriate constant MaxLevel. The level of a list is the maximum level currently in the list (or 1 if the list is empty). The header of a list has forward pointers at levels one through MaxLevel. The forward pointers of the header at levels higher than the current maximum level of the list point to NULL.

Initialization

An element NIL is allocated and given a key greater than any legal key. All levels of all skip lists are terminated with NIL. A new list is initialized so that the the level of the list is equal to 1 and all forward pointers of the list's header point to NIL.

Search for an element

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

Insertion and Deletion Algorithms

To insert or delete a node, we simply search and splice. A vector update is maintained so that when the search is complete (and we are ready to perform the splice), update[i] contains a pointer to the rightmost node of level i or higher that is to the left of the location of the insertion/deletion. If an insertion generates a node with a level greater than the previous maximum level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector. After each deletion, we check if we have deleted the maximum element of the list and if so, decrease the maximum level of the list.

Choosing a Random Level

Initially, we discussed a probability distribution where half of the nodes that have level i pointers also have level i+1 pointers. To get away from magic constants, we say that a fraction p of the nodes with level i pointers also have level i+1 pointers. (for our original discussion, p = 1/2). Levels are generated randomly by an algorithm. Levels are generated without reference to the number of elements in the list

Performance

In a simple linked list that consists of n elements, to perform a search n comparisons are required in the worst case. If a second pointer pointing two nodes ahead is added to every node, the number of comparisons goes down to $n/2 + 1$ in the worst case.

Adding one more pointer to every fourth node and making them point to the fourth node ahead reduces the number of comparisons to $\lceil n/2 \rceil + 2$. If this strategy is continued so that every node with i pointers points to $2 * i - 1$ nodes ahead, O($logn$) performance is obtained and the number of pointers has only doubled ($n + n/2 + n/4 + n/8 + n/16 + \ldots. = 2n$).

The find, insert, and remove operations on ordinary binary search trees are efficient, O($logn$), when the input data is random; but less efficient, O(n), when the input data are ordered. Skip List performance for these same operations and for any data set is about as good as that of randomly-built binary search trees - namely O($logn$).

Comparing Skip Lists and Unrolled Linked Lists

In simple terms, Skip Lists are sorted linked lists with two differences:

- The nodes in an ordinary list have one next reference. The nodes in a Skip List have many *next* references (also called $forward$ references).
- The number of $forward$ references for a given node is determined probabilistically.

We speak of a Skip List node having levels, one level per forward reference. The number of levels in a node is called the *size* of the node. In an ordinary sorted list, insert, remove, and find operations require sequential traversal of the list. This results in O(n) performance per operation. Skip Lists allow intermediate nodes in the list to be skipped during a traversal - resulting in an expected performance of O($logn$) per operation.

Implementation

```
import java.util.Random;
public class SkipList<T extends Comparable<T>, U>{
    private class Node{
            public T key;
            public U value;
            public long level;
            public Node next;
```

```
		public Node down;
		public Node(T key, U value, long level, Node next, Node down){
			this.key = key;
			this.value = value;
			this.level = level;
			this.next = next;
			this.down = down;
		}
	}
	private Node head;
	private Random _random;
	private long size;
	private double _p;
	private long level(){
		long level = 0;
		while (level <= size && _random.nextDouble() < _p) {
			level++;
		}
		return level;
	}
	public SkipList(){
		head = new Node(null, null, 0, null, null);
		_random = new Random();
		size = 0;
		_p = 0.5;
	}
	public void add(T key, U value){
		long level = level();
		if (level > head.level) {
			head = new Node(null, null, level, null, head);
		}
		Node cur = head;
		Node last = null;
		while (cur != null) {
			if (cur.next == null || cur.next.key.compareTo(key) > 0) {
				if (level >= cur.level) {
					Node n = new Node(key, value, cur.level, cur.next, null);
					if (last != null) {
						last.down = n;
					}
					cur.next = n;
					last = n;
				}
				cur = cur.down;
				continue;
			} else if (cur.next.key.equals(key)) {
				cur.next.value = value;
				return;
			}
			cur = cur.next;
		}
		size++;
	}
	public boolean containsKey(T key){
		return get(key) != null;
	}
	public U remove(T key){
		U value = null;
		Node cur = head;
		while (cur != null) {
			if (cur.next == null || cur.next.key.compareTo(key) >= 0) {
```

```
                                    if (cur.next != null && cur.next.key.equals(key)) {
                                            value = cur.next.value;
                                            cur.next = cur.next.next;
                                    }
                                    cur = cur.down;
                                    continue;
                            }
                            cur = cur.next;
                    }
                    size--;
                    return value;
            }
            public U get(T key){
                    Node cur = head;
                    while (cur != null) {
                            if (cur.next == null || cur.next.key.compareTo(key) > 0) {
                                    cur = cur.down;
                                    continue;
                            } else if (cur.next.key.equals(key)) {
                                    return cur.next.value;
                            }
                            cur = cur.next;
                    }
                    return null;
            }
    }
    public class SkipListsTest{
            public static void main(String [] args){
                    SkipList s = new SkipList();
                    s.add(1,100);
                    System.out.println(s.get(1));
            }
    }
```

3.11 Problems on Linked Lists

Problem-1 Implement Stack using Linked List

Solution: Refer *Stacks* chapter.

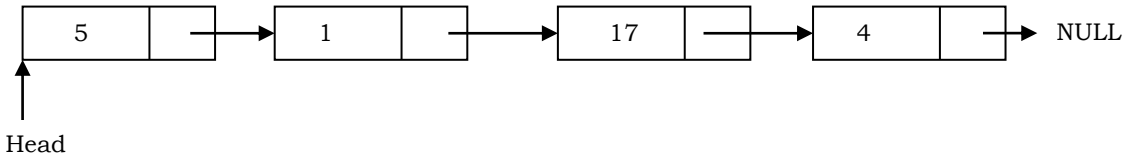
Problem-2 Find n^{th} node from the end of a Linked List.

Solution: Brute-Force Approach: In this method, start with the first node and count the number of nodes present after that node. If the number of nodes are $< n-1$ then return saying "fewer number of nodes in the list". If the number of nodes are $> n-1$ then go to next node. Continue this until the numbers of nodes after current node are $n-1$.

Time Complexity: O(n^2), for scanning the remaining list (from current node) for each node.
Space Complexity: O(1).

Problem-3 Can we improve the complexity of Problem-2?

Solution: Yes, using hash table. As an example consider the following list.



In this approach, create a hash table whose entries are < *position of node, node address* >. That means, key is the position of the node in the list and value is the address of that node.

Position in List	Address of Node
1	Address of 5 node
2	Address of 1 node
3	Address of 17 node

4	Address of 4 node

By the time we traverse the complete list (for creating hash table), we can find the list length. Let us say, the list length is M. To find n^{th} from end of linked list, we can convert this to $M- n + 1^{th}$ from the beginning. Since we already know the length of the list, it's just a matter of returning $M- n + 1^{th}$ key value from the hash table.

Time Complexity: Time for creating the hash table. Therefore, $T(m)$ = O(m). Space Complexity: O(m). Since, we need to create a hash table of size m.

Problem-4 Can we use Problem-3 approach for solving Problem-2 without creating the hash table?

Solution: Yes. If we observe the Problem-3 solution, what actually we are doing is finding the size of the linked list. That means, we are using hash table to find the size of the linked list. We can find the length of the linked list just by starting at the head node and traversing the list. So, we can find the length of the list without creating the hash table. After finding the length, compute $M - n + 1$ and with one more scan we can get the $M - n + 1^{th}$ node from the beginning. This solution needs two scans: one for finding the length of list and other for finding $M - n + 1^{th}$ node from the beginning.

Time Complexity: Time for finding the length + Time for finding the M - $n + 1^{th}$ node from the beginning. Therefore, $T(n)$ = O(n) +O(n) ≈O(n). Space Complexity: O(1). Since, no need of creating the hash table.

Problem-5 Can we solve Problem-2 in one scan?

Solution: Yes. **Efficient Approach**: Use two pointers $pNthNode$ and $pTemp$. Initially, both points to head node of the list. $pNthNode$ starts moving only after $pTemp$ made n moves. From there both moves forward until $pTemp$ reaches end of the list. As a result $pNthNode$ points to n^{th} node from end of the linked list.

Note: at any point of time both moves one node at time.

```
public ListNode  NthNodeFromEnd(ListNode head , int NthNode) {
      ListNode pTemp = head, pNthNode = null;
      for(int count =1; count< NthNode;count++) {
         if(pTemp != null)
            pTemp = pTemp.getNext();
      }
      while(pTemp!= null){
         if(pNthNode == null)
              pNthNode = head;
         else
              pNthNode = pNthNode.getNext();
         pTemp = pTemp.getNext();
      }
      if(pNthNode !=  null)
            return pNthNode;
      return null;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-6 Can we solve Problem-5 with recursion?

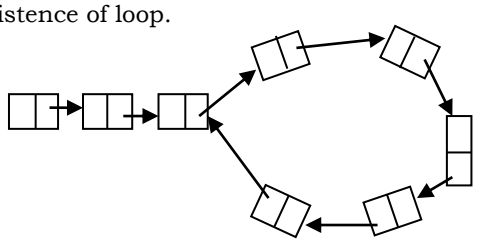
Solution: We can use a global variable to track the post recursive call and when it is same as Nth', return the node.

```
public ListNode NthNodeFromEnd(ListNode head, int Nth) {
   if(head != null) {
      NthNodeFromEnd(head.next, Nth);
      counter++;
      if(Nth ==counter) {
         return head;
      }
   }
   return null;
}
```

Time Complexity: O(n) for pre recursive calls and O(n) for post recursive calls, which is = O(2n) =O(n).
Space Complexity: O(n) for recursive stack.

Problem-7 Check whether the given linked list is either NULL-terminated or ends in a cycle (cyclic)

Solution: Brute-Force Approach. As an example consider the following linked list which has a loop in it. The difference between this list and the regular list is that, in this list there are two nodes whose next pointers are same. In regular singly linked lists (without loop) each nodes next pointer is unique. That means, the repetition of next pointers indicates the existence of loop.



One simple and brute force way of solving this is, start with the first node and see whether there is any node whose next pointer is current node's address. If there is a node with same address then that indicates that some other node is pointing to the current node and we can say loops exists. Continue this process for all the nodes of the linked list.

Does this method works? As per the algorithm we are checking for the next pointer addresses, but how do we find the end of the linked list (otherwise we will end up in infinite loop)?

Note: If we start with a node in loop, this method may work depending on the size of the loop.

Problem-8 Can we use hashing technique for solving Problem-7?

Solution: Yes. Using Hash Tables we can solve this problem.

Algorithm

- Traverse the linked list nodes one by one.
- Check if the address of the node is available in the hash table or not.
- If it is already available in the hash table then that indicates that we are visiting the node that was already visited. This is possible only if the given linked list has a loop in it.
- If the address of the node is not available in the hash table then insert that nodes address into the hash table.
- Continue this process until we reach end of the linked list *or* we find loop.

Time Complexity*:* O(n) for scanning the linked list. Note that we are doing only scan of the input.
Space Complexity*:* O(n) for hash table.

Problem-9 Can we solve the Problem-6 using sorting technique?

Solution: No. Consider the following algorithm which is based on sorting. And then, we see why this algorithm fails.

Algorithm

- Traverse the linked list nodes one by one and take all the next pointer values into some array.
- Sort the array that has next node pointers.
- If there is a loop in the linked list, definitely two nodes next pointers will pointing to the same node.
- After sorting if there is a loop in the list, the nodes whose next pointers are same will come adjacent in the sorted list.
- If any such pair exists in the sorted list then we say the linked list has loop in it.

Time Complexity*:* O($nlogn$) for sorting the next pointers array. Space Complexity*:* O(n) for the next pointers array.

Problem with above algorithm? The above algorithm works only if we can find the length of the list. But if the list is having loop then we may end up in infinite loop. Due to this reason the algorithm fails.

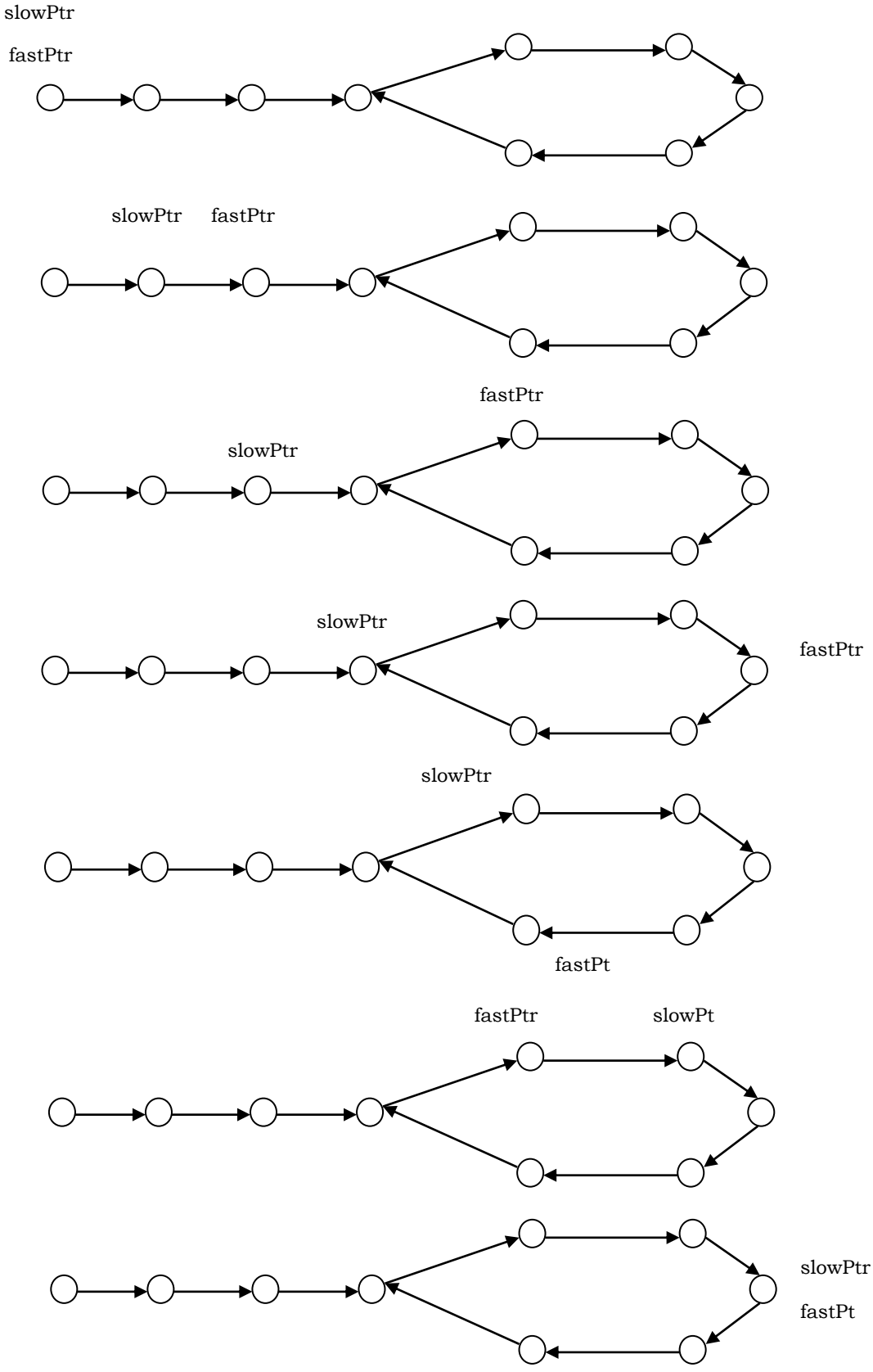
Problem-10 Can we solve the Problem-6 in O(n)?

Solution: Yes. **Efficient Approach (Memory less Approach):** This problem was solved by *Floyd*. The solution is named as Floyd cycle finding algorithm. It uses 2 pointers moving at different speeds to walk the linked list. Once they enter the loop they are expected to meet, which denotes that there is a loop. This works because the only way a faster moving pointer would point to the same location as a slower moving pointer is, if somehow the entire list or a part of it is circular.

Think of a tortoise and a hare running on a track. The faster running hare will catch up with the tortoise if they are running in a loop. As an example, consider the following example and trace out the Floyd algorithm. From

the diagrams below we can see that after the final step they are meeting at some point in the loop which may not be the starting of the loop.

Note: *slowPtr* (*tortoise*) moves one pointer at a time and *fastPtr* (*hare*) moves two pointers at a time



```
private static boolean findIfLoopExistsUsingFloyds(ListNode head){
    ListNode fastPtr = head;
    ListNode slowPtr = head;
    while (fastPtr != null && fastPtr.getNext() != null) {
        fastPtr = fastPtr.getNext().getNext();
        slowPtr = slowPtr.getNext();
        if (slowPtr == fastPtr)
            return true;
    }
    return false;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-11 You are given a pointer to the first element of a linked list L. There are two possibilities for L, it either ends (snake) or its last element points back to one of the earlier elements in the list (snail). Task is to devise an algorithm that tests whether a given list L is a snake or a snail.

Solution: It is same as Problem-6.

Problem-12 Check whether the given linked list is either NULL-terminated or not. If there is a cycle find the start node of the loop.

Solution: The solution is an extension to the previous solution (Problem-10). After finding the loop in the linked list, we initialize the *slowPtr* to head of the linked list. From that point onwards both *slowPtr* and *fastPtr* moves only one node at a time. The point at which they meet is the start of the loop. Generally we use this method for removing the loops.

```
private static ListNode findBeginofLoop(ListNode head){
    ListNode fastPtr = head;
    ListNode slowPtr = head;
    boolean loopExists = false;
    while (fastPtr != null && fastPtr.getNext() != null) {
        fastPtr = fastPtr.getNext().getNext();
        slowPtr = slowPtr.getNext();
        if (slowPtr == fastPtr) {
            loopExists = true;
            break;
        }
    }
    if (loopExists) {
        slowPtr = head;
        while (slowPtr != fastPtr) {
            slowPtr = slowPtr.getNext();
            fastPtr = fastPtr.getNext();
        }
        return fastPtr;
    } else
        return null;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-13 From the previous problems we understand that the meeting of tortoise and hare meeting concludes the existence of loop, but how does moving tortoise to beginning of linked list while keeping the hare at meeting place, followed by moving both one step at a time make them meet at starting point of cycle?

Solution: This problem is the heart of number theory. In Floyd cycle finding algorithm, notice that the tortoise and the hare will meet when they are $n \times L$, where L is the loop length. Furthermore, the tortoise is at the midpoint between the hare and the beginning of the sequence, because of the way they move. Therefore the tortoise is $n \times L$ away from the beginning of the sequence as well.

If we move both one step at a time, from the position of tortoise and from the start of the sequence, we know that they will meet as soon as both are in the loop, since they are $n \times L$, a multiple of the loop length, apart. One of them is already in the loop, so we just move the other one in single step until it enters the loop, keeping the other $n \times L$ away from it at all times.

Problem-14 In Floyd cycle finding algorithm, does it work if we use the steps 2 and 3 instead of 1 and 2?

Solution: Yes, but the complexity might be high. Trace out some example.

Problem-15 Check whether the given linked list is NULL-terminated. If there is a cycle find the length of the loop.

Solution: This solution is also an extension to the basic cycle detection problem. After finding the loop in the linked list, keep the *slowPtr* as it is. *fastPtr* keeps on moving until it again comes back to *slowPtr*. While moving *fastPtr*, use a counter variable which increments at the rate of 1.

```
private static int findLengthOfTheLoop(ListNode head){
    ListNode fastPtr = head;
    ListNode slowPtr = head;
    boolean loopExists = false;
    while (fastPtr != null && fastPtr.getNext() != null) {
        fastPtr = fastPtr.getNext().getNext();
        slowPtr = slowPtr.getNext();
        if (slowPtr == fastPtr) {
            loopExists = true;
            break;
        }
    }
    int length = 0;
    if (loopExists) {
        do {
            slowPtr = slowPtr.getNext();
            length++;
        } while (slowPtr != fastPtr);
    }
    return length;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-16 Insert a node in a sorted linked list

Solution: Traverse the list and find a position for the element and insert it.

```
    public ListNode InsertInSortedList(ListNode head, ListNode newNode) {
            ListNode current = head;
            if(head == null) return newNode;
            // traverse the list until you find item bigger the new node value
            while (current != null && current.getData() <  newNode.getData()){
                    temp = current;
                    current = current.getNext();
            }
            //  insert the new node before the big item
            newNode.setNext(current);
            temp.setNext(newNode);
            return head;
    }
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-17 Reverse a singly linked list

Solution:

Iterative version:

```
public static ListNode reverseListIterative(ListNode head){
    //initially Current is head
    ListNode current = head;
    //initially previous is null
    ListNode prev = null;
    while (current != null) {
        //Save the next node
        ListNode next = current.getNext();
        //Make current node points to the previous
        current.setNext(prev);
        //update previous
```

```
        prev = current;
        //update current
        current = next;
    }
    return prev;
}
```

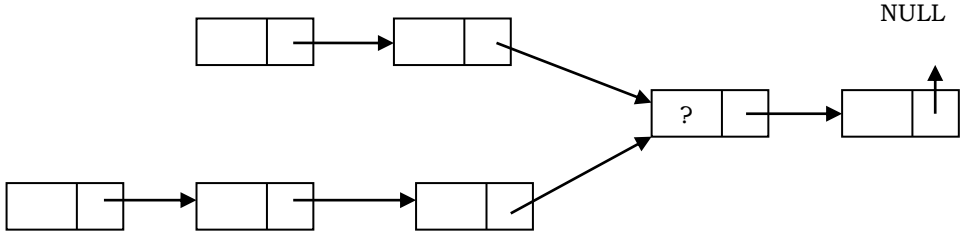
Time Complexity: O(n). Space Complexity: O(1).

Recursive version:

```
public static void reverseLinkedListRecursive(ListNode current, ListNode[] head){
    if (current == null) {
        return;
    }
    ListNode next = current.getNext();
    if (next == null) {
        head[0] = current;
        return;
    }
    reverseLinkedListRecursive(next, head);
    //Make next node points to current node
    next.setNext(current);
    //Remove existing link
    current.setNext(null);
}
```

Time Complexity: O(n). Space Complexity: O(n) for recursive stack.

Problem-18 Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the list before they intersect are unknown and both list may have it different. $List1$ may have n nodes before it reaches intersection point and $List2$ might have m nodes before it reaches intersection point where m and n may be $m = n, m < n$ or $m > n$. Give an algorithm for finding the merging point.



Solution: Brute-Force Approach: One easy solution is to compare every node pointer in the first list with every other node pointer in the second list by which the matching node pointers will lead us to the intersecting node. But, the time complexity in this case will O(mn) which will be high.

Time Complexity: O(mn). Space Complexity: O(1).

Problem-19 Can we solve Problem-18 using sorting technique?

Solution: No. Consider the following algorithm which is based on sorting and see why this algorithm fails.

Algorithm

- Take first list node pointers and keep in some array and sort them.
- Take second list node pointers and keep in some array and sort them.
- After sorting, use two indexes: one for first sorted array and other for second sorted array.
- Start comparing values at the indexes and increment the index whichever has lower value (increment only if the values are not equal).
- At any point, if we were able to find two indexes whose values are same then that indicates that those two nodes are pointing to the same node and we return that node.

Time Complexity: Time for sorting lists + Time for scanning (for comparing) =O($mlogm$) + O($nlogn$) +O($m + n$). We need to consider the one that gives the maximum value. Space Complexity: O(1).

Problem with the above algorithm? Yes. In the algorithm, we are storing all the node pointers of both the lists and sorting. But we are forgetting the fact that, there can be many repeated elements. This is because after the

merging point all node pointers are same for both the lists. The algorithm works fine only in one case and it is when both lists have ending node at their merge point.

Problem-20 Can we solve Problem-18 using hash tables?

Solution: Yes.

Algorithm:

- Select a list which has less number of nodes (If we do not know the lengths beforehand then select one list randomly).
- Now, traverse the other list and for each node pointer of this list check whether the same node pointer exists in the hash table.
- If there is a merge point for the given lists then we will definitely encounter the node pointer in the hash table.

Time Complexity: Time for creating the hash table + Time for scanning the second list = $O(m) + O(n)$ (or $O(n) + O(m)$, depends on which list we select for creating the hash table). But in both cases the time complexity is same. Space Complexity: $O(n)$ or $O(m)$.

Problem-21 Can we use stacks for solving the Problem-18?

Solution: Yes.

Algorithm:

- Create two stacks: one for the first list and one for the second list.
- Traverse the first list and push all the node address on to the first stack.
- Traverse the second list and push all the node address on to the second stack.
- Now both stacks contain the node address of the corresponding lists.
- Now, compare the top node address of both stacks.
- If they are same, then pop the top elements from both the stacks and keep in some temporary variable (since both node addresses are node, it is enough if we use one temporary variable).
- Continue this process until top node addresses of the stacks are not same.
- This point is the one where the lists merge into single list.
- Return the value of the temporary variable.

Time Complexity: $O(m + n)$, for scanning both the lists. Space Complexity: $O(m + n)$, for creating two stacks for both the lists.

Problem-22 Is there any other way of solving the Problem-18?

Solution: **Yes**. Using "finding the first repeating number" approach in an array (for algorithm refer *Searching* chapter).

Algorithm:

- Create an array A and keep all the next pointers of both the lists in the array.
- In the array find the first repeating element in the array [Refer *Searching* chapter for algorithm].
- The first repeating number indicates the merging point of the both lists.

Time Complexity: $O(m + n)$. Space Complexity: $O(m + n)$.

Problem-23 Can we still think of finding an alternative solution for the Problem-18?

Solution: Yes. By combining sorting and search techniques we can reduce the complexity.

Algorithm:

- Create an array A and keep all the next pointers of the first list in the array.
- Sort these array elements.
- Then, for each of the second list element, search in the sorted array (let us assume that we are using binary search which gives $O(logn)$).
- Since we are scanning the second list one by one, the first repeating element that appears in the array is nothing but the merging point.

Time Complexity: Time for sorting + Time for searching = $O(Max(mlogm, nlogn))$. Space Complexity: $O(Max(m, n))$.

Problem-24 Can we improve the complexity for the Problem-18?

Solution: Yes.

Efficient Approach:

- Find lengths (L1 and L2) of both list -- $O(n) + O(m) = O(max(m, n))$.
- Take the difference d of the lengths -- $O(1)$.
- Make d steps in longer list -- $O(d)$.

- Step in both lists in parallel until links to next node match -- O($min(m,n)$).
- Total time complexity = O($max(m,n)$).
- Space Complexity = O(1).

```
public static ListNode findIntersectingNode(ListNode list1, ListNode list2) {
    int L1=0, L2=0, diff=0;
    ListNode head1 = list1, head2 = list2;
    while(head1 !=  null) {
        L1++;
        head1 = head1.getNext();
    }
    while(head2 !=  null) {
        L2++;
        head2 = head2.getNext();
    }
    if(L1 < L2) {
            head1 = list2;
            head2 = list1;
            diff = L2 - L1;
    } else{
         head1 = list1;
        head2 = list2;
        diff = L1 – L2;
    }
    for(int i = 0; i < diff; i++)
         head1 = head1.getNext();
    while(head1 !=  null && head2 != null) {
            if(head1 == head2)
                return head1.getData();
            head1= head1.getNext();
            head2= head2.getNext();
    }
    return null;
}
```

Problem-25 How will you find the middle of the linked list?

Solution: Brute-Force Approach: For each of the node count how many nodes are there in the list and see whether it is the middle.

Time Complexity: O(n^2). Space Complexity: O(1).

Problem-26 Can we improve the complexity of Problem-25?

Solution: Yes.
Algorithm:

- Traverse the list and find the length of the list.
- After finding the length, again scan the list and locate $n/2$ node from the beginning.

Time Complexity: Time for finding the length of the list + Time for locating middle node = O(n) + O(n) ≈ O(n). Space Complexity: O(1).

Problem-27 Can we use hash table for solving Problem-25?

Solution: Yes. The reasoning is same as that of Problem-3.

Time Complexity: Time for creating the hash table. Therefore, $T(n)$ = O(n). Space Complexity: O(n). Since, we need to create a hash table of size n.

Problem-28 Can we solve Problem-25 just in one scan?

Solution: Efficient Approach: Use two pointers. Move one pointer at twice the speed of the second. When the first pointer reaches end of the list, the second pointer will be pointing to the middle node.

Note: If the list has even number of nodes, the middle node will be of $\lfloor n/2 \rfloor$.

```
public static ListNode findMiddle(ListNode head) {
    ListNode ptr1x, ptr2x;
    ptr1x = ptr2x = head;
    int i=0;
    // keep looping until we reach the tail (next will be NULL for the last node)
```

```
        while(ptr1x.getNext() != null) {
                if(i == 0) {
                    ptr1x = ptr1x.getNext(); //increment only the 1st pointer
                    i=1;
                }
                else if( i == 1) {
                    ptr1x = ptr1x.getNext(); //increment both pointers
                    ptr2x = ptr2x.getNext();
                    i = 0;
                }
        }
        return ptr2x;       //now return the ptr2 which points to the middle node
  }
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-29 How will you display a linked list from the end?

Solution: Traverse recursively till end of the linked list. While coming back, start printing the elements.

```
  //This Function will print the linked list from end
  public static void printListFromEnd(ListNode head) {
      if(head == null)
          return;
      printListFromEnd(head.getNext());
      System.out.println(head.getData());
  }
```

Time Complexity: O(n). Space Complexity: O(n)→ for Stack.

Problem-30 Check whether the given Linked List length is even or odd?

Solution: Use $2x$ pointer. Take a pointer that moves at $2x$ [two nodes at a time]. At the end, if the length is even then pointer will be NULL otherwise it will point to last node.

```
  public int IsLinkedListLengthEven(ListNode listHead) {
      while(listHead != null && listHead.getNext() != null)
          listHead = listHead.getNext().getNext();
      if(listHead == null) return 0;
      return 1;
  }
```

Time Complexity: O(⌊n/2⌋) ≈O(n). Space Complexity: O(1).

Problem-31 If the head of a linked list is pointing to kth element, then how will you get the elements before kth element?

Solution: Use Memory Efficient Linked Lists [XOR Linked Lists].

Problem-32 Given two sorted Linked Lists, we need to merge them into the third list in sorted order.

Solution:

```
public ListNode mergeTwoLists(ListNode head1, ListNode head2) {
    if(head1 == null)
        return head2;
    if(head2 == null)
        return head1;
    ListNode head = new ListNode(0);
    if(head1.data <= head2.data){
        head = head1;
        head.next = mergeTwoLists(head1.next, head2);
    }else{
        head = head2;
        head.next = mergeTwoLists(head2.next, head1);
    }
    return head;
}
```

Time Complexity – O(n).

Problem-33 Can we solve Problem-32 without recursion?

Solution:

```
public ListNode mergeTwoLists(ListNode head1, ListNode head2) {
    ListNode head = new ListNode(0);
    ListNode curr = head;
    while(head1 != null && head2 != null){
        if(head1.data <= head2.data){
            curr.next = head1;
            head1 = head1.next;
        }else{
            curr.next = head2;
            head2 = head2.next;
        }
    }
    if(head1 != null)
        curr.next = head1;
    else if(head2 != null)
        curr.next = head2;
    return head.next;
}
```

Time Complexity – O(n).

Problem-34 Reverse the linked list in pairs. If you have a linked list that holds $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow X$, then after the function has been called the linked list would hold $2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow X$.

Solution:

```
//Recursive Version
public static ListNode ReversePairRecursive(ListNode head) {
    ListNode temp;
    if(head ==NULL || head.next ==NULL)
            return;   //base case for empty or 1 element list
    else {
            //Reverse first pair
            temp = head.next;
            head.next = temp.next;
            temp.next = head;
            head = temp;
            //Call the method recursively for the rest of the list
            head.next.next = ReversePairRecursive(head.next.next);
            return head;
    }
}
/*Iterative version*/
public static ListNode  ReversePairIterative(ListNode  head) {
    ListNode  temp1 = null;
    ListNode  temp2 = null;
    while (head != null && head.next != null) {
        if (temp1 != null) {
            temp1.next.next = head.next;
        }
        temp1 = head.next;
        head.next = head.next.next;
        temp1.next = head;
        if (temp2 == null)
            temp2 = temp1;
        head = head.next;
    }
    return temp2;
}
```

Time Complexity – O(n). Space Complexity - O(1).

Problem-35 Given a binary tree convert it to doubly linked list.

Solution: Refer *Trees* chapter.

Problem-36 How do we sort the Linked Lists?

Solution: Refer *Sorting* chapter.

Problem-37 If we want to concatenate two linked lists which of the following gives O(1) complexity?
1) Singly linked lists 2) Doubly linked lists 3) Circular doubly linked lists

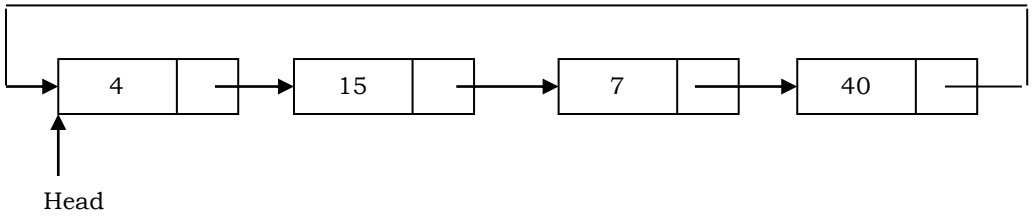
Solution: Circular Doubly Linked Lists. This is because for singly and doubly linked lists, we need to traverse the first list till the end and append the second list. But in case of circular doubly linked lists we don't have to traverse the lists.

Problem-38 Split a Circular Linked List into two equal parts. If the number of nodes in the list are odd then make first list one node extra than second list.

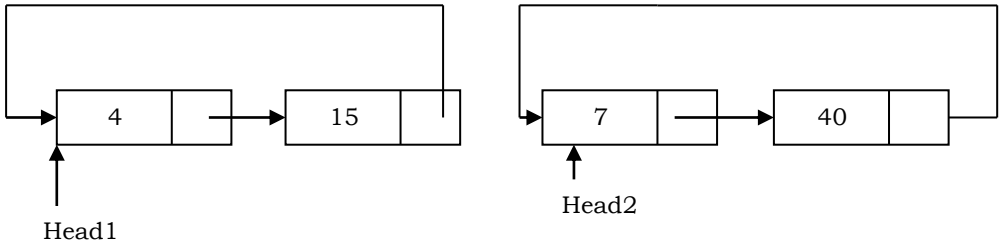
Solution:
Algorithm

- Store the mid and last pointers of the circular linked list using Floyd cycle finding algorithm.
- Make the second half circular.
- Make the first half circular.
- Set head pointers of the two linked lists.

As an example, consider the following circular list.



After the split, the above list will look like:




```
public static void SplitList(ListNode head, ListNode head1, ListNode head2) {
    ListNode slowPtr = head, fastPtr = head;
    if(head == nlu) return;
    /* If there are odd nodes in the circular list then fastPtr.getNext() becomes
       head and for even nodes fastPtr.getNext().getNext() becomes head */
       while(fastPtr.getNext() != head && fastPtr.getNext().getNext() != head)    {
            fastPtr = fastPtr.getNext().getNext();
            slowPtr = slowPtr.getNext();
    }
    /* If there are even elements in list then move fastPtr */
    if(fastPtr.getNext().getNext() == head)
            fastPtr = fastPtr.getNext();
    /* Set the head pointer of first half */
    head1 = head;
    /* Set the head pointer of second half */
    if(head.getNext() != head)
            head2 = slowPtr.getNext();
    /* Make second half circular */
    fastPtr.setNext(slowPtr.getNext());
    /* Make first half circular */
    slowPtr.setNext(head);
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-39 How will you check if the linked list is palindrome or not?

Solution:

Algorithm

1. Get the middle of the linked list.
2. Reverse the second half of the linked list.
3. Compare the first half and second half.
4. Construct the original linked list by reversing the second half again and attaching it back to the first half.

Time Complexity: O(n). Space Complexity: O(1).

Problem-40 Exchange the adjacent elements in a link list.

Solution:

```
public ListNode exchangeAdjacentNodes (ListNode head) {
    ListNode temp = new ListNode(0);
    temp.next = head;
    ListNode prev = temp, curr = head;
    while(curr != null && curr.next != null){
        ListNode tmp = curr.next.next;
        curr.next.next = prev.next;
        prev.next = curr.next;
        curr.next = tmp;
        prev = curr;
        curr = prev.next;
    }
    return temp.next;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-41 For a given K value ($K > 0$) reverse blocks of K nodes in a list.

Example: Input: 1 2 3 4 5 6 7 8 9 10, Output for different K values:
For $K = 2$: 2 1 4 3 6 5 8 7 10 9, For $K = 3$: 3 2 1 6 5 4 9 8 7 10, For $K = 4$: 4 3 2 1 8 7 6 5 9 10

Solution:

Algorithm: This is an extension of swapping nodes in a linked list.

1) Check if remaining list has K nodes.
    a. If yes get the pointer of $K + 1^{th}$ node.
    b. Else return.
2) Reverse first K nodes.
3) Set next of last node (after reversal) to $K + 1^{th}$ node.
4) Move to $K + 1^{th}$ node.
5) Go to step 1.
6) $K - 1^{th}$ node of first K nodes becomes the new head if available. Otherwise, we can return the head.

```
public static ListNode reverseKNodesRecursive(ListNode head, int k){
    ListNode current = head;
    ListNode next = null;
    ListNode prev = null;
    int count = k;
    //Reverse K nodes
    while (current != null && count > 0) {
        next = current.getNext();
        current.setNext(prev);
        prev = current;
        current = next;
        count--;
    }
    //Now next points to K+1 th node, returns the pointer to the head node
    if (next != null) {
        head.setNext(reverseKNodesRecursive(next, k));
    }
    //return head node
    return prev;
}
```

```
public static ListNode reverseKNodes(ListNode head, int k){
    //Start with head
    ListNode current = head;
    //last node after reverse
    ListNode prevTail = null;
    //first node before reverse
    ListNode prevCurrent = head;
    while (current != null) {
        //loop for reversing K nodes
        int count = k;
        ListNode tail = null;
        while (current != null && count > 0) {
            ListNode next = current.getNext();
            current.setNext(tail);
            tail = current;
            current = next;
            count--;
        }
        //reversed K Nodes
        if (prevTail != null) {
            //Link this set and previous set
            prevTail.setNext(tail);
        } else {
            //We just reversed first set of K nodes, update head point to the Kth Node
            head = tail;
        }
        //save the last node after reverse since we need to connect to the next set.
        prevTail = prevCurrent;
        //Save the current node, which will become the last node after reverse
        prevCurrent = current;
    }
    return head;
}
```

Problem-42 Can we solve Problem-39 with recursion?

Solution:

```
public static ListNode reverseKNodes(ListNode head, int k){
    //Start with head
    ListNode current = head;
    //last node after reverse
    ListNode prevTail = null;
    //first node before reverse
    ListNode prevCurrent = head;
    while (current != null) {
        //loop for reversing K nodes
        int count = k;
        ListNode tail = null;
        while (current != null && count > 0) {
            ListNode next = current.getNext();
            current.setNext(tail);
            tail = current;
            current = next;
            count--;
        }
        //reversed K Nodes
        if (prevTail != null) {
            //Link this set and previous set
            prevTail.setNext(tail);
        } else {
            //We just reversed first set of K nodes, update head point to the Kth Node
            head = tail;
        }
        //save the last node after reverse since we need to connect to the next set.
```

```
        prevTail = prevCurrent;
        //Save the current node, which will become the last node after reverse
        prevCurrent = current;
    }
    return head;
}
```

Problem-43 Is it possible to get O(1) access time for Linked Lists?

Solution: Yes. Create a linked list at the same time keep it in a hash table. For n elements we have to keep all the elements into hash table which gives preprocessing time of O(n). To read any element we require only constant time O(1) and to read n elements we require $n * 1$ unit of time $= n$ units. Hence by using amortized analysis we can say that element access can be performed within O(1) time.

Time Complexity – O(1) [Amortized]. Space Complexity - O(n) for Hash.

Problem-44 **JosephusCircle:** N people have decided to elect a leader by arranging themselves in a circle and eliminating every M^{th} person around the circle, closing ranks as each person drops out. Find which person will be the last one remaining (with rank 1).

Solution: Assume the input is a circular linked list with N nodes and each node has a number (range 1 to N) associated with it. The head node has number 1 as data.

```
public ListNode  GetJosephusPosition(int N, int M) {
    ListNode p, q;
    // Create circular linked list containing all the players:
    p.setData(1);
    q = p;
    for (int i = 2; i <= N; ++i) {
            p = p.getNext();
            p.setData(i);
    }
    p.setNext(q);  // Close the circular linked list by having the last node point to the first.
    // Eliminate every M-th player as long as more than one player remains:
    for (int count = N; count > 1; --count) {
            for (int i = 0; i < M - 1; ++i)
                    p = p.getNext();
            p.setNext(p.getNext().getNext());  // Remove the eiminated player from the list.
    }
    System.out.println("Last player left standing (Josephus Position) is " + p.getData());
}
```

Problem-45 Given a linked list consists of data, next pointer and also a random pointer which points to a random node of the list. Give an algorithm for cloning the list.

Solution: We can use the hash table to associate newly created nodes with the instances of node in the given list.

Algorithm:

- Scan the original list and for each node X create a new node Y with data of X, then store the pair (X, Y) in hash table using X as a key. Note that during this scan we set $Y.next$ and $Y.random$ to $NULL$ and we will fix them in the next scan
- Now for each node X in the original list we have a copy Y stored in our hash table. We scan again the original list and set the pointers buildings the new list

```
public RandomListNode copyRandomList(RandomListNode head) {
  RandomListNode X = head, Y;
  Map<RandomListNode, RandomListNode> map = new HashMap<RandomListNode, RandomListNode>();
  while(X != null) {
    Y = new RandomListNode(X.label);
    Y.next = null;
    Y.random = null;
    map.put(X, Y);
    X = X.next;
  }
  X = head;
  while(X != null){
    Y = map.get(X);
    Y.next = map.get(X.next);
```

```
        Y.random = map.get(X.random);
        X = X.next;
    }
    return map.get(head);
}
```

Time Complexity: O(n). Space Complexity: O(n).

Problem-46 In a linked list with n nodes, the time taken to insert an element after an element pointed by some pointer is

(A) O(1) (B) O($logn$) (C) O(n) (D) O($n1ogn$)

Solution: A.

Problem-47 **Find modular node:** Given a singly linked list, write a function to find the last element from the beginning whose $n\%k == 0$, where n is the number of elements in the list and k is an integer constant. For example, if $n = 19$ and $k = 3$ then we should return 18^{th}node.

Solution: For this problem the value of n is not known in advance.

```
public ListNode modularNodes(ListNode head, int k){
    ListNode modularNode;
    int i=0;
    if(k<=0)
        return null;
    for (;head!= null; head = head.getNext()){
        if(i%k == 0){
            modularNode = head;
        }
        i++;
    }
    return modularNode;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-48 **Find modular node from end:** Given a singly linked list, write a function to find the first element from the end whose $n\%k == 0$, where n is the number of elements in the list and k is an integer constant. For example, if $n = 19$ and $k = 3$ then we should return 16^{th}node.

Solution: For this problem the value of n is not known in advance and it is same as finding the k^{th}element from end of the linked list.

```
public ListNode modularNodes(ListNode *head, int k){
    ListNode modularNode=null;
    int i=0;
    if(k<=0)
        return null;
    for (i=0; i < k; i++){
        if(head!=null)
            head = head.getNext();
        else
            return null;
    }
    while(head!= null)
        modularNode = modularNode.getNext();
        head = head.getNext();
    }
    return modularNode;
}
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-49 **Find fractional node:** Given a singly linked list, write a function to find the $\frac{n}{k}th$ element, where n is the number of elements in the list.

Solution: For this problem the value of n is not known in advance.

```
public ListNode fractionalNodes(ListNode head, int k){
```

```
        ListNode fractionalNode;
        int i=0;
        if(k<=0)
            return null;
        for (;head!= null; head = head.getNext()){
            if(i%k == 0){
                if(fractionalNode!=null)
                    fractionalNode = head;
                else fractionalNode = fractionalNode.getNext();
            }
            i++;
        }
        return fractionalNode;
    }
```

Time Complexity: O(n). Space Complexity: O(1).

Problem-50 Median in an infinite series of integers

Solution: Median is the middle number in a sorted list of numbers (if we have odd number of elements). If we have even number of elements, median is the average of two middle numbers in a sorted list of numbers.

We can solve this problem with linked lists (with both sorted and unsorted linked lists).

First, let us try with *unsorted* linked list. In an unsorted linked list, we can insert the element either at the head or at the tail. The disadvantage with this approach is that, finding the median takes O(n). Also, the insertion operation takes O(1).

Now, let us with *sorted* linked list. We can find the median in O(1) time if we keep track of the middle elements. Insertion to a particular location is also O(1) in any linked list. But, finding the right location to insert is not O($logn$) as in sorted array, it is instead O(n) because we can't perform binary search in a linked list even if it is sorted.

So, using a sorted linked list doesn't worth the effort, insertion is O(n) and finding median is O(1), same as the sorted array. In sorted array insertion is linear due to shifting, here it's linear because we can't do binary search in a linked list.

Note: For efficient algorithm refer *Priority Queues and Heaps* chapter.

Problem-51 Consider the following Java program code, whose runtime F is a function of the input size n.

```
java.util.ArrayList<Integer> list = new java.util.ArrayList<Integer>();
for( int i = 0 ; i < n; i ++)
    list.add (0 , i ) ;
```

Which of the following is correct?

A) F(n)=Θ(n) B) F(n)=Θ(n^2) C) F(n)=Θ(n^3) D) F(n)=Θ(n^4) F) F(n)=Θ($nlogn$)

Solution: B. The operation list.add(0,i) on ArrayList has linear time complexity, with respect to the current size of the data structure. Therefore, overall we have quadratic time complexity.

Problem-52 Consider the following Java program code, whose runtime F is a function of the input size n.

```
java.util.ArrayList<Integer> list = new java.util.ArrayList<Integer>();
for(int i = 0 ; i < n; i ++)
    list.add ( i , i ) ;
for(int j = 0 ; j < n; j++)
    list.remove(n-j -1 );
```

Which of the following is correct?

A) F(n)=Θ(n) B) F(n)=Θ(n^2) C) F(n)=Θ(n^3) D) F(n)=Θ(n^4) F) F(n)=Θ($nlogn$)

Solution: A. Both loop bodies have constant time complexity because they operate the end of the ArrayList.

Problem-53 Consider the following Java program code, whose runtime F is a function of the input size n.

```
    java.util.LinkedList<Integer> k = new java.util.LinkedList<Integer>();
    for(int i = 0 ; i < n; i ++)
       for(int j = 0 ; j < n; j++)
            k.add(k.size()/2,j);
```

Which of the following is correct?

Solution: D. The LinkedList grows to quadratic size during the execution of the program fragment. Thus the body of the inner loop has quadratic complexity. The inner loop itself is executed n^2 times.

Problem-54 Given a singly linked list L: L_1-> L_2-> L_3…-> L_{n-1}-> L_n, reorder it to: L_1-> L_n-> L_2-> L_{n-1}……

Solution: We divide the list in two parts for instance 1->2->3->4->5 will become 1->2->3 and 4->5, we have to reverse the second list and give a list that alternates both. The split is done with a slow and fast pointer. First solution (using a stack for reversing the list):

```
public class ReorderLists {
   public void reorderList(ListNode head) {
           if(head == null)
          return;
      ListNode slowPointer = head;
      ListNode fastPointer = head.next;
      while(fastPointer!=null && fastPointer.next !=null){
         fastPointer = fastPointer.next.next;
         slowPointer = slowPointer.next;
      }
      ListNode head2 = slowPointer.next;
      slowPointer.next = null;
      LinkedList<ListNode> queue = new LinkedList<ListNode>();
      while(head2!=null){
         ListNode temp = head2;
         head2 = head2.next;
         temp.next =null;
         queue.push(temp);
      }
      while(!queue.isEmpty()){
         ListNode temp = queue.pop();
         temp.next = head.next;
         head.next = temp;
         head = temp.next;
      }
   }
}
```

Alternative Approach:

```
public class ReorderLists {
   public void reorderList(ListNode head) {
      if(head == null)
         return;
      ListNode slowPointer = head;
      ListNode fastPointer = head.next;
      while(fastPointer!=null && fastPointer.next !=null){
         fastPointer = fastPointer.next.next;
         slowPointer = slowPointer.next;
      }
      ListNode head2 = slowPointer.next;
      slowPointer.next = null;
      head2= reverseList(head2);
      alternate (head, head2);
   }
   private ListNode reverseList(ListNode head){
     if (head == null)
         return null;
      ListNode reversedList = head;
      ListNode pointer = head.next;
      reversedList.next=null;
      while (pointer !=null){
         ListNode temp = pointer;
         pointer = pointer.next;
         temp.next = reversedList;
```

```
            reversedList = temp;
        }
        return reversedList;
    }
    private void alternate (ListNode head1, ListNode head2){
        ListNode pointer = head1;
        head1 = head1.next;
        boolean nextList1 = false;
        while(head1 != null || head2 != null){
            if((head2 != null && !nextList1) || (head1==null)){
                pointer.next = head2;
                head2 = head2.next;
                nextList1 = true;
                pointer = pointer.next;
            }
            else {
                pointer.next = head1;
                head1 = head1.next;
                nextList1 = false;
                pointer = pointer.next;
            }
        }
    }
}
```

Time Complexity: O(*n*). Space Complexity: O(1).

Problem-55 Which sorting algorithm is easily adaptable to singly linked lists?

Solution: Simple Insertion sort is easily adabtable to singly linked list. To insert the an element, the linked list is traversed until the proper position is found, or until the end of the list is reached. It be inserted into the list by merely adjusting the pointers without shifting any elements unlike in the array. This reduces the time required for insertion but not the time required for searching for the proper position.

Problem-56 How do you implement insertion sort for linked lists?

Solution:

```
public static ListNode insertionSortList(ListNode head) {
    if (head == null || head.next == null)
        return head;
    ListNode newHead = new ListNode(head.data);
    ListNode pointer = head.next;
    // loop through each element in the list
    while (pointer != null) {
        // insert this element to the new list
        ListNode innerPointer = newHead;
        ListNode next = pointer.next;
        if (pointer.data <= newHead.data) {
            ListNode oldHead = newHead;
            newHead = pointer;
            newHead.next = oldHead;
        } else {
            while (innerPointer.next != null) {

                if (pointer.data > innerPointer.data && pointer.data <= innerPointer.next.data) {
                    ListNode oldNext = innerPointer.next;
                    innerPointer.next = pointer;
                    pointer.next = oldNext;
                }
                innerPointer = innerPointer.next;
            }
            if (innerPointer.next == null && pointer.data > innerPointer.data) {
                innerPointer.next = pointer;
                pointer.next = null;
            }
```

```
                }
                // finally
                pointer = next;
        }
        return newHead;
    }
```

Note: For details on insertion sort refer Sorting chapter.

Problem-57 Given a list, rotate the list to the right by k places, where k is non-negative. For example: Given 1->2->3->4->5->NULL and k = 2, return 4->5->1->2->3->NULL.

Solution:

```
public ListNode rotateRight(ListNode head, int n) {
    if(head == null || head.next == null)
        return head;
    ListNode rotateStart = head, rotateEnd = head;
    while(n-- > 0){
        rotateEnd = rotateEnd.next;
        if(rotateEnd == null){
            rotateEnd = head;
        }
    }
    if(rotateStart == rotateEnd)
        return head;
    while(rotateEnd.next != null){
        rotateStart = rotateStart.next;
        rotateEnd = rotateEnd.next;
    }
    ListNode temp = rotateStart.next;
    rotateEnd.next = head;
    rotateStart.next = null;
    return temp;
}
```

Time Complexity: O(*n*). Space Complexity: O(1).

Problem-58 You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list. For example with input: (3 -> 4 -> 3) + (5 -> 6 -> 4); the output should be 8 -> 0 -> 8.

Solution:

```
public ListNode addTwoNumbers(ListNode list1, ListNode list2) {
    if(list1 == null)
        return list2;
    if(list2 == null)
        return list1;
    ListNode head = new ListNode(0);
    ListNode cur = head;
    int advance = 0;
    while(list1 != null && list2 != null){
        int sum = list1.data + list2.data + advance;
        advance = sum / 10;
        sum = sum % 10;
        cur.next = new ListNode(sum);
        cur = cur.next;
        list1 = list1.next;
        list2 = list2.next;
    }
    if(list1 != null){
        if(advance != 0)
            cur.next = addTwoNumbers(list1, new ListNode(advance));
        else
            cur.next = list1;
    }else if(list2 != null){
```

```
        if(advance != 0)
            cur.next = addTwoNumbers(list2, new ListNode(advance));
        else
            cur.next = list2;
    }else if(advance != 0){
        cur.next = new ListNode(advance);
    }
    return head.next;
}
```

Problem-59 Given a linked list and a value K, partition it such that all nodes less than K come before nodes greater than or equal to K. You should preserve the original relative order of the nodes in each of the two partitions. For example, given 1->4->3->2->5->2 and K = 3, return 1->2->2->4->3->5.

Solution:

```
public ListNode partition(ListNode head, int K) {
    ListNode root = new ListNode(0);
    ListNode pivot = new ListNode(K);
    ListNode rootNext = root, pivotNext = pivot;
    ListNode currentNode = head;
    while(currentNode != null){
        ListNode next = currentNode.next;
        if(currentNode.data >= K){
            pivotNext.next = currentNode;
            pivotNext = currentNode;
            pivotNext.next = null;
        }else{
            rootNext.next = currentNode;
            rootNext = currentNode;
        }
        currentNode = next;
    }
    rootNext.next = pivot.next;
    return root.next;
}
```

Time Complexity: O(*n*). Space Complexity: O(1).

Problem-60 Merge k sorted linked lists and return it as one sorted list.

Solution: Refer Priority Queues chapter.

Problem-61 Given a unordered linked list, how do you remove duplicates in it?

Solution:

```
public static void removeDuplicates2(ListNode head) {
        ListNode curr = head;
        if(curr == null || curr.getNext() == null) {
                return; // 0 or 1 nodes in the list so no duplicates
        }
        ListNode curr2;
        ListNode prev;
        while(curr != null) {
                curr2 = curr.getNext();
                prev = curr;
                while(curr2 != null) {
                        // check and see if curr and curr2 values are the same, if they are then delete curr2
                        if(curr.getData()==curr2.getData()) {
                                prev.setNext(curr2.getNext());
                        }
                        prev = curr2;
                        curr2 = curr2.getNext();
                }
                curr = curr.getNext();
        }
}
```

Time Complexity: O(n^2). Space Complexity: O(1).

Problem-62 Can reduce the time complexity of Problem-61?

Solution: We can simply use hash table and check whether an element already exist.

```
// using a temporary buffer O(n)
public static void removeDuplicates(ListNode head) {
        Map<Integer, Boolean> mapper = new HashMap<Integer, Boolean>();
        ListNode curr = head;
        ListNode next;
        while (curr.getNext() != null) {
                next = curr.getNext();
                if(mapper.get(next.getData()) != null) {
                        // already seen it, so delete
                        curr.setNext(next.getNext());
                } else {
                        mapper.put(next.getData(), true);
                        curr = curr.getNext();
                }
        }
}
```

Time Complexity: O(n). Space Complexity: O(n) for hash table.