Linked List

- A linked list is a linear data structure.
- Linked list is the second most-used data structure after array.
- Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
- There are many variants of linked lists: singly linked list, doubly linked list and circular linked list. Singly linked list

Singly linked list

- A singly linked list is a data structure that contains a sequence of nodes such that each node contains an object and a reference to the next node in the list.
- A node contains two fields, one is data field and another is next field.
    - Data field : which store a data called an element or an object.
    - Next field : which contains the address of the next node in the list.
- The first node is referred to as the head.
- The last node is referred to as the tail; the tail's next field is null.
- The singly linked list can be traversed only in one direction. Because each node contains only next pointer, therefore traversing the list in the reverse direction is not possible.
- The structure of a singly linked list is shown in below figure.

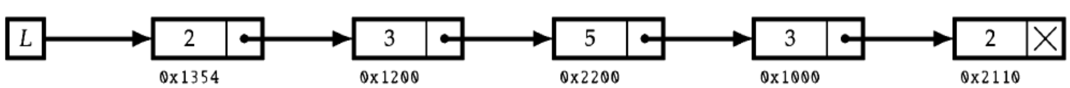


Figure 1: Example of a singly linked list.

Doubly linked list

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the list.

- A node contains three fields, data field, previous field and next field.

    - Data field : which store a data called an element or an object.

    - Next field : which contains the address of the next node in the list.

    - Previous field : which contains the address of the previous node in the list.

- The first node is referred to as the head; the head's previous field is null.

- The last node is referred to as the tail; the tail's next field is null.

- The structure of a doubly linked list is shown in below figure.

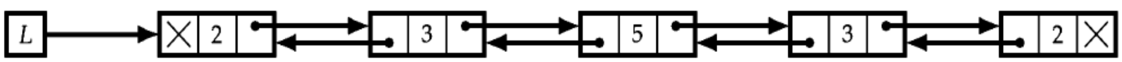


Figure 2: Example of a doubly linked list.

Program:

```
package SimpleLinkedList;

// Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}
```

```java
package SimpleLinkedList;

public class LinkedList {

	// Create a new Linked List node and insert the nodes at first
	public static Node<Integer> insert(int d, Node node){

		Node<Integer> newNode = new Node<Integer>(d);
		newNode.next = node.next;
		node.next = newNode;
		return node;
	}

	// Insert newNode after given node.
	public static void insertAfter(Node<Integer> node, Node<Integer> newNode) {

		newNode.next = node.next;
		node.next = newNode;
	}


	// This function delete the an element next to head
	public static void deleteList(Node aNode) {

		aNode.next = aNode.next.next;
	}

	// returns the position of the first occurrence of the given
	// value (-1 if not found)
	public static int search (Node L, Integer key) {
		int index = 0;
		Node current = L;
		while (current != null) {
			if (current.data == key) {
				return index;
			}
			current = current.next;
			index++;
		}
		return -1;
	}
```

```
//return the size of the linked list
public static int size(Node head) {

	// initialize the count
	int count = 0;

	// if no element in the list, re
	if(head == null) {

		return count;
	}
	else {

		//count = 0;
		Node node = head;
		while(node.next != null) {

			count++;
			node = node.next;
		}
		return count;
	}
}

//Display the list
public static void display(Node node) {

	// display the linked list header/number,
	// for example, LinledList 1 or LinledList 2
	System.out.print("LinkedList " + node.data + " : " );

	//Display the data present in the list
	while(node.next != null) {

		System.out.print(node.next.data + " --> ");
		node = node.next;
	}

	System.out.println("null");
}

public static void main(String[] args) {

	//Create head node in Linked list with header/number,
	// for example, LinledList 1 or LinledList
	Node<Integer> head = new Node<>(1);
```

```
        //insert 4 element in the list
        for(int i=0;i<4;i++) {

            head=insert(i+2,head);
        }

        System.out.println("The elements in the linked list: " );
        display(head);

        int length = size(head);
        System.out.println("The size of the linked list " + length);

        // Create a node and its data is 7. Insert the node
        // after second  node (position 3 in list) from head
        System.out.println("Insert a new node after a specified node in the linked list: ");
        Node<Integer> newNode = new Node<Integer>(7);
        insertAfter(head.next.next, newNode);
        display(head);

        //element next to head will be deleted
        System.out.println("The linked list, after deletion of element next to head: ");
        deleteList(head);
        display(head);

        //returns the position of the first occurrence of searching element
        int position = search(head, 3);
        if(position > 0)
            System.out.println("The searching element present in Linked List at
                                        position: " + position);
        else
            System.out.println("The searching element is not present in Linked List.");

    }
}
```

Output:

The elements in the linked list:
LinkedList 1 : 5 --> 4 --> 3 --> 2 --> null
The size of the linked list 4
Insert a new node after a specified node in the linked list:
LinkedList 1 : 5 --> 4 --> 7 --> 3 --> 2 --> null
The linked list, after deletion of element next to head:
LinkedList 1 : 4 --> 7 --> 3 --> 2 --> null
The searching element present in Linked List at position: 3

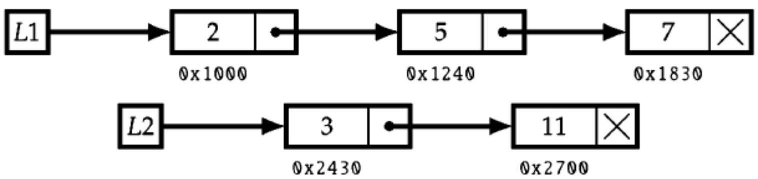
Time and Space Complexity:

- Insert and delete are local operations and have O*(1)* time complexity. Search requires traversing the entire list, e.g., if the key is at the last node or is absent, so its time complexity is *0(n)*, where *n* is the number of nodes.

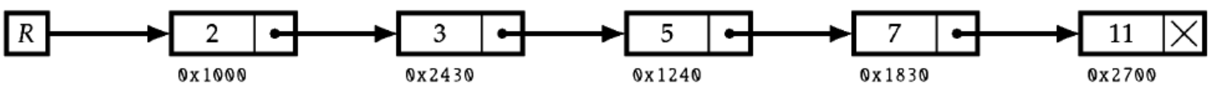
MERGE TWO SORTED LISTS

- Consider two singly linked lists in which each node holds a number. Assume the lists are sorted, i.e., numbers in the lists appear in ascending order within each list.

- The merge of the two lists is a list consisting of the nodes of the two lists in which numbers appear in ascending order.



(a) Two sorted lists.



(b) The merge of the lists in (a).

Figure 2: Merging sorted lists.

Write a program that takes two sorted linked lists, and returns their merge. The merge list also a sorted list. The list should be made by splicing together the nodes of the first two lists.

Example:

- Let consider the following two sorted linked list for input to the program

  LinkedList 1: 1 --> 3 --> 5 --> 7 --> null

  LinkedList 2: 2 --> 5 --> 8 --> null

- The output is a merged list:

  LinkedList 3: 1 --> 2 --> 3 --> 5 --> 5 --> 7 --> 8 --> null

Program:

```
package MergeTwoSortedList;

//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}
```

```
package MergeTwoSortedList;

public class MergeTwoSortedListProg1 {

	// Create a new Linked List node and insert the nodes at first
	public static Node<Integer> insert(int d, Node node){

		Node<Integer> newNode = new Node<Integer>(d);
		newNode.next = node.next;
		node.next = newNode;
		return node;
	}

	//Display the list
	public static void display(Node node) {

		// display the linked list header/number,
		// for example, LinledList 1 or LinledList 2
		System.out.print("LinkedList " + node.data + " : " );

		//Display the data present in the list
		while(node.next != null) {

			System.out.print(node.next.data + " --> ");
```

```
            node = node.next;
        }

        System.out.println("null");
    }


public static Node<Integer>mergeTwoSortedLists(Node l1, Node l2){

        // Create a dummy node to hang the result on
        Node<Integer> dummy = new Node<>(3);

        // current pointer points to the last result node
        Node<Integer> current = dummy;

        // index p1 point to linked list 1
        Node<Integer> p1 = l1.next;

        // index p2 point to linked list 2
        Node<Integer> p2 = l2.next;

        //traverse the two lists
        while(p1 != null && p2 != null) {

            // Compare the data of the two lists whichever
            // lists' data is smaller, append it into tail
            // and advance the curr pointer to the next Node
             if(p1.data < p2.data) {

                current.next = p1;
                p1 = p1.next;
            }

            else {

                current.next = p2;
                p2 = p2.next;
            }
            current = current.next;
        }

        // Appends the remaining nodes of pi or p2.
        if(p1 != null) {

            current.next = p1;
        }
```

```
        else {

            current.next = p2 ;
        }

        return dummy;
    }

    public static void main(String[] args) {

        // create first sorted linked, i.e.,, LinledList 1
        // insert 4 element in the list, ascending order
        Node<Integer> head1=new Node<>(1);
        for(int i = 7; i > 0 ; i-=2) {
            head1=insert(i,head1);
        }
        // Display first linked list
        display(head1);

        // create second sorted linked, i.e.,, LinledList 2
        // insert 3 element in the list, ascending order
        Node<Integer> head2=new Node<>(2);
        for(int j = 8; j > 0 ; j-=3) {
            head2=insert(j,head2);
        }

        // Display second linked list
        display(head2);

        //Create list3
        Node<Integer> head3=new Node<>(3);
        head3 = mergeTwoSortedLists(head1, head2);
        display(head3);
    }
}
```

Output:

LinkedList 1 : 1 --> 3 --> 5 --> 7 --> null
LinkedList 2 : 2 --> 5 --> 8 --> null
LinkedList 3 : 1 --> 2 --> 3 --> 5 --> 5 --> 7 --> 8 --> null

Time and Space Complexity:

- The worst-case, the time complexity is *0(n + m)*. Since we reuse the existing nodes, the space complexity is *0(1)*.

REVERSE A SINGLE SUBLIST

Program:

```
package ReverseSingleSublist;

//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}



package ReverseSingleSublist;

public class ReverseSingleSublistProg1 {

	// Create a new Linked List node and insert the nodes at first
	public static Node<Integer> insert(int d, Node node){

		Node<Integer> newNode = new Node<Integer>(d);
		newNode.next = node.next;
		node.next = newNode;
		return node;
	}

	//Display the list
	public static void display(Node node) {

		// display the linked list header/number,
		// for example, LinledList 1 or LinledList 2
		System.out.print("LinkedList " + node.data + ": " );

		//Display the data present in the list
		while(node.next != null) {

			System.out.print(node.next.data + " --> ");
			node = node.next;
```

```
        }

        System.out.println("null");
}


// reverseSublist() method used to reverse a linked list
public static Node<Integer> reverseSublist(Node<Integer> L, int start, int finish) {

        // No need to reverse since start == finish.
        if (start == finish) {

                return L;
        }

        Node<Integer> dummyHead = new Node<>(0);
        dummyHead.next = L;
        Node<Integer> sublistHead = dummyHead;
        int k = 1;
        while (k++ < start) {

                sublistHead = sublistHead.next ;
        }

        // Reverse sublist.
        Node<Integer> sublistlter = sublistHead.next ;

        while (start++ < finish) {

                Node<Integer> temp = sublistlter.next ;
                sublistlter.next = temp.next;
                temp.next = sublistHead.next ;
                sublistHead.next = temp;
        }

        return dummyHead.next ;
}

public static void main(String[] args) {

        // create first sorted linked, i.e.,, LinledList 1
        // insert 4 element in the list, ascending order
        Node<Integer> head1= new Node<>(1);
        for(int i = 14; i > 0 ; i-=2) {
                head1=insert(i,head1);
        }
```

```
            // Display the elements in linked list
            System.out.println("The elements in the linked list: " );
            display(head1);

            // select node 2 to node 5 as sublist and reverse
            System.out.println("After reverring sublist, the elements in the linked list: " );
            reverseSublist(head1.next, 2, 5);
            // Display first linked list
            display(head1);
        }
}
```

Output:

The elements in the linked list:
LinkedList 1: 2 --> 4 --> 6 --> 8 --> 10 --> 12 --> 14 --> null
After reverring sublist, the elements in the linked list:
LinkedList 1: 2 --> 10 --> 8 --> 6 --> 4 --> 12 --> 14 --> null

Time and Space Complexity:

- The time complexity is dominated by the search for the end node (let it's position m in the linked list), hence the time complexity is *0(m).*

DELETE A NODE FROM A SINGLY LINKED LIST

Write a program which deletes a node in a singly linked list. It is guaranteed that the node to be deleted is not a tail node in the list.

Example:

- The elements in the linked list:

  LinkedList 1: 0 --> 1 --> 3 --> 8 --> 5 --> null

- After Deleting kth elements from start of the linked list:

  LinkedList 1: 0 --> 3 --> 8 --> 5 --> null

Approach:

- Instead traverses the linked list from head to find the node to be deleted, direct access to the node to be deleted.

- After access to the node to be deleted, first copy the data from the next node to the node to be deleted and delete the next node.

Program:

```
package DeleteNodeFromLinkedList;

//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}


package DeleteNodeFromLinkedList;

import java.util.Random;

public class DeleteNodeFromLinkedListProg1 {

		// Create a new Linked List node and insert the nodes at first
		public static Node<Integer> insert(int d, Node node){

			Node<Integer> newNode = new Node<Integer>(d);
			newNode.next = node.next;
			node.next = newNode;
			return node;
		}

		//Display the list
		public static void display(Node node) {

			// display the linked list header/number,
			// for example, LinledList 1 or LinledList 2
			System.out.print("LinkedList " + node.data + ": " );

			//Display the data present in the list
			while(node.next != null) {
```

```
                System.out.print(node.next.data + " --> ");
                node = node.next;
            }

            System.out.println("null");
        }

        // Assumes nodeToDelete is not tail.
        public static void deletionFromList(Node<Integer> nodeToDelete) {

            nodeToDelete.data = nodeToDelete.next.data ;
            nodeToDelete.next = nodeToDelete.next.next ;
        }


        public static void main(String[] args) {

            // create instance of Random class
            Random rand = new Random();

            // create linked list, i.e.,, LinledList 1
            // insert 5 element in the list
            Node<Integer> head1= new Node<>(1);
            for(int i = 0; i < 5 ; i++) {

                int num = rand.nextInt(10);
                head1=insert(num,head1);
            }
            // Display the elements in linked list
            System.out.println("The elements in the linked list: " );
            display(head1);

          //Delete kth element
           deletionFromList(head1.next.next);
           // Display the elements in linked list
           System.out.println("After Deleting kth node from start of the linked list:  " );
           display(head1);
        }
}
```

Output:

The elements in the linked list:
LinkedList 1: 0 --> 1 --> 3 --> 8 --> 5 --> null
After Deleting kth elements from start of the linked list:
LinkedList 1: 0 --> 3 --> 8 --> 5 --> null

Time and Space Complexity:

- The time complexity is O(1).

REMOVE THE Kth LAST ELEMENT FROM A LIST

Given a singly linked list and an integer *k*, write a program to remove the kth last element from the list.

Example:

- Let consider the following linked list and the k = 3, for input to the program

```
LinkedList 1: 8 --> 2 --> 6 --> 5 --> 1 --> null
```

- After remove 3rd element (i.e., 6) from the end of the list, the output

```
LinkedList 1: 8 --> 2 --> 5 --> 1 --> null
```

Approach:

- This approach, use two iterators to traverse the list.
- The first iterator is advanced by *k* steps, and then the second iterators advance in tandem.
- When the first iterator reaches the tail, the second iterator is at the $(k + 1)^{th}$ last node, and then remove the Kth node.

Program:

```
package RemoveKthLastElementFromList;

//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}
```

```
package RemoveKthLastElementFromList;

import java.util.Random;

public class RemoveKthLastElementFromListProg1 {

	// Create a new Linked List node and insert the nodes at first
	public static Node<Integer> insert(int d, Node node){

		Node<Integer> newNode = new Node<Integer>(d);
		newNode.next = node.next;
		node.next = newNode;
		return node;
	}

	//Display the list
	public static void display(Node node) {

		// display the linked list header/number,
		// for example, LinledList 1 or LinledList 2
		System.out.print("LinkedList " + node.data + ": " );

		//Display the data present in the list
		while(node.next != null) {

			System.out.print(node.next.data + " --> ");
			node = node.next;
		}

		System.out.println("null");
	}

	// Assumes L has at least k nodes, deletes the k-th last node in L.
	public static Node<Integer> removeKthLast(Node <Integer> L, int k) {

		Node <Integer> dummyHead = new Node<>(0);
		dummyHead.next = L;

		Node <Integer> first = dummyHead.next ;

		while (k-- > 0) {

			first = first.next;
		}
```

```
        Node <Integer> second = dummyHead;

        while (first != null) {

            second = second. next;
            first = first.next;
        }

        // second points to the (k + l)-th last node, deletes its successor.
        second.next = second.next.next ;
        return dummyHead.next ;
    }


    public static void main(String[] args) {

        // create instance of Random class
        Random rand = new Random();

        // create linked list, i.e.,, LinledList 1
        // insert 5 element in the list.
        Node<Integer> head1= new Node<>(1);
        for(int i = 0; i < 5 ; i++) {

            int num = rand.nextInt(10);
            head1=insert(num,head1);
        }

        // Display the elements in linked list
        System.out.println("The elements in the linked list: " );
        display(head1);


        System.out.println("After removing kth elements from end of the linked list: " );
        removeKthLast(head1.next, 3);
        // Display the linked list
        display(head1);
    }
}
```

Output:

The elements in the linked list:
LinkedList 1: 8 --> 2 --> 6 --> 5 --> 1 --> null
After removing kth elements from end of the linked list:
LinkedList 1: 8 --> 2 --> 5 --> 1 --> null

Time and Space Complexity:

- The time complexity is that of list traversal, i.e.,*O(n)*,where *n* is the length of the list. The space complexity is*0(1)*,since there are only two iterators.

REMOVE DUPLICATES FROM A SORTED LIST

- This problem is concerned with removing duplicates from a sorted list of integers.

Write a program that takes as input a singly linked list of integers in sorted order, and removes duplicates from it. The list should be sorted.

- Let the input the sorted linked list:

LinkedList 1 : 2 --> 5 --> 7 --> 7 --> 11 --> null

- The output is deletion of the duplicate elements from the linked list:

LinkedList 1 : 2 --> 5 --> 7 --> 11 --> null

Approach:

- Exploit the sorted nature of the list.

- As traverse the list, remove all successive nodes with the same value as the current node.

Program:

```
//Linked list Node
public class Node<T>{

	public T data;
	public Node<T> next;

	//Constructor
	Node(T d){

		data = d;
		next = null;
	}
}
```

```java
public class RemoveDuplicatesFromSortedListProg1 {

	// Create a new Linked List node and insert the nodes at first
	public static Node<Integer> insert(int d, Node node){

		Node<Integer> newNode = new Node<Integer>(d);
		newNode.next = node.next;
		node.next = newNode;
		return node;
	}

	//Display the list
	public static void display(Node node) {

		// display the linked list header/number,
		// for example, LinledList 1 or LinledList 2
		System.out.print("LinkedList " + node.data + " : " );

		//Display the data present in the list
		while(node.next != null) {

			System.out.print(node.next.data + " --> ");
			node = node.next;
		}

		System.out.println("null");
	}



	public static Node <Integer> removeDuplicates (Node <Integer> L) {

		Node <Integer> iter = L;

		while (iter != null) {

			// Uses nextDistinct to find the next distinct value.
			Node <Integer> nextDistinct = iter. next;

			while (nextDistinct != null && nextDistinct . data == iter.data) {

				nextDistinct = nextDistinct . next ;
			}

			iter.next = nextDistinct ;
			iter = nextDistinct;
```

```
        }

        return L;
    }

    public static void main(String[] args) {

        // create sorted linked, i.e.,, LinledList 1
        // insert 5 element in the list, ascending order
        Node<Integer> head1=new Node<>(1);
        head1=insert(11, head1);
        head1=insert(7, head1);
        head1=insert(7, head1);
        head1=insert(5, head1);
        head1=insert(2, head1);

        System.out.println("The elements in linked list: ");
        // Display linked list
        display(head1);

       System.out.println("After deletion of the duplicate elements from the linked list: ");
       removeDuplicates(head1);
       // Display linked list
       display(head1);
    }
}
```

Output:

The elements in linked list:
LinkedList 1 : 2 --> 5 --> 7 --> 7 --> 11 --> null
After deletion of the duplicate elements from the ascending order sorted linked list:
LinkedList 1 : 2 --> 5 --> 7 --> 11 --> null

Time and Space Complexity:

- The linked list is traversed once, hence the time complexity is O*(n).* The space complexity is O*(1).*

TEST WHETHER A SINGLY LINKED LIST IS PALINDROMIC

- A palindromic list is the one which is equivalent to the reverse of itself.

- The list given in the below figure is a palindrome since it is equivalent to its reverse list, i.e., 2, 3, 5, 3, 2.

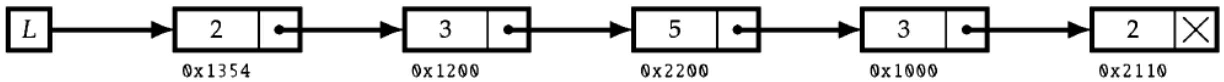


Figure 3: Palindromic List

Write a program that tests whether a singly linked list is palindromic.

Example:

Approach:

- First find the middle of the linked list.
  - Traverse the linked list by using two pointers, i.e., fast and slow.
  - Move fast pointer by one step and the slow pointer by two.
  - When the fast pointer reaches the end slow pointer will reach the middle of the linked list.
  - Now the list is divided into two parts or sub list, i.e., first half and second half.
- Reverse the second half of the list
- If the first half and the reversed second half are equal, then the linked list is palindrome otherwise not palindrome.
- To restore the original list reverse the reversed second half.