1. Implement Node Class

public class Node<T> {

private T data;

private Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

public T getData() {

return data;

}

public void setData(T data) {

this.data = data;

}

public Node<T> getNext() {

return next;

}

public void setNext(Node<T> next) {

this.next = next;

}

}

1. Generics

public class Box<T> {

private T contents;

public void setContents(T contents) {

this.contents = contents;

}

public T getContents() {

return contents;

}

}

Box<Integer> integerBox = new Box<>();

integerBox.setContents(42);

int value = integerBox.getContents();

1. Implement SinglyLinkedList Class

public class SinglyLinkedList<T> {

private Node<T> head;

private int size;

public SinglyLinkedList() {

head = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int getSize() {

return size;

}

public void addFirst(T data) {

Node<T> newNode = new Node<>(data);

newNode.setNext(head);

head = newNode;

size++;

}

public void addLast(T data) {

Node<T> newNode = new Node<>(data);

if (isEmpty()) {

head = newNode;

} else {

Node<T> current = head;

while (current.getNext() != null) {

current = current.getNext();

}

current.setNext(newNode);

}

size++;

}

public void removeFirst() {

if (isEmpty()) {

throw new NoSuchElementException("List is empty.");

}

head = head.getNext();

size--;

}

public void removeLast() {

if (isEmpty()) {

throw new NoSuchElementException("List is empty.");

}

if (size == 1) {

head = null;

} else {

Node<T> current = head;

while (current.getNext().getNext() != null) {

current = current.getNext();

}

current.setNext(null);

}

size--;

}

1. Implement Basic Methods of SinglyLinkedList

* isEmpty()
* size()
* first()
* last()
* addFirst()
* addLast()
* removeFirst()

public class SinglyLinkedList<T> {

private Node<T> head;

private int size;

public SinglyLinkedList() {

head = null;

size = 0;

}

public boolean isEmpty() {

return size == 0;

}

public int size() {

return size;

}

public T first() {

if (isEmpty()) {

throw new NoSuchElementException("List is empty.");

}

return head.getData();

}

public T last() {

if (isEmpty()) {

throw new NoSuchElementException("List is empty.");

}

Node<T> current = head;

while (current.getNext() != null) {

current = current.getNext();

}

return current.getData();

}

public void addFirst(T data) {

Node<T> newNode = new Node<>(data);

newNode.setNext(head);

head = newNode;

size++;

}

public void addLast(T data) {

Node<T> newNode = new Node<>(data);

if (isEmpty()) {

head = newNode;

} else {

Node<T> current = head;

while (current.getNext() != null) {

current = current.getNext();

}

current.setNext(newNode);

}

size++;

}

public void removeFirst() {

if (isEmpty()) {

throw new NoSuchElementException("List is empty.");

}

head = head.getNext();

size--;

## Homework

1. develop an implementation of the equals method in the context of the SinglyLinkedList class.

concatenateLists(z, y):

if z is empty:

return y

if yis empty:

return z

current = z.head

while current.next is not null:

current = current.next

current.next = y.head

z' = z

return z'

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

SinglyLinkedList<?> otherList = (SinglyLinkedList<?>) obj;

if (size() != otherList.size()) {

return false;

}

Node<T> currentThis = head;

Node<?> currentOther = otherList.head;

while (currentThis != null) {

if (!currentThis.getData().equals(currentOther.getData())) {

return false;

}

currentThis = currentThis.getNext();

currentOther = currentOther.getNext();

}

return true;

}

1. Give an algorithm for finding the second-to-last node in a singly linked list in which the last node is indicated by a null next reference.

public class LinkedList {

private Node head;

public Node findSecondToLastNode() {

if (head == null || head.next == null) {

return null;

}

Node current = head;

Node previous = null;

while (current.next != null) {

pre = current;

current = current.next;

}

return pre;

}

private class Node {

private int data;

private Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

}

public class LinkedList {

private Node head;

public Node findSecondToLastNode() {

if (head == null || head.next == null) {

return null;

}

Node current = head;

Node previous = null;

while (current.next != null) {

previous = current;

current = current.next;

}

return previous;

}

private class Node {

private int data;

private Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

}

1. Give an implementation of the size( ) method for the SingularlyLinkedList class, assuming that we did not maintain size as an instance variable.

public class SingularlyLinkedList {

private Node head;

public int size() {

int count = 0;

Node current = head;

while (current != null) {

count++;

current = current.next;

}

return count;

}

private class Node {

private int data;

private Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

}

public int size() {

int count = 0;

Node current = head;

while (current != null) {

count++;

current = current.next;

}

return count;

}

1. Describe an algorithm for concatenating two singly linked lists L and M, into a single list L′ that contains all the nodes of L followed by all the nodes of M.

public class SinglyLinkedList {

private Node head;

public void concatenate(SinglyLinkedList otherList) {

if (head == null) {

head = otherList.head;

} else {

Node current = head;

while (current.next != null) {

current = current.next;

}

current.next = otherList.head;

}

}

private class Node {

private int data;

private Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

}

1. Implement a rotate( ) method in the SinglyLinkedList class, which has semantics equal to addLast(removeFirst( )), yet without creating any new node.

public class SinglyLinkedList {

private Node head;

public void rotate() {

if (head == null || head.next == null) {

return;

}

Node previous = null;

Node current = head;

while (current.next != null) {

previous = current;

current = current.next;

}

current.next = head;

head = current;

previous.next = null;

}

private class Node {

private int data;

private Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

}

public void rotate() {

if (head == null || head.next == null) {

return;

}

Node previous = null;

Node current = head;

while (current.next != null) {

previous = current;

current = current.next;

}

current.next = head;

head = current;

previous.next = null;

}

1. Describe in detail an algorithm for reversing a singly linked list L using only a constant amount of additional space.

public class SinglyLinkedList {

private Node head;

public void reverse() {

if (head == null || head.next == null) {

return;

}

Node previous = null;

Node current = head;

while (current != null) {

Node next = current.next;

current.next = previous;

previous = current;

current = next;

}

head = previous;

}

private class Node {

private int data;

private Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

}

public class SinglyLinkedList {

private Node head;

public void reverse() {

if (head == null || head.next == null) {

return;

}

Node previous = null;

Node current = head;

while (current != null) {

Node next = current.next;

current.next = previous;

previous = current;

current = next;

}

head = previous;

}

private class Node {

private int data;

private Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

}

SinglyLinkedList list = new SinglyLinkedList();

list.addLast(1);

list.addLast(2);

list.addLast(3);

list.addLast(4);

System.out.println("Original List: " + list.toString());

list.reverse();

System.out.println("Reversed List: " + list.toString());

Output:

Original List: 1 -> 2 -> 3 -> 4

Reversed List: 4 -> 3 -> 2 -> 1