Single Link List

Used Node and Class/Interface in single link list

---------------------------------------------------------------

public interface Lap<K, V> {

public K getHead();

public V getTail();

public void put(K k, V v);

}

===========================

public class SLHashLap implements Lap<SLNode<Integer>, SLNode<Integer>> {

public SLNode<Integer> head = null;

private SLNode<Integer> tail = null;

@Override

public SLNode<Integer> getHead() {

return head;

}

public void setHead(SLNode<Integer> head) {

this.head = head;

}

@Override

public SLNode<Integer> getTail() {

return tail;

}

public void setTail(SLNode<Integer> tail) {

this.tail = tail;

}

@Override

public void put(SLNode<Integer> k, SLNode<Integer> v) {

setHead(k);

setTail(v);

}

}

public class SLNode<K> {

public K k;

public SLNode<K> next;

public SLNode(K k) {

this.k = k;

this.next = null;

}

}

Used interface

----------------------------------------------------

public interface ISLinkList<K> {

/\* insert collection of element \*/

public Lap<SLNode<K>, SLNode<K>> insertArrLast(Lap<SLNode<K>, SLNode<K>> lap, K k[]);

/\* insert single element \*/

public Lap<SLNode<K>, SLNode<K>> insertElementLast(Lap<SLNode<K>, SLNode<K>> lap, K k);

/\* insert collection of element in sorted order \*/

public SLNode<K> insertArrSorted(SLNode<K> head, K k[]);

/\* insert at given position head is at 1st position \*/

public SLNode<K> insertAtPosition(SLNode<K> head, int pos, K k);

/\* print the list \*/

public void printList(SLNode<K> node);

/\*\* 4. Linked List Deletion (Deleting a given key) \*\*/

/\* delete the first occurrence of k \*/

public SLNode<K> deleteFirstK(SLNode<K> head, K k);

/\* Recursive \*/

public int lengthRecursive(SLNode<K> head);

/\* Recursive \*/

**public** SLNode<K> searchRecursive(SLNode<K> head, K k);

/\*\* 8. Swap nodes in a linked list without swapping data \*\*/

public SLNode<K> swapNodes(SLNode<K> head, K x, K y);

/\*\* 12. Write a function to delete a Linked List \*\*/

public void deleteList(SLNode<K> k);

/\*\* 14. Reverse a linked list recursive \*\*/

public SLNode<K> reverseRcv(SLNode<K> k);

/\*\* 15. Detect loop in a linked list \*\*/

/\* create list that captain loop by mistake \*/

public SLNode<K> linkContainLoop(SLNode<K> head, K[] k);

/\* detect and then remove loop in list if any \*/

public SLNode<K> detectAndRemoveLoop(SLNode<K> k);

/\*\* 16. Merge two sorted linked lists \*\*/

public SLNode<K> mergeTwoSotedList(SLNode<K> k1, SLNode<K> k2);

/\*\* 18. Function to check if a singly linked list is palindrome \*\*/

public boolean isPalindrome(SLNode<K> k);

/\*\* 19. Intersection point of two Linked Lists. \*\*/

public SLNode<K> intersectionNode(SLNode<K> k1, SLNode<K> k2);

/\*\* 25. Intersection of two Sorted Linked Lists \*\*/

public SLNode<K> intersectionOfLists(SLNode<K> k1, SLNode<K> k2);

/\*\* 27. Alternating split of a given Singly Linked List \*\*/

public Lap<SLNode<K>, SLNode<K>> alternativeSplit(SLNode<K> k);

/\*\* 29. Merge Sort for Linked Lists \*\*/

public SLNode<K> mergeSort(SLNode<K> k);

/\* get middle node of list \*/

public SLNode<K> getMiddle(SLNode<K> k);

/\* merge two sorted list \*/

public SLNode<K> sortedMerge(SLNode<K> k1, SLNode<K> k2);

/\*\* 33. Segregate even and odd nodes in a Linked List \*\*/

public Lap<SLNode<K>, SLNode<K>> seperateOddEvenList(SLNode<K> k);

/\*\* 34. Detect and Remove Loop in a Linked List \*\*/

public SLNode<K> isLoopInList(SLNode<K> k);

public SLNode<K> dectedAndRemoveLoopInList(SLNode<K> k);

/\*\* 37. Union and Intersection of two Linked Lists \*\*/

public SLNode<K> unionOfLists(SLNode<K> k1, SLNode<K> k2);

public SLNode<K> intersectionOfListsX(SLNode<K> k1, SLNode<K> k2);

/\*\* 38. Find a triplet from linked lists with sum equal to a given number \*\*/

public SLNode<K> sortList(SLNode<K> k);

public List<List<Integer>> tripletInList(SLNode<K> a, SLNode<K> b, SLNode<K> c, int sum);

/\*\* 39. Rotate a Linked List \*\*/

public SLNode<Integer> rotateList(SLNode<K> k, int noOfRotation, boolean isClockWise);

/\*\* 42. Sort a linked list of 0s, 1s and 2s \*\*/

public SLNode<K> sort0s1s2sList(SLNode<K> k);

/\*\* 86. Check whether the length of given linked list is Even or Odd \*\*/

public boolean evenOrOdd(SLNode<K> k);

/\*\* 97. Merge two sorted lists (in-place) \*\*/

public SLNode<K> merge2SortedList(SLNode<K> k1, SLNode<K> k2);

/\*\* 98. Sort a linked list of 0s, 1s and 2s by changing links \*\*/

public SLNode<K> sort0s1s2sListByChangingLink(SLNode<K> k);

}

=====================================================================================Implementation of interface

-------------------------------------------------------------------------------------

public class SLinkListImpl implements ISLinkList<Integer> {

public SLNode<Integer> head = null;

public SLNode<Integer> tail = null;

/\*\* 3. Linked List Insertion \*\*/

@Override

public Lap<SLNode<Integer>, SLNode<Integer>> insertArrLast(Lap<SLNode<Integer>, SLNode<Integer>> lap, Integer a[]) {

if (a == null || a.length == 0)

return lap;

if (lap == null)

lap = new SLHashLap();

head = lap.getHead();

tail = lap.getTail();

int i = 0;

if (head == null) {

head = tail = new SLNode<Integer>(a[0]);

i++;

}

for (; i < a.length; i++) {

tail.next = new SLNode<Integer>(a[i]);

tail = tail.next;

}

lap.put(head, tail);

return lap;

}

/\* insert single element \*/

@Override

public Lap<SLNode<Integer>, SLNode<Integer>> insertElementLast(Lap<SLNode<Integer>, SLNode<Integer>> lap,

Integer x) {

SLNode<Integer> temp = new SLNode<Integer>(x);

if (lap == null) {

lap = new SLHashLap();

head = tail = temp;

} else {

tail = lap.getTail();

tail.next = temp;

tail = tail.next;

}

lap.put(head, tail);

return lap;

}

/\* insert collection of element in sorted order \*/

@Override

public SLNode<Integer> insertArrSorted(SLNode<Integer> head, Integer a[]) {

for (int i = 0; i < a.length; i++) {

SLNode<Integer> temp = new SLNode<Integer>(a[i]);

if (head == null) {

head = tail = temp;

} else if (a[i] <= head.k) { // = applied later

temp.next = head;

head = temp;

} else {

SLNode<Integer> p = head;

while (p.next != null) {

if (a[i] <= p.next.k) { // = applied later

temp.next = p.next;

p.next = temp;

break;

}

p = p.next;

}

if (p.next == null)

p.next = temp;

}

}

return head;

}

/\* insert at given position head is at 1st position \*/

@Override

public SLNode<Integer> insertAtPosition(SLNode<Integer> head, int pos, Integer x) {

if (pos < 1)

return head;

SLNode<Integer> temp = new SLNode<Integer>(x);

if (head != null) {

if (pos == 1) {

temp.next = head;

head = temp;

} else {

SLNode<Integer> p = head;

for (int i = 1; i < pos - 1; i++, p = p.next)

;

if (p != null) {

temp.next = p.next;

p.next = temp;

} else {

System.out.println("\n position :" + pos + " is out of list");

}

}

} else {

if (pos == 1)

head = temp;

}

return head;

}

/\* print the link list nodes \*/

public void printList(SLNode<Integer> node) {

for (; node != null; System.out.print(node.k + "->"), node = node.next)

;

}

/\*\* 4. Linked List Deletion (Deleting a given key) \*\*/

/\* delete the first occurrence of k \*/

@Override

public SLNode<Integer> deleteFirstK(SLNode<Integer> head, Integer x) {

boolean isFound = false;

if (head == null)

return null;

SLNode<Integer> q = head;

if (head.k == x) {

head = head.next;

q.next = null;

q = null;

isFound = true;

} else {

SLNode<Integer> node = head.next;

SLNode<Integer> prev = head;

for (; node != null; prev = node, node = node.next) {

if (node.k == x) {

q = node;

prev.next = q.next;

q.next = null;

q = null;

break;

}

}

}

if (isFound)

System.out.println("\n First" + x + ": is deleted");

else

System.out.println(x + ": is not found");

return head;

}

/\* Recursive \*/

@Override

public int lengthRecursive(SLNode<Integer> head) {

if (head == null)

return 0;

return 1 + lengthRecursive(head.next);

}

/\* Recursive \*/

@Override

public SLNode<Integer> searchRecursive(SLNode<Integer> head, Integer x) {

if (head == null || head.k == x)

return head;

return searchIterative(head.next, x);

}

/\*\* 8. Swap nodes in a linked list without swapping data \*\*/

@Override

public SLNode<Integer> swapNodes(SLNode<Integer> head, Integer x, Integer y) {

// Nothing to do if x and y are same

if (x == y)

return head;

// Search for x (keep track of prevX and CurrX)

SLNode<Integer> prevX = null, currX = head;

while (currX != null && currX.k != x) {

prevX = currX;

currX = currX.next;

}

// Search for y (keep track of prevY and currY)

SLNode<Integer> prevY = null, currY = head;

while (currY != null && currY.k != y) {

prevY = currY;

currY = currY.next;

}

// If either x or y is not present, nothing to do

if (currX == null || currY == null)

return head;

// If x is not head of linked list

if (prevX != null)

prevX.next = currY;

else // make y the new head

head = currY;

// If y is not head of linked list

if (prevY != null)

prevY.next = currX;

else // make x the new head

head = currX;

// Swap next pointers

SLNode<Integer> temp = currX.next;

currX.next = currY.next;

currY.next = temp;

return head;

}

/\*\* 12. Write a function to delete a Linked List \*\*/

@Override

public void deleteList(SLNode<Integer> h) {

SLNode<Integer> p = h;

SLNode<Integer> prev = p;

for (; p != null; p = p.next, prev.next = null, prev = p)

;

h = null;

}

/\*\* 14. Reverse a linked list recursive \*\*/

@Override

public SLNode<Integer> reverseRcv(SLNode<Integer> node) {

if (node == null || node.next == null)

return node;

SLNode<Integer> newNode = reverseRcv(node.next);

node.next.next = node;

node.next = null;

return newNode;

}

/\*\* 15. Detect loop in a linked list \*\*/

/\* create list that contain loop by mistake \*/

@Override

public SLNode<Integer> linkContainLoop(SLNode<Integer> node, Integer[] k) {

SLNode<Integer> head, tail;

head = tail = null;

SLNode<Integer> x = null;

for (int i = 0; i < k.length; i++) {

x = new SLNode<Integer>(k[i]);

if (head == null)

head = tail = x;

else {

tail.next = x;

tail = x;

}

}

/\* do mistake and make point of tail.next to some where \*/

x = head;

for (int i = 0; i < 4 && x != null; x = x.next, i++)

;

tail.next = x;

return head;

}

/\* detect and then remove loop in list if any \*/

@Override

public SLNode<Integer> detectAndRemoveLoop(SLNode<Integer> k) {

boolean isLoop = false;

SLNode<Integer> p = k;

Set<SLNode<Integer>> set = new HashSet<>();

for (; p != null; p = p.next) {

if (set.contains(p.next)) {

isLoop = true;

break;

}

set.add(p);

}

// if loop exits need to remove

if (isLoop)

p.next = null;

return k;

}

/\*\* 16. Merge two sorted linked lists \*\*/

@Override

public SLNode<Integer> mergeTwoSotedList(SLNode<Integer> k1, SLNode<Integer> k2) {

if (k1 == null)

return k2;

if (k2 == null)

return k1;

SLNode<Integer> head, tail;

head = tail = null;

while (k1 != null && k2 != null) {

if (k1.k < k2.k) {

if (head == null) {

head = tail = k1;

} else {

tail.next = k1;

tail = k1;

}

k1 = k1.next;

} else {

if (head == null) {

head = tail = k2;

} else {

tail.next = k2;

tail = k2;

}

k2 = k2.next;

}

}

if (k1 != null) {

while (k1 != null) {

tail.next = k1;

tail = k1;

k1 = k1.next;

}

}

if (k2 != null) {

while (k2 != null) {

tail.next = k2;

tail = k2;

k2 = k2.next;

}

}

return head;

}

/\*\* 18. Function to check if a singly linked list is palindrome \*\*/

@Override

public boolean isPalindrome(SLNode<Integer> h) {

Stack<Integer> s = new Stack<>();

SLNode<Integer> p = h;

while (p != null) {

s.add(p.k);

p = p.next;

}

p = h;

while (p != null) {

if (!p.k.equals(s.pop()))

return false;

p = p.next;

}

return true;

}

/\*\* 19. Intersection point of two Linked Lists. \*\*/

@Override

public SLNode<Integer> intersectionNode(SLNode<Integer> k1, SLNode<Integer> k2) {

SLNode<Integer> p1 = k1;

SLNode<Integer> p2 = k2;

Set<SLNode<Integer>> s1 = new HashSet<>();

while (p1 != null) {

s1.add(p1);

p1 = p1.next;

}

while (p2 != null) {

if (s1.contains(p2))

return p2;

p2 = p2.next;

}

return null;

}

/\*\* 25. Intersection of two Sorted Linked Lists assume element are unique \*\*/

@Override

public SLNode<Integer> intersectionOfLists(SLNode<Integer> k1, SLNode<Integer> k2) {

SLNode<Integer> q = k1;

Set<SLNode<Integer>> set = new HashSet<>();

while (q != null) {

set.add(q);

q = q.next;

}

SLNode<Integer> p = k2;

SLNode<Integer> prev = k2;

while (p != null) {

if (!set.contains(p)) {

if (p.equals(k2)) {

SLNode<Integer> temp = p;

p = p.next;

k2 = k2.next;

temp.next = null;

temp = null;

} else {

k2 = deleteFirstK(k2, p.k);

p = prev;

}

} else {

prev = p;

p = p.next;

}

}

return k2;

}

/\*\* 27. Alternating split of a given Singly Linked List \*\*/

@Override

public Lap<SLNode<Integer>, SLNode<Integer>> alternativeSplit(SLNode<Integer> k) {

Lap<SLNode<Integer>, SLNode<Integer>> lap = new SLHashLap();

if (k == null)

return lap;

SLNode<Integer> q = k;

SLNode<Integer> temp = null;

SLNode<Integer> tempPrev = null;

;

SLNode<Integer> k2 = k.next;

while (q != null && q.next != null) {

tempPrev = temp;

temp = q.next;

q.next = temp.next;

if (tempPrev != null)

tempPrev.next = temp;

q = q.next;

}

lap.put(k, k2);

return lap;

}

/\*\* 29. Merge Sort for Linked Lists \*\*/

@Override

public SLNode<Integer> mergeSort(SLNode<Integer> k) {

if (k == null || k.next == null)

return k;

SLNode<Integer> middle = getMiddle(k);

SLNode<Integer> nextMiddle = middle.next;

middle.next = null;

SLNode<Integer> left = mergeSort(k);

SLNode<Integer> right = mergeSort(nextMiddle);

SLNode<Integer> sortedList = sortedMerge(left, right);

return sortedList;

}

/\* get middle node of list \*/

@Override

public SLNode<Integer> getMiddle(SLNode<Integer> k) {

if (k == null)

return k;

SLNode<Integer> slow = k;

SLNode<Integer> fast = k.next;

while (fast != null) {

fast = fast.next;

if (fast != null) {

fast = fast.next;

slow = slow.next;

}

}

return slow;

}

/\* merge two sorted list \*/

@Override

public SLNode<Integer> sortedMerge(SLNode<Integer> k1, SLNode<Integer> k2) {

SLNode<Integer> result = null;

if (k1 == null)

return k2;

if (k2 == null)

return k1;

if (k1.k <= k2.k) {

result = k1;

result.next = sortedMerge(k1.next, k2);

} else {

result = k2;

result.next = sortedMerge(k1, k2.next);

}

return result;

}

/\*\* 33. Segregate even and odd nodes in a Linked List \*\*/

@Override

public Lap<SLNode<Integer>, SLNode<Integer>> seperateOddEvenList(SLNode<Integer> node) {

SLNode<Integer> ep, eh = null;

SLNode<Integer> op, oh = null;

SLNode<Integer> p = head;

ep = null;

op = null;

while (p != null) {

if (p.k % 2 == 0) {

if (eh == null) {

eh = p;

ep = p;

} else {

ep.next = p;

ep = ep.next;

}

} else {

if (oh == null) {

oh = p;

op = p;

} else {

op.next = p;

op = op.next;

}

}

p = p.next;

}

// managing last pointer every list connected to last node

if (ep.next != null && ep.next.k % 2 != 0)

ep.next = null;

if (op.next != null && op.next.k % 2 == 0)

op.next = null;

SLHashLap lap = new SLHashLap();

lap.setHead(eh);

lap.setTail(oh);

return lap;

}

/\*\* 34. Detect and Remove Loop in a Linked List \*\*/

@Override

public SLNode<Integer> isLoopInList(SLNode<Integer> head) {

Set<Integer> set = new HashSet<>();

SLNode<Integer> hPrev = head;

while (head != null) {

if (!set.contains(head.k))

set.add(head.k);

else

return hPrev;

hPrev = head;

head = head.next;

}

return null;

}

@Override

public SLNode<Integer> dectedAndRemoveLoopInList(SLNode<Integer> head) {

SLNode<Integer> p = head;

p = isLoopInList(p);

if (p != null)

p.next = null;// breaking the loop

return head;

}

/\*\* 37. Union and Intersection of two Linked Lists \*\*/

@Override

public SLNode<Integer> unionOfLists(SLNode<Integer> h1, SLNode<Integer> h2) {

List<Integer> list = new ArrayList<>();

SLNode<Integer> p = h1;

SLNode<Integer> prev = h1;

while (p != null) {

if (searchRecursive(h2, p.k) != null)

list.add(p.k);

prev = p;

p = p.next;

}

prev.next = h2;

for (Integer inte : list)

h1 = deleteLastK(h1, inte);

return h1;

}

@Override

public SLNode<Integer> intersectionOfListsX(SLNode<Integer> h1, SLNode<Integer> h2) {

List<Integer> list = new ArrayList<>();

SLNode<Integer> p = h1;

while (p != null) {

if (searchRecursive(h2, p.k) == null)

list.add(p.k);

p = p.next;

}

for (Integer inte : list)

h1 = deleteFirstK(h1, inte);

return h1;

}

/\*\* 38. Find a triplet from linked lists with sum equal to a given number \*\*/

@SuppressWarnings("unchecked")

@Override

public SLNode<Integer> sortList(SLNode<Integer> head) {

int max = Integer.MIN\_VALUE;

SLNode<Integer> p = head;

// O(n)

for (; p != null; p = p.next) {

if (max < p.k)

max = p.k;

}

SLNode<Integer> arr[] = new SLNode[max + 1];

// O(n)

for (p = head; p != null; p = p.next)

arr[p.k] = p;

// O(max)

for (int i = 0, k = 0; i < arr.length; i++) {

if (arr[i] != null) {

if (k == 0) {

head = arr[i];

p = arr[i];

k++;

} else {

p.next = arr[i];

p = p.next;

}

}

}

// last node must point to null;

p.next = null;

return head;

}

@Override

public List<List<Integer>> tripletInList(SLNode<Integer> ah, SLNode<Integer> bh, SLNode<Integer> ch,

int givenNumber) {

List<List<Integer>> ll = new ArrayList<>();

bh = sortList(bh);

ch = sortList(ch);

SLNode<Integer> a, b, c;

a = ah;

while (a != null) {

// For every node of list a, prick two nodes

// from lists b abd c

b = bh;

c = ch;

while (b != null && c != null) {

// If this a triplet with given sum, print

// it and return true

int sum = a.k + b.k + c.k;

if (sum == givenNumber) {

List<Integer> l = new ArrayList<>();

l.add(a.k);

l.add(b.k);

l.add(c.k);

ll.add(l);

b = b.next;

c = c.next;

}

// If sum of this triplet is smaller, look for

// greater values in b

else if (sum < givenNumber)

b = b.next;

else // If sum is greater, look for smaller values in c

c = c.next;

}

a = a.next; // Move ahead in list a

}

return ll;

}

/\*\* 39. Rotate a Linked List \*\*/

@Override

public SLNode<Integer> rotateList(SLNode<Integer> head, int noOfRotation, boolean isClockWise) {

int n = lengthRecursive(head);

SLNode<Integer> p, q, newHead;

p = head;

q = null;

int rotateLen = isClockWise ? n - noOfRotation : noOfRotation;

for (int i = 1; i < rotateLen; i++)

p = p.next;

q = p.next;

p.next = null;

newHead = q;

while (q.next != null)

q = q.next;

q.next = head;

head = newHead;

return head;

}

/\*\* 40. Flattening a Linked List \*\*/

@Override

public FlatLNode<Integer> insertDownFlatList(FlatLNode<Integer> head, Integer k) {

FlatLNode<Integer> temp = new FlatLNode<Integer>(k);

if (head == null)

head = temp;

else {

FlatLNode<Integer> x = head;

while (x.down != null)

x = x.down;

x.down = temp;

}

return head;

}

/\*\* 42. Sort a linked list of 0s, 1s and 2s \*\*/

@SuppressWarnings("unchecked")

public SLNode<Integer> sort0s1s2sList(SLNode<Integer> head) {

SLNode<Integer> p = head;

// keep the value in 2 arraylist

List<SLNode<Integer>> arList[] = new ArrayList[3];

for (int i = 0; i <= 2; i++)

arList[i] = new ArrayList<>();

// O(n)

for (p = head; p != null; p = p.next)

arList[p.k].add(p);

// O(2\*(n/3))

for (int i = 0, k = 0; i < arList.length; i++) {

for (SLNode<Integer> node : arList[i])

if (node != null) {

if (k == 0) {

head = node;

p = node;

k++;

} else {

p.next = node;

p = p.next;

}

}

}

// last node must point to null;

p.next = null;

return head;

}

/\*\* 86. Check whether the length of given linked list is Even or Odd \*\*/

public boolean evenOrOdd(SLNode<Integer> head) {

return lengthRecursive(head) % 2 == 0;

}

/\*\* 97. Merge two sorted lists (in-place) \*\*/

@Override

public SLNode<Integer> merge2SortedList(SLNode<Integer> h1, SLNode<Integer> h2) {

SLNode<Integer> p, q, temp, head;

temp = head = null;

p = h1;

q = h2;

while (p.next != null && q.next != null) {

if (p.k < q.k) {

temp = p.next;

if (head == null)

head = p;

p.next = q;

p = temp;

} else {

temp = q.next;

if (head == null)

head = q;

q.next = p;

q = temp;

}

}

// link the last node

if (q.next == null) {

q.next = p.next;

p.next = q;

}

if (p.next == null) {

p.next = q;

q.next = p;

}

return head;

}

/\*\* 98. Sort a linked list of 0s, 1s and 2s by changing links \*\*/

@Override

public SLNode<Integer> sort0s1s2sListByChangingLink(SLNode<Integer> head) {

SLNode<Integer> h0, h0next, h1, h1next, h2, h2next, p;

p = head;

h0 = h0next = h1 = h1next = h2 = h2next = null;

for (; p != null; p = p.next) {

switch (p.k) {

case 0:

if (h0 == null) {

h0 = p;

h0next = p;

} else {

h0next.next = p;

h0next = h0next.next;

}

break;

case 1:

if (h1 == null) {

h1 = p;

h1next = p;

} else {

h1next.next = p;

h1next = h1next.next;

}

break;

case 2:

if (h2 == null) {

h2 = p;

h2next = p;

} else {

h2next.next = p;

h2next = h2next.next;

}

break;

}

}

// link the 0s,1s and 2s node

if (h0next != null) {

h0next.next = h1;

if (h1next != null)

h1next.next = h2;

else// if 1 not exits

h0next.next = h2;

} // 0 not exists

else {

if (h1next != null) {

h1next.next = h2;

return h1;

} else {// only 2 exists

return h2;

}

}

return h0;

}

}

/\*\* 101. Count rotations in sorted and rotated linked list \*\*/

@Override

public int countRotationClkWise(SLNode<Integer> head) {

SLNode<Integer> q = head;

int count = 1;

while (q != null && q.next != null) {

if (q.k > q.next.k)

break;

count++;

q = q.next;

}

return count;

}

=====================================================================================

Test case:

-----------------------------------------------------------------

public class ISLinkListTest {

public ISLinkList<Integer> isl = null;

@Before

public void init() {

isl = new SLinkListImpl();

}

/\*\* 3. Linked List Insertion \*\*/

@Test

public void insertArrTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

SLNode<Integer> head = lap.getHead();

// isl.printList(head);

// 1->2->3->4->5->6->7->8->9->

Assert.assertTrue(head.k == 1);

Assert.assertTrue(head.next.next.k == 3);

Assert.assertTrue(head.next.next.next.next.k == 5);

Integer b[] = { 10, 11, 12 };

lap = isl.insertArrLast(lap, b);

// isl.printList(head);

// 1->2->3->4->5->6->7->8->9->10->11->12->

}

/\* insert single element \*/

public void insertElementLastTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

SLNode<Integer> head = lap.getHead();

// isl.printList(head);

Assert.assertTrue(head.k == 1);

Assert.assertTrue(head.next.next.k == 3);

Assert.assertTrue(head.next.next.next.next.k == 5);

Integer b[] = { 10, 11, 12 };

lap = isl.insertArrLast(lap, b);

// isl.printList(head);

}

/\* insert collection of element in sorted order \*/

@Test

public void insertArrSortedTest() {

Integer a[] = { 8, 7, 9, 2, 3, 4, 0, 1, 5, 6 };

SLNode<Integer> head = isl.insertArrSorted(null, a);

// System.out.println("\nprintSorted order list");

isl.printList(head);

Assert.assertTrue(head.k == 0);

Assert.assertTrue(head.next.next.k == 2);

Assert.assertTrue(head.next.next.next.next.k == 4);

}

/\* insert at given position head is at 1st position \*/

@Test

public void insertAtPositionTest() {

Integer a[] = { 3, 5, 6, 7, 8 };

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

SLNode<Integer> head = lap.getHead();

isl.insertAtPosition(head, 1, 2);

isl.insertAtPosition(head, 1, 1);

/\*

\* isl.insertAtPosition(head, 4, 4); isl.insertAtPosition(head, 9, 9);

\* isl.insertAtPosition(head, 0, 10); isl.insertAtPosition(head, 10, 10);

\*/

System.out.println("\n");

isl.printList(head);

}

/\*\* 4. Linked List Deletion (Deleting a given key) \*\*/

@Test

public void deleteFirstKTest() {

Integer k[] = {};

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, k);

Assert.assertTrue(lap == null);

Integer a[] = { 1 };

lap = isl.insertArrLast(null, a);

SLNode<Integer> head = lap.getHead();

head = isl.deleteFirstK(head, 1);

Assert.assertTrue(head == null);

Integer b[] = { 10, 11 };

lap = isl.insertArrLast(null, b);

head = lap.getHead();

head = isl.deleteFirstK(head, 11);

Assert.assertTrue(head.k == 10);

Assert.assertTrue(head.next == null);

// isl.printList(head);

}

/\* Recursive \*/

@Test

public void lengthRecursiveTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> head = null;

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

head = lap.getHead();

Assert.assertTrue(isl.lengthRecursive(head) == 9);

}

/\* Recursive \*/

@Test

public void searchRecursiveTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> head = null;

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

head = lap.getHead();

Assert.assertTrue(isl.searchRecursive(head, 1).k == 1);

Assert.assertTrue(isl.searchRecursive(head, 9).k == 9);

Assert.assertTrue(isl.searchRecursive(head, 5).k == 5);

Assert.assertTrue(isl.searchRecursive(head, 10) == null);

}

/\*\* 8. Swap nodes in a linked list without swapping data \*\*/

@Test

public void swapNodesTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> head = null;

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

head = lap.getHead();

head = isl.swapNodes(head, 1, 9);

head = isl.swapNodes(head, 1, 6);

head = isl.swapNodes(head, 9, 8);

head = isl.swapNodes(head, 3, 7);

// System.out.println("\nafter swap of node");

// isl.printList(head);

}

/\*\* 12. Write a function to delete a Linked List \*\*/

@Test

public void deleteListTest() {

Integer a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> head = null;

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

head = lap.getHead();

isl.deleteList(head);

Assert.assertTrue(head.next == null);

}

/\*\* 14. Reverse a linked list recursive \*\*/

@Test

public void reverseRcvTest() {

Integer a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> head = null;

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

head = lap.getHead();

head = isl.reverseRcv(head);

Assert.assertTrue(head.k == 9);

Assert.assertTrue(head.next.k == 8);

Assert.assertTrue(head.next.next.k == 7);

}

/\*\* 15. Detect loop in a linked list \*\*/

/\* create list that captain loop by mistake \*/

@Test

public void linkContainLoopTes() {

Integer a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> head = isl.linkContainLoop(null, a);

int count = 0;

for (; head != null; head = head.next) {

if (count++ == 100)

break;

}

Assert.assertTrue(count == 101);

}

/\* detect and then remove loop in list if any \*/

@Test

public void detectAndRemoveLoopTest() {

Integer a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> head = isl.linkContainLoop(null, a);

head = isl.detectAndRemoveLoop(head);

int count = 0;

for (; head != null; head = head.next) {

if (count++ == 100)

break;

}

Assert.assertTrue(count == 10);

}

/\*\* 16. Merge two sorted linked lists \*\*/

@Test

public void mergeTwoSotedListTest() {

SLNode<Integer> result = null;

Integer a[] = { 1, 3, 5, 7, 9, 0 };

SLNode<Integer> h1 = isl.insertArrSorted(null, a);

result = isl.mergeTwoSotedList(h1, null);

Assert.assertTrue(isl.lengthRecursive(result) == 6);

result = isl.mergeTwoSotedList(null, h1);

Assert.assertTrue(isl.lengthRecursive(result) == 6);

Integer b[] = { 2, 4, 6, 8, 10 };

SLNode<Integer> h2 = isl.insertArrSorted(null, b);

result = isl.mergeTwoSotedList(h1, h2);

SLNode<Integer> p = result;

for (int i = 0; i <= 10; p = p.next, i++) {

Assert.assertTrue(p.k == i);

}

}

/\*\* 18. Function to check if a singly linked list is palindrome \*\*/

@Test

public void isPalindromeTest() {

Integer a[] = { 1, 2, 3, 4, 5, 3, 2, 1 };

Integer b[] = { 1, 2, 3, 4, 3, 2, 1 };

Integer c[] = { 1, 2, 3, 4, 4, 3, 2, 1 };

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

Assert.assertTrue(!isl.isPalindrome(lap.getHead()));

lap = isl.insertArrLast(null, b);

Assert.assertTrue(isl.isPalindrome(lap.getHead()));

lap = isl.insertArrLast(null, c);

Assert.assertTrue(isl.isPalindrome(lap.getHead()));

}

/\*\* 19. Intersection point of two Linked Lists. \*\*/

@Test

public void intersectionNodeTest() {

Integer a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Integer b[] = { 0, 1, 2, 3 };

Lap<SLNode<Integer>, SLNode<Integer>> lap1 = isl.insertArrLast(null, a);

Lap<SLNode<Integer>, SLNode<Integer>> lap2 = isl.insertArrLast(null, b);

SLNode<Integer> tail2 = lap2.getTail();

SLNode<Integer> p = lap1.getHead();

int i = 0;

while (i != 6) {

p = p.next;

i++;

}

tail2.next = p;

i = 0;

SLNode<Integer> q = lap2.getHead();

while (i != 5) {

q = q.next;

i++;

}

// here testing is is connecting or not at 6 of first list

Assert.assertTrue(q.k == 7);

Assert.assertTrue(isl.intersectionNode(lap1.getHead(), lap2.getHead()).k == 6);

}

/\*\* 25. Intersection of two Sorted Linked Lists \*\*/

@Test

public void intersectionOfListsTest() {

Integer a[] = { 1, 2, 3, 4, 5 };

Integer b[] = { -1, 0, 1, 2, 3, 4, 6 };

Lap<SLNode<Integer>, SLNode<Integer>> k1 = isl.insertArrLast(null, a);

Lap<SLNode<Integer>, SLNode<Integer>> k2 = isl.insertArrLast(null, b);

SLNode<Integer> k = isl.intersectionOfLists(k1.getHead(), k2.getHead());

// isl.printList(k);

for (int i = 1; i < 5; i++, k = k.next)

Assert.assertTrue(k.k == i);

}

/\*\* 27. Alternating split of a given Singly Linked List \*\*/

@Test

public void alternativeSplit() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

lap = isl.alternativeSplit(lap.getHead());

SLNode<Integer> k1 = lap.getHead();

SLNode<Integer> k2 = lap.getTail();

for (int i = 1; i < 10; i += 2, k1 = k1.next)

Assert.assertTrue(i == k1.k);

for (int i = 2; i <= 10; i += 2, k2 = k2.next)

Assert.assertTrue(i == k2.k);

}

/\*\*

\* 29. Merge Sort for Linked Lists...getMiddle and sortedMerge internally called

\* so tested here only.

\*\*/

@Test

public void mergeSortTest() {

Integer a1[] = { 1, 3, 2, 5, 4, 7, 6, 9, 8, 10, 0 };

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a1);

SLNode<Integer> k = isl.mergeSort(lap.getHead());

// isl.printList(k);

for (int i = 0; i < 11; i++, k = k.next)

Assert.assertTrue(k.k == i);

}

/\*\* 33. Segregate even and odd nodes in a Linked List \*\*/

@Test

public void seperateOddEvenListTest() {

Integer a[] = { 1, 3, 5, 2, 4, 6, 7, 8, 9, 10, 11 };

Lap<SLNode<Integer>, SLNode<Integer>> lap = isl.insertArrLast(null, a);

lap = isl.seperateOddEvenList(lap.getHead());

SLNode<Integer> evn = lap.getHead();

SLNode<Integer> odd = lap.getTail();

Assert.assertTrue(evn.k == 2);

Assert.assertTrue(evn.next.k == 4);

Assert.assertTrue(odd.k == 1);

Assert.assertTrue(odd.next.k == 3);

}

/\*\* 34. Detect and Remove Loop in a Linked List \*\*/

@Test

public void isLoopInListTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> q = null;

Lap<SLNode<Integer>, SLNode<Integer>> lap1 = isl.insertArrLast(null, a);

SLNode<Integer> tail = lap1.getTail();

SLNode<Integer> head = lap1.getHead();

q = head;

for (int i = 0; i < 3; i++)

q = q.next;

tail.next = q;

Assert.assertTrue(isl.isLoopInList(head).k == 9);

}

@Test

public void dectedAndRemoveLoopInListTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> q = null;

Lap<SLNode<Integer>, SLNode<Integer>> lap1 = isl.insertArrLast(null, a);

SLNode<Integer> tail = lap1.getTail();

SLNode<Integer> head = lap1.getHead();

q = head;

for (int i = 0; i < 3; i++)

q = q.next;

tail.next = q;

head = isl.detectAndRemoveLoop(head);

Assert.assertTrue(tail.next == null);

}

/\*\* 37. Union and Intersection of two Linked Lists \*\*/

@Test

public void unionOfListsTest() {

Integer a[] = { 8, 9, 1, 2, 6, 7, 3, 5 };

Integer b[] = { 8, 9, 7, 2, 4 };

Lap<SLNode<Integer>, SLNode<Integer>> lap1 = isl.insertArrLast(null, a);

Lap<SLNode<Integer>, SLNode<Integer>> lap2 = isl.insertArrLast(null, b);

SLNode<Integer> h1 = lap1.getHead();

SLNode<Integer> h2 = lap2.getHead();

h1 = isl.unionOfLists(h1, h2);

isl.printList(h1);

}

@Test

public void intersectionOfListsXTest() {

Integer a[] = { 8, 9, 1, 2, 6, 7, 3, 5 };

Integer b[] = { 8, 9, 7, 2, 4 };

Lap<SLNode<Integer>, SLNode<Integer>> lap1 = isl.insertArrLast(null, a);

Lap<SLNode<Integer>, SLNode<Integer>> lap2 = isl.insertArrLast(null, b);

SLNode<Integer> h1 = lap1.getHead();

SLNode<Integer> h2 = lap2.getHead();

h1 = isl.intersectionOfLists(h1, h2);

isl.printList(h1);

}

/\*\* 38. Find a triplet from linked lists with sum equal to a given number \*\*/

@Test

public void sortListTest() {

Integer a[] = { 15, 8, 9, 1, 2, 6, 7, 3, 5, 4 };

Lap<SLNode<Integer>, SLNode<Integer>> lap1 = isl.insertArrLast(null, a);

SLNode<Integer> head = lap1.getHead();

head = isl.sortList(head);

isl.printList(head);

}

@Test

public void tripletInListTest() {

Integer a[] = { 20, 4, 15, 10 };

Integer b[] = { 10, 9, 4, 2 };

Integer c[] = { 1, 2, 8, 4 };

Lap<SLNode<Integer>, SLNode<Integer>> ha = isl.insertArrLast(null, a);

Lap<SLNode<Integer>, SLNode<Integer>> hb = isl.insertArrLast(null, b);

Lap<SLNode<Integer>, SLNode<Integer>> hc = isl.insertArrLast(null, c);

List<List<Integer>> ll = isl.tripletInList(ha.getHead(), hb.getHead(), hc.getHead(), 25);

Assert.assertTrue(ll.size() == 2);

}

/\*\* 39. Rotate a Linked List \*\*/

@Test

public void rotateListTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Lap<SLNode<Integer>, SLNode<Integer>> ha = isl.insertArrLast(null, a);

SLNode<Integer> head = ha.getHead();

head = isl.rotateList(head, 4, true);

head = isl.rotateList(head, 4, false);

int i = 0;

for (; head != null; head = head.next)

Assert.assertTrue(head.k == a[i++]);

}

/\*\* 42. Sort a linked list of 0s, 1s and 2s \*\*/

@Test

public void sort0s1s2sListTest() {

Integer a[] = { 2, 1, 0, 0, 1, 2, 2, 1, 1, 0, 0, 0, 1, 2 };

Lap<SLNode<Integer>, SLNode<Integer>> ha = isl.insertArrLast(null, a);

SLNode<Integer> head = ha.getHead();

head = isl.sort0s1s2sList(head);

isl.printList(head);

}

/\*\* 97. Merge two sorted lists (in-place) \*\*/

@Test

public void merge2SortedListTest() {

Integer a[] = { 1, 3, 5, 7, 9 };

Integer b[] = { 2, 4, 6, 8 };

SLNode<Integer> h1 = isl.insertArrLast(null, a).getHead();

SLNode<Integer> h2 = isl.insertArrLast(null, b).getHead();

h1 = isl.merge2SortedList(h1, h2);

isl.printList(h1);

}

/\*\* 98. Sort a linked list of 0s, 1s and 2s by changing links \*\*/

@Test

public void sort0s1s2sListByChangingLinkTest() {

Integer a[] = { 1, 0, 2, 1, 1, 0, 0, 2, 2, 0, 0, 1, 0, 2 };

Integer b[] = { 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 2, 2, 2, 2 };

SLNode<Integer> head = isl.insertArrLast(null, a).getHead();

head = isl.sort0s1s2sListByChangingLink(head);

// isl.printList(head);

for (int i = 0; head != null; head = head.next)

Assert.assertTrue(head.k == b[i++]);

}

/\*\* 99. Insert a node after the n-th node from the end \*\*/

@Test

public void insertAfterNthNodeFromEndTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 10, 11 };

SLNode<Integer> head = isl.insertArrLast(null, a).getHead();

head = isl.insertAfterNthNodeFromEnd(head, 9, 3);

for (int i = 1; i <= a.length - 3 + 1; i++) {

head = head.next;

}

// isl.printList(head);

Assert.assertTrue(head.k == 9);

}

/\*\* 100. Rotate Linked List block wise \*\*/

@Test

public void blockWiseRotationTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

Integer b[] = { 2, 3, 1, 5, 6, 4, 8, 9, 7, 10, 11 };

SLNode<Integer> head = isl.insertArrLast(null, a).getHead();

head = isl.blockWiseRotation(head, 3, 2);

// isl.printList(head);

int i = 0;

for (; head != null; head = head.next)

Assert.assertTrue(head.k == b[i++]);

}

/\*\* 101. Count rotations in sorted and rotated linked list \*\*/

@Test

public void countRotationClkWiseTest() {

Integer a[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

SLNode<Integer> head = isl.insertArrLast(null, a).getHead();

head = isl.rotateList(head, 2, true);

Assert.assertTrue(isl.countRotationClkWise(head) == 2);

}