Single Circular Link List

Used Node and Class/Interface in single link list

**public** **class** BTNode<T> {

**public** T t;

**public** BTNode<T> lt, rt, pt, next, prev; // next point to successor/prdecessor of node

**public** **int** h, ind;

**public** BTNode(T t) {

**this**.t = t;

**this**.lt = **this**.rt = **this**.pt = **this**.next = **this**.prev = **null**;

**this**.h = 0;

**this**.ind = 0;

}

}

**public** **class** DLNode<K> {

**public** K k;

**public** DLNode<K> next, prev;

**public** DLNode(K k) {

**this**.k = k;

**this**.next = **this**.prev = **null**;

}

}

public interface Lap<K, V> {

public K getHead();

public V getTail();

public void put(K k, V v);

}

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

**public** **class** DLHashLap **implements** Lap<DLNode<Integer>, DLNode<Integer>> {

**public** DLNode<Integer> head = **null**;

**public** DLNode<Integer> tail = **null**;

@Override

**public** DLNode<Integer> getHead() {

**return** head;

}

**public** **void** setHead(DLNode<Integer> head) {

**this**.head = head;

}

@Override

**public** DLNode<Integer> getTail() {

**return** tail;

}

**public** **void** setTail(DLNode<Integer> tail) {

**this**.tail = tail;

}

@Override

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

setHead(k);

setTail(v);

}

}

**public** **class** BTreeUitls<K **extends** Comparable<K>> {

**public** BTNode<K> createBst(BTNode<K> node, K k) {

**if** (node == **null**)

node = **new** BTNode<K>(k);

**else** {

// replacing

**if** (k.equals(node.t)) {

node.t = k;

} **else** **if** (k.compareTo(node.t) < 0) {

node.lt = createBst(node.lt, k);

} **else** {

node.rt = createBst(node.rt, k);

}

}

**return** node;

}

**public** HTNode<K> createBst(HTNode<K> node, K k) {

**if** (node == **null**)

node = **new** HTNode<K>(k);

**else** {

// replacing

**if** (k.equals(node.k)) {

node.k = k;

} **else** **if** (k.compareTo(node.k) < 0) {

node.lt = createBst(node.lt, k);

node.lt.pt = node;

} **else** {

node.rt = createBst(node.rt, k);

node.rt.pt = node;

}

}

**return** node;

}

**public** **void** inOrderPrint(HTNode<K> node) {

**if** (node != **null**) {

inOrderPrint(node.lt);

System.***out***.print("(" + node.k + ")->");

inOrderPrint(node.rt);

}

}

**public** HTNode<K> recvSearch(HTNode<K> node, K k) {

**if** (node != **null**) {

**if** (node.k.equals(k)) {

**return** node;

} **else** **if** (node.k.compareTo(k) < 0) {

**return** recvSearch(node.rt, k);

} **else** {

**return** recvSearch(node.lt, k);

}

}

**return** **null**;

}

**public** HTNode<K> delete(HTNode<K> node, K key) {

**if** (node == **null**) {

System.***out***.println(key + " not found in tree");

**return** **null**;

} **else** **if** (key.compareTo(node.k) < 0) {

delete(node.lt, key);

} **else** **if** (key.compareTo(node.k) > 0) {

delete(node.rt, key);

} **else** **if** (node.k.equals(key)) // found the node

{

// case 1: if no child of deleting node

**if** (node.lt == **null** && node.rt == **null**) {

HTNode<K> q = node;

// if deleting node is left node of its parent

**if** (q.pt.lt == node) {

q.pt.lt = **null**;

} **else** // deleting node is right child of its parent

{

q.pt.rt = **null**;

}

q = **null**;

} **else** // deleting node have left as well as right child ..eighter

// manage by right succesor or left predecesor

{

HTNode<K> p = node.lt;

**if** (p != **null**) // find largest in left subtree

{

**if** (p.lt == **null** && p.rt == **null**) {

node.k = node.lt.k;

node.lt = **null**;

} **else** **if** (p.rt != **null**) {

**while** (p.rt != **null**) {

p = p.rt;

}

HTNode<K> q = p;

p.pt.rt = q.lt; // add left predecesor to right to its

// parent

node.k = q.k; // copy the right precedesor to deleing

// node and nullify it

q = **null**;

} **else** {

node.k = node.lt.k;

node.lt = node.lt.lt;

node.lt.pt = node;

}

} **else** // find smallest in right subtree

{

HTNode<K> x = node.rt;

**if** (x.lt == **null** && x.rt == **null**) {

node.k = node.rt.k;

node.rt = **null**;

} **else** **if** (x.lt != **null**) {

**while** (x.lt != **null**) {

x = x.lt;

}

HTNode<K> y = x;

x.pt.lt = y.rt; // add left predecesor to right to its

// parent

node.k = y.k; // copy the right precedesor to deleing

// node and nullify it

x = **null**;

} **else** {

node.k = node.rt.k;

node.rt = node.rt.rt;

node.rt.pt = node;

}

}

}

}

**return** node;

}

**public** BTNode<K> bstToListUtils(BTNode<K> k) {

**if** (k == **null**)

**return** k;

// convert left subtree to list and link to root/k

**if** (k.lt != **null**) {

BTNode<K> left = bstToListUtils(k.lt);

// go to inorder predecessor

**for** (; left.rt != **null**; left = left.rt)

;

left.rt = k;

k.lt = left;

}

// convert right subtree to list and link to root/k

**if** (k.rt != **null**) {

BTNode<K> right = bstToListUtils(k.rt);

// go to inorder predecessor

**for** (; right.lt != **null**; right = right.lt)

;

right.lt = k;

k.rt = right;

}

**return** k;

}

}

Used interface

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

**public** **interface** IDLinkList<K> {

/\*\* 1. Doubly Linked List Introduction and Insertion \*\*/

/\* insert collection of element \*/

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

/\* insert element at first \*/

**public** Lap<DLNode<K>, DLNode<K>> insertElementFirst(Lap<DLNode<K>, DLNode<K>> lap, K k);

/\* insert single element at end of list \*/

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

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

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

/\* print list \*/

**public** **void** print(DLNode<K> k);

/\*\* 2. Delete a node in a Doubly Linked List \*\*/

/\* delete first occurrence \*/

**public** DLNode<K> deleteNodeFirstK(DLNode<K> h, K k);

/\* delete all the node having value k \*/

**public** DLNode<K> deleteAllOccurance(DLNode<K> h, K k);

/\* delete first occurrence \*/

**public** DLNode<K> deleteNodeLastK(DLNode<K> h, K k);

/\*\* 3. Reverse a Doubly Linked List \*\*/

/\* iterative way to reverse \*/

**public** DLNode<K> reverseIterative(DLNode<K> h);

/\* recursive way to reverse \*/

**public** DLNode<K> reverseRecursive(DLNode<K> node);

/\*\* 4. The Great Tree-List Recursion Problem. \*\*/

**public** BTNode<K> bstToList(BTNode<K> k);

/\*\* 5. Copy a linked list with next and arbit pointer \*\*/

/\* create random pointer list \*/

**public** DLNode<K> createRandomPointerList(K[] k);

/\* clone random pointer list \*/

**public** DLNode<K> cloneRandomPointerList(DLNode<K> h);

/\*\* 6. QuickSort on Doubly Linked List \*\*/

**public** DLNode<K> partitation(DLNode<K> h, DLNode<K> l);

**public** **void** quickSort(DLNode<K> h, DLNode<K> t);

/\*\* 7. Swap Kth node from beginning with Kth node from end in a Linked List \*\*/

**public** DLNode<K> swapKthNodeFromBegAndEnd(DLNode<K> node, **int** k);

/\*\* 8. Merge Sort for Doubly Linked List \*\*/

/\* get middle node of list \*/

**public** DLNode<K> getMiddle(DLNode<K> k);

/\* merge two sorted list \*/

**public** DLNode<Integer> sortedMerge(DLNode<K> k1, DLNode<K> k2);

/\* merge sort of list \*/

**public** DLNode<K> mergeSort(DLNode<K> k);

/\*\* 9. Create a Doubly Linked List from a Ternary Tree \*\*/

**public** TTreeNode<K> createTernaryTree(TTreeNode<K> node, **int** key);

**public** **void** printTTree(TTreeNode<K> node);

**public** TTreeNode<K> createFromTernaryTree(TTreeNode<K> node);

**public** **void** printTTOfDl(TTreeNode<K> node);

/\*\* 10. Find pairs with given sum in doubly linked list \*\*/

**public** List<List<DLNode<K>>> pairOfGivenSum(DLNode<K> k, **int** sum);

/\*\* 11. Insert value in sorted way in a sorted doubly linked list \*\*/

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

/\*\* 12. Delete a Doubly Linked List node at a given position \*\*/

**public** DLNode<Integer> deletePos(DLNode<Integer> h, **int** pos);

/\*\*

\* 13. Count triplets in a sorted doubly linked list whose sum is equal to a

\* given value x

\*\*/

**public** **int** countPairs(DLNode<Integer> first, DLNode<Integer> second, **int** value);

**public** **int** tripletROfGivenSum(DLNode<K> k, **int** sum);

/\*\* 14. Remove duplicates from a sorted doubly linked list \*\*/

**public** DLNode<K> removeDuplicate(DLNode<K> k);

/\*\* 15. Delete all occurrences of a given key in a doubly linked list \*\*/

**public** DLNode<K> deleteAllOccuranceOfX(DLNode<K> k, **int** x);

/\*\* 16. Remove duplicates from an unsorted doubly linked list \*\*/

**public** DLNode<K> remveDuplicateFromNonSotedList(DLNode<K> k);

/\*\* 17. Sort the biotonic doubly linked list \*\*/

**public** DLNode<K> sortBitonicList(DLNode<K> k);

/\*\* 18. Sort a k sorted doubly linked list \*\*/

**public** DLNode<K> sortAKSortedDLL(DLNode<K> k, **int** x);

/\*\* 19. Convert a given Binary Tree to Doubly Linked List | Set \*\*/

// @See 4.

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

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

**public** **class** DLinkListImpl **implements** IDLinkList<Integer> {

**public** BTreeUitls<Integer> util = **new** BTreeUitls<>();

**public** DLNode<Integer> head = **null**;

**public** DLNode<Integer> tail = **null**;

/\*\* 1. Doubly Linked List Introduction and Insertion \*\*/

/\* insert collection of element \*/

@Override

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

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

**return** lap;

**if** (lap == **null**)

lap = **new** DLHashLap();

head = lap.getHead();

tail = lap.getTail();

**int** i = 0;

**if** (head == **null**) {

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

i++;

}

DLNode<Integer> temp = **null**;

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

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

tail.next = temp;

temp.prev = tail;

tail = tail.next;// tail=temp;

}

lap.put(head, tail);

**return** lap;

}

/\* insert element at first \*/

**public** Lap<DLNode<Integer>, DLNode<Integer>> insertElementFirst(Lap<DLNode<Integer>, DLNode<Integer>> lap,

Integer k) {

DLNode<Integer> temp = **new** DLNode<Integer>(k);

**if** (lap == **null**)

lap = **new** DLHashLap();

head = lap.getHead();

tail = lap.getTail();

**if** (head == **null**) {

head = tail = temp;

} **else** {

head.prev = temp;

temp.next = head;

head = head.prev;// head=temp;

}

lap.put(head, tail);

**return** lap;

}

/\* insert single element at end of list \*/

@Override

**public** Lap<DLNode<Integer>, DLNode<Integer>> insertElementLast(Lap<DLNode<Integer>, DLNode<Integer>> lap,

Integer k) {

DLNode<Integer> temp = **new** DLNode<Integer>(k);

**if** (lap == **null**)

lap = **new** DLHashLap();

head = lap.getHead();

tail = lap.getTail();

**if** (head == **null**) {

head = tail = temp;

} **else** {

tail.next = temp;

temp.prev = tail;

tail = tail.next;// tail=temp;

}

lap.put(head, tail);

**return** lap;

}

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

@Override

**public** DLNode<Integer> insertAtPosition(DLNode<Integer> head, **int** pos, Integer k) {

**if** (pos < 1)

**return** head;

DLNode<Integer> temp = **new** DLNode<Integer>(k);

**if** (head != **null**) {

**if** (pos == 1) {

temp.next = head;

head.prev = temp;

head = temp;

} **else** {

DLNode<Integer> p = head;

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

;

**if** (p != **null**) {

temp.next = p.next;

p.next = temp;

temp.prev = p;

temp.next.prev = temp;

} **else** {

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

}

}

} **else** { // ie no node exists and pos=1

**if** (pos == 1)

head = temp;

}

**return** head;

}

/\* print list \*/

@Override

**public** **void** print(DLNode<Integer> head) {

System.***out***.println("forward:print");

DLNode<Integer> prv = head;

**for** (; head != **null**; prv = head, head = head.next)

System.***out***.print(head.k + "<=>");

System.***out***.println("\nBackward:print");

**for** (; prv != **null**; prv = prv.prev)

System.***out***.print(prv.k + "<=>");

}

/\*\* 2. Delete a node in a Doubly Linked List \*\*/

@Override

**public** DLNode<Integer> deleteNodeFirstK(DLNode<Integer> head, Integer k) {

**boolean** isFound = **false**;

**if** (head == **null**)

**return** **null**;

DLNode<Integer> q = head;

**if** (head.k == k) {

head = head.next;

**if** (head != **null**)// if only one node exits that is going to delete

head.prev = **null**;

q.next = **null**;

q = **null**;

isFound = **true**;

} **else** {

DLNode<Integer> node = head.next;

DLNode<Integer> prev = head;

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

**if** (node.k == k) {

q = node;

prev.next = q.next;

**if** (q.next != **null**)// if q is last node

q.next.prev = q.prev;

q.next = q.prev = **null**;

q = **null**;

**break**;

}

}

}

**if** (isFound)

System.***out***.println("\n" + k + ": is deleted");

**else**

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

**return** head;

}

/\* delete all the node having value k \*/

@Override

**public** DLNode<Integer> deleteAllOccurance(DLNode<Integer> head, Integer k) {

**boolean** isFound = **false**;

**if** (head == **null**)

**return** **null**;

DLNode<Integer> q = **null**;

**while** (head != **null** && head.k == k) {

q = head;

head = head.next;

head.prev = **null**;

q.next = **null**;

q = head;

isFound = **true**;

}

DLNode<Integer> node = head.next;

DLNode<Integer> prev = head;

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

**if** (node.k == k) {

q = node;

prev.next = q.next;

**if** (q.next != **null**) // is q is the last node

q.next.prev = q.prev;

q.next = **null**;

q = **null**;

node = prev;

isFound = **true**;

}

}

**if** (isFound)

System.***out***.println("\nall: " + k + ": is deleted");

**else**

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

**return** head;

}

/\* delete first occurrence \*/

@Override

**public** DLNode<Integer> deleteNodeLastK(DLNode<Integer> head, Integer k) {

**boolean** isFound = **false**;

DLNode<Integer> prev, prevK, q;

q = prev = head;

prevK = **null**;

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

**if** (q.k == k)

prevK = prev;

}

**if** (prevK == head) {

head = head.next;

head.prev = **null**;

prevK.next = **null**;

prevK = **null**;

isFound = **true**;

} **else** **if** (prevK != **null**) {

DLNode<Integer> temp = prevK.next;

prevK.next = temp.next;

**if** (temp.next != **null**)// if temp is the last node

temp.next.prev = temp.prev;

temp.prev = **null**;

temp.next = **null**;

temp = **null**;

isFound = **true**;

}

**if** (isFound)

System.***out***.println("\nlast: " + k + ": is deleted");

**else**

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

**return** head;

}

/\*\* 3. Reverse a Doubly Linked List \*\*/

/\* iterative way to reverse \*/

@Override

**public** DLNode<Integer> reverseIterative(DLNode<Integer> head) {

DLNode<Integer> p = head;

DLNode<Integer> t = **null**;

**while** (p != **null**) {

t = p.prev;

p.prev = p.next;

p.next = t;

p = p.prev;

}

**return** t.prev;

}

/\* recursive way to reverse \*/

@Override

**public** DLNode<Integer> reverseRecursive(DLNode<Integer> node) {

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

**return** node;

DLNode<Integer> newNode = reverseRecursive(node.next);

newNode.prev = **null**;

node.next.next = node;

node.prev = node.next;

node.next = **null**;

**return** newNode;

}

/\*\* 4. The Great Tree-List Recursion Problem. \*\*/

**public** BTNode<Integer> bstToList(BTNode<Integer> k) {

BTNode<Integer> node = util.bstToListUtils(k);

**while** (node.lt != **null**)

node = node.lt;

**return** node;

}

/\*\* 5. Copy a linked list with next and arbit pointer \*\*/

/\* create random pointer list \*/

@Override

**public** DLNode<Integer> createRandomPointerList(Integer[] k) {

Lap<DLNode<Integer>, DLNode<Integer>> lap = insertArrLast(**null**, k);

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

DLNode<Integer> tail = lap.getTail();

head.prev = head.next.next;

head.next.prev = head;

head.next.next.prev = tail;

tail.next = head.next;

head.next.next.next.prev = head.next.next;

**return** head;

}

/\* clone random pointer list \*/

@Override // **TODO**

**public** DLNode<Integer> cloneRandomPointerList(DLNode<Integer> h) {

/\*

\* DLNode<Integer> cloneHead = null; DLNode<Integer> cloneTail = null;

\* DLNode<Integer> p = head; DLNode<Integer> temp = null;

\* ArrayList<DLNode<Integer>> old = new ArrayList(); ArrayList<DLNode<Integer>>

\* newly = new ArrayList();

\*

\* Set<DLNode<Integer>> set = new HashSet<>(); for (; !set.contains(p); p =

\* p.next) { set.add(p); temp = new DLNode<Integer>(p.k); if (cloneHead == null)

\* { cloneHead = cloneTail = temp; cloneHead.prev = head; newly.add(cloneHead);

\* } else { cloneTail.next = temp; cloneTail = temp; cloneTail.prev = p;

\* newly.add(cloneTail); } old.add(p); } // select the corresponding node of x

\* in newly list //and point to for (int i = 0; i < old.size(); i++) {

\* DLNode<Integer> x = old.get(i); DLNode<Integer> y = newly.get(i); y.prev.prev

\* = newly.get(x.prev.k); } return cloneHead;

\*/

**return** **null**;

}

/\*\* 6. QuickSort on Doubly Linked List \*\*/

@Override

**public** DLNode<Integer> partitation(DLNode<Integer> h, DLNode<Integer> t) {

// set pivot as h element

Integer x = t.k;

// similar to i = l-1 for array implementation

DLNode<Integer> i = h.prev;

// Similar to "for (int j = l; j <= h- 1; j++)"

**for** (DLNode<Integer> j = h; j != t; j = j.next) {

**if** (j.k <= x) {

// Similar to i++ for array

i = (i == **null**) ? h : i.next;

**int** temp = i.k;

i.k = j.k;

j.k = temp;

}

}

i = (i == **null**) ? h : i.next; // Similar to i++

**int** temp = i.k;

i.k = h.k;

h.k = temp;

**return** i;

}

@Override

**public** **void** quickSort(DLNode<Integer> h, DLNode<Integer> t) {

**if** (h != **null** && h != t && h != t.next) {

DLNode<Integer> tmp = partitation(h, t);

quickSort(h, tmp.prev);

quickSort(tmp.next, t);

}

}

/\*\* 7. Swap Kth node from beginning with Kth node from end in a Linked List \*\*/

**public** DLNode<Integer> swapKthNodeFromBegAndEnd(DLNode<Integer> head, **int** k) {

**int** n = 1;

DLNode<Integer> x, y;

x = y = head;

**while** (y.next != **null**) {

n++;

y = y.next;

}

// Check if k is valid

**if** (n < k)

**return** head;

// If x (kth node from start) and y(kth node from end)

// are same

**if** (2 \* k - 1 == n)

**return** head;

**int** c = 1;

**while** (c != k) {

x = x.next;

y = y.prev;

c++;

}

DLNode<Integer> xprev, xnext, yprev, ynext;

xprev = xnext = yprev = ynext = **null**;

xprev = x.prev;

xnext = x.next;

yprev = y.prev;

ynext = y.next;

**if** (xprev != **null** && ynext != **null**) {

xprev.next = yprev.next;

y.prev = xprev;

y.next = xnext;

xnext.prev = y;

yprev.next = x;

x.prev = yprev;

x.next = ynext;

ynext.prev = x;

} **else**// swaping data

{

**int** temp = x.k;

x.k = y.k;

y.k = temp;

}

**return** head;

}

/\*\* 8. Merge Sort for Doubly Linked List \*\*/

/\* get middle node of list \*/

@Override

**public** DLNode<Integer> getMiddle(DLNode<Integer> k) {

**if** (k == **null**)

**return** k;

DLNode<Integer> slow = k;

DLNode<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** DLNode<Integer> sortedMerge(DLNode<Integer> k1, DLNode<Integer> k2) {

DLNode<Integer> result = **null**;

DLNode<Integer> temp = **null**;

**if** (k1 == **null**)

**return** k2;

**if** (k2 == **null**)

**return** k1;

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

result = k1;

temp = sortedMerge(k1.next, k2);

result.next = temp;

temp.prev = result;

} **else** {

result = k2;

temp = sortedMerge(k1, k2.next);

result.next = temp;

temp.prev = result;

}

**return** result;

}

/\* merge sort of list \*/

@Override

**public** DLNode<Integer> mergeSort(DLNode<Integer> k) {

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

**return** k;

DLNode<Integer> middle = getMiddle(k);

DLNode<Integer> nextMiddle = middle.next;

middle.next = **null**;

nextMiddle.prev = **null**;

DLNode<Integer> left = mergeSort(k);

DLNode<Integer> right = mergeSort(nextMiddle);

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

**return** sortedList;

}

/\*\* 9. Create a Doubly Linked List from a Ternary Tree \*\*/

@Override

**public** **void** printTTree(TTreeNode<Integer> node) {

Queue<TTreeNode<Integer>> queue = **new** LinkedList<>();

queue.add(node);

**while** (!queue.isEmpty()) {

TTreeNode<Integer> x = queue.poll();

System.***out***.print(x.dt + "->");

**if** (x.lt != **null**)

queue.add(x.lt);

**if** (x.md != **null**)

queue.add(x.md);

**if** (x.rt != **null**)

queue.add(x.rt);

}

}

@Override

**public** TTreeNode<Integer> createTernaryTree(TTreeNode<Integer> node, **int** key) {

**if** (node == **null**)

node = **new** TTreeNode<Integer>(key);

**else** **if** (node.lt == **null** || node.md == **null** || node.rt == **null**) {

**if** (node.lt == **null**)

node.lt = createTernaryTree(node.lt, key);

**else** **if** (node.md == **null**)

node.md = createTernaryTree(node.md, key);

**else** **if** (node.rt == **null**)

node.rt = createTernaryTree(node.rt, key);

} **else** {

**if** (node.lt != **null**)

createTernaryTree(node.lt, key);

**else** **if** (node.md != **null**)

createTernaryTree(node.md, key);

**else** **if** (node.rt != **null**)

createTernaryTree(node.rt, key);

}

**return** node;

}

**public** TTreeNode<Integer> createFromTernaryTree(TTreeNode<Integer> root) {

TTreeNode<Integer> node = root;

TTreeNode<Integer> tail = **null**;

Queue<TTreeNode<Integer>> queue = **new** LinkedList<>();

queue.add(node);

**while** (!queue.isEmpty()) {

TTreeNode<Integer> x = queue.poll();

**if** (x.lt != **null**)

queue.add(x.lt);

**if** (x.md != **null**)

queue.add(x.md);

**if** (x.rt != **null**)

queue.add(x.rt);

**if** (tail == **null**) {

tail = x;

x.lt = x.rt = x.md = **null**;

} **else** {

tail.rt = x;

x.lt = tail;

x.md = x.rt = **null**;

tail = x;

}

}

**return** root;

}

**public** **void** printTTOfDl(TTreeNode<Integer> node) {

**while** (node != **null**) {

System.***out***.print(node.dt + "->");

node = node.rt;

}

}

/\*\* 10. Find pairs with given sum in doubly linked list \*\*/

@Override // O(nlogn) use hashig O(logn)

**public** List<List<DLNode<Integer>>> pairOfGivenSum(DLNode<Integer> head, **int** x) {

head = mergeSort(head);

List<List<DLNode<Integer>>> ll = **null**;

List<DLNode<Integer>> l = **null**;

// and second to the end of DLL.

DLNode<Integer> first = head;

DLNode<Integer> second = head;

**while** (second.next != **null**)

second = second.next;

**while** (first != **null** && second != **null** && first != second && second.next != first) {

// pair found

**if** (first.k + second.k == x) {

**if** (ll == **null**)

ll = **new** ArrayList<>();

l = **new** ArrayList<>();

l.add(first);

l.add(second);

ll.add(l);

first = first.next;

second = second.prev;

} **else** {

**if** ((first.k + second.k) < x)

first = first.next;

**else**

second = second.prev;

}

}

**return** ll;

}

/\*\* 11. Insert value in sorted way in a sorted doubly linked list \*\*/

@Override

**public** DLNode<Integer> insertArrSorted(DLNode<Integer> head, Integer k[]) {

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

DLNode<Integer> temp = **new** DLNode<Integer>(k[i]);

**if** (head == **null**) {

head = tail = temp;

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

temp.next = head;

head.prev = temp;

head = temp;

} **else** {

DLNode<Integer> p = head;

**while** (p.next != **null**) {

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

temp.next = p.next;

p.next = temp;

temp.prev = p;

temp.next.prev = temp;

**break**;

}

p = p.next;

}

// add to last node

**if** (p.next == **null**) {

p.next = temp;

temp.prev = p;

}

}

}

**return** head;

}

/\*\* 12. Delete a Doubly Linked List node at a given position \*\*/

**public** DLNode<Integer> deletePos(DLNode<Integer> head, **int** pos) {

**if** (pos < 1)

**return** head;

**if** (head != **null**) {

DLNode<Integer> q = **null**;

**if** (pos == 1) {

q = head;

head = head.next;

**if** (head != **null**) // if only one node

head.prev = **null**;

q.next = **null**;

q = **null**;

} **else** {

DLNode<Integer> p = head;

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

;

/\* p.next is the deleting node \*/

**if** (p != **null** && p.next != **null**) {

q = p.next;

p.next = q.next;

**if** (q.next != **null**) // last node

q.next.prev = q.prev;

q.next = q.prev = **null**;

q = **null**;

} **else** {

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

}

}

}

**return** head;

}

/\*\*

\* 13. Count triplets in a sorted doubly linked list whose sum is equal to a

\* given value x

\*\*/

// function to count pairs whose sum equal to given 'value'

@Override

**public** **int** countPairs(DLNode<Integer> first, DLNode<Integer> second, **int** value) {

**int** count = 0;

**while** (first != **null** && second != **null** && first != second && second.next != first) {

// pair found

**if** (first.k + second.k == value) {

count++;

first = first.next;

second = second.prev;

}

**else** **if** ((first.k + second.k) > value)

second = second.prev;

**else**

first = first.next;

}

**return** count;

}

@Override

**public** **int** tripletROfGivenSum(DLNode<Integer> head, **int** x) {

// if list is empty

**if** (head == **null**)

**return** 0;

DLNode<Integer> current, first, last;

**int** count = 0;

last = head;

**while** (last.next != **null**)

last = last.next;

// traversing the doubly linked list

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

first = current.next;

count += countPairs(first, last, x - current.k);

}

// required count of triplets

**return** count;

}

/\*\* 14. Remove duplicates from a sorted doubly linked list \*\*/

@Override

**public** DLNode<Integer> removeDuplicate(DLNode<Integer> head) {

DLNode<Integer> prev, p = head.next;

prev = head;

DLNode<Integer> temp = **null**;

**while** (p != **null**) {

**if** (prev.k.equals(p.k)) {

**if** (p.next == **null**)// ie last node

{

p.prev = **null**;

prev.next = **null**;

p = **null**;

**break**;

}

temp = p;

prev.next = temp.next;

temp.next.prev = prev;

temp.prev = temp.next = **null**;

temp = **null**;

p = prev;// since p is deleted

}

prev = p;

p = p.next;

}

**return** head;

}

/\*\* 15. Delete all occurrences of a given key in a doubly linked list \*\*/

@Override

**public** DLNode<Integer> deleteAllOccuranceOfX(DLNode<Integer> head, **int** x) {

DLNode<Integer> prev, p;

prev = head;

DLNode<Integer> temp = **null**;

**while** (prev != **null** && prev.k.equals(x)) {

temp = prev;

prev = prev.next;

prev.prev = **null**;

temp.next = **null**;

temp = **null**;

}

head = prev;

// all node or except one deleted

**if** (prev == **null** || prev.next == **null**)

**return** head;

p = prev.next;

**while** (p != **null**) {

**if** (p.k.equals(x)) {

**if** (p.next == **null**)// ie last node

{

p.prev = **null**;

prev.next = **null**;

p = **null**;

**break**;

}

temp = p;

prev.next = temp.next;

temp.next.prev = prev;

temp.prev = temp.next = **null**;

temp = **null**;

p = prev;// since p is deleted

}

prev = p;

p = p.next;

}

**return** head;

}

/\*\* 16. Remove duplicates from an unsorted doubly linked list \*\*/

@Override // O(nlogn)

**public** DLNode<Integer> remveDuplicateFromNonSotedList(DLNode<Integer> head) {

head = mergeSort(head);

**return** removeDuplicate(head);

}

/\*\* 17. Sort the biotonic doubly linked list \*\*/

@Override

**public** DLNode<Integer> sortBitonicList(DLNode<Integer> head) {

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

**return** head;

DLNode<Integer> current = head.next;

**while** (current != **null**) {

**if** (current.k < current.prev.k)

**break**;

current = current.next;

}

// if true, then list is already sorted

**if** (current == **null**)

**return** head;

// spilt into two lists, one starting with 'head'

// and other starting with 'current'

current.prev.next = **null**;

current.prev = **null**;

// reverse the list starting with 'current'

current = reverseRecursive(current);

// merge the two lists and return the

// final merged doubly linked list

**return** sortedMerge(head, current);

}

/\*\* 18. Sort a k sorted doubly linked list \*\*/

// function to sort a k sorted doubly linked list

**public** **static** Comparator<DLNode<Integer>> *idComparator* = **new** Comparator<DLNode<Integer>>() {

@Override

**public** **int** compare(DLNode<Integer> p1, DLNode<Integer> p2) {

**return** p1.k - p2.k;

}

};

**public** DLNode<Integer> sortAKSortedDLL(DLNode<Integer> head, **int** k) {

// if list is empty

**if** (head == **null**)

**return** head;

Queue<DLNode<Integer>> pq = **new** PriorityQueue<DLNode<Integer>>(*idComparator*);

DLNode<Integer> last, newHead = **null**;

last = **null**;

// Create a Min Heap of first (k+1) elements from

// input doubly linked list

**for** (**int** i = 0; head != **null** && i <= k; i++) {

// push the node on to 'pq'

pq.add(head);

head = head.next;

}

**while** (!pq.isEmpty()) {

**if** (newHead == **null**) {

newHead = pq.peek();

newHead.prev = **null**;

last = newHead;

}

**else** {

last.next = pq.peek();

pq.peek().prev = last;

last = pq.peek();

}

pq.poll();

**if** (head != **null**) {

pq.add(head);

head = head.next;

}

}

last.next = **null**;

**return** newHead;

}

}

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

Test case:

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

**public** **class** IDLinkListTest {

**public** IDLinkList<Integer> idll = **null**;

**public** BTreeUitls<Integer> util = **new** BTreeUitls<>();

@Before

**public** **void** init() {

idll = **new** DLinkListImpl();

}

@Test

**public** **void** insertArrLastTest() {

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

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, a);

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

DLNode<Integer> tail = lap.getTail();

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

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

Assert.*assertTrue*(tail.k == 8);

Assert.*assertTrue*(tail.prev.k == 7);

// idll.print(head);

}

@Test

**public** **void** insertElementFirst() {

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

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, a);

lap = idll.insertElementLast(lap, 9);

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

DLNode<Integer> tail = lap.getTail();

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

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

Assert.*assertTrue*(tail.k == 9);

Assert.*assertTrue*(tail.prev.k == 8);

// idll.print(head);

}

@Test

**public** **void** insertElementLast() {

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

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, a);

lap = idll.insertElementFirst(lap, 1);

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

DLNode<Integer> tail = lap.getTail();

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

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

Assert.*assertTrue*(tail.k == 9);

Assert.*assertTrue*(tail.prev.k == 8);

}

@Test

**public** **void** insertAtPosition() {

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

DLNode<Integer> head = idll.insertArrSorted(**null**, a);

idll.print(head);

}

@Test

**public** **void** deleteFirstKTest() {

Integer k[] = {};

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, k);

Assert.*assertTrue*(lap == **null**);

Integer a[] = { 1 };

lap = idll.insertArrLast(**null**, a);

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

head = idll.deleteNodeFirstK(head, 1);

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

Integer b[] = { 10, 11 };

lap = idll.insertArrLast(**null**, b);

head = lap.getHead();

head = idll.deleteNodeFirstK(head, 11);

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

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

// idll.printList(head);

}

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

@Test

**public** **void** deleteAllXTest() {

DLNode<Integer> head = **null**;

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

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

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, a);

head = lap.getHead();

// idll.printList(head);

head = idll.deleteAllOccurance(head, 1);

// idll.printList(head);

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

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

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

head = **null**;

lap = idll.insertArrLast(**null**, b);

// idll.printList(head);

head = idll.deleteAllOccurance(lap.getHead(), 2);

// idll.printList(head);

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

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

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

}

/\* delete the last occurrence of node \*/

@Test

**public** **void** deleteLastKTest() {

DLNode<Integer> head = **null**;

Integer a[] = { 2, 4, 0, 1, 0, 2, 3, 1 };

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, a);

head = lap.getHead();

head = idll.deleteNodeLastK(head, 2);

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

head = idll.deleteNodeLastK(head, 2);

head = idll.deleteNodeLastK(head, 1);

head = idll.deleteNodeLastK(head, 10);

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

}

@Test

**public** **void** deletePosTest() {

Integer b[] = { 1 };

DLNode<Integer> head = **null**;

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, b);

head = lap.getHead();

head = idll.deletePos(head, 1);

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

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

head = **null**;

lap = idll.insertArrLast(**null**, a);

head = lap.getHead();

head = idll.deletePos(head, 1);

head = idll.deletePos(head, 2);

head = idll.deletePos(head, 7);

head = idll.deletePos(head, 7);

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

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

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

}

/\*\* 3. Reverse a Doubly Linked List \*\*/

@Test

**public** **void** reverseIterativeTest() {

DLNode<Integer> head = **null**;

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

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, a);

DLNode<Integer> hd = lap.getHead();

head = idll.reverseIterative(idll.reverseIterative(lap.getHead()));

**for** (; head != **null** && hd != **null**; head = head.next, hd = hd.next)

Assert.*assertTrue*(head.k == hd.k);

// idll.print(head);

}

@Test

**public** **void** reverseRecursiveTest() {

DLNode<Integer> head = **null**;

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

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, a);

DLNode<Integer> hd = lap.getHead();

head = idll.reverseRecursive(idll.reverseRecursive(lap.getHead()));

**for** (; head != **null** && hd != **null**; head = head.next, hd = hd.next)

Assert.*assertTrue*(head.k == hd.k);

// idll.print(head);

}

/\*\* 4. The Great Tree-List Recursion Problem. \*\*/

@Test

**public** **void** bstToListTest() {

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

BTNode<Integer> root = **null**;

**for** (**int** i = 0; i < arr.length; i++)

root = util.createBst(root, arr[i]);

root = idll.bstToList(root);

**while** (root.lt != **null**)

root = root.lt;

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

**int** i = 0;

**for** (; root != **null**; root = root.rt) {

Assert.*assertTrue*(root.t == a[i++]);

}

}

/\*\* 5. Copy a linked list with next and arbit pointer \*\*/

/\*\* 6. QuickSort on Doubly Linked List \*\*/

@Test

**public** **void** quickSortTest() {

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

Lap<DLNode<Integer>, DLNode<Integer>> lap = idll.insertArrLast(**null**, a);

DLNode<Integer> h = lap.getHead();

DLNode<Integer> t = h;

idll.print(h);

**while** (t.next != **null**)

t = t.next;

idll.quickSort(h, t);

// idll.print(h);

}

/\*\* 7. Swap Kth node from beginning with Kth node from end in a Linked List \*\*/

@Test

**public** **void** swapKthNodeFromBegAndEndTest() {

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

DLNode<Integer> h = idll.insertArrLast(**null**, a).getHead();

h = idll.swapKthNodeFromBegAndEnd(h, 2);

idll.print(h);

}

/\*\* 8. Merge Sort for Doubly Linked List \*\*/

@Test

**public** **void** mergeSortTest() {

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

DLNode<Integer> h = idll.insertArrLast(**null**, a).getHead();

h = idll.mergeSort(h);

idll.print(h);

}

/\*\* 9. Create a Doubly Linked List from a Ternary Tree \*\*/

@Test

**public** **void** createTernaryTreeTest() {

Integer a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };

TTreeNode<Integer> root = **null**;

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

root = idll.createTernaryTree(root, a[i]);

idll.printTTree(root);

}

@Test

**public** **void** createFromTernaryTreeTest() {

Integer a[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15 };

TTreeNode<Integer> root = **null**;

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

root = idll.createTernaryTree(root, a[i]);

TTreeNode<Integer> head = idll.createFromTernaryTree(root);

idll.printTTOfDl(head);

}

/\*\* 10. Find pairs with given sum in doubly linked list \*\*/

@Test

**public** **void** pairOfGivenSumTest() {

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

DLNode<Integer> head = idll.insertArrLast(**null**, a).getHead();

List<List<DLNode<Integer>>> ll = idll.pairOfGivenSum(head, 10);

Assert.*assertTrue*(ll.size() == 4);

}

@Test

**public** **void** insertArrSorted() {

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

DLNode<Integer> head = idll.insertArrSorted(**null**, a);

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

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

**while** (head.next.next != **null**)

head = head.next;

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

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

}

/\*\*

\* 13. Count triplets in a sorted doubly linked list whose sum is equal to a

\* given value x

\*\*/

@Test

**public** **void** tripletROfGivenSumTest() {

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

DLNode<Integer> head = idll.insertArrSorted(**null**, a);

**int** count = idll.tripletROfGivenSum(head, 17);

Assert.*assertTrue*(count == 2);

}

/\*\* 14. Remove duplicates from a sorted doubly linked list \*\*/

@Test

**public** **void** removeDuplicateTest() {

Integer a[] = { 4, 4, 4, 4, 6, 8, 8, 10, 12, 12 };

DLNode<Integer> head = idll.insertArrSorted(**null**, a);

head = idll.removeDuplicate(head);

idll.print(head);

}

/\*\* 15. Delete all occurrences of a given key in a doubly linked list \*\*/

@Test

**public** **void** deleteAllOccuranceOfXTest() {

Integer a[] = { 4, 4, 4, 5, 6, 4, 8, 10, 12, 4 };

DLNode<Integer> head = idll.insertArrSorted(**null**, a);

head = idll.deleteAllOccuranceOfX(head, 4);

idll.print(head);

}

/\*\* 16. Remove duplicates from an unsorted doubly linked list \*\*/

@Test

**public** **void** remveDuplicateFromNonSotedListTest() {

Integer a[] = { 4, 4, 10, 12, 12, 4, 4, 6, 8, 8 };

DLNode<Integer> head = idll.insertArrLast(**null**, a).getHead();

head = idll.remveDuplicateFromNonSotedList(head);

idll.print(head);

}

/\*\* 17. Sort the biotonic doubly linked list \*\*/

@Test

**public** **void** sortBitonicListTest() {

Integer a[] = { 2, 5, 7, 12, 10, 6, 4, 1 };

DLNode<Integer> head = idll.insertArrLast(**null**, a).getHead();

head = idll.sortBitonicList(head);

idll.print(head);

}

/\*\* 18. Sort a k sorted doubly linked list \*\*/

@Test

**public** **void** sortAKSortedDLLTest() {

Integer a[] = { 3, 6, 2, 12, 56, 8 };

DLNode<Integer> head = idll.insertArrLast(**null**, a).getHead();

head = idll.sortAKSortedDLL(head, 2);

idll.print(head);

}

}