public class LinkedList{

public Node head = null;

private class Node{

private int element;

private Node next;

private Node( int element , Node next){

this.element = element;

this.next = next;

}

}

public void insert( int value){

head = new Node (value , head);

}

public void dump(){

Node current = head;

while(current != null){

System.out.print(current.element);

if(current.next != null){

System.out.print(" --> ");

}

current = current.next;

}

System.out.println();

}

// delet Max Iterativ

public void deleteMax(){

if( head == null ){

return;

}

Node current = head;

Node previous = current;

Node currentMax = null;

Node previousMax = null;

int max = current.element;

while ( current !=null ){

if( max < current.element){

max = current.element;

previousMax = previous;

currentMax = current;

}

previous = current;

current = current.next;

}

if( max == head.element){

head = head.next;

return;

}

previousMax.next = currentMax.next;

}

public static void main(String args[]){

LinkedList l = new LinkedList();

int []arr = { 2 , 10 , 2 , 2 , 8 };

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

l.insert(arr[i]);

}

l.dump();

l.deleteMax();

l.dump();

}

}

public class LinkedList{

public Node head = null;

public int anzahlNode(){

Node cur = head;

int erg =0;

while(cur!=null){

erg++;

cur=cur.next;

}

return erg;

}

public void verdoppelt(){

Node current = head;

while(current !=null){

Node doppel = new Node(current.element);

doppel.next = current.next;

current.next = doppel;

current = current.next.next;

}

}

public boolean swap(Node n1, Node n2) {

if (n1==null || n2==null) return false;

if (n1.next!=n2) return false;

// search predecessor of n1

Node pred = head;

while (pred.next != n1){

pred = pred.next;

if (pred==null) return false;

}

pred.next = n2;

n1.next = n2.next;

n2.next = n1;

return true;

}

// erste und letzte vertauschen

public void switchFrontBack(){

Node current = head;

Node prev = head;

while(current.next!=null) {

prev = current;

current = current.next;

}

if( head.next==current){

current.next = head;

head.next = null;

head = current;

}else{

current.next=head.next;

prev.next = head;

head.next = null;

head = current;

}

}

public void swap(int x , int y){

Node curX=head;

Node prevX=null;

if (x==y) return;

while(curX.element !=x && curX!=null){

prevX=curX;

curX = curX.next;

}

Node curY=head;

Node prevY=null;

while(curY.element !=y && curY!=null){

prevY=curY;

curY = curY.next;

}

if(prevX !=null){

prevX.next = curY;

}

else{

head = curY;

}

if(prevY !=null){

prevY.next = curX;

}

else{

head = curX;

}

Node tmp = curX.next;

curX.next = curY.next;

curY.next = tmp;

}

public void sort(){

Node current = head;

boolean getaucht;

do{

getaucht = false;

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

if(current.element>current.next.element){

swap(current.element , current.next.element);

getaucht =true;

}

current = current.next;

}

current = head;

}

while(getaucht);

}

// delet Max Iterativ

public void deleteMax(){

if( head == null ){

return;

}

Node current = head;

Node previous = current;

Node currentMax = null;

Node previousMax = null;

int max = current.element;

while ( current !=null ){

if( max < current.element){

max = current.element;

previousMax = previous;

currentMax = current;

}

previous = current;

current = current.next;

}

if( max == head.element){

head = head.next;

return;

}

previousMax.next = currentMax.next;

)

public void deleteMin(){

Node min = head;

Node current1 = head.next;

while(current1!=null){

if(current1.element<min.element){

min = current1;

}

current1 = current1.next;

}

if (min == head){

head = head.next;

}

Node current2 = head;

while(current2 != null && current2.next != null){

if(current2.next.element == min.element){

current2.next = current2.next.next;

}

current2=current2.next;

}

}

public void inseAtEnd(Node nd){

if( head == null){

head = nd ;

}

Node current = head ;

while(current.next != null){

current = current.next;

}

current.next= nd;

}

public void insertAt(Node newNode, int pos){

Node current = head;

for(int i = 1; i <pos; i++){

current = current.next;

}

if(pos == 0){

newNode.next = head;

head = newNode;

}else{

newNode.next = current.next;

current.next = newNode;

}

}

public int findMax(){

Node max = head;

Node current = head.next;

while(current !=null ){

if ( current.element > max.element ){

max = current;

}

current = current.next;

}

return max.element;

}

public int maxRekursiv(Node node , int max ){

if(node==null){

return 0;

}

if(node.next>max){

max = node.next;

}

return maxRekursiv(node.next , max);

}

public void insert(Node n) {

n.next = head;

head = n;

}

public void delete(int nummer) {

Node current = head;

// Liste leer?

if (head==null) return;

// Sonderfall, soll das erste Element geloescht werden?

if (head.element == nummer) {

head = head.next;

return ;

}

// current!=null ist wichtig, falls das letzte Element

// in der Liste geloescht wird

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

if ( current.next.element == nummer) {

current.next = current.next.next;

}

current = current.next;

}

}

public void dump() {

Node current = head;

System.out.println("Ausgabe der kompletten Liste");

while (current!=null) {

System.out.println(" Element=" + current.element);

current = current.next;

}

}

}

private static void flip(Node first, Node second) {

int tmp = first.value;

first.value = second.value;

second.value = tmp;

)

public void Bubblesort() {

boolean alreadySorted = false;

while(!alreadySorted) {

alreadySorted = true;

Node current = head;

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

if(current.value > current.next.value) {

alreadySorted = false;

flip(current, current.next);

}

current = current.next;

}

}

}

public class BinSearchTree {

public BinNode root;

public boolean serchRekursiv(int v){

if ( root == null) return false;

return serchRekursiv(root, v);

}

public boolean serchRekursiv (BinNode bn,int v){

if(bn == null) return false;

if(v>bn.element) return serchRekursiv(bn.right,v);

if(v<bn.element) return serchRekursiv(bn.left,v);

return true;

}

// Tiefe linke Seite

public int getLeftHeight(BinNode n){

if(n==null) return 0;

int height=0;

while(n.left!=null){

height++;

n = n.left;

}

return height;

}

// Tiefe rechte Seite

public int getRightHeight(BinNode n){

if(n==null) return 0;

int height=0;

while(n.right!=null){

height++;

n = n.right;

}

return height;

}

public int tief(){

if ( root == null) return 0;

return tief(root);

}

public int tief (BinNode bn ){

if ( bn == null) return 0;

int tiefL = tief(bn.left);

int tiefR = tief(bn.right);

return Math.max(tiefL , tiefR)+1;

}

public int max(){

if (root== null) return -1;

return max(root);

}

public int maxRekursiv(BinNode bn){

if (bn.right == null){

return bn.element;

}

return maxRekursiv(bn.right);

}

// MAX iterativ

public int max(BinNode bn){

while(root.right !=null){

root=root.right;

}

return root.element;

}

public int countK(){

return countKind(root);

}

public int countKind(BinNode bn){

int zahl =0;

if(bn.left == null && bn.right == null){

zahl++;

}

if (bn.left != null){

zahl = zahl +countKind(bn.left);

}

if (bn.right != null){

zahl = zahl +countKind(bn.right);

}

return zahl;

}

//Anzhl Knoten Iterativ

public int countNodsIterativ(){

int nright = 0;

int nleft = 0 ;

if (root == null) return 0;

while(root!= null) {

if(root.right!= null){

nright++;

root = root.right;

}

if (root.left!= null){

nleft++;

root= root.left;

}

}

return nleft + nright + 1 ;

}

public int count(){

if ( root == null) return 0;

return countNode(root);

}

public int countNode(BinNode bn){

int zaehler=1;

if (bn.left !=null){

zaehler = zaehler + countNode(bn.left);

}

if (bn.right !=null){

zaehler = zaehler + countNode(bn.right);

}

return zaehler;

}

// Anzahl Knoten , die geraden Schlüssel haben

public int countNodes(BinNode bn) {

int n = 1 ;

n = countNode(bn.left) + countNode(bn.right);

if(bn.element %2 != 0){

return n+1;

else

retern n ;

}

// Loeschen eines Knoten

public void delete(int element) {

root = delete(root, element);

}

// rekursive Funktion zum Loeschen eines beliebigen Knoten

private BinNode delete(BinNode bn, int element) {

if (bn==null)

return null;

if (element < bn.element)

bn.left = delete(bn.left, element);

else if (element > bn.element)

bn.right = delete(bn.right, element);

else {

// zu löschender Knoten gefunden?

if (bn.right==null) return bn.left;

if (bn.left==null) return bn.right;

// jetzt behandeln wir den Fall, dass der zu loeschende

// Knoten 2 Nachfolger hat

BinNode tmp = bn;

// suche Minimum im rechten Teilbaum des zu loeschenden Knoten

bn = findMin(tmp.right);

// Minimum im rechten Teilbaum loeschen

bn.right = deleteMin(tmp.right);

// links-Zeiger aktualisieren

bn.left = tmp.left;

}

return bn;

}

// Minimum ab 'bn' loeschen

private BinNode deleteMin(BinNode bn) {

if (bn.left == null)

return bn.right;

bn.left = deleteMin(bn.left);

return bn;

}

// Minimum ab 'root' rekursiv suchen

public BinNode findMinimum() {

return findMin(root);

}

// Minimum ab 'bn' rekursiv suchen

private BinNode findMin(BinNode bn) {

if (bn.left==null)

return bn;

else

return findMin(bn.left);

}

// Maximum ab 'bn' rekursiv suchen

public int findMaximumRec() {

return findMax(root);

}

// Maximum ab 'bn' rekursiv suchen

private int findMax(BinNode bn) {

if (bn.right==null)

return bn.element;

else

bn = bn.right;

return findMax(bn);

}

// Maximum ab 'root' iterativ suchen

public int findMaximumIt() {

BinNode cur = root;

while (cur.right != null) {

cur = cur.right;

}

return cur.element;

}

public boolean insertNode(BinNode bn) {

BinNode child=null, parent=null;

// Knoten suchen, nach welchem eingefuegt wird

child = root;

while( child != null) {

parent = child;

if (bn.element == child.element) return false;

else if (bn.element < child.element) child = child.left;

else child = child.right;

}

// Baum leer?

if (parent==null) root = bn;

// Einfuegen nach parent, links

else if (bn.element < parent.element) parent.left = bn;

// Einfuegen nach parent, rechts

else parent.right = bn;

return true;

}

public BinNode findNode(int element) {

BinNode n = root;

while (n != null) {

if (element == n.element) { return n; }

else if (element < n.element) { n = n.left; }

else { n = n.right; }

}

return null;

}

}

public String toString() {

return root.toString();

}

// Ausgabe des Baums

public void print() {

// Puffer fuer Baum

char buffer[][] = new char[25][80];

// Puffer initialisieren

for (int y=0; y<25; y++) {

for (int x=0; x<80; x++) {

buffer[y][x] = ' ';

}

}

// Baum auslesen

int tiefe = print(root, 40, 1, buffer);

// Ausgabe des Baums

for (int y=0; y<tiefe; y++) {

for (int x=0; x<80; x++) {

System.out.print(buffer[y][x]);

}

System.out.println(); System.out.println();

}

}

// rekursive Hilfsfunktion zur Ausgabe des Baums

private int print(BinNode node, int indent, int tiefe, char buffer[][]) {

int ltiefe, rtiefe;

// Abbruch der Rekursion

if (node==null)

return tiefe;

/\* Knoten "ausgeben" \*/

String str = Integer.toString(node.element);

int startpos = str.length() / 2;

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

buffer[tiefe][indent - startpos + i] = str.charAt(i);

}

/\* Linken und rechten Teilbaum ausgeben \*/

ltiefe = print(node.left, indent-12/tiefe, tiefe+1, buffer);

rtiefe = print(node.right, indent+12/tiefe, tiefe+1, buffer);

if (ltiefe > rtiefe) return ltiefe;

return rtiefe;

}

}