Stack

```
/**
 * 2) Napisati operacije ubaci i izbaci nad stakom koji je
implementiran preko
 * niza. (15 poena)
 */
public class ArrayStack {
	private int size = 10;

	private Object[] array = new Object[size];

	private int count = 0;

	/*
	 * TODO implementaciju konstruktora opet ostavljamo gospodi
iz struktura
	 * podataka
	 */

	public void push(Object o) {
		if (count >= size)
			return;// stek je pun
		array[count++] = o;

	}

	public Object pop() {
		if (count == 0)
			return null;
		return array[--count];
	}

/**
 * 1) Napišite operacije gurni i povuci nad stakom koji je
implementiran kao
 * jednostruko spegnuta lista.
 */
public class LinkedListStack {
	LinkedList list = new LinkedList();

	/*
	 * TODO implementaciju konstruktora opet ostavljamo gospodi
iz struktura
	 * podataka
	 */public void push(Object o) {
		list.Add(o);
	}

	public Object pop() {
		return list.Remove();
```

```
	}

/**
	 * 4. Napisati funkciju transformisi(STAK s1, STAK* s2)
koja æe od staka
	 * koji je implementiran kao jednostruko spregunuta lista
formirati novi
	 * stak koji je implementiran preko niza.
	 */
	public static
DataStructures.LinearStructures.Stack.ArrayStack transform(
			LinkedListStack s1) {
		LinkedListStack temp = new LinkedListStack();
		int count = 0;
		Object element;
		while ((element = s1.Pop()) != null) {
			temp.Push(element);
			count++;
		}

		DataStructures.LinearStructures.Stack.ArrayStack stek
= new DataStructures.LinearStructures.Stack.ArrayStack(
				count);
		while ((element = temp.Pop()) != null) {
			stek.Push(element);
		}
		return stek;

	}

	public static void clone(DataStructures.Interfaces.IStack
izvor,
			DataStructures.Interfaces.IStack klon) {
		Object temp = izvor.Pop();
		if (temp == null)
			return;
		clone(izvor, klon);
		klon.Push(temp);
```

Redovi

```
/**
 * 1) Napisati operacije ubaci i izbaci nad redom koji je
implementiran preko
 * niza. (15 poena)
 */
public class ArrayQueue {
	private int size = 10;

	private Object[] array = new Object[size];

	private int start = -1;

	private int end = -1;

	private int count = 0;

	/*
	 * TODO implementaciju konstruktora opet ostavljamo gospodi
iz struktura
	 * podataka
	 */

	public void push(Object o) {
		if (count >= size)
			return;// red je pun
		/*
		 * Postoje dva pokazivaca na index u nizu gde red
pocinje i na index gde
		 * se zavrsava. Ostatak pri deljenju se koristi da se
index resetuje na
		 * pocetak kad se dodje do kraja niza.
		 */
		array[end = ++end % size] = o;
		count++;

	}

	public Object pop() {
		if (count == 0)
			return null;// red je prazan
		count--;
		return array[start = ++start % size];
	}

/**
 * 2) Napišite operacije gurni i povuci nad redom koji je
implementiran kao
 * jednostruko spegnuta lista.
 */
public class LinkedListQueue {
```

```
      LinkedList list = new LinkedList();

      /*
       * TODO implementaciju konstruktora opet ostavljamo gospodi
iz struktura
       * podataka
       */public void gurni(Object o) {
            list.Add(o);
      }

      public Object povuci() {
            return list.RemoveFromEnd();
      }
```

Liste

```
public class CharNode {
      public char data;

      public CharNode next;

      public CharNode(char data, CharNode next) {
            super();
            this.data = data;
            this.next = next;
      }

      public CharNode(char data) {
            super();
            this.data = data;
      }
}

      /**
       * 3) Od dve jednostruke spregnute liste celih brojeva
sortiranih u rastuæem
       * redosledu formirajte treæu sortiranu u opadajuæem
redosledu i to tako da
       * efikasnost algoritma bude O(n+m), gde su n i m brojevi
elemenata u datim
       * listama. Date dve liste treba da ostanu kakve su bile
(tj. ne uništavaju
       * se).
       */
      public static LinkedList mergeAsc(LinkedList a, LinkedList
b) {
            LinkedList list = new LinkedList();
            while (!(a.PeekCurrent() == null && b.PeekCurrent() ==
null)) {
                  Integer temp = (Integer) a.PeekCurrent();
                  while (b.PeekCurrent() != null
                  // znak < jer se dodaje na head
                              && (((Integer)
b.PeekCurrent()).intValue() < temp
                                          .intValue()) || temp ==
null) {
                        list.Add(b.PeekCurrent());
                        b.MoveNext();
                  }
                  if (b.PeekCurrent() != null) {
                        list.Add(temp);
                        a.MoveNext();
                  }

            }
            return list;
```

```
	}

	/**
	 * ISTO TO SAMO AKO JE DAT POKAZIVAC NA PRVI CVOR LISTE 3) Od dve
	 * jednostruke spregnute liste celih brojeva sortiranih u rastuæem redosledu
	 * formirajte treæu sortiranu u opadajuæem redosledu i to tako da efikasnost
	 * algoritma bude O(n+m), gde su n i m brojevi elemenata u datim listama.
	 * Date dve liste treba da ostanu kakve su bile (tj. ne uništavaju se).
	 */
	public static ListIntNode mergeAscending(ListIntNode a, ListIntNode b) {
		ListIntNode head = null;
		while (a != null) {
			/*
			 * Uzimamo jedan iz liste a; sve dok u b ima manjih (manjih jer se
			 * dodaju na pocetak liste) stavljamo ove iz b; kada vise nema
			 * manjih, stavljamo ovaj iz a;
			 */
			while (b != null && b.Data < a.Data) {
				head = new ListIntNode(b.Data, head);
				b = b.Next;
			}
			head = new ListIntNode(a.Data, head);
			a = a.Next;
		}
		/*
		 * posto smo izpraznili listu a, sve iz b dodajemo
		 */
		while (b != null) {
			head = new ListIntNode(b.Data, head);
			b = b.Next;
		}
		return head;
	}

	/**
	 * 4) Od dve jednostruke spregnute liste celih brojeva sortiranih u
	 * opadajuæem redosledu formirajte treæu sortiranu u rastuæem redosledu i to
	 * tako da efikasnost algoritma bude O(n+m), gde su n i m brojevi elemenata
	 * u datim listama. Date dve liste treba da ostanu kakve su bile (tj. ne
	 * uništavaju se).
```

```
       */
      public static LinkedList mergeDesc(LinkedList a, LinkedList
b) {
            LinkedList list = new LinkedList();
            while (!(a.PeekCurrent() == null && b.PeekCurrent() ==
null)) {
                  Integer temp1 = (Integer) a.PeekCurrent();
                  while (b.PeekCurrent() != null
                              && ((Integer)
b.PeekCurrent()).intValue() > temp1
                                          .intValue()) {
                        list.Add(b.PeekCurrent());
                        b.MoveNext();
                  }
                  list.Add(temp1);
                  a.MoveNext();

                  if (a.PeekCurrent() == null)
                        while (b.PeekCurrent() != null) {
                              list.Add(b.PeekCurrent());
                              b.MoveNext();
                        }

            }
            return list;
      }

      /**
       * ISTO TO SAMO AKO JE DAT POKAZIVAC NA PRVI CVOR LISTE 4)
Od dve
       * jednostruke spregnute liste celih brojeva sortiranih u
opadajuæem
       * redosledu formirajte treæu sortiranu u rastuæem
redosledu i to tako da
       * efikasnost algoritma bude O(n+m), gde su n i m brojevi
elemenata u datim
       * listama. Date dve liste treba da ostanu kakve su bile
(tj. ne uništavaju
       * se).
       */
      public static ListIntNode mergeDescending(ListIntNode a,
ListIntNode b) {
            ListIntNode head = null;
            while (a != null) {
                  /*
                   * Uzimamo jedan iz liste a; sve dok u b ima
vecih (vecih jer se
                   * dodaju na pocetak liste) stavljamo ove iz b;
kada vise nema
                   * vecih, stavljamo ovaj iz a;
                   */
                  while (b != null && b.Data > a.Data) {
```

```
                        head = new ListIntNode(b.Data, head);
                        b = b.Next;
                  }
                  head = new ListIntNode(a.Data, head);
                  a = a.Next;
            }
            /*
             * posto smo izpraznili listu a, sve iz b dodajemo
             */
            while (b != null) {
                  head = new ListIntNode(b.Data, head);
                  b = b.Next;
            }
            return head;
      }

      /**
       * 8) Ako je skup je implementiran preko jednostruko
spregnute liste,
       * napišite funkciju koja prihvata kao argumente dva skupa
i vraæa treæi
       * koji je njihova razlika/presek/unija.
       *
       * Zbog implementacije klase LinkedList, nemoguce je ovo
resiti, pa cemo
       * promeniti tekst tako da bude dat pokazivac na prvi
element skupa.
       */
      protected static boolean contains(ListNode list, ListNode
element) {
            if (list == null)
                  return false;
            if (list.Data.equals(element.Data))
                  return true;
            return contains(list.Next, element);
      }

      public static ListNode razlika(ListNode a, ListNode b) {
            ListNode head = null;

            while (a != null) {
                  if (!contains(b, a)) {
                        head = new ListNode(a.Data, head);
                  }
                  a = a.Next;
            }

            return head;
      }

      public static ListNode presek(ListNode a, ListNode b) {
            ListNode head = null;
```

```
            while (a != null) {
                  if (contains(b, a)) {
                        head = new ListNode(a.Data, head);
                  }
                  a = a.Next;
            }
            return head;
      }

      public static ListNode unija(ListNode a, ListNode b) {
            ListNode head = null;
            while (a != null) {
                  if (!contains(head, a)) {
                        head = new ListNode(a.Data, head);

                  }
                  a = a.Next;
            }
            while (b != null) {
                  if (!contains(head, b)) {
                        head = new ListNode(b.Data, head);

                  }
                  b = b.Next;
            }

            return head;
      }

      /**
       * 7) Svaki element jedne jednostruko spregnute liste
sadrži pokazivaè na
       * drugu jednostruko spregnutu listu. Ako je dat pokazivaè
na prvi element
       * prve liste, koliki je broj/suma elemenata u svim listama.
       */
      public static int count2D(ListNode node) {
            int count = 0;
            while (node != null) {
                  LinkedList list = (LinkedList) node.Data;
                  while (list.Remove() != null)
                        count++;
                  node = node.Next;
            }
            return count;
      }

      // ~---------CIKLICNA
LISTA-------------------------------------------------

      /**
```

```
      * 1. Napišite funkciju int uporedi(element *L1, element*
L2) koja poredi
      * dva stringa koja su implementirana preko jednostruko
spregnute cilične
      * liste. Funkcija vraæa –1 ako je string predstavljen
preko L1 manji od
      * stringa predstavljenog preko L2, 0 ako su jednaki i 1
ako je string
      * predstavljen L1 veæi.
      */
     public static int uporedi(ListNode a, ListNode b) {
          ListNode iterA = a;
          ListNode iterB = b;

          int sizeA = 1;
          int sizeB = 1;
          while (iterA.Next != a) {
               iterA = iterA.Next;
               sizeA++;
          }
          while (iterB.Next != b) {
               iterB = iterB.Next;
               sizeB++;
          }
          if (sizeA == sizeB)
               return 0;
          if (sizeA > sizeB)
               return 1;
          else
               return -1;
     }

     /**
      * AKO SE MISLI NA POREDJENJE STRINGOVA PO VREDNOSTI SLOVA
      *
      * 1. Napišite funkciju int uporedi(element *L1, element*
L2) koja poredi
      * dva stringa koja su implementirana preko jednostruko
spregnute cilične
      * liste. Funkcija vraæa –1 ako je string predstavljen
preko L1 manji od
      * stringa predstavljenog preko L2, 0 ako su jednaki i 1
ako je string
      * predstavljen L1 veæi.
      */
     public static int compare(CharNode a, CharNode b) {
          CharNode iterA = a;
          CharNode iterB = b;

          do {
               if (iterA.data < iterB.data)
                    return -1;
```

```
                if (iterA.data > iterB.data)
                      return 1;
                // ovde mora jednostruki operator &
          } while ((iterA = iterA.next) != a & (iterB =
iterB.next) != b);

          if (iterA == a && iterB != b)
                return -1;
          if (iterA != a && iterB == b)
                return 1;
          return 0;
     }

     /**
      * 2) Dat je samo pokazivaè na neki èvor jednostruko
spregnute cikliène
      * liste koja je sortirana u opadajuæem redosledu. Napisati
funkciju koja æe
      * izbaciti dati element iz liste.
      */
     public static void remove(ListNode node) {
          ListNode previous = node;

          while (previous.Next != node) {
                previous = previous.Next;
          }

          previous.Next = node.Next;

     }

public class DoubleLinkedListIntNode {
     public int Data;

     public DoubleLinkedListIntNode Next;

     public DoubleLinkedListIntNode Previous;

     public DoubleLinkedListIntNode(int aData) {
          this(aData, null, null);
     }

     public DoubleLinkedListIntNode(int aData,
DoubleLinkedListIntNode aNext) {
          this(aData, aNext, null);
     }

     public DoubleLinkedListIntNode(int aData,
DoubleLinkedListIntNode aNext,
                DoubleLinkedListIntNode aPrevious) {
          Data = aData;
          Next = aNext;
```

```
            Previous = aPrevious;
      }
}


/**
       * 1) Dat je pokazivaè na poèetni èvor dvostruko spregnute
liste sortirane u
       * rastuæem redosledu koja sadrži pozitivne cele brojeve.
Napisati funkciju
       * koja æe između svih onih elementa liste koji se po
vrednosti razlikuju za
       * više od 1 ubaciti u datu listu nove elemente tako da
lista posle poziva
       * operacije ima u sebi sukcesivne cele brojeve. Na primer
ako lista sadrži
       * 3, 5, 8 nakon poziva ove opercije sadržaæe 3, 4, 5, 6,
7, 8
       */
      public static void fill(DoubleLinkedListIntNode node) {
            if (node.Next == null)
                  return;
            if (node.Next.Data - node.Data > 1) {
                  /*
                   * Ako se tekuci i sledeci cvor razlikuju za vise
od 1, ubaci jedan
                   * cvor izmedju i pozovi rekurzivno funkciju nad
ubacenim cvorom
                   */
                  DoubleLinkedListIntNode temp = new
DoubleLinkedListIntNode(
                              node.Data + 1);
                  // nema provere za null, jer se uvek ubacuje
izmedju dva
                  temp.Next = node.Next;
                  temp.Previous = node;
                  node.Next.Previous = temp;
                  node.Next = temp;
            }
            fill(node.Next);

      }

      /**
       * 2) Dat je pokazivaè na poèetni èvor dvostruko spregnute
liste sortirane u
       * opadajuæem redosledu koja sadrži pozitivne cele brojeve.
Napisati
       * funkciju koja æe između svih onih elementa liste koji se
po vrednosti
       * razlikuju za više od 1 ubaciti u datu listu nove
elemente tako da lista
```

```
       * posle poziva operacije ima u sebi sukcesivne cele
brojeve. Na primer ako
       * lista sadrži 10, 7, 5 nakon poziva ove opercije sadržaæe
10, 9, 8, 7, 6,
       * 5
       */
      public static void fillAgain(DoubleLinkedListIntNode node) {
            if (node.Next == null)
                  return;
            if (node.Data - node.Next.Data > 1) {
                  /*
                   * Ako se tekuci i sledeci cvor razlikuju za vise
od 1, ubaci jedan
                   * cvor izmedju i pozovi rekurzivno funkciju nad
ubacenim cvorom
                   */
                  DoubleLinkedListIntNode temp = new
DoubleLinkedListIntNode(
                              node.Data - 1);
                  // nema provere za null, jer se uvek ubacuje
izmedju dva
                  temp.Next = node.Next;
                  temp.Previous = node;
                  node.Next.Previous = temp;
                  node.Next = temp;
            }
            fillAgain(node.Next);

      }

      /**
       * 3) Dat je pokazivaè na neki èvor dvostruko spregnute
liste. Napisati
       * funkciju koja æe vratiti broj elemenata u listi.
       *
       */
      public static int count(DoubleLinkedListIntNode n) {
            if (n == null)
                  return 0;

            return 1 + before(n.Previous) + after(n.Next);
      }

      /**
       * broji koliko ih je levo od datog cvora
       */
      private static int before(DoubleLinkedListIntNode n) {
            if (n == null)
                  return 0;
            return 1 + before(n.Previous);
      }
```

```
	/**
	 * broji koliko ih je desno od datog cvora
	 */
	private static int after(DoubleLinkedListIntNode n) {
		if (n == null)
			return 0;
		return 1 + after(n.Next);
	}

	/**
	 * 4) Dat je pokazivaè na neki element dvostruko spregnute
liste celih
	 * brojeva. Napisati funkciju koja æe izbaciti iz liste
onaj element koji
	 * sadrži najmanji/najveæi broj u listi.
	 */
	public static int min(DoubleLinkedListIntNode node) {
		if (node == null)
			return Integer.MIN_VALUE;
		/*
		 * prvo se pronadje najmanji element i on se snimi u
promenjivu min
		 */
		DoubleLinkedListIntNode min = findMin(node);
		/*
		 * onda se min izbacuje tako sto se prethodnom stavi
pokazivac na
		 * sledeci posle min-a, pod uslovom da prethodni nije
null (ovde nema
		 * veze da li je sledeci null) ,a onda se sledecem
stavi pokazivac na
		 * prethodni pre min, pod uslovom da sledeci nije
null(sada nema veze da
		 * li je prethodni null)
		 *
		 *
		 */
		if (min.Previous != null)
			min.Previous.Next = min.Next;
		if (min.Next != null)
			min.Next.Previous = min.Previous;
		return min.Data;
	}

	private static DoubleLinkedListIntNode
findMin(DoubleLinkedListIntNode node) {
		// postavlja tekuci element za minimum
		DoubleLinkedListIntNode min = node;

		int minValue = node.Data;
		// ide do levog kraja i trazi manji
		DoubleLinkedListIntNode iterator = node;
```

```
            while (iterator.Next != null) {
                  iterator = iterator.Next;
                  if (iterator.Data < minValue) {
                        min = iterator;
                        minValue = iterator.Data;
                  }
            }
            iterator = node;
            // ide do desnog kraja i trazi jos manji
            while (iterator.Previous != null) {
                  iterator = iterator.Previous;
                  if (iterator.Data < minValue) {
                        min = iterator;
                        minValue = iterator.Data;
                  }
            }
            return min;
      }

      /**
       * 5. Dat je pokazivaè na neki èvor dvostruko spregnute
liste celih brojeva
       * koja sigurno sadrži najmanje 4 elementa. Napisati
funkciju ubaciNti(int
       * A, int N) koja æe ubaciti novi element sa sadržajem A i
to tako da on
       * bude na N-toj poziciji od poèetka .
       */
      public static boolean ubaciNti(DoubleLinkedListIntNode
node, int a, int n) {
            if (node == null)
                  return false;
            // resetuje listu na pocetak
            while (node.Previous != null)
                  node = node.Previous;
            DoubleLinkedListIntNode temp = new
DoubleLinkedListIntNode(a);
            if (n == 0) {
                  temp.Previous = null;
                  temp.Next = node;
                  if (temp.Next != null)
                        temp.Next.Previous = temp;
                  return true;
            }
            int i = 0;
            while (i++ < n - 1) {
                  // ako dodje do null pre n-te pozicije, staje jer
ne moze da stavi
                  // na n-tu
                  if (node.Next == null)
                        return false;
                  node = node.Next;
```

```
            }

            temp.Previous = node;
            temp.Next = node.Next;
            if (node.Next != null)
                  node.Next.Previous = temp;
            node.Next = temp;
            return true;
      }

      /**
       * 8. Dat je pokazivaè na neki èvor dvostruko spregnute
liste koji je jedini
       * ulazni podatak. Napisati funkciju koja æe ukazani
element prebaciti na
       * drugo/pretposlednje mesto u listi.
       */
      public static void moveToSecond(DoubleLinkedListIntNode
node) {
            DoubleLinkedListIntNode iterator = node;
            while (iterator.Previous != null)
                  iterator = iterator.Previous;
            // sada je iterator prvi element liste

            // ako je prvi taj koji treba da se prebaci, samo
zameni prvi i drugi
            if (iterator == node) {
                  DoubleLinkedListIntNode prvi, drugi;
                  prvi = iterator;
                  drugi = iterator.Next;

                  drugi.Previous = null;
                  prvi.Next = drugi.Next;
                  prvi.Previous = drugi;
                  drugi.Next = prvi;
                  if (prvi.Next != null)
                        prvi.Next.Previous = prvi;

                  return;
            }

            // izbacujemo node
            node.Previous.Next = node.Next;
            if (node.Next != null)
                  node.Next.Previous = node.Previous;

            // ubacujemo node posle prvog
            node.Next = iterator.Next;
            node.Previous = iterator;
            if (iterator.Next != null)
                  iterator.Next.Previous = node;
            iterator.Next = node;
```

```
	}

	// ~-----CIKLICNA
LISTA-----------------------------------------
	/**
	 * 1. Dat je pokazivaè na neki èvor dvostruko spregnute
cikliène liste koja
	 * je sortirana u opadajuæem redosledu. Napisati funkciju
koja æe vratiti
	 * pokazivaè na poslednji/pretposlednji/drugi element u
listi.
	 */
	public static DoubleLinkedListIntNode
last(DoubleLinkedListIntNode node) {
		while (node.Next.Data < node.Data)
			node = node.Next;
		return node;
	}

	public static DoubleLinkedListIntNode pretposlednji(
			DoubleLinkedListIntNode node) {
		while (node.Next.Next.Data < node.Next.Data)
			node = node.Next;
		return node;
	}

	public static DoubleLinkedListIntNode
second(DoubleLinkedListIntNode node) {
		while (node.Previous.Data > node.Data)
			node = node.Previous;
		return node.Next;
	}

	/**
	 * 2. Dat je pokazivaè na neki èvor dvostruko spregnute
cikliène liste koja
	 * je sortirana u rastuæem redosledu. Napisati funkciju
koja æe vratiti
	 * pokazivaè na prvi/drugi element u listi.
	 */
	public static DoubleLinkedListIntNode
first(DoubleLinkedListIntNode node) {
		while (node.Previous.Data < node.Data)
			node = node.Previous;
		return node;
	}

/**
 * 6. Data je dvostruko spregnuta lista celih brojeva sortirana u
rastuæem
 * redosledu i pokazivaè p koji pokazuje na poslednji pronaðeni
element u listi.
```

```
 * Definiše ovu strukturu kao apstraktni tip i implementirajte
algoritam za
 * pretraživanje koji koristi i održava pokazivaè p.
 */
public abstract class DLList {
	private DoubleLinkedListIntNode pointer;

	/*
	 * TODO Pri implementaciji konstruktora i metode add i
remove treba se
	 * pobrinuti da pointer ne bude null;
	 */
	public abstract void add(int data);

	public abstract void remove(int data);

	public DoubleLinkedListIntNode find(int data) {

		if (data > pointer.Data) {
			DoubleLinkedListIntNode iterator = pointer;
			while (iterator != null && data >= pointer.Data)
{
				if (iterator.Data == data) {
					pointer = iterator;
					return iterator;
				}
				iterator = iterator.Next;
			}
			return null;

		} else {

			DoubleLinkedListIntNode iterator = pointer;
			while (iterator != null && data <= pointer.Data)
{
				if (iterator.Data == data) {
					pointer = iterator;
					return iterator;
				}
				iterator = iterator.Previous;
			}
			return null;

		}
	}
```

Sortiranje

```
/**
       * Klasicno pretrazivanje - rastuci redosled. Nalazi se
minimum u ostatku
       * niza i onda se zamenjuju mesta tekuceg elementa i
minimuma. Kompleksnost
       * O(n^2) zbog broja poredjenja, gledano u odnosu na broj
premestanja je
       * O(n)
       *
       * Efikasnost O(n^2/4)
       *
       * @param array
       *            niz koji treba da se sortira
       * @return niz sortiran u rastucem redosledu
       */
      public static int[] selectionSortAsc(int[] array) {
            if (array.length == 0)
                  return null;
            for (int i = 0; i < array.length; i++) {
                  int minInd = i;
                  int min = array[i];
                  for (int j = i; j < array.length; j++)
                        if (array[j] < min) {
                              minInd = j;
                              min = array[j];
                        }
                  if (minInd != i) {
                        int temp = array[i];
                        array[i] = array[minInd];
                        array[minInd] = temp;
                  }
            }

            return array;
      }

      /**
       * Klasicno pretrazivanje - opadajuci redosled. Nalazi se
maksimum u ostatku
       * niza i onda se zamenjuju mesta tekuceg elementa i
maksimuma. Efikasnost
       * O(n^2) zbog broja poredjenja, gledano u odnosu na broj
premestanja je
       * O(n)
       *
       * @param array
       *            niz koji treba da se sortira
       * @return niz sortiran u opadajucem redosledu
       */
      public static int[] selectionSortDesc(int[] array) {
```

```
            if (array.length == 0)
                  return null;
            for (int i = 0; i < array.length; i++) {
                  int maxInd = i;
                  int max = array[i];
                  for (int j = i; j < array.length; j++)
                        if (array[j] < max) {
                              maxInd = j;
                              max = array[j];
                        }
                  if (maxInd != i) {
                        int temp = array[i];
                        array[i] = array[maxInd];
                        array[maxInd] = temp;
                  }
                  // System.out.print("SelectionSort: ");
                  // TestSort.pritArray(array);
            }

            return array;
      }

      /**
       * Bubble Sort - Tezi mehurici padaju dole. Samo ima zvucno
ime, inace je
       * najgori algoritam.
       *
       * U spoljasnjoj petlji se ide od kraj ka pocetku, a onda
se pomocu
       * unutrasnje petlje namestanaju veci na kraj. U
unutrasnjoj petlji se svaki
       * poredi sa susednim elementom i veci se postavlja blize
kraju.
       *
       * Efikasnost O(n^2) prema broju poredjenja, gledano u
odnosu na broj
       * premestanja je isto O(n^2)
       *
       * @param array
       *            niz koji treba da se sortira
       * @return niz sortiran u rastucem redosledu
       */
      public static int[] bubbleSort(int[] array) {
            if (array.length == 0)
                  return null;
            for (int i = array.length - 1; i > 0; i--)
                  for (int j = 0; j < i; j++)
                        if (array[j] > array[j + 1]) {
                              int temp = array[j + 1];
                              array[j + 1] = array[j];
                              array[j] = temp;
                        }
```

```
            return array;
      }

      /**
       * Insertion Sort - Sluzi za skoro sortiran niz. Svaki
element se prvo
       * izbaci iz niza, onda se svi elementi koji su pre njega,
a veci su od
       * njega pomere za po jedno mesto da bi zauzeli njegovo
mesto, a on dolazi
       * na mesto poslednjeg pomerenog. Prakticno se ceo blok
emelenata koji su
       * pre tekuceg i veci su od njega shiftuje u desno za jedno
mesto.
       *
       * Efikasnost O(n^2) ali za skoro sortiran niz radi skoro
O(n)
       *
       * @param array
       *            niz koji treba da se sortira
       * @return niz sortiran u rastucem redosledu
       */
      public static int[] insertionSort(int[] array) {
            if (array.length == 0)
                  return null;
            for (int i = 1; i < array.length; i++) {
                  int j = 1;
                  int tekuci = array[i];
                  while ((i >= j) && (array[i - j] > tekuci)) {
                        array[i - j + 1] = array[i - j];
                        j++;
                  }
                  array[i - j + 1] = tekuci;
                  // System.out.print("InsertionSort: ");
                  // TestSort.pritArray(array);
            }

            return array;
      }

      /**
       *
       * Najefikasniji od O(n^2)
       *
       * Porede se elementi na svakoj k-toj poziciji i veci se
pomeraju blize
       * kraju, a onda se kroz iteracije k-inkrement-smanjuje.
       *
       * @param array
       *            niz koji treba da se sortira
       * @return niz sortiran u rastucem redosledu
```

```
	 */
	public static int[] shellSort(int[] array) {
		int increment = 3;
		int temp;

		while (increment > 0) {
			for (int i = 0; i < array.length; i++) {
				int j = i;
				temp = array[i];
				/*
				 * Gledano s kraja (koji je ogranicen sa i)
sve dok je element
				 * na prethodnoh k-toj poziciji veci od
ovoga na tekucoj pomeri
				 * ga za k unapred. Na kraju prvi koji je
sklonjen stavi na
				 * poziciju poslednjeg pomerenog.
				 */
				while ((j >= increment) && (array[j -
increment] > temp)) {
					array[j] = array[j - increment];
					j = j - increment;
				}
				array[j] = temp;
				// System.out.print("ShellSort: ");
				// TestSort.pritArray(array);
			}
			/*
			 * ovim se smanjuje inkrement
			 */

			/*
			 * ovo je kod koji je zamenjen ovim dole. Trebalo
bi da to dodje na
			 * isto. if (increment/2 != 0) increment =
increment / 2; else if
			 * (increment == 1) increment = 0; else increment
= 1;
			 */
			if (increment == 1)
				increment = 0;
			else
				increment = increment / 2;

		}

		return array;
	}

/**
	 * Ideja ovoga je da se niz smesti u heap i na taj nacin
sortira a da se
```

```
      * posle iz tog heapa samo prebaci u niz. Iako je ideja
vrlo intuitivna,
      * ovde se implementira na jedna vrlo neintuitivan nacin da
bi se sve
      * transformacije vrsile unutar istog niza. Lepsa
implementacija
      * podrazumevala bi pomocni niz za heap. Najsporiji od O(n
log n) algoritama
      *
      * @param array
      * @return niz sortiran u rastucem poretku
      */
     public static int[] heapSort(int[] array) {
          if (array.length == 0)
               return null;
          int temp;

          for (int i = (array.length / 2) - 1; i >= 0; i--)
               // obratiti paznju na index array.length-1 !!!
               shiftDown(array, i, array.length - 1);

          for (int i = array.length - 1; i >= 1; i--) {
               temp = array[0];
               array[0] = array[i];
               array[i] = temp;
               shiftDown(array, 0, i - 1);
          }
          return array;
     }

     private static void shiftDown(int[] array, int root, int
bottom) {

          int maxChild;
          int temp;

          while ((root * 2 <= bottom)) {
               if (root * 2 == bottom)
                    maxChild = root * 2;
               else if (array[root * 2] > array[root * 2 + 1])
                    maxChild = root * 2;
               else
                    maxChild = root * 2 + 1;

               if (array[root] < array[maxChild]) {
                    temp = array[root];
                    array[root] = array[maxChild];
                    array[maxChild] = temp;
                    root = maxChild;
               } else
                    break;
          }
```

```
	}

	/**
	 * Sortiranje se obavlja tako sto se prvi element u svakoj
iteraciji izabere
	 * za pivot. Onda se elementi rasporede tako da manji od
pivota budu pre
	 * njega, a veci budu posle. Zatim se pozove quick sort nad
oba segmenta
	 * niza - pre i posle pivota.
	 *
	 * Najbolje vreme O(n Log2 n)
	 *
	 * Najgore vreme O(n^2)
	 *
	 * @param array
	 * @return niz sortiran u rastucem poretku
	 */
	public static int[] quickSort(int[] array) {
		if (array.length == 0)
			return null;
		quickSort(array, 0, array.length);
		return array;
	}

	private static void quickSort(int[] array, int a, int b) {
		if (b - a <= 1)
			return;
		int pivot = array[a];
		int pivotInd = a;
		int right = b - 1;
		int left = a;

		while (left < right) {

			int temp;
			/*
			 * Smanjuj desni kraj sve dok su elementi veci od
pivota
			 */
			while ((array[right] >= pivot) && (left < right))
				right--;

			while ((array[left] <= pivot) && (left < right))
				/*
				 * Povecavaj levi kraj sve dok su elementi
manji od pivota
				 */
				left++;
			/*
			 * Sada je left na poziciji prvog elementa sleva
koji je veci od
```

```
                 * pivota, a right na poziciji prvog sdesna koji
je manji od pivota.
                 * Zato ova dva zamene mesta. Tako sve dok se dva
kraja ne sastave.
                 */
                temp = array[left];
                array[left] = array[right];
                array[right] = temp;
                if (left == pivotInd)
                     pivotInd = right;
           }
           /*
            * Kad su se dva kraja sastavila pivot se stavlja u
sredinu. Na taj
            * nacin dobili smo niz u kome su svi elementi pre
pivota manji od
            * njega, a svi posle veci.
            */
           array[pivotInd] = array[left];
           array[left] = pivot;
           pivotInd = left;

           // System.out.print("QuickSort: ");
           // TestSort.pritArray(array);

           quickSort(array, a, pivotInd);
           quickSort(array, pivotInd + 1, b);

     }

     /**
      * Uzima se niz i parcijalno se sortira - svaka polovina za
sebe, a onda se
      * te dve polovine spajaju. Jedva brži od Heap sort
algoritma na velikom
      * skupu podataka
      *
      * @param array
      * @return niz sotriran u rastucem poretku
      * @see mergeSort(int[],int,int)
      * @see merge(int[],int,int,int)
      */
     public static int[] mergeSort(int[] array) {
          if (array.length == 0)
               return null;
          mergeSort(array, 0, array.length - 1);
          return array;
     }

     private static void mergeSort(int[] array, int a, int b) {
          if (b - a < 1)
               return;
```

```
            int middle = (a + b) / 2;
            /*
             * Sortira se niz parcijalno i onda se spaja
             */
            mergeSort(array, a, middle);
            mergeSort(array, middle + 1, b);

            merge(array, a, middle + 1, b);
      }

      private static void merge(int[] array, int a, int middle,
int b) {
            /*
             * Spajanje se obavlja tako sto se napravi pomocni niz
i u njega se
             * smestaju elementi. Pretpostavlja se da su elementi
u svakoj polovini
             * niza vec sortirani. Uzema se prvi element iz prve
polovine niza i
             * poredi se sa prvim elementom druge polovine niza.
Sve dok se u drugoj
             * polovini nalaze manji elementi oni se stavljaju u
pomocni niz. Kad se
             * u drugoj polovini naidje na element veci od ovog iz
prve polovine, u
             * pomocni niz se dodaje taj elemnt iz prve polovine i
sve tako dok se
             * svi ne poredjaju u pomocni niz.
             *
             * Onda se pomocni niz prebacuje u originalni, na
odgovarajuca mesta.
             */
            int[] temp = new int[b - a + 1];
            int count = 0;
            int j = middle;
            for (int i = a; i < middle; i++) {
                  int candidate = array[i];
                  /*
                   * Ako se na ovom mestu stavi array[j]>candidate
dobija se
                   * sortiranje u opadajucem redosledu
                   */
                  while ((j <= b) && (array[j] < candidate))
                        temp[count++] = array[j++];
                  temp[count++] = candidate;
            }
            if (count == 0)
                  return;

            for (int i = 0; i < count; i++)
                  array[a + i] = temp[i];
```

```
        // System.out.print("MergeSort: ");
        // TestSort.pritArray(array);

    }
```

Pretraživanje

```
	/**
	 * Samo poziva metodu binaryRecursionAscending(int[] array,
int element,int
	 * a, int b)
	 *
	 * @param array
	 * @param element
	 * @return indeks elementa u nizu ili -1 ako nije nadjen
	 * @see #binaryRecursionAscending(int[],int,int,int)
	 *
	 */
	public static int binarySearchRecAsc(int[] array, int
element) {

		if (array.length == 0)
			return -1;

		return binaryRecursionAscending(array, element, 0,
array.length - 1);

	}

	/**
	 * Rekurzivan binary search. Uzima element iz sredine niza
i poredi ga sa
	 * trazenim. Ako je element u sredini veci od trazenog
pretrazuje dalje u
	 * prvoj polovini niza, inace u drugoj. Pretpostavlja se da
je niz sortiran
	 * u rastucem redosledu.
	 *
	 * Efikasnost log2N
	 *
	 * @param array
	 *            niz sortiran u rastucem redosledu
	 * @param element
	 *            element koji se trazi
	 * @param a
	 *            pocetak niza
	 * @param b
	 *            kraj niza
	 * @return indeks elementa u nizu ili -1 ako nije nadjen
	 *
	 */
	private static int binaryRecursionAscending(int[] array,
int element,
			int a, int b) {

		int middle = (a + b) / 2;
```

```
            /*
             * provera da li je izasao iz opsega
             */
            if (element < array[a] || element > array[b])
                  return -1;

            if (array[middle] == element)
                  return middle;

            if (array[middle] < element)
                  return (binaryRecursionAscending(array, element,
middle + 1, b));
            else
                  return (binaryRecursionAscending(array, element,
a, middle - 1));

      }

      /**
       * Samo poziva metodu binaryRecursionDescending(int[]
array, int element,int
       * a, int b)
       *
       * @param array
       * @param element
       * @return indeks elementa u nizu ili -1 ako nije nadjen
       * @see #binaryRecursionDescending(int[],int,int,int)
       */
      public static int binarySearchRecDesc(int[] array, int
element) {
            if (array.length == 0)
                  return -1;
            return binaryRecursionDescending(array, element, 0,
array.length - 1);
      }

      /**
       * Rekurzivan binary search. Uzima element iz sredine niza
i poredi ga sa
       * trazenim. Ako je element u sredini veci od trazenog
pretrazuje dalje u
       * drugoj polovini niza, inace u prvoj. Pretpostavlja se da
je niz sortiran
       * u opadajucem redosledu.
       *
       * @param array
       *            niz sortiran u opadajucem redosledu
       * @param element
       *            element koji se trazi
       * @param a
       *            pocetak niza
```

```
       * @param b
       *            kraj niza
       * @return indeks elementa u nizu ili -1 ako nije nadjen
       */

      private static int binaryRecursionDescending(int[] array,
int element,
                  int a, int b) {

            int middle = (a + b) / 2;

            /*
             * provera da li je izasao iz opsega
             */
            if (element > array[a] || element < array[b])
                  return -1;

            if (array[middle] == element)
                  return middle;

            if (array[middle] > element)
                  return (binaryRecursionAscending(array, element,
b, middle - 1));
            else
                  return (binaryRecursionAscending(array, element,
middle + 1, a));

      }

      /**
       * Algoritam za binarny search bez rekurzije, za niz
sortiran u rastucem
       * poretku.
       *
       * @param array
       * @param element
       * @return indeks elementa u nizu ili -1 ako nije nadjen
       */
      public static int binarySearchIterativeAsc(int[] array, int
element) {
            if (array.length == 0)
                  return -1;
            int a = 0;
            int b = array.length - 1;
            int middle;

            do {
                  /*
                   * Ako je element izvan opsega vrati -1 kao znak
da nije nadjen
                   */
                  if ((element < array[a]) | (element > array[b]))
```

```
                    return -1;
               middle = (a + b) / 2;
               if (array[middle] == element)
                    return middle;
               if (element > array[middle])
                    a = middle + 1;
               else
                    b = middle - 1;
          } while (a != b);
          if (array[a] == element)
               return a;
          return -1;
     }

     /**
      * Algoritam za binarny search bez rekurzije, za niz
sortiran u opadajucem
      * poretku.
      *
      * @param array
      * @param element
      * @return indeks elementa u nizu ili -1 ako nije nadjen
      */
     public static int binarySearchIterativeDesc(int[] array,
int element) {
          if (array.length == 0)
               return -1;
          int a = 0;
          int b = array.length - 1;
          int middle;

          do {
               /*
                * Ako je element izvan opsega vrati -1 kao znak
da nije nadjen
                */
               if ((element > array[a]) | (element < array[b]))
                    return -1;
               middle = (a + b) / 2;
               if (array[middle] == element)
                    return middle;
               if (element < array[middle])
                    a = middle + 1;
               else
                    b = middle - 1;
          } while (a != b);
          if (array[a] == element)
               return a;
          return -1;
     }

     /**
```

```
       * Samo poziva metodu interpolationSearchRecursiveAsc(int[]
array,int
       * element, int min, int max)
       *
       * @param array
       * @param element
       * @return indeks elementa u nizu ili -1 ako nije nadjen
       * @see #interpolationSearchRecursiveAsc(int[],int,int,int)
       */
      public static int interpolationSearchRecursiveAsc(int[]
array, int element) {
            if (array.length == 0)
                  return -1;
            return interpolationSearchRecursiveAsc(array, element,
0,
                        array.length - 1);
      }

      /**
       * Pretrazivanje niza sortiranog u rastucem poretku.
Sledeci element za
       * poredjenje bira se po formuli <b>index = min + (max -
min) * (element -
       * array[min])/(array[max] - array[min]);</b>
       *
       * Ako su kljucevi uniformno rasporedjeni zahteva vreme
log2(log2N)
       *
       * @param array
       *            niz sortiran u rastucem redosledu
       * @param element
       *            element koji se trazi
       * @param min
       *            minimalni element (na pocetku niza)
       * @param max
       *            maksimalni element (na kraju niza)
       * @return indeks elementa u nizu ili -1 ako nije nadjen
       *
       */
      private static int interpolationSearchRecursiveAsc(int[]
array,
                  int element, int min, int max) {
            if ((element < array[min]) || (element > array[max]))
                  return -1;
            /*
             * racuna se indeks sledeceg elementa koji ce se
pretraziti, i pri tome
             * se vodi racuna da imenilac ne bude nula
             */
            int index = min + (max - min) * (element - array[min])
                        / Math.max(1, array[max] - array[min]);
            /*
```

```
            * ako je to taj koji se trazi, vrati index
            */
           if (array[index] == element)
                return index;
           /*
            * ako je element na poziciji indeksa manji od ovog
koji se trazi,
            * pretrazi segment niza od indeksa do kraja
            */
           if (array[index] < element)
                return interpolationSearchRecursiveAsc(array,
element, index + 1,
                           max);
           else
                /*
                 * ako je element na poziciji indeksa veci od
ovog koji se trazi,
                 * pretrazi segment niza od pocetka do indeksa
                 */
                return interpolationSearchRecursiveAsc(array,
element, min,
                           index - 1);

     }

     /**
      * Samo poziva metodu
interpolationSearchRecursiveDesc(int[] array,int
      * element, int max, int min)
      *
      * @param array
      * @param element
      * @return indeks elementa u nizu ili -1 ako nije nadjen
      * @see #interpolationSearchRecursiveDesc(int[],int,int,int)
      */
     public static int interpolationSearchRecursiveDesc(int[]
array, int element) {
           if (array.length == 0)
                return -1;
           return interpolationSearchRecursiveDesc(array,
element, 0,
                     array.length - 1);
     }

     /**
      * Pretrazivanje niza sortiranog u rastucem poretku.
Sledeci element za
      * poredjenje bira se po formuli <b>index = max + (min -
max) * (array[max] -
      * element)/ (array[max] - array[min]);</b>
      *
      * @param array
```

```
       *            niz sortiran u opadajucem redosledu
       * @param element
       *            element koji se trazi
       * @param min
       *            minimalni element (na kraju niza)
       * @param max
       *            maksimalni element (na pocetku niza)
       * @return indeks elementa u nizu ili -1 ako nije nadjen
       *
       */
      private static int interpolationSearchRecursiveDesc(int[]
array,
                  int element, int max, int min) {
            if ((element < array[min]) | (element > array[max]))
                  return -1;
            /*
             * racuna se indeks sledeceg elementa koji ce se
pretraziti, i pri tome
             * se vodi racuna da imenilac ne bude nula
             */
            int index = max + (min - max) * (array[max] - element)
                        / Math.max(1, array[max] - array[min]);
            /*
             * ako je to taj koji se trazi, vrati index
             */
            if (array[index] == element)
                  return index;
            /*
             * ako je element na poziciji indeksa veci od ovog
koji se trazi,
             * pretrazi segment niza od indeksa do kraja
             */
            if (array[index] > element)
                  return interpolationSearchRecursiveDesc(array,
element, index + 1,
                              min);

            else
                  /*
                   * ako je element na poziciji indeksa manji od
ovog koji se trazi,
                   * pretrazi segment niza od pocetka do indeksa
                   */
                  return interpolationSearchRecursiveDesc(array,
element, max,
                              index - 1);
      }

      /**
       * Iterativni algoritam za interpolaciono pretrazivanje
niza sortiranog u
       * rastucem redosledu.
```

```
       *
       * @param array
       *            sortiran u rastucem redosledu
       * @param element
       *            koji se trazi
       * @return index elementa u nizu ili -1 ako nije nadjen
       */
      public static int interpolationSearchIterativeAsc(int[]
array, int element) {
            int a = 0;
            int b = array.length - 1;
            int index;
            do {
                  /*
                   * ovo sluzi da se rano otkrije da element nije u
nizu, ako je izvan
                   * opsega
                   */
                  if (element < array[a] || element > array[b])
                        return -1;

                  index = a + (element - array[a]) * (b - a)
                              / Math.max(1, array[b] - array[a]);
                  if (array[index] == element)
                        return index;
                  if (array[index] > element)
                        /*
                         * ove provere sa min i max sluze da ne ode
slucajno index u
                         * minus i da ne premasi duzinu niza, jer
ne znamo tacno u kom
                         * opsegu se krece vrednost one formule za
index
                         *
                         */
                        b = Math.max(a, index - 1);
                  else
                        a = Math.min(index + 1, b);

            } while (a != b);
            // ako smo do ovde stigli, a=b
            if (array[a] == element)
                  return a;
            return -1;
      }

}
```

Stabla

```
public class Node {
	public Node left;

	public Node right;

	public int data;

	public Node(Node left, Node right, int data) {
		super();
		this.left = left;
		this.right = right;
		this.data = data;
	}

	public Node(int data) {
		super();
		this.data = data;
	}

	@Override
	public String toString() {
		// TODO Auto-generated method stub
		return "" + data;
	}

}

	/**
	 * 1) Napisati funkciju koja prihvata pokazivaè na binarno
stablo koje
	 * predstavlja neki izraz (listovi su operandi, a
unutrašnji èvorovi su
	 * binarni operatori) i štampa prefiks predstavu izraza
	 */
	public static void prefix(Node n) {
		if (n == null)
			return;
		/*
		 * Kad se ovako odstampaju operatori i operandi,
dobiju se operacije u
		 * lispu, prefiksna notacija e.g. (+ 5 6)
		 */
		System.out.println("( ");

		System.out.println(" " + n.data + " ");
		prefix(n.left);
		prefix(n.right);

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

```
	/**
	 * 2) Napisati funkciju koja prihvata pokazivaè na binarno
stablo koje
	 * predstavlja neki izraz (listovi su operandi, a
unutrašnji èvorovi su
	 * binarni operatori) i štampa infiks predstavu izraza
	 */
	public static void infix(Node n) {
		if (n == null)
			return;
		System.out.println("( ");
		infix(n.left);
		System.out.println(" " + n.data + " ");
		infix(n.right);
		System.out.println(" )");
	}

	/**
	 * 3) Napisati funkciju koja prihvata pokazivaè na koren
nekog binarnog
	 * stabla koje sadrži cele brojeve i štampa sadržaj èvorova
stabla koji su
	 * negativni u postfiks redosledu.
	 */
	public static void postfix(Node n) {
		if (n == null)
			return;
		postfix(n.left);
		postfix(n.right);
		if (n.data < 0)
			System.out.println(" " + n.data + " ");

	}

	/**
	 * 4) Dat je pokazivaè na koren binarnog stabla èiji
èvorovi sadrže cele
	 * brojeve. Napisati funkciju koja æe vratiti pokazivaè na
èvor list koji je
	 * na najveæoj dubini u stablu.
	 */
	public static Node deepestNode(Node root) {
		if (root == null)
			return null;
		/*
		 * ako je to list, znaci nasli smo ga
		 */
		if (root.left == null && root.right == null)
			return root;
		/*
		 * ako nije to list, idi ka dubljem listu
```

```
             */
            if (height(root.left) > height(root.right))
                  return deepestNode(root.left);
            else
                  return deepestNode(root.right);

      }

      private static int height(Node root) {
            if (root == null)
                  return 0;
            return 1 + Math.max(height(root.right),
height(root.left));
      }

      /**
       * 5) Dat je pokazivaè na koren binarnog stabla èiji
èvorovi sadrže cele
       * brojeve. Napisati funkciju koja æe vratiti pokazivaè na
èvor list koji je
       * na najmanjoj dubini u stablu.
       *
       */
      public static Node shallowestNode(Node root) {
            if (root == null)
                  return null;
            /*
             * ako je to list, znaci nasli smo ga
             */
            if (root.left == null && root.right == null)
                  return root;
            /*
             * ako nije to list, idi ka plicem listu
             */
            if (firstLeafLevel(root.left) <
firstLeafLevel(root.right))
                  return shallowestNode(root.left);
            else
                  return shallowestNode(root.right);

      }

      private static int firstLeafLevel(Node root) {
            if (root == null)
                  /*
                   * na ovaj nacin ignorisu se cvorovi kojima je
samo jedno dete null
                   */
                  return Integer.MAX_VALUE;
            /*
             * ako je to list, nasli smo ga, vrati 0
             */
```

```
            if (root.left == null && root.right == null)
                  return 0;
            /*
             * ako nije to list, trazi dalje
             */
            return 1 + Math.min(firstLeafLevel(root.right),
                        firstLeafLevel(root.left));
      }

      /**
       * 6) Dva binarna stabla su identièna ako su oba prazna ili
ako oba imaju
       * koren èiji sadržaj je jednak a njihova leva i desna
podstabla su
       * identièna. Napišite funkciju koja æe proveriti da li su
dva binarna
       * stabla identièna
       */
      public static boolean areEqual(Node a, Node b) {
            if (a == null && b == null)
                  return true;
            // ^ == XOR
            if (a == null ^ b == null)
                  /*
                   * ako je samo jedan null return false
                   */
                  return false;
            /*
             * sada znamo da su oba !=null
             */
            return a.data == b.data && areEqual(a.left, b.left)
                        && areEqual(a.right, b.right);
      }

      /**
       * 18) Dva stabla su «slièna kao u ogledalu» ako su oba
prazna ili ako nisu
       * prazna, ako je levo stablo svakog stabla «slièno kao u
ogledalu» desnom
       * stablu onog drugog.
       *
       * @param a
       *            koren jednog stabla
       * @param b
       *            koren drugog stabla
       * @return true ako je a «slièno kao u ogledalu» sa b
       */
      public static boolean mirrorAlike(Node a, Node b) {
            if (a == null && b == null)
                  return true;
            // ^ == XOR
            if (a == null ^ b == null)
```

```
                /*
                 * ako je samo jedan null return false
                 */
                return false;
            /*
             * sada znamo da su oba !=null
             */
            return (mirrorAlike(a.right, b.left) &&
mirrorAlike(a.left, b.right));

      }

      /**
       * 8) Napišite funkciju int roditelj(cvor *k, cvor * p)
koja prihvata
       * pokazivaè na koren binarnog stabla i pokazivaè na neki
èvor u stablu, a
       * vraæa pokazivaè na roditelja èvora p (vraæa NULL ako
roditelj ne
       * postoji). Pri tome èvor stabla ima samo pokazivaèe na
svoju decu
       *
       * U postavci je greška treba Node a ne int
       *
       * @see #areEqual (Node a ,Node b) iz zadatka 6.
       *
       */
      public static Node parent(Node root, Node p) {
            if (root == null)
                  return null;
            if (root.left != null) {
                  if (areEqual(root.left, p))
                        return root;
                  Node n = parent(root.left, p);
                  if (n != null)
                        return n;
            }
            if (root.right != null) {
                  if (areEqual(root.right, p))
                        return root;
                  Node n = parent(root.right, p);
                  if (n != null)
                        return n;
            }
            return null;
      }

      /**
       * Na drugi nacin
       *
       * 8) Napišite funkciju int roditelj(cvor *k, cvor * p)
koja prihvata
```

```
	 * pokazivaè na koren binarnog stabla i pokazivaè na neki
èvor u stablu, a
	 * vraæa pokazivaè na roditelja èvora p (vraæa NULL ako
roditelj ne
	 * postoji). Pri tome èvor stabla ima samo pokazivaèe na
svoju decu
	 *
	 * U postavci je greška treba Node a ne int
	 *
	 */
	public static Node parent2(Node root, Node p) {
		if (root == null || root == p)
			return null;
		if (root.left != null && root.left == p)
			return root;
		if (root.right != null && root.right == p)
			return root;
		Node parent = parent2(root.left, p);
		if (parent == null)
			parent = parent2(root.right, p);
		return parent;
	}

	/**
	 * 13) Dat je pokazivaè na koren binarnog stabla èiji
èvorovi sadrže cele
	 * brojeve. Napišite funkciju koja æe vratiti broj èvorova
koji su po
	 * sadržaju veæi od sadržaja svih svojih potomoka.
	 */

	public static int howMany(Node root) {
		if (root == null)
			return 0;
		if (root.left == null && root.right == null)
			return 0;// ignorisi listove

		int increment = 0;
		if (root.data > max(root.left) && root.data >
max(root.right))
			increment = 1;

		return increment + howMany(root.left) +
howMany(root.right);

	}

	private static int max(Node root) {
		if (root == null)
			return Integer.MIN_VALUE;
		return Math.max(root.data, Math.max(max(root.left),
max(root.right)));
```

```
	}

	/**
	 * 17) Napišite funkciju int nivo(cvor *k, cvor * p) koja
prihvata pokazivaè
	 * na koren binarnog stabla i pokazivaè na neki èvor u
stablu i vraæa nivo
	 * na kome se pokazani èvor nalazi.
	 *
	 */

	public static int level(Node root, Node n, int level) {
		if (root == null)
			return -1;
		if (root == n)
			/*
			 * Moze i root.equals(n) i root.data==n.data
zavisi sta se trazi u
			 * zadatku. Posto nista konkretno ne kaze, moze
po izboru.
			 */
			return level;
		int temp = level(root.left, n, level + 1);
		if (temp != -1)
			return temp;
		return level(root.right, n, level + 1);

	}

	/**
	 * 19. Napišite funkciju cvor * nivo(cvor *k, cvor *p, cvor
*q) koja
	 * prihvata pokazivaè k na koren i pokazivaèe p i q na na
neke èvorove u
	 * binarnom stablu i vraæa pokazivaè na onaj èvor koji se
nalazi dublje u
	 * stablu (tj. èiji je nivo na kome se nalazi veæi).
	 *
	 * p.s. Ako neko zna cemu im sluze ove zvezdice u postavci
zadatka, molim da
	 * me obavesti preko gore navedenog e-maila. To mi je
enigma.
	 */
	public static Node deeperNode(Node root, Node a, Node b) {
		int levelA = level(root, a, 0);
		int levelB = level(root, b, 0);

		if (levelA > levelB)
			return a;
		else
			return b;
```

```
	}

	/**
	 * 20) Napisati rekurzivni algoritam za pretraživanje
binarnog stabla koje
	 * je uređeno (BST stablo).
	 */
	public static Node searchBST(Node root, int data) {
		if (root == null)
			return null;
		if (root.data == data)
			return root;
		if (root.data < data)
			return searchBST(root.right, data);
		return searchBST(root.left, data);

	}

	/**
	 * Ova metoda stampa sve cvorove na putu od korena BINARNOG
STABLA do
	 * zadatog cvora. Dato stablo nije ni BST, ni AVL. Ovo je
DepthFirstSearch
	 * <a
	 * href="http://en.wikipedia.org/wiki/Depth-
first_search">http://en.wikipedia.org/wiki/Depth-first_search</a>
	 * Varijanta koja stampa cvorove od zadatog cvora do korena
	 *
	 * Ovaj zadatak nije bio zadat na ispitu, vec je dat zbog
implemntacije
	 * teorijski znacajnog algoritma
	 */
	public static boolean printFromNodeToRoot(Node root, Node
node) {
		// nigde se ne kaze u kom redosledu da stampa pa moze
i od kraja ka
		// root-u
		if (root == null)
			return false;
		if (root.equals(node)) {
			System.out.println("-> " + root.data);
			return true;
		}
		if (printFromNodeToRoot(root.left, node)) {
			System.out.println(root.data);
			return true;
		}
		if (printFromNodeToRoot(root.right, node)) {
			System.out.println(root.data);
			return true;
		}
		return false;
```

```
	}

	/**
	 * Ova metoda stampa sve cvorove na putu od korena BINARNOG
STABLA do
	 * zadatog cvora. Dato stablo nije ni BST, ni AVL. Ovo je
DepthFirstSearch
	 * <a
	 * href="http://en.wikipedia.org/wiki/Depth-
first_search">http://en.wikipedia.org/wiki/Depth-first_search</a>
	 * Varijanta koja stampa cvorove od korena do zadatog
cvora. Za to sluzi
	 * Stack u kome se cvorovi cuvaju, pa se na kraju stampaju.
	 *
	 * Ovaj zadatak nije bio zadat na ispitu, vec je dat zbog
implemntacije
	 * teorijski znacajnog algoritma
	 */
	public static void printFromRootToNode(Node root, Node
node) {
		stack = new Stack<Node>();
		boolean found = printFromRoot2Node(root, node);
		if (!found) {
			System.out.println("not found");
			return;
		}
		System.out.println(stack);

	}

	private static java.util.Stack<Node> stack;

	private static boolean printFromRoot2Node(Node root, Node
node) {
		// ako im je bitan redosled stavi sve u Stack pa onda
odstapmaj
		if (root == null)
			return false;
		// stavi tekuci cvor na stek
		stack.add(root);
		if (root.equals(node)) {
			return true;
		}

		if (printFromRoot2Node(root.left, node)) {
			return true;
		} else if (printFromRoot2Node(root.right, node)) {
			return true;
		}
		/*
		 * posto nema ni levo ni desno nista pametno, sklanjaj
ovaj cvor iz
```

```
             * steka
             */
            stack.remove(root);
            return false;
      }

      /**
       * Jun 2006 zadatak 5) Napisati proceduru koja štampa
sadržaj svih èvorova
       * binarnog stabla (nije BST) na putanji od korena do
najdubljeg èvora.
       *
       * @see #height(Node) iz 4. zadatka
       */

      public static void printToDeepest(Node root) {
            if (root == null)
                  return;
            System.out.println(root.data);
            if (height(root.left) > height(root.right))
                  printToDeepest(root.left);
            else
                  printToDeepest(root.right);
      }

      /**
       * Jun 2006 zadatak 7) Napisati proceduru koja štampa
sadržaj svih èvorova
       * binarsnog stabla (nije BST) na putanji od korena do
èvora koji ima
       * najmanju vrednost u stablu.
       */
      public static void printToMin(Node root) {
            if (root == null)
                  return;
            System.out.println(root.data);
            if (min(root) == root.data)
                  return;
            if (min(root.left) < min(root.right))
                  printToMin(root.left);
            else
                  printToMin(root.right);
      }

      private static int min(Node root) {
            if (root == null)
                  return Integer.MAX_VALUE;
            return Math.min(root.data, Math.min(min(root.left),
min(root.right)));
      }

      /**
```

```
	 * Jun 2006 zadatak 5) Napisati proceduru koja štampa
sadržaj svih èvorova
	 * binarsnog stabla (nije BST) na putanji od korena do
èvora koji ima
	 * najvecu vrednost u stablu.
	 *
	 * @see #max(Node) iz 13. zadatka
	 */

	public static void printToMax(Node root) {
		if (root == null)
			return;
		System.out.println(root.data);
		if (max(root) == root.data)
			return;
		if (max(root.left) > max(root.right))
			printToMax(root.left);
		else
			printToMax(root.right);
	}

	/**
	 * Oktobar 2003 I grupa 5. zadatak
	 *
	 * Dat je pokazivac ma koren binarnog stabla ciji cvorovi
sadrze cele
	 * brojeve. Napisati f-ju koja ce vratiti pokazivac na cvor
u stablu kod
	 * koga je najmanji (najveci) proizvod sadrzaja cvorova iz
njegovog desnog
	 * podstabla. --
	 */
	public static Node minRightProduct(Node root) {
		min = null;
		minValue = Integer.MAX_VALUE;
		findMinRightProduct(root);
		return min;
	}

	/*
	 * ako zanemarimo besmisao upotrebe private static void
metode, dobijemo
	 * sledecu metodu
	 */
	private static void findMinRightProduct(Node root) {
		if (root == null)
			return;
		if (root.right == null) {

			/*
			 * cvor nema desno podstablo pa ga ne uzimamo u
obzir i pretrazujemo
```

```
                 * dalje u njegovom levom podstablu
                 */
                findMinRightProduct(root.left);
                return;

          }

          int temp;
          if ((temp = product(root.right)) < minValue) {
                min = root;
                minValue = temp;
          }
          findMinRightProduct(root.right);
          findMinRightProduct(root.left);
     }

     private static Node min;

     private static int minValue;

     private static int product(Node root) {
          if (root == null)
                return 1;
          return root.data * product(root.right) *
product(root.left);

     }

}

/**
      * Napisati funkciju koja prihvata pokazivaè na koren AVL
binarnog stabla i
      * štampa sadržaj èvorova stabla u rastuæem redosledu.
      *
      * ova imena asterix i obelix su samo da ne bi stalno bilo
infix i slicno
      */
     public static void asterix(Node root) {
          // važi za svako BST, ne samo za AVL
          if (root == null)
                return;
          asterix(root.left);
          System.out.println(root.data);
          asterix(root.right);

     }

     /**
      * Napisati funkciju koja prihvata pokazivaè na koren AVL
binarnog stabla i
      * štampa sadržaj èvorova stabla u opadajuæem redosledu.
```

```
	 */
	public static void obelix(Node root) {
		// važi za svako BST
		if (root == null)
			return;
		obelix(root.right);
		System.out.println(root.data);
		obelix(root.left);

	}

	/**
	 * 3) Dat je pokazivaè na koren binarnog stabla. Napišite
funkciju koja æe
	 * vratiti broj èvorova koji ispunjavaju uslove za AVL
stablo.
	 */

	public static int howManyAVL(Node root) {
		if (root == null)
			return 0;

		int balanceFactor = height(root.left) -
height(root.right);

		if (isBSTNode(root) && balanceFactor >= -1 &&
balanceFactor <= 1)
			return 1 + howManyAVL(root.left) +
howManyAVL(root.right);
		else
			return howManyAVL(root.left) +
howManyAVL(root.right);

	}

	private static int height(Node root) {
		if (root == null)
			return 0;
		return 1 + Math.max(height(root.left),
height(root.right));
	}

	private static boolean isBSTNode(Node root) {
		/*
		 * provera da li je BST; Ovo se najverovatnije ne
trazi, ali iz teksta
		 * zadatka sledi da i to treba proveriti (Treba uzeti
u obzir cinjenicu
		 * da autori teksta zadatka bas i nisu knjizevno
potkovani)
		 */
```

```
            boolean isBST = false;

            if (root.left != null) {
                  if (root.right != null) {
                        // oba !=null
                        if (root.data > root.left.data && root.data
< root.right.data)
                              isBST = true;

                  } else {
                        // left!=nill right=null
                        if (root.data > root.left.data)
                              isBST = true;
                  }
            } else {
                  // left=null
                  if (root.right != null && root.data <
root.right.data)
                        isBST = true;
                  else {
                        // oba su null
                        isBST = true;
                  }
            }

            return isBST;

      }

      /**
       * 4) Dat je pokazivaè na koren binarnog stabla. Napišite
funkciju koja æe
       * vratiti broj èvorova koji ne ispunjavaju uslove za AVL
stablo.
       */
      public static int howManyNonAVL(Node root) {
            if (root == null)
                  return 0;

            int balanceFactor = height(root.left) -
height(root.right);

            if (!(isBSTNode(root) && balanceFactor >= -1 &&
balanceFactor <= 1))
                  return 1 + howManyNonAVL(root.left) +
howManyNonAVL(root.right);
            else
                  return howManyNonAVL(root.left) +
howManyNonAVL(root.right);

      }
```

```
      /**
       * 10) Dat je pokazivaè na koren AVL stabla èiji èvorovi
sadrže cele brojeve
       * i drugi pokazivaè na neki èvor u stablu. Napisati
funkciju koja æe
       * odštampati sve èvorove koji su na putanji od korena do
datog èvora.
       *
       * AVL je ujedno i BST, pa se ta svojstva koriste
       */
      public static void printFromRootToNode(Node root, Node
node) {
            if (root == null) {
                  System.out.println("Not found");
                  return;
            }
            if (root.data == node.data) {
                  System.out.println("found: " + root.data);
                  return;
            } else
                  System.out.println(root.data);
            if (root.data > node.data)
                  printFromRootToNode(root.left, node);
            else
                  printFromRootToNode(root.right, node);
      }
```