Tietorakenteet ja algoritmit II

Loppuraportti, ryhmä 7

Turun Yliopisto
Informaatioteknologian laitos
Kevät 2008
Jonne Pohjankukka, jjepoh@utu.fi, 73116
Simo Savonen, sipesa@utu.fi, 56572
Jyri Lehtonen, jkoleh@utu.fi, 72039

Sisällysluettelo

1. Suunnitelma

1.1. Tehtävän kuvaus ja analysointi

Ryhmälle 7 annettiin tehtäväksi verrata AVL-puun tehokkuutta tavalliseen binääripuuhun. Puut tuli toteuttaa Java-kielellä mahdollisimman elegantisti. Puiden metodien aikakompleksisuuksien pitäisi olla yhteneviä luentomonisteessa mainittuihin.

Tehtävä tähtää havainnollistamaan mikä tekee AVL-puista parempia kuin binääripuut. Asia on helppo ymmärtää paperilla, haasteeksi nouseekin lähinnä asian toteaminen käytännön ohjelmakoodilla.

Ryhmässä on kolme jäsentä, joten työ on luontevaa jakaa kolmeen osaan, tai luokkaan, joista kukin jäsen ohjelmoi yhden.

1.2. Ratkaisuperiaate

Molempien puiden toteuttavat ohjelmaluokat tulisi sisältää samat julkiset metodit, jotta puiden vertailu olisi mahdollista. Tämä on kätevä toteuttaa yhteisellä julkisella rajapinnalla *Tree*. Luokkien muut metodit (rotaatiot, ym. apumetodit) ovat yksityisiä. Puiden käyttämät solmut sisältävät paljon yhteisiä piirteitä, joten tehdään luokka BinaryNode, jonka luokka AVLNode perii, lisäten siihen tiedon solmun korkeudesta.

Sijoitetaan kukin luokka omaan tiedostoonsa, jotta työnjako helpottuu. Jätetään käyttämättä sisäluokkia. Jos tietoa halutaan kapseloida, määritellään luokat jakamaan yhteinen paketti. Puiden testaamista varten tehdään testiluokka, joka lukee tekstitiedostoa ja lisää puihin alkioita, mittaa operaatioihin käytetyn ajan erittäin tarkasti, sekä tulostaa yhteenvedon puun operaatioiden suoritusnopeuksista.

1.3. Ohjelman osien kuvaaminen

1.3.1. interface Tree

Rajapintaan määritellään molempien puiden julkisten metodien signatuurit:

```
public void add(Comparable object);
public void delete(Comparable object);
public Comparable find(Comparable object);
public Comparable min();
public Comparable max();
```

Jätetään toteuttamatta metodit `size()`, `clear()` rajataksemme harjoitustyötä.

1.3.2. class BinarySearchTree implements Tree

Tasaisesti täyttyneen puun metodien kompleksisuus on yleisesti O(h) = O(lg n),
mutta pahimmassa tapauksessa on O(h) = O(n)

```
private BinaryNode root; // puun juuri

/**
 * Lisää alkion puuhun.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 */
public void add(Comparable object);

/**
 * Poistaa alkion puusta.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 * Seuraajan etsimisen kompleksisuus.
 */
public void delete(Comparable object);

/**
 * Etsii alkiota puusta, palauttaa löydetyn alkion.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 * Sekä iteratiivisen, että rekursiivisen on O(h).
 */
public Comparable find(Comparable object);

/**
 * Palauttaa puun pienimmän alkion.
 * Alkion vertailu compareTo() avulla.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 */
public Comparable min();

/**
 * Palauttaa puun suurimman alkion.
 * Alkion vertailu compareTo() avulla.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 */
public Comparable max();

/**
 * Palauttaa solmun seuraajan.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 * Kuljetaan polkua joko ylös tai alas, polun pituuden
 * yläraja on h.
 */
private BinaryNode successor(BinaryNode);
```

1.3.3. class AVLTree implements Tree

```
private AVLNode root; // puun juuri

/**
 * Lisää alkion puuhun.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 * Korkeuksien päivityksen ja uudelleenjärjestämisen
 * kompleksisuus on O(lg n).
 */
public void add(Comparable object);

/**
 * Poistaa alkion puusta.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 * Korkeuksien päivityksen ja uudelleenjärjestämisen
 * kompleksisuus on O(lg n).
 */
public void delete(Comparable object);

/**
 * Etsii alkiota puusta, palauttaa löydetyn alkion.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 */
public Comparable find(Comparable object);

/**
 * Palauttaa puun pienimmän alkion.
 * Alkion vertailu compareTo() avulla.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 */
public Comparable min();

/**
 * Palauttaa puun suurimman alkion.
 * Alkion vertailu compareTo() avulla.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 */
public Comparable max();

/**
 * Palauttaa solmun seuraajan.
 * Kompleksisuus O(h) = O(lg n), h on puun korkeus.
 */
private AVLNode successor(AVLNode);
```

```
/**
 * Rotaatio vasemmalle.
 * Kompleksisuus Θ(1).
 */
private AVLNode leftRotate(AVLNode);

/**
 * Rotaatio oikealle.
 * Kompleksisuus Θ(1).
 */
private AVLNode rightRotate(AVLNode);

/**
 * Kaksoisrotaatio vasemmalla alipuulla.
 * Kompleksisuus Θ(1).
 */
private AVLNode doubleRotateLeft(AVLNode);

/**
 * Kaksoisrotaatio oikealla alipuulla.
 * Kompleksisuus Θ(1).
 */
private AVLNode doubleRotateRight(AVLNode);
```

1.3.4. class BinaryNode

```
private Comparable key; // solmun sisältämä satelliittidata
private BinaryNode left; // vasen lapsisolmu
private BinaryNode right; // oikea
private BinaryNode parent; // isäsolmu

public Comparable getKey();
public BinaryNode getParent();
public BinaryNode getLeftChild();
public BinaryNode getRightChild();
public void setLeftChild(BinaryNode);
public void setRightChild(BinaryNode);
public void setParent(BinaryNode);
public void setKey(Comparable);
```

1.3.5. class AVLNode

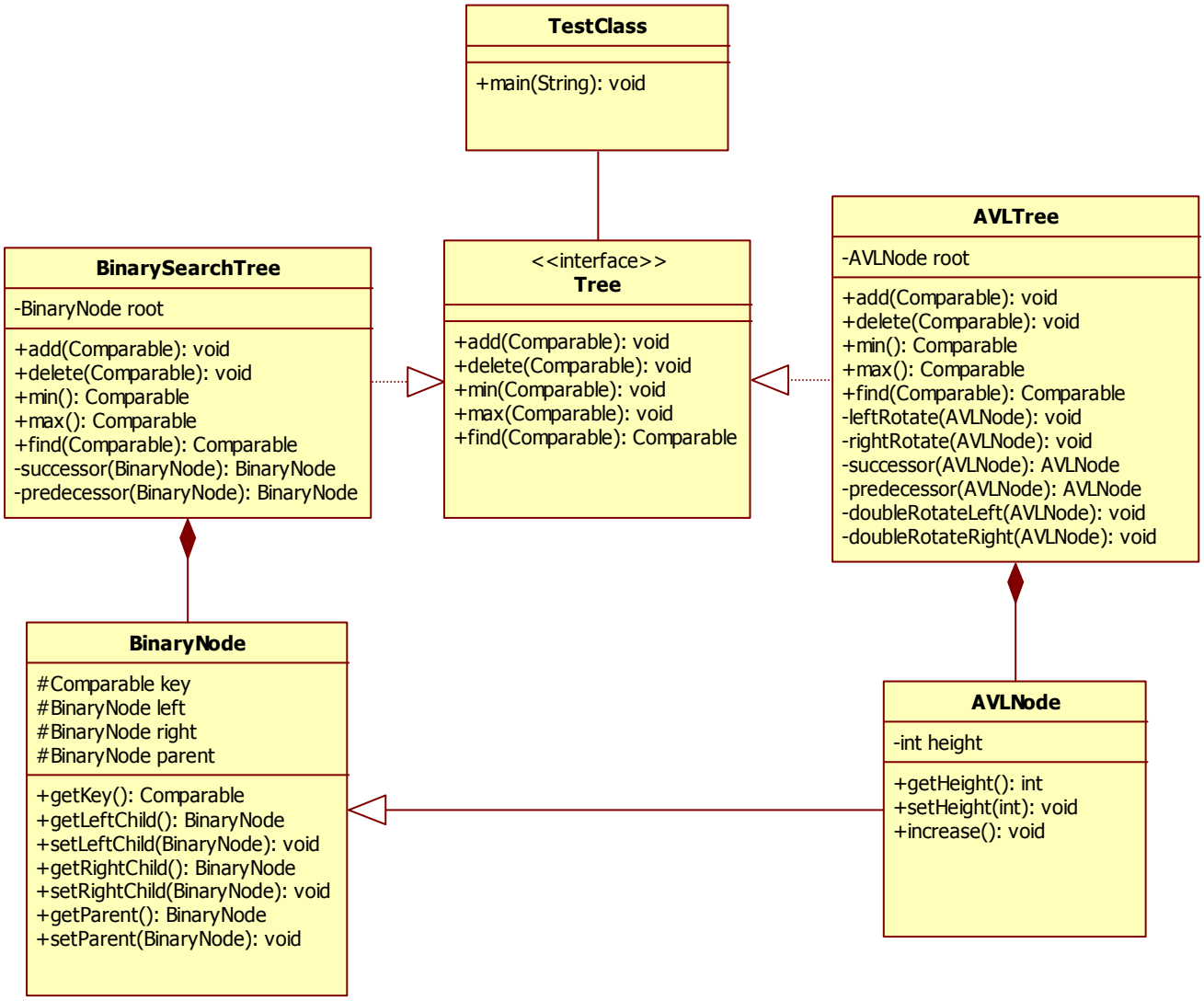
AVLNode-luokassa on samat metodit ja piirteet(AVL-tyyppisinä) kuin BinaryNode-luokallakin, sekä lisäksi seuraavat lisäykset :

```
private int height; // solmun korkeus

public int getHeight();
public void setHeight(int);
```

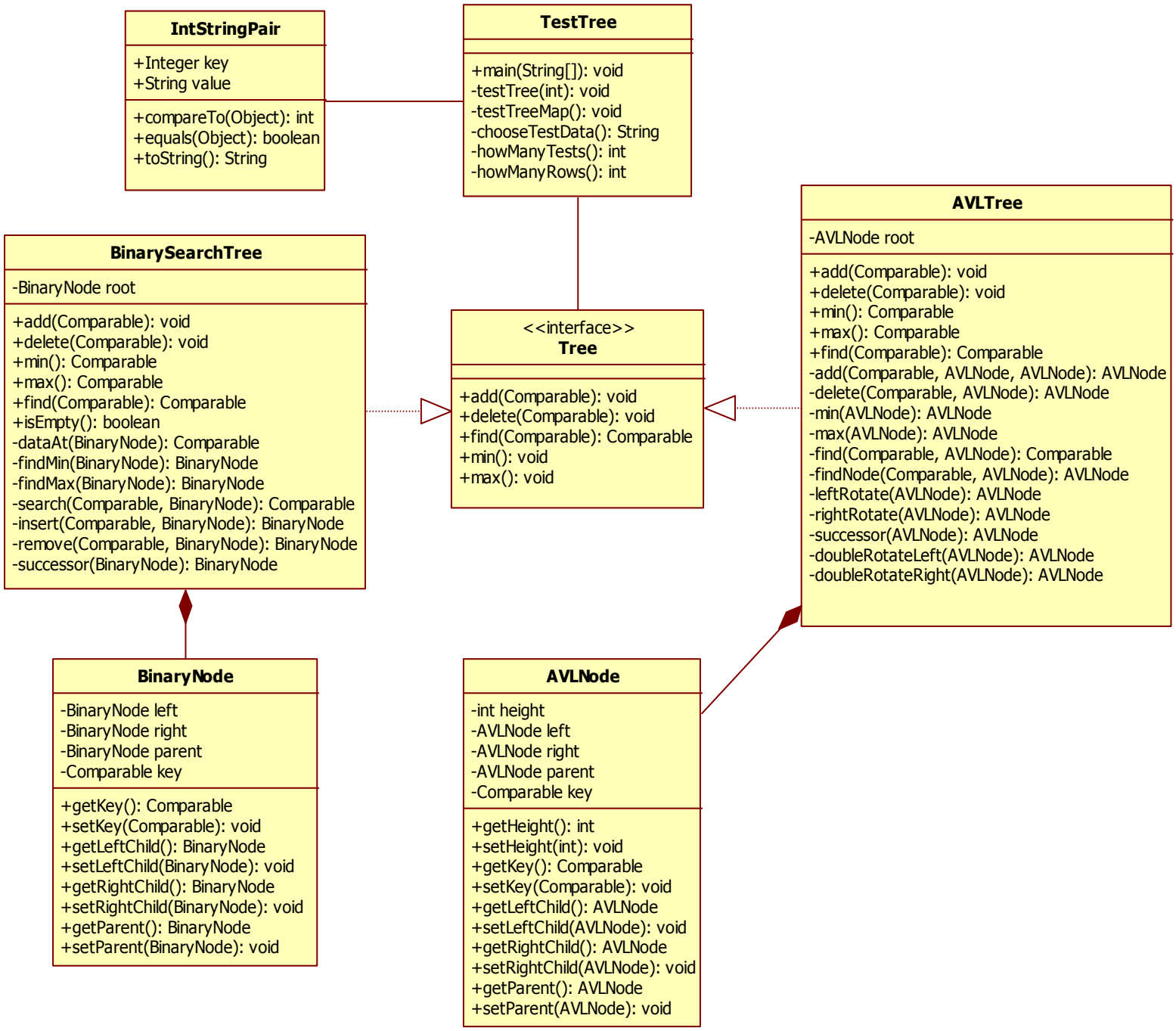
1.3.6. class TestTree

TestTree –luokassa luetaan testiainestoa ja luodaan IntStringPair-olioita, sijoitetaan ne puihin, tehdään hakuja ja poistoja sekä vertaillaan operaatioiden suoritusaikoja.

1.3.7. Alkuperäisen suunnitelman UML luokkakaavio



1.3.8. Toteutetun suunnitelman UML luokkakaavio



1.3.9. Työnjako

Ryhmässä on kolme jäsentä, ja toteutettavia luokkakokonaisuuksia on samoin kolme.

Ohjelmointityöt jaetaan seuraavasti:

BinarySearchTree	Jyri Lehtonen
AVLTree	Jonne Pohjankukka
TestClass	Simo Savonen

Puiden käyttämät solmuluokat ohjelmoitiin yhdessä.

2. Toteutus

Suunnitelmasta poikettiin seuraavasti:

- Koodauksen helpottamiseksi, AVLNode ei peri BinaryNode –luokkaa.
- Esittelytilaisuudessa mainittu ”Tree on yliluokka” -toteutustapa ei lopulta vaikuttanut hyvältä. Olisimme joka tapauksessa joutuneet ylikirjoittamaan metodit perivissä luokissa.
- Puihin lisättävä Comparable –rajapinnan toteuttava olio on IntStringPair –olio, joka sisältää kokonaisluvun ja merkkijonon.
- TestClass -luokasta tuli TreeTest –luokka, joka testaa BinarySearchTree:n ja AVLTree:n lisäksi java.util.TreeMap punamustaa puuta, verrataksemme tuloksia.
- Testaamme puihin lisäämistä, niistä hakemista ja poistamista.
- Toteutuksessa lisättiin luokka IntStringPair :

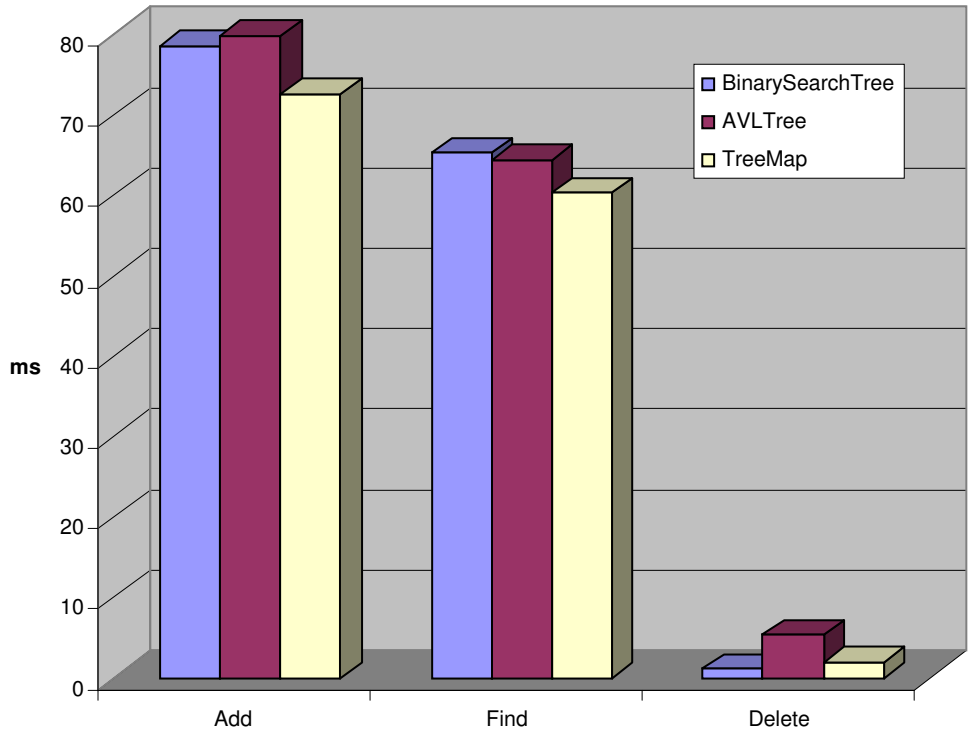
```
class IntStringPair implemets Comparable

IntStringPair-luokasta luodaan olioita, joiden attribuuteiksi
asetetaan testiaineiston dataa.
IntStringPair-luokka toteuttaa rajapinnan Comparable.

public Integer key;
public String value;

public int compareTo(Object o);
public boolean equals(Object o);
public String toString();
```

3. Testaus

TreeTest –luokka kysyy käyttäjältä testattavan puun, testiaineiston ja montako solmua puusta etsitään ja poistetaan.
Käytimme testiaineistoina annettuja data1HT.txt ja data2HT.txt tiedostoja, jotka sisälsivät kokonaisluku-merkkijono pareja. Tämän lisäksi järjestimme data1HT.txt kasvavaan numerojärjestykseen, ja otimme sen ensimmäiset 4000 riviä tiedostoon data1HT4k.txt. Tällä aineistoilla saimme näkyville BinarySearchTree:n heikkouden järjestetyn aineiston suhteen.

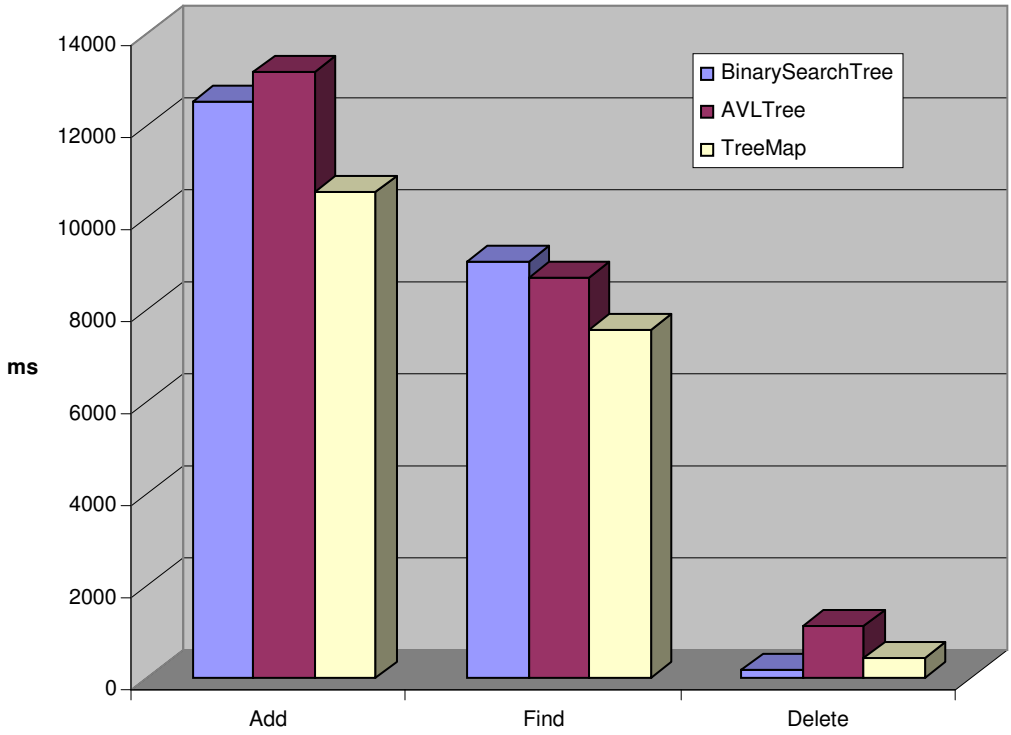
- Add on aineiston rivien lisäyksen viemä aika millisekunteina.
- Find kuvaa aikaa mikä kului kun puusta etsittiin n. solmua.
- Delete kuvaa aikaa mikä kului puun maksimin poistamiseen n. kertaa.

3.1. data1HT.txt, 10.000 hakua ja maksimin poistoa



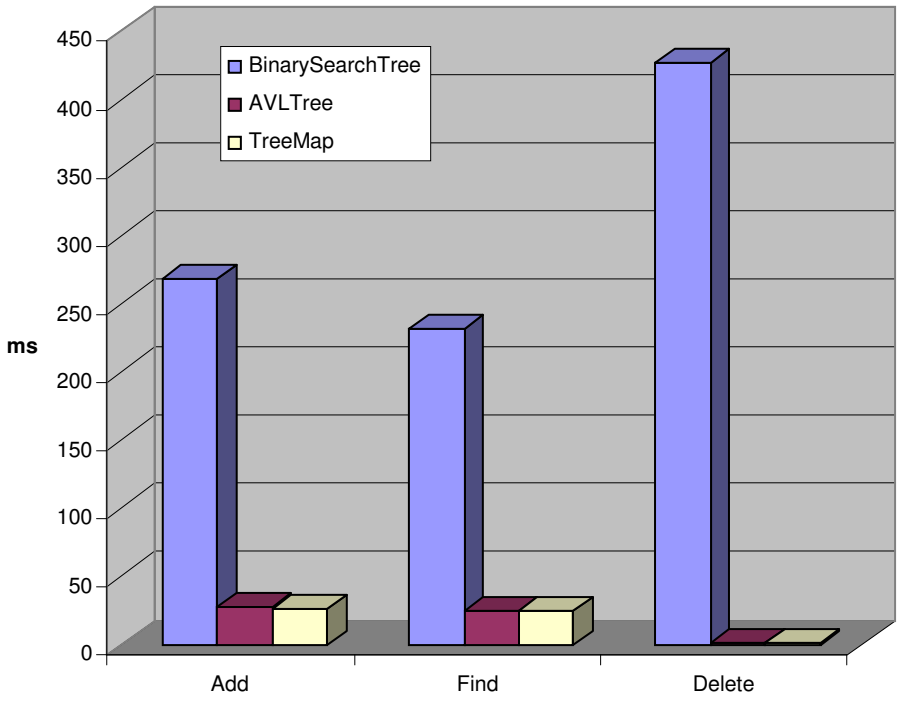
BinarySearchTree pärjää hyvin tasapainoitettujen puiden rinnalla.
AVLTree on hieman nopeampi hauissa, joka selittynee tasapainoituksilla. Sen sijaan lisääminen ja poistaminen AVLTree:hen on hitaampaa kuin BinarySearchTree:hen.

3.2. data2HT.txt, 1.000.000 hakua ja maksimin poistoa



BinarySearchTree on nopeampi lisättäessä ja poistettaessa puusta. AVLTree voittaa vain hauissa.

3.3. data1HT4k.txt järjestetty, 4000 hakua ja maksimin poistoa

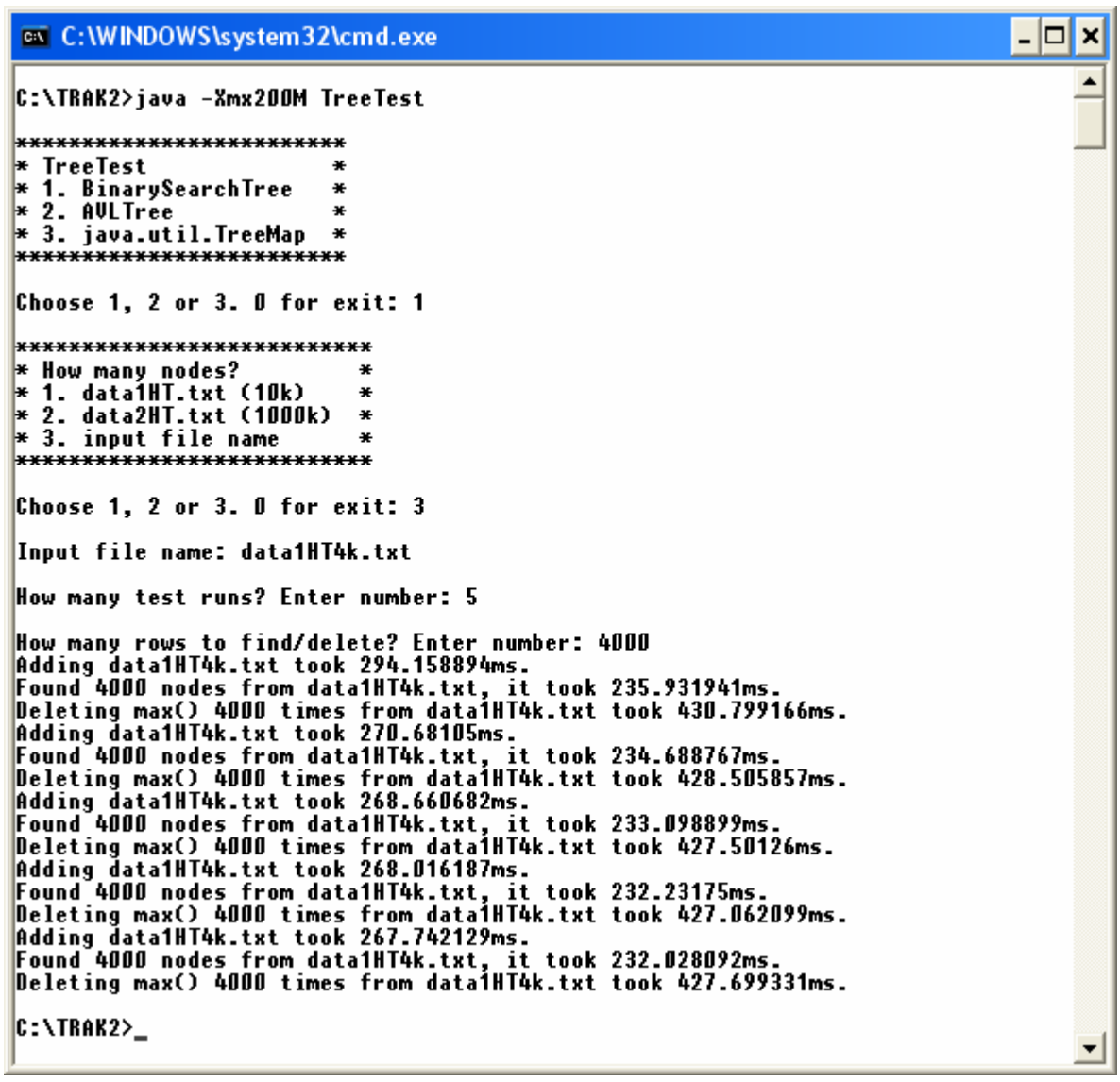


Käytettäessä valmiiksi suuruusjärjestyksessä olevaa aineistoa, BinarySearchTree:n suurin heikkous käy ilmi. AVLTree:n tasapainoitusten edut ovat selvät.

A. Käyttöohje

Ajaessasi java tulkkia, anna sille lisää muistia –Xmx200M kytkimellä, jotta testiajot eivät kaadu.
Oletetaan, että ohjelmakoodipaketti on purettu kansioon C:\TRAK2
Puiden testaus onnistuu TreeTest –luokalla.

```
C:\TRAK2\javac *.java
C:\TRAK2\java -Xmx200M TreeTest
```



TreeTest tallentaa testitulokset tiedostoihin.

B. Koodilistaukset

Tree.java

```
/**
 * @(#)Tree.java
 *
 *
 * @author Jonne Pohjankukka
 * @version 1.00 2008/3/6
 */
public interface Tree
  {
    public void add(Comparable key);

    public void delete(Comparable key);

    public Comparable min();

    public Comparable max();

    public Comparable find(Comparable key);

  }
```

BinaryNode.java

```
/**
 * BinaryNode on binäärisen hakupuun solmu.
 *
 * @author Simo Savonen
 * @version 1.0 (13.2.2008)
 */
public class BinaryNode {

  protected Comparable key; // solmun satelliittidata

  protected BinaryNode left;    // vasen lapsi
  protected BinaryNode right;    // oikea lapsi
  protected BinaryNode parent;  // solmun isä

  /**
   * Konstruktori. Viittaukset lapsisolmuihin ja isäsolmuun
   * asetetaan vasta lisättäessä solmu puuhun.
   *
   * @param data   satelliittidata, toteuttaa rajapinnan Comparable
   */
  public BinaryNode(Comparable data) {
    key = data;
    left = null;
    right = null;
    parent = null;
  }

  public Comparable getKey() { return key; }
  public BinaryNode getLeftChild() { return left; }
  public BinaryNode getRightChild() { return right; }
  public BinaryNode getParent() { return parent; }

  public void setKey(Comparable k) { key = k; }
  public void setLeftChild(BinaryNode node) { left = node; }
  public void setRightChild(BinaryNode node) { right = node; }
  public void setParent(BinaryNode node) { parent = node; }
}
```

BinarySearchTree.java

```
/**
 * Luokka mallintaa binäärisen hakupuun toimintoja.
 * Binäärinen hakupuu on tietynlainen binääripuu.
 *
 *  Operaatiot:
 *    - Haku     ( find() -> search()       )
 *    - Lisäys   ( add() -> insert()       )
 *    - Poisto   ( delete() -> remove()       )
 *    - Minimi   ( min() -> findMin()       )
 *    - Maksimi   ( max() -> find max()       )
 *    - Tyhjyys   ( isEmpty()           )
 *    - Seuraaja   ( findSuccessor -> successor()   )
 *
 * @author Jyri Lehtonen
 * @version 1.3 (12.3.2008)
 */
public class BinarySearchTree implements Tree {

/*********************************
 * ILMENTYMÄMUUTTUJAT
 *********************************/

  //Viittaa puun juureen, jos null niin puu on tyhjä.
  private BinaryNode root;


/*********************************
 * KONSTRUKTORI
 *********************************/
  public BinarySearchTree() {
    this.root = null;
  }


/*********************************
 * PROSEDUURIT
 *********************************/

  //Lisää alkion puuhun (kutsumalla apumetodia insert())
  public void add(Comparable x) {
    root = insert(x, root);
  }
  //Poistaa alkion puusta (kutsumalla apumetodia remove())
  public void delete(Comparable x) {
    remove(x,root);
  }
  //Etsii puun pienimmän alkion (kutsumalla apumetodia (findMin())
  public Comparable min() {
    return dataAt(findMin(root));
  }
  //Etsii puun suurimman alkio (kutsumalla apumetodia findMax())
  public Comparable max(){
    return dataAt(findMax(root));
  }
  //Etsii alkion puusta (kutsumalla apumetodia search())
  public Comparable find(Comparable x) {
    return search(x, root);
  }
  //Tarkistaa onko puu tyhjä
    public boolean isEmpty(){
      return root == null;
   }
```

```
/*********************************
 * FUNKTIOT
 *********************************/

  /**
   * Apumetodi, joka auttaa alkion datan hankinnassa.
   *
   * @.pre = on saatava parametri
   * @param t parametrinä saatu alkio
   * @.post palauttaa alkion datan tai null jos t on null.
   */
        private Comparable dataAt(BinaryNode t) {
      if (t == null) {
        return null;
      }

      else {
        return t.key;
      }
    }

  /**
   * Apumetodi, joka etsii pienimmän alkion puusta.
   * Idea: Lähdetään liikkeelle juuresta, ja pienin alkio on puun
   *       kaikista vasemmanpuoleisin alkio. Edetään niin kauan,
   *        kunnes alkiolla ei ole enää lasta vasemmalla puolella,
   *      tämä alkio on silloin pienin ja se palautetaan.
   *
   * @.pre = on saatava parametri
   * @param x parametrina saatu alkio (puun juuri)
   * @.post = palauttaa pienimmän alkion arvon puusta
   */
   private BinaryNode findMin(BinaryNode x) {
     if (x== null) {
       return null;
     }

     while (x.left != null) {
       x = x.left;
     }

     return x;
   }

   /**
    * Apumetodi, joka etsii suurimman alkion puusta.
    * Idea: Lähdetään liikkeelle juuresta, ja suurin alkio on puun
    *       kaikista oikeanpuoleisin alkio. Edetään niin kauan,
    *        kunnes alkiolla ei ole enää lasta oikealla puolella,
    *      tämä alkio on silloin suurin ja se palautetaan.
    *
    * @.pre = on saatava parametri
    * @param x parametrina saatu alkio (puun juuri)
    * @.post = palauttaa suurimman alkion arvon puusta
    */
   private BinaryNode findMax(BinaryNode x) {
     if (x== null) {
       return null;
     }

     while (x.right != null) {
       x = x.right;
     }

     return x;
   }
```

```
/**
 * Toteuttaa haun puuhun.
 * Jos verrattava on null, ilmoitetaan siitä palauttamalla null.
 *
 * Jos verrattava on jotain muuta, verrataan metodin parametrejä
 * keskenään (x ja t). Jos vertailun tulos on -1 (aka < 0)
 * asetetaan seuraavaan rekursiiviseen kutsuun haku vasempaan
 * lapseen. Jos vertailun tulos on 1 (aka > 0), asetetaan seuraavaan
 * rekursiiviseen kutsuun haku oikeaan lapseen. Jos vertailun tulos
 * on 0, (aka  == 0), on etsitty alkio löydetty, joten palautetaan se.
 *
 * @.pre = (x != null) && (t != null)
 * @param x on se alkio, jota puusta etsitään.
 * @param t on se alkio, joka on puun juurena.
 * @.post = palauttaa alkion, jota on etsitty.
 */
private Comparable search(Comparable x, BinaryNode t) {
  if (x == null) {
    return null;
  }

  else {
    if  ( x.compareTo(t.key) < 0 ) {
        return search(x, t.left); }
    if  ( x.compareTo(t.key) > 0 ) {
        return search(x, t.right); }
    if  ( x.compareTo(t.key) == 0 ) {
        return x; }
  }

  return null;
}

/**
 * Toteuttaa lisäyksen puuhun.
 * Jos puu on tyhjä (null), luodaan uusi alkio (ali)puun juureen.
 * Jotta lisäys löytää oikeaan paikkaan puussa, on verrattava
 * puun alkioita haun tapaisesti kunnes oikea pakka on löytynyt.
 *     compareTo arvo -1 -> Edetään vasemmalle
 *    compareTo arvo +1 -> Edetään oikealle
 *    compareTo arvo 0 -> Duplikaatti, ei tehdä mitään.
 *
   * @.pre = lisättävä alio on annettava
   * @param x lisättävä alkio
   * @param t puun juuri
   * @return uusi juuri
   * @.post = uusi alkio on lisätty puuhun
   */
   private BinaryNode insert(Comparable x, BinaryNode node) {
     if (node==null) {
       node=new BinaryNode(x);
       node.setParent(node);
     }

     else {
       if (x.compareTo(node.key) < 0) {
         node.left = insert(x, node.left); }
       if (x.compareTo(node.key) > 0) {
         node.right= insert(x, node.right); }
     }

     return node;
   }
```

```
/**
 * Toteuttaa poiston puusta.
 * Jos alkiota ei löydetä, ei tehdä mitään.
 *
 * Tämän jälkeen on käsiteltävä binäärisen hakupuun poiston
 * erilaiset tapaukset,
 *    tapaus 1: solmulla ei ole lapsia (null & null)
 *    tapaus 2: solmulla on yksi lapsi (null & not null)
 *    tapaus 3: solmulla on kaksi lasta (not null & not null)
 *          (talletetaan seuraajan avainarvo ja data
 *          ja poistetaan seuraaja alkio. Kopioidaan
 *          seuraajan tiedot poistettavan alkion päälle)
 *
 * @param x poistettava solmu
 * @param t puun juuri
 * @return tapauskohtainen
 */
private BinaryNode remove(Comparable x, BinaryNode t) {
 BinaryNode y;

 if (t == null) {
   return t;
 }

 else {  //Etsitään vasemmalta
      if(x.compareTo(t.getKey()) < 0) {
        t.setLeftChild(remove(x, t.getLeftChild()));
      }
      //Etsitään oikealta
      if(x.compareTo(t.getKey()) > 0) {
        t.setRightChild(remove(x, t.getRightChild()));
     }
     //Löytyi, käsitellään tapaukset
     if(x.compareTo(t.getKey()) == 0) {
       if(t.getLeftChild() == null & t.getRightChild() == null) {
              return null;
       }
       if(t.getLeftChild() != null & t.getRightChild() == null) {
         t.getLeftChild().setParent(t.getParent());
           t = t.getLeftChild();
           return t;
       }
       if(t.getLeftChild() == null & t.getRightChild() != null) {
           t.getRightChild().setParent(t.getParent());
           t = t.getRightChild();
           return t;
         }
         if(t.getLeftChild() != null & t.getRightChild() != null) {
           Comparable succKey = successor(t).getKey();
           t = remove(succKey, t);
           t.setKey(succKey);
           //return t;
         }
       }
     }

   return t;
 }
```

```
  /**
   * Etsii annetun alkion seuraajan.
   *
   * Jos oikea alipuu ei ole tyhjä, niin x:n seuraaja on oikean alipuun
   * pienin alkio. Oikealla on x:ää suuremmat, otetaan niistä pienin.
   * Jos oikea alipuu on tyhjä, niin x:n seuraaja etsitään kipuamalla
   * ylös puuta, kunnes löydetään ensimmäinen solmu y, jonka vasempaan
   * alipuuhun x kuuluu. Se on ensimmäinen alkio, joka on x:ää suurempi.
   */
  public BinaryNode successor(BinaryNode x) {
    if (x.right != null) {
      return findMin(x.right);}

    BinaryNode y = x.parent;
    while ((y != null) && (x.equals(y.right))) {
      x = y;
      y = y.parent;
    }
    return y;
  }

  /****************************************
     * MAIN METODI PUUN TOIMINNAN TESTAUKSEEN
     ****************************************/

    /*
    public static void main(String [] args) {

      BinarySearchTree t = new BinarySearchTree();
          t.add(10);  //      10
           t.add(5);  //    5    15
           t.add(15);  //    3   7   12   18
           t.add(7);  //
           t.add(12);
           t.add(3);
           t.add(18);

           System.out.println("Puun Juuri: " +t.root.getKey());
           System.out.println("Juuren vasen lapsi: "+t.root.getLeftChild().getKey());
           System.out.println("Juuren oikea lapsi: "+t.root.getRightChild().getKey());

      System.out.println("Puun minimi: " +t.min());
         System.out.println("Puun maksimi: "+t.max());

         System.out.println("Haku, haetaan 7: "+ t.find(7));

         System.out.println("Poistetaan root! (oli 10, oltava 12 nyt)");
         t.delete(t.root.getKey());
         System.out.println("Haetaan uusi root: " +t.root.getKey());
         System.out.println("Poistetaan 15 (yksi lapsi nyt)");
         t.delete(15);
  }*/

}
```

AVLNode.java

```
/**
 * AVLNode on AVL-puun tasapainoitettu solmu.
 *
 * @author Jonne Pohjankukka
 * @version 2.0 2008/3/6
 */
public class AVLNode
  {

    //JÄSENMUUTTUJAT

    private int height; // solmun korkeus
    private AVLNode left; // vasen lapsi
    private AVLNode right; // oikea lapsi
    private AVLNode parent; // vanhempi
    private Comparable key; // solmun satelliittidata

    // KONSTRUKTORI

    /**
     * @param data satelliittidata, toteuttaa rajapinnan Comparable.
     * @post AVLNode konstruoitu
     */
    public AVLNode(Comparable data)
      {
        key = data;
        left = null;
        right = null;
        parent = null;
        height = 0; // solmu lisätään aina lehdeksi
      }

    // FUNKTIOT

    public Comparable getKey() { return key; }
    public AVLNode getLeftChild() { return left; }
    public AVLNode getRightChild() { return right; }
    public AVLNode getParent() { return parent; }
    public int getHeight() { return height; }

    // METODIT
    public void setKey(Comparable newKey) { key = newKey; }
    public void setLeftChild(AVLNode node) { left = node; }
    public void setRightChild(AVLNode node) { right = node; }
    public void setParent(AVLNode node) { parent = node; }
    public void setHeight(int h) { height = h; }

  }//class AVLNode
```

AVLTree.java

```
/**
 * @(#)AVLTree.java
 *
 *
 * @author Jonne Pohjankukka
 * @version 2.00 2008/3/6
 */

public class AVLTree implements Tree{

  // JÄSENMUUTTUJA AVL-puun juuri
  private AVLNode root;

  /**
   * AVL-puun konstruktori
   */
    public AVLTree()
      {
        root = null;
      }

    /**
     * julkinen metodi rajapinnalle,
     * joka palauttaa AVL-puun minimin.
     * @param AVLNode Root
     * @post minimi palautettu
     */
    public Comparable min()
      {
        return min(root).getKey();
      }
    /**
     * julkinen metodi rajapinnalle,
     * joka palauttaa AVL-puun maksimin.
     * @param AVLNode Root
     * @post minimi palautettu
     */
    public Comparable max()
      {
        AVLNode result = max(root);
        if(result != null) return result.getKey();
        else return null;
      }

    /**
     * julkinen metodi rajapinnalle, joka etsii solmun AVL-puusta.
     * @param Comparable key
     * @pre key != null
     * @post etsitty solmu palautettu
     */
    public Comparable find(Comparable key)
      {
        return find(key, root);
      }

    /**
     * julkinen metodi rajapinnalle, joka lisää solmun AVL-puuhun.
     * @param Comparable key
     * @pre key != null
     * @post solmu lisätty
     */
    public void add(Comparable key)
      {
        root = add(key, root, null);
      }
```

```
/**
 * julkinen metodi rajapinnalle, joka poistaa solmun AVL-puusta.
 * @param Comparable key
 * @pre key != null
 * @post solmu poistettu
 */
public void delete(Comparable key)
  {
    root = delete(key, root);
  }

/**
 * Julkinen metodi, joka tulostaa puun solmujen
 * avainarvot suuruusjärjestyksessä.
 * @param AVLNode t
 * @pre t != null
 * @post puu tulostettuna
 */
public void outputTree(AVLNode node)
   {
     if( node != null )
       {
          outputTree( node.getLeftChild() );
          System.out.println( node.getKey() );
          outputTree( node.getRightChild() );
        }
   }

/**
 * privaatti metodi, joka palauttaa AVL-puun minimin.
 * @param AVLNode Root
 * @post minimi palautettu
 */
private AVLNode min(AVLNode Root)
  {
  if(Root == null) return null;

  else
    {
    AVLNode node = Root;
    while(node.getLeftChild() != null)
      {
        node = node.getLeftChild();
      }

      return node;

    }
  }

/**
 * privaatti metodi, joka palauttaa AVL-puun maksimin.
 * @param AVLNode Root
 * @post maksimi palautettu
 */
private AVLNode max(AVLNode Root) {
  if(Root == null) return null;
  else
    {
      AVLNode node = Root;
      while(node.getRightChild() != null)
        {
          node = node.getRightChild();
       }
      return node;
    }
  }
```

```
/**
 * privaatti metodi, joka etsii
 * parametrina annetun solmun.
 * @param Comparable key
 * @param AVLNode node
 * @pre key != null
 * @post etsitty solmu palautettu
 */
private Comparable find(Comparable key, AVLNode node)
  {
    while(node != null)
      {
        if(key.compareTo(node.getKey()) == -1)
          {
            node = node.getLeftChild();
          }
        else
          {
            if(key.compareTo(node.getKey()) == 1)
              {
                node = node.getRightChild();
              }

            else
              {
                // solmu löytyi
                return node.getKey();
              }
          }
      }
  return null; // kyseistä solmua ei ole puussa

  }

 /**
 * privaatti metodi, joka etsii
 * parametrina annetun solmun. Huom.
 * palauttaa AVLNode:n ei sen dataa.
 * @param Comparable key
 * @param AVLNode node
 * @pre key != null
 * @post etsitty solmu palautettu
 */
private AVLNode findNode(Comparable key, AVLNode node)
  {
    while(node != null)
      {
        if(key.compareTo(node.getKey()) == -1)
          {
            node = node.getLeftChild();
          }
        else
          {
            if(key.compareTo(node.getKey()) == 1)
              {
                node = node.getRightChild();
              }

            else
              {
                // solmu löytyi
                return node;
              }
          }
      }
  return null; // kyseistä solmua ei ole puussa
  }
```

```
/**
 * privaatti metodi, joka poistaa solmun AVL-puusta.
 * @param AVLNode node
 * @param Comparable key
 * @pre key != null
 * @post solmu poistettu
 */
private AVLNode delete(Comparable key, AVLNode node)
  {
if(node == null)
{
  return null;
}

if(key.compareTo(node.getKey()) == 0) // SOLMU LÖYTYI
  {
    if(node.getLeftChild() == null & node.getRightChild() == null)
    {
      return null;
    }
    if(node.getLeftChild() != null & node.getRightChild() == null)
    {
      node.getLeftChild().setParent(node.getParent());
      node = node.getLeftChild();
      return node;
    }
    if(node.getLeftChild() == null & node.getRightChild() != null)
    {
      node.getRightChild().setParent(node.getParent());
      node = node.getRightChild();
      return node;
    }
    if(node.getLeftChild() != null & node.getRightChild() != null)
    {
      int korkennen = node.getHeight();
      Comparable successorKey = successor(node).getKey();
      Comparable nodeKey = node.getKey();
      AVLNode nodeOrig = node;
      node = delete(successorKey, node);
      int korkjälkeen = node.getHeight();
      if(korkennen != korkjälkeen) {node.setHeight(maxKorkeus(node.getLeftChild(),
        node.getRightChild()) + 1);}
      nodeOrig.setKey(successorKey);
      return node;
    }
  }
        if(key.compareTo(node.getKey()) == -1)
          {
            node.setLeftChild(delete(key, node.getLeftChild()));
            if(height(node.getRightChild())-height(node.getLeftChild()) ==2)
              {
                if((node.getRightChild() != null &
                     node.getRightChild().getRightChild() != null) |
                   (node.getRightChild() != null &
                     node.getRightChild().getRightChild() != null &
                    node.getRightChild().getLeftChild() != null))
                {
                  node = leftRotate(node);
                }
                else
                {
                  node = doubleRotateRight(node);
                }
              }
          }
        else {
            if(key.compareTo(node.getKey()) == 1) {
              node.setRightChild(delete(key, node.getRightChild()));
```

```
            if(height(node.getLeftChild())-height(node.getRightChild()) == 2)
            {
              if((node.getLeftChild() != null &
                  node.getLeftChild().getLeftChild() != null) |
                 (node.getLeftChild() != null &
                   node.getLeftChild().getLeftChild() != null &
                   node.getLeftChild().getRightChild() != null))
              {
                node = rightRotate(node);
              }
              else
              {
                node = doubleRotateLeft(node);
              }
            }
          }
      node.setHeight(maxKorkeus(node.getLeftChild(), node.getRightChild()) + 1);
      return node;
    }

/**
 * privaatti metodi, joka lisää AVL-puuhun solmun.
 * @param AVLNode node, parent
 * @param Comparable key
 * @pre key != null
 * @post solmu lisätty
 */
private AVLNode add(Comparable key, AVLNode node, AVLNode parent)
  {
  if(node == null)
    {
      node = new AVLNode(key);
      node.setParent(parent);
    }
  else {
    if(key.compareTo(node.getKey()) == -1)
      {
      node.setLeftChild(add(key, node.getLeftChild(), node));
        if(height(node.getLeftChild())-height(node.getRightChild()) == 2) {
          if(key.compareTo(node.getLeftChild().getKey()) == -1) {
              node = rightRotate(node);
            }
          else {
              node = doubleRotateLeft(node);
            }
        }
      }
    else {
      if(key.compareTo(node.getKey()) == 1)
        {
          node.setRightChild(add(key, node.getRightChild(), node));
            if(height(node.getRightChild())-height(node.getLeftChild()) == 2) {
              if(key.compareTo(node.getRightChild().getKey()) == 1) {
                  node = leftRotate(node);
                }
              else {
                  node = doubleRotateRight(node);
                }
            }
        }
      }
      node.setHeight(maxKorkeus(node.getLeftChild(), node.getRightChild()) + 1);
      return node;
    }
```

```
/**
 * apumetodi lisäykselle ja poistolle,
 * joka palauttaa solmun korkeuden.
 * @param AVLNode node
 * @post korkeus palautettu
 */
private int height(AVLNode node)
  {
    return node == null ? -1 : node.getHeight();
  }

/**
 * palauttaa solmun lapsien suuremmman korkeuden.
 * @param AVLNode leftC, rightC
 * @pre leftC != null & rightC != null
 * @post suurempi korkeus palautettu
 */
private int maxKorkeus(AVLNode leftC, AVLNode rightC)
  {
  int leftH, rightH;
  if(leftC == null)
    {
      leftH = -1;
    }
  else
    {
       leftH = leftC.getHeight();
    }
  if(rightC == null)
    {
      rightH = -1;
    }
  else
    {
      rightH = rightC.getHeight();
    }

  return leftH > rightH ? leftH : rightH;

  }

/**
 * privaatti metodi, joka palauttaa solmun seuraajan.
 * @param AVLNode node
 * @pre node != null
 * @post seuraajasolmu palautettu
 */
private AVLNode successor(AVLNode node)
  {
    if(node.getRightChild() != null)
      {
        return min(node.getRightChild());
      }
    AVLNode nodeY = node.getParent();
    while(nodeY != null & node == node.getRightChild())
      {
        node = nodeY;
        nodeY = nodeY.getParent();
      }
    return nodeY;
  }
```

```
/**
 * Vasen rotaatio solmulla node.
 * @param AVLNode node
 * @pre node != null
 * @post rotaatio tehty
 */
private AVLNode leftRotate(AVLNode node)
  {

    AVLNode rightChild = node.getRightChild(); // solmun oikea lapsi
    node.setRightChild(rightChild.getLeftChild()); // solmun oikeaksi lapseksi
        asetetaan oikean lapsen vasen lapsi
    rightChild.setParent(node.getParent()); //   oikean lapsen vanhemmaksi entisen
        vanhemman vanhempi
    node.setParent(rightChild); // solmun vanhemmaksi asetetaan oikea lapsi
    rightChild.setLeftChild(node); // oikean lapsen vasemmaksi lapseksi solmu

    // päivitetään solmujen korkeudet
    node.setHeight(maxKorkeus(node.getLeftChild(), node.getRightChild()) + 1);
    rightChild.setHeight(maxKorkeus(rightChild.getLeftChild(),
        rightChild.getRightChild()) + 1);

    return rightChild; // palautetaan oikea lapsi
  }

/**
 * Oikea rotaatio solmulla node.
 * @param AVLNode node
 * @pre node != null
 * @post rotaatio tehty
 */
private AVLNode rightRotate(AVLNode node)
  {

    AVLNode leftChild = node.getLeftChild(); // solmun vasen lapsi
    node.setLeftChild(leftChild.getRightChild()); // solmun vasemmaksi lapseksi
        asetetaan vasemman lapsen oikea lapsi
    leftChild.setParent(node.getParent()); //   vasemman lapsen vanhemmaksi entisen
        vanhemman vanhempi
    node.setParent(leftChild); // solmun vanhemmaksi asetetaan vasen lapsi
    leftChild.setRightChild(node); // vasemman lapsen oikeaksi lapseksi solmu

    // päivitetään solmujen korkeudet
    node.setHeight(maxKorkeus(node.getLeftChild(), node.getRightChild()) + 1);
    leftChild.setHeight(maxKorkeus(leftChild.getLeftChild(),
        leftChild.getRightChild()) + 1);

    return leftChild; // palautetaan vasen lapsi
  }

/**
 * Kaksoisrotaatio
 * Ensin tehdään rotaatio vasemmalle solmun vasemmalla lapsella
 * ja sitten oikea rotaatio solmulla.
 * @param AVLNode node
 * @pre node != null
 * @post rotaatio tehty
 */
private AVLNode doubleRotateLeft(AVLNode node)
  {
    node.setLeftChild(leftRotate(node.getLeftChild()));
    return rightRotate(node);
  }
```

```
    /**
     * Kaksoisrotaatio
     * Ensin tehdään rotaatio oikealle solmun oikealla lapsella
     * ja sitten vasen rotaatio solmulla.
     * @param AVLNode node
     * @pre node != null
     * @post rotaatio tehty
     */
    private AVLNode doubleRotateRight(AVLNode node)
      {
        node.setRightChild(rightRotate(node.getRightChild()));
        return leftRotate(node);
      }


} //class AVLTree
```

IntStringPair.java

```
/**
 * IntStringPair on binääripuiden testaukseen
 * käytettävä Comparable rajapinnan tot. luokka.
 *
 * @author Simo Savonen
 * @version 1.1 (11.3.2008)
 */
public class IntStringPair implements Comparable {

  public Integer key;
  public String value;

  public IntStringPair(Integer k, String v) {
    key = k;
    value = v;
  }

  public int compareTo(Object o) {
    IntStringPair isp = (IntStringPair)o;
    return key.compareTo(isp.key);
  }

  public boolean equals(Object o) {
    IntStringPair isp = (IntStringPair)o;
    return key.equals(isp.key);
  }

  public String toString() {
    return key + ", " + value;
  }
}
```

TreeTest.java

```
import java.io.*;
import java.util.*;

/**
 * TreeTest testaa binääripuita.
 *
 * @author Simo Savonen
 * @version 1.0 (10.3.2008)
 */
public class TreeTest {

  /**
   * Testaa Tree -rajapinnan toteuttavia binääripuita.
   */
  private static void testTree(int choise) {
    String file = chooseTestData();
    int tests = howManyTests();
    int rows = howManyRows();

    try {
      FileWriter fstream = null;

      if(choise == 1) fstream = new FileWriter(tests + "_BinarySearchTree_" + file);
      else if(choise == 2) fstream = new FileWriter(tests + "_AVLTree_" + file);

      BufferedWriter out = new BufferedWriter(fstream);
      out.write("Add\tFind\tDelete"); // otsikot sarakkeille
      out.newLine();

      Scanner scanner = null;

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

        Tree tree = null;

        if(choise == 1) tree = new BinarySearchTree();
        else if (choise == 2) tree = new AVLTree();

        /** add */
        scanner = new Scanner(new File(file));

        long addStart = System.nanoTime(); // aloitetaan ajanotto

        while(scanner.hasNextInt()) {
          tree.add(new IntStringPair(scanner.nextInt(), scanner.next()));
        }

        double addElapsed = (System.nanoTime() - addStart) / 1000000.0; // kesto

        System.out.println("Adding " + file + " took " + addElapsed + "ms.");

        scanner.close();
```

```
      /** find */
      scanner = new Scanner(new File(file));

      long findStart = System.nanoTime(); // aloitetaan ajanotto

      int j = 1;
      int laskuri = 0;
      while(scanner.hasNextInt() & j <= rows) {
        if(tree.find(new IntStringPair(scanner.nextInt(), scanner.next())) != null){
        laskuri++; }
        j++;

      }

      double findElapsed = (System.nanoTime() - findStart) / 1000000.0; // kesto

      System.out.println("Found " + laskuri + " nodes from " + file +
                         ", it took " + findElapsed + "ms.");

      scanner.close();

      /** delete */
      long deleteStart = System.nanoTime(); // aloitetaan ajanotto

      int k = 1;
      while(k <= rows) {
        tree.delete(tree.max());
        k++;

      }

      double deleteElapsed = (System.nanoTime() - deleteStart) / 1000000.0; // kesto

      System.out.println("Deleting max() " + rows + " times from " + file +
                         " took " + deleteElapsed + "ms.");

      /** kirjoitetaan tulokset tiedostoon */
      out.write(addElapsed + "\t" + findElapsed + "\t" + deleteElapsed);
      out.newLine();

      tree = null;

    } // for

    out.close();
  }
  catch(Exception e) {
    e.printStackTrace();
  }
}
```

```
/**
 * Testaa java.util.TreeMap punamustaa binääripuuta.
 */
private static void testTreeMap() {
  String file = chooseTestData();
  int tests = howManyTests();
  int rows = howManyRows();

  try {
    FileWriter fstream = new FileWriter(tests + "_TreeMap_" + file);
    BufferedWriter out = new BufferedWriter(fstream);
    out.write("Add\tFind\tDelete"); // otsikot sarakkeille
    out.newLine();

    Scanner scanner = null;

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

      TreeMap<Integer, String> tree = new TreeMap<Integer, String>();

      /** add */
      scanner = new Scanner(new File(file));

      long addStart = System.nanoTime(); // aloitetaan ajanotto

      while(scanner.hasNextInt()) {
        tree.put(scanner.nextInt(), scanner.next());
      }

      double addElapsed = (System.nanoTime() - addStart) / 1000000.0; // kesto

      System.out.println("Adding " + file + " took " + addElapsed + "ms.");

      scanner.close();


      /** find */
      scanner = new Scanner(new File(file));

      long findStart = System.nanoTime(); // aloitetaan ajanotto

      int j = 1;
      int laskuri = 0;
      while(scanner.hasNextInt() & j <= rows) {
        if(tree.get(scanner.nextInt()) != null) laskuri++;
        scanner.next();
        j++;

      }

      double findElapsed = (System.nanoTime() - findStart) / 1000000.0; // kesto

      System.out.println("Found " + laskuri + " nodes from " + file +
                         ", it took " + findElapsed + "ms.");

      scanner.close();
```

```
      /** delete */
      long deleteStart = System.nanoTime(); // aloitetaan ajanotto

      int k = 1;
      while(k <= rows) {
        try {
          tree.remove(tree.lastKey());
        }
        catch(NoSuchElementException nse) {
          // ei välitetä mahd. dublikaateista
        }
        k++;
      }

      double deleteElapsed = (System.nanoTime() - deleteStart) / 1000000.0; // kesto

      System.out.println("Deleting lastKey() " + rows + " times from " + file +
                         " took " + deleteElapsed + "ms.");

      /** kirjoitetaan tulokset tiedostoon */
      out.write(addElapsed + "\t" + findElapsed + "\t" + deleteElapsed);
      out.newLine();

      tree = null;
    } // for

    out.close();
  }
  catch(Exception e) {
    System.err.println("Error: " + e);
  }
}

public static void main(String[] args) {
  System.out.println("\n*************************");
  System.out.println("* TreeTest              *");
  System.out.println("* 1. BinarySearchTree   *");
  System.out.println("* 2. AVLTree            *");
  System.out.println("* 3. java.util.TreeMap  *");
  System.out.println("*************************");

  System.out.print("\nChoose 1, 2 or 3. 0 for exit: ");

  try {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    int choise = Integer.parseInt(br.readLine().trim());
    switch(choise) {
      case 0:
        System.exit(0);
      case 1:
        testTree(choise);
        break;
      case 2:
        testTree(choise);
        break;
      case 3:
        testTreeMap();
        break;
      default:
        System.out.println("Invalid choise entered, exiting program.");
        break;
    }
  }
  catch(IOException e) {
    System.err.println("Exception: " + e);
  }
}
```

```
  private static String chooseTestData() {

    System.out.println("\n**************************");
    System.out.println("* How many nodes?         *");
    System.out.println("* 1. data1HT.txt (10k)    *");
    System.out.println("* 2. data2HT.txt (1000k)  *");
    System.out.println("* 3. input file name      *");
    System.out.println("**************************");

    System.out.print("\nChoose 1, 2 or 3. 0 for exit: ");

    try {
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      int choise = Integer.parseInt(br.readLine().trim());

      switch(choise) {
        case 0:
          System.exit(0);
        case 1:
          return new String("data1HT.txt");
        case 2:
          return new String("data2HT.txt");
        case 3:
          System.out.print("\nInput file name: ");
          return new String(br.readLine().trim());
        default:
          return null;
      }
    }
    catch(IOException e) {
      System.err.println("Exception: " + e);
    }

    return null;

  }

  private static int howManyTests() {

    System.out.print("\nHow many test runs? Enter number: ");

    try {
      BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
      return Integer.parseInt(br.readLine().trim());
    }
    catch(IOException e) {
      System.err.println("Exception: " + e);
    }
    return 0;
  }

  private static int howManyRows() {

      System.out.print("\nHow many rows to find/delete? Enter number: ");

      try {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        return Integer.parseInt(br.readLine().trim());
      }
      catch(IOException e) {
        System.err.println("Exception: " + e);
      }
      return 0;
  }
}
```