Rolf Axelsson

Laboration 12 – binärt sökträd, balanserat träd

Packa upp laborationsfilerna i paketet **laboration12**.

Uppgifter

1. Kör main-metoden i **Laboration12.java**. Se till att metoden **exercise1** anropas. Rita på ett papper hur trädet som skapas ser ut.

   Kontrollera sedan ditt resultat genom att ändra sista raden i exercise1() till
   ```
   bst.root().showTree();
   ```
   och köra programmet på nytt.

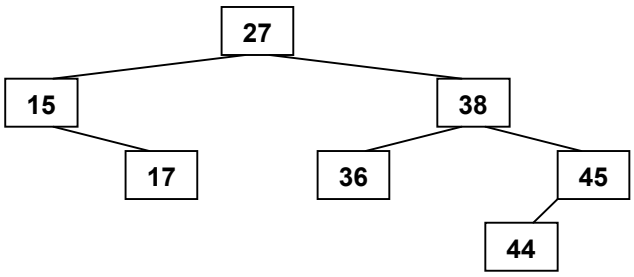
2. Kör main-metoden i **Laboration12.java**. Se till att metoden **exercise2** anropas. Rita på ett papper hur trädet som skapas ser ut.

   Kontrollera sedan ditt resultat genom att lägga till raden
   ```
   bst.root().showTree();
   ```
   sist i metoden exercise2 och köra programmet på nytt.

3. Kör main-metoden i **Laboration12.java**. Se till att metoden **exercise3** anropas. Rita på ett papper hur trädet som skapas ser ut.

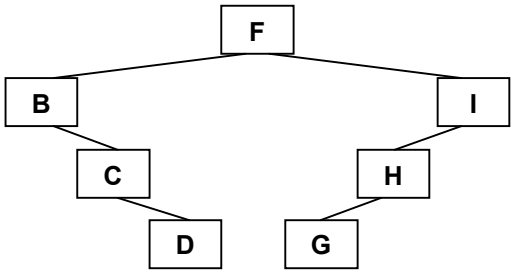
   Kontrollera sedan ditt resultat genom att lägga till raden
   ```
   bst.root().showTree();
   ```
   sist i metoden exercise3 och köra programmet på nytt.

4. Skapa ett binärt sökträd med nedanstående utseende. Nycklarna är av typen Integer.



   Placera din kod i metoden **exercise4**.

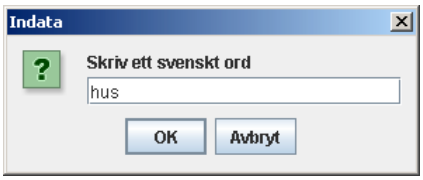
5. Skapa ett binärt sökträd med nedanstående utseende. Nycklarna är av typen String.



   Placera din kod i metoden **exercise5**.

6. Metoden **svenskEngelsk** skapar ett träd med följande innehåll:
   Nyckel (key): ett ord på svenska
   Värde (value): samma ord på engelska
   Komplettera **exercise6** med kod så att användaren upprepat får mata in svenska ord (tills användaren matar in ”slut”).
   Om ordet finns i trädet, t.ex. hus, så ger programmet utskriften

```
hus ger resultatet house
```



   Om ordet inte finns i trädet, t.ex. trädgård, så ger programmet utskriften

```
trädgård ger resultatet null
```

7. I metoden **capitals** skapas ett objekt av typen TreeMap (finns i paketet java.util). En TreeMap innehåller bl.a. metoderna **put(key, value)** och **get(key) : value**. Både *key* och *value* är någon typ av objekt.

   Komplettera metoden **capitals** med kod vilken lägger till par med följande innehåll:
   Nyckel: Ett land (t.ex. ”Sverige”)
   Värde: Huvudstaden i landet (t.ex. ”Stockholm”)

   Minst 10 (land, huvudstad) - par ska lagras i trädet.

   Skriv sedan metoden **exercise7** vilken ska låta användaren mata in namnet på ett land och om landet är en nyckel i trädet så ska landets huvudstad skrivas ut. Om staden inte är en nyckel ska meningen ”huvudstad i AAA är okänd” (AAA är det inmatade landet). Användaren ska få skriva in nya land ända tills användaren matar in ”slut” eller klickar på Avbryt.

8a. Komplettera klassen **BinarySearchTree** med metoden **public int size1()** vilken ska returnera antalet element som lagras i trädet. Lös uppgiften genom att anropa size-metoden i klassen BSTNode.

   Testa metoden på ett träd utan element (resultatet ska bli 0) och ett träd med ett antal element.

8b. Komplettera klassen **BinarySearchTree** med metoden **public int size2()** vilken ska returnera antalet element som lagras i trädet. Lös uppgiften utan att använda dig av size-metoden i klassen BSTNode.

   Algoritm
   Om en node inte är null så är antalet element (inklusive noden)
   1 + antal element i vänster subträd + antal element i höger subträd.

   Det är lämpligt att *size2()*-metoden returnerar värdet av anrop till en *size2(Node<K,V>)*-metod. Du har hjälp av att studera metoden **height** i klassen **BinarySearchTree**.

   Testa metoden på ett träd utan element (resultatet ska bli 0) och ett träd med ett antal element.

8c. Klassen **BinarySearchTree** innehåller metoden **public int size()** vilken returnerar värdet i instansvariabeln **size**. Problemet är att variabelns värde inte ökar när användaren placerar element i trädet och minskar när element tas bort. Se till att *size* ändras på lämpligt sätt i *put*- resp *remove*-metoden.

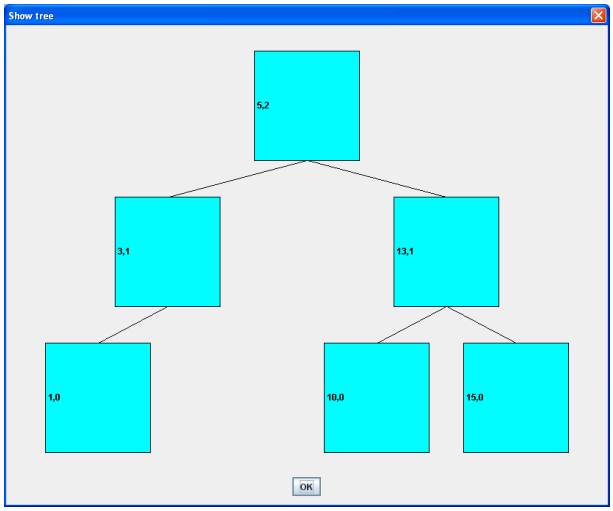
9. Komplettera klassen **BinarySearchTree** med metoden **public V first()** vilken ska returnera värdet i första elementet i trädet (elementet med minst nyckel). Metoden kan följa algoritmen:

Algoritm
```
tilldela variabeln node värdet av tree
om node == null
   returnera null
medan node.left != null
   tilldela node värdet av node.left
returnera node.value
```

10. Komplettera klassen **BinarySearchTree** med metoden **last()** vilken ska returnera värdet i det sista elementet i trädet (elementet med störst nyckel).
Denna gång gäller det att finna elementet längst till höger i trädet.

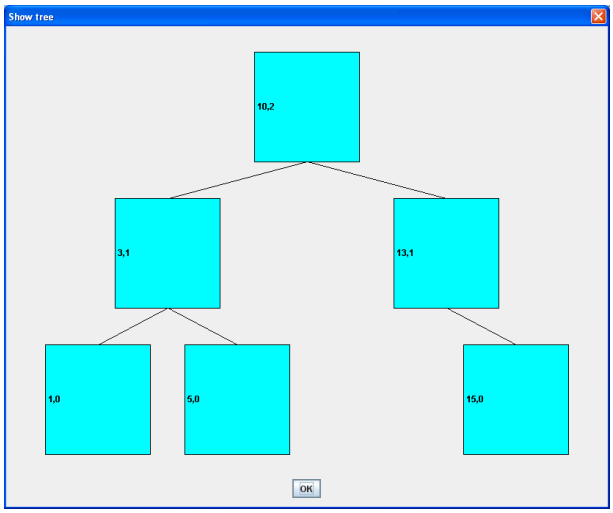
11a Komplettera metoden
private <K,V> AVLNode<K,V> rotateRight(AVLNode<K,V> node)
i klassen **Laboration12** med kod. Metoden ska genomföra en rotation åt höger runt noden *node* och slutligen returnera en referens till noden som är högst upp efter rotationen.
I föreläsningsunderlaget ser du vilka referenser som ska ändras. Försök lösa problemet på egen hand. Om du inte lyckas kan du följa denna algoritm:

- tilldela lokala variabeln *rootNode* referens till *node.left*
- tilldela *node.left* referensen till *rootNode.right*
- tilldela *rootNode.right* referensen till *node*
- returnera *rootNode*



exercise11a ska ge trädet i figuren till höger (andra figuren som visas) då **rotateRight** fungerar korrekt.

11b Komplettera metoden
private <K,V> AVLNode<K,V> rotateLeft(AVLNode<K,V> node)
i klassen **Laboration12** med kod. Metoden ska genomföra en rotation åt vänster runt noden *node* och slutligen returnera en referens till noden som är högst upp efter rotationen.
I föreläsningsunderlaget ser du vilka referenser som ska ändras. Försök lösa problemet på egen hand.

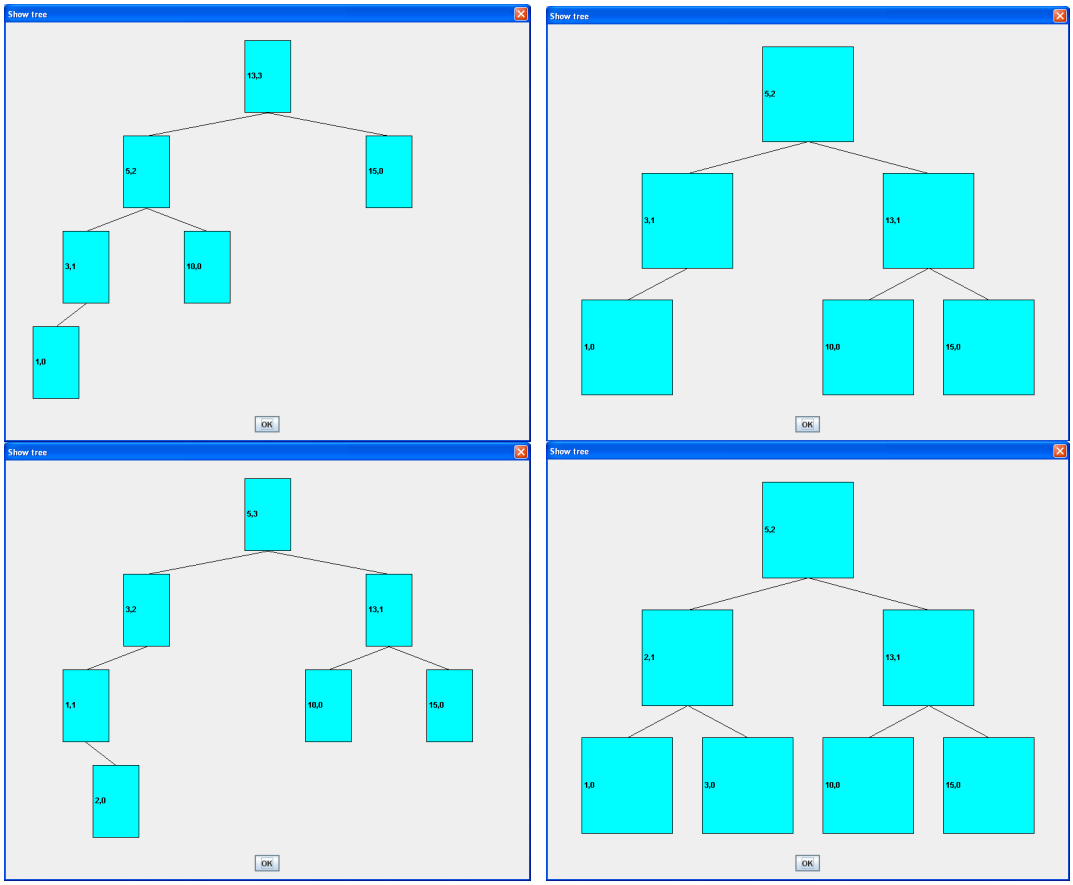


exercise11b ska ge trädet i figuren till höger (andra figuren som visas) då **rotateLeft** fungerar korrekt.

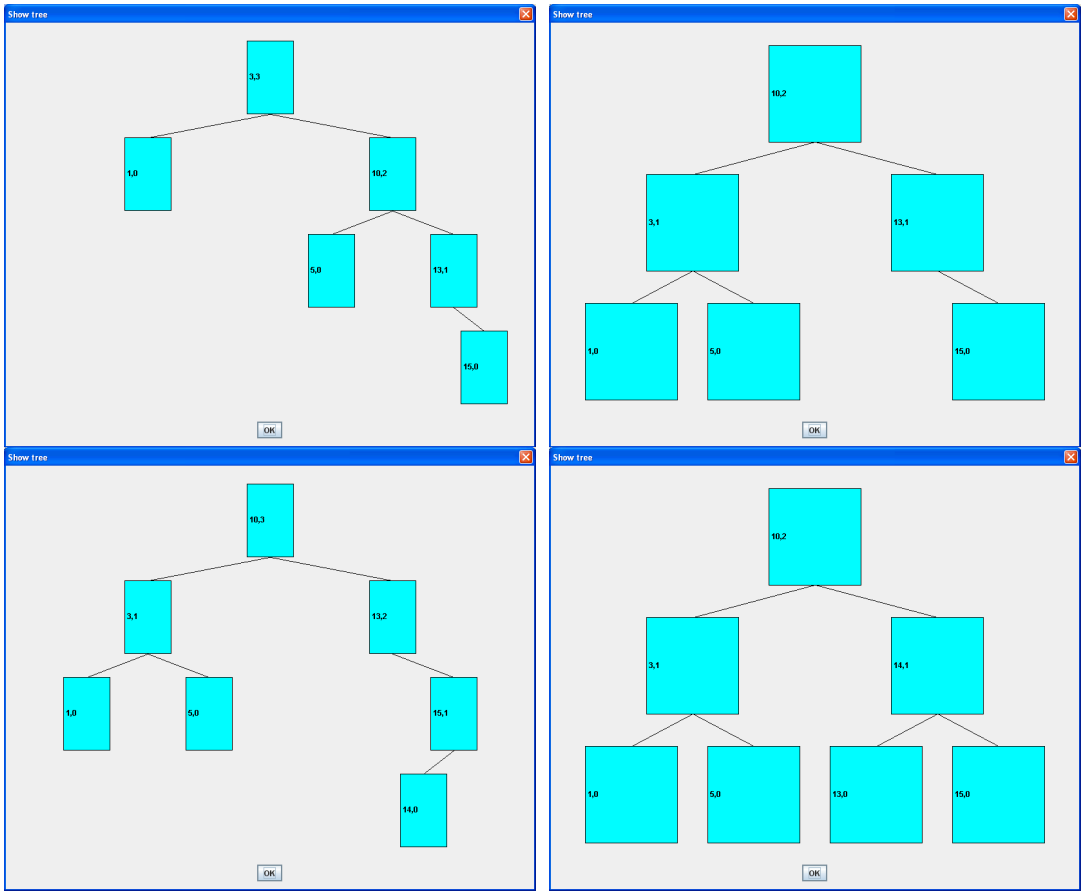
12a Komplettera metoden **balanceLeft** i klassen **Laboration12**. Metoden anropas då en nod är vänstertung, dvs då subträdet till vänster är två nivåer högre än subträdet till höger. Metoden ska genomföra en eller två rotationer, beroende på översta noden i subträdet till vänster.
I föreläsningsunderlaget ser du reglerna som gäller. Försök lösa problemet på egen hand. Om du inte lyckas kan du följa denna algoritm:

Om *height(node.left) - height(node.right)* är 2 så
  Om *heigth(node.left.left) – height(node.left.right)* är -1
    node.left tilldelas resultatet av *rotateLeft(node.left)*
    node tilldelas resultatet av *rotateRight(node)*
  Annars
    node tilldelas resultatet av *rotateRight(node)*
Returnera *node*

exercise12a ska ge sekvensen av figurer nedan då **balanceLeft** fungerar korrekt.



12b Komplettera metoden **balanceRight** i klassen **Laboration12**. Metoden anropas då en nod är högertung, dvs då subträdet till vänster är två nivåer lägre än subträdet till höger. Metoden ska genomföra en eller två rotationer, beroende på översta noden i subträdet till höger.
I föreläsningsunderlaget ser du reglerna som gäller. Försök lösa problemet på egen hand..
exercise12b ska ge sekvensen av figurer på nästa sida då **balanceRight** fungerar korrekt.



13 Klassen **AVLTree** i laboration12 är identisk med **BinarySearchTree** förutom att **AVLNode**-objekt används i stället för **BSTNode**-objekt. Med någon liten skillnad: Klassen innehåller färdiga skal till metoderna du arbetat med i uppgift 11 och 12. Metoden *balanceNode* är färdigskriven (väldigt enkel metod). Den kan man anropa om man inte är säker på vilken sida av noden som är tyngst (används i *remove*).

Nu ska du föra över dina fungerande lösningar på metoderna *rotateRigth*, *rotateLeft*, *balanceLeft* och *balanceRight*.

När du fört över metoderna ska du se till att balansering av trädet sker i *put* respektive *remove*-metoden:
I **put**-metoden ska balanseringen ske efter att en ny nod placerats. Och balanseringen ska ske i varje nod som passerats på väger ner till insättningen, fast nerifrån och upp. Det är på två ställen som balansering behöver ske, nämligen direkt efter de rekursiva anropen till *put*:

```
if(comparator.compare(key,node.key)<0) {
    node.left = put(node.left,key,value);
    node = balanceLeft(node);
} else if(comparator.compare(key,node.key)>0) {
    node.right = put(node.right,key,value);
    node = balanceRight(node);
}
```

I **remove**-metoden efter borttagningen av noden skett:

```
    :
    node = balanceNode( node );
    return node;
}
```

Nu ska du testa att AVL-trädet verkligen fungerar (om än ineffektivt).

14. Klassen **BinarySearchTree** innehåller metoden **print** vilken i sin tur anropar metoden **print(Node<K,V> node)**. Komplettera *print(Node)*-metoden med kod. Metoden ska skriva ut innehållet i ett träd i bokstavsordning, dvs *inorder*. Du kan följa algoritmen:

    ```
    om node inte är null
      rekursivt anrop med node.left som argument
      skriv ut key och value i noden
      rekursivt anrop med node.right som argument
    ```

    Testa din lösning genom att anropa metoden mot slutet av t.ex. exercise 1.

15. Komplettera klassen **BinarySearchTree** med metoden **printPreorder** vilken ska skriva ut innehållet i ett träd i preorder.

Extra

16. Komplettera klassen **BinarySearchTree** med metoden **printPostorder** vilken ska skriva ut innehållet i ett träd i postorder.

17. Komplettera klassen **BinarySearchTree** med metoden **printLevelOrder** vilken ska skriva ut innehållet i ett träd i nivå för nivå. Importera gärna *LinkedQueue* från paketet *laboration9* i din lösning.

18. Klassen **BinarySearchTree** innehåller metoden **public List<K> keys()**, vilken returnerar ett **List**-objekt (ArrayList i det här fallet).
    Meningen är att List-objektet som returneras ska innehålla samtliga nycklar som finns i trädet. Nycklarna ska vara ordnade i listan.
    Ovanstående metod anropar metoden
    private void keys(Node<K,V> node, ArrayList<K> list)
    Din uppgift är att komplettera metoden *keys(Node, List)* med kod vilken ska fylla ArrayList-objektet med value-objekt. Lösningen påminner till viss del om print-metoden i exercise 12.

19. Komplettera klassen **BinarySearchTree** med metoden **public List<V> values()** vilken ska returnera samtliga värden i trädet. Värdena ska ha samma ordning som nycklarna i uppgift 16. Returnera en LinkedList i din lösning.

20. Nu är det dags att effektivisera AVLTree. Och det handlar om att använda instansvariabeln *height* i AVLNode-objekten i stället för att i tid och otid beräkna nodens höjd.

    Först ska du ändra metoden **height(AVLNode<K,V> node)** så att metoden, liksom tidigare, returnerar -1 om *node* är *null*. Men om så inte är fallet ska *node.height* returneras.

    Sedan ska du ska skriva metoden **countHeight(AVLNode<K,V> node)** vilken ska beräkna en nods höjd. Det sker med uttrycket:

    ```
    Math.max( height(node.left) , height(node.right) ) + 1
    ```

    dvs 1 mer än största höjden av barnen. Lagra värdet i *node.height*.

    Slutligen ska du se till att *height* i trädets noder beräknas vid behov. Det behövs innan balanseringar, vid rotationer och dessutom vid borttagning av nod med två barn.

    - Lägg till anrop till **countHeight** på följande ställen (`countHeight(node);`):
      - I **put**-metoden innan anrop av balanserings-metoder
      - I **remove**-metoden före anropet av *balanceNode*. Men gör anropet endast om node!=null.
      - I **rotateRight** och i **rotateLeft** ska *countHeight* anropas för båda noderna som skiftat barn. Gör anropen mot slutet innan return-satsen.

- I **remove**: Lägg till `min.height = node.height;` efter anropet till *getMin*.

21. Komplettera klassen **BinarySearchTree** med metoden
    public void traverse(Action<V> action)
    vilken för varje värde i trädet ska anropa metoden **action(V value)** i Action-implementeringen. Interfacet *Action* är givet och ser ut så här:

```
public interface Action<V> {
    public void action( V value );
}
```

    Om du anropar metoden *traverse* med ett objekt av klassen *Print* (se nedan) på trädet i uppgift 4 får du resultatet:

```
15 17 27 36 38 44 45
```

```
public class Print<T> implements Action<T> {
    public void action( T value ) {
        System.out.print( value + " " );
    }
}
```

22. Skriv klassen **CollectValues<T>** vilken ska **implementera interfacet Action<T>**. Klassen ska dessutom innehålla metoden
    public List<T> getValues()
    vilken ska returnera ett List-objekt.
    Meningen med klassen är att samtliga värden i ett träd ska samlas i en lista. Genom att anropa metoden **getValues** får man ta del av listan med värden.

    Exempel:

```
BinarySearchTree<Integer,Integer> bst = new BinarySearchTree <
Integer,Integer >();
CollectValues<Integer> cv = new CollectValues<Integer>();

bst.put( new Integer(27), new Integer(27) );
bst.put( new Integer(15), new Integer(15) );
bst.put( new Integer(38), new Integer(38) );
bst.put( new Integer(17), new Integer(17) );
bst.put( new Integer(36), new Integer(36) );
bst.put( new Integer(45), new Integer(45) );
bst.put( new Integer(44), new Integer(44) );

bst.traverse( cv );
List<Integer> values = cv.getValues();
for(Integer i : values) {
    System.out.print( i + " " );
}
```

    Körresultat:

```
15 17 27 36 38 44 45
```

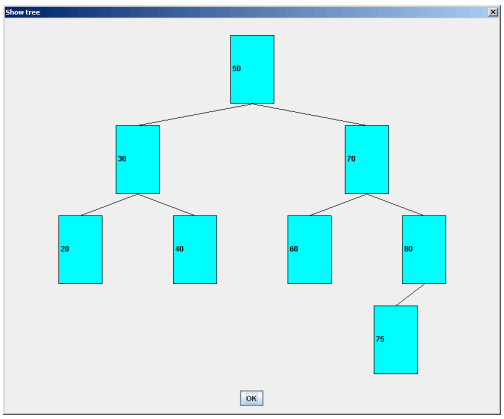
23. Skriv klassen **CollectEvenValues<T>** vilken ska fungera som CollectValues men med skillnaden att endast jämna värden (2, 4, 6, 8, …) samlas i listan.
    Om du ersätter CollectValues med CollectEvenValues i exemplet i uppgift 19 får du tuskriften:

```
36 38 44
```

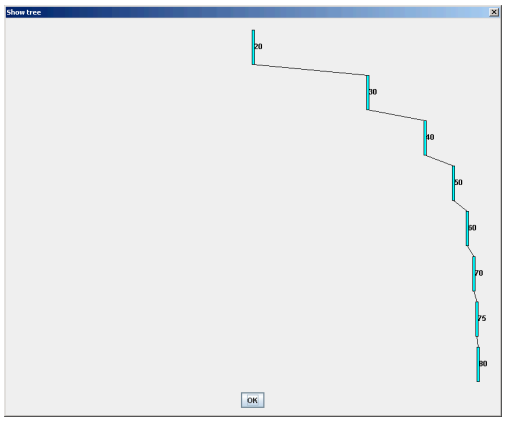
24. Komplettera klassen **BinarySearchTree** så att förenklad for-loop kan användas tillsammans med objekt av klassen.

Lösningar

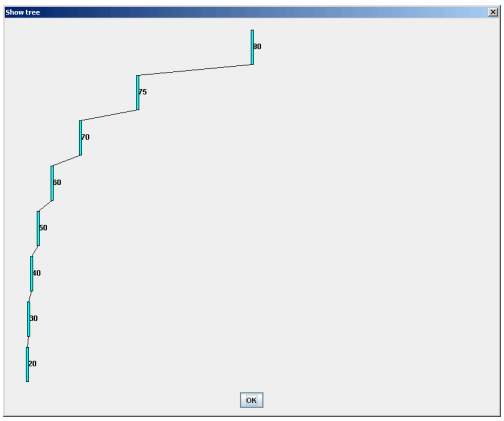
Uppgift 1



Uppgift 2



Uppgift 3



Uppgift 4

T.ex.

```
public void exercice4() {
    BinarySearchTree<Integer,Integer> bst = new BinarySearchTree
<Integer,Integer>();
    bst.put(new Integer(27), new Integer(27));
    bst.put(new Integer(15), new Integer(15));
    bst.put(new Integer(17), new Integer(17));
    bst.put(new Integer(38), new Integer(38));
    bst.put(new Integer(36), new Integer(36));
    bst.put(new Integer(45), new Integer(45));
    bst.put(new Integer(44), new Integer(44));
}
```

Uppgift 5

```
public void exercise5() {
    BinarySearchTree <String,Integer> bst = new BinarySearchTree
<String,Integer>();
    bst.put("F", new Integer(6));
    bst.put("B", new Integer(2));
    bst.put("C", new Integer(3));
    bst.put("I", new Integer(9));
    bst.put("D", new Integer(4));
    bst.put("H", new Integer(8));
    bst.put("G", new Integer(7));
    bst.root().showTree();
}
```

Uppgift 6

```
public void exercise6() {
    BinarySearchTree <String,String> tree = svenskEngelsk();
    String engelska, svenska = javax.swing.JOptionPane.showInputDialog( "Skriv
ett svenskt ord" );
    while( svenska!=null && !svenska.equals("slut") ) {
        engelska = tree.get(svenska);
        javax.swing.JOptionPane.showMessageDialog( null, svenska + " ger
resultatet " + engelska );
        svenska = javax.swing.JOptionPane.showInputDialog( "Skriv ett svenskt
ord" );
}
```

Uppgift 7

```
private TreeMap<String,String> capitals() {
    TreeMap<String,String> tree = new TreeMap<String,String>();
    tree.put("Sverige","Stockholm");
    tree.put("Danmark","Köpenhamn");
    tree.put("Norge","Oslo");
    tree.put("Finland","Helsingfors");
    tree.put("Polen","Warszawa");
    tree.put("Litauen","Villnius");
    tree.put("Estland","Talinn");
    tree.put("Lettland","Riga");
    tree.put("Tyskland","Berlin");
    tree.put("Ryssland","Moskva");
    return tree;
}

public void exercise 7() {
    TreeMap<String,String> tree = capitals();
    String capital, country = javax.swing.JOptionPane.showInputDialog( "Skriv
ett land" );
    while( country!=null && !country.equals("slut") ) {
        capital = (String)tree.get(country);
        if( capital!=null )
            javax.swing.JOptionPane.showMessageDialog( null, "Huvudstad i " +
country + " är " + capital );
        else
            javax.swing.JOptionPane.showMessageDialog( null, "Huvudstad i " +
country + " är okänd" );
        country = javax.swing.JOptionPane.showInputDialog( "Skriv ett land" );
    }
}
```

Uppgift 8a

```
public int size1() {
    if( tree == null )
        return 0;
    return tree.size();
}
```

Uppgift 8b

```
public int size2() {
    return size2( tree );
}

public int size2(BSTNode<V,K> node) {
    if( node == null)
        return 0;
    return 1 + size2(node.left) + size2(node.right);

}
```

Uppgift 8c

Skriv `size++;` i **put**-metoden i anslutning till att **BSTNode**-objekt skapats.
Skriv `size--;` efter anropet `remove(tree, key);` i **remove(K key)**

Uppgift 9

```
public V first() {
    BSTNode<K,V> node = tree;
    if( node== null )
        return null;
    while( node.left!=null ) {
        node = node.left;
    }
    return node.value;
}
```

Uppgift 10

```
public V last() {
    BSTNode<K,V> node = tree;
    if( node== null )
        return null;
    while( node.right!=null ) {
        node = node.right;
    }
    return node.value;
}
```

Uppgift 11a+b

```
private AVLNode<K,V> rotateLeft(AVLNode<K,V> node) {
    AVLNode<K,V> rootNode = node.right;
    node.right = rootNode.left;
    rootNode.left = node;
    return rootNode;
}

private AVLNode<K,V> rotateRight(AVLNode<K,V> node) {
    AVLNode<K,V> rootNode = node.left;
    node.left = rootNode.right;
    rootNode.right = node;
    return rootNode;
}
```

Uppgift 12a+b

```
private AVLNode<K,V> balanceLeft(AVLNode<K,V> node) {
    if (height(node.left) - height(node.right) == 2) {
        if (height(node.left.left) - height(node.left.right) == -1) {
            node.left = rotateLeft(node.left);
        }
        node = rotateRight(node);
    }
    return node;
}

private AVLNode<K,V> balanceRight(AVLNode<K,V> node) {
    if (height(node.left) - height(node.right) == -2) {
        if (height(node.right.left) - height(node.right.right) == 1) {
            node.right = rotateRight(node.right);
        }
        node = rotateLeft(node);
    }
    return node;
}
```

Uppgift 14

```
private void print( BSTNode<K,V> node ) {
    if( node!= null ) {
        print( node.left );
        System.out.println( node.key + ", " + node.value );
        print( node.right );
    }
}
```

Uppgift 15

```
public void printPreorder() {
    printPreorder( tree );
```

```
}
private void printPreorder(BSTNode<K,V> node ) {
    if( node!= null ) {
        System.out.println( node.key + ", " + node.value );
        printPreorder( node.left );
        printPreorder( node.right );
    }
}
```

Uppgift 16

```
public void printPostorder() {
    printPostorder( tree );
}
private void printPostorder(BSTNode<K,V> node ) {
    if( node!= null ) {
        printPostorder( node.left );
        printPostorder( node.right );
        System.out.println( node.key + ", " + node.value );
    }
}
```

Uppgift 17

```
public void printLevelOrder() {
    LinkedQueue<BSTNode<K,V>> queue = new LinkedQueue<BSTNode<K,V>>();
    BSTNode<K,V> node;
    queue.enqueue(tree);
    while(!queue.empty()) {
        node = queue.dequeue();
        System.out.print(node.value+" ");
        if(node.left!=null)
            queue.enqueue(node.left);
        if(node.right!=null)
            queue.enqueue(node.right);
    }
}
```

Uppgift 18

```
private void keys( BSTNode<K,V> node, ArrayList<K> list ) {
    if( node != null ) {
        keys( node.left, list );
        list.add( list.size(), node.key ); // list.addLast(node.key);
        keys( node.right, list );
    }
}
```

Uppgift 19

```
private void values( Node<K,V> node, LinkedList<V> list ) {
    if( node != null ) {
        values( node.left, list );
        list.add( list.size(), node.value ); // list.addLast(node.key);
        values( node.right, list );
    }
}

public List<V> values() {
    LinkedList<V> list = new LinkedList<V>();
    values( tree, list );
    return list;
}
```

Uppgift 20

```
private AVLNode<K,V> put(AVLNode<K,V> node, K key, V value) {
    :
        if(comparator.compare(key,node.key)<0) {
            node.left = put(node.left,key,value);
            countHeight(node);
            node = balanceLeft(node);
        } else if(comparator.compare(key,node.key)>0) {
            node.right = put(node.right,key,value);
            countHeight(node);
            node = balanceRight(node);
        }
    }
    return node;
}

private AVLNode<K,V> remove(AVLNode<K,V> node, K key) {
    :
    if(node!=null)
        countHeight(node);
    node = balanceNode(node);
    return node;
}

private AVLNode<K,V> rotateLeft(AVLNode<K,V> node) {
    AVLNode<K,V> rootNode = node.right;
    node.right = rootNode.left;
    rootNode.left = node;
    countHeight(node);
    countHeight(rootNode);
    return rootNode;
}

private AVLNode<K,V> rotateRight(AVLNode<K,V> node) {
    AVLNode<K,V> rootNode = node.left;
    node.left = rootNode.right;
    rootNode.right = node;
    countHeight(node);
    countHeight(rootNode);
    return rootNode;
}
```

Uppgift 21

```
private void traverse( BSTNode<K,V> node, Action<V> action ) {
    if( node!=null ) {
        traverse( node.left, action );
        action.action(node.value);
        traverse( node.right, action );
    }
}

public void traverse( Action<V> action ) {
    traverse( tree, action );
}
```

Uppgift 22

```
public class CollectValues implements Action<V> {
    private LinkedList<V> values = new LinkedList<V>();

    public List<V> getList() {
        return values;
    }

    public void action( V value ) {
        values.add( values.size(), obj);
    }
}
```

Uppgift 23

```
public class CollectEvenValues<T> extends CollectValues<T> {
    public void action(T obj) {
        if(obj instanceof Integer && ((Integer)obj) % 2 == 0)
            super.action(obj);
    }
}
```

Uppgift 24

```
public class BinarySearchTree<K,V> implements SearchTree<K,V>, Iterable<V> {
```