Problem1/1

```
    public boolean areMirror(BT<T> tree)
    {
     return areMirror(root,tree.root);
    }

    private boolean areMirror(BTNode<T> t1, BTNode<T> t2)
    {
        if (t1 == null && t2 == null)
            return true;

        if (t1 == null || t2 == null)
            return false;

        return  t1.data == t2.data && areMirror(t1.left,
t2.right) && areMirror(t1.right, t2.left);

    }
```

Problem1/2

```
    public void swap()
    {
     swap(root);
    }

    private void swap(BTNode<T> t)
    {
     T x;

     if (t != null)
     {
          if(t.left != null)
          {
               x = t.data;
               t.data = t.left.data;
               t.left.data = x;
          }
          else if(t.right != null)
          {
               x = t.data;
               t.data = t.right.data;
               t.right.data = x;
          }

          swap(t.left);
          swap(t.right);
     }
    }
```

Problem2/1

```
     public static <T> LinkedList<BTNode<T>> collectLeaves(BT<T>
bt)
     {
          LinkedList<BTNode<T>> l = new LinkedList<BTNode<T>>();
          LinkStack<BTNode<T>> nodes = new
LinkStack<BTNode<T>>();

          bt.find(Relative.Root);
          nodes.push(bt.root);

          while (! nodes.empty())
          {
               BTNode<T> current = nodes.pop();

               if (current.right != null)
                    nodes.push(current.right);

               if (current.left == null && current.right== null)
                    l.insert(current);

               if (current.left != null)
                    nodes.push(current.left);

          }

          return l;
     }
```

Problem2/2

```
   public LinkedList<T> collectLeaves()
    {
          LinkedList<T> l = new LinkedList<T>();
          return collectLeaves(root,l);
    }

    private LinkedList<T> collectLeaves(BTNode<T>
t,LinkedList<T> l)
    {
          if (t != null)
          {
          l = collectLeaves(t.left,l);

          if (t.left == null && t.right == null)
               l.insert(t.data);

          l = collectLeaves(t.right,l);
          }

          return l;
   }
```

Problem3/1

```
     public static boolean isBST(BT<Integer> bt)
     {
          LinkedList<Integer> list = new LinkedList<Integer>();
          rec_isBST(list, bt.root);

          boolean isBST = true;

          int min = Integer.MIN_VALUE;

          list.findFirst();

          while(! list.last())
          {
               if(min > list.retrieve())
               {
                    isBST = false;
                    break;
               }

               min = list.retrieve();
               list.findNext();
          }

          if(min > list.retrieve())
               isBST = false;

          return isBST;
     }

     private static void rec_isBST(LinkedList<Integer> l,
BTNode<Integer> n)
     {
          if(n == null)
               return;

          rec_isBST(l, n.left);
          l.insert(n.data);
          rec_isBST(l, n.right);
     }
```

Problem3/2

```
    public static boolean find(BT<Integer>bt , int k)
    {
        if (bt.empty())
            return false ;

        bt.find(Relative.Root);

        return recFind(bt,k);
    }

    private static boolean recFind(BT<Integer> bt, int k)
    {
        if (bt.retrieve() == k)
            return true;

        if (bt.retrieve() < k)
        {
            if (bt.find(Relative.RightChild))
                return recFind(bt,k);

            return false;
        }


        if (bt.find(Relative.LeftChild))
            return recFind(bt , k);

        return false;
    }
```

Problem4/1

```
  public void swapData(int k,int a)
   {
        swapData(k);
   }

   private void swapData(int k)
   {
        BSTNode<T> p = root;

        while(p != null && p.key != k)
        {
             if (k < p.key)
                  p = p.left;

             if(k > p.key)
                  p = p.right;
        }

        if(p != null)
        {
             if (p == root)
                  return;
             else
             {
                  LinkStack<BSTNode<T>> stack = new
LinkStack<BSTNode<T>>();
                  BSTNode<T> q = root;

                  while (q.right != p && q.left != p)
                  {
                       if (q.right != null)
                            stack.push(q.right);

                         if (q.left != null)
                              q = q.left;
                         else
                              q = stack.pop();
                    }

                    T x = p.data;
                    p.data = q.data;
                    q.data = x;
             }
        }
```

Problem4/2

```
  public int nbInRange(int k1, int k2)
   {
         return nbInRange(root, k1,k2);
   }

   private int nbInRange(BSTNode<T> t, int k1, int k2)
   {
         if (t == null)
              return 0;
         else if (k1 <= t.key && k2 >= t.key)
              return 1 + nbInRange(t.left, k1, k2) +
nbInRange(t.right, k1, k2);
         else
              return nbInRange(t.left, k1, k2) +
nbInRange(t.right, k1, k2);
   }
```

Problem4/3

```
   public int deepestKey()
   {
          int lvl1 = height(root.left);
          int lvl2 = height(root.right);

          if (lvl1 > lvl2)
               return rec_deepestKey(root.left, lvl1, 1).key;
          else if (lvl2 > lvl1)
               return rec_deepestKey(root.right, lvl2, 1).key;
          else
          {
               BSTNode<T> d1 = null;
               d1 = rec_deepestKey(root.left, lvl1, 1);

               BSTNode<T> d2 = null;
               d2 = rec_deepestKey(root.right, lvl2, 1);

               if(d1.key < d2.key)
                    return d1.key;
               else
                    return d2.key;
          }
   }
   private BSTNode<T> rec_deepestKey(BSTNode<T> t, int lvl, int 
depth)
   {
        if (t == null)
             return null;
        
        if(t.left == null && t.right == null)
        {
             if (depth == lvl)
             {
                    return new BSTNode<T>(t.key, t.data);
             }
        }
        
        if (t.right != null)
             return rec_deepestKey(t.right, lvl, depth + 1);
        else
             return rec_deepestKey(t.left, lvl, depth + 1);
     }

     private int height(BSTNode<T> t)
     {
          if (t == null)
               return 0;
          
          return (1 + Math.max(height(t.left), 
height(t.right)));
     }
```