Aufgabe 1

a) Für die unterschiedlichen Reihenfolgen müssen nur Ausgabe und die rekursiven Aufrufe vertauscht werden.

```
private void ausgabePreRek(BinaryTree<Kontakt> pb)
{
   Kontakt k;
   if (!pb.isEmpty())
   {
      k = pb.getContent();
      Console.println(k.getNachname() + ", " + k.getVorname());
      ausgabeInRek(pb.getLeftTree());
      ausgabeInRek(pb.getRightTree());
   }
}

private void ausgabeInRek(BinaryTree<Kontakt> pb)
{
   Kontakt k;
   if (!pb.isEmpty())
   {
      ausgabeInRek(pb.getLeftTree());
      k = pb.getContent();
      Console.println(k.getNachname() + ", " + k.getVorname());
      ausgabeInRek(pb.getRightTree());
   }
}

private void ausgabePostRek(BinaryTree<Kontakt> pb)
{
   Kontakt k;
   if (!pb.isEmpty())
   {
      ausgabeInRek(pb.getLeftTree());
      ausgabeInRek(pb.getRightTree());
      k = pb.getContent();
      Console.println(k.getNachname() + ", " + k.getVorname());
   }
}

// Wrapper-Methode
public void namenAusgeben()
{
   ausgabeInRek(wurzel);
}
```

Für die Ausgabe der Namen sollte die Inorder-Reihenfolge gewählt werden. Der Suchbaum ist alphabetisch sortiert, d.h. alle Knoten des linken Teilbaums liege alphabetisch VOR dem aktuellen Knoten, alle Knoten des rechten Teilbaums DANACH. Die Inorder-Reihenfolge funktioniert genau so: erst alle Knoten des linken Teilbaums ausgeben, dann den Knoten selbst, und dann alle Knoten des rechten Teilbaums.

b)

```
private void listeSpRek(BinaryTree<Kontakt> pb, List<Kontakt> pl)
{
   if (!pb.isEmpty())
   {
      pl.append(pb.getContent());
      listeSpRek(pb.getLeftTree(), pl);
      listeSpRek(pb.getRightTree(), pl);
   }
}

// Wrapper-Methode
public List<Kontakt> listeSpeichern()
{
   List<Kontakt> liste = new List();
   listeSpRek(wurzel, liste);
   return liste;
}
```

Der Suchbaum sollte Preorder-Reihenfolge traversiert werden, um die Liste zu füllen. Das bedeutet: Zuerst die Wurzel, dann der oberste Knoten des linken Teilbaums (und der ganze linke Teilbaum), dann der oberste Knoten des rechten Teilbaums (und der ganze recht Teilbaum).
Auf diese Weise kann man später genau den gleichen Suchbaum aufbauen, indem man die Kontakte einfach der Reihe nach in einen leeren Suchbaum einfügt.

c)

```
public void listeLaden(List<Kontakt> pl)
{
   wurzel = new BinaryTree();   // löscht bestehende Kontakte
   pl.toFirst();
   while (pl.hasAccess())
   {
      einfügen(pl.getContent());
      pl.next();
   }
}
```

d)

```
private int zählenRek(BinaryTree<Kontakt> pb)
{
   if (!pb.isEmpty())
   {
      return zählenRek(pb.getLeftTree())
           + 1
           + zählenRek(pb.getRightTree());
   }
   else return 0;
}

public int getAnzahl()
{
   return zählenRek(wurzel);
}
```

Aufgabe 2

a) Blätter zählen

```
// Gibt die Anzahl der Blätter des Binärbaums zurück
private int zähleBlätterRek(BinaryTree<Adresse> knoten)
{
   // Falls der Knoten leer ist, beende die Rekursion.
   // Dieser Fall kommt vor, wenn der Baum ganz leer ist,
   // oder der vorige Knoten nur einen Nachfolger hat.
   if (knoten.isEmpty())
   {
      return 0;
   }
   // Falls der Knoten ein Blatt ist, zähle 1 u. beende die Rekursion.
   if (knoten.getLeftTree().isEmpty() && knoten.getLeftTree().isEmpty())
   {
      return 1;
   }
   // Sonst gehe nach links u. rechts in die Rekursion.
   return
       zähleBlätterRek(knoten.getLeftTree())
     + zähleBlätterRek(knoten.getRightTree());
}
```

b) Maximale Tiefe

```
// Hilfsmethode: gibt das Maximum von a und b zurück
private int max(int a, int b)
{
   if (a > b) return a;
   else       return b;
}

// Gibt die maximale Tiefe des Binärbaums zurück.
private int maxTiefeRek(BinaryTree<Adresse> knoten)
{
   if (knoten.isEmpty())
   {
      return 0;
   }
   else
   {
      return max(maxTiefeRek(knoten.getLeftTree()),
                 maxTiefeRek(knoten.getRightTree())) + 1;
   }
}
```