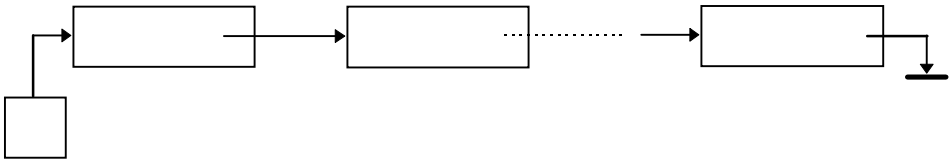
9 Dynamische Datenstrukturen

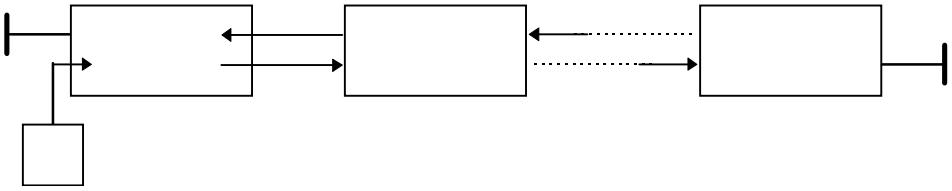
Datenstrukturen, die einmal in einer fixen Größe erzeugt werden, nennt man statische Datenstrukturen oder Datentypen. Ein Array, das mit einer fixen Größe erzeugt wird, ist ein Beispiel für eine statische Struktur. Ändert sich die Größe oder die Form einer Struktur während der Ausführung so spricht man von dynamischen Datenstrukturen. Das .NET Framework bietet eine Vielzahl von dynamischen Typen als oft fertige Klassen zur Verwendung an. Das sind vor allem eine Fülle von verschiedenen *Collections* und z.B. bei den Steuerelementen die `TreeView`. Trotzdem lohnt es sich, sich mit dynamischen Datenstrukturen auf einer grundlegenden Ebene zu beschäftigen. Beim objektorientierten Programmieren ist die Erzeugung von Objekten ja eine ständig angewandte Programmiertechnik. Für den Schritt zu dynamischen Strukturen fehlt nur noch die Verbindung der Objekte.

9.1 Wichtige dynamische Datenstrukturen

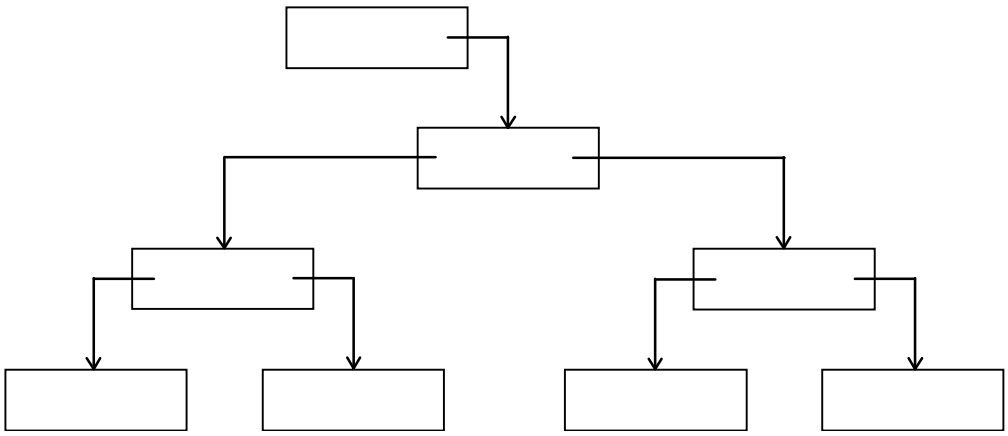
Einfach verkettete Liste:



Zweifach verkette Liste:



Binärbaum:



Um solche Strukturen aufbauen zu können benötigen die einzelnen Elemente neben der eigentlichen Information Referenzen (Zeiger) auf ein oder mehrere weitere Elemente. Die Information kann z.B. eine Zahl, eine Zeichenkette usw. sein.

Dadurch ergeben sich Klassen, die Verweise auf ein Objekt des selben Typs als Datenfelder enthalten. Dadurch ergibt sich eine Analogie zu rekursiven Methoden.

Klasse für ein Element einer einfach verketteten Liste:

```
class ListNode
{
   string content;
   ListNode next;
   .....
}
```

Ist `listNode` ein Objekt des Typs `ListNode`, so erzeugt man sich einen Nachfolger des Elements mit den Anweisungen:

```
listNode.Next = new ListNode("Inhalt");
```

Dabei wird ein Konstruktor verwendet, der den Inhalt festlegt und die Referenz auf den nächsten Knoten `null` setzt.

Klasse für ein Element eines binären Baumes:

```
class Node
{
   string content;
   Node left;
   Node right;
   .....
}
```

Klasse für ein Element eines Baumes mit einer beliebigen Anzahl von Kindknoten eines Knotens.

```
class Node
{
   string content;
   ArrayList nodes;
   .....
}
```

9.2 Einfach verkettete Liste

Eine einfach verkettete Liste stellt die für eine komfortable Collection üblichen Methoden zur Verfügung. Das sind genau jene Aufgaben die man für dynamische Strukturen zu lösen hat: Hinzufügen eines neuen Elements am Ende, einfügen an einer bestimmten Stelle, löschen eines Elements. Die Collections im Framework stellen das natürlich alles zur Verfügung, aber das folgende Beispiel soll einen Eindruck davon vermitteln, was da dahinter steckt.

```
public class ListNode
{
    string content;
    ListNode next = null;

    // + properties and default constructor
    public ListNode(string content)
    {
       this.content = content;
    }

}

public class LinkedList
{
    ListNode root;
    int count = 0;

    public ListNode Root
    {
       get { return root; }
    }

    public int Count
    {
       get { return count; }
    }

    public LinkedList()
    {
    }


    // Indexer to get the node at a given index
    public ListNode this[int index]
    {
       get
       {
          if (index < 0)
             return null;
          ListNode node = root;
          for (int i = 0; i < index && node != null; i++)
          {
             node = node.Next;
          }
          return node;
       }
    }

    public ListNode Add(string content)
    {
       ListNode newNode = new ListNode(content);
       if (count == 0)
          root = newNode;
       else
       {
          ListNode lastNode = this[count-1];
          lastNode.Next = newNode;
       }
       count++;
       return newNode;
    }
```

```
    public void RemoveAt(int index)
    {
       if (index == 0)
       {
          root = root.Next;
       }
       else
       {
          ListNode node = this[index-1];   // get the node before
          node.Next = node.Next.Next;
       }
       count--;
    }
       public ListNode Insert(int index, string content)
    {
       ListNode node = this[index];
       if (node == null)      // list has length 0
       {
          root = new ListNode(content);
          count++;
          return root;
       }
       // insert a node after the node at position index
       // with the content of the old content at position index
       // set the new content as content of node
       ListNode newNode = new ListNode(node.Content);
       newNode.Next = node.Next;
       node.Next = newNode;
       node.Content = content;
       count++;
       return node;
    }

    public string[] ToStringArray()
    {
       string [] elements = new string[count];
       ListNode node = root;
       int i = 0;
       while (node != null)
       {
          elements[i++] = node.Content;
          node = node.Next;
       }
       return elements;
    }

}
```

Der Code zeigt, wie man diese Grundaufgaben lösen kann. Grundsätzlich gibt es einen Unterschied ob man z.B. einen Knoten nach einem aktuellen Knoten einfügen soll oder vor diesem Knoten, oder ob man den aktuellen Knoten löschen soll oder den Nachfolger. Problematisch ist immer, dass man wegen der Verkettung in nur einer Richtung den Vorgänger nicht verfügbar hat.

Einfügen nach einem Knoten und den Nachfolger löschen ist einfach. Es ist eine Besonderheit, dass man in C# wegen des Garbage-Collectors Objekte nicht explizit löschen muss. Sobald es auf Objekte keine Referenzen mehr gibt, werden sie früher oder später automatisch gelöscht.

Für das Einfügen vor einem Knoten k gibt es einen Trick. Man fügt einen neuen Knoten nach dem Knoten k ein und tauscht die Inhalte des Knotens k mit dem Inhalt des neuen Knotens.

9.3 TreeView

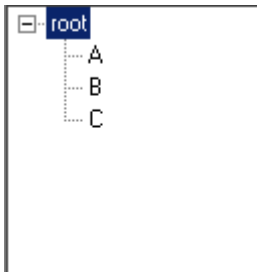
Als Beispiel für einen Baum betrachten wir einige Eigenschaften und Methoden der TreeView Klasse. Die TreeView wird immer dann verwendet, wenn man Baumstrukturen darstellen will und die Auswahl von Elementen in einem UI ermöglichen will. Dazu muss man die TreeView mit einer entsprechenden Datenstruktur verbinden. Ein typisches Beispiel für eine solche Datenstruktur ist der Verzeichnisbaum der Dateien. Aber auch für sich allein zeigt uns eine TreeView eine Vielzahl von typischen Operationen auf einem Baum.

9.3.1 Überblick

Die TreeView Property `Nodes` liefert eine Liste der Knoten (*nodes*) in der obersten Ebene. Einen Wurzelknoten vom Typ `TreeNode` gibt es nicht. Die TreeView selber ist quasi der Wurzelknoten. Wie bei jeder Collection kann man mit `Add` neue Knoten hinzufügen. Natürlich kann man auch Knoten einfügen und löschen. Jeder Knoten vom Typ `TreeNode` hat wieder eine Collection von Knoten, den Kindknoten. Zunächst ist eine TreeView einfach leer. Anmerkung: Man kann schon im Designer eine `Nodes` Collection erzeugen, aber das macht in der Regel keinen Sinn. Die folgenden Zeilen zeigen, wie man einen Knoten in der ersten Ebene einfügt und diesem Knoten dann drei Kindknoten hinzufügt.

```
// einen Knoten tn erzeugen und in der ersten Ebene einfügen
TreeNode tn;
tn = new TreeNode("root");
treeView.Nodes.Add(tn);
// Knoten erzeugen und als Kindknoten des Knotens tn einfügen
TreeNode child;
child = new TreeNode("A");
tn.Nodes.Add(child);
// TreeNode mit Standardkonstruktor erzeugen
child = new TreeNode();
child.Text = "B";
tn.Nodes.Add(child);
// weitere Variante
tn.Nodes.Add(new TreeNode("C"));
treeView.ExpandAll();
```

`ExpandAll` expandiert alle Knoten und zeigt die TreeView mit den erzeugten Knoten.



Details des Aussehens einer TreeView werden über Properties eingestellt, die selbsterklärend sind. Die Property `Nodes.Count` ist die Anzahl der Knoten in der ersten Ebene (hier 1) und nicht die Gesamtzahl der Knoten.

Das wohl wichtigste Ereignis der TreeView ist das Event `AfterSelect`. Wird es abonniert, so wird folgende Ereignisprozedur verwendet:

```
void treeView_AfterSelect(object sender, TreeViewEventArgs e)
{
   selectedNode = e.Node;
   lblNodeInfo.Text =
      String.Format("Current node: {0}", selectedNode.Text);
```

```
}
```

Das Argument e enthält eine Referenz auf den ausgewählten Knoten. Der Beispielcode verwendet eine Variable `selectedNode`, um sich den jeweils selektierten Knoten zu merken und kopiert den Text des Knotens in ein Label.

Üblicherweise wird vor den Knoten ein kleines Bild angezeigt. Will man eine TreeView mit diesen Bildern, so muss man zunächst Bitmaps (16 x 16) erzeugen.

Project → Add New Item → Resources → Bitmap File

Üblicherweise ändert sich das Bild für selektierte Knoten, d.h. man benötigt für jeden Knotentyp, den man unterschiedlich darstellen will, zwei Bilder. Zwei weitere Bilder sollte man für jene Knoten reservieren, denen man nicht explizit ein Bild zugeordnet hat.

Diese Bilder fügt man in eine ImageList ein. Es bewährt sich die Bilder so einzufügen, dass jeweils das Bild für den selektierten Knoten gleich anschließend an das Bild für den entsprechenden nicht selektierten Knoten kommt. Hinweis: Wenn Sie die Bilder ändern, so wird diese Änderung erst wirksam, nachdem man das Bild in der ImageList gelöscht und wieder neu eingefügt hat. Am Ende dieser Vorbereitungen haben sie eine Liste von Bildern, die man über einen fortlaufenden Index auswählen kann.

Die Standard Bilder für die Knoten legen zwei TreeView Eigenschaften fest:

```
treeView.ImageIndex = 0;
treeView.SelectedImageIndex = 1;
```

Für einen Knoten kann diese Einstellung geändert werden:

```
selectedNode.ImageIndex = 2;
selectedNode.SelectedImageIndex = 3;
```

9.3.2 Elementare Operationen

Wir wollen die TreeView nützen, um einige elementare Operationen auf Bäumen zu zeigen. Da gibt es z.B. die Aufgabe, die Knoten eines Baumes von der Wurzel bis zu den Endknoten, den Blättern (*leaves*) zu "besuchen". Besuchen steht hier, für die Ausführung einer beliebigen Operation auf diesem Knoten. Es ist nahe liegend, dafür die Technik der Rekursion zu nutzen.

```
void ProcessTreeInPreOrder(TreeNode node)
{
   textOutput.Text += node.Text + "\r\n";
   foreach (TreeNode tn in node.Nodes)
   {
      ProcessTreeInPreOrder(tn);
   }
}
```

Wird diese Methode für einen Knoten aufgerufen, so wird zuerst die gewünschte Operation auf diesem Knoten ausgeführt. Anschließend wird die Methode rekursiv für die Kindknoten aufgerufen.

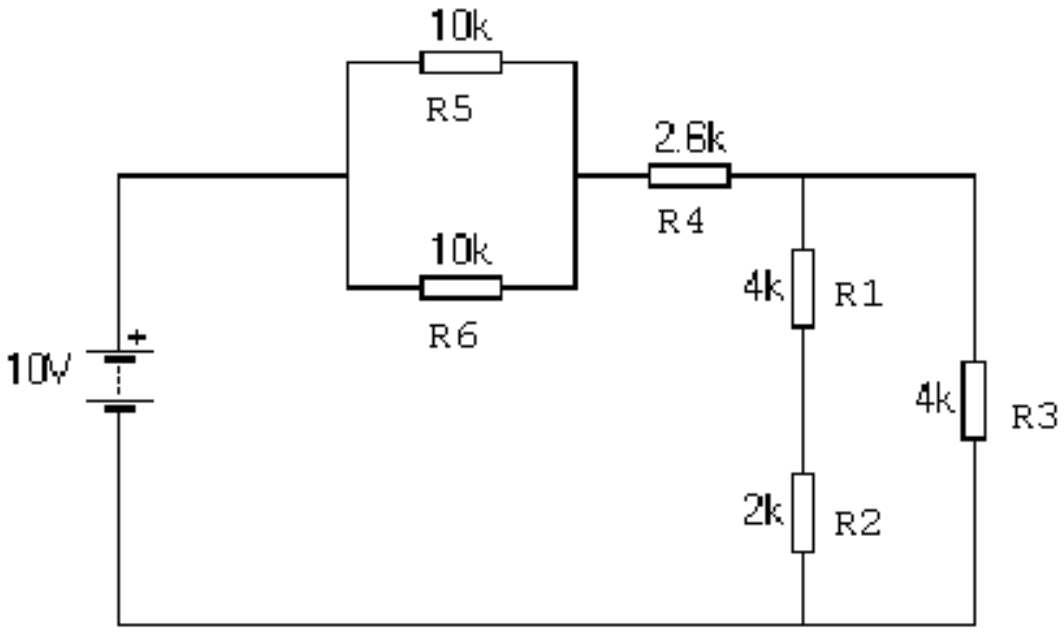
Der umgekehrte Fall ist, die Knoten eines Baumes in umgekehrter Richtung zu besuchen, also von den Blättern zur Wurzel zu wandern:

```
void ProcessTreeInPostOrder(TreeNode node)
{
   foreach (TreeNode tn in node.Nodes)
   {
      ProcessTreeInPostOrder(tn);
   }
   textOutput.Text += node.Text + "\r\n";
}
```

Hier ist die gewünschte Operation auf den einzelnen Knoten am Ende der Methode anzugeben. Dadurch wird die Operation erst ausgeführt, wenn ein Knoten keine Kindknoten mehr hat. Die Rekursion sorgt dafür, dass mit diesen wenigen Programmzeilen alle Knoten besucht werden.

9.3.3 Kontext Menü

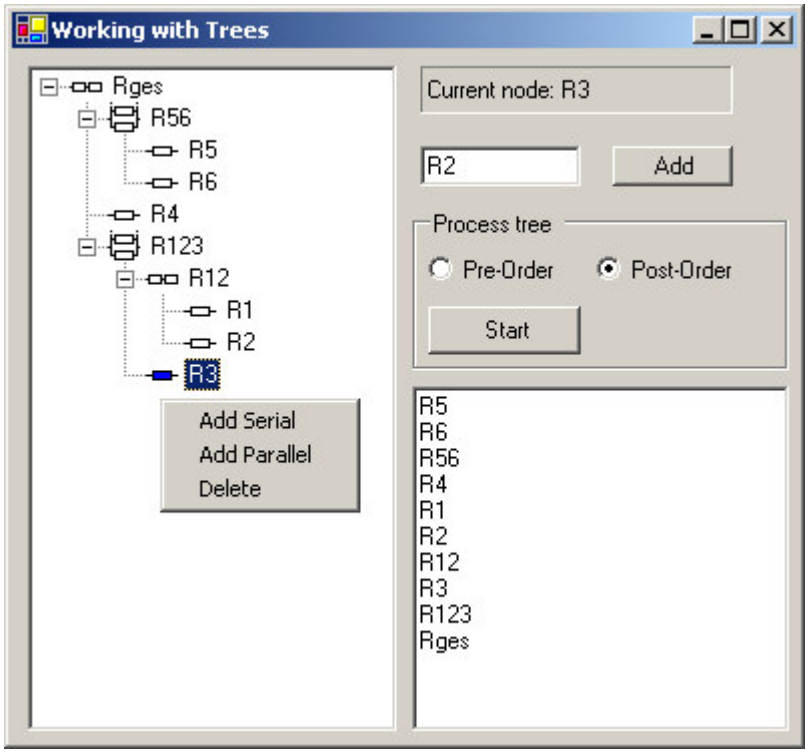
Kontext Menüs sind aus einem typischen UI nicht mehr wegzudenken. Ein Klicks mit der rechten Maustaste öffnet ein Menü mit Kommandos, die zum jeweiligen Kontext passen. Kontext Menüs kann man sich mit dem Designer leicht erstellen und dann einem Control zuordnen. Um die wichtigsten Dinge für die Verwendung einer TreeView zu demonstrieren, verwenden wir ein Kontext Menü, das es erlaubt, dem selektierten Knoten Kindknoten hinzuzufügen. Der Hintergrund für diese Arbeit mit einer TreeView ist, dass man den Rechengang, um die Ströme und Spannungen in einer Schaltung aus Widerständen mit zwei Anschlussklemmen in einem Baum darstellen kann.



Der Rechengang um den Gesamtwiderstand für diese Schaltung auszurechnen ist:

```
R12 = R1 + R2
R123 = R12 // R3
R56 = R5 // R6
Rges = R56 + R4 + R123
```

Die Darstellung in einem Baum ist:



Das Programm protokolliert auch die Möglichkeiten, einen Baum von der Wurzel zu den Blättern und umgekehrt abzuarbeiten. Was hier für Post-Order angezeigt wird, ist die notwendige Reihenfolge der Berechungen:

```
R56 = R5 // R6
R4
R12 = R1 + R2
R3
R123 = R12 // R3
Rges = R56 + R4 + R123
```

9.4 Zweipolbaum

Im Gegensatz zu dem bisherigen Beispiel ist eine TreeView in der Praxis nur die Darstellung einer Datenstruktur, die für sich existieren muss. Wir wollen nun eine Datenstruktur bilden, welche alle Informationen enthält, um die oben angedeutete Berechung auch tatsächlich auszuführen und die TreeView für deren Darstellung verwenden.

Ein Knoten in diesem Zweipolbaum steht jeweils für einen Zweipol. Ist es ein Endknoten, so ist es ein einzelner Widerstand. Die anderen Knoten sind jeweils die Zusammenfassung einer Serien- oder Parallelschaltung von Kindknoten. Für jeden Knoten benötige wir folgende Daten:
Name, Widerstand, Schaltung der Kindknoten, Liste der Kindknoten. Für die spätere Berechung von Strom und Spannung nehmen wir Strom und Spannung gleich dazu.

Das ergibt folgende Klassendefinition:

```
public class ZNode
{
   string name;
   double z;
   double voltage;
   double current;
   Connection connectionType;
   ArrayList childNodes;

   // Properties for the data fields

   // Methods
}
```

Für `connectionType` verwenden wir eine Enumeration:

```
public enum Connection { None, Serial, Parallel }
```

Eine wichtige Methode ist die Methode zur Berechnung des Widerstandes eines Knotes. Dafür müssen wir eine rekursive Methode schreiben, die nach dem Post-Order Prinzip die Berechnung mit den Blättern beginnt und sich in Richtung Wurzel bewegt.

```
public double CalcZ()
{
   double result = 0.0;
   int i;
   switch (connectionType)
   {
      case Connection.None:
         result = z;
         break;
      case Connection.Serial:
         for (i = 0; i < childNodes.Count; i++)
            result += ((ZNode) childNodes[i]).CalcZ();
         break;
      case Connection.Parallel:
         for (i = 0; i < childNodes.Count; i++)
            result += 1.0 / ((ZNode) childNodes[i]).CalcZ();
         result = 1.0 / result;
         break;
   }
   return result;
}
```

9.4.1 Daten mit TreeView verbinden

Im Grunde müssen wir nur für jeden Knoten der TreeView wissen, welchen Knoten des Zweipolbaumes er darstellt. Meist genügt eine Referenz auf das entsprechende Objekt. Für diese Referenz ist die Property `Tag` eines `TreeNode` vorgesehen. Momentan erzeugen wir den Zweipolbaum über die TreeView. Die wesentlichen Zeilen im Quellcode dafür sind:

```
private void mnuAddSerial_Click(object sender, System.EventArgs e)
{
   ZNode z = new ZNode(editNewValue.Text, 4.0);
   docNode.AddChild(z);
   docNode.ConnectionType = Connection.Serial;
   TreeNode tn = new TreeNode(editNewValue.Text);
   tn.Tag = z;
   node.ImageIndex = 2;
   node.SelectedImageIndex = 3;
   node.Nodes.Add(tn);
   UpdateView();
}
```

Zuerst wir der Knoten im Dokument erzeugt, dann der Knoten in der TreeView und der Dokumentknoten `z` mit dem TreeView-Knoten verbunden. `UpdateView` berechnet den Widerstand für den Knoten neu und zeigt das Resultat an.

Analog schaut der Code für das Hinzufügen eines Widerstandes in einer Parallelschaltung aus.

Nach der Selektion eines neuen Knotens müssen wir die Referenzen auf den aktuellen Knoten der TreeView und den aktuellen Dokumentknoten auf den aktuellen Stand bringen.

```
private void treeView_AfterSelect(object sender, TreeViewEventArgs e)
{
   node = e.Node;
   docNode = (ZNode) node.Tag;
   UpdateView();
}
```