Teil I

OOP - Objektorientierte Programmierung

Inhalt

Das Abspeichern von Daten in einem Array ist möglich, aber mit einigen Einschränkungen verbunden. Das Array muss beim Design des Programms festgelegt werden. Eine Größenänderung ist nachträglich nicht möglich[1]
Eine verkettete Liste bietet die Möglichkeit beliebig viele Einträge zu verwalten. Alle Einträge werden als Pointer auf dem Heap abgelegt. Die physikalische Reihenfolge ist für die logische Reihenfolge unerheblich.
In der Liste sind die Daten abgespeichert. Zusätzlich verweist ein Eintrag der Liste auf den nächsten Eintrag. Der letzte Eintrag hat als Nachfolger (nächster Eintrag) `nullptr` eingetragen.
Um die Liste zu verwalten, benötige ich (mindestens) zwei Klassen. Eine Klasse (im Beispiel `ModelData`) verwaltet die Liste. Sie ist auch die Schnittstelle im Programm zu den anderen Klassen (wenn vorhanden). `Entry` soll die Klasse sein, die die Daten beinhaltet. Diese Klasse erhält zusätzlich zu den Attributen der Daten (siehe Bild 1).

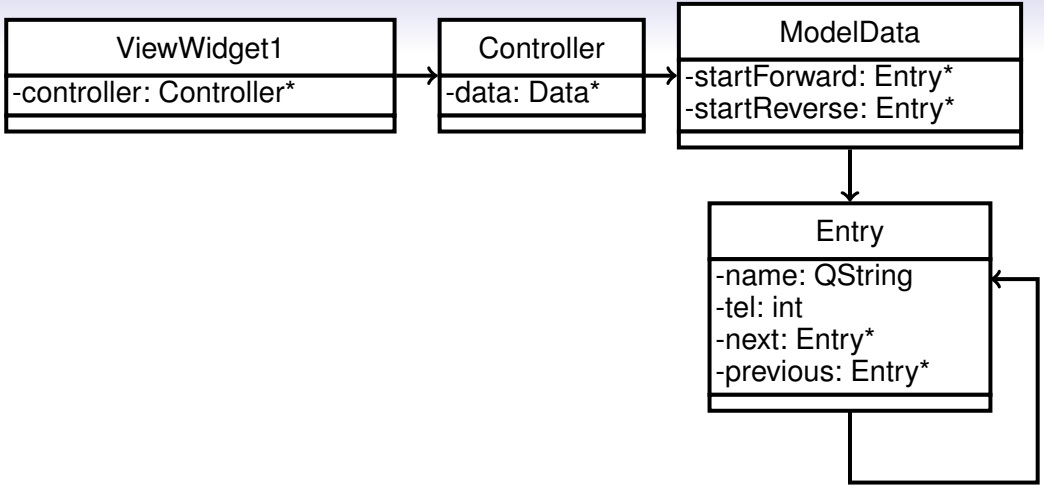

Abbildung: Klassendiagramm zu Programm, das die verkettete Liste verwaltet

Bild 2 zeigt das Programm zur Laufzeit in Form eines sogenannten

Objektdiagramms. Hier mit einer leeren Liste - ohne Objekte zur Ablage von Daten in der Liste. Jedes Rechteck stellt ein Objekt dar. Der Name des Objekts steht in der ersten Zeile vor dem ‚: ' dahinter folgt der Name der Klasse. Im Diagramm sind Objekte der Liste als Zahlen (1, 2, 3, . . .) eingetragen. Unterhalb des Namens des Objekts stehen die Attribute mit dem aktuellen Wert. Das Objektdiagramm zeigt einen Zustand der Attribute und Objekte zu einem bestimmten Zeitpunkt.

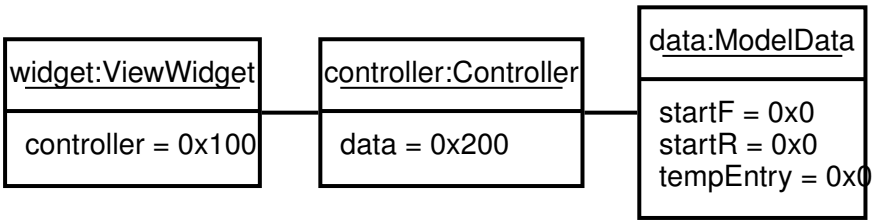

Abbildung: Objektdiagramm, leere Liste

In Bild 3 sind in der Liste drei Einträge vorhanden. Das Attribut `startF` stellt den Beginn der Liste dar. Die Adressen hier sind fiktiv. `startR` ist ein Zeiger auf das

Ende der Liste. Dieser ist befüllt und wird nur bei einer doppelt verketteten Liste benötigt. Das Attribut `next` in den Objekten von `Entry` beinhaltet die Adresse, unter der das nachfolgende Element der Liste auf dem Heap gespeichert ist. Wenn kein Element folgt (hier bei Objekt 3) wird für `next` der Wert `nullptr` ($0x00^2$) gespeichert.

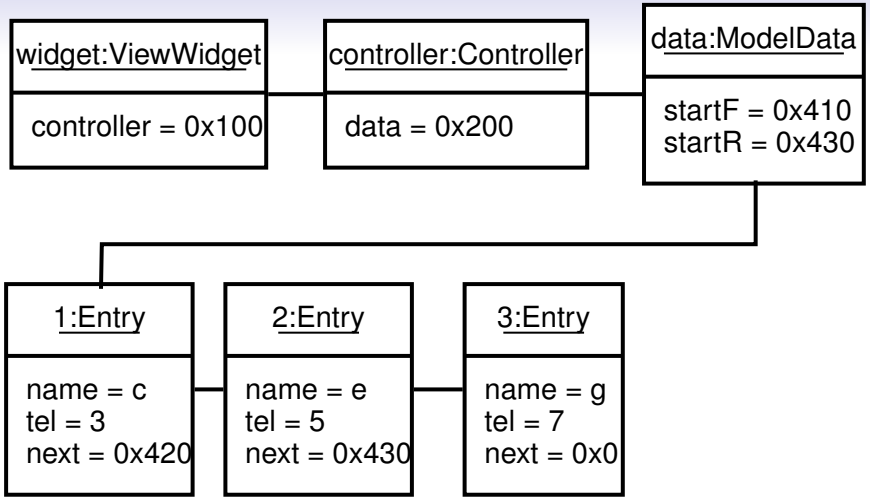

Abbildung: Objektdiagramm, drei Einträge in verkettete Liste

In den folgenden Abschnitten betrachte ich das Einfügen in eine Liste, die in einer Richtung verkettet ist. Diese Form wird einfach verkettete Liste genannt. Einfach bezieht sich dabei auf die mögliche Richtung der Navigation.
In späteren Abschnitten befasse ich mich dann mit einer sogenannten doppelt verketteten Liste. Diese Form kann in zwei Richtungen (von vorne nach hinten und von hinten nach vorne durchlaufen werden).
Wenn neue Einträge hinzugefügt werden sollen, müsste man bei einem Array alle Einträge, die nach der entsprechenden Position folgen, um eine Stelle nach rechts verschieben. Dies würde durch Kopieren der Daten erfolgen (vom Ende rückwärts bis zur betroffenen Stelle). Anschließend kann in dem freien Feld das Datum eingetragen werden.
Bei einer Liste wird ein neues Objekt mit `new Entry` angelegt (Bild 4, Listing 1).

```
  void ModelData::insertNewEntryFront(QString nameIn, QString tel
2 {
    Entry * newEntry = new Entry;

4
```

```
    newEntry->setName(nameIn);
6   newEntry->setTelNr(telIn);
```

Listing 1: Einfügen am Anfang (Teil 1)

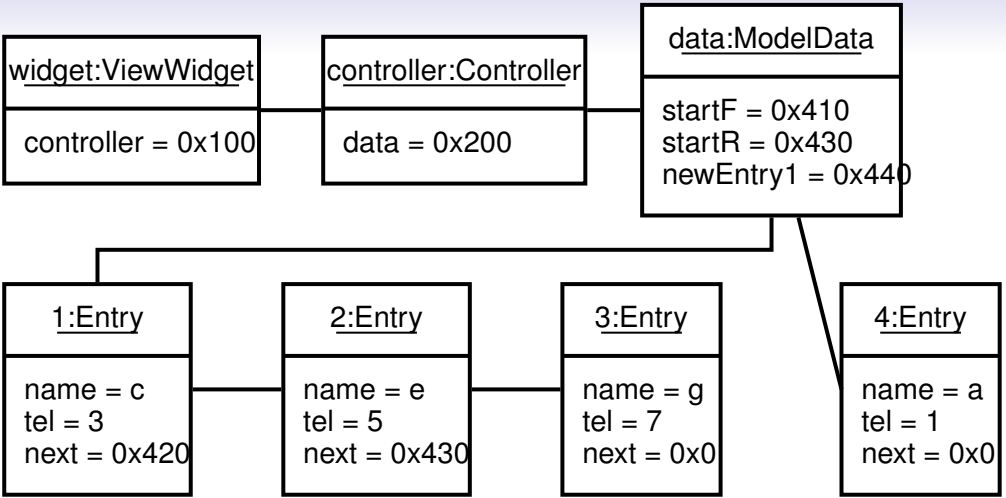

Abbildung: Objektdiagramm, ein Objekt vorne Einfügen, neues Objekt

Anschließend werden (Reihenfolge der Arbeitsschritte beachten!) die Zeiger aktualisiert (Bilder 5 und 6). In Bild 5 ist der Nachfolger von Objekt 4 auf Objekt 1 verlinkt. Somit kann ich das Objekt 1 über `startF` und zusätzlich über den temporären Zeiger `newEntry` erreichen.
Im Listing 2 prüfe ich in Zeile 1, ob es eine Liste gibt. Falls nicht, wird nur der Zeiger `startF` verändert (Zeile 3). Der else-Zweig ist in der ausführlichen Fassung bei einer einfach verketteten List nicht nötig. Ich habe den Code aus einem Programmbeispiel entnommen, das doppelt verkettet ist (vorwärts und rückwärts).

```
    if(nullptr == fListStart)
2   {
      fListStart = newEntry;
4   }
    else
6   {
      if(nullptr != fListStart)
```

```
8         {
            newEntry->setNext(fListStart);
10          fListStart = newEntry;
            newEntry = nullptr;
12        }
        }
14    }
```

Listing 2: Einfügen am Anfang (Teil 2)

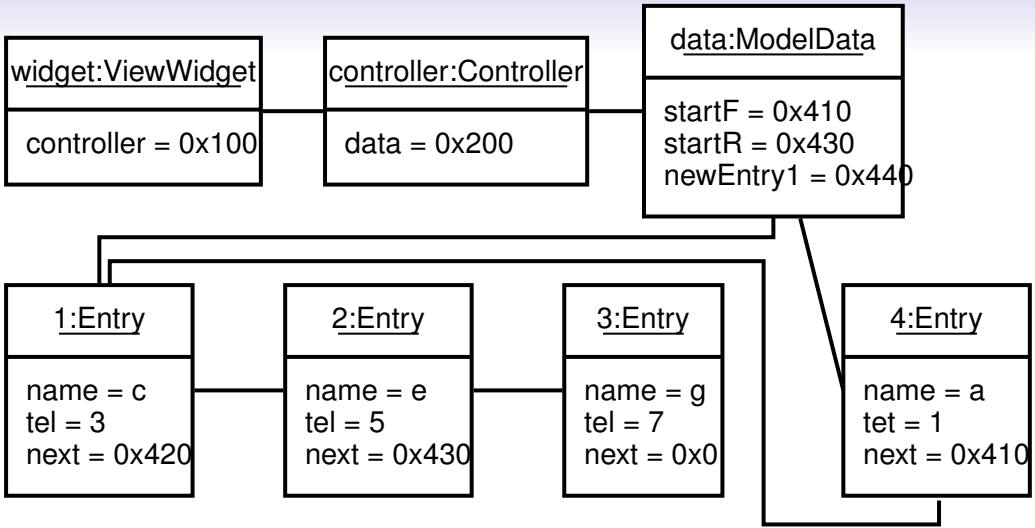

Abbildung: Objektdiagramm, ein Objekt vorne Einfügen, neues Objekt

In Bild 6 ist dann der Zeiger `startF` angepasst, der `newEntry` ist wieder auf 0x00 gesetzt. Damit ist das Einfügen am Anfang der Liste abgeschlossen.

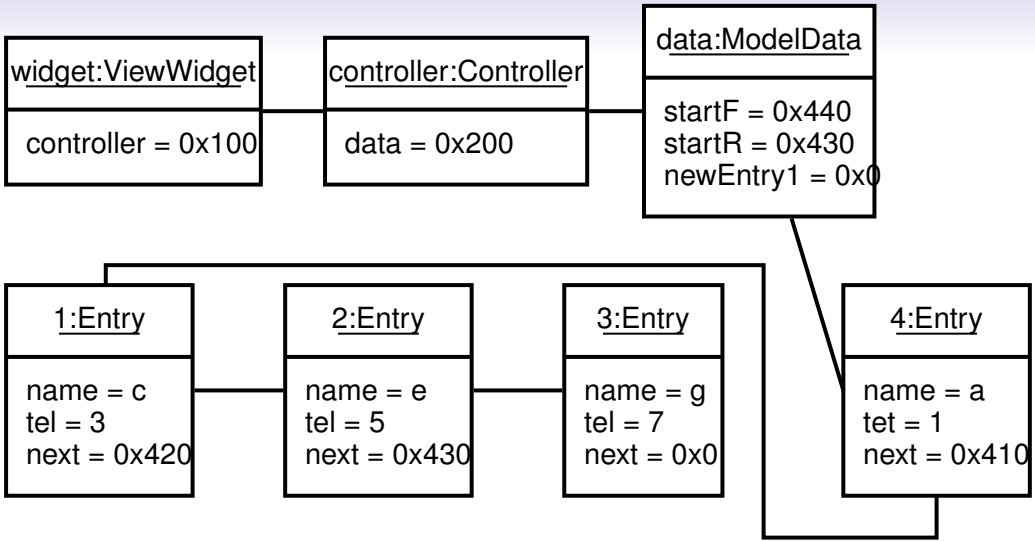

Abbildung: Objektdiagramm, ein Objekt vorne Einfügen, neues Objekt

Einen Eintrag vorne in der verketteten Liste einzufügen hat den Vorteil, dass man nur beim neuen Element den Nachfolger eintragen muss und `startF` korrigiert. Es kann jedoch sein, dass man einen Eintrag in der Mitte der Liste oder am Ende einfügen soll.
Hier muss ich zuerst die Position suchen. Bei einer einfach verketteten Liste benötige ich ggf. einen zweiten Hilfszeiger (Vorgänger), um „zurückgehen zu können“.
Im ersten Schritt erstelle ich wieder ein Objekt von `Entry` (siehe Bild 4) Das Objekt wird über `newEntry` referenziert.
Danach, in Schritt zwei muss ich die Position finden, an der das neue Element eingefügt werden soll. Hierfür verwende ich den Hilfszeiger `tempEntry`. Bei der Suche muss ich regelmäßig prüfen, ob das Objekt, auf das `tempEntry` verweist, vorhanden ist (nicht `nulltr`) und ob der Nachfolger ebenfalls nicht `nullptr` ist (Bilder 7 bis 9). In meinem Beispiel soll ein Eintrag mit dem Buchstaben f zwischen e und g eingefügt werden. Im Bild 9 zeigt der Hilfszeiger auf den Eintrag 3, Buchstabe g. Mein Ziel ist erreicht, ich habe die Position gefunden, vor der ich einfügen soll. In diesem Fall wäre es auch möglich die Position 2 (Bild 8) zu

suchen, um dahinter einzufügen. Es kann jedoch sein, dass ich nicht sicher sagen kann, ob direkt nach dem betreffenden Eintrag (hier Objekt 2) die Stelle ist, an der das neue Objekt eingefügt werden soll.
Ich weiß jetzt, dass das neue Objekt vor dem Objekt an der Adresse `0x430` eingefügt werden soll. Die Adresse des zusätzlichen Objekts (hier `0x440`) ist auch bekannt. Jedoch fehlt mir in diesem Moment die Adresse des Vorgängers zu Objekt 3. Ich könnte jetzt erneut von `startF` suchen, bis ich das Objekt finde, bei dem als `next` der Wert `0x430` eingetragen ist. Das würde die Suche erheblich verzögern[3]. Daher speichere ich zusätzlich zu `tempEntry1` auch `tempEntry2` ab (Bild 10).

```
void ModelData::insertNewEntryBeforeCurrent(QString nameVor,
String nameIn, QString telIn)
{
  Entry * tempEntry1 = fListStart;
  Entry * tempEntry2 = fListStart;
  Entry * newEntry = new Entry;
```

```
8       newEntry->setName(nameIn);
        newEntry->setTelNr(telIn);
10
        while((nullptr != tempEntry1->getNext()) &&
12      (0 != tempEntry1->getName().compare(nameVor)))
        {
14        tempEntry2 = tempEntry1;
          tempEntry1 = tempEntry1->getNext();
16      }

18      if(nullptr != tempEntry1)
        {
20        newEntry->setNext(tempEntry1);
          tempEntry2->setNext(newEntry);
22      }
```

```
}
```

Listing 3: Einfügen in der Mitte

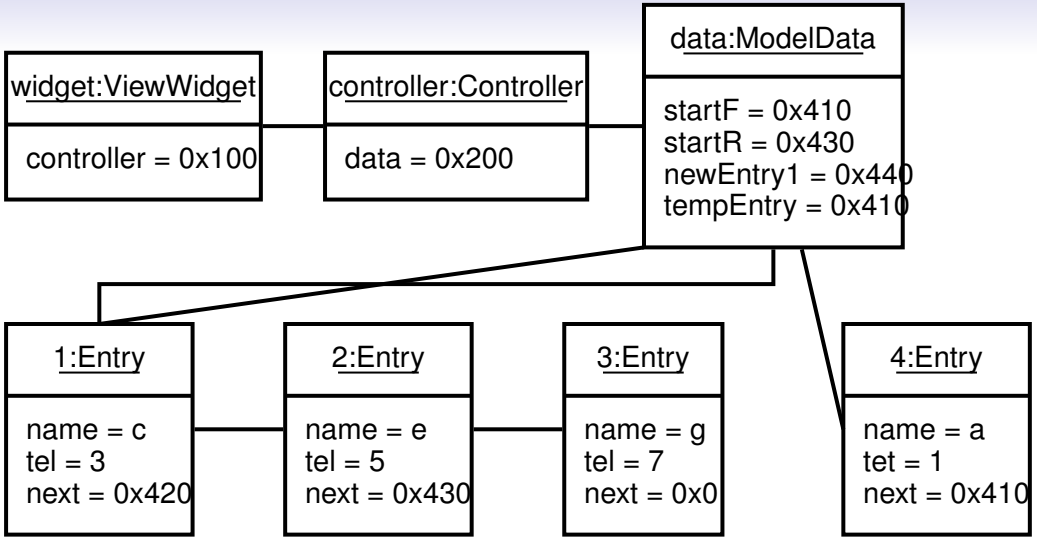

Abbildung: Objektdiagramm, ein Objekt mittig Einfügen, suche nach der Position zum

Einfügen

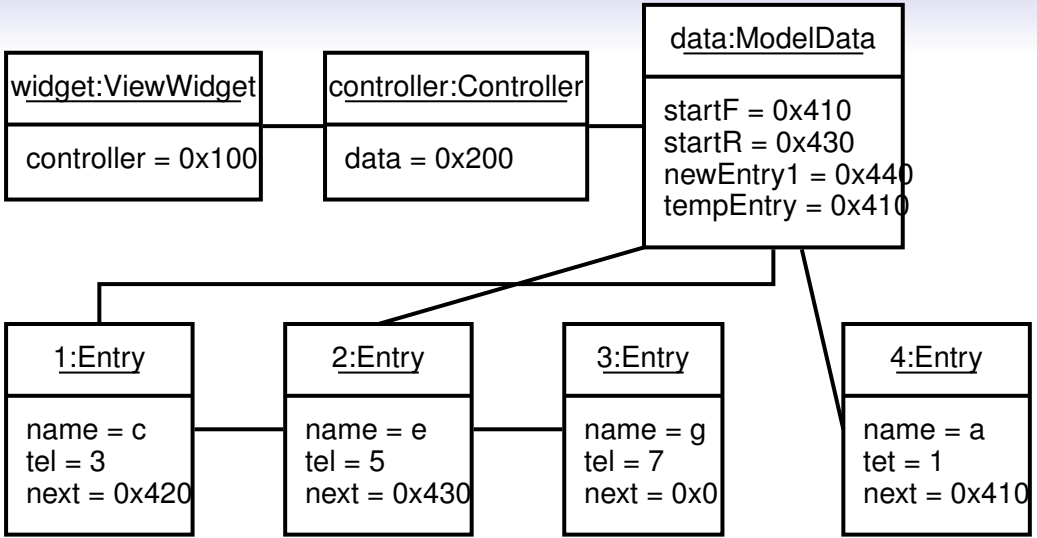

Abbildung: Objektdiagramm, ein Objekt mittig Einfügen, suche nach der Position zum

Einfügen

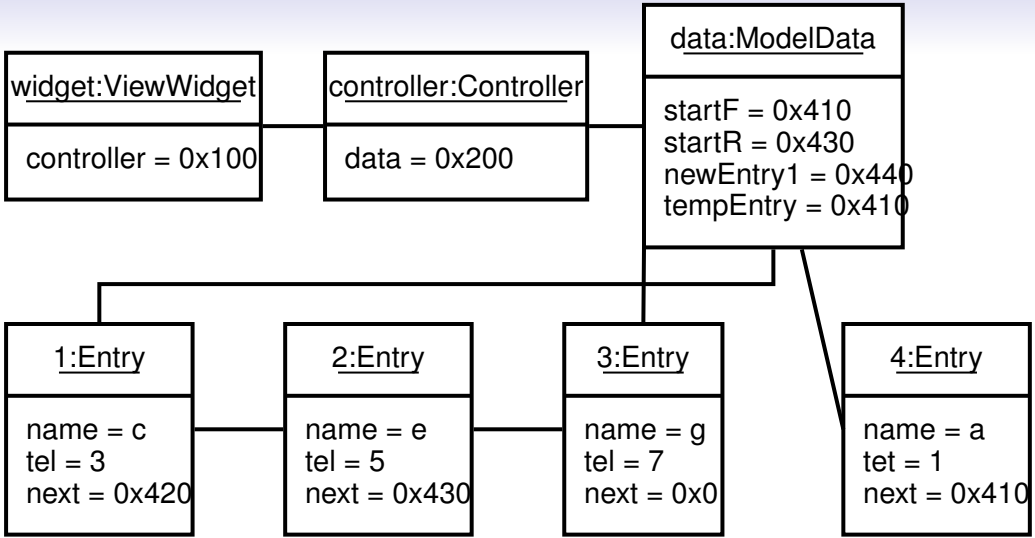

Abbildung: Objektdiagramm, ein Objekt mittig Einfügen, suche nach der Position zum

Einfügen

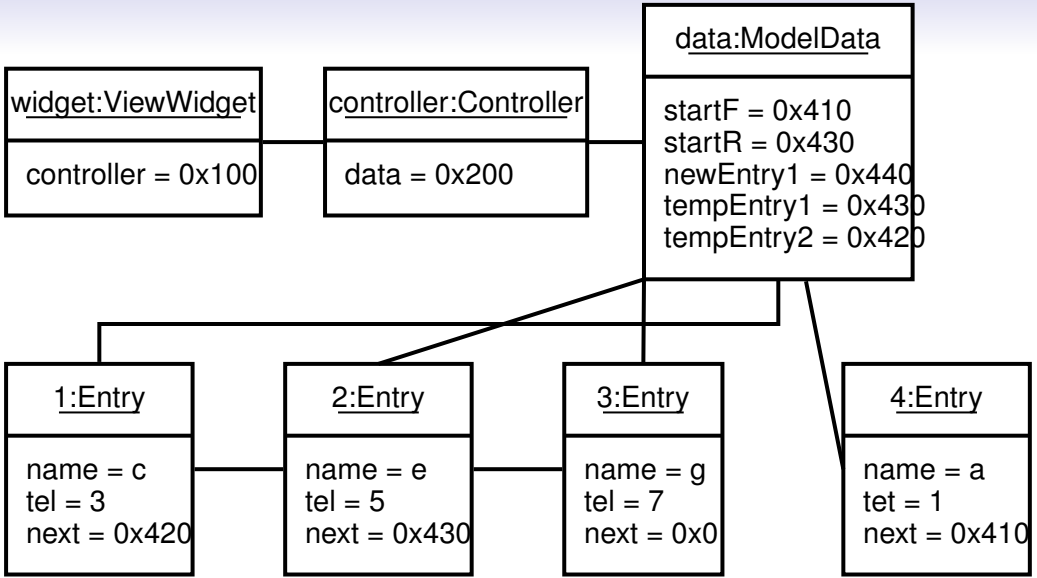
data:ModelData
startF = 0x410
startR = 0x430
newEntry1 = 0x440
tempEntry1 = 0x430
tempEntry2 = 0x420
widget:ViewWidget
controller = 0x100
controller:Controller
data = 0x200
1:Entry
name = c
tel = 3
next = 0x420
2:Entry
name = e
tel = 5
next = 0x430
3:Entry
name = g
tel = 7
next = 0x0
4:Entry
name = a
tet = 1
next = 0x410

Abbildung: Objektdiagramm, ein Objekt mittig Einfügen, suche nach der Position zum Einfügen mit zweitem Hilfszeiger auf Vorgänger

Beim Löschen eines Eintrags muss ich wieder die Adresse des Vorgänger-Elements kennen. Hier verwende ich wieder zwei Hilfszeiger. Das gesuchte Element wird in diesem Fall nicht mit new angelegt, sondern mit dem Hilfszeiger `tempEntry1` markiert. `tempEntry2` bekommt die Adresse des Vorgängers. Beim Austragen aus der Liste wird der Nachfolger von `tempEntry2` auf den Nachfolger von `tempEntry1` gesetzt. Anschließend kann das Objekt, das an `tempEntry1` hängt, mit `delete` entsorgt werden.

```
bool ModelData::removeEntry(QString nameIn)
{
  Entry * tempEntry1 = fListStart;
  Entry * tempEntry2 = fListStart; //Vorgaenger

  while((nullptr != tempEntry1) &&
  (0 != tempEntry1->getName().compare(nameIn)))
  {
    tempEntry2 = tempEntry1;
```

```
10        tempEntry1 = tempEntry1->getNext();
        }
12
        if(nullptr != tempEntry1)
14      {
          if(fListStart == tempEntry1)
16        {
            fListStart = tempEntry1->getNext();
18          if(nullptr != fListStart)
            fListStart->setPrevious(nullptr);
20          if(rListStart == tempEntry1)
            rListStart = nullptr;
22        }
          else
24        {
            tempEntry2->setNext(tempEntry1->getNext());
26          if(nullptr != tempEntry2->getNext())
```

```
            tempEntry2->getNext()->setPrevious(tempEntry2);
28          else
            rListStart = tempEntry2;
30        }
          delete tempEntry1;
32        return true;
        }
34      return false;
    }
```

Listing 4: Löschen eines Objekts aus der Liste

Die Rückgabe der Methode meldet, ob das Element gefunden und entsorgt wurde.

Objektreferenz

- Ein Array mit Pointern (als Attribut der Klasse)
- Verweis auf Objekte (auf dem Heap)
- Umsortieren ist einfach, da nur Pointer kopiert werden.

Klasse DataModel

```
1  class DataModel
   {
3  public:
       DataModel();
5      static const unsigned short arraySizeEntry = 20;
       void insertEntry(int pos, int val);
7      void removeEntry(int pos);
       int findEntryPos(int pos);
9      int findEntryVal(int val);
       bool appendEntryBack(int val);
11
   private:
13     Entry* entryAray[arraySizeEntry];///< Links auf die Objekte
                                            // in der Warteschlange.
15 };
```

Ring-Puffer

- Array ist linear.
- Problem: Kontinuierliche Messwerte,
  alte werden automatich überschrieben.
- logisch: Ring
- zwei “Zeiger” (schreiben / lesen)

Ring-Puffer

- Array ist linear.
- Problem: Kontinuierliche Messwerte,
  alte werden automatich überschrieben.
- logisch: Ring
- zwei “Zeiger” (schreiben / lesen)
- reale Struktur: Array (in der Regel)

Realisierung

- Array für Werte (z.B. Array für Int)
- Zeiger lesen und Zeiger schreiben
- Lesen darf schreiben nicht überholen
- Wenn schreiben am Ende des Arrays ist,
  Sprung auf Pos 0. Ggf. Lese-Zeiger verschieben.