Arbeitsunterlagen zu Prin Q1 bis Q4 (C++)

Thomas Maul

05.12.2025

Inhaltsverzeichnis

1 OOP - Objektorientierte Programmierung

1.1 Verkettete Liste

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 1.1).

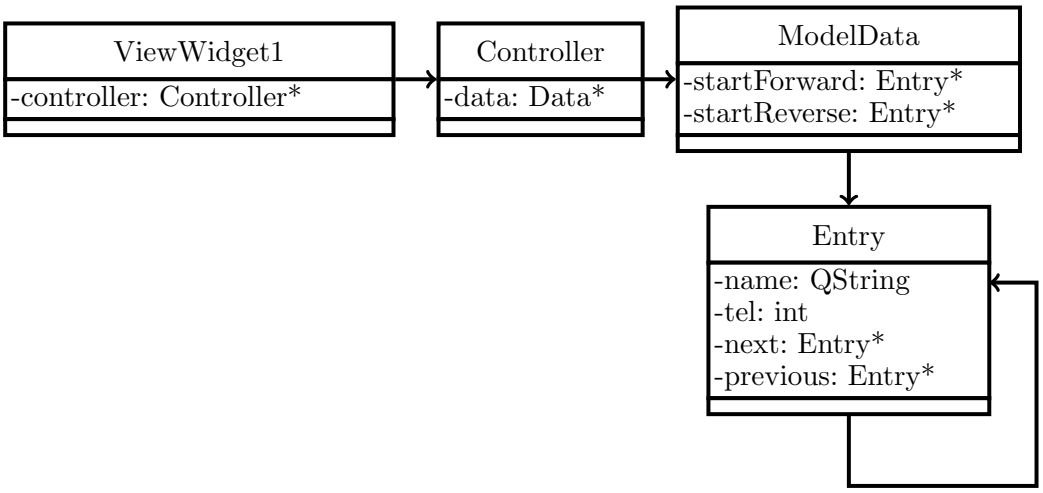

Abbildung 1.1: Klassendiagramm zu Programm, das die verkettete Liste verwaltet

Bild 1.2 zeigt das Programm zur Laufzeit in Form eines sogenannten Objektdiagramms. Hier mit einer leeren Liste - ohne Objekte zur Ablage von Daten in der Liste.

[1] Bei einem dynastisch angelegten Array (als Pointer, mit new) ist die Auswahl der Größe zur Laufzeit möglich.)

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.

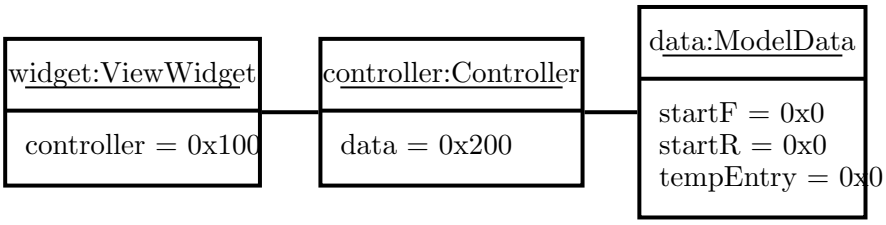

Abbildung 1.2: Objektdiagramm, leere Liste

In Bild 1.3 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.

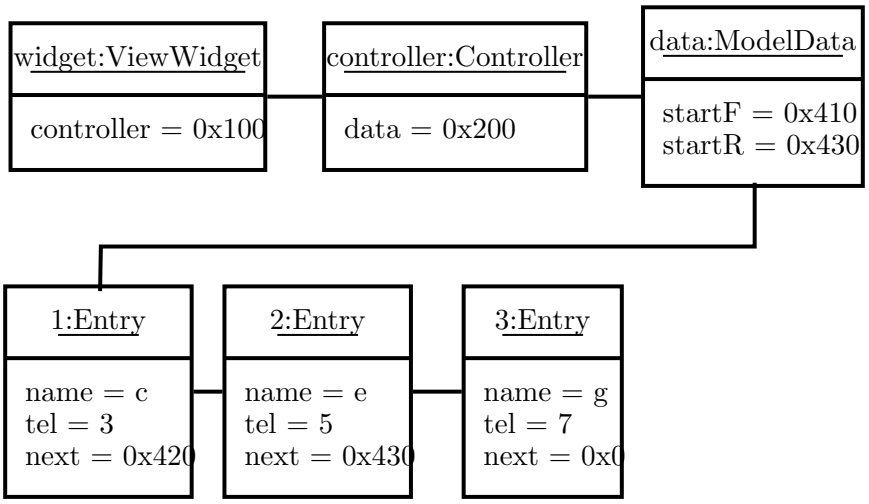

Abbildung 1.3: Objektdiagramm, drei Einträge in verkettete Liste

[2]0x bedeutet, dass die Zahl hexadezimal dargestellt wird

1.2 Einfügen in die 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).

1.2.1 Einfügen vorne

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 1.4, Listing 1.1).

Listing 1.1: Einfügen am Anfang (Teil 1)

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

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

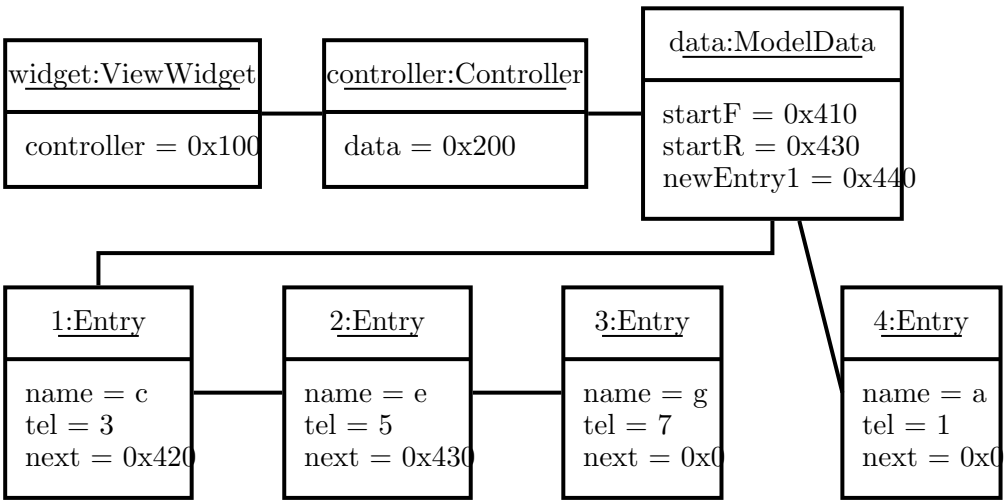

Abbildung 1.4: Objektdiagramm, ein Objekt vorne Einfügen, neues Objekt

Anschließend werden (Reihenfolge der Arbeitsschritte beachten!) die Zeiger aktualisiert (Bilder 1.5 und 1.6). In Bild 1.5 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 1.2 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).

Listing 1.2: Einfügen am Anfang (Teil 2)

```
if(nullptr == fListStart)
{
  fListStart = newEntry;
}
else
{
  if(nullptr != fListStart)
  {
    newEntry->setNext(fListStart);
    fListStart = newEntry;
    newEntry = nullptr;
  }
}
```

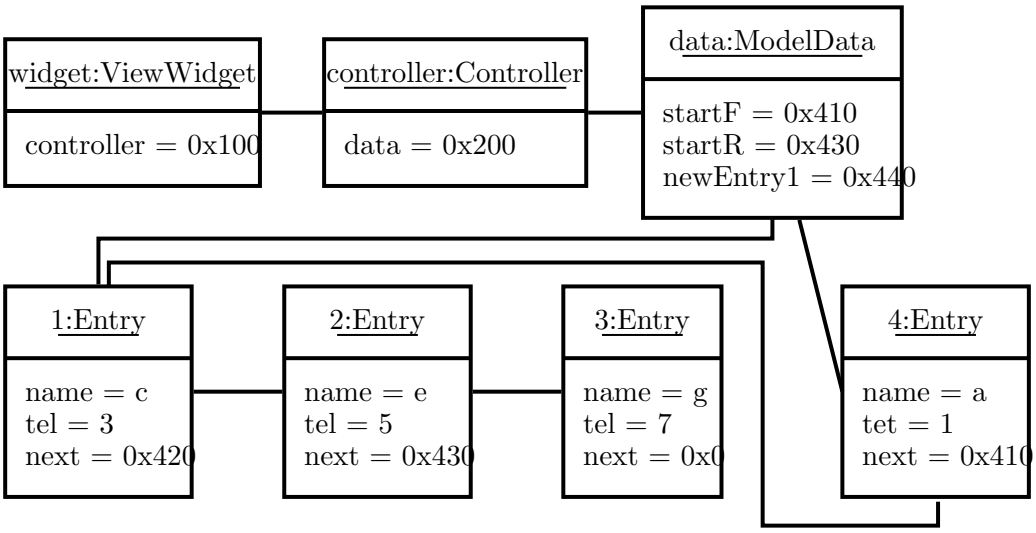

Abbildung 1.5: Objektdiagramm, ein Objekt vorne Einfügen, neues Objekt

In Bild 1.6 ist dann der Zeiger startF angepasst, der newEntry ist wieder auf 0x00 gesetzt. Damit ist das Einfügen am Anfang der Liste abgeschlossen.

1.2.2 Einfügen in der Mitte

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 1.4) 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 1.7 bis 1.9). In meinem Beispiel soll ein Eintrag mit dem Buchstaben f zwischen e und g eingefügt werden. Im Bild 1.9 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 1.8) 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 1.10).

Listing 1.3: Einfügen in der Mitte

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

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

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

[3]Bei 10 000 Einträgen und erheblich mehr wäre der zeitliche Ablauf sicher nicht mehr nahe 0 mS

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

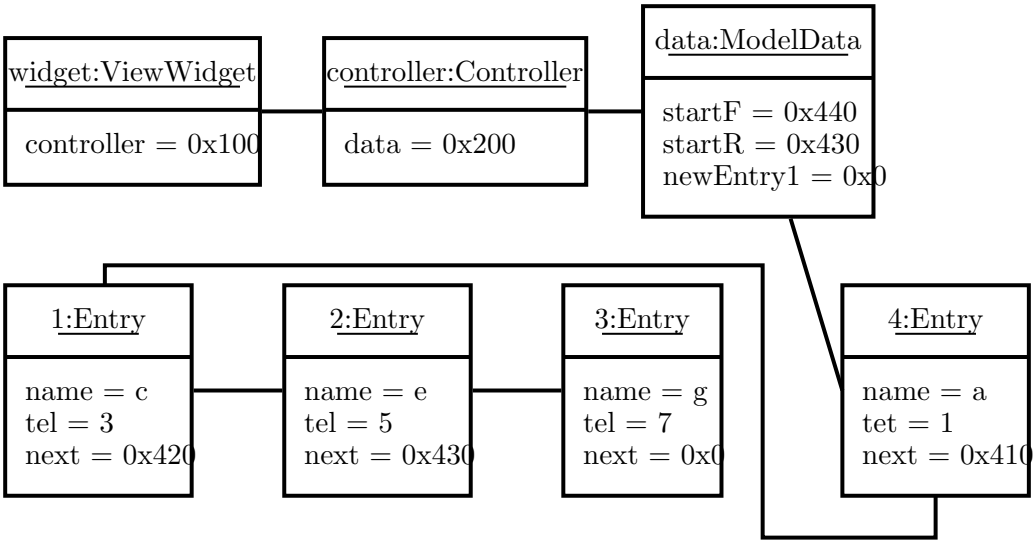

Abbildung 1.6: Objektdiagramm, ein Objekt vorne Einfügen, neues Objekt

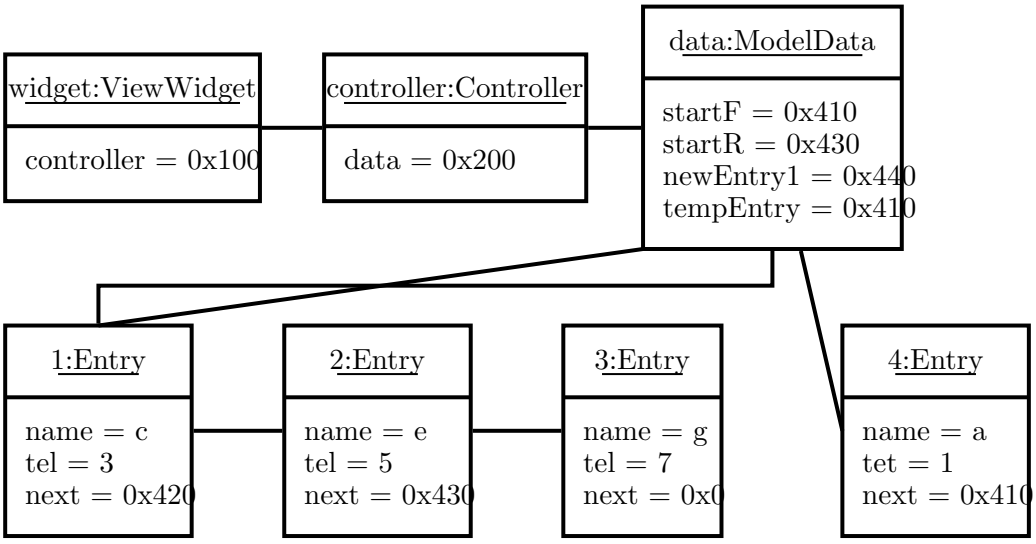

Abbildung 1.7: Objektdiagramm, ein Objekt mittig Einfügen, suche nach der Position zum Einfügen

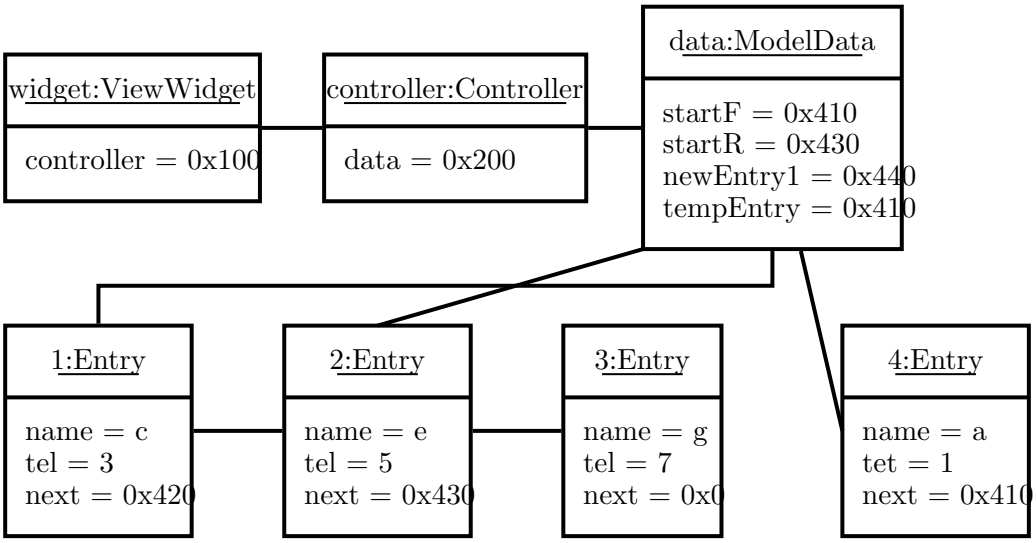

Abbildung 1.8: Objektdiagramm, ein Objekt mittig Einfügen, suche nach der Position zum Einfügen

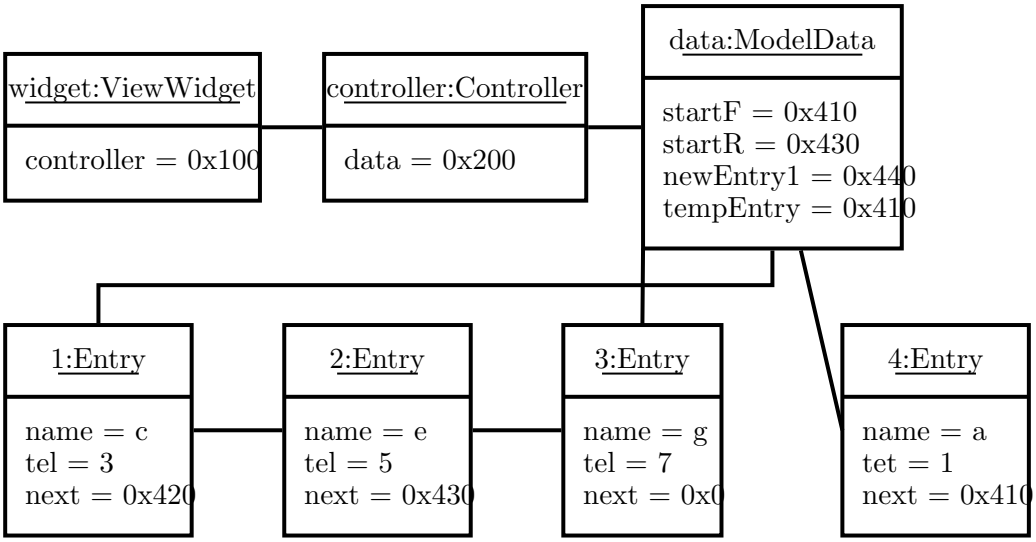

Abbildung 1.9: Objektdiagramm, ein Objekt mittig Einfügen, suche nach der Position zum Einfügen

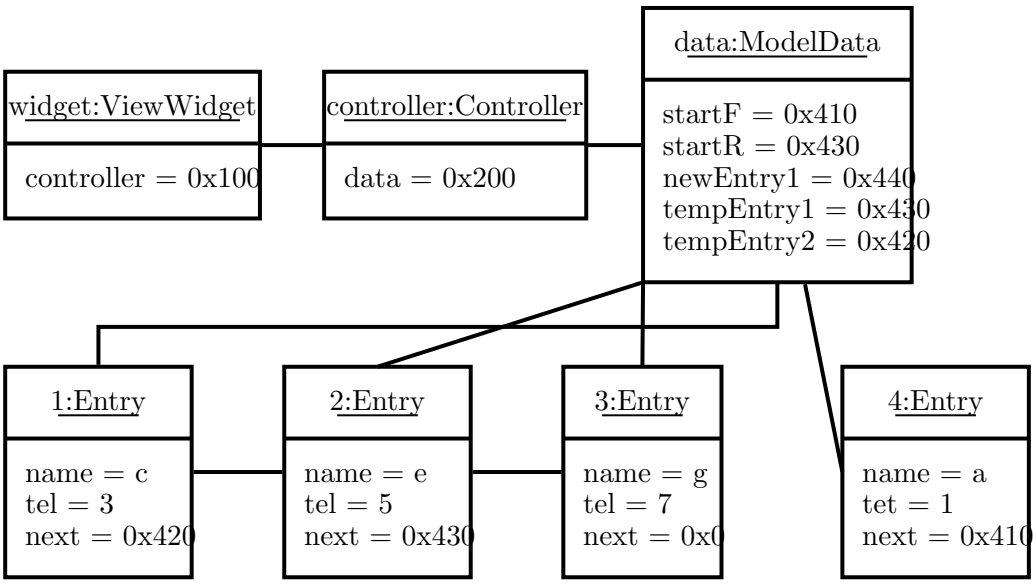

Abbildung 1.10: Objektdiagramm, ein Objekt mittig Einfügen, suche nach der Position zum Einfügen mit zweitem Hilfszeiger auf Vorgänger

1.2.3 Löschen eines Eintrags

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.

Listing 1.4: Löschen eines Objekts aus der Liste

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

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

  if(nullptr != tempEntry1)
  {
    if(fListStart == tempEntry1)
    {
      fListStart = tempEntry1->getNext();
      if(nullptr != fListStart)
      fListStart->setPrevious(nullptr);
      if(rListStart == tempEntry1)
      rListStart = nullptr;
    }
    else
    {
      tempEntry2->setNext(tempEntry1->getNext());
      if(nullptr != tempEntry2->getNext())
      tempEntry2->getNext()->setPrevious(tempEntry2);
      else
      rListStart = tempEntry2;
    }
    delete tempEntry1;
    return true;
  }
  return false;
```

}

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

1.3 Objektreferenz

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

```
  class DataModel
{
public:
    DataModel();
    static const unsigned short arraySizeEntry = 20;
    void insertEntry(int pos, int val);
    void removeEntry(int pos);
    int findEntryPos(int pos);
    int findEntryVal(int val);
    bool appendEntryBack(int val);

private:
    Entry* entryAray[arraySizeEntry];///< Links auf die Objekte
                                     // in der Warteschlange.
};
```

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

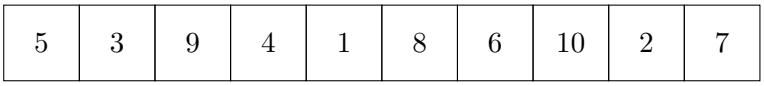

Abbildung 1.11: Array, Ausgangssituation

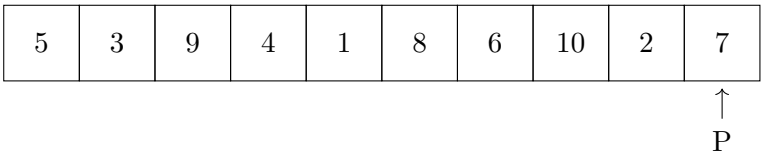

Abbildung 1.12: Array, Ausgangssituation

1.4 Quicksort

Es wird oft gewünscht, dass Daten sortiert vorliegen. Dabei ist es in der Regel gleichwertig, ob sie aufsteigend alphabetisch, absteigend oder nach anderen Kriterien sortiert werden.

Sortieralgorithmen sind daher ein häufiges Thema in der Informatik. Bubblesort ist einfach zu erlernen, aber relativ langsam. Bei großen Datenmengen wird Bubblesort eher nicht verwendet.

Quicksort wird häufiger verwendet. Der Algorithmus ist relativ schnell in der Ausführung. Quicksort kann in verschiedenen Varianten implementiert werden. Ich verwende die sogenannte in-Place-Variante, bei der zwei Zeiger die Elemente absuchen und direkt vertauschen, falls nötig.

Im Beispiel möchte ich das folgende Array mit Zahlen sortieren lassen.

1.4.1 Quicksort, Version in-place

Im ersten Schritt wird ein Pivotelement festgelegt. Das Element wird später benötigt, um andere Einträge mit dem Pivotelement zu vergleichen. Im ersten Schritt ist es gleichgültig, ob ich das erste oder letzte Element wähle. Ich wähle das letzte Element des Arrays (siehe Bild 1.12).

Jetzt werden zwei Hilfszeiger benötigt (klein und gross). Gross (g) beginnt am linken Ende des Arrays, klein (k) am rechten neben dem Pivotelement (siehe Bild 1.13). In jedem Arbeitsschritt prüfe ich, ob die Stelle bei klein kleiner ist als das Pivotelement und gross größer oder gleich groß gegenüber dem Pivotelement ist. Falls klein kleiner oder gleich ist im Vergleich zum Pivotelement lasse ich es an der Position stehen und setze den Suchzeiger (k) um eine Position weiter nach rechts. Falls ich ein Element finde, das größer als der Pivotelement ist, lasse ich den Suchzeiger k dort stehen und suche mit gross (g) nach einem Element, das kleiner oder gleich gegenüber dem Pivotelement ist.

Im Beispiel sucht der Zeiger ‚g‘ zuerst von links kommend das erste Element, das größer als 7 ist. Anschließend sucht k das nächste Element, das kleiner oder gleich 7 ist (Bild 1.14). Hier entsprechen die Werte 5 und 2 an den ersten Positionen von g und k

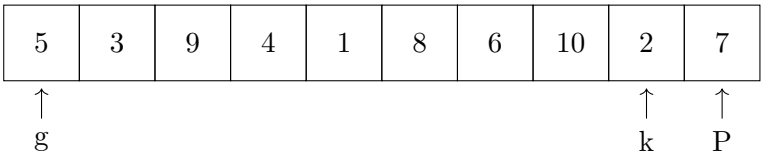

Abbildung 1.13: Array, Zeiger auf kleineres und größeres Element (relativ zu Pivotelement)

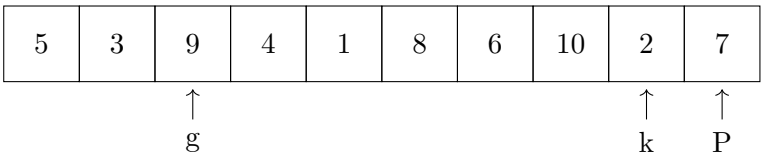

Abbildung 1.14: Array, Zeiger auf Elemente zum Tausch

den Kriterien der Suche.

Im folgenden Schritt werden die Einträge, auf die g und k zeigen, vertauscht. Anschließend sucht der Algorithmus wieder, solange, bis die Hilfszeiger wieder passende Elemente gefunden haben. Falls g und k bei der Suche aneinander vorbeilaufen, bricht der Such- und Vertauschungs-Algorithmus ab. In Bild 1.15 stehen 8 und 6 zum Tauscht an.

Zum Abschluss des Arbeitsschritts wird geprüft, ob die Inhalte an den Positionen P und g getauscht werden müssen. Wenn das Element an der Position g größer ist, als das an P (Bild 1.16), werden die Inhalte ausgetauscht. Damit ist die erste Runde des Algorithmus beendet.

Runde 2

In der nächsten runde werden die Abschnitte links und rechts von P_1 zu eigenen Teilen, die einzeln betrachtet werden. Im linken Abschnitt wird 6 zum neuen Pivotelement, 8 ist im rechten Abschnitt das neue Pivotelement (Bild 1.18).

In Runde zwei der Sortierung betrachte ich zuerst den linken Teil und suche hier wieder die Elemente, die kleiner und größer als das Pivotelement sind (Bild 1.18). Da kein Element aus der linken Teilmenge größer als 7 ist, wird hier nicht getauscht.

Im rechten Teil werden die Elemente 10 und 9 nicht getauscht, da 9 größer als 8 ist.

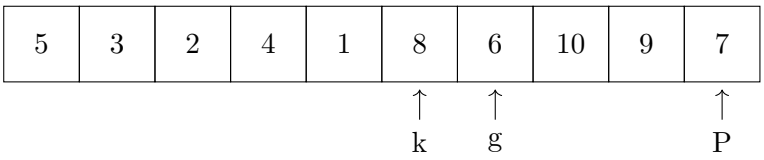

Abbildung 1.15: Zweites Paar zum Tauschen

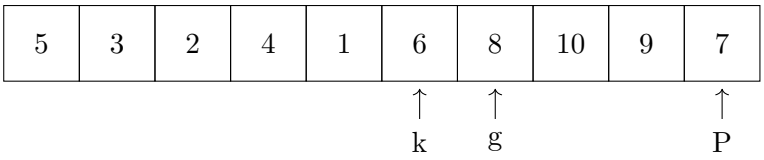

Abbildung 1.16: Array, Hilfszeiger haben die Position gewechselt

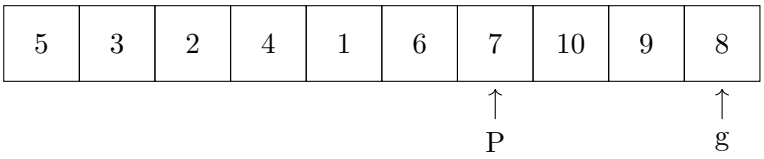

Abbildung 1.17: Tausch ‚gross‘ mit Pivotelement

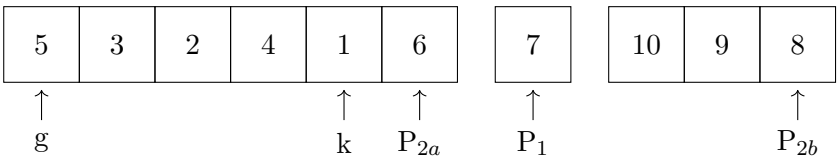

Abbildung 1.18: Aufteilung, P_1 ist alleine, links und rechts neue Bereiche

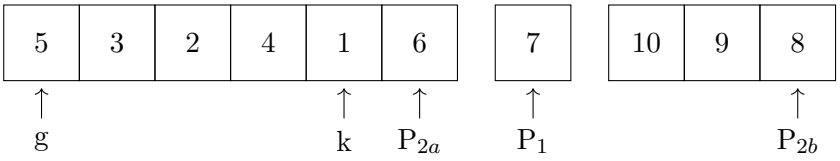

Abbildung 1.19: Suche links ist abgeschlossen ($5 < 7$)

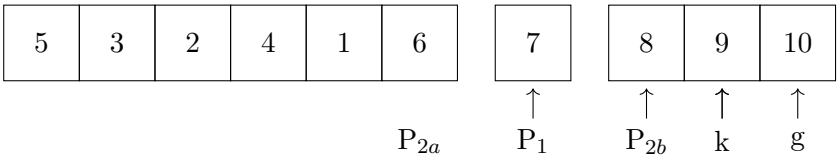

Abbildung 1.20: Tausch rechts (10 und 8)

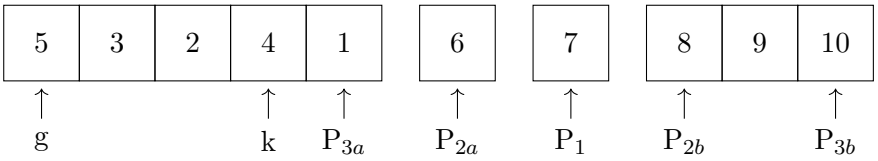

Abbildung 1.21: Runde 3, suche links (g, k, P_{3a})

Die zweite Prüfung ($10 > 8$) fällt positiv aus. Daher werden 10 (k) und P_{2b} getauscht. Somit sind 6 und 10 sortiert.

Rund 3

Da keine Zahl kleiner als 1 ist (Bild 1.21), werden hier keine Werte von g und k vertauscht. Lediglich 1 und 5 werden vertauscht. In der rechten Teilmenge ist nur noch 9 als Element vorhanden, hier ist der Vergleich $9 > 10$ negativ, daher bleiben die Elemente an der Position stehen. Somit ist auch Runde 3 angeschlossen.

Runde 4

In Runde 4 wird kein Element vertauscht (Bild 1.22).

Runde 5

In Runde 5 (Bild 1.23) werden wieder die Elemente 2 und 3 vertauscht. Bei der Prüfung in Runde 6 (1 und 2) gibt es keine weiteren Vertauschungen, damit ist der Sortieralgorithmus abgeschlossen.

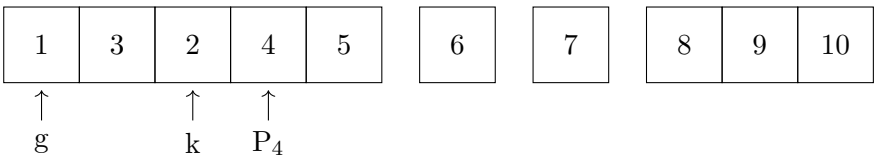

Abbildung 1.22: Runde 4, suche links (g, k, P_4)

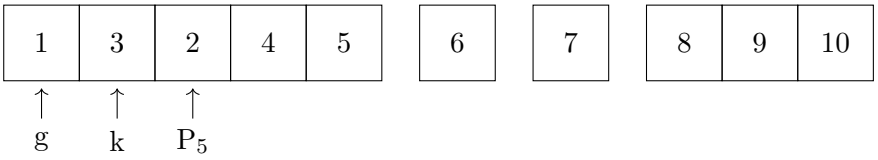

Abbildung 1.23: Runde 5, suche links (g, k, P_5)

1.4.2 Quicksort, zweite Implementierung

Bei der zweiten Variante wird ebenfalls ein Pivotelement festgelegt und dann alle andere Elemente danach ausgerichtet. Elemente, die kleiner sind kommen auf die linke Seite, Elemente, die größer sind auf die rechte Seite. Anschließend wird der Bereich am Pivotelement getrennt und jeweils der linke und der rechte Bereich als neue Gesamtbereiche betrachtet.