Algorithmen & Datenstrukturen: Liste mit Einfügen & Löschen

Klaus Kusche

Gesucht ist ein C++-Programm, das in einer *einfach verketteten Liste* an *beliebiger Stelle* Elemente *einfügen und herauslöschen* kann.

Unsere Liste soll *Kommazahlen* speichern. Neue Werte sollen *immer so eingefügt* werden, dass die Werte in der Liste *aufsteigend sortiert* sind.

Als erstes brauchen wir eine *Klasse* **`Elem`** für *ein einzelnes Listen-Element*.
Sie soll *ausschließlich für unsere Listen-Klasse verwendbar* sein (wie schafft man das?) und enthält:

- *Zwei Member*: Eines für den **`double`** (die "Nutzdaten" unserer Liste) und eines für die *Listen-Verkettung*.

  Deklariere den **`double`** so, dass er nach der Initialisierung im Konstruktor *nicht mehr geändert* werden kann (denn das würde ja die Sortierung zerstören).

- Einen *Konstruktor mit zwei Parametern*, die in die beiden Member des neuen Objektes gespeichert werden (d.h. man legt gleich beim Anlegen des Elementes seine Nutzdaten und seine anfängliche Verkettung fest).

Set- oder Get-Methoden braucht die Klasse nicht, da die Listen-Klasse die beiden *Member direkt lesen und schreiben* kann.

Dann kommt die *Listen-Klasse* (ich habe sie **`SortList`** genannt). Sie enthält:

- Ein *rein internes Member*: Den *Listenkopf-Pointer* (einen Listenende-Zeiger brauchen wir nicht, da wir keine "hinten anhängen"-Methode haben).
- Einen *Konstruktor ohne Parameter*, der die Liste auf *"leer"* setzt (was muss er dazu wo speichern?).
- Eine Methode **`insert`** mit **`double`**-Parameter und *ohne* Returnwert.
  Sie soll ein *neues Element* mit diesem Wert *dynamisch anlegen* und an der *richtigen Stelle* in die Liste *einfügen*.

  Dazu muss sie *zwei Fälle* unterscheiden:

  - Die Liste ist *leer*, oder der einzufügende Wert ist *kleiner als das erste Element*:
    In diesem Fall wird das neue Element *als vorderstes Element* eingehängt.
  - Sonst brauchen wir eine *Schleife*, die das *hinterste Element* in der Liste sucht, dessen *Wert kleiner als der einzufügende Wert* ist.
    Dazu muss die Schleife *ein Element vorausschauen* (ob es *noch ein Element* gibt, und ob das nächste Element noch *kleiner* als der einzufügende Wert ist).

    Hat die Schleife das letzte kleinere Element gefunden, wird das neue Element *zwischen diesem und dem darauffolgenden* eingehängt (der Code sollte *genauso* funktionieren, wenn das neue Element *als Letztes* an die Liste angehängt wird).

- Eine Methode **`remove`** mit **`double`**-Parameter und **`bool`**-Returnwert.
  Sie soll den Wert *aus der Liste entfernen* und den *Erfolg im Returnwert* anzeigen
  (**`true`** … Wert gelöscht, **`false`** … Wert nicht gefunden).
  Bei *mehreren* Elementen mit dem zu löschenden Wert
  soll sie *nur das erste davon* entfernen.

  Wir müssen *mehrere Fälle* unterscheiden:

  - Die Liste ist *leer*, oder das *erste Element* der Liste ist schon *größer* als der zu löschende Wert: In diesem Fall endet die Methode mit *"nicht gefunden"*.
  - Das *vorderste Element* hat den *gesuchten Wert*.
    In diesem Fall wird das vorderste Element aus der Liste *entfernt* und *vernichtet*.
  - Sonst brauchen wir wieder die *Schleife*, die mit *einem Element Vorausschau* die Liste *durchläuft* und uns einen Zeiger auf das *hinterste Element* der Liste liefert, dessen Wert *kleiner als der zu löschende Wert* ist.

  Nach dieser Schleife sind wieder zwei Fälle zu unterscheiden:

  - Es gibt *kein nachfolgendes Element* mehr,
    oder das *nachfolgende Element ist größer* als der zu suchende Wert.
    In diesem Fall endet die Methode mit *"nicht gefunden"*.
  - Das *nachfolgende Element* hat den *gesuchten Wert*.
    In diesem Fall muss es aus der Liste *entfernt* und dann *vernichtet* werden
    (welche beiden Elemente werden beim Entfernen verkettet?).
- Eine Methode **`output`** *ohne* Parameter und *ohne* Returnwert:
  Sie soll die Liste mit der üblichen "Listen-Durchlauf-Schleife" *komplett durchgehen*
  und die **`double`**-Werte der Reihe nach *ausgeben*.
  Außerdem soll sie dabei *mitzählen* und am Ende die *Anzahl der Elemente ausgeben*.

Das **Hauptprogramm** legt zuerst eine *Liste* an (*als lokale Variable*, nicht dynamisch).

Dann liest es in einer *Schleife* immer wieder einen **`double`** von **`cin`**,
bis das Lesen fehlschlägt. Dieser Wert wird geprüft:

- Ist er *größer* als **`0`**, wird er mit **`insert`** in die Liste *eingefügt*.
- Ist er *kleiner* als **`0`**, rufen wir **`remove`** mit dem *entsprechenden positiven Wert* auf
  und geben *je nach Returnwert eine Meldung* aus
  (*"… wurde gelöscht"* oder *"… wurde nicht gefunden"*).
- Ist er *gleich* **`0`**, *verlassen* wir die Schleife, und das Programm endet.
- Am Ende jedes Schleifendurchganges wird die Liste mit **`output`** *ausgegeben*.

Zusatzaufgabe 1: "Pointer-auf-Pointer-Trick"

Die Mutigen können versuchen, **`insert`** und **`remove`** mit dem *"Pointer-auf-Pointer-Trick"*
zu implementieren: Man verwendet einen Pointer, der *auf den Pointer zeigt*,
der *die zu verändernde Verkettung* enthält: Am *Anfang* zeigt er auf den *Listenkopf-Pointer*,
dann auf den *Verkettungs-Pointer* jenes Elementes, *hinter dem gelöscht oder eingefügt* wird.

Bei beiden Methoden sollte dadurch die *Sonderbehandlung* für das Einfügen oder Löschen
des *vordersten Elementes wegfallen*, d.h. **`insert`** besteht dann nur mehr aus *einem Fall*
statt zwei und **`remove`** aus *zwei Fällen* (gefunden / nicht gefunden) statt vier.

Zusatzaufgabe 2: Doppelt verkettete Liste

Wir wollen unser Beispiel von einer einfach auf eine *doppelt verkettete Liste* umbauen (aufbauend auf das *ursprüngliche* Beispiel, *nicht* auf die "Pointer-auf-Pointer"-Variante):

In der Klasse **`Elem`** kommt ein *Member* für die *Rückwärtsverkettung* dazu.
Des weiteren bekommt der *Konstruktor* einen *zusätzlichen Parameter*
um dieses Member zu initialisieren.

In der *Listen-Klasse* ändert sich folgendes:

- Im **`insert`** ändert sich betreffend Fallunterscheidung und Such-Logik nichts.
  Nur beim *eigentlichen Einfügen* sind in beiden Fällen auch die *Rückwärts-Pointer entsprechend zu verändern*.
  Achtung: Dafür ist in beiden Fällen eine *zusätzliche Abfrage nötig*!
- Das **`remove`** wollen wir *komplett umbauen*.

  Dazu schreiben wir zuerst einmal eine *neue Methode* **`find`**.
  Sie bekommt einen **`double`**-*Parameter* und soll einen *Pointer*
  auf das erste *Element mit diesem Wert* als *Returnwert* liefern
  (oder den *Nullpointer*, wenn es *kein Element* mit diesem Wert gibt).

  Für **`find`** genügt eine *ganz normale Listen-Durchlauf-Schleife* ohne Vorschau:
  Finden wir das Element, returnieren wir einen *Pointer darauf*.
  Ist das aktuelle Element schon *größer* als der gesuchte Wert,
  oder erreichen wir das *Listenende*, geben wir den *Nullpointer* zurück.
- Das **`remove`** ruft zuerst **`find`** auf.
  Liefert **`find`** den *Nullpointer*, kehrt **`remove`** sofort mit **`false`** zurück.

  Sonst müssen wir das Element, das **`find`** geliefert hat, aus der Liste *entfernen*:

  - Hat das Element einen *Vorgänger*, müssen wir *dessen Verkettung* ändern, *ansonsten* den *Listenkopf*.
  - Hat das Element einen *Nachfolger*, müssen wir *dessen Rückwärts-Verkettung* ändern.

  Am Ende wird das Element wieder *vernichtet*, damit der Speicher freigegeben wird.

Alles andere bleibt gleich.

Zusatzaufgabe 3: Listenklassen als Template

Eigentlich ist es eine unnötige Einschränkung, dass unsere sortierten Listen
nur für Kommazahlen verwendbar sind: Von der Logik her würden sie ja unverändert
für *alle Elementtypen* funktionieren, die man vergleichen und ordnen kann.

Wir wollen daher die *beiden Klassen* aus unserem ursprünglichen Beispiel
(ohne Zusatzaufgaben) *auf Templates umbauen*.

- Über beide Klassendefinitionen kommt ein **`template`**-Befehl.
- In den Klassen werden alle Vorkommen von **`double`**
  durch den Template-Parameter-Typ **`T`** ersetzt.
- Dann muss man überlegen, an welchen Stellen von **`SortList`**
  das bisherige **`Elem`** durch **`Elem<T>`** ersetzt werden muss.

- Etwas mehr Arbeit macht das **`friend class SortList`** :
  Auch hier muss **`SortList`** durch **`SortList<T>`** ersetzt werden. An dieser Stelle weiß der Compiler aber noch nicht, dass **`SortList`** ein Template ist.
  Daher ist *vor* der Klasse **`Elem`** eine *Vorab-Template-Klassendeklaration* von **`SortList`** nötig.
- Wenn man dann im *Hauptprogramm* eine **`double`**-Instanz von **`SortList`** verwendet, sollte alles so wie bisher funktionieren.

Wenn das klappt, wollen wir testen, ob unser Template *auch für Strings* funktioniert.
Dazu sind folgende Änderungen im Hauptprogramm nötig:

- Unsere Einlese-Variable wird ein **`string`**,
  und unsere **`SortList`** wird eine **`SortList`** für **`string`**-Werte.
- Die **`if`**-Bedingungen für die Fallunterscheidung ändern sich:
  - Bei einer Eingabe, die nicht mit **`'-'`** beginnt: Einfügen
  - Bei einer Eingabe, die mit **`'-'`** beginnt:
    Den String ohne das **`'-'`** am Anfang rauslöschen
    (Tipp: Die Klasse **`string`** enthält eine Methode **`substr`**)
  - Und wenn die Eingabe nur aus **`'-'`** besteht: Aufhören