Algorithmen & Datenstrukturen: Radix-Sort mit Listen

Klaus Kusche

Gesucht ist ein C++-Programm, das den ***Radix-Sort (Distribution-Sort)*** implementiert, und zwar unter Verwendung *einfach verketteter Listen* sowohl für die *Gesamt-Datenmenge* als auch für die *einzelnen Buckets im Verteil-Schritt*.

Der *Schlüssel*, nach dem die Daten sortiert werden, soll eine *ganze Zahl fixer Länge* sein, aber nicht als ein **`int`** gespeichert, sondern als *Array einzelner Ziffern*.

Zuerst definiert unser Programm *zwei Konstanten*:

- Die *Anzahl der Elemente*, die wir sortieren wollen.
- Die *Anzahl der Ziffern* im Sortierschlüssel.

Dann 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:

- *Vier Member*:
  - Ein *Array* von Ziffernanzahl vielen **`short int`**'s
    für die *Ziffern des Sortierschlüssels*.
  - Einen **`int`** für die *fortlaufende Nummer*,
    die jedem Element *bei der Erzeugung automatisch zugewiesen* wird.
  - Die *Listen-Verkettung*.
  - Einen ***klassenweiten*** **`int`** für die *automatische Nummerierung* der Elemente
    (vergiss nicht, für dieses klassenweite Member auch *Speicher anzulegen*
    und es auf den Wert **`1`** zu *initialisieren*!).
- Einen *Konstruktor ohne Parameter*.

  Er gibt dem Objekt die *nächste fortlaufende Nummer*,
  setzt den *Verkettungs-Pointer* auf den *Nullpointer*,
  und *befüllt* mit einer Schleife *jede Ziffer* im neuen Objekt
  mit einem *zufälligen Wert* von **`0`** bis **`9`** (**`rand`** verwenden).
- Eine Methode **`print`** *ohne* Parameter und *ohne* Returnwert:
  Sie gibt die *Element-Nummer*, einen Doppelpunkt, ein Tab,
  die *Ziffern des Sortierschlüssels* des Elementes und einen Zeilenvorschub aus.

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 **`Bucket`** genannt). Sie enthält:

- Zwei *rein interne Member*:
  Je einen *Pointer* auf das *erste* und das *letzte* Element der Liste.
  (der Radix-Sort hängt ja Elemente *hinten* an die Listen an!).
- Einen *Konstruktor ohne Parameter*, der die Liste auf *"leer"* setzt
  (was muss er dazu wo speichern?).

- Eine Methode **`removeFirst`** *ohne* Parameter,
  die das *vorderste Element* aus der Liste *aushängt*
  und einen *Pointer darauf* als *Returnwert* zurückliefert.

  Ist die Liste *leer*, returniert die Methode den *Nullpointer*.

- Eine Methode **append** *ohne* Returnwert
  und mit einem *Zeiger auf ein Listenelement* als Parameter.

  Sie soll dieses Element *hinten* an die Liste *anhängen*
  Dazu brauchst du eine *Fallunterscheidung*,
  ob die Liste bisher leer ist oder schon Elemente enthält.
  Der *Verkettungspointer des Elementes* muss ebenfalls entsprechend gesetzt werden.

- Eine zweite Methode **append**, ebenfalls *ohne* Returnwert,
  aber mit einem *Listen-Objekt* als Parameter (*"by reference"* übergeben).

  Diese Methode soll *alle Elemente* aus der übergebenen Liste *entnehmen*
  (die übergebene Liste wird daher am Ende der Methode auf "leer" gesetzt)
  und *hinten an die eigene Liste anhängen*, unter *Erhaltung ihrer Reihenfolge*.

  Das könnte man mit einer Schleife elementweise mit unserer Methode
  `removeFirst` und dem anderen **append** machen, aber es ist *viel zu ineffizient*,
  die Liste in einzelne Elemente zu zerlegen und wieder zusammenzusetzen.
  Wir wollen die Liste daher *als Ganzes umhängen*.

  Wenn die übergebene Liste *leer* ist, kehrt die Methode *sofort zurück*.

  Sonst *hängt* sie die Elemente der übergebenen Liste *als Ganzes an die eigene Liste* an
  (dazu brauchst du wieder eine *Fallunterscheidung*, ob die eigene Liste leer ist)
  und setzt dann die Liste im übergebenen Listeobjekt auf *"leer"*.

- Eine Methode **`sort`** *ohne* Parameter und *ohne* Returnwert,
  die *die eigene Liste sortieren* soll.

  Sie benötigt ein *Array von 10 Hilfs-Buckets* (*lokal* angelegt, nicht dynamisch)
  zum Verteilen der Listenelemente nach den Ziffernwerten 0 bis 9.

  Sie enthält eine *äußere Schleife*, die die *einzelnen Ziffern-Positionen*
  von *hinten* (letzte Ziffer) nach *vorne* (erste Ziffer) durchgeht.
  Die Schleife macht pro Ziffern-Position *einen Verteil- und einen Einsammel-Schritt*:

  - Der **Verteil-Schritt** baut mit einer *weiteren Schleife* die eigene Liste ab:
    Er *entnimmt* mit **`removeFirst`** das *vorderste* Element (wenn *keines mehr* da ist,
    endet die Schleife), *extrahiert* daraus die *Ziffer* an der aktuellen Position,
    und *fügt* das entnommene Element mit **append** *hinten an denjenigen Bucket* an,
    der dieser Ziffer entspricht.

  - Der **Einsammel-Schritt** besteht aus einer *Schleife*, die *von 0 bis 9* zählt
    und in jedem Durchlauf den *i-ten Hilfs-Bucket* mit dem anderen **append**
    hinten an die eigene Liste (die ja nach dem Verteil-Schritt leer ist) *anhängt*.

- Eine Methode **`fill`** mit einem **`int`**-Parameter *n* und *ohne* Returnwert:
  Sie *erzeugt* in einer *Schleife n* Mal dynamisch ein *neues Listen-Element*
  und *hängt* es mit **append** hinten *an die eigene Liste* an.

- Eine Methode **`print`** *ohne* Parameter und *ohne* Returnwert.
  Sie durchläuft die Liste mit einer ganz normalen *Listen-Durchlauf-Schleife*
  und ruft *für jedes Element* dessen **`print`**-Methode auf.
  Nach der Schleife gibt sie noch einen Zeilenvorschub aus.

Das **Hauptprogramm**

- sorgt zuerst dafür, dass **`rand`** bei jedem Programmlauf *andere Zufallszahlen* liefert,
- legt dann eine *Liste* an (*als lokale Variable*, nicht dynamisch),
- und ruft für diese Liste der Reihe nach **`fill`** (mit der Elementanzahl-Konstanten), **`print`**, **`sort`** und nochmals **`print`** auf.