Imperative Programmierung
Aufgabenblatt XI

Insgesamt wurde alle benötigten Funktionen in der Datei „liste.c“ bzw. „liste.h“ implementiert. Diese wurde komplett selbst geschrieben und verfügt auch noch über Funktionen die eigentlich nicht für die Aufgaben benötigt werden.

Die Liste basiert auf einer Datenstruktur mit dem Namen „list“, die einen Pointer zum ersten, letzten und aktuellen Element gespeichert. Außerdem werden die Länge und der Typ der Liste gespeichert. Das aktuelle Element kann durch die Funktionen „toFirst()“, „toLast()“ oder „next()“ verschoben werden und wird zum Einfügen eines Elementes benötigt (immer hinter dem aktuellen Element).

In „listNodes“ auf die die „list“ referenziert haben als einzige Attribute eine Referenz auf das nächste Element und natürlich die Daten, die in diesem Eintrag gespeichert werden sollen. Diese Daten sind vom Typ „void *“ damit verschiedene Datentypen in der Liste gespeichert werden können.

Alle Aufgaben, die nun beschrieben werden, sind in „liste_Aufgaben.c“ implementiert. Dabei wird „liste.h“ importiert und dementsprechend die Funktionen genutzt.

Auf Grund der Tatsache das Code dementsprechend in 2. Dateien steht, müssen beim Kompilieren auch beide Dateien kompiliert werden. Dazu kann z.B. folgender Befehl benutzt werden:

$$gcc\ -o\ liste\_Aufgaben\ liste\_Aufgaben.c\ liste.c$$

1. Bei der ersten Aufgabe muss die Liste einfach nur eingelesen werden, dazu wurde eine Methode „append()“ bzw. „appendString()“ geschrieben, mit deren Hilfe ein String ans Ende der Liste eingefügt werden kann.
   Dann kann einfach mit der „newList()“ eine neue Liste erstellt werden und dann einfach die Datei mit der „appendString()“ Methode Zeile für Zeile zur Liste hinzugefügt werden. Dabei entsteht auch nicht das Problem, dass die Liste falschherum eingelesen wird.
2. Zwar entstand bei meiner Implementierung nicht das beschriebene Problem, allerdings habe ich trotzdem eine Funktion „reverse()“ implementiert. Diese Methode arbeitetet „inPlace“ es muss also nicht extra eine neue Liste erstellt werden
3. Hierzu wurde eine Funktion „merge“, die dem jeweiligen Datentypen nach die Funktion „_mergeList“ aufruft. Dies ist notwendig da beim Aufruf von „_mergeList“ eine „Vergleichs“-Funktion benötigt wird, nach der die Einträge verglichen werden sollen. Da die Listen sortiert sind, wird einfach jeweils ein aktuelles Element festgelegt. Dieses liegt am Anfang jeweils beim ersten Eintrag. Diese Einträge werden dann verglichen und das kleinere zur resultierenden Liste hinzugefügt („append()“). Dann wird das aktuelle Element der Liste, aus der das Element stammte, auf das nächste Element gesetzt.
   Dieser Prozess wird so lange wiederholt, bis bei beiden Eingabelisten das Ende erreicht wurde.

Falls die Liste keine Standarddatentypen verwendet (also ein struct gespeichert wird), muss an die Funktion „merge“ eine selbst geschriebene „Vergleichs“-Funktion übergeben werden. Andernfalls wird einfach NULL übergeben.
Beispiele für solche Vergleichsfunktionen sind z.B. „compareString()“ oder „compareInt()“.

4. Um eine Liste aufzuteilen, kann einfach die Mitte der Liste berechnet werden (Länge Attribut der Liste). Dann werden einfach zwei neue Listen erstellt und jeweils die erste bzw. die zweite Hälfte in die Listen mit der Funktion „append()“ hinzugefügt.
   Bei ungeraden Längen von Listen ist das mittlere Element das erste Element der zweiten Liste. Damit ist die zweite Liste auch eins länger als die erste.