Prof. Dr. Wolfgang Effelsberg
Dirk Stegemann Mannheim, den 06.06.2005

Vorlesung **Programmierkurs 2**, Sommersemester 2005

Übungsblatt 8

Abgabe spätestens am Montag, 13. Juni 2005, um 13.45 Uhr.

Aufgabe 1: Minesweeper Teil II

(12 Punkte, Abgabedatei `minesweeper.c`)

Basierend auf der Aufgabe 2 vom Übungsblatt 6 soll das Minesweeper-Spiel in C implementiert werden. Es gelten die folgenden Spielregeln:

(i) Der Benutzer gibt die Höhe und die Breite des Spielfelds (jeweils zwischen 1 und 26) sowie die Anzahl der Minen an, die auf dem Spielfeld verteilt werden sollen. Der Computer erzeugt ein zufälliges Spielfeld mit diesen Parametern und gibt das Spielfeld verdeckt aus.

(ii) Der Benutzer gibt ein Koordinatenpaar (x, y) ein. Jetzt gibt es drei Möglichkeiten:

a) An der Position (x, y) auf dem Spielfeld ist eine Mine. In diesem Fall hat der Benutzer verloren, und das Spiel ist beendet.

b) An der Position (x, y) ist eine Zahl, und es wird nur das Feld (x, y) aufgedeckt.

c) An der Position (x, y) ist ein `'-'`. Der Computer deckt rekursiv die komplette Nachbarschaft von (x, y) auf, d.h. alle Pfade von (x, y) zu mit Zahlen beschrifteten Feldern (i, j), die zwischen (x, y) und (i, j) nur mit `'-'` beschriftete Felder enthalten. Zulässige Nachfolger eines Felds (x, y) auf einem Pfad sind nur die Felder $(x-1, y)$, $(x+1, y)$, $(x, y-1)$ und $(x, y+1)$, sofern diese existieren. Pfade können also nicht diagonal verlaufen.

Dieser Schritt wird solange wiederholt, bis der Benutzer entweder auf eine Mine getreten ist oder alle Felder ohne Minen aufgedeckt sind.

(iii) Am Ende gibt der Computer die Auflösung an, d.h. alle Felder mit Minen werden aufgedeckt.

Beispiel:

```
Breite des Felds eingeben: 5
Hoehe des Felds eingeben: 5
Anzahl der Minen eingeben: 2
  abcde
  +++++
a|xxxxx
b|xxxxx
c|xxxxx
d|xxxxx
e|xxxxx
Welchen Eintrag soll ich aufdecken? aa
  abcde
  +++++
a|---1x
b|---11
c|11---
d|x1---
e|x1---
Welchen Eintrag soll ich aufdecken? ae
Herzlichen Glueckwunsch, du hast alle Minen gefunden!
  abcde
  +++++
a|---1M
b|---11
c|11---
d|M1---
e|11---
```

Du kannst vereinfachend annehmen, dass dein Programm nur korrekt formatierte Eingaben erhält und dass die Buchstaben `a,...,z` zusammenhängend im Zeichensatz der Maschine angeordnet sind. Ansonsten ist die Implementation der Benutzerschnittstelle dir überlassen. Achte darauf, dein Programm sinnvoll in Funktionen zu zerlegen.

Aufgabe 2: Generische verkettete Listen

(8 Punkte, Abgabedatei `genlist.c`)

Nachdem wir im Rahmen des letzten Übungsblatts verkettete Listen zum Speichern von `int`s erzeugt haben, wollen wir nun von konkreten Datentypen abstrahieren und eine generische Liste erzeugen, die in den Knoten Zeiger auf die zu verwaltenden Daten anstatt der Daten selbst speichert. Dazu definieren wir die Knoten als

```
typedef struct listelement{
  void* inhalt;
  struct listelement* next;
} knoten;
```

Implementiere die folgenden Funktionen zur Verwaltung von generischen verketteten Listen:

- `knoten* neuerKnoten(void* inhalt, knoten* next)` erzeugt einen neuen Knoten, dessen Inhalt durch `inhalt` referenziert wird und der auf den Knoten `next` zeigt. Die Funktion gibt `NULL` zurück, wenn der Knoten nicht erzeugt werden konnte.
- `knoten* neueListe()` erzeugt eine neue verkettete Liste und gibt einen Zeiger auf den ersten Knoten der Liste zurück. Die Funktion gibt `NULL` zurück, wenn die Liste nicht erzeugt werden konnte.
- `knoten* vorneEinfuegen(knoten* liste, void* inhalt)` fügt einen neuen Knoten mit dem übergebenen `inhalt` an den Anfang der `liste` an und gibt den Knoten zurück. Die Funktion gibt `NULL` zurück, wenn der neue Knoten nicht erzeugt werden konnte.
- `knoten* vorneEntnehmen(knoten* liste)` entnimmt den ersten Knoten der Liste und gibt `NULL` zurück, falls die Liste leer ist.
- `knoten* einsortieren(knoten* liste, void* inhalt, int (*comp)(void*, void*))` erzeugt einen neuen Knoten mit dem übergebenen Inhalt und sortiert den Knoten aufsteigend mit Hilfe der Funktion `comp` in die `liste` ein. `comp` vergleicht die durch die beiden Zeiger referenzierten Knoteninhalte und gibt einen Wert < 0 zurück, wenn der erste Knoteninhalt kleiner ist als der zweite, 0 wenn beide gleich sind, und einen Wert größer als 0 sonst. `einsortieren` gibt `NULL` zurück, falls der neue Knoten nicht erzeugt werden konnte.
- `knoten* listeFreigeben(knoten* liste)` gibt den durch die `liste` belegten Speicherplatz wieder frei.

Die Funktionen `vorneEinfuegen`, `vorneEntnehmen` und `einsortieren` sollen auch im Fehlerfall die Liste in einem konsistenten Zustand hinterlassen.

Teste deine Listenimplementation, indem du eine `main`-Funktion schreibst, die vom Benutzer 5 Vornamen mit jeweils höchstens 15 Zeichen einliest und diese Namen mit Hilfe einer verketteten Liste lexikografisch absteigend sortiert.