Dr. Johannes Mayer
Axel Blumenstock
Christian Ehrhardt
Abteilung Angewandte Informationsverarbeitung

13. Dezember 2005
Blatt 8

Allgemeine Informatik III / Systemnahe Software I (WS 2005/2006)

Abgabetermin: 20. Dezember 2005

Wort des Jahres (10 Punkte)

Sprach- und Medienforscher interessieren sich sehr dafür, welche Wörter „in aller Munde“ sind, wie sich ihre Verwendung einbürgert, und wie andere wieder in Vergessenheit geraten. Auf `http://wortschatz.uni-leipzig.de/wort-des-tages` finden Sie Wörter, die an diesem Tag überdurchschnittlich häufig in den Medien auftauchen. Und bestimmt war da auch mal das (bald wieder zu wählende) Wort des Jahres dabei.
In diesem Blatt wollen wir ein einfaches Programm erstellen, das Textdateien aus der Standardeingabe liest, in Wörter zerlegt, eine Worthäufigkeitstabelle erstellt und diese auf der Standardausgabe ausgibt. Auf die GNU GPL angewandt, könnte das wie folgt aussehen:

```
$ ./a.out < COPYING | head
    57 x a
     6 x above
     1 x absence
     1 x absolutely
     2 x accept
     1 x acceptance
     2 x access
     1 x accompanies
     3 x accompany
     1 x accord
```

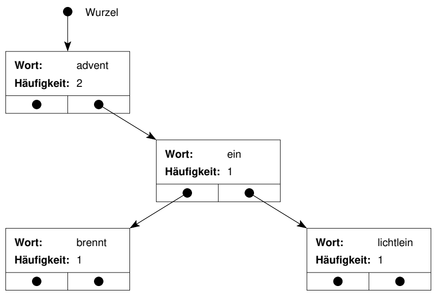
Wir wollen nicht zwischen Groß- und Kleinschreibung unterscheiden und wandeln dazu alle Wörter in Kleinbuchstaben um. Ferner soll genügen, als Wort solche Zeichenketten anzusehen, die nur aus Buchstaben bestehen; sämtliche anderen Zeichen, insbesondere Bindestriche oder Ziffern, fungieren als Worttrenner und sind ansonten zu ignorieren.
Da wir vorab nicht wissen, wie viele verschiedene Wörter vorkommen werden, wollen wir eine dynamische Datenstruktur verwenden. Um einen schnellen Zugriff mit der Möglichkeit einer einfachen sortierten Ausgabe zu verbinden, bietet sich an, die Worthäufigkeitstabelle als binären Suchbaum zu realisieren.
Seine Knoten sind structs mit je einem Element für

- das eigentliche Wort als `char *`
- dessen absolute Häufigkeit als `int`
- sowie je einen Zeiger auf einen (möglichen) linken und einen rechten Nachkommen.

In Ihrem Programm müssen Sie lediglich einen Zeiger auf den Wurzelknoten deklarieren. Das Einfügen eines Wortes vollzieht sich dann wie folgt: Hat der Baum noch keinen Knoten (der Zeiger auf die Wurzel ist also NULL), erzeugen Sie einfach eine Wurzel mit dem einzufügenden Wort und der Häufigkeit 1. Ansonsten durchlaufen wir eine Reihe von Knoten, beginnend mit der Wurzel, wie folgt:

- Wir vergleichen das einzufügende Wort mit dem im aktuellen Knoten vermerkten. Sind sie gleich, inkrementieren wir den Häufigkeitszähler und sind fertig.
- Ist das einzufügende Wort lexikalisch größer (d.h. in der alphabetischen Sortierung weiter hinten), so kommt als nächstes der rechte, ansonsten der linke Nachkomme in Betracht.
- Existiert dieser Nachkomme nicht, so legen wir einen neuen Knoten mit dem einzufügenden Wort an und hängen ihn als genau diesen Nachkommen in den Baum. Ansonsten setze die Verarbeitung mit diesem Nachkommen bei Punkt 1 fort.

Nach Abarbeiten des Satzes „Advent, Advent, ein Lichtlein brennt“ sollte der Suchbaum also wie folgt aussehen:



Wir stellen zudem fest, dass wir mit einem *Inorder*-Baumdurchlauf[1] die Knoten in alphabetischer Reihenfolge ausgeben können.
Es folgt eine Liste der wesentlichen Wegmarken beim Erstellen dieses Programms; gewissen Code in weitere Funktionen auszulagern bietet sich immer mal wieder an.

(a) Schreiben Sie eine Funktion, die die Standardeingabe liest, in einzelne Wörter zerlegt und alles zu Kleinbuchstaben macht. Testen Sie diese Funktion, indem Sie sie beispielsweise in einer kleinen *main* verwenden und gelesene Wörter zeilenweise ausgeben.

[1]Das heißt: Aus Sicht jedes Knotens verarbeite zunächst rekursiv den linken Teilbaum, dann den Knoten selbst, dann den rechten Teilbaum.

(b) Definieren Sie zur Repräsentation der Knoten eine `struct Node`.

(c) Schreiben Sie eine Funktion

```
struct Node *createNode(char *word)
```

welche für das gegebene Wort einen neuen Knoten im Speicher anlegt (*malloc*) und passend initialisiert. Beachten Sie, dass Sie das gegebene Wort kopieren müssen, damit es beim Einlesen des nächsten nicht überschrieben wird.

(d) Erstellen Sie ferner eine Funktion wie

```
void findAndUpdate(struct Node *node, char *word)
```

welche das oben beschriebene Vorgehen zum Suchen, Aktualisieren bzw. Hinzufügen eines entsprechenden Knotens im Baum realisiert.

(e) Für die Inorder-Ausgabe der Häufigkeiten nebst zugehörigen Wörtern sehen Sie eine Funktion vor wie

```
void printFrequencies(struct Node *node)
```

(f) Fügen Sie auch eine Funktion wie

```
void freeTree(struct Node *node)
```

hinzu, die in einem Postorder-Baumdurchlauf[2] den Speicherplatz des gesamten Baumes wieder freigibt.

(g) *(Zusatzaufgabe, 3 Punkte)* Anstatt die gesamte Liste der Wörter auszugeben, wollen wir nur die Top 20 sehen. Legen Sie dazu ein Array an, welches (Zeiger auf) alle Knoten Ihres Suchbaumes enthält, und sortieren Sie es (unter Verwendung der Funktion *qsort()*) primär nach absteigender Häufigkeit, und Wörter mit der gleichen Häufigkeit alphabetisch. Von diesem Array geben Sie dann die ersten 20 Elemente aus.

Tips Geben Sie die Häufigkeiten in jeder Zeile zuerst und in einer festen Breite aus, können Sie durch Anwendung von `sort` und `head` an der Kommandozeile auch ohne weiteren Aufwand eine Top-N-Liste sehen:

```
$ ./a.out < COPYING | sort -r | head
   194 x the
   108 x to
   104 x of
    77 x or
    76 x you
    72 x and
    71 x program
    57 x a
    53 x is
    49 x this
```

[2]Also zunächst – falls vorhanden – beide Nachkommen rekursiv verarbeiten, danach den Knoten selbst.

Einige nützliche Bibliotheksfunktionen[3]:

- `int tolower(int c)` gibt, falls *c* ein Großbuchstabe ist, den zugehörigen Kleinbuchstaben, ansonsten *c* unverändert zurück.
- `int strlen(char *a)` gibt die Länge des Strings *a* zurück.
- `char *strcpy(char *a, char *b)` realisiert in C die „Stringzuweisung" $a \leftarrow b$, wobei hier (im Gegensatz zu `a = b`) nicht die Adresse des ersten Zeichens, sondern der gesamte Inhalt kopiert wird. Das Zielarray *a* muss vorher angelegt sein und groß genug, um *b* **inklusive des abschließenden Nullzeichens** aufnehmen zu können.
- `char *strdup(char *a)` dupliziert den gegebenen String und gibt einen Zeiger auf die Kopie zurück. Der Speicher kann später mit *free()* freigegeben werden.
- `int strcmp(char *a, char *b)` vergleicht zwei Strings und gibt eine Ganzzahl größer null, null oder kleiner null zurück, wenn der erste String alphabetisch größer, gleich bzw. kleiner ist.
- `sizeof(x)` ist keine Bibliotheksfunktion, sondern ein C-Primitiv, das den Speicherbedarf eines Objekts *x* (einer Variable oder eines Typs) zurückgibt. Es ermittelt beispielsweise *nicht* die tatsächliche Länge eines Strings.
- `void *malloc(int n)` reserviert einen Speicherbereich von *n* Bytes Größe und gibt einen (untypisierten) Zeiger darauf zurück. Dieser kann danach in einen beliebigen Typ umgewandelt und dereferenziert werden.
- `void free(void *p)` gibt einen zuvor mit *malloc()* angelegten Speicherbereich wieder frei. Danach darf *p* nicht mehr dereferenziert werden.
- `scanf("%[A-Za-z]", s)` ist eine einfache Möglichkeit, eine Zeichenkette, die nur aus A–Z und a–z bestehen darf, in ein char-Array *s* einzulesen. Durch ein `^` als erstes Zeichen nach der öffnenden eckigen Klammer wird die Zeichenmenge negiert, d.h., es wird alles gelesen, was *nicht* auf den nachfolgend angegebenen Zeichenbereich passt. Wie gewohnt gibt der Rückgabewert Auskunft über den Leseerfolg der Funktion. *s* muss vorher angelegt und groß genug sein, um die gesamte Zeichenkette (inkl. abschließendes Nullzeichen) aufzunehmen. Es empfiehlt sich in jedem Fall, durch Angabe einer Feldbreite nach dem `%` die Zahl eingelesener und zugewiesener Zeichen zu beschränken.

Für weitere Details bzw. die jeweils einzubindenden Header-Dateien beachten Sie bitte die jeweiligen Manualseiten.
Viel Erfolg!

[3]Die Signaturen sind nicht ganz korrekt wiedergegeben, um die Liste übersichtlicher zu halten.