Dr. Johannes Mayer
Axel Blumenstock
Christian Ehrhardt

Abteilung Angewandte Informationsverarbeitung

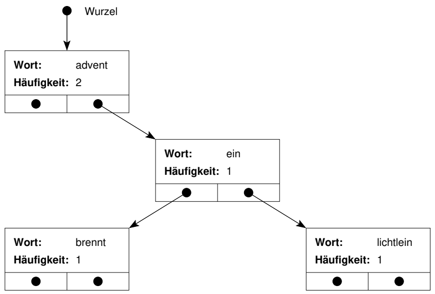
10. Januar 2006
Blatt 10

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

Abgabetermin: 17. Dezember 2005

Hash Mich (10 Punkte)

In Blatt 8 haben wir einen Worthäufigkeitszähler implementiert. Als Datenstruktur lag ein dynamischer Binärbaum zu Grunde, in dem jedes gelesene Wort nachgeschlagen werden konnte. Falls es gefunden wurde, wurde ein assoziierter Zähler inkrementiert, ansonsten ein neuer Eintrag mit dem Zählerstand 1 angelegt. Schließlich wurde der Baum durchlaufen und (Teile davon) als Worthäufigkeitstabelle ausgegeben.



Die in diesem Ablauf notwendigen Operationen setzen keineswegs voraus, dass die Datenstruktur ein Binärbaum ist. Wir können Vorbedingungen und Effekte dieser Operationen ohne Bezug auf die konkrete Implementierung beschreiben und zu einem abstrakten Datentyp namens *Map* (oder: Abbildung) zusammenfassen.
Zweck einer solchen Map ist, eine Menge von Schlüssel-Werte-Paaren so zu verwalten, dass jederzeit ein neuer Wert mit einem Schlüssel assoziiert werden sowie der zu einem Schlüssel gehörende Wert nachgeschlagen werden kann. Um unsere Implementierung nicht mehr auf eine Abbildung von Wörtern auf Häufigkeiten (Typ: `char * -> int`) festzulegen, sondern für Daten beliebigen Typs offen zu halten, verwenden wir für Schlüssel und Werte nunmehr untypisierte Zeiger (also `void *`).
Im einzelnen soll unser Interface folgende Operationen umfassen:

- `Map *mapCreate(Hasher, Comparator)`
  erzeugt eine neue Map. Die beiden übergebenen Funktionen[1] werden von der Map verwendet, um einen Schlüssel (der ja jetzt nicht mehr unbedingt ein String sein muss) auf einen Hashwert abzubilden, bzw. zwei Schlüssel zu vergleichen.
- `void mapPut(Map *map, void *key, void *value)`
  assoziiert den gegebenen Wert *value* mit dem Schlüssel *key*. (Weder Wert noch Schlüssel werden dabei kopiert. Der Schlüssel darf danach nicht mehr verändert werden.) Ist bereits ein Eintrag mit demselben Schlüssel vorhanden, wird dieser überschrieben.
- `void *mapGet(Map *map, void *key)`
  liefert den zuletzt mit *key* assoziierten Wert zurück, oder NULL, falls kein passender Eintrag gefunden wurde.
- `int mapSize(Map *map)`
  gibt die Zahl der Einträge in der gegebenen Map zurück.
- `void mapDump(Map *map, MapEntry **array)`
  legt Zeiger auf alle Einträge der Map in dem gegebenen *array* ab. Die Reihenfolge ist nicht spezifiziert. An dieser Stelle werden die Map-Einträge selbst (`struct`s mit zwei untypisierten Zeigern) sichtbar.
- `void mapFree(Map *map)`
  gibt den Speicher der gesamten Map-Struktur sowie sämtlicher Inhalte[2] frei.
- `char *mapIdentify()`
  gibt lediglich einen String wie „tree map" zur Identifikation der Implementierung zurück und sorgt für ein kleines schnelles Erfolgserlebnis.

Im Beispielverzeichnis zu diesem Blatt finden Sie zwei Dateien: die Headerdatei `map.h`, die im Wesentlichen obiges Map-Interface enthält, sowie `wordfreq.c`, welches dieses Interface anwendet, um analog zu Blatt 8 einen Worthäufigkeitszähler zu realisieren.

(a) *(7 Punkte)* Implementieren Sie in einer Datei `treemap.c` dieses Interface als unbalancierter Binärbaum, ganz analog zu Blatt 8. Die Hashfunktion benötigen Sie hier natürlich nicht. Funktionen wie `createEntry()`, die nicht speziell mit Binärbäumen zu tun haben, sollten Sie in einer weiteren Quelltextdatei (`map.c`) ablegen.

(b) *(3 Punkte)* Schreiben Sie ein Makefile. Es soll insbesondere für das Target `wf-tree` den Worthäufigkeitszähler mit der Objektdatei `treemap.o` compilieren.

(c) *(Zusatzaufgabe, 5 Punkte)* Implementieren Sie dasselbe Interface aus `map.h` als Hashtabelle in `hashmap.c` und erweitern Sie das Makefile entsprechend, so dass für das Target `wf-hash` die Objektdatei `hashmap.o` *ohne* `treemap.o` herangezogen wird. Auf diese Weise umgehen wir Namenskonflikte.

[1]Deklaration: `typedef int (*Hasher)(void *)` bzw. `typedef int (*Comparator)(void *, void *)`
[2]In praktischen Anwendungen wäre die Freigabe der Inhalte möglicherweise nicht gewünscht.

Die genaue Umsetzung einer Hashtabelle sei Ihnen überlassen. Sie können sich aber an folgenden Eckpunkten orientieren:

- Verwenden Sie ein Array mit N Einträgen. Da N mit dem unten gewählten s teilerfremd sein soll, bietet sich an, hier eine Primzahl zu verwenden. Alternativ beispielsweise eine Zweierpotenz, wenn unten s immer ungerade gewählt wird.
- Beim Eintragen oder Nachschlagen einer Assoziation bilden Sie aus dem Schlüssel den Hashwert, der modulo N einen Index i in das Array ergibt.
- Ist die entsprechende Zelle leer (beim Eintragen) bzw. werden (beim Nachschlagen) die Schlüssel von der Comparator-Funktion als identisch ausgewiesen, ist die Suche beendet.
- Ansonsten leiten wir aus dem Hashwert eine Schrittweite s ab und setzen unsere Suche an den Stellen $(i + k \cdot s) \mod N$ mit $k = 1, 2, \ldots$ fort. Wenn N zu s teilerfremd ist und mindestens eine freie Zelle im Array existiert, muss diese Iteration nach spätestens N Schritten abbrechen. (Wir haben dann alle Zellen besucht).
- Ist das Array nach einem `mapPut()` zu mehr als 75 % gefüllt, verdoppeln wir seine Größe (in etwa; N muss ggf. wieder prim sein) und nehmen alle Einträge neu vor.

Viel Erfolg!