Prof. Franz Schweiggert
Christian Ehrhardt
Abteilung Angewandte Informationsverarbeitung
6. Dezember 2004
Blatt 7

Allgemeine Informatik 3 (WS 2004/2005)

Abgabetermin 21.12.2004

Priority Queue (20 Punkte)

Eine Priority Queue ist eine Datenstruktur, die die folgenden Operationen (in effizienter Weise) erlaubt:

INSERT Ein (im allgemeinen beliebiges) Objekt wird zu der Datenstruktur hinzugefügt. Dabei wird zusätzlich eine positive Zahl – die Priorität des Elements – angegeben.

GET Bei dieser Operation wird das Element mit der höchsten Priorität aus der Datenstruktur entfernt und zurückgeliefert.

EMPTY Diese Operation liefert TRUE, wenn keine Elemente in der Priority Queue sind.

Das Ganze funktioniert also fast wie eine normale Queue oder wie ein Stack: Man kann Elemente einfügen und wieder entfernen. Allerdings werden die Elemente bei einer normalen Queue genau in der Reihenfolge entfernt, wie sie eingefügt wurden, und bei einem Stack wird das Element, das zuletzt eingefügt wurde zuerst entfernt. Bei einer Priority Queue wird dagegen immer das Element mit der zur Zeit höchsten Priorität entfernt.
In diesem Blatt sollen verschiedene Implementierungen einer solchen Datenstruktur im gleichen Programm implementiert, parallel zueinander verwendet und verglichen werden. Dabei soll das Programm sinnvoll in verschiedene Module aufgeteilt werden.
Der Einfachheit halber sind die eingefügten Elemente in diesem Blatt nur ganze Zahlen und der Wert der Zahl ist gleichzeitig die Priorität.

Zwei Implementierungen

Ein Array als Priority Queue

Bei dieser Implementierung werden die eingefügten Elemente in einem (dynamisch wachsenden!) Array unsortiert gespeichert. Neu hinzukommende Elemente werden am Ende angefügt, das Element mit der höchsten Priorität muß bei GET erst gesucht werden. Beim Entfernen eines Elements aus dem Array kann der frei werdende Platz einfach mit dem letzten Eintrag im Array aufgefüllt werden.

Eine sortierte Liste als Priority Queue

Bei dieser Variante werden alle Elemente in einer nach Priorität sortierten Liste verwaltet. GET nimmt einfach das erste Element der Liste. Bei INSERT muß der geeignete Platz in der Liste erst gesucht werden.

Zwei Module (6 Punkte)

Schreibt für die beiden vorgestellten Implementierungen je ein Modul, das eine Priority Queue entsprechend implementiert. Beide Implementierungen sollen gleichzeitig in einem einzigen Programm verwendet werden können. Testet Eure Implementierungen mit einem geeigneten Hauptprogramm.
Wie verhält sich bei den beiden Implementierungen der Aufwand für INSERT bzw. GET zur Anzahl der Elemente in der Queue.

Ein Interface (4 Punkte)

Jetzt sollen beide Implementierungen auf die gleiche Art und Weise verwendet werden können. Ob die Arrayimplementierung oder die Listenimplementierung verwendet wird, soll nur noch beim Erzeugen der Queue eine Rolle spielen. Bei den drei Operationen INSERT, GET und EMPTY soll dem Aufruf nicht mehr anzusehen sein, welche Implementierung verwendet wird. So könnte also ein Fragment des Hauptprogramms aussehen:

```
/* Eine mit Hilfe eines Arrays implementierte
 * Priority Queue erzeugen.
 */
q1 = array_priority_queue_create ();

/* Eine weiter mit Hilfe einer Liste
 * implementierte Priority Queue erzeugen.
 */
q2  = list_priority_queue_create ();

/* Das Element 30 in die Queue q1 einfuegen. */
insert (q1, 30);

/* Das Element 10 in die Queue q2 einfuegen. */
insert (q2, 10);

/* Das Element mit der h"ochsten Priorit"at
 * aus der Queue q2 holen. ==> ret = 10
 */
ret = get (q2);

/* Das Element mit der h"ochsten Priorit"at
 * aus der Queue q1 holen. ==> ret = 30
```

```
  */
 ret = get (q1);
```

Die Funktionen `array_priority_queue_create` und `list_priority_queue_create` werden von der jeweiligen Implementierung zur Verfügung gestellt, `insert`, `get` und `empty` von dem neu geschriebenen Interface.
Hinweis: Funktionszeiger und Zeiger vom Typ `void *` können bei diesem Teil sehr nützlich sein.

Eine effiziente Implementierung (6 Punkte)

Wird eine wirklich effiziente Implementierung benötigt, so wird meinst ein etwas anderes Verfahren verwendet (N ist die Anzahl der Elemente, die Moment in der Priority Queue sind):

- Die Elemente werden wie oben ohne "Lücken" in einem Array abgespeichert, der kleinste gültige Index ist also 0, der größte ist $N-1$.
- $Prio(x)$ bezeichnet die Priorität des Elements mit Index x.
- Für jeden gültigen Index $i > 0$ gilt dann $Prio(i) \leq Prio((i-1)\ \textbf{DIV}\ 2)$.
- Anders formuliert bedeutet das, daß für alle i gilt:
  $Prio(i) \geq Prio(2 \cdot i + 1)$ und $Prio(i) \geq Prio(2 \cdot i + 2)$.
- Durch diese Bedingungen wird sicher gestellt, daß es kein Element gibt, das eine höhere Priorität hat als das an Index 0.
- Ist i der Index eines Elements, dann bezeichnet:

  $\boldsymbol{Parent(i)}$ den Index $(i-1)$ **DIV** 2,

  $\boldsymbol{Left(i)}$ den Index $2 \cdot i + 1$ und

  $\boldsymbol{Right(i)}$ den Index $2 \cdot i + 2$.
- **Anmerkung:** Das Array kann auch als Binärbaum interpretiert werden, wobei $Left(i)$ die Wurzel des linken Teilbaums und $Right(i)$ die Wurzel des rechten Teilsbaums von i ist.
- Bei INSERT wird das neue Element zunächst ganz am Ende eingefügt. Wenn i der Index ist, an dem das neue Element im Moment gerade steht, dann werden die Priorität von i und die Priorität von $Parent(i)$ miteinander verglichen. Ist die Priorität von $Parent(i)$ kleiner als die von i, dann werden die beiden Elemente vertauscht. Das wird so lange fortgesetzt, bis $i = 0$ oder $Prio(Parent(i)) \geq Prio(i)$ ist. Dabei ist i immer der Index, an dem sich das neue Element im Moment befindet, und der Wert ändert sich natürlich durch eine Vertauschung. Erklärt Eurem Tutor, warum die oben angegebenen Ungleichungen anschließend wieder im gesamten Array gelten.
- Bei GET ist ja bereits bekannt, daß das Element an Index 0 die höchste Priorität hat. Um dieses Element aus dem Array zu entfernen wird es zunächst mit dem letzten Element vertauscht und das dann letzte Element wird entfernt. Anschließend muß dafür gesorgt werden, daß die Ungleichungen wieder gelten. Sei i jetzt der jeweils aktuelle Index des Elements, das sich zu Beginn an Index 0 befindet (durch Vertauschungen ändert sich der Index später). Dann wird die Priorität $Prio(i)$ mit $Prio(Left(i))$ und $Prio(Right(i))$ verglichen. Ist $Prio(i)$ der größte der drei Werte, dann sind wir fertig. Ansonsten wird das Element an Index i mit dem der beiden anderen Elemente mit der größten Priorität vertauscht. Das wird wieder so lange wiederholt, bis $Prio(i) \geq Prio(Left(i))$ und $Prio(i) \geq Prio(Right(i))$ ist. Macht Euch und Eurem Tutor auch hier wieder klar, daß dann die oben angegebenen Ungleichungen wieder für alle Elemente im Array gelten.

Schreibt ein weiteres Modul, das eine Priority Queue wie oben beschrieben implementiert und überlegt auch hier, wie sich der Aufwand für INSERT bzw. GET zur Anzahl der Elemente in der Queue verhält.
Anmerkung: Diese Datenstruktur wird in der Literatur in der Regel **Heap** genannt. Trotz der Namensgleichheit hat diese Datenstruktur aber nur wenig mit dem Begriff **Heap** zu tun, der im Zusammenhang mit Speicherverwaltung und `malloc` auftaucht.

Ein Makefile (4 Punkte)

Nachdem das gesamte Projekt jetzt aus ca. 4 Modulen und einem Hauptprogramm besteht, lohnt es sich, ein Makefile dafür zu schreiben. Eventuell ist es auch sinnvoll, diesen Teil der Aufgabe etwas früher anzupacken.