Universität des Saarlandes
FR 6.2 Informatik
Prof. Dr. Hans-Peter Lenhof
PD Dr. Peter Sanders
Dipl. Inform. Andreas Hildebrandt

Programmierung II, SS 2003
Musterlösung zur 5. Übung

Aufgabe 1: Unbounded Arrays, Amortisierte Analyse (10 Punkte)
Wir nehmen der Einfachheit halber zunächst mal an, dass es sich bei m um eine Zweierpotenz handelt.

Wir suchen eine Sequenz $\sigma :=< \sigma_1, \ldots, \sigma_m >$, die zu möglichst hohen Kosten führt. Die teuerste Operation die wir zur Verfügung haben ist das Vergrößern bzw. Verkleinern des Arrays. Laßst man andauernd push und pop aufeinander folgen, so erhält man zwar eine Sequenz mit maximaler Anzahl an solchen resize - Operationen, da aber jedesmal nur ein einziges Element kopiert werden muss, sind die Kosten der gesamten Sequenz in $\mathcal{O}(m)$, und damit zu klein.
Wir müssen also einen Kompromiss finden, der unser Array zuerst auf eine gewisse Größe bringt und trotzdem noch *genügend* resize-Operationen ermöglicht.
Dazu füllen wir mit den ersten $\frac{m}{2}$ Operationen das Array. Dies führt schon mal zu Kosten von $\frac{m}{2}$ für das Setzen der Elemente plus den resize - Kosten in dieser Teilsequenz. Da für das Array immer doppelt so viel Platz angelegt wird wie aktuell benötigt, wird nach jeder Zweierpotenz von push Operationen ein resize durchgeführt, also in der $1., 2., 4., \ldots, \frac{m}{2}$. Operation. Also sind die Kosten für diesen Teil von σ

$$\begin{aligned} T_1(m) &= \underbrace{\frac{m}{2}}_{\text{Setzen der Elemente}} + \underbrace{2 + 4 + \cdots + \frac{m}{2}}_{\text{Resize}} \\ &= \frac{m}{2} + \sum_{i=1}^{\log_2(\frac{m}{2})} 2^i \\ &= \frac{m}{2} + m - 2 \end{aligned}$$

Danach führen wir immer abwechselnd ein popBack und ein pushBack aus. In diesem Teil müssen also in jedem Schritt mindestens $\frac{m}{2}$ (bzw. bei den popBack's $\frac{m}{2} - 1$) Elemente kopiert werden. Dies passiert $\frac{m}{2}$ - mal.

$$\Rightarrow T(m) = T_1(m) + \frac{m}{2}\frac{m}{2} = \frac{m}{2} + m - 2 + \frac{m^2}{4} \in \Theta(m^2)$$

.
Was passiert nun, wenn m keine Zweierpotenz sein sollte? Zweierpotenz p zu m wählen: $p := 2^{\lfloor \log_2(m) \rfloor}$ und die Sequenz, die wir uns oben überlegt haben, mit der Länge p ausführen. Damit haben wir schon mal Kosten von $\Theta(p^2)$. $\lfloor \log_2(m) \rfloor$ liegt natürlich in $\Theta(\log_2(m))$, und damit können wir auch leicht zeigen, dass $p \in \Theta(m)$ und $p^2 \in \Theta(m^2)$ liegt. Nun müssen wir nur noch die restlichen $m - p$ Operationen verwenden (dies sind

offensichtlich $\Theta(1)$ viele, da ja $p \in \Theta(m)$). Mit diesen machen wir nun irgendetwas unkritisches, z.B. eine Folge von pushBack und popBack unterhalb der Schwelle, bei der vergrößert oder verkleinert wird. Dadurch fallen diese Kosten in der Berechnung nicht mehr ins Gewicht, und wir sind fertig.

Aufgabe 2: Binäre Zähler, Amortisierte Analyse (15 Punkte)

(a) Was ist das schlimmste was uns hier passieren kann? Sehen wir uns eine typische Sequenz von Inkrementen an:

$$0 \to 1 \to 10 \to 11 \to 100 \to 101 \to 110 \to 111 \to 1000$$

Unangenehm wird es offensichtlich vor allem dann, wenn wir zur nächsten Zweierpotenz springen müssen. Dann müssen alle bisherigen bits verändert und ein neues bit eingefügt werden. In diesem schlimmsten Fall ist also m eine Zweierpotenz, $\exists i \in \mathbb{N} : m = 2^i$. Diese wird dargestellt durch $i + 1$ bits $\Rightarrow T(m) \in \mathcal{O}(i + 1) = \mathcal{O}(i) = \mathcal{O}(log_2(m))$ Falls m keine Zweierpotenz sein sollte, dann wird m dargestellt durch $\lceil \log_2(m) \rceil$ bits, und damit liegt natürlich auch in diesem Fall die Laufzeit in $\mathcal{O}(log_2(m))$.

(b) Wir verwenden hier wieder die sogenannte *Bankkontomethode* zur amortisierten Analyse. Diese läuft typischerweise in den folgenden Schritten ab:

- Identifikation der elementaren Operationen. Diese sollten in $\mathcal{O}(1)$ ablaufen.
- Welche der elementaren Operationen werden in jedem Schritt $\mathcal{O}(1)$ mal durchgeführt? Diese Operationen nennen wir *constant bounded*, die anderen *non-constant bounded*.
- Nun brauchen wir eine eins-zu-eins Abbildung aller non-constant bounded-Operationen auf constant-bounded Operationen. (Die constant-bounded *zahlen ein*, die non-constant bounded *heben ab*).

Wie können wir das auf unseren binären Zähler anwenden? Gehen wir die Schritte einzeln durch.

- Elementare Operationen: ein bit kann entweder *gesetzt* oder *gelöscht* werden. Beides geht offensichtlich in $\mathcal{O}(1)$, und dies sind die einzigen Elementaroperationen die wir benötigen (das eventuelle Anhängen eines zusätzlichen bits beim Springen auf die nächste Zweierpotenz betrachten wir auch als Setzen eines bits).
- Identifikation der *constant bounded* Operationen: **Beobachtung:** Wenn der Zähler immer nur inkrementiert wird, dann wird in jedem Schritt *maximal ein einziges bit* gesetzt. $\Rightarrow$ die Elementaroperation *bit setzen* ist *constant bounded*.
- Eins-zu-eins Abbildung. Wir müssen also eine Möglichkeit finden, jedes *Löschen* eines bit (non-constant bounded) schon beim *Setzen* mitzuzählen. Dann müssen wir es nicht mehr separat berücksichtigen. **Idee:** jedes bit, das gelöscht werden soll, muss vorher einmal gesetzt worden sein. Und damit haben wir unsere

eins-zu-eins - Abbildung gefunden. Jedes setzen eines bit wird doppelt gezählt und *bezahlt* damit schon die Kosten für die spätere Löschung mit. Da in jedem der m Schritte maximal ein bit gesetzt wird, benötigen wir also für die gesamte Sequenz maximal $2m \in \mathcal{O}(m)$ Operationen, und damit sind die amortisierten Kosten jeder der m Operationen, also die Kosten der gesamten Sequenz geteilt durch die Anzahl der Operationen, $\frac{2m}{m} = 2 \in \mathcal{O}(1)$

(c) Die Vorgehensweise ist hier ähnlich wie bei Aufgabe 1. Wir versuchen möglichst häufig *schlimme* Fälle zu konstruieren. $\Rightarrow$ zuerst Inkrementieren wir c bis auf $\frac{m}{2}$. Die Laufzeit dieses Teils der Sequenz, $T_1(m)$, liegt nach Teil (b) in $\mathcal{O}(\frac{m}{2})$. Danach führen wir immer abwechselnd Inkremente und Dekremente durch, so dass in jedem der Fälle $\lceil \log_2(\frac{m}{2}) \rceil$ bits verändert werden müssen (Bsp.: $0 \to 1 \to 10 \to 11 \to 100 \to 101 \to 110 \to 101 \to 110 \to 101 \ldots$). Dies geschieht $\frac{m}{2}$-mal.

$$\begin{aligned} \Rightarrow T(m) &= T_1(m) + \frac{m}{2}\lceil \log_2(\frac{m}{2}) \rceil \\ &= T_1(m) + \frac{m}{2}\lceil \log_2(m) - \log_2(2) \rceil \end{aligned}$$

Und da $T_1(m) \in \mathcal{O}(\log_2(\frac{m}{2})) = \mathcal{O}(\log_2(m))$, folgt

$$T(m) \in \Theta(m \log_2(m))$$

(d) Die einfachste Variante besteht hier in der Verwendung einer Art *Strichliste*. Dazu verwendet man einen Stack, auf den bei jedem Inkrement eine 1 gepusht wird und bei jedem Dekrement eine 1 entfernt wird. Dies geht in konstanter Zeit, verbraucht aber offensichtlich $\mathcal{O}(m)$ Speicher. Das *Auslesen* dieses Zählers ist natürlich nun auch teuer, weshalb dieser Zähler fast nie in Software verwendet wird. In einigen Anwendungen wird ein solcher Zähler jedoch direkt in Hardware implementiert.

Aufgabe 3: Implementierung einer FIFO Warteschlange (*queue*) (15 Punkte)
Der folgende Code steht in der Datei `slist.h`:

```
// Aus Gruenden der Einfachheit _ohne_ free list

#ifndef SLIST
#define SLIST

#include <assert.h>


template<class T>
class slist {

  class node;
  node hdr;
```

```
public:

class node {
  friend class slist;
  node* next;
  T inf;
  public:
  node() { }
  node(node* x, T i) : next(x), inf(i) {}
};

slist() { hdr.next = &hdr; }
~slist() { make_empty(); }

// Ueberprueft, ob die Liste leer ist.
bool is_empty() const { return ( hdr.next == &hdr ); }

// Gibt den Header (das Element _vor_ dem ersten Element) zurueck.
node* header()          const { return const_cast<node*>(&hdr); }

// Ueberprueft, ob ein gegebenes Element der header ist.
bool is_header(node* p) const { return ( p == &hdr); }

// Gibt einen Pointer auf das erste gespeicherte Element zurueck
node* first()           const { return hdr.next; }

// Ueberprueft, ob wir am letzten Element angekommen sind.
bool is_last(node* p)   const { return (p->next == &hdr); }

// Gibt das naechste Element zurueck.
node* next(node* p)    const { return p->next; }

// Gibt den Inhalt eines Elements zurueck.
const T& operator[](node* p)  const      { return p->inf; }
T& operator[](node* p)        { return p->inf; }

// Einfuegen _nach_ dem Element, auf das p zeigt.
node* insert_after(node* p, const T& e)
{ p->next = new node(p->next, e);
  return p->next;
}

// Vorne einfuegen.
node* add_at_front(const T& e)
{ return insert_after(&hdr,e); }

// Loeschen _nach_ dem Element, auf das p zeigt.
void delete_after(node* p)
```

```
  { node* x = p->next;
    assert(x != &hdr);
    p->next = x->next;
    delete x;
  }

  // Loeschen des ersten Elements.
  void delete_first() { delete_after(&hdr); }

  // Liste leeren.
  void make_empty() { while (hdr.next != &hdr) { delete_first(); } }

  // "Sichere" Version von "next"
  node* succ(node* p)
  { if ( is_last(p) ) return nil;
    return p->next;
  }

  // Splice-Funktion
  void splice(node* insert_after_me, node* from_after_me, node* until)
  { if (from_after_me == until) return;
    node*        x         =   insert_after_me->next;
    insert_after_me->next   = from_after_me->next;
    from_after_me->next     =     until->next;
    until->next             = x;
  }

  private:

  slist(const slist& other);            // hide copy constructor
  slist& operator=(const slist& other); // hide assignment
};

template <class T>
class queue
{
  public:

    queue() {last_ = 0;}
    ~queue() {}

    void pushBack( const T& e ) { if (last_ == 0) last_ = list_.header();
      last_ = list_.insert_after( last_, e ); }

    T popFront() { T result = list_[list_.first()];
      list_.delete_first(); return result; }
    T first() { return list_[list_.first()]; }
```

```
  private:
    slist<T>::node* last_;
    slist<T> list_;
};

#endif
```

Ein kleines Testprogramm dazu gibt es in `slist.cpp`

```
#include "slist.h"

#include <iostream>

int main(int argc, char **argv)
{
  queue<int> q;

  q.pushBack(1);
  q.pushBack(2);
  q.pushBack(3);

  std::cout << q.popFront() << " "
            << q.popFront() << " "
            << q.popFront() << std::endl;

  return 0;
}
```

Aufgabe 4. (Unbounded Arrays Bonuspunkte)
10 Wir gehen in dieser Aufgabe davon aus, dass wir ein Array in konstanter Zeit *verkleinern* können, ohne es im Speicher umkopieren zu müssen. Dies lässt sich z.B. durch Überladen von `delete` unter Verwendung von `realloc` erreichen, hat aber den Nachteil, dass der vorhandene Speicher u.U. wüst fragmentiert wird.
Die Idee besteht hier darin, zusätzlich zum *eigentlichen* Array ein 2. Array doppelter Grösse bereitzuhalten, und bei jedem pushBack schon ein Element aus dem alten Array in das neue mitzukopieren.
Beim Verkleinern hingegen wird einfach der nicht mehr benötigte Speicher freigegeben.

Abgabe: 30.05.2003