# 2. Übung Informatik I

Marcus Rickert

30. Dezember 1995

## 1 Aufgabe 1

### 1.1 Erläuterungen

Bei den Prozeduren *clear_queue, clear_stack, insert_queue* und *insert_stack* werden keine Plausibilitätsüberprüfungen durchgeführt. Es ist also darauf zu achten, daß die Stacks und Queues entsprechend groß genug sind.

#### 1.1.1 Die Prozeduren *clear_stack* und *insert_stack*

Als Übergabeparameter erwarten die beiden Prozeduren jeweils den Stapel, den Stapelzeiger und die Nummer der Einfüge- bzw. Löschstelle, wobei das Stapelelement, das als letztes gepusht worden ist, die Nummer 1 erhält. Die Prozedur *insert_stack* braucht zusätzlich noch den einzusetzenden Wert, der den Platz der Einfügestelle einnimmt.

#### 1.1.2 Die Prozeduren *clear_queue* und *insert_queue*

Als Übergabeparameter erwarten die beiden Prozeduren jeweils die Queue, den Anfangszeiger, den Endzeiger, die Länge der Queue und die Nummer der Einfügestelle, wobei das Queueelement, das als nächtes gelesen wird, die Nummer 1 erhält. Die Prozedur *insert_queue* braucht zusätzlich noch den einzusetzenden Wert.

## 1.2 Bezeichnungen

| | |
|---|---|
| stapel | Stapel, der an Prozedur übergeben wird |
| sp | Stackzeiger obigen Stapels |
| queue | Queue, die an Prozedur übergeben wird |
| anfang,ende | Anfang und Ende obiger Queue |
| maxlaenge | maximale Länge obiger Queue |
| laenge | Länge der übergebenen Queue |
| stapel2 | lokaler Stapel |
| sp2 | Stackzeiger obigen Stapels |
| n | Nummer der Einfügestelle |
| memo | Zwischenspeicher für Stapelelemente |
| i | Zähler |

## 1.3 Algorithmen

```
PROCEDURE clear_stack (stapel,sp,n)
    BEGIN
    i:=1
    WHILE i<n               n − 1 Elemente vom Stapel holen
        BEGIN
        pop(stapel,sp,memo)
        push(stapel2,sp2,memo)          und zwischenspeichern
        i:=i+1
        END
    pop(stapel,sp,memo)             n-tes Element löschen
    i:=1
    WHILE i<n
        BEGIN
        pop(stapel2,sp2,memo)
        push(stapel,sp,memo)            Stapel wiederherstellen
        i:=i+1
        END
    END

PROCEDURE insert_stack (stapel,sp,n,neu)
    BEGIN
    i:=1
    WHILE i<n               n − 1 Elemente vom Stapel holen
        BEGIN
        pop(stapel,sp,memo)
        push(stapel2,sp2,memo)          und zwischenspeichern
        i:=i+1
        END
    push(stapel,sp,neu)             neuen Wert an n-te Stelle einsetzen
```

```
    i:=1
    WHILE i<n
        BEGIN
        pop(stapel2,sp2,memo)
        push(stapel,sp,memo)              Stapel wiederherstellen
        i:=i+1
        END
    END

PROCEDURE clear_queue (queue,anfang,ende,maxlaenge,n)
    BEGIN
    IF anfang<ende           bestimme Länge der Queue
        laenge:=maxlaenge+anfang-ende
    ELSE
        laenge:=anfang-ende
    i:=1
    WHILE i<n                übergehe die ersten n − 1 Elemente
        BEGIN
        lese(queue,anfang,ende,maxlaenge,memo)
        schreibe(queue,anfang,ende,maxlaenge,memo)
        i:=i+1
        END
    lese(queue,anfang,ende,maxlaenge,memo)          lösche n-tes Element
    WHILE (i<laenge)           übergehe die letzten laenge − n − 1 Elemente
        BEGIN
        lese(queue,anfang,ende,maxlaenge,memo)
        schreibe(queue,anfang,ende,maxlaenge,memo)
        i:=i+1
        END
    END

PROCEDURE insert_queue (queue,anfang,ende,maxlaenge,n,neu)
    BEGIN
    IF anfang<ende           bestimme Länge der Queue
        laenge:=maxlaenge+anfang-ende
    ELSE
        laenge:=anfang-ende
    i:=1
    WHILE i<n                übergehe die ersten n − 1 Elemente
        BEGIN
        lese(queue,anfang,ende,maxlaenge,memo)
        schreibe(queue,anfang,ende,maxlaenge,memo)
        i:=i+1
        END
    schreibe(queue,anfang,ende,maxlaenge,neu)          füge neues Element ein
    WHILE (i<laenge+1)           übergehe die letzten laenge − n Elemente
```

```
      BEGIN
      lese(queue,anfang,ende,maxlaenge,memo)
      schreibe(queue,anfang,ende,maxlaenge,memo)
      i:=i+1
      END
   END
```

# 2 Aufgabe 2

## 2.1 Erläuterungen

Der Algorithmus *invert* beruht darauf, daß die Elemente der Liste der Reihe nach ausgelesen und in einem Stapel zwischen gespeichert werden. Der von den Elementen der Liste belegte Speicher wird dabei freigegeben. Anschließend werden die Elemente aus dem Stapel ausgelesen und in eine neue — jetzt invertierte Liste — abgespeichert. Im Spezialfall einer leeren Liste wird nichts unternommen.

## 2.2 Bezeichnungen

| | |
|---|---|
| anfang | Zeiger auf erstes Listenelement |
| z | Zeiger auf aktuelles Listenelement |
| alt | Zeiger auf vorheriges Listenelement |
| stapel | Stack, der Datenelemente der Liste aufnehmen kann |
| sp | Stackzeiger |

## 2.3 Algorithmus

```
PROCEDURE invert (anfang)
   BEGIN
   IF anfang≠NIL
      BEGIN
      sp:=0              Stack initialisieren
      z:=anfang              Zeiger aufs erste Element
      WHILE z≠NIL
         BEGIN
         push(stapel,sp,z→daten)          Speichere Daten im Stapel
         alt:=z
         z:=z→zeiger            nächstes Element
         free(alt)          durch Element belegten Speicherplatz freigeben
         END
      anfang:=NIL            neue Liste als leer markieren
      WHILE sp>0
         BEGIN
```

```
      new(z)                Speicherplatz bereitstellen
      IF anfang=NIL              wenn erstes Element der neuen Liste
         anfang:=z             dann Anfangszeiger auf erstes Element setzen
      ELSE
         alt→zeiger:=z           sonst Element anhängen
      pop(stapel,sp,z→Daten)          Daten aus Stapel holen
      alt:=z
      END
   z→zeiger:=NIL            Ende der Liste markieren
   END
END
```

### 2.4 Laufzeit

Die beiden *WHILE*-Schleifen werden jeweils so oft durchlaufen wie die Liste lang ist, also hat der Algorithmus die Laufzeit $O(2 * n)$.

## 3 Aufgabe 3

### 3.1 Erläuterungen

Der Algorithmus ist so ausgelegt, daß die Elemente der Liste Y in die Liste X eingefügt werden. Dazu werden zwei Zeiger definiert, von denen der erste (*xlinks*) vor die Einfügestelle und der zweite (*srechts*) hinter die Einfügestelle zeigt. Solange die Liste Y noch nicht zu Ende ist, wird jeweils ein Element eingesetzt. Falls eine der beiden Listen zu Ende ist, wird die *WHILE*-Schleife verlassen. Falls noch Y-Elemente übrig sind, so werden diese an die Liste X angehängt. Sind nur noch X-Elemente vorhanden, so ist nichts zu tun.

### 3.2 Bezeichnungen

| | |
|---|---|
| xanfang | Zeiger auf erstes Element der Liste X |
| yanfang | Zeiger auf erstes Element der Liste Y |
| zanfang | Zeiger auf erstes Element der Liste Z |
| xlinks | Zeiger auf Element der Liste X, hinter dem ein Element der Liste Y eingefügt wird |
| xrechts | Zeiger auf Element der Liste X, vor dem ein Element der Liste Y eingefügt wird |
| y | Zeiger auf einzufügendes Element der Liste Y |
| ynext | Zeiger auf nächstes Y-Element |

## 3.3 Algorithmus

```
PROCEDURE merge (xanfang,yanfang,zanfang)
    BEGIN
    IF xanfang=NIL                  wenn Liste X leer
        zanfang:=yanfang            dann Liste Z gleich Liste Y
    ELSE
        BEGIN
        zanfang:=xanfang            beginne mit erstem Element der Liste X
        y:=yanfang
        xlinks:=xanfang
        xrechts:=xlinks→zeiger
        WHILE xrechts≠NIL and y≠NIL         solange nicht Ende von X oder Y
            BEGIN
            ynext:=y→zeiger
            y→zeiger:=xrechts       setze Y-Element zwischen xlinks und xrechts ein
            xlinks→zeiger:=y
            y:=ynext                nächstes Y-Element
            xlinks:=xrechts         Einfügestelle eins nach rechts schieben
            xrechts:=xrechts→zeiger
            END
        IF y≠NIL                    wenn noch Elemente in Liste Y
            xlinks→zeiger:=y        hänge Ende der Liste Y an Liste X an
        END
    END
```

## 3.4 Laufzeit

Die Laufzeit dieser Prozedur wird durch die kürzere der beiden Listen bestimmt, da die *WHILE*-Schleife verlassen wird, falls eine der Listen zu Ende ist. Wenn $m$ die Länge der ersten Liste und $n$ die der zweiten Liste ist, so beläuft sich die Rechenzeit auf $O(min(m,n))$.

# 4 Aufgabe 4

## 4.1 Erläuterungen

Da die Datenstruktur laut Aufgabenstellung frei wählbar war, habe ich mich für eine in C sehr oft gebrauchte Struktur entschieden. Es handelt sich dabei um Zeiger auf Felder (Arrays), für die dynamisch Speicherplatz belegt werden kann. Die einzelnen Elemente des Feldes werden wie gewohnt über Indices angesprochen. Für die Speicherverwaltung stehen die Befehle

- allocate(zeiger,anzahl) und
- reallocate(zeiger,anzahl)

zur Verfügung, wobei der erste soviel Speicherplatz reserviert, daß für *anzahl* Feldelemente Platz ist, und anschließend *zeiger* auf den Anfang des reservierten Bereiches setzt. Der zweite Befehl macht das gleiche, nur gibt er zusätzlich den vorher von *zeiger* belegten Speicher frei.

In dieser Aufgabe werden die einzelnen Koeffizienten des Polynoms in einem Feld der oben beschriebenen Struktur abgelegt. Dabei entspricht der Index der zum Koeffizienten gehörigen Potenz. Die Felder sind so angelegt, daß der höchste Koeffizient ungleich null ist und die *laenge* des Feldes gleich dem höchsten Koeffizienten plus 1 ist. Im Spezialfall, daß alle Koeffizienten gleich null sind, hat das Polynom die Länge 0.

### 4.1.1 Die Prozedur *add*

In dieser Prozedur wird zuerst als Länge des Ergebnispolynoms die maximale Länge der Ausgangspolynome angenommen. Anschließend werden die entsprechenden Koeffizienten addiert. Dabei wird der höchste Index, dessen Koeffizient ungleich 0 ist, zwischengespeichert. Am Ende der Prozedur wird das Polynom so gekürzt, daß der höchste Koeffizient wieder ungleich null ist.

### 4.1.2 Die Prozedur *multiply*

In dieser Prozedur wird zuerst überprüft, ob eine der beiden Polynome das Nullpolynom ist. Wenn dies der Fall ist, wird als Ergebnis ebenfalls das Nullpolynom zurückgegeben. Wenn beide Polynome mindestens den konstanten Term enthalten, werden die einzelnen Koeffizienten in zwei ineinander verschachtelten Schleifen miteinander multipliziert. Die Ergebnisse der Multiplikationen werden zu den jeweiligen Koeffizienten von *pol3* addiert, dessen Länge sich aus Summe der Längen der Ausgangspolynome ergibt.

## 4.2 Bezeichnugen

| | |
|---|---|
| polx | Zeiger auf das Polynom X |
| laengex | Laenge des Polynoms X |
| max | Höchster Index von *pol3*, der ungleich null ist |
| i,j,k | Zähler |

## 4.3 Algorithmen

**PROCEDURE add** (pol1,laenge1,pol2,laenge2,pol3,laenge3)
**BEGIN**

```
    laenge3:=max(laenge1,laenge2)
    allocate(pol3,laenge3)          belege Speicher für neues Polynom
    max:=−1
    i:=0
    WHILE i<laenge3
        BEGIN
        pol3[i]:=pol1[i]+pol2[i]          addiere Koeffizienten
        IF pol3[i]≠0
            max:=i          merke höchsten Koeffizienten ungleich null
        i:=i+1
        END
    reallocate(pol3,max+1)          kürze Polynom, wenn nötig
    END

PROCEDURE multiply (pol1,laenge1,pol2,laenge2,pol3,laenge3)
    BEGIN
    IF laenge1*laenge2=0
        allocate(pol3,0)          Nullpolynom als Ergebnis
    ELSE
        BEGIN
        laenge3:=laenge1+laenge2−1          neue Länge wird durch höchste Indices bestimmt
        allocate(pol3,laenge3)          belege Speicher für neues Polynom
        k:=0
        WHILE k<laenge3
            BEGIN
            pol3[k]:=0          alle Koeffizienten von pol3 auf null setzen
            k:=k+1
            END
        i:=0
        WHILE i<laenge1          Schleife mit allen Koeffizienten von pol1
            BEGIN
            j:=0
            WHILE j<laenge2          Schleife mit allen Koeffizienten von pol2
                BEGIN
                k:=i+j
                pol3[k]:=pol3[k]+pol1[i]*pol2[j]
                j:=j+1
                END
            i:=i+1
            END
        END
    END
```

## 4.4 Laufzeiten

Sei $n$ die Länge des Polynoms *pol1* und $m$ die Länge des Polynoms *pol2*.

### 4.4.1 Die Prozedur *add*

Die Laufzeit dieser Prozedur wird durch das Maximum der Längen der beiden Ausgangspolynome bestimmt. Die *WHILE*-Schleife wird $max(m, n)$-mal durchlaufen. Also ist die Laufzeit $O(max(m, n))$.

### 4.4.2 Die Prozedur *multiply*

Die Laufzeit dieser Prozedur wird durch Produkt der Längen der beiden Ausgangspolynome bestimmt, da zwei ineinander verschachtelte Schleifen abgearbeitet werden müssen. Also ist die Laufzeit $O(m * n)$.