Universität Innsbruck
Erste Prüfung

Institut für Informatik
3. Juli 2008, 8:15–9:45

Algorithmen und Datenstrukturen

Name:	
Matrikelnr:	

Die Prüfung besteht aus 7 Aufgaben. Die verfügbaren Punkte für jede Aufgabe sind am Rand angegeben. Um die Prüfung erfolgreich abzuschließen sind 40 von 80 Punkten ausreichend.

Aufgabe	Erreichte Punkte
1	8
2	12
3	15
4	9
5	10
6	8
7	18
Summe	80

1 C Programmieren

[8] 1. Welche Ausgabe produziert folgendes C Programm?

```
#include <stdio.h>

void deci(int* i)
{
  (*i)--;
}

void incp(char* s)
{
  s += 2;
}

int main()
{
  int i = 3, x = 2;
  char* s = "hello world!";

  deci(&i);
  printf("i = %d\n", i);

  incp(s);
  printf("%s\n", s);

  if (x = 3) {
    x = x / 2;
    printf("x = %d\n", x);
  }
  else {
    int x = 7;
    x = x / 2;
    printf("x = %d\n", x);
  }

  return 0;
}
```

Lösung.
Die Ausgabe ist:
i = 2
hello world!
x = 1

2 Auswertung von Ausdrücken in C

Bestimmen Sie für die folgenden Codefragmente, ob das Ergebnis der Ausgabe compilerabhängig ist. Geben Sie die möglichen Ausgaben an.

[4] (a)

```
if (1 || 1 && 0) printf("3 + 4 * 2 = %i", 3 + 4 * 2);
else   printf("7 - 14 / 2 = %i", 7 - 14 / 2);
```

Lösung.
Compilerabhängig: ☐ ja ☒ nein
Mögliche Ausgaben:
$3 + 4 * 2 = 11$

[4] (b)

```
int j = 0;
printf("(j == (j == 0)) = %i", j == (j == 0));
```

Lösung.
Compilerabhängig: ☐ ja ☒ nein
Mögliche Ausgaben:
$(j == (j == 0)) = 0$

[4] (c)

```
int a[2] = {2,2}; int i=0; a[i]=i++;
printf("a=[%i, %i]\n", a[0], a[1]);
```

Lösung.
Compilerabhängig: ☒ ja ☐ nein
Mögliche Ausgaben:
$a = [0, 2]$ oder $a = [2, 0]$

3 Verkettete Listen

Geben Sie Pseudocode für eine Vorgängerfunktion auf einfach verketteten Listen an. Diese Funktion soll als Eingabe zwei Zeiger erhalten, wobei *list* auf den Beginn der Liste und *elt* auf das gewünschte Element zeigt. Die Rückgabe der Funktion ist ein Zeiger auf den Vorgänger von *elt* in der gegebenen Liste.

Nehmen Sie dabei den Datentyp in Listing 1 an und vervollständigen Sie den in Listing 3 gegebenen Pseudocode:

[5] (a) Geben Sie die Spezifikation für die Ausgabe *pred* an.

[10] (b) Geben Sie den Algorithmus selbst an.

Listing 1 Verkettete Liste

```
record List =
begin
value: integer;
next: ^List;
end
```

Listing 2 Vorgänger eines Listenelements

Eingabe: *list*: Zeiger auf die Liste, *elt*: Zeiger auf ein Element
Ausgabe: *pred*: Zeiger auf den Vorgänger von *elt* in *list*, oder *nil*

```
begin




end
```

Lösung.

Listing 3 Vorgänger eines Listenelements

Eingabe: $list, elt$
Ausgabe: $pred$: Zeiger auf den Vorgänger von elt in $list$

```
1: begin
2: pred : ^List;
3: list = list.next;
4: while !list.value = elt do
5:    pred = list;
6:    list = list.next;
7: return pred;
8: end
```

4 Hashing mit offener Adressierung [9]

Gegeben sind die folgenden Werte:

$$[4, 24, 28, 60, 71, 21, 68, 73, 16].$$

Fügen Sie die oben angegebenen Werte in eine anfangs leere Hashtabelle der Größe 11 in der gegebenen Reihenfolge ein. Verwenden Sie lineares Hashing mit der folgenden Hashfunktion: $h(k, j) = ((h_1(k)+j) \mod 11)$, wobei $h_1(k) = k \mod 11$. Geben Sie alle Zwischenschritte sowie in der zweiten Tabelle die Anzahl und die Positionen der Kollisionen im jeweiligen Schritt an.

Lösung.

K	0	1	2	3	4	5	6	7	8	9	10
4					4						
24			24		4						
28			24		4		28				
60			24		4	60	28				
71			24		4	60	28	71			
21			24		4	60	28	71			21
68			24	68	4	60	28	71			21
73			24	68	4	60	28	71	73		21
16			24	68	4	60	28	71	73	16	21

Koll.	Pos.
0	
0	
0	
0	
2	5,6
0	
1	2
1	7
4	5,6,7,8

5 **Sortieralgorithmen**

[10] Kreuzen Sie an, für welche Sortieralgorithmen welche Eigenschaften zutreffen. Mehrfachnennungen sind möglich.

Eigenschaft	Selection sort	Insertion sort	Quick sort	Merge sort	Heap sort
Benötigter zusätzlicher Speicherplatz ist unabhängig von der Anzahl der Elemente					
Aufwand im worst case ist $O(n^2)$					
Die Anzahl der Vergleiche ist entscheidend für die Aufwandsklasse im average case					
Die Anzahl der Vertauschungen ist entscheidend für die Aufwandsklasse im average case					
Aufwand im average case ist $O(n \log n)$					
Der Sortieraufwand ist abhängig von der Anzahl der zu sortierenden Elemente					
Verwendet eine "teile & herrsche" Strategie					
Aufwand im worst case ist $\Theta(n \log n)$					
Aufwand im worst case ist $O(n \log n)$					
Aufwand im average case ist $\Theta(n^2)$					

Lösung.

Eigenschaft	Selection sort	Insertion sort	Quick sort	Merge sort	Heap sort
Benötigter zusätzlicher Speicherplatz ist unabhängig von der Anzahl der Elemente	X	X			X
Aufwand im worst case ist $O(n^2)$	X	X	X	X	X
Die Anzahl der Vergleiche ist entscheidend für die Aufwandsklasse im average case	X	X^*	X	X	X
Die Anzahl der Vertauschungen ist entscheidend für die Aufwandsklasse im average case					
Aufwand im average case ist $O(n \log n)$			X	X	X
Der Sortieraufwand ist abhängig von der Anzahl der zu sortierenden Elemente	X	X	X	X	X
Verwendet eine "teile & herrsche" Strategie			X	X	
Aufwand im worst case ist $\Theta(n \log n)$				X	X
Aufwand im worst case ist $O(n \log n)$				X	X
Aufwand im average case ist $\Theta(n^2)$	X	X			

(*) Beide Antwort sind Richtig

6 Analyse von Algorithmen

[5] (a) Betrachten Sie folgendes Code-Fragment und stellen Sie die exakte Rekursionsgleichung für die Anzahl $T(n)$ der **print** Ausgaben in Abhängigkeit vom Eingabeparameter n auf. Nehmen Sie dazu an, dass der Eingabeparameter n eine Zweierpotenz ist, d.h., $n = 2^k$ für $k \in \mathbb{N}$.

[3] (b) Geben Sie ausgehend von dieser Rekursionsgleichung eine asymptotische Schranke für die Anzahl der **print** Ausgaben an.

Algorithmus 4 *SayGoodbye*

Input: $n \in \mathbb{N}$

```
begin
if n < 1 then
  return;
for j := 1 to 2n do
  j := j + 1;
  print ''goodbye'';
call SayGoodbye(n/2);
end
```

Lösung.

$$T(n) = T(\frac{n}{2}) + 2n$$

mit Anfangsbedingung $T(1) = 2$. Falls n eine Zweierpotenz ist, ist diese Rekursion einfach zu lösen, und die explizite Lösung ist:

$$T(n) = 4n - 2$$

Also ist $T(n) \in \Theta(n)$, was man auch einfach mit Hilfe des Master Theorems ermitteln hätte können.

7 Heapkonstruktion und Heapsort

(a) [8] Verwenden Sie eine der in der Vorlesung vorgestellten Methoden, um für das Feld

$$A = [0, 7, 4, 6, 17, 5, 9, 11, 20].$$

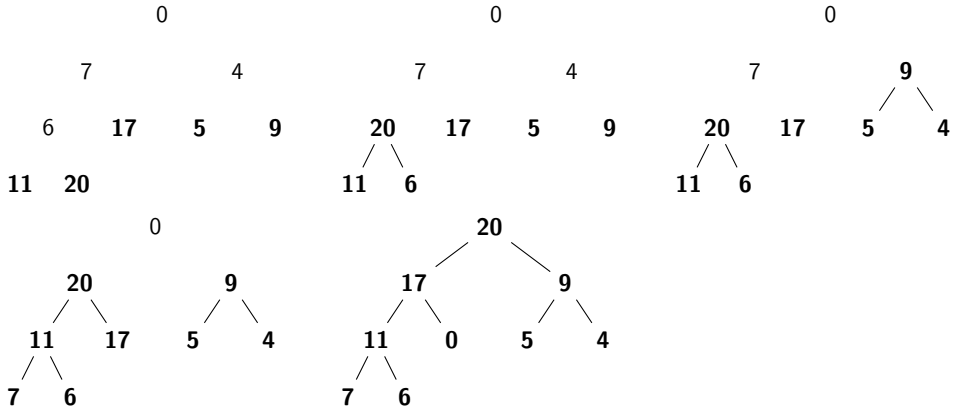
einen Heap *in-place* zu konstruieren.

Hinweis: Sie können den Heap sowie die Felder in den Zwischenschritten als Bäume darstellen.

Lösung.

Im folgenden wird der Heap *bottom-up* konstruiert. Zu Beginn werden alle Knoten $A[5, \ldots, 9]$ als Menge von Heaps mit je einem Knoten aufgefasst. Die Knoten an den Positionen 4, 3, 2 und 1 im Feld werden sukzessive hinzugefügt und sinken in den Heap ein.

Damit erhält man die folgenden Felder (in Baumdarstellung):



(b) [10] Verwenden Sie das Verfahren *HeapSort* um das gegebene Feld zu sortieren.

Lösung.

Ausgehend vom in der vorhergehenden Aufgabe konstruierten Heap erhält man die folgenden Felder: