Teilklausur II in Informatik

Es sind insgesamt 80 Punkte zu erreichen
(Jeder Punkt entspricht einer Bearbeitungszeit von etwa 1 Minute)

Aufgabe 1: Binäre Suchbäume (10 Punkte) – Einfügen, andere Reihenfolgen
Aufgabe 2: Sortieren (16 Punkte) – 2(+1) Verfahren + Quicksort
Aufgabe 3: Dijkstra-Algorithmus (15 Punkte) – Beschreiben und Durchführen
Aufgabe 4: Programm-Analyse (18 Punkte) – Durchführen, begründen, O-Notation
Aufgabe 5: Listen-Programmierung (21 Punkte) – Programmieren, O-Notation

Bitte kennzeichnen Sie jedes Blatt lesbar mit Ihrem Namen und Ihrer Matrikelnummer

Bearbeiten Sie die Aufgaben – soweit möglich – auf den Aufgabenblättern

Zugelassene Hilfsmittel: Keine

Lesen Sie die Aufgaben in Ruhe durch!
Viel Erfolg!

Aufgabe 1 — 10 Punkte

(6+2+2=10 Punkte) **Binäre Suchbäume:**

- Fügen Sie die Zahlen 18, 12, 5, 13, 21, 42 und 37 in einen anfangs leeren binären Suchbaum ein. Es reicht die Angabe des Suchbaums, Sie müssen nicht das Verfahren des Einfügens beschreiben.
- Geben Sie eine andere Reihenfolge der Zahlen an, so dass beim Einfügen der Zahlen in dieser Reihenfolge derselbe binäre Suchbaum wie in der ersten Teilaufgabe entsteht, oder begründen Sie, warum es keine weitere solche Reihenfolge der Zahlen geben kann.
- Geben Sie eine andere Reihenfolge der Zahlen an, so dass beim Einfügen der Zahlen in dieser Reihenfolge der resultierende binäre Suchbaum minimale Höhe hat.

Aufgabe 2 — 16 Punkte

(6+10 = 16 Punkte) **Sortieren:**

- (2*3 Punkte) Welcher Sortieralgorithmus wurde verwendet (falls es mehrere korrekte Antworten gibt, so genügt die Angabe von jeweils einem Algorithmus)? Begründungen sind hier nicht gefordert.

  1. Gegeben ist das Feld mit den Werten
     (14,21,52,63,18,42,37). Nach dem ersten Tausch (u.U. können vorher schon Vergleiche stattgefunden haben) hat das Feld den Inhalt
     (14,21,52,18,63,42,37), nach dem zweiten Tausch (auch hier können zwischenzeitlich weitere Vergleiche stattgefunden haben) hat das Feld den Inhalt
     (14,21,52,18,42,63,37).
  2. Gegeben ist das Feld mit den Werten
     (14,21,52,63,18,42,37). Nach dem ersten Tausch (u.U. können vorher schon Vergleiche stattgefunden haben) hat das Feld den Inhalt
     (14,21,52,18,63,42,37), nach dem zweiten Tausch (auch hier können zwischenzeitlich weitere Vergleiche stattgefunden haben) hat das Feld den Inhalt
     (14,21,18,52,63,42,37).

3. Zusatzaufgabe (2+1 Punkte): Gegeben ist das Feld mit den Werten
(14,21,52,63,18,42,37). Nach dem ersten Tausch (u.U. können vorher schon Vergleiche stattgefunden haben) hat das Feld den Inhalt
(14,63,52,21,18,42,37), nach dem zweiten Tausch (auch hier können zwischenzeitlich weitere Vergleiche stattgefunden haben) hat das Feld den Inhalt
(63,14,52,21,18,42,37).
Wie lautet der Inhalt nach dem dritten Tausch?

- (10 Punkte) Führen Sie mit dem Quicksort-Algorithmus den ersten Quicksort-Schritt (also ein komplettes Aufteilen in Elemente kleiner gleich und größer gleich dem Pivot-Element) in dem Feld mit den folgenden Zahlen durch. Wählen Sie als Pivot-Element das mittlere Element (42) und geben Sie an, welche beiden rekursiven Aufrufe nach diesem Quicksort-Schritt aufgerufen werden.

  Aus Ihrer Bearbeitung muss hervorgehen, welche Zahlen bei dem Prozess vertauscht wurden und wie diese zu vertauschenden Zahlen gefunden wurden. Ebenso muss ersichtlich sein, wie Sie die Parameter der beiden rekursiven Aufrufe ermittelt wurden.

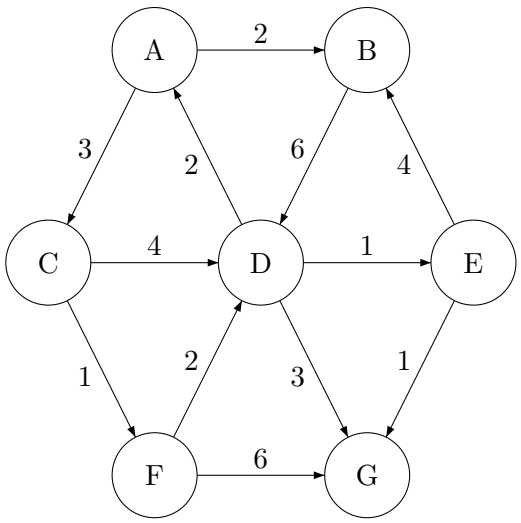
  14 21 52 63 18 42 37 42 86 42 13

Aufgabe 3 — 15 Punkte

(6+9 = 15 Punkte) **Kürzeste-Wege-Berechnung mit dem Algorithmus von Dijkstra:**

(6 Punkte) Beschreiben Sie in 3-4 Sätzen das Vorgehen beim Dijkstra-Algorithmus (Sie müssen dabei nicht begründen, warum das Vorgehen korrekt ist).

(9 Punkte) Führen Sie nun den Dijkstra-Algorithmus auf folgendem Graphen aus:
Zu Beginn ist lediglich die Entfernung zum Startknoten A mit d(A)=0 bekannt. Im Dijkstra-Algorithmus wird in jeder Iteration ein Knoten aus der Randmenge R entfernt. Geben Sie den Zustand der Menge R jeweils vor diesem Entfernen eines Knotens an. Geben Sie die berechnete Entfernung d() für alle Knoten an (die Vorgänger-Knoten pred() müssen nicht mit berechnet werden).



Aufgabe 4 — 18 Punkte

(8+7+3 = 18 Punkte) **Analyse eines Sortieralgorithmus:** Sie haben in einem Buch folgenden Sortieralgorithmus gefunden:

```
const int N = 7;
int tosort[N] = {5,8,2,1,4,9,6};
```

```
void UnknownSort() {
  int z,h;
  for(int i=1;i<N;i++) {
    z=0;
    for(int j=0;j<i;j++) { if (tosort[j]<tosort[i]) z=z+1; } // zähle "<tosort[i]"
    h=tosort[i];
    for(int k=i;k>z;k--) { tosort[k]=tosort[k-1]; } // verschiebe
    tosort[z]=h;
  }
}
```

Dieser sortiert – so steht es im Buch – das Feld `tosort[]`. Die Aussage soll überprüft und die Laufzeit analysiert werden.

- (8 Punkte) Führen Sie den Algorithmus aus und geben Sie den Inhalt des Feldes `tosort[]` nach jedem Durchlauf der äußeren Schleife an. Beachten Sie dabei, dass in C/C++ Arrays mit Index 0 beginnen.
- (7 Punkte) Formulieren Sie die Idee, die hinter dem Algorithmus steckt, und begründen Sie, warum der Algorithmus tatsächlich sortiert.
- (3 Punkte) Geben Sie die Laufzeit des Algorithmus in O-Notation an.

Aufgabe 5 — 21 Punkte

(3+18 = 21 Punkte) **Listen-Programmierung:** Wir betrachten in dieser Aufgabe einfach verkettete Listen aus `int`-Zahlen, die sortiert sind. Der Anker einer Liste ist ein Zeiger auf das erste Element der Liste (oder 0, falls die Liste leer ist). Ihre Aufgabe ist, eine Funktion zu schreiben, die zwei Anker auf zwei sortierte einfach verkettete Listen erhält und den Anker auf eine neue Liste als Ergebnis zurückliefert. Dabei sollen die beiden übergebenen Listen erhalten bleiben! D.h., sie müssen die Elemente für die Ergebnisliste neu (mittels `new`) erzeugen. (Hinweis: eine Möglichkeit zur Lösung der Aufgabe ist, den Mischprozess vom Mergesort-Algorithmus, der dort auf Arrays durchgeführt wird, auf Listen zu übertragen.)

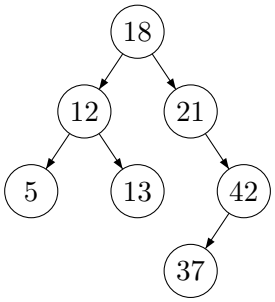
- Geben Sie eine C/C++-Datenstruktur `Liste` für eine einfach verkettete Liste von `int`-Werten an.
- Schreiben Sie eine Funktion `Liste* mische(Liste *eins, Liste *zwei)`, die als Parameter die Anker `eins` und `zwei` auf die erste bzw. zweite Liste erhält und als Ergebnis einen Anker auf eine Liste zurückgibt, die alle Elemente der ersten und zweiten Liste in sortierter Reihenfolge enthält. Beachten Sie dabei, dass die Listen jeweils leer sein können.
- Zusatzaufgabe (3 Punkte): Geben Sie die Laufzeit Ihres Programms in O-Notation in Abhängigkeit der Größen der beiden Listen an. Die Liste `eins` soll dabei n_1 Elemente und die Liste `zwei` n_2 Elemente enthalten.

Hinweis: es ist nur C/C++-Code für die Datenstruktur und die Mische-Funktion gefordert. Sie müssen sich nicht um die Ein- und Ausgabe der Listen kümmern und auch kein Hauptprogramm (`int main()`) schreiben.

Teilklausur II in Informatik – Lösungshinweise

Aufgabe 1 — Binäre Suchbäume

Einfügen erfolgt wie Suchen – auf dem ersten freien Platz wird das Element dann eingefügt. Der resultierende Baum hat Höhe 4, wählt man stattdessen die 37 als Wurzel des rechten Unterbaums, ergibt sich die Höhe 3. Die gesuchten Einfügereihenfolgen ergeben sich z.B. durch Preorder oder Levelorder – eine alternative Reihenfolge für den angegebenen Baum wären damit neben der Preorder 18,12,5,13,21,42,37 noch 18,12,21,5,13,42,37 – Reihenfolgen für den Suchbaum mit Höhe 3 sind 18,12,5,13,37,21,42 bzw. 18,12,37,5,13,21,42.



Aufgabe 2 — Sortieren

Beim ersten Beispiel wandert das größte Element nach rechts wie bei Bubblesort. Beim zweiten Beispiel wandert das kleinste Element nach links wie bei Insertionsort. Bei der Zusatzaufgabe handelt es sich um Heapsort – nach einem weiteren Tausch (das weitere Absinken der 14 während des Heapaufbaus) erhält man das Feld (63,21,52,14,18,42,37).

Bei Quicksort wird zunächst die 52 mit der 13 getauscht, danach die 63 mit der rechtesten 42 und noch die beiden verbleibenden 42 miteinander. Nach diesem letzten Tausch sind die beiden Index-Zeiger jeweils auf der 37, die äußere while-Schleife wird noch betreten, der linke Index-Zeiger wandert eins nach rechts. Es findet kein Tausch mehr statt. Die beiden rekursiven Quicksortaufrufe sind QS(0,6) (linker Rand bis zum rechten Index-Zeiger) und QS(7,10) (linker Index-Zeiger bis zum rechten Rand).

Aufgabe 3 — Dijkstra

Im Dijkstra-Algorithmus wird R, eine Teilmenge der Knoten, verwaltet – zu Beginn ist diese Menge mit dem Startknoten initialisiert. Die berechneten Entfernungen d(·) sind für diesen Startknoten 0, für alle anderen Knoten unbekannt (∞). Solange diese Menge R nicht leer ist, wird der Knoten u aus R mit minimalem Wert d(u) entfernt und überprüft, ob die vom Knoten u ausgehenden Kanten (u,v) die Werte d(v) verringern können (setze ggf. d(v)=d(u)+γ(u,v) und füge v zu R hinzu). Merkt man sich beim Setzen der d(v)-Werte den Vorgänger u (also pred(v)=u), so lassen sich mit diesen Werten auch die zugehörigen kürzesten Wege angeben.

Initialisiert wird mit d(A)=0 und R={}. A wird als kleinster Wert wieder entfernt. Die Nachfolger von A sind B und C, die Werte werden auf d(B)=2 und d(C)=3 gesetzt, R={B,C}.
B hat den kleinsten Wert, von dort ist mit einer Kante nur D erreicht (d(D)=8, R={C,D}.
C hat den kleinsten Wert, von dort sind mit einer Kante D und F erreichbar, der Wert d(D) verringert sich, F kommt neu zu R hinzu (d(D)=7, d(F)=4, R={D,F}.
F hat den kleinsten Wert, von dort sind mit einer Kante D und G erreichbar, der Wert d(D) verringert sich nochmals, G kommt neu zu R hinzu (d(D)=6, d(G)=10, R={D,G}.
D hat den kleinsten Wert, von dort sind mit einer Kante A, E und G erreichbar, der Wert d(G) verringert sich, A bleibt unverändert, E kommt neu zu R hinzu (d(G)=9, d(E)=7, R={E,G}.
E hat den kleinsten Wert, von dort sind mit einer Kante B und G erreichbar, der Wert d(G) verringert sich, B bleibt unverändert (d(G)=8, R={G}.
G hat den kleinsten Wert, von dort sind keine weiteren Knoten erreichbar.
R ist leer, der Algorithmus terminiert. Die kürzesten Entfernungen sind d=(0,2,3,6,7,4,8).

Aufgabe 4 — Analyse eines Sortieralgorithmus

Zu Beginn: `(5,8,2,1,4,9,6)`;
nach Durchlauf `i=1`: `(5,8,2,1,4,9,6)`; nach Durchlauf `i=2`: `(2,5,8,1,4,9,6)`;
nach Durchlauf `i=3`: `(1,2,5,8,4,9,6)`; nach Durchlauf `i=4`: `(1,2,4,5,8,9,6)`;
nach Durchlauf `i=5`: `(1,2,4,5,8,9,6)`; nach Durchlauf `i=6`: `(1,2,4,5,6,8,9)`.

Das Verfahren arbeitet wie Insertionsort. Im `i`-ten Durchlauf wird gezählt, wieviel Elemente kleiner als `tosort[i]` sind, und in der zweiten inneren Schleife, die Elemente von der Zielposition `z` bis `i-1` um eins nach rechts im Array verschoben. So ist nach dem `i`-ten Durchlauf das Array von Index `0` bis `i` sortiert.

Der Aufwand der beiden inneren Schleifen ist jeweils $O(n)$ – da diese nicht ineinander geschachtelt sind, ist der Aufwand in der Summe ebenfalls $O(n)$. Dieser Aufwand ist $n-1$ Mal (äußere Schleife) durchzuführen. Gesamtaufwand damit $O(n^2)$.

Aufgabe 5 — Listen-Programmierung

Die Idee ist wie beim Mischen im Mergesort-Algorithmus: Wir haben zwei Zeiger, einen auf das erste (= das kleinste) der ersten Liste, einen auf das erste der zweiten Liste. Wir vergleichen jeweils die beiden kleinsten Elemente, fügen das kleinere als nächstes an die neue Liste an und rücken den entsprechenden Zeiger ein Element weiter. Ist die eine Liste am Ende angelangt, muss der Rest der anderen Liste noch kopiert werden.

Erzeugt man immer ein Listenelement für die zusammengesetzte Liste im voraus, so vereinfachen sich etwas die Sonderfälle, sobald eine der Listen leer ist bzw. falls bereits zu Anfang eine der Listen leer waren.
Der Aufwand ist offensichtlich linear (in jedem Schritt wird ein Element der einen oder der anderen Liste verarbeitet), somit $O(n_1 + n_2)$.

```
struct Liste{
  int value;
  Liste* next;
};

Liste* mische(Liste *eins, Liste *zwei){
  Liste * p1=eins;
  Liste * p2=zwei;
  Liste * p3=0;
  Liste * anker=0;
  if (p1!=0 || p2!=0) {
    anker = new Liste; p3=anker; // wird gleich gefüllt ...
    while (p1!=0 && p2!=0) { // es kommen wenigstens noch zwei Werte
      if (p1->value<p2->value) {
        p3->value=p1->value; p1=p1->next;
      } else {
        p3->value=p2->value; p2=p2->next;
      }
      p3->next=new Liste; p3=p3->next; // mind. 1 Element kommt noch
    } // jetzt ist entweder p1 oder p2 am Ende (aber nicht beide)
    while (p1!=0) { // entweder Rest von Liste 1 noch kopieren
      p3->value=p1->value; p1=p1->next;
      if (p1!=0) { p3->next=new Liste; p3=p3->next; }
    }
    while (p2!=0) { // oder Rest von Liste 2 noch kopieren
      p3->value=p2->value; p2=p2->next;
      if (p2!=0) { p3->next=new Liste; p3=p3->next; }
    }
  }
  return anker;
};
```