Prof. Dr. Thomas Breuel
Dipl.-Inf. Patrick Michel (AG ST)
Dipl.-Ing. Ludwig Schmidt-Hackenberg

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Übungsblatt 11: Software-Entwicklung 1 (WS 2010/11)

Ausgabe: in der Woche vom 17.01. bis zum 21.01.11
Abgabe: in der Woche vom 24.01. bis zum 28.01.11
Abnahme: max. zwei Tage nach der Übung

Aufgabe 1 Zusicherungen und Verifikation (Präsenzaufgabe)

a) Geben Sie vor dem Feldzugriff eine Assertion an, welche garantiert, dass der Feldzugriff sicher ausgeführt werden kann. Bestimmen Sie dann eine möglichst präzise Assertion für den Anfang der Prozedur.

```
int helper(int index, int first, int last) {
  int i;
  int[] feld;
  feld = new int[last - first + 1];
  initialize(feld);
  i = index - first;
  return feld[i];
}
```

b) Zeigen Sie mit einem Annotationsbeweis, dass die folgende Prozedur ihre Spezifikation erfüllt:

```
/*@ requires N > 0
  @ modifies \nothing
  @ ensures \result == N*N*N + 4*N*N + 4*N
*/
double exec(double N) {
    double s, f1, f2, f3, result;

  assert


  assert


  assert


  assert


  assert

    f1 = 2*N + 4;

  assert

    f2 = N + 2;

  assert

    f3 = N * f1 * f2;
```

```
  assert

    s = f3 / 2;

  assert

    result = s;

  assert

    return result;
}
```

Aufgabe 2 Zusicherungen und Verifikation (Einreichaufgabe)

a) Betrachten Sie folgende Implementierung der binären Suche auf Arrays:

```
boolean binary(int[] sorted, int first, int last, int key) {
  if (sorted == null || 0 > first || last >= sorted.length) return false;
  if (first < last) {
    int mid = first + (last - first) / 2;
    int x = sorted[mid];
    if (key < x) { return binary(sorted, first, mid, key); }
    if (key > x) { return binary(sorted, mid + 1, last, key); }
    return true;
  }
  return false;
}
```

Fügen Sie in die Implementierung die nötigen Zusicherungen ein, um plausibel zu machen, dass auf `sorted` nur zugegriffen wird, wenn es nicht `null` ist und der Index innerhalb der gültigen Schranken ist.

Welche Zusicherungen können Sie direkt als korrekt annehmen und welche gelten aus anderen Gründen? Begründen Sie jede Zusicherung!

Arbeitet die Funktion nun korrekt, unter der Annahme Ihre Zusicherungen werden erfüllt?

b) Betrachten Sie folgende Implementierung der Funktion n^3. Führen Sie einen Annotationsbeweis und zeigen Sie damit, dass die Funktion ihre Spezifikation erfüllt. Verwenden Sie ausschließlich in Java ausführbare Zusicherungen und überprüfen Sie Ihren Beweis für mehrere Eingaben, indem Sie ihn mit Java ausführen.

Hinweis: Zusicherungen werden beim Ausführen Ihres Programmes überprüft, wenn Sie den Java-Interpreter mit der Option `-enableassertions` (oder kurz `-ea`), also "`java -ea <Programm>`" aufrufen.

```
/*@ requires N > 0
  @ modifies \nothing
  @ ensures \result == N*N*N
*/
int cube(int N) {
    int i, s, t, result;
    i = 1;
    s = 1;
    while (i < N) {
        t = i;
        i = i + 1;
        t = t * i;
        s = s + 3 * t + 1;
    }
    result = s;
    return result;
}
```

Aufgabe 3 Listen und Komplexität (Einreichaufgabe)

Hinweis: Diese Aufgabe bezieht sich auf die Listen von Aufgabe 2 aus Übungsblatt 9.

a) Betrachten Sie eine Sequenz von n Aufrufen der Prozedur `append` mit vorheriger Erzeugung (`create`) der leeren Liste. Leiten Sie die Aufwandsfunktion $A(n)$ für diese Sequenz von Aufrufen her und bestimmen Sie dann deren Komplexitätsklasse.

   Als Kostenmaß verwenden Sie einheitliche Kosten von 1 für Zuweisungen, Vergleiche, arithmetischen Operationen, Verbundkomponentenzugriffe und Feldzugriffe. Die Kosten für das Erzeugen eines neuen Verbundobjekts sollen $n+1$ sein, wobei n die Anzahl der Komponenten des Verbunds sind. Im speziellen Fall von Feldern sollen sie ebenfalls $n + 1$ betragen, wobei n hier die Länge des Feldes ist.

b) Wir betrachten nun eine mögliche Optimierung der implementierten Listen. Dazu erweitern wir den Zustand des Listenverbunds um eine Komponente für die Anzahl der tatsächlich belegten Arrayzellen. Jedesmal wenn das Array vergrößert werden muss, legen wir es größer an als nötig, wir verdoppeln es in der Länge.

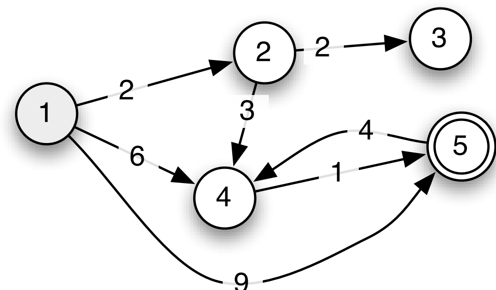
   Implementieren Sie diese Optimierung und passen Sie die restlichen Prozeduren entsprechend an. Was sind die Vor- und Nachteile dieser Implementierung und in welchen Fällen sind diese besonders relevant?

c) Führen Sie erneut eine Analyse der Zeitkomplexität durch, wie Sie es in a) für die erste Implementierung getan haben. Gehen Sie hier allerdings vereinfachend davon aus, dass es ein k mit $n = 2^k + 1$ gibt.

   Hinweis: Geben Sie bei Prozeduren mit Fallunterscheidungen verschiedene Aufwandsfunktionen für die einzelnen Fälle an und verwenden Sie an den Aufrufstellen dieser Prozeduren die entsprechend auf die Parameter passende Variante.

Aufgabe 4 Routenplanung (Einreichaufgabe)

Am Ende des Abschnitts 4.3 der Vorlesung wurde die Algorithmenentwicklung anhand eines Beispiels erläutert: Routenplanung in einer Stadt. Hier sehen Sie ein Beispiel für einen Stadtplan als Graph mit gewichteten Kanten. Der schnellste Weg vom Startknoten 1 zum Zielknoten 5 führt über die Knoten 2 und 4.



Laden Sie sich zu dieser Aufgabe die Datei “`Routenplanung.zip`” von der Webseite der Vorlesung herunter. Sie enthält u.a. eine Datei namens “`Routenplanung.java`”, in die Sie bitte Ihre Lösungen einfügen. Es ist jeweils angegeben, wohin welche Lösung gehört.

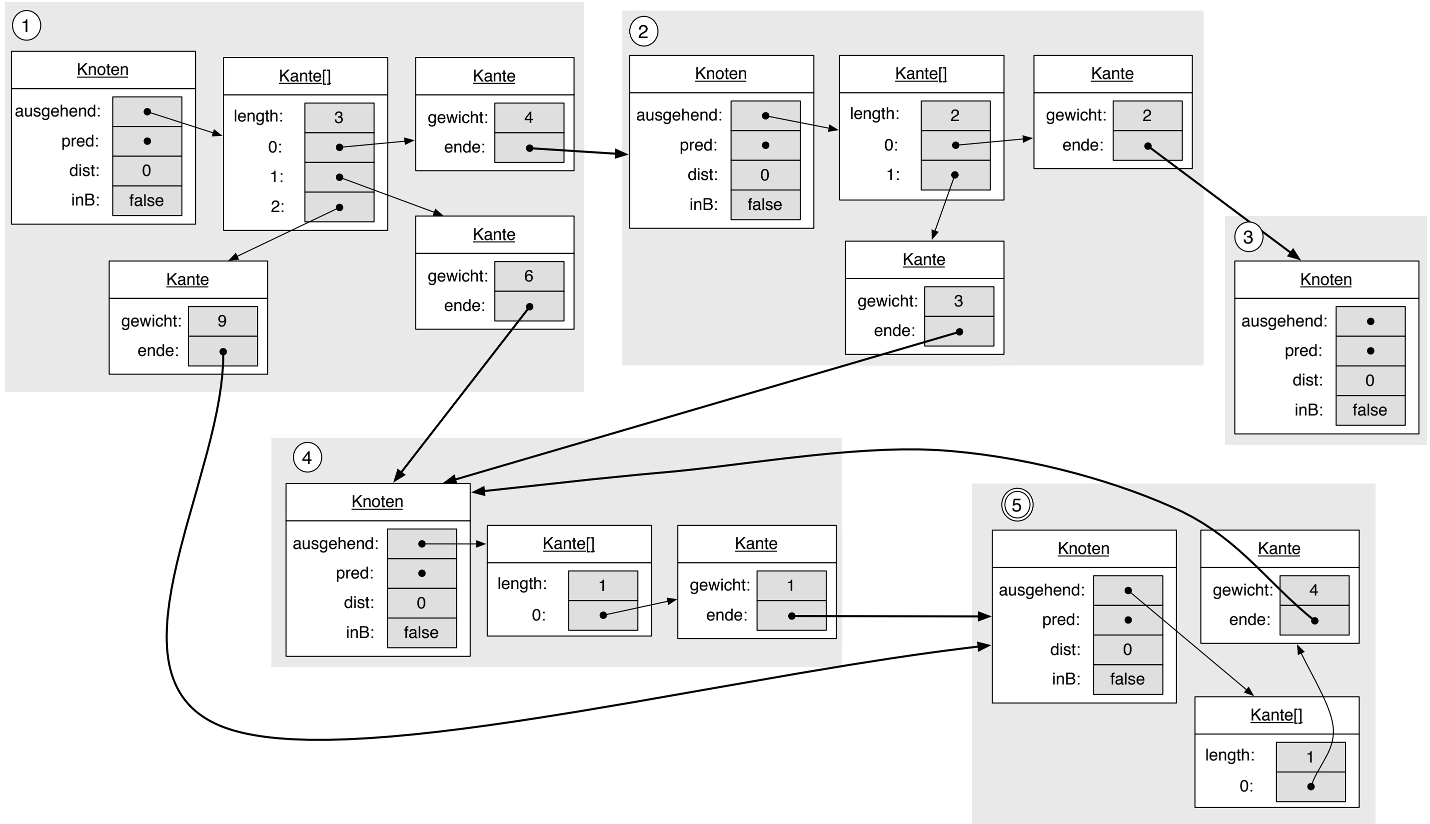
a) In der Grafik auf der nächsten Seite sehen Sie die Repräsentation des obigen Stadtplans als Geflecht. Erstellen Sie Verbundtypen gemäß dieser Graphik und Code um den Stadtplan als Geflecht anzulegen.

b) In der Vorlesung haben Sie gesehen, dass neben dem Graph zur Darstellung der Karte auch noch der Rand des bearbeiteten Areals dargestellt werden muss. Code für den Rand ist in der heruntergeladenen Datei enthalten. Er ist nur als Class-Datei verfügbar. Sie können folgende Klasse und Methoden verwenden.

```
class R.Rand { ... } // Verbund zur Darstellung eines Rand als Heap
R.Rand R.erzeugeRand() { ... }
boolean R.leer(R.Rand r) { ... }
Knoten R.naechsterKnoten(R.Rand r) { ... } // entferne den am naechsten gelegenen
                                            // Knoten und liefere ihn zurueck
void R.entfernen(R.Rand r, Knoten k) { ... } // entferne den uebergebenen
                                              // Knoten aus dem Rand
void R.einfuegen(R.Rand r, Knoten k) { ... } // fuege einen Knoten zum Rand hinzu
```

Implementieren Sie nun den in der Vorlesung entwickelten Algorithmus zum Finden der kürzesten Route. Testen Sie ihn am obigen Beispiel.

c) In dem heruntergeladenen Code ist auch ein Stadtplan enthalten. Testen Sie Ihre Routenplanung mit diesem Anhand der Start- und Zielknoten-Paare, die in der Methode `main` stehen. Entfernen Sie dazu die entsprechenden Kommentarzeichen. Eine Lösung ist jeweils darunter angegeben. Den gesamten Stadtplan finden Sie in der Datei Stadtplan.pdf.



Aufgabe 5 Speicherverwaltung in Java (Präsenzaufgabe)

Kreuzen Sie an, ob folgende Aussagen wahr oder falsch sind.

wahr	falsch	
☐	☐	Speicher für Verbunde und Felder wird auf dem Stack allokiert.
☐	☐	Lokale Variablen werden auf dem Heap angelegt.
☐	☐	Der Stackspeicher wird vom Programmierer nicht selbst verwaltet.
☐	☐	Der Programmierer muss den Heap selbst verwalten.
☐	☐	Ein nicht erreichbares Objekt kann von der Speicherbereinigung aus dem Speicher entfernt werden.
☐	☐	Ein nicht erreichbares Objekt kann später im Programm wieder erreichbar sein.
☐	☐	Mit `new` wird Speicher auf dem Heap allokiert.
☐	☐	Das Nichtfreigeben von Speicher ist in Sprachen ohne automatische Speicherbereinigung eine häufige Problemquelle.
☐	☐	Ein `new`-Ausdruck kann nicht immer erfolgreich ausgewertet werden.