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

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

Übungsblatt 10: Software-Entwicklung 1 (WS 2010/11)

Ausgabe: in der Woche vom 10.01. bis zum 14.01.11
Abgabe: in der Woche vom 17.01. bis zum 21.01.11
Abnahme: max. zwei Tage nach der Übung

Aufgabe 1 Verkettete Listen (Präsenzaufgabe)

Wir betrachten den Verbundtyp `IntList` aus der Vorlesung, mit dem verkettete Listen von ganzen Zahlen in Java dargestellt werden: `class IntList { int head; IntList tail; }`

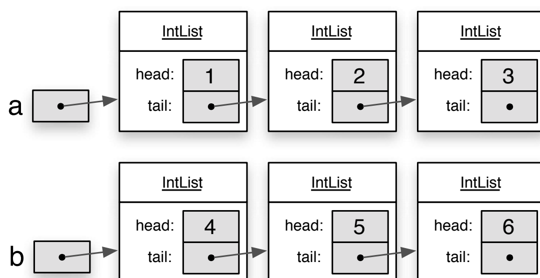
a) Schreiben Sie eine Java-Prozedur `int length(IntList l)`, welche die Länge der durch `l` referenzierten Liste berechnet und zurückgibt.

b) Schreiben Sie eine Java-Prozedur `boolean contains(int i, IntList l)`, welche überprüft, ob ein gegebenes Element `i` in der durch `l` referenzierten Liste enthalten ist.

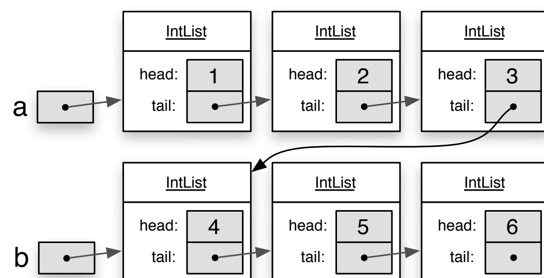
c) Schreiben Sie eine Java-Prozedur `void printList(IntList l)`, welche die durch `l` referenzierte Liste in Haskell-Notation in einer Zeile ausgibt. `Beispiel: [1, 2, 3, 4, 5]`

d) Schreiben Sie eine Funktion `IntList concat(IntList a, IntList b)`, die eine verkette Liste an eine andere hinten anhängt, indem sie `a` verändert (das letzte Element zeigt dann auf `b`). Wenn `a` leer ist (also `null`), dann gibt die Funktion einfach `b` zurück, ansonsten die Referenz auf `a`.

Beispiel:



nach Aufruf von `concat(a, b)`*:*



e) Wir wollen eine iterative Funktion `IntList reverse(IntList ls)` schreiben, welche eine Liste umkehrt, indem sie die `tail`-Referenzen der einzelnen Listenelemente verändert.

Das Vorgehen ist so, dass wir nach und nach die gegebene Liste `ls` abbauen und eine neue Liste `ms` (anfangs `null`) aufbauen. Dabei wird in jedem Durchlauf

1. ein Element der alten Liste `ls` vorne weg genommen und vorne an die neue Liste `ms` angehängt.
2. `ls` auf den Rest der alten Liste gesetzt und `ms` auf die neu entstandene Liste, sprich auf das neu angefügte Element.

Dies machen wir so lange, bis die alte Liste `ls` leer ist (`null`). Die umgekehrte Liste findet sich dann in `ms` und wird als Ergebnis zurück gegeben.

Visualisieren Sie auf dem Zusatzblatt das Geflecht an Verbunden bei einer Ausführung des oben beschriebenen Algorithmus. Stellen Sie jeden Zustand nach einer vollendeten Wiederholung dar.

f) Schreiben Sie eine Java-Funktion `IntList reverse(IntList l)`, welche die mit `l` referenzierte Liste umkehrt, indem sie Referenzen nach dem beschriebenen Verfahren umsetzt. Als Ergebnis wird die Referenz auf das nun erste Element zurück gegeben.

Aufgabe 2 Verkettete Listen (Einreichaufgabe)

Gehen Sie nun davon aus, dass als Parameter übergebene Listen *aufsteigend sortiert* sind.

a) Schreiben Sie eine Java-Funktion `IntList remove(int i, IntList ls)`, die ein Element mit dem Wert `i` aus der mit `ls` referenzierten Liste entfernt. Rückgabewert der Funktion ist die Referenz auf das erste Element der Ergebnisliste. Es soll nur ein Vorkommen von `i` entfernt werden und die Liste unverändert bleiben wenn `i` nicht enthalten ist.

b) Betrachten Sie folgenden Anfangszustand und folgendes Programm:

```
Anfangszustand:                     Programm:
  a zeigt auf   [1, 2, 4, 7, 8]       b = remove(7, a);
  b ist null:   []                    c = remove(1, a);
  c ist null:   []                    b = remove(4, a);
```

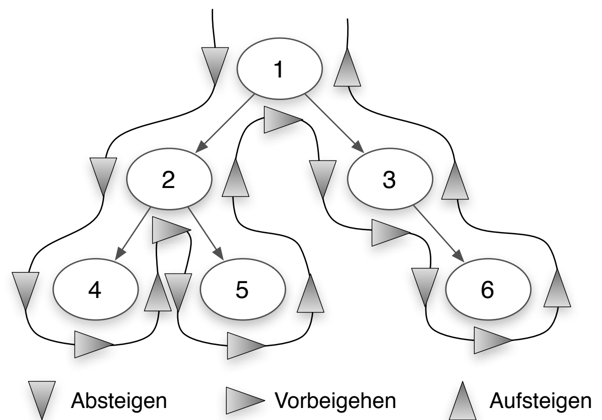
Stellen Sie vor und nach jeder Anweisung die von den Variablen `a`, `b` und `c` referenzierten Listen als Objektdiagramme – wie in Aufgabe 1d – dar.

Aufgabe 3 Binärbäume als Geflechte (Einreichaufgabe)

Wir betrachten den Verbundtyp `Node` aus der Vorlesung, mit dem mit ganzen Zahlen markierte Binärbäume in Java dargestellt werden:

```
class Node {
  int mark;
  Node left, right;
}
```

Die nebenstehende Abbildung zeigt einen Binärbaum, der von einer Kurve umhüllt wird. Jeder Knoten des Baums wird von dieser insgesamt dreimal besucht: Jeweils einmal beim Absteigen, Vorbeigehen und Aufsteigen.
Gibt man die Markierung jedes Knotens bei einem solchen Durchlauf beim *Vorbeigehen* aus, so spricht man von der *Inorder*-Reihenfolge.



a) Schreiben Sie eine Prozedur `printTree`, die alle Markierungen eines Baumes zeilenweise ausgibt, wobei der Durchlauf in *Preorder*-Reihenfolge erfolgen soll. Diese ergibt sich, wenn Markierungen beim Absteigen ausgegeben werden. Wiederholen Sie dies mit den *Inorder*- und *Postorder*-Reihenfolgen, die entsprechend durch Ausgabe beim Vorbeigehen bzw. Aufsteigen entstehen.

b) Schreiben Sie eine dritte Variante, die einen *Breitendurchlauf* verwendet. Beim Breitendurchlauf werden die Markierungen Niveau für Niveau von links nach rechts ausgegeben. In der Abbildung sind die Markierungen des Baums entsprechend ihrer Reihenfolge im Breitendurchlauf gewählt.

Hinweis: Der Breitendurchlauf ist deutlich schwieriger zu realisieren als die oben genannten Durchlaufreihenfolgen. Sie brauchen eine Hilfsdatenstruktur, da Sie nicht mit einfacher Rekursion auskommen.

c) Geben Sie eine einfachere Lösung für den Breitendurchlauf an, nachdem Sie den Verbundtyp für Binärbäume passend erweitert haben. Was ist der Nachteil einer solchen Lösung?

d) Implementieren Sie eine Prozedur `void replace(boolean[] position, Node sub, Node tree)`, die einen Unterbaum an einer durch `position` bestimmten Stelle im Baum `tree` durch `sub` ersetzt. Dabei repräsentiert `position` einen Pfad von der Wurzel in `tree` zu der Stelle, die ersetzt werden soll: Wenn ein `position` Eintrag `true` ist, wird der rechte Teilbaum, ansonsten der linke Teilbaum ausgewählt. Das Feld `{false, true}` würde in der Abbildung den Teilbaum mit der Markierung 5 bezeichnen.

e) Geben Sie eine sinnvolle Menge an zulässigen Parametern für die Prozedur `replace` an. Ist immer garantiert, dass das Ergebnis einer Ersetzung noch ein Baum ist?

Aufgabe 4 Verkettete Listen zur Speicherverwaltung (Einreichaufgabe)

In der Vorlesung haben Sie am Beispiel `MyArray` gesehen, wie man in Java einen Hauptspeicher und dessen Speicherverwaltung am Beispiel von Feldern simulieren kann. Im Prinzip handelt es sich dabei ebenfalls um eine Verwendung von verketteten Listen, wobei zu ihrer Realisierung keine Java Referenzen, sondern einfache Zahlen verwendet werden.

```
public class PC {
  static final int nil = -1;            // constant representing an invalid address
  static int[]  memory = new int[2048]; // main memory
  static int      root = nil;           // pointer to the first array

  static int   get(int a, int index)            { ... }  // get a value from an array
  static void put(int a, int index, int value) { ... }  // put a value into an array
  static int     alloc(int n)                   { ... }  // allocate a new array
  static boolean  free(int a)                   { ... }  // free an array
  static void printMemory()                     { ... }  // output the memory layout
}
```

Laden Sie sich zunächst die Datei "`PC.java`" von der Vorlesungsseite, die alle diese Definitionen und Signaturen enthält. Die Funktionen `get` und `put` sind ebenfalls schon im Sinne der Vorlesung implementiert. Schauen Sie sich auch die Diagramme in den Vorlesungsfolien gut an – Kapitel "Felder im Speicher", direkt vor Abschnitt 4.2.10.

Das Feld `memory` stellt hier den Hauptspeicher dar, der Zeiger `root` zeigt auf das erste Array im Speicher und zwar auf dessen logischen `next` Eintrag. Dahinter liegt also der logische `length` Eintrag und dann kommen die Daten des Arrays. Ist `root` der `nil` Wert, gibt es noch kein Array im Speicher. Ist der logische `next` Wert eines Arrays `nil`, so ist es das letzte im Speicher.

`alloc` soll `nil` zurück geben, wenn nicht mehr genug Speicher vorhanden ist. `free` soll zurück liefern, ob wirklich ein Array gelöscht wurde oder ob die Adresse gar kein Array bezeichnet hat.

Hinweis: Achten Sie darauf, dass die logischen `next` Werte eines Array Blocks auf den Anfang des nächsten Array Blocks zeigen, der Benutzer bekam von `alloc` und gibt `free` die Adresse des ersten Datums des Arrays!

a) Schreiben Sie die Funktion `printMemory`, welche die aktuelle Einteilung des Hauptspeichers ausgibt. Wie Sie diese Informationen genau formatieren bleibt Ihnen überlassen. Dabei steht in jeder Zeile die Adresse des nächsten Blocks, dessen Typ, seine Größe und bei Arrays noch abgeleitete Zusatzinformationen.

```
memory layout
  0000 (array)  130 words (at: 0002, length:  128, next: 0260)
  0130 (empty)  130 words
  0260 (array)   66 words (at: 0262, length:   64, next: 0326)
  0326 (array)  194 words (at: 0328, length:  192, next: 0714)
  0520 (empty)  194 words
  0714 (array)   34 words (at: 0716, length:   32, next:   -1)
  0748 (empty) 1300 words
```

Testen Sie die Funktion mit handgeschriebenen Beispielen für alle Randfälle.

b) Implementieren Sie die Funktionen `alloc` und `free`, wobei Sie für `alloc` die *best-fit* Strategie verwenden sollen, d.h. Sie wählen stets den kleinsten `empty` Block aus, in den der neue `array` Block noch paßt.

Testen Sie die Funktionen und geben Sie Zwischenzustände mit `printMemory` aus. Sie dürfen in den Funktionen `alloc` und `free` davon ausgehen, dass `memory` und `root` konsistent sind.

c) *(freiwillige Zusatzaufgabe)* Reichern Sie Ihre Implementierung von `printMemory` so mit `assertion` Annotationen an, dass Sie während der Ausgabe auch gleichzeitig eine komplette Überprüfung auf Konsistenz vornehmen. Schalten Sie die Überprüfung dieser Zusicherungen mit dem Flag `-ea` ein. Es sollte nun vor allem nicht mehr vorkommen, dass Ihre Funktion mit einer anderen Ausnahme als `AssertionError` abrupt terminiert! Testen Sie dies mit fehlerhaften Hauptspeichern.

Aufgabe 5 Aufwand und Komplexität (Einreichaufgabe)

a) Betrachten Sie folgende Implementierung der Matrixmultiplikation von Blatt 9:

```
Matrix mmult(Matrix a, Matrix b) {
  Matrix res = create(rows(a), cols(b));
  for (int i = 0; i < rows(a); i++) {
    for (int j = 0; j < cols(b); j++) {
      int sum = 0;
      for (int k = 0; k < cols(a); k++) {
        sum = sum + get(a, i, k) * get(b, k, j);
      }
      set(res, i, j, sum);
    }
  }
  return res;
}
```

Bestimmen Sie die Aufwandsfunktion $A(x, y, z)$ für diese Implementierung in Abhängigkeit der Größe der Eingabe, also der drei Werte:[1] $x =$ `rows(a)` $y =$ `cols(a)` $z =$ `cols(b)`

Als Kostenmaß nehmen wir für die Matrixoperationen (`create`, `get`, `set`, `rows`, `cols`) einheitlich Kosten von 1 an und vernachlässigen alle anderen Operationen.

In welcher Komplexitätsklasse liegt die Implementierung für quadratische Matrizen?

b) Betrachten Sie folgende Implementierung der Fibonacci-Funktion von Blatt 9:

```
int fib(int n) {
  Matrix m = create(2,2);
  set(m, 0, 0, 1); set(m, 0, 1, 1);
  set(m, 1, 0, 1); set(m, 1, 1, 0);

  Matrix v = create(2,1);
  set(v, 0, 0, 1);
  set(v, 1, 0, 0);

  Matrix res = create(2,2);
  set(res, 0, 0, 1); set(res, 0, 1, 0);
  set(res, 1, 0, 0); set(res, 1, 1, 1);

  for (int i = 0; i < n; i++) { res = mult(res, m); }
  res = mult(res, v);

  return get(res, 0, 1);
}
```

Geben Sie für diese Implementierung die Aufwandsfunktion $B(n)$ in Abhängigkeit vom Wert des Parameters n an, wobei Sie dasselbe Kostenmaß wie in a) verwenden. Geben Sie an, in welcher Komplexitätsklasse die Implementierung liegt.

c) Testen Sie die Ergebnisse Ihrer Aufwandsfunktionen für konkreten Eingaben, indem Sie in der Implementierung die Aufrufe der Matrixfunktionen in einer globalen Variablen mitzählen.

[1] Wir gehen davon aus, dass `cols(a) == rows(b)` gilt.