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

TU Kaiserslautern
Fachbereich Informatik
AG Bildverstehen und Mustererkennung

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

Die Hinweise und Vorschläge in diesem Dokument sollen der Lösungsfindung dienen und erheben demnach weder Anspruch auf Vollständigkeit noch Korrektheit. Sollten Sie Fehler finden, würden wir uns freuen, wenn Sie uns diese mitteilen. (Kontaktdaten finden Sie auf unserer Webpräsenz.)

Die auf diesem Blatt zur Lösung verwendeten Sprachmittel beschränken sich auf prozedurales Java mit Verbunden und Feldern, sowie Geflechten.

Aufgabe 1 Verkettete Listen (Präsenzaufgabe)

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

```
static int length(IntList l) {
  if (l == null)
    return 0;
  else
    return 1 + length(l.tail);
}
// oder

static int length(IntList l) {
  int res = 0;
  while (l != null) {
    res++;
    l = l.tail;
  }
  return res;
}
```

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.

```
static boolean contains(int i, IntList l) {
  if (l == null)
    return false;
  else
    return l.head == i || contains(i, l.tail);
}
// oder

static boolean contains(int i, IntList l) {
  while (l != null) {
    if (l.head == i)
      return true;
    l = l.tail;
  }
  return false;
}
```

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]`

```
static void printList(IntList l) {
  if (l == null) {
    IO.println("[]");
    return;
  }

  IO.print("[");
  IO.print(l.head);
  l = l.tail;

  while(l != null) {
    IO.print(", ");
    IO.print(l.head);
    l = l.tail;
  }

  IO.println("]");
}
```

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`.

```
static IntList concat(IntList a, IntList b) {
  if (a == null) return b;

  IntList current = a;
  while (current.tail != null)
    current = current.tail;

  current.tail = b;
  return a;
}
```

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

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.

Hinweise: Siehe Zusatzblatt am Ende des Dokuments.

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.

```
static IntList reverse(IntList l) {
  IntList m = null;

  while (l != null) {
    IntList temp = l.tail;
    l.tail = m;
    m = l;
    l = temp;
  }

  return m;
}
```

Aufgabe 2 Verkettete Listen (Einreichaufgabe)

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

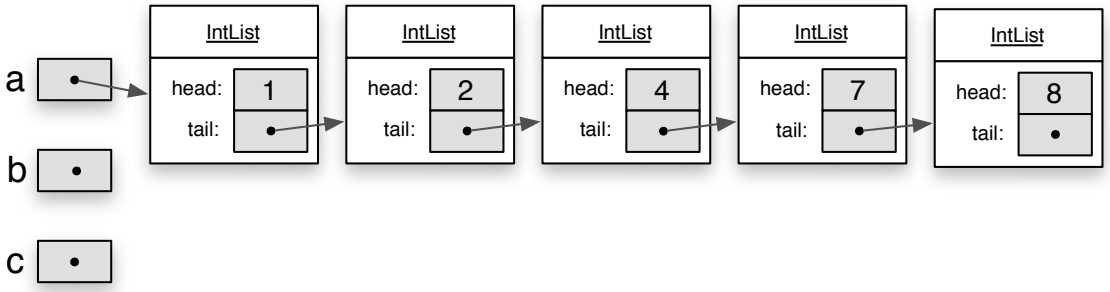
a) Schreiben Sie eine Java-Funktion `IntList remove(int i, IntList ls)`, die ein Element mit dem Wert `i` aus der mit `ls` referenzierten Liste entfernt. Rückgabewert der Funktion ist die Referenz auf das erste Element der Ergebnisliste. Es soll nur ein Vorkommen von `i` entfernt werden und die Liste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

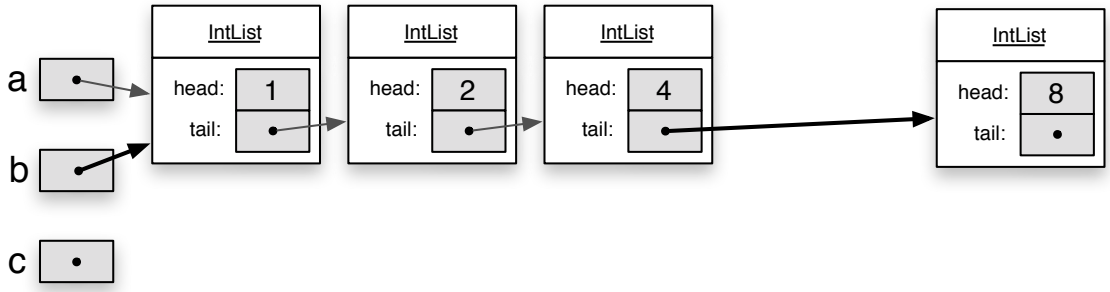
```
static IntList remove(int i, IntList l) {
  if (l == null) {
    return null;
  } else if (l.head == i) {
    return l.tail;
  } else {
    if (l.head < i) l.tail = remove(i, l.tail);
    return l;
  }
}

// oder

static IntList remove(int i, IntList l) {
  if (l == null) {
    return null;
  } else if (l.head == i) {
    return l.tail;
  }

  for (IntList cur = l; cur.tail != null; cur = cur.tail) {
    if (cur.tail.head == i) {
      cur.tail = cur.tail.tail;
      return l;
    } else if (cur.tail.head > i) {
      return l;
    }
  }

  return l;
}
```

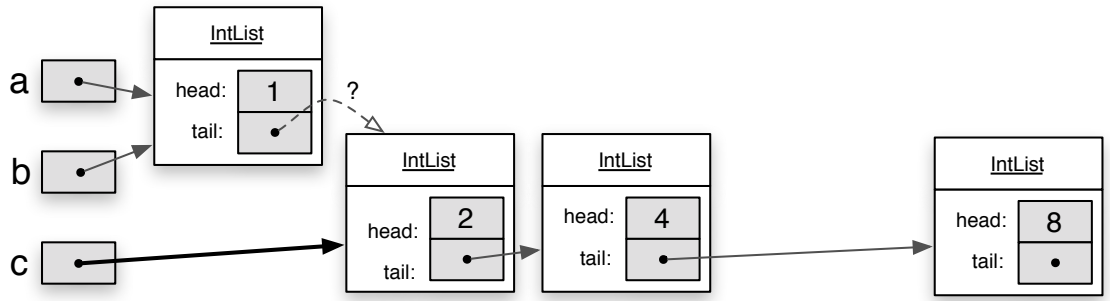
b) Betrachten Sie folgenden Anfangszustand und folgendes Programm:

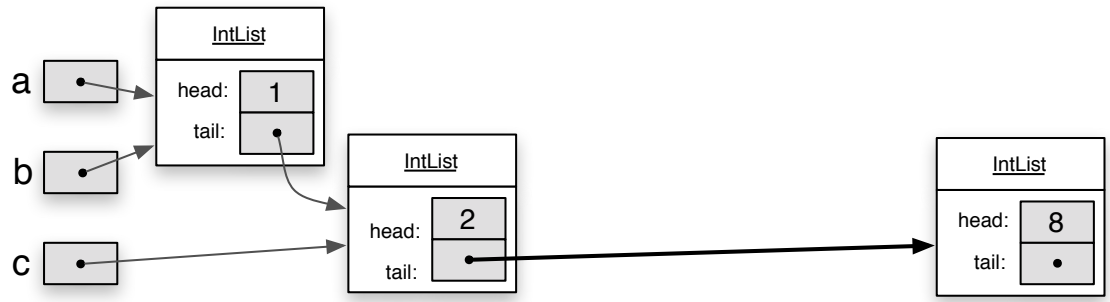
```
Anfangszustand:                    Programm:
  a zeigt auf   [1, 2, 4, 7, 8]      b = remove(7, a);
  b ist null:   []                   c = remove(1, a);
  c ist null:   []                   b = remove(4, a);
```

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









Die Existenz der gestrichelten und mit einem Fragezeichen versehenen Referenz im dritten Diagramm hängt von der Implementierung von `remove` ab. Hier hat man im Fall, dass das erste Element entfernt werden soll, die Wahl dessen Referenz auf das zweite Element intakt zu lassen oder zu löschen.

Das letzte Diagramm geht also davon aus, dass diese Referenz bestehen blieb. Alternativ wäre das zu entfernende Element mit dem Wert `4` von `a` aus nicht erreichbar gewesen und es wäre nichts passiert.

Aufgabe 3 Binärbäume als Geflechte (Einreichaufgabe)

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.

```
// Preorder
static void printTree(Node b) {
  if (b != null) {
    IO.println(b.mark);
    printTree(b.left);
    printTree(b.right);
  }
}

// Postorder
static void printTree(Node b) {
  if (b != null) {
    printTree(b.left);
    printTree(b.right);
    IO.println(b.mark);
  }
}

// Inorder entsprechend den IO Befehl zwischen den beiden Conditionals...
```

b) Schreiben Sie eine vierte 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.

```
// Breitendurchlauf
static void printTree(Node b) {
  if (b != null) {                    // todo is the current level
    Node[] todo = new Node[1];        // at the root its just one node
    todo[0] = b;
    while (todo.length > 0) {         // current level is not empty
      for (int i = 0; i < todo.length; i++)
        IO.println(todo[i].mark);     // print the complete level
      todo = successors(todo);        // calculate next level from this one
    }
  }
}

static Node[] successors (Node[] bl) {
  int count = 0;
  for (int i = 0; i < bl.length; i++) {   // calculate number of nodes
    count += bl[i].left == null ? 0 : 1;  // on this level, i.e. 0 or 1 per
    count += bl[i].right == null ? 0 : 1; // possible child
  }
  Node[] result = new Node[count];        // create the level array and
  for (int i = 0, j = 0; i < bl.length; i++) { // add all children
    if (bl[i].left != null) result[j++] = bl[i].left;
    if (bl[i].right != null) result[j++] = bl[i].right;
  }
  return result;
}
```

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?

Führe im Verbund einen neuen Eintrag `next` ein, der auf das nächste Element im Baum gemäß einem Breitendurchlauf zeigt. Damit wird der Aufbau und die Verwaltung des Baums komplizierter, aber der Breitendurchlauf läuft sehr schnell.

```
static void printTree(Node b) {
  if (b != null) {
    IO.println(b.mark);
    printTree(b.next);
  }
}
```

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.

```
static void replace(boolean[] position, Node sub, Node tree) {
  if (position == null || position.length == 0 || tree == null) return;
  for (int i = 0; i < position.length - 1; i++) {
    tree = position[i] ? tree.right : tree.left;
  }
  if (position[position.length - 1]) {
    tree.right = sub;
  } else {
    tree.left = sub;
  }
}
```

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?

In der vorgestellten Implementierung betrachten wir für den `position` Parameter nur solche Werte als zulässig, wenn im Binärbaum `tree` ein Pfad von der Wurzel zu dieser Position existiert.

Als zusätzliche Einschränkung betrachten wir nur solche Parameter, die dazu führen, dass `tree` nach Modifikation von `replace` wieder ein Baum ist, d.h. keine Zyklen eingebaut werden. Dies kann zum Beispiel durch folgenden Aufruf passieren: `replace({true}, b, b)`. Eine konservative Einschränkung wäre, erstmal zu verbieten, dass `sub` einen Eintrag von `tree` referenziert.

Aufgabe 4 Verkettete Listen zur Speicherverwaltung (Einreichaufgabe)

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.

```
// output the main memory - checks memory for consistency
static void printMemory() {
  assert root == nil || root >= 0;
  assert root == nil || root < memory.length - 1;

  IO.println("memory layout");
  if (root == nil) {
    printEmpty(0, memory.length);
    return;
  }

  printEmpty(0, root);
  printArray(root);
}
```

```
static void printIndent(int n, String text) {
  IO.print("  ");
  IO.print(pad(n, '0'));
  IO.print(" ");
  IO.println(text);
}

static String pad(int n, char c) {
  String s = "" + n;
  for (int i = s.length(); i < (int) Math.ceil(Math.log10(memory.length)); i++)
    s = (n < 0 ? ' ' : c) + s;
  return s;
}

static void printArray(int a) {
  assert a >= 0;
  assert a + 1 < memory.length;

  int next = memory[a];
  int len = memory[a + 1];

  assert next == nil || next < memory.length - 1;
  assert next == nil || a + 1 + len < next;
  assert a + 1 + len < memory.length;

  printIndent(a, "(array) " + pad(len + 2, ' ') +
    " words (at: " + pad(a + 2, '0') + ", length: " + pad(len, ' ') +
    ", next: " + pad(next, '0') + ")");

  if (next == nil) {
    printEmpty(a + 2 + len, memory.length - a - 2 - len);
    return;
  }

  printEmpty(a + 2 + len, next - a - 2 - len);
  printArray(next);
}

static void printEmpty(int a, int n) {
  assert n >= 0;
  if (n == 0) return;
  printIndent(a, "(empty) " + pad(n, ' ') + " word" + (n == 1 ? "" : "s"));
}
```

Die Lösung teilt sich auf mehrere Hilfsfunktionen auf, so dass man sich jeweils nur mit einer Problematik beschäftigen muss. Damit erledigt `printMemory` selbst die Sonderfälle bezüglich `root`, `printArray` und `printEmpty` erledigen die beiden Varianten. Nur die `printArray` Funktion ist dann rekursiv und ruft *immer* `printEmpty` mit dem entsprechenden Restspeicher oder Speicher zum nächsten Block auf. Leere `empty` Blöcke werden dann von `printEmpty` einfach nicht ausgegeben. Die Funktionen `pad` und `printIndent` dienen nur dem Layout und können entsprechend vernachlässigt werden.

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.

```
static int alloc(int n) {
  if (n < 0) return nil;

  if (root == nil) {
    root = 0;
    memory[0] = nil;
    memory[1] = n;
    return 2;
  }

  int cand = nil;
  int cand_size = 0;

  for (int cur = root; cur != nil; cur = memory[cur]) {
    int next = memory[cur];
    int len = memory[cur + 1];
    int size = (next != nil ? next : memory.length) - cur - 2 - len;

    if (size - 2 >= n && (cand == nil || size - 2 < cand_size)) {
      cand = cur;
      cand_size = size - 2;
    }
  }

  if (cand == nil) return nil;

  int a = cand + 2 + memory[cand + 1];
  memory[a] = memory[cand];
  memory[a + 1] = n;
  memory[cand] = a;

  return a + 2;
}

static boolean free(int a) {
  if (a < 2) return false;

  int rec = a - 2;

  if (rec == root) {
    root = memory[rec];
    return true;
  }

  int cur = root;
  while (cur != nil) {
    if (memory[cur] == rec) {
      memory[cur] = memory[rec];
      return true;
    } else {
      if (rec <= cur) return false;
      cur = memory[cur];
    }
  }

  return false;
}
```

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.

Dies ist in der Lösung von a) schon geschehen.

Aufgabe 5 Aufwand und Komplexität (Einreichaufgabe)

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

```
Matrix mult(Matrix a, Matrix b) {
  Matrix res = create(rows(a), cols(b));          // 3
  for (int i = 0; i < rows(a); i++) {             // 1
    for (int j = 0; j < cols(b); j++) {           // 1
      int sum = 0;
      for (int k = 0; k < cols(a); k++) {         // 1
        sum = sum + get(a, i, k) * get(b, k, j); // 2
      }
      set(res, i, j, sum);                        // 1
    }
  }
  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 den Kommentaren im Programm wurde jeweils für die Programmzeile gezählt, wieviele Aufrufe der entsprechenden Prozeduren enthalten sind.

In welcher Komplexitätsklasse liegt die Implementierung für quadratische Matrizen?

Schleifen werden zu Summen über dem Aufwand ihres Rumpfes. Oft führt dies – wie in diesem Fall – zu verschachtelten Summen. Innerhalb eines Rumpfes sind die Operationen des Kostenmaß dann zu zählen, auf die Kosten von evtl. enthaltenen Schleifen zu addieren, und dann aufzusummieren. Durch diese Umwandlung kommen wir auf folgende Formel:

$$A(x,y,z) = 3 + \sum_{i=0}^{x-1}\left(1 + \sum_{j=0}^{z-1}\left(1 + \left(\sum_{k=0}^{y-1} 3\right) + 2\right) + 1\right) + 1$$

Die innerste 3 ergibt sich einfach aus Summe von Schleifenbedingung und Rumpf, also $1+2$ (siehe Kommentare).

Auf der Ebene der nächsten Schleife kommt wieder eine Schleifenbedinung zu der Summe, in diesem Fall 1, plus die Anweisung am Ende der Schleife (auch 1). Nicht vergessen darf man allerdings, dass die innere Schleife auch einmal am Ende die Bedingung mit negativem Ausgang testet! Damit kommt noch eine 1 dazu, weshalb wir $1+S+2$ für den Rumpf erhalten (wobei S die Summe der innersten Schleife ist). So setzt sich die Argumentation bis auf oberste Ebene fort. Nun vereinfachen wir die Formel:

$$\begin{aligned}
&= 3 + \sum_{i=0}^{x-1}\left(1 + \sum_{j=0}^{z-1}\left(1 + y\cdot 3 + 2\right) + 1\right) + 1 \\
&= 3 + \sum_{i=0}^{x-1}\left(1 + z\cdot\left(1 + y\cdot 3 + 2\right) + 1\right) + 1 \\
&= 3 + x\cdot\left(1 + z\cdot\left(1 + y\cdot 3 + 2\right) + 1\right) + 1 \\
&= x\cdot\left(z\cdot\left(3y + 3\right) + 2\right) + 4 \\
&= x\cdot(3yz + 3z + 2) + 4 \\
&= 3xyz + 3xz + 2x + 4
\end{aligned}$$

Für quadratische Matrizen, sprich $n = x = y = z$, ergibt sich:

$$A(n) = 3n^3 + 3n^2 + 2n + 4 \in O(n^3)$$

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

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.

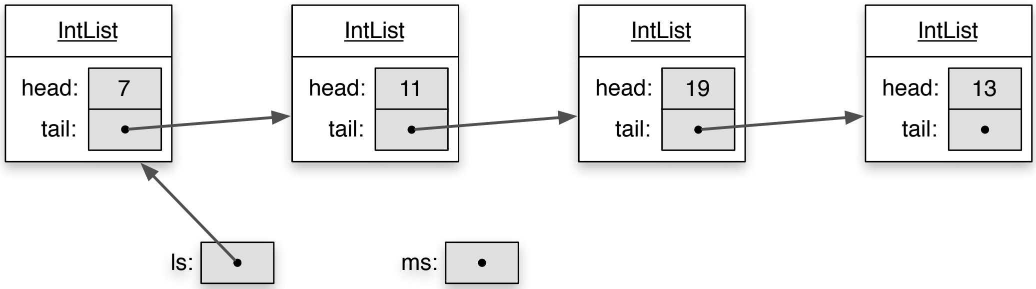
Hier ist eine Menge konstant und die einzige Schleife benutzt `mult`, für die wir den Aufwand schon kennen. Damit kommen wir auf:

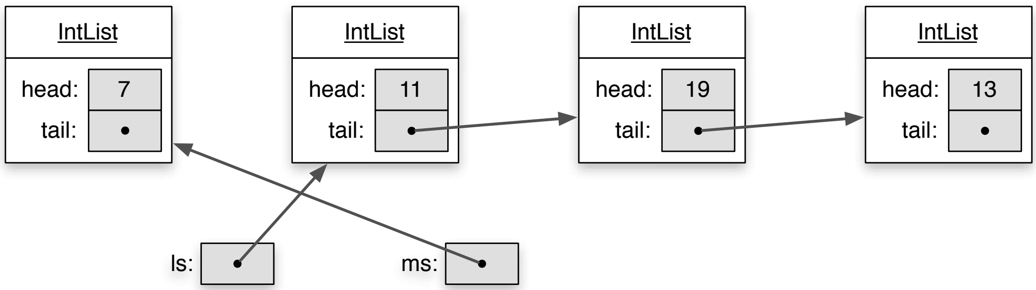
$$
\begin{aligned}
B(n) &= 5 + 3 + 5 + \sum_{i=0}^{n-1} \Big(A(2,2,2) \Big) + A(2,2,1) + 1 \\
&= 14 + \sum_{i=0}^{n-1} \Big(3 \cdot 2 \cdot 2 \cdot 2 + 3 \cdot 2 \cdot 2 + 2 \cdot 2 + 4 \Big) + 3 \cdot 2 \cdot 2 \cdot 1 + 3 \cdot 2 \cdot 1 + 2 \cdot 2 + 4 \\
&= 14 + \sum_{i=0}^{n-1} \Big(24 + 12 + 4 + 4 \Big) + 12 + 6 + 4 + 4 \\
&= 14 + \sum_{i=0}^{n-1} 44 + 26 \\
&= 44\,n + 40 \\
&\in O(n)
\end{aligned}
$$

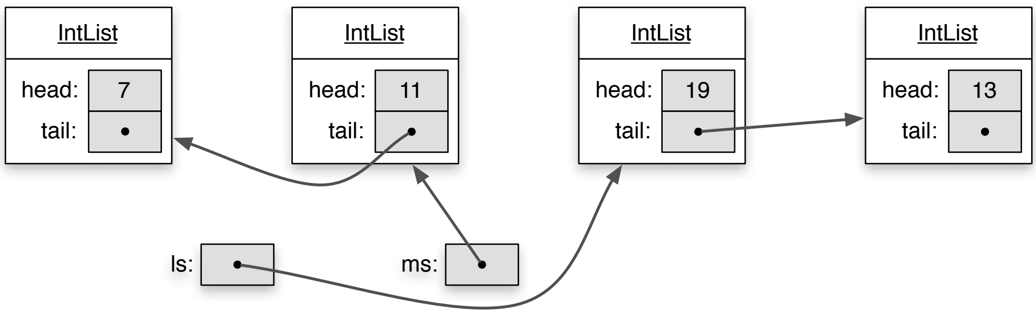
c) Testen Sie die Ergebnisse Ihrer Aufwandsfunktionen für konkreten Eingaben, indem Sie in der Implementierung die Aufrufe der Matrixfunktionen in einer globalen Variablen mitzählen.

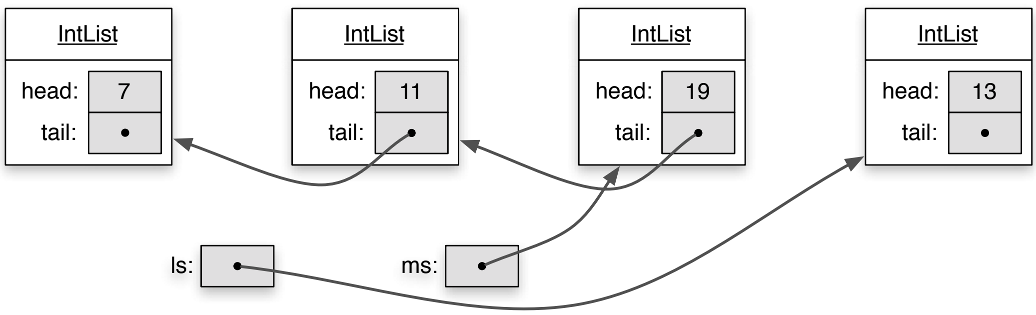
Es genügt eine Variable einzufügen, die am Anfang jeder Prozedur um eins erhöht wird, also in der Art:

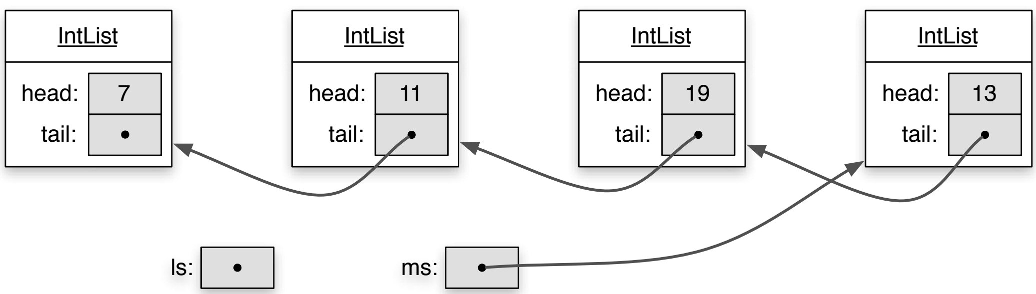
```
static int aufwand = 0;

static class Matrix {
  int rows, cols;
  int[] data;
}

static Matrix create(int rows, int cols) {
  aufwand++;
  ...
}

static void set(Matrix m, int i, int j, int value) {
  aufwand++;
  ...
}

...
```


IntList
head: 7
tail:
IntList
head: 11
tail:
IntList
head: 19
tail:
IntList
head: 13
tail:
ls:
ms:


IntList
head: 7
tail:
IntList
head: 11
tail:
IntList
head: 19
tail:
IntList
head: 13
tail:
ls:
ms:


IntList
head: 7
tail:
IntList
head: 11
tail:
IntList
head: 19
tail:
IntList
head: 13
tail:
ls:
ms:


IntList
head: 7
tail:
IntList
head: 11
tail:
IntList
head: 19
tail:
IntList
head: 13
tail:
ls:
ms:


IntList
head: 7
tail:
IntList
head: 11
tail:
IntList
head: 19
tail:
IntList
head: 13
tail:
ls:
ms: