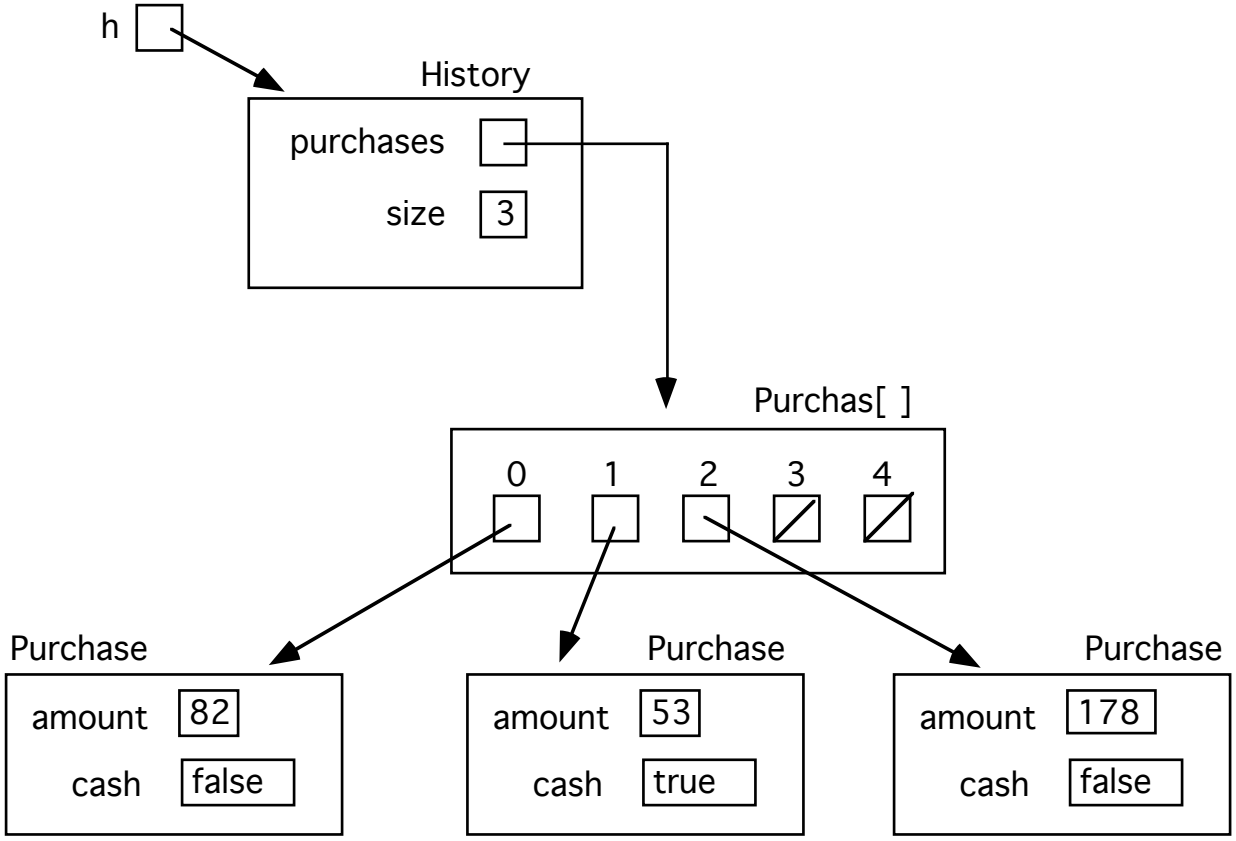
Wellesley College ◊ CS111 Computer Programming and Problem Solving ◊ Fall 2007

FINAL EXAM REVIEW SOLUTIONS

This draft contains solutions to all problems except Problems 8, 11, & 12

Problem 1: Sales Statistics *(Data Abstraction, Arrays, Lists, Objects, Object Diagrams)*

Part a.



Part b. Finishing the instance methods of the History class:

```
public class History {

  // Instance Variables:
  private Purchase [ ] purchases;
  private int size;

  // Constructor Method:
  public History (int maxEntries) {
    purchases = new Purchase[maxEntries];
    size = 0;
  }

  // Instance Methods:
  public void add (int amount, boolean cash) {
    if (size < purchases.length) {
      purchases[size] = new Purchase(amount, cash);
      size = size + 1;
    }
```

```
  }

  public int size(){
    return size;
  }


  // min() and max() are instances of the general array accumulation idiom
  public int min(boolean cash){
    int minVal = Integer.MAX_VALUE; // Positive infinity is identity of min
    for (int i = 0; i < size; i ++){
      if (purchases[i].getCash() == cash){
        minVal = Math.min(minVal, purchases[i].getAmount());
      }
    }
    return minVal;
  }

  public int max(boolean cash){
    int maxVal = Integer.MIN_VALUE; // Negative infinity is identity of max
    for (int i = 0; i < size; i ++){
      if (purchases[i].getCash() == cash){
        maxVal = Math.max(maxVal, purchases[i].getAmount());
      }
    }
    return maxVal;
  }

  public int average(boolean cash){
    int sum = 0;
    int count = 0;
    for (int i = 0; i < size; i++){
      if (purchases[i].getCash() == cash){
        sum = sum + purchases[i].getAmount();
        count = count + 1;
      }
    }
    if (count == 0){
      return 0;
    } else {
      return sum/count;
    }
  }

  public int number (boolean cash){
    int count = 0;
    for (int i = 0; i < size; i ++) {
      if (purchases[i].getCash() == cash){
        count = count + 1;
      }
    }
    return count;
  }

  public int percentage (boolean cash){
    if (size > 0){
      return (number(cash) * 100)/size;
    } else {
      return 0;
    }
  }
}
```

Part c. You cannot write the constructor method as either of:

```
public Purchase (int amount, boolean cash) {
    amount = amount;
    cash = cash;
  }

public Purchase (int a, boolean c) {
    int amount = a;
    int cash = c;
  }
```

In both cases, the variables `amount` and `cash` refer to *local* variables in the execution frame, not the instance variables in the instance. If there is a local variable with the same name as the instance variable, the instance variable must be referred to as **this**.amount.

Part d.

1) Implement the `Purchase` class with the instance variable as an array of two integers:

```
public class Purchase {

  // Instance Variable
  private int [] sale;

  // Constructor Method
  public Purchase (int i, boolean b) {
    sale = new int [2];
    sale[0] = i;
    sale[1] = intToBool(b);
  }

  // Instance Methods
  public int getAmount () {return sale[0];}

  public void setAmount (int newAmount) {sale[0] = newAmount;}

  public boolean getCash () {return (sale[1] == 1);}

  public void setCash (boolean newCash) {
    sale[1] = intToBool(newCash);
  }

  // Useful helper method
  private bool boolToInt (boolean b) {
    if (b) {
      return 1;
    } else {
      return 0;
    }
  }
```

2) Implement `Purchase` class with an integer list:

```
public class Purchase {

  // Instance Variable
  private IntList sale;

  // Constructor Method
  public Purchase (int i, boolean b) {
    sale = prepend(i, prepend(intToBool(b), empty()));
  }

  // Instance Methods
  public int getAmount () {
    return head(sale);
  }

  public void setAmount (int newAmount) {
    sale = prepend(newAmount, tail(sale));
  }

  public boolean getCash () {
    return (head(tail(sale)) == 1);
  }

  public void setCash (boolean newCash) {
    sale = prepend(head(sale), prepend(intToBool(newCash), empty()));
  }

  // Useful helper method
  private bool boolToInt (boolean b) {
    if (b){
      return 1;
    } else {
      return 0;
    }
  }
}
```

3) Implement `Purchase` class as a single positive or negative integer:

```
public class Purchase {

  // Instance Variable
  private int sale;

  // Constructor Method
  public Purchase (int i, boolean b) {
    sale = (sign(b)) * i
  }

  // Instance Methods
  public int getAmount () {
    return Math.abs(sale);
  }

  public void setAmount (int newAmount) {
    sale = sign(sale) * Math.abs(newAmount);
  }

  public boolean getCash () {
    return (sale > 0);
  }

  public void setCash (boolean newCash) {
    sale = sign(newCash) * Math.Abs(sale);
  }

  // Useful helper method
  public int sign (boolean b) {
    if (b) {
      return 1;
    } else {
      return –1;
    }
  }
}
```

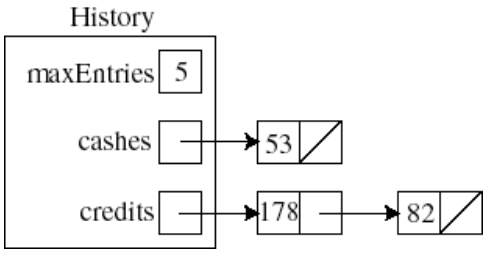
4) Implement `Purchase` class as a single integer where (i) amount is one half of integer (via integer division) and (ii) even is cash, odd is credit.

```
public class Purchase {

  // Instance Variable
  private int sale;

  // Constructor Method
  public Purchase (int i, boolean b) {
    sale = 2*i + intToBool(b);
  }

  // Instance Methods
  public int getAmount () {
    return sale/2; // integer division
  }

  public void setAmount (int newAmount) {
    sale = (newAmount*2) + (sale%2);
  }

  public boolean getCash () {
    return (sale%2) == 1;
  }

  public void setCash (boolean newCash) {
    sale = 2*(sale/2) + intToBool(newCash);
  }

  // Useful helper method
  private bool boolToInt (boolean b) {
    if (b){
      return 1;
    } else {
      return 0;
    }
  }
}
```

Part e. Silly Bud! Making the instance variables of the `Purchase` class `public` would tie the hands of the implementer, making it more difficult to change the representation of `Purchase`.

Part f. Implement history with 3 instance variables: `maxEntries`, and two `IntLists`:

i.



ii. Implementation

```
public class History {

  // Instance Variables:
  private int maxEntries;
  private IntList cashes;
  private IntList credits;

  // Constructor Method:
  public History (int maxEntries) {
    this.maxEntries = maxEntries;
    cashes = empty();
    credits = empty();
  }

  // Instance Methods:
  public void add (int amount, boolean cash) {
    if (length(cashes) + length(credits) < maxEntries) {
      if (cash){
        cashes = prepend(amount, cashes);
      } else {
        credits = prepend(amount, credits);
      }
    }
  }

  public int min (boolean cash){
    if (cash) {
      return minOfList(cashes);
    } else {
      return minOfList(credits);
    }
  }

  public int minOfList (IntList L){
    if(isEmpty(L)){
      return Integer.MAX_VALUE;
    } else {
      return Math.min(head(L), minOfList(tail(L)));
    }
  }

  public int size () {return length(cashes) + length(credits);}

  public int max (boolean cash) { //similar to min--left as exercise.}
```

```
public int average (boolean cash){
    if (cash) {
      if (isEmpty(cashes) {
        return 0;
      } else {
        return sum(cashes)/length(cashes);
      }
    } else {
      if (isEmpty(credits) {
        return 0;
      } else {
        return sum(credits)/length(cashes);
      }
    }
  }

  private int sum (IntList L) {
    if (isEmpty(L)) {
      return 0;
    } else {
      return head(L) + sum(tail(L));
    }
  }

  private int length (IntList L) {
    if (isempty(L)) {
      return 0;
    } else {
      return 1 + length(tail(L));
    }
  }

  public int number (boolean cash){
    if (cash) {
      return length(cashes);
    } else {
      return length(credits);
    }
  }


  public int percentage (boolean cash){
    if (size() != 0){
      if (cashes){
        return length(cashes) * 100/size();
      } else {
        return length(credits) * 100/size();
      }
    } else {
      return 0;
    }
  }
}
```

Part g. Left as exercise for the student.

Problem 2: Squares *(Recursion, Iteration, TurtleWorld, BuggleWorld, PictureWorld, Java Graphics)*

Part a. *Turtle World*

```
public void squares(int n, int len) {
  if (n > 0) {
    drawSquare(len);
    fd(len);
    squares(n - 1, len/2);
    bd(len);
  }
}

public void drawSquare(int length) {
  for (int i = 0; i < 4; i++) {
    fd(length);
    lt(90);
  }
}
```

Part b. *Buggle World*

```
public void squares(int n, int len) {
  if (n > 0) {
    bagelRect(len,len);
    forward(len);
    squares(n - 1, len/2);
    backward(len);
  }
}

// Draw a rectangle of bagels with the given width and height.
// The state of the buggle is invariant under this method.
public void bagelRect(int width, int height) {
  if (height > 0) {
    bagelLine(width);
    left();
    forward(1);
    right();
    bagelRect(width, height-1);
    left();
    backward(1);
    right();
}

// Drop a line of width bagels, leaving the state of the buggle invariant
public void bagelLine(int width) {
  if (width > 0) {
    dropBagel();
    forward();
    bagelLine(width-1);
    backward();
}
```

Part c. *Picture World*

```
public Picture squares(int n, Color c1, Color c2) {
  if (n <= 0) {
    return empty();
  }else {
    return fourPics(empty(), empty(), patch(c1),
                    squares(n-1, c2, c1);
  }
}
```

Part d. *Java Graphics*

```
public void squares (Graphics g, int n, int len, Color c1, Color c2) {
  int x = 0;
  int y = 0;
  while (n > 0) {
    g.setColor(c1);
    g.drawRect(x,y,len,len);

    // Update state variables of iteration
    n = n – 1;
    x = x + len;
    y = y + len/2;
    len = len/2;
    // swap colors
    Color temp = c1;
    c1 = c2;
    c2 = temp;
  }
}
```

Problem 3: Greatest Common Divisor *(Iteration)*

Part a. Here is the iteration table for the iterative calculation of `GCDTail(95,60)`.

A	B
95	60
60	35
35	25
25	10
10	5
5	0

Part b.

```
public static int GCDWhile (int A, int B) {
  while (B != 0) {
    // Need a temporary variable to perform updates!
    // Could make a temp for either A or B; here we choose A.
    int oldA = A;
    A = B;
    B = oldA % B;
  }
  return A;
}
```

Part c. It is *always* possible to express a **`while`** loop as a **`for`** loop, but the result may not be particularly natural. Indeed, in this case it would not be natural to re-express Wyla's GCDTail program as a **for** loop. For loops best "fit" cases where there is a state variable controlling the number of times through the loop that is independent of other state variables. In the case of GCD, the state variable B controls the number of times through the loop, suggesting the following skeleton:

```
public static int GCDFor (int A, int B) {
  for (; B != 0; update) { // No initialization of B necessary
    statements
  }
  return A;
}
```

But what should replace *update* and *statements*? One approach is to leave *update* blank:

```
public static int GCDFor (int A, int B) {
  for (; B != 0;) { // No initialization of B necessary
    // Need a temporary variable to perform updates!
    int oldA = A;
    A = B;
    B = oldA % B;
  }
  return A;
}
```

Indeed, **`for`** `(; test;) {statements}` is always equivalent to **`while`** `(test) {statements}`. But a **`for`** loop that is missing both its initialization and update statements is not very compelling!

If we put `B = A % B` in the *update* position, then the `A` in `A % B` (which is executed after updates) is the value of `A` after the assignment, when we want the value of `A` before the assignment. We need a temporary variable to circumvent this problem, as shown below:

```
public static int GCDFor (int A, int B) {
  int oldA;
  for (; B != 0; B = oldA % B) { // No initialization of B necessary
    oldA = A;
    A = B;
  }
  return A;
}
```

This method is less straightforward than the **`while`** loop, which is clearer and therefore preferred.

Problem 4: Array Reversal *(Arrays, Iteration)*

Part a.

```
public static int [] reverseCopy (int [] a) {
  int len = a.length;
  int [] result = new int[len];
  for (int i = 0; i < len; i++) {
    result[i] = a[(len–1)–i]; // (len–1) is last slot
  }
  return result;
}
```

Part b.

```
public static void reverseInPlace (int [] a) {
  int len = a.length;
  // Swap first (len/2) slots with last (len/2) slots
  // If len is odd, middle slot stays unchanged
  for (int i = 0; i < len/2; i++) {
    // Swap contents of slot i and (len-1)-i
    int old_a_sub_i  = a[i];
    a[i] = a[(len-1)-i];
    a[(len-1)-i] = old_a_sub_i;
  }
}
```

Problem 5: Inversions *(Arrays, Iteration, Lists)*

Part a.

```
// Returns the number of inversions in a:
public static int countInversions (int [] a) {
  int invs = 0; // Counts number of inversions
  for (int i = 0; i < a.length; i++) {
    for (int j = i+1; j < a.length; j++) {
      if a[i] > a[j] {
        invs = invs + 1;
      }
    }
  }
  return invs;
}
```

Part b. Suppose that `PL.` is the prefix used for `PointList` operations:

```
// Returns the inversions in a as a list of points
public static PointList listInversions (int [] a) {
  PointList points = PL.empty(); // Collects inversions as list of points
  for (int i = 0; i < a.length; i++) {
    for (int j = i+1; j < a.length; j++) {
      if a[i] > a[j] {
         points = PL.prepend(new Point(i,j), points);
      }
    }
  }
  return points;
}
```

Problem 6: Converting Betweeen Different Forms of Iteration *(Lists, Arrays, Iteration)*

Part a.

```
// Tail Recursion
public static int weightedSum (IntList L) {
  return weightedSumTail (L, 1, 0);
}

public static int weightedSumTail (IntList L, int index, int total) {
  if (isEmpty(L)) {
    return total;
  } else {
    return weightedSumTail(tail(L), index + 1, (index*head(L)) + total);
  }
}

// While loop
public static int weightedSumWhile (IntList L) {
  int index = 1;
  int total = 0;
  while (!isEmpty(L)) {
    total = total + (index*head(L));
    index = index + 1;
    L = tail(L);
  }
  return total;
}

// For loop
public static int weightedSumFor (IntList L) {
  int index = 1;
  int total = 0;
  for (; !isEmpty(L); L = tail(L);) {
    total = total + (index*head(L));
    index = index + 1;
  }
  return total;
}
```

Part b.

```
// While loop
public static boolean isMember (int n, int [] a) {
  int i = a.length - 1;
  while ((i >= 0) && (a[i] != n)) {
    i = i - 1;
  }
  return (i >= 0); // Will only be true if n is in a.
}
```

```
// For loop
public static boolean isMemberFor (int n, int [] a) {
  for (int i = a.length – 1; i>=0; i--) {
    if a[i] == n {
      return true;
    }
  }
  return false;
}

// Tail recursion
public static boolean isMemberIter (int n, int [] a) {
  return isMemberTail(n, a, a.length - 1);
}

public static boolean isMemberTail (int n, int [] a, int i) {
    if (i < 0) {
      return false;
    } else if (a[i] == n) {
      return true;
    } else {
      return isMemberTail (n, a, i-1);
    }
}
```

Part c.

```
// For loop
public static void partialSum (int [] a) {
  int sum = 0;
  for (int i = 0; i < a.length; i++) {
    sum = sum + a[i];
    a[i] = sum;
  }
}

// While loop
public static void partialSumWhile (int [] a) {
  int i = 0;
  int sum = 0;
  while (i < a.length) {
   sum = sum + a[i];
    a[i] = sum;
    i++;
  }
}

// Tail recursion
public static void partialSumIter (int [] a) {
  return partialSumTail(a,0, 0);
}

public static void partialSumTail (int [] a, int i, int sum) {
  if (i < a.length) {
    int newsum = sum + a[i];
    a[i] = newsum;
    partialSumTail(a, i+1, newsum);
  }
}
```

Part d.

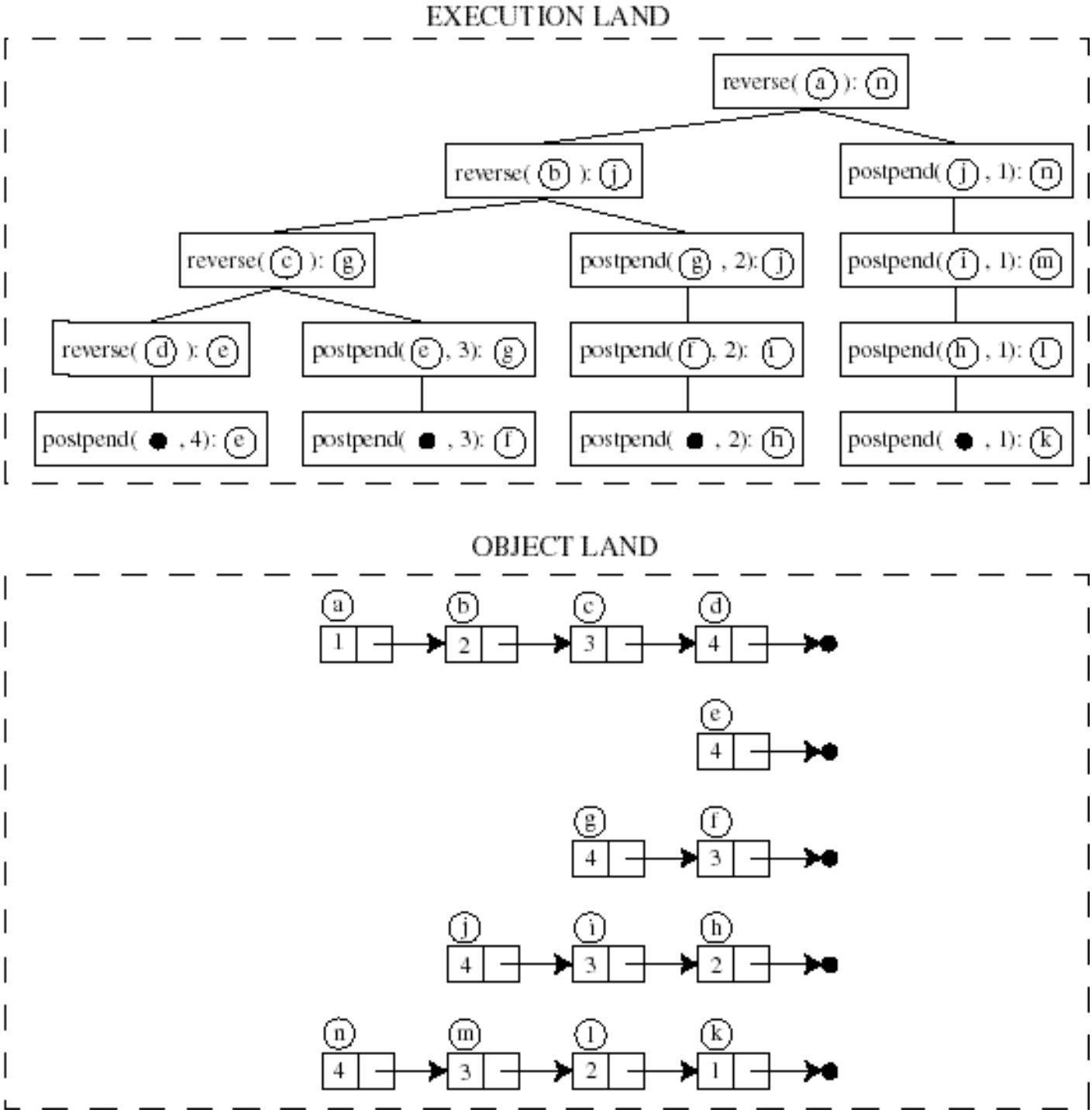
```
// Tail recursion
public static void squiggle (Graphics g, int x1, int y1, int x2, int y2) {
  if ((x1 > 0) || (y1 > 0) || (x2 > 0) || (y2 > 0)) {
    g.drawLine(x1, y1, x2, y2);
    squiggle(g, x2, y2, y1/4, x1*2);
  }
}

// While loop
public static void squiggleWhile (Graphics g, int x1, int y1, int x2, int y2) {
  while ((x1 > 0) || (y1 > 0) || (x2 > 0) || (y2 > 0)) {
    g.drawLine(x1, y1, x2, y2);
    // Introduce temporaries for x1, y1 so don't lose their values!
    old_x1 = x1;
    old_y1 = y1;
    x1 = x2;
    y1 = y2;
    x2 = old_y1/4
    y2 = old_x1*2;
  }
}

// For loop
public static void squiggleFor (Graphics g, int x1, int y1, int x2, int y2) {
  // A particularly uncompelling example of a for loop!
  for (;(x1 > 0) || (y1 > 0) || (x2 > 0) || (y2 > 0);) {
    g.drawLine(x1, y1, x2, y2);
    // Introduce temporaries for x1, y1 so don't lose their values!
    old_x1 = x1;
    old_y1 = y1;
    x1 = x2;
    y1 = y2;
    x2 = old_y1/4
    y2 = old_x1*2;
  }
}
```

Problem 7: Converting Betweeen Arrays and Lists *(Lists, Arrays, Iteration)*

```
public static int [] listToArray (IntList L) {
  int [] result = new int [IL.length(L)];
  int i = 0;
  while (! isEmpty(L)) {
    result[i] = IL.head(L);
    L = IL.tail(L);
    i = i + 1;
  }
  return result;
}

public static IntList arrayToList (int [] a) {
  IntList L = IL.empty();
  // traverse the array from back to front in order
  // to prepend elements in the correct order.
  for (int i = a.length – 1; i >= 0; i--) {
    L = prepend(a[i], L);
  }
  return L;
}
```

Problem 9: Iterative List Reversal *(Invocation Trees, Recursion, Iteration, Lists,)*

Part a. Below is a so-called *invocation tree* that summarizes all the execution frames in ExecutionLand for the given example. A node in the invocation tree stands for a execution frame whose internal details are not shown. The value after a colon is the result returned by the invocation (in this case, references to list nodes in ObjectLand).



Part b.

Tail recursive solution:

```
public static IntList reverseIter (IntList L) {
  return reverseTail(L, IL.empty());
}

public static IntList reverseTail (IntList list, IntList result) {
  if (IL.isEmpty(list)) {
    return result;
  } else {
    return reverseTail(IL.tail(list), IL.prepend(IL.head(list), result));
  }
}
```

While loop solution:

```
public static IntList reverseWhile (IntList L) {
  IntList result = IL.empty();
  while (! IL.isEmpty(L)) {
    result = IL.prepend(IL.head(L), result);
    L = IL.tail(L);
  }
  return result;
}
```

For loop solution:

```
public static IntList reverseFor (IntList L) {
  IntList result = IL.empty();
  for (; ! IL.isEmpty(L); L = IL.tail(L)) // Note empty initializer
    result = IL.prepend(IL.head(L), result);
  }
  return result;
}
```

Problem 10: Bank Accounts *(Data Abstraction, Lists)*

Part a.

```
class Account {

  private int savings;
  private int total;

  public account () {
    this.savings = 0;
   this.total = 0;
  }

  public int getSavings() {return this.savings;}
  public int getChecking() {return this.total – this.savings;}
  public int getTotal() {return this.total;}

  public void depositToSavings(int amountToAdd) {
    this.savings = this.savings + amountToAdd;
    this.total = this.total + amountToAdd;
  }

  public void transferFromSavingsToChecking(int transferAmount) {
    this.savings = this.savings - transferAmount;
  }

  public void withdrawFromChecking(int withdrawalAmount) {
    this.total = this.total - withdrawalAmount;
  }
}
```

Part b.

```
class Account {

  private IntList accountInfo; // List of savings, checking, total

  public account () {set(0,0,0);}

  // Very useful helper method
  private set (int s, int c, int t) {
    accountInfo = IL.prepend(s, IL.prepend(c, IL.prepend(t IL.empty())));
  }

  public int getSavings() {return IL.head(accountInfo);}
  public int getChecking() {return IL.head(IL.tail(accountInfo));}
  public int getTotal() {return IL.head(IL.tail(IL.tail(accountInfo)));}

  public void depositToSavings(int amount) {
    set(getSavings() + amount, getChecking(),getTotal() + amount);
  }

  public void transferFromSavingsToChecking(int amount) {
    set(getSavings() – amount, getChecking() + ammount, getTotal());
  }

  public void withdrawFromChecking(int amount) {
    set(getSavings(), getChecking() – amount, getTotal() – amount);
  }
}
```