Test Cases for JADE System

Cristinel Mateis
Institut für Informationssysteme, TU Wien
A-1040 Wien, Austria
email: mateis@dbai.tuwien.ac.at

January 15, 1999

This paper contains several programs which exhibit some of the most common basic bugs [1] that may occur in Java programs. One might say that some methods used here are useless and they could have been avoided or they could have been written in a simpler way, hence possible bugs introduced by these methods could have been avoided. The goal was to create simple situations where such bugs may occur. Implementation efficiency is of secondary interest.

The methods were implemented using the level 1 Java subset.

1 Basic Mathematical Operations

The program from Figure 1 implements the basic mathematical operations for the addition and multiplication of two integer numbers. The methods **myMult** and **myAdd** are defined in recursive form and the former is based on the latter and on **goToNull** which is another user-defined method. Moreover, *myAdd* makes use of *goToNull* and another user-defined method, **succ**. For the addition operation, the idea is to increment or decrement by one the value of the second operand a number of times equal to the absolute value of the first operand. For the multiplication operation, the idea is to add the second operand to the global result which initially is zero, a number of times equal to the absolute value of the first operand.

Informally, if the first argument x of $myMult(x, y)$ is zero the method simply returns zero, otherwise it returns the result of the addition of y if $x > 0$ (resp., $-y$ if $x < 0$) to $myMult(x-1, y)$ (resp., $myMult(x+1, y)$).

If the first argument x of $myAdd(x, y)$ is zero the method simply returns the second argument y, otherwise it returns the successor of $myAdd(goToNull(x), y)$ in

[1] A *bug* is an error in the structure of a program.

the indicated order (e.g., ascending or descending), where $goToNull(x)$ increments or decrements by one the value of x, depending on whether x is negative or positive, respectively. All three methods are correctly implemented and their behaviors can be summarized as follows.

$$
\begin{aligned}
myMult(x,y) &= xy && \forall x,y \in \mathbf{Z} \\
myAdd(x,y) &= x+y && \forall x,y \in \mathbf{Z} \\
succ(x,ascending) &= \begin{cases} x+1 & \text{if } ascending = true \\ x-1 & \text{otherwise} \end{cases} && \forall x \in \mathbf{Z} \\
goToNull(x) &= \begin{cases} x+1 & \text{if } x<0 \\ x-1 & \text{otherwise} \end{cases} && \forall x \in \mathbf{Z}
\end{aligned}
$$

where $\mathbf{Z}$ is the set of the integer numbers. Possible bugs can be introduced as follows.

1. Replace line 9 by **if (x == 1)**.
   The effect of this bug is that $myMult(x,y)$ either does not terminate or returns an erroneous result. In particular, if $x \leq 0$, then $goToNull(x)$ cannot reach the value 1 which is the condition of exit from the recursive call of $myMult$ and $myMult$ tends to call itself recursively infinitely many times and then a stack overflow error results. On the other hand, if $x > 0$ y is added to the global result once less and $myMult(x,y)$ returns $(x-1)y$ instead of xy.

   The erroneous behavior of $myMult$ can be summarized as follows.

$$myMult(x,y) = \begin{cases} -\infty & \text{if } x \leq 0 \\ xy - x & \text{if } x > 0 \end{cases} \quad \forall x,y \in \mathbf{Z}.$$

2. Replace line 10 by **return y**.
   In this case, when $myMult(x,y)$ is called, the global result does not start from zero, but from y and $myMult$ returns $(x+1)y$ instead of xy, whatever the values of $x,y \in \mathbf{Z}$. The erroneous behavior of $myMult$ can be summarized as follows.

$$myMult(x,y) = xy + y \quad \forall x,y \in \mathbf{Z}.$$

3. Replace line 13 by **return myAdd(x,myMult(goToNull(y),x))**.
   In this case, $myMult(x,y)$ returns an erroneous result only if $x > 0$ and $y < 0$. The erroneous behavior can be summarized as follows.

$$myMult(x,y) = \begin{cases} xy + 2x & \text{if } x > 0 \text{ and } y < 0 \\ xy & \text{otherwise} \end{cases} \quad \forall x,y \in \mathbf{Z}.$$

4. Replace line 13 by **return myAdd(-y,myMult(goToNull(x),y))**.
   The effect of this bug is that if $x > 0$, then $myMult(x, y)$ considers $x < 0$ and returns $-xy$ instead of xy. The erroneous behavior can be summarized as follows.

$$myMult(x, y) = \begin{cases} -xy & \text{if } x > 0 \\ xy & \text{if } x \leq 0 \end{cases} \quad \forall x, y \in \mathbf{Z}.$$

5. Replace line 15 by **return myAdd(-x,myMult(goToNull(y),x))**.
   In this case, $myMult(x, y)$ returns an erroneous result only if $x < 0$ and $y \geq 0$. The erroneous behavior can be summarized as follows.

$$myMult(x, y) = \begin{cases} xy - 2x & \text{if } x < 0 \text{ and } y \geq 0 \\ xy & \text{otherwise} \end{cases} \quad \forall x, y \in \mathbf{Z}.$$

6. Replace line 15 by **return myAdd(y,myMult(goToNull(x),y))**.
   The effect of this bug is that if $x < 0$ then $myMult(x, y)$ considers $x > 0$ and returns $-xy$ instead of xy. The erroneous behavior can be summarized as follows.

$$myMult(x, y) = \begin{cases} -xy & \text{if } x < 0 \\ xy & \text{if } x \geq 0 \end{cases} \quad \forall x, y \in \mathbf{Z}.$$

7. Replace line 18 by **if (x == 1)**.
   The effect of this bug results in two kinds of anomalies: (i) if the initial value of the argument x of *myAdd* is zero or negative then $goToNull(x)$ can never reach the value 1 which is the condition of exit from the recursive call of *myAdd* and consequently, *myAdd* tends to call itself recursively infinitely many times and a stack overflow error results, and (ii) if the initial value of the argument x of *myAdd* is positive then the method terminates but the result is less by one than the correct result.

   The effect of the erroneous behavior of *myAdd* on $myMult(x, y)$ is that $myMult$ does not terminate and a stack overflow error results when $x \neq 0$ and $xy \leq 0$. If $x \neq 0$ and $xy > 0$, $myMult$ terminates but the result is less by $|x|$ (i.e., the number of the calls to *myAdd*) than the correct result. $myMult(x, y)$ returns the correct result only if $x = 0$.

   The erroneous behaviors of $myMult$ and *myAdd* can be summarized as follows.

$$myMult(x, y) = \begin{cases} -\infty & \text{if } x \neq 0 \text{ and } xy \leq 0 \\ xy - |x| & \text{if } x \neq 0 \text{ and } xy > 0 \\ xy & \text{if } x = 0 \end{cases} \quad \forall x, y \in \mathbf{Z}$$

$$myAdd(x, y) = \begin{cases} -\infty & \text{if } x \leq 0 \\ x + y - 1 & \text{otherwise} \end{cases} \quad \forall x, y \in \mathbf{Z}$$

8. Replace line 19 by **return x**.
The effect of this bug is that the result outputed by $myAdd(x, y)$ is the initial value of the argument x. When the last recursive call of $myAdd$ is performed, the actual value of the argument x is zero whatever are the initial values of x and y. Thus, the last call of $myAdd$ returns the null value which is incremented (resp., decremented) x many times by the previous recursive calls of $myAdd$. It follows that the outputed result is x, whatever are the values of x and y.

   The effect of the erroneous behavior of $myAdd$ on $myMult(x, y)$ is that if $x \neq 0$, the result of $myMult$ is always $|y|$ if $xy \geq 0$ and $-|y|$ if $xy < 0$. The result of $myMult(x, y)$ is correct only if $x = 0$.

   This erroneous behavior of $myMult$ and $myAdd$ can be summarized as follows.

$$
\begin{aligned}
myMult(x,y) &= \begin{cases} |y| & \text{if } x \neq 0 \text{ and } xy \geq 0 \\ -|y| & \text{if } x \neq 0 \text{ and } xy < 0 \\ xy & \text{if } x = 0 \end{cases} && \forall x, y \in \mathbf{Z} \\
myAdd(x,y) &= x && \forall x, y \in \mathbf{Z}
\end{aligned}
$$

9. Replace line 21 by **return succ(myAdd(goToNull(y),x),x>0)**.
In this case the problem is that there is an incoherence between the condition of exit from the recursive call and the recursive call of $myAdd$. In particular, the exit condition is tested on the first argument of $myAdd$ and, if it fails $myAdd$ recursively calls itself as if the condition were tested on the second argument. Moreover, there exists an incoherence between the second argument of *succ* (i.e., $x > 0$), which induces the successor order of the first argument, and the argument which should go to zero (i.e., the argument of $goToNull$). However, in some cases $myAdd(x, y)$ returns the correct result but this depends on the values of x and y.

   The effect of the erroneous behavior of $myAdd$ on $myMult$ is that $myMult(x, y)$ returns the correct result only if $xy \geq 0$. If $xy < 0$, the result outputed by $myMult(x, y)$ is less by 2 than the correct result.

   The erroneous behavior of $myMult$ and $myAdd$ can be summarized as follows.

$$
\begin{aligned}
myMult(x,y) &= \begin{cases} xy - 2 & \text{if } xy < 0 \\ xy & \text{if } xy \geq 0 \end{cases} && \forall x, y \in \mathbf{Z} \\
|myAdd(x,y) - (x+y)| &\leq 2 && \forall x, y \in \mathbf{Z}
\end{aligned}
$$

10. Replace line 25 by **return x++**.
The effect of this bug is that the method $myAdd(x, y)$ returns the correct

value only if x is negative. If x is positive, $myAdd(x, y)$ simply returns y. The reason is that the value returned by $succ(x, ascending)$, if $ascending$ is true, is exactly x, because the value of x is incremented only after its current value is used.

The erroneous behaviors of $myMult$, $myAdd$, and $succ$ can be summarized as follows.

$$myMult(x,y) = \begin{cases} xy & \text{if } xy \leq 0 \\ 0 & \text{if } xy > 0 \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$myAdd(x,y) = \begin{cases} y & \text{if } x > 0 \\ x + y & \text{otherwise} \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$succ(x, ascending) = \begin{cases} x & \text{if } ascending = true \\ x - 1 & \text{otherwise} \end{cases} \qquad \forall x \in \mathbf{Z}$$

11. Replace line 27 by **return x−−**.
The effect of this bug is that the method $myAdd(x, y)$ returns the correct value only if x is positive. If x is negative, $myAdd(x, y)$ simply returns y. The reason is the erroneous behavior of $succ(x, ascending)$ when $ascending$ is false. In this case, $succ(x, ascending)$ returns exactly x, because the value of x is decremented only after its current value is used.

    The erroneous behavior of $myMult$, $myAdd$, and $succ$ can be summarized as follows.

$$myMult(x,y) = \begin{cases} xy & \text{if } xy \leq 0 \\ 0 & \text{if } xy > 0 \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$myAdd(x,y) = \begin{cases} x + y & \text{if } x \geq 0 \\ y & \text{otherwise} \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$succ(x, ascending) = \begin{cases} x + 1 & \text{if } ascending = true \\ x & \text{otherwise} \end{cases} \qquad \forall x \in \mathbf{Z}$$

12. Replace line 25 by **return −−x**.
The effect of this bug is that the method $myAdd(x, y)$ returns the correct value only if x is negative. If x is positive, $myAdd(x, y)$ returns $y - x$. The reason is that the result of $succ(y, ascending)$ when $ascending$ is true (i.e., x is positive) is as if $ascending$ were false (i.e., x were negative), respectively it returns $y - 1$.

The erroneous behavior of $myMult$, $myAdd$, and $succ$ can be summarized as follows.

$$myMult(x,y) = \begin{cases} xy & \text{if } xy = 0 \\ -|xy| & \text{otherwise} \end{cases} \quad \forall x, y \in \mathbf{Z}$$

$$myAdd(x,y) = \begin{cases} -x+y & \text{if } x > 0 \\ x+y & \text{otherwise} \end{cases} \quad \forall x, y \in \mathbf{Z}$$

$$succ(x, ascending) = x - 1 \quad \forall x \in \mathbf{Z},\ \forall\ ascending \in \{\mathbf{false}, \mathbf{true}\}$$

13. Replace line 27 by **return ++x**.
The effect of this bug is that the method $myAdd(x,y)$ returns the correct value only if x is positive. If x is negative, $myAdd(x,y)$ returns $y - x$. The reason is that the behavior of $succ(y, ascending)$ when $ascending$ is false (i.e., x is negative) is as if $ascending$ were true (i.e., x were positive), respectively it returns $y + 1$.

   The erroneous behavior of $myMult$, $myAdd$, and $succ$ can be summarized as follows.

$$myMult(x,y) = \begin{cases} xy & \text{if } xy = 0 \\ |xy| & \text{otherwise} \end{cases} \quad \forall x, y \in \mathbf{Z}$$

$$myAdd(x,y) = \begin{cases} x+y & \text{if } x \geq 0 \\ -x+y & \text{otherwise} \end{cases} \quad \forall x, y \in \mathbf{Z}$$

$$succ(x, ascending) = x + 1 \quad \forall x \in \mathbf{Z},\ \forall\ ascending \in \{\mathbf{false}, \mathbf{true}\}$$

14. Replace line 31 by **return x++**.
In this case $myAdd(x,y)$ returns the correct result only if $x \geq 0$. If $x < 0$, the condition of exit from the recursive call of $myAdd$ is never satisfied, because $goToNull(x)$ returns the value of x unchanged and a stack overflow error results.

   The erroneous behavior of $myMult$, $myAdd$, and $goToNull$ can be summarized as follows.

$$myMult(x,y) = \begin{cases} -\infty & \text{if } x < 0 \text{ or } y < 0 \\ xy & \text{otherwise} \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$myAdd(x,y) = \begin{cases} x+y & \text{if } x \geq 0 \\ -\infty & \text{otherwise} \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$goToNull(x) = \begin{cases} x & \text{if } x < 0 \\ x-1 & \text{otherwise} \end{cases} \qquad \forall x \in \mathbf{Z}$$

15. Replace line 31 by **return −−x**.
    As in the previous case, $myAdd(x,y)$ returns the correct result only if $x \geq 0$. If $x < 0$, the condition of exit from the recursive call of *myAdd* is never satisfied, because $goToNull(x)$ returns a value farther from zero instead of nearer to zero w.r.t. the value of x. A stack overflow error results.

    The erroneous behavior of *myMult*, *myAdd*, and *goToNull* can be summarized as follows.

$$myMult(x,y) = \begin{cases} -\infty & \text{if } x < 0 \text{ or } y < 0 \\ xy & \text{otherwise} \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$myAdd(x,y) = \begin{cases} x+y & \text{if } x \geq 0 \\ -\infty & \text{otherwise} \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$goToNull(x) = x - 1 \qquad \forall x \in \mathbf{Z}$$

16. Replace line 33 by **return x−−**.
    In this case $myAdd(x,y)$ returns the correct result only if $x \leq 0$. If $x > 0$, the condition of exit from the recursive call of *myAdd* is never satisfied, because $goToNull(x)$ returns the value of x unchanged and a stack overflow error results.

    The erroneous behavior of *myMult*, *myAdd*, and *goToNull* can be summarized as follows.

$$myMult(x,y) = \begin{cases} xy & \text{if } x \leq 0 \text{ and } y \geq 0 \\ -\infty & \text{otherwise} \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$myAdd(x,y) = \begin{cases} x+y & \text{if } x \leq 0 \\ +\infty & \text{otherwise} \end{cases} \qquad \forall x, y \in \mathbf{Z}$$

$$goToNull(x) = \begin{cases} x+1 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \qquad \forall x \in \mathbf{Z}$$

17. Replace line 33 by **return ++x**.

    As in the previous case, $myAdd(x, y)$ returns the correct result only if $x \leq 0$. If $x > 0$, the condition of exit from the recursive call of $myAdd$ is never satisfied, because $goToNull(x)$ returns a value farther from zero instead of nearer to zero w.r.t. the value of x. A stack overflow error results.

    The erroneous behavior of $myMult$, $myAdd$, and $goToNull$ can be summarized as follows.

$$
\begin{aligned}
myMult(x, y) &= \begin{cases} xy & \text{if } x \leq 0 \text{ and } y \geq 0 \\ -\infty & \text{otherwise} \end{cases} && \forall x, y \in \mathbf{Z} \\
myAdd(x, y) &= \begin{cases} x + y & \text{if } x \leq 0 \\ +\infty & \text{otherwise} \end{cases} && \forall x, y \in \mathbf{Z} \\
goToNull(x) &= x + 1 && \forall x \in \mathbf{Z}
\end{aligned}
$$

2 Operations on Lists of Integers

The primitive types of the level 1 Java subset are **integer** and **boolean**. The class shown in Figure 2 implements a list of integers and some basic operations on lists. The internal data structure consists of an array of integers which stores the elements of the list and an integer which gives the current length of the list. When an object of the class is created, if the constructor does not receive any positive integer (specifying the size of the array), the memory for 100 integers is allocated.

The method $isEmpty()$ returns **true** if the list is empty and **false** otherwise.

The method $update$ is an overloaded method and it is used only for internal purposes (i.e., it is declared $private$). The first definition, $update(int\ i,\ list\ temp)$, simply adds to the list $temp$ the elements from $this.arrayOfInts$ whose indexes are greater or equal to i, if any. The second definition, $update(int\ i,\ list\ temp,\ int\ val)$, performs the same task of the previous one, but skips from $this.arrayOfInts$ all elements equal to val.

The method $listClone()$ returns a copy of the list. First, an empty list $temp$ is created. $update(0, temp)$ is then invoked and all elements of the current list are added to $temp$ exactly in the same order.

The method $first()$ returns the first element of the list. If the list is empty, then the method returns the constant -100000 which is considered to be a kind of error message.

The method $rest()$ returns a new list containing all elements of the list, apart from the first one.

```java
class arithm {
   public static void main(String[] args) {
      int temp1 = myAdd(-15,6);
      int temp2 = myMult(-15,6);
      System.out.println("myAdd(-15,6) = " + temp1);
      System.out.println("myMult(-15,6) = " + temp2);
   }

   static int myMult(int x, int y) {
      if (x == 0)
         return 0;
      else
         if (x > 0)
            return myAdd(y,myMult(goToNull(x),y));
         else
            return myAdd(-y,myMult(goToNull(x),y));
   }

   static int myAdd(int x, int y) {
      if (x == 0)
         return y;
      else
         return succ(myAdd(goToNull(x),y),x>0);
   }

   static int succ(int x, boolean ascending) {
      if (ascending)
         return ++x;
      else
         return --x;
   }

   static int goToNull(int x) {
      if (x < 0)
         return ++x;
      else
         return --x;
   }
}
```

Figure 1: User-Defined Addition and Multiplication Operations

The method $addElement(x)$ appends x at the end of the array and increments the length of the list.

The method $delElement(x)$ removes all occurrences of x from the list. First, an empty list *temp* is created. $update(0, temp, x)$ is then invoked and all elements of the current list different from x are added to *temp* exactly in the same order. Finally, the current list is substituted by the new created list *temp*.

The method $includes(x)$ is defined in recursive form and returns **true** if x is contained in the list.

The method $min()$ returns the least integer of the list. If the list is empty, then the method returns the constant -100000 indicating that an error occurred.

Bugs can be caused by the fact that, in Java, the first index of an array is 0 and not 1, like in other common languages. Very often, programmers start searching elements of an array from the index 1 and consequently, the first element of the array, which has the index 0, is not considered.

Another source of bugs in Java programs is the fact that some data types, such as objects and arrays, are reference data types. The value of a variable whose data type is a reference type is a reference (or a pointer in other terminology) to the actual data. If $l1$ and $l2$ are two objects of the class *list*, if the assignment $l1 = l2$ is performed, then $l1$ will point to the object pointed by $l2$. A C++ programmer might think that, after the assignment, $l1$ will contain a new instance of the class *list* equal to $l2$, but in fact, $l1$ and $l2$ are two references pointing to the same instance of the class *list*. On the other hand, the parameters of a method which are of reference data types, are always passed by reference, even if it might seem that they are passed by value.

Following, we show how possible bugs can be introduced in the implementation of the methods of the class *list* from Figure 2.

1. Replace line 16 by **if (i <= length)**.
   The consequence of this bug is that an element, which does not exist in the list, is accessed and copied to *temp*. That is, *temp* contains one element too much and the value of this element is the value found in the memory at a location which has not been accessed by any *addElement*. The effects of this bug on the behavior of other methods are as following.

   $listClone()$ and $rest()$ returns lists which have one element more than expected.

   Consider a call $includes(x)$. If the list is not empty and it does not contain x, then $includes(x)$ does not terminate, because the length of $rest()$ does not decrease (i.e., rest().length = this.length), hence $includes(x)$ is called infinitely many times at line 60.

   The behavior of $min()$ is similar to those of $includes()$. If $length > 1$, $min()$ does not terminate and a stack overflow error results. The reason is that

$rest().min()$ does not terminate, because the length of $rest()$ does not decrease (i.e., rest().length = this.length), hence $min()$ is called infinitely many times at line 69.

2. Replace line 18 by **update(i,temp)**.
   In this case, if $i < length$, *update* tends to call itself infinitely many times, and each time adds to *temp* the same element, $arrayOfInts[i]$. Since the memory allocated for the elements of a list is finite, an "array index out of bounds" error results. The impact of this bug on the behaviors of *listClone*, *rest*, *includes*, and *min* is obvious.

3. Replace line 22 by **if (i <= length)**.
   The effect of this bug is that, when *delElement* is called, a spurious element is added to the list.

4. Replace line 37 by **return arrayOfInts[1]**.
   In this case, $first()$ returns the second element of the list instead of the first one. Consequently, $includes(x)$ does not return the correct result if x is the first element in the list, or x is not in the list and the spurious value $arrayOfInts[length]$ is exactly x. Moreover, when $min()$ is called, the spurious value $arrayOfInts[length]$ is also taken into consideration and it might influence the result.

5. Replace line 41 by **update(2,temp)**.
   In this case, the list returned by $rest()$ will miss the first element. As a consequence, *includes* and *min* will consider only a subset of the list and their results may only casually be correct.

6. Replace line 45 by **arrayOfInts[++length] = el**.
   The effect of this bug is that the elements of a list of size n are stored at the locations of intexes from 1 to n. However, the other methods consider that the first element of the list is stored at the location of index 0, hence the last added element can be viewed only after a further element is added. Moreover, since storing elements starts at location of index 1, the element at location of index 0 will have an indefinite value. Thus, the lists returned by *listClone*, *rest*, and *delElement* will miss the last element. The methods *includes* and *min* may also return incorrect values, because during the recursive calls of $rest()$ at line 60 (resp., 69), elements of the list are lost.

7. Replace line 49 by **update(1,temp,el)**.
   The effect of this bug is that the first element of the list is removed, even if it is different from *el*.

8. Replace line 69 by **if (first() > rest().min())**.
   It is clear that in this case, $min()$ would return the maximum instead of the minimum element of the list.

Next we show how the principal sorting algorithms can be implemented as methods of the class *list* and we analyze possible bugs that a programmer might commit while implementing them.

2.1 Bubblesort

The simplest sorting method one can devise is perhaps the bubblesort algorithm. The basic idea behind bubblesort is to imagine that the elements to be sorted are kept in an array held vertically. The elements of low values are "light" and bubble up to the top. We make repeated passes over the array, from bottom to top. As we go, if two adjacent elements are out of order, that is, if the "lighter" one is below, we reverse them. The effect of this operation is that on the first pass the "lightest" element, that is, the element with the lowest value, rises all the way to the top. On the second pass, the second lowest element rises to the second position, and so on. We need not, on pass two, try to bubble up to position one, because we know the lowest element already resides there. In general, pass i need not try to bubble up past position i. A correct implementation of this algorithm is showed in Figure 3.

2.2 Shellsort

The procedure Shellsort of Fig. 4 sorts the elements of a list of n elements by sorting $n/2$ pairs $(l[i], l[n/2+i])$ for $1 \leq i \leq n/2$ in the first pass, $n/4$ four-tuples $l[i], l[n/4+i], l[n/2+i], l[3n/4+i])$ for $1 \leq i \leq n/4$ in the second pass, $n/8$ eight-tuples in the third pass, and so on. In each pass the sorting is done using insertion sort in which we stop sorting once we encounter two elements in the proper order.

2.3 Insertion Sort

The procedure Insertionsort of Fig. 5, on the i^{th} pass, inserts the i^{th} element of a list l into its rightful place among $l[1]$, $l[2]$, ..., $l[i-1]$, which were previously placed in sorted order. After doing this insertion, $l[1]$, ..., $l[i]$ are in sorted order. When deciding whether to push $l[i]$ up before position j, we must first check whether $j = 0$, and if not, we compare $l[i]$ (which is now in position j) with $l[j-1]$.

2.4 Selection Sort

The idea behind "selection sorting" is also elementary. In the i_{th} pass, we select the lowest value of the list, among $l[i], \ldots, l[n]$, and swap it with $l[i]$. As a result, after

```
class list {
   int[] arrayOfInts;
   int   length;

   list() {
      arrayOfInts = new int[100];
      length = 0;
   }

   list(int size) {
      arrayOfInts = new int[size];
      length = 0;
   }

   boolean isEmpty() {
      return (length == 0);
   }

   private void update(int i, list temp) {
      if (i < length) {
         temp.addElement(arrayOfInts[i]);
         update(i+1,temp);
      }
   }

   private void update(int i, list temp, int val) {
      if (i < length) {
         if (arrayOfInts[i] != val)
            temp.addElement(arrayOfInts[i]);
         update(i+1,temp,val);
      }
   }

   list listClone() {
      list temp = new list();
      update(0,temp);
      return temp;
   }
```

it follows

```
int first() {
   if (isEmpty())
      return -100000;
   else
      return arrayOfInts[0];
}

list rest() {
   list temp = new list();
   update(1,temp);
   return temp;
}

void addElement(int el) {
   arrayOfInts[length++] = el;
}

void delElement(int el) {
   list temp = new list();
   update(0,temp,el);
   this.arrayOfInts = temp.arrayOfInts;
   this.length = temp.length;
}

boolean includes(int x) {
   if (isEmpty())
      return false;
   else
      if (first() == x)
         return true;
      else
         return rest().includes(x);
}
```

it follows

```
   int min() {
      if (isEmpty())
         return -100000;
      else
         if (length == 1)
            return first();
         else
            if (first() <= rest().min())
               return first();
            else
               return rest().min();
   }
}
```

Figure 2: The List Class with some Basic Operations on Lists

```
private void bubble(int i, int j) {
   if (i < length)
      if (j > i) {
         if (arrayOfInts[j] < arrayOfInts[i]) {
            int temp = arrayOfInts[i];
            arrayOfInts[i] = arrayOfInts[j];
            arrayOfInts[j] = temp;
         }
         bubble(i,j-1);
      }
      else
         bubble(i+1,length-1);
}

void bubbleSort() {
   bubble(0,length-1);
}
```

Figure 3: The Bubblesort Method

```
private void shell1(int incr, int j) {
   if (j >= 0) {
      if (arrayOfInts[j] > arrayOfInts[j+incr]) {
         int temp = arrayOfInts[j];
         arrayOfInts[j] = arrayOfInts[j+incr];
         arrayOfInts[j+incr] = temp;
         shell1(incr,j-incr);
      }
   }
}

private void shell2(int incr, int i) {
   if (i < length) {
      shell1(incr,i-incr);
      shell2(incr,i+1);
   }
}

private void shell3(int incr) {
   if (incr > 0) {
      shell2(incr,incr);
      shell3(incr/2);
   }
}

void shellSort() {
   shell3(length/2);
}
```

Figure 4: The Shellsort Method

```
private void insertion1(int j) {
   if ((j > 0) && (arrayOfInts[j] < arrayOfInts[j-1])) {
      int temp = arrayOfInts[j-1];
      arrayOfInts[j-1] = arrayOfInts[j];
      arrayOfInts[j] = temp;
      insertion1(j-1);
   }
}

private void insertion2(int i) {
    if (i < length) {
       insertion1(i);
       insertion2(i+1);
    }
 }

 void insertionSort() {
    insertion2(1);
 }
```

Figure 5: The Insertion Sort Method

i passes, the i lowest values will occupy $l[1], \ldots, l[i]$, in sorted order.

2.5 Quicksort

The quicksort algorithm presented in Figure 7 is probably the most efficient algorithm for internal sorting. The essence of quicksort is to sort an array $l[1], \ldots, l[n]$ by picking some value v in the array as a *pivot* element around which to rearrange the elements in the array. We hope that the pivot is near the median value in the array, so that it is preceded by about half the elements and followed by about half. We permute the elements in the array so that for some j, all the values less than v appear in $l[1], \ldots, l[j]$, and all those equal or greater than v appear in $l[j+1], \ldots, l[n]$. We then apply quicksort recursively to $l[1], \ldots, l[j]$ and to $l[j+1], \ldots, l[n]$ to sort both these groups of elements. Since all values in the first group precede all values in the second group, the entire array will thus be sorted.

The method $findPivot$ determines whether the values $l[i], \ldots, l[j]$ are all the same. If $findPivot$ never finds two different values, it returns -1. Otherwise, it returns the index of the larger of the first two different values. This larger value becomes the pivot element.

Next, we have to permute $l[i], \ldots, l[j]$, in place, so that all the elements smaller

```
private void selection1(int j) {
   if (j == length-1)
      return j;
   else {
      int lowIndex = selection1(j+1);
      if (arrayOfInts[j] < arrayOfInts[lowIndex])
         return j;
      else
         return lowIndex;
   }
}

private void selection2(int i) {
   if (i < length - 1) {
      int lowIndex = selection1(i);
      int temp = arrayOfInts[i];
      arrayOfInts[i] = arrayOfInts[lowIndex];
      arrayOfInts[lowIndex] = temp;
      selection2(i+1);
   }
}

void selectionSort() {
   selection2(0);
}
```

Figure 6: The Selection Sort Method

than the pivot value appear to the left of the others. The parameters $leftIndex$ and $rightIndex$ of the method *partition* are to be viewed as cursors which are initially at the left and right ends of the portion of l being sorted, respectively. At all times, the elements to the left of $leftIndex$, that is, $l[i], \ldots, l[leftIndex - 1]$ will have values less than the pivot. Elements to the right of $rightIndex$, that is, $l[rightIndex + 1], \ldots, l[j]$, will have values equal to or greater than the pivot, and elements in the middle will be mixed. Initially, $i = leftIndex$ and $j = rightIndex$ so the above statement holds since nothing is to the left of $leftIndex$ or to the right of $rightIndex$. We repeatedly do the following steps, which move $leftIndex$ right and $rightIndex$ left, until they finally cross, whereupon $l[i], \ldots, l[leftIndex-1]$ will contain all the values less than the pivot and $l[rightIndex + 1], \ldots, l[j]$ all values equal to or greater than the pivot.

1. *Scan.* Move $leftIndex$ right over any element which is less than the pivot. Move $rightIndex$ left over any element which is equal to or greater than the pivot. Note that selection of the pivot by $findPivot$ guarantees that there is at least one element less than the pivot and at least one not less than the pivot,so $leftIndex$ and $rightIndex$ will surely come to rest before moving outside the range i to j.

2. *Test.* If $leftIndex > rightIndex$ (which in practice means $leftIndex = rightIndex + 1$), then we have successfully partitioned $l[i], \ldots, l[j]$, and we are done.

3. *Switch.* If $leftIndex < rightIndex$ (note we cannot come to rest during the scan with $leftIndex = rightIndex$, because one or the other of these will move past any given element), then swap $l[leftIndex]$ with $l[rightIndex]$. After doing so, $l[leftIndex]$ has an element less than the pivot and $l[rightIndex]$ has an element at least equal to the pivot, so we know that in the next scan phase, $leftIndex$ will move at least one position right, over the old $l[rightIndex]$, and $rightIndex$ will move at least one position left.

2.6 Heapsort

The heapsort algorithm presented in Figure 8 uses the definition of partially ordered tree. A partially ordered tree can be represented by a *heap*, an array $l[1], \ldots, l[n]$, whose elements have the partially ordered tree property: $l[i] \leq l[2i]$ and $l[i] \leq l[2i+1]$. If we think of the elements at $2i$ and $2i+1$ as the "children" of the element at i, then the array forms a balanced binary tree in which the value of the parent never exceeds the values of the children. By the partially ordered tree property, the smallest element is always in $l[1]$. At the iteration i, the first element of the array $l[1], \ldots, l[i]$ (i.e., the smallest element) is swapped with $l[i]$. Since the new $l[1]$ (old

```java
private int findPivot(int i, int j, int k) {
   if (k <= j) {
      int firstValue = arrayOfInts[i];
      if (arrayOfInts[k] > firstValue)
         return k;
      else
         if (arrayOfInts[k] < firstValue)
            return i;
         else
            return findPivot(i,j,k+1);
   }
   else
      return -1;
}

private int toRight(int leftIndex, int pivot) {
   if (arrayOfInts[leftIndex] < pivot)
      return toRight(leftIndex+1,pivot);
   else
      return leftIndex;
}

private int toLeft(int rightIndex, int pivot) {
   if (arrayOfInts[rightIndex] >= pivot)
      return toLeft(rightIndex-1,pivot);
   else
      return rightIndex;
}

private int partition(int leftIndex, int rightIndex, int pivot) {
   int temp = arrayOfInts[leftIndex];
   arrayOfInts[leftIndex] = arrayOfInts[rightIndex];
   arrayOfInts[rightIndex] = temp;
   leftIndex = toRight(leftIndex,pivot);
   rightIndex = toLeft(rightIndex,pivot);
   if (leftIndex <= rightIndex)
      return partition(leftIndex,rightIndex,pivot);
   else
      return leftIndex;
}
```

it follows

```
private void quick(int i, int j) {
   int pivotIndex = findPivot(i,j,i+1);
   if (pivotIndex != -1) {
      int pivot = arrayOfInts[pivotIndex];
      int k = partition(i,j,pivot);
      quick(i,k-1);
      quick(k,j);
   }
}

void quickSort() {
   quick(0,length-1);
}
```

Figure 7: The Quicksort Method

$l[i]$) violates the partially ordered tree property, we must push it down the tree. Here, we use the method *pushDown* which, by a sequence of swaps, pushes element $l[1]$ down through its descendants to its proper position in the tree.

3 Matching of Lists of Terms

The example program of Figure 9 performs the matching of two list of atoms. An atom can be either a constant or a variable. Variables can be matched either to single atoms or to patterns (i.e., lists of atoms). Constants can only be positive integers. Variables are positive integers (identifiers of variables) preceded by the constant -1 which indicates that the integer following it, is the identifier of a variable and not a constant (e.g., the list $(1\ X)$ is encoded in $(1\ -1\ 1)$).

The output is a list of pairs separated by -2. The first member of a pair is always a variable and we omit the reserved constant -1 for this member. The second member of a pair is the matching of the first member and can be either an atom or a pattern. If the two input lists do not match, $unify$ returns a list only containing -2. Note that -1 and -2 are reserved constants.

Suppose that the identifiers of the variables X, Y, Z, and W are the integers 1, 2, 3, and 4, respectively. Then, on inputs $p_1 = (2\ -1\ 1\ -1\ 3)$ (i.e., $p_1 = (2\ X\ Y)$) and $p_2 = (2\ -1\ 2\ 2\ -1\ 4)$ (i.e., $p_2 = (2\ Y\ 2\ W)$), the output of $unify$ is $(1\ -1\ 2$ $-2\ 3\ 2\ -1\ 4)$ (i.e., $[(X\ Y)\ (Z\ (2\ W))]$).

The method $unify$ recursively compares the two patterns, p_1 and p_2, updating the environment *env* when a variable is encountered in either pattern. *doWhile* is the recursive form of the "repeat" loop and tries to match p_1 with $head \cup tail$, at

```
private void pushDown(int first, int last) {
   if (2*first+1 <= last)
      if (last == 2*first+1) {
         // first has one child at 2*first+1
         if (arrayOfInts[first] > arrayOfInts[2*first+1]) {
            int temp = arrayOfInts[first];
            arrayOfInts[first] = arrayOfInts[2*first+1];
            arrayOfInts[2*first+1] = temp;
         }
      }
      else
         // first has two children, elements at 2*first+1 and 2*first+2
         if ( (arrayOfInts[first] > arrayOfInts[2*first+1])
              && (arrayOfInts[2*first+1] <= arrayOfInts[2*first+2]) ) {
            // swap first with left child
            int temp = arrayOfInts[first];
            arrayOfInts[first] = arrayOfInts[2*first+1];
            arrayOfInts[2*first+1] = temp;
            pushDown(2*first+1,last);
         }
         else
            if ( (arrayOfInts[first] > arrayOfInts[2*first+2])
                 && (arrayOfInts[2*first+2] < arrayOfInts[2*first+1]) ) {
               // swap first with right child
               int temp = arrayOfInts[first];
               arrayOfInts[first] = arrayOfInts[2*first+2];
               arrayOfInts[2*first+2] = temp;
               pushDown(2*first+2,last);
            }
}

private void heap1(int i) {
   if (i >= 0) {
      pushDown(i,length-1);
      heap1(i-1);
   }
}
```

it follows

```
private void heap2(int i) {
   if (i > 0) {
      // remove minimum from front of heap
      int temp = arrayOfInts[0];
      arrayOfInts[0] = arrayOfInts[i];
      arrayOfInts[i] = temp;
      // re-establish partially ordered tree property
      pushDown(0,i-1);
      heap2(i-1);
   }
}

void heapSort() {
   // establish the partially ordered tree property initially
   heap1(length/2-1);
   heap2(length - 1);
}
```

Figure 8: The Heapsort Method

each iteration moving the first atom from *tail* to the end of *head*, such that *head* is the binding of the first variable of p_1 and the rest of p_1 matchs with *tail*, until either a solution was found or all possibilities were tried. $isVar$ returns the value *true* when its argument is a variable. $isConst$ returns the value *true* when its argument is an atom (i.e., not a list or variable). *theSame* compares two constants. $getFirst$ returns the first atom of a list (i.e., it returns the first integer of the list if the first atom is a constant, otherwise it returns the first two integers and the first one is -1). *getRest* returns a copy of the list input without the first atom. *var2var* compares two variables and, if necessary, updates the environment after binding one to the other. *var2pat* matches a variable with a pattern, making sure that the pattern is compatible with the current environment. *lookup* returns the binding of a given variable within an environment. *getBinding* returns the first binding of the environment (it has the same task as $getFirst$ has for lists, but for environments). *getRestEnv* returns a copy of the environment, but it skips the first binding (it has the same task as *getRest* has for lists, but for environments). $extendVar$ and *extendPattern* add a new binding to an environment (in the first case the binding is a variable and, in the second case the binding is a pattern). $isFreeOf$ verifies that a pattern does not contain any instance or binding of a particular variable.

```
list unify(list p1, list p2, list env) {
   list temp = new list();
   if ( (env.first() == -2) || ( (p1.isEmpty()) && (!p2.isEmpty()) ) ||
        ( (!p1.isEmpty()) && (p2.isEmpty()) ) )
      temp.addElement(-2);
   else
      if (isVar(p1))
         if (isVar(p2))
            temp = var2var(p1.rest().first(), p2.rest().first(), env);
         else
            temp = var2pat(p1.rest().first(),p2,env);
      else
         if (isVar(p2))
            temp = var2pat(p2.rest().first(),p1,env);
         else
            if (isConst(p1))
               if (isConst(p2))
                  if (theSame(p1,p2))
                     temp = env.listClone();
                  else
                     temp.addElement(-2);
               else
                  temp.addElement(-2);
            else
               if (isConst(p2))
                  temp.addElement(-2);
               else {
                  list head = new list();
                  list tail = p2.listClone();
                  temp = doWhile(env,p1,head,tail,temp);
                  if ( (temp.first() == -2) && (isVar(getFirst(p2))) ) {
                     head = new list();
                     tail = getRest(p1);
                     head.addElement(p1.first());
                     if (isVar(getFirst(p1)))
                        head.addElement(p1.rest().first());
                     temp = doWhile(env,p2,head,tail,temp);
                  }
               }
   return temp;
}
```

it follows

```
list doWhile(list env, list p1, list head, list tail, list temp) {
   head.addElement(tail.first());
   if (isVar(getFirst(tail)))
      head.addElement(tail.rest().first());
   tail = getRest(tail);
   temp = unify(getRest(p1),tail,unify(getFirst(p1),head,env));
   if ((temp.first() == -2) && !getFirst(tail).isEmpty()
       && !getRest(tail).isEmpty())
      return doWhile(env,p1,head,tail,temp);
   else
      return temp;
}

boolean isVar(list p) {
   if ( (p.first() == -1) && isConst(p.rest()) )
      return true;
   else
      return false;
}

boolean isConst(list p) {
   if ( !p.isEmpty() && p.rest().isEmpty() && (p.first() != -1) )
      return true;
   else
      return false;
}

boolean theSame(list p1, list p2) {
   if (p1.first() == p2.first())
      return true;
   else
      return false;
}

list getFirst(list p) {
   list temp = new list();
   if (p.first() == -1) {
      temp.addElement(-1);
      temp.addElement(p.rest().first());
   }
   else
      temp.addElement(p.first());
   return temp;
}
```

it follows

```
list getRest(list p) {
   if (isConst(getFirst(p)))
      return p.rest();
   else
      return p.rest().rest();
}

list var2var(int v1, int v2, list env) {
   list temp;
   if (v1 == v2)
      temp = env.listClone();
   else
      if (lookup(v1,env).isEmpty())
         if (lookup(v2,env).isEmpty())
            temp = extendVar(v1,v2,env);
         else {
            list t = new list();
            t.addElement(-1);
            t.addElement(v1);
            temp = unify(t,lookup(v2,env),env);
         }
      else
         if (lookup(v2,env).isEmpty()) {
            list t = new list();
            t.addElement(-1);
            t.addElement(v2);
            temp = unify(lookup(v1,env),t,env);
         }
         else
            temp = unify(lookup(v1,env),lookup(v2,env),env);
   return temp;
}

list var2pat(int var, list val, list env) {
   list temp = new list();
   if (lookup(var,env).isEmpty())
      if (isFreeOf(var,val,env))
         temp = extendPattern(var,val,env);
      else
         temp.addElement(-2);
   else
      temp = unify(lookup(var,env),val,env);
   return temp;
}
```

it follows

```
list lookup(int var, list env) {
   list temp = new list();
   if (env.isEmpty())
      ;
   else
      if (var == env.first())
         temp = getBinding(env.rest(), new list());
      else
         temp = lookup(var, getRestEnv(env));
   return temp;
}

list getBinding(list env, list result) {
   list temp;
   if (env.first() == -2)
      temp = result.listClone();
   else {
      result.addElement(env.first());
      temp = getBinding(env.rest(), result);
   }
   return temp;
}

list getRestEnv(list env) {
   list temp;
   if (env.first() == -2)
      temp = env.rest().listClone();
   else
      temp = getRestEnv(env.rest());
   return temp;
}

list extendVar(int v1, int v2, list env) {
   list temp = env.listClone();
   temp.addElement(v1);
   temp.addElement(-1);
   temp.addElement(v2);
   temp.addElement(-2);
   return temp;
}
```

it follows

```
boolean isFreeOf(int var, list val, list env) {
   if (isConst(val))
      return true;
   else
      if (isVar(val))
         if (var == val.rest().first())
            return false;
         else
            if (lookup(val.rest().first(),env).isEmpty())
               return true;
            else
               return isFreeOf(var,lookup(val.rest().first(),env),env);
      else
         if (isFreeOf(var,getFirst(val),env))
            return isFreeOf(var,getRest(val),env);
         else
            return false;
}

list extendPattern(int var, list val, list env) {
   list temp = env.listClone();
   temp.addElement(var);
   list t = expand(val,temp);
   t.addElement(-2);
   return t;
}

list expand(list val, list env) {
   list temp;
   if (val.isEmpty())
      temp = env.listClone();
   else {
      list t = env.listClone();
      t.addElement(val.first());
      temp = expand(val.rest(),t);
   }
   return temp;
}
```

Figure 9: The Matching Methods