# CAB301 Assignment 2
# Empirical Comparison of Four Algorithms for Testing Set Membership

Student Names
(Student Numbers)

Date submitted: 9th September 2007

## Summary

This report describes the outcomes of several experiments conducted to directly compare the efficiency of four different 'set membership' algorithms. The original intention was to compare three well-known algorithms which represent sets of numbers as unordered linked lists, ordered linked lists and binary search trees, respectively. However, while performing the experiments it was noticed that the ordered linked list algorithm could be optimised, so an 'efficient' variant of this algorithm was also included.

For each of the four algorithms the basic operations they perform were counted and their total execution times were measured. Care was taken to test the algorithms using identical data, so that the results could be compared meaningfully. The observed numbers of basic operations were consistent with the theoretical predictions for the four algorithms. It was also found that the optimisation of the ordered linked list algorithm made a small, but noticeable, improvement to the corresponding program's execution time.

## 1 Description of the Algorithms

In their introductory textbook on computer programming, Abelson and Sussman note that the choice of data structures has a significant impact on program efficiency. As an illustrative example they briefly describe how different representations of mathematical sets can affect the efficiency of programs that implement set operators [1, §2.3.3]. Their examples include sets of numbers represented as unordered linked lists, ordered linked lists and binary search trees.

There are several operations relevant to sets, including set insertion, set union, set intersection, etc. In this experiment we studied the relative efficiency of algorithms for testing set membership. Given a set of items and a particular value, which acts as a search key, the algorithm is required to return 'true' if the key is a member of the set, or 'false' otherwise. The efficiency of a set membership algorithm depends on the data structure used to represent sets and whether or not the key is in the set. (For reasons of storage efficiency we chose to study iterative versions of the algorithms, although Abelson and Sussman present recursive versions.)

The first algorithm (Figure 1) is for the situation where sets are represented as unordered linked lists. In this case the strategy is to simply begin with the first node (statement b), and compare each item in the list in turn with the search key until either the key is found in the list (condition d), in which case the set membership algorithm returns 'true' (statement e), or the end of the list is reached (condition c), in which case the algorithm returns 'false' (statement g).

```
a.  ALGORITHM ElementOfUnorderedList(searchNum)
      // Returns true if and only if item searchNum is in the set pointed to by
      // global variable firstNode, where sets are represented as unordered
      // linked lists
b.    currNode ← firstNode
c.    while currNode ≠ Null do
d.      if currNode↑Contents = searchNum
e.        return true
f.      currNode ← currNode↑Next
g.    return false
```

Figure 1: A set membership algorithm for unordered linked lists. Let 'Null' denote the null pointer; assume that linked list nodes have fields 'Contents' and 'Next'; and let '$p\uparrow f$' denote field $f$ of the node pointed to by pointer $p$.

```
h.  ALGORITHM ElementOfOrderedList(searchNum)
      // Returns true if and only if item searchNum is in the set pointed to by
      // global variable firstNode, where sets are represented as ordered
      // linked lists
i.    currNode ← firstNode
j.    while currNode ≠ Null do
k.      if currNode↑Contents = searchNum
l.        return true
        else
m.        if searchNum < currNode↑Contents
n.          return false
          else
o.          currNode ← currNode↑Next
p.    return false
```

Figure 2: Abelson and Sussman's 'inefficient' set membership algorithm for ordered linked lists [1, p. 154].

The second algorithm (Figure 2) is for use when sets are represented as linked lists which have been ordered so that smaller values appear in the list before larger ones. Again the strategy is to follow the list starting from the first node (statement i) and check each node's contents against the search key (condition k). However, since the list is ordered, the search can stop as soon as a number is found that is larger than the key (condition m), because we then know that the key is not in the list even without checking the remaining items. The version of this algorithm shown in Figure 2 is a direct translation of the one presented by Abelson and Sussman [1, p. 154].

While conducting the experiments, we noticed that Abelson and Sussman's set membership algorithm for ordered linked lists compares each number in the list smaller than the search key with the search key twice. Both an equality and a 'less than' test are performed in the **if** statements (conditions k and m, respectively). We realised that a more efficient algorithm would be to search through the list until an item is found that is *not* smaller than the search key and only then perform the equality test. We therefore wrote an 'efficient' version of the algorithm using this strategy (Figure 3), and decided to compare it with Abelson and Sussman's version, as well as comparing their three algorithms with one another. In the efficient version the main loop terminates either when the end of the list is reached or an item not less than the search key is found (condition s). The equality test is then performed only if the end of the list has not been encountered (statement u). Note that it is assumed in Figure 3 that conjuncts in Boolean expressions are evaluated lazily from

```
q.  ALGORITHM ElementOfOrderedList2(searchNum)
      // Returns true if and only if item searchNum is in the set pointed to by
      // global variable firstNode, where sets are represented as ordered
      // linked lists
r.    $currNode \leftarrow firstNode$
s.    while $currNode \neq$ Null and $currNode\uparrow$Contents $< searchNum$ do
t.      $currNode \leftarrow currNode\uparrow$Next
u.    return $currNode \neq$ Null and $currNode\uparrow$Contents $= searchNum$
```

Figure 3: An 'efficient' set membership algorithm for ordered linked lists.

```
v.  ALGORITHM ElementOfSearchTree(searchNum)
      // Returns true if and only if item searchNum is in the set pointed to by
      // global variable rootNode, where sets are represented as binary
      // search trees
w.    $currNode \leftarrow rootNode$
x.    while $currNode \neq$ Null do
y.      if $currNode\uparrow$Contents $= searchNum$
z.        return true
        else
α.        if $searchNum < currNode\uparrow$Contents
β.          $currNode \leftarrow currNode\uparrow$Left
          else
γ.          $currNode \leftarrow currNode\uparrow$Right
δ.    return false
```

Figure 4: Set membership algorithm for binary search trees [2, §4.5.2]. Assume that search tree nodes have fields 'Contents', 'Left' and 'Right'.

left to right.

The fourth algorithm (Figure 4) is for use with sets represented as binary search trees. It follows the usual strategy for searching an ordered binary tree [2, §4.5.2]. Starting from the root node (statement w) it returns 'false' (statement $\delta$) if the current node is null. Otherwise it checks the current node's contents to see if it equals the search key (condition y) and, if so, returns 'true' (statement z). If this is not the case the algorithm moves on to the left-hand (statement $\beta$) or right-hand (statement $\gamma$) branch of the current node, depending on whether the search key is less than (condition $\alpha$) or greater than the current node's contents, respectively.

## 2 Theoretical Analysis of the Algorithms

### 2.1 Choice of Basic Operation

Since a set membership algorithm is a form of searching algorithm, we adopted the usual convention of treating comparison of the search key with an item in the collection being searched as the basic operation [4, p. 47]. (Abelson and Sussman treat each iteration/recursion of their algorithms as a single basic operation [1]. This is equivalent to Levitin's simplifying notion of combining consecutive comparisons into a single 'three-way' comparison [4, p. 136].)

## 2.2 Choice of Problem Size

The obvious choice of 'problem size' for this application is the number of items in the given set. (However, the number of items in a set is not necessarily the same as the number of items inserted into it. Since sets may not contain duplicate items, the number of insertions performed can be greater than the total number of items in the resulting set. Allowance was made for this possibility when conducting the experiments, as explained in Appendix J and Appendix K.)

## 2.3 Average-Case Efficiency

Each of the algorithms can 'exit early' when the item being searched for is found, so the best-, worst- and average-case efficiency of each algorithm may be different. In this report we are interested in their average-case efficiencies only. Furthermore, the behaviour of searching algorithms can be different for successful and unsuccessful searches, so we needed to distinguish set membership operations where the search key is, and is not, in the given set.

The algorithm for unordered linked lists (Figure 1) is clearly just a version of a traditional sequential or linear search algorithm [4, p. 47][2, §2.6.1]. The basic operation is the equality test in the **if** statement. In the case of an unsuccessful search all $n$ items in the set (list) must be examined. The number of comparisons, using Levitin's notation, is thus

$$\begin{aligned} C^{no}_{avg}(n) &= n \\ &\in \Theta(n)\,. \end{aligned}$$

For a successful search, previous analyses of linear searches [4, p. 49][2, p. 35] tell as that the average-case number of comparisons is

$$\begin{aligned} C^{yes}_{avg}(n) &= \frac{n+1}{2} \\ &\in \Theta(n)\,, \end{aligned}$$

which is consistent with our intuition that the item of interest will be found about half-way along the list on average.

Abelson and Sussman's note that their 'inefficient' ordered list algorithm (Figure 2), has the advantage that the search can stop as soon as the position in the ordered list where the search key is expected to be found is reached [1]. Assuming a uniform distribution of search keys and items in the set, this implies that the algorithm requires around $n/2$ iterations only for both successful and unsuccessful searches. Thus the algorithm should be superior to the one using unordered lists in the case of an unsuccessful search, since it can stop earlier on average.

However, if we count the number of individual comparisons, rather than iterations, the results do not look as good. As shown in Figure 2, the algorithm performs two comparisons for each iteration as long as the search key is greater than the list item, one for the equality in the outermost **if** statement and one for the inequality in the innermost **if** statement. Thus the number of individual comparisons for an unsuccessful search is two comparisons for each of the items smaller than the search key, which will be around half of the items in the set on average, plus two more comparisons for the next item, which is bigger than the search key, i.e.,

$$\begin{aligned} C^{no}_{avg}(n) &\approx 2 * \frac{n}{2} + 2 \\ &= n + 2 \\ &\in \Theta(n)\,. \end{aligned}$$

In the case of a successful search the number of comparisons will be two for each item smaller than the search key, which will be around half of the items in the set on average, plus one more equality test to recognise that the item of interest has been found, i.e.,

$$\begin{aligned} C_{avg}^{yes}(n) &\approx 2 * \frac{n}{2} + 1 \\ &= n + 1 \\ &\in \Theta(n)\,. \end{aligned}$$

The order of growth is still linear, but the number of individual comparisons is higher than necessary due to the many 'false' equality tests performed. Notice that the number of comparisons is *worse* than for an unordered list in the case of a successful search!

Based on this observation, our 'efficient' version of the set membership algorithm for ordered lists (Figure 3) is designed to perform an equality comparison only when all values smaller than the search key have been passed. In the case where the item of interest is not in the set, the inequality in the **while** loop will be performed for each number in the set smaller than the search key (when the inequality returns 'true'), which will be about half of the items on average. The inequality will be performed once more (returning 'false') on the next item to exit the loop. Then the equality test in the **return** statement will be performed (returning 'false') on the last item examined. Thus the number of comparisons performed for an unsuccessful set membership operation should be

$$\begin{aligned} C_{avg}^{no}(n) &\approx \frac{n}{2} + 1 + 1 \\ &= \frac{n}{2} + 2 \\ &\in \Theta(n)\,. \end{aligned}$$

In the case where the item of interest *is* in the set, the inequality in the **while** loop will be performed for each number in the set smaller than the search key (returning 'true' each time), which will be about half of the items on average. The test will be performed once more (returning 'false') on the next item to exit the loop. Then the equality test in the **return** statement will be performed (returning 'true') on the last item examined. Thus the number of comparisons performed for an successful set membership operation should be around the same as for the unsuccessful case:

$$\begin{aligned} C_{avg}^{yes}(n) &\approx \frac{n}{2} + 1 + 1 \\ &= \frac{n}{2} + 2 \\ &\in \Theta(n)\,. \end{aligned}$$

Thus our 'efficient' set membership algorithm for ordered linked lists should require around half as many basic operations as Abelson and Sussman's version, for both successful and unsuccessful cases. Both versions of the algorithm exhibit a linear order of growth, however.

The algorithm for the binary search tree representation of sets (Figure 4) is a classic example of a divide-and-conquer algorithm which halves the problem size at each step. Berman and Paul note that devising a precise average-case formula for a binary tree search is difficult [2, §6.7] since it relies on the probability distribution of values in the tree and the tree's 'shape', which is determined by the order in which items were inserted into the tree. Nevertheless, it is widely understood that for a 'balanced' binary tree, i.e., one in which each node's left-hand and right-hand branches are of approximately the same size, the time required to find a particular node is proportional to the depth of the tree, which is itself logarithmic in the number of nodes. Therefore, we simply assume that

$$\begin{aligned} C_{avg}^{no}(n) &\approx \log_2 n \\ &\in \Theta(\log n) \end{aligned}$$

and

$$\begin{aligned} C^{yes}_{avg}(n) &\approx \log_2 n \\ &\in \Theta(\log n)\,. \end{aligned}$$

(As the experiments below reveal, the binary search tree representation is so efficient compared to the list representations of sets that its exact efficiency equation is irrelevant!)

## 3 Methodology, Tools and Techniques

### 3.1 Programming Environment

1. To implement the algorithms and the experiments we decided to use the Python programming language [5, 3] having used it before and found it elegant and easy to use.

2. The experiments were performed on an Apple Macintosh PowerBook G4 laptop computer, running the UNIX-based Mac OS X operating system. Python's pseudo-random number generator [5, p. 301] was used to produce test data and Python's time module [5, p. 246] was used for measuring execution times.

3. To produce graphs of the experimental results, we used Apple's Grapher utility. The programs were designed so that they wrote their results to text files in a format that could be imported directly into Grapher. Figures 5 to 8 in this report were all produced in Grapher, and exported in Portable Document Format. This report was prepared using the LaTeX typesetting system.

### 3.2 Implementation of the Algorithms

The set membership algorithms rely on the existence of a data structure representing a set, either an unordered list, ordered list, or binary search tree. Therefore, we first wrote appropriate classes to support the creation of unordered lists of numbers (Appendix A), ordered lists (Appendix B) and binary search trees (Appendix C).

The algorithms themselves were then implemented as methods within these classes. The four algorithms in Figures 1 to 4 were implemented in Python as shown in Appendix D to Appendix G, respectively. In each case the translation from pseudocode into Python was straightforward.

### 3.3 Generating Test Data and Running the Experiments

To test the correctness of the implementations of the algorithms we wrote a small test program for each which populated a set with some given numbers and then did a set membership operation for a number known to be in the set and one known not to be in the set (Appendix H).

To count the number of basic operations performed by each of the algorithms, we firstly inserted code into the corresponding programs to increment a counter each time one of the basic operations is performed (Appendix I). We then wrote two programs which created sets of different sizes and performed set membership operations, keeping track of the average number of basic operations performed (Appendix J). Two nearly-identical programs were developed, one that guaranteed the search key was in the set and one that guaranteed it wasn't. Importantly, the programs populated all four sets with exactly the same data values for each trial, to ensure that the comparisons between the different algorithms were meaningful.

To measure the execution times for each of the algorithms, two similar programs were developed (Appendix K), except that these programs used the original versions of the programs and measured the total execution time to perform the set membership operations. Since the algorithms executed very quickly, each test was repeated many times to produce a total execution time, which was then divided by the number of repetitions. (As the graphs below show, this seemed to produce accurate results.) Again, care was taken to apply all four algorithms to sets populated with the same data values, to produce meaningful comparisons.

## 4 Experimental Results

### 4.1 Functional Testing

To test the correctness of each of the implementations of the four algorithms the approach described in Appendix H was used. For instance, a test for the binary search tree implementation, inserting the numbers 3, 3, 0, 5, 5, 3, 3, 1, 2, 3, 0, 4, 5, 3, 2, 8, 9, 2, 7, 6, 1, 5 into the set, in that order, produced the following output:

```
The set is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
The set contains 10 items
Number 5 is in the set
Number 11 is not in the set
```

This confirms that the tree is correctly sorted, that duplicate insertions are ignored, and that the set membership method produces the correct results.

Similarly, a test in which the following numbers 5, 5, 4, 3, 2, 1, 1, 1 were inserted into the 'efficient' linked list implementation produced the following output:

```
The set is {1, 2, 3, 4, 5}
The set contains 5 items
Number 5 is in the set
Number 11 is not in the set
```

This confirmed that the algorithm works even for the extreme case of input data that is in reverse order.

In another test the numbers 2, 3, 4, 4, 4, 5, 6, 7, 8, 9 were inserted into the unordered linked list implementation, producing the following output:

```
The set is {2, 3, 4, 5, 6, 7, 8, 9}
The set contains 8 items
Number 5 is in the set
Number 11 is not in the set
```

This confirmed that the algorithm works even for the extreme case of input data that is already in order. These, and other tests, demonstrated that each of the algorithms worked correctly.

### 4.2 Average-Case Number of Basic Operations for an Item in the Set

Figure 5 shows the results of an experiment conducted to count the number of basic operations performed by each of the four algorithms, in the situation where the search key was guaranteed to be in the set. See Appendix J for the program that produced this result.

Each of the data points in the graph is an average of 100 trials using different data values in the set and different search keys, all randomly generated. It is immediately obvious that the binary search tree has the predicted logarithmic efficiency, while the three list-based algorithms produced linear growth, as expected.

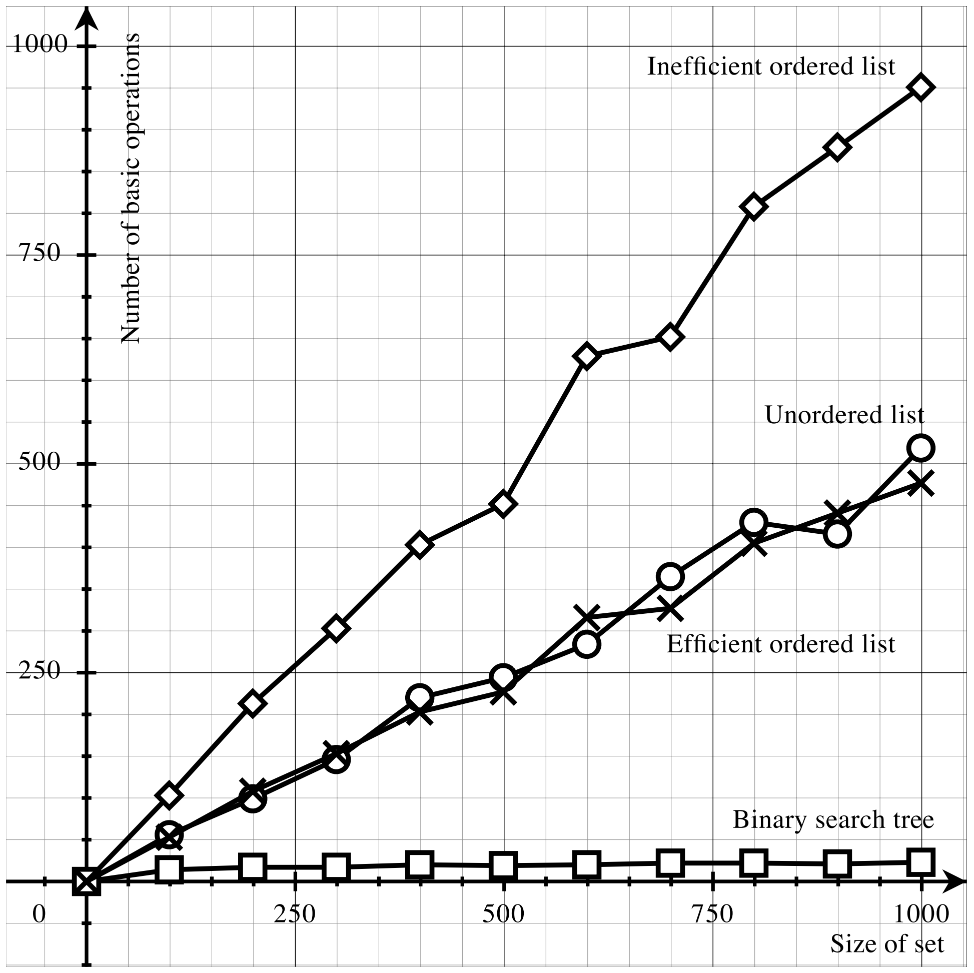

Figure 5: Measured number of basic operations required to successfully find an item in a given set for each of the four algorithms. Each of the data points shown is the average of 100 tests using distinct, randomly-generated set contents and search keys.

The search key could be expected to occur around half-way along the list on average, which explains why both the unordered list and 'efficient' ordered list algorithms exhibit a clear $n/2$ growth. Abelson and Sussman's 'inefficient' ordered list algorithm produces an order $n$ growth due to the fact that it does two basic operations at each iteration. All of these outcomes are entirely consistent with the theoretical predictions.

## 4.3 Average-Case Number of Basic Operations for an Item Not in the Set

Figure 6 shows the results of an experiment conducted to count the number of basic operations performed by each of the four algorithms, in the situation where the search key is guaranteed *not* to be in the set (Appendix J).

Again, each of the data points in the graph is an average of 100 trials using different data values in the set, all randomly generated. As in the previous experiment, it is obvious straight away that the binary search tree has logarithmic efficiency, while the three list-based algorithms are linear.

Since the search key is not in the list, the unordered list algorithm searches all the way to the end of the list each time and produces a perfectly straight line. By contrast, the

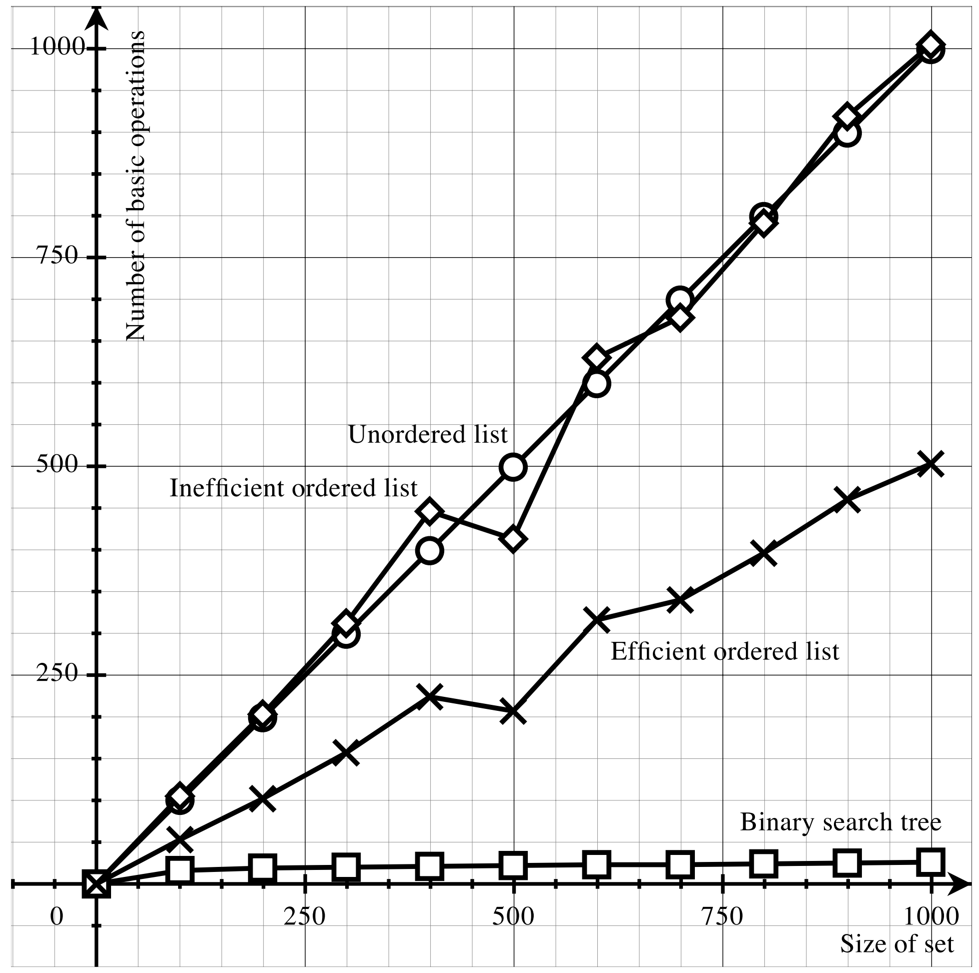

Figure 6: Measured number of basic operations required to discover that an item is not in a given set for each of the four algorithms. Each of the data points shown is the average of 100 tests using distinct, randomly-generated set contents and a search key that is guaranteed not to be in the set.

efficient ordered list algorithm stops searching half-way through the list on average, when it recognises that it has passed the point where the search key 'should' be. Abelson and Sussman's 'inefficient' ordered list algorithm also shares this advantage, but still performs twice as many basic operations as necessary due to the many (false) equality tests. Once again, these outcomes are consistent with the theoretical predictions.

## 4.4 Average-Case Execution Time for an Item in the Set

The experiments above confirmed that the number of basic operations performed by each of the four algorithms was as expected. In particular, they showed that our 'efficient' ordered list algorithm performed half as many basic operations as Abelson and Sussman's 'inefficient' one.

Next we performed an experiment to measure the execution time of the four algorithms, when the search key is in the set. Figure 7 shows the outcome. See Appendix K for the program that produced this result.

Each of the data points in Figure 7 is the result of 10,000 trials, comprising 100 distinct tests using different, randomly-produced data values, with each set-membership operation

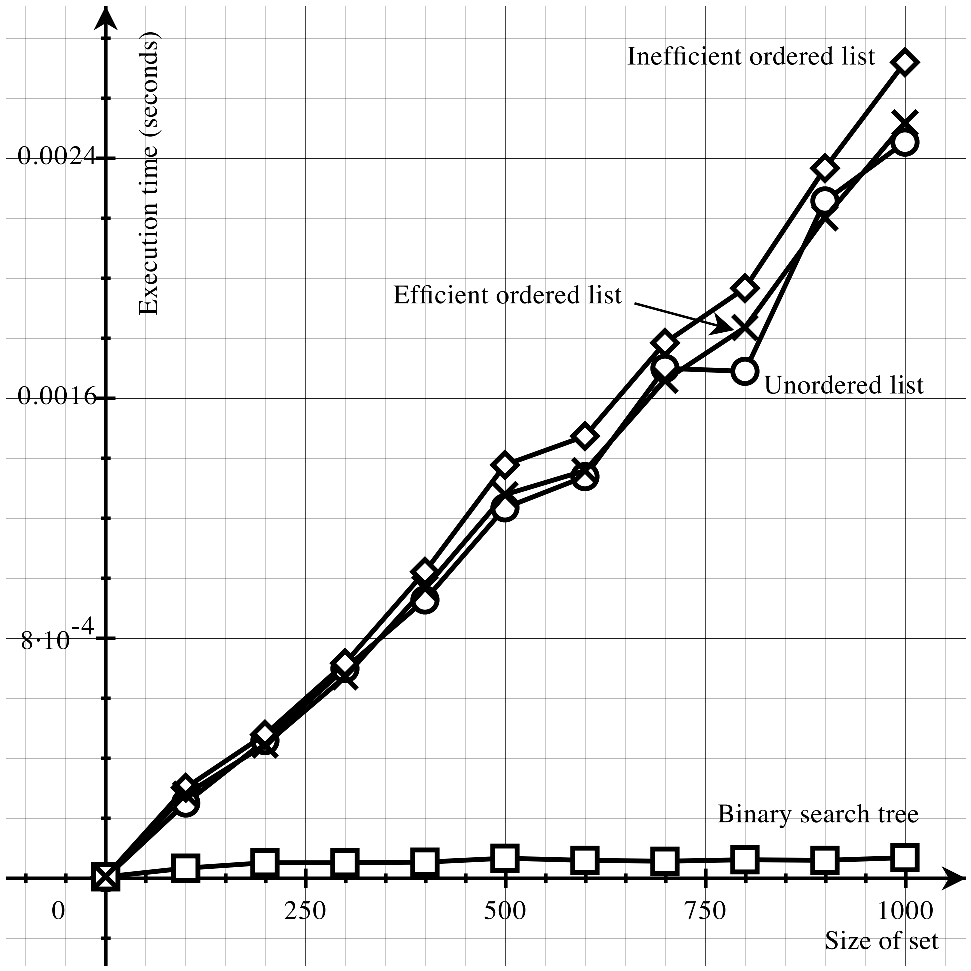

Figure 7: Measured execution time required to successfully find an item in a given set for each of the four algorithms. Each of the data points shown is the average of 100 tests using distinct, randomly-generated set contents and search keys, with the same test repeated 100 times to produce measurable execution times.

repeated 100 times to produce a measurable execution time. (In practice, the execution time of the code to populate the sets with test data far outweighed the execution times of the set-membership methods of interest.)

The execution times in Figure 7 confirm the dramatic improvement in efficiency afforded by a binary tree implementation, and the linear growth of the list-based programs. However, compared to Figure 5, the graph in Figure 7 reveals that the execution time of Abelson and Sussman's 'inefficient' algorithm is little different from our 'efficient' version. Although our optimised program is consistently faster than Abelson and Sussman's, the difference is only slight. The redundant equality tests in Abelson and Sussman's algorithm seem to have little impact on the overall execution time of the corresponding program. Indeed, in the case of a successful search, when the search key is found half-way along the list on average, all three list-based implementations performed similarly.

## 4.5 Average-Case Execution Time for an Item Not in the Set

Finally, Figure 8 shows the execution times for an experiment conducted where the set membership operation must produce 'false' because the search key is not in the set (Ap-

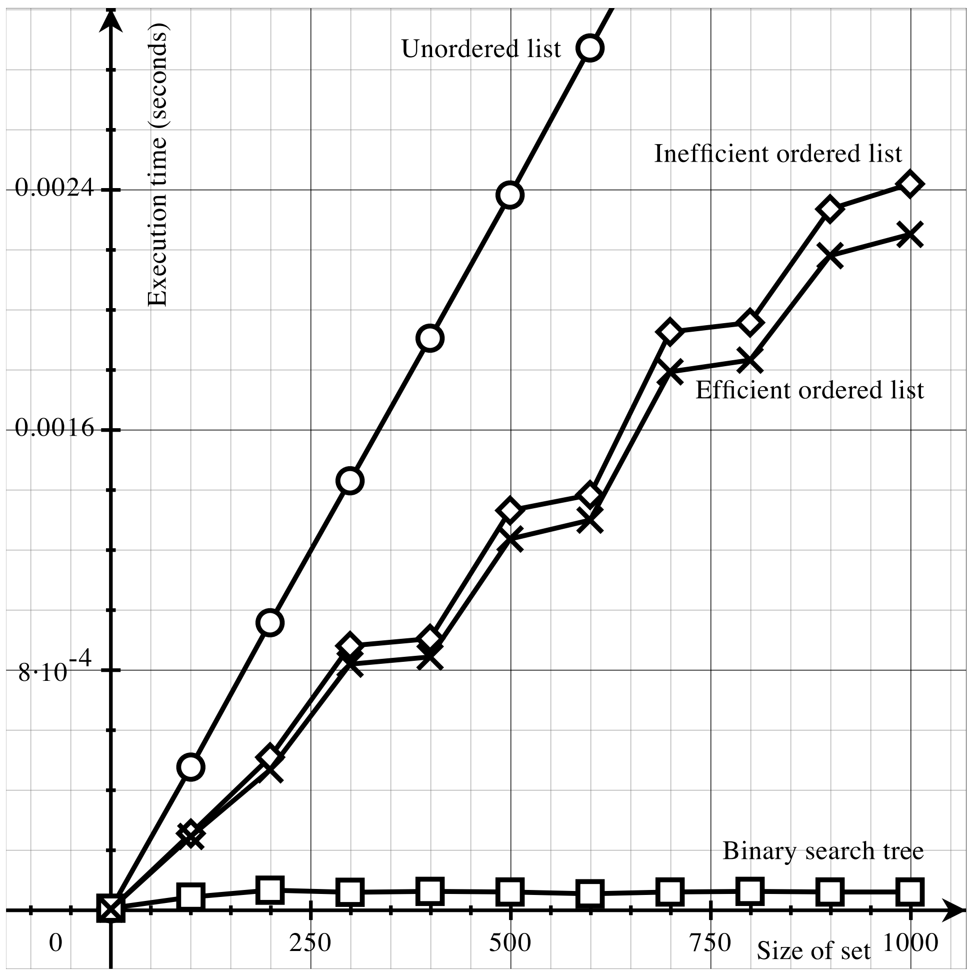

Figure 8: Measured execution time required to discover that an item is not in a given set for each of the four algorithms. Each of the data points shown is the average of 100 tests using distinct, randomly-generated set contents, and a search key that is guaranteed not to be in the set, with each test repeated 100 times to produce measurable execution times.

pendix K). Again, each of the data points in the figure is the result of 10,000 trials.

The execution time graph in Figure 8 once more shows the advantage of a tree representation. The unordered list program is always forced to search to the end of the list so it has the worst execution time of all four algorithms. Both versions of the ordered list program execute in about half the time of the unordered list one, because they can stop searching about half-way through the list on average.

Although the graph shows that our ‘efficient’ program outperforms Abelson and Sussman’s ‘inefficient’ one, the difference is again much less dramatic than the count of basic operations in Figure 6 would suggest. Evidently the time required to evaluate the redundant equalities is trivial compared to the overall execution time of the set-membership methods.

## References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, second edition, 1996. ISBN 0-262-51087-1.

[2] K. A. Berman and J. L. Paul. *Algorithms: Sequential, Parallel and Distributed*. Thomson, 2005. ISBN 0-534-42057-5.

[3] A. Downey, J. Elkner, and C. Meyers. *How to Think Like a Computer Scientist: Learning with Python*. Green Tea Press, 2002. http://www.greenteapress.com/thinkpython/.

[4] A. Levitin. *Introduction to the Design and Analysis of Algorithms*. Addison-Wesley, second edition, 2007. ISBN 0-321-36413-9.

[5] A. Martelli. *Python in a Nutshell*. O'Reilly, 2003. ISBN 0-596-00188-6.

## Appendix A  Code to Support the Unordered Linked List Algorithm

The algorithm in Figure 1 assumes that a set has been implemented as an unordered linked list. This appendix presents the Python code for creating such lists.

The program comprises two classes. The first, shown below, implements linked list nodes as objects which have two attributes, the node's contents (`Contents`) and the next node in the list (`Next`). Python's 'null' value `None` is used as the null pointer. In Python the special method `init` is invoked when an object is created and special method `str` is invoked when the 'string' version of an object is required (e.g., in a `print` statement). In our case the `init` method (lines 3–5) just stores the node's contents and its next 'pointer' as local attributes, and the `str` method (lines 7–8) merely returns the string representation of the node's contents.

```
 1. class ListNode:
 2.
 3.   def __init__(self, initContents=None, initNext=None):
 4.     self.Contents = initContents
 5.     self.Next = initNext
 6.
 7.   def __str__(self):
 8.     return str(self.Contents)
```

The code for the unordered linked list class is shown below. Here a set has two attributes, a pointer to the first node in the linked list holding the set's contents (`firstNode`), and the set's size (`setSize`). (The algorithms in Figures 1, 2 and 3 did not need the set's size, but this attribute proved useful to determine the set's true size, as opposed to the number of insertions performed, to allow for duplicate items.)

When a linked-list set is created it has no contents and is of size zero (lines 3–5). The method for returning the contents of a set as a string (lines 22–31) is based on some similar code for printing linked lists described by Downey *et al* [3, pp. 181, 186]. It was used when testing the algorithm's correctness.

The `SetInsert` method (lines 7–20) inserts new items into the set, taking care not to add duplicate items. If the set is empty the new item is simply inserted (lines 9–11). Otherwise the list is searched until either the end is reached or the item to be inserted is encountered (lines 13–16). If the end of the list was reached without the new item being found then it is added to the end of the list (lines 18–20). Thus items appear in the list in the order they were inserted.

```
 1. class UnorderedList:
 2.
 3.   def __init__(self):
 4.     self.setSize = 0
 5.     self.firstNode = None
 6.
 7.   def SetInsert(self, newNum):
 8.     if self.firstNode == None:
 9.       newNode = ListNode(newNum, self.firstNode)
10.       self.firstNode = newNode
11.       self.setSize = self.setSize + 1
12.     else:
13.       currNode = self.firstNode
14.       while currNode.Next != None and
15.             currNode.Contents != newNum:
        currNode = currNode.Next
      if currNode.Contents != newNum:
        newNode = ListNode(newNum, currNode.Next)
        currNode.Next = newNode
        self.setSize = self.setSize + 1

  def __str__(self):
    def PrintSetItems(currNode):
      stringRep = ""
      while currNode != None:
        stringRep = stringRep + str(currNode.Contents)
        currNode = currNode.Next
        if currNode != None:
          stringRep = stringRep + ", "
      return stringRep
    return "{" + PrintSetItems(self.firstNode) + "}"
```

## Appendix B    Code to Support the Ordered Linked List Algorithms

The algorithms in Figures 2 and 3 assume that the given set is represented as an ordered linked list. This appendix presents the Python code used to construct such lists.

The ordered linked list class, `OrderedList`, used the `ListNode` class, and `init` and `str` methods from Appendix A. However, to insert a new item into a set it used the following method.

```
 1. def SetInsert(self, newNum):
 2.   if self.firstNode == None or
 3.      newNum < self.firstNode.Contents:
 4.     newNode = ListNode(newNum, self.firstNode)
 5.     self.firstNode = newNode
 6.     self.setSize = self.setSize + 1
 7.   else:
 8.     currNode = self.firstNode
 9.     while currNode.Next != None and
10.           currNode.Next.Contents <= newNum:
11.       currNode = currNode.Next
12.     if currNode.Contents != newNum:
13.       newNode = ListNode(newNum, currNode.Next)
14.       currNode.Next = newNode
15.       self.setSize = self.setSize + 1
```

If the list is empty or the new item is smaller than the first item in the list, it is inserted at the beginning of the list (lines 4–6). Otherwise the method searches for the place where the new item ‘should’ be in the ordered list (lines 8–11). If the item is not found it is then inserted in the appropriate spot to maintain the list’s ordering (lines 13–15).

## Appendix C    Code to Support the Binary Search Tree Algorithm

The algorithm in Figure 4 assumes that the given set is represented as a binary search tree. This appendix presents the Python code for creating such trees.

The program comprises two classes. The first, shown below, implements binary tree nodes as objects which have three attributes, the node's contents (`Contents`), and its left-hand and right-hand branches (`Left` and `Right`, respectively). The special `init` method initialises the three attributes for a new binary tree node (lines 3–7) and the `str` method defines the string representation of a binary tree node to be the string representation of its contents (lines 9–10).

```
 1. class TreeNode:
 2.
 3.   def __init__(self, initContents=None, initLeft=None,
 4.                initRight=None):
 5.     self.Contents = initContents
 6.     self.Left = initLeft
 7.     self.Right = initRight
 8.
 9.   def __str__(self):
10.     return str(self.Contents)
```

The second class implements binary search trees, as shown below. The special `init` method creates a tree with two attributes, the root node and the size of the set (lines 3–5). (The algorithm in Figure 4 does not require the size of the set to be maintained, but this feature was useful when analysing the algorithm, since the number of items in the set may be less than the number of insertions, due to duplicate items.) The string representation of a binary search tree is produced using a recursive inorder traversal of the binary tree (lines 31–46). (This recursive procedure was useful for small sets only, because it quickly reached Python's limit on the depth of the run-time stack.)

Most importantly, the method for inserting items into the set (lines 7–29) followed the usual strategy for maintaining a binary search tree [2, §4.5.3]. If the tree is empty the new item is inserted as the root node (lines 9–10). Otherwise a search is performed to find where the new item 'should' be in the tree (lines 12–29). The left branch is followed if the new item is less than the one at the current node (line 22). The right branch is followed if the new item is greater than the one at the current node (line 29). If it is found that the new item is smaller than the one at the current node and there is no left-hand branch then the new item is added as a new left-hand node (lines 18–20). If it is found that the new item is greater than the one at the current node and there is no right-hand branch then the new item is added as a new right-hand node (lines 25–27). If the new item is found to already be in the tree then no insertion is performed.

```
 1. class SearchTree:
 2.
 3.   def __init__(self):
 4.     self.setSize = 0
 5.     self.rootNode = None
 6.
 7.   def SetInsert(self, newNum):
 8.     if self.rootNode == None:
 9.       self.rootNode = TreeNode(newNum)
10.       self.setSize = self.setSize + 1
11.     else:
12.       currNode = self.rootNode
13.       inserted = False
14.       while currNode.Contents != newNum and
15.             not inserted:
16.         if newNum < currNode.Contents:
17.           if currNode.Left == None:
```

```
            currNode.Left = TreeNode(newNum)
            self.setSize = self.setSize + 1
            inserted = True
          else:
            currNode = currNode.Left
        else:
          if currNode.Right == None:
            currNode.Right = TreeNode(newNum)
            self.setSize = self.setSize + 1
            inserted = True
          else:
            currNode = currNode.Right

  def __str__(self):

    def PrintInorder(currNode):
      if currNode == None:
        return ""
      else:
        stringRep = str(currNode.Contents)
        leftBranch = PrintInorder(currNode.Left)
        if leftBranch != "":
          stringRep = leftBranch + ", " + stringRep
        rightBranch = PrintInorder(currNode.Right)
        if rightBranch != "":
          stringRep = stringRep + ", " + rightBranch
        return stringRep

    return "{" + PrintInorder(self.rootNode) + "}"
```

## Appendix D Implementation of the Set Membership Algorithm for Unordered Lists

Our Python implementation of the algorithm from Figure 1 is shown below, in the context of the `UnorderedList` class described in Appendix A. There is a direct correspondence between the algorithm and the code; the letters after the line numbers refer to statements in the algorithm.

```
1(a). def ElementOf(self, searchNum):
2(b).   currNode = self.firstNode
3(c).   while currNode != None:
4(d).     if currNode.Contents == searchNum:
5(e).       return True
6(f).     currNode = currNode.Next
7(g).   return False
```

## Appendix E Implementation of the ‘Inefficient’ Set Membership Algorithm for Ordered Lists

Our Python implementation of the algorithm from Figure 2 is shown below, in the context of the `OrderedList` class described in Appendix B. The correspondence between the algorithm and the code is obvious, except that Python’s ‘`elif`’ construct has been used instead of nested 'two-way' **if** statements. This notational difference has no effect on the algorithm's efficiency.

```
 1(h). def ElementOf(self, searchNum):
 2(i).   currNode = self.firstNode
 3(j).   while currNode != None:
 4(k).     if currNode.Contents == searchNum:
 5(l).       return True
 6(m).     elif searchNum < currNode.Contents:
 7(n).       return False
    8.     else:
 9(o).       currNode = currNode.Next
10(p).   return False
```

## Appendix F Implementation of the 'Efficient' Set Membership Algorithm for Ordered Lists

Our Python implementation of the algorithm from Figure 3 is shown below, in the context of the `OrderedList` class described in Appendix B. There is a direct correspondence between the algorithm and the code; the letters after the line numbers refer to statements in the algorithm.

```
 1(q). def ElementOf(self, searchNum):
 2(r).   currNode = self.firstNode
 3(s).   while currNode != None and
 4(s).         currNode.Contents < searchNum:
 5(t).     currNode = currNode.Next
 6(u).   return currNode != None and
 7(u).          currNode.Contents == searchNum
```

## Appendix G Implementation of the Set Membership Algorithm for Binary Search Trees

Our Python implementation of the algorithm from Figure 4 is shown below, in the context of the `SearchTree` class described in Appendix C. The correspondence between the algorithm and the code is obvious, except that Python's '`elif`' construct has been used instead of nested 'two-way' **if** statements. This notational difference has no effect on the algorithm's efficiency.

```
 1(v). def ElementOf(self, searchNum):
 2(w).   currNode = self.rootNode
 3(x).   while currNode != None:
 4(y).     if currNode.Contents == searchNum:
 5(z).       return True
 6(α).     elif searchNum < currNode.Contents:
 7(β).       currNode = currNode.Left
    8.     else:
 9(γ).       currNode = currNode.Right
10(δ).   return False
```

## Appendix H Code for Functional Testing

To test the correctness of the set membership method we wrote a small program to create a set and then search for an item which was known to be in the set and an item known not to be in the set. The following version of the test program is for sets implemented as unordered lists. Near-identical versions were created for the three other representations of sets.

Firstly a new set is created by inserting several numbers (lines 1–4). Some duplicate insertions are included, although these should have no effect. A number `inSet` which is known to be in the set and a number `notInSet` which is known not to be in the set are defined (lines 6 and 7). The string representation of the set and the number of items in the set are then printed (lines 9–11). Two tests are then performed to see if the selected numbers are in the set (lines 13–16) and not in the set (lines 18–21), as expected.

```
 1. setOfNumbers = UnorderedList()
 2. for setItem in [3, 3, 0, 5, 5, 3, 3, 1, 2, 3, 0, 4,
 3.                 5, 3, 2, 8, 9, 2, 7, 6, 1, 5]:
 4.   setOfNumbers.SetInsert(setItem)
 5.
 6. inSet = 5
 7. notInSet = 11
 8.
 9. print "The set is", setOfNumbers
10. print "The set contains " + str(setOfNumbers.setSize) +
11.       " items"
12.
13. if setOfNumbers.ElementOf(inSet):
14.   print "Number " + str(inSet) + " is in the set"
15. else:
16.   print "Number " + str(inSet) + " is not in the set"
17.
18. if setOfNumbers.ElementOf(notInSet):
19.   print "Number " + str(notInSet) + " is in the set"
20. else:
21.   print "Number " + str(notInSet) + " is not in the set"
```

By varying the items inserted into the set, and those that were searched for, we confirmed that the sets were being constructed correctly for each of the three data structures and each of the four set-membership algorithms.

## Appendix I Code Modifications for Counting the Number of Basic Operations

To measure the number of basic operations (Section 2.1) performed by the algorithms it was necessary to modify the code to maintain a counter variable, and to have each set-membership method return the number of operations. This was done for each of the four algorithms as follows.

The following version of the `ElementOf` method from Appendix D was created to count basic operations for set membership of unordered lists. Underlined code is new or modified.

```
1(a). def ElementOfCount(self, searchNum):
   2.   opCount = 0
3(b).   currNode = self.firstNode
```

```
 4(c).   while currNode != None:
    5.     opCount = opCount + 1
 6(d).     if currNode.Contents == searchNum:
    7.       return opCount
 8(f).     currNode = currNode.Next
    9.   return opCount
```

Firstly, a new counter variable `opCount` is initialised (line 2). It is incremented immediately before the contents of the current linked list node are compared with the search key (line 5). When the method terminates it returns the value of the counter, rather than the usual Boolean result (lines 7 and 9).

We made similar modifications to the method for ordered linked lists from Appendix E. The counter is incremented on line 6 to account for the equality comparison on line 5. It has 2 added to it on line 9, because if the code reaches this point then both the equality comparison on line 5 and the inequality comparison on line 8 must have been evaluated. Similarly, 2 is added on line 12 because the comparisons on lines 5 and 8 must have been performed to reach this point. Again the code is modified to return the operation count, instead of the usual Boolean result (lines 7, 10 and 14).

```
 1(h). def ElementOfCount(self, searchNum):
    2.   opCount = 0
 3(i).   currNode = self.firstNode
 4(j).   while currNode != None:
 5(k).     if currNode.Contents == searchNum:
    6.       opCount = opCount + 1
    7.       return opCount
 8(m).     elif searchNum < currNode.Contents:
    9.       opCount = opCount + 2
   10.       return opCount
   11.     else:
   12.       opCount = opCount + 2
13(o).       currNode = currNode.Next
   14.   return opCount
```

We modified the ‘efficient’ code for set membership from Appendix F similarly. The operation counter is incremented in the loop body (line 6) to account for the inequality comparison in the loop (line 5). Recall from the algorithm in Figure 3 that an additional equality comparison is done at the end of the code if the end of the list has not been reached (line 7 in Appendix F). Therefore, the code below either adds 2 to the counter when the end of the list has not been reached (line 9), to count the final evaluation of the loop condition on line 5 and the equality comparison in the `return` statement on line 7 of Appendix F, or simply returns the counter value (line 11 below), because the end of the list has been reached and neither of these comparisons is performed.

```
 1(q). def ElementOfCount(self, searchNum):
    2.   opCount = 0
 3(r).   currNode = self.firstNode
 4(s).   while currNode != None and
 5(s).         currNode.Contents < searchNum:
    6.     opCount = opCount + 1
 7(t).     currNode = currNode.Next
    8.   if currNode != None:
    9.     return opCount + 2
   10.   else:
   11.     return opCount
```

We modified the set-membership code for binary search trees from Appendix G as follows. The operation counter is incremented on line 6 to account for the equality comparison on line 5. It has 2 added to it on line 9 because both comparisons on lines 5 and 8 must have been performed to reach this point. Similarly on line 12. The counter value is returned when the method terminates (lines 7 and 14).

```
 1(v). def ElementOfCount(self, searchNum):
    2.   opCount = 0
 3(w).   currNode = self.rootNode
 4(x).   while currNode != None:
 5(y).     if currNode.Contents == searchNum:
    6.       opCount = opCount + 1
    7.       return opCount
 8(α).     elif searchNum < currNode.Contents:
    9.       opCount = opCount + 2
10(β).       currNode = currNode.Left
   11.     else:
   12.       opCount = opCount + 2
13(γ).       currNode = currNode.Right
   14.   return opCount
```

## Appendix J Code to Count the Number of Basic Operations

This appendix presents the programs written to count the number of basic operations performed by the four algorithms for sets of different sizes. There were two programs, one for checking set membership when the search key is in the set and one for when the search key is not in the set. The two programs are nearly identical, so only one is described in detail.

The following program counts the number of basic operations performed when the search key is in the set. It relies on the augmented versions of the four `ElementOf` methods shown in Appendix I. This is the program used to generate the results presented in Figure 5.

```
 1. itemRange = 1000000
 2. maxSetSize = 1000
 3. setSep = 100
 4. numTests = 100
 5.
 6. unorderedTrace = open('opsUnorderedAvg.txt', mode='w')
 7. orderedTrace = open('opsOrderedAvg.txt', mode='w')
 8. efficientTrace = open('opsEfficientAvg.txt', mode='w')
 9. treeTrace = open('opsTreeAvg.txt', mode='w')
10.
11. setSize = 0
12. while setSize <= maxSetSize:
13.
14.   unorderedOpsTotal = 0
15.   orderedOpsTotal = 0
16.   efficientOpsTotal = 0
17.   treeOpsTotal = 0
18.
19.   setSizeTotal = 0
20.
```

```
21.   for testNum in xrange(numTests):
22.
23.     searchKey = int(itemRange * random.random())
24.     keyPosition = int(setSize * random.random())
25.
26.     unorderedSet = UnorderedList()
27.     orderedSet = OrderedList()
28.     efficientSet = EfficientList()
29.     treeSet = SearchTree()
30.
31.     for insertNum in xrange(setSize):
32.
33.       newNum = int(itemRange * random.random())
34.       while newNum == searchKey:
35.         newNum = int(itemRange * random.random())
36.
37.       if insertNum == keyPosition:
38.          newNum = searchKey
39.
40.       unorderedSet.SetInsert(newNum)
41.       orderedSet.SetInsert(newNum)
42.       efficientSet.SetInsert(newNum)
43.       treeSet.SetInsert(newNum)
44.
45.     setSizeTotal = setSizeTotal + unorderedSet.setSize
46.
47.     unorderedOpsTotal = unorderedOpsTotal +
48.                 unorderedSet.ElementOfCount(searchKey)
49.     orderedOpsTotal = orderedOpsTotal +
50.                   orderedSet.ElementOfCount(searchKey)
51.     efficientOpsTotal = efficientOpsTotal +
52.                 efficientSet.ElementOfCount(searchKey)
53.     treeOpsTotal = treeOpsTotal +
54.                      treeSet.ElementOfCount(searchKey)
55.
56.   unorderedTrace.write(str(setSizeTotal // numTests) +
57.        " " + str(unorderedOpsTotal // numTests) + "\n")
58.   orderedTrace.write(str(setSizeTotal // numTests) +
59.          " " + str(orderedOpsTotal // numTests) + "\n")
60.   efficientTrace.write(str(setSizeTotal // numTests) +
61.        " " + str(efficientOpsTotal // numTests) + "\n")
62.   treeTrace.write(str(setSizeTotal // numTests) + " " +
63.                   str(treeOpsTotal // numTests) + "\n")
64.
65.   setSize = setSize + setSep
66.
67. unorderedTrace.close()
68. orderedTrace.close()
69. efficientTrace.close()
70. treeTrace.close()
```

Firstly some constants are defined to control the experiment. The numbers used to populate the set are in the range 0 to `itemRange`$-1$ (line 1). Trials are performed with sets ranging in size from 0 to `maxSetSize`, in steps of size `setSep` (lines 2–3). For each size of set an average result is produced for `numTests` randomly-generated sets (line 4).

Four separate files are opened to hold the test results (lines 5–9) and are closed at the end of the experiment (lines 67–70). The `setSize` variable (line 11) is used to control the series of trials performed (lines 12 and 65).

Since the search key is required to appear in the set, the code predetermines what the search key will be (line 23) and when to insert it while populating the set (line 24). The four different sets are then created (lines 26–29), and are filled with random numbers (lines 31–43). Care is taken to choose a number other than the search key (lines 33–35), except when the predetermined insertion point is reached (lines 37–38) in which case the predetermined number is used. The chosen new number is then inserted into each of the four sets. This approach ensures that all four sets are populated with exactly the same data.

For each size of set (line 12), four variables are initialised to hold the total number of basic operations performed for each of the four algorithms (lines 14–17). The set membership calculation is then performed using each of the four algorithms (lines 47–54), and the average value for each size of set is calculated and written to the trace files (lines 56–63).

Variable `setSizeTotal` (lines 19 and 45) is used to accumulate the total number of items in the sets, as opposed to the number of items inserted. This was done to allow for the possibility that duplicate values are produced by the random number generator for insertion into the sets. Since duplicates do not make the sets larger, this could skew the results. (As it happened, the random number generator rarely produced duplicate numbers, and this refinement made no significant difference to the results.)

The program for counting the number of basic operations when the search key is *not* in the set was exactly the same as that above except that lines 37 and 38 were omitted. This ensured that every number inserted into the sets was distinct from the search key. Thus the code above, with lines 37 and 38 removed, was used to produce the results shown in Figure 6.

## Appendix K Code to Measure Execution Times

This appendix presents the programs written to measure the execution times of the four algorithms for sets of different sizes. There are two programs, one for checking set membership when the search key is in the set and one for when the search key is not in the set. The two programs are nearly identical, so only one is described in detail.

The following program measures the amount of time required to test set membership when the search key is in the set. This is the program used to generate the results presented in Figure 7. The program's structure is very similar to that described for counting basic operations in Appendix J, so only the new features are described below. Note that this program relies on the unaugmented `ElementOf` methods shown in Appendix D to Appendix G because we didn't want to include the time required to maintain the operation counters.

```
 1. itemRange = 1000000
 2. maxSetSize = 1000
 3. setSep = 100
 4. numTests = 100
 5. numRepeats = 100
 6.
 7. unorderedTrace = open('timesUnorderedAvg.txt', mode='w')
 8. orderedTrace = open('timesOrderedAvg.txt', mode='w')
 9. efficientTrace = open('timesEfficientAvg.txt', mode='w')
10. treeTrace = open('timesTreeAvg.txt', mode='w')
11.
12. setSize = 0
```

```
while setSize <= maxSetSize:

  unorderedTimeTotal = 0
  orderedTimeTotal = 0
  efficientTimeTotal = 0
  treeTimeTotal = 0

  setSizeTotal = 0

  for testNum in xrange(numTests):

    searchKey = int(itemRange * random.random())
    keyPosition = int(setSize * random.random())

    unorderedSet = UnorderedList()
    orderedSet = OrderedList()
    efficientSet = EfficientList()
    treeSet = SearchTree()

    for insertNum in xrange(setSize):

      newNum = int(itemRange * random.random())
      while newNum == searchKey:
        newNum = int(itemRange * random.random())

      if insertNum == keyPosition:
        newNum = searchKey

      unorderedSet.SetInsert(newNum)
      orderedSet.SetInsert(newNum)
      efficientSet.SetInsert(newNum)
      treeSet.SetInsert(newNum)

    setSizeTotal = setSizeTotal + unorderedSet.setSize

    StartTime = time.clock()
    for repeatSearch in xrange(numRepeats):
      unorderedSet.ElementOf(searchKey)
    EndTime = time.clock()
    unorderedTimeTotal = unorderedTimeTotal +
                         EndTime - StartTime

    StartTime = time.clock()
    for repeatSearch in xrange(numRepeats):
      orderedSet.ElementOf(searchKey)
    EndTime = time.clock()
    orderedTimeTotal = orderedTimeTotal +
                       EndTime - StartTime

    StartTime = time.clock()
    for repeatSearch in xrange(numRepeats):
      efficientSet.ElementOf(searchKey)
    EndTime = time.clock()
    efficientTimeTotal = efficientTimeTotal +
67.                          EndTime - StartTime
68.
69.     StartTime = time.clock()
70.     for repeatSearch in xrange(numRepeats):
71.       treeSet.ElementOf(searchKey)
72.     EndTime = time.clock()
73.     treeTimeTotal = treeTimeTotal +
74.                     EndTime - StartTime
75.
76.   unorderedTrace.write(str(setSizeTotal // numTests) +
77.     " " +
78.     str(unorderedTimeTotal / (numTests * numRepeats)) +
79.     "\n")
80.   orderedTrace.write(str(setSizeTotal // numTests) +
81.     " " +
82.     str(orderedTimeTotal / (numTests * numRepeats)) +
83.     "\n")
84.   efficientTrace.write(str(setSizeTotal // numTests) +
85.      " " +
86.     str(efficientTimeTotal / (numTests * numRepeats)) +
87.      "\n")
88.   treeTrace.write(str(setSizeTotal // numTests) + " " +
89.     str(treeTimeTotal / (numTests * numRepeats)) + "\n")
90.
91.   setSize = setSize + setSep
92.
93. unorderedTrace.close()
94. orderedTrace.close()
95. efficientTrace.close()
96. treeTrace.close()
```

Since it is difficult to accurately measure short execution times, each set-membership operation was executed several times and the total time was then divided by the number of repetitions. Constant `numRepeats` determines how many times to repeat each test (line 5). After the sets are populated, each of the set-membership methods is called repeatedly, and the total execution time is measured (lines 48–73). The resulting execution times written to the trace files (lines 75–89) were the total divided by the number of tests and the number of repetitions.

The program to measure execution times when the search key is *not* in the set was exactly the same as that above except that lines 38 and 39 were omitted, to ensure that the search key is not inserted into the set. With this modification, the program above produced the results shown in Figure 8.