Incorporating Coverage Criteria in Bounded Exhaustive Test Generation of Structural Inputs

Nazareno Aguirre\textsuperscript{1}, Valeria Bengolea\textsuperscript{1}, Marcelo Frias\textsuperscript{2}, and Juan Pablo Galeotti\textsuperscript{3}

\textsuperscript{1} Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto and CONICET, Río Cuarto, Córdoba, Argentina. E-mail: \{naguirre, vbengolea\}@dc.exa.unrc.edu.ar

\textsuperscript{2} Departamento de Ingeniería Informática, Instituto Tecnológico Buenos Aires and CONICET, Buenos Aires, Argentina. E-mail: mfrias@itba.edu.ar

\textsuperscript{3} Departamento de Computación, FCEyN, Universidad de Buenos Aires and CONICET, Buenos Aires, Argentina. E-mail: jgaleotti@dc.uba.ar

Abstract. The automated generation of test cases for heap allocated, complex, structures is particularly difficult. Various state of the art tools tackle this problem by bounded exhaustive exploration of potential test cases, using constraint solving mechanisms based on techniques such as search, model checking, symbolic execution and combinations of these. In this report we present a technique for improving the bounded exhaustive constraint based test case generation of structurally complex inputs, for “filtering” approaches. The technique works by guiding the search considering a given black box test criterion. Such a test criterion is incorporated in the constraint based mechanism so that the exploration of potential test cases can be pruned without missing coverable classes of inputs, corresponding to the test criterion.

We present the technique, together with some case studies illustrating its performance for some white box/black box testing criteria. The experimental results associated with these case studies are shown in the context of Korat, a state of the art tool for constraint based test case generation, but the approach is applicable in other contexts using a filtering approach to test generation.

1 Introduction

Testing is a powerful and widely used technique for software quality assurance\textsuperscript{[8]}. The technique essentially consists of executing a piece of code, whose quality needs to be assessed, under a number of particular inputs, or test cases. For these test cases to be adequate, they generally need to try the software under different circumstances. A variety of test criteria have been devised, which basically define which are the different situations that a set of test cases must exercise, or cover\textsuperscript{[5]}.

Generating test cases is generally a complex activity, in which the engineer in charge of the generation has to come up with inputs that satisfy, in many cases,
complex constraints. The problem is particularly difficult when the inputs to be
generated involve complex, heap allocated, structures, such as balanced trees,
grahps, etc. Some tools [3, 19, 9, 13, 6] tackle this problem rather successfully, by
bounded exhaustive exploration of potential test cases. More precisely, these tools
work by generating all the inputs, within certain bounds (maximum number of
objects of each of the classes involved in the structure), that satisfy a given
constraint using some kind of constraint solver. Among the possible constraint
solving techniques, model checking and other search related mechanisms have
been implemented into state of the art tools.

In order to make the bounded exhaustive generation feasible, different mech-
anisms are implemented so that, in some cases, redundant structures are avoided,
and such that parts of the state space corresponding to invalid structures are not
explored. For instance, Korat implements a symmetry breaking mechanism to-
gether with an approach for avoiding the generation of invalid structures based
on a sophisticated pruning technique; TestEra uses Alloy and its underlying
symmetry breaking and optimisation mechanisms to improve the generation;
UDITA implements a novel lazy evaluation mechanism, which in combination
with symbolic execution greatly improve the test generation process.

In this work, we consider a complement to the so called filtering approach
[6] to bounded exhaustive test generation, i.e., the process of exhaustively gen-
erating all possible structures (within the established bounds) and “filtering” to
keep the valid or well formed ones. This complement takes into account a test
criterion as part of the “generate and filter” process. Basically, in the same way
that symmetric structures are avoided, we propose to also avoid the exploration
of portions of the search space for test input candidates when such portions
are guaranteed to provide test inputs corresponding to classes already covered
by other test inputs previously generated. The result is somehow in between
bounded exhaustive and “optimal” equivalence class coverage, and the actual
“exhaustiveness” of the technique depends on the interaction of the test crite-
rior (e.g., the adequacy of the predicates used for equivalence class coverage)
and the generation procedure.

The motivation for this work lies in the fact that, by bounded exhaustive
generation of test cases, in many cases, it becomes costly or infeasible to test
a piece of code for all valid bounded inputs, even for small bounds, due to the
large number of inputs obtained. Thus, a test criterion might be employed in
order to “prune” the test generation, achieving a bounded exhaustive coverage
of equivalence classes associated with the criterion. For instance, suppose that
one counts with a test criterion for a given program to test. If one is interested
in equivalence class coverage, it would be enough to generate a single test input
per each (feasible) equivalence class. On the other hand, by bounded exhaustive
generation one would be building all valid structures within the provided bounds.
Instead, we propose to do a kind of exhaustive generation, but exploiting the
possibility of pruning parts of the search space when one is certain that all
candidates in the pruned part correspond to classes already covered.
More precisely, our proposed technique works based on the following observation. The test generation mechanisms that follow a bounded exhaustive, filtering approach, generally contain a process for avoiding the generation of redundant structures. Independently of how such processes are implemented, they all correspond essentially to a pruning operation. For instance, the symmetry breaking formulas that TestEra incorporates in the Alloy model resulting from a program to be tested, instruct the underlying SAT solver to skip parts of the search space (in this context, assignments to propositional variables), thus constituting a pruning [9]. Similarly, UDITA prunes the search space to avoid isomorphic structures by incorporating isomorphism avoidance into the object pool abstraction and the operations for obtaining new objects from it in the construction of heap allocated structures [6]; Korat performs a similar pruning, imposing an ordering in the objects of the same type in the process of building the heap allocated structures [3]. Our approach proposes to take advantage of such pruning processes but in a different way; instead of just eliminating isomorphic (redundant) structures, we propose to take extra advantage of the pruning, and use it for skipping portions of the search space that would produce test cases covering classes that have already been covered by previous tests.

We present the approach by implementing it as a variant of the Korat algorithm/tool [13]. This variant is based on the use of a routine that we call eqClass(), that given a (valid) test input indicates which is the equivalence class it corresponds to, according to a test criterion. As for repOk(), we will require this routine to be deterministic. Basically, our variant of Korat, which we will refer to as Korat+, works as follows: when a candidate is found to be a valid test case (i.e., it satisfies the repOk()), we invoke the eqClass() routine for this candidate, and look at what fields are observed to determine its equivalence class. We then try to “skip” the structures that coincide, for the observed fields, with the current candidate. Clearly, for any other candidate with the same values in these observed fields, its equivalence class would be the same as that of the current candidate.

We describe our approach in detail, via the mentioned variant of Korat, we provide examples and cases studies with their associated experimental results, using black box as well as white box test criteria. As it will be shown later on, our variant results in significant improvements for Korat’s search for some of our case studies, in particular for black box testing. As it is further explained later on, in some cases we were able to reduce the search space substantially, as well as to produce significantly less test cases, compared to bounded exhaustive generation. As we mentioned before, the technique results to be between bounded exhaustive, and “optimal” equivalence class coverage.

2 Preliminaries

In this section we describe the Korat algorithm, which we use to present our technique, and discuss briefly the notions of coverage and test criterion. We also introduce a motivating example to drive the presentation.
2.1 Test Criteria

Exhaustively testing a piece of code is generally infeasible, due to the usually very large (many times infinite) space of possible inputs for the code under analysis. Thus, one generally has to consider only a restricted set of inputs to test the software. In order to test the software under various different situations, and thus increase the chances of finding bugs, different test criteria have been proposed [5]. A test criterion allows one to check whether a set of test cases is adequate, in the sense that it exercises the software under a sufficiently heterogeneous set of different situations. There exist two broad classes of test criteria, white box, which take into account the structure of software, and black box, which take into account only the specification of software, but disregard its internal structure.

As an example of a black box test criterion, we might consider parameterless boolean predicate (PBP) coverage [11]. This test criterion applies to procedures whose input is an object, in the object oriented programming sense. It consists of observing the class corresponding to the input, and the (public) parameterless boolean services it provides. For instance, a stack might have parameterless boolean predicates `isEmpty()` and `isFull()`. A set of test cases is adequate with respect to this test criterion if each feasible combination of boolean values for the parameterless boolean predicates is satisfied by at least one of the test cases. For instance, for stacks with `isEmpty()` and `isFull()`, we would need in principle four test cases, one that is empty and full, one that is empty and not full, one that is not empty and full, and one that is neither empty nor full. Clearly, only the first of these is infeasible. Technically, it corresponds to a special case of equivalence class partitioning [2], the main black box technique employed in this report.

As an example of a white box test criterion, we can consider, for instance, decision coverage. According to this criterion, a set of test cases is adequate if each decision in the program under analysis (guards of if-then-else, guards of iteration statements, etc.) is evaluated as true and as false by some test cases.

2.2 The Korat Algorithm

Korat is an algorithm, and a tool implementing it, that allows for the generation of test cases composed of complex, heap allocated, structures [3]. Suppose, for instance, that we need to test a procedure that takes as a parameter a sorted singly linked list. Let us consider the following definition of the structure of sorted singly linked lists (a variant of `SinglyLinkedList`, as provided in Korat’s distribution):

```java
class SortedSinglyLinkedList {
    Node header;
    int size;
}

class Node {
    Integer elem;
    Node next;
}
```

A list is composed of a reference to a header node, and an integer value indicating the length of the list. The linked list of nodes starts with a header node
with no element (a traditional dummy node); the actual contents of the list
starts from the second node. The list should be acyclic, sorted (disregarding
the header node), and the number of nodes in it minus one (the header) should
coincide with the size value of the list. Korat can be used to generate such
lists automatically, so that the procedure under analysis can be tested. Korat
requires two routines accompanying the class associated with the input (in our
example, SortedSinglyLinkedList). One of them is a boolean parameterless
routine, called repOk() [10], that checks whether the structure satisfies its rep-
resentation invariant. In our case, repOk() should check that the header is not
null, and that no element is stored in it, that the list is acyclic and sorted,
and the number of nodes in it minus one coincides with the value of the size
field, as we explained before. The other routine that Korat requires is a fini-
tisation procedure, that provides the bounds for the domains involved in the
structure. This routine indicates the range for primitive type fields (e.g., that
size in SortedSinglyLinkedList goes from 0 to 3), and the minimum/maxi-
mum number of objects of the classes involved in the structure (e.g., 1 list, 0 to
4 nodes, 1 to 3 integer objects).

Korat generates all possible valid structures within the provided bounds. By
valid we mean that they satisfy the repOk() routine. For our example, this
means that Korat will generate all acyclic sorted singly linked lists with dummy
header, where the size coincides with the number of nodes in the linked list
minus one, of size at most 3, containing integers from 1 to 3. In order to do so,
Korat builds a tuple, where each entry corresponds to a value of a field of the
involved objects. In our example, the tuple would have length 10, two values for
the header and size of the lone list object, and the other 8 for the correspon-
ding two fields of the four nodes that the list might at most contain. For instance,
tuple

\[(0, 0, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL)\]

would represent the empty list (where the first zero in the tuple is the reference
to the first node object). Each entry in this tuple has a domain, which is defined
by the finitisation procedure. Korat’s actual algorithm works on what are called
candidate vectors, vectors that represent the candidate tuples, but where the
actual entries are replaced by indices into the respective domains. For instance,
the candidate vector \( (1, 0, 0, 0, 0, 0, 0, 0, 0, 0) \) would correspond to the previously
shown candidate tuple (each tuple entry has the first possible value in its domain,
i.e., the value with index 0, except for the first entry, the head of the list, which
points to the first node). Typically, most of the candidate vectors correspond to
invalid structures, i.e., structures that do not satisfy the repOk(). Indeed, the
space of candidates is in our example 3200000 \((5^5 \times 4^5)\), but there exist only 8
singly sorted linked lists (up to isomorphism, as it will be explained later on),
within the provided bounds.

Korat exhaustively explores the space of candidate vectors, using backtracking
with a sophisticated pruning mechanism. More precisely, Korat works as fol-
low: it starts with the initial candidate vector, with all indices in zero. It
then executes \texttt{repOk()} on this candidate, monitoring the fields accessed in the execution, and storing these in a stack. Korat will then use this stack in order to backtrack over candidate vectors, as follows. If the current candidate satisfies \texttt{repOk()}, it is considered a valid test case (in this case, all reachable fields must be in the stack of accessed fields). If \texttt{repOk()} fails, then the candidate is discarded. In order to build the next candidate, Korat increments the last accessed field to its next value. If one or more of the last accessed fields are already in their corresponding maximum values, then these are resetted to 0, and the field accessed before them is incremented. If all fields are already at their maximum values, then the state space of candidate vectors has been exhaustively explored, and Korat terminates.

Notice that when \texttt{repOk()} fails, not all reachable fields might have been accessed, since its failure might be determined before exploring all reachable fields (for instance, in our example, if the first two nodes of the list are unordered, then \texttt{repOk()} fails without the need to explore the remaining part of the structure). Backtracking only on accessed fields is what enables Korat to prune large parts of the space of candidate vectors. It is sound since if the last accessed field is not modified, the output for \texttt{repOk()} would not change due to its determinism, (i.e., the parts of the structure visited by \texttt{repOk()} would remain the same, and therefore \texttt{repOk()} would fail again).

Besides the described search mechanism, with its incorporated search pruning, Korat also avoids generating isomorphic candidates \cite{3}. Basically, two candidates are isomorphic if they only differ in the object identities of their constituents (i.e., if one of the candidates can be obtained from the other by changing the object identities). Most applications do not depend on the actual identities of objects (which represent the memory addresses or heap references of objects), and thus if one generated a structure, it is desirable to avoid generating its isomorphic structures, whose treatment would be redundant. Korat avoids generating isomorphic candidates by defining a lexicographic order between candidate vectors, and generating only the smallest in the order, among all isomorphic candidates. Basically, when considering the range of a class-typed field (i.e., its possible values) in the construction of candidates during the search, it is restricted to up to one “untouched” (i.e., not previously referenced in the structure) object of its corresponding domain. For example, suppose that in the construction of candidates one needs to consider different values for a given position \(i\) in the candidate vector. Suppose further that the \(i\)th position corresponds to a class domain \(D\), and no fields of that domain have been accessed before \(i\) in the last invocation of \texttt{repOk()}. Then the only possible value for the \(i\)th position is 0. More generally, if \(k\) different objects of domain \(D\) have been accessed before in the last invocation of \texttt{repOk()}, these must be indexed 0 to \(k - 1\), and thus the \(i\)th position can go from 0 to \(k\), but not further from \(k\). Korat’s pruning and isomorphism elimination mechanisms allow the tool to reduce the search space significantly, in many cases. For our example, for instance, Korat explores only 319 out of the 3200000 possible cases, for linked lists with length 0 to 3, up to
4 nodes, and values ranging in integer objects from 1 to 3. For more details, we refer the reader to [3, 13].

3 Incorporating Coverage to Bounded Exhaustive Search

In this section, we describe our proposal for improving a filtering approach to bounded exhaustive generation, by incorporating pruning associated with test criteria. Essentially, the approach is based on the observation that in many cases, the number of valid test cases, bounded by a value \( k \), can be too large even for small bounds, and therefore evaluating the software under all these cases might be impractical. Then, our intention is to skip the generation of some test cases; the idea is to avoid generating test cases whose corresponding equivalence classes, for the test criterion under consideration, have already been covered. The idea is not to do “optimal” equivalence class coverage (one per equivalence class), but to approximate somehow to bounded exhaustive generation. That is, we would like to do a kind of bounded exhaustive generation, but with some pruning based on what the test criterion provides as information. For instance, if a certain branch has already been covered by some test case, then other test cases exercising the same branch can be avoided, without compromising branch coverage.

We present the approach by implementing it as a variant of the Korat algorithm, introduced in the previous section. In the same way that Korat requires an imperative predicate \( \text{repOk}() \), we require a routine that we call \( \text{eqClass}() \). This routine returns, given a valid candidate (i.e., a candidate satisfying \( \text{repOk}() \)), the equivalence class the candidate corresponds to, according to a test criterion. As for \( \text{repOk}() \), this routine must be deterministic (for exactly the same reason that \( \text{repOk}() \) must be deterministic). As opposed to Korat, which prunes (advances various candidates at once) only when \( \text{repOk}() \) fails, since if it does not fail all reachable fields must be in the stack of accessed fields, we prune the search space both when \( \text{repOk}() \) succeeds and when it fails: when \( \text{repOk}() \) fails, we advance various candidates at once using the fact that if none of the accessed fields is changed, then \( \text{repOk}() \) would fail again. When \( \text{repOk}() \) succeeds, we execute \( \text{eqClass}() \) and monitor the accessed fields; we then advance various candidates at once to force a change in the last accessed field, since if none of the accessed fields changes the equivalence class would be the same of the previous valid candidate, which is already covered.

In order to better understand how this mechanism works, let us briefly expand our example. Suppose that we need a procedure \texttt{listAsSet} that, given a list \( l \) and a set \( s \), both implemented over linked lists, determines whether \( s \) is the result of converting \( l \) to a set, i.e., disregarding repetitions and the order of elements in the list. From an implementation point of view, and taking into account the representation invariant of sets over singly linked lists, \( s \) should be the result of removing repetitions and sorting the list \( l \). It is not difficult to find contexts in which a procedure of the kind of \texttt{listAsSet} is relevant. An obvious application of such a function would be an oracle for checking whether a list-to-set routine works as expected. If we want to generate test cases for
**listAsSet**, we need to provide two objects, namely an arbitrary (acyclic) singly linked list of integers (the list \( l \)), and a strictly sorted acyclic singly linked list (the “set” \( s \)); the repOk() routine for this pair of objects checks first whether \( l \) is acyclic, and if so, it then checks whether \( s \) is acyclic and strictly sorted. Korat’s candidate vectors will be composed of values for the fields of all objects of the two lists. Moreover, suppose that our test criterion takes into consideration all the combinations of four predicates:

- the first list is empty
- the first list has repeated elements
- the first list is sorted
- the second list is empty

The criterion is satisfied if at least one test case is produced for each of the satisfiable combinations of the truth values for the above predicates. Now, suppose that, in the search for valid candidates, Korat constructs the following pair of lists (for the linked list and the “set”, respectively):

\[
\begin{array}{c}
N_0 \\
\text{header} \\
N_0' \\
\text{header'} \\
N_1 \\
1
\end{array}
\]

Let us refer to this pair of lists as \((1, s)\). Clearly, \((1, s)\) satisfies repOk(). Let us analyse how Korat would proceed. Since repOk() is satisfied, Korat will move to the next candidate, which corresponds to advancing the last accessed field, i.e., \( N_1'.next \), assuming that repOk() checks first the representation invariants of \( l \) and \( s \) in this order. Furthermore, because of the way repOk() works, Korat will produce all valid sets “greater than” (in the sense of the order in which Korat produces them) \( s \), in combination with the empty \( l \), before advancing \( l \), i.e., producing a nonempty list.

Now let us analyse how our approach, that as we said we will refer to as Korat+, would proceed. According to the test criterion described before, this pair of lists corresponds to the equivalence class \((T, F, T, F)\) (i.e., the first list is empty, with no repeated elements and sorted, whereas the second one is nonempty). In order to determine the equivalence class for this test case, the parts of the structure that are examined are header \((N_0)\), \( N_0.next \), header’ \((N_0')\), \( N_0'.next \), in this order. Thus, if none of these fields are modified, the candidates produced would correspond to the same equivalence class as our current candidate \((1, s)\), which is already covered by this valid candidate. So, the approach “prunes” the search by attempting to advance the last accessed field, namely \( N_0'.next \), which is already at its maximum possible value (due to the rule of “at most one untouched object”). We move then to trying to advance header’, which again is at its maximum for the same reason as before, and thus we start considering greater values for \( N_0.next \). Notice how we avoided generating many (nonempty) sets, which in combination with the empty list would cover an already covered equivalence class. For example, if the finitisation procedure establishes that both lists have 0 to 4 nodes, and integers go from 1 to 3, then the described pruning,
associated with our approach, constructs 679 candidates (320 of which are valid), skipping the construction of 14000 candidates (240 of which are valid, but cover already covered classes) that Korat would generate.

3.1 Soundness of the Approach

Let us argue about the soundness of the approach with respect to equivalence class coverage, i.e., that any valid test case in the pruned state space corresponds to a previously covered equivalence class. For comparison purposes, let us introduce a pseudo-code description of the standard Korat algorithm:

```
function korat() {
    Vector current = initVector;
    Stack accessedFields = new Stack();
    boolean ok;
    do {
        (ok, accessedFields) = current.repOk();
        if (ok) {
            reportValid(current);
            accessedFields.push(current.reachableFields - accessedFields);
        }
        field = accessedFields.pop();
        while (!accessedFields.isEmpty() &&
            current[field] >= nonIsoMax(current, accessedFields, field)) {
            current[field] = 0;
            field = accessedFields.pop();
        }
        if (!accessedFields.isEmpty()) current[field]++;
    } while (current != lastVector && !accessedFields.isEmpty())
}
```

In this pseudo-code description of the algorithm, we make an abuse of notation and make `repOk()`, which applies to candidate vectors, return both the result of executing this function on the corresponding vector (a boolean, indicating whether the candidate is a valid one or not) and a stack with the fields accessed in the execution (`fields`). Notice how the backtracking is performed on the fields accessed by `repOk()`: also, when the current vector is valid, then all reachable fields are forced into the accessed fields, so that these are considered for backtracking and no candidates are missed. Finally, notice that an auxiliary function called `nonIsoMax`, which returns the maximum index possible for a given field, is used in order to determine the range of values for each field. This is crucial for the generation of nonisomorphic instances [3].

Our technique, which in this context we present as a variant of Korat referred to as Korat+, performs an extra pruning. It works by “popping out” more items from `fields`, the stack of accessed fields. In order to perform this pruning, the algorithm needs to compute the equivalence class for each valid candidate, monitoring the fields accessed in this computation (stored in `eqFields`). It then checks whether Korat’s standard “next candidate” computation already advanced some of the the fields accessed by the `eqClass()` routine, and if not it forces such an advance. The pseudocode for our variant is the following:
function koratPlus() {
    Vector curr = initVector;
    Stack fields = new Stack();
    boolean ok;
    do {
        (ok, fields) = curr.repOk();
        if (ok) {
            reportValid(curr);
            fields.push(curr.Fields - fields);
            (eqClass, eqFields) = curr.eqClass();
            reportEqClass(eqClass);
        }
        List modified = new List();
        field = fields.pop();
        while (!fields.isEmpty() &&
               curr[field] >= nonIsoMax(curr, fields, field)) {
            curr[field] = 0;
            modified.add(field);
            field = fields.pop();
        }
        if (!fields.isEmpty()) {
            curr[field]++;
            modified.add(field);
        }
        // extra pruning
        if (ok &&
            (eqFields - modified == eqFields)) {
            for each field in modified {
                curr[field] = 0
            }
        }
        boolean found = false;
        while (!fields.isEmpty() && !found) {
            field = fields.pop();
            if (eqFields.contains(field)) {
                found = true;
            } else {
                curr[field] = 0;
            }
        }
        if (found) {
            while (!fields.isEmpty() &&
                   curr[field] >= nonIsoMax(curr, fields, field)) {
                curr[field] = 0;
                field = fields.pop();
            }
            if (!fields.isEmpty()) {
                curr[field]++;
            }
        }
    }
}
Guaranteeing the soundness of this pruning approach is relatively straightforward. First, notice that the backtracking order of the original Korat algorithm is preserved: Korat+ backtracks over fields, the fields accessed by repOk(). Our variant can only “pop” more items, but not modify the accessed fields (and thus the order of backtracking) in any other way.

Let us see that this new pruning can only skip valid candidates of already covered classes. Suppose that this new pruning stage is activated. Then, the previous candidate, which we will refer to as v_p, is a valid candidate, since ok is true; moreover, the standard Korat computation of the next candidate did not modify any of the fields accessed by eqClass(). This last pruning stage modifies the last field, according to fields, appearing in eqFields. Let v be a candidate vector pruned by this process. Assume further that v is a valid candidate. Since this candidate was pruned in this extra pruning, it corresponds to the pruned search space, which coincides in its values of the eqFields with v_p. Then, v corresponds to the same test equivalence class as v_p, due to the determinism of eqClass(). Therefore, the candidates pruned in the extra pruning stage correspond to the same equivalence class of v_p, which has already been covered by this test case.

4 Case Studies

We now describe some of the case studies we selected for assessing the technique. At the end of this section we will briefly analyse the experimental results associated with these case studies.

4.1 Black Box Case Studies

listAsSet. Our first case study corresponds to the listAsSet routine, and the black box test criterion described before, which requires covering all combinations of the predicates “first list is empty”, “first list has repeated elements”, “first list is sorted”, and “second list is empty”. The repOk() and eqClass() routines have been implemented as described in Section 3. This is a simple case study, with few equivalence classes, but serves the purpose of showing the benefits of the technique. The experimental results are shown in the tables below. The scope indicates the size ranges for the two lists (as separated scopes), the maximum number of nodes in each list (as separated scopes), and the number of different integer values allowed, respectively. For Korat and Korat with coverage pruning (Korat+), the first table shows the number of explored vectors, together with how many of these are valid test cases (i.e., satisfying repOk()). We also indicate the number of classes covered, for the corresponding scope (the covered classes are the same for Korat and Korat+, due to the soundness of the technique). The second table and the chart in Fig. 1 compare the running times of Korat and Korat+ for this case study.
Our second case study has to do with binomial heaps. A fundamental operation of binomial heaps is the merge of two heaps, which can be performed very efficiently. Assuming one is interested in testing such a routine, it is necessary to provide pairs of binomial heaps. The merging of two binomial heaps depends very much on how these are composed, and the degrees of their composing binomial trees. Considering equivalence class partitioning as the black box test criterion, the following predicates should provide a suitable coverage:

- the first heap is empty,
- the second heap is empty,
- the first heap has more elements than the second,
- both heaps have the same number of elements,
- the first heap has a larger degree than the second,
- both heaps have the same degree, and
- the heaps contain a tree with the same degree.

We have used the implementation of binomial heaps, with its corresponding `rep0k()`, exactly as provided in the Korat distribution, replicated for the two binomial heaps. The domains for each of these have been defined disjoint, in the finitisation procedure. The experimental results are shown in the tables below. The scope indicates the maximum number of elements both heaps might have. The keys in the nodes range from zero to this value (repeated elements are allowed). The first table shows the number of explored vectors, together with how many of these are valid test cases. We also indicate the number of equivalence classes covered. The second table and the chart in Fig. 2 compare the running times of Korat and Korat+ for this case study.

### Table 1: Experimental Results

<table>
<thead>
<tr>
<th>Scope</th>
<th>Korat</th>
<th>Korat+</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-2,0-2,3,3,3</td>
<td>1,121(91)</td>
<td>185(26)</td>
<td>8</td>
</tr>
<tr>
<td>0-4,0-4,3,3,3</td>
<td>1,485(91)</td>
<td>211(26)</td>
<td>8</td>
</tr>
<tr>
<td>0-4,0-4,4,4,3</td>
<td>14,679(320)</td>
<td>679(80)</td>
<td>10</td>
</tr>
<tr>
<td>0-5,0-5,5,5,4</td>
<td>1,274,977(5,456)</td>
<td>6,798(682)</td>
<td>10</td>
</tr>
<tr>
<td>0-5,0-5,5,5,5</td>
<td>6,692,357(24,211)</td>
<td>16,369(1,562)</td>
<td>10</td>
</tr>
<tr>
<td>0-6,0-6,6,6,5</td>
<td>197,651,224(124,992)</td>
<td>89,650(7,812)</td>
<td>10</td>
</tr>
<tr>
<td>0-7,0-7,7,7,6</td>
<td>TIMEOUT</td>
<td>1,453,804(111,974)</td>
<td>10</td>
</tr>
</tbody>
</table>

### Table 2: Running Times

<table>
<thead>
<tr>
<th>Scope</th>
<th>Korat</th>
<th>Korat+</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-4,0-4,4,4,3</td>
<td>0.422s</td>
<td>0.269s</td>
</tr>
<tr>
<td>0-5,0-5,5,5,4</td>
<td>2.39s</td>
<td>0.395s</td>
</tr>
<tr>
<td>0-6,0-6,6,6,5</td>
<td>329,809s</td>
<td>0.817s</td>
</tr>
<tr>
<td>0-7,0-7,7,7,6</td>
<td>TIMEOUT</td>
<td>3.36s</td>
</tr>
</tbody>
</table>


Fig. 2. Binomial Heaps (Merge)

Directed Graphs. Our third case study corresponds to generating test cases for a routine manipulating a directed graph. The implementation of directed graphs is a standard object oriented implementation, consisting of a vector of vertices, each of which has a corresponding strictly sorted linked list, its adjacency list. Suppose that one is interested in generating case studies of varied arc “densities” and covering border cases; so, the combined graph characteristics considered for equivalence class partitioning could be the following:

- emptiness,
- density, and
- completeness.

The experimental results for this case study are shown in the tables below. The scope indicates the exact number of nodes in the directed graph. As for the previous cases, the first table shows the number of explored vectors, together with how many of these are valid test cases, and the number of classes covered. Notice that the number of valid cases grows too quickly, preventing us from reporting results for scopes higher than 3. The second table and the chart in Fig. 3 compare the running times of Korat and Korat+ for this case study.
**Weighted Directed Graphs.** Our fourth case study extends the previous one, to generating test cases for *weighted* directed graphs. The graph implementation is an extension of the one described above, in which each entry in the adjacency list of a vertex has a corresponding weight.

Some typical algorithms on weighted directed graphs are calculations of transitive closure or minimal path information, as for instance using Floyd’s algorithm. From the definition of minimal path some representative equivalence classes can be defined, based on properties of the graph:

- acyclicity,
- presence of negative weights, and
- connectedness of the graph.

They all play significant roles in the calculation of transitive closure or minimal path information. Thus, these are adequate predicates to consider for equivalence class coverage. In order to also get cases of varied arc “densities” and cover border cases, we also take into account emptiness, density and completeness of the structure, as for the previous case study. The experimental results for this case study are shown in the tables below. The scope indicates the exact number of nodes in the directed graph, and the range for weights. As for the previous cases, the first table shows the number of explored vectors, together with how many of these are valid test cases, and the number of classes covered. The second table and the chart in Fig. 4 compare the running times of Korat and Korat+ for this case study.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Korat</th>
<th>Korat+</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,-1-1</td>
<td>1,062(332)</td>
<td>984(256)</td>
<td>11</td>
</tr>
<tr>
<td>2,-2-2</td>
<td>2,272(1,542)</td>
<td>1,750(1,022)</td>
<td>11</td>
</tr>
<tr>
<td>3,-1-1</td>
<td>18,003,420(493,232)</td>
<td>17,815,155(304,982)</td>
<td>13</td>
</tr>
<tr>
<td>3,-2-2</td>
<td>33,122,848(15,612,660)</td>
<td>25,486,513(7,976,340)</td>
<td>13</td>
</tr>
<tr>
<td>3,-3-3</td>
<td>205,397,228(187,887,040)</td>
<td>103,127,315(85,617,142)</td>
<td>13</td>
</tr>
<tr>
<td>4,-4-4</td>
<td>TIMEOUT</td>
<td>TIMEOUT</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scope</th>
<th>Korat</th>
<th>Korat+</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,-2-2</td>
<td>0,632s</td>
<td>0,534s</td>
</tr>
<tr>
<td>3,-3-3</td>
<td>2812,665s</td>
<td>1333,942s</td>
</tr>
</tbody>
</table>

**Fig. 3.** Directed Graphs

**Fig. 4.** Weighted Directed Graphs
**Search Tree (Delete).** Our fifth black box case study is concerned with deletion in search trees. In this case, the test data to generate is composed of a combination of a search tree and a value to be deleted from it. The search tree implementation we considered is the one provided in the Korat distribution. The test case equivalence classes in this case correspond to the “position” of the value to be deleted in the tree; we have chosen the following cases:

- the value is not in the tree,
- the value is in the root,
- the value is in a leaf,
- the value is in a node with two (nonempty) subtrees,
- the value is in a node with a left subtree only, and
- the value is in a node with a right subtree only.

The experimental results for this case study are shown in the tables below. The scope indicates the maximum number of nodes in the tree, the range for the size of the tree, and the number of keys allowed in the tree. The second table and the chart in Fig. 5 compare the running times of Korat and Korat+ for this case study.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Korat</th>
<th>Korat+ CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,0-3,3</td>
<td>534(45)</td>
<td>500(43)</td>
</tr>
<tr>
<td>3,0-3,4</td>
<td>1,152(148)</td>
<td>1,011(125)</td>
</tr>
<tr>
<td>3,0-3,6</td>
<td>4,290(822)</td>
<td>3,331(586)</td>
</tr>
<tr>
<td>3,0-3,8</td>
<td>12,144(2,760)</td>
<td>8,675(1,793)</td>
</tr>
<tr>
<td>4,0-4,7</td>
<td>46,795(5,005)</td>
<td>34,240(3,089)</td>
</tr>
<tr>
<td>5,0-5,8</td>
<td>477,888(29,416)</td>
<td>338,292(16,137)</td>
</tr>
<tr>
<td>6,0-6,9</td>
<td>4,597,299(167,814)</td>
<td>3,213,270(83,511)</td>
</tr>
<tr>
<td>10,0-10,13</td>
<td>TIMEOUT</td>
<td>TIMEOUT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scope</th>
<th>Korat</th>
<th>Korat+</th>
</tr>
</thead>
<tbody>
<tr>
<td>3,0-3,6</td>
<td>0,423s</td>
<td>0,359s</td>
</tr>
<tr>
<td>4,0-4,7</td>
<td>0,88s</td>
<td>0,748s</td>
</tr>
<tr>
<td>5,0-5,8</td>
<td>1,607s</td>
<td>1,333s</td>
</tr>
<tr>
<td>6,0-6,9</td>
<td>7,529s</td>
<td>5,724s</td>
</tr>
</tbody>
</table>

**Fig. 5.** Search Tree

**SinglyLinkedList.** Our last black box case study has to do with acyclic singly linked lists of integers. Suppose that one is interested in testing a stable sorting algorithm. Considering equivalence class coverage, some representative equivalence classes can be defined based on the following predicates: “the list is empty”,
"the list has repetitions" and "the list elements are unsorted". The experimental results for this case study are shown in the table below.

The scope indicates the range for the size of the list, the maximum numbers of nodes in the list and the numbers of integer values allowed in each node of the list. The table shows the numbers of explored candidates together with the number of valid lists found and the number of covered classes. Notice that Korat as well as Korat+ produce the same output (explored vectors and valid test cases), i.e., not extra pruning is performed for Korat+. This is so because in order to determine the equivalence class to which a given test case belongs, the whole structure is examined.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Korat/Korat+</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,3,4,3</td>
<td>319(40)</td>
<td>4</td>
</tr>
<tr>
<td>0,4,5,4</td>
<td>3,388(341)</td>
<td>4</td>
</tr>
<tr>
<td>0,5,6,5</td>
<td>46,684(3,906)</td>
<td>4</td>
</tr>
<tr>
<td>0,6,7,6</td>
<td>781,960(55,987)</td>
<td>4</td>
</tr>
<tr>
<td>0,7,8,7</td>
<td>15,349,933(960,804)</td>
<td>4</td>
</tr>
</tbody>
</table>

4.2 White Box Case Studies

We have also analysed the technique for a few white box case studies. As it becomes apparent from the experimental results, the technique resulted to be more beneficial in some of the cases associated with black box testing.

*Insertion in Red Black Trees (Decision coverage).* Our first white box case study corresponds to analysing an insertion procedure for red black trees. The red black tree structure we used for the experiments is the one provided with the Korat distribution. The test criterion employed in this case is *decision coverage*. This coverage is quite suitable, since the procedure has only one composite condition (in a while loop). The experimental results for this case study are shown in the tables below, as well as Figure 6. The scope indicates the maximum number of nodes, the range for the size of the tree, and the range for the keys stored in the tree.
Fig. 6. Insertion in Red Black Trees

Search in Red Black Trees (Decision coverage). Our second white box case study corresponds to the same structure as the previous one, red black trees, now for a search operation. The test criterion employed in again decision coverage. The experimental results for this case study are shown in the tables below, as well as Figure 7. The scope indicates the maximum number of nodes, the range for the size of the tree, and the range for the keys stored in the tree.
Insertion in Search Trees (Decision coverage). We analysed a structure related to the previous one, namely search trees. First, we considered decision coverage for the insertion routine in search trees. The search tree implementation considered is the one available with the Korat distribution. All decision points in the insertion procedure for search trees are atomic, so decision coverage seems adequate. The experimental results for this case study are shown in the tables below, as well as Figure 8. The scope indicates the maximum number of nodes, the range for the size of the tree, and the range for the keys stored in the tree.
Finally, we generated test cases for search trees, taking into account the search operation and decision coverage. The structure employed is the same as for the previous case study. The search routine has no composite decision points, so again decision coverage seems to provide a suitable white box coverage. The experimental results for this case study are shown in the tables below, and Figure 9. The scope indicates the maximum number of nodes, the range for the size of the tree, and the range for the keys stored in the tree.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Korat</th>
<th>Korat+</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,0-2,2</td>
<td>62(10)</td>
<td>57(10)</td>
<td>6</td>
</tr>
<tr>
<td>2,0-2,4</td>
<td>228(68)</td>
<td>198(60)</td>
<td>7</td>
</tr>
<tr>
<td>3,0-3,3</td>
<td>534(45)</td>
<td>500(43)</td>
<td>8</td>
</tr>
<tr>
<td>3,0-3,6</td>
<td>4,290(222)</td>
<td>3,331(586)</td>
<td>8</td>
</tr>
<tr>
<td>4,0-4,4</td>
<td>4,660(204)</td>
<td>4,314(181)</td>
<td>8</td>
</tr>
<tr>
<td>4,0-4,8</td>
<td>87,032(10,600)</td>
<td>60,102(6,121)</td>
<td>8</td>
</tr>
<tr>
<td>5,0-5,5</td>
<td>41,540(940)</td>
<td>38,040(767)</td>
<td>8</td>
</tr>
<tr>
<td>5,0-5,10</td>
<td>1,720,830(142,250)</td>
<td>1,076,259(68,258)</td>
<td>8</td>
</tr>
<tr>
<td>6,0-6,6</td>
<td>371,700(43,386)</td>
<td>338,985(3,290)</td>
<td>8</td>
</tr>
<tr>
<td>6,0-6,12</td>
<td>32,702,676(1,960,884)</td>
<td>18,830,994(801,733)</td>
<td>8</td>
</tr>
<tr>
<td>7,0-7,7</td>
<td>3,301,956(20,650)</td>
<td>3,015,006(14,280)</td>
<td>8</td>
</tr>
</tbody>
</table>

Fig. 9. Search in Search Trees

5 Analysing the Assessment of our Case Studies

Let us briefly discuss now the results of our experimental analyses. First, notice that we have chosen to report, for each of the case studies, the number of explored candidates, accompanied by the corresponding number of valid candidates found. This is, in our opinion, the most reasonable measure to employ if one is interested
in evaluating the level of pruning that our technique contributes to standard filtering. In our cases, these numbers reflect directly in running times, because our \texttt{eqClass()} routines, the most influential (with respect to running time) part of the extra pruning section of our variant of Korat, do not increase in a noticeable way the running times of Korat for the scopes considered in these case studies. This is confirmed by the time tables corresponding to the experiments. However, we only report the running time for generation. One would expect that this would also reflect in the time necessary for actually testing for the produced inputs. All the experiments were run on a 3.06GHz Intel Core 2 Duo with 4GB of RAM, and the reported data correspond to experiments that terminated within our timeout of 5 hours.

The performance of technique, in this case implemented as a variant of Korat, depends greatly on the quality of \texttt{repOK()} and \texttt{eqClass()}, and how these relate to each other. For instance, in cases in which \texttt{eqClass()} needs to visit the whole structure in order to determine the equivalence class for the test case, there will be no extra pruning at all; this is due to the fact that the “next candidate” computation of Korat would have already advanced one of the fields observed by \texttt{eqClass()}, since it “observes everything”. So, the technique provides better results when the test criterion under consideration is such that by examining a small part of the structure one can determine a test case’s equivalence class. This is exactly the case in our two first black box case studies, in which the technique exhibits a better profit.

Another important factor in the performance of our technique implemented as Korat+ compared to Korat is in the size of the “valid candidates” space over the search space. More precisely, when \texttt{repOK()} fails very often, i.e., when the conditions for valid structures are stronger, then Korat exploits its associated pruning mechanism. It is when \texttt{repOK()} succeeds more often than it fails when Korat+ contributes more to the pruning, since while Korat would advance to the next candidate with no pruning, our extra pruning mechanism would try to prune candidates corresponding to the just covered equivalence class. This is observed, for instance, in the white box case studies that we presented. Notice that when for Korat the number of valid test cases is large in comparison with the number of explored candidates (\texttt{repOK()} succeeds more often), our extra pruning tends to contribute more to the pruning.

We have tried to foresee potential threats to the validity of our experimental results. We tried to be careful about the chosen case studies. Although our case studies correspond to relatively small pieces of code, they represent, in our opinion, rather natural testing situations in the context of the implementation of complex, heap allocated data structures (which is the main target for Korat). We have accompanied the presentation of each case study by a short justification of its appropriateness. We have included in our evaluation some case studies that have been successfully tackled by Korat, employing the same implementation available with Korat’s distribution (for which \texttt{repOK()} routines are tailored to exploit Korat’s search process).
One might argue that the equivalence classes used in these cases might prune too much, i.e., that these would show good pruning but would not be helpful for finding bugs. We decided then to take the three case studies for which we achieved more pruning, and make an analysis of how good would the obtained test suites be for finding bugs. These case studies are list as set, binomial heaps and search trees. We took three programs, namely standard implementations of listToSet, merge and deleteFromTree, for these structures, and performed the following experiment. We used muJava [12] in order to generate all method mutants of these three programs, and employed the test cases produced by Korat, by Korat+ and optimal equivalence class coverage (i.e., “one per equivalence class”), to see how many mutants can be killed by each of these test suites. The mutants are those obtained by the application of 12 different method-level mutation operators, e.g., arithmetic, logical and relational operator replacements, etc. (see [14] for a complete list of method level mutation operators). Not all of these mutation operators were applicable to our programs (6 were applicable to list as set, 5 were applicable to binomial heaps, and 4 were applicable to search trees). The results obtained are shown in the tables at the end of this section. Each table shows the total number of mutants and how many remained live after testing using the corresponding test suite. Notice that the results for Korat correspond to optimal mutant killing, since their corresponding test suites are bounded exhaustive (i.e., Korat kills as many mutants as possible within the corresponding bounds). In order to obtain a test suite for optimal equivalence class coverage (one per equivalence class), we take the first test case of each equivalence class from the bounded exhaustive test suite produced by Korat. As these experiments show, we achieve better results compared to one per equivalence class, and as the bounds are increased we get closer to bounded exhaustive test suites. Our intuition of being somehow “in between” optimal equivalence class coverage and bounded exhaustive is supported by the results.

<table>
<thead>
<tr>
<th>List as Set (49 mutants)</th>
<th>Scope</th>
<th>Korat</th>
<th>Korat+</th>
<th>One Per Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-2,0-2</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>0-4,0-4</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>0-4,0-4</td>
<td>3</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>0-5,0-5</td>
<td>3</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>0-5,0-5</td>
<td>3</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Binomial Heaps (117 mutants)</th>
<th>Scope</th>
<th>Korat</th>
<th>Korat+</th>
<th>One Per Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>38</td>
<td>39</td>
<td>44</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>7</td>
<td>7</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>17</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Search Trees (24 mutants)</th>
<th>Scope</th>
<th>Korat</th>
<th>Korat+</th>
<th>One Per Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3-3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3-3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3-3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>5-5</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>6-6,6</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
6 Conclusions and Future Work

We have presented a technique for improving bounded exhaustive test case generation using a filtering approach, by incorporating black box test criteria and employing these for pruning the search of valid test inputs. The approach targets structurally complex inputs, and essentially consists of incorporating into the usual pruning processes present in test generation techniques, an extra pruning that skips parts of the search space when one is certain that only candidates of classes of inputs already covered would be found. We implemented this technique as a variant of Korat, a tool/algorithm that automatically generates test cases by a “generate and filter” mechanism [3]. We argued about the technique’s correctness, and developed some case studies, whose associated experimental results enabled us to assess the benefits of the technique. The technique is somehow in between bounded exhaustive and “optimal” equivalence class coverage, and the actual “exhaustiveness” of the technique depends on the interaction of the test criterion (e.g., the adequacy of the predicates used for equivalence class coverage) and the generation procedure.

We presented the technique, and argued about its correctness. We also developed some case studies, providing some experimental results, and illustrating the benefits of the technique. Although in principle it applies both to black box and white box test case criteria, it appears to be more suitable for black box criteria. A reason for this is that the algorithm does not directly take into account the code being analysed, only the portion of the structure of the input visited by the analysed code. Thus, it does not provide the level of guidance for test case generation that other techniques, e.g. those based on model checking [18], provide. This is reflected by the fact that, in our implementation, the performance depends on the quality of the repOk() and eqClass(), and how these relate to each other. In particular, when eqClass() roughly respects the order in which repOk() visits the fields of the structure, and in cases in which a relatively small part of it suffices to determining its equivalence class, the technique is more beneficial. We also found that standard Korat works well when the valid test cases are relatively few with respect to the number of general structures, i.e., when the restrictions for the structure to be valid are stronger. In these cases, repOk() fails often, and thus Korat’s pruning improves the search significantly. On the contrary, when repOk() does not fail very often, Korat’s pruning is not exercised much. These are the cases in which our technique shows more profit. For instance, structures such as directed acyclic graphs or linked lists show better results than structures such as red black or AVL trees.

Automatic test case generation is an active area of research. For the particular case of test case generation of structurally complex, heap allocated inputs, various tools have been proposed. Among these we may cite Java PathFinder [18], Alloy [7], CUTE [15] and obviously Korat. A thorough comparison between these tools, reported in [16], shows that Korat (seen as a kind of specialised solver) is generally the most efficient, justifying our effort put in this approach. We already mentioned that Java PathFinder, based on model checking, is better suited for white box criteria, compared with our variant of Korat. Alloy, based
on SAT solving, is generally less efficient for test case generation, due to the
generation of isomorphic instances and its generally poorer performance, com-
pared to Korat. Recently, some of the authors of this paper have made significant
improvements in symmetry breaking and the scalability of SAT based analysis
using Alloy [4]. Thus, we are currently exploring the use of SAT based analysis
for test case generation guided by test criteria, using these recent improvements.
Besides this line of future work, we are also exploring the combination of Parallel
Korat [17] with our pruning based on test criteria.

References
1. M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann and
M. Veanes, Model-Based Testing with AsmL .NET, in Proceedings of the 1st Euro-
2. B. Beizer, Black-Box Testing: Techniques for Functional Testing of Software and
3. C. Boyapati, S. Khurshid and D. Marinov, Korat: Automated Testing based on Java
Predicates, in Proceedings of International Symposium on Software Testing and
efficient bounded verification, in Proceedings of the 19th International Symposium
5. C. Ghezzi, M. Jazayeri and D. Mandrioli, Fundamentals of Software Engineering,
6. M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak and D. Marinov, Test
generation through programming in UDITA, in Proceedings of the 32nd ACM/IEEE
International Conference on Software Engineering ICSE 2010, Cape Town, South
2006.
8. C. Kaner, J. Bach and B. Pettichord, Lessons Learned in Software Testing, Wiley,
9. S. Khurshid and D. Marinov, TestEra: Specification-Based Testing of Java Programs
10. B. Liskov and J. Guttag, Program Development in Java: Abstraction, Specification
11. L. Liu, B. Meyer and B. Schoeller, Using Contracts and Boolean Queries to Improve
the Quality of Automatic Test Generation, in Proceedings of 1st. International Con-
ference on Tests and Proofs TAP 2007, Zurich, Switzerland, LNCS 4454, Springer,
2007.
12. Y.-S. Ma, J. Offutt and Y.-R. Kwon, MuJava : An Automated Class Mutation
13. A. Milicevic, S. Misailovic, D. Marinov and S. Khurshid, Korat: A Tool for Gener-
ating Structurally Complex Test Inputs, in Proceedings of International Conference
C, in Proceedings of the 5th joint European Software Engineering Conference and
Symposium on the Foundations of Software Engineering ESEC/FSE 2005, ACM


**Appendix**

**ListAsSet**

```java
package koratPlus.ListAsSet;

import korat.finitization.IClassDomain;
import korat.finitization.IFinitization;
import korat.finitization.IIntSet;
import korat.finitization.IObjSet;
import korat.finitization.impl.FinitizationFactory;
import java.math.BigInteger;

import java.io.*;

/**
 * Class ListAsSet defines the set of objects needed to test a procedure
 * listToSet that given a list 'l' and a set 's' determines whether
 * 's' is the result of converting 'l' to a set, i.e. disregarding
 * repetitions and the order of element in 'l'.
 * This class needs to implement the Serializable interface since it
 * is required for the korat "serialize" option.
 * @author Nazareno M. Aguirre, Valeria Bengolea.
 *
 * @public
 * public class ListAsSet implements Serializable{

 public static final long serialVersionUID = 1;

 public SinglyLinkedList list;
 public StrictlySortedSinglyLinkedList set;

 /**
 * RepOk checks whether 'list' and 'set' satisfy their
 * representation invariant.
 * @return true iff 'list' and 'set' of the current instance
 * satisfy their representation invariant, false otherwise.
 *
 * @public
 * public boolean repOK() {
 * if (!list.repOK())
 * return false;
 * return set.repOK();
 * }
```
/**
 * eqClass returns the equivalence class in which the current
 * instance of ListAsSet belongs to. In this case, the equivalence
 * classes are given by all the possible combinations of predicates
 * isEmpty(), sorted() and noReps() over the list; and isEmpty()
 * over the set.
 * @return The equivalence class in which this instance belongs
 */
public BigInteger eqClass() {
    boolean[] classes = new boolean[4];
    classes[0] = list.isEmpty();
    classes[1] = list.noReps();
    classes[2] = list.sorted();
    classes[3] = set.isEmpty();
    return getClass(classes);
}

/**
 * getClass translates a given array of boolean into a Integer
 * @param classes The array to be translated
 * @return The integer that represents the given array
 * @see eqClass()
 */
public BigInteger getClass(boolean[] classes) {
    BigInteger res = new BigInteger("0");
    BigInteger dos = new BigInteger("2");
    for (int i = 0; i < classes.length; i++) {
        if (classes[i]) {
            BigInteger t = dos.pow(i);
            res = res.add(t);
        }
    }
    return res;
}

/**
 * listToSet checks whether 'set' is the result of converting 'list'
 * to a set.
 * @return true iff 'set' is the result of converting 'list' to a
 * set, i.e disregarding repetitions and the order of elements of
 * 'list'
 */
public boolean listToSet() {
    // converts the list in to a set/
    StrictlySortedSinglyLinkedList s = new
    StrictlySortedSinglyLinkedList();
    for (int i = 0; i < list.size; i++) {
        Integer value = list.get(i);
        this.addToSet(s, value);
    }
    // checks whether the to set are equals/
    for (int i = 0; i < s.size; i++) {
        Integer value = s.get(i);
        if (!s.contains(value)) {
            return false;
        }
    }
    for (int i = 0; i < s.size; i++) {
        Integer value = s.get(i);
        if (!set.contains(value)) {
            return false;
        }
    }
    return true;
}
** addToSet adds the given value to the given set.**

@param set, The set to add value into

@param value, the value to be added into the given set.

```java
public void addToSet(StrictlySortedSinglyLinkedList set, Integer value)
{
    Node current = set.header.next;
    Node previous = set.header;
    while (current != null && current.element.intValue() < value.intValue()){
        previous = current;
        current = current.next;
    }
    Node n = new Node();
    n.element = value;
    if (current == null) {
        previous.next = n;
        set.size++;
    } else {
        if (current.element.intValue() != value.intValue()) {
            previous.next = n;
            n.next = current;
            set.size++;
        }
    }
}
```

/**
 * finListAsSet provides a bound on the number of objects to be used to generate instances of ListAsSet.
 * @param minSizeList minimum size of the generated lists
 * @param maxSizeList maximum size of the generated lists
 * @param minSizeSet minimum size of the generated sets
 * @param maxSizeSet maximum size of the generated sets
 * @param numEntriesList number of entries that the list may contain
 * @param numEntriesSet number of entries that the set may contain
 * @param numElems The range of the elements contained in each entry of both, the list and the set, is between [1...numElems].
 * @return the object Korat needs for setting up the bounds during the search.
 */
```

```java
public static IFinimization finListAsSet(int minSizeList, int maxSizeList, int minSizeSet, int maxSizeSet,
    int numEntriesList, int numEntriesSet, int numElems) {
    IFinimization f = FinimizationFactory.create(ListAsSet.class);
    IObjSet entriesList = f.createObject(SinglyLinkedList.class, false);
    IObjSet entriesSet = f.createObject(StrictlySortedSinglyLinkedList.class, false);
    entriesList.addClassDomain(f.createClassDomain(SinglyLinkedList.class, 1));
    entriesSet.addClassDomain(f.createClassDomain(StrictlySortedSinglyLinkedList.class, 1));
    IObjSet entries = f.createObject(Entry.class, true);
```
```java
entries.addClassDomain(f.createClassDomain(Entry.class, numEntriesList));

IIntSet sizesList = f.createIntSet(minSizeList, maxSizeList);

IObjSet nodes = f.createObjectSet(Node.class, true);
nodes.addClassDomain(f.createClassDomain(Node.class, numEntriesSet));

IIntSet sizesSet = f.createIntSet(minSizeSet, maxSizeSet);

IObjSet elems = f.createObjectSet(Integer.class);
IClassDomain elemsClassDomain = f.createClassDomain(Integer.class);
elemsClassDomain.includeInIsomorphismCheck(false);

for (int i = 1; i <= numElems; i++)
elemsClassDomain.addObject(new Integer(i));
elems.addClassDomain(elemsClassDomain);
elems.setNullAllowed(true);

f.set("list", entriesList);
f.set("set", entriesSet);
f.set("SinglyLinkedList.header", entries);
f.set("StrictlySortedSinglyLinkedList.header", nodes);

f.set("SinglyLinkedList.size", sizesList);
f.set("StrictlySortedSinglyLinkedList.size", sizesSet);

f.set(Entry.class, "element", elems);
f.set(Entry.class, "next", entries);

f.set(Node.class, "element", elems);
f.set(Node.class, "next", nodes);

return f;
```
public boolean contains(Integer value) {
    Node current = header.next;
    while (current != null && current.element.intValue() <= value.intValue()) {
        if (current.element.intValue() == value.intValue())
            return true;
        current = current.next;
    }
    return false;
}

public boolean add(Integer value) {
    Node current = header.next;
    Node previous = header;
    while (current != null && current.element.intValue() < value.intValue()) {
        previous = current;
        current = current.next;
    }
    Node n = new Node();
    n.element = value;
    if (current == null) {
        previous.next = n;
        size++;
        return true;
    }
    if (current.element.intValue() == value.intValue())
        return false;
    if (current.element.intValue() > value.intValue()) {
        previous.next = n;
        n.next = current;
        size++;
        return true;
    }
    return true;
}

public Integer get(int index) {
    Node current = header.next;
    int i = 0;
    while (current != null && i < index) {
        current = current.next;
        i++;
    }
    if (current != null) {
        return current.element;
    }
    return null;
}

public int getSize() {
    return size;
}

/**
 * RepOk checks whether the list satisfies its invariant.
 */
public boolean repOK() {
    if (!repOkCommon())
        return false;
    return repOkSorted();
}

public boolean repOkCommon() {
    if (header == null)
        return false;
    if (header.element != null)
        return false;
    Set<Node> visited = new java.util.HashSet<>();
    visited.add(header);
    Node current = header;
    while (true) {
        Node next = current.next;
        if (next == null)
            break;
        if (next.element == null)
            return false;
        if (!visited.add(next))
            return false;
        current = next;
        if (visited.size() - 1 != size)
            return false;
    }
    return true;
}

public boolean repOkSorted() {
    if (!repOkCommon())
        return false;
    if (size > 1) {
        for (Node current = header.next; current.next != null;
             current = current.next) {
            if (current.element.compareTo(current.next.element) >= 0)
                return false;
        }
    }
    return true;
}

/**
 * Checks whether or not the current list has not elements.
 * @return true iff the current list is empty, false otherwise.
 */

public boolean isEmpty() {
    return header.next == null;
}

public String toString() {
    String res = "{";
    if (header != null) {
        Node cur = header.next;
        while (cur != null && cur != header) {
            res += cur.toString();
        }
    }
    return res + "}";
}
160 cur = cur.next;
161 }
162 } return res + "}";
164 }
165 } //End Class

package koratPlus.ListAsSet;
import java.io.Serializable;

/**
 * StrictlySortedSinglyLinkedList's nodes
 * This class needs to implement the Serializable interface
 * since it is required for the korat "serialize" option.
 * @author
 */
public class Node implements Serializable{

    public static final long serialVersionUID = 1;
    public Integer element;
    public Node next;

    public String toString() {
        return "[" + (element != null ? element.toString() : "null") + "]";
    }
}

package koratPlus.ListAsSet;
import java.util.Set;
import java.io.Serializable;

/**
 * Class SinglyLinkedList defines Singly linked List
 * This class needs to implement the Serializable interface
 * since it is required for the korat "serialize" option.
 * @author
 */
public class SinglyLinkedList implements Serializable{

    public static final long serialVersionUID = 1;

    public Entry header;
    public int size = 0;

    public SinglyLinkedList(){
        header = new Entry();
        size = 0;
    }

    public Integer get(int index){
        Entry current = header.next;
        int i = 0;
        while(current!=null && i< index){
            current = current.next;
            i++;
        }
        if(current!=null){

    }
    return current.element;
  }
  return null;
}

/**
 * RepOk checks whether the singlyLinkedList satisfies its
 * representation invariant.
 * @return True iff the current list satisfies its representation
 * invariant.
 */
public boolean repOK() {
  if (header == null)
    return false;
  if (header.element != null)
    return false;
  Set<Entry> visited = new java.util.HashSet<Entry>();
  visited.add(header);
  Entry current = header;
  while (true) {
    Entry next = current.next;
    if (next == null)
      break;
    if (next.element == null)
      return false;
    if (!visited.add(next))
      return false;
    current = next;
  }
  if (visited.size() - 1 != size)
    return false;
  return true;
}

public int getSize(){
  return size;
}

/**
 * Checks whether or not the current list has not elements.
 * @return true iff the current list is empty, false otherwise.
 */
public boolean isEmpty(){
  if (!isEmpty()){
    for (Entry current = header.next; current.next != null;
         current = current.next) {
      if (current.element.intValue()== current.next.element.
          intValue())
        return false;
    }
  }
  return true;
}

/**
 * Checks whether or not the current list has not repeated
 * elements.
 * @return true iff all the elements in the list are
 * different each other.
 */
public boolean noReps(){
  if (!isEmpty()){
    for (Entry current = header.next; current.next != null;
         current = current.next) {
      if (current.element.intValue()== current.next.element.
          intValue())
        return false;
    }
  }
  return true;
}

/**
 * Checks whether or not the current list is sorted.
 * Checks whether or not the current list is sorted.
 */
public boolean sorted() {
    if (!isEmpty()) {
        for (Entry current = header.next; current.next != null; 
            current = current.next) {
            if (current.element.intValue() > current.next.element. 
                intValue()) 
                return false;
        }
    }
    return true;
}

public String toString() {
    String res = "( ";
    if (header != null) {
        Entry cur = header.next;
        while (cur != null && cur != header) {
            res += cur.toString();
            cur = cur.next;
        }
    }
    return res + ")";
}

// End Class

package koratPlus.ListAsSet;
/**
 * SinglyLinkedList's nodes
 * This class needs to implement the Serializable interface
 * since it is required for the korat "serialize" option.
 * @author
 */
import java.io.Serializable;

public class Entry implements Serializable {
    public static final long serialVersionUID = 1;
    public Integer element;
    public Entry next;

    public String toString() {
        return "[" + (element != null ? element.toString() : 
            "null") + "]";
    }
}

Binomial Heaps (Merge)

package koratPlus.mergebinheap;
import java.math.BigInteger;
import korat.finitization.IClassDomain;
import korat.finitization.IFinitization;
import korat.finitization.IObjSet;
import korat.finitization.impl.FinitizationFactory;
import java.io.*;
/**
 * Class MergeBinHeap defines the set of object needed to test the
public class MergeBinHeap implements Serializable {

    public static final long serialVersionUID = 1;

    public BinomialHeap first;
    public BinomialHeap second;

    /**
     * checks that the two binomial heaps satisfy their representation invariants.
     * @return true iff 'first' and 'second' satisfy the corresponding representation invariants.
     */
    public boolean repOK() {
      if (!first.repOK())
        return false;
      return second.repOK();
    }

    public BinomialHeap merge() {
      BinomialHeap b = second.convert();
      return first.union(b);
    }

    /**
     * binaryFirstSize translates the size of the first binomialHeap to a string as an unsigned integer in base 2.
     * @return a string representation of the first's size as an unsigned integer in base 2.
     */
    public String binaryFirstSize() {
      return Integer.toBinaryString(first.getSize());
    }

    /**
     * binarySecondSize translates the size of the second binomialHeap to a string as an unsigned integer in base 2.
     * @return a string representation of the second's size as an unsigned integer in base 2.
     */
    public String binarySecondSize() {
      return Integer.toBinaryString(second.getSize());
    }

    /**
     * Checks whether or not the two binomialHeaps have not degree in common, using the binary representation of their sizes.
     * @return true iff first and second binomialHeaps do not have degree in common.
     */
    public boolean NoDegreeInCommon() {
      String first = binaryFirstSize();
      String second = binarySecondSize();
      int f = first.length() - 1;
      int s = second.length() - 1;
      int i = 0;
      while (f - i >= 0 && s - i >= 0) {
        if (first.charAt(f - i) != second.charAt(s - i))
          return false;
        i++;
      }
      return true;
    }
  }
if((first.charAt(f - i) == second.charAt(s - i)) &amp; (first.charAt(f - i) == '1'))
    return false;
else
    
    i++;
}
return true;
}

/**
 * getClass translates a given array of boolean into a Integer
 * @param classes The array to be translated
 * @return The integer that represents the given array
 * @see eqClass()
 */
public BigInteger getClass(boolean[] classes) {
    BigInteger res = new BigInteger("0");
    BigInteger dos = new BigInteger("2");
    for (int i = 0; i < classes.length; i++) {
        if (classes[i]) {
            BigInteger t = dos.pow(i);
            res = res.add(t);
        }
    }
    return res;
}

/**
 * eqClass returns the equivalence class in which the current
 * instance of MergeBinHeap belongs to. In this case,
 * the equivalence classes are given by all the
 * possible combinations of the following predicates:
 * isEmpty() over the first binomialHeap,
 * isEmpty() over the second binomialHeap,
 * size of the first binomialHeap is equal to the size of the
 * second binomialHeap, size of the first binomialHeap greater
 * than the size of the second binomialHeap,
 * first and second binomialHeaps have same degree
 * first's degree greater than second's degree
 * @return The equivalence class in which this instance belongs
 */
public BigInteger eqClass() {
    boolean[] classes = new boolean[7];
    classes[0] = first.isEmpty();
    classes[1] = second.isEmpty();
    classes[2] = first.getSize() == second.getSize();
    classes[3] = first.getSize() > second.getSize();
    if((classes[0]) &amp; (!classes[1])){
        classes[4] = false; // if first is empty and second is not
        //empty, then they do not have the same degree
        classes[5] = false; // if first is empty and second is not
        //empty, then first's degree not greater than second's
    }
    else if((!(classes[0]) &amp; (classes[1]))){
        classes[4] = false; // if second is empty and first is not
        //empty, then they do not have the same degree
        classes[5] = true; // if second is empty and first is not
        //empty, first's degree greater than second's
    }
    else {
        classes[4] = binaryFirstSize().length() == binarySecondSize() .length();
        classes[5] = binaryFirstSize().length() > binarySecondSize() .length();
    }
    classes[6] = NoDegreeInCommon();
    return getClass(classes);
}
/**
 * finMergeBinHeap provides a bound on the number of objects
 * to be used to generate instances of MergeBinHeap.
 * @param sizeF size of the first BinomialHeap.
 * @param sizeS size of the second BinomialHeap.
 * @return the object Korat needs for setting up the bounds
 * during the search.
 */

public static IFinInitiation finMergeBinHeap(int sizeF, int sizeS) {
    IFinInitiation f = FinInitiationFactory.create(MergeBinHeap.class);

    IObjSet entriesFirst = f.createObjectSet(BinomialHeap.class, false);
    entriesFirst.addClassDomain(f.createClassDomain(BinomialHeap.class, 1));
    IObjSet entriesSecond = f.createObjectSet(BinomialHeapS.class, false);
    entriesSecond.addClassDomain(f.createClassDomain(BinomialHeapS.class, 1));

    IClassDomain heapsDomainF = f.createClassDomain(BinomialHeapNode.class, sizeF);
    IObjSet heapsF = f.createObjectSet(BinomialHeapNode.class);
    heapsF.setNullAllowed(true);
    heapsF.addClassDomain(heapsDomainF);

    IClassDomain heapsDomainS = f.createClassDomain(BinomialHeapNodeS.class, sizeS);
    IObjSet heapsS = f.createObjectSet(BinomialHeapNodeS.class);
    heapsS.setNullAllowed(true);
    heapsS.addClassDomain(heapsDomainS);

    f.set("first", entriesFirst);
    f.set("second", entriesSecond);
    f.set(”BinomialHeap.size”, f.createIntSet(0, sizeF));
    f.set(”BinomialHeapS.size”, f.createIntSet(0, sizeS));
    f.set(”BinomialHeap.Nodes”, heapsF);
    f.set(”BinomialHeapS.Nodes”, heapsS);
    f.set(”BinomialHeapNode.class”, "parent", heapsF);
    f.set(”BinomialHeapNode.class”, "child", heapsF);
    f.set(”BinomialHeapNode.class”, ”key”, f.createIntSet(1, sizeF));
    f.set(”BinomialHeapNode.class”, ”degree”, f.createIntSet(0, sizeF));

    return f;
}
public class BinomialHeap implements Serializable {

    public static final long serialVersionUID = 1;

    public BinomialHeapNode Nodes;

    public int size;

    /**
     * Creates a new binomial heap that contains all the elements of
     * two binomial heaps. One of the original binomial heaps is
     * the object on which this method is called; the other is
     * specified by the parameter. The two original binomial heaps
     * should no longer be used after this operation.
     *
     * @param h2 The binomial heap to be merged with this one.
     * @return The new binomial heap that contains all the elements
     * of this binomial heap and <code>h2</code>.
     */
    public BinomialHeap union (BinomialHeap h2) {
        BinomialHeap h = new BinomialHeap();
        h.size = this.size + h2.size;
        h.Nodes = binomialHeapMerge(this, h2);
        this.Nodes = null; // no longer using the...
        h2.Nodes = null; // ...two input lists
        if (h.Nodes == null)
            return h;

        BinomialHeapNode prevX = null;
        BinomialHeapNode x = h.Nodes;
        BinomialHeapNode nextX = x.sibling;

        while (nextX != null) {
            if (x.degree != nextX.degree ||
                (nextX.sibling != null && nextX.sibling.degree == x.
                    degree)) {
                // Cases 1 and 2.
                prevX = x;
                x = nextX;
            } else {
                if (x.key < nextX.key) {
                    // Case 3.
                    x.sibling = nextX.sibling;
                    binomialLink(nextX, x);
                } else {
                    // Case 4.
                    if (prevX == null)
                        h.Nodes = nextX;
                    else
                        prevX.sibling = nextX;
                    binomialLink(x, nextX);
                    x = nextX;
                }
            }
        }
        return h;
    }
}
71 }  
72 nextX = x.sibling;  
73 }  
74 return h;  
75 }  
76 }  
77 }  
78 /**  
79 * Links one binomial tree to another.  
80 * @param y The root of one binomial tree.  
81 * @param z The root of another binomial tree; this root becomes  
82 * the parent of <code>y</code>.  
83 */  
84 private void binomialLink(BinomialHeapNode y, BinomialHeapNode z) {  
85     y.parent = z;  
86     y.sibling = z.child;  
87     z.child = y;  
88     z.degree++;  
89 }  
90  
91 /**  
92 * Merges the root lists of two binomial heaps together into a  
93 * single root list. The degrees in the merged root list appear  
94 * in monotonically increasing order.  
95 * @param h1 One binomial heap.  
96 * @param h2 The other binomial heap.  
97 * @return The head of the merged list.  
98 */  
99 private static BinomialHeapNode binomialHeapMerge(BinomialHeap h1,  
100     BinomialHeap h2) {  
101     // If either root list is empty, just return the other.  
102     if (h1.Nodes == null)  
103         return h2.Nodes;  
104     else if (h2.Nodes == null)  
105         return h1.Nodes;  
106     else {  
107         // Neither root list is empty. Scan through both, always  
108         // using the node whose degree is smallest of those not  
109         // yet taken.  
110         BinomialHeapNode head; // head of merged list  
111         BinomialHeapNode tail; // last node added to merged list  
112         BinomialHeapNode h1Next = h1.Nodes,  
113         h2Next = h2.Nodes; // next nodes to be examined in h1 and h2  
114         if (h1.Nodes.degree <= h2.Nodes.degree) {  
115             head = h1.Nodes;  
116             h1Next = h1Next.sibling;  
117         }  
118         else {  
119             head = h2.Nodes;  
120             h2Next = h2Next.sibling;  
121         }  
122         tail = head;  
123         // Go through both root lists until one of them is  
124         // exhausted.  
125         while (h1Next != null && h2Next != null) {  
126             if (h1Next.degree <= h2Next.degree) {  
127                 h1Next = h1Next.sibling;  
128             }  
129             else {  
130                 h2Next = h2Next.sibling;  
131             }  
132         }  
133     }  
134  
135 }
tail = tail.sibling;
}

// The above loop ended because exactly one of the root
// lists was exhausted. Splice the remainder of whichever
// root list was not exhausted onto the list we're
// constructing.
if (h1Next != null)
tail.sibling = h1Next;
else
tail.sibling = h2Next;

return head; // all done!

public int getSize() {
    return size;
}

public String toString() {
    if (Nodes == null)
        return size + "()";
    else
        return size + "" + Nodes.toString();
}

/**
   * Checks whether the current binomial heap is empty.
   * @return true iff the current binomial heap is empty,
   * false otherwise.
   */
public boolean isEmpty() {
    return size == 0;
}

public int degree() {
    return Nodes.degree;
}

/**
   * checks that the current binomial heap satisfies its
   * representation invariants, which means checking that
   * list of trees has no cycles, the total size is consistent,
   * the degrees of all trees are binomial and the keys are
   * heapified.
   * @return true iff this binomialHeap satisfies the corresponding
   * representation invariants.
   */
public boolean repOK() {
    if (size == 0)
        return (Nodes == null);
    if (Nodes == null)
        return false;
    java.util.Set<NodeWrapper> visited = new java.util.HashSet<NodeWrapper>();
    for (BinomialHeapNode current = Nodes; current != null; current = current.sibling) {
        /** checks that the list has no cycles */
        if (!visited.add(new NodeWrapper(current)))
            return false;
        if (!current.isTree(visited, null))
            return false;
}
/∗∗ checks that the total size is consistent ∗/
if (visited.size() != size)
    return false;
/∗∗ checks that the degrees of all trees are binomial ∗/
if (!checkDegrees())
    return false;
/∗∗ checks that keys are heapified ∗/
if (!checkHeapified())
    return false;
return true;

boolean checkDegrees() {
    int degree = size;
    int rightDegree = 0;
    for (BinomialHeapNode current = Nodes; current != null; current = current.sibling) {
        if (degree == 0)
            return false;
        while ((degree & 1) == 0) {
            rightDegree++;
            degree /= 2;
        }
        if (current.degree != rightDegree)
            return false;
        if (!current.checkDegree(rightDegree))
            return false;
        rightDegree++;;
        degree /= 2;
    }
    return (degree == 0);
}
boolean checkHeapified() {
    for (BinomialHeapNode current = Nodes; current != null; current = current.sibling) {
        if (!current.isHeapified())
            return false;
    }
    return true;
}

package koratPlus.mergebinheap;
import java.util.HashSet;
import java.io.*;
/**
 * This class defines BinomialHeap's nodes
 * This class needs to implement the Serializable interface
 * since it is required for the korat "serialize" option.
 * @author */
public class BinomialHeapNode implements Serializable{
    public static final long serialVersionUID = 1;
    /**
     * element in current node
     */
    public int key;
    /**
     * depth of the binomial tree having the current
24 * node as its root
25 */
26 public int degree;
27
28 /**
29 * pointer to the parent of the current node
30 */
31 public BinomialHeapNode parent;
32
33 /**
34 * pointer to the next binomial tree in the list
35 */
36 public BinomialHeapNode sibling;
37
38 /**
39 * pointer to the first child of the current node
40 */
41 public BinomialHeapNode child;
42
43 public int getSize() {
44    return (1 + ((child == null) ? 0 : child.getSize()) + ((sibling == null) ? 0 : sibling.getSize()));
45 }
46
47 public String toString() {
48    BinomialHeapNode temp = this;
49    String ret = "";
50    while (temp != null) {
51        ret += "(";
52        if (temp.parent == null)
53            ret += "Parent: null";
54        else
55            ret += "Parent: " + temp.parent.key;
56        ret += " Degree: " + temp.degree + " Key: " + temp.key + "");
57        if (temp.child != null)
58            ret += temp.child.toString();
59        temp = temp.sibling;
60    }
61    if (parent == null)
62        ret += " ";
63    return ret;
64 }
65
66 private boolean repCheckWithRepetitions(int key, int degree, 
67 Object parent, HashSet<BinomialHeapNode> nodesSet) {
68    BinomialHeapNode temp = this;
69    int rightDegree = 0;
70    if (parent == null) {
71        while ((degree & 1) == 0) {
72            rightDegree += 1;
73            degree /= 2;
74        }
75        degree /= 2;
76    } else
77        rightDegree = degree;
78    while (temp != null) {
79        if ((temp.degree != rightDegree) || (temp.parent != parent) 
80            || (temp.key < key) || (nodesSet.contains(temp)))
81            return false;
82        else
83            nodesSet.add(temp);
84        if (temp.child == null) {
85            /*
86             */
87            /*
88             */
89        } else
90            /*
91            */
92    }
temp = temp.sibling;
if (parent_ == null) {
  if (degree_ == 0)
    return (temp == null);
  while ((degree_ & 1) == 0) {
    rightDegree += 1;
    degree_ /= 2;
  }
  degree_ /= 2;
  rightDegree++;
} else
  rightDegree--;
} else {
  boolean b = temp.child.repCheckWithRepetitions(
    temp.key, temp.degree - 1, temp, nodesSet);
  if (!b)
    return false;
  else {
    temp = temp.sibling;
    if (parent_ == null) {
      if (degree_ == 0)
        return (temp == null);
      while ((degree_ & 1) == 0) {
        rightDegree += 1;
        degree_ /= 2;
      }
      degree_ /= 2;
      rightDegree++;
    } else
      rightDegree--;
  }
}
return true;

private boolean repCheckWithoutRepetitions(int key_,
  HashSet<Integer> keysSet, int degree_, // equal keys not allowed
  Object parent_, HashSet<BinomialHeapNode> nodesSet) {
  BinomialHeapNode temp = this;
  int rightDegree = 0;
  if (parent_ == null) {
    while ((degree_ & 1) == 0) {
      rightDegree += 1;
      degree_ /= 2;
    }
    degree_ /= 2;
  } else
    rightDegree = degree_;
  while (temp != null) {
    if (((temp.degree != rightDegree) || (temp.parent != parent_))
      || (temp.key <= key_) || (nodesSet.contains(temp))
      || (keysSet.contains(new Integer(temp.key))))
      return false;
  } else {
    nodesSet.add(temp);
    keysSet.add(new Integer(temp.key));
    if (temp.child == null) {
      temp = temp.sibling;
      if (parent_ == null) {

if (degree_ == 0)
    return (temp == null);
while (((degree_ & 1) == 0) {
    rightDegree += 1;
    degree_ /= 2;
} 
degree_ /= 2;
    rightDegree++;
} else
    rightDegree --;
} else {
    boolean b = temp.child.repCheckWithoutRepetitions( 
        temp.key, keysSet, temp.degree - 1, temp,
        nodesSet);
    if (!b)
        return false;
    else {
        temp = temp.sibling;
        if (parent_ == null) {
            if (degree_ == 0)
                return (temp == null);
            while (((degree_ & 1) == 0) {
                rightDegree += 1;
                degree_ /= 2;
            } 
            degree_ /= 2;
            rightDegree++;
        } else
            rightDegree --;
    }
}
return true;

boolean checkDegree(int degree) {
    for (BinomialHeapNode current = this.child; current != null; 
        current = current.sibling) {
        degree --;
        if (current.degree != degree)
            return false;
        if (!current.checkDegree(degree))
            return false;
    }
    return (degree == 0);
}

boolean isHeapified() {
    for (BinomialHeapNode current = this.child; current != null; 
        current = current.sibling) {
        if (!key <= current.key)
            return false;
        if (!current.isHeapified())
            return false;
    }
    return true;
}

boolean isTree(java.util.Set<NodeWrapper> visited,
        BinomialHeapNode parent) {
    if (this.parent != parent)
        return false;
    for (BinomialHeapNode current = this.child; current != null; 
        current = current.sibling) {
        if (!visited.add(new NodeWrapper(current)))
            return false;
return false;
if (!current.isTree(visisted, this))
return false;
}
return true;

public boolean repOk(int size) {
    /*replace ‘repCheckWithoutRepetitions’ with
     ‘repCheckWithRepetitions’ if you don’t want to allow equal keys
     */
    return repCheckWithRepetitions(0, size, null,
    new HashSet<BinomialHeapNodeS>());
}
public int degree()
{
    return Nodes.degree;
}

/**
 * checks that the current binomial heap satisfies its
 * representation invariant, which means checking that
 * list of trees has no cycles, the total size is consistent,
 * the degrees of all trees are binomial and the keys are
 * heapified.
 * @return true iff this binomialHeapS satisfies the
 * corresponding representation invariant.
 */

public boolean repOK()
{
    if (size == 0)
        return (Nodes == null);
    if (Nodes == null)
        return false;
    java.util.Set<NodeWrapperS> visited = new java.util.HashSet();
    for (BinomialHeapNodeS current = Nodes; current != null; current = current.sibling)
    {
        /** checks that the list has no cycles */
        if (!visited.add(new NodeWrapperS(current)))
            return false;
        if (!current.isTree(visited, null))
            return false;
    }
    /** checks that the total size is consistent */
    if (visited.size() != size)
        return false;
    /** checks that the degrees of all trees are binomial */
    if (!checkDegrees())
        return false;
    /** checks that keys are heapified */
    if (!checkHeapified())
        return false;
    return true;
}

boolean checkDegrees()
{
    int degree_ = size;
    int rightDegree = 0;
    for (BinomialHeapNodeS current = Nodes; current != null; current = current.sibling)
    {
        if (degree_ == 0)
            return false;
        while (((degree_ & 1) == 0) {
            rightDegree++;
            degree_ /= 2;
        }
        if (current.degree != rightDegree)
            return false;
        if (!current.checkDegree(rightDegree))
            return false;
        rightDegree++;
        degree_ /= 2;
    }
    return (degree_ == 0);
}

boolean checkHeapified()
{
    for (BinomialHeapNodeS current = Nodes; current != null; current = current.sibling)
    {
        if (!current.isHeapified())
            return false;
    }
    return true;
}
package koratPlus.mergebinheap;

import java.util.HashSet;
import java.io.

/**
 * This class defines BinomialHeapS’s nodes.
 * This class needs to implement the Serializable interface since it is required for the korat “serialize” option.
 * @author
 */

public class BinomialHeapNodeS implements Serializable{

    public static final long serialVersionUID = 1;

    /**
     * element in current node
     */
    public int key;

    /**
     * depth of the binomial tree having the current node as its root
     */
    public int degree;

    /**
     * pointer to the parent of the current node
     */
    public BinomialHeapNodeS parent;

    /**
     * pointer to the next binomial tree in the list
     */
    public BinomialHeapNodeS sibling;

    /**
     * pointer to the first child of the current node
     */
    public BinomialHeapNodeS child;

    private BinomialHeapNode search(java.util.ArrayList<Wrapper> list, BinomialHeapNodeS s){
        for(int i = 0; i<list.size(); i++){
            Wrapper w = list.get(i);
            if(w.bhn==s){ //quiero comparar referencias
                return w.bhn;
            }
        }
        return null;
    }

    public BinomialHeapNode conversion(java.util.ArrayList<Wrapper> list){
        BinomialHeapNode n = new BinomialHeapNode();
        n.key = this.key;
        n.degree = this.degree;
        Wrapper w = new Wrapper(this,n);
    }
}
```java
list.add(w);
if(this.parent != null) {
    n.parent = search(list, this);
}
if(this.sibling != null)
    n.sibling = this.sibling.conversion(list);
if(this.child != null)
    n.child = this.child.conversion(list);
return n;
}

public int getSize() {
    return (1 + ((child == null) ? 0 : child.getSize()) + ((sibling == null) ? 0 : sibling.getSize()));
}

public String toString() {
    BinomialHeapNodeS temp = this;
    String ret = "";
    while (temp != null) {
        ret += "(";
        if (temp.parent == null)
            ret += "Parent: null";
        else
            ret += "Parent: " + temp.parent.key;
        ret += " Degree: " + temp.degree + " Key: " + temp.key + ") ";
        if (temp.child != null)
            ret += temp.child.toString();
        temp = temp.sibling;
    }
    if (parent == null)
        ret += " ";
    return ret;
}

private boolean repCheckWithRepetitions(int key, int degree, Object parent, HashSet<BinomialHeapNodeS> nodesSet) {
    BinomialHeapNodeS temp = this;
    int rightDegree = 0;
    if (parent == null) {
        while ((degree & 1) == 0) {
            rightDegree += 1;
            degree /= 2;
        }
    } else
        rightDegree = degree;
    while (temp != null) {
        if (((temp.degree == rightDegree) || (temp.parent != parent))
            || (temp.key < key) || (nodesSet.contains(temp)))
            return false;
        else
            nodesSet.add(temp);
        if (temp.child == null) {
            temp = temp.sibling;
        } else {
            if (parent == null) {
                if (degree == 0)
                    return (temp == null);
                while ((degree & 1) == 0) {
```
rightDegree += 1;
degree_ /= 2;
} else {
    degree_ /= 2;
    rightDegree++;
} else
    rightDegree--;
else {
    boolean b = temp.child.repCheckWithRepetitions(temp.key, temp.degree - 1, temp, nodesSet);
    if (!b)
        return false;
    else {
        leftDegree = degree_;
        while (leftDegree != 0) {
            degree_ /= 2;
            rightDegree += 1;
        }
        return true;
    }
}
private boolean repCheckWithoutRepetitions(int key_,
        HashSet<Integer> keysSet , int degree_, /**equal keys not allowed*/
        Object parent_, HashSet<BinomialHeapNodeS> nodesSet) {
    BinomialHeapNodeS temp = this;
    int rightDegree = 0;
    if (parent_ == null) {
        if (degree_ == 0)
            return (temp == null);
        while ((degree_ & 1) == 0) {
            rightDegree += 1;
            degree_ /= 2;
        }
        degree_ /= 2;
    } else
        rightDegree = degree_;
    while (temp != null) {
        if (((temp.degree != rightDegree) || (temp.parent != parent_)
            || (temp.key <= key_) || (nodesSet.contains(temp))
            || (keysSet.contains(new Integer(temp.key))))
            return false;
        else {
            nodesSet.add(temp);
            keysSet.add(new Integer(temp.key));
            if (temp.child == null) {
                temp = temp.sibling;
                if (parent_ == null) {
                    if (degree_ == 0)
                        return (temp == null);
                    while ((degree_ & 1) == 0) {
                        rightDegree += 1;
                        degree_ /= 2;
                    }
                }
            }
        }
    }
192  
193  degree_ /= 2;
194  rightDegree++; 
195 } else
196  rightDegree--; 
197 } else {
198  boolean b = temp.child.repCheckWithoutRepetitions(
199  temp.key, keysSet, temp.degree - 1, temp, 
200  nodesSet); 
201  if (!b)
202  return false; 
203  else 
204  {
205    temp = temp.sibling; 
206    if (parent_ == null) {
207      if (degree_ == 0)
208        return (temp == null); 
209      while (((degree_ & 1) == 0) {
210        degree_ /= 2; 
211      } 
212      degree_ /= 2; 
213      rightDegree++; 
214    } else
215      rightDegree--; 
216    } 
217  }
218  
219  return true; 
220 } 
221 
222 boolean checkDegree(int degree) {
223  for (BinomialHeapNodeS current = this.child; current != null; 
224    current = current.sibling) {
225    degree--; 
226    if (current.degree != degree)
227      return false; 
228    if (!current.checkDegree(degree))
229      return false; 
230  } 
231  return (degree == 0); 
232 } 
233 
234 boolean isHeapified() {
235  for (BinomialHeapNodeS current = this.child; current != null; 
236    current = current.sibling) {
237    if (!key <= current.key))
238      return false; 
239    if (!current.isHeapified())
240      return false; 
241  } 
242  return true; 
243 } 
244 
245 boolean isTree(java.util.Set<NodeWrapperS> visited,
246  BinomialHeapNodeS parent) {
247  if (this.parent != parent)
248    return false; 
249  for (BinomialHeapNodeS current = this.child; current != null; 
250    current = current.sibling) {
251    if (!visited.add(new NodeWrapperS(current)))
252      return false; 
253    if (!current.isTree(visited, this))
254      return false; 
255  } 
256  return true; 
257 }
public boolean repOk(int size) {
    /** replace 'repCheckWithoutRepetitions' with 'repCheckWithRepetitions' if you don't want to allow equal keys*/
    return repCheckWithRepetitions(0, size, null, new HashSet<BinomialHeapNodeS>());
}

Directed Graphs

package koratPlus.AltGraph;

import korat.finitization.IArraySet;
import korat.finitization.IFinitization;
import korat.finitization.IIntSet;
import korat.finitization.IObjSet;
import korat.finitization.impl.FinitizationFactory;
import java.math.BigInteger;
import java.util.Set;
import java.util.Iterator;

/**
 * Class AltGraph defines Directed Graphs using adjacent lists
 * @author Nazareno M. Aguirre, Valeria Bengolea.
 */
public class AltGraph {
    Vertex[] vertices;
    int size;

    /**
     * RepOk checks whether the directed graph satisfies its
     * representation invariant.
     * @return true iff the current graph satisfies its
     * representation invariant
     */
    public boolean repOK() {
        if (vertices==null)
            return false;
        if (vertices.length != size)
            return false;
        /**check if each adj. list satisfies the
         * list's representation invariant.
         */
        for (int i=0; i<vertices.length; i++) {
            if (vertices[i] == null)
                return false;
            if (!vertices[i].repOK())
                return false;
        }
        /** check if the label of each vertex is equal to
         * the corresponding position in the array.
         */
        for (int i=0; i<vertices.length; i++) {
            if (vertices[i].label!=i)
                return false;
        }
        return true;
    }

    /**finAltGraph provides a bound on the number of objects to be
public static IInitialization finAltGraph(int numVertices) {
    IInitialization f = FinitizationFactory.create(AltGraph.class);
    IIntSet arrLen = f.createIntSet(numVertices);
    IObjSet entries = f.createObjSet(Entry.class, true);
    //create enough entries for all adj. lists
    entries.addClassDomain(f.createClassDomain(Entry.class, numVertices * numVertices));
    IObjSet vertices = f.createObjSet(Vertex.class, true);
    vertices.addClassDomain(f.createClassDomain(Vertex.class, numVertices));
    IObjSet lists = f.createObjSet(SinglyLinkedList.class, numVertices, false);
    IArraySet verticesArray = f.createArraySet(Vertex[].class, arrLen, vertices, numVertices);
    f.set(Entry.class, "label", f.createIntSet(0, numVertices - 1));
    f.set(Entry.class, "adjacent", lists);
    f.set(SinglyLinkedList.class, "header", entries);
    f.set(Entry.class, "next", entries);
    f.set("vertices", verticesArray);
    f.set("size", f.createIntSet(numVertices));
    return f;
}

/**
 * Returns all the reachable nodes from a given node following a
 * depth first strategy.
 * @param n The starting node
 * @param visited A set used for keeping track the visited nodes.
 * @return The set containing all the reachable nodes from 'n'
 */
public Set<Integer> dfs(Integer n, Set<Integer> visited) {
    SinglyLinkedList s = vertices[n].adjacent;
    for (Entry curr=s.header; curr!=null; curr=curr.next) {
        if(visited.add(curr.element.label)){
            dfs(curr.element.label, visited);
        }
    }
    return visited;
}

/**
 * Checks whether or not a node belongs to a given set.
 * @param s Set to be tested.
 * @param n The element whose presence in this set is to be tested.
 * @return true iff 'n' is in the set 's', false otherwise
 */
public boolean isIn(Set<Integer> s, Integer n){
    Iterator it = s.iterator();
    while(it.hasNext()){  
        if(n.intValue() == ((Integer)it.next()).intValue())
            return true;
    }
}
public boolean all(Set<Integer> s) {
    int i = 0;
    while (i < vertices.length) {
        if (isIn(s, i) == false) {
            return false;
        }
        i++;
    }
    return true;
}

/**
 * Checks whether or not a set contains all the nodes of the current graph.
 * @param s The set to be tested.
 * @return true is returned iff all nodes in the graph are in s, false otherwise.
 */
public boolean stronglyConnected() {
    for (int i = 0; i < vertices.length; i++) {
        Set<Integer> visited = new java.util.HashSet<Integer>();
        visited = dfs(i, visited);
        if (!all(visited)) {
            return false;
        }
    }
    return true;
}

/**
 * Checks whether or not the current graph has one loop.
 * @return True iff a loop is found, false otherwise.
 */
public boolean cyclic() {
    for (int i = 0; i < vertices.length; i++) {
        Set<Integer> visited = new java.util.HashSet<Integer>();
        visited = dfs(i, visited);
        if (isIn(visited, i))
            return true;
    }
    return false;
}

/**
 * Checks whether or not the current graph is empty.
 * @param s The set to be tested.
 * @return true is returned iff all nodes in the graph are in s, false otherwise.
 */
public boolean isEmpty() {
    for (int i = 0; i < vertices.length; i++) {
        if (vertices[i].adjacent.header != null) return false;
    }
    return true;
}
/*
 * @return True is returned iff the graph is dense, false otherwise
 */

public boolean isDense() {
    int numArcs = 0;
    for (int i = 0; i < vertices.length; i++) {
        for (Entry current = vertices[i].adjacent.header; current != null; current = current.next) {
            numArcs++;
            if (numArcs > ((size * (size - 1)) / 2)) return true;
        }
    }
    return false;
}

/**
 * Checks whether or not the current graph is complete
 * @return True iff the current graph is complete, false otherwise
 */

public boolean complete() {
    for (int i = 0; i < vertices.length; i++) {
        int numArcs = 0;
        for (Entry current = vertices[i].adjacent.header; current != null; current = current.next) {
            numArcs++;
        }
        if (numArcs != size) return false;
    }
    return true;
}

/**
 * getClass translates a given array of boolean into a Integer
 * @param classes The array to be translated
 * @return The integer that represents the given array
 * @see eqClass()
 */

public BigInteger getClass(boolean[] classes) {
    BigInteger res = new BigInteger("0");
    BigInteger dos = new BigInteger("2");
    for (int i = 0; i < classes.length; i++) {
        if (classes[i]) {
            BigInteger t = dos.pow(i);
            res = res.add(t);
        }
    }
    return res;
}

/**
 * eqClass given a graph it returns the equivalence class
 * in which this graph belongs to. In this case, the
 * equivalence classes are given by all the possible combinations
 * of predicates isEmpty(), complete() and isDense().
 * @return The equivalence class in which the current graph belongs
 */

public BigInteger eqClass() {
    boolean[] classes = new boolean[3];
    if (isEmpty()) {
        classes[0] = true;
        classes[1] = false; // if empty, then it's not dense
        classes[2] = false; // if empty, then it's not complete
    } else {
        classes[1] = isDense();
        if (classes[1]) {
            classes[2] = complete();
        } else {
            classes[2] = complete();
        }
    }
}
classes[2] = false; // if not dense, not complete
}
return this.getClass(classes);
}

package koratPlus.AltGraph;
import java.util.Set;
import java.math.BigInteger;
import korat.finitization.IClassDomain;
import korat.finitization.IFinitization;
import korat.finitization.INtSet;
import korat.finitization.IObjSet;
import korat.finitization.impl.FinitizationFactory;
/**
 * Class vertex defines vertex of the AltGraph (Directed graph).
 * @author Nazareno M. Aguirre
 */
public class Vertex {
    public int label;
    public SinglyLinkedList adjacent;

    /**
     * check if adjacent satisfies the list's representation invariant.
     * @return True iff the current vertex satisfies its representation invariant.
     */
    public boolean repOK() {
        if (adjacent==null) return false;
        return (adjacent.repOK());
    }
}

package koratPlus.AltGraph;
import java.util.Set;
import java.math.BigInteger;
import korat.finitization.IClassDomain;
import korat.finitization.IFinitization;
import korat.finitization.INtSet;
import korat.finitization.IObjSet;
import korat.finitization.impl.FinitizationFactory;
/**
 * Class SinglyLinkedList defines Singly linked List
 * @author
 */
public class SinglyLinkedList {
    public Entry header;

    /**
     * RepOk checks whether the singlyLinkedList satisfies
     */
    public boolean repOK() {
public boolean repOK() {
    return repOkCommon();
}

public boolean repOkCommon() {
    Set<Entry> visited = new java.util.HashSet<Entry>();
    Entry current = header;
    while (current!=null) {
        if (!visited.add(current))
            return false;
        if (current.element == null)
            return false;
        current = current.next;
    }
    return true;
}

public boolean repOkSorted(int maxVertex) {
    if (!repOkCommon())
        return false;
    for (Entry current = header; current != null; current = current.next) {
        if (current.next!=null) {
            if (current.element.label>=current.next.element.label)
                return false;
        }
    }
    return true;
}

public String toString() {
    String res = " ( ";
    if (header != null) {
        Entry cur = header.next;
        while (cur != null && cur != header) {
            res += cur.toString();
            cur = cur.next;
        }
    }
    return res + ")";
}

package koratPlus.AltGraph;

/**
 * Entries of the SinglyLinkedList class.
 * @author */

public class Entry {
    public Vertex element;
    public Entry next;

    public String toString() {
        return "[" + (element != null ? element.toString() : "null") + ""]";
    }
}

Weighted Directed Graphs
package koratPlus.LabelledAltGraph;

import korat.finitization.IArraySet;
import korat.finitization.IFinitization;
import korat.finitization.IIntSet;
import korat.finitization.IObjSet;
import korat.finitization.impl.FinitizationFactory;

import java.math.BigInteger;
import java.util.Set;
import java.util.Iterator;

/**
   * Class that implements Weighted Directed Graphs using adjacent lists.
   * In this implementation each entry in the adjacent list of a vertex
   * has a corresponding weight.
   * @author Nazareno M. Aguirre, Valeria Bengolea.
   */
public class LabelledAltGraph {

    Vertex[] vertices;
    int size;

    /**
     * RepOk checks whether the weighted directed graph satisfies its
     * representation invariant.
     * @return True iff the current graph satisfies its representation
     * invariant.
     */
    public boolean repOK() {
        if (vertices==null)
            return false;
        if (vertices.length != size)
            return false;
        for (int i=0; i<vertices.length; i++) {
            if (vertices[i] == null)
                return false;
            if (!vertices[i].repOK())
                return false;
        }
        /** check if the label of each vertex is equal to
         * the corresponding position in the array.
         */
        for (int i=0; i<vertices.length; i++) {
            if (vertices[i].label!=i)
                return false;
        }
        return true;
    }

    /**
     * finLabelledAltGraph provides a bound on the number of objects to
     * be used to generate instances of LabelledAltGraph.
     * @param numVertices the number of vertices that any instance of
     * LabelledAltGraph may have.
     * @param minNumLabels minimum value that the vertex’s weight may
     * take.
     * @param maxNumLabels maximum value that the vertex’s weight may
     * take.
     * @return the object Korat needs for setting up the bounds during
     * the search.
     */
    public static IFinitization finLabelledAltGraph(int numVertices, int minNumLabels, int maxNumLabels) {
IFinitalization f = FinitizationFactory.create(LabelledAltGraph.class);

IIntSet arrLen = f.createIntSet(numVertices);

IObjSet entries = f.createObjSet(Entry.class, true);
entries.addClassDomain(f.createClassDomain(Entry.class, numVertices*numVertices)); // create enough entries for all adj. lists

IObjSet vertices = f.createClassDomain(Vertex.class, numVertices);

IObjSet lists = f.createClassSet(SortedSinglyLinkedList.class, numVertices, false);

IArraySet verticesArray = f.createArraySet(Vertex[].class, arrLen, vertices, numVertices);

f.set(Vertex.class, "label", f.createIntSet(0, numVertices-1));

f.set(Vertex.class, "adjacent", lists);

f.set(SortedSinglyLinkedList.class, "header", entries);

f.set(Entry.class, "element", vertices);

f.set(Entry.class, "transLabel", f.createClassDomain(minNumLabels, maxNumLabels));

f.set(Entry.class, "next", entries);

f.set("vertices", verticesArray);

f.set("size", f.createIntSet(numVertices));

return f;

/**
 * Returns all the reachable nodes from a given node following a depth first strategy.
 * @param n The starting node.
 * @param visited A set used for keeping track the visited nodes.
 * @return The set containing all the reachable nodes from 'n'
 */

public Set<Integer> dfs(Integer n, Set<Integer> visited) {

SortedSinglyLinkedList s = vertices[n].adjacent;
for (Entry curr=s.header; curr!=null; curr=curr.next) {
    if (visited.add(curr.element.label)){
        dfs(curr.element.label, visited);
    }
}
return visited;
}

/**
 * Checks whether or not a node belongs to a given set.
 * @param s Set to be tested.
 * @param n The element whose presence in this set is to be tested.
 * @return true iff 'n' is in the set 's', false otherwise.
 */

public boolean isIn(Set<Integer> s, Integer n){

Iterator it = s.iterator();
while(it.hasNext()){
    if (n.intValue() == ((Integer)it.next()).intValue())
        return true;
}
return false;
}
/**
   * Checks whether or not a set contains all the nodes of the current graph.
   * @param s
   * @return true is returned iff all nodes in the graph are in s, false otherwise.
   */
   public boolean all(Set<Integer> s) {
       int i = 0;
       while (i < vertices.length) {
           if (isIn(s, i) == false) {
               return false;
           }
           i++;
       }
       return true;
   }

   /**
   * Checks whether or not the current graph is strongly connected.
   * @return True is returned iff the current graph is strongly connected, false otherwise.
   */
   public boolean stronglyConnected() {
       for (int i = 0; i < vertices.length; i++) {
           Set<Integer> visited = new java.util.HashSet<Integer>();
           visited = dfs(i, visited);
           visited.add(i);
           if (!all(visited)) {
               return false;
           }
       }
       return true;
   }

   /**
   * Checks whether or not the current graph has at least one loop.
   * @return True iff a loop is found, false otherwise.
   */
   public boolean cyclic() {
       for (int i = 0; i < vertices.length; i++) {
           Set<Integer> visited = new java.util.HashSet<Integer>();
           visited = dfs(i, visited);
           if (isIn(visited, i)) {
               return true;
           }
       }
       return false;
   }

   /**
   * Checks whether or not the current graph is empty.
   * @return True iff the current graph is empty, false otherwise.
   */
   public boolean isEmpty() {
       for (int i = 0; i < vertices.length; i++) {
           if (vertices[i].adjacent.header != null) return false;
       }
       return true;
   }
public boolean negativeWeight() {
    for (int i = 0; i < vertices.length; i++) {
        if (vertices[i].negativeWeight())
            return true;
    }
    return false;
}

public boolean complete() {
    for (int i = 0; i < vertices.length; i++) {
        int numArcs = 0;
        for (Entry current = vertices[i].adjacent.header; current != null; current = current.next) {
            numArcs++;
            if (numArcs != size)
                return false;
        }
        return true;
    }
}

public boolean isDense() {
    int numArcs = 0;
    for (int i = 0; i < vertices.length; i++) {
        for (Entry current = vertices[i].adjacent.header; current != null; current = current.next) {
            numArcs++;
            if (numArcs > (size * (size - 1)) / 2)
                return true;
        }
    }
    return false;
}

getClass translates a given array of boolean into a Integer.

public BigInteger getClass(boolean[] classes) {
    BigInteger res = new BigInteger("0");
    BigInteger dos = new BigInteger("2");
    for (int i = 0; i < classes.length; i++) {
        if (classes[i]) {
            BigInteger t = dos.pow(i);
            res = res.add(t);
        }
    }
    return res;
public BigInteger eqClass()
{
    boolean[] classes = new boolean[6];
    classes[0] = isEmpty();
    if (classes[0]) {
        classes[1] = false; // if empty, then it's not dense
        classes[2] = false; // if empty, not cyclic
        classes[3] = false; // if empty, not strongly connected
        classes[4] = false; // if empty, no negative weights
        classes[5] = true; // if empty, not complete
    } else {
        classes[1] = isDense();
        classes[2] = cyclic();
        classes[3] = stronglyConnected();
        classes[4] = negativeWeight();
        if (classes[1])
            classes[5] = complete();
        else
            classes[5] = false; // if not dense, then not complete
    }
    return this.getClass(classes);
}
if (curr.transLabel < 0) {
    return true;
}
return false;

package koratPlus.LabelledAltGraph;

import java.util.Set;

/**
 * Class SortedSinglyLinkedList defines Sorted Singly linked List
 * @author
 */

public class SortedSinglyLinkedList {
    public Entry header;

    /**
     * RepOk checks whether the sorted singlyLinkedList satisfies its
     * representation invariant (acyclicity and sortedness).
     * @return True iff the current list satisfies its representation
     * invariant, false otherwise.
     */
    public boolean repOk() {
        return repOkSorted();
    }

    public int length() {
        Entry current = header;
        int count = 0;
        while (current != null) {
            count++;
        }
        return count;
    }

    public boolean repOkCommon() {
        Set<Entry> visited = new java.util.HashSet<Entry>();
        Entry current = header;
        while (current != null) {
            if (!visited.add(current))
                return false;
            if (current.element == null)
                return false;
            current = current.next;
        }
        return true;
    }

    public boolean repOkSorted() {
        if (!repOkCommon()) return false;
        for (Entry current = header; current != null; current = current.next) {
            if (current.next != null) {
                if (current.element.label >= current.next.element.label)
                    return false;
            }
        }
        return true;
    }
}
public String toString() {
    String res = "(";
    if (header != null) {
        Entry cur = header.next;
        while (cur != null && cur != header) {
            res += cur.toString();
            cur = cur.next;
        }
        return res + ")";
    }
}

package koratPlus.LabelledAltGraph;

/**
 * Entries of the SortedSinglyLinkedList class.
 */
public class Entry {
    public Vertex element;
    public int transLabel;
    public Entry next;
    public String toString() {
        return "[" + (element != null ? element.toString() : "null") + "]";
    }
}

Search Trees (Delete)

/**
 * Class that defines SearchTrees. In this class a new integer attribute ('param') was added since, in this case, what we
 * want to test is the search method (black box coverage) and
 * we want korat to generate all valid entries to test the
 * search routine on searchTree, which are pairs of SearchTrees
 * and integers.
 * This class needs to implement the Serializable interface
 * since it is required for the korat "serialize" option.
 * This class needs to implement the Serializable interface since it
 * is required for the korat "serialize" option.
 */
package koratPlusSearchTreeBB;

import java.util.HashSet;
import java.util.LinkedList;
import java.util.Set;
import java.math.BigInteger;
import korat.finitization.IClassDomain;
import korat.finitization.IFinitization;
import korat.finitization.IIntSet;
import korat.finitization.IObjSet;
import korat.finitization.impl.FinitizationFactory;
import java.io.*;
```java
public class SearchTreeBB implements Serializable {

    public static final long serialVersionUID = 1;

    /*
    * Parameter to be searched. This new attribute is
    * necessary since we want korat to generate
    * all valid entries to test the search routine on SearchTree,
    * which are pairs of SearchTrees and integers.
    */
    public Integer param = new Integer(0);

    /*
    * root node
    */
    public Node root;

    /*
    * number of nodes in the tree
    */
    public int size;

    public SearchTreeBB(Node r) {
        root = r;
    }

    /*
    * Checks that the current SearchTree satisfies its
    * representation invariant, which means checking that
    * it is a well built tree and the data is ordered.
    * @return True iff this searchTree satisfies the
    * corresponding representation invariant.
    */
    public boolean repOK() {
        if (param == null)
            return false;
        if (root == null)
            return size == 0;
        if (!isAcyclic())
            return false;
        if (numNodes(root) != size)
            return false;
        if (!isOrdered(root))
            return false;
        return true;
    }

    /*
    * Checks that the tree has no cycle
    * @return true iff the tree has no cycle
    */
    private boolean isAcyclic() {
        Set visited = new HashSet();
        visited.add(root);
        LinkedList workList = new LinkedList();
        workList.add(root);
        while (!workList.isEmpty()) {
            Node current = (Node) workList.removeFirst();
            if (current.left != null) {
                if (!visited.add(current.left))
                    return false;
            }
            if (current.right != null) {
                if (!visited.add(current.right))
                    return false;
            }
            if (current != root) {
                workList.addLast(current.right);
                workList.addLast(current.left);
            }
        }
        return true;
    }
```
private int numNodes(Node n) {
    if (n == null)
        return 0;
    return 1 + numNodes(n.left) + numNodes(n.right);
}

private int compare(Object k1, Object k2) {
    return ((Comparable) k1).compareTo(k2);
}

private boolean isOrdered(Node e, Object min, Object max) {
    if (e.info.intValue() == -1)
        return false;
    if ((min != null) && (compare(e.info.intValue(), min) <= 0))
        return false;
    if (e.left != null)
        if (!isOrdered(e.left, min, e.info.intValue()))
            return false;
    if (e.right != null)
        if (!isOrdered(e.right, e.info.intValue(), max))
            return false;
    return true;
}

public static IFinItiZation finSearchTreeBB(int numNodes) throws Exception {
    return finSearchTreeBB(numNodes, numNodes, numNodes, numNodes);
}

/**
 * finSearchTreeBB provides a bound on the number of objects to be
 * used to generate instances of SearchTreeBB.
 * @param numNodes number of entries to be used to generate instance
 * of SearchTreeBB.
 * @return the object Korat needs for setting up the bounds during
 * the search.
 * @throws Exception
 */

private int workList.add(current.left);
}
if (current.right != null) {
    if (!visited.add(current.right))
        return false;
    workList.add(current.right);
}
return true;
}
public static IInitialization finSearchTreeBB(int numNodes, int minSize, int maxSize, int numKeys) throws Exception {
    IInitialization f = InitializationFactory.create(SearchTreeBB.class);
    IObjSet nodes = f.createObjSet(Node.class, numNodes);
    nodes.setNullAllowed(true);
    IIntSetsizes = f.createIntSet(minSize, maxSize);
    IObjSet keys = f.createObjSet(Integer.class);
    IClassDomain elemsClassDomain = f.createClassDomain(Integer.class);
    elemsClassDomain.includeInIsomorphismCheck(false);
    for (int i = 1; i < numKeys; i++)
        keys.addClassDomain(elemsClassDomain);
    keys.setNullAllowed(false);
    f.set("root", nodes);
    f.set("size", sizes);
    f.set("Node.left", nodes);
    f.set("Node.right", nodes);
    f.set("Node.info", keys);
    f.set("param", keys);
    return f;
}

public BigInteger getClass(boolean[] classes){
    BigInteger res = new BigInteger("0");
    BigInteger dos = new BigInteger("2");
    for (int i = 0; i < classes.length; i++) {
        if (classes[i]){
            BigInteger t = dos.pow(i);
            res = res.add(t);
        }
    }
    return res;
}

public Node Search() {
    Node p = root;
    while (p != null) {
        if (param < p.info)
```java
public boolean isLeaf(Node b) {
    return (b.left == null) && (b.right == null);
}

public boolean hasTwoChildren(Node b) {
    return (b.left != null) && (b.right != null);
}

public boolean hasJustLeftChild(Node b) {
    return (b.left != null) && (b.right == null);
}

public boolean hasJustRightChild(Node b) {
    return (b.left == null) && (b.right != null);
}

public boolean isRoot(Node n) {
    return (n != null && n == root);
}

public String toString() {
    StringBuffer buf = new StringBuffer();
    buf.append(size);
    buf.append("{");
    if (root != null) {
        buf.append(root.toString());
    }
    return buf.toString();
}
```

```java
buf.append("\n");
return buf.toString();
*/

/** eqClass returns the equivalence class in which the current
 * instance of searchTreeBB belongs to. In this case, the
 * equivalence classes are given by all the possible combinations
 * of the following predicates apply on the node containing
 * 'param' as info: isNotOntheTree, isLeaf, hasTwoChildren,
 * hasJustLeftChild, hasJustRightChild and isRoot.
 * @return The equivalence class in which this instance belongs
 */
public BigInteger eqClass()
{
    boolean[] classes = new boolean[6];
    Node b = this.Search():
    classes[0] = (b==null); /**param is not in the tree*/
    classes[1] = (b!=null) && (this.isLeaf(b));
    classes[2] = (b!=null) && (this.hasTwoChildren(b));
    classes[3] = (b!=null) && (this.hasJustLeftChild(b));
    classes[4] = (b!=null) && (this.hasJustRightChild(b));
    classes[5] = isRoot(b);

    return this.getClass(classes);
}

public void delete() {
    root = remove(root);
}

/**
 * Internal method to remove from a subtree.
 * @param x the item to remove.
 * @param t the node that roots the tree.
 * @return the new root.
 */
protected Node remove(Node t) {
    if( t == null ){
        return null;
    }
    if( param.intValue() < t.info )
        t.left = remove(t.left);
    else{
        if( param.intValue() > t.info )
            t.right = remove(t.right);
    else{
            if( t.left != null && t.right != null ){
                t.info = findMin(t.right).info;
                t.right = removeMin(t.right);
            }
        else{
            if(t.left != null){
                size--;
                t=t.left;
            }else{
                size--;
                t= t.right;
            }
        }
    }
    return t;
}

/**
protected Node removeMin(Node t) {
  if (t == null) {
    return null;
  } else {
    if (t.left != null) {
      t.left = removeMin(t.left);
      return t;
    } else {
      size--;
      return t.right;
    }
  }
}

protected Node findMin(Node t) {
  if (t != null) {
    while (t.left != null)
      t = t.left;
  return t;
  }
```java
    this.right = right;
    this.info = info;
}
Node(Integer i) {
    this.info = i;
}

public Node() {

public boolean equals(Object that) {
    if (!that instanceof Node)
        return false;
    Node n = (Node) that;
    if (this.info.intValue() > (n.info.intValue()))
        return false;
    boolean b = true;
    if (left == null)
        b = b && (n.left == null);
    else
        b = b && (left.equals(n.left));
    if (right == null)
        b = b && (n.right == null);
    else
        b = b && (right.equals(n.right));
    return b;
}

public String toStrings() {
    Set visited = new HashSet();
    visited.add(this);
    return toString(visited);
}

private String toString(Set visited) {
    StringBuffer buf = new StringBuffer();
    buf.append("{");
    if (left != null)
        if (visited.add(left))
            buf.append(left.toString(visited));
        else
            buf.append("!tree");
    buf.append(" + this.info + ");
    if (right != null)
        if (visited.add(right))
            buf.append(right.toString(visited));
        else
            buf.append("!tree");
    return buf.toString();
}

Insertion and Search in RedBlackTrees

package koratPlus.RedBlackTreeWB;
import java.util.Set;
import java.util.Arrays;
import korat.finitization.IClassDomain;
import korat.finitization.IFinitization;
import korat.finitization.IIntSet;
import korat.finitization.IObjSet;
```
import korat.finitization.impl.FinitizationFactory;

import java.util.Arrays;
import java.math.BigInteger;

/**
 * Class that defines Red Black trees. In this class the methods
 * insert and search were instrumented to be used as eqClass.
 * (White Box coverage criteria).
 * @author
 */
public class RedBlackTreeWB {

    public static class Node {
        Integer key;
        int value;
        int color = BLACK;
        Node left = null;
        Node right = null;
        Node parent;

        public Node(Integer key_param, Node parent_param) {
            key = key_param;
            parent = parent_param;
        }

        public Node(){}
    }

    /**
     * parameter to be inserted/searched. This new attribute is
     * necessary since we want korat to generate
     * all valid entries to test the insert and search routines on RBT,
     * which are pairs of RedBlackTrees and integers.
     */
    private Integer param = new Integer(0);

    private Node root = null;
    private int size = 0;

    private static final int RED = 0;
    private static final int BLACK = 1;

    public RedBlackTreeWB(Node r){
        root = r;
    }

    public static IFinitization finRedBlackTreeWB(int size) {
        return finRedBlackTreeWB(size, size, size, size);
    }

    /**
     * finRedBlackTreeWB provides a bound on the number of objects to be
     * used to generate instances of RedBlackTreeWB.
     * @param numEntries number of entries to be used to generate
     * @instance of RedBlackTreeWB.
     */
public static IFinInitialization finRedBlackTreeWB(int numEntries, int minSize, int maxSize, int numKeys) {
    IFinInitialization f = FinInitializationFactory.create(RedBlackTreeWB.class);
    IClassDomain entryDomain = f.createClassDomain(Node.class, numEntries);
    IObjSet entries = f.createObjectSet(Node.class, numEntries, true);
    entries.addClassDomain(entryDomain);
    IIntSet sizes = f.createIntSet(minSize, maxSize);
    IObjSet keys = f.createObjectSet(Integer.class);
    IClassDomain elemsClassDomain = f.createClassDomain(Integer.class);
    elemsClassDomain.includeInIsomorphismCheck(false);
    for (int i = 1; i <= numKeys; i++)
        elemsClassDomain.addObject(new Integer(i));
    keys.addClassDomain(elemsClassDomain);
    keys.setNullAllowed(false);
    IIntSet values = f.createIntSet(0);
    IIntSet colors = f.createIntSet(0, 1);
    f.set("root", entries);
    f.set("size", sizes);
    f.set("Node.left", entries);
    f.set("Node.right", entries);
    f.set("Node.parent", entries);
    f.set("Node.color", colors);
    f.set("Node.value", values);
    f.set("Node.key", keys);
    f.set("param", keys);

    return f;
}

public boolean repOK() {
    if (this.param == null)
        return false;
    if (root == null)
        return size == 0;
    // RootHasNoParent
    if (root.parent != null)
        return debug("RootHasNoParent");
    Set visited = new java.util.HashSet();
    visited.add(new Wrapper(root));
}
java.util.LinkedList workList = new java.util.LinkedList();
workList.add(root);
while (!workList.isEmpty()) {
    // Acyclic
    // // if (!visited.add(new Wrapper(current)))
    // // return debug("Acyclic");
    // Parent Definition
    Node cl = current.left;
    if (cl != null) {
        if (!visited.add(new Wrapper(cl)))
            return debug("Acyclic");
        if (cl.parent != current)
            return debug("parent_INPUT1");
        workList.add(cl);
    }
    Node cr = current.right;
    if (cr != null) {
        if (!visited.add(new Wrapper(cr)))
            return debug("Acyclic");
        if (cr.parent != current)
            return debug("parent_INPUT2");
        workList.add(cr);
    }
    // SizeOk
    if (visited.size() != size)
        return debug("SizeOk");
    if (!repOkColors())
        return false;
}
return repOkKeysAndValues();

/** Checks whether this RedBlackTree is balanced according the color
 * of its nodes.
 * @return true iff Root is black, Red nodes only have black children
 * and all the path of the tree.
 * from root to null have the same numbers of black nodes.
 */
private boolean repOkColors() {
    if ((root != null) && root.color == RED) {
        return debug("Root must be Black");
    }
    java.util.LinkedList workList = new java.util.LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current = (Node) workList.removeFirst();
        Node cl = current.left;
        Node cr = current.right;
        if (current.color == RED) {
            if (cl != null && cl.color == RED)
                return debug("RedHasOnlyBlackChildren1");
            if (cr != null && cr.color == RED)
                return debug("RedHasOnlyBlackChildren2");
        }
        if (cl != null)
            workList.add(cl);
        if (cr != null)
            workList.add(cr);
    }
    /* SimplePathsFromRootToNILHaveSameNumberOfBlackNodes*/
    int numberOfBlack = -1;
    workList = new java.util.LinkedList();
    workList.add(new Pair(root, 0));
while (! workList.isEmpty()) {
    Pair p = (Pair) workList.removeFirst();
    Node e = p.e;
    int n = p.n;
    if (e != null && e.color == BLACK)
        n++;
    if (e == null) {
        if (numberOfBlack == -1)
            numberOfBlack = n;
        else if (numberOfBlack != n)
            return debug("SimplePathsFromRootToNILHaveSameNumberOfBlackNodes ");
    } else {
        workList.add(new Pair(e.left, n));
        workList.add(new Pair(e.right, n));
    }
}
return true;
}
/**
 * checks whether this tree is a binary search tree.
 * @return true iff this tree is a binary search tree (Its keys are ordered).
 */
private boolean repOkKeysAndValues() {
    //BST1 and BST2
    this was the old way of determining if the keys are ordered
    java.util.LinkedList workList = new java.util.LinkedList();
    workList.add(root);
    while (! workList.isEmpty()) {
        Entry current = (Entry) workList.removeFirst();
        Entry cl = current.left;
        Entry cr = current.right;
        if (current.key==current.key) {
            if (cl != null)
                if (compare(current.key, current.maximumKey()) < 0)
                    return debug("BST1");
            workList.add(cl);
        }
        if (cr != null) {
            if (compare(current.key, current.minimumKey()) >= 0)
                return debug("BST2");
            workList.add(cr);
        }
    }
    return true;
}

private boolean orderedKeys(Node e, Object min, Object max) {
    if (e.key.intValue() == -1)
        return true;
    }
274    return false;
275    if (((min != null) && (compare(e.key.intValue(), min) <= 0))
276        || ((max != null) && (compare(e.key.intValue(), max) >= 0)))
277        return false;
278    if (e.left != null)
279        if (!orderedKeys(e.left, min, e.key.intValue()))
280            return false;
281    if (e.right != null)
282        if (!orderedKeys(e.right, e.key.intValue(), max))
283            return false;
284    return true;
285}
286
287    private final boolean debug(String s) {
288        //System.out.println(s);
289        return false;
290    }
291
292    private final class Pair {
293        Node e;
294
295        int n;
296
297        Pair(Node e, int n) {
298            this.e = e;
299            this.n = n;
300        }
301    }
302
303    private static final class Wrapper {
304        Node e;
305
306        Wrapper(Node e) {
307            this.e = e;
308        }
309
310        public boolean equals(Object obj) {
311            if (! (obj instanceof Wrapper))
312                return false;
313            return e == ((Wrapper) obj).e;
314        }
315
316        public int hashCode() {
317            return System.identityHashCode(e);
318        }
319    }
320
321    private int compare(Object k1, Object k2) {
322        return ((Comparable) k1).compareTo(k2);
323    }
324
325    public boolean isEmpty() {
326        return root == null;
327    }
328
329    /***getClass translates a given array of boolean into a Integer
330     * @param classes The array to be translated
331     * @return The integer that represents the given array
332     * @see eqClass()
333     */
334    public BigInteger getClass(boolean[] classes) {
335        BigInteger res = new BigInteger("0");
336        BigInteger dos = new BigInteger("2");
337        for (int i = 0; i < classes.length; i++)
338            if (classes[i]) {
339                BigInteger t = dos.pow(i);
340                res = res.add(t);
341    }
342    }
343    return res;
344 }
345
346 /**
347 * init sets all the given array position in false
348 * @param cl The array to be initialized.
349 */
350 private void init(boolean [] cl){
351    int i = 0;
352    while(i < cl.length){
353        cl[i]=false;
354        i++;
355    }
356 }
357
358 /** Insertion method instrumented to be used as a eqClass during
359 * the pruning (Decision Coverage).
360 * The element to be added is 'param'
361 * @return the equivalence class in which this instance belong to.
362 */
363 public BigInteger eqClassInsertionDC() {
364    boolean [] classes = new boolean[100];
365    init(classes); // it sets all array position in false
366    Node t = root;
367    if (t == null) {
368        classes[0] = true;
369        root = new Node(param, null);
370        size = 1;
371        return this.getClass(classes);
372    } else classes[1] = true;
373    Node parent;
374    int cmp;
375    int i = 0;
376    do {
377        if (i != 0) {
378            classes[6] = true;
379        }
380        parent = t;
381        if (t.key > param) {
382            t = t.left;
383            cmp = -1;
384            classes[2] = true;
385        } else {
386            classes[3] = true;
387            if (t.key < param) {
388                t = t.right;
389                cmp = +1;
390                classes[4] = true;
391            } else {
392                classes[5] = true;
393                return this.getClass(classes);
394            }
395        }
396        i++;
397    } while (t != null);
398    classes[7] = true;
399    Node x = new Node(param, parent);
400    if (cmp < 0) {
401        parent.left = x;
402        classes[8] = true;
403    } else {
404        parent.right = x;
405    }
```java
    classes[9] = true;
} //fixAfterInsertion
x.color = RED;
Node n1, n2, n3, n4 = null;
while (x != null && x != root && x.parent.color == RED) {
    classes[98] = true;
    if (x == null) {
        classes[10] = true;
        n1 = null; //parentOf(x)
    }
    else {
        classes[11] = true;
        n1 = x.parent;
    }
    if (n1 == null) {
        classes[12] = true;
        n2 = null; //parentOf(parentOf(x)) parentOf(n1)
    }
    else classes[15] = true;
    if (n1 != null) {
        classes[16] = true;
    } else {
        Node d1 = null; //leftOf(parentOf(parentOf(x))) leftOf(n2)
        if (n2 != null) {
            classes[14] = true;
            d1 = n2.left;
        } else classes[15] = true;
        if (n1 != d1) {
            classes[16] = true;
        } Node y = null;
        if (n2 != null) { //Node y = rightOf(n2);
            classes[18] = true;
            y = n2.right;
        } else classes[15] = true;
        //colorOf(y)
        int cs = 0;
        if (y == null) {
            classes[20] = true;
            cs = BLACK;
        } else {
            classes[21] = true;
            cs = y.color;
        }
        if (cs == RED) {
            classes[22] = true;
            if (n1 != null) {
                classes[24] = true;
                n1.color = BLACK;
            } else classes[25] = true;
            if (y != null) {
                classes[26] = true;
                y.color = BLACK;
            } else classes[27] = true;
            if (n2 != null) {
                classes[28] = true;
                n2.color = RED;
            } else classes[29] = true;
        } else {
            classes[23] = true;
        }
    } else {
        classes[23] = true;
    }
    if (n1 != null) {
        classes[30] = true;
        d = n1.right;
    } else classes[31] = true;
    if (x == d) {
```
```java
if (x != null) {
    x.right = r.left;
    if (r.left != null) {
        r.left.parent = x;
        r.parent = x.parent;
        if (x.parent != null) {
            classes[38] = true;
            root = r;
        } else {
            classes[39] = true;
            if (x.parent.left == x) {
                Classes[40] = true;
            } else {
                x.parent.right = r;
                classes[41] = true;
            }
        }
    }
    r.left = x;
    x.parent = r;
} else classes[35] = true; // end rotateLeft(x)
}
else classes[33] = true; // if (x == rightOf(parentOf(x))) if (x == d)
if (x == null) {
    classes[42] = true;
    n3 = null; // parentOf
} else {
    classes[43] = true;
    n3 = x.parent;
}
if (n3 != null) {
    classes[44] = true;
    n4 = null; // parentOf(parentOf(x)) parentOf(n1)
} else {
    classes[45] = true;
    n4 = n3.parent;
}
if (n3 != null) {
    classes[46] = true;
    n3.color = BLACK;
} else classes[47] = true;
if (n4 != null) {
    classes[48] = true;
    n4.color = RED;
} else classes[49] = true;
Node n = n4;
// rotateRight(n);
if (n != null) {
    classes[50] = true;
    Node l = n.left;
    l.left = l.right;
    if (l.right != null) {
        classes[52] = true;
        l.right.parent = n;
    } else classes[53] = true;
    l.parent = n.parent;
```
if (n.parent == null)
    classes[54] = true;
root = i;
} else {
    classes[55] = true;
    if (n.parent.right == n)
        classes[56] = true;
    n.parent.right = i;
} else {
    classes[57] = true;
    n.parent.left = i;
}
}
}
}
// RotateRight(n)

if (colorOf(y) == RED)
}
else {
    if (x == d)
        x = n1;
    // RotateRight(x)
    if (x != null)
        classes[74] = true;
    Node l = x.left;
    x.left = l.right;
    if (l.right != null)
        classes[76] = true;
    l.right.parent = x;
} else classes[77] = true;
l.parent = x.parent;
if (x.parent == null) {
    classes[78] = true;
    root = l;
} else {
    classes[79] = true;
    if (x.parent.right == x) {
        classes[80] = true;
        x.parent.right = l;
    } else {
        classes[81] = true;
        x.parent.left = l;
    }
}
else classes[75] = true;

//rotateRight(x);
if (x == null) {
    classes[82] = true;
    n3 = null; //parentOf(x)
} else {
    classes[83] = true;
    n3 = x.parent;
    if (n3 == null) {
        classes[84] = true;
        n4 = null; //parentOf(parentOf(x)) parentOf(n1)
    } else {
        classes[85] = true;
        n4 = n3.parent;
        if (n4 == null) {
            classes[86] = true;
            n3.color = BLACK;
        } else classes[87] = true;
    }
if (n4 != null) {
    classes[88] = true;
    n4.color = RED;
} else classes[89] = true;
Node p = n4;
//rotateLeft(p);
if (p != null) {
    classes[90] = true;
    Node r = p.right;
    p.right = r.left;
    if (r.left != null) {
        classes[91] = true;
        r.left.parent = p;
    } else classes[92] = true;
    r.parent = p.parent;
    if (p.parent == null) {
        classes[94] = true;
        root = r;
    } else {
        classes[95] = true;
        if (p.parent.left == p) {
            classes[96] = true;
            p.parent.left = r;
        } else {
            
        }
    } else {
        classes[82] = true;
        n3 = null; //parentOf(x)
    } else {
        classes[83] = true;
        n3 = x.parent;
        if (n3 == null) {
            classes[84] = true;
            n4 = null; //parentOf(parentOf(x)) parentOf(n1)
        } else {
            classes[85] = true;
            n4 = n3.parent;
            if (n4 == null) {
                classes[86] = true;
                n3.color = BLACK;
            } else classes[87] = true;
        }
    }
if (n4 != null) {
    classes[88] = true;
    n4.color = RED;
} else classes[89] = true;
Node p = n4;
//rotateLeft(p);
if (p != null) {
    classes[90] = true;
    Node r = p.right;
    p.right = r.left;
    if (r.left != null) {
        classes[91] = true;
        r.left.parent = p;
    } else classes[92] = true;
    r.parent = p.parent;
    if (p.parent == null) {
        classes[94] = true;
        root = r;
    } else {
        classes[95] = true;
        if (p.parent.left == p) {
            classes[96] = true;
            p.parent.left = r;
        } else {
            
        }
    } else {
        classes[82] = true;
        n3 = null; //parentOf(x)
    } else {
        classes[83] = true;
        n3 = x.parent;
        if (n3 == null) {
            classes[84] = true;
            n4 = null; //parentOf(parentOf(x)) parentOf(n1)
        } else {
            classes[85] = true;
            n4 = n3.parent;
            if (n4 == null) {
                classes[86] = true;
                n3.color = BLACK;
            } else classes[87] = true;
        }
    }
    
} else {

}
classes[97] = true;
p.parent.right = r;
}
r.left = p;
p.parent = r;
}
} // end rotateLeft(p);
}
}
} // end while
classes[99] = true;
root.color = BLACK;
// end fixAfterInsertion
size++;
return this.getClass(classes);
}

/** Search method instrumented to be used as a eqClass during
 * the pruning (Desicion Coverage).
 * The element to be searched is 'param'.
 * @return the equivalence class in which this instance belong to.
 */
public BigInteger eqClassSearchDC() {
    boolean[] classes = new boolean[6];
    init(classes); // it sets all array position in false
    Node p = root;
    while (p != null) {
        classes[0] = true;
        if (param < p.key) {
            classes[2] = true;
            p = p.left;
        } else {
            classes[3] = true;
            if (param > p.key) {
                classes[4] = true;
                p = p.right;
            } else {
                classes[5] = true;
                return this.getClass(classes);
            }
        }
        classes[1] = true;
        return this.getClass(classes);
    }
}

Insertion and Search in SearchTrees

package koratPlus.SearchTreeWB;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Set;
import java.math.BigInteger;
import korat.finitzation.IColorDomain;
import korat.finitzation.IFinitzation;
import korat.finitzation.IntSet;
import korat.finitzation.ImplFinitzationFactory;

/**
 * Class that defines Search trees where the methods
 * insert and search were instrumented to be used as eqClass
 * (White Box coverage criteria).
 */

public class SearchTreeWB {

    public static class Node {
        /**
         * left child
         */
        Node left;

        /**
         * right child
         */
        Node right;

        /**
         * data
         */
        Integer info;

        Node(Node left, Node right, int info) {
            this.left = left;
            this.right = right;
            this.info = info;
        }

        Node(Integer i) {
            this.info = i;
        }

        public Node(){}

        public boolean equals(Object that) {
            if (! (that instanceof Node))
                return false;
            Node n = (Node) that;
            if (this.info.intValue() > (n.info.intValue()))
                return false;
            boolean b = true;
            if (left == null)
                b = b && (n.left == null);
            else
                b = b && (left.equals(n.left));
            if (right == null)
                b = b && (n.right == null);
            else
                b = b && (right.equals(n.right));
            return b;
        }
    }

    /**
     * parameter to be inserted/searched. This new attribute is
     * necessary since we want korat to generate
     * all valid entries to test the insert and search routines on RBT,
     * which are pairs of SearchTrees and integers.
     */
    private Integer param = new Integer(0);

    /**
private Node root;

/**
 * number of nodes in the tree
 */
private int size;

public SearchTreeWB(Node r)
{
    root = r;
}

/**
 * checks that the current SearchTree satisfies its representation
 * invariant, which means checking that it is a well built tree
 * and the data is ordered.
 * @return true iff this searchTree satisfies the corresponding
 * representation invariant.
 */
public boolean repOK()
{
    if (param == null)
        return false;
    /**
     * checks that empty tree has size zero*
     */
    if (root == null)
        return size == 0;
    /**
     * checks that the input is a tree*/
    if (!isAcyclic())
        return false;
    /**
     * checks that size is consistent*/
    if (numNodes(root) != size)
        return false;
    /**
     * checks that data is ordered*/
    if (!isOrdered(root))
        return false;
    return true;
}

/**
 * checks that the tree has no cycle
 * @return true iff the tree has no cycle
 */
private boolean isAcyclic()
{
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty())
    {
        Node current = (Node) workList.removeFirst();
        if (current.left != null)
        {
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null)
        {
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }
    return true;
}

/**
 * counts the number of nodes in the tree starting from 'n'
 * @param n the starting node
 * @return the number of nodes in the current tree
 */
private int numNodes(Node n) {
    if (n == null)
        return 0;
    return 1 + numNodes(n.left) + numNodes(n.right);
}

private int compare(Object k1, Object k2) {
    return ((Comparable) k1).compareTo(k2);
}

private boolean isOrdered(Node n) {
    return isOrdered(n, null, null);
}

private boolean isOrdered(Node e, Object min, Object max) {
    if (e.info.intValue() == -1)
        return false;
    if (((min != null) && (compare(e.info.intValue(), min) <= 0))
        || ((max != null) && (compare(e.info.intValue(), max) >= 0)))
        return false;
    if (e.left != null)
        if (!isOrdered(e.left, min, e.info.intValue()))
            return false;
    if (e.right != null)
        if (!isOrdered(e.right, e.info.intValue(), max))
            return false;
    return true;
}

public String toString() {
    StringBuffer buf = new StringBuffer();
    if (root != null)
        buf.append(root.toString());
    return buf.toString();
}

public static IFinInitialization finSearchTreeWB(int numNodes) throws Exception {
    return finSearchTreeWB(numNodes, numNodes, numNodes, numNodes);
}

/*
 * finSearchTreeBB provides a bound on the number of objects to be
 * used to generate instances of SearchTreeBB.
 * @param numNodes number of entries to be used to generate
 * instance of SearchTreeBB.
 * @param minSize minimum size of the generated SearchTreeBB
 * @param maxSize maximum size of the generated SearchTreeBB
 * @param numKeys The range of keys of each entry goes between
 * 1 and numKeys.
 * @return the object Korat needs for setting up the bounds
 * during the search.
 * @throws Exception
 */

public static IFinInitialization finSearchTreeWB(int numNodes, int minSize, int maxSize, int numKeys) throws Exception {
    IFinInitialization f = FinitizationFactory.create(SearchTreeWB.class);
    IObjSet nodes = f.createObjectSet(Node.class, numNodes);
    ...
214    nodes.setNullAllowed(true);
215    IIntSet sizes = f.createIntSet(minSize, maxSize);
216    IObjSet keys = f.createObjectSet(Integer.class);
217    IClassDomain elemsClassDomain = f.createClassDomain(Integer.class);
218    elemsClassDomain.includeInIsomorphismCheck(false);
219    for (int i = 1; i <= numKeys; i++)
220        elemsClassDomain.addObject(new Integer(i));
221    keys.addClassDomain(elemsClassDomain);
222    keys.setNullAllowed(false);
223    f.set("root", nodes);
224    f.set("size", sizes);
225    f.set("Node.left", nodes);
226    f.set("Node.right", nodes);
227    f.set("Node.info", keys);
228    f.set("param", keys);
229    return f;
230 }
231 }
232 */
233 /**
234 * getClass translates a given array of boolean into a Integer
235 * @param classes The array to be translated
236 * @return The integer that represents the given array
237 * @see eqClass()
238 */
239 public BigInteger getClass(boolean [] classes)
240 {
241     BigInteger res = new BigInteger("0");
242     BigInteger dos = new BigInteger("2");
243     for (int i=0; i<classes.length; i++)
244         if (classes[i])
245             BigInteger t = dos.pow(i);
246             res = res.add(t);
247 }
248 return res;
249 }
250 */
251 /**
252 * init sets all the given array position in false
253 * @param cl The array to be initialized.
254 */
255 private void init(boolean [] cl){
256     int i = 0;
257     while(i < cl.length){
258         cl[i]=false;
259         i++;
260     }
261 }
262 */
263 /**
264 * Search method instrumented to be used as a eqClass during
265 * the pruning (Desicion Coverage).
266 * @return the equivalence class in which this instance belong to.
267 */
268 public BigInteger eqClassSearchDC() {
269     boolean [] classes = new boolean [6];
270     init(classes);
271     Node p = root;
272     while (p != null) {
273         classes[0] = true;
274         if (param < p.info){
275             classes[2] = true;
276             p = p.left;
277         }
278         else {
279             if (param > p.info){
280                 classes[1] = true;
281                 p = p.right;
282             }
283         }
284     }
285     return getClass(classes);
286 }
287 }


```java
} else {
    classes[3] = true;
    if (param > p.info) {
        classes[4] = true;
        p = p.right;
    } else {
        classes[5] = true;
        return this.getClass(classes);
    }
}

    classes[1] = true;
    return this.getClass(classes);
}

// Insertion method instrumented to be used as a eqClass during
// the pruning (Desicion Coverage).
// @return
public BigInteger eqClassInsertionDC() {
    boolean[] classes = new boolean[10];
    init(classes);
    if (root == null) {
        classes[0] = true;
        root = new Node(param);
    } else {
        classes[1] = true;
        Node p = root;
        Node t = root;
        Node n = new Node(param);
        while (p != null) {
            classes[2] = true;
            t = p;
            if (param < p.info) {
                classes[4] = true;
                p = p.left;
            } else {
                classes[5] = true;
                if (param > p.info) {
                    classes[6] = true;
                    p = p.right;
                } else {
                    classes[7] = true;
                    return this.getClass(classes);
                }
            } classes[3] = true;
            if (param < t.info) {
                classes[8] = true;
                t.left = n;
            } else {
                classes[9] = true;
                t.right = n;
            }
            size++;
        }
        return this.getClass(classes);
    }
```
SinglyLinkedList

```java
package koratPlusExtras.Lists;

import java.util.Set;
import java.math.BigInteger;
import korat.finitization.IClassDomain;
import korat.finitization.IFinitization;
import korat.finitization.IIntSet;
import korat.finitization.IObjSet;
import korat.finitization.impl.FinitizationFactory;

public class SinglyLinkedList {
    public static class Entry {
        Integer element;
        Entry next;

        public String toString() {
            return "[ " + (element != null ? element.toString() : "null") + "]";
        }
    }

    public Entry header;
    private int size = 0;

    public boolean repOkCommon() {
        if (header == null)
            return false;

        if (header.element != null)
            return false;

        Set<Entry> visited = new java.util.HashSet<Entry>();
        Entry current = header;

        while (true) {
            Entry next = current.next;
            if (next == null)
                break;

            if (next.element == null)
                return false;

            if (!visited.add(next))
                return false;

            current = next;
        }

        if (visited.size() - 1 != size)
            return false;

        return true;
    }

    public boolean repOK() {
        if (!repOkCommon())
            return false;

        return true;
    }

    public boolean isEmpty() {

```
if (header.next == null)
    return true;
else{
    return false;
}
}

public boolean sorted() {
    if (size > 1) {
        for (Entry current = header.next; current.next != null;
            current = current.next) {
            if (current.element.compareTo(current.next.element) >= 0)
                return false;
        }
    }
    return true;
}

/**
 * Checks whether or not the current list has not repeated elements.
 * @return true iff all the elements in the list are different
 * each other.
 */
public boolean noReps() {
    if (!isEmpty()) {
        for (Entry current = header.next; current.next != null;
            current = current.next) {
            if (current.element.intValue() == current.next.
                element.intValue())
                return false;
        }
    }
    return true;
}

public BigInteger getClass(boolean [] classes) {
    BigInteger res = new BigInteger("0");
    BigInteger dos = new BigInteger("2");
    for (int i = 0; i < classes.length; i++) {
        if (classes[i]){
            BigInteger t = dos.pow(i);
            res = res.add(t);
        }
    }
    return res;
}

public BigInteger eqClass(){
    boolean [] classes = new boolean [3];
    classes [0] = isEmpty();
    classes [1] = sorted();
    classes [2] = noReps();
    return this.getClass(classes);
}

public String toString() {
    String res = "(\n    if (header != null) {
        Entry cur = header.next;
        while (cur != header && cur != null &
            cur != header) {
            res += cur.toString();
            cur = cur.next;
        }
    }
    return res + ")";
}
public static IFinInitialization finSinglyLinkedList(int minSize, int maxSize,
int numEntries, int numElems) {
    IFinInitialization f = FinInitializationFactory.create(SinglyLinkedList.class);
    IObjSet entries = f.createObjectSet(Entry.class, true);
    entries.addClassDomain(f.createClassDomain(Entry.class, numEntries));
    IIntSet sizes = f.createIntSet(minSize, maxSize);
    IObjSet elems = f.createObjectSet(Integer.class);
    IClassDomain elemsClassDomain = f.createClassDomain(Integer.class);
    elemsClassDomain.includeInIsomorphismCheck(false);
    for (int i = 1; i <= numElems; i++)
        elemsClassDomain.addObject(new Integer(i));
    elems.addClassDomain(elemsClassDomain);
    elems.setNullAllowed(true);
    f.set("header", entries);
    f.set("size", sizes);
    f.set(Entry.class, "element", elems);
    f.set(Entry.class, "next", entries);
    return f;
}

package koratPlusExtras.Lists;
public class Entry {
    Integer element;
    Entry next;
    public String toString() {
        return "[" + (element != null ? element.toString() : "null")
            + 
"]";
    }
}

Appendix: Mutation Testing

We used muJava tool to obtain the mutants used in the experimental analyses, the following are the method level operator that can be applied for muJava:

- AORb: Arithmetic Operator Replacement
- AORs: Arithmetic Operator Replacement
- AODs: Arithmetic Operator Insertion
- AODu: Arithmetic Operator Insertion
- AOIu: Arithmetic Operator Deletion
- AOIs: Arithmetic Operator Deletion
- ROR: Relational Operator Replacement
- COR: Conditional Operator Replacement
- COI: Conditional Operator Insertion
- COD: Conditional Operator Deletion
- SOR: Shift Operator Replacement
– LOR: Logical Operator Replacement
– LOI: Logical Operator Insertion
– LOD: Logical Operator Deletion
– ASRs: Assignment Operator Replacement

Notice that some of the operators are subdivided into two, according to the number and type of operand. For example, AOR operator is subdivided into AORb and AORs. AORb is for binary arithmetic operator and AORs is for short-cut arithmetic operator [12].

We asked muJava to applied all these operators over the 3 selected cases studies: ListASet(ListToSet), BinomialHeap(Merge) and SearchTree(Delete), but not all the operator were applicable for each of them. We now describe the operators applied in each case and the obtained mutants.

**ListAsSet.**

For the listToSet method, the following are the mutation operator applied:

– AORs: Arithmetic Operator Replacement(short-cut)
– AOIu: Arithmetic Operator Deletion(unary)
– AOIs: Arithmetic Operator Deletion(short-cut)
– ROR: Relational Operator Replacement
– COR: Conditional Operator Replacement
– COD: Conditional Operator Deletion
– LOI: Logical Operator Insertion

The Generated mutants are showed in the table below. We indicated, together with the mutation, the operator applied and the line number where the mutation is placed in the code (see Appendix above).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Line Number</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AOIS</td>
<td>87</td>
<td>i =&gt; ++i</td>
</tr>
<tr>
<td>AOIS</td>
<td>87</td>
<td>i =&gt; --i</td>
</tr>
<tr>
<td>AOIS</td>
<td>87</td>
<td>i =&gt; i++</td>
</tr>
<tr>
<td>AOIS</td>
<td>87</td>
<td>list.size =&gt; ++list.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>87</td>
<td>list.size =&gt; --list.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>87</td>
<td>list.size =&gt; list.size++</td>
</tr>
<tr>
<td>AOIS</td>
<td>87</td>
<td>list.size =&gt; list.size--</td>
</tr>
<tr>
<td>AOIS</td>
<td>88</td>
<td>i =&gt; i++</td>
</tr>
<tr>
<td>AOIS</td>
<td>93</td>
<td>i =&gt; ++i</td>
</tr>
<tr>
<td>AOIS</td>
<td>93</td>
<td>i =&gt; --i</td>
</tr>
<tr>
<td>AOIS</td>
<td>93</td>
<td>i =&gt; i++</td>
</tr>
<tr>
<td>AOIS</td>
<td>93</td>
<td>set.size =&gt; ++set.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>93</td>
<td>set.size =&gt; --set.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>93</td>
<td>set.size =&gt; set.size++</td>
</tr>
<tr>
<td>AOIS</td>
<td>93</td>
<td>set.size =&gt; set.size--</td>
</tr>
<tr>
<td>AOIS</td>
<td>94</td>
<td>i =&gt; i++</td>
</tr>
<tr>
<td>AOIS</td>
<td>100</td>
<td>i =&gt; ++i</td>
</tr>
<tr>
<td>AOIS</td>
<td>100</td>
<td>i =&gt; --i</td>
</tr>
<tr>
<td>AOIS</td>
<td>100</td>
<td>i =&gt; i++</td>
</tr>
<tr>
<td>AOIS</td>
<td>100</td>
<td>s.size =&gt; ++s.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>100</td>
<td>s.size =&gt; --s.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>100</td>
<td>s.size =&gt; s.size++</td>
</tr>
<tr>
<td>AOIS</td>
<td>100</td>
<td>s.size =&gt; s.size--</td>
</tr>
<tr>
<td>AOIS</td>
<td>101</td>
<td>i =&gt; i++</td>
</tr>
<tr>
<td>ROR</td>
<td>126</td>
<td>current == null =&gt; current != null</td>
</tr>
<tr>
<td>ROR</td>
<td>131</td>
<td>current.element.intValue() != value.intValue() =&gt; current.element.intValue() &gt; value.intValue()</td>
</tr>
<tr>
<td>ROR</td>
<td>131</td>
<td>current.element.intValue() != value.intValue() =&gt; current.element.intValue() &gt;= value.intValue()</td>
</tr>
<tr>
<td>ROR</td>
<td>131</td>
<td>current.element.intValue() != value.intValue() =&gt; current.element.intValue() &lt; value.intValue()</td>
</tr>
<tr>
<td>ROR</td>
<td>131</td>
<td>current.element.intValue() != value.intValue() =&gt; current.element.intValue() &lt;= value.intValue()</td>
</tr>
<tr>
<td>ROR</td>
<td>131</td>
<td>current.element.intValue() != value.intValue() =&gt; current.element.intValue() == value.intValue()</td>
</tr>
<tr>
<td>COR</td>
<td>120</td>
<td>current != null &amp;&amp; current.element.intValue() &lt; value.intValue() =&gt; current != null</td>
</tr>
<tr>
<td>COR</td>
<td>120</td>
<td>current != null &amp;&amp; current.element.intValue() &lt; value.intValue() =&gt; current != null ^ current.element.intValue() &lt; value.intValue()</td>
</tr>
<tr>
<td>COD</td>
<td>95</td>
<td>!s.contains(value) =&gt; s.contains(value)</td>
</tr>
<tr>
<td>COD</td>
<td>102</td>
<td>!set.contains(value) =&gt; set.contains(value)</td>
</tr>
<tr>
<td>LOI</td>
<td>87</td>
<td>i =&gt; ~i</td>
</tr>
<tr>
<td>LOI</td>
<td>87</td>
<td>list.size =&gt; -list.size</td>
</tr>
<tr>
<td>LOI</td>
<td>88</td>
<td>i =&gt; ~i</td>
</tr>
<tr>
<td>LOI</td>
<td>93</td>
<td>i =&gt; ~i</td>
</tr>
<tr>
<td>LOI</td>
<td>93</td>
<td>set.size =&gt; -set.size</td>
</tr>
<tr>
<td>LOI</td>
<td>94</td>
<td>i =&gt; ~i</td>
</tr>
<tr>
<td>LOI</td>
<td>100</td>
<td>i =&gt; ~i</td>
</tr>
<tr>
<td>LOI</td>
<td>100</td>
<td>s.size =&gt; -s.size</td>
</tr>
<tr>
<td>LOI</td>
<td>101</td>
<td>i =&gt; ~i</td>
</tr>
<tr>
<td>AORS</td>
<td>87</td>
<td>i++ =&gt; i--</td>
</tr>
<tr>
<td>AORS</td>
<td>93</td>
<td>i++ =&gt; i--</td>
</tr>
<tr>
<td>AORS</td>
<td>100</td>
<td>i++ =&gt; i--</td>
</tr>
</tbody>
</table>
BinomialHeap(Merge).

For the *merge* method, the following are the mutation operator applied:

- **AORb**: Arithmetic Operator Replacement(binary)
- **AOIu**: Arithmetic Operator Deletion(unary)
- **AOIs**: Arithmetic Operator Deletion(short-cut)
- **ROR**: Relational Operator Replacement
- **COR**: Conditional Operator Replacement
- **COD**: Conditional Operator Deletion
- **LOI**: Logical Operator Insertion

The Generated mutants are showed in the table below. We indicated, together with the mutation, the operator applied and the line number where the mutation is placed in the code (see Appendix above).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Line Number</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AORB</td>
<td>38</td>
<td>this.size + h2.size =&gt; this.size * h2.size</td>
</tr>
<tr>
<td>AORB</td>
<td>38</td>
<td>this.size + h2.size =&gt; this.size / h2.size</td>
</tr>
<tr>
<td>AORB</td>
<td>38</td>
<td>this.size + h2.size =&gt; this.size % h2.size</td>
</tr>
<tr>
<td>AORB</td>
<td>38</td>
<td>this.size + h2.size =&gt; this.size - h2.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>38</td>
<td>this.size =&gt; ++this.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>38</td>
<td>this.size =&gt; --this.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>38</td>
<td>this.size =&gt; this.size++</td>
</tr>
<tr>
<td>AOIS</td>
<td>38</td>
<td>this.size =&gt; this.size--</td>
</tr>
<tr>
<td>AOIS</td>
<td>38</td>
<td>h2.size =&gt; ++h2.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>38</td>
<td>h2.size =&gt; --h2.size</td>
</tr>
<tr>
<td>AOIS</td>
<td>38</td>
<td>h2.size =&gt; h2.size++</td>
</tr>
<tr>
<td>AOIS</td>
<td>38</td>
<td>h2.size =&gt; h2.size--</td>
</tr>
<tr>
<td>AOIS</td>
<td>50</td>
<td>x.degree =&gt; ++x.degree</td>
</tr>
<tr>
<td>AOIS</td>
<td>50</td>
<td>x.degree =&gt; --x.degree</td>
</tr>
<tr>
<td>AOIS</td>
<td>50</td>
<td>x.degree =&gt; x.degree++</td>
</tr>
<tr>
<td>AOIS</td>
<td>50</td>
<td>x.degree =&gt; x.degree--</td>
</tr>
<tr>
<td>AOIS</td>
<td>50</td>
<td>nextX.degree =&gt; ++nextX.degree</td>
</tr>
<tr>
<td>AOIS</td>
<td>50</td>
<td>nextX.degree =&gt; --nextX.degree</td>
</tr>
<tr>
<td>AOIS</td>
<td>50</td>
<td>nextX.degree =&gt; nextX.degree++</td>
</tr>
<tr>
<td>AOIS</td>
<td>50</td>
<td>nextX.degree =&gt; nextX.degree--</td>
</tr>
</tbody>
</table>
nextX.sibling.degree => ++nextX.sibling.degree
nextX.sibling.degree => --nextX.sibling.degree
nextX.sibling.degree => nextX.sibling.degree++
nextX.sibling.degree => nextX.sibling.degree--
x.degree => ++x.degree
x.degree => --x.degree
x.degree => x.degree++
x.degree => x.degree--
x.key => ++x.key
x.key => --x.key
x.key => x.key++
x.key => x.key--
nextX.key => ++nextX.key
nextX.key => --nextX.key
nextX.key => nextX.key++
nextX.key => nextX.key--
h1.Nodes.degree => ++h1.Nodes.degree
h1.Nodes.degree => --h1.Nodes.degree
h1.Nodes.degree => h1.Nodes.degree++
h1.Nodes.degree => h1.Nodes.degree--
h2.Nodes.degree => ++h2.Nodes.degree
h2.Nodes.degree => --h2.Nodes.degree
h2.Nodes.degree => h2.Nodes.degree++
h2.Nodes.degree => h2.Nodes.degree--
h1Next.degree => ++h1Next.degree
h1Next.degree => --h1Next.degree
h1Next.degree => h1Next.degree++
h1Next.degree => h1Next.degree--
h2Next.degree => ++h2Next.degree
h2Next.degree => --h2Next.degree
h2Next.degree => h2Next.degree++
h2Next.degree => h2Next.degree--
nextX != null => h1.Nodes != null
nextX != null => nextX == null
x.degree != nextX.degree =>
x.degree > nextX.degree
x.degree != nextX.degree =>
x.degree >= nextX.degree
x.degree != nextX.degree =>
x.degree >= nextX.degree
x.degree != nextX.degree =>
x.degree < nextX.degree
x.degree != nextX.degree =>
x.degree <= nextX.degree
x.degree != nextX.degree =>
<table>
<thead>
<tr>
<th>ROR</th>
<th>50</th>
<th>x.degrees == nextX.degrees</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROR</td>
<td>50</td>
<td>nextX.sibling != null =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nextX.sibling == null</td>
</tr>
<tr>
<td>ROR</td>
<td>50</td>
<td>nextX.sibling.degree == x.degrees =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nextX.sibling.degree &gt; x.degrees</td>
</tr>
<tr>
<td>ROR</td>
<td>50</td>
<td>nextX.sibling.degree == x.degrees =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nextX.sibling.degree &gt;= x.degrees</td>
</tr>
<tr>
<td>ROR</td>
<td>50</td>
<td>nextX.sibling.degree == x.degrees =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nextX.sibling.degree &lt; x.degrees</td>
</tr>
<tr>
<td>ROR</td>
<td>50</td>
<td>nextX.sibling.degree == x.degrees =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nextX.sibling.degree &lt;= x.degrees</td>
</tr>
<tr>
<td>ROR</td>
<td>64</td>
<td>prevX == null =&gt; prevX != null</td>
</tr>
<tr>
<td>ROR</td>
<td>104</td>
<td>h1.Nodes == null =&gt; h1.Nodes != null</td>
</tr>
<tr>
<td>ROR</td>
<td>106</td>
<td>h2.Nodes == null =&gt; h2.Nodes != null</td>
</tr>
<tr>
<td>ROR</td>
<td>117</td>
<td>h1.Nodes.degree &lt;= h2.Nodes.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1.Nodes.degree &gt; h2.Nodes.degree</td>
</tr>
<tr>
<td>ROR</td>
<td>117</td>
<td>h1.Nodes.degree &lt;= h2.Nodes.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1.Nodes.degree &gt;= h2.Nodes.degree</td>
</tr>
<tr>
<td>ROR</td>
<td>117</td>
<td>h1.Nodes.degree &lt;= h2.Nodes.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1.Nodes.degree &lt; h2.Nodes.degree</td>
</tr>
<tr>
<td>ROR</td>
<td>117</td>
<td>h1.Nodes.degree &lt;= h2.Nodes.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1.Nodes.degree == h2.Nodes.degree</td>
</tr>
<tr>
<td>ROR</td>
<td>117</td>
<td>h1.Nodes.degree &lt;= h2.Nodes.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1.Nodes.degree != h2.Nodes.degree</td>
</tr>
<tr>
<td>ROR</td>
<td>129</td>
<td>h1Next != null =&gt; h1Next == null</td>
</tr>
<tr>
<td>ROR</td>
<td>129</td>
<td>h2Next != null =&gt; h2Next == null</td>
</tr>
<tr>
<td>ROR</td>
<td>130</td>
<td>h1Next.degree &lt;= h2Next.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1Next.degree &gt; h2Next.degree</td>
</tr>
<tr>
<td>ROR</td>
<td>130</td>
<td>h1Next.degree &lt;= h2Next.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1Next.degree &gt;= h2Next.degree</td>
</tr>
<tr>
<td>ROR</td>
<td>130</td>
<td>h1Next.degree &lt;= h2Next.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1Next.degree &lt; h2Next.degree</td>
</tr>
<tr>
<td>ROR</td>
<td>130</td>
<td>h1Next.degree &lt;= h2Next.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1Next.degree == h2Next.degree</td>
</tr>
<tr>
<td>ROR</td>
<td>130</td>
<td>h1Next.degree &lt;= h2Next.degree =&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h1Next.degree != h2Next.degree</td>
</tr>
<tr>
<td>COR</td>
<td>50</td>
<td>nextX.sibling != null &amp;&amp;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nextX.sibling.degree == x.degrees =&gt;</td>
</tr>
<tr>
<td>COR</td>
<td>50</td>
<td>nextX.sibling != null</td>
</tr>
<tr>
<td></td>
<td></td>
<td>nextX.sibling.degree == x.degrees =&gt;</td>
</tr>
<tr>
<td>Line</td>
<td>Condition</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td><code>nextX.sibling != null ^ nextX.sibling.degree == x.degree</code></td>
<td></td>
</tr>
<tr>
<td>COR 50</td>
<td>`x.degree != nextX.degree</td>
<td></td>
</tr>
<tr>
<td>COR 50</td>
<td>`x.degree != nextX.degree</td>
<td></td>
</tr>
<tr>
<td>COR 129</td>
<td>`h1Next != null &amp;&amp; h2Next != null =&gt; h1Next != null</td>
<td></td>
</tr>
<tr>
<td>COR 129</td>
<td><code>h1Next != null &amp;&amp; h2Next != null =&gt; h1Next != null ^ h2Next != null</code></td>
<td></td>
</tr>
<tr>
<td>COI 42</td>
<td><code>h.Nodes == null =&gt; !(h.Nodes == null)</code></td>
<td></td>
</tr>
<tr>
<td>COI 49</td>
<td><code>nextX != null =&gt; !(nextX != null)</code></td>
<td></td>
</tr>
<tr>
<td>COI 50</td>
<td><code>x.degree != nextX.degree =&gt; !(x.degree != nextX.degree)</code></td>
<td></td>
</tr>
<tr>
<td>COI 50</td>
<td><code>nextX.sibling != null =&gt; !(nextX.sibling != null)</code></td>
<td></td>
</tr>
<tr>
<td>COI 50</td>
<td><code>nextX.sibling.degree == x.degree =&gt; !(nextX.sibling.degree == x.degree)</code></td>
<td></td>
</tr>
<tr>
<td>COI 50</td>
<td><code>nextX.sibling != null &amp;&amp; nextX.sibling.degree == x.degree =&gt; !(nextX.sibling != null &amp;&amp; nextX.sibling.degree == x.degree)</code></td>
<td></td>
</tr>
<tr>
<td>COI 50</td>
<td>`x.degree != nextX.degree</td>
<td></td>
</tr>
<tr>
<td>COI 57</td>
<td><code>x.key &lt; nextX.key =&gt; !(x.key &lt; nextX.key)</code></td>
<td></td>
</tr>
<tr>
<td>COI 64</td>
<td><code>prevX == null =&gt; !(prevX == null)</code></td>
<td></td>
</tr>
<tr>
<td>COI 104</td>
<td><code>h1.Nodes == null =&gt; !(h1.Nodes == null)</code></td>
<td></td>
</tr>
<tr>
<td>COI 106</td>
<td><code>h2.Nodes == null =&gt; !(h2.Nodes == null)</code></td>
<td></td>
</tr>
<tr>
<td>COI 117</td>
<td><code>h1.Nodes.degree &lt;= h2.Nodes.degree =&gt; !(h1.Nodes.degree &lt;= h2.Nodes.degree)</code></td>
<td></td>
</tr>
<tr>
<td>COI 129</td>
<td><code>h1Next != null =&gt; !(h1Next != null)</code></td>
<td></td>
</tr>
<tr>
<td>COI 129</td>
<td><code>h2Next != null =&gt; !(h2Next != null)</code></td>
<td></td>
</tr>
<tr>
<td>COI 129</td>
<td><code>h1Next != null &amp;&amp; h2Next != null =&gt; !(h1Next != null &amp;&amp; h2Next != null)</code></td>
<td></td>
</tr>
<tr>
<td>COI 130</td>
<td><code>h1Next.degree &lt;= h2Next.degree =&gt; !(h1Next.degree &lt;= h2Next.degree)</code></td>
<td></td>
</tr>
<tr>
<td>COI 129</td>
<td><code>h1Next != null =&gt; !(h1Next != null)</code></td>
<td></td>
</tr>
<tr>
<td>LOI 38</td>
<td><code>this.size =&gt; -this.size</code></td>
<td></td>
</tr>
<tr>
<td>LOI 38</td>
<td><code>h2.size =&gt; -h2.size</code></td>
<td></td>
</tr>
</tbody>
</table>
SearchTree(Delete).

For the delete method, the following are the mutation operator applied:

- AORs: Arithmetic Operator Replacement (short-cut)
- ROR: Relational Operator Replacement
- COR: Conditional Operator Replacement
- COI: Conditional Operator Insertion

The Generated mutants are showed in the table below. We indicated, together with the mutation, the operator applied and the line number where the mutation is placed in the code (see Appendix above).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Line Number</th>
<th>Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AORS</td>
<td>349</td>
<td>size-- =&gt; size++</td>
</tr>
<tr>
<td>AORS</td>
<td>352</td>
<td>size-- =&gt; size++</td>
</tr>
<tr>
<td>AORS</td>
<td>378</td>
<td>size-- =&gt; size++</td>
</tr>
<tr>
<td>ROR</td>
<td>333</td>
<td>t == null =&gt; t != null</td>
</tr>
<tr>
<td>ROR</td>
<td>343</td>
<td>t.left != null =&gt; t.left == null</td>
</tr>
<tr>
<td>ROR</td>
<td>343</td>
<td>t.right != null =&gt; t.right == null</td>
</tr>
<tr>
<td>ROR</td>
<td>348</td>
<td>t.left != null =&gt; t.left == null</td>
</tr>
<tr>
<td>ROR</td>
<td>369</td>
<td>t == null =&gt; t != null</td>
</tr>
<tr>
<td>ROR</td>
<td>374</td>
<td>t.left != null =&gt; t.left == null</td>
</tr>
<tr>
<td>ROR</td>
<td>390</td>
<td>t != null =&gt; t == null</td>
</tr>
<tr>
<td>ROR</td>
<td>391</td>
<td>t.left != null =&gt; t.left == null</td>
</tr>
<tr>
<td>COR</td>
<td>343</td>
<td>t.left != null &amp;&amp; t.right != null =&gt; t.left != null</td>
</tr>
<tr>
<td>COR</td>
<td>343</td>
<td>t.left != null &amp;&amp; t.right != null =&gt; t.left != null ^ t.right != null</td>
</tr>
<tr>
<td>COI</td>
<td>Line</td>
<td>Condition</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td>COI</td>
<td>333</td>
<td>t == null =&gt; !(t == null)</td>
</tr>
<tr>
<td>COI</td>
<td>337</td>
<td>param.intValue() &lt; t.info =&gt; !(param.intValue() &lt; t.info)</td>
</tr>
<tr>
<td>COI</td>
<td>340</td>
<td>param.intValue() &gt; t.info =&gt; !(param.intValue() &gt; t.info)</td>
</tr>
<tr>
<td>COI</td>
<td>343</td>
<td>t.left != null =&gt; !(t.left != null)</td>
</tr>
<tr>
<td>COI</td>
<td>343</td>
<td>t.right != null =&gt; !(t.right != null)</td>
</tr>
<tr>
<td>COI</td>
<td>343</td>
<td>t.left != null &amp;&amp; t.right != null =&gt; !(t.left != null &amp;&amp; t.right != null)</td>
</tr>
<tr>
<td>COI</td>
<td>348</td>
<td>t.left != null =&gt; !(t.left != null)</td>
</tr>
<tr>
<td>COI</td>
<td>369</td>
<td>t == null =&gt; !(t == null)</td>
</tr>
<tr>
<td>COI</td>
<td>374</td>
<td>t.left != null =&gt; !(t.left != null)</td>
</tr>
<tr>
<td>COI</td>
<td>390</td>
<td>t != null =&gt; !(t != null)</td>
</tr>
<tr>
<td>COI</td>
<td>391</td>
<td>t.left != null =&gt; !(t.left != null)</td>
</tr>
</tbody>
</table>