# Incorporating Coverage Criteria in Bounded Exhaustive Test Generation of Structural Inputs

Nazareno Aguirre[1], Valeria Bengolea[1], Marcelo Frias[2], and Juan Pablo Galeotti[3]

[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
[2] Departamento de Ingeniería Informática, Instituto Tecnológico Buenos Aires and CONICET, Buenos Aires, Argentina. E-mail: mfrias@itba.edu.ar
[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 [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* [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, graphs, 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 mechanisms 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 together 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 generating 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 criterion (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):

```
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 representation 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 *finitisation* 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/maximum 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 corresponding two fields of the four nodes that the list might at most contain. For instance, tuple

$$\langle 0, 0, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL}, \mathit{NULL} \rangle$$

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 $\langle 1, 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle$ 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 follows: it starts with the initial candidate vector, with all indices in zero. It

then executes `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 `repOk()`, it is considered a valid test case (in this case, all reachable fields must be in the stack of accessed fields). If `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 `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 `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 `repOk()` would not change due to its determinism, (i.e., the parts of the structure visited by `repOk()` would remain the same, and therefore `repOk()` would fail again).

Besides the described search mechanism, with its incorporated search pruning, Korat also avoids generating isomorphic candidates [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 invokation of `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 invokation of `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.

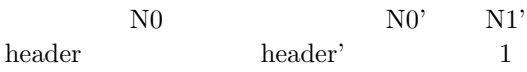
We present the approach by implementing it as a variant of the Korat algorithm, introduced in the previous section. In the same way that Korat requires an imperative predicate `repOk()`, we require a routine that we call `eqClass()`. This routine returns, given a valid candidate (i.e., a candidate satisfying `repOk()`), the equivalence class the candidate corresponds to, according to a test criterion. As for `repOk()`, this routine must be deterministic (for exactly the same reason that `repOk()` must be deterministic). As opposed to Korat, which prunes (advances various candidates at once) only when `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 `repOk()` succeedes and when it fails: when `repOk()` fails, we advance various candidates at once using the fact that if none of the accessed fields is changed, then `repOk()` would fail again. When `repOk()` succeedes, we execute `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 `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 `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):




Let us refer to this pair of lists as `(l,s)`. Clearly, `(l,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., `N1'.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 $\langle T, F, T, F \rangle$ (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` (`N0`), `N0.next`, `header'` (`N0'`), `N0'.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 `(l,s)`, which is already covered by this valid candidate. So, the approach "prunes" the search by attempting to advance the last accessed field, namely `N0'.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 `N0.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]++;
          }
        }
```

```
      }
  } while (!fields.isEmpty())
}
```

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.

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

| Scope | Korat | Korat+ |
|---|---|---|
| 0-4,0-4,4,4,3 | 0,422s | 0,269s |
| 0-5,0-5,5,5,4 | 2,39s | 0,395s |
| 0-6,0-6,6,6,5 | 329,809s | 0,817s |
| 0-7,0-7,7,7,6 | TIMEOUT | 3,36s |

**Fig. 1.** ListAsSet

*Binomial Heap (Merge).* 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 `repOk()`, 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.

| Scope | Korat | Korat+ | CC |
|---|---|---|---|
| 2 | 348(36) | 147(15) | 6 |
| 3 | 5,389(784) | 1,315(56) | 10 |
| 4 | 150,448(14,400) | 46,786(435) | 10 |
| 5 | 3,125,314(876,096) | 647,410(1,872) | 10 |
| 6 | 274,808,123(57,790,404) | 55,745,855(43,134) | 10 |

| Scope | Korat | Korat+ |
|---|---|---|
| 2 | 0,42s | 0,394s |
| 3 | 0,656s | 0,454s |
| 4 | 1,436s | 0,86s |
| 5 | 16,347s | 2,492s |
| 6 | 1360,323s | 178,072s |

**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.

| Scope | Korat | Korat+ | CC |
|---|---|---|---|
| 2 | 1,624(382) | 518(126) | 4 |
| 3 | 372,861,255(47,672,840) | 11,670,154(899,852) | 4 |
| 4 | TIMEOUT | TIMEOUT | |

| Scope | Korat | Korat+ |
|---|---|---|
| 2 | 0,343s | 0,265s |
| 3 | 1145,341s | 37,854s |

**Fig. 3.** Directed Graphs

*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.

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

| Scope | Korat | Korat+ |
|---|---|---|
| 2,-2,2 | 0,632s | 0,534s |
| 3,-3,3 | 2812,665s | 1333,942s |

**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.

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

| Scope | Korat | Korat+ |
|---|---|---|
| 3,0-3,6 | 0,423s | 0,359s |
| 4,0-4,7 | 0,88s | 0,748s |
| 5,0-5,8 | 1,607s | 1,333s |
| 6,0-6,9 | 7,529s | 5,724s |

**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.

| Scope | Korat/Korat+ | CC |
|---|---|---|
| 0,3,4,3 | 319(40) | 4 |
| 0,4,5,4 | 3,388(341) | 4 |
| 0,5,6,5 | 46,684(3,906) | 4 |
| 0,6,7,6 | 781,960(55,987) | 4 |
| 0,7,8,7 | 15,349,933(960,804) | 4 |

### 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.

| Scope | Korat | Korat+ | CC |
|---|---|---|---|
| 3,0-3,3 | 552(36) | 532(34) | 10 |
| 3,0-3,6 | 2,598(462) | 2,333(350) | 18 |
| 4,0-4,4 | 3,300(116) | 3,205(102) | 19 |
| 4,0-4,8 | 30,408(3,656) | 25,202(2,272) | 27 |
| 5,0-5,5 | 19,035(370) | 18,461(300) | 27 |
| 6,0-6,6 | 106,650(1,206) | 102,962 (892) | 29 |
| 7,0-7,7 | 588,203(4,011) | 566,075(2,695) | 31 |
| 5,0-5,10 | 375,620(31,970) | 282,067(16,378) | 29 |
| 8,0-8,8 | 3,027,008(13,384) | 2,912,167(8,143) | 47 |
| 8,0-8,11 | 21,026,038(362,890) | 16,714,960(160,483) | 47 |
| 9,0-9,9 | 14,944,140(43,992) | 14,422,677(24,257) | 47 |
| 9,0-9,11 | 48,226,046(417,340) | 41,675,314(187,511) | 49 |
| 10,0-10,10 | 75,236,610(141,010) | 73,048,869(70,877) | 49 |
| 11,0-11,11 | 394,857,826(440,814) | 385,920,181(203,605) | 49 |

| Scope | Korat | Korat+ |
|---|---|---|
| 3,0-3,3 | 0,388s | 0,366s |
| 4,0-4,4 | 0,517s | 0,442s |
| 5,0-5,5 | 0,701s | 0,736s |
| 6,0-6,6 | 1,275s | 1,143s |
| 7,0-7,7 | 2,58s | 2,474s |
| 8,0-8,8 | 10,316s | 9,709s |
| 9,0-9,9 | 52,887s | 49,007s |
| 10,0-10,10 | 281,931s | 271,16s |

**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.

| Scope | Korat | Korat+ | CC |
|---|---|---|---|
| 3,0-3,3 | 552(36) | 532(34) | 7 |
| 3,0-3,6 | 2,698(462) | 2,333(350) | 7 |
| 4,0-4,4 | 3,300(116) | 3,205(102) | 8 |
| 4,0-4,8 | 30,408(3,656) | 25,202(2,272) | 8 |
| 5,0-5,5 | 19,035(370) | 18,461(300) | 8 |
| 5,0-5,10 | 375,620(31,970) | 282,067(16,378) | 8 |
| 6,0-6,12 | 4,482,228(284,220) | 3,071,411(123,490) | 8 |
| 7,0-7,14 | 52,105,144(2,551,486) | 32,878,324(956,280) | 8 |
| 8,0-8,8 | 3,027,008(13,384) | 2,912,167(8,143) | 8 |

| Scope | Korat | Korat+ |
|---|---|---|
| 3,0-3,6 6 | 0,462s | 0,452s |
| 4,0-4,8 | 1,082s | 1,098s |
| 5,0-5,10 | 2,167s | 1,876s |
| 6,0-6,12 | 15,674s | 10,135s |
| 7,0-7,14 | 180,46s | 109,814s |

**Fig. 7.** Search in Red Black Trees

*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.

| Scope | Korat | Korat+ | CC |
|---|---|---|---|
| 5,0-5,5 | 41,540(940) | 38,040(767) | 9 |
| 5,0-5,10 | 1,720,830(142,250) | 1,076,259(68,258) | 9 |
| 6,0-6,6 | 371,700(4,386) | 338,985(3,290) | 9 |
| 6,0-6,12 | 32,702,676(1,960,884) | 18,830,994(801,733) | 9 |
| 7,0-7,7 | 3,301,956(20,650) | 3,015,006(14,280) | 9 |
| 7,0-7,14 | 601,871,102(27,563,746) | 323,780,633(9,803,030) | 9 |
| 8,0-8,8 | 29,007,816(97,880) | 26,601,485(62,624) | 9 |

| Scope | Korat | Korat+ |
|---|---|---|
| 5,0-5,10 | 3,589 | 2,636 |
| 6,0-6,12 | 60,802 | 32,653 |
| 7,0-7,14 | 1175,809 | 564,917 |

**Fig. 8.** Insertion in Search Trees

*Search in Search Trees (Decision coverage).* 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.

| Scope | Korat | Korat+ | CC |
|---|---|---|---|
| 2,0-2,2 | 62(10) | 57(10) | 6 |
| 2,0-2,4 | 228(68) | 198(60) | 7 |
| 3,0-3,3 | 534(45) | 500(43) | 8 |
| 3,0-3,6 | 4,290(822) | 3,331(586) | 8 |
| 4,0-4,4 | 4,660(204) | 4,314(181) | 8 |
| 4,0-4,8 | 87,032(10,600) | 60,102(6,121) | 8 |
| 5,0-5,5 | 41,540(940) | 38,040(767) | 8 |
| 5,0-5,10 | 1,720,830(142,250) | 1,076,259(68,258) | 8 |
| 6,0-6,6 | 371,700(4,386) | 338,985(3,290) | 8 |
| 6,0-6,12 | 32,702,676(1,960,884) | 18,830,994(801,733) | 8 |
| 7,0-7,7 | 3,301,956(20,650) | 3,015,006(14,280) | 8 |

| Scope | Korat | Korat+ |
|---|---|---|
| 2,0-2,4 | 0,262s | 0,264s |
| 3,0-3,6 | 0,425s | 0,352s |
| 4,0-4,8 | 0,993s | 1,019s |
| 5,0-5,10 | 3,427s | 2,643s |
| 6,0-6,12 | 56,518s | 31,852s |

**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 pruningthat our technique contributes to standard filtering. In our cases, these numbers reflect directly in running times, because our `eqClass()` routines, the most influencial (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 `repOk()` and `eqClass()`, and how these relate to each other. For instance, in cases in which `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 `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 `repOK()` fails very often, i.e., when the conditions for valid structures are stronger, then Korat exploits its associated pruning mechanism. It is when `repOK()` succedes 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 (`repOK()` succeedes 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 `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.

| List as Set (49 mutants) | | | |
|---|---|---|---|
| Scope | Korat | Korat+ | One Per Class |
| 0-2,0-2,3,3,3 | 3 | 9 | 15 |
| 0-4,0-4,3,3,3 | 3 | 9 | 15 |
| 0-4,0-4,4,4,3 | 3 | 9 | 11 |
| 0-5,0-5,5,5,4 | 3 | 9 | 10 |
| 0-5,0-5,5,5,5 | 3 | 9 | 10 |

| Binomial Heaps (117 mutants) | | | |
|---|---|---|---|
| Scope | Korat | Korat+ | One Per Class |
| 2 | 38 | 39 | 44 |
| 3 | 8 | 8 | 17 |
| 4 | 7 | 7 | 17 |
| 5 | 7 | 7 | 17 |
| 6 | 7 | 7 | 17 |

| Search Trees (24 mutants) | | | |
|---|---|---|---|
| Scope | Korat | Korat+ | One Per Class |
| 3,0-3,3 | 2 | 2 | 2 |
| 3,0-3,4 | 2 | 2 | 2 |
| 3,0-3,6 | 2 | 2 | 2 |
| 3,0-3,8 | 2 | 2 | 2 |
| 5,0-5,8 | 0 | 0 | 2 |
| 6,0-6,9 | 0 | 0 | 2 |

## 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, compared 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 European Conference on Model-Driven Software Engineering, 2003.
2. B. Beizer, *Black-Box Testing: Techniques for Functional Testing of Software and Systems*, Wiley, 1995.
3. C. Boyapati, S. Khurshid and D. Marinov, *Korat: Automated Testing based on Java Predicates*, in Proceedings of International Symposium on Software Testing and Analysis ISSTA 2002, ACM Press, 2002.
4. J. P. Galeotti, N. Rosner, C. López Pombo and M. Frias, *Analysis of invariants for efficient bounded verification*, in Proceedings of the 19th International Symposium on Software Testing and Analysis ISSTA 2010, Trento, Italy, ACM Press, 2010.
5. C. Ghezzi, M. Jazayeri and D. Mandiroli, *Fundamentals of Software Engineering*, Second Edition, Prentice-Hall, 2003.
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 Africa, ACM Press, 2010.
7. D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2006.
8. C. Kaner, J. Bach and B. Pettichord, *Lessons Learned in Software Testing*, Wiley, 2001.
9. S. Khurshid and D. Marinov, *TestEra: Specification-Based Testing of Java Programs Using SAT*, Automated Software Engineering 11(4), Springer, 2004.
10. B. Liskov and J. Guttag, *Program Development in Java: Abstraction, Specification and Object-Oriented Design*, Addison-Wesley, 2000.
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 Conference 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 System*, Journal of Software Testing, Verification and Reliability, 15(2), Wiley, 2005.
13. A. Milicevic, S. Misailovic, D. Marinov and S. Khurshid, *Korat: A Tool for Generating Structurally Complex Test Inputs*, in Proceedings of International Conference on Software Engineering ICSE 2007, IEEE Press, 2007.
14. MuJava Home Page, `http://www.cs.gmu.edu/ offutt/mujava/`.
15. K. Sen, D. Marinov and G. Agha, *CUTE: A Concolic Unit Testing Engine for C*, in Proceedings of the 5th joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering ESEC/FSE 2005, ACM Press, 2005.

16. J. Siddiqui and S. Khurshid, *An Empirical Study of Structural Constraint Solving Techniques*, in Proceedings of the 11th International Conference on Formal Engineering Methods ICFEM 2009, LNCS, Springer, 2009.
17. J. Siddiqui and S. Khurshid, *PKorat: Parallel Generation of Structurally Complex Test Inputs*, in Proceedings of the 2nd International Conference on Software Testing Verification and Validation ICST 2009, IEEE Computer Society, 2009.
18. W. Visser, C. Pasareanu and S. Khurshid, *Test Input Generation with Java PathFinder*, in Proceedings of International Symposium on Software Testing and Analysis ISSTA 2004, ACM Press, 2004.
19. T. Xie, D. Marinov and D. Notkin, *Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests*, in Proceedings of the 19th IEEE International Conference on Automated Software Engineering ASE 2004, Linz, Austria, IEEE Computer Society, 2004.
20. H. Zhu, P. Hall and J. May, *Software Unit Test Coverage and Adequacy*, ACM Computing Surveys 29(4), ACM Press, 1997.

# Apendix

## ListAsSet

```
1  package koratPlus.ListAsSet;
2
3  import korat.finitization.IClassDomain;
4  import korat.finitization.IFinitization;
5  import korat.finitization.IIntSet;
6  import korat.finitization.IObjSet;
7  import korat.finitization.impl.FinitizationFactory;
8  import java.math.BigInteger;
9
10 import java.io.*;
11
12 /**
13  * Class ListAsSet defines the set of object needed to test a procedure
14  * listToSet that given a list  * 'l' and a set 's' determines whether
15  * 's'  is the result of converting 'l' to a set, i.e  disregarding
16  * repetitions and the order of element in 'l'.
17  * This class needs to  implement the Serializable interface since it
18  * is required for the korat "serialize" option.
19  * @author Nazareno M. Aguirre, Valeria Bengolea.
20  */
21 public class ListAsSet implements Serializable{
22
23     public  static final long serialVersionUID = 1;
24
25     public SinglyLinkedList list;
26     public StrictlySortedSinglyLinkedList set;
27
28
29     /**
30      * RepOk checks whether 'list' and 'set'  satisfy their
31      * representation invariant.
32      * @return true iff 'list' and 'set' of the current instance
33      * satisfy their representation invariant,  false otherwise.
34      */
35     public boolean repOK() {
36         if (!list.repOK())
37             return false;
38         return set.repOK();
39     }
```

```
40
41      /** eqClass returns the equivalence class in which the current
42       *  instance of ListAsSet belongs to. In this case, the equivalence
43       *  classes are given by all the possible combinations of predicates
44       *  isEmpty(), sorted() and noReps() over the list; and isEmpty()
45       *  over the set.
46       * @return The equivalence class in which this instance belongs
47       */
48      public BigInteger eqClass() {
49          boolean[] classes = new boolean[4];
50          classes[0] = list.isEmpty();
51          classes[1] = list.noReps();
52          classes[2] = list.sorted();
53          classes[3] = set.isEmpty();
54
55          return getClass(classes);
56      }
57
58      /**getClass translates a given array of boolean into a Integer
59       * @param classes The array to be translated
60       * @return The integer that represents the given array
61       * @see eqClass()
62       */
63      public BigInteger getClass(boolean [] classes){
64          BigInteger res = new BigInteger("0");
65          BigInteger dos =new BigInteger("2");
66          for (int i=0; i<classes.length; i++) {
67              if (classes[i]){
68                  BigInteger t = dos.pow(i);
69                  res = res.add(t);
70              }
71          }
72          return res;
73
74      }
75
76      /**
77       * listToSet checks whether 'set' is the result of converting 'list'
78       * to a set.
79       * @return true iff 'set' is the result of converting 'list' to a
80       * set, i.e disregarding repetitions and the order of elements of
81       * 'list'
82       */
83      public boolean listToSet(){
84
85          /*converts the list in to a set*/
86          StrictlySortedSinglyLinkedList s = new
                StrictlySortedSinglyLinkedList();
87          for(int i=0; i<list.size; i++){
88              Integer value = list.get(i);
89              this.addToSet(s, value);
90          }
91
92          /*checks wether the to set are equals*/
93          for(int i = 0; i < set.size; i++ ){
94              Integer value = set.get(i);
95              if(!(s.contains(value))){
96
97                  return false;
98              }
99          }
100         for(int i = 0; i < s.size; i++ ){
101             Integer value = s.get(i);
102             if(!(set.contains(value))){
103                 return false;
104             }
105         }
106         return true;
```

```
107
108         }
109
110
111         /**
112          * addToSet adds the given value to the given set.
113          * @param set , The set to add value into
114          * @param value, the value to be added into the given set.
115          */
116         public void addToSet(StrictlySortedSinglyLinkedList set, Integer
                value){
117             Node current = set.header.next;
118             Node previous = set.header;
119
120             while(current!=null && current.element.intValue()< value.
                    intValue()){
121                 previous = current;
122                 current = current.next;
123             }
124             Node n = new Node();
125             n.element =  value;
126             if (current == null) {
127                 previous.next = n;
128                 set.size++;
129             }
130             else{
131                 if (current.element.intValue() != value.intValue()) {
132                     previous.next = n;
133                     n.next = current;
134                     set.size++;
135                 }
136
137             }
138         }
139
140         /**
141          * finListAsSet provides a bound on the number of objects to be used
142          * to generate instances of ListAsSet.
143          * @param minSizeList minimum size of the generated lists
144          * @param maxSizeList maximum size of the generated lists
145          * @param minSizeSet minimum size of the generated sets
146          * @param maxSizeSet maximum size of the generated sets
147          * @param numEntriesList number of entries that the list may contain
148          * @param numEntriesSet number of entries that the set may contain
149          * @param numElems The range of the elements contained in each entry
150          * of both, the list and the set,  is between [1,..numElems].
151          * @return the object Korat needs for setting up the bounds during
                 the search.
152          */
153         public static IFinitization finListAsSet(int minSizeList, int
                maxSizeList, int minSizeSet, int maxSizeSet,
154                     int numEntriesList, int numEntriesSet,int numElems) {
155
156
157             IFinitization f = FinitizationFactory.create(ListAsSet.class);
158
159             IObjSet entriesList = f.createObjSet(SinglyLinkedList.class,
                        false);
160             IObjSet entriesSet = f.createObjSet(
                        StrictlySortedSinglyLinkedList.class, false);
161
162             entriesList.addClassDomain(f.createClassDomain(SinglyLinkedList.
                        class, 1));
163             entriesSet.addClassDomain(f.createClassDomain(
                        StrictlySortedSinglyLinkedList.class, 1));
164
165             IObjSet entries = f.createObjSet(Entry.class, true);
```

```
166             entries.addClassDomain(f.createClassDomain(Entry.class,
                    numEntriesList));
167             IIntSet sizesList = f.createIntSet(minSizeList, maxSizeList);
168 
169             IObjSet nodes = f.createObjSet(Node.class, true);
170             nodes.addClassDomain(f.createClassDomain(Node.class,
                    numEntriesSet));
171             IIntSet sizesSet = f.createIntSet(minSizeSet, maxSizeSet);
172 
173             IObjSet elems = f.createObjSet(Integer.class);
174             IClassDomain elemsClassDomain = f.createClassDomain(Integer.
                    class);
175             elemsClassDomain.includeInIsomorphismCheck(false);
176 
177             for (int i = 1; i <= numElems; i++)
178                 elemsClassDomain.addObject(new Integer(i));
179             elems.addClassDomain(elemsClassDomain);
180             elems.setNullAllowed(true);
181 
182             f.set("list",entriesList);
183             f.set("set",entriesSet);
184 
185             f.set("SinglyLinkedList.header", entries);
186             f.set("StrictlySortedSinglyLinkedList.header", nodes);
187 
188             f.set("SinglyLinkedList.size", sizesList);
189             f.set("StrictlySortedSinglyLinkedList.size", sizesSet);
190 
191 
192             f.set(Entry.class, "element", elems);
193             f.set(Entry.class, "next", entries);
194 
195             f.set(Node.class, "element", elems);
196             f.set(Node.class, "next", nodes);
197 
198             return f;
199         }
200 
201 
202     }
```

```
1   package koratPlus.ListAsSet;
2   
3   import java.io.Serializable;
4   import java.util.Set;
5   
6   
7   /**
8    * Class  StrictlySortedSinglyLinkedList defines Strictly Sorted, Singly
9    * linked List.
10   * This class needs to implement the Serializable interface since it is
11   * required for the korat "serialize" option.
12   * @author
13   */
14  public class StrictlySortedSinglyLinkedList implements Serializable{
15  
16      public  static final long serialVersionUID = 1;
17  
18      public Node header;
19      public int size = 0;
20  
21      public StrictlySortedSinglyLinkedList(){
22          header = new Node();
23          size = 0;
24      }
25  
26      /**
27       * This method checks whether a given Integer value belongs to the
```

```
28          * current list.
29          * @param value The element whose presence in this list is to be
30          * tested.
31          * @return true iff value is in the current list.
32          */
33         public boolean contains(Integer value){
34             Node current = header.next;
35             while(current!=null && current.element.intValue()<= value.
                   intValue()){
36                 if(current.element.intValue()==value.intValue())
37                     return  true;
38                 current = current.next;
39             }
40             return false;
41 
42         }
43 
44         public boolean add(Integer value){
45             Node current = header.next;
46             Node previous = header;
47 
48             while(current!=null && current.element.intValue()< value.
                   intValue()){
49                 previous = current;
50                 current = current.next;
51             }
52             Node n = new Node();
53             n.element =  value;
54             if (current==null){
55                 previous.next = n;
56                 size++;
57                 return true;
58             }
59             if(current.element.intValue()==value.intValue())
60                 return  false;
61 
62             if(current.element.intValue()>value.intValue()){
63                 previous.next = n;
64                 n.next = current;
65                 size++;
66                 return  true;
67             }
68             return true;
69 
70         }
71 
72         public Integer get(int index){
73             Node current = header.next;
74             int i = 0;
75             while(current!=null && i< index){
76                 current = current .next;
77                 i++;
78             }
79             if(current!=null){
80                 return current.element;
81             }
82             return null;
83         }
84 
85 
86         public int getSize(){
87             return size;
88         }
89 
90 
91         /**
92          * RepOk checks whether the list satisfies its representation
93          * invariant.
```

```
94        * @return True iff the current list satisfies its representation
95        * invariant.
96        */
97       public boolean repOK() {
98           if (!repOkCommon())
99               return false;
100          return repOkSorted();
101      }
102
103      public boolean repOkCommon() {
104          if (header == null)
105              return false;
106
107          if (header.element != null)
108              return false;
109          Set<Node> visited = new java.util.HashSet<Node>();
110          visited.add(header);
111          Node current = header;
112
113          while (true) {
114              Node next = current.next;
115              if (next == null)
116                  break;
117
118              if (next.element == null)
119                  return false;
120
121              if (!visited.add(next))
122                  return false;
123
124              current = next;
125          }
126          if (visited.size() - 1 != size)
127              return false;
128
129          return true;
130      }
131
132      public boolean repOkSorted() {
133          if (!repOkCommon())
134              return false;
135          if (size>1) {
136              for (Node current = header.next; current.next != null;
                     current = current.next) {
137                  if (current.element.compareTo(current.next.element) >=
                         0)
138                      return false;
139              }
140          }
141          return true;
142      }
143
144
145      /**
146       * Checks whether or not the current list has not elements.
147       * @return true iff the current list is empty, false otherwise.
148       */
149
150       public boolean isEmpty(){
151          return header.next== null;
152      }
153
154      public String toString() {
155          String res = "{";
156          if (header != null) {
157              Node cur = header.next;
158              while (cur != null && cur != header) {
159                  res += cur.toString();
```

```
160                     cur = cur.next;
161                 }
162             }
163             return res + "}";
164     }
165
166     }//End Class
```

```
1   package koratPlus.ListAsSet;
2
3   import java.io.Serializable;
4
5
6   /**
7   * StrictlySortedSinglyLinkedList's nodes
8   * This class needs to implement the Serializable interface
9   * since it is required for the korat "serialize" option.
10  * @author
11  */
12
13  public  class Node implements Serializable{
14
15      public  static final long serialVersionUID = 1;
16      public Integer element;
17
18      public Node next;
19
20      public String toString() {
21              return "[" + (element != null ? element.toString() : "null")
                        + "]";
22      }
23  }
```

```
1   package koratPlus.ListAsSet;
2
3   import java.util.Set;
4   import java.io.Serializable;
5
6
7   /**
8    * Class  SinglyLinkedList defines Singly linked List
9    * This class needs to implement the Serializable interface
10   * since it is required for the korat "serialize" option.
11   * @author
12   */
13  public class SinglyLinkedList implements Serializable{
14
15
16      public  static final long serialVersionUID = 1;
17
18      public Entry header;
19      public int size = 0;
20
21
22      public  SinglyLinkedList(){
23          header = new Entry();
24          size = 0;
25      }
26
27      public Integer get(int index){
28          Entry current = header.next;
29          int i = 0;
30          while(current!=null && i< index){
31              current = current .next;
32              i++;
33          }
34          if(current!=null){
```

```
35              return current.element;
36          }
37          return null;
38      }
39
40      /**
41       * RepOk checks whether the singlyLinkedList satisfies its
42       * representation invariant.
43       * @return True iff the current list satisfies its representation
44       * invariant.
45       */
46      public boolean repOK() {
47          if (header == null)
48              return false;
49          if (header.element != null)
50              return false;
51          Set<Entry> visited = new java.util.HashSet<Entry>();
52          visited.add(header);
53          Entry current = header;
54          while (true) {
55              Entry next = current.next;
56              if (next == null)
57                  break;
58              if (next.element == null)
59                  return false;
60              if (!visited.add(next))
61                  return false;
62              current = next;
63          }
64          if (visited.size() - 1 != size)
65              return false;
66          return true;
67      }
68
69
70      public int getSize(){
71          return size;
72      }
73
74      /**
75       * Checks whether or not the current list has not elements.
76       * @return true iff the current list is empty, false otherwise.
77       */
78      public boolean isEmpty(){
79          return header.next== null;
80      }
81
82
83      /**
84       * Checks whether or not the current list has not repeated
85       * elements.
86       * @return true iff all the elements in the list are
87       * different each other.
88       */
89      public boolean noReps(){
90          if(!isEmpty()){
91              for (Entry current = header.next; current.next != null;
                     current = current.next) {
92                  if (current.element.intValue()== current.next.element.
                         intValue())
93                      return false;
94              }
95          }
96          return true;
97      }
98
99      /**
100      * Checks whether or not the current list is sorted.
```

```
101         * @return true iff the list is sorted.
102         */
103        public boolean sorted(){
104            if(!isEmpty()){
105                for (Entry current = header.next; current.next!= null;
                       current = current.next) {
106                    if (current.element.intValue()> current.next.element.
                           intValue())
107                        return false;
108                }
109            }
110            return true;
111        }
112
113
114        public String toString() {
115            String res = "(";
116            if (header != null) {
117                Entry cur = header.next;
118                while (cur != null && cur != header) {
119                    res += cur.toString();
120                    cur = cur.next;
121                }
122            }
123            return res + ")";
124        }
125
126    }//End Class
```

```
1   package koratPlus.ListAsSet;
2   /**
3   * SinglyLinkedList's nodes
4   * This class needs to implement the Serializable interface
5   * since it is required for the korat "serialize" option.
6   * @author
7   */
8
9   import java.io.Serializable;
10
11  public  class Entry implements Serializable{
12
13      public  static final long serialVersionUID = 1;
14
15      public Integer element;
16
17      public Entry next;
18
19      public String toString() {
20              return "[" + (element != null ? element.toString() : "null")
                        + "]";
21      }
22  }
```

## Binomial Heaps (Merge)

```
1   package koratPlus.mergebinheap;
2
3   import java.math.BigInteger;
4
5   import korat.finitization.IClassDomain;
6   import korat.finitization.IFinitization;
7   import korat.finitization.IObjSet;
8   import korat.finitization.impl.FinitizationFactory;
9   import java.io.*;
10
11  /**
12   * Class MergeBinHeap defines the set of object needed to test the
```

```
13   * procedure 'merge' on binomial heaps, for which is necessary to
14   * provide pairs of binomial heaps.
15   * This class needs to implement the Serializable interface since it
16   * is required for the korat "serialize" option.
17   * @author Nazareno M. Aguirre, Valeria Bengolea.
18   */
19
20  public class MergeBinHeap implements Serializable{
21
22      public  static final long serialVersionUID = 1;
23
24      public BinomialHeap first;
25      public BinomialHeapS second;
26
27
28      /**
29       * checks that the two binomial heaps satisfy their representation
30       * invariants.
31       * @return true iff 'first' and 'second' satisfy the corresponding
32       * representation invariants.
33       */
34      public boolean repOK(){
35          if (!first.repOK())
36              return false;
37
38          return second.repOK();
39      }
40
41
42      public BinomialHeap merge(){
43          BinomialHeap b = second.convert();
44          return first.union(b);
45      }
46
47      /**
48       * binaryFirstSize translates the size of the first binomialHeap
49       * to a string as an unsigned integer in base 2.
50       * @return returns a string representation of the first's size
51       * as an unsigned integer in base 2.
52       */
53      public String binaryFirstSize(){
54          return Integer.toBinaryString(first.getSize());
55      }
56
57      /**
58       * binarySecondSize translates the size of the Second binomialHeap
59       * to a string as an unsigned integer in base 2.
60       * @return returns a string representation of the Second's size as
61       * an unsigned integer in base 2.
62       */
63      public String binarySecondSize(){
64          return Integer.toBinaryString(second.getSize());
65      }
66
67      /**
68       * Checks whether o not the two binomialHeaps have not degree in
69       * common, using the binary representation of their sizes
70       * @return returns true iff  first and second binomailHeaps do
71       * not have degree in common.
72       */
73      public boolean  NoDegreeInCommon(){
74          String first = binaryFirstSize();
75          String second = binarySecondSize();
76          int f = first.length()-1;
77          int s = second.length()-1;
78          int i = 0;
79          while(f - i >= 0 && s - i >= 0){
```

```
80              if((first.charAt(f - i) == second.charAt(s - i)) && (first.
                    charAt(f - i) == '1'))
81                  return false;
82              i++;
83          }
84          return true;
85      }
86
87
88      /**getClass translates a given array of boolean into a Integer
89       * @param classes The array to be translated
90       * @return The integer that represents the given array
91       * @see eqClass()
92       */
93      public BigInteger getClass(boolean [] classes){
94          BigInteger res = new BigInteger("0");
95          BigInteger dos =new BigInteger("2");
96          for (int i=0; i<classes.length; i++) {
97              if (classes[i]){
98                  BigInteger t = dos.pow(i);
99                  res = res.add(t);
100             }
101         }
102         return res;
103
104     }
105
106
107     /** eqClass returns the equivalence class in which the current
108      * instance of MergeBinHeap belongs to. In this case,
109      * the equivalence classes are given by all the
110      * possible combinations of the following predicates:
111      * isEmpty() over the first binomialHeap,
112      * isEmpty() over the second binomialHeap,
113      * size of the first binomialHeap is equal to the size of the
114      * second binomialHeap,  size of the first binomialHeap greater
115      * than the size of the second binomialHeap,
116      * first and second binomialHeaps have same degree
117      * first's degree greater than second's degree
118      * @return The equivalence class in which this instance belongs
119      */
120     public BigInteger eqClass() {
121         boolean[] classes = new boolean[7];
122         classes[0] = first.isEmpty();
123         classes[1] = second.isEmpty();
124         classes[2] = first.getSize() == second.getSize();
125         classes[3] = first.getSize() > second.getSize();
126
127         if((classes[0]) && !(classes[1])){
128             classes[4] = false; //if first is empty and second is not
129                     //empty, then they do not have the same degree
130             classes[5] = false; //if first is empty and second is not
131             //empty, then first's degree not greater than second's
132             //degree
133         }else if(!(classes[0]) && (classes[1])){
134             classes[4] = false; //if second is empty and first is not
135              //empty, then they do not have the same degree
136             classes[5] = true; //if second is empty and first is not
137              //empty, first's degree greater than second's degree
138         }else{
139             classes[4] = binaryFirstSize().length() == binarySecondSize
                    ().length();
140             classes[5] = binaryFirstSize().length() > binarySecondSize()
                    .length();
141         }
142         classes[6] = NoDegreeInCommon();
143         return getClass(classes);
144     }
```

```
145
146
147         /**
148          * finMergeBinHeap provides a bound on the number of objects
149          * to be used to generate  instances of mergeBinHeap.
150          * @param sizeF size of the first binomialHeap.
151          * @param sizeS size of the second binomialHeap.
152          * @return the object Korat needs for setting up the bounds
153          * during the search.
154          */
155         public static IFinitization finMergeBinHeap(int sizeF, int sizeS) {
156
157             IFinitization f = FinitizationFactory.create(MergeBinHeap.class)
                    ;
158
159             IObjSet entriesFirst = f.createObjSet(BinomialHeap.class, false)
                    ;
160             IObjSet entriesSecond = f.createObjSet(BinomialHeapS.class,
                    false);
161
162             entriesFirst.addClassDomain(f.createClassDomain(BinomialHeap.
                    class, 1));
163             entriesSecond.addClassDomain(f.createClassDomain(BinomialHeapS.
                    class, 1));
164
165             IClassDomain heapsDomainF = f.createClassDomain(BinomialHeapNode
                    .class,sizeF);
166             IObjSet heapsF = f.createObjSet(BinomialHeapNode.class);
167             heapsF.setNullAllowed(true);
168             heapsF.addClassDomain(heapsDomainF);
169
170
171             IClassDomain heapsDomainS = f.createClassDomain(
                    BinomialHeapNodeS.class,sizeS);
172             IObjSet heapsS = f.createObjSet(BinomialHeapNodeS.class);
173             heapsS.setNullAllowed(true);
174             heapsS.addClassDomain(heapsDomainS);
175
176             f.set("first",entriesFirst);
177             f.set("second",entriesSecond);
178
179             f.set("BinomialHeap.size", f.createIntSet(0, sizeF));
180             f.set("BinomialHeapS.size", f.createIntSet(0, sizeS));
181
182             f.set("BinomialHeap.Nodes", heapsF);
183             f.set("BinomialHeapS.Nodes", heapsS);
184
185             f.set(BinomialHeapNode.class, "parent", heapsF);
186             f.set(BinomialHeapNode.class, "sibling", heapsF);
187             f.set(BinomialHeapNode.class, "child", heapsF);
188             f.set(BinomialHeapNode.class, "key", f.createIntSet(1, sizeF));
189             f.set(BinomialHeapNode.class, "degree", f.createIntSet(0, sizeF)
                    );
190
191             f.set(BinomialHeapNodeS.class, "parent", heapsS);
192             f.set(BinomialHeapNodeS.class, "sibling", heapsS);
193             f.set(BinomialHeapNodeS.class, "child", heapsS);
194             f.set(BinomialHeapNodeS.class, "key", f.createIntSet(1, sizeS));
195             f.set(BinomialHeapNodeS.class, "degree", f.createIntSet(0, sizeS
                    ));
196
197             return f;
198         }
199     }
```

```
1   package koratPlus.mergebinheap;
2
3   import java.io.*;
```

```
4  /**
5   * This Class is a definition of Binomial Heaps
6   * This class needs to implement the Serializable interface since
7   * it is required for the korat "serialize" option.
8   * @author
9   *
10  */
11 public class BinomialHeap  implements Serializable{
12
13     public  static final long serialVersionUID = 1;
14
15     public BinomialHeapNode Nodes;
16
17     public int size;
18
19
20     /**
21      * Creates a new binomial heap that contains all the elements of
22      * two binomial heaps.  One of the original binomial heaps is
23      * the object on which this method is called; the other is
24      * specified by the parameter.  The two original binomial heaps
25      * should no longer be used after this operation.
26      *
27      * <p>
28      *
29      * Implements the Binomial-Heap-Union procedure on page 463,
30      * with h1 being this object.
31      *
32      * @param h2 The binomial heap to be merged with this one.
33      * @return The new binomial heap that contains all the elements
34      * of this binomial heap and <code>h2</code>.
35      */
36     public BinomialHeap union(BinomialHeap h2){
37         BinomialHeap h = new BinomialHeap();
38         h.size = this.size + h2.size;
39         h.Nodes = binomialHeapMerge(this, h2);
40         this.Nodes = null;  // no longer using the...
41         h2.Nodes = null; // ...two input lists
42         if (h.Nodes == null)
43             return h;
44
45         BinomialHeapNode prevX = null;
46         BinomialHeapNode x = h.Nodes;
47         BinomialHeapNode nextX = x.sibling;
48
49         while (nextX != null) {
50             if (x.degree != nextX.degree ||
51                     (nextX.sibling != null && nextX.sibling.degree == x.
                             degree)) {
52                 // Cases 1 and 2.
53                 prevX = x;
54                 x = nextX;
55             }
56             else {
57                 if (x.key< nextX.key) {
58                     // Case 3.
59                     x.sibling = nextX.sibling;
60                     binomialLink(nextX, x);
61                 }
62                 else {
63                     // Case 4.
64                     if (prevX == null)
65                         h.Nodes = nextX;
66                     else
67                         prevX.sibling = nextX;
68                     binomialLink(x, nextX);
69                     x = nextX;
70                 }
```

```
71              }
72
73              nextX = x.sibling;
74          }
75
76          return h;
77      }
78
79      /**
80       * Links one binomial tree to another.
81       *
82       * @param y The root of one binomial tree.
83       * @param z The root of another binomial tree; this root becomes
84       * the parent of <code>y</code>.
85       */
86      private void binomialLink(BinomialHeapNode y, BinomialHeapNode z){
87          y.parent = z;
88          y.sibling = z.child;
89          z.child = y;
90          z.degree++;
91      }
92
93      /**
94       * Merges the root lists of two binomial heaps together into a
95       * single root list.  The degrees in the merged root list appear
96       * in monotonically increasing order.
97       *
98       * @param h1 One binomial heap.
99       * @param h2 The other binomial heap.
100      * @return The head of the merged list.
101      */
102     private static BinomialHeapNode binomialHeapMerge(BinomialHeap h1,
            BinomialHeap h2){
103         // If either root list is empty, just return the other.
104         if (h1.Nodes == null)
105             return h2.Nodes;
106         else if (h2.Nodes == null)
107             return h1.Nodes;
108         else {
109             // Neither root list is empty.  Scan through both, always
110             // using the node whose degree is smallest of those not
111             // yet taken.
112             BinomialHeapNode head; // head of merged list
113             BinomialHeapNode tail; // last node added to merged list
114             BinomialHeapNode h1Next = h1.Nodes,
115             h2Next = h2.Nodes;// next nodes to be examined in h1 and h2
116
117             if (h1.Nodes.degree <= h2.Nodes.degree) {
118                 head = h1.Nodes;
119                 h1Next = h1Next.sibling;
120             }
121             else {
122                 head = h2.Nodes;
123                 h2Next = h2Next.sibling;
124             }
125             tail = head;
126
127             // Go through both root lists until one of them is
128             // exhausted.
129             while (h1Next != null && h2Next != null) {
130                 if (h1Next.degree <= h2Next.degree) {
131                     tail.sibling = h1Next;
132                     h1Next = h1Next.sibling;
133                 }
134                 else {
135                     tail.sibling = h2Next;
136                     h2Next = h2Next.sibling;
137                 }
```

```
138
139                 tail = tail.sibling;
140             }
141
142             // The above loop ended because exactly one of the root
143             // lists was exhuasted.  Splice the remainder of whichever
144             // root list was not exhausted onto the list we're
145             // constructing.
146             if (h1Next != null)
147                 tail.sibling = h1Next;
148             else
149                 tail.sibling = h2Next;
150
151             return head;     // all done!
152         }
153     }
154
155     public int getSize() {
156         return size;
157     }
158
159     public String toString() {
160         if (Nodes == null)
161             return size + "()";
162         else
163             return size + " " + Nodes.toString();
164     }
165
166
167     /**
168      * Checks whether the current binomial heap is empty.
169      * @return true iff the current binomial heap is empty,
170      * false otherwise.
171      */
172     public boolean isEmpty(){
173         return size == 0;
174     }
175
176     public int degree(){
177         return Nodes.degree;
178
179     }
180
181
182     /**
183      * checks that the current binomial heap satisfies its
184      * representation invariants, which means  checking that
185      * list of trees has no cycles, the total size is consistent,
186      * the degrees of all trees are binomial and the keys are
187      * heapified.
188      * @return true iff this binomialHeap satisfies the corresponding
189      * representation invariants.
190      */
191     public boolean repOK() {
192         if (size == 0)
193             return (Nodes == null);
194         if (Nodes == null)
195             return false;
196         java.util.Set<NodeWrapper> visited = new java.util.HashSet<
                NodeWrapper>();
197         for (BinomialHeapNode current = Nodes; current != null; current
                = current.sibling) {
198             /**checks that the list has no cycle*/
199             if (!visited.add(new NodeWrapper(current)))
200                 return false;
201             if (!current.isTree(visited, null))
202                 return false;
203         }
```

```
204         /**checks that the total size is consistent*/
205         if (visited.size() != size)
206             return false;
207         /**checks that the degrees of all trees are binomial*/
208         if (!checkDegrees())
209             return false;
210         /**checks that keys are heapified*/
211         if (!checkHeapified())
212             return false;
213         return true;
214     }
215
216
217     boolean checkDegrees() {
218         int degree_ = size;
219         int rightDegree = 0;
220         for (BinomialHeapNode current = Nodes; current != null; current
                = current.sibling) {
221             if (degree_ == 0)
222                 return false;
223             while ((degree_ & 1) == 0) {
224                 rightDegree++;
225                 degree_ /= 2;
226             }
227             if (current.degree != rightDegree)
228                 return false;
229             if (!current.checkDegree(rightDegree))
230                 return false;
231             rightDegree++;
232             degree_ /= 2;
233         }
234         return (degree_ == 0);
235     }
236
237     boolean checkHeapified() {
238         for (BinomialHeapNode current = Nodes; current != null; current
                = current.sibling) {
239             if (!current.isHeapified())
240                 return false;
241         }
242         return true;
243     }
244
245 }
```

```
1  package koratPlus.mergebinheap;
2
3  import java.util.HashSet;
4  import java.io.*;
5
6  /**
7   * This class defines BinomialHeap's nodes
8   * This class needs to implement the Serializable interface
9   * since it is required for the korat "serialize" option.
10  * @author
11  */
12 public class BinomialHeapNode implements Serializable{
13
14     public  static final long serialVersionUID = 1;
15
16     /**
17      * element in current node
18      */
19     public int key;
20
21
22     /**
23      * 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
45                     : sibling.getSize()));
46         }
47
48         public String toString() {
49             BinomialHeapNode temp = this;
50             String ret = "";
51             while (temp != null) {
52                 ret += "(";
53                 if (temp.parent == null)
54                     ret += "Parent: null";
55                 else
56                     ret += "Parent: " + temp.parent.key;
57                 ret += "  Degree: " + temp.degree + "  Key: " + temp.key + "
                       ) ";
58                 if (temp.child != null)
59                     ret += temp.child.toString();
60                 temp = temp.sibling;
61             }
62             if (parent == null)
63                 ret += " ";
64             return ret;
65         }
66
67
68         private boolean repCheckWithRepetitions(int key_, int degree_,
69                 Object parent_, HashSet<BinomialHeapNode> nodesSet) {
70
71             BinomialHeapNode temp = this;
72
73             int rightDegree = 0;
74             if (parent_ == null) {
75                 while ((degree_ & 1) == 0) {
76                     rightDegree += 1;
77                     degree_ /= 2;
78                 }
79                 degree_ /= 2;
80             } else
81                 rightDegree = degree_;
82
83             while (temp != null) {
84                 if ((temp.degree != rightDegree) || (temp.parent != parent_)
85                         || (temp.key < key_) || (nodesSet.contains(temp)))
86                     return false;
87                 else {
88                     nodesSet.add(temp);
89                     if (temp.child == null) {
```

```
90                      temp = temp.sibling;
91
92                      if (parent_ == null) {
93                          if (degree_ == 0)
94                              return (temp == null);
95                          while ((degree_ & 1) == 0) {
96                              rightDegree += 1;
97                              degree_ /= 2;
98                          }
99                          degree_ /= 2;
100                         rightDegree++;
101                     } else
102                         rightDegree--;
103                 } else {
104                     boolean b = temp.child.repCheckWithRepetitions(
105                             temp.key, temp.degree - 1, temp, nodesSet);
106                     if (!b)
107                         return false;
108                     else {
109                         temp = temp.sibling;
110
111                         if (parent_ == null) {
112                             if (degree_ == 0)
113                                 return (temp == null);
114                             while ((degree_ & 1) == 0) {
115                                 rightDegree += 1;
116                                 degree_ /= 2;
117                             }
118                             degree_ /= 2;
119                             rightDegree++;
120                         } else
121                             rightDegree--;
122                     }
123                 }
124             }
125         }
126
127         return true;
128     }
129
130     private boolean repCheckWithoutRepetitions(int key_,
131             HashSet<Integer> keysSet, int degree_, // equal keys not
                    allowed
132             Object parent_, HashSet<BinomialHeapNode> nodesSet) {
133         BinomialHeapNode temp = this;
134
135         int rightDegree = 0;
136         if (parent_ == null) {
137             while ((degree_ & 1) == 0) {
138                 rightDegree += 1;
139                 degree_ /= 2;
140             }
141             degree_ /= 2;
142         } else
143             rightDegree = degree_;
144
145         while (temp != null) {
146             if ((temp.degree != rightDegree) || (temp.parent != parent_)
147                     || (temp.key <= key_) || (nodesSet.contains(temp))
148                     || (keysSet.contains(new Integer(temp.key)))) {
149                 return false;
150             } else {
151                 nodesSet.add(temp);
152                 keysSet.add(new Integer(temp.key));
153                 if (temp.child == null) {
154                     temp = temp.sibling;
155
156                     if (parent_ == null) {
```

```
157                         if (degree_ == 0)
158                             return (temp == null);
159                         while ((degree_ & 1) == 0) {
160                             rightDegree += 1;
161                             degree_ /= 2;
162                         }
163                         degree_ /= 2;
164                         rightDegree++;
165                     } else
166                         rightDegree--;
167                 } else {
168                     boolean b = temp.child.repCheckWithoutRepetitions(
169                             temp.key, keysSet, temp.degree - 1, temp,
170                             nodesSet);
171                     if (!b)
172                         return false;
173                     else {
174                         temp = temp.sibling;
175 
176                         if (parent_ == null) {
177                             if (degree_ == 0)
178                                 return (temp == null);
179                             while ((degree_ & 1) == 0) {
180                                 rightDegree += 1;
181                                 degree_ /= 2;
182                             }
183                             degree_ /= 2;
184                             rightDegree++;
185                         } else
186                             rightDegree--;
187                     }
188                 }
189             }
190         }
191 
192         return true;
193     }
194 
195     boolean checkDegree(int degree) {
196         for (BinomialHeapNode current = this.child; current != null;
                 current = current.sibling) {
197             degree--;
198             if (current.degree != degree)
199                 return false;
200             if (!current.checkDegree(degree))
201                 return false;
202         }
203         return (degree == 0);
204     }
205 
206     boolean isHeapified() {
207         for (BinomialHeapNode current = this.child; current != null;
                 current = current.sibling) {
208             if (!(key <= current.key))
209                 return false;
210             if (!current.isHeapified())
211                 return false;
212         }
213         return true;
214     }
215 
216     boolean isTree(java.util.Set<NodeWrapper> visited,
217             BinomialHeapNode parent) {
218         if (this.parent != parent)
219             return false;
220         for (BinomialHeapNode current = this.child; current != null;
                 current = current.sibling) {
221             if (!visited.add(new NodeWrapper(current)))
```

```
222                     return false;
223                 if (!current.isTree(visited, this))
224                     return false;
225             }
226             return true;
227         }
228 
229         public boolean repOk(int size) {
230             /**replace 'repCheckWithoutRepetitions' with
231             'repCheckWithRepetitions' if you don't want to allow equal keys
                    */
232             return repCheckWithRepetitions(0, size, null,
233                     new HashSet<BinomialHeapNode>());
234         }
235 }
```

```
1  package koratPlus.mergebinheap;
2  
3  import java.io.*;
4  /**
5   * This Class is a definition of Binomial Heaps.
6   * This class needs to implement the Serializable interface since
7   * it is required for the korat "serialize" option.
8   * @author
9   *
10  */
11 public class BinomialHeapS  implements Serializable{
12 
13     public  static final long serialVersionUID = 1;
14 
15     public BinomialHeapNodeS Nodes;
16 
17     public int size;
18 
19     public BinomialHeap convert(){
20         BinomialHeap bh = new BinomialHeap();
21         bh.size = this.size;
22         java.util.ArrayList <Wrapper> l = new java.util.ArrayList<
               Wrapper>(this.size);
23         try {
24             bh.Nodes = this.Nodes.convertion(l);
25         }
26         catch(Exception e){
27         }
28 
29         return bh;
30     }
31 
32     public int getSize() {
33         return size;
34     }
35 
36     public String toString() {
37         if (Nodes == null)
38             return size + "\n()\n";
39         else
40             return size + "\n" + Nodes.toString();
41     }
42 
43     /**
44      * Checks whether the current binomial heap is empty
45      * @return true iff the current binomial heap is empty, false
              otherwise
46      */
47     public boolean isEmpty(){
48         return size == 0;
49     }
50 
```

```
51      public int degree(){
52          return Nodes.degree;
53
54      }
55
56      /**
57       * checks that the current binomial heap satisfies its
58       * representation invariant, which means checking that
59       * list of trees has no cycles, the total size is consistent,
60       * the degrees of all trees are binomial and the keys are
61       * heapified.
62       * @return true iff this binomialHeapS satisfies the
63       * corresponding representation invariant.
64       */
65
66      public boolean repOK() {
67          if (size == 0)
68              return (Nodes == null);
69          if (Nodes == null)
70              return false;
71          /**checks that list of trees has no cycles*/
72          java.util.Set<NodeWrapperS> visited = new java.util.HashSet<
                NodeWrapperS>();
73          for (BinomialHeapNodeS current = Nodes; current != null; current
                 = current.sibling) {
74              /**checks that the list has no cycle*/
75              if (!visited.add(new NodeWrapperS(current)))
76                  return false;
77              if (!current.isTree(visited, null))
78                  return false;
79          }
80          /**checks that the total size is consistent*/
81          if (visited.size() != size)
82              return false;
83          /**checks that the degrees of all trees are binomial*/
84          if (!checkDegrees())
85              return false;
86          /**checks that keys are heapified*/
87          if (!checkHeapified())
88              return false;
89          return true;
90      }
91
92      boolean checkDegrees() {
93          int degree_ = size;
94          int rightDegree = 0;
95          for (BinomialHeapNodeS  current = Nodes; current != null;
                current = current.sibling) {
96              if (degree_ == 0)
97                  return false;
98              while ((degree_ & 1) == 0) {
99                  rightDegree++;
100                 degree_ /= 2;
101             }
102             if (current.degree != rightDegree)
103                 return false;
104             if (!current.checkDegree(rightDegree))
105                 return false;
106             rightDegree++;
107             degree_ /= 2;
108         }
109         return (degree_ == 0);
110     }
111
112     boolean checkHeapified() {
113         for (BinomialHeapNodeS current = Nodes; current != null; current
                 = current.sibling) {
114             if (!current.isHeapified())
```

```
115                     return false;
116                 }
117                 return true;
118             }
119 
120 
121 }
```

```
  1 package koratPlus.mergebinheap;
  2 
  3 import java.util.HashSet;
  4 import java.io.*;
  5 
  6 /**
  7  * This class defines BinomialHeapS's nodes.
  8  * This class needs to implement the Serializable interface since
  9  *  it is required for the korat "serialize" option.
 10  * @author
 11  */
 12 public class BinomialHeapNodeS implements Serializable{
 13 
 14     public  static final long serialVersionUID = 1;
 15 
 16     /**
 17      * element in current node
 18      */
 19     public int key;
 20 
 21     /**
 22      *depth of the binomial tree having the current node as its root
 23      */
 24     public int degree;
 25 
 26     /**
 27      *pointer to the parent of the current node
 28      */
 29     public  BinomialHeapNodeS parent;
 30 
 31     /**
 32      *pointer to the next binomial tree in the list
 33      */
 34     public BinomialHeapNodeS sibling;
 35 
 36     /**
 37      *pointer to the first child of the current node
 38      */
 39     public BinomialHeapNodeS child;
 40 
 41     private BinomialHeapNode search(java.util.ArrayList<Wrapper> list,
         BinomialHeapNodeS s){
 42 
 43         for(int i = 0; i<list.size(); i++){
 44             Wrapper w = list.get(i);
 45             if(w.bhns==s){  //quiero comparar referencias
 46                 return w.bhn;
 47             }
 48         }
 49         return null;
 50     }
 51 
 52 
 53 
 54     public BinomialHeapNode convertion(java.util.ArrayList<Wrapper> list
         ){
 55         BinomialHeapNode n = new BinomialHeapNode();
 56         n.key = this.key;
 57         n.degree = this.degree;
 58         Wrapper w = new Wrapper(this,n);
```

```
59          list.add(w);
60          if(this.parent !=null){
61
62              n.parent = search(list, this);
63          }
64          if(this.sibling!=null)
65              n.sibling = this.sibling.convertion(list);
66          if(this.child!=null)
67              n.child = this.child.convertion(list);
68          return n;
69
70      }
71
72      public int getSize() {
73          return (1 + ((child == null) ? 0 : child.getSize()) + ((sibling
                == null) ? 0
74                  : sibling.getSize()));
75      }
76
77      public String toString() {
78          BinomialHeapNodeS temp = this;
79          String ret = "";
80          while (temp != null) {
81              ret += "(";
82              if (temp.parent == null)
83                  ret += "Parent: null";
84              else
85                  ret += "Parent: " + temp.parent.key;
86              ret += "  Degree: " + temp.degree + "  Key: " + temp.key + "
                    ) ";
87              if (temp.child != null)
88                  ret += temp.child.toString();
89              temp = temp.sibling;
90          }
91          if (parent == null)
92              ret += " ";
93          return ret;
94      }
95
96
97      private boolean repCheckWithRepetitions(int key_, int degree_,
98              Object parent_, HashSet<BinomialHeapNodeS> nodesSet) {
99
100         BinomialHeapNodeS temp = this;
101
102         int rightDegree = 0;
103         if (parent_ == null) {
104             while ((degree_ & 1) == 0) {
105                 rightDegree += 1;
106                 degree_ /= 2;
107             }
108             degree_ /= 2;
109         } else
110             rightDegree = degree_;
111
112         while (temp != null) {
113             if ((temp.degree != rightDegree) || (temp.parent != parent_)
114                     || (temp.key < key_) || (nodesSet.contains(temp)))
115                 return false;
116             else {
117                 nodesSet.add(temp);
118                 if (temp.child == null) {
119                     temp = temp.sibling;
120
121                     if (parent_ == null) {
122                         if (degree_ == 0)
123                             return (temp == null);
124                         while ((degree_ & 1) == 0) {
```

```
125                             rightDegree += 1;
126                             degree_ /= 2;
127                         }
128                         degree_ /= 2;
129                         rightDegree++;
130                     } else
131                         rightDegree--;
132                 } else {
133                     boolean b = temp.child.repCheckWithRepetitions(
134                             temp.key, temp.degree - 1, temp, nodesSet);
135                     if (!b)
136                         return false;
137                     else {
138                         temp = temp.sibling;
139 
140                         if (parent_ == null) {
141                             if (degree_ == 0)
142                                 return (temp == null);
143                             while ((degree_ & 1) == 0) {
144                                 rightDegree += 1;
145                                 degree_ /= 2;
146                             }
147                             degree_ /= 2;
148                             rightDegree++;
149                         } else
150                             rightDegree--;
151                     }
152                 }
153             }
154         }
155 
156         return true;
157     }
158 
159     private boolean repCheckWithoutRepetitions(int key_,
160             HashSet<Integer> keysSet, int degree_, /**equal keys not
                    allowed*/
161             Object parent_, HashSet<BinomialHeapNodeS> nodesSet) {
162         BinomialHeapNodeS temp = this;
163 
164         int rightDegree = 0;
165         if (parent_ == null) {
166             while ((degree_ & 1) == 0) {
167                 rightDegree += 1;
168                 degree_ /= 2;
169             }
170             degree_ /= 2;
171         } else
172             rightDegree = degree_;
173 
174         while (temp != null) {
175             if ((temp.degree != rightDegree) || (temp.parent != parent_)
176                     || (temp.key <= key_) || (nodesSet.contains(temp))
177                     || (keysSet.contains(new Integer(temp.key)))) {
178                 return false;
179             } else {
180                 nodesSet.add(temp);
181                 keysSet.add(new Integer(temp.key));
182                 if (temp.child == null) {
183                     temp = temp.sibling;
184 
185                     if (parent_ == null) {
186                         if (degree_ == 0)
187                             return (temp == null);
188                         while ((degree_ & 1) == 0) {
189                             rightDegree += 1;
190                             degree_ /= 2;
191                         }
```

```
192                         degree_ /= 2;
193                         rightDegree++;
194                     } else
195                         rightDegree--;
196                 } else {
197                     boolean b = temp.child.repCheckWithoutRepetitions(
198                             temp.key, keysSet, temp.degree - 1, temp,
199                             nodesSet);
200                     if (!b)
201                         return false;
202                     else {
203                         temp = temp.sibling;
204 
205                         if (parent_ == null) {
206                             if (degree_ == 0)
207                                 return (temp == null);
208                             while ((degree_ & 1) == 0) {
209                                 rightDegree += 1;
210                                 degree_ /= 2;
211                             }
212                             degree_ /= 2;
213                             rightDegree++;
214                         } else
215                             rightDegree--;
216                     }
217                 }
218             }
219         }
220 
221         return true;
222     }
223 
224     boolean checkDegree(int degree) {
225         for (BinomialHeapNodeS current = this.child; current != null;
                 current = current.sibling) {
226             degree--;
227             if (current.degree != degree)
228                 return false;
229             if (!current.checkDegree(degree))
230                 return false;
231         }
232         return (degree == 0);
233     }
234 
235     boolean isHeapified() {
236         for (BinomialHeapNodeS current = this.child; current != null;
                 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;
                 current = current.sibling) {
250             if (!visited.add(new NodeWrapperS(current)))
251                 return false;
252             if (!current.isTree(visited, this))
253                 return false;
254         }
255         return true;
256     }
```

```
257
258     public boolean repOk(int size) {
259         /**replace 'repCheckWithoutRepetitions' with
260             'repCheckWithRepetitions' if you don't want to allow equal
                keys*/
261         return repCheckWithRepetitions(0, size, null,
262                 new HashSet<BinomialHeapNodeS>());
263     }
264
265 }
```

## Directed Graphs

```
 1 package koratPlus.AltGraph;
 2
 3 import korat.finitization.IArraySet;
 4 import korat.finitization.IFinitization;
 5 import korat.finitization.IIntSet;
 6 import korat.finitization.IObjSet;
 7 import korat.finitization.impl.FinitizationFactory;
 8
 9 import java.math.BigInteger;
10 import java.util.Set;
11 import java.util.Iterator;
12
13 /**
14  * Class AltGraph defines Directed Graphs using adjacent lists
15  * @author Nazareno M. Aguirre, Valeria Bengolea.
16  */
17 public class AltGraph {
18
19     Vertex[] vertices;
20
21     int size;
22
23
24     /**
25      * RepOk checks whether the directed graph satisfies its
26      * representation invariant.
27      * @return true iff the current graph satisfies its
28      * representation invariant
29      */
30     public boolean repOK() {
31         if (vertices==null)
32             return false;
33         if (vertices.length != size)
34             return false;
35         /**check if each adj. list satisfies the
36          * list's representation invariant.
37          */
38         for (int i=0; i<vertices.length; i++) {
39             if (vertices[i] == null)
40                 return false;
41             if (!vertices[i].repOK())
42                 return false;
43         }
44         /** check if the label of each vertex is equal to
45          *  the corresponding position in the array.
46          */
47         for (int i=0; i<vertices.length; i++) {
48             if (vertices[i].label!=i)
49                 return false;
50         }
51         return true;
52     }
53
54     /**finAltGraph provides a bound on the number of objects to be
```

```
55          * used to generate instances of AltGraph.
56          * @param numVertices the number of vertices that any instance
57          * of AltGraph may have.
58          * @return  the object Korat needs for setting up the bounds
59          * during the search.
60          */
61         public static IFinitization finAltGraph(int numVertices) {
62             IFinitization f = FinitizationFactory.create(AltGraph.class);
63 
64             IIntSet arrLen = f.createIntSet(numVertices);
65 
66             IObjSet entries = f.createObjSet(Entry.class, true);
67             //create enough entries for all adj. lists
68             entries.addClassDomain(f.createClassDomain(Entry.class,
69                 numVertices*numVertices));
70 
71             IObjSet vertices = f.createObjSet(Vertex.class, true);
72             vertices.addClassDomain(f.createClassDomain(Vertex.class,
73                 numVertices));
74 
75             IObjSet lists = f.createObjSet(SinglyLinkedList.class,
76                 numVertices, false);
77 
78             IArraySet verticesArray = f.createArraySet(Vertex[].class,
79                 arrLen, vertices, numVertices);
80 
81             f.set(Vertex.class, "label", f.createIntSet(0,numVertices-1));
82             f.set(Vertex.class, "adjacent", lists);
83 
84             f.set(SinglyLinkedList.class, "header", entries);
85             f.set(Entry.class, "element", vertices);
86             f.set(Entry.class, "next", entries);
87             f.set("vertices", verticesArray);
88             f.set("size", f.createIntSet(numVertices));
89 
90             return f;
91         }
92 
93 
94         /**
95          * Returns all the reachable nodes from a given node following a
96          * depth first strategy.
97          * @param n The starting node
98          * @param visited A set used for keeping track the visited nodes.
99          * @return The set containing all the reachable nodes from 'n'
100         */
101        public Set<Integer> dfs(Integer n, Set<Integer> visited) {
102            SinglyLinkedList s = vertices[n].adjacent;
103            for (Entry curr=s.header; curr!=null; curr=curr.next) {
104                if(visited.add(curr.element.label)){
105                    dfs(curr.element.label, visited);
106                }
107            }
108            return visited;
109        }
110 
111        /**
112         * Checks whether or not a node belongs to a given set.
113         * @param s Set to be tested.
114         * @param n The element whose presence in this set is to be tested.
115         * @return true iff 'n' is in the set 's', false otherwise
116         */
117        public boolean isIn(Set<Integer> s, Integer n){
118            Iterator it = s.iterator();
119            while(it.hasNext()){
120                if(n.intValue() == ((Integer)it.next()).intValue())
121                    return true;
122            }
```

```
119             return false;
120         }
121 
122         /**
123          * Checks whether or not a set contains all the nodes of the
124          * current graph.
125          * @param s The set to be tested.
126          * @return true is returned iff all nodes in the graph are in s,
127          * false otherwise.
128          */
129         public boolean all(Set<Integer> s){
130             int i= 0;
131             while(i < vertices.length){
132                 if (isIn(s,i) == false){
133                     return false;
134                 }
135                 i++;
136 
137             }
138             return true;
139         }
140 
141         /**
142          * Checks whether o not the current graph is strongly connected
143          * @return True  iff the current graph is strongly connected,
144          * false otherwise.
145          */
146         public boolean stronglyConnected(){
147             for(int i = 0; i< vertices.length; i++){
148                 Set<Integer> visited = new java.util.HashSet<Integer>();
149                 visited = dfs(i, visited);
150                 visited.add(i);
151                 if(!all(visited)){
152                     return false;
153                 }
154             }
155             return true;
156         }
157 
158         /**
159          * Checks whether o not the current graph has a least one loop.
160          * @return True iff a loop is found, false otherwise
161          */
162         public boolean cyclic(){
163             for(int i = 0; i< vertices.length; i++){
164                 Set<Integer> visited = new java.util.HashSet<Integer>();
165                 visited = dfs(i, visited);
166                 if(isIn(visited, i))
167                     return true;
168             }
169             return false;
170 
171         }
172 
173 
174         /**
175          * Checks whether o not the current graph is empty
176          * @return True iff the current graph is empty, false otherwise
177          */
178         public boolean isEmpty() {
179             for (int i=0; i<vertices.length; i++) {
180                 if (vertices[i].adjacent.header!=null) return false;
181             }
182             return true;
183         }
184 
185         /**
186          * Checks whether o not the current graph is dense
```

```
187         * @return True is returned iff the graph is dense, false otherwise
188         */
189        public boolean isDense() {
190            int numArcs = 0;
191            for (int i=0; i<vertices.length; i++) {
192                for (Entry current = vertices[i].adjacent.header; current !=
                        null; current = current.next) {
193                    numArcs++;
194                    if (numArcs>((size*(size-1))/2)) return true;
195                }
196            }
197            return false;
198        }
199
200        /**
201         * Checks whether o not the current graph is complete
202         * @return True iff the current graph is complete, false otherwise
203         */
204        public boolean complete() {
205            for (int i=0; i<vertices.length; i++) {
206                int numArcs = 0;
207                for (Entry current = vertices[i].adjacent.header; current !=
                        null; current = current.next) {
208                    numArcs++;
209                }
210                if (numArcs!=size)
211                    return false;
212            }
213            return true;
214        }
215
216        /**getClass translates a given array of boolean into a Integer
217         * @param classes The array to be translated
218         * @return The integer that represents the given array
219         * @see eqClass()
220         */
221        public BigInteger getClass(boolean [] classes){
222            BigInteger res = new BigInteger("0");
223            BigInteger dos =new BigInteger("2");
224            for (int i=0; i<classes.length; i++) {
225                if (classes[i]){
226                    BigInteger t = dos.pow(i);
227                    res = res.add(t);
228                }
229            }
230            return res;
231        }
232
233
234        /** eqClass given a graph it returns the equivalence class
235         * in which this graph belongs to. In this case, the
236         * equivalence classes are given by all the possible combinations
237         * of predicates isEmpty(), complete() and isDense().
238         * @return The equivalence class in which the current graph belongs
239         */
240        public BigInteger eqClass(){
241            boolean[] classes = new boolean[3];
242            classes[0] = isEmpty();
243            if (classes[0]) {
244                classes[1] = false; // if empty, then it's not dense
245                classes[2] = false; // if empty, then it's not complete
246            }
247            else {
248                classes[1] = isDense();
249                if (classes[1]) {
250                    classes[2] = complete();
251                }
252                else {
```

```
253                     classes[2] = false; // if not dense, not complete
254                 }
255             }
256             return this.getClass(classes);
257         }
258 
259 }//end Class
```

```
 1 package koratPlus.AltGraph;
 2 
 3 import java.util.Set;
 4 
 5 import java.math.BigInteger;
 6 import korat.finitization.IClassDomain;
 7 import korat.finitization.IFinitization;
 8 import korat.finitization.IIntSet;
 9 import korat.finitization.IObjSet;
10 import korat.finitization.impl.FinitizationFactory;
11 
12 /**
13  * Class vertex defines vertex of the AltGraph (Directed graph).
14  * @author Nazareno M. Aguirre
15  *
16  */
17 public class Vertex {
18 
19     public int label;
20 
21     public SinglyLinkedList adjacent;
22 
23 
24     /**
25      *check if adjacent  satisfies the list's
26      *representation invariant.
27      * @return True iff the current vertex satisfies its
28      * representation invariant.
29      */
30     public boolean repOK() {
31         if (adjacent==null) return false;
32             return (adjacent.repOK());
33     }
34 
35 
36 }
```

```
 1 package koratPlus.AltGraph;
 2 
 3 import java.util.Set;
 4 
 5 import java.math.BigInteger;
 6 import korat.finitization.IClassDomain;
 7 import korat.finitization.IFinitization;
 8 import korat.finitization.IIntSet;
 9 import korat.finitization.IObjSet;
10 import korat.finitization.impl.FinitizationFactory;
11 
12 /**
13  * Class  SinglyLinkedList defines Singly linked List
14  * @author
15  */
16 public class SinglyLinkedList {
17 
18     public Entry header;
19 
20 
21     /**
22      * RepOk checks whether the singlyLinkedList satisfies
```

```
23        * its representation invariant  (acyclicity).
24        * @return True iff the current list satisfies its
25        * resentation invariant.
26        */
27       public boolean repOK() {
28           return repOkCommon();
29       }
30
31       public boolean repOkCommon() {
32           Set<Entry> visited = new java.util.HashSet<Entry>();
33           Entry current = header;
34           while (current!=null) {
35                if (!visited.add(current))
36                    return false;
37                if (current.element == null)
38                    return false;
39                current = current.next;
40           }
41           return true;
42       }
43
44       public boolean repOkSorted(int maxVertex) {
45           if (!repOkCommon()) return false;
46           for (Entry current = header; current != null; current = current.
                 next) {
47                if (current.next!=null) {
48                    if (current.element.label>=current.next.element.label)
                          return false;
49                }
50           }
51           return true;
52       }
53
54
55       public String toString() {
56           String res = "(";
57           if (header != null) {
58                Entry cur = header.next;
59                while (cur != null && cur != header) {
60                    res += cur.toString();
61                    cur = cur.next;
62                }
63           }
64           return res + ")";
65       }
66
67   }
```

```
1   package koratPlus.AltGraph;
2
3   /**
4    * Entries of the SinglyLinkedList class.
5    * @author
6    */
7   public class Entry {
8
9       public Vertex element;
10
11      public Entry next;
12
13      public String toString() {
14          return "[" + (element != null ? element.toString() : "null") + "
                ]";
15      }
16  }
```

## Weighted Directed Graphs

```java
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 implementes 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;
            /**check if each adj. list is strictly sorted*/
            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) {
```

```
68          IFinitization f = FinitizationFactory.create(LabelledAltGraph.
                class);
69
70          IIntSet arrLen = f.createIntSet(numVertices);
71
72          IObjSet entries = f.createObjSet(Entry.class, true);
73          entries.addClassDomain(f.createClassDomain(Entry.class,
                numVertices*numVertices)); // create enough entries for all
                adj. lists
74
75          IObjSet vertices = f.createObjSet(Vertex.class, true);
76          vertices.addClassDomain(f.createClassDomain(Vertex.class,
                numVertices));
77
78          IObjSet lists = f.createObjSet(SortedSinglyLinkedList.class,
                numVertices, false);
79
80          IArraySet verticesArray = f.createArraySet(Vertex[].class,
                arrLen, vertices, numVertices);
81
82          f.set(Vertex.class, "label", f.createIntSet(0,numVertices-1));
83          f.set(Vertex.class, "adjacent", lists);
84
85          f.set(SortedSinglyLinkedList.class, "header", entries);
86          f.set(Entry.class, "element", vertices);
87          f.set(Entry.class, "transLabel", f.createIntSet(minNumLabels,
                maxNumLabels));
88          f.set(Entry.class, "next", entries);
89          f.set("vertices", verticesArray);
90          f.set("size", f.createIntSet(numVertices));
91
92          return f;
93      }
94
95
96      /**
97       * Returns all the reachable nodes from a given node following a
98       * depth first strategy.
99       * @param n The starting node.
100      * @param visited A set used for keeping track the visited nodes.
101      * @return The set containing all the reachable nodes from 'n'
102      */
103     public Set<Integer> dfs(Integer n, Set<Integer> visited) {
104         SortedSinglyLinkedList s = vertices[n].adjacent;
105         for (Entry curr=s.header; curr!=null; curr=curr.next) {
106             if(visited.add(curr.element.label)){
107                 dfs(curr.element.label, visited);
108             }
109         }
110         return visited;
111     }
112
113
114     /**
115      * Checks whether or not a node belongs to a given set.
116      * @param s Set to be tested.
117      * @param n The element whose presence in this set is to be tested.
118      * @return true iff 'n' is in the set 's', false otherwise.
119      */
120     public boolean isIn(Set<Integer> s, Integer n){
121         Iterator it = s.iterator();
122         while(it.hasNext()){
123             if(n.intValue() == ((Integer)it.next()).intValue())
124                 return true;
125         }
126         return false;
127     }
128
```

```
129
130         /**
131          * Checks whether or not a set contains all the nodes of the
132          * current graph.
133          * @param s
134          * @return true is returned iff all nodes in the graph are in s,
135          * false otherwise.
136          */
137         public boolean all(Set<Integer> s){
138             int i= 0;
139 
140             while(i < vertices.length){
141                 if (isIn(s,i) == false){
142                     return false;
143                 }
144                 i++;
145 
146             }
147             return true;
148         }
149 
150 
151 
152         /**
153          * Checks whether o not the current graph is strongly connected.
154          * @return True is returned iff the current graph is strongly
155          * connected, false otherwise.
156          */
157         public boolean stronglyConnected(){
158             for(int i = 0; i< vertices.length; i++){
159                 Set<Integer> visited = new java.util.HashSet<Integer>();
160                 visited = dfs(i, visited);
161                 visited.add(i);
162                 if(!all(visited)){
163                     return false;
164                 }
165             }
166             return true;
167         }
168 
169 
170         /**
171          * Checks whether o not the current graph has a least one loop.
172          * @return True iff a loop is found, false otherwise.
173          */
174         public boolean cyclic(){
175             for(int i = 0; i< vertices.length; i++){
176                 Set<Integer> visited = new java.util.HashSet<Integer>();
177                 visited = dfs(i, visited);
178                 if(isIn(visited, i))
179                     return true;
180             }
181             return false;
182 
183         }
184 
185         /**
186          * Checks whether o not the current graph is empty.
187          * @return True iff the current graph is empty, false otherwise.
188          */
189 
190         public boolean isEmpty() {
191             for (int i=0; i<vertices.length; i++) {
192                 if (vertices[i].adjacent.header!=null) return false;
193             }
194             return true;
195         }
196 
```

```
197     /**
198      * Checks whether o not the current labelled graph
199      * contains at least one  negative weight.
200      * @return true iff a negative weight is found in the
201      * current graph, false otherwise.
202      */
203     public boolean negativeWeight() {
204         for (int i = 0;i<vertices.length;i++) {
205             if (vertices[i].negativeWeight())
206                 return true;
207         }
208         return false;
209     }
210 
211 
212     /**
213      * Checks whether o not the current graph is complete.
214      * @return True iff the current graph is complete,
215      * false otherwise.
216      */
217     public boolean complete() {
218         for (int i=0; i<vertices.length; i++) {
219             int numArcs = 0;
220             for (Entry current = vertices[i].adjacent.header; current !=
                     null; current = current.next) {
221                 numArcs++;
222             }
223             if (numArcs!=size)
224                 return false;
225         }
226         return true;
227     }
228 
229     /**
230      * Checks whether o not the current graph is dense.
231      * @return True is returned iff the graph is dense,
232      * false otherwise.
233      */
234     public boolean isDense() {
235         int numArcs = 0;
236         for (int i=0; i<vertices.length; i++) {
237             for (Entry current = vertices[i].adjacent.header; current !=
                     null; current = current.next) {
238                 numArcs++;
239                 if (numArcs>((size*(size-1))/2)) return true;
240             }
241         }
242         return false;
243     }
244 
245 
246 
247     /**getClass translates a given array of boolean into a Integer.
248      * @param classes The array to be translated.
249      * @return The integer that represents the given array.
250      * @see eqClass()
251      */
252     public BigInteger getClass(boolean [] classes){
253         BigInteger res = new BigInteger("0");
254         BigInteger dos =new BigInteger("2");
255         for (int i=0; i<classes.length; i++) {
256             if (classes[i]){
257                 BigInteger t = dos.pow(i);
258                 res = res.add(t);
259             }
260         }
261         return res;
262     }
```

```
263
264     /**eqClass given a LabelledAltgraph it returns the equivalence
265      * class in which this graph belongs to. In this case,
266      * the equivalence classes are given by all the possible
267      * combinations of predicates isEmpty(), complete(), isDense(),
268      * negativeWeight(),cyclic  and stronglyConnected().
269      * @return The equivalence class in which the current graph
270      * belongs.
271      */
272     public BigInteger eqClass(){
273         boolean[] classes = new boolean[6];
274         classes[0] = isEmpty();
275         if (classes[0]) {
276             classes[1] = false; // if empty, then it's not dense
277             classes[2] = false; // if empty, not cyclic
278             classes[3] = false; // if empty, not strongly connected
279             classes[4] = false; // if empty, no negative weights
280             classes[5] = true;  // if empty, not complete
281         }
282         else {
283             classes[1] = isDense();
284             classes[2] = cyclic();
285             classes[3] = stronglyConnected();
286             classes[4] = negativeWeight();
287             if (classes[1])
288                 classes[5] = complete();
289             else
290                 classes[5] = false; // if not dense, then not complete
291         }
292         return this.getClass(classes);
293     }
294 }
```

```
1  package koratPlus.LabelledAltGraph;
2
3  /**
4   * Class vertex defines vertex of the LabelledAltGraph
5   * (Weighted Directed graph).
6   * @author Nazareno M. Aguirre
7   *
8   */
9
10 public class Vertex {
11
12     public int label;
13
14     public SortedSinglyLinkedList adjacent;
15
16
17     /**
18      *check if adjacent  satisfies the sorted list's
19      *representation invariant.
20      * @return True iff the current vertex satisfies its
21      * representation invariant.
22      */
23     public boolean repOK() {
24         if (adjacent==null) return false;
25         return (adjacent.repOK());
26     }
27
28     /**
29      * Checks whether o not the adjacent list associated to the
30      * current vertex  contains at least one  negative weight.
31      * @return true iff a negative weight is found in the current
32      * vertex's adj. list, false otherwise.
33      */
34     public boolean negativeWeight(){
35         for (Entry curr=adjacent.header; curr!=null; curr=curr.next) {
```

```
36              if(curr.transLabel<0){
37                  return true;
38              }
39          }
40          return false;
41      }
42
43
44  }
```

```
1   package koratPlus.LabelledAltGraph;
2
3   import java.util.Set;
4
5   /**
6    * Class  Sorted SinglyLinkedList defines Sorted Singly linked List
7    * @author
8    */
9
10  public class SortedSinglyLinkedList {
11
12      public Entry header;
13
14      /**
15       * RepOk checks whether the sortedsinglyLinkedList satisfies its
16       * representation invariant (acyclicity and sortedness).
17       * @return True iff the current list satisfies its representation
18       * invariant, false otherwise.
19       */
20
21      public boolean repOK() {
22          return repOkSorted();
23      }
24
25
26      public int length(){
27          Entry current = header;
28          int count=0;
29          while (current!=null) {
30              count++;
31          }
32          return count;
33
34      }
35      public boolean repOkCommon() {
36          Set<Entry> visited = new java.util.HashSet<Entry>();
37          Entry current = header;
38          while (current!=null) {
39              if (!visited.add(current))
40                  return false;
41              if (current.element == null)
42                  return false;
43              current = current.next;
44          }
45          return true;
46      }
47
48      public boolean repOkSorted() {
49          if (!repOkCommon()) return false;
50          for (Entry current = header; current != null; current = current.
                next) {
51              if (current.next!=null) {
52                  if (current.element.label>=current.next.element.label)
                        return false;
53              }
54          }
55          return true;
56      }
```

```
57
58
59      public String toString() {
60          String res = "(";
61          if (header != null) {
62              Entry cur = header.next;
63              while (cur != null && cur != header) {
64                  res += cur.toString();
65                  cur = cur.next;
66              }
67          }
68          return res + ")";
69      }
70
71  }
```

```
1   package koratPlus.LabelledAltGraph;
2
3   /**
4    * Entries of the SortedSinglyLinkedList class.
5    * @author
6    */
7
8   public class Entry {
9
10      public Vertex element;
11
12      public int transLabel;
13
14      public Entry next;
15
16      public String toString() {
17          return "[" + (element != null ? element.toString() : "null") + "
                ]";
18      }
19  }
```

## Search Trees (Delete)

```
1   /**
2    * Class that defines SearchTrees. In this class a  new integer
3    * attribute ('param') was added since, in this case, what we
4    * want to test is the search method (black box coverage) and
5    * we want korat to generate  all valid entries to test the
6    * search routine on searchTree, which are pairs of SearchTrees
7    * and integers.
8    * This class needs to implement the Serializable interface
9    * since it is required for the korat "serialize" option.
10   * This class needs to  implement the Serializable interface since it
11   * is required for the korat "serialize" option.
12   */
13
14  package koratPlus.SearchTreeBB;
15
16  import java.util.HashSet;
17  import java.util.LinkedList;
18  import java.util.Set;
19
20  import java.math.BigInteger;
21
22  import korat.finitization.IClassDomain;
23  import korat.finitization.IFinitization;
24  import korat.finitization.IIntSet;
25  import korat.finitization.IObjSet;
26  import korat.finitization.impl.FinitizationFactory;
27  import java.io.*;
28
```

```
29  public class SearchTreeBB implements Serializable{
30
31      public  static final long serialVersionUID = 1;
32
33
34      /**
35       * Parameter to be searched. This new attribute is
36       * necessary since we want korat to generate
37       * all valid entries to test the search routine on SearchTree,
38       * which are pairs of SearchTrees and integers.
39       */
40
41      public Integer param = new Integer(0);
42
43      /**
44       *root node
45       */
46      public Node root;
47
48      /**
49       *number of nodes in the tree
50       */
51      public int size;
52
53      public SearchTreeBB(Node r){
54          root = r;
55      }
56
57      /**
58       * Checks that the current SearchTree satisfies its
59       * representation invariant, which means checking that
60       * it is a well built tree and the data is ordered.
61       * @return True iff this searchTree satisfies the
62       * corresponding representation invariant.
63       */
64
65      public boolean repOK() {
66          if (param == null)
67              return false;
68          /**checks that empty tree has size zero*/
69          if (root == null)
70              return size == 0;
71          /**checks that the input is a tree*/
72          if (!isAcyclic())
73              return false;
74          /**checks that size is consistent*/
75          if (numNodes(root) != size)
76              return false;
77          /**checks that data is ordered*/
78          if (!isOrdered(root))
79              return false;
80          return true;
81      }
82
83      /**
84       *checks that the tree has no cycle
85       * @return true iff the tree has no cycle
86       */
87      private boolean isAcyclic() {
88          Set visited = new HashSet();
89          visited.add(root);
90          LinkedList workList = new LinkedList();
91          workList.add(root);
92          while (!workList.isEmpty()) {
93              Node current = (Node) workList.removeFirst();
94              if (current.left != null) {
95                  if (!visited.add(current.left))
96                      return false;
```

```
97                  workList.add(current.left);
98              }
99              if (current.right != null) {
100                 if (!visited.add(current.right))
101                     return false;
102                 workList.add(current.right);
103             }
104         }
105         return true;
106     }
107
108     /**
109      * counts the number of nodes in the tree starting from 'n'
110      * @param n the starting node
111      * @return the number of nodes in the current tree
112      */
113     private int numNodes(Node n) {
114         if (n == null)
115             return 0;
116         return 1 + numNodes(n.left) + numNodes(n.right);
117     }
118
119
120     private int compare(Object k1, Object k2) {
121         return ((Comparable) k1).compareTo(k2);
122     }
123
124     private boolean isOrdered(Node n) {
125         return isOrdered(n, null, null);
126     }
127
128     private boolean isOrdered(Node e, Object min, Object max) {
129         if (e.info.intValue() == -1)
130             return false;
131         if (((min != null) && (compare(e.info.intValue(), min) <= 0))
132                 || ((max != null) && (compare(e.info.intValue(), max) >=
                        0)))
133             return false;
134         if (e.left != null)
135             if (!isOrdered(e.left, min, e.info.intValue()))
136                 return false;
137         if (e.right != null)
138             if (!isOrdered(e.right,e.info.intValue(), max))
139                 return false;
140         return true;
141     }
142
143
144     /**
145      * finSearchTreeBB provides a bound on the number of objects to be
146      * used to generate instances of SearchTreeBB.
147      * @param numNodes number of entries to be used to generate instance
148      * of SearchTreeBB.
149      * @return  the object Korat needs for setting up the bounds during
150      * the search.
151      * @throws Exception
152      */
153     public static IFinitization finSearchTreeBB(int numNodes) throws
            Exception {
154         return finSearchTreeBB(numNodes, numNodes, numNodes, numNodes);
155     }
156
157
158
159     /**
160      * finSearchTreeBB provides a bound on the number of objects to be
161      * used to generate instances of SearchTreeBB.
162      * @param numNodes number of entries to be used to generate instance
```

```
163         * of SearchTreeBB.
164         * @param minSize minimum size of the generated SearchTreeBB
165         * @param maxSize maximum size of the generated SearchTreeBB
166         * @param numKeys The range of keys of each entry goes between 1 and
167         * numKeys.
168         * @return  the object Korat needs for setting up the bounds during
169         * the search.
170         * @throws Exception
171         */
172        public static IFinitization finSearchTreeBB(int numNodes, int
               minSize,
173                int maxSize, int numKeys) throws Exception {
174
175            IFinitization f = FinitizationFactory.create(SearchTreeBB.class)
                   ;
176
177            IObjSet nodes = f.createObjSet(Node.class, numNodes);
178            nodes.setNullAllowed(true);
179
180            IIntSet sizes = f.createIntSet(minSize, maxSize);
181
182            IObjSet keys = f.createObjSet(Integer.class);
183            IClassDomain elemsClassDomain = f.createClassDomain(Integer.
                   class);
184            elemsClassDomain.includeInIsomorphismCheck(false);
185            for (int i = 1; i <= numKeys; i++)
186                elemsClassDomain.addObject(new Integer(i));
187            keys.addClassDomain(elemsClassDomain);
188            keys.setNullAllowed(false);
189
190            f.set("root", nodes);
191            f.set("size", sizes);
192            f.set("Node.left", nodes);
193            f.set("Node.right", nodes);
194            f.set("Node.info", keys);
195            f.set("param", keys);
196
197            return f;
198
199        }
200
201        /**getClass translates a given array of boolean into a Integer
202         * @param classes The array to be translated
203         * @return The integer that represents the given array
204         * @see eqClass()
205         */
206        public BigInteger getClass(boolean [] classes){
207            BigInteger res = new BigInteger("0");
208            BigInteger dos =new BigInteger("2");
209            for (int i=0; i<classes.length; i++) {
210                if (classes[i]){
211                    BigInteger t = dos.pow(i);
212                    res = res.add(t);
213                }
214            }
215            return res;
216
217        }
218
219        /**
220         * @return The node containing 'param' in its field info.
221         *if it is not found the method returns null.
222         */
223
224        public Node Search() {
225            Node p = root;
226            while (p != null) {
227                if (param < p.info)
```

```
228                 p = p.left;
229
230             else{
231                 if (param > p.info)
232                     p = p.right;
233                 else
234                     return p;
235             }
236         }
237         return null;
238     }
239
240     /**
241      * checks whether or not the given node is a leaf (has no children)
242      * @param b the node to be checked
243      * @return true iff the node 'b' has not children
244      */
245     public boolean isLeaf(Node b) {
246         return (b.left ==null) && (b.right ==null);
247     }
248
249     /**
250      * checks whether or not the given node has both left and right
251      * children.
252      * @param b the node to be checked
253      * @return true iff the node 'b' has left and right children
254      */
255     public boolean hasTwoChildren(Node b) {
256         return (b.left !=null) && (b.right !=null);
257     }
258
259     /**
260      * checks whether or not the given node has left child but not
261      * the right one.
262      * @param b the node to be checked
263      * @return true iff the node 'b' has left child and not right child
264      */
265     public boolean hasJustLeftChild(Node b) {
266         return (b.left !=null) && (b.right ==null);
267     }
268
269     /**
270      * checks whether or not the given node has right child but not the
           left one.
271      * @param b the node to be checked
272      * @return true iff the node 'b' has right child and not left child
273      */
274     public boolean hasJustRightChild(Node b) {
275         return (b.left ==null) && (b.right !=null);
276     }
277
278
279     /**
280      * checks whether or not the given node is root
281      * @param b the node to be checked
282      * @return true iff the node 'b' is the root of th current tree
283      */
284     public boolean isRoot(Node n) {
285         return (n!=null && n == root);
286     }
287
288
289     public String toString() {
290         StringBuffer buf = new StringBuffer();
291         buf.append(size);
292         buf.append("{");
293         if (root != null)
294             buf.append(root.toStrings());
```

```
295             buf.append("}");
296             return buf.toString();
297         }
298 
299 
300         /** eqClass returns the equivalence class in which the current
301          *  instance of searchTreeBB belongs to. In this case, the
302          *  equivalence classes are given by all the possible combinations
303          *  of the following predicates apply on the node containing
304          *  'param' as info: isNotOntheTree, isLeaf, hasTwoChildren,
305          *  hasJustLeftChild, hasJustRightChild and isRoot.
306          * @return The equivalence class in which this instance belongs
307          */
308         public BigInteger eqClass(){
309             boolean[] classes = new boolean[6];
310             Node b = this.Search();
311             classes[0] = (b==null); /**param is not in the tree*/
312             classes[1] = (b!=null) && (this.isLeaf(b));
313             classes[2] = (b!=null) && (this.hasTwoChildren(b));
314             classes[3] = (b!=null) && (this.hasJustLeftChild(b));
315             classes[4] = (b!=null) && (this.hasJustRightChild(b));
316             classes[5] = isRoot(b);
317 
318             return this.getClass(classes);
319         }
320 
321 
322         public void delete() {
323             root = remove(root);
324         }
325 
326         /**
327          * Internal method to remove from a subtree.
328          * @param x the item to remove.
329          * @param t the node that roots the tree.
330          * @return the new root.
331          */
332         protected Node remove(Node t) {
333             if( t == null ){
334                 return null;
335             }
336 
337             if( param.intValue() < t.info )
338                 t.left = remove(t.left);
339             else{
340                 if( param.intValue() > t.info )
341                     t.right = remove(t.right);
342                 else{
343                     if( t.left != null && t.right != null ){
344                         t.info = findMin( t.right ).info;
345                         t.right = removeMin( t.right);
346                     }
347                     else{
348                         if(t.left != null){
349                             size--;
350                             t=t.left;
351                         }else{
352                             size--;
353                             t= t.right;
354                         }
355                     }
356 
357                 }
358             }
359             return t;
360         }
361 
362         /**
```

```
363         * Internal method to remove minimum item from a subtree.
364         * @param t the node that roots the tree.
365         * @return the new root.
366         * @throws ItemNotFoundException if x is not found.
367         */
368        protected Node removeMin(Node t) {
369            if( t == null ){
370                return null;
371            }
372            else{
373
374                if( t.left != null ) {
375                    t.left = removeMin( t.left);
376                    return t;
377                } else{
378                    size--;
379                    return t.right;
380                }
381            }
382        }
383
384        /**
385         * Internal method to find the smallest item in a subtree.
386         * @param t the node that roots the tree.
387         * @return node containing the smallest item.
388         */
389        protected Node findMin(Node t ) {
390            if( t != null )
391                while( t.left != null )
392                    t = t.left;
393            return t;
394        }
395
396
397    }//End Class
```

```
1   package koratPlus.SearchTreeBB;
2
3   import java.util.Set;
4   import java.util.HashSet;
5   import java.io.*;
6
7   /** This class needs to  implement the Serializable interface
8    * since it is required for the korat "serialize" option.
9    */
10  public class Node implements Serializable{
11
12      public  static final long serialVersionUID = 1;
13
14      /**
15       *left child
16       */
17      Node left;
18
19
20      /**
21       *right child
22       */
23      Node right;
24
25
26      /**
27       * data
28       */
29      Integer info;
30
31      Node(Node left, Node right, int info) {
32          this.left = left;
```

```
33          this.right = right;
34          this.info = info;
35      }
36
37      Node(Integer i) {
38          this.info = i;
39      }
40
41      public Node(){}
42
43      public boolean equals(Object that) {
44          if (!(that instanceof Node))
45              return false;
46          Node n = (Node) that;
47          if (this.info.intValue() > (n.info.intValue()))
48              return false;
49          boolean b = true;
50          if (left == null)
51              b = b && (n.left == null);
52          else
53              b = b && (left.equals(n.left));
54          if (right == null)
55              b = b && (n.right == null);
56          else
57              b = b && (right.equals(n.right));
58          return b;
59      }
60
61
62      public String toStrings() {
63          Set visited = new HashSet();
64          visited.add(this);
65          return tostring(visited);
66      }
67
68      private String tostring(Set visited) {
69          StringBuffer buf = new StringBuffer();
70          buf.append(" {");
71          if (left != null)
72              if (visited.add(left))
73                  buf.append(left.tostring(visited));
74              else
75                  buf.append("!tree");
76
77          buf.append("" + this.info + "");
78
79          if (right != null)
80              if (visited.add(right))
81                  buf.append(right.tostring(visited));
82              else
83                  buf.append("!tree");
84          buf.append("} ");
85          return buf.toString();
86      }
87
88  }
```

### Insertion and Search in RedBlackTrees

```
1   package koratPlus.RedBlackTreeWB;
2
3   import java.util.Set;
4   import java.util.Arrays;
5   import korat.finitization.IClassDomain;
6   import korat.finitization.IFinitization;
7   import korat.finitization.IIntSet;
8   import korat.finitization.IObjSet;
```

```
 9  import korat.finitization.impl.FinitizationFactory;
10
11
12  import java.util.Arrays;
13  import java.math.BigInteger;
14
15  /**
16   * Class that defines Red Black trees. In this class the methods
17   * insert and search were instrumented to be used as eqClass.
18   * (White Box coverage criteria).
19   * @author
20   */
21  public class RedBlackTreeWB {
22
23      public static class Node  {
24
25          Integer key;
26          int value;
27
28          int color = BLACK;
29
30          Node left = null;
31
32          Node right = null;
33
34          Node parent;
35
36          public Node(Integer key_param, Node parent_param) {
37              key = key_param;
38              parent = parent_param;
39          }
40
41          public Node(){}
42
43      }
44
45      /**
46       * parameter to be inserted/searched. This new attribute is
47       * necessary since we want korat to generate
48       * all valid entries to test the insert and search routines on RBT,
49       * which are pairs of RedBlackTrees and integers.
50       */
51      private Integer param = new Integer(0);
52
53      private Node root = null;
54
55      private int size = 0;
56
57      private static final int RED = 0;
58
59      private static final int BLACK = 1;
60
61
62
63      public RedBlackTreeWB(Node r){
64          root = r;
65      }
66
67      public static IFinitization finRedBlackTreeWB(int size) {
68          return finRedBlackTreeWB(size, size, size, size);
69      }
70
71
72      /**
73       * finRedBlackTreeWB provides a bound on the number of objects to be
74       * used to generate instances of RedBlackTreeWB.
75       * @param numEntries number of entries to be used to generate
76       * instance  of RedBlackTreeWB.
```

```
77          * @param minSize minimum size of the generated RedBlackTreeWB
78          * @param maxSize maximum size of the generated RedBlackTreeWB
79          * @param numKeys The range of keys of each entry goes between 1 and
80          * numKeys.
81          * @return the object Korat needs for setting up the bounds during
82          * the search.
83          */
84         public static IFinitization finRedBlackTreeWB(int numEntries, int
                minSize,
85                 int maxSize, int numKeys) {
86             IFinitization f = FinitizationFactory.create(RedBlackTreeWB.
                    class);
87
88             IClassDomain entryDomain = f.createClassDomain(Node.class,
                    numEntries);
89             IObjSet entries = f.createObjSet(Node.class, numEntries,true);
90             entries.addClassDomain(entryDomain);
91             IIntSet sizes = f.createIntSet(minSize, maxSize);
92
93             IObjSet keys = f.createObjSet(Integer.class);
94
95             IClassDomain elemsClassDomain = f.createClassDomain(Integer.
                    class);
96             elemsClassDomain.includeInIsomorphismCheck(false);
97             for (int i = 1; i <= numKeys; i++)
98                 elemsClassDomain.addObject(new Integer(i));
99             keys.addClassDomain(elemsClassDomain);
100            keys.setNullAllowed(false);
101
102            IIntSet values = f.createIntSet(0);
103
104            IIntSet colors = f.createIntSet(0, 1);
105
106            f.set("root", entries);
107            f.set("size", sizes);
108            f.set("Node.left", entries);
109            f.set("Node.right", entries);
110            f.set("Node.parent", entries);
111            f.set("Node.color", colors);
112            f.set("Node.value", values);
113
114            f.set("Node.key", keys);
115            f.set("param", keys);
116
117            return f;
118
119        }
120
121
122        /**
123         * checks that the current redBlackTree satisfies its representation
124         * invariant, which means checking that it is a well built tree, is
125         * balanced according to the color of its nodes and it is a search
126         * tree.
127         * @return true iff this redBlackTree satisfies the corresponding
128         * representation invariant.
129         */
130
131        public boolean repOK() {
132            if(this.param == null)
133                return false;
134            if (root == null)
135                return size == 0;
136            // RootHasNoParent
137            if (root.parent != null)
138                return debug("RootHasNoParent");
139            Set visited = new java.util.HashSet();
140            visited.add(new Wrapper(root));
```

```
141         java.util.LinkedList workList = new java.util.LinkedList();
142         workList.add(root);
143         while (!workList.isEmpty()) {
144             Node current = (Node) workList.removeFirst();
145             // Acyclic
146             // // if (!visited.add(new Wrapper(current)))
147             // // return debug("Acyclic");
148             // Parent Definition
149             Node cl = current.left;
150             if (cl != null) {
151                 if (!visited.add(new Wrapper(cl)))
152                     return debug("Acyclic");
153                 if (cl.parent != current)
154                     return debug("parent_Input1");
155                 workList.add(cl);
156             }
157             Node cr = current.right;
158             if (cr != null) {
159                 if (!visited.add(new Wrapper(cr)))
160                     return debug("Acyclic");
161                 if (cr.parent != current)
162                     return debug("parent_Input2");
163                 workList.add(cr);
164             }
165         }
166         // SizeOk
167         if (visited.size() != size)
168             return debug("SizeOk");
169         if (!repOkColors()){
170             return false;
171         }
172         return repOkKeysAndValues();
173     }
174 
175 
176     /**
177      * Checks whether this RedBlackTree is balanced according the color
178      * of its nodes.
179      * @return true iff Root is black,Red nodes only have black children
180      * and all the path of the tree.
181      * from root to null have the same numbers of black nodes.
182      */
183     private boolean repOkColors() {
184 
185         if((root!=null) && root.color == RED){
186             return debug("Root must be Black");
187         }
188         java.util.LinkedList workList = new java.util.LinkedList();
189         workList.add(root);
190         while (!workList.isEmpty()) {
191             Node current = (Node) workList.removeFirst();
192             Node cl = current.left;
193             Node cr = current.right;
194             if (current.color == RED) {
195                 if (cl != null && cl.color == RED)
196                     return debug("RedHasOnlyBlackChildren1");
197                 if (cr != null && cr.color == RED)
198                     return debug("RedHasOnlyBlackChildren2");
199             }
200             if (cl != null)
201                 workList.add(cl);
202             if (cr != null)
203                 workList.add(cr);
204         }
205         /**SimplePathsFromRootToNILHaveSameNumberOfBlackNodes*/
206         int numberOfBlack = -1;
207         workList = new java.util.LinkedList();
208         workList.add(new Pair(root, 0));
```

```
209             while (!workList.isEmpty()) {
210                 Pair p = (Pair) workList.removeFirst();
211                 Node e = p.e;
212                 int n = p.n;
213                 if (e != null && e.color == BLACK)
214                     n++;
215                 if (e == null) {
216                     if (numberOfBlack == -1)
217                         numberOfBlack = n;
218                     else if (numberOfBlack != n)
219                         return debug("
                                SimplePathsFromRootToNILHaveSameNumberOfBlackNodes
                                ");
220                 } else {
221                     workList.add(new Pair(e.left, n));
222                     workList.add(new Pair(e.right, n));
223                 }
224             }
225             return true;
226         }
227 
228         /**
229          * checks whether this tree is a binary search tree.
230          * @return true iff this tree is a binary search tree
231          * (Its keys are ordered).
232          */
233         private boolean repOkKeysAndValues() {
234             /**BST1 and BST2
235             this was the old way of determining if the keys are ordered
236             java.util.LinkedList workList = new java.util.LinkedList();
237             workList = new java.util.LinkedList();
238             workList.add(root);
239             while (!workList.isEmpty()) {
240             Entry current = (Entry)workList.removeFirst();
241             Entry cl = current.left;
242             Entry cr = current.right;
243             if (current.key==current.key) ;
244             if (cl != null) {
245             if (compare(current.key, current.maximumKey()) <= 0)
246             return debug("BST1");
247             workList.add(cl);
248             }
249              if (cr != null) {
250              if (compare(current.key, current.minimumKey()) >= 0)
251              return debug("BST2");
252              workList.add(cr);
253              }
254              }
255              this is the new (Alex's) way to determine if the keys are
                    ordered*/
256             if (!orderedKeys(root, null, null))
257                 return debug("BST");
258             // touch values
259             java.util.LinkedList workList = new java.util.LinkedList();
260             workList.add(root);
261             while (!workList.isEmpty()) {
262                 Node current = (Node) workList.removeFirst();
263 
264                 if (current.left != null)
265                     workList.add(current.left);
266                 if (current.right != null)
267                     workList.add(current.right);
268             }
269             return true;
270         }
271 
272         private boolean orderedKeys(Node e, Object min, Object max) {
273             if (e.key.intValue() == -1)
```

```
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 
359     /** Insertion method instrumented to be used as a eqClass during
360      * the pruning (Desicion Coverage).
361      * The element to be added is 'param'
362      * @return the equivalence class in which this instance belong to.
363      */
364     public BigInteger eqClassInsertionDC() {
365         boolean [] classes = new boolean[100];
366         init(classes);//it sets all array position in false
367         Node t = root;
368         if (t == null) {
369             classes[0] = true;
370             root = new Node(param, null);
371             size = 1;
372             return this.getClass(classes);
373         }else classes[1] = true;
374         Node parent;
375         int cmp;
376 
377         int i = 0;
378         do {
379             if(i != 0){
380                 classes[6] = true;
381             }
382             parent = t;
383             if (t.key >param) {
384                 t = t.left;
385                 cmp = -1;
386                 classes[2] = true;
387             }else{
388                 classes[3] = true;
389                 if (t.key< param) {
390                     t = t.right;
391                     cmp = +1;
392                     classes[4] = true;
393                 }else{
394                     classes[5] = true;
395                     return this.getClass(classes);
396                 }
397             }
398             i++;
399         }while (t != null);
400         classes[7] = true;
401 
402         Node x = new Node(param, parent);
403         if (cmp < 0){
404             parent.left = x;
405             classes[8] = true;
406         }
407         else{
408             parent.right = x;
```

```
409             classes[9] = true;
410         }
411         //fixAfterInsertion
412         x.color = RED;
413         Node n1,n2,n3,n4 = null;
414 
415         while (x != null && x != root && x.parent.color == RED) {
416             classes[98] = true;
417             if(x == null){
418                 classes[10] = true;
419                 n1 = null; //parentOf(x)
420             }
421             else{
422                 classes[11] = true;
423                 n1 = x.parent;
424             }
425             if(n1 == null){
426                 classes[12] = true;
427                 n2 = null; //parentOf(parentOf(x)) parentOf(n1)
428             }
429             else{
430                 classes[13] = true;
431                 n2 = n1.parent;
432             }
433             Node d1= null;//leftOf(parentOf(parentOf(x))) leftOf(n2)
434             if(n2 != null){
435                 classes[14] = true;
436                 d1 =n2.left;
437             }else classes[15] = true;
438             if (n1 == d1) {
439                 classes[16] = true;
440                 Node y =null;
441                 if(n2 != null){ //Node y = rightOf(n2);
442                     classes[18] = true;
443                     y= n2.right;
444                 }else classes[19] = true;
445                 //colorOf(y)
446                 int cs = 0;
447                 if(y == null){
448                     classes[20] = true;
449                     cs = BLACK;
450                 }else{
451                     classes[21] = true;
452                     cs = y.color;
453                 }
454                 if (cs == RED) {
455                     classes[22] = true;
456                     if (n1 != null){
457                         classes[24] = true;
458                         n1.color = BLACK;
459                     }else classes[25] = true;
460                     if (y != null){
461                         classes[26] = true;
462                         y.color = BLACK;
463                     }else classes[27] = true;
464                     if (n2 != null){
465                         classes[28] = true;
466                         n2.color = RED;
467                     }else classes[29] = true;
468                     x = n2;
469                 }else {
470                     classes[23] = true;
471                     Node d = null;
472                     if(n1 != null){
473                         classes[30] = true;
474                         d= n1.right;
475                     }else classes[31] = true;
476                     if (x == d) {
```

```
477                         classes[32] = true;
478                         x = n1;
479                         //rotateLeft(x)
480                         if (x != null) {
481                             classes[34] = true;
482                             Node r = x.right;
483                             x.right = r.left;
484                             if (r.left != null){
485                                 classes[36] = true;
486                                 r.left.parent = x;
487                             }else classes[37] = true;
488                             r.parent = x.parent;
489                             if (x.parent == null){
490                                 classes[38] = true;
491                                 root = r;
492                             }else {
493                                 classes[39] = true;
494                                 if (x.parent.left == x){
495                                     x.parent.left = r;
496                                     classes[40] = true;
497                                 }else{
498                                     x.parent.right = r;
499                                     classes[41] = true;
500                                 }
501                             }
502                             r.left = x;
503                             x.parent = r;
504                         }else classes[35] = true;//end rotateLeft(x)
505 
506                     } else classes[33] = true; //f (x == rightOf(
                            parentOf(x))) if (x == d)
507                     if(x == null){
508                         classes[42] = true;
509                         n3 = null; //parentOf(x)
510                     }
511                     else{
512                         classes[43] = true;
513                         n3 = x.parent;
514                     }
515                     if(n3 == null){
516                         classes[44] = true;
517                         n4 = null; //parentOf(parentOf(x)) parentOf(n1)
518                     }
519                     else{
520                         classes[45] = true;
521                         n4 = n3.parent;
522                     }
523                     if (n3 != null){
524                         classes[46] = true;
525                         n3.color = BLACK;
526                     }else classes[47] = true;
527                     if (n4 != null){
528                         classes[48] = true;
529                         n4.color = RED;
530                     }else classes[49] = true;
531 
532                     Node n = n4;
533                     //rotateRight(n);
534                     if (n != null) {
535                         classes[50] = true;
536                         Node l = n.left;
537                         n.left = l.right;
538                         if (l.right != null){
539                             classes[52] = true;
540                             l.right.parent = n;
541                         }else classes[53] = true;
542 
543                         l.parent = n.parent;
```

```
544                         if (n.parent == null){
545                             classes[54] = true;
546                             root = l;
547                         }else{
548                             classes[55] = true;
549                             if (n.parent.right == n){
550                                 classes[56] = true;
551                                 n.parent.right = l;
552                             }else{
553                                 classes[57] = true;
554                                 n.parent.left = l;
555                             }
556                         }
557                         l.right = n;
558                         n.parent = l;
559                     }else classes[51] = true;//end rotateRight(n);
560                 }//else (colorOf(y) == RED)
561             }else { //parentOf(x) == leftOf(parentOf(parentOf(x))) if (
                    n1 == d1)
562                 classes[17] = true;
563                 Node y = null;
564                 if(n2 != null){
565                     classes[58] = true;
566                     y= n2.left;
567                 }else classes[59] = true;
568                 //colorOf(y)
569                 int cs = 0;
570                 if(y == null) {
571                     classes[60] = true;
572                     cs = BLACK;
573                 }else{
574                     classes[61] = true;
575                     cs = y.color;
576                 }
577                 if (cs == RED) {
578                     classes[62] = true;
579                     if (n1 != null){
580                         classes[64] = true;
581                         n1.color = BLACK;
582                     }else classes[65] = true;
583                     if (y != null){
584                         classes[66] = true;
585                         y.color = BLACK;
586                     }else classes[67] = true;
587                     if (n2 != null){
588                         classes[68] = true;
589                         n2.color = RED;
590                     }else classes[69] = true;
591                     x = n2;
592                 } else {
593                     classes[63] = true;
594                     Node d= null;
595                     if(n1 != null){
596                         classes[70] = true;
597                         d = n1.left;
598                     }else classes[71] = true;
599                     if (x == d) {
600                         classes[72] = true;
601                         x = n1;
602                         //rotateRight(x);
603                         if (x != null) {
604                             classes[74] = true;
605                             Node l = x.left;
606                             x.left = l.right;
607                             if (l.right != null){
608                                 classes[76] = true;
609                                 l.right.parent = x;
610                             }else classes[77] = true;
```

```
611                         l.parent = x.parent;
612                         if (x.parent == null){
613                             classes[78] = true;
614                             root = l;
615                         }else{
616                             classes[79] = true;
617                             if (x.parent.right == x){
618                                 classes[80] = true;
619                                 x.parent.right = l;
620                             }
621                             else{
622                                 classes[81] = true;
623                                 x.parent.left = l;
624                             }
625                         }
626                         l.right = x;
627                         x.parent = l;
628                     }else classes[75] = true;//end //rotateRight(x);
629                 }else classes[73] = true;//end if (x == leftOf(
                        parentOf(x)))
630                 if(x == null){
631                     classes[82] = true;
632                     n3 = null; //parentOf(x)
633                 }
634                 else{
635                     classes[83] = true;
636                     n3 = x.parent;
637                 }
638                 if(n3 == null){
639                     classes[84] = true;
640                     n4 = null; //parentOf(parentOf(x)) parentOf(n1)
641                 }
642                 else{
643                     classes[85] = true;
644                     n4 = n3.parent;
645                 }
646                 if (n3 != null){
647                     classes[86] = true;
648                     n3.color = BLACK;
649                 }else classes[87] = true;
650 
651                 if (n4 != null){
652                     classes[88] = true;
653                     n4.color = RED;
654                 }else    classes[89] = true;
655 
656                 Node p = n4;
657                 //rotateLeft(p);
658                 if (p != null) {
659                     classes[90] = true;
660                     Node r = p.right;
661                     p.right = r.left;
662                     if (r.left != null){
663                         classes[92] = true;
664                         r.left.parent = p;
665                     }else classes[93] = true;
666 
667                     r.parent = p.parent;
668                     if (p.parent == null){
669                         classes[94] = true;
670                         root = r;
671                     }else{
672                         classes[95] = true;
673                         if (p.parent.left == p){
674                             classes[96] = true;
675                             p.parent.left = r;
676                         }
677                         else{
```

```
678                                     classes[97] = true;
679                                     p.parent.right = r;
680                                 }
681                             }
682                             r.left = p;
683                             p.parent = r;
684                         }else classes[91] = true;//end rotateLeft(p);
685 
686                     }
687                 }
688             }//end while
689             classes[99] = true;
690             root.color = BLACK;
691             //endfixAfterInsertion
692             size++;
693             return this.getClass(classes);
694         }
695 
696 
697         /** Search method instrumented to be used as a eqClass during
698          * the pruning (Desicion Coverage).
699          * The element to be searched is 'param'
700          * @return the equivalence class in which this instance belong to.
701          */
702         public BigInteger  eqClassSearchDC() {
703             boolean [] classes = new boolean[6];
704             init(classes);//it sets all array position in false
705             Node p = root;
706             while (p != null) {
707                 classes[0] = true;
708                 if (param < p.key){
709                     classes[2]= true;
710                     p = p.left;
711                 }else{
712                     classes[3] = true;
713                     if (param > p.key){
714                         classes[4] = true;
715                         p = p.right;
716                     }else{
717                         classes[5] = true;
718                         return this.getClass(classes);
719                     }
720                 }
721             }
722             classes[1] = true;
723             return this.getClass(classes);
724 
725         }
726 
727 
728     }
```

### Insertion and Search in SearchTrees

```
1   package koratPlus.SearchTreeWB;
2 
3   import java.util.HashSet;
4   import java.util.LinkedList;
5   import java.util.Set;
6 
7   import java.math.BigInteger;
8 
9   import korat.finitization.IClassDomain;
10  import korat.finitization.IFinitization;
11  import korat.finitization.IIntSet;
12  import korat.finitization.IObjSet;
13  import korat.finitization.impl.FinitizationFactory;
```

```
14
15  /**
16   * Class that defines Search trees where the methods
17   * insert and search were instrumented to be used as eqClass
18   * (White Box coverage criteria).
19   * @author
20   */
21  public class SearchTreeWB {
22
23
24      public static class Node {
25
26          /**
27           *left child
28           */
29          Node left;
30
31          /**
32           *right child
33           */
34          Node right;
35
36          /**
37           * data
38           */
39          Integer info;
40
41          Node(Node left, Node right, int info) {
42              this.left = left;
43              this.right = right;
44              this.info = info;
45          }
46
47          Node(Integer i) {
48              this.info = i;
49          }
50
51          public Node(){}
52
53          public boolean equals(Object that) {
54              if (!(that instanceof Node))
55                  return false;
56              Node n = (Node) that;
57              if (this.info.intValue() > (n.info.intValue()))
58                  return false;
59              boolean b = true;
60              if (left == null)
61                  b = b && (n.left == null);
62              else
63                  b = b && (left.equals(n.left));
64              if (right == null)
65                  b = b && (n.right == null);
66              else
67                  b = b && (right.equals(n.right));
68              return b;
69          }
70
71      }
72
73      /**
74       * parameter to be inserted/searched. This new attribute is
75       * necessary since we want korat to generate
76       * all valid entries to test the insert and search routines on RBT,
77       * which are pairs of SearchTrees and integers.
78       */
79      private Integer param = new Integer(0);
80
81      /**
```

```
82          *root node
83          */
84         private Node root;
85
86         /**
87          * number of nodes in the tree
88          */
89         private int size;
90
91         public SearchTreeWB(Node r){
92             root = r;
93         }
94
95         /**
96          * checks that the current SearchTree satisfies its representation
97          * invariant, which means checking that it is a well built tree
98          * and the data is ordered.
99          * @return true iff this searchTree satisfies the corresponding
100         * representation invariant.
101         */
102        public boolean repOK() {
103            if (param == null)
104                return false;
105            /**checks that empty tree has size zero*/
106            if (root == null)
107                return size == 0;
108            /**checks that the input is a tree*/
109            if (!isAcyclic())
110                return false;
111            /**checks that size is consistent*/
112            if (numNodes(root) != size)
113                return false;
114            /**checks that data is ordered*/
115            if (!isOrdered(root))
116                return false;
117            return true;
118        }
119
120
121        /**
122         *checks that the tree has no cycle
123         * @return true iff the tree has no cycle
124         */
125        private boolean isAcyclic() {
126            Set visited = new HashSet();
127            visited.add(root);
128            LinkedList workList = new LinkedList();
129            workList.add(root);
130            while (!workList.isEmpty()) {
131                Node current = (Node) workList.removeFirst();
132                if (current.left != null) {
133                    if (!visited.add(current.left))
134                        return false;
135                    workList.add(current.left);
136                }
137                if (current.right != null) {
138                    if (!visited.add(current.right))
139                        return false;
140                    workList.add(current.right);
141                }
142            }
143            return true;
144        }
145
146        /**
147         * counts the number of nodes in the tree starting from 'n'
148         * @param n the starting node
149         * @return the number of nodes in the current tree
```

```
150         */
151        private int numNodes(Node n) {
152            if (n == null)
153                return 0;
154            return 1 + numNodes(n.left) + numNodes(n.right);
155        }
156
157        private int compare(Object k1, Object k2) {
158            return ((Comparable) k1).compareTo(k2);
159        }
160
161        private boolean isOrdered(Node n) {
162            return isOrdered(n, null, null);
163        }
164
165        private boolean isOrdered(Node e, Object min, Object max) {
166            if (e.info.intValue() == -1)
167                return false;
168            if (((min != null) && (compare(e.info.intValue(), min) <= 0))
169                    || ((max != null) && (compare(e.info.intValue(), max) >=
                            0)))
170                return false;
171            if (e.left != null)
172                if (!isOrdered(e.left, min, e.info.intValue()))
173                    return false;
174            if (e.right != null)
175                if (!isOrdered(e.right,e.info.intValue(), max))
176                    return false;
177            return true;
178        }
179
180
181
182        public String toString() {
183            StringBuffer buf = new StringBuffer();
184            buf.append("{");
185            if (root != null)
186                buf.append(root.toString());
187            buf.append("}");
188            return buf.toString();
189        }
190
191        public static IFinitization finSearchTreeWB(int numNodes) throws
                Exception {
192            return finSearchTreeWB(numNodes, numNodes, numNodes, numNodes);
193        }
194
195        /**
196         * finSearchTreeBB provides a bound on the number of objects to be
197         * used to generate instances of SearchTreeBB.
198         * @param numNodes number of entries to be used to generate
199         * instance of SearchTreeBB.
200         * @param minSize minimum size of the generated SearchTreeBB
201         * @param maxSize maximum size of the generated SearchTreeBB
202         * @param numKeys The range of keys of each entry goes between
203         * 1 and numKeys.
204         * @return  the object Korat needs for setting up the bounds
205         * during the search.
206         * @throws Exception
207         */
208        public static IFinitization finSearchTreeWB(int numNodes, int
                minSize,
209                    int maxSize, int numKeys) throws Exception {
210
211            IFinitization f = FinitizationFactory.create(SearchTreeWB.class)
                    ;
212
213            IObjSet nodes = f.createObjSet(Node.class, numNodes);
```

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

```
281                 }
282                 else{
283                     classes[3] = true;
284                     if (param > p.info){
285                         classes[4] = true;
286                         p = p.right;
287                     }else{
288                         classes[5] = true;
289                         return this.getClass(classes);
290 
291 
292                     }
293                 }
294             }
295             classes[1] = true;
296             return this.getClass(classes);
297 
298         }
299 
300         /** Insertion method instrumented to be used as a eqClass during
301          * the pruning (Desicion Coverage).
302          * @return
303          */
304         public BigInteger  eqClassInsertionDC() {
305             boolean [] classes = new boolean[10];
306             init(classes);
307             if(root ==null){
308                 classes[0] = true;
309                 root = new Node(param);
310             }else{
311                 classes[1] = true;
312                 Node p = root;
313                 Node t = root;
314                 Node n = new Node(param);
315                 while (p != null) {
316                     classes[2] = true;
317                     t = p;
318                     if (param < p.info){
319                         classes[4] = true;
320                         p = p.left;
321                     }
322                     else{
323                         classes[5] = true;
324                         if (param > p.info){
325                             classes[6] = true;
326                             p = p.right;
327                         }
328                         else{
329                             classes[7] = true;
330                             return this.getClass(classes);
331 
332                         }
333                     }
334                 }
335                 classes[3] = true;
336                 if(param < t.info){
337                     classes[8] = true;
338                     t.left = n;
339                 }
340                 else{
341                     classes[9] = true;
342                     t.right = n;
343                 }
344                 size++;
345             }
346             return this.getClass(classes);
347 
348         }
```

```
349
350 }
```

### SinglyLinkedList

```
 1  package koratPlusExtras.Lists;
 2
 3  import java.util.Set;
 4
 5  import java.math.BigInteger;
 6  import korat.finitization.IClassDomain;
 7  import korat.finitization.IFinitization;
 8  import korat.finitization.IIntSet;
 9  import korat.finitization.IObjSet;
10  import korat.finitization.impl.FinitizationFactory;
11
12  public class SinglyLinkedList{
13
14      public static class Entry {
15          Integer element;
16          Entry next;
17
18          public String toString() {
19              return "[" + (element != null ? element.toString() : "null")
                        + "]";
20          }
21      }
22
23      public Entry header;
24      private int size = 0;
25
26      public boolean repOkCommon() {
27          if (header == null)
28              return false;
29
30          if (header.element != null)
31              return false;
32          Set<Entry> visited = new java.util.HashSet<Entry>();
33          visited.add(header);
34          Entry current = header;
35
36          while (true) {
37              Entry next = current.next;
38              if (next == null)
39                  break;
40
41              if (next.element == null)
42                  return false;
43
44              if (!visited.add(next))
45                  return false;
46
47              current = next;
48          }
49
50          if (visited.size() - 1 != size)
51              return false;
52
53          return true;
54      }
55
56      public boolean repOK() {
57          if (!repOkCommon())
58              return false;
59          return true;
60      }
61      public boolean isEmpty(){
```

```
62          if(header.next ==null)
63              return true;
64          else{
65              return false;
66          }
67      }
68
69      public boolean sorted() {
70          if (size>1) {
71              for (Entry current = header.next; current.next != null;
                    current = current.next) {
72                  if (current.element.compareTo(current.next.element) >=
                        0)
73                      return false;
74              }
75          }
76          return true;
77      }
78
79      /**
80       * Checks whether or not the current list has not repeated elements.
81       * @return true iff all the elements in the list are different
82       * each other.
83       */
84      public boolean noReps(){
85          if(!isEmpty()){
86              for (Entry current = header.next; current.next != null;
                    current = current.next) {
87                  if (current.element.intValue()== current.next.
                        element.intValue())
88                      return false;
89              }
90          }
91          return true;
92      }
93
94      public BigInteger getClass(boolean [] classes){
95          BigInteger res = new BigInteger("0");
96          BigInteger dos =new BigInteger("2");
97          for (int i=0; i<classes.length; i++) {
98              if (classes[i]){
99                  BigInteger t = dos.pow(i);
100                 res = res.add(t);
101             }
102         }
103         return res;
104     }
105
106     public BigInteger eqClass(){
107         boolean[] classes = new boolean[3];
108         classes[0] = isEmpty();
109         classes[1] = sorted();
110         classes[2] = noReps();
111         return this.getClass(classes);
112     }
113
114     public String toString() {
115         String res = "(";
116         if (header != null) {
117             Entry cur = header.next;
118             while (cur != null && cur != header) {
119                 res += cur.toString();
120                 cur = cur.next;
121             }
122         }
123         return res + ")";
124     }
125
```

```
126     public static IFinitization finSinglyLinkedList(int minSize, int
            maxSize,
127             int numEntries, int numElems) {
128         IFinitization f = FinitizationFactory.create(SinglyLinkedList.
                class);
129
130         IObjSet entries = f.createObjSet(Entry.class, true);
131         entries.addClassDomain(f.createClassDomain(Entry.class,
                numEntries));
132
133         IIntSet sizes = f.createIntSet(minSize, maxSize);
134
135         IObjSet elems = f.createObjSet(Integer.class);
136         IClassDomain elemsClassDomain = f.createClassDomain(Integer.
                class);
137         elemsClassDomain.includeInIsomorphismCheck(false);
138         for (int i = 1; i <= numElems; i++)
139             elemsClassDomain.addObject(new Integer(i));
140         elems.addClassDomain(elemsClassDomain);
141         elems.setNullAllowed(true);
142
143         f.set("header", entries);
144         f.set("size", sizes);
145         f.set(Entry.class, "element", elems);
146         f.set(Entry.class, "next", entries);
147         return f;
148
149     }
150
151 }
```

```
1  package koratPlusExtras.Lists;
2
3  public  class Entry {
4          Integer element;
5
6          Entry next;
7
8          public String toString() {
9              return "[" + (element != null ? element.toString() : "null")
                       + "]";
10         }
11 }
```

## Apendix: 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 Apendix above).

| Operator | Line Number | Mutation |
|---|---|---|
| AOIS | 87 | `i => ++i` |
| AOIS | 87 | `i => --i` |
| AOIS | 87 | `i => i++` |
| AOIS | 87 | `list.size => ++list.size` |
| AOIS | 87 | `list.size => --list.size` |
| AOIS | 87 | `list.size => list.size++` |
| AOIS | 87 | `list.size => list.size--` |
| AOIS | 88 | `i => i++` |
| AOIS | 93 | `i => ++i` |
| AOIS | 93 | `i => --i` |
| AOIS | 93 | `i => i++` |
| AOIS | 93 | `set.size => ++set.size` |
| AOIS | 93 | `set.size => --set.size` |
| AOIS | 93 | `set.size => set.size++` |

| | | |
|---|---|---|
| AOIS | 93 | `set.size => set.size--` |
| AOIS | 94 | `i => i++` |
| AOIS | 100 | `i => ++i` |
| AOIS | 100 | `i => --i` |
| AOIS | 100 | `i => i++` |
| AOIS | 100 | `s.size => ++s.size` |
| AOIS | 100 | `s.size => --s.size` |
| AOIS | 100 | `s.size => s.size++` |
| AOIS | 100 | `s.size => s.size--` |
| AOIS | 101 | `i => i++` |
| ROR | 126 | `current == null  =>  current != null` |
| ROR | 131 | `current.element.intValue() != value.intValue()  =>`<br>`current.element.intValue() > value.intValue()` |
| ROR | 131 | `current.element.intValue() != value.intValue()  =>`<br>`current.element.intValue() >= value.intValue()` |
| ROR | 131 | `current.element.intValue() != value.intValue()  =>`<br>`current.element.intValue() < value.intValue()` |
| ROR | 131 | `current.element.intValue() != value.intValue()  =>`<br>`current.element.intValue() <= value.intValue()` |
| ROR | 131 | `current.element.intValue() != value.intValue()  =>`<br>`current.element.intValue() == value.intValue()` |
| COR | 120 | `current != null &&`<br>`current.element.intValue() < value.intValue()  =>`<br>`current != null ||`<br>`current.element.intValue() < value.intValue()` |
| COR | 120 | `current != null &&`<br>`current.element.intValue() < value.intValue()  =>`<br>`current != null ^`<br>`current.element.intValue() < value.intValue()` |
| COD | 95 | `!s.contains( value ) => s.contains( value )` |
| COD | 102 | `!set.contains( value ) => set.contains( value )` |
| LOI | 87 | `i => ~i` |
| LOI | 87 | `list.size => -list.size` |
| LOI | 88 | `i => ~i` |
| LOI | 93 | `i => ~i` |
| LOI | 93 | `set.size => -set.size` |
| LOI | 94 | `i => ~i` |
| LOI | 100 | `i => ~i` |
| LOI | 100 | `s.size => -s.size` |
| LOI | 101 | `i => ~i` |
| AORS | 87 | `i++ => i--` |
| AORS | 93 | `i++ => i--` |
| AORS | 100 | `i++ => i--` |

| | | |
|---|---|---|
| AOIU | 88 | `i => -i` |
| AOIU | 94 | `i => -i` |
| AOIU | 101 | `i => -i` |

**BinomailHeap(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 Apendix above).

| Operator | Line Number | Mutation |
|---|---|---|
| AORB | 38 | `this.size + h2.size => this.size * h2.size` |
| AORB | 38 | `this.size + h2.size => this.size / h2.size` |
| AORB | 38 | `this.size + h2.size => this.size % h2.size` |
| AORB | 38 | `this.size + h2.size => this.size - h2.size` |
| AOIS | 38 | `this.size => ++this.size` |
| AOIS | 38 | `this.size => --this.size` |
| AOIS | 38 | `this.size => this.size++` |
| AOIS | 38 | `this.size => this.size--` |
| AOIS | 38 | `h2.size => ++h2.size` |
| AOIS | 38 | `h2.size => --h2.size` |
| AOIS | 38 | `h2.size => h2.size++` |
| AOIS | 38 | `h2.size => h2.size--` |
| AOIS | 50 | `x.degree => ++x.degree` |
| AOIS | 50 | `x.degree => --x.degree` |
| AOIS | 50 | `x.degree => x.degree++` |
| AOIS | 50 | `x.degree => x.degree--` |
| AOIS | 50 | `nextX.degree => ++nextX.degree` |
| AOIS | 50 | `nextX.degree => --nextX.degree` |
| AOIS | 50 | `nextX.degree => nextX.degree++` |
| AOIS | 50 | `nextX.degree => nextX.degree--` |

| | | |
|---|---|---|
| AOIS | 50 | `nextX.sibling.degree => ++nextX.sibling.degree` |
| AOIS | 50 | `nextX.sibling.degree => --nextX.sibling.degree` |
| AOIS | 50 | `nextX.sibling.degree => nextX.sibling.degree++` |
| AOIS | 50 | `nextX.sibling.degree => nextX.sibling.degree--` |
| AOIS | 50 | `x.degree => ++x.degree` |
| AOIS | 50 | `x.degree => --x.degree` |
| AOIS | 50 | `x.degree => x.degree++` |
| AOIS | 50 | `x.degree => x.degree--` |
| AOIS | 57 | `x.key => ++x.key` |
| AOIS | 57 | `x.key => --x.key` |
| AOIS | 57 | `x.key => x.key++` |
| AOIS | 57 | `x.key => x.key--` |
| AOIS | 57 | `nextX.key => ++nextX.key` |
| AOIS | 57 | `nextX.key => --nextX.key` |
| AOIS | 57 | `nextX.key => nextX.key++` |
| AOIS | 57 | `nextX.key => nextX.key--` |
| AOIS | 117 | `h1.Nodes.degree => ++h1.Nodes.degree` |
| AOIS | 117 | `h1.Nodes.degree => --h1.Nodes.degree` |
| AOIS | 117 | `h1.Nodes.degree => h1.Nodes.degree++` |
| AOIS | 117 | `h1.Nodes.degree => h1.Nodes.degree--` |
| AOIS | 117 | `h2.Nodes.degree => ++h2.Nodes.degree` |
| AOIS | 117 | `h2.Nodes.degree => --h2.Nodes.degree` |
| AOIS | 117 | `h2.Nodes.degree => h2.Nodes.degree++` |
| AOIS | 117 | `h2.Nodes.degree => h2.Nodes.degree--` |
| AOIS | 129 | `h1Next.degree => ++h1Next.degree` |
| AOIS | 130 | `h1Next.degree => --h1Next.degree` |
| AOIS | 130 | `h1Next.degree => h1Next.degree++` |
| AOIS | 130 | `h1Next.degree => h1Next.degree--` |
| AOIS | 130 | `h2Next.degree => ++h2Next.degree` |
| AOIS | 130 | `h2Next.degree => --h2Next.degree` |
| AOIS | 130 | `h2Next.degree => h2Next.degree++` |
| AOIS | 130 | `h2Next.degree => h2Next.degree--` |
| ROR | 42 | `h.Nodes == null  =>  h.Nodes != null` |
| ROR | 49 | `nextX != null  =>  nextX == null` |
| ROR | 50 | `x.degree != nextX.degree  =>`<br>`x.degree > nextX.degree` |
| ROR | 50 | `x.degree != nextX.degree  =>`<br>`x.degree >= nextX.degree` |
| ROR | 50 | `x.degree != nextX.degree  =>`<br>`x.degree < nextX.degree` |
| ROR | 50 | `x.degree != nextX.degree  =>`<br>`x.degree <= nextX.degree` |
| ROR | 50 | `x.degree != nextX.degree  =>` |

| | | |
|---|---|---|
| | | `x.degree == nextX.degree` |
| ROR | 50 | `nextX.sibling != null  =>`<br>`nextX.sibling == null` |
| ROR | 50 | `nextX.sibling.degree == x.degree  =>`<br>`nextX.sibling.degree > x.degree` |
| ROR | 50 | `nextX.sibling.degree == x.degree  =>`<br>`nextX.sibling.degree >= x.degree` |
| ROR | 50 | `nextX.sibling.degree == x.degree  =>`<br>`nextX.sibling.degree < x.degree` |
| ROR | 50 | `nextX.sibling.degree == x.degree  =>`<br>`nextX.sibling.degree <= x.degree` |
| ROR | 50 | `nextX.sibling.degree == x.degree  =>`<br>`nextX.sibling.degree != x.degree` |
| ROR | 64 | `prevX == null  =>  prevX != null` |
| ROR | 104 | `h1.Nodes == null  =>  h1.Nodes != null` |
| ROR | 106 | `h2.Nodes == null  =>  h2.Nodes != null` |
| ROR | 117 | `h1.Nodes.degree <= h2.Nodes.degree  =>`<br>`h1.Nodes.degree > h2.Nodes.degree` |
| ROR | 117 | `h1.Nodes.degree <= h2.Nodes.degree  =>`<br>`h1.Nodes.degree >= h2.Nodes.degree` |
| ROR | 117 | `h1.Nodes.degree <= h2.Nodes.degree  =>`<br>`h1.Nodes.degree < h2.Nodes.degree` |
| ROR | 117 | `h1.Nodes.degree <= h2.Nodes.degree  =>`<br>`h1.Nodes.degree == h2.Nodes.degree` |
| ROR | 117 | `h1.Nodes.degree <= h2.Nodes.degree  =>`<br>`h1.Nodes.degree != h2.Nodes.degree` |
| ROR | 129 | `h1Next != null  =>  h1Next == null` |
| ROR | 129 | `h2Next != null  =>  h2Next == null` |
| ROR | 130 | `h1Next.degree <= h2Next.degree  =>`<br>`h1Next.degree > h2Next.degree` |
| ROR | 130 | `h1Next.degree <= h2Next.degree  =>`<br>`h1Next.degree >= h2Next.degree` |
| ROR | 130 | `h1Next.degree <= h2Next.degree  =>`<br>`h1Next.degree < h2Next.degree` |
| ROR | 130 | `h1Next.degree <= h2Next.degree  =>`<br>`h1Next.degree == h2Next.degree` |
| ROR | 130 | `h1Next.degree <= h2Next.degree  =>`<br>`h1Next.degree != h2Next.degree` |
| ROR | 146 | `h1Next != null  =>  h1Next == null` |
| COR | 50 | `nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree  =>`<br>`nextX.sibling != null ||`<br>`nextX.sibling.degree == x.degree` |
| COR | 50 | `nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree  =>` |

| | | |
|---|---|---|
| | | `nextX.sibling != null ^`<br>`nextX.sibling.degree == x.degree` |
| COR | 50 | `x.degree != nextX.degree || nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree =>`<br>`x.degree != nextX.degree && (nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree)` |
| COR | 50 | `x.degree != nextX.degree || nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree =>`<br>`x.degree != nextX.degree ^ (nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree)` |
| COR | 129 | `h1Next != null && h2Next != null =>`<br>`h1Next != null || h2Next != null` |
| COR | 129 | `h1Next != null && h2Next != null =>`<br>`h1Next != null ^ h2Next != null` |
| COI | 42 | `h.Nodes == null => !(h.Nodes == null)` |
| COI | 49 | `nextX != null => !(nextX != null)` |
| COI | 50 | `x.degree != nextX.degree =>`<br>`!(x.degree != nextX.degree)` |
| COI | 50 | `nextX.sibling != null => !(nextX.sibling != null)` |
| COI | 50 | `nextX.sibling.degree == x.degree =>`<br>`!(nextX.sibling.degree == x.degree)` |
| COI | 50 | `nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree =>`<br>`!(nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree)` |
| COI | 50 | `x.degree != nextX.degree || nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree =>`<br>`!(x.degree != nextX.degree || nextX.sibling != null &&`<br>`nextX.sibling.degree == x.degree)` |
| COI | 57 | `x.key < nextX.key => !(x.key < nextX.key)` |
| COI | 64 | `prevX == null => !(prevX == null)` |
| COI | 104 | `h1.Nodes == null => !(h1.Nodes == null)` |
| COI | 106 | `h2.Nodes == null => !(h2.Nodes == null)` |
| COI | 117 | `h1.Nodes.degree <= h2.Nodes.degree =>`<br>`!(h1.Nodes.degree <= h2.Nodes.degree)` |
| COI | 129 | `h1Next != null => !(h1Next != null)` |
| COI | 129 | `h2Next != null => !(h2Next != null)` |
| COI | 129 | `h1Next != null && h2Next != null =>`<br>`!(h1Next != null && h2Next != null)` |
| COI | 130 | `h1Next.degree <= h2Next.degree =>`<br>`!(h1Next.degree <= h2Next.degree)` |
| COI | 129 | `h1Next != null => !(h1Next != null)` |
| LOI | 38 | `this.size => -this.size` |
| LOI | 38 | `h2.size => -h2.size` |

| | | |
|---|---|---|
| LOI | 50 | `x.degree => -x.degree` |
| LOI | 50 | `nextX.degree => -nextX.degree` |
| LOI | 50 | `nextX.sibling.degree => -nextX.sibling.degree` |
| LOI | 50 | `x.degree => -x.degree` |
| LOI | 57 | `x.key => -x.key` |
| LOI | 57 | `nextX.key => -nextX.key` |
| LOI | 90 | `z.degree => -z.degree` |
| LOI | 117 | `h1.Nodes.degree => -h1.Nodes.degree` |
| LOI | 117 | `h2.Nodes.degree => -h2.Nodes.degree` |
| LOI | 130 | `h1Next.degree => -h1Next.degree` |
| LOI | 130 | `h2Next.degree => -h2Next.degree` |

**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 Apendix above ).

| Operator | Line Number | Mutation |
|---|---|---|
| AORS | 349 | `size-- => size++` |
| AORS | 352 | `size-- => size++` |
| AORS | 378 | `size-- => size++` |
| ROR | 333 | `t == null  =>  t != null` |
| ROR | 343 | `t.left != null  =>  t.left == null` |
| ROR | 343 | `t.right != null  =>  t.right == null` |
| ROR | 348 | `t.left != null  =>  t.left == null` |
| ROR | 369 | `t == null  =>  t != null` |
| ROR | 374 | `t.left != null  =>  t.left == null` |
| ROR | 390 | `t != null  =>  t == null` |
| ROR | 391 | `t.left != null  =>  t.left == null` |
| COR | 343 | `t.left != null && t.right != null  =>`<br>`t.left != null \|\| t.right != null` |
| COR | 343 | `t.left != null && t.right != null  =>`<br>`t.left != null ^ t.right != null` |

| | | |
|---|---|---|
| COI | 333 | `t == null  =>  !(t == null)` |
| COI | 337 | `param.intValue() < t.info  =>`<br>`!(param.intValue() < t.info)` |
| COI | 340 | `param.intValue() > t.info  =>`<br>`!(param.intValue() > t.info)` |
| COI | 343 | `t.left != null  =>  !(t.left != null)` |
| COI | 343 | `t.right != null  =>  !(t.right != null)` |
| COI | 343 | `t.left != null && t.right != null  =>`<br>`!(t.left != null && t.right != null)` |
| COI | 348 | `t.left != null  =>  !(t.left != null)` |
| COI | 369 | `t == null  =>  !(t == null)` |
| COI | 374 | `t.left != null  =>  !(t.left != null)` |
| COI | 390 | `t != null  =>  !(t != null)` |
| COI | 391 | `t.left != null  =>  !(t.left != null)` |