Dynamic Symbolic Data Structure Repair

Ishtiaque Hussain, Christoph Csallner
Software Engineering Research Center (SERC)
Computer Science and Engineering Department
University of Texas at Arlington
April 23, 2010.

Why Data Structure Repair is important

- All software use some sort of data structure underneath.
- Data structure corruption might result into software crash
- User intervention might not be an option for corruption repair, e.g., real time systems.

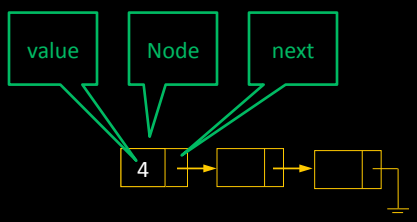
Goal:

- Automatically repair data structures to prevent software crash.
- Perform such repair actions effectively and in a time efficient manner. Cannot wait forever!

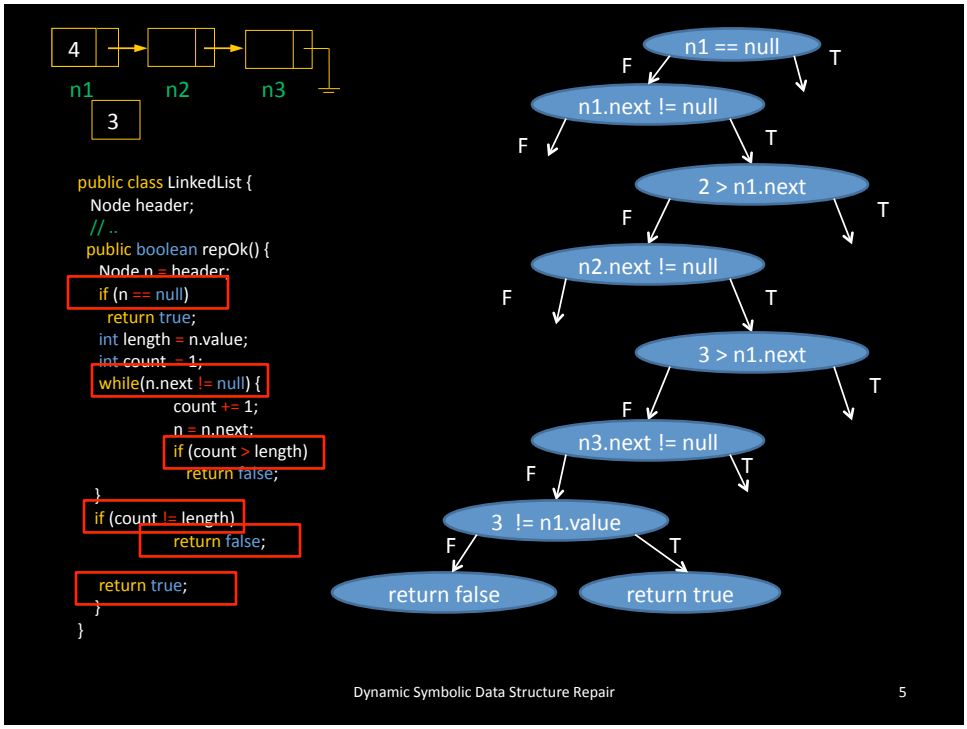
Example: Single linked list

```
public class LinkedList {
  Node header;
  // ..
  public boolean repOk() {
    Node n = header;
    if (n == null)
      return true;
    int length = n.value;
    int count  = 1;
    while(n.next != null) {
            count += 1;
            n = n.next;
            if (count > length)
              return false;
    }
    if (count != length)
            return false;

    return true;
    }
}
```

```
public class Node {
   int value;
   Node next;
   // ..
}
```



First node has a value that is equal to the number of nodes in the list.

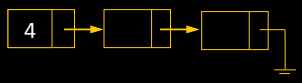


Our approach to the problem:

```
public class LinkedList {
  Node header;
  // ..
 public boolean repOk() {
   Node n = header;
   if (n == null)
     return true;
   int length = n.value;
   int count  = 1;
   while(n.next != null) {
            count += 1;
            n = n.next;
            if (count > length)
              return false;
   }
   if (count != length)
            return false;

   return true;
  }
}
```

```
public class Node {
  int value;
  Node next;
  // ..
}
```

n1 n2 n3

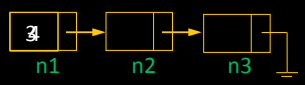


First node has a value that is equal to the number of nodes in the list.

Our approach to the problem:

```
public class LinkedList {
  Node header;
  // ..
 public boolean repOk() {
   Node n = header;
   if (n == null)
    return true;
   int length = n.value;
   int count  = 1;
   while(n.next != null) {
            count += 1;
            n = n.next;
            if (count > length)
              return false;
   }
   if (count != length)
            return false;

   return true;
   }
}
```

```
public class Node {
   int value;
   Node next;
   // ..
}
```



First node has a value that is equal to the number of nodes in the list.

Constraints:

```
n1 != null
n1.next != null
2 <= n1.value
n2.next != null
3 <= n1.value
n3.next == null
3!= n1.value    3 == n1.value
```

Solve

3 = n1.value

Existing state of the art tool: Juzi

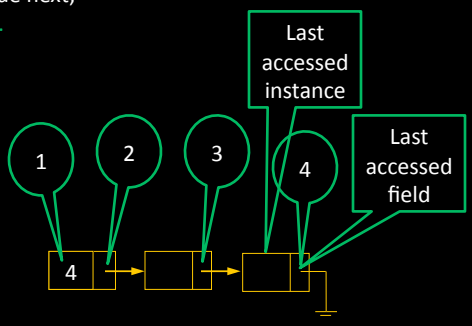
Few facts about Juzi:

- Developed by Bassem Elkarablieh and Sarfaraz Khurshid of University of Texas at Austin.
  - http://users.ece.utexas.edu/~elkarabl/Juzi/index.html
- Assumes last instance and field accessed before returning are corrupt.
- Uses constraint solving techniques for primitive fields.
- Uses exhaustive search technique for reference fields.
- Does not suggest new instances of classes.

Juzi's approach in solving our example problem:

```
public class LinkedList {
  Node header;
  // ..
 public boolean repOk() {
   Node n = header;
   if (n == null)
     return true;
   int length = n.value;
   int count  = 1;
   while(n.next != null) {
            count += 1;
            n = n.next;
            if (count > length)
              return false;
   }
   if (count != length)
            return false;

   return true;
   }
}
```

```
public class Node {
   int value;
   Node next;
   // ..
}
```

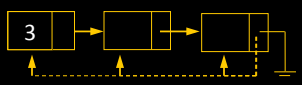


First node has a value that is equal to the number of nodes in the list.

Juzi's approach in solving our example problem:

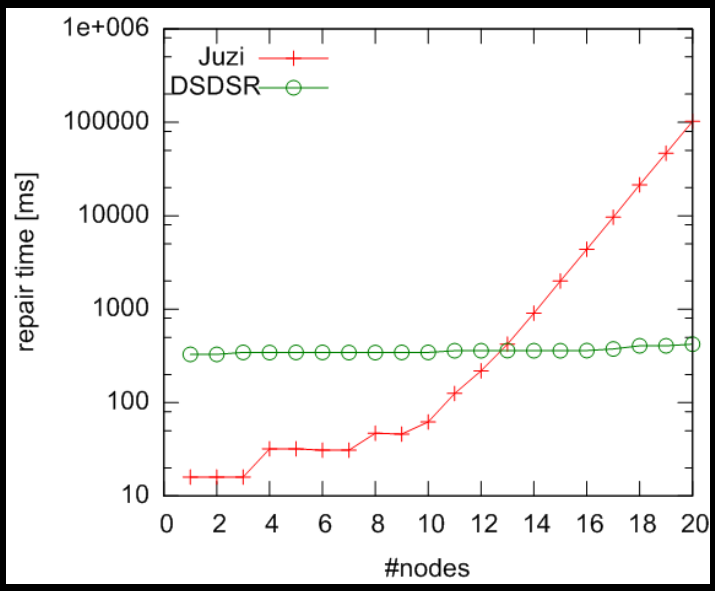
```
public class LinkedList {
  Node header;
  // ..
 public boolean repOk() {
   Node n = header;
   if (n == null)
     return true;
   int length = n.value;
   int count  = 1;
   while(n.next != null) {
            count += 1;
            n = n.next;
            if (count > length)
              return false;
   }
   if (count != length)
            return false;

   return true;
   }
}
```

```
public class Node {
   int value;
   Node next;
   // ..
}
```



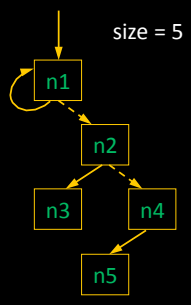
First node has a value that is equal to the number of nodes in the list.

Performance of Juzi and DSDSR for linkedlist



Another example: Binary tree

```
public boolean repOk() {
    boolean result = true;  // An empty tree == zero in size
   if (root == null){
      if (size  != 0)
          result = false;
       return result;
   }
   // ...
    while (!workList.isEmpty()) {
       Node current = workList.removeFirst();
       if (current.left != null) {     // no cycles along left
           if (!visited.add(current.left)) {
               result =  false;
           } else
               workList.add(current.left);
       }
       if (current.right != null) {  // no cycles along right
           if (!visited.add(current.right)) {
               result =  false;
           } else
               workList.add(current.right);
       }
    }
      if (visited.size() != size)  // size == #visited nodes
         result = false;

     return result;
   }
}
```



- Acyclic along left and right pointers.
- Cannot share nodes.
- #nodes reachable from root along left and right fields is stored in the size field.

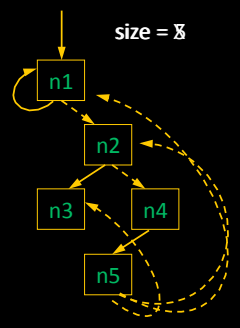
Another example: Binary tree

Observations:

- Defining correctness properties are hard.
- Writing a correctness condition that checks correctly, easy to read and meets repair specific style – is even harder.
- Juzi requires specific style in correctness condition, namely, last accessed field and instance should be the actual corrupt ones.

Binary tree: modified repOk - Juzi

```
public boolean repOk() {
   boolean result = true; // An empty tree == zero in size
   if (root == null){
      if (size  != 0)
         result = false;
      return result;
   }
   // ...
    while (!workList.isEmpty()) {
        Node current = workList.removeFirst();
        if (current.left != null) {     // no cycles along left
            if (!visited.add(current.left)) {
                result =  false;
            } else
                workList.add(current.left);
        }
        if (current.right != null) {  // no cycles along right
            if (!visited.add(current.right)) {
                result =  false;
            } else
                workList.add(current.right);
        }
    }
      if (visited.size() != size)  // size == #visited nodes
         result = false;

    return result;
   }
}
```



- Acyclic along left and right pointers.
- Cannot share nodes.
- #nodes reachable from root along left and right fields is stored in the size field.

Binary tree: modified repOk - DSDSR

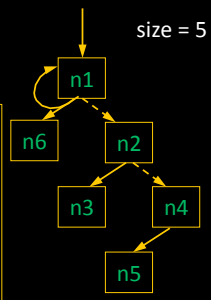
```
public boolean repOk() {
    boolean result = true;   // An empty tree == zero in size
   if (root == null){
     if (size  != 0)
        result = false;
      return result;
   }
   // ...
    while (!workList.isEmpty()) {
       Node current = workList.removeFirst();
       if (current.left != null) {     // no cycles along left
           if (!visited.add(current.left)) {
               result =  false;
           } else
               workList.add(current.left);
       }
       if (current.right != null) {  // no cycles along right
           if (!visited.add(current.right)) {
               result =  false;
           } else
               workList.add(current.right);
       }
    }
     if (visited.size() != size)  // size == #visited nodes
        result = false;

    return result;
   }
}
```

Constraints:

n1 != null
n1.left != null
~~n1.left == n1~~ n1.left != n1
n1.right != null
~~n1.right != n1~~
n2.left != null
n2.left != n1
n2.left != n2
...
5 == size

Solve

n1.left = new_Node



Binary tree: modified repOk

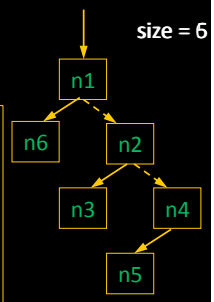
```
public boolean repOk() {
    boolean result = true;   // An empty tree == zero in size
   if (root == null){
     if (size  != 0)
        result = false;
      return result;
   }
   // ...
    while (!workList.isEmpty()) {
       Node current = workList.removeFirst();
       if (current.left != null) {     // no cycles along left
           if (!visited.add(current.left)) {
               result =  false;
           } else
               workList.add(current.left);
       }
       if (current.right != null) {  // no cycles along right
           if (!visited.add(current.right)) {
               result =  false;
           } else
               workList.add(current.right);
       }
    }
     if (visited.size() != size)  // size == #visited nodes
        result = false;

    return result;
   }
}
```

Constraints:

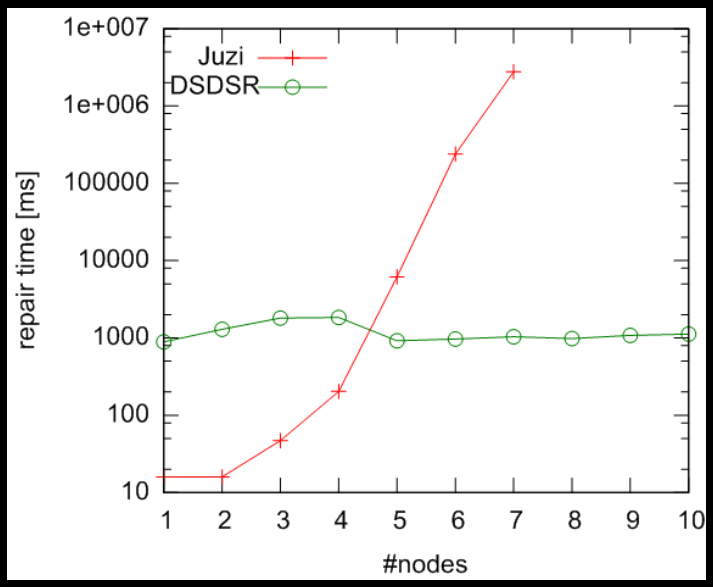
n1 != null
n1.left != null
n1.left != n1
n1.right != null
n1.right != n1
n1.right != n6
...
~~6 != size~~
6 == size

Solve

size = 6

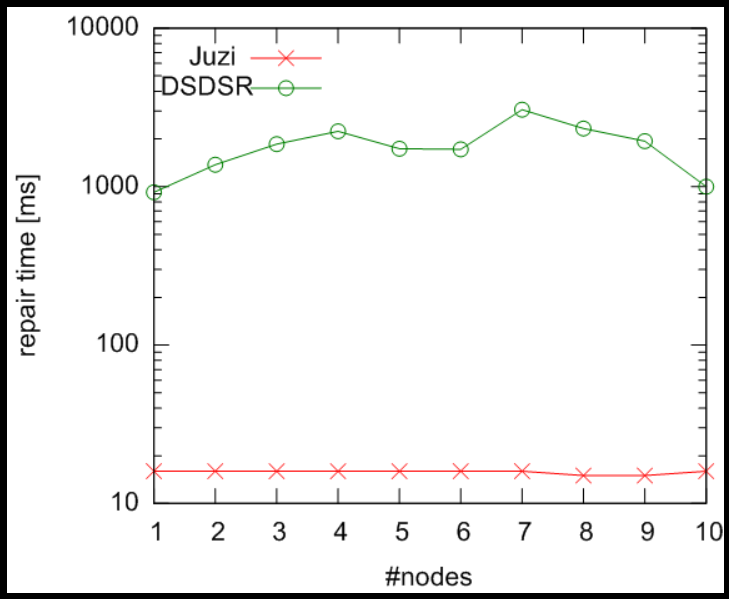


Results (1): Juzi Vs. DSDSR, return late



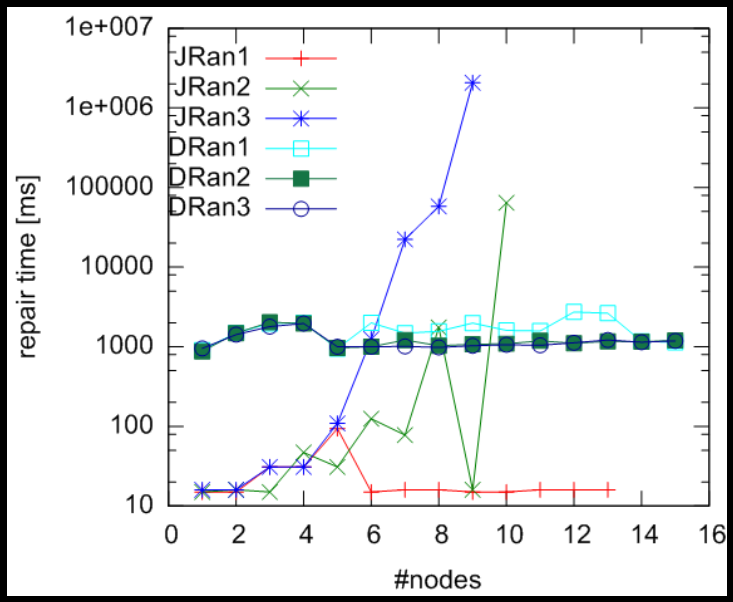
Dynamic Symbolic Data Structure Repair 17

Results (2): Juzi Vs. DSDSR, return immediate

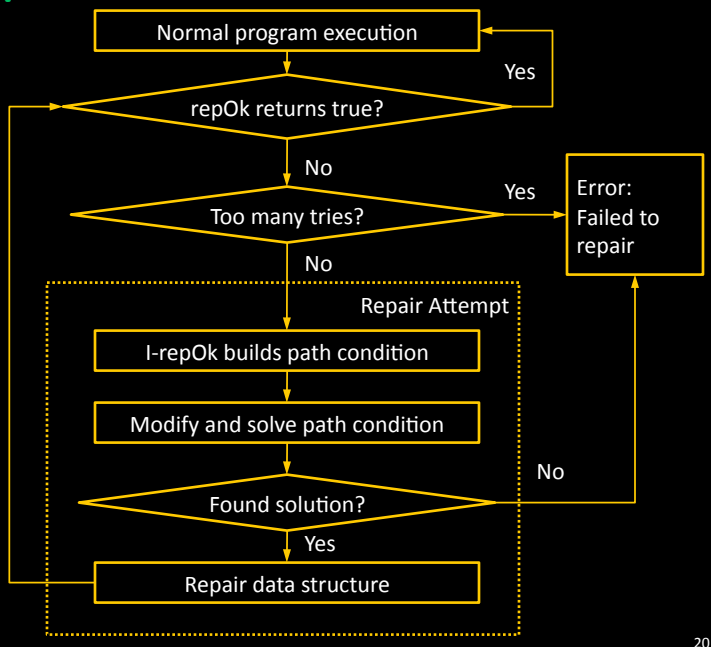


Dynamic Symbolic Data Structure Repair 18

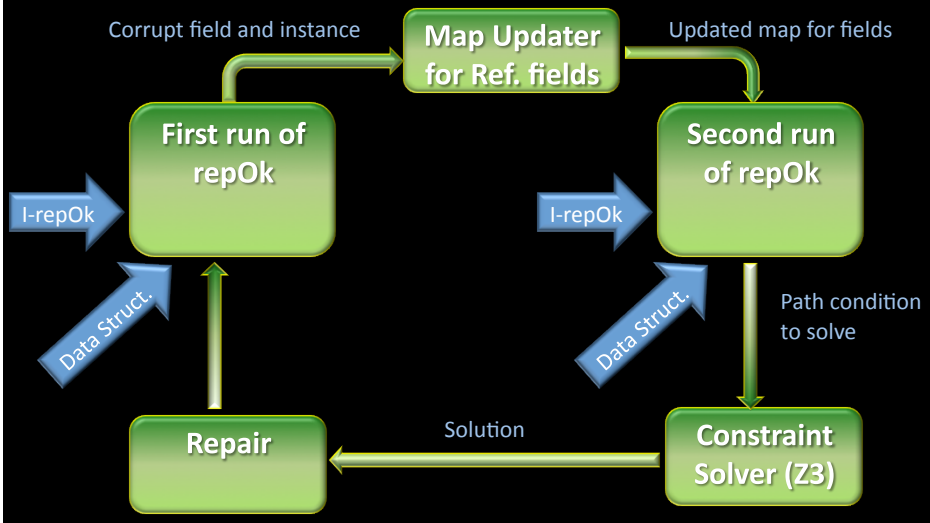
Results (3): Random



Algorithm:



Implementation:



Thank you!

We specially thank **Bassem Elkarablieh** and Sarfaraz Khurshid for helping us with Juzi.