

# Implementation of algorithms and data structures

## 3. seminar

Jirka Fink

<https://ktiml.mff.cuni.cz/~fink/>

Department of Theoretical Computer Science and Mathematical Logic  
Faculty of Mathematics and Physics  
Charles University in Prague

Winter semestr 2025/26

Last change 16. října 2025

Licence: Creative Commons BY-NC-SA 4.0

## Description (Wikipedia)

Fuzzing or fuzz testing is an automated software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

## Motivation

- For advanced algorithm, we are unable to create unit tests for all possible cases.
- Testing data consistency helps to fix a bug, but we have to be able to cause an error.
- Therefore, we provide our program random data, both correct and incorrect.

## Simple example

```
1 elements = list(range(start, stop, step))
2 random.shuffle(elements)
3 TEST(sorted(elements) == list(range(start, stop, step)))
```

## Question

What is the chance that a fuzz test finds an error?

## Infinite monkey theorem

- A monkey hitting keys at random on a typewriter keyboard for an infinite amount of time will almost surely type any given text, such as the complete works of William Shakespeare.
- The probability that monkeys filling the entire observable universe would type a single complete work is so tiny that the chance of it occurring during a period of time hundreds of thousands of orders of magnitude longer than the age of the universe is extremely low (but technically not zero).

## Collorary and practice

- A random generator finds a sequence of operations leading to an existing error after sufficient amount of time, assuming the generator can create such a sequence.
- The necessary amount of time is increasing with the length of input pretty fast.
- Most bugs in practice can be caused by many inputs which reduced the time for testing.
- Even fuzz testing cannot find all bugs, e.g. using formal verification a bug was found in the standard implementation of Timsort and the smallest input was of size 67 108 864.

## Example

```
1 t := TestedTree()
2 for i in 1:10 ... 0 do
3   | t.insert(random())
4   | t.test_data_consistency()
5 for i in 1:10 ... 0 do
6   | t.find(random())
7 for i in 1:10 ... 0 do
8   | t.delete(random())
9   | t.test_data_consistency()
```

## Why this is not sufficient?

- Data can be stored correctly even if the tree is losing elements.
- A bug may be caused by an insertion after some deletions.
- The correctness of query operations should be tested, but how do we determine the correct answer?

## Goal

Verify that expected elements are stored in our data structure and queries gives correct answers.

## Approach

- Choose an appropriate container from the standard library, e.g. array or a hash table
- Create for testing an auxiliary class containing an instance of
  - our tested data structure and
  - selected (referee) container.
- Ensure that both data structures contains the same elements:
  - New elements are inserted into both data structures
  - Deleted elements are removed from both data structures
- Query operations compare results from both data structures
  - The referee data structure does not need to be efficient.
  - It is sufficient that it provides correct answers.
- We also tests that both data structures contains the same elements using e.g.
  - simultaneous traverses of both data structures if elements are stored in a sorted order
  - traversing elements in one data structure and marking them in the other

## Example of a testing class

```
1 class tester {
2     my_buggy_tree examinee;
3     set referee;
4
5     void insert(element e) {
6         examinee.insert(e);
7         referee.insert(e);
8         compare_content();
9     }
10
11    void compare_content() {
12        iterator e = examinee.begin();
13        for(auto r : referee) { // Iterate elements in both trees
14            TEST(e != examinee.end());
15            TEST(e.key() == r.key()); // Comparing data in elements
16            e = e.next();
17        }
18        TEST(e == examinee.end());
19        TEST(examinee.size() == referee.size());
20    }
21
22    void find(key k) {
23        TEST(examinee.find(k) == referee.find(k));
24    }
25};
```

## Goal

- Create data for insertion, deletions, queries ...
- Choose order of these operations

## Simple sequential data generation

- Insert (e.g.) even numbers
- Try to find all number, including number out of range
- Try to delete all numbers
- Results of these operations are known, so we also verify our testing class

## Example

```
1 void test_sequence(int length) {
2     tester t;
3     for(int i = 1; i <= length; i++)
4         t.insert(2*i);
5     for(int i = 0; i <= 2*length+5; i++)
6         TEST(t.find(i) == (iseven(i) && 1 <= i <= 2*length));
7     for(int i = 0; i <= 2*length+5; i++)
8         t.remove(i);
9 }
```

## Methods

- Insert random elements
- Search for random elements which usually are not found
- Search for elements randomly selected from the referee which has to be found
- Delete both random elements and randomly selected from the referee

## Příklad

```
1 void test_random_data(int length) {
2     tester t;
3     for(int i = 0; i < length; i++)
4         t.insert(random()); // Duplicity keys may be inserted
5     for(key k : t.referee) {
6         TEST(t.find(k)); // Test a key should be stored
7         t.find(random()); // Test a key which is not most likely stored
8     }
9     while(!t.empty()) {
10        TEST(t.remove(random(t.referee))); // Random element from referee
11        t.remove(random());
12    }
13 }
```

## Motivation

Some errors may be caused by sequence alternating insertions and deletions

## Example of approach

```
1 void test_random_order(int length) {  
2     tester t;  
3     for(int i = 0; i < length; i++) {  
4         switch(random() % 7) {  
5             case 0: t.insert(random());  
6             case 1: t.insert(random(t.referee));  
7             case 2: t.find(random());  
8             case 3: t.find(random(t.referee));  
9             case 4: t.remove(random());  
10            case 5: t.remove(random(t.referee));  
11            case 6: ...  
12        }  
13    }  
14 }
```

## Questions

- How to obtain a correct solution for a random input?
- How to test correctness if there are multiple correct solutions?
  - E.g. there may be many shortest paths between given a pair of vertices.

## How to obtain a correct solution?

- Use available libraries if possible
  - E.g. sorting or red-black trees are implemented in standard libraries
- A given problem may have other slower but easier algorithms
  - E.g. for testing Strassen algorithm, use definition of matrix multiplication
- Faster algorithms are often obtained by improving slower ones
  - Keep the slower algorithm for testing
- Use theoretical knowledge of a problem or an algorithm

## Simple approach when a correct implementation is available

- Let  $A$  be an input sequence and  $B$  be the output of our program
- Let  $C$  be sorted sequence of  $A$  by a correct implementation
- Compare  $B$  and  $C$

What if we develop the first algorithm for some problem?

## Verifying correctness without sorting

- For every  $i$  test  $B[i] < B[i + 1]$
- Store all elements on  $A$  in a hash table  $T$
- For every  $i$  test that  $T$  contains  $B[i]$  and remove it
- Test that  $T$  is empty

If duplicates are all possible, use a counter for every element in the hash table.

## Expected output

- Connected: Returns a spanning tree represented by a vector  $P$  of predecessors to a root
- Disconnected: Returns a subset of vertices  $A$  such that both  $A$  and  $V \setminus A$  are non-empty and there is no edge between  $A$  and  $V \setminus A$

How to verify the correctness of this output?

## Optimization problem

The task is to find a solution satisfying given constraints and minimizing or maximizing given objective function.

- Shortest path
- Minimum spanning tree
- Maximum network flow

## Verifying correctness of solutions of a random input

- Test satisfaction of all constraints
- Test the optimality if possible

## Output of Dijkstra's algorithm

The output contains two arrays  $d$  and  $p$

- $d_u$  is the distance from a starting vertex  $s$  to  $u$
- $p_u$  is the predecessor of  $u$  on the shortest path from  $s$  to  $u$

## Verifying feasibility

For every vertex  $u$  different from  $s$ :

- The graph has an edge  $p_u u$
- The length of the edge  $p_u u$  is  $d_u - d_{p_u}$
- Going from  $u$  by predecessors, we must reach  $s$ , not a cycle
- $d_s = 0$

Can we test these conditions in linear time?

## Theorem

Let  $G$  be a directed graph,  $l_{uv}$  be length of edge  $uv$  and  $s$  be a starting vertex. Then  $d_u$  are lengths of the shortest path from  $s$  to all vertices  $u$  if and only if

- $d_s = 0$
- $d_v \leq l_{uv} + d_u$  holds for every edge  $uv$
- for every vertex  $v$  except  $s$  there exists an edge  $uv$  such that  $d_v = l_{uv} + d_u$ .

- Interval query
- Searching a substring in a string
- Connected components
- Minimum spanning tree
- Maximum flow
- Maximum matching
- Travelling salesman problem